EEE3095S CS Project

Tumi Mokoka
(MKKBOI005)

Matome Mbowene
(MBWMAT002)

Cassandra Wallace
(WLLCAS004)

Table of Contents

Introduction

In this project, we embark on an exciting journey to create a messaging system by harnessing the power of light signals. By utilising STM development boards, we aim to demonstrate the seamless exchange of information between two embedded systems. Our primary goal is to transmit analog data, specifically the readings from an on-board POT sampled through an ADC, from one board, known as the Transmitter or Sensor Node (SN), to another, the Receiver or Central Node (CN).

Design Choices

Key design choices include the use of push-buttons to trigger data sampling, a direct GPIO (general-purpose input/output) connection to transmit messages, and a 'light-of-things' (LoT) message protocol that encapsulates the data transmission process, including length-bit data frames and parity bit checks. While we carefully considered various methods for transmitting data via light, we have opted for the GPIO ‘Big-Bang’ method. This approach involves directly manipulating the GPIO pins to transmit binary data in serial format by modulating the state of a live voltage line. This method offers flexibility and control over the signal transmission process, making it suitable for our project’s requirements.

Project Activity Allocation

For this project, our team has carefully divided responsibilities to optimise efficiency and productivity. Each member is tasked with specific activities, ranging from hardware setup to software development to detailed documentation. This approach ensured seamless collaboration and the successful execution of our project.

Report Structure

This report outlines our journey, beginning with the project's inception and design, followed by the implementation details and performance evaluation. Ultimately, we will evaluate the success of our system and discuss its potential applications.

Requirements and LoT Message Protocol

This section outlines the essential system requirements and introduces the 'light-of-things' (LoT) message protocol for our project, which aims to establish a communication system using light signals. The project involves two crucial components: the Transmitter (Sensor Node) and the Receiver (Central Node), each with distinct functionalities.

System Description & Diagram

Figure 1 illustrates the high-level overview of the system's operation, demonstrating the interaction between the Transmitter and Receiver. It serves as a reference point for designing and implementing the system's key operations, offering a clear understanding of how the messaging system functions.

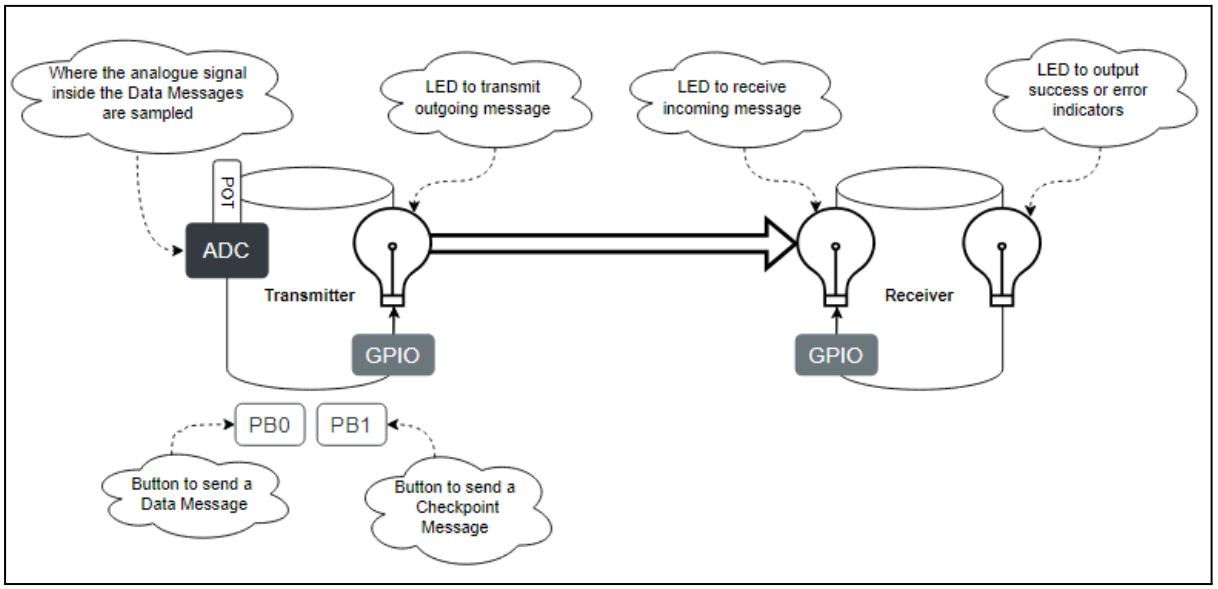

Figure 1: *High-level System Diagram.*

Transmitter (Sensor Node)

The Transmitter, equipped with an Analogue-to-Digital Converter (ADC), interfaces with an on-board potentiometer (POT). It awaits a push-button press (PA0) to signal the initiation of a Data Message transmission. This process involves the sampling of the POT's value, conversion of this value into a binary message, and transmitting the binary message as light pulses via an LED. This binary message adheres to the structure outlined in our LoT Message Protocol, detailed below, which includes start/stop bits and parity checks. Additionally, the

binary message is duplicated and sent to a GPIO pin for direct wiring to the Receiver or Central Node.

Furthermore, the Transmitter keeps track of the number of samples sent by incrementing a sample counter. This count is included in a Checkpoint Message that sends the number of samples sent to the Receiver. A press of PA1 initiates a Checkpoint Message transmission.

Receiver (Central Node)

The Receiver remains in a continuous listening mode, awaiting messages directly through a GPIO pin. Upon receiving these light pulses, the Receiver efficiently decodes them into a binary message. The resulting binary data is then displayed on the LCD. In the case of decoding errors, an error message is presented.

In addition to decoding incoming data, the Receiver keeps track of the samples received by incrementing a sample counter. To ensure data consistency, the Receiver compares the sample counter in the checkpoint message received from the Transmitter to the count of samples it has received. If these counts differ, it indicates a missing samples alert by rapidly flashing its LED for a duration of 2 seconds. If the checkpoint and received samples match, the Receiver indicates the ‘all-received’ alert by having its LED remain on for 2 seconds. This mechanism ensures the reliability of data transmission and reception.

LoT Message Protocol

For our communication system, we have defined two types of messages:

- **Data Message**

This type of message is used to send the value of the Transmitter’s POT, which is sampled by the ADC, to the Receiver. The precise format of the data message is as follows:

1. **Start-of-Text (SOT) [1-bit]:** The start bit, which is indicated by an LED being turned on for a duration of 1 second.
2. **Message Identifier [1-bit]:** A bit that allows the Receiver to identify the message type of the incoming transmission. The LED is turned off for 1 second to indicate that it is a Data Message.

3. **Data Packet [16-bit]:** The primary data transmission format consists of a sequence of the ADC value. Each bit is represented by an LED being turned on for 500 milliseconds, to signify a '1', and turned off for 500 milliseconds, to signify a '0'.

4. **Even Parity-Bit [1-bit]:** A low-cost form of error-detection to ensure the reliable transmission of the data packet. It is indicated by the LED being turned off or on for 1 second, depending on the total sum of '1' decoded in the data packet, plus the parity-bit itself.

5. **End-of-Text (EOT) [1-bit]:** The stop bit, indicated by the LED flashing at 100 millisecond intervals for a total duration of 1 second.

