RTOS Project

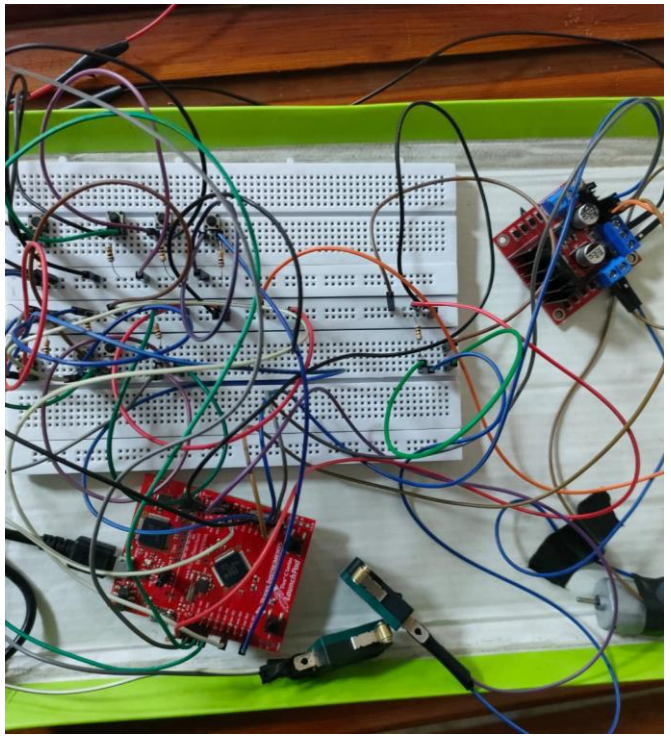
POWER WINDOW CONTROLSYSTEM USING TIVA C RUNNING FREERTOS

Project Scope

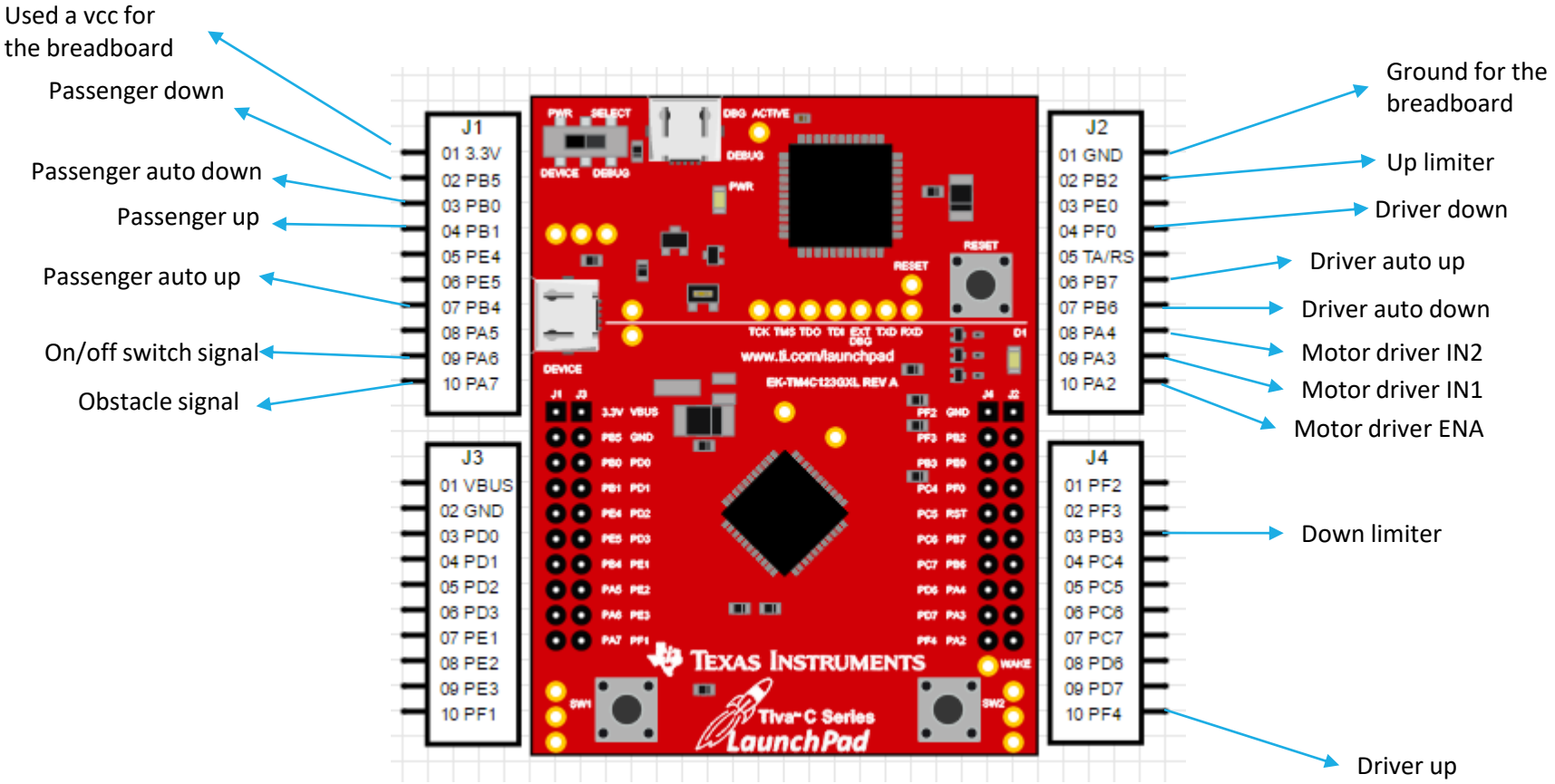
1. Implementation of front passenger door window with both passenger and driver control panels.
2. Implementation of 2 limit switches to limit the window motor from top and bottom limits of the window.
3. Obstacle detection
4. ON/OFF switch to operate locking of the passenger panel from the driver panel.

