ECE 455

Traffic Light System

Design and Implementation of a Real-Time Traffic Controller

Course: Real Time Computer Systems Design Project
Term: Spring 2026
Instructor: Amirali Baniasadi
Project No: 01

Project Team

John Soliman (V01028377)
David Emelu (V01025755)

University of Victoria

Department of Electrical & Computer Engineering

March 3, 2026

Contents

1 Introduction **1**

2 Design Solution **1**
2.1 System Overview Architecture 1

3 Middleware Design and Implementation **3**
3.1 GPIO Configuration 3
3.2 ADC Configuration 5
3.3 Shift Register Abstraction 6
3.4 FreeRTOS Primitives 7

4 Software Architecture **8**
4.1 Task Responsibilities 8
4.2 Software Task Summary 10

5 Algorithmic Flowcharts **12**
5.1 Traffic Generating Algorithm 12
5.2 System Display Algorithm 13

6 Discussion, Limitations, and Summary **13**
6.1 Testing Methodology & Challenges 13
6.2 Limitations 13
6.3 Summary 14

A Source Code **16**

List of Figures

1 FreeRTOS Task and Data Flow Architecture for Traffic Light Controller 2
2 Hardware design 3
3 Flowchart of the Traffic Generating Algorithm (`Create_Traffic_Task`). 12
4 Flowchart of the System Display Algorithm (`Display_Traffic_Task`). 13

List of Tables

1 Complete GPIO Pin Configuration (`main.c` Implementation) 4

1 Introduction

The objective of this project is to design and implement a real-time Traffic Light System (TLS) using the FreeRTOS real-time operating system on the STM32F4 Discovery platform (STM32F407VG). The system simulates a single-lane, one-way road intersection where traffic flow is dynamically controlled by user input, as defined in the Project 1 specification [1]. The project demonstrates core real-time concepts including deterministic task scheduling, inter-task communication via queues, software timer management, and hardware abstraction layer (HAL) design.

- **System Environment:** The firmware is developed in C using the Atollic TrueSTUDIO IDE. The target hardware is the STM32F4 Discovery board featuring an ARM Cortex-M4 processor running at 168 MHz. FreeRTOS v9 is utilized to manage task concurrency and resource sharing, consistent with the project requirements [1].
- **Functional Objectives:** The system generates pseudo-random traffic patterns proportional to a potentiometer input, visualizes vehicle movement on a linear LED array driven by shift registers, and controls a standard Red-Amber-Green traffic light. The potentiometer must simultaneously influence both traffic generation rate and light timing as required in the project manual [1].
- **Key Requirements:**
  - **Concurrency:** The application is partitioned into independent FreeRTOS tasks as required by the lab and project specifications [2].
  - **Communication:** All data exchange between tasks occurs via FreeRTOS queues; global variables for inter-task communication are strictly prohibited [2].
  - **Timing:** Traffic light state transitions are managed exclusively by FreeRTOS software timers [2].

2 Design Solution

2.1 System Overview Architecture

The system implements the recommended four-task architecture communicating through five FreeRTOS queues and three one-shot software timers for light phases, as outlined in the project manual [1].

