Embedded Systems II
EEE3096S/EEE3095S
Mini-Project Part A

Message by Light

YNGFYN001
NTHCHI002
GMLMOG016
MHMUWA001

OCTOBER 24, 2022

Table of Contents

Note: Code files and the demo video (demo_A.mp4) are uploaded to git repository
https://github.com/youngfynn/YNGFYN001_NTHCHI002_GMLMOG016_MHMUWA001_Mini_Project_Part_A/upload

1 Introduction

In this project we were tasked with designing and developing an embedded system which makes use of two STMDiscovery boards to transmit data over a LoT (Light of Things) network.

One board is used to send data packets and the other to receive data packets. This embedded system requires aspects of hardware and software interfacing, an ADC (analog to digital converter) and developing a communications protocol to transmit a sample from the ADC via light by using an LED which transmits ADC samples to an LDR on the receiver. In reality, the data is transmitted over a GPIO connection between the two boards. Once the data is received, the receiving STMDiscovery board will decode the sent data and display it via UART on a PC.

For the implementation of the embedded system, two STM32F051R8Tx development boards were used along with a UART. An LED was connected to the transmitting STMDiscovery to provide a visual representation of the data being sent in the packet. To receive the data, a GPIO line was used to transfer the packet of data from the transmitting STMDiscovery board to the receiving STMDiscovery board.

The LoT message protocol is designed to send data packets from the transmitting STMDiscovery board to the receiving STMDiscovery board. The data is packaged into a packet of 16 bits. The bits in the packet are made up of a start bit, a type bit, 12 bits of data which are a binary representation of the data being sent, a parity bit and a stop bit.

2 Requirements & LoT Message Protocol

The transmission of data was done by connecting the two STM boards with a wire instead of using light since this was deemed to be more reliable and simpler to implement. This reduced the scope of the project which made it easier to handle.

2.1 Block Diagram

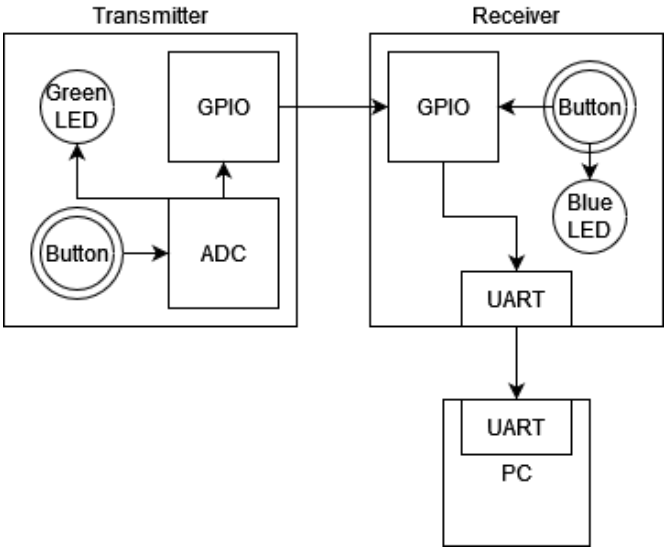

Figure 1 – block diagram of the system

Figure 1 shows the simple idea of how our Transmitter and Receiver works. The Transmitter and Receiver are both STM32 boards using a UART to send messages between them. The system is simplex because messages can only travel from the transmitter to the receiver. Messages are sent in serial and are checked and validated by our message structure which is shown next.

The system diagram will be explained in more detail in the next section.

2.2 Message Structure

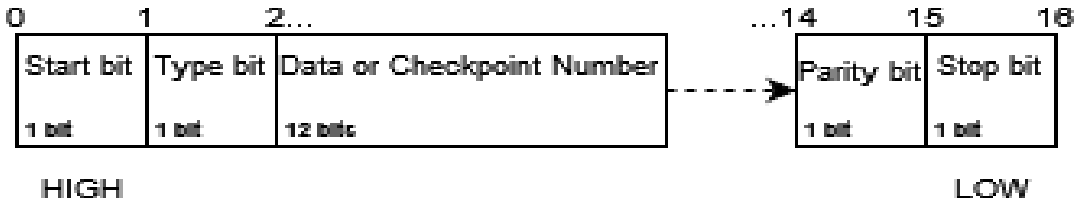

Figure 2 – the message structure

The message structure is shown in *Figure 2* and is as follows:

- **Bit 0:** The start bit, notifies the receiver that message transmission is commencing (HIGH)
- **Bit 1:** The type bit, determines whether the message is an ADC sample (HIGH) or a checkpoint message (LOW)
- **Bit 2 – 13:** The data bits, contains either the ADC sample or the number of samples sent by the transmitter if it is a checkpoint message
- **Bit 14:** The parity bit, used as a basic form of error detection. Even parity is used for this system
- **Bit 15:** The stop bit, notifies the receiver that message transmission is complete (LOW)

2.3 Timing Diagram

Figure 3 – timing diagram

The timing diagram in *Figure 3* is an example of how the message would be sent to the receiver.