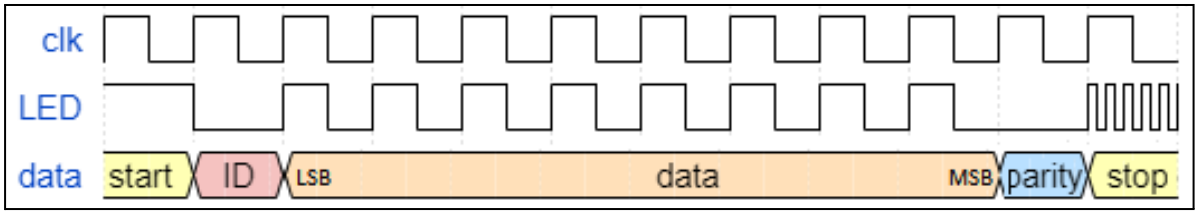

Figure 2: *Ideal Timing Diagram for a Data Message, where the Transmitter is attempting to send a data packet containing a sampled ADC. Note that each clock cycle has a duration of 1 second, and within the Data Payload, a half-cycle represents 1-bit.*

➢ Checkpoint Message

This is a special pre-defined binary message that indicates the presence of a Checkpoint Message. It consists of the count of samples that has been successfully sent by the Transmitter, allowing the Receiver to determine if any Data Messages were lost in transmission. More importantly, it ensures the completeness and order of received messages, making it a crucial feature for maintaining data integrity. The precise format of the Checkpoint Message is as follows:

1. **Start-of-Text (SOT) [1-bit]:** The start bit, indicated by an LED being turned on for a duration of 1 second.

2. **Message Identifier [1-bit]:** A bit to allow the Receiver to identify the message type of the incoming transmission. The LED is turned on for a duration of 1 second if it is a Checkpoint Message.

3. **Checkpoint Packet [16-bit]:** The primary data transmission format consists of a sequence of the value of the current samples transmitted counter by the Transmitter.. Each bit is represented by an LED being turned on for 500 milliseconds, to signify a '1', and turned off for 500 milliseconds, to signify a '0'.

4. **Even Parity-Bit [1-bit]:** A low-cost form of error-detection to ensure the reliable transmission of the checkpoint packet, and is indicated by the LED being turned off or on for 1 second, depending on the total sum of '1' decoded in the data packet, plus the parity-bit itself.

5. **End-of-Text (EOT) [1-bit]:** The stop bit, which is indicated by the LED flashing at 100 millisecond intervals for a total duration of 1 second.

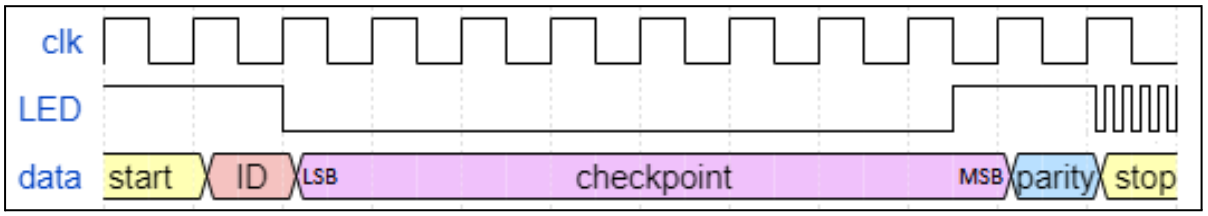

Figure 3: *Ideal Timing Diagram for a Checkpoint Message, where the Transmitter is attempting to send a Checkpoint Payload of 1, which means that one Data Packet has been transmitted successfully. Note that each clock cycle has a duration of 1 second, and within the Checkpoint Payload, a half-cycle represents 1-bit.*

A Brief Note about Message Length & Duration

For both our Data and Checkpoint Message, we have chosen a maximum and minimum size of 16-bits for both the Data and Checkpoint Payload. This decision is based on the fact that the Transmitter's ADC can produce a maximum value of 4095 (12-bit ADC, i.e. 2^{12}-1), which makes a 16-bit size efficient for data representation in comparison to using 32-bit. This constraint also results in the transmitted Data Message counter having a maximum value of 65536 (2^{16}), which is unlikely to ever reach such high values. Thus, the minimum amount of time for both a Data or Checkpoint Message to be transmitted is approximately 12 seconds, to allow for human observation of each pulse.

Major Departures & Additions from Original Description

In the context of the original project description, our primary departure from the original design involves our choice to use the **GPIO 'Bit-Bang'** method for transmitting data through light signals. Whilst we did explore the possibility of incorporating a Light-Dependent Resistor (LDR) for use on the Receiver side, due to project constraints, we ultimately opted for the more time-efficient direct GPIO connection between the two modules. This offers us greater precision and control over the light message transmission, enabling the customisation of encoding and decoding processes.

Additionally, in contrast to the original project description, we have introduced an essential feature in our communication system: the inclusion of **parity bits**. The decision to include parity bits stems from our dedication to ensuring the utmost data integrity and reliability in the face of potential errors. Parity checks are integral to our design for several reasons.

1. Firstly, parity bits serve as a robust means to validate the accuracy of each data packet. They provide a mechanism to detect single-bit errors, which can occur during the transmission process. This addition greatly enhances the system's resistance to data corruption, minimising the likelihood of incorrect readings and ensuring that the transmitted data remains intact and reliable.

2. Secondly, the checkpoint message system, while essential for identifying the loss of entire messages, may not address isolated errors, such as single-bit flips, that may occur within the actual data packet. Parity checks supplement this checkpoint mechanism by offering an additional layer of protection against data corruption, rather than just complete data loss.

In addition to the significant departures from the original project design, we have made a pivotal change regarding the Receiver's functionality. Instead of relying on a physical push-button to initiate the listening mode, **our Receiver is always in a listening state, continuously monitoring incoming messages.** This alteration aligns with modern technology trends, particularly in contexts like smart homes, where sensors, devices, and communication systems must be prepared to respond to dynamic changes and data inputs in real-time. By maintaining constant vigilance for incoming signals, our Receiver can promptly react to sensor data and other messages, ensuring a seamless and efficient operation that

meets the demands of contemporary applications. This continuous listening approach optimises the system's responsiveness and adaptability, enhancing its suitability for dynamic and evolving environments.

With this system description and LoT message protocol in place, we proceeded to design and implement a messaging system that not only aligns with the project’s goals but also enhances data reliability and accuracy, as further detailed in the subsequent sections of this report.

Specification and Design

This section provides a comprehensive overview of the specification and design aspects of our messaging system, shedding light on the key operations executed by the Transmitter (Sensor Node) and the Receiver (Central Node). The design ensures the efficient exchange of analogue data via light signals, leveraging the GPIO 'Bit-Bang' approach, while also prioritising data integrity through a structured methodology.

Main System Operation Diagrams

Firstly, for the Transmitter (Sensor Node):

1. Initiating Data Transmission

The Transmitter incorporates interrupts associated with the PA0 and PA1 push-button presses. When the PA0 push-button is pressed, it triggers the transmission of a Data Message, as detailed in **(2)**. If the PA1 push-button is pressed, the Transmitter initiates the transmission of a Checkpoint Message, as described in **(4)**.

