

CECS 346 Fall 2021 Project 2

Drag Race

By

Dylan Dang

December 10, 2021

The purpose of this project was to design a system based on the ‘Christmas Tree’, the light signal that drivers use to determine when to accelerate, which is simulated with LEDs. Implementation of the Moore Finite State Machine, SysTick Timer/ Interrupt, Edge Interrupt, and GPIO was necessary. Two sensors will be used for each lane of the system and the winner will be determined by whoever releases their respective lane sensor button the fastest after the green LEDs are lit. A reset button is implemented to restart the system back to the initial state of the FSM.

**Introduction**

The objective is to simulate the ‘Christmas Tree’ and to determine a winner. There is a clear left and right lane in the system and each ‘driver’ will be responsible for their respective lane sensor button.

There are 2 lane sensor buttons that will be divided to the right and left lane of the drag race system. The driver of each lane has their own lane sensor button and is responsible for pushing/releasing the button. To initiate the sequence of light signals, both drivers must hold down their respective lane sensor buttons simultaneously. The goal of both drivers is to release their button the fastest after the green LED has been lit. If either or both drivers have released their button before the green LED is lit, the red LED will light up in the respective lane(s) of whichever driver was responsible. Both drivers will have to hold down their buttons simultaneously to reinitiate the sequence. Both the lane sensor buttons will be implemented with a GPIO interrupt of priority 2 to be detected on both edges.

There is also a reset button implemented in the system. When pressed, all 8 LEDs of the system will light up, signaling the initialization state. This state also resets the SysTick Timer. After 1 second, the system will transition to the state where no LEDs are lit, the wait for staging state, waiting for both lane sensor buttons to be held down simultaneously. The reset button will be implemented with a GPIO interrupt of priority 1 to be detected on level sensitive.

To implement the ‘Christmas Tree’, there will be 8 LEDs interfaced through the GPIO ports to simulate the different states of the light signals. Since there are two lanes in the system, left and right, each lane will get 4 LEDs. Each lane will have 2 yellow LEDs, 1 green LED, and 1 red LED. Yellow LED tells the drivers to wait. Green LED tells the drivers to release their lane sensor button. Red LED tells the drivers that one or both has released their lane sensor button too early/before the green LEDs has lit. The LEDs will always transition from Yellow 1 (Y1R & Y1L) – Yellow 2 (Y2R & Y2L) – Green (GL, GR). If either or both drivers have released their lane sensor button before the green LEDs has lit up, the system will immediately light up the red LED (RR, RL) of that respective driver(s).

The time durations of each LED state for both lanes are: Yellow 1 > 0.5s > Yellow 2 > 0.5s > Green. Anytime during the Yellow 1 or Yellow 2 LED, if either lane sensor button is released the system will transition to the False Start state and the red LED will light up and will transition back to Wait for Staging state. During the Green LED state, the system will check every 0.05 seconds to see if either button is released. If left driver releases first, the left green LED will light up. If right driver releases first, the right green LED will light up. If both drivers release at the same time, the left and right green LED will light up.

**Operation**

*SENSOR\_PORTA* is used to control PA2 & PA3 which controls the right lane sensor and left lane sensor respectively. While in the Wait for Staging state, pressing, and holding both lane sensor buttons is the only way for the state transitions to happen. Holding only of the buttons will not transition the state of the system. When both PA2 & PA3 are held, the Yellow 1 LEDs of both lanes will light up. If one of the drivers release their button, the system will transition to the False Start state and the red LED will light up on the lane of the driver responsible. The red LEDs of both lanes could light up if both drivers released their buttons simultaneously during the Yellow 1 state. The same could be said during the Yellow 2 state. During the False Start state, both drivers will have to hold both their buttons again to restart the light signal. In the Green state, the driver that releases their button the fastest will have the Green LED lit up in their lane, and the Green LED in the loser’s lane will turn off. Both Green LEDs will be lit if both drivers release their button simultaneously. The lane sensor buttons operate on negative logic and the reset button operates on positive logic.

