



Traffic Light System Report

Amaan Makhani

March 24, 2021

B02



1. Introduction

This project was to design and implement a Traffic Light System. This Traffic Light System was aimed to provide us hands on experience with various FreeRTOS features. Specifically, this project allowed us to use tasks, queues, and software timers. Due to the virtual nature of these labs the circuit was implemented for us and an interface was given to control the system. The implemented system communicated with the software via middleware created using device drivers.

The Traffic Light System is shown in figure1 below. As seen in figure 1 there is a set of LEDs representing the traffic light, and a row of LEDs to represent traffic. This is aimed to recreate a one-way, one-lane road. To adjust the flow of traffic the potentiometer value can be increased which will result in more cars being produced. If a LED within the lane row is on this indicates a car is present in this location. More detailed requirements are outlined in section 3.1.

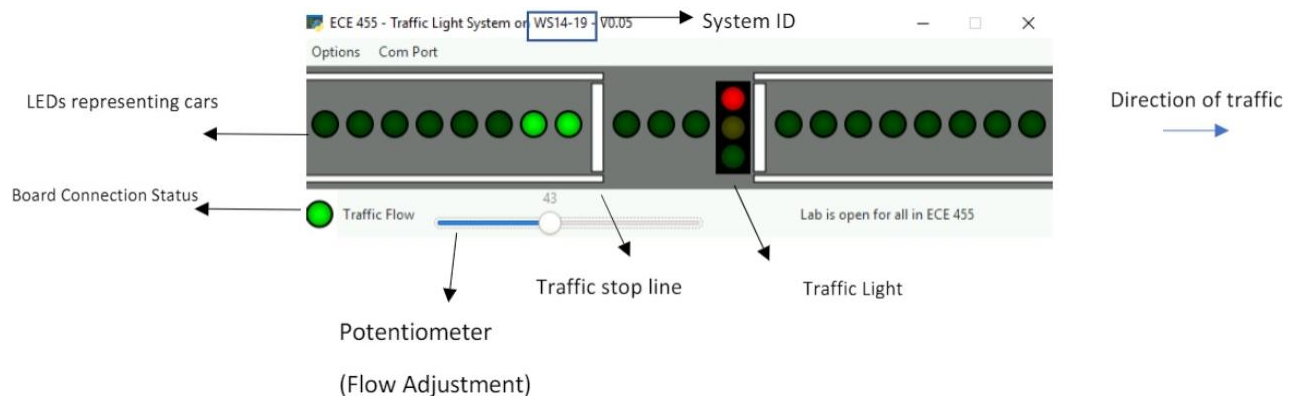


Figure 1: Traffic Light System Interface [1]

2. Design Document

As seen in the below figures. There are 4 main tasks in the program. Two queues are used for these tasks to communicate. The ADC queue stores the latest ADC measurement which is used to manipulate the traffic generator and the timing of the lights. The request queue stores requests that modify the application appearance this includes changing lights, adding cars, and progressing cars. The request queue serves to separate GUI requests from the backend logic. Both the traffic light task and the traffic generator tasks are implemented as timer call backs as each task occurs periodically. This preliminary plan was adapted well to the environment when implemented and remained the same as the final solution.

Blocks for a set amount of time before preforming it's periodic task.

Blocks on the reading of a queue

Implemented as a software timer callback function.

