University of Victoria

Department of Electrical and Computer Engineering

ECE 455 - Real Time Computer Systems

Project 1 - Traffic Light System

Brett Dionello V01026046
Mudit Jaswal V00982906

TABLE OF CONTENTS

INTRODUCTION

The objective of this project is to design and implement a Traffic Light System (TLS) using middleware and FreeRTOS features such as tasks, queues, and software timers. The system simulates vehicle traffic on a one-way, one-lane road with a simplified intersection controlled by a single traffic light. The TLS dynamically adjusts the traffic flow rate using a potentiometer, represents moving cars with LEDs, and controls traffic at the intersection with a traffic light.

FUNCTIONAL REQUIREMENTS

- **Green light**: Cars proceed through the intersection.
- **Yellow light**: Cars prepare to stop.
- **Red light**: Cars must stop before the intersection.
- Traffic generation should be random and proportional to the potentiometer value.
- Vehicles (represented by LEDs) must move at a constant speed (1-3 LED shifts per second), stopping at red lights.
- The **green light** duration must be directly proportional to the traffic flow rate.
- The **red light** duration must be inversely proportional to the traffic flow rate.
- The **yellow light** duration must remain constant.
- At **maximum traffic flow**, cars should appear bumper-to-bumper, and the green light should stay on twice as long as the red light.
- At **minimum traffic flow**, cars should be spaced 5-6 LEDs apart, and the red light should stay on twice as long as the green light.
- Technical requirements include appropriate usage of Tasks, Queues, and Software Timers.

DESIGN SOLUTION

Our design involved breaking the problem down into smaller subtasks. These could be categorized as circuit design, GPIO Initialization, ADC Initialization, Traffic Flow Adjustment Task, Traffic Generation Task LED display, and Traffic Light State Task.

Our design involved breaking the problem down into smaller subtasks. These subtasks can be categorized as circuit design, GPIO Initialization, ADC Initialization, Traffic Flow Adjustment, Traffic Generation, LED display, and Traffic Light State.

To ensure that the system completes the necessary processes in the correct order, periodically and without error, a thorough design process is required. Our overall design process began with creating an early sketch of overall data flow and description of how the queues and different tasks worked together.

Timers

To simulate the traffic lights three separate timers are required and callback functions for handling the transition between light states are needed.

Timer 1 – “Green Done”

- Handle the condition that the green light timer has expired taking the following steps:
    1. Green light off
    2. Amber light on
    3. Start the amber timer
    4. Update queue 2: “Amber”

Timer 2 – “Red Done”

- Handle the condition that the red light timer has expired taking the following steps:
    1. Red light off
    2. Green light on
    3. Calculate green timer duration based on flow
    4. Start the green timer
    5. Update queue 2: “Green”

Timer 3 – “Amber Done”

- Handle the condition that the amber light timer has expired taking the following steps:
    1. Amber light off
    2. Red light on
    3. Calculate the red timer duration based on flow
    4. Start the red timer
    5. Update queue 2: “Red”

Queues

The inter-task communication is facilitated entirely through the use of queues, three queues are used and configured as follows. All tasks are configured with a size of 1.

Queue 1 – Traffic Flow

- Contains the scaled ADC value in range 0-100 for 0-3 volts
- Task 1 “Overwrites”
- Task 2 and timer call back “Peek”
    - Red and Green lights adjust based on flow

Queue 2 – Traffic Light

- Contains the name of the currently activated light (Green, Amber, Red)
- Task 3 “Overwrites”
- Task 4 “Peeks”

Queue 3 – Traffic Generator

- Contains the value of the next traffic state, 0 = No car arrival, 1 = One new car arrival
- Task 2 “Sends”
- Task 4 “Receives”

Scheduling

Using a priority driven task design, tasks 1 and 2 are configured with priority and delays, task 3 being used only to initialize the traffic light timer sequence, and task 4 being queue driven. The scheduling timing diagram can be seen below in Figure 1, the important features of the scheduler are explained in the following points.

- Priority is used to ensure the operations occur in sequence
    1. ADC measures potentiometer
    2. The traffic flow value is generated (car or no car)
    3. The display updates the regular traffic flow

- Task 3 runs once to initialize the first green light
    - The traffic lights are timer based and not scheduled tasks

- Task 4 is constantly checking queue 3 and only runs when it successfully receives
- Pre-emption never occurs due to the task delays and short processing durations

Note: The task duration is exaggerated for demonstration purposes, the tasks will complete their processes almost instantly.

