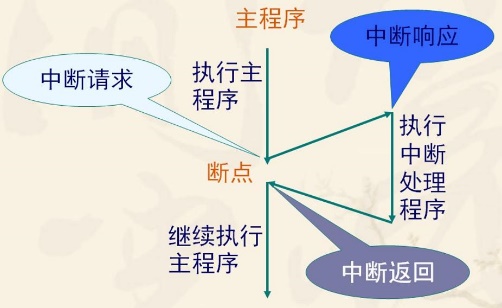
# 中断和异常

## 中断和异常基本概念

**CPU的中断和异常**

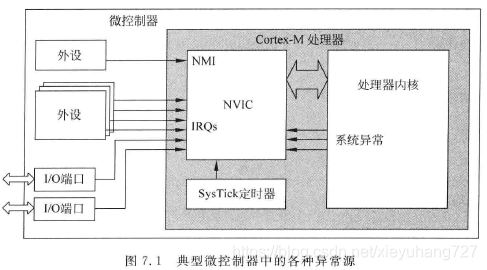
程序运行过程中，发生了不可预知的事件，需要暂停当前的任务立刻处理

* 查看按键是否按下，查看定时器是否超时
* 除法除数为零、系统故障



CPU的中断和异常由嵌套向量中断控制器NVIC管理：

* 灵活的中断和异常管理：禁止、使能、软件触发，清除、电平触发、脉冲触发
* 支持嵌套：优先级管理与抢占
* 向量化入口：自动定位中断
* 中断屏蔽

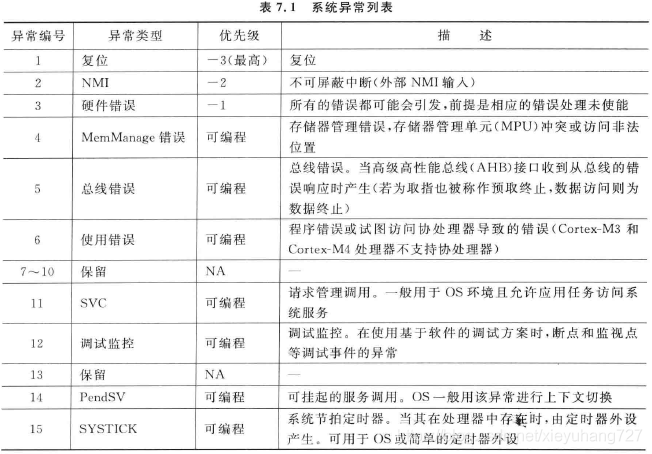


Cortex-M架构支持多种异常和外部中断，编号1~15为系统异常，16及以上的则为中断输入。

在ARM中，一般由处理器系统内部引发的事件，称作异常，如重启Reset，硬件错误等。由处理器外部，也就是外设引发的事件，称作中断，如GPIO模块等。

包括中断在内的大多数的异常的优先级都是可编程的，一些系统异常具有固定的优先级。

系统异常列表：



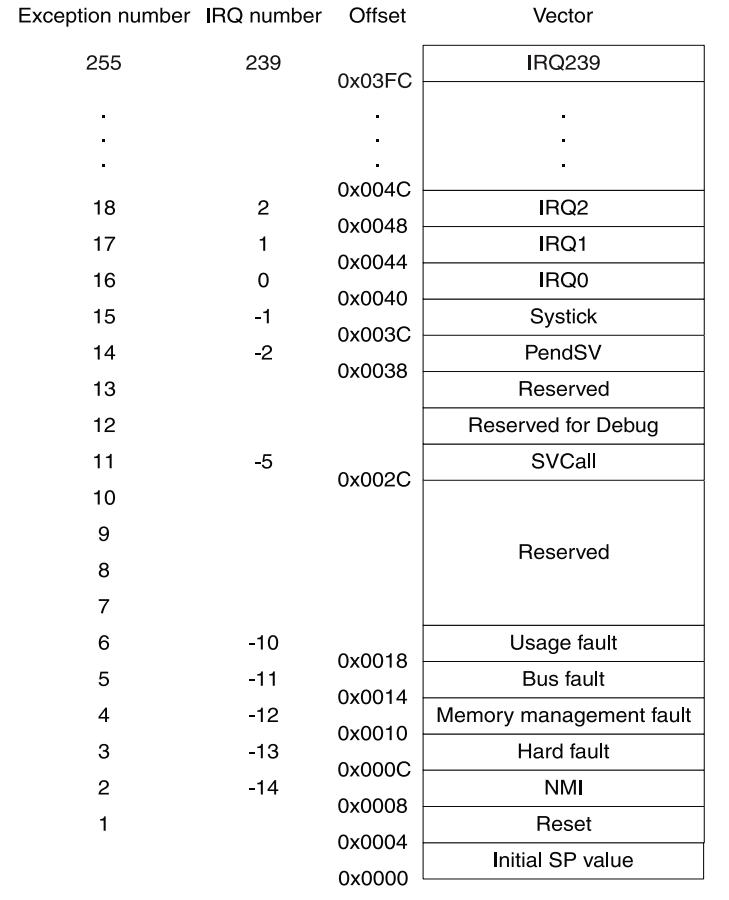
外部中断列表：



以上异常和中断事件中，优先级可编程的事件，可以被屏蔽，优先级不可编程的事件，系统必须响应。

异常编号与中断编号不同，中断编号=异常编号-16

STM32F401MCU的中断与异常列表：

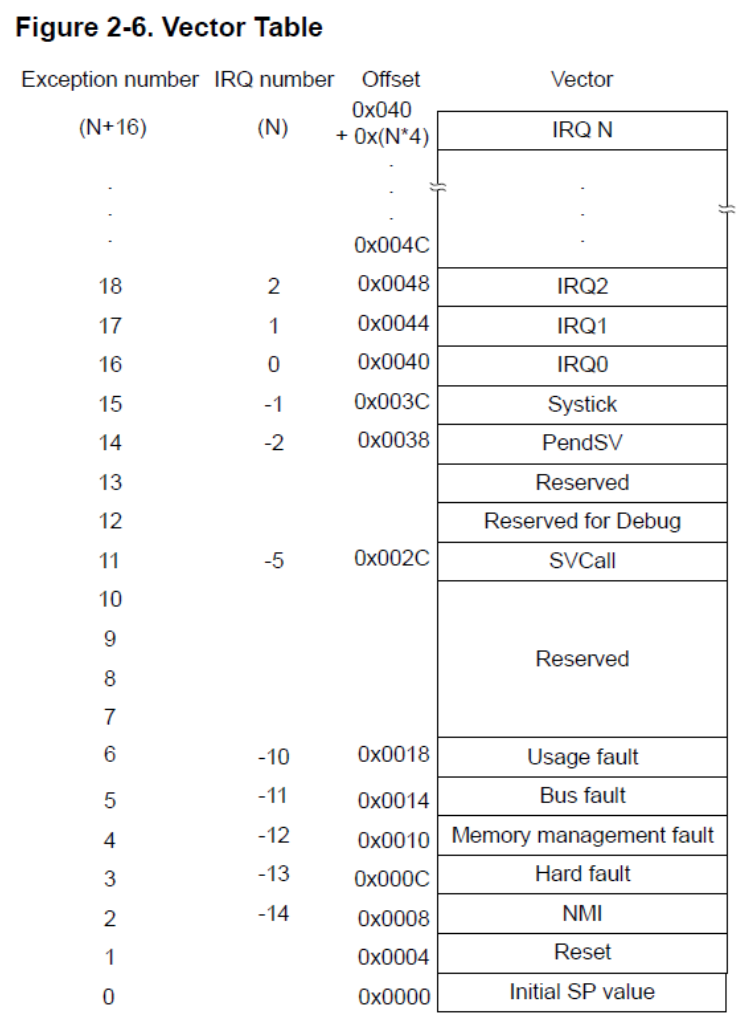


## 中断的响应流程

中断的响应流程



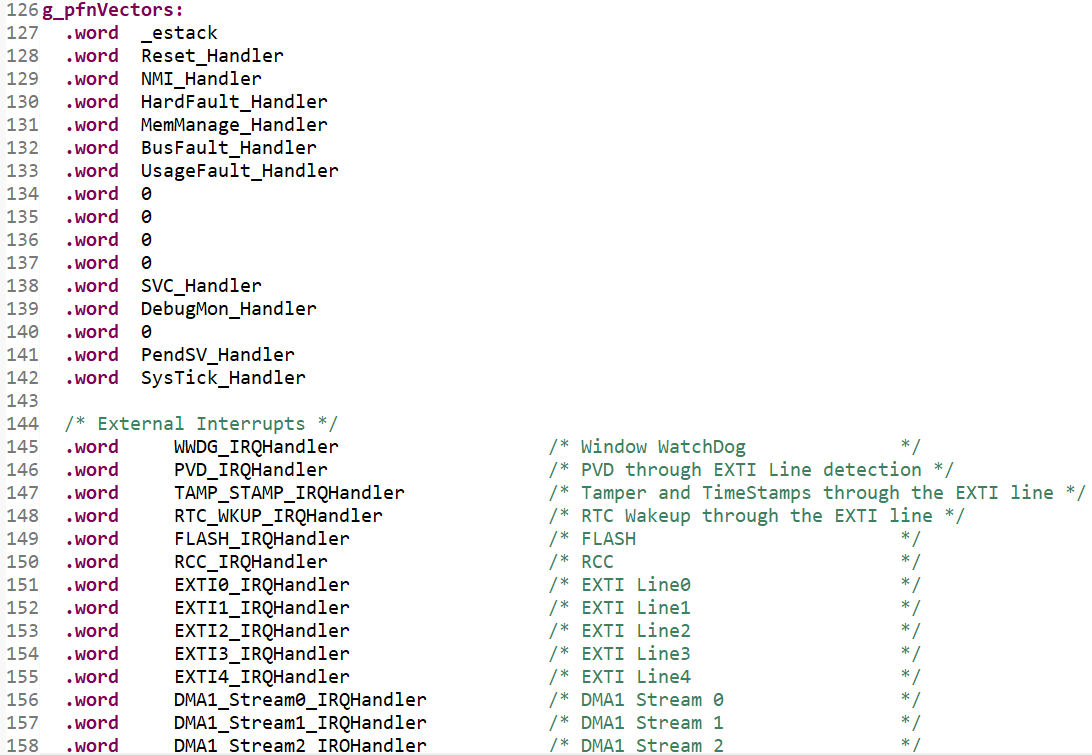
1. **接受异常请求**：处理器能够接受异常并执行异常的条件有
   1. 处理器正在运行
   2. 异常处于使能状态
   3. 异常的优先级高于当前等级，如果新异常的优先级不高于当前异常的等级，那么异常进入挂起状态，当前异常执行相应完毕后，才会接收新异常的请求。
   4. 异常未被屏蔽
2. **异常进入**：当系统进入异常时，NVIC会做一系列的操作，保证处理完中断处理函数（ISR）之后，异常发送之前的程序继续执行
   1. CPU内核寄存器和返回地址被压入当前使用的栈
   2. 取出中断向量



