University of Victoria

Department of Electrical & Computer Engineering

Winter 2023

Project 1: Traffic Light System

Salem Ait Ami V00932871

Jay Saini V00870475

Lab Section B02

March 7, 2023

Table of Contents

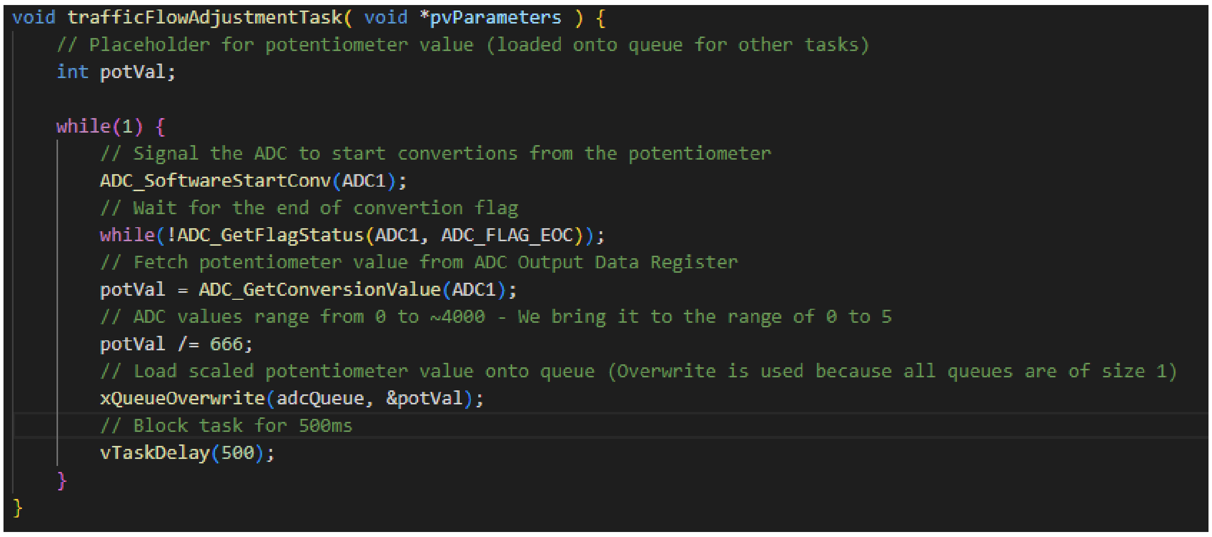
Figures

Introduction

This project involves the design and implementation of a Traffic Light System (TLS) by using middleware, along with FreeRTOS features such as tasks and queues. The TLS is physically constructed on a breadboard by using 19 green LEDs, to simulate traffic, along with 1 red, green, and amber LED, to represent a traffic light at an intersection. The circuit is also connected to a potentiometer whose variable voltage influences both traffic flow and traffic light timing (i.e. at max voltage, flow is maximized and the light is green for double the amount of time at min voltage). Once the circuit is built, FreeRTOS features are used along with built-in middleware APIs to create and manage several tasks, where queues are used for inter-task communication.

Design Solution

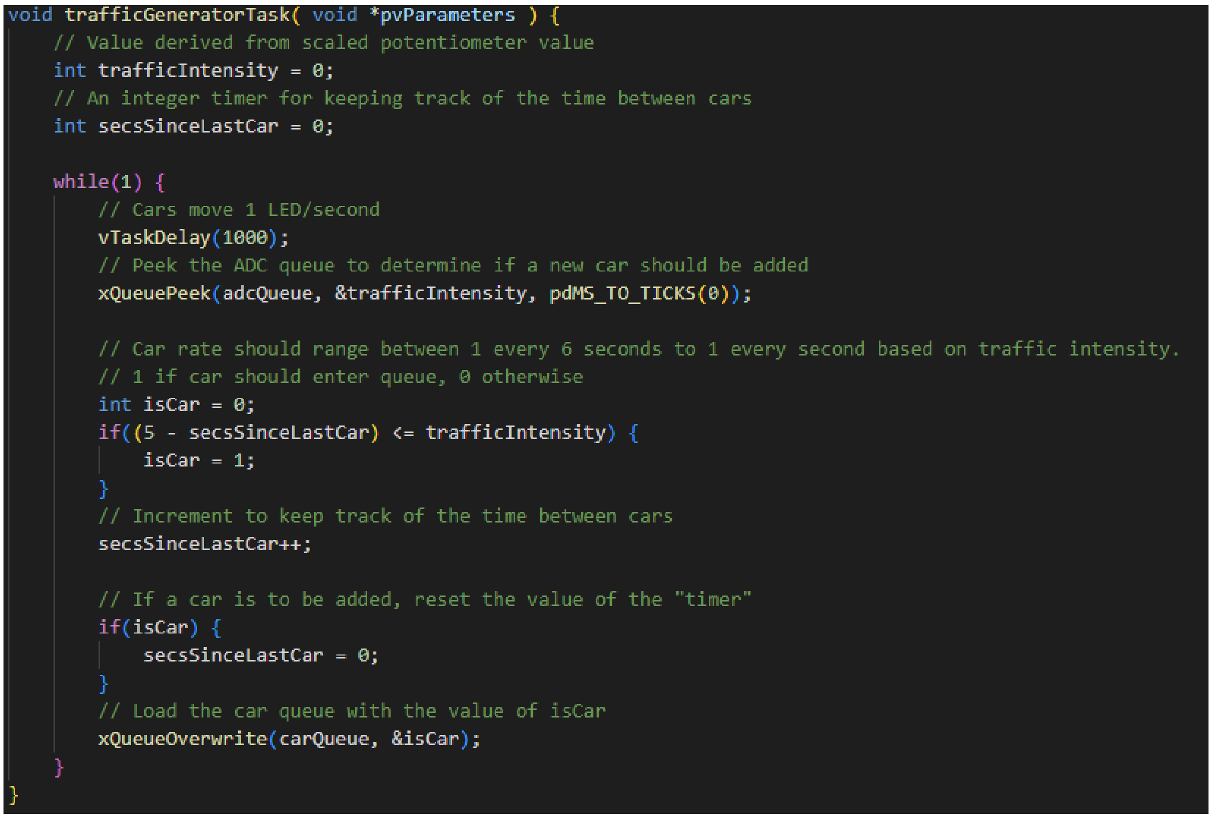
Our solution follows the lab manual's suggestion of using 4 tasks, along with 3 queues for communication between tasks. The 4 tasks are trafficFlowAdjustmentTask, trafficGeneratorTask, trafficLightStateTask, and systemDisplayTask. The 3 queues are carQueue, trafficLightQueue, and adcQueue. The following code snippets show the fully commented code for each task along with a description of what other tasks it communicates with, and which queues are used to do so.



```
void trafficFlowAdjustmentTask( void *pvParameters ) {
    // Placeholder for potentiometer value (loaded onto queue for other tasks)
    int potVal;

    while(1) {
        // Signal the ADC to start convertions from the potentiometer
        ADC_SoftwareStartConv(ADC1);
        // Wait for the end of convertion flag
        while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));
        // Fetch potentiometer value from ADC Output Data Register
        potVal = ADC_GetConversionValue(ADC1);
        // ADC values range from 0 to ~4000 - We bring it to the range of 0 to 5
        potVal /= 666;
        // Load scaled potentiometer value onto queue (Overwrite is used because all queues are of size 1)
        xQueueOverwrite(adcQueue, &potVal);
        // Block task for 500ms
        vTaskDelay(500);
    }
}
```

