

Distributed Intelligence Systems

by Daniel Bridgewater

Submission date: 15-Nov-2024 02:17PM (UTC+0000)

Submission ID: 242799574

File name: 42541_Daniel_Bridgewater_Distributed_Intelligence_Systems_523660_1107158909.docx (652.29K)

Word count: 2341

Character count: 12790

DISTRIBUTED INTELLIGENCE SYSTEMS

ACDA5018

11/11/2024

Daniel Bridgewater

Contents

Abstract 2

Introduction / Research 2

 Point to point 2

 Point to multipoint 2

 Multi-master 2

Design..... 3

Development..... 5

Analysis 8

Conclusion..... 11

References..... 11

Abstract

The aim of this report is to investigate the suitability of Multi-Master communication architecture for controlling hardware in a hypothetical automotive system, through using two PIC18F2525 microcontrollers sharing a FM24C04B FRAM (Ferromagnetic RAM) data storage device to exchange information. The importance of Quality Management Systems is also highlighted, for its suitability in guiding development of a safety-compliant product. FRAM was chosen for its high write durability and speed, which are both important traits for automotive systems. The system accepts inputs from the user with button presses and commands the lighting subsystem accordingly via distributed control. To ensure reliability, bus arbitration and command acknowledgements were employed. A simulated environment was used to rapidly prototype the system, taking advantage of advanced debugging features and demonstrating the design functions properly, demonstrating the potential for Multi-Master I2C devices within an automobile.

Introduction

The I2C bus is principally designed for one microcontroller to communicate with one or more peripheral devices

Quality Management Systems are essential bureaucratic processes to ensure effective use of resources during development through management ensuring that stakeholder requirements are being met, while the delivered product is assured to a good standard of quality through regular reviews and continual improvement of associated processes. The processes are also designed to consider long and short-term effects of any decisions, balancing cost with suitability and sustainability. Without these processes, there would be no standard to ensure products comply with safety regulations, possibly leading to fatalities if the product malfunctions or fails.

Point to point

In a point-to-point transmission system, each node has a direct connection to all other node it communicates with, and messages are never passed between nodes. If all nodes need connections to all other nodes, then the topology is a fully connected network, and the number of connections follow the Triangular Number sequence [1, p. 159] according to the following equation:

$$\frac{n(n-1)}{2}$$

Equation 1: nth Triangular Number [1, p. 154]

Point to multipoint

Often a point-to-point network is impractical to implement, as the number of connections grows exponentially it can quickly become expensive or impossible to create all the required links. The solution is to have at least one intermediary node responsible for passing data between other nodes. While this substantially reduces the number of connections required, it either requires sending the data everywhere (broadcasting) or coordination to identify which path leads to the recipient device.

Multi-master

It is possible to have a configuration of multi-point connected nodes where some nodes can be “masters”, capable of addressing any of the other “slave” nodes. Slaves can only respond to requests

from master nodes and cannot initiate communication autonomously. I2C supports having multiple masters, wherein each can address any of the connected slaves, but not directly to other masters. For master-master communication, there must be some in-between slave that can be used to share data between the two master nodes.

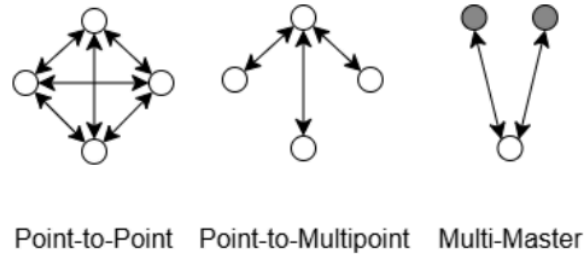


Figure 1: Diagram of different transmission system configurations

Design

The author chose the PIC18F2525 microcontroller as they have 3 IO ports, providing sufficient connectivity for the task and allowing for future expansion if more features need to be added to future revisions. One PIC is responsible for sensing user button inputs, henceforth referred to as the User Input PIC, storing commands into the shared slave FRAM using I2C. The other PIC, henceforth referred to as the Light Controller PIC, turns the lights on one corner of the vehicle on and off according to commands stored in the FRAM from the User Input PIC.

