不讲理论的STM32教程

进阶部分

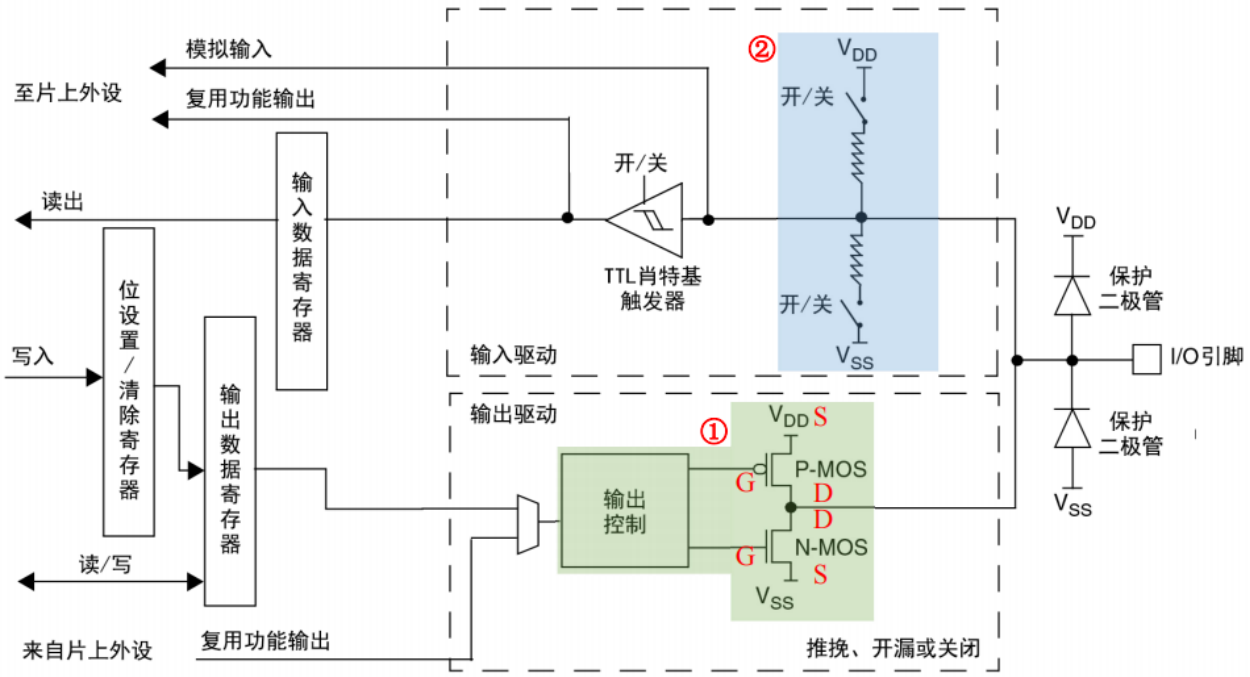
第六章：GPIO

1.关于GPIO

GPIO（General-Purpose IO ports，通用输入/输出接口），通俗来说就是常用引脚，可以控制引脚的高低电平，对其进行读取或者写入，STM32F103C8T6一共有48个引脚，除去电源引脚、晶振时钟引脚、复位引脚、启动选择引脚、程序下载引脚（大部分为最小系统必须引脚），剩下的则是GPIO引脚。

STM32将这些众多按GPIOx（x为A、B、C等）分组，每组包含16个引脚，编号为0~15。STM32F103C8T6只有2组GPIO，每组16个引脚，即32个GPIO引脚，一般讲GPIOA_1引脚简称为PA1，其他引脚也是同样的叫法。

下图 为STM32F103系列GPIO的基本结构，左侧连接MCU内部，中间上半部分为输入，中间下半部分为输出，右侧为MCU引出的外设I/O引脚



这里不会对原理图进行讲解，太枯燥了，反正主要就是非为输入输出两种情况，而两种情况下有分成了8种工作模式，感兴趣的小伙伴可以自己学习一下他是怎么通过电路实现的，下面我们只会对八种模式进行讲解

