

Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David



L5 – Distributed Intelligence Systems (ACDA5018)

A Simple Token Based Distributed Network

David Mathew Lewis

2219439

Electrical and Electronic Engineering

Table of Contents

1.0 Introduction	4
2.0 Research.....	4
2.1 Point-to-Point Transmission	4
2.2 Point-to-Multipoint Transmission.....	4
2.3 Multi-Master Transmission	5
3.0 Design.....	5
3.1 Overview of System Architecture	5
3.2 Hardware Configuration	5
3.3 RS232 Communication Setup.....	5
4.0 Development.....	6
4.1 Overview	6
4.2 Program Structure.....	6
4.3 Code Explanation & Screenshots	6
Button Detection and Toggling (U2)	6
Control Byte Construction and Transmission (U2)	7
UART Reception and LED Control (U1)	7
Timer-Based Flashing for Indicators (U1).....	7
Hazard Priority over Indicators (U1)	8
4.4 Compilation and Testing	8
4.5 Program Flowchart (U1 & U2 Combined)	9
5. Analysis	9
5.1 System Operation	9
5.2 Flow and Modularity	10
5.3 Quality and Safety Considerations.....	10
6. Conclusion.....	10
7. References	11
8. Appendices.....	11
U1 Code.....	17
U2 Code.....	18

Table of Figures

Figure 1 - U2 Button Detection	7
Figure 2 - U2 Control Byte.....	7
Figure 3 - U1 Reception and LED Control.....	7
Figure 4 - Indicator Flashing	7
Figure 5 - Hazard Priority (1)	8
Figure 6 - Hazard Priority (2)	8
Figure 7 - Program Flowchart	9
Figure 8 - Proteus Schematic	11
Figure 9 - Headlights On.....	12
Figure 10 - Left Indicator On	13
Figure 11 - Right Indicator On	14
Figure 12 - Hazard Lights On	15
Figure 13 - Fog Lights On.....	16

1.0 Introduction

The implementation of distributed systems in automotive applications plays a pivotal role in enhancing safety, modularity, and system reliability. This assignment involved the design and simulation of a simple token-based distributed control system using two PIC18F4550 microcontrollers. The goal was to control lighting functions for one corner of a motor vehicle, including headlights, indicators, hazard lights, and fog lights.

The system was designed with a clear division of responsibilities between two microcontrollers:

U2: Functioned as the input controller, reading button states from PORTA and constructing a control byte to represent the status of each lighting function.

U1: Acted as the output controller, receiving the control byte and activating the appropriate LEDs on PORTB.

Communication between U1 and U2 was established using RS232-style serial communication implemented via the internal UART modules of the PIC18F4550s. The serial data was transmitted using 8-bit, 9600 baud asynchronous communication over a single TX-to-RX line. This simplified implementation mimics standard RS232 behaviour, allowing lightweight serial communication without additional physical transceivers.

The system was implemented and tested in Proteus Design Suite using CCS C Compiler. Functionality such as toggle-based control, timed flashing, and hazard light override for indicators was successfully implemented and validated in simulation.

2.0 Research

2.1 Point-to-Point Transmission

A point-to-point transmission system involves a dedicated direct link between two devices, allowing exclusive communication without interference from other nodes. In this setup, data flows from a single sender to a single receiver, offering high reliability and simple implementation. Point-to-point links are common in early serial communications, such as RS-232 systems [2], where only two devices exchange information directly.

Although simple, point-to-point transmission is not scalable for complex systems with multiple devices, as each additional connection requires its own physical wiring.

2.2 Point-to-Multipoint Transmission

A point-to-multipoint transmission system allows one device (master) to communicate with multiple other devices (slaves) over a shared communication medium. Only one device typically transmits at a time, while multiple receivers listen. This configuration is used in systems like RS-485 networks or broadcast Ethernet communication [3].

Point-to-multipoint improves scalability compared to point-to-point, reducing wiring complexity and allowing centralised control. However, communication must be carefully managed to avoid data collisions, usually through protocols like polling or token passing.

2.3 Multi-Master Transmission

A multi-master transmission system permits multiple devices to initiate communication on the shared network bus. This model is used in distributed, decentralized systems where there is no single controlling master device. A common example is the Controller Area Network (CAN) bus, widely used in automotive systems [1].

Multi-master networks enhance fault tolerance and flexibility, as any device can transmit when necessary. Arbitration mechanisms are essential to prevent communication conflicts. In CAN, this is achieved through priority-based bitwise arbitration, ensuring that the highest-priority message wins access to the bus without delay [1].

3.0 Design

3.1 Overview of System Architecture

The project utilized a distributed two-processor architecture based on PIC18F4550 microcontrollers. The architecture was divided into two main units:

U2 (Input Controller):

Responsible for reading button inputs and transmitting a control byte representing the lighting status.

U1 (Output Controller):

Responsible for receiving the control byte and controlling the corresponding LEDs for headlights, indicators, hazard lights, and fog lights.

Communication between U1 and U2 was established using a UART (Universal Asynchronous Receiver Transmitter) interface at a baud rate of 9600. UART was selected for its simplicity, reliability, and direct support by the PIC18F4550 hardware [4].

The system was designed and tested using Proteus Design Suite for simulation, and all microcontroller programming was done in C using the CCS C Compiler.

3.2 Hardware Configuration

Each PIC microcontroller was configured with internal 8 MHz oscillators. Button inputs and LED outputs were mapped to specific ports as shown below:

<u>Function</u>	<u>Button</u>	<u>Pin</u>	<u>Port</u>
Headlights	RA0	Pin 2	PORTA
Left Indicator	RA1	Pin 3	PORTA
Right Indicator	RA2	Pin 4	PORTA
Hazard Lights	RA3	Pin 5	PORTA
Fog Lights	RA4	Pin 6	PORTA

Table 1 - Table of I/O

3.3 RS232 Communication Setup

Both microcontrollers were configured to use RS232-style asynchronous serial communication via their internal UART modules. Communication was established at:

Baud rate: 9600

Data bits: 8

Stop bits: 1

Parity: None

The RS232 connection allowed U2 to transmit a single control byte representing the current lighting state. U1 continuously monitored its receive line (RC7) for incoming data and updated its LED outputs accordingly. Although the communication occurred at logic voltage levels (0–5V), the configuration reflects the RS232 standard in terms of framing and protocol [4], [5].

