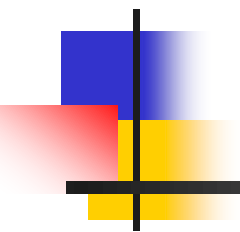


# 实验三：GPIO输出

## --固件库点亮LED灯





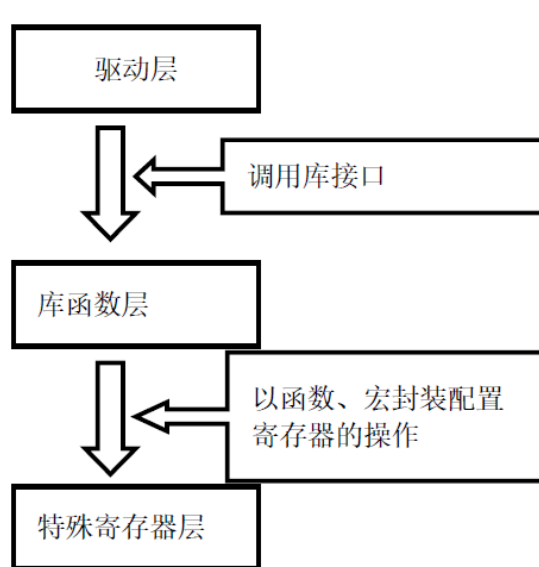
# 实验原理

---

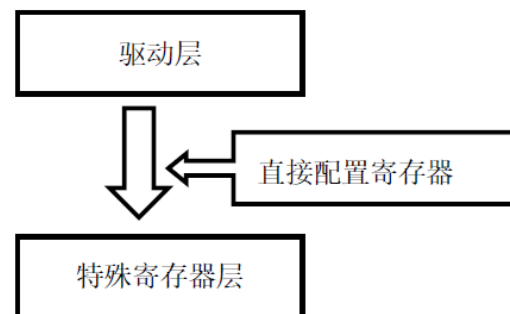
- 在上一个实验**LED**点亮
  - 用寄存器开发
  - 对每个控制的寄存器位手工写入特定参数
    - 照着硬件手册寄存器的说明
    - 配置时非常容易出错
    - 代码还不好理解，不便于维护
- 最好的方法：使用软件库
  - **STM32 标准函数库**
  - **ST 公司对STM32 提供的函数接口，即API**

# 实验原理

- 软件库是寄存器与用户驱动层之间的代码
  - 向下处理与寄存器直接相关的配置
  - 向上为用户提供配置寄存器的接口



库开发方式



直接配置寄存器方式

# 实验原理

寄存器名称	寄存器地址
GPIOH_MODER	0x4002 1C00
GPIOH_OTYPER	0x4002 1C04
GPIOH_OSPEEDR	0x4002 1C08
GPIOH_PUPDR	0x4002 1C0C
GPIOH_IDR	0x4002 1C10
GPIOH_ODR	0x4002 1C14
GPIOH_BSRR	0x4002 1C18
GPIOH_LCKR	0x4002 1C1C
GPIOH_AFR1	0x4002 1C20
GPIOH_AFRH	0x4002 1C24

## ■ 构建库函数

- 在寄存器点亮 LED 的代码
- 把代码一层层封装
- 实现库的最初的雏形

## ■ 寄存器地址封装

- 外设寄存器的地址是基地址的偏移地址逐个连续递增的
- 这种方式跟结构体里面的成员类似

# 实验原理

- 结构体成员的顺序按照寄存器的偏移地址从低到高排列
- 成员类型跟寄存器类型一样

```
1 //volatile 表示易变的变量，防止编译器优化
2 #define      __IO      volatile
3 typedef unsigned int uint32_t;
4 typedef unsigned short uint16_t;
5
6 /* GPIO 寄存器列表 */
7 typedef struct {
8     __IO      uint32_t MODER;
9     __IO      uint32_t OTYPER;
10    __IO      uint32_t OSPEEDR;
11    __IO      uint32_t PUPDR;
12    __IO      uint32_t IDR;
13    __IO      uint32_t ODR;
14    __IO      uint16_t BSRRL;
15    __IO      uint16_t BSRRH;
16    __IO      uint32_t LCKR;
17    __IO      uint32_t AFR[2];
18 } GPIO_TypeDef;
```

# 实验原理

- 定义访问外设的结构体指针
  - 要给结构体的首地址赋值

```
1 /*定义 GPIOA-H 寄存器结构体指针*/
2 #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
3 #define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
4 #define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
5 #define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
6 #define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
7 #define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
8 #define GPIOG ((GPIO_TypeDef *) GPIOG_BASE)
9 #define GPIOH ((GPIO_TypeDef *) GPIOH_BASE)
10
11 /*定义 RCC 外设 寄存器结构体指针*/
12 #define RCC ((RCC_TypeDef *) RCC_BASE)
```

## stm32f4xx.h文件

```
5 /*GPIO 外设基地址*/
6 #define GPIOH_BASE (AHB1PERIPH_BASE + 0x1C00)
```

# 实验原理

- 定义访问外设的结构体指针（续）
  - 通过强制把外设的基地址转换成 `GPIO_TypeDef` 类型的地址
  - 通过结构体指针操作，可访问外设的寄存器

```
/*GPIOH MODER10 清空*/  
GPIOH->MODER  &= ~( 0x03<< (2*10));  
/*PH10 MODER10 = 01b 输出模式*/  
GPIOH->MODER |= (1<<2*10);
```

结构体指针操作

地址操作

```
/*GPIOH MODER10 清空*/  
GPIOH_MODER  &= ~( 0x03<< (2*10));  
/*PH10 MODER10 = 01b 输出模式*/  
GPIOH_MODER |= (1<<2*10);
```

