Pedestrian Crossing

William Sjöberg

December 2024

Summary

The problem is addressed and specific background details are given to build an understanding of what can be done. The method describes the hardware coupled with the software and how these two work together to simulate a pedestrian crossing. The software is always tested during the development phase and assures the different methods that the correct functionality is achieved. The results are then shown which describes what the program can do, what restrictions the pedestrian crossing has and what can be done to further improve the work of the pedestrian crossing.

Contents

1 Introduction

In the beginning, the report describes the pedestrian crossing. It provides a background to why traffic control is important, and what technology is used to simulate a pedestrian crossing using an embedded system.

1.1 Problem Description

The problem that needs to be solved is the simulation of a traffic crossing on a NUCLEO-L476RG microcontroller that involves both cars and pedestrians. This includes the management of car and pedestrian traffic lights, where cars can have red, orange, or green, and pedestrians can only have red or green. Pedestrians can also choose to cross the pedestrian crossing by pushing a button, which triggers a sequence of events, making it possible for them to cross. This project aims to replicate a real-life scenario of a single pedestrian crossing.

1.2 Background

The control of traffic lights is crucial to maintaining a safe flow of traffic while also keeping pedestrians and vehicles safe. People are not faultless, thus designing different signals to slow or stop traffic can reduce the severity of different accidents that occur[1]. To be able to control the different signals and traffic flow, traffic light controllers are used together with embedded systems to manage different states and conditions[2].

An embedded system combines hardware and software, with the possibility of additional parts, working together to perform a dedicated function. Embedded systems are often regarded as components in a larger system. For example, one embedded system monitors the vehicle's emissions while another displays the information on a dashboard[3]. The STM32 Nucleo-64 board, which is a microcontroller, provides flexibility when building prototypes by offering different features. Arduino Uno V3 connectivity support allows for adding different types of shields and connecting them to the board[4].

The shift register that is connected to the traffic light shield is an 8-bit serial-in/serial or parallel-out shift, which has a storage register and 3-state outputs. A serial input (DS) cascades data into the shift register bit by bit while the serial output (Q7S) can cascade to another shift register if it is connected to the serial input of that shift register. There exists a master reset (MR) and when it is LOW, the shift register resets. Data gets shifted in the shift register when the shift clock pulse (SHCP) goes from LOW to HIGH and data is transferred to the storage register when the storage clock pulse (STCP) does the same transition. To be able to see the output, the output enable input (OE) needs to be LOW, while a HIGH puts the outputs

into a high impedance state[5].

By incorporating a microcontroller with a dedicated traffic light shield, we can simulate a pedestrian crossing with LEDs and buttons that will trigger different events and signals for our simulation. These events and signals can happen simultaneously and handling these problems provides valuable experience when designing and operating embedded systems.

2 Method

The hardware consists of the NUCLEO-L476RG microcontroller and the traffic light shield that connects to the microcontroller. On the traffic light shield, we have LEDs and switches that we will interact with and simulate a traffic crossing with. The equipment that was used to create this pedestrian crossing simulation was a computer and a USB cable to be able to upload code to the hardware. The software that was used was the STM32CubeIDE, which enabled the generation and creation of code/modules for our board, and pin-out configurations for the hardware components. Testing was made in parallel with the software to test the modules that were needed by the pedestrian crossing to function properly and to go back if something changed anything in these modules.

2.1 Hardware

The hardware can be described using a block diagram where all the hardware parts that are used in the project are shown. The hardware that is on the traffic light shield is the only thing we are concerned about and not the traffic light shield itself, because it only serves as a connection between different hardware components and the nucleo board.

