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

MICROPROCESSOR MICRONTROLLER (CO3009)

LAB 3
BUTTONS/SWITCHES

Class: L04 – Semester HK251
Submission Date: 31/10/2025

Instructor: Mr. Ton Huynh Long

Student Name	**Student ID**
Phạm Công Võ	2313946

Ho Chi Minh City, October 2025

Table of Contents

List of Figures

1 Objectives

In this lab, you will

- Learn how to add new C source files and C header files in an STM32 project,
- Learn how to read digital inputs and display values to LEDs using a timer interrupt of a microcontroller (MCU).
- Learn how to debounce when reading a button.
- Learn how to create an FSM and implement an FSM in an MCU.

2 Introduction

Embedded systems usually use buttons (or keys, or switches, or any form of mechanical contacts) as part of their user interface. This general rule applies from the most basic remote-control system for opening a garage door, right up to the most sophisticated aircraft autopilot system. Whatever the system you create, you need to be able to create a reliable button interface.

A button is generally hooked up to an MCU so as to generate a certain logic level when pushed or closed or “active"and the opposite logic level when unpushed or open or “inactive."The active logic level can be either ‘0’ or ‘1’, but for reasons both historical and electrical, an active level of ’0’ is more common.

We can use a button if we want to perform operations such as:

- Drive a motor while a switch is pressed.
- Switch on a light while a switch is pressed.
- Activate a pump while a switch is pressed.

These operations could be implemented using an electrical button without using an MCU; however, use of an MCU may well be appropriate if we require more complex behaviours. For example:

- Drive a motor while a switch is pressed.

  Condition: If the safety guard is not in place, don’t turn the motor. Instead sound a buzzer for 2 seconds.
- Switch on a light while a switch is pressed.

  Condition: To save power, ignore requests to turn on the light during daylight hours.
- Activate a pump while a switch is pressed

  Condition: If the main water reservoir is below 300 litres, do not start the main pump: instead, start the reserve pump and draw the water from the emergency tank.

In this lab, we consider how you read inputs from mechanical buttons in your embedded application using an MCU.

3 Basic techniques for reading from port pins

3.1 The need for pull-up resistors

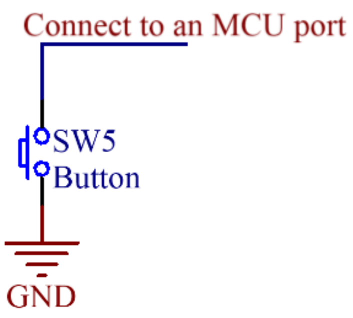


Figure 1: *Connecting a button to an MCU*

Figure 1 shows a way to connect a button to an MCU. This hardware operates as follows:

- When the switch is open, it has no impact on the port pin. An internal resistor on the port “pulls up"the pin to the supply voltage of the MCU (typically 3.3V for STM32F103). If we read the pin, we will see the value ‘1’.
- When the switch is closed (pressed), the pin voltage will be 0V. If we read the pin, we will see the value ‘0’.

However, if the MCU does not have a pull-up resistor inside, when the button is pressed, the read value will be ‘0’, but even we release the button, the read value is still ‘0’ as shown in Figure 2.

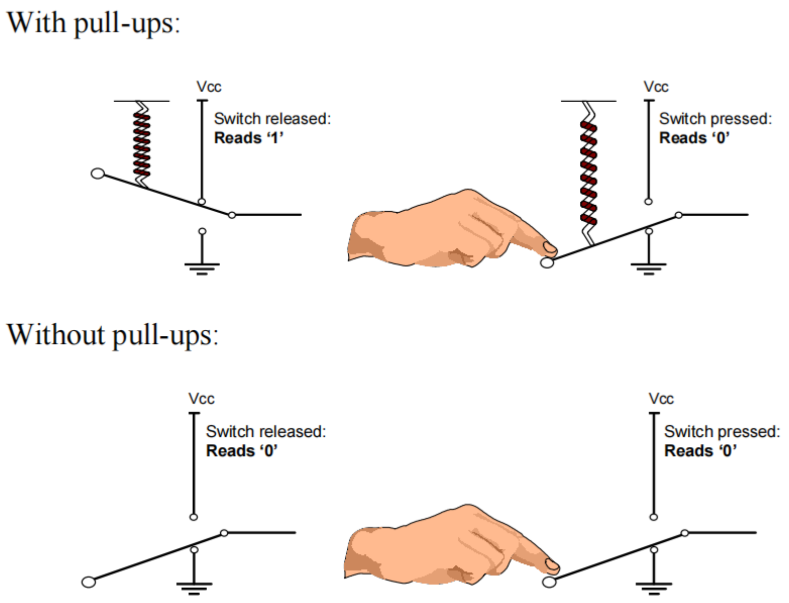


Figure 2: *The need of pull up resistors*

So a reliable way to connect a button/switch to an MCU is that we explicitly use an external pull-up resistor as shown in Figure 3.

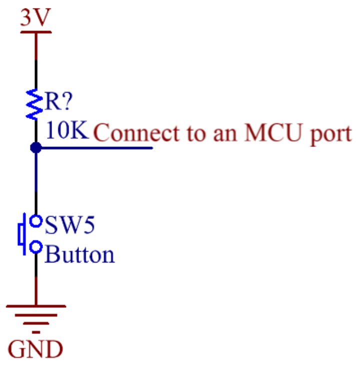


Figure 3: *A reliable way to connect a button to an MCU*

3.2 Dealing with switch bounces

In practice, all mechanical switch contacts bounce (that is, turn on and off, repeatedly, for short period of time) after the switch is closed or opened as shown in Figure 4.

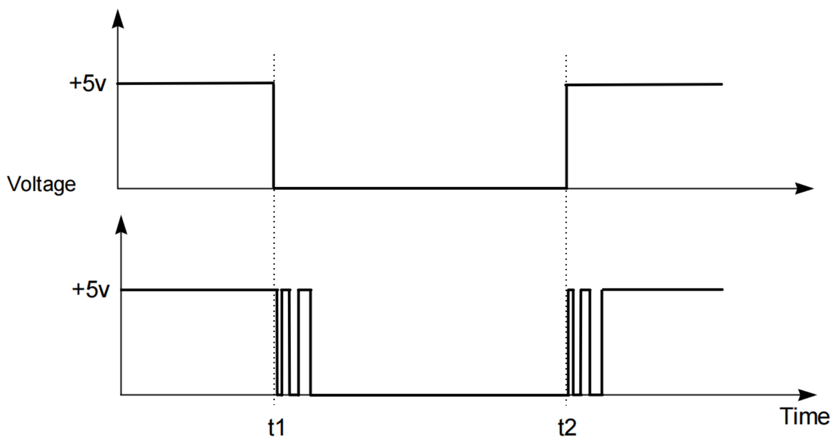


Figure 4: *Switch bounces*

Every system that uses any kind of mechanical switch must deal with the issue of debouncing. The key task is to make sure that one mechanical switch or button action is only read as one action by the MCU, even though the MCU will typically be fast enough to detect the unwanted switch bounces and treat them as separate events. Bouncing can be eliminated by special ICs or by RC circuitry, but in most cases debouncing is done in software because software is “free".

As far as the MCU concerns, each “bounce"is equivalent to one press and release of an “ideal” switch. Without appropriate software design, this can give several problems:

- Rather than reading ‘A’ from a keypad, we may read ‘AAAAA’
- Counting the number of times that a switch is pressed becomes extremely difficult
- If a switch is depressed once, and then released some time later, the ‘bounce’ may make it appear as if the switch has been pressed again (at the time of release).

The key to debouncing is to establish a minimum criterion for a valid button push, one that can be implemented in software. This criterion must involve differences in time - two button presses in 20ms must be treated as one button event, while two button presses in 2 seconds must be treated as two button events. So what are the relevant times we need to consider? They are these:

- Bounce time: most buttons seem to stop bouncing within 10ms
- Button press time: the shortest time a user can press and release a button seems to be between 50 and 100ms
- Response time: a user notices if the system response is 100ms after the button press, but not if it is 50ms after

Combining all of these times, we can set a few goals

- Ignore all bouncing within 10ms
- Provide a response within 50ms of detecting a button push (or release)
- Be able to detect a 50ms push and a 50ms release

The simplest debouncing method is to examine the keys (or buttons or switches) every N milliseconds, where N $>$ 10ms (our specified button bounce upper limit) and N $<=$ 50ms (our specified response time). We then have three possible outcomes every time we read a button:

- We read the button in the solid '0' state
- We read the button in the solid '1' state
- We read the button while it is bouncing (so we will get either a '0' or a '1')

Outcomes 1 and 2 pose no problems, as they are what we would always like to happen. Outcome 3 also poses no problem because during a bounce either state is acceptable. If we have just pressed an active-low button and we read a '1' as it bounces, the next time through we are guaranteed to read a '0' (remember, the next time through all bouncing will have ceased), so we will just detect the button push a bit later. Otherwise, if we read a '0' as the button bounces, it will still be '0' the next time after all bouncing has stopped, so we are just detecting the button push a bit earlier. The same applies to releasing a button. Reading a single bounce (with all bouncing over by the time of the next read) will never give us an invalid button state. It's only reading multiple bounces (multiple reads while bouncing is occurring) that can give invalid button states such as repeated push signals from one physical push.

So if we guarantee that all bouncing is done by the time we next read the button, we're good. Well, almost good, if we're lucky...

MCUs often live among high-energy beasts, and often control the beasts. High energy devices make electrical noise, sometimes great amounts of electrical noise. This noise can, at the worst possible moment, get into your delicate button-and-high-value-pullup circuit and act like a real button push. Oops, missile launched, sorry!

If the noise is too intense we cannot filter it out using only software, but will need hardware of some sort (or even a redesign). But if the noise is only occasional, we can filter it out in software without too much bother. The trick is that instead of regarding a single button 'make' or 'break' as valid, we insist on N contiguous makes or breaks to mark a valid button event. N will be a factor of your button scanning rate and the amount of filtering you want to add. Bigger N gives more filtering. The simplest filter (but still a big improvement over no filtering) is just an N of 2, which means compare the current button state with the last button state, and only if both are the same is the output valid.

Note that now we have not two but three button states: active (or pressed), inactive (or released), and indeterminate or invalid (in the middle of filtering, not yet filtered). In most cases we can treat the invalid state the same as the inactive state, since we care in most cases only about when we go active (from whatever state) and when we cease being active (to inactive or invalid). With that simplification we can look at simple N = 2 filtering reading a button wired to STM32 MCU:

```c
void button_reading(void){
    static unsigned char last_button;
    unsigned char raw_button;
    unsigned char filtered_button;
    last_button = raw_button;
    raw_button = HAL_GPIO_ReadPin(BUTTON_1_GPIO_Port, BUTTON_1_Pin);
    if(last_button == raw_button){
        filtered_button = raw_button;
    }
}
```

The function button_reading() must be called no more often than our debounce time (10ms).

