VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MICROPROCESSORS - MICROCONTROLLERS

Assignment

Traffic Light using STM32F103RB

Advisor(s):	Le Trong Nhan	
Student(s):	Dinh Minh Tri	2153059
	Vo Dang Duy Tien	2053498
	Phan Le Khanh Trinh	2151268

HO CHI MINH CITY, FEBRUARY 2024

Contents

1 Introduction

Microprocessor-microprocessor is a computer processor that incorporates the functions of a central processing unit on a single integrated circuit (IC), or sometimes up to 8 integrated circuits. The microprocessor is a multipurpose, clock driven, register based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory and provides results (also in binary form) as output. In short, it is a controlling unit of micro-computer, fabricated on a small chip capable of performing ALU operation and communicating with the other devices connected to it.

As microprocessors get smaller and more affordable, an increasing number of devices become 'smart,' as a result of microprocessor implanted in them. Automated teller machines (ATMs), multifunction wristwatches, PDAs, PLCs, medical appliances, and many more applications are examples microprocessor applications. These application areas range from household appliances to communication devices. As the globe undergoes unprecedented research in hardware technology (nano-technology, quantum physics, etc.), packing greater power into a single chip will soon become a reality. As a result, microprocessor applications might be described as a dreamer's paradise with endless possibilities.

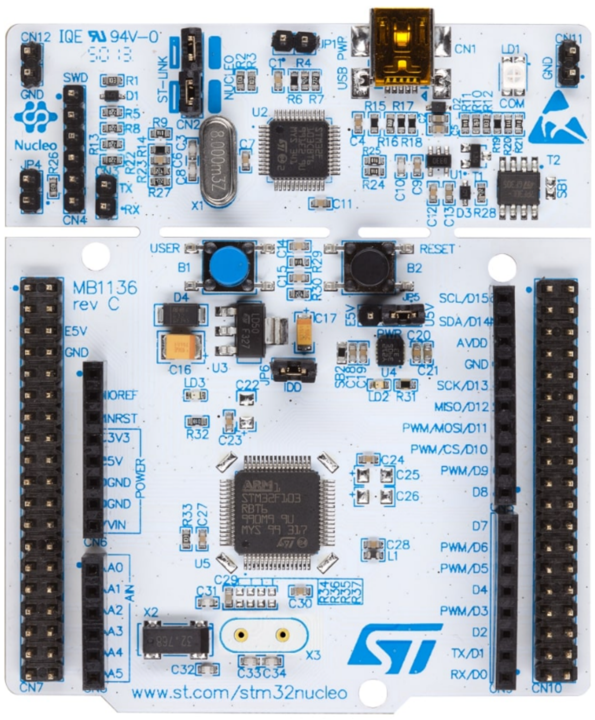


Figure 1.1: NUCLEO-F103RB

In the scope of this paper, a traffic light is implemented on the STM32106RB platform, featuring multiple functional buttons to modify the light's timing and request pedestrian passage. Additionally, a speaker is incorporated to notify when the crossing time is nearly over. This paper is divided into 4 sections: Theoretical foundation, System Finite Machine design, Implementation, Discussion and Extension.

2 Theoretical Foundations

2.1 General Purpose Input/Output (GPIO)

A GPIO (General-Purpose Input/Output) port is a versatile interface found in microcontrollers and embedded systems. It can function both as an input and an output, allowing for bidirectional communication with external devices. In its input configuration, the GPIO port receives signals from switches or sensors, conveying this information to the CPU for processing. In its output configuration, the GPIO port executes CPU instructions to drive external operations, such as illuminating an LED display or sending drive signals to a motor.

The term "general purpose" reflects the port's flexibility, as each pin can be freely set as either an input or an output, enabling various combinations based on specific requirements, unlike earlier MCU designs where ports were exclusively designated for either input or output. The function of a GPIO pin is customizable and can be controlled by the software.

Each port bit of GPIO ports can be individually configured by software in several modes:

- Input/Output mode
- Analog
- Alternative function (AF)

In this assignment, our main focus is to configure the input mode for buttons, and the output mode for traffic lights and buzzer.

Only two states are possible: HIGH or LOW. Depending on how a device is activated by a GPIO, active low or active high is defined:

- Active HIGH: the device is activated when the GPIO is HIGH.
- Active LOW: the device is activated when the GPIO is LOW.

The following image shows how a GPIO is wired to buttons or LEDs, to work as active LOW or HIGH.

