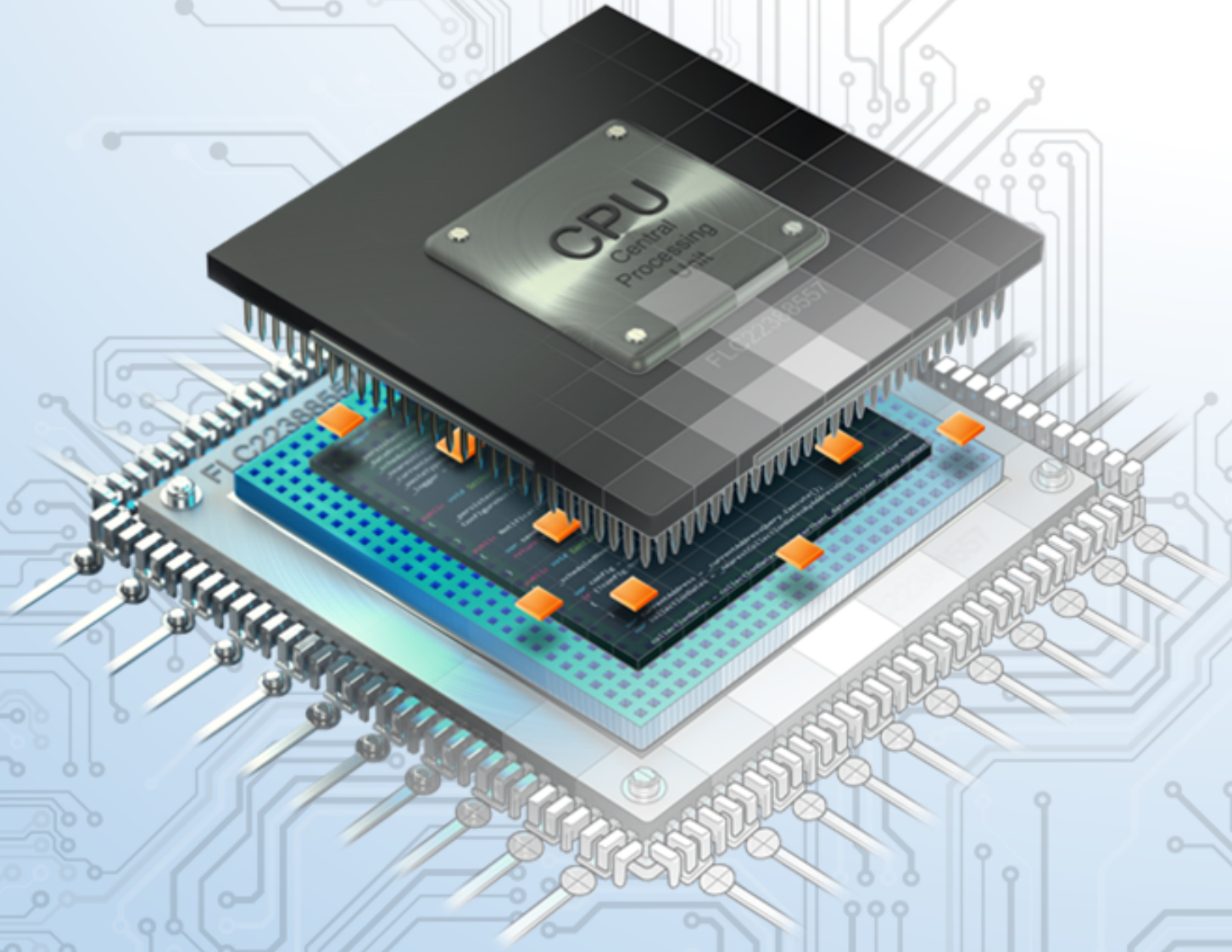
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

