零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

4 配套板子，全面玩转。

5 强强联合，不断更新。

野火团队 Wild Fire Team

0、友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，与野火同行，乐意惬无边。

另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

5、流水灯的前后今生

通过前面的内容，读者对库仅仅是建立了一个非常模糊的印象。

作为大家的第一个 STM32 例程，野火认为很有必要进行足够深入的分析，才能从*根本*上扫清读者对使用库函数的*困惑*。而且，只要读者利用这个 LED 例程，真正领会了库开发的流程以及原理，再进行其它外设的开发就变得相当简单了。

所以本章的任务是：

- 从 STM32 库的*实现原理*上解答 库到底是什么、为什么要用库、用库与直接配置寄存器的区别等问题。
- 让读者了解具体利用库的开发流程，熟悉库函数的结构，达到举一反三的效果，这次可就不是喝稀粥了，保证有吃干饭，所学就是所用的效果。

5.1 STM32 的 GPIO

想要控制 LED 灯，当然是通过控制 STM32 芯片的 I/O 引脚*电平的高低*来实现。在 STM32 芯片上，I/O 引脚可以被软件设置成各种*不同的功能*，如输入或输出，所以被称为 GPIO (General-purpose I/O)。而 GPIO 引脚又被分为 GPIOA、GPIOB……GPIOG 不同的组，每组端口分为 0~15，共 16 个*不同的引脚*，对于不同型号的芯片，端口的组和引脚的数量不同，具体请参考相应芯片型号的 datasheet。

于是，控制 LED 的步骤就自然整理出来了：

1. GPIO 端口引脚多 --> 就要选定需要控制的*特定引脚*
2. GPIO 功能如此丰富 --> 配置需要的*特定功能*
3. 控制 LED 的亮和灭 --> 设置 GPIO 输出电压的*高低*

继续思考，要控制 GPIO 端口，就要涉及到控制相关的寄存器。这时我们就要查一查与 GPIO 相关的寄存器了，可以通过《STM32 参考手册》来查看，见图 5-1

图 5-1

图中的 7 个寄存器，相应的功能在文档上有详细的说明。可以分为以下 4 类，其功能简要概括如下：

1. 配置寄存器：选定 GPIO 的*特定功能*，最基本的如：选择作为输入还是输出端口。
2. 数据寄存器：保存了 GPIO 的*输入电平* 或 将要*输出的电平*。
3. 位控制寄存器：设置某引脚的*数据* 为 1 或 0，控制输出的电平。
4. 锁定寄存器：设置某*锁定引脚*后，就不能修改其配置。

注：要想知道其功能严谨、详细的描述，请读者养成习惯在正式使用时，要以官方的 datasheet 为准，在这里只是简单地概括其功能进行说明。

关于寄存器名称上*标号 x* 的意义，如：GPIOx_CRL、GPIOx_CRH ，这个 x 的取值可以为图中括号内的值(A……E)，表示这些寄存器也跟 GPIO 一样，也是*分组*的。也就是说，对于端口 GPIOA 和 GPIOB，*它们都有互不相干的一组寄存器*，如控制 GPIOA 的寄存器名为 GPIOA_CRL、GPIOA_CRH 等，而控制 GPIOB 的则是不同的、被命名为 GPIOB_CRL、GPIOB_CRH 等寄存器。

我们的程序代码以野火 STM32 第二代开发板为例，根据其硬件连接图来分析，见图 5-2 及图 5-3 **错误！未找到引用源。**

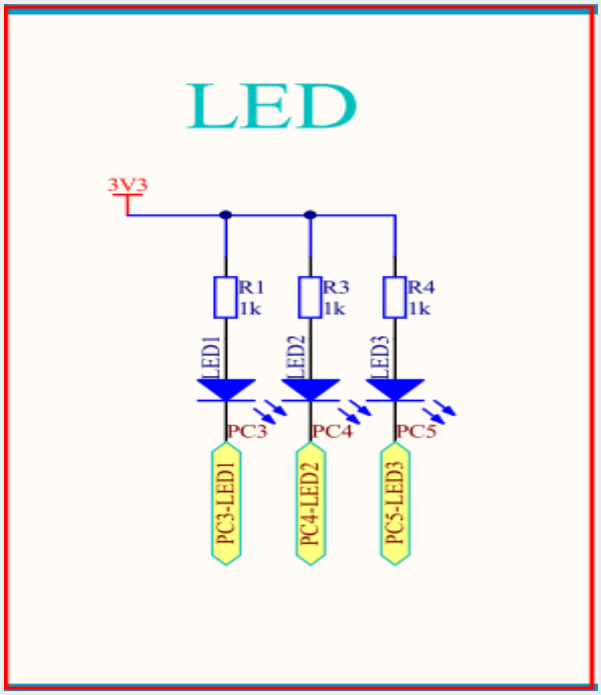


图 5-2

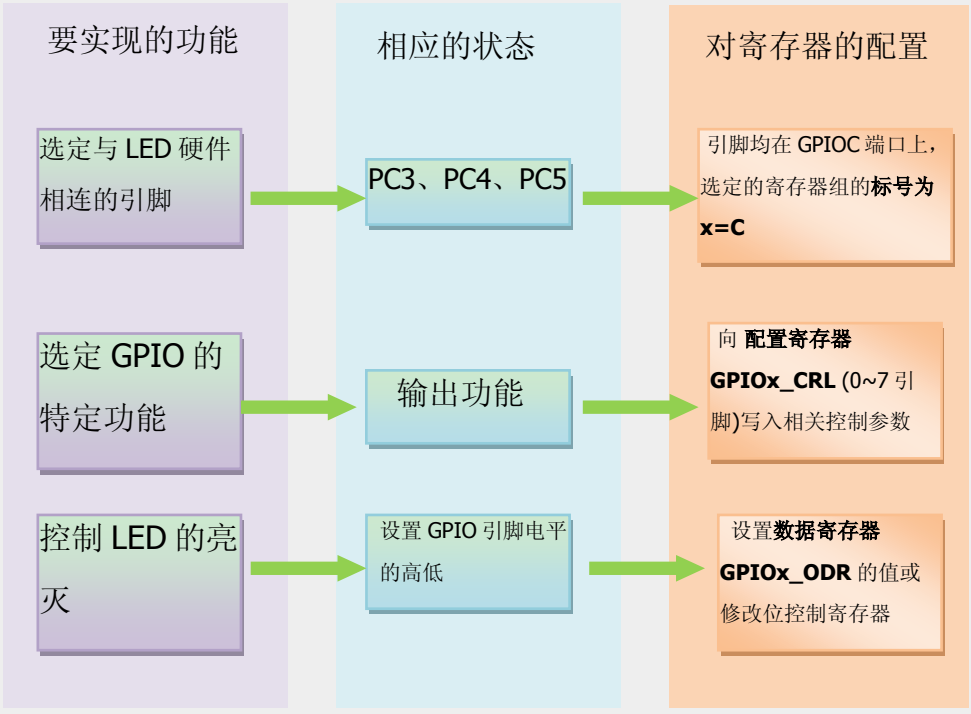


图 5-3

从这个图我们可以知道 STM32 的功能，实际上也是通过配置寄存器来实现的。配置寄存器的具体参数，需要参考《STM32 参考手册》的寄存器说明。见图 5-4。

8.2.2 端口配置高寄存器(GPIOx_CRH) (x=A..E)

偏移地址：0x04

复位值：0x4444 4444

每 4 个寄存器位配置一个引脚　　*这 4 位控制 pin12*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[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
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 8…15) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位9:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 8…15) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度50MHz

CNFy:向这两个位写入不同的值，设置引脚为不同的功能。y 表示第 y 个引脚。

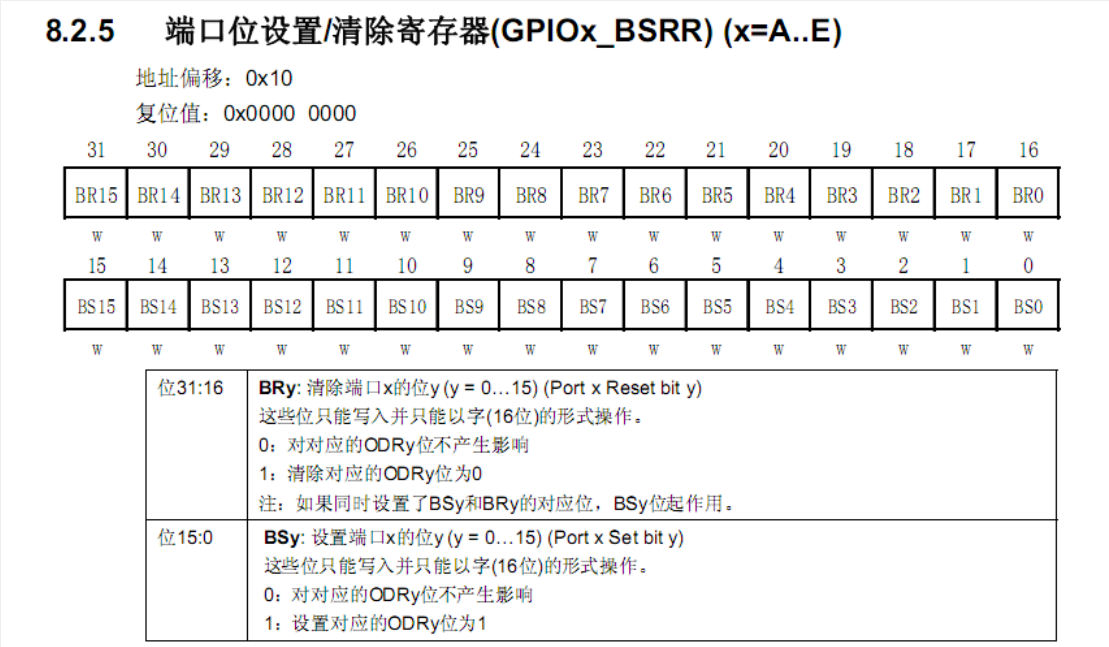
MODEy:向这两个位写入不同的值，设置引脚为不同的最大输出速率，或设置为输入模式。

图 5-4

如图，对于 GPIO 端口，每个端口有 16 个引脚，*每个引脚* 的模式由寄存器的 ***4 个位***控制，每四位又分为两位控制引脚配置***(CNFy[1:0])***，两位控制引脚的 模式及最高速度***(MODEy[1:0])***，其中 y 表示第 y 个引脚。这个图是 GPIOx_CRH 寄存器的说明，配置 GPIO 引脚模式的一共有两个寄存器，CRH 是高寄存器，用来配置高 8 位引脚：pin8~pin15。还有一个称为 CRL 寄存器，如果我们要配置 pin0~pin7 引脚，则要在寄存器 CRL 中进行配置。

举例说明对 CRH 的寄存器的配置：当给 GPIOx_CRH 寄存器的*第 28 至 29 位*设置为参数***"11"***，并在*第 30 至 31 位* 设置为参数***"00"***，则把 x 端口*第 15 个引脚* 的模式配置成了***"输出的最大速度为 50MHz 的 通用推挽输出模式、"***，其它引脚可通过其 GPIOx_CRH 或 GPIOx_CRL 的其它寄存器位来配置。至于 x 端口的 x 是指端口 GPIOA 还是 GPIOB 还要具体到不同的寄存器基址，这将在后面分析。

接下来分析要控制引脚电平高低，需要对寄存器进行什么具体的操作。见图 5-5。



8.2.5 端口位设置/清除寄存器(GPIOx_BSRR) (x=A..E)

地址偏移：0x10

复位值：0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	**BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0 注：如果同时设置了BSy和IBRy的对应位，BSy位起作用。
位15:0	**BSy**: 设置端口x的位y (y = 0…15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：设置对应的ODRy位为1