Hardware Implementation

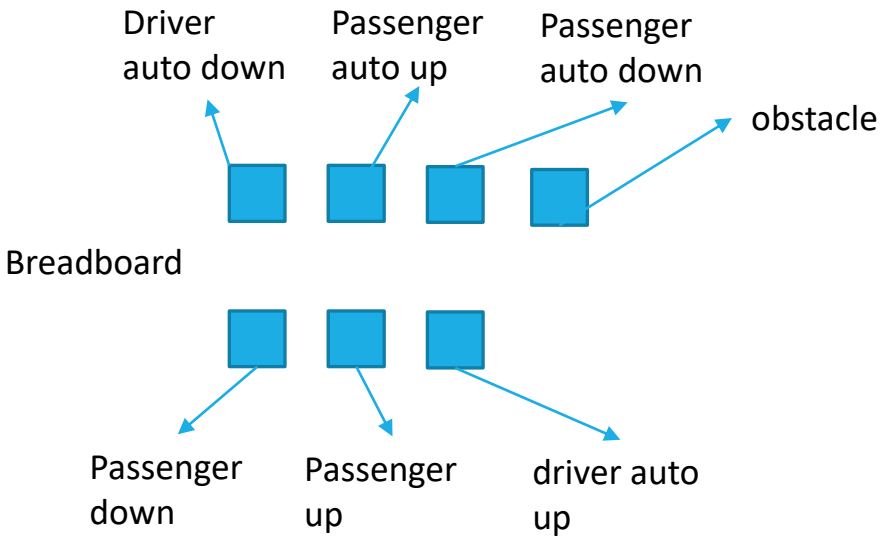
1. A motor driver is used to control the DC motor with a 12v power supply.
2. The motor movement is simulated using push buttons connected to the tiva.
3. The automatic up and down operations are simulated using separate push buttons.
4. 2 limiter switches are used to limit the motor up and down movement.
5. The obstacle signal will be simulated using a push button.
6. On/Off switch is used to lock the passenger control.

