

Harry Woollands - DIS Assignment

by Harry Woollands

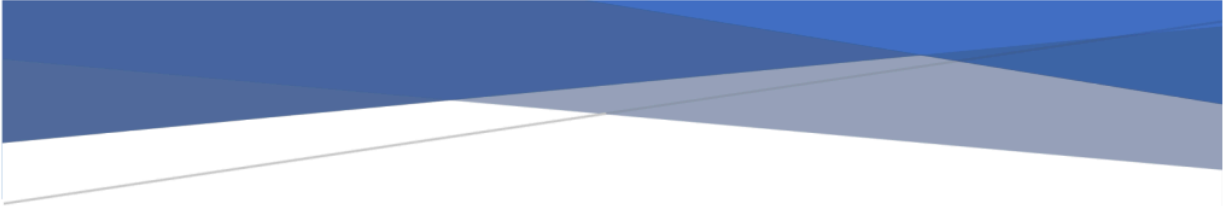
Submission date: 12-Nov-2024 09:18AM (UTC+0000)

Submission ID: 242391560

File name: 42453_Harry_Woollands_Harry_Woollands_-_DIS_Assignment_523660_1845982634.docx
(1,021.66K)

Word count: 3063

Character count: 15250



A SIMPLE TOKEN BASED DISTRUBUTED NETWORK

Distributed Intelligence Systems

2647 WORDS

Abstract

This report details the design and development of a simple token based distributed network, using two PIC Microcontrollers, acting as master and slave PIC's. The system is designed using Proteus simulation software, and coded in Embedded C. The system operates LED's to operate lights, mimicking one corner of a vehicle. The system uses I2C communication protocol, to send and receive data to one another, and to external memory.

Harry Woollands

2214706

Contents

Table of Figures.....	2
Introduction	3
Task 1	4
Task 1a	4
Task 1b	4
Point to Point Transmission System	4
Point to Multipoint Transmission System.....	4
Multi-master Transmission System	4
Task 2	6
The Inputs	6
The Outputs	7
The Master-Slave	8
Task 3	10
Master.....	10
Slave	14
Task 4	18
Results.....	18
Conclusion.....	20
References	21

Table of Figures

Figure 1. Complete Proteus Design.....	6
Figure 2. Button Inputs	7
Figure 3. LED Outputs	8
Figure 4. Master-Slave main function	9
Figure 5. Declaring functions and variables for master PIC	10
Figure 6. read_buttons routine.....	10
Figure 7. Memory write routine	10
Figure 8. Memory read routine.....	11
Figure 9. Switch case.....	11
Figure 10. Case 1	12
Figure 11. Cases 2,4,8 and 16	13
Figure 12. Declaring functions and variables for slave PIC	14
Figure 13. memory_write and memory_read functions	14
Figure 14. Indicator flash routine.....	15
Figure 15. Timer routine	15
Figure 16. Button press loop.....	16
Figure 17. Case A.....	16
Figure 18. Cases B, C, D and E	17
Figure 19. Flowchart	18
Figure 20. LED's on.....	19
Figure 21. LED's off.....	20

Introduction

In the automotive industry, lights and indicators are fundamental components of vehicles, ensuring the safety of all road users. This assignment largely focuses on the design and development of a communications system that operates lights and indicators for a vehicle.

This report begins with a brief description of quality management systems, and transmission systems, including point to point transmission, point to multipoint transmission, and multi-master transmission. Next, the design and development of a simple token-based distribution network for an automotive application will be detailed, developed in *Proteus Simulation Software*. The system uses a pair of PIC18F2525 Microcontrollers, which supplies signals to operate a set of LED's. These LED's represent the lights found on a single corner of a vehicle, and will be manually operated with the push of a push button. The code written to design this system will be shown, with an explanation of each section following that. Alongside the design and operation of the system, a flowchart will be included, illustrating the process, showing the interaction between the devices.

Task 1

Task 1 is a research task, that will explain the need for quality management systems to ensure standards, including what it is, what it is used for, and why it is needed. It will then describe different types of transmission systems, including; a point to point transmission system, a point to multipoint transmission system, and a multi-master transmission system. These transmission types are fundamental in telecommunications.

Task 1a

A Quality Management System (QMS) is defined as a set of processes and responsibilities that ensure an organisation can operate as it is supposed to [1]. Alongside this, QMS is used to continuously improve and further develop operation and influence safety standards. To achieve this, organisations frequently conduct reviews of their procedures, and refine them in order to produce higher quality products, more efficiently.

QMS is needed because it is important to be able to constantly adapt in order to meet changing customer requirements. The existence of a QMS ensures that an organisation can continue to deliver for a customer, being able to integrate new and adapt existing processes. A QMS is also needed because it allows for processes to be applied in the same way, resulting in more reliable and consistent products, resulting in greater customer satisfaction.

QMS is also used to allow organisations to comply with any set of required regulations enforced upon them, such as certain standards or obligations. These allows for the organisation to operate safely, whilst simultaneously delivering high quality products.

