University of Victoria

Department of Electrical Engineering

ECE 455 - Project 1 - Traffic Light System

Isaac Northrop - V00976549

Liam Tanner - V00982393

Submission Date: March 6th, 2025

Table of Contents

Introduction

The objective of this project is to create a traffic light simulation using hardware, middleware and, FreeRTOS using C code implemented on an STM32 microcontroller [1]. The traffic lights are simulated by a green, yellow and, red LED. The road is simulated by 19 individual green LEDs, where LEDs that are on represent cars, and LEDs that are off represent the gaps between cars. It follows that when the light is green, the cars progress at a consistent rate along the road, and when the light is yellow or red, the cars behind the stop light stop, and the cars after the light continue at a consistent rate. Traffic flow is controlled by a potentiometer, which when set to a minimum resistance produces cars with a 5-6 car gap and a maximum red light length. When set to maximum resistance, bumper-to-bumper traffic is produced, and the green light length is at a maximum.

Middleware

GPIO Pins

To allow our code to interact with the road of LEDs and the traffic light LEDs, we used several GPIO pins. The full mapping of the GPIO pins to their purposes can be found in Table 1 below.

Port	**Purpose**
PC0	Red Light
PC1	Yellow Light
PC2	Green Light (Traffic)
PC3	Potentiometer Input
PC4	Shift Register Clear
PC5	Shift Register Clock
PC6	Shift Register 1 Data In

Table 1: GPIO Port Mappings

Wiring and Systems

The previously listed GPIO pins were connected to a breadboard to create a circuit with 19 ‘road’ LEDs, 3 ‘traffic light’ LEDs, 3 shift registers, a potentiometer and, 23 resistors. These elements can be divided into 3 different functionalities:

1. The potentiometer and ADC.
2. The traffic light LEDs.
3. The road and shift registers.

Potentiometer and ADC

The potentiometer is connected to the circuit through 3 pins, the first two being simply ground and power. Although the potentiometer ground pin can be connected directly to an STM32 ground pin, the power pin must be connected in series with a 3,300 Ω resistor through which the 5 Volt power supply must flow to ensure no damage occurs. The third potentiometer pin is responsible for outputting the analog resistance value to the appropriate GPIO (see Table 1). Since this output is in analog form, we must convert it to a digital value to calculate both traffic flow and delay values. This is accomplished by placing this value in the ADC conversion register.

Traffic Light LEDs

The wiring to control the traffic light LEDs is very simple. As with any LED, the cathodes were connected to the ground. Each anode was then connected to their own GPIO pin through a 300 Ω resistor in order to prevent damage to the LED. The full mapping of GPIO pins to their specific traffic light colour can be seen in Table 1.

The Road and Shift Registers

To control traffic flow along our road of 19 LEDs, we use shift registers, daisy-chained together to continually update the 19-bit binary number representing the LEDs. To synchronize all 3 shift registers, the clock of all 3 shift registers is connected to the same GPIO pin. The clear ports of each shift register are also connected to a singular GPIO pin for the same purpose.

The first shift register in the daisy chaining sequence is responsible for handling the first 8 LEDs which represent cars before the intersection. The current value of the binary number for the road is sent from the GPIO pin PC6, into the first shift register through the data in port. To receive this data, each LED in this first segment of the road is connected to a different shifter register output pin; these being: A, B, C, D, E, F, G, H. The binary data is then daisy-chained from the H pin of the shift register to the data in the port of the next shift register. This second register is responsible for handling the 3 LEDs that represent cars in the intersection. Since this shift register handles only 3 cars, only the A, B, and C output pins are needed and the C pin is used for daisy chaining to the final shift register. This final register is wired nearly identically to the first shift register although its output pins are connected to the 8 cars after the intersection and it does not daisy to another register.

We must also note that each of the ‘road’ LEDs is connected in series with a 660 Ω resistor to ensure that the light is not too intense to not burn out the LEDs. Another small yet critical component of this wiring is that the two data ports of each shift register must be bridged to allow the data to travel to all 8 of the output pins of the register.

Software

System Overview

The system uses two FreeRTOS tasks to accomplish the objective of the project. The Stop Light task controls the three traffic lights and their intervals. The Traffic Generation task controls car generation and display. These tasks communicate using a single queue, which holds the current value of the stop light. This arrangement is described in the diagram below.

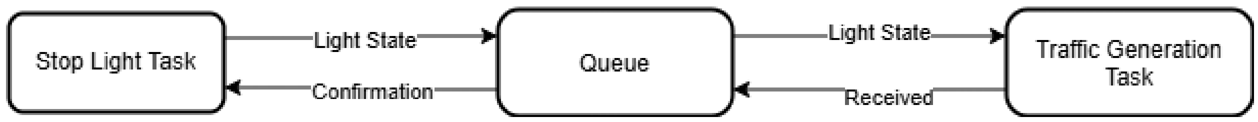

Figure 1: System Overview Diagram

Traffic Flow

The traffic flow value is based directly on the value from the potentiometer. This value is acquired by calling ADC_Convert(), which falls between 0Ω and 4095Ω but is incremented to between 1Ω and 4096Ω to prevent divide-by-zero errors. Using this value, Get_Flow() is called. This function the following linear interpolation formula:

$$\text{flow\_value} = \text{low\_flow\_value} + \frac{(\text{observed\_pot\_value} - \text{low\_pot\_value}) \cdot (\text{high\_flow\_value} - \text{low\_flow\_value})}{\text{high\_pot\_value} - \text{low\_pot\_value}}$$

Figure 2: Flow Value Linear Interpolation Formula

Where:
- Flow_value: desired value, between 1 and 10
- Low_flow_value = 1
- High_flow_value = 10
- Low_pot_value = 1Ω
- High_pot_value = 4096Ω
- Observed_pot_value = Pot value from ADC

The range of 1Ω and 4096Ω is converted to a range of 1 to 10 to represent traffic flow.

Traffic Generation

Using the traffic flow value we just derived, traffic can now be generated. Generally speaking, the following process for traffic generation occurs (ignoring the current light state). A random value between 1 and 10 is generated. If that randomly generated value is less than or equal to the current traffic flow value, a car is generated.

