Design and Implementation of an Embedded Traffic Light Control System

Sebastian Segerstein

January 2025

Contents

Summary

This report describes the design and implementation of an embedded system for managing traffic signals at a pedestrian crossing using FreeRTOS. The objective of the project was to ensure synchronized operation of traffic lights for vehicles and pedestrians while handling button-press events and preventing signal conflicts.

The system was developed on an STM32L476 Nucleo-64 microcontroller. FreeRTOS was employed to manage concurrency with tasks for pedestrian logic, vehicle logic, and timing mechanisms. A mutex ensured thread-safe operations on shared resources, and unit testing was used to validate functionality.

The results demonstrate a functioning traffic control system capable of handling real-world scenarios. The project highlights the advantages of modular design and real-time operating systems in embedded development. However, the system has limitations regarding scalability to complex intersections, which are discussed in the report along with suggestions for future work.

Chapter 1

Introduction

Embedded systems are widely used in real-time applications where precise control and timing are critical. This project focuses on designing and implementing a real-time embedded system that simulates a traffic intersection. The entire system is implemented on an STM32 Nucleo-64 microcontroller, with all traffic signals and pedestrian buttons physically mounted on a traffic shield that connects to the microcontroller. The shield contains multiple LEDs representing traffic lights and push buttons simulating pedestrian crossing requests. The microcontroller interfaces with these components via GPIO inputs for buttons and an SPI-controlled shift register system for LEDs.

The main challenge of this project is to design a real-time system that ensures pedestrian and vehicle signals remain synchronized while handling button press events in a timely manner.

Chapter 2

Main Section

2.1 Methods

The system is implemented on an STM32L476 Nucleo-64 microcontroller, with all traffic lights and pedestrian buttons housed on a custom-designed traffic shield. This shield is designed to be physically mounted onto the microcontroller, ensuring a compact and self-contained system. The traffic shield contains multiple LEDs arranged to represent different traffic signals and pedestrian indicators, along with push buttons that serve as pedestrian crossing request inputs.

To control multiple LEDs efficiently, the traffic shield incorporates daisy-chained shift registers, reducing the number of GPIO pins required for LED management. The shift registers are controlled via the SPI protocol, while additional GPIO lines are used for control signals such as enable, clock, and reset.

Testing followed a Test-Driven Development (TDD) approach, where unit tests were written before implementing functionality. Each module was validated individually before integration into the full system. Special attention was given to edge cases.

2.1.1 System Initialization

Before the system begins normal operation, several key components must be initialized. The initialization process ensures that the microcontroller's peripherals, including SPI, GPIO, and FreeRTOS tasks, are configured correctly.

Upon startup, the STM32 HAL (Hardware Abstraction Layer) is used to configure the system clock and initialize peripherals [1]. The SPI interface is set up with an appropriate baud rate to communicate reliably with the

74HC595 shift registers. GPIO pins connected to the pedestrian buttons are configured as input with pull-down resistors to prevent floating states.

After hardware initialization, FreeRTOS is started, creating tasks for pedestrian and vehicle signal control [2]. These tasks run concurrently and execute state transitions based on real-time conditions. A mutex is initialized to protect shared resources within the IntersectionController structure, ensuring safe data access between tasks.

This initialization phase is critical for ensuring that all components operate within their expected parameters before the system begins handling pedestrian and vehicle signals.

2.1.2 Testing Procedures

Testing was performed using Test-Driven Development (TDD) to ensure correctness at each stage of implementation. Each module was gradually developed and validated to ensure correctness and reliability.

2.2 System Architecture

2.2.1 Hardware Architecture

The system is built around an STM32L476 Nucleo-64 microcontroller, which manages traffic lights and pedestrian crossing signals through a custom traffic shield circuit.

To efficiently control multiple LEDs representing traffic lights on the traffic light shield, the system utilizes daisy-chained shift registers. This configuration minimizes the number of GPIO pins required by leveraging the SPI protocol along with a few additional control signals for data input, enable, clock, and reset. By chaining the shift registers, the output of one feeds directly into the input of the succeeding, enabling expansion of output without sacrificing additional GPIO pins.

Pedestrian push buttons are connected to the microcontroller's GPIO inputs, which detect button presses and trigger pedestrian crossing sequences.

A block diagram of the system is shown in Figure 2.1.

STM32L476 Nucleo-64 The central processing unit that runs FreeRTOS, handling pedestrian and vehicle logic. It receives pedestrian crossing requests via GPIO and sends traffic light control signals over SPI.

Traffic Shield A custom add-on board that mounts on the STM32 Nucleo, containing shift registers, LEDs, and push buttons for pedestrian requests.