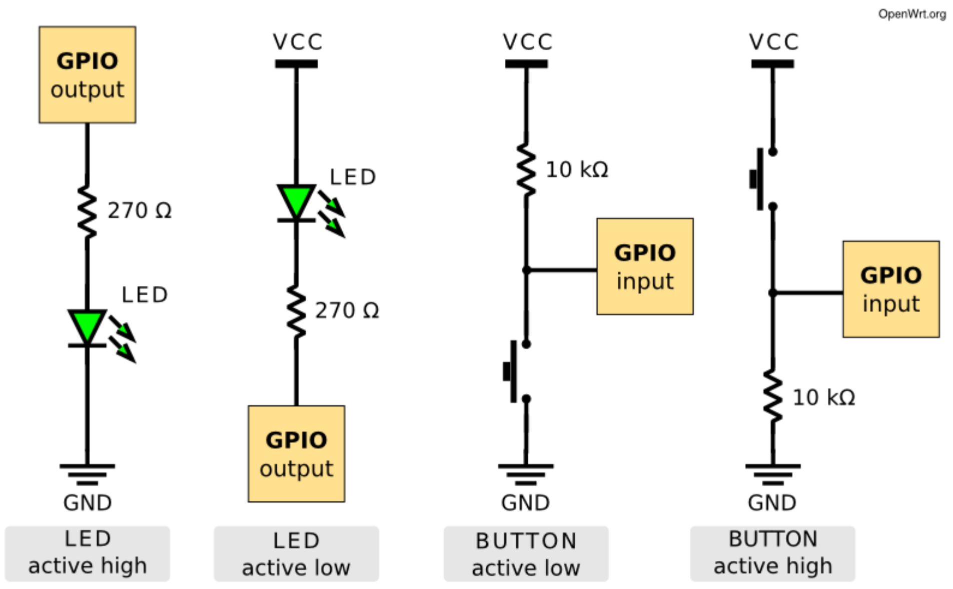


Figure 2.1: GPIO Active-high and Active-low

2.2 Timer

A Timer Module in its most basic form is a digital logic circuit that counts up every clock cycle. More functionalities are implemented in hardware to support the timer module so it can count up or down. It can have a Prescaler to divide the input clock frequency by a selectable value. It can also have circuitry for input capture, PWM signal generation.

A timer module can also operate in a counter mode where the clock source is not known, it's actually an external signal. Maybe from a push button, so the counter gets incremented every rising or falling edge from the button press.

STM32 Timers Hardware:

- Basic Timers Modules
- Low-Power Timers Modules
- General-Purpose Timers Modules
- Advanced-Control Timers Modules
- High-Resolution Timers Modules

STM32 Timers modes Of operation:

- Timer Mode
- Counter Mode
- PWM Mode
- Advanced PWM Mode
- Output Compare Mode
- One-Pulse Mode
- Input Capture Mode
- Encoder Mode
- Timer Gate Mode

To the scope of this assignment, we only use basic timers modules as well as PWM mod and Timer mode

2.3 Pulse width modulation (PWM)

Pulse width modulation reduces the average power delivered by an electrical signal by converting the signal into discrete parts. In the PWM technique, the signal's energy is distributed through a series of pulses rather than a continuously varying (analogue) signal.

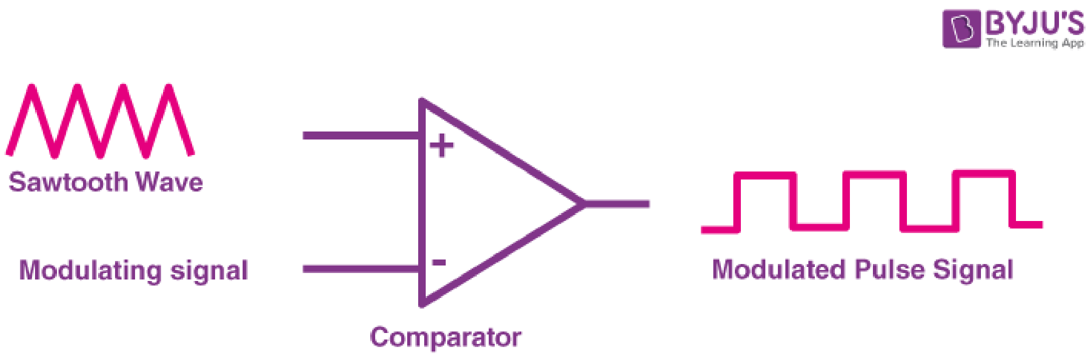


Figure 2.2: PWM

A pulse width modulating signal is generated using a comparator. The modulating signal forms one part of the input to the comparator, while the non-sinusoidal wave or sawtooth wave forms the other part of the input. The comparator compares two signals and generates a PWM signal as its output waveform.

If the sawtooth signal is more than the modulating signal, then the output signal is in a "High" state. The value of the magnitude determines the comparator output which defines the width of the pulse generated at the output.

- **Duty cycle**

As we know, a PWM signal stays "ON" for a given time and stays "OFF" for a certain time. The percentage of time for which the signal remains "ON" is known as the duty cycle. If the signal is always "ON," then the signal must have a 100- duty cycle. The formula to calculate the duty cycle is given as follows:

$$Duty\ Cycle = \frac{Turn\ On\ Time}{Turn\ On\ Time + Turn\ Off\ Time}$$

The average value of the voltage depends on the duty cycle. As a result, the average value can be varied by controlling the width of the "ON" of a pulse.

- **Frequency**

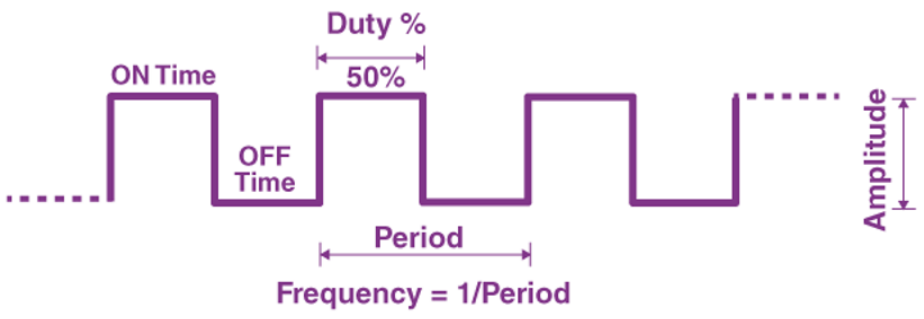


Figure 2.3: Duty Cycle

The frequency of PWM determines how fast a PWM completes a period. The frequency of a pulse is shown in the figure above.

