MCU REPORT 2023

(CO3009) Microprocessor – Microcontroller

Final Project

Submitted to: Mr. LÊ TRỌNG NHÂN

Tô Hoàng Phong - 2112012
Trần Tiến Đạt - 2113162
Trần Anh Tài - 2114700

Ho Chi Minh University of Technology (HCMUT)
Computer science and engineering faculty

Mục lục

1 Github repository URL and hardware implementation video

- Github repository URL of the final project: https://github.com/kido2k3/TRAFFIC_LIGHT_PROJECT
- Hardware implementation video of the final project: https://youtu.be/YdtGXknd5L8

2 Requirement

2.1 Project Description:

Using STM32F103RBT to build traffic lights, which has normal mode, pedestrian mode, adjustment mode and manually Set mode.

2.2 Some Requirements

2.2.1 Software

- *Normal mode*: operates like a normal traffic light
- *Pedestrian mode*: for walkers who want to cross the street
- *Adjustment mode*: for admin who wants to adjust the parameters (countdown number) of each light.
- *Manually Set mode*: operates based on how the admin controls it.

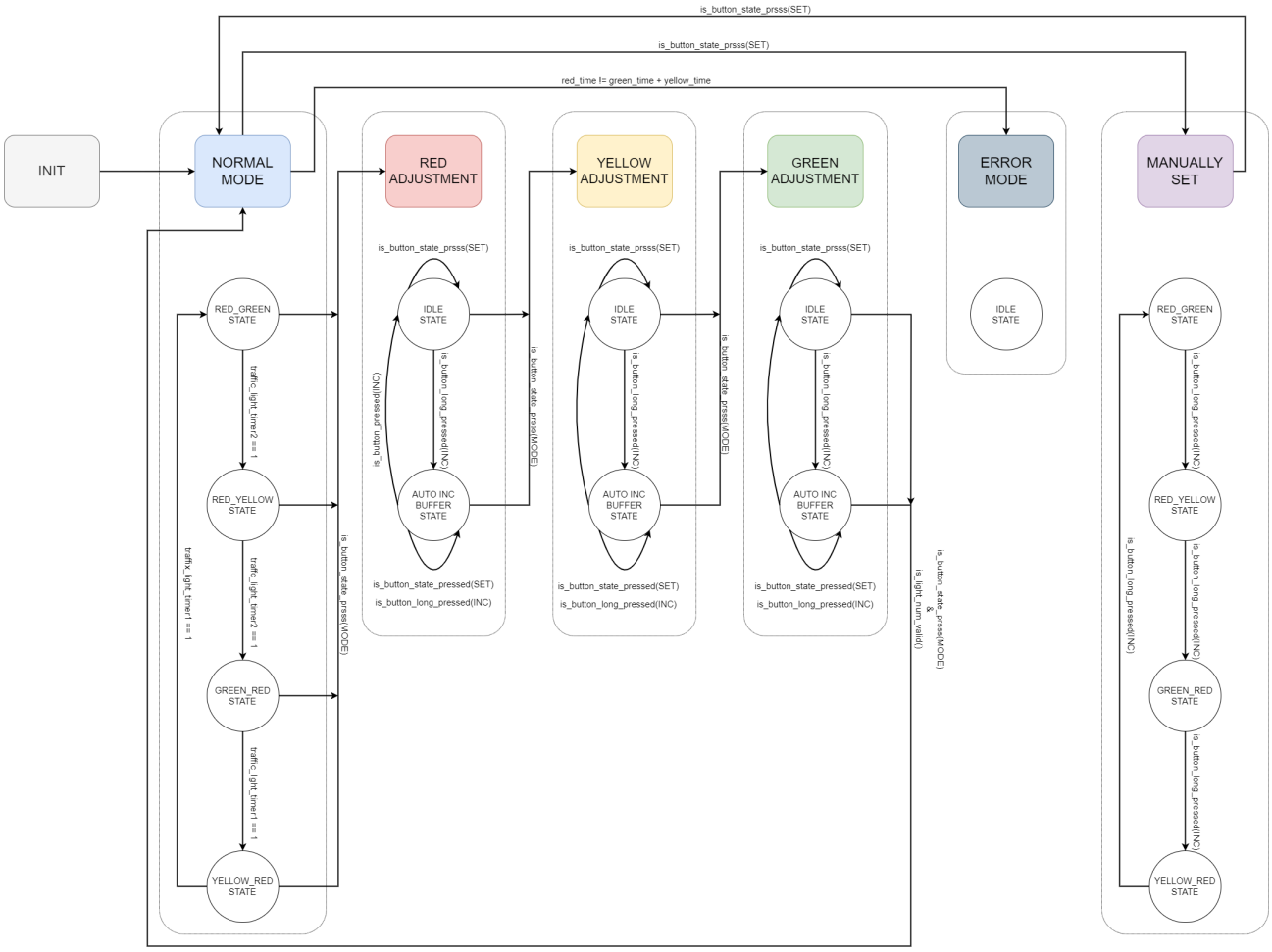
2.2.2 Hardware

- *Light*:
  - First way light (Red + Green)
  - Second way light (Red + Green)
  - Pedestrian light (Red + Green)
- *Button*:
  - Mode button: for switching mode
  - Set button: for setting temporary value in buffer to LED's buffer
  - Increase button: for increasing temporary value in buffer
  - Pedestrian button: switch the current traffic light's mode to pedestrian mode.
- *Buzzer*:
  In pedestrian mode, when the red light countdowns to 5, the buzzer starts ringing with **louder sound and faster cycle**
- *UART*:
  Sending current light number or adjustment information via UART