图 5-5

由寄存器说明图可知，一个引脚 y 的输出数据由 GPIOx_BSRR 寄存器位的 2 个位来控制分别为 ***BRy (Bit Reset y)***和 ***BSy (Bit Set y)***，***BRy*** 位用于写 1 清零，使引脚输出*低*电平，***BSy*** 位用来写 1 置 1，使引脚输出*高*电平。而对这两个位进行写零都是无效的。(还可以通过设置寄存器 ODR 来控制引脚的输出。)

例如：对 x 端口的寄存器 GPIOx_BSRR 的***第 0 位(BS0)*** 进行写 1，则 x 端口的第 0 引脚被设置为 1，输出*高电平*，若要令第 0 引脚再输出*低电平*，则需要向 GPIOx_BSRR 的***第 16 位(BR0)*** 写 1。

5.2 STM32 的地址映射

温故而知新——stm32f10x.h 文件

首先请大家回顾一下在 51 单片机上点亮 LED 是怎样实现的。这太简单了，几行代码就搞定。

```
1.  #include<reg52.h>
2.  int main (void)
3.  {
```

```
    P0=0;
  while(1);
}
```

以上代码就可以点亮 P0 端口与 LED 阴极相连的 LED 灯了，当然，这里省略了启动代码。为什么这个 ***P0 =0;*** 句子就能控制 P0 端口为低电平?很多刚入门 51 单片机的同学还真解释不来，关键之处在于这个代码所包含的头文件 ***<reg52.h>***。

在这个文件下有以下的定义：

```
/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;
```

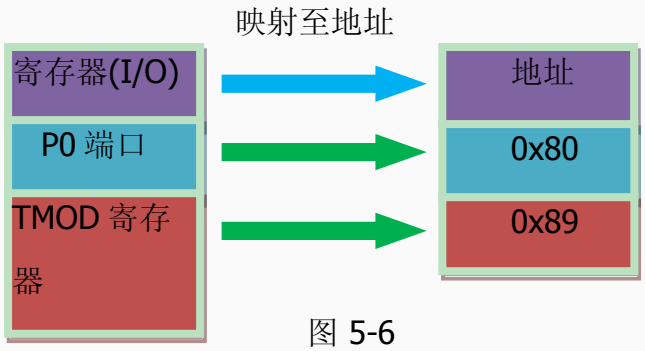


图 5-6

这些定义被称为*地址映射*。

所谓地址映射，就是将芯片上的*存储器* 甚至 ***I/O*** 等资源与地址建立一一对应的关系。如果某地址对应着某*寄存器*，我们就可以运用 c 语言的指针来寻址并修改这个地址上的内容，从而实现修改该寄存器的内容。

正是因为 ***<reg52.h>***头文件中有了对于各种*寄存器*和 ***I/O*** 端口的*地址映射*，我们才可以在 51 单片机程序中方便地使用 ***P0 =0xFF; TMOD =0xFF*** 等赋值句子对寄存器进行配置，从而控制单片机。

Cortex-M3 的地址映射也是类似的。Cortex-M3 有 32 根地址线，所以它的寻址空间大小为 2^{32} bit=4GB。ARM 公司设计时，预先把这 4GB 的寻址空间大致地分配好了。它把地址从 0x4000 0000 至 0x5FFF FFFF(512MB)的地址分配给片上外设。通过把片上外设的寄存器映射到这个地址区，就可以简单地以访

问内存的方式，访问这些外设的寄存器，从而控制外设的工作。结果，片上外设可以使用 C 语言来操作。M3 存储器映射见图 5-7

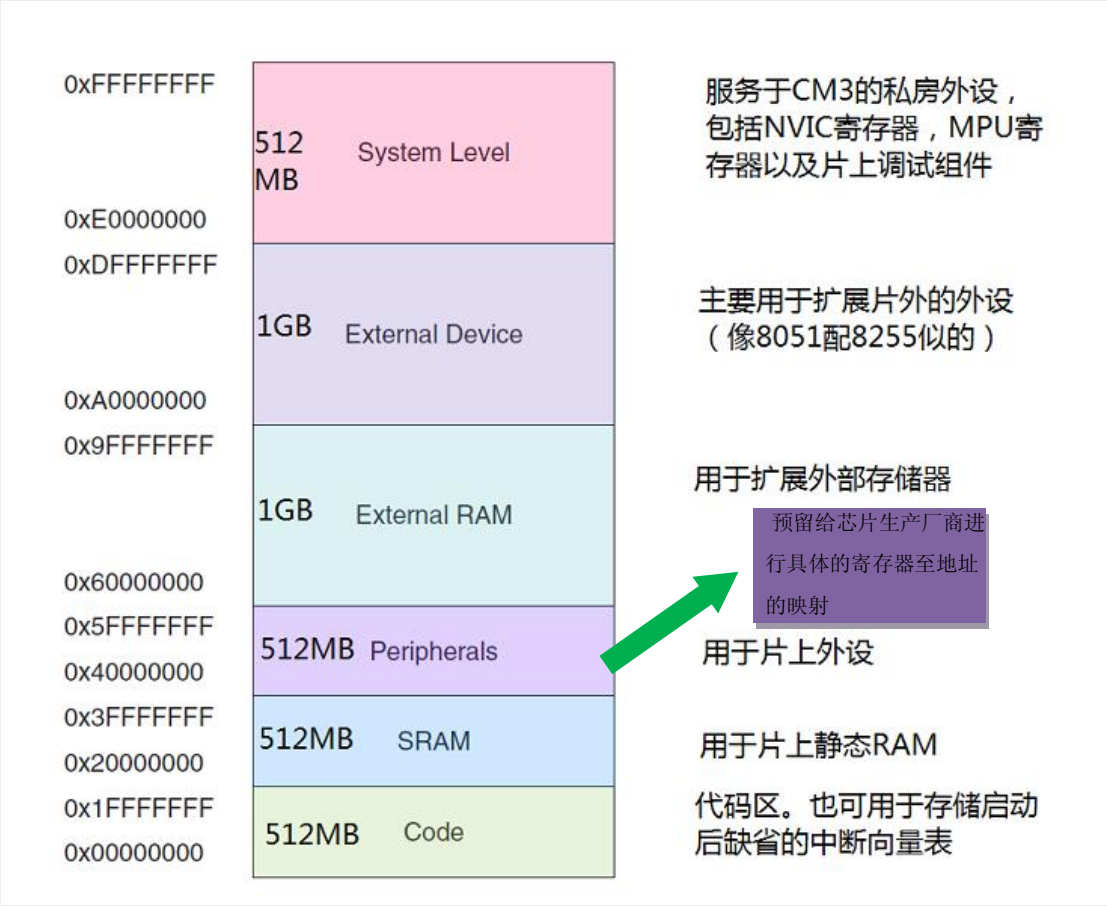


图 5-7

stm32f10x.h 这个文件中重要的内容就是把 STM32 的所有寄存器进行地址映射。如同 51 单片机的***<reg52.h>***头文件一样，***stm32f10x.h*** 像一个大表格，我们在使用的时候就是通过宏定义进行类似查表的操作，大家想像一下没有这个文件的话，我们要怎样访问 STM32 的寄存器？有什么缺点？

不进行这些宏定义的缺点有：

1、地址容易写错

2、我们需要查大量的手册来确定哪个地址对应哪个寄存器

3、看起来还不好看，且容易造成编程的错误，效率低，影响开发进度。

当然，这些工作都是由 ST 的固件工程师来完成的，只有设计 M3 的人才是最了解 M3 的，才能写出完美的库。

在这里我们以外接了 LED 灯的外设 GPIOC 为例，在这个文件中有这样的一系列宏定义：

```
1.  #define GPIOC_BASE              (APB2PERIPH_BASE + 0x1000)
2.  #define APB2PERIPH_BASE         (PERIPH_BASE + 0x10000)
3.  #define PERIPH_BASE             ((uint32_t)0x40000000)
```

这几个宏定义是从文件中的几个部分抽离出来的，具体的读者可参考 ***stm32f10x.h 源码。***

外设基地址

首先看到 ***PERIPH_BASE*** 这个宏，宏展开为 ***0x4000 0000***,并把它强制转换为 uint32_t 的 32 位类型数据，这是因为地 STM32 的地址是 32 位的，是不是觉得 ***0x4000 0000*** 这个地址很熟？是的，这个是 Cortex-M3 核分配给片上外设的从 0x4000 0000 至 0x5FFF FFFF 的 512MB 寻址空间中 的第一个地址，我们把 ***0x4000 0000*** 称为外设基地址。

总线基地址

接下来是宏 ***APB2PERIPH_BASE***，宏展开为 ***PERIPH_BASE（外设基地址）*** 加上偏移地址 ***0x1 0000，*** 即指向的地址为 0x4001 0000。这个 ***APB2PERIPH_BASE*** 宏是什么地址呢？STM32 不同的外设是挂载在不同的总线上的，见图 5-8。有 AHB 总线、APB2 总线、APB1 总线，挂载在这些总线上

的外设有特定的地址范围。

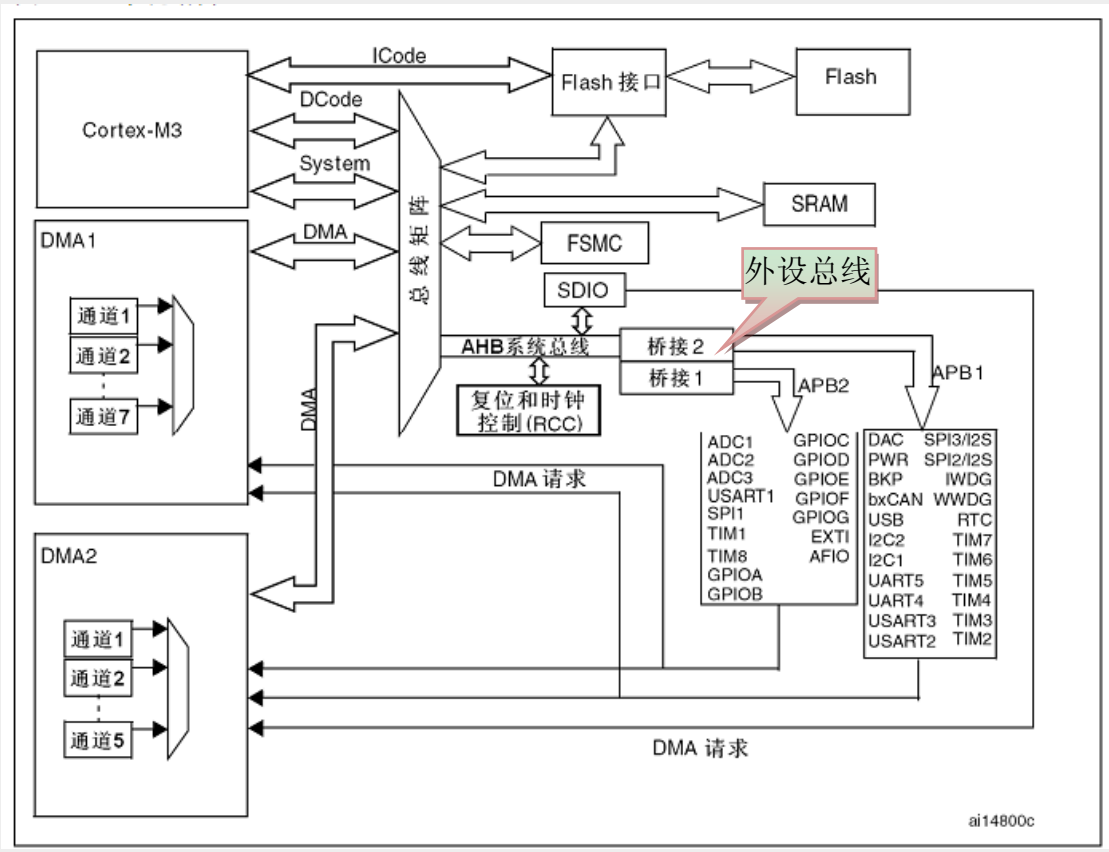


图 5-8

其中像 GPIO、串口 1、ADC 及部分定时器是挂载这个被称为 APB2 的总线上，挂载到 APB2 总线上的外设地址空间是从 0x4001 0000 至地址 0x4001 3FFF。这里的第一个地址，也就是 0x4001 0000,被称为 ***APB2PERIPH_BASE*** (APB2 总线外设的基地址)。