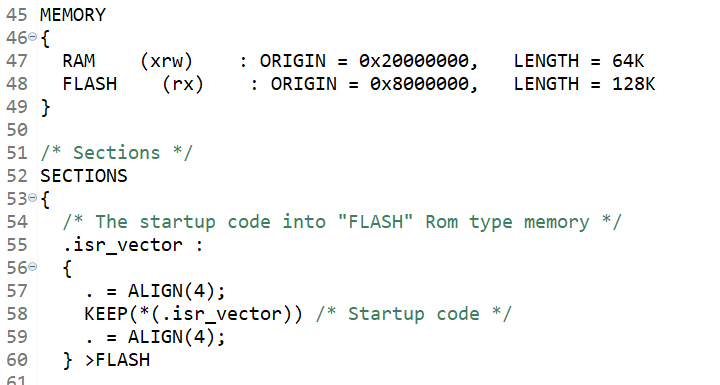
* 1. 取出待执行异常处理的指令
  2. 更新NVIC寄存器和内核寄存器，包括挂起状态寄存器（这个需要软件来清掉）和异常的活跃状态，处理器内核寄存器中的寄存器包括程序状态寄存器（PSR），链接寄存器（LR），程序计数器（PC）以及栈指针（SP）。
* **中断向量表**

上电后中断向量表存在0x00000000位置，其第一个元素为栈指针，第二个元素是复位向量。可以通过修改VTOR寄存器修改中断向量表的位置。

在启动文件startup\_stm32f401rbtx.s中定义了中断向量表**g\_pfnVectors**：



在STM32F401RBTX\_FLASH.ld文件中，规定了**g\_pfnVectors**的存储位置：



1. **执行异常处理**：执行异常处理函数
   1. 执行相应的中断服务函数
   2. 如果有更高优先级的异常产生，则接受新的异常，当前程序被抢占（异常嵌套）
   3. 如果新的异常的优先级等于或者低于当前的异常，则等到当前异常处理完成后才会得到处理
2. **异常返回**：在异常处理的结尾，程序代码执行的返回会引起EXC\_RETURN数值被加载到程序计数器（PC），并触发异常返回机制。
   1. CPU内核寄存器出栈
   2. NVIC寄存器更新
   3. CPU取出之前被中断的程序指令，程序恢复执行

中断响应流程示意图，压栈和取向量可以同时进行：



嵌套中断的处理，当两个中断同时发生时，优先服务优先级高的中断



尾链：

若某个异常产生时处理器正在处理另一个具有相同或更高优先级的异常，该异常会进入挂起状态。在处理器执行完当前的异常处理后，他可以继续执行挂起的异常和中断请求。这个紧接着执行中断的过程不会重复的压栈出栈，这样就大大的节约了时间，对于无等待状态的存储器系统，末尾连锁的中断等待时间仅为6个时钟周期。末尾连锁优化还给处理器带来了更佳的能耗效率，这是因为栈存储器访问的总数少了，而每次存储器传输都会消耗能量。



延迟到达：

当异常产生时，处理器会接受异常请求并开始压栈操作。若在压栈操作期间产生了另外一个更高优先级的异常，则更高优先级的后来的异常会首先得到服务。

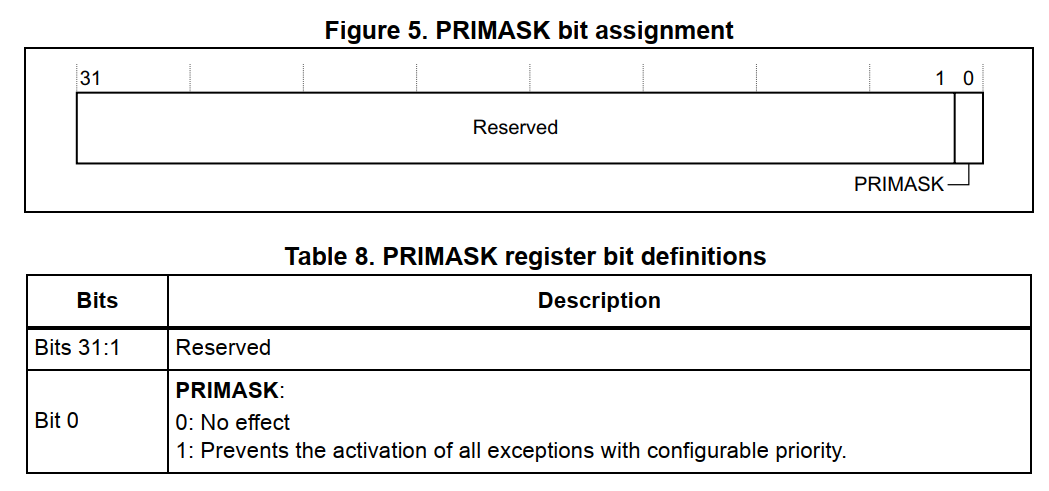


## 中断控制寄存器的访问

异常及中断的管理都是通过NVIC的寄存器和CPU的寄存器来控制的。

**总中断屏蔽寄存器**（PRIMASK）（内核）





CPU寄存器组中的PRIMASK寄存器的最低位控制CPU是否接收中断请求。必须将PRIMASK设置为零，才能使CPU接收中断。

由于PRIMASK是CPU寄存器组中的寄存器，在总线上没有地址，因此必须使用汇编整理对其操作：



CMSIS库函数提供了**void \_\_enable\_irq(void)**函数开启CPU中断，该函数在cmsis\_gcc.h中定义。

/\*\*

\brief Enable IRQ Interrupts

\details Enables IRQ interrupts by clearing the I-bit in the CPSR.

Can only be executed in Privileged modes.

\*/

\_\_STATIC\_FORCEINLINE **void** **\_\_enable\_irq**(**void**)

{

\_\_ASM **volatile** ("cpsie i" : : : "memory");

}

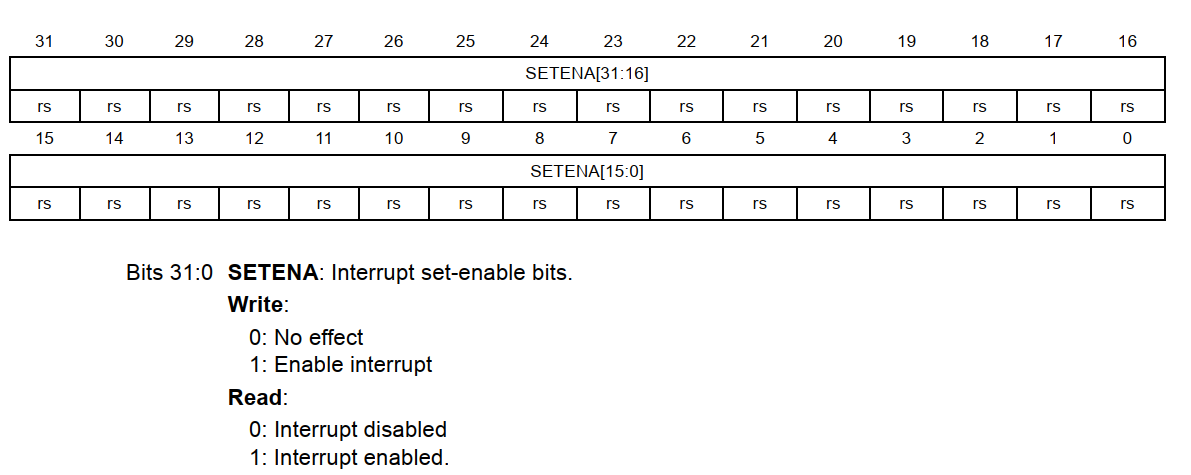
可以看出，由于PRIMASK寄存器位于CPU内核中，因此该函数调用了cpsie I 指令，打开CPU中断。

**NVIC的中断接收设置：**

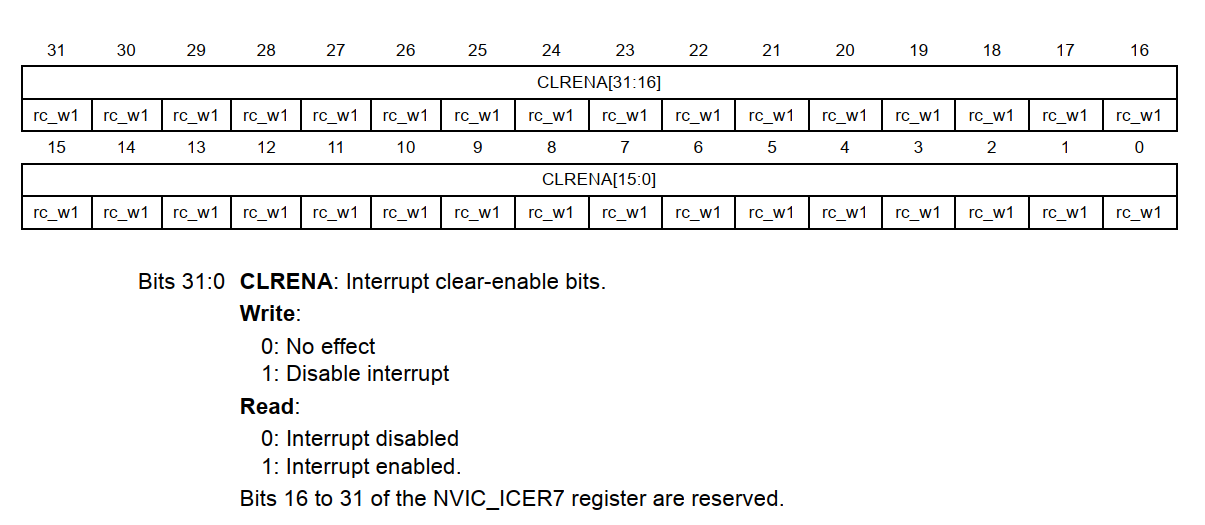


**中断使能寄存器（**NVIC\_ISERx**）**：x=0-7有8个中断使能寄存器，管理0-239号中断（中断编号）

NVIC\_ISER0 bits 0 to 31 are for interrupt 0 to 31, respectivelyNVIC\_ISER1 bits 0 to 31 are for interrupt 32 to 63, respectively....NVIC\_ISER6 bits 0 to 31 are for interrupt 192 to 223, respectivelyNVIC\_ISER7 bits 0 to 15 are for interrupt 224 to 239, respectively