# Microcontroller

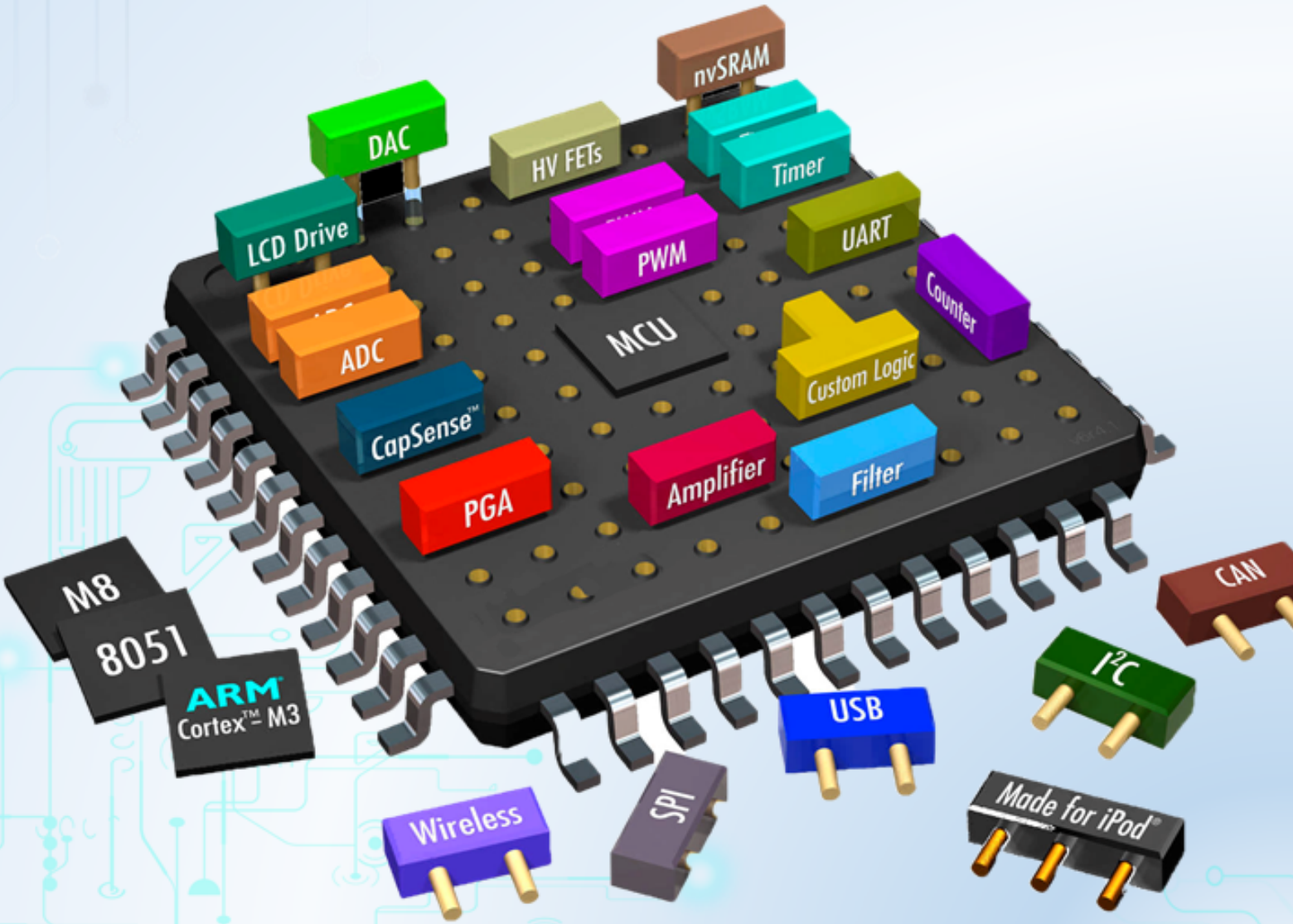



**Lecture: Nguyễn Thiên Ân**
**Student: Đỗ Huy Minh Dũng**

# Mục lục

# CHƯƠNG 1

## Buttons/Switches



# 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.

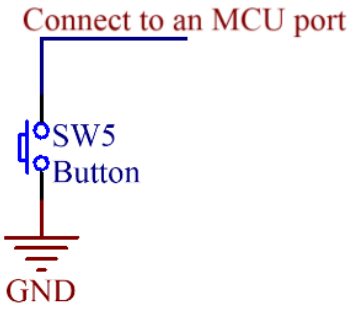
- Activate a pump while a switch is pressed

  **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




*Hình 1.1*: *Connecting a button to an MCU*

Figure 1.1 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 1.2.

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 1.3.

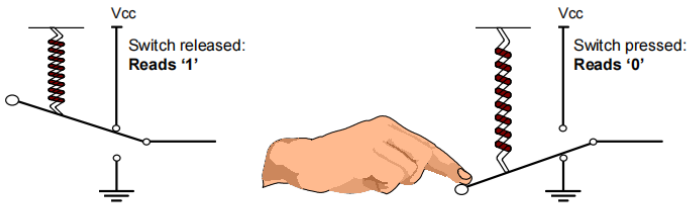
## 3.2 Dealing with switch bounces

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 1.4.

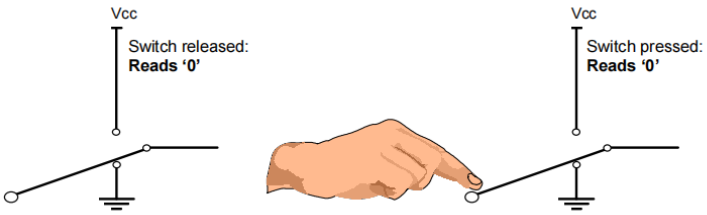
Every system that uses any kind of mechanical switch must deal with the issue of debouncing. The key task is to make sure that one mechanical switch or button action is only read as one action by the MCU, even though the MCU will typically be fast enough to detect the unwanted switch bounces and treat 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:

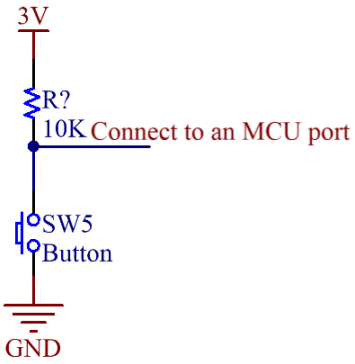
With pull-ups:



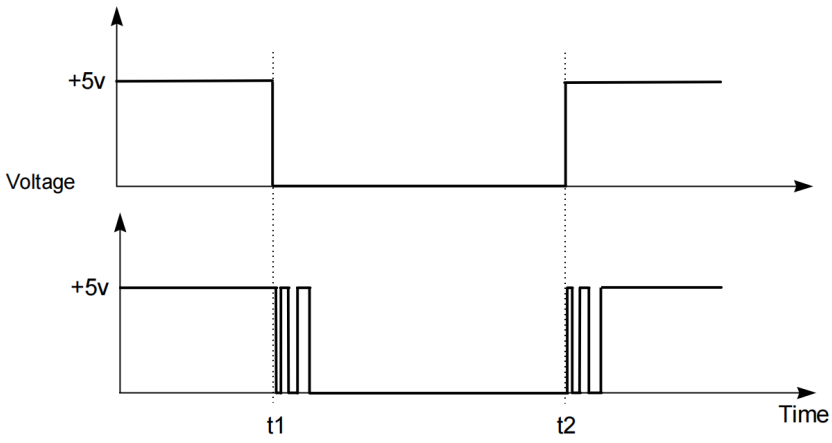

Without pull-ups:




*Hình 1.2: The need of pull up resistors*




*Hình 1.3: A reliable way to connect a button to an MCU*




*Hình 1.4: Switch bounces*

- 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:

```
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;
    }
}
```

Program 1.1: Read port pin and deboucing

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).