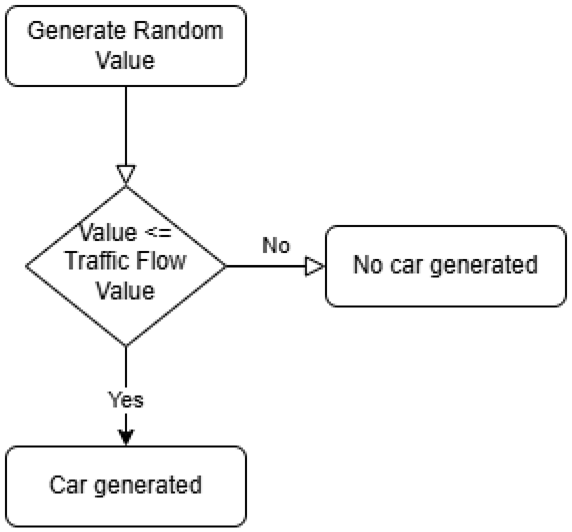

Figure 3: Traffic Generation Algorithm

Traffic Light

At the beginning of every traffic light cycle, the traffic light delay value is determined. The traffic light delay is based directly on the value from the potentiometer. This value is acquired by calling ADC_Convert(), which falls between 0Ω and 4095Ω but is incremented to between 1Ω and 4096Ω to prevent divide-by-zero errors. Using this value, Get_Delay() is called. This function the following linear interpolation formula:

$$\text{delay\_value} = \text{low\_delay\_value} + \frac{(\text{ob\_pot\_value} - \text{low\_pot\_value}) \cdot (\text{high\_delay\_value} - \text{low\_delay\_value})}{\text{high\_pot\_value} - \text{low\_pot\_value}}$$

Figure 4: Delay Value Linear Interpolation Formula

Where:

- Delay_value: desired value, between 4s and 10s
- Low_delay_value = 4s
- High_delay_value = 10s
- Low_pot_value = 1Ω
- High_pot_value = 4096Ω
- Observed_pot_value = Pot value from ADC

The range of 1Ω and 4096Ω is converted to a range of 4s to 10s to create an accurate time interval for the traffic lights. The green light delay is directly derived from the traffic delay value, the yellow delay is set to a constant value of 3s [2], and the red light delay is inverted by taking the sum of the lower and upper bound and subtracting the current traffic delay value. This creates a system where the green light is directly proportional to the traffic delay value, and to double the red light value at maximum flow setting. It also allows the red light to be inversely proportional to the traffic delay value, and to double the green light value at minimum flow setting. 4s was chosen as the low traffic delay value because it represents a reasonable lower bound for traffic light time (1s for instance is very unreasonable). The light colour is then set to green, and the system queue is notified of this change. The light length is then achieved by delaying the task using vTaskDelay() by the traffic delay value. The light colour is then set to yellow, and the system queue is notified that the light colour is red (because yellow and red lights are treated the same before the stop line). The light length is then achieved by delaying the task using vTaskDelay() by the constant 3s. The light colour is then set to red. The light length is then achieved by delaying the task using vTaskDelay() by the inverted traffic delay value. This cycle is repeated.

System Display

To display the traffic lights, Set_Colour() is called, where the appropriate pin is set to high, and the other two are set to low.

To display the cars, a much more advanced algorithm is used.

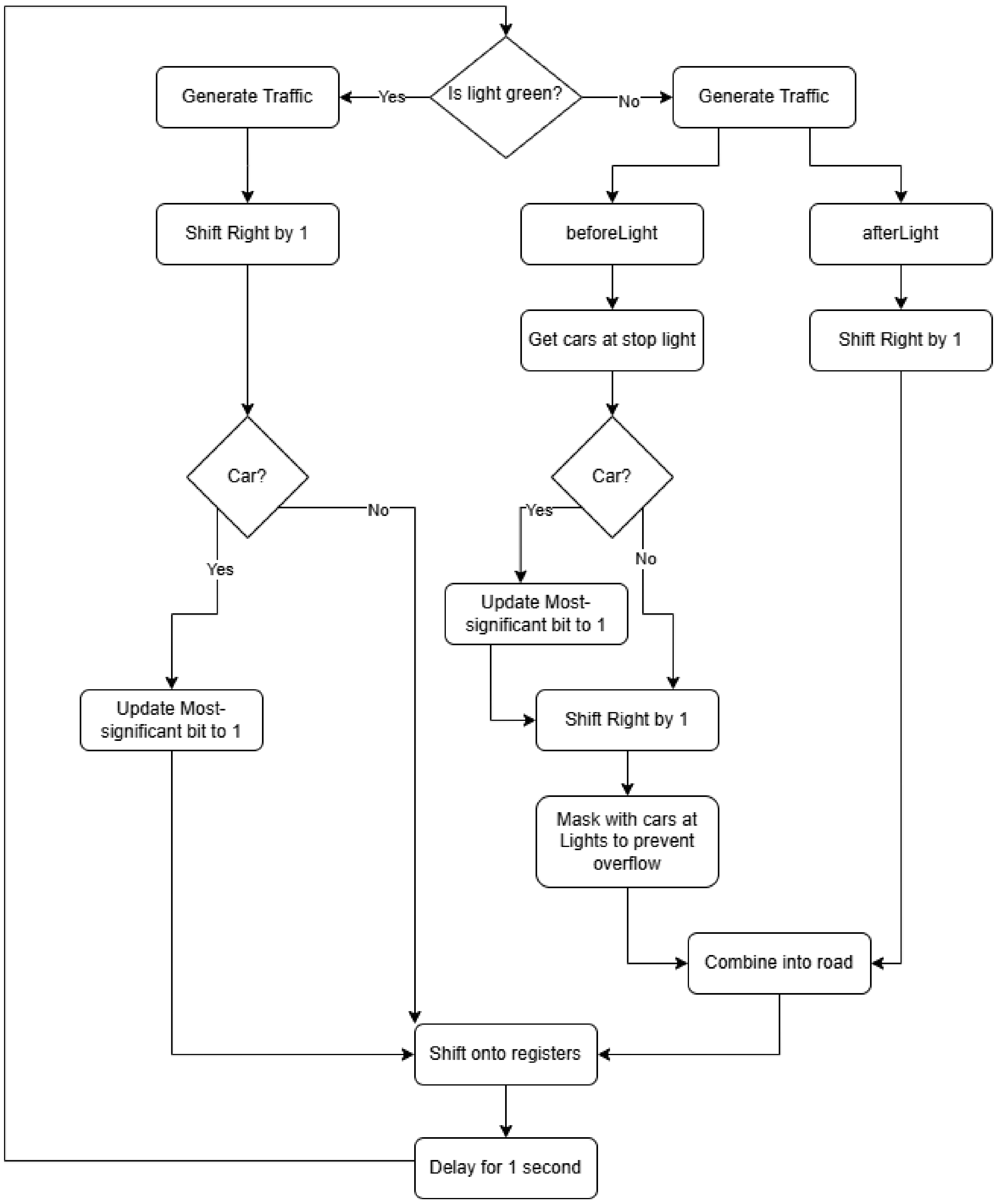

Figure 5: System Display algorithm

The road is represented by a 19-bit number, where each bit represents a car position, 1 represents the presence of a car, and 0 represents the lack of a car. At the beginning of the cycle, the queue is checked to see if there is an update to the light colour. If there is an update, the colour variable is updated to reflect that, where 0 is green and 1 is red. Regardless of the colour, the traffic generation routine runs. If the light is green, the road is bit-shifted to the right by one. If there is a car generated, the most significant bit is updated to a 1. If the light is red, the road is split in two: beforeLight and afterLight. In beforeLight, the cars at the stop light are detected. Then beforeLight is shifted right, and masked by the cars at the stop light to prevent overflow. AfterLight is simply shifted right. These two roads are then combined into the larger road. Regardless of light colour, the road is then shifted onto the registers, and the task is delayed by 1 second, and then repeated. This algorithm accurately moves cars at 1 car per second and stops them correctly given their location when the light turns red.

