



**POWER WINDOW CONTROL SYSTEM using Tiva C running FreeRTOS**

Real-time and Embedded Systems Design- CSE411

**Submitted to**

Prof. Sherif Hammad - Eng. Ayman Baharia

**Team: 21**

|  |  |
| --- | --- |
| Ahmed Wael Ibrahim Mohamed | 20P3343 |
| Mustafa Usama Abdelrahman | 20P5556 |
| Ahmed Sameh Mahmoud Nabieh | 20P2914 |
| Farah Ahmed Abdelrehim Tharwat | 20P1269 |
| Salma Mohamed Elsoly | 20P6375 |

Contents

[Introduction 3](#_Toc166256581)

[Project Scope 3](#_Toc166256582)

[System Basic Features 3](#_Toc166256583)

[System Architecture 4](#_Toc166256584)

[Software Components Layout 4](#_Toc166256585)

[FreeRTOS Tasks Structure 5](#_Toc166256586)

[Window Task 5](#_Toc166256587)

[Motor Task 8](#_Toc166256588)

[Commands and Queues 8](#_Toc166256589)

[Semaphores and Jamming 9](#_Toc166256590)

[Demonstration 10](#_Toc166256591)

[Hardware Items List 10](#_Toc166256592)

[Circuit Wiring and System Configurations 11](#_Toc166256593)

# Introduction

In the realm of modern vehicle technology, the Power Window Control System stands out as a pivotal component, elevating passenger convenience and safety. Our endeavor centers on crafting a versatile system capable of seamless integration with a spectrum of vehicle systems. Utilizing the Tiva C microcontroller platform and fortified with the scalability of FreeRTOS for task management, our focus lies on developing a solution that is both generic and adaptable. This approach aims to ensure compatibility across different vehicle models, ultimately enhancing the overall passenger experience.

## Project Scope

The primary objectives of our project are as follows:

1. **Front Passenger Door Window Implementation:** We aim to create a reliable and generic power window system for passengers and driver windows. Both the passenger and driver control panels will have seamless access to window operation.
2. **Power Window (Manual and Automatic Functionalities):** We aim to provide manual and automatic window movement function with the utilization of simple push-buttons or any cheap available switches.
3. **Generic System:** We aim to build a configurable, generic and scalable system conforming to industry standards, helping the software system to be installed on different vehicle systems of different designs.
4. **FreeRTOS Integration:** We aim to build the system over FreeRTOS open-source kernel, to leverage and demonstrate the power of parallelized tasks execution.
5. **Limit Switch Implementation:** To ensure safe window operation, we will incorporate **two limit switches**. These switches will prevent the window motor from exceeding the top and bottom limits of the window frame.
6. **Obstacle Detection:** While a current stall sensor is unnecessary, we will employ a simple push-button mechanism to detect obstructions. If an obstacle is detected during the one-touch auto-close operation, the system will halt the window movement within **0.5 seconds**.

## System Basic Features

Our power window control system will offer the following features:

1. **Manual Open/Close Function:**
   * When the power window switch is continuously pushed or pulled, the window will open or close until the switch is released.
   * This feature provides flexibility for precise window positioning.
2. **One-Touch Auto Open/Close Function:**
   * A short press of the power window switch will fully open or close the window.
   * Passengers can effortlessly adjust the window position without continuous input.
3. **Window Lock Function:**
   * Enabling the window lock switch will disable the opening and closing of all windows except the driver’s window.
   * Enhances security and prevents accidental window adjustments by passengers.
4. **Jam Protection Function:**
   * In case foreign matter obstructs the window during one-touch auto-close operation, the system will automatically stop the window.
   * Within **0.5 seconds**, the window will move downward slightly to release the obstruction.

# System Architecture

## Software Components Layout

Our project utilized the industry standard layered architecture as illustrated in the following figure.