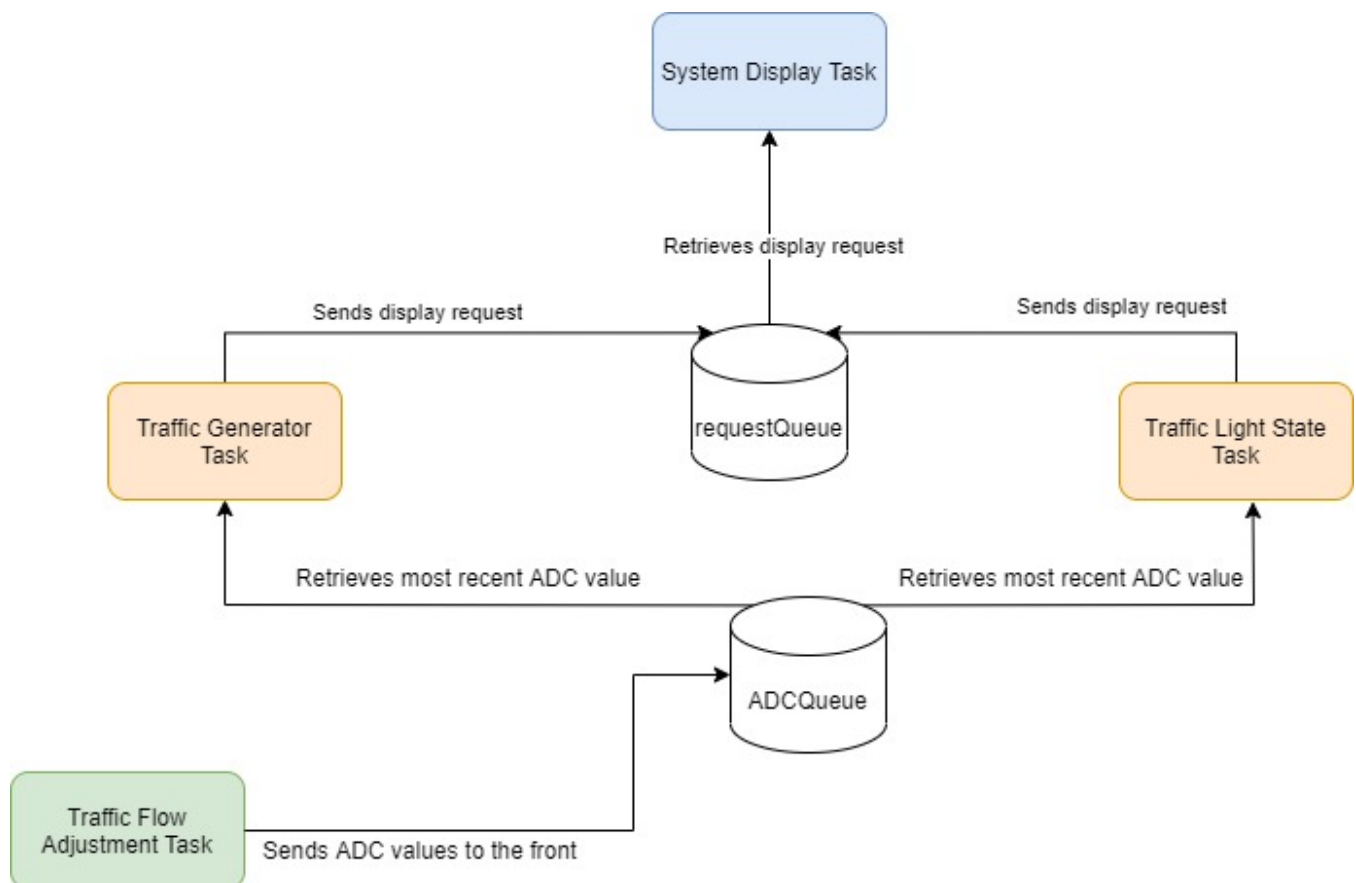


Figure 2: Design document

3. Discussion and Design Solution

This section further details the subsets of the solution and discusses the implementation of the solution as a whole.

3.1. System Overview

To further refine the systems abilities the following requirements were to be met [1]:

- The traffic must be generated randomly and at a rate that is directly proportional to the resistance of the potentiometer.
- All cars must “move” at a constant speed of approximately 1-3 LEDs per second (unless stopped at a red light).
- Cars can only proceed through the intersection when the light is green and must stop at the traffic stop line if the light is yellow or red.
- The duration of the green traffic light must be directly proportional to the traffic flow rate.
- The duration of the red traffic light must be inversely proportional to the traffic flow rate.
- The duration of the yellow traffic light must be constant.
- When the traffic flow is at its maximum setting:
 - There should be no gaps between new cars appearing on the road (i.e. bumper-to-bumper traffic).
 - The traffic light must stay green approximately twice as long as it stays red.
- When the traffic flow is at its minimum setting:
 - Traffic should be created with a gap of approximately 5 or 6 LEDs between cars.
 - The traffic light must stay red approximately twice as long as it stays green.

The following FreeRTOS constraints were also given [1]:

- Tasks must be used to control code execution.
- Queues must be used for inter-task communications. Global variables may NOT be used for this purpose.
- Software timers must be used to control the state of the traffic lights.

3.2. Middleware Description

Middleware is software that defines the interaction between the software and the device drivers. The software has the following stack configuration and uses the middleware as the figure below depicts.

Traffic Light System Report

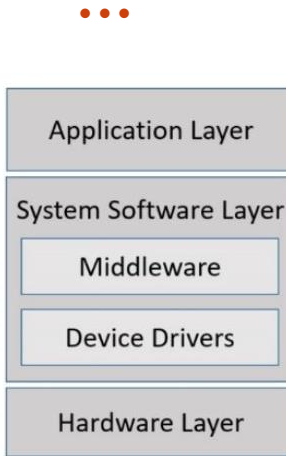


Figure 3: System model [1]

Due to the virtual nature of these labs the hardware was set up and we were given defined pins to use. These pins can be seen in the figure below.

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

Figure 4: Port pin configuration [1]

The middleware written used existing APIs to communicate with registers. For the Traffic Light System project, the following middleware was written:

- Middleware configure the STM32's GPIO pins and the ADC:

Before making use of the GPIO or ADC as specified in the pin table above we must first initialize it as follows:

- Enable AHB1 clock
- Set up pins 0, 1, and 2
 - Set the pins above to output pins
 - Set the pins to push and pull for the correct response
 - Set no pull ups
- Set up the shift register using pins 6, 7, and 8
 - Set the pins above to output pins
 - Set the pins to push and pull for the correct response
 - Set no pull ups
- Set up pin 3 for the potentiometer
 - Set the mode to an input
 - Set no pull ups
- For GPIO pins a high speed slew rate was used as recommended
- Enable APB2 clock



- Setup the ADC
 - Disable continuous conversions
 - Set alignment left
 - Set resolution to 8 bits
 - Disable external triggers
 - Set up the channel in this case channel 13
 - Setup the desired cycles