Figure 1: Traffic Flow Adjustment Task

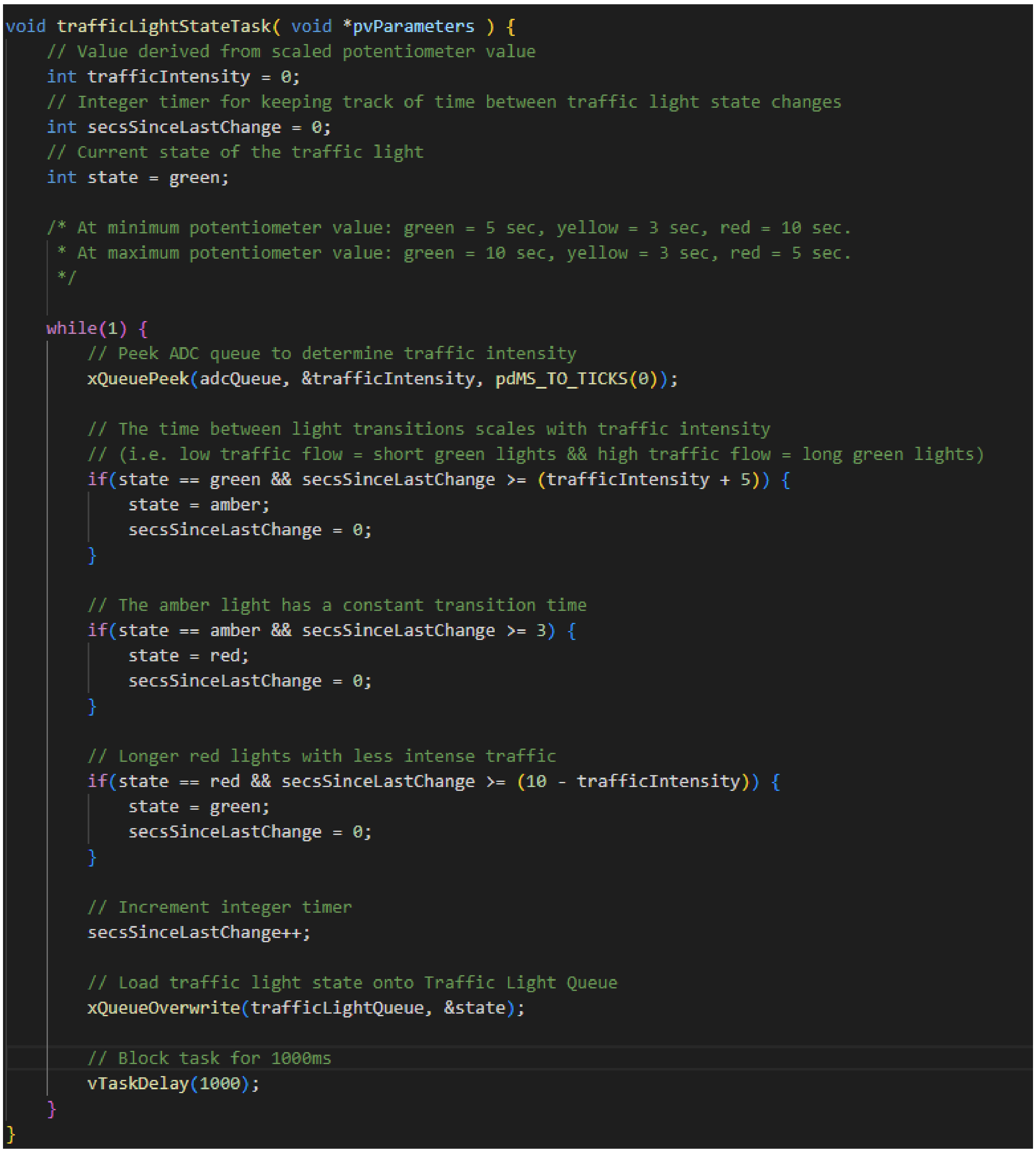
The Traffic Flow Adjustment Task is responsible for reading potentiometer values from the ADC and writing them onto the adcQueue. This queue is later accessed by the Traffic Generator Task and the Traffic Light State Task.



```
void trafficGeneratorTask( void *pvParameters ) {
    // Value derived from scaled potentiometer value
    int trafficIntensity = 0;
    // An integer timer for keeping track of the time between cars
    int secsSinceLastCar = 0;

    while(1) {
        // Cars move 1 LED/second
        vTaskDelay(1000);
        // Peek the ADC queue to determine if a new car should be added
        xQueuePeek(adcQueue, &trafficIntensity, pdMS_TO_TICKS(0));

        // Car rate should range between 1 every 6 seconds to 1 every second based on traffic intensity.
        // 1 if car should enter queue, 0 otherwise
        int isCar = 0;
        if((5 - secsSinceLastCar) <= trafficIntensity) {
            isCar = 1;
        }
        // Increment to keep track of the time between cars
        secsSinceLastCar++;

        // If a car is to be added, reset the value of the "timer"
        if(isCar) {
            secsSinceLastCar = 0;
        }
        // Load the car queue with the value of isCar
        xQueueOverwrite(carQueue, &isCar);
    }
}
```

