CSE411

Real-Time Embedded Systems Design

Project Report

Team	Name	ID
1	Matthew Sherif Shalaby	20P6785
	Karim Bassel Samir	20P6794
	Fady Fady Fouad	20P7341
	Mina Fadi Mohsen	20P6996
	Shady Emad Sabry	20P7239
	Salma Sherif Mohamed	20P5762

Table of Contents

Video & Code Link **3**

Introduction: **3**

FreeRtos: **4**

Project Requirement: **5**

- **Power Window** 5

Components: **6**

Finite State Machine: **7**

- **Power Window Control System Finite State Machine** 9

Diagrams: **10**

- **Context Diagram** 10
- **Activity Diagram** 11
- **Block Diagram** 12

Pin Configuration **13**

Pin Initialization **13**

Functions Used **14**

Tasks Used **15**

- Task Safety (Priority = 4) 15
- Task Receive (Priority = 3) 15
- Task Driver (Priority = 1) 16
- Task Passenger (Priority = 1) 18

Queues Implementation **20**

- Task Driver 20
- Task Passenger 20
- Task Receive 21

Semaphore Implementation **22**

- Task Safety 22
- ISR Giving the Semaphore to Task Safety 22

Mutex Implementation(Queue Shared Resource) **23**

- Task Driver 23
- Task Passenger 24

Problems Faces 25

System Flowchart 26

Conclusion: 27

Table of Figures

Figure 1 example..6
Figure 2 Limit switch..6
Figure 3 DC motor..6
Figure 4 Push buttons..7
Figure 5 Tiva-C board..7
Figure 6 Simulink State Machine..8
Figure 7 Context Diagram..10
Figure 8 Activity Diagram..11
Figure 9 Block Diagram..12
Figure 10: System Flowchart..26

Video & Code Link

https://drive.google.com/drive/folders/1ZICCM4KZqfKyTxxZ0_v_i7u1cCeKr1OW?usp=sharing

Introduction:

In today's automotive landscape, the integration of advanced electronic control systems has become pivotal in enhancing both comfort and safety for vehicle occupants. One such system, the power window control system, serves as a quintessential example of this technological advancement. Our project endeavors to delve into this domain by implementing a sophisticated power window control system utilizing the Tiva C microcontroller platform and FreeRTOS for task management.

The objective of this project is multifaceted, aiming to develop a robust and efficient control system for the front passenger door window. This includes the implementation of both passenger and driver control panels to facilitate seamless operation. Embracing the principles of modern embedded systems design, our solution prioritizes responsiveness, reliability, and safety.

Central to our endeavor is the utilization of the Tiva C microcontroller, renowned for its computational prowess and versatile interfacing capabilities. Leveraging FreeRTOS as our real-time operating system ensures efficient task scheduling and execution, critical for meeting stringent automotive performance standards.

Beyond mere functionality, our project emphasizes safety as a paramount concern. Integration of limit switches enables precise control of the window's range of motion, preventing potential damage and ensuring user safety. Furthermore, obstacle detection mechanisms are employed to identify and mitigate potential hazards, thereby enhancing the overall safety profile of the system.

FreeRtos:

FreeRTOS is a real-time operating system kernel for embedded devices that has been ported to 35 microcontroller platforms. It is distributed under the MIT License.

The FreeRTOS kernel was originally developed by Richard Barry around 2003, and was later developed and maintained by Barry's company, Real Time Engineers Ltd. In 2017, the firm passed stewardship of the FreeRTOS project to Amazon Web Services (AWS). Barry continues to work on FreeRTOS as part of an AWS team.

FreeRTOS is designed to be small and simple. It is mostly written in the C programming language to make it easy to port and maintain. It also comprises a few assembly language functions where needed, mostly in architecture-specific scheduler routines.

- **Process management**

  FreeRTOS provides methods for multiple threads or tasks, mutexes, semaphores and software timers. A tickless mode is provided for low power applications. Thread priorities are supported. FreeRTOS applications can be statically allocated, but objects can also be dynamically allocated with five schemes of memory management (allocation):

Allocate only:

- allocate and free with a very simple, fast, algorithm;
- a more complex but fast allocate and free algorithm with memory coalescence;
- an alternative to the more complex scheme that includes memory coalescence that allows a heap to be broken across multiple memory areas.
- and C library allocate and free with some mutual exclusion protection.

RTOSes typically do not have the more advanced features that are found in operating systems like Linux and Microsoft Windows, such as device drivers, advanced memory management, and user accounts. The emphasis is on compactness and speed of execution. FreeRTOS can be thought of as a thread library rather than an operating system, although command line interface and POSIX-like input/output (I/O) abstraction are available.

FreeRTOS implements multiple threads by having the host program call a thread tick method at regular short intervals. The thread tick method switches tasks depending on priority and a round-robin scheduling scheme. The usual interval is 1 to 10 milliseconds (1⁄1000 to 1⁄100 of a second) via an interrupt from a hardware timer, but this interval is often changed to suit a given application.