**Link for Video Demonstration:** https://youtu.be/n-gslkqV1Zw

**Theory**

Implementation of the 8 LEDs of the system, lane sensor buttons, and reset button was not possible without the complete understanding of the GPIO. For example, *SENSOR\_PORTA*, which controls the lane sensor buttons, was interfaced digitally so it was crucial to enable digital signals on ports used by *SENSOR\_PORTA*. Learning to enable the pull-down resistors via GPIO for the ports used by *SENSOR\_PORTA* would have spared us two resistors.

Since using a software loop delay was efficient, but not accurate, learning to implement a hardware timer was necessary for the traffic light controller. The SysTick Timer was used due to its ability to generate accurate timing for an embedded system. Since there is a delay of 0.25s, using a SysTick Timer was a must.

Learning both the SysTick Interrupt and Edge Interrupt was necessary for this system to work. An interrupt is the automatic transfer of software execution in response to a hardware event (trigger) that is asynchronous with current software execution. Since the winner is decided by whoever releases their buttons the fastest during the Go state, having the most precise method will provide a fair playing field for the players and will also result in better efficiency within the system than using if-else statements.

Understanding the Moore Finite State Machine was the most important perquisite for this project. Although the project could be done with conditionals, using the FSM makes the process of implementing the traffic light controller more efficient and simpler. Using an FSM makes debugging each state effortless as removing/adding a state does not require multiple conditionals as it only takes one line to do with the FSM. With a FSM, each combination of the *SENSOR* outputs can be accounted for immediately.

