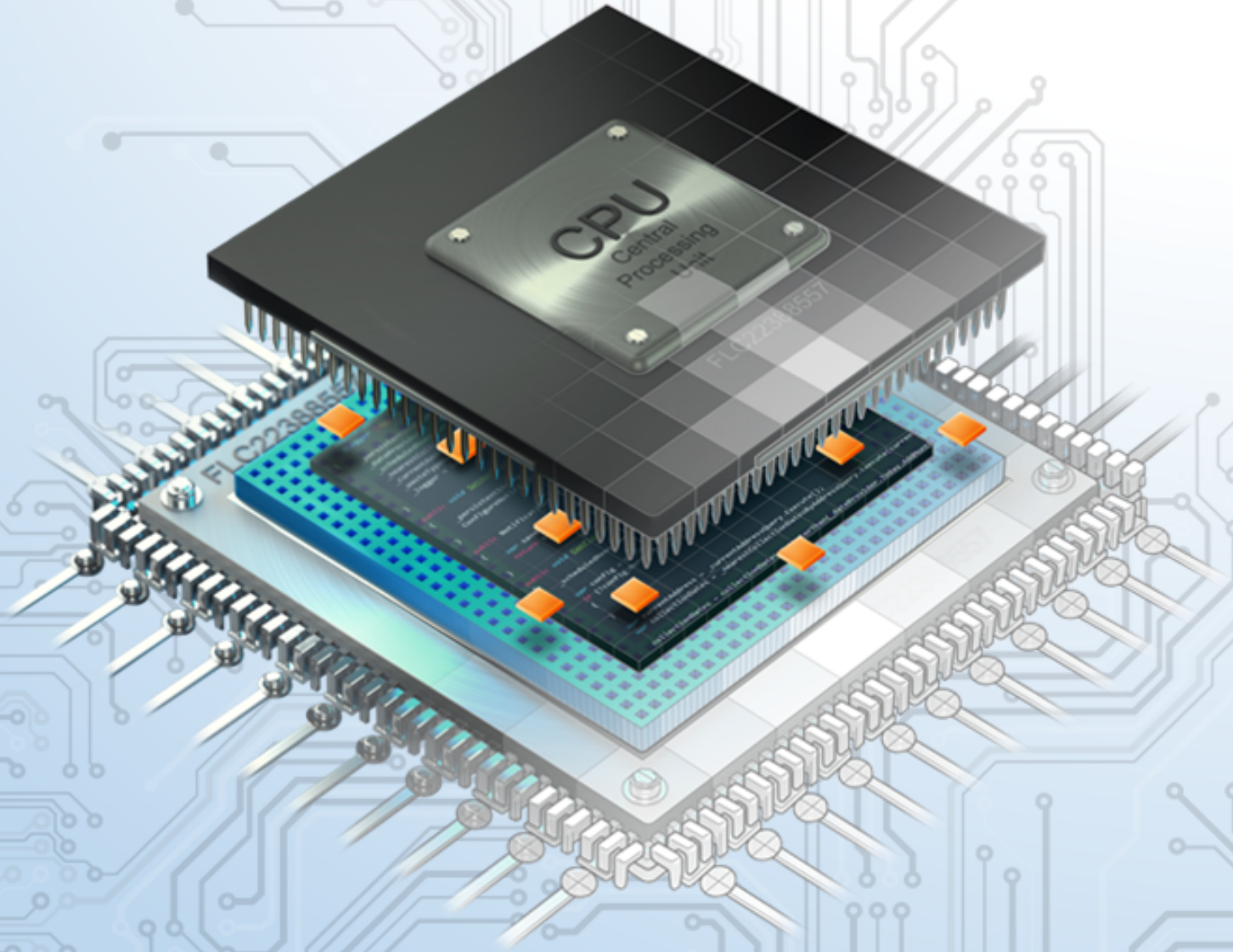
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller

