[STM32]STM32F407系列教程之二，gpio输入输出实验

原创

BACKHEART 于 2019-06-20 23:49:04 发布 6094 收藏 29

分类专栏： # STM32系列 文章标签： STM32F407 嵌入式 单片机 例程

本文链接：https://blog.csdn.net/weixin_39869569/article/details/93141430

文章目录

一、实验准备

1.模板demo

2.板级支持包

二、实验目的

三、实验前理论准备

二、C语言那些被遗忘的语法

1.结构体

2.位操作

3.移位操作

4.数据的存放

三、开始实验

1.添加需要的ST固件库

2.添加需要的BSP文件

3.在main.h中注册

4.设置h文件的包含路径

5.主函数中书写测试代码

四、分析BSP配置代码，学习GPIO

五、问题思考

1.以同样的方式看一下BSP_KEY_InitConfig()的配置过程

2.打开”Bsp_led.c”文件，找到第24行

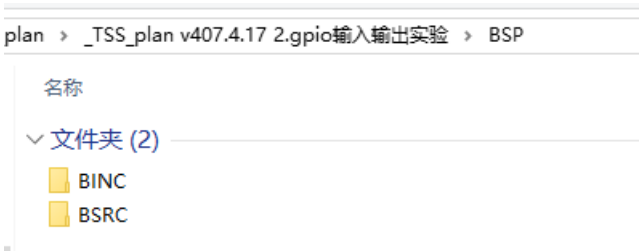
一、实验准备

1.模板demo

原因呢，我在第一讲中已经说过，费尽千辛万苦搭建了一个模板，流过多少泪、费劲多少事，只有亲自搭建过的才会体会到（第一讲我只是讲了组成，后续有机会，我将带大家亲手搭建一个），搭建完成后，备份压缩（玩过Linux的大概有点感触，配置了一个新环境，就得做个镜像，哪天系统崩了，就可以重装系统，这里也是一样的）。

2.板级支持包

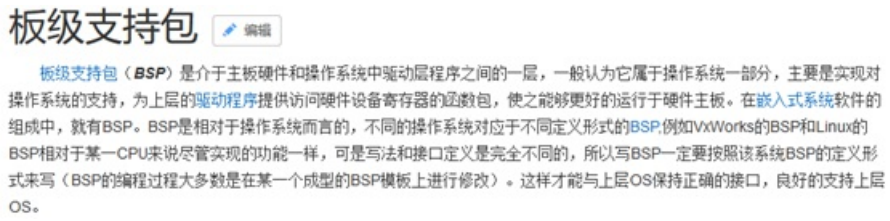
就是在我的工程文件中，所有以”Bsp_”开头的文件，位于BSP文件夹下：



BSP文件夹下有两个文件夹：BINC、BSRC。一个存放h文件，一个存放c文件，是可以放到一个文件中，我只是为分类能够明显点。

PS：关于这种分类，我在其他文件夹中也有类似的设计，有兴趣可自行查找，另外无论去不去查找，请记住这里的东西，我在本章后面还会用到这里的知识点。

另外介绍一下什么“板级支持包“：



百度到的一段介绍，本来BSP是用来实现对操作系统的支持，比如我们在使用PC时，有显示器、有键盘、有鼠标，PC的操作系统，比如Windows、Linux系统，它们只是用来管理这些资源，向用户或者应用提供使用接口（如果你《大学计算机基础》这门课还记得的话）操作系统通过BSP来访问硬件设备。单片机到现在（起码到我们正在学习的这个初期阶段）还无法移植操作系统。但是我们也需要去使用外设的，如使用usart来跟PC进行通信，我们需要配置usart的工作模式。这里，BSP就涵盖了一些对硬件外设的操作。

总之，一句话，有了BSP，我们可以随心所欲的配置单片机上的外设，而不需要去关心如何去配置，当然，如果想研究比如usart的工作原理时，不好意思，读源码吧。

另外，补充一点，BSP是我在学习时候编写的，是基于ST公司的固件库（位于SYSLIB文件夹中）所做的二次开发。

二、实验目的

1. 配置gpio工作模式，使其可以读取一个外部按键的状态
2. 配置gpio工作模式，使其可以控制一个LED灯

三、实验前理论准备

打开”STM32F4xx中文参考手册.pdf”文件（我很反感使用中文手册，虽然我英语也不是很好，我建议可能的话尽量使用英文原版的datasheet，建议，建议），第7章，讲的是通用I/O（GPIO）。

