**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

中断系统

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

Stm32中有16个内部异常和68个外部中断(相对cortex m3内核来讲的外部)。

内部异常我们无法控制，但是68个外部中断我们可以控制。这里的68个外部中断是指和cortex-m3内核相连的NVIC控制器，分出了68个通道(cortex-m3的NVIC其实提供了240个通道，但是stm32只使用了68个通道)，供外设从通道发送中断信号。

而NVIC根据每个通道的优先级，确定谁先得到响应。并且这68个通道已经固定的分配给了外设。其中，有xx个通道，是分配给了单片机的外部IO口产生中断信号

外部中断有16个优先级，用4bit控制。4bit要分成两组看，分别代表抢占优先级和子优先级。具体给抢占优先级和子优先级划分几位，要配置AIRC寄存器。假如各分到了2位，那么抢占优先级和子优先级的的优先序号都是0 1 2 3。数字越小优先级越高

具体优先级的确定和嵌套规则。ARM cortex\_m3（STM32）规定

a/ 只能高抢先优先级的中断可以打断低抢先优先级的中断服务，构成中断嵌套。

b/ 当 2（n）个相同抢先优先级的中断出现，它们之间不能构成中断嵌套，但 STM32 首

先响应子优先级高的中断。

c/ 当 2（n）个相同抢先优先级和相同子优先级的中断出现，STM32 首先响应中断通道

所对应的中断向量地址低的那个中断（见 ROM0008，表 52）。

具体一点：

0 号抢先优先级的中断，可以打断任何中断抢先优先级为非 0 号的中断；1 号抢先优先

级的中断，可以打断任何中断抢先优先级为 2、3、4 号的中断；

……；

构成中断嵌套。

如果两个中断的抢先优先级相同，谁先出现，就先响应谁，不构成嵌套。如果一起出现（或挂在那里等待），就看它们 2 个谁的子优先级高了，如果子优先级也相同，就看它们的中断向量位置了。

NVIC中有很多寄存器可以设置中断通道的属性和参数。Stm32只使用了部分NVIC的寄存器

另外的几个寄存器，也是需要使用的（请具体参考相关的资料）

\_\_IO uint8\_t SHP[12]; /\*!<System Handlers Priority Registers(4-7,8-11,12-15) \*/

同每个外部中断通道优先级定义字相同，它们是内核中断通道4-15的优先级定义字所在

的寄存器。用户可以通过设置SHP[n]，改变内部中断通道的优先级。(改变内部异常的处理优先级)

\_\_IO uint32\_t VTOR; /\*!< Vector Table Offset Register \*/

如果你的代码要在RAM中启动执行，就需要对这个寄存器进行设置。

中断处理和默认情况，都是使用msp的

代码主动申请进入异常，通常作为系统调用

把返回地址直接存储在寄存器中。这样足以使很多只有 1 级子程序调用的代码无需访问内存（堆栈内存），从而提高了子程序调用的效率。如果多于 1 级，则需要把前一级的 R14 值压到堆栈里。

复位后，处理器默认进入线程模式，特权极访问。

可嵌套中断支持的作用范围很广，覆盖了所有的外部中断和绝大多数系统异常。外在表现是，

这些异常都可以被赋予不同的优先级。当前优先级被存储在 xPSR 的专用字段中。当

一个异常发生 时，硬件会自动比较该异常的优先级是否比当前的异常优先级更高。如果发现来了更高优先级的异常，处理器就会中断当前的中断服务例程（或者是普通程序），而服务新来的异常——即立即抢占。

所谓的重入，就是指某段子程序还没有执行完，就因为中断或者是多任务操作系统的调度原因，导致该子程序在一个新的寄存器上下文中被执行（请不要把重入与递归混淆，它们有本质的区别）。这种情况常常会闹出乱 子，因此有“可重入性”的研究(重入没有保存当前堆栈和cpu环境。递归保存了)