# 实验原理

- 使用函数来封装GPIO 的基本操作
  - |=与=是等效的

```
22 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
23 {
24     /*设置 GPIOx 端口 BSRRH 寄存器的第 GPIO_Pin 位,使其输出低电平*/
25     /*因为 BSRR 寄存器写 0 不影响,
26     宏 GPIO_Pin 只是对应位为 1, 其它位均为 0, 所以可以直接赋值*/
27
28     GPIOx->BSRRH = GPIO_Pin;
29 }
```

封装后

```
/*PH10 BSRR 寄存器的 BR10 置 1, 使引脚输出低电平*/
GPIOH_BSRR |= (1<<16<<10);
```

封装前



# 实验原理

- 使用函数来封装GPIO 的基本操作（续）
  - 操作函数使用范例

```
22 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
23 {
24     /*设置 GPIOx 端口 BSRRH 寄存器的第 GPIO_Pin 位,使其输出低电平*/
25     /*因为 BSRR 寄存器写 0 不影响,
26     宏 GPIO_Pin 只是对应位为 1, 其它位均为 0, 所以可以直接赋值*/
27
28     GPIOx->BSRRH = GPIO_Pin;
29 }
```

## 使用范例

```
4 /*控制 GPIOH 的引脚 10 输出低电平*/
5 GPIO_ResetBits(GPIOH, (uint16_t) (1<<10));
```

# 实验原理

## ■ 使用函数来封装GPIO 的基本操作（续）

### ■ 封装引脚

```
2 #define GPIO_Pin_0      (uint16_t)0x0001)
3 #define GPIO_Pin_1      ((uint16_t)0x0002)
4 #define GPIO_Pin_2      ((uint16_t)0x0004)
5 #define GPIO_Pin_3      ((uint16_t)0x0008)
6 #define GPIO_Pin_4      ((uint16_t)0x0010)
7 #define GPIO_Pin_5      ((uint16_t)0x0020)
8 #define GPIO_Pin_6      ((uint16_t)0x0040)
9 #define GPIO_Pin_7      ((uint16_t)0x0080)
10 #define GPIO_Pin_8      ((uint16_t)0x0100)
11 #define GPIO_Pin_9      ((uint16_t)0x0200)
12 #define GPIO_Pin_10     ((uint16_t)0x0400)
13 #define GPIO_Pin_11     ((uint16_t)0x0800)
14 #define GPIO_Pin_12     ((uint16_t)0x1000)
15 #define GPIO_Pin_13     ((uint16_t)0x2000)
16 #define GPIO_Pin_14     ((uint16_t)0x4000)
17 #define GPIO_Pin_15     ((uint16_t)0x8000)
18 #define GPIO_Pin_All     ((uint16_t)0xFFFF)
```

```
4 /*控制 GPIOH 的引脚 10 输出低电平*/
```

```
5 GPIO_ResetBits(GPIOH,GPIO_Pin_10);
```

# 实验原理

- 定义初始化结构体GPIO\_InitTypeDef
  - 初始化GPIO 引脚各模式以结构体形式封装

```
1 typedef uint8_t unsigned char;
2 /**
3  * GPIO 初始化结构体类型定义
4  */
5 typedef struct {
6     uint32_t GPIO_Pin;          /*!< 选择要配置的 GPIO 引脚
7                                  可输入 GPIO_Pin_ 定义的宏 */
8
9     uint8_t GPIO_Mode;          /*!< 选择 GPIO 引脚的工作模式
10                                   可输入二进制值： 00 、 01、 10、 11
11                                   表示输入/输出/复用/模拟 */
12
13     uint8_t GPIO_Speed;         /*!< 选择 GPIO 引脚的速率
14                                   可输入二进制值： 00 、 01、 10、 11
15                                   表示 2/25/50/100MHz */
16
17     uint8_t GPIO_OType;         /*!< 选择 GPIO 引脚输出类型
18                                   可输入二进制值： 0 、 1
19                                   表示推挽/开漏 */
20
21     uint8_t GPIO_PuPd;          /*!<选择 GPIO 引脚的上/下拉模式
22                                   可输入二进制值： 00 、 01、 10
23                                   表示浮空/上拉/下拉*/
24 } GPIO_InitTypeDef;
```

# 实验原理

## ■ 定义引脚模式的枚举类型

```
1  /**
2   * GPIO 端口配置模式的枚举定义
3   */
4  typedef enum {
5      GPIO_Mode_IN    = 0x00, /*!< 输入模式 */
6      GPIO_Mode_OUT   = 0x01, /*!< 输出模式 */
7      GPIO_Mode_AF     = 0x02, /*!< 复用模式 */
8      GPIO_Mode_AN     = 0x03 /*!< 模拟模式 */
9  } GPIO_Mode_TypeDef;
11 /**
12  * GPIO 输出类型枚举定义
13  */
14 typedef enum {
15     GPIO_OType_PP = 0x00, /*!< 推挽模式 */
16     GPIO_OType_OD = 0x01 /*!< 开漏模式 */
17 } GPIO_OType_TypeDef;
```

# 实验原理

## ■ 定义引脚模式的枚举类型（续）

```
19 /**
20  * GPIO 输出速率枚举定义
21  */
22 typedef enum {
23     GPIO_Speed_2MHz    = 0x00, /*!< 2MHz    */
24     GPIO_Speed_25MHz   = 0x01, /*!< 25MHz   */
25     GPIO_Speed_50MHz   = 0x02, /*!< 50MHz   */
26     GPIO_Speed_100MHz  = 0x03  /*!<100MHz  */
27 } GPIO_Speed_TypeDef;
28
29 /**
30  *GPIO 上/下拉配置枚举定义
31  */
32 typedef enum {
33     GPIO_PuPd_NOPULL   = 0x00, /*浮空*/
34     GPIO_PuPd_UP       = 0x01, /*上拉*/
35     GPIO_PuPd_DOWN     = 0x02  /*下拉*/
36 } GPIOPuPd_TypeDef;
```

