

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



MICROPROCESSORS-MICROCONTROLLERS (CO3009)

Assignment (Semester: 231)

“Traffic Light System with Integrated Pedestrian Light”

Advisor: Le Trong Nhan.
Students: Le Quoc Tuan - 2153944.
Tran Nguyen Gia Huy - 2153395.
Nguyen Van Duc Long - 2153535.

HO CHI MINH CITY, DECEMBER 2023



Contents

1	Introduction	2
1.1	Member list and workload	2
1.2	Project description	2
2	Requirements	3
2.1	Software requirements	3
2.2	Hardware requirements	3
3	Theory background	4
3.1	Finite state machine	4
3.2	Button debounce	4
3.3	UART communication	5
3.3.1	UART Basics:	5
3.3.2	STM32 UART Hardware:	5
3.3.3	Configuring UART in STM32:	5
3.3.4	Transmission and Reception:	5
3.3.5	UART Communication Modes:	6
3.3.6	Error Handling and Flow Control:	6
3.3.7	Application and Use Cases:	6
3.4	Cooperative scheduler	6
3.4.1	What is a scheduler ?	6
3.4.2	Cooperative scheduler:	6
3.4.3	Key components:	7
4	System Design	8
4.1	Finite state machine	8
4.1.1	FSM for Mode 1 - Automatic mode	8
4.1.2	FSM for Mode 2, 3, and 4 - Manual modes	8
4.2	Pedestrian Light	10
4.3	UART communication	11
5	Implementation	12
5.1	FSM for automatic mode	12
5.2	FSM for manual modes	14
5.3	FSM for button processing	16
5.4	Button debounce	18
5.5	Pedestrian Light	20
5.5.1	On and Off Pedestrian Light functions	20
5.5.2	Pedestrian Light Configuration	20
5.6	Buzzer	21
5.7	UART communication	24
5.8	Cooperative scheduler	25
6	Conclusion	26
6.1	System evaluation	26
6.2	Future system development	26

1 Introduction

1.1 Member list and workload

No.	Fullname	Student ID	Problems	Contribution
1	Lê Quốc	2153944	- Traffic light - UART communication - Report	33.33%
2	Trần Nguyễn Gia Huy	2153395	- Pedestrian buzzer - Cooperative scheduler - Report	33.33%
3	Nguyễn Văn Đức Long	2153535	- Pedestrian light - Cooperative scheduler - Report	33.33%

1.2 Project description

In this project, the STM32F103RB is used to simulate the 2-way traffic light system, having some main features:

- **Automatic mode:** The system operates as normal. The light colors are red, yellow, and green.
- **Manual mode:** 3 buttons are used to adjust the traffic light following the logic bellow:
 - Button 1: Change between 4 modes (will be specified later in this report).
 - Button 2: Modify the time for each color traffic light.
 - Button 3: Set the chosen value.
- **Pedestrian light & buzzer:** 1 led & 1 buzzer are used to administer the pedestrian light following the logic:
 - Whenever the button is pressed, the led is turned on reversely to the traffic light.
 - When the light is green, the buzzer sounds with increasing volume and frequency.
- **UART communication:** Display traffic light time.
- **Cooperative scheduler:** Used to operate the whole system.

Here is the GitHub link of our project: https://github.com/tuanle186/MCU_Assignment

2 Requirements

2.1 Software requirements

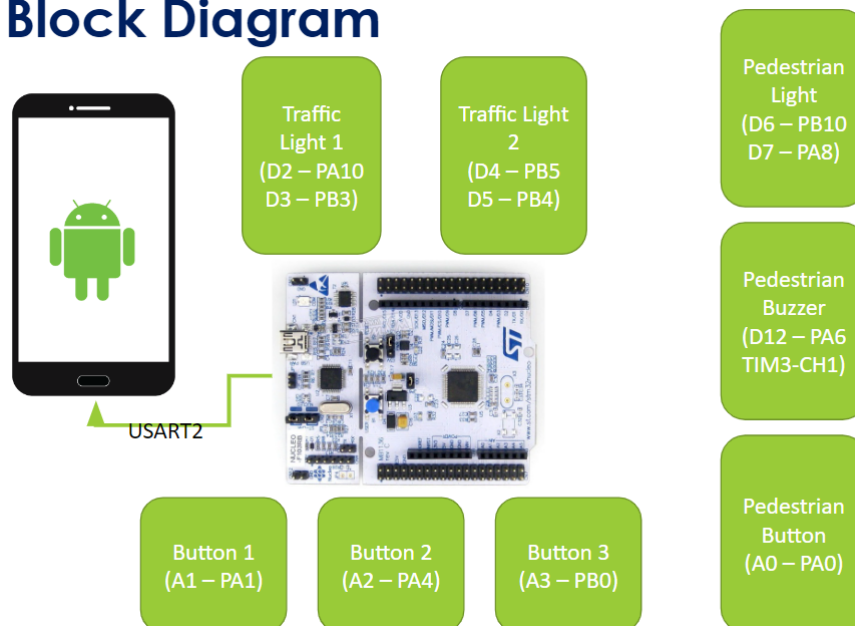
Implement a program that operates effectively and smoothly according to logic requirements.
Some softwares are required:

- STM32CubeIDE: For programming.
- STM32CubeProgrammer: For loading *.hex* file into STM32 microcontroller devices.
- PuTTY: For displaying UART Communication.

2.2 Hardware requirements

- STM32F103RB with NUCLEO-F103RB board.
- For traffic light: 2 lights for 2 lanes, each included 3 different colors LED (red, green, yellow) and 3 buttons.
- For pedestrian light: a LED that can display 2 colors (green & red), 1 buzzer and 1 button to control pedestrian light behaviour.
- All the component is connect to the board follow the block diagram below:

Block Diagram



Hình 1: Block diagram

3 Theory background

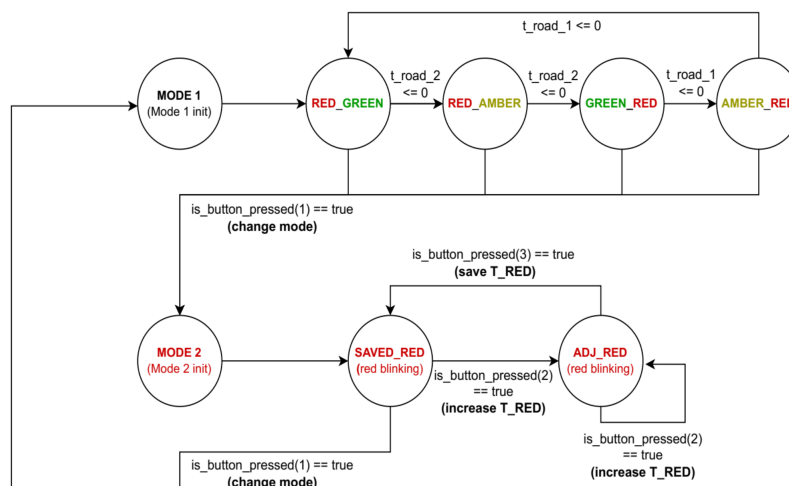
3.1 Finite state machine

A finite state machine (FSM) is a computational model used to design and illustrate the behavior of systems that have a finite number of states. It's comprised of a set of states, transitions between these states, and actions or conditions associated with each transition.

