UNIVERSITY OF VICTORIA

Department of Electrical and Computer Engineering

ECE 455:

REAL TIME COMPUTER SYSTEMS DESIGN

PROJECT

PROJECT 1 : TRAFFIC LIGHT SYSTEM

REPORT SUBMITTED ON: 2023-03-09

NAME : Berkan Ozturk (V00892651)

Reece Pretorius (V00880300)

TABLE OF CONTENTS

Introduction

In this lab experiment, we design and implement a Traffic Light System (TLS) using middleware, and FreeRTOS features. The TLS simulates car traffic on a one-way, one-lane road with a simplified intersection that has a single traffic light. The TLS consists of three components:

1. The traffic flow adjustment potentiometer

The potentiometer is used to dynamically adjust the traffic flow rate at run-time.

2. LEDs representing cars

The LEDs are used to represent the position of cars at different points in time. If an LED is on, it means a car is on the road at that LED's position; if an LED is off it means there is no car at that LED's position

3. Traffic Light

A traffic light with three LEDs (one green, one yellow, one red) is used to control the traffic at the intersection.

Design Solution

Software

Atollic TrueSTUDIO was used to develop this system. Throughout this project, we have used FreeRTOS tasks, queues and timers. Below, we explain the tasks we have created for our system.

Tasks

As per the lab manual and our initial design document, the following 3 tasks were implemented as the main components of the TLS.

Traffic_Flow_Adjustment_Task_Potentiometer

This task adjusts the flow rate of the traffic using the potentiometer resistance value converted to a percentage range (0-98%) because of how we implemented this task there is no 100% flow-rate as the scaling value we used was the closest we could get to 100 without going over. If we were to improve this in future work, it could be scaled with more intervals and any value over 100% could be scaled down to 100%, then it would produce more accurate results. The adc_value is split into 7 increments which are then scaled by multiplying by 14, giving us a range of (0-98) and this range is used to set the flow rate in the xQueue_FLOW queue.

```
void Traffic_Flow_Adjustment_Task_Potentiometer(void *pvParameters)
    uint16_t adc_value = 0;
    uint16_t increment_value = 0;
    uint16_t current_increment = 0;
    uint16_t increment_flag;
    int flow_rate_percent = 0;

    while(1) {
        adc_value = ADC_Start_Conversion();
        increment_value = adc_value/512;

        increment_flag = abs(increment_value - current_increment);

        if(increment_flag !=  0)
        {
            current_increment = increment_value;
            flow_rate_percent = increment_value * 14; // scale
increment value between range (0-100%)

            if(xQueueOverwrite(xQueue_FLOW, &flow_rate_percent)) {
// Overwrite existing flow in queue
                vTaskDelay(pdMS_TO_TICKS(100));
            }
        }
    }
```

Initialize_Traffic_Lights_Task

Initializes the traffic light timers by starting the green timer, the traffic lights and their timers are handled by a single handler function **`traffic_light_timer_handler`** with individual callback functions and id's assigned to each of the three traffic light colors. When this task runs it starts the timer for the green traffic light and writes the state of the light to xQueue_TRAFFICLIGHTS, the rest of the traffic light functionality is handled by the timers and callback handler function, which are explained in the "**Queues and Timers**" section.

```
void Initialize_Traffic_Lights_Task(void *pvParameters) {
    xTimerStart(xTimer_LIGHT_GRN, 0);

    uint16_t LIGHT = GREEN_PIN;
    xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);

    while(1) { vTaskDelay(100 * pdMS_TO_TICKS(100)); }
}
```

The following two tasks work together to generate and display the traffic/cars using the LEDs.

Traffic_Generator_Task

This task adds a car to the xQueue_TRAFFICGEN queue (pushes a 1 onto the queue to indicate there is a car ready) every set interval using vTaskDelay with a variable time based on the flow rate from the potentiometer, the display task checks the same queue to receive new cars when available to control the LEDs.

```
void Traffic_Generator_Task(void *pvParameters) {
    BaseType_t xStatus;
    int flow;
    int car;

    flow = car = 1;

    while(1) {
        xStatus = xQueuePeek(xQueue_TRAFFICFLOW, &flow, pdMS_TO_TICKS(100));

        if (xStatus == pdPASS) {
            int scaled = flow * SCALE_VALUE;
```

```
            xQueueOverwrite(xQueue_TRAFFICGEN, &car);
            vTaskDelay(pdMS_TO_TICKS(4000 - scaled));
        }
    }
}
```

Display_Task

Additional details can be seen in the source code for this task as it is a bit lengthy, we will instead discuss important segments from the task itself.
At the end of this task the xQueue_TRAFFICGEN queue is checked and if there is a car available then it is removed from the queue and added to the cars array.