The frequency of PWM can be calculated as follows:

$$Frequency = \frac{1}{Time\ Period}$$

$$Time\ Period = On\ Time + OFF\ time$$

2.4 Universal Asynchronous Receiver / Transmitter (UART)

UART stands for Universal Asynchronous Receiver Transmitter Protocol. U stands for Universal which means this protocol can be applied to any transmitter and receiver, and A is for Asynchronous which means one cannot use clock signal for communication of data and R and T refers to Receiver and Transmitter hence UART refers to a protocol in which serial data communication will happen without clock signal.

UART is established for serial communication.

2.4.1 UART Basics

Two wires are established here in which only one wire is used for transmission whereas the second wire is used for reception. Data format and transmission speeds can be configured here. So, before starting with the communication define the data format and transmission speed. Data format and transmission speed for communication will be defined here and we do not have a clock over here that's why it is referred to as asynchronous communication with UART protocol. Here we will see how this protocol is designed physically.

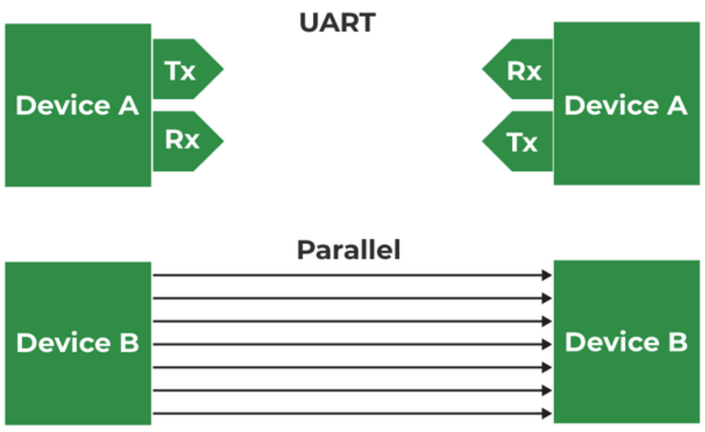


Figure 2.4: UART Basic

2.4.2 UART Data Format

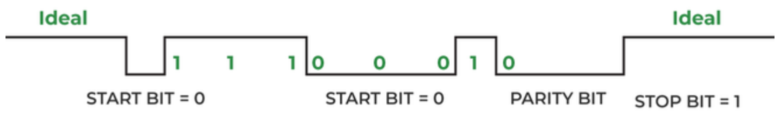


Figure 2.5: UART Data Format

Suppose DEVICE A is sending data to DEVICE B, and the transmitter of DEVICE A will send data to the receiver of Device B then it will be logic high. Now, send the start bit that will be logic 0 and once we have the start bit, DEVICE B will know that somebody is communicating. Now there is the same speed configuration with both devices. So, after the start bit, DEVICE A can forward data.

Consider 8 bits of data length, so we will be forwarding 8 bits and those 8 bits will be received by DEVICE B a parity bit can also be used which is optional, but this is quite effective. By using the parity bit, it can be identified whether the received data is correct or not. Suppose we are sending 1 1 1 0 0 0 1 0. Now, we have 4 ones; an even number of ones are there hence the parity is even and for that, logic 0 will be assigned. Suppose we are receiving data with some error, say zero is converted into one; Now incorrect data that is 1 1 1 1 0 0 1 0 for this incorrect data parity will be 0 as there are 5 ones, here is a mismatch in the parity bit and hence it is confirmed that the received data has some error.

2.5 Scheduler

In the context of microcontrollers (MCUs), a scheduler typically refers to a software component responsible for managing and coordinating the execution of tasks or processes. Microcontrollers are embedded systems that perform specific functions, and they often need to handle multiple tasks simultaneously or in a time-sensitive manner. The scheduler helps in organizing and controlling the execution of these tasks.

There are different types of schedulers, and their design can vary based on the application requirements. Some common types of schedulers in the context of MCUs include:

- Simple Task Scheduler
- Real-Time Operating System (RTOS) Scheduler

- Cooperative Scheduler
- Preemptive Scheduler

In the context of this project, we will discuss about Cooperative and Preemptive Scheduler.

2.5.1 Cooperative Scheduler

A co-operative scheduler provides a single-tasking system architecture.

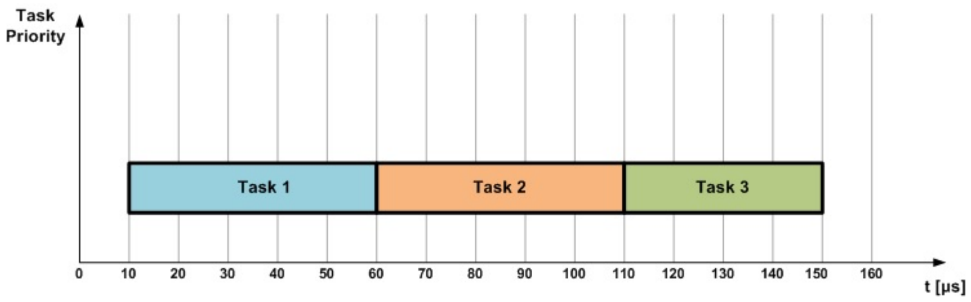


Figure 2.6: Cooperative Scheduler

If we employ a non-preemptive algorithm to schedule the three tasks,for example, the resulting behavior is illustrated in Figure 2.6. In this scenario, each task initiates its operation and completes it before the subsequent task begins.

This scheduler is simple, reliable and safe. However, the task we add latest will come at the highest of respond times. This reduces its applicability in complex real-time systems where quick and predictable task execution is essential.

2.5.2 Preemptive Scheduler

A pre-emptive scheduler provides a multitasking system architecture.

Preemptive scheduling enables the interruption of a currently executing task, allowing a task with a higher "urgent" status to take precedence. The scheduler forcibly transitions the interrupted task from the running state to the ready state. This dynamic task switching, inherent in the algorithm, constitutes a form of multitasking. It involves the assignment of priority levels to each task, enabling the interruption of a running task if a higher-priority task enters the queue.