而 APB2 总线基地址相对于外设基地址的偏移量为 0x1 0000 个地址，即为 APB2 相对外设基地址的偏移地址。

见表：

地址范围	总线	总线基地址	总线基地址相对外设基地址（0x4000 000）的偏移量
0x4001 8000 -0x5003 FFFF	AHB	0x4001 8000	0x1 8000

0x4001 0000 - 0x4001 7FFF	APB2	0x4001 0000	0x1 0000
0x4000 0000 - 0x4000FFFF	APB1	0x4000 0000	0x0 0000

由这个表我们可以知道，***stm32f10x.h*** 这个文件中必然还有以下的宏：

```
1.  #define APB1PERIPH_BASE       PERIPH_BASE
```

因为偏移量为零，所以 APB1 的地址直接就等于外设基地址

寄存器组基地址

最后到了宏 ***GPIOC_BASE，宏展开为 APB2PERIPH_BASE*** (APB2 总线外设的基地址)加上相对 APB2 总线基地址的偏移量 ***0x1000*** 得到了 GPIOC 端口的寄存器组的基地址。这个所谓的寄存器组又是什么呢？它包括什么寄存器？

细看 ***stm32f10x.h*** 文件，我们还可以发现以下类似的宏：

```
1.  #define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
2.  #define GPIOB_BASE            (APB2PERIPH_BASE + 0x0C00)
3.  #define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)
4.  #define GPIOD_BASE            (APB2PERIPH_BASE + 0x1400)
```

除了 GPIOC 寄存器组的地址，还有 GPIOA、GPIOB、GPIOD 的地址，并且这些地址是不一样的。

前面提到，每组 GPIO 都对应着独立的一组寄存器，查看 stm32 的 datasheet，看到寄存器说明如下图：

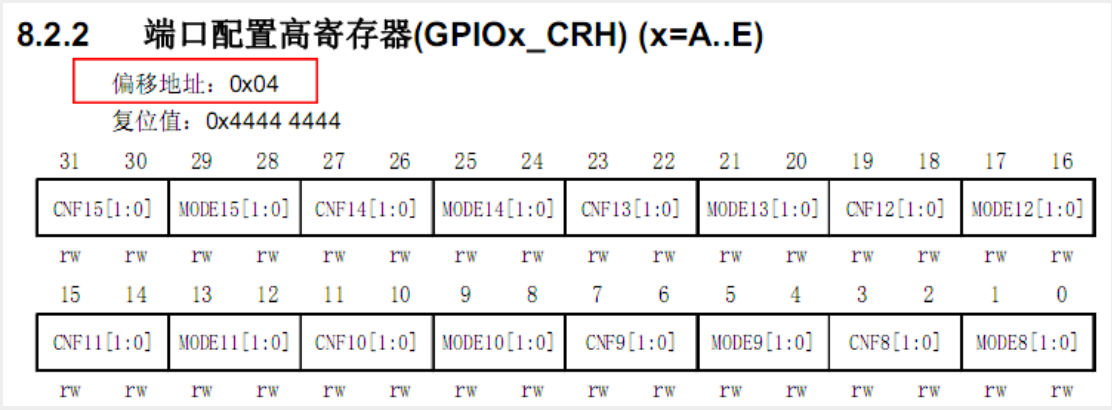


图 5-9

注意到这个说明中有一个偏移地址：0x04，这里的偏移地址的是相对哪个地址的偏移呢？下面进行举例说明。

对于 GPIOC 组的寄存器，GPIOC 含有的 ***端口配置高寄存器(GPIOC_CRH)*** 寄存器地址为：***GPIOC_BASE +0x04。***

假如是 GPIOA 组的寄存器，则 GPIOA 含有的 ***端口配置高寄存器(GPIOA_CRH)***寄存器地址为：***GPIOA_BASE+0x04。***

也就是说，这个偏移地址，就是该寄存器 相对所在寄存器组基地址的偏移量。

于是，读者可能会想，大概这个文件含有一个类似如下的宏(当初野火也是这么想的)：

```
1.  #define  GPIOC_CRH   (GPIOC_BASE + 0x04)
```

这个宏，定义了 GPIOC_CRH 寄存器的具体地址，然而，在 stm32f10x.h 文件中并没有这样的宏。ST 公司的工程师采用了更巧妙的方式来确定这些地址，请看下一小节——STM32 库对寄存器的封装。

5.3 STM32 库对寄存器的封装

ST 的工程师用结构体的形式，封装了寄存器组，c 语言结构体学的不好的同学，可以在这里补补课了。在 stm32f10x.h 文件中，有以下代码：

```
1.  #define GPIOA              ((GPIO_TypeDef *) GPIOA_BASE)
2.  #define GPIOB              ((GPIO_TypeDef *) GPIOB_BASE)
3.  #define GPIOC              ((GPIO_TypeDef *) GPIOC_BASE)
```

有了这些宏，我们就可以定位到具体的寄存器地址，在这里发现了一个陌生的类型 ***GPIO_TypeDef***，追踪它的定义，可以在 ***stm32f10x.h*** 文件中找到如下代码：

```
1.  typedef struct
2.  {
3.    __IO uint32_t CRL;
4.    __IO uint32_t CRH;
5.    __IO uint32_t IDR;
6.    __IO uint32_t ODR;
7.    __IO uint32_t BSRR;
8.    __IO uint32_t BRR;
```

```
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

其中 __IO 也是一个 ST 库定义的宏，宏定义如下：

```
#define     __O     volatile /*!< defines 'write only' permissions  */
#define     __IO    volatile /*!< defines 'read / write' permissions */
```

volatitle 是 c 语言的一个关键字，有关 volatitle 的用法可查阅相关的 C 语言书籍。

回到 ***GPIO_TypeDef*** 这段代码，这个代码用 ***typedef*** 关键字声明了名为 ***GPIO_TypeDef*** 的结构体类型，结构体内又定义了 7 个 ***__IO uint32_t*** 类型的变量。这些变量每个都为 32 位，也就是每个变量占内存空间 4 个字节。在 c 语言中，结构体内变量的存储空间是连续的，也就是说假如我们定义了一个 ***GPIO_TypeDef***，这个结构体的首地址（变量 CRL 的地址）若为 ***0x4001 1000***， 那么结构体中第二个变量（CRH）的地址即为 ***0x4001 1000 +0x04***，加上的这个 ***0x04***，正是代表 4 个字节地址的偏移量。

细心的读者会发现，这个 0x04 偏移量，正是 GPIOx_CRH 寄存器相对于所在寄存器组的偏移地址，见图 5-9。同理，***GPIO_TypeDef*** 结构体内其它变量的偏移量，也和相应的寄存器偏移地址相符。于是，只要我们匹配了结构体的首地址，就可以确定各寄存器的具体地址了。

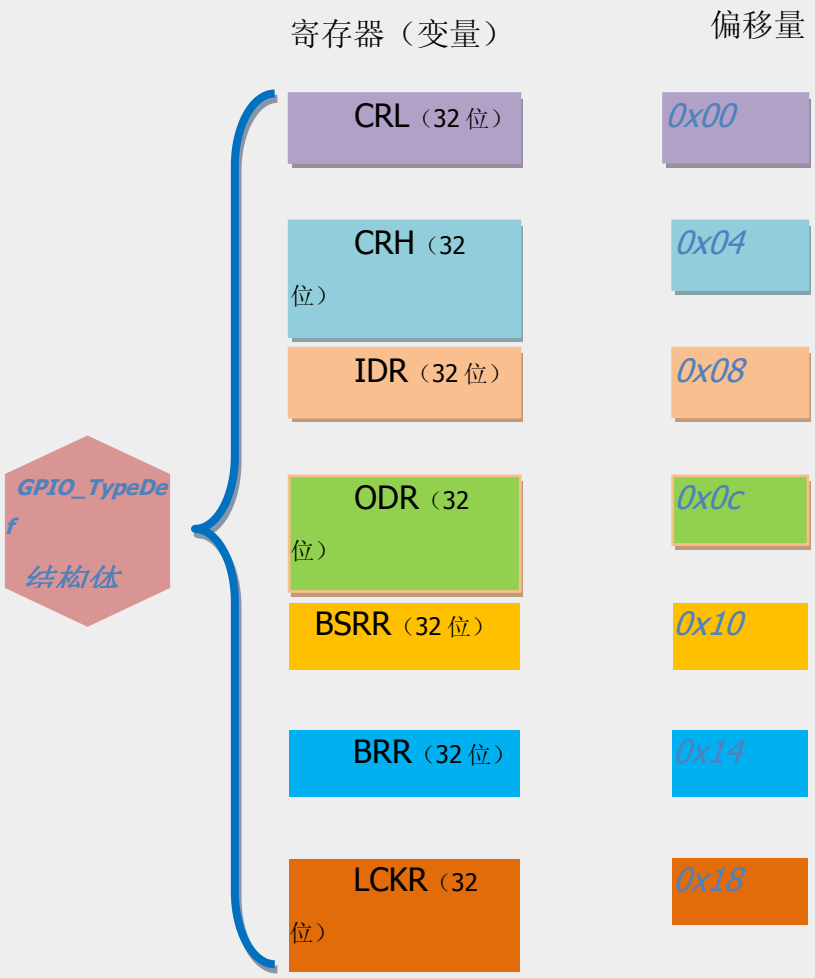


图 0-10

有了这些准备，就可以分析本小节的第一段代码了：

```
4.  #define GPIOA                   ((GPIO_TypeDef *) GPIOA_BASE)
5.  #define GPIOB                   ((GPIO_TypeDef *) GPIOB_BASE)
6.  #define GPIOC                   ((GPIO_TypeDef *) GPIOC_BASE)
```

GPIOA_BASE 在上一小节已解析，是一个代表 GPIOA 组寄存器的基地址。***(GPIO_TypeDef *)*** 在这里的作用则是把 ***GPIOA_BASE*** 地址转换为 ***GPIO_TypeDef*** 结构体指针类型。

有了这样的宏，以后我们写代码的时候，如果要修改 GPIO 的寄存器，就可以用以下的方式来实现。代码分析见注释。