# 实验原理

- 使用枚举类型定义的GPIO\_InitTypeDef结构体成员

```
4 typedef struct {  
5     uint32_t GPIO_Pin;  
6     GPIOMode_TypeDef GPIO_Mode;  
7     GPIOSpeed_TypeDef GPIO_Speed;  
8     GPIOOType_TypeDef GPIO_OType;  
9     GPIOPuPd_TypeDef GPIO_PuPd;  
10 } GPIO_InitTypeDef;
```

# 实验原理

## ■ 给GPIO\_InitTypeDef 初始化结构体赋值 范例

```
1 GPIO_InitTypeDef InitStruct;
2
3 /* LED 端口初始化 */
4 /*选择要控制的 GPIO 引脚*/
5 InitStruct.GPIO_Pin = GPIO_Pin_10;
6 /*设置引脚模式为输出模式*/
7 InitStruct.GPIO_Mode = GPIO_Mode_OUT;
8 /*设置引脚的输出类型为推挽输出*/
9 InitStruct.GPIO_OType = GPIO_OType_PP;
10 /*设置引脚为上拉模式*/
11 InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
12 /*设置引脚速率为 2MHz */
13 InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
```

# 实验原理-GPIO初始化

```
2  /**
3   *函数功能：初始化引脚模式
4   *参数说明：GPIOx，该参数为 GPIO_TypeDef 类型的指针，指向 GPIO 端口的地址
5   *          GPIO_InitTypeDef:GPIO_InitTypeDef 结构体指针，指向初始化变量
6   */
7 void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
8 {
9     uint32_t pinpos = 0x00, pos = 0x00 , currentpin = 0x00;
10
11     /*-- GPIO Mode Configuration --*/
12     for (pinpos = 0x00; pinpos < 16; pinpos++) {
13         /*以下运算是为了通过 GPIO_InitStruct->GPIO_Pin 算出引脚号 0-15*/
14
15         /*经过运算后 pos 的 pinpos 位为 1，其余为 0，与 GPIO_Pin_x 宏对应。
16         pinpos 变量每次循环加 1，*/
17         pos = ((uint32_t)0x01) << pinpos;
18
19         /* pos 与 GPIO_InitStruct->GPIO_Pin 做 & 运算，
20         若运算结果 currentpin == pos，
21         则表示 GPIO_InitStruct->GPIO_Pin 的 pinpos 位也为 1，
22         从而可知 pinpos 就是 GPIO_InitStruct->GPIO_Pin 对应的引脚号：0-15*/
23         currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
24     }
```

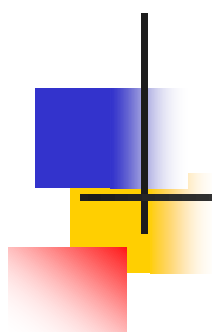


# 实验原理-GPIO初始化（续）

```
25      /*currentpin == pos 时执行初始化*/
26      if (currentpin == pos) {
27          /*GPIOx 端口, MODER 寄存器的 GPIO_InitStruct->GPIO_Pin 对应的引脚,
28          MODER 位清空*/
29          GPIOx->MODER  &= ~(3 << (2 *pinpos));
30
31          /*GPIOx 端口, MODER 寄存器的 GPIO_Pin 引脚,
32          MODER 位设置"输入/输出/复用输出/模拟"模式*/
33          GPIOx->MODER |= (((uint32_t)GPIO_InitStruct->GPIO_Mode) << (2 *pinpos));
34
35          /*GPIOx 端口, PUPDR 寄存器的 GPIO_Pin 引脚,
36          PUPDR 位清空*/
37          GPIOx->PUPDR &= ~(3 << ((2 *pinpos)));
38
39          /*GPIOx 端口, PUPDR 寄存器的 GPIO_Pin 引脚,
40          PUPDR 位设置"上/下拉"模式*/
41          GPIOx->PUPDR |= (((uint32_t)GPIO_InitStruct->GPIO_PuPd) << (2 *pinpos));
```

# 实验原理-GPIO初始化（续）

```
43      /*若模式为"输出/复用输出"模式，则设置速度与输出类型*/
44      if ((GPIO_InitStruct->GPIO_Mode == GPIO_Mode_OUT) ||
45          (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_AF)) {
46          /*GPIOx 端口，OSPEEDR 寄存器的 GPIO_Pin 引脚，
47          OSPEEDR 位清空*/
48          GPIOx->OSPEEDR &= ~(3 << (2 *pinpos));
49          /*GPIOx 端口，OSPEEDR 寄存器的 GPIO_Pin 引脚，
50          OSPEEDR 位设置输出速度*/
51          GPIOx->OSPEEDR |= ((uint32_t) (GPIO_InitStruct->GPIO_Speed)<<(2 *pinpos));
52          /*GPIOx 端口，OTYPER 寄存器的 GPIO_Pin 引脚，
53          OTYPER 位清空*/
54          GPIOx->OTYPER  &= ~(1 << (pinpos)) ;
55          /*GPIOx 端口，OTYPER 位寄存器的 GPIO_Pin 引脚，
56          OTYPER 位设置"推挽/开漏"输出类型*/
57          GPIOx->OTYPER |= (uint16_t)(( GPIO_InitStruct->GPIO_OType)<< (pinpos));
58      }
59  }
60 }
61 }
62 }
```



```
5 #include "stm32f4xx_gpio.h"
12 int main(void)
13 {
14     GPIO_InitTypeDef InitStruct;
15
16     /*开启 GPIOH 时钟，使用外设时都要先开启它的时钟*/
17     RCC->AHB1ENR |= (1<<7);
18
19     /* LED 端口初始化 */
20
21     /*初始化 PH10 引脚*/
22     /*选择要控制的 GPIO 引脚*/
23     InitStruct.GPIO_Pin = GPIO_Pin_10;
24     /*设置引脚模式为输出模式*/
25     InitStruct.GPIO_Mode = GPIO_Mode_OUT;
26     /*设置引脚的输出类型为推挽输出*/
27     InitStruct.GPIO_OType = GPIO_OType_PP;
28     /*设置引脚为上拉模式*/
29     InitStruct.GPIO_PuPd = GPIO_PuPd_UP;
30     /*设置引脚速率为 2MHz */
31     InitStruct.GPIO_Speed = GPIO_Speed_2MHz;
32     /*调用库函数，使用上面配置的 GPIO_InitStructure 初始化 GPIO*/
33     GPIO_Init(GPIOH, &InitStruct);
```