和51单片机不同，STM32等一些相对高端的单片机，它的I/O口可以工作在输入、输出、复用、模拟，不仅如此，还可以上拉、下拉、浮空，没完呢，还有推挽模式、开漏模式。

当初我在学习时，那叫一个头疼，去**的（和谐音），好难。不过仔细想想，这么设计也是必须的，存在即合理，接下来，解释一下这些东西：

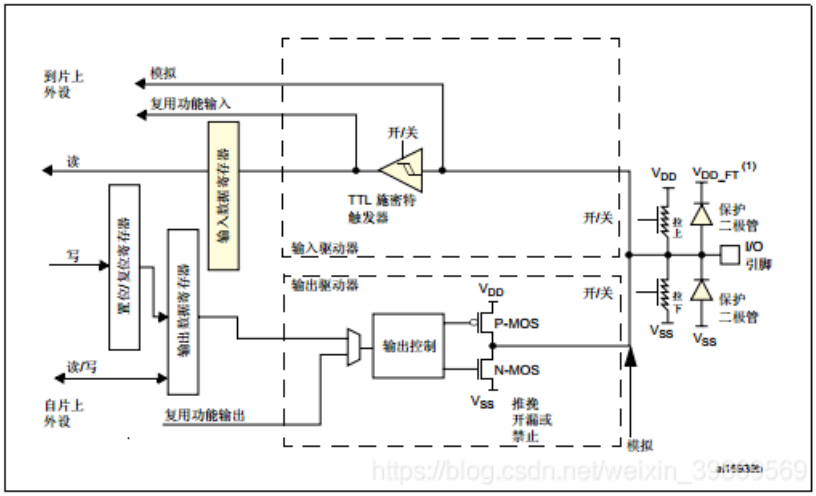
（都是大白话，不要见怪……）
输入：数字信号流入单片机引脚；
输出：数字信号从单片机引脚流出；
复用：有些引脚在被设计成特殊功能后，硬件会控制它，不需要用户在程序中操作；
模拟：模拟信号流入单片机引脚；

上拉：引脚通过一个电阻连接到VDD；
下拉：引脚通过一个电阻连接到GND；
浮空：引脚啥也不接；

推挽模式：可理解为普通；
开漏模式：漏极开路（理解它，需要一点模电知识，场效应管相关知识）；

用大白话解释了一遍，接下来，用几个硬件电路来从理论方面解释一下：

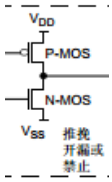
下图是”STM32F4xx中文参考手册.pdf”文件中的第176页图17：



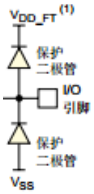
这张图，完整的展示了GPIO的工作方式，特别说明，这只是一个引脚，在STM32F407VE上，每16个为一组，像这样的一组，STM32F407VE有好几组（GPIOA、GPIOB、GPIOC…自己去找找吧，我就不多说了）。

PS：为了能够看懂这张图，我们先来解释一些特殊符号：

TTL施密特触发器：详细介绍请百度，我这里只告诉大家，它可以容忍信号的一些小波动（毕竟空间里的信号永远不会稳定，不服来辩）。

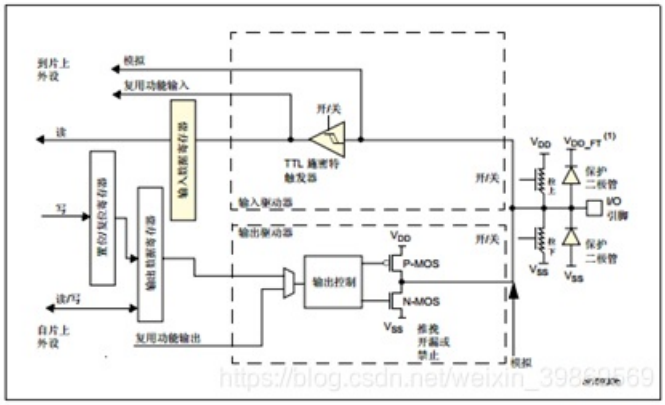


场效应管：详细介绍也请百度，理解为电子开关，左侧的两根线时控制信号（注意，这里是两个不同的场效应管！），另外，有小圆圈的表示低电平有效。



二极管保护电路：这个嘛，是为了容忍引脚输入5V信号的，因为STM32F407VE是3.3V的，没有这个电路的话，分分钟钞票送到马云爸爸兜里啊。当然，我们本章实验最高也就接入3.3V，所以，这个电路就没作用，既然没作用，也就不用考虑了。