Discussion

Implementing the hardware/ wiring for this system came with a large learning curve. Since many of the components used on the breadboard and even the breadboard itself were largely unfamiliar at the beginning of this project, many iterations of a wiring schematic were needed before a viable solution was found. Fortunately, the students used breadboarding software to prototype a solution before wiring it on the physical breadboard. This saved an immense amount of time and effort as any backtracking in the process required a few clicks of a mouse rather than unplugging, rearranging, and wire stripping an entire circuit. The most difficult component of the wiring was learning how to daisy chain the shift registers together. Initially, it was assumed by the students that the H port could not be connected to an LED as it was being used to output information to the next register in the daisy chain. This caused the students to plug only 7 LEDs into the first shift register and the data to transfer improperly to the intersection of the road as we were essentially skipping a bit on the road. Thankfully, Brent, the lab technician, pointed out that although the H port is used to daisy chain to the next shift register, an LED should still be connected to it. During this conversation, Brent also pointed out to the students that the data ports of each shift register should be connected. After implementing these two fixes, the hardware started behaving as expected.

There were several difficulties associated with the software of this project. Firstly, the students struggled to initialize the GPIO and ADC modules, because the sample code wasn't out yet. They were, however, able to solve this problem using their own experience as well as the help of the lab technician, Brent. This initialization led to the largest problem of the project though, as LEDs would intermittently flash on and off at seemingly random times. It turns out that for the shift registers properly, the GPIO ports must be set to a 2 Mhz clock, and the students had set them to 50 Mhz. Finally, the students struggled to effectively split the road into two parts, do operations on each of them, and then correctly stitch them back together using proper bit manipulation. With enough trial and error, however, this too was resolved.

Appendices

Design Document

1. Introduction

Overview:

The goal of this project is to build an embedded system in C on an STM32F4 breakout board that simulates a traffic light on a road using FreeRTOS. A link to this simulation can be found in Appendix A [1].

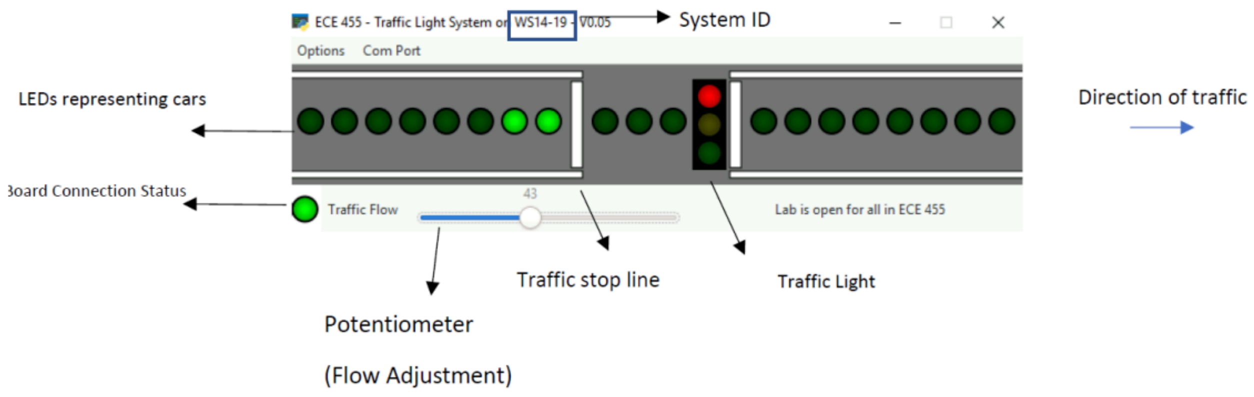

Figure 1: Traffic Light Simulation [2]

---

2. System Requirements

Functional Requirements

FR-001: Stop Light

- **Requirement ID**: FR-001
- **Title**: Stop Light
- **Description**: A red, yellow, and green LED flash like a traffic light.
- **Input**:
  - Traffic flow
- **Processing/Logic**: Every time the light changes from red to yellow to green or vice versa, the system calculates a new light length. This light length is proportional to the potentiometer resistance for the green light, inversely proportional to the potentiometer resistance for the red light, and constant for the yellow light.
- **Output**:

    - Red, green, and yellow lights are of an appropriate length.
- **Error Handling**:
    - If no traffic flow is detected, the system should halt.
- **Assumptions**:
    - The task can properly communicate with the Traffic Flow task.
- **Dependencies**: FR-003
- **Constraints**:
    - The lights need to switch between each other immediately.
    - The green light needs to be approximately twice as long as the red light.
- **Acceptance Criteria**:
    - The red light stays on for a time inversely proportional to the potentiometer resistance.
    - The green light stays on for a time proportional to the potentiometer resistance.
    - The yellow light stays on for a constant time.
    - The lights change properly after the correct time has elapsed.

FR-002: Traffic Generation

- **Requirement ID**: FR-002
- **Title**: Traffic Generation
- **Description**: New car traffic is generated.
- **Input**:
    - Traffic flow
- **Processing/Logic**: New traffic is generated randomly at a rate proportional to the resistance of the potentiometer.
- **Output**:
    - Traffic nodes are sent to the system display task.
- **Error Handling**:
    - If no traffic is detected, the system should halt.
- **Assumptions**:
    - The task can properly communicate with the Traffic Flow task.
- **Dependencies**: FR-003
- **Constraints**:
    - When traffic flow is at a minimum, there should be a gap of 5-6 LEDs between cars.
    - When traffic flow is at a maximum, there should be no LED gap between cars.
- **Acceptance Criteria**:
    - Traffic is generated at a rate proportional to the potentiometer resistance.

