IS1300 Project: Traffic shield

Sandberg, Mattias
matsandb@kth.se

December 20, 2023

Contents

1 Summary

This report details the development of a pedestrian crossing simulation on an STM32 Nucleo-L476RG[1] microcontroller using a traffic shield. The project adheres to specific requirements governing traffic light and pedestrian crossing behavior, utilizing hardware components like shift registers[2] and LEDs. The software architecture employs a well-organized state machine, including states such as `START`, `TL_TO_ORANGE`, `TL_TO_RED`, and `WALKING_DELAY`. The system is thoroughly stress-tested to ensure robust performance and prevent glitches during user interactions. The iterative development process follows Test-Driven Development[3] principles, contributing to high code quality. The report outlines successful simulation results, discusses limitations, and suggests potential improvements for future work, such as multi-intersection support (TASK2 and TASK3) and real-time operating system (RTOS) integration.

2 Introduction

The Projects objective task was to simulate a pedestrian crossing on a traffic shield mounted on a STM32 Nucleo-L476RG [1]. The project entails the implementation of specific requirements governing the behavior of the traffic light and pedestrian crossing system. During initialization, the pedestrian crossing displays a red signal, while the car signal exhibits a green indication. Upon activation of the pedestrian button, the indicator light toggles at a defined frequency until the pedestrian crossing signal turns green.

Subsequently, all car signals in each active lane crossing the crosswalk transition to red after a specified duration following a pedestrian button press. The pedestrian signal remains green for a designated period, and its state interacts with the car signal turning red.

These requirements collectively define the operational logic and timing characteristics of the traffic light and pedestrian crossing system, ensuring safe and efficient traffic flow in a controlled environment. The report will cover the method, result and discussion.

3 Method

3.1 Planning, Architecture and Structure Hardware

In the beginning of the course, there were two mandatory seminars that covered the hardware and software architecture, making it easier to grasp the concepts and interface between the hardware and software. The hardware setup included an STM32 Nucleo-L476RG [1]. and a traffic shield designed by Matthias Becker. The traffic shield featured 8-bit serial-in, serial or parallel-out shift registers with output latches (74HC595D [2]) to control components such as the LEDs for the traffic light and pedestrian lights.

The LEDs connected to the shift registers were controlled using SPI (Serial Peripheral Interface), which is a widely adopted synchronous serial communication standard. SPI operates in a master/slave architecture, where one main device (master) coordinates communication with multiple peripheral devices (slaves). This full-duplex communication system involves the main device transmitting data on the MOSI (Main Out Sub In) line while receiving data on the MISO (Main In Sub Out) line from the subservient peripherals. SPI is commonly employed in embedded systems for short-distance communication between integrated circuits, facilitating seamless connectivity between microcontrollers and various peripheral chips.

The buttons was essential for the task since it triggered the pedestrian lights. The buttons where implemented using GPIO (General-purpose input/output) which is a versatile interface that allows for programmable input and output operations. GPIO provides a mechanism for the microcontroller to interact with external devices, in this case, the buttons.

In the context of the system, GPIO was employed to configure specific pins on the microcontroller for input, allowing it to sense the state of the buttons. This interaction is fundamental for responding to user input, such as pressing the buttons to initiate actions like activating the pedestrian lights.

In the STM32Cube framework, the Hardware Abstraction Layer (HAL) facilitates communication with external peripherals like shift registers and buttons through well-defined APIs. For shift registers using SPI, the HAL provides ready-to-use APIs for initialization, configuration, and management of data transfers. These high-level, feature-oriented SPI APIs abstract the hardware complexities, easing integration with shift registers. For buttons utilizing GPIO, the HAL offers GPIO-related APIs to configure pins for button input, simplifying tasks like initialization and event handling. Figure 1 is a simple block diagram of the hardware architecture:

