正在准备做毕业设计，配置LED_Config()的时候，又看到了位带操作的宏定义，我又嘀咕了，什么是位带操作，一年前在使用位带操作的时候，就查阅过好多资料，Core-M3也看过，但是对于博主这种“低能儿”来说，你不把它说的白一点，就是感觉理解的不够透彻，于是今天又一次，查阅了各种手册，也算是基本弄懂了，鉴于博主的个人特点，所以本人的介绍也会十分浅显易懂，希望能帮到各位！

首先，抛砖引玉，来两个问题：

1）为什么STM32里面会有位带操作？

2）STM32里面的位带操作是什么意思？

我也不想去弄什么官方定义了，来两个例子，相信各位心里即使不能给出一个确切的定义，也不会再去纠结这个问题，

答：

1）51单片机相信各位都用过，假设P1.1的IO口上挂了一个LED，那么你单独对LED的操作就是P1.1 = 0或P1.1 = 1，注意，是你可以单独的对P1端的第一个IO口进行操作，然而STM32是不允许这样做的，那么为了像51单片机一样能够单独的对某个端的某一个IO单独操作，就引入了位带操作这样的概念，简而言之，言而总之，就是为了去单独操作32里面PA端的第1个IO口，所以才有了位带这样的操作机制。

2）打个形象的比方，以某个村，就张村把，该村有3户人家分别为A,B,C，我想给张村的A送礼，但是明文规定，不能给具体的个人送礼，但是可以给村委会送礼，那我该怎么办呢，OK，即日起，A不叫A了，改名叫做村委会1，B和C分别改叫做村委会2和村委会3，哦了，可以给A送礼了，虽然我送礼的对象是村委会1，听起来好像比个人级别高一点，但是最终收到礼物的还是个人A。同理，STM32不允许对某个端的某一个IO口进行操作，也就是PA.1 = 0或者PA.1 = 1这样的操作是非法的，好了，那我就给PA.1起个别名，将原来PA.1的地址扩展成一个32位的字地址，对32位的地址进行操作，这个是STM32允许的，必需可以的,STM32对所有的寄存器配置，都是对某个32位地址的操作，因此说白了，就是某个IO端口进行操作，这就是位带操作。

大白话说完，还是得回归官方介绍，不过这时候你在看，应该会好很多了。我们一步一步来，首先你应该知道的
位带区，和位带别名区，位带区，就是就是你想单独操作的IO的区域，也就是PA,PB……等这一堆IO口的内存所在区，而位带别名区，就是你给每一位重新起了个名字的那一片地址区域。可以看下表，M3内核存储器映射表，你能看到1M内存的BitBand区，还有与之对应的32M内存的BitBand别名区，因为你将每一位膨胀成为了一个32位的地址，所以相应的别名区的内存也会是位带区的32倍。