FR-003: Traffic Flow

- **Requirement ID**: FR-003
- **Title**: Traffic Flow
- **Description**: Traffic flow can be controlled from the potentiometer
- **Input**:
    - Resistance from the potentiometer

- **Processing/Logic**: Resistance from the potentiometer determines the traffic flow on a scale from 0 - 1001.
- **Output**:
    - Traffic Flow value
- **Error Handling**:
    - If no resistance is detected, the system should halt.
- **Assumptions**:
    - The potentiometer is calibrated and working correctly.
- **Dependencies**: N/A
- **Constraints**: N/A
- **Acceptance Criteria**:
    - Traffic is generated at a rate proportional to the potentiometer resistance.

FR-004: System Display

- **Requirement ID**: FR-004
- **Title**: System Display
- **Description**: LEDs properly display traffic flow.
- **Input**:
    - Traffic Flow
    - Light state
- **Processing/Logic**: Traffic progresses along LEDs based on Traffic Flow value. Traffic stops when there is a yellow or red light.
- **Output**:
    - Green LEDs (cars)
- **Error Handling**:
    - If no traffic light value is detected, the system halts.
    - If no traffic flow value is detected, the system halts.
- **Assumptions**:
    - The potentiometer is calibrated and working correctly.
- **Dependencies**: FR-003, FR-001
- **Constraints**: LEDs must move between 1-3 LEDs per second.
- **Acceptance Criteria**:
    - Traffic is generated at a rate proportional to the potentiometer resistance.

Hardware Requirements:

- STM32F4 Evaluation Board
- Breadboard
- 19 Green LEDs for cars
- 1 green, 1 yellow, 1 red LED for traffic lights
- 1 5000 Ohm potentiometer
- 19 620 Ohm resistors for car LEDs
- 1 3300 Ohm resistor for potentiometer
- 3 300 Ohm resistors for traffic light LEDs

- 3 74HC164 shift registers
- Wire

Software Requirements:

- FreeRTOS
- STM32F4 Drivers

---

3. System Design

Hardware Design:

Circuit

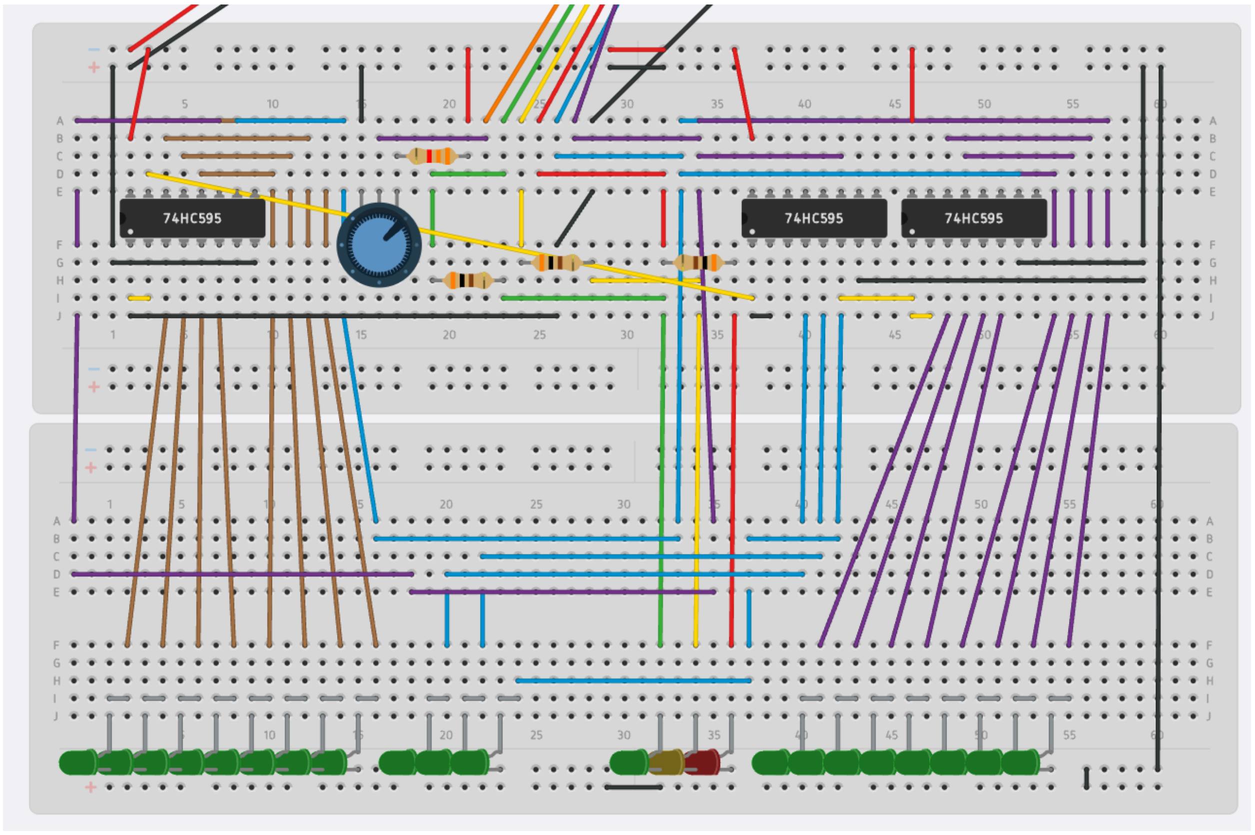

Figure 2: Breadboard Diagram

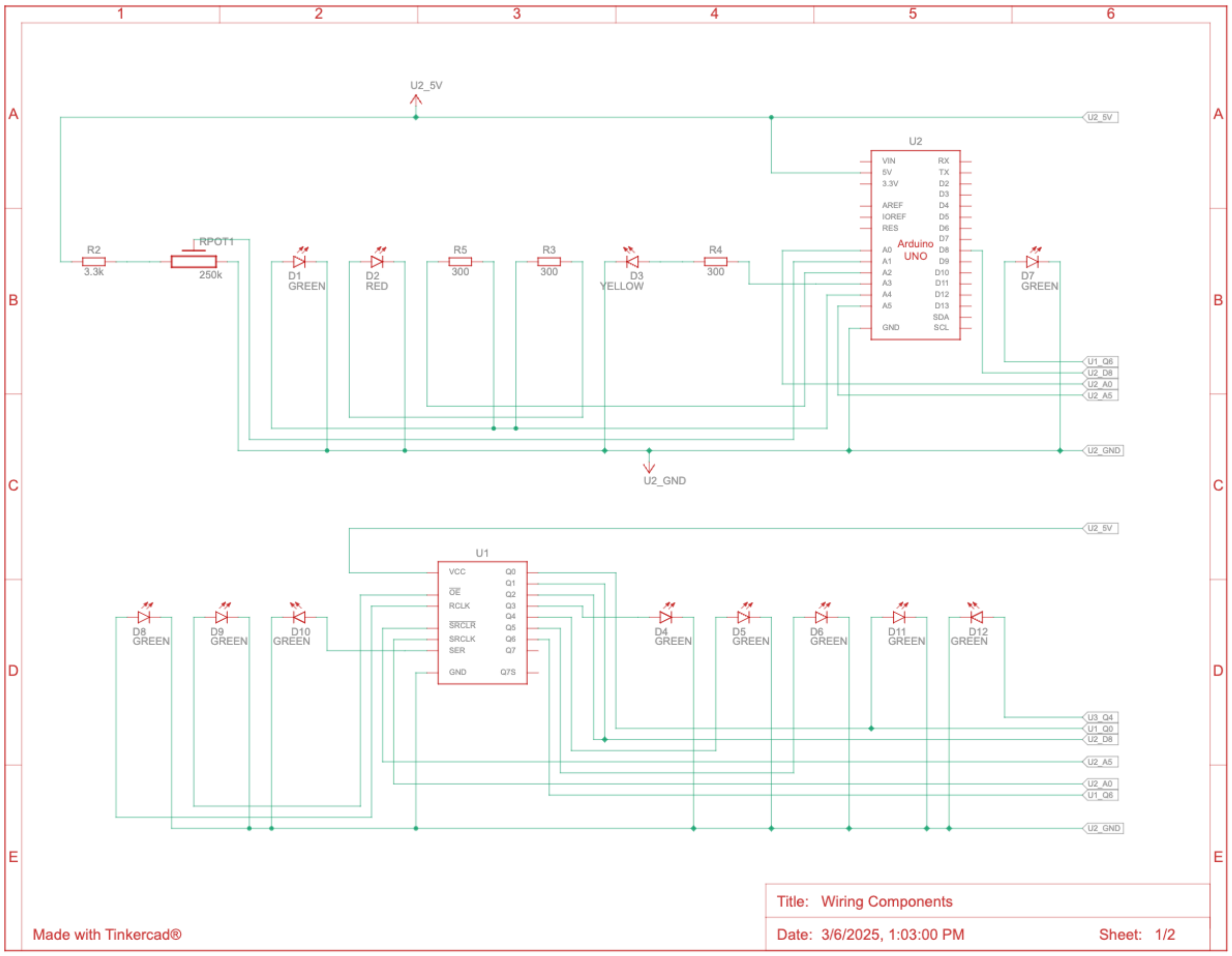

Figure 3: Breadboard Schematic 1

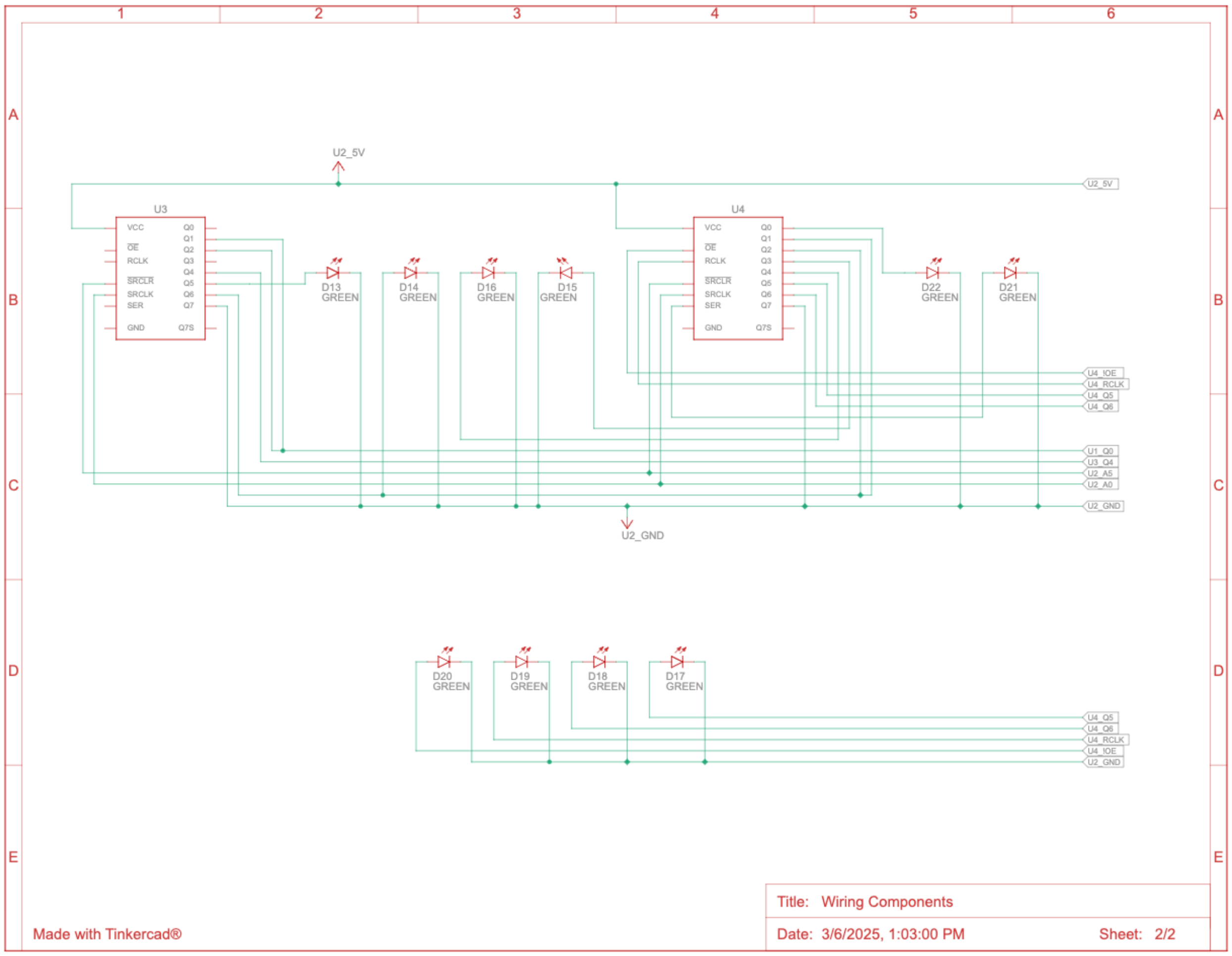

Figure 4: Breadboard Schematic 2

GPIO Ports

Port	Purpose
PC0	Red Light
PC1	Yellow Light
PC2	Green Light (Traffic)
PC3	Potentiometer Input
PC4	SR1 Clear
PC5	SR1 Clock
PC6	SR1 Data

PC7	SR2 Reset
PC8	SR2 Clock
PC9	SR2 Data
PC10	SR3 Reset
PC11	SR3 Clock
PC12	SR3 Data