The first bit past idle is the HIGH starting bit which lets the receiver know a message is about to be sent.

The next bit, the type bit, could be either HIGH or LOW but for this example it has been set to HIGH to indicate the following bits will be an ADC sample and not a checkpoint number.

The next 12 bits contains the data *01011100101*, which is then validated with the even parity bit which is set to low because there is an even number of 1's in the data.

Finally, the last bit is set to LOW to indicate the end of the message.

3 Specification and Design

3.1 Transmitter

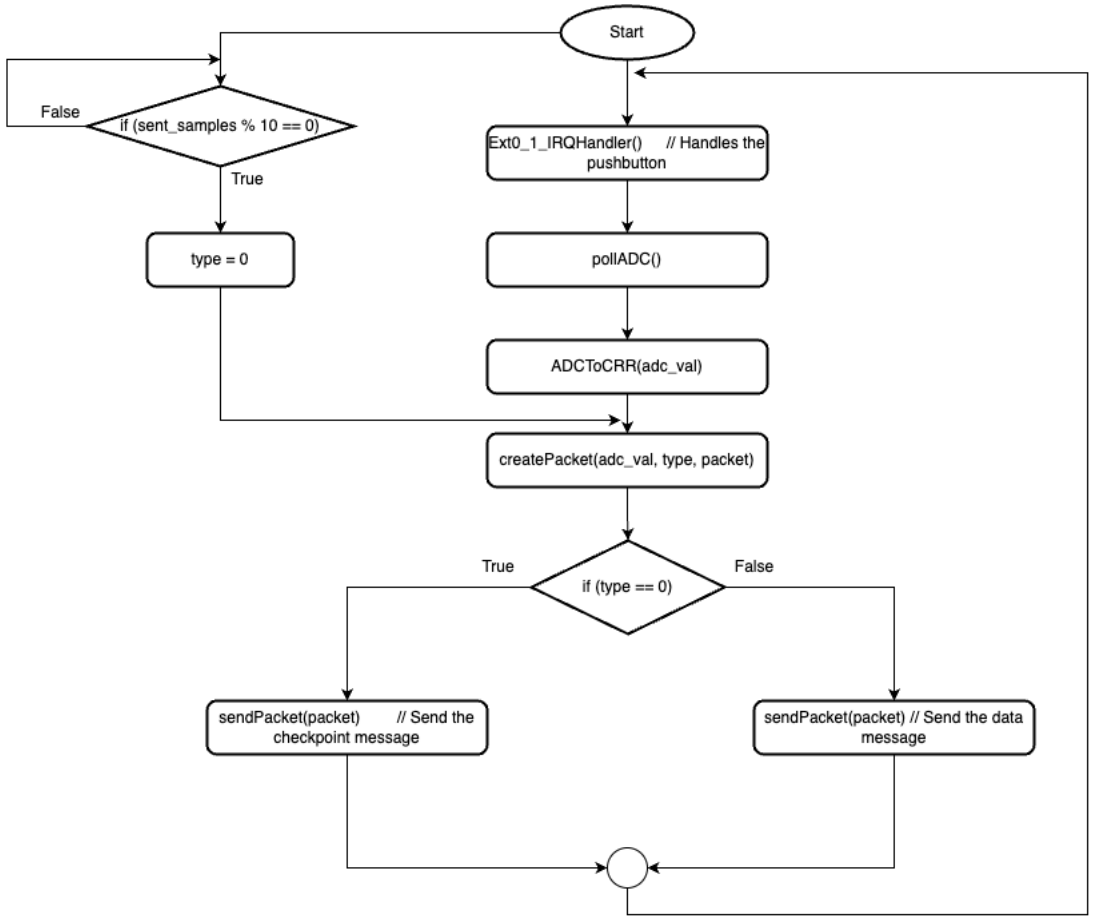

Figure 4 – flowchart for the LoT transmitter

Figure 4 shows how the transmitter operates. At the start, it waits for the pushbutton to be pressed to initiate transmission. This generates an interrupt which is then handled by polling the ADC using *pollADC()*, converting this value to a CRR value using *ADCtoCRR()* (this is for setting the brightness of the green LED) and then creating and sending a packet using *createPacket()* and *sendPacket()* respectively.