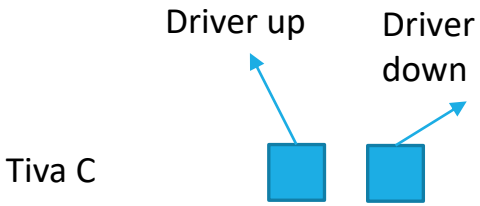


Connections to the Tiva

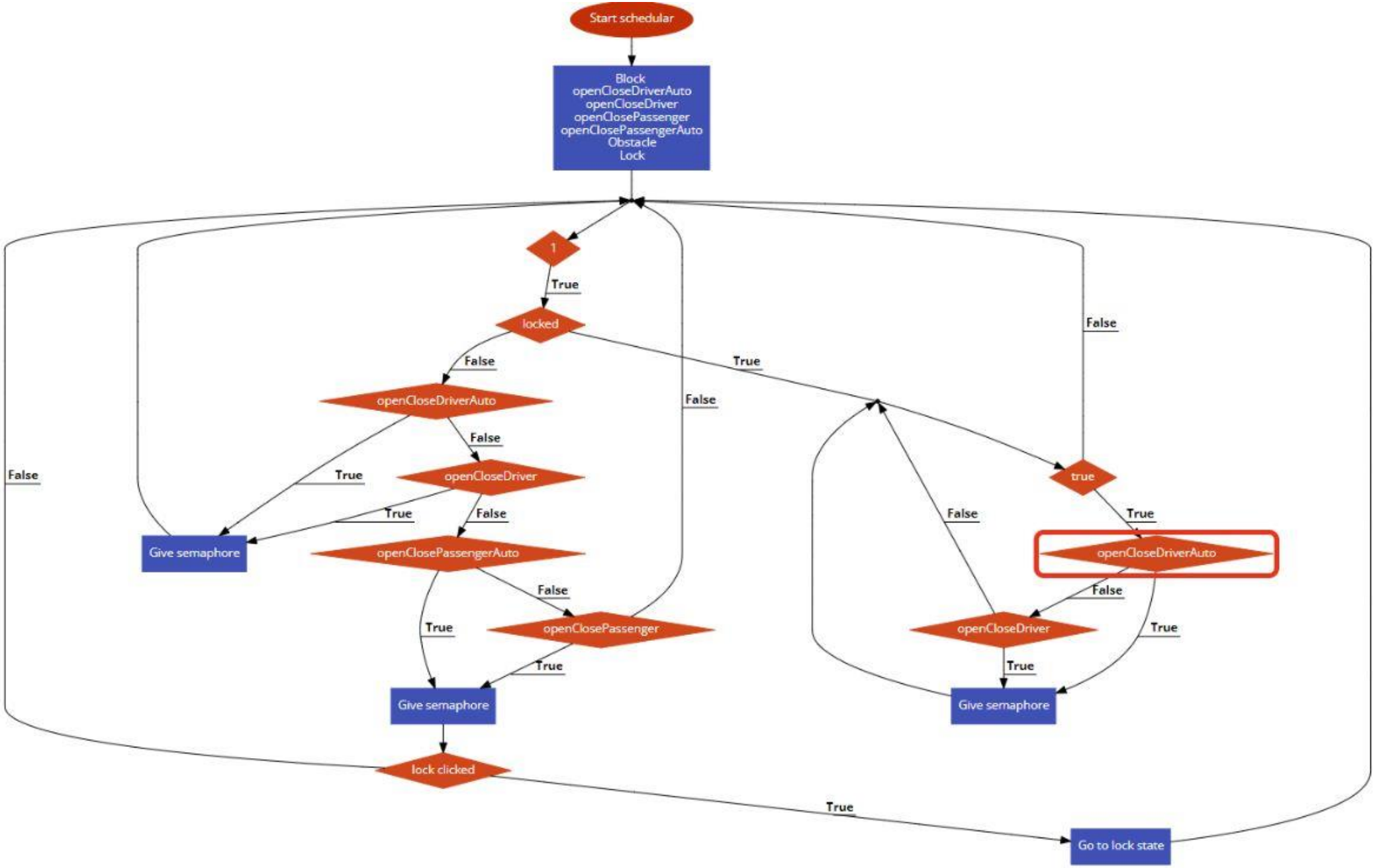


Button Layout





Flow Chart


Start schedular
Block
openCloseDriverAuto
openCloseDriver
openClosePassenger
openClosePassengerAuto
Obstacle
Lock
1
True
locked
False
True
False
openCloseDriverAuto
False
False
True
openCloseDriver
True
False
False
Give semaphore
openClosePassengerAuto
False
True
openClosePassenger
True
Give semaphore
lock clicked
True
Go to lock state
false
True
False
openCloseDriverAuto
False
True
openCloseDriver
True
Give semaphore