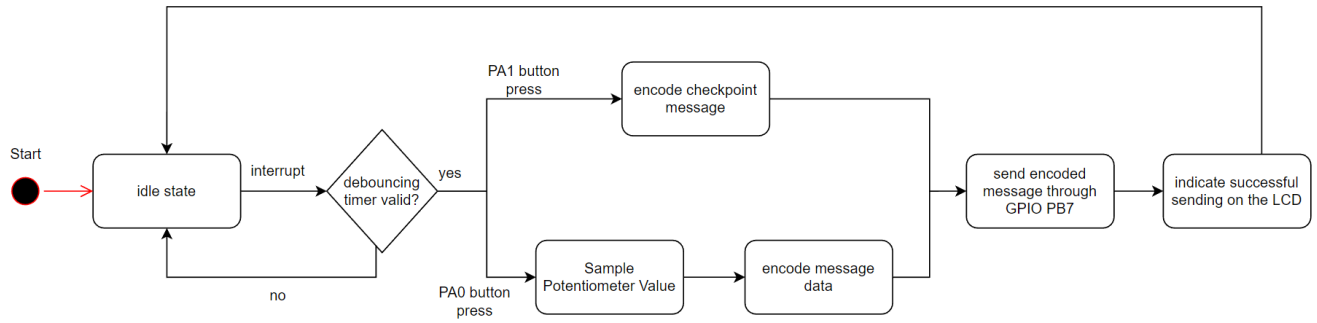

Figure 4: *Flowchart of interrupt coordination with debouncing.*

2. Encoding the Data Message

Upon triggering of a Data Message transmission, the Transmitter begins by polling the Analogue-to-Digital Converter (ADC) to acquire the current POT value. This sampled POT as the data payload within the data packet. The process includes data packet creation, comprising the start-bit, Data Message ID and the stop-bit. Additionally, it also consists of the Even Parity-Bit calculation. Once completed, the Transmitter has a fully-formed Data Message ready for transmission.

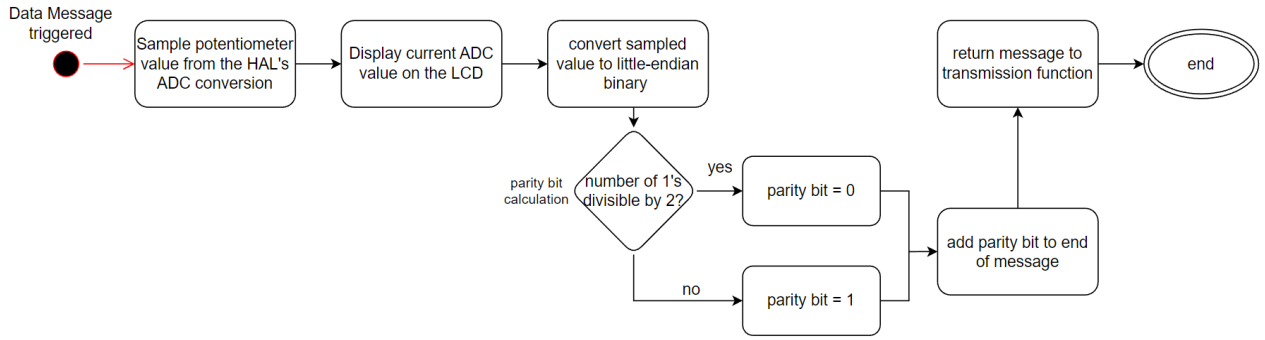

Figure 5: *Flowchart of Data Message creation and encoding.*

3. Encoding the Checkpoint Message

When a Checkpoint Message transmission is initiated, the Transmitter retrieves the current count of transmitted Data Messages and encapsulates it as per the Checkpoint Message Protocol, incorporating appropriate start/stop bits and parity-bit calculation.

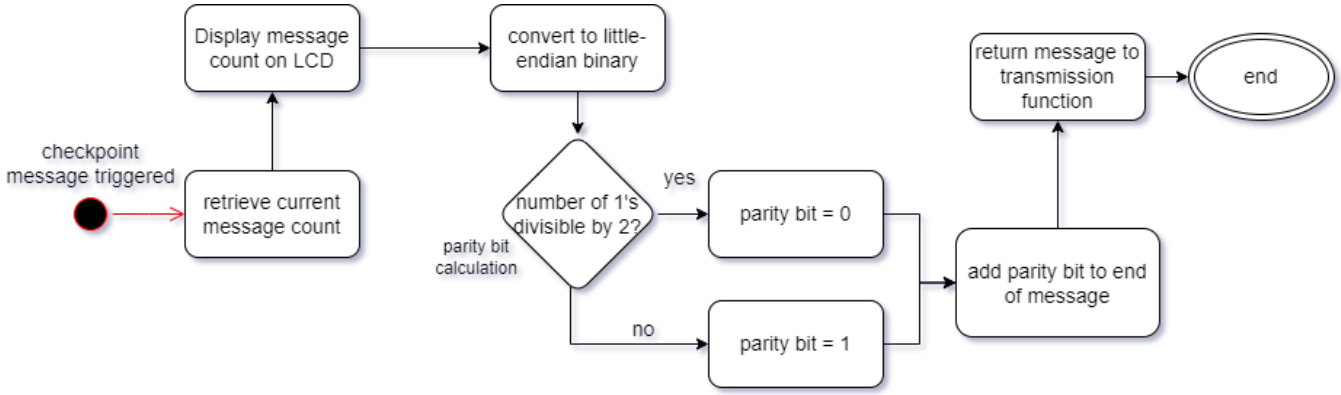

Figure 6: *Flowchart of Checkpoint Message creation and encoding.*

4. Transmitting the Data/Checkpoint Message

Upon the finalisation of either **(2)** or **(3)**, the Transmitter has a complete encoded message ready to transmit to the Receiver. The Transmitter commences the transmission process by modulating the LED state, sending the binary data message. This LED is connected to a GPIO pin that the Receiver monitors. In the event of a successful Data Message transmission, the Transmitter increments its transmitted Data Message counter.

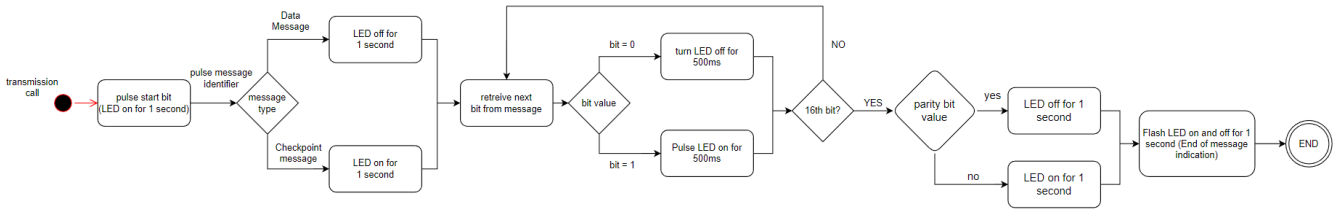

Figure 7: *Flowchart depicting the transmission of either a Data or Checkpoint Message to the GPIO output line.*

Lastly, for the Receiver (Central Node):

1. **Listening For & Preliminary Decoding of a Received Message**

The Receiver operates in a continuous listening mode, awaiting incoming messages via a GPIO pin. Upon message reception, the Receiver conducts efficient decoding, converting the received signals into binary messages. After removing the start-bit, the Receiver analyses the Message ID bit to determine whether the message is a Data Message ('0') or a Checkpoint Message ('1'), guiding the subsequent decoding process.