# 实验原理——LED灯点亮

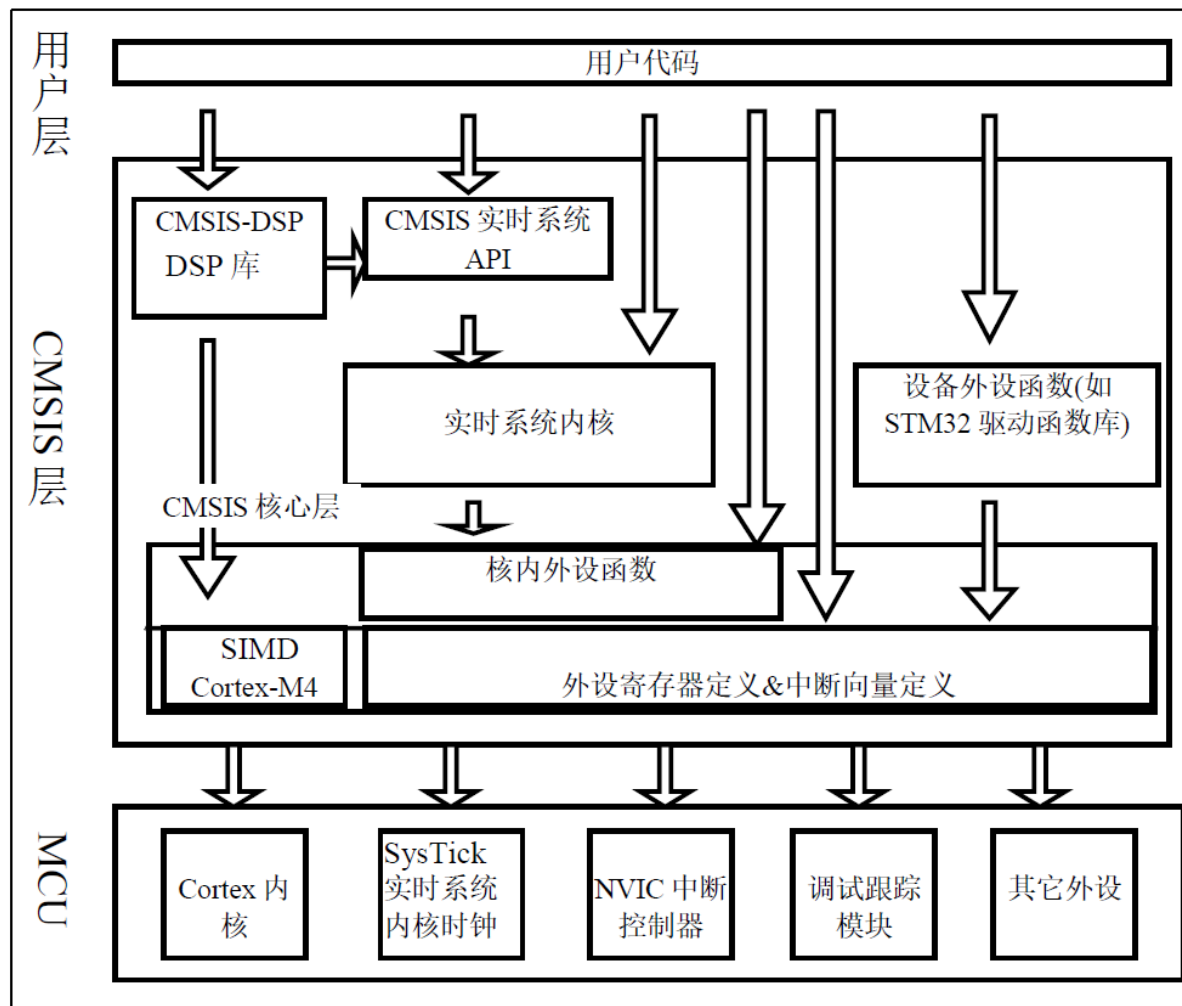
```
34
35     /*使引脚输出低电平,点亮 LED1*/
36     GPIO_ResetBits(GPIOH,GPIO_Pin_10);
37
38     /*延时一段时间*/
39     Delay(0xFFFFF);
40
41     /*使引脚输出高电平,关闭 LED1*/
42     GPIO_SetBits(GPIOH,GPIO_Pin_10);
43
44     /*初始化 PH11 引脚*/
45     InitStruct.GPIO_Pin = GPIO_Pin_11;
46     GPIO_Init(GPIOH,&InitStruct);
47
48     /*使引脚输出低电平,点亮 LED2*/
49     GPIO_ResetBits(GPIOH,GPIO_Pin_11);
50
51     while (1);
52
53 }
```



# CMSIS标准及库层次关系

- 基于**Cortex** 系列芯片采用的内核都是相同的，区别主要为核外的片上外设的差异
- 这些差异却导致软件在同内核，不同外设的芯片上移植困难
- ARM与芯片厂商建立了**CMSIS 标准(Cortex MicroController Software Interface Standard)**
- CMSIS 标准，新建了一个软件抽象层

