

EEE 3096S

Embedded Systems II

Mini Project - Message by Light

20 October 2022

BRYLIA002
STRREI003
HGXSEB001
NCHGAB001

1. Introduction

The aim of this investigation is to assess the feasibility and implementation of a Light of Things (LOT) data transmission link. This will be done by examining the client's requirements, reformulating them as specifications and design choices, from which an implementation stage will follow. This implementation will then be validated and its performance analysed, after which conclusions are drawn.

Due to the nature of the implementation, there are certain restrictions that need to be accounted for. Since only one communication channel is available, an asynchronous signal will be chosen. Due to the limitations specific to this problem, a custom communication will be used, including rudimentary error checking such as parity bits and send / receive counters. Due to the presence of ambient light, a digital communication will be used, as opposed to an analog one, as this should make the transfer of information much more reliable, and less prone to errors. The receiver will read the received intensity as an analog signal, as this allows the user to choose a logic threshold level more easily. An alternative to this would be using a comparator or Schmidt trigger, however setting the threshold would likely involve changing the circuit, which is far less convenient than doing it in software.

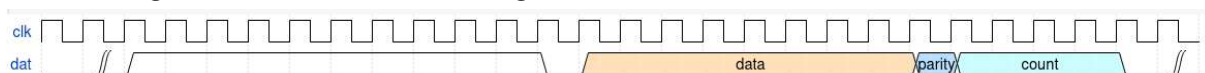
Since both a transmitter and receiver need to be implemented, the team will be split into two groups, each developing one of these independently. As soon as both have working prototypes, an integration phase will follow, in which the complete system is tested. Requirements, specifications and implementation details will be assessed individually for the transmitter and receiver, while performance, validation and conclusions will be investigated for the integrated system.

2. Requirements

Protocol requirements

Message structure: Since only one communication channel is available, the communication has to be asynchronous. Both the receiver and transmitter therefore need to keep their own time.

The message structure has the following features:



- data: 8 bit value to be read from the potentiometer. This is just sent as binary data, where a high signal for 1 clock cycle indicates a 1, and a low indicates a 0.
- parity bit: Even parity is chosen. This ensures that the longest period of consecutive 1's is 8 cycles. This occurs when data is 0b11111111. Since this has an even parity, the parity bit will be 0. This is important for the start condition.
- Count: a 4 bit number sent to ensure that the receiver has not missed any packets. Both transmitter and receiver have a counter, which is incremented for each transmitted / received package. The transmitter count is sent at the end of every

packet. If there is a mismatch in count between transmitter and receiver, the receiver can notify the user that a packet has been missed.

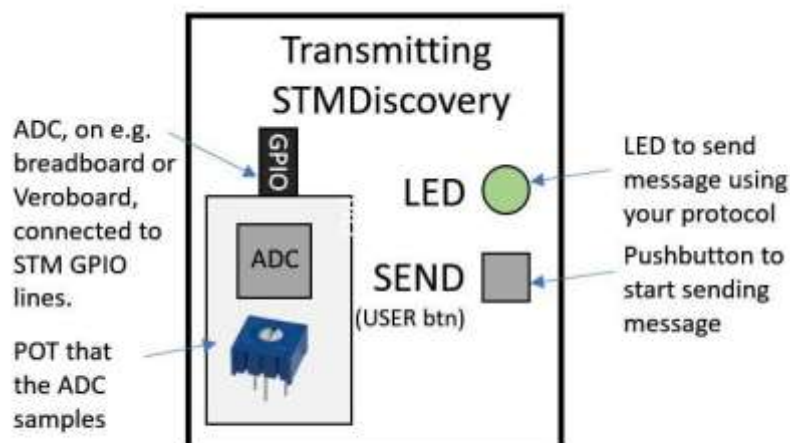
- Start condition: As described in the parity bit section, the longest possible period of consecutive 1's is 8 clock cycles. In order to ensure that the receiver starts reading at the beginning of the packet, the start condition is chosen to be a high output for 10 clock cycles. This should prevent any unsynchronised reading, with some extra redundancy included to make errors more unlikely to occur.
- Sync bit: Due to the implementation of the transmitting and receiving code, it may be difficult to guarantee that both transmitter and receiver have the same cycle speed. To avoid any timing related errors, a transition from the high start condition to the low sync bit is used to synchronise the receiver to the transmitter.

Due to the same synchronisation issue described in the sync bit section, the data is sampled at the centre of a clock cycle, meaning that there is a margin for error if there is a mismatch in the cycle times of the transmit and receive setups.