To expand to greater filtering (larger N), keep in mind that the filtering technique essentially involves reading the current button state and then either counting or reseting the counter. We count if the current button state is the same as the last button state, and if our count reaches N we then report a valid new button state. We reset the counter if the current button state is different than the last button state, and we then save the current button state as the new button state to compare against the next time. Also note that the larger our value of N the more often our filtering routine must be called, so that we get a filtered response within our specified 50ms deadline. So for example with an N of 8 we should be calling our filtering routine every 2 - 5ms, giving a response time of 16 - 40ms (>10ms and <50ms).

4 Reading switch input (basic code) using STM32

To demonstrate the use of buttons/switches in STM32, we use an example which requires to write a program that

- Has a timer which has an interrupt in every 10 milliseconds.
- Reads values of button PB0 every 10 milliseconds.
- Increases the value of LEDs connected to PORTA by one unit when the button PB0 is pressed.
- Increases the value of PORTA automatically in every 0.5 second, if the button PB0 is pressed in more than 1 second.

4.1 Input Output Processing Patterns

For both input and output processing, we have a similar pattern to work with. Normally, we have a module named driver which works directly to the hardware. We also have a buffer to store temporarily values. In the case of input processing, the driver will store the value of the hardware status to the buffer for further processing. In the case of output processing, the driver uses the buffer data to output to the hardware.

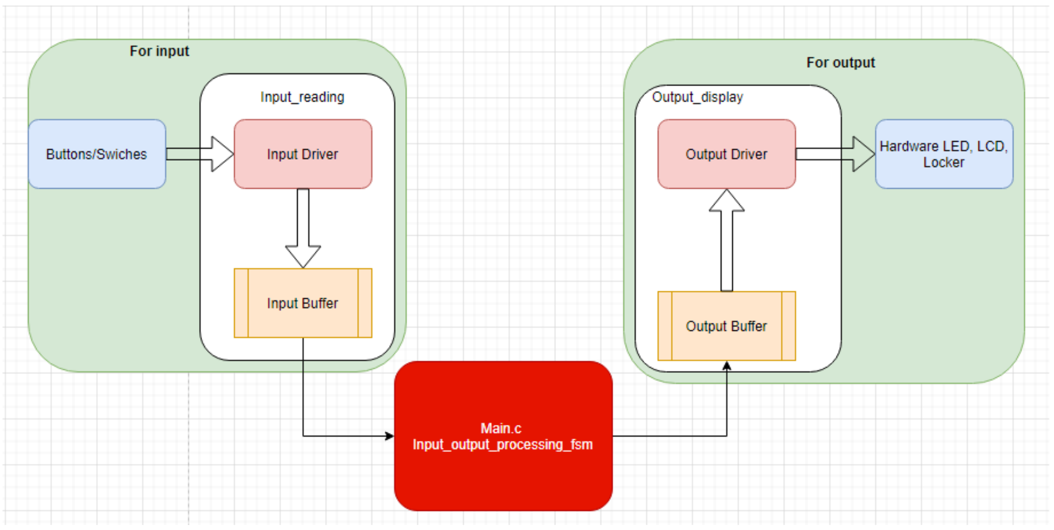


Figure 5: *Input Output Processing Patterns*

Figure 5 shows that we should have an *input_ reading* module to processing the buttons, then store the processed data to the buffer. Then a module of *input_ output_ processing_ fsm* will process the input data, and update the output buffer. The output driver gets the value from the output buffer to transfer to the hardware.

4.2 Setting up

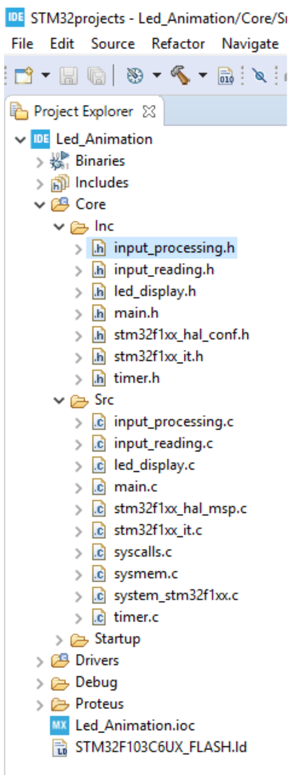
4.2.1 Create a project

Please follow the instruction in Labs 1 and 2 to create a project that includes:

- PB0 as an input port pin,
- PA0-PA7 as output port pins, and
- Timer 2 10ms interrupt

4.2.2 Create a file C source file and header file for input reading

We are expected to have files for button processing and led display as shown in Figure 6.

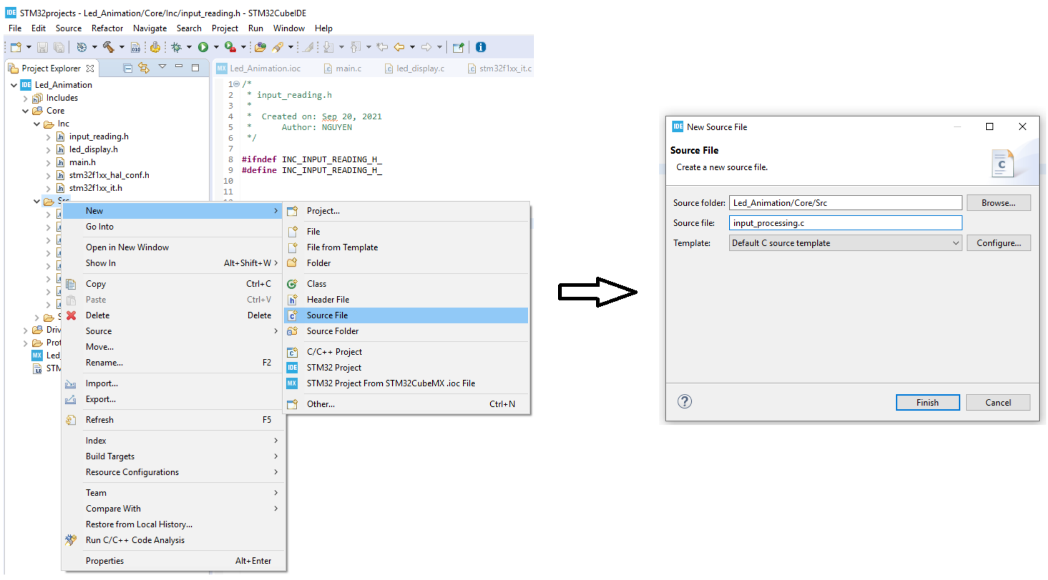


```
Led_Animation
  Binaries
  Includes
  Core
    Inc
      input_processing.h
      input_reading.h
      led_display.h
      main.h
      stm32f1xx_hal_conf.h
      stm32f1xx_it.h
      timer.h
    Src
      input_processing.c
      input_reading.c
      led_display.c
      main.c
      stm32f1xx_hal_msp.c
      stm32f1xx_it.c
      syscalls.c
      sysmem.c
      system_stm32f1xx.c
      timer.c
    Startup
  Drivers
  Debug
  Proteus
  Led_Animation.ioc
  STM32F103C6UX_FLASH.ld
```

Figure 6: *File Organization*

Steps 1 (Figure 7): Right click to the folder **Src**, select **New**, then select **Source File**. There will be a pop-up. Please type the file name, then click **Finish**.

Step 2 (Figure 8): Do the same for the C header file in the folder **Inc**.

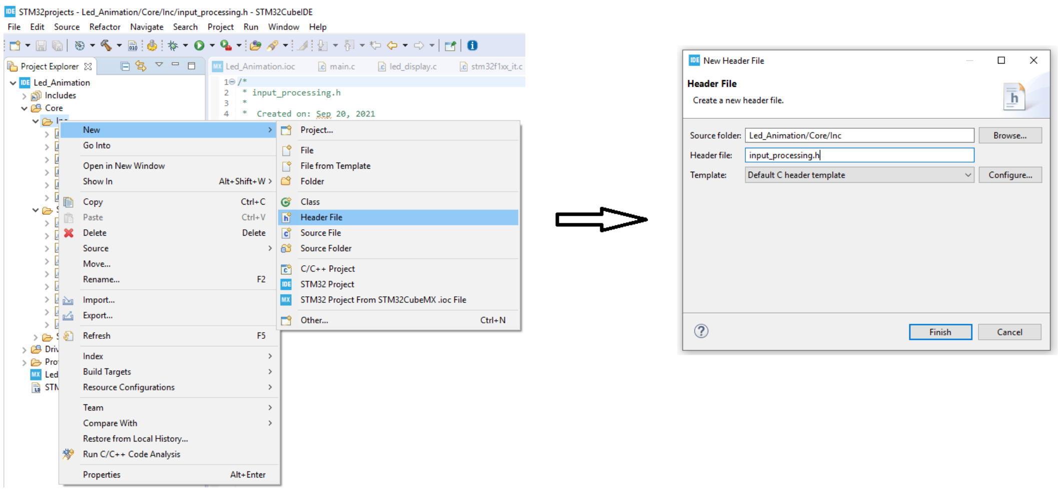


```
/*
 * input_reading.h
 *
 *  Created on: Sep 20, 2021
 *      Author: NGUYEN
 */

#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_
```

Source folder: Led_Animation/Core/Src
Source file: input_processing.c
Template: Default C source template

Figure 7: *Step 1: Create a C source file for input reading*



```
/*
 * input_processing.h
 *
 *  Created on: Sep 20, 2021
```

Source folder: Led_Animation/Core/Inc
Header file: input_processing.h
Template: Default C header template