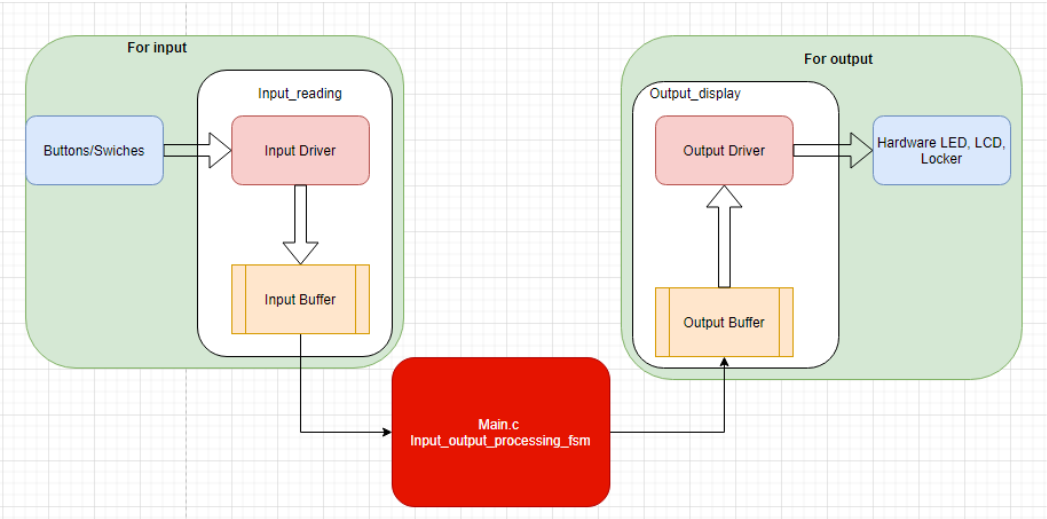
# 4 Reading switch input (basic code) using STM32

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.




*Hình 1.5*: *Input Output Processing Patterns*

Figure 1.5 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_processin* 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

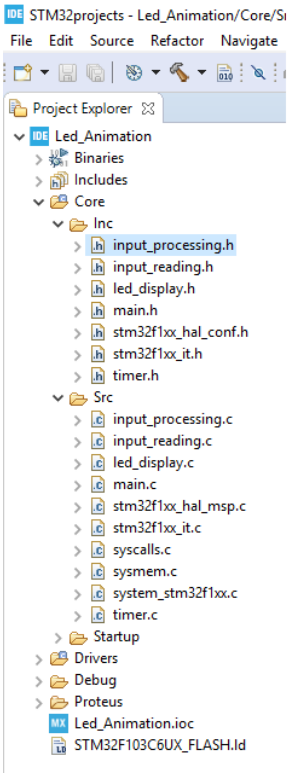
### 4.2.1 Create a project

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

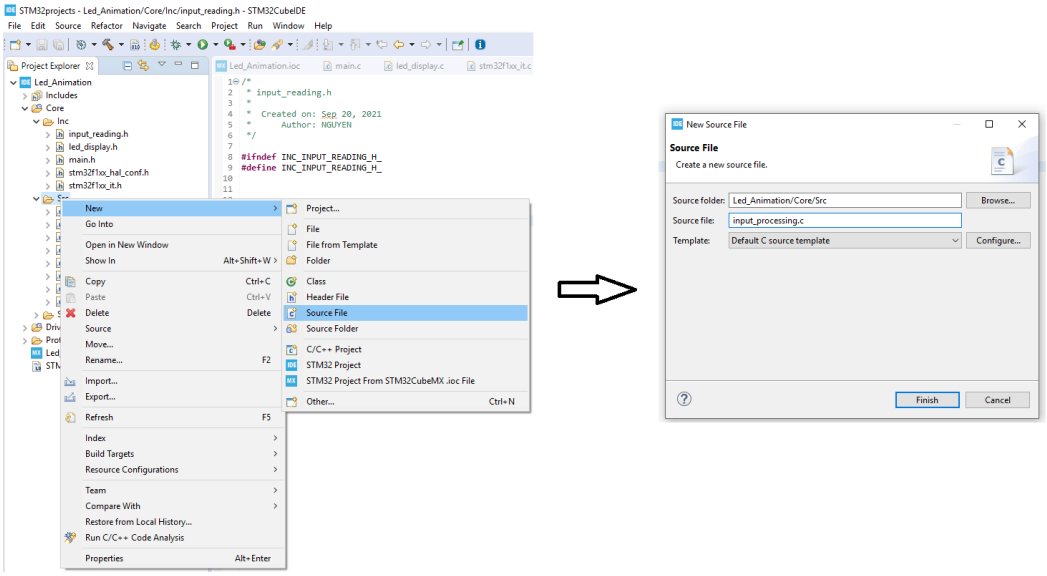
We are expected to have files for button processing and led display as shown in Figure 1.6.