Note that by increasing the length of the start condition, it should be possible to differentiate between two types of messages, for example a data message vs. a count message. However, since the count is transmitted with every data packet, this is not implemented. The only modification needed to implement this is changing the start condition check, as well as adding a new condition to check for the new message type.

Transmitter requirements:

Diagram 1: Showing the transmitter system layout.

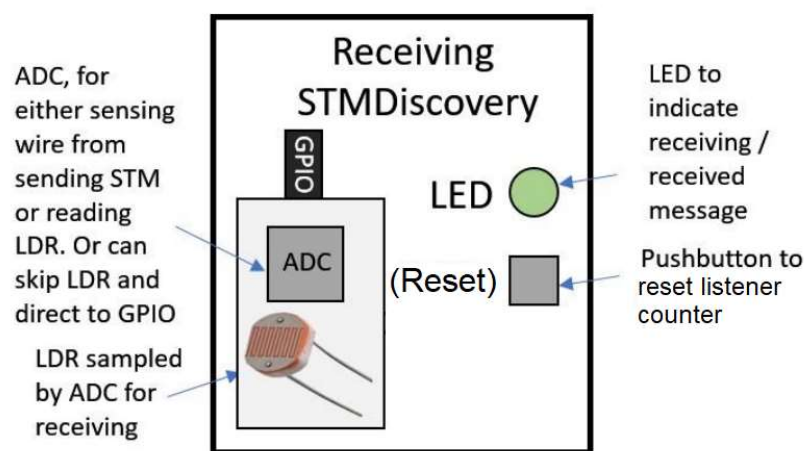


- The c program must wait until the onboard STM user push-button is pressed.
- When the push-button is pressed, the value from the POT must be sampled as a decimal value.
- The program must convert this decimal value into a binary value.
- Using an output pin that fluctuates between high(1) and low(0) the binary message must be outputted to this pin using a delay between each bit.
- An LED circuit must be set up to go on when the output pin is high and off when the output pin is low.

- The program must count the number of 1's in the binary message and determine whether the parity bit that is outputted is a 1 or a 0.
- The parity bit must be sent in the same way as the binary message.
- The program must keep track of how many times the pushbutton is pressed as this represents the number of samples that have been sent.
- The number of samples sent must be converted to binary and sent in the same way as the binary message.

Receiver requirements:

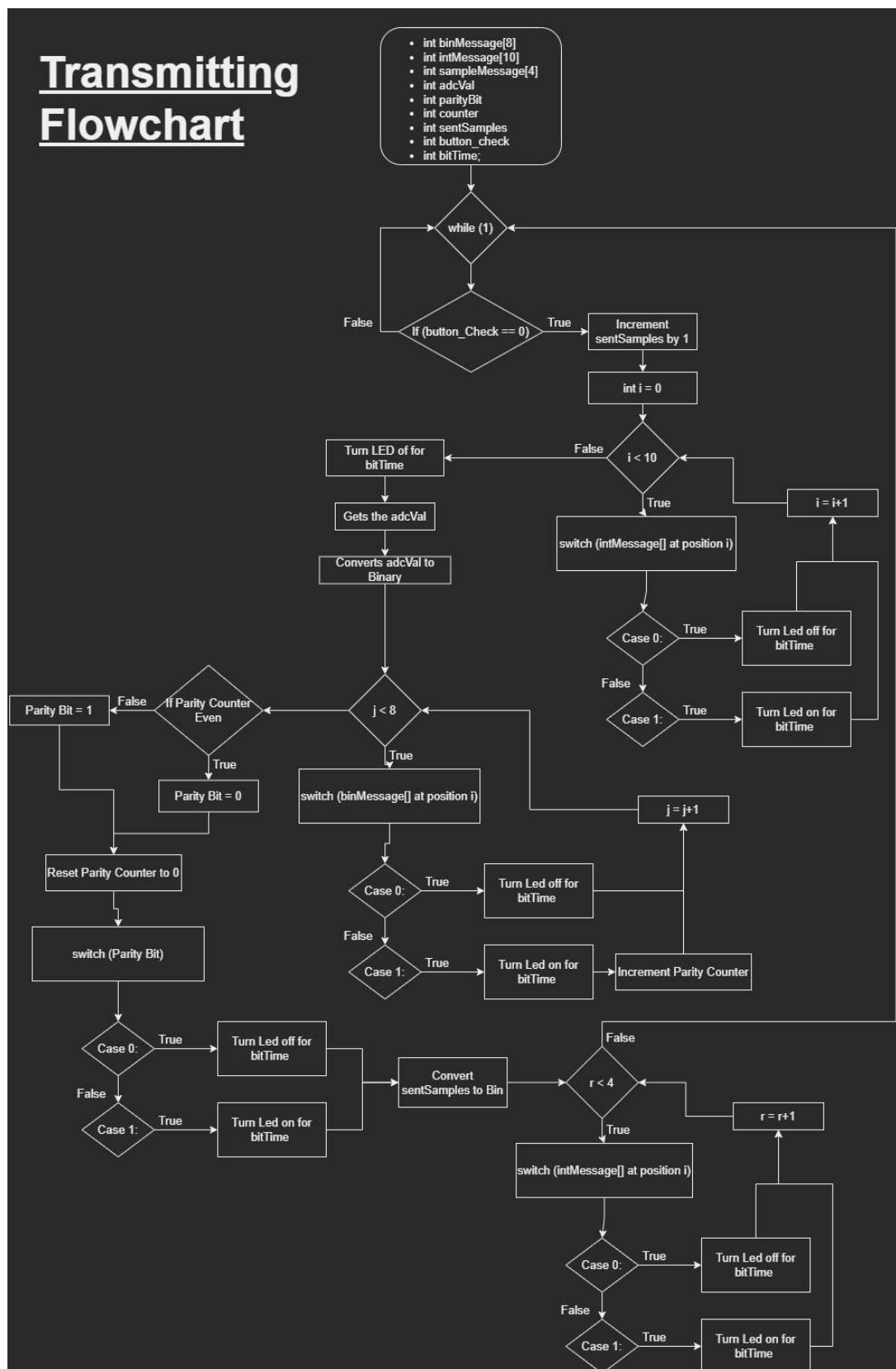
Diagram 2: Showing the Receiver system layout.