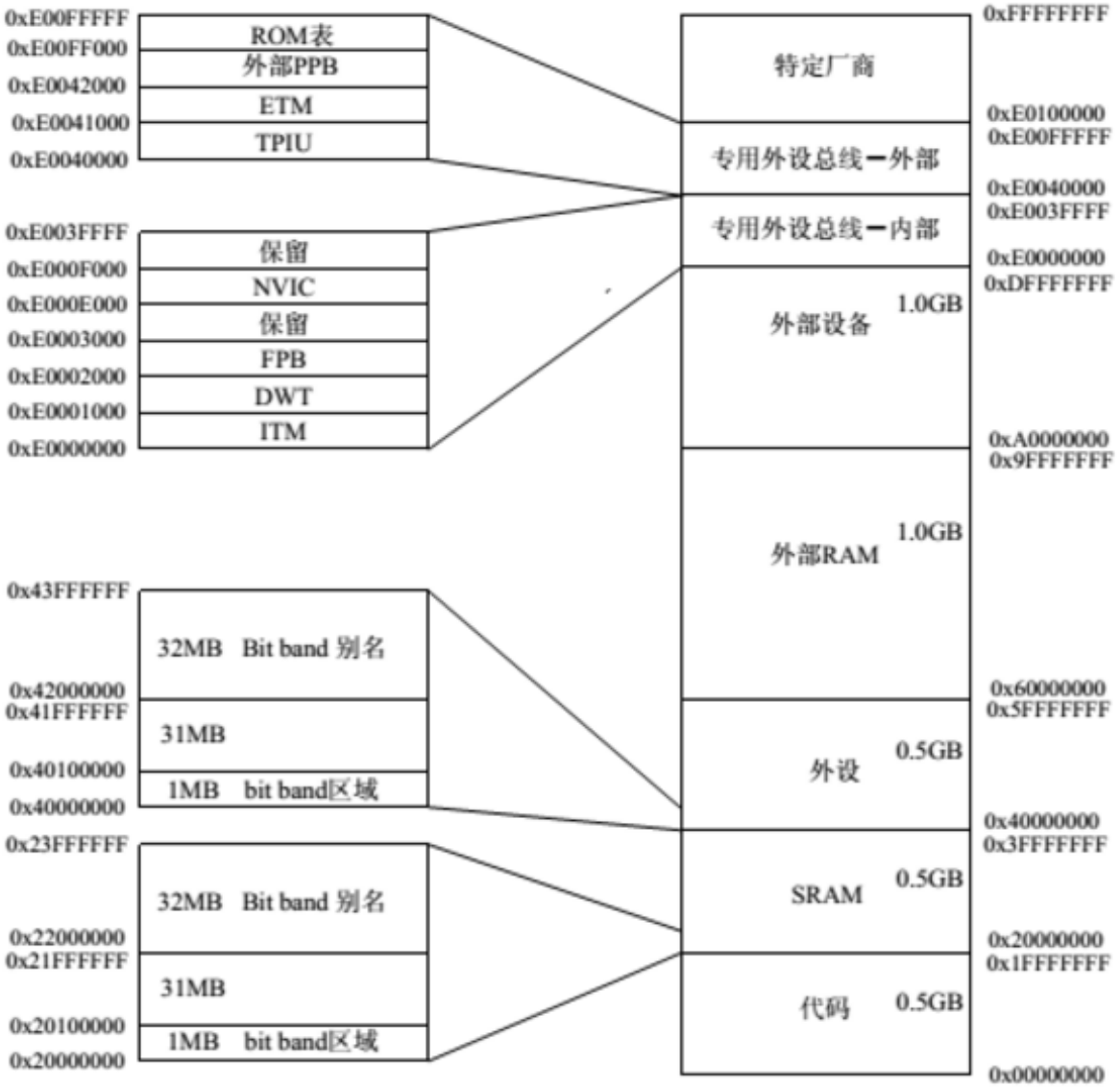


图 4-1 Cortex-M3 存储器映射

OK，现在我们应该能够知道，你想进行位带操作去操作某个IO口的某一位，那么在STM32的环境下，你应该去找该位对应的别名区的地址，

找到了这个地址，对这个地址进行操作，那么实际上也就是对该位进行操作了，接下来，我们要去找位所对应的地址了。

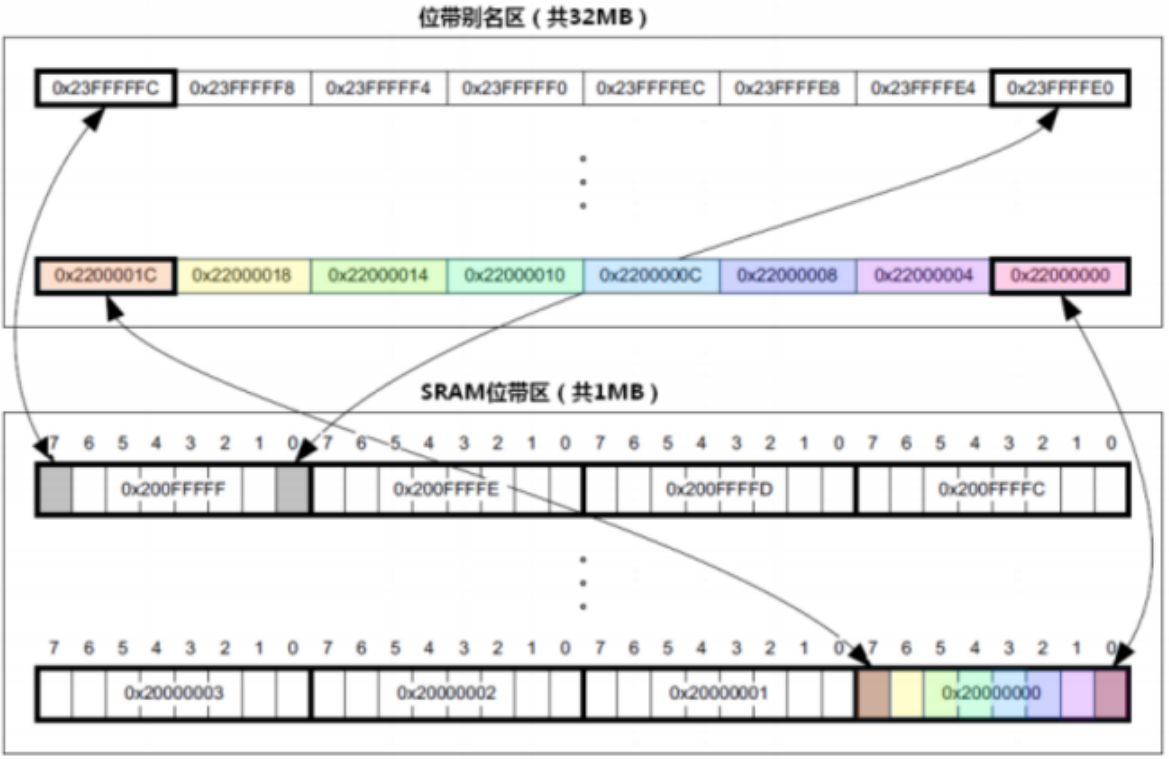
官方给出了相应的计算公式，我们以外设部分为例，毕竟用的多的还是外设部分的端口，具体到PA.1把