Figure 2: Traffic Generator Task

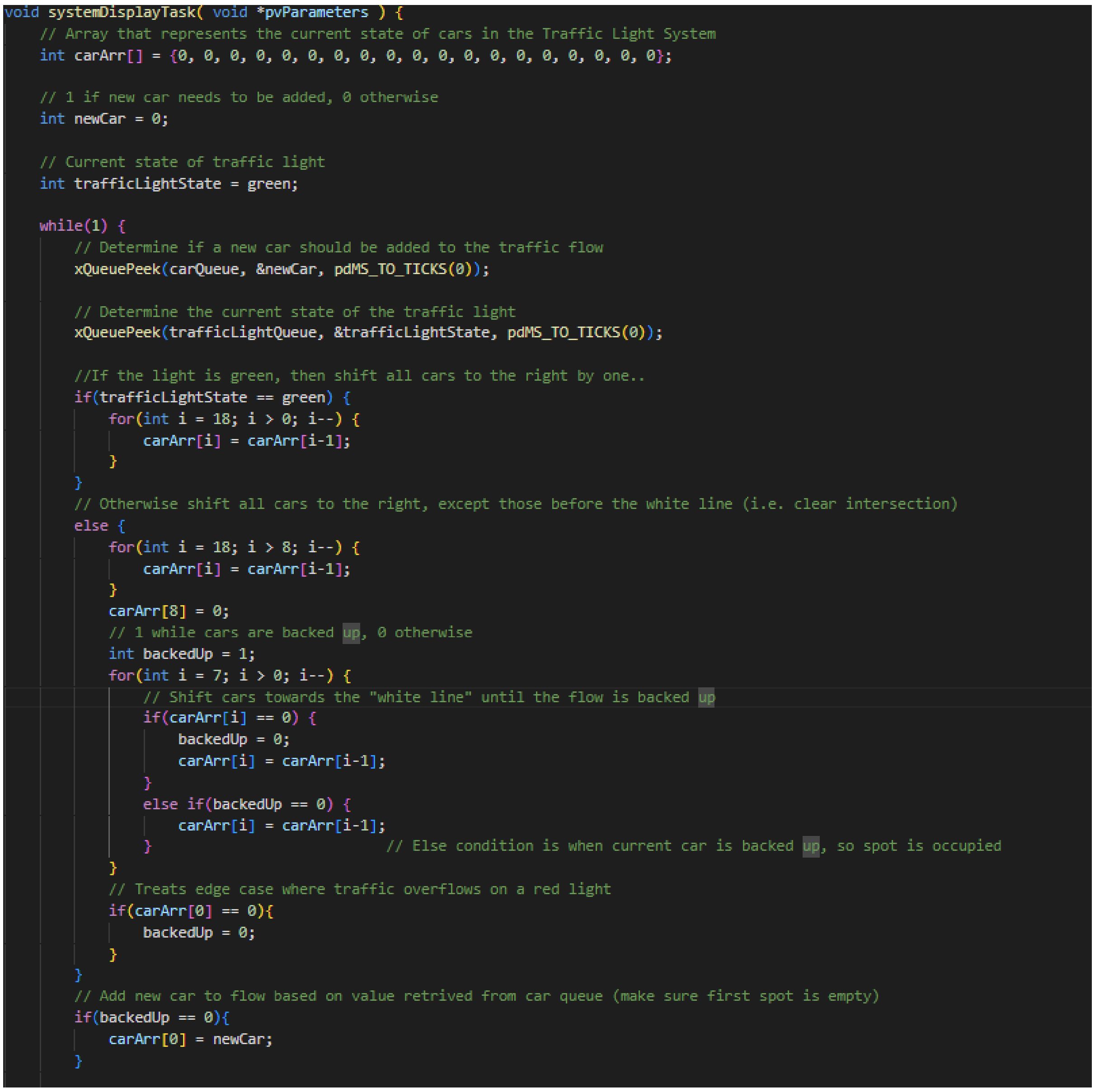
The Traffic Generator Task is responsible for loading a boolean value onto the carQueue to signal the System Display Task that a new car should be added to the traffic flow. It reads from the ADC queue and writes to the car queue.



```c
void trafficLightStateTask( void *pvParameters ) {
    // Value derived from scaled potentiometer value
    int trafficIntensity = 0;
    // Integer timer for keeping track of time between traffic light state changes
    int secsSinceLastChange = 0;
    // Current state of the traffic light
    int state = green;

    /* At minimum potentiometer value: green = 5 sec, yellow = 3 sec, red = 10 sec.
     * At maximum potentiometer value: green = 10 sec, yellow = 3 sec, red = 5 sec.
     */

    while(1) {
        // Peek ADC queue to determine traffic intensity
        xQueuePeek(adcQueue, &trafficIntensity, pdMS_TO_TICKS(0));

        // The time between light transitions scales with traffic intensity
        // (i.e. low traffic flow = short green lights && high traffic flow = long green lights)
        if(state == green && secsSinceLastChange >= (trafficIntensity + 5)) {
            state = amber;
            secsSinceLastChange = 0;
        }

        // The amber light has a constant transition time
        if(state == amber && secsSinceLastChange >= 3) {
            state = red;
            secsSinceLastChange = 0;
        }

        // Longer red lights with less intense traffic
        if(state == red && secsSinceLastChange >= (10 - trafficIntensity)) {
            state = green;
            secsSinceLastChange = 0;
        }

        // Increment integer timer
        secsSinceLastChange++;

        // Load traffic light state onto Traffic Light Queue
        xQueueOverwrite(trafficLightQueue, &state);

        // Block task for 1000ms
        vTaskDelay(1000);
    }
}
```