3 Finite state machine and behaviors of each state

3.1 System Finite State Machine

3.1.1 Finite State Machine



Hình 1: System Finite State Machine

3.1.2 Behaviors

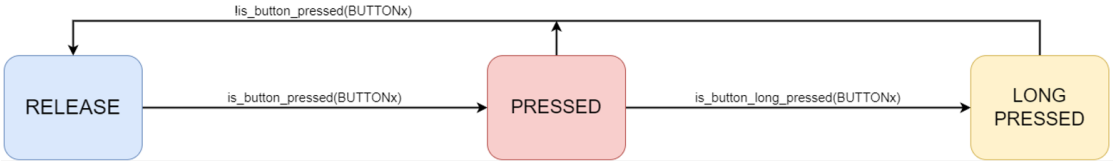
Short brief: System Finite State Machine has 5 states (TRAFFIC_LIGHT, RED_ADJUSTMENT, YELLOW_ADJUSTMENT, GREEN_ADJUSTMENT, ERROR_). Each state has some behaviors (see figure **2**).

Name	Main functions
TRAFFIC_LIGHT	Display via UART Decrease traffic light timer every 1s light_pre_st = TRAFFIC_LIGHT Run Pedestrian finite state machine
RED ADJUSTMENT	Blink red led in 2Hz Display red buffer via UART light_pre_st = RED_ADJUSTMENT
YELLOW ADJUSTMENT	Blink yellow led in 2Hz Display yellow buffer via UART light_pre_st = YELLOW_ADJUSTMENT
GREEN ADJUSTMENT	Blink green led in 2Hz Display green buffer via UART light_pre_st = GREEN_ADJUSTMENT
MANUALLY SET	Display current light
ERROR	Turn off all LEDs

Hình 2: Behavior of each state

3.2 Button Finite State Machine

3.2.1 Finite State Machine



Hình 3: State machine of each button

3.2.2 Behaviors

This project has 4 buttons: MODE button; SET Value; INC button; PEDES button. MODE, SET, PEDES buttons do not have LONG_PRESSED state in their fsm.

a. Button 0 (MODE)

The MODE button is used to transmit the current mode of traffic light

Name	Main functions
RELEASE	Do nothing
PRESSED	Change light state (light_st) Set up environment of next state

Hình 4: Behavior of each state

b. Button 1 (SET)

The SET button is used to set value to LED's buffer

Name	Main functions
RELEASE	Do nothing
PRESSED	Change light state (in TRAFFIC_LIGHT or MANUAL_SET state) Set the temporary buffer to current LED's buffer (in ADJUSTMENT state)

Hình 5: Behavior of each state

c. Button 2 (INC)

The INC button is used to increase the value of the LED's buffer.

Name	Main functions
RELEASE	Do nothing
PRESSED	Increase the current buffer (1 unit) (in ADJUSTMENT state)
LONG PRESSED	Increase the current buffer (every 0.25s) (in ADJUSTMENT state)

Hình 6: Behavior of each state

d. Button 3 (PEDES)

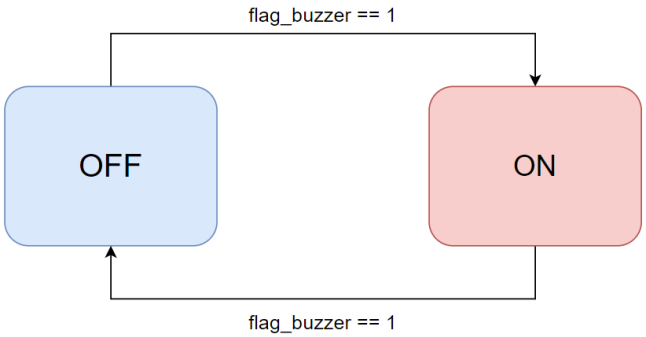
The PEDES button is used for walker (pedestrian light)

Name	Main functions
RELEASE	Do nothing
PRESSED	Run pedestrian countdown (in TRAFFIC_LIGHT or MANUALLY_SET state)

Hình 7: Behavior of each states

3.3 Buzzer Finite State Machine

3.3.1 Finite State Machine



Hình 8: Behavior of each state

3.3.2 Behaviors

The buzzer's intensity is an attribute of the buzzer structure. Therefore, FSM will not execute it.