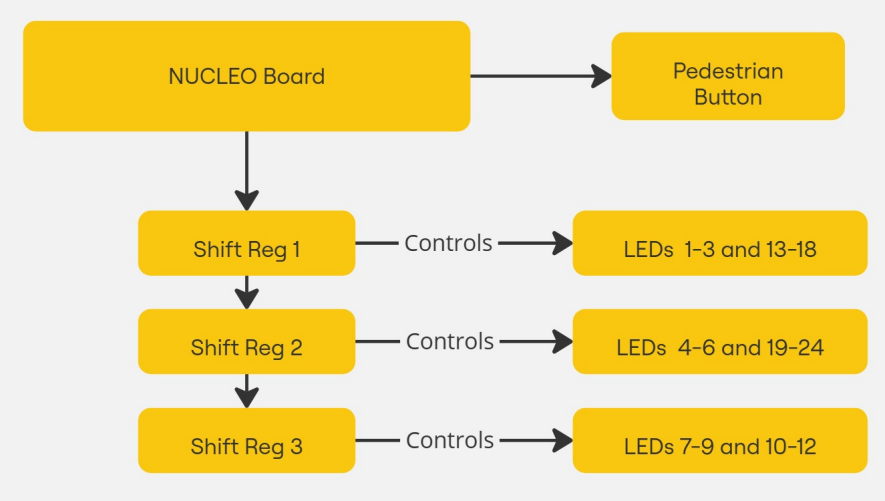

Figure 1: Block diagram of the hardware components interacting with the board.

Both of the components, pedestrian buttons and shift registers, use general-purpose input/output (GPIO) to be able to communicate and control these components. Two pedestrian buttons are linked to the same GPIO pin. The shift register is a little bit different because it needs more than one GPIO pin to function properly and is also connected in series, making it a 24-bit shift register. The shift registers can be controlled with either SPI or GPIO, but I went with GPIO because it felt more comfortable for the project. LEDs that are relevant to the pedestrian crossing are LEDs 4-6, 10-12 and 19-24, which means that shift register 1 is not needed in this case. To turn on the LEDs in shift registers 2 and 3, cascading is used where the serial output in shift register 1 is connected to the data pin of shift register 2 and the serial output of shift register 2 is connected to the data pin of shift register 3. This way we can move data from one shift register to another in an efficient way.

Hardware and software relate to each other, meaning that the hardware gives the physical interface while the software controls the hardware. The software adapts to what the hardware has to offer, for example, if every LED needed a GPIO pin, it would run out of pins pretty quickly but since there are shift registers makes it so the software can control many more LEDs. These GPIO pins are configured either as inputs or outputs depending on the desired functionality.

2.2 Software

Software is developed and configured in the STM32CubeIDE application and a clear software architecture was developed with the project in mind. An

application with high-level application code was developed which involved logic for different states, followed by middle-ware level functions that the application can use which abstracts the hardware and complex parts.

The software architecture begins with a block diagram of where the hardware view acts like the base and boxes are added to represent software functionality. The reasoning behind this diagram is to make it clear what things are critical for it to be able to work as intended.

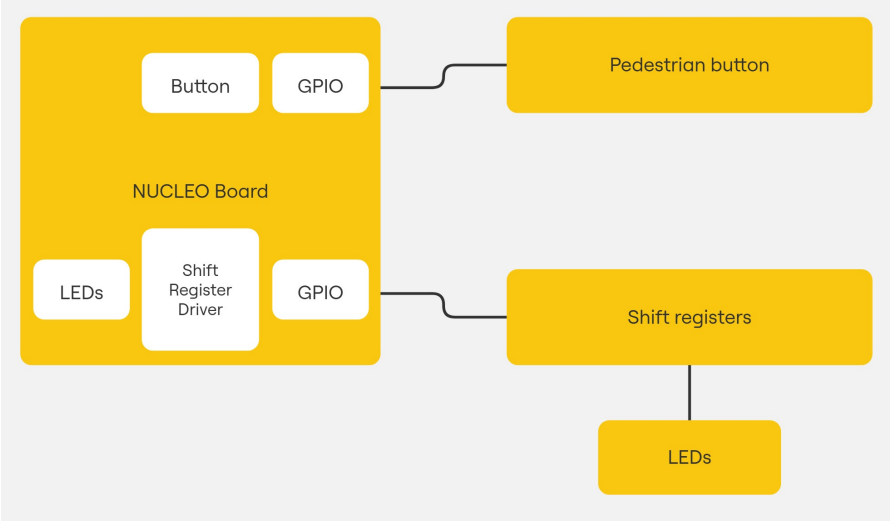

Figure 2: Block diagram of the software using the hardware as the base.