Figure 3: Traffic Light State Task

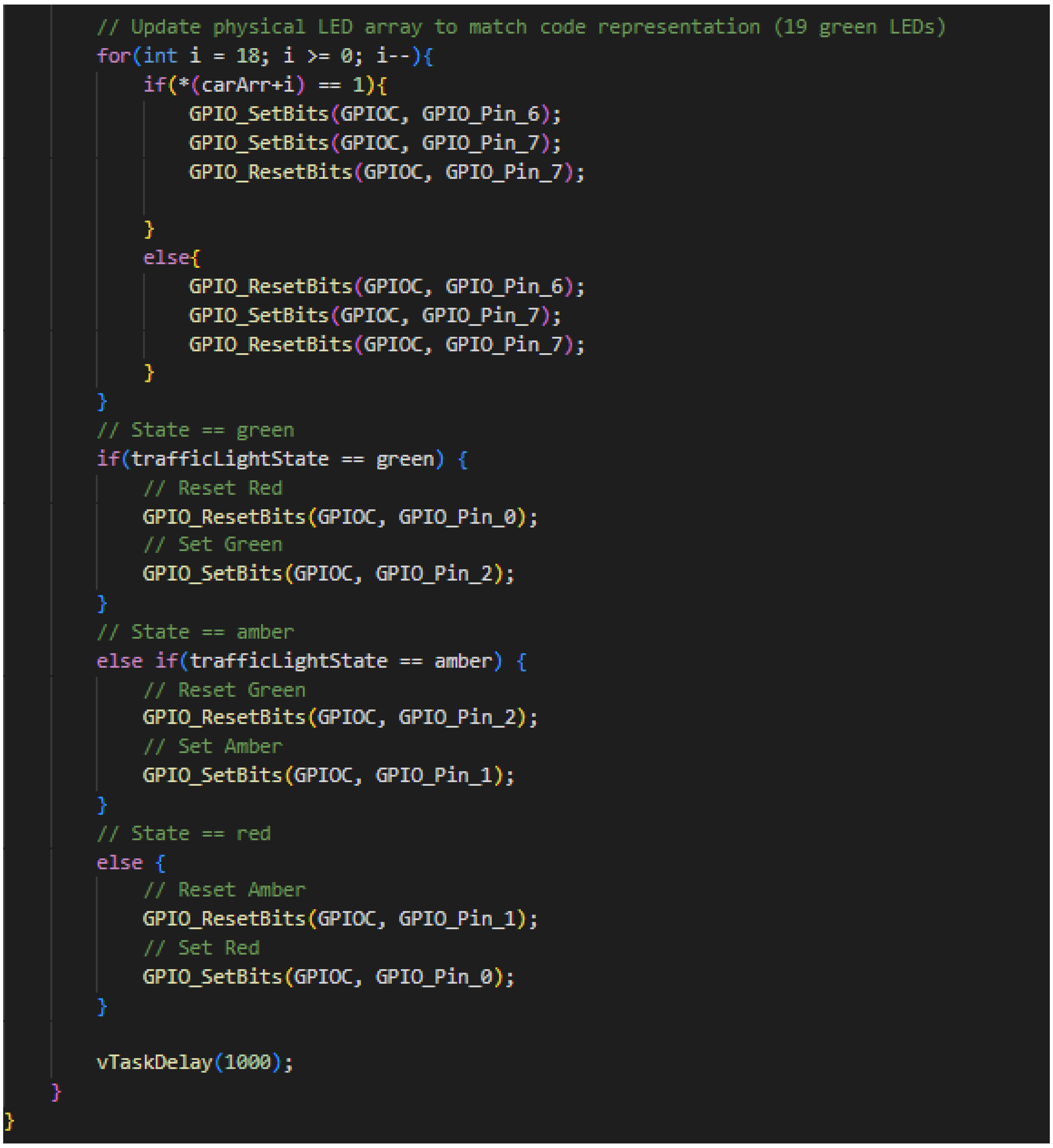
The Traffic Light State Task is responsible for managing the traffic light LEDs along with their transition intervals. It reads from the ADC queue and writes to the traffic light queue.



```c
void systemDisplayTask( void *pvParameters ) {
    // Array that represents the current state of cars in the Traffic Light System
    int carArr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    // 1 if new car needs to be added, 0 otherwise
    int newCar = 0;

    // Current state of traffic light
    int trafficLightState = green;

    while(1) {
        // Determine if a new car should be added to the traffic flow
        xQueuePeek(carQueue, &newCar, pdMS_TO_TICKS(0));

        // Determine the current state of the traffic light
        xQueuePeek(trafficLightQueue, &trafficLightState, pdMS_TO_TICKS(0));

        //If the light is green, then shift all cars to the right by one..
        if(trafficLightState == green) {
            for(int i = 18; i > 0; i--) {
                carArr[i] = carArr[i-1];
            }
        }
        // Otherwise shift all cars to the right, except those before the white line (i.e. clear intersection)
        else {
            for(int i = 18; i > 8; i--) {
                carArr[i] = carArr[i-1];
            }
            carArr[8] = 0;
            // 1 while cars are backed up, 0 otherwise
            int backedUp = 1;
            for(int i = 7; i > 0; i--) {
                // Shift cars towards the "white line" until the flow is backed up
                if(carArr[i] == 0) {
                    backedUp = 0;
                    carArr[i] = carArr[i-1];
                }
                else if(backedUp == 0) {
                    carArr[i] = carArr[i-1];
                }                           // Else condition is when current car is backed up, so spot is occupied
            }
            // Treats edge case where traffic overflows on a red light
            if(carArr[0] == 0){
                backedUp = 0;
            }
        }
        // Add new car to flow based on value retrived from car queue (make sure first spot is empty)
        if(backedUp == 0){
            carArr[0] = newCar;
        }
```