然后，我们就可以分析了：
首先是输入模式：

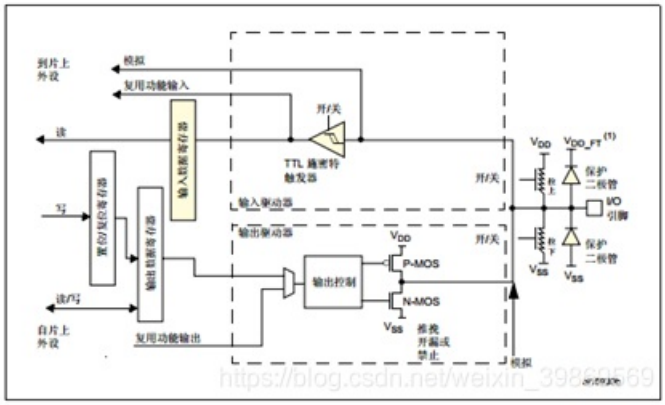


图中的红线就是输入信号的流动过程，下方的输出我们就不考虑了。

输入模式的信后从引脚传入，进来后，有一个上拉电阻和下拉电阻的选择，是可以控制的（拉上开启，接入上拉电阻；拉下开启接入，下拉电阻；都关闭，浮空；都开启，你丫是有病吧！），之后，信号可能传入模拟通道（如果开启了ADC设备，信号会被送到ADC中），然后就是TTL施密特触发器，经过TTL施密特触发器，信号相对稳定，一些小的波动被忽略了；然后，信号可以进入复用功能输入通道，如果开启了复用工作模式，相关硬件会来接管这个信号，硬件会自行判断；最后的最后，进入输入数据寄存器，到这里，信号就被存储到单片机里了，用户就可以访问这个所谓的输入数据寄存器来感知传入的信号是高电平还是低电平了。

PS：到底信号能否进入模拟通道、复用功能输入通道，是靠一个开关（学名叫寄存器）来控制的，想知道的自己查找喽。
另外，说一个冷门的知识点，不管模拟通道、复用功能输入通道是否打开，信号总会进入输入信号寄存器，也就是说，无论是什么信号（模拟也好、数字也好，还是复不复用都好），用户总是可以去主动读取这个输入信号。（注意，这个以后你一定会用到的，我以我这辈子的桃花做担保，用不到，我一辈子遇不到她！）

再来是输出模式：

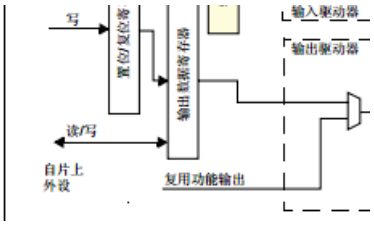


输出模式比较麻烦了，不过也不是没有突破点。

首先，按照我之前说的，不要看二极管保护电路（不知道那个是，倒回去看去！），然后，暂时（注意是暂时！）不看输入驱动器（再强调一起，暂时不看！）。好了，该说的都说了，来分析吧。

信号是从左到右，从单片机内部到引脚：首先是写，这个写需要注意一下，不是直接写进输出寄存器，而是通过向置位和复位寄存器（这是两个寄存器！）写数据，然后单片机硬件再把置位和复位寄存器里的数据转移到输出寄存器中**（PS：置位和复位寄存器的值是为1有效，如果置位寄存器的某一位为1，则这个位对应的引脚最终会输出高电平，为0无效；如果复位寄存器的某一位为1，则这个位对应的引脚最终会输出低电平，为0无效。建议大家仔细阅读这段话。）。**

为什么要这么麻烦？有道理的，往下看：



输出数据寄存器的数据还会来自片上外设，如USART1，会使用PA9和PA10来传输数据。设计成这么复杂，目的只有一个，在当USART1工作时，可以通过关闭从置位和复位寄存器到输出数据寄存器的通道，来防止用户对片上外设的输出操作产生干扰。