As for the hardware, the NUCLEO board can control the pedestrian button and shift registers, but for it to work it needs software functionality. Both the pedestrian button and shift registers make use of configured GPIO pins to control the different components. Since the pedestrian button makes use of GPIO, simple logic will be needed to control it, that is also the case for the shift registers. The shift registers need a dedicated shift register driver so that the functionality becomes easier to understand. To be able to control the LEDs, specific program logic to control the different traffic light states has to be implemented that makes use of the shift register driver.

A hierarchy of control is made to show an overview of what components call what in a system. This diagram will easily show the higher levels and lower layers of the project.

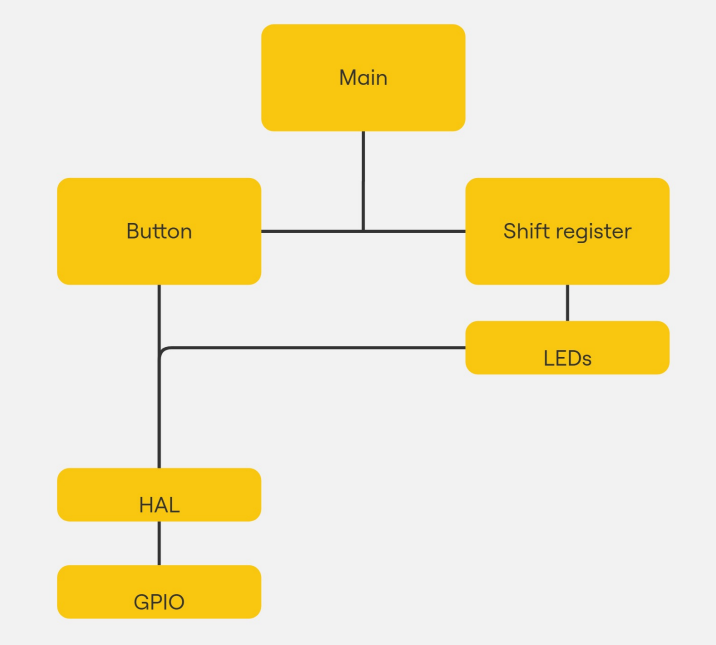

Figure 3: Diagram of the hierarchy of control in the software.

The hierarchy of control shows that a main function can call both buttons and shift registers to get things done in the software. For the button to work, it calls the hardware abstraction layer which then calls the GPIO to control the state of the button. The same can done for the shift register, that is when "main" calls the shift register controls the LEDs which are then controlled by HAL and GPIO that are connected to the shift register.

The code that is created in the project is divided into different files which are in different levels of the software architecture. There exists a high level where application code is used that the user easily understands. Code is also created in the lower levels where different functions are defined and then used in the higher level. These different levels can be shown in a block diagram.

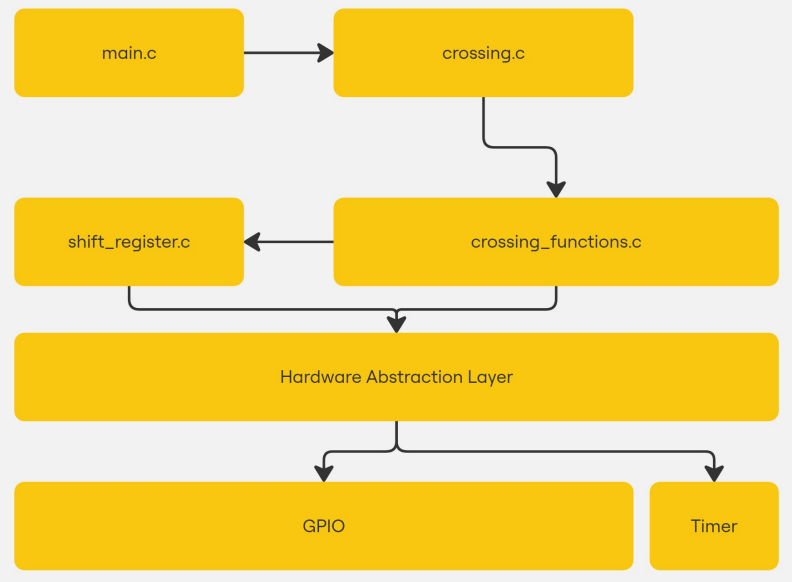

Figure 4: Diagram of the software in different levels.