Figure 4.1: System Display Task

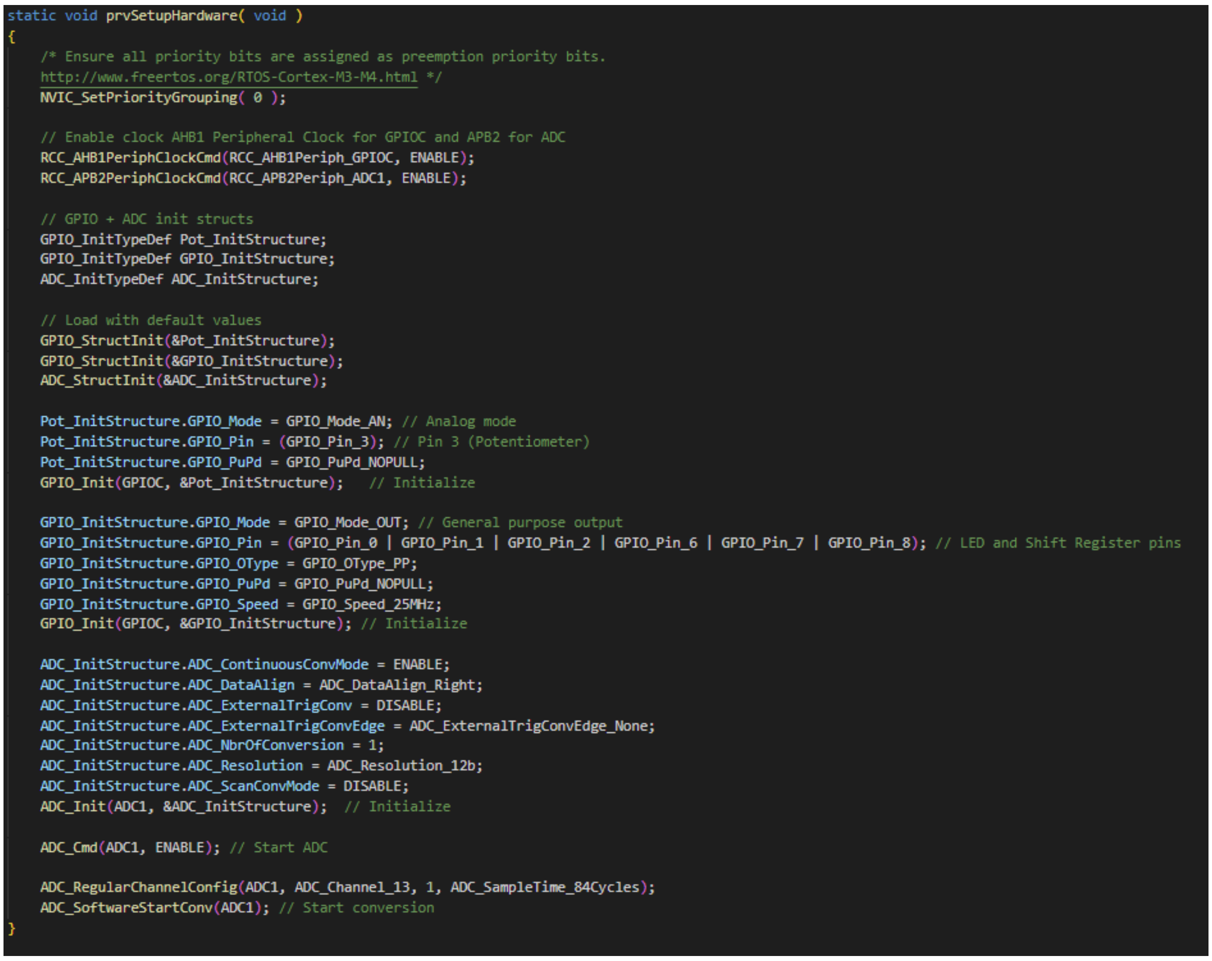
The first half of the System Display Task is responsible for shifting the flow of LEDs based on the traffic light state. It reads from the car queue to determine whether a new car should be added and it reads from the traffic light queue in order to see the current traffic light state.



```
        // Update physical LED array to match code representation (19 green LEDs)
        for(int i = 18; i >= 0; i--){
            if(*(carArr+i) == 1){
                GPIO_SetBits(GPIOC, GPIO_Pin_6);
                GPIO_SetBits(GPIOC, GPIO_Pin_7);
                GPIO_ResetBits(GPIOC, GPIO_Pin_7);

            }
            else{
                GPIO_ResetBits(GPIOC, GPIO_Pin_6);
                GPIO_SetBits(GPIOC, GPIO_Pin_7);
                GPIO_ResetBits(GPIOC, GPIO_Pin_7);
            }
        }
        // State == green
        if(trafficLightState == green) {
            // Reset Red
            GPIO_ResetBits(GPIOC, GPIO_Pin_0);
            // Set Green
            GPIO_SetBits(GPIOC, GPIO_Pin_2);
        }
        // State == amber
        else if(trafficLightState == amber) {
            // Reset Green
            GPIO_ResetBits(GPIOC, GPIO_Pin_2);
            // Set Amber
            GPIO_SetBits(GPIOC, GPIO_Pin_1);
        }
        // State == red
        else {
            // Reset Amber
            GPIO_ResetBits(GPIOC, GPIO_Pin_1);
            // Set Red
            GPIO_SetBits(GPIOC, GPIO_Pin_0);
        }

        vTaskDelay(1000);
    }
}
```

Figure 4.2: System Display Task

The second half of the System Display Task is responsible for updating the hardware such that it matches the software representation of the Traffic Light System. The loop updates the traffic flow LEDs while the conditional block updates the traffic light LEDs accordingly.



```c
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 );

    // Enable clock AHB1 Peripheral Clock for GPIOC and APB2 for ADC
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    // GPIO + ADC init structs
    GPIO_InitTypeDef Pot_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;

    // Load with default values
    GPIO_StructInit(&Pot_InitStructure);
    GPIO_StructInit(&GPIO_InitStructure);
    ADC_StructInit(&ADC_InitStructure);

    Pot_InitStructure.GPIO_Mode = GPIO_Mode_AN; // Analog mode
    Pot_InitStructure.GPIO_Pin = (GPIO_Pin_3); // Pin 3 (Potentiometer)
    Pot_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &Pot_InitStructure);   // Initialize

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; // General purpose output
    GPIO_InitStructure.GPIO_Pin = (GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8); // LED and Shift Register pins
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure); // Initialize

    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_ExternalTrigConv = DISABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_Init(ADC1, &ADC_InitStructure);  // Initialize

    ADC_Cmd(ADC1, ENABLE); // Start ADC

    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_84Cycles);
    ADC_SoftwareStartConv(ADC1); // Start conversion
}
```

Figure 5: Hardware Setup

The first function called in main is responsible for initializing all the necessary hardware components with the appropriate configurations. Middleware APIs are used to simplify this process.