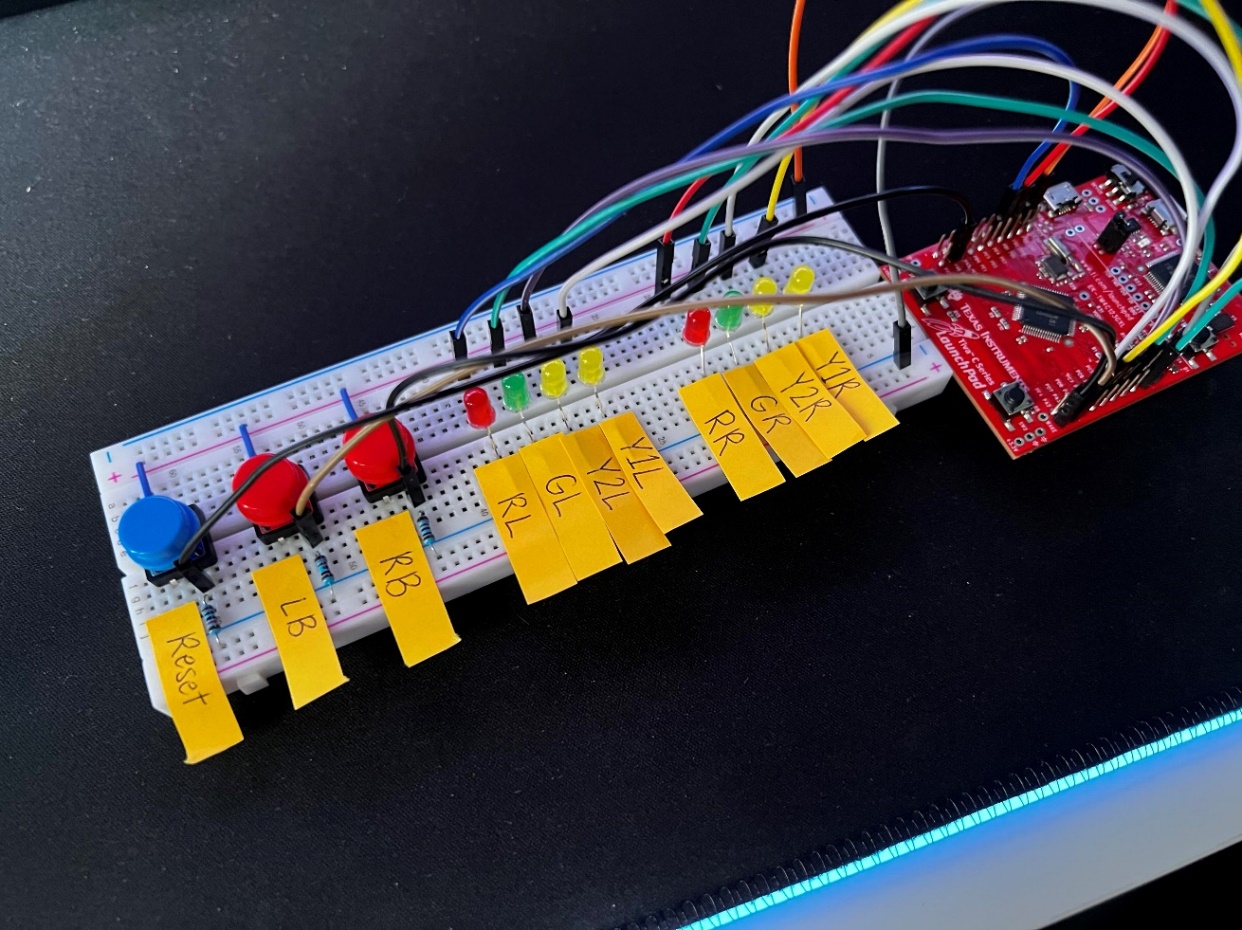
**Hardware Design**

Schematic:

Diagram

Description automatically generated

Board:



**Software Design**

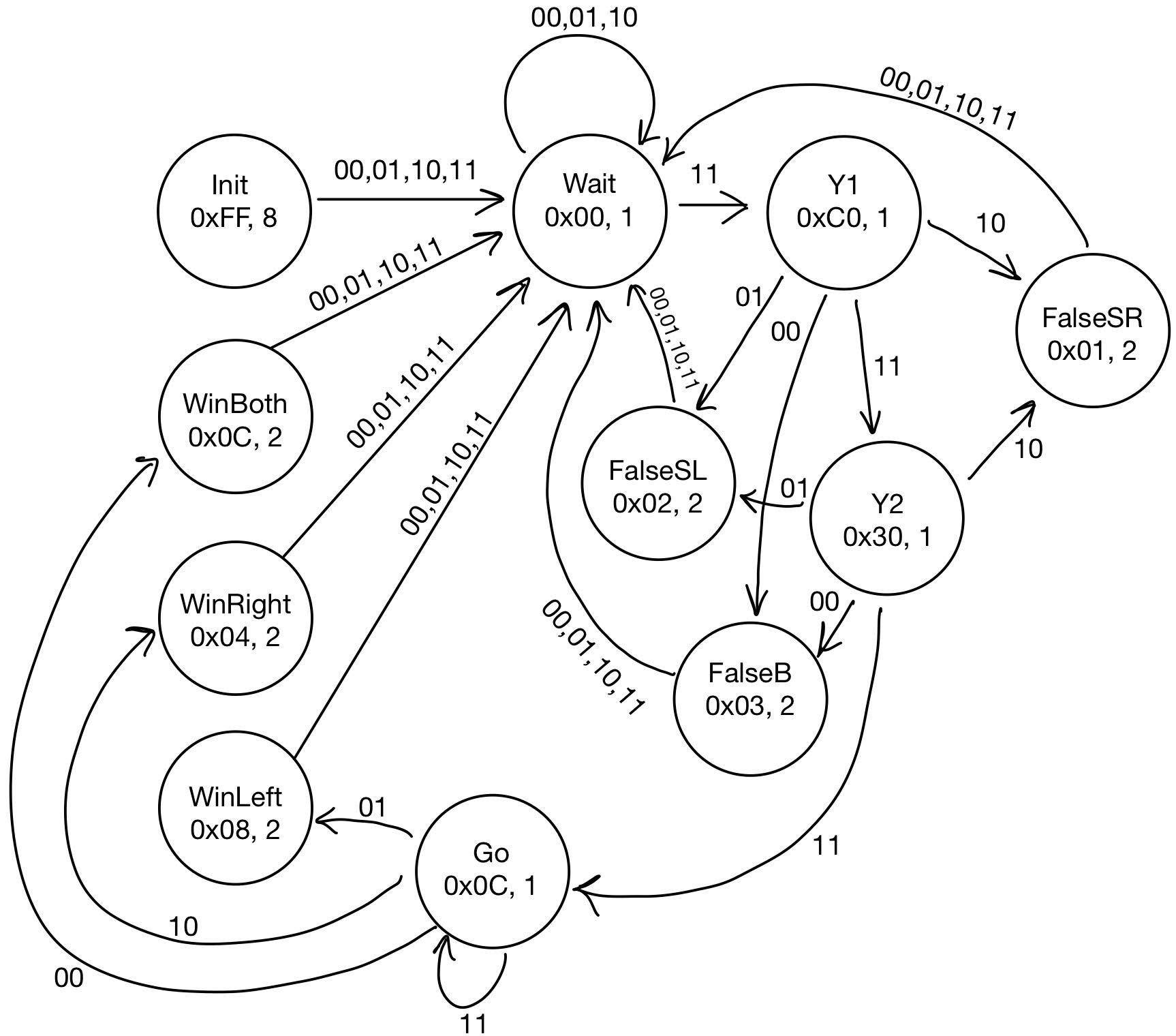
|  |  |
| --- | --- |
| LEDs | PORT |
| Yellow 1 Left (Y1L) | PC7 |
| Yellow 2 Right (Y1R) | PC6 |
| Yellow 2 Left (Y2L) | PC5 |
| Yellow 2 Right (Y2R) | PC4 |
| Green Left (GL) | PB3 |
| Green Right (GR) | PB2 |
| Red Left (RL) | PB1 |
| Red Right (RR) | PB0 |

