

Henry Assignment DIS

by Henry Rennison

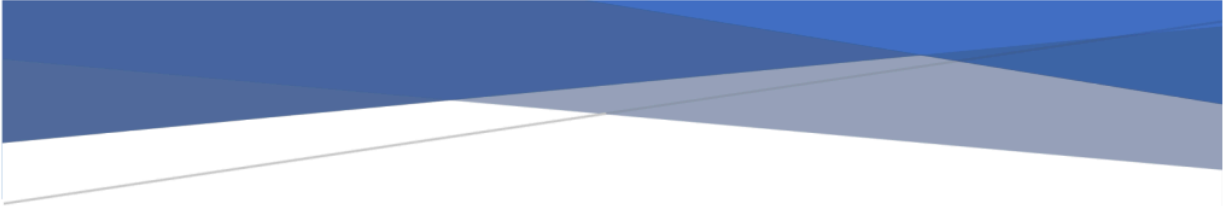
Submission date: 13-Nov-2024 05:27PM (UTC+0000)

Submission ID: 242564709

File name: 42283_Henry_Rennison_Henry_Assignment_DIS_523660_463204779.docx (639.42K)

Word count: 2559

Character count: 12745



SIMPLE TOKEN BASED DISTRIBUTED NETWORK

Distributed Intelligent systems

Abstract

This report covers three of the main data transmission types. It also covers the design and creation of a simple token-based system to replicate that of the back-right corner of a car. This was completed using two microcontrollers with one as the master and the other as a slave. The system will be created and designed using proteus. The design utilizes I2C protocol.

Henry Rennison (2218326)

Contents

Table of figures 2

Introduction 3

Research (Task 1a) 3

Research (Task 1b) 3

Task 2 – Designing the system on proteus..... 4

Analysis/ Results 12

Conclusion..... 13

Bibliography 14

Table of figures

Figure 1- Point to multi point	4
Figure 2 Proteus Design	5
Figure 3 Master Variables and routines.....	6
Figure 4 Read buttons Function (Master)	7
Figure 5 void main (master 1)	8
Figure 6 void main (master 2)	8
Figure 7 virtual terminals	9
Figure 8 read write functions (Master)	9
Figure 9 Global variables (Slave)	10
Figure 10 Timer (slave).....	10
Figure 11 void main (Slave)	11
Figure 12 payload function (Slave)	11
Figure 13 Memory (Slave)	12
Figure 14 flow chart	12
Figure 15 Leds and virtual terminal	13

Introduction

This report will involve some research tasks looking into and giving definitions of three different transmission types: point to point transmission, point to multi point transmission and multi-master transmission. The second part of this essay will be designing a system on proteus using a one pic as a master and one as slave. The system is meant to imitate the back-right corner of a car. The choice of pic and certain components will also be covered.

Research (Task 1a)

Quality management system

Quality management systems are used for maintaining and ensuring continuous improvement within companies and large organisations. It helps to ensure that products and services meet the required quality levels, this can help to build a better reputation for the company, it can also enhance the overall quality of the service or product. Another focus of QMS is to streamline processes which can in turn improve efficiency, lower costs and potentially reduce waste.

One of the other very important roles of QMS is to promote and advise on safety standards. This is done by following simple guidelines that are already established. QMS does this by regularly assessing organisations. If any issues are found they suggest methods in order to fix them. Ultimately the point of this is to reduce the chance of accidents and try to promote safer working conditions.

QMS in communication is vital in reducing errors and troubleshooting. QMS includes protocols that are made to identify, analyse and correct things like data loss or signal degradation. It does this using standardised troubleshooting methods. [1]

Research (Task 1b)

Point to Point transmission system

A point to point transmission is a communication method in which the system has one designated transmitter and one designated receiver. The receiver and transmitter have direct connection with each other, this can be through wireless, copper cable and fiber optic cable. It is widely used in telecommunications. Some very common examples of a point to point transmission are radio links, phone calls or a dedicated data link between two locations (wireless network beams) [2].