This setup can be seen in the code below.

```
void initGPIOandADC(void)
{
    // Enable GPIOC clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    // Initialize GPIO for the traffic light
    GPIO_InitTypeDef TrafficLight_GPIO_Struct;
    // Using pins 0, 1, and 2
    // Red, amber, and green respectively
    TrafficLight_GPIO_Struct.GPIO_Pin = 0b111;
    TrafficLight_GPIO_Struct.GPIO_Mode = GPIO_Mode_OUT;
    TrafficLight_GPIO_Struct.GPIO_OType = GPIO_OType_PP;
    TrafficLight_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &TrafficLight_GPIO_Struct);

    // Initialize GPIO for the shift register
    GPIO_InitTypeDef ShiftReg_GPIO_Struct;
    // Using pins 6, 7, and 8
    ShiftReg_GPIO_Struct.GPIO_Pin = 0b11100000;
    ShiftReg_GPIO_Struct.GPIO_Mode = GPIO_Mode_OUT;
    ShiftReg_GPIO_Struct.GPIO_OType = GPIO_OType_PP;
    ShiftReg_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    // An arbitrary speed was chosen
    ShiftReg_GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &ShiftReg_GPIO_Struct);

    // Initialize GPIO for the potentiometer
    GPIO_InitTypeDef Potentiometer_GPIO_Struct;
    // Using pin 3
    Potentiometer_GPIO_Struct.GPIO_Pin = 0b1000;
    Potentiometer_GPIO_Struct.GPIO_Mode = GPIO_Mode_AN;
    Potentiometer_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &Potentiometer_GPIO_Struct);

    //Initialize ADC
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

```

ADC_InitTypeDef ADC_Struct;
ADC_Struct.ADC_ContinuousConvMode = DISABLE;
ADC_Struct.ADC_DataAlign = ADC_DataAlign_Left;
ADC_Struct.ADC_Resolution = ADC_Resolution_8b;
ADC_Struct.ADC_ScanConvMode = DISABLE;
ADC_Struct.ADC_ExternalTrigConv = DISABLE;
ADC_Init(ADC1, &ADC_Struct);
ADC_Cmd(ADC1, ENABLE );
ADC-RegularChannelConfig(ADC1, ADC_Channel_13 , 1, ADC_SampleTime_56Cycles);

// Initialize the lights so that only the green light is on at the start
GPIO_ResetBits(GPIOC, 0);
GPIO_ResetBits(GPIOC, 1);
GPIO_SetBits(GPIOC, 4);
}

```

- Middleware to read the current value of the ADC:

Once the ADC is configured it can be used to retrieve the ADC value as follows. First, we start the conversion, we then wait till this conversion is complete, and lastly we get the conversion value. These steps all make use of the APIs existing and is shown below in the code snippet.

```

static void flowAdjustmentTask( void *pvParameters )
{
    uint16_t adc_value;
    display temp;
    temp.reqType = flow;
    while(1)
    {
        ADC_SoftwareStartConv(ADC1);
        // Wait for the conversion to finish
        while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
        // Get the ADC value
        adc_value = ADC_GetConversionValue(ADC1);
        printf("ADC Value: %d\n", adc_value);
        temp.flow = adc_value;
        // Ensure most up to date ADC value is used
        xQueueSendToFront(ADCQueue, &temp, pdMS_TO_TICKS(500));
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

```

The flow chart of this above function is also depicted below in figure 5.

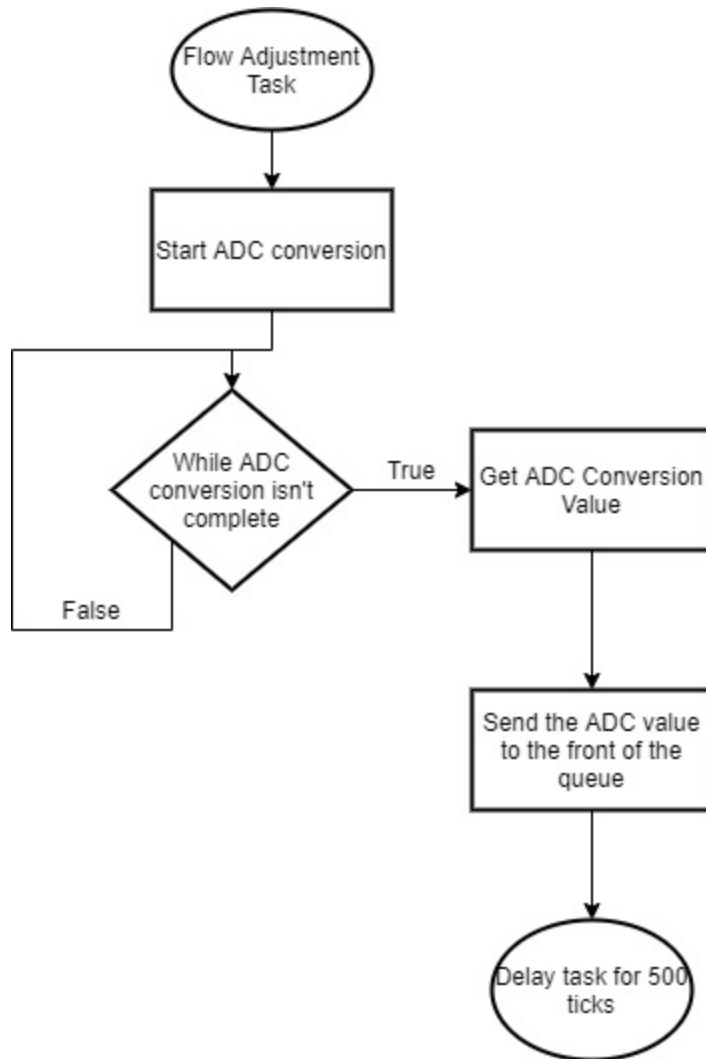


Figure 5: Flow adjustment task flow chart

- Middleware to output the traffic to the shift register:

There are three daisy-chained shift registers to act as a single serial-to-parallel converter (SPC). In this project 3 shift registers are stacked together to increase the number of outputs using a single input.

To send a high or low bit the figure below details how each pin should be set. Once shifted the new entry is placed at Q0 and Q18 is pushed out by the inputs that are shifted. The code to send a low or high bit is shown below.




Operating Mode	Reset PC8	Data PC6	Clock PC7	Q0	Q1-Q18
Reset(Clear)		X	X	L	L
Insert H + Shift	H	H		L	Q0-Q17
Insert L + Shift	H	L		H	Q0-Q17

Figure 6: Shift register operating mode chart [2]

To set a high bit:

- Set the rest bit
- Set the data bit
- Set clock low
- Clear clock to go back to high

To set a low bit:

- Set the rest bit
- Clear the data bit
- Set clock low
- Clear clock to go back to high

