ECE 455
Project 1 Report

March 5th, 2025

Tay Munro – V00965447

Jett Lam – V00959283

Table of Contents

1.0 Introduction

In the lab section of this course, we were tasked with constructing a traffic simulator using an STM32 Discovery board, a breadboard with components, and code to tie it all together. This project focused on using FreeRTOS to allow different parts of the system to work together smoothly. The traffic light system simulates a one-way road with an intersection and a traffic light in the middle. The cars are represented by LED lights that turn on and move along the road by cycling power states.

The system includes three main parts:

1. **Traffic Light** – A set of red, yellow, and green LEDs that tell the cars when they can go or stop.
2. **Cars** – LEDs that show where the cars are along the road and move forward over time.
3. **Potentiometer** – A component used to control how often new cars appear on the road.

This project organizes the system into separate tasks such as Traffic Flow Adjustment, Traffic Generator, Traffic Light State, and System Display to ensure that the cars, lights, and potentiometer all work together as expected. Our code also makes use of queues to store the light data, cars, and traffic flow, ensuring efficient system operation.

2.0 Design

Before starting with the code, or the wiring on this project, our group put together a design document that can be seen in this section. After completion of the design document, we got to work on the actual project. The final design of the project can also be seen in this section along with the differences between the design talked about in the design document and the final design.

2.1 Design Document

We will use an iterative approach for this design while incrementally working on the software and hardware. To set up a microcontroller-controlled traffic light system we first initialize the GPIO pins and connect red, yellow, and green LEDs. Then, program traffic light timers for correct operation. Next, configure the ADC and GPIO for a potentiometer, connect it, and adjust the light timings based on its value. Initialize GPIO ports for shift registers, wire them to control car movement and verify functionality. Finally, program the

shift registers to simulate traffic flow, stopping cars at red/yellow lights and increasing car frequency when the potentiometer value is higher. Here are the steps to follow:

1. Initialize GPIO pins for the traffic light.
2. Wire red, yellow and green LEDs to the breadboard and ensure lights turn on.
3. Program traffic light timers to turn on and off with correct timings.
4. Initialize ADC and GPIO pins for the potentiometer.
5. Add the potentiometer to the breadboard and connect it to the microcontroller.
6. Program the traffic light timers to react to the changes in the ADC value.
7. Initialize GPIO ports for shift registers.
8. Connect shift registers to car lights and microcontroller.
9. Make sure the lights turn on.
10. Program shift registers to move cars along and stop cars on red/yellow lights.
11. Program shift registers to add cars more often if ADC value is higher.

Here is a rough hardware design made using a breadboard design tool can be seen in Figure 1 below.