When the number of samples sent, *sent_samples*, reaches a multiple of ten, a checkpoint message is sent with the type argument set to 0.

3.2 Receiver

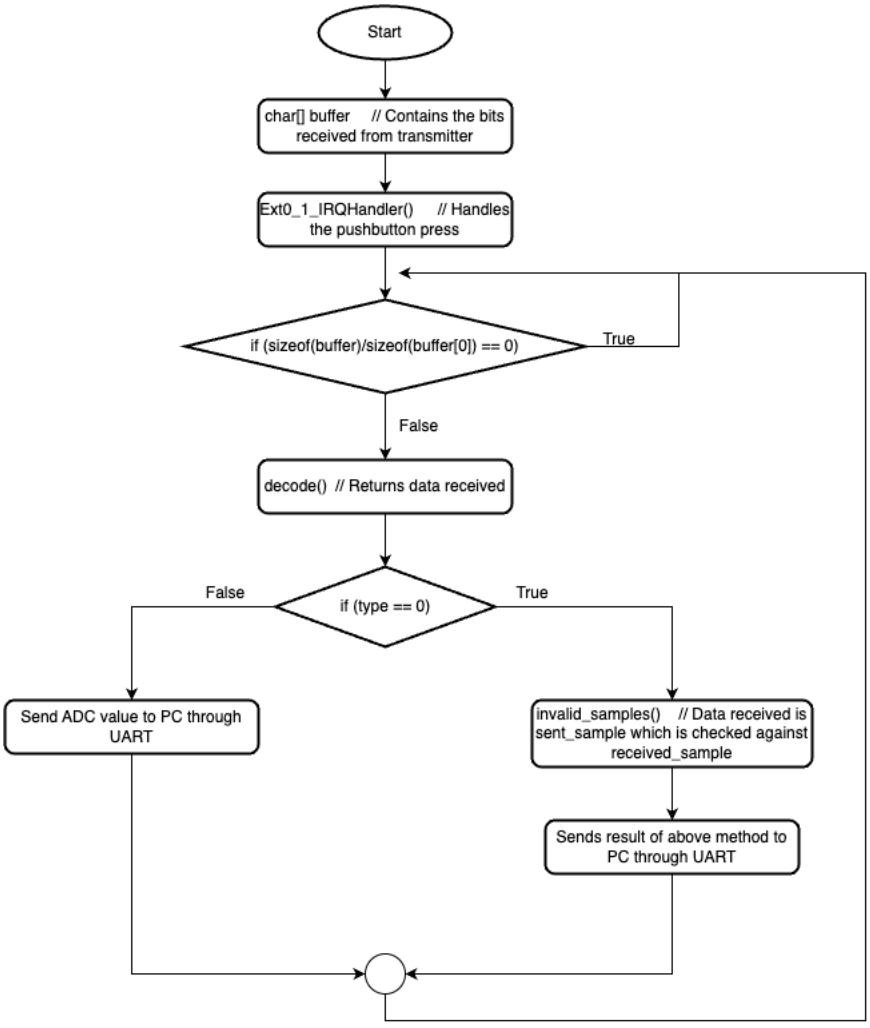

Figure 5 – flowchart for the LoT receiver

Figure 5 shows how data is received and handled before it is transmitted via UART to the desktop computer and displayed. The system waits for the pushbutton to be pressed and then starts listening for data transmissions. When data is received, it is saved in the *buffer* variable. The *decode()* method is used to extract the data and convert it to a decimal value.

If the type bit is zero, the transmission is a checkpoint message and the data sent is the number of samples transmitted by the transmitter, *sent_samples*. This value is compared to the receiver's own counter of samples received, *rcvd_samples*, and if the two differ the *invalid_samples()* method is used to reset *rcvd_samples* to the transmitter's value and display an invalid signal on the blue LED. A checkpoint message is then sent via UART to the PC.

If the transmission is of type one, an ADC sample, the data is decoded and then sent via UART to the PC.

