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

MICROPROCESSORS - MICROCONTROLLERS (CO3009)

Project

"Traffic Light using STM32F103RB

CLASS: CC01

Lecturer: Lê Trọng Nhân

Students: Bùi Phát Lộc - 1752326
Nguyễn Lê Anh Tuấn - 1852837
Lê Bá Thành - 1852739

HO CHI MINH CITY, December 2023

Contents

Member list and contributions

No.	Fullname	Student ID	Workload	Contribution
1	Bùi Phát Lộc	1752326	- Buttons - Software timers - UART	100%
2	Nguyễn Lê Anh Tuấn	1852837	- Automatic mode - Tuning mode - Pedestrian scramble - Buzzer	100%
3	Lê Bá Thành	1852739	- Scheduler - Manual mode - Writing report	100%

1 Github

The source code for the project is available at: Github link

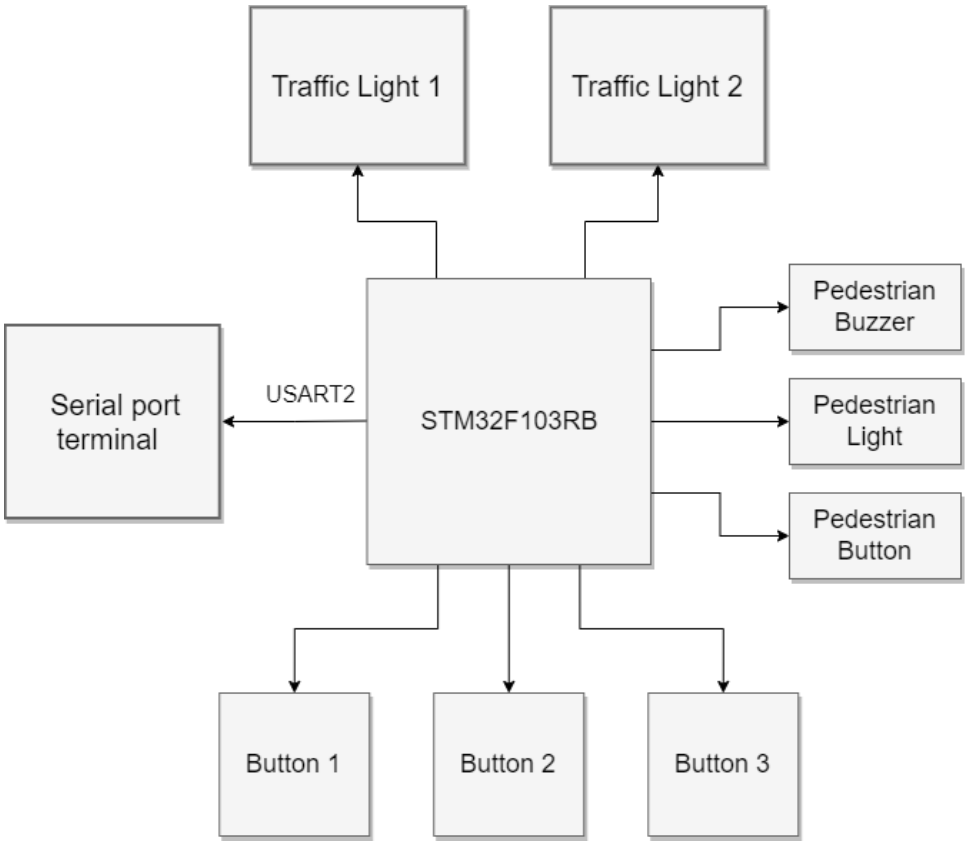
2 Introduction

Traffic lights, the common signals governing intersections, form the backbone of organized urban transportation. These crucial systems rely on a clever mix of electronics and precise timing to ensure the smooth flow of vehicles and pedestrians. The STM32F103RB microcontroller takes center stage in this project, as it main purpose is to replicate a 2-way traffic light system, introducing features that go beyond the conventional red, yellow, and green signals. In this assignment, we will design a traffic light system using the STM32F103RB microcontroller. Going beyond the basics, the project introduces various modes of operation:

- **Automatic mode:** This mode mirrors the standard traffic light sequence, smoothly transitioning between red, yellow, and green lights.
- **Manual mode:** This mode adds a layer of interactivity, allowing users to switch between light colors using a button. This may be used to manually change the flow of traffics in rush hour.
- **Tuning mode:** This mode provide the users the ability to adjust the timing length of each light, allowing for customization based on specific requirements.
- **Pedestrian scramble:** When activated by pressing a dedicated button, this feature synchronizes the pedestrian light inversely to the vehicular lights, contributing to overall traffic efficiency and safety.