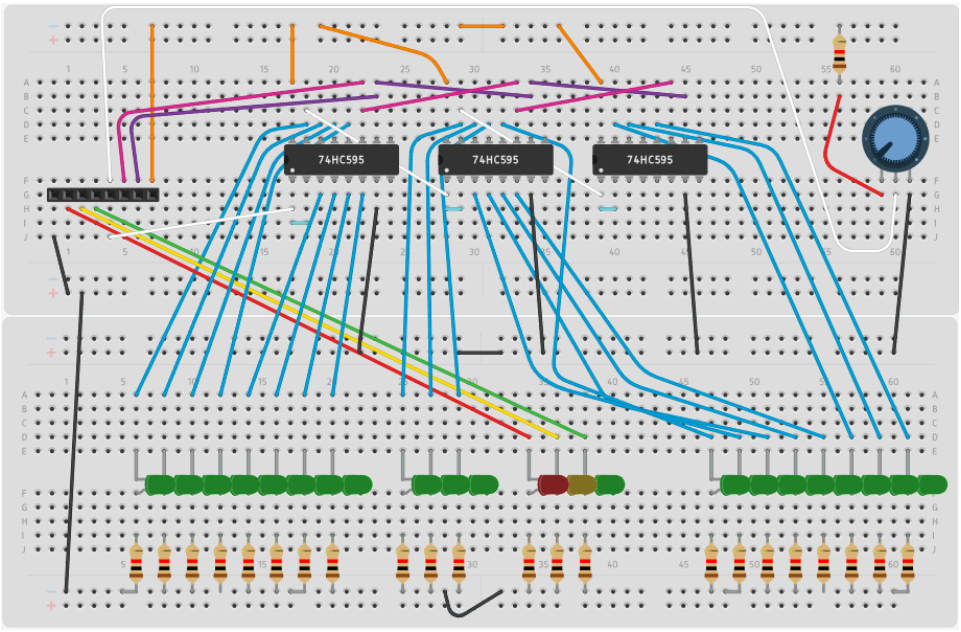

Figure 1: Rough hardware design for traffic simulator project.

2.2 Final Design

The final design of the traffic light system integrates FreeRTOS tasks, queues, and GPIO control to create a functional traffic simulation (see Figure 3 in Appendix for system overview diagram). The system is structured into four main tasks:

1. **Traffic Light Task** – This task manages the state of the traffic light, cycling through green, yellow, and red LEDs based on the traffic flow rate. The duration of each light is dynamically adjusted using ADC input from the potentiometer.
2. **Traffic Generator Task** – This task generates new cars at a rate determined by the potentiometer. The generated cars are added to a queue and processed accordingly (see Figure 4 in Appendix for flow diagram).
3. **System Display Task** – This task updates the LED representation of the cars on the road. It shifts LED states to simulate car movement and changes its behavior based on the state of the light (see Figure 5 in Appendix for flow diagram).
4. **Traffic Flow Task** – This task reads the potentiometer input and determines the appropriate timing for traffic light changes and car generation.

The system uses FreeRTOS queues to pass information between tasks, to avoid global variables. Shift registers are used to control the car LEDs to ensure they move properly. Timers are used to make sure the cars, lights, and traffic generation are all synced up according to the project specifications. The GPIO pins we used are the same as stated in the lab manual and can be seen in Table 1.

STM32F0 Port Pin	Signal	Notes
PC0	Red Light	Traffic Light
PC1	Amber Light	Traffic Light
PC2	Green Light	Traffic Light
		Traffic flow shift register
PC8	Shift Register Reset	Active Low, minimum 1us period
PC7	Shift Register Clock	Falling edge trigger
PC6	Shift Register Data	After clock falling edge data hold time of a minimum of 1 us
PC3	Potentiometer input	0 – 3 Volt Input

Table 1: GPIO ports on the STM32 Discovery board and their uses.

The most complex of the 4 tasks we implemented was the System Display Task. This was the task that oversaw the movement of the cars. We used an array-based implementation to implement the cars. We created an array of 19 integers all initially set to 0 and used this array to simulate each light on the road. If the integer in the array was set to 1 then there was a car in that place on the road, id it was set to 0 then there was no car in that place. When the light was set to green, the array was shifted one place to the left for each tick of the system as shown in the below code snippet:

```
   if(traffic_light_status == GREEN_LED){
            // If the light is green, shift the array one unit to the right for
each tick
            for(int i = 19; i>0; i--){
                car_traffic[i] = car_traffic[i-1];
            }
        }
```

If the light turned yellow or red, all lights before the intersection would stack up but not move beyond the intersection, and all of the lights in and after the intersection would move as normal. This behavior can be seen in the code snippet below:

```
else {
            // Else, shift cars according to what needs to be done on a
red/yellow light
            for (int i=8; i>0; i--){
                // First 8 cars (18 to 11) stop before the lights
                if(!(car_traffic[i])){
                    // If next car is empty, shift current car to next position
and set current position to empty
                    car_traffic[i] = car_traffic[i-1];
                    car_traffic[i-1] = 0;
                }
            }
            for (int i=19; i>9; i--){
                // Move cars normally after the lights.
                car_traffic[i] = car_traffic[i-1];
                car_traffic[i-1] = 0;
            }
        }
```

At the end of the task, the code checks if a new car is waiting in the traffic generator queue and if so, it adds a car to the array. An image of the final wiring of the system can be seen in Figure 2.

