

JS DIS assignment

by Jacob Smelt

Submission date: 15-Nov-2024 02:22PM (UTC+0000)

Submission ID: 242800168

File name: 41191_Jacob_Smelt_JS_DIS_assignment_523660_380577324.docx (1.05M)

Word count: 3071

Character count: 15045

Distributed Intelligence Systems Assignment

ACDA5018

JACOB SMELT (2215291)

Table of Contents

Abstract.....	3
Introduction	3
Quality management systems	3
Transmission Systems	3
Point to Point	3
Multipoint	3
Multi-Master	4
My System Design	5
Proteus circuit diagram	5
Components.....	5
PIC18F2525	5
24LC02B Module	6
I2C debugger	6
The Systems Programs	6
Purpose of the Program	6
Master PIC Code.....	6
Slave PIC code	10
System Flowchart	15
Conclusion.....	15
References	16

Figure 1 Multimaster communication [5]	4
Figure 2 Proteus Circuit.....	5
Figure 3 Defining functions and variables.....	6
Figure 4 Main program part one.....	8
Figure 5 Main program part two.....	8
Figure 6 Read_buttons function	9
Figure 7 EEPROM write function.....	9
Figure 8 Variable and function declarations.....	10
Figure 9 Timing function	10
Figure 10 Main function code	11
Figure 11 Payload function part one.....	12
Figure 12 Payload function part two	13
Figure 13 Ep_w and ep_r functions	14
Figure 14 Flash_control function	14
Figure 15 System flowchart	15

Abstract

This report will detail the design and programming of a token-based distributed network system for controlling vehicle lights, utilizing two PIC18F2525 microcontrollers. The system operates using I2C communication to transfer control tokens between a master and slave microcontroller. The master microcontroller receives input from switches simulating vehicle functions such as running lights, brake lights, indicators, fog lights, and reversing lights. The slave then uses these tokens to execute different commands turning the lights on. The project is built in proteus and uses I2C communication and EEPROM storage to create a reliable lighting control system for a vehicle.

Introduction

In this assignment I will design and develop a token based distributed network that is used to control a set of lights on a selected corner of a vehicle. This system will be designed using proteus by using two of the PIC 18F2525. These PICs are used to communicate with each other using a token system, one PIC takes some inputs from buttons and assigns different tokens depending on the button pressed. This token is then transmitted to the second PIC where it is used to control the lights for the rear left light cluster on a vehicle.

Quality management systems

ASQ states "A quality management system (QMS) is defined as a formalized system that documents processes, procedures, and responsibilities for achieving quality policies and objectives" [1] this shows that QMS' are an essential structure ensuring that all work meets standards that help with improvement and safety. It ensures that people are adhering to industry standards and regulations to make sure that the work completed is compliant. QMS also ensures that customer and employee feedback is taken into account to identify areas for improvement by using key performance indicators. A good QMS has preventative measures that ensures issues are addressed quickly and effectively whilst also providing documentation and accountability. QMS makes sure that operational and safety standards are constantly being improved.[2].

Transmission Systems

Point to Point

A point-to-point transmission system is a system in which data is sent directly from a device/sensor to a control unit like a PIC rather than going through a central hub. As Sanny telecom states point to point "facilitates private and direct data exchange" [3] meaning that data is more secure and has much less latency than if it were to pass through a central exchange hub. The system also has a higher reliability than the others listed below as there is a dedicated data link between the control device and the sensor meaning there will not be a problem with data traffic or interference. There are a few downsides to this type of system, and these include scalability and flexibility as each new device added onto the system requires a dedicated direct communication line to the PIC.