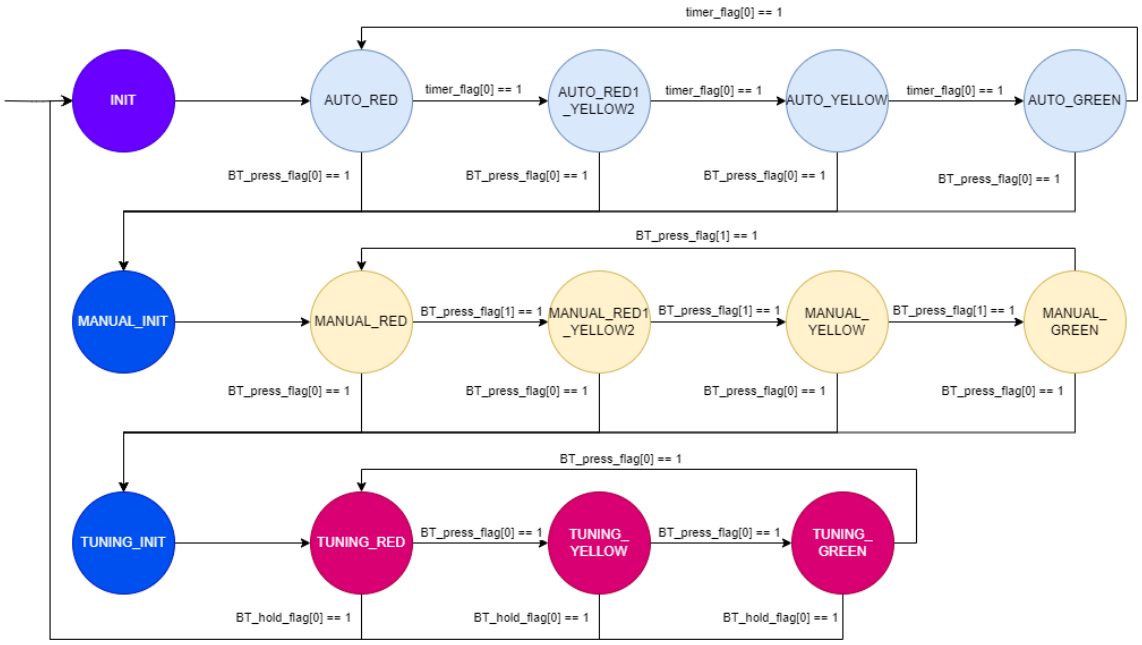
As we dive deeper into the implementation of the traffic lights, we will showcase the potential of utilizing the STM32F103RB microcontroller in enhancing everyday systems. This report will detail the technical aspects of the project, shedding light on the practicality embedded in the design of an intelligent traffic management system.

