《PWM+TIM 实现全彩灯》实验报告

课程名称： 嵌入式系统设计	年级：23 级	上机实践成绩：
指导教师：郭建	姓名： 张建夫	
上机实践名称：	学号：10235101477	上机实践日期：
PWM+TIM 实现全彩灯		2025/05/20
上机实践编号：	组号：	上机实践时间：
		14：50~16：30

一、 目的与要求

- 了解 STM32 通用定时器相关知识
- 学会使用通用定时器的 PWM 输出

二、 内容与实验原理

1. 实验内容：

- 了解通用定时器的相关知识
- 理解通用定时器的 PWM 输出功能
- 学会使用 PWM 实现全彩灯

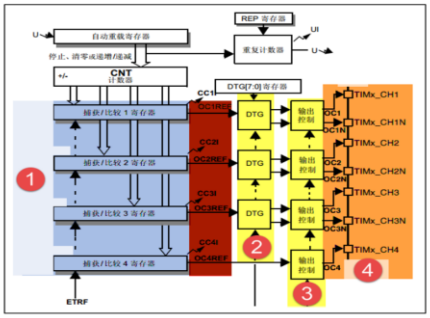
2. 实验原理：

（1）通用 TIM 介绍：

- STM32F42xxx 系列控制器有 10 个通用定时器(见下图)，本节使用 TIM5.控制器上所有定时器都是彼此独立的，不共享任何资源。

定时器类型	Timer	计数器分辨率	计数器类型	预分频系数	DMA 请求生成	捕获/比较通道	互补输出	最大接口时钟(MHz)	最大定时器时钟(MHz)
高级控制	TIM1 和 TIM8	16 位	递增、递减、递增/递减	1~65536(整数)	有	4	有	90 (APB2)	180
通用	TIM2、TIM5	32 位	递增、递减、递增/递减	1~65536(整数)	有	4	无	45 (APB1)	90/180
	TIM3、TIM4	16 位	递增、递减、递增/递减	1~65536(整数)	有	4	无	45 (APB1)	90/180
	TIM9	16 位	递增	1~65536(整数)	无	2	无	90 (APB2)	180
	TIM10、TIM11	16 位	递增	1~65536(整数)	无	1	无	90 (APB2)	180
	TIM12	16 位	递增	1~65536(整数)	无	2	无	45 (APB1)	90/180
	TIM13、TIM14	16 位	递增	1~65536(整数)	无	1	无	45 (APB1)	90/180
基本	TIM6 和 TIM7	16 位	递增	1~65536(整数)	有	0	无	45 (APB1)	90/180

通用定时器包含基本定时器的所有功能，下图给出的是一张通用定时器的部分功能结构图(本节所涉及到的)



图中的白色部分为基本定时器的功能，彩色部分为通用定时器独有功能。

里面的 tim 原理是通过上图计数器 CNT 的值跟比较寄存器 CCR(捕获比较寄存器）的值进行比较，相等的时候，输出参考信号 OCxREF 的信号的极性就会改变，其中 OCxREF=1（高电平）称之为有效电平， OCxREF=0（低电平）称之为无效电平，并且会产生比较中断 CCxI，相应的标志位 CCxIF（ SR 寄存器中）会置位。**然后 OCxREF 再经过一系列的控制之后就成为真正的输出信号 OCx/OCxN。**

而这些输出信号最终是通过定时器的外部 io 进行输出，分别为 CH1/2/3/4。

（2）PWM 介绍：

PWM 指的是通过调节脉冲信号的占空比来控制输出信号平均电压或功率的技术。

基本原理

- PWM信号是一个周期性的方波，其占空比（Duty Cycle）是指高电平时间与整个周期的比值
- 周期为 T，高电平时间为 T_on ，则占空比（D）为：D=T_on/T
- 输出信号的平均电压（V_avg）与占空比成正比: V_avg= V_max*D
- 占空比越高，输出信号的平均电压或功率越大
- 占空比越低，输出信号的平均电压或功率越小

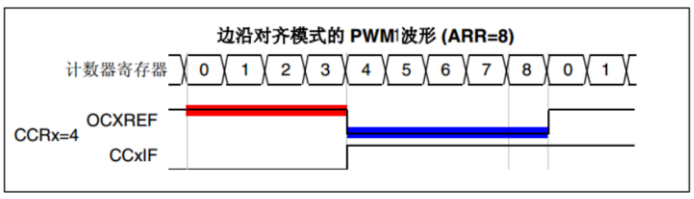
PWM输出功能介绍：

- 脉冲宽度调制(PWM)，是英文 “Pulse Width Modulation”的缩写，简称脉宽调制。STM32的定时器除了TIM6和TIM7之外，其他的定时器都可以用来产生PWM输出，通用定时器能同时产生4路的PWM输出（上图中的CH1、CH2、CH3、CH4）。
- PWM 输出就是对外输出脉宽（即占空比）可调的方波信号，信号频率由自动重装寄存器 ARR 的值决定，占空比由比较寄存器 CCR 的值决定。
- STM32的PWM输出有两种模式，模式1和模式2(CNT 代表计数值)

