
REPORT

Title : LED toggle with Push-Button



| | | | |
|-----------------|---------------------|----------------|------------------------------------|
| Submission Date | 2021.10.08 | Major | Mechanical and control engineering |
| Subject | Embedded Controller | Professor | Young-Keun Kim |
| Name | Yeong-Won Song | Student Number | 21700375 |
| Partner Name | Jo-seph Shin | Partner Number | 21700395 |

Contents

| | |
|---|-----------|
| I. Introduction | 3 |
| 1.1 Purpose | 3 |
| 1.2 Parts List..... | 3 |
| II. Experimental Method | 4 |
| 2.1 EC_HAL driver | 4 |
| 2.2 Toggling LED | 10 |
| 2.3 Multiple LEDs Sequence | 13 |
| 2.3.1 external Circuit | 17 |
| III. Experimental Results..... | 18 |
| 3.1 Result Validity..... | 18 |
| IV. Discussion | 19 |
| V. Conclusion..... | 20 |

Appendix

I. Introduction

1.1 Purpose

Embedded systems have built-in electronic control systems that combine hardware and software. Software can be controlled to perform various functions intended by the manufacturer through a microprocessor or microcontroller. Therefore, in this experiment, we intend to create a HAL API drive to define the input and output types desired by users and implement them in hardware through LEDs.

1.2 Parts List

| Parts | Specification | Quantity |
|--|-----------------------------|----------|
| LED Register breadboard NUCLEO-F411RE | Red, white, yellow | 3 |
| | 330[Ω] | 3 |
| | - | 1 |
| | Arm®(a) Cortex®-M4 with FPU | 1 |
| M-F Jumper Wire | - | 5 |
| M-M Jumper Wire | - | 5 |

Table 1. Part List






| | | | | |
|---|---|---|--|---|
|  |  |  |  |  |
| NUCLEO-F411RE | LED | Register | Breadboard | M-F/M-M jumper |

Table 2. experiment equipment

II. Experimental Method

2.1 EC_HAL driver

Define a function for controlling the GPIO register in the EC_HAL driver.

- GPIO_MODER

| | | | | | | | | | | | | | | | |
|--------------|----|--------------|----|--------------|----|--------------|----|--------------|----|--------------|----|-------------|----|-------------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| MODER15[1:0] | | MODER14[1:0] | | MODER13[1:0] | | MODER12[1:0] | | MODER11[1:0] | | MODER10[1:0] | | MODER9[1:0] | | MODER8[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MODER7[1:0] | | MODER6[1:0] | | MODER5[1:0] | | MODER4[1:0] | | MODER3[1:0] | | MODER2[1:0] | | MODER1[1:0] | | MODER0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

Figure 1. Configuration of GPIO_MODER register

```
#define RESET 3UL

//MODER
#define INPUT      0
#define OUTPUT     1
#define ALTERNATE  2
#define ANALOG     3
```

Figure 2. Pre-defined Mode

For the user's convenience, four modes were previously defined.

```
void GPIO_mode(GPIO_TypeDef *Port, int pin, uint32_t mode){
    Port->MODER  &= ~(RESET<<(pin *2));
    Port->MODER  |= mode<<(pin *2);
}
```

Figure 3. MODER function code

Mode is controlled through two bits. Since two bits are used, the corresponding two bits are cleared first. And then the two bits of the mode is moved to the bit of the pin so that it is changed to the value set by the user.

- GPIO_OTYPER

| | | | | | | | | | | | | | | | |
|----------|------|------|------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OT15 | OT14 | OT13 | OT12 | OT11 | OT10 | OT9 | OT8 | OT7 | OT6 | OT5 | OT4 | OT3 | OT2 | OT1 | OT0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the output type of the I/O port.

0: Output push-pull (reset state)

1: Output open-drain

Figure 4. Configuration of GPIO_OTYPER register

```
//OTYPE
#define OUTPUT_PP    0
#define OUTPUT_OD    1
```

Figure 5. Pre-defined OTYPE

For the user's convenience, two modes were previously defined.

```
void GPIO_otype(GPIO_TypeDef* Port, int pin, uint32_t type)
{
    if(type == 0) //Output push-pull
    {
        Port->OTYPER &= ~(1<<pin);
    }
    if(type == 1) //Output Open-drain
    {
        Port->OTYPER |= 1<<pin ;
    }
}
```

Figure 6. OTYPE function code

Mode is controlled through one bit. Since one bit are used, the corresponding one bits are not cleared. And then the one bits of the mode is moved to the bit of the pin so that it is changed to the value set by the user.

- GPIO_OSPEEDR

| | | | | | | | | | | | | | | | |
|--------------------|-----|--------------------|-----|--------------------|-----|--------------------|-----|--------------------|-----|--------------------|-----|-------------------|-----|-------------------|-----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| OSPEEDR15 [1:0] | | OSPEEDR14 [1:0] | | OSPEEDR13 [1:0] | | OSPEEDR12 [1:0] | | OSPEEDR11 [1:0] | | OSPEEDR10 [1:0] | | OSPEEDR9 [1:0] | | OSPEEDR8 [1:0] | |
| r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| OSPEEDR7[1:0] | | OSPEEDR6[1:0] | | OSPEEDR5[1:0] | | OSPEEDR4[1:0] | | OSPEEDR3[1:0] | | OSPEEDR2[1:0] | | OSPEEDR1 [1:0] | | OSPEEDR0 [1:0] | |
| r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w | r/w |

Bits $2y:2y+1$ **OSPEEDRy[1:0]**: Port x configuration bits ($y = 0..15$)

These bits are written by software to configure the I/O output speed.