There are two cases to handle to display the traffic/cars moving left to right using the LEDs. The first case is when the traffic light is green, in this case we just shift the array values along the entire array by 1 per interval ([0,1,0,0,1,0,0,0,1, …, 0]) each index of this array represents an LED on the breadboard and a value of 1 indicates that there is a car at that index, shifting this array then determines which LEDs to turn on and which ones to turn off. We use a for loop to set and reset each LED before checking and shifting the array if needed. For the second case (when the traffic light is not green) will only shift the first 8 indexes of the array (these are the lights before the traffic light) up to index 7 and any gaps in traffic (index with 0 value) will be filled as cars are shifted only in this range and stopped at the traffic light. The other half of the array (other side of the traffic light, index 8-19) are shifted as normal. and again the set/reset loop is run to actually display this with the LEDs.

```
for (int i = 19; i > 0; i--) {
    if (cars[i] == 1) { GPIO_SetBits(GPIOC, DATA_PIN); }
    else { GPIO_ResetBits(GPIOC, DATA_PIN); }

    GPIO_SetBits(GPIOC, CLOCK_PIN);
    GPIO_ResetBits(GPIOC, CLOCK_PIN);
}

if (LIGHT == GREEN_PIN) {
    for (int i = 19; i > 0; i--) { cars[i] = cars[i-1]; }
}
else {
    for (int i = 8; i > 0; i--) {
        if (cars[i] == 0) {
```

```
            cars[i] = cars[i-1];
            cars[i-1] = 0;
        }
    }
    for (int i = 19; i > 9; i--) {
        cars[i] = cars[i-1];
        cars[i-1] = 0;
    }
}
```

Additional helper functions that were implemented are discussed below.

ADC_Start_Conversion

This function simply returns the converted ADC value in a range of (0-4096) and was developed using the slides provided in the lab. This value is used by the `Traffic_Flow_Adjustment_Task_Potentiometer` task to adjust the traffic flow.

```
uint16_t ADC_Start_Conversion(void) {
    uint16_t converted_data;
    // Start ADC Conversion
    ADC_SoftwareStartConv(ADC1);
    // Wait until conversion is finished
    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
    // Get the value
    converted_data = ADC_GetConversionValue(ADC1);
    return converted_data;
}
```

traffic_light_timer_handler

This is the handler function mentioned in the `Initialize_Traffic_Lights_Task` section earlier. This handler function has two main jobs, it determines which traffic light should be turned on next and it also adjusts the length of time that the red and green traffic lights stay lit. All three callback functions for the traffic light timers call this function with the id of the next light in the sequence (green->yellow->red) whenever the timer finishes. The next light to be pushed to the xQueue_TRAFFICLIGHTS queue has its timer started. If the next light in the

sequence is green or red then the value of the potentiometer is grabbed from the xQueue_TRAFFICFLOW queue and the timer is adjusted using xTimerChangePeriod. The SCALE_VALUE is used to scale the potentiometer flow percentage value to a value we subtract or add to the timer's initial time, this is done so that the green traffic light is directly proportional to the flow rate of traffic, and it is the inverse for the red traffic light.

```
void traffic_light_timer_handler(TimerHandle_t xTimer, int color_id) {
    int flow;
    int scaled;

    //Get flow value from queue.
    BaseType_t xStatus = xQueuePeek(xQueue_TRAFFICFLOW, &flow,
pdMS_TO_TICKS(100));

    if(color_id == 1) {
        xTimerStart(xTimer_LIGHT_YEL, 0);
        uint16_t LIGHT = YELLOW_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }

    if(color_id == 0 && xStatus == pdPASS) {
        scaled = flow * SCALE_VALUE;

        xTimerChangePeriod(xTimer_LIGHT_RED, pdMS_TO_TICKS(8000 - scaled),
0);
        uint16_t LIGHT = RED_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }

    if(color_id == 2 && xStatus == pdPASS) {
        scaled = flow * SCALE_VALUE;

        xTimerChangePeriod(xTimer_LIGHT_GRN, pdMS_TO_TICKS(4000 + scaled),
0);
        uint16_t LIGHT = GREEN_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }
}
```