The software distribution contains prepared configurations and demonstrations for every port and compiler, allowing rapid application design. The project website provides documentation and RTOS tutorials, and details of the RTOS design.

Project Requirement:

Power Window

Automobiles use electronics for control operations such as:

- Opening and closing windows and sunroof
- Adjusting mirrors and headlights
- Locking and unlocking doors

These systems are subject to stringent operation constraints. Failures can cause dangerous and possibly life-threatening situations. As a result, careful design and analysis are needed before deployment.

This example focuses on the design of a power window system of an automobile, particularly the passenger-side window. A critical aspect of this system is that it cannot exert a force of more than 100 N on an object when the window closes. When the system detects such an object, it must lower the window by about 10 cm.

As part of the design process, the example considers:

- Quantitative requirements for the window control system, such as timing and force requirements
- System requirements, captured in activity diagrams
- Data definitions for the signals used in activity diagrams

Other aspects of the design process that this example contains are:

- Managing the components of the system
- Building the model
- Validating the results of system simulation
- Generating code

Design Requirements

In this example, consider the passenger-side power window system of an automobile. This system can never exert a force of more than 100 N on an object when closing the window.

When the model detects such an object, the model must lower the window by about 10 centimeters.

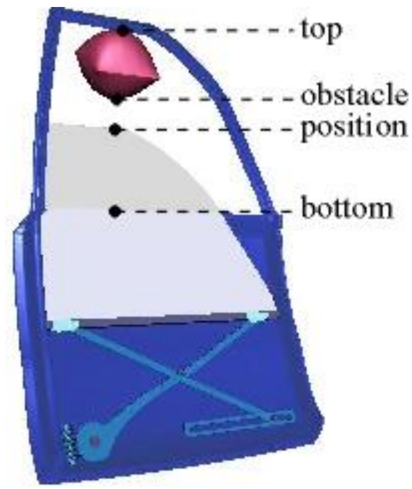

Figure 1 example

Components:

Figure 2 Limit switch

Figure 3 DC motor

Figure 4 Push buttons

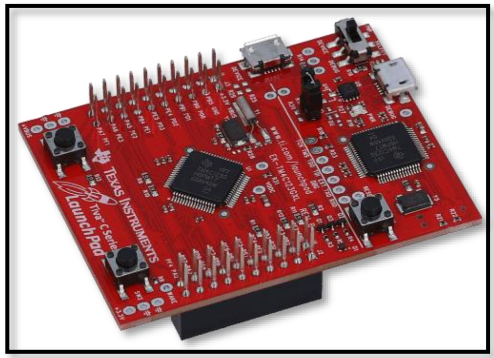

Figure 5 Tiva-C board

Finite State Machine:

This state machine contains the basic states of the power window system: up, auto-up, down, auto-down, rest, and emergency. It models the state transitions and accounts for the precedence of driver commands over the passenger commands. It also includes emergency behavior that activates when the software detects an object between the window and the frame while moving up.