```
1.  GPIO_TypeDef * GPIOx;          //定义一个 GPIO_TypeDef 型结构体指针 GPIOx
2.  GPIOx = GPIOA;                 //把指针地址设置为宏 GPIOA 地址
3.  GPIOx->CRL = 0xffffffff;       //通过指针访问并修改 GPIOA_CRL 寄存器
```

通过类似的方式，我们就可以给具体的寄存器写上适当的参数，控制 STM32 了。是不是觉得很巧妙？但这只是库开发的皮毛，而且实际上我们并不是这样使用库的，库为我们提供了更简单的开发方式。M3 的库可谓尽情绽放了 c 的魅力，如果你是单片机初学者，c 语言初学者，那么请你不要放弃与 M3 库邂逅的机会。是否选择库，就差你一个闪亮的回眸。

5.4 STM32 的时钟系统

STM32 芯片为了实现低功耗，设计了一个功能完善但却非常复杂的时钟系统。普通的 MCU，一般只要配置好 GPIO 的寄存器，就可以使用了，但 STM32 还有一个步骤，就是开启外设时钟。

5.4.1 时钟树&时钟源

首先，从整体上了解 STM32 的时钟系统。见图 0-11

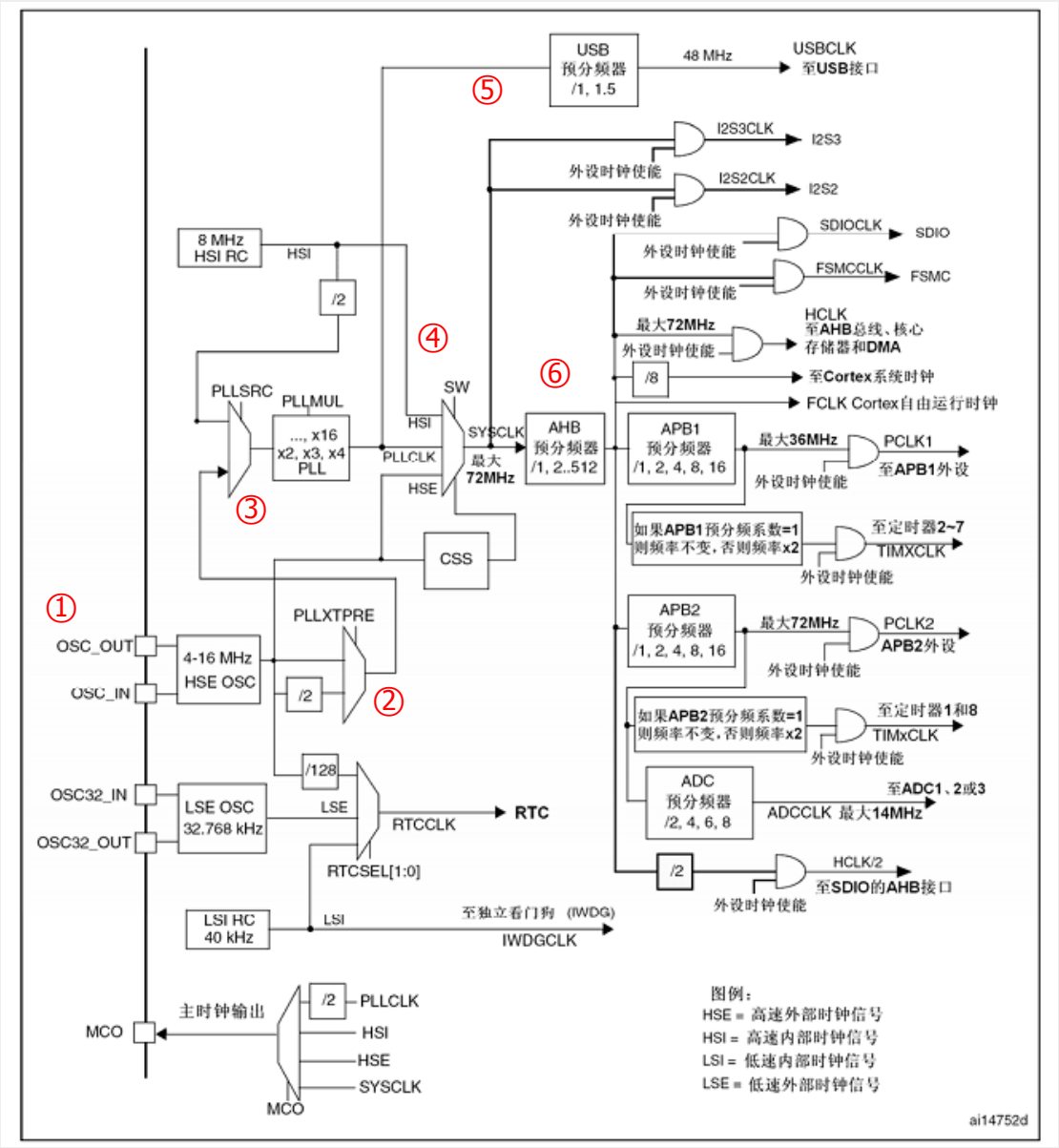


图 0-11

这个图说明了 STM32 的时钟走向，从图的左边开始，从时钟源一步步分配到外设时钟。

从时钟频率来说，又分为*高速时钟*和*低速时钟*，高速时钟是提供给芯片主体的主时钟，而低速时钟只是提供给芯片中的 RTC（实时时钟）及独立看门狗使用。

从芯片角度来说，时钟源分为*内部时钟*与*外部时钟源*，内部时钟是在芯片内部 RC 振荡器产生的，起振较快，所以时钟在芯片刚上电的时候，默认使用内部高速时钟。而外部时钟信号是由外部的晶振输入的，在精度和稳定性上都有很大优势，所以上电之后我们再通过软件配置，转而采用外部时钟信号。

所以，STM32 有以下 4 个时钟源：

高速外部时钟（HSE）：以外部晶振作时钟源，晶振频率可取范围为 4~16MHz，我们一般采用 8MHz 的晶振。

高速内部时钟（HSI）： 由内部 RC 振荡器产生，频率为 8MHz，但不稳定。

低速外部时钟（LSE）：以外部晶振作时钟源，主要提供给实时时钟模块，所以一般采用 32.768KHz。*野火 M3 实验板上用的是 32.768KHz，6p 负载规格的晶振。*

低速内部时钟（LSI）：由内部 RC 振荡器产生，也主要提供给实时时钟模块，频率大约为 40KHz。

5.4.2 高速外部时钟（HSE）

我们以最常用的高速外部时钟为例分析，首先假定我们在外部提供的晶振的频率为 8MHz 的。

1、 从左端的 OSC_OUT 和 OSC_IN 开始，这两个引脚分别接到外部晶振的两端。

2、 8MHz 的时钟遇到了第一个*分频器 PLLXTPRE*（HSE divider for PLL entry），在这个分频器中，可以通过寄存器配置，选择它的输出。它的输出时钟可以是对输入时钟的*二分频或不分频*。本例子中，我们选择不分频，所以经过 PLLXTPRE 后，还是 8MHz 的时钟。

3、 8MHz 的时钟遇到*开关 PLLSRC（PLL entry clock source）*，我们可以选择其输出，输出为外部高速时钟（HSE）或是内部高速时钟（HSI）。这里选择输出为 HSE，接着遇到*锁相环 PLL*，具有倍频作用，在这里我们可以输入*倍频因子 PLLMUL（PLL multiplication factor）*，哥们，你要是想超频，就得在这个寄存器上做手脚啦。经过 PLL 的时钟称为 *PLLCLK*。倍频因子我们设定为 9 倍频，也就是说，经过 PLL 之后，我们的时钟从原来 8MHz 的 HSE 变为 72MHz 的 PLLCLK。

4、 紧接着又遇到了一个*开关 **SW***，经过这个开关之后就是 STM32 的*系统时钟（**SYSCLK**）*了。通过这个开关，可以切换 SYSCLK 的时钟源，可以选择为 HSI、PLLCLK、HSE。我们选择为 PLLCLK 时钟，所以 SYSCLK 就为 72MHz 了。

5、 PLLCLK 在输入到 SW 前，还流向了 USB 预分频器，这个分频器输出为 USB 外设的时钟（USBCLK）。

6、 回到 SYSCLK，SYSCLK 经过 ***AHB** 预分频器*，分频后再输入到其它外设。如输出到称为 HCLK、FCLK 的时钟，还直接输出到 SDIO 外设的 SDIOCLK 时钟、存储器控制器 FSMC 的 FSMCCLK 时钟，和作为 APB1、APB2 的预分频器的输入端。本例子设置 AHB 预分频器不分频，即输出的频率为 72MHz。

7、 ***GPIO** 外设*是挂载在 APB2 总线上的， APB2 的时钟是 ***APB2** 预分频器*的输出，而 APB2 预分频器的时钟来源是 ***AHB** 预分频器*。因此，把 APB2 预分频器设置为不分频，那么我们就可以得到 GPIO 外设的时钟也等于 HCLK，为 72MHz 了。

5.4.3 HCLK、FCLK、PCLK1、PCLK2

从时钟树的分析，看到经过一系列的倍频、分频后得到了几个与我们开发密切相关的时钟。

SYSCLK：系统时钟，STM32 大部分器件的时钟来源。主要由 AHB 预分频器分配到各个部件。

HCLK:由 AHB 预分频器直接输出得到，它是高速总线 AHB 的时钟信号，提供给存储器，DMA 及 cortex 内核，是 cortex 内核运行的时钟，***cpu** 主频*就是这个信号，它的大小与 STM32 运算速度，数据存取速度密切相关。

FCLK：同样由 AHB 预分频器输出得到，是内核的“自由运行时钟”。“自由”表现在它不来自时钟 HCLK，因此*在 **HCLK** 时钟停止时 **FCLK** 也继续运行*。它的存在，可以保证在处理器休眠时，也能够采样和到中断和跟踪休眠事件，它与 HCLK 互相同步。

PCLK1：外设时钟，由 ***APB1 预分频器***输出得到，最大频率为 ***36MHz***，提供给挂载在 APB1 总线上的外设。

PCLK2：外设时钟，由 ***APB2 预分频器***输出得到，最大频率可为 ***72MHz***，提供给挂载在 APB2 总线上的外设。

为什么 STM32 的时钟系统如此复杂，有倍频、分频及一系列的外设时钟的开关。需要倍频是考虑到*电磁兼容性*，如外部直接提供一个 72MHz 的晶振，太高的振荡频率可能会给制作电路板带来一定的难度。分频是因为 STM32 既有高速外设又有低速外设，各种外设的*工作频率不尽相同*，如同 pc 机上的南北桥，把高速的和低速的设备分开来管理。最后，每个外设都配备了外设时钟的开关，当我们不使用某个外设时，可以把这个外设时钟关闭，从而*降低 **STM32** 的整体功耗*。所以，当我们使用外设时，一定要记得开启外设的时钟啊，亲。

5.5 LED 具体代码分析

有了以上对 STM32 存储器映像，时钟系统，以及基本的库函数知识，我们就可以分析 LED 例程的代码了，不知现在你有没饱饱的感觉了，如果还饿，那继续。

5.5.1 实验描述及工程文件清单

实验描述	该实验讲解了如何运用 ST 的库来操作 I/O 口，使 I/O 口产生置位(1)和复位(0)信号，从而来控制 LED 的亮灭。
硬件连接	PC3 – LED1、PC4 – LED2、PC5 – LED3
用到的库文件	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c*
用户编写的文件	*USER/main.c*

	USER/stm32f10x_it.c *USER/led.c* *USER/led.h*

5.5.2 配置工程环境

LED 实验中用到了 GPIO 和 RCC(用于设置外设时钟)这两个片上外设，所以在操作 I/O 之前我们需要把关于这两个外设的库文件添加到工程模板之中。它们分别为 ***stm32f10x_gpio.c*** 和 ***stm32f10x_rcc.c*** 文件 。其中 ***stm32f10x_gpio.c*** 用于操作 I/O，而 ***stm32f10x_rcc.c*** 用于配置系统时钟和外设时钟，由于每个外设都要配置时钟，所以它是每个外设都需要用到的库文件。

在添加完这两个库文件之后立即编译的话会出错，因为每个外设库对应于一个 ***stm32f10x_xxx.c*** 文件的同时还对应着一个 ***stm32f10x_xxx.h*** 头文件，头文件包含了相应外设的 c 语言函数实现的声明，只有我们把相应的头文件也包含进工程才能够使用这些外设库。在库中有一个专门的文件 ***stm32f10x_conf.h*** 来管理所有库的头文件，***stm32f10x_conf.h*** 源码如下：

