**嵌入式、C语言位操作的一些常见用法归纳**

嵌入式大杂烩 2019-11-28 00:20:09

分享关于位操作、寄存器配置的一些笔记：

**一、位操作简单介绍**

首先，以下是按位运算符：



在嵌入式编程中，常常需要对一些寄存器进行配置，有的情况下需要改变一个字节中的某一位或者几位，但是又不想改变其它位原有的值，这时就可以使用按位运算符进行操作。下面进行举例说明，假如有一个8位的TEST寄存器：



当我们要设置第0位bit0的值为1时，可能会这样进行设置：

TEST = 0x01;

但是，这样设置是不够准确的，因为这时候已经同时操作到了高7位：bit1~bit7，如果这高7位没有用到的话，这么设置没有什么影响；但是，如果这7位正在被使用，结果就不是我们想要的了。

在这种情况下，我们就可以借用按位操作运算符进行配置。

对于二进制位操作来说，不管该位原来的值是0还是1，它跟0进行&运算，得到的结果都是0，而跟1进行&运算，将保持原来的值不变；不管该位原来的值是0还是1，它跟1进行|运算，得到的结果都是1，而跟0进行|运算，将保持原来的值不变。

所以，此时可以设置为：

TEST = TEST | 0x01;

其意义为：TEST寄存器的高7位均不变，最低位变成1了。在实际编程中，常改写为：

TEST |= 0x01;

这种写法可以一定程度上简化代码，是 C 语言常用的一种编程风格。设置寄存器的某一位还有另一种操作方法，以上的等价方法如：

TEST |= (0x01 << 0);

第几位要置1就左移几位。

同样的，要给TEST的低4位清0，高4位保持不变，可以进行如下配置：

TEST &= 0xF0;

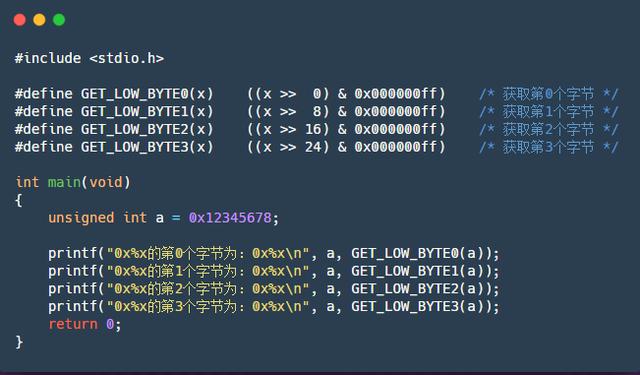
**二、嵌入式中位操作一些常见用法**

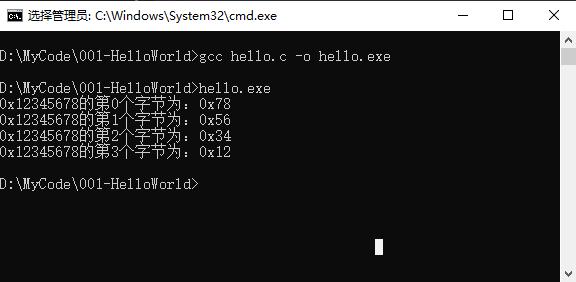
**1、一个32bit数据的位、字节读取操作**

**（1）获取单字节：**

#define GET\_LOW\_BYTE0(x) ((x >> 0) & 0x000000ff) /\* 获取第0个字节 \*/  
#define GET\_LOW\_BYTE1(x) ((x >> 8) & 0x000000ff) /\* 获取第1个字节 \*/  
#define GET\_LOW\_BYTE2(x) ((x >> 16) & 0x000000ff) /\* 获取第2个字节 \*/  
#define GET\_LOW\_BYTE3(x) ((x >> 24) & 0x000000ff) /\* 获取第3个字节 \*/