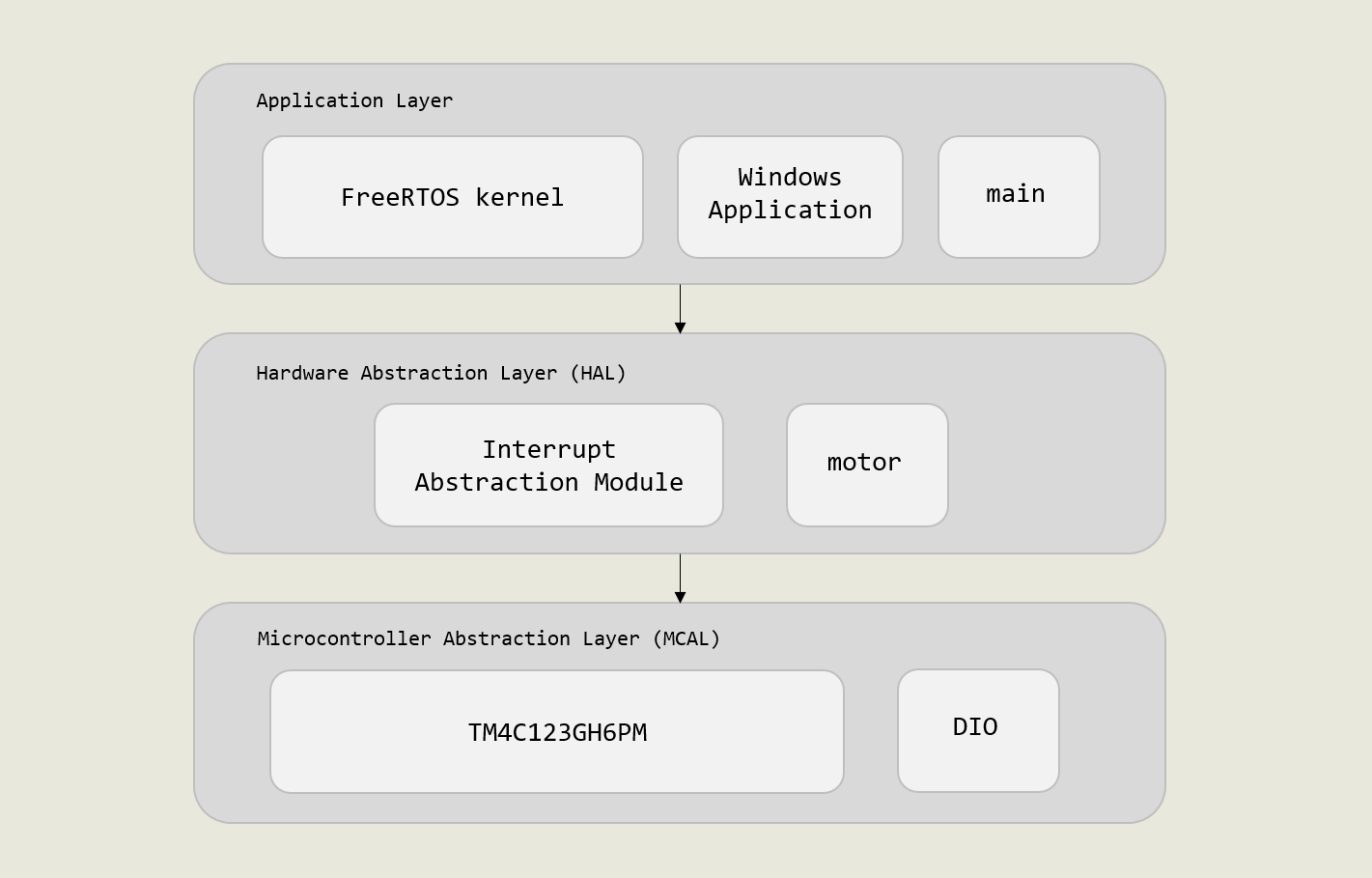


Figure 1: Layered Architecture Implementation

**Application Layer**

This is the layer wrapping the actual application code used for building the system. This layer comprises of: FreeRTOS Kernel files, Windows.h header files and main.c main function.

To create a scalable, modular and layered system, windows tasks definitions and functions are declared in a separate software module called windows.h, the main objective is to define the system and task definitions that are loaded into the FreeRTOS kernel for parallel execution. The windows module will be further explained in a later section in this document.

The main.c module initializes the FreeRTOS scheduler, system modules then loads the task definitions into the FreeRTOS application.

**Hardware Abstraction Layer**

In this layer, we have implemented generic software modules that interface with the underlying hardware components needed for the windows operation: motors and interrupt modules. It is worth noting that the motor driver module is built based on L298N H-bridge Motor Control Board that interfaces with DC Motors. A software modification might be required to re-interface with Stepper motors or any other kind of motors.