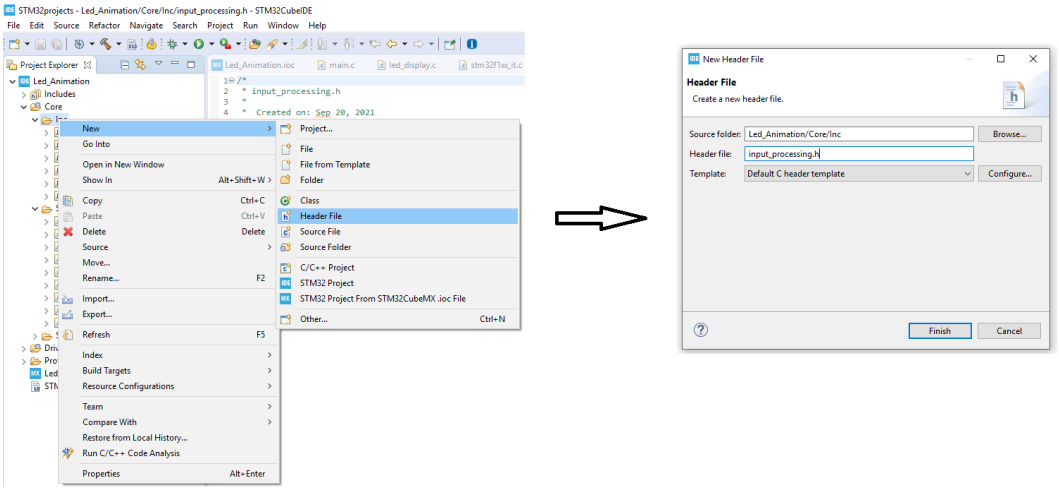

*Hình 1.6*: *File Organization*

Steps 1 (Figure 1.7): 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 1.8): Do the same for the C header file in the folder **Inc**.




*Hình 1.7: Step 1: Create a C source file for input reading*




*Hình 1.8: 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

```
#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;
      }
  }
}
```

Program 1.2: Define constants buffers and button_reading function

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

Program 1.3: Checking a button is pressed or not

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

Program 1.4: Checking a button is pressed more than a second or not

### 4.3.2 The code in the input_reading.h file

```c
#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_ */
```

Program 1.5: Prototype in input_reading.h file

### 4.3.3 The code in the timer.c file

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

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

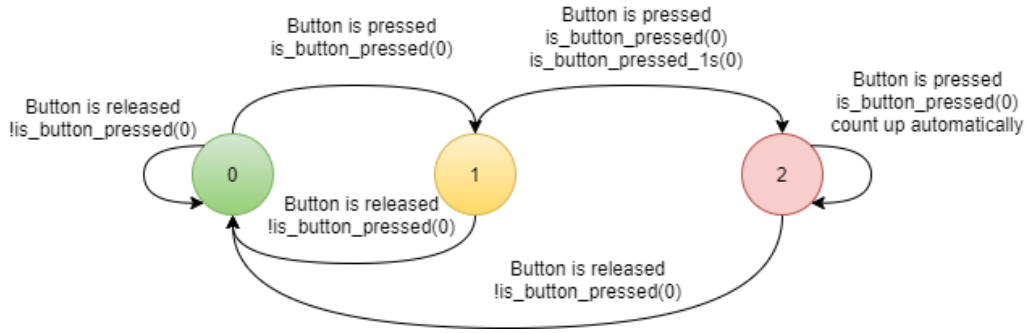
Program 1.6: Timer interrupt callback function

## 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.




*Hình 1.9: 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.

```
#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;
  }
}
```

Program 1.7: The code in the input_processing.c file

### 4.4.3 The code in the input_processing.h

```
#ifndef INC_INPUT_PROCESSING_H_
#define INC_INPUT_PROCESSING_H_

void fsm_for_input_processing(void);

#endif /* INC_INPUT_PROCESSING_H_ */
```