Code Abstraction

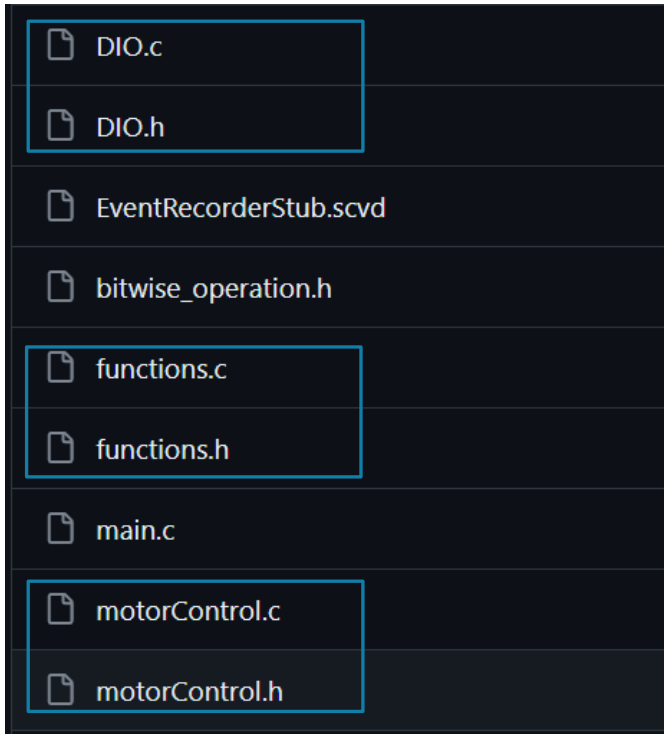
The Software implementation of the operating system is distributed across several files each to handle a single objective.

The DIO file is used to handle the ports initialization and read/write operations on pins and ports

The functions file is used to handle the driver and passenger Open/Close functions as well as the auto Open/Close

The motorControl contains the functions that operates the motor driver

The main file contains the tasks and the semaphores that handles the functions in the functions file



motorControl.c

void motorUP()

This function writes 1 on port A on pins 3 and 2 Which enables ENA and IN1 in the motor driver and IN2 remains 0 so the motor rotates clockwise which simulate the window close.

void motorDown()

Does the same as motorUP() , but a 0 is written on pin 3 and 1 is written on pin 4 which enables the ENA in the motor driver and IN2, so the motor rotates anti-clockwise which simulates the window close.

void motorOFF()

This function forces a zero on all pins that are directed to the motor driver, so the motor is turned off

```
1   #include "FreeRTOS.h"
2   #include "task.h"
3   #include "FreeRTOSConfig.h"
4   #include "DIO.h"
5   #include "bitwise_operation.h"
6   #include "types.h"
7
8   void motorUP(){
9   //DIO_WritePin(&GPIO_PORTA_DATA_R,4,0);
10  //DIO_WritePin(&GPIO_PORTA_DATA_R,3,1);
11
12  DIO_WritePort(&GPIO_PORTA_DATA_R,0xc);
13
14  }
15  void motorDOWN(){
16          //DIO_WritePort(&GPIO_PORTA_DATA_R,0xc);
17          DIO_WritePort(&GPIO_PORTA_DATA_R,0x14);
18  }
19
20  void motorOFF(){
21          DIO_WritePort(&GPIO_PORTA_DATA_R,0x0);
22  }
23
24
```

functions.c

All the queues, task handlers and semaphores handlers defined in main are being being redefined here as extern to be able to use them with the functions defined in functions.c.

```
1   #include "FreeRTOS.h"
2   #include "task.h"
3   #include "FreeRTOSConfig.h"
4   #include "DIO.h"
5   #include "motorControl.h"
6   #include "bitwise_operation.h"
7   #include "types.h"
8   #include <semphr.h>
9   #include "queue.h"
10
11
12  extern xQueueHandle xQueuePD;
13  extern xTaskHandle openCloseDriverHandler;
14  extern xTaskHandle openCloseDriverAutoHandler;
15  extern xTaskHandle openClosePassengerHandler;
16  extern xTaskHandle openClosePassengerAutoHandler;
17  extern xTaskHandle controlHandler;
18  extern xSemaphoreHandle xBinarySemaphore1;
19  extern xSemaphoreHandle xBinarySemaphore2;
20  extern xSemaphoreHandle xBinarySemaphoreDriverAuto;
21  extern xSemaphoreHandle xBinarySemaphorePassengerAuto;
22
```

functions.c

openCloseDriver()

The function runs an infinite loop and waits for an order to open or close a driver. It uses a binary semaphore (xBinarySemaphore1) to synchronize access to the shared resource and a queue (xQueuePD) to receive orders from other tasks.