00: Low speed

01: Medium speed

10: Fast speed

11: High speed

Figure 7. Configuration of GPIO_OSPEEDR register

```
//OSPEED
#define LOW_SPEED      0
#define MEDIUM_SPEED  1
#define FAST_SPEED     2
#define HIGH_SPEED     3
```

Figure 8. Pre-defined OSPEED

For the user's convenience, four modes were previously defined.

```
void GPIO_ospeed(GPIO_TypeDef* Port, int pin, uint32_t speed){
    Port->OSPEEDR  &= ~(RESET<<(pin *2));
    Port->OSPEEDR  |= speed<<(pin *2);
}
```

Figure 9. OSPEED function code

Mode is controlled through two bits. Since two bits are used, the corresponding two bits are cleared first. And then the two bits of the mode is moved to the bit of the pin so that it is changed to the value set by the user.

- GPIO_PUPDR

| | | | | | | | | | | | | | | | |
|--------------|----|--------------|----|--------------|----|--------------|----|--------------|----|--------------|----|-------------|----|-------------|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| PUPDR15[1:0] | | PUPDR14[1:0] | | PUPDR13[1:0] | | PUPDR12[1:0] | | PUPDR11[1:0] | | PUPDR10[1:0] | | PUPDR9[1:0] | | PUPDR8[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PUPDR7[1:0] | | PUPDR6[1:0] | | PUPDR5[1:0] | | PUPDR4[1:0] | | PUPDR3[1:0] | | PUPDR2[1:0] | | PUPDR1[1:0] | | PUPDR0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 2y:2y+1 **PUPDRy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O pull-up or pull-down

00: No pull-up, pull-down

01: Pull-up

10: Pull-down

11: Reserved

Figure 10. Configuration of GPIO_OSPEEDR register

```
//PUPDR
#define NOPUPD      0
#define PU          1 //pull-up
#define PD          2 //pull-down
#define RESERVED    3
```

Figure 11. Pre-defined OSPEED

For the user's convenience, four modes were previously defined.

```
void GPIO_pudr(GPIO_TypeDef* Port, int pin, uint32_t pudr){
    Port->PUPDR  &= ~(RESET<<( pin *2));
    Port->PUPDR  |= pudr<<(pin *2);
}
```

Figure 12. OSPEED function code

Mode is controlled through two bits. Since two bits are used, the corresponding two bits are cleared first. And then the two bits of the mode is moved to the bit of the pin so that it is changed to the value set by the user.

- GPIO_read

```
uint32_t GPIO_read(GPIO_TypeDef *Port, int pin){
    return (Port->IDR) & (1<<pin) ;
}
```

Figure 13. read function code

- GPIO_write

```
void GPIO_write(GPIO_TypeDef *Port, int pin, uint32_t Output){
    if(Output == 0) //Output Low
    {
        Port->ODR &= ~(1<<pin);
    }
    if(Output == 1) //Output High
    {
        Port->ODR |= 1<<pin ;
    }
}
```

Figure 14. write function code

- GPIO_init

```
void GPIO_init(GPIO_TypeDef *Port, int pin, uint32_t mode){
    if (Port == GPIOA)
        RCC_GPIOA_enable();

    if (Port == GPIOB)
        RCC_GPIOB_enable();

    if (Port == GPIOC)
        RCC_GPIOC_enable();

    GPIO_mode(Port, pin, mode);
}
```

Figure 15. init function code

Select the GPIO to be activated and set the mode for the pin to be used

<ecRCC.h>

```
#ifndef __EC_RCC_H
#define __EC_RCC_H

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#include "stm32f411xe.h"

void RCC_HSI_init(void);
void RCC_PLL_init(void);
void RCC_GPIOA_enable(void);
void RCC_GPIOB_enable(void);
void RCC_GPIOC_enable(void);
// void RCC_GPIO_enable(GPIO_TypeDef * GPIOx);

extern int EC_SYSCLOCK;

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif
```

ecRCC.c : See Appendix

<ecGPIO.h>

```
// Distributed for LAB: GPIO

#include "stm32f411xe.h"
#include "ecRCC.h"

#ifndef __ECGPIO_H
#define __ECGPIO_H

#define LOW          0
#define HIGH         1

#define RESET 3UL

//MODER
#define INPUT        0
#define OUTPUT       1
#define ALTERNATE    2
#define ANALOG       3

//OSPEED
#define LOW_SPEED    0
#define MEDIUM_SPEED 1
#define FAST_SPEED   2
#define HIGH_SPEED   3

//PUPDR
#define NOPUPD       0
#define PU            1 //pull-up
#define PD            2 //pull-down
#define RESERVED     3

//OTYPE
#define OUTPUT_PP     0
#define OUTPUT_OD     1

#define LED_PIN       5
#define BUTTON_PIN    13

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

void GPIO_init(GPIO_TypeDef *Port, int pin, uint32_t mode);
void GPIO_mode(GPIO_TypeDef* Port, int pin, uint32_t mode);

void GPIO_write(GPIO_TypeDef *Port, int pin, uint32_t Output);
uint32_t GPIO_read(GPIO_TypeDef *Port, int pin);
void GPIO_ospeed(GPIO_TypeDef* Port, int pin, uint32_t speed);
void GPIO_otype(GPIO_TypeDef* Port, int pin, uint32_t type);
void GPIO_pudr(GPIO_TypeDef* Port, int pin, uint32_t pudr);

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

ecGPIO.c : See Appendix

2.2 Toggling LED

We want to code a system that toggles an LED when a button is pressed. On the surface, it looks like the LED is toggled whenever the button is pressed. However, in Pull-up mode, the default value is maintained at 1, so when the button is pressed, it becomes LOW, but when the button is not pressed, it maintains the HIGH state again. In other words, the LED should not be toggled immediately when the button is pressed but toggled when the button is pressed and released. Therefore, we need to code the MCU to recognize that the user presses and releases the button.