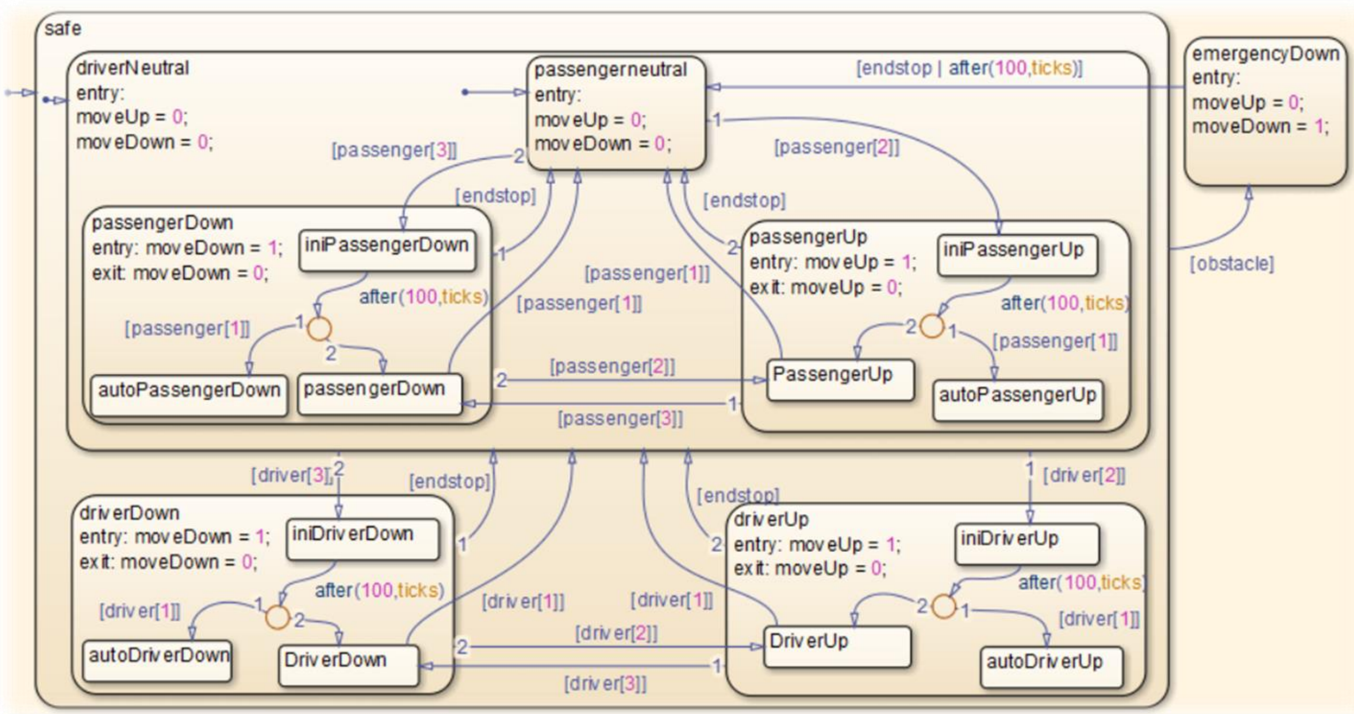

Figure 6 Simulink State Machine

State changes because of passenger commands are encapsulated in a super state that does not correspond to an active driver command.

Consider the control of the passenger window. The passenger or driver can move this window up and down.

Power Window Control System Finite State Machine

1. **Up State:**
- Description: The window is moving upward.
- Transitions:
- Transition to Rest State: When the window reaches the fully closed position.
- Transition to Auto-Up State: When the auto-up button is pressed.
- Transition to Down State: When the down button is pressed.

2. Auto-Up State:

- Description: The window is automatically moving upward.
- Transitions:
- Transition to Rest State: When the window reaches the fully closed position.
- Transition to Up State: When the auto-up process is interrupted or completed.
- Transition to Down State: When the down button is pressed or an obstacle is detected.

3. Down State:

- Description: The window is moving downward.
- Transitions:
- Transition to Rest State: When the window reaches the fully open position.
- Transition to Auto-Down State: When the auto-down button is pressed.
- Transition to Up State: When the up button is pressed.

4. Auto-Down State:

- Description: The window is automatically moving downward.
- Transitions:
- Transition to Rest State: When the window reaches the fully open position.
- Transition to Down State: When the auto-down process is interrupted or completed.
- Transition to Up State: When the up button is pressed or an obstacle is detected.

5. Rest State:

- Description: The window is fully closed or fully open, and no movement is occurring.
- Transitions:
- Transition to Up State: When the up button is pressed.
- Transition to Down State: When the down button is pressed.
- Transition to Auto-Up State: When the auto-up button is pressed.
- Transition to Auto-Down State: When the auto-down button is pressed.
- Transition to Emergency State: When an emergency is detected.

6. Emergency State:

- Description: A critical situation is detected, requiring immediate action.
- Transitions:
- Transition to Rest State: When the emergency is resolved.

Diagrams:

Context Diagram

The figure represents the context diagram of a power window system. The square boxes capture the environment, in this case, the driver, passenger, and window. Both the driver and passenger can send commands to the window to move it up and down. The controller infers the correct command to send to the window actuator (e.g., the driver command has priority over the passenger command).

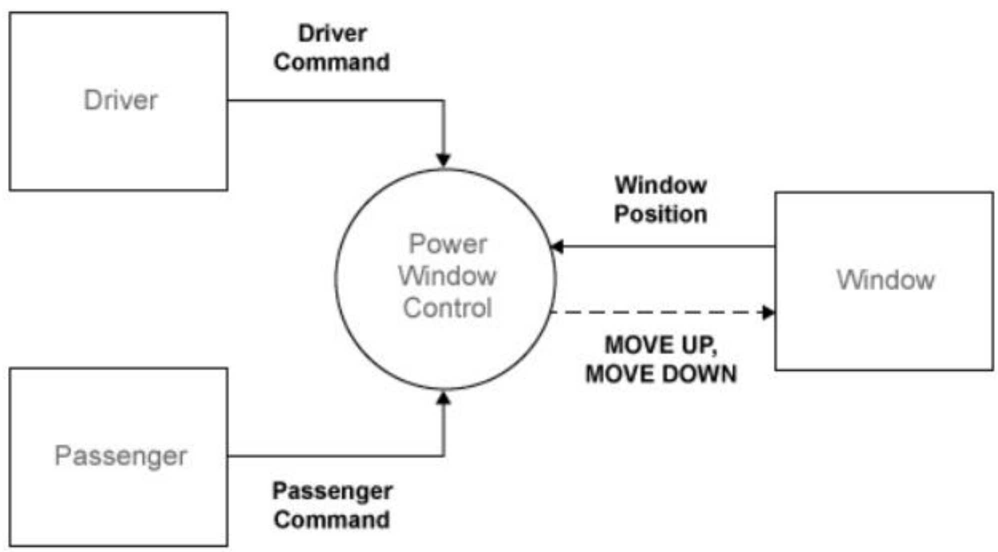

Figure 7 Context Diagram

In addition, diagram monitors the state of the window system to establish when the window is fully opened and closed and to detect if there is an object between the window and frame.

