嵌入式系统实验报告

实验名称：	Lab6 操作 STM32 外设的 4 种方式
姓　　名：	江姝潼
学　　号：	2019211653
学 院(系)：	计算机学院
专　　业：	网络工程
指导教师：	戴志涛、刘健培

2021 年 12 月 15 日

1 实验目的

在 MDK 环境操作 STM32 的 4 种方式，观察和分析其不同的特点。

2 实验环境

- ST-Link 仿真器
- KeiluVision5 MDK 集成开发软件
- PC 机 Window10 (64bit)

3 实验要求

将 LED 引脚改为 PA7，按键引脚改为 PC3，LED 与按键状态改为反相输出。重做此 4 种方式，将程序源码和程序输出截图贴在作业答卷里。

4 不使用 STM32 标准外设库+直接操作寄存器

本方法是以最底层的方式实现，直接找到寄存器所在的位置并进行操作。编程开发的步骤如下：

1. 首先使能 APB 总线上该引脚所属 GPIO 端口的时钟，查阅的相关资料如下：

Table 3. Register boundary addresses

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC	AHB	*Section 21.6.9 on page 563*
0x5000 0000 - 0x5003 FFFF	USB OTG FS		*Section 28.16.6 on page 912*
0x4003 0000 - 0x4FFF FFFF	Reserved		-
0x4002 8000 - 0x4002 9FFF	Ethernet		*Section 29.8.5 on page 1069*
0x4002 3400 - 0x4002 7FFF	Reserved		-
0x4002 3000 - 0x4002 33FF	CRC		*Section 4.4.4 on page 65*
0x4002 2000 - 0x4002 23FF	Flash memory interface		-
0x4002 1400 - 0x4002 1FFF	Reserved		-
0x4002 1000 - 0x4002 13FF	Reset and clock control RCC		*Section 7.3.11 on page 120*
0x4002 0800 - 0x4002 0FFF	Reserved		-
0x4002 0400 - 0x4002 07FF	DMA2		*Section 13.4.7 on page 288*
0x4002 0000 - 0x4002 03FF	DMA1		
0x4001 8400 - 0x4001 FFFF	Reserved		-
0x4001 8000 - 0x4001 83FF	SDIO		*Section 22.9.16 on page 620*

可知 RCC_APB2ENR 所在的基址为 0x40021000，再加上偏移量 0x18，是在所在的实际的地址。

```
#define RCC_APB2ENR (*((volatile unsigned long*)(0x40021000 + 0x18)))
```

7.3.7 APB2 peripheral clock enable register (RCC_APB2ENR)

Address: 0x18

Reset value: 0x0000 0000

Access: word, half-word and byte access

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										TIM11 EN	TIM10 EN	TIM9 EN	Reserved		
										rw	rw	rw			
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADC3 EN	USART 1EN	TIM8 EN	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	IOPG EN	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw

本实验要使能的为 GPIOA 和 GPIOC 端口，分别再 APB2 起始位置后 2 和 4 个 bit 的位置，用位操作可以使得他们激活成功。

```
void Button_Init(void) {
  /* Enable GPIOC clock           */
  RCC_APB2ENR |= (1ul << 4);
```

```
void LED_Init(void) {
  /* Enable GPIOA clock           */
  RCC_APB2ENR |= (1ul << 2);
```

2.下面初始化 GPIO 引脚配置，此实验需要将 LED 对应的 GPIO 引脚配置为 output 模式，按键对应的 GPIO 引脚配置为 input 模式，其中 LED output 使用上下拉模式，按键 input 使用 float input 模式。

下面以 GPIOC.3 为例，介绍寻址过程：

通过查阅资料可知，8-15 寄存器在 GPIO x_CRH 里，而 0-7 寄存器在 GPIO x_CRL 里，实验中需要将 GPIOC.13 换成 GPIOC.3，则需要将基址 GPIOC_CRL 的基址：

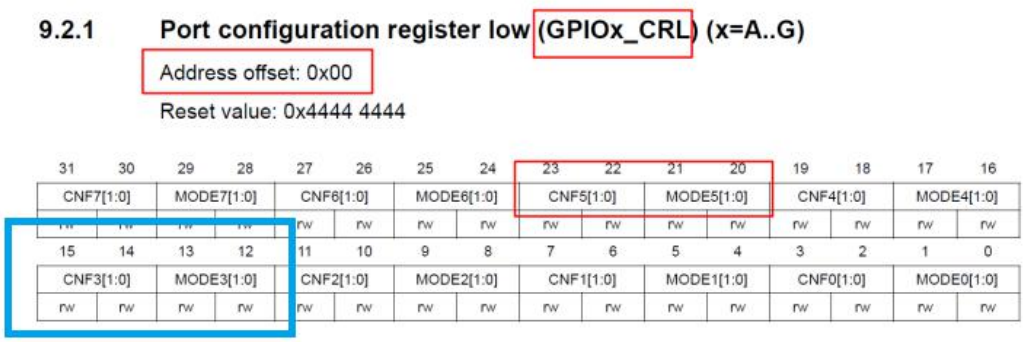
```
#define BUTTON_PIN  3 //PC.3
#define GPIOC_CRL   (*((volatile unsigned long*)(0x40011000 + 0x00)))
#define GPIOC_IDR   (*((volatile unsigned long*)(0x40011000 + 0x08)))
```