示例：

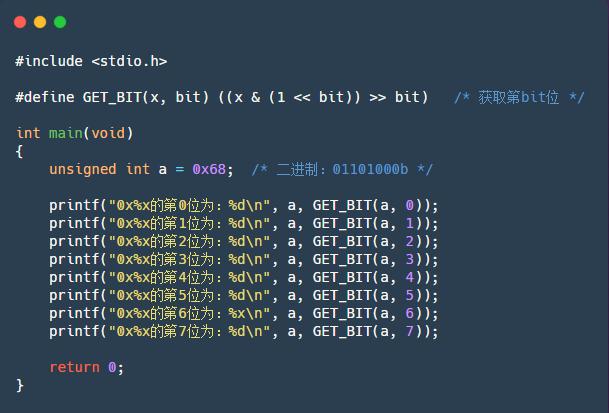


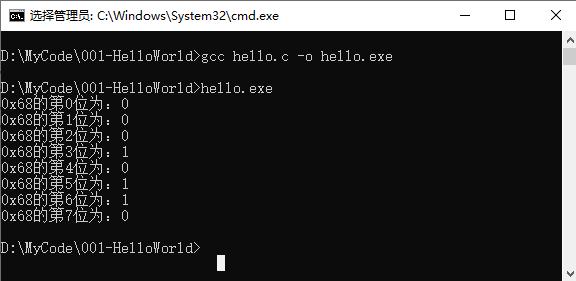


**（2）获取某一位：**

#define GET\_BIT(x, bit) ((x & (1 << bit)) >> bit) /\* 获取第bit位 \*/

示例：



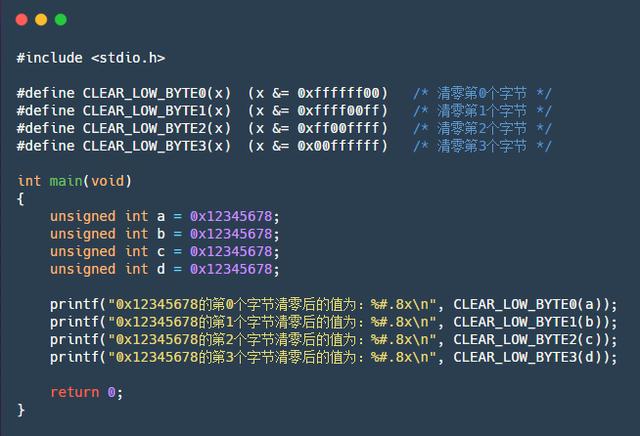


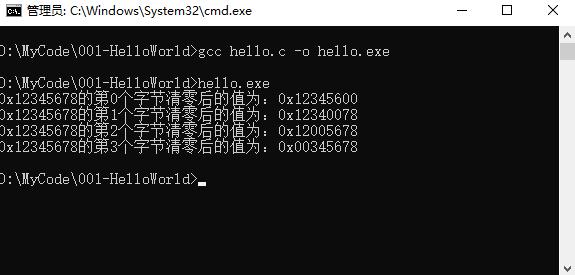
**2、一个32bit数据的位、字节清零操作**

**（1）清零某个字节：**

#define CLEAR\_LOW\_BYTE0(x) (x &= 0xffffff00) /\* 清零第0个字节 \*/  
#define CLEAR\_LOW\_BYTE1(x) (x &= 0xffff00ff) /\* 清零第1个字节 \*/  
#define CLEAR\_LOW\_BYTE2(x) (x &= 0xff00ffff) /\* 清零第2个字节 \*/  
#define CLEAR\_LOW\_BYTE3(x) (x &= 0x00ffffff) /\* 清零第3个字节 \*/