4.0 Development

4.1 Overview

The development process involved writing two separate C programs using the CCS C Compiler for the PIC18F4550 microcontrollers. The microcontrollers were programmed to operate as two independent modules: an input controller (U2) and an output controller (U1). The firmware was developed, compiled, and tested within the Proteus Design Suite, where the full system was simulated, including button inputs, UART communication, and LED outputs.

4.2 Program Structure

U2 – Input Controller

The firmware on U2 was responsible for:

Reading five push buttons connected to PORTA.

Toggling the state of each lighting function when a button was pressed.

Building a compact 8-bit control byte representing the current lighting state.

Transmitting the control byte over RS232 using the internal UART module.

U1 – Output Controller

The firmware on U1 handled:

Receiving the control byte via RS232.

Decoding each bit to control individual LEDs connected to PORTB.

Flashing the indicator LEDs using a timer interrupt.

Giving hazard lights priority over left and right indicators.

The control logic was based on binary flag toggling and bitwise manipulation, a common approach in embedded systems for efficient status tracking and serial communication [6].

4.3 Code Explanation & Screenshots

Button Detection and Toggling (U2)

This code checks which button was pressed and toggles the associated flag. Each flag represents a lighting function, and the change is only registered on a rising edge.

```

while (TRUE) {
    int btn = input_a() & 0x1F;    // Read RA0-RA4 only
    int new_btn = btn & (~prev_btn); // Detect newly pressed buttons
    int changed = 0;              // Flag to check if any button changed

```

Figure 1 - U2 Button Detection

This section prevents multiple toggles from a single press by using edge detection logic. If a button is pressed (going from 0 to 1), it flips the corresponding flag (e.g. headlights ON → OFF → ON).

Control Byte Construction and Transmission (U2)

```

void send_control_byte() {
    unsigned int8 control_byte = 0;    // Control byte initialised
    if (headlights) control_byte |= 0b00000001;
    if (left_ind)   control_byte |= 0b00000010;
    if (right_ind)  control_byte |= 0b00000100;
    if (hazard)     control_byte |= 0b00001000;
    if (fog)        control_byte |= 0b00010000;

    putc(control_byte); // Send control byte via UART
}

```

Figure 2 - U2 Control Byte

The control byte is built by combining 5 flags into a single 8-bit value. Each bit position corresponds to a specific function. This byte is then transmitted to U1 via RS232 using `putc()`. The `putc()` function from the CCS C compiler directly transmits the 8-bit value using RS232 protocol conventions, which are suitable for low-bandwidth embedded systems [5].

UART Reception and LED Control (U1)

```

while (TRUE) {
    if (kbhit()) { // Check if UART recieved a byte
        control_byte = getc(); // Read control byte
    }
    update_leds(); // Update LEDs according to control byte
    delay_ms(50); // Short refresh delay
}

```

Figure 3 - U1 Reception and LED Control

U1 continuously checks if a byte has been received over RS232. When it detects a new byte (`kbhit()`), it reads it and stores it in `control_byte`. This value is then used to control all LEDs.

Timer-Based Flashing for Indicators (U1)

```

void TIMER1_isr() {
    counter++;
    if (counter >= 1) { // 500ms at 50ms Loop
        flash_state = !flash_state;
        counter = 0;
    }
}

```

Figure 4 - Indicator Flashing

Timer1 toggles a boolean variable `flash_state` every 500ms. This variable is checked when deciding whether to show/hide the indicator LEDs.

Hazard Priority over Indicators (U1)

```
// Flashing Hazard (overrides indicators)
if (bit_test(control_byte, 3)) {
    if (flash_state) {
        output_high(PIN_B2); // Left
        output_high(PIN_B3); // Right
    }
} else {
    // Flashing Left Indicator
    if (bit_test(control_byte, 1) && flash_state) {
        output_high(PIN_B2);
    }
}
```

Figure 5 - Hazard Priority (1)

```
// Flashing Right Indicator
if (bit_test(control_byte, 2) && flash_state) {
    output_high(PIN_B3);
}
}
```

Figure 6 - Hazard Priority (2)

This logic ensures hazard lights override individual indicators. If the hazard bit is active, both LEDs flash together. Otherwise, left and right indicators flash independently based on their flags.

4.4 Compilation and Testing

The final .c files were compiled into .hex files using the CCS C Compiler and loaded into the respective microcontrollers in Proteus. Testing involved:

- Pressing individual buttons in Proteus to simulate user input
- Monitoring the response via LED toggling and flashing
- Verifying hazard light override behavior

All functions operated as intended with reliable RS232 communication between U2 and U1. The system was tested entirely in software simulation using Proteus, which is commonly used for embedded system prototyping and supports real-time UART interaction [7].

4.5 Program Flowchart (U1 & U2 Combined)

The following flowchart illustrates the interaction between the two microcontrollers in the system. It shows the sequential processes carried out by each unit: U2 continuously monitors button inputs, encodes their states into a control byte, and transmits this over RS232. U1 receives the control byte, decodes the lighting commands, and controls the relevant LEDs accordingly. A timer interrupt on U1 manages the flashing logic for indicators and hazard lights. This flowchart provides a visual summary of the overall program logic and real-time communication between the distributed components.