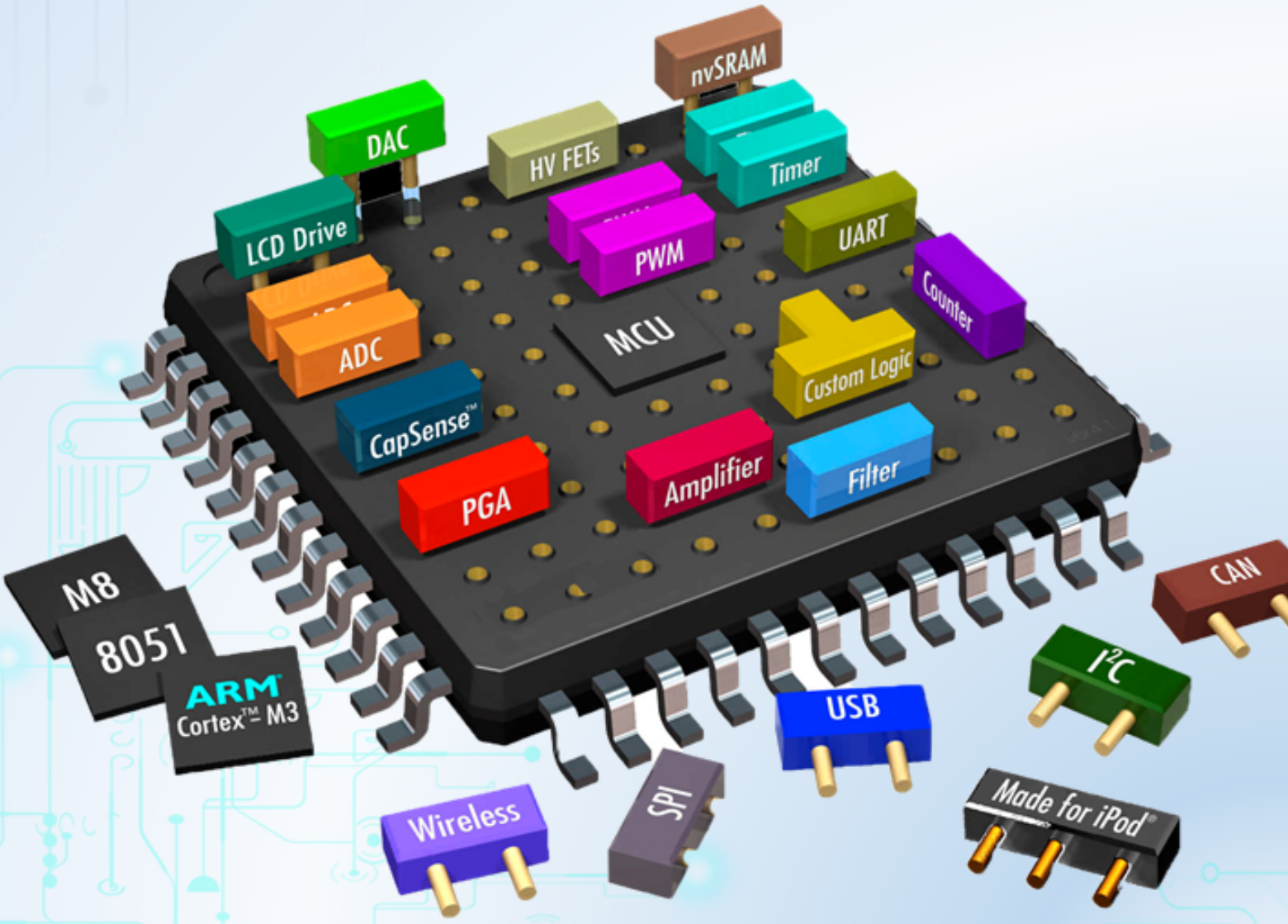


Dr. Le Trong Nhan

Contents

CHAPTER 1

Buttons/Switches



1 Objectives

In this lab, you will

- Learn how to use a timer and timer interrupt in STM32, and
- Learn how to read digital inputs and display values to LEDs using a timer

2 Introduction

Embedded systems usually use button/switches 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/switch interface.
In this lab, we consider how you read inputs from mechanical switches in your embedded application.
Before considering button/switches themselves, we will consider the process of reading the state of port pins.