Figure 8: *Step 2: Create a C header file for input processing*

4.3 Code For Read Port Pin and Debouncing

4.3.1 The code in the input_reading.c file

```c
#include "main.h"
//we aim to work with more than one buttons
#define NO_OF_BUTTONS                     1
//timer interrupt duration is 10ms, so to pass 1 second,
//we need to jump to the interrupt service routine 100 time
#define DURATION_FOR_AUTO_INCREASING     100
#define BUTTON_IS_PRESSED                  GPIO_PIN_RESET
#define BUTTON_IS_RELEASED                 GPIO_PIN_SET
//the buffer that the final result is stored after
//debouncing
static GPIO_PinState buttonBuffer[NO_OF_BUTTONS];
//we define two buffers for debouncing
static GPIO_PinState debounceButtonBuffer1[NO_OF_BUTTONS];
static GPIO_PinState debounceButtonBuffer2[NO_OF_BUTTONS];
//we define a flag for a button pressed more than 1 second.
static uint8_t flagForButtonPress1s[NO_OF_BUTTONS];
//we define counter for automatically increasing the value
//after the button is pressed more than 1 second.
static uint16_t counterForButtonPress1s[NO_OF_BUTTONS];
void button_reading(void){
  for(char i = 0; i < NO_OF_BUTTONS; i ++){
    debounceButtonBuffer2[i] =debounceButtonBuffer1[i];
    debounceButtonBuffer1[i] = HAL_GPIO_ReadPin(BUTTON_1_GPIO_Port,
  BUTTON_1_Pin);
    if(debounceButtonBuffer1[i] == debounceButtonBuffer2[i])
      buttonBuffer[i] = debounceButtonBuffer1[i];
      if(buttonBuffer[i] == BUTTON_IS_PRESSED){
      //if a button is pressed, we start counting
        if(counterForButtonPress1s[i] < DURATION_FOR_AUTO_INCREASING
  ){
          counterForButtonPress1s[i]++;
        } else {
        //the flag is turned on when 1 second has passed
        //since the button is pressed.
```

```
          flagForButtonPress1s[i] = 1;
          //todo
        }
      } else {
        counterForButtonPress1s[i] = 0;
        flagForButtonPress1s[i] = 0;
      }
  }
}
```

```
unsigned char is_button_pressed(uint8_t index){
  if(index >= NO_OF_BUTTONS) return 0;
  return (buttonBuffer[index] == BUTTON_IS_PRESSED);  }
```

```
unsigned char is_button_pressed_1s(unsigned char index){
  if(index >= NO_OF_BUTTONS) return 0xff;
  return (flagForButtonPress1s[index] == 1);  }
```

4.3.2 The code in the input_reading.h file

```
#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_
void button_reading(void);
unsigned char is_button_pressed(unsigned char index);
unsigned char is_button_pressed_1s(unsigned char index);
#endif /* INC_INPUT_READING_H_ */
```

4.3.3 The code in the timer.c file

```
#include "main.h"
#include "input_reading.h"

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim->Instance == TIM2){
    button_reading();
  }
}
```

4.4 Button State Processing

4.4.1 Finite State Machine

To solve the example problem, we define 3 states as follows:

- State 0: The button is released or the button is in the initial state.
- State 1: When the button is pressed, the FSM will change to State 1 that is increasing the values of PORTA by one value. If the button is released, the FSM goes back to State 0.
- State 2: while the FSM is in State 1, the button is kept pressing more than 1 second, the state of FSM will change from 1 to 2. In this state, if the button is kept pressing, the value of PORTA will be increased automatically in every 500ms. If the button is released, the FSM goes back to State 0.

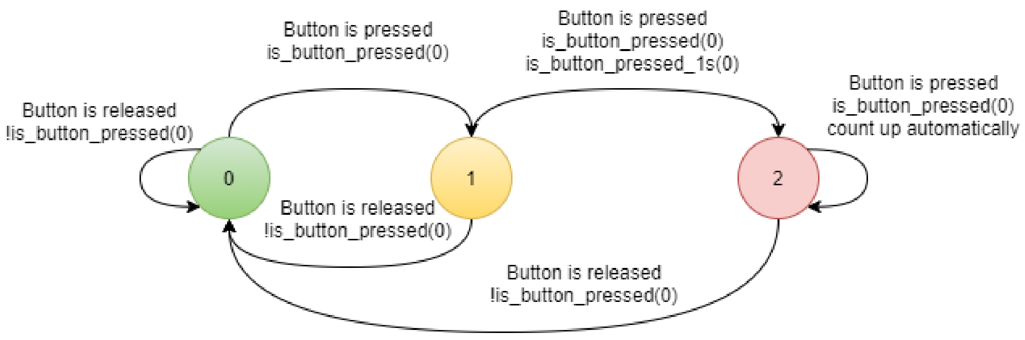


Figure 9: *An FSM for processing a button*

4.4.2 The code for the FSM in the input_processing.c file

Please note that *fsm_for_input_processing* function should be called inside the super loop of the main functin.

```c
#include "main.h"
#include "input_reading.h"

enum ButtonState{BUTTON_RELEASED, BUTTON_PRESSED,
   BUTTON_PRESSED_MORE_THAN_1_SECOND} ;
enum ButtonState buttonState = BUTTON_RELEASED;
void fsm_for_input_processing(void){
  switch(buttonState){
  case BUTTON_RELEASED:
    if(is_button_pressed(0)){
      buttonState = BUTTON_PRESSED;
      //INCREASE VALUE OF PORT A BY ONE UNIT
    }
    break;
  case BUTTON_PRESSED:
    if(!is_button_pressed(0)){
      buttonState = BUTTON_RELEASED;
    } else {
      if(is_button_pressed_1s(0)){
        buttonState = BUTTON_PRESSED_MORE_THAN_1_SECOND;
      }
    }
    break;
  case BUTTON_PRESSED_MORE_THAN_1_SECOND:
    if(!is_button_pressed(0)){
      buttonState = BUTTON_RELEASED;
    }
    //todo
    break;
  }
}
```

4.4.3 The code in the input_processing.h

```c
#ifndef INC_INPUT_PROCESSING_H_
#define INC_INPUT_PROCESSING_H_

void fsm_for_input_processing(void);

#endif /* INC_INPUT_PROCESSING_H_ */
```

4.4.4 The code in the main.c file

```c
#include "main.h"
#include "input_processing.h"
//don't modify this part
int main(void){
    HAL_Init();
    /* Configure the system clock */
    SystemClock_Config();
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_TIM2_Init();
    while (1)
    {
        //you only need to add the fsm function here
        fsm_for_input_processing();
    }
}
```

5 Exercises and Report

The complete LAB3 project source code is available at: GitHub - Traffic Light Control System.
A demonstration of the system's operation can be viewed at: Video Demo Link LAB 3.

5.1 Specifications

You are required to build an application of a traffic light in a cross road which includes some features as described below:

- The application has 12 LEDs including 4 red LEDs, 4 amber LEDs, 4 green LEDs.
- The application has 4 seven segment LEDs to display time with 2 for each road. The 2 seven segment LEDs will show time for each color LED corresponding to each road.
- The application has three buttons which are used

  - to select modes,

  - to modify the time for each color led on the fly, and

  - to set the chosen value.
- The application has at least 4 modes which is controlled by the first button. Mode 1 is a normal mode, while modes 2 3 4 are modification modes. You can press the first button to change the mode. Modes will change from 1 to 4 and back to 1 again.

  Mode 1 - Normal mode:

  - The traffic light application is running normally.

  Mode 2 - Modify time duration for the red LEDs: This mode allows you to change the time duration of the red LED in the main road. The expected behaviours of this mode include:

  - All single red LEDs are blinking in 2 Hz.

  - Use two seven-segment LEDs to display the value.

  - Use the other two seven-segment LEDs to display the mode.

  - The second button is used to increase the time duration value for the red LEDs.

  - The value of time duration is in a range of 1 - 99.

  - The third button is used to set the value.

  Mode 3 - Modify time duration for the amber LEDs: Similar for the red LEDs described above with the amber LEDs.

  Mode 4 - Modify time duration for the green LEDs: Similar for the red LEDs described above with the green LEDs.

5.2 Exercise 1: Sketch an FSM

This exercise requires you to design a Finite State Machine (FSM) that represents your proposed solution strategy. Figure 10 illustrates the FSM for button processing, showing the system's states and the transition conditions that handle user interactions. Figure 11 presents the complete flowchart of the traffic-light system modeled in Proteus, clearly depicting the control logic and the relationships between different operating modes.