GPIOA 各寄存器的位置没有发生改变，还是原来的位置：

```
#define LED_PIN 7   //PA.7
#define GPIOA_CRL   (*((volatile unsigned long*)(0x40010800 + 0x00)))
#define GPIOA_ODR   (*((volatile unsigned long*)(0x40010800 + 0x0C)))
#define GPIOA_BSRR  (*((volatile unsigned long*)(0x40010800 + 0x10)))
#define GPIOA_BRR   (*((volatile unsigned long*)(0x40010800 + 0x14)))
```

4.下面操作各引脚，

在 Button_Init 中，需要初始化 GPIOC.3，对应的位置为 12~15 位，需要先清零再置成 0b0100，代表为 Floating input 和 Input Mode



9.2.1 Port configuration register low (GPIOx_CRL) (x=A..G)

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[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
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

具体代码如下：

```
void Button_Init(void) {
  /* Enable GPIOC clock           */
  RCC_APB2ENR |= (1ul << 4);

  /* Configure Button (PC.3) pins as input */
  GPIOC_CRL &= ~(0xFul << 12);  //Clear Bit[12..15]
  GPIOC_CRL |=  (0x4ul << 12); //Set Bit[12..15] = 0b0100
}
```

在 LED_Init 中，需要初始化 GPIOA.7，对应的位置为 28~31 位，需要先清零再置成 0b0001，代表为 General purpose output push-pull 和 Output Mode:

9.2.1 Port configuration register low (GPIOx_CRL) (x=A..G)

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[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
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

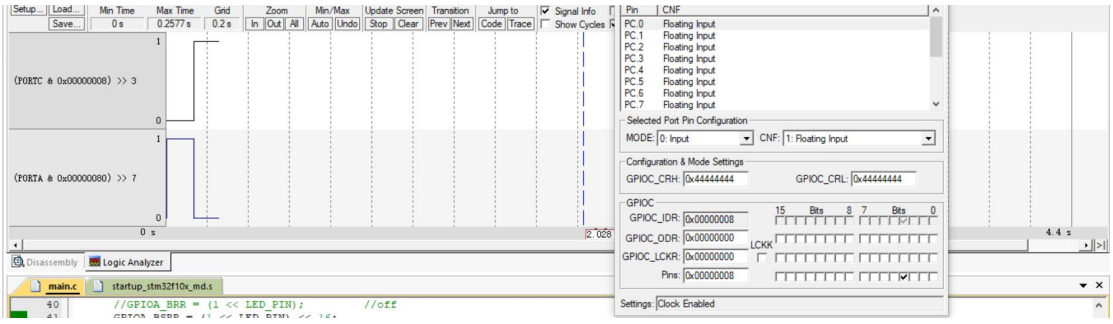
具体代码如下:

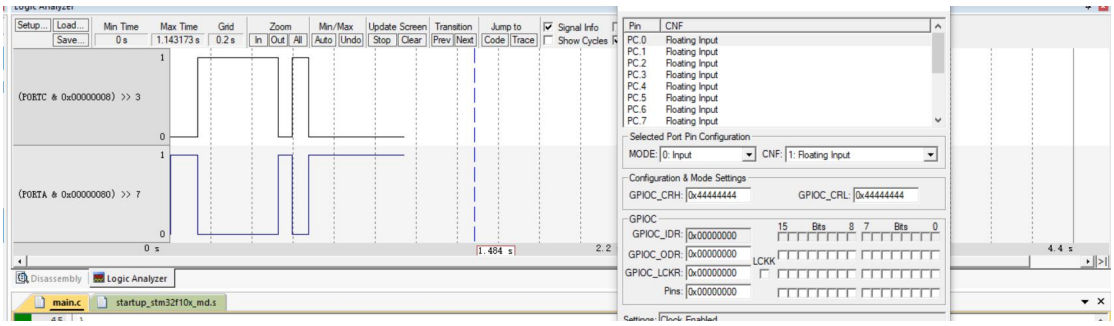
```
void LED_Init(void) {
  /* Enable GPIOA clock                */
  RCC_APB2ENR |= (1ul << 2);

  /* Configure LED (PA.7) pins as output */
  GPIOA_CRL &= ~(0xFul << 28); //Clear Bit[28..31]
  GPIOA_CRL |=  ( 0x1ul << 28); //Set Bit[28..31] = 0x0001
}
```

5.为了修改使得 LED 与按键状态改为反相输出，只需对 0 和 1 的状态对应反向处理即可，这里将 state 从 0 改成了 1

```
37 void LED_Write(int state) {
38     //修改使得反方向跳变，将state从1改成0即可
39     if(state == 1) {
40       GPIOA_BSRR = (1 << LED_PIN) << 16;
41     } else {
42       GPIOA_BSRR = (1 << LED_PIN);          //on
43     }
44 }
45
```