模式	计数器 CNT 计算方式	说明
PWM1	递增	CNT<CCR，通道 CH 为有效，否则为无效
	递减	CNT>CCR，通道 CH 为无效，否则为有效
PWM2	递增	CNT<CCR，通道 CH 为无效，否则为有效
	递减	CNT>CCR，通道 CH 为有效，否则为无效

本次实验选用的是 PWM1，其输出的工作原理如下：

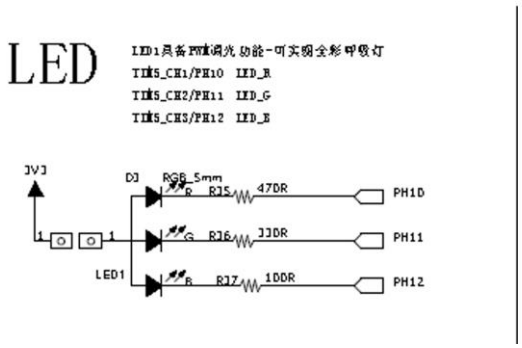
在边沿对齐模式下，计数器 CNT 只工作在一种模式，递增或者递减模式。这里我们以CNT 工作在递增模式为例，在中， ARR=8，CCR=4 CNT 从 0 开始计数，当 CNT<CCR 的值时， OCxREF 为有效的高电平，于此同时，比较中断寄存器 CCxIF 置位。当CCR=<CNT<=ARR 时 OCxREF 为无效的低电平。然后 CNT 又从 0 开始计数并生成计数器上溢事件，以此循环往复。



注：边沿对齐模式大家不必关注，它只是比基本定时器多了一种向下计数功能

前面已经说明上页提及的OCxREF经过一系列的控制最后输出到输出到本CCR（比较寄存器）的相应CH通道。

本次实验使用TIM5定时器的原因在于TIM5的3个输出通道已经和PH10、PH11、PH12相连。



（3）TIM 编程介绍：

TIM_TimeBaseInitTypeDef

输出比较结构体 TIM_TimeBaseInitTypeDef和基本定时器一致。

```
typedef struct {
    uint16_t TIM_Prescaler;          // 预分频器
    uint16_t TIM_CounterMode;        // 计数模式
    uint32_t TIM_Period;             // 定时器周期
    uint16_t TIM_ClockDivision;      // 时钟分频
    uint8_t TIM_RepetitionCounter;   // 重复计算器
} TIM_TimeBaseInitTypeDef;
```

- TIM_CounterMode：定时器计数方式，可设置为向上计数、向下计数以及中心对齐。通用定时器允许选择任意一种。
- TIM_ClockDivision：时钟分频，设置定时器时钟 CK_INT 频率与死区发生器以及数字滤波器采样时钟频率分频比。（本节未用）
- TIM_RepetitionCounter：重复计数器，只有 8 位，只存在于高级定时器。（本节未用）

TIM_OCInitTypeDef（编程重要）！！！！！

输出比较结构体TIM_OCInitTypeDef用于输出比较模式，与 TIM_OCxInit 里面库函数配合使用完成指定定时器输出通道初始化配置。

代码清单 32-2 定时器比较输出初始化结构体

```
typedef struct {
    uint16_t TIM_OCMode;        // 比较输出模式
    uint16_t TIM_OutputState;   // 比较输出使能
    uint16_t TIM_OutputNState;  // 比较互补输出使能
    uint32_t TIM_Pulse;         // 脉冲宽度
    uint16_t TIM_OCPolarity;    // 输出极性
    uint16_t TIM_OCNPolarity;   // 互补输出极性
    uint16_t TIM_OCIdleState;   // 空闲状态下比较输出状态
    uint16_t TIM_OCNIdleState;  // 空闲状态下比较互补输出状态
} TIM_OCInitTypeDef;
```

TIM_OCMode：比较输出模式选择，总共有八种，常用的为 PWM1/PWM2。
TIM_OutputState：比较输出使能，决定最终的输出比较信号 OCx 是否通过外部引脚输出。
TIM_Pulse：比较输出脉冲宽度，实际设定比较寄存器 CCR 的值，决定脉冲宽度。可设置范围为 0 至 65535。
TIM_OCPolarity：比较输出极性，可选 OCx 为高电平有效或低电平有效。它决定着定时器通道有效电平，意思就是说决定了前面例子中前半段(CNT<CCR)时的电平。