```
1.  * Includes ------------------------------------------------------------------
    */
2.  /* Uncomment the line below to enable peripheral header file inclusion */
3.  /* #include "stm32f10x_adc.h" */
4.  /* #include "stm32f10x_bkp.h" */
5.  /* #include "stm32f10x_can.h" */
6.  /* #include "stm32f10x_crc.h" */
7.  /* #include "stm32f10x_dac.h" */
8.  /* #include "stm32f10x_dbgmcu.h" */
9.  /* #include "stm32f10x_dma.h" */
10. /* #include "stm32f10x_exti.h" */
11. /* #include "stm32f10x_flash.h"*/
12. /* #include "stm32f10x_fsmc.h" */
```

```
13. /* #include "stm32f10x_gpio.h" */
14. /* #include "stm32f10x_i2c.h" */
15. /* #include "stm32f10x_iwdg.h" */
16. /* #include "stm32f10x_pwr.h" */
17. /* #include "stm32f10x_rcc.h" */
18. /* #include "stm32f10x_rtc.h" */
19. /* #include "stm32f10x_sdio.h" */
20. /* #include "stm32f10x_spi.h" */
21. /* #include "stm32f10x_tim.h" */
22. /* #include "stm32f10x_usart.h" */
23. /* #include "stm32f10x_wwdg.h" */
24. /*#include "misc.h"*/  /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

这是没有修改过的代码，默认情况下所有外设的头文件包含都被*注释*掉了。当我们需要用到某个外设驱动时直接把相应的注释去掉即可，非常方便。如本 LED 实验中我们用到了 ***RCC*** 跟 ***GPIO*** 这两个外设，所以我们应取消其注释，使第 13、17 行的代码***#include "stm32f10x_gpio.h"***、 ***#include "stm32f10x_rcc.h"*** 这两个语句生效，修改后如下所示：

```
1.  /* Includes ------------------------------------------------------------------
    */
2.  /* Uncomment the line below to enable peripheral header file inclusion */
3.  /* #include "stm32f10x_adc.h" */
4.  /* #include "stm32f10x_bkp.h" */
5.  /* #include "stm32f10x_can.h" */
6.  /* #include "stm32f10x_crc.h" */
7.  /* #include "stm32f10x_dac.h" */
8.  /* #include "stm32f10x_dbgmcu.h" */
9.  /* #include "stm32f10x_dma.h" */
10. /* #include "stm32f10x_exti.h" */
11. /* #include "stm32f10x_flash.h"*/
12. /* #include "stm32f10x_fsmc.h" */
13. #include "stm32f10x_gpio.h"
14. /* #include "stm32f10x_i2c.h" */
15. /* #include "stm32f10x_iwdg.h" */
16. /* #include "stm32f10x_pwr.h" */
```

```
17. #include "stm32f10x_rcc.h"
18. /* #include "stm32f10x_rtc.h" */
19. /* #include "stm32f10x_sdio.h" */
20. /* #include "stm32f10x_spi.h" */
21. /* #include "stm32f10x_tim.h" */
22. /* #include "stm32f10x_usart.h" */
23. /* #include "stm32f10x_wwdg.h" */
24. /*#include "misc.h"*/  /* High level functions for NVIC and SysTick (add-
    on to CMSIS functions) */
```

到这里，我们就可以用库自带的函数来操作 I/O 口了，这时我们可以编译一下，会发现既没有 Warning 也没有 Error。

5.5.3 编写用户文件

前期工程环境设置完毕，接下来我们就可以专心编写自己的应用程序了。我们把应用程序放在 USER 这个文件夹下，这个文件夹下至少包含了 ***main.c***、***stm32f10x_it.c***、***xxx.c*** 这三个源文件。其中 main 函数就位于 ***main.c*** 这个 c 文件中，main 函数只是用来测试我们的应用程序。***stm32f10x_it.c*** 为我们提供了 M3 所有中断函数的入口，默认情况下这些中断服务程序都为空，等到用到的时候需要用户自己编写。所以现在我们把 ***stm32f10x_it.c*** 包含到 USER 这个目录可以了。

而 ***xxx.c*** 就是由用户编写的文件，***xxx*** 是应用程序的名字，用户可自由命名。我们把应用程序的具体实现放在了这个文件之中，程序的实现和应用分开在不同的文件中，这样就实现了很好的封装性。本书的例程都严格遵从这个规则，每个外设的用户文件都由独立的源文件与头文件构成，这样可以更方便地实现代码重用了。

于是，我们在工程中新建两个文件，分别为 ***led.c*** 和 ***led.h***，保存在 USER 目录下，并把 led.c 添加到工程之中。led.c 文件中输入代码如下：

```
1. /********************* (C) COPYRIGHT 2012 WildFire Team *********
2.  * 文件名  : led.c
3.  * 描述    : led 应用函数库
4.  * 实验平台：野火 STM32 开发板
5.  * 硬件连接：-----------------
6.  *          |   PC3 - LED1     |
```

```
 7.   *           |    PC4 - LED2      |
 8.   *           |    PC5 - LED3      |
 9.   *            -------------------
10.  * 库版本   : ST3.5.0
11.  * 作者     : wildfire team
12.  * 论坛     : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
13.  * 淘宝     : http://firestm32.taobao.com
14. ******************************************************/
15. #include "led.h"
16.
17. /*
18.  * 函数名：LED_GPIO_Config
19.  * 描述   ：配置 LED 用到的 I/O 口
20.  * 输入   ：无
21.  * 输出   ：无
22.  */
23. void LED_GPIO_Config(void)
24. {
25.     /*定义一个 GPIO_InitTypeDef 类型的结构体*/
26.     GPIO_InitTypeDef GPIO_InitStructure;
27.
28.     /*开启 GPIOC 的外设时钟*/
29.     RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);
30.
31.     /*选择要控制的 GPIOC 引脚
    */
32.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5
    ;
33.
34.     /*设置引脚模式为通用推挽输出*/
35.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
36.
37.     /*设置引脚速率为 50MHz */
38.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
39.
40.     /*调用库函数，初始化 GPIOC*/
41.     GPIO_Init(GPIOC, &GPIO_InitStructure);
42.
43.     /* 关闭所有 led 灯 */
44.     GPIO_SetBits(GPIOC, GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5);
45. }
46.
47.
48. /********* (C) COPYRIGHT 2012 WildFire Team *****END OF FILE********/
```

在这个文件中，我们定义了一个函数 ***LED_GPIO_Config()***，在这个函数里，实现了所有为点亮 led 的配置。

5.5.4 初始化结构体——GPIO_InitTypeDef 类型

LED_GPIO_Config()函数中，在文件的第 26 行的代码：

GPIO_InitTypeDef GPIO_InitStructure; 这是利用库，定义了一个名为 ***GPIO_InitStructure*** 的结构体，结构体类型为 ***GPIO_InitTypeDef***。

GPIO_InitTypeDef类型与前面介绍的库对寄存器的封装类似，是库文件利用

关键字 ***typedef*** 定义的新类型。追踪其定义原型如下，位于 ***stm32f10x_gpio.h*** 文件中：

```
1.  typedef struct
2.  {
3.    uint16_t GPIO_Pin;              /*指定将要进行配置的 GPIO 引脚*/
4.    GPIOSpeed_TypeDef GPIO_Speed;   /*指定 GPIO 引脚可输出的最高频率*/
5.    GPIOMode_TypeDef GPIO_Mode;     /*指定 GPIO 引脚将要配置成的工作状态*/
6.  }GPIO_InitTypeDef;
```

于是我们知道，***GPIO_InitTypeDef*** 类型的结构体有三个成员，分别为 uint16_t 类型的 ***GPIO_Pin***，GPIOSpeed_TypeDef 类型的 ***GPIO_Speed*** 及 GPIOMode_TypeDef 类型的 ***GPIO_Mode***。

uint16_t 类型的 GPIO_Pin 为我们将要选择配置的引脚，在 stm32f10x_gpio.h 文件中有如下宏定义：

```
1.  #define GPIO_Pin_0          ((uint16_t)0x0001)  /*!< Pin 0 selected */
2.  #define GPIO_Pin_1          ((uint16_t)0x0002)  /*!< Pin 1 selected */
3.  #define GPIO_Pin_2          ((uint16_t)0x0004)  /*!< Pin 2 selected */
4.  #define GPIO_Pin_3          ((uint16_t)0x0008)  /*!< Pin 3 selected */
```

这些宏的值，就是允许我们给结构体成员 GPIO_Pin 赋的值，如我们给 GPIO_Pin 赋值为宏 GPIO_Pin_0，表示我们选择了 GPIO 端口的第 0 个引脚，在后面会通过一个函数把这些宏的值进行处理，设置相应的寄存器，实现我们对 GPIO 端口的配置。如 led.c 代码中的第 32 行，意义为我们将要选择 GPIO 的 Pin3、Pin4、Pin5 引脚进行配置。

GPIOSpeed_TypeDef 和 GPIOMode_TypeDef 又是两个库定义的新类型，GPIOSpeed_TypeDef 原型如下：

```
1.  typedef enum
2.  {
3.    GPIO_Speed_10MHz = 1, //枚举常量，值为 1，代表输出速率最高为 10MHz
4.    GPIO_Speed_2MHz,      //对不赋值的枚举变量，自动加 1，此常量值为 2
5.    GPIO_Speed_50MHz      //常量值为 3
6.  }GPIOSpeed_TypeDef;
```

这是一个枚举类型，定义了三个枚举常量，即 ***GPIO_Speed_10MHz=1，GPIO_Speed_2MHz=2，***

GPIO_Speed_50MHz=3。这些常量可用于标识 GPIO 引脚可以配置成的各个最高速度。所以我们在为结构体中的 *GPIO_Speed* 赋值的时候，就可以直接用这些含义清晰的*枚举标识符*了。如 led.c 代码中的第 38 行，给 GPIO_Speed 赋值为 3，意义为使其最高频率可达到 50MHz。

同样，GPIOMode_TypeDef 也是一个枚举类型定义符，原型如下：

```
1.  typedef enum
2.  { GPIO_Mode_AIN = 0x0,              //模拟输入模式
3.    GPIO_Mode_IN_FLOATING = 0x04,     //浮空输入模式
4.    GPIO_Mode_IPD = 0x28,             //下拉输入模式
5.    GPIO_Mode_IPU = 0x48,             //上拉输入模式
6.    GPIO_Mode_Out_OD = 0x14,          //开漏输出模式
7.    GPIO_Mode_Out_PP = 0x10,          //通用推挽输出模式
8.    GPIO_Mode_AF_OD = 0x1C,           //复用功能开漏输出
9.    GPIO_Mode_AF_PP = 0x18            //复用功能推挽输出
10. }GPIOMode_TypeDef;
```

这个枚举类型也定义了很多含义清晰的枚举常量，是用来帮助配置 GPIO 引脚的模式的，如 GPIO_Mode_AIN 意义为模拟输入、GPIO_Mode_IN_FLOATING 为浮空输入模式。在 led.c 代码中的第 35 行意义为把引脚设置为通用推挽输出模式。

于是，我们可以总结 GPIO_InitTypeDef 类型结构体的作用，整个结构体包含 *GPIO_Pin*、*GPIO_Speed*、*GPIO_Mode* 三个成员，我们对这三个成员赋予不同的数值可以对 GPIO 端口进行不同的配置，而这些可配置的数值，已经由 ST 的库文件封装成见名知义的枚举常量。这使我们编写代码变得非常简便。