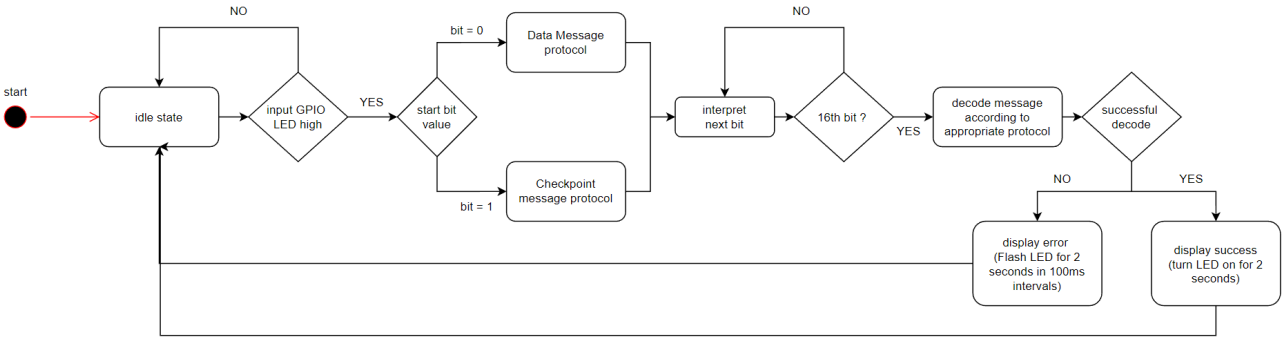

Figure 8: *Flowchart depicting Receiver logic.*

2. **Data Message Decoding**

When the Receiver identifies a received message as a Data Message, it extracts the sampled POT value. To ensure data integrity, the Receiver validates the parity-bit. In the case of successful validation, it proceeds to display or process the received ADC value. A success indication is triggered, involving turning on an LED for 2 seconds. In the event of unsuccessful validation, an error indication is initiated, causing the LED to flash at 100-millisecond intervals for a total duration of 2 seconds.

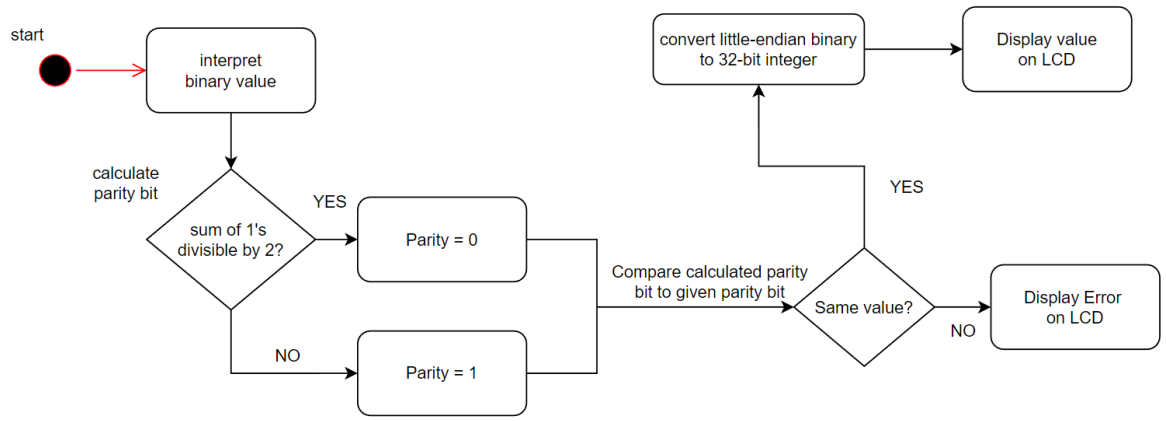

Figure 9: *Flowchart depicting the decoding of a Data Message by the Receiver.*

3. Checkpoint Message Decoding

Upon recognising the received message as a Checkpoint Message, the Receiver compares the count of received samples to the count indicated in the message. If a match is detected, the Receiver concludes that no Data Messages were lost during transmission and activates a success indication by turning on an LED for 2 seconds. In cases of non-matching counts, an error indication is initiated, resulting in the LED flashing at 100-millisecond intervals for a total duration of 2 seconds.

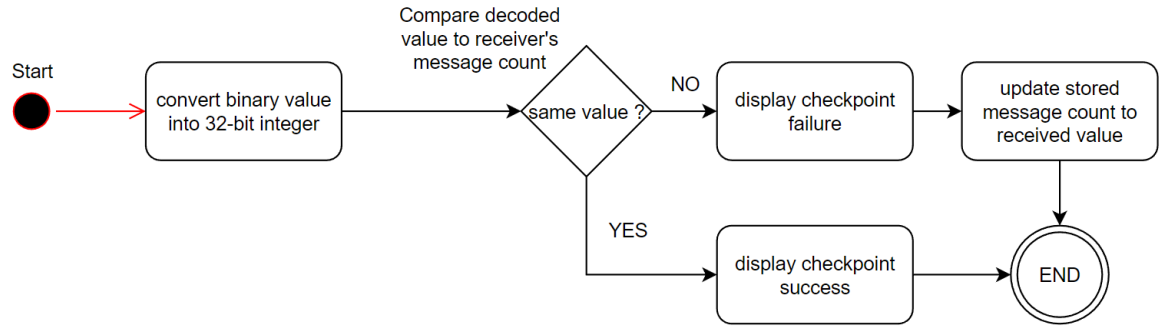

Figure 10: *Flowchart depicting decoding of a Checkpoint Message by the Receiver.*

Circuit Diagram

The following circuit diagram provides a visual representation of the physical connections and components involved in the system, aiding in a deeper understanding of the hardware aspects.

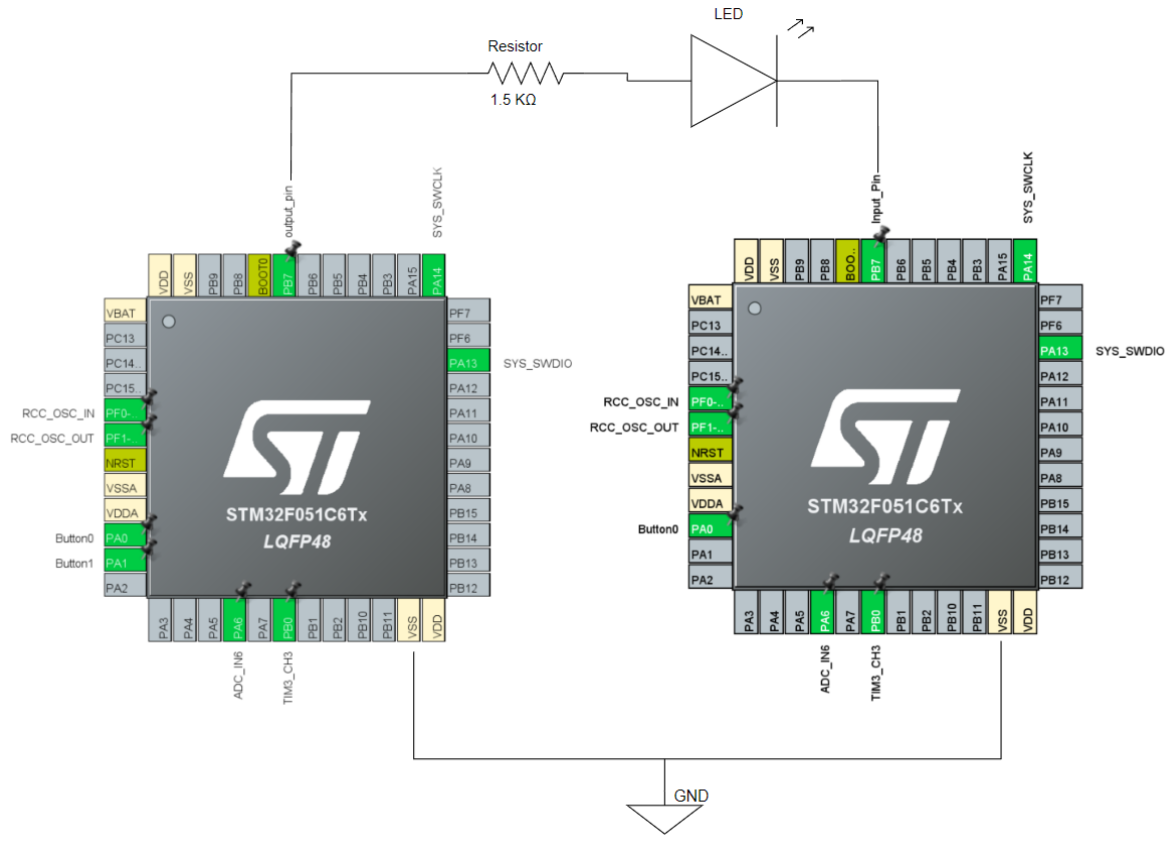

Figure 11: *Circuit diagram depicting the connections between our two STM development boards and implementation of the direct GPIO wire.*