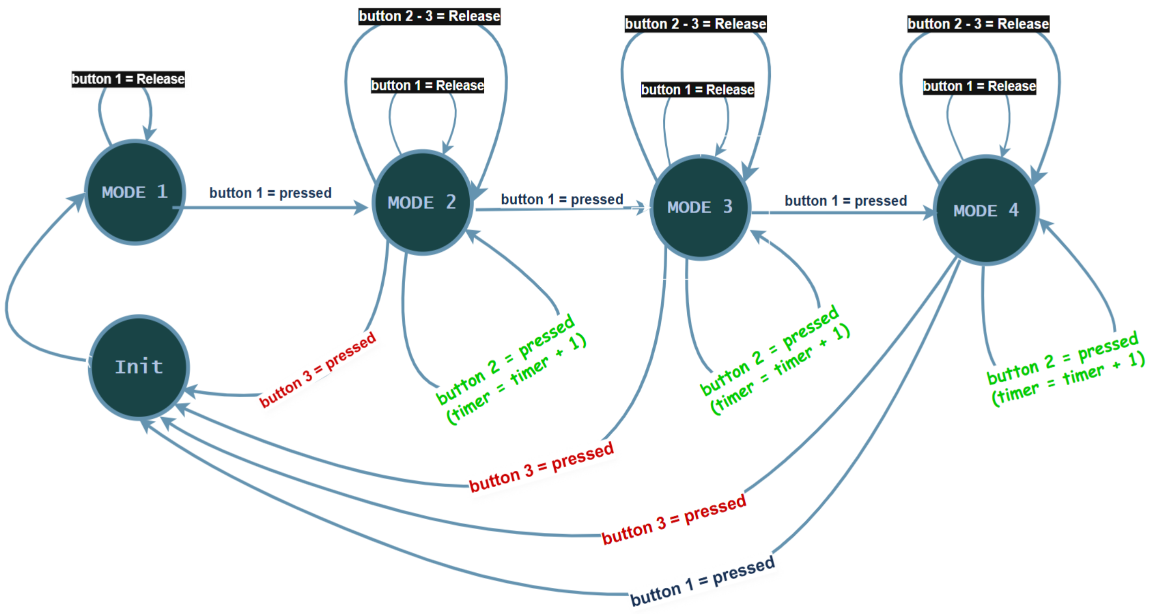


Figure 10: *An FSM for processing a button*

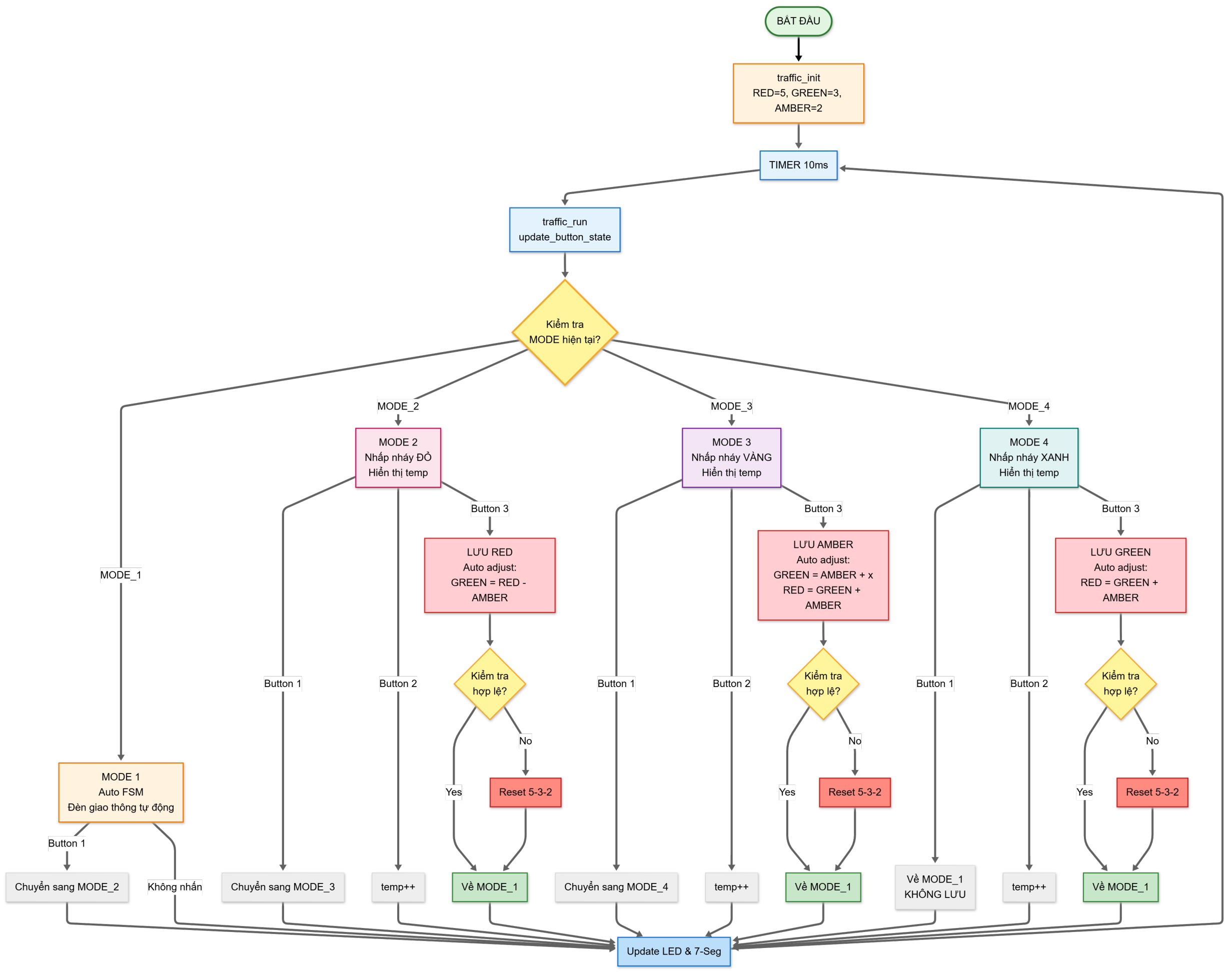


Figure 11: *Flowchart of the Traffic Light Control System*

5.3 Exercise 2: Proteus Schematic

Your task in this exercise is to draw a Proteus schematic for the problem above.

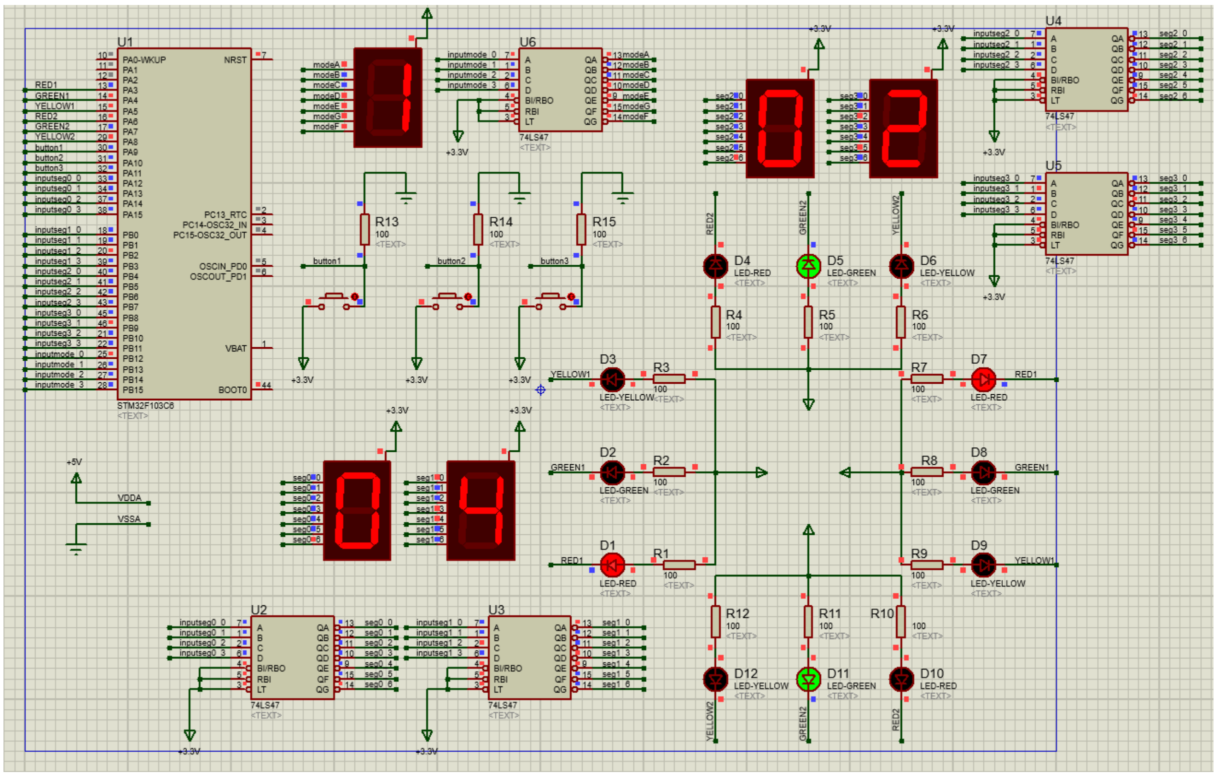


Figure 12: *Proteus of Traffic Light System*

5.4 Exercise 3: Create STM32 Project

Your task in this exercise is to create a project that has pin corresponding to the Proteus schematic that you draw in previous section. You need to set up your timer interrupt is about 10ms.

5.4.1 Timer Configuration Analysis

```
/**
 * Timer 2 Configuration for 10ms Interrupt
 * Given Configuration:
 * - Prescaler = 7999
 * - Period (ARR) = 9
 * - System Clock = 8MHz (HSI - Internal Oscillator)
 *
 * Calculation:
 * Timer Clock = System Clock / (Prescaler + 1)
 *             = 8,000,000 Hz / (7999 + 1)
 *             = 8,000,000 Hz / 8000 = 1,000 Hz (1 kHz)
 * Interrupt Frequency = Timer Clock / (Period + 1)
 *                     = 1,000 Hz / (9 + 1)
 *                     = 1,000 Hz / 10
 *                     = 100 Hz
 * Interrupt Period = 1 / 100 Hz = 10ms
 */
static void MX_TIM2_Init(void)
{
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 7999;      // Divide clock by 8000 1kHz
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 9;            // Counter 0 - 9
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;

    if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) !=
   HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig
   ) != HAL_OK)
    {
        Error_Handler();
    }
}
```

5.4.2 Complete GPIO Pin Configuration

Table 1 presents the detailed GPIO configuration of the STM32F103C6 microcontroller used in the traffic light control system. Each pin is assigned to a specific function corresponding to the Proteus schematic, including LED control, button inputs, and 7-segment display connections.

Bảng 1: *Pin configuration table for STM32F103C6 in Traffic Light System*

No.	Pin Name	Configured Name	Function	Mode
1	PA3	RED1	Red light 1	Output Push-Pull
2	PA4	GREEN1	Green light 1	Output Push-Pull
3	PA5	YELLOW1	Yellow light 1	Output Push-Pul
4	PA6	RED2	Red light 2	Output Push-Pull
5	PA7	GREEN2	Green light 2	Output Push-Pull
6	PA8	YELLOW2	Yellow light 2	Output Push-Pull
7	PA9	button1	Push button 1	Input Pull Up
8	PA10	button2	Push button 2	Input Pull Up
9	PA11	button3	Push button 3	Input Pull Up
10	PA112	inputseg0_0	7-seg display group 0	Output Push-Pull
11	PA13	inputseg0_1	7-seg display group 0	Output Push-Pull
12	PA14	inputseg0_2	7-seg display group 0	Output Push-Pull
13	PA15	inputseg0_3	7-seg display group 0	Output Push-Pull
14	PB0	inputseg1_0	7-seg display group 1	Output Push-Pull
15	PB1	inputseg1_1	7-seg display group 1	Output Push-Pull
16	PB2	inputseg1_2	7-seg display group 1	Output Push-Pull
17	PB3	inputseg1_3	7-seg display group 1	Output Push-Pull
18	PB4	inputseg2_0	7-seg display group 2	Output Push-Pull
19	PB5	inputseg2_1	7-seg display group 2	Output Push-Pull
20	PB6	inputseg2_2	7-seg display group 2	Output Push-Pull
21	PB7	inputseg2_3	7-seg display group 2	Output Push-Pull
22	PB8	inputseg3_0	7-seg display group 3	Output Push-Pull
23	PB9	inputseg3_1	7-seg display group 3	Output Push-Pull
24	PB10	inputseg3_2	7-seg display group 3	Output Push-Pull
25	PB11	inputseg3_3	7-seg display group 3	Output Push-Pull
26	PB12	inputmode0	Mode select 0	Output Push-Pull
27	PB13	inputmode1	Mode select 1	Output Push-Pull
28	PB14	inputmode2	Mode select 2	Output Push-Pull
29	PB15	inputmode3	Mode select 3	Output Push-Pull

Figure 13 illustrates the pin mapping of the STM32F103C6 microcontroller used in this project. Each GPIO pin is configured to correspond with the Proteus schematic, defining specific roles such as LED control, button inputs, and segment display connections.