Name	Main functions
OFF	Set up environment of ON state
ON	Generate PWM depending on light countdown

Hình 9: Behavior of each state

3.4 Universal Asynchronous Receiver-Transmitter - UART

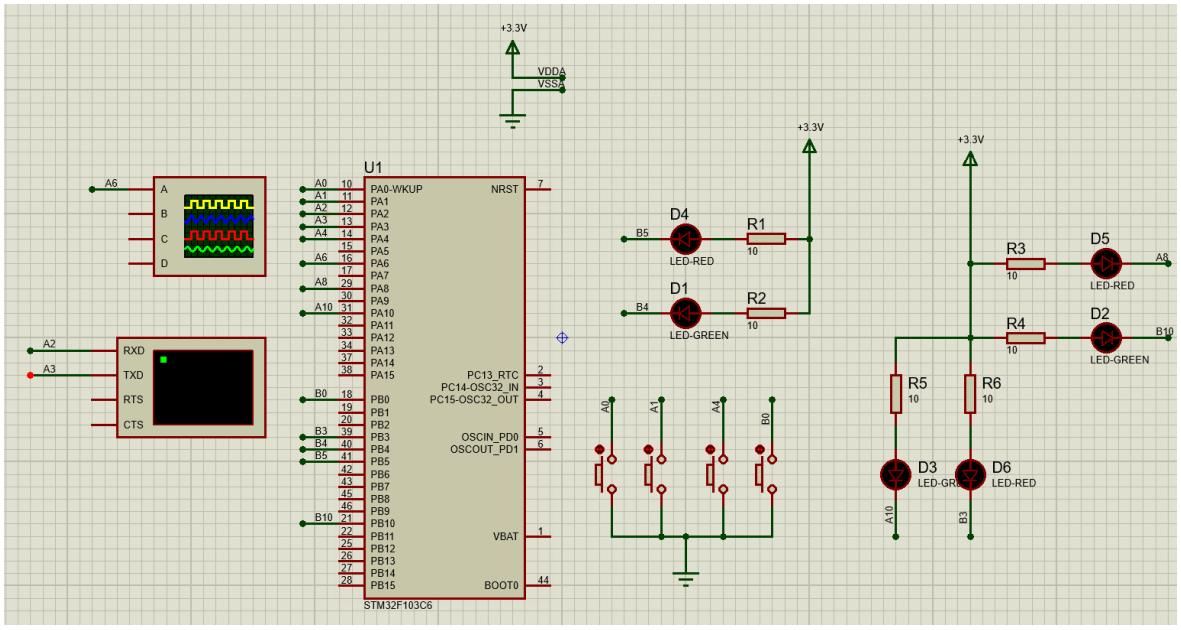
3.4.1 Data Package Format:

The UART communication consists of 3 types of information: MODE information, LIGHT information, and 7SEGMENT information.

Information	Format	Description
Mode	!MODE:<*number* >#	Send Mode number
7-segment	!7SEG<led_index>:<*number* >#	Send 2 7-segments number
Red	!RED:<*number* >#	Send Red's buffer number
Yellow	!YELLOW:<*number* >#	Send Yellow's buffer number
Green	!GREEN:<*number* >#	Send Green's buffer number

Hình 10: UART Communication Format

4 Schematic and mapping table



Hình 11: Schematic

5 File organization

- my_define.h: Define all ports used in the system
- my_button.h - my_button.c: Define the number of buttons, the structure of buttons, some button functions, hardware button-reading functions
- my_buzzer.h - my_buzzer.c: Define the number of buzzers, the structure of buzzers, some buzzer functions, API to generate PWN of buzzers.
- my_led.h - my_led.c: Control red, green and yellow lightf in traffic light
- my_system.h - my_system.c: Define the Interrupt Service Routine of timer, start timer function, and initialize hardware components.
- my_fsm.h - my_fsm.c: Define all fsm functions, their state variables.
- my_scheduler.h - my_scheduler.c: Define all scheduler APIs.
- my_uart.h - my_uart.c: Define the UART's finite state machine to communicate with terminal.

6 Some important structs and functions

6.1 Pin Assignment

```
// All buttons are pull-up
#define BUTTON1_PORT GPIOA
#define BUTTON1 GPIO_PIN_1
#define BUTTON2_PORT GPIOA
#define BUTTON2 GPIO_PIN_4
#define BUTTON3_PORT GPIOB
#define BUTTON3 GPIO_PIN_0
// define pedestrian button - pull up
#define BUTTON4_PORT GPIOA
#define BUTTON4 GPIO_PIN_0
```

```
// define output

#define TL_GREEN_PORT1 GPIOB // Traffic light 1st
#define TL_GREEN1 GPIO_PIN_10
#define TL_RED_PORT1 GPIOB // Traffic light 1st
#define TL_RED1 GPIO_PIN_4

#define TL_PORT2 GPIOB // Traffic light 2nd
#define TL_GREEN2 GPIO_PIN_5
#define TL_RED2 GPIO_PIN_3

#define TL_PED_GREEN_PORT GPIOC // Pedestrian light
#define TL_PED_GREEN GPIO_PIN_7
#define TL_PED_RED_PORT GPIOA // Pedestrian light
#define TL_PED_RED GPIO_PIN_8

typedef uint8_t bool;
```

6.2 Scheduler

6.2.1 Update O(1):

```
void sch_update(void)
{
    if (stack_task.top == 0)
        return;
    if (stack_task.top->counter > 0)
    {
        stack_task.top->counter--;
    }
}
```