The simple circuit diagram provides a direct connection between both of the PB7 GPIO pins on either STM32 board. The transmitter's board has its PB7 pin configured to output mode and the receiver's board has its pin configured to input mode allowing them to communicate using the direct voltage line. We decided to add a limiting resistor and LED for display purposes, the LED provides visual feedback to indicate the status of the communication line between the two boards. This enhances the user experience by providing a visual and intuitive way of understanding what is happening in the system while allowing us to debug and verify the communication line easily.

The transmitter's two buttons SW1 (PB0) and SW2 (PB1) are configured to be input buttons used to transmit messages. SW1 activates the protocol to send the sample potentiometer value and SW2 sends the checkpoint value using a different message format. The GPIO pin PA6 on the transmitter contains the Analogue to digital conversion logic used to sample the potentiometer value, this value ranges from 0 to 4095.

While this section focused on the core system operations, the subsequent sections will delve into finer details, including code modules and classes, ensuring a comprehensive and well-structured implementation of the messaging system.

Implementation

This section provides insights into the practical implementations of our messaging system, offering explanations and insights into key elements of the system, particularly focusing on aspects that might not be readily apparent from the design diagrams from the previous section. The practical implementation of our messaging system is organised into modular functions, each dedicated to specific operations.

For both the Transmitter (Sensor Node) and Receiver (Central Node):

Even Parity-Bit Calculation: The calculation of the Even Parity-Bit within the messages is a critical step to ensure data integrity. This involves counting the number of '1' bits within the desired payload and appending an additional parity-bit to make the total count even. For an even sum, the parity-bit is set to '0'. For an odd sum, the parity-bit is set to '1'. The parity calculation is executed by the following function:

```
/**
* @brief Calculate the parity bit for a 16-bit data value.
* This function calculates the parity bit for a 16-bit data value by
* performing an XOR operation on all the individual bits. Parity is used
* to check for data integrity and to ensure that the number of set bits in
* the data is even.
*
* @param data The 16-bit data for which to calculate the parity.
* @return The calculated parity bit (0 or 1) representing data integrity.
*/
uint8_t calculateParity(uint16_t data) {
        uint8_t parity = 0;
        for (int i = 0; i < 16; i++) {
                // XOR each bit to calculate the parity
                parity ^= (data >> i) & 0x01;
        }
        return parity;
}
```

`calculateParity` is used by both the Transmitter and Receiver to determine the parity-bit for a 16-bit data value. This function iterates through each bit of the 16-bit data using a for loop and performs an XOR operation on each bit's value, which is

obtained by right-shifting the data by `i` positions and applying a bitwise AND with 0x01 to isolate the LSB.

The result of the XOR operations accumulates in the parity variable, which, when the loop finishes, represents the calculated parity bit (either 0 or 1). It's a simple yet effective way to check data integrity by verifying that the number of set bits in the data is consistent with the expected parity. This function can be used to append a parity bit to the message and later to check if the received data's parity bit matches the recalculated one to detect errors.

For the Transmitter (Sensor Node):

Polling the ADC: To perform this operation, we implement a dedicated function that interfaces with the STM32 microcontroller's ADC peripheral, to obtain the real-time value from the connected potentiometer on the STM Board. The ADC values are then processed to convert into the desired binary message format, effectively encapsulating the sampled data, which will be included in a Data Message to be sent to the Receiver for further processing.

```
/**
* @brief Polls the ADC to get the current ADC value.
* This function initiates an ADC conversion and blocks until the conversion is
* completed, obtaining the current ADC value. If the ADC conversion is successful,
* it returns the ADC value; otherwise, it returns an error value (0xFFFFFFFF).
*
* @return The current ADC value if the conversion is successful,
*                              or 0xFFFFFFFF in case of an error.
*/
uint32_t pollADC(void){
        // Start ADC Conversion
        HAL_ADC_Start(&hadc);
        uint32_t val = 0;

        // Wait for ADC to finish
        // HAL_MAX_DELAY ensures enough blocking until finish
        if (HAL_ADC_PollForConversion(&hadc, HAL_MAX_DELAY) == HAL_OK) {
                // Successful ADC Conversion

                // Read the ADC value
                val = HAL_ADC_GetValue(&hadc);
```

```
                // Stop ADC Conversion
                HAL_ADC_Stop(&hadc);
        // If something went wrong in ADC Conversion
        } else {
                val = 0xFFFFFFFF;
        }
        return val;
}
```

`pollADC` is used by the Transmitter to poll the ADC (Analog-to-Digital Converter) for the current ADC value. When called, the function starts an ADC conversion using `HAL_ADC_Start` and then waits for the conversion to complete, blocking until it's finished. If the conversion is successful, the function retrieves the ADC value using `HAL_ADC_GetValue`, stops the ADC conversion using `HAL_ADC_Stop`, and returns the obtained ADC value. If there is an issue during the ADC conversion (for example, a timeout or other errors), the function returns an error value of 0xFFFFFFFF.

Transmission of Data/Checkpoint Message: The actual transmission of a Data or Checkpoint Message to the LED is a fundamental component of our system. It involves 'bit-banging' the GPIO pin connected to the LED. Our code establishes a mechanism to modulate the GPIO pin's state in alignment with the binary data, effectively encoding the message, which is the sampled ADC value that was previously polled.

```
/**
* @brief Transmit data over a communication channel, including control bits.
* This function is responsible for transmitting data, along with control bits,
* over a communication channel represented by an LED. The data can be a Data Message
* or a Checkpoint Message, and control bits are represented by LED states.
*
* @param data The 16-bit data to be transmitted.
* @param val A control variable to determine the message type:
* - If val is 1, the message is a Checkpoint Message (LED on).
* - If val is 0, the message is a Data Message (LED off).
*/
void transmitMessage(uint16_t data, uint8_t val) {
        // Start bit (SOT) - LED on for 1 second
```

```
        HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, GPIO_PIN_SET);
        delay(200000);
        // Message Identifier
        if (val == 1) {
                // Checkpoint Message - LED on for 1 second
                HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, GPIO_PIN_SET);
        } else if (val == 0) {
                // Data Message - LED off for 1 second
                HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, GPIO_PIN_RESET);
        }
        delay(200000);

        // Send the data in Little Endian binary format
        for (int i = 0; i < 16; i++) {
                // Extract each bit from the data
                uint8_t bit = (data >> i) & 0x01;

                // LED on for 500 ms for '1', LED off for 500 ms for '0'
                HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, (bit == 1) ?
                GPIO_PIN_SET : GPIO_PIN_RESET);
                delay(100000);
        }

        // Calculate the parity bit
        uint8_t parity = calculateParity(data);

        // LED on or off for 1 second based on parity bit
        HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, (parity == 1) ?
        GPIO_PIN_SET : GPIO_PIN_RESET);
        delay(200000);

        // Stop (EOT) - Toggling in 100ms intervals for 1 second
        uint8_t flag = 0;
        for (int i = 0; i < 10; i++) {
                HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, (flag == 1) ?
                GPIO_PIN_SET : GPIO_PIN_RESET);
                flag = !flag;
                delay(20000);
        }

        // Ensure Back to Idle State (OFF)
        HAL_GPIO_WritePin(LED7_GPIO_Port, LED7_Pin, GPIO_PIN_RESET);
}
```

`transmitMessage` is used by the Transmitter to transmit Data or Checkpoint Messages through an LED. The function accepts two parameters: `data`, which is the 16-bit payload to be transmitted, and `val`, a control variable to determine the message type. If `val` is set to 1, indicating a Checkpoint Message, the LED remains on for 1 second; if `val` is 0, indicating a Data Message, the LED turns off for 1 second, following our LoT (Light-of-Things) message protocol.