**中断禁止寄存器（NVIC\_ICERx）,** x=0-7有8个中断使禁止寄存器，管理0-239号中断（中断编号）



每一位控制一个中断的开启或者关闭。

读取结果为0，表明中断关闭，结果为1，表明中断开启。

这两组寄存器，写0没有意义。

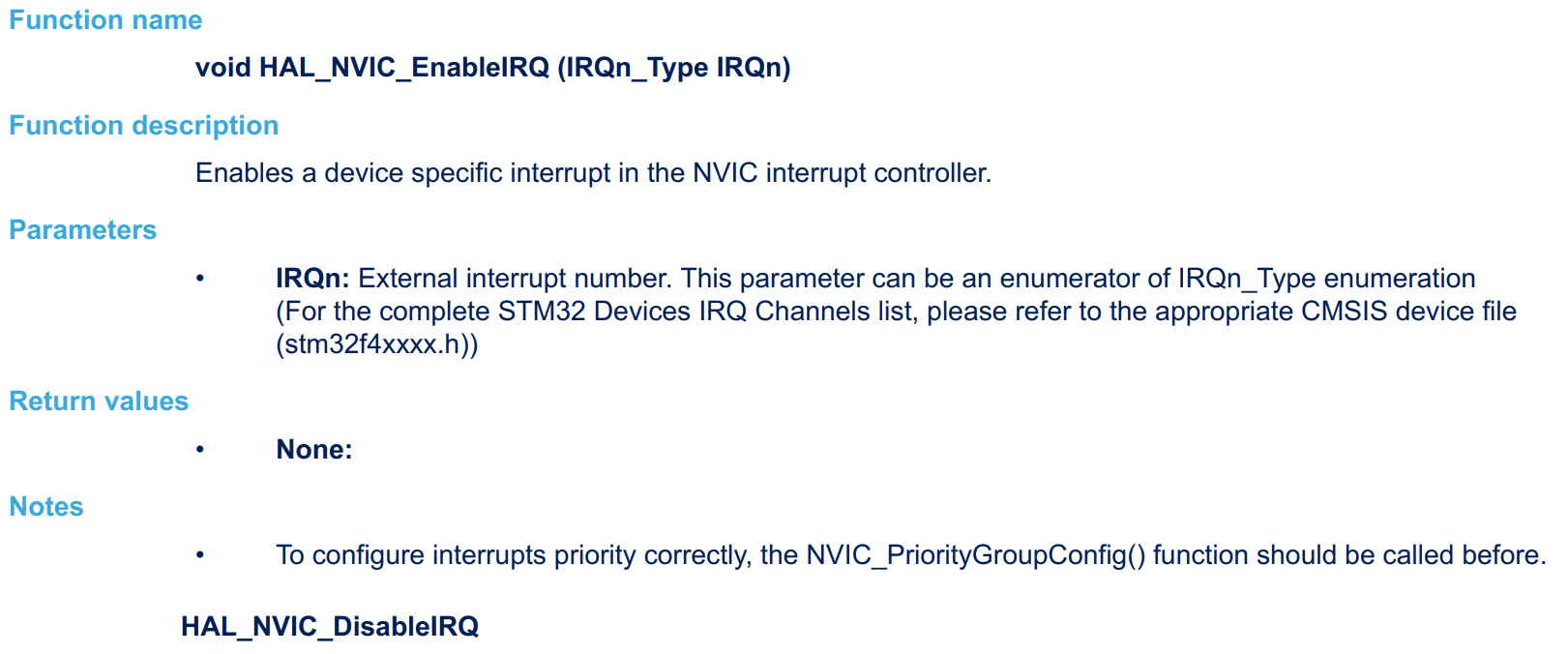
对NVIC\_ISERx的某位写1，开启对应的中断，对**NVIC\_ICERx**的某位写1，关闭对应的中断。

这两组寄存器，只管理外部中断，不管理系统异常，系统异常的中断管理，在各自的异常处理模块中。

HAL库函数中，提供了HAL\_NVIC\_EnableIRQ()和HAL\_NVIC\_DisableIRQ ()函数，来开启和关闭指定的中断和异常，如：

HAL\_NVIC\_EnableIRQ(*EXTI15\_10\_IRQn*);开启来自外部中断[10-15]模块的中断请求

HAL\_NVIC\_EnableIRQ函数在dm00105879-description-of-stm32f4-hal-and-lowlayer-drivers-stmicroelectronics.pdf中的说明如下：



也就是打开NVIC中指定的中断，他的具体实现方式在Drivers\STM32F4xx\_HAL\_Driver\Src\stm32f4xx\_hal\_cortex.c文件中：

**void** **HAL\_NVIC\_EnableIRQ**(IRQn\_Type IRQn)

{

/\* Check the parameters \*/

assert\_param(IS\_NVIC\_DEVICE\_IRQ(IRQn));

/\* Enable interrupt \*/

NVIC\_EnableIRQ(IRQn);

}

其中调用了NVIC\_EnableIRQ函数，NVIC\_EnableIRQ函数的定义如下

**#define** NVIC\_EnableIRQ \_\_NVIC\_EnableIRQ

\_\_STATIC\_INLINE **void** **\_\_NVIC\_EnableIRQ**(IRQn\_Type IRQn)

{

**if** ((int32\_t)(IRQn) >= 0)

{

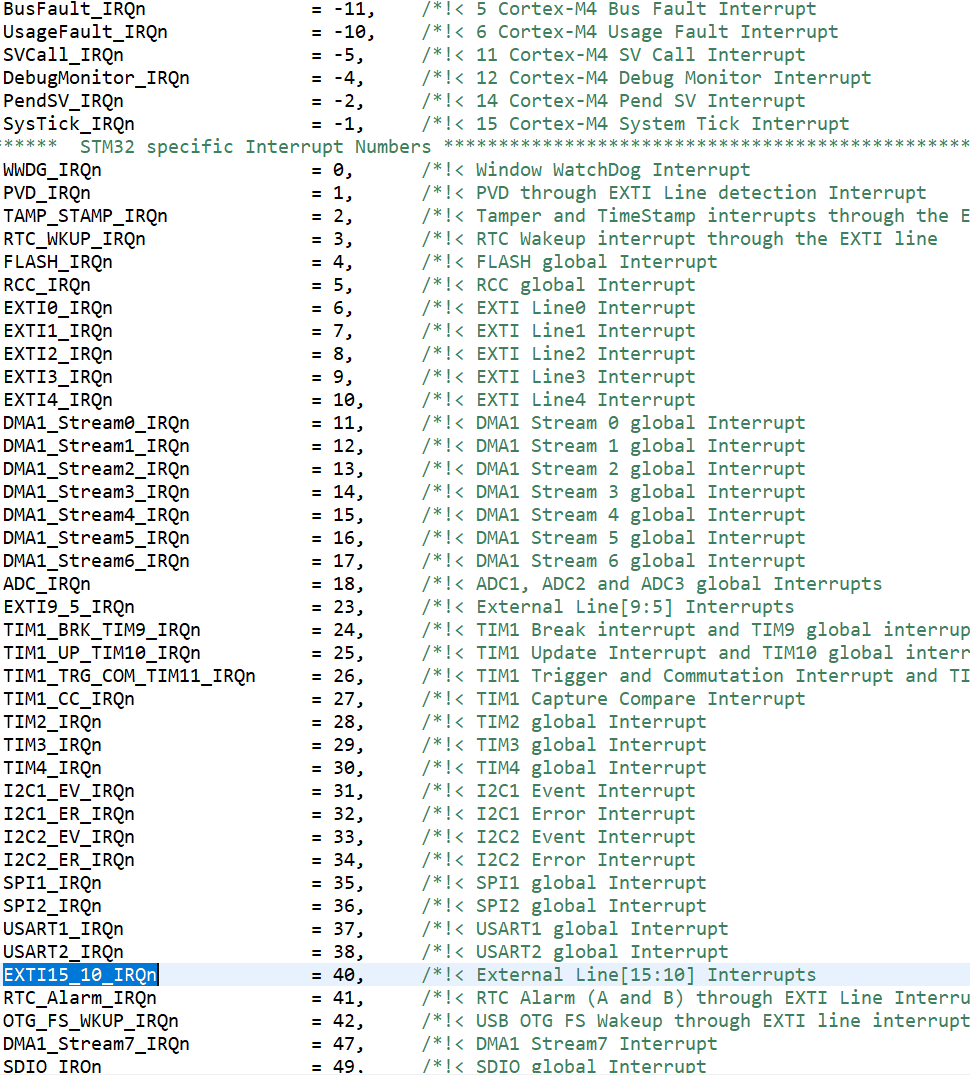
NVIC->ISER[(((uint32\_t)IRQn) >> 5UL)] = (uint32\_t)(1UL << (((uint32\_t)IRQn) & 0x1FUL));

}

}

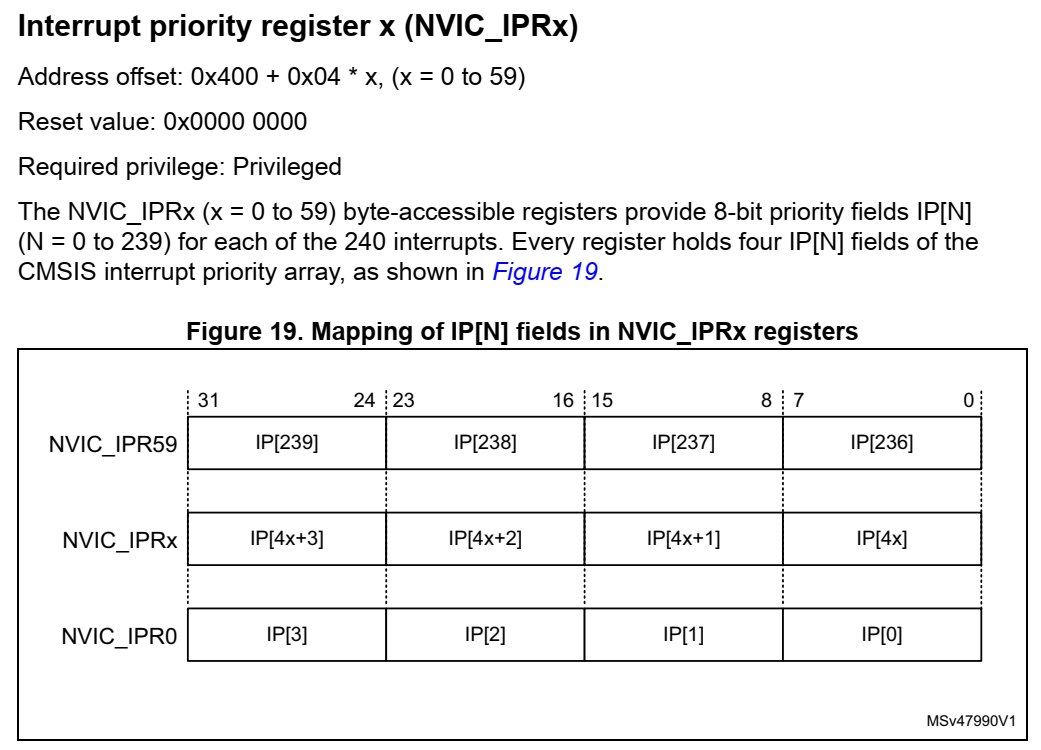
阅读代码可知，该函数传入的参数IRQn为中断编号，首先根据参数IRQn获得要设置的中断的管理位在哪个NVIC\_ISERx寄存器里，然后对这个位写1。

而NVIC\_EnableIRQ的输入参数IRQn与HAL\_NVIC\_EnableIRQ的输入参数IRQn一致，为中断编号。为了便于阅读，采用宏定义代替了各个中断编号的数字：



根据HAL\_NVIC\_EnableIRQ函数的描述，调用该函数之前，需要先调用HAL\_NVIC\_SetPriority函数。该函数的作用是设置中断的优先级，