```
while(1){  
  
    if(GPIO_read(GPIOC, BUTTON_PIN) == 1){  
        current_state = 0;  
    }  
    if(GPIO_read(GPIOC, BUTTON_PIN) == 0){  
        prev_state = 0;  
        current_state = 1;  
    }else current_state = 0;  
    if(current_state == 0 && prev_state == 0){  
        toggle = !toggle;  
        GPIO_write(GPIOA, LED_PIN, toggle);  
        prev_state = 1;  
    }  
}
```

Figure 16. Toggling LED function code

The prev_state in the code above is given a value of 0 when the button is pressed and 1 when the button is not pressed. Current_state is given a value of 0 when the button is not pressed and 1 when the button is pressed. When the button is pressed, the value of prev_state becomes 0 and current_state becomes 0, so toggling is performed once. Once implemented toggling, the value of prev_state is set to 1 again to prevent toggling until the next button is pressed and released. When controlling the operation of DC-motor through the mbed compiler in LAB1, interruptIn is used to change the state when the user button is pressed. Similarly, in this experiment, when a button is pressed, it escapes from the toggling conditional statement and executing again, like the interruptIn method.

I tried running LOW, MEDIUM, FAST, and HIGH SPEED by changing the speed of the LED. LEDs work well at low speeds, so there is no need to use high speeds. Therefore, it was difficult to determine the speed difference easily with the naked eye. Using unnecessarily high speed only increases energy consumption. In case of high speed, it is mainly used when fast communication is required.

- Source code

<PART 2_Toggle LED>

```
#include "stm32f4xx.h"
#include "ecRCC.h"
#include "ecGPIO.h"

#define LED_PIN 5
#define BUTTON_PIN 13

void setup(void);
int main(void) {
    // Initialiization -----
    setup();

    uint32_t toggle = 1;
    uint32_t current_state = 0; // 0=released ,1=pressed
    uint32_t prev_state = 0;    // 0=pressed ,1=released

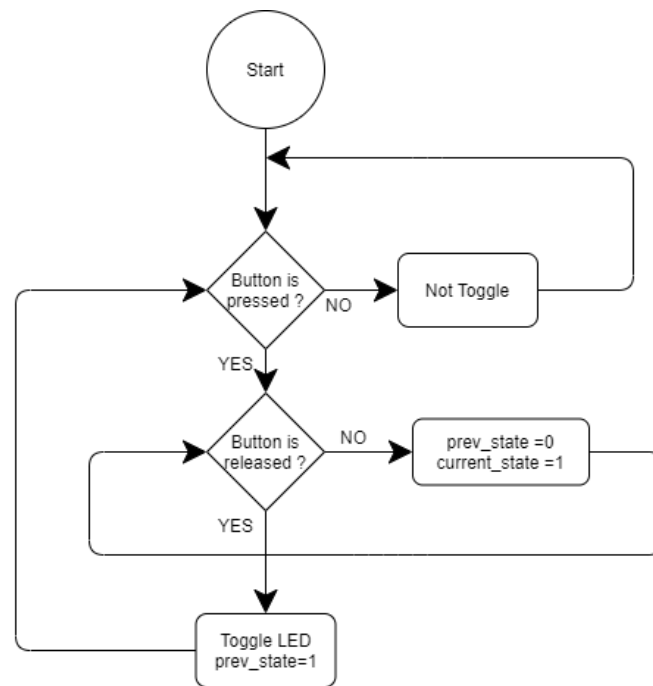
    // Inifinite Loop -----
    while(1){
        if(GPIO_read(GPIOC, BUTTON_PIN) == 1){
            current_state =0;
        }
        if(GPIO_read(GPIOC, BUTTON_PIN) == 0){
            prev_state =0;
            current_state =1;
        }else current_state = 0;
        if(current_state == 0 && prev_state == 0){
            toggle = !toggle;
            GPIO_write(GPIOA, LED_PIN, toggle);
            prev_state = 1;
        }
    }
}

// Initialiization
void setup(void)
{
    RCC_HSI_init();

    //GPIO A
    GPIO_init(GPIOA, LED_PIN, OUTPUT);    // calls RCC_GPIOA_enable()
    GPIO_mode(GPIOA, LED_PIN, OUTPUT);
    GPIO_ospeed(GPIOA, LED_PIN, MEDIUM_SPEED);
    GPIO_otype(GPIOA, LED_PIN, OUTPUT_PP);
    GPIO_pudr(GPIOA, LED_PIN, PU);

    //GPIO C
    GPIO_init(GPIOC, BUTTON_PIN, INPUT); // calls RCC_GPIOC_enable()
    GPIO_mode(GPIOC, BUTTON_PIN, INPUT);
    GPIO_pudr(GPIOC, BUTTON_PIN , PU);
}
```

- **Flow chart**



2.3 Multiple LEDs Sequence