6.编译运行，运行结果如下：





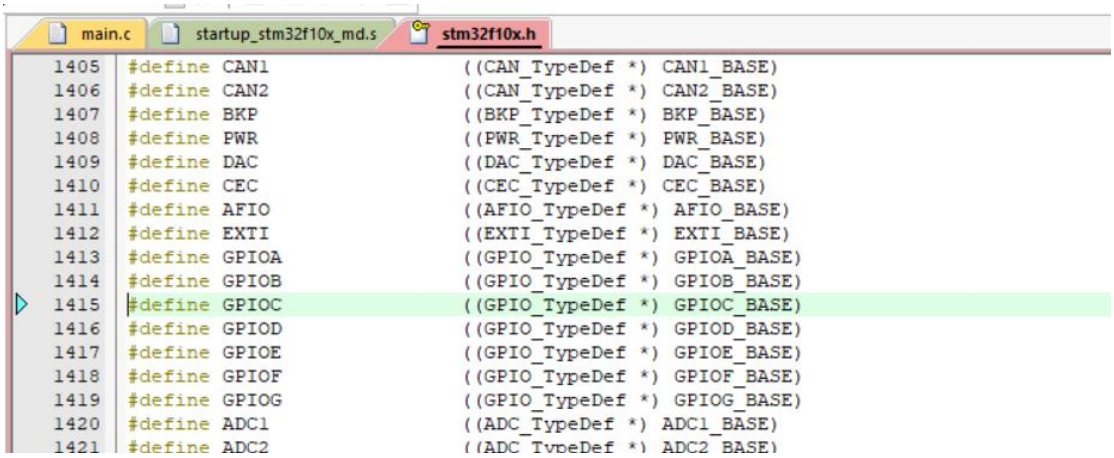
可以看出，实现了 LED 与按键状态改为反相输出，当按键状态为 0 时，输出的为 1；反之则为 0。

5 使用 STM32 标准外设库+直接操作寄存器

这个实验和上一个不同的是，这个引入了 STM32 标准库：

```
#include "stm32f10x.h"
```

上个实验操作的寄存器的地址在这个库中已经定义好，例如选择 GPIOC，并 go to Definition，可以看到定位到了这个库内部，查看参数，发现和上一题自己定义的一致，用库的好处就是省去了自己查找地址的麻烦。



```
1405  #define CAN1              ((CAN_TypeDef *) CAN1_BASE)
1406  #define CAN2              ((CAN_TypeDef *) CAN2_BASE)
1407  #define BKP               ((BKP_TypeDef *) BKP_BASE)
1408  #define PWR               ((PWR_TypeDef *) PWR_BASE)
1409  #define DAC               ((DAC_TypeDef *) DAC_BASE)
1410  #define CEC               ((CEC_TypeDef *) CEC_BASE)
1411  #define AFIO              ((AFIO_TypeDef *) AFIO_BASE)
1412  #define EXTI              ((EXTI_TypeDef *) EXTI_BASE)
1413  #define GPIOA             ((GPIO_TypeDef *) GPIOA_BASE)
1414  #define GPIOB             ((GPIO_TypeDef *) GPIOB_BASE)
1415  #define GPIOC             ((GPIO_TypeDef *) GPIOC_BASE)
1416  #define GPIOD             ((GPIO_TypeDef *) GPIOD_BASE)
1417  #define GPIOE             ((GPIO_TypeDef *) GPIOE_BASE)
1418  #define GPIOF             ((GPIO_TypeDef *) GPIOF_BASE)
1419  #define GPIOG             ((GPIO_TypeDef *) GPIOG_BASE)
1420  #define ADC1              ((ADC_TypeDef *) ADC1_BASE)
1421  #define ADC2              ((ADC_TypeDef *) ADC2_BASE)
```

```
#define AFIO_BASE             (APB2PERIPH_BASE + 0x0000)
#define EXTI_BASE             (APB2PERIPH_BASE + 0x0400)
#define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
#define GPIOB_BASE            (APB2PERIPH_BASE + 0x0C00)
#define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)
#define GPIOD_BASE            (APB2PERIPH_BASE + 0x1400)
```

下面修改各个寄存器：

在 Button_Init 函数中，由于从 GPIO13 改成了 GPIO3，需要将将 CRH 换成 CRL，

同时将 BUTTON_PIN - 8 改成 BUTTON_PIN。

```
void Button_Init(void) {
  /* Enable GPIOC clock           */
  RCC->APB2ENR |= (1ul << 4);

  /* Configure Button (PC.13) pins as input */
  GPIOC->CRL &= ~(0xFul << (4*(BUTTON_PIN))); //Clear Bit[15..12]
  GPIOC->CRL |=  (0x4ul << (4*(BUTTON_PIN))); //Set Bit[15..12] = 0b0100
}
```