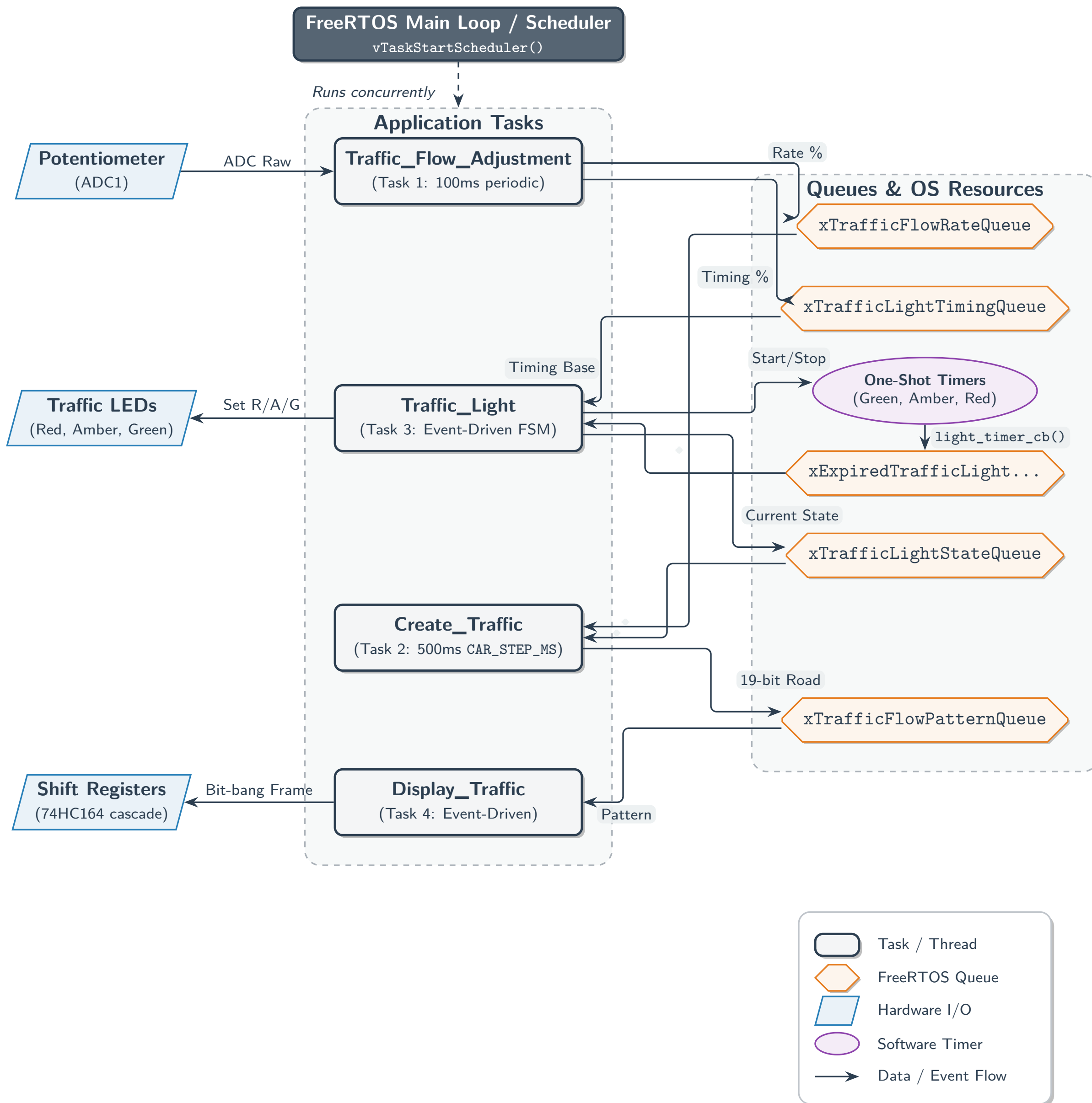

Figure 1: FreeRTOS Task and Data Flow Architecture for Traffic Light Controller

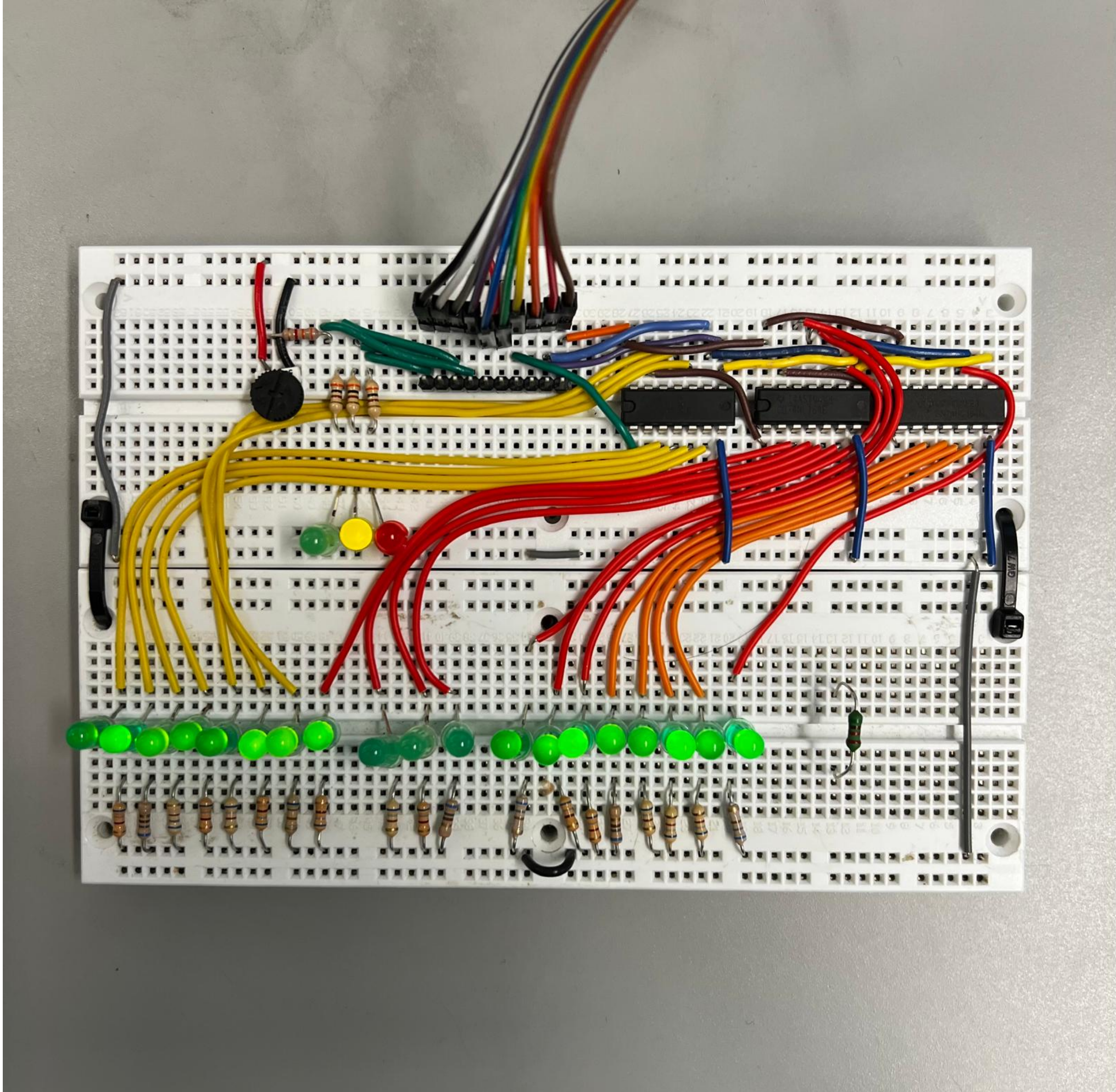

Figure 2: Hardware design

3 Middleware Design and Implementation

The middleware serves as a hardware abstraction layer (HAL), isolating the application logic from direct register access as recommended in the project documentation [1].

3.1 GPIO Configuration

All General Purpose Input/Output (GPIO) pins are located on Port C according to the project pin mapping reference [1]. Before configuring the pins, the AHB1 peripheral clock for GPIOC was enabled using `RCC_AHB1PeriphClockCmd`. This step is required because STM32 peripherals must be clocked before register access is permitted.

Table 1: Complete GPIO Pin Configuration (`main.c` Implementation)

Pin	Port	Mode	OType	PuPd	Function / Hardware
PC0	GPIOC	Output	Push-Pull	No Pull	Red Traffic Light LED
PC1	GPIOC	Output	Push-Pull	No Pull	Amber Traffic Light LED
PC2	GPIOC	Output	Push-Pull	No Pull	Green Traffic Light LED
PC3	GPIOC	Analog	N/A	No Pull	Potentiometer (ADC1 Channel 13)
PC6	GPIOC	Output	Push-Pull	No Pull	Shift Register Serial Data
PC7	GPIOC	Output	Push-Pull	No Pull	Shift Register Clock
PC8	GPIOC	Output	Push-Pull	No Pull	Shift Register Reset

```
static void prvSetupHardware(void)
{
    NVIC_SetPriorityGrouping(0);

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    GPIO_InitTypeDef GPIO_struct;

    /* LEDs + shift register control pins */
    GPIO_struct.GPIO_Pin = red | amber | green | data | clock | reset;
    GPIO_struct.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_struct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_struct.GPIO_OType = GPIO_OType_PP;
    GPIO_struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_struct);

    /* Pot as analog input */
    GPIO_struct.GPIO_Pin = pot;
    GPIO_struct.GPIO_Mode = GPIO_Mode_AN;
    GPIO_struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOC, &GPIO_struct);

    /* ADC setup */
    ADC_InitTypeDef pot_ADC_struct;
    ADC_StructInit(&pot_ADC_struct);
    ADC_Init(ADC1, &pot_ADC_struct);

    ADC_Cmd(ADC1, ENABLE);
    ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1,
                             ADC_SampleTime_144Cycles);
}

```

Listing 1: Hardware Initialization Function

Configuration Explanation

To begin utilizing the GPIO pins, they were first initialized within the hardware setup routine. This process began by enabling the GPIOC clock on the AHB1 bus. A `GPIO_InitTypeDef` structure was then created to configure the required pin properties.

The LED pins (PC0-PC2) and shift-register control lines (PC6-PC8) were configured together

since they share identical characteristics: output mode, push-pull output type, 50 MHz speed, and no internal pull-up or pull-down resistors. Push-pull mode was selected because the LEDs and shift register require actively driven logic-high and logic-low levels. Open-drain mode was not used, as it would require external pull-up resistors and is unnecessary for this application [1].

The potentiometer input (PC3) was configured separately due to its analog measurement requirement. It was set to analog mode with no internal pull-up or pull-down resistors to prevent distortion of the measured voltage. Enabling any internal resistor would introduce an unintended voltage divider effect and reduce ADC accuracy [1]. The pin was then connected to ADC1 Channel 13 and configured with a sampling time of 144 cycles to ensure stable conversion results.

Once the structure fields were defined appropriately, invoking `GPIO_Init()` completed the initialization process.

