# 第5章 STM32单片机外部中断EXTI



退出

# 5.1 中断的基本概念

生活实例：

假如你有朋友下午要来拜访，可又不知道他具体什么时候到。为了提高效率，你就边看书边等。在看书过程中。。。。。。

解决方法：

- 无条件传送方式：无须了解外部设备状态，直接向外部设备发送数据，因此适合于快速设备或者状态明确的外部设备。
- 查询方式：依据查询状态传输数据。在单片机任务不太繁忙，对外部设备响应速度要求不高的情况常采用这种方法。
- 中断方式：外部设备发送请求，单片机接到请求后立即中断当前工作，处理外部设备的请求，处理完毕后继续处理未完成的工作。

## 中断定义

**CPU暂时中止其正在执行的程序，转去执行请求中断的那个外设或事件的服务程序，等处理完毕后再返回执行原来中止的程序，叫做中断。**

中断的作用：

- 提高CPU工作效率
- 具有实时处理功能
- 具有故障处理功能
- 实现分时操作

退出

## 中断源

中断源：能引发中断的事件。通常，中断源都与外设有关。

- 在生活中，门铃的铃声是一个中断源，它由门铃这个外设发出。
- 在嵌入式系统中，常见的中断源有按键按下和释放、定时器溢出、串口收到数据等，与此相关的外设有键盘、定时器和串口等。

中断请求标志位：每个中断源都有它对应的中断标志位。

- 一旦该中断发生，它的中断标志位就会被置位。如果中断标志位被清除，那么它所对应的中断便不会再被响应。因此，一般在中断服务程序最后要将对应的中断标志位清零。

## 中断屏蔽

可以通过设置相应的中断屏蔽位，禁止CPU响应某个中断，从而实现中断屏蔽。

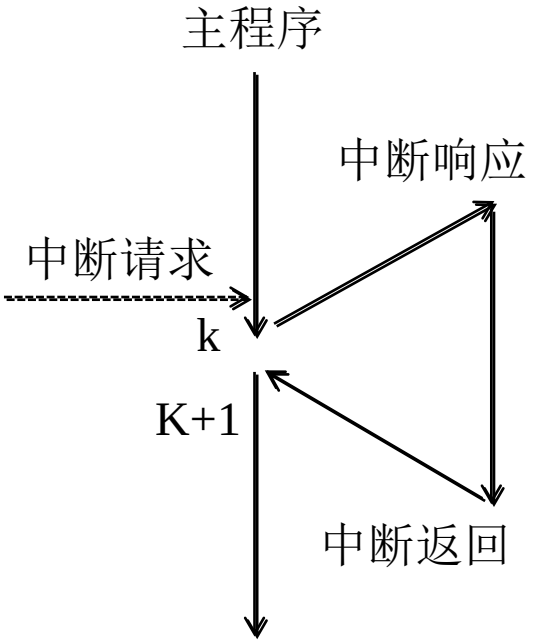
- **可屏蔽中断**

  在生活中，门铃的铃声是一个可屏蔽中断。

- **不可屏蔽中断**

  在计算机系统中，电源故障、内存出错、总线出错等是不可屏蔽中断。

## 中断处理过程




- **中断响应（硬件自动实现）**
  - ✓ 保护现场
  - ✓ 找到该中断对应的中断服务程序的地址——中断向量表
- **执行中断服务程序（用户编程）**

中断服务程序通常是由用户编写的特殊函数，用来实现对该中断真正的处理操作，具有以下特点：

- ✓ 中断服务程序既没有参数，也没有返回值，更不由用户调用，而是当某个事件产生一个中断时由硬件自动调用。
- ✓ 在中断服务程序中修改、在其他程序中访问的变量，在其定义和声明时要在前面加上volatile修饰词。
- ✓ 中断服务程序要求应当尽量的简短。