The circle (also known as a bubble) represents the power window controller. The circle is the graphical notation for a process. Processes capture the transformation of input data into output data. Primitive process might also generate.

Activity Diagram

The power window control consists of three processes. Two processes validate the driver and passenger input to ensure that their input is meaningful given the state of the system. For example, if the window is completely opened, the MOVE DOWN command does not make sense. The remaining process detects if the window is completely opened or completely closed and if an object is present.

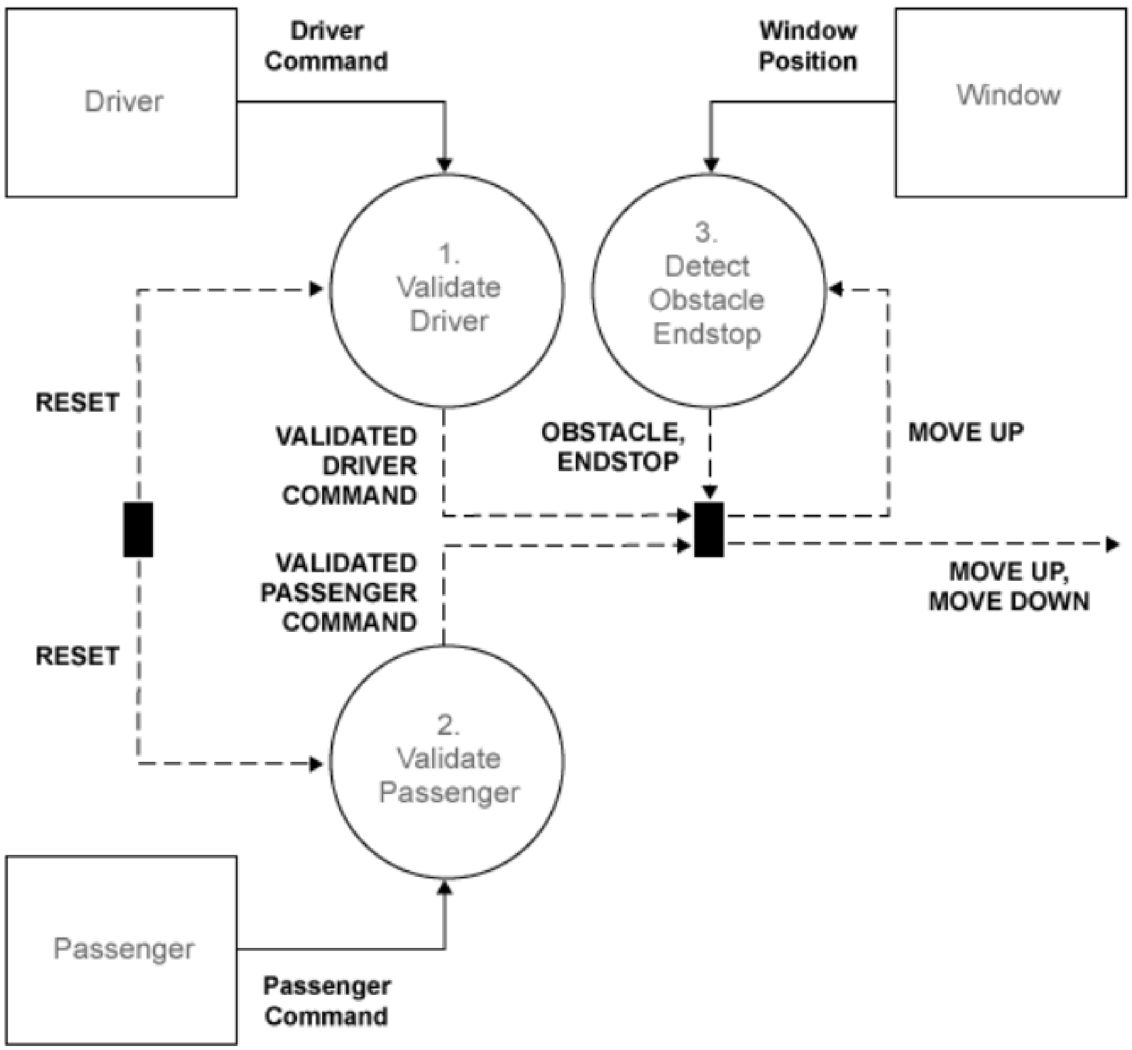

Figure 8 Activity Diagram

Block Diagram

This diagram illustrates the main components of the power window control system project, provides a high-level overview of the system architecture and the interactions between its various components.