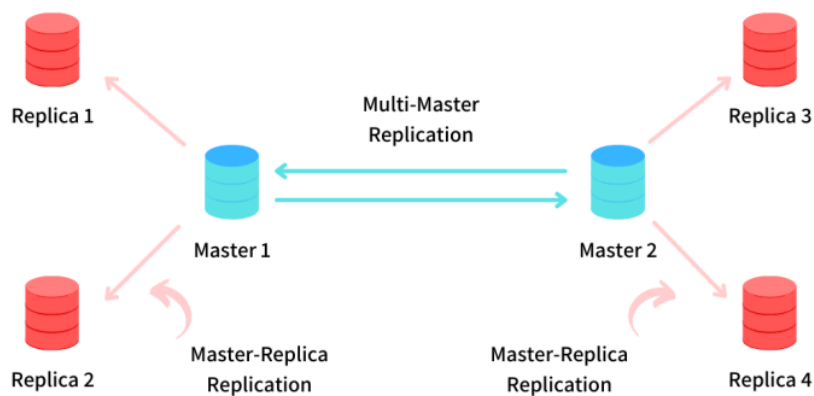
Multipoint

A multipoint transmission system is one in which multiple sensors/devices are connected to a single communication channel that then connects to the control device. This is like a Bus network meaning that there us a single transmission line, having this single line means that there is potential for traffic build up if the bandwidth is limited making it slower than a point point, this system topology also "does not provide security and privacy because communication channel is shared"[4] increasing the chance of interception. Scalability is also much less of a problem with this system as you can just

easily add another device to the main transmission line, this also makes it more cost effective than point to point.

Multi-Master

A multi-master transmission system (as seen in Figure 1) uses the same kind of bus system that the multipoint uses however there are multiple masters connected to the main transmission line; multiple masters can initiate the communication with other things such as sensors independently. This communication method uses certain protocols to stop data conflicts when multiple devices are communicating at the same time. Multi master is much more efficient than the multipoint since there can be multiple masters on the system, however it is much more complex than the other transmission systems and requires a larger bandwidth due to the simultaneous communication.



Multi-Master Replication

Figure 1 Multimaster communication [5]