```

/*-----*/
void turnOnLED()
{
    // Parallel shift registers used
    // Clock is reset, data is set, clock is then set, and cleared
    GPIO_SetBits(GPIOC, GPIO_Pin_8);
    GPIO_SetBits(GPIOC, GPIO_Pin_6);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
    GPIO_ResetBits(GPIOC,GPIO_Pin_7);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
}

/*-----*/
void turnOffLED()
{
    // Parallel shift registers used
    // Clock is reset, data is set, clock is then set, and cleared
    GPIO_SetBits(GPIOC, GPIO_Pin_8);
    GPIO_ResetBits(GPIOC, GPIO_Pin_6);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
    GPIO_ResetBits(GPIOC,GPIO_Pin_7);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
}

```



- Middleware to set the traffic light to a certain state:

The traffic lights could theoretically all be on as they are independent GPIO pins. The below commands show how individual lights are turned on and off. However, only one light bit is set at a time. This can be seen in more detail when the traffic light task is discussed below.

Pin 0 = 0b001 = 1

Pin 1 = 0b010 = 2

Pin 2 = 0b100 = 4

These values shown above isolate individual pins.

```
// Turn on the green light
GPIO_SetBits(GPIOC, 4);
// Turn on the amber light
GPIO_SetBits(GPIOC, 2);
// Turn on the red light
GPIO_SetBits(GPIOC, 1);
// Turn off the green light
GPIO_ResetBits(GPIOC, 4);

// Turn off the amber light
GPIO_ResetBits(GPIOC, 2);