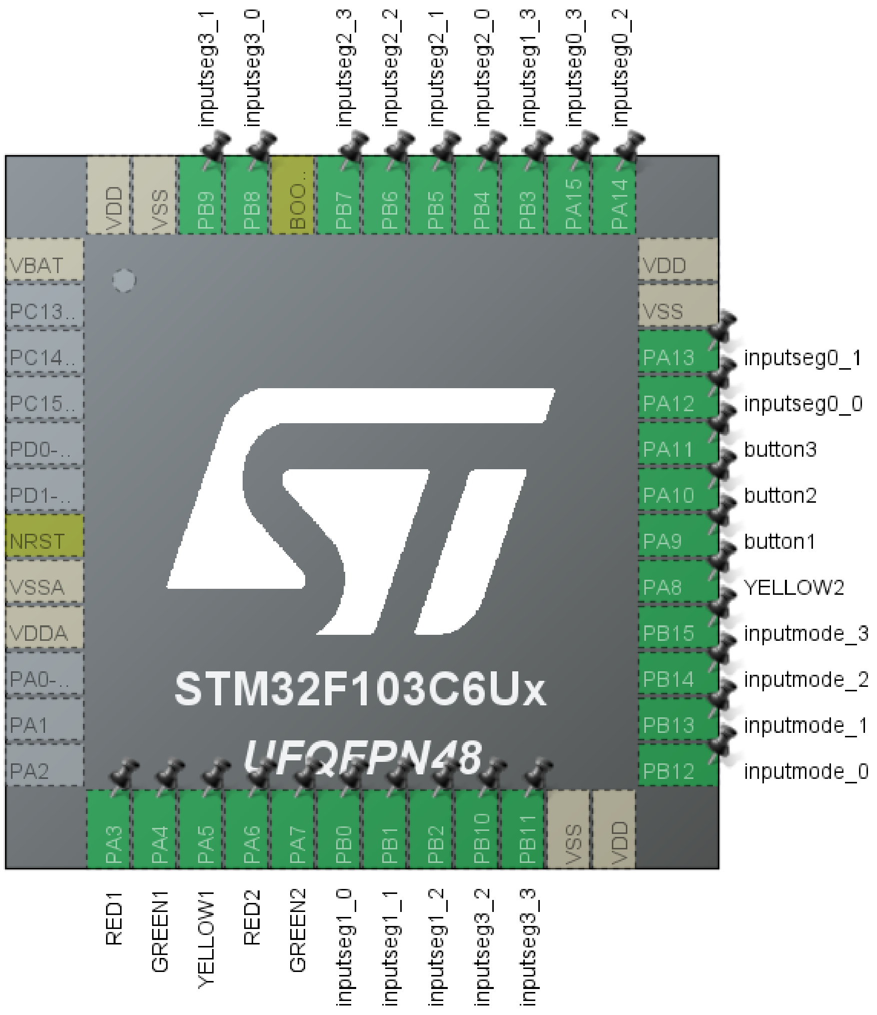


Figure 13: *Proteus of Traffic Light System*

Main Header File

```
// Define to prevent recursive inclusion
#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C"
{
#endif

// Includes
#include "stm32f1xx_hal.h"

// Exported functions prototypes
```

```c
void Error_Handler(void);

/* -------Private defines ---------*/
#define RED1_Pin GPIO_PIN_3
#define RED1_GPIO_Port GPIOA
#define GREEN1_Pin GPIO_PIN_4
#define GREEN1_GPIO_Port GPIOA
#define YELLOW1_Pin GPIO_PIN_5
#define YELLOW1_GPIO_Port GPIOA
#define RED2_Pin GPIO_PIN_6
#define RED2_GPIO_Port GPIOA
#define GREEN2_Pin GPIO_PIN_7
#define GREEN2_GPIO_Port GPIOA
#define inputseg1_0_Pin GPIO_PIN_0
#define inputseg1_0_GPIO_Port GPIOB
#define inputseg1_1_Pin GPIO_PIN_1
#define inputseg1_1_GPIO_Port GPIOB
#define inputseg1_2_Pin GPIO_PIN_2
#define inputseg1_2_GPIO_Port GPIOB
#define inputseg3_2_Pin GPIO_PIN_10
#define inputseg3_2_GPIO_Port GPIOB
#define inputseg3_3_Pin GPIO_PIN_11
#define inputseg3_3_GPIO_Port GPIOB
#define inputmode_0_Pin GPIO_PIN_12
#define inputmode_0_GPIO_Port GPIOB
#define inputmode_1_Pin GPIO_PIN_13
#define inputmode_1_GPIO_Port GPIOB
#define inputmode_2_Pin GPIO_PIN_14
#define inputmode_2_GPIO_Port GPIOB
#define inputmode_3_Pin GPIO_PIN_15
#define inputmode_3_GPIO_Port GPIOB
#define YELLOW2_Pin GPIO_PIN_8
#define YELLOW2_GPIO_Port GPIOA
#define button1_Pin GPIO_PIN_9
#define button1_GPIO_Port GPIOA
#define button2_Pin GPIO_PIN_10
```

```
#define button2_GPIO_Port GPIOA
#define button3_Pin GPIO_PIN_11
#define button3_GPIO_Port GPIOA
#define inputseg0_0_Pin GPIO_PIN_12
#define inputseg0_0_GPIO_Port GPIOA
#define inputseg0_1_Pin GPIO_PIN_13
#define inputseg0_1_GPIO_Port GPIOA
#define inputseg0_2_Pin GPIO_PIN_14
#define inputseg0_2_GPIO_Port GPIOA
#define inputseg0_3_Pin GPIO_PIN_15
#define inputseg0_3_GPIO_Port GPIOA
#define inputseg1_3_Pin GPIO_PIN_3
#define inputseg1_3_GPIO_Port GPIOB
#define inputseg2_0_Pin GPIO_PIN_4
#define inputseg2_0_GPIO_Port GPIOB
#define inputseg2_1_Pin GPIO_PIN_5
#define inputseg2_1_GPIO_Port GPIOB
#define inputseg2_2_Pin GPIO_PIN_6
#define inputseg2_2_GPIO_Port GPIOB
#define inputseg2_3_Pin GPIO_PIN_7
#define inputseg2_3_GPIO_Port GPIOB
#define inputseg3_0_Pin GPIO_PIN_8
#define inputseg3_0_GPIO_Port GPIOB
#define inputseg3_1_Pin GPIO_PIN_9
#define inputseg3_1_GPIO_Port GPIOB

#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */
```

5.4.3 Main Program Structure

```c
int main(void)
{
    /* ===== HAL Initialization ===== */
    HAL_Init();
    /* ===== System Clock Configuration ===== */
    SystemClock_Config();
    /* ===== Peripheral Initialization ===== */
    MX_GPIO_Init();      // Initialize all GPIO pins
    MX_TIM2_Init();      // Initialize Timer 2
    /* ===== Application Initialization ===== */
    traffic_init();      // Initialize traffic light system
    /* ===== Start Timer Interrupt ===== */
    HAL_TIM_Base_Start_IT(&htim2);  // Enable 10ms timer interrupt
    /* ===== Main Loop ===== */
    while (1)
    {
        // All processing happens in timer interrupt
    }
}
```

5.4.4 Timer Interrupt Handler

```c
/**
 * Timer Interrupt Callback - Every 10ms
*/
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM2)
    {   // Software Timer Management
        timerRun();
        /* Button Input Processing */
        getKeyInput();
        /* Traffic Control Logic */
        traffic_run();
    }
}
```

5.4.5 System Heartbeat Loop - traffic_run() function

```c
/**
 * traffic_run() - Main loop called every 10ms
 * CALL RATE: 100Hz (every 10ms)
 */
void traffic_run(void)
{
    // Step 1: Read button state
    update_button_state();
    // Step 2: Process logic based on current mode
    switch(current_mode) {
        case MODE_1_NORMAL:
            fsm_normal_mode();          // Automatic operation
            break;
        case MODE_2_RED_MODIFY:
            fsm_red_modify_mode();      // Adjust RED duration
            break;
        case MODE_3_AMBER_MODIFY:
            fsm_amber_modify_mode();    // Adjust AMBER duration
            break;
        case MODE_4_GREEN_MODIFY:
            fsm_green_modify_mode();    // Adjust GREEN duration
            break;
        default:
            // Optional: handle invalid mode
            break;
    }
    // Step 3: Update LED hardware
    update_led_display();
    // Step 4: Update 7-segment display
    update_7seg_display();
}
```

5.5 Exercise 4: Modify Timer Parameters

Your task in this exercise is to modify the timer settings so that when we want to change the time duration of the timer interrupt, we change it the least and it will not affect the overall system. For example, the current system we have implemented is that it can blink an LED in 2 Hz, with the timer interrupt duration is 10ms. However, when we want to change the timer interrupt duration to 1ms or 100ms, it will not affect the 2Hz blinking LED.

[Report Solution]:

The blinking period of the LED is determined based on the desired frequency according to the basic formula:

$$T = \frac{1}{f}$$

Where:

- T is the full cycle of the LED (seconds).
- f is the blinking frequency of the LED (Hz).

For example, in this traffic light system, the LED blinks at **2 Hz**, meaning the LED changes state every 250 ms, resulting in a full LED cycle of **500 ms**.

According to the exercise requirements, when changing the timer interrupt period, we should only need to adjust a single parameter without affecting the overall system. Therefore, we define a constant `TIMER_INTERRUPT_MS`, representing the timer interrupt period in milliseconds.

In each specific case, to change the timer interrupt duration, we only need to modify the value of `TIMER_INTERRUPT_MS` to match the desired interrupt period, while all other parameters and system logic remain unchanged. This approach ensures flexibility, maintainability, and that the LED and other functions operate correctly without recalculating the prescaler or counter.

The actual period of each timer interrupt is calculated as:

$$T_{\text{timer}} = \frac{(Prescaler + 1) \cdot (CounterPeriod + 1)}{f_{\text{clock}}}$$

Where:

- T_{timer} is the actual period between two consecutive interrupts (seconds).
- **Prescaler** is the division factor used to reduce the clock frequency to a suitable level.
- **CounterPeriod** is the timer's maximum count value (auto-reload).
- f_{clock} is the clock frequency supplied to the timer.

To ensure the LED blinks at the desired frequency, we need to determine the **number of timer interrupts** required for one LED state:

$$\text{Number of interrupts} = \frac{\text{LED state duration}}{T_{\text{timer}}}$$

Using the constant `TIMER_INTERRUPT_MS`, the key parameters can be calculated automatically:

- `MAX_COUNTER` – the number of interrupts needed for the LED to change state (250 ms):

$$MAX\_COUNTER = \frac{250}{TIMER\_INTERRUPT\_MS}$$

- `TimeCycle` – the number of interrupts in 1 second:

$$TimeCycle = \frac{1000}{TIMER\_INTERRUPT\_MS}$$

With this approach, whenever the timer interrupt period changes (e.g., 1 ms, 10 ms, or 100 ms), the number of interrupts required for the LED to change state every 250 ms is automatically adjusted, ensuring the LED always blinks at **2 Hz** without modifying the control logic.

```c
 /* INSTRUCTION: Only modify the value of TIMER_INTERRUPT_MS to adjust
 * EXAMPLES:
 * - TIMER_INTERRUPT_MS = 1  - Timer interrupt occurs every 1 ms
 * - TIMER_INTERRUPT_MS = 10 - Timer interrupt occurs every 10 ms
 * - TIMER_INTERRUPT_MS = 100 - Timer interrupt occurs every 100 ms
 */

// CONFIGURATION PARAMETER
#define TIMER_INTERRUPT_MS 10 // Timer interrupt interval

// AUTO-CALCULATED CONSTANTS
// Number of interrupts per second
#define CYCLES_PER_SECOND (1000 / TIMER_INTERRUPT_MS)
// Number of interrupts per 250 ms
#define CYCLES_PER_250MS (250 / TIMER_INTERRUPT_MS)

// Derived constants for system logic
#define TIMER_CYCLE CYCLES_PER_SECOND  // 1s traffic light updates
#define MAX_BLINK_COUNTER CYCLES_PER_250MS // LED blinking at 2 Hz
```

5.6 Exercise 5: Adding code for button debouncing

Following the example of button reading and debouncing in the previous section, your tasks in this exercise are:

- To add new files for input reading and output display,
- To add code for button debouncing,
- To add code for increasing mode when the first button is pressed.

