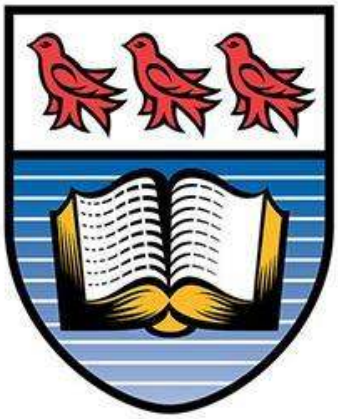
University of Victoria

Department of Electrical and Computer Engineering



ECE 455 Project 1: Traffic light system

Date of Submission: March 4th, 2026

Clifton Tollefson - V01004341
Khephren Gould - V01012827

Introduction

The objective of this project is to design and implement a functional Traffic Light System by utilizing the FreeRTOS real-time operating system on an STM32F4 Discovery Board. The system simulates a one-way, one-lane road intersection where vehicle traffic is represented by a series of LEDs. A core requirement of the design is the integration of hardware and software components, specifically using middleware to bridge the gap between application code and device drivers for the microcontroller's GPIO and ADC peripherals.

The simulation is driven by three primary interactive components:

- **Traffic Flow Adjustment:** A potentiometer is used to dynamically adjust the rate of car generation in real-time.
- **Vehicle Representation**: A chain of LEDs visualizes car positions, with vehicles "moving" across the display through coordinated LED toggling.
- **Traffic Light Control**: A three-LED traffic light (green, yellow, and red) manages the flow of traffic at a stop line.

The implementation leverages core FreeRTOS features, including tasks for modular execution, queues for inter-task communication, and software timers to manage the state transitions of the traffic lights. By satisfying these functional and technical requirements, the project demonstrates the integration of a structured software architecture with microcontroller peripherals to implement a responsive real-time application.

Design solution

The real time traffic flow simulator was implemented with a combination of hardware and software systems. The software was developed as a standard FreeRTOS application using the Attolic True studio development software. The hardware required for constructing the circuit, including equipment for assembling the circuit, was provided. A schematic for the circuit was developed using KiCAD and the circuit was subsequently assembled according to the schematic. A high level description of both the software and hardware systems is provided below.

Software System Overview

The software was organized into 4 processes consisting of traffic flow adjustment, traffic generation, traffic light state management, and a traffic state display task. The traffic generation and display processes were implemented as FreeRTOS tasks whereas the traffic flow adjustment and traffic light state management processes were implemented as timer callbacks.

Inter- task communication was implemented using three FreeRTOS queues. The current value for traffic flow is transmitted to both the traffic generator and traffic light state management processes using a queue of length one. At each timestep, the traffic flow adjustment task uses the xQueueOverwrite function of the queue API to overwrite the queue value with the calculated value for traffic flow. Both the traffic

generation and traffic light state tasks use xQueuePeek to read from the queue without discarding the value. The second queue holds a Boolean value representing the state of the car to be added to the road. This queue is overwritten by the traffic generator task and checked (peeked) by the system state display task at each time step. Finally, a queue is used for enabling transmission of the traffic light state from the traffic light management task to the system state display task. At each time step, the display task peeks this queue to obtain the state to display on the traffic light (red, green, amber).

A block diagram representing the logic described above is shown in Figure 1.

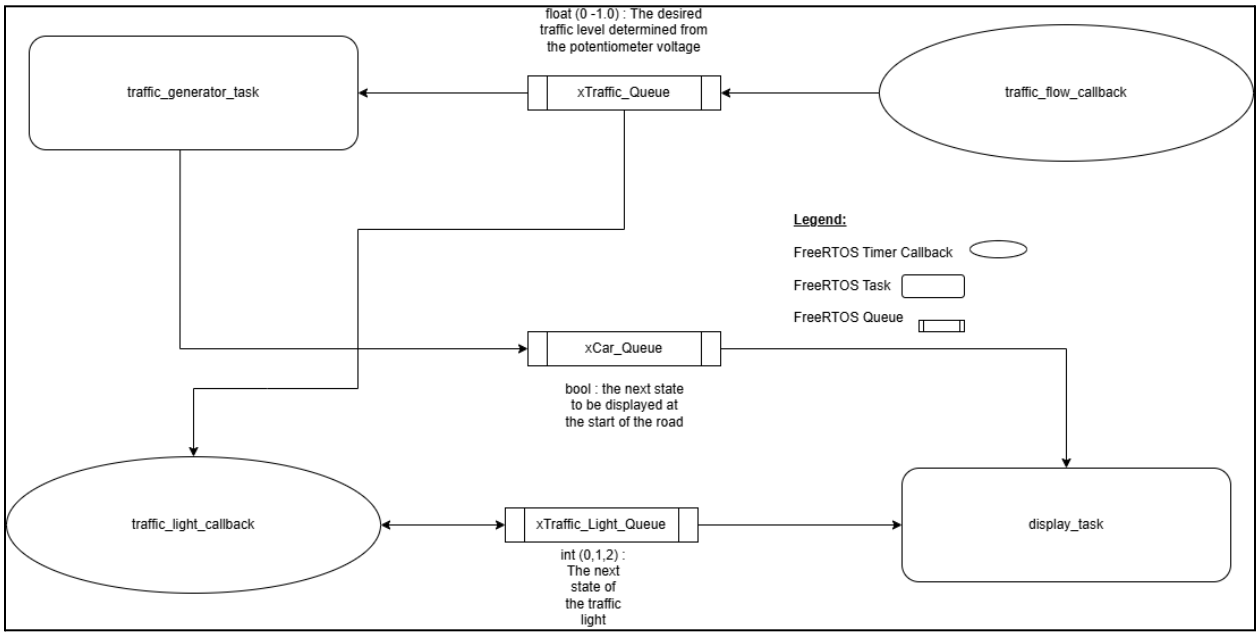


Figure 1: Software Design Block Diagram

Hardware Overview and Pinout

The hardware used to implement the traffic simulator is shown in Table 1.

Component	Quantity
STM32407 Discovery	1
74HC164 Shift Register	3
Green LED	20
Red LED	1
Amber LED	1
680 Ohm Resistor	19
300 Ohm Resistor	3

5k Potentiometer	1

Table 1: Real Time Traffic Simulator BOM

As shown in the table, the traffic state was displayed using 19 green LEDs connected to the outputs of three 74HC164 shift registers. The target microcontroller for the application was the STM32F407VGT6 on the STM32 Discovery development board. This microcontroller is based on the ARM Cortex-M4 processor. The circuit schematic for the traffic light system hardware is shown in Figure 3 and the physical breadboard circuit is shown in figure 2 below.