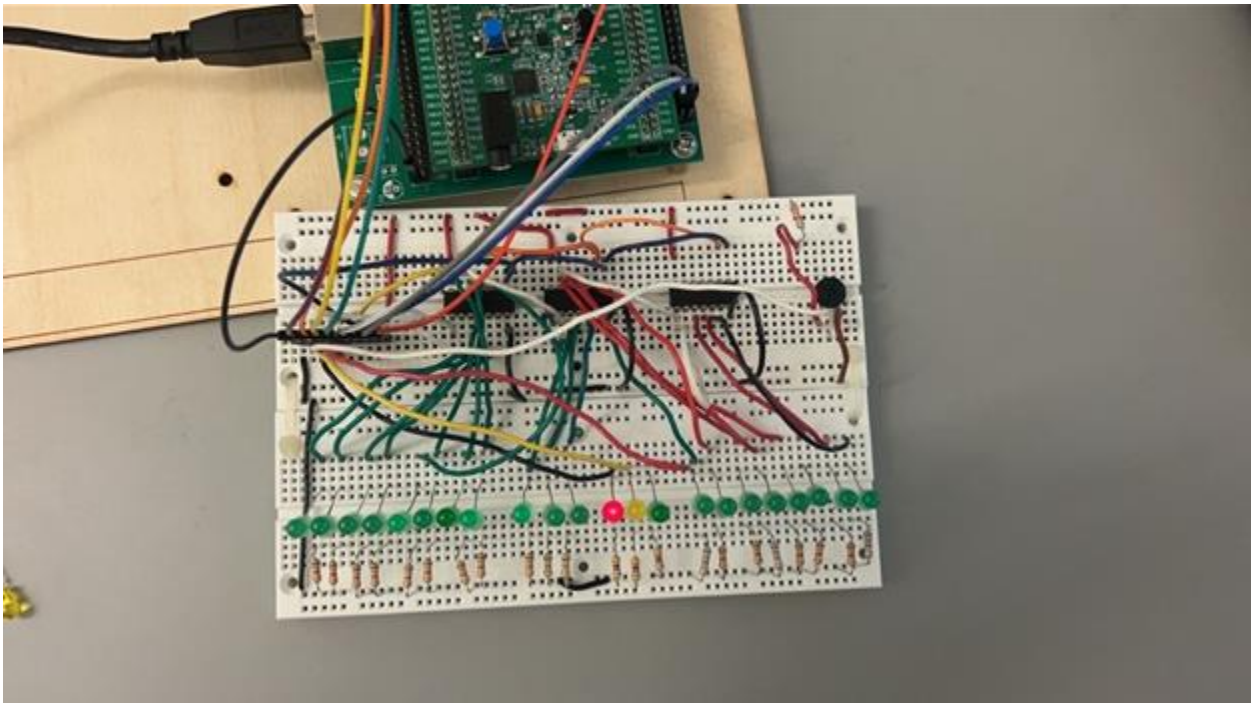

Figure 2: Final bread board design.

2.3 Differences between designs

Our initial design laid out a step-by-step process in completing and testing each part separately before combining all the sections together. We started by setting up the traffic lights, then added the potentiometer for adjusting traffic flow, and finally worked on moving the cars using shift registers. This made it easier to test each component, but it also meant we had to rewrite parts of the system to make everything work together smoothly in the final version.

One of the biggest changes was how we handled car movement. At first, we planned to use shift registers alone to move cars, but in the final version, we used an array to track car positions. This made the movement more organized and realistic, as cars would properly stop at red lights instead of moving randomly.

Another change that can be seen when comparing Figures 1 and 2 is the way the shift registers connect to the car LEDs. When we created the schematic in Figure 1, we didn't understand how the shift registers worked and wired the lights to the wrong shift register pins. This was fixed in the final design.