Queues and Timers

We did not deviate from our initial design in regards to how many queues we ended up using. We used 3 queues, xQueue_TRAFFICFLOW, xQueueTRAFFICLIGHTS, and xQueueTRAFFICGEN. In order the queues were used to store and update the converted percentage value from the potentiometer for the traffic flow, xQueueTRAFFICLIGHTS stores the current state of the traffic light (whether it is red, green or yellow) and finally xQueueTRAFFICGEN which holds a single value to indicate that a car has been generated and is ready to be added to the cars array and is also delayed longer or shorter based on the traffic flow value. We also used 3 timers, one for each traffic light (red, green, yellow) to determine how long each light should be turned on before switching to the next.

Design Document

The design document below was our first design before we even started implementing our program. We had planned to use 4 tasks, 3 queues, and 3 timers. In addition, a draft drawing of how we thought we would build the system on the breadboard can be found on our design document.

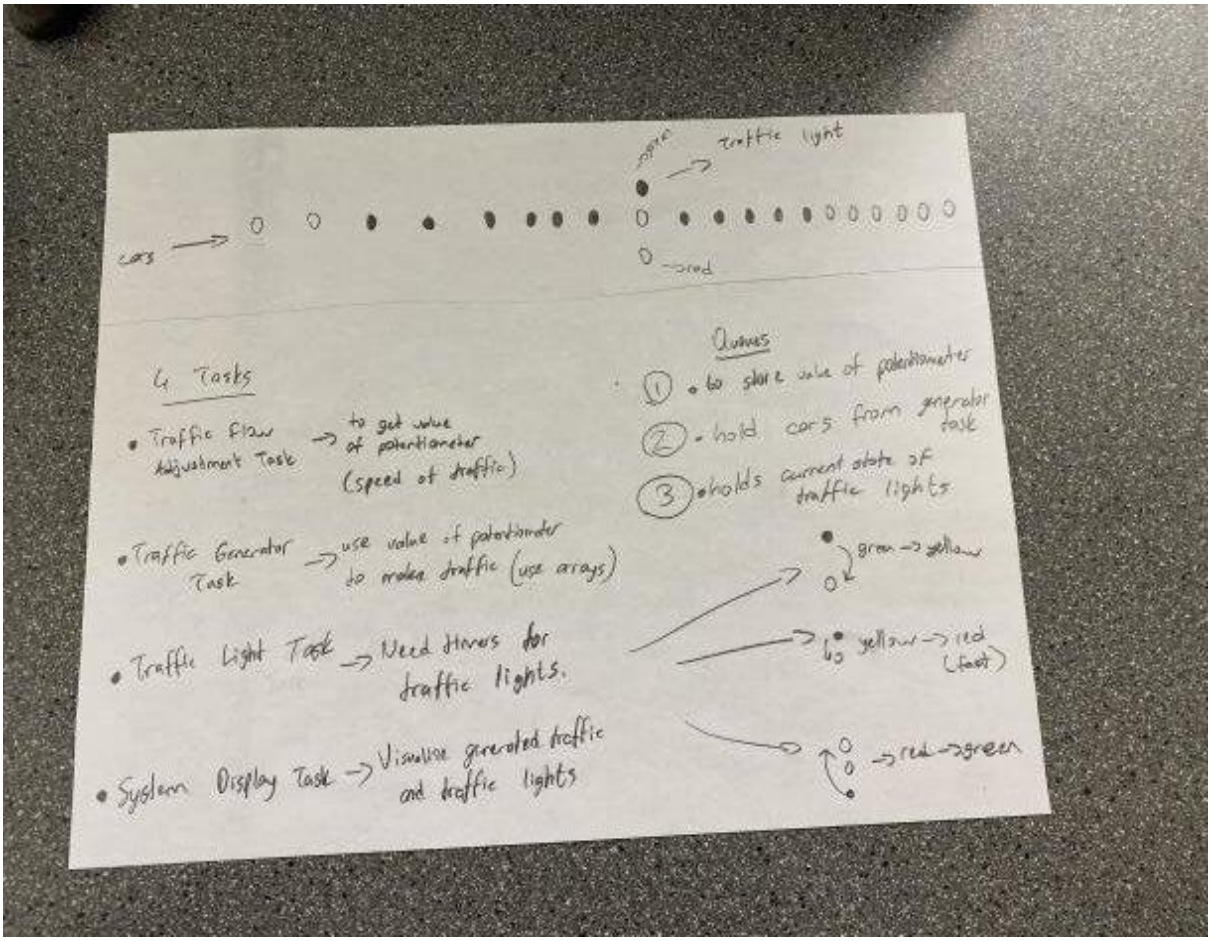

Image 1. Design Document

The image below is our final design. Comparing our initial design and the final design was not too different from each other after we completed the project.