中断优先级寄存器NVIC\_IPRx

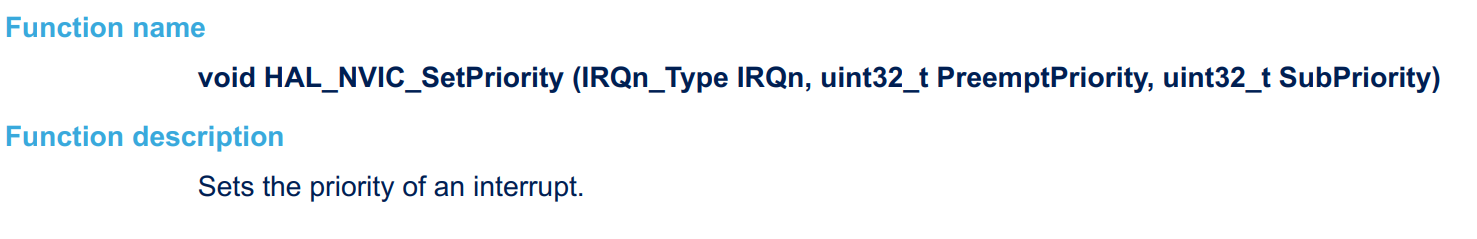


共有60个中断优先级寄存器，管理240个中断的优先级。每个中断的优先级有8位，数字越小，优先级约高。

表示优先级的8位数字，只有高4位[7:4]是有效的，低四位[3:0]读写都是零。

高优先级的中断（数字小）可以抢占低优先级（数字大）的中断，但不能抢占同等优先级和更高优先级的中断。

HAL\_NVIC\_SetPriority的说明如下：



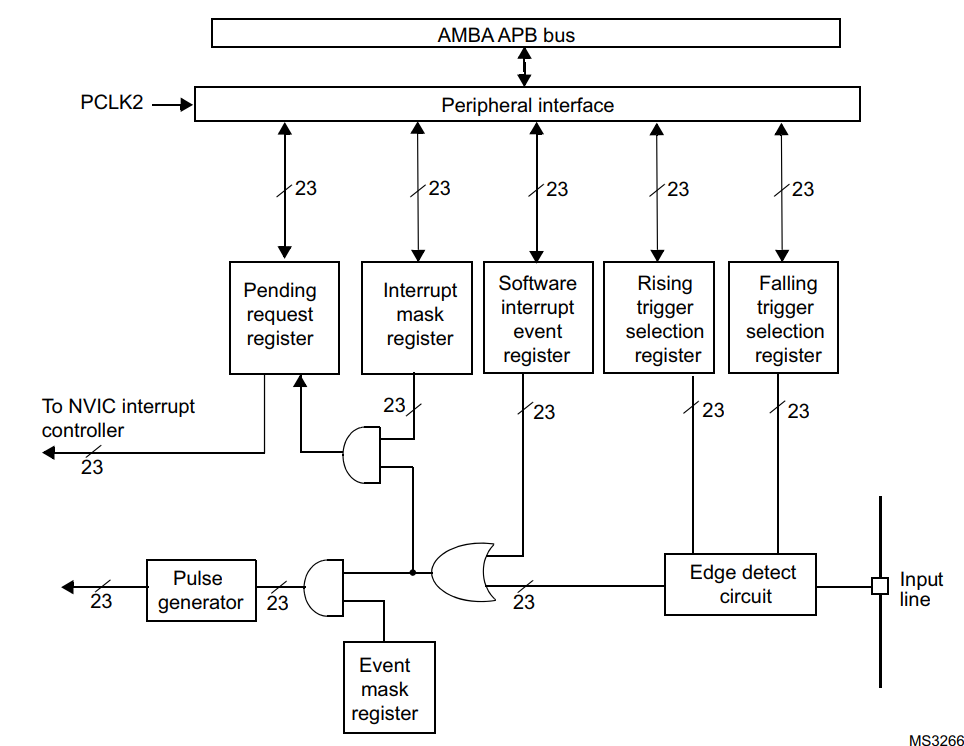
可见其功能就是设置某一中断的优先级。优先级有两个参数，主优先级和子优先级。此处我们只设置主优先级（0-15），子优先级保持为0；

## 外部中断的中断源管理



STM32F401中，管理GPIO输入信号中断的外设叫做“External interrupt/event controller (EXTI)”。

他可以检测GPIO引脚上的上升沿或者下降沿，然后发出中断请求给NVIC。

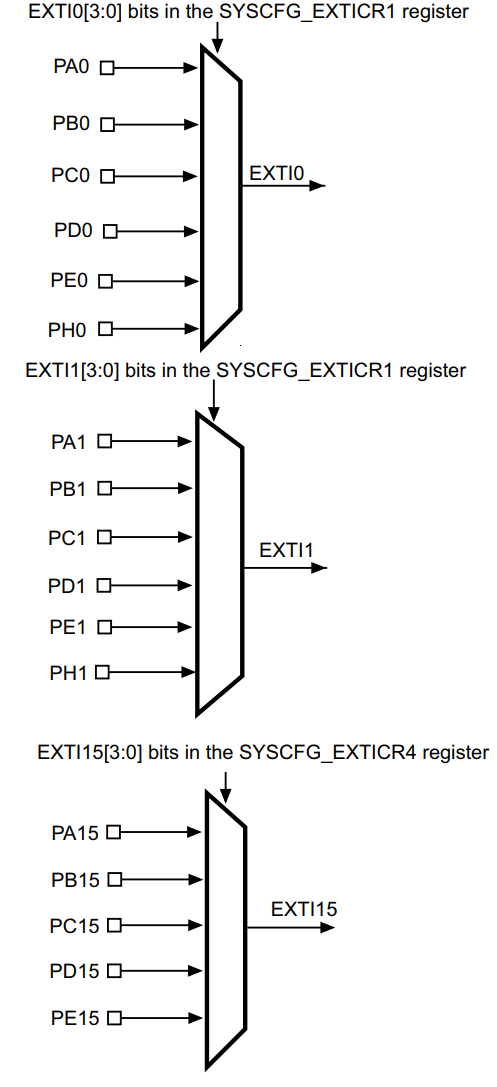


可见，有专门的寄存器专门控制边沿检测电路（Rising trigger selection register 和Falling trigger selection register），用于检测输入信号的上升沿和下降沿。

也可以通过Software interrupt event register用代码出发中断。

有专用的Interrupt mask register使能和屏蔽中断，还有Pending request register用于向NVIC发送中断请求。

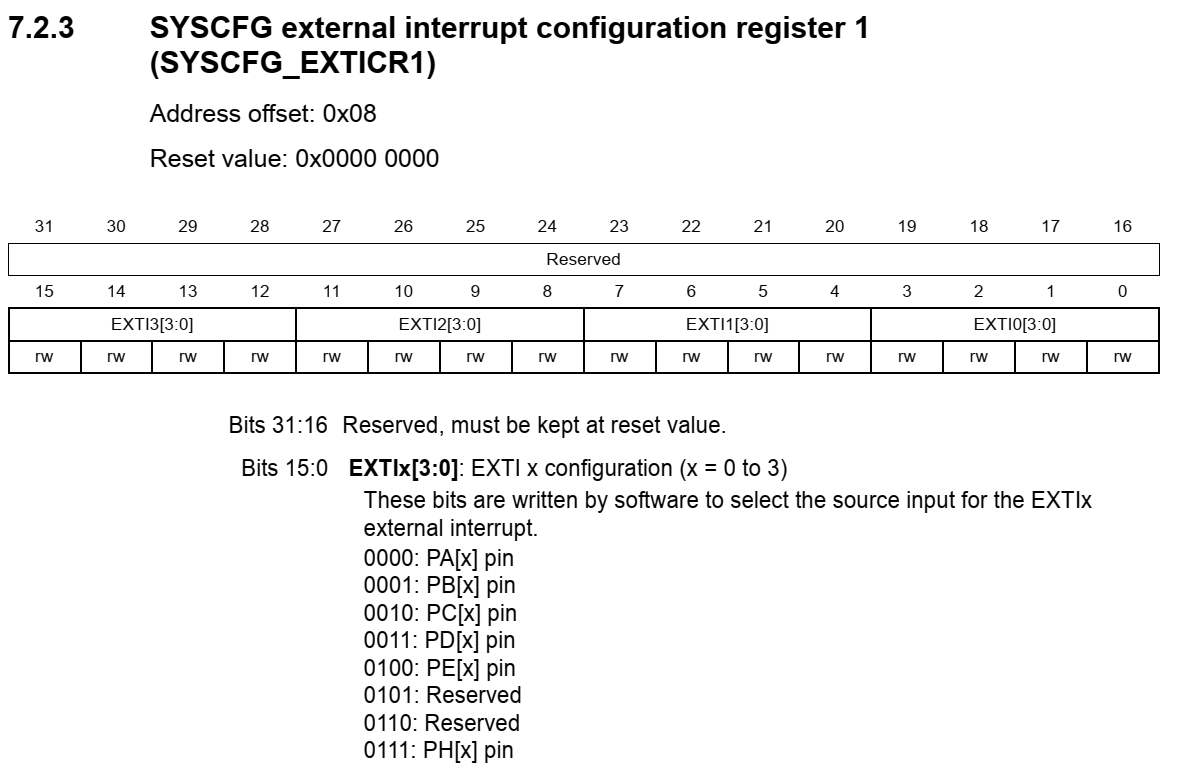
并不是所有的引脚都能产生独立的中断请求信号，在STM32F401中，多个引脚经过边沿检测后，合成一个信号：



16个引脚对应16个信号。

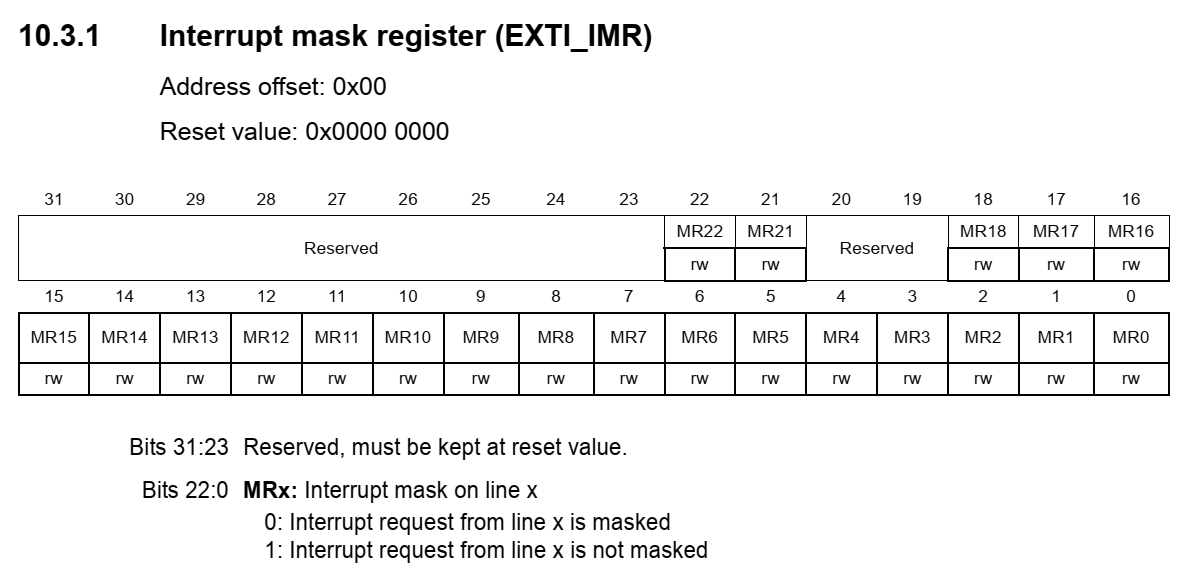
这16个信号中，EXTI0、EXTI1、EXTI2、EXTI3、EXTI4五个信号可以独立生成中断请求发送给NVIC，EXTI5、EXTI6、EXTI7、EXTI8、EXTI9合成一个信号EXTI9\_5发送给NVIC。EXTI10、EXTI11、EXTI12、EXT13、EXTI14、EXTI15合成一个信号EXTI15\_10发送给NVIC。

有寄存器SYSCFG\_EXTICR1、SYSCFG\_EXTICR2、SYSCFG\_EXTICR3、SYSCFG\_EXTICR4控制GPIO引脚与EXTI信号的对应关系。