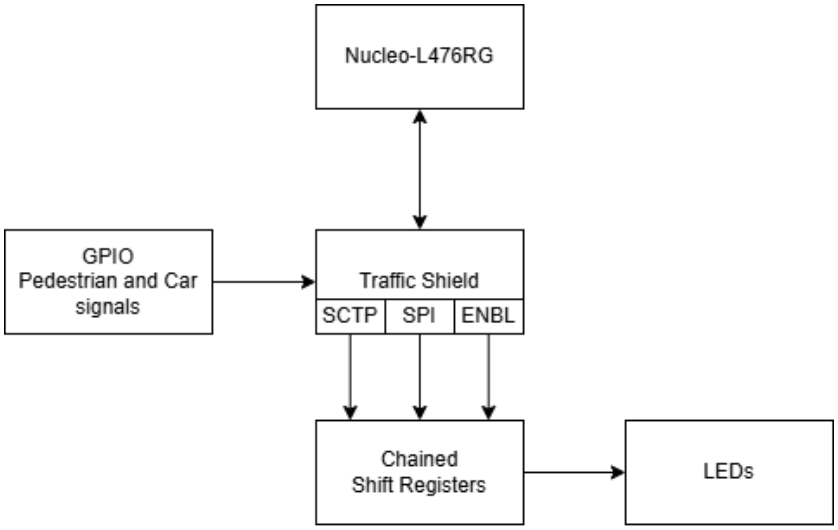

Figure 2.1: Hardware Block Diagram

Shift Registers (74HC595) Used to expand the STM32's output capability. They take serial SPI data from the microcontroller and convert it into parallel outputs to control multiple traffic lights. The SPI communication between the STM32L476 Nucleo-64 microcontroller and the shift registers required careful configuration to ensure reliable data transmission. Initially, a higher baud rate was set for SPI communication, but this led to inconsistent behavior in the output signals of the shift registers. Upon reviewing the datasheet of the 74HC595 shift register, it became evident that the propagation delay and maximum clock frequency are dependent on the supply voltage [3].
Since the microcontroller operates at a supply voltage of 3.3V when powered via USB, the maximum allowable clock frequency for the shift register was found to be lower than expected. To ensure stable operation, the SPI baud rate was reduced to match the shift register's timing constraints, preventing data corruption during transmission [4].

Traffic Lights (LEDs) Indicate traffic states for cars and pedestrians. The lights are connected to the shift registers and updated accordingly.

Pedestrian Buttons (GPIO Inputs) Allow pedestrians to request a crossing. When pressed, the system schedules a signal change at an appropriate time.

2.2.2 Software Architecture

The software architecture of the traffic intersection system is structured to ensure modularity, scalability, and real-time responsiveness. Built around FreeRTOS, it consists of two well-defined tasks that manage different aspects of the system. The core components include the IntersectionController, the pedestrian and car signal logic modules, and the SPI communication module for interacting with the shift registers that control the LEDs.

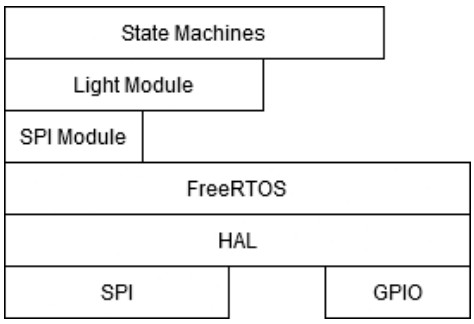

Figure 2.2: Layered View of the Software Architecture

The IntersectionController serves as the central data repository, maintaining the current states of traffic lights, pedestrian signals, and system timing variables. This structure ensures consistency across tasks while allowing efficient state transitions. PedestrianState and CarSignalState manage individual signal logic for pedestrians and vehicles respectively, interacting with the IntersectionController to maintain synchronization.

The system follows a modular design, with each function operating independently. Pedestrian signal logic is responsible for handling button inputs and updating LED states, using bitwise operations. Similarly, car signal logic manages transitions between red, yellow, and green lights, relying on FreeRTOS timing functions to maintain smooth operation. The SPI communication module ensures that the microcontroller efficiently transmits data to the shift registers, minimizing GPIO usage.

Synchronization between tasks is maintained using a mutex, preventing race conditions and ensuring that only one task modifies shared resources at a time.

Bitwise operations play a key role in efficiently managing the traffic light states. Each light state is stored in a compact memory structure, with bitwise manipulation allowing quick updates. This ensures that signal transitions are performed with minimal processing overhead. Each light state enforces exclusivity by using bitwise operations to clear conflicting states before setting

the desired one. When a traffic light transitions to a new state, all other potential states for the same light are explicitly cleared using bitwise AND operations with negated bit masks. The new state is then applied using bitwise OR operations, ensuring that only one state is active at any given time. This approach guarantees that a traffic light cannot simultaneously display multiple signals, such as red and green at the same time.