## FreeRTOS Tasks Structure

The system was structured with an object-oriented mindset. The system integrator must firstly define the number of windows present in the vehicle then adjust the corresponding static configurations in windows.h header file, this maintains system scalability where the system will define a set of generic parameterized window tasks whose number corresponds to the actual hardware requirements.

In the following diagram, the tasks are divided into: **Window Control Tasks**, **Motor Task** and **Jamming Tasks**. The system integrator can instantiate any number of generic Passenger tasks as will be explained in **Window Task**, but only 1 Motor and 1 Semaphore Jamming tasks can be instantiated. The motor and jamming tasks were built in a generic form to reduce the overhead of creating multiple tasks for multiple window control panels which might introduce a performance bottleneck in the system.

A screenshot of a computer

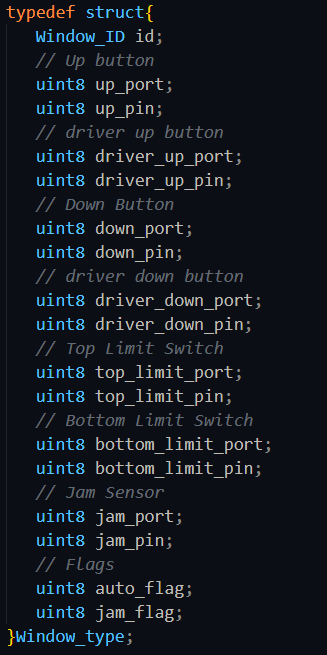
Description automatically generated

Figure 2: FreeRTOS Tasks Structure

### Window Task

This is a generic window control task responsible for handling user inputs from the window control panel. The task is generic meaning that the tasks is only defined once but is instantiable to any number of times corresponding to the static configuration “**NUM\_OF\_WINDOWS**” found in windows.h. Preferably, this configuration could be **NUM\_OF\_WINDOWS** = 2 or 4 to conform to standard vehicle designs.

The task parameters is sent as a structure comprising the necessary input controls GPIO ports and pins, as illustrated in the following snapshot.

* **Window\_id:** Is an enumeration used to identify window task, as an example (Driver Task has the id: 0, while Passenger1 Task has the id: 1).
* **GPIO ports:** These are a group of 8-bit variables that identify the GPIO pin corresponding to each panel input.
* **auto\_flag, jam\_flag:** These are Boolean flags used to identify the automatic state of the system and facilitate the jamming operation.

This parameter structure approach facilitates building decoupled tasks, meaning system integrators nor developers are required to define additional functions in the system to add hardware components.

In this sense, the system initialization simply initializes the GPIO pins for panels and fills the structure, an the user can instantiate multiple task functions only by appropriate construction of the parameters.

To automate the task initialization for system integrators, a public array of windows is defined in windows.h and can be externed in the main function to only interate through the array and create a corresponding task as illustrated in the following code example.

|  |
| --- |
| *// Variables are externed from module -- Check for documentation*  extern Window\_type windows[NUM\_OF\_WINDOWS];  extern QueueHandle\_t xCommandQueue;  extern xSemaphoreHandle xBinarySemaphore;  int main( void )  {  *// Initializing Windows GPIO*      WINDOW\_init();  *// Initializing commands queue*      xCommandQueue = xQueueCreate(20, sizeof(uint8\_t));  *// Initializing binary semaphore*      vSemaphoreCreateBinary(xBinarySemaphore);      if (xBinarySemaphore != NULL){  *// Semaphore is created successfully*          for (uint8 i=0; i<NUM\_OF\_WINDOWS; i++){  *// Create Passenger Task (PASSENGER1, PASSENGER2, PASSENGER3*              xTaskCreate(                  WINDOW\_PassengerTask,                  "Window Task",                  300,                  (void \*)&windows[i],                  1,                  NULL              );          }  *// Create Motor Task*          xTaskCreate(              WINDOW\_MotorTask,              "Window Task",              300,              NULL,              1,              NULL          );  *// Create jamming semaphore task of higher priority*          xTaskCreate(              WINDOW\_JammingSemaphoreTask,              "Window Task",              300,              NULL,              3,              NULL          );  *// Start the scheduler and run tasks*          vTaskStartScheduler();      }  */\* If all is well we will never reach here as the scheduler will noe*  *be running the tasks. If we reach here then it is likely that there was*  *insufficient heap memory available for a resource to be created \*/*      for(;;);  } |

The windows task main aim is to read the user’s input by reading the GPIO pins declared in the parameter structure, process the inputs to define the operation to be executed then communicate a command to the motor task to move interface with the motor hardware.The task communication process is further explained in the **Commands and Queues** section.