3 Basic techniques for reading from port pins

3.1 The need for pull-up resistors

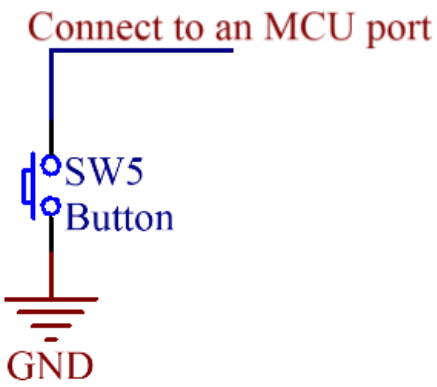


Figure 1.1: *Button/switch connects to an MCU*

Figure 1.1 shows a normal way to connect a button/switch 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 microcontroller (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’.

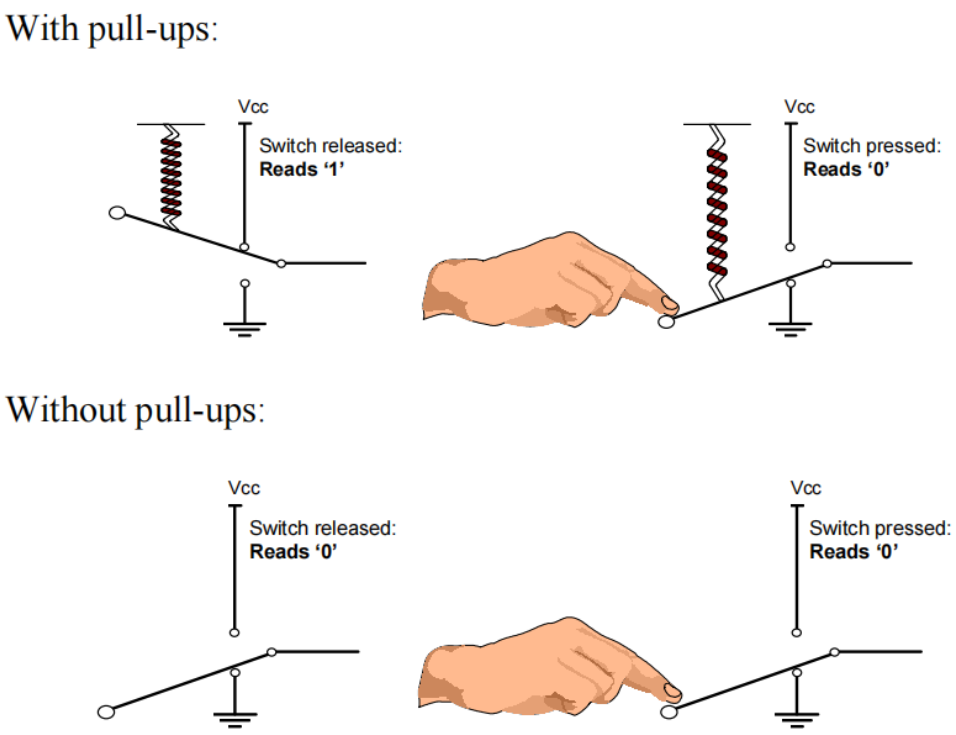


Figure 1.2: The need of pull up resistors

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.

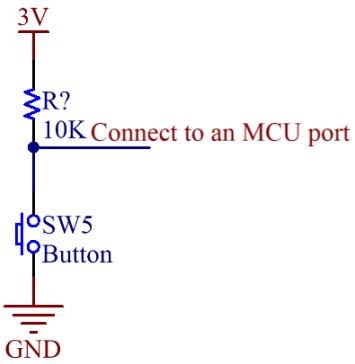


Figure 1.3: A reliable way to connect a button to an MCU

3.2 Dealing with switch bounce

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.
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).