The aforementioned FRAM device is the FM24C04B. According to the datasheet [2], '[this] is a 4-Kbit non-volatile memory employing an advanced ferromagnetic process ... capable of supporting 10^{14} read/write cycles'. This is far greater than that of an EEPROM, ensuring the system will remain reliable for many years into the future. A conventional EEPROM additionally requires multiple milliseconds to store incoming data, while F-RAM writes are instantaneous, reducing the latency between the user pressing a button and the light turning on, potentially improving safety.

The only downside to FRAM is cost; an EEPROM typically costs £0.26 [3] while an equivalent FRAM device costs £1.15 [4]. However, the author contends this is a reasonable price considering the reliability improvements and certification for use in an automotive context (AEC-Q100 grade 3 qualified [2]).

The initial system design was done in simulation to enable rapid prototyping without incurring costs associated with physical manufacture. In Figure 2: Simulation Circuit Diagram, 4 standard normally-open push button simulation primitives are in the top-left for user interaction, connected to the top User Input PIC via Port B. For simulating the lights, there are an equal number of LEDs in the bottom-left connected to the bottom Light Controller PIC, each connected to a current-limiting resistor and a common-cathode. To keep the Light Controller code readable, lights which do not need to blink are connected to Port B, while the indicator light needs to regularly blink and therefore is connected to Port A.

The PICs share an I2C bus with the FRAM to exchange information about which buttons are pressed/released, and to ensure they both don't access the FRAM simultaneously, there's an additional bus arbitration signal. This is also called a handshake signal. Normally the signal is kept at 5V through a 10K pull-up resistor to indicate the bus is idle, until a PIC wishes to use the bus, wherein it discharges the line by connecting it to ground. Should the other PIC attempt to access the FRAM

while already in use, it must wait for the other PIC to finish its operation and signal such by disconnecting the bus arbitration signal from ground. The signal then returns to 5V, allowing the second PIC to pull it low and safely begin accessing the FRAM. Using bus arbitration in this manner is essential to avoid “collisions” where both PICs attempt to access the FRAM simultaneously, inadvertently distorting each other’s signals. It may even cause damage to the PICs through excessive current flow if one pulls the signal low and the other pulls it high.

The I2C bus also has 10K resistors for a similar purpose when the PICs communicate with the FRAM. To inspect the I2C bus for debugging/troubleshooting purposes, the author has attached an I2C debugger block provided by the simulation software. This provides detailed information about the signals sent back and forth by both PICs and FRAM, which was essential information for isolating issues in the PIC programming during development. This could have been much harder to do if development were taking place in the real world, as debuggers can be expensive to buy.