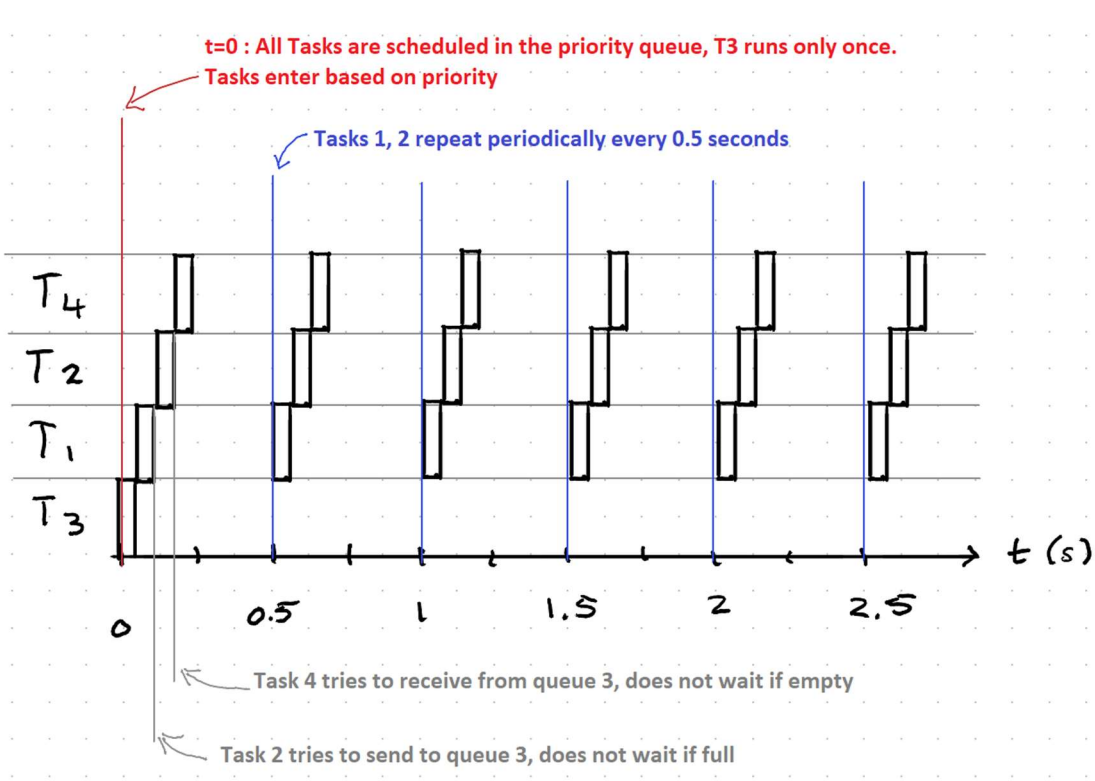


Figure 1: Task scheduling - timing diagram

Tasks:

The necessary tasks are listed below with their relative priorities and delays required for the successful operation of the traffic light system, the tasks will be described further in the following sections

- Task 1 – Flow Adjust
  - Priority = 3 (second highest)
  - Delay = 0.5 seconds

- Task 2 – Traffic Generate
  - Priority = 2
  - Delay = 0.5 seconds

- Task 3 – Light State
  - Priority = 4 (highest)
  - Delay = N/A Runs once to initialize the green light state

- Task – 4 Display
  - Priority = 1 (lowest)
  - Queue event driven

Inter-task Communication

The tasks use the three queues to pass the information required to complete their processes, the tasks use the queues in different ways as depicted in the figures below.

- Queues 1 and 2 use overwrite and peak to transfer the flow and light values between tasks, this method eliminates the synchronization requirements between these tasks and allows the priority driven schedule scheme to determine the system behavior. These queues are essentially uses as global variables, but the data hazards are handled by the scheduling and having only one writing task.

- Queue 3 uses the send and receive method of communication to transmit the traffic generation value which ensures that the display task only makes updates for a new traffic value only one time. The tasks that send and receive through queue 3 do not wait, the sending task runs periodically and the receive task checks constantly.

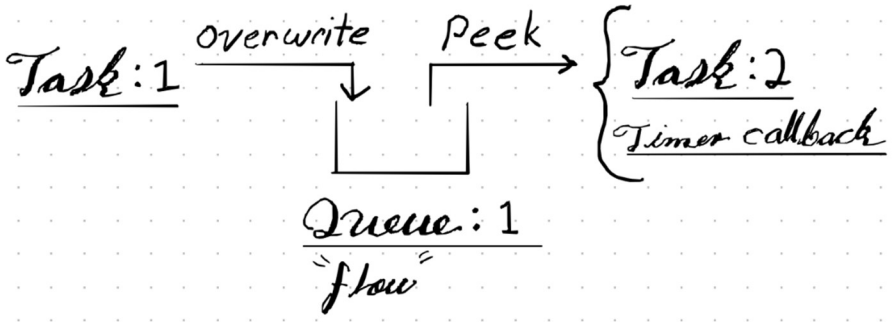


Figure 2: Queue 1 inter-task communication

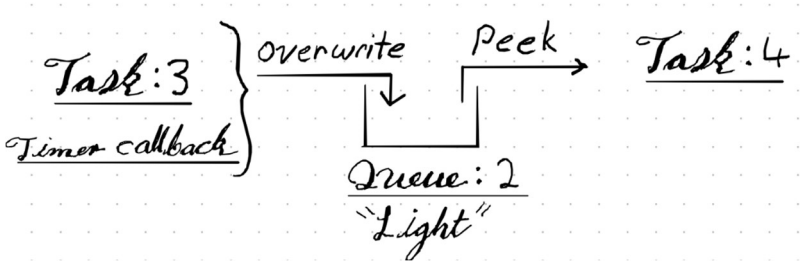


Figure 3: Queue 2 inter-task communication

Note: In Figure 3 task 3 only writes once then the timers write afterwards in regular sequence.

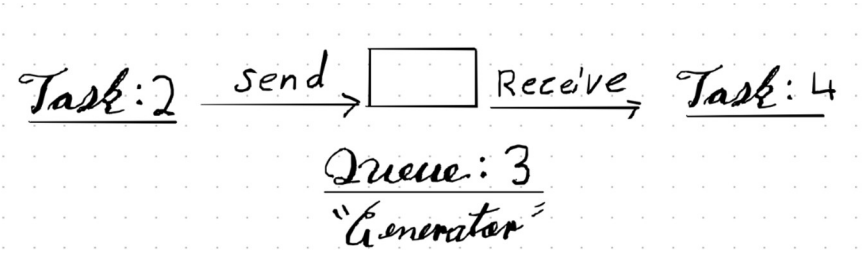


Figure 4: Queue 3 inter-task communication

TASK DESCRIPTIONS

Traffic Flow Adjustment Task

The Traffic_Flow_Adjustment_Task function is responsible for dynamically adjusting the rate if upcoming traffic based on the reading from a potentiometer. This simulates varying traffic conditions by controlling how frequently cars enter.

The system reads an analog input from the potentiometer and that raw ADC value is then converted to a scaled traffic flow rate (0-10). The calculated traffic flow value is then stored in a queue for other tasks to use. Traffic density is dynamically adjusted based on real time user input.