Here's a breakdown of the components of a finite state machine:

- **States:** States are the different conditions or situations that the system can be in at any given time.
- **Transitions:** Transitions describe the movement or change of the system from one state to another based on certain inputs or conditions.
- **Inputs(Conditions):** Inputs are the signals(conditions) that drive the system's transitions from one state to another.

For example:



Hình 2: Part of our FSM

In this case, States are *MODE 1*, *RED_GREEN*,.... Transitions are represented by all the arrows. Meanwhile, Conditions are *is_button_pressed(1)==true*, *t_road_2 <= 0*,...

3.2 Button debounce

- **Bouncing Issue:** When a mechanical switch is activated, it generates multiple rapid electrical signals (bouncing) which can cause the MCU to interpret multiple false presses.
- **Debouncing Methods:**
 - Hardware solutions like special ICs or RC circuitry can eliminate bouncing, but software debouncing is commonly used due to its cost-effectiveness.
 - Software-based debouncing involves establishing a minimum criterion for a valid button push, usually involving time differences.
- **Time Considerations:** Bounce-time typically lasts within 10ms. Button-press-time ranges between 50 and 100ms. Response-time generally noticeable if it exceeds 100ms after button press.

- **Method used in this assignment:** Regularly read buttons at intervals greater than 10ms but less than or equal to 50ms. Interpret three possible outcomes: solid '0' state, solid '1' state, or bouncing state.

3.3 UART communication

UART (Universal Asynchronous Receiver/Transmitter) communication in STM32 microcontrollers is a widely used serial communication protocol that enables data transfer between the MCU and other devices, such as sensors, displays, or other microcontrollers. Here is a theory on UART communication specifically tailored to STM32:

3.3.1 UART Basics:

- UART is a serial communication protocol that transmits data serially, one bit at a time, using two lines: a transmit line (TX) and a receive line (RX).
- It is asynchronous, meaning there is no separate clock signal; instead, data is transmitted using start and stop bits, alongside the actual data bits.

3.3.2 STM32 UART Hardware:

- STM32 microcontrollers have built-in hardware support for UART communication with multiple UART interfaces (USART1, USART2, etc.).
- Each UART interface is associated with specific pins on the microcontroller and can be configured for communication through the GPIO (General Purpose Input/Output) peripheral.

3.3.3 Configuring UART in STM32:

Initialization involves configuring various parameters:

- Baudrate: Determines the speed of data transmission.
- Data bits, stop bits, and parity: Configure the number of data bits, stop bits, and whether parity is used for error checking.
- Hardware flow control: Optional control lines (RTS/CTS) for flow control.
- Interrupts or polling: Choose between interrupt-driven or polling-based communication handling.

3.3.4 Transmission and Reception:

- Data transmission:
 - Data to be transmitted is written to the UART transmit buffer.
 - The UART hardware handles the transmission of data bits, start and stop bits automatically.
- Data reception:
 - Received data is stored in the UART receive buffer.
 - The microcontroller can retrieve the received data from the buffer for processing.

3.3.5 UART Communication Modes:

- Full-duplex: Simultaneous transmission and reception.
- Half-duplex: Alternating between transmission and reception, useful in certain communication scenarios.

3.3.6 Error Handling and Flow Control:

- Error flags: UART hardware provides error flags for parity errors, framing errors, or overrun errors.
- Flow control: Hardware flow control (RTS/CTS) or software-based flow control can be employed to manage data flow between devices.

3.3.7 Application and Use Cases:

UART is widely used in various applications such as IoT devices, sensor interfacing, communication between microcontrollers, and more due to its simplicity and versatility.

3.4 Cooperative scheduler

3.4.1 What is a scheduler ?

There are two ways of viewing a scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis.
- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks.

As a result, only one timer needs to be initialized, and any changes to the timing generally requires only one function to be altered.

Furthermore, we can generally use the same scheduler whether we need to execute one, ten or 100 different tasks.

3.4.2 Cooperative scheduler:

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

- **Operation:**
 - Tasks are scheduled to run at specific times (either on a periodic or one-shot basis)
 - When a task is scheduled to run it is added to the waiting list
 - When the CPU is free, the next waiting task (if any) is executed
 - The task runs to completion, then returns control to the scheduler
- **Implementation:**
 - The scheduler is simple and can be implemented in a small amount of code
 - The scheduler must allocate memory for only a single task at a time
 - The scheduler will generally be written entirely in a high-level language
 - The scheduler is not a separate application; it becomes part of the developer's code
- **Performance:** Obtaining rapid responses to external events requires care at the design stage Reliability and safety

3.4.3 Key components:

A scheduler has the following key components:

- The scheduler data structure.
- An initialization function.
- A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
- A function for adding tasks to the scheduler.
- A dispatcher function that causes tasks to be executed when they are due to run.
- A function for removing tasks from the scheduler (not required in all applications)

Co-operate scheduling is simple, predictable, reliable and safe.

4 System Design

4.1 Finite state machine

There are 4 finite state machines (FSMs) for the 4 modes. These 4 FSMs are connected together to form the main FSM of the whole system (Figure 2). Two variables are used to keep track of the state changing, which are t_road_1 and t_road_2 (remaining light time of road 1 and 2, respectively). Each FSM and its states are explained detailedly in this section.