Task 1b

Point to Point Transmission System

A point to point transmission system is a form of data connection that links two nodes together [2], such as a transmitter and receiver, one node being a start point, and the other being an endpoint. Data is transmitted between these two nodes, which can be wired together, or it can be transmitted wirelessly. The pros of using a point to point system is that it can transmit data fast and securely. An example of a point to point system are cellular networks, where it is used to connect cell towers to the base network infrastructure [2]. This allows for large amounts of data to be transferred between the cell towers and the base network infrastructure. The main downside with point to point transmission is that it can only reach one node at a time, a problem that point to multipoint solves.

Point to Multipoint Transmission System

Point to multipoint transmission is a system where one central node can transmit data to multiple other nodes at the same time [3]. The biggest advantage of using a point to multipoint system is its efficiency. As a single transmitter node can send data to multiple receiver nodes at the same time, the time taken to transmit all of the data is reduced. Point to multipoint systems become more efficient the larger the network is. A common example of a point to multipoint transmission system is a Wi-Fi router, which is the central node that transmits data to multiple devices that are connected on the network.

Multi-master Transmission System

A multi-master transmission system is a system where multiple nodes, that are labelled 'masters', can transmit data at the same time to one or more other nodes, that are labelled 'slaves' [4]. The biggest advantage of using a multi-master system is that any device that any 'master' device can transmit to a 'slave' device, rather than the one transmitter node used in point to point and point to

multipoint transmission. An example of a multi-master transmission system is a Controller Area Network (CAN), which can be found in automobiles. In automobiles, multiple 'masters' communicate with multiple 'slaves', such as different sensors which are constantly monitoring the vehicle.

Task 2

Task 2 was to design the master-slave system. In this task, each section of the Proteus design will be detailed. The system will be split into three main sections, the inputs, the outputs, and the master-slave section. The brief states to only design the system for one side of the vehicle. For this assignment, the rear corner was chosen. Figure 1 below shows the completed Proteus design.

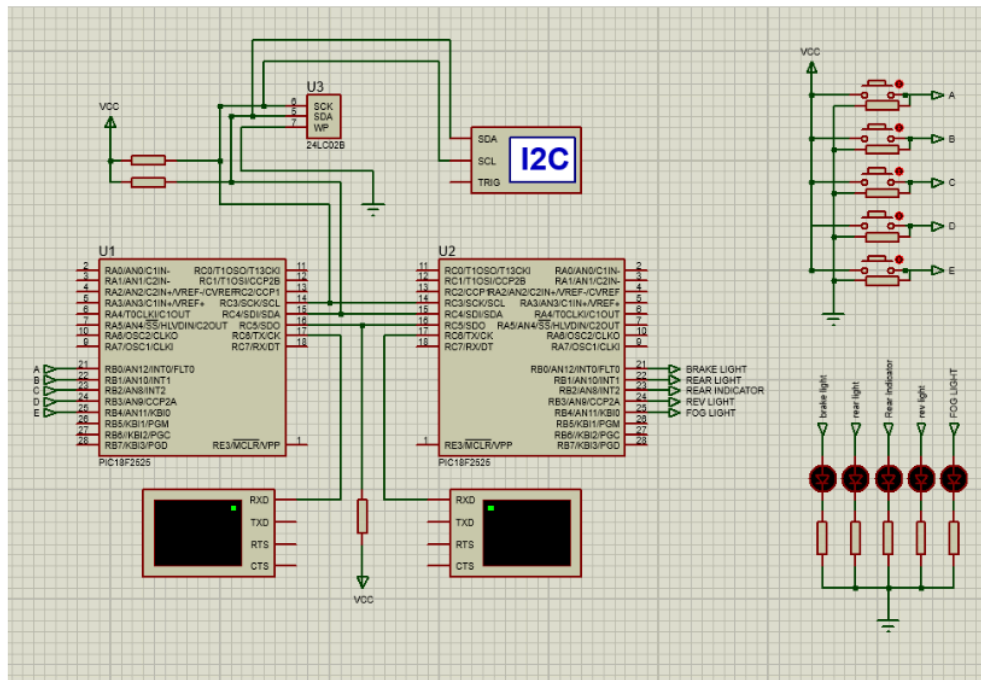


Figure 1. Complete Proteus Design

The Inputs

The first section of the Proteus Schematic is the inputs. These consist of five push buttons that are connected to both power and ground. The five buttons are given output labels 'A' through to 'E'. Pressing these buttons will act as a person driving the vehicle. Figure 2 below shows the input section of the schematic.