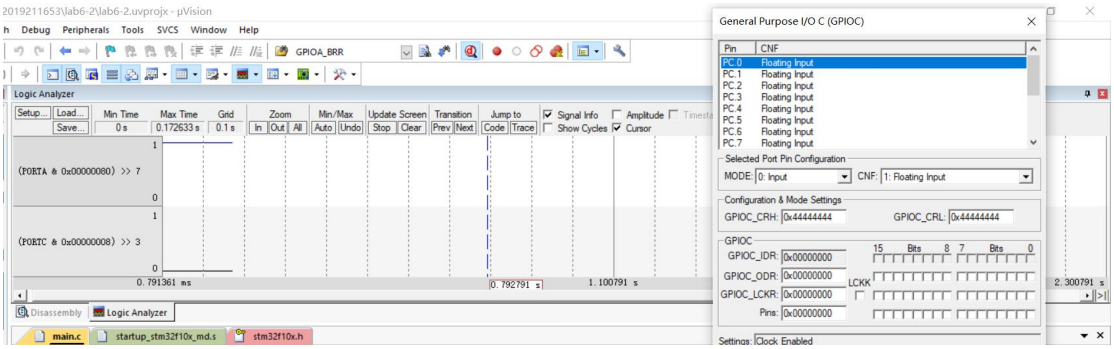
其他的操作和上一个实验相同，只需要直接用寄存器名称就可以：

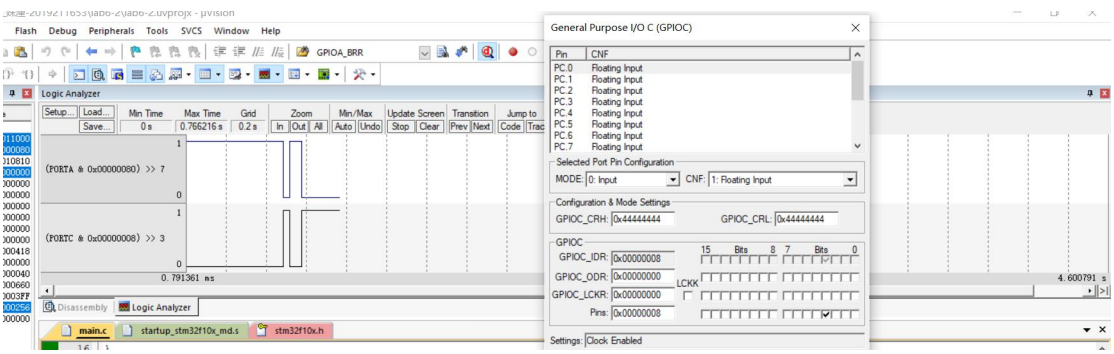
```
int Button_Read(void) {
  return ((GPIOC->IDR >> BUTTON_PIN) & 1);
}

void LED_Init(void) {
  /* Enable GPIOA clock            */
  RCC->APB2ENR |= (1ul << 2);

  /* Configure LED (PA.5) pins as output */
  GPIOA->CRL &= ~(0xFul << (4*LED_PIN)); //Clear Bit[31..26]
  GPIOA->CRL |=  ( 0x1ul << (4*LED_PIN)); //Set Bit[31..26] = 0b0001
}
```

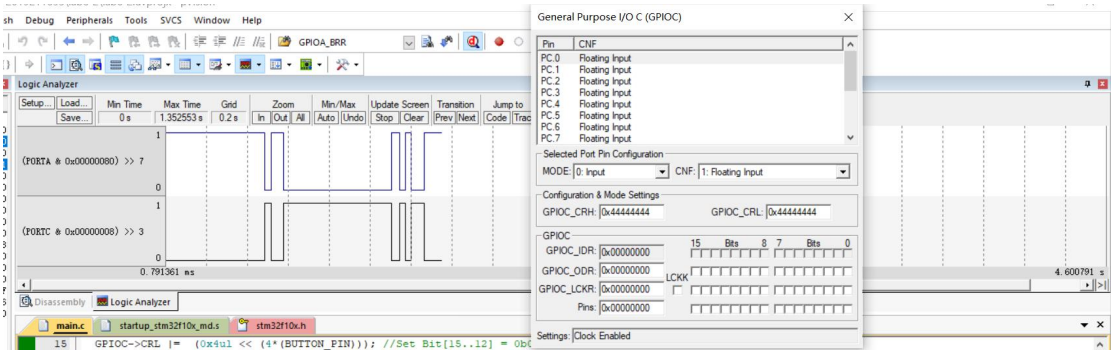
为了修改使得 LED 与按键状态改为反相输出，将 state 从 0 改成了 1

```
void LED_Write(int state) {
    //修改使得反方向跳变，将state从1改成0即可
    if(state == 1) {
      GPIOA_BSRR = (1 << LED_PIN) << 16;
    } else {
      GPIOA_BSRR = (1 << LED_PIN);          //on
    }
}
```

编译运行，结果如下，可以发现是 LED 与按键状态改为反相输出，且按键是由 pins3 操作的：

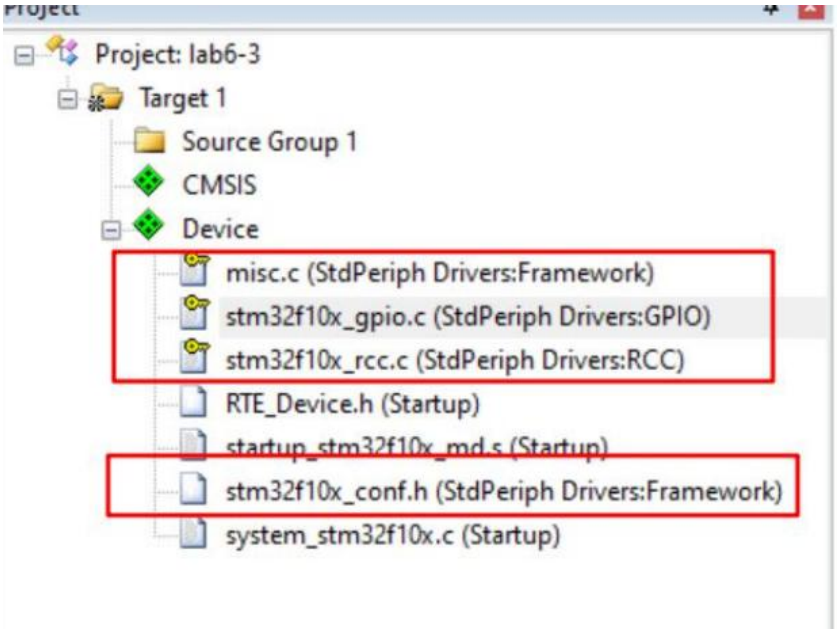






6 使用 STM32 标准外设库+API 操作寄存器。

本次实验也同样使用了 STM32 标准外设库，这次则是用 API 来操作寄存器，在创建工程时加上了 Framework、RCC、GPIO 的代码和配置文件 stm32f10x_conf.h：