// Turn off the red light
GPIO_ResetBits(GPIOC, 1);
```

3.3. Software Description

The software written made use of FreeRTOS tasks, queues, and timers. The following tasks are used as was described in the design document.

1. Traffic Flow Adjustment Task: The task reads the value of the potentiometer at certain intervals and send the values to other tasks. The lower the value the lighter the traffic.
2. Traffic Generator Task: This task generates new traffic with a rate that is proportional to the potentiometer's value. The generated traffic is sent to another to be displayed.
3. Traffic Light State Task: This task controls the timing of the traffic lights and outputs the lights state. The timing of the lights is affected by the traffic flow. When the traffic is heavier the duration of the green light increases, and the duration of the red light decreases. This is the opposite when the traffic is lighter.
4. System Display Task: This task controls the traffic light LED and the car LED's.

3.4. Traffic Generating Algorithm

The traffic generating algorithm creates an empty traffic spot if the distance between cars is less than a certain distance. When the distance between the cars exceeds this distance, a car is generated at the start of the road. The distance between cars changes based on the traffic flow. A few examples are shown below based on ADC values obtained in lab.

Flow (ADC value)	(int) Flow/10,000	Distance between cars
1,025 (min potentiometer value)	0	$6 - 0 = 6$
30,000	3	$6 - 3 = 3$
64,000 (max potentiometer value)	6	$6 - 6 = 0$

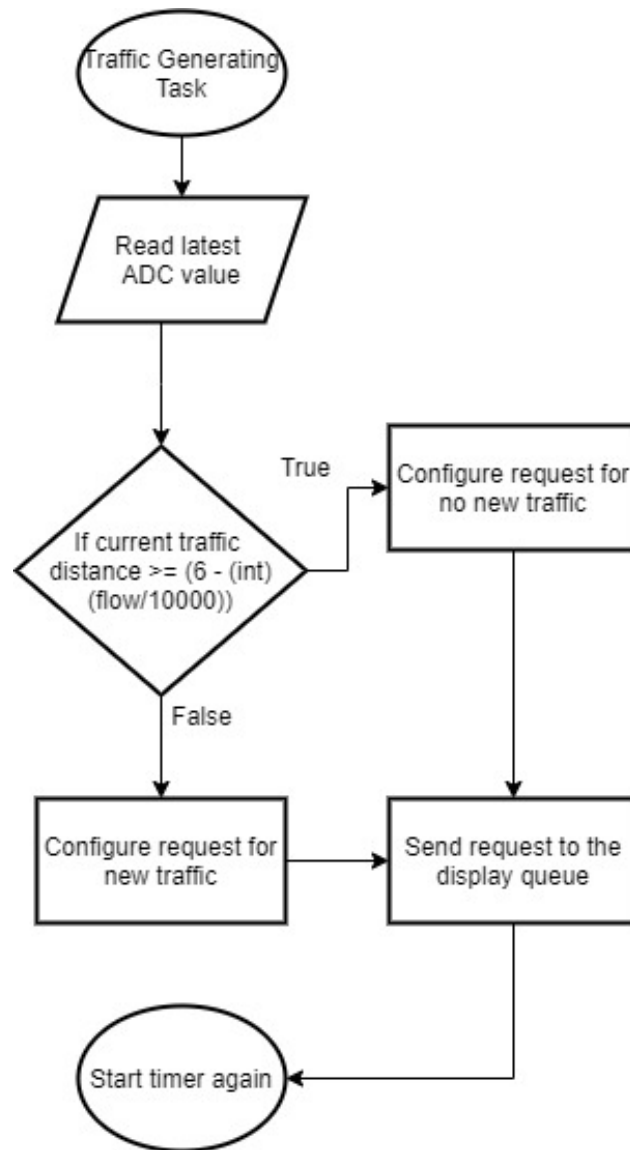


Figure 7: System display flow chart



3.5. System Display Algorithm

The system display flow chart is shown below. If the request matches a light the bit for that light is set and the other light bits are cleared. If the type of the request is of a flow type. The cars are shifted for each flow request but if the flow holds a value of 1 a new car is added to the beginning of the displayed traffic array. Every time the cars are updated the entire traffic array is re-displayed. This is because there are situations where traffic beyond the stop line continues to advance. Since we are using a shift register there is no way to only modify later bits. They must all be pushed out.

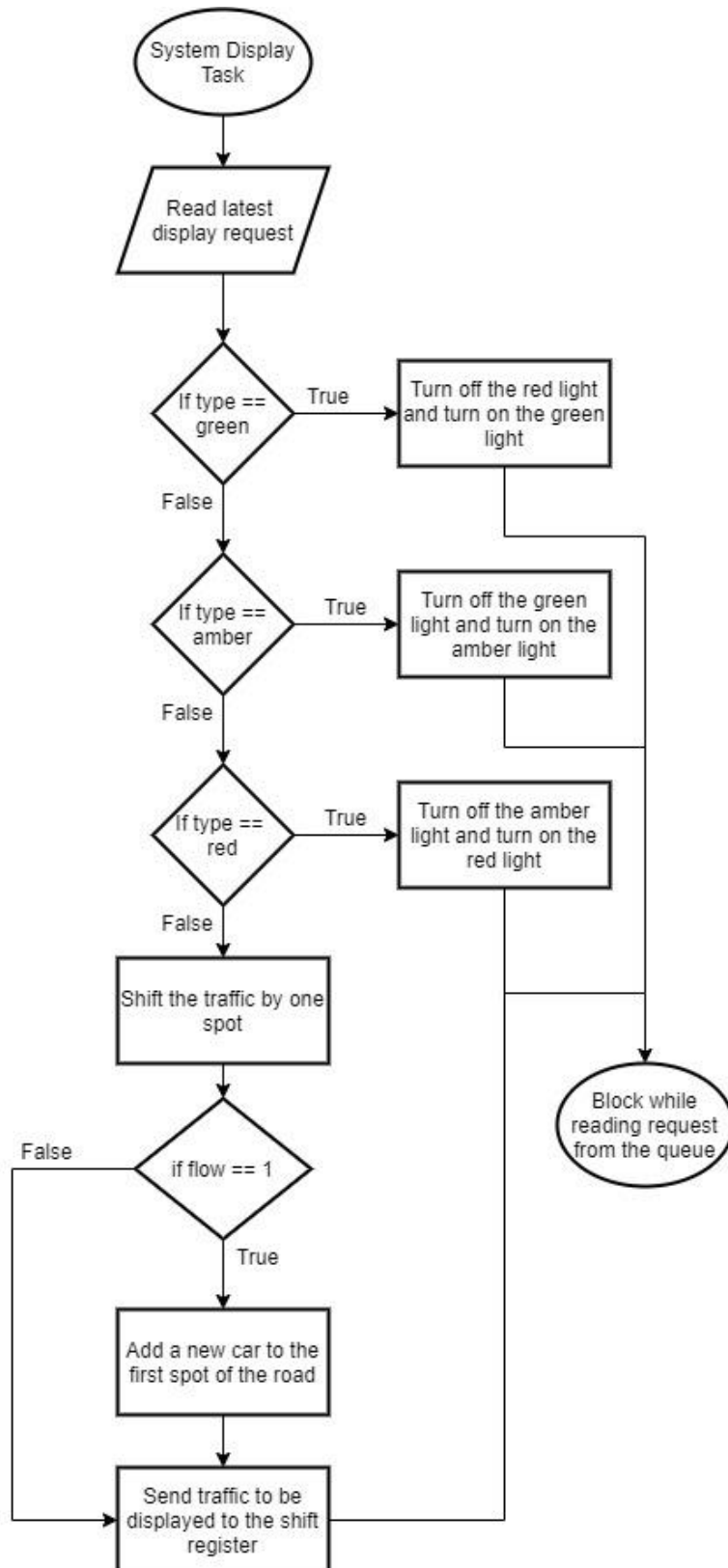


Figure 8: System display flow chart

3.6 Traffic Light Algorithm

The software timer identifies the next lights state. For example a id of 0 indicates the next light is green light. An id of 1 indicate the next light is amber. The timer ID is incremented one value until 2 where the value wraps back around to 0. The period of the red and green light is also altered based on the flow value. While the yellow light period is of a constant value. The switching point of the formula is at an ADC value of 30,000. This is a limitation of the system and it'll be discussed in the later section. The example period values are shown below based on the formula used in my code.

Flow (ADC value)	Green Light period	Red Light period
1,025 (min potentiometer value)	750 ms	1500 ms
30,000	2150	4500 ms
64,000 (max potentiometer value)	13,000	7500 ms

Formula before an ADC value 30,000:

Red Light period = $750 * (\text{int})(\text{request.flow}/10000)$;

Green Light period = $1500 * (\text{int})(\text{request.flow}/10000)$;

Formula after an ADC value 30,000:

Red Light period = $(\text{int})(3000 + 750 * (\text{request.flow}/10000))$;

Green Light period = $(\text{int})(4000 + 1500 * (\text{request.flow}/10000))$;

This was aimed to meet the following requirements:

- When the traffic flow is at its maximum setting:
 - The traffic light must stay green approximately twice as long as it stays red.
- When the traffic flow is at its minimum setting:
 - The traffic light must stay red approximately twice as long as it stays green.

The flow chart of this algorithm omits these period calculations but describes the other steps followed in this task.

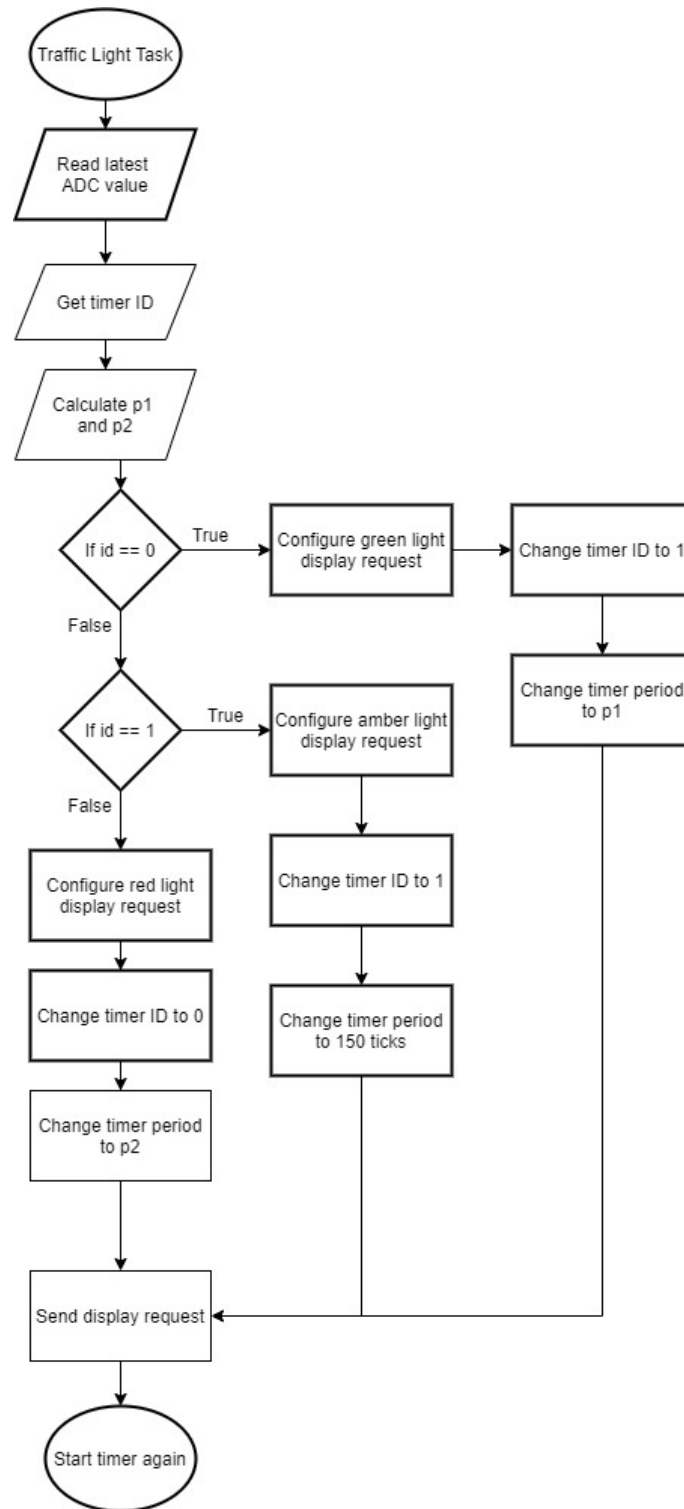


Figure 9: Traffic Light flow chart

4. Limitations and Possible Improvements