When an order is received, it checks whether it is an order to open or close the driver. If the order is to open the driver (order == 1), it calls motorUP() function to raise the driver, then enters a while loop that waits until a limit switch is activated to stop the motor. Similarly, if the order is to close the driver (order == 0), it calls motorDOWN() function to lower the driver, then enters a while loop that waits until a limit switch is activated to stop the motor. Finally, it turns off the motor using motorOFF() function and yields the CPU to other tasks using taskYIELD() function.

This function is simulated using the 2 buttons dedicated for the driver's manual close and open window

```
86    void openCloseDriver() {
87            int order;
88            while(1)
89            {
90            xSemaphoreTake(xBinarySemaphore1,portMAX_DELAY);
91
92      xQueueReceive(xQueuePD,&order ,portMAX_DELAY);
93            if(order == 1){
94
95                    motorUP();
96                    while(DIO_ReadPin(&GPIO_PORTF_DATA_R,4) == 0 && DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1);
97            }
98            else if(order == 0){
99
100                   motorDOWN();
101                   while(DIO_ReadPin(&GPIO_PORTF_DATA_R,0) == 0 && DIO_ReadPin(&GPIO_PORTB_DATA_R,3) != 1);
102
103
104
105           }
106           motorOFF();
107           taskYIELD();
108   }
109    }
```

functions.c

openClosePassenger()

This function is used to manipulate the passenger Open/Close. It uses a binary semaphore (xBinarySemaphore2) to synchronize access to the shared resource and a queue (xQueuePD) to receive orders from other tasks.

When an order is received, it checks whether it is an order to move the elevator up (order == 1) or down (order == 0). If the order is to move the elevator up, it calls motorUP() function to raise the elevator and enters a while loop that waits until the elevator reaches the desired floor. The loop waits for the limit switch on the target floor (GPIO_PORTB_DATA_R,4) to be activated while also checking that the limit switch on the current floor (GPIO_PORTB_DATA_R,2) is not activated. Inside the loop, there is a for loop that introduces a delay between iterations. This delay is meant to avoid tight loops that can consume CPU resources.

Similarly, if the order is to move the elevator down, it calls motorDOWN() function to lower the elevator and enters a while loop that waits until the elevator reaches the desired floor. The loop waits for the limit switch on the target floor (GPIO_PORTB_DATA_R,6) to be activated while also checking that the limit switch on the current floor (GPIO_PORTB_DATA_R,3) is not activated. Again, there is a for loop that introduces a delay between iterations.

Finally, it turns off the motor using motorOFF() function and yields the CPU to other tasks using taskYIELD() function.

This function is simulated using the 2 buttons dedicated for the passenger's manual close and open window

```
111     void openClosePassenger() {
112             int order;
113             while(1)
114             {
115             xSemaphoreTake(xBinarySemaphore2,portMAX_DELAY);
116              xQueueReceive(xQueuePD,&order ,portMAX_DELAY);
117             if(order == 1){
118
119
120                     motorUP();
121                     while(DIO_ReadPin(&GPIO_PORTB_DATA_R,4) == 1 && DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1){
122                     for(int i=0;i<100;i++){}
123                     }
124             }
125             else if(order == 0){
126
127                     motorDOWN();
128                     while(DIO_ReadPin(&GPIO_PORTB_DATA_R,6) == 1 && DIO_ReadPin(&GPIO_PORTB_DATA_R,3) != 1){
129                     for(int i=0;i<100;i++){}
130                     }
131             }
132
133             motorOFF();
134             taskYIELD();
135     }
136
137     }
```

functions.c

openCloseDriverAuto()

This function is used to manipulate the passenger Open/Close. It uses a binary semaphore (xBinarySemaphore2) to synchronize access to the shared resource and a queue (xQueuePD) to receive orders from other tasks.