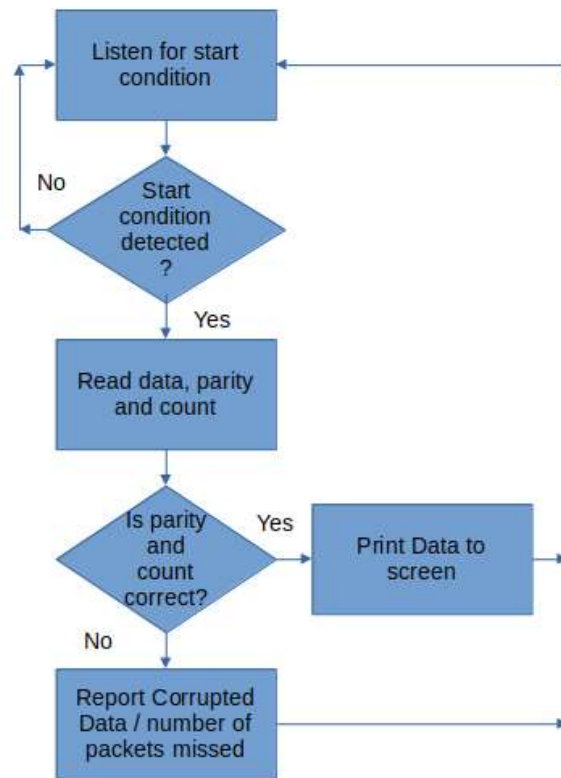
- Receiver must be continuously listening in order to pick up a message from the Transmitter
- Receiver should keep track of how many samples it has received and compare it to the number of samples transmitted
- Receiver must only start recording data when the start condition is met (High signal for 10 clock cycles)
- Must be able to sample the following data at the correct time intervals to avoid missing bits or sampling the same bit twice
- Must be able to check the parity bit to see if an error has occurred.
- Must be set up to separate the ADV data, the parity bit and the sent-count binary values.
- ADC must be set up to be able to distinguish between the LED state being ON vs OFF
- ADC must still be able to function even when small changes in ambient light occur - such as a person walking passed casting a shadow
- Receiver program must be able to convert the binary data back to decimal values
- Internal counter must be reset by the reset button or by reflashing the STM

3.Specification and Design

Transmitting Flowchart



Receiver flowchart



The receiver has three main states:

1. Listening for a start condition
2. Reading the data stream
3. Processing and displaying data

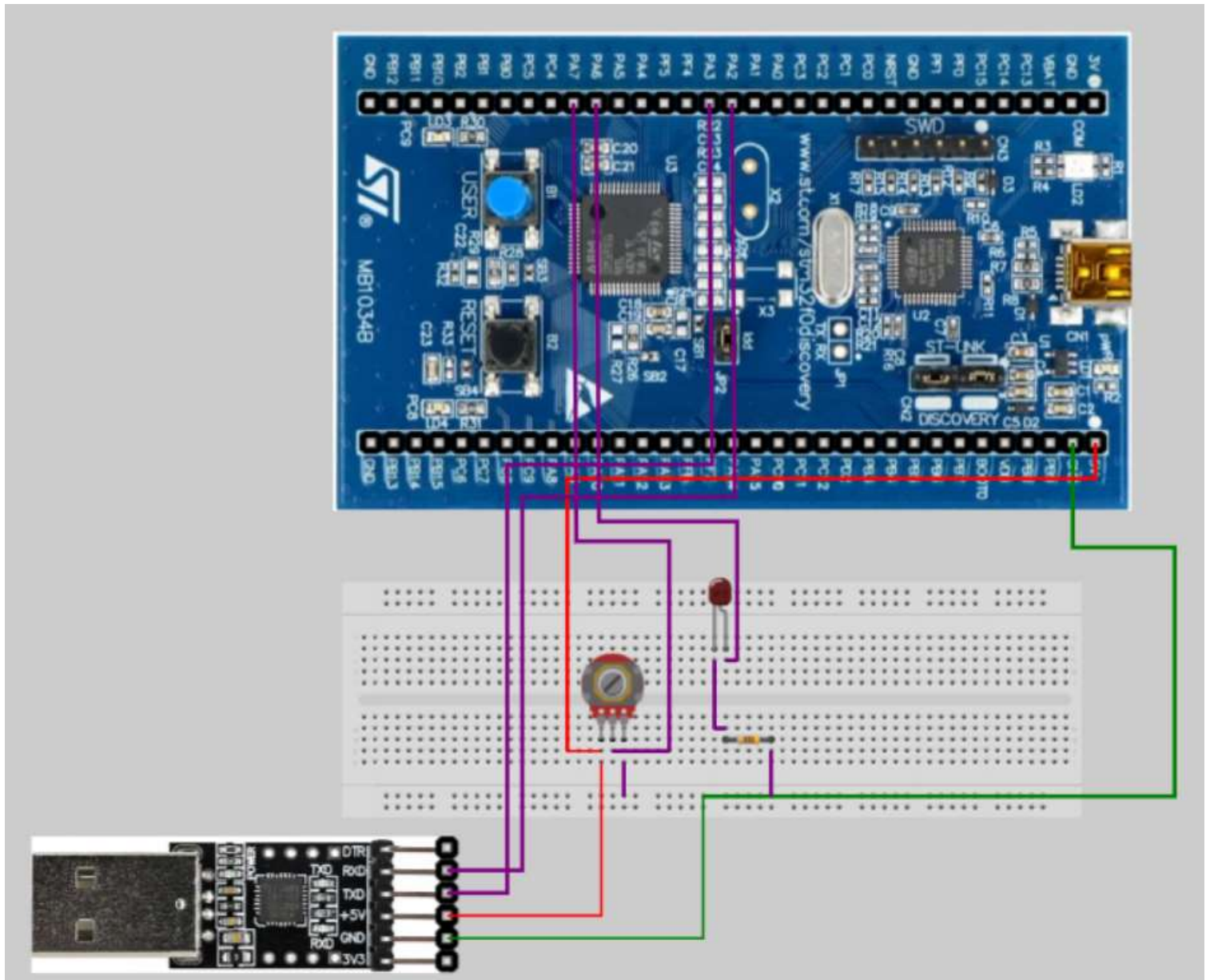
The receiver will likely spend most of its time listening for data.

Once a signal is detected, it will advance into the next state, in which an incoming data stream is read.

As soon as all data has been captured, this data is processed, after which it is reported to the user. If any errors have occurred, a warning will be sent as well.

Having reported the processed data, the receiver transitions back into the listening state.