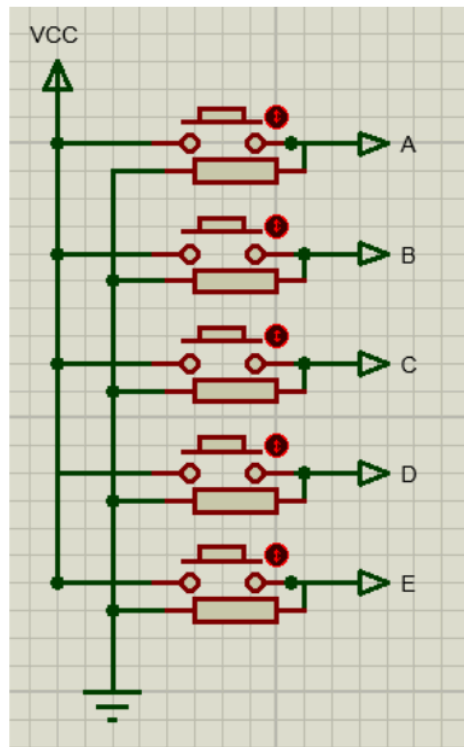


Figure 2. Button Inputs

The Outputs

The next section of the Proteus schematic is the outputs. These consist of five LED's, connected to ground. The five LED's represent the brake lights, rear lights, rear indicator, reversing light, and fog light, and are each given their respective input labels. To make the design as realistic as possible, the rear and brake lights use red LED's the indicator uses a yellow LED, and the reversing light and fog light use white LED's. Figure 3 below shows the outputs section of the schematic.

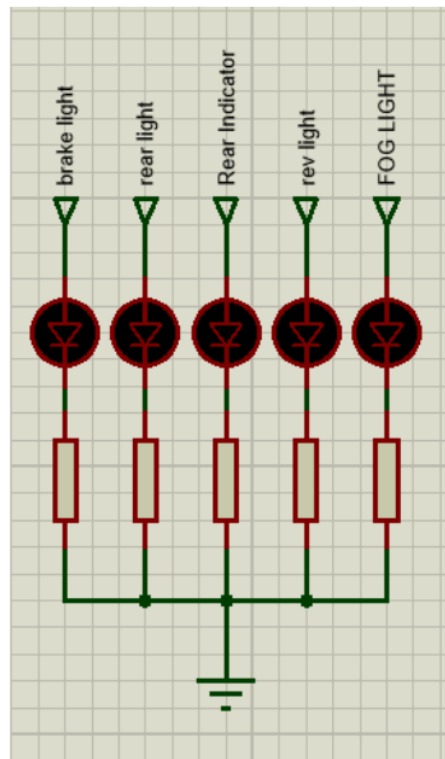


Figure 3. LED Outputs

The Master-Slave

The final section of the Proteus schematic is the master-slave section. This section mainly consists of two PIC18F2525's. The PIC18F2525 is ideal for this assignment due to the number of ports it has. This means it has plenty of space for all of the buttons and LED's to be connected, as well as adequate space to add more components to make the design more complicated if necessary. Located on Port B on the first PIC is the input labels 'A' through to 'E', which correspond to the outputs on the push buttons. Then, located on Port B on the second PIC are the output labels of the LED's. Having these labels both on Port B on their respective PIC's allows for direct communication between the buttons and the LED's.

Next, connected to both slave and master PIC's is I2C serial memory, and an I2C debugger. I2C is a communication protocol that uses a serial data line and a serial clock line [5]. Finally, a virtual terminal is connected to each of the PIC's. These virtual terminals are used to visualise the tokens that are sent and received every time a button is pressed. Figure 4 below shows the master-slave section of the schematic.