3 Block diagram



Hình 1: *Block diagram of the system*

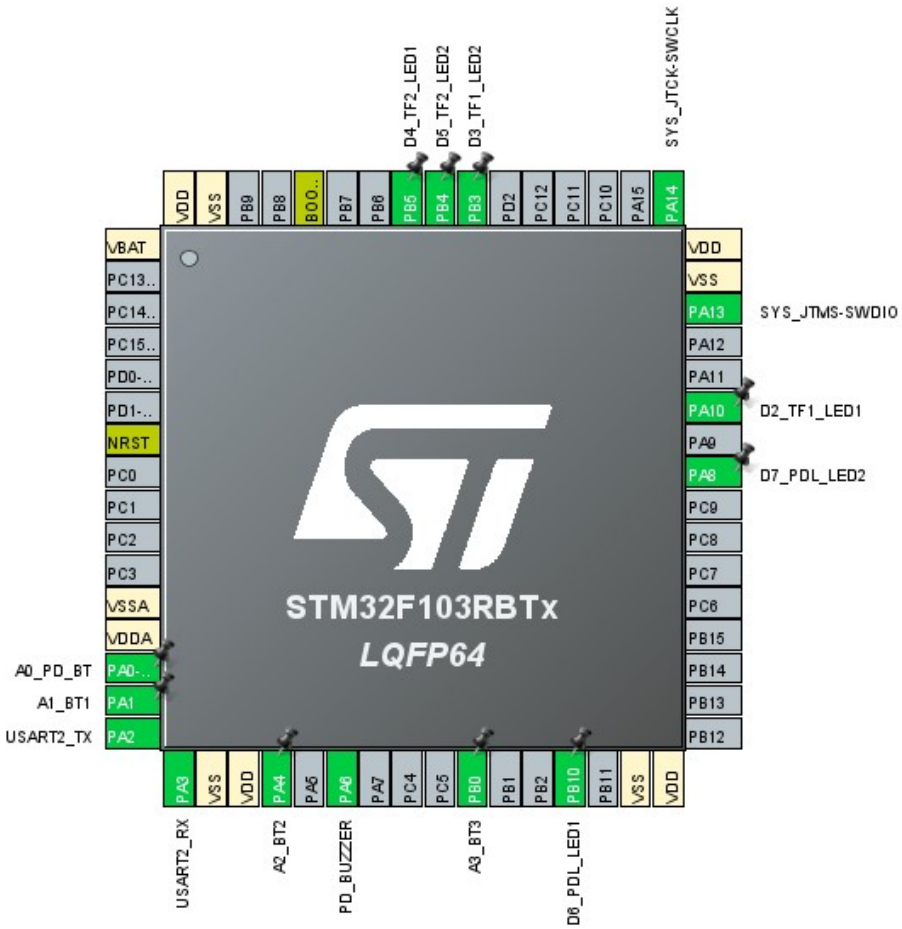
4 Finite state machine



Hình 2: *Finite state machine of the system*

5 Configuration

5.1 Pin configuration



Hình 3: *Pinout view*

- SYS: Debug is Serial Wire
- GPIO:
  1. PA10, PB3: LEDs on the main traffic light
  2. PB4, PB5: LEDs on the secondary traffic light
  3. PB10, PA8: LED on the pedestrian light
  4. PA1, PA4, PB0, PA0-WKUP: Button 1, 2, 3 and pedestrian, respectively

Pin Name	Signal on Pin	GPIO output le...	GPIO mode	GPIO Pull-up/...	Maximum outp...	User Label
PA0-WKUP	n/a	n/a	Input mode	Pull-up	n/a	A0_PD_BT
PA1	n/a	n/a	Input mode	Pull-up	n/a	A1_BT1
PA4	n/a	n/a	Input mode	Pull-up	n/a	A2_BT2
PA8	n/a	Low	Output Push ...	No pull-up and...	Low	D7_PDL_LED2
PA10	n/a	Low	Output Push ...	No pull-up and...	Low	D2_TF1_LED1
PB0	n/a	n/a	Input mode	Pull-up	n/a	A3_BT3
PB3	n/a	Low	Output Push ...	No pull-up and...	Low	D3_TF1_LED2
PB4	n/a	Low	Output Push ...	No pull-up and...	Low	D5_TF2_LED2
PB5	n/a	Low	Output Push ...	No pull-up and...	Low	D4_TF2_LED1
PB10	n/a	Low	Output Push ...	No pull-up and...	Low	D6_PDL_LED1

Hình 4: *GPIO configuration*

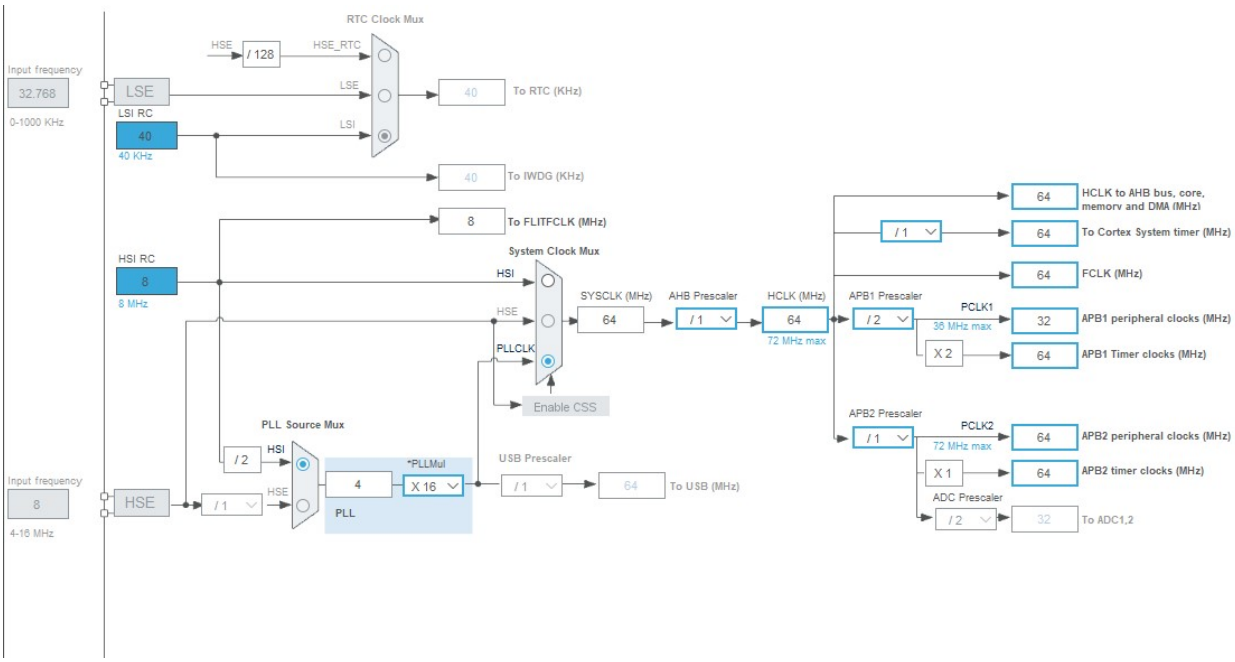
- Timers:
  1. Timer2:
     - Clock source: Internal clock
     - Prescaler: 7999
     - Counter period: 79