As mentioned above the formula that I used to determine the period of the lights is not as effective as it could be. This is because the red light will still be of increased duration until an ADC value of 30,000. At this point the green light is then of double duration than the red. The possible improvement for this issue would be a gradual constant equation. I feel like providing this as a requirement for the project would be a large improvement and something that I would modify if I were to go through this project again.

5. Summary

Overall, this project worked successfully. The demonstration of the program was a success and met the requirements. This project was able to allow me to get hands on experience with tasks, queues, and software timers. These FreeRTOS features are extremely important in real time systems and this project was able to demonstrate that.

6. References

- [1] LAB MANUAL ECE 455: REAL TIME COMPUTER SYSTEMS DESIGN PROJECT, 1st ed. Victoria: University of Victoria, 2019.
- [2] Project-1-Explained.pptx, Victoria: University of Victoria, 2021.

7. Appendix: Project Code

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"
/* Kernel includes. */
#include "stm32f4xx.h"
#include "../FreeRTOS_Source/include/FreeRTOS.h"
#include "../FreeRTOS_Source/include/queue.h"
#include "../FreeRTOS_Source/include/semphr.h"
#include "../FreeRTOS_Source/include/task.h"
#include "../FreeRTOS_Source/include/timers.h"
#include "../Libraries/STM32F4xx_StdPeriph_Driver/inc/stm32f4xx_gpio.h"
#include "../Libraries/STM32F4xx_StdPeriph_Driver/inc/stm32f4xx_adc.h"
/*-----*/
static void prvSetupHardware( void );

// Struct and enum used for queue communication
typedef enum type {green, amber, red, flow} type;
typedef struct display {
    type reqType;
    uint16_t flow;
} display;

// Queues, timers, and traffic array
xQueueHandle ADCQueue;
xQueueHandle requestQueue;
xTimerHandle lightTimer;
xTimerHandle trafficTimer;
int traffic[19];

/*-----*/
void initGPIOandADC(void)
{
    // Enable GPIOC clock
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    // Initialize GPIO for the traffic light
    GPIO_InitTypeDef TrafficLight_GPIO_Struct;
    // Using pins 0, 1, and 2
    // Red, amber, and green respectively
    TrafficLight_GPIO_Struct.GPIO_Pin = 0b111;
    TrafficLight_GPIO_Struct.GPIO_Mode = GPIO_Mode_OUT;
    TrafficLight_GPIO_Struct.GPIO_OType = GPIO_OType_PP;
    TrafficLight_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC , &TrafficLight_GPIO_Struct);

    // Initialize GPIO for the shift register
    GPIO_InitTypeDef ShiftReg_GPIO_Struct;
    // Using pins 6, 7, and 8
```

```

ShiftReg_GPIO_Struct.GPIO_Pin = 0b111000000;
ShiftReg_GPIO_Struct.GPIO_Mode = GPIO_Mode_OUT;
ShiftReg_GPIO_Struct.GPIO_OType = GPIO_OType_PP;
ShiftReg_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
// An arbitrary speed was chosen
ShiftReg_GPIO_Struct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC , &ShiftReg_GPIO_Struct);

// Initialize GPIO for the potentiometer
GPIO_InitTypeDef Potentiometer_GPIO_Struct;
// Using pin 3
Potentiometer_GPIO_Struct.GPIO_Pin = 0b1000;
Potentiometer_GPIO_Struct.GPIO_Mode = GPIO_Mode_AN;
Potentiometer_GPIO_Struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOC, &Potentiometer_GPIO_Struct);

//Initialize ADC
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
ADC_InitTypeDef ADC_Struct;
ADC_Struct.ADC_ContinuousConvMode = DISABLE;
ADC_Struct.ADC_DataAlign = ADC_DataAlign_Left;
ADC_Struct.ADC_Resolution = ADC_Resolution_8b;
ADC_Struct.ADC_ScanConvMode = DISABLE;
ADC_Struct.ADC_ExternalTrigConv = DISABLE;
ADC_Init(ADC1, &ADC_Struct);
ADC_Cmd(ADC1, ENABLE );
ADC_RegularChannelConfig(ADC1, ADC_Channel_13 , 1, ADC_SampleTime_56Cycles);

// Initialize the lights so that only the green light is on at the start
GPIO_ResetBits(GPIOC, 0);
GPIO_ResetBits(GPIOC, 1);
GPIO_SetBits(GPIOC, 4);
}

/*-----*/
void turnOnLED()
{
    // Parallel shift registers used
    // Clock is reset, data is set, clock is then set, and cleared
    GPIO_SetBits(GPIOC, GPIO_Pin_8);
    GPIO_SetBits(GPIOC, GPIO_Pin_6);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
    GPIO_ResetBits(GPIOC,GPIO_Pin_7);
    GPIO_SetBits(GPIOC, GPIO_Pin_7);
}

/*-----*/
void turnOffLED()
{