As an example let's have three tasks called Task 1, Task 2 and Task 3. Task 1 has the lowest priority and Task 3 has the highest priority.

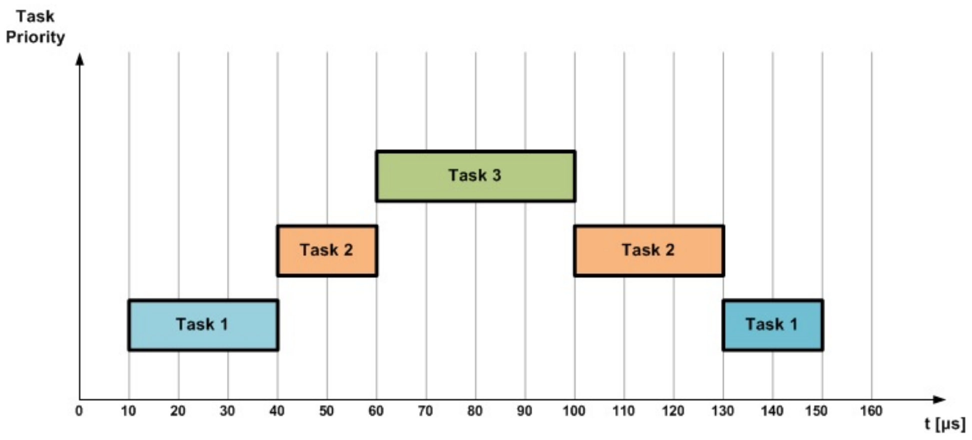


Figure 2.7: Preemptive Scheduler

In Fig. 2.7 we can see that Task 1 is the first to start executing, as it is the first one to arrive (at t = 10 μs). Task 2 arrives at t = 40 μs and since it has a higher priority, the scheduler interrupts the execution of Task 1 and puts Task 2 into running state. Task 3 which has the highest priority arrives at t = 60 μs. At this moment Task 2 is interrupted and Task 3 is put into running state. As it is the highest priority task it runs until it completes at t = 100 μs. Then Task 2 resumes its operation as the current highest priority task. Task 1 is the last to complete is operation.

To conclude, the scheduler plays a crucial role in managing the overall system performance, ensuring that tasks are executed in a timely and efficient manner. The choice of a scheduler depends on the specific requirements of the application and the complexity of the system. It is an essential component for developing reliable and responsive embedded systems based on microcontrollers.

3 Finite State Machine design

The program can be represented by a nested finite state machine with 5 main state: AUTO, MANUAL, RED, AMBER, and GREEN as shown in the diagram below:

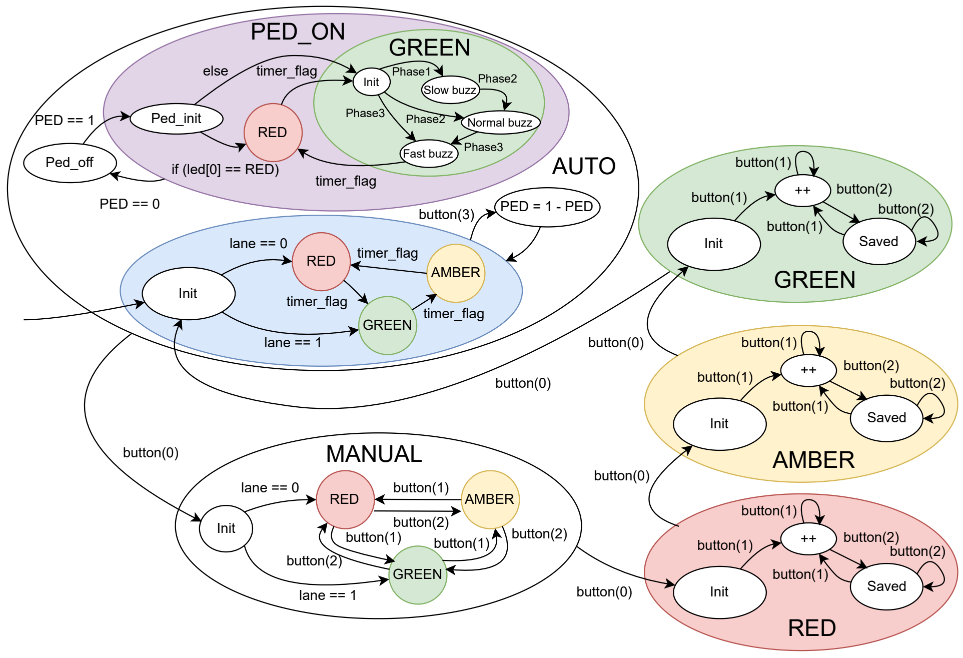


Figure 3.1: Block diagram

The five states are changed in the order indicated above when the BUTTON0 is pressed, details of each states are:

- **AUTO:** The default state of the program. Initially, the traffic lights set red light for one lane, green light for the other lane, and the duration for red light, yellow light, and green light are by default 5, 2, and 3, respectively. Then they will automatically change light when the duration of a light is expired. During the AUTO state, UART is used to print the timer to an external terminal, which is discussed in a later section.
  Additionally, another finite state machine is used in the nested state for the Pedestrian feature, which is only active when PED == 1. The pedestrian light when turned on will mimic a lane's light (which will be automatically reversed compared to the other lane). If the pedestrian light is GREEN, the speaker will have three states which changes depending on the remaining time: SLOW BUZZ, NORMAL BUZZ, and FAST BUZZ.