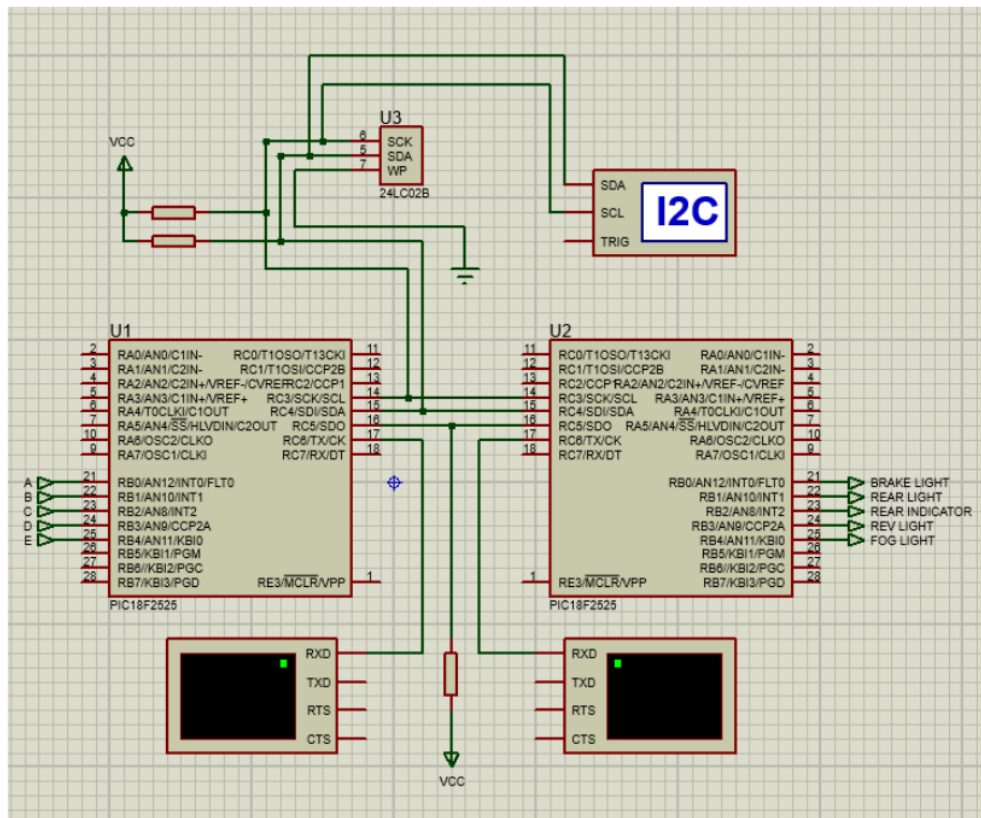


Figure 4. Master-Slave main function

Task 3

Task 3 will detail the code written to develop the programs on the master PIC and the slave PIC. It will break down and explain the key points for each of the programs.

Master

Before starting to develop the program, the backbones of the code had to be set out. This included defining variables and routines that would be used later during the tasks. Figure 5 below shows the declared variables and routines that will be used throughout the code. These are declared using the 'void' function, and the 'char' function. 'Void' is used when declaring generic functions, and 'char' is used when returning a character.

```
void memory_write(char, char);           // Function to write data to the memory location
char read_buttons(void);                 // Function to read the value of the buttons
char memory_read(char);                  // Function to read data from the memory location
char local_output;                       // Variable that stores the current status of the outputs
char txbyte;                             // Variable used to store the token character
```

Figure 5. Declaring functions and variables for master PIC

The next stage in the development of the master code was to create the read_buttons routine. The purpose of this routine is to monitor if a button is being pressed at a given moment. If a button is being pressed, the value of that button is given to the 'value' variable. This value is finally returned, and will be used later on in the program. Figure 6 below shows this section of code.

```
char read_buttons(void)
{
    char value = 0;                       // Variable to hold the button press state
    while (!input_b());                   // Wait for a button press
    value = input_b();                     // Store the value of Port B to the variable 'value'
    delay_ms(500);                         // Delay to allow button release
    return value;                          // Return the value of the buttons pressed
}
```

Figure 6. read_buttons routine

The next routine that was written in the development of the code is the memory_write routine. The purpose of this routine is to write data to the I2C memory. In this routine, the characters 'where' and 'what' are defined. The 'where' character is used to store a specific I2C memory address, and the 'what' character is used to store the data. Figure 7 below shows this section of code.

```
void memory_write(char where, char what)
{
    while (!input(hs));                   // Wait for the memory chip to be available
    output_low(hs);                        // Set the hs line low to start communication
    i2c_start();                           // Start I2C communication
    i2c_write(0xA0);                       // Send the I2C address for write operation
    i2c_write(where);                      // Write the data to the memory
    delay_ms(20);
    i2c_stop();                           // Stop I2C communication
    delay_ms(20);
    input(hs);
}
```

Figure 7. Memory write routine

After creating the routine that can write data, the next routine that was written in the development of the code is the memory_read routine. The purpose of this routine is to read data from the I2C memory, that has already been written to it. In this routine, two different memory addresses are written to. The reason for this is that the first memory address is specified so that the data can be written to that address, and the second address is specified so that when the data needs to be read from the memory, it knows where to read it from. Once the memory address that the data is stored in is identified, the data stored is stored in the variable 'data'. Finally, the variable 'data' is returned. Figure 8 below shows this section of code. This routine is also the final routine written outside of void main.

```
char memory_read(char where)
{
    char data = 0;

    while (input(hs));           // Wait for the memory chip to be available
    output_low(hs);              // Set the hs line low to start communication
    i2c_start();                 // Start I2C communication
    i2c_write(0xA0);             // Send the I2C address for write operation
    i2c_write(where);
    delay_ms(20);
    i2c_start();
    delay_ms(20);
    i2c_write(0xA1);
    data = i2c_read(0);          // Read the data from the memory
    delay_ms(20);
    i2c_stop();                  // Stop I2C communication
    delay_ms(20);
    input(hs);

    return data;
}
```

Figure 8. Memory read routine

Now that the routines have all be created, the next step was to set up a while true loop, a function that allows different sections of the code to run depending on which button is pressed. Inside of this loop, the variable 'buts' is declared as the read_buttons routine. Next, a switch case is used, allowing the variable 'buts' to be tested against the button presses. Figure 9 below shows this section of code.

```
void main()
{
    char buts = 0;               // Variable to hold the current state of the buttons

    while (TRUE)                 // While True Loop
    {
        buts = read_buttons();   // Read the value of buttons pressed and store it in the variable "buts"

        switch (buts)
```