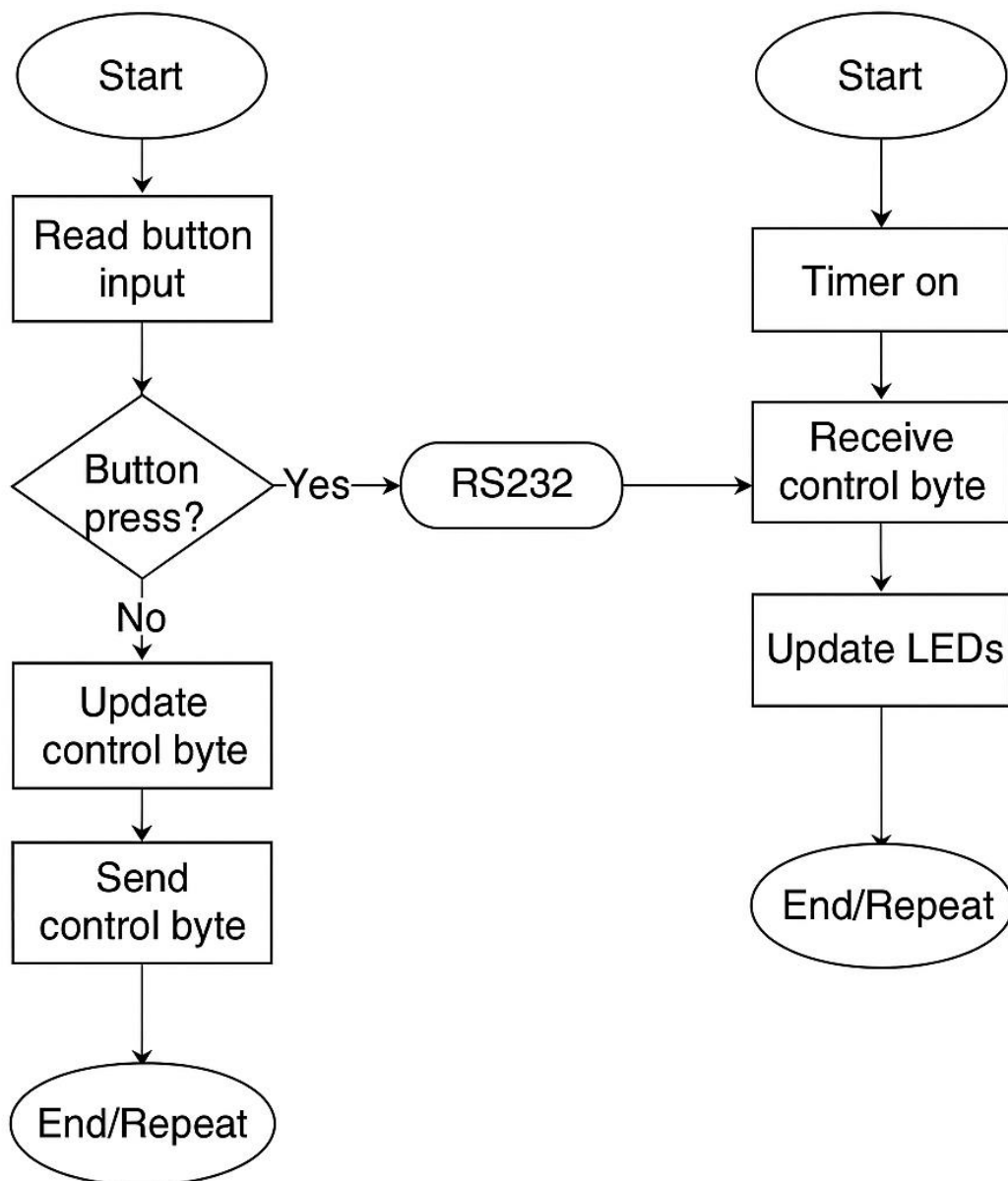


Figure 7 - Program Flowchart

5. Analysis

5.1 System Operation

The distributed lighting system functioned exactly as intended during simulation in Proteus. The U2 microcontroller reliably captured user input through push buttons connected to PORTA. Button

presses were processed using rising edge detection to allow toggle-based control of each lighting function. The current states were encoded into a single 8-bit control byte and transmitted to U1 using RS232-style serial communication.

On the receiving end, U1's firmware continuously monitored for incoming control bytes. Once received, the control byte was decoded and used to update the states of six LEDs connected to PORTB. A hardware-based Timer1 interrupt on U1 was used to generate a flashing state (flash_state) every 500 ms. This mechanism allowed the left and right indicators, as well as hazard lights, to toggle on and off with consistent timing.

Importantly, the hazard lights were given priority: if the hazard bit was set in the control byte, the system would override individual indicator commands and flash both indicator LEDs together. This emulates real-world automotive lighting behaviour and ensures the system adheres to typical safety expectations.

5.2 Flow and Modularity

The system demonstrated clear modularity and separation of responsibilities between the two microcontrollers. U2 operated independently of U1, only responsible for collecting input and sending a status byte. U1 acted purely on received data, meaning that changes in user input logic (e.g., adding extra buttons) could be made without modifying the LED control code.

The use of a single-token control byte allowed simple and fast serial communication, reducing the number of physical wires needed and avoiding the complexity of polling or interrupt-driven message handling. The system looped smoothly with minimal latency due to the short transmission time of a single byte and the low computational overhead.

5.3 Quality and Safety Considerations

The system design followed key embedded programming best practices:

- Use of bitwise operations to compactly represent and interpret system states [6]
- Timer-based interrupts to implement consistent and non-blocking LED flashing [5]
- Use of RS232 protocol for simple and robust point-to-point communication between devices [4]

To ensure safety, the flashing logic for hazard lights overrides the indicators, consistent with real vehicle requirements. The modular design also allows fault isolation: if U2 fails, U1 will simply hold its last received state. In a real system, additional safety mechanisms such as timeout detection or CRC error checking could be added to improve reliability and fault tolerance.