- **中断返回（硬件自动实现）**

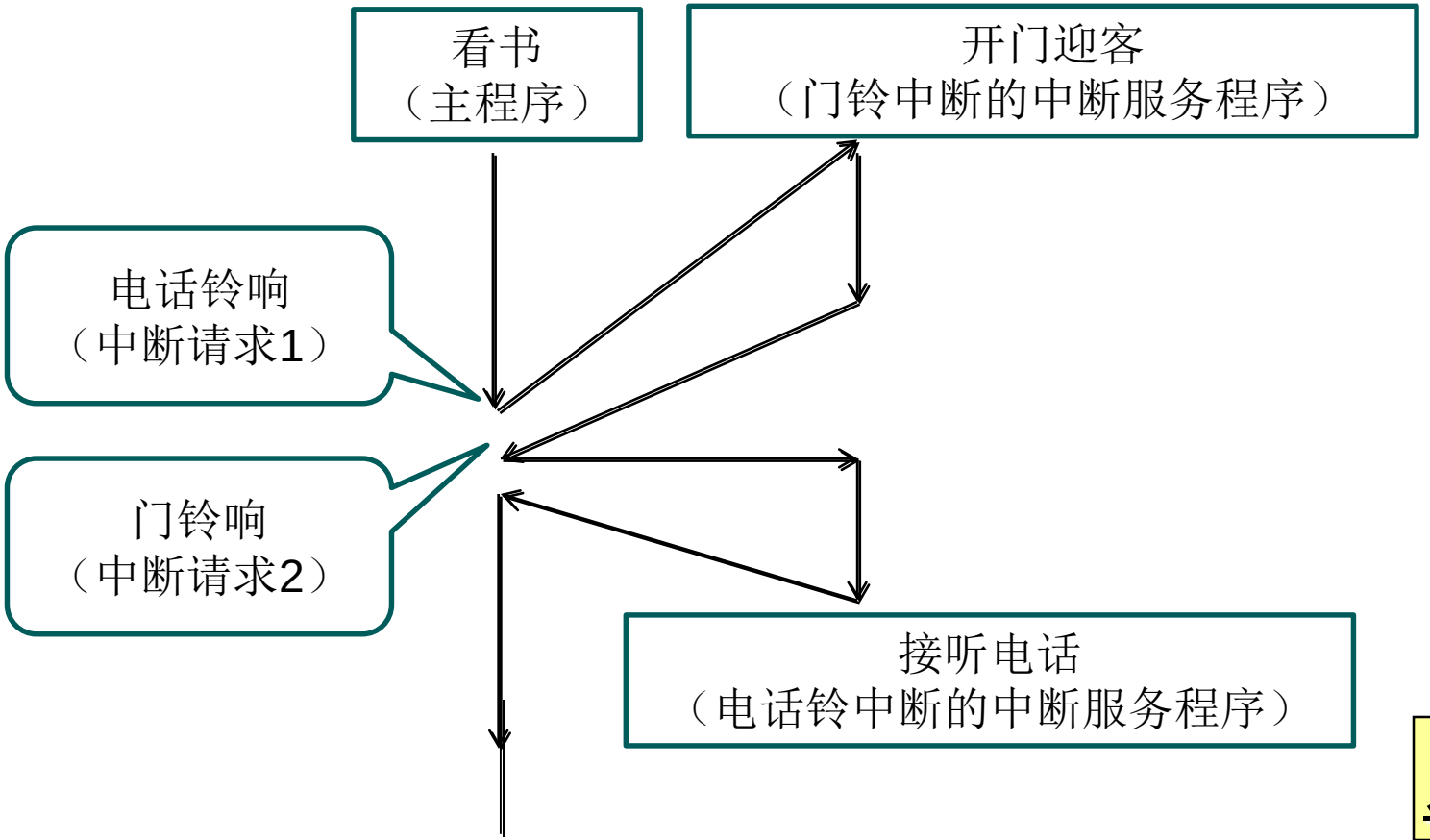
恢复现场

退出

## 中断优先级

嵌入式系统中的中断往往不止一个，那么，对于多个同时发生的中断或者嵌套发生的中断，CPU又该如何处理？应该先响应哪一个中断？

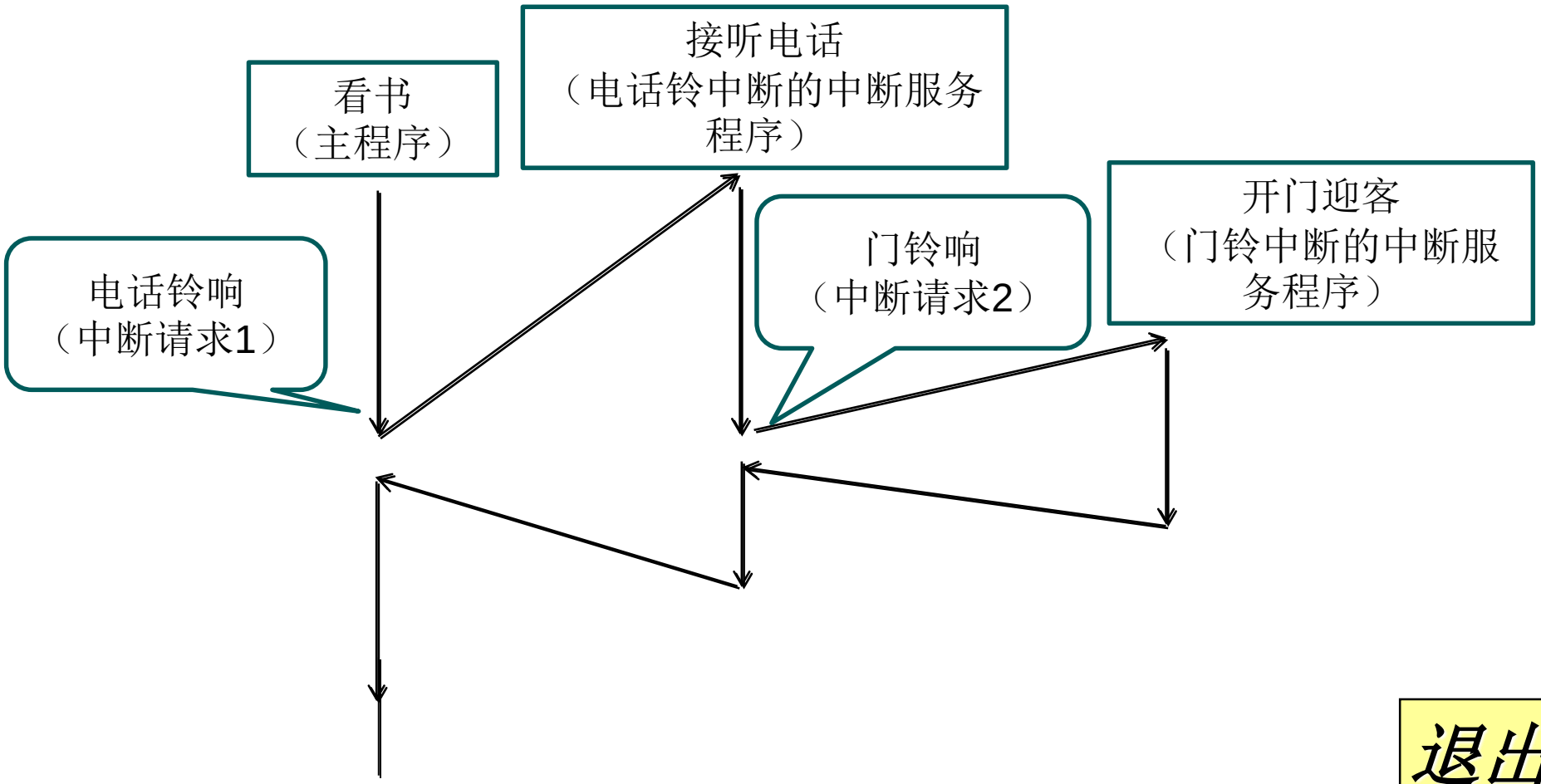
答案就是**中断优先级。**




退出

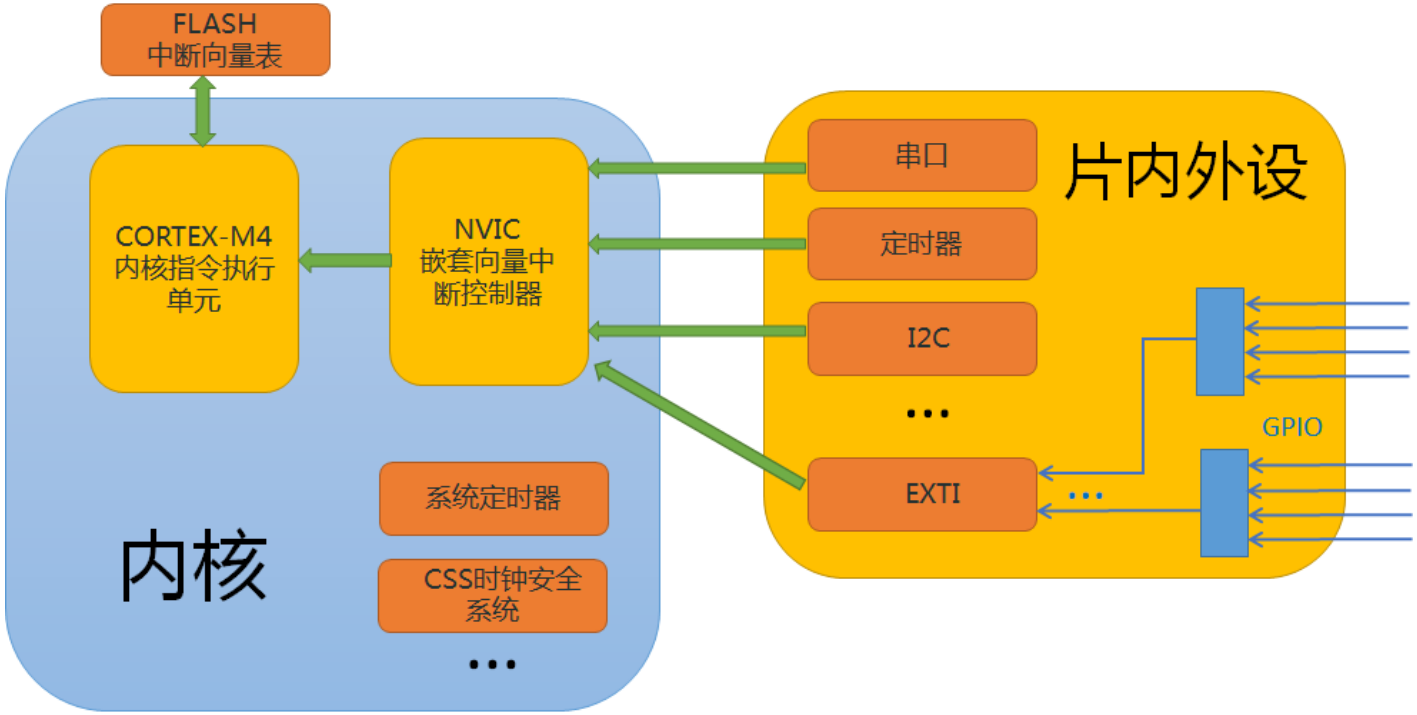
## 中断嵌套

在嵌入式系统中，中断嵌套是指当系统正在执行一个中断服务时，又有新的中断事件发生而产生了新的中断请求。




退出

# 5.2 STM32F103中断系统




退出

## 5.2.1 嵌套向量中断控制器NVIC

NVIC集成在ARM Cortex-M3内核中，与中央处理器核心CM3Core紧密耦合，从而实现低延迟的中断处理和高效地处理晚到的较高优先级的中断。

- Cortex-M3内核支持 256 个中断，其中包含了 16 个内核中断和 240 个非内核中断。但 STM32F10x 只使用 Cortex-M3 内核的一部分功能：支持84个中断，包括16个内部中断和68个非内核中断；
- 使用4位优先级设置，具有16级可编程中断优先级
- 中断响应时处理器状态的自动保存，无需额外指令；
- 中断返回时处理器状态的自动恢复，无需额外指令；
- 支持嵌套和向量中断。

退出

## STM32F103中断优先级

### 1. 抢占优先级（Preempting Priority）

高抢占式优先级的中断事件会打断当前的中断程序运行，称为中断嵌套。

### 2. 响应优先级（Subpriority）

在抢占式优先级相同的情况下，高响应优先级的中断优先被响应。

### 3. 判断中断是否会被响应的依据

首先是抢占式优先级，其次是响应优先级；抢占式优先级决定是否会有中断嵌套。

### 4. 优先级冲突的处理

具有高抢占式优先级的中断可以在具有低抢占式优先级的中断处理过程中被响应，即中断的嵌套，或者说高抢占式优先级的中断可以嵌套低抢占式优先级的中断。

退出

例如有四个中断向量：

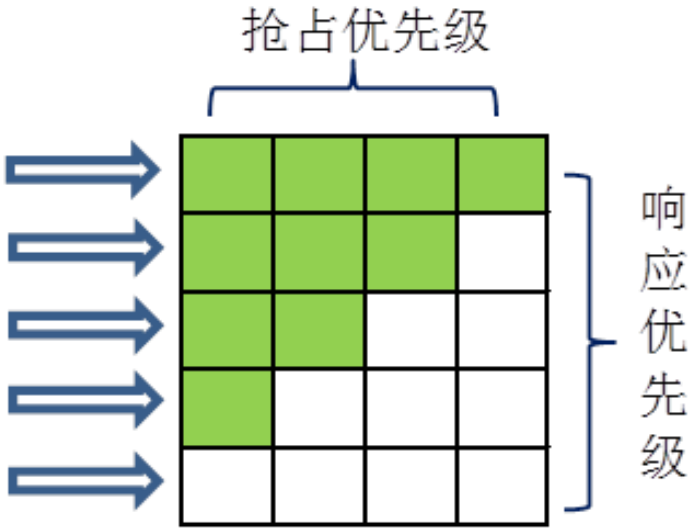
| 中断向量 | 抢占优先级 | 响应优先级 |
| --- | --- | --- |
| A | 0 | 0 |
| B | 1 | 0 |
| C | 1 | 1 |
| D | 0 | 1 |

如果内核正在执行C的中断服务程序，则能被抢占优先级更高的中断A打断；由于B和C的抢占优先级相同，所以C不会被B打断；但如果B和C两个中断同时到达，则响应优先级更高的B会被先执行。

## STM32中对中断优先级的定义

16个可编程优先级，用4个bit位表示。

| 优先级组别 | 抢占式优先级 | | 响应式优先级 | |
|---|---|---|---|---|
| | 位数 | 级数 | 位数 | 级数 |
| 4组 | 4 | 16 | 0 | 0 |
| 3组 | 3 | 8 | 1 | 2 |
| 2组 | 2 | 4 | 2 | 4 |
| 1组 | 1 | 2 | 3 | 8 |
| 0组 | 0 | 0 | 4 | 16 |



## STM32中断优先级总结

- 抢占优先级的级别高于响应优先级。而**数值越小所代表的优先级就越高**。同一时刻发生的中断，优先处理优先级较高的中断。
- 高优先级的抢占优先级是可以打断正在进行的低抢占优先级中断的。而**抢占优先级相同的中断，高优先级的响应优先级不可以打断低响应优先级的中断**。
- 抢占优先级相同就看响应优先级，同样数值越小优先级越高。
- 如果两个中断的抢占优先级和响应优先级都是一样的话，则看**哪个中断先发生就先执行**。如果同时发生则优先处理中断向量表中排位顺序较小的那个。

退出

# STM32F103中断向量表

STM32F103各个中断对应的中断服务程序的入口地址统一存放在STM32F103的中断向量表中。STM32F103的中断向量表（课本P60），一般位于其存储器的0地址处。

| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|---|---|---|---|---|---|
| | - | - | - | 保留 | 0x0000_0000 |
| | -3 | 固定 | Reset | 复位 | 0x0000_0004 |
| | -2 | 固定 | NMI | 不可屏蔽中断RCC时钟安全系统(CSS)联接到NMI向量 | 0x0000_0008 |
| | -1 | 固定 | 硬件失效 | 所有类型的失效 | 0x0000_000C |
| | 0 | 可设置 | 存储管理 | 存储器管理 | 0x0000_0010 |
| | 1 | 可设置 | 总线错误 | 预取指失败，存储器访问失败 | 0x0000_0014 |
| | 2 | 可设置 | 错误应用 | 未定义的指令或非法状态 | 0x0000_0018 |
| | - | - | - | 保留 | 0x0000_001C |
| | - | - | - | 保留 | 0x0000_0020 |
| | - | - | - | 保留 | 0x0000_0024 |
| | - | - | - | 保留 | 0x0000_0028 |
| | 3 | 可设置 | SVCall | 通过SWI指令的系统服务调用 | 0x0000_002C |
| | 4 | 可设置 | 调试监控(DebugMonitor) | 调试监控器 | 0x0000_0030 |
| | - | - | - | 保留 | 0x0000_0034 |
| | 5 | 可设置 | PendSV | 可挂起的系统服务 | 0x0000_0038 |
| | 6 | 可设置 | SysTick | 系统嘀嗒定时器 | 0x0000_003C |
| 0 | 7 | 可设置 | WWDG | 窗口定时器中断 | 0x0000_0040 |
| 1 | 8 | 可设置 | PVD | 连到EXTI的电源电压检测(PVD)中断 | 0x0000_0044 |
| 2 | 9 | 可设置 | TAMPER | 侵入检测中断 | 0x0000_0048 |
| 3 | 10 | 可设置 | RTC | 实时时钟(RTC)全局中断 | 0x0000_004C |
| 4 | 11 | 可设置 | FLASH | 闪存全局中断 | 0x0000_0050 |
| 5 | 12 | 可设置 | RCC | 复位和时钟控制(RCC)中断 | 0x0000_0054 |