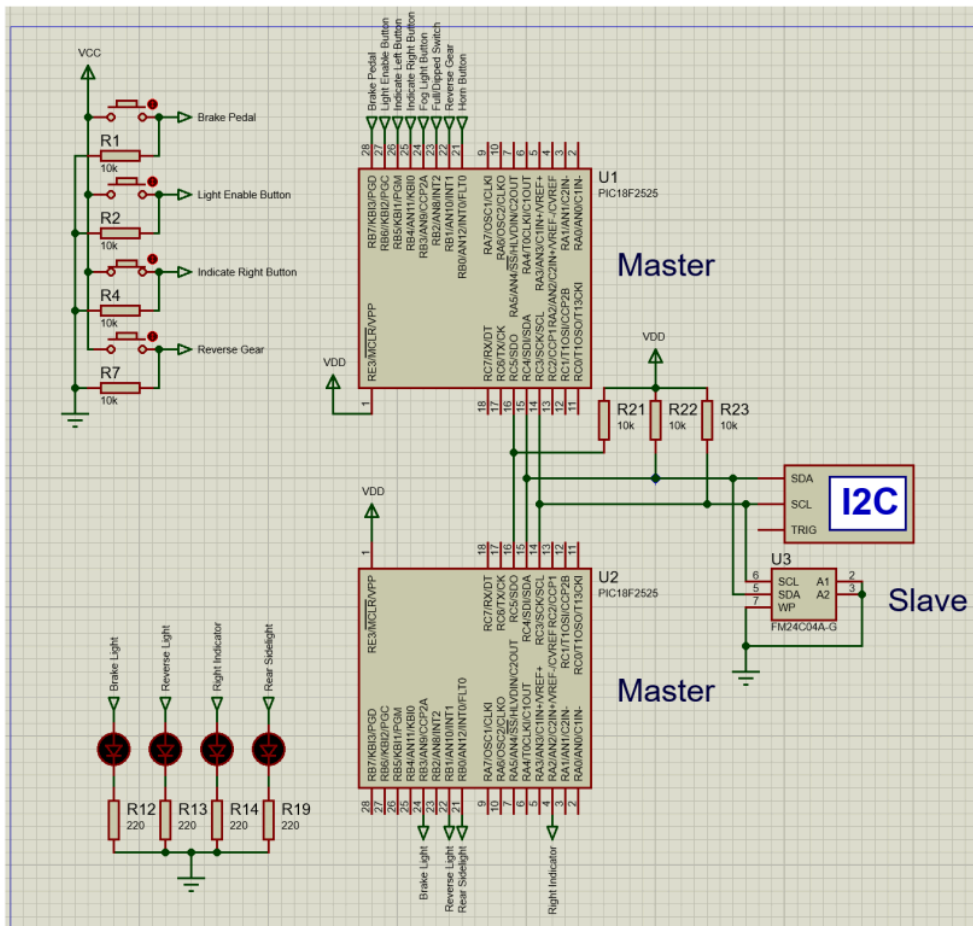


Figure 2: Simulation Circuit Diagram

Development

user_input.h

```
1 #include <18F2525.h>
2 #device ADC=10
3 #CASE // Enable case sensitivity
4
5 #FUSES NOWDT           //No Watch Dog Timer
6 #FUSES FCMEN           //Fail-safe clock monitor enabled
7 #FUSES IESO            //Internal External Switch Over mode enabled
8 #FUSES PUT             //Power Up Timer
9 #FUSES BROWNOUT        //Reset when brownout detected
10 #FUSES BORV21          //Brownout reset at 2.1V
11 #FUSES LPT1OSC         //Timer1 configured for low-power operation
12 #FUSES STVREN          //Stack full/underflow will cause reset
13 #FUSES NOPROTECT       //Code not protected from reading
14 #FUSES NOWRT           //Program memory not write protected
15 #FUSES NOEBTR          //Memory not protected from table reads
16
17 #use delay(internal=16MHz)
18 #use rs232(baud=9600,parity=N,xmit=PIN_C6,rcv=PIN_C7,bits=8,stream=PORT1,errors)
19 #use i2c(master, FAST, I2C1, stream=I2CPORT)
20 #define BUS_ARBITRATION PIN_C5
21
22 // Function definitions
23 void fram_write(char address, char data);
24 char fram_read(char address);
```

user_input.c

```
8 #include <user_input.h>
9
10 void main() {
11     setup_adc_ports(NO_ANALOGS, VSS_VDD);
12
13     unsigned int buttons_prev = 0;
14
15     while (TRUE) {
16         // Read current button input
17         unsigned int buttons = input_b();
18         // Check if any have changed
19         if (buttons != buttons_prev) {
20             // Identify which bits have flipped
21             unsigned int changed = buttons ^ buttons_prev;
22             // If a bit has flipped and it was previously on, it went from 1 to 0
23             unsigned int turned_off = changed & buttons_prev;
24             // Record current button state
25             buttons_prev = buttons;
26
27             // Test each bit starting from MSB
28             for (unsigned int i=0; i<=7; i++) {
29                 char mask = 0x80>>i;
30                 // Has this bit flipped?
31                 if (changed & mask) {
32                     char output = 'A'+i; // Calculate token
33                     // If the bit went Low...
34                     if (turned_off & mask) {
35                         // ... make token lowercase
36                         output |= 0x20;
37                     }
38                     // Prevent overwriting existing command
39                     while (fram_read(0)) delay_ms(20);
40
41                     //printf("%c", output);
42                     fram_write(0, output);
43                 }
44             }
45         }
46     }
47 }
48 }
```

```

50 // I2C Ferromagnetic-RAM read/write routines
51 void fram_write(char address, char data) {
52     while (!input(BUS_ARBITRATION));
53     output_low(BUS_ARBITRATION); // signal that the i2c bus is now in use
54     i2c_start();
55     i2c_write(0xa0); // write (indicated by 0) the word address to modify
56     i2c_write(address); // select word address
57     i2c_write(data); // send data to be stored
58     i2c_stop();
59
60     // Note: F-RAM stores the data instantaneously and therefore does not require a delay after writing
61     input(BUS_ARBITRATION); // allow other chip to use i2c
62 }
63
64 char fram_read(char address) {
65     while (!input(BUS_ARBITRATION));
66     output_low(BUS_ARBITRATION); // signal that the i2c bus is now in use
67     i2c_start();
68     i2c_write(0xa0); // write (indicated by 0) the word address to read
69     i2c_write(address); // select word address
70
71     i2c_start();
72     i2c_write(0xa1); // 1 to read data
73     char data = i2c_read(0); // Read reply from ram and don't acknowledge response to signal end of read
74     i2c_stop();
75     input(BUS_ARBITRATION); // allow other chip to use i2c
76
77     return data;
78 }
79