6. Conclusion

This project successfully implemented a simple RS232-style distributed lighting control system using two PIC18F4550 microcontrollers. One unit handled user input via push buttons, while the other controlled LEDs based on a received control byte.

All system functions operated as intended: lights toggled correctly, indicators and hazards flashed at regular intervals, and hazard lights correctly overrode individual indicators. The use of UART-based communication, timer interrupts, and bitwise logic followed standard embedded system practices.

The system is modular, easy to expand, and reflects real-world automotive control strategies, making it a practical example of distributed embedded design.

7. References

- [1] R. Bosch GmbH, CAN Specification Version 2.0, Robert Bosch GmbH, 1991.
- [2] W. Stallings, Data and Computer Communications, 7th ed., Upper Saddle River, NJ, USA: Prentice Hall, 2004.
- [3] T. U. Zwick, Introduction to Digital Communication, Wiley, 2021.
- [4] M. Predko, Programming and Customizing PICmicro Microcontrollers, 3rd ed., McGraw-Hill, 2000.
- [5] J. Axelson, Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems, 2nd ed., Lakeview Research, 2007.
- [6] T. Wilmshurst, Designing Embedded Systems with PIC Microcontrollers, 2nd ed., Newnes, 2009.
- [7] Y. Tian, "Simulation of Embedded Systems using Proteus," International Journal of Engineering Research and Technology, vol. 5, no. 3, pp. 245–249, 2016.