```
while(1){

    if(GPIO_read(GPIOC, BUTTON_PIN) == 1){
        current_state =0;
    }
    if(GPIO_read(GPIOC, BUTTON_PIN) == 0){
        prev_state =0;
        current_state =1;
        LED_Control =1;
    }else current_state = 0;
    if(current_state == 0 && prev_state == 0){
        if(LED_Control == 1){
            button_nextstate = ButtonST(button_nextstate); //check button is pressed and go to next state
        }
        if(button_nextstate ==1){
            GPIO_write(GPIOA, LED_PA5, HIGH);
            GPIO_write(GPIOA, LED_PA6, LOW);
            GPIO_write(GPIOA, LED_PA7, LOW);
            GPIO_write(GPIOB, LED_PB6, LOW);
            prev_state = 1;
            LED_Control = 0;
        }
    }
}
```

Figure 17. Multiple LEDs Sequence function code

It proceeded with an algorithm like LED toggling. whenever the button is pressed, LEDs 0,1,2,3 turn on and off in sequence. ButtonST function is defined to check how many times the button is pressed. ButtonST function is executed when the button is pressed.

```
uint32_t ButtonST(uint32_t value){
    uint32_t next =0;
    if(value == 0){
        next =1;
    }
    if(value ==1){
        next =2;
    }
    if(value ==2){
        next =3;
    }
    if(value ==3){
        next =4;
    }
    if(value ==4){
        next =1;
    }
    return next;
}
```

Figure 18. ButtonST function code

The ButtonST function returns the input integer value, that is, +1 to the number of times the button is pressed. When button_nextstate is 4, since LED 3 is turned on by pressing the button 4 times, the value of 1 is returned because LED 0 should be turned on again when the button is pressed next time.

<PART 3_Multiple LEDs On/Off in Sequence>

```
#include "stm32f4xx.h"
#include "ecRCC.h"
#include "ecGPIO.h"

#define LED_PA5 5
#define LED_PA6 6
#define LED_PA7 7
#define LED_PB6 6
#define BUTTON_PIN 13

uint32_t ButtonST(uint32_t value){
    uint32_t next =0;
    if(value == 0){
        next =1;
    }
    if(value ==1){
        next =2;
    }
    if(value ==2){
        next =3;
    }
    if(value ==3){
        next =4;
    }
    if(value ==4){
        next =1;
    }
    return next;
}

void setup(void);
int main(void) {
    // Initialization -----
    setup();
    uint32_t current_state = 0; // 0=released ,1=pressed //
    uint32_t prev_state = 1; // 0=pressed ,1=released // button
    uint32_t button_nextstate = 0; //button pressed state
    uint32_t LED_Control = 0; //button pressed state
    // Infinite Loop -----
    while(1){

        /*if(GPIO_read(GPIOC, BUTTON_PIN) == 1){
            current_state =0;
        }*/
        if(GPIO_read(GPIOC, BUTTON_PIN) == 0){
            prev_state =0;
            current_state =1;
            LED_Control =1;
        }else current_state = 0;
        if(current_state == 0 && prev_state == 0){
            if(LED_Control == 1){
                button_nextstate = ButtonST(button_nextstate); //check button is pressed and go to next sta
            }
            if(button_nextstate ==1){
                GPIO_write(GPIOA, LED_PA5, HIGH);
                GPIO_write(GPIOA, LED_PA6, LOW);
                GPIO_write(GPIOA, LED_PA7, LOW);
                GPIO_write(GPIOB, LED_PB6, LOW);
                prev_state = 1;
                LED_Control = 0;
            }
        }
    }
}
```

```

        if(button_nextstate ==2){
            GPIO_write(GPIOA, LED_PA5, LOW);
            GPIO_write(GPIOA, LED_PA6, HIGH);
            GPIO_write(GPIOA, LED_PA7, LOW);
            GPIO_write(GPIOB, LED_PB6, LOW);
            prev_state = 1;
            LED_Control = 0;
        }
        if(button_nextstate ==3){
            GPIO_write(GPIOA, LED_PA5, LOW);
            GPIO_write(GPIOA, LED_PA6, LOW);
            GPIO_write(GPIOA, LED_PA7, HIGH);
            GPIO_write(GPIOB, LED_PA6, LOW);
            prev_state = 1;
            LED_Control = 0;
        }
        if(button_nextstate ==4){
            GPIO_write(GPIOA, LED_PA5, LOW);
            GPIO_write(GPIOA, LED_PA6, LOW);
            GPIO_write(GPIOA, LED_PA7, LOW);
            GPIO_write(GPIOB, LED_PB6, HIGH);
            prev_state = 1;
            LED_Control = 0;
        }
    }
}

// Initialiization
void setup(void)
{
    RCC_HSI_init();
    GPIO_init(GPIOA, LED_PA5, OUTPUT);    // calls RCC_GPIOA_enable()    PA5
    GPIO_init(GPIOA, LED_PA6, OUTPUT);    // calls RCC_GPIOA_enable()    PA6
    GPIO_init(GPIOA, LED_PA7, OUTPUT);    // calls RCC_GPIOA_enable()    PA7
    GPIO_init(GPIOB, LED_PB6, OUTPUT);    // calls RCC_GPIOB_enable()    PB6
    GPIO_init(GPIOC, BUTTON_PIN, INPUT);  // calls RCC_GPIOC_enable()

    //GPIOA PA5
    GPIO_mode(GPIOA, LED_PA5, OUTPUT);
    GPIO_ospeed(GPIOA, LED_PA5, MEDIUM_SPEED);
    GPIO_otype(GPIOA, LED_PA5, OUTPUT_PP);
    GPIO_pudr(GPIOA, LED_PA5, PU);