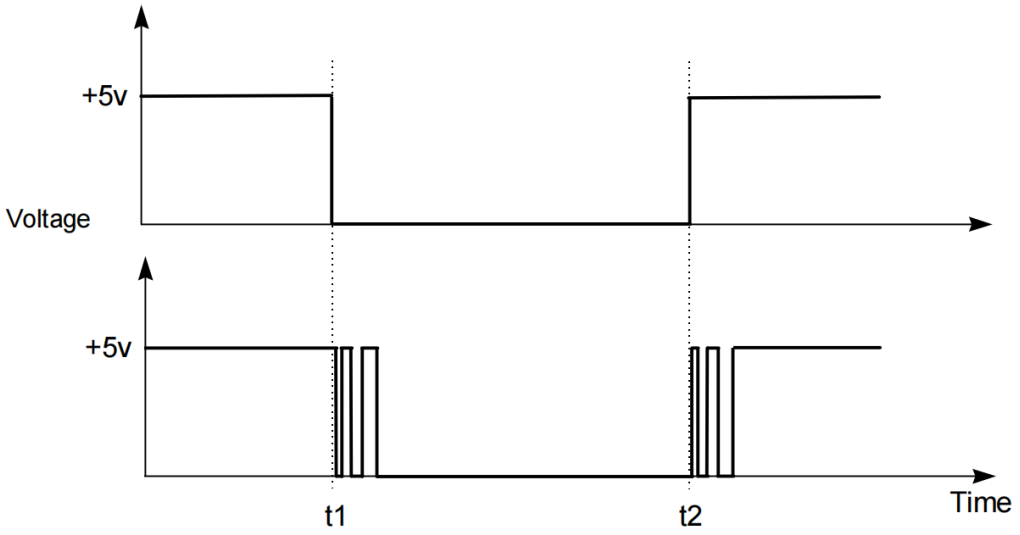


Figure 1.4: Switch bounces

Creating a simple software to check for a valid switch input is straightforward:

- Read the relevant port pin
- If we think we have detected a switch depression, we wait for 20ms and then read the pin again.
- If the second reading confirms the first reading, we assume the switch really has been depressed.

Note that the figure of '20ms' will depend on the switch used and the deployed environment.

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.

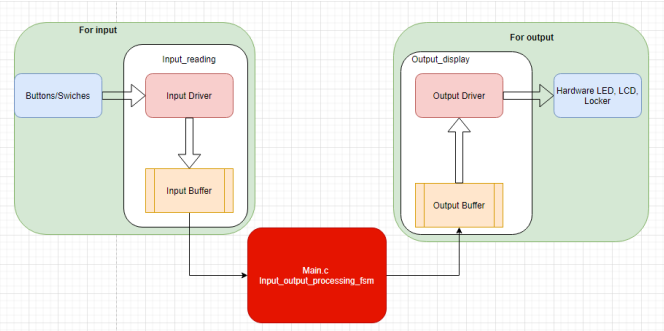


Figure 1.5: Input Output Processing Patterns

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_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 as shown in Figure 1.6,
- PA0-PA7 as output port pins as shown in Figure 1.6
- Timer 2 10ms interrupt

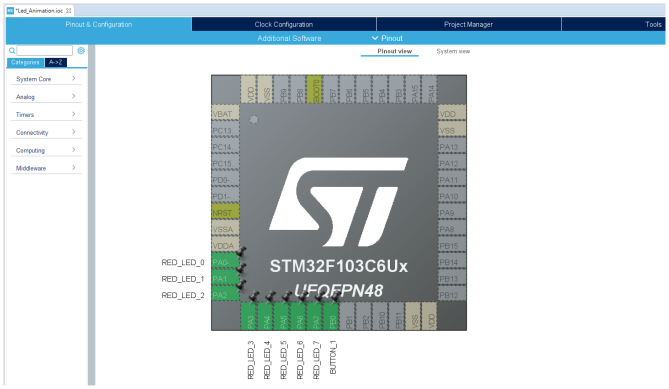


Figure 1.6: Input Output Setting

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 1.7.

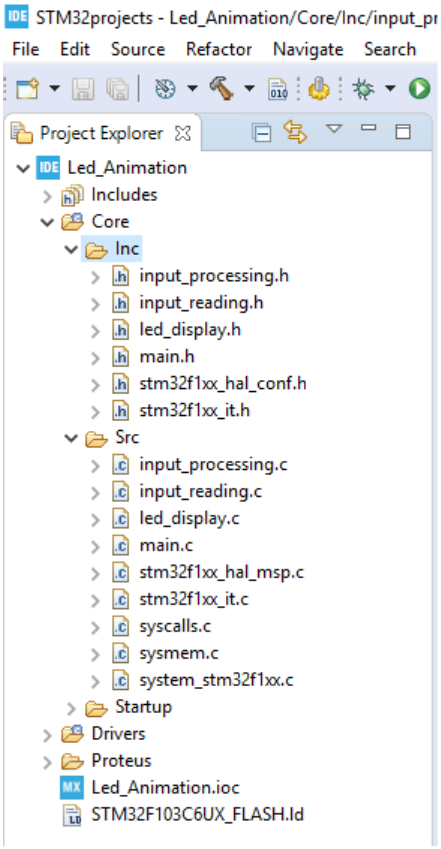


Figure 1.7: *File Organization*

Steps 1,2 (Figure 1.8): Right click to the folder **Src**, select **New**, then select **Source File**. There will be a pop-up shown in Figure 1.9. Please type the file name, then click **Finish**.
Step 3,4 (Figure 1.10, 1.11): Do the same for the C header file in the folder **Inc**.