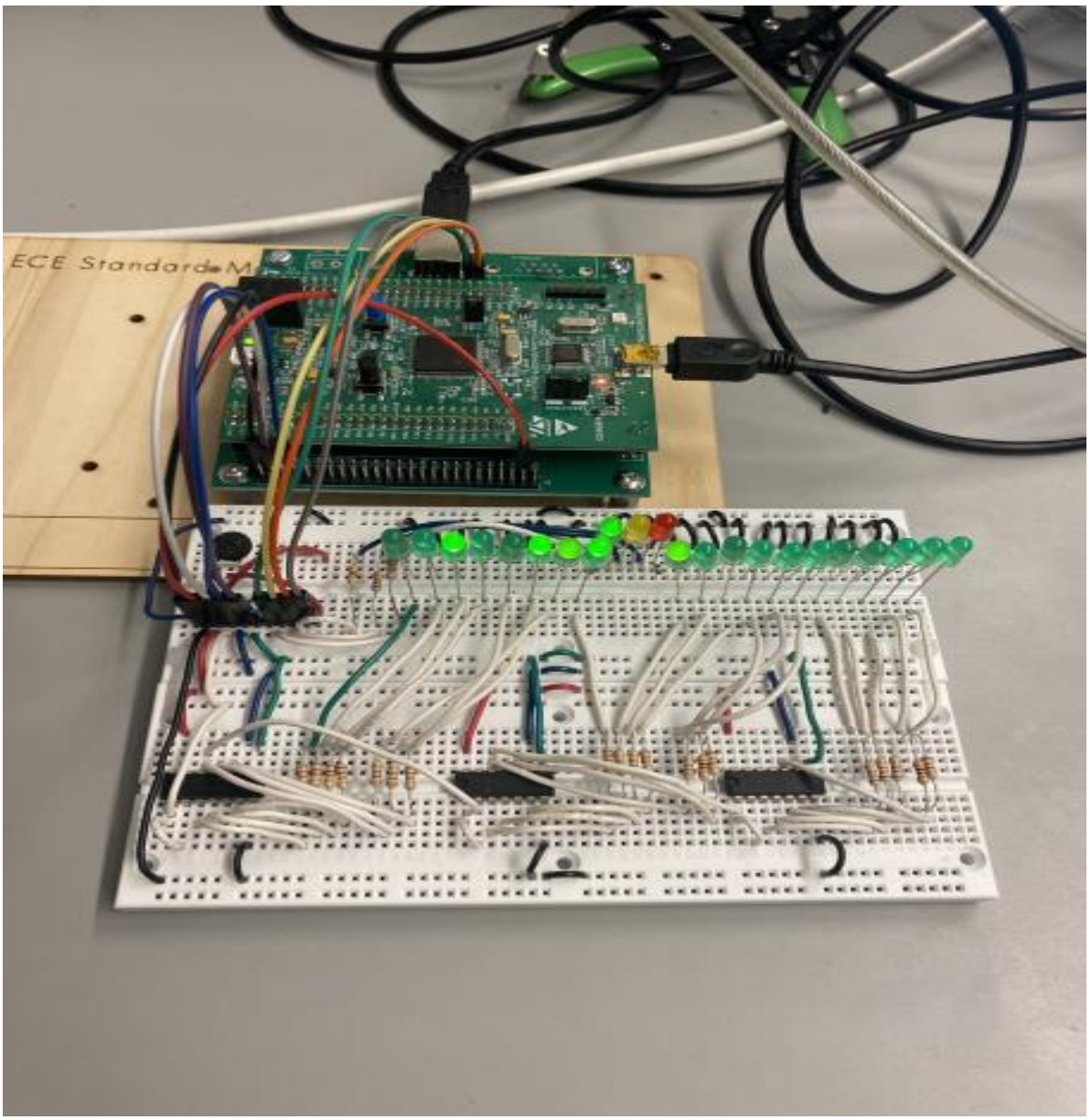

Image 2. Final Design

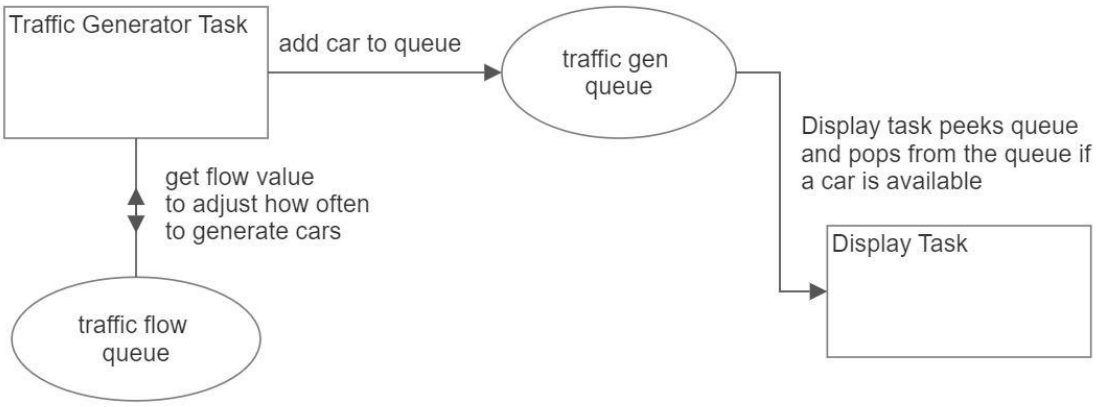

Image 3. Traffic Generation Flow

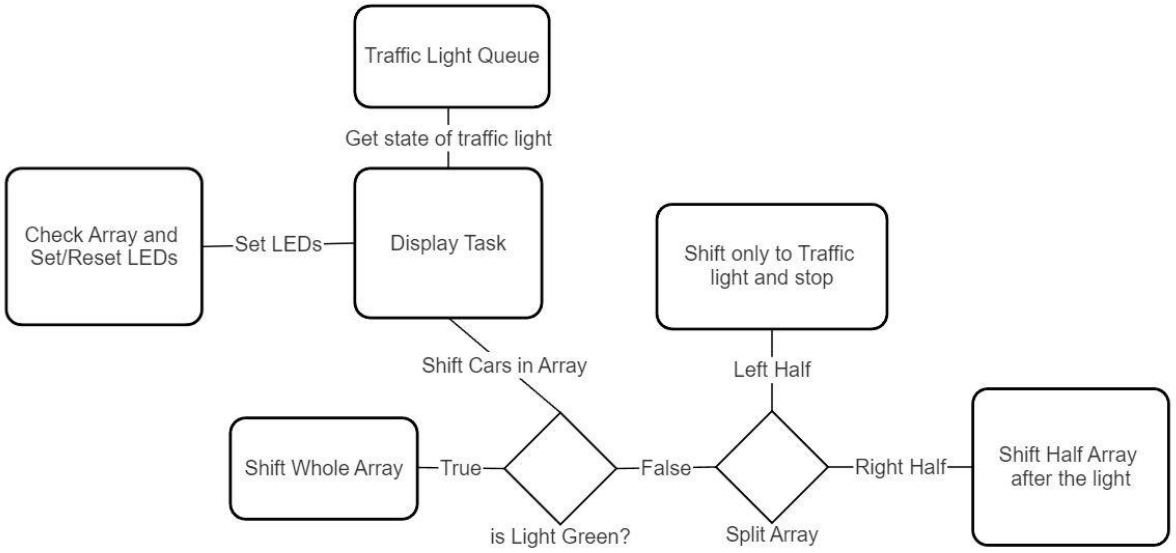

Image 4. Display Task Flow

Discussion

Comparing our final design to our design document, we did not have many differences. We still used 4 different tasks and timers for traffic lights and multiple queues to hold values. On the other hand, the design on the breadboard was a little different than the one on the design document. Our final design seemed more complex.