My System Design

Proteus circuit diagram

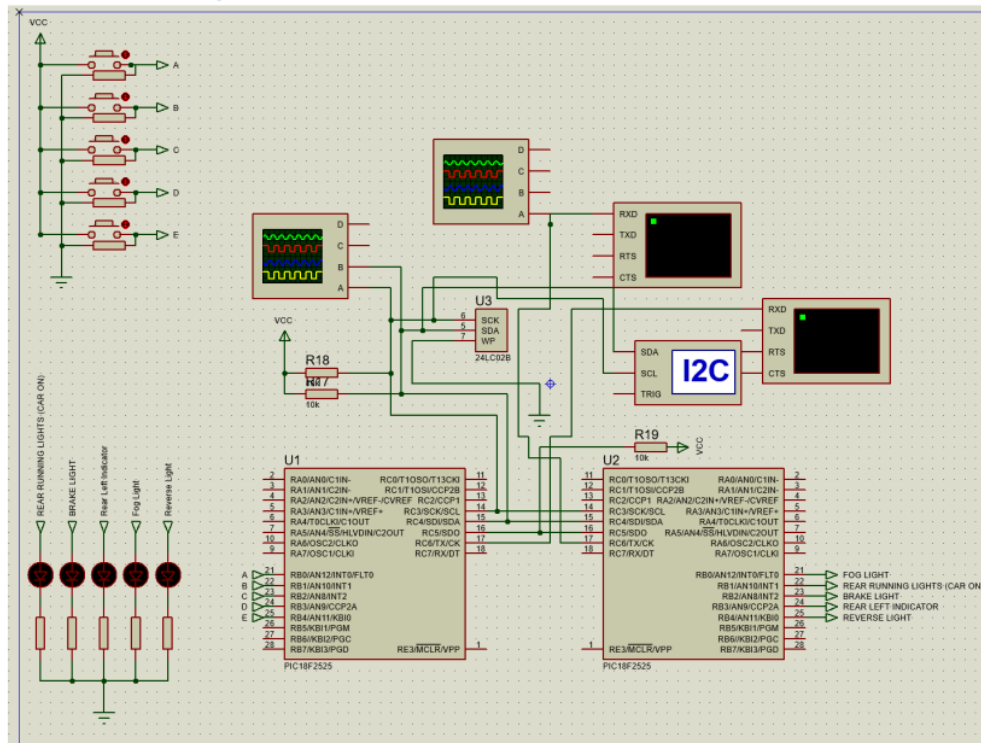


Figure 2 Proteus Circuit

Components

- 2 * PIC18F2525
- 24LC02B Module
- Resistors
- Switches
- LED's
- Oscilloscope
- I2C debugger
- Virtual Terminal

PIC18F2525

The PIC 18F2525 is an 8 bit microcontroller that can handle high speed processing and can also be used for I2C communication methods. In this project I made use of two of the PIC's , on the first PIC I used port B as input ports to take the inputs from the switches. On the second PIC I have also used port B but this time I have used it to connect outputs to control and turn on the different LED's I have set up. I have connected both PIC's SCL (Serial Clock) , SDA (Serial Data) and also the SDO (Serial Data Out) ports together, this allows the PIC's to successfully communicate with each other so that the token method can be used.

24LC02B Module

This module is an EEPROM module, EEPROM (Electrically Erasable Programmable Read Only Memory) is used in this circuit to store and retrieve the data used such as the tokens created by the first PIC that need to be retrieved by the second PIC. The WP Pin (Write Protect) has been connected to ground so that the write feature is enabled in the EEPROM module.

I2C debugger

The I2C debugger in the circuit is used to monitor and view the I2C communication line, this helps with fault finding and allows you to see the values changing as the PIC's communicate with each other.

The Systems Programs

In this section of my report, I will be providing screenshots of my commented code for both the Master and the Slave PIC and explaining what each line of code does and why I have programmed it to do so.

Purpose of the Program

This program has been created to mimic the rear left light module from a vehicle. The first switch (Switch A) activates the running lights on the car and is representative of the vehicle running after being turned on. The second switch (Switch B) activates the brake lights. The third switch (Switch C) activates the rear left indicators, this uses a flash control function so that the indicators flash like a real vehicle rather than staying solid. The fourth switch (Switch D) is used to activate the fog lights. The final switch (Switch E) is used to activate the reversing light, this would only be activated when the vehicle is in the reverse gear. All the lights apart from the indicators will turn off when the first switch is turned off as this simulates the vehicle having been turned off. The indicators will still be able to flash if they are activated when the car is turned off as this happens in real life when the hazard lights are turned on. The fog lights, reversing light and brake light will also reactivate if previously turned on before the running lights are turned off (First switch).

Master PIC Code

Figure 3 shows the beginning of my code for the Master controller PIC. This section of code is used to assign some variables and to define functions with different names. The functions are seen in lines 4 and 5. The one that starts with char is a function where the result from running the function is expected to be a character. The part in the brackets tells the code what is expected to be given to the function, so when it has "char,char" this means it is expecting two characters to be inputted into the function. local_output and txbyte both also have char in front of them, this is assigning a variable that is also expected to be a character.

```
*main.c main.c
1 #include <main.h> // Include the main header file
2
3
4 void ep_w(char, char); // Write data to a specified Location
5 char read_buttons(void); // Read button states
6
7
8 char local_output; // Tracks the current output status for toggling
9 char txbyte; // Byte to be transmitted depending on what button has been pressed
10
```

Figure 3 Defining functions and variables

Figure 4 Main program part oneFigure 4 and Figure 5 show the main program. Here a new variable is defined labelled "buts", below this in the infinite loop "buts" is given a value by calling the "read_buttons" function (Figure 6) and using the output from that function as the new value for buts. This function is used to determine which one of the five buttons has been pressed and works by reading the inputs on port b of the first PIC. Once the input has been read it assigns it to a variable called value and then has a short delay before returning that value to where the function was called from in turn assigning the read value to "buts". This value of "buts" is then used to determine which case number to run.