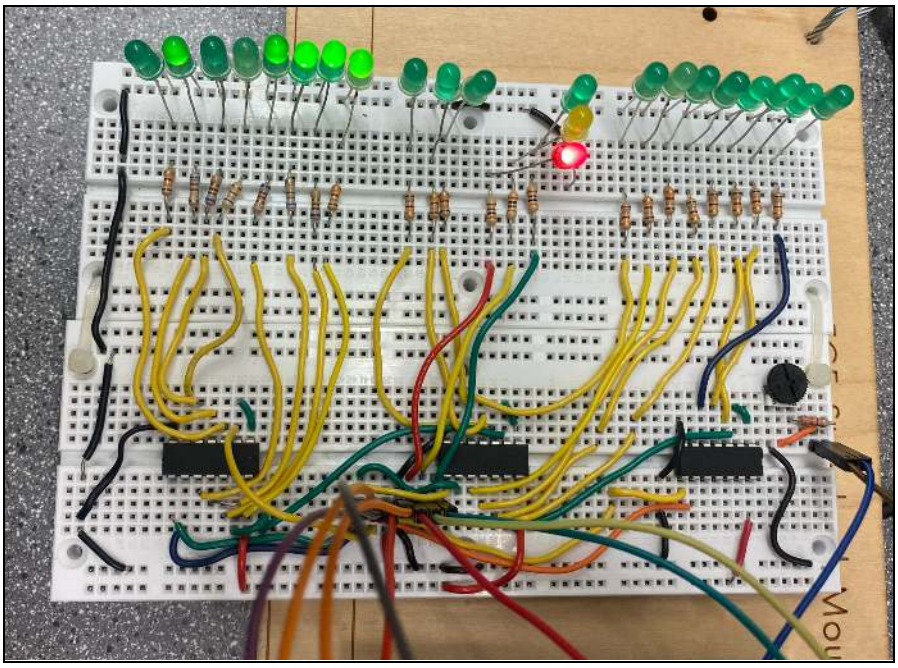


Figure 2: Real Time Traffic Simulator Breadboard Circuit.

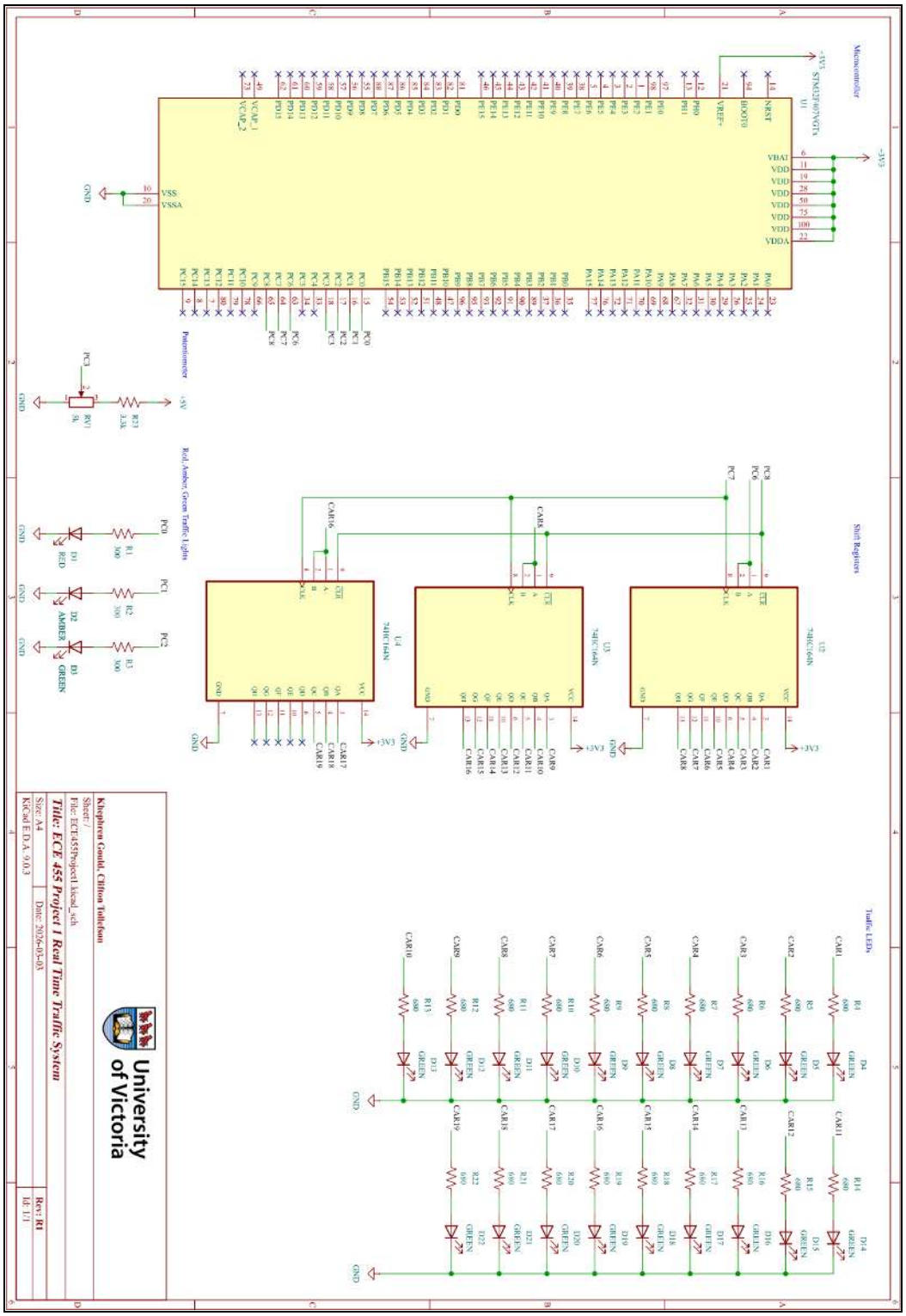


Figure 3: Real Time Traffic Circuit Schematic

The shift register circuit followed a daisy chain pattern in which the QH output of each register connects to the DSA and DSB inputs of the subsequent register. This enables a serial input to be clocked through all 24 outputs of the shift registers using a single GPIO to drive the DSA/B input of the first register in the chain. The clock and reset inputs of the registers are connected in parallel, requiring only two digital outputs from the microcontroller (in addition to the data output) to operate the registers. The logic voltage (GPIO output voltage) for the STM32407 is 3.3V. As listed on the datasheet for the STM32F407 Discovery board, Vref is set at 3.3V. This quantity is needed for determining the ADC measured voltage. The output pins utilized are listed in Table 2.

MCU Pin	Purpose
PC0	Red Traffic LightLED
PC1	Amber Traffic Light LED
PC2	Green Traffic Light LED
PC3	0-3V Potentiometer input
PC6	Shift Register Data
PC7	Shift Register Clock
PC8	Shift Register Reset

Table 2: Pinout Configuration for the STM32407

Initialization

Figure 4 shows the definitions used throughout the application software.

```
#define TRAFFIC_QUEUE_LENGTH (1)
#define DEFAULT_PRIORITY (configMAX_PRIORITIES -1)
#define ADC_RESOLUTION (12)
#define ADC_SAMPLING_PERIOD_MS (200)
#define CAR_QUEUE_LENGTH (1)
#define ROAD_LENGTH (20)
#define DEFAULT_GREEN_LIGHT_DURATION (3000)
#define YELLOW_LIGHT_DURATION (3000)
#define CAR_PROPAGATION_PERIOD (400)
```

Figure 4: Application Macro Definitions

Figure 5 shows the FreeRTOSConfig.h file used for the application.

```
extern uint32_t SystemCoreClock;

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                             1
#define configUSE_TICK_HOOK                             0
#define configCPU_CLOCK_HZ                              ( SystemCoreClock )
#define configTICK_RATE_HZ                              ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES                    ( 5 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 130 )
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 7 * 1024 ) )
#define configMAX_TASK_NAME_LEN                 ( 10 )
#define configUSE_TRACE_FACILITY                0
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_MUTEXES                               1
#define configQUEUE_REGISTRY_SIZE               8
#define configCHECK_FOR_STACK_OVERFLOW  2
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_MALLOC_FAILED_HOOK    1
#define configUSE_APPLICATION_TASK_TAG  0
#define configUSE_COUNTING_SEMAPHORES   1
#define configGENERATE_RUN_TIME_STATS   0

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Software timer definitions. */
#define configUSE_TIMERS                                1
#define configTIMER_TASK_PRIORITY               ( 3 )
#define configTIMER_QUEUE_LENGTH                5
#define configTIMER_TASK_STACK_DEPTH    ( configMINIMAL_STACK_SIZE * 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                             1
#define INCLUDE_vTaskCleanUpResources   1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                              1
```