|  |  |
| --- | --- |
| Sensors/button | port |
| Right Lane Sensor Button (RB) | PA2 |
| Left Lane Sensor Button (LB) | PA3 |
| Reset Button (Reset) | PE3 |

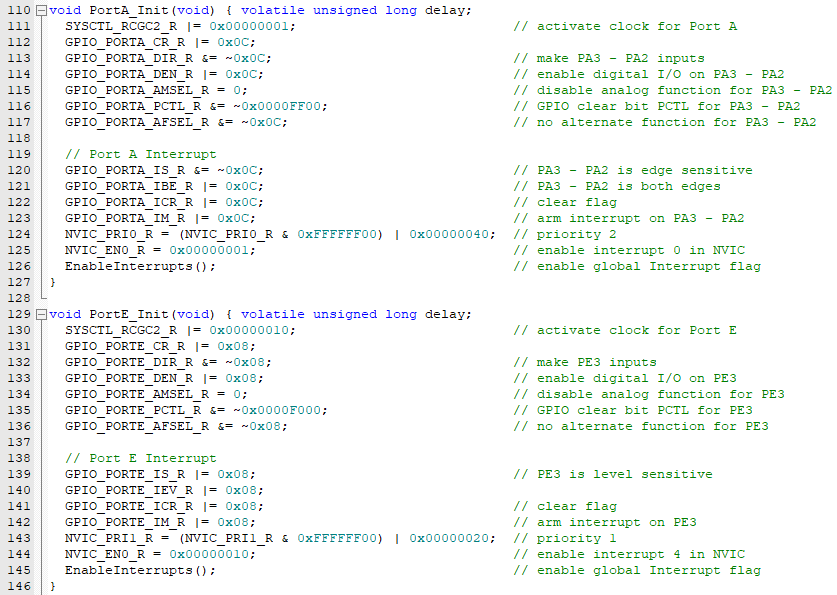
**Moore FSM: State Table**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **State Index** | **Current State** | **Time (0.5s)** | **Output (Port B & C)** | **00 (left, right pressed)** | **01 (right released)** | **10 (left released)** | **11 (both released)** |
| **0** | **Init** | **2** | **0xFF** | **Wait** | **Wait** | **Wait** | **Wait** |
| **1** | **Wait** | **1** | **0x00** | **Wait** | **Wait** | **Wait** | **Y1** |
| **2** | **Y1** | **1** | **0xC0** | **FalseB** | **FalseSL** | **FalseSR** | **Y2** |
| **3** | **Y2** | **1** | **0x30** | **FalseB** | **FalseSL** | **FalseSR** | **Go** |
| **4** | **Go** | **1** | **0x0C** | **WinBoth** | **WinLeft** | **WinRight** | **Go** |
| **5** | **FalseSL** | **2** | **0x02** | **Wait** | **Wait** | **Wait** | **Wait** |
| **6** | **FalseSR** | **2** | **0x01** | **Wait** | **Wait** | **Wait** | **Wait** |
| **7** | **FalseB** | **2** | **0x03** | **Wait** | **Wait** | **Wait** | **Wait** |
| **8** | **WinLeft** | **2** | **0x08** | **Wait** | **Wait** | **Wait** | **Wait** |
| **9** | **WinRight** | **2** | **0x04** | **Wait** | **Wait** | **Wait** | **Wait** |
| **10** | **WinBoth** | **2** | **0x0C** | **Wait** | **Wait** | **Wait** | **Wait** |