# STM32固件库



# STM32固件库

STM32F4xx固件库文件分析		
外设相关	stm32f4xx.h	外设寄存器定义
	system_stm32f4xx.h	1、用于系统初始化
	system_stm32f4xx.c	2、用于配置系统时钟
	stm32f4xx_xx.h	外设固件库头文件
	stm32f4xx_xx.c	外设固件库
	misc.h	跟中断相关的固件库
	misc.c	
内核相关	core_cm4.h	内核寄存器定义
	core_cmFunc.h	操作内核相关的固件库，用的非常少
	core_cmInstr.h	
	core_cmSimd.h	
用户相关	stm32f4xx_it.h	用户编写的中断服务函数都放在这里
	stm32f4xx_it.c	
	main.c	main函数存在的地方



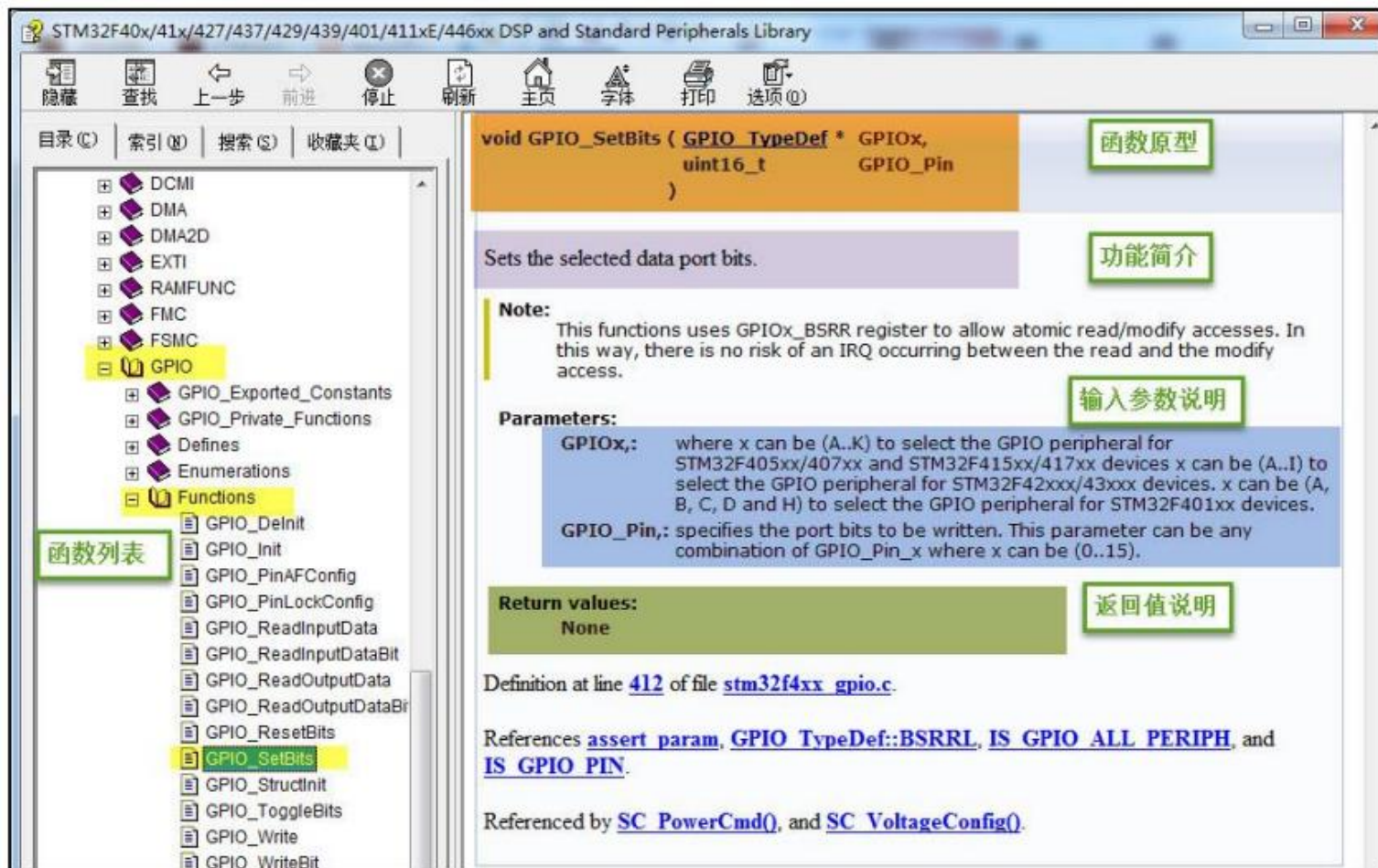
# STM32固件库

名称	作用
STARTUP	存放汇编的启动文件：startup_stm32f429_439xx.s
STM32F4xx StdPeriph Driver	与 STM32 外设相关的库文件 misc.c stm32f4xx_XXX.c（XXX 代表外设名称）
USER	用户编写的文件： main.c：main 函数文件，暂时为空 stm32f4xx_it.c：跟中断有关的函数都放这个文件，暂时为空
DOC	工程说明.txt：程序说明文件，用于说明程序的功能和注意事项等