Figure 5: FreeRTOSConfig.h

The entrypoint for the real time traffic simulator application is shown in Figure 6. The main function initializes all peripherals, and calls the FreeRTOS API functions for creating the tasks, timers and queues discussed in the system overview section.

```
int main(void)
{

	//hardware initialization
	init_trafficlight_gpio();
	init_adc();
	init_shift_register_gpio();
	//stm32 hardware random number generator
	init_rng();
	prvSetupHardware();

	//Create Queues
	xTraffic_Queue_handle = xQueueCreate(TRAFFIC_QUEUE_LENGTH, sizeof(float));
	xCar_Queue_handle = xQueueCreate(CAR_QUEUE_LENGTH, sizeof(bool));
	xTraffic_Light_Queue_handle = xQueueCreate(1, sizeof(int));
	vQueueAddToRegistry(xCar_Queue_handle,"Car Queue");
	vQueueAddToRegistry(xTraffic_Queue_handle, "Traffic Queue");
	vQueueAddToRegistry(xTraffic_Light_Queue_handle,"Traffic light Queue");

	//Create Tasks
	xTaskCreate(traffic_generator_task,"Traffic Generation", configMINIMAL_STACK_SIZE,NULL,DEFAULT_PRIORITY,NULL);
	xTaskCreate(display_task,"Display Task", configMINIMAL_STACK_SIZE,NULL,DEFAULT_PRIORITY,NULL);

	//Create Timers
	xFlow_Adjustment_Timer = xTimerCreate("Traffic Flow Adjustment",pdMS_TO_TICKS(ADC_SAMPLING_PERIOD_MS),
pdTRUE,NULL,traffic_flow_callback);
	xTraffic_Light_Timer = xTimerCreate("Traffic Light timer",pdMS_TO_TICKS(DEFAULT_GREEN_LIGHT_DURATION),
pdTRUE,NULL,traffic_light_callback);
	if(xFlow_Adjustment_Timer != NULL && xTraffic_Light_Timer != NULL){
		BaseType_t FlowTimerStarted = xTimerStart(xFlow_Adjustment_Timer,0);
		BaseType_t LightTimerStarted = xTimerStart(xTraffic_Light_Timer,0);
		if(FlowTimerStarted == pdPASS && LightTimerStarted == pdPASS){
			//start scheduler if timers created successfully
			vTaskStartScheduler();

		}
	}

	for(;;){
		printf("uh oh, vTaskStartScheduler returned :(");
	}

	return 0;
}
```

Figure 6: main program listing

The two tasks utilized were configured with a priority of 4 (one below the configured maximum priority as shown in Figure 4). This is because of the properties of the default scheduler employed by FreeRTOS. With configUsePreemption set to 1, the scheduler is a pre-emptive scheduler with round robin time slicing of equal priority tasks. As all the tasks for the application can be characterized as periodic, the application is not required to handle the arrival of unexpected high-priority asynchronous tasks (which would require unequal task priorities). When paired with a round-robin scheduler, the use of equal task priorities (and appropriate blocking times as discussed below) ensures no task is starved and allows for simple debugging.

As shown in Figure 6, the traffic_flow_adjustment timer was configured with a period of 200ms. This causes the callback to run approximately twice between each execution of the system display task, ensuring the system appears responsive to the traffic flow set by the user.

The traffic light timer is initialized with a period of 3 seconds. This results in a green light with 3 second duration as the first traffic light state. The timer period is subsequently adjusted based on the traffic flow level within the timer callback itself.

The software used to initialize all digital input GPIO pins is shown in Figure 7.

```
void init_trafficlight_gpio(void){
	GPIO_InitTypeDef  GPIO_InitStructure;
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
	//traffic lights on PC0 through PC 2
	GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
	GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
	GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
	GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
	GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
	GPIO_Init(GPIOC, &GPIO_InitStructure);
	GPIO_ResetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2);
}
```

Figure 7: PC0 through PC2 Pin Initialization

The pins are configured as outputs with “push-pull” as the output type. The internal pull up resistor is activated to prevent the pin voltage from floating when not driven.

```
void init_shift_register_gpio(void){
	GPIO_InitTypeDef  GPIO_InitStructure;
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
	// Shift Register Pins on Pin 6 through Pin 8 On GPIO Peripheral C
	GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
	GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
	GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
	GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
	GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
	GPIO_Init(GPIOC, &GPIO_InitStructure);
	GPIO_ResetBits(GPIOC, GPIO_Pin_6 | GPIO_Pin_7);
	GPIO_SetBits(GPIOC, GPIO_Pin_8);

}
```

Figure 8: PC6 through PC8 Pin Initialization

The shift register GPIO pin configuration is identical to the traffic light pin configuration. Pin PC8 is set to high by default as it is connected to the reset input of the shift registers. Initializing the pin as “High” ensures reset is inactive by default.

```
void init_adc(void){
	ADC_InitTypeDef ADC_Init_Structure;
	GPIO_InitTypeDef  GPIO_InitStructure;
	//PC3 used as ADC input Pin
	GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;

	//Configure PC3 as an analog input
	GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
	GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
	GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
	GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
	//STM32407 Driver function for initialization of ADC with suitable defaults
	//continuous conversion is disabled by this function
	ADC_StructInit(&ADC_Init_Structure);
	RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);


	GPIO_Init(GPIOC, &GPIO_InitStructure);
	ADC_Init(ADC1, &ADC_Init_Structure);
	ADC_Cmd(ADC1, ENABLE);
	//ADC1 channel 13 is used.
	ADC_RegularChannelConfig(ADC1 ,ADC_Channel_13,1,ADC_SampleTime_3Cycles);
}
```

Figure 9: PC 3 and ADC Peripheral Initialization

The ADC initialization requires initializing both the GPIO peripheral and the ADC peripheral for the desired analog input pin. The PC3 GPIO pin mode is set to analog and the internal pull up and pull down resistors are disabled. For the ADC peripheral, an STM32 driver function (ADC_Struct_Init) was used to apply a suitable default configuration. This function disables continuous conversion, configures the ADC resolution as 12 bits and disables external conversion triggering. Finally, ADC channel configuration was applied. Channel 13 of ADC1 was selected and a sampling time of 3 cycles was selected. These configurations were applied based on Table 7 (page 49) of the MCU reference manual. Although shorter sampling times reduce the accuracy of the readings, the error introduced was not sufficient enough as to prevent the traffic flow from being approximately directly proportional to the ADC reading.

```
void init_rng(void){
	RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);
	RNG_Cmd(ENABLE);
}
```

Figure 10: RGN Peripheral Initialization

The RNG peripheral was utilized for randomizing the generation of cars within the traffic_flow_generator task. Initialization is straightforward and consists solely of enabling the clock for the peripheral (the RNG peripheral is on the AHB2 clock bus) and enabling the peripheral. The RNG peripheral is discussed in section 24 of the STM32F407 reference manual [3].