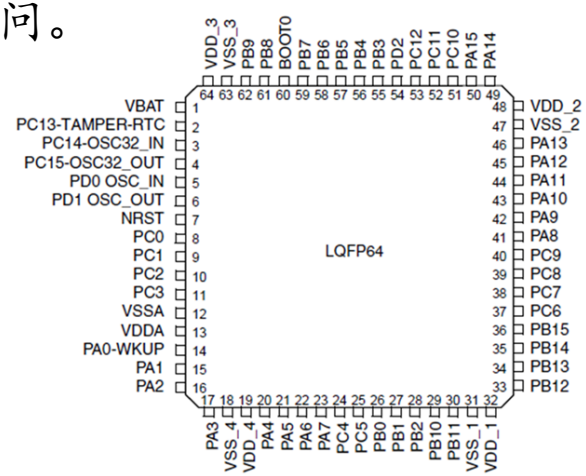
| | | | | | |
|---|---|---|---|---|---|
| 6 | 13 | 可设置 | EXTI0 | EXTI线0中断 | 0x0000_0058 |
| 7 | 14 | 可设置 | EXTI1 | EXTI线1中断 | 0x0000_005C |
| 8 | 15 | 可设置 | EXTI2 | EXTI线2中断 | 0x0000_0060 |
| 9 | 16 | 可设置 | EXTI3 | EXTI线3中断 | 0x0000_0064 |
| 10 | 17 | 可设置 | EXTI4 | EXTI线4中断 | 0x0000_0068 |
| 11 | 18 | 可设置 | DMA1通道1 | DMA1通道1全局中断 | 0x0000_006C |
| 12 | 19 | 可设置 | DMA1通道2 | DMA1通道2全局中断 | 0x0000_0070 |
| 13 | 20 | 可设置 | DMA1通道3 | DMA1通道3全局中断 | 0x0000_0074 |
| 14 | 21 | 可设置 | DMA1通道4 | DMA1通道4全局中断 | 0x0000_0078 |
| 15 | 22 | 可设置 | DMA1通道5 | DMA1通道5全局中断 | 0x0000_007C |
| 16 | 23 | 可设置 | DMA1通道6 | DMA1通道6全局中断 | 0x0000_0080 |
| 17 | 24 | 可设置 | DMA1通道7 | DMA1通道7全局中断 | 0x0000_0084 |
| 18 | 25 | 可设置 | ADC1_2 | ADC1和ADC2全局中断 | 0x0000_0088 |
| 19 | 26 | 可设置 | CAN1_TX | CAN1发送中断 | 0x0000_008C |
| 20 | 27 | 可设置 | CAN1_RX0 | CAN1接收0中断 | 0x0000_0090 |
| 21 | 28 | 可设置 | CAN1_RX1 | CAN1接收1中断 | 0x0000_0094 |
| 22 | 29 | 可设置 | CAN_SCE | CAN1 SCE中断 | 0x0000_0098 |
| 23 | 30 | 可设置 | EXTI9_5 | EXTI线[9:5]中断 | 0x0000_009C |
| 24 | 31 | 可设置 | TIM1_BRK | TIM1刹车中断 | 0x0000_00A0 |
| 25 | 32 | 可设置 | TIM1_UP | TIM1更新中断 | 0x0000_00A4 |
| 26 | 33 | 可设置 | TIM1_TRG_COM | TIM1触发和通信中断 | 0x0000_00A8 |
| 27 | 34 | 可设置 | TIM1_CC | TIM1捕获比较中断 | 0x0000_00AC |
| 28 | 35 | 可设置 | TIM2 | TIM2全局中断 | 0x0000_00B0 |
| 29 | 36 | 可设置 | TIM3 | TIM3全局中断 | 0x0000_00B4 |
| 30 | 37 | 可设置 | TIM4 | TIM4全局中断 | 0x0000_00B8 |

# 启动配置

STM32F103系列单片机因为固定的存储器映像，代码区始终从地址（0x00000000）最初开始，通过Icode和Dcode总线访问。启动之后，CPU从最初地址开始获取堆栈顶的地址，并**从启动存储器0x00000004指示的地址开始执行代码**。而SARM始终从0x20000000开始，直接通过系统总线访问。

由于单片机CPU始终从ICode总线获取复位向量，因此启动只能从代码区开始，常用的启动方法是使用Flash启动（程序代码常存放于Flash中）。

但是STM32F103提供了特殊的机制，系统可以从三个地方进行启动：**Flash、系统存储区、内置SRAM**。



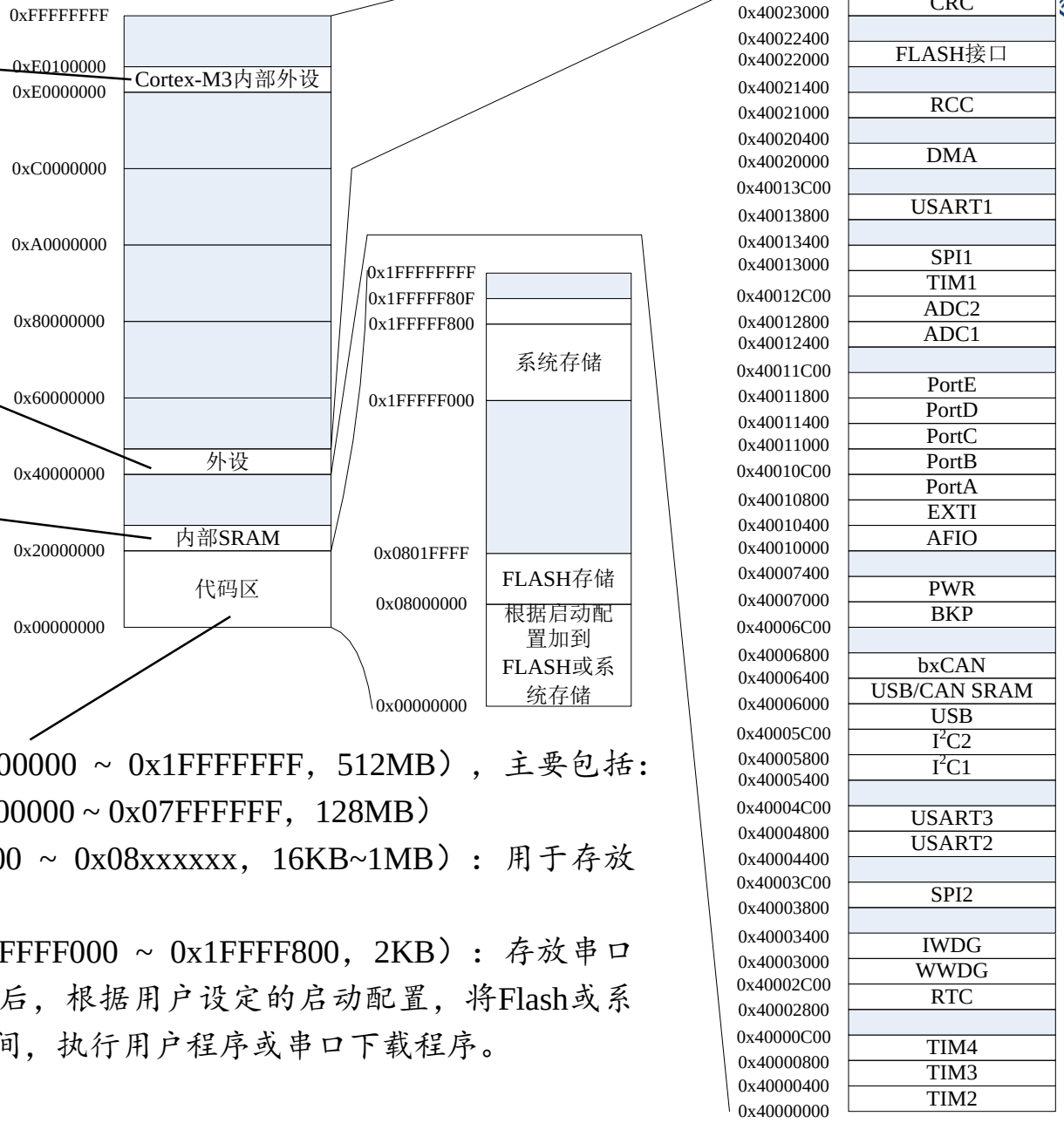

| 启动模式 | 选择引脚 | 启动模式 | 说明 |
|---|---|---|---|
| BOOT1 | BOOT0 | × | × |
| X | 0 | 主闪存存储器 | Flash被选为启动区域 |
| 0 | 1 | 系统存储器 | 系统存储区被选为启动区域 |
| 1 | 1 | 内置SRAM | 内置SRAM被选为启动区域 |