The binary data is sent using a process known as 'bit-banging' – each bit of the data is extracted, and the LED is turned on for 500 ms if it's a '1' or off for 500 ms if it's a '0'.

After sending the data, the code calculates the parity bit and transmits it by toggling the LED state for 1 second.

Finally, it sends the stop bit (EOT) by toggling the LED in 100 ms intervals for 1 second to indicate the end of the message. The code ensures that the LED returns to an idle state (off) after message transmission. This function handles the core process of encoding and transmitting Data or Checkpoint Messages in accordance with the specified message protocol.

Triggering the Transmission of a Data Message: To send the currently polled ADC value (which is visible on the LCD) to the Receiver as a Data Message, one just needs to press the PA0 push-button.

```
/**
* @brief Interrupt handler for EXTI0_1 (PA0 button press).
* @note This function is called every time PA0 is pressed.
*/
void EXTI0_1_IRQHandler(void)
{
	// Getting the current time
	curr_millis = HAL_GetTick();
	// Debouncing to prevent false triggers
	if ((curr_millis - prev_millis) >= 500) {

		//*************************************************
		// When user presses PA0, this function initiates the
		// transmission process:
		// 1) Polls the current ADC value
		// 2) Writes the current ADC value to LCD
```

```
                // 3) Converts to binary and pulses LED0
                //*****************************************************

                // Read ADC Value from POT1
                adc_val = pollADC();

                // Display the current ADC value on the LCD
                char lcd_message[20];
                sprintf(lcd_message, "ADC Value: %d", adc_val);
                lcd_command(CLEAR);
                lcd_putstring(lcd_message);
                lcd_command(LINE_TWO);
                lcd_putstring("Sending...");

                // Transmit the Data Message
                transmitMessage(adc_val, 0);

                // Indicate successful sending
                lcd_command(LINE_TWO);
                lcd_putstring("Sent to CN!");
                transmitted_counter++;

                // Add a delay for visual feedback
                delay(400000);

                // Update previous time for debouncing
                prev_millis = curr_millis;
        }

        HAL_GPIO_EXTI_IRQHandler(Button0_Pin); // Clear interrupt flags
}
```

Above defines an interrupt handler function, `EXTI0_1_IRQHandler`, which is executed whenever the PA0 button is pressed. To prevent false triggers due to mechanical button bouncing, it implements a debouncing mechanism by checking the time elapsed between button presses (the `curr_millis` and `prev_millis` variables).

When the debouncing period of 500 milliseconds (0.5 seconds) has passed since the last button press, the function initiates the transmission process of a Data Message. The steps involved include: polling the current value from an Analog-to-Digital Converter (ADC), displaying this value on an LCD screen, converting the value into

binary form and transmitting it as a Data Message through the `transmitMessage` function. An acknowledgment message is displayed on the LCD to indicate a successful transmission, and the `transmitted_counter` is incremented to keep track of the number of messages sent. The code concludes by clearing the interrupt flags and updating the previous time (`prev_millis`) for debouncing. This function is responsible for triggering the transmission of Data Messages when the PA0 button is pressed, ensuring data is sent accurately and without false triggers.

Triggering the Transmission of a Checkpoint Message: This method continuously monitors the state of the PA1 button using a polling approach. We adopted this strategy because Checkpoint Messages are less frequent than data messages, making it a more efficient choice compared to employing an additional interrupt.

```
/**
* @brief The application entry point.
* @retval int
*/
int main(void) {
	…
	/* Infinite loop */
	while (1)
	{
		// Listening for Checkpoint Message Transmission
		if (!(GPIOA->IDR & GPIO_IDR_1)) {
			HAL_NVIC_DisableIRQ(EXTI0_1_IRQn);
			triggerCheckpoint();
			HAL_NVIC_EnableIRQ(EXTI0_1_IRQn);
		}
		…
	}
}
```

The provided code above is a part of the main function and is designed to listen for the transmission of a Checkpoint Message without using an interrupt, allowing for more efficient handling. It continually checks the state of the PA1 button, and when it detects a button press (i.e., when the input at PA1 is LOW), it triggers the triggerCheckpoint function. To ensure that a Checkpoint Message transmission is not interrupted by a trigger of a Data Message transmission (PB0), we temporarily disable the PB1 interrupt using `HAL_NVIC_DisableIRQ(EXTI0_1_IRQn)` before

initiating the Checkpoint transmission and re-enable it using `HAL_NVIC_EnableIRQ(EXTI0_1_IRQn)` once the transmission is completed. This enhancement guarantees that the Checkpoint Message transmission remains uninterrupted, addressing potential issues that might arise if another interrupt occurs during this critical process.

```
/**
* @brief Trigger a checkpoint transmission.
* Function responsible for transmitting a Checkpoint Message to a destination.
* It includes the counter value in the message and updates the LCD display.
*
* The function performs the following steps:
* 1. Construct a message containing the current counter value.
* 2. Display the message on the first line of the LCD.
* 3. Display a "Sending..." message on the second line of the LCD.
* 4. Transmit the checkpoint message to the destination.
* 5. Display a "Sent to CN!" message on the second line of the LCD.
* 6. Add a delay for visual feedback.
*/
void triggerCheckpoint(void) {

        // Construct a message with the current counter value
        char checkpoint[20];
        sprintf(checkpoint, "Counter: %d", transmitted_counter);

        // Display the Checkpoint Message on the LCD
        lcd_command(CLEAR);
        lcd_putstring(checkpoint);
        lcd_command(LINE_TWO);
        lcd_putstring("Sending...");

        // Transmit the Checkpoint Message
        transmitMessage(transmitted_counter, 1);

        // Indicate successful sending
        lcd_command(LINE_TWO);
        lcd_putstring("Sent to CN!");

        // Add a delay for visual feedback
        delay(400000);
}
```

This function is responsible for transmitting a Checkpoint Message to a destination, which includes the current counter value in the message. The key steps performed by `triggerCheckpoint` include constructing a message with the current counter value, displaying the Checkpoint Message on the LCD, transmitting the Checkpoint Message, indicating successful transmission on the LCD, and adding a delay for visual feedback. This approach efficiently handles Checkpoint Message transmissions, ensuring that they are sent when needed, without using a dedicated interrupt, making it suitable for scenarios where Checkpoint Messages are less frequent than Data Messages.