# STM32固件库-库帮助文档的使用

stm32f4xx\_dsp\_stdperiph\_lib\_um.chm





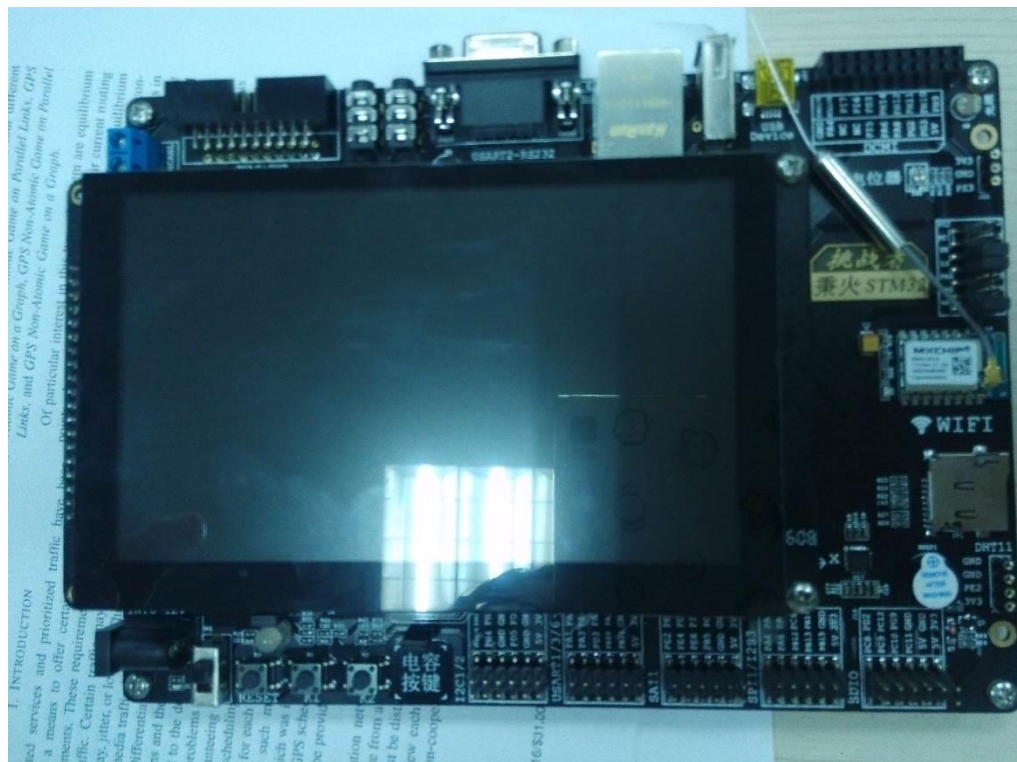
# 实验目的

---

- 学会使用固件库点亮**LED**灯

# 实验设备

- 软件Keil5(keil提供了软件仿真功能);
- STM32开发板





# 实验内容

---

- 学会使用固件库点亮**LED**灯

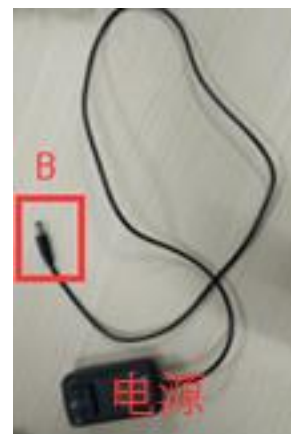
# 一、连接开发板

把仿真器用**USB**线连接电脑，如果仿真器的灯亮表示正常，可以使用。然后把仿真器的另外一端连接到开发板，给开发板上电，然后就可以通过软件**KEIL**给开发板下载程序。



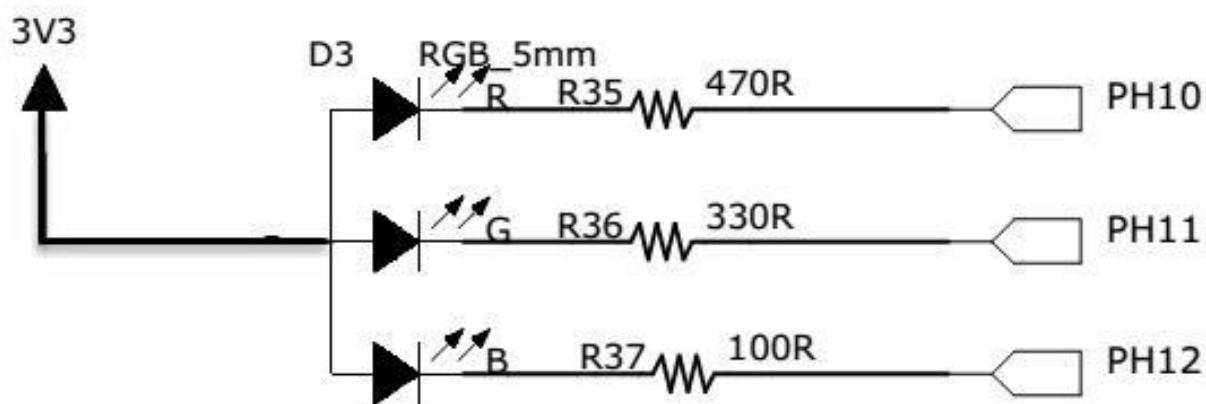
# 一、连接开发板

现在你们手里有三样东西，一个开发板，一个仿真器和一个电源，如图所示，将仿真器的A端插入开发板的A端，**注意正反面，正面的仿真器插头有长方形凸起**，仿真器的另一端用USB连接电脑，电源的B口插入开发板的B口，另一端插入电源。**C**处是开发板的电源开关，拨上去后**LED**灯亮则连接正常。



## 二、使用固件库点亮LED灯

### 1.硬件连接—STM32芯片与LED灯的连接



PH10:红色LED灯;

PH11:绿色LED灯;

PH12:蓝色LED灯;

图：LED灯电路连接图

【注】：通过控制引脚输出  
不同颜色LED灯！

## 二、使用固件库点亮LED灯

### 1.硬件连接—STM32芯片与LED灯的连接

图中从3个LED灯的阳极引出连接到3.3V电源，阴极各经过1个电阻引入至STM32的3个GPIO引脚PH10、PH11、PH12中，所以我们只要控制这三个引脚输出高低电平，即可控制其所连接LED灯的亮灭。