When an order is received, it checks whether it is an order to move the elevator up (order == 1) or down (order == 0). If the order is to move the elevator up, it calls motorUP() function to raise the elevator and enters a while loop that waits until the elevator reaches the desired floor. The loop waits for the limit switch on the target floor (GPIO_PORTB_DATA_R,4) to be activated while also checking that the limit switch on the current floor (GPIO_PORTB_DATA_R,2) is not activated. Inside the loop, there is a for loop that introduces a delay between iterations. This delay is meant to avoid tight loops that can consume CPU resources.

Similarly, if the order is to move the elevator down, it calls motorDOWN() function to lower the elevator and enters a while loop that waits until the elevator reaches the desired floor. The loop waits for the limit switch on the target floor (GPIO_PORTB_DATA_R,6) to be activated while also checking that the limit switch on the current floor (GPIO_PORTB_DATA_R,3) is not activated. Again, there is a for loop that introduces a delay between iterations.

Finally, it turns off the motor using motorOFF() function and yields the CPU to other tasks using taskYIELD() function.

This function is simulated using the 2 buttons dedicated for the driver's auto close and open window

```
54  void openCloseDriverAuto() {
55
56          int order;
57          while(1)
58          {
59          xSemaphoreTake(xBinarySemaphoreDriverAuto,portMAX_DELAY);
60
61    xQueueReceive(xQueuePD,&order ,portMAX_DELAY);
62          if(order == 1){
63                  motorUP();
64                  //while(DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1);
65                  while(((*(&GPIO_PORTB_DATA_R) & (1<<2))>>2) != 1){
66                  for(int i=0;i < 1000;i++){}
67                  }
68                  int x = 0;
69          }
70          else if(order == 0){
71
72                  motorDOWN();
73                  while(((*(&GPIO_PORTB_DATA_R) & (1<<3))>>3) != 1){
74                  for(int i=0;i < 1000;i++){}
75                  }
76                  int x = 0;
77
78          }
79          motorOFF();
80          taskYIELD();
81  }
```

functions.c

openClosePassengerAuto()

This function operates similar to openCloseDriverAuto(), but instead it is manipulated using the auto buttons of the passenger

It continuously waits for a binary semaphore xBinarySemaphorePassengerAuto and a queue xQueuePD to receive orders. If the received order is 1, it calls the motorUP() function to move a motor upwards, and waits for a limit switch connected to pin 2 of GPIO port B to be activated, by repeatedly checking the value of the pin until it reads as 1. Then, it turns off the motor by calling motorOFF().

If the received order is 0, it calls the motorDOWN() function to move the motor downwards, and waits for a limit switch connected to pin 3 of GPIO port B to be activated, by repeatedly checking the value of the pin until it reads as 1. Then, it turns off the motor by calling motorOFF().

Like openCloseDriverAuto(), the function includes a call to taskYIELD() at the end of each iteration of the loop, allowing other tasks in the system to execute.This function is simulated using the 2 buttons dedicated for the passenger's manual close and open window

```
24  void openClosePassengerAuto(){
25
26          int order;
27          while(1)
28          {
29          xSemaphoreTake(xBinarySemaphorePassengerAuto,portMAX_DELAY);
30
31    xQueueReceive(xQueuePD,&order ,portMAX_DELAY);
32          if(order == 1){
33                  motorUP();
34                  //while(DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1);
35                  while(((*(&GPIO_PORTB_DATA_R) & (1<<2))>>2) != 1){
36                  for(int i=0;i < 1000;i++){}
37                  }
38                  int x = 0;
39          }
40          else if(order == 0){
41
42                  motorDOWN();
43                  while(((*(&GPIO_PORTB_DATA_R) & (1<<3))>>3) != 1){
44                  for(int i=0;i < 1000;i++){}
45                  }
46                  int x = 0;
47
48          }
49          motorOFF();
50          taskYIELD();
51          }
52  }
```

functions.c

openClosePassengerAuto()

This function operates similar to openCloseDriverAuto(), but instead it is manipulated using the auto buttons of the passenger

It continuously waits for a binary semaphore xBinarySemaphorePassengerAuto and a queue xQueuePD to receive orders. If the received order is 1, it calls the motorUP() function to move a motor upwards, and waits for a limit switch connected to pin 2 of GPIO port B to be activated, by repeatedly checking the value of the pin until it reads as 1. Then, it turns off the motor by calling motorOFF().