Point to Multi Point system

A point to multi point system is a communication method in which there is one master/ transmitter and multiple receivers that connect to the transmitter. Common examples of this type of system are WIFI solutions, television/radio broadcasts and data distribution in cellular networks. Point to multi point systems usually work using a shared channel to reach multiple receivers. [3]

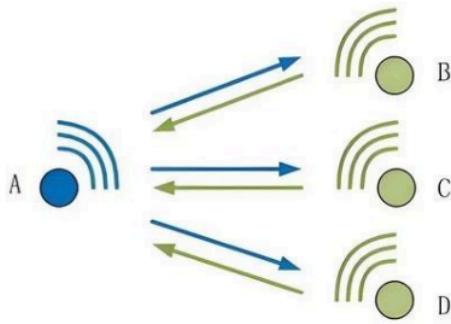


Figure 1- Point to multi point

Multi-Master Transmission system

A multi-master transmission system is a communication method in which multiple devices (masters) can send data within the same network. This allows for a much more flexible non-centralised communication system. Multi Master systems allow any device to send and receive data. Common uses of multi-master system are: Smartphones and computer components, they use the I²C protocol which allows multi-master communication among peripherals. This allows components to communicate with each other and operate without relying on a central processor. [4]

For this Transmission system to work effectively it uses I²C. I²C allows multiple devices to communicate with each other using only two wires: SCL (Serial Clock Line) and SDA (Serial Data Line). SCL is used to carry the clock signal from the master device. SDA is used for bidirectional data transfer between devices. I²C clock can be controlled by multiple master devices and can send to multiple slave devices. [5]

Task 2 – Designing the system on proteus

The next task this report will cover is the designing of the system proteus, it will include descriptions as to why the PICs and other components were selected. It will also give a brief overview of how the system is going to work. The circuit design created was meant to represent the back-right corner of a car.

The first thing to do when designing the system on proteus as to decide which PIC to use. The PIC used was the PIC18F2525, this was because it has an 8-bit core with an enhanced instruction set meaning it can handle more complex tasks more efficiently than a standard 8 bit PIC. The PIC18F2525 also has 25 general purpose I/O ports which will be more than enough for this design. Another key point when selecting this PIC was its availability and price point, this will make it much easier to get hold of if the proteus design was ever to be created using physical hardware. [6]

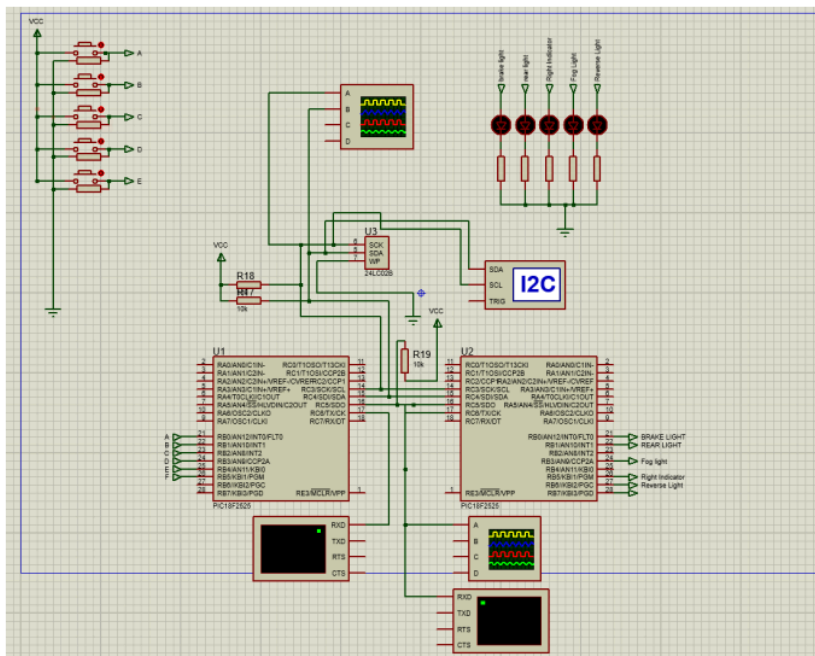


Figure 2 Proteus Design

As shown in the image above the design consists of a PIC on the left which is the master and a PIC on the right is the slave. It also has five push buttons, five LEDs it also consists of a I2C clock which when used in proteus it needs an additional chip (24LC02B) to work correctly.