随着信号接着看，输出数据寄存器的数据到了输出控制那里，这里是很复杂的一块电路，不过只要知道这里会完成信号的调理、逻辑转换即可，还有的就是可以选择用不用后面的逻辑门（通过这个来选择工作在开漏模式还是推挽模式）。同时，复用功能的输出也会对输出进行控制。输出控制之后是开关门，通过了逻辑门，信号才会被转化为真正的信号（因为在此之前，信号可能只是存在于寄存器中的一个表示形式，好比U盘里的数据，虽然这个比喻不太恰当）；通过了逻辑门，就到了上拉、下拉电阻接入控制区，在之后，就是输出引脚了。

特别注意一下，右下角有一个模拟输出，不要忘了，毕竟还有个东西叫DAC！

PS：输出模式的工作原理基本说完了，不过我还要补充一点，我再讲输出模式时，说过暂时忽略这个模式下的输入通道，好了，现在要开始讲了：

首先贴几个截图，都可以在”STM32F4xx中文参考手册.pdf”文件的第7章找见：

7.3.9 输入配置

对 I/O 端口进行编程作为输入时：

- 输出缓冲器被关闭
- 施密特触发器输入被打开

7.3.10 输出配置

对 I/O 端口进行编程作为输出时：

- 输出缓冲器被打开：
  - 开漏模式：输出寄存器中的“0”可激活 N-MOS，而输出寄存器中的“1”会使端口保持高组态 (Hi-Z)（P-MOS 始终不激活）。
  - 推挽模式：输出寄存器中的“0”可激活 N-MOS，而输出寄存器中的“1”可激活 P-MOS。
- 施密特触发器输入被打开

7.3.11 复用功能配置

对 I/O 端口进行编程作为复用功能时：

- 可将输出缓冲器配置为开漏或推挽
- 输出缓冲器由来自外设的信号驱动（发送器使能和数据）
- 施密特触发器输入被打开

7.3.12 模拟配置

对 I/O 端口进行编程作为模拟配置时：

- 输出缓冲器被禁止。

可以发现，GPIO模式的工作方式无非就是输入、输出、模拟、复用模式，除了模拟模式时，其他所有的模式都开启了施密特触发器，说明了什么？说明了什么？说明了什么？说明了，这几种模式下，我们输出了信号之后，可以回读信号；也可以输出信号，然后读入信号而不需要切换工作模式，只需要读信号时对输出做些处理。
同时，可以根据这几个截图的信息重新再回顾一下输入\输出模式的工作原理（有点马后炮了，早不提醒！实际上，工作原理需要大家下点功夫去学习，3遍、5遍是少不了的）。

二、C语言那些被遗忘的语法

本来不想去说的，不过当下某学校的C语言这么课，讲的真是，***（和谐音），实在看不下去了，我就再啰嗦几句吧，自认为C语言很棒的，自动跳过：

1.结构体

首先结构体的定义：

```
struct
{
 uint8_t  param_a;
 uint16_t param_b;
} a_struct;
```

或：

```
struct newStruct
{
 uint8_t  param_a;
 uint16_t param_b;
};
struct newStruct a_struct;
```

或：

```
typedef struct
{
 uint8_t  param_a;
 uint16_t param_b;
} newStruct ;
newStruct a_struct;
```

PS：顺便问一个小问题，变量a_struct所占存储空间大小是多少？

还有，无比仔细琢磨三种方式的特点；

结构体是可以嵌套的：
比如：

```
struct tag_1
{
 struct tag_1 *ps;  //注意，只能是个指针！
 uint8_t  param_a;
 uint16_t param_b;
} ;
```

但是，这样就错误了：

```
typedef struct
{
 NODE *ps;   //错误，NODE还没有被定义！
 uint8_t  param_a;
 uint16_t param_b;
} NODE;
```

好了，暂时结构体先会这些，我主张的学习的总之就是：缺哪补哪，打哪指哪；

2.位操作

这个是在普通C中常常被忽略的东西，因为在PC端，我们学习C的目的是为了简化繁琐的科学计算，所以往往很少去纠结位与位的操作；
首先，回忆一下位操作符：&、|、~、^；（我记得就这4个吧，不好意思，忘了）；
然后，出几个题：

```
0xFF & 0xFE = ?
0xA0 | 0x0B = ?
0x11 ^ 0x11 = ?
…
```