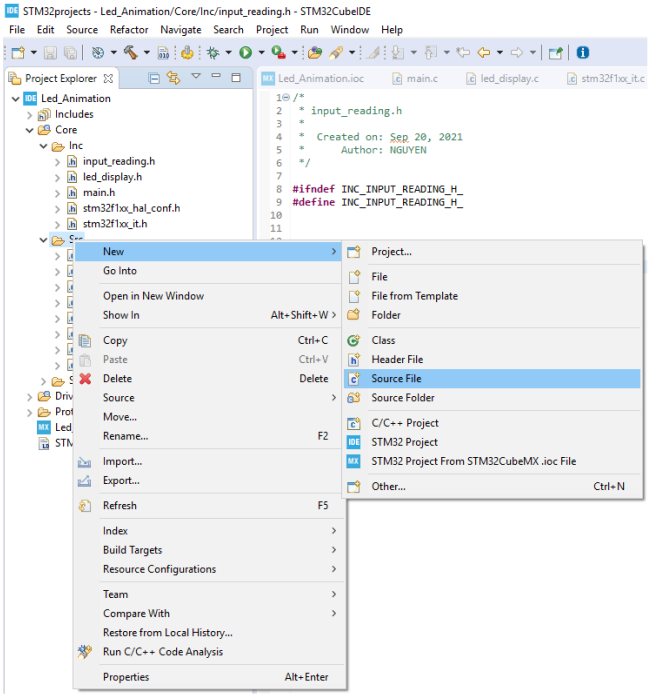


Figure 1.8: Step 1: Create a C source file for input reading

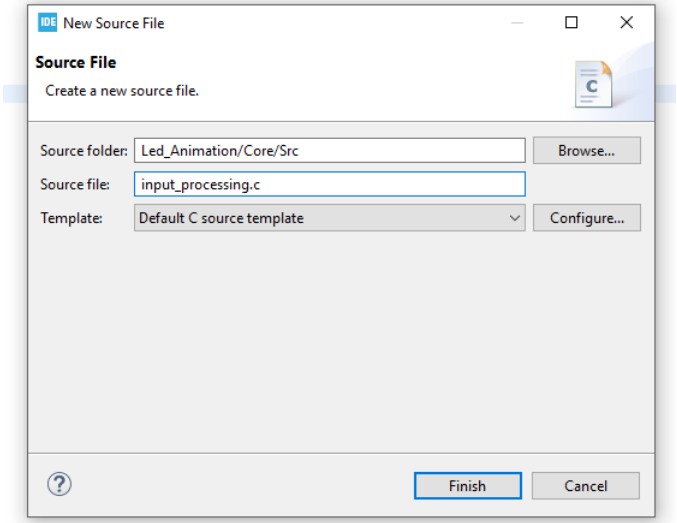


Figure 1.9: Step 2: Create a C source file for input reading

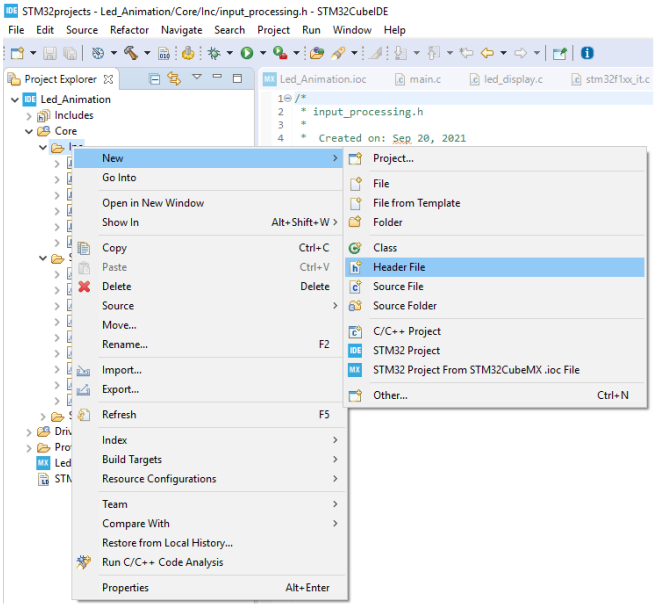


Figure 1.10: Step 3: Create a C header file for input processing

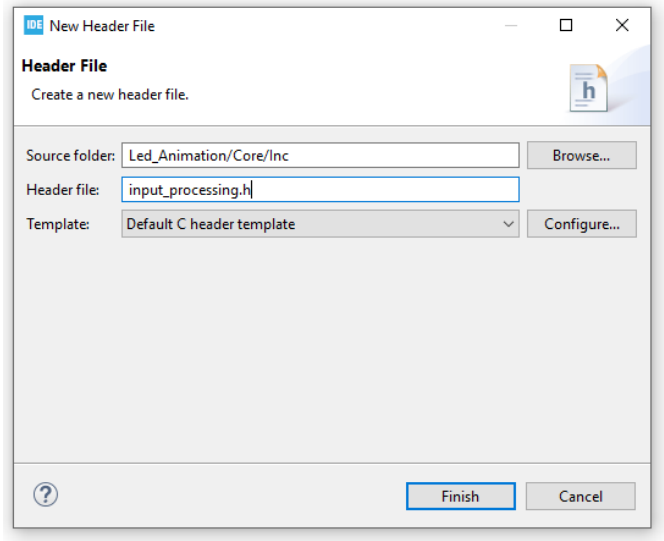


Figure 1.11: Step 4: Create a C header file for input processing

4.3 Code For Read Port Pin and Debouncing

4.3.1 The code in the input_reading.c file

```
#include "main.h"
#define NO_OF_BUTTONS          1
#define DURATION_FOR_AUTO_INCREASING  100
//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];
static uint16_t counterForButtonPress1s[NO_OF_BUTTONS];
static uint8_t flagForButtonPress1s[NO_OF_BUTTONS];
```

Program 1.1: Define buffers

The program reads all buttons two consecutive times and compare the values. If the values are the same, update the value to buttonBuffer. This function should be called inside the timer interrupt service routine.

```
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] == GPIO_PIN_SET){
        if(counterForButtonPress1s[i] <
   DURATION_FOR_AUTO_INCREASING){
          counterForButtonPress1s[i]++;
        } else {
          flagForButtonPress1s[i] = 1;
          //todo
        }

      } else {
        counterForButtonPress1s[i] = 0;
      }
  }
}
```

Program 1.2: Read port pin and debouncing

```
GPIO_PinState get_button_value(uint8_t index){
  if(index >= NO_OF_BUTTONS) return 0xff;
  return buttonBuffer[index];
}
```

Program 1.3: Get button state

```
unsigned char get_flag_for_button_press_1s(unsigned char
    index){
  if(index >= NUMBER_OF_BUTTON) return 0xff;
  return flagForButtonPress1s[index];
}
```

Program 1.4: Get state of pressing more than 1 second

4.3.2 The code in the input_reading.h file

```
#ifndef INC_INPUT_READING_H_
#define INC_INPUT_READING_H_
unsigned char get_button_value(unsigned char index);
void button_reading(void);
unsigned char get_flag_for_button_press_1s(unsigned char
    index);
#endif /* INC_INPUT_READING_H_ */
```

Program 1.5: Prototype in input_reading.h file

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 intial state.
- State 1: When the button is pressed, the FSM will change to State 1 that increasing the values of PORTA by one value.
- 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.