# 嵌入式系统

**内部外设区：** 用于调试组件等私有外设。例如：FPB, DWT, ITM, ETM, TPIU, ROM表等。

**外设区：** 外设寄存器地址空间。

**内部SRAM**（0x20000000 ~ 0x200xxxxx，6~96KB）：保护程序运行时产生的临时数据的随机存储器。




**代码区：**（0x00000000 ~ 0x1FFFFFFF，512MB），主要包括：

**启动空间**（0x00000000 ~ 0x07FFFFF，128MB）

**Flash**（0x08000000 ~ 0x08xxxxxx，16KB~1MB）：用于存放用户编写的程序

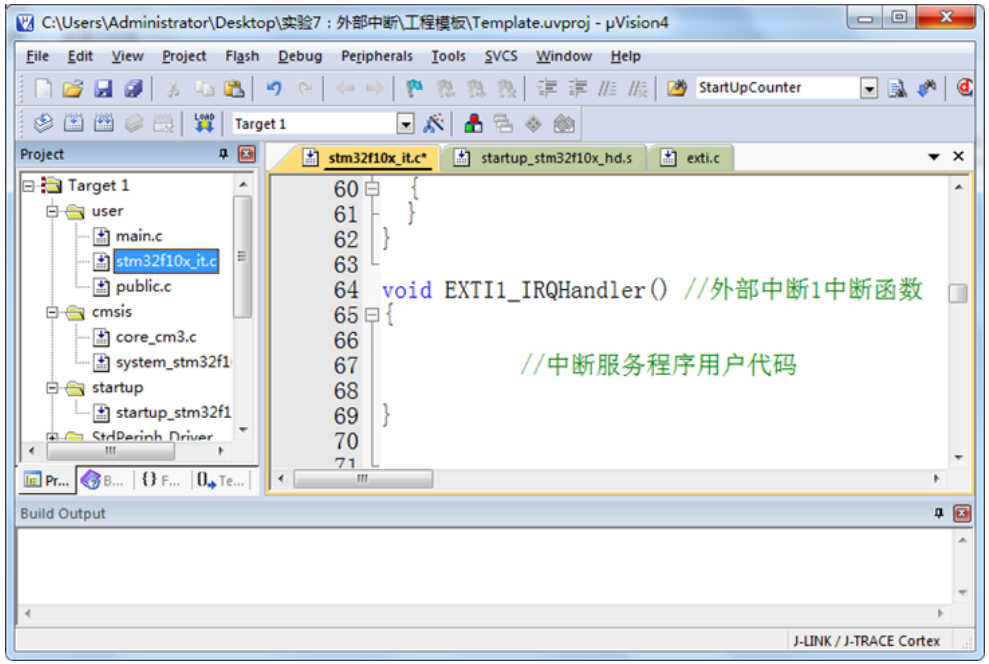
**系统存储区**（0x1FFFF000 ~ 0x1FFFF800，2KB）：存放串口下载程序，当系统上电后，根据用户设定的启动配置，将Flash或系统存储区映射到启动空间，执行用户程序或串口下载程序。

# STM32F103中断服务函数

STM32所有的中断服务函数，在该微控制器所属产品系列的启动代码文件中都有预先定义。（STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\ startup\arm\startup_stm32f10x_hd.s）

用户开发自己的STM32F103应用时可在文件stm32f10x_it.c中使用C语言编写函数重新定义。

STM32定义的中断服务函数，在启动文件和stm32f10x_it.c中通常以PPP_IRQHanlder命名。



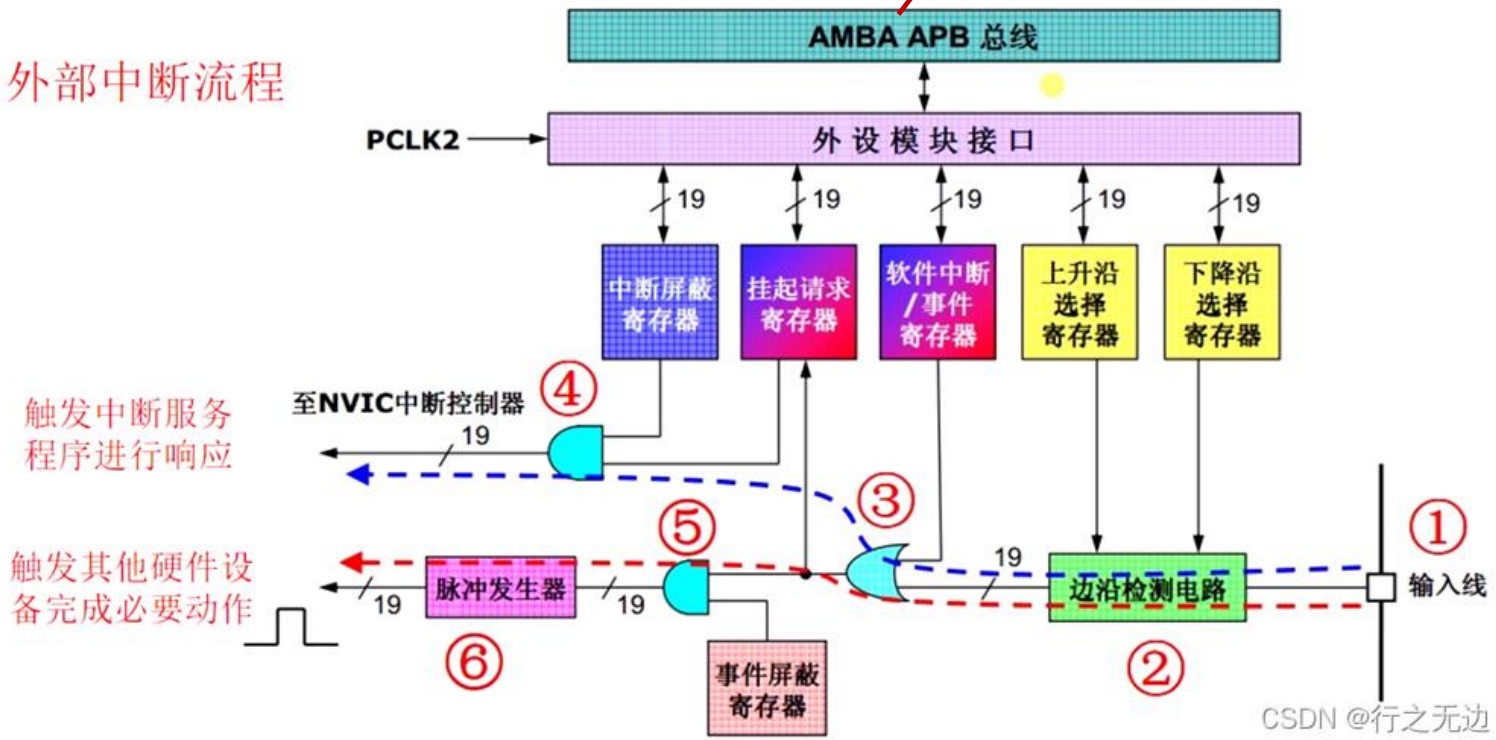

退出

APB外设模块接口是STM32微控制器每个功能模块都有的部分，CPU通过这样的接口访问各个功能模块。

需要注意的是，如果使用STM32引脚的外部中断/事件映射功能，**必须打开APB2总线上该引脚对应端口的时钟以及AFIO功能时钟**。

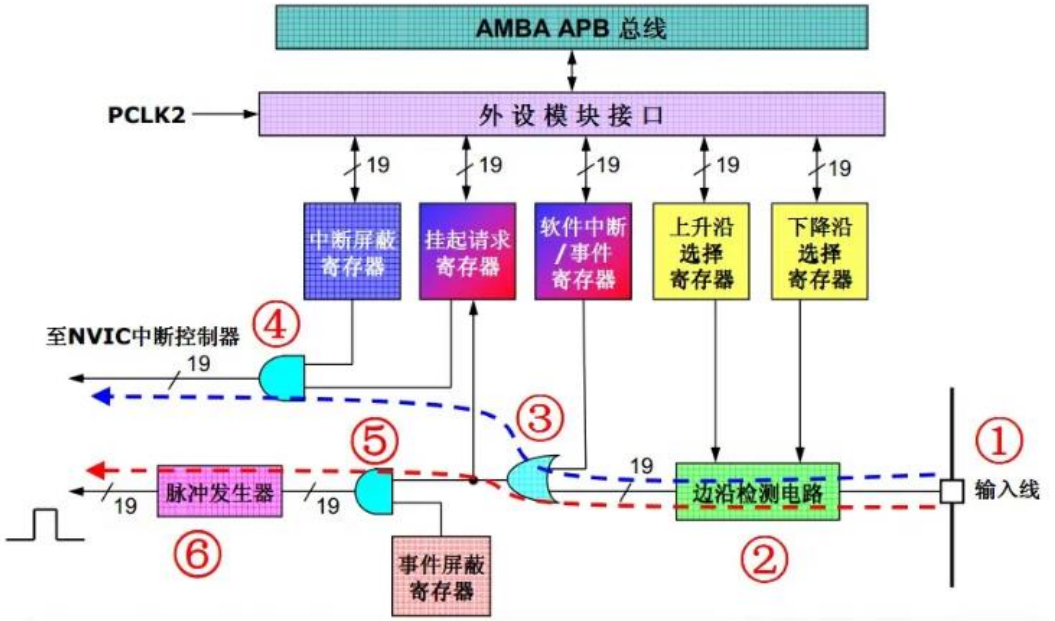
# 5.3 STM32F103外部中断控制器 EXTI

## 外部中断输入




上图是STM32一条外部中断线或外部事件线的示意图，图中信号线上划有一条斜线，旁边标注19字样，表示这样的线路共有19条。图中的蓝色虚线箭头，标出了外部中断信号的传输路径。红色虚线箭头，标出了外部事件信号的传输路径。

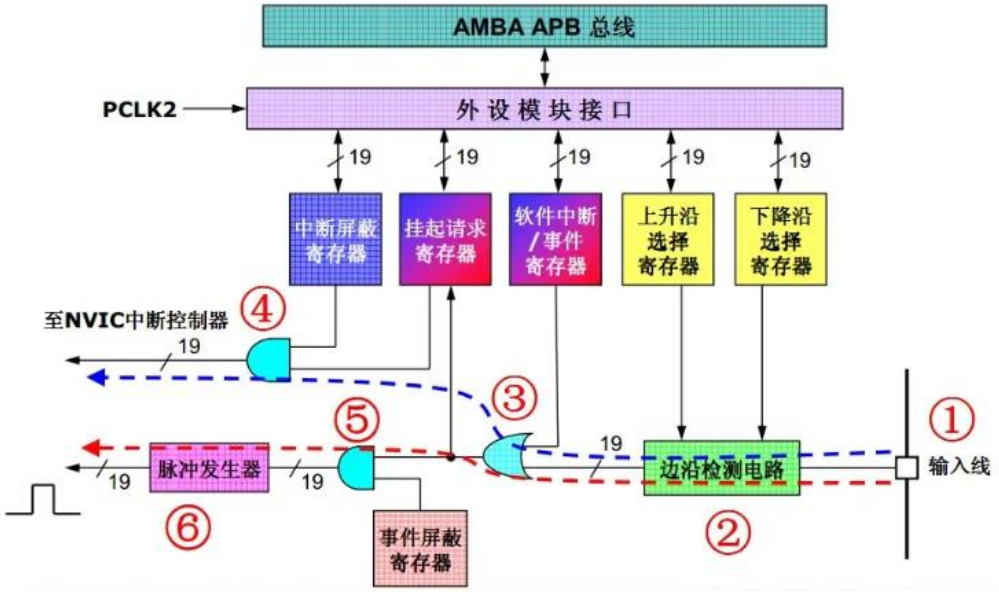
## 外部中断请求的产生和传输




- 外部信号从编号①的STM32F103微控制器引脚进入。
- 编号②是一个边沿检测电路，这个边沿检测电路受到上升沿触发选择寄存器和下降沿触发选择寄存器控制，用户可以配置这两个寄存器选择在哪一个边沿产生中断/事件。边沿检测电路以输入线①作为信号输入端，如果检测到有边沿跳变就输出有效信号1给编号③电路，否则输出无效信号0。
- 经过编号③的或门，这个或门的另一个输入是软件中断/事件寄存器。两个输入任意一个有效信号1就可以输出1流向编号④和编号⑥电路。
- 一个中断或事件请求信号经过编号③的或门后，进入挂起请求寄存器。到此之前，中断和事件的信号传输通路都是一致的，也就是说，挂起请求寄存器中记录了外部信号的电平变化。
- 外部请求信号进入编号④的与门，它的另外一个输入来自中断屏蔽寄存器(EXTI_IMR)。与门电路要求输入都为1才输出1，其结果是若EXTI_IMR设置为0，最终电路④输出的信号都为0；只有EXTI_IMR设置为1时，电路④输出的信号才由电路③的输出信号决定，这样就可以通过设置EXTI_IMR来控制是否产生中断。
- 信号最终输出到NVIC，实现系统中断事件控制。