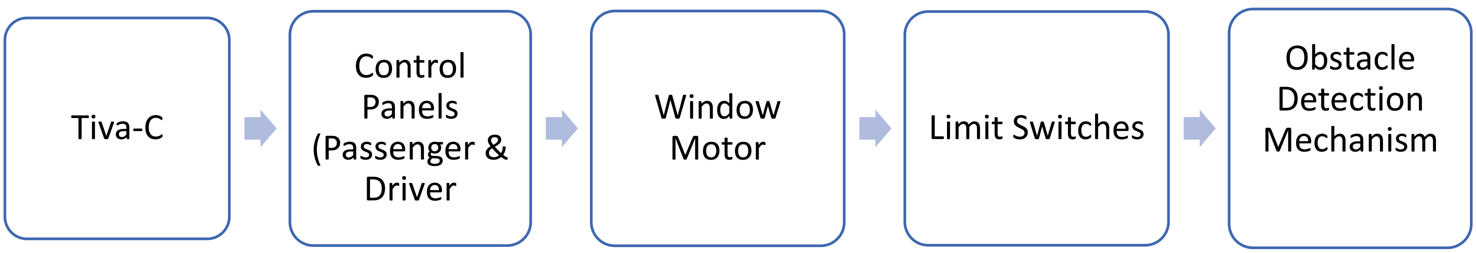

Figure 9 Block Diagram

1. **Tiva C Microcontroller:**
   - Serves as the central processing unit, responsible for interfacing with various components and executing control algorithms.
2. **Control Panels (Passenger & Driver):**
   - Interface with users to receive input commands for controlling the window operation.
3. **Window Motor:**
   - Actuates the movement of the window glass based on commands received from the microcontroller.
4. **Limit Switches:**
   - Provide feedback to the microcontroller about the position of the window glass, enabling precise control and preventing damage by limiting movement within safe bounds.
5. **Obstacle Detection Mechanism:**
   - Detects obstacles or obstructions during window operation, triggering appropriate responses to ensure user safety.

These components work together under the supervision of the Tiva C microcontroller to facilitate the safe and efficient operation of the power window control system.

Pin Configuration

Functionality	Pins
Motor Pins	PC4, PC5
Driver & Passenger Push Buttons	PD0, PD1, PD2, PD3
Jam Protection Pin	PF0 || PF4 (Tiva Push Buttons)
Lock Switch	PA7
Limit Switch Up	PA2
Limit Switch Down	PA3

Pin Initialization

```
void Motor_Init(){
 SYSCTL->RCGCGPIO |= 0x00000004;
 GPIO_PORTC_LOCK_R = 0x4C4F434B;
 GPIO_PORTC_CR_R = 0x30;
 GPIO_PORTC_DIR_R = 0x30;
  GPIO_PORTC_DEN_R = 0x30;
}

void Switches_Init(){
   SYSCTL->RCGCGPIO |= 0x00000008;
   GPIOD->LOCK = 0x4C4F434B;
   GPIOD->CR = 0xF;
   GPIOD->DIR = 0x00;
   GPIOD->PUR = 0xF;
   GPIOD->DEN = 0xF;
}
void Switch_Init(){
 SYSCTL->RCGCGPIO |= 0x00000001;
 GPIO_PORTA_LOCK_R = 0x4C4F434B;
 GPIO_PORTA_CR_R = 0x8C;
 GPIO_PORTA_DIR_R = 0x00;
 GPIO_PORTA_DEN_R = 0x8C;
 GPIO_PORTA_PUR_R = 0x8C;
}
```

```c
void IR_Init(){
    SYSCTL->RCGCGPIO |= 0x00000020;
   GPIOF->LOCK = 0x4C4F434B; // 2) unlock PortF PF0
   GPIOF->CR = 0x1F; // allow changes to PF4-0
   GPIOF->AMSEL= 0x00; // 3) disable analog function
   GPIOF->PCTL = 0x00000000; // 4) GPIO clear bit PCTL
   GPIOF->DIR = 0x0E; // 5) PF4,PF0 input, PF3,PF2,PF1 output
    GPIOF->AFSEL = 0x00; // 6) no alternate function
   GPIOF->PUR = 0x11; // enable pullup resistors on PF4,PF0
   GPIOF->DEN = 0x1F; // 7) enable digital pins PF4-PF0
  GPIOF->DATA = 0x00;
  // Setup the interrupt on PortF
  GPIOF->ICR = 0x11; // Clear any Previous Interrupt
  GPIOF->IM |=0x11; // Unmask the interrupts for PF0 and PF4
  GPIOF->IS |= 0x11; // Make bits PF0 and PF4 level sensitive
  GPIOF->IEV &= ~0x11; // Sense on Low Level
  NVIC_PRI7_R |= (5<<21);
  NVIC_EnableIRQ(PortF_IRQn); // Enable the Interrupt for PortF in NVIC
}
```