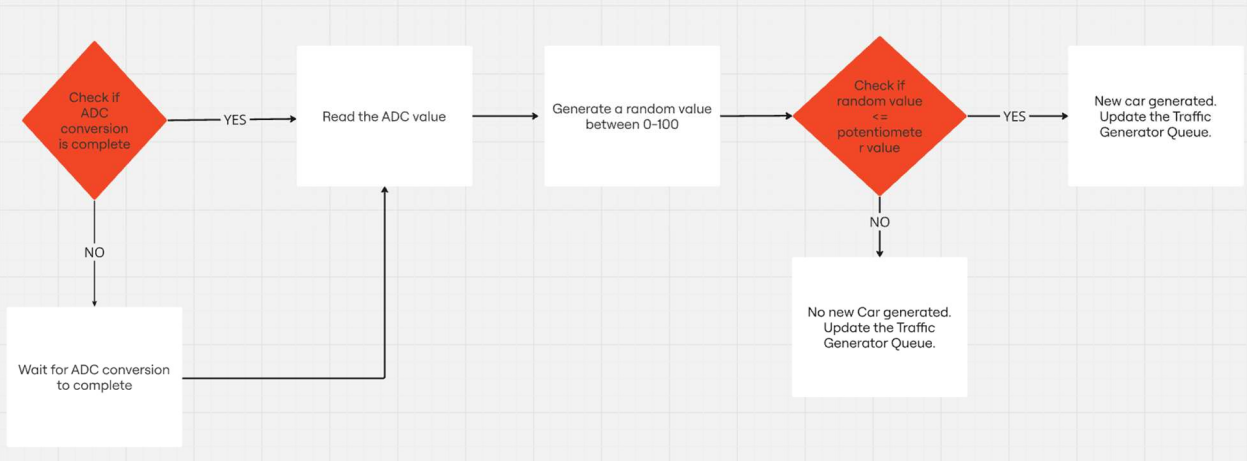


Figure 5: Flow chart for flow adjustment

Description: The Traffic_Generator_Task function is responsible for creating new cars based on the current traffic flow rate. It determines whether a new car should be generated by using a random number and comparing it with the traffic flow value received from the Traffic_Flow_Adjustment_Task function.

- The system reads the current flow traffic value from the queue and generates a random value between 0-100.
- Then, this random value is compared to the potentiometer value and if the random value is less than or equal to the potentiometer value, new car is generated, and the traffic generator queue is updated. If not, no new car is generated, and we still update the traffic generator queue.

Traffic Light State Task

The Traffic_Light_State_Task function is responsible for managing the state of the traffic light in the system.

- Initializes the traffic light system with a green light.
- Starts the timer for the green light to control the light's duration.
- Updates the traffic light queue with the current light state.

Traffic Light Timers

We created three timers, one to represent each traffic light color. We then also created three callback functions. Each callback function is executed when a particular light duration expires.

- **Green light expired:** The function xGreenDone is triggered when green light timer expires. The system transitions from green to amber, the traffic light queue is updated, and the amber light timer is started (constant).

- **Amber light expired:** The function xAmberDone is triggered when amber light timer expires. The system transitions from amber to red. We retrieve the current traffic flow from traffic flow queue and then calculate the red-light duration such that, if flow is high, red-light duration decreases and vice-versa. The traffic queue is updated then and red-light timer starts.

- **Red light expired:** The function xRedDone is triggered when red light timer expires. The system transitions from red to green. We retrieve the current traffic flow from traffic flow queue and then calculate the green ight duration such that, if flow is high, green light duration increases and vice-versa. The traffic queue is updated then and green light timer starts.

Display task:

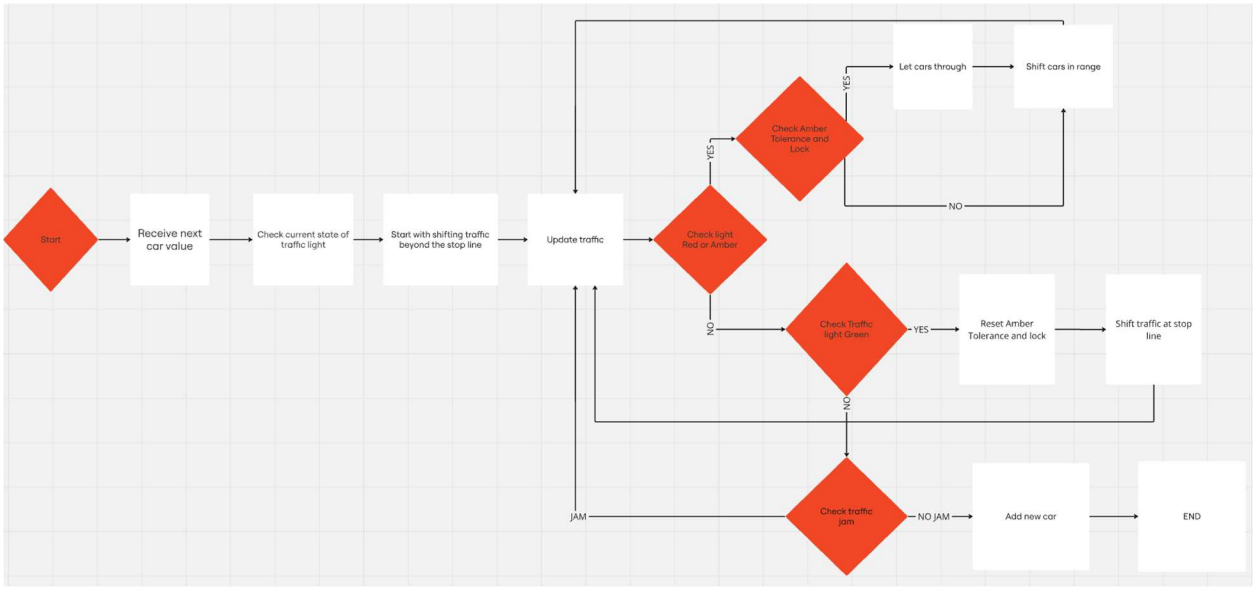


Figure 6: Flow chart of display algorithm

Description: The System_Display_Task function is responsible for managing and representing the car position values. A car position can be on or off, representing either a car at that position or no car at that position. A flow chart describing the algorithm is shown above. The flow chart is also explained below describing all the keys steps:

- **Initialization:** The new_car holds the value of an incoming vehicle (1 if a car arrives, 0 otherwise). Traffic_pos[] maintains the position of vehicles in a simulated traffic lane. current_light_state stores the latest traffic light state. Amber_tolerance allows vehicles to move for a short period after the light turns amber. Lock ensures that vehicles do not move a red light unless conditions allow.