## 外部中断请求的产生和传输

- 外部事件信号经过编号③的或门后，进入编号⑤的与门，这个与门的作用与编号④的与门类似，用于引入事件屏蔽寄存器的控制；最后由编号为⑥的脉冲发生器的一个跳变的信号转变为一个单脉冲，输出到芯片中的其它功能模块。
- 这个脉冲信号，就是产生事件的线路最终的产物，这个脉冲信号可以给其他外设电路使用，比如定时器 TIM、模拟数字转换器 ADC等等，这样的脉冲信号一般用来触发 TIM 或者 ADC开始转换。



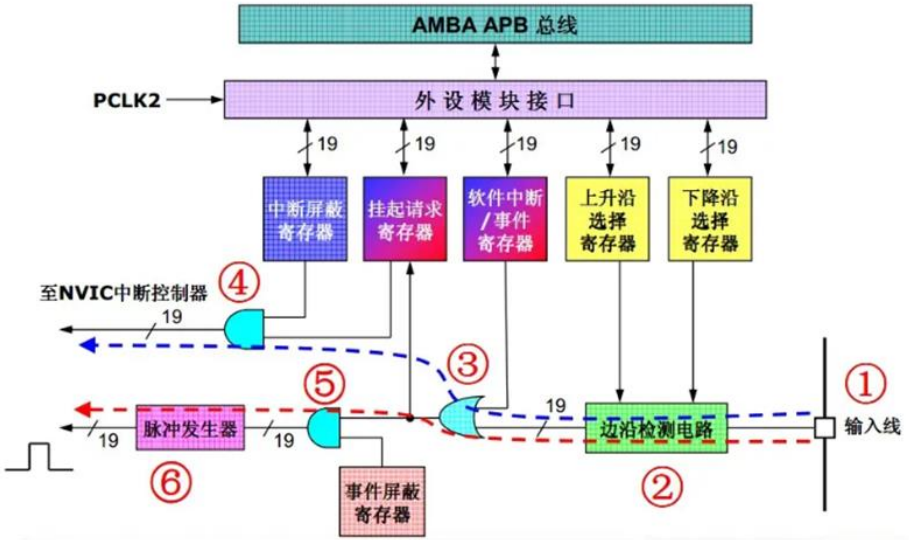

**小结：**从图中我们可以看到，从外部激励信号来看，中断和事件的产生源都可以是一样的，但可以采用不同的方式去处理。

中断是需要CPU参与的，产生中断线路目的是把输入信号输入到 NVIC，进一步会运行中断服务函数，实现功能，这是软件级的。

而产生事件线路目的就是传输一个脉冲信号给其他外设，进而由外设自动完成这个事件产生的结果，这是电路级别的信号传输，属于硬件级的。当然相应的联动部件需要先设置好，比如引起DMA操作、AD转换等。

## EXTI主要特性

- 每个外部中断/事件输入线都可以独立地配置它的触发事件（上升沿、下降沿或双边沿），并能够单独地被屏蔽。
- 每个外部中断都有专用的标志位（请求挂起寄存器），保持着它的中断请求。
- 可以将多达112个通用I/O引脚映射到16个外部中断/事件输入线上。




退出

## STM32的外部中断线

- STM32的每个IO都可以作为外部中断输入。
- STM32的中断控制器支持19个外部中断/事件请求：

  线0~15：对应外部IO口的输入中断。

  线16：连接到PVD输出。

  线17：连接到RTC闹钟事件。

  线18：连接到USB唤醒事件。

  每个外部中断线可以独立的配置触发方式（上升沿，下降沿或者双边沿触发），触发/屏蔽，专用的状态位。

STM32供IO使用的中断线只有16个，但是STM32F10x系列的IO口多达上百个，那么中断线是如何与IO口对应的呢？

如果将STM32F103的I/O引脚映射为EXTI的外部中断/事件输入线，必须将该引脚设置为输入模式。

## 引脚分组

### 引脚分组

① 尾号相同的引脚一组，接入1个外部中断线

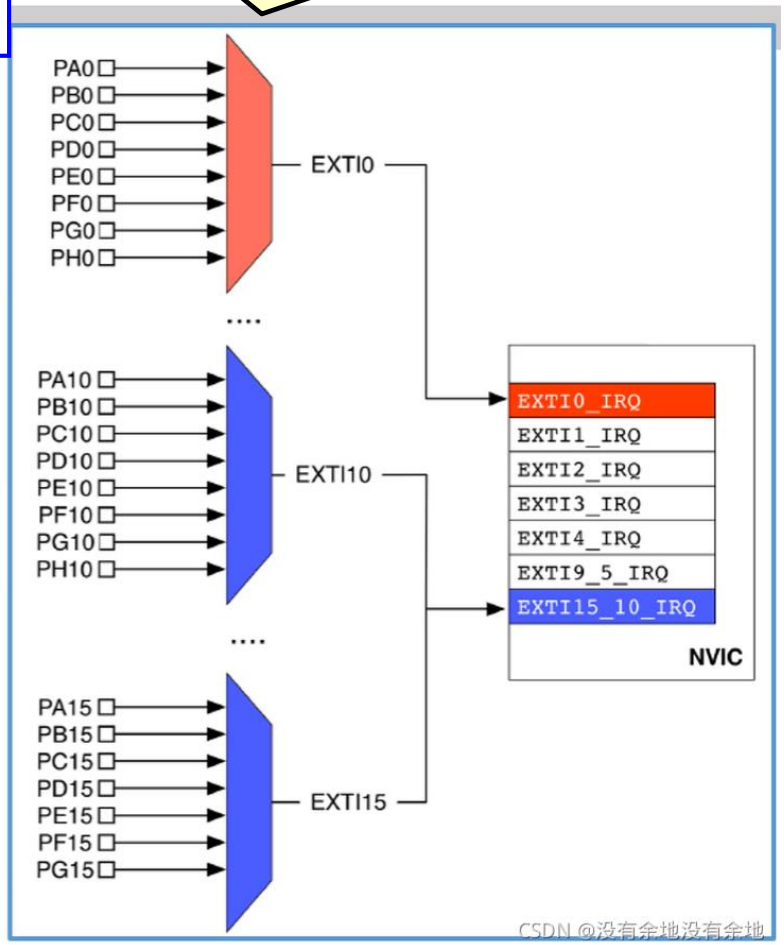
② 同组引脚只能有一个设置为外部中断功能

### 中断通道

① EXTI0~EXIT4分别具有独立的中断通道

② EXTI5~EXIT9共享同一个中断通道

③ EXTI10~EXIT15共享同一个中断通道



| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|---|---|---|---|---|---|
| 6 | 13 | 可设置 | EXTI0 | EXTI线0中断 | 0x0000_0058 |
| 7 | 14 | 可设置 | EXTI1 | EXTI线1中断 | 0x0000_005C |
| 8 | 15 | 可设置 | EXTI2 | EXTI线2中断 | 0x0000_0060 |
| 9 | 16 | 可设置 | EXTI3 | EXTI线3中断 | 0x0000_0064 |
| 10 | 17 | 可设置 | EXTI4 | EXTI线4中断 | 0x0000_0068 |
| 23 | 30 | 可设置 | EXTI9_5 | EXTI线[9:5]中断 | 0x0000_009C |
| 40 | 47 | 可设置 | EXTI15_10 | EXTI线[15:10]中断 | 0x0000_00E0 |

从表中可以看出，IO口外部中断在中断向量表中只分配了7个中断向量，也就是只能使用7个中断服务函数。其中，外部中断线 5~9 分配一个中断向量，共用一个服务函数；外部中断线 10~15 分配一个中断向量，共用一个中断服务函数。

退出

# 5.4 STM32 中断控制库函数

## 中断控制寄存器

- 中断设置允许寄存器（NVIC_ISER）
- 中断清除允许寄存器（NVIC_ICER）
- 中断设置挂起寄存器（NVIC_ISPR）
- 中断清除挂起寄存器（NVIC_ICPR）
- 中断状态寄存器（NVIC_IABR）
- 中断优先级由中断优先级寄存器组IPR(Interrupt Priority Registers)控制。这个寄存器组包含15个32位的寄存器，一个可屏蔽中断占用8Bit，因此一个寄存器可以控制4个可屏蔽中断，一共15*4=60。

## 5.4.1 嵌套向量中断控制器（NVIC）库函数

### 1. 函数NVIC_DeInit

| | |
|---|---|
| 函数名 | NVIC_DeInit |
| 函数原形 | void NVIC_DeInit(void) |
| 功能描述 | 将外设 NVIC 寄存器重设为缺省值 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

## 2. 函数NVIC_PriorityGroupConfig

| | |
|---|---|
| 函数名 | NVIC_PriorityGroupConfig |
| 函数原形 | void NVIC_PriorityGroupConfig(u32 NVIC_PriorityGroup) |
| 功能描述 | 设置优先级分组：抢占优先级和响应优先级 |
| 输入参数 | NVIC_PriorityGroup：优先级分组位长度 参阅 Section：NVIC_PriorityGroup 查阅更多该参数允许取值范围 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 优先级分组只能设置一次 |
| 被调用函数 | 无 |