4.1.1 FSM for Mode 1 - Automatic mode

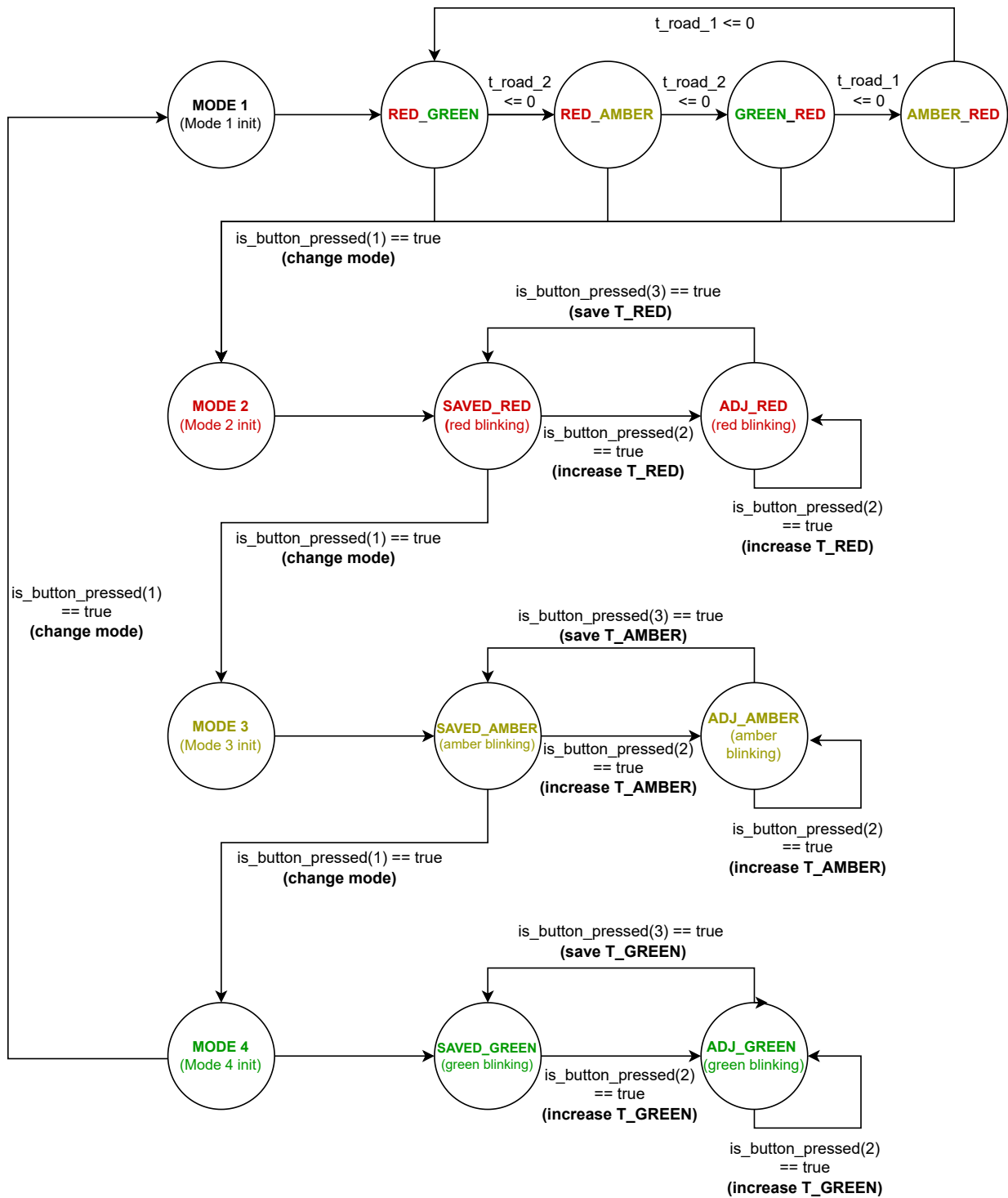
This FSM is responsible for controlling the automatic mode of the traffic light. There is 1 state for initialization and 4 states for automatic traffic light behaviours.

- **MODE_1:** This is the initializing state of the automatic FSM. In this state, appropriate set-ups required for the automatic traffic light are carried out. After having finished initializing, the state is switched to **RED_GREEN**.
- **RED_GREEN:** LEDs are configured to match the behaviour of this state, road 1's light is **RED** and road 2's light is **GREEN**. The state is switched to **RED_AMBER** when t_road_2 reaches 0.
- **RED_AMBER:** LEDs are configured to match the behaviour of this state, road 1's light is still **RED** and road 2's light is **AMBER**. The state is switched to **GREEN_RED** when t_road_1 or t_road_2 reaches 0.
- **GREEN_RED:** LEDs are configured to match the behaviour of this state, road 1's light is **GREEN** and road 2's light is **RED**. The state is switched to **AMBER_RED** when t_road_1 reaches 0.
- **AMBER_RED:** LEDs are configured to match the behaviour of this state, road 1's light is **AMBER** and road 2's light is **RED**. The state is switched to **AMBER_RED** when t_road_1 or t_road_2 reaches 0.

4.1.2 FSM for Mode 2, 3, and 4 - Manual modes

The FSM design for these modes are the same. There are a total of 3 states for each FSM (Hình 1) and they are explained as follows:

- **MODE_2, MODE_3, and MODE_4:** These are the initializing states for the manual modes. In these states, appropriate set-ups required for the manual modes are executed. After having finished initializing, the state is switched to **SAVED_XXX** state.
- **SAVED_RED, SAVED_AMBER, and SAVED_GREEN:** These states indicate that the time duration of the blinking LED has been saved and can be switched to the next mode by pressing button 3. However, if button 2 is pressed, the time duration of the blinking led is increased by one second and the state is switched to **ADJ_XXX**.
- **ADJ_RED, ADJ_AMBER, and ADJ_GREEN:** These states indicate that the time duration of the blinking led is being adjusted and is only saved when button 3 is pressed. When button 2 is pressed, the time duration of the blinking led is increased by one second. When button 3 is pressed, the time duration of this LED is saved and other LEDs' time duration is also recalculated to match the logic of the traffic light ($T_RED = T_AMBER + T_GREEN$).



Hình 3: Finite state machine for Traffic Light System

4.2 Pedestrian Light

There are four components that are used to build the pedestrian feature: an LED, a buzzer, and a button.



Hình 4: Pedestrian light in real life



Hình 5: Pedestrian buzzer & button in real life

The pedestrian feature of the system is designed as follows:

- The pedestrian light is turned on only when the button is clicked
- The pedestrian light's color is always opposite to the traffic light. If the traffic light is green or yellow, the pedestrian light is red. If the traffic light is red, the pedestrian light is green.
- When the pedestrian light is turned on and it is green (the pedestrian is allowed to cross the road), the buzzer will generate a beeping sound with an increasing frequency to warn the pedestrian about the remaining time that he/she is allowed to cross the road. The faster the beeping sound is, the less time he/she has to cross the road.
- The pedestrian light and the buzzer is turned off immediately when the pedestrian light's color turns from green to red.

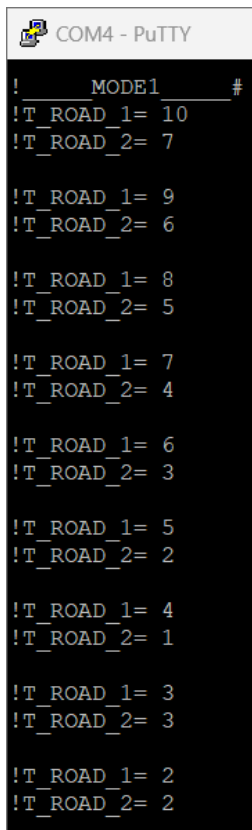
In order to implement all the above principles of the pedestrian light, the current status of the main finite state machine (as described in section 4.1) is used to track and implement the finite state machine of the pedestrian feature.