     $f = \frac{F}{(PSC+1)(ARR+1)} = \frac{64000000}{(7999+1)(79+1)} = 100$ Hz
  2. Timer3:
     - Channel1: PWM Generation CH1
     - Prescaler: 63
     - Counter period: 999

     $f = \frac{F}{(PSC+1)(ARR+1)} = \frac{64000000}{(63+1)(999+1)} = 1000$ Hz
- USART: USART2
  - Mode: Asynchronous
  - Baud rate: 115200 Bits/s
  - Word Length: 8 Bits
  - Parity: None
  - Stop Bits: 1

5.2 Clock configuration

The system clock is 64 MHz



Hình 5: *Clock configuration*

6 Functions

Here are the tables of functions used in the project

- global.c:

Return type	Function name	Parameters	Description
void	controlTrafficLights	int color1 int color2	Controls the main traffic lights based on two color inputs
void	controlPedLights	int color	Controls the pedestrian lights based on a single color input
void	clearTrafficLights	void	Turns off all traffic light LEDs (both main and secondary)
void	toggleRedLED	void	Toggles the red LEDs for both main and secondary traffic lights.
void	toggleGreenLED	void	Toggles the green LEDs for both main and secondary traffic lights
void	toggleYellowLED	void	Toggles all yellow LEDs for both main and secondary traffic lights

- button.c:

Return type	Function name	Parameters	Description
void	getKeyInput	void	Debouncing buttons and detecting buttons press/hold
int	isBTPressed	int index	Checks if a button was pressed
int	isBTHold	int index	Checks if a button was held

- software_timer.c:

Return type	Function name	Parameters	Description
void	setTimer	int duration, int id	Sets a timer's duration and resets its flag
void	timerRun	void	Checks all timers, decrements active ones, and raises their flag upon reaching zero
void	resetAllTimers	void	Set all timers flags and duration to 0

- scheduler.c:

Return type	Function name	Parameters	Description
void	SCH_Init	int duration, int id	Initializes all tasks
void	SCH_Update	void	Checks the first task, decrease the delay or run the task when delay reach 0
uint32_t	SCH_Add_Task	void (*p_function)(), uint32_t DELAY, uint32_t PERIOD	Inserts a new task into the scheduler based on its delay and available slots
void	SCH_Dispatch _Tasks	void	Checks the first task, decrease the delay or run the task when
uint8_t	SCH_Delete_Task	uint32_t TASK_ID	Remove the task with the ID
uing32_t	Get_New_Task_ID	void	Get a unique task ID

- buzzer.c:

Return type	Function name	Parameters	Description
void	buzzer_init	TIM_TypeDef *tim	Initiate the buzzer
void	buzzer_sound	TIM_HandleTypeDef htim, int volume	Control the buzzer frequency and volume
void	buzzer_run	void	Run the buzzer
void	buzzer_set	int value	Set buzzer state

- fsm_automatic.c

Return type	Function name	Parameters	Description
void	fsm_automatic_run	void	FSM for automatic mode

- fsm_manual.c

Return type	Function name	Parameters	Description
void	fsm_manual_run	void	FSM for manual mode

- fsm_tuning.c

Return type	Function name	Parameters	Description
void	fsm_tuning_run	void	FSM for tuning mode
void	ledBalance	void	Balance out the value of each light

7 Detail descriptions of some crucial functions

7.1 fsm_automatic_run

This function is a FSM responsible for the automatic mode. It starts at INIT state and will cycle through 4 state: AUTO_RED, AUTO_RED1_YELLOW2, AUTO_GREEN, AUTO_YELLOW.