- **Receiving and shifting positions:** The function receives a new car from the traffic queue generator and also retrieves the current light state from the traffic light queue. The display is updated using the update_traffic_display function which sets the data pin high if a car is in the spot.

- **Handling red and amber lights:** Vehicles must stop when the light is red. If the light is amber and amber_tolerance is greater than zero, some vehicles may still pass. The lock mechanism prevents uncontrolled movement when light is red.

- **Handling green light and adding a new car:** When the light turns green, amber_tolerance is reset to 2 and cars are allowed to move forward. If the road is clear (stopped_position < 1), a new car is added, otherwise, the new car is placed at position 0 only if no traffic jam exists. The display updates accordingly.

IMPLEMENTATION

Hardware Implementation

The hardware components consist of LEDs, resistors, shift registers, a breadboard, jumper cables, and an STM32F0 development board. The STM32 GPIO configuration is explained in the next section, once the software is configured the STM32 ports are verified on the electronic instruments before connecting to the breadboard.

The suggested STM32 port connections are listed in the table below and provided in the project lab manual.

Table 1: STM32 Pinout

STM32 PIN	Signal Name
PC0	Red Light
PC1	Amber Light
PC2	Green Light
PC3	Potentiometer input
PC6	Shift Register - Data
PC7	Shift Register - Clock
PC8	Shift Register - Reset

Using the data sheet for the 74HC164 the pinout and signal levels can be found, and the required connections can be made.

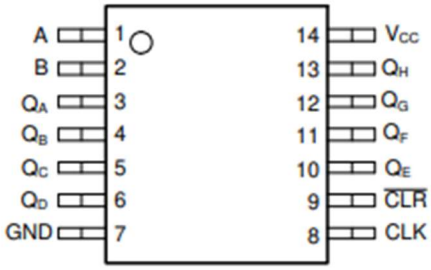


Figure 7: 74HC164 Shift Register Pinout

Using an organized wiring layout troubleshooting is made easy and the hardware implementation was wired as shown in Figure 8 and was tested using benchtop electronic instruments before it was connected to the STM32.

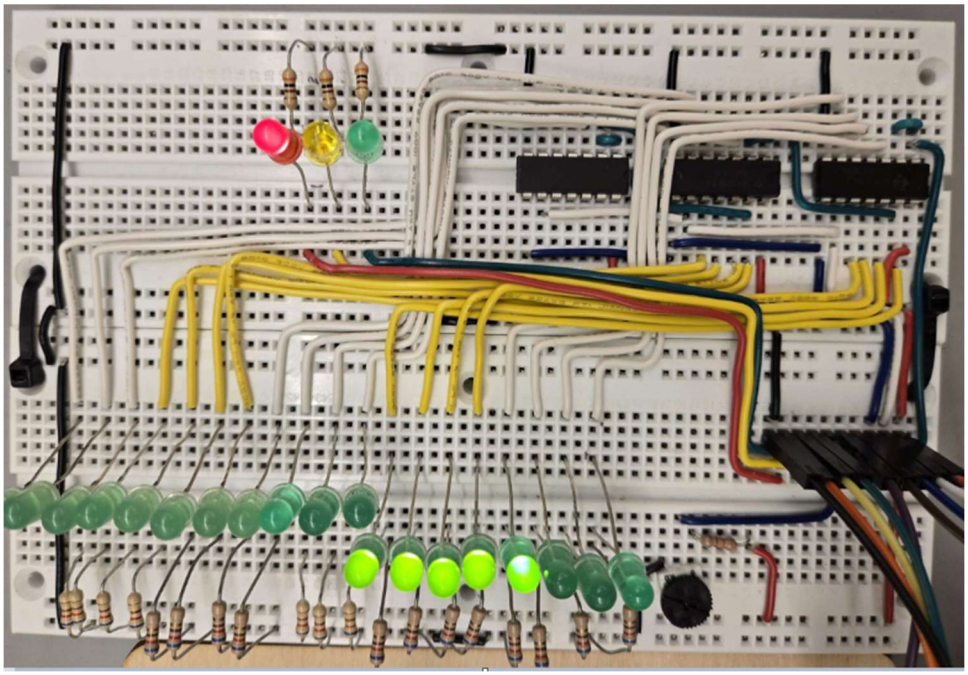


Figure 8: Complete hardware implementation

GPIO Initialization

Our GPIO_out_init function initializes specific GPIO pins and we enabled all the GPIO clocks first before calling this function. We configured GPIOC for the shift registers and also for the different pins we were supposed to use. We followed the information provided in the slides to initialize this part.

```
void GPIO_out_init( void ){
        GPIO_InitTypeDef GPIOC_InitStruct;
        GPIOC_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
        GPIOC_InitStruct.GPIO_OType = GPIO_OType_PP;
        GPIOC_InitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
        GPIOC_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
        GPIOC_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
        //GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
        GPIO_Init(GPIOC, &GPIOC_InitStruct);
```

Figure 9: GPIO Initialization code snippet

ADC Initialization

Similar to the GPIO initialization, the my_adc_init function is responsible for setting up the Analog-to-Digital Converter (ADC) on the microcontroller. The ADC allows the system to read analog signals (e.g., from sensors) and convert them into digital values for processing. We enabled ADC and set it to channel 13.

```
void my_adc_init(){
        GPIO_InitTypeDef GPIOC_InitStruct;
        GPIOC_InitStruct.GPIO_Mode = GPIO_Mode_AN;
        GPIOC_InitStruct.GPIO_OType = GPIO_OType_PP;
        GPIOC_InitStruct.GPIO_Pin = GPIO_Pin_3;
        GPIOC_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
        GPIOC_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
        //GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
        GPIO_Init(GPIOC, &GPIOC_InitStruct);

        ADC_InitTypeDef ADC_InitStruct;
        ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;
        ADC_InitStruct.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
        ADC_InitStruct.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
        ADC_InitStruct.ADC_NbrOfConversion = 1;
        ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
        ADC_InitStruct.ADC_ScanConvMode = DISABLE;

        // void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct)
        ADC_Init(ADC1, &ADC_InitStruct);
        ADC_Cmd (ADC1, ENABLE);
        // void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)
        ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
}
```