For the Receiver (Central Node):

Listening for Incoming Message: One of the fundamental operations in the Receiver is the continuous listening to the GPIO pin for incoming messages.

```
/**
* @brief The application entry point.
* @retval int
*/
int main(void) {
	…
	lcd_putstring("Listening...");

	/* Infinite loop */
	while (1)
	{
		// Listen to the GPIO PB7 (GPIO_IDR_7) for incoming data,
		// i.e. the SOT bit (LED HIGH)
		if (LL_GPIO_IsInputPinSet(GPIOB, LL_GPIO_PIN_7)) {
			lcd_command(CLEAR);
			lcd_putstring("Found Message!");
			lcd_command(LINE_TWO);
			lcd_putstring("Decoding Now...");

			receiveMessage();

			lcd_command(CLEAR);
			lcd_putstring("Listening…");
		}
	}
}
```

It operates as an infinite loop that monitors the state of a GPIO pin 7 for incoming data, specifically focusing on the Start of Transmission (SOT) bit, indicated by the LED being HIGH. When an SOT bit is detected, the LCD display is updated to indicate that a message has been found and is currently being decoded.

The `receiveMessage` function is called to handle the message's reception, including parsing it, identifying whether it's a Data Message or Checkpoint Message, and subsequently decoding it according to its type. The Receiver ensures that the appropriate actions are taken based on the type of message received. Once the message is successfully processed, the LCD display is cleared and returns to the listening state to await the next incoming message. This process allows the Receiver to continuously monitor and decode incoming messages in real-time, making it an essential part of the communication system.

Receiving a Data/Checkpoint Message: Once an incoming message has been sensed, as detailed above, the Receiver must receive the message before beginning the decoding & processing of.

```
/**
* @brief Receive and process an incoming message.
* This function is responsible for receiving and processing incoming messages,
* distinguishing between Data and Checkpoint Messages, and ensuring the integrity
* of the received data.
*/
void receiveMessage(void) {
        // If this function is invoked, it means that the SOT was found
        delay(200000);

        // Message Identifier
        // If pin is low next -> Data Message
        // If pin is high next -> Checkpoint Message
        uint8_t message_type = LL_GPIO_IsInputPinSet(GPIOB, LL_GPIO_PIN_7);
        delay(200000);

        uint16_t received_data = 0;

        // Receive the data bits in Little Endian format (LSB first)
        for (int i = 0; i < 16; i++) {
                if (LL_GPIO_IsInputPinSet(GPIOB, LL_GPIO_PIN_7)) {
                        received_data |= (1 << i); // Set the bit to 1 if received
```

```
                }

                // Wait for 500ms for each bit
                delay(100000);
        }

        // At this point, received_data should hold the Data/Checkpoint Payload
        // Receive the parity bit
        uint8_t received_parity = LL_GPIO_IsInputPinSet(GPIOB, LL_GPIO_PIN_7);
        delay(200000);

        uint32_t EOT_high = 0;
        uint32_t EOT_low = 0;
        // Receive the stop bit (EOT)
        for (int i = 0; i < 10; i++)
        {
                if (LL_GPIO_IsInputPinSet(GPIOB, LL_GPIO_PIN_7))
                {
                        EOT_high ++;
                }
                else
                {
                        EOT_low++;
                }
                delay(20000);
        }
        if (EOT_high == 5 && EOT_low == 5)
        {
                // Message Successfully Received
                decodeMessage(message_type, received_data, received_parity);
        }
        else
        {
                lcd_command(CLEAR);
                lcd_putstring("incomplete -");
                lcd_command(LINE_TWO);
                lcd_putstring("message !!!");
                displayError();
        }
}
```

When `receiveMessage` is invoked, it implies that the Start of Transmission (SOT) bit has been detected. The code first introduces a delay to ensure proper synchronisation, and then identifies the type of the incoming message by checking the next bit: the Message Identifier. If the pin is low, it indicates a Data Message; if it's high, it indicates a Checkpoint Message.

After distinguishing the message type, the code proceeds to receive and decode the 16 bits of message data in Little Endian format (LSB first), continuously monitoring the state of the GPIO pin to set or clear bits in the `received_data` variable. A delay of 500ms is applied for each received bit.

Once the data payload is fully received, it collects the parity bit, which will then be validated in the decoding process.

After receiving the data payload and parity bit, the code introduces a mechanism to check for the End of Transmission (EOT) sequence. If there are precisely five occurrences of the high state and five occurrences of the low state, it signifies the successful reception of the EOT sequence, as per our message protocol, ensuring that the message transmission is complete and has not been prematurely interrupted. If the EOT sequence is successfully detected, the code proceeds to invoke the `decodeMessage` for further processing. If the EOT sequence is not found as expected, indicating an incomplete message, it displays an error message and calls `displayError` to handle this scenario, providing robust error detection in the Receiver's message reception process.

This function plays a crucial role in ensuring that the incoming message is processed correctly, maintaining data integrity and responding accordingly to Data or Checkpoint Messages. It represents the core of the message reception mechanism within the Receiver module.

Decoding a Data/Checkpoint Message: This operation is the core of the Receiver code, which outlines the decoding of a message successfully received from the Transmitter.

```
/**
* @brief Decodes a received message, verifying its integrity, message type, and content.
* @param message_type The type of the received message
* (1 for Checkpoint Message, 0 for Data Message).
* @param received_data The data received in the message.
* @param received_parity The received parity bit.
*/
void decodeMessage(
        uint8_t message_type,
```

```
uint16_t received_data,
uint8_t received_parity) {

// Calculate the expected parity for the received data
uint8_t expected_parity = calculateParity(received_data);

// Check if the received data has the correct parity
if (received_parity == expected_parity) {

// If parity is correct, it's a valid message
// Check if it's a Checkpoint Message or a Data Message
if (message_type == 1) {
        // Checkpoint Message
        lcd_command(CLEAR);

        if (received_data == received_counter) {
                // Counter matches the expected value
                // Display success message on LCD
                lcd_putstring("Checkpoint Good!");
                lcd_command(LINE_TWO);
                char lcd_message[20];
                sprintf(lcd_message, "All OK = %d", received_data);
                lcd_putstring(lcd_message);
                displaySuccess();
        } else {
                // Counter does not match the expected value
                // Display error message on LCD
                lcd_putstring("Counter Mismatch");
                lcd_command(LINE_TWO);
                char lcd_message[20];
                sprintf(lcd_message, "Updating %d->%d", received_counter, received_data);
                lcd_putstring(lcd_message);
                received_counter = received_data;
                displayError();
        }
        } else if (message_type == 0) {
                // Data Message

                // Display the received ADC value on the LCD
                char lcd_message[20];
                sprintf(lcd_message, "ADC Value: %d", received_data);
                lcd_command(CLEAR);
                lcd_putstring(lcd_message);
                lcd_command(LINE_TWO);
                lcd_putstring("RECEIVED");
```

```
                        // Successful receive
                        received_counter++;
                        displaySuccess();
                }
        } else {
                // Parity error - corruption has occurred
                // Display error on LED0
                lcd_command(CLEAR);
                lcd_putstring("Invalid Parity");
                lcd_command(LINE_TWO);
                lcd_putstring("Corruption Found!");
                displayError();
        }
}
```

`decodeMessage` defines the implementation for decoding and processing received messages in the Receiver module.

It starts by calculating the expected parity bit for the received data and then checks if the received parity matches the expected value, indicating the integrity of the message. If the parity is correct, it proceeds to identify the message type (Checkpoint or Data) and performs the relevant actions. If there's a parity error, indicating data corruption, the function displays an error message.

For Checkpoint Messages, it compares the received counter to the expected value, displaying success or error messages on the LCD accordingly.

In the case of Data Messages, it displays the received ADC value and increments the received counter.

This function is essential for maintaining data integrity, interpreting message types, and providing feedback based on the received data.