- INIT: Initialize necessary variables like counters and timers, transmit a message through UART to indicate the mode: "!AUTOMATIC#".
  The next state is AUTO_RED.
- AUTO_RED: Turn red light on the main lights and green light on the secondary lights. Check for buttons and flags state:
  - timer_flag[0]: This is the timer with the interval of 1s, this is used to decrement the counter for the light every second. If the timer_flag[0] is 1 and the respective counter for the light is 0, we will transition to the next state. The counter for the 7 segment will also be transmit through the UART in the form of: "!7SEG:counter1||counter2#"
  - timer_flag[1]: This is the timer with the interval of 8s, this works as a timeout for the pedestrian light, the flag will be reset every time the pedestrian button is pressed, therefore if the button is not pressed for 8s, it will turn off the pedestrian light.
  - isBTPressed(0): This checks whether button 1 is pressed, this button is responsible for changing the mode. It will also turn off the pedestrian light in case it is still on while someone has pressed this button. The next mode is manual mode.
  - isBTPressed(3): This checks whether the pedestrian button is pressed, if it is then it will turn on the respective light for the pedestrian light, in this case, it is green as the main light is currently red. It will also set the timeout timer for the pedestrian light and transmit a message through UART in the form of: "!PEDESTRIAN_GREEN#". The buzzer will also be turned on.
  - isBTHold(3): This checks whether the pedestrian button is being hold, if it is then it works the same way as the pedestrian button being repeatedly pressed.

  The next state is AUTO_RED1_YELLOW2.
- AUTO_RED1_YELLOW2: This is a transition state for the secondary lights, the first lights will still be red while the secondary lights will change from green to yellow. Buttons and timers flags are still being checked the same way as the previous state, however, there are some differences:
  - timer_flag[0]: In addition to the previous behaviours, it will now turn off the buzzer if the counter of the respective light reach 0, as the main light will turn green and the pedestrian light will turn red.
  - timer_flag[5]: This is an additional timer for the pedestrian light, this will flash the yellow light. The time interval for this timer is 0.25s
  - isBTPressed(3): It will now set the buzzer state to a state that indicating the yellow light of the pedestrian light is flashing, which means it will sound louder and faster.

  The next state is AUTO_GREEN.

- AUTO_GREEN: Works the same way as the previous state, the color is green for the main lights and red for the secondary lights. Buttons and timers still works the same way as AUTO_RED state, but with the value of the respective color.
  The next state is AUTO_YELLOW
- AUTO_YELLOW: Works the same way as the previous state, the color is yellow for the main lights and red for the secondary lights. Buttons and timers still works the same way as AUTO_RED state, but with the value of the respective color.
  The next state go back to AUTO_RED.

7.2 fsm_manual_run

This function is a FSM responsible for the manual mode. It starts at MANUAL_INIT state and will cycle through 4 state: MANUAL_RED, MANUAL_RED1_YELLOW2, MANUAL_GREEN, MANUAL_YELLOW. In general, it works the same way as automatic mode for the most part, what's different is now the light need to be manually changed by using the second button. The pedestrian scramble still works the same way but now the buzzer will be off.

- MANUAL_INIT: Initialize timers and transmit a message through UART to indicate the mode: "!MANUAL#".
  The next state is MANUAL_RED.
- MANUAL_RED: Set the main light to red and the secondary light to green.
  - timer_flag[2]: This is the timeout timer for the pedestrian light.
  - isBTPressed(0): This checks whether button 1 is pressed, this button is responsible for changing the mode. It will also turn off the pedestrian light in case it is still on while someone has pressed this button. The next mode is tuning mode.
  - isBTPressed(1): This checks whether button 2 is pressed, if it is then it will change to the next state.
  - isBTPressed(3): This checks whether the pedestrian button is pressed, if it is then it will turn on the pedestrian light, but the buzzer won't be turned on.
  - isBTHold(3): This checks whether the pedestrian button is being hold, if it is then it works the same way as the pedestrian button being repeatedly pressed.

  The next state is MANUAL_RED1_YELLOW2.
- MANUAL_RED1_YELLOW2: Set the main light to red and the secondary light to yellow, buttons and timers work the same way as the previous state, but we have an additional timer:
  - timer_flag[6]: This is the timer for blinking yellow light with the interval of 0.25s.

  The next state is MANUAL_GREEN.
- MANUAL_GREEN: Set the main light to green and the secondary light to red, buttons and timers work the same way as the MANUAL_RED state, but with the respective value for the color.
  The next state is MANUAL_YELLOW.
- MANUAL_YELLOW: Set the main light to yellow and the secondary light to red, buttons and timers work the same way as the MANUAL_RED state, but with the respective value for the color.
  The next state go back to MANUAL_RED.