Figure 10: ADC Initialization code snippet

FreeRTOS Configuration

A FreeRTOS application is configured using the FreeRTOSConfig.h header file. In this project we used the default configuration with the important settings as shown in Figure 11.

```
#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
```

Figure 11: FreeRTOS Configuration

Task Initialization

The tasks are created with the following commands and the relative priorities can be noted as described in previous sections.

```
xTaskCreate(Traffic_Flow_Adjustment_Task, "Traffic_Flow_Adjustment", configMINIMAL_STACK_SIZE, NULL, 3, NULL);
xTaskCreate(Traffic_Light_State_Task, "Traffic_Light_State", configMINIMAL_STACK_SIZE, NULL, 4, NULL);
xTaskCreate(Traffic_Generator_Task, "Traffic_Generator", configMINIMAL_STACK_SIZE, NULL, 2, NULL);
xTaskCreate(System_Display_Task, "System_Display", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
```

Figure 12: Task initalization code snippet

Queue Initialization

The queues required for the system are created and registered with the following commands, the size and data type of each can be noted as described in previous sections.

```
xQueue_TRAFFIC_FLOW = xQueueCreate(1, sizeof(uint32_t));
xQueue_TRAFFIC_LIGHT = xQueueCreate(1, sizeof(uint16_t));
xQueue_TRAFFIC_GENERATOR = xQueueCreate(1, sizeof(uint8_t));

vQueueAddToRegistry(xQueue_TRAFFIC_FLOW, "xQueue_TRAFFIC_FLOW" );
vQueueAddToRegistry( xQueue_TRAFFIC_LIGHT, "xQueue_TRAFFIC_LIGHT" );
vQueueAddToRegistry( xQueue_TRAFFIC_GENERATOR, "xQueue_TRAFFIC_GENERATOR" );
```

Figure 13: Queue initialization code snippet

Timers

Initialization

```
xTim_GREEN = xTimerCreate("TIM1", pdMS_TO_TICKS(12000), pdFALSE, 0, xGreenDone);
xTim_RED = xTimerCreate("TIM2", pdMS_TO_TICKS(5000), pdFALSE, 0, xRedDone);
xTim_AMBER = xTimerCreate("TIM3", pdMS_TO_TICKS(4000), pdFALSE, 0, xAmberDone);
```

Figure 14: Timer initialization code snippet

Timer Callback Function

As explained earlier, the timers require callback functions to simulate the TLS and achieve the required behavior, an example of one timer callback function is displayed below. Note the timer duration adjustment, the LED control as well as the queue access function calls.

```
void xRedDone ( TimerHandle_t xTim){
    uint32_t flow;
    uint32_t timer_ms;
        printf("TIMER Callback - Red expired:\n");
        printf("\t\t Red = OFF, Green = ON\n");
        xQueuePeek( xQueue_TRAFFIC_FLOW, &flow, TICKS );
    // Set the traffic light to AMBER
        uint16_t green = GREEN;
        GPIO_ResetBits(GPIOC, RED);       //reset RED
        GPIO_SetBits(GPIOC, GREEN);
        xQueueOverwrite(xQueue_TRAFFIC_LIGHT,(void *) &green);
        // Red light range from ~ 18000ms to 12000ms for flow from 10 to 0
        timer_ms = 10000 + 100*flow;
        xTimerChangePeriod( xTim_GREEN, pdMS_TO_TICKS(timer_ms), 0 );
        printf("\t\t Green Timer =%u ms\n", timer_ms);
        printf("\t Light Queue Updated:\n");
        printf("\t\t Light = %s \n\n", "Green");
}
```

Figure 15: Timer callback function code snippet

Timer duration update

The calculations for the red and green light on-time duration are showed in the below images with the comments listing the relative times for the min and max traffic flow ranges.

```
// Red light range from ~ 6000ms to 12000ms
timer_ms = 20000 - 100*flow;
xTimerChangePeriod( xTim_RED, pdMS_TO_TICKS(timer_ms), 0); //AS FLOW INCREASES PERIOD DECREASES
// Red light range from ~ 18000ms to 12000ms for flow from 10 to 0
timer_ms = 10000 + 100*flow;
xTimerChangePeriod( xTim_GREEN, pdMS_TO_TICKS(timer_ms), 0 );
```

Figure 16: Timer duration adjustment code snippet

Testing

To confirm correct system operation and to troubleshoot issues we used the tools available in the Atollic TrueSTUDIO IDE such as the SWV console print statements and the FreeRTOS task list and Queues panels. Print statements were placed in every task and timer callback to indicate the process id, relevant values, and queue interactions. The task list and queue panel were used to verify the scheduler was behaving as designed by indicating the state transitions of each FreeRTOS component at each breakpoint.