Middleware is a software that defines the interactions between application software and the device drivers. For this project, middleware was written to configure the STM32's GPIO pins and the ADC to allow application code to interact with hardware components. Below is a list of all the GPIO pins we have used:

- PC0 for Red Light
- PC1 for Yellow (Amber) Light
- PC2 for Green Light
- PC8 for Shift Register Reset
- PC7 for Shift Register Clock
- PC6 for Shift Register Data
- PC3 for Potentiometer

Limitations and Possible Improvements

For limitations, we had a problem scaling the value we receive from the potentiometer. We could not scale the flow value on a 0-100 range, instead we scaled it to a 0-98 range, and that was the best range we could come up with.

For potential improvement, we could add additional lanes, or make the traffic 2-way traffic.

Summary

In the summary, we have designed and implemented a traffic light system. We benefited from the FreeRTOS features as we have used tasks, queues, and timers for our project. At the end, we made a demo of our project using Atollic TrueSTUDIO.

First, we set the value of potentiometer to the lowest possible value, and executed our program. The traffic light stayed on the red more than it stayed on green. When we set the value of the potentiometer to the highest possible value, there were no gaps between cars, and the green light stayed on more than it stayed on red.

Appendix (with source code)

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

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

//Defining GPIO pins
#define POTEN_PIN GPIO_Pin_3
#define RED_PIN GPIO_Pin_0
#define YELLOW_PIN GPIO_Pin_1
#define GREEN_PIN GPIO_Pin_2
#define DATA_PIN GPIO_Pin_6
#define CLOCK_PIN GPIO_Pin_7
#define RESET_PIN GPIO_Pin_8

#define SCALE_VALUE 50

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

//TASKS
void Traffic_Flow_Adjustment_Task_Potentiometer (void *pvParameters);
void Initialize_Traffic_Lights_Task (void *pvParameters);
void Traffic_Generator_Task (void *pvParameters);
void Display_Task (void *pvParameters);
```

```c
//INITS
void Init_GPIOC (void);
void Init_ADC (void);

//HELPERS
uint16_t ADC_Start_Conversion(void);

//CALLBACKS
void vCallbackGreen (TimerHandle_t xTimer);
void vCallbackYellow (TimerHandle_t xTimer);
void vCallbackRed (TimerHandle_t xTimer);

//QUEUES + TIMERS
xQueueHandle xQueue_TRAFFICFLOW = 0;
xQueueHandle xQueue_TRAFFICLIGHTS = 0;
xQueueHandle xQueue_TRAFFICGEN = 0;
TimerHandle_t xTimer_LIGHT_GRN  = 0;
TimerHandle_t xTimer_LIGHT_YEL = 0;
TimerHandle_t xTimer_LIGHT_RED  = 0;

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

int main(void) {
    Init_GPIOC();
    Init_ADC();

    xQueue_TRAFFICFLOW  = xQueueCreate(1, sizeof(int));
    xQueue_TRAFFICLIGHTS = xQueueCreate(1, sizeof(uint16_t));
    xQueue_TRAFFICGEN = xQueueCreate(1, sizeof(int));

    if(xQueue_TRAFFICFLOW != NULL   && xQueue_TRAFFICLIGHTS != NULL &&
xQueue_TRAFFICGEN != NULL) {
        xTaskCreate(
            Traffic_Flow_Adjustment_Task_Potentiometer,
            "AdjustTrafficFlow",
            configMINIMAL_STACK_SIZE,
            NULL, 1, NULL
        );
        xTaskCreate(
            Initialize_Traffic_Lights_Task,
            "TrafficLightStart",
            configMINIMAL_STACK_SIZE,
            NULL, 1, NULL
        );
        xTaskCreate(
            Traffic_Generator_Task,
            "TrafficGen",
            configMINIMAL_STACK_SIZE,
```