Traffic Flow adjustment task

To read the traffic value set on the potentiometer from the user, we employ an ADC (analog to digital converter). This ADC channel is polled every 200 milliseconds (0.2 seconds). To accomplish this, the task is written as a callback function, a FreeRTOS feature in which the function gets called at each instant that the timer expires. This is effectively a periodic task with a period of 200 milliseconds.

```
//function to sample ADC, converts adc value to traffic flow rate, write value to queue
static void traffic_flow_callback(TimerHandle_t xTimer){
        float adc_code;
        float probability;
        ADC_SoftwareStartConv(ADC1);
        while (ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC)!= SET ){}
        adc_code = ADC_GetConversionValue(ADC1);
        probability = adc_code / pow(2,ADC_RESOLUTION);
        xQueueOverwrite(xTraffic_Queue_handle, &probability);
}
```

Figure 11: Traffic Flow Callback Function

The task itself is quite simple, ADC1 is started using the command "ADC_SofwareStartConv(ADC1)". Once completed, ADC_FLAG_EOC is set automatically to signify its completion. Now that the conversion is complete, the code is stored in the float "adc_code" and is a value between $0\text{-}2^{12}$. We can now convert this code into a probability by normalizing it to a value between 0 and 1, simply by dividing the adc_code by 2^{12}. This probability is the traffic level, or probability that a car will enter the road. To access this value in other tasks we write to the traffic queue using the "xQueueOverwrite()" function which replaces the value in the queue with the current traffic level.

Traffic Generator Task

To represent a car we chose to use a bool since a space on the road either has a car present or not. Cars should be generated at a rate proportional to the traffic level, and so a high level has a high probability and a low level is a low probability of a car being generated.

A Bernoulli variable is a discrete random variable with exactly two outcomes, either true or false with a probability. In our case, true represents a car and false is no car. This function was implemented in our code in the Bernoulli function, which takes a float p, the probability, as a parameter and returns either true or false. The stm_32 has an onboard RNG (random number generator) which returns a value between 0 and 2^{32}. We next have to normalize this value to be between 0 and 1 to be compatible with our p, this value is called: "random_number". This value is then compared to our probability and if the random number is lower than p we return true, otherwise return false. So if p=1, all possible random numbers would be lower and it would always return true (car), and if p=0, no random numbers would be lower and it would always return false (no car).

```
//generates a car with probability proportional to the function's argument
bool bernoulli(float p){
        while(!RNG_GetFlagStatus(RNG_FLAG_DRDY));
        float random_number = RNG_GetRandomNumber() / pow(2,32);
        if(p > 0.98){
                return true;
        }else if(p < 0.02){
                return false;
        }
        else if(random_number  <= p){
                return true;
        }else{
                return false;
        }
}
```

Figure 12: Bernoulli Function

The traffic flow level is first read from the traffic queue using the xQueuePeek() function. This function allows us to read the value without removing it from the queue, preserving its value for other tasks to read. The value from the queue is stored in the float “probability”. If the value was successfully read from the queue then the Bernoulli function is called to determine if the next space in the road contains a car or not. This Bool is then written to the queue using the xQueueOverwrite function, replacing the previous car.

```
//function to get current traffic flow rate value and generate sequence of cars
static void traffic_generator_task(void *pvParameters){
        for(;;){
                float probability;
                bool car;
                BaseType_t queue_read = xQueuePeek(xTraffic_Queue_handle,&probability,0);
                if(queue_read == pdPASS){
                        car = bernoulli(probability);
                        xQueueOverwrite(xCar_Queue_handle, &car);
                }
                vTaskDelay(pdMS_TO_TICKS(400));
        }

}
```

Figure 13: Traffic Generator Task Program Listing

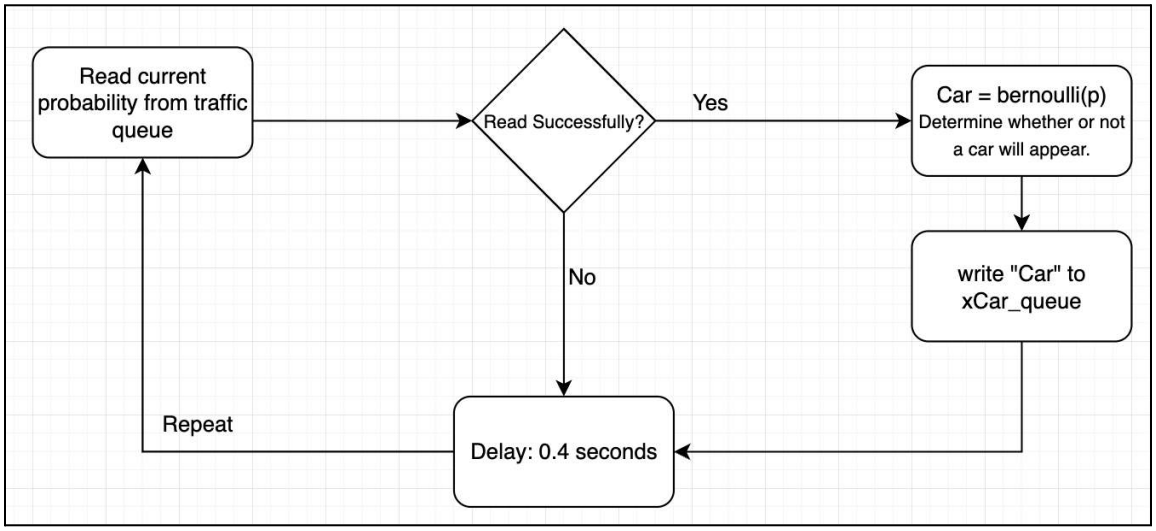


Figure 14: Traffic generator task flowchart

Traffic Light State Task

Figure 11 shows the code implementing the timer callback for the traffic_light process. The callback begins by obtaining the previous state of the traffic light from the traffic_light queue. This state is represented with an integer and is set to either 0 (green), 1 (amber), or 2 (red). Upon obtaining the previous state, the traffic flow level is also red from the corresponding queue. The callback verifies that each read is successful and initializes the traffic_light state with a default value of 0 in case the queue read operations were unsuccessful. This occurs during the first iteration of this callback due to the offset value selected for the timer. The offset of 0 causes this callback to run before the first iteration of the tasks.

For each case of the light state, the timer period is adjusted ensuring the red light duration is inversely proportional to traffic flow and the green light duration is directly proportional. At a maximum traffic level, the green light is 6 seconds and the red light is 3 seconds. At a minimum traffic level, the green light is 3 seconds and the red light is 6 seconds. The amber light duration remains at a constant 3 seconds regardless of the traffic flow. Upon setting the timer period, the light state is updated to be the next state in the sequence (green, amber, red, green …) and the traffic_light queue is overwritten with the updated value.

```
//function to get current traffic flow rate, determine light durations, manage traffic light transitions
static void traffic_light_callback(TimerHandle_t xTimer){
        //0 = green
        //1 = amber
        //2 = red
        int traffic_light;
        float traffic_level;
        BaseType_t light_queue_read = xQueuePeek(xTraffic_Light_Queue_handle,&traffic_light,0);
        BaseType_t traffic_queue_read = xQueuePeek(xTraffic_Queue_handle,&traffic_level,0);
        if(!light_queue_read || !traffic_queue_read){
                //queue read fails on startup, set initial light as green
                traffic_light = 0;
                printf("Light Callback: failed to read from queue");
        }
        // light is green, next state yellow
        if(traffic_light == 0){
                xTimerChangePeriod(xTimer,pdMS_TO_TICKS(YELLOW_LIGHT_DURATION),0);
                traffic_light++;
        // light is yellow, next state red
        }else if(traffic_light == 1){
                int duration = 6000 - (3000 * traffic_level);
                traffic_light++;
                xTimerChangePeriod(xTimer,pdMS_TO_TICKS(4000),0);


        // light is red, next state green
        }else{
                traffic_light = 0;
                int duration = 3000.0 + (3000.0 * traffic_level);
                xTimerChangePeriod(xTimer,pdMS_TO_TICKS(duration),0);
        }
        xQueueOverwrite(xTraffic_Light_Queue_handle, &traffic_light);
}
```