8. Appendices

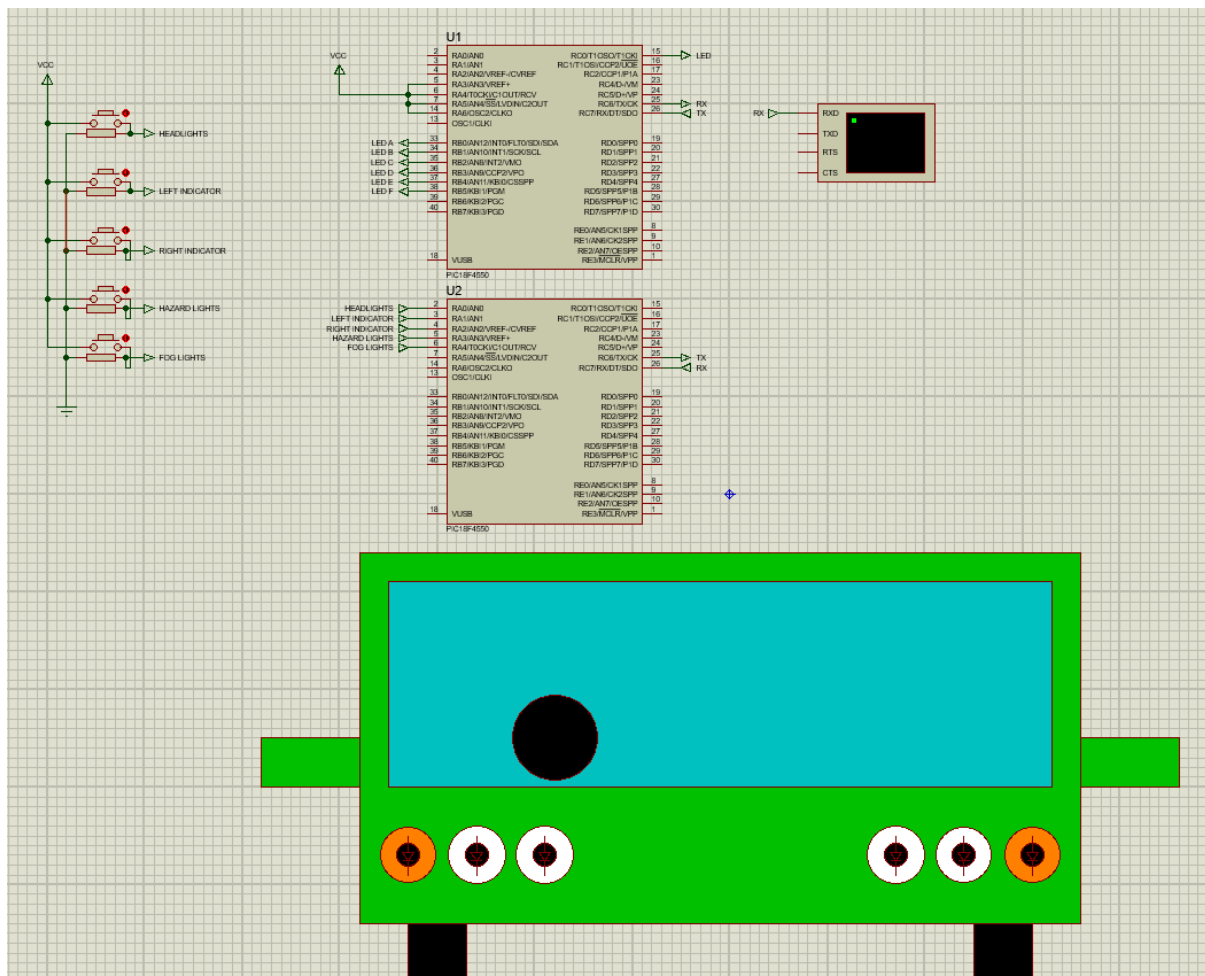


Figure 8 - Proteus Schematic