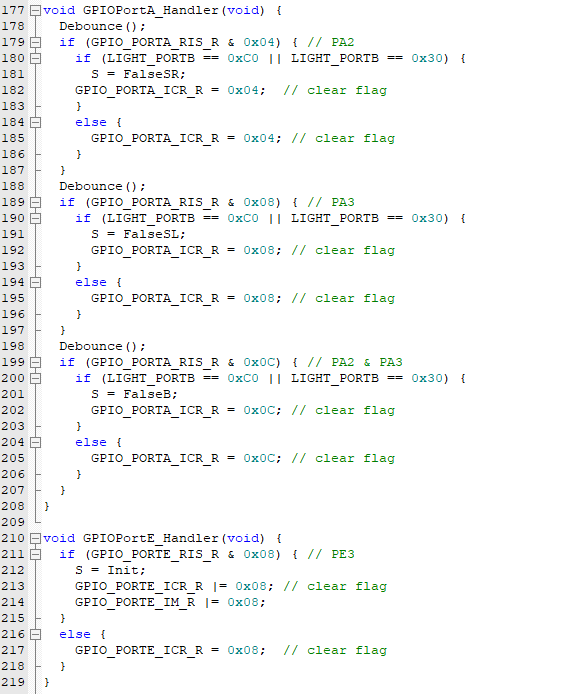
**Moore FSM: State Transition Diagram**

****

**Edge Interrupt: Port A & Port E**

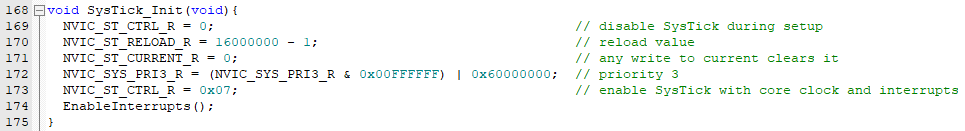


Port A and Port E are both inputs utilizing the edge interrupt function. I was tasked to make the lane sensor buttons, which is interfaced to Port A, run on an edge sensitive, both edges triggered interrupt. By setting GPIO\_PORTA\_IS\_R to a 0, this tells the system that we want the trigger to be edge triggered and not level sensitive. Setting GPIO\_PORTA\_IBE\_R to a 1, tells the system that we want both edges to trigger the interrupt. Line 124, NVIC\_PRI0\_R is set in a way to make sure it has a priority level of 2. For the reset button, interfaced to Port E, I was tasked with making it level sensitive. To do that, GPIO\_PORTE\_IS\_R is set to a 1, indicating level sensitive trigger. Since it is level sensitive, there is no need to include GPIO\_PORTE\_IBE\_R since that concerns with edge triggered interrupts. The priority for the reset button has to be priority 1 which is a level higher than that of the lane sensor buttons. Line 143, NVIC\_PRI1\_R is set in a way to make it priority 1.

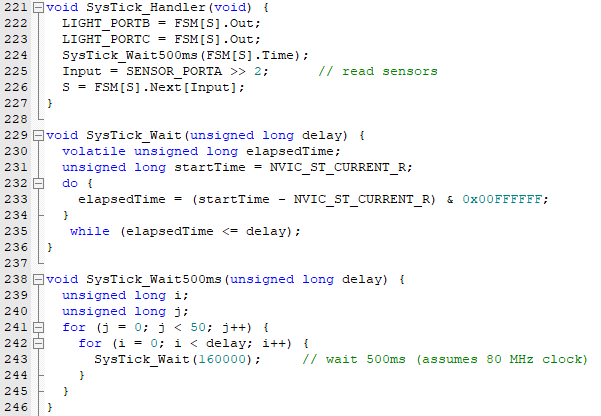
**Edge Handlers: Port A & Port E**

Because edge triggered interrupts are used, there needs to be a handler function to handle every time there is an interrupt. Since the lane sensor buttons operates on negative logic, the Port A handler function checks if the buttons are held during the countdown sequence (when the Yellow LEDs are lit), if not held, the state transitions to a false start. Since the reset button also operates on positive logic, whenever the reset button is pressed, the state transitions to the Init state. Clearing the flag in the ISR is necessary after every interrupt acknowledgement or else there will be endless interrupts. Debounce makes sure code is triggered once per user input.