4.3 UART communication

UART communication in this project is responsible for displaying the current status and information of the system, which includes:

- The current mode (e.g. !____MODE1____#)
- The remaining time of the traffic light of the 2 roads (e.g. !T_ROAD_1 = 10#)
- The time being adjusted when the state is in manual modes. (e.g. !T_RED = 10#)

Here are some demonstrations of the UART communication:



```

!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

!T_ROAD_1= 8
!T_ROAD_2= 5

!T_ROAD_1= 7
!T_ROAD_2= 4

!T_ROAD_1= 6
!T_ROAD_2= 3

!T_ROAD_1= 5
!T_ROAD_2= 2

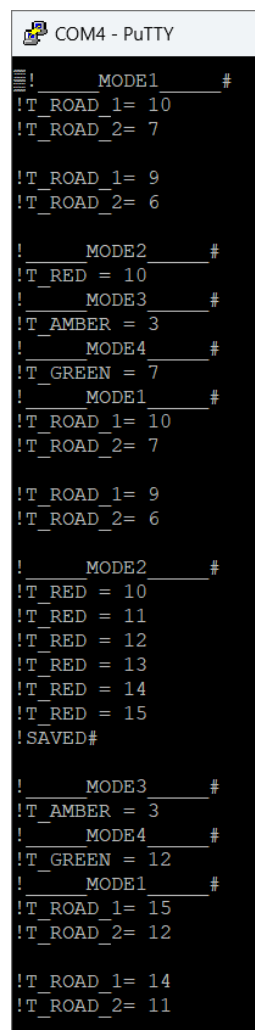
!T_ROAD_1= 4
!T_ROAD_2= 1

!T_ROAD_1= 3
!T_ROAD_2= 3

!T_ROAD_1= 2
!T_ROAD_2= 2

```

Hình 6: MODE 1



```

!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

!____MODE2____#
!T_RED = 10
!____MODE3____#
!T_AMBER = 3
!____MODE4____#
!T_GREEN = 7
!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

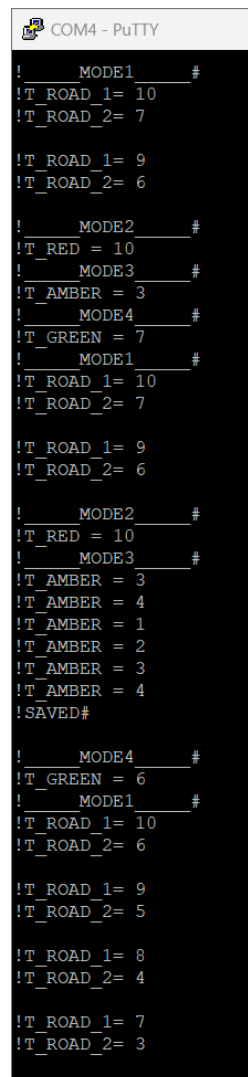
!____MODE2____#
!T_RED = 10
!T_RED = 11
!T_RED = 12
!T_RED = 13
!T_RED = 14
!T_RED = 15
!SAVED#

!____MODE3____#
!T_AMBER = 3
!____MODE4____#
!T_GREEN = 12
!____MODE1____#
!T_ROAD_1= 15
!T_ROAD_2= 12

!T_ROAD_1= 14
!T_ROAD_2= 11

```

Hình 7: MODE 2



```

!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

!____MODE2____#
!T_RED = 10
!____MODE3____#
!T_AMBER = 3
!T_AMBER = 4
!T_AMBER = 1
!T_AMBER = 2
!T_AMBER = 3
!T_AMBER = 4
!SAVED#

!____MODE4____#
!T_GREEN = 6
!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 6

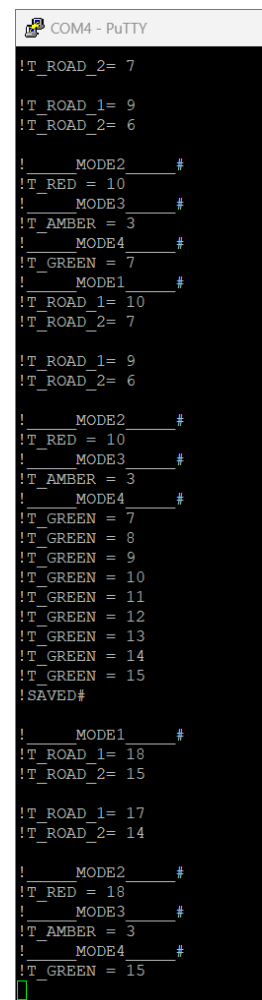
!T_ROAD_1= 9
!T_ROAD_2= 5

!T_ROAD_1= 8
!T_ROAD_2= 4

!T_ROAD_1= 7
!T_ROAD_2= 3

```

Hình 8: MODE 3



```

!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

!____MODE2____#
!T_RED = 10
!____MODE3____#
!T_AMBER = 3
!____MODE4____#
!T_GREEN = 7
!____MODE1____#
!T_ROAD_1= 10
!T_ROAD_2= 7

!T_ROAD_1= 9
!T_ROAD_2= 6

!____MODE2____#
!T_RED = 10
!____MODE3____#
!T_AMBER = 3
!____MODE4____#
!T_GREEN = 7
!T_GREEN = 8
!T_GREEN = 9
!T_GREEN = 10
!T_GREEN = 11
!T_GREEN = 12
!T_GREEN = 13
!T_GREEN = 14
!T_GREEN = 15
!SAVED#

!____MODE1____#
!T_ROAD_1= 18
!T_ROAD_2= 15

!T_ROAD_1= 17
!T_ROAD_2= 14

!____MODE2____#
!T_RED = 18
!____MODE3____#
!T_AMBER = 3
!____MODE4____#
!T_GREEN = 15

```

Hình 9: MODE 4

5 Implementation

The general system design has been discussed in the previous section. In this section, the implementation of the aforementioned design is explained in details.