三、 使用环境

调用 dxdiag 工具：

Operating System: Windows 11 家庭中文版 64-bit (10.0, Build 22621) (22621.ni_release.220506-1250)
Language: Chinese (Simplified) (Regional Setting: Chinese (Simplified))
System Manufacturer: HP
System Model: HP Pavilion Aero Laptop 13-be2xxx
BIOS: F.13 (type: UEFI)
Processor: AMD Ryzen 5 7535U with Radeon Graphics (12 CPUs), ~2.9GHz
Memory: 16384MB RAM
Available OS Memory: 15574MB RAM
Page File: 27604MB used, 5685MB available
Windows Dir: C:\WINDOWS
DirectX Version: DirectX 12
DX Setup Parameters: Not found
User DPI Setting: 144 DPI (150 percent)
System DPI Setting: 192 DPI (200 percent)
DWM DPI Scaling: UnKnown
Miracast: Available, with HDCP
Microsoft Graphics Hybrid: Not Supported

四、主要实验内容和结果展示

本次实验使用的是之前提供的模板，需要将 bsp_key 的头文件和源文件均改成 bsp_color_led 的文件内容。另外，对于要求的拓展实验，还需要增加基本定时器的初始化相关头文件和源文件。

1. 示例实验：

按照 ppt 上的教程，先要在头文件（bsp_color_led.h）对串口的变量进行映射（宏定义）：

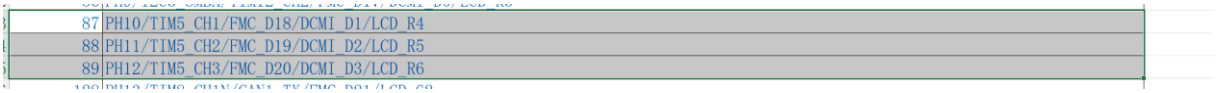
```
#define COLOR_LED1_PIN                GPIO_Pin_10
#define COLOR_LED1_GPIO_PORT          GPIOH
#define COLOR_LED1_GPIO_CLK           RCC_AHB1Periph_GPIOH
#define COLOR_LED1_PINSOURCE              GPIO_PinSource10
#define COLOR_LED1_AF             GPIO_AF_TIM5

#define COLOR_LED2_PIN                GPIO_Pin_11
#define COLOR_LED2_GPIO_PORT          GPIOH
#define COLOR_LED2_GPIO_CLK           RCC_AHB1Periph_GPIOH
#define COLOR_LED2_PINSOURCE              GPIO_PinSource11
#define COLOR_LED2_AF             GPIO_AF_TIM5

#define COLOR_LED3_PIN                GPIO_Pin_12
#define COLOR_LED3_GPIO_PORT          GPIOH
#define COLOR_LED3_GPIO_CLK           RCC_AHB1Periph_GPIOH
#define COLOR_LED3_PINSOURCE              GPIO_PinSource12
#define COLOR_LED3_AF             GPIO_AF_TIM5
```

该处是对各个灯引脚的定义，这里使用的三个的引脚分别是 10，11，12 号引脚，GPIO 端口编号为 H，这里使用这三个引脚的原因有两点：

- 这三个引脚本身就对应着红绿蓝三个颜色的灯
- Tim5 引脚的三个输出通道已经和 PH10、PH11、PH12 相连。在老师提供的“STM32F429IGT6 管脚汇总（含第二功能）”文件中，可以看到：



这三个端口含有通用计时器 tim5 分别的三个输出通道

仔细观察可以发现，这几个宏的定义仅仅只有下面的 pinsourse（用于配置引脚复用功能）和 af（指定对应引脚的复用功能为 tim5）与之前 gpio 的宏定义不一样，其他的端口和引脚实际上是一样的。

接下来定义定时器相关的宏：

```
// 定时器
#define COLOR_TIM                       TIM5
#define COLOR_TIM_CLK                   RCC_APB1Periph_TIM5
#define COLOR_LED1_CCRx                 CCR1
#define COLOR_LED2_CCRx                 CCR2
#define COLOR_LED3_CCRx                 CCR3

#define COLOR_LED1_TIM_CHANNEL                      TIM_Channel_1
#define COLOR_LED2_TIM_CHANNEL                      TIM_Channel_2
#define COLOR_LED3_TIM_CHANNEL                      TIM_Channel_3

#define COLOR_TIM_IRQn TIM5_IRQn
#define COLOR_TIM_IRQHandler TIM5_IRQHandler
```