Table 1: GPIO Port Mappings

LEDs

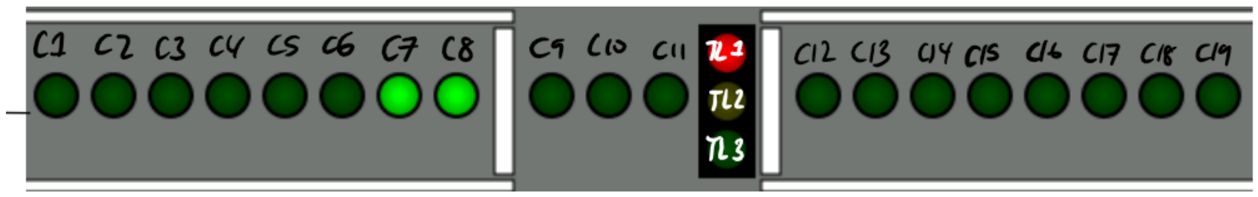

Figure 3: Car and Traffic Light LED names

Shift Registers

Shift Register 1:

A	B	C	D	E	F	G	H
C1	C2	C3	C4	C5	C6	C7	C8

Table 2: SR1 Mapping

Shift Register 2:

A	B	C	D	E	F	G	H
C9	C10	C11	TL1	TL2	TL3	C12	C13

Table 3: SR2 Mapping

Shift Register 3:

A	B	C	D	E	F	G	H
C14	C15	C16	C17	C18	C19	-	-

Table 4: SR3 Mapping

Software Design:

Initialization

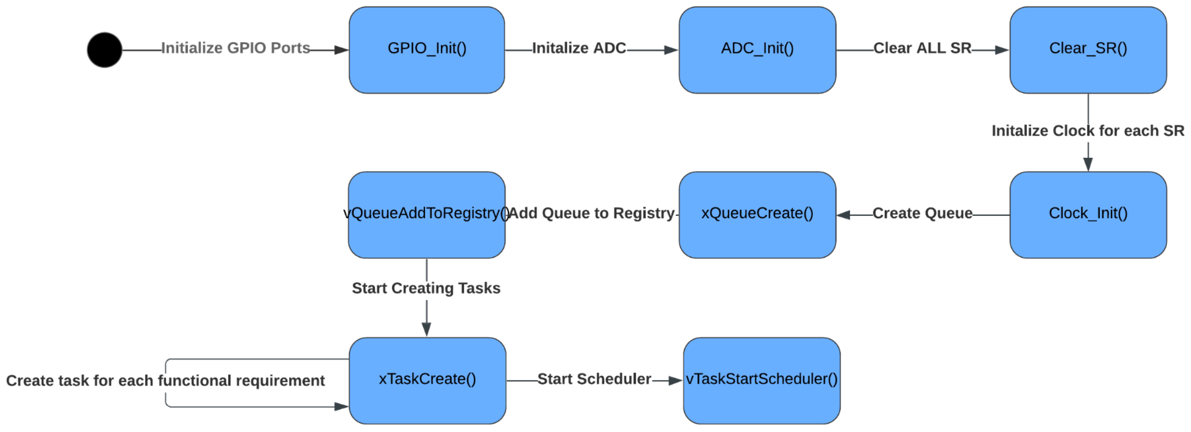

Figure 4: Initialization Sequence

Traffic Flow Task

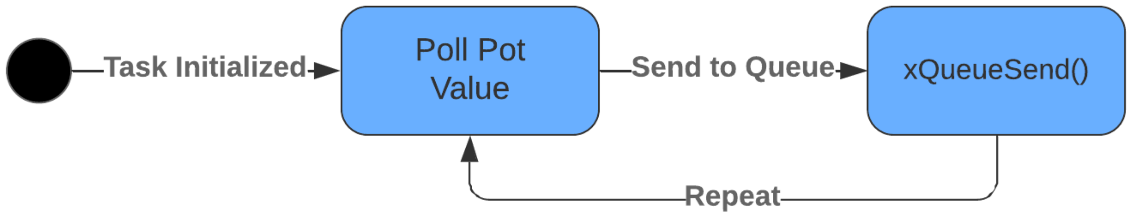

Figure 5: Traffic Flow Diagram

Stop Light Task

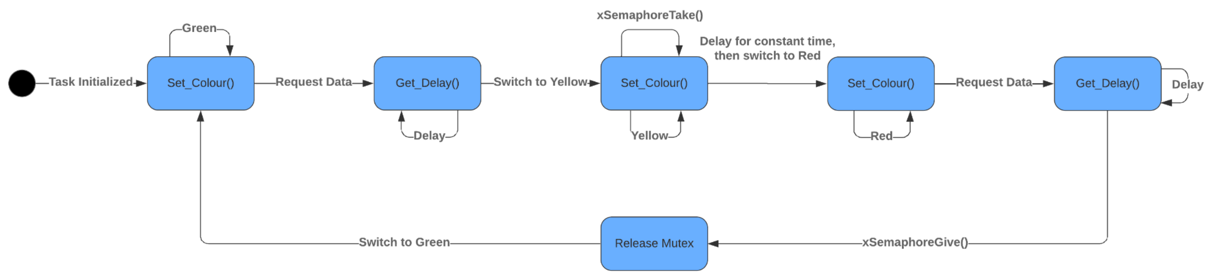

Figure 6: Stop Light Diagram

Traffic Generation Task

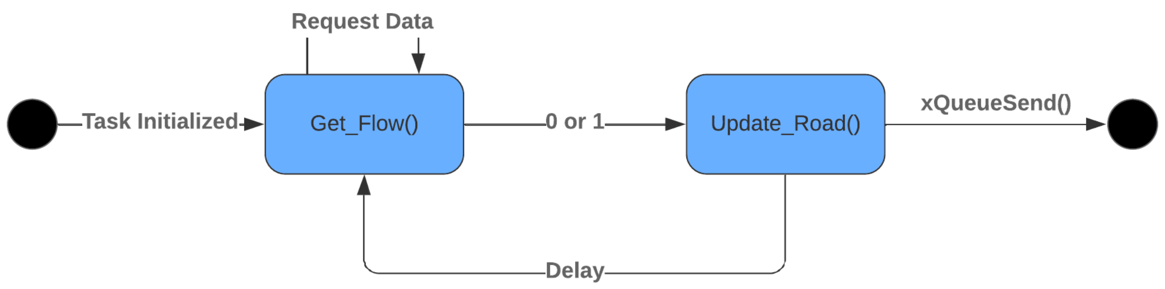

Figure 7: Traffic Generation Diagram

System Display Task

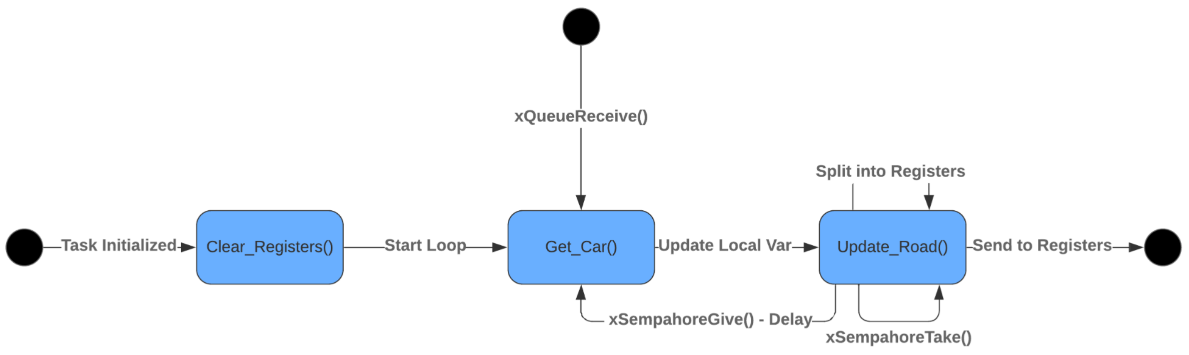

Figure 8: Display Diagram

4. System Integration

Integration Plan

Circuit

1. Design circuit on Tinkercad breadboard diagramming software.
2. Debug using the aforementioned software.
3. Physically wire circuit on a breadboard.

Software

Build tasks in order laid out by diagrams in section 3, and build functions in each task by sequence in diagrams.

Testing Strategy

Test each function individually, and once the functions for a task are complete, integrate them and test them as a whole. Once all tasks are complete, integrate each task sequentially.

5. Appendices

Appendix A: Links

Traffic Light Simulation

Appendix B: References

[1] University of Victoria, "ECE 455: Real Time Computer Systems Design Project," Nov. 2019.

[2] P. Shiri, "ECE455 Project 1 - Traffic Light System," University of Victoria, Feb. 2021.

[3] National Association of City Transportation Officials, "Signal Cycle Lengths," *National Association of City Transportation Officials*. https://nacto.org/publication/urban-street-design-guide/intersection-design-elements/traffic-signals/signal-cycle-lengths/

Code Listing

Main.c

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"
/* Kernel includes. */
#include "stm32f4xx.h"
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/semphr.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"
/* Library Include */
#include "../Libraries/include/Initialization.h"
#include "../Libraries/include/StopLight.h"
#include "../Libraries/include/TrafficGeneration.h"
#include "../Libraries/include/RTOS.h"

/*-----------------------------------------------------------*/

int main(void)
{

	SystemInit();
	SystemCoreClockUpdate();

	GPIOC_Init();
	ADC1_Init();
	Clear_SR();
	Start_RTOS();

	return 0;
}
```

Initialization.c

```
#include "../Libraries/include/Initialization.h"

#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"
#include "stm32f4xx.h"

#define mainQUEUE_LENGTH 100

void GPIOC_Init(){

    GPIO_InitTypeDef GPIO_InitStruct;

    // Enable GPIOC clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    // Configure GPIOC pins as output (except Pin 3)
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0 |
                    GPIO_Pin_1 |
                            GPIO_Pin_2 |
                            GPIO_Pin_4 |
                            GPIO_Pin_5 |
                            GPIO_Pin_6;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_InitStruct);

   // Initialize Pin 3 as Input
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void ADC1_Init(){
	// Enable ADC Clock
	RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

	ADC_InitTypeDef ADC_InitStruct;
	ADC_StructInit(&ADC_InitStruct);

	// Configure ADC1
	ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
	ADC_InitStruct.ADC_ScanConvMode = DISABLE;
	ADC_InitStruct.ADC_ContinuousConvMode = ENABLE;
   ADC_InitStruct.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1; // Trigger by
TIM1 Update event
   ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right; // Right alignment of data
   ADC_InitStruct.ADC_NbrOfConversion = 1;  // Single conversion (for one channel)


	ADC_Init(ADC1, &ADC_InitStruct);
	ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1,
ADC_SampleTime_3Cycles);
	ADC_Cmd(ADC1, ENABLE);
   ADC_SoftwareStartConv(ADC1);

}

void Clear_SR(){
	GPIO_ResetBits(GPIOC, GPIO_Pin_4); // Set CLR to low
	GPIO_SetBits(GPIOC, GPIO_Pin_4); // Set CLR to high;
}
```

StopLight.c

```
#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"
#include "stm32f4xx.h"
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/semphr.h"
```

```
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"

#define green          0
#define yellow  1
#define red     2

void Set_Colour(int colour){
        if(colour == green){ // set green pin clear everything else
                GPIO_ResetBits(GPIOC, GPIO_Pin_0);
                GPIO_ResetBits(GPIOC, GPIO_Pin_1);
                GPIO_SetBits(GPIOC, GPIO_Pin_2);
        } else if(colour == yellow){ // set yellow pin clear everything else
                GPIO_ResetBits(GPIOC, GPIO_Pin_0);
                GPIO_ResetBits(GPIOC, GPIO_Pin_2);
                GPIO_SetBits(GPIOC, GPIO_Pin_1);
        } else if(colour == red){ // set red pin clear everything else
                GPIO_ResetBits(GPIOC, GPIO_Pin_1);
                GPIO_ResetBits(GPIOC, GPIO_Pin_2);
                GPIO_SetBits(GPIOC, GPIO_Pin_0);
        }

}

int Get_Delay(int Traffic_Flow_Value){
        int low_pot = 1; // min pot value
        int high_pot = 4096; // max pot value
        int low_time = 4; // min light time (4 seconds)
        int high_time = 10; // max light time (10 seconds)

        return (low_time + (Traffic_Flow_Value - low_pot) * (high_time - low_time) /
(high_pot - low_pot)) * 1000; // get total delay time in milliseconds
}
```

TrafficGeneration.c

```
#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"
```

```c
#include "stm32f4xx.h"

int Get_Flow(int Traffic_Flow_Value){
	int low_pot = 1; // min pot value
	int high_pot = 4096; // max pot value
	int low_flow = 1; // min flow
	int high_flow = 10; // max flow

	return (low_flow + (Traffic_Flow_Value - low_pot) * (high_flow - low_flow) /
(high_pot - low_pot));
}

uint32_t Get_LSB_Mask(uint32_t beforeLights){
	beforeLights >>= 11;
	uint16_t mask = 0;
	while(beforeLights & 1){
		mask <<= 1;
		mask |= 1;
		beforeLights >>= 1;
	}
	return mask;
}
```

RTOS.c

```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "stm32f4_discovery.h"
#include "stm32f4xx.h"
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/semphr.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"
#include "../Libraries/include/RTOS.h"
#include "../Libraries/include/StopLight.h"
#include "../Libraries/include/TrafficGeneration.h"
```