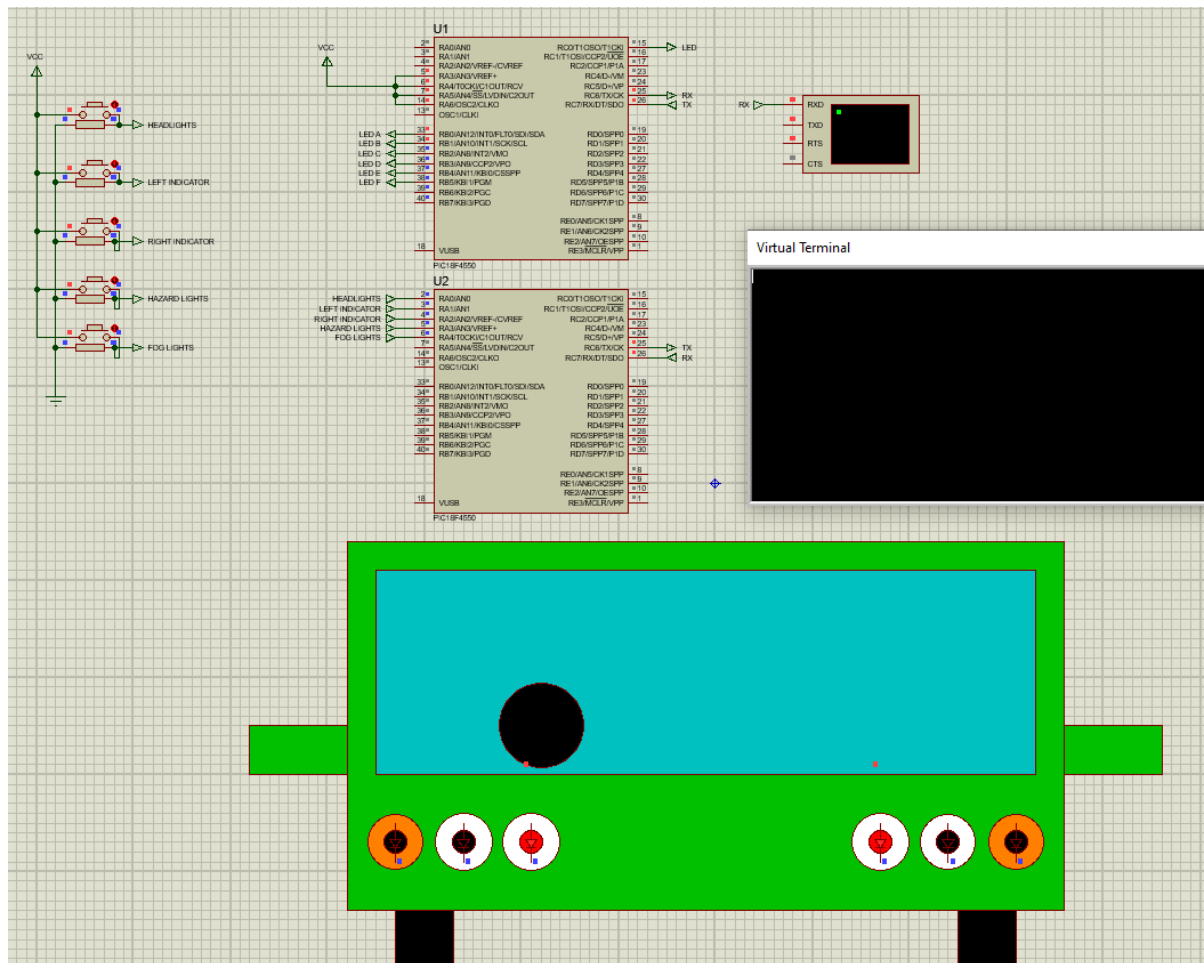
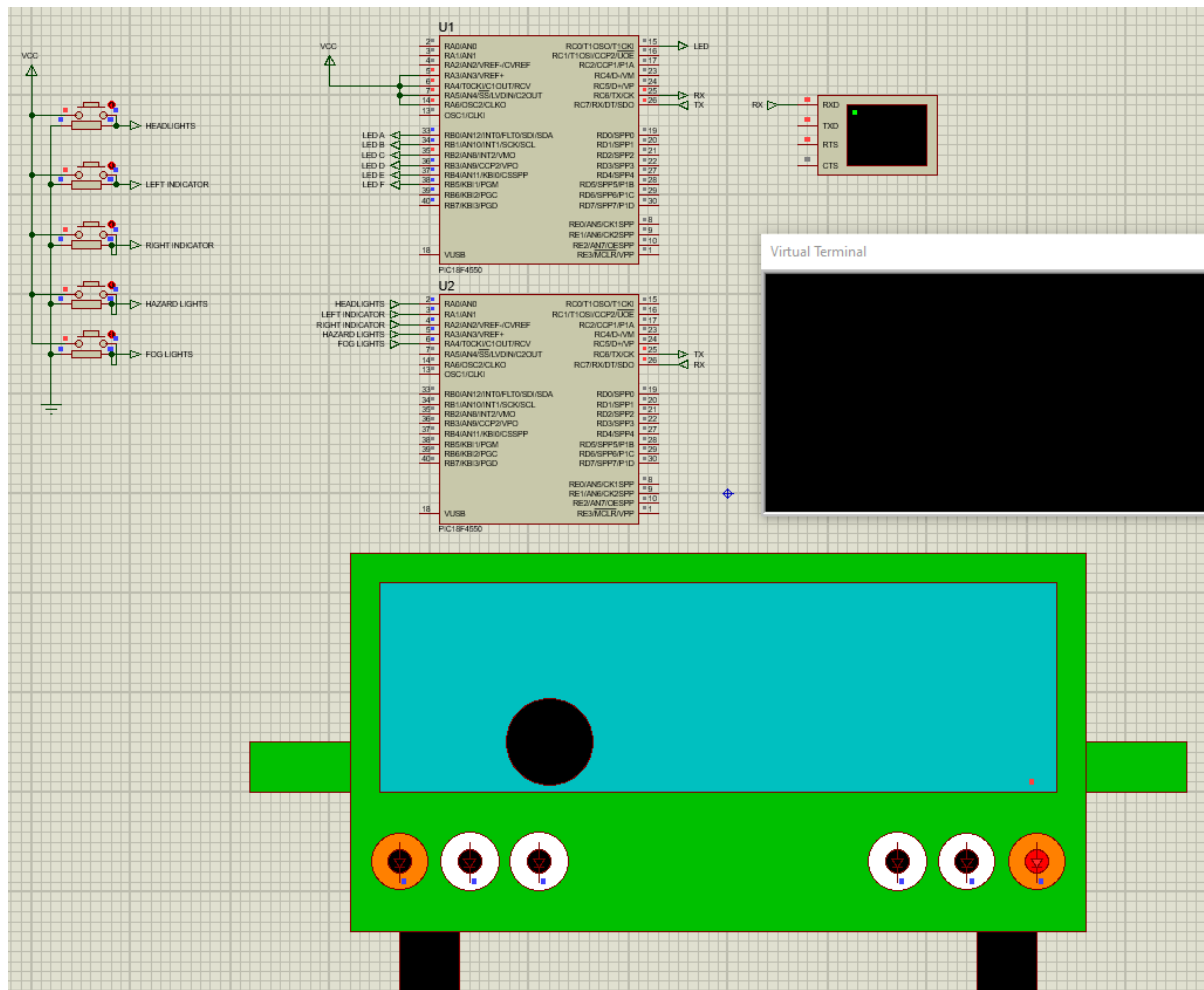