```

Both PIC programs include the fram_read and fram_write routines, but for brevity they are omitted from the light_controller code screenshots. The light_controller.h screenshot also only shows the changed section.

light_controller.h

```

22 // Bit definitions for pin mappings
23 #define BIT_BRAKE_LIGHTS (1<<3)
24 #define BIT_RIGHT_IND (1<<2)
25 #define BIT_REV_LIGHTS (1<<1)
26 #define BIT_SIDE_LIGHTS 1
27
28 // Token definitions
29 #define BRAKE_LIGHTS_ON 'A'
30 #define BRAKE_LIGHTS_OFF 'a'
31 #define SIDE_LIGHTS_ON 'B'
32 #define SIDE_LIGHTS_OFF 'b'
33 #define INDICATOR_RIGHT_ON 'D'
34 #define INDICATOR_RIGHT_OFF 'd'
35 #define REVERSE_LIGHTS_ON 'G'
36 #define REVERSE_LIGHTS_OFF 'g'

```

light_controller.c

```

8  #include <light_controller.h>
9
10
11 unsigned int light_state = 0;
12
13 int blinky; // Keeps track of blinking
14 // Interrupt routine for blinking indicator light
15 #INT_TIMER0
16 void Timer0_isr(void) {
17     blinky ^= 1; // Toggle blink
18     // Check whether the indicator should be blinking
19     // This can be extended to include multiple lights if needed
20     unsigned int out = light_state & BIT_RIGHT_IND;
21     // If blinky=1
22     output_a(blinky? out:0);
23     set_timer0(100); // Timer0 preload value
24     clear_interrupt(INT_TIMER0); // Clear Timer0 overflow bit
25 }
26
27 void main() {
28     // Configure timer 0 for indicator blinking
29     clear_interrupt(INT_TIMER0); // Clear Timer0 overflow bit
30     enable_interrupts(INT_TIMER0); // Enable Timer0 interrupt
31     enable_interrupts(GLOBAL); // Enable global interrupts
32     setup_timer_0(T0_INTERNAL | T0_DIV_32); // Timer0 configuration: internal clock source + 256 prescaler
33     set_timer0(100); // Timer0 preload value to increase blinking speed
34
35     setup_adc_ports(NO_ANALOGS, VSS_VDD);
36
37     unsigned int enable_lights = 0;
38
39     while (TRUE) {
40         // Read command from master
41         unsigned int command = 0;
42         do {
43             command = fram_read(0);
44             delay_ms(200); // Don't constantly read; allow some time for the other master to access the slave
45         } while (command == 0);
46         fram_write(0,0);
47
48         switch (command) {
49             case BRAKE_LIGHTS_ON:
50                 // Enable a single bit
51                 light_state |= BIT_BRAKE_LIGHTS;
52                 break;
53             case BRAKE_LIGHTS_OFF:
54                 // Mask out this single bit
55                 light_state &= ~BIT_BRAKE_LIGHTS;
56                 break;
57             case SIDELIGHTS_ON:
58                 light_state |= BIT_SIDELIGHTS;
59                 break;
60             case SIDELIGHTS_OFF:
61                 light_state &= ~BIT_SIDELIGHTS;
62                 break;
63             case INDICATOR_RIGHT_ON:
64                 light_state |= BIT_RIGHT_IND;
65                 break;
66             case INDICATOR_RIGHT_OFF:
67                 light_state &= ~BIT_RIGHT_IND;
68                 break;
69             case REVERSE_LIGHTS_ON:
70                 light_state |= BIT_REV_LIGHTS;
71                 break;
72             case REVERSE_LIGHTS_OFF:
73                 light_state &= ~BIT_REV_LIGHTS;
74                 break;
75         }
76
77         output_b(light_state);
78     }
79 }
80
81
82
83
84
85
86

```

Analysis

In the `user_input.h` file, various parameters are defined such as the device type, fuses and clock speed. Use of the I2C peripheral is enabled, specifying to use I2C1 located on Port C. Additionally, the pin used for bus arbitration pin is set to C5, and function definitions for FRAM reading/writing routines are declared.