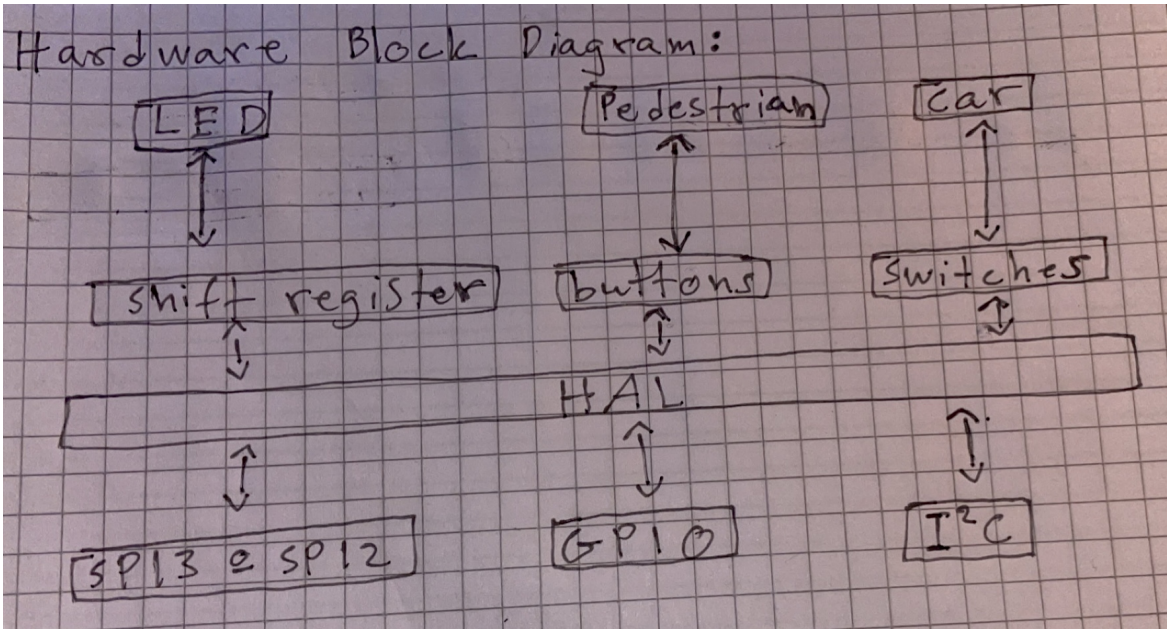

Figure 1: Hardware Block Diagram

The practical execution of the hardware aspect of the project to be able to complete TASK1 was first to start a new STM32CubeIDE [4] project and select the STM32 Nucleo-L476RG[1] board and in the code generation section select "Generate peripheral initialization as a pair of .c/.h files per peripheral". Then the port configuration was done by looking at the `IS1300_TrafficLight_Schematics.pdf` [5] were the ports on the STM32 Nucleo-L476RG[1] could be mapped to the traffic shield. Since Task1 was the sole focus, the consideration of cars was initially included due to uncertainty regarding project time availability. In figure 2 the pinout view can be seen for an easy understanding of the pin configuration:

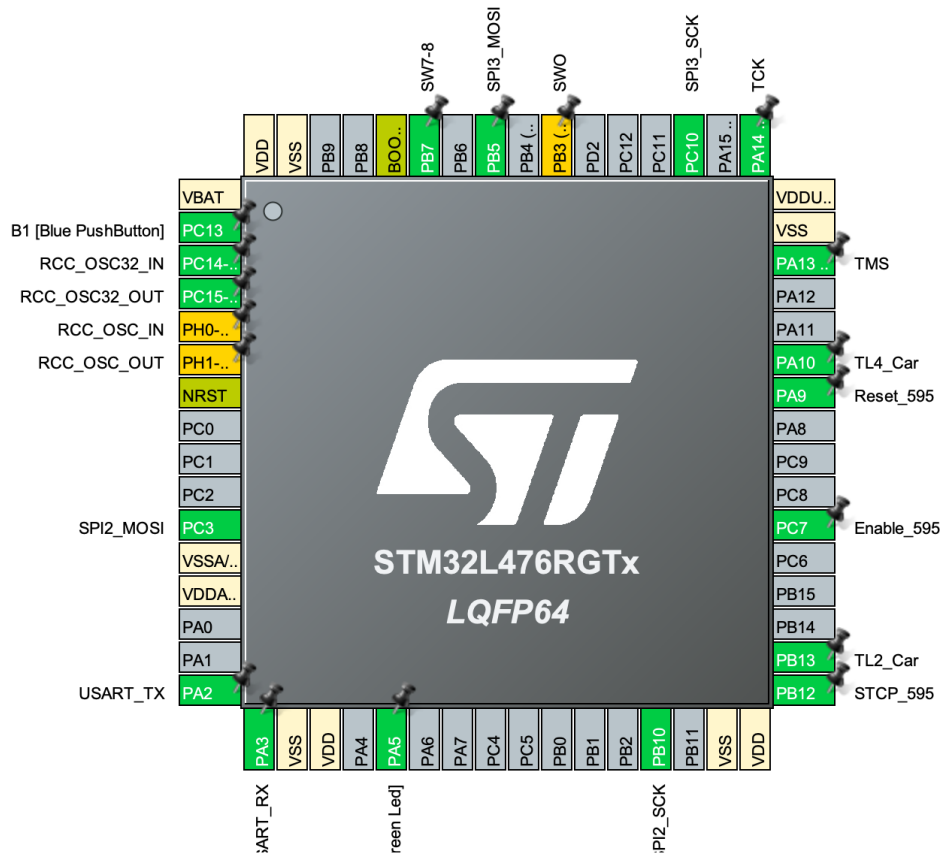

Figure 2: Pinout View

The next step was to initiate SPI2 and SPI3 and set them to Half-Duplex Master on the designated pins. The data size had to be increased to 8 bit to support the LEDs and shift registers[2]. There was also a vital setting that had to be applied which was that the baud rate had to be dropped with a prescaler of 4 to reduce the baud rate to 20 Mbits/sec to eliminate glitches due to a excessive data rate.

RTOS (real-time operating system) which is designed for applications with critical time constraints, ensuring precise processing of data and events was not used in the solution of TASK1. Unlike time-sharing operating systems, RTOS prioritizes tasks based on real-time requirements, aiming for minimal interrupt and thread switching latency. After checking the `IS1300_TrafficLight_Schematics.pdf`[5] the GPIO ports could be specified and customized such as in figure 3 which can be seen below:

Pin Name	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pull...	Maximum output ...	Fast Mode	User Label	Modified
PA5	n/a	Low	Output Push Pull	Pull-up	Low	n/a	LD2 [green Led]	☑
PA9	n/a	High	Output Push Pull	Pull-up	Low	n/a	Reset_595	☑
PA10	n/a	n/a	Input mode	Pull-up	n/a	n/a	TL4_Car	☑
PB7	n/a	n/a	Input mode	Pull-up	n/a	n/a	SW7-8	☑
PB12	n/a	Low	Output Push Pull	Pull-up	Low	n/a	STCP_595	☑
PB13	n/a	n/a	Input mode	Pull-up	n/a	n/a	TL2_Car	☑
PC7	n/a	Low	Output Push Pull	Pull-down	Low	n/a	Enable_595	☑
PC13	n/a	n/a	External Interrupt ...	Pull-up	n/a	n/a	B1 [Blue PushButt...	☑

Figure 3: STMCubeIDE GPIO Settings

3.2 Planning, Architecture and Structure Software

The software architecture for the developed system is structured around a state machine implemented in the `Task1.c` file. This state machine controls the behavior of the traffic light system, incorporating various states and transitions. The primary states include `START`, `TL_TO_ORANGE`, `TL_TO_RED`, and `WALKING_DELAY`. The state machine is responsible for managing the sequence of traffic light states and pedestrian light toggling.

The `Task1.c` file utilizes several functions from `Task1_functions.c` to implement the behavior associated with each state. These functions include `toggle_PLblue_TLgreen`, `toggle_PLblue_TLorange`, `toggle_PLblue_TLred`, and `walkingDelay`. Each function is designed to handle specific visual changes in the traffic and pedestrian lights, and they are tightly integrated with the state transitions.

The `Toggle_PLblue_TLgreen` function, for example, toggles the blue pedestrian light while keeping the green traffic lights static. Similarly, `Toggle_PLblue_TLorange` and `Toggle_PLblue_TLred` handle the toggling of the blue pedestrian lights in conjunction with static orange and red traffic lights, respectively. These functions incorporate a time-based animation strategy `(HAL_GetTick)`, controlling the duration and frequency of LED shifts to achieve the desired visual effects.

Additionally, the walkingDelay function simulates a walking delay animation for the traffic lights, displaying a sequence of traffic and pedestrian light configurations in a timed manner. This function contributes to the overall dynamic behavior of the traffic light system and is responsible for turning the pedestrian light green and then transition back to the start state.