Figure 15: Traffic light callback function

System Display Task

Figure 19 shows a flowchart for the system display task and the full program listing can be seen in the appendix. This task relies on two internal variables to maintain the road's state, a traffic light state variable and an array of Bools which represents all the spaces for cars on the road. The task begins by writing the previous road state to the shift registers using the function: output_traffic. This function is shown in figure 16 below and it works by simulating clock pulses by repeatedly setting and resetting GPIO pin 7 which is the clock on the shift registers. We then write the state of the car by writing either a 0 or a 1 to the data pin. This is repeated 20 times, once for each car and one more pulse which pushes the last car through the register.

```
//generates clock pulses for shift registers, toggles data pin to transmit traffic array to register
void output_traffic(bool* traffic_buffer){
        //start at end of buffer so that last bit set to last LED after 20 clks
        for(int i = ROAD_LENGTH-1; i >= 0; i--){
                        GPIO_SetBits(GPIOC, GPIO_Pin_7);
                        if(traffic_buffer[i]){
                                GPIO_SetBits(GPIOC, GPIO_Pin_6);
                        }else{
                                GPIO_ResetBits(GPIOC, GPIO_Pin_6);
                        }
                        GPIO_ResetBits(GPIOC,GPIO_Pin_7);
                        for(int i = 0; i < 100; i ++){
                        }
                }
}
```

Figure 16: Traffic output function

Next, the task reads the traffic light state from the traffic queue and based on its value (0,1 or 2) the system will update the LEDs accordingly. When the traffic light state is 0, this means we have a green light. We can turn on the green LED by setting the gpio pin C3 using the GPIO_SetBits and ensure that the red and amber LEDs are off using GPIO_ResetBits. Since we have a green light, all cars are allowed to move one space down the road so each index in the traffic array is incremented in the temp array.

```
//light is green, all cars propagate
if(traffic_light_state == 0){
        GPIO_SetBits(GPIOC,GPIO_Pin_2);
        GPIO_ResetBits(GPIOC, GPIO_Pin_1 | GPIO_Pin_0);
        for(int i = 0; i < 20;i++){
                temporary[i+1] = traffic_array[i];
        }
```

Figure 17: traffic light state 0

When the traffic light state is 1 or 2, this means that the light is amber or red, respectively. During these states, cars after the stop line (index 8) are allowed to propagate and cars before the line are only allowed to move forwards if a space is available. To accomplish this, the following algorithm is used:

1. Copy index 8 of traffic array to temp array
2. For each index from 0-7:
    a. Check if the next index in temp array is empty (FALSE)
    b. If it is, write the current value of the traffic array to the next position in the temp array and write a zero to the current position in the temp array.
    c. Otherwise, write the current value of the traffic array in the same position in the emp array.
3. Increment all the indexes after 8 (9-19)

The traffic lights are also updated in these states. Similarly to the green state, when the traffic light state is 1, we set pin 1 and reset pins 0 and 2. Effectively turning on the yellow LED and turning off the green and red. When the traffic light state is 1, we set pin 0 and reset pins 1 and 2. Turning on the red LED and turning off the green and yellow LEDs.

```
        }
//light is yellow, stop all cars before line (positions 0 through 8), propagate all cars past position 8
}else if(traffic_light_state == 1){
        GPIO_SetBits(GPIOC,GPIO_Pin_1);
        GPIO_ResetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_2);
        temporary[8] = traffic_array[8];
        for(int i = 7; i >= 0; i--){
                if(temporary[i+1]){
                        temporary[i] = traffic_array[i];
                }else{
                        temporary[i+1] = traffic_array[i];
                        temporary[i] = false;
                }
        }
        for(int i = 9; i < 20;i++){
                temporary[i+1] = traffic_array[i];
        }
//light is red stop all cars before line (positions 0 through 8), propagate all cars past position 8
}else{
        GPIO_SetBits(GPIOC,GPIO_Pin_0);
        GPIO_ResetBits(GPIOC, GPIO_Pin_1 | GPIO_Pin_2);
        temporary[8] = traffic_array[8];
        for(int i = 7; i >= 0; i--){
                if(temporary[i+1]){
                        temporary[i] = traffic_array[i];
                }else{
                        temporary[i+1] = traffic_array[i];
                        temporary[i] = false;
                }
        }
        for(int i = 9; i < 20;i++){
                temporary[i+1] = traffic_array[i];
        }
```

Figure 18: traffic light state 1 and 2

Once each state is complete, we add the next car from the car queue to the first position of the road if it is clear and then the temp array is copied back to the traffic array, setting it up for the next incrementation. As a final check, if a car has not been added to the road in 5 iterations, then we manually add one. This is a worst case since if the probability is set too low then it will never add a car.

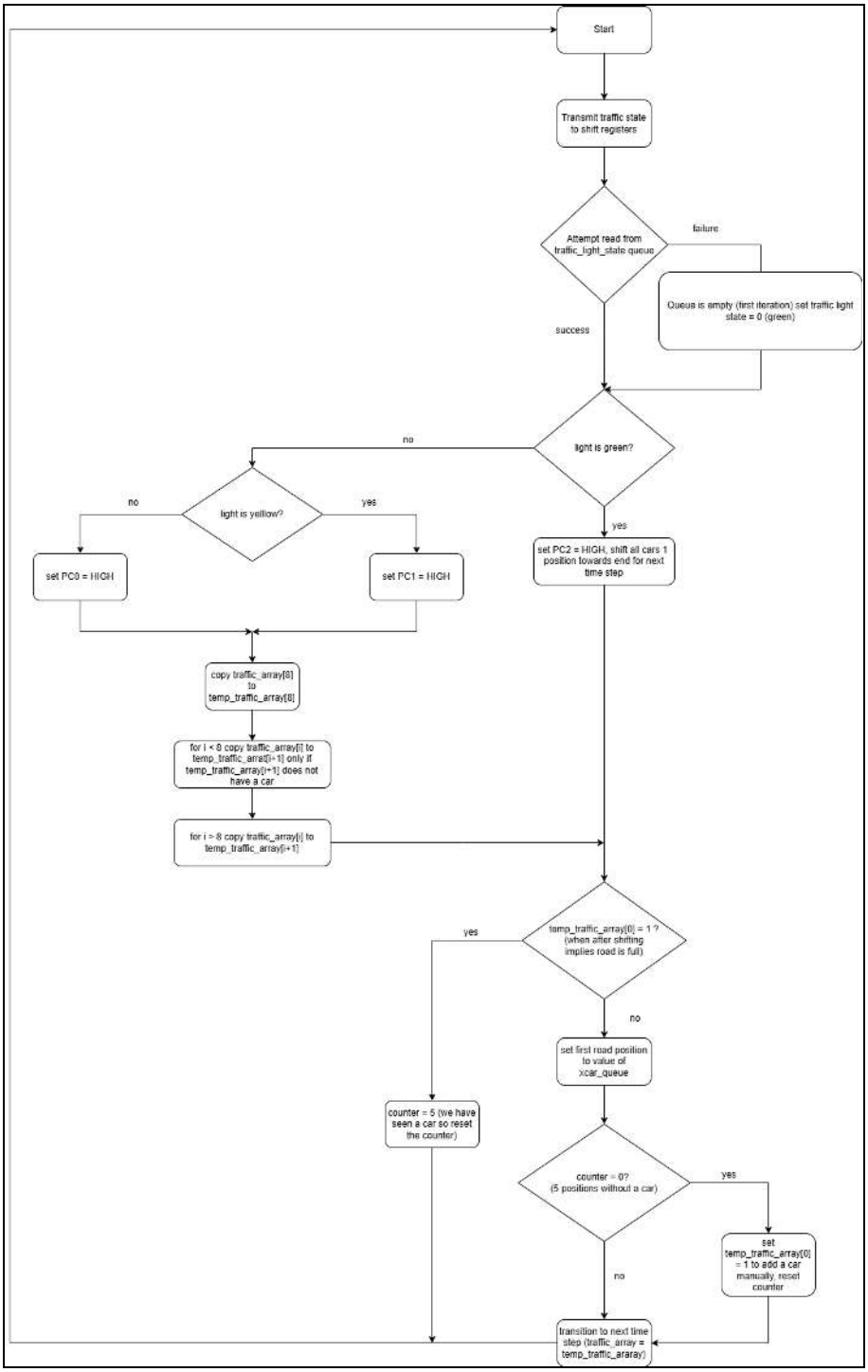


Figure 19: Display Task Flowchart

Limitations and possible improvements

Although our solution met all the requirements specified for the project, improvements could be made in the resource usage of the application.