3.2 ADC Configuration

ADC1 is enabled on the APB2 bus using:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

The potentiometer is connected to PC3, which is configured in analog mode:

```
GPIO_struct.GPIO_Pin = pot;          /* pot = GPIO_Pin_3 */
GPIO_struct.GPIO_Mode = GPIO_Mode_AN;
GPIO_Init(GPIOC, &GPIO_struct);
```

ADC1 is configured to sample Channel 13 (which corresponds to PC3):

```
ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1,
                         ADC_SampleTime_144Cycles);
```

ADC conversions are started manually in software using:

```
ADC_SoftwareStartConv(ADC1);
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) != 1) {}
return ADC_GetConversionValue(ADC1);
```

This conversion routine is called inside the `Traffic_Flow_Adjustment_Task`, ensuring that sampling occurs periodically within the task context:

```
uint16_t adc = (uint16_t)get_ADC_val();
uint16_t pct = adc_to_pct(adc);
```

The returned ADC value is treated as a 12-bit quantity in the range 0 to 4095, as shown in the scaling function:

```
if (adc > 4095) adc = 4095;
return (uint16_t)((adc * 100u) / 4095u);
```

The scaled percentage value is then published to the inter-task queues to control both vehicle spawn rate and traffic light timing.

3.3 Shift Register Abstraction

The shift-register interface is implemented using three GPIO control signals defined as:

```
#define data    GPIO_Pin_6
#define clock   GPIO_Pin_7
#define reset   GPIO_Pin_8
```

All three signals are configured as push-pull outputs in the hardware initialization routine:

```
GPIO_struct.GPIO_Pin = red | amber | green | data | clock | reset;
GPIO_struct.GPIO_Mode = GPIO_Mode_OUT;
GPIO_struct.GPIO_OType = GPIO_OType_PP;
GPIO_Init(GPIOC, &GPIO_struct);
```

The system defines a 24-bit shift frame:

```
#define SHIFT_BITS 24
```

The road model consists of 19 LEDs:

```
#define ROAD_LEN 19
```

Two bit positions are explicitly reserved as cascade link bits and are not used as LED outputs:

```
#define CASCADE_GAP_BIT1  7
#define CASCADE_GAP_BIT2  15
```

A static lookup table maps the 19 logical road indices to their corresponding shift-register bit positions:

```
static const uint8_t ROAD2BIT[ROAD_LEN] = {
    0, 1, 2, 3, 4, 5, 6,
    8, 9,10,11,12,13,14,
   16,17,18,19,20
};
```

During display updates, the 19-bit road pattern is converted into a 24-bit frame while explicitly forcing cascade bits low:

```
uint8_t frame[SHIFT_BITS] = {0};

frame[CASCADE_GAP_BIT1] = 0;
frame[CASCADE_GAP_BIT2] = 0;

for (int r = 0; r < ROAD_LEN; r++) {
    int b = road_to_shiftbit(r);
    frame[b] = (pattern & (1u << (uint32_t)r)) ? 1u : 0u;
}
```

The frame is then shifted out serially using a software bit-banged routine:

```
for (int bit = SHIFT_BITS - 1; bit >= 0; bit--) {
    shift_reg(frame[bit]);
}
```

The `shift_reg()` function drives the data and clock lines directly:

```
GPIO_ResetBits(GPIOC, clock);
GPIO_SetBits(GPIOC, clock);
```

The reset line clears the chained registers before shifting:

```
GPIO_ResetBits(GPIOC, reset);
GPIO_SetBits(GPIOC, reset);
```

3.4 FreeRTOS Primitives

All queues are created with a fixed length defined as:

```
#define mainQUEUE_LENGTH 5
```

Queue handles are declared for inter-task communication:

```
static xQueueHandle xTrafficFlowRateQueue_handle = 0;
static xQueueHandle xTrafficLightTimingQueue_handle = 0;
static xQueueHandle xTrafficLightStateQueue_handle = 0;
static xQueueHandle xExpiredTrafficLightStateQueue_handle = 0;
static xQueueHandle xTrafficFlowPatternQueue_handle = 0;
```

Each queue is created with length `mainQUEUE_LENGTH`:

```
xTrafficFlowRateQueue_handle =
    xQueueCreate(mainQUEUE_LENGTH, sizeof(uint16_t));
```

To ensure that the most recent value is always stored even when the queue is full, a helper wrapper function is implemented:

```
static void q_send_latest_u16(xQueueHandle q, uint16_t v)
{
    if (xQueueSend(q, &v, 0) != pdPASS) {
        uint16_t dump;
        (void)xQueueReceive(q, &dump, 0);
        (void)xQueueSend(q, &v, 0);
    }
}
```

This function attempts a non-blocking send. If the queue is full, the oldest value is removed using `xQueueReceive()` before pushing the new value.
Similarly, the most recent queue value can be retrieved using:

```
static uint16_t q_drain_u16_latest(xQueueHandle q, uint16_t fallback)
{
    uint16_t v = fallback, tmp;
    while (xQueueReceive(q, &tmp, 0) == pdPASS) v = tmp;
    return v;
}
```

This routine drains all pending entries and returns only the most recent value, ensuring tasks operate on the latest system state.

4 Software Architecture

The application is structured around four FreeRTOS tasks created in `main()` using:

```
xTaskCreate(Traffic_Flow_Adjustment_Task, "Traffic_Flow_Adjustment",
            configMINIMAL_STACK_SIZE, NULL, 1, NULL);
xTaskCreate(Create_Traffic_Task, "Create_Traffic",
            configMINIMAL_STACK_SIZE, NULL, 1, NULL);
xTaskCreate(Traffic_Light_Task, "Traffic_Light",
            configMINIMAL_STACK_SIZE, NULL, 1, NULL);
xTaskCreate(Display_Traffic_Task, "Display_Traffic",
            configMINIMAL_STACK_SIZE, NULL, 1, NULL);
```

Each task is assigned priority level 1, as indicated by the final parameter in `xTaskCreate(..., 1, NULL)`. Because all tasks share the same priority, scheduling occurs when a task voluntarily blocks or delays execution.

This cooperative yielding behavior is visible throughout the code via blocking FreeRTOS calls such as:

```
vTaskDelay(pdMS_TO_TICKS(100));
vTaskDelayUntil(&last, pdMS_TO_TICKS(CAR_STEP_MS));
xQueueReceive(..., portMAX_DELAY);
```

These blocking calls ensure that no task busy-waits indefinitely, and CPU time is redistributed by the scheduler when a task enters the Blocked state.

4.1 Task Responsibilities

`Traffic_Flow_Adjustment_Task` This task periodically samples the ADC using:

```
uint16_t adc = (uint16_t)get_ADC_val();
uint16_t pct = adc_to_pct(adc);
```

The sampling interval is controlled by:

```
vTaskDelay(pdMS_TO_TICKS(100));
```

This establishes a 100 ms polling period.
The scaled percentage value is then published to two separate queues:

```
q_send_latest_u16(xTrafficFlowRateQueue_handle, pct);
q_send_latest_u16(xTrafficLightTimingQueue_handle, pct);
```

This design decouples traffic density control from light timing control while using the same physical input signal.