示例：

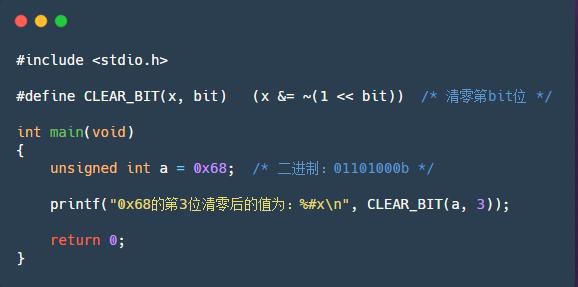


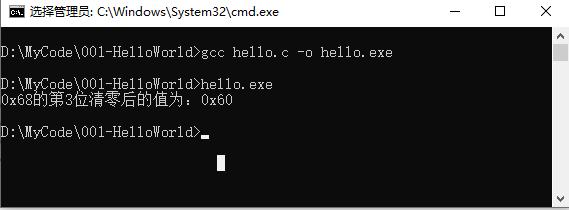


**（2）清零某一位：**

#define CLEAR\_BIT(x, bit) (x &= ~(1 << bit)) /\* 清零第bit位 \*/

示例：



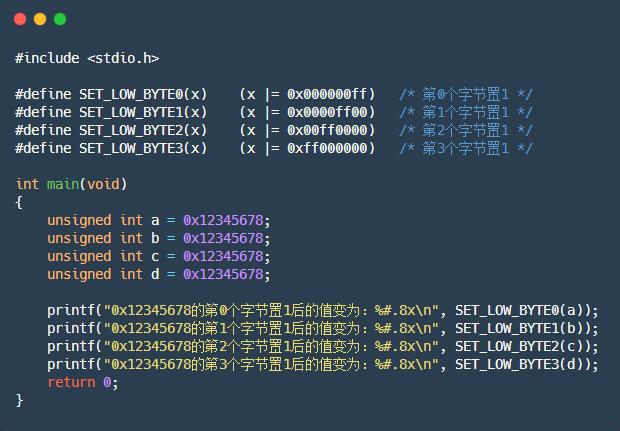


**3、一个32bit数据的位、字节置1操作**

**（1）置某个字节为1：**

#define SET\_LOW\_BYTE0(x) (x |= 0x000000ff) /\* 第0个字节置1 \*/   
#define SET\_LOW\_BYTE1(x) (x |= 0x0000ff00) /\* 第1个字节置1 \*/   
#define SET\_LOW\_BYTE2(x) (x |= 0x00ff0000) /\* 第2个字节置1 \*/   
#define SET\_LOW\_BYTE3(x) (x |= 0xff000000) /\* 第3个字节置1 \*/