3.0 Discussion

Running our code (see Appendix) for the traffic simulator on the breadboard shown in Figure 2, the observed results match the expected results outlined in the lab manual. Cars are generated on the right-hand side and travel at a constant speed to the left, passing through the intersection on green lights and stopping on yellow and red. On yellow lights, cars properly clear the intersection before it turns red. At the minimum traffic setting we observed gaps of 5 LEDs between cars, with the green light running for ~3 seconds and the red light running for ~8 seconds. At the maximum there were no gaps between cars, and the green light ran for ~8 seconds and the red light for ~3 seconds. The breadboard itself worked as intended, with no shortages in components and wiring that was relatively easy to trace. Overall, our traffic system was a success.

4.0 Limitations and Possible Improvements

The greatest limitation to this project was the need for lab equipment (lab computer and STM32F4 board) in order to work on and test the source code. This required us to not only find time in both of our schedules during the day but also make extra commutes to UVic. A possible improvement would be to have some form of digital board that code could be run on remotely, allowing groups to work at any time.

Another limitation is in the "realism" of the traffic system. The design specifications required a single lane moving in the same direction (left to right). A more realistic approach would be to include multiple lanes moving in opposite directions, although this is unfeasible in the time allocated for this project and would require additional weeks of work.

5.0 Summary

The intent of this project was to design and implement a traffic simulator using LEDs controlled by software using FreeRTOS. The LEDs for the cars and traffic lights were set on a breadboard, with the former being controlled by shift registers also on the breadboard. Also on the breadboard was a potentiometer used to control traffic flow. An STM32F4 board housed the software and connected to the breadboard via GPIO pins. The software code implemented four FreeRTOS tasks to control traffic processes: a traffic generator task, traffic flow task, traffic light task, and system display task. These tasks passed information to one another through FreeRTOS queues, and used FreeRTOS delays and timers to manage input and control output.

Our final design was fairly similar to the one outlined in our design document, with minor adjustments made to fix issues with the shift registers and how car locations and movement were handled. In the end, our project was successful in simulating a traffic control system with varying amounts of traffic flow, and taught us more about how task scheduling is leveraged in various systems.

6.0 Appendix 1

In Figures 3-5 below you will find multiple diagrams showing how the components of our system communicated, as well as how the logic worked.