The system's modular structure enhances maintainability and readability, allowing for easy modification or extension of specific functionalities. The only state that is triggered by external events is the START state which gets triggered by the pedestrian button. This means that the program cannot be corrupted by spamming the buttons while the state machine is in another state which makes this solution stress test approved. The use of modular functions enables clear separation of concerns and facilitates code reuse, making the software architecture both scalable and comprehensible. Figure 4 shows the the basics of the state machine implementation:

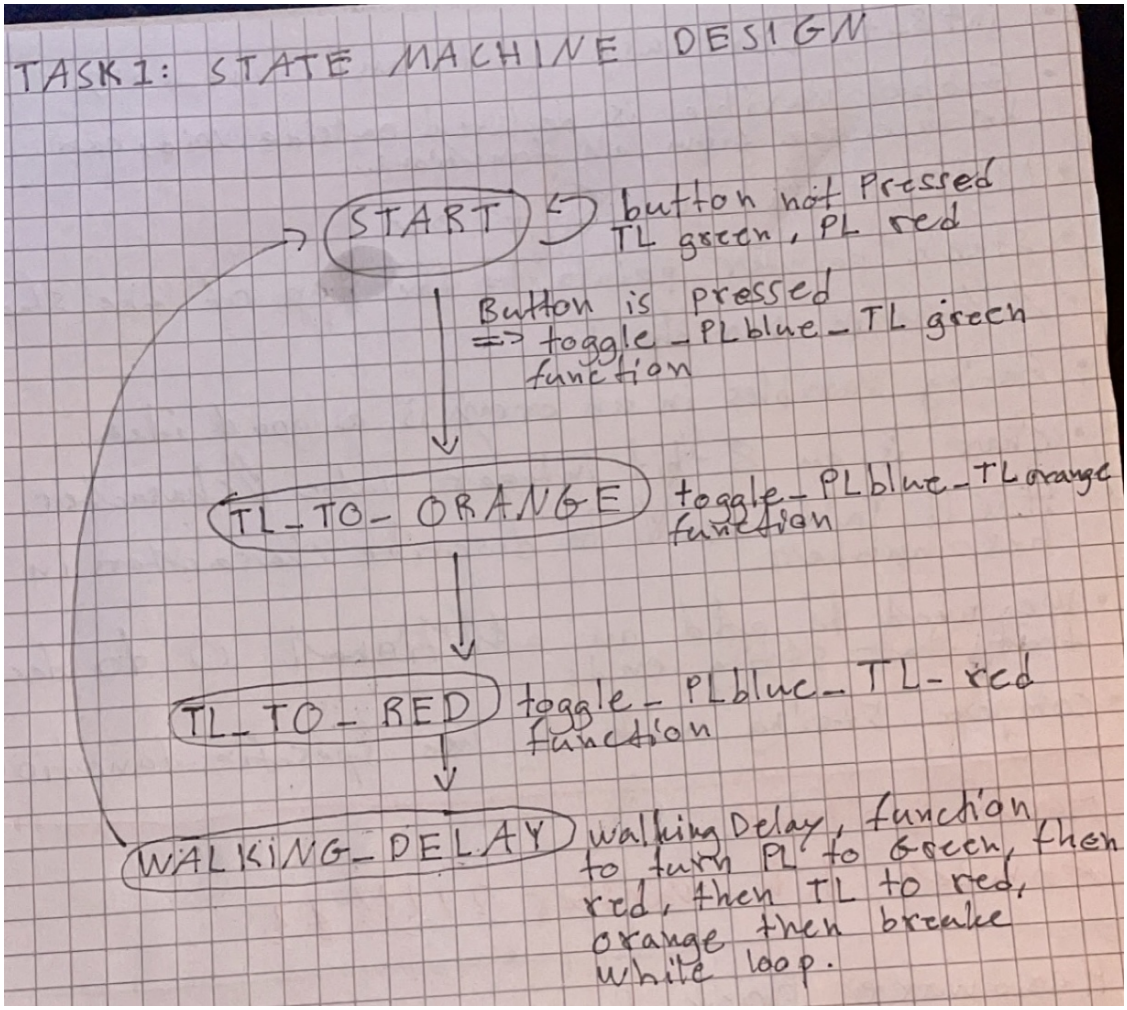

Figure 4: Software State Machine

A tricky part at the beginning of the software initiation was to handle ShiftLED function. The ShiftLED function employs SPI communication to transmit LED data stored in the buffer array to a shift register[2]. After the data transmission, a brief delay of 1 millisecond allows for processing, and then the STCP pin is set to latch the data into the storage register. Subsequently, the STCP pin is reset, completing the process and preparing for the next set of LED data. When the logic of ShiftLED was established, the ability to light up individual LEDs where straight forward by looking at the `IS1300_TrafficLight_Schematics.pdf`[5] and which pin of the shift register 74HC595D[2] correspond to which LED. To do this the reference sheet for 74HC595D[2] was used where it said that 8 of the pins (Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7) where parallel data output. Therefore 8*3 (three shift registers) = 24 possible output pins and an example of lighing up all the LEDs looks like this: `uint32_t LEDsAll[] = {0b00111111100111111100111111};`. It would of course be okey to write the bits in hexadecimal or similarly but the simplicity and intuity made binary a better choice. In figure 5 the function of the described ShiftLED is shown:

```c
/**
@brief      Shifts LED Data to a Shift Register
@details    Transmits data to a shift register using SPI communication
@param      buffer: An array containing the LED data
@param      regs: Number of registers to shift
*/

void ShiftLED(uint32_t buffer[], uint8_t regs)
{
    HAL_SPI_Transmit(&hspi3, (uint8_t*)&buffer[0], regs, 100);

    HAL_Delay(1);

    HAL_GPIO_WritePin(STCP_595_GPIO_Port, STCP_595_Pin, GPIO_PIN_SET);

    HAL_GPIO_WritePin(STCP_595_GPIO_Port, STCP_595_Pin, GPIO_PIN_RESET);
}
```

Figure 5: ShiftLED Function