5.1 FSM for automatic mode

As mentioned in the system design section, the whole system consists of many finite state machines and there are 4 main finite state machines (corresponding to the 4 modes). First of all, the automatic mode of the system, which is the most important module, is examined.

```
1 int t_road_1 = 0, t_road_2 = 0;
2 void fsm_automatic() {
3     switch (status) {
4         case INIT:
5             status = MODE1;
6             break;
7         case MODE1:
8             led_config();
9             t_road_1 = T_RED;
10            t_road_2 = T_GREEN;
11            HAL_UART_Transmit(&huart2, "!____MODE1____#\n\r", 20, 50);
12            disp_time_uart(t_road_1, t_road_2);
13            setTimer4(1000);
14            buzzer_period=400;
15            volume = 15;
16            status = RED_GREEN;
17            break;
18        case RED_GREEN:
19            led_config();
20            if (timer4_flag == 1) {
21                t_road_1--;
22                t_road_2--;
23                T_CHECK = t_road_2;
24                if (t_road_2 <= 0) { // next state pre-setup
25                    t_road_2 = T_AMBER;
26                    buzzer_period = 75;
27                    volume = 100;
28                    status = RED_AMBER; // Change state
29                }
30                if (T_CHECK <= T_GREEN/2 && T_CHECK > 0) { // next state pre-setup
31                    buzzer_period = 200;
32                    volume = 50;
33                }
34                disp_time_uart(t_road_1, t_road_2);
35                setTimer4(1000);
36            }
37            break;
38        case RED_AMBER:
39            led_config();
40            if (timer4_flag == 1) {
41                t_road_1--;
42                t_road_2--;
43                if (t_road_1 <= 0) { // next state pre-setup
44                    t_road_1 = T_GREEN;
45                    t_road_2 = T_RED;
46                    status = GREEN_RED; // Change state
47                    curr_ped_status=PED_OFF;
```

```

48         }
49         disp_time_uart(t_road_1, t_road_2);
50         setTimer4(1000);
51     }
52     break;
53     case GREEN_RED:
54         led_config();
55         if (timer4_flag == 1) {
56             t_road_1--;
57             t_road_2--;
58             if (t_road_1 <= 0) { // next state pre-setup
59                 t_road_1 = T_AMBER;
60                 status = AMBER_RED; // Change state
61             }
62             disp_time_uart(t_road_1, t_road_2);
63             setTimer4(1000);
64         }
65         break;
66     case AMBER_RED:
67         led_config();
68         if (timer4_flag == 1) {
69             t_road_1--;
70             t_road_2--;
71             if (t_road_1 <= 0) { // next state pre-setup
72                 t_road_1 = T_RED;
73                 t_road_2 = T_GREEN;
74                 buzzer_period=500;
75                 volume = 20;
76                 status = RED_GREEN; // Change state
77             }
78             disp_time_uart(t_road_1, t_road_2);
79             setTimer4(1000);
80         }
81         break;
82     default:
83         break;
84 }
85 }

```

Listing 1: Finite state machine for automatic mode

In the above FSM, *void fsm_automatic()*, there are 6 states, 2 states for initialization and 4 states for the automatic behaviour. The trackers for state changing in this machine are the *t_road_1* and *t_road_2* variables, which is similar to the real system (when the remaining time of the light reaches 0, the color is changed).

- **INIT:** This is the initialization state for the whole system. However, there is no initialization needed for the whole system in this project's scope so this state is left blank and it does not do anything. For future development reason, it is still kept here in the structure of the system.
- **MODE1:** This is the initialization state for **Mode 1**. As can be seen in the code, a few necessary conditions are set up in order for this finite state machine to work correctly:
 - All LEDs are cleared
 - Appropriate time duration values are assigned to *t_road_1* and *t_road_2*
 - A software timer is set with the period of 1 second for counting down and ensuring real-time operation of the system.

- The status is assigned to RED_GREEN so that the FSM can start operating
- There are also other setups related to the buzzer and UART communication; however, they will be discussed in the next few sections.
- **RED_GREEN, RED_AMBER, GREEN_RED, AMBER_RED:** These states are responsible for the automatic behaviour of the traffic light. Their structures are quite similar, the differences lie in the assigned values and the setups which have to be different depending on the current state. In general, each state is implemented as follows:
 - LEDs are configured to match the current state with the *led_config()* function.
 - An if statement to keep track of the timer flag, responsible for counting down *t_road_1* and *t_road_2* variables.
 - A nested if statement to check if the remaining time has reached 0. If the remaining time has reached 0, necessary pre-setups are carried out (*t_road_1* and *t_road_2* are assigned new values); finally the FSM jumps to next state by reassigning the *status* variable.

5.2 FSM for manual modes

These 3 finite state machine hold the mission of handling the manual modes. The structure of these finite state machines are quite the same, just different setups and values:

- **MODE2, MODE3, MODE4:** these are the initialization states of these FSMs and necessary setups are carried out, LEDs are cleared and timer is set to prepare for the blinking behaviour of some LEDs.
- **AUTO_X and ADJ_X:** These states are responsible for blinking the LEDs. The time duration adjustment from the user is handled in the FSM for button processing, not in these FSMs.

```
1 void fsm_red_manual() {
2     switch(status) {
3         case MODE2:
4             led_config();
5             setTimer1(500);
6             HAL_UART_Transmit(&huart2, "!____MODE2____#\n\r", 20, 50);
7             disp_t_red_uart();
8             status = AUTO_RED;
9             break;
10        case AUTO_RED:
11            if (timer1_flag == 1) {
12                toggle_red();
13                setTimer1(500);
14            }
15            break;
16        case ADJ_RED:
17            if (timer1_flag == 1) {
18                toggle_red();
19                setTimer1(500);
20            }
21            break;
22        default:
23            break;
24    }
25 }
```

```
27 void fsm_amber_manual() {
28     switch(status) {
29         case MODE3:
30             led_config();
31             setTimer1(500);
32             HAL_UART_Transmit(&huart2, "!____MODE3____#\n\r", 20, 50);
33             disp_t_amber_uart();
34             status = AUTO_AMBER;
35             break;
36         case AUTO_AMBER:
37             if (timer1_flag == 1) {
38                 toggle_amber();
39                 setTimer1(500);
40             }
41             break;
42         case ADJ_AMBER:
43             if (timer1_flag == 1) {
44                 toggle_amber();
45                 setTimer1(500);
46             }
47             break;
48         default:
49             break;
50     }
51 }
52
53 void fsm_green_manual() {
54     switch(status) {
55         case MODE4:
56             led_config();
57             setTimer1(500);
58             HAL_UART_Transmit(&huart2, "!____MODE4____#\n\r", 20, 50);
59             disp_t_green_uart();
60             status = AUTO_GREEN;
61             break;
62         case AUTO_GREEN:
63             if (timer1_flag == 1) {
64                 toggle_green();
65                 setTimer1(500);
66             }
67             break;
68         case ADJ_GREEN:
69             if (timer1_flag == 1) {
70                 toggle_green();
71                 setTimer1(500);
72             }
73             break;
74         default:
75             break;
76     }
77 }
```