示例：

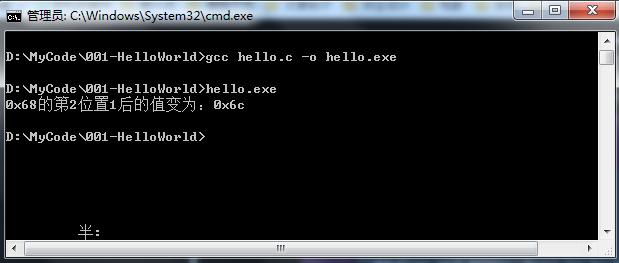




**（2）置位某一位：**

#define SET\_BIT(x, bit) (x |= (1 << bit)) /\* 置位第bit位 \*/





**4、判断某一位或某几位连续位的值**

**（1）判断某一位的值**

举例说明：判断0x68第3位的值。



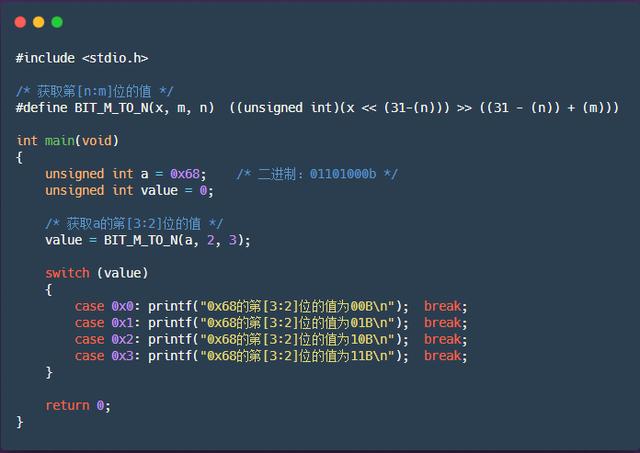


也就是说，要判断第几位的值，if里就左移几位（当然别过头了）。在嵌入式编程中，可通过这样的方式来判断寄存器的状态位是否被置位。

**（2）判断某几位连续位的值**

/\* 获取第[n:m]位的值 \*/  
#define BIT\_M\_TO\_N(x, m, n) ((unsigned int)(x << (31-(n))) >> ((31 - (n)) + (m)))

示例：





这是一个查询连续状态位的例子，因为有些情况不止有0、1两种状态，可能会有多种状态，这种情况下就可以用这种方法来取出状态位，再去执行相应操作。

以上是对32bit数据的一些操作进行总结，其它位数的数据类似，可根据需要进行修改。

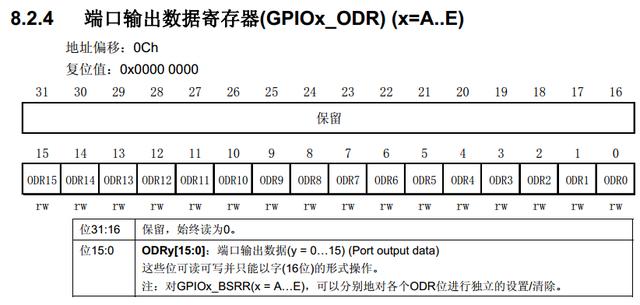
**三、STM32寄存器配置**

STM32有几套固件库，这些固件库函数以函数的形式进行1层或者多层封装（软件开发中很重要的思想之一：分层思想），但是到了最里面的一层就是对寄存器的配置。我们平时都比较喜欢固件库来开发，大概是因为固件库用起来比较简单，用固件库写出来的代码比较容易阅读。最近一段时间一直在配置寄存器，越发地发现使用寄存器来进行一些外设的配置也是很容易懂的。使用寄存器的方式编程无非就是往寄存器的某些位置1、清零以及对寄存器一些状态位进行判断、读取寄存器的内容等。

这些基本操作在上面的例子中已经有介绍，我们依旧以实例来巩固上面的知识点（以STM32F1xx为例）：

**（1）寄存器配置**

看一下GPIO功能的端口输出数据寄存器 (GPIOx\_ODR) (x=A..E) ：



假设我们要让PA10引脚输出高、输出低，可以这么做：

**方法一：**

GPIOA->ODR |= 1 << 10; /\* PA10输出高（置1操作） \*/  
GPIOA->ODR &= ~（1 << 10）; /\* PA10输出低（清0操作） \*/

也可用我们上面的置位、清零的宏定义：

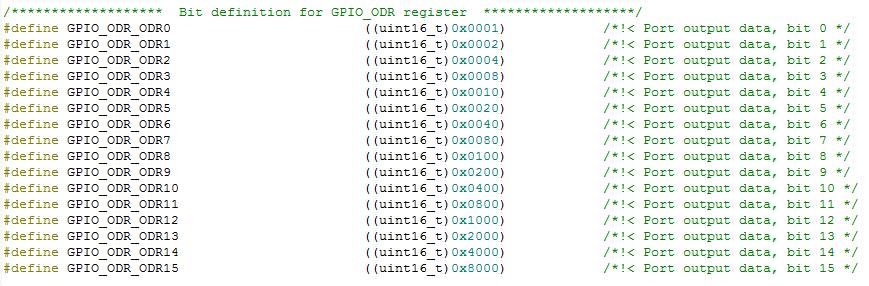
SET\_BIT(GPIOA->ODR, 10); /\* PA10输出高（置1操作） \*/  
CLEAR\_BIT(GPIOA->ODR, 10); /\* PA10输出低（清0操作） \*/