Discussion

The project shows that the design specifications can be met in a reasonable amount of time, with the hardware provided. In comparison to past embedded programming projects, less reading was required from the reference manual, as the built-in middleware APIs simplified the whole process by a significant amount. The guidelines provided in the lab manual were also very helpful for creating a mental model of the system in terms of tasks and queues. Furthermore, since we could build the circuit however we wanted, it was very useful to build a circuit for a single function. This allowed us to approach the problem one task at a time until they could all be integrated. Since traffic moves 1 LED/s and the system display task needs to update the hardware every second, all necessary computations need to be completed before the system display task gets called again. This is why the traffic flow adjustment task has the shortest delay, so we can ensure a fresh value is read from the potentiometer at all times.

We also found it to be much more helpful to have queues of size 1 and constantly peeking to read the values, and overwriting to write new values to the queue. This was found to be simultaneously much simpler, and more reliable than our original approach of pushing to and popping values from the queue. We also had significant and unpredictable issues at demo time but it was later found that these were being caused by a faulty diode on the STM board we were using to demo, and were easily resolved by using the pins on the extension board instead of the STM.

Limitations and Possible Improvements

We did test extensively for any bugs or unexpected behavior, and found some niche ones (such as the first LED blinking off even when we knew there should be a car in that spot), but we were able to resolve all of them with relative ease. So, to our understanding, our code worked flawlessly and exactly as defined by the specifications in the lab manual. Nonetheless, we do have some improvements we could incorporate into the design if given the chance:

- Adding a third side to the intersection so behavior could be designed to simulate a T-intersection
- Adding a second row of LEDs for the reverse flow of traffic, so that the road could be simulated as a 2-way.
- Cleaning up the physical design of our circuit. Despite us trying quite hard, the implementation of our circuit wiring quickly became quite messy. It would be nice to have a sleeker and cleaner looking circuit. Figure 6 shows our final hardware implementation.

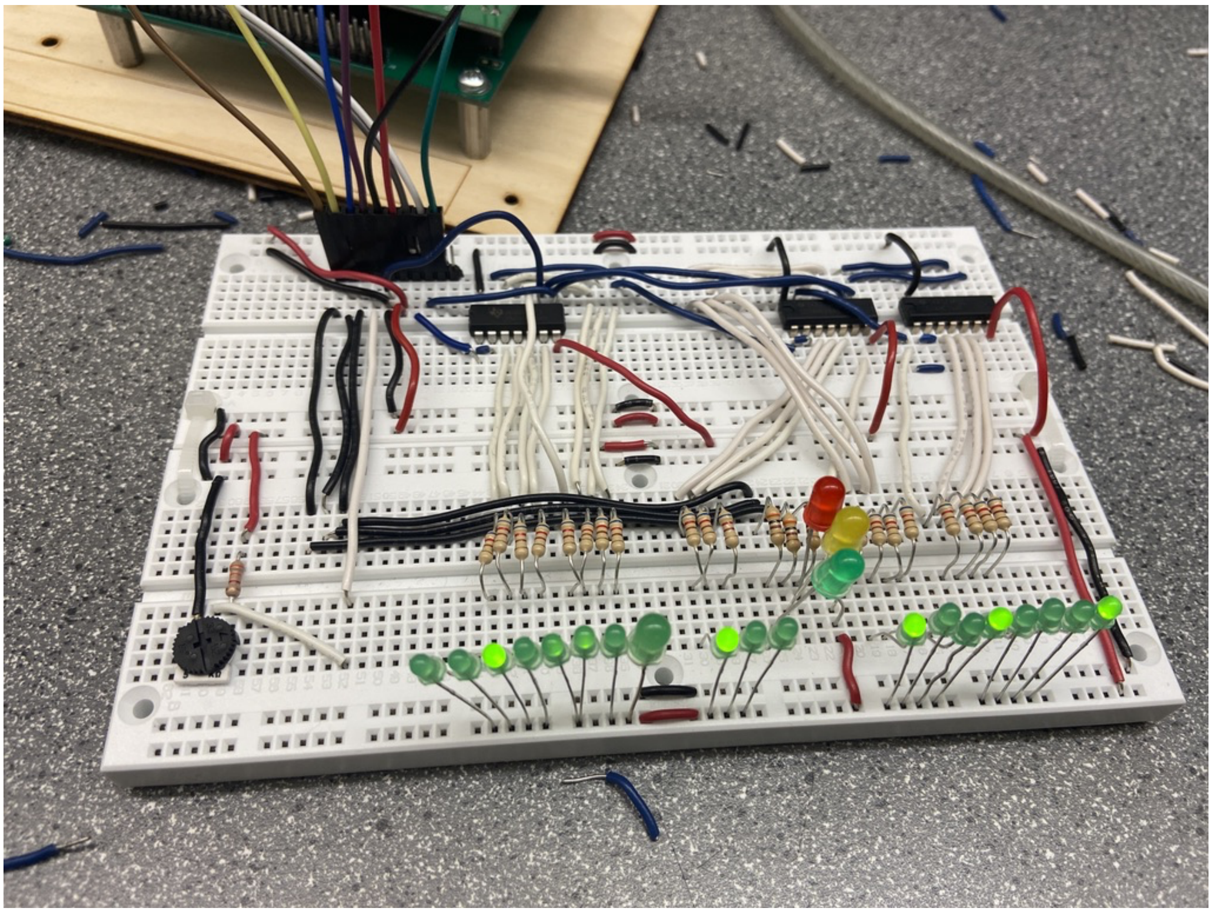


Figure 6: Physical Circuit

Summary

A lot of the workload from this project comes from setting up the circuit and creating the functional model for the system. Afterwards, the actual implementation is fairly simple and does not require any complicated code. One can choose to build the entire circuit at once or for one task at a time. Once each individual task functions as it should, synchronizing them is easily done by using queues and delays. A minimum of 4 tasks and 3 queues is required to complete the project, but more could be used depending on what extra functionality someone would want to add.

Appendix A - Design Document

The code should be structured as described in the lab manual, with four tasks and queues to transfer data between them. In the code itself, each task should have a dedicated function for itself, with helper functions as needed. The four tasks should be for traffic flow adjustment, traffic generation, tracking the traffic light state, and system display. As well as three queues for tracking the light state, recording the traffic queue, and a queue to track the return value from the ADC. The ADC itself should be returning the digital value from the potentiometer.