Figure 9. Switch case

Within the switch case, are the five different sections of code, each section relating to one of the buttons. Case 1 is shown below in Figure 10, and relates to the brake light. The if statement checks whether the brake light is currently on. If the light is on, then 1 is subtracted from local_output, turning the brake light off, whilst setting the variable 'txbyte' to 'a'. If the light is off, then 1 is added to local_output, turning the brake light on, whilst setting the variable 'txbyte' to 'A'. These txbyte values are visually represented in the virtual terminal.

```

case 1:
    if (local_output & 0x01)           // Check if the brake light is on
    {
        local_output -= 1;             // Turn off the brake light by subtracting 1 from local_output
        txbyte = 'a';                 // Set the token byte to 'a'
    }
    else
    {
        local_output += 1;             // Turn on the brake light by adding 1 to local_output
        txbyte = 'A';                 // Set the token byte to 'A'
    }
    break;

```

Figure 10. Case 1

The final four cases are very similar to case 1, the difference being which hexadecimal value is used to check whether the LED is on or not. As a result, the values added and subtracted to turn the LED's on and off are also correspondingly changed. Case 2 is used to turn off and on the rear lights, case 4 is used to turn off and on the indicator, case 8 is used to turn off and on the reversing light, and finally case 16 is used to turn off and on the fog light. Figure 11 below shows the remaining cases.

```

case 2:
    if (local_output & 0x02)           // Check if the rear Lights are on
    {
        local_output -= 2;           // Turn off the Lights by subtracting 2 from Local_output
        txbyte = 'b';               // Set the token byte to 'b'
    }
    else
    {
        local_output += 2;           // Turn on the Lights by adding 2 to Local_output
        txbyte = 'B';               // Set the token byte to 'B'
    }
    break;

case 4:
    if (local_output & 0x04)           // Check if the rear indicator is on
    {
        local_output -= 4;           // Turn off the rear indicator by subtracting 4 from Local_output
        txbyte = 'c';               // Set the token byte to 'c'
    }
    else
    {
        local_output += 4;           // Turn on the rear indicator by adding 4 to Local_output.
        txbyte = 'C';               // Set the token byte to 'C'
    }
    break;

case 8:
    if (local_output & 0x08)           // Check if the reverse Light is on
    {
        local_output -= 8;           // Turn off the rerverse Light by subtracting 8 from Local_output
        txbyte = 'd';               // Set the token byte to 'd'
    }
    else
    {
        local_output += 8;           // Turn on the rerverse Light by adding 8 to Local_output.
        txbyte = 'D';               // Set the token byte to 'D'
    }
    break;

case 16:
    if (local_output & 0x10)           // Check if the fog Lights are on
    {
        local_output -= 0x10;        // Turn off the fog Lights by subtracting 0x10 from Local_output
        txbyte = 'e';               // Set the token byte to 'e'
    }
    else
    {
        local_output += 0x10;        // Turn on the fog Lights by adding 0x10 to Local_output
        txbyte = 'E';               // Set the token byte to 'E'
    }
    break;
default:
    break;

```

Figure 11. Cases 2,4,8 and 16

Slave

As with the master PIC code, before beginning development the variables and routines needed to be defined. Some routines such as `local_output`, `memory_read` and `memory_write` are familiar as they were used in the master PIC code, and new variables and routines have been added. Figure 12 below shows the functions and routines used in the slave PIC code.

```
char local_output = 0;           // Variable that stores the current status of the outputs
char control_byte = 0;          // Variable that stores the state of lights
unsigned int counter = 0;        // Counter
unsigned int flash = 0;          // Flashing state (used for blinkers)
unsigned int ind = 0;            // State of the left indicator (turn signal)
char buts = 0;                  // Variable to store the button press
char old_but = 0;               // Variable to store the previous button press state for comparison
char memory_read(char);         // Function to read data from memory
void memory_write(char where, char what); // Function to write data to memory
void flash_control(void);        // Function to control flashing (e.g., blinkers)
```

Figure 12. Declaring functions and variables for slave PIC

As aforementioned, the `memory_read` and `memory_write` routines are included in both master and slave PIC code. This is because both the master and the slave need to read and write to the I2C memory. Therefore, the code in the slave PIC for these sections is identical to the code in the master PIC. Figure 13 below shows the code for these two routines.

```
void memory_write(char where, char what)
{
    while (!input(hs));          // Wait for the memory chip to be available
    output_low(hs);              // Set the hs line low to start communication
    i2c_start();                 // Start I2C communication
    i2c_write(0xA0);             // Send the I2C address for write operation
    i2c_write(where);            // Write the data to the memory
    i2c_write(what);
    delay_ms(20);
    i2c_stop();                 // Stop I2C communication
    delay_ms(20);
    input(hs);
}

char memory_read(char where)
{
    char data = 0;

    while (!input(hs));          // Wait for the memory chip to be available
    output_low(hs);              // Set the hs line low to start communication
    i2c_start();                 // Start I2C communication
    i2c_write(0xA0);             // Send the I2C address for write operation
    i2c_write(where);
    delay_ms(20);
    i2c_start();
    delay_ms(20);
    i2c_write(0xA1);             // Read the data from the memory
    data = i2c_read(0);
    delay_ms(20);
    i2c_stop();                 // Stop I2C communication
    delay_ms(20);
    input(hs);

    return data;
}
```