既可以屏蔽优先级低于某个阈值的中断/异常［译注 8］(设置 BASEPRI 寄存器)，也可以全体封杀(设 置 PRIMASK 和 FAULTMASK 寄存器)。

有两条代码存储区总线负责对代码存储区的访问，分别是 I-Code 总线和 D‐Code 总线。前者用于取指，后者用于查表等操作，它们按最佳执行速度进行优化。

系统总线用于访问内存和外设，覆盖的区域包括 SRAM，片上外设，片外 RAM，片外扩展设备，以及系统级存储区的部分空间

它支持 16‐4‐1=11 种系统异常（保留了 4+1 个档位），

CM3 的所有中断机制都由 NVIC 实现。除了支持 240 条中断之外，NVIC 还支持 16‐4‐1=11 个内部异常源，可以实现 fault 管理机制。结果，CM3 就有了 256 个预定义的异常类型，

SVCall 可编程 系统服务调用

PendSV 可编程 为系统设备而设的“可悬挂请求”（pendable request）

SysTick 可编程 系统滴答定时器（也就是周期性溢出的时基定时器)

**16** IRQ #0 可编程 外中断#0

**17** IRQ #1 可编程 外中断#1

**…** … … …

**255** IRQ #239 可编程 外中断#239

虽然 CM3 是支持 240 个外中断的，但具体使用了多少个是由芯片生产商决定。CM3 还有

一个 NMI（不可屏蔽中断）输入脚。当它被置为有效（assert）时，NMI 服务例程会无条件地执行。(stm32支持了68个)

要注意的是，并不是每个应用都必须用齐两个堆栈指针。简单的应用程序只使用 MSP

就够了。堆栈指针用于访问堆栈，并且 PUSH 指令和 POP 指令默认使用 SP。

PUSH {R0} ; \*(--R13)=R0。R13 是 long\*的指针

POP {R0} ; R0=\*R13++

另外，PUSH 和 POP 还 能一次操作多个寄存器，如下所示：

subroutine\_1

PUSH {R0

-

R7, R12, R14} ; 保存寄存器列表

… ; 执行处理

POP {R0

-

R7, R12, R14} ; 恢复寄存器列表

BX R14 ; 返回到主调函数

寄存器的 PUSH 和 POP 操作永远都是 4 字节对齐的

——也就是说他们的地址必须是 0x4,0x8,0xc,……。这样一来，R13 的最低两位被硬线连接到0,并且总是读出0（Read As Zero）。

为了避免系统堆栈因应用程序的错误使用而毁坏，你可以给应用程序专门配

一个堆栈，不让它共享操作系统内核的堆栈。在这个管理制度下，运行在线程模式的用户代码使用 PSP，而异常服务例程则使用 MSP。这两个堆栈指针的切换是全自动的，就在出入异常服务例程时由硬件处理。第 8 章将详细讨论此主题。

作为中断功能的强化，NVIC 还有一条 NMI 输入信号线。NMI 究竟被拿去做什么，还要

视处理器的设计而定。在多数情况下，NMI 会被连接到一个看门狗定时器，有时也会是电压监视功能块，以便在电压掉至危险级别后警告处理器。NMI 可以在任何时间被激活，甚至是在处理器刚刚复位之后。

当一个发生的异常被 CM3 内核接受，对应的异常 handler 就会执行。为了决定 handler 的入 口地址，CM3 使用了“向量表查表机制”。这里使用一张向量表。向量表其实是一个 WORD （32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该异常 handler 的入口地址。向量表的存储位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0 处必须包含一张向量表，用于初始时的异常分配。0 号异常的功能则是个另类，它并不是什么入口地址，而是给出了复位后 MSP 的初值。

在离开复位状态后，CM3 做的第一件事就是读取下列两个 32 位整数的值：

z 从地址 0x0000,0000 处取出 MSP 的初始值。

z 从地址 0x0000,0004 处取出 PC 的初始值

——这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取指。

**使用stm32的EXTI**

首先stm32的exti是stm32中断系统68个中断源的子集。一共可以使用19个exti中断，但是有些exti中断是被放到1根线上的，所以实际上不必使用19根NVIC的中断通道。

起始点是输入线，EXTI 控制器有 19 个中断/事件输入线，这些输入线0-15分别是给GPIOx0-GPIOx15使用的，通过配置寄存器选择EXITx连到GPIOx的哪个口。可以通过寄存器设置为任意一个 GPIO，也可以是一些外设的事件，输入线一般是存在电平变化的信号。

**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"。