Program 1.8: Code in the input_processing.h file

### 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();
    }
}
```

Program 1.9: The code in the main.c file

# 5 Exercises and Report

## 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.

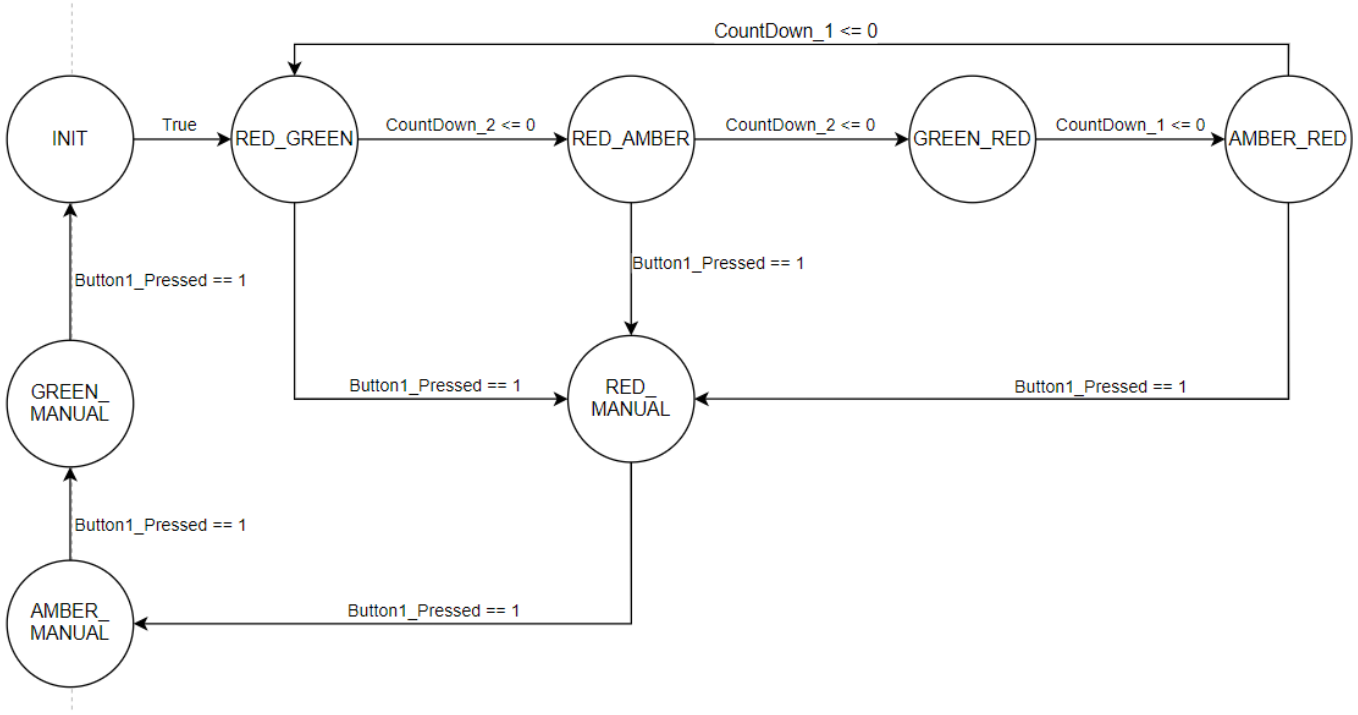
  - The third button is used to set the value.

  **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

Your task in this exercise is to sketch an FSM that describes your idea of how to solve the problem.
Please add your report here. To handle both roads at a crossroad, this FSM has 6 auto-states, including the INIT state and the remaining states represent the color state of the LEDs for both roads. This FSM has 3 manual-state for setting count down value of 3 leds.




*Hình 1.10*: Traffic Application Finite State Machine

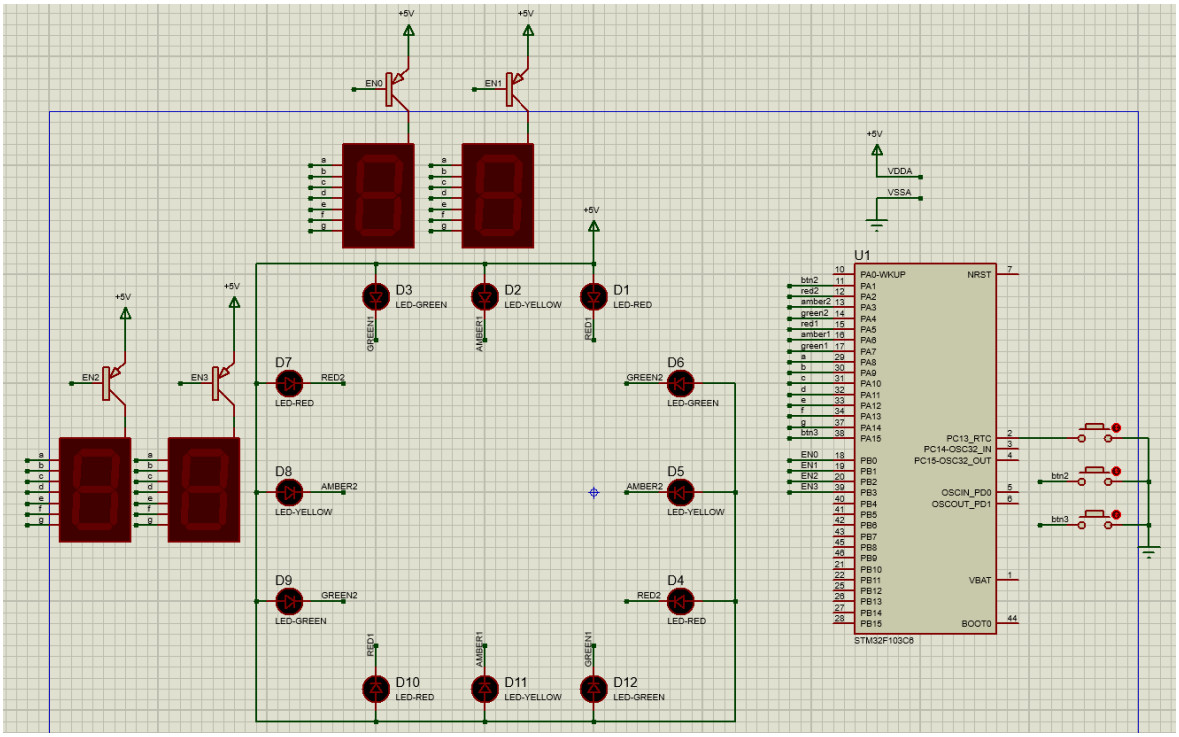
This FSM uses 3 variables to control the states of the system:

- CountDown_1 stands for the countdown value of the current state (red, amber, green) of the first road.
- CountDown_2 stands for the countdown value of the current state (red, amber, green) of the second road.
- Button1_Pressed is triggered (set to 1) when the first button is pressed to switch the system's mode.

## 5.3 Exercise 2: Proteus Schematic

Your task in this exercise is to draw a Proteus schematic for the problem above.
Please add your report here.

The schematic for simulation is presented below.
Detail about mapping pins lie on exercise 3.



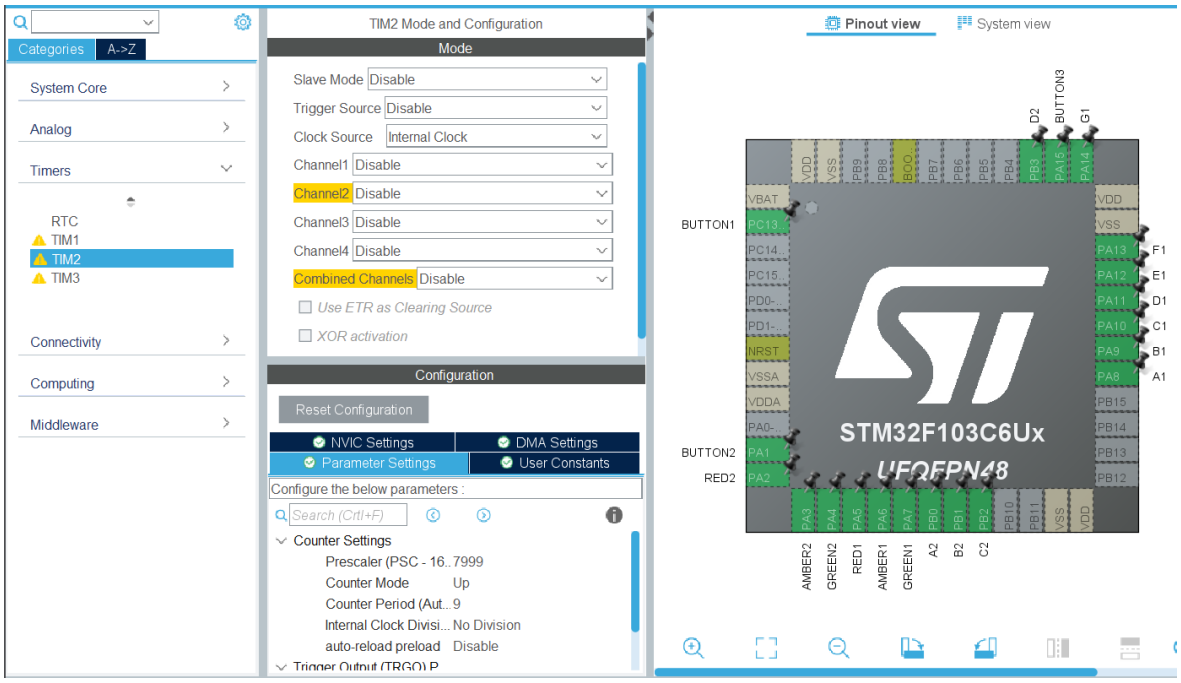

*Hình 1.11*: Proteus Schematic

## 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.
Please add your report here.

| PIN | GPIO mode | STM32 Label | Proteus Label | Behavior |
|---|---|---|---|---|
| PA1 | Input | BUTTON2 | btn2 | increase the time duration value |
| PA2 | Output | RED2 | red2 | red led of second road |
| PA3 | Output | AMBER2 | amber2 | amber led of second road |
| PA4 | Output | GREEN2 | green2 | green led of second road |
| PA5 | Output | RED1 | red1 | red led of first road |
| PA6 | Output | AMBER1 | amber1 | amber led of first road |
| PA7 | Output | GREEN1 | green1 | green led of first road |
| PA8 | Output | A1 | a | first segment of 7 segment led |
| PA9 | Output | B1 | b | second segment of 7 segment led |
| PA10 | Output | C1 | c | third segment of 7 segment led |
| PA11 | Output | D1 | d | fourth segment of 7 segment led |
| PA12 | Output | E1 | e | fifth segment of 7 segment led |
| PA13 | Output | F1 | f | sixth segment of 7 segment led |
| PA14 | Output | G1 | g | seventh segment of 7 segment led |
| PA15 | Input | BUTTON3 | btn3 | set the time duration value |
| PB0 | Output | A2 | EN0 | enable led of tens unit (first road) |
| PB1 | Output | B2 | EN1 | enable led of ones unit (first road) |
| PB2 | Output | C2 | EN2 | enable led of tens unit (second road) |
| PB3 | Output | D2 | EN3 | enable led of ones unit (second road) |
| PC13 | Input | BUTTON1 | btn1 | switching mode |

Bảng 1.1: PINs Description




*Hình 1.12*: PINs label and setting timer interrupt to 10ms

## 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.
Please add your report here.

The solution here is define a TICK that present the timer interrupt duration. We divide this the real-time interrupt to this TICK everytime call the setTimer function:

```
#define TICK 10
void setTimer(int *timer,int time){
    *timer = time / TICK;
}
```

If the original timer interrupt duration is 10ms, TICK would be defined as 10. If we change the interrupt to 1ms or 100ms, we just need to adjust calculations on TICK by redefine it to 1 or 100.

## 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.

Please add your report here.

**Add files for input reading and output display:**
This project include 5 files that help control the system:

- **softwaretimer:** Create timer interrupt signals for the system.
- **button:** Create input signals when user press buttons.
- **global:** For global variables and display seven-segment LEDs.
- **fsm_auto:** Operate the system and output signals in auto mode.
- **fsm_manual:** Operate the system and output signals in manual mode.

**Code for button debouncing:**

```
/////////////// BUTTON_1 FOR SELECT MODE /////////////////
int reg0 = NORMAL_STATE;
int reg1 = NORMAL_STATE;
int reg2 = NORMAL_STATE;
int reg3 = NORMAL_STATE;
int timerForPress = 2000;
int btn_flag1 = 0;
int isButton1_pressed(){
  if(btn_flag1 == 1){
    btn_flag1 = 0;
    return 1;
  }
  return 0;
}
void getKeyInput1(){
  reg0 = reg1;
  reg1 = reg2;
  reg2 = HAL_GPIO_ReadPin(BUTTON1_GPIO_Port, BUTTON1_Pin);
  if(reg0 == reg1 && reg1 == reg2){
    if(reg2 != reg3){
      reg3 = reg2;//for check the condition if user hold
   button
      if(reg3 == PRESSED_STATE){
        timerForPress = HOLD_DELAY / TICK;
        btn_flag1 = 1;
      }
    }
    else{
      --timerForPress;
      if(timerForPress == 0){
        timerForPress = HOLD_DELAY / TICK;
        if(reg3 == PRESSED_STATE)
          btn_flag1 = 1;
      }
    }
  }
}
////////////// BUTTON_2 FOR MODIFY TIMING /////////////////
#define TIMER_HOLD 100
int btn2_reg0 = NORMAL_STATE;
int btn2_reg1 = NORMAL_STATE;
int btn2_reg2 = NORMAL_STATE;
int btn2_reg3 = NORMAL_STATE;
int timerForPress2 = 50;
int btn_flag2 = 0;
int isButton2_pressed(){
  if(btn_flag2){
    btn_flag2 = 0;
    return 1;
```