    //GPIOA PA6
    GPIO_mode(GPIOA, LED_PA6, OUTPUT);
    GPIO_ospeed(GPIOA, LED_PA6, MEDIUM_SPEED);
    GPIO_otype(GPIOA, LED_PA6, OUTPUT_PP);
    GPIO_pudr(GPIOA, LED_PA6, PU);

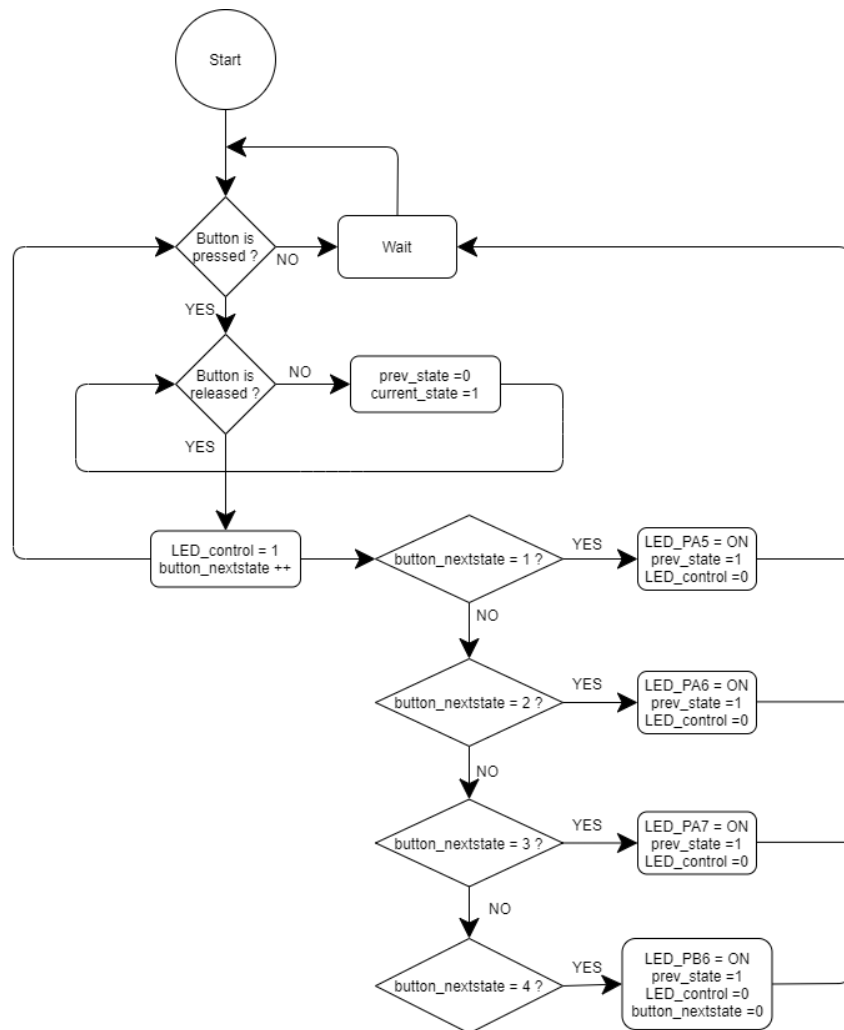
    //GPIOA PA7
    GPIO_mode(GPIOA, LED_PA7, OUTPUT);
    GPIO_ospeed(GPIOA, LED_PA7, MEDIUM_SPEED);
    GPIO_otype(GPIOA, LED_PA7, OUTPUT_PP);
    GPIO_pudr(GPIOA, LED_PA7, PU);

    //GPIOB PB6
    GPIO_mode(GPIOB, LED_PB6, OUTPUT);
    GPIO_ospeed(GPIOB, LED_PB6, MEDIUM_SPEED);
    GPIO_otype(GPIOB, LED_PB6, OUTPUT_PP);
    GPIO_pudr(GPIOB, LED_PB6, PU);