```

```

// Parallel shift registers used
// Clock is reset, data is set, clock is then set, and cleared
GPIO_SetBits(GPIOC, GPIO_Pin_8);
GPIO_ResetBits(GPIOC, GPIO_Pin_6);
GPIO_SetBits(GPIOC, GPIO_Pin_7);
GPIO_ResetBits(GPIOC,GPIO_Pin_7);
GPIO_SetBits(GPIOC, GPIO_Pin_7);
}

/*-----*/
void sendToShiftRegister()
{
    // Used to output traffic array
    for (int i = 18; i >= 0 ; i--){
        if(traffic[i] == 1){
            turnOnLED();
        } else {
            turnOffLED();
        }
    }
}

/*-----*/
void shiftArrayByOne (int canAdvance)
{
    if(canAdvance == 1){
        // Green light, all traffic moves up one spot
        for (int i = 17; i >= 0; i--){
            traffic[i + 1] = traffic[i];
        }
        traffic[0] = 0;
    } else {
        // Traffic before the light moves up one if the next spot is free
        for (int i = 6; i >= 0; i--){
            if(traffic[i + 1] != 1){
                traffic[i + 1] = traffic[i];
                traffic[i] = 0;
            }
        }
        // Traffic in or after the intersection moves up one spot
        for (int i = 17; i >= 8; i--){
            traffic[i + 1] = traffic[i];
        }
        traffic[8] = 0;
        traffic[0] = 0;
    }
}

/*
The traffic flow that enters the intersection is set by a potentiometer. This task reads the