中断相关的功能，在EXIT模块中设置：

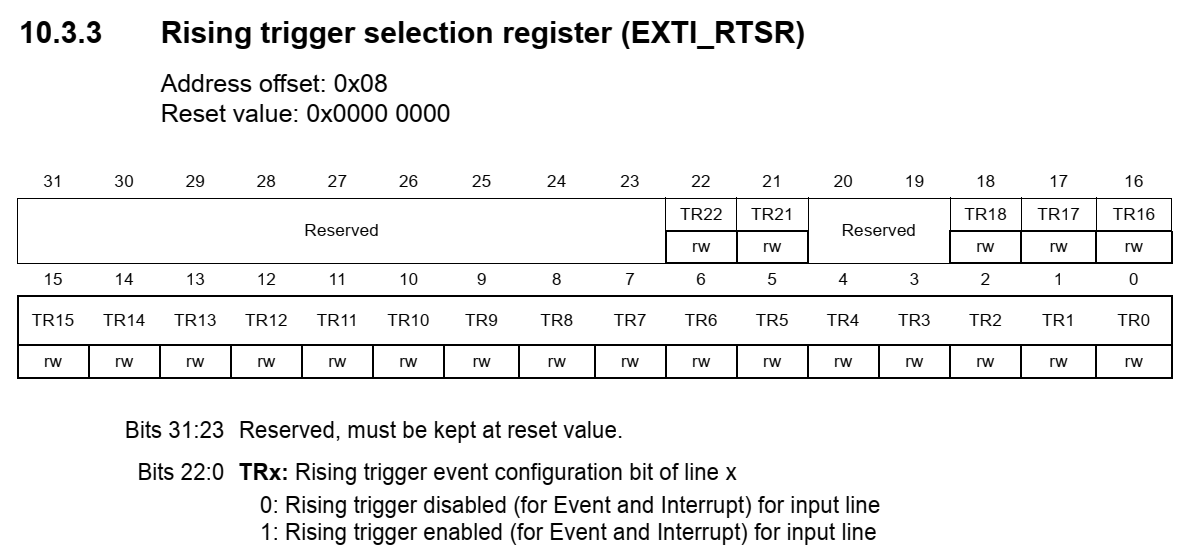
中断屏蔽使能寄存器EXTI\_IMR



其中0到15位控制了信号EXIT0到EXIT15信号是否产生中断请求到NVIC

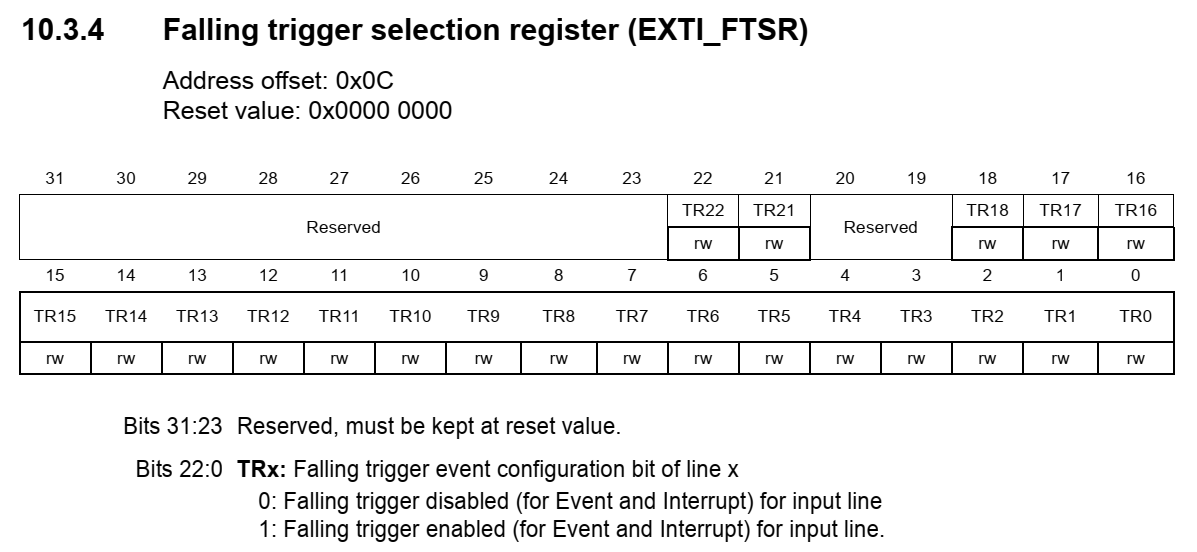
0代表中断被屏蔽，1代表中断使能

上升沿选择寄存器EXTI\_RTSR



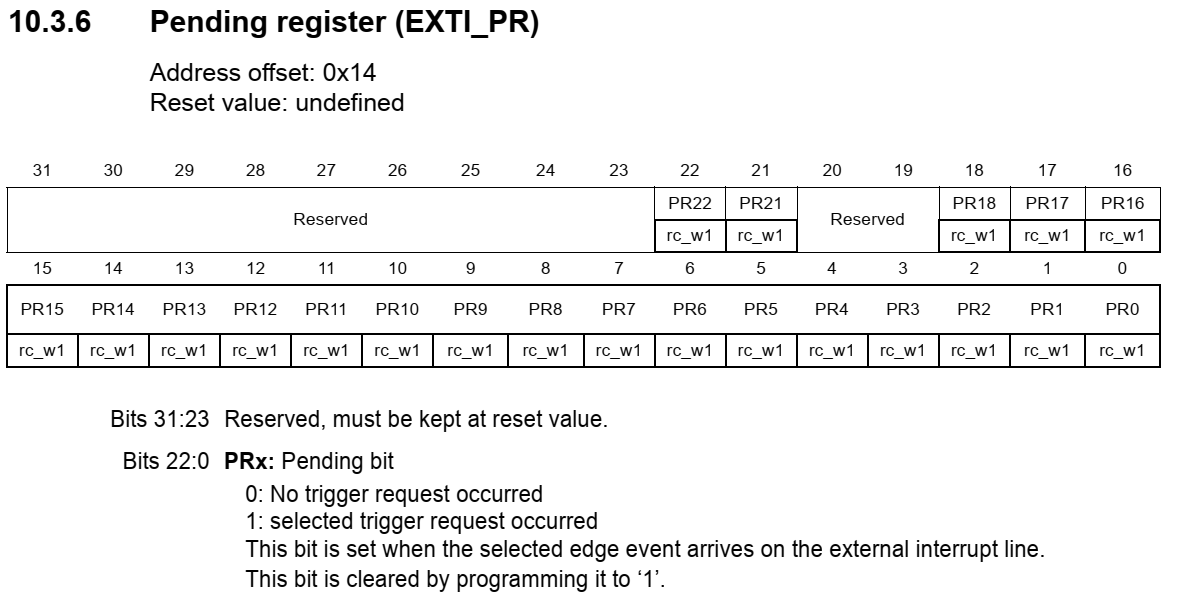
其中0到15位控制了信号EXIT0到EXIT15信号边沿检测功能，置1代表打开上升沿检测功能

下降沿选择寄存器EXTI\_FTSR



其中0到15位控制了信号EXIT0到EXIT15信号边沿检测功能，置1代表打开下降沿检测功能

中断等待与清除**EXTI\_PR**



其中0到15位代表是否有中断发生，1代表有中断发生。如果不清除中断，EXTI模块会不断的向NVIC发送中断请求，造成中断的重复发送，因此要写1清除中断。

GPIO中断从发送到响应的过程：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 外设 | | NVIC | CPU | 中断响应  判断中断的具体来源 | 外设 |
| 产生中断 | 发出中断 | 接收中断 | 接收中断 | 清除中断 |
| 设置要操作的引脚的中断的产生条件：电平还是边沿、高（上升）或低（下降） | EXTI\_IMR中使能中断 |  |  | EXTI\_PR |