Case 1 gets activated if the first button is pressed, this is used to activate the rear running lights. The code on the first PIC simply transmits a character to the second PIC using I2C where it is then used to run another set of code, however, to decide which letter to send across different cases are used. When the first button is pressed and case 1 is incidentally activated it first checks to see if the button has already been previously activated. This is done with the use of the if statement. If the least significant bit (the first bit) is already set (button has been pressed before) then it runs "local_output -= 1 and txbyte = 'a' " this clears the least significant bit by decrementing the local output by one. It then assigns the letter "a" to the txbyte and transmits it to the second PIC. If the least significant bit is not set (the button hasn't been pressed before) then it adds one to the local output and transmits "A".

For case 2 two is either added or removed from the local output and "b" or "B" is transmitted. For case 4 four is either added or removed and "c" or "C" this convention is repeated for case 8 and case 16, with the upper-case letters being transmitted when the switch is supposed to be turned on and lower case when it is being turned off. Once a cases code has been run the program breaks from the switch loop and uses the printf function and the ep_w function shown at the bottom of Figure 5. The printf function is used to print the txbyte to the virtual terminal so that viewing of the txbyte is available for testing purposes. The ep_w function is then called and two values are given to the function "(0,txbyte)" the "0" is used to tell the function where to write in the EEPROM and "txbyte" is used to tell the function what to write.

```

main.c main.c
13 // Main
14 void main()
15 {
16     char buts = 0; // Variable for button status
17
18     while (TRUE) //Loop
19     {
20         buts = read_buttons(); // Read the current button state
21
22         // Switch case to handle actions based on which button is pressed
23         switch (buts)
24         {
25             case 1: // Case 1: Rear running Lights
26                 if (local_output & 0x01) // Check if the Least significant bit is set (button is already activated)
27                 {
28                     local_output -= 1; // Clear the bit by decrementing
29                     txbyte = 'a'; // Set transmit byte to 'a'
30                 }
31                 else
32                 {
33                     local_output += 1; // Set the bit by incrementing
34                     txbyte = 'A'; // Set transmit byte to 'A'
35                 }
36                 break;
37
38             case 2: // Case 2: Rear brake Lights
39                 if (local_output & 0x02) // Check if the 2nd bit is set (button is already activated)
40                 {
41                     local_output -= 2; // Clear the bit
42                     txbyte = 'b'; // Set transmit byte to 'b'
43                 }
44                 else
45                 {
46                     local_output += 2; // Set the bit
47                     txbyte = 'B'; // Set transmit byte to 'B'
48                 }
49                 break;
50
51             case 4: // Case 3: Rear Left indicator
52                 if (local_output & 0x04) // Check if the 3rd bit is set (button is already activated)
53                 {
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97

```

Figure 4 Main program part one

```

main.c main.c
50
51 case 4: // Case 3: Rear Left indicator
52     if (local_output & 0x04) // Check if the 3rd bit is set (button is already activated)
53     {
54         local_output -= 4; // Clear the bit
55         txbyte = 'c'; // Set transmit byte to 'c'
56     }
57     else
58     {
59         local_output += 4; // Set the bit
60         txbyte = 'C'; // Set transmit byte to 'C'
61     }
62     break;
63
64 case 8: // Case 4: Fog Lights
65     if (local_output & 0x08) // Check if the 4th bit is set (button is already activated)
66     {
67         local_output -= 8; // Clear the bit
68         txbyte = 'd'; // Set transmit byte to 'd'
69     }
70     else
71     {
72         local_output += 8; // Set the bit
73         txbyte = 'D'; // Set transmit byte to 'D'
74     }
75     break;
76
77 case 16: // Case 5: Reversing Light
78     if (local_output & 0x10) // Check if the 5th bit is set (button is already activated)
79     {
80         local_output -= 0x10; // Clear the bit
81         txbyte = 'e'; // Set transmit byte to 'e'
82     }
83     else
84     {
85         local_output += 0x10; // Set the bit
86         txbyte = 'E'; // Set transmit byte to 'E'
87     }
88     break;
89
90 default: // No button pressed state
91     break;
92
93 }
94
95 printf("%c", txbyte); // Print the current txbyte value to the virtual terminal
96 ep_w(0, txbyte); // Write txbyte to the EEPROM using the ep_w function defined below
97 }