For the hardware itself, we have designed the following circuit diagram as a guide for the layout of the physical circuit:

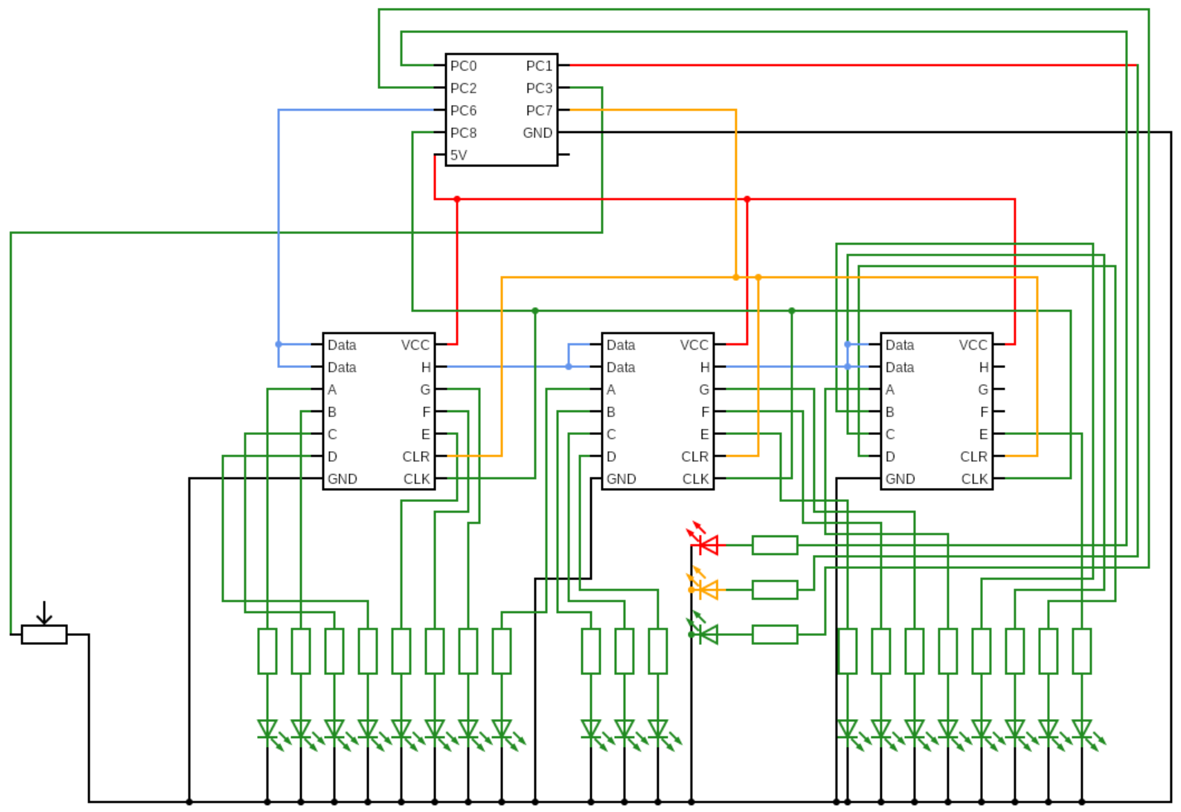


Figure 7: Circuit Design

Appendix B - C Code

```
/* Standard includes. */
#include <stdint.h>
#include <stdio.h>
#include <stdlib.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"



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

static void prvSetupHardware( void );
static void trafficFlowAdjustmentTask( void *pvParameters );
static void trafficGeneratorTask( void *pvParameters );
static void trafficLightStateTask( void *pvParameters );
static void systemDisplayTask( void *pvParameters );

/*
 * The queue send and receive tasks as described in the comments at the
top of
 * this file.
 */

#define red        0
#define amber    1
#define green    2


xQueueHandle carQueue;                     // Tracks locations of cars
xQueueHandle trafficLightQueue;     // Tracks color of traffic light
xQueueHandle adcQueue;                     // Tracks return value from
ADC


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

int main(void)
```

```c
{
    // Configure the system ready to run the demo.
    prvSetupHardware();
    printf("GPIO Initialized!\n");


    // Initialize Queues
    carQueue = xQueueCreate(1, sizeof(uint16_t));
    trafficLightQueue = xQueueCreate(1, sizeof(uint16_t));
    adcQueue = xQueueCreate(1, sizeof(uint16_t));

    // Add to the registry, for the benefit of kernel aware debugging
    vQueueAddToRegistry(carQueue, "Car Array Queue");
    vQueueAddToRegistry(trafficLightQueue, "Traffic Light Queue");
    vQueueAddToRegistry(adcQueue, "ADC Value Queue");

    // Create tasks
    xTaskCreate(trafficFlowAdjustmentTask, "Traffic Flow Adjustment",
configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(trafficGeneratorTask, "Traffic Generator",
configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(trafficLightStateTask, "Traffic Light State",
configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(systemDisplayTask, "System Display",
configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    // Start tasks
    vTaskStartScheduler();

    printf("Program End\n");         // This should never execute

    return 0;
}

void trafficFlowAdjustmentTask( void *pvParameters ) {
    int potVal;                            // Return value from ADC

    while(1) {
       // Retrieve value from ADC
       ADC_SoftwareStartConv(ADC1);
       while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC));        // Wait for
conversion to end
       potVal = ADC_GetConversionValue(ADC1);

//     printf("Potentiometer Value: %d\n", potVal);
       // ADC values range from 0 to ~4000 - We bring it to the range of
0 to 5
       potVal /= 666;
       xQueueOverwrite(adcQueue, &potVal);                        //
Save value to queue
       vTaskDelay(500);
    }
}

void trafficGeneratorTask( void *pvParameters ) {
    int trafficIntensity = 0;              // Range from 0 to 5, 0 is
least intense
    int secsSinceLastCar = 0;

    while(1) {
       // Cars move 1 LED/second
       vTaskDelay(1000);
       xQueuePeek(adcQueue, &trafficIntensity, pdMS_TO_TICKS(0));
//     printf("Traffic Intensity: %d\n", trafficIntensity);

       // Car rate should range between 1 every 6 seconds to 1 every
second based on traffic intensity.
       int isCar = 0; // 1 if car should enter queue, 0 otherwise
       if((5 - secsSinceLastCar) <= trafficIntensity) {
             isCar = 1;
       }

       secsSinceLastCar++;

       if(isCar) {
             secsSinceLastCar = 0;
       }

//     printf("Is car queued: %d\n", isCar);
       xQueueOverwrite(carQueue, &isCar);
    }
}

void trafficLightStateTask( void *pvParameters ) {
    int trafficIntensity = 0;              // Range from 0 to 5, 0 is
least intense
    int secsSinceLastChange = 0;
    int state = green;                           // Color of traffic
light

    /* At minimum potentiometer value: green = 5 sec, yellow = 3 sec,
red = 10 sec.
```

```
     * At maximum potentiometer value: green = 10 sec, yellow = 3 sec,
red = 5 sec.
     */

    while(1) {
       xQueuePeek(adcQueue, &trafficIntensity, pdMS_TO_TICKS(0));

       // Update traffic light color if conditions are met
       if(state == green && secsSinceLastChange >= (trafficIntensity +
5)) {
             state = amber;
             secsSinceLastChange = 0;
       }

       if(state == amber && secsSinceLastChange >= 3) {
             state = red;
             secsSinceLastChange = 0;
       }

       if(state == red && secsSinceLastChange >= (10 -
trafficIntensity)) {
             state = green;
             secsSinceLastChange = 0;
       }

//     printf("Seconds since last change: %d\n", secsSinceLastChange);
//     printf("Current state: %d\n", state);

       secsSinceLastChange++;
       xQueueOverwrite(trafficLightQueue, &state);
       vTaskDelay(1000);
    }
}

void systemDisplayTask( void *pvParameters ) {
    int carArr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0};