AliasAddr＝ 0x42000000+((A－0x40000000)*8+n)*4

=0x42000000+ (A－0x40000000)*32 + n*4

AliasAddr是别名区的地址，A是GPIOA->ODR的地址，n是该端口的上的某一位，这里就是1，通过这个公式你可以找到对应的别名区的地址，接下来就是对这个地址进行操作了，你给他写1，该位输出1，写0，就输出0。

在这里我想解释以下，为什么这个公式是这个样子的，因为我也思考了很久！借助于下面这个图：



0x42000000是位带别名区域的起始地址，A是输出数据寄存器GPIOA->ODR的地址，A的地址先减去位带区基地址，得到的是相对于位带区基地址的偏移地址，那么膨胀之后还是一个偏移地址，是相对于位带别名区基地址的偏移量，加上位带别名区域基地址，就得到了其对应的别名区地址，这是总的原理，

((A - 0x40000000)*8+n)*4 =0x42000000+ (A - 0x40000000)*32 + n*4

这部分是膨胀公式,乘8是先把单元内的每一位上升到字节的高度上，这样，你想设置第二位，就直接在原来的基地址上+2就可以了，确定完是第几位，再乘4，就是把位再上升到字的高度上，也就是每一位对应一个32位的字，这样最终的地址转换就完成，关键还是要注意两点，一是，两部分地址的互相转换，主要是每一部分的基地址。二就是位上升的32位地址这样的一个方法概念。

说到这里，基本已经介绍了80% 了，多数情况下，大家见到的代码，应该是以下这个样子，一共分为三步，

```
1#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr &0xFFFFF)<<5)+(bitnum<<2))       2 #define MEM_ADDR(addr)  *((volatile unsigned long  *)(addr))                                                                                3
#define BIT_ADDR(addr, bitnum)   MEM_ADDR(BITBAND(addr, bitnum))
```

第一步，就是我们上面分析的，得到位带别名区域的32位地址，至于第二步嘛，其实就是一个转换，给各位举个例子，如下，我想直接访问0x00000001这个地址，并且给这个地址写1，该怎么做呢，

```
1 # define ADDR 0x0000000123 *(int *)ADDR = 1;
```

第二步的操作就是将第一步得到的32位地址，给转换成一个指针变量，并且操作这个地址里的值，唯一的区别，就是由于安全的考虑，多加了一个volatile 这样的关键字，但是他不会对我们产生其他的影响，而第三步，就是将前两部，结合在一起，根据传入的addr和bit计算得到32位的地址，然后强制类型转换，使得我们可以去操作这个地址里的值，OK，大功告成，整个的思路基本就是这样，应该不是很难把，至此相信各位已经能够理解什么是位带，以及该怎么去操作位带。

接下来，再写一种常用的位带操作的用法。由于上面的传入的addr是整个区域的基地址，因此，当你想去使用不同GPIO口的时候，采用上面的写法，你将麻烦需要多写好几个步骤，我自己常用的一种写法是下面这个样子的。

```
  # define BITBAND_REG(Reg,Bit) (*((uint32_t volatile*)(0x42000000u + (((uint32_t)&(Reg) - (uint32_t)0x40000000u)<<5) + (((uint32_t)(Bit))<<2))))
```

```
# define LED0 BITBAND_REG(GPIOF->ODR,9)

# define LED1 BITBAND_REG(GPIOF->ODR,10)
```

短短的三行代码，就已经解决了所有问题，输出控制小灯泡，即使再换用其他的端口，改动括号内的内容即可。Reg是操作部分的基地址，Bit就是第几位了。

原理就是，我已经知道，GPIO部分的基地址是0x42000000u ，那么我每次传入具体的GPIOx->ODR寄存器，在定义中，对其取地址，这样可以灵活访问各个不用IO输出，相当于把我们的操作给具体化了，<<5,<<2这两个就是乘32，乘4这样的概念，只不过位操作，会更快一点。