Displaying Success: `displaySuccess` is a short but essential piece of code that manages the visual feedback to inform the user when a Data/Checkpoint Message transmission & receiving has been successful. It uses the LED3 (connected to PB3) to indicate success. When invoked, it activates LED3 by setting the corresponding GPIO pin to the high state for 2 seconds. The LED is then turned off by resetting the

corresponding GPIO pin to the low state, ensuring it's ready for the next feedback event. This function contributes to the user interface and feedback mechanism in the system, making it clear when tasks have been executed without issues.

```
/**
* @brief Display success by turning on LED3 (PB3) for 2 seconds.
*/
void displaySuccess(void) {
        LL_GPIO_SetOutputPin(GPIOB, GPIO_PIN_3);
        delay(400000);
        LL_GPIO_ResetOutputPin(GPIOB, GPIO_PIN_3);
}
```

Displaying Error: `displayError`, similar to the `displaySuccess` function, handles visual feedback, but for error conditions, when an error has occurred or been found in the transmission or decoding of a Data/Checkpoint Message. It uses LED3 (connected to PB3) to indicate errors. When invoked, it activates LED3 by repeatedly toggling its state every 100ms for a duration of 2 seconds (a total of 20 toggles). This flashing pattern provides a distinct visual cue for error situations. After the flashing sequence, the function ensures the LED is turned off by resetting the corresponding GPIO pin to the low state. This visual signal is valuable for notifying users about issues or errors in the system, aiding in prompt error recognition and diagnosis.

```
/**
* @brief Display error by flashing LED3 (PB3) for 2 seconds with 100ms intervals.
*/
void displayError(void) {
        for (int i = 0; i < 20; i++) {
                HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_3);
                delay(40000);
        }
        LL_GPIO_ResetOutputPin(GPIOB, GPIO_PIN_3);
}
```

Validation and Performance

The performance of our light-based messaging system has been a critical aspect of our project, aiming to establish a robust and reliable communication channel between the Transmitter and Receiver nodes. Our system demonstrated impressive performance during testing, consistently transmitting and receiving data messages with adequate responsiveness. Each transmitted data message was reliably received and decoded, confirming that the messaging system's encoding and decoding processes work seamlessly. To verify the accuracy of our data transmission, we conducted numerous test cases using a voltmeter to measure the ADC values at both the Transmitter and Receiver ends. The results consistently matched the expected values, affirming the system's ability to maintain data integrity throughout the communication process.

For a more tangible demonstration of our system's performance, we have included a video showcasing its operation. The video presents a step-by-step walkthrough of our project, illustrating the message transmission from the Transmitter to the Receiver. It highlights the process of encoding, transmitting, and decoding Data Messages, emphasising the reliability and precision of our messaging system. The video also includes scenarios where Checkpoint Messages are transmitted, ensuring that data consistency is maintained and any potential loss of data messages is detected.

Demonstration:

https://drive.google.com/file/d/18FBRoR1yJhWo7C8lpgA_s_45IFujpcr8/view?usp=drivesdk

Our system's successful operation, combined with the following test cases and the demonstration video, underlines its high performance and effectiveness in real-world scenarios.

Test Case	Actions	Expected Behaviour	Success?
Transmitting a Data Message with the maximum POT value (Data Payload)	- Turn the potentiometer knob to the maximum value - Press PB0	- Receiver node interprets this value (4095) and displays it on the screen	☑
Transmitting a Data Message with the	- Turn the potentiometer knob to	- Receiver node interprets this value (0)	☑

minimum POT value (Data Payload)	the minimum value - Press PB0	and displays it on the screen	
Transmitting a Data Message with the random POT value (Data Payload)	- Turn the potentiometer knob to a desired value - Press PB0	- Receiver node interprets this value and displays it on the screen	☑
Transmitting a Checkpoint Message after normal previous communication	- Send a few Data Messages (above actions) - Press PB1	- Receiver node interprets checkpoint value - Corrects stored value if they do not match	☑
Transmitting a Checkpoint Message after abnormal previous communication	- Press PB1 after sending an interrupted message	- Receiver node interprets checkpoint value - Corrects stored value if they do not match	☑
Interrupting the transmission of a Data Message with RESET	- Start sending a Data Message by pressing PB0 - Before the message is completed, press RESET button	- Message fails to interpret - Error message displayed on LCD - Error flashed on PB3	☑
Interrupting the transmission of a Checkpoint Message with RESET	- Start sending a Data Message by pressing PB1 - Before the message is completed, press RESET button	- Message fails to interpret - Error message displayed on LCD - Error flashed on PB3	☑
Interrupting the transmission of a Data Message with a Checkpoint Message	- Start sending a Data Message by pressing PB0 - Before the message is completed, send a Checkpoint Message using PB1	- Sender does not respond to PB1 interrupts - Sender completes sending Data Message - Receiver interprets complete message	☑
Interrupting the transmission of a Checkpoint Message with a Data Message	- Start sending a Checkpoint Message by pressing PB1 - Before the message is completed, send a Data	- Sender does not respond to PB0 interrupts - Sender completes sending Checkpoint Message	☑

	Message using PB0	- Receiver interprets complete message	
Attempting to transmit overlapping Data Messages	- Send a Data Message using PB0 - Before it is completed, attempt to send another Data Message using PB0	- Sender does not respond to second interrupt - Receiver interprets complete first message	☑
Attempting to transmit overlapping Checkpoint Messages	- Send a Checkpoint Message using PB1 - Before it is completed, send another Data Message using PB0	- Sender does not respond to second interrupt - Receiver interprets complete first message	☑

In summary, our light-based messaging system has proven its reliability and performance through extensive testing, including ADC value measurements and successful message transmission between the Transmitter and Receiver nodes. The addition of the video demonstration provides a comprehensive overview of our system's operation, further substantiating its ability to accurately transmit and receive data messages. This validation process assures us that our messaging system is not only a concept but a practical solution for effective light-communication in embedded systems.

Conclusion

In conclusion, our light-based messaging system has proven to be highly successful in its primary goal of enabling efficient and reliable communication between embedded systems. Throughout the project, we meticulously designed, implemented, and tested our messaging system, ensuring its robustness and precision. The performance validation demonstrated its ability to consistently transmit and receive data messages while maintaining data integrity. With our sophisticated encoding and decoding process, as well as the implementation of Checkpoint Messages, our system has shown great promise in facilitating reliable and error-resistant communication.

The system's approach of utilising light signals for data transmission offers a multitude of potential applications and could indeed be considered a useful product. Embedded Systems often require reliable and efficient communication methods, and our system's performance showcases its potential in various real-world scenarios. It can find applications in areas where wireless communication may be challenging or energy-intensive, making it a cost-effective solution. Additionally, the system's flexibility and precision in transmitting analogue data opens up opportunities in fields such as industrial automation, environmental monitoring, and home automation. As technology continues to evolve, innovative solutions like our light-based messaging system have the potential to address communication challenges in numerous impactful ways.

References

Stenitzer, G. (2021). ‘How to tell if ongoing messages differ enough to be impactful in seconds?’ [online] Crystal Clear Communications. Available at: https://crystalclearcomms.com/how-to-tell-if-ongoing-messages-differ-enough-to-be-impactful-in-seconds/ [Accessed 27 Sep. 2023].

GeeksforGeeks. (2018). *Manchester Encoding in Computer Network*. [online] Available at: https://www.geeksforgeeks.org/manchester-encoding-in-computer-network/ [Accessed 3 Oct. 2023].

Wavedrom.com. (2016). *WaveDrom - Digital timing diagram everywhere*. [online] Available at: https://wavedrom.com/ [Accessed 21 Oct. 2023].