In `user_input.c`, there is an integer for keeping track of how the buttons have been pressed and an infinite loop to continually check the button inputs. When there is a difference between the current button state and that previous recorded, it means at least one button was either pressed or released. The loop then stores this change, identifies which specific buttons states have changed, and determines if each button has been pressed or released. Then for each button that has changed, it computes a unique token representing a single change, where uppercase tokens indicate an input going from low to high and a lowercase letter indicates the input went from high to low. This token gets stored into the FRAM for the Light Controller PIC to process, but first the User Input PIC checks if there is already a token stored to avoid overwriting a token not yet acknowledged by the Light Controller PIC. Doing this ensures that tokens cannot be lost if multiple buttons states change simultaneously. Once the token is sent, the next token is computed if necessary, otherwise the loop returns to waiting for another button press.

To read and write data to the FRAM, two routines exist on both PICs; `fram_write` and `fram_read`. To ensure there aren't any bus collisions where both PICs attempt to access the FRAM simultaneously, a `BUS_ARBITRATION` pin is connected to both PICs for communicating when the other is using the bus. Each routine will check this pin and wait for it to become High, indicating the other PIC has concluded its action. Once this condition is met, the pin gets pulled Low to prevent the other PIC from interfering, then the I2C communication commences.

In `light_controller.h`, everything is the same to `user_input.h`, except for adding extra definitions for which Port bits control different lights, and human-readable identifiers for different tokens produced by the User Input PIC. This approach gives clear names for what would otherwise be 'magic values', greatly improving the code readability.

In `light_controller.c`, there is a timer interrupt routine alongside the infinitely-looping main function responsible for regularly blinking indicator lights. It monitors a global `light_state` variable to check if the right indicator has been commanded to turn on and therefore blink. There is a distinction between whether an indicator has been commanded to blink (`light_state`) versus whether it is currently on or off in a blink cycle. The blink cycle alternates continuously in a regular fashion by repeatedly applying an XOR to the lowest bit of the blinky variable with 1. 1 XOR 1 becomes 0, and 0 XOR 1 becomes 1. The `light_state` contains all lights, so it must be filtered to only contain bits representing lights which should be controlled by this routine, which is achieved by masking `light_state` with `BIT_RIGHT_IND`. Then, if the blinky variable is 1, either the mask result is output onto Port A, or the port is cleared.

Within the light controller main function, the first step sets up the aforementioned interrupt routine to regularly execute approximately twice a second to blink the indicator light. Then it enters an infinite loop to wait until a command appears in the FRAM slave, sent from the User Input PIC. Once this happens, the command is retrieved and then cleared from the FRAM to indicate that the token has been received, allowing the UIP to quickly store another token. The LCP then parses the received command token to determine which light it influences.

Each token causes a specific bit in the light_state variable to be set or cleared with bitwise operations. For example, 'B' corresponds to SIDELIGHTS_ON, so light_state will be OR'ed with BIT_SIDELIGHTS (0x02 or 1<<1), setting the second bit to 1. When 'b' is received, light_state will be AND'ed with the inverse of BIT_SIDELIGHTS (0xfd or 11111101) to set the second bit to 0. These operations are more complex than adding or subtracting BIT_SIDELIGHTS, but have the significant advantage of being robust; repeated additions or subtractions will alter other bits in light_state and cause undesired behaviour, while repeated bitwise operations can never affect other bits.

Once light_state has been appropriately adjusted, it is output onto Port B. However, this is only suitable for lights which do not need to blink, so the timer interrupt routine for blinking lights operates on Port A.