这一串代码的意思是指定对应的定时器 tim5，并设置其时钟域为 RCC_APB1Periph_TIM5，接着定义 tim 的三个通道 1，2，3 的比较寄存器 CCR，后面改变这些值就可以改变相应的占空比。

最后两行代码在本次实验中是多余的，我看了整个 ppt 没有发现使用这两个中断定义的地方，应该在 stm32f4xx_it.c 中书写相应的中断服务函数才能实现具体的功能。

接着定义 rgb 颜色值并进行函数声明：

```
 #define COLOR_WHITE   0xFFFFFF
#define COLOR_BLACK   0x000000
#define COLOR_GREY    0xC0C0C0
#define COLOR_BLUE    0x0000FF
#define COLOR_RED     0xFF0000
#define COLOR_MAGENTA 0xFF00FF
#define COLOR_GREEN   0x00FF00
#define COLOR_CYAN    0x00FFFF
#define COLOR_YELLOW  0xFFFF00

void ColorLED_Config(void);
void SetRGBColor(uint32_t rgb);
void SetColorValue(uint8_t r,uint8_t g, uint8_t b);
```

然后围绕这三个函数分别讲解，第一个是 ColorLED_Config:

```
void ColorLED_Config(void)
{
    TIMx_GPIO_Config();

    TIM_Mode_Config();

    SetColorValue(0xff, 0xff, 0xff);
}
```

这个函数用于 tim5 的初始化配置，并将初始颜色设为白色（setcolorvalue 函数），接下来我们具体看 timx_gpio_config 的设置：

```
/*定义一个GPIO_InitTypeDef类型的结构体*/
GPIO_InitTypeDef GPIO_InitStructure;

/*开启LED相关的GPIO外设时钟*/
RCC_AHB1PeriphClockCmd(COLOR_LED1_GPIO_CLK | COLOR_LED2_GPIO_CLK | COLOR_LED3_GPIO_CLK,ENABLE);

GPIO_PinAFConfig(COLOR_LED1_GPIO_PORT, COLOR_LED1_PINSOURCE, COLOR_LED1_AF);
GPIO_PinAFConfig(COLOR_LED2_GPIO_PORT, COLOR_LED2_PINSOURCE, COLOR_LED2_AF);
GPIO_PinAFConfig(COLOR_LED3_GPIO_PORT, COLOR_LED3_PINSOURCE, COLOR_LED3_AF);

/* COLOR LED1 */

GPIO_InitStructure.GPIO_Pin = COLOR_LED1_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用模式，信号源来自于其他部件
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出模式
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //无上下拉模式
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(COLOR_LED1_GPIO_PORT, &GPIO_InitStructure);
```

这个过程和实验三里面的 gpio 初始化过程大致一样，我们可以将两个初始化的过程做一个对比：

```
/*定义一个GPIO_InitTypeDef类型的结构体*/
GPIO_InitTypeDef GPIO_InitStructure;

/*开启LED相关的GPIO外设时钟*/
RCC_AHB1PeriphClockCmd ( LED1_GPIO_CLK|
                         LED2_GPIO_CLK|
                         LED3_GPIO_CLK, ENABLE);

/*选择要控制的GPIO引脚*/
GPIO_InitStructure.GPIO_Pin = LED1_PIN;

/*设置引脚模式为输出模式*/
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;

/*设置引脚的输出类型为推挽输出*/
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;

/*设置引脚为上拉模式*/
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;

/*设置引脚速率为2MHz */
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

/*调用库函数，使用上面配置的GPIO_InitStructure初始化GPIO*/
GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStructure);
```

```
/*定义一个GPIO_InitTypeDef类型的结构体*/
GPIO_InitTypeDef GPIO_InitStructure;

/*开启LED相关的GPIO外设时钟*/
RCC_AHB1PeriphClockCmd(COLOR_LED1_GPIO_CLK | COLOR_LED2_GPIO_CLK | COLOR_LED3_GPIO_CLK,ENABLE);

GPIO_PinAFConfig(COLOR_LED1_GPIO_PORT, COLOR_LED1_PINSOURCE, COLOR_LED1_AF);
GPIO_PinAFConfig(COLOR_LED2_GPIO_PORT, COLOR_LED2_PINSOURCE, COLOR_LED2_AF);
GPIO_PinAFConfig(COLOR_LED3_GPIO_PORT, COLOR_LED3_PINSOURCE, COLOR_LED3_AF);

/* COLOR LED1 */

GPIO_InitStructure.GPIO_Pin = COLOR_LED1_PIN;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用模式，信号源来自于其他部件
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出模式
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //无上下拉模式
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(COLOR_LED1_GPIO_PORT, &GPIO_InitStructure);
```

（左侧为实验 3 中的 gpio 初始化，右侧是本次实验的初始化）

可以看到引脚模式由输出模式变为复用模式，引脚改为无上下拉的模式（实际上这个不影响），速率由 2MHz 变为 100MHz

除此之外，因为要进行引脚复用，这里还多出来了调用 gpio_pinafconfig 函数进行引脚和 tim5 的绑定.