Figure 9 - Headlights On



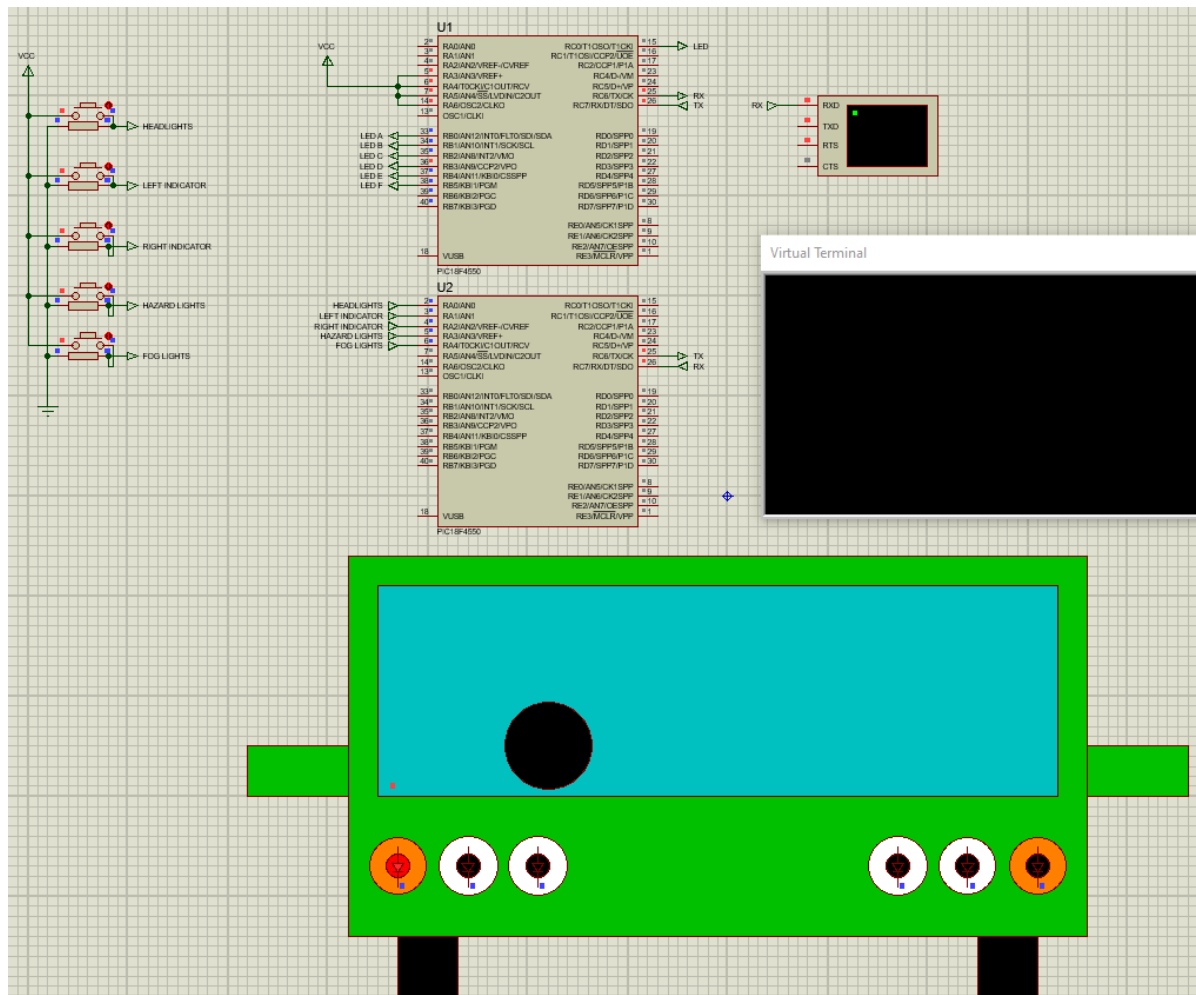


Figure 11 - Right Indicator On

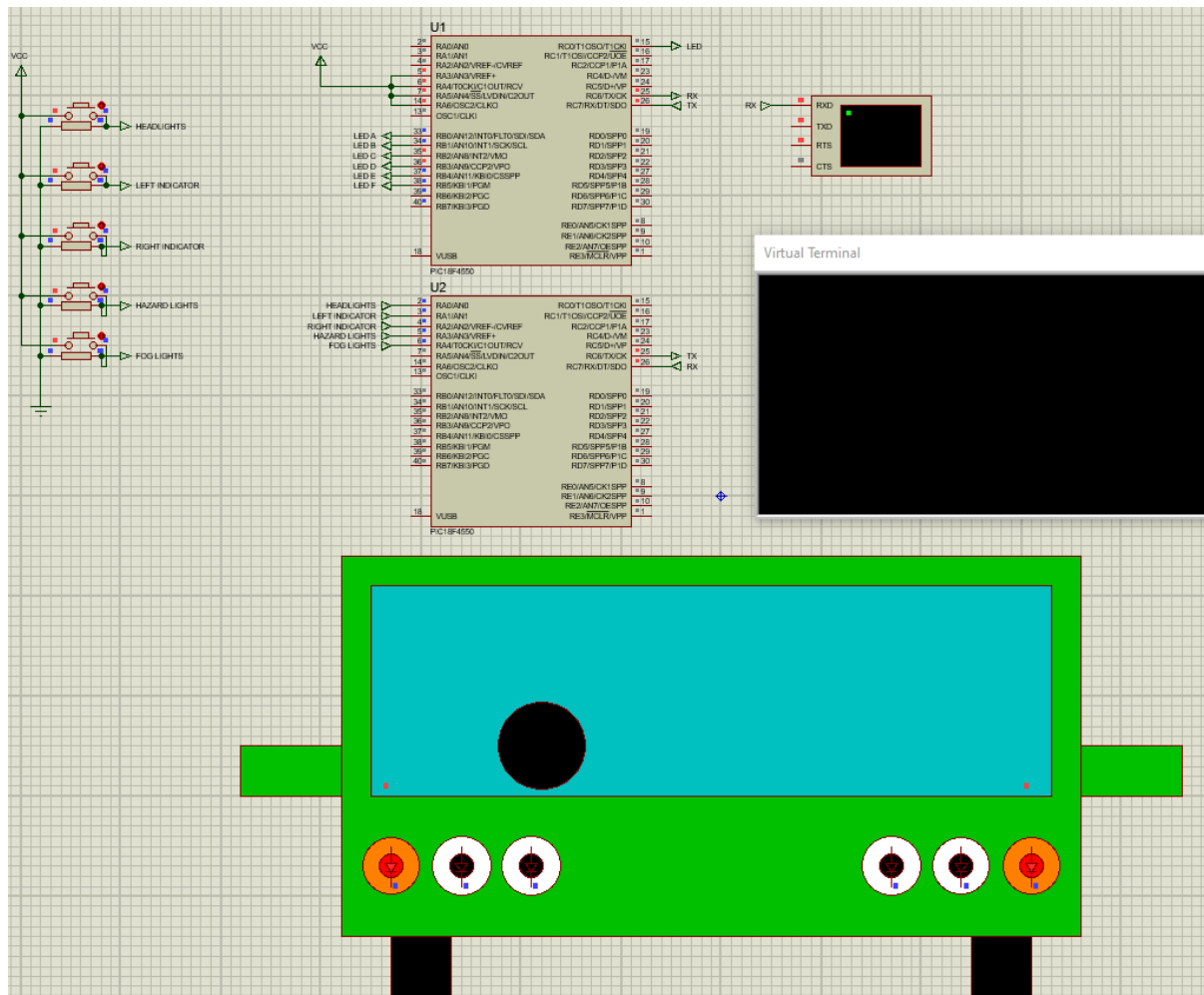


Figure 12 - Hazard Lights On



Appendices

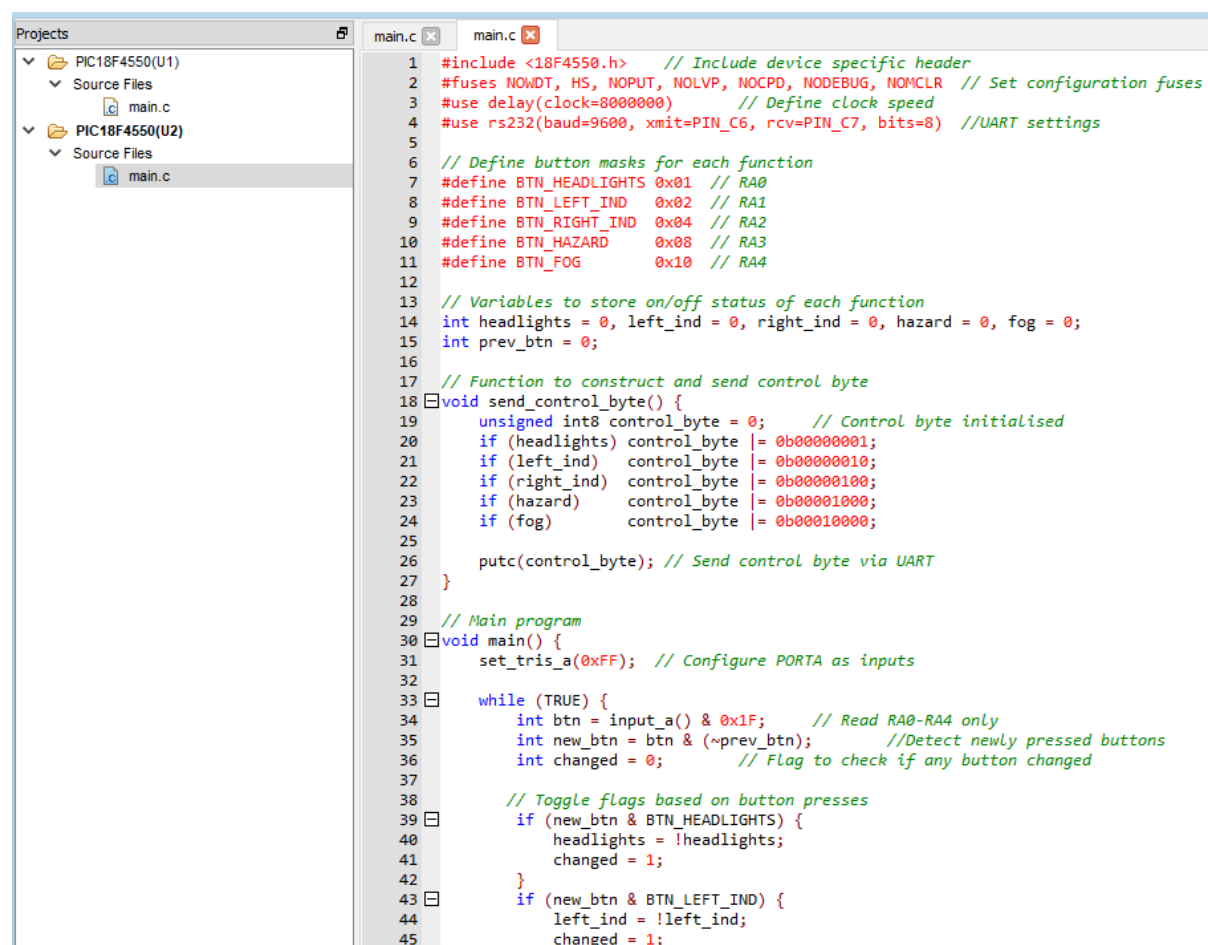
U1 Code

```

13 void TIMER1_isr() {
14     counter++;
15     if (counter >= 1) { // 500ms at 50ms loop
16         flash_state = !flash_state;
17         counter = 0;
18     }
19 }
20
21 //Function to configure Timer1
22 void setup_flashing_timer() {
23     setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
24     enable_interrupts(INT_TIMER1);
25     enable_interrupts(GLOBAL);
26 }
27
28 // Update LEDs based on control byte
29 void update_leds() {
30     // Clear all LEDs first
31     output_low(PIN_B0); output_low(PIN_B1); // Headlights
32     output_low(PIN_B2); output_low(PIN_B3); // Indicators
33     output_low(PIN_B4); output_low(PIN_B5); // Fog Lights
34
35     // Turn ON Headlights
36     if (bit_test(control_byte, 0)) {
37         output_high(PIN_B0); output_high(PIN_B1);
38     }
39
40     // Turn ON Fog Lights
41     if (bit_test(control_byte, 4)) {
42         output_high(PIN_B4); output_high(PIN_B5);
43     }
44
45     // Flashing Hazard (overrides indicators)
46     if (bit_test(control_byte, 3)) {
47         if (flash_state) {
48             output_high(PIN_B2); // Left
49             output_high(PIN_B3); // Right
50         }
51     } else {
52         // Flashing Left Indicator
53         if (bit_test(control_byte, 1) && flash_state) {
54             output_high(PIN_B2);
55         }
56
57         // Flashing Right Indicator
58         if (bit_test(control_byte, 2) && flash_state) {
59             output_high(PIN_B3);
60         }
61     }
62 }
63
64 // Main program
65 void main() {
66     set_tris_b(0x00); // Set Port B as outputs for LEDs
67     setup_flashing_timer(); // Start timer for flashing
68
69     while (TRUE) {
70         if (kbhit()) { // Check if UART recieved a byte
71             control_byte = getc(); // Read control byte
72         }
73         update_leds(); // Update LEDs according to control byte
74         delay_ms(50); // Short refresh delay
75     }
76 }

```

U2 Code



```
1 #include <18F4550.h> // Include device specific header