According to the FreeRTOS API documentation, message and stream buffers are queue alternatives that are faster and consume less RAM. As message buffers are built on stream buffers, the stream buffer is the most resource efficient. A stream buffer could be used as an efficient alternative for the car queue. Stream (and message) buffers are not a suitable alternative for the light state or traffic generation queues as they assume a single reader and single writer. As shown in the block diagram of Figure 1, these queues have a single writer but multiple readers [2].

For the purposes of the lab demonstration, suitable periods for the tasks were found using trial and error. The application could be made more deterministic using a formal clock driven scheduling scheme. Each task could be assumed to be periodic and suitable values for the period (P), the worst case execution time (WCET), and the relative deadline (D) which characterize the periodic task could be determined in advance. These parameters could be used to determine the clock driven scheduler's required parameters (such as the hyper period).

Dynamic allocation is done on the heap using the chosen memory allocator from FreeRTOS (heap1,heap2,heap3 etc.) and is employed when the non-static API functions for creating FreeRTOS primitives are used. Static memory is placed in the section of memory dedicated to data for the application and is done at compile time. For the lab demonstration, the non-static API functions were used to create each queue, and the default memory allocator (heap4) was used. Because the queues are never deleted (de-allocated), there are no benefits to counteract the runtime overhead and the added risks of memory fragmentation and allocation failure that arise from dynamic allocation. Thus, allocating queues statically instead of dynamically would improve the reliability and efficiency of the system. This would require creating the queues using the xQueueCreateStatic FreeRTOS API functions.

The ADC sampling was implemented using periodic polling. Polling the ADC peripheral utilizes CPU cycles which could be saved by using an interrupt driven scheme. To improve the efficiency of determining the desired traffic level from the user, the ADC could be configured in continuous conversion mode, the EOC (end of conversion) interrupt for the ADC could be enabled and an ISR could be written to handle the EOC interrupts generated by the ADC. Instead of using the xQueueOverwite API function, the xQueueOverwriteFromISR could be used. Moreover, the most efficient scheme of data acquisition from a peripheral is the use of the STM32's DMA (direct memory access) controller. When configured, this controller can completely offload all read and write operations from the CPU. If DMA was used, a queue would not be required however mutual exclusion would likely be required to prevent concurrent access to the region of memory used to store the ADC readings by the tasks.

Summary

In this project, a real-time traffic light simulator was successfully designed and implemented using FreeRTOS on the STM32F407 Discovery board. The system integrated both hardware and software components to simulate dynamic traffic conditions and corresponding traffic light behavior in a structured and predictable manner.

On the software side, the application was divided into separate tasks and timer callbacks to manage traffic flow adjustment, car generation, traffic light state transitions, and system display. Queues were used for inter-task communication, allowing different processes to share data in a controlled and thread-safe way. The preemptive round-robin scheduling provided by FreeRTOS ensured that tasks of equal priority executed fairly without starvation, while maintaining a relatively simple and understandable system architecture. On the hardware side, shift registers were used to efficiently control multiple LEDs while minimizing GPIO usage. The ADC was configured to read the potentiometer value and convert it into a traffic probability, and the onboard random number generator was used to model traffic generation as a Bernoulli process. These elements demonstrated effective hardware–software integration within an embedded system context.

Although the system satisfied all project requirements, several potential improvements were identified. Static memory allocation could improve reliability by eliminating risks associated with heap usage, and interrupt- or DMA-based ADC sampling could reduce CPU overhead. Additionally, a more formal timing analysis could improve system determinism compared to the empirically selected task periods used in this implementation.

The project provided practical experience in real-time system design, task scheduling, peripheral configuration, and embedded hardware integration. It reinforced key concepts in RTOS-based development and demonstrated how structured software architecture can be combined with microcontroller peripherals to implement a responsive real-time application.

References

[1] University of Victoria, "Project 1: Traffic Light System," in *ECE 455: Real Time Computer Systems Lab Manual*, Victoria, BC: Dept. of Electrical and Computer Engineering, 2019, pp. 11-15.

[2] FreeRTOS.org, “Decrease Ram footprint and accelerate execution with FreeRTOS notifications - freertosTM,” FreeRTOS, https://www.freertos.org/Community/Blogs/2020/decrease-ram-footprint-and-accelerate-execution-with-freertos-notifications (accessed Mar. 4, 2026).

[3] STMicroelectronics, "RM0090 Reference manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs," Rev. 18, Feb. 2019. [Online]. Available: http://www.st.com

Appendix A - Complete Application Code Listings

Listing 1: main.c

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include <stdbool.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"
/* User includes */
#include "_sys_init.h"


#define TRAFFIC_QUEUE_LENGTH (1)
#define DEFAULT_PRIORITY (configMAX_PRIORITIES -1)
#define ADC_RESOLUTION (12)
#define ADC_SAMPLING_PERIOD_MS (200)
#define CAR_QUEUE_LENGTH (1)
#define ROAD_LENGTH (20)
#define DEFAULT_GREEN_LIGHT_DURATION (3000)
#define YELLOW_LIGHT_DURATION (3000)
#define CAR_PROPAGATION_PERIOD (400)

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

static void prvSetupHardware( void );
//timer callbacks
static void traffic_flow_callback(TimerHandle_t xTimer);
static void traffic_light_callback(TimerHandle_t xTimer);
//tasks
static void traffic_generator_task(void *pvParameters);
static void display_task(void* pvParameters);


/*-----------------------------------------------------------*/
```

```
//Queues
xQueueHandle xTraffic_Queue_handle = NULL;
xQueueHandle xCar_Queue_handle = NULL;
xQueueHandle xTraffic_Light_Queue_handle = NULL;
//Timers
TimerHandle_t xFlow_Adjustment_Timer = NULL;
TimerHandle_t xTraffic_Light_Timer = NULL;

//generates clock pulses for shift registers, toggles data pin to transmit
traffic array to register
void output_traffic(bool* traffic_buffer){
      //start at end of buffer so that last bit set to last LED after 20
clks
      for(int i = ROAD_LENGTH-1; i >= 0; i--){
                  GPIO_SetBits(GPIOC, GPIO_Pin_7);
                  if(traffic_buffer[i]){
                        GPIO_SetBits(GPIOC, GPIO_Pin_6);
                  }else{
                        GPIO_ResetBits(GPIOC, GPIO_Pin_6);
                  }
                  GPIO_ResetBits(GPIOC,GPIO_Pin_7);
                  for(int i = 0; i < 100; i ++){
                  }
            }
}

//generates a car with probability proportional to the function's argument
bool bernoulli(float p){
      while(!RNG_GetFlagStatus(RNG_FLAG_DRDY));
      float random_number = RNG_GetRandomNumber() / pow(2,32);
      if(p > 0.98){
            return true;
      }else if(p < 0.02){
            return false;
      }
      else if(random_number  <= p){
            return true;
      }else{
            return false;
      }
}
```