通过使用实验 3 中 gpio 初始化多个引脚的方法，对本次实验的绿灯和蓝灯也进行相应的初始化：

```
// led2  GPIO初始化
GPIO_InitStructure.GPIO_Pin = COLOR_LED2_PIN;
GPIO_Init(COLOR_LED2_GPIO_PORT, &GPIO_InitStructure);

// led3  GPIO初始化
GPIO_InitStructure.GPIO_Pin = COLOR_LED3_PIN;
GPIO_Init(COLOR_LED3_GPIO_PORT, &GPIO_InitStructure);
```

下一个初始化函数是 TIM_Mode_Config，这个函数初始化了两个东西：

- 定时器的基本参数
- PWM 初始化

先看定时器的初始化：

```
TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
TIM_OCInitTypeDef  TIM_OCInitStructure;

RCC_APB1PeriphClockCmd(COLOR_TIM_CLK, ENABLE);

TIM_TimeBaseStructure.TIM_Period = 256-1;
TIM_TimeBaseStructure.TIM_Prescaler = ((SystemCoreClock/2)/1000000)*30-1;


TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1 ;//设置时钟分频系数:不分频
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式

TIM_TimeBaseInit(COLOR_TIM, &TIM_TimeBaseStructure);
```

首先使能了本次实验的定时器的时钟，并对设置计数器的溢出值为 255（由于计数是从 0 开始的，实际上会计数 256 次才结束一个计时周期）

接下来设置预分频，这里我在看代码的时候有些疑问，这个预分频的计算为什么这么复杂，它到底在算什么，而且还使用了 SystemCoreClock，后面询问了大模型，大模型给出的答案是系统主频（即 SystemCoreClock）过高，需要“提前降频”，这里的预分频器就是为了这个目的，大模型给出了一个计算公式：

1. **SystemCoreClock / 2 = 84 MHz**（因为 TIM5 在 APB1，总线分频）
2. **(84 MHz / 1 MHz) × 30 - 1 = 84 × 30 - 1 = 2519**

所以：

```
c                                                    复制  编辑

Prescaler = 2519
Period = 255
```

定时器计数频率（计一次的时间）：

$$f_{计数} = \frac{84\text{MHz}}{Prescaler + 1} = \frac{84\text{MHz}}{2520} = 33333.33\text{Hz}$$

所以定时器**每计一次需要的时间**是：

$$T_{tick} = \frac{1}{33333.33} \approx 30\mu s$$

计数频率 = 84MHz/（Prescaler 的值 + 1）

这里就解释了为什么使用 SystemCoreClock，SystemCoreClock 的值为 168MHz，其除以 2 正好是 84MHz，可以和公式分子的 84MHz（实际上是该寄存器的时钟源频率）抵消，接着除以 1 百万乘以 30，就相当于每次计数的时长为 30

* 10^-6 ，也就是 30 微秒

最后就是设置定时器时钟 CK_INT 频率与死区发生器以及数字滤波器采样时钟频率分频比以及计数模式（向上计数）。

再看 PWM 的初始化：

```
// PWM初始化
// 通道初始化
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;    // 设置PWM模式1
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;    // 打开输出通道
TIM_OCInitStructure.TIM_Pulse = 0;     // 设置初始占空比
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;  // 输出极性为低有效，即低电平时点亮led

TIM_OC1Init(COLOR_TIM, &TIM_OCInitStructure); // 使能通道1

TIM_OC1PreloadConfig(COLOR_TIM, TIM_OCPreload_Enable); // 使能通道1重载

TIM_OC2Init(COLOR_TIM, &TIM_OCInitStructure); // 使能通道2

TIM_OC2PreloadConfig(COLOR_TIM, TIM_OCPreload_Enable); // 使能通道2重载

TIM_OC3Init(COLOR_TIM, &TIM_OCInitStructure); // 使能通道3

TIM_OC3PreloadConfig(COLOR_TIM, TIM_OCPreload_Enable); // 使能通道3重载

TIM_ARRPreloadConfig(COLOR_TIM, ENABLE);     // 使能TIM重载寄存器
// 使能计数器
TIM_Cmd(COLOR_TIM, ENABLE);
```