If the received order is 0, it calls the motorDOWN() function to move the motor downwards, and waits for a limit switch connected to pin 3 of GPIO port B to be activated, by repeatedly checking the value of the pin until it reads as 1. Then, it turns off the motor by calling motorOFF().

Like openCloseDriverAuto(), the function includes a call to taskYIELD() at the end of each iteration of the loop, allowing other tasks in the system to execute.This function is simulated using the 2 buttons dedicated for the passenger's manual close and open window

```
24  void openClosePassengerAuto(){
25
26          int order;
27          while(1)
28          {
29          xSemaphoreTake(xBinarySemaphorePassengerAuto,portMAX_DELAY);
30
31    xQueueReceive(xQueuePD,&order ,portMAX_DELAY);
32          if(order == 1){
33                  motorUP();
34                  //while(DIO_ReadPin(&GPIO_PORTB_DATA_R,2) != 1);
35                  while(((*(&GPIO_PORTB_DATA_R) & (1<<2))>>2) != 1){
36                  for(int i=0;i < 1000;i++){}
37                  }
38                  int x = 0;
39          }
40          else if(order == 0){
41
42                  motorDOWN();
43                  while(((*(&GPIO_PORTB_DATA_R) & (1<<3))>>3) != 1){
44                  for(int i=0;i < 1000;i++){}
45                  }
46                  int x = 0;
47
48          }
49          motorOFF();
50          taskYIELD();
51          }
52  }
```

functions.c

obstacle()

This function seems to be handling an obstacle that interrupts the movement of the window. It uses a binary semaphore xBinarySemaphoreObstacle to wait for an obstacle signal. Once the signal is received, the motor is turned off and the window is moved down. After a delay, the motor is turned off again and the task yields to let other tasks run.

```
27  void Obstacle(){
28
29          int order;
30          while(1)
31          {
32          xSemaphoreTake(xBinarySemaphoreObstacle,portMAX_DELAY);
33          motorOFF();
34          motorDOWN();
35          for(int k=0; k<4000000; k++);
36          motorOFF();
37          for(int k=0; k<2000000; k++);
38          vTaskPrioritySet(NULL,2);
39          taskYIELD();
40          }
41
42
43  }
```

functions.c

Void control()

The function continuously checks the state of various input pins and sends corresponding messages to a queue xQueuePD and semaphores xBinarySemaphore1, xBinarySemaphore2, xBinarySemaphoreDriverAuto, and xBinarySemaphorePassengerAuto.

The function first creates a queue xQueuePD with a maximum length of 1 and a size of long. Then, it enters an infinite loop and checks the state of various pins using DIO_ReadPin() function from GPIO ports B and F.

If pin 4 of GPIO port F is read as 0, the function sends the value 1 to the queue xQueuePD and gives the semaphore xBinarySemaphore1.

If pin 0 of GPIO port F is read as 0, the function sends the value 0 to the queue xQueuePD and gives the semaphore xBinarySemaphore1.

If pin 0 of GPIO port B is read as 1, the function sends the value 1 to the queue xQueuePD and gives the semaphore xBinarySemaphoreDriverAuto.

If pin 5 of GPIO port B is read as 1, the function sends the value 0 to the queue xQueuePD and gives the semaphore xBinarySemaphoreDriverAuto.

If pin 4 of GPIO port B is read as 1, the function sends the value 1 to the queue xQueuePD and gives the semaphore xBinarySemaphore2.

If pin 6 of GPIO port B is read as 1, the function sends the value 0 to the queue xQueuePD and gives the semaphore xBinarySemaphore2.

If pin 1 of GPIO port B is read as 1, the function sends the value 1 to the queue xQueuePD and gives the semaphore xBinarySemaphorePassengerAuto.

If pin 7 of GPIO port B is read as 1, the function sends the value 0 to the queue xQueuePD and gives the semaphore xBinarySemaphorePassengerAuto.