```

Figure 5 Main program part two

```

// Function to read the button states
char read_buttons(void)
{
    char value = 0;           // Variable to hold button state
    while (!input_b());       // Wait until button input is available
    value = input_b();        // Read the button state
    delay_ms(500);            // Delay to avoid bouncing effect
    return (value);           // Return the button state
}

```

Figure 6 Read_buttons function

Figure 7 is the function used to write to the EEPROM. When the function is called two values are given to it separated by a comma. These are then inputted into the where, and what. In the case of this program the where is "0" and the what is "txbyte". The program waits for the hs signal to be high which means that the other PIC isn't using the communication line at that time. It then starts the I2C communication and writes the "txbyte" into the address that has been given, "0" in this case. There are then a few delays making sure that there is enough time for it to be processed. After stopping the I2C communication the hs line is then brought back high so the other PIC can begin reading or writing without causing a collision of traffic.

```

98 // Function to write a byte 'what' to a given address 'where' via I2C
99 void ep_w(char where, char what)
100 {
101     while (!input(hs));      // Wait until the 'hs' signal is high
102     output_low(hs);          // Lower the 'hs' signal
103     i2c_start();             // Start I2C communication
104     i2c_write(0xA0);         // Send the device address
105     i2c_write(where);        // Send the specified address to write to
106     i2c_write(what);         // Send the data to write
107     delay_ms(20);            // Delay to ensure completion
108     i2c_stop();              // Stop I2C communication
109     delay_ms(20);            // Delay for stability
110     input(hs);               // Restore 'hs' signal state
111 }

```

Figure 7 EEPROM write function

Slave PIC code

The Slave codes purpose is to take the transmitted byte (Letter) from the EEPROM and in turn use it to activate different functions that turn different lights on. Figure 8 shows the start of the slave PICS code where different functions and variables that are used later on in the code are declared.

```
main.c main.c
1  #include <main.h> // Includes main header file
2
3  // variable declarations
4  char control_byte = 0; // Stores control byte for external functions
5  unsigned int lights = 0; // Status of lights (1 = ON, 0 = OFF)
6  unsigned int counter = 0; // Counter for timing
7  unsigned int flash = 0; // Flash state (used for blinking effect)
8  unsigned int left = 0; // Left indicator status
9  CHAR buts = 0; // Holds current button state
10 char old_but = 0; // Holds previous button state
11 int D_activated = 0; // Status of 'D' activation
12
13 // Function declarations
14 char ep_r(char); // Reads a value from a specified address
15 void ep_w(char where, char what); // Writes a value to a specified address
16 void flash_control(void); // Controls flashing for indicator lights
17 void payload(void); // Processes actions based on button presses
18
```

Figure 8 Variable and function declarations

Before beginning the main program, another function is declared and written. This function seen in Figure 9 is used as a timer for the indicators. This makes use of a counter to toggle the flashing of the indicator also using another function called “flash_control” this toggles the LED on and off every 160 ms by toggling the state of the variable called “flash”.

```
21 // Interrupt Service Routine for Timer2
22 #INT_TIMER2
23 void TIMER2_isr(void)
24 {
25     counter++; // Increment the counter
26     if (counter == 40) // Check if counter reached 40 (timer threshold)
27     {
28         counter = 0; // Reset counter
29         flash = !flash; // Toggle flash state
30         flash_control(); // Update flash control
31     }
32 }
```