5.5.5 初始化库函数——GPIO_Init()

在前面我们已经接触到 ST 的库文件，以及各种各样由 ST 库定义的新类型，但所有的这些，都只是为库函数服务的。在 led.c 文件的第 41 行，我们用到了第一个用于初始化的库函数 GPIO_Init()。

在我们应用库函数的时候，只需要知道它的功能及输入什么类型的参数，允许的参数值就足够了，这些我们都可以能通过查找库帮助文档获得，详

细方法见**错误！未找到引用源。**使用库帮助文档小节。查询结果见图 0-12。

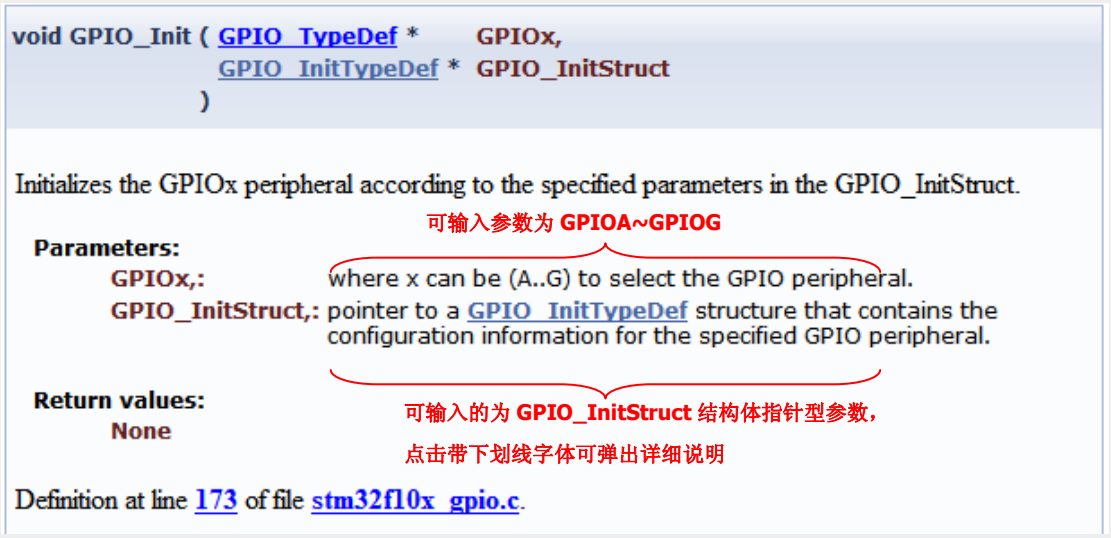


图 0-12 GPIO_Init 函数

这个函数有两个输入参数，分别为 ***GPIO_TypeDef*** 和 ***GPIO_InitTypeDef*** 型的指针。其允许值为 GPIOA……GPIOG，和 ***GPIO_InitTypeDef*** 型指针变量。

在调用的时候，如 led.c 文件的第 41 行，***GPIO_Init(GPIOC, &GPIO_InitStructure);***第一个参数，说明它将要对 GPIOC 端口进行初始化。初始化的配置以第二个参数 GPIO_InitStructure 结构体的成员值为准。这个结构体的成员，我们在调用 ***GPIO_Init()***前，已对它们赋予了控制参数。

```
31.     /*选择要控制的 GPIOC 引脚
    */
32.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5
    ;
33.
34.     /*设置引脚模式为通用推挽输出*/
35.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
36.
37.     /*设置引脚速率为 50MHz */
38.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
39.
```

于是，在调用 GPIO_Init()函数后，GPIOC 的 Pin3、Pin4、Pin5 就被配置成了最高频率为 50MHz 的通用推挽输出模式了。

在这个函数的内部，实现了把输入的这些参数按照一定的规则转化，进而写入寄存器，实现了配置 GPIO 端口的功能。函数的实现将在 0 小节进行详细分析。

5.5.6 开启外设时钟

调用了 GPIO_Init()函数之后，对 GPIO 的初始化也就基本完成了，那还缺少什么呢？就是在前面强调过的必须要开启外设时钟，在开启外设时钟之前，我们首先要配置好系统时钟 SYSCLK， 0 小节提到，为配置 SYSCLK，要设置一系列的时钟来源、倍频、分频等控制参数。这些工作由 SystemInit()库函数完成。

5.5.6.1 启动文件及 SystemInit()函数分析

在 ***startup_stm32f10x_hd.s*** 启动文件中，有如下一段启动代码：

```
1.  ;Reset_Handler 子程序开始
2.  Reset_Handler    PROC
3.
4.   ;输出子程序 Reset_Handler 到外部文件
5.                   EXPORT  Reset_Handler             [WEAK]
6.
7.  ;从外部文件中引入 main 函数
8.                   IMPORT   __main
9.
10. ;从外部文件引入 SystemInit 函数
11.                 IMPORT  SystemInit
12.
13. ;把 SystemInit 函数调用地址加载到通用寄存器 r0
14.                  LDR      R0, =SystemInit
15.
16. ;跳转到 r0 中保存的地址执行程序（调用 SystemInit 函数）
17.                  BLX      R0
18.
19. ;把 main 函数调用地址加载到通用寄存器 r0
20.                  LDR      R0, =__main
21.
22. ;跳转到 r0 中保存的地址执行程序（调用 main 函数）
23.                  BX       R0
24.
25. ;Reset_Handler 子程序结束
26.                  ENDP
```

注：这是一段汇编代码，对汇编比较陌生的读者请配以 ” ; ” 后面的注释来阅读，”; ”表示注释其后的单行代码，相当于 c 语言中的” // ” 和 ” /* */ ”。

当芯片被复位(包括上电复位)的时候，将开始运行这一段代码，运行过程为先调用了 ***SystemInit()***函数，再进入 c 语言中的 ***main*** 函数执行。读者是否曾思考过？为什么 c 语言程序都从 main 函数开始执行？就是因为我们的启动文件中有了这一段代码，可以尝试一下把第 8 行引入 main 函数，及第 20 行的加载 main 函数的标识符修改掉，看其效果。如改成：

```
IMPORT   __wildfire
.....
LDR    R0 ,=__wildfire
```

这样修改以后，内核就会从 wildfire()函数中开始执行第一个 c 语言的代码啦。有些比较狡猾的朋友就会这么干，让人家看他的代码时找不到 main 函数，何其险恶呀:)。

但是，前面强调了，进入 main 函数之前调用了一个名为 ***SystemInit()*** 的函数。这个函数的定义在 ***system_stm32f10x.c*** 文件之中。它的作用是设置系统时钟 ***SYSCLK***。函数的执行流程是先将与配置时钟相关的寄存器都复位为默认值，复位寄存器后，调用了另外一个函数 ***SetSysClock()***，***SetSysClock()***代码如下：

```
1.  static void SetSysClock(void)
2.  {
3.  #ifdef SYSCLK_FREQ_HSE
4.    SetSysClockToHSE();
5.  #elif defined SYSCLK_FREQ_24MHz
6.    SetSysClockTo24();
7.  #elif defined SYSCLK_FREQ_36MHz
8.    SetSysClockTo36();
9.  #elif defined SYSCLK_FREQ_48MHz
10.   SetSysClockTo48();
11. #elif defined SYSCLK_FREQ_56MHz
12.   SetSysClockTo56();
13. #elif defined SYSCLK_FREQ_72MHz
14.   SetSysClockTo72();
15. #endif
16.
17.  /* If none of the define above is enabled, the HSI is used as System
    clock
18.     source (default after reset) */
19. }
```

从 ***SetSysClock()***代码可以知道，它是根据我们设置的条件编译宏来进行不同的时钟配置的。

在 *system_stm32f10x.c* 文件的开头，已经默认有了如下的条件编译定义：

```
1. #if defined (STM32F10X_LD_VL) || (defined STM32F10X_MD_VL) || (defined
    STM32F10X_HD_VL)
2. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
3.  #define SYSCLK_FREQ_24MHz  24000000
4. #else
5. /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
6. /* #define SYSCLK_FREQ_24MHz  24000000 */
7. /* #define SYSCLK_FREQ_36MHz  36000000 */
8. /* #define SYSCLK_FREQ_48MHz  48000000 */
9. /* #define SYSCLK_FREQ_56MHz  56000000 */
10.#define SYSCLK_FREQ_72MHz  72000000
11.#endif
```

在第 10 行定义了 *SYSCLK_FREQ_72MHz* 条件编译的标识符，所以在 *SetSysClock()*函数中将调用 *SetSysClockTo72()*函数把芯片的系统时钟 *SYSCLK* 设置为 *72MHz* 当然，前提是输入的外部时钟源 HSE 的振荡频率要为 *8MHz*。

其中的 *SetSysClockTo72()* 函数就是最底层的库函数了，那些跟寄存器打交道的活都是由它来完成的，如果大家想知道我们的系统时钟是如何配置成 72M 的话，可以研究这个函数的源码。但大可不必这样，我们应该抛开传统的直接跟寄存器打交道来学单片机的方法，而是直接用 ST 的库给我们提供的上层接口，这样会简化我们很多的工作，还能提高我们开发产品的效率，何乐而不为呢？对这一类直接跟寄存器打交道的函数分析在 0 小节以 GPIO_Init()函数为例来分析。

注意：3.5 版本的库在启动文件中调用了 SystemInit()，所以不必在 main()函数中再次调用。但如果使用的是 3.0 版本的库则必须在 main 函数中调用 SystemInit()，以设置系统时钟，因为在 3.0 版本的启动代码中并没有调用 SystemInit()函数。

5.5.6.2 开启外设时钟

SYSCLK 由 SystemInit()配置好了，而 GPIO 所用的时钟 *PCLK2* 我们采用默认值，也为 72MHz。我们采用默认值可以不修改分频器，但外设时钟默认是处在关闭状态的。所以外设时钟一般会在初始化外设的时候设置为开启(根据设计的产品功耗要求，也可以在使用的时候才打开) 。开启和关闭外设时钟也有