Figure 13. `memory_write` and `memory_read` functions

The next section of the code is a routine not previously seen in the master PIC code. This routine is used to blink the indicator LED. If the variables 'ind' and 'flash' are both true, then the pin with the indicator LED on is set to high. If one or both of the variables are not true, then the pin is set to low. Figure 14 below shows this section of code.

```
void flash_control(void)
{
    if (ind & flash)
        output_high(pin_b2); // If left indicator is set pin_b2 high
    else
        output_low(pin_b2); // Else set pin_b2 low
}
```

Figure 14. Indicator flash routine

To set the variable 'flash' to true or false, the next routine, a timer routine is required. This uses the counter variable that was previously declared, and increments it until it reaches 40. When it reaches 40, the counter is reset, and sets the variable flash to true or false, depending on its current state. The flash_control routine above is then called. This enables the indicator LED to continuously turn on and off, until the push button manually stops the loop. Figure 15 below shows this routine, the final routine outside of the main loop.

```
void TIMER2_isr(void)
{
    counter++; // Increment the counter

    if (counter == 40)
    {
        counter = 0; // Reset the counter
        if (flash)
            flash = 0; // If flash is on, turn it off
        else
            flash = 1; // If flash is off, turn it on
        flash_control();
    }
}
```

Figure 15. Timer routine

At the start of the main routine, the I2C memory is written to and read from. The variable 'buts' is set to memory address of the button currently being pressed, and the variable 'old_but' is set to the memory address of the previous button pressed. If the address of the previous and current button presses is different, then the current button press is written to the old_but address. This ensures that this section of code is run every time a different button is pressed. Figure 16 below shows this section of code.

```

void main()
{
    setup_timer_2(T2_DIV_BY_16, 249, 15);
    enable_interrupts(INT_TIMER2);
    enable_interrupts(GLOBAL);

    while (true)
    {
        buts = memory_read(0x00);           // Read the button state from memory (address 0x00)

        if (old_but != buts)
        {
            printf("%c", buts);             // Print the buts variable value
            memory_write(0x10, buts);       // Add the buts variable value to memory address 0x10
            delay_ms(20);
        }
    }
}

```

Figure 16. Button press loop

After this, a switch case is used, allowing the variable 'buts' to be tested against the button presses. Within the switch case, are the five different sections of code, each section relating to one of the buttons. Case A is shown below in Figure 17, and relates to the brake light. When the button is pressed and the LED is off, 1 is added to the local_output and control_byte using the hexadecimal value 0x01, turning on the LED. When the button is pressed again, 1 is taken away from the local_output and control_byte, using the hexadecimal value 0xFE, returning both variables back to zero. Figure 17 below shows this section of code.

```

switch (buts)
{
    case 'A':
        local_output += 0x01;           // Turn on brake light
        control_byte += 0x01;
        break;

    case 'a':
        local_output = local_output & 0xFE; // Turn off brake light
        control_byte = control_byte & 0xFE;
        break;
}

```

Figure 17. Case A

Cases B, C, D and E are very similar to Case A, with the exception of the hexadecimal values added and subtracted. Case C is slightly different, with the variable 'ind' set to high or low, depending on whether the indicator light is on or off. This variable is seen in the indicator flash routine seen previously in the report. Figure 18 below shows the code for the remaining cases.

```

case 'B':
    local_output += 0x02;           // Turn on rear light
    control_byte += 0x02;
    break;

case 'b':
    local_output = local_output & 0xFD; // Turn off rear light
    control_byte = control_byte & 0xFD;
    break;

case 'C':
    local_output += 0x04;           // Turn on indicator
    control_byte += 0x04;
    ind = 1;                        // Set indicator variable to on
    break;

case 'c':
    local_output = local_output & 0xFB; // Turn off indicator
    control_byte = control_byte & 0xFB;
    ind = 0;                        // Set indicator variable to on
    break;

case 'D':
    local_output += 0x08;           // Turn on reverse light
    control_byte += 0x08;
    break;

case 'd':
    local_output = local_output & 0xF6; // Turn off reverse light
    control_byte = control_byte & 0xF6;
    break;

case 'E':
    local_output += 0x10;           // Turn on fog light
    control_byte += 0x10;
    break;

case 'e': // Button 'e' pressed
    local_output = local_output & 0xEE; // Turn off fog light
    control_byte = control_byte & 0xEE;
    break;

```

Figure 18. Cases B, C, D and E

Task 4

Task 4 was to design an operational flowchart to show the interaction between the devices. Figure 19 below shows the flowchart.