2 #fuses NOWDT, HS, NOPUT, NOLVP, NOCPD, NODEBUG, NOMCLR // Set configuration fuses
3 #use delay(clock=8000000) // Define clock speed
4 #use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, bits=8) //UART settings
5
6 // Define button masks for each function
7 #define BTN_HEADLIGHTS 0x01 // RA0
8 #define BTN_LEFT_IND 0x02 // RA1
9 #define BTN_RIGHT_IND 0x04 // RA2
10 #define BTN_HAZARD 0x08 // RA3
11 #define BTN_FOG 0x10 // RA4
12
13 // Variables to store on/off status of each function
14 int headlights = 0, left_ind = 0, right_ind = 0, hazard = 0, fog = 0;
15 int prev_btn = 0;
16
17 // Function to construct and send control byte
18 void send_control_byte() {
19     unsigned int8 control_byte = 0; // Control byte initialised
20     if (headlights) control_byte |= 0b00000001;
21     if (left_ind) control_byte |= 0b00000010;
22     if (right_ind) control_byte |= 0b00000100;
23     if (hazard) control_byte |= 0b00001000;
24     if (fog) control_byte |= 0b00010000;
25
26     putc(control_byte); // Send control byte via UART
27 }
28
29 // Main program
30 void main() {
31     set_tris_a(0xFF); // Configure PORTA as inputs
32
33     while (TRUE) {
34         int btn = input_a() & 0x1F; // Read RA0-RA4 only
35         int new_btn = btn & (~prev_btn); // Detect newly pressed buttons
36         int changed = 0; // Flag to check if any button changed
37
38         // Toggle flags based on button presses
39         if (new_btn & BTN_HEADLIGHTS) {
40             headlights = !headlights;
41             changed = 1;
42         }
43         if (new_btn & BTN_LEFT_IND) {
44             left_ind = !left_ind;
45             changed = 1;
46         }
```

```
40     }
47     if (new_btn & BTN_RIGHT_IND) {
48         right_ind = !right_ind;
49         changed = 1;
50     }
51     if (new_btn & BTN_HAZARD) {
52         hazard = !hazard;
53         changed = 1;
54     }
55     if (new_btn & BTN_FOG) {
56         fog = !fog;
57         changed = 1;
58     }
59
60     // Send control byte only if something has changed
61     if (changed) {
62         send_control_byte();
63     }
64
65     prev_btn = btn; // Update previous button state
66     delay_ms(50); // Short debounce
67 }
68 }
```