### NVIC_PriorityGroup 值

| NVIC_PriorityGroup | 描述 |
|---|---|
| NVIC_PriorityGroup_0 | 抢占优先级 0 位，响应优先级 4 位 |
| NVIC_PriorityGroup_1 | 抢占优先级 1 位，响应优先级 3 位 |
| NVIC_PriorityGroup_2 | 抢占优先级 2 位，响应优先级 2 位 |
| NVIC_PriorityGroup_3 | 抢占优先级 3 位，响应优先级 1 位 |
| NVIC_PriorityGroup_4 | 抢占优先级 4 位，响应优先级 0 位 |

退出

## 3. 函数NVIC_Init

| | |
|---|---|
| 函数名 | NVIC_Init |
| 函数原形 | void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct) |
| 功能描述 | 根据 NVIC_InitStruct 中指定的参数初始化外设 NVIC 寄存器 |
| 输入参数 | NVIC_InitStruct：指向结构 NVIC_InitTypeDef 的指针，包含了外设 GPIO 的配置信息 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

```
typedef struct
{
    uint8_t NVIC_IRQChannel;     // 指定IRQ通道
    uint8_t NVIC_IRQChannelPreemptionPriority; // 设置指定通道的抢占优先级
    uint8_t NVIC_IRQChannelSubPriority;  // 设置指定通道的响应优先级
    FunctionalState NVIC_IRQChannelCmd;  // 指定通道使能或失能
} NVIC_InitTypeDef;
```

退出

## NVIC_IRQChannel:

| NVIC_IRQChannel | 描述 |
|---|---|
| WWDG_IRQChannel | 窗口看门狗中断 |
| PVD_IRQChannel | PVD 通过 EXTI 探测中断 |
| TAMPER_IRQChannel | 篡改中断 |
| RTC_IRQChannel | RTC 全局中断 |
| FlashItf_IRQChannel | FLASH 全局中断 |
| RCC_IRQChannel | RCC 全局中断 |
| EXTI0_IRQChannel | 外部中断线 0 中断 |
| EXTI1_IRQChannel | 外部中断线 1 中断 |
| EXTI2_IRQChannel | 外部中断线 2 中断 |
| EXTI3_IRQChannel | 外部中断线 3 中断 |
| EXTI4_IRQChannel | 外部中断线 4 中断 |
| DMAChannel1_IRQChannel | DMA 通道 1 中断 |
| DMAChannel2_IRQChannel | DMA 通道 2 中断 |
| DMAChannel3_IRQChannel | DMA 通道 3 中断 |
| DMAChannel4_IRQChannel | DMA 通道 4 中断 |
| DMAChannel5_IRQChannel | DMA 通道 5 中断 |
| DMAChannel6_IRQChannel | DMA 通道 6 中断 |
| DMAChannel7_IRQChannel | DMA 通道 7 中断 |
| ADC_IRQChannel | ADC 全局中断 |
| USB_HP_CANTX_IRQChannel | USB 高优先级或者 CAN 发送中断 |
| USB_LP_CAN_RX0_IRQChannel | USB 低优先级或者 CAN 接收 0 中断 |
| CAN_RX1_IRQChannel | CAN 接收 1 中断 |
| CAN_SCE_IRQChannel | CAN SCE 中断 |

## NVIC_IRQChannel:

| | |
|---|---|
| EXTI9_5_IRQChannel | 外部中断线 9-5 中断 |
| TIM1_BRK_IRQChannel | TIM1 暂停中断 |
| TIM1_UP_IRQChannel | TIM1 刷新中断 |
| TIM1_TRG_COM_IRQChannel | TIM1 触发和通讯中断 |
| TIM1_CC_IRQChannel | TIM1 捕获比较中断 |
| TIM2_IRQChannel | TIM2 全局中断 |
| TIM3_IRQChannel | TIM3 全局中断 |
| TIM4_IRQChannel | TIM4 全局中断 |
| I2C1_EV_IRQChannel | I2C1 事件中断 |
| I2C1_ER_IRQChannel | I2C1 错误中断 |
| I2C2_EV_IRQChannel | I2C2 事件中断 |
| I2C2_ER_IRQChannel | I2C2 错误中断 |
| SPI1_IRQChannel | SPI1 全局中断 |
| SPI2_IRQChannel | SPI2 全局中断 |
| USART1_IRQChannel | USART1 全局中断 |
| USART2_IRQChannel | USART2 全局中断 |
| USART3_IRQChannel | USART3 全局中断 |
| EXTI15_10_IRQChannel | 外部中断线 15-10 中断 |
| RTCAlarm_IRQChannel | RTC 闹钟通过 EXTI 线中断 |
| USBWakeUp_IRQChannel | USB 通过 EXTI 线从悬挂唤醒中断 |

## 5.4.2 STM32 外部中断 EXTI 相关库函数

### 1. 函数EXTI_DeInit

| | |
|---|---|
| 函数名 | EXTI_DeInit |
| 函数原形 | EXTI_DeInit(void) |
| 功能描述 | 将外设 EXTI 寄存器重设为缺省值 |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

## 2. 函数EXTI_Init

| | |
|---|---|
| 函数名 | EXTI_Init |
| 函数原形 | void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct) |
| 功能描述 | 根据 EXTI_InitStruct 中指定的参数初始化外设 EXTI 寄存器 |
| 输入参数 | EXTI_InitStruct：指向结构 EXTI_InitTypeDef 的指针，包含了外设 EXTI 的配置信息，参阅 Section： EXTI_InitTypeDef 查阅更多该参数允许取值范围 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

EXTI_InitTypeDef 定义于文件“stm32f10x_exti.h”：参考课本P71

```
typedef struct {
    u32 EXTI_Line;    // 选择待使能或失能的外部线路：EXTI-Line0 ~ EXTI-Line18
    EXTIMode_TypeDef EXTI_Mode; // 被使能线路模式：事件请求或中断请求
    EXTIrigger_TypeDef EXTI_Trigger; //被使能线路的触发边沿
    FunctionalState EXTI_LineCmd;    // 新状态 ENABLE 或 DISABLE
} EXTI_InitTypeDef;
```

退出

除了读取中断标志位，还查看EXT_IMR寄存器是否对该中断进行屏蔽，在中断挂起&没有屏蔽的情况下就会响应中断。

## 3. 函数EXTI_GetITStatus

| 函数名 | EXTI_GetITStatus |
|---|---|
| 函数原形 | ITStatus EXTI_GetITStatus(u32 EXTI_Line) |
| 功能描述 | 检查指定的 EXTI 线路触发请求发生与否 |
| 输入参数 | EXTI_Line：待检查 EXTI 线路的挂起位<br>参阅 Section： EXTI_Line 查阅更多该参数允许取值范围 |
| 输出参数 | 无 |
| 返回值 | EXTI_Line 的新状态（SET 或者 RESET） |
| 先决条件 | 无 |
| 被调用函数 | 无 |

只读取中断标志位的状态，但是不一定会响应中断（EXT_IMR寄存器对该中断进行屏蔽）

## 4. 函数EXTI_GetFlagStatus

| 函数名 | EXTI_GetFlagStatus |
|---|---|
| 函数原形 | FlagStatus EXTI_GetFlagStatus(u32 EXTI_Line) |
| 功能描述 | 检查指定的 EXTI 线路标志位设置与否 |
| 输入参数 | EXTI_Line：待检查的 EXTI 线路标志位 |
| 输出参数 | 无 |
| 返回值 | EXTI_Line 的新状态（SET 或者 RESET） |
| 先决条件 | 无 |
| 被调用函数 | 无 |

## 5. 函数EXTI_ClearFlag

二者功能完全相同！

| | |
|---|---|
| 函数名 | EXTI_ClearFlag |
| 函数原形 | void EXTI_ClearFlag(u32 EXTI_Line) |
| 功能描述 | 清除 EXTI 线路挂起标志位 |
| 输入参数 | EXTI_Line：待清除标志位的 EXTI 线路 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

## 6. 函数EXTI_ClearITPendingBit

| | |
|---|---|
| 函数名 | EXTI_ClearITPendingBit |
| 函数原形 | void EXTI_ClearITPendingBit(u32 EXTI_Line) |
| 功能描述 | 清除 EXTI 线路挂起位 |
| 输入参数 | EXTI_Line：待清除 EXTI 线路的挂起位 |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

退出

## 5.4.3 EXTI中断线GPIO引脚映射库函数

### GPIO_EXTILineConfig

| | |
|---|---|
| 函数名 | GPIO_EXTILineConfig |
| 函数原形 | void GPIO_EXTILineConfig(u8 GPIO_PortSource, u8 GPIO_PinSource) |
| 功能描述 | 选择 GPIO 管脚用作外部中断线路 |
| 输入参数 1 | GPIO_PortSource: 选择用作外部中断线源的 GPIO 端口<br>参阅 Section： GPIO_PortSource 查阅更多该参数允许取值范围 |
| 输入参数 2 | GPIO_PinSource：待设置的外部中断线路<br>该参数可以取 GPIO_PinSourcex(x 可以是 0-15) |
| 输出参数 | 无 |
| 返回值 | 无 |
| 先决条件 | 无 |
| 被调用函数 | 无 |