这时各个对寄存器的操作已经在库函数中封装好了，若需要对寄存器进行操

作则需要查看各个函数在库中的定义，需要什么参数，又返回什么内容。这时对底层寄存器的操作过程其实对编写代码者来说是不透明的，仅知道调用函数可以实现相关功能而已。

首先定下需要操作的端口号：

```
#include "stm32f10x.h"

#define LED_PIN GPIO_Pin_7  //PA.7

#define BUTTON_PIN GPIO_Pin_3 //PC.3
```

接下来初始化需要操作的端口，以 GPIOC 端口为例：

首先查看 GPIO_Init 函数，需要传入 GPIOx，GPIO_InitTypeDef 两个类型的指针参数。

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
{
  uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
  uint32_t tmpreg = 0x00, pinmask = 0x00;
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
  assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));

/*---------------------------- GPIO Mode Configuration -----------------------*/
  currentmode = ((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x0F);
  if ((((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
  {
    /* Check the parameters */
    assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
    /* Output mode */
    currentmode |= (uint32_t)GPIO_InitStruct->GPIO_Speed;
  }
/*---------------------------- GPIO CRL Configuration ------------------------*/
  /* Configure the eight low port pins */
  if (((uint32_t)GPIO_InitStruct->GPIO_Pin & ((uint32_t)0x00FF)) != 0x00)
  {
    tmpreg = GPIOx->CRL;
    for (pinpos = 0x00; pinpos < 0x08; pinpos++)
    {
```

再查看 GPIO_InitTypeDef 的定义，发现里面包含引脚、模式、频率的信息，将引脚和模式设置成我们需要的信息即可：

```
typedef struct
{
  uint16_t GPIO_Pin;             /*!< Specifies the GPIO pins to be configured.
                                      This parameter can be any value of @ref GPIO_pins_define */

  GPIOSpeed_TypeDef GPIO_Speed;  /*!< Specifies the speed for the selected pins.
                                      This parameter can be a value of @ref GPIOSpeed_TypeDef */

  GPIOMode_TypeDef GPIO_Mode;    /*!< Specifies the operating mode for the selected pins.
                                      This parameter can be a value of @ref GPIOMode_TypeDef */
}GPIO_InitTypeDef;
```

具体代码如下：

```
void Button_Init(void) {
  /* Enable GPIOC clock                */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  /* Configure Button (PC.3) pins as input */
  GPIO_InitTypeDef gpio;
  gpio.GPIO_Pin = BUTTON_PIN;
  gpio.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOC, &gpio);
}
```

用同样的方法操作 GPIOA.7 的端口操作：