PS：琢磨一下我出的三个题的特点，听我的，马上就会有用的。

最后，跟赋值运算符结合"="一下：于是就有了：&=、|=、^=；

3.移位操作

老实说，这个也是位操作的范畴，因为太常用了，我把它揪出来单独说：
首先，移位操作符有两个：>>和<<；
然后，还是出几个题：

```
a = 0x01;
b = a<<2;
b = ?

a = 0x80;
b = a>>6;
b = ?
…
```

4.数据的存放

数据在存储器中是如何存放的？

学过C语言的我们都知道，char、int、float、double，学习了C#之后，还有string等，学习了Python后，还有complex啥的（虽然人家python主打的概念是标签）等等。

那么，数据到底如何存放？

我想这么来说：首先，计算机能识别的，就是通电和断电（抽象点，就是0和1），这个概念叫二进制，一个二进制只有两个值，换句话说，只有两个状态，也就只能表示两个结果，这样的一个二进制称之为"1位数据（bit）"，表示的数据种类太少了。将8个二进制位捆绑在一起，这样就可以表示2^8种结果（不会的，回去复习高中的排列、组合），就可以表示最大数值为255的十进制整数，这样的8个二进制位称为"1个字节（Byte）"。想表示的数值再大怎么办，16个二进制呗，没错，不过别叫16个二进制位了，叫两个字节。再大呢，4个字节（以后会知道，4个字节叫字（Word），所以2个字节也叫半字（Half Word）），显而易见，8个字节叫双字（Double Word）。想表示小数怎么办，最开始是使用定点数，后来就使用浮点数（**PS：关于定点数和浮点数，不是我这里的重点，请自行百度**）。

所以，数据在存储器中是按"组"存放的，这个"组"，指的就是字节、半字、字、双字等。至于什么char、int、float，只是对这个"组"按照不同的需求进行别名：比如，若一个字节内存放的时ASCII码，那么这块数据的类型就是char；当然，若使用两个字节存放一个有符号的整数，那么这块数据的类型就是int；其他的也类似。所以，要明白，所谓的类型，只是对数据类型的说明和所占空间长度的说明，数据存放的本质，还是以二进制的形式按照不同的空间分块方式（1个字节、2个字节…）来存放。

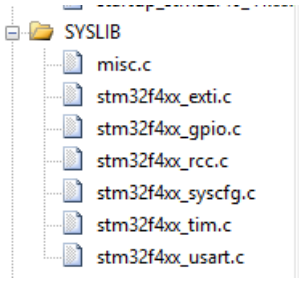
PS：似乎我说的比较毁三观，也有点懵，不过不要紧，多写写程序，就会明白：什么char、int，在我STM32的世界里，统统是int8_t、uint8_t、int16_t、uint16_t、int32_t、uint32_t。