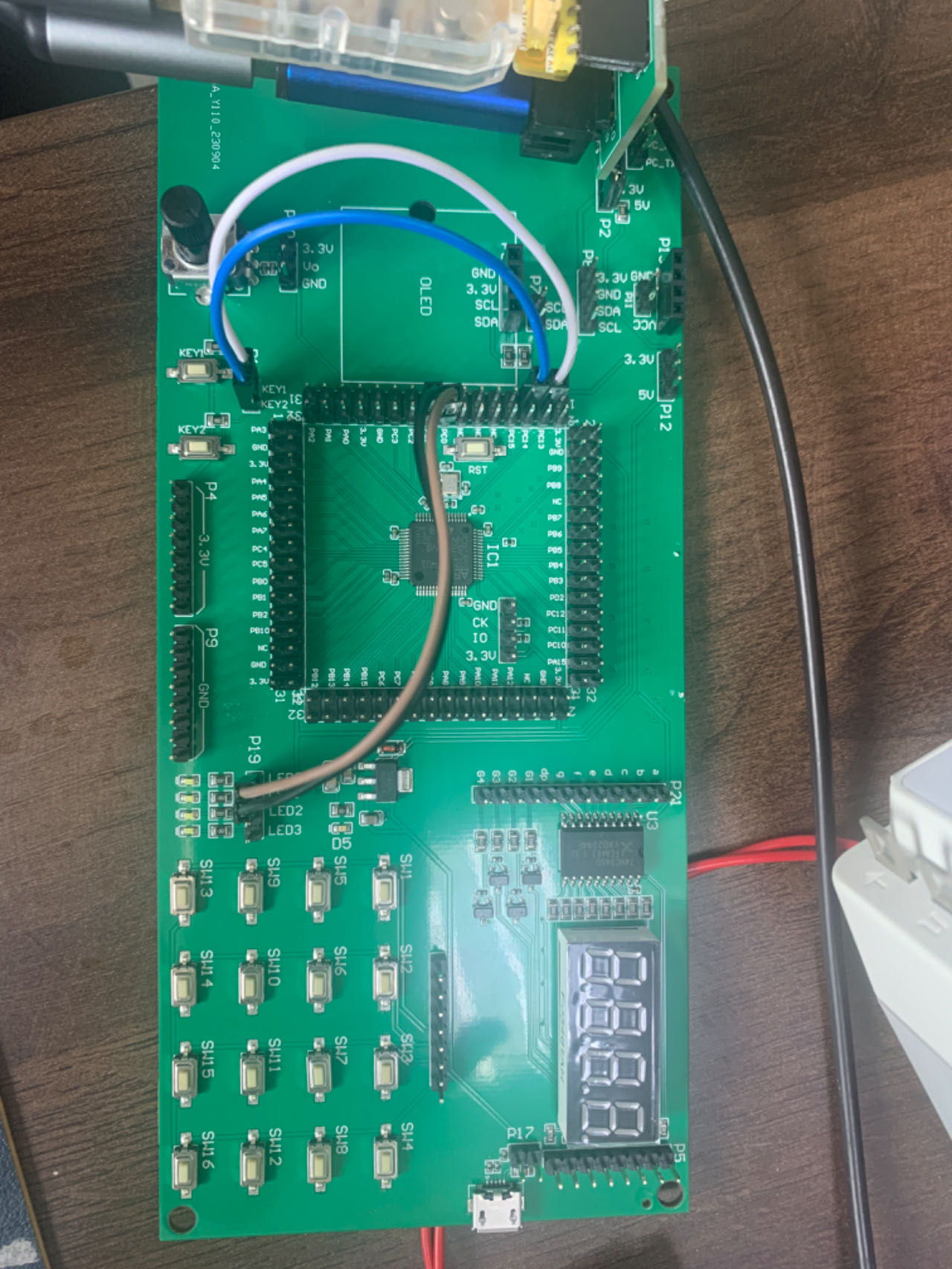
# GPIO中断

## 设置GPIO中断:简单按键示例

**简单按键示例的中断响应示例：**

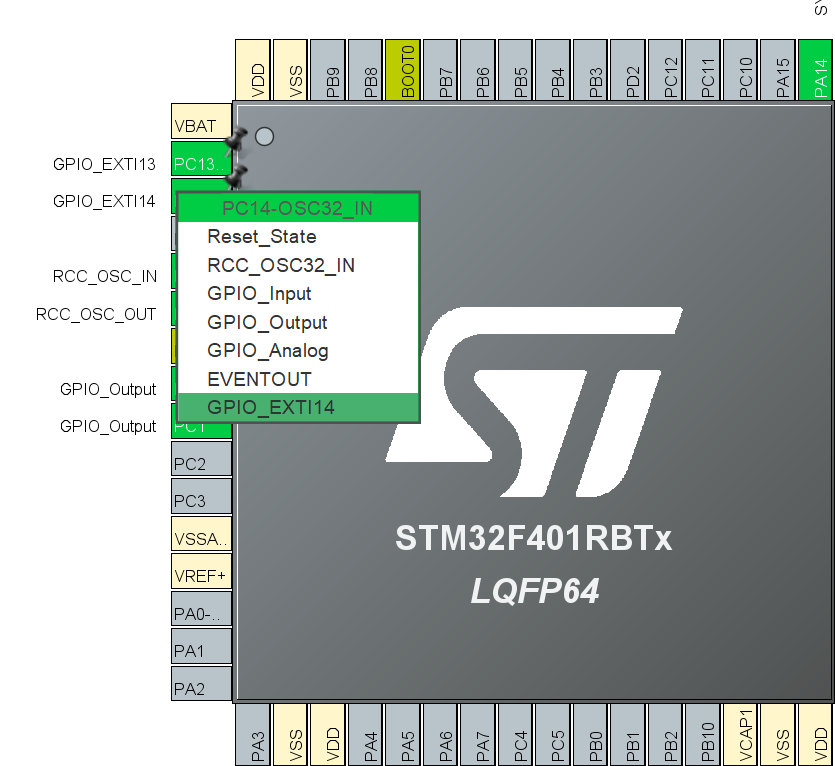
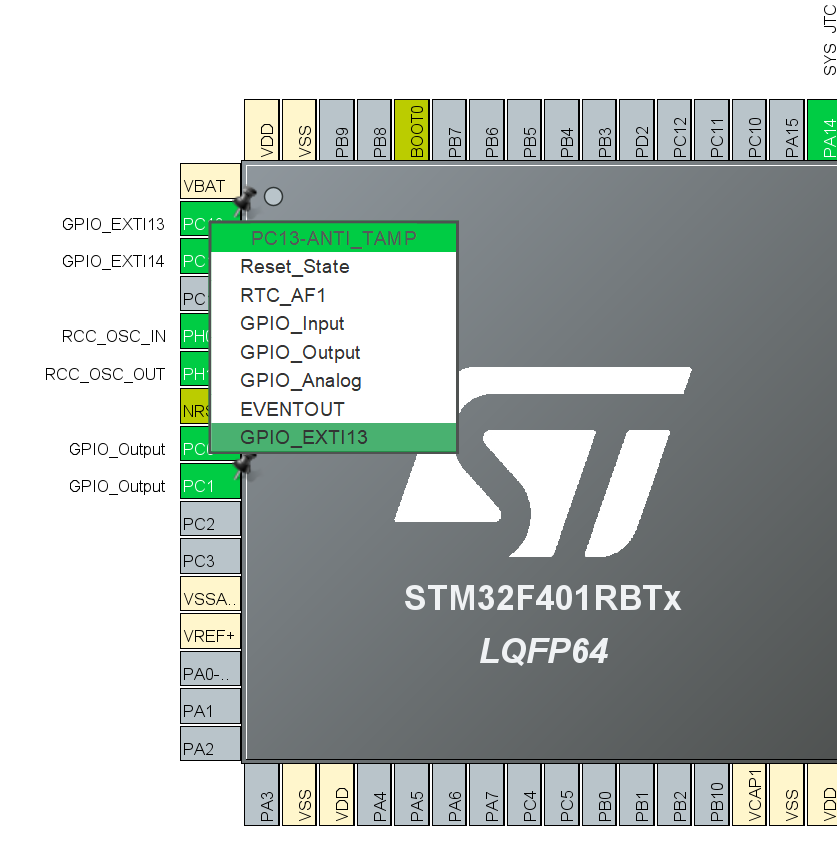
传统的简单按键检测，是在主循环里面不停的查询按键的状态。如果主函数的执行时间较长，容易造成遗漏。因此可以改用中断的方式检测按键，可以避免错过按键的检测。

将KEY1和KEY2连接的PC13和PC14，使用按键信号的下降沿触发外部中断。

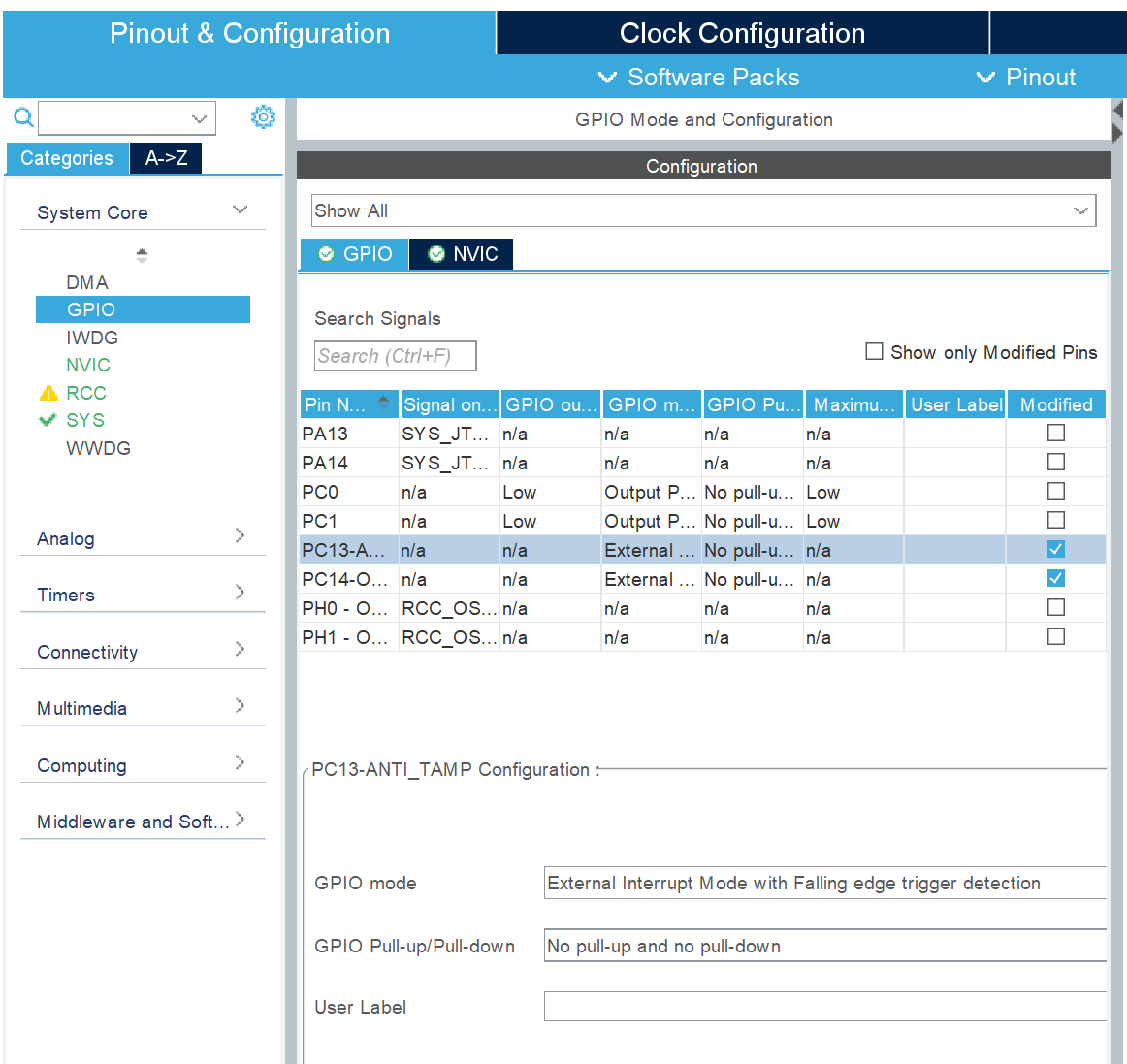


首先连接按钮与PC13和PC14。

然后打开器件配置界面，左键单击PC13引脚，在弹出的选项中选择GPIO\_EXIT13。同样的，左键单击PC14引脚，在弹出的选项中选择GPIO\_EXIT14。

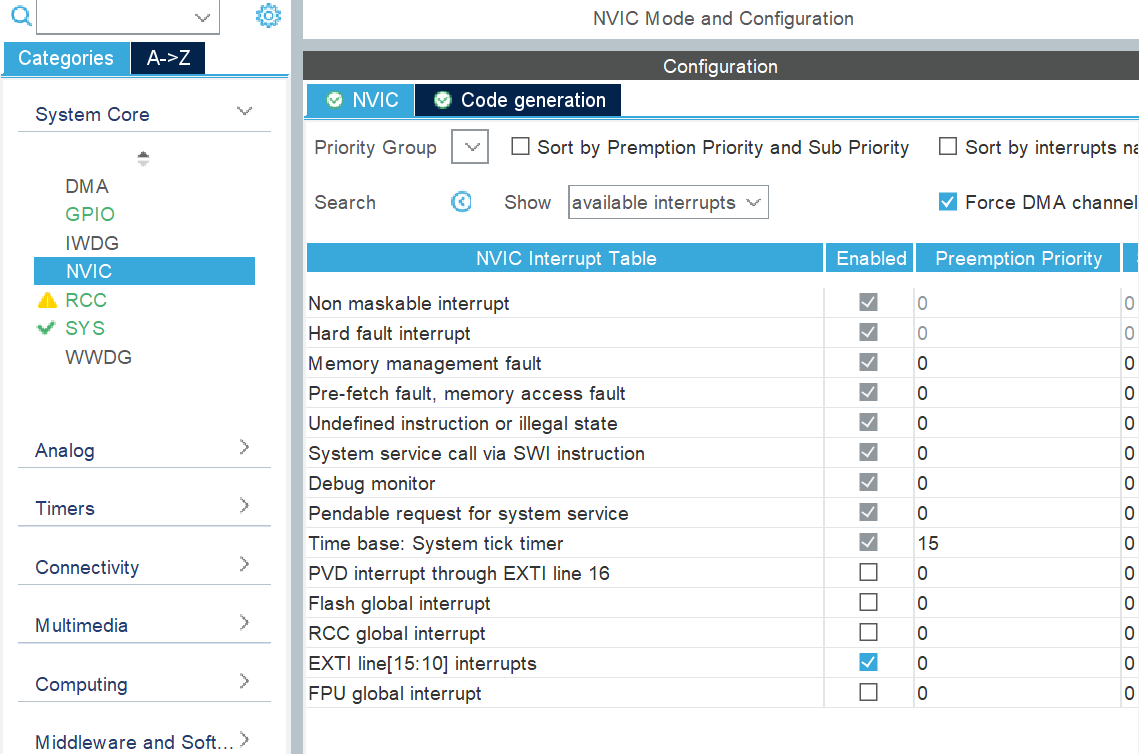


然后点击左侧的System Core->GPIO。在弹出的配置界面中单击选择PC13，修改GPIO mode 为External Interrupt Mode with Falling edge trigger detection。