```
#include "../Libraries/include/Initialization.h"

#define mainQUEUE_LENGTH 100
#define maxTicks 13000
#define roadLength 19
#define preLightsLength 8

void Start_RTOS();

void Stop_Light_Task(void *pvParameters);

void Traffic_Generation_Task(void *pvParameters);

xQueueHandle xQueue_handle = 0;

void Start_RTOS(){
	xQueue_handle = xQueueCreate(	mainQUEUE_LENGTH,		/* The 
number of items the queue can hold. */
							sizeof( uint16_t ) );	/* The size of each 
item the queue holds. */

	/* Add to the registry, for the benefit of kernel aware debugging. */
	vQueueAddToRegistry( xQueue_handle, "MainQueue" );

	xTaskCreate( Stop_Light_Task, "Stop_Light", configMINIMAL_STACK_SIZE, 
NULL, 1, NULL);
	xTaskCreate( Traffic_Generation_Task, "Traffic_Generation_Task", 
configMINIMAL_STACK_SIZE, NULL, 1, NULL);

	vTaskStartScheduler();
}

void Stop_Light_Task(void *pvParameters){

	Set_Colour(0); // stop light starts green

	int delay = 0;

	uint16_t tx_data = 0;
```

```
        while(1){
                delay = Get_Delay(ADC_GetConversionValue(ADC1)+1); // get pot value
                Set_Colour(0);
                if(xQueueSend(xQueue_handle, &tx_data, 1000)){ // tell the road that the light
is turning red
                        tx_data = 2;
                }
                vTaskDelay(delay); // delay for time proportional to pot
                Set_Colour(1);
                if(xQueueSend(xQueue_handle, &tx_data, 1000)){ // tell the road that the light
is turning green
                        tx_data = 0;
                }
                vTaskDelay(3000); // 3 seconds
                Set_Colour(2);
                vTaskDelay(maxTicks - delay); // delay for time inversely proportional to pot
        }

}

void Traffic_Generation_Task(void *pvParameters){

        int minValue = 0;
        int maxValue = 10;
        int flow_value = 0;
        uint32_t road = 0x0;
        uint32_t beforeLight = 0x0;
        uint32_t afterLight = 0x0;
        uint16_t rx_data = 0;
        int colour = 0;

        srand(time(NULL));

        GPIO_SetBits(GPIOC, GPIO_Pin_4); // Set CLR to high

        while(1){ // red light shift
                if(xQueueReceive(xQueue_handle, &rx_data, 500)){ // check for new light
colour
                        if(rx_data == 0)
                                colour = 0;
```

```
            else
                colour = 1;
        }

        int rand_value = rand() % (maxValue - minValue + 1) + minValue;
        flow_value = Get_Flow(ADC_GetConversionValue(ADC1)+1);
        if(colour == 0){ // LIGHT IS GREEN
            if(rand_value <= flow_value){ // advance light as normal
                road = (road >> 1);
                road |= 0b1000000000000000000;
            } else{
                road = (road >> 1);
            }
        } else { // LIGHT IS RED
            beforeLight = road & 0b1111111100000000000; // get cars before
intersection
            uint32_t overflowMask = Get_LSB_Mask(beforeLight); // get the cars
at the stop line
            if(rand_value <= flow_value){ // generate traffic for cars before light
                beforeLight >>= 1;
                beforeLight |= 0b1000000000000000000;
            } else {
                beforeLight >>= 1;
            }
            beforeLight |= (overflowMask << 11); // ensure cars remain at light
            beforeLight &= 0b1111111100000000000; // ensure no cars have passed
light
            afterLight = road & 0b0000000011111111111; // get cars after light
            afterLight >>= 1; // advance as normal
            road = beforeLight | afterLight; // combine two sets of cars to create
road

        }
        Clear_SR();
        for(int i = 0; i<19; i++){ // push 19 bits (cars) of info to SR
       if (road & (1 << i)) {
        GPIO_SetBits(GPIOC, GPIO_Pin_6);
       } else {
        GPIO_ResetBits(GPIOC, GPIO_Pin_6);
       }
```

```
                GPIO_ResetBits(GPIOC, GPIO_Pin_5);
                GPIO_SetBits(GPIOC, GPIO_Pin_5);
                GPIO_ResetBits(GPIOC, GPIO_Pin_5);
                   }
                   vTaskDelay(1000); // 1 second

        }


}

void vApplicationMallocFailedHook( void )
{
        /* The malloc failed hook is enabled by setting
        configUSE_MALLOC_FAILED_HOOK to 1 in FreeRTOSConfig.h.

        Called if a call to pvPortMalloc() fails because there is insufficient
        free memory available in the FreeRTOS heap.  pvPortMalloc() is called
        internally by FreeRTOS API functions that create tasks, queues, software
        timers, and semaphores.  The size of the FreeRTOS heap is set by the
        configTOTAL_HEAP_SIZE configuration constant in FreeRTOSConfig.h. */
        for( ;; );
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook( xTaskHandle pxTask, signed char *pcTaskName )
{
        ( void ) pcTaskName;
        ( void ) pxTask;

        /* Run time stack overflow checking is performed if
        configconfigCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.  This hook
        function is called if a stack overflow is detected.  pxCurrentTCB can be
        inspected in the debugger if the task name passed into this function is
        corrupt. */
        for( ;; );
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook( void )
```

```
{
volatile size_t xFreeStackSpace;

        /* The idle task hook is enabled by setting configUSE_IDLE_HOOK to 1 in
        FreeRTOSConfig.h.

        This function is called on each cycle of the idle task.  In this case it
        does nothing useful, other than report the amount of FreeRTOS heap that
        remains unallocated. */
        xFreeStackSpace = xPortGetFreeHeapSize();

        if( xFreeStackSpace > 100 )
        {
                /* By now, the kernel has allocated everything it is going to, so
                if there is a lot of heap remaining unallocated then
                the value of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h can be
                reduced accordingly. */
        }
}
```

References

[1] University of Victoria, “ECE 455: Real Time Computer Systems Design Project,” Nov. 2019.

[2] P. Shiri, “ECE455 Project 1 - Traffic Light System,” University of Victoria, Feb. 2021.

[2] National Association of City Transportation Officials, “Signal Cycle Lengths,” *National Association of City Transportation Officials*. https://nacto.org/publication/urban-street-design-guide/intersection-design-elements/traffic-signals/signal-cycle-lengths/

Table of Figures

Figure Number	Description
1	System Overview Diagram

2	Flow Value Linear Interpolation Formula
3	Traffic Generation Algorithm
4	Delay Value Linear Interpolation Formula
5	System Display Algorithm