封装好的库函数 ***RCC_APB2PeriphClockCmd()***。在 led.c 文件中的第 29 行，我们调用了这个函数。

查看其使用手册见图 0-13

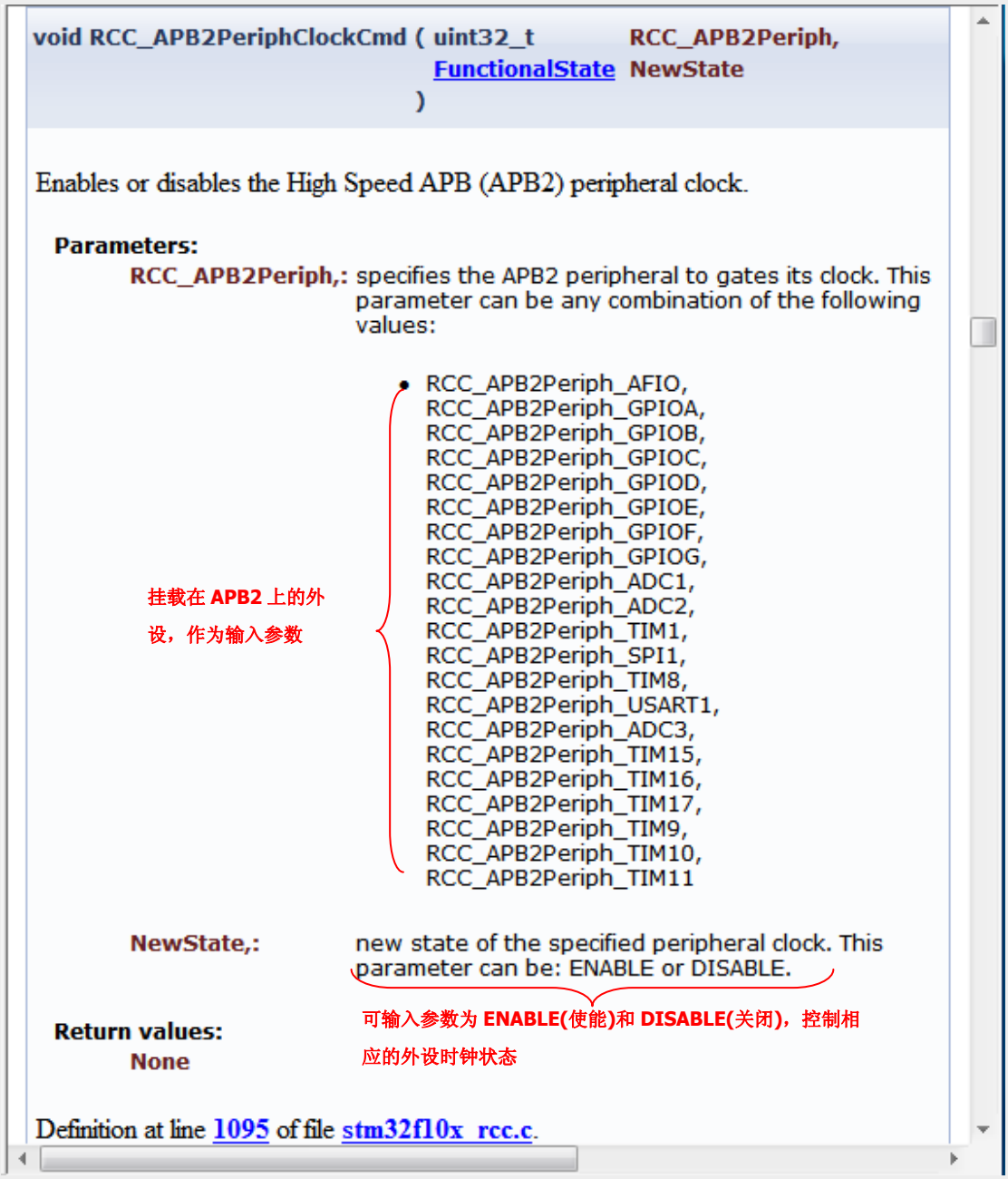


图 0-13 APB2 时钟使能函数

调用的时候需要向它输入两个参数，一个参数为将要控制的，挂载在 APB2 总线上的外设时钟，第二个参数为选择要开启还是关闭该时钟。

led.c 文件中对它的调用：

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

就表示将要 ENABLE(使能)GPIOC 外设时钟。

在这里强调一点，如果我们用到了 I/O 的引脚*复用功能*，还要开启其*复用功能时钟*。

如 GPIOC 的 Pin4 还可以作为 ADC1 的输入引脚，现在我们把它作为 ADC1 来使用，除了开启 GPIOC 时钟外，还要开启 ADC1 的时钟：

```
RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOC, ENABLE);
RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC1, ENABLE);
```

我们知道有的外设是挂载在高速外设总线 APB2 上使用 PCLK2 时钟，还有的是挂载在低速外设总线 APB1 上，使用 PCLK1 时钟。既然时钟源是不同的，当然也就有另一个函数来开启 APB1 总线外设的时钟：

*RCC_APB1PeriphClockCmd()*函数，这两个函数名，正是根据其挂载在的总线命名的。可输入的参数自然也就不一样，使用的时候要注意区分。其中所有的 GPIO 都是挂载在 APB2 上的。

5.5.7 控制 I/O 输出高、低电平

前面我们选择好了引脚，配置了其功能及开启了相应的时钟，我们可以终于可以正式控制 I/O 口的电平高低了，从而实现控制 LED 灯的亮与灭。

前面提到过，要控制 GPIO 引脚的电平高低，只要在 GPIOx_BSRR 寄存器相应的位写入控制参数就可以了。ST 库也为我们提供了具有这样功能的函数，可以分别是用 *GPIO_SetBits()*控制输出高电平，和用 *GPIO_ResetBits()*控制输出低电平。见图 0-14 及图 0-15

```
void GPIO_SetBits ( GPIO_TypeDef * GPIOx,
                    uint16_t       GPIO_Pin
                  )
```

Sets the selected data port bits.

Parameters:

GPIOx,: where x can be (A..G) to select the GPIO peripheral.

GPIO_Pin,: specifies the port bits to be written. This parameter can be any combination of GPIO_Pin_x where x can be (0..15).

Return values:

None

图 0-14 GPIO 引脚置 1 函数

```
void GPIO_ResetBits ( GPIO_TypeDef * GPIOx,
                      uint16_t       GPIO_Pin
                    )
```

Clears the selected data port bits.

Parameters:

GPIOx,: where x can be (A..G) to select the GPIO peripheral.

GPIO_Pin,: specifies the port bits to be written. This parameter can be any combination of GPIO_Pin_x where x can be (0..15).

Return values:

None

图 0-15 GPIO 引脚清零函数

输入参数有两个，第一个为将要控制的 GPIO 端口：GPIOA……GPIOG，第二个为要控制的引脚号：Pin0~Pin15。

在 led.c 文件的第 44 行，***LED_GPIO_Config()***函数中，我们在调用 GPIO_Init()函数之后就调用了 ***GPIO_SetBits()***函数，从而让这几个引脚输出高电平，使三盏 LED 初始化后都处于灭状态。

5.5.8 led.h 文件

接下来，分析 led.h 文件。其内容如下

```
1.  #ifndef __LED_H
2.  #define __LED_H
3.
4.  #include "stm32f10x.h"
5.
6.  /* the macro definition to trigger the led on or off
7.   * 1 - off
8.   - 0 - on
9.   */
10. #define ON  0
11. #define OFF 1
12.
13. //带参宏，可以像内联函数一样使用
14. #define LED1(a) if (a)   \
15.                     GPIO_SetBits(GPIOC,GPIO_Pin_3);\
16.                     else        \
17.                     GPIO_ResetBits(GPIOC,GPIO_Pin_3)
18.
19. #define LED2(a) if (a)   \
20.                     GPIO_SetBits(GPIOC,GPIO_Pin_4);\
21.                     else        \
22.                     GPIO_ResetBits(GPIOC,GPIO_Pin_4)
23.
24. #define LED3(a) if (a)   \
25.                     GPIO_SetBits(GPIOC,GPIO_Pin_5);\
26.                     else        \
27.                     GPIO_ResetBits(GPIOC,GPIO_Pin_5)
28.
29. void LED_GPIO_Config(void);
30.
31. #endif /* __LED_H */
```

这个头文件的内容不多，但也把它独立成一个头文件，方便以后扩展或移植使用。希望读者养成良好的工程习惯，在写头文件的时候，加上类似以下这样的条件编译。

```
#ifndef __LED_H
#define __LED_H
.....
#endif
```

这样可以防止头文件重复包含，使得工程的兼容性更好。读者问为什么要加两个下划线"__"？在这里加两个下划线可以避免这个宏标识符与其它定义重名，因为在其它部分代码定义的宏或变量，一般都不会出现这样有下划线的名字。

在 led.h 头文件的部分，首先包含了前面提到的最重要的 ST 库必备头文件 ***stm32f10x.h***。有了它我们才可以使用各种库定义、库函数。

在 led.h 文件的第 14~27 行，是我们利用 ***GPIO_SetBits()***、***GPIO_ResetBits()*** 库函数编写的带参宏定义，带参宏与 C++中的内联函数作

用很类似。在编译过程，编译器会把带参宏展开，在相应的位置*替换*为宏展开代码。其中的反斜杠符号“ \”叫做续行符，用来连接上下行代码，表示下面一行代码属于“\”所在的代码行，这在 ST 库经常出现。“\”的语法要求极其严格，在它的后面不能有空格、注释等一切“杂物”，在论坛上经常有读者反映遇到编译错误，却不知道正是错在这里。群里很多朋友都问到“ \”是个什么东西，那野火可要打你 pp 了，你这是 c 语言不及格呀，亲。

最后，在 led.h 文件中的第 29 行代码，*声明*了我们在 led.c 源文件定义的 LED_GPIO_Config()用户函数。因此，我们要使用 led.c 文件定义的函数时，只要把 led.h 包含到调用到函数的文件中就可以了。

5.5.9 main 文件

写好了 led.c、led.h 两个文件，我们控制 LED 灯的驱动程序就全部完成了。接下来，就可以利用写好的驱动文件，在 main 文件中编写应用程序代码了。本 LED 例程的 main 文件内容如下：

```
1.  /******* (C) COPYRIGHT 2012 WildFire Team **************************
2.   * 文件名  : main.c
3.   * 描述    : LED 流水灯，频率可调……
4.   * 实验平台 : 野火 STM32 开发板
5.   * 库版本   : ST3.5.0
6.   *
7.   * 作者    : wildfire team
8.   * 论坛    : www.ourdev.cn/bbs/bbs_list.jsp?bbs_id=1008
9.   * 淘宝    : http://firestm32.taobao.com
10. ******************************************************************/
11. #include "stm32f10x.h"
12. #include "led.h"
13.
14. void Delay(__IO u32 nCount);
15.
16. /*
17.  * 函数名: main
18.  * 描述  : 主函数
19.  * 输入  : 无
20.  * 输出  : 无
21.  */
22. int main(void)
23. {
24.     /* LED 端口初始化 */
25.     LED_GPIO_Config();
26.
27.     while (1)
28.     {
29.         LED1( ON );            // 亮
30.         Delay(0x0FFFEF);
31.         LED1( OFF );           // 灭
```

```
32.
33.         LED2( ON );
34.         Delay(0x0FFFEF);
35.         LED2( OFF );
36.
37.         LED3( ON );
38.         Delay(0x0FFFEF);
39.         LED3( OFF );
40.     }
41. }
42.
43. void Delay(__IO u32 nCount)  //简单的延时函数
44. {
45.     for(; nCount != 0; nCount--);
46. }
47.
48.
49. /******* (C) COPYRIGHT 2012 WildFire Team *****END OF FILE********/
```

main 文件的开头部分首先包含所需的头文件，***stm32f10x.h*** 和 ***led.h***。

在第 14 行还声明了一个简单的延时函数，其定义在 main 文件的末尾。它是利用 for 循环实现的，用作短暂的，对精度要求不高的延时，延时的时间与输入的参数并无准确的计算公式，请不要深究。需要精准的延时的时候，我们会采用定时器来精确控制。

在芯片上电(复位)后，经过启动文件中 ***SystemInit()***函数配置好了时钟，就进入 main 函数了。接下来，从 main 函数开始分析代码的执行。

首先，调用了在 led.c 文件编写好的 ***LED_GPIO_Config()***函数，完成了对 GPIOC 的 Pin3、Pin4、Pin5 的初始化。紧接着就在 while 死循环里不断执行在 led.h 文件中编写的带参宏代码，并加上延时函数，使各盏 LED 轮流亮灭。当然，在 LED 控制的部分，如果不习惯带参宏的方式，读者也可以直接使用 ***GPIO_SetBits()***和 ***GPIO_ResetBits()***函数实现对 LED 的控制。

如果使用的是 ***3.0 版本*** 的库，由于启动文件中没有调用 ***SystemInit()*** 函数，所以要在初始化 GPIO 等外设之前，也就是在 main 函数的第 1 行代码，就调用 ***SystemInit()***函数，以完成对系统时钟的配置。