Button A – Brake Light

Button B – Rear Light

Button C – Right Indicator

Button D – Fog Light

Button E – Reverse Light

The programme works by pressing one of the buttons on the top left which correlate with the lights in the top right.

Once the button has been pressed it sends the data through the I2C clock then to the slave unit and corresponding light to show that it is on. There is also a virtual terminal representing what the Master PIC is sending and the Slave PIC is receiving. This is a visual aid to help the user what buttons have been pressed and ensuring that what is being sent is the same as what is being received.

Task 3

This part of the report will be covering the code used to run the programme breaking down the code in both the master PIC and the Slave PIC.

Master

The first steps were to set out the variables and routines for the master PIC:

```
#include <main.h>
// master

void ep_w(char,char); // Writing to the I2C eeprom memory
char read_buttons(void); //function that returns what button has been pressed
char ep_r(char ); // Reading from the I2C eeprom memory

char local_output; // Variable to store the current output state
char txbyte; // Variable for transmitting data
unsigned int value=0x00;
//unsianed int value=0xff;
```

Figure 3 Master Variables and routines

Above in figure 3 is where all the routines and variables are declared in the master. The “Void” keyword is used to define routines and send the data to the memory. The “char” key word at the beginning of the function definition indicates that ep_r returns a single “char” value. In this Case it is taking the read data from the I2C memory as a single value as an individual byte of data.

This image above also shows the “read_buttons” function being defined. This is the function that will look to read data as to when one of the buttons are pressed.

The next part of the code in figure 4 was to define the function called “read buttons” that returns “char” value and takes no arguments. This return type is used for a single byte of data that represent the state of each of the buttons in the circuit. The next line of code “char value=0” indicates a variable named value to 0, this variable stores the state of port B when the buttons are pressed.

The “while(!input_b()); creates a loop that will continuously look for the state of port B until any button is pressed. The input_b() reads the value of the buttons on port b, the value will be 1 if the button is pressed and 0 if not. Once a button is pressed the while condition becomes false and the routine stops running. The return value variable contains the bitwise representation of which buttons were pressed.

It then introduces a delay of 500 milliseconds, which is intended to allow time for the button to be fully released after pressing, it also helps avoid multiple readings from a single press. It then goes to the return (value) variable which contains the data representation of which buttons were pressed on port B.

```

11
12
13 char read_buttons(void)
14 {
15     char value=0;
16     while(!input_b());
17     value=input_b();
18     delay_ms(500);
19     return (value);
20 }
21

```

// setup a variable for the buttons
// wait for button press
// get the value of port b
// time button release
// return the value of the buttons pressed

Figure 4 Read buttons Function (Master)

The following will be covering the content of figure 5 and 6. This part of the programme reads the button inputs, toggles various car related lights on and off. This part of the report will briefly go over some of the important lines of code from the figures below.

The “char buts = 0;” declares a variable to store the value of buttons pressed as detected by the previous “read_buttons()” function.

The “while(TRUE)” creates an infinite loop that will continuously check for button presses and then responds when pressed. The next line “buts = read_buttons()” is used for reading the current button state and stores them. Each button corresponds to a specific case in the switch statement.

The next part of the code uses “switch (buts)” to select what output to select depending on which button is pressed. Each button is programmed to turn on a specific LED. The cases for this are as follows:

Case 1: Controls the brake lights.