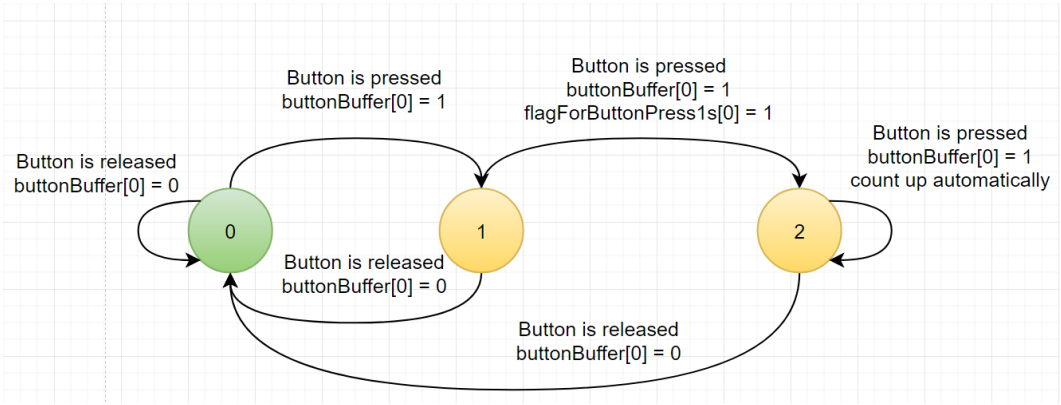


Figure 1.12: FSM for button processing

4.4.2 The code for the FSM

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(get_button_value(0) == GPIO_PIN_SET){
      buttonState = BUTTON_PRESSED;
      //INCREASE VALUE OF PORT A BY ONE UNIT
    }
    break;
  case BUTTON_PRESSED:
    if(get_button_value(0) == GPIO_PIN_RESET){
      buttonState = BUTTON_RELEASED;
    } else {
      if(get_flag_for_button_press_1s(0) == 1){
        buttonState = BUTTON_PRESSED_MORE_THAN_1_SECOND;
      }
    }
    break;
  case BUTTON_PRESSED_MORE_THAN_1_SECOND:
    if(get_button_value(0) == GPIO_PIN_RESET){
      buttonState = BUTTON_RELEASED;
    }
    //todo
    break;
  }
}
```

Program 1.6: Input processing file

5 Problem 1

In this week lab, assume you have two button and 8 LEDs, you have to write a program that

- Has a timer which has an interrupt in every 10 milliseconds.
- Reads values of buttons 1 and 2 every 10 milliseconds. The read button function should be called inside the timer interrupt service routine.
- Increases the value of LEDs connected to PORTA when the button 1 is pressed.

- Increases the value of PORTA automatically in every 0.5 second, if the button 1 is pressed in more than 1 second.
- Increases the value of PORTA automatically in every 0.1 second, if the button 1 is pressed in more than 3 seconds.
- Decreases the value of PORTA when the button 2 is pressed
- Decreases the value of PORTA automatically in every 0.5 second, if the button 2 is pressed in more than 1 second.
- Decreases the value of PORTA automatically in every 0.1 second, if the button 2 is pressed in more than 3 seconds.
- The above values such as 10 ms, 0.5 s, 0.1s, 1 s and 3 s are fixed values as an example. Your program, however, needs to provide an easy way to change those values. For example, you should use DEFINE to define those values.
- If both buttons 1 and 2 are pressed, button 1 has a higher priority.

6 Problem 2 - A digital clock

You have to write a program that mimics a Casio digital watch. Assume you have 2 buttons and 6 seven-segment LED, a program should have

- A normal clock that shows hours, minutes, and seconds on 6 seven-segment LEDs.
- A stopwatch that shows minutes, seconds and 1/100 second
- We use button 1 to change modes of a digital watch. Every time button 1 is pressed, the watch will change to the next mode.
- Mode 0: runs normal clock (default)
- Mode 1: modifies hours. In this mode, the hour number is blinking, and button 2 is used for increasing the hour number. If button 2 is kept pressed more than 1 second, the hour number will increase automatically, i.e., 5 times per second. Please note that the hour number should be returned to 0 when it reaches 23.
- Mode 2: modifies minutes. Similar to mode 1, the minute number is blinking and increases when button 2 is pressed.
- Mode 3: modifies seconds. Similar to mode 1, the second number is blinking and increases when button 2 is pressed.
- Mode 4: runs a stopwatch. Using button 2 to start and stop the watch.

Please note that while the stopwatch is running, the normal clock is still running in the background.

7 Instructions

Your task is to sketch an FSM that describes your idea of how to solve the above problems and writes a firmware that runs on STM32.

8 Submission

You need to

- Demonstrate your work in the lab class and then
- Submit your FSM sketches and source code to the BKeL.