The function does not have any explicit delay or yield statements, which could cause it to consume significant CPU time.

```
146         while(1){
147                 if(DIO_ReadPin(&GPIO_PORTF_DATA_R,4) == 0){
148                         xQueueSend(xQueuePD, &up, 0);
149                         xSemaphoreGive(xBinarySemaphore1);
150
151                 }
152                 else if(DIO_ReadPin(&GPIO_PORTF_DATA_R,0) == 0){
153                         xQueueSend(xQueuePD, &down, 0);
154                         xSemaphoreGive(xBinarySemaphore1);
155
156                 }
157                 else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,0) == 1){// for Driver auto up
158                         xQueueSend(xQueuePD, &up, 0);
159                         xSemaphoreGive(xBinarySemaphoreDriverAuto);
160
161                 }else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,5) == 1){// for Driver auto down
162                         xQueueSend(xQueuePD, &down, 0);
163                         xSemaphoreGive(xBinarySemaphoreDriverAuto);
164
165                 }
166
167                 else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,4) == 1){// for passenger down
168                 //high pri
169                         xQueueSend(xQueuePD, &up, 0);
170                         xSemaphoreGive(xBinarySemaphore2);
171
172                 }
173                 else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,6) == 1){// for passener up
174                         xQueueSend(xQueuePD, &down, 0);
175                         xSemaphoreGive(xBinarySemaphore2);
176
177                 }
178
179                 else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,1) == 1){// for passener up auto
180                         xQueueSend(xQueuePD, &up, 0);
181                         xSemaphoreGive(xBinarySemaphorePassengerAuto);
182
183                 }
184                 else if(DIO_ReadPin(&GPIO_PORTB_DATA_R,7) == 1){// for passener up auto
185                         xQueueSend(xQueuePD, &down, 0);
186                         xSemaphoreGive(xBinarySemaphorePassengerAuto);
187
188                 }
189
190
191         }
192
193 }
```

main.c

First, the queue, task handlers and semaphore handlers are declared.

And also a declaration for vApplicationIdleHook() is included

```
11    xQueueHandle xQueuePD;
12    xTaskHandle openCloseDriverHandler;
13    xTaskHandle openCloseDriverAutoHandler;
14    xTaskHandle openClosePassengerHandler;
15    xTaskHandle openClosePassengerAutoHandler;
16    xTaskHandle controlHandler;
17    xSemaphoreHandle xBinarySemaphore1;
18    xSemaphoreHandle xBinarySemaphore2;
19    xSemaphoreHandle xBinarySemaphoreDriverAuto;
20    xSemaphoreHandle xBinarySemaphorePassengerAuto;
21    void vApplicationIdleHook(){
22    }
23
```

main.c

This code creates binary semaphores and tasks for controlling the car windows. There are four tasks for opening and closing the driver and passenger windows, two of which are automatic tasks. There is also a control task that manages the overall system.

The DIO_init() function initializes the digital input/output pins of the microcontroller, and the xBinarySemaphore variables are used to synchronize the tasks. Each task controls the opening and closing of a specific window using the openCloseDriver() and openClosePassenger() functions, which are triggered by the control task. The automatic tasks use sensors to detect obstacles and stop the window movement if necessary.ض

The code enters an infinite loop after starting the FreeRTOS scheduler, indicating that the microcontroller will continue running the tasks until it is reset or turned off.

```
25  int main(){
26          DIO_init();
27          //while(((*(&GPIO_PORTB_DATA_R) & (1<<3))>>3) != 1);
28          //while(1);
29          //while(1);
30          xBinarySemaphore1 = xSemaphoreCreateBinary();
31          xBinarySemaphore2 = xSemaphoreCreateBinary();
32          xBinarySemaphoreDriverAuto = xSemaphoreCreateBinary();
33          xBinarySemaphorePassengerAuto= xSemaphoreCreateBinary();
34          xTaskCreate(openCloseDriver,"openCloseDriver",240,NULL,2,&openCloseDriverHandler);
35          xTaskCreate(openCloseDriverAuto,"openCloseDriverAuto",240,NULL,2,&openCloseDriverAutoHandler);
36          xTaskCreate(openClosePassenger,"openClosePassenger",240,NULL,2,&openClosePassengerHandler);
37
38          xTaskCreate(openClosePassengerAuto,"openClosePassengerAuto",240,NULL,2,&openClosePassengerAutoHandler);
39          xTaskCreate(control,"control",240,NULL,1,&controlHandler);
40          //xTaskCreate(&fun2,"fun2",240,NULL,2,NULL);
41          vTaskStartScheduler();
42          for(;;);
43
44  return 0;
45  }
```