Case 2: Controls the Rear lights

Case 4: Controls the right indicator

Case 8: Controls the rear fog lights

Case 16: Controls the Reversing lights

Each of the above cases sets the “txbyte” to a unique character. It sets it to lowercase if the light is turned off and uppercase if the light is turned on. It will then output the character “printf(“%c”, txbyte);” to the virtual terminal displaying the current state of the light as shown in figure 7. It will then store the character in the EEPROM using “ep_w(0, txbyte);” It does this in order to retrieve the last state of the light if the simulation is stopped and restarted.

```

void main()
{
    char buts=0;                // button value

    while(TRUE)
    {
        buts=read_buttons();    // get the value of buts as a character
        switch (buts)
        {
            case 1: if(local_output & 0x01)                // toggle the brake lights lights on and off
            {
                local_output-=1;                // set bit off
                txbyte='a';                    // set the token
            }
            else
            {
                local_output+=1;                // set bit on
                txbyte='A';                    // set the token
            }
            break;
            case 2: if(local_output & 0x02)                // toggle the rear lights on and off
            {
                local_output-=2;
                txbyte='b';
            }
            else
            {
                local_output+=2;
                txbyte='B';
            }
            break;
            case 4: if(local_output & 0x04)                // toggle the right indicator on and off
            {
                local_output-=4;
                txbyte='c';
            }
            else
            {
                local_output+=4;
                txbyte='C';
            }
            break;
        }
    }
}

```

Figure 5 void main (master 1)

```

        case 8: if(local_output & 0x08)                // toggle the fog lights on and off
        {
            local_output-=8;
            txbyte='d';
        }
        else
        {
            local_output+=8;
            txbyte='D';
        }
        break;
        case 16: if(local_output & 0x10)                // toggle the reverse lights on and off
        {
            local_output-=0x10;
            txbyte='e';
        }
        else
        {
            local_output+=0x10;
            txbyte='E';
        }
        break;
        default:
        break;
    }
    printf("%c",txbyte);                // print the token character to the rs232 output
    ep_w(0,txbyte);                    // store the data in mem location 0;
}

```

Figure 6 void main (master 2)

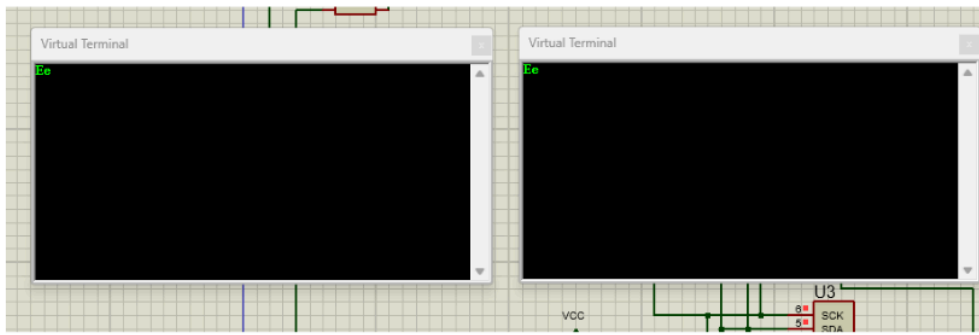


Figure 7 virtual terminals

The final part of the code is the “ep_w” and the ep_r” functions as seen in figure 8 below. They perform the read and write operations to the memory chip used in the I2C protocol. The “void ep_w(char where, char what)” function starts the I2C communication with the memory chip and writes the data to specified memory addresses. It adds a delay to ensure that the chip has enough time to process each command. The other function “char ep_r(char where)” sets up the read data from a specified address in the I2C communication.

```
void ep_w(char where, char what)
{
    while(!input(hs));           // Wait until memory chip is available
    output_low(hs);              // Set hs line Low to indicate busy state
    i2c_start();                 // Initiate I2C start condition
    i2c_write(0xA0);             // Send chip type and device address (write mode)
    i2c_write(where);           // Send target memory address for writing
    i2c_write(what);             // Write data to the specified address
    delay_ms(20);               // Wait for write operation to complete
    i2c_stop();                 // Send I2C stop condition
    delay_ms(20);               // Allow time for the memory chip to store data
    input(hs);                  // Release hs line to indicate availability
}

char ep_r(char where)
{
    char data = 0;
    while(!input(hs));           // Wait until memory chip is available
    output_low(hs);              // Set hs line Low to indicate busy state
    i2c_start();                 // Initiate I2C start condition
    i2c_write(0xA0);             // Send chip type and device address (write mode)
    i2c_write(where);           // Send memory address to read from
    delay_ms(20);               // Delay to allow memory chip to process read request
    i2c_start();                 // Repeat start condition for read operation
    delay_ms(20);               // Delay for chip readiness
    i2c_write(0xA1);             // Send chip type and device address (read mode)
    data = i2c_read(0);          // Read data from the specified memory address
    delay_ms(20);               // Wait after reading
    i2c_stop();                 // Send I2C stop condition
    delay_ms(20);               // Final delay for chip to finish processing
    input(hs);                  // Release hs line to indicate availability
    return(data);               // Return the read data
}
```