**SysTick Timer Initialization**

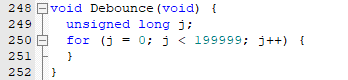
****

In the SysTick\_Init(void) function, RELOAD is set to ‘16000000– 1’ to get the desired time of 0.5 seconds. Line 173, NVIC\_SYS\_PRI3\_R, is set to that value to get a priority level of 3. Interrupts are enabled so that the timer is synced with the LEDs whenever the lane sensor buttons interrupt.

**SysTick Timer & Handler Function**

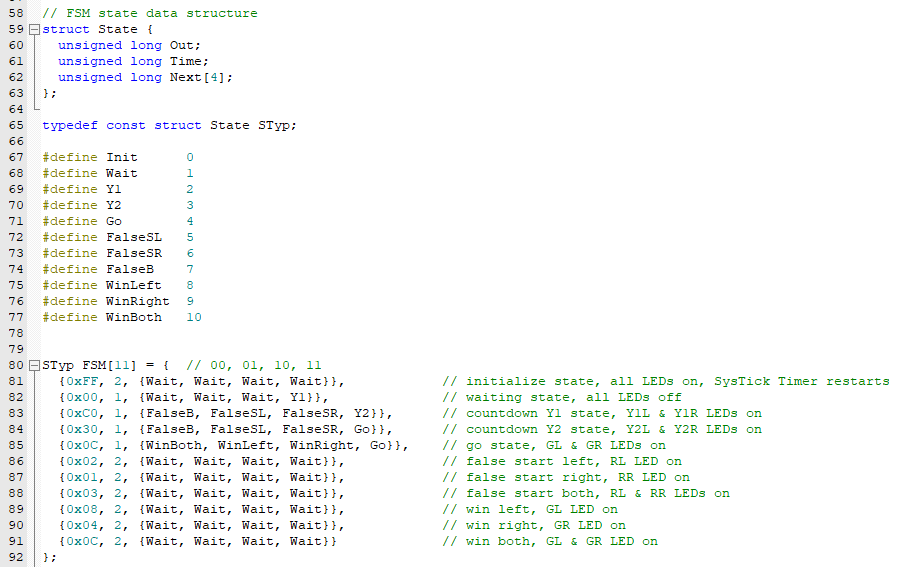
Since we are using SysTick interrupt, a SysTick handler function is needed to execute code whenever there is a SysTick interrupt. LIGHT\_PORTB and LIGHT\_PORTC, the two ports responsible for the 8 LEDs, receive information from the FSM to determine which LED(s) will light up. The SysTick\_Wait500ms function is there to ensure the correct amount of delay depending on which state the FSM is currently in (i.e., 0.5 sec or 1 sec). Bit shifting by 2 was necessary to correctly read the inputs of the buttons. Line 226 will determine which state the FSM should be in. The SysTick\_Wait function is the foundation for the SysTick\_Wait500ms function. Assuming that the board is LaunchPad is running at 80 MHz, 160000 is the value used to get the 0.5 second delay.

**Debounce Function:**

****

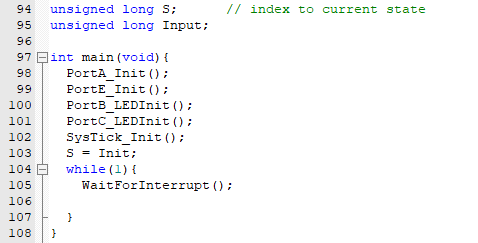
The Debounce function ensures that the code executed in the handler functions will execute only once per interrupt. The function will go on for this certain number of increments to make sure there is no multiple input readings from one press of the lane sensor buttons.