Listing 2: FSMs for manual modes

5.3 FSM for button processing

This FSM is responsible for processing the input from the user. Button 0 is for the pedestrian feature, Button 1 is for mode changing, Button 2 is for increasing the time periods of the traffic light, Button 3 is for saving the time period and recalculating other time periods to match the correct behaviour of traffic lights ($T_{RED} = T_{AMBER} + T_{GREEN}$).

```
1 enum ButtonState{BUTTON_RELEASED, BUTTON_PRESSED, BUTTON_PRESSED_MORE_THAN_1_SECOND};
2 enum ButtonState button_0_state = BUTTON_PRESSED;
3 enum ButtonState button_1_state = BUTTON_PRESSED;
4 enum ButtonState button_2_state = BUTTON_PRESSED;
5 enum ButtonState button_3_state = BUTTON_PRESSED;
6
7 void fsm_button_processing() {
8     switch (button_0_state) {
9         case BUTTON_RELEASED:
10             if (is_button_pressed(0)) {
11                 button_0_state = BUTTON_PRESSED;
12                 if (curr_ped_status==PED_OFF&&(status==MODE1||status==RED_GREEN||status==
13                 RED_AMBER||status==GREEN_RED||status==AMBER_RED)){
14                     curr_ped_status=PED_ON;
15                 }
16             }
17             break;
18         case BUTTON_PRESSED:
19             if (!is_button_pressed(0)) {
20                 button_0_state = BUTTON_RELEASED;
21             } else {
22                 if (is_button_pressed_1s(0)) {
23                     button_0_state = BUTTON_PRESSED_MORE_THAN_1_SECOND;
24                 }
25             }
26             break;
27         case BUTTON_PRESSED_MORE_THAN_1_SECOND:
28             if (!is_button_pressed(0)) {
29                 button_0_state = BUTTON_RELEASED;
30             }
31             // do nothing, wait for the button to be released
32             break;
33     }
34     switch (button_1_state) {
35         case BUTTON_RELEASED:
36             if (is_button_pressed(1)) {
37                 button_1_state = BUTTON_PRESSED;
38                 if (status == RED_GREEN) status = MODE2;
39                 if (status == AUTO_RED) status = MODE3;
40                 if (status == AUTO_AMBER) status = MODE4;
41                 if (status == AUTO_GREEN) status = MODE1;
42             }
43             break;
44         case BUTTON_PRESSED:
45             if (!is_button_pressed(1)) {
46                 button_1_state = BUTTON_RELEASED;
47             } else {
48                 if (is_button_pressed_1s(1)) {
49                     button_1_state = BUTTON_PRESSED_MORE_THAN_1_SECOND;
50                 }
51             }
52             break;
```

```

52     case BUTTON_PRESSED_MORE_THAN_1_SECOND:
53         if (!is_button_pressed(1)) {
54             button_1_state = BUTTON_RELEASED;
55         }
56         // do nothing, wait for the button to be released
57         break;
58     }
59
60     switch (button_2_state) {
61         case BUTTON_RELEASED:
62             if (is_button_pressed(2)) {
63                 button_2_state = BUTTON_PRESSED;
64                 if (status == AUTO_RED || status == ADJ_RED) {
65                     status = ADJ_RED;
66                     T_RED++;
67                     if (T_RED >= 99) T_RED = 1;
68                     disp_t_red_uart();
69                 }
70                 if (status == AUTO_AMBER || status == ADJ_AMBER) {
71                     status = ADJ_AMBER;
72                     T_AMBER++;
73                     if (T_AMBER >= 5) T_AMBER = 1;
74                     disp_t_amber_uart();
75                 }
76                 if (status == AUTO_GREEN || status == ADJ_GREEN) {
77                     status = ADJ_GREEN;
78                     T_GREEN++;
79                     if (T_GREEN >= 99) T_GREEN = 1;
80                     disp_t_green_uart();
81                 }
82             }
83             break;
84         case BUTTON_PRESSED:
85             if (!is_button_pressed(2)) {
86                 button_2_state = BUTTON_RELEASED;
87             } else {
88                 if (is_button_pressed_1s(2)) {
89                     button_2_state = BUTTON_PRESSED_MORE_THAN_1_SECOND;
90                 }
91             }
92             break;
93         case BUTTON_PRESSED_MORE_THAN_1_SECOND:
94             if (!is_button_pressed(2)) {
95                 button_2_state = BUTTON_RELEASED;
96             }
97             // do nothing, wait for the button to be released
98             break;
99     }
100
101     switch (button_3_state) {
102         case BUTTON_RELEASED:
103             if (is_button_pressed(3)) {
104                 button_3_state = BUTTON_PRESSED;
105                 if (status == ADJ_RED) {
106                     status = AUTO_RED;
107                     if (T_RED <= T_AMBER) T_RED = T_AMBER + 1;
108                     T_GREEN = T_RED - T_AMBER;
109                 }
110             }

```

```

111         if (status == ADJ_AMBER) {
112             status = AUTO_AMBER;
113             if (T_RED <= T_AMBER) T_RED = T_AMBER + 1;
114             T_GREEN = T_RED - T_AMBER;
115         }
116         if (status == ADJ_GREEN) {
117             status = AUTO_GREEN;
118             if (T_AMBER + T_GREEN >= 99) {
119                 T_AMBER = 4;
120                 T_GREEN = 95;
121             }
122             T_RED = T_AMBER + T_GREEN;
123         }
124         HAL_UART_Transmit(&huart2, "!SAVED#\n\r", 11, 50);
125         HAL_UART_Transmit(&huart2, "\n\r", 4, 50);
126     }
127     break;
128     case BUTTON_PRESSED:
129         if (!is_button_pressed(3)) {
130             button_3_state = BUTTON_RELEASED;
131         } else {
132             if (is_button_pressed_1s(3)) {
133                 button_3_state = BUTTON_PRESSED_MORE_THAN_1_SECOND;
134             }
135         }
136         break;
137     case BUTTON_PRESSED_MORE_THAN_1_SECOND:
138         if (!is_button_pressed(3)) {
139             button_3_state = BUTTON_RELEASED;
140         }
141         // do nothing, wait for the button to be released
142         break;
143     }
144 }

```

Listing 3: FSM for button processing

5.4 Button debounce

This is how the button debouncing is handled based on the button debouncing theory discussed in the Theory Background section.

```

1 // we aim to work with 3 buttons
2 #define NO_OF_BUTTONS          4
3
4 // timer interrupt duration is 10ms, so to pass 1 second,
5 // we need to jump to the interrupt service routine 100 time
6 #define DURATION_FOR_AUTO_INCREASING    100
7 #define BUTTON_IS_PRESSED                GPIO_PIN_RESET
8 #define BUTTON_IS_RELEASED              GPIO_PIN_SET
9 #define BUTTON_1 GPIO
10
11 // the buffer that the final result is stored after debouncing
12 static GPIO_PinState buttonBuffer[NO_OF_BUTTONS];
13
14 // we define two buffers for debouncing
15 static GPIO_PinState debounceButtonBuffer1[NO_OF_BUTTONS];
16 static GPIO_PinState debounceButtonBuffer2[NO_OF_BUTTONS];
17