```
            NULL, 1, NULL
        );
        xTaskCreate(
            Display_Task,
            "Display",
            configMINIMAL_STACK_SIZE,
            NULL, 1, NULL
        );

        xTimer_LIGHT_GRN = xTimerCreate(
                "GreenLightTimer",
                pdMS_TO_TICKS(4000),
                pdFALSE,
                0,
                vCallbackGreen);
        xTimer_LIGHT_YEL =  xTimerCreate(
                "YellowLightTimer",
                pdMS_TO_TICKS(2000),
                pdFALSE,
                0,
                vCallbackYellow);
        xTimer_LIGHT_RED = xTimerCreate(
                "RedLightTimer",
                pdMS_TO_TICKS(4000),
                pdFALSE,
                0,
                vCallbackRed);

        vTaskStartScheduler();
    }
    return 0;
}

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

void Init_GPIOC() {
    // GPIO Led Clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    // Initializing
    GPIO_InitTypeDef GPIO_InitStructure;

    // Enable GPIO pins for Traffic lights
    GPIO_InitStructure.GPIO_Pin = RED_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
```

```
    GPIO_InitStructure.GPIO_Pin = YELLOW_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GREEN_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    // Enable GPIO pins for potentiometer
    GPIO_InitStructure.GPIO_Pin = POTEN_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    // Enable GPIO pins for DATA, CLOCK and RESET
    GPIO_InitStructure.GPIO_Pin = DATA_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = CLOCK_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = RESET_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

}

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

void Init_ADC() {

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    //Init ADC1
    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;
```

```
    ADC_Init(ADC1, &ADC_InitStruct);
    ADC_Cmd(ADC1, ENABLE);

    //Select input channel for ADC1
    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1,
ADC_SampleTime_144Cycles);
}

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

uint16_t ADC_Start_Conversion(void) {
    uint16_t converted_data;
    // Start ADC Conversion
    ADC_SoftwareStartConv(ADC1);
    // Wait until conversion is finish
    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
    // Get the value
    converted_data = ADC_GetConversionValue(ADC1);
    return converted_data;
}

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

void Traffic_Flow_Adjustment_Task_Potentiometer(void *pvParameters) {
    uint16_t adc_value = 0;
    uint16_t increment_value = 0;
    uint16_t current_increment = 0;

    uint16_t increment_flag;

    int flow_rate_percent = 0;

    while(1) {
        adc_value = ADC_Start_Conversion();
        increment_value = adc_value/512;

        increment_flag = abs(increment_value - current_increment);

        if(increment_flag !=  0)
        {
            current_increment = increment_value;
            flow_rate_percent = increment_value * 14; // scale increment
value between range (0-100%)

            if(xQueueOverwrite(xQueue_FLOW, &flow_rate_percent)) { //
Overwrite existing flow in queue
                vTaskDelay(pdMS_TO_TICKS(100));
            }
```

```
        }
    }
}

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

//Task to start timer for green traffic light. Switching is handled by
callback functions and timer handler.
void Initialize_Traffic_Lights_Task(void *pvParameters) {
    xTimerStart(xTimer_LIGHT_GRN, 0);

    uint16_t LIGHT = GREEN_PIN;
    xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);

    while(1) {
        vTaskDelay(100 * pdMS_TO_TICKS(100));
    }
}

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

//Handler for adding lights to the traffic lights queue, also adjusts timers
for red and green lights based on traffic flow rate.
void traffic_light_timer_handler(TimerHandle_t xTimer, int color_id) {
    int flow;
    int scaled;

    //Get flow value from queue.
    BaseType_t xStatus = xQueuePeek(xQueue_TRAFFICFLOW, &flow,
pdMS_TO_TICKS(100));

    if(color_id == 1) {
        xTimerStart(xTimer_LIGHT_YEL, 0);
        uint16_t LIGHT = YELLOW_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }

    if(color_id == 0 && xStatus == pdPASS) {
        scaled = flow * SCALE_VALUE;

        xTimerChangePeriod(xTimer_LIGHT_RED, pdMS_TO_TICKS(8000 - scaled),
0);
        uint16_t LIGHT = RED_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }

    if(color_id == 2 && xStatus == pdPASS) {
        scaled = flow * SCALE_VALUE;
```