Figure 9 Timing function

Figure 10 shows the main function for the slave PIC. First the timer is setup as this is used in the flash function. There is then an infinite loop in which ep_r is used to assign a value to a variable "buts". Ep_r is a function used to read from the EEPROM. This reads that value in the EEPROM that has been stored by the master PIC and uses it in the slave PIC's code. This main function is used to constantly read the txbyte stored in the EEPROM and uses the if statement to reprint the txbyte in the virtual terminal if it changes. It also then writes the new value of "buts" into a separate address in the EEPROM for use later and begins running the "payload" function.

```
*main.c *main.c
33 // Main function
34 void main()
35 {
36     setup_timer_2(T2_DIV_BY_16, 249, 15); // Setup Timer2
37     enable_interrupts(INT_TIMER2);        // Enable Timer2 interrupt
38     enable_interrupts(GLOBAL);           // Enable global interrupts
39
40     while (true)                          // Main Loop
41     {
42         buts = ep_r(0x00); // Read current button state from address 0x00
43
44         if (old_but != buts) // Check if button state has changed
45         {
46             printf("%c", buts); // Print the current button state
47             ep_w(0x10, buts);    // Write the new button state to address 0x10
48             delay_ms(20);        // Wait for debounce
49             payload();           // Process button actions
50         }
51
52         old_but = buts; // Update the previous button state
53     }
54 }
```

Figure 10 Main function code

The payload function shown below in Figure 11 works by taking the variable “buts” which now has a character assigned to it and using the character to run different pieces of code. If the first button is pressed the txbyte “A” will be transmitted to the slave PIC and case ‘A’ will be run. This case adds two to the control_byte variable and changes the “lights” variable to be high. It then also checks to see if “D_activated” is true. This is used so that if the fog lights have previously been turned on then they will be reactivated when the running lights are also reactivated. It does this by adding one to the control_byte. When case “a” is selected (Button A has been turned off) the code turns off all of the lights (simulating engine shutdown) apart from the indicators as these can be used when the engine is off for hazard lights. It does this by setting all bits apart from the indicators to 0.

The if statement in case ‘B’ makes it so that the brake lights can only be activated if the running lights are also on (engine on). Finally, when case ‘C’ is activated or deactivated it sets a variable lights to either a 1 or a 0, this is used later on in the flash function to flash the indicators.

```
void payload(void)
{
    switch (buts)                // Determine action based on button press
    {
        case 'A':                // If txbyte "A" present
            control_byte += 0x02; // Update control_byte
            lights = 1;           // Turn on Lights
            if (D_activated)      // If 'D' is activated
                control_byte += 0x01; // Update control_byte for 'D'
            break;

        case 'a':                // If txbyte "a" present
            control_byte &= 0x08; // Clear bits in control_byte
            lights = 0;           // Turn off Lights
            break;

        case 'B':                // If txbyte "B" present
            if (lights)          // If lights are on
            {
                control_byte += 0x04; // Update control_byte
            }
            break;

        case 'b':                // If txbyte "b" present
            control_byte &= 0xFB; // Clear bit in control_byte
            break;

        case 'C':                // If txbyte "C" present
            left = 1;             // Enable Left indicator
            break;

        case 'c':                // If txbyte "c" present
            left = 0;             // Disable Left indicator
            break;
    }
}
```

Figure 11 Payload function part one

Figure 12 has the last part of the payload. These cases are similar to the previous ones however there is a variable that is set high if the foglights are turned on. This is used so that when the running lights are turned back on after being turned off, if the fog lights were previously active=ated they will then turn back on again. The final line of the payload is "output_b(control_byte)" this line outputs the binary value stored in "control_byte" to port b of the slave PIC thus turning the correct lights on.

```

90
91     case 'D':           // If txbyte "D" present
92     if (lights)         // If lights are on
93     {
94         control_byte += 0x01;      // Update control_byte
95         D_activated = 1;           // Set D_activated to 1
96     }
97     break;
98
99     case 'd':           // If txbyte "d" present
100    control_byte &= 0xFE;          // Clear bit in control_byte
101    D_activated = 0;               // set D_activated to 0
102    break;
103
104    case 'E':           // If txbyte "E" present
105    if (lights)         // If lights are on
106    {
107        control_byte += 0x10;      // Update control_byte
108    }
109    break;
110
111    case 'e':           // If txbyte "e" present
112    control_byte &= 0xEF;          // Clear bit in control_byte
113    break;
114 }
115
116 output_b(control_byte); // Output control_byte to hardware
117 }