- **MANUAL:** This state is used to change light using BUTTON1 and BUTTON2 in stead of the timer as in the AUTO state. BUTTON1 will change the lights in forward direction, and reversely when BUTTON2 is pressed.
- **RED, AMBER, GREEN:** The remaining three states are used to change the duration of the three lights respectively, in which BUTTON1 and BUTTON2 are also in use. BUTTON1 will increase the duration by 1 second (which is limited in the range 1-99), and BUTTON2 will save the configuration. UART is also used to print the current duration of the timer, or to show if the user has saved the configuration or not by pressing BUTTON2.

4 Implementation

4.1 Project configurations

A new project is created with following configurations, concerning the UART for communications, TIM2 for timer, and TIM3 for pulse width modulation generation.

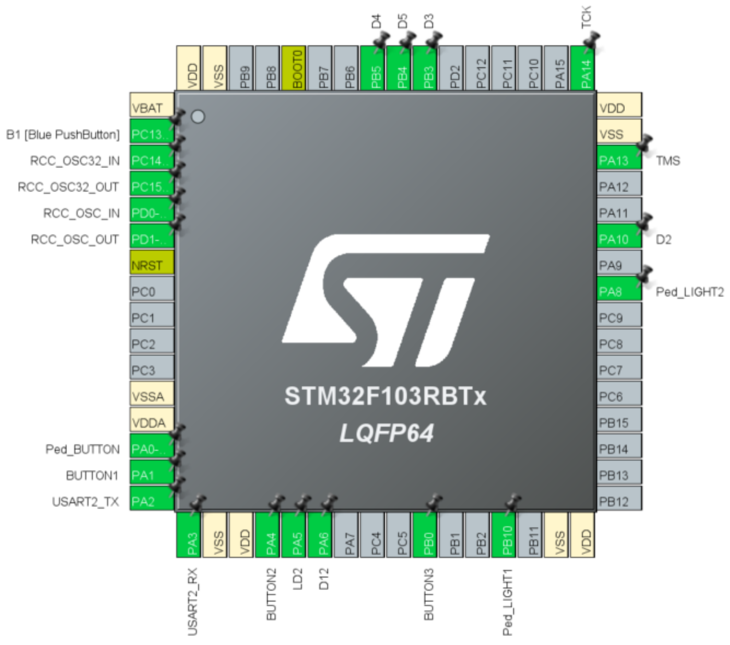
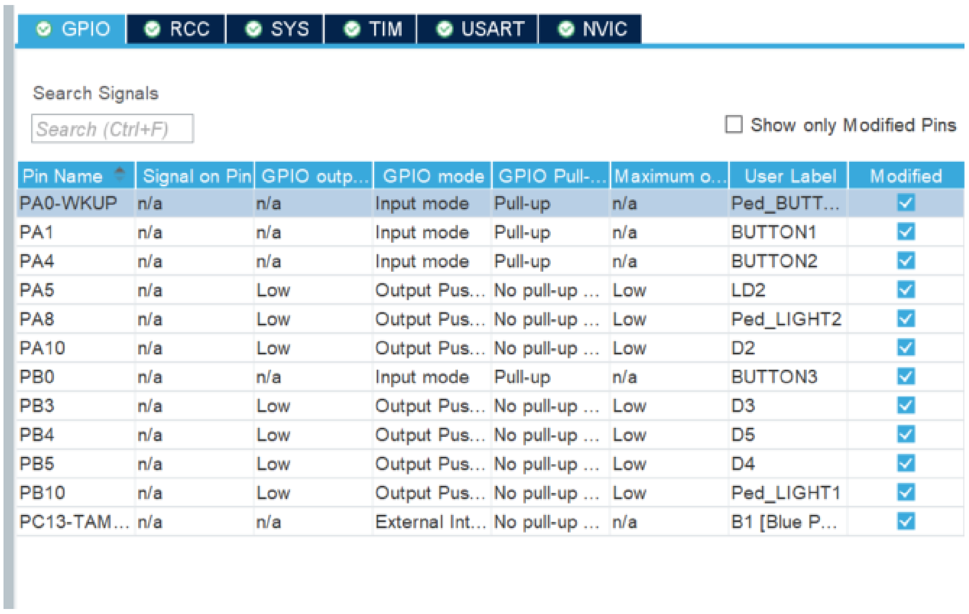


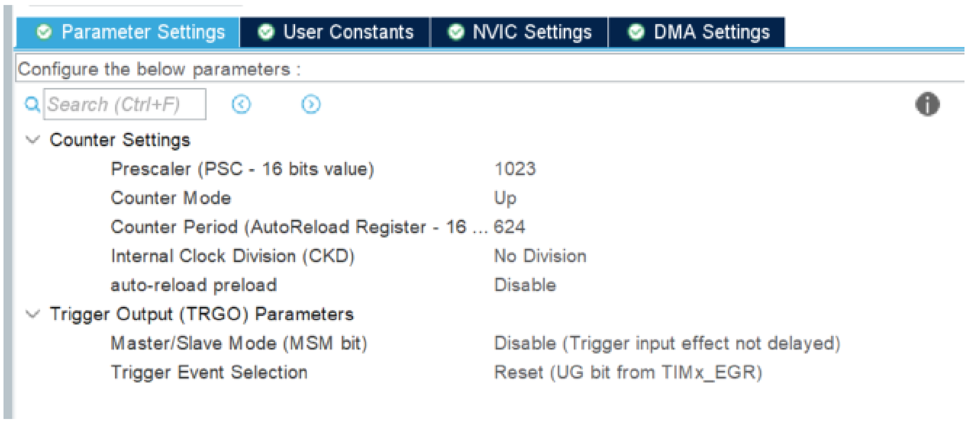
Figure 4.1: Pinout configuration

In which, the GPIO, TIM2, TIM3, and UART are configured as the following:



Pin Name	Signal on Pin	GPIO outp...	GPIO mode	GPIO Pull-...	Maximum o...	User Label	Modified
PA0-WKUP	n/a	n/a	Input mode	Pull-up	n/a	Ped_BUTT...	✓
PA1	n/a	n/a	Input mode	Pull-up	n/a	BUTTON1	✓
PA4	n/a	n/a	Input mode	Pull-up	n/a	BUTTON2	✓
PA5	n/a	Low	Output Pus...	No pull-up ...	Low	LD2	✓
PA8	n/a	Low	Output Pus...	No pull-up ...	Low	Ped_LIGHT2	✓
PA10	n/a	Low	Output Pus...	No pull-up ...	Low	D2	✓
PB0	n/a	n/a	Input mode	Pull-up	n/a	BUTTON3	✓
PB3	n/a	Low	Output Pus...	No pull-up ...	Low	D3	✓
PB4	n/a	Low	Output Pus...	No pull-up ...	Low	D5	✓
PB5	n/a	Low	Output Pus...	No pull-up ...	Low	D4	✓
PB10	n/a	Low	Output Pus...	No pull-up ...	Low	Ped_LIGHT1	✓
PC13-TAM...	n/a	n/a	External Int...	No pull-up ...	n/a	B1 [Blue P...	✓

Figure 4.2: GPIO configuration with Output being Output Push Pull and Input mode having Pull-up configuration



Configure the below parameters :

Counter Settings

Prescaler (PSC - 16 bits value)	1023
Counter Mode	Up
Counter Period (AutoReload Register - 16 ...	624
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

Trigger Output (TRGO) Parameters

Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

Figure 4.3: Both TIM2 and TIM3 are configured with the same Prescaler and Counter of 1023 and 624 respectively