然后我在留一个小问题：负数是怎么存放的？

三、开始实验

1.添加需要的ST固件库

这个可能我在模板里已经添加了，没关系，一起来熟悉一下：

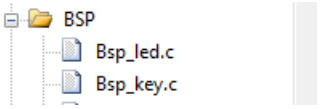


要添加的是：stm32f4xx_gpio.c和stm32f4xx_rcc.c，双击”SYSLIB”在之前说的SYSLIB文件夹下找到这两个文件，添加；

其他的嘛，是单片机正常工作所必须的，慢慢就会知道了，这里只有stm32f4xx_gpio.c和stm32f4xx_rcc.c和本次实验密切相关；

2.添加需要的BSP文件

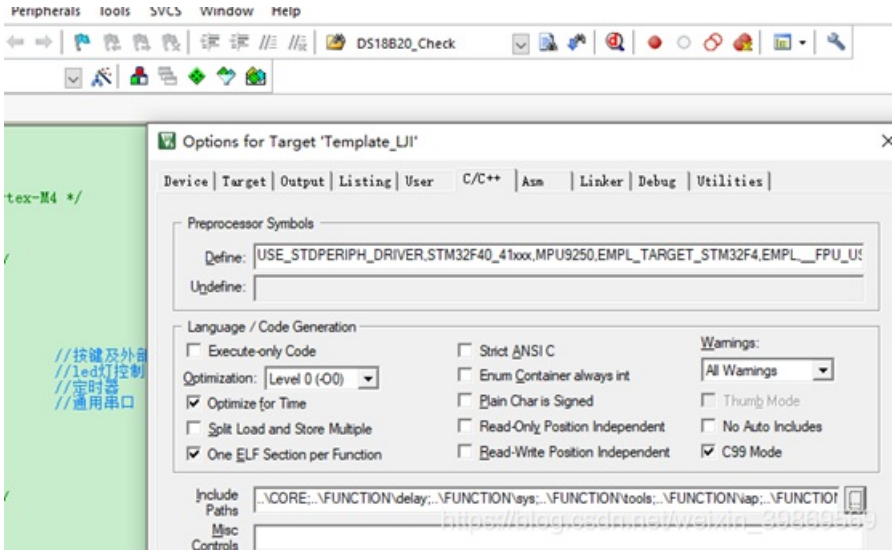
同样，双击”BSP”在之前说的BSP文件夹下找到这两个文件，添加；



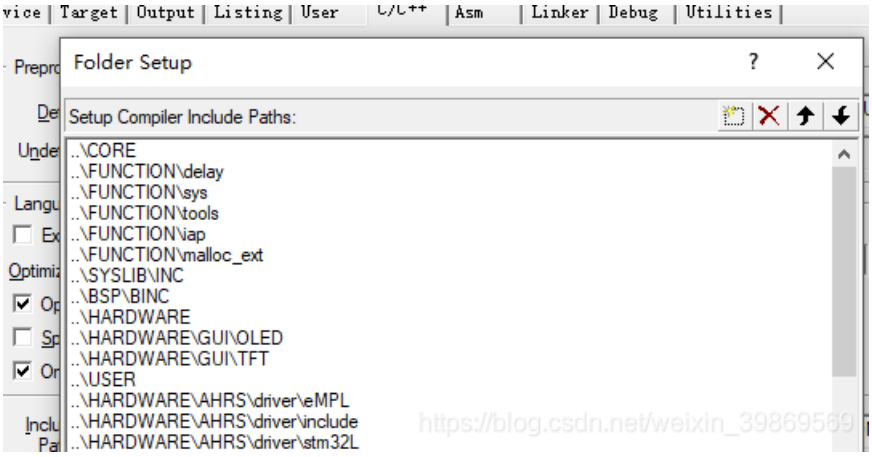
3.在main.h中注册

```
40
41  /* 片上外设接口 */
42  #include "Bsp_key.h"          //按键及外部中断
43  #include "Bsp_led.h"          //led灯控制
```