**Definition of FSM & Structure:**

****

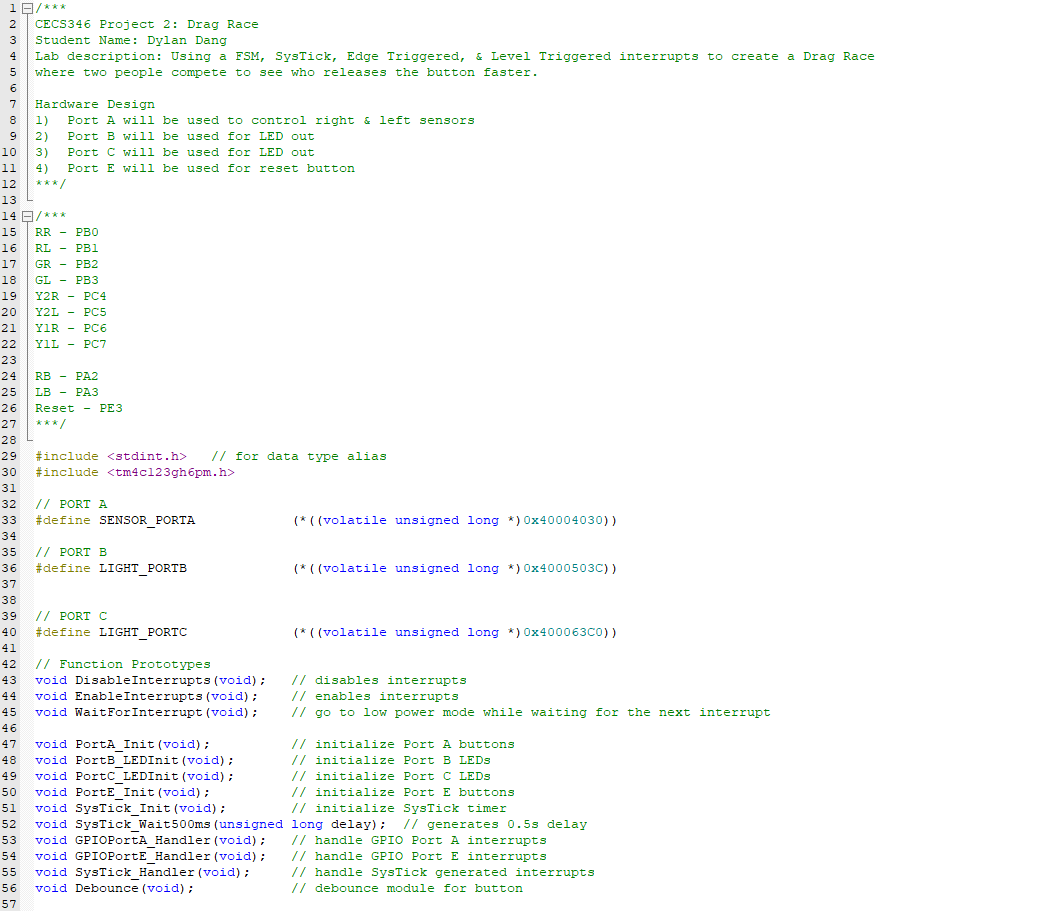
There are a total of 11 states within the FSM. The start state of the FSM is the Init state, where all 8 LEDs are lit. For each state there is an output and a delay. Since the Init state runs for 1 sec, a delay value of 2 is needed as 2\* 0.5 sec = 1 sec. The comments besides the FSM describes which LEDs are lit during each state of the FSM.

**FSM Engine:**

****

The main function is where the inputs and outputs are declared which also includes the SysTick timer. The FSM is also set to the starting state, Init. Since interrupts are used for this project, there is no reason to have any loop or code execution within the while loop besides the WaitForInterrupt command.