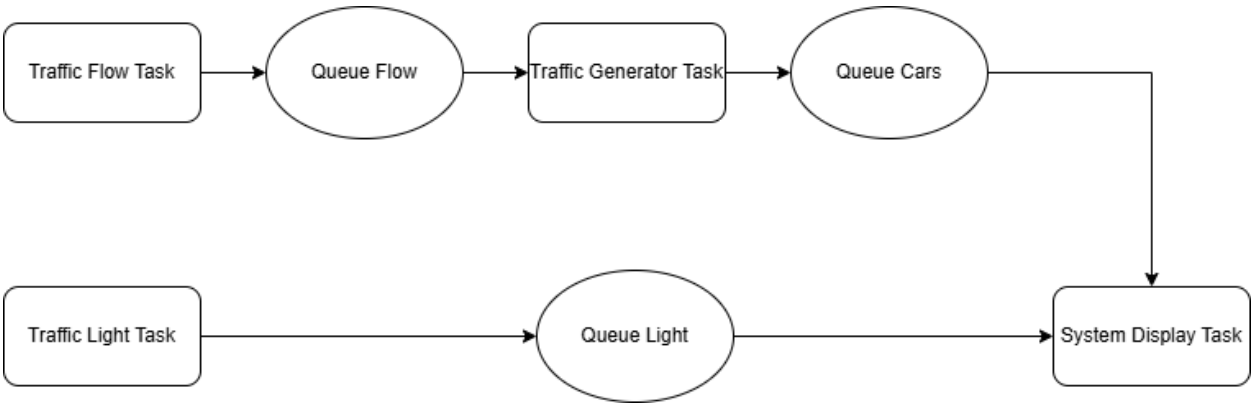

Figure 3: System overview diagram for tasks and queues

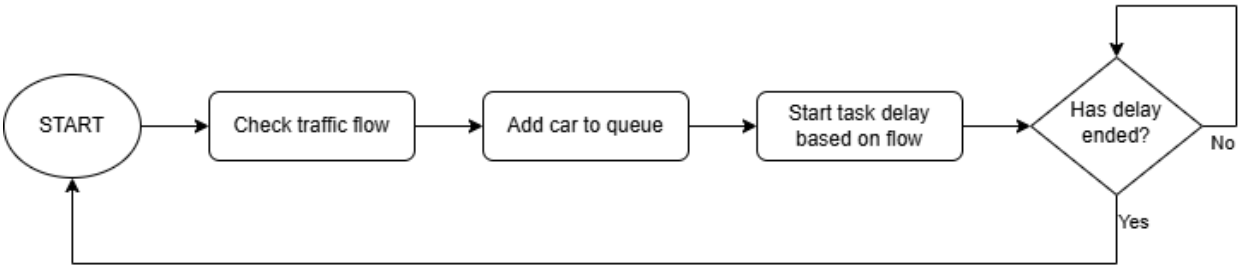

Figure 4: Fow diagram of traffic generator task

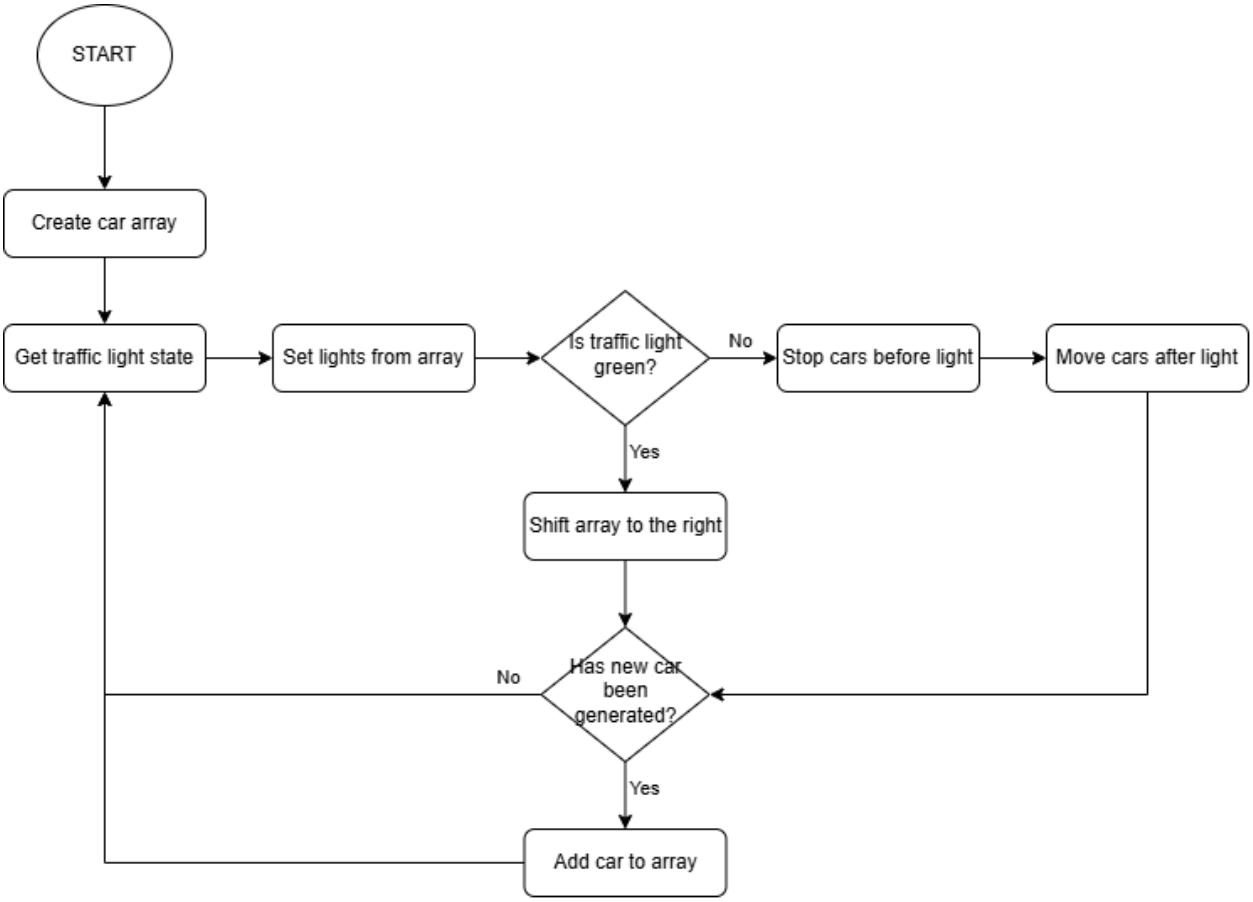

Figure 5: Flow diagram of system display task

7.0 Appendix 2

Below you will find our complete code implementation for the traffic simulator project:

```
// Tay Munro V00965447
// Jett Lam  V00959283

// ##############################################################################
//          IMPORTS GLOBAL VARIABLES AND FUNCTION DECLARATIONS
// ##############################################################################

/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include "stm32f4_discovery.h"
#include "stm32f4xx_adc.h"
#include "stm32f4xx_gpio.h"
```

```c
/* 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"

#define TRAFFIC_LIGHT_QUEUE_MAX     1
#define RED_LED                     GPIO_Pin_0
#define YELLOW_LED                  GPIO_Pin_1
#define GREEN_LED                   GPIO_Pin_2
#define POT                         GPIO_Pin_3
#define SHIFT_DATA                  GPIO_Pin_6
#define SHIFT_CLOCK                 GPIO_Pin_7
#define SHIFT_RESET                 GPIO_Pin_8
#define MIN_TIME                    3000
#define MAX_TIME                    9000

static void Traffic_Light_Task(void *pvParameters);
static void Traffic_Flow_Task(void *pvParameters);
static void System_Display_Task(void *pvParameters);
static void Traffic_Generator_Task(void *pvParameters);

static void GPIO_INIT();
static void ADC_INIT();
static void Turn_On_Green(uint16_t GLED);
static void Turn_On_Red(uint16_t RLED);
static void Turn_On_Yellow(uint16_t YLED);
static uint16_t Get_ADC_Val();
static uint16_t Time_Scale(uint16_t led);

xQueueHandle xQueue_Light = 0;
xQueueHandle xQueue_Flow = 0;
xQueueHandle xQueue_Cars = 0;

// ###########################################################################
//                              MAIN FUNCTION
// ###########################################################################

int main(void)
{
```

```
    // Initialize LEDs
    GPIO_INIT();
    ADC_INIT();

    // Create the queue used by the queue send and queue receive tasks
    xQueue_Light = xQueueCreate(TRAFFIC_LIGHT_QUEUE_MAX, sizeof( uint16_t ) );
    xQueue_Flow = xQueueCreate(TRAFFIC_LIGHT_QUEUE_MAX, sizeof( uint16_t ) );
    xQueue_Cars = xQueueCreate(TRAFFIC_LIGHT_QUEUE_MAX, sizeof( uint16_t ) );

    // Add to the registry, for the benefit of kernel aware debugging
    vQueueAddToRegistry( xQueue_Light, "LightQueue" );
    vQueueAddToRegistry( xQueue_Flow, "FlowQueue" );
    vQueueAddToRegistry( xQueue_Cars, "CarQueue" );