Figure 3: Flow diagram for PIC -PIC communications illustrates how the Light Controller PIC continuously checks for commands stored in the FRAM, how both PICs treat a nil-value 0 to mean there is no pending command, and how the command to turn on brake lights 'A' gets passed between the two PICs. If the User Input PIC were to immediately store the new command, it's possible a command would be missed if the Light Controller PIC hasn't yet read the FRAM contents. Acknowledging receipt of messages with 0 is a key part of the devised communication protocol. Currently only address #0 of the FRAM is utilised, but other addresses could be used to communicate with other Light Controller PICs, or to be reserved for indicating errors such as a lightbulb failure.

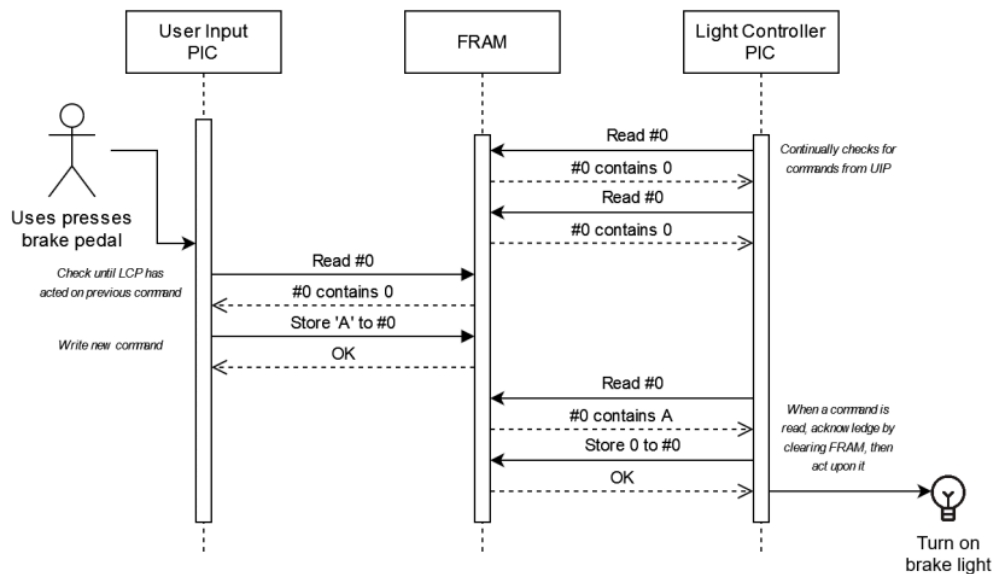


Figure 3: Flow diagram for PIC -PIC communications

Results

To test the system, the author pressed the simulation buttons and observed how the corresponding light illuminated in response. To verify the tokens were being sent properly, a diagnostic feature of the simulator was used to directly show the FRAM contents, which would not be possible in a real circuit. When the Light Enable button is pressed, in Figure 5, token 'B' (SIDELIGHTS_ON) is stored in the FRAM. A moment later, in Figure 5, this is detected by the LCP, acknowledged, and the light turns on.