```c
int main(void)
{

	//hardware initialization
	init_trafficlight_gpio();
	init_adc();
	init_shift_register_gpio();
	//stm32 hardware random number generator
	init_rng();
	prvSetupHardware();

	//Create Queues
	xTraffic_Queue_handle = xQueueCreate(TRAFFIC_QUEUE_LENGTH,
sizeof(float));
	xCar_Queue_handle = xQueueCreate(CAR_QUEUE_LENGTH, sizeof(bool));
	xTraffic_Light_Queue_handle = xQueueCreate(1, sizeof(int));
	vQueueAddToRegistry(xCar_Queue_handle,"Car Queue");
	vQueueAddToRegistry(xTraffic_Queue_handle, "Traffic Queue");
	vQueueAddToRegistry(xTraffic_Light_Queue_handle,"Traffic light
Queue");

	//Create Tasks
	xTaskCreate(traffic_generator_task,"Traffic Generation",
configMINIMAL_STACK_SIZE,NULL,DEFAULT_PRIORITY,NULL);
	xTaskCreate(display_task,"Display Task",
configMINIMAL_STACK_SIZE,NULL,DEFAULT_PRIORITY,NULL);

	//Create Timers
	xFlow_Adjustment_Timer = xTimerCreate("Traffic Flow
Adjustment",pdMS_TO_TICKS(ADC_SAMPLING_PERIOD_MS),
pdTRUE,NULL,traffic_flow_callback);
	xTraffic_Light_Timer = xTimerCreate("Traffic Light
timer",pdMS_TO_TICKS(DEFAULT_GREEN_LIGHT_DURATION),
pdTRUE,NULL,traffic_light_callback);
	if(xFlow_Adjustment_Timer != NULL && xTraffic_Light_Timer != NULL){
		BaseType_t FlowTimerStarted =
xTimerStart(xFlow_Adjustment_Timer,0);
		BaseType_t LightTimerStarted =
xTimerStart(xTraffic_Light_Timer,0);
		if(FlowTimerStarted == pdPASS && LightTimerStarted == pdPASS){
			//start scheduler if timers created successfully
			vTaskStartScheduler();
```

```
            }
      }

      for(;;){
            printf("uh oh, vTaskStartScheduler returned :(");
      }

      return 0;
}


/*-----------------------------------------------------------*/
                  /* USER TASKS*/
/*-----------------------------------------------------------*/

//function to sample ADC, converts adc value to traffic flow rate, write
value to queue
static void traffic_flow_callback(TimerHandle_t xTimer){
      float adc_code;
      float probability;
      ADC_SoftwareStartConv(ADC1);
      while (ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC)!= SET ){}
      adc_code = ADC_GetConversionValue(ADC1);
      probability = adc_code / pow(2,ADC_RESOLUTION);
      xQueueOverwrite(xTraffic_Queue_handle, &probability);
}

//function to get current traffic flow rate value and generate sequence of
cars
static void traffic_generator_task(void *pvParameters){
      for(;;){
            float probability;
            bool car;
            BaseType_t queue_read =
xQueuePeek(xTraffic_Queue_handle,&probability,0);
            if(queue_read == pdPASS){
                  car = bernoulli(probability);
                  xQueueOverwrite(xCar_Queue_handle, &car);
            }
            vTaskDelay(pdMS_TO_TICKS(400));
      }
```

```
}

//function to get current traffic flow rate, determine light durations,
manage traffic light transitions
static void traffic_light_callback(TimerHandle_t xTimer){
	//0 = green
	//1 = amber
	//2 = red
	int traffic_light;
	float traffic_level;
	BaseType_t light_queue_read =
xQueuePeek(xTraffic_Light_Queue_handle,&traffic_light,0);
	BaseType_t traffic_queue_read =
xQueuePeek(xTraffic_Queue_handle,&traffic_level,0);
	if(!light_queue_read || !traffic_queue_read){
		//queue read fails on startup, set initial light as green
		traffic_light = 0;
		printf("Light Callback: failed to read from queue");
	}
	// light is green, next state yellow
	if(traffic_light == 0){

xTimerChangePeriod(xTimer,pdMS_TO_TICKS(YELLOW_LIGHT_DURATION),0);
		traffic_light++;
	// light is yellow, next state red
	}else if(traffic_light == 1){
		int duration = 6000 - (3000 * traffic_level);
		traffic_light++;
		xTimerChangePeriod(xTimer,pdMS_TO_TICKS(4000),0);


	// light is red, next state green
	}else{
		traffic_light = 0;
		int duration = 3000.0 + (3000.0 * traffic_level);
		xTimerChangePeriod(xTimer,pdMS_TO_TICKS(duration),0);
	}
	xQueueOverwrite(xTraffic_Light_Queue_handle, &traffic_light);
}

//reads traffic light state and car queues, maintains and updates the road
state, outputs the traffic light and road states to LEDs
static void display_task(void* pvParameters){
```

```
	bool traffic_array[ROAD_LENGTH] =
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	int traffic_light_state;

	//enforce maximum gap of 5 at minimum traffic flow rate
	int counter = 5;
	for(;;){
		output_traffic(traffic_array);
		BaseType_t queue_read =
xQueuePeek(xTraffic_Light_Queue_handle,&traffic_light_state,0);
		if(!queue_read){
			traffic_light_state = 0;
		}
		bool temp_traffic_array[ROAD_LENGTH] = {};
		//light is green, all cars propagate
		if(traffic_light_state == 0){
			GPIO_SetBits(GPIOC,GPIO_Pin_2);
			GPIO_ResetBits(GPIOC, GPIO_Pin_1 | GPIO_Pin_0);
			for(int i = 0; i < 20;i++){
				temp_traffic_array[i+1] = traffic_array[i];
			}
		//light is yellow, stop all cars before line (positions 0
through 8), propagate all cars past position 8
		}else if(traffic_light_state == 1){
			GPIO_SetBits(GPIOC,GPIO_Pin_1);
			GPIO_ResetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_2);
			temp_traffic_array[8] = traffic_array[8];
			for(int i = 7; i >= 0; i--){
				if(temp_traffic_array[i+1]){
					temp_traffic_array[i] = traffic_array[i];
				}else{
					temp_traffic_array[i+1] = traffic_array[i];
					temp_traffic_array[i] = false;
				}
			}
			for(int i = 9; i < 20;i++){
				temp_traffic_array[i+1] = traffic_array[i];
			}
		//light is red stop all cars before line (positions 0 through
8), propagate all cars past position 8
		}else{
			GPIO_SetBits(GPIOC,GPIO_Pin_0);
```

```
                GPIO_ResetBits(GPIOC, GPIO_Pin_1 | GPIO_Pin_2);
                temp_traffic_array[8] = traffic_array[8];
                for(int i = 7; i >= 0; i--){
                      if(temp_traffic_array[i+1]){
                            temp_traffic_array[i] = traffic_array[i];
                      }else{
                            temp_traffic_array[i+1] = traffic_array[i];
                            temp_traffic_array[i] = false;
                      }
                }
                for(int i = 9; i < 20;i++){
                      temp_traffic_array[i+1] = traffic_array[i];
                }
          }
          if(!temp_traffic_array[0]){
                xQueuePeek(xCar_Queue_handle,temp_traffic_array,0);
                counter--;
          }else{
          //add a car to the road if first 5 spots are missing cars
                counter=5;
          }
          if(counter == 0){
                temp_traffic_array[0] = true;
                counter = 5;
          }
          for(int i = 0; i < ROAD_LENGTH; i++){
                traffic_array[i] = temp_traffic_array[i];
          }
          //This controls car propagation speed
          vTaskDelay(pdMS_TO_TICKS(CAR_PROPAGATION_PERIOD));
      }