    //GPIOC
    GPIO_mode(GPIOC, BUTTON_PIN, INPUT);
    GPIO_pudr(GPIOC, BUTTON_PIN, PU);
}

```

- **Flow chart**



2.3.1 External circuit

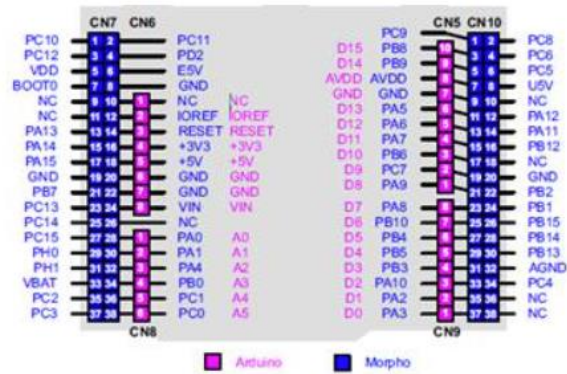


Figure 18. NUCLEO-F411RE

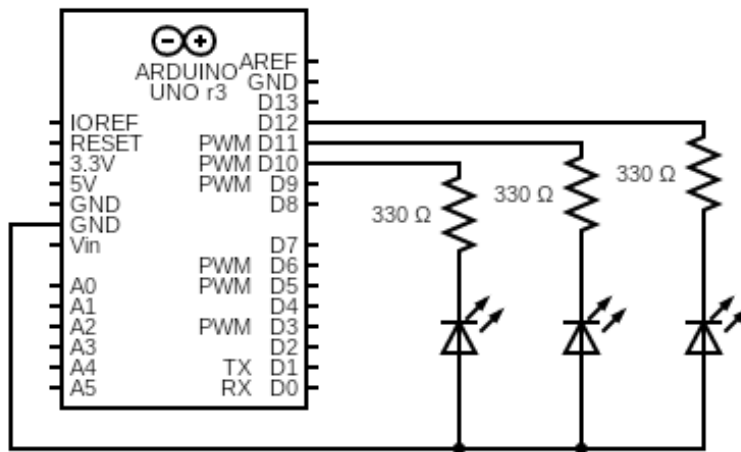


Figure 19. External circuit diagram

PA6, PA7, and PB6 of the NUCLEO-F411RE are used as terminals D12, D11, and D10, respectively.

III. Experimental Result

3.1 Result Validity

| Button_nextState | LED0 | LED1 | LED2 | LED3 | Validity |
|----------------------|------|------|------|------|----------|
| 0(initial condition) | X | X | X | X | O |
| 1 | O | X | X | X | O |
| 2 | X | O | X | X | O |
| 3 | X | X | O | X | O |
| 4 | X | X | X | O | O |

Table 3. Result check

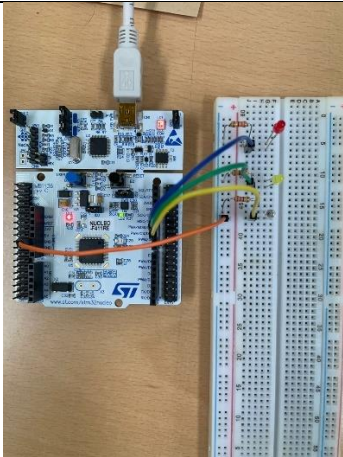
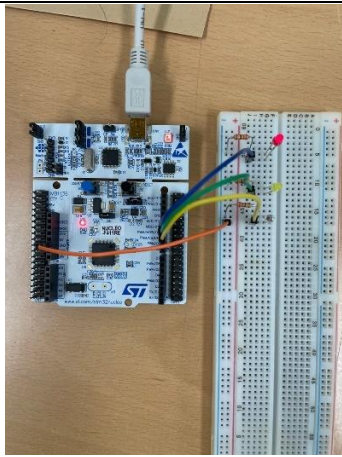
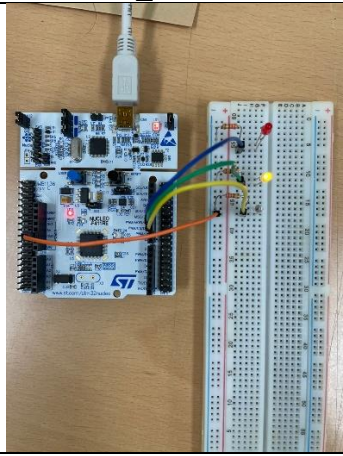
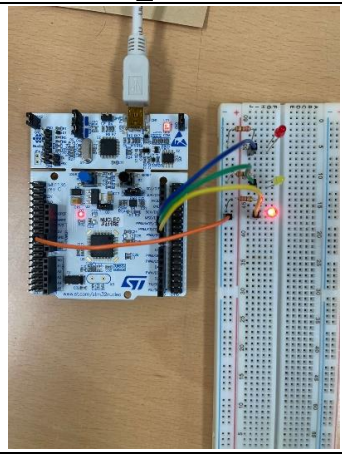
| | |
|---|--|
|  |  |
| Button_nextState 1 | Button_nextState 2 |
|  |  |
| Button_nextState 3 | Button_nextState 4 |

Table 4. LED output result

IV. Discussion

1) What the differences between open-drain and Push-pull for output pin?

- Push-pull

It allows the pin to supply and absorb current. In push-pull, PMOS is connected to Vcc terminal and NMOS is connected to GND. Push-pull output makes the output voltages 0 or 1 value according to the input voltage. Push-pull can serve both supply and absorb. If it is in the push state, it pushes the current to the outside using the VCC voltage, so Output = 1. That is, it can be viewed as supply. In the pull state, it is connected to GND and output = 0. That is, drain the current.

- Open-drain

It allows the pin to only absorb current. Open-drain has only one NMOS, and if input Vin is 1, output is open and floating. The floating state is high-impedance or tri-state. Therefore, a pull-up type voltage is additionally applied to the circuit to remove the floating phenomenon and to output a stable output value. If Vin is 0, the output is LOW (0).

2) Find out a typical solution for software debouncing and hardware debouncing.

What method of debouncing did this NUCLEO board used for the push-button(B1)?

- Software

We can solve debouncing not only in hardware but also in software. The simplest way to debounce a switch in software is to give the micro-controller a time delay after it detects the first pulse before checking again. Therefore, a stable signal can be transmitted after the time delay. In this lab, we did not use the delay function. Therefore, we tried to solve the debouncing problem by giving a time delay until the next switch is pressed after pressing the switch once through the if conditional statement.

- Hardware

Most MCUs have an internal pullup resistor that can be enabled for a given GPIO pin. Schmitt trigger is one solution. Schmitt trigger is a voltage comparator with hysteresis. Switches with hysteresis use the upper threshold when the low state last appeared and the lower threshold when in the high state. This leaves a "dead band" where the range between the two thresholds does not transition. This can be solved by hardware. Another simple hardware solution is to use resistors and capacitors. The R and C values should be chosen to match the desired time for debounce. Connecting the capacitor directly to the switch will cause unwanted high-frequency voltage noise, so an appropriate resistor value must be set.

3) Troubleshooting

Q1. After binary is exported to MCU LED does not blink even the button is toggled.

A1. Press Reset Button

Q2. How to solve bouncing phenomenon?

A2. Through software method, I solve the bouncing phenomenon. Once implemented toggling, the value of `prev_state` is set to 1 again to prevent toggling until the next button is pressed and released.

Q3. In part 3, how did the program check the number of times the button was pressed?

A3. Define new function called `ButtonST`. `ButtonST` function is defined to count how many times the button is pressed. `ButtonST` function is executed when the button is pressed.

V. Conclusion

Through this lab, we learned how to control the HAL API. we could create a user-defined function according to the user's settings and apply it directly. Also, it was possible to know the principle of controlling the output according to the pull-up and pull-down. In addition, system debouncing was discussed in terms of software and hardware. In this experiment, we did not design and debounce circuits ourselves, but I think it will be very helpful in making the circuit by designing the RC Circuit and Filter directly based on the discussion.

Appendix

- Source code

<ecRCC.c>