Functions Used

```c
void MotorUp(){
   GPIO_PORTC_DATA_R=0x20;
}
void MotorDown(){
   GPIO_PORTC_DATA_R=0x10;
}
void MotorStop(){
   GPIO_PORTC_DATA_R=0x00;
}
```

Tasks Used

Task Safety (Priority = 4)

```
void Task_Safety(){
  xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
        MotorDown();
        for(int i=0;i<1000000;i++);
        MotorStop();
        xSemaphoreGive( xBinarySemaphore );
    }
}
```

Task Receive (Priority = 3)

```
static void Task_Rec(){
  int rec;
  portBASE_TYPE xStatus;
  for( ;; )
    {
    if( uxQueueMessagesWaiting( xQueue ) != 1 )
    {
      //vPrintString( "Queue should have been full!\n" );
    }
    xStatus = xQueueReceive( xQueue, &rec, portMAX_DELAY )

    if( xStatus == pdPASS )
    {
      if(rec == 2){
        MotorUp();
      }
      else if(rec ==1){
        MotorStop();
      }
      else if(rec==4){
        MotorDown();
      }
    }
    else
    {
    }
  }
}
```

Task Driver (Priority = 1)

```
void Task_Driver(){
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
int down=4;
int up=2;// up
int stop=1;//auto stop

  for(;;){
    xSemaphoreTake( xMutex, portMAX_DELAY );
    if((GPIO_PORTD_DATA_R & 0x4)==0x0){//driver up
      //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)+1);
      while((GPIO_PORTD_DATA_R & 0x4)==0x0){
        xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
        counter++;
        if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
      }
      if(counter < 200000){
        while((GPIO_PORTA_DATA_R &0x4) != 0){
          xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
          if( xStatus != pdPASS )
            {
            /* We could not write to the queue because it was full - this must
            be an error as the receiving task should make space in the queue
            as soon as both sending tasks are in the Blocked state. */
            //vPrintString( "Could not send to the queue.\n" );
```

```c
        counter=0;
        xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }

        //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)-1);
      }
      /************************************************************************************
      if((GPIO_PORTD_DATA_R & 0x1)== 0x0){//driver down
        while((GPIO_PORTD_DATA_R & 0x1)==0x0){
          xStatus = xQueueSendToBack( xQueue, &down, xTicksToWait );
          counter++;
          if( xStatus != pdPASS )
            {
            }
        }
        if(counter < 200000){
          while((GPIO_PORTA_DATA_R &0x8) != 0){
            xStatus = xQueueSendToBack( xQueue, &down, xTicksToWait );
            if( xStatus != pdPASS )
              {
              }
          }
        }
        counter=0;
```

```c
          counter=0;
          xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
          if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
        }
        xSemaphoreGive( xMutex );
    }
}
```

Task Passenger (Priority = 1)

```
void Task_Passenger(){
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
int down=4;
int up=2;// up
int stop=1;//auto stop
int counter=0;
  for(;;){
    xSemaphoreTake( xMutex, portMAX_DELAY );

    if(((GPIO_PORTD_DATA_R & 0x2)==0x0) && ((GPIO_PORTA_DATA_R & 0x80)==0x0)){//passenger up
      //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)+1);
      while((GPIO_PORTD_DATA_R & 0x2)==0x0){
        xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
        counter++;
        if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
      }
      if(counter < 200000){
        while((GPIO_PORTA_DATA_R &0x4) != 0){
          xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
          if( xStatus != pdPASS )
            {
            /* We could not write to the queue because it was full - this must
            be an error as the receiving task should make space in the queue
            as soon as both sending tasks are in the Blocked state. */
            //vPrintString( "Could not send to the queue.\n" );
            }
        }
      }
      counter=0;
      xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
          if( xStatus != pdPASS )
            {
            /* We could not write to the queue because it was full - this must
            be an error as the receiving task should make space in the queue
            as soon as both sending tasks are in the Blocked state. */
            //vPrintString( "Could not send to the queue.\n" );
            }

      //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)-1);
    }
    /*******************************************************************************************
    if(((GPIO_PORTD_DATA_R & 0x8)==0x0)&&((GPIO_PORTA_DATA_R & 0x80)==0x0)){//passenger down
      while((GPIO_PORTD_DATA_R & 0x8)==0x0){
        xStatus = xQueueSendToBack( xQueue, &down, xTicksToWait );
        counter++;
        if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
      }
```

```
            }
        }
        if(counter < 200000){
          while((GPIO_PORTA_DATA_R &0x8) != 0){
            xStatus = xQueueSendToBack( xQueue, &down, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }
          }
        }
        counter=0;
        xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
        if( xStatus != pdPASS )
        {
        /* We could not write to the queue because it was full - this must
        be an error as the receiving task should make space in the queue
        as soon as both sending tasks are in the Blocked state. */
        //vPrintString( "Could not send to the queue.\n" );
        }
      }
      xSemaphoreGive( xMutex );
    }
}
```