`Create_Traffic_Task` This task enforces strict periodic execution using:

```
vTaskDelayUntil(&last, pdMS_TO_TICKS(CAR_STEP_MS));
```

The constant `CAR_STEP_MS` is defined as:

```
#define CAR_STEP_MS 500
```

This guarantees car motion updates every 500 ms.
Before computing motion, the task retrieves the most recent control values using:

```
pot_pct = q_drain_u16_latest(xTrafficFlowRateQueue_handle, pot_pct);
light_state = q_drain_u8_latest(xTrafficLightStateQueue_handle, light_state);
```

Vehicle motion respects the defined stop line index:

```
#define STOP_LINE_IDX 7
...
if ((light_state != GREEN_STATE) && (i == STOP_LINE_IDX)) {
    continue;
}
```

Vehicle spawning gap is computed dynamically using a xorshift32-based pseudo-random number generator:

```
uint8_t gap = gap_from_pct(pot_pct, &rng);
```

The gap function itself contains the xorshift implementation:

```
x ^= x << 13; x ^= x >> 17; x ^= x << 5;
```

Finally, the road state is encoded into a 19-bit integer:

```
for (int i = 0; i < ROAD_LEN; i++) {
    if (cars[i]) pattern |= (1u << (uint32_t)i);
}
```

`Traffic_Light_Task` The light state machine is initialized as GREEN:

```
uint8_t state = GREEN_STATE;
green_control(1);
amber_control(0);
red_control(0);
```

The task blocks indefinitely waiting for a timer-expiration event:

```
xQueueReceive(xExpiredTrafficLightStateQueue_handle,
              &expired_id, portMAX_DELAY);
```

Upon wake-up, it reads the most recent timing percentage:

```
pot_pct = q_drain_u16_latest(xTrafficLightTimingQueue_handle, pot_pct);
```

Phase durations are recalculated using:

```
uint32_t g_ms = green_ms_from_pct(pot_pct);
uint32_t r_ms = red_ms_from_pct(pot_pct);
```

Timers are then reconfigured using:

```
xTimerChangePeriod(green_timer, pdMS_TO_TICKS(g_ms), 0);
xTimerStart(green_timer, 0);
```

This confirms that light timing dynamically depends on the potentiometer percentage.

`Display_Traffic_Task` This task blocks waiting for a road pattern:

```
xQueueReceive(xTrafficFlowPatternQueue_handle,
              &pattern, portMAX_DELAY);
```

The 19-bit pattern is expanded into a 24-bit frame:

```
uint8_t frame[SHIFT_BITS] = {0};
```

Cascade link bits are explicitly forced low:

```
#define CASCADE_GAP_BIT1 7
#define CASCADE_GAP_BIT2 15
...
frame[CASCADE_GAP_BIT1] = 0;
frame[CASCADE_GAP_BIT2] = 0;
```

To prevent concurrent modification during bit shifting, the critical section macros are used:

```
taskENTER_CRITICAL();
...
taskEXIT_CRITICAL();
```

The frame is then shifted out MSB to LSB:

```
for (int bit = SHIFT_BITS - 1; bit >= 0; bit--) {
    shift_reg(frame[bit]);
}
```

This confirms that display output is fully serialized and protected during hardware updates.

4.2 Software Task Summary

The software operates through four coordinated FreeRTOS tasks that collectively implement a real-time traffic simulation. Each task is responsible for a specific subsystem: user input acquisition, traffic generation, traffic light control, and hardware display output. The tasks communicate exclusively through queues, ensuring modular design and clear separation of responsibilities.

`Traffic_Flow_Adjustment_Task` This task acts as the system's input interface. It periodically samples the potentiometer using the ADC and converts the raw 12-bit value into a percentage between 0 and 100. The sampling interval is fixed at 100 ms, establishing a stable and predictable control update rate.

The computed percentage is then published to two independent queues: one controlling vehicle spawn rate and the other controlling traffic light timing. By broadcasting the same scaled input to separate consumers, the design cleanly decouples traffic density control from phase duration control while relying on a single physical input source.

`Create_Traffic_Task` This task implements the traffic flow simulation. It runs with strict periodicity using `vTaskDelayUntil()`, guaranteeing that vehicle movement updates occur every 500 ms. This enforces a constant vehicle speed across the simulation.

At each execution cycle, the task retrieves the most recent traffic density percentage and light state from their respective queues. Cars are then advanced along the 19-LED road model while respecting two constraints: collision avoidance (a car cannot move into an occupied position) and stop-line enforcement (vehicles at index 7 remain stationary during RED or AMBER phases).

Vehicle spawning is controlled through a dynamically computed gap. A lightweight xorshift32 pseudo-random generator introduces controlled variability in spawn intervals, preventing unrealistic uniform traffic spacing. Higher potentiometer percentages reduce the gap, resulting in denser traffic flow.

After updating positions, the task encodes the road state into a 19-bit integer pattern and transmits it to the display task via a queue.

`Traffic_Light_Task` This task implements the traffic light finite state machine (GREEN → AMBER → RED → GREEN). It initializes in the GREEN state and controls the hardware LEDs accordingly.

Rather than polling, the task blocks indefinitely on a queue that receives expiration events from FreeRTOS software timers. When a phase expires, the task retrieves the latest potentiometer percentage and recalculates the next phase duration.

Green duration scales proportionally with the percentage, while Red duration scales inversely. Amber remains fixed at 2000 ms. The appropriate timer is then restarted with the updated period. This structure ensures deterministic phase transitions while allowing real-time user control over light timing.

`Display_Traffic_Task` The display task is responsible for hardware serialization. It blocks until a new 19-bit road pattern is received from the traffic task.

The logical road model is expanded into a 24-bit shift-register frame. Two specific bit positions (7 and 15) are explicitly forced low to accommodate cascade link connections in the chained registers.

To prevent race conditions during hardware updates, the shifting routine is enclosed within a FreeRTOS critical section. The frame is then bit-banged from MSB to LSB into the chained shift registers, ensuring that LED outputs are updated atomically and without interruption.

Overall Architecture Behavior Together, these four tasks form a deterministic, event-driven system. Input sampling, vehicle motion, phase timing, and hardware output are fully compartmentalized. All inter-task communication occurs through bounded queues, and blocking calls are used deliberately to avoid busy waiting. The result is a modular traffic simulation that responds to user input in real time while maintaining predictable scheduling and safe hardware access.

5 Algorithmic Flowcharts

To clearly document the operational logic of the traffic movement and display synchronization, two flowcharts are provided representing the core algorithms [2].