3.3 Circuit diagram

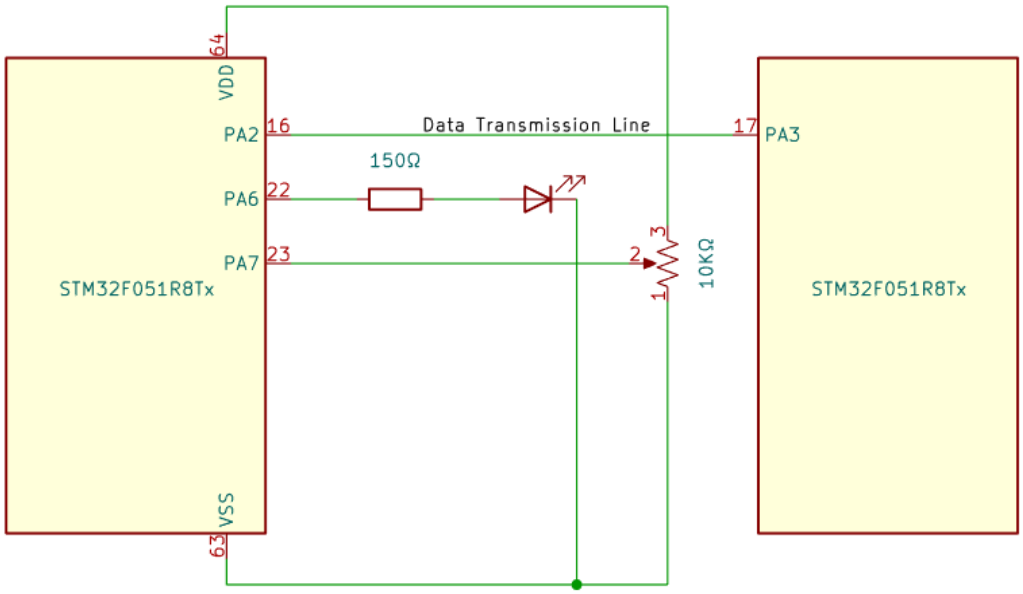

Figure 6 – circuit diagram for the LoT system

As shown in *Figure 6*, the left STM32 acts as the transmitter and the right STM32 acts as receiver. The Data Transmission Line using pin PA2 on the transmitter and pin PA3 on the receiver is used to send bits from the transmitter to the receiver.

The LED connected to pin PA6 on the transmitter is used to show when a message is being sent through the data line. The potentiometer is used to change the voltage value that is read by the ADC on pin PA7.

4 Implementation

Transmitter

createPacket

```
void createPacket(uint32_t data, char type, char *packet)   {
        char data_binary[12] = {0};
        char parity = 0;

        packet[0] = 1;                  //start bit
        packet[1] = type;    //type bit

        // convert data to binary
        for(int i = 0; data > 0; i++)
        {
                data_binary[i] = data % 2;
                data = data / 2;
                if (data_binary[i] == 1)
                        parity += 1;
        }
        // package binary data
        for(int i = 2; i <= 13; i++)
        {
                packet[i] = data_binary[i - 2];
        }
        //parity bit
        if (parity % 2 == 0)
                packet[14] = 0;
        else
                packet[14] = 1;
}
```

Code Snippet 1 – the createPacket method

The *createPacket* method assembles a binary data packet containing a start bit, type bit, data bits, parity bit, and end bit.

sendPacket

```
void sendPacket(char *packet)		{
	// send to receiver over GPIO line
	for (int i = 0; i < 16; i++)		{
		sprintf(buffer, "%d", packet[i]);
		HAL_UART_Transmit(&huart2, (uint8_t*)buffer, 1, 1000);
	}

	// send over LED (human-viewable)

	//send start bit (exaggerated delay)
	HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);
	HAL_Delay(2000);

	// send rest of packet
	for (int i = 1; i < 16; i++)		{
		if (packet[i] == 1)
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);
		else if (packet[i] == 0)
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);

		HAL_Delay(500);
	}

	// send ending bits (pulses)
	HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);
	for (int i = 0; i < 10; i++)		{
		HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_6);
		HAL_Delay(100);
	}
	sent_samples++;	//increment counter for samples sent
}
```

Code Snippet 2 – the sendPacket method

The *sendPacket* method sends a data packet to the receiver over the GPIO line and then sends it in a human-viewable form over the attached LED with appropriate delays for easier viewing. The message starts with a start bit of 2 seconds and ends with a series of 4 pulses.

pollADC

```
uint32_t pollADC(void)		{
	// poll ADC for voltage value
	HAL_ADC_Start(&hadc);
	HAL_ADC_PollForConversion(&hadc, 1000);
	int val = HAL_ADC_GetValue(&hadc);
	HAL_ADC_Stop(&hadc);
	return val;
}
```

Code Snippet 3 – the pollADC method

The *pollADC* method returns an integer value received from the potentiometer connected to the onboard ADC. It first starts the ADC, polls it, gets a value, and then stops the ADC.