退出

# 5.5 外部中断使用流程

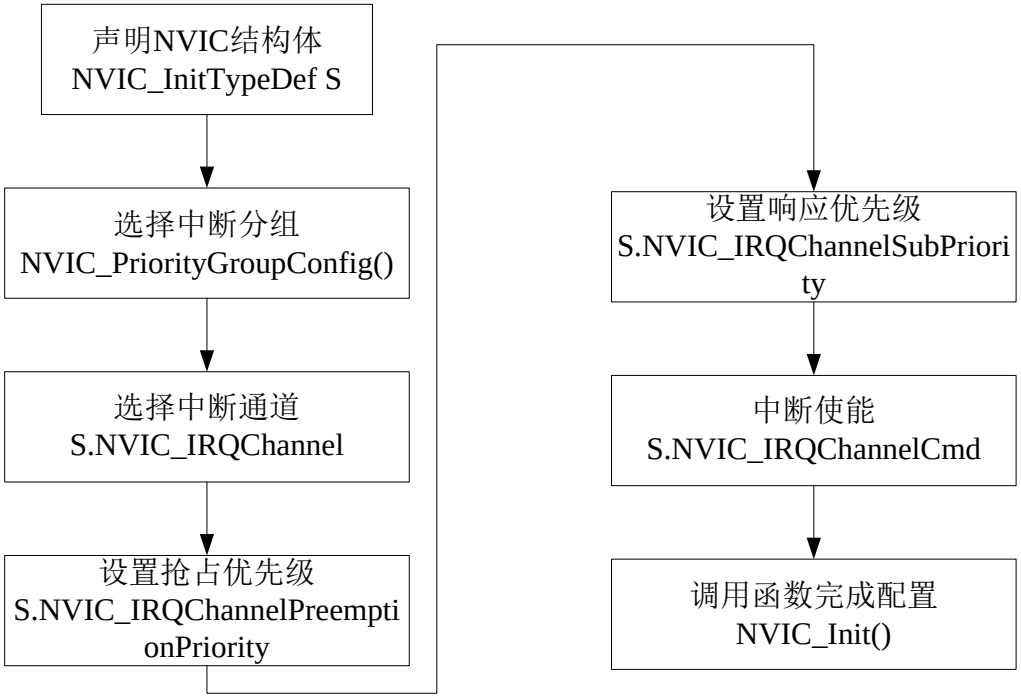
## 1. NVIC设置

(1) 根据需要对中断优先级进行分组，确定抢占优先级和响应优先级的个数。

(2) 选择中断通道。

(3) 根据系统要求设置中断优先级，包括抢占优先级和响应优先级。

(4) 使能相应的中断，完成NVIC配置。



中断通道的设置见 P68 表5-4

例：

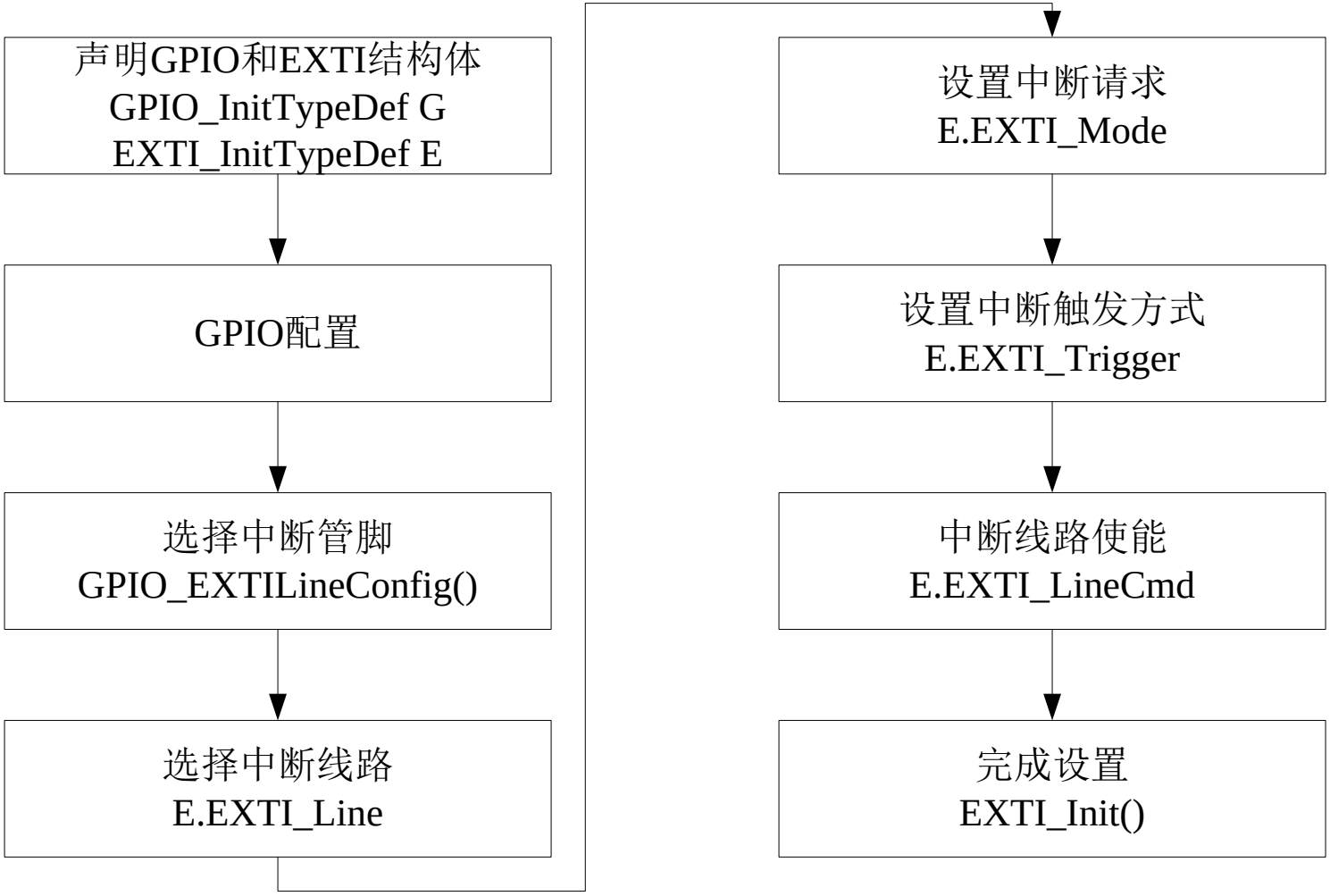
```
NVIC_InitTypeDef NVIC_InitStructure;

NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;

NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;

NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

NVIC_Init (&NVIC_InitStructure);
```

## 2. 中断端口配置

(1) 对IO引脚进行配置，使能引脚，具体方法参考第4章，如果使用了复用功能需要打开复用时钟。

(2) 对外部中断方式进行配置，包括中断线路设置、中断或事件选择、触发方式设置、使能中断线，完成设置。

其中中断线路EXTI_Line0~ EXTI_Line15分别对应EXTI0~EXTI15，即每个端口的16个引脚。EXTI_Line16~ EXTI_Line18分别对应PVD输出事件、RTC闹钟事件和USB唤醒事件。


声明GPIO和EXTI结构体
GPIO_InitTypeDef G
EXTI_InitTypeDef E
GPIO配置
选择中断管脚
GPIO_EXTILineConfig()
选择中断线路
E.EXTI_Line
设置中断请求
E.EXTI_Mode
设置中断触发方式
E.EXTI_Trigger
中断线路使能
E.EXTI_LineCmd
完成设置
EXTI_Init()