```

```
18 // we define a flag for a button pressed more than 1 second.
19 static uint8_t flagForButtonPress1s[N0_OF_BUTTONS];
20
21 // we define counter for automatically increasing the value
22 // after the button is pressed more than 1 second.
23 static uint16_t counterForButtonPress1s[N0_OF_BUTTONS];
24
25 static GPIO_TypeDef* button_ports[N0_OF_BUTTONS] = {A0_PedButton_GPIO_Port,
26     A1_Button1_GPIO_Port, A2_Button2_GPIO_Port, A3_Button3_GPIO_Port};
27
28 static uint16_t button_pins[N0_OF_BUTTONS] = {A0_PedButton_Pin, A1_Button1_Pin,
29     A2_Button2_Pin, A3_Button3_Pin};
30
31 void button_reading(void) {
32     for (int i = 0; i < N0_OF_BUTTONS; i++) {
33         debounceButtonBuffer2[i] = debounceButtonBuffer1[i];
34         debounceButtonBuffer1[i] = HAL_GPIO_ReadPin(button_ports[i], button_pins[i]);
35
36         if (debounceButtonBuffer1[i] == debounceButtonBuffer2[i])
37             buttonBuffer[i] = debounceButtonBuffer1[i];
38
39         if (buttonBuffer[i] == BUTTON_IS_PRESSED) {
40             //if a button is pressed, we start counting
41             if (counterForButtonPress1s[i] < DURATION_FOR_AUTO_INCREASING) {
42                 counterForButtonPress1s[i]++;
43             } else {
44                 //the flag is turned on when 1 second has passed
45                 //since the button is pressed.
46                 flagForButtonPress1s[i] = 1;
47             }
48         } else {
49             counterForButtonPress1s[i] = 0;
50             flagForButtonPress1s[i] = 0;
51         }
52     }
53 }
54
55 unsigned char is_button_pressed(uint8_t index){
56     if (index >= N0_OF_BUTTONS) return 0;
57     return (buttonBuffer[index] == BUTTON_IS_PRESSED);
58 }
59
60 unsigned char is_button_pressed_1s(unsigned char index){
61     if (index >= N0_OF_BUTTONS) return 0;
62     return (flagForButtonPress1s[index] == 1);
63 }
```

Listing 4: Button Debouncing

5.5 Pedestrian Light

5.5.1 On and Off Pedestrian Light functions

```
1 void off_pedestrian(){
2     HAL_GPIO_WritePin (D6_PedLED_GPIO_Port , D6_PedLED_Pin , 0);
3     HAL_GPIO_WritePin (D7_PedLED_GPIO_Port , D7_PedLED_Pin , 0);
4 }
5 void on_red_pedestrian(){
6     HAL_GPIO_WritePin (D6_PedLED_GPIO_Port , D6_PedLED_Pin , 1);
7     HAL_GPIO_WritePin (D7_PedLED_GPIO_Port , D7_PedLED_Pin , 0);
8 }
9 void on_green_pedestrian(){
10    HAL_GPIO_WritePin (D6_PedLED_GPIO_Port , D6_PedLED_Pin , 0);
11    HAL_GPIO_WritePin (D7_PedLED_GPIO_Port , D7_PedLED_Pin , 1);
12 }
```

Listing 5: On and Off Pedestrian Light functions

These are helper functions that can be used to either turn off or turn on color red or green of the pedestrian light.

5.5.2 Pedestrian Light Configuration

```
1 void pedestrian_led_config(){
2     switch (status) {
3         case MODE1:
4             off_pedestrian();
5             break;
6         case RED_GREEN:
7             if(curr_ped_status==PED_ON){
8                 on_green_pedestrian();
9                 break;
10            }
11            off_pedestrian();
12            break;
13         case RED_AMBER:
14             if(curr_ped_status==PED_ON){
15                 on_green_pedestrian();
16                 break;
17            }
18            off_pedestrian();
19            break;
20         case GREEN_RED:
21             if(curr_ped_status==PED_ON){
22                 on_red_pedestrian();
23                 break;
24            }
25            off_pedestrian();
26            break;
27         case AMBER_RED:
28             if(curr_ped_status==PED_ON){
29                 on_red_pedestrian();
30                 break;
31            }
32            off_pedestrian();
33            break;
34         case MODE2:
35             curr_ped_status=PED_OFF;
```

```
36     off_pedestrian();  
37     break;  
38     case MODE3:  
39         curr_ped_status=PED_OFF;  
40         off_pedestrian();  
41         break;  
42     case MODE4:  
43         curr_ped_status=PED_OFF;  
44         off_pedestrian();  
45         break;  
46     default:  
47         break;  
48 }  
49 }
```

Listing 6: Pedestrian Light Configuration

This part of code will check the current state of the FSM for automatic mode to set the suitable pedestrian light behavior. Here's an explanation of each case within the switch statement:

- **MODE1**
 - Turns off pedestrian LEDs by calling `off_pedestrian()`.
- **RED_GREEN, RED_AMBER, GREEN_RED, AMBER_RED:**
 - Checks if the current pedestrian status (`curr_ped_status`) is set to `PED_ON`.
 - If true, it calls either `on_green_pedestrian()` or `on_red_pedestrian()`.
 - If false, it turns off pedestrian LEDs by calling `off_pedestrian()`.
- **MODE2, MODE3, MODE4**
 - Sets the current pedestrian status (`curr_ped_status`) to `PED_OFF`.
 - Turns off pedestrian LEDs by calling `off_pedestrian()`.

5.6 Buzzer

```
1 void buzzer(int vol){  
2     __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, vol);  
3 }
```

Listing 7: Buzzer volume set function

We use this function to set volume for buzzer according to the parameter *vol*
vol ranges from 0 to 100.

```
1 void set_buzzer(){  
2     if (timer6_flag == 1){  
3         setTimer6(buzzer_period);  
4         if (buzzer_flag == 0){  
5             buzzer(volume);  
6             buzzer_flag = 1;  
7         }else {  
8             buzzer(0);  
9             buzzer_flag = 0;  
10        }  
11    }  
12 }
```

Listing 8: Buzzer volume & frequency adjusting function

We use above function to set volume & frequency for pedestrian buzzer:

- volume ranges from 0 - 100
- buzzer__period: 3 level of frequency

```
1 void pedestrian_buzzer_config(){
2     switch (status) {
3         case MODE1:
4             buzzer(0);
5             break;
6         case RED_GREEN:
7             if(curr_ped_status==PED_ON){
8                 set_buzzer();
9                 break;
10            }
11            buzzer(0);
12            break;
13        case RED_AMBER:
14            if(curr_ped_status==PED_ON){
15                set_buzzer();
16                break;
17            }
18            buzzer(0);
19            break;
20        case GREEN_RED:
21            if(curr_ped_status==PED_ON){
22                buzzer(0);
23                break;
24            }
25            buzzer(0);
26            break;
27        case AMBER_RED:
28            if(curr_ped_status==PED_ON){
29                buzzer(0);
30                break;
31            }
32            buzzer(0);
33            break;
34        case MODE2:
35            curr_ped_status=PED_OFF;
36            buzzer(0);
37            break;
38        case MODE3:
39            curr_ped_status=PED_OFF;
40            buzzer(0);
41            break;
42        case MODE4:
43            curr_ped_status=PED_OFF;
44            buzzer(0);
45            break;
46        default:
47            break;
48    }
49 }
```

Listing 9: Pedestrian Buzzer Configuration

This part of code will check the current state of the FSM for automatic mode to set the suitable pedestrian buzzer behavior. Here's an explanation of each case within the switch statement:

- **MODE1**

- Turns off pedestrian buzzer by calling **buzzer(0)**.