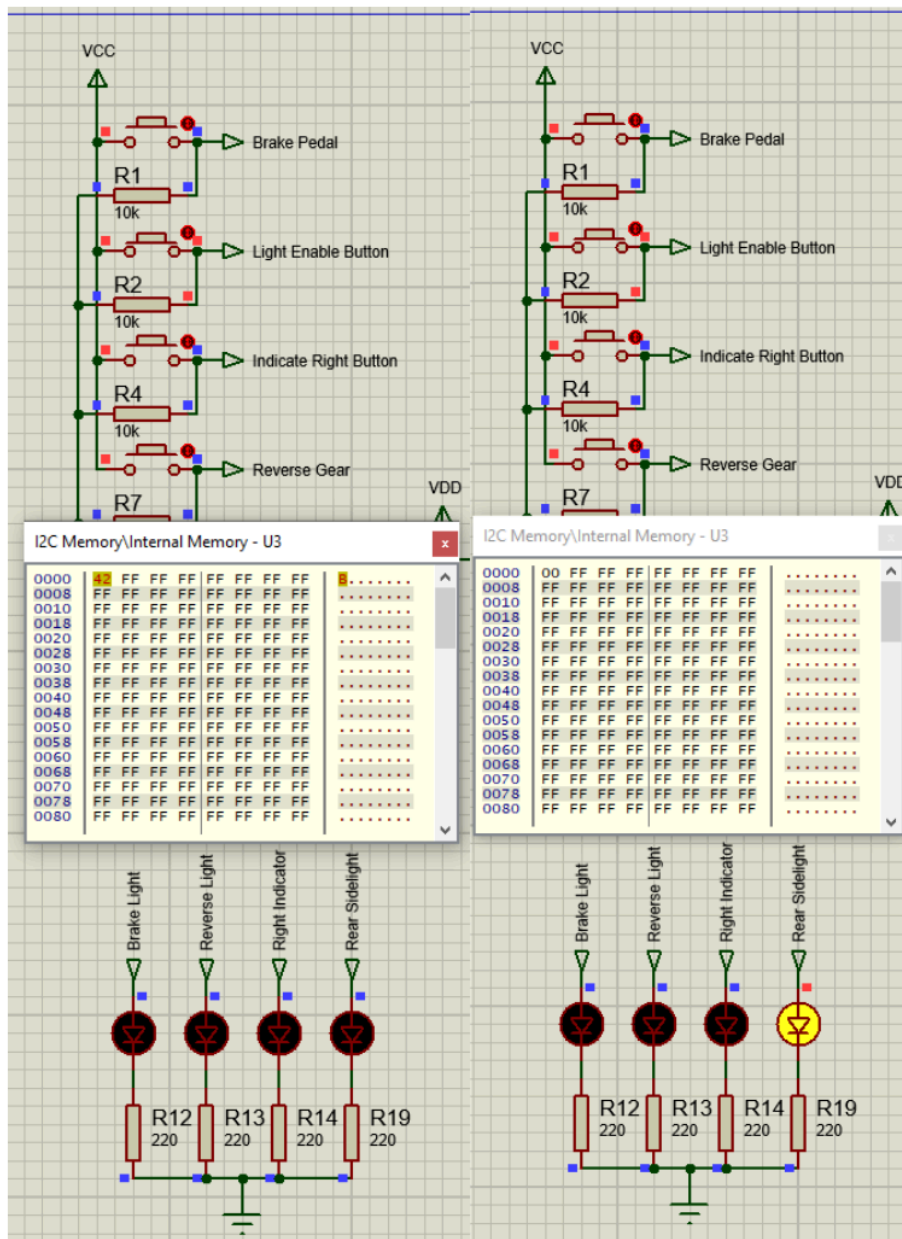


Figure 5

Figure 4

Conclusion

Multi-Master I2C can be an effective way to implement distributed intelligence, and use of simulation software is beneficial for developing software spread across multiple devices which must intercommunicate. The author has successfully used a Multi-Master architecture to implement light controls within a section of a hypothetical automobile, using components rated for use in the automotive industry. Principles for quality described by Quality Management Systems were utilised through careful design considerations and rigorous functionality testing. Since I2C is serving the same purpose as a conventional CAN bus, it's theoretically possible to extend this design and use I2C in place of CAN.

References

- [1] J. KChoi and D. E. Lee, "Higher Dimensional Triangular and Square Numbers," *Communications on Applied Nonlinear Analysis*, vol. 31, no. 6s, pp. 153-159, 2024.
- [2] Cypress, "FM24C04B, 4-Kbit (512 × 8) Serial (I2 C) F-RAM," 12 05 2015. [Online]. Available: https://www.mouser.co.uk/datasheet/2/196/Infineon_FM24C04B_4_Kbit__512_8__Serial__I2C__F_RA-3361561.pdf. [Accessed 15 11 2024].
- [3] Mouser Electronics, "onsemi N24C04UVTG Product Details," 2024. [Online]. Available: <https://www.mouser.co.uk/ProductDetail/onsemi/N24C04UVTG>. [Accessed 15 11 2024].
- [4] Mouser Electronics, "Infineon Technologies FM24C04B-G Product Details," 2024. [Online]. Available: <https://www.mouser.co.uk/ProductDetail/Infineon-Technologies/FM24C04B-G>. [Accessed 15 11 2024].

Distributed Intelligence Systems

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

GRADING FORM: SMU GENERIC_GJ

DANIEL BRIDGEWATER

71.30

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, you did not make use of rs232 to aid with operational testing. Your system requires that all buttons are latching not momentary which is acceptable but potentially a limitation to expansion. 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. 74%

11.25