7.3 fsm_tuning_run

This function is a FSM responsible for the tuning mode. It starts at TUNING_INIT state and will cycle through 3 state: TUNING_RED, TUNING_GREEN, TUNING_YELLOW.

- TUNING_INIT: Initialize timers and buffers for the counters, transmit a message through UART to indicate the mode: "!TUNING#".
- TUNING_RED: Changing red state, checks for buttons and timers flags:
  - timer_flag[3]: This is the timer for flashing the respective light, in this case, it is the red lights, the interval is 0.25s.
  - isBTPressed(0): Check whether button 1 is pressed, if it is then change to the next state.
  - isBTHold(0): Check whether button 1 is hold, if it is then change to the state of the next mode. The next mode go back to automatic mode with the state INIT. The value of each light are also balanced out in case it is set to an unbalanced value.
  - isBTPressed(1): Check whether button 2 is pressed, if it is then increment the temporary variable for the counter by 1 unit.
  - isBTHold(1): Check whether button 2 is being hold, if it is then increment the temporary variable for the counter by 10 unit.
  - isBTPressed(2): Check whether button 3 is pressed, if it is then assign the value of the temporary variable to the counter.

  The next state is TUNING_YELLOW.
- TUNING_YELLOW: Changing yellow state, buttons and timers work the same way as the previous state.
  The next state is TUNING_GREEN.
- TUNING_GREEN: Changing green state, buttons and timers work the same way as the previous state.
  The next state go back to TUNING_RED

7.4 buzzer_sound

This function controls the sound of the buzzer.

- The volume can be changed by using "__HAL_TIM_SET_COMPARE". We can control the volume of the buzzer by changing the last parameter of the function.
- The tone or frequency of the buzzer can be controlled by changing the PSC value of timer3. By using the tone library in pitches.h, we can set the according value to the PSC based on the frequency of the note. For example, if we want to play a E7 note, this note will have the frequency of 2637, to determine the value of PSC, we use the following formula:

$$PSC = \frac{F}{f(ARR+1)}$$

So for the E7 note, we will assign the following PSC:

$$PSC = \frac{F}{f(ARR+1)} - 1$$
$$PSC = \frac{64000000}{2637(999+1)} - 1$$
$$PSC \approx 23.2700 = 23$$

7.5 SCH_Add_Task

This function is responsible for adding new task to the scheduler:

- The function iterates through the existing tasks (SCH_MAX_TASKS) to find the appropriate position for the new task based on its initial delay.
- It calculates the sum delay of existing tasks until the insertion point is found.
- If the sum delay exceeds the desired delay for the new task, the insertion point is determined, and the delay of the existing task at that position is adjusted.
- The function then shifts the existing tasks to make room for the new task at the determined index.
- The details of each task (function pointer, delay, period, run flag, and task ID) are updated accordingly for both the existing and new tasks.

7.6 SCH_Update

This function is regularly called to update the state of the tasks managed by the scheduler:

- The function will checks if the first task (SCH_tasks_G[0]) is ready to run (RunMe == 0), and if its execution delay (Delay) has reached zero.
- If the task has a non-zero delay, decrement its delay counter.
- If the task delay has reached zero, set the RunMe flag to indicate that the task is ready to run.

7.7 SCH_Dispatch_Tasks

This function is responsible for dispatching tasks that are ready to run. It also handles the removal and rescheduling of tasks after they have been executed:

- Checks if the first task (SCH_tasks_G[0]) is ready to run (RunMe $>$ 0).
- Executes the task by calling its associated function using the function pointer.
- Resets the RunMe flag for the executed task to 0, indicating that it has been run.
- Temporarily stores the details of the executed task in a local variable (temtask).
- Remove the executed task from the scheduler.
- If the task has a non-zero period, indicating that it should be repeated, adds the task back to the scheduler with its period as the delay.
- After task execution and management, the function typically enters a low-power mode (e.g., sleep mode) to conserve power.