The behavior of both pedestrian and car traffic signals is governed by state machines, ensuring that transitions follow a structured and predictable pattern. These state machines operate within their respective FreeRTOS tasks and modify system states based on predefined rules.

The pedestrian state machine transitions through the following states:

RED: The pedestrian crossing is blocked, and vehicles are allowed to pass. RED_WITH_INDICATOR: A pedestrian has pressed the button, triggering the waiting indicator light to flash. GREEN: The pedestrian crossing is active, stopping vehicle traffic.

The system evaluates pedestrian button presses and car signal states before determining whether to transition from RED_WITH_INDICATOR to GREEN. After the crossing duration expires, the system returns to RED.

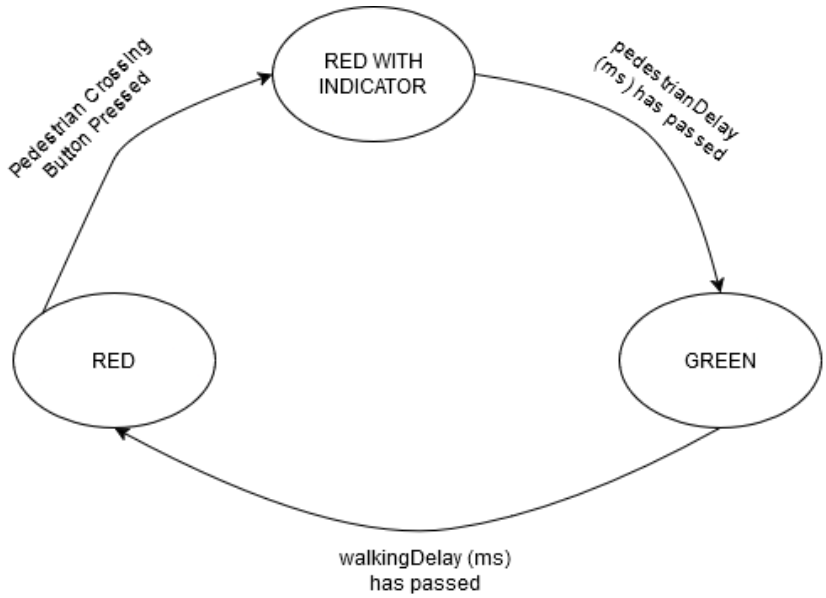

Figure 2.3: Pedestrian Logic State Machine

The car signal state machine follows a strict transition order:

RED: Vehicles are stopped while pedestrians cross. YELLOW: A brief warning before transitioning between RED and GREEN. GREEN: Vehicles are allowed to proceed while pedestrian signals remain red.

Each state is updated based on timing constraints and the presence of pedestrians or vehicles. By structuring the logic this way, the system prevents conflicting signals and ensures safe pedestrian crossings while optimizing vehicle flow.

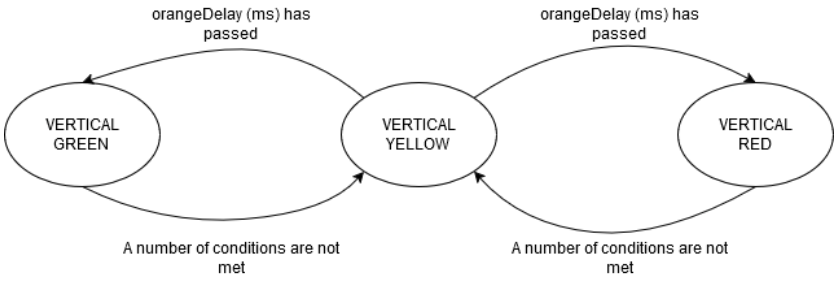

Figure 2.4: Car Logic State Machine

Task execution is managed through FreeRTOS scheduling, ensuring real-time responsiveness. The pedestrian task continuously monitors button presses and updates pedestrian signals, while the car task controls vehicle lights based on predefined timing rules. The SPI task periodically transmits updated LED states to the shift registers, keeping the hardware synchronized with the software state.