6.2.2 Add Task O(n):

```
bool sch_add_task(void (*pTask)(), uint16_t delay, uint16_t period)
{
    struct task *my_task = (struct task *)malloc(sizeof(struct task));
    my_task->pTask = pTask;
    my_task->counter = delay *FREQUENCY_OF_TIM/1000;
    my_task->period = period;
    my_task->next_task = 0;
    if (stack_task.top == 0)
    {
        stack_task.top = my_task;
        // stack_task.bottom = stack_task.top;
        // stack_task.time_length = stack_task.top->counter;
        return 1;
    }
    // if (delay >= stack_task.time_length)
    // {
    //     my_task->counter = delay - stack_task.time_length;
    //     stack_task.bottom->next_task = my_task;
    //     stack_task.bottom = stack_task.bottom->next_task;
    //     stack_task.time_length += my_task->counter;
    //     return 1;
    // }
    struct task *pre = stack_task.top;
    struct task *cur = stack_task.top;
    while (cur && my_task->counter >= cur->counter)
    {
        my_task->counter = my_task->counter - cur->counter;
        pre = cur;
        cur = cur->next_task;
    }
    if (pre != cur)
    {
        pre->next_task = my_task;
        my_task->next_task = cur;
    }
    else
    {
```

```
        my_task->next_task = cur;
        stack_task.top = my_task;
    }
    if (cur)
        cur->counter -= my_task->counter;
    return 1;
}
```

6.2.3 Remove Task:

```
void sch_delete_task(struct task *del_task)
{
    if (del_task == 0)
    {
        return;
    }
    free(del_task);
}
```

6.2.4 Dispatch O(1):

```
bool sch_dispatch(void)
{
    if (stack_task.top == 0)
        return 0;
    if (stack_task.top->counter == 0)
    {
        (*stack_task.top->pTask)();
        struct task *temp = stack_task.top;
        stack_task.top = stack_task.top->next_task;
        temp->next_task = 0;
        if (temp->period != 0)
        {
            sch_add_task(temp->pTask, temp->period, temp->period);
        }
        sch_delete_task(temp);
        return 1;
    }
    return 0;
}
```

6.3 Initial Function

```
void init(void)
{
  HAL_TIM_Base_Start_IT(&htim2);
  button_init();
  init_led();
  uart_Init();
  buzzer_init();
  sch_add_task(button_read, 0, READ_BUTTON_TIME);
}
```

6.4 Running Scheduler Function

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == htim2.Instance)
  {
    sch_update();
  }
}
```

```
void loop(void)
{
  sch_dispatch();
  fsm();
}
```

6.5 LED

6.5.1 LED Structure

```
struct {
  GPIO_TypeDef *green_port;
  GPIO_TypeDef *red_port;
  uint16_t green;
  uint16_t red;
} traffic_light[2], pedestrian_light;
```

6.5.2 LED Initialization

LED Assignment

```
void init_led(void) {
  init_traffic_light();
}
void init_traffic_light(void) {
  traffic_light[0].green_port = TL_GREEN_PORT1;
  traffic_light[0].green = TL_GREEN1;
  traffic_light[0].red_port = TL_RED_PORT1;
  traffic_light[0].red = TL_RED1;
  traffic_light[1].green_port = TL_PORT2;
  traffic_light[1].red_port = TL_PORT2;
  traffic_light[1].green = TL_GREEN2;
  traffic_light[1].red = TL_RED2;

  pedestrian_light.green_port = TL_PED_GREEN_PORT;
  pedestrian_light.red_port = TL_PED_RED_PORT;
  pedestrian_light.red = TL_PED_RED;
  pedestrian_light.green = TL_PED_GREEN;
}
```

6.5.3 LED Control

```
/*
 * @brief:  display traffic light function
 * @para: i - id of traffic light
 *       red, green - state of red and green led (1: on, 0: off)
 * @retval: none*/
void control_traffic_light(unsigned i, GPIO_PinState red, GPIO_PinState green) {
  HAL_GPIO_WritePin(traffic_light[i].red_port, traffic_light[i].red, red);
  HAL_GPIO_WritePin(traffic_light[i].green_port, traffic_light[i].green, green);
}
/*
 * @brief:  display pedestrian light function
 * @para: red, green - state of red and green led (1: on, 0: off)
 * @retval: none*/
void control_pedestrian_light(GPIO_PinState red, GPIO_PinState green) {
  HAL_GPIO_WritePin(pedestrian_light.red_port, pedestrian_light.red, red);
  HAL_GPIO_WritePin(pedestrian_light.green_port, pedestrian_light.green, green);
}
```

6.6 Button

6.6.1 Button define

```
#define NUMBER_OF_BUTTON  4

#define RELEASE       1
#define PRESSED       0
#define LONG_PRESSED_TIME 150 // 1.5s
```

6.7 Button Structure

```
struct {
  bool reg[3];
  bool is_pressed;
  bool is_long_pressed;
  unsigned int timer;

  GPIO_TypeDef *port;
  uint16_t pin;
} button[NUMBER_OF_BUTTON];
```

6.7.1 Button Initialization

Button Assignment

```
/*
 * button[0]: transitioning-mode button
 * button[1]: increasing-value button
 * button[2]: setting-value button
 * button[3]: pedestrian button*/
void button_init(void) {
  for (int i = 0; i < NUMBER_OF_BUTTON; i++) {
    button[i].reg[0] = button[i].reg[1] = button[i].reg[2] = RELEASE;
    button[i].is_long_pressed = 0;
    button[i].is_pressed = 0;
    button[i].timer = LONG_PRESSED_TIME;
  }
  // port and pin were matched by hand
  button[0].port = BUTTON1_PORT;
  button[0].pin = BUTTON1;
  button[1].port = BUTTON2_PORT;
  button[1].pin = BUTTON2;
  button[2].port = BUTTON3_PORT;
  button[2].pin = BUTTON3;
  button[3].port = BUTTON4_PORT;
  button[3].pin = BUTTON4;
}
```

6.7.2 Button Control