This block diagram describes which files interact with each other and describes it in levels where the high level is at the top and the lower level is at the bottom. The program starts in "main.c" where it decides to run the function in "crossing.c" which has the application code for the pedestrian crossing. The application code can then call functions in "crossing_functions.c" which works like abstractions for the application code to use. At the same level, there is also "shift_register.c" which works like a driver for the shift register which is used to control the different bits in the shift register. Both files on this level make use of HAL which in turn uses GPIO to control the different parts. The timer is used by "crossing_functions.c" to be able to retrieve the current time on the timer for the application code to use.

GPIO pins were configured according to their desired functionality. The pedestrian button was configured as an input with a pull-up mode, meaning that the button is HIGH when not pressed and LOW when pressed. The shift register had multiple GPIO pins in use and all of them were configured as output push-pull with no pull-up or pull-down.

The different code modules that were implemented for the pedestrian crossing were a function for turning on the LEDs and one for turning them off. Then a function for the button was implemented that checks if the button was pressed or not. Lastly, two functions were made and one returns the time of the timer on the microcontroller and the other one simulates a delay for a pedestrian wait time. The shift register has 3 code modules, one is for sending a set of data into the shift register, another module resets the entire shift register and the last one sends the data to the storage so the output can be seen by the user.

Modules were created in the "crossing_functions.c" file and they fill a purpose for the application. "Led_on" was created and it has one parameter which is a numbered LED on the traffic light shield. This led number corresponds to a bit pattern that masks the current state of the shift register that then needs to cascade into the shift register. It checks if the LED is already on and if is not on, then masks the bits and sends them into the shift register. The "Led_off" module does the same thing but performs the masking if it is already on. The masking procedure is made by doing a bit-wise XOR which removes the bits if they are the same and adds them if they are not the same. A module for the button was made called "PL2_hit" which returns true if the button is pushed down and false if it is not. Another module "Get_timer_time" returns the current time from the timer and a "Walk_across_time" simulates a delay which is determined by a parameter "walkingDelay".

Driver modules for the shift register in "shift_register.c" were also created such as "Reset_shift_data" which turns the master reset GPIO pin to LOW and then HIGH to reset the shift register connected in series. another module called "Set_state" has a variable "data" as a parameter which is the bit-pattern that is going to be shifted in the shift register bit by bit. The LSB corresponds to the first bit that is cascaded in the shift register and is going to be the bit that corresponds to the last bit used by the LEDs in the third shift register. First, The data GPIO pin is set to 1 or 0 depending on the bit pattern from the data variable, to shift the bit into the shift register, the shift clock pulse is first set to HIGH to shift the bit and then then LOW to be able to make the transition for upcoming bits. Lastly, a module called "Latch_shift_data" sets the storage clock pulse to HIGH to send the bits to the storage and then LOW, this makes it so the corresponding LEDs light up on the traffic light shield.

When all the functionalities of the different components are working as intended, the application controlling the traffic lights can be created in "crossing.c". It functions like a state machine where it has three different states the pedestrian crossing can be at, "CarsPassing", "PedestrianWait" or "PedestrianPassing".

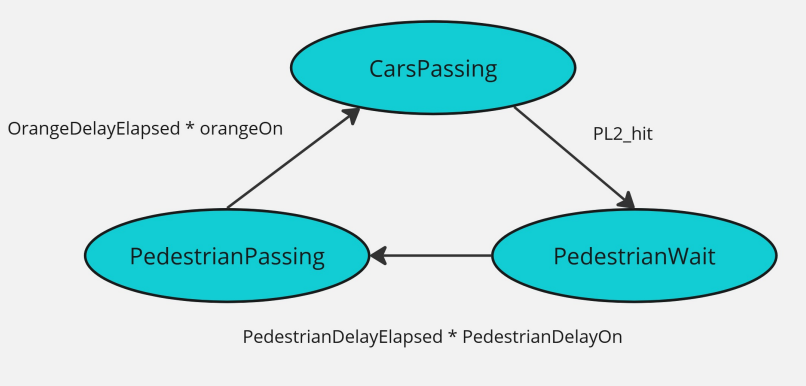

Figure 5: State diagram of the application that controls the pedestrian crossing.