Figure 8 read write functions (Master)

Slave

This section of code will break down the code used in the slave PIC. The overall function of the slave is to read inputs and control the outputs based on specific commands, in this case the outputs are the LEDs.

The global variables are used throughout the program and are shown in figure 9 below, they are used to monitor the input and output states, counters and specific device states.

```
char local_output = 0;           // Stores the local output state
char control_byte = 0;          // Controls the output state sent to port B
unsigned int lights = 0;         // Flag indicating if lights are on
unsigned int counter = 0;        // Counter for timer interrupt
unsigned int flash = 0;          // Flashing state (toggles in timer)
unsigned int left = 0;           // Flag for left indicator state
unsigned int right = 0;          // Flag for right indicator state
CHAR buts = 0;                  // Stores current input from keyboard
char old_but = 0;               // Stores previous input from keyboard
```

Figure 9 Global variables (Slave)

The next part of the code was the timer interrupt (Timer2_isr). This routine is called 15 milliseconds as set up in the main, it then toggles the flash variable every 600 ms. This is used to control the flashing indicator (right indicator). This is shown below in figure 10.

```
#INT_TIMER2
void TIMER2_isr(void)
{
    counter++;
    if (counter == 40) // 40 * 15 ms = 600 ms
    {
        counter = 0;
        flash = !flash; // Toggle flash between 0 and 1
        flash_control(); // Update flash-based controls
    }
}
```

Figure 10 Timer (slave)

The next part of the code is the “main” function. This part sets up the timer, enables interrupts and constantly checks for any inputs from the buttons. If it notices a change in the input it will then process the command and write it to memory. It also sends the corresponding character to the virtual terminal and displays it, this is to show the user what lights should or shouldn't be active. This is all shown below in figure 11.

```

void main()
{
    // char value=0xff;
    // char q=0;
    setup_timer_2(T2_DIV_BY_16,249,15); //1.0 ms overflow, 15.0 ms interrupt
    enable_interrupts(INT_TIMER2);
    enable_interrupts(GLOBAL);

    while(true)
    {
        buts=ep_r(0x00); // read data byte from keyboard
        if(old_but!=buts) // if buts <> old_but
        {
            printf("%c",buts); // print the character of buts to the virtual terminal
            ep_w(0x10,but); // write the value recieved into the memory @ address 16 (0x10)
            delay_ms(20); // wait 20ms
            payload(); // run the payload operations.
        }
        old_but=but; // copy the value of buts into old_but (becomes the previous value)
    }
}

```

Figure 11 void main (Slave)