## EXTI的初始化

```
EXTI_InitTypeDef EXTI_InitStructure;

EXTI_InitStructure.EXTI_Line = EXTI_Line12 | EXTI_Line14;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

## 3. 中断处理

### 1） 中断请求和优先级

如果系统中存在多个中断源，处理器要先对当前中断的优先级进行判断。多个中断请求同时到达时，先响应优先级高的中断。

如果它们的抢占优先级相同，则先处理响应优先级高的中断。

## 2）中断响应

在中断时间产生后，处理器响应中断要满足下列条件：

(1) 无同级或高级中断正在服务。

(2) 当前指令周期结束，如果查询中断请求的机器周期不是当前指令的最后一个周期，则无法执行当前中断请求。

(3) 若处理器正在执行系统指令，则需要执行到当前指令及下一条指令才能响应中断请求。

如果中断发生，且处理器满足上述条件，系统将**按照下面步骤执行相应中断请求**，具体如下：

(1) 置位中断优先级有效触发器，即关闭同级和低级中断。

(2) 调用入口地址，断点入栈。

(3) 进入中断服务程序。

**中断入口：P74**

### 3）中断处理

中断处理就是执行中断服务程序，从中断入口地址开始执行直到返回指令为止，具体的执行过程包括三部分内容：

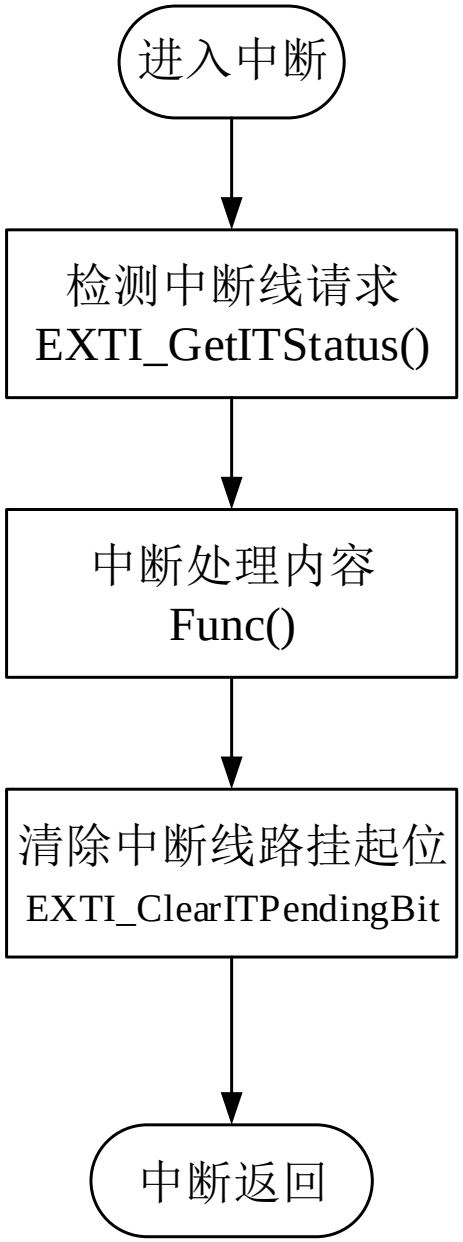
(1) 中断现场保护。

(2) 处理中断源的请求。

(3) 恢复中断现场。

### 4）中断返回

中断返回是指中断服务完成后，处理器返回到原来程序断点继续执行原来程序。



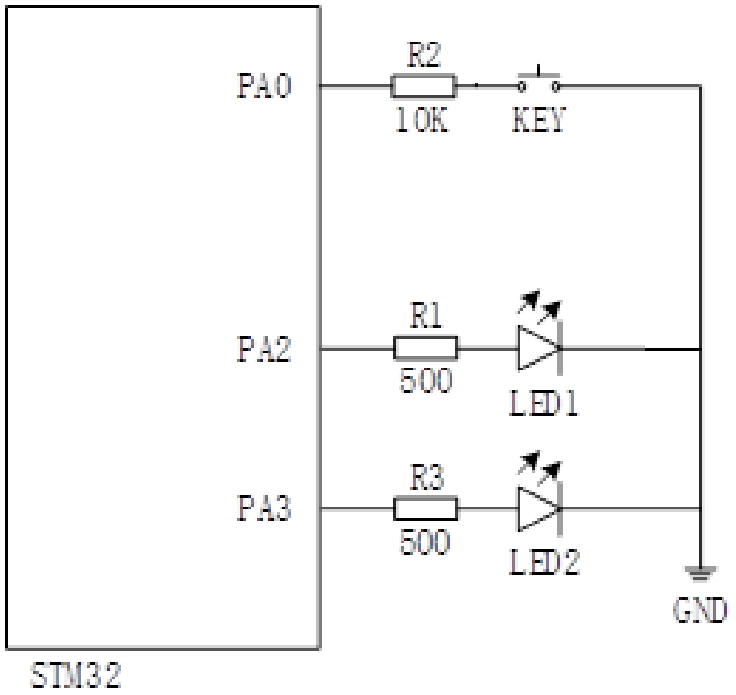

如果进入中断不清除标志位，那么这一中断服务程序结束后，由于标志位还是置位的并且中断是允许的，那么还会再次进入该中断，就会发生一直在执行中断程序的情况。

例如外部中断0的中断服务程序：

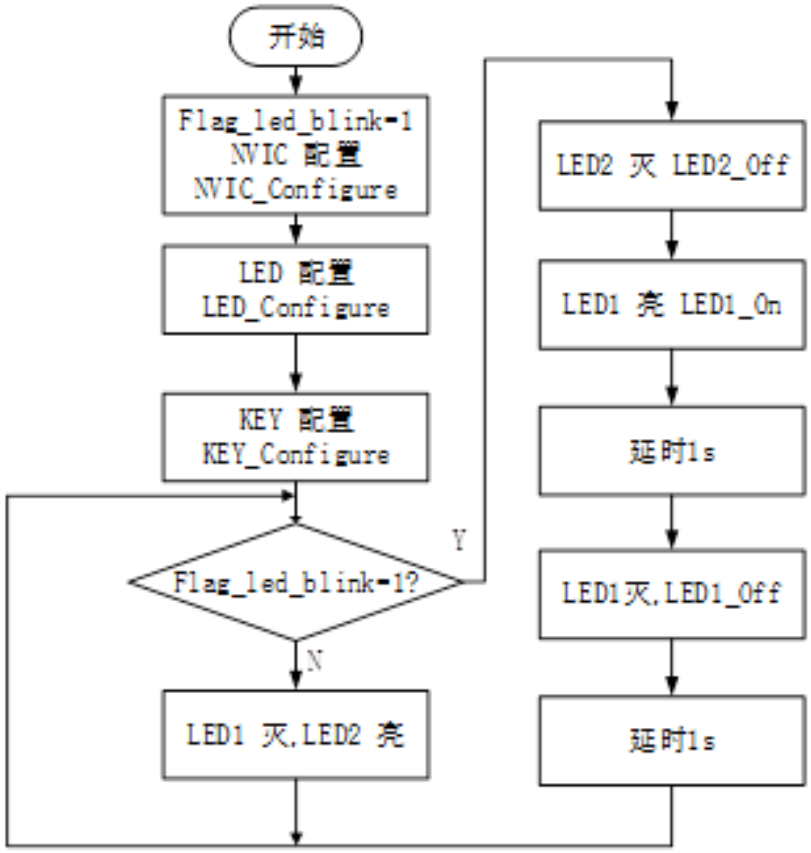
```
void EXTI0_IRQHandler (void)
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET) //确保是否产生了EXTI Line中断
    {
        /*中断功能*/
        ……
        ……
        EXTI_ClearITPendingBit(EXTI_Line0);    //清除中断标志位
    }
}
```

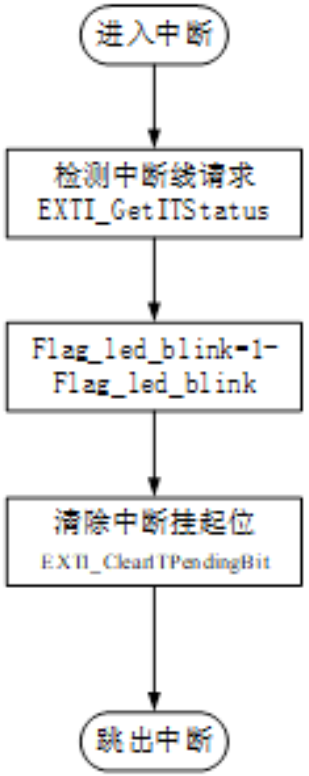
# 5.6 STM32外部中断应用设计实例

利用按键中断实现LED1和LED2状态控制，可用于模拟诸如手术室工作状态指示等应用。电路原理图如下。要求：① 无手术时LED1（绿灯）间隔1s闪烁，LED2（红灯）灭；② 医生进入手术室，可以按下按键，使LED1灭，LED2稳定点亮；③ 手术结束，再次按下按键，恢复LED1闪烁且LED2灭。



## 主程序和中断程序流程

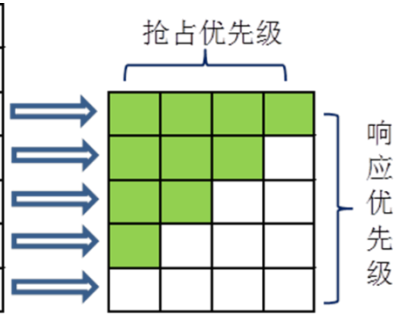

开始
Flag_led_blink=1
NVIC 配置
NVIC_Configure
LED 配置
LED_Configure
KEY 配置
KEY_Configure
Flag_led_blink=1?
Y
N
LED1 灭,LED2 亮
LED2 灭 LED2_Off
LED1 亮 LED1_On
延时1s
LED1灭,LED1_Off
延时1s


进入中断
检测中断线请求
EXTI_GetITStatus
Flag_led_blink=1-
Flag_led_blink
清除中断挂起位
EXTI_ClearITPendingBit
跳出中断

```
uint8_t flag_led_blink =1;
int  main (void)
{
    NVIC_Configure () ;
    KEY_Configure () ;
    LED_Configure () ;
    while (1) {
        if (flag_led_blink) {
            LED2_Off ();
            LED1_On ();
            delay ();
            LED1_Off ();
            delay ();
        }
        else {
            LED1_Off ();
            LED2_On ();
        }
    }
}
```

```
void  NVIC_Configure (void)
{
    NVIC_InitTypeDef NVIC_InitStructure;

    /* NVIC优先级分组 抢占式优先2位四级 响应式优先级2位四级 */
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

     NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn; //指定中断通道

     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; //配置抢占式优先级

     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;  //配置响应式优先级

     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;  //中断使能

     NVIC_Init (&NVIC_InitStructure);
}
```

| 优先级组别 | 抢占式优先级 | | 响应式优先级 | |
|---|---|---|---|---|
| | 位数 | 级数 | 位数 | 级数 |
| 4组 | 4 | 16 | 0 | 0 |
| 3组 | 3 | 8 | 1 | 2 |
| 2组 | 2 | 4 | 2 | 4 |
| 1组 | 1 | 2 | 3 | 8 |
| 0组 | 0 | 0 | 4 | 16 |



```
void  KEY_Configure (void)  {
    GPIO_InitTypeDef GPIO_InitStructure; //声明一个结构体变量，用来初始化GPIO
    EXTI_InitTypeDef EXTI_InitStructure;

    RCC_APB2PeriphClockCmd (RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO , ENABLE);
     // 开启GPIO时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init (GPIOA, &GPIO_InitStructure);

 /* 选择GPIO管脚用作外部中断线路  此处一定要记住给端口管脚加上中断外部线路*/
    GPIO_EXTILineConfig (GPIO_PortSourceGPIOA, GPIO_PinSource0);

 /* 设置外部中断的模式 PA0 中断初始化 */
     EXTI_InitStructure.EXTI_Line = EXTI_Line0;
     EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
     EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising_Falling;
     EXTI_InitStructure.EXTI_LineCmd = ENABLE;
     EXTI_Init(&EXTI_InitStructure);
}
```

```
void  LED_Configure (void)  {
    GPIO_InitTypeDef GPIO_InitStructure; //声明一个结构体变量，用来初始化GPIO
    RCC_APB2PeriphClockCmd (RCC_APB2Periph_GPIOA, ENABLE);      // 开启GPIO时钟
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init (GPIOA, &GPIO_InitStructure);
    LED1_Off ();
    LED2_Off ();
}
```

```
void  LED1_On (void)  {
    GPIO_SetBits (GPIOA, GPIO_Pin_2);
}

void  LED1_Off (void)  {
    GPIO_ResetBits (GPIOA, GPIO_Pin_2);
}


void  LED2_On (void)  {
    GPIO_SetBits (GPIOA, GPIO_Pin_3);
}

void  LED2_Off (void)  {
    GPIO_ResetBits (GPIOA, GPIO_Pin_3);
}
```

```
/* 函数名必须和系统预定义的一样，不能更改。预定义见 startup_stm32f10x_hd.s。*/
void  EXTI0_IRQHandler (void)
{
     if ( EXTI_GetITStatus (EXTI_Line0) != RESET ) {

          flag_led_blink = 1- flag_led_blink ;

          EXTI_ClearITPendingBit (EXTI_Line0) ;
         /*   清除EXTI线路挂起标志位, 与函数 EXTI_ClearFlag(EXTI_Line0) 完全相同  */
      }
}
```

**思考:**

**1、 if ( EXTI_GetITStatus (EXTI_Line0) != RESET ) 的作用？**

**2、 函数 EXTI0_IRQHandler () 何时执行？**

```c
void EXTI3_IRQHandler()
{

    /*...想要中断处理的代码...*/
    //将中断标志位清为0
    EXTI_ClearITPendingBit(EXTI_Line3);
}
```

```c
void EXTI9_5_IRQHandler(void)
{
    //先判断 EXTi5  EXTI6的哪个标志位置为1
    if(EXTI_GetITStatus(EXTI_Line5) == 1)
    {
        //能进来这里表示是EXTI5触发的中断

        EXTI_ClearITPendingBit(EXTI_Line5);
    }
    if(EXTI_GetITStatus(EXTI_Line6) == 1)
    {
        //能进来这里表示是EXTI6触发的中断

        EXTI_ClearITPendingBit(EXTI_Line6);

    }

}
```

作业：

1、完成课本P49 4.4.2 实例工程的开发；

2、完成课本P75 5.6.1 实例工程的开发；

在实验课上调试通过上述两个程序。

* 3、思考C语言关键字 static，const，volatile 的用法。