5.6.1 Files for Input Reading and Button Debouncings

To handle button inputs reliably, we need to create a dedicated module that addresses the bouncing phenomenon - a mechanical issue where metal contacts oscillate during press/release, generating multiple pulses instead of a single pulse.

The button module consists of two files: `button.h` (declarations) and `button.c` (implementation). The debouncing algorithm uses 4 registers (KeyReg0 - 3) to store the state history. Only when 3 consecutive reads (10ms apart) yield identical results is the state accepted as stable, creating a 30ms debounce time.

The `getKeyInput()` function is called every 10ms in the Timer Interrupt to read GPIO pins, compare the 3 reads, detect state changes, and set the `button_flag`.

`button.h`

```c
#ifndef INC_BUTTON_H_
#define INC_BUTTON_H_

#include "main.h"

// Button states (Active-Low with Pull-up)
#define NORMAL_STATE  SET    // Not pressed
#define PRESSED_STATE RESET  // Pressed

// Flags: [0]=Button1, [1]=Button2, [2]=Button3
extern int button_flag[3];          // Short press
extern int button_long_pressed[3]; // Long press (>500 ms)

// Button event checks (return 1 once per event)
int isButton1Pressed(void);
int isButton2Pressed(void);
int isButton3Pressed(void);
int isButton1LongPressed(void);
int isButton2LongPressed(void);
int isButton3LongPressed(void);

// Call every 1 ms in timer interrupt for debouncing
void getKeyInput(void);
/*
 * Usage:
 * - Call getKeyInput() every 1 ms (timer interrupt)
 * - In main:
 *      if (isButton1Pressed()) { ... }
 *      if (isButton1LongPressed()) { ... }
 */
#endif /* INC_BUTTON_H_ */
```

button.c

```c
#include "button.h"
// ============================================================
// GLOBAL VARIABLES
// ============================================================
// Button state history (for debouncing)
int KeyReg0[3] = {NORMAL_STATE, NORMAL_STATE, NORMAL_STATE};
int KeyReg1[3] = {NORMAL_STATE, NORMAL_STATE, NORMAL_STATE};
int KeyReg2[3] = {NORMAL_STATE, NORMAL_STATE, NORMAL_STATE};
int KeyReg3[3] = {NORMAL_STATE, NORMAL_STATE, NORMAL_STATE};

// Long press countdown (100 x 10 ms = 1 s)
int TimeOutForKeyPress[3] = {100, 100, 100};

// Event flags
int button_flag[3] = {0, 0, 0};         // Short press
int button_long_pressed[3] = {0, 0, 0}; // Long press

// Ignore button input during first 100 ms
int startup_counter = 10; // 10 x 10 ms

// ============================================================
// EVENT CHECK FUNCTIONS
// ============================================================
int isButton1Pressed()
{
  if (button_flag[0])
  {
    button_flag[0] = 0;
    return 1;
  }
  return 0;
}
int isButton2Pressed()
{
  if (button_flag[1])
  {
    button_flag[1] = 0;
    return 1;
  }
  return 0;
}
int isButton3Pressed()
{
  if (button_flag[2])
  {
    button_flag[2] = 0;
    return 1;
  }
  return 0;
}

int isButton1LongPressed()
{
  if (button_long_pressed[0])
  {
    button_long_pressed[0] = 0;
    return 1;
  }
  return 0;
```

```c
}
int isButton2LongPressed()
{
  if (button_long_pressed[1])
  {
    button_long_pressed[1] = 0;
    return 1;
  }
  return 0;
}
int isButton3LongPressed()
{
  if (button_long_pressed[2])
  {
    button_long_pressed[2] = 0;
    return 1;
  }
  return 0;
}

// ==============================================================
// HELPER
// ==============================================================
void subKeyProcess(int index) { button_flag[index] = 1; }

// ==============================================================
// MAIN HANDLER (CALL EVERY 10 ms IN TIMER INTERRUPT)
// ==============================================================
void getKeyInput()
{
  if (startup_counter > 0)
  {
    startup_counter--;
    return;
  }

```

```c
for (int i = 0; i < 3; i++)
{
  // Shift history
  KeyReg2[i] = KeyReg1[i];
  KeyReg1[i] = KeyReg0[i];

  // Read GPIO
  switch (i)
  {
  case 0:
    KeyReg0[i] = HAL_GPIO_ReadPin(button1_GPIO_Port, button1_Pin);
    break;
  case 1:
    KeyReg0[i] = HAL_GPIO_ReadPin(button2_GPIO_Port, button2_Pin);
    break;
  case 2:
    KeyReg0[i] = HAL_GPIO_ReadPin(button3_GPIO_Port, button3_Pin);
    break;
  }

  // Debounce check
  if ((KeyReg0[i] == KeyReg1[i]) && (KeyReg1[i] == KeyReg2[i]))
  {
    // Detect state change
    if (KeyReg3[i] != KeyReg2[i])
    {
      KeyReg3[i] = KeyReg2[i];

      if (KeyReg3[i] == PRESSED_STATE)
      {
        subKeyProcess(i);
        TimeOutForKeyPress[i] = 100; // Reset long-press timer
      }
    }
    else
    {
          // Handle long press
          TimeOutForKeyPress[i]--;
          if (TimeOutForKeyPress[i] == 0)
          {
            TimeOutForKeyPress[i] = 100;
            if (KeyReg3[i] == PRESSED_STATE)
              button_long_pressed[i] = 1;
          }
        }
      }
  }
}
```

5.6.2 Add Code for Increasing Mode When Button 1 is Pressed

The system has 4 modes that cycle sequentially: MODE 1 (automatic) → MODE 2 (red adjustment) → MODE 3 (amber adjustment) → MODE 4 (green adjustment) → MODE 1. Button 1 performs the mode switching function.

To avoid detecting multiple presses when the button is held, we use the **rising edge detection** technique: processing only occurs when `prevState == RELEASE` and `currState == PRESS`. The `update_button_state()` function reads the state from `isButton1Pressed()` and compares it with the previous cycle. In each FSM mode function, when a rising edge of the MODE button is detected, the system transitions to the next mode and updates `temp_duration`.

The `update_button_state()` function is responsible for reading button states and detecting rising edges:

```c
void update_button_state(void)
{
    for(int i = 0; i < 3; i++) {
        // Save previous state
        prevState[i] = currState[i];

        // Read new state from hardware
        switch(i) {
            case 0:  // MODE button
                if(isButton1Pressed()) {
                    currState[i] = BTN_PRESS;
                } else {
                    currState[i] = BTN_RELEASE;
                }
                break;

            case 1:  // MODIFY button
                if(isButton2Pressed()) {
                    currState[i] = BTN_PRESS;
                } else {
                    currState[i] = BTN_RELEASE;
                }
                break;

            case 2:  // SET button
                if(isButton3Pressed()) {
                    currState[i] = BTN_PRESS;
                } else {
                    currState[i] = BTN_RELEASE;
                }
                break;
        }
    }
}
```

`FSM Mode 1 - Automatic Mode`

In the function `fsm_normal_mode()`, pressing Button 1 switches the system to MODE 2:

```
void fsm_normal_mode(void)
{
    static int timer_counter = 0;

    // Rising edge detection on MODE button
    if(currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE) {
        current_mode = MODE_2_RED_MODIFY;
        temp_duration = duration_RED;
        turn_off_all_leds();
        return;
    }

    // Timer cycle counter for traffic light sequence
    timer_counter++;
    if(timer_counter < TIMER_CYCLE) {
        return;
    }
    timer_counter = 0;

    // Traffic light FSM logic
    switch(traffic_state) {
        case INIT:
            traffic_state = RED_GREEN;
            counter_road1 = duration_RED;
            counter_road2 = duration_GREEN;
            break;

        case RED_GREEN:
            counter_road1--;
            counter_road2--;
            if(counter_road2 <= 0) {
                traffic_state = RED_AMBER;
                counter_road1 = duration_AMBER;
                counter_road2 = duration_AMBER;
            }
            break;

        case RED_AMBER:
            counter_road1--;
            counter_road2--;
            if(counter_road2 <= 0) {
                traffic_state = GREEN_RED;
                counter_road1 = duration_GREEN;
                counter_road2 = duration_RED;
            }
            break;

        case GREEN_RED:
            counter_road1--;
            counter_road2--;
```

```c
            if(counter_road1 <= 0) {
                traffic_state = AMBER_RED;
                counter_road1 = duration_AMBER;
                counter_road2 = duration_AMBER;
            }
            break;

        case AMBER_RED:
            counter_road1--;
            counter_road2--;
            if(counter_road2 <= 0) {
                traffic_state = RED_GREEN;
                counter_road1 = duration_RED;
                counter_road2 = duration_GREEN;
            }
            break;
    }

    if(counter_road1 < 0) counter_road1 = 0;
    if(counter_road2 < 0) counter_road2 = 0;
}
```

FSM Mode 2 - Red Light Adjustment

In `fsm_red_modify_mode()`, pressing Button 1 switches to MODE 3:

```c
void fsm_red_modify_mode(void)
{
    // MODE button -> switch to AMBER adjustment
    if(currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE) {
        current_mode = MODE_3_AMBER_MODIFY;
        temp_duration = duration_AMBER;
        return;
    }

    // ... (handle MODIFY and SET buttons)
}
```

FSM Mode 3 - Amber Light Adjustment

In `fsm_amber_modify_mode()`, pressing Button 1 switches to MODE 4:

```c
void fsm_amber_modify_mode(void)
{
    // MODE button -> switch to GREEN adjustment
    if(currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE) {
        current_mode = MODE_4_GREEN_MODIFY;
        temp_duration = duration_GREEN;
        return;
    }

    // ... (handle MODIFY and SET buttons)
}
```

FSM Mode 4 - Green Light Adjustment

In `fsm_green_modify_mode()`, pressing Button 1 returns to MODE 1:

```c
void fsm_green_modify_mode(void)
{
    // MODE button -> return to automatic mode (no save)
    if(currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE) {
        current_mode = MODE_1_NORMAL;
        traffic_state = INIT;
        turn_off_all_leds();
        return;
    }

    // ... (handle MODIFY and SET buttons)
}
```

5.7 Exercise 6: Adding code for displaying modes

Your tasks in this exercise are:

- To add code for display mode on seven-segment LEDs, and
- To add code for blinking LEDs depending on the mode that is selected.