```
#include "stm32f4xx.h"
#include "ecRCC.h"

volatile int EC_SYSCLK=16000000;

void RCC_HSI_init() {
    // Enable High Speed Internal Clock (HSI = 16 MHz)
    //RCC->CR |= ((uint32_t)RCC_CR_HSION);
    RCC->CR |= 0x00000001U;

    // wait until HSI is ready
    //while ( (RCC->CR & (uint32_t) RCC_CR_HSIRDY) == 0 ) {};
    while ( (RCC->CR & 0x00000002U) == 0 ) {}

    // Select HSI as system clock source
    RCC->CFGR &= (uint32_t) (~RCC_CFGR_SW); // not essential
    RCC->CFGR |= (uint32_t) RCC_CFGR_SW_HSI; //00: HSI16 oscillator used as system clock

    // Wait till HSI is used as system clock source
    while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) != 0 );

    //EC_SYSTEM_CLK=16000000;
    //EC_SYSCLK=16000000;
    EC_SYSCLK=16000000;
}

void RCC_PLL_init() {
    // To Correctly read data from FLASH memory, the number of wait states (LATENCY)
    // must be correctly programmed according to the frequency of the CPU clock
    // (HCLK) and the supply voltage of the device.
    FLASH->ACR &= ~FLASH_ACR_LATENCY;
    FLASH->ACR |= FLASH_ACR_LATENCY_2WS;
}

void RCC_PLL_init() {
    // To Correctly read data from FLASH memory, the number of wait states (LATENCY)
    // must be correctly programmed according to the frequency of the CPU clock
    // (HCLK) and the supply voltage of the device.
    FLASH->ACR &= ~FLASH_ACR_LATENCY;
    FLASH->ACR |= FLASH_ACR_LATENCY_2WS;

    // Enable the Internal High Speed oscillator (HSI)
    RCC->CR |= RCC_CR_HSION;
    while((RCC->CR & RCC_CR_HSIRDY) == 0);
    // Disable PLL for configuration
    RCC->CR &= ~RCC_CR_PLLON;
    // Select clock source to PLL
    RCC->PLLCFGR &= ~RCC_PLLCFGR_PLLSRC; // Set source for PLL: clear bits
    RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSI; // Set source for PLL: 0 =HSI, 1 = HSE
    // Make PLL as 84 MHz
    // f(VCO clock) = f(PLL clock input) * (PLLN / PLLM) = 16MHz * 84/8 = 168 MHz
    // f(PLL_R) = f(VCO clock) / PLLP = 168MHz/2 = 84MHz
    RCC->PLLCFGR = (RCC->PLLCFGR & ~RCC_PLLCFGR_PLLN) | 84U << 6;
    RCC->PLLCFGR = (RCC->PLLCFGR & ~RCC_PLLCFGR_PLLM) | 8U ;
    RCC->PLLCFGR &= ~RCC_PLLCFGR_PLLP; // 00: PLLP = 2, 01: PLLP = 4, 10: PLLP = 6, 11: PLLP = 8

    // Enable PLL after configuration
    RCC->CR |= RCC_CR_PLLON;
    while((RCC->CR & RCC_CR_PLLRDY)>>25 != 0);

    // Select PLL as system clock
    RCC->CFGR &= ~RCC_CFGR_SW;
    RCC->CFGR |= RCC_CFGR_SW_PLL;

    // Wait until System Clock has been selected
    while ((RCC->CFGR & RCC_CFGR_SWS) != 8UL);
}
```

```

// The maximum frequency of the AHB and APB2 is 100MHz,
// The maximum frequency of the APB1 is 50 MHz.
RCC->CFGR &= ~RCC_CFGR_HPRE;    // AHB prescaler = 1; SYSCLK not divided (84MHz)
RCC->CFGR &= ~RCC_CFGR_PPRE1;    // APB high-speed prescaler (APB1) = 2, HCLK divided by 2 (42MHz)
RCC->CFGR |= RCC_CFGR_PPRE1_2;
RCC->CFGR &= ~RCC_CFGR_PPRE2;    // APB high-speed prescaler (APB2) = 1, HCLK not divided (84MHz)

EC_SYSCLK=84000000;
}

void RCC_GPIOA_enable()
{
    // HSI is used as system clock
    RCC_HSI_init();
    // RCC Peripheral Clock Enable Register
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
}

void RCC_GPIOB_enable()
{
    // HSI is used as system clock
    RCC_HSI_init();
    // RCC Peripheral Clock Enable Register
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;
}

void RCC_GPIOC_enable()
{
    // HSI is used as system clock
    RCC_HSI_init();
    // RCC Peripheral Clock Enable Register
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
}

```

<ecGPIO.c>