我们的目标是把GPIO的引脚设置成推挽输出模式(默认下拉)，输出低电平，这样就能让LED灯亮起来了。



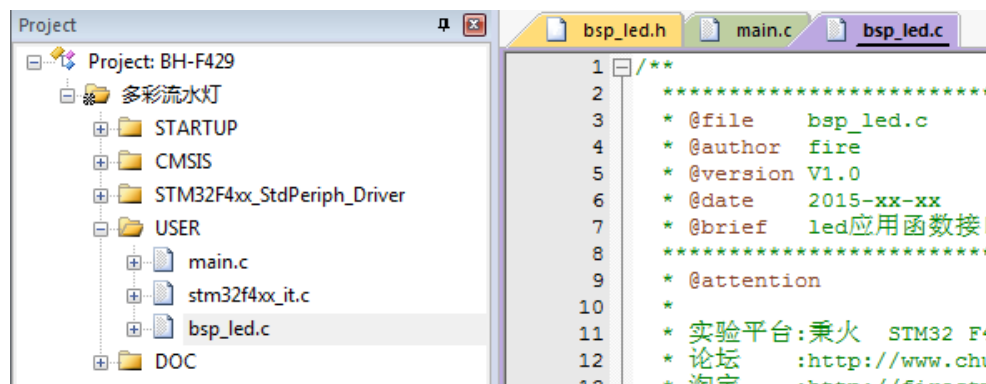
## 二、使用固件库点亮LED灯

### 2.示例工程分析

(0) 利用给的工程模板

(1) 接下来，我们以示例工程讲解如何通过固件库来点亮LED灯。

(2) 找到该示例工程后，在工程目录下找到后缀为“.uvprojx”的文件，用KEIL5打开即可。打开该工程，见下图，可看到一共有三个文件，分别**stm32f4xx\_it.c**、**bsp\_led.c** 以及**main.c**，下面我们对后两个文件进行简单的讲解。





## 二、使用固件库点亮LED灯

### 2.示例工程分析—bsp\_led.h

在分析bsp\_led.c和main.c文件之前，先来看一下bsp\_led.h头文件；

**【注】**：在编写应用程序的过程中，要考虑更改硬件环境的情况，例如LED灯的控制引脚与当前的不一样，我们希望程序只需要**做最小的修改**即可在新的环境正常运行。这个时候一般把硬件相关的部分使用宏来封装，若更改了硬件环境，只修改这些**硬件相关的宏**即可，这些定义一般存储在头文件，即本例子中的“bsp\_led.h”文件中，代码见下页ppt...

## 二、使用固件库点亮LED灯

### 2.示例工程分析—bsp\_led.h—LED灯引脚宏定义

```
6 //引脚定义
7 /*****
8 //R 红色灯
9 #define LED1_PIN          GPIO_Pin_10
10 #define LED1_GPIO_PORT    GPIOH
11 #define LED1_GPIO_CLK     RCC_AHB1Periph_GPIOH
12
13 //G 绿色灯
14 #define LED2_PIN          GPIO_Pin_11
15 #define LED2_GPIO_PORT    GPIOH
16 #define LED2_GPIO_CLK     RCC_AHB1Periph_GPIOH
17
18 //B 蓝色灯
19 #define LED3_PIN          GPIO_Pin_12
20 #define LED3_GPIO_PORT    GPIOH
21 #define LED3_GPIO_CLK     RCC_AHB1Periph_GPIOH
22
23 /*****/
```

以上代码分别把控制LED灯的GPIO端口、GPIO引脚号以及GPIO端口时钟封装起来了。在实际控制的时候我们就直接用这些宏，以达到应用代码硬件无关的效果。

【注】其中的GPIO时钟宏“RCC\_AHB1Periph\_GPIOH”是STM32标准库定义的GPIO端口时钟相关的宏，它的作用与“GPIO\_Pin\_x”这类宏类似，是用于指示寄存器位的，方便库函数使用。

## 二、使用固件库点亮LED灯

### 2.示例工程分析—bsp\_led.h—控制LED灯亮灭的宏

```
26  /* 直接操作寄存器的方法控制Io */
27  #define digitalHi(p,i)      {p->BSRRL=i;}           //设置为高电平
28  #define digitalLo(p,i)     {p->BSRRH=i;}           //输出低电平
29  #define digitalToggle(p,i) {p->ODR ^=i;}           //输出反转状态
30
31
32  /* 定义控制Io的宏 */
33  #define LED1_TOGGLE      digitalToggle(LED1_GPIO_PORT,LED1_PIN)
34  #define LED1_OFF         digitalHi(LED1_GPIO_PORT,LED1_PIN)
35  #define LED1_ON          digitalLo(LED1_GPIO_PORT,LED1_PIN)
36
37  #define LED2_TOGGLE      digitalToggle(LED2_GPIO_PORT,LED2_PIN)
38  #define LED2_OFF         digitalHi(LED2_GPIO_PORT,LED2_PIN)
39  #define LED2_ON          digitalLo(LED2_GPIO_PORT,LED2_PIN)
40
41  #define LED3_TOGGLE      digitalToggle(LED3_GPIO_PORT,LED3_PIN)
42  #define LED3_OFF         digitalHi(LED3_GPIO_PORT,LED3_PIN)
43  #define LED3_ON          digitalLo(LED3_GPIO_PORT,LED3_PIN)
44
45  //红
46  #define LED_RED          LED1_ON;LED2_OFF;LED3_OFF
47
48  //绿
49  #define LED_GREEN        LED1_OFF;LED2_ON;LED3_OFF
50
51  //蓝
52  #define LED_BLUE         LED1_OFF;LED2_OFF;LED3_ON
```