Transmitter Schematic



Connections

UART

- RX connected to PA2
- TX connected to PA3
- GND to GND
- 5v to 5v

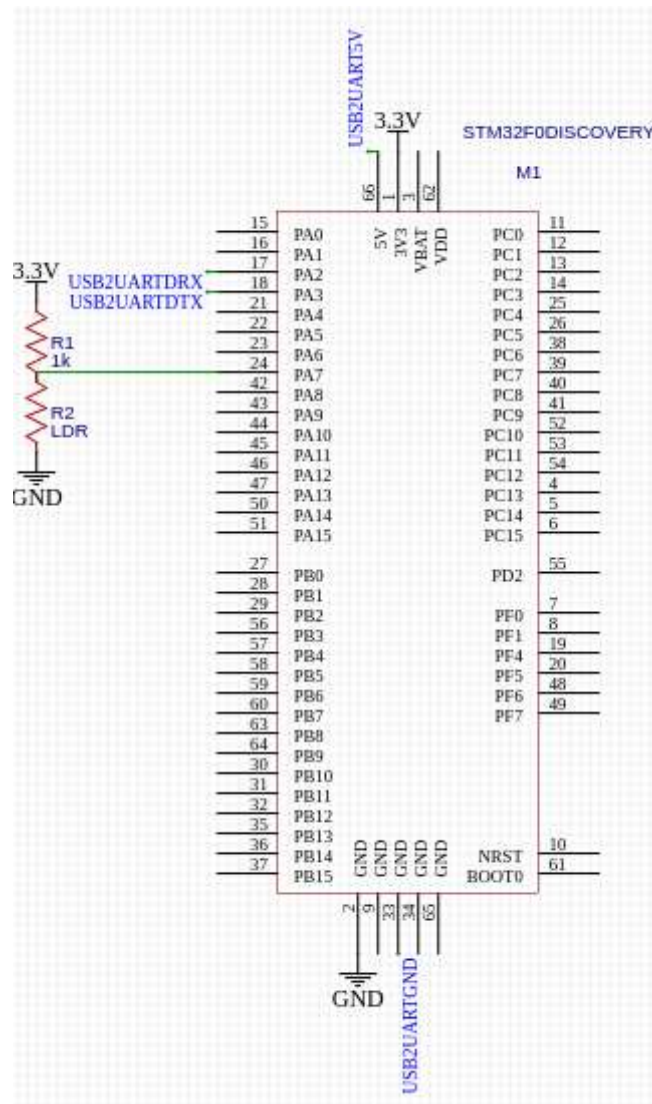
ADC input

- Pot output connected to PA7

LED and Resistor

- LED connected to output pin PA6

Receiver Schematic



The LDR is set up as a simple voltage divider, and is fed into the ADC at PA7. Through testing, this has proven to give a reliable range in outputs, from which an LED on state can easily be differentiated. Even with a relatively large amount of ambient light, the difference in input voltage was significant enough to be detected. Note that with the current configuration of the LDR, a drop in voltage signifies a high LED reading. This has to be accounted for in software. This is done to reduce current flow through the voltage divider when no signal is present, as the default state is off. Through experimentation, it was found that checking whether the ADC value drops below 2500 is reliable in determining if a signal is present.

4. Implementation

Transmitter Implementation:

The transmitter system uses important blocks of code to take in, analyse and output data.

Code snippet 1: Showing the process of sending the interrupt message

```
if(delay == 1){
    sentSamples++;

    //Printing the interrupt message
    sprintf(buffer, "The Interrupt Message: \r\n");
    HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);

    for(int i=0;i<10;i++){

        sprintf(buffer1, " %d \r\n",intMessage[i]);
        HAL_UART_Transmit(&huart2, buffer1, sizeof(buffer1), 1000);

    }

    //Outputting interrupt to LED
    for(int j=0;j<10;j++){

        switch(intMessage[j]){
            case 0:
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);
                HAL_Delay(500);
                break;
            case 1:
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);
                HAL_Delay(500);
                break;
        }

    }

    //Low after interrupt
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);
    HAL_Delay(500);
}
```

This block of code waits for the pushbutton to be pressed. Once the pushbutton is pressed the if statement becomes true. The string "The Interrupt Message: " is then displayed to putty followed by the interrupt message in binary. This interrupt message contains ten 1's and is used by the receiver to make acknowledge that a binary message is about to be sent. The interrupt message is then outputted to the LED circuit with a 500ms delay between each bit.

Code snippet 2: Showing the scaling the ADC value

```
//Converting the ADC value to a smaller value to be made into 8 bits
adcVal = pollADC();
adcVal = adcVal*6;
adcVal = adcVal/100;
```

Originally the ADC value is a very large number (approximately 4000). To get this into an 8 bit binary number the maximum value it can be is 255. By scaling the ADC value down by a factor of 0.06, the value can then be converted to an 8 bit binary message.

Code snippet 3: Showing the decimal ADC to binary ADC conversion

```
void decToBin(adcVal){  
  
    for(int i=7;adcVal>0;i--)  
    {  
        binMessage[i]=adcVal%2;  
        adcVal=adcVal/2;  
    }  
  
}
```

This function converts the decimal adc value into a binary value and stores it in the binary message array. Note that the array is populated from the last index to the first to ensure that when the binary message is sent, the output is in the correct form.