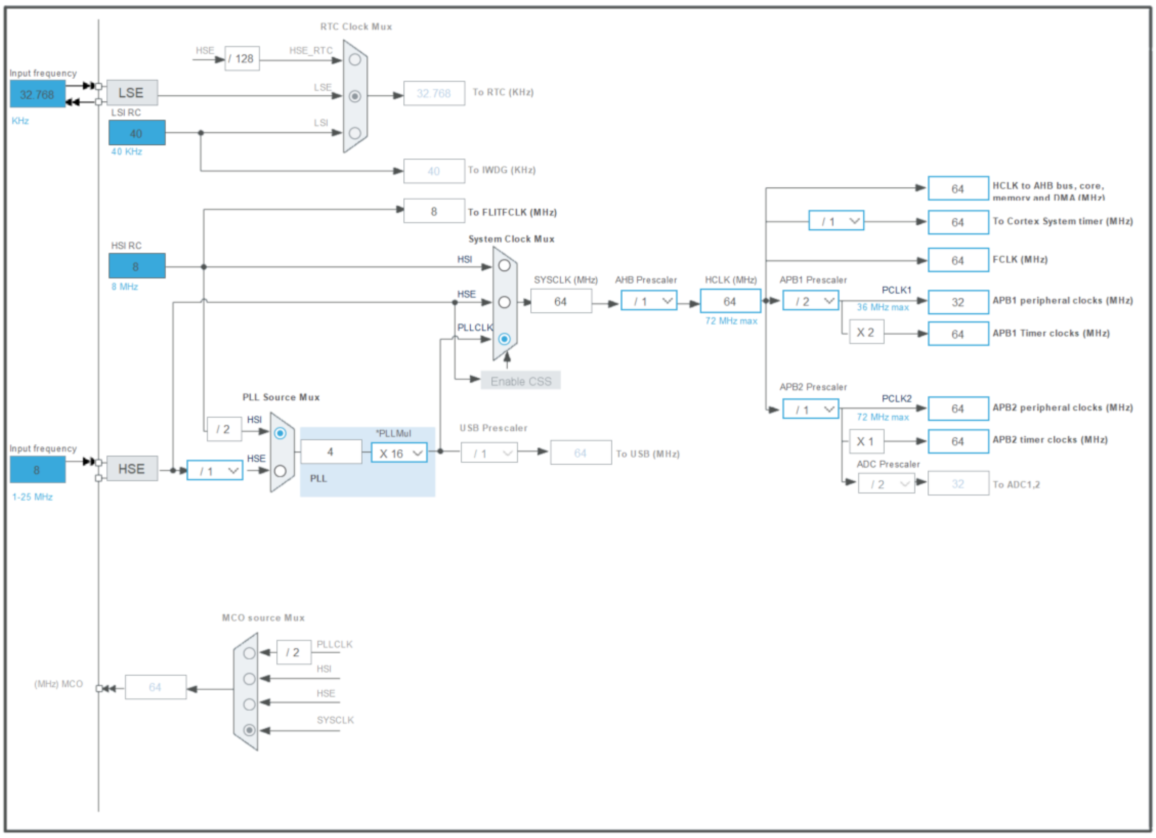


Figure 4.4: Clock configuration

4.1.1 Finite State Machine for AUTO state

The finite state machine has 5 states: 1 initial state, 1 off state, and 3 states according to 3 color of the light. This finite state machine is designed in such a way so that it is reusable for lights of both lanes, and also for the pedestrian light.

```
//Helper functions
int initMode[3] = {RED, GREEN, OFF};

int initDuration(int lane){
  return (lane == 1)? GREEN_DURATION : RED_DURATION;
}

//fsm_automatic_run
void fsm_automatic_run(int lane){
    //Switch case depending on the light's current mode
    switch(LED_MODE[lane]){
        case INIT:
            //Turn off all lights
            setTrafficLight(lane, OFF);

            //Change mode and duration
            LED_MODE[lane] = initMode[lane];
            setTimer(lane, initDuration(lane)*100);
            break;

        case RED:
            setTrafficLight(lane, RED);

            //Timer flag to change light
            if(timer_flag[lane] && TRAFFIC_MODE != MANUAL){
                    LED_MODE[lane] = GREEN;
                    setTimer(lane, GREEN_DURATION*100);
            }
            break;

        case AMBER:
            if (lane == 2) setTrafficLight(lane, GREEN);
            else setTrafficLight(lane, AMBER);

            //Timer flag to change light
            if(timer_flag[lane] && TRAFFIC_MODE != MANUAL){
                    LED_MODE[lane] = RED;
                    setTimer(lane, RED_DURATION*100);
            }

            break;

        case GREEN:
            setTrafficLight(lane, GREEN);

            //Timer flag to change light
            if(timer_flag[lane] && TRAFFIC_MODE != MANUAL){
                    LED_MODE[lane] = AMBER;
                    setTimer(lane, AMBER_DURATION*100);
            }

            break;
        case OFF:
            break;
        default:
            break;
    }
}
```

In general, the function is used to set the light to the desired color, and switch state when the duration expired. As mentioned, this FSM is designed to be reusable, and it is used for both the traffic lights, and the pedestrian light (in line 24 and 25):

```
if (lane == 2) setTrafficLight (lane, GREEN);
else setTrafficLight (lane, AMBER);
```

In AMBER mode, if it is pedestrian light, set light to GREEN, else set to AMBER.

4.1.2 Finite State Machine for MANUAL state

Despite the name that this is only for the MANUAL state, this FSM can be considered the parent FSM, because it also controls the three TUNING states, and the condition for the FSMs of AUTO state and its PED state to run.

```
void fsm_manual_run(){
  switch(TRAFFIC_MODE){
    case AUTO:
        //Display countdown via UART
        displayCountdown();

        //Change state
        if (isButtonPressed(0)){
            changeMode(MANUAL, 0, INIT);
            displayUART(MANUAL, 0, 0, LIGHT[manualMode[0]], LIGHT[manualMode[1]], 0, 0);
        }
```

```
      break;

  case MANUAL:
    //Change state
    if (isButtonPressed(0)) changeMode(RED, RED_DURATION, OFF);

      //Change lights in forward direction
    if (isButtonPressed(1)){
      manualPlus(0);
      manualPlus(1);
      displayUART(MANUAL, 0, 0, LIGHT[manualMode[0]], LIGHT[manualMode[1]], 0, 0);
    }

      //Change lights in reversed direction
    if (isButtonPressed(2)){
      manualMinus(0);
      manualMinus(1);
      displayUART(MANUAL, 0, 0, LIGHT[manualMode[0]], LIGHT[manualMode[1]], 0, 0);
    }
    break;

  case RED:
    //Change state
    if (isButtonPressed(0)) changeMode(AMBER, AMBER_DURATION, OFF);

    //Change duration
    if (isButtonPressed(1)) changeDuration(RED);

    //Save duration
    if (isButtonPressed(2)){
      RED_DURATION = tempDuration;
      checkDuration(RED);
```

```
        display(SAVED, tempDuration);
      }

      //Toggle lights
      if (timer_flag[3]) toggle(RED);
      break;

    case AMBER:
      //Similar to state RED with changed parameters
                //...
                break;

    case GREEN:
      //Similar to state RED with changed parameters
                //...
                break;

    default:
       break;
  }
}
```

4.1.3 Finite State Machine for PED mode

The buzzer is implemented in this FSM, if PED_MODE is in AUTO and the light is not RED. In this program, the function is divided into three periods, with slow, normal, and fast buzzer sound, automatically change state when timer goes down.

```
void fsm_ped_run(){
    switch(PED_MODE){
    case INIT:
        if (PED == 1){
            turnPedLed(PED);
            PED_MODE = AUTO;
        }
        else {
            turnPedLed(PED);
            PED_MODE = OFF;
        }
        setTimer(5,100);
        break;

    //Off mode
    case OFF:
        __HAL_TIM_SET_AUTORELOAD(&htim3, 5*localVal);
        __HAL_TIM_SET_COMPARE(&htim3,TIM_CHANNEL_1, 0.6 *
(5*0));
        localVal = MAX;
        if (isButtonPressed(3)){
            //Change mode
            if (TRAFFIC_MODE == AUTO){
                //Mimic lane 0 light
                turnPedLed(1);
                setTimer(4,(int)(ceil((GREEN_DURATION +
AMBER_DURATION)*100/3)));
                PED = 1;
                PED_MODE = AUTO;
            }
        }
        break;

    //On mode
    case AUTO:
        //TODO
            if (timer_flag[4]){
                if (LED_MODE[2] != RED){
                    __HAL_TIM_SET_AUTORELOAD(&htim3, 5*
localVal);
                    __HAL_TIM_SET_COMPARE(&htim3,
TIM_CHANNEL_1, 0.6 * (5*localVal));
                    localVal = localVal - MAX/3;
                    if (localVal < 0 ){
                        localVal = MAX;
```