首先按照 ppt 上的要求设置 PWM 为模式 1，并将输出通道打开（第二行的 enable），接下来设置占空比为 0%，即 led 初始为熄灭状态，并设置输出极性为低电平有效，最后就是使能通道并进行预装载（这里老师是用重载表示预装载，作为 c++程序员感觉实在是不严谨，c 里面没有重载机制，容易让人误解）

这里的预装载机制我专门查了一下，发现这种机制是基于一个影子寄存器的概念实现的，由于在用户更改灯的亮度（即更改占空比）时，有两个更改的时间，第一个是立即更改，另一个是本次 PWM 周期结束时更改，而预装载采用的是后者，因此，需要有一个寄存器将预装载的值存好，直到周期结束时才写入 CCR 比较寄存器，存储这个值的寄存器称为影子寄存器，每一个通道都一个影子寄存器。

然后使能预装载寄存器，即开启影子寄存器功能。

最后的最后，需要使能计数器，不然颜色不会更改。

接下来看第二个函数：SetRGBColor

```
void SetRGBColor(uint32_t rgb){
    uint8_t r = 0, g = 0, b = 0;
    r = (uint8_t)(rgb >> 16); // 取高8位，然后强制转化为8bit
    g = (uint8_t)(rgb >> 8);  // 取高16位，然后强制转化为8bit，相当于取了中间8位
    b = (uint8_t) rgb;        // 取低8位

    // 根据PWM表修改定时器的比较寄存器值
    COLOR_TIM->COLOR_LED1_CCRx = r;
    COLOR_TIM->COLOR_LED2_CCRx = g;
    COLOR_TIM->COLOR_LED3_CCRx = b;
}
```

这个函数是传入一个 4 字节的数（实际上只有低 24 位有效），并取出相应位置的值作为对应颜色的占空比，例如高 8 位表示红色（rgb 右移 16 位那一行），并用 PWM 表（实际上是一个结构体，如下图所示）找到对应的 CCR 比较寄存器，之后赋值。

```
typedef struct
{
  __IO uint16_t CR1;         /*!< TIM control register 1,              Address offset: 0x00 */
  uint16_t      RESERVED0;   /*!< Reserved, 0x02                                            */
  __IO uint16_t CR2;         /*!< TIM control register 2,              Address offset: 0x04 */
  uint16_t      RESERVED1;   /*!< Reserved, 0x06                                            */
  __IO uint16_t SMCR;        /*!< TIM slave mode control register,     Address offset: 0x08 */
  uint16_t      RESERVED2;   /*!< Reserved, 0x0A                                            */
  __IO uint16_t DIER;        /*!< TIM DMA/interrupt enable register,   Address offset: 0x0C */
  uint16_t      RESERVED3;   /*!< Reserved, 0x0E                                            */
  __IO uint16_t SR;          /*!< TIM status register,                 Address offset: 0x10 */
  uint16_t      RESERVED4;   /*!< Reserved, 0x12                                            */
  __IO uint16_t EGR;         /*!< TIM event generation register,       Address offset: 0x14 */
  uint16_t      RESERVED5;   /*!< Reserved, 0x16                                            */
  __IO uint16_t CCMR1;       /*!< TIM capture/compare mode register 1, Address offset: 0x18 */
  uint16_t      RESERVED6;   /*!< Reserved, 0x1A                                            */
  __IO uint16_t CCMR2;       /*!< TIM capture/compare mode register 2, Address offset: 0x1C */
  uint16_t      RESERVED7;   /*!< Reserved, 0x1E                                            */
  __IO uint16_t CCER;        /*!< TIM capture/compare enable register, Address offset: 0x20 */
  uint16_t      RESERVED8;   /*!< Reserved, 0x22                                            */
  __IO uint32_t CNT;         /*!< TIM counter register,                Address offset: 0x24 */
  __IO uint16_t PSC;         /*!< TIM prescaler,                       Address offset: 0x28 */
  uint16_t      RESERVED9;   /*!< Reserved, 0x2A                                            */
  __IO uint32_t ARR;         /*!< TIM auto-reload register,            Address offset: 0x2C */
  __IO uint16_t RCR;         /*!< TIM repetition counter register,     Address offset: 0x30 */
  uint16_t      RESERVED10;  /*!< Reserved, 0x32                                            */
  __IO uint32_t CCR1;        /*!< TIM capture/compare register 1,      Address offset: 0x34 */
  __IO uint32_t CCR2;        /*!< TIM capture/compare register 2,      Address offset: 0x38 */
  __IO uint32_t CCR3;        /*!< TIM capture/compare register 3,      Address offset: 0x3C */
  __IO uint32_t CCR4;        /*!< TIM capture/compare register 4,      Address offset: 0x40 */
  __IO uint16_t BDTR;        /*!< TIM break and dead-time register,    Address offset: 0x44 */
  uint16_t      RESERVED11;  /*!< Reserved, 0x46                                            */
  __IO uint16_t DCR;         /*!< TIM DMA control register,            Address offset: 0x48 */
  uint16_t      RESERVED12;  /*!< Reserved, 0x4A                                            */
  __IO uint16_t DMAR;        /*!< TIM DMA address for full transfer,   Address offset: 0x4C */
  uint16_t      RESERVED13;  /*!< Reserved, 0x4E                                            */
  __IO uint16_t OR;          /*!< TIM option register,                 Address offset: 0x50 */
  uint16_t      RESERVED14;  /*!< Reserved, 0x52                                            */
} TIM_TypeDef;
```