5.7.1 Displaying the Mode on Seven-Segment LEDs

(Implemented in `7segment_display.c`**, function** `update_7seg_display()`**)**

This function displays the current mode number (MODE 1–4) on two seven-segment LEDs, synchronized with the traffic lights in both directions to make system status easily observable.
Specifically:

- **MODE 1**: displays “01” – normal operation mode.
- **MODE 2–4**: display “02”, “03”, and “04” respectively when adjusting the timing of the **RED**, **YELLOW**, and **GREEN** lights.

`7segment_display.c`

```c
/*
 * seven_segment.c
 * BCD (Binary Coded Decimal) 7-segment LED display module
 */
#include "7segment_display.h"
/**
 * update_7seg_display() - Update all 7-segment displays
 *
 * Display logic:
 * - MODE 1: Show countdown timers for both roads
 * - MODE 2-4: Show temp_duration being adjusted
 */
void update_7seg_display(void)
{
    // Normal operation mode
    if (current_mode == MODE_1_NORMAL)
    {
        display_7seg_left(counter_road1);  // Left: road 1 timer
        display_7seg_right(counter_road2); // Right: road 2 timer
        display_7seg_mode(1);             // Display mode number = 1
    }

    // Adjustment modes (2, 3, 4)
    else
    {
        // Both displays show the value being adjusted
        display_7seg_left(temp_duration);  // Left: temp value
        display_7seg_right(temp_duration); // Right: temp value
        display_7seg_mode(current_mode);   // Display current mode
    }
}

/* ==========================================================
 * BCD 7-SEGMENT DISPLAY FUNCTIONS
 * ========================================================== */
/**
 * Display 2-digit number on LEFT 7-segment pair
 *
 * Mechanism:
 * - Split number into tens and units
 * - Encode each digit into 4-bit BCD
 * - Output to corresponding GPIO pins
 */
void display_7seg_left(int num)
{
    // Split into 2 digits
    int tens = num / 10;  // Tens digit (e.g., 45 -> 4)
    int units = num % 10; // Units digit (e.g., 45 -> 5)

    // SEG0 - Display TENS digit using PA12-PA15
    HAL_GPIO_WritePin(GPIOA, inputseg0_0_Pin, (tens & 0x01) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 0 (LSB)
    HAL_GPIO_WritePin(GPIOA, inputseg0_1_Pin, (tens & 0x02) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 1
    HAL_GPIO_WritePin(GPIOA, inputseg0_2_Pin, (tens & 0x04) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 2
    HAL_GPIO_WritePin(GPIOA, inputseg0_3_Pin, (tens & 0x08) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 3 (MSB)
    // SEG1 - Display UNITS digit using PB0-PB3
    HAL_GPIO_WritePin(GPIOB, inputseg1_0_Pin, (units & 0x01) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 0
    HAL_GPIO_WritePin(GPIOB, inputseg1_1_Pin, (units & 0x02) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 1
    HAL_GPIO_WritePin(GPIOB, inputseg1_2_Pin, (units & 0x04) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 2
    HAL_GPIO_WritePin(GPIOB, inputseg1_3_Pin, (units & 0x08) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 3
}

// Display 2-digit number on RIGHT 7-segment pair
void display_7seg_right(int num)
{
    // Split into 2 digits
    int tens = num / 10;  // Tens digit
    int units = num % 10; // Units digit
    // SEG2 - Display TENS digit using PB4-PB7
    HAL_GPIO_WritePin(GPIOB, inputseg2_0_Pin, (tens & 0x01) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg2_1_Pin, (tens & 0x02) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg2_2_Pin, (tens & 0x04) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg2_3_Pin, (tens & 0x08) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    // SEG3 - Display UNITS digit using PB8-PB11
    HAL_GPIO_WritePin(GPIOB, inputseg3_0_Pin, (units & 0x01) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg3_1_Pin, (units & 0x02) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg3_2_Pin, (units & 0x04) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, inputseg3_3_Pin, (units & 0x08) ?
       GPIO_PIN_SET : GPIO_PIN_RESET);
}
```

```c
/**
 * display_7seg_mode() - Display current MODE number
 * Shows current system operation mode:
 * - Mode 1: Normal operation
 * - Mode 2: RED adjustment
 * - Mode 3: AMBER adjustment
 * - Mode 4: GREEN adjustment
 */
void display_7seg_mode(int mode)
{
    // Display mode using PB12-PB15
    HAL_GPIO_WritePin(GPIOB, inputmode_0_Pin, (mode & 0x01) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 0
    HAL_GPIO_WritePin(GPIOB, inputmode_1_Pin, (mode & 0x02) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 1
    HAL_GPIO_WritePin(GPIOB, inputmode_2_Pin, (mode & 0x04) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 2
    HAL_GPIO_WritePin(GPIOB, inputmode_3_Pin, (mode & 0x08) ?
       GPIO_PIN_SET : GPIO_PIN_RESET); // Bit 3
}
```

5.7.2 Controlling and Blinking Individual LEDs by Mode

(Implemented in `led_display.c`**, functions** `update_leds_display()` **and** `handle_led_blinking()`**)**

The system uses individual LEDs to indicate the traffic light status for each direction and to reflect the current operation mode.

- In **MODE 1**, the LEDs operate according to the normal traffic light cycle.
- In **MODE 2, 3, and 4**, the corresponding LEDs (**RED**, **YELLOW**, **GREEN**) **blink periodically** to indicate the mode being adjusted.

The blinking effect is controlled by a 1Hz timer interrupt (500ms ON – 500ms OFF), providing a clear, intuitive, and synchronized visual indication of the current mode.

`led_display.c`

```c
/*
 * led_display.c
 * LED control module for traffic light system
 * Controls 6 LEDs: RED1, YELLOW1, GREEN1, RED2, YELLOW2, GREEN2
 */
```

```c
#include "led_display.h"
/* ==========================================================
 * LED DISPLAY UPDATE
 * ==========================================================
 */
/**
 * update_led_display() - Update LED display based on current mode
 *
 * Logic:
 * - MODE_1_NORMAL: Display traffic lights based on traffic_state
 * - MODE 2/3/4: Display blinking LEDs based on flags
 */
void update_led_display(void)
{
    // Normal operation mode
    if (current_mode == MODE_1_NORMAL)
    {
        // Display traffic lights based on FSM state
        switch (traffic_state)
        {
        case INIT:
            turn_off_all_leds();
            break;

        case RED_GREEN:      // Road 1: RED, Road 2: GREEN
            set_traffic_led(0, 1, 0, 0); // Road 1: only red
            set_traffic_led(1, 0, 0, 1); // Road 2: only green
            break;

        case RED_AMBER:      // Road 1: RED, Road 2: AMBER
            set_traffic_led(0, 1, 0, 0); // Road 1: only red
            set_traffic_led(1, 0, 1, 0); // Road 2: only amber
            break;

        case GREEN_RED:      // Road 1: GREEN, Road 2: RED
            set_traffic_led(0, 0, 0, 1); // Road 1: only green
```

```c
                set_traffic_led(1, 1, 0, 0); // Road 2: only red
                break;

            case AMBER_RED:      // Road 1: AMBER, Road 2: RED
                set_traffic_led(0, 0, 1, 0); // Road 1: only amber
                set_traffic_led(1, 1, 0, 0); // Road 2: only red
                break;
            }
    }
    // Adjustment modes (MODE 2/3/4)
    else
    {
    // Display LEDs based on flags updated by handle_led_blinking()
        // Update red LEDs for both roads
        displayLED_RED(flagRed[0], 0);
        displayLED_RED(flagRed[1], 1);

        // Update amber LEDs for both roads
        displayLED_YELLOW(flagYellow[0], 0);
        displayLED_YELLOW(flagYellow[1], 1);

        // Update green LEDs for both roads
        displayLED_GREEN(flagGreen[0], 0);
        displayLED_GREEN(flagGreen[1], 1);
    }
}
/* =====================================================
 * BASIC LED CONTROL
 * =====================================================
 */
void turn_off_all_leds(void)
{
    // Road 1 LEDs
    HAL_GPIO_WritePin(GPIOA, RED1_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOA, YELLOW1_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOA, GREEN1_Pin, GPIO_PIN_RESET);
    // Road 2 LEDs
    HAL_GPIO_WritePin(GPIOA, RED2_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOA, YELLOW2_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOA, GREEN2_Pin, GPIO_PIN_RESET);
}

/**
 * Set traffic light state for one road
 * @param road: Road index (0 = Road 1, 1 = Road 2)
 * @param red: Red light state (1 = on, 0 = off)
 * @param amber: Amber light state (1 = on, 0 = off)
 * @param green: Green light state (1 = on, 0 = off)
 */
void set_traffic_led(int road, int red, int amber, int green)
{
    if (road == 0)
    { // Road 1
        HAL_GPIO_WritePin(GPIOA, RED1_Pin,
                        red ? GPIO_PIN_RESET : GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOA, YELLOW1_Pin,
                        amber ? GPIO_PIN_RESET : GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOA, GREEN1_Pin,
                        green ? GPIO_PIN_RESET : GPIO_PIN_SET);
    }
    else
    { // Road 2
        HAL_GPIO_WritePin(GPIOA, RED2_Pin,
                        red ? GPIO_PIN_RESET : GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOA, YELLOW2_Pin,
                        amber ? GPIO_PIN_RESET : GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOA, GREEN2_Pin,
                        green ? GPIO_PIN_RESET : GPIO_PIN_SET);
    }
}