7.8 getKeyInput

This function is responsible for reading the state of multiple buttons and managing their press and hold events:

- The function maintains four arrays (KeyReg0, KeyReg1, KeyReg2, KeyReg3) to store the state of each button for the current and previous cycles.
- It checks if the current state (KeyReg0) is the same as the two previous states (KeyReg1 and KeyReg2). If true, it means the button state has remained consistent for three consecutive cycles.
- If the state has changed (KeyReg2[i] != KeyReg3[i]), it updates KeyReg3 and checks if the button is in the pressed state. If pressed, it sets a timeout value and a flag (BT_flag) indicating a button press.
- If the button was pressed (KeyReg3[i] == PRESSED_STATE), it sets a timeout (TimeOutForKeyPress) and sets a flag (BT_flag) to indicate a button press event.
- The function decrements the timeout counter (TimeOutForKeyPress[i]) for each button. If the timeout reaches zero, it resets the timeout and checks for button hold events.
- If the button returns to the normal state (KeyReg3[i] == NORMAL_STATE) and a press event flag is set (BT_flag[i] == 1), it sets a flag (BT_press_flag) to indicate a button was pressed and released.
- If the button is in the pressed state and the timeout expires, it sets a flag (BT_hold_flag) to indicate a button hold event.

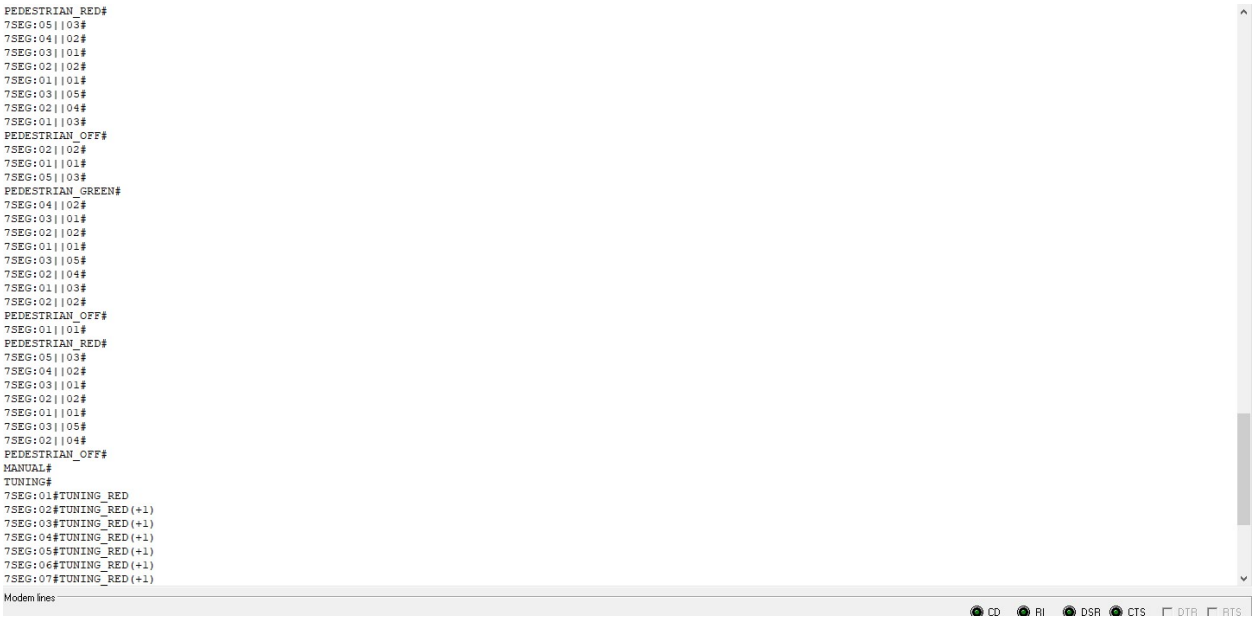
8 Results

+ The traffic lights work as intended on all 3 modes.
+ The USART displays the correct information.
+ The pedestrian light displayed the correct color based on the color of the main traffic light. The pedestrian light also automatically turn off after 8s.
+ The buzzer produces sound with the correct tone and volume. It also increases the loudness and speed when the pedestrian light is flashing yellow.
- The system can detect button press and hold, but the logic for handling button hold is still incorrect and unoptimized in several way.
- The increase in loudness and speed of the pedestrian buzzer is not dynamically bound to the counter of the main traffic light, but instead depend only on the state of the main traffic light.
- The logic for pedestrian scramble and UART are not separated but instead being embedded in the 3 FSM, which lead to less code readability as the code is longer.