输出模式

八种模式分别为：

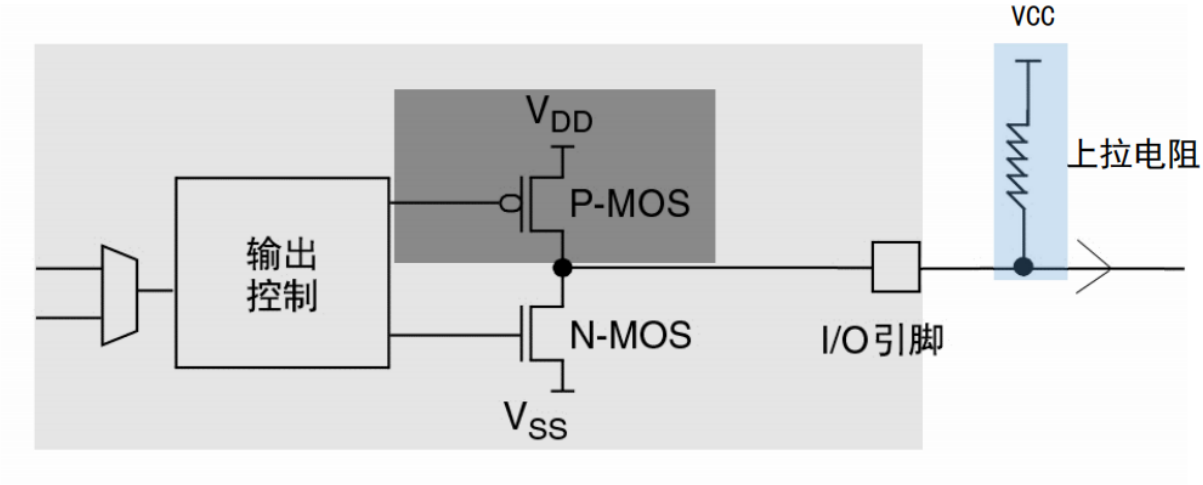
输出模式有四种：推挽输出、开漏输出、复用推挽输出、复用开漏输出

输入模式有四种：上拉输入、下拉输入、浮空输入、模拟输入

下面我们来单独讲解我们的八种模式：

推挽输出（Push-Pull，PP）：输出高电平时就是直接接到了我们的单片机的到VDD（3.3V），输出高电平时就是直接接到了我们的单片机的到Vss（0V）,最直接的输出方式，让“输出控制”变为了VDD/Vss输出，使得输出电流增大，提高了输出引脚的驱动能力，提高了电路的负载能力和开关的动作速度

开漏输出（Open-Drain，OD）：推挽输出是直接连接VDD(3.3V)和Vss(0V)进行输出，开漏输出则不同，它只连接了我们的Vss(0V)，如果模式为开漏输出的话，正常情况他只能输出低电平，无法输出高电平，如果想要输出高电平怎么办呢？



如图所示，我们需要在外部电路连接上拉电阻，这样就可以输出高电平了，并且这样的好处就是，我们的高电平将会是Vcc的电压，是我们自己可以控制的电压，，从而实现了电平转换的效果

复用推挽/开漏输出（Alternate Function，AF）：这两个放在一起介绍，GPIO引脚除了作为通用输入/输出引脚使用外，还可以作为片上外设（USART、I2C、SPI等）专用引脚，即一个引脚可以有多种用途，但同一时刻一个引脚只能使用复用功能中的一个。当引脚设置为复用功能时，可选择复用推挽输出模式或复用开漏输出模式，在设置为复用开漏输出模式时，需要外接上拉电阻。

上拉输入（Input Pull-up）：顾名思义，设置为此选项，外部没有信号传入时，默认为高电平，该模式的典型应用就是外接按键，当没有按键按下时候，引脚为确定的高电平，当按键按下时候，引脚电平被拉为低电平

下拉输入（Input Pull-down）：和上拉输入差不多，只不过没接到信号传入时，默认为低电平

浮空输入（Floating Input）：此时I/O引脚浮空，读取的电平是不确定的，外部信号是什么电平，引脚就输入什么电平，芯片复位上电后，默认为浮空输入模式

模拟输入（Analog mode）：引脚信号直接连接模拟输入，实现对外部信号的采集，可以收集0~Vss的电压值

GPIO除了有输出模式以外，还有输出速度需要了解，接下来介绍输出速度

输出速度

STM32的I/O引脚工作在输出模式下时，需要配置I/O引脚的输出速度。该输出速度不是输出信号的速度，而是I/O口驱动电路的响应速度。

STM32提供三个输出速度：2MHz、10MHz、50MHz。实际开发中需要结合实际情况选择合适的相应速度，以兼顾信号的稳定性和低功耗。通常，当设置为高速时，功耗高、噪声大、电磁干扰强；当设备为低速时，功耗低、噪声小、电磁干扰弱。