The following features are implemented in this module as follows:

* **Manual Control:** the GPIO pin is read then vTaskDelay() is called to wait for a specific period of time, then re-read the GPIO pin. A post-delay ON reading on the GPIO pin indicates the user’s continuous press on the switch triggering the manual movement of the windows. It is worth noting that the jamming operation is only accounted for in the automatic operation as defined in the design specifications document.
* **Automatic Control:** As opposed to Manual Control operation, a post-delay OFF reading indicates the user’s one-touch press which triggers the automatic movement of the windows.
* **xQueueSend:** The operation command is pushed into the commands queue to be continuously read by the motor task for execution.
* **Driver-to-passenger Control:** In the parameter structure, a corresponding driver\_up\_GPIO and driver\_down\_GPIO where they are also read in the generic task, however, the reading operation is conditional, meaning, it can only be read, if the private global variable g\_isLocked is on, in other words, the driver locked the windows operations. This makes the windows only controllable by the driver in case of locked state.
* **Limit Switches:** Limit switches GPIO parameters set a group of conditions that resets the task operation and force it to send a Motor\_stop command to the motor task.

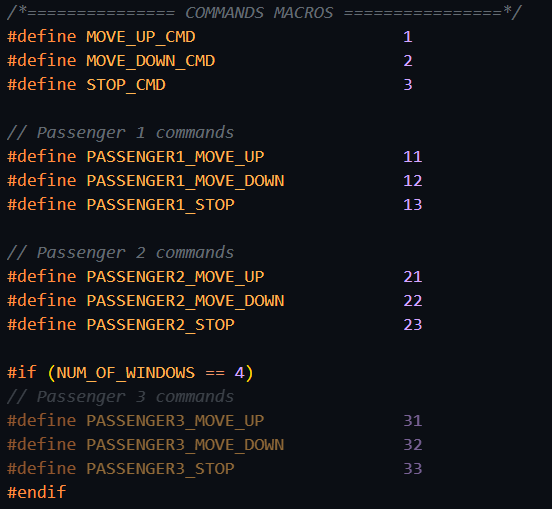
### Motor Task

The motor task simple continuously receives from the commands queue task commands, decodes the commands then call the corresponding function from the motor driver module to interface with the hardware.

### Commands and Queues

A queue is defined to create a communication channel between the window control tasks and the motor tasks to help send/receive commands and execute operations.

The following code snapshot illustrates the basic commands defined in the system (Command definitions can be found in the Operational Macros section in windows.h).

As discussed in the Window Task section, a reduction in the tasks creation overhead is achieved by creating a single generic Motor task. This is achieved by creating a commands protocol which the motor task could decode and understand which motor to be activated and with which operation.

As discussed also, the parameter structure contain a window\_id, which can be used to identify the target motor hardware, our implementation of the command is simple and goes as follows:

For example, the command for triggering PASSENGER2 windows of id=2 to move upwards would be as follows:

### Semaphores and Jamming

Jamming is considered a safety-critical issue whose priority exceeds all operations of the system, consecutively, the implementation of jamming operation is done using a higher-priority task that directly controls the motors and have access to windows array.

Referring to **Figure 2: FreeRTOS Tasks Structure** the Jamming task is initially blocked as the semaphore is not given, until the Jamming\_ISR defined in windows.c toggles the jam\_flag in a corresponding windows array element, after wise, the ISR gives the semaphore to the Jamming task then forces context switching.

The semaphore task then iterates through the array of windows till it finds a window object whose jam\_flag is on, it then assesses the auto\_flag which is only raised in case of automatic move up operation. If both flags are raised, the Jamming task forces the motor to move in the opposite direction for a statically configured JAM\_DELAY\_MS then stops the motor. It is worth noting that the Jamming task also checks for bottom\_limit\_switches to avoid moving the window outside its boundaries. The following code snapshots showcase the implementation of the semaphore task and ISR.