```

value of the potentiometer at an appropriate interval and sends its value to other tasks.

```
-----*/
static void flowAdjustmentTask( void *pvParameters )
{
    uint16_t adc_value;
    display temp;
    temp.reqType = flow;
    while(1)
    {
        ADC_SoftwareStartConv(ADC1);
        // Wait for the conversion to finish
        while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
        // Get the ADC value
        adc_value = ADC_GetConversionValue(ADC1);
        printf("ADC Value: %d\n", adc_value);
        temp.flow = adc_value;
        // Ensure most up to date ADC value is used
        xQueueSendToFront(ADCQueue, &temp, pdMS_TO_TICKS(500));
        vTaskDelay(500);
    }
}

/*-----*/
int getDistance(){
    // Distance from the start to the next car
    for(int i = 0; i < 18; i++){
        if(traffic[i] == 1){
            return i;
        }
    }
}

/*
This task randomly generates new traffic with a rate that is proportional to the
potentiometer's value this value is received from the Traffic Flow Adjustment Task.
The generated traffic is then sent to another task so that it can be
displayed on the road.
-----*/
void trafficGeneratorTask( xTimerHandle lightTimer )
{
    display temp;
    xQueueReceive(ADCQueue, &temp, pdMS_TO_TICKS(500));
    if(getDistance() >= (6 - (int)(temp.flow/10000)){
        // New traffic is needed
        temp.flow = 1;
    } else {
        // No new traffic is needed
        temp.flow = 0;
    }
    // Send a display request for the traffic
}
```

```

xQueueSend(requestQueue, &temp, pdMS_TO_TICKS(500));
xTimerStart(trafficTimer, 0);
}

/* This task controls the timing of the traffic lights and outputs it's current state.
The timing of the lights is affected by the load of the traffic which is received from the
Traffic Flow Adjustment Task.
-----*/
void trafficLightTask( xTimerHandle lightTimer )
{
    display request;
    xQueueReceive(ADCQueue, &request, pdMS_TO_TICKS(500));
    int temp = (int)pvTimerGetTimerID(lightTimer);
    int p1 = 0;
    int p2 = 0;
    // Calculate the timer period based on the flow
    // If the traffic flow is high the green light should be twice as long as the red light
    // If the traffic flow is low the green light should be half as long as the red light
    if(request.flow > 30000){
        p2 = 750 * (int)(request.flow/10000);
        p1 = 1500 * (int)(request.flow/10000);
    } else {
        p1 = (int)(3000 + 750* (request.flow/10000));
        p2 = (int)(4000 + 1500 * (request.flow/10000));
    }

    // Set new timer period and change the id based on the current lights status
    if(temp == 0){
        request.reqType = green;
        vTimerSetTimerID( lightTimer, (void *) 1);
        xTimerChangePeriod( lightTimer, pdMS_TO_TICKS(p1), 100);
    } else if (temp == 1) {
        request.reqType = amber;
        vTimerSetTimerID( lightTimer, (void *) 2);
        xTimerChangePeriod( lightTimer, pdMS_TO_TICKS(1500), 100);
    } else if (temp == 2) {
        request.reqType = red;
        vTimerSetTimerID( lightTimer, (void *) 0);
        xTimerChangePeriod( lightTimer, pdMS_TO_TICKS(p2), 100);
    }
    request.flow = 0;
    // Send a display request for the light
    xQueueSend(requestQueue, &request, pdMS_TO_TICKS(500));
    xTimerStart(lightTimer, 0);
}

```

/* This task controls all LEDs in the system and is responsible for visualizing all vehicle traffic and the traffic lights. It receives information from the Traffic Generator Task as well as the Traffic Light State Task and controls the system's LEDs accordingly. This task also refreshes the car LEDs at a certain interval to emulate the flow of the traffic.

```

-----*/
static void systemDisplayTask( void *pvParameters )
{
    display temp;
    /* Traffic cannot advance if it is not in or beyond the intersection on a red or amber
    light. A canAdvance with value 1 indicates it is a green light. */
    int canAdvance = 1;
    while(1)
    {
        if(xQueueReceive(requestQueue, &temp, pdMS_TO_TICKS(500))){
            if(temp.reqType == green){
                // Turn of red and turn on green
                GPIO_ResetBits(GPIOC, 1);
                GPIO_SetBits(GPIOC, 4);
                canAdvance = 1;
            } else if (temp.reqType == amber) {
                // Turn of green and turn on amber
                GPIO_ResetBits(GPIOC, 4);
                GPIO_SetBits(GPIOC, 2);
                canAdvance = 0;
            } else if (temp.reqType == red) {
                // Turn of amber and on red
                GPIO_ResetBits(GPIOC, 2);
                GPIO_SetBits(GPIOC, 1);
                canAdvance = 0;
            } else {
                // Advance traffic
                shiftArrayByOne(canAdvance);
                // Add traffic if needed
                if(temp.flow == 1){
                    traffic[0] = 1;
                }
                // Display traffic
                sendToShiftRegister();
            }
        } else {
            vTaskDelay(250);
        }
    }
}

/*-----*/
int main(void)
{
    // Initialize the ADC, GPIO, and the hardware
    prvSetupHardware();
    initGPIOandADC();
    // Create the queues
    ADCQueue = xQueueCreate(100, sizeof(display));
    requestQueue = xQueueCreate(100, sizeof(display));
}

```

```

// Add to the registry, for the benefit of kernel aware debugging.
vQueueAddToRegistry(ADCQueue, "ADCQueue" );
vQueueAddToRegistry(requestQueue, "requestQueue" );
// Create a ADC task and a Display task
xTaskCreate(flowAdjustmentTask, "ADC_task", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
xTaskCreate(systemDisplayTask, "systemDisplayTask", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
// Create a timer to generate the traffic and generate new traffic periodically
lightTimer = xTimerCreate("trafficLightTask", pdMS_TO_TICKS(1500), pdFALSE, (void *) 0, t
trafficLightTask);
xTimerStart(lightTimer, 0);
trafficTimer = xTimerCreate("trafficGeneratorTask", pdMS_TO_TICKS(1000), pdFALSE, (void *)
) 3, trafficGeneratorTask);
xTimerStart(trafficTimer, 0);
// Start the scheduler
vTaskStartScheduler();
return 0;
}

/*-----*/

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
    configCHECK_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 xFreeStackSize;

    /* 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. */
    xFreeStackSize = xPortGetFreeHeapSize();

    if( xFreeStackSize > 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. */
}

```