    xTaskCreate( Traffic_Light_Task, "Traffic_Light", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);
    xTaskCreate( Traffic_Flow_Task, "Traffic_Flow", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);
    xTaskCreate( System_Display_Task, "System_Display", configMINIMAL_STACK_SIZE,
NULL, 1, NULL);
    xTaskCreate( Traffic_Generator_Task, "Traffic_Generator",
configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    // Start the tasks and timer running
    vTaskStartScheduler();

    for ( ;; );

    return 0;
}

// ###########################################################################
//                  INITIALIZATION FUNCTIONS AND TASKS
// ###########################################################################
static void GPIO_INIT()
{
    // First enable the GPIOC clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    // Setup GPIO structure for traffic lights
    GPIO_InitTypeDef GPIO_InitStruct;
    GPIO_InitStruct.GPIO_Pin =      RED_LED | YELLOW_LED | GREEN_LED;
    GPIO_InitStruct.GPIO_Mode =     GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_Speed =    GPIO_Speed_50MHz;
```

```
    GPIO_Init(GPIOC, &GPIO_InitStruct);

    // Setup GPIO structure for pot
    GPIO_InitStruct.GPIO_Pin =      POT;
    GPIO_InitStruct.GPIO_Mode =     GPIO_Mode_AN;
    GPIO_InitStruct.GPIO_PuPd =     GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_InitStruct);

    // Setup shift registers for traffic flow
    GPIO_InitStruct.GPIO_Pin =      SHIFT_DATA | SHIFT_CLOCK | SHIFT_RESET;
    GPIO_InitStruct.GPIO_Mode =     GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_Speed =    GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStruct);
}

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

static void ADC_INIT()
{
    // First enable the ADC clock
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    // Setup ADC structure
    ADC_InitTypeDef ADC_InitStruct;
    ADC_InitStruct.ADC_ContinuousConvMode =     DISABLE;
    ADC_InitStruct.ADC_DataAlign =              ADC_DataAlign_Right;
    ADC_InitStruct.ADC_Resolution =             ADC_Resolution_12b;
    ADC_InitStruct.ADC_ScanConvMode =           DISABLE;
    ADC_InitStruct.ADC_ExternalTrigConv =       DISABLE;
    ADC_InitStruct.ADC_ExternalTrigConvEdge =   DISABLE;

    // Apply the setup to the ADC and enable
    ADC_Init(ADC1, &ADC_InitStruct);
    ADC_Cmd(ADC1, ENABLE);

    // Set the channel
    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_144Cycles);
}

// ###########################################################################
//                  TRAFFIC LIGHT TASK AND FUNCTIONS
// ###########################################################################

static void Traffic_Light_Task( void *pvParameters )
```

```
{