Code snippet 4: Showing the binary ADC value being displayed and outputted

```
//Printing the binary message values  
for(int i=0;i<8;i++){  
  
    sprintf(buffer4, "%d \r\n",binMessage[i]);  
    HAL_UART_Transmit(&huart2, buffer4, sizeof(buffer4), 1000);  
  
}  
  
for(int j=0;j<8;j++){  
  
    switch(binMessage[j]){  
        case 0:  
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET);  
            HAL_Delay(500);  
            break;  
        case 1:  
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);  
            HAL_Delay(500);  
            counter++;  
            break;  
    }  
  
}
```

This code outputs the binary ADC value to putty. This is done to ensure that the ADC value recorded is converted to the correct binary message. The binary message is then outputted to PA6 where a 1 represents a high and 0 represents a low. This output has a 500ms delay between each bit.

Code snippet 5: Showing the parity bit

```
if(counter%2 == 0) {  
    parityBit = 0;  
}  
else {  
    parityBit = 1;  
}  
  
counter = 0;
```

Each time the binary message outputs a 1, the “counter” is incremented by 1. Using the above code, the parity bit is determined using even parity. The check for even parity is then used on the receiver side to determine whether the information was corrupted.

Code snippet 6: Showing the samples sent in decimal to binary

```
void sampToBin(dec){  
    for(int i=3;dec>0;i--)  
    {  
        sampleMessage[i]=dec%2;  
        dec=dec/2;  
    }  
}
```

Each time the button is pressed the number of samples sent is incremented by 1. This number must be converted to binary and sent at the end of the binary message. On the receiver side this number is then checked and determines whether it is missing information (a byte of data).

Display 1: Showing the structure of sent data on the transmitter side using putty.

```
The Interrupt Message:  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
1  
ADC: 135  
The Binary Message:  
1  
0  
0  
0  
0  
1  
1  
1  
The Parity Bit:  
0  
The Samples Sent: 1  
0  
0  
0  
1
```

Receiver Implementation:

The following code is grouped together into functional groups loosely representing the receiver flow chart.

Listen for start Condition & Start condition detected:

```
firstTick = HAL_GetTick();    //refresh start time

adcVal = pollADC(); //read value

if (adcVal < 2500){ //if receiver reads high signal
    while (adcVal < 2500){ //while the signal stays high
        secondTick = HAL_GetTick(); //refresh stop time
        adcVal = pollADC(); //refresh signal reading
    }
    if ((secondTick - firstTick)>4750){ //once the signal goes low,
        //check if long enough to qualify for start condition
        sprintf(buffer, "start condition >:D \r\n"); //show that start condition has been met
        HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);
        HAL_UART_Transmit(&huart2, "\r\n", sizeof("\r\n"), 1000);
    }
}
```

This snippet of code waits for a high signal. Once a high signal is detected, the duration of the high signal is checked using the getTick function. If the high signal is longer than a specified time, it is considered a start condition.

Read Data, Parity & Count:

```
for (int i=0; i<13; i++){ //read contents of message
    vals[i] = pollADC();
    HAL_Delay(500); //wait for 1 cycle before reading next
}

for (int i=0; i<9; i++){ //interpret ADC signal as binary data
    if (vals[i] < 2500){
        bits[i] = 1;
        parityCheck++; //update parity counter
    }
    else{
        bits[i] = 0;
    }
}

for (int i=0; i<13; i++){ //convert count to bits without incrementing parity
    if (vals[i] < 2500){
        bits[i] = 1;
    }
    else{
        bits[i] = 0;
    }
}
```

In the above code, 13 readings are taken at the correct frequency. Once all of them have been read, they are converted to actual 1's or 0's. Each time a data bit is read, the parity counter is incremented.

Concatenate bits into single variable:

```

//reset temp variables
parityCheck = 0;
tempOut = 0;
senderCount = 0;
for (int i=0; i<8; i++){ //concatenate bits into a number
    if (bits[i]){
        tempOut = tempOut | (0b1 << (7-i));
    }
}

for (int i=9; i<13; i++){ //Concatenate count into a number
    if (bits[i]){
        senderCount = senderCount | (0b1 << (3-i));
    }
}

```

This snippet of code converts the array of bits extracted in the previous snippet into actual numbers. The first one being the value read from the potentiometer on the transmit side, the second being the transmit counter.