Queues Implementation

Task Driver

```
        }
        if(counter < 200000){
          while((GPIO_PORTA_DATA_R &0x4) != 0){
            xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }
          }
        }
        counter=0;
        xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }

        //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)-1);
      }
```

Task Passenger

```
        if(counter < 200000){
          while((GPIO_PORTA_DATA_R &0x4) != 0){
            xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }
          }
        }
        counter=0;
        xStatus = xQueueSendToBack( xQueue, &stop, xTicksToWait );
            if( xStatus != pdPASS )
              {
              /* We could not write to the queue because it was full - this must
              be an error as the receiving task should make space in the queue
              as soon as both sending tasks are in the Blocked state. */
              //vPrintString( "Could not send to the queue.\n" );
              }

        //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)-1);
```

Task Receive

```c
static void Task_Rec(){
  int rec;
  portBASE_TYPE xStatus;
  for( ;; )
    {
    if( uxQueueMessagesWaiting( xQueue ) != 1 )
    {
      //vPrintString( "Queue should have been full!\n" );
    }
    xStatus = xQueueReceive( xQueue, &rec, portMAX_DELAY );

    if( xStatus == pdPASS )
    {
      if(rec == 2){
        MotorUp();
      }
      else if(rec ==1){
        MotorStop();
      }
      else if(rec==4){
        MotorDown();
      }
    }
    else
    {
    }
  }
}
```

Semaphore Implementation

Task Safety

```c
void Task_Safety(){
  xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
        MotorDown();
        for(int i=0;i<1000000;i++);
        MotorStop();
        xSemaphoreGive( xBinarySemaphore );
    }
}
```

ISR Giving the Semaphore to Task Safety

```c
void GPIOF_Handler(void){
  GPIO_PORTF_ICR_R  =0x1;
  portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Clear the software interrupt bit using the interrupt controllers
    Clear Pending register. */
    //GPIO_PORTF__R  =0x1;

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the Cortex M3 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

Mutex Implementation(Queue Shared Resource)

Task Driver

```c
void Task_Driver(){
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
int down=4;
int up=2;// up
int stop=1;//auto stop

  for(;;){
    xSemaphoreTake( xMutex, portMAX_DELAY );
    if((GPIO_PORTD_DATA_R & 0x4)==0x0){//driver up
      //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)+1);
      while((GPIO_PORTD_DATA_R & 0x4)==0x0){
        xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
        counter++;
        if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
      }
      if(counter < 200000){
        while((GPIO_PORTA_DATA_R &0x4) != 0){
          xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
          if( xStatus != pdPASS )
            {
```

Task Passenger

```
void Task_Passenger(){
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
int down=4;
int up=2;// up
int stop=1;//auto stop
int counter=0;
  for(;;){
    xSemaphoreTake( xMutex, portMAX_DELAY );