4.设置h文件的包含路径



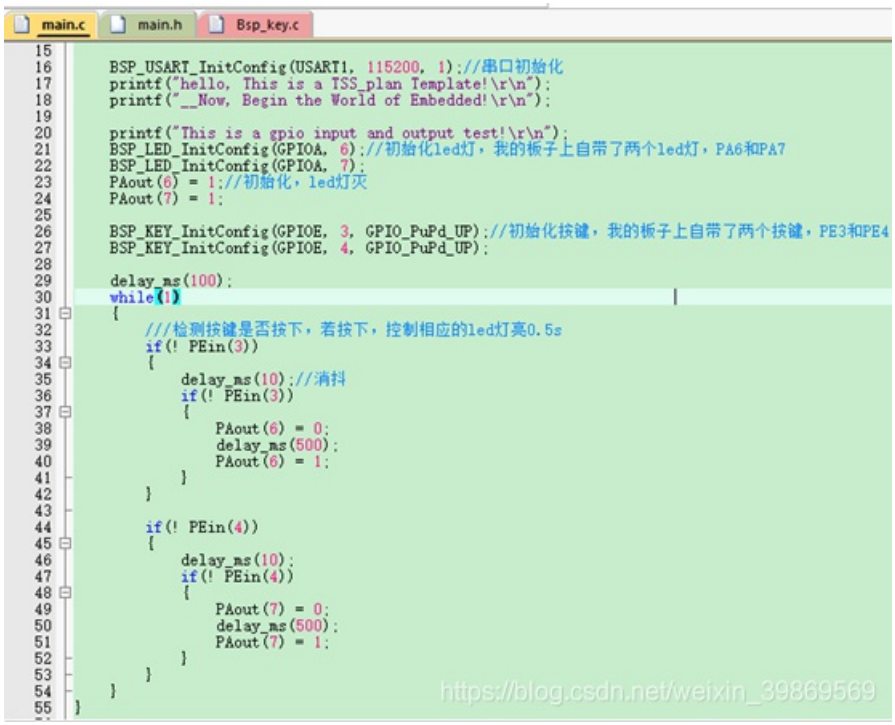
然后点击下图中红框中的按钮：



添加如下内容：

当然了，我已经添加好了，不过大家可以去熟悉一下流程；

5.主函数中书写测试代码



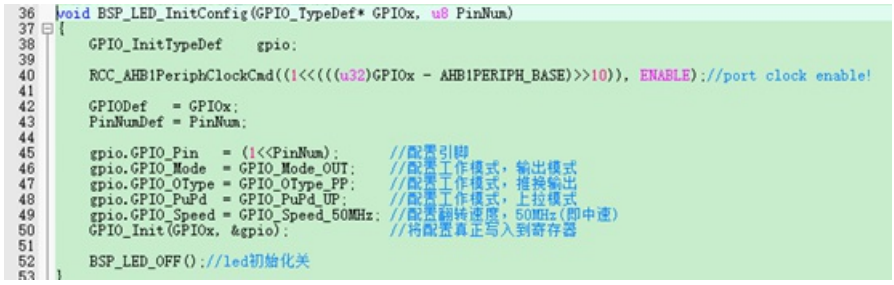
```c
BSP_USART_InitConfig(USART1, 115200, 1);//串口初始化
printf("hello, This is a TSS_plan Template!\r\n");
printf("__Now, Begin the World of Embedded!\r\n");

printf("This is a gpio input and output test!\r\n");
BSP_LED_InitConfig(GPIOA, 6);//初始化led灯，我的板子上自带了两个led灯，PA6和PA7
BSP_LED_InitConfig(GPIOA, 7);
PAout(6) = 1;//初始化，led灯灭
PAout(7) = 1;

BSP_KEY_InitConfig(GPIOE, 3, GPIO_PuPd_UP);//初始化按键，我的板子上自带了两个按键，PE3和PE4
BSP_KEY_InitConfig(GPIOE, 4, GPIO_PuPd_UP);

delay_ms(100);
while(1)
{
    ///检测按键是否按下，若按下，控制相应的led灯亮0.5s
    if(! PEin(3))
    {
        delay_ms(10);//消抖
        if(! PEin(3))
        {
            PAout(6) = 0;
            delay_ms(500);
            PAout(6) = 1;
        }
    }

    if(! PEin(4))
    {
        delay_ms(10);
        if(! PEin(4))
        {
            PAout(7) = 0;
            delay_ms(500);
            PAout(7) = 1;
        }
    }
}
```

程序其实很简单，使用BSP提供的接口初始化led灯和按键（其实就是配置gpio的输入和输出方式，详细配置我们一会挨个看）。

PS：特别说明的是，程序中用到了一个比较经典的案例，就是按键消抖，同样，什么是按键消抖，请百度。

四、分析BSP配置代码，学习GPIO

```
/**
  * @brief  :led指示灯初始化
  * @note   :--
  * @param  :*GPIOx, 管脚号
             PinNum, 引脚号(0~15)
  * @return :void
  *
  * @data   :2016/10/25
  * @design :
  **/
```



```
void BSP_LED_InitConfig(GPIO_TypeDef* GPIOx, u8 PinNum)
{
    GPIO_InitTypeDef    gpio;

    RCC_AHB1PeriphClockCmd((1<<((((u32)GPIOx - AHB1PERIPH_BASE)>>10)), ENABLE);//port clock enable!

    GPIODef   = GPIOx;
    PinNumDef = PinNum;

    gpio.GPIO_Pin   = (1<<PinNum);      //配置引脚
    gpio.GPIO_Mode  = GPIO_Mode_OUT;    //配置工作模式，输出模式
    gpio.GPIO_OType = GPIO_OType_PP;    //配置工作模式，推挽输出
    gpio.GPIO_PuPd  = GPIO_PuPd_UP;     //配置工作模式，上拉模式
    gpio.GPIO_Speed = GPIO_Speed_50MHz; //配置翻转速度，50MHz(即中速)
    GPIO_Init(GPIOx, &gpio);            //将配置真正写入到寄存器

    BSP_LED_OFF();//led初始化关
}
```

大家打开”Bsp_led.c”文件，找到第24行，这个就是gpio方式的配置过程：
第40行：

```
RCC_AHB1PeriphClockCmd((1<<((((u32)GPIOx - AHB1PERIPH_BASE)>>10)), ENABLE);
```