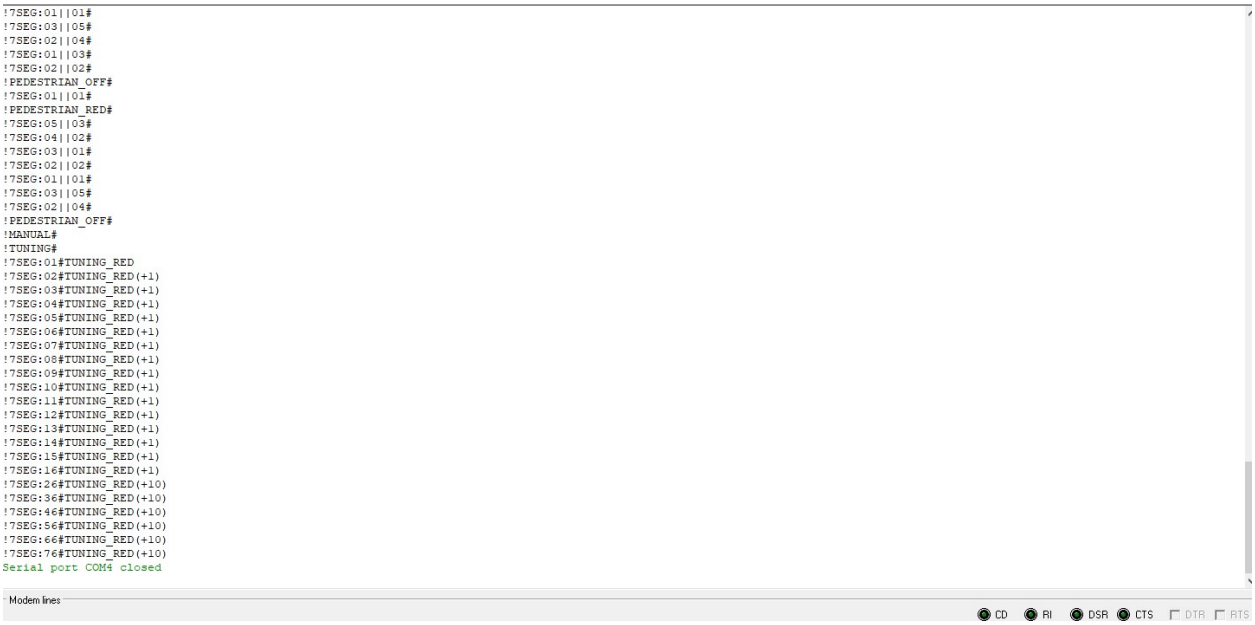
Testing the traffic light system on the STM32F103RB, we can observe the following results on the UART terminal:

```
Serial port COM4 closed
Serial port COM4 opened
SEG:02||02!7SEG:01||01#
!7SEG:05||03#
!7SEG:04||02#
!7SEG:03||01#
!7SEG:02||02#
!7SEG:01||01#
!7SEG:03||05#
!PEDESTRIAN_RED#
!7SEG:02||04#
!7SEG:01||03#
!7SEG:02||02#
!7SEG:01||01#
!7SEG:05||03#
!7SEG:04||02#
!7SEG:03||01#
!7SEG:02||02#
!PEDESTRIAN_OFF#
!7SEG:01||01#
!7SEG:03||05#
!7SEG:02||04#
!7SEG:01||03#
!7SEG:02||02#
!PEDESTRIAN_RED#
!7SEG:01||01#
!7SEG:05||03#
!7SEG:04||02#
!7SEG:03||01#
!7SEG:02||02#
!7SEG:01||01#
!7SEG:03||05#
!7SEG:02||04#
!7SEG:01||03#
!7SEG:02||02#
!7SEG:01||01#
!PEDESTRIAN_RED#
!7SEG:05||03#
!7SEG:04||02#
!7SEG:03||01#
!7SEG:02||02#
!7SEG:01||01#
```

Hình 6: *UART terminal 1: Automatic mode + Pedestrian scramble*



Hình 7: *UART terminal 2: Manual + Tuning mode*



Hình 8: *UART terminal 3: Tuning mode, changing value*

9 Conclusion

In conclusion, we've navigated through the essential components and functionalities that make this system works. From the standard automatic mode, mimicking the everyday traffic lights signal, to the interactive manual mode, offering users the control of the traffic lights, this project taps into the versatility of microcontrollers for practical applications.

The inclusion of a tuning mode allows for changing the timing of each light, showcasing adaptability to specific needs. Additionally, the pedestrian scramble feature adds a safety dimension, synchronizing pedestrian signals with vehicular lights to enhance overall traffic management efficiency.

As we reflect on this project, it becomes evident that the STM32F103RB microcontroller, with its capabilities and programmability, serves as a valuable tool in shaping intelligent traffic solutions. By emphasizing simplicity and practicality, this report underscores the potential of embedded systems in real-world applications, contributing to the ongoing evolution of transportation technology. In addition, though this project, we as a team have earn a lot of valuable experience on working with the STM32F103RB microcontroller, as well as how to design a efficient FSM, and most important of all, how to work as a team and contribute to the whole project.