One of TASK1 most advanced problems was to be able to toggle the blue pedestrian lights when one of the buttons was pushed. The C code to solve this utilizes a time-based animation strategy to control the sequence and duration of LED shifts, ensuring a visually appealing display. The logic involves defining the toggle frequency (toggleFreq) and the total time (totalTime) for the animation. The function employs a state variable (status) to alternate between two LED configurations: one with only green traffic lights and red pedestrian lights (ledsTLGPLR) and another with the added blue pedestrian lights (ledsTLGPLRB). This implementation, driven by the timing constraints and status toggling, achieves the desired visual effect of blue LEDs toggling. In figure 6 a Screenshot of the implementation is shown:

```c
void toggle_PLblue_TLgreen(void)
{
    int toggleFreq = 500; // 500 milliseconds
    int totalTime = 5000; // 5000 milliseconds

    uint8_t bytes = 3;
    uint32_t ledsTLGPLR[] = {0b000000000000110000100000};       // TL Green, PL Red
    uint32_t ledsTLGPLRB[] = {0b000000000010110000100000};      // TL Green, PL Red & Blue
    uint8_t status = 0;
    uint32_t startTime = HAL_GetTick();     // Get the current time in milliseconds

    while ((HAL_GetTick() - startTime) < totalTime)     // Check elapsed time
    {
        status = !status;
        if (status)
        {
            ShiftLED(ledsTLGPLR, bytes);
        } else
        {
            ShiftLED(ledsTLGPLRB, bytes);
        }
        HAL_Delay(toggleFreq);
    }
}
```

Figure 6: Blue Pedestrian Lights Toggle Function

3.3 Testing

In planning the development of the Traffic Shield programming, a methodology inspired by Test-Driven Development (TDD)[3] principles was employed. Similar to TDD[3], the process began by outlining the specific requirements of the implementation. This involved envisioning the initial state where the pedestrian crossing is red and the car signal is green. Iteratively adding tests to simulate pedestrian and trafficlight interactions was a key aspect of this approach.

Following the TDD[3] approach, the iterative cycle involved consistently running tests to ensure that the new features were introduced for expected reasons and validating that the test was functioning correctly. This practice prevented the inclusion of invalid tests and maintained a strong focus on requirements before the actual code was written.

The subsequent steps involved writing the simplest code necessary to pass the newly added tests, with an emphasis on avoiding unnecessary complexity. Continuous testing played a crucial role, ensuring that the new code met the specified requirements without breaking existing features. Additionally, a refactoring phase was implemented, enhancing the code for readability and maintainability while ensuring that the functionality remained intact. The focus was to comment the overall functionality of each folder and functions to easily be able to present the code. Although there would have been easier to implement more states to the state machine, the solution of making walkingDelay a more complex function with three different time interval was an achievement and made the code more easily readable and compact.