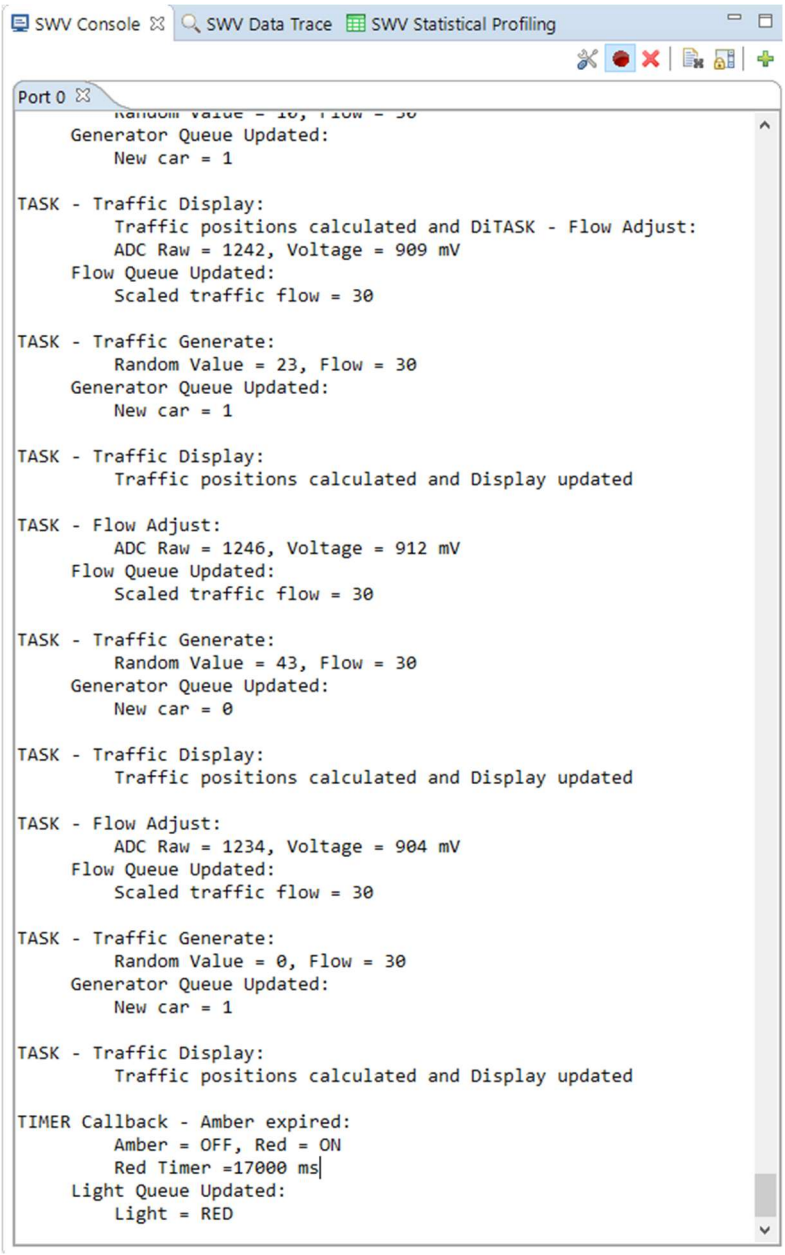
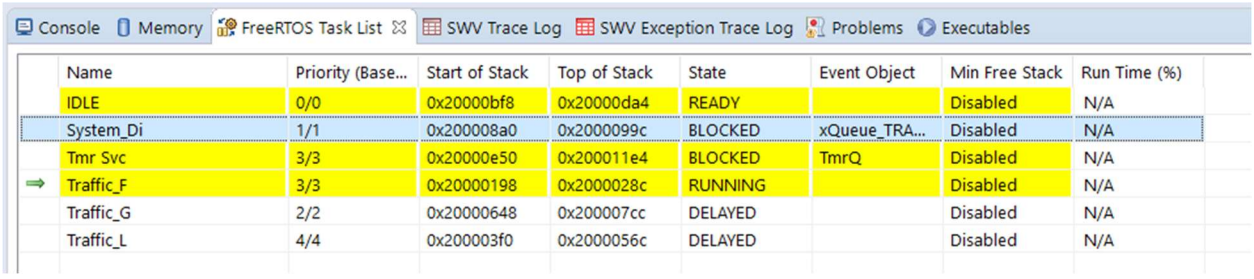


Figure 17: SWD console output



Name	Priority (Base...	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
IDLE	0/0	0x20000bf8	0x20000da4	READY		Disabled	N/A
System_Di	1/1	0x200008a0	0x2000099c	BLOCKED	xQueue_TRA...	Disabled	N/A
Tmr Svc	3/3	0x20000e50	0x200011e4	BLOCKED	TmrQ	Disabled	N/A
Traffic_F	3/3	0x20000198	0x2000028c	RUNNING		Disabled	N/A
Traffic_G	2/2	0x20000648	0x200007cc	DELAYED		Disabled	N/A
Traffic_L	4/4	0x200003f0	0x2000056c	DELAYED		Disabled	N/A

Figure 18: FreeRTOS Task panel

Name	Address	Max Length	Item Size	Current Length	# Waiting
TmrQ	0x20000b20	5	12	0	0
xQueue_TRAFFIC_FLOW	0x200000a8	1	4	1	0
xQueue_TRAFFIC_GENERATOR	0x20000148	1	1	0	0
xQueue_TRAFFIC_LIGHT	0x200000f8	1	2	1	0

Figure 19: FreeRTOS Queues panel

CONCLUSION

The purpose of this project was to get hands-on experience with real time systems by creating a simulated intersection using LED's to represent cars and a potentiometer to control the flow of traffic. Since one of our team members had prior experience with hardware modules, getting started was not too difficult, but integrating everything still required a lot of effort.

At times, it was hard to tell whether issues were caused by faulty wiring or timing problems. We spent a significant amount of time troubleshooting and despite the challenges, we managed to get the system running smoothly with some time to spare. In the end, it was a rewarding project as we were able to see everything come together and function as intended.

APPENDIX A: CODE

```
/*-----------------------------------------------------------*/
#define GREEN   GPIO_Pin_2
#define RED     GPIO_Pin_0
#define AMBER   GPIO_Pin_1
#define DATA    GPIO_Pin_6
#define CLOCK   GPIO_Pin_7
#define RESET   GPIO_Pin_8
#define TICKS pdMS_TO_TICKS(100)
#define CAR_SPOTS 19
#define STOP_POSITION 9 //Position where cars should stop if light turns yellow
#define YELLOW_LIGHT_POSITION 6 //Position where light turns yellow
static void prvSetupHardware( void );

xQueueHandle xQueue_TRAFFIC_FLOW = 0;
xQueueHandle xQueue_TRAFFIC_LIGHT = 0;
xQueueHandle xQueue_TRAFFIC_GENERATOR = 0;

TimerHandle_t xTim_GREEN=0;
TimerHandle_t xTim_RED=0;
TimerHandle_t xTim_AMBER =0;

void GPIO_out_init( void );
void my_adc_init( void );
uint16_t my_adc_convert( void );
void initialize_traffic_lights( void );

void Traffic_Flow_Adjustment_Task   ( void *pvParameters );
void Traffic_Light_State_Task(void *pvParameters);
void Traffic_Generator_Task(void *pvParameters);
void System_Display_Task(void *pvParameters);
void xAmberDone ( TimerHandle_t xTim);
void xGreenDone ( TimerHandle_t xTim);
void xRedDone ( TimerHandle_t xTim);
```