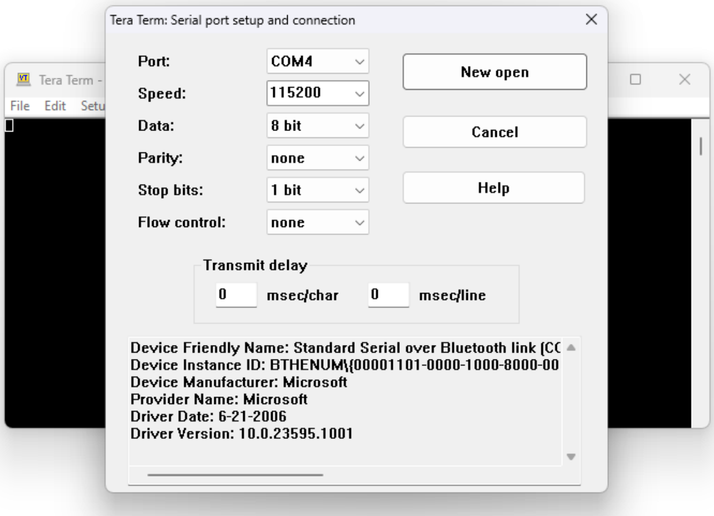
```c
                    }
                }
            }
            if (timer_flag[4]) setTimer(4,(int)(ceil((GREEN_DURATION + AMBER_DURATION)*100/3)));
            if (LED_MODE[2] == RED){
                localVal = MAX;
                __HAL_TIM_SET_AUTORELOAD(&htim3, 5*localVal);
                __HAL_TIM_SET_COMPARE(&htim3,TIM_CHANNEL_1, 0.6 * (5*0));
            }

        if (isButtonPressed(3)){
            //Turn off ped light
            turnPedLed(0);

            //Change mode
            PED = 0;
            PED_MODE = OFF;
        }
        break;
    default:
        break;
    }
}
```

4.1.4 UART Transmit

- **Tera Term setup and connection**

UART in this assignment is used to replace the function of a normal 7-SEG LED, which normally shows the countdown of the lights, or the duration when in tuning mode.

To test the implementation of the UART, Tera Term (which is an open-source, free, software implemented, terminal emulator program) will be used.



Port: COM4
Speed: 115200
Data: 8 bit
Parity: none
Stop bits: 1 bit
Flow control: none

Transmit delay
0 msec/char 0 msec/line

Device Friendly Name: Standard Serial over Bluetooth link (CC
Device Instance ID: BTHENUM\{00001101-0000-1000-8000-00
Device Manufacturer: Microsoft
Provider Name: Microsoft
Driver Date: 6-21-2006
Driver Version: 10.0.23595.1001

Figure 4.5: Serial port setup and configuration

- **Source code**

To make the output more readable, the terminal will be cleared and cursor set to original position before printing the new output using the following code:

```c
HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[2J", 0x1b), 100);
HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[0;0H", 0x1b), 100);
```

All the transmission of the program is put into one file uart_timer.c with the function displayUART() as the following:

```c
void displayUART(int mode, int num0, int num1, int led0, int led1, int led2, int pedLed){
    //Current light mode
    char* curMode0 = (led0 == RED)? "RED" : (led0 == AMBER)? "AMBER" : "GREEN";
    char* curMode1 = (led1 == RED)? "RED" : (led1 == AMBER)? "AMBER" : "GREEN";
    char* curMode2 = (led2 == RED)? "RED" : "GREEN";

    switch (mode){
        //Show countdown of the lights
        case COUNTDOWN:
            if (num0 != 0 && num1 != 0){
                HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[2J", 0x1b), 100);
                HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[0;0H", 0x1b), 100);
                HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "Lane 0: %s\n\rCountdown: %d\n\r", curMode0, num0), 100);
                HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "Lane 1: %s\n\rCountdown: %d\n\r", curMode1, num1), 100);
                if (pedLed == AUTO){
                    int num2 = (led0 == GREEN)? num0 + 2: num0;
                    HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "Ped light: %s\n\rCountdown: %d\n\r", curMode2, num2), 100);
                }
            }
            break;

        //Show tuning duration
        case RED:
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[2J", 0x1b), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[0;0H", 0x1b), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "!TUNING#\n\r"), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "!RED=%d#\n\r", num0), 100);
            break;
        case AMBER:
            //Similar to state RED with changed parameters
            //...
            break;
        case GREEN:
            //Similar to state RED with changed parameters
            //...
            break;

        //Button2 is pressed to save the duration
        case SAVED:
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "!SAVED#\n\r"), 100);
            break;

        //Show lights in manual mode
        case MANUAL:
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[2J", 0x1b), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "%c[0;0H", 0x1b), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "!MANUAL#\n\r"), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "Lane 0: %s\n\r", curMode0), 100);
            HAL_UART_Transmit(&huart2, (void*) str, sprintf(str, "Lane 1: %s\n\r", curMode1), 100);
            break;

        default:
            break;
    }
}
```

5 Discussion and Extension

The buzzer in this report is implemented into three phases, each with the duration:

$$PHASE\_DURATION = \frac{RED\_DURATION}{3}$$

When tested, sometimes it happens to be a delay when switching phases, causing the speaker to not working continuously, with a silent period between phases. This bug may origin from the fact that the PHASE_DURATION is the result of the division of 3, therefore, it may be irrational and leads to delay in software_timer. If given more time, a better implementation will be used for the buzzer.

In this report, we chose to go with implementing each lane's lights and pedestrian light being a separate finite state machine. The advantage of this is the reduction in code we have to work with, which we can reuse the code for different use-case, but there is an increase in the logic of the code one has to work out. There is another implementation, which merges all the lanes and pedestrian light into only one finite state machine with much more states. One possible extension of the project is to compare the performance of two different strategies to determine which is the better option to use for the assignment

6 Source code

The page can be accessed here: https://github.com/stryz-0709/MCUAssignment231