```

Figure 12 Payload function part two

Figure 13 shows the ep_w function that has already been explained and the ep_r function. The ep_r function is very similar to the write function. It waits until the EEPROM is free and then reads the data from the specified address. It then stores this data in a variable called "data" and returns this data to where the function was called from, in this case a variable called "buts"

```

119 // Writes data to specified address over I2C
120 void ep_w(char where, char what)
121 {
122     while (!input(hs)); // Wait until the line is available
123     output_low(hs); // Pull line low to initiate communication
124     i2c_start(); // Start I2C communication
125     i2c_write(0xA0); // Send write command
126     i2c_write(where); // Send address to write to
127     i2c_write(what); // Send data to write
128     delay_ms(20); // Delay for device processing
129     i2c_stop(); // End I2C communication
130     delay_ms(20); // Additional delay for safety
131     input(hs); // Release the line
132 }
133
134 // Reads data from specified address over I2C
135 char ep_r(char where)
136 {
137     char data = 0; // Data variable to hold read value
138     while (!input(hs)); // Wait until the line is available
139     output_low(hs); // Pull line low to initiate communication
140     i2c_start(); // Start I2C communication
141     i2c_write(0xA0); // Send read command
142     i2c_write(where); // Send address to read from
143     delay_ms(20); // Delay for device processing
144     i2c_start(); // Restart communication
145     delay_ms(20); // Delay
146     i2c_write(0xA1); // Send read command again
147     data = i2c_read(0); // Read data
148     delay_ms(20); // Delay
149     i2c_stop(); // End communication
150     delay_ms(20); // Additional delay
151     input(hs); // Release the line
152     return data; // Return the read data
153 }
154

```

Figure 13 Ep_w and ep_r functions

Finally, in Figure 14 the flash control function is seen. This function uses an if statement so that both the left variable and flash variable are set for it to output high onto the indicator pin. This allows for the use of the timing interrupt to make the LED flash correctly.

```

155 // Controls flashing output for left indicator based on flash state
156 void flash_control(void)
157 {
158     if (left & flash) // If left indicator is on and flash state is active
159         output_high(pin_b3); // Turn on pin for left indicator
160     else
161         output_low(pin_b3); // Turn off pin for left indicator
162 }

```

Figure 14 Flash_control function

System Flowchart

Figure 15 shows the system flowchart.