- **RED_GREEN, RED_AMBER, GREEN_RED, AMBER_RED:**

- Checks if the current pedestrian status (**curr_ped_status**) is set to **PED_ON**.
- If true, it calls **set_buzzer()**.
- If false, it turns off pedestrian buzzer by calling **buzzer(0)**.

- **MODE2, MODE3, MODE4**

- Sets the current pedestrian status (**curr_ped_status**) to **PED_OFF**.
- Turns off pedestrian buzzer by calling **buzzer(0)**.

```
1  case MODE1:
2      ...
3      buzzer_period=300;
4      volume = 5;
5      status = RED_GREEN; // Change state
6      break;
7  case RED_GREEN:
8      ...
9      if (t_road_2 <= 0) {
10         ...
11         buzzer_period = 50;
12         volume = 100;
13         status = RED_AMBER; // Change state
14     }
15     if (T_CHECK <= T_GREEN/2 && T_CHECK > 0) {
16         buzzer_period = 150;
17         volume = 15;
18     }
19     ...
20 }
21 break;
22 ...
23 case AMBER_RED:
24     ...
25     if (t_road_1 <= 0) {
26         ...
27         buzzer_period=300;
28         volume = 5;
29         status = RED_GREEN; // Change state
30     }
31     ...
32 }
33 break;
```

Listing 10: Buzzer volume & frequency adjusting function

Finally, we set the *volume* & *buzzer_period* in the *fsm_automatic.c* following the logic:

- **MODE1:** In this mode, because the next state is *RED_GREEN*, it sets *volume* to 5 and *buzzer_period* to 300ms, which is the first level (low & slow).
Moving to the next state: *RED_GREEN*

- **RED_GREEN:**

In this mode, it divide the state into 2 period of time using *T_CHECK* parameter.

In the first half, it sets *volume* to 15 and *buzzer_period* to 150ms, which is the middle level.

In the second half, it sets *volume* to 100 and *buzzer_period* to 50ms, which is the last level (fast & loud). Then moving to the next state *RED_AMBER*

- **AMBER_RED:** In this state, because it will go back to *RED_GREEN* state, it sets *volume* to 5 and *buzzer_period* to 300ms, which is the first level (low & slow).

The 2-parameter (*volume* & *buzzer_period*) can be adjusted easily if needed.

5.7 UART communication

In this project, as mentioned in the system design section, UART is used for displaying the current status of the system, here is how the UART APIs are implemented:

```
1 void disp_t_red_uart() {
2     HAL_UART_Transmit(&huart2, "!T_RED = ", 11, 50);
3     char buffer[16];
4     HAL_UART_Transmit(&huart2, (uint8_t*)buffer, sprintf(buffer, "%d\n\r", T_RED), 50);
5 }
6
7 void disp_t_amber_uart() {
8     HAL_UART_Transmit(&huart2, "!T_AMBER = ", 11, 50);
9     char buffer[16];
10    HAL_UART_Transmit(&huart2, (uint8_t*)buffer, sprintf(buffer, "%d\n\r", T_AMBER), 50);
11 }
12
13 void disp_t_green_uart() {
14     HAL_UART_Transmit(&huart2, "!T_GREEN = ", 11, 50);
15     char buffer[16];
16     HAL_UART_Transmit(&huart2, (uint8_t*)buffer, sprintf(buffer, "%d\n\r", T_GREEN), 50);
17 }
18
19 void disp_time_uart(int t_road_1, int t_road_2) {
20     char buffer[16];
21     HAL_UART_Transmit(&huart2, "!T_ROAD_1= ", 12, 50);
22     HAL_UART_Transmit(&huart2, (uint8_t*)buffer, sprintf(buffer, "%d\n\r", t_road_1), 50);
23     HAL_UART_Transmit(&huart2, "!T_ROAD_2= ", 12, 50);
24     HAL_UART_Transmit(&huart2, (uint8_t*)buffer, sprintf(buffer, "%d\n\r", t_road_2), 50);
25     HAL_UART_Transmit(&huart2, "\n", 2, 50);
26 }
```

Listing 11: uart.c

- **disp_t_red_uart(), disp_t_amber_uart(), disp_t_green_uart():** these functions are used for displaying the values of *T_RED*, *T_AMBER*, *T_GREEN* variables. These functions are called in the manual modes, to let the user know the value that they are adjusting.
- **disp_time_uart():** this function is used for displaying the remaining time of the traffic light, it is called in the automatic mode.

5.8 Cooperative scheduler

```
1 SCH_Add_Task(timer_run, 0, 1);  
2 SCH_Add_Task(button_reading, 0, 1);  
3 SCH_Add_Task(fsm_automatic, 0, 1);  
4 SCH_Add_Task(fsm_red_manual, 0, 1);  
5 SCH_Add_Task(fsm_amber_manual, 0, 1);  
6 SCH_Add_Task(fsm_green_manual, 0, 1);  
7 SCH_Add_Task(fsm_button_processing, 0, 1);
```

Listing 12: Cooperative scheduler

Every task corresponds to a function that will be executed for every 10ms by the scheduler, with 0 initial delay .

6 Conclusion

6.1 System evaluation

Based on the real-life demonstration and the test cases we have used for testing the system, the system is able to operate correctly, following the initial system design without any error.

In terms of real-time operation, the system successfully runs with real-time. Therefore, this system can be a potential candidate for the practical use in the real-life traffic.

6.2 Future system development

In the future, this system can be deployed and used in real-life traffic, especially in Vietnam where the pedestrian feature is not widely integrated. With the pedestrian feature, it is potential that this system will improve the safety of the pedestrians greatly, avoiding unwanted accidents due to improper road crossing.

Moreover, thanks to the finite state machine design and modular project organizing, new features can be built and implemented with ease. For example, in the future, when self-driving cars are used widely, the traffic light may have a feature that it can send signals to the autopilot cars to stop (or slow down) when the pedestrians need to cross the road.