Receiver

While loop

```
// if button has been pressed, start listening for transmissions from transmitter
if (start == 1)         {
            HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
            // check if data is in buffer
            if (__HAL_UART_GET_FLAG(&huart2, UART_FLAG_RXNE) == SET)      {
                        // get packet from buffer
                        HAL_UART_Receive(&huart2, (uint8_t*)rcv_buffer, sizeof(rcv_buffer), 1000);

                        // if received sample
                        if (rcv_buffer[1] == 49)         {
                                    rcvd_samples++;                 // increment num. of samples received
                                    int val = decode();  // get value of ADC sample

                                    // transmit to UART (to PC/PuTTY)
                                    memset(trans_buffer, 0, sizeof(trans_buffer));     // clear buffer
                                    sprintf(trans_buffer, "\r\nADC Sample Value: %d", val);
                                    HAL_UART_Transmit(&huart2, (uint8_t*)trans_buffer, sizeof(trans_buffer), 1000);
                        }
                        // if received checkpoint message
                        else if (rcv_buffer[1] == 48)   {
                        int check = decode();
                        // check if sent vs received samples match
                        if (check != rcvd_samples)      {
                                    sprintf(trans_buffer, "\r\nCheckpoint failed, updating to %d samples received (was %d)",
check, rcvd_samples);
                                    HAL_UART_Transmit(&huart2, (uint8_t*)trans_buffer, sizeof(trans_buffer), 1000);
                                    invalidSamples(check);
                        }
                        else        {
                                    sprintf(trans_buffer, "\r\nCheckpoint passed (%d samples received)", rcvd_samples);
                                    HAL_UART_Transmit(&huart2, (uint8_t*)trans_buffer, sizeof(trans_buffer), 1000);
                        }
            }
}
HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_8);
}
```

Code Snippet 4 – the code in the while loop of the receiver

The while loop code for the receiver handles the receiving of messages from the transmitter, it waits until the blue button is pressed (*if start == 1*), then waits until there is data in the buffer, and then performs necessary actions to decode it.

decode

```
int decode(void)    {
        char data[12];
        int decimal_num = 0;

        memcpy(data, rcv_buffer + 2, 12);       // copy data part of packet to new array

        // convert binary data to decimal value
        for (int i = 0; i < 12; i++)
        {
                if (data[i] == 49)
                        decimal_num = pow(2, i) + decimal_num;
        }
        return decimal_num;
}
```

Code Snippet 5 – the decode method

The *decode* method takes in a data packet from the buffer and extracts the data part from it, taking care that the data is represented in little endian format. It then returns the data value.

5 Validation and Performance

The system performance is described in detail in the attached video (*demo_A.mp4*), but it proceeds as follows:

1. The blue button is pressed on the receiver to enable listening for transmissions
2. The blue button on the transmitter is pressed to get an ADC sample, package it, and send it to the receiver.
    a. The green LED on the transmitter is used to display the value of the ADC sample (in terms of brightness)
    b. The red LED attached to the breadboard shows a human-viewable representation of the data packet
    c. The sample value is displayed on the PC
3. At a checkpoint (after 10 samples are sent), a checkpoint message is sent by the transmitter
    a. If the receiver's sample count is the same, a message is displayed on the PC and the operation continues
    b. If the receiver's sample count differs, an error message is displayed on the PC, the receiver's blue LED pulses 4 times, and the receiver's sample count is updated to reflect that of the transmitter
4. The blue button on the receiver is pressed again to disable listening for transmissions

The system was validated by checking the binary messages that were sent against the values of the ADC sampled. For example, the ADC value was logged on the transmitter side and the binary value received on the receiver side. These were then compared to validate the transmission.

Similarly, a parity bit was used to verify the integrity of each data packet sent.

6 Conclusion

Based on the validation of the system, the design and development of the data communication protocol that enabled the sending and receiving data between two boards was a success.

This system could be a useful product if a longer connecting line was used between two boards, one in a sensor location (transmitter) and one in a more central location (receiver). However, without the full implementation of the LoT aspect, it is obviously less useful in transmitting data by LED.

In conclusion, the system performs as it should in a reliable and efficient manner, while meeting the minimum system requirements.