这个是使能时钟，就是给所使用的GPIO开启能量，这个是我写的一个通用方式，其一般形式是：
AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);不要纠结，以后我会详细讲解。

第45至50行：

```
gpio.GPIO_Pin = (1<<PinNum);  //配置引脚
gpio.GPIO_Mode = GPIO_Mode_OUT; //配置工作模式，输出模式
gpio.GPIO_OType = GPIO_OType_PP; //配置工作模式，推挽输出
gpio.GPIO_PuPd = GPIO_PuPd_UP;  //配置工作模式，上拉模式
gpio.GPIO_Speed = GPIO_Speed_50MHz; //配置翻转速度，50MHz(即中速)
GPIO_Init(GPIOx, &gpio);   //将配置真正写入到寄存器
```

这个是STM32中的初始化常用方式，我说过，ST的外设太多，导致寄存器很多，一般人根本记不住，所以ST公司就提供了固件库，用户通过ST提供的接口函数的参数进行设置，然后调用初始化函数，进行初始化。

至于为什么使用结构体，想想就知道：要配置的内容太多，而且类型又不一样，有8位数据，有16位数据，所以数组是不能用的，结构体恰恰是最佳选择。

而且GPIO_Init()这个函数的实质，就是把结构体gpio包含的内容，组合后赋给指定的寄存器：

```
212   /* ------------------------ Configure the port pins ---------------- */
213   /*-- GPIO Mode Configuration --*/
214   for (pinpos = 0x00; pinpos < 0x10; pinpos++)
215   {
216     pos = ((uint32_t)0x01) << pinpos;
217     /* Get the port pins position */
218     currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
219
220     if (currentpin == pos)
221     {
222       GPIOx->MODER  &= ~(GPIO_MODER_MODER0 << (pinpos * 2));
223       GPIOx->MODER |= (((uint32_t)GPIO_InitStruct->GPIO_Mode) << (pinpos * 2));
224
225       if ((GPIO_InitStruct->GPIO_Mode == GPIO_Mode_OUT) || (GPIO_InitStruct->GPIO_Mode
226       {
227         /* Check Speed mode parameters */
228         assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
229
230         /* Speed mode configuration */
231         GPIOx->OSPEEDR &= ~(GPIO_OSPEEDER_OSPEEDR0 << (pinpos * 2));
232         GPIOx->OSPEEDR |= ((uint32_t)(GPIO_InitStruct->GPIO_Speed) << (pinpos * 2));
233
234         /* Check Output mode parameters */
235         assert_param(IS_GPIO_OTYPE(GPIO_InitStruct->GPIO_OType));
236
237         /* Output mode configuration*/
238         GPIOx->OTYPER  &= ~((GPIO_OTYPER_OT_0) << ((uint16_t)pinpos)) ;
239         GPIOx->OTYPER |= (uint16_t)(((uint16_t)GPIO_InitStruct->GPIO_OType) << ((uint16
240       }
241
242       /* Pull-up Pull down resistor configuration*/
243       GPIOx->PUPDR &= ~(GPIO_PUPDR_PUPDR0 << ((uint16_t)pinpos * 2));
244       GPIOx->PUPDR |= (((uint32_t)GPIO_InitStruct->GPIO_PuPd) << (pinpos * 2));
245     }
246   }
247 }
```


这个是对GPIO_Init()函数执行Go to the Definition操作（还记得怎么操作么？忘记的话回去看第一讲），可以发现，好多”&=”和”|=”，我这里先不介绍怎么用，只是说它们一个叫置位、一个叫复位，相关细节，我会拿出一讲详细介绍。

PS：仔细看”Bsp_led.c”文件的第45行，这个移位指令能告诉我当PinNum为2是1<< PinNum是十六进制的多少么？然后使用Go to the Definition操作看一下GPIO_Pin_2是多少？

五、问题思考

1.以同样的方式看一下BSP_KEY_InitConfig()的配置过程

2.打开”Bsp_led.c”文件，找到第24行

```
RCC_AHB1PeriphClockCmd((1<<(((u32)GPIOx - AHB1PERIPH_BASE)>>10)), ENABLE);
```

当GPIOx为GPIOA时，((1<<(((u32)GPIOx - AHB1PERIPH_BASE)>>10))的十六进制结果是多少？
同样使用Go to the Definition操作看一下RCC_AHB1Periph_GPIOA的十六进制结果是多少？发现了什么？