Check parity:

```

if (parityCheck % 2 == 0){ //check if parity is correct

    sprintf(buffer, "signal good\r\n");
    HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);

}
else{ //check if parity is incorrect
    sprintf(buffer, "signal corrupted\r\n");
    HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);
}

```

The above code checks if data parity check is upheld. If not, the error is reported to the user via UART.

Compare counters:

```

if (receiverCount +1 == senderCount){
    receiverCount++;
    sprintf(buffer, "num sent = received\r\n");
    HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);
}
else {

    sprintf(buffer, "missed %d packet(s)\r\n", senderCount - (receiverCount+1));
    HAL_UART_Transmit(&huart2, buffer, sizeof(buffer), 1000);
    receiverCount = senderCount;
}

```

This snippet checks if the transmitted counter is the same as the receiver counter. If not, the receiver reports that packets have been missed.

5. Validation & Performance

The demonstration video along with the project files can be found in a separate google Drive with this link:
https://drive.google.com/drive/folders/1phgRF0HJMYpffcl-Q9tZm3j_LpRx9h54?usp=sharing

The following properties needed to be tested and validated to insure that the system operated at peak performance:



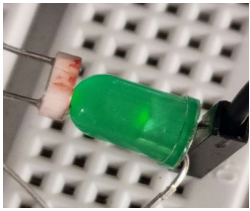
1. Compare LED colours
2. Threshold for LDR
3. Start condition length
4. Operating frequency
5. Possible operating distance.

Simple tests were run for the transmitter and receiver individually. This was done by slowing the communication speed down drastically, which allowed the team to inspect transmitter data visually, and manually input receiver data by turning on or off an LED by hand. Additionally, both modules made use of a USB to UART interface, which was used for trouble-shooting and verifying the correctness of the transmitted or received signal. Having verified that both of these systems worked individually, the system was integrated.

This allowed for a number of validation tests:

1. LED colour comparison

During the design process, there came a point where a choice of LED colour had to be made. Ideally the LED that shines the brightest and reduces the value of the LDR the most should be used. To test this, the LEDs were shone on the LDR for 1 minute and the lowest and highest values of the LDR were recorded. Three different colours of LEDs were used: Red, Blue and Green. This test was also tested with no LED lit to get a reading of the LDR when exposed to just ambient light.

Colour	Red LED	Blue LED	Green LED	Ambient Light
Picture				No Picture needed
Min Value	1972	2185	2769	2831
Max Value	2107	2321	2843	3274

After all the tests were completed and the data was recorded, it was clear that the red LED had the lowest value and is therefore the most suitable. Alternatively the Blue LED could be

used as well as it also significantly reduces the LDR value. However the same cannot be said about the Green LED which performed poorly compared to the other two LEDs. its max value also overlapped with the minimum value of the ambient light which would make choosing a possible threshold value impossible. Therefore the Red LED was chosen for the final system.

2. Logic level threshold value for receiver sensor

As mentioned previously, the receiving microcontroller reads an analog value from the receiver sensor. This allows the user to adapt the system specific to their operating environment. The logic level threshold mainly depends on two parameters: ambient light and brightness of the LED. Having chosen the best-working LED in the previous section, the sensor's analog value was measured. To get the least amount of error when receiving, the threshold should be set as close to this reading as possible. However, setting it too close might cause information to be lost, for example if the distance between transmitter or receiver changes, or if the supply voltages of transmitter or receiver fluctuate. With the red LED chosen in the previous section, the ADC read a value of roughly 2000.

To prevent false data from ambient light, the threshold should be set as low as possible. During the LED threshold test, the LDR was tested under only ambient light. On average this resulted in an LDR reading of 3000. To find a balance between noise tolerance and data integrity, a threshold of 2500 was chosen. Further testing revealed this to be an effective value for short distance communication. Note that using a laser diode instead of an LED would produce far better results, as the received light intensity would be much higher, and ambient light influences would thus be of far less concern.

3. Starting condition length

In previous sections, the length of the starting condition was decided on. Note however, that since some amount of timing inaccuracies are present, the expected start condition length should be slightly shorter than the actual on-time, but long enough so that a stream of data might not be interpreted wrongly. As discussed previously, the longest possible sequence of consecutive 1's is 8 clock cycles. Since the start condition is 10 clock cycles of 1, even if the receiver only checks if the input signal is longer than 9.5 clock cycles, this would be more than enough to unmistakingly detect a start condition. Note however that, in order not to interfere with any further timing, the receiver waits until the transmitted signal drops to low, before resuming. This is to ensure that the transmitter and receiver remain synchronised. Putting the above to the test, it was found that the above approach was able to detect start conditions very reliably.