通常简单外设，比如LED灯、蜂鸣器灯，建议使用2MHz的输出速度，而复用为I2C、SPI等通信信号引脚时，建议使用10MHz或50MHz以提高响应速度。

此外GPIO还有一些其他的知识，这里暂时不往外延申，我们等遇到在进行解释，目前只需要知道这些

2.GPIO的使用

说完了基本的GPIO，我们现在开始简单的做一个小项目来使用一下GPIO的输入以及输出，大概熟悉一下怎么使用

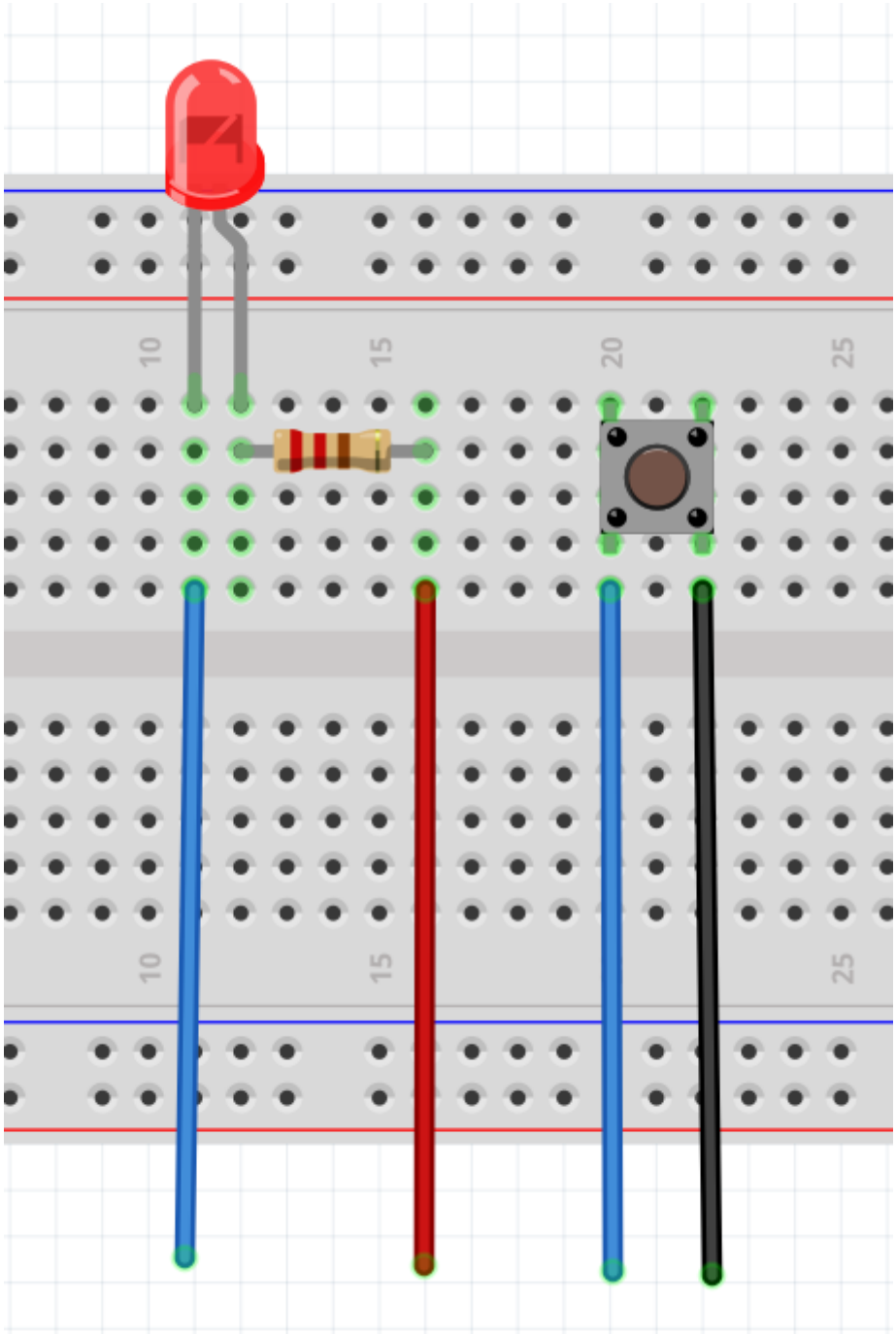
项目：

按键控制LED的亮灭

材料：

面包板，LED小灯泡一个，1K电阻一个，独立按键一个，导线若干

原理图：



大概的原理图就是这样，蓝色的连接我们的GPIO引脚，这里阿熊使用PA1，和PA0，分别连接LED和独立按键，红色连接我们的VCC，黑色连接GND，相信学过51的小伙伴们肯定知道，我们的按键和LED的电路并不是和普通电路一样，而是独立分开的，当我们的独立按键按下时，我们的PA0引脚就会收到低电平，然后做出对应的操作让我们的LED亮起来

思路分析：

初始化GPIO：

PA1连接LED肯定就是输出模式，并且是推挽输出

PA0连接独立按键，我们将其初始化为上拉输入，前面也说过，按键一般都是使用上拉输出，我们没有按下按键时，默认都为高电平，只有按下按键时，PA0直接连接GND，则会接收到低电平信号，这样可以减少外部因素干扰