A computer screen shot of text

Description automatically generated

Figure 3: Jam Interrupt Service Routine

A computer code on a black background

Description automatically generated

Figure 4: Jamming semaphore task implementation

# Demonstration

## Hardware Items List

|  |  |  |
| --- | --- | --- |
| **Item Image** | **Item Name/Code** | **Functionality** |
| L298N Motor Drive Controller Board Module Dual H Bridge DC Stepper  ForArduino Smart Car Robot UNO MEGA R3 MEGA 2560 ESP32 ESP826 | Dual H-Bridge Motor Driver | Hardware control board that bridges and facilitates the connection of the DC motors with Tiva C board |
| Mini DC Gearbox Motors Pair (2 Motors) With Back Shaft | Mini DC Gearbox Motors with Back Shaft | DC Motors used to move windows in both directions, a gearbox is used to reduce the RPM speed of motors with no software intervention |
| Micro Switch - MS2 | Micro Limit Switches | Used to signal whether the window reached the maximum height limit or the maximum depth limit |
| SW4 - ON/OFF/ON Switch 3 pin | 3-level Rocker Switch | Used in the window control panel to signal up/down commands |
| Press 4pin 6x6x4.5 mm Tack Switch | Push button | Used to emulate GPIO input for Jamming Stall Sensors |
| EK-TM4C123GXL Evaluation board | TI.com | Tiva C Launchpad | Arm Cortex M4 – based microcontroller |