```
// Distributed for LAB: GPIO
#include "stm32f4xx.h"
#include "stm32f4llxe.h"
#include "ecGPIO.h"

void GPIO_init(GPIO_TypeDef *Port, int pin, uint32_t mode){
    if (Port == GPIOA)
        RCC_GPIOA_enable();

    if (Port == GPIOB)
        RCC_GPIOB_enable();

    if (Port == GPIOC)
        RCC_GPIOC_enable();

    GPIO_mode(Port, pin, mode);
}

void GPIO_mode(GPIO_TypeDef *Port, int pin, uint32_t mode){
    Port->MODER &= ~(RESET<<( pin *2));
    Port->MODER |= mode<<(pin *2);
}

void GPIO_write(GPIO_TypeDef *Port, int pin, uint32_t Output){
    if(Output == 0) //Output Low
    {
        Port->ODR &= ~(1<<pin);
    }
    if(Output == 1) //Output High
    {
        Port->ODR |= 1<<pin ;
    }
}

uint32_t GPIO_read(GPIO_TypeDef *Port, int pin){
    return (Port->IDR) & (1<<pin) ;
}

void GPIO_ospeed(GPIO_TypeDef* Port, int pin, uint32_t speed){
    Port->OSPEEDR &= ~(RESET<<( pin *2));
    Port->OSPEEDR |= speed<<(pin *2);
}

void GPIO_otype(GPIO_TypeDef* Port, int pin, uint32_t type){ // 0 : Output push-pull 1: Output open-drain
    if(type == 0) //Output push-pull
    {
        Port->OTYPER &= ~(1<<pin);
    }
    if(type == 1) //Output Open-drain
    {
        Port->OTYPER |= 1<<pin ;
    }
}

void GPIO_pudr(GPIO_TypeDef* Port, int pin, uint32_t pudr){
    Port->PUPDR &= ~(RESET<<( pin *2));
    Port->PUPDR |= pudr<<(pin *2);
}
```


- Documentation

GPIO_init()

Initializes GPIO pins with default setting and Enables GPIO Clock. Mode: In/Out/AF/Analog

```
void GPIO_init(GPIO_TypeDef *Port, int pin, uint32_t mode);
```

Parameters

- **Port:** Port Number, GPIOA~GPIOH
- **pin:** pin number (int) 0~15
- **mode:** INPUT (0), OUTPUT (1), ALTERNATE(2), ANALOG (3)

Example code

```
GPIO_init(GPIOA, 5, OUTPUT);  
GPIO_init(GPIOC, 13, INPUT); //GPIO_init(GPIOC, 13, 0);
```

GPIO_mode()

Configures GPIO pin modes: In/Out/AF/Analog

```
void GPIO_mode(GPIO_TypeDef* Port, int pin, uint32_t mode);
```

Parameters

- **Port:** Port Number, GPIOA~GPIOH
- **pin:** pin number (int) 0~15
- **mode:** INPUT (0), OUTPUT (1), ALTERNATE(2), ANALOG (3)

Example code

```
GPIO_mode(GPIOA, 5, OUTPUT);
```

GPIO_pudr()

Configures GPIO pin modes: NOPUPD/PU/PD/RESERVED

```
void GPIO_pudr(GPIO_TypeDef* Port, int pin, uint32_t pudr);
```

Parameters

- **Port:** Port Number, GPIOA~GPIOH
- **pin:** pin number (int) 0~15
- **pupdr:** NOPUPD (0), PU(1), PD(2), RESERVED(3)

Example code

```
GPIO_pudr(GPIOA, 5, PU)
```

GPIO_ospeed()

Configures GPIO pin modes: Low speed/Medium speed/Fast speed/High speed

```
void GPIO_ospeed(GPIO_TypeDef* Port, int pin, uint32_t speed);
```

Parameters

- **Port:** Port Number, GPIOA-GPIOH
- **pin:** pin number (int) 0-15
- **ospeed:** LOW_SPEED(0), MEDIUM_SPEED(1), FAST_SPEED(2), HIGH_SPEED(3)

Example code

```
GPIO_ospeed(GPIOA, 5, 1); // 1 : Medium speed
```

GPIO_otype()

Configures GPIO pin modes: Output push-pull/Output open-drain

```
void GPIO_otype(GPIO_TypeDef* Port, int pin, uint32_t type);
```

Parameters

- **Port:** Port Number, GPIOA-GPIOH
- **pin:** pin number (int) 0-15
- **otype:** OUTPUT_PP(0), OUTPUT_OD(1)

Example code

```
GPIO_otype(GPIOA, 5, 0); // 0 : Output push-pull
```

GPIO_read()

Read the value of the pin

```
uint32_t GPIO_read(GPIO_TypeDef *Port, int pin);
```

Parameters

- **Port:** Port Number, GPIOA-GPIOH
- **pin:** pin number (int) 0-15

Example code

```
if(GPIO_read(GPIOC, BUTTON_PIN) == 0){  
    GPIO_write(GPIOA, LED_PA5, HIGH);  
}
```

GPIO_write()

Configures GPIO pin modes: Low/High

```
void GPIO_write(GPIO_TypeDef *Port, int pin, uint32_t Output);
```

Parameters

- **Port:** Port Number, GPIOA-GPIOH
- **pin:** pin number (int) 0-15

Example code

```
GPIO_write(GPIOA, 5, LOW);
```

ButtonST()

count how many times the button is pressed

```
uint32_t ButtonST(uint32_t value);
```

Parameters

- **value:** button pressed times

Example code

```
button_nextstate = ButtonST(button_nextstate); //update variable button_nextstate
```