```
/*
 * @brief:  read the value of all buttons
 * @para: none
 * @retval: none
 * */
void button_read(void) {
  for (unsigned i = 0; i < NUMBER_OF_BUTTON; i++) {
    button[i].reg[0] = button[i].reg[1];
    button[i].reg[1] = button[i].reg[2];
    button[i].reg[2] = HAL_GPIO_ReadPin(button[i].port, button[i].pin);
    if (button[i].reg[0] == button[i].reg[1]
        && button[i].reg[1] == button[i].reg[2]) {
      //stable state, not bouncing
      if (button[i].reg[2] == PRESSED) {
        button[i].is_pressed = 1;
        //decrease counter to toggle is_long_pressed flag
        if (button[i].timer > 0) {
          button[i].timer--;
        } else {
          button[i].is_long_pressed = 1;
        }
      } else {
        button[i].is_long_pressed = button[i].is_pressed = 0;
        button[i].timer = LONG_PRESSED_TIME;
      }
    }
  }
}
/*
 * @brief:  return the is_pressed flag
 * @para: i - id of button
 * @retval: is_pressed (0: released, 1: pressed)
 * */
```

```c
bool is_button_pressed(unsigned i) {
  if (i >= NUMBER_OF_BUTTON)
    return ERROR;
  return button[i].is_pressed;
}
/*
 * @brief:  return the is_long_pressed flag
 * @para: i - id of button
 * @retval: is_pressed (1: long-pressed, 0: not)
 * */
bool is_button_long_pressed(unsigned i) {
  if (i >= NUMBER_OF_BUTTON)
    return ERROR;
  return button[i].is_long_pressed;
}
```

6.8 UART

6.8.1 UART Control:

```c
/*brief: send the current mode once.
 * para: mode - the current number mode
 * retval: none
 * */
void uart_SendMode(uint8_t mode) {
  if (mode > 4 || mode == 0)
    return;
  char str[45];
  uint16_t len = sprintf(str, "!MODE:%d#\n", mode);
  HAL_UART_Transmit(&huart2, (uint8_t*) str, len, 10);
}
/*brief: send the traffic light timer in traffic mode
 * para:  id - the index of traffic light 0 or 1
 *      number - the value of timers
 * retval: none
 * */
void uart_SendTimeTraffic(uint8_t id, uint16_t number) {
  if (id > 1)
    return;
  char str[20];
  uint16_t len = sprintf(str, "!7SEG%d:%d#\n", id, number);
  HAL_UART_Transmit(&huart2, (uint8_t*) str, len, 10);
}
/*brief: send the red buffer in RED_ADJUSTMENT
 * para:  number - the value of buffers
 * retval: none
 * */
void uart_SendBufferRed(uint16_t number) {
  char str[20];
  uint16_t len = sprintf(str, "!RED:%d#\n", number);
  HAL_UART_Transmit(&huart2, (uint8_t*) str, len, 10);
}
/*brief: send the green buffer in GREEN_ADJUSTMENT
 * para:  number - the value of buffers
 * retval: none
 * */
void uart_SendBufferGreen(uint16_t number) {
  char str[20];
  uint16_t len = sprintf(str, "!GREEN:%d#\n", number);
  HAL_UART_Transmit(&huart2, (uint8_t*) str, len, 10);
}
/*brief: send the yellow buffer in YELLOW_ADJUSTMENT
 * para:  number - the value of buffers
 * retval: none
 * */
void uart_SendBufferYellow(uint16_t number) {
  char str[20];
  uint16_t len = sprintf(str, "!YELLOW:%d#\n", number);
  HAL_UART_Transmit(&huart2, (uint8_t*) str, len, 10);
}
```

6.9 Finite state machine

6.9.1 All FSM Encode Number

```
enum {
  release, pressed, long_pressed
} /*state variable of button*/button_st[4];

enum {
  TRAFFIC_LIGHT,
  RED_ADJUSTMENT,
  YELLOW_ADJUSTMENT,
  GREEN_ADJUSTMENT,
  SET_VALUE,
  INCREASE_BY_1,
  INCREASE_BY_1_OVER_TIME,
  MANUALLY_SET
} /*state variable of system*/light_st = TRAFFIC_LIGHT,
/* previous state variable of system*/light_pre_st = TRAFFIC_LIGHT;
enum {
  RED_GREEN, RED_YELLOW, GREEN_RED, YELLOW_RED, UNEQUAL
} /* state variable of traffic light*/tl_st = RED_GREEN, man_tl_st = RED_GREEN;
enum {
  ON, OFF
} /* state variable of single led*/led_st, pedestrian_st = OFF;
enum {
  BUZZER_ON, BUZZER_OFF
} /* state variable of single led*/bz_st = BUZZER_OFF;
```

6.9.2 FSM's variables Initialiaztion