4. Operating frequency

The maximum operating frequency is restricted by various parameters. Due to the code implementation, both transmitter and receiver have relatively inaccurate operating frequencies. Since the transmission speed was not specified, this should not be of much concern. Since inaccuracies are in the order of milliseconds, if the transmission and receiving frequencies are extended, this has a non-substantial effect. Additionally, the

transmitter and receiver are resynchronised before every packet of data, by listening to the falling edge of the start condition. This should help minimise any synchronisation issues, especially because the transmitted data is short.

Another issue that could potentially affect the maximum operating frequency, is the latency involved in turning on and off a diode, as well as the time response of the LDR sensor.

To assess the extent to which these factors influence transmission rate, transmit and receive frequencies were steadily increased, until reliable communication was no longer possible.

As an initial frequency, 1Hz was chosen, mainly due to ease of visual debugging. At this frequency, data transfer proved to be very accurate, with essentially no errors occurring. Having set a benchmark, the frequency was doubled numerous times. While 2 Hz and 4 Hz worked outstandingly, 8 Hz suffered from lost or misread data. This was validated using the USB to UART links. Since 4Hz proved to be reliable, it was chosen as the operating frequency.

5. Possible Operating Distance

The system was tested with varying distances between the LED and the LDR.

(0, 1, 2, 3, 4 cm)

The system output was perfect for the first 4 sets of tests and then failed when we ran it at the 4cm distance. This is due to the LDR not being able to consistently pick up the changes in the LED over the given distance. To overcome this limitation a stronger light source should be incorporated, possibly a form of laser.

6. Conclusion

The aim of this investigation was to assess the feasibility and implementation of a Light of Things (LOT) data transmission link. Through the requirements, detailed specifications were made allowing the device to be created. An ADC was sampled every time the push button on the microcontroller was pressed. This decimal value was then scaled in magnitude so that its binary equivalent could fit into 8 bits of data (max ADC value = 255). An asynchronous method was used as the devices could not be connected at all. Using an interrupt on the transmitter system, the receiver system could identify when a message was about to be sent. This was a 10 bit high followed by a 1 bit low. After the interrupt, the data message was sent, followed by a parity bit. Even parity was used and determined on the transmitter side. On the receiver side the parity bit was then checked to make sure the data it received was accurate. The number of samples sent needed to be tracked. This was done by incrementing a counter each time the push-button was pressed. The value was then converted into a binary 4 bit value and sent after the data message. These binary values were then converted back to decimal values and displayed on the receiver screen using putty.

The binary 1's (highs) and 0's (lows) were communicated on the transmitter side using a simple LED circuit as shown in Figure ... Using PA6 on the STM32F0 Discovery Board, the LED turned on when the pin was high and off when the pin was low. Each bit was separated using a 500ms delay. This delay was initially set at 1000ms. Through testing the best

performance was found to be 500ms as 250ms resulted in a small number of incorrect values when the experiment was repeated multiple times.

The most effective LED was found to be the Red LED as this produced the most ambient light, making the distance between the LED and LDR the greatest. The distance between the transmitter LED and receiver LDR was initially set to 1mm. Through testing, the maximum distance without substantial errors was found to be 30mm. When this value was increased past 30mm, inconsistencies between transmitter data and receiver data started to emerge.

Overall the experiment was a success. An ADC value could be accurately sampled, converted into binary and sent via an LED. A parity could be added and the number of samples could be attached to the binary message. The receiver could accurately sample the data and convert it back to decimal form as well as check the parity bit to ensure the accuracy of the data. The system was sampled at multiple distances and performed accurately until the 4cm mark. All the requirements were met and the client is happy.

This system accurately sends and receives a data message. The extent to which this system could be used is vast. The system could be designed with a more powerful way of sending and receiving light. This would allow the system to be used in any way limited only to line of sight. This extends to using a satellite as a means of communication. With a powerful enough laser, a beam could be sent to a satellite and reflected down to a receiver anywhere in the world. This system would still accurately transmit and receive a short data message. The system could also possibly be in a war time setting where transmission security is paramount. The system could use an infra-red laser making it invisible to the naked eye. This would ensure the absolute security of the transmission as the only way to intercept this message would be to precisely position some form of receiver in the path of the laser prior to the transmission. If the system was set up with both a transmitter and receiver on both ends the transmission would be completely secure as the transmitter on the receiver side could respond with acknowledged bits meaning that if something interrupts the line of sight the signal will stop transmitting.