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

int main(void){
    initialize_traffic_lights();
    //queues for traffic system components
    xQueue_TRAFFIC_FLOW = xQueueCreate(1, sizeof(uint32_t));
    xQueue_TRAFFIC_LIGHT = xQueueCreate(1, sizeof(uint16_t));
    xQueue_TRAFFIC_GENERATOR = xQueueCreate(1, sizeof(uint8_t));
    prvSetupHardware();

    //check if all queues were created successfully
if (xQueue_TRAFFIC_FLOW && xQueue_TRAFFIC_LIGHT && xQueue_TRAFFIC_GENERATOR){
    xTaskCreate(Traffic_Flow_Adjustment_Task, "Traffic_Flow_Adjustment", configMINIMAL_STACK_SIZE, NULL,
3, NULL);
    xTaskCreate(Traffic_Light_State_Task, "Traffic_Light_State", configMINIMAL_STACK_SIZE, NULL, 4, NULL);
    xTaskCreate(Traffic_Generator_Task, "Traffic_Generator", configMINIMAL_STACK_SIZE, NULL, 2, NULL);
    xTaskCreate(System_Display_Task, "System_Display", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    //expires after 5 sec, pdFALSE means timer will expire once and will not restart again automatically
    xTim_GREEN = xTimerCreate("TIM1", pdMS_TO_TICKS(12000), pdFALSE, 0, xGreenDone);
    xTim_RED = xTimerCreate("TIM2", pdMS_TO_TICKS(5000), pdFALSE, 0, xRedDone);
    xTim_AMBER = xTimerCreate("TIM3", pdMS_TO_TICKS(4000), pdFALSE, 0, xAmberDone);

    vQueueAddToRegistry(xQueue_TRAFFIC_FLOW, "xQueue_TRAFFIC_FLOW" );
    vQueueAddToRegistry( xQueue_TRAFFIC_LIGHT, "xQueue_TRAFFIC_LIGHT" );
    vQueueAddToRegistry( xQueue_TRAFFIC_GENERATOR, "xQueue_TRAFFIC_GENERATOR" );
}
    /* Start the tasks and timer running. */
    vTaskStartScheduler();
    return 0;
}

/*-----------------------------------------------------------*/
void initialize_traffic_lights( void ){
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
    GPIO_out_init();
    my_adc_init();
}
```

```
void GPIO_out_init( void ){
    GPIO_InitTypeDef GPIOC_InitStruct;
    GPIOC_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
    GPIOC_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIOC_InitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_6 | GPIO_Pin_7 |
GPIO_Pin_8;
    GPIOC_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIOC_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    //GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
    GPIO_Init(GPIOC, &GPIOC_InitStruct);
}
void my_adc_init(){
    GPIO_InitTypeDef GPIOC_InitStruct;
    GPIOC_InitStruct.GPIO_Mode = GPIO_Mode_AN;
    GPIOC_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIOC_InitStruct.GPIO_Pin = GPIO_Pin_3;
    GPIOC_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIOC_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    //GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
    GPIO_Init(GPIOC, &GPIOC_InitStruct);

    ADC_InitTypeDef ADC_InitStruct;
    ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;
    ADC_InitStruct.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
    ADC_InitStruct.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStruct.ADC_NbrOfConversion = 1;
    ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStruct.ADC_ScanConvMode = DISABLE;

    // void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct)
    ADC_Init(ADC1, &ADC_InitStruct);
    ADC_Cmd (ADC1, ENABLE);
    // void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t
ADC_SampleTime)
    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
}
uint16_t my_adc_convert( void ) {
    uint16_t sample_data;
    ADC_SoftwareStartConv(ADC1);
```

```
    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
    sample_data = ADC_GetConversionValue(ADC1);
    return sample_data;
}
void Traffic_Light_State_Task( void *pvParameters ){
    uint16_t LIGHT= GREEN;
    printf("TASK - Light State:\n");
    printf("\t\t Green light initalized\n");
    xTimerStart( xTim_GREEN, 0);
    xQueueOverwrite(xQueue_TRAFFIC_LIGHT, (void *)&LIGHT);
    GPIO_SetBits(GPIOC, GREEN);
    printf("\t Light Queue Updated:\n");
    printf("\t\t Light = %s \n\n", "GREEN");
    for( ;; ){
        vTaskDelay(100 * TICKS );
    }
}
void Traffic_Flow_Adjustment_Task(void *pvParameters){
    uint32_t res_val;
    uint32_t flow_val;
    uint32_t voltage;
    for(;;){
        printf("TASK - Flow Adjust:\n");
        //scale the potentiometer value to a range of 0-10
        res_val = my_adc_convert();
        flow_val = (res_val/409)*10;
        voltage = (float)res_val*0.7326;
        printf("\t\t ADC Raw = %u, Voltage = %u mV \n", res_val, voltage);
        //save new potentiometer value in queue
        //not sure if xQueueSend would be better
        xQueueOverwrite(xQueue_TRAFFIC_FLOW,(void*) &flow_val);
        printf("\t Flow Queue Updated:\n");
        printf("\t\t Scaled traffic flow = %u \n\n", flow_val);
        vTaskDelay(TICKS*5);
    }
}
void Traffic_Generator_Task(void *pvParameters){
    uint32_t flow_value = 0;
    uint16_t random  = 0;
```