```
unsigned red_time = RED_TIME_INIT;
unsigned green_time = GREEN_TIME_INIT;
unsigned yellow_time = YELLOW_TIME_INIT;

unsigned red_time_buffer = RED_TIME_INIT;
unsigned green_time_buffer = GREEN_TIME_INIT;
unsigned yellow_time_buffer = YELLOW_TIME_INIT;

unsigned traffic_light_timer1 = RED_TIME_INIT;
unsigned traffic_light_timer2 = GREEN_TIME_INIT;

unsigned pedestrian_timer = 0; // timer to auto turn off pedestrian light when no one
    press for a while

bool flag_toggle_led = 1;
bool flag_countdown = 1;
bool flag_increase_over_time = 1;
bool flag_pedestrian_on = 1;
bool flag_buzzer = 1;
```

6.9.3 System FSM

```
/**
 * @brief  Top-layer finite state machine
 * @param  None
 * @retval None
 */
void fsm(void) {
  switch (light_st) {
  case TRAFFIC_LIGHT:
    if (red_time != green_time + yellow_time) {
      // off all leds
      control_traffic_light(0, 0, 0);
      control_traffic_light(1, 0, 0);
    } else {
      // decrease timer every 1s
      if (flag_countdown == 1) {
        flag_countdown = 0;
        uart_SendTimeTraffic(0, traffic_light_timer1);
        uart_SendTimeTraffic(1, traffic_light_timer2);
```

```
        sch_add_task(task_countdown_1sec, ONE_SECOND, 0);
      }
      traffic_light_fsm();
    }
    if(!pedestrian_timer){
      buzzer_calculation(6);
      buzzer_off();

      control_pedestrian_light(0, 0);
    } else if(pedestrian_timer){
      if (flag_pedestrian_on) {
        flag_pedestrian_on = 0;
        sch_add_task(task_countdown_pedestrian_timer, ONE_SECOND, 0);
      }
      switch(tl_st){
      case RED_GREEN:
        control_pedestrian_light(0, 1);
        buzzer_fsm();
        break;
      case RED_YELLOW:
        control_pedestrian_light(0, 1);
        buzzer_fsm();
        break;
      case GREEN_RED:
        control_pedestrian_light(1, 0);
        buzzer_calculation(6);
        buzzer_off();
        break;
      case YELLOW_RED:
        control_pedestrian_light(1, 0);
        buzzer_calculation(6);
        buzzer_off();
        break;
      default:
        break;
      }
    }
    // transition to adjustment mode
    button0_fsm();
    // transistion to manual setting mode
    button2_fsm();
    // button to control pedestrian led
    button3_fsm();
    break;
  case RED_ADJUSTMENT:
    // update buffer of red with the condition that previous state has to be different
     from changing-value states
    if (light_pre_st != INCREASE_BY_1 && light_pre_st != SET_VALUE
        && light_pre_st != INCREASE_BY_1_OVER_TIME) {
      red_time_buffer = red_time;
    }
    // update buffer of four 7-seg leds: value of red buffer and the mode (2)

    fsm_led();
    // transition mode function
    button0_fsm();
    button1_fsm();
    button2_fsm();

    break;
  case YELLOW_ADJUSTMENT:
    // update buffer of yellow with the condition that previous state has to be
    different from changing-value states
    if (light_pre_st != INCREASE_BY_1 && light_pre_st != SET_VALUE
        && light_pre_st != INCREASE_BY_1_OVER_TIME)
      yellow_time_buffer = yellow_time;
    // update buffer of four 7-seg leds: value of yellow buffer and the mode (3)

    fsm_led();
    // transition mode function
    button0_fsm();
    button1_fsm();
    button2_fsm();

    break;
```

```
92    case GREEN_ADJUSTMENT:
93      // update buffer of green with the condition that previous state has to be
        different from changing-value states
94      if (light_pre_st != INCREASE_BY_1 && light_pre_st != SET_VALUE
95          && light_pre_st != INCREASE_BY_1_OVER_TIME)
96        green_time_buffer = green_time;
97      // update buffer of four 7-seg leds: value of yellow buffer and the mode (4)
98
99      fsm_led();
100     // transition mode function
101     button0_fsm();
102     button1_fsm();
103     button2_fsm();
104
105     break;
106   case SET_VALUE:
107     // update the time value based-on previous state
108     if (light_pre_st == RED_ADJUSTMENT) {
109       red_time = red_time_buffer;
110     } else if (light_pre_st == YELLOW_ADJUSTMENT) {
111       yellow_time = yellow_time_buffer;
112     } else if (light_pre_st == GREEN_ADJUSTMENT) {
113       green_time = green_time_buffer;
114     }
115
116     light_st = light_pre_st;
117     light_pre_st = SET_VALUE;
118     break;
119   case INCREASE_BY_1:
120       // increase the time value based-on previous state (short-pressed)
121       increase_value();
122       switch (light_pre_st) {
123       case RED_ADJUSTMENT:
124         uart_SendMode(2);
125         uart_SendBufferRed(red_time_buffer);
126         break;
127       case YELLOW_ADJUSTMENT:
128         uart_SendMode(3);
129         uart_SendBufferYellow(yellow_time_buffer);
130         break;
131       case GREEN_ADJUSTMENT:
132         uart_SendMode(4);
133         uart_SendBufferGreen(green_time_buffer);
134         break;
135       default:
136         break;
137       }
138       light_st = light_pre_st;
139       light_pre_st = INCREASE_BY_1;
140       break;
141   case INCREASE_BY_1_OVER_TIME:
142     // increase the time value every 0.25s based-on previous state (short-pressed)
143     if (light_pre_st == RED_ADJUSTMENT) {
144
145     } else if (light_pre_st == YELLOW_ADJUSTMENT) {
146
147     } else if (light_pre_st == GREEN_ADJUSTMENT) {
148
149     }
150     if (flag_increase_over_time == 1) {
151       flag_increase_over_time = 0;
152       sch_add_task(task_increase_over_time, INCREASE_TIME, 0);
153     }
154     button1_fsm();
155     break;
156   case MANUALLY_SET:
157     manually_traffic_state();
158     if(!pedestrian_timer){
159       control_pedestrian_light(0, 0);
160     } else if(pedestrian_timer){
161       if (flag_pedestrian_on) {
162         flag_pedestrian_on = 0;
163         sch_add_task(task_countdown_pedestrian_timer, ONE_SECOND, 0);
164       }
165       switch(man_tl_st){
```