```
void LED_Init(void) {
  /* Enable GPIOA clock                */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  /* Configure LED (PA.7) pins as output */
  GPIO_InitTypeDef gpio;
  gpio.GPIO_Pin = LED_PIN;
  gpio.GPIO_Mode = GPIO_Mode_Out_PP;
  gpio.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_Init(GPIOA, &gpio);
}
```

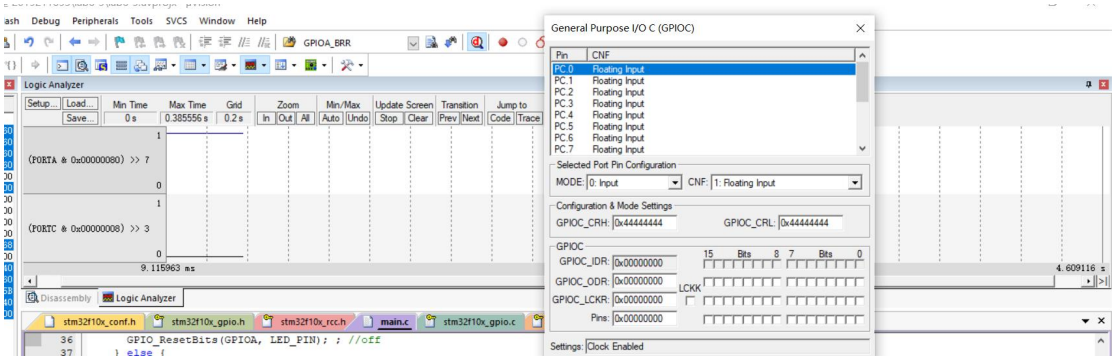
下面读取按钮的值，可以直接通过调用 GPIO_ReadInputDataBit()函数实现，查看定义得：

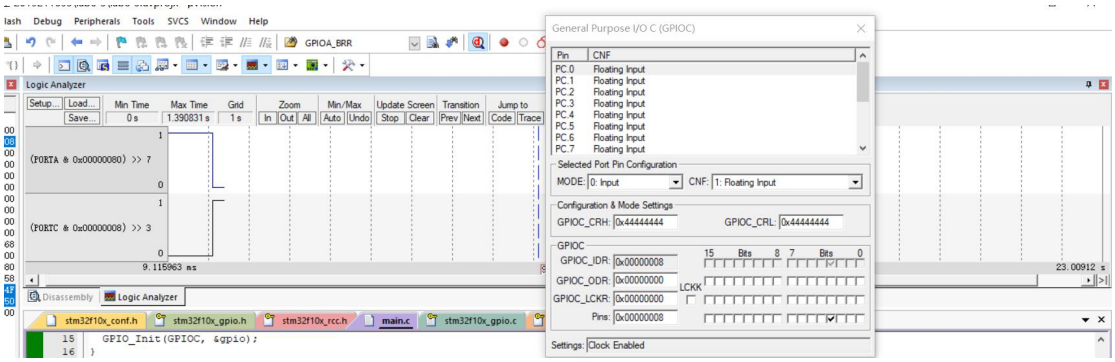
```
281  uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
282  {
283    uint8_t bitstatus = 0x00;
284
285    /* Check the parameters */
286    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
287    assert_param(IS_GET_GPIO_PIN(GPIO_Pin));
288
289    if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)Bit_RESET)
290    {
291      bitstatus = (uint8_t)Bit_SET;
292    }
293    else
294    {
295      bitstatus = (uint8_t)Bit_RESET;
296    }
297    return bitstatus;
298  }
```

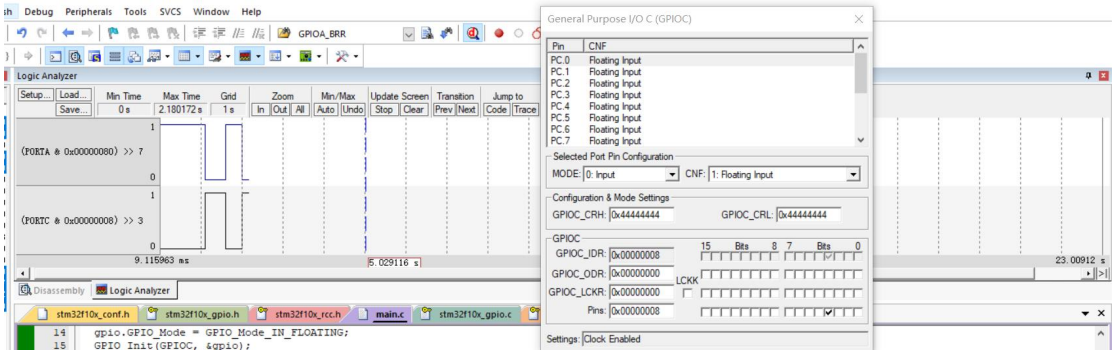
有 GPIO 端口和引脚两个参数，设成我们需要的值就可以：