同样的单击选择PC14，修改GPIO mode 为External Interrupt Mode with Falling edge trigger detection。

然后点击击左侧的System Core->NVIC，在弹出的配置界面中，勾选EXTI line[15:10] interrupts。



配置完成后，保存并生成代码。

生成代码后，发现在MX\_GPIO\_Init函数中，PC13和PC14的初始化代码为：

/\*Configure GPIO pins : PC13 PC14 \*/

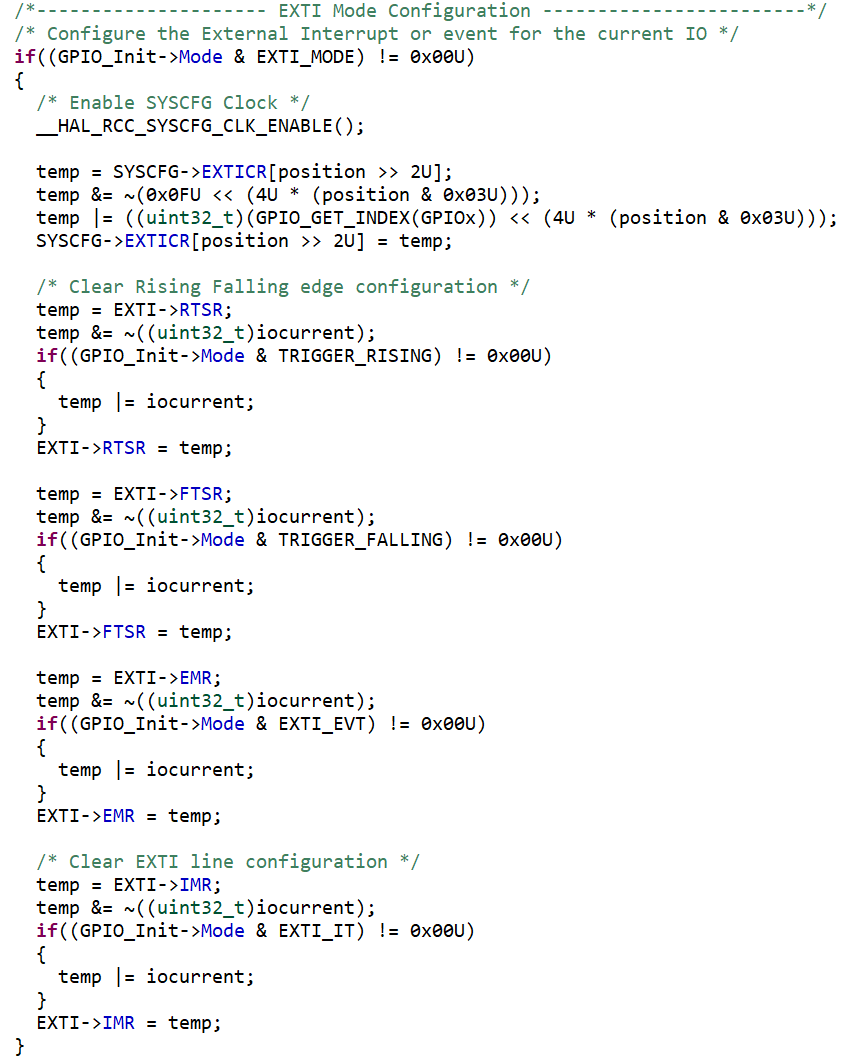
GPIO\_InitStruct.Pin = GPIO\_PIN\_13|GPIO\_PIN\_14;

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_FALLING;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

HAL\_GPIO\_Init(GPIOC, &GPIO\_InitStruct);

可见，初始化时，将引脚的Mode设置为GPIO\_MODE\_IT\_FALLING。这样，在HAL\_GPIO\_Init函数里，会根据该参数设置SYSCFG\_EXTICR4寄存器，将PC13的信号通往EXTI13，将PC14信号通往EXTI14。然后设置EXTI\_RTSR、EXTI\_FTSR选择上升沿和下降沿检测，最后设置EXTI\_IMR寄存器，开启中断。



初始化引脚后，MX\_GPIO\_Init函数中调用了

/\* EXTI interrupt init\*/

HAL\_NVIC\_SetPriority(*EXTI15\_10\_IRQn*, 0, 0);

HAL\_NVIC\_EnableIRQ(*EXTI15\_10\_IRQn*);

开启EXTI15\_10中断，并设置优先级。

在stm32f4xx\_it.c文件中，自动生成了中断服务函数EXTI15\_10\_IRQHandler：

**void** **EXTI15\_10\_IRQHandler**(**void**)

{

/\* USER CODE BEGIN EXTI15\_10\_IRQn 0 \*/

/\* USER CODE END EXTI15\_10\_IRQn 0 \*/

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_13);

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_14);

/\* USER CODE BEGIN EXTI15\_10\_IRQn 1 \*/

/\* USER CODE END EXTI15\_10\_IRQn 1 \*/

}

中断服务函数分别对PC13和PC14产生的中断进行处理，调用了HAL\_GPIO\_EXTI\_IRQHandler处理两个引脚产生的中断。HAL\_GPIO\_EXTI\_IRQHandler的定义如下：

**void** **HAL\_GPIO\_EXTI\_IRQHandler**(uint16\_t GPIO\_Pin)

{

/\* EXTI line interrupt detected \*/

**if**(\_\_HAL\_GPIO\_EXTI\_GET\_IT(GPIO\_Pin) != *RESET*)

{

\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT(GPIO\_Pin);

HAL\_GPIO\_EXTI\_Callback(GPIO\_Pin);

}

}

可见HAL\_GPIO\_EXTI\_IRQHandler首先调用\_\_HAL\_GPIO\_EXTI\_GET\_IT函数查看当前信号的中断状态，\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT的定义如下：

**#define** \_\_HAL\_GPIO\_EXTI\_GET\_IT(\_\_EXTI\_LINE\_\_) (EXTI->PR & (\_\_EXTI\_LINE\_\_))

可见，\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT函数其实是查看EXTI\_PR寄存器，查看输入的引脚上是否产生了中断，如果产生了中断，就调用\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT函数清除该中断。\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT的定义如下：

**#define** \_\_HAL\_GPIO\_EXTI\_CLEAR\_IT(\_\_EXTI\_LINE\_\_) (EXTI->PR = (\_\_EXTI\_LINE\_\_))

可见，\_\_HAL\_GPIO\_EXTI\_CLEAR\_IT通过写EXTI\_PR寄存器，清除中断，防止中断的重复发送。

清除完中断后，调用了HAL\_GPIO\_EXTI\_Callback函数，来处理中断请求。HAL\_GPIO\_EXTI\_Callback函数的定义如下：

/\*\*

\* @brief EXTI line detection callbacks.

\* @param GPIO\_Pin Specifies the pins connected EXTI line

\* @retval None

\*/

\_\_weak **void** **HAL\_GPIO\_EXTI\_Callback**(uint16\_t GPIO\_Pin)

{

/\* Prevent unused argument(s) compilation warning \*/

UNUSED(GPIO\_Pin);

/\* NOTE: This function Should not be modified, when the callback is needed,

the HAL\_GPIO\_EXTI\_Callback could be implemented in the user file

\*/

}

HAL\_GPIO\_EXTI\_Callback函数被定义为一个弱函数，里面什么也没做。

在其他位置重新定义HAL\_GPIO\_EXTI\_Callback函数，可以替代这个弱函数，完成需要的功能。

比如，在**main.c**的/\* USER CODE BEGIN PV \*/和/\* USER CODE END PV \*/之间定义两个变量，用于记录中断发送的次数。

/\* USER CODE BEGIN PV \*/

int32\_t key1Count=0,key2Count=0;

/\* USER CODE END PV \*/

然后，在**main.c**的/\* USER CODE BEGIN 0 \*/和/\* USER CODE END 0 \*/之间重新定义HAL\_GPIO\_EXTI\_Callback函数：

/\* USER CODE BEGIN 0 \*/

**void** **HAL\_GPIO\_EXTI\_Callback**(uint16\_t GPIO\_Pin){

**if**(GPIO\_Pin==GPIO\_PIN\_13){

key1Count++;

}

**if**(GPIO\_Pin==GPIO\_PIN\_14){

key2Count++;

}

}

/\* USER CODE END 0 \*/

这样就可以方便的实现中断的响应。在这里，首先判断是哪个引脚产生了中断，如果是PC13引脚的下降沿产生了中断，则使key1Count自加。

如果是PC14引脚的下降沿产生了中断，则使key2Count自加。

调试时，可以通过现场表达式，观察key1Count和key2Count的值。

## 设置GPIO中断，矩阵键盘示例

**矩阵键盘的中断响应示例：**

传统的矩阵键盘扫描程序，是在主循环里面不停的查询检测线的状态。如果主函数的执行时间较长，容易造成遗漏。因此可以改用中断的方式检测按键，可以避免错过按键的检测。

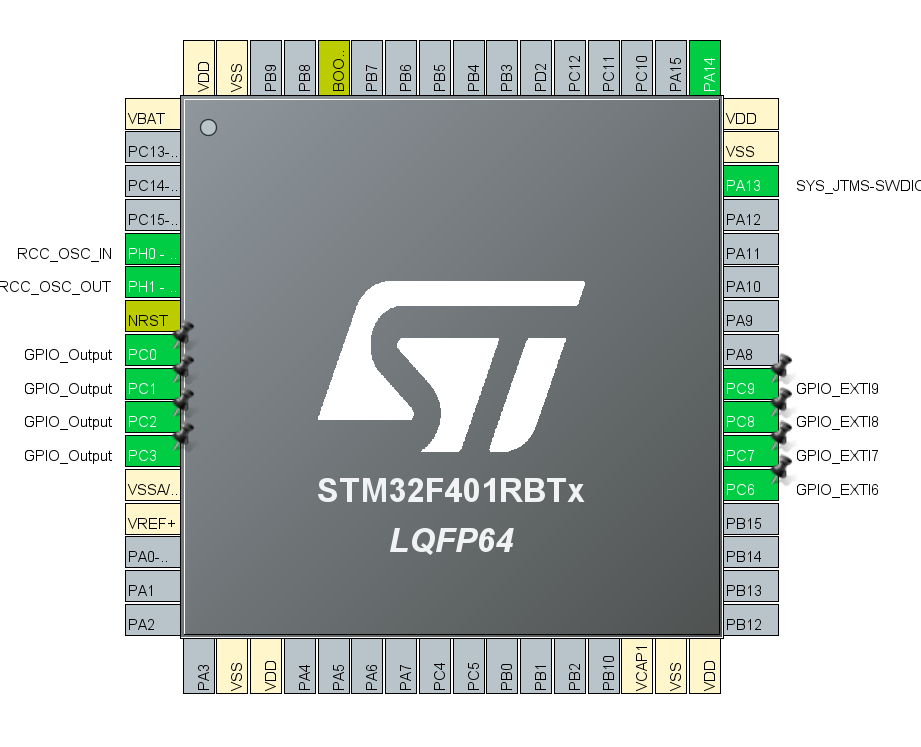
假设键盘的行线连接到PC0、PC1、PC2、PC3四个引脚，键盘的列线连接到PC6、PC7、PC8、PC9四个引脚，如图所示。