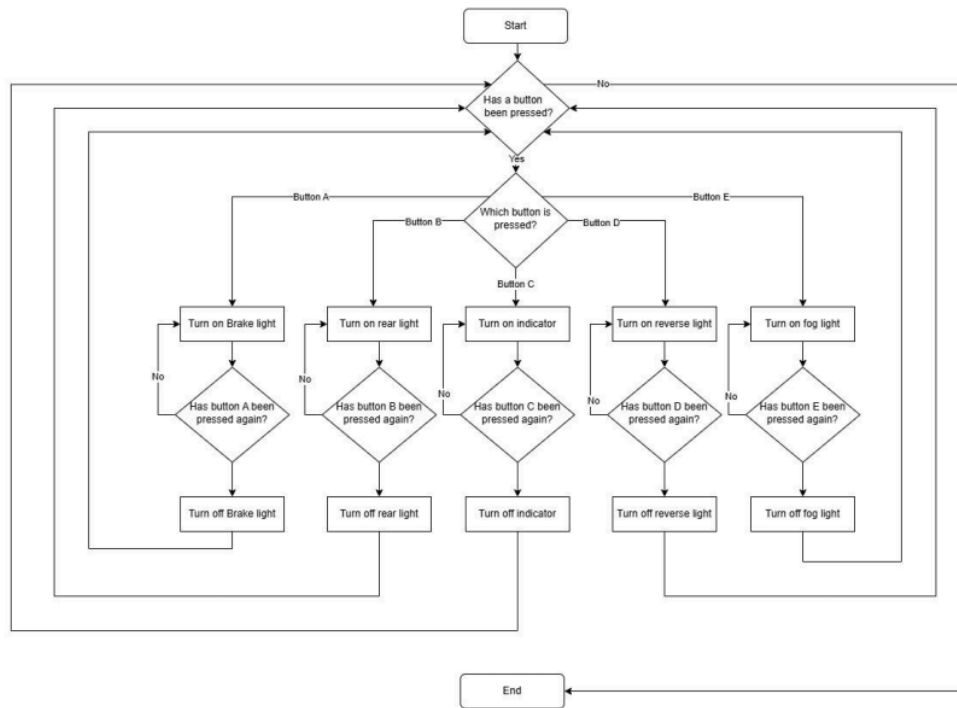


Figure 19. Flowchart

Results

This section of the report will include screenshots of my simulation in action in order to prove that the programs are working as intended.

Figure 20 below shows my system with all of the LED's set to high. The capital letters displayed on the virtual terminal indicate that the buttons have each been pressed once, after starting with the LED's off.

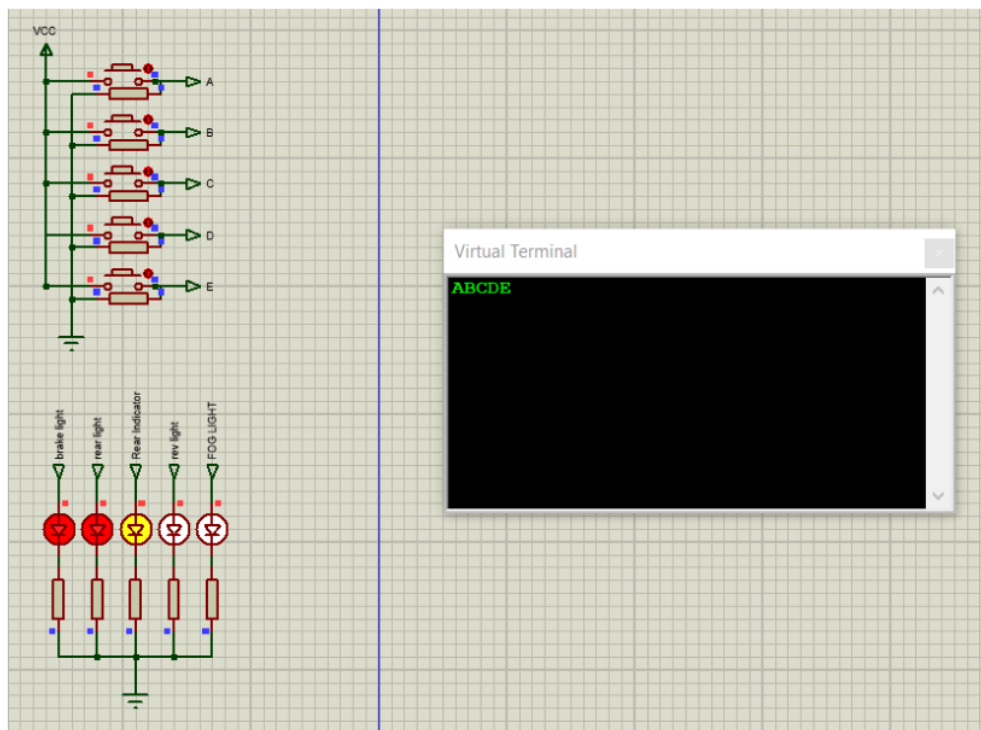


Figure 20. LED's on

Figure 21 below shows my system once all of the LED's have been turned off. The lowercase letters displayed on the virtual terminal after the capital letters indicate that the buttons have each been pressed once again, resetting the LED's back to their original off state.