```
  }
  return 0;
}
void getKeyInput2(){
  btn2_reg0 = btn2_reg1;
  btn2_reg1 = btn2_reg2;
  btn2_reg2 = HAL_GPIO_ReadPin(BUTTON2_GPIO_Port,
   BUTTON2_Pin);
  if(btn2_reg0 == btn2_reg1 && btn2_reg1 == btn2_reg2){
    if(btn2_reg2 != btn2_reg3){
      btn2_reg3 = btn2_reg2;
      if(btn2_reg3 == PRESSED_STATE){
        timerForPress2 = HOLD_DELAY / TICK;
        btn_flag2 = 1;
      }
    }
    else{
      --timerForPress2;
      if(timerForPress2 <= 0){
        timerForPress2 = TIMER_HOLD / TICK;
        if(btn2_reg3 == PRESSED_STATE)
          btn_flag2 = 1;
      }
    }
  }
}
////////////////// BUTTON_3 FOR SET VALUE /////////////////
int btn3_reg0 = NORMAL_STATE;
int btn3_reg1 = NORMAL_STATE;
int btn3_reg2 = NORMAL_STATE;
int btn3_reg3 = NORMAL_STATE;
int timerForPress3 = 200;
int btn_flag3 = 0;
int isButton3_pressed(){
  if(btn_flag3){
    btn_flag3 = 0;
    return 1;
  }
  return 0;
}
void getKeyInput3(){
  btn3_reg0 = btn3_reg1;
  btn3_reg1 = btn3_reg2;
  btn3_reg2 = HAL_GPIO_ReadPin(BUTTON3_GPIO_Port,
   BUTTON3_Pin);
  if(btn3_reg0 == btn3_reg1 && btn3_reg1 == btn3_reg2){
    if(btn3_reg2 != btn3_reg3){
      btn3_reg3 = btn3_reg2;
      if(btn3_reg3 == PRESSED_STATE){
```

```
96          timerForPress3 = HOLD_DELAY / TICK;
97          btn_flag3 = 1;
98        }
99      }
100     else{
101       --timerForPress3;
102       if(timerForPress3 <= 0){
103         timerForPress3 = HOLD_DELAY / TICK;
104         if(btn3_reg3 == PRESSED_STATE)
105           btn_flag3 = 1;
106       }
107     }
108   }
109 }
```

Debouncing the button press is done by adding multiple check variables and ensuring they are all equal to PRESS_STATE before setting the button flag to 1. The first and third buttons have a 2-second delay before setting the button flag to 1 each time the user holds (long presses) the button, while the second button only delays for the first time if the user wants to increase the countdown value faster.

**Code for increasing mode when the first button is pressed:**
Lies on 2 function fsm_autoRun() and fsm_manualRun() (from fsm_auto.c and fsm_manual.c). Can be describe as follows, for detail please go to Github.

```
1  void fsm_autoRun(){
2      ...
3      if(automatic == 1){
4          if(isButton1_pressed()){
5              automatic = 0;
6              state = RED_MAN;
7              setTraffic1(0, 1, 1);
8              setTraffic2(0, 1, 1);
9              setTimerCD(BLINK_TIME);
10             setTimerSeg(500);
11             EN = 0;
12         }
13     }
14 }
15
16 void fsm_manualRun(){
17     switch(state){
18     case RED_MAN:
19         ...
20         if(isButton1_pressed()){
21             state = AMBER_MAN;
22             setTraffic1(1, 0, 1);
23             setTraffic2(1, 0, 1);
24             setTimerCD(BLINK_TIME);
25         }
```

```
        ...
        break;
    case AMBER_MAN:
        ...
        if(isButton1_pressed()){
            state = GREEN_MAN;
            setTraffic1(1, 1, 0);
            setTraffic2(1, 1, 0);
            setTimerCD(BLINK_TIME);
        }
        ...
        break;
    case GREEN_MAN:
        ...
        if(isButton1_pressed()){
            setBuffer2();
            state = INIT;
        }
        ...
        break;
    default: break;
    }
}
```

## 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.

Please add your report here.

**Code for display mode on seven-segment LEDs:**

```
#define LED_CYCLE 250
uint8_t EN = 0;
void helpDisplaySeg(int a, int b, int c, int d, int e, int
   f, int g){
  HAL_GPIO_WritePin(A1_GPIO_Port, A1_Pin, a);
  HAL_GPIO_WritePin(B1_GPIO_Port, B1_Pin, b);
  HAL_GPIO_WritePin(C1_GPIO_Port, C1_Pin, c);
  HAL_GPIO_WritePin(D1_GPIO_Port, D1_Pin, d);
  HAL_GPIO_WritePin(E1_GPIO_Port, E1_Pin, e);
  HAL_GPIO_WritePin(F1_GPIO_Port, F1_Pin, f);
  HAL_GPIO_WritePin(G1_GPIO_Port, G1_Pin, g);
}
void displaySeg(int num){
```