5.1 Traffic Generating Algorithm

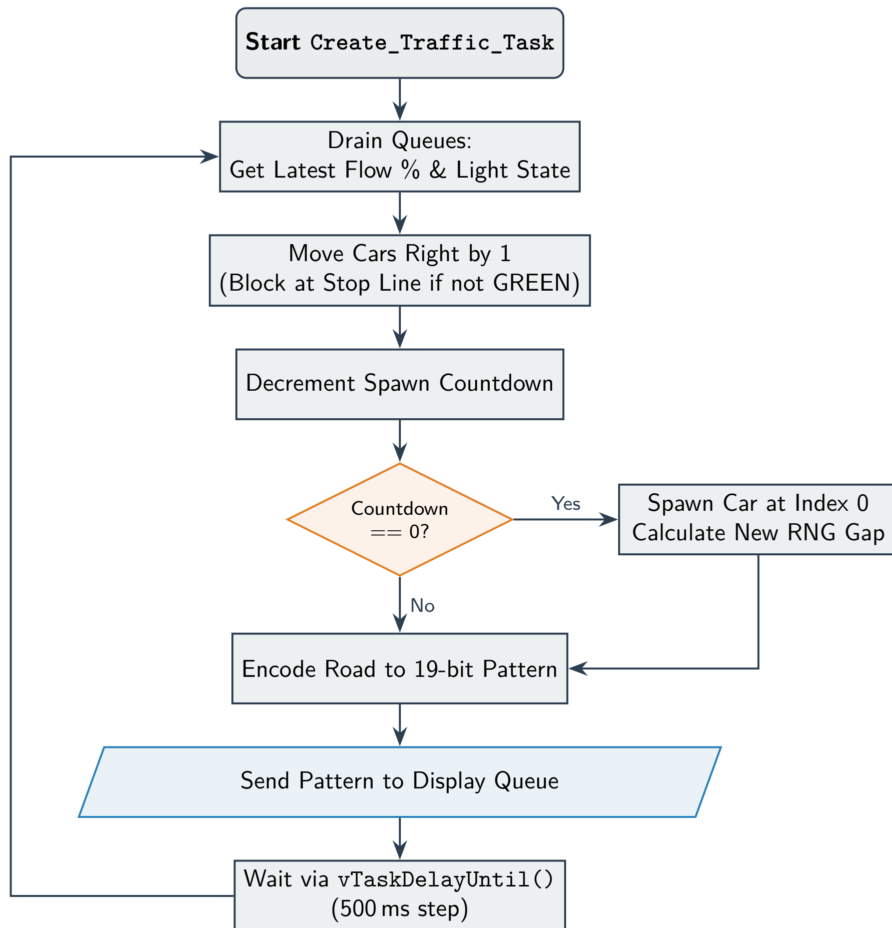

Figure 3: Flowchart of the Traffic Generating Algorithm (`Create_Traffic_Task`).

5.2 System Display Algorithm

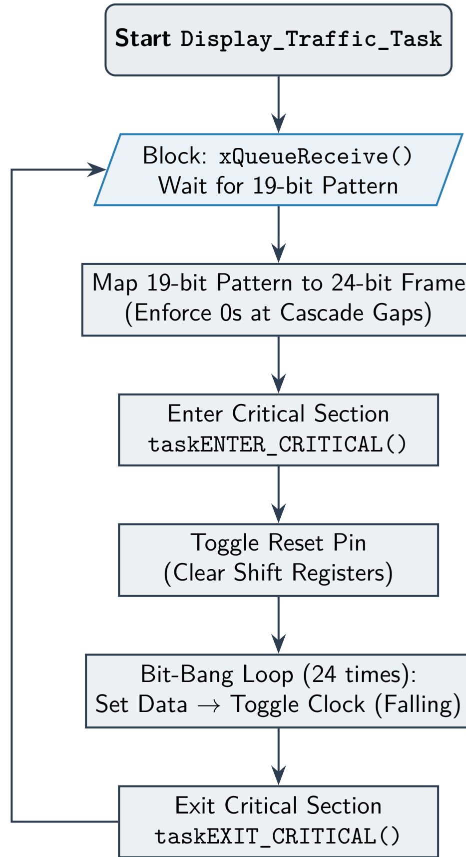

Figure 4: Flowchart of the System Display Algorithm (`Display_Traffic_Task`).

6 Discussion, Limitations, and Summary

6.1 Testing Methodology & Challenges

Testing followed the integration process recommended in the lab manual [2]. A primary challenge encountered was ensuring stale data did not queue up and delay hardware responsiveness. This was resolved by creating custom queue wrappers that overwrite the oldest queue items. Furthermore, proper interrupt abstraction was maintained: the timer callback `light_timer_cb` executes in the Timer Daemon task context and avoids blocking calls, correctly signaling the FSM via a zero-delay queue send [1].

6.2 Limitations

While functional, the current implementation polls the ADC utilizing a blocking `while` loop waiting for the EOC flag. While brief, this blocks the `Traffic_Flow_Adjustment_Task`. A future improvement would utilize an ADC End-of-Conversion Interrupt Service Routine (ISR) paired with a binary semaphore to yield the CPU during conversion.

6.3 Summary

This Traffic Light System successfully demonstrates the application of FreeRTOS principles task concurrency, queue-based synchronization, and software timer management. The architecture strictly complies with the specifications outlined in the ECE 455 Project 1 documentation [1] and the accompanying Lab Manual [2].

References

[1] A. Sepahi, *Project 1: Traffic Light System*, 2025. Student Manual. ECE 455 B04 Real-Time Computer Systems. STM32F4 Discovery Board, FreeRTOS v9, GPIO, ADC & Shift Registers.

[2] A. Jooya, S. Khoshbakht, N. Agarwal, S. Campbell, N. J. Dimopoulos, *et al.*, *ECE 455: Real Time Computer Systems Design Project*, 2019. Accessed Feb. 26, 2026. [Online].