## Circuit Wiring and System Configurations

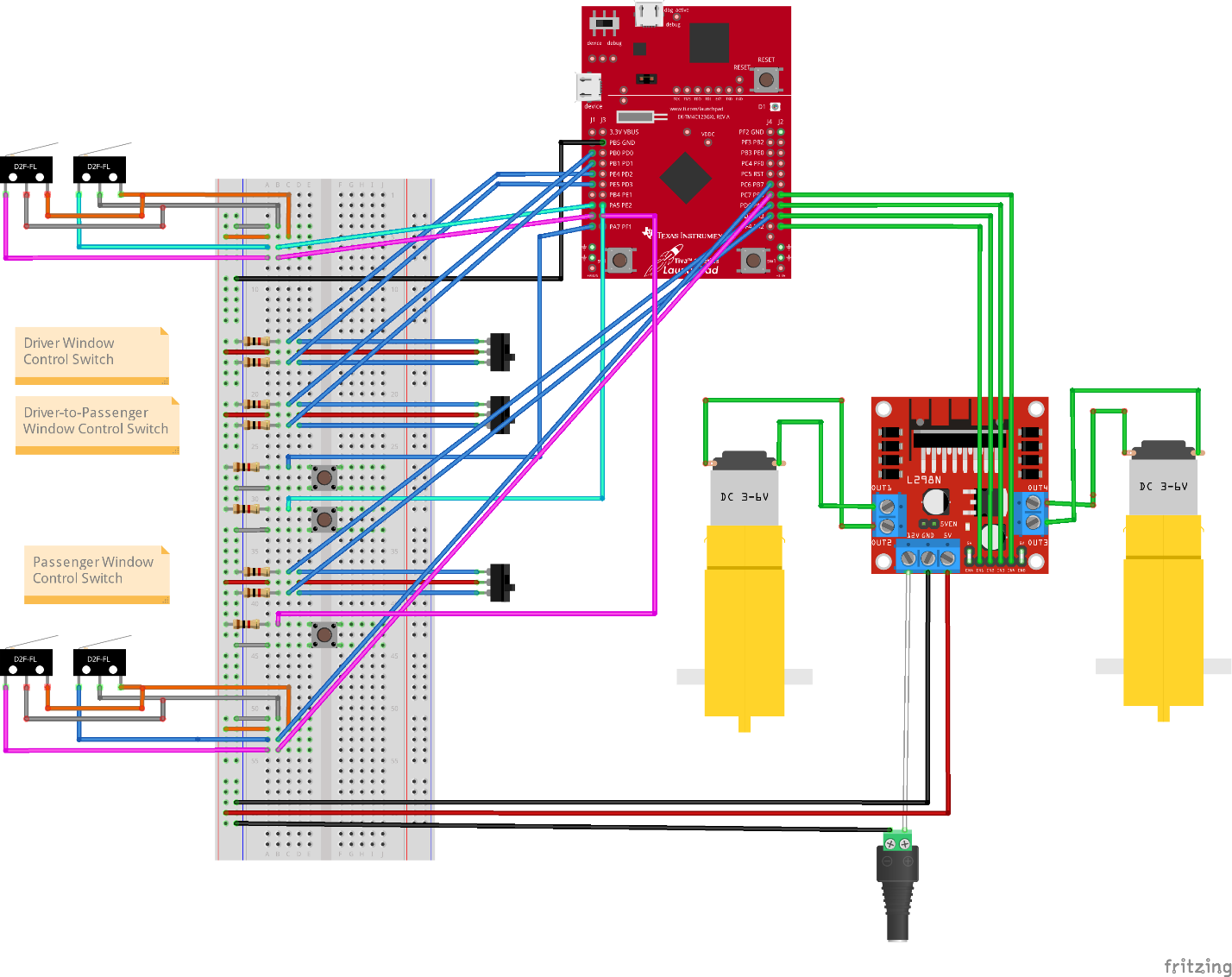


Figure 5: Demo project circuit wiring using Fritzing

System Configurations are adjusted in the statics configurations section in windows.h header file as follows

1. Firstly, adjust **NUM\_OF\_WINDOWS** = 2. In this demo project we will use the 2-seat vehicle design with only 1 Driver and 1 Passenger Control panels.
2. Adjust all **ports** and **pins** accordingly
3. Use the code sample in **Window Task** section.

The following are the static configurations used in the demo project.

|  |
| --- |
| */\*=============== SYSTEM STATIC CONFIGURATIONS ================\*/*  #define NUM\_OF\_WINDOWS     2 *// Define the number of windows in the system (4: Sedan - 2: Coupe)*  #define USED\_MOTOR\_TYPE    DC *// Define the type of motor used for windows movement*  */\*=============== DRIVER ================\*/*  *// Driver UP Button*  #define DRIVER\_UP\_PORT                    PORTE\_ID  #define DRIVER\_UP\_PIN                     PIN4\_ID  *// Driver DOWN Button*  #define DRIVER\_DOWN\_PORT                  PORTE\_ID  #define DRIVER\_DOWN\_PIN                 PIN5\_ID  *// Driver Control for passenger1*  #define DRIVER\_PASSENGER1\_UP\_PORT       PORTB\_ID  #define DRIVER\_PASSENGER1\_UP\_PIN        PIN1\_ID  #define DRIVER\_PASSENGER1\_DOWN\_PORT     PORTB\_ID  #define DRIVER\_PASSENGER1\_DOWN\_PIN      PIN0\_ID  *// Driver Lock Button*  #define DRIVER\_LOCK\_PORT                PORTB\_ID  #define DRIVER\_LOCK\_PIN                 PIN3\_ID  *// Driver Top Limit Switch*  #define DRIVER\_TOP\_LIMIT\_PORT           PORTA\_ID  #define DRIVER\_TOP\_LIMIT\_PIN            PIN6\_ID  *// Driver Bottom Limit Switch*  #define DRIVER\_BOTTOM\_LIMIT\_PORT        PORTA\_ID  #define DRIVER\_BOTTOM\_LIMIT\_PIN         PIN5\_ID  *// Driver Jam Sensor*  #define DRIVER\_JAM\_PORT                 PORTE\_ID  #define DRIVER\_JAM\_PIN                  PIN2\_ID  */\*=============== PASSENGER 1 ================\*/*  *// Passenger UP Button*  #define PASSENGER1\_UP\_PORT             PORTD\_ID  #define PASSENGER1\_UP\_PIN              PIN7\_ID  *// Passenger DOWN Button*  #define PASSENGER1\_DOWN\_PORT           PORTD\_ID  #define PASSENGER1\_DOWN\_PIN            PIN6\_ID  *// Passenger Top Limit Switch*  #define PASSENGER1\_TOP\_LIMIT\_PORT      PORTC\_ID  #define PASSENGER1\_TOP\_LIMIT\_PIN       PIN7\_ID  *// Passenger Bottom Limit Switch*  #define PASSENGER1\_BOTTOM\_LIMIT\_PORT   PORTC\_ID  #define PASSENGER1\_BOTTOM\_LIMIT\_PIN    PIN6\_ID  *// Passenger Jam Sensor*  #define PASSENGER1\_JAM\_PORT            PORTE\_ID  #define PASSENGER1\_JAM\_PIN             PIN3\_ID |