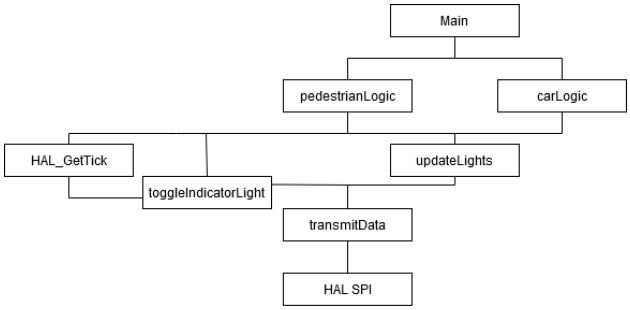

Figure 2.5: Hierarchy of Control in the Software System

2.3 Software Development

The software was developed in a modular fashion to separate pedestrian and vehicle logic into distinct modules. The pedestrian logic handles button press detection and updates pedestrian traffic lights accordingly, while the vehicle logic ensures that cars receive green signals when the intersection is

clear. The two tasks operate independently but communicate by reading each other's state variables.

Concurrency was managed using FreeRTOS, allowing tasks to run concurrently while ensuring safe access to shared resources. A mutex was introduced to prevent data inconsistencies when pedestrian and vehicle tasks modified the same variables.

Timing mechanisms were handled using FreeRTOS functions instead of blocking delays, ensuring that the system remains responsive. The xTaskGetTickCount() function was used to track elapsed time, while osDelay() allowed periodic execution of tasks without affecting overall system performance.

Chapter 3

Results

3.1 Summary of Results

The final implementation successfully managed traffic signals for both pedestrians and vehicles. The system accurately processed pedestrian crossing requests and updated signals accordingly, ensuring that pedestrian and vehicle lights never conflicted. The use of FreeRTOS allowed smooth task execution, with pedestrian and vehicle logic running independently while maintaining synchronization through shared memory.

The system performed as expected, with pedestrian button presses correctly triggering signal changes while ensuring pedestrian safety. The daisy-chained shift register configuration effectively controlled multiple LEDs while minimizing the number of required microcontroller pins. The overall design demonstrated a functional and efficient real-time traffic control system.

3.2 Limitations and potential improvements

Despite its success, the implementation has certain limitations. The current design is tailored for a single intersection and does not scale to multiple intersections or more complex traffic scenarios. Additionally, the system relies on fixed timing values for traffic light transitions, making it unable to adapt dynamically to real-time traffic conditions. Another limitation is the lack of a dedicated fault-handling mechanism, such as watchdog timers, to detect and recover from unexpected failures.

Another limiting factor in the implementation was the reliance on currentTime via xTaskGetTickCount() to track elapsed time. While this approach ensures a straightforward implementation, it does not allow precise event-

driven behavior, as the system relies on periodic polling within tasks rather than responding instantly to changes.

A more efficient approach could involve utilizing interrupts for critical events such as pedestrian button presses and car arrivals. This would allow the system to handle state transitions immediately without waiting for the next task execution cycle. Additionally, using hardware timers could improve the accuracy of delay enforcement.

3.2.1 Automated Testing with Mock Functions

Currently, testing relies on physically observing LED indicators to verify outputs, making debugging inefficient and limiting automation. A major improvement would be to introduce mock functions to replace hardware-dependent functions such as GPIO writes and SPI communication.

By using mock implementations, tests could capture and verify function calls and expected behaviors without requiring physical hardware. For instance:

- Mock GPIO Functions: Instead of setting an LED, the function could store the pin state in a variable that the test framework can inspect.
- Mock SPI Communication: Rather than sending actual SPI signals, the mock function could log transmitted data and compare it against expected values.

This would allow test cases to run in a controlled environment, making them fully automated, repeatable, and easier to debug. It would also facilitate continuous integration testing, where tests can be executed on a development machine without hardware access.

3.3 Conflicting Requirements and Design Trade-offs

During the implementation of the traffic control system, certain requirements were found to be conflicting, requiring modifications or prioritization to ensure a functional system. In cases like this a trade off would have to be made. One instance of this is are requirements R1.4 and R1.5 in combination with R2.7. These requirements create a fundamental contradiction, if R2.7 were to be strictly followed a pedestrian light might turn green, only to immediately lose the right of way due to car traffic arriving at the crossing lane.

Bibliography

[1] "Introduction description of stm32l4/l4+ hal and low-layer drivers um1884 user manual." [Online]. Available: https://www.st.com/resource/en/user$_m anual/dm$00173145 − $description - of - stm32l4l4 - hal - and - lowlayer - drivers - stmicroelectronics.pdf$

[2] "Freertos documentation - freertos™," Freertos.org, 2024. [Online]. Available: https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS$_b ook$

[3] "74hc595; 74hct595 8-bit serial-in, serial or parallel-out shift register with output latches; 3-state," 2017. [Online]. Available: https://assets.nexperia.com/documents/data-sheet/74HC$_H CT$595.*pdf*

[4] "Um1724 user manual," 2020. [Online]. Available: https://www.st.com/resource/en/user$_m anual/um$1724$-stm$32$-nucleo$64$-boards - mb$1136 $- stmicroelectronics.pdf$