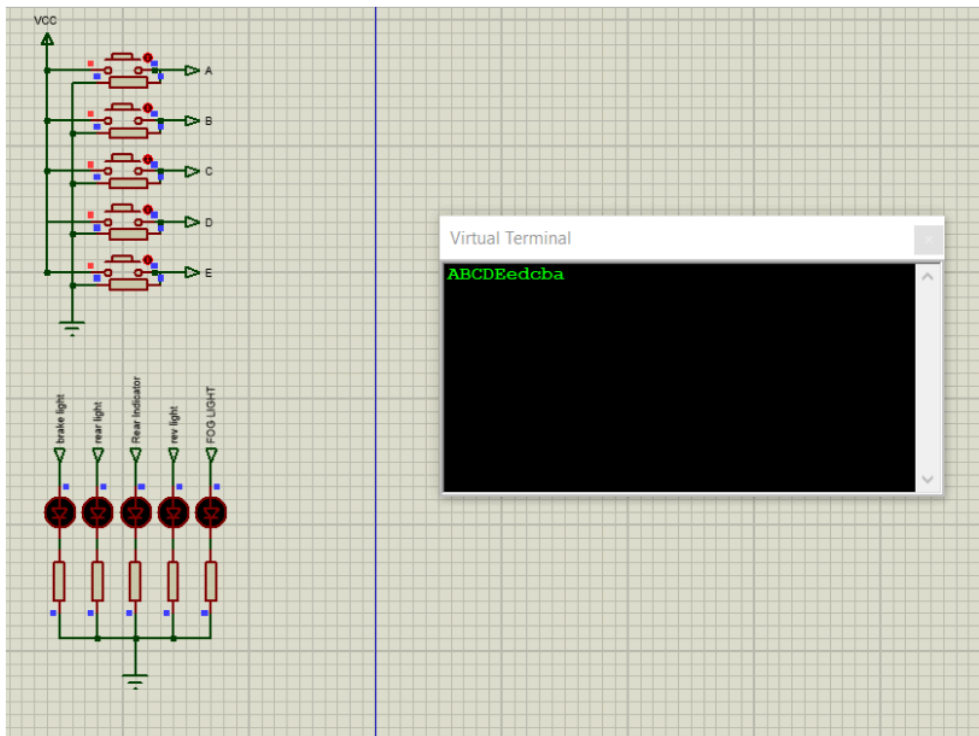


Figure 21. LED's off

Conclusion

In conclusion, this report has demonstrated an example of a simple token based distributed network, using master to slave transmission, utilising Proteus simulation software and Embedded C code. It has been brought to life through the visualisation of LED car lights, as well as the use of a tokens displayed on a virtual terminal, transmitted and received using I2C communication.

One bug that exists within the code is that on occasion, when the simulation is first ran, a lowercase 'e' letter appears on the virtual terminal without pressing any button. Whilst it is irritating, it does not impact on the functionality of the simulation, and could be fixed with further development of the software.

References

- [1] International Organization for Standardization , "Quality management systems: An introduction," ISO. <https://www.iso.org/quality-management/what-is-qms>
- [2] A. Chen, "Point to Point Communication Comprehensive Guide," *Sanny Telecom*, Apr. 27, 2024. <https://www.sannytelecom.com/point-to-point-communication-comprehensive-guide/>
- [3] L. MacGillivray, "The Benefits of Point to Multipoint Connectivity in Communications," *www.chicomm.com*, Aug. 22, 2024. <https://www.chicomm.com/blog/the-benefits-of-point-to-multipoint-connectivity-in-communications>
- [4] Microchip Technology, "25.7 Multi-master Systems and Arbitration," *Microchip.com*, 2022. <https://onlinedocs.microchip.com/oxy/GUID-80B1922D-872B-40C8-A8A5-0CBE009FD908-en-US-3/GUID-6AE16474-5C34-41C5-AA23-9C5519D30FDF.html>
- [5] J. Wu, "A Basic Guide to I 2 C," Nov. 2022. Accessed: Nov. 05, 2024. [Online]. Available: <https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1730786065980>

Harry Woollands - DIS Assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

71 /100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

GRADING FORM: SMU GENERIC_GJ

HARRY WOOLLANDS

71.45

RESEARCH-INFORMED LITERATURE



You have explained what point to point and point to multipoint and multimaster are correctly. 68%

13.6

KNOWLEDGE AND UNDERSTANDING



The design of your circuit is suitable for the task 75%

18.75

ANALYSIS



You have completed the programs for both devices and the circuit works well. 70%

17.5

PRACTICAL APPLICATION AND DEP.

 Your explanation is good and acknowledges that both pics speak to memory IC 72%

10.8

SKILLS FOR PROFESSIONAL PRAC.

 The report is well presented with sections to explain the code and include a limited number of references, and a good flowchart. 72%

10.8