The program starts at the ”CarsPassing” state where the car traffic lights are green and pedestrian lights are red. A pedestrian can push the pedestrian button and make it switch state to the ”PedestrianWait” state where the traffic lights will undergo a transition from the initial state to a state where it is ready to let out the pedestrians on the crosswalk. When the program has come to the point where it is ready to let out the pedestrians, it switches state to ”PedestrianPassing” which gives the pedestrian light green and traffic light red. It allocates some time for the pedestrians to walk across and when that time is up, the pedestrian light goes from green to red and the traffic light transitions from red to orange to green, which then switches the state to ”CarsPassing”.

Different variables are used to tune how the traffic light control works and all are given in milliseconds. The variables are ”toggleFreq” which determines the toggle frequency of the indicator light, ”pedestrianDelay” which determines how long the traffic light shall stay red before giving the pedestrian light green, ”walkingDelay” tells how long the pedestrian light stays green and lastly, ”orangeDelay” determines how long the traffic light stays orange.

2.3 Testing

The testing was made from a dedicated testing file where all the functionalities that were going to be implemented were tested before being added to the main application handling the traffic light controls. A Test Driven Development cycle was used to test each implemented function. First, a test is written for the desired functionality and if it fails we need to refactor the code and try again to achieve the desired functionality. The main strength of TDD is that if something does not work, you can always go back to the test program to see how it is supposed to work.

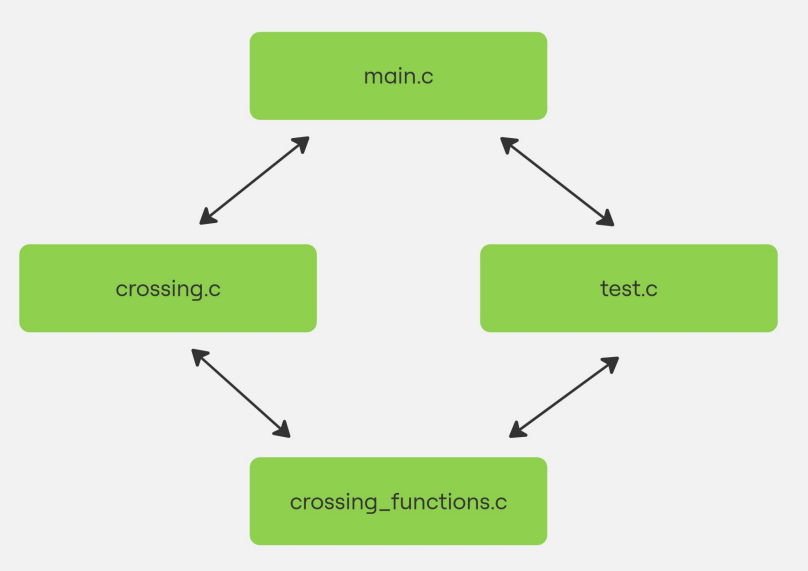

Figure 6: Diagram of how the testing was made during Test Driven Development.

This diagram describes how the testing works, it gets called by main instead of the main program to use then "crossing_functions.c" to test different functions/modules to see if they work. "Led_on" was tested by lighting up every LED that is used in the project and not the other ones that exist on the traffic light shield. "Led_off" is tested in the same way but it calls the test where it lights up every LED that is relevant to the project and then turns them off one by one. A button test was also made for "PL2_hit" where if the button is pressed, it starts toggling the blue pedestrian light. Testing the timer was done similarly to the button test where we toggle the blue LED depending on a certain frequency using the timer to receive the current time and waiting till it has exceeded a specified amount of seconds.