The next function is the payload function this is used to monitor what characters are being received and then sending the corresponding command to Port B which will then turn on the light if the character in question is a capital and turn it off if the character is lower case. This shown below in figure 12.

```

void payload(void)
{
    // carry out the payload operations.
    switch(buts)
    {
        case 'A': local_output+=0x01; control_byte+=0x01; break;
        case 'a': local_output=local_output & 0xfe; control_byte=control_byte & 0xfe; // AND the local output byte with 0xfe to clear bit 0
                  break;
        case 'B': local_output+=0x02; control_byte=control_byte+0x05; //turn on rear light
                  break;
        case 'b': local_output=local_output & 0xf1; control_byte=control_byte & 0xf1; // AND the local output byte with 0xfe to clear bits 1,2,3
                  lights=0; //output_low(pin_a7); break;
        case 'C': local_output+=0x20; //turn on Right indicator
                  right=1; break;
        case 'c': local_output=local_output & 0xdf; // AND the local output byte with 0xfe to clear bit 5
                  right=0; break;
        case 'D': //output_high(pin_A7); // turn on bit 7 (fog) if lights are on
                  control_byte=control_byte+0x08; break;
        case 'd': control_byte=control_byte & 0xf7; //output_low(pin_a7); // turn off bit 7
                  break;
        case 'E': local_output+=0x40; //turn on rev lights
                  control_byte=control_byte+0x40; break;
        case 'e': local_output=local_output & 0xbf; control_byte=control_byte & 0xbf; // AND the local output byte with 0xfe to clear bit 6
                  break;
    }
    output_b(control_byte); // send the data to the b Port.
}

```

Figure 12 payload function (Slave)

The next function is the write to memory function “void ep_w”. This function writes a byte of data to a specified memory address using I2C protocol. The “ep_r” or read from memory is the function that reads data from the specified memory address using I2C protocol and returns it. This h=shown below in figure 13

```

void ep_w(char where, char what)
{
    while(!input(hs)); // wait for mem chip to be available
    output_low(hs); // set hs line low (mem busy)
    i2c_start(); // issue i2c start pulse
    i2c_write(0xA0); // issue chip type and number
    i2c_write(where); // send the address to write to
    i2c_write(what); // send the data to be written
    delay_ms(20); // issue an i2c stop pulse
    i2c_stop(); // allow i2c chip to store data
    input(hs); // release hs line
}

char ep_r(char where)
{
    char data=0;
    while(!input(hs)); // wait for mem chip to be available
    output_low(hs); // set hs line low (mem busy)
    i2c_start(); // issue i2c start pulse
    i2c_write(0xA0); // issue chip type and number
    i2c_write(where); // send address to be read from
    delay_ms(20); // wait for chip to set its for read
    i2c_start(); // issue another start pulse
    i2c_write(0xA1); // issue chip type and number read command
    data=i2c_read(0); // read data from the chip
    delay_ms(20);
    i2c_stop();
    input(hs);
    return(data);
}

```

Figure 13 Memory (Slave)

The last function is the “flash_control”. This function controls the state of the right indicator based on the “flash” variable set by “TIMER2_isr”. If “flash” and “right” are both set it will then turn on pin_b5 to enable flashing, otherwise it turns off “pin_b5”.

Analysis/ Results

Below is a very simple flow chart showing how the system works and the master and slave communicate with each other.

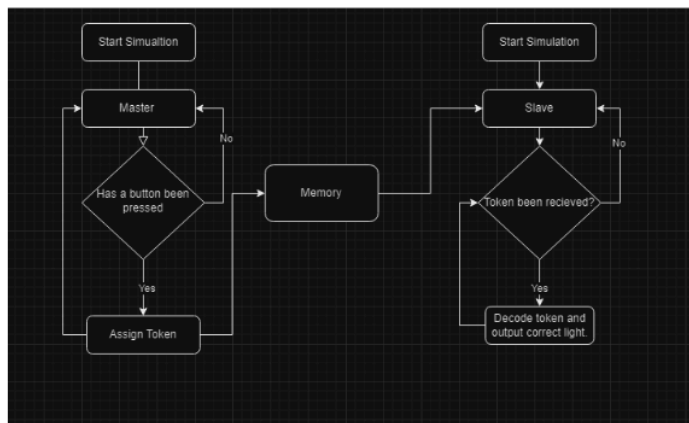


Figure 14 flow chart

The next image figure 15 shows the system once it had been started after the reverse list (E) had been left on when it was last stopped. The brake light was then turned on after the simulation had started as shown with the corresponding light being on and a capital A in the virtual terminal.