(1) 为了方便控制LED灯，我们把LED灯常用的亮、灭及状态反转的控制也直接定义成宏。

(2) 这部分宏控制LED亮灭的操作是直接向BSRR寄存器写入控制指令来实现的，对BSRRL写1输出高电平，对BSRRH写1输出低电平，对ODR寄存器某位进行异或操作可反转位的状态。

# 二、使用固件库点亮LED灯

## 2.示例工程分析—bsp\_led.c—LED GPIO初始化函数

```
18 #include "../led/bsp_led.h"
19 void LED_GPIO_Config(void)
20 {
21     /*定义一个GPIO_InitTypeDef类型的结构体*/
22     GPIO_InitTypeDef GPIO_InitStructure;
23
24     /*开启LED相关的GPIO外设时钟*/
25     RCC_AHB1PeriphClockCmd ( LED1_GPIO_CLK|LED2_GPIO_CLK|LED3_GPIO_CLK, ENABLE);
26
27     /*选择要控制的GPIO引脚*/
28     GPIO_InitStructure.GPIO_Pin = LED1_PIN;
29
30     /*设置引脚模式为输出模式*/
31     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
32
33     /*设置引脚的输出类型为推挽输出*/
34     GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
35
36     /*设置引脚为上拉模式*/
37     GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
38
39     /*设置引脚速率为2MHz */
40     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
41
42     /*调用库函数，使用上面配置的GPIO_InitStructure初始化GPIO*/
43     GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStructure);
44
45     /*选择要控制的GPIO引脚*/
46     GPIO_InitStructure.GPIO_Pin = LED2_PIN;
47     GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStructure);
48
49     /*选择要控制的GPIO引脚*/
50     GPIO_InitStructure.GPIO_Pin = LED3_PIN;
51     GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStructure);
52 }
```

(1) 初始化GPIO端口时钟时采用了STM32库函数；

(2) 函数执行流程

见下页ppt...



## 二、使用固件库点亮LED灯

### 2.示例工程分析—bsp\_led.c—LED GPIO初始化函数

- (1) 使用GPIO\_InitTypeDef定义GPIO初始化结构体变量，以便下面用于存储GPIO配置。
- (2) 调用库函数RCC\_AHB1PeriphClockCmd来使能LED灯的GPIO端口时钟。该函数有两个输入参数，第一个参数用于指示要配置的时钟；第二个参数用于设置状态，可输入“Disable”关闭或“Enable”使能时钟。
- (3) 向GPIO初始化结构体赋值，把引脚初始化成推挽输出模式。
- (4) 使用以上初始化结构体的配置，调用GPIO\_Init函数向寄存器写入参数，完成GPIO的初始化。
- (5) 使用同样的初始化结构体，只修改控制的引脚和端口，初始化其它LED灯使用的GPIO引脚。

## 二、使用固件库点亮LED灯

### 2.示例工程分析—main.c—控制LED灯

```
17 #include "stm32f4xx.h"
18 #include "../led/bsp_led.h"
19 void Delay(__IO u32 nCount);
20
21 /**
22  * @brief 主函数
23  * @param 无
24  * @retval 无
25  */
26 int main(void)
27 {
28     /* LED 端口初始化 */
29     LED_GPIO_Config();
30     /* 控制LED灯 */
31     while (1)
32     {
33
34         /*轮流显示 红绿蓝黄紫青白 颜色*/
35         LED_RED;
36         Delay(0xFFFFFF);
37
38         LED_GREEN;
39         Delay(0xFFFFFF);
40
41         LED_BLUE;
42         Delay(0xFFFFFF);
43     }
44 }
45
46 void Delay(__IO uint32_t nCount) //简单的延时函数
47 {
48     for(; nCount != 0; nCount--);
49 }
```

在main函数中，调用我们前面定义的LED\_GPIO\_Config初始化好LED的控制引脚，然后直接调用各种控制LED灯亮灭的宏来实现LED灯的控制。



## 二、使用固件库点亮LED灯

### 3. 下载程序到开发板一下载器设置

在开始将写好的程序烧写到开发板之前，我们需要对魔术棒中的Utilities和Debug两个选项进项配置，具体配置过程和之前一样，见“使用寄存器点亮LED灯” ppt...

两个选项设置好之后，就可将程序烧写到开发板上，并观察实验现象：“红-绿-蓝”滚动点亮~





## 三、第三次课堂作业

仿照以上讲的实验例程，在原来实验例程的代码基础上修改代码，以完成第三次课堂小作业：

- (1) 完成示例的实验。
- (2) 完善bsp\_led.h和bsp\_led.c文件的内容，使得与硬件相关的都写到库文件中，然后重写main.c文件，完成“红-绿-蓝”滚动点亮~
- (3) 修改代码使得实验结果的现象为：“红-绿-蓝-黄-紫-青-白”滚动点亮~

# 写头文件注意

## 代码清单 11-9 防止头文件重复包含的宏

```
1 #ifndef __LED_H
2 #define __LED_H
3
4 /*此处省略头文件的具体内容*/
5
6 #endif /* end of __LED_H */
```

在头文件的开头，使用“#ifndef”关键字，判断标号“\_\_LED\_H”是否被定义，若没有被定义，则从“#ifndef”至“#endif”关键字之间的内容都有效，也就是说，这个头文件若被其它文件“#include”，它就会被包含到其该文件中了，且头文件中紧接着使用“#define”关键字定义上面判断的标号“\_\_LED\_H”。当这个头文件被同一个文件第二次“#include”包含的时候，由于有了第一次包含中的“#define \_\_LED\_H”定义，这时再判断“#ifndef \_\_LED\_H”，判断的结果就是假了，从“#ifndef”至“#endif”之间的内容都无效，从而防止了同一个头文件被包含多次，编译时就不会出现“redefine（重复定义）”的错误了。