A Source Code

```
#include <stdint.h>
#include <stdio.h>
#include "stm32f4_discovery.h"

#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"

/*-----------------------------------------------------------*/
/* Pin mapping (this matches how I wired it on the STM32 + breadboard) */
#define red     GPIO_Pin_0
#define amber   GPIO_Pin_1
#define green   GPIO_Pin_2

#define pot     GPIO_Pin_3

#define data    GPIO_Pin_6
#define clock   GPIO_Pin_7
#define reset   GPIO_Pin_8

/* Traffic light states */
#define RED_STATE    0
#define AMBER_STATE  1
#define GREEN_STATE  2

/*-----------------------------------------------------------*/
/* Project constants */
#define SHIFT_BITS        24      /* 3 shift registers x 8 bits */
#define ROAD_LEN          19      /* 19 physical LEDs for cars */

/*
  I'm chaining 74HC164s using Q7 as the serial-out into the next chip.
  That means Q7 of reg1 and Q7 of reg2 are NOT usable LED outputs (they're "link" bits),
  so I treat those bit positions like "gaps" in software.
*/
#define CASCADE_GAP_BIT1  7       /* reg1 Q7 -> reg2 IN */
#define CASCADE_GAP_BIT2  15      /* reg2 Q7 -> reg3 IN */

/*
  Stop line: cars must stop here when the light is RED or AMBER.
  With my LED layout, physical LED8 is where the car should stop.
  Since cars[] is 0-based, that stop LED is index 7.
*/
#define STOP_LINE_IDX     7

/* Car movement speed: 500ms per step = 2 LEDs/sec (within 1-3 LEDs/sec requirement) */
#define CAR_STEP_MS       500

/* Light timings */
#define AMBER_MS          2000
#define GREEN_MIN_MS      2500
```

```
55  #define GREEN_MAX_MS      5000
56  #define RED_MIN_MS        2500
57  #define RED_MAX_MS        5000
58
59  /* The manual uses a standard queue length (I kept it consistent here) */
60  #define mainQUEUE_LENGTH  5
61
62  /*-----------------------------------------------------------*/
63  /* Hardware setup + helpers */
64  static void prvSetupHardware(void);
65
66  static int  get_ADC_val(void);
67  static void clear_lane(void);
68  static void red_control(int value);
69  static void amber_control(int value);
70  static void green_control(int value);
71  static void shift_reg(int value);
72
73  /* Helper math for converting pot -> % and % -> timing */
74  static uint16_t adc_to_pct(uint16_t adc);
75  static uint32_t green_ms_from_pct(uint16_t pct);
76  static uint32_t red_ms_from_pct(uint16_t pct);
77  static uint8_t  gap_from_pct(uint16_t pct, uint32_t *rng);
78
79  /*-----------------------------------------------------------*/
80  /* Tasks (4 total, like the manual's structure) */
81  static void Traffic_Flow_Adjustment_Task(void *pvParameters); /* Task 1: read pot */
82  static void Create_Traffic_Task(void *pvParameters);          /* Task 2: move/spawn cars */
83  static void Traffic_Light_Task(void *pvParameters);           /* Task 3: traffic light FSM */
84  static void Display_Traffic_Task(void *pvParameters);         /* Task 4: shift out LEDs */
85
86  /* 3 one-shot timers: one for each light phase */
87  static void light_timer_cb(xTimerHandle xTimer);
88
89  /*-----------------------------------------------------------*/
90  /* 5 queues (manual-style inter-task communication) */
91
92  /* 1) pot -> Create_Traffic (controls spawn rate) */
93  static xQueueHandle xTrafficFlowRateQueue_handle = 0;         /* uint16_t pct */
94
95  /* 2) pot -> Traffic_Light (controls green/red durations) */
96  static xQueueHandle xTrafficLightTimingQueue_handle = 0;      /* uint16_t pct */
97
98  /* 3) Traffic_Light -> Create_Traffic (current light state) */
99  static xQueueHandle xTrafficLightStateQueue_handle = 0;       /* uint8_t state */
100
101 /* 4) Timer callbacks -> Traffic_Light (phase expired event) */
102 static xQueueHandle xExpiredTrafficLightStateQueue_handle = 0;/* uint8_t expired id */
103
104 /* 5) Create_Traffic -> Display_Traffic (19-bit road pattern) */
105 static xQueueHandle xTrafficFlowPatternQueue_handle = 0;      /* uint32_t bits 0..18 */
106
107 /* Timers */
108 static xTimerHandle green_timer = 0;
109 static xTimerHandle amber_timer = 0;
110 static xTimerHandle red_timer   = 0;
111
```

```c
112 /*---------------------------------------------------------*/
113 /*
114   Mapping from my "road index" (0..18) to the actual shift-register bit index (0..23).
115   Because of the cascade gaps, the physical LEDs are not perfectly contiguous in the 24-bit ↩
      chain.
116 */
117 static const uint8_t ROAD2BIT[ROAD_LEN] = {
118     0, 1, 2, 3, 4, 5, 6,       /* LED1..LED7   -> reg1 Q0..Q6 */
119     8, 9,10,11,12,13,14,       /* LED8..LED14  -> reg2 Q0..Q6 */
120    16,17,18,19,20              /* LED15..LED19 -> reg3 Q0..Q4 */
121 };
122
123 static inline int road_to_shiftbit(int road_idx)
124 {
125     return ROAD2BIT[road_idx];
126 }
127
128 /*---------------------------------------------------------*/
129 /*
130   These helpers let me "send the latest value" even though the queues have fixed length.
131   If the queue is full, I pop one old value and push the new one.
132 */
133 static void q_send_latest_u16(xQueueHandle q, uint16_t v)
134 {
135     if (xQueueSend(q, &v, 0) != pdPASS) {
136         uint16_t dump;
137         (void)xQueueReceive(q, &dump, 0);
138         (void)xQueueSend(q, &v, 0);
139     }
140 }
141
142 static void q_send_latest_u8(xQueueHandle q, uint8_t v)
143 {
144     if (xQueueSend(q, &v, 0) != pdPASS) {
145         uint8_t dump;
146         (void)xQueueReceive(q, &dump, 0);
147         (void)xQueueSend(q, &v, 0);
148     }
149 }
150
151 static void q_send_latest_u32(xQueueHandle q, uint32_t v)
152 {
153     if (xQueueSend(q, &v, 0) != pdPASS) {
154         uint32_t dump;
155         (void)xQueueReceive(q, &dump, 0);
156         (void)xQueueSend(q, &v, 0);
157     }
158 }
159
160 /* Drain a queue to get the most recent value (I use this when I only care about the latest ↩
      state) */
161 static uint16_t q_drain_u16_latest(xQueueHandle q, uint16_t fallback)
162 {
163     uint16_t v = fallback, tmp;
164     while (xQueueReceive(q, &tmp, 0) == pdPASS) v = tmp;
165     return v;
166 }
```

```
167
168 static uint8_t q_drain_u8_latest(xQueueHandle q, uint8_t fallback)
169 {
170     uint8_t v = fallback, tmp;
171     while (xQueueReceive(q, &tmp, 0) == pdPASS) v = tmp;
172     return v;
173 }
174
175 /*-----------------------------------------------------------*/
176
177 int main(void)
178 {
179     prvSetupHardware();
180     clear_lane(); /* make sure the shift register outputs start cleared */
181
182     /* Create queues (all length = 5) */
183     xTrafficFlowRateQueue_handle          = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint16_t));
184     xTrafficLightTimingQueue_handle       = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint16_t));
185     xTrafficLightStateQueue_handle        = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
186     xExpiredTrafficLightStateQueue_handle = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint8_t));
187     xTrafficFlowPatternQueue_handle       = xQueueCreate(mainQUEUE_LENGTH, sizeof(uint32_t));
188
189 #if (configQUEUE_REGISTRY_SIZE > 0)
190     /* Helps the FreeRTOS debug view label queues nicely */
191     vQueueAddToRegistry(xTrafficFlowRateQueue_handle,          "Q_FlowRate");
192     vQueueAddToRegistry(xTrafficLightTimingQueue_handle,       "Q_LightTiming");