```
int Button_Read(void) {
  return GPIO_ReadInputDataBit(GPIOC, BUTTON_PIN);
}
```

运行出的结果如下：







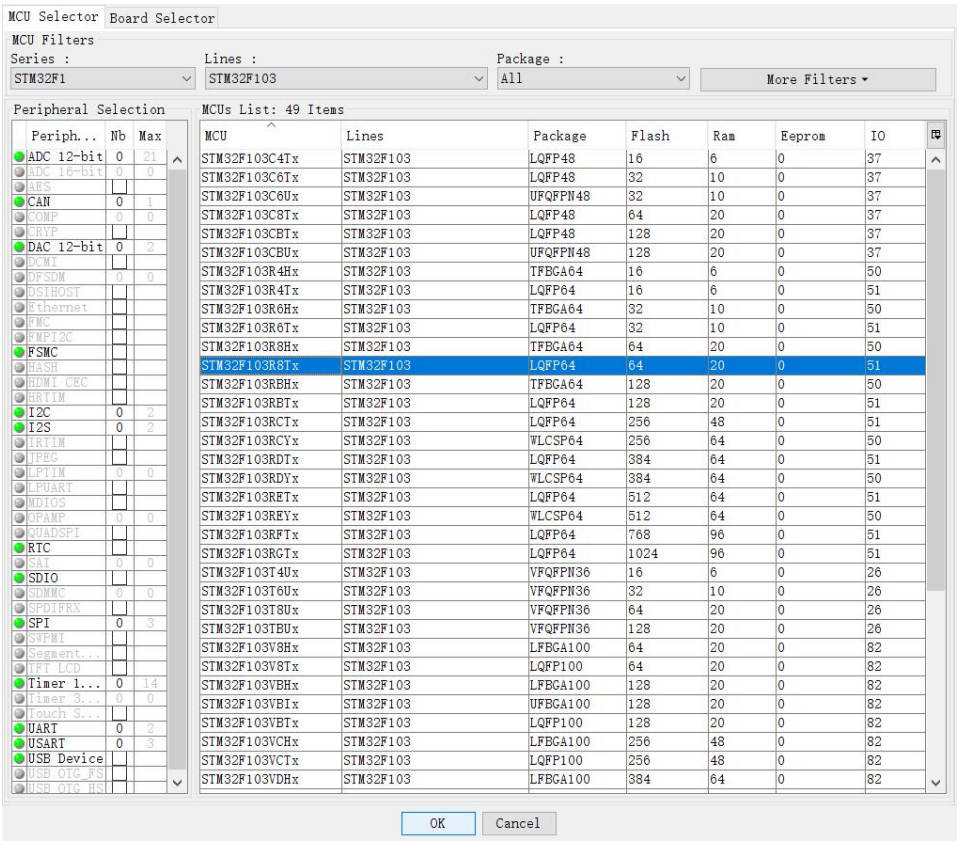
编译运行，结果如下，可以发现是 LED 与按键状态改为反相输出，且按键是由 pins3 操作的。

7 使用 STM32CubeMX 生成代码+API 操作寄存器

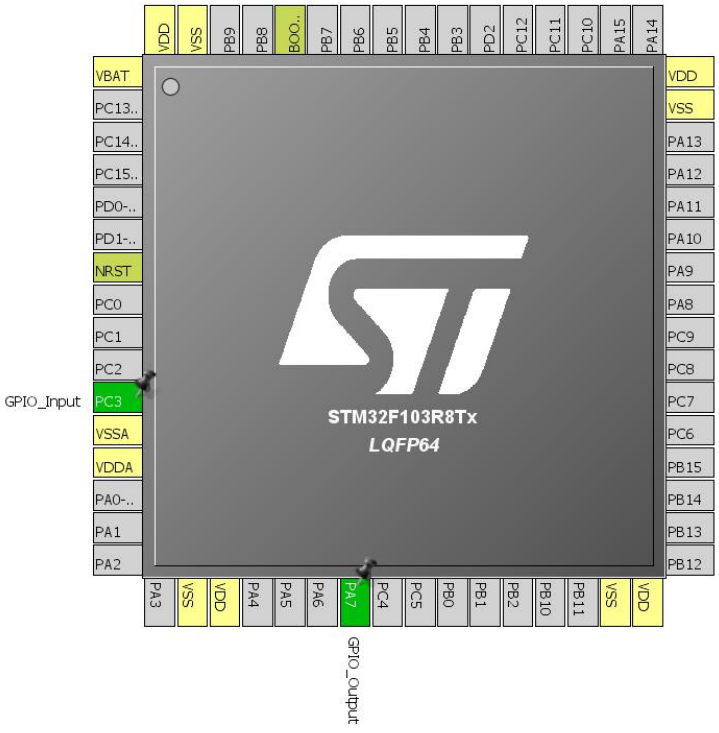
这种情况直接使用 STM32CubeMX 生成代码，这种方式的好处是，可以用可视化的方式直接定好引脚的功能，并自动生成好代码，且在代码中已经完成 GPIO 的初始化，不需要我们手动初始化，操作起来更加方便。

同样不足之处也在此，如何进行初始化这个过程我们并不了解。

在 STM32Cube32MX 中创建工程，选择芯片 STM32F103RBTx



下面配置芯片引脚。将 PA7 设为输出模式，PC3 为输入模式。



查看生成的代码，可以发现已经完成了初始化的过程，查看初始化函数，可以看到在这里面完成了对 GPIOA.7 和 GPIOC.3 的配置。

```
150 */
151 static void MX_GPIO_Init(void)
152 {
153
154   GPIO_InitTypeDef GPIO_InitStruct;
155
156   /* GPIO Ports Clock Enable */
157   __HAL_RCC_GPIOC_CLK_ENABLE();
158   __HAL_RCC_GPIOA_CLK_ENABLE();
159
160   /*Configure GPIO pin Output Level */
161   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, GPIO_PIN_RESET);
162
163   /*Configure GPIO pin : PC3 */
164   GPIO_InitStruct.Pin = GPIO_PIN_3;
165   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
166   GPIO_InitStruct.Pull = GPIO_NOPULL;
167   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
168
169   /*Configure GPIO pin : PA7 */
170   GPIO_InitStruct.Pin = GPIO_PIN_7;
171   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
172   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
173   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
174
175 }
```

下面编辑代码，在 main.c 的 while(1)循环里增加下述代码：