    if(((GPIO_PORTD_DATA_R & 0x2)==0x0) && ((GPIO_PORTA_DATA_R & 0x80)==0x0)){//passenger up
      //vTaskPrioritySet(NULL,uxTaskPriorityGet(NULL)+1);
      while((GPIO_PORTD_DATA_R & 0x2)==0x0){
        xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
        counter++;
        if( xStatus != pdPASS )
          {
          /* We could not write to the queue because it was full - this must
          be an error as the receiving task should make space in the queue
          as soon as both sending tasks are in the Blocked state. */
          //vPrintString( "Could not send to the queue.\n" );
          }
      }
      if(counter < 200000){
        while((GPIO_PORTA_DATA_R &0x4) != 0){
          xStatus = xQueueSendToBack( xQueue, &up, xTicksToWait );
```

Problems Faces

- Hardware Issues : bad wires, insufficient voltage
- Task Rec isn't blocked when queue is empty
    - Fix: Macro was given a wrong value, delay parameter was set to zero in the QueueReceive function parameter, to fix this, delay is set to portMAXDELAY.

System Flowchart

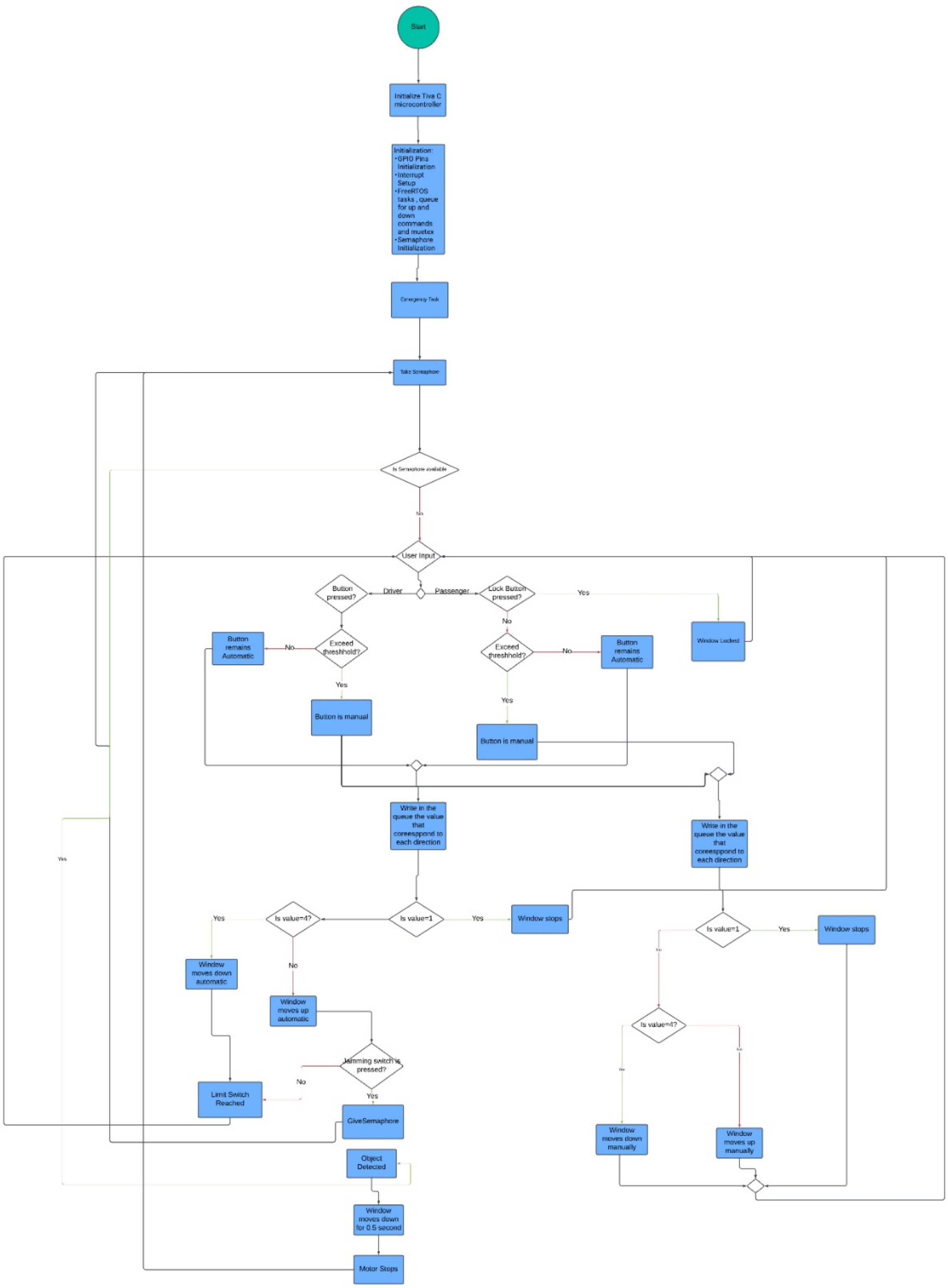

Figure 10: System Flowchart

Conclusion:

In summary, the power window control system project has been a comprehensive exploration into the integration of advanced electronic control systems in automotive applications. Through the utilization of the Tiva C microcontroller platform and FreeRTOS, we have developed a robust and efficient system for controlling the front passenger door window.

Reflecting on the journey of this project, several key insights have emerged. We encountered various challenges throughout the design and implementation phases, ranging from hardware compatibility issues to software optimization constraints. However, each obstacle served as an opportunity for learning and growth, ultimately leading to the successful realization of our objectives.

One of the most significant takeaways from this project is the importance of system reliability and safety. The incorporation of limit switches and obstacle detection mechanisms has proven essential in ensuring the smooth operation of the window control system while mitigating potential hazards. This emphasis on safety aligns with contemporary automotive design principles and underscores our commitment to user-centric engineering.

Looking ahead, there are several avenues for future improvements and extensions to the project. Enhanced diagnostics capabilities, such as real-time feedback on window position and health status, could provide valuable insights for maintenance and troubleshooting. Additionally, exploring advanced control algorithms and sensor fusion techniques may further optimize the system's performance and responsiveness.

In conclusion, the power window control system project exemplifies the synergy between innovative technology and practical automotive engineering. By embracing the challenges and opportunities inherent in modern embedded systems design, we have delivered a solution that not only meets functional requirements but also prioritizes user safety and satisfaction. As we continue to push the boundaries of technological innovation, the lessons learned from this project will serve as guiding principles for future endeavors in the dynamic field of automotive electronics.