到此，我们整个控制 LED 灯的工程的讲解就完成了。

5.5.10 实验现象

将程序烧写到野火 STM32 开发板中，即可看到 3 个 LED 一定的频率闪烁。

5.6 GPIO_Init()函数的实现

在我们控制 LED 灯的工程中，调用了很多库函数，有 ***SystemInit()***、***GPIO_Init()***、***GPIO_SetBits()***、***GPIO_ResetBits()***等等。虽说为了开发速度，我们只管函数的功能和如何调用就行了，但免不了有种不踏实的感觉。

所以在本小节以 GPIO_Init()函数实现的分析为例，可以帮助读者理解 ST 库的本质，让读者在使用库开发的时候心里更有底。

5.6.1 规范的位操作方法

由于库函数的实现涉及到不少位操作，首先为读者介绍一下几个常用的位操作方法，排除阅读代码的障碍。

1、 将 char 型变量 a 的第七位(bit6)*清 0*，*其它位不变*。

```
1、  a &= ~(1<<6);    //括号内 1 左移 6 位，得二进制数：0100 0000
2、                   //按位取反，得 1011 1111 ，所得的数与 a 作”位与&”运算，
3、                   // a 的第 7 位（bit6）被置零，而其它位不变。
```

2、 同理，将变量 a 的第七位(bit6)*置 1*，其它位不变的方法如下。

```
1、  a |= (1<<6);          //把第七位（bit6）置 1，其它为不变
```

3、 将变量 a 的第七位(bit6)*取反*，其它位不变。

```
1、  a ^=(1<<6);    //把第七位（bit6）取反，其它位不变
```

5.6.2 GPIO_Init()实现代码分析

有了上面的位操作知识准备后，就可以分析 GPIO_Init()函数的定义代码了。

```
1. void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
2. {
3.   uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x
   00;
4.   uint32_t tmpreg = 0x00, pinmask = 0x00;
5.   /* 断言，用于检查输入的参数是否正确 */
6.   assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
7.   assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
8.   assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
```

```
9.
10. /*---------------------------- GPIO 的模式配置 -----------------------*/
11. /*把输入参数 GPIO_Mode 的低四位暂存在 currentmode*/
12.   currentmode = ((uint32_t)GPIO_InitStruct-
    >GPIO_Mode) & ((uint32_t)0x0F);
13. /*判断是否为输出模式，输出模式，可输入参数中输出模式的 bit4 位都是 1*/
14.   if ((((uint32_t)GPIO_InitStruct-
    >GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
15.   {
16.     /* 检查输入参数 */
17.     assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
18.     /* 输出模式，所以要配置 GPIO 的速率:00(输入模式) 01(10MHz) 10(2MHz) 11 */
19.     currentmode |= (uint32_t)GPIO_InitStruct->GPIO_Speed;
20.   }
21. /*----------------------------配置 GPIO 的 CRL 寄存器 -----------------------
    -*/
22.   /* 判断要配置的是否为 pin0 ~~ pin7 */
23.   if (((uint32_t)GPIO_InitStruct-
    >GPIO_Pin & ((uint32_t)0x00FF)) != 0x00)
24.   {
25. /*备份原 CRL 寄存器的值*/
26. tmpreg = GPIOx->CRL;
27. /*循环，一个循环设置一个寄存器位*/
28.     for (pinpos = 0x00; pinpos < 0x08; pinpos++)
29. {
30. /*pos 的值为 1 左移 pinpos 位*/
31.       pos = ((uint32_t)0x01) << pinpos;
32.       /* 令 pos 与输入参数 GPIO_PIN 作位与运算，为下面的判断作准备 */
33.       currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
34. /*判断，若 currentpin=pos,说明 GPIO_PIN 参数中含的第 pos 个引脚需要配置*/
35.       if (currentpin == pos)
36.       {
37. /*pos 的值左移两位（乘以 4），因为寄存器中 4 个寄存器位配置一个引脚*/
38.         pos = pinpos << 2;
39.  /*以下两个句子，把控制这个引脚的 4 个寄存器位清零，其它寄存器位不变*/
40.         pinmask = ((uint32_t)0x0F) << pos;
41.         tmpreg &= ~pinmask;
42.         /* 向寄存器写入将要配置的引脚的模式 */
43.         tmpreg |= (currentmode << pos);
44.         /* 复位 GPIO 引脚的输入输出默认值*/
45. /*判断是否为下拉输入模式*/
46.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
47.         {
48. /*下拉输入模式，引脚默认置 0，对 BRR 寄存器写 1 可对引脚置 0*/
49.           GPIOx->BRR = (((uint32_t)0x01) << pinpos);
50.         }
51.         else
52.         {
53.           /*判断是否为上拉输入模式*/
54.           if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
55.           {
56. /*上拉输入模式，引脚默认值为 1，对 BSRR 寄存器写 1 可对引脚置 1*/
57.             GPIOx->BSRR = (((uint32_t)0x01) << pinpos);
58.           }
59.         }
60.       }
61. }
62. /*把前面处理后的暂存值写入到 CRL 寄存器之中*/
63.     GPIOx->CRL = tmpreg;
64.   }
65. /*---------------------------- 以下部分是对 CRH 寄存器配置的 -----------------

66. --------当要配置的引脚为 pin8 ~~ pin15 的时候，配置 CRH 寄存器，-----
```

```
67. --------------    -----这过程和配置 CRL 寄存器类似-------------------------------
    ------
68. --------读者可自行分析，看看自己是否了解了上述过程--^_^-----------*/
69.   /* Configure the eight high port pins */
70.   if (GPIO_InitStruct->GPIO_Pin > 0x00FF)
71.   {
72.     tmpreg = GPIOx->CRH;
73.     for (pinpos = 0x00; pinpos < 0x08; pinpos++)
74.     {
75.       pos = (((uint32_t)0x01) << (pinpos + 0x08));
76.       /* Get the port pins position */
77.       currentpin = ((GPIO_InitStruct->GPIO_Pin) & pos);
78.       if (currentpin == pos)
79.       {
80.         pos = pinpos << 2;
81.         /* Clear the corresponding high control register bits */
82.         pinmask = ((uint32_t)0x0F) << pos;
83.         tmpreg &= ~pinmask;
84.         /* Write the mode configuration in the corresponding bits */
85.         tmpreg |= (currentmode << pos);
86.         /* Reset the corresponding ODR bit */
87.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
88.         {
89.           GPIOx->BRR = (((uint32_t)0x01) << (pinpos + 0x08));
90.         }
91.         /* Set the corresponding ODR bit */
92.         if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
93.         {
94.           GPIOx->BSRR = (((uint32_t)0x01) << (pinpos + 0x08));
95.         }
96.       }
97.     }
98.     GPIOx->CRH = tmpreg;
99.   }
100.   }
```

这部分代码比较长，请读者配合代码中的注释，《STM32 中文参考手册》中的 CRL 寄存器的说明图 0-16，及**错误！未找到引用源。**来理解这个函数。

8.2.1 端口配置低寄存器(GPIOx_CRL) (x=A..E)

偏移地址：0x00

复位值：0x4444 4444

每 4 个寄存器位配置一个引脚　　这 4 位控制 pin4

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

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度50MHz

CNFy:向这两个位写入不同的值，设置引脚为不同的功能。y 表示第 y 个引脚。

MODEy:向这两个位写入不同的值，设置引脚为不同的最大输出速率，或设置为输入模式。

图 0-16 GPIOx_CRL 寄存器

图 0-17 GPIO_Init 分析

2、 第 35 行，对*.GPIO_Mode* 赋值为 *GPIO_Mode_Out_PP*，宏展开为*（0001 0100）B*，表明我们要把这三个引脚都设置为通用推挽模式。

3、 第 38 行，对*.GPIO_Speed* 赋值为 *GPIO_Speed_50MHz*，宏展开为*（0011）B*，表明我们设置这三个引脚的输出最大速度都为 50MHz。

led.c 的第 41 行调用 GPIO_Init()的时候，就把 GPIOC 和上面这三个参数输入到函数了，经过这个函数处理，最终它向 GPIOC 组的 CRL 配置寄存器写入了一个值：

```
1.  GPIOC->CRL = 0x44333444;
2.   //二进制表示为(0100 0100 0011 0011 0011 0100 0100 0100)
```

把这个值化为二进制为：(0100 0100 0011 0011 0011 0100 0100 0100)B；这个值的每 4 个二进制位代表一组引脚的控制值。Pin3、Pin4、Pin5 的控制值都是(0011) B，有心的读者可以对比一下 *CRL* 寄存器的说明，这些控制值正好可以把 GPIO 设置为符合我们输入参数要求的状态，为最大速率为 50MHz 的通用推挽输出模式。

5.6.3 再论开发方式

了解库函数的实现后，我们现在就可以用实例来分析使用库函数与直接配置寄存器的区别了。

用直接配置寄存器的方法，只需要一个语句：

```
1.  GPIOC->CRL = 0x44333444;
```

这样直接向寄存器赋值就完成了，以这样的方式配置是内核执行效率最高的方式，内核的工作是简单了，但我们为实现所需的配置，确定这样的一个值，却是一件麻烦事，工程量大的时候，缺点就显而易见了。

配置寄存器还可以用一些相对缓和的方法，前面提到的三种*位操作方式*。如：

```
GPIOC->CRL &=~(uint32_t)(1111<<4*3); //清空 Pin3 的 4 个控制位
GPIOC->CRL |=(uint32_t)(0011<<4*3);  //配置 Pin3 的 4 个控制位
GPIOC->CRL &=~(uint32_t)(1111<<4*4); //清空 Pin4 的 4 个控制位
GPIOC->CRL |=(uint32_t)(0011<<4*4);  //配置 Pin4 的 4 个控制位
GPIOC->CRL &=~(uint32_t)(1111<<4*5); //清空 Pin5 的 4 个控制位
GPIOC->CRL |=(uint32_t)(0011<<4*5);  //配置 Pin5 的 4 个控制位
```

这个方法也可以实现我们所需的配置，而且修改起来比较容易，但执行的效率就比第一个方法要低了。

最后就是我们的调用库函数的方法，从内核的执行效率上看，首先库函数在被*调用*的时候要耗费*调用时间*；在函数内部，把输入参数*转换*为可以直接写入到寄存器的值也耗费了一些*运算时间*。而其它的宏、枚举等*解释*操作是作*编译过程完成*的，这部分并不消耗内核的时间。而优点呢？则是我们可以快速上手 STM32 控制器；配置外设状态时，不需要再纠结要向寄存器写入什么数值；交流方便，查错简单。这就是我们选择库的原因。

现在的处理器的主频是越来越高，我们需不需要担心 cpu 耗费那么多时间来干活会不会被累倒，野火要告诉你的是，不需要，还是担心下自己字字查询 datasheet 会不会被累倒吧。

至此，我们就把 GPIO_Init()库函数的实现分析完毕了。分析它纯粹是为了满足自己的求知欲，学习其编程的方式、思想，这对提高我们的编程水平是很有好处的，顺便感受一下 ST 库设计的严谨性，野火认为这样的代码不仅严谨且华丽优美，不知读者你是否也有这样的感受。就像野火在论坛里面说过：要我操作寄存器，我宁愿回家种田。

我们在以后开发的工程中，一般不会去分析 ST 的库函数的实现了。因为这些库函数是很类似的，都是把原来封装好的宏或枚举标识符转化成相应的值，写入到寄存器之中。这些都是十分*枯燥和机械*的工作，既然我们已经知道它的原理，又有现成的函数可供调用，就没必要再去探究了。

到了这里流水灯这个例程就算讲完了，如果你搞明白了流水灯编程的来龙去脉，那么后面的 M3 的学习路程将会简单而有趣。后面的例程也不再会像这个例程那么详细，所以大家要重点把握《4、初始 STM32 库》和《5、流水灯的前后今生》，把库的编程思想了然于胸。