    uint16_t GLED = GREEN_LED;
    uint16_t YLED = YELLOW_LED;
    uint16_t RLED = RED_LED;

    // Initial reset to make sure we are always starting on neutral
    GPIO_SetBits(GPIOC, RESET);

    while(1)
    {
        Turn_On_Green(GLED);
        vTaskDelay(pdMS_TO_TICKS(Time_Scale(GLED)));

        Turn_On_Yellow(YLED);
        vTaskDelay(pdMS_TO_TICKS(MIN_TIME));

        Turn_On_Red(RLED);
        vTaskDelay(pdMS_TO_TICKS(Time_Scale(RLED)));
    }
}

static void Turn_On_Red( uint16_t RLED )
{
    xQueueOverwrite(xQueue_Light, &RLED);
    GPIO_ResetBits(GPIOC, YELLOW_LED);
    GPIO_SetBits(GPIOC, RLED);
}

static void Turn_On_Yellow( uint16_t YLED )
{
    xQueueOverwrite(xQueue_Light, &YLED);
    GPIO_ResetBits(GPIOC, GREEN_LED);
    GPIO_SetBits(GPIOC, YLED);
}

static void Turn_On_Green( uint16_t GLED )
{
    xQueueOverwrite(xQueue_Light, &GLED);
    GPIO_ResetBits(GPIOC, RED_LED);
    GPIO_SetBits(GPIOC, GLED);
}

// ###########################################################################
//                  SYSTEM DISPLAY TASK AND FUNCTIONS
```

```
// ###########################################################################

static void System_Display_Task( void *pvParameters )
{
    // Create a list of 19 cars.
    // The first 8 cars (cars 18 to 11) are before the stop line and will stop on
red.
    // The next 11 cars (10 to 0) are after the stop line and don't get affected
by the lights.
    int car_traffic[19] = {0};
    uint16_t traffic_light_status;
    GPIO_SetBits(GPIOC, SHIFT_RESET);

    // Start loop of program running
    while(1){
        // Get colour of the traffic light with 5000 ms timeout
        xQueuePeek(xQueue_Light, &traffic_light_status, 5000);
        for (int i=19; i>0; i--){
            if(car_traffic[i]){
                // If there is a car in this spot of the array, turn on
corresponding light
                GPIO_SetBits(GPIOC, SHIFT_DATA);
            } else {
                // Else turn off the corresponding light
                GPIO_ResetBits(GPIOC, SHIFT_DATA);
            }
            // Move to the next spot in the shift register
            GPIO_SetBits(GPIOC, SHIFT_CLOCK);
            GPIO_ResetBits(GPIOC, SHIFT_CLOCK);
        }
        if(traffic_light_status == GREEN_LED){
            // If the light is green, shift the array one unit to the right for
each tick
            for(int i = 19; i>0; i--){
                car_traffic[i] = car_traffic[i-1];
            }
        } else {
            // Else, shift cars according to what needs to be done on a
red/yellow light
            for (int i=8; i>0; i--){
                // First 8 cars (18 to 11) stop before the lights
                if(!(car_traffic[i])){
