**阅读笔记**

FSMC:是STM32系列采用的一种新型的[存储器](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E5%99%A8/1583185" \t "https://baike.baidu.com/item/_blank)扩展技术。在外部存储器扩展方面具有独特的优势，可根据系统的应用需要，方便地进行不同类型大容量[静态存储器](https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E5%AD%98%E5%82%A8%E5%99%A8/6797116" \t "https://baike.baidu.com/item/_blank)的扩展。

STM32位段:位操作就是可以读/写单独的一个比特位，在STM32中没有像51单片机的sbit来实行位定义，但是它可以通过位带别名区来实现。在STM32中有两个地方实现了位带操作，一个是SRAM区的最低1MB空间，另一个是外设区最低1MB空间。

这两个1MB的空间可以像普通RAM一样操作外(修改内容时用读-改-写)，它们还有自己的位带别名区，位带别名区把这1MB的空间的每一位膨胀为一个32位的字。确切的说，这个字就是一个地址，当操作这个地址时，就可以达到操作这个位带区某个位的目的。

在位带区中，每个比特位都映射到别名地址区的一个地址，注意，这只是只有LSB有效的字(最低一位有效的字)。当一个别名地址被访问时，会把该地址转换为为位带操作。

位段操作，使得1MB的SRAM就有32MB的对应别名区空间，1位膨胀到32位，但效率更高，(在中断的时候)具有更安全的作用。

简单说，位段区就是把1MB的SRAM空间和外设空间映射到一个更高的地址区域。并且膨胀了32倍(stm32是32位的)。操作原始地址和更高地址都有效。

Stm32 Flash:根据用途，STM32片内的FLASH分成两部分：主存储块、信息块。 主存储块用于存储程序，我们写的程序一般存储在这里。 信息块又分成两部分：系统存储器、选项字节。系统存储器存储用于存放在系统存储器自举模式下的启动程序（BootLoader），当使用ISP方式加载程序时，就是由这个程序执行。这个区域由芯片厂写入BootLoader，然后锁死，用户是无法改变这个区域的。 选项字节存储芯片的配置信息及对主存储块的保护信息。比如stm32f103c8t6有2K系统存储器区和64K Flash。

使用ISP下载时，要选择在系统存储器区启动

所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式

**开发笔记**

Extern ‘’C”

比如说你用C 开发了一个DLL 库，为了能够让C ++语言也能够调用你的DLL 输出(Export) 的函数，你需要用extern "C" 来强制编译器不要修改你的函数名。

通常，在C 语言的头文件中经常可以看到类似下面这种形式的代码：

**C代码  IMG_256**

1. #ifdef \_\_cplusplus
2. **extern** "C" {
3. #endif
5. /\*\*\*\* some declaration or so \*\*\*\*\*/
7. #ifdef \_\_cplusplus
8. }
9. #endif

那么，这种写法什么用呢？实际上，这是为了让CPP 能够与C 接口而采用的一种语法形式。之所以采用这种方式，是因为两种语言之间的一些差异所导致的。由于CPP 支持多态性，也就是具有相同函数名的函数可以完成不同的功能，CPP 通常是通过参数区分具体调用的是哪一个函数。在编译的时候，CPP 编译器会将参数类型和函数名连接在一起，于是在程序编译成为目标文件以后，CPP 编译器可以直接根据目标文件中的符号名将多个目标文件连接成一个目标文件或者可执行文件。但是在C 语言中，由于完全没有多态性的概念，C 编译器在编译时除了会在函数名前面添加一个下划线之外，什么也不会做（至少很多编译器都是这样干的）。由于这种的原因，当采用CPP 与C 混合编程的时候，就可能会出问题。假设在某一个头文件中定义了这样一个函数：

int foo(int a, int b);

而这个函数的实现位于一个.c 文件中，同时，在.cpp 文件中调用了这个函数。那么，当CPP 编译器编译这个函数的时候，就有可能会把这个函数名改成\_fooii ，这里的ii 表示函数的第一参数和第二参数都是整型。而C编译器却有可能将这个函数名编译成\_foo 。也就是说，在CPP 编译器得到的目标文件中，foo() 函数是由\_fooii符号来引用的，而在C 编译器生成的目标文件中，foo() 函数是由\_foo 指代的。但连接器工作的时候，它可不管上层采用的是什么语言，它只认目标文件中的符号。于是，连接器将会发现在.cpp 中调用了foo() 函数，但是在其它的目标文件中却找不到\_fooii 这个符号，于是提示连接过程出错。extern "C" {} 这种语法形式就是用来解决这个问题的。

综合上面的分析，我们可以得出如下结论：采用extern "C" {} 这种形式的声明，可以使得CPP 与C之间的接口具有互通性，不会由于语言内部的机制导致连接目标文件的时候出现错误。

注意：

用g++编译cpp程序时，编译器会定义宏 \_\_cplusplus ,可根据\_\_cplusplus是否定义决定是否需要extern "C"。