```
85   /* USER CODE BEGIN WHILE */
86   while (1)
87   {
88   /* USER CODE END WHILE */
89     GPIO_PinState key = HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_3);
90     key = !key;
91     HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, key);
92   /* USER CODE BEGIN 3 */
93
94   }
```

其中 key 是 HAL_GPIO_ReadPin 函数返回值，查看该函数的定义：

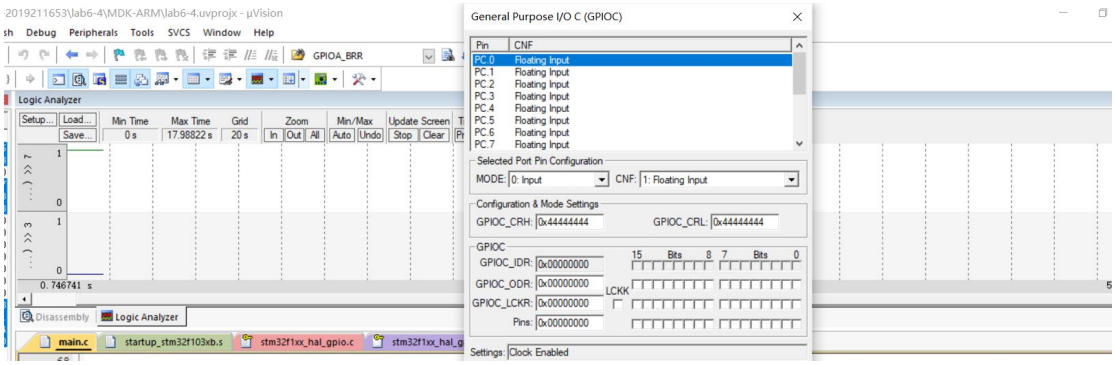
```
446   */
447 GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
448 {
449   GPIO_PinState bitstatus;
450
451   /* Check the parameters */
452   assert_param(IS_GPIO_PIN(GPIO_Pin));
453
454   if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)GPIO_PIN_RESET)
455   {
456     bitstatus = GPIO_PIN_SET;
457   }
458   else
459   {
460     bitstatus = GPIO_PIN_RESET;
461   }
462   return bitstatus;
463 }
464
```

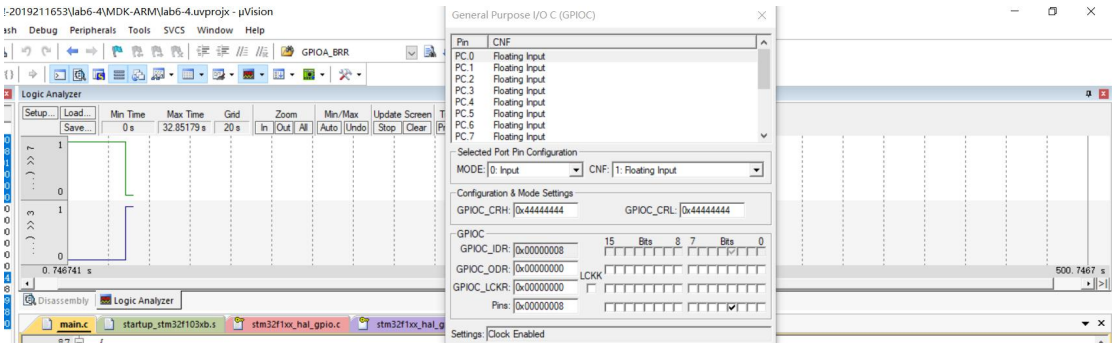
可以看出返回的是 GPIO_PIN_SET 或 GPIO_PIN_RESET 两个 bitstatus 的状态，再查看参数定义：

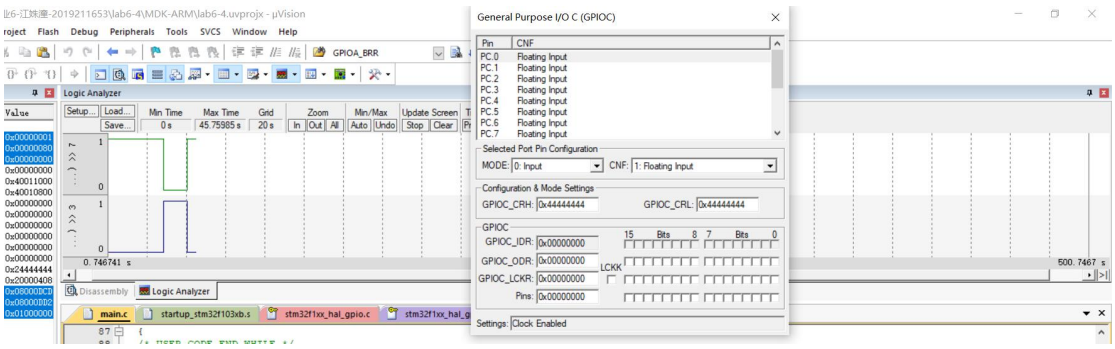
```
  */
typedef enum
{
  GPIO_PIN_RESET = 0,
  GPIO_PIN_SET
}GPIO_PinState;
```

可以发现，reset 和 set 分别对应 0 和 1，这和我们日常的理解也是一致的，要实现反向控制，可以将返回的 key 取个非，这样传入 HAL_GPIO_WritePin 的参数 key 和实际（模拟出的）按键状态相反。

编译运行，结果如下：







8 实验总结

本实验使用 4 种方式来实现了调用 2 个 GPIO 口来输入输出，能够读取按键的状态控制 LED 的亮灭。

本次实验的 4 种方式从上到下，自动化和抽象化程度越高，编程越容易，但是离底层的操作原理也越远，与库的绑定也越紧密，冗余代码也越多。从一开始的不使用 STM32 标准外设库并直接操作寄存器，然后用 STM32 标准外设库来简化寻址的过程，再用 API 操作寄存器，包装底层对寄存器的具体操作过程。最后又使用 STM32CubeMX 这个工具来生成代码，非常简单直观，对于编写者来说只需要考虑代码编写过程就可以了，而不用考虑硬件的初始化过程。

本次实验带给我印象最深的就是第一种方式了，这是最底层的方式，直接找到寄存器的地址和各位代表的含义，直接用位操作来修改值。好处是非常简单直观，不足是都是对位进行操作，需要非常小心，防止不小心操作错位，这也是我前面结果一直无法显示的原因。一旦理解了底层的操作，后面实现起来就非常简单了。这个实验用 4 种不同的方式自底向上地帮助我了解了 STM32 芯片的构造和操作过程，收获非常丰富。