```

```c
/**
 * displayLED_RED() - Control RED LED for specific road
 * @param IS_ON: Desired state (1 = on, 0 = off)
 * @param index: Road index (0 = Road 1, 1 = Road 2)
 */
void displayLED_RED(int IS_ON, int index)
{
    switch (index)
    {
    case 0: // Road 1
        HAL_GPIO_WritePin(GPIOA, RED1_Pin,
                          IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    case 1: // Road 2
        HAL_GPIO_WritePin(GPIOA, RED2_Pin,
                          IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    }
}
/**
 * displayLED_YELLOW() - Control AMBER LED for specific road
 */
void displayLED_YELLOW(int IS_ON, int index)
{
    switch (index)
    {
    case 0: // Road 1
        HAL_GPIO_WritePin(GPIOA, YELLOW1_Pin,
                          IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    case 1: // Road 2
        HAL_GPIO_WritePin(GPIOA, YELLOW2_Pin,
                          IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    }
}
```

```c
/**
 * displayLED_GREEN() - Control GREEN LED for specific road
 */
void displayLED_GREEN(int IS_ON, int index)
{
    switch (index)
    {
    case 0: // Road 1
        HAL_GPIO_WritePin(GPIOA, GREEN1_Pin,
                        IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    case 1: // Road 2
        HAL_GPIO_WritePin(GPIOA, GREEN2_Pin,
                        IS_ON ? GPIO_PIN_SET : GPIO_PIN_RESET);
        break;
    }
}

/* ====================================================
 * LED BLINKING - 2Hz = 500ms ON/OFF CYCLE
 * ====================================================
 */
/**
 * handle_led_blinking() - Handle LED blinking effect in adjustment
   mode
 * @param led_type: LED type to blink
 *                  0 = RED (Mode 2)
 *                  1 = AMBER (Mode 3)
 *                  2 = GREEN (Mode 4)
 */
void handle_led_blinking(int led_type)
{
    // Increment blink counter
    blink_counter++;


    // Check if enough time has passed (50 x 10ms = 500ms)
    if (blink_counter >= MAX_BLINK_COUNTER)
    {
        // Reset counter for next cycle
        blink_counter = 0;
        // Toggle blink flag: 0 - 1 or 1 - 0
        flag_blink = !flag_blink;

        // Step 1: Turn off all LEDs
        flagRed[0] = 1;
        flagRed[1] = 1;
        flagGreen[0] = 1;
        flagGreen[1] = 1;
        flagYellow[0] = 1;
        flagYellow[1] = 1;
        // Step 2: Enable only the LED being adjusted
        switch (led_type)
        {
        case 0: // MODE 2: RED duration adjustment
            // Only red LEDs on BOTH roads blink simultaneously
            flagRed[0] = flag_blink;
            flagRed[1] = flag_blink;
            break;
        case 1: // MODE 3: AMBER duration adjustment
            // Only amber LEDs on BOTH roads blink simultaneously
            flagYellow[0] = flag_blink;
            flagYellow[1] = flag_blink;
            break;
        case 2: // MODE 4: GREEN duration adjustment
            // Only green LEDs on BOTH roads blink simultaneously
            flagGreen[0] = flag_blink;
            flagGreen[1] = flag_blink;
            break;
        }
    }
}
```

5.8 Exercise 7: Adding code for increasing time duration value for the red LEDs

Your tasks in this exercise are:

- to use the second button to increase the time duration value of the red LEDs
- to use the third button to set the value for the red LEDs.

[Report Solution]:

```c
/* ============================================================
 * FSM MODE 2 - RED DURATION ADJUSTMENT
 * ============================================================ */
/**
 * fsm_red_modify_mode() - Adjust RED light duration
 *
 * - Blinks RED LEDs
 * - MODE button: Switch to MODE 3 (AMBER adjust)
 * - MODIFY button: Increase temp_duration (1 - 99 - 1)
 * - SET button: Save and auto-adjust other durations
 */
void fsm_red_modify_mode(void)
{
    // MODE button 1 - switch to AMBER adjust
    if (currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE)
    {
        current_mode = MODE_3_AMBER_MODIFY;
        temp_duration = duration_AMBER;
        return;
    }
    // MODIFY button 2 - increase value
    if (currState[1] == BTN_PRESS && prevState[1] == BTN_RELEASE)
    {
        temp_duration++;
        if (temp_duration > 99)
        {
            temp_duration = 1;
        }
    }
    // SET button 3 - save and auto-adjust
    if (currState[2] == BTN_PRESS && prevState[2] == BTN_RELEASE)
    {
        duration_RED = temp_duration;
        // Auto-adjust other durations
        auto_adjust_duration(0); // 0 = RED was modified
        current_mode = MODE_1_NORMAL;
        traffic_state = INIT;
        turn_off_all_leds();    // Turn off all LEDs
        return;
    }

    // Blink RED LEDs
    handle_led_blinking(0); // 0 = RED
}
```

5.9 Exercise 8: Adding code for increasing time duration value for the amber LEDs

Your tasks in this exercise are:

- to use the second button to increase the time duration value of the amber LEDs
- to use the third button to set the value for the amber LEDs.

[Report Solution]:

```c
/* ===========================================================
 * FSM MODE 3 - AMBER DURATION ADJUSTMENT
 * =========================================================== */

/**
 * fsm_amber_modify_mode() - Adjust AMBER light duration
 *
 * - Blinks AMBER LEDs
 * - MODE button: Switch to MODE 4 (GREEN adjust)
 * - MODIFY button: Increase temp_duration
 * - SET button: Save and auto-adjust
 */
void fsm_amber_modify_mode(void)
{

    // MODE button 1 - switch to GREEN adjust
    if (currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE)
    {
        current_mode = MODE_4_GREEN_MODIFY;
        temp_duration = duration_GREEN;
        return;
    }

    // MODIFY button 2 - increase value
    if (currState[1] == BTN_PRESS && prevState[1] == BTN_RELEASE)
    {
        temp_duration++;
        if (temp_duration > 99)
            temp_duration = 1;
    }
    // SET button 3 - save and auto-adjust
    if (currState[2] == BTN_PRESS && prevState[2] == BTN_RELEASE)
    {
        duration_AMBER = temp_duration;
        // Auto-adjust other durations
        auto_adjust_duration(1); // 1 = AMBER was modified
        current_mode = MODE_1_NORMAL;   // Return to normal mode
        traffic_state = INIT;
        turn_off_all_leds();      // Turn off all LEDs
        return;
    }

    // Blink AMBER LEDs
    handle_led_blinking(1); // 1 = AMBER
}
```

5.10 Exercise 9: Adding code for increasing time duration value for the green LEDs

Your tasks in this exercise are:

- to use the second button to increase the time duration value of the green LEDs
- to use the third button to set the value for the green LEDs.

[Report Solution]:

```c
/* ==============================================================
 * FSM MODE 4 - GREEN DURATION ADJUSTMENT
 * ==============================================================
   */
/**
 * fsm_green_modify_mode() - Adjust GREEN light duration
 *
 * - Blinks GREEN LEDs
 * - MODE button: Return to MODE 1 (no save)
 * - MODIFY button: Increase temp_duration
 * - SET button: Save and auto-adjust
 */
void fsm_green_modify_mode(void)
{
    // MODE button 1 - return to auto mode (no save)
    if (currState[0] == BTN_PRESS && prevState[0] == BTN_RELEASE)
    {
        current_mode = MODE_1_NORMAL;
        traffic_state = INIT;
        turn_off_all_leds();
        return;
    }

    // MODIFY button 2 - increase value
    if (currState[1] == BTN_PRESS && prevState[1] == BTN_RELEASE)
    {
        temp_duration++;
        if (temp_duration > 99)
            temp_duration = 1;
    }

    // SET button 3 - save and auto-adjust
    if (currState[2] == BTN_PRESS && prevState[2] == BTN_RELEASE)
    {
        duration_GREEN = temp_duration;
        // Auto-adjust other durations
        auto_adjust_duration(2); // 2 = GREEN was modified
        current_mode = MODE_1_NORMAL;
        traffic_state = INIT;
        turn_off_all_leds();         // Turn off all LEDs
        return;
    }

    // Blink GREEN LEDs
    handle_led_blinking(2); // 2 = GREEN
}
```

*** Automatic Duration Adjustment**

To maintain balanced and synchronized traffic light cycles, the system includes an automatic adjustment mechanism implemented in the function `auto_adjust_duration()`. This function enforces the timing constraint:

$$\text{RED} = \text{GREEN} + \text{AMBER}$$

Whenever one light duration is modified (during MODE 2–4), the system automatically recalculates the remaining durations to preserve this relationship and ensure valid timing values. This mechanism prevents configuration errors and guarantees smooth and consistent traffic light transitions during operation.

The automatic adjustment logic is implemented as follows:

```
/* ========================================================
 * AUTO-ADJUST DURATION FUNCTION
 * ======================================================== */
/**
 * Auto-adjust other durations to maintain constraint
 * CONSTRAINT: duration_RED = duration_GREEN + duration_AMBER
 *
 * STRATEGY:
 * - Modified RED (0): Keep AMBER, calculate GREEN = RED - AMBER
 * - Modified AMBER (1): Update GREEN = AMBER + 4, calculate RED =
   GREEN + AMBER
 * - Modified GREEN (2): Keep AMBER, calculate RED = GREEN + AMBER
 *
 * PARAMETERS:
 *   modified_light: Which light was modified
 *                   0 = GREEN, 1 = AMBER, 2 = RED
 * RETURN:
 *   1: Adjusted or reset
 *   0: No adjustment needed (already valid)
 * DEFAULT ON RESET: RED=5, GREEN=3, AMBER=2
 */
int auto_adjust_duration(int modified_light)
{
    // Check if constraint is already satisfied
    if (duration_RED == (duration_GREEN + duration_AMBER))
    {
        return 0; // No adjustment needed
    }

    switch (modified_light)
    {
    case 0: // RED was modified
        // Strategy: Keep AMBER, calculate GREEN
        duration_GREEN = duration_RED - duration_AMBER;

        // Validate GREEN
        if (duration_GREEN < 1 || duration_GREEN > 99)
        {
            duration_GREEN = duration_RED - duration_AMBER;
            duration_AMBER = duration_RED - duration_GREEN;

            // Validate AMBER
            if (duration_AMBER < 1 || duration_AMBER > 99)
            {
                // Reset to defaults
                duration_RED = 5;
                duration_GREEN = 3;
                duration_AMBER = 2;
            }
        }
        break;

    case 1: // AMBER was modified
        // Strategy: GREEN = AMBER + 4, RED = GREEN + AMBER
        duration_GREEN = duration_AMBER + 4;
        duration_RED = duration_GREEN + duration_AMBER;

        // Check if RED exceeds limit
        if (duration_RED > 99)
        {
            // Adjust to fit within limits
            duration_AMBER = (99 - 3) / 2;       // = 48
            duration_GREEN = duration_AMBER + 3; // = 51
            duration_RED = 99;

            if (duration_AMBER < 1)
            {
                // Reset if invalid
                duration_RED = 5;
                duration_GREEN = 3;
                duration_AMBER = 2;
            }
        }

        // Validate GREEN
        if (duration_GREEN < 1 || duration_GREEN > 99)
        {
            duration_RED = 5;
            duration_GREEN = 3;
            duration_AMBER = 2;
        }
        break;

    case 2: // GREEN was modified
        // Strategy: Keep AMBER, calculate RED
        duration_RED = duration_GREEN + duration_AMBER;

        // Check if RED exceeds limit
        if (duration_RED > 99)
        {
            // Reduce AMBER to fit
            duration_AMBER = 99 - duration_GREEN;
            duration_RED = 99;

            // Validate AMBER
            if (duration_AMBER < 1)
            {
                // Reset if invalid
                duration_RED = 5;
                duration_GREEN = 3;
                duration_AMBER = 2;
            }
        }
        break;
    }

    return 1; // Adjustment completed
}
```

Tài liệu