193     vQueueAddToRegistry(xTrafficLightStateQueue_handle,        "Q_LightState");
194     vQueueAddToRegistry(xExpiredTrafficLightStateQueue_handle, "Q_LightExpired");
195     vQueueAddToRegistry(xTrafficFlowPatternQueue_handle,       "Q_RoadPattern");
196 #endif
197
198     /* Create 3 one-shot timers (one per light phase) */
199     green_timer = xTimerCreate("Tgreen", pdMS_TO_TICKS(1000), pdFALSE, (void*)GREEN_STATE, ↩
        light_timer_cb);
200     amber_timer = xTimerCreate("Tamber", pdMS_TO_TICKS(1000), pdFALSE, (void*)AMBER_STATE, ↩
        light_timer_cb);
201     red_timer   = xTimerCreate("Tred",   pdMS_TO_TICKS(1000), pdFALSE, (void*)RED_STATE,   ↩
        light_timer_cb);
202
203     /* Create tasks */
204     xTaskCreate(Traffic_Flow_Adjustment_Task, "Traffic_Flow_Adjustment", ↩
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
205     xTaskCreate(Create_Traffic_Task,          "Create_Traffic",          ↩
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
206     xTaskCreate(Traffic_Light_Task,           "Traffic_Light",           ↩
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
207     xTaskCreate(Display_Traffic_Task,         "Display_Traffic",         ↩
        configMINIMAL_STACK_SIZE, NULL, 1, NULL);
208
209     vTaskStartScheduler();
210     return 0;
211 }
212
213 /*-----------------------------------------------------------*/
214 /* TASK 1: Read the pot (ADC) and publish a 0..100% value.
215    I send the same % into two queues:
216    - one that controls car spawn rate
```

```
217        - one that controls light timing
218    */
219    static void Traffic_Flow_Adjustment_Task(void *pvParameters)
220    {
221        (void)pvParameters;
222
223        while (1) {
224            uint16_t adc = (uint16_t)get_ADC_val();
225            uint16_t pct = adc_to_pct(adc); /* 0..100 */
226
227            q_send_latest_u16(xTrafficFlowRateQueue_handle, pct);
228            q_send_latest_u16(xTrafficLightTimingQueue_handle, pct);
229
230            vTaskDelay(pdMS_TO_TICKS(100));
231        }
232    }
233
234    /*-----------------------------------------------------------*/
235    /* TASK 2: Create_Traffic
236       - Moves cars one LED every CAR_STEP_MS (constant speed)
237       - Spawns cars randomly with a pot-controlled gap (higher pot => smaller gap)
238       - Stops cars at the stop line when the light is RED or AMBER
239       - Outputs the road state as a 19-bit pattern to the display task
240    */
241    static void Create_Traffic_Task(void *pvParameters)
242    {
243        (void)pvParameters;
244
245        TickType_t last = xTaskGetTickCount();
246
247        int cars[ROAD_LEN] = {0};
248
249        uint16_t pot_pct = 0;
250        uint8_t  light_state = GREEN_STATE;
251
252        /* Simple RNG seed (good enough for random-ish spacing) */
253        uint32_t rng = ((uint32_t)xTaskGetTickCount() << 16) ^ (uint32_t)get_ADC_val() ^ 0xA5A5u;
254        uint8_t countdown = 0;
255
256        while (1) {
257            /* Grab the latest pot value for traffic flow */
258            pot_pct = q_drain_u16_latest(xTrafficFlowRateQueue_handle, pot_pct);
259
260            /* Grab the latest light state */
261            light_state = q_drain_u8_latest(xTrafficLightStateQueue_handle, light_state);
262
263            /* ---- Move cars right by 1 (iterate from right to left so cars don't double-move) ↩
           ---- */
264            for (int i = ROAD_LEN - 1; i >= 0; i--) {
265                if (cars[i] == 0) continue;
266
267                /* car leaves the road at the far right */
268                if (i == ROAD_LEN - 1) {
269                    cars[i] = 0;
270                    continue;
271                }
272
```

```
273             /* don't run into the car in front */
274             if (cars[i + 1] == 1) continue;
275
276             /* stop-line rule: the stop-line car can't advance unless green */
277             if ((light_state != GREEN_STATE) && (i == STOP_LINE_IDX)) {
278                 continue;
279             }
280
281             cars[i] = 0;
282             cars[i + 1] = 1;
283         }
284
285         /* ---- Spawn cars with pot-based random spacing ---- */
286         if (countdown > 0) countdown--;
287
288         if (countdown == 0) {
289             if (cars[0] == 0) {
290                 cars[0] = 1;
291             }
292             /* if cars[0] is occupied, I just drop the spawn (traffic jam) */
293
294             uint8_t gap = gap_from_pct(pot_pct, &rng); /* 0..6 */
295             countdown = gap; /* 0 => spawn every tick (bumper-to-bumper at max pot) */
296         }
297
298         /* ---- Pack road into a 19-bit pattern and send to display task ---- */
299         uint32_t pattern = 0;
300         for (int i = 0; i < ROAD_LEN; i++) {
301             if (cars[i]) pattern |= (1u << (uint32_t)i);
302         }
303         q_send_latest_u32(xTrafficFlowPatternQueue_handle, pattern);
304
305         vTaskDelayUntil(&last, pdMS_TO_TICKS(CAR_STEP_MS));
306     }
307 }
308
309 /*-----------------------------------------------------------*/
310 /* TASK 3: Traffic_Light
311    This is my light state machine:
312    GREEN -> AMBER -> RED -> GREEN ...
313    Timers:
314    - GREEN duration scales with pot (%)
315    - RED duration inversely scales with pot (%)
316    - AMBER is constant
317 */
318 static void Traffic_Light_Task(void *pvParameters)
319 {
320     (void)pvParameters;
321
322     uint8_t state = GREEN_STATE;
323     uint16_t pot_pct = 0;
324
325     /* Start green */
326     green_control(1);
327     amber_control(0);
328     red_control(0);
329     q_send_latest_u8(xTrafficLightStateQueue_handle, state);
```

```
330
331     /* Start green timer based on the current pot */
332     pot_pct = q_drain_u16_latest(xTrafficLightTimingQueue_handle, pot_pct);
333     uint32_t g_ms = green_ms_from_pct(pot_pct);
334
335     xTimerStop(red_timer, 0);
336     xTimerStop(amber_timer, 0);
337     xTimerChangePeriod(green_timer, pdMS_TO_TICKS(g_ms), 0);
338     xTimerStart(green_timer, 0);
339
340     while (1) {
341         /* Block until one of the timers expires */
342         uint8_t expired_id;
343         if (xQueueReceive(xExpiredTrafficLightStateQueue_handle, &expired_id, portMAX_DELAY) ↩
        != pdPASS) {
344             continue;
345         }
346
347         /* Update pot so next phase uses the newest value */
348         pot_pct = q_drain_u16_latest(xTrafficLightTimingQueue_handle, pot_pct);
349
350         if (state == GREEN_STATE) {
351             /* GREEN -> AMBER */
352             green_control(0);
353             amber_control(1);
354             red_control(0);
355
356             state = AMBER_STATE;
357             q_send_latest_u8(xTrafficLightStateQueue_handle, state);
358
359             xTimerStop(green_timer, 0);
360             xTimerStop(red_timer, 0);
361             xTimerChangePeriod(amber_timer, pdMS_TO_TICKS(AMBER_MS), 0);
362             xTimerStart(amber_timer, 0);
363         }
364         else if (state == AMBER_STATE) {
365             /* AMBER -> RED */
366             green_control(0);
367             amber_control(0);
368             red_control(1);
369
370             state = RED_STATE;
371             q_send_latest_u8(xTrafficLightStateQueue_handle, state);
372
373             uint32_t r_ms = red_ms_from_pct(pot_pct);
374
375             xTimerStop(amber_timer, 0);