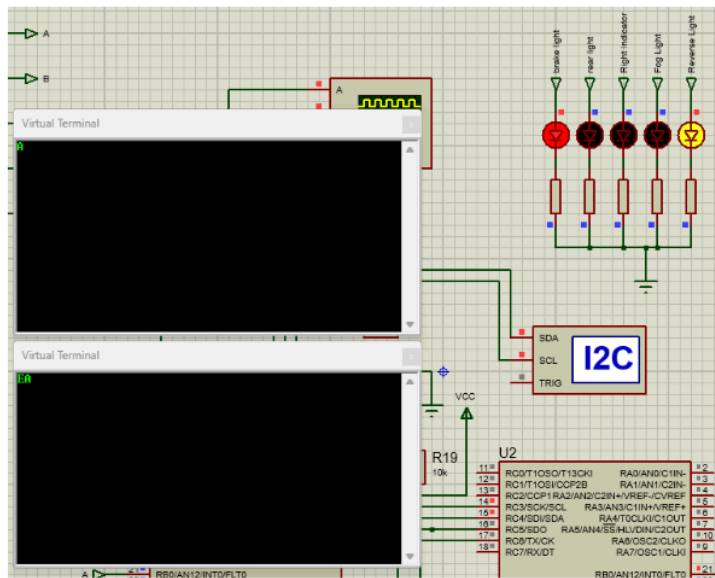


Figure 15 Leds and virtual terminal

Conclusion

In conclusion, this report has covered three different transmission types from point-to-point, point-to-multi-point, and multi-master systems. The uses for each transmission type was explained with examples given. There was also a design task to create an I2C-based control system using two PIC microcontrollers, one of which was set up as the master and the other the slave. This task was completed using proteus. Overall this report combined research and practical work in order to create a reliable transmission control system. It will be useful for potential future projects that need efficient communication between multiple devices.

There is one bug that if all lights are left on when the simulation is stopped only one comes on. If only one is left on then it works as it should. This problem could be solved with more time developing the code. Other functionality that could have been added:

- Added lights for the other corners of a car.
- Possibility of adding a horn (using a buzzer on proteus)
- More condition statements could have been added to better imitate a real car.

On the other hand, in order eliminate any issues in the code it could have been made much simpler by using only one or two lights making it much simpler.

Bibliography

[1]

"What is a Quality Management System (QMS)?," *SimplerQMS*, 2023.
<https://simplerqms.com/quality/qms/what-is/>

[2]

A. S. Tanenbaum and D. Wetherall, *Computer networks*. Pearson India Education Services Pvt, Limited, India, 2014. Available: <https://csc-knu.github.io/sys-prog/books/Andrew%20S.%20Tanenbaum%20-%20Computer%20Networks.pdf>

[3]

A. Chen, "What is Point-to-Multipoint Communication?," *Sanny Telecom*, May 18, 2024.
<https://www.sannytelecom.com/what-is-point-to-multipoint-communication/>

[4]

"TB3191 I²C Master Mode," 2018. Accessed: Nov. 13, 2024. [Online]. Available:
<https://ww1.microchip.com/downloads/en/AppNotes/I2C-Master-Mode-30003191A.pdf>

[5]

S. Campbell, "Basics of the I2C Communication Protocol," *Circuit Basics*, Feb. 13, 2016.
<https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/>

[6]

"8 Bit MCU, Flash, PIC18 Family PIC18F25xx Series Microcontrollers, PIC18, 40 MHz, 48 KB, 28 Pins," *Farnell.com*, 2024. <https://uk.farnell.com/microchip/pic18f2525-i-so/mcu-8bit-pic18-40mhz-soic-28/dp/1212694> (accessed Nov. 07, 2024).

Henry Assignment DIS

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

70/100

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

HENRY RENNISON

70.35

RESEARCH-INFORMED LITERATURE



You have explained what point to point and point to multipoint and multimaster are correctly. 65%

13

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, except the fog light should be disabled unless the main lights are active 68%

17

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 reasonable number of references. 72%

10.8