最后一个函数是 SetColorValue，直接分别传入三个字节，表示红绿蓝，其实就是上一个函数少了分解参数的过程：

```
void SetColorValue(uint8_t r, uint8_t g, uint8_t b){
    // 根据PWM表修改定时器的比较寄存器值
    COLOR_TIM->COLOR_LED1_CCRx = r;
    COLOR_TIM->COLOR_LED2_CCRx = g;
    COLOR_TIM->COLOR_LED3_CCRx = b;

}
```

然后看主函数的实现：

```
int main(void)
{
    ColorLED_Config();

    while(1){

        SetRGBColor(0xFF0000); // 仅红色亮
        Delay(0xFFFFFF);

        SetRGBColor(0x00FF00); // 仅绿色亮
        Delay(0xFFFFFF);

        SetRGBColor(0x0000FF); // 仅蓝色亮
        Delay(0xFFFFFF);
    }
}
```

主函数的实现比较简单，在设置完灯的占空比后，使用一个延时函数保证占空比在该时间段内不会被修改即可。

演示视频见提交文件。

2. 参考所给项目，自行用 PWM 实现三个通道(从 1-3)的占空比依次为自己学号后三个字节(从高-低)，观察 LED 的变化。(如学号为 10156160233，则后三个字节为 0x160233)

这个要求比较特别，每个人都有自己的学号，意味着呈现出来的亮度都不相同，我的学号后 6 位为 0x101477，故将示例实验中对应红灯的亮度改为 0x10，绿灯的亮度改为 0x14，蓝灯的亮度改为 0x77 即可：

```
int main(void)
{
    ColorLED_Config();

    while(1){

        // 我的学号为10235101477，后六位转化为16进制：0x101477
        SetRGBColor(0x100000); // 仅前两位亮
        Delay(0xFFFFFF);

        SetRGBColor(0x001400); // 仅中间两位亮亮
        Delay(0xFFFFFF);

        SetRGBColor(0x000077); // 仅后面两位亮
        Delay(0xFFFFFF);
    }
}
```

演示视频见提交文件。

3. 参考实验项目，完成红灯的呼吸灯，即使红灯由暗到亮，然后再回到暗，一直反复。（提示：修改红灯的占空比，也就是修改红灯对应的 CCR 寄存器的值）

如提示所言，由于占空比决定了灯的亮度，实现呼吸灯的思路就是通过循环改变占空比的值，另外，由于占空比和灯的亮度是线性相关的，我们可以直接使用迭代时的计数器表示占空比（相当于哈希了），最后，题目要求红灯由暗到亮，再由亮到暗，因此一个周期内包含两个循环，一个循环递增，一个循环递减：

```
int main(void)
{
    ColorLED_Config();


    while(1){
    for(int i = 0;i < 256;i++){
        SetColorValue(i, 0, 0);
        Delay(0xFFFFF);
    }

    for(int i = 255;i > -1;i--){
        SetColorValue(i, 0, 0);
        Delay(0xFFFFF);
    }
    }
}
```

需要注意的是，0xFF = 255，这个值是占空比的最大值，因此迭代的计数器最

大值是 255，同时为了使灯的亮度变化明显，我将 delay 的时间改为了 0xFFFFF。

演示视频见提交文件。

4. **把示例中的延时函数改用基本定时器定为 1s。**

该实验的原理和之前那次拓展实验的原理一样，但是我在实现的过程却因为一个编译器的特性而一直实验失败，不过后面发现原因后实验也顺利成功，此处便记录相关尝试。

为实现延时函数为固定的一秒，而且又要使用之前的基本定时器设置时间，此处再次解释一遍基本定时器计算时间的方式，由于基本定时器的时钟源频率为 90MHz（之前讲过通用寄存器的时钟源频率为 84MHz，那个分子的值），首先定时器会根据这个时钟频率进行预分频，此处我才用的预分频就是按照 ppt 内的预分频：

```
TIM_TimeBaseStructure.TIM_Prescaler = 9000-1;
```