```
13    switch(num){
14    case 0:
15      helpDisplaySeg(0, 0, 0, 0, 0, 0, 1);
16      break;
17    case 1:
18      helpDisplaySeg(1, 0, 0, 1, 1, 1, 1);
19      break;
20    case 2:
21      helpDisplaySeg(0, 0, 1, 0, 0, 1, 0);
22      break;
23    case 3:
24      helpDisplaySeg(0, 0, 0, 0, 1, 1, 0);
25      break;
26    case 4:
27      helpDisplaySeg(1, 0, 0, 1, 1, 0, 0);
28      break;
29    case 5:
30      helpDisplaySeg(0, 1, 0, 0, 1, 0, 0);
31      break;
32    case 6:
33      helpDisplaySeg(0, 1, 0, 0, 0, 0, 0);
34      break;
35    case 7:
36      helpDisplaySeg(0, 0, 0, 1, 1, 1, 1);
37      break;
38    case 8:
39      helpDisplaySeg(0, 0, 0, 0, 0, 0, 0);
40      break;
41    case 9:
42      helpDisplaySeg(0, 0, 0, 0, 1, 0, 0);
43      break;
44    default: break;
45    }
46  }
47
48  void displaySegProcess(int CD1, int CD2){
49    switch(EN){
50    case 0:
51      displaySeg(CD1/10);
52      HAL_GPIO_WritePin(A2_GPIO_Port, A2_Pin, 0);
53      HAL_GPIO_WritePin(B2_GPIO_Port, B2_Pin, 1);
54      HAL_GPIO_WritePin(C2_GPIO_Port, C2_Pin, 1);
55      HAL_GPIO_WritePin(D2_GPIO_Port, D2_Pin, 1);
56      break;
57    case 1:
58      displaySeg(CD1%10);
59      HAL_GPIO_WritePin(A2_GPIO_Port, A2_Pin, 1);
60      HAL_GPIO_WritePin(B2_GPIO_Port, B2_Pin, 0);
61      HAL_GPIO_WritePin(C2_GPIO_Port, C2_Pin, 1);
```

```
62      HAL_GPIO_WritePin(D2_GPIO_Port, D2_Pin, 1);
63      break;
64    case 2:
65      displaySeg(CD2/10);
66      HAL_GPIO_WritePin(A2_GPIO_Port, A2_Pin, 1);
67      HAL_GPIO_WritePin(B2_GPIO_Port, B2_Pin, 1);
68      HAL_GPIO_WritePin(C2_GPIO_Port, C2_Pin, 0);
69      HAL_GPIO_WritePin(D2_GPIO_Port, D2_Pin, 1);
70      break;
71    case 3:
72      displaySeg(CD2%10);
73      HAL_GPIO_WritePin(A2_GPIO_Port, A2_Pin, 1);
74      HAL_GPIO_WritePin(B2_GPIO_Port, B2_Pin, 1);
75      HAL_GPIO_WritePin(C2_GPIO_Port, C2_Pin, 1);
76      HAL_GPIO_WritePin(D2_GPIO_Port, D2_Pin, 0);
77      break;
78    default: break;
79    }
80    EN = (EN+1) % 4;
81  }
```

These function will be called in 2 fsm function when the timer flag for them is triggered. In fsm_auto (CD1 and CD2 is the coundown values of current state):

```
1  if(isTimerFlagSeg()){
2      displaySegProcess(CD1, CD2);
3      setTimerSeg(LED_CYCLE);
4  }
```

In fsm_manual (COLOR can be RED, AMBER or GREEN and X can be 1, 2 or 3):

```
1  if(isTimerFlagSeg()){
2      setTimerSeg(LED_CYCLE);
3      displaySegProcess(COLOR_TIMER, X);
4  }
```

**Code for blinking LEDs depending on the mode that is selected:**
Lies on fsm_manual() function, X1 can be RED1, AMBER1 or GREEN1 and X2 can be RED2, AMBER2 or GREEN2 depending on the mode that is selected:

```
1  if(isTimerFlagCD()){
2      HAL_GPIO_TogglePin(X1_GPIO_Port, X1_Pin);
3      HAL_GPIO_TogglePin(X2_GPIO_Port, X2_Pin);
4      setTimerCD(BLINK_TIME);
5  }
```

For detail please go to Github.

## 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.

Please add your report here.

These tasks are lie on fsm_manual() function:

```
void fsm_manualRun(){
    switch(state){
    case RED_MAN:
              ...
    if(isButton2_pressed()){
      if(++RED_TIMER >= 100) RED_TIMER = 1;
    }
    if(isButton3_pressed()){
      buffer1[RED] = RED_TIMER;
    }
    break;
  case AMBER_MAN:
              ...
    if(isButton2_pressed()){
      if(++AMBER_TIMER >= 100) AMBER_TIMER = 1;
    }
    if(isButton3_pressed()){
      buffer1[AMBER] = AMBER_TIMER;
    }
    break;
  case GREEN_MAN:
              ...
    if(isButton2_pressed()){
      if(++GREEN_TIMER >= 100) GREEN_TIMER = 1;
    }
    if(isButton3_pressed()){
      buffer1[GREEN] = GREEN_TIMER;
    }
    break;
  default: break;
  }
}
```

For detail please go to Github.

## 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.

Please add your report here.
**These tasks are presented in exercise 7 already.**

## 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.

Please add your report here.
**These tasks are presented in exercise 7 already.**

## 5.11 Exercise 10: To finish the project

Your tasks in this exercise are:

- To integrate all the previous tasks to one final project
- To create a video to show all features in the specification
- To add a report to describe your solution for each exercise.
- To submit your report and code on the BKeL

View the final project on Github.
Video record: Click here.
Other infomations about this application:
When the user adjusts the countdown values in manual mode, only the first road's countdown is adjusted according to the user's input. The countdown values in the second road will be recalculated to ensure the traffic system works with the correct traffic logic as follows:

- RED_Count_Down_2ndRoad = AMBER_Count_Down_1stRoad + GREEN_Count_Down

- AMBER_Count_Down_2ndRoad = AMBER_Count_Down_1stRoad;
- GREEN_Count_Down_2ndRoad = RED_Count_Down_1stRoad - AMBER_Count_Down_1stRoad

The duration values for the LEDs are adjusted to avoid logic errors:

- RED LED is in a range of 5 - 99.
- AMBER LED is in a range of 1 - 5.
- GREEN LED is in a range of 3 - 94.