376             xTimerStop(green_timer, 0);
377             xTimerChangePeriod(red_timer, pdMS_TO_TICKS(r_ms), 0);
378             xTimerStart(red_timer, 0);
379         }
380         else {
381             /* RED -> GREEN */
382             green_control(1);
383             amber_control(0);
384             red_control(0);
385
```

```
            state = GREEN_STATE;
            q_send_latest_u8(xTrafficLightStateQueue_handle, state);

            uint32_t g2_ms = green_ms_from_pct(pot_pct);

            xTimerStop(red_timer, 0);
            xTimerStop(amber_timer, 0);
            xTimerChangePeriod(green_timer, pdMS_TO_TICKS(g2_ms), 0);
            xTimerStart(green_timer, 0);
        }
    }
}

    /* Timer callback: I keep this super short-just push an "expired" event into a queue */
static void light_timer_cb(xTimerHandle xTimer)
{
    uint8_t id = (uint8_t)(uintptr_t)pvTimerGetTimerID(xTimer); /* which phase expired */
    (void)xQueueSend(xExpiredTrafficLightStateQueue_handle, &id, 0);
}

/*-----------------------------------------------------------*/
/* TASK 4: Display_Traffic
   Receives a 19-bit road pattern, maps it into 24 shift bits (with cascade gaps),
   and shifts it out to the 3 chained 74HC164s.
*/
static void Display_Traffic_Task(void *pvParameters)
{
    (void)pvParameters;

    uint32_t pattern = 0;

    while (1) {
        /* Wait for the next road pattern */
        if (xQueueReceive(xTrafficFlowPatternQueue_handle, &pattern, portMAX_DELAY) != pdPASS)
     {
            continue;
        }

        /* Convert the 19-bit road pattern into the 24-bit shift-register frame */
        uint8_t frame[SHIFT_BITS] = {0};

        /* Make sure the cascade link bits stay off */
        frame[CASCADE_GAP_BIT1] = 0;
        frame[CASCADE_GAP_BIT2] = 0;

        for (int r = 0; r < ROAD_LEN; r++) {
            int b = road_to_shiftbit(r);
            frame[b] = (pattern & (1u << (uint32_t)r)) ? 1u : 0u;
        }

        /* Shift out: I'm shifting MSB->LSB because of how the 164 chain lines up */
        taskENTER_CRITICAL();
        clear_lane();
        for (int bit = SHIFT_BITS - 1; bit >= 0; bit--) {
            shift_reg(frame[bit]);
        }
        taskEXIT_CRITICAL();
```

```
442        }
443    }
444
445    /*-----------------------------------------------------------*/
446    /* FreeRTOS hooks (just trap here if something goes wrong) */
447    void vApplicationMallocFailedHook(void) { for(;;); }
448    void vApplicationStackOverflowHook(xTaskHandle pxTask, signed char *pcTaskName)
449    {
450        (void)pcTaskName; (void)pxTask;
451        for(;;);
452    }
453    void vApplicationIdleHook(void) {}
454
455    /*-----------------------------------------------------------*/
456    /* Hardware init */
457    static void prvSetupHardware(void)
458    {
459        NVIC_SetPriorityGrouping(0);
460
461        RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
462        RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
463
464        GPIO_InitTypeDef GPIO_struct;
465
466        /* LEDs + shift register control pins */
467        GPIO_struct.GPIO_Pin = red | amber | green | data | clock | reset;
468        GPIO_struct.GPIO_Mode = GPIO_Mode_OUT;
469        GPIO_struct.GPIO_Speed = GPIO_Speed_50MHz;
470        GPIO_struct.GPIO_OType = GPIO_OType_PP;
471        GPIO_struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
472        GPIO_Init(GPIOC, &GPIO_struct);
473
474        /* Pot as analog input */
475        GPIO_struct.GPIO_Pin = pot;
476        GPIO_struct.GPIO_Mode = GPIO_Mode_AN;
477        GPIO_struct.GPIO_PuPd = GPIO_PuPd_NOPULL;
478        GPIO_Init(GPIOC, &GPIO_struct);
479
480        /* ADC setup */
481        ADC_InitTypeDef pot_ADC_struct;
482        ADC_StructInit(&pot_ADC_struct);
483        ADC_Init(ADC1, &pot_ADC_struct);
484
485        ADC_Cmd(ADC1, ENABLE);
486        ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_144Cycles);
487    }
488
489    /*-----------------------------------------------------------*/
490    /* ADC read (polling) */
491    static int get_ADC_val(void)
492    {
493        ADC_SoftwareStartConv(ADC1);
494        while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) != 1) {}
495        return ADC_GetConversionValue(ADC1);
496    }
497
498    /*-----------------------------------------------------------*/
```

```
499 /* Convert ADC (0..4095) into percentage (0..100) */
500 static uint16_t adc_to_pct(uint16_t adc)
501 {
502     if (adc > 4095) adc = 4095;
503     return (uint16_t)((adc * 100u) / 4095u);
504 }
505
506 /* Green time scales up with % */
507 static uint32_t green_ms_from_pct(uint16_t pct)
508 {
509     if (pct > 100) pct = 100;
510     return GREEN_MIN_MS + ((GREEN_MAX_MS - GREEN_MIN_MS) * (uint32_t)pct) / 100u;
511 }
512
513 /* Red time scales down with % */
514 static uint32_t red_ms_from_pct(uint16_t pct)
515 {
516     if (pct > 100) pct = 100;
517     return RED_MAX_MS - ((RED_MAX_MS - RED_MIN_MS) * (uint32_t)pct) / 100u;
518 }
519
520 /*
521   Spawn gap function:
522   - pct=100 -> gap ~ 0 (new car every tick)
523   - pct=0   -> gap ~ 5 or 6
524   I add a tiny bit of randomness so it doesn't look too "perfect."
525 */
526 static uint8_t gap_from_pct(uint16_t pct, uint32_t *rng)
527 {
528     if (pct > 100) pct = 100;
529     uint32_t inv = 100u - (uint32_t)pct;
530
531     /* 0..11 represents 0..5.5 */
532     uint32_t gap_x2 = (inv * 11u + 50u) / 100u;
533
534     uint8_t base = (uint8_t)(gap_x2 / 2u);
535     uint8_t rem  = (uint8_t)(gap_x2 % 2u);
536
537     /* xorshift32 RNG */
538     uint32_t x = *rng;
539     x ^= x << 13; x ^= x >> 17; x ^= x << 5;
540     *rng = x;
541
542     if (rem && (x & 1u)) base++;
543     if (base > 6u) base = 6u;
544
545     return base;
546 }
547
548 /*-----------------------------------------------------------*/
549 /* Shift register helpers */
550 static void clear_lane(void)
551 {
552     /* Reset line clears the chained registers */
553     GPIO_ResetBits(GPIOC, reset);
554     for (int i = 0; i < 50; i++) {}
555     GPIO_SetBits(GPIOC, reset);
```

```c
556 }
557
558 /* Simple GPIO control for the traffic light LEDs */
559 static void red_control(int value)
560 {
561     if (value) GPIO_SetBits(GPIOC, red);
562     else       GPIO_ResetBits(GPIOC, red);
563 }
564
565 static void amber_control(int value)
566 {
567     if (value) GPIO_SetBits(GPIOC, amber);
568     else       GPIO_ResetBits(GPIOC, amber);
569 }
570
571 static void green_control(int value)
572 {
573     if (value) GPIO_SetBits(GPIOC, green);
574     else       GPIO_ResetBits(GPIOC, green);
575 }
576
577 /*
578   shift_reg(value):
579   Bit-bang one bit into the 74HC164 chain.
580   I'm clocking on the falling edge (based on how my board behaved reliably).
581 */
582 static void shift_reg(int value)
583 {
584     if (value) {
585         GPIO_SetBits(GPIOC, data);
586         GPIO_ResetBits(GPIOC, clock);     /* falling edge clocks in */
587         for (int i = 0; i < 50; i++) {}
588         GPIO_SetBits(GPIOC, clock);
589         GPIO_ResetBits(GPIOC, data);
590     } else {
591         GPIO_ResetBits(GPIOC, data);
592         GPIO_ResetBits(GPIOC, clock);     /* falling edge clocks in */
593         for (int i = 0; i < 50; i++) {}
594         GPIO_SetBits(GPIOC, clock);
595     }
596 }
```

Listing 2: project 1 code