**方法二：**

GPIOA->ODR |= (uint16\_t)0x0400; /\* PA10输出高（置1操作） \*/  
GPIOA->ODR &= ~(uint16\_t)0x0400; /\* PA10输出低（清0操作） \*/

貌似第二种方法更麻烦？还得去细心地去构造一个数据。

但是，其实第二种方法其实是ST推荐我们用的方法，为什么这么说呢？因为ST官方已经把这些我们要用到的值给我们配好了，在stm32f10x.h中：



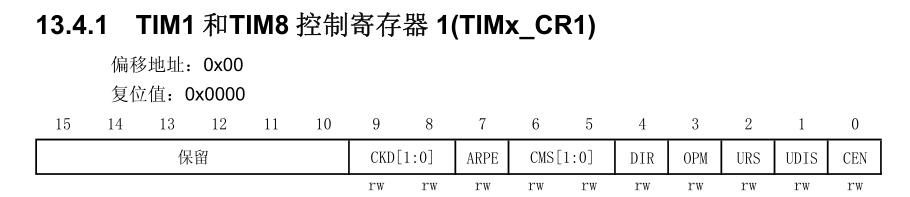
这个头文件中存放的就是外设寄存器的一些位配置。

所以我们的方法二等价于：

GPIOA->ODR |= GPIO\_ODR\_ODR10; /\* PA10输出高（置1操作） \*/  
GPIOA->ODR &= ~GPIO\_ODR\_ODR10; /\* PA10输出低（清0操作） \*/

两种方法都是很好的方法，但方法一似乎更好理解。

配置连续几位的方法也是一样的，就不介绍了。简单介绍配置不连续位的方法，以TIM1的CR1寄存器为例：



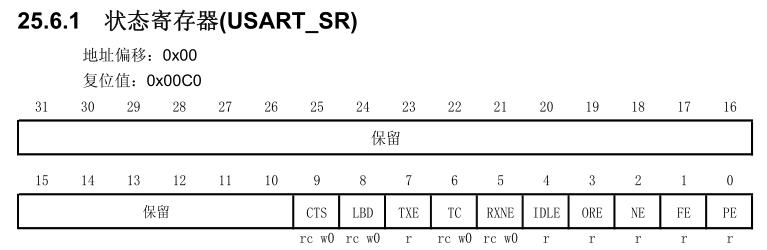
设置CEN位为1、设置CMS[1:0]位为01、设置CKD[1:0]位为10：

TIM1->CR1 |= （0x1 << 1)| (0x1 << 5) |(0x2 << 8);

这是组合的写法。当然，像上面一样拆开来写也是可以的。

**（2）判断标志位**

以状态寄存器(USART\_SR) 为例：



判断RXNE是否被置位：

/\* 数据寄存器非空，RXNE标志置位 \*/  
if (USART1->SR & (1 << 5))  
{  
 /\* 其它代码 \*/  
   
 USART1->SR &= ~(1 << 5); /\* 清零RXNE标志 \*/  
}

或者：

/\* 数据寄存器非空，RXNE标志置位 \*/  
if (USART1->SR & USART\_SR\_RXNE)  
{  
 /\* 其它代码 \*/  
   
 USART1->SR &= ~USART\_SR\_RXNE; /\* 清零RXNE标志 \*/  
}

**四、总结**

以上就是本次关于位操作的一点总结笔记，有必要掌握。虽然说在用STM32的时候有库函数可以用，但是最接近芯片内部原理的还是寄存器。有可能之后有用到其它芯片没有像ST这样把寄存器相关配置封装得那么好，那就不得不直接操控寄存器了。

此外，使用库函数的方式代码占用空间大，用寄存器的话，代码占用空间小。之前有个需求，我能用的Flash的空间大小只有4KB，遇到类似这样的情况就不能那么随性的用库函数了。