}

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. */
      }
}
/*-----------------------------------------------------------*/

static void prvSetupHardware( void )
{
      /* Ensure all priority bits are assigned as preemption priority bits.
      http://www.freertos.org/RTOS-Cortex-M3-M4.html */
      NVIC_SetPriorityGrouping( 0 );
}
```

Listing 2: sys_init.c

```
#include "_sys_init.h"
#include <math.h>

void init_trafficlight_gpio(void){
      GPIO_InitTypeDef  GPIO_InitStructure;
      RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
      //traffic lights on Pin 0 through Pin 2 on GPIO Peripheral C
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
      GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(GPIOC, &GPIO_InitStructure);
      GPIO_ResetBits(GPIOC, GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2);
}

void init_shift_register_gpio(void){
      GPIO_InitTypeDef  GPIO_InitStructure;
      RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
      // Shift Register Pins on Pin 6 through Pin 8 On GPIO Peripheral C
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
      GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_Init(GPIOC, &GPIO_InitStructure);
      GPIO_ResetBits(GPIOC, GPIO_Pin_6 | GPIO_Pin_7);
```

```
	GPIO_SetBits(GPIOC, GPIO_Pin_8);

}

void init_adc(void){
	ADC_InitTypeDef ADC_Init_Structure;
	GPIO_InitTypeDef  GPIO_InitStructure;
	//PC3 used as ADC input Pin
	GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;

	//Configure PC3 as an analog input
	GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
	GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
	GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
	GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
	//STM32407 Driver function for initialization of ADC with suitable
defaults
	//continuous conversion is disabled by this function
	ADC_StructInit(&ADC_Init_Structure);
	RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
	RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);


	GPIO_Init(GPIOC, &GPIO_InitStructure);
	ADC_Init(ADC1, &ADC_Init_Structure);
	ADC_Cmd(ADC1, ENABLE);
	//ADC1 channel 13 is used.
	ADC_RegularChannelConfig(ADC1 ,ADC_Channel_13,1,ADC_SampleTime_3Cycles);
}

void init_rng(void){
	//initialize the random number generator peripheral of the STM32
	 RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);
	 RNG_Cmd(ENABLE);
}



void test_traffic_light(void){
	//test traffic light GPIOs
	GPIO_SetBits(GPIOC, GPIO_Pin_0);
	for(int i = 0; i < 5000000; i++){
	}
	GPIO_ResetBits(GPIOC, GPIO_Pin_0);
	GPIO_SetBits(GPIOC, GPIO_Pin_1);
```

```
	for(int i = 0; i < 5000000; i++){
	}
	GPIO_ResetBits(GPIOC, GPIO_Pin_1);
	GPIO_SetBits(GPIOC, GPIO_Pin_2);
	for(int i = 0; i < 5000000; i++){
	}
	GPIO_ResetBits(GPIOC, GPIO_Pin_2);
	printf("traffic light complete\n");
}

void test_adc(void){
	for(int i= 0;i<6;i++){
		ADC_SoftwareStartConv(ADC1);
		while (ADC_GetFlagStatus(ADC1,ADC_FLAG_EOC)!= SET ){}
		double ADC_CODE;
		double ADC_Voltage;
		ADC_CODE = ADC_GetConversionValue(ADC1);
		ADC_Voltage = (ADC_CODE*3.3/pow(2, 12)) * 1000;
		for(int i = 0; i < 5000000; i++){
		}
	}
	printf("ADC test complete\n");
}
void test_shift_register(void){
	uint8_t pattern[20] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
	for(int i = 0; i < 20; i++){
		GPIO_SetBits(GPIOC, GPIO_Pin_7);
		if(pattern[i]){
			GPIO_SetBits(GPIOC, GPIO_Pin_6);
		}else{
			GPIO_ResetBits(GPIOC, GPIO_Pin_6);
		}
		GPIO_ResetBits(GPIOC,GPIO_Pin_7);
		for(int i = 0; i < 1000; i ++){
		}
	}

}

void system_test(void){
	test_traffic_light();
	test_adc();
	test_shift_register();
}
```

Listing 3: sys_init.h

```
#ifndef STM32F4XX_STDPERIPH_DRIVER_INC__SYS_INIT_H_
#define STM32F4XX_STDPERIPH_DRIVER_INC__SYS_INIT_H_
#include "stm32f4xx.h"
void init_trafficlight_gpio(void);
void init_shift_register_gpio(void);
void init_adc(void);
void init_rng(void);
void test_traffic_light(void);
void test_adc(void);
void test_shift_register(void);
void system_test(void);
#endif /* STM32F4XX_STDPERIPH_DRIVER_INC__SYS_INIT_H_ */
```

Appendix B - Project Design Document

Processes

1. System Initialization
    a. Description:This code should run before the scheduler begins. It is responsible for initializing all peripherals, creating the required tasks, creating the resource management structures (Queues, streams etc.)
    b. Requirements
        i. Initialize GPIOs (PC0 through PC6 as digital outputs)
        ii. Initialize GPIO 3 as an analog input (potentiometer)
        iii. Create all tasks, assign priority
        iv. Start the scheduler
    c. Internal Variables
        i. None
2. Traffic Flow Adjustment
    a. Description: This task must use the ADC to read from the external potentiometer. Polling rate should be fast enough that the user does not notice a latency in the traffic. The task should convert from the measured voltage to a traffic level. The task should develop a conversion mechanism (thresholds) to convert from the measured voltage to a traffic level. The task should add the traffic level to a queue
    b. Requirements
        i. Establish a polling rate for the ADC such that the traffic level is responsive (immediate) when the pot is adjusted (VtaskDelayUntil)
        ii. Read the voltage measured by the ADC at the established polling rate
        iii. Convert from voltage to traffic level
        iv. Add the traffic level to a queue
    c. Internal Variables
        i. None
3. Traffic Light State
    a. Description: Similar to how lab 0 was done. we will have a set of subtasks that turn on/off the leds. Have separate tasks to control each light. Depending on the traffic flow level, the durations for blocking the tasks should change.
    b. Requirements
        i. Proportional to the traffic flow level, the durations of red, and green should change.
        ii. At Maximum flow: light should stay green twice as long as red
        iii. At minimum flow: The light should stay re twice as long as green.
        iv. Amber light duration should always stay constant.
    c. Internal Variables
        i. Light State (int)
4. Display Task
    a. Description: This task is responsible for moving the cars down the road. This task should maintain an internal array. It should "divide" the array into two sections.

For all indices after the first 8, this task should always propagate them towards the end at each iteration. For the first 8 positions this task should monitor the light state. If the light is green, cars should be propagated as usual. If the light is yellow or red, cars should be stopped. They should not propagate past the last position. Cars generated behind these cars should continue propagating.

i. Also:
    1. Updates all LEDs on the board.
    2. Fetch the Light state and the traffic array from other tasks, then push to the board.
    3. Should create a buffer for the shift register.

b. Requirements
    i. Maintain u8 arr[19] for car positions.
    ii. Shift cars every cycle (vTaskDelayUntil).
    iii. Indices 8–18: always move forward.
    iv. Indices 0–7: move only if light is Green; stop if Red/Yellow.
    v. Do not allow cars past final index.
    vi. Get Light State from queue.
    vii. Update LEDs using a shift register buffer.
c. Internal Variables
    i. Light state (int)
    ii. Traffic Array (u8 arr[19])