```
    uint8_t new_car = 0;
    for(;;){
        printf("TASK - Traffic Generate:\n");
        xQueuePeek(xQueue_TRAFFIC_FLOW, &flow_value, 1000);
        //generate random number between 0-100
        random = rand()%101;
        printf("\t\t Random Value = %u, Flow = %u \n", random, flow_value);
        //if random number is greater than pot val then generate a new car if not do not generate a new
car
        if(random <= flow_value) { //Create new car
            //Send car to new queue
            new_car = 1;
            xQueueSend(xQueue_TRAFFIC_GENERATOR, &new_car, 1000);
        } else{
            new_car = 0;
            xQueueSend(xQueue_TRAFFIC_GENERATOR, &new_car, 1000);
        }
        printf("\t Generator Queue Updated:\n");
        printf("\t\t New car = %u \n\n", new_car);
        vTaskDelay(TICKS*5);
    }
}
void update_traffic_display(int traffic_pos[CAR_SPOTS]) {
    //Reset all the car spots
    GPIO_ResetBits(GPIOC, RESET);
    GPIO_SetBits(GPIOC, RESET);
    for(int i=CAR_SPOTS-1; i>=0; i--) {
        //If there is a car in the spot, set the data pin high
        if(traffic_pos[i]){
            GPIO_SetBits(GPIOC, DATA);
        }
        GPIO_SetBits(GPIOC, CLOCK);
        GPIO_ResetBits(GPIOC, DATA | CLOCK);
    }
}
/*-----------------------------------------------------------*/
/*
 * System Display Task
 * Description: task that updates the system display based on the traffic light state and the car queue.
```

```
 * Input: void *pvParameters - pointer to parameters
 * Output: void
 */
void System_Display_Task( void *pvParameters ){
    uint8_t new_car = 0;
    int traffic_pos[19] = {};
    uint16_t current_light_state;
    int stopped_position = STOP_POSITION-1;
    int amber_tolerance = 2;
    int lock = 1;

    for(;;){
        if(xQueueReceive(xQueue_TRAFFIC_GENERATOR, &new_car, 1000)) { // Update traffic
            xQueuePeek( xQueue_TRAFFIC_LIGHT, &current_light_state, TICKS ); // get light state
            printf("TASK - Traffic Display:\n");
            printf("\t\t Traffic positions calculated and Display updated\n\n");
            // Start with traffic beyond stop line, always shift
            for(int i=CAR_SPOTS-1; i>=STOP_POSITION; i--) {
                traffic_pos[i] = traffic_pos[i-1];
            }
            update_traffic_display(traffic_pos);
            // Traffic past yellow light position can go if light is yellow
            if((current_light_state == RED) || (current_light_state == AMBER)) {
                // Let a few cars through
                if (amber_tolerance <= 0 && lock){
                    if(stopped_position >= 1){
                        stopped_position -=1;
                        traffic_pos[STOP_POSITION-1]=0; // no more cares get through
                    }
                    lock = 0;
                }
                //Shift all cars by 1 in range of stopped_pos to 1
                for (int i=stopped_position; i>=1; i--) {
                    traffic_pos[i] = traffic_pos[i-1];
                    if(traffic_pos[stopped_position] == 1) {
                        lock = 1;
                    }
                }
                update_traffic_display(traffic_pos);
```

```
                amber_tolerance -=1;
            }
            // Traffic at stop line, shift when traffic light is green
            if(current_light_state == GREEN){
                amber_tolerance = 2;
                stopped_position = STOP_POSITION-1;
                lock = 1;
                for(int i=STOP_POSITION; i>=1; i--) {
                    traffic_pos[i] = traffic_pos[i-1];
                }
                update_traffic_display(traffic_pos);
                //traffic_pos[0] = 0;
            }
            // Add new car value if no traffic jam and new car value is 1
            if(stopped_position < 1){
                traffic_pos[0] = 1;
            }
            else{
                traffic_pos[0] = new_car;
            }
            update_traffic_display(traffic_pos);
        }
    }
}
void xGreenDone ( TimerHandle_t xTim){
    // Set the traffic light to AMBER
    uint16_t amber = AMBER;
    printf("TIMER Callback - Green expired:\n");
    printf("\t\t Green = OFF, Amber = ON\n");
    GPIO_ResetBits(GPIOC, GREEN);       //reset green
    GPIO_SetBits(GPIOC, AMBER);
    xQueueOverwrite(xQueue_TRAFFIC_LIGHT,(void *) &amber);
    printf("\t Light Queue Updated:\n");
    printf("\t\t Light = %s \n\n", "AMBER");
    xTimerStart( xTim_AMBER, 0);
}
//inversely proportional to potentiometer value
void xAmberDone ( TimerHandle_t xTim){
    uint32_t flow;
```

```
    uint32_t timer_ms;
    printf("TIMER Callback - Amber expired:\n");
    printf("\t\t Amber = OFF, Red = ON\n");
    xQueuePeek( xQueue_TRAFFIC_FLOW, &flow, TICKS );
    // Set the traffic light to AMBER
    uint16_t red= RED;
    GPIO_ResetBits(GPIOC, AMBER);       //reset amber
    GPIO_SetBits(GPIOC, RED);
    xQueueOverwrite(xQueue_TRAFFIC_LIGHT,(void *) &red);
    // Red light range from ~ 6000ms to 12000ms
    timer_ms = 20000 - 100*flow;
    xTimerChangePeriod( xTim_RED, pdMS_TO_TICKS(timer_ms), 0); //AS FLOW INCREASES PERIOD DECREASES
    printf("\t\t Red Timer =%u ms\n", timer_ms);
    printf("\t Light Queue Updated:\n");
    printf("\t\t Light = %s \n\n", "RED");
}
//proportional to potentiometer value
//BaseType_t xStatus =
void xRedDone ( TimerHandle_t xTim){
    uint32_t flow;
    uint32_t timer_ms;
    printf("TIMER Callback - Red expired:\n");
    printf("\t\t Red = OFF, Green = ON\n");
    xQueuePeek( xQueue_TRAFFIC_FLOW, &flow, TICKS );
    // Set the traffic light to AMBER
    uint16_t green = GREEN;
    GPIO_ResetBits(GPIOC, RED);       //reset RED
    GPIO_SetBits(GPIOC, GREEN);
    xQueueOverwrite(xQueue_TRAFFIC_LIGHT,(void *) &green);
    // Red light range from ~ 18000ms to 12000ms for flow from 10 to 0
    timer_ms = 10000 + 100*flow;
    xTimerChangePeriod( xTim_GREEN, pdMS_TO_TICKS(timer_ms), 0 );
    printf("\t\t Green Timer =%u ms\n", timer_ms);
    printf("\t Light Queue Updated:\n");
    printf("\t\t Light = %s \n\n", "Green");
}
```