```
        xTimerChangePeriod(xTimer_LIGHT_GRN, pdMS_TO_TICKS(4000 + scaled),
0);
        uint16_t LIGHT = GREEN_PIN;
        xQueueOverwrite(xQueue_TRAFFICLIGHTS, &LIGHT);
    }
}

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

void Traffic_Generator_Task(void *pvParameters) {
    BaseType_t xStatus;
    int flow;
    int car;

    flow = car = 1;

    while(1) {
        xStatus = xQueuePeek(xQueue_TRAFFICFLOW, &flow, pdMS_TO_TICKS(100));

        if (xStatus == pdPASS) {
            int scaled = flow * SCALE_VALUE;

            xQueueOverwrite(xQueue_TRAFFICGEN, &car);
            vTaskDelay(pdMS_TO_TICKS(4000 - scaled));
        }
    }
}

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

void Display_Task(void *pvParameters) {
    BaseType_t xStatus_LIGHT;
    BaseType_t xStatus_GENERATOR;
    uint16_t LIGHT;

    int car;
    int cars[19] = {0};

    GPIO_SetBits(GPIOC, RESET_PIN);

    while(1) {
        xStatus_LIGHT = xQueuePeek(xQueue_TRAFFICLIGHTS, &LIGHT,
pdMS_TO_TICKS(100));

        if (xStatus_LIGHT == pdTRUE) {
            GPIO_ResetBits(GPIOC, RED_PIN);
            GPIO_ResetBits(GPIOC, YELLOW_PIN);
```

```
            GPIO_ResetBits(GPIOC, GREEN_PIN);
            GPIO_SetBits(GPIOC, LIGHT);
        }

        for (int i = 19; i > 0; i--) {
            if (cars[i] == 1) { GPIO_SetBits(GPIOC, DATA_PIN); }
            else { GPIO_ResetBits(GPIOC, DATA_PIN); }

            GPIO_SetBits(GPIOC, CLOCK_PIN);
            GPIO_ResetBits(GPIOC, CLOCK_PIN);
        }

        // When light is green, we shift all car traffic LEDs to the right.
        if (LIGHT == GREEN_PIN) {
            for (int i = 19; i > 0; i--) { cars[i] = cars[i-1]; }
        }

        // If the light is Yellow or Red, we have two cases for shifting the
car LEDs.
        else {
            // FIRST CASE: Shifting only the cars(LEDs) in the left half of
the array(before the traffic light)
            // it will stop at the 8th LED (simulating the car stopping at
the light)
            for (int i = 8; i > 0; i--) {
                if (cars[i] == 0) {
                    cars[i] = cars[i-1];
                    cars[i-1] = 0;
                }
            }

            // SECOND CASE: Shifting the LEDs to the right(after) the
traffic light.
            // any cars(LEDs) that have passed the yellow/red light will
continue to shift as normal.
            for (int i = 19; i > 9; i--) {
                cars[i] = cars[i-1];
                cars[i-1] = 0;
            }
        }

        // Check the Traffic Generator Queue for newly generated cars.
        xStatus_GENERATOR = xQueueReceive(xQueue_TRAFFICGEN, &car,
pdMS_TO_TICKS(100));

        // If there is a car in the queue then add it to the cars array.
        if (xStatus_GENERATOR == pdTRUE && car == 1) { cars[0] = 1; }
        vTaskDelay( 5 * pdMS_TO_TICKS(100) );
```

```
    }

}

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

//Callback functions start the timers for the next light in line. (yellow
after green, red after yellow, and green after red)
void vCallbackGreen (TimerHandle_t xTimer) {
    traffic_light_timer_handler(xTimer, 1); // pass yellow pin to handler.
}
void vCallbackYellow (TimerHandle_t xTimer) {
    traffic_light_timer_handler(xTimer, 0); // pass red pin to handler.
}
void vCallbackRed (TimerHandle_t xTimer) {
    traffic_light_timer_handler(xTimer, 2); // pass green pin to handler.
}

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

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

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

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

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

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

void vApplicationIdleHook( void )
{
    volatile size_t xFreeStackSpace;

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

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

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

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

References

1 - ECE 455 Lab Manual and other resources under announcements on Brightspace

2 - Mastering the FreeRTOS Real Time Kernel - a Hands on Tutorial Guide