                    // If next car is empty, shift current car to next position
and set current position to empty
```

```
                    car_traffic[i] = car_traffic[i-1];
                    car_traffic[i-1] = 0;
                }
            }
            for (int i=19; i>9; i--){
                // Move cars normally after the lights.
                car_traffic[i] = car_traffic[i-1];
                car_traffic[i-1] = 0;
            }
        }
        // Check if a car is coming and add it to the array
        uint16_t car = 0;
        xQueueReceive(xQueue_Cars, &car, pdMS_TO_TICKS(100));
        car_traffic[0] = 0;
            if(car == 1)
            {
                car_traffic[0] = 1;
            }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

// ###########################################################################
//                  TRAFFIC FLOW TASK AND FUNCTIONS
// ###########################################################################

static void Traffic_Flow_Task( void *pvParameters )
{
    while(1)
    {
        // Use the time scale function to determine the flow rate
        // If the light is red for the maximum amount of time, flow is a gap of 5
between cars
        // If the light is red for the minimum amount of time, flow is a gap of 0
between cars
        uint16_t flow_rate = (uint16_t)(Time_Scale(RED_LED)/1000) - 3;
        xQueueOverwrite(xQueue_Flow, &flow_rate);
    }
}

static uint16_t Get_ADC_Val()
{
    ADC_SoftwareStartConv(ADC1);
```

```
    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
    return ADC_GetConversionValue(ADC1);
}

static uint16_t Time_Scale( uint16_t led )
{
    uint16_t pot_val = Get_ADC_Val();
    if (led == GREEN_LED){
        return pdMS_TO_TICKS(MIN_TIME) + ((pot_val * (pdMS_TO_TICKS(6000))) /
4095);
    }
    else {
        return pdMS_TO_TICKS(MAX_TIME) - ((pot_val * (pdMS_TO_TICKS(6000))) /
4095);
    }
}

// ###########################################################################
//                  TRAFFIC GENERATOR TASK AND FUNCTIONS
// ###########################################################################
static void Traffic_Generator_Task( void *pvParameters )
{
    uint16_t flow;
    uint16_t car = 1;

    while(1)
    {
        xQueuePeek(xQueue_Flow, &flow, 5000);
        xQueueOverwrite(xQueue_Cars, &car);
        vTaskDelay(pdMS_TO_TICKS(500 * flow));
    }
}

// ###########################################################################
//                      ERROR HANDLING FUNCTIONS
// ###########################################################################

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 );

    /* TODO: Setup the clocks, etc. here, if they were not configured before
    main() was called. */
}
```