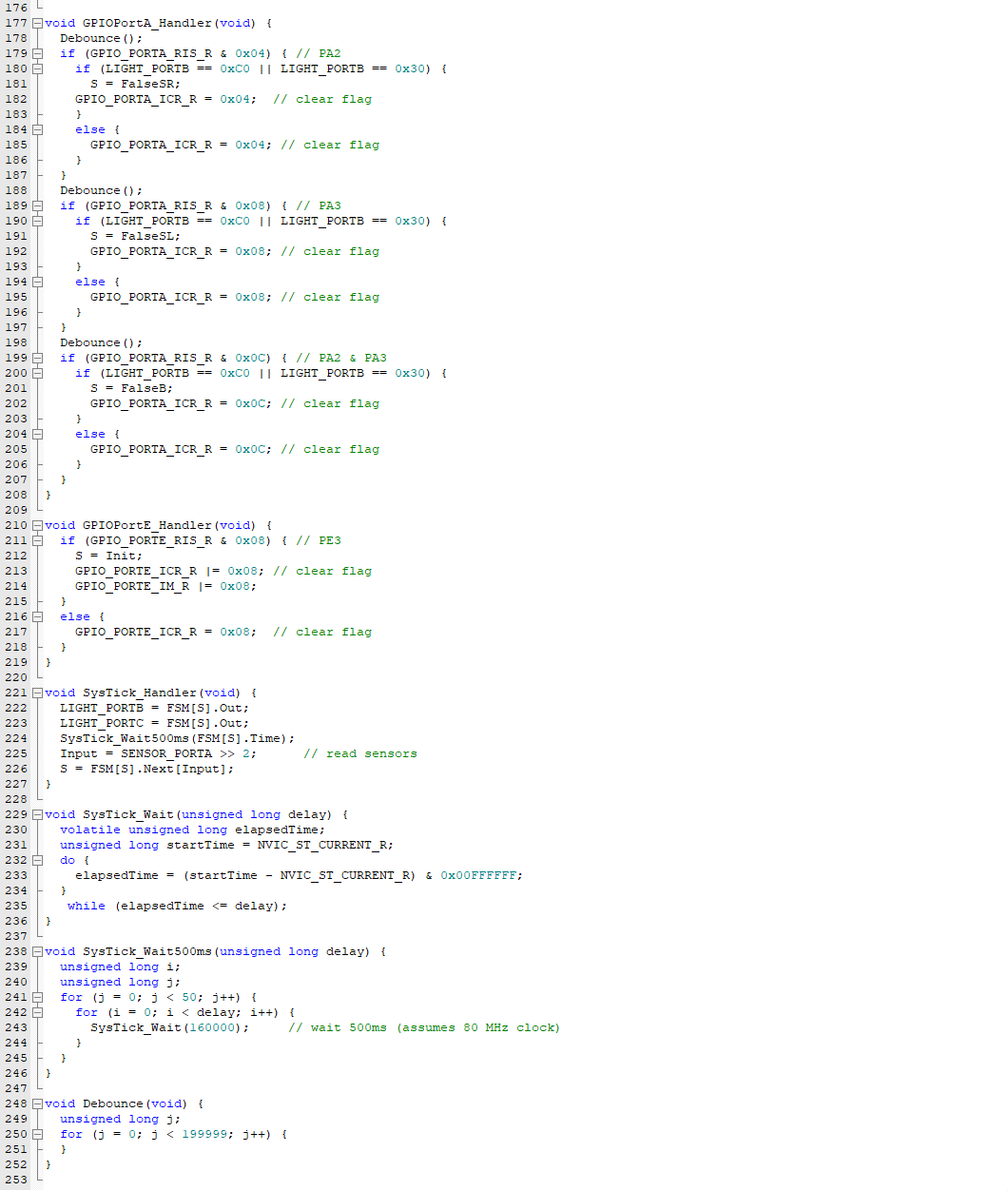
**Source Code:**

****

**Text

Description automatically generatedTable

Description automatically generated**

****

**Conclusion**

The project this time around was not as difficult as I thought it would be. Understanding the GPIO, FSM, and SysTick timer from previous labs and project made the process clear. Importing the tm4c123gh6pm.h file meant less clutter in the project as I didn’t have to define a lot of the GPIO port definitions and the SysTick timer definitions. Most of time spent during the process was on the implementation of the edge interrupt and the handler functions. Because this was the newest topic I have been introduced to, I had to do research and learn the behaviors of the interrupt. Overall, the lectures slide and zoom recordings taught me a lot about this topic and gave me the confidence to complete this project. I learned the importance of interrupts during this assignment and that many things in life such as our computers and mobile phones use interrupts to execute code efficiently.