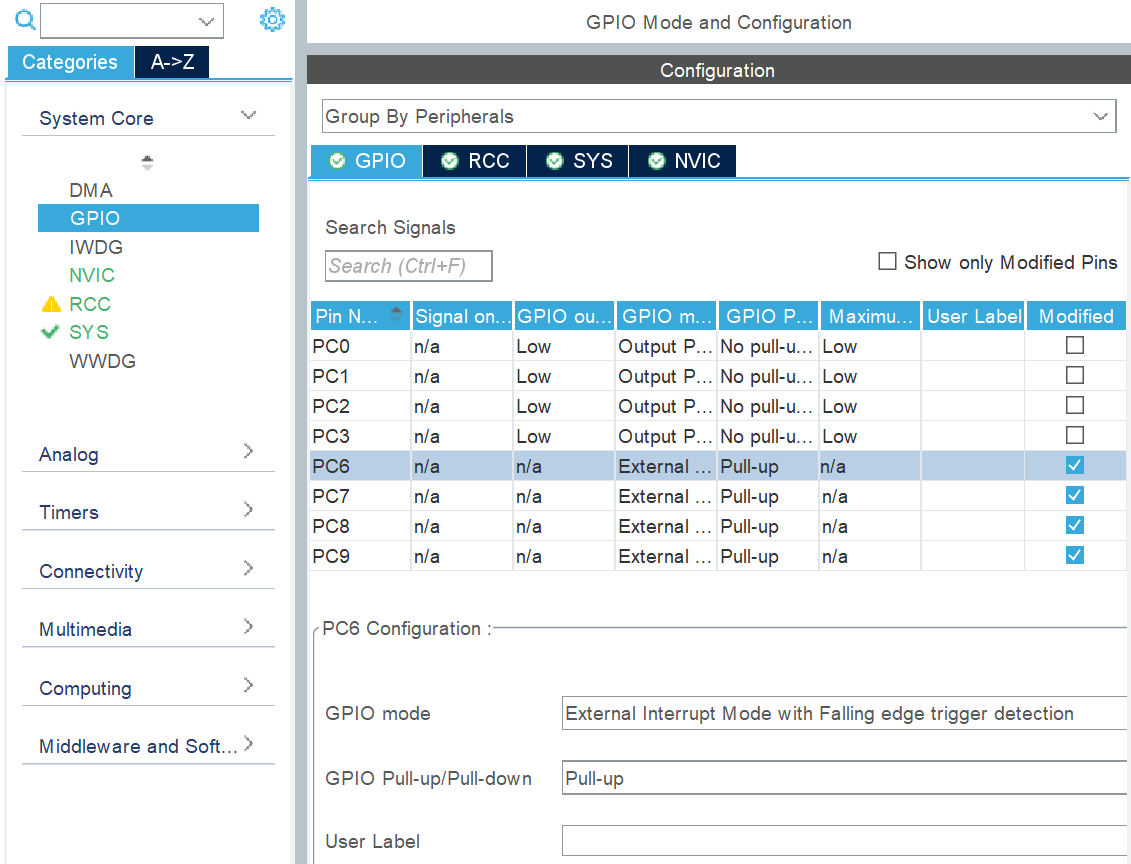
连接好硬件后，使用器件参数配置工具，修改MCU的配置。

由于PC0、PC1、PC2、PC3四个引脚产生的中断是四个单独的中断信号，而PC6、PC7、PC8、PC9四个引脚产生的中断汇集到EXTI9\_5后，发送一个中断请求到NVIC。因此这本示例中，将列线PC6、PC7、PC8、PC9四个引脚当作检测线，将行线PC0、PC1、PC2、PC3四个引脚四个引脚当作控制线。

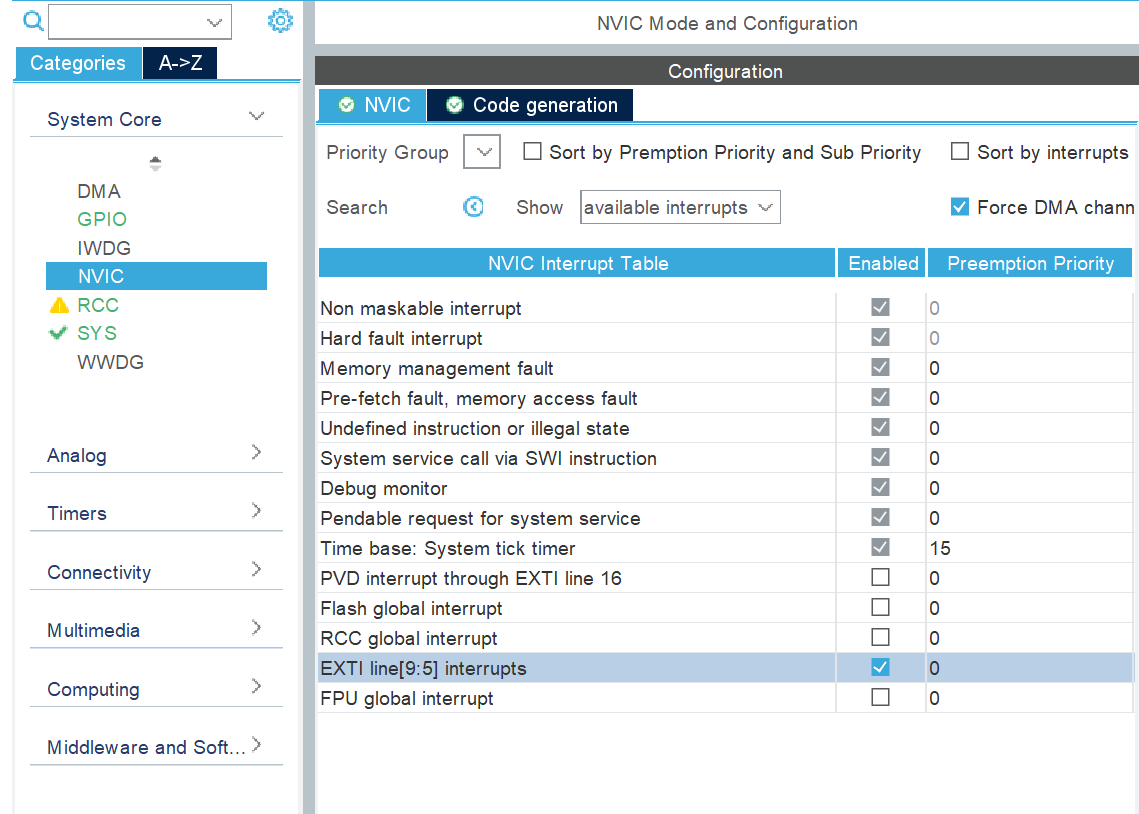
在器件配置界面中配置PC0、PC1、PC2、PC3四个引脚为输出，PC6、PC7、PC8、PC9四个引脚为外部中断输入：



然后在左侧System Core->GPIO中，将PC6、PC7、PC8、PC9四个引脚的GPIO mode配置为 External Interrupt Mode with Falling edge trigger detection，将GPIO Pull-up/Pull-down改为Pull-up，启用上拉电阻。



在System Core->NVIC中，勾选EXTI line[9:5] interrupts



保持并生成初始化代码。

首先在/\* USER CODE BEGIN PV \*/和/\* USER CODE END PV \*/之间定义全局变量，用于查看按键的值。

/\* USER CODE BEGIN PV \*/

int32\_t keyValue=0;

/\* USER CODE END PV \*/

在进入主循环之前，首先将控制线置低电平：

/\* USER CODE BEGIN WHILE \*/

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_0|GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3,*GPIO\_PIN\_RESET*);

**while** (1)

{

/\* USER CODE END WHILE \*/

然后在/\* USER CODE BEGIN 0 \*/和/\* USER CODE END 0 \*/之间重新定义中断响应函数HAL\_GPIO\_EXTI\_Callback，检测按键的值：

/\* USER CODE BEGIN 0 \*/

**void** **HAL\_GPIO\_EXTI\_Callback**(uint16\_t GPIO\_Pin){

int32\_t NewkeyValue=0;

int32\_t row,col,colLineState;

**if**(GPIO\_Pin==GPIO\_PIN\_6)col=0;

**if**(GPIO\_Pin==GPIO\_PIN\_7)col=1;

**if**(GPIO\_Pin==GPIO\_PIN\_8)col=2;

**if**(GPIO\_Pin==GPIO\_PIN\_9)col=3;

NewkeyValue=0;

**for**(row=0;row<4;row++){

**if**(row==0){

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3,*GPIO\_PIN\_SET*);

HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_0, *GPIO\_PIN\_RESET*);

**while**(HAL\_GPIO\_ReadPin(GPIOC, GPIO\_PIN\_1)!=*GPIO\_PIN\_SET*);

}

**if**(row==1){

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_0|GPIO\_PIN\_2|GPIO\_PIN\_3,*GPIO\_PIN\_SET*);

HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_1, *GPIO\_PIN\_RESET*);

**while**(HAL\_GPIO\_ReadPin(GPIOC, GPIO\_PIN\_1)!=*GPIO\_PIN\_RESET*);

}

**if**(row==2){

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_0|GPIO\_PIN\_1|GPIO\_PIN\_3,*GPIO\_PIN\_SET*);

HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_2, *GPIO\_PIN\_RESET*);

**while**(HAL\_GPIO\_ReadPin(GPIOC, GPIO\_PIN\_2)!=*GPIO\_PIN\_RESET*);

}

**if**(row==3){

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_0|GPIO\_PIN\_1|GPIO\_PIN\_2,*GPIO\_PIN\_SET*);

HAL\_GPIO\_WritePin(GPIOC, GPIO\_PIN\_3, *GPIO\_PIN\_RESET*);

**while**(HAL\_GPIO\_ReadPin(GPIOC, GPIO\_PIN\_3)!=*GPIO\_PIN\_RESET*);

}

colLineState=(GPIOC->IDR>>6)&0x0f;

**if**(colLineState!=0x0f){

NewkeyValue=row\*4+col+1;

}

**if**(NewkeyValue){

keyValue=NewkeyValue;

**break**;

}

}

HAL\_GPIO\_WritePin(GPIOC,GPIO\_PIN\_0|GPIO\_PIN\_1|GPIO\_PIN\_2|GPIO\_PIN\_3,*GPIO\_PIN\_RESET*);

}

/\* USER CODE END 0 \*/

PC6、PC7、PC8、PC9四个引脚，任意一个引脚上的下降沿，都会触发EXTI9\_5中断，然后进入HAL\_GPIO\_EXTI\_Callback函数。

所以进入HAL\_GPIO\_EXTI\_Callback函数后，首先检测通过GPIO\_Pin参数检测是哪个引脚产生的中断。

如果是PC6上的下降沿产生的中断，则说明是第0列上的按键被按下。如果是PC7上的下降沿产生的中断，则说明是第1列上的按键被按下。如果是PC8上的下降沿产生的中断，则说明是第2列上的按键被按下。如果是PC9上的下降沿产生的中断，则说明是第3列上的按键被按下。根据这个原理，首先给col变量赋值。

然后开始对行进行扫描，让每一行轮流置低，检测列线的信号，一旦检测的列线上存在0，说明当前扫描的行，便是按键按下的行，根据行号row和列号计算出按键的编号。NewkeyValue=row\*4+col+1;

然后将新的按键赋值给keyValue，观察keyValue变量，可以得知是哪个按键被按下。

注意，在中断响应函数中，不可使用HAL\_Delay函数延时，因此改变控制引脚的状态后，需要读回控制引脚，确认引脚上的电平已经发生了变化。