Throughout the development process, this iterative cycle of testing, coding, and refactoring was repeated for each new piece of functionality. This iterative and incremental approach allowed for a continual focus on software quality, addressing potential issues early in the development cycle. Ultimately, the application of these principles fostered a clearer and cleaner design, enhancing the overall success of the Traffic Shield programming project. In the `Test.c` file there is a `Test_program` function were the programmer easily can choose which functionality to test. In the folder `Test.c` there is both some basic test of the STM32 Nucleo-L476RG [1] to traffic shield connection such as `BUTTONtest();` and `LEDtest();` but also tests of each of the functions utilized in the final state machine responsible for solving TASK1. In the `main.c` file the user can easily change between the test program and the main TASK1. In figure 7 a screenshot of the `Test_program` can be seen to get an understanding of the testing approach.

```c
/**
@brief      Main Test Program Function
@details    Calls functions to test button and LED functionality
*/

void Test_program(void)
{
    BUTTONtest();                           // Test the Pedestrian buttons
    // LEDtest();                           // Uncomment to test LED functionality
    // TEST_toggle_PLblue_TLgreen();        // Continuous testing for PL Blue toggle and TL Green static.
    // TEST_toggle_PLblue_TLorange();       // Continuous testing for PL Blue toggle and TL Orange static.
    // TEST_toggle_PLblue_TLred();          // Continuous testing for PL Blue toggle and TL Red static.
    // TEST_walkingDelay();                 // Continuous testing for walking delay animation.

}
```

Figure 7: `Test_program`

4 Results

The program for TASK1 considers only the street direction (2 and 4) and can do the following:

- Start state turns on green traffic lights and red pedestrian light. If SW7 or SW8 is pushed the blue lights starts to toggle with a frequency of toggleFreq which is set to 500ms. This occurs for 5s before the state machine jumps to the next state to make the cars be able to slow down like in a real life situation.
- In the next state the blue pedestrian light continues to toggle with toggleFreq but the orange traffic light and red pedestrian light is static. After 3000ms the state machine jumps to the next state.
- In the next state the blue pedestrian light continues to toggle with toggleFreq but the red traffic light and red pedestrian light is static. After 3000ms the state machine jumps to the next state.
- In the next state the traffic lights are red and the pedestrian lights green enabling crossing for 3000ms. Then the pedestrian lights shifts back to red for 2000ms enabling the traffic lights to turn orange for 3000ms before going back to the Start state.

5 Discussion

While going trough all the different states no user interaction can make the program glitch since the functions and code are stress tested. An example of this is that the button can be spammed but it will only effect the traffic system when the state is in Start state as intended. This is possible due to the implementation of the `HAL_GPIO_ReadPin(GPIOB, SW7_8_Pin) == GPIO_PIN_RESET` and the position of the next state and break logic in the state machine.

An obvious restriction for the program is that it only considers street direction 2 and 4 as specified in TASK1. All the LEDs and some of the cars were implemented in the software and hardware at first since there could have been more allocated time

for the project TASK2 and TASK3. A further implementation of TASK2 and TASK3 would have provide a more realistic simulation of urban traffic scenarios, incorporating interactions between different crossings. Although an Implement of dynamic traffic patterns, considering varying traffic densities, different signal timings, and adaptive responses to real-time changes in pedestrian and vehicle flow was not required in TASK1. It would have added complexity and realism to the traffic simulation.

An improvement could have been to explore the implementation of a real-time operating system (RTOS) to address critical time constraints more efficiently. An RTOS could have optimized tasks prioritization and therefore minimize latency, leading to a improved system responsiveness.

References

[1] Stm32 nucleo-l476rg. https://www.st.com/en/evaluation-tools/nucleo-l476rg.html. accessed: 2023-12-03.

[2] Nexperia. 74hc595d. https://assets.nexperia.com/documents/data-sheet/74HC$_H CT595.pdf. Published : 2021 - 09 - 11.$

[3] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and software Technology*, 46(5):337–342, 2004.

[4] ST. Stmcubeide. https://www.st.com/en/development-tools/stm32cubeide.html. accessed: 2023-12-02.

[5] Matthias Becker. Is1300$_t raf ficlight_s chematics.pdf. accessed : 2023 - 12 - 04.$