3 Result

The hardware uses LEDs that control the traffic lights and pedestrian lights which are controlled by shift registers that are in series. A button for the indicator light for pedestrians is also used in the hardware. The software has a software architecture that abstracts the complex parts and uses high-level logic to control the pedestrian crossing. The software was tested using the Test Driven Development cycle that ensures that the functionality is correct but also if something breaks, one can go back to the test and be assured that it works.

The fully developed program together with hardware can simulate one pedestrian crossing where cars can pass from both sides. At Initialization,

the pedestrian crossing is red and traffic lights are green, to start the transition of cars passing to pedestrians passing, the pedestrian has to click the button that toggles the indicator light and the whole procedure of walking across will undergo. When the procedure is done, the traffic state goes back to what it was at Initialization, being the cars having green and pedestrians having red.

3.1 Restrictions

The restrictions that are prevalent in the implementation is that if you want to turn on the LEDs, you have to do it one by one and this also applies to the disabling of LEDs. This leads to multiple "Led_on" and "Led_off" calls, which can be problematic if many LEDs need controlling. Another restriction is that this program does not consider if many cars are approaching the pedestrian crossing, meaning that there can be many interruptions for the traffic state if the pedestrian indicator button is pressed immediately as a car has gotten a green light. Another restriction is that if a pedestrian enters the pedestrian crossing at the last possible time, it can become dangerous because the pedestrian light goes red right when the car traffic light goes from red to orange.

3.2 Discussion

The hardware and software that are used in the project are easy to use and made the implementation of a pedestrian crossing easier than I thought it would. The hardware was easy to set up and the traffic light shield gave a clear understanding of what was needed to be able to simulate the pedestrian crossing. The software configuration of pins where also easy to set up, but the hardest thing about the project was the project code and how to structure a software architecture smartly and correctly. It was hard to know what code was supposed to go where and if it violated any rules of high-level vs low-level.

Things that could be improved upon in further work with this project are that one can create a method of controlling multiple LEDs with one function call instead of multiple ones. This can be solved using SPI or it can be solved by implementing a method of enabling LEDs where the argument is for example "green" and both of the traffic lights for the cars are turned green. Another thing that can be improved upon is the handling of setting a delay for a traffic light, this delay can be implemented more easily, using for example a Real-Time Operating System that has different tasks that execute. By doing this, the implementation can also be easier to understand when the project is divided into specific tasks for different functionalities in a traffic crossing. One last thing that could be improved upon is the logic that is used in the application code. In the current implementation, it uses

a lot of variables that dictate which code snippets are supposed to run, this can be made a lot simpler and more understandable with the usage of an RTOS. This can then be made with tasks that have different priorities and the usage of mutexes that can lock several of the shared resources of the pedestrian crossing.

In conclusion, the project works as expected and the hardware and software made the implementation easier than I thought it was going to be. There are some restrictions in the pedestrian crossing, but there exists room for improvement that can ensure a better-functioning pedestrian crossing which can then scale into a bigger traffic crossing with multiple lanes and pedestrian crossings.

4 Bibliography

References

[1] A. Spears, "The Purpose of Traffic Signals — ELTEC". Accessed: Dec. 11, 2024. [Online]. Available: https://elteccorp.com/news/other/what-is-the-purpose-of-traffic-signals/

[2] M. Thomas, "50 Real-World Examples of Embedded Systems — Coderus Embedded Software Guides", Coderus. Accessed: Dec. 11, 2024. [Online]. Available: https://www.coderus.com/examples-of-embedded-systems/

[3] Michael Barr and Anthony Massa. Programming Embedded Systems: With C and GNU Development Tools. O'Reilly Media, 2nd edition, 2009.

[4] [4] STMicroelectronics, "NUCLEO-L476RG—STM32 Nucleo-64 Development Board with STM32L476RG MCU, Supports Arduino and ST Morpho Connectivity." Accessed: Dec. 11, 2024. [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-l476rg.html#overview

[5] Nexperia, "8-bit serial-in, serial or parallel-out shift register with output latches; 3-state", 74HC595; 74HCT595 datasheet, [Revised Mar. 2024]