```
      case RED_GREEN:
        control_pedestrian_light(0, 1);
        break;
      case RED_YELLOW:
        control_pedestrian_light(0, 1);
        break;
      case GREEN_RED:
        control_pedestrian_light(1, 0);
        break;
      case YELLOW_RED:
        control_pedestrian_light(1, 0);
        break;
      default:
        break;
      }
    }
    //control traffic light
    button1_fsm();
    //return to traffic light mode
    button2_fsm();
    // button to control pedestrian led
    button3_fsm();
    break;

  }
}
```

6.9.4 TrafficLight FSM

```
/*
 * @brief:  finite state machine to control behavior of traffic light
 * @para: none
 * @retval: none*/
void traffic_light_fsm(void) {
  switch (tl_st) {
  case RED_GREEN:
    control_traffic_light(0, 1, 0);
    control_traffic_light(1, 0, 1);
    if (traffic_light_timer2 <= 0) {
      traffic_light_timer2 = yellow_time;
      tl_st = RED_YELLOW;
    }
    break;
  case RED_YELLOW:
    control_traffic_light(0, 1, 0);
    control_traffic_light(1, 1, 1);
    if (traffic_light_timer2 <= 0) {
      traffic_light_timer1 = green_time;
      traffic_light_timer2 = red_time;
      tl_st = GREEN_RED;
    }
    break;
  case GREEN_RED:
    control_traffic_light(0, 0, 1);
    control_traffic_light(1, 1, 0);
    if (traffic_light_timer1 <= 0) {
      traffic_light_timer1 = yellow_time;
      tl_st = YELLOW_RED;
    }
    break;
  case YELLOW_RED:
    control_traffic_light(0, 1, 1);
    control_traffic_light(1, 1, 0);
    if (traffic_light_timer1 <= 0) {
      traffic_light_timer1 = red_time;
      traffic_light_timer2 = green_time;
      tl_st = RED_GREEN;
    }
    break;
  default:
    break;
  }
}
```

6.9.5 ManuallyTrafficLight FSM

```
/*
 * @brief:  finite state machine to display traffic light manually
 * @para: none
 * @retval: none*/
void manually_traffic_state(void) {
  switch (man_tl_st) {
  case RED_GREEN:
    control_traffic_light(0, 1, 0);
    control_traffic_light(1, 0, 1);
    break;

  case GREEN_RED:
    control_traffic_light(0, 0, 1);
    control_traffic_light(1, 1, 0);
    break;
  default:
    control_traffic_light(0, 1, 1);
    control_traffic_light(1, 1, 1);
    break;
  }
}
```

6.9.6 LED FSM

```
/*@brief: state machine to blink led in 2Hz
 * @para: none
 * @retval: none*/
void fsm_led(void) {
  // transition state in 0.25s
  if (flag_toggle_led) {
    flag_toggle_led = 0;
    sch_add_task(task_toggle_led, TOGGLE_TIME, 0);
  }
  switch (led_st) {
  case ON:
    switch (light_st) {
    case RED_ADJUSTMENT:
      // turn red led on
      control_traffic_light(0, 1, 0);
      control_traffic_light(1, 1, 0);
      break;
    case YELLOW_ADJUSTMENT:
      // turn yellow led on
      control_traffic_light(0, 1, 1);
      control_traffic_light(1, 1, 1);
      break;
    case GREEN_ADJUSTMENT:
      // turn green led on
      control_traffic_light(0, 0, 1);
      control_traffic_light(1, 0, 1);
      break;
    default:
      break;
    }
    break;
  case OFF:
    control_traffic_light(0, 0, 0);
    control_traffic_light(1, 0, 0);
    break;
  }
}
```

6.9.7 Button FSM

```
/*
 * @brief:  mode button fsm - 2 states
 * @para: none
 * @retval: 1 - successful
 *       0 - fail
```