    int newCar = 0;                             // 1 if new car needs to be
added, 0 otherwise
    int trafficLightState = green;
    int backedUp = 1;                                 // 1 while cars are
backed up, 0 otherwise

    while(1) {
       xQueuePeek(carQueue, &newCar, pdMS_TO_TICKS(0));
       xQueuePeek(trafficLightQueue, &trafficLightState,
pdMS_TO_TICKS(0));
        backedUp = 1;

        if(trafficLightState == green) {
                backedUp = 0;                                   // When green, no
traffic
                for(int i = 18; i > 0; i--) {
                        carArr[i] = carArr[i-1];
                }
        } else {
                for(int i = 18; i > 8; i--) {
                        carArr[i] = carArr[i-1];     // Move cars ahead of
white line (intersection marker) forward by 1
                }
                carArr[8] = 0;
                for(int i = 7; i > 0; i--) {
                        if(carArr[i] == 0) {
                                backedUp = 0;
                                carArr[i] = carArr[i-1];
                        } else if(backedUp == 0) {
                                carArr[i] = carArr[i-1];
                        }                                       // Else condition
is when current car is backed up, so spot is occupied
                }
                if(carArr[0] == 0) {                    // This conditions
treats very specific edge case where the first spot is left unfilled on
red light
                        backedUp = 0;
                }
        }

        if(backedUp == 0) {                             // Make sure first spot
is empty
                carArr[0] = newCar;
        }


        // Update physical LED array to match code representation
        for(int i = 18; i >= 0; i--) {
                if(*(carArr+i) == 1){
                        GPIO_SetBits(GPIOC, GPIO_Pin_6);
                        GPIO_SetBits(GPIOC, GPIO_Pin_7);
                        GPIO_ResetBits(GPIOC, GPIO_Pin_7);

                }
                else {
                   GPIO_ResetBits(GPIOC, GPIO_Pin_6);
                   GPIO_SetBits(GPIOC, GPIO_Pin_7);
                   GPIO_ResetBits(GPIOC, GPIO_Pin_7);
             }
       }

       if(trafficLightState == green) {                      // State ==
green
             GPIO_ResetBits(GPIOC, GPIO_Pin_0);              // Reset Red
             GPIO_SetBits(GPIOC, GPIO_Pin_2);                // Set Green
       } else if(trafficLightState == amber) {               // State ==
amber
             GPIO_ResetBits(GPIOC, GPIO_Pin_2);              // Reset
Green
             GPIO_SetBits(GPIOC, GPIO_Pin_1);                // Set Amber
       } else {
// State == red
             GPIO_ResetBits(GPIOC, GPIO_Pin_1);              // Reset
Amber
             GPIO_SetBits(GPIOC, GPIO_Pin_0);                // Set Red
       }

       vTaskDelay(1000);
    }
}

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

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

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

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

void vApplicationIdleHook( void )
{
volatile size_t xFreeStackSpace;

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

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

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

static void prvSetupHardware( void )
{
    /* Ensure all priority bits are assigned as preemption priority
bits.
    http://www.freertos.org/RTOS-Cortex-M3-M4.html */
    NVIC_SetPriorityGrouping( 0 );
    // Enable clock AHB1 Peripheral Clock for GPIOC and APB2 for ADC
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    // GPIO + ADC init structs
    GPIO_InitTypeDef Pot_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;

    // Load with default values
    GPIO_StructInit(&Pot_InitStructure);
    GPIO_StructInit(&GPIO_InitStructure);
    ADC_StructInit(&ADC_InitStructure);

    Pot_InitStructure.GPIO_Mode = GPIO_Mode_AN; // Analog mode
    Pot_InitStructure.GPIO_Pin = (GPIO_Pin_3); // Pin 3 (Potentiometer)
    Pot_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &Pot_InitStructure);     // Initialize

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT; // General purpose
output
    GPIO_InitStructure.GPIO_Pin = (GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2
| GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8); // LED and Shift Register pins
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure); // Initialize

    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_ExternalTrigConv = DISABLE;
    ADC_InitStructure.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
    ADC_InitStructure.ADC_NbrOfConversion = 1;
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;
    ADC_Init(ADC1, &ADC_InitStructure);  // Initialize

    ADC_Cmd(ADC1, ENABLE); // Start ADC

    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1,
ADC_SampleTime_84Cycles);
    ADC_SoftwareStartConv(ADC1); // Start conversion
}
```