此处减 1 的原因不再赘述，9000*1/90M，得到 100 微秒，也就是说，预分频后的 tick+1 的时间是 100 微秒，由于 ppt 的要求是 1 秒，100 微秒 * 10000 就是 1 秒，因此分频改为 10000 - 1 即可：

```
TIM_TimeBaseStructure.TIM_Period = 10000-1;
```

其他的初始化过程和之前的扩展实验相同，此处不再赘述

接下来是延时函数的改造：

由于每个计时周期结束时只会跳转到中断函数来处理本周期，因此，我的第一个思路是把中断函数内的代码通过某种形式移到 Delay 函数中，其中我发现之前的中断函数有一个函数用于判断是不是真的产生了中断，便进行了如下改变：

```
static void  Delay() // 简单的延时函数，固定延时1秒
{
    // for(; nCount != 0; nCount--);
    // TIM_Cmd(TIM7, ENABLE);
    while(1){
        // 确保中断位设置
        if ( TIM_GetITStatus( TIM7, TIM_IT_Update) != RESET )
        {
        TIM_ClearITPendingBit(TIM7 , TIM_IT_Update);
            break;
        }
    }
    // TIM_Cmd(TIM7, DISABLE);
}
```

（上下各有一个 TIM_Cmd 的原因是后面自己 1.0 版本没做对后加上的，我查了资料发现是通过这两个语句控制计数器的启用与停止，不过加上了也不对就是了）

但是这是无效的，而且相当危险，这个函数本身是***验证***中断是否产生，而不是***判断***中断是否产生，第一次实验失败。

后面我又想了一个办法，是通过一个额外的全局变量（定义在 stm32f4xx_it.c

里面），当该变量为 0 时，主线程在 delay 函数中就会阻塞，只有在中断发生后，delay 被置为 1，回到主函数，才会停止 delay 函数的执行，下面是变量定义，中断函数的实现以及 delay 函数的实现：

```
uint8_t delay_finished = 0;

void  BASIC_TIM_IRQHandler (void)
{
    if (TIM_GetITStatus(BASIC_TIM, TIM_IT_Update) != RESET)
    {
        TIM_ClearITPendingBit(BASIC_TIM, TIM_IT_Update);
        delay_finished = 1;
        TIM_Cmd(BASIC_TIM, DISABLE);
    }


}

void  Delay() // 简单的延时函数，固定延时1秒
{
    delay_finished = 0;
    TIM_Cmd(BASIC_TIM, ENABLE);
    while (!delay_finished);  // 等待中断设置标志
}
```

（Delay_finished 定义在 stm32f4xx_it.c 中，Delay 函数定义在主文件中）

具体实现逻辑如下：

首先进入 delay 函数，将 delay_finished 变量设为 0，然后激活定时器，此时定时器开始计时，下面的 while 循环则是实现忙等。

接着等待时间结束触发中断，在中断函数内，delay_finished 被设置为 1，并在同时关闭定时器。

回到 Delay 函数，while 循环下一次判断 delay_finished 的值时就会跳出循环，这样就成功设置了延时时间为 1 秒的延时函数。

这个 2.0 版本的实现逻辑上是没有问题的，问题出在 delay_finished 没有加上阻止编译器将其优化的 volatile 关键字，这会导致编译器将 while 循环优化成一个死循环（delay_finished 的逻辑取反可以提前算出来，为 1，而且编译器不知道我在中断函数内将其取反了），导致实验失败。

这个问题的解释相当简单但是找到问题的过程却相当困难，由于最开始并没有编译器优化而导致实验失败的例子，对编译器导致的问题是相当不敏感，一直以为是实现逻辑或者是理解不够透彻，将实验的代码改了很多次，甚至将 extern 的解释以及可能出现的问题看了很多次（因为我在不同文件里引用了同一个变量 delay_finished），最后甚至将 Delay 函数写到了 stm32f4xx_it.c 文件中避免使用 extern，但是很显然，这些方式都失败了，最后询问了同学，他们指出可能是编译器优化的问题才解决。

演示视频见提交文件。

五、实验总结

本次实验我收获颇多，尤其是工程方面的经验。对于 ppt 上的基础实验部分，我初步了解了 PWM 的原理以及其和通用定时器的结合使用，并对定时器的预分频机制有了更加深入的了解。在扩展实验中，我第一次遇到了因编译器优化产生的问题，从某种意义上说，这是一次特别好的实验经历，这为我之后进行嵌入式开发提供了非常宝贵的工程经验，我也真实体验到了遇到问题却查资料查不出来的情况，整个过程虽然相当崩溃，结果还是实验成功了，也知道了失败的原因。