```
 * */
bool button0_fsm(void) {
  switch (button_st[0]) {
  case release:
    if (is_button_pressed(0) == 1) {
      // to do
      light_pre_st = light_st;
      control_pedestrian_light(0, 0);
      pedestrian_timer = 0;
      switch (light_st) {
      case TRAFFIC_LIGHT:
        red_time_buffer = red_time;
        yellow_time_buffer = yellow_time;
        green_time_buffer = green_time;
        light_st = RED_ADJUSTMENT;
        uart_SendMode(2);
        uart_SendBufferRed(red_time_buffer);
        break;
      case RED_ADJUSTMENT:
        light_st = YELLOW_ADJUSTMENT;
        uart_SendMode(3);
        uart_SendBufferYellow(yellow_time_buffer);
        break;
      case YELLOW_ADJUSTMENT:
        light_st = GREEN_ADJUSTMENT;
        uart_SendMode(4);
        uart_SendBufferGreen(green_time_buffer);
        break;
      case GREEN_ADJUSTMENT:
        light_st = TRAFFIC_LIGHT;
        tl_st = RED_GREEN;
        traffic_light_timer1 = red_time;
        traffic_light_timer2 = green_time;
        break;
      default:
        break;
      }
      button_st[0] = pressed;
    } else if (is_button_pressed(0) == ERROR)
      return 0;
    break;
  case pressed:
    if (!is_button_pressed(0)) {
      button_st[0] = release;
    } else {
      return 0;
    }
    break;
  default:
    return 0;
  }
  return 1;
}
/*
 * @brief:  setting-value button fsm - 2 states
 * @para: none
 * @retval: 1 - successful
 *       0 - fail
 * */
bool button2_fsm(void) {
  switch (button_st[2]) {
  case release:
    if (is_button_pressed(2) == 1) {
      // to do
      light_pre_st = light_st;
      switch (light_st) {
      case TRAFFIC_LIGHT:
        light_st = MANUALLY_SET;
        break;
      case MANUALLY_SET:
        light_st = TRAFFIC_LIGHT;
        break;
      default:
        light_st = SET_VALUE;

```

```
      }
      button_st[2] = pressed;
    } else if (is_button_pressed(2) == ERROR)
      return 0;
    break;
  case pressed:
    if (!is_button_pressed(2)) {
      button_st[2] = release;
    } else {
      return 0;
    }
    break;
  default:
    return 0;
    break;
  }
  return 1;
}
/*
 * @brief:  increasing-value button fsm - 3 states
 * @para: none
 * @retval: 1 - successful
 *       0 - fail
 * */
bool button1_fsm(void) {
  switch (button_st[1]) {
  case release:
    if (is_button_pressed(1) == 1) {
      // to do
      light_pre_st = light_st;
      switch (light_st) {
      case MANUALLY_SET:
        switch (man_tl_st) {
        case RED_GREEN:
          man_tl_st = GREEN_RED;
          break;
        case GREEN_RED:
          man_tl_st = RED_GREEN;
          break;
        default:
          man_tl_st = GREEN_RED;
        }
        break;
      default:
        light_st = INCREASE_BY_1;
        break;
      }
      button_st[1] = pressed;
    } else if (is_button_pressed(1) == ERROR)
      return 0;
    break;
  case pressed:
    if (!is_button_pressed(1)) {
      button_st[1] = release;
    } else if (is_button_long_pressed(1) == 1) {
      button_st[1] = long_pressed;
    } else {
      return 0;
    }
    break;
  case long_pressed:
    // to do
    if (light_st != INCREASE_BY_1_OVER_TIME) {
      light_pre_st = light_st;
      light_st = INCREASE_BY_1_OVER_TIME;
    }
    if (!is_button_pressed(1)) {
      light_st = light_pre_st;
      light_pre_st = INCREASE_BY_1_OVER_TIME;
      button_st[1] = release;
    }
    break;
  default:
    return 0;
    break;
```

```
  }
  return 1;
}
/*
 * @brief:  pedestrian button
 * @para: none
 * @retval: 1 - successful
 *      0 - fail
 * */
bool button3_fsm(void) {
  switch (button_st[3]) {
  case release:
    if (is_button_pressed(3) == 1) {
      // to do
      pedestrian_timer = PEDESTRIAN_TIMER;
//      flag_pedestrian_on = 1;
      button_st[3] = pressed;
    } else if (is_button_pressed(3) == ERROR)
      return 0;
    break;
  case pressed:
    if (!is_button_pressed(2)) {
      button_st[3] = release;
    } else {
      return 0;
    }
    break;
  default:
    return 0;
    break;
  }
  return 1;
}
```

6.9.8 Buzzer FSM

```
void buzzer_fsm(void){
  switch(bz_st){
  case BUZZER_ON:
    if(flag_buzzer){
      flag_buzzer = 0;
      sch_add_task(task_buzzer_on, buzzer_getToggle_time(), 0);
      bz_st = BUZZER_OFF;
    }
    break;
  case BUZZER_OFF:
    if(flag_buzzer){
      flag_buzzer = 0;
      sch_add_task(task_buzzer_off, buzzer_getToggle_time(), 0);
      bz_st = BUZZER_ON;
    }
    break;
  }
}
```

6.10 Main Loop and Set Up before Main Loop

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
```

```
15
16    /* Configure the system clock */
17    SystemClock_Config();
18
19    /* USER CODE BEGIN SysInit */
20
21    /* USER CODE END SysInit */
22
23    /* Initialize all configured peripherals */
24    MX_GPIO_Init();
25    MX_TIM2_Init();
26    MX_TIM3_Init();
27    MX_USART2_UART_Init();
28    /* USER CODE BEGIN 2 */
29    init();
30    /* USER CODE END 2 */
31
32    /* Infinite loop */
33    /* USER CODE BEGIN WHILE */
34    while (1) {
35      loop();
36      /* USER CODE END WHILE */
37
38      /* USER CODE BEGIN 3 */
39    }
40    /* USER CODE END 3 */
41  }
```