逻辑书写：

当我们按下按键，也就是PA0为低电平时，我们让我们的LED亮起，再次按下按钮，我们让我们的LED熄灭，以此往复

所以代码主要的逻辑应该是：

判断（按键按下）：

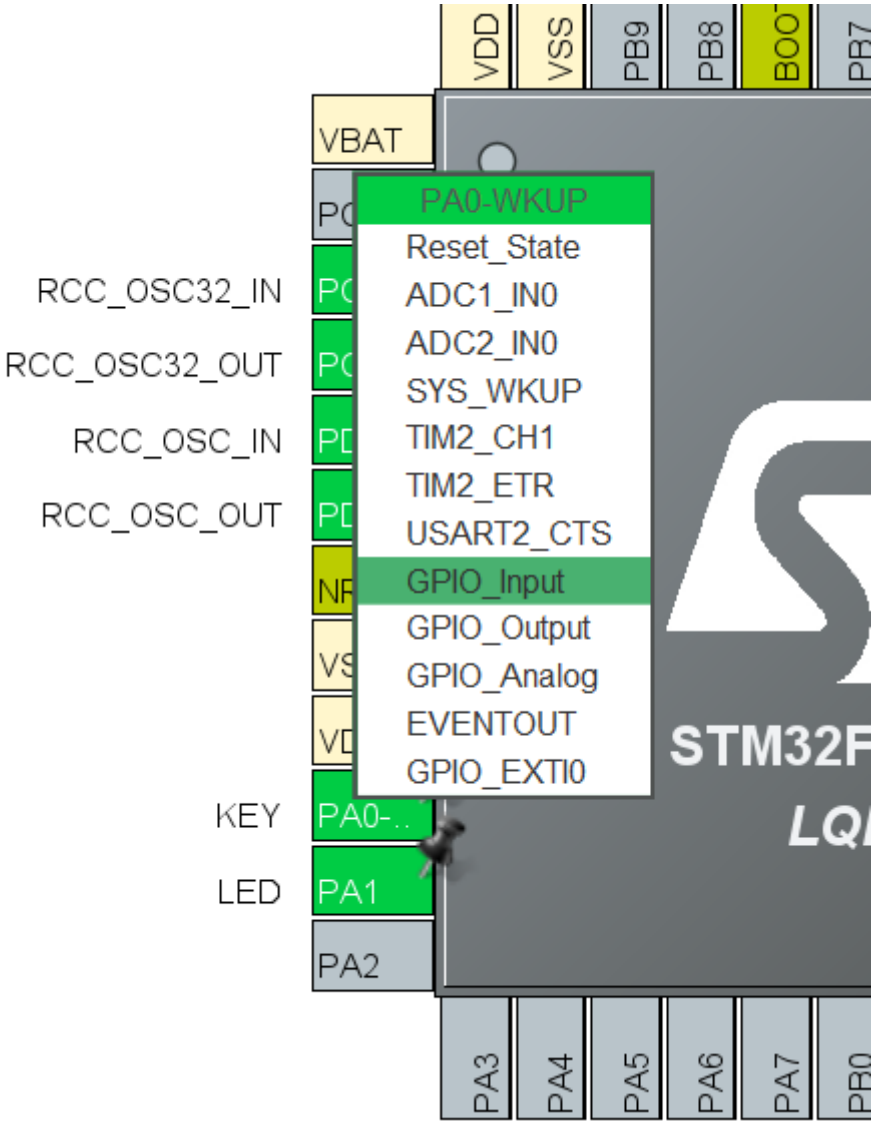
LED状态取反

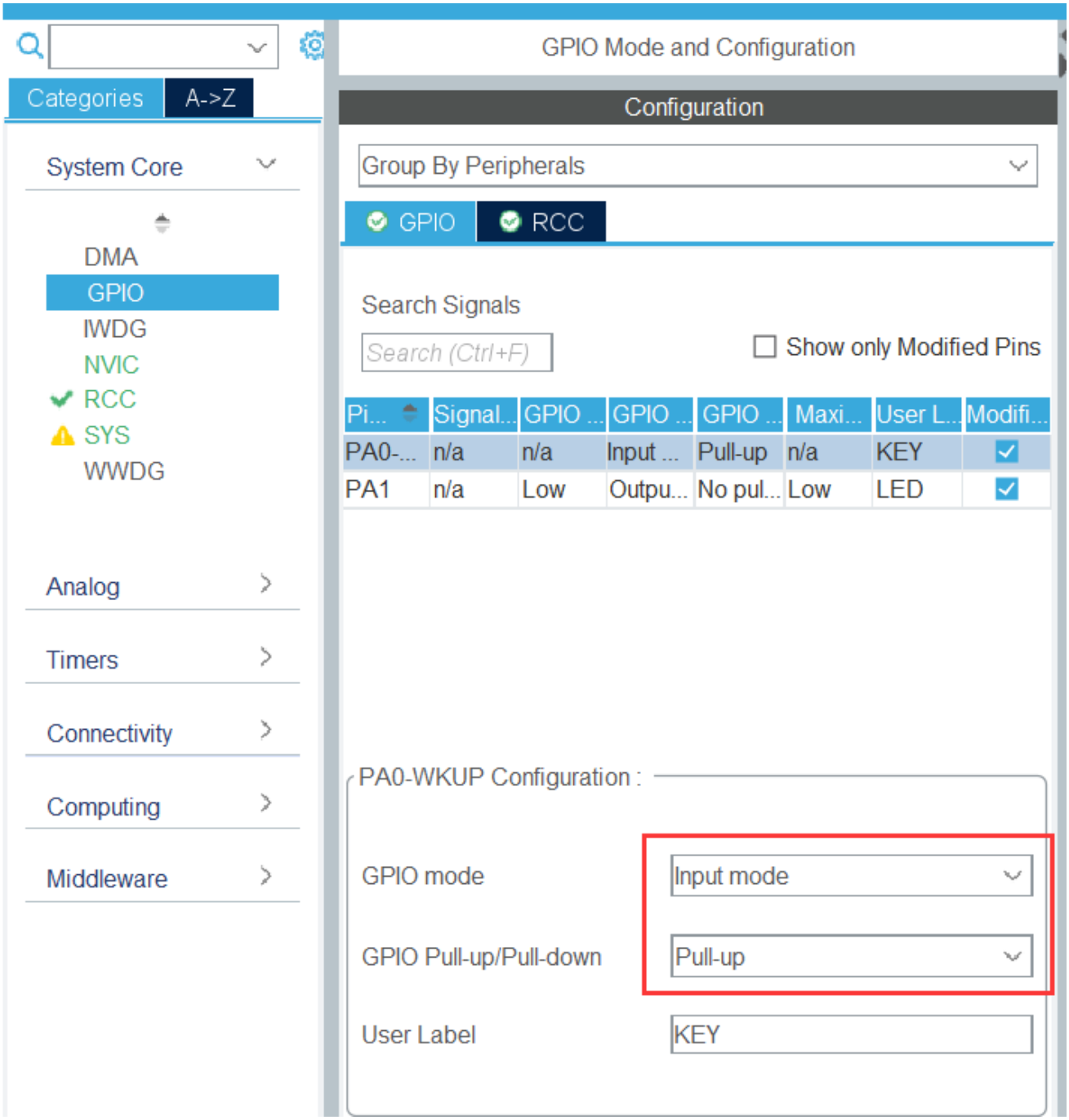
大体就是这样的一个流程，现在我们开始新建项目以及写代码

项目建立

新建项目这里就为大家省略了，如果不会的小伙伴可以去看一下我们的前几章内容

我们要注意的是我们的项目配置


VDD
VSS
PB9
PB8
VBAT
PA0-WKUP
Reset_State
ADC1_IN0
ADC2_IN0
SYS_WKUP
TIM2_CH1
TIM2_ETR
USART2_CTS
GPIO_Input
GPIO_Output
GPIO_Analog
EVENTOUT
GPIO_EXTI0
RCC_OSC32_IN
RCC_OSC32_OUT
RCC_OSC_IN
RCC_OSC_OUT
KEY
LED
PA1
PA2
PA3
PA4
PA5
PA6
PA7



这里是将我们的PA0-KEY引脚设置为了输入模式，并且使用的是我们的上拉输入，一般情况按键都是使用上拉输入

然后就是我们PA1-LED的设置，这里就不过多赘述了，不记得可以回头看一下前面的内容

然后就是时钟设置，前面我们也讲过，这里也是直接搬运过来就好

然后我们开始最重要的代码书写

代码书写：

```
#include "main.h"
#include "gpio.h"

void SystemClock_Config(void);
```

```c
int main(void)
{
  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
      if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0){//判断按键是否按下
          HAL_Delay(20);//延时消抖
          if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0){//再次判断按键是否
按下
              HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, 1-
HAL_GPIO_ReadPin(LED_GPIO_Port,LED_Pin));//小灯泡状态取反
          }
          while(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0);//按键是否松开
      }
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the RCC Oscillators according to the specified
parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
```

```
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line
number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
```

```
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and
line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

这是main.c中的全部内容，现在讲解一下我们的逻辑部分

```
while (1)
  {
    /* USER CODE END WHILE */
      if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0){//判断按键是否按下
          HAL_Delay(20);//延时消抖
          if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0){//再次判断按键是否
按下
              HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, 1-
HAL_GPIO_ReadPin(LED_GPIO_Port,LED_Pin));//小灯泡状态取反
          }
          while(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0);//按键是否松开
      }
  }
```

首先是：if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0)判断一下我们的按键是否按下，这里我们使用的函数是HAL库中GPIO部分里的Read Pin()函数，需要传入两个参数，第一个是传入我们的引脚端口，这里我们就直接填写系统宏定义好的KEY_GPIO_Port，然后第二个参数是传入我们的引脚号，这里我们也直接使用系统写好的宏定义KEY_Pin，然后当他==0时，代表已经按下按键

第二部： HAL_Delay(20);//延时消抖

这里也是使用了我们HAL库的延时函数，传入需要延时的时间，单位是毫秒

接着是：if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0)再次判断按键是否按下，和第一步是一个原理

接着就是第四步：

HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, 1-HAL_GPIO_ReadPin(LED_GPIO_Port,LED_Pin));

这里我们使用的函数是HAL库中GPIO部分里的WritePin()函数，可以手动写入我们引脚状态，也就是改变小灯泡的状态，这里我们为了让我们的LED可以每次按下按键状态取反，先使用HAL库中GPIO部分里的Read Pin()函数去读取他的状态，然后我们就使用1-HAL_GPIO_ReadPin(LED_GPIO_Port,LED_Pin)，就可以让他进行取反，这里方法不唯一，小伙伴们可以使用自己的想法，阿熊这里只是一个示例

最后：while(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0);//按键是否松开

判断我们的按键是否松开就好了，防止按键多次触发

好了，我们就已成完成了所有代码书写，虽然比较简单，但是我们使用了GPIO的最常用的输入输出模式，对GPIO有了一个基本的了解，下期阿熊将会为大家介绍中断系统，下期见！