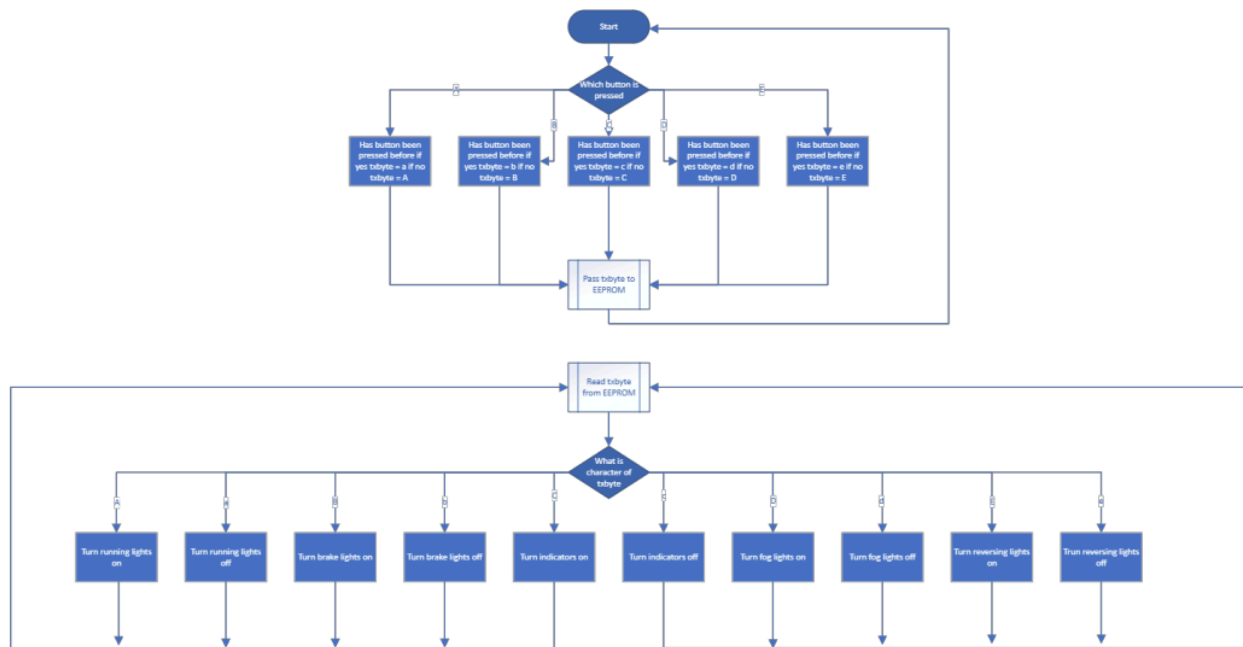


Figure 15 System flowchart

Conclusion

In this project, I successfully designed and implemented a token-based distributed network system to control vehicle lights using two PIC18F2525 microcontrollers. The system operates through I2C communication, where the master PIC reads inputs from switches to generate tokens, which are then transmitted to the slave microcontroller to be used to control the corresponding lights. The use of EEPROM in this design allows for storage and retrieval of token values, ensuring accurate communication between the two devices. The system also supports multiple functions, such as controlling the running lights, brake lights, indicators, fog lights, and reversing lights, whilst ensuring that certain lights can only be activated when the running lights are on.

Overall, the project demonstrates the use of I2C communication, microcontroller programming, and a distributed network design in a real-world scenario and the use of Proteus for the simulation and testing made it easy to debug.

References

- [1] "What is a Quality Management System (QMS)? | ASQ," Asq.org, 2015. https://asq.org/quality-resources/quality-management-system?srltid=AfmBOopXWe07-mnGFOHn_rTDuPyoRmaeZ22rhX0RtpEhDKZfICGIQzEf (accessed Nov. 12, 2024).
- [2] "What is a Quality Management System (QMS)?," SimplerQMS, 2023. <https://simplerqms.com/quality/qms/what-is/> (accessed Nov. 13, 2024).
- [3] "Point to Point Communication Comprehensive Guide," Sanny Telecom, Apr. 27, 2024. <https://www.sannytelecom.com/point-to-point-communication-comprehensive-guide/> (accessed Nov. 11, 2024).
- [4] "Differences between Point-to-Point and Multi-point Communication," GeeksforGeeks, May 10, 2019. <https://www.geeksforgeeks.org/differences-between-point-to-point-and-multi-point-communication/> (accessed Nov. 13, 2024).
- [5] "What is multi-master replication and why it matters at scale," Arpit Bhayani. <https://arpitbhayani.me/blogs/multi-master-replication/> (accessed Nov. 13, 2024).

JS DIS assignment

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

69/100

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

JACOB SMELT**68.75****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
72%

18**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 but does not acknowledge that both pics speak to memory IC in multimaster mode and not directly to each other
62%

9.3

SKILLS FOR PROFESSIONAL PRAC.



The report is well presented with sections to explain the code and include a limited number of references, and some very good graphics and a good flowchart. 69%

10.35