不讲理论的STM32教程

进阶部分

第七章：中断系统

1.中断的说明

中断的概念在51里面我们就已经接触过了，只不过我们51的中断只有那么五六个，对于我们的STM32来说肯定是不够的，比究竟我们的STM32高级很多，首先是我们的Cortex-M3 的中断和优先级

Cortex-M3 的中断和优先级

通常，把CPU内部产生的紧急事件叫做异常，比如非法指令（除零）、地址访问越界等；把来自CPU外部的片上外设产生的紧急事件叫做中断，比如GPIO引脚电平变化、定时器溢出等。异常和中断的效果基本一致，都是暂停当前任务，优先执行紧急事件，因此一般将中断和异常统称为中断

Cortex-M3内核有256种异常和中断，其中编号1~15是系统异常，16~256是外部中断，听起来是不是有点夸张，但是的确是这样的

编号	名称	优先级	优先级类型	说明

编号	名称	优先级	优先级类型	说明
0	N/A	N/A	N/A	没有异常，正常运行
1	**Reset**	-3	固定	复位
2	**NMI**	-2	固定	不可屏蔽中断（来自外部引脚**NMI**）
3	**HardFault**	-1	固定	只要**FaultMask**没有被复位，**HardFault**会被强制执行
4	MemManageFault	\	可编程	存储器管理Fault，MPU访问违例以及访问非法位置均可引发
5	BusFault	\	可编程	从总线系统收到了错误响应，原因可能是指令预取Abort或数据Abort
6	UsageFault	\	可编程	由于程序错误导致的异常，通常是使用了一条无效指令或者是非法的状态转换
7~10	Reserved	N/A	N/A	保留
11	SVCall	\	可编程	执行系统服务调用指令（SVC）引发的异常
12	Debug Monitor	\	可编程	调试监视器（断点、数据观察点或外部调试请求）
13	Reserved	N/A	N/A	N/A
14	PendSV	\	可编程	为系统设备而设置的“可挂起请求”
15	SysTick	\	可编程	系统滴答定时器
……	……	……	……	……
256	IRQ_239	\	可编程	外部中断#239

如此多的中断，导致了一些新问题。比如两个中断同时发生，应该先执行哪个中断任务？又比如一个中断发生了，又来了一个更紧急的中断，是该继续执行原来的中断，还是执行新的紧急中断？

针对这些问题，Cortex-M3内核有一个专门管理中断的外设NVIC（Nested Vectored Interrupt Controller，嵌套向量中断控制器），通过优先级控制中断的嵌套和调度。NVIC是一个总的中断控制器，无论是来在内核的异常还是外设的外部中断，都由NVIC统一进行管理。

Reset（复位）、NMI（Non Maskable Interrupt，不可屏蔽中断）、HardFault（硬件异常）的优先级是固定的，且优先级是负数，也就是最高的（优先级数字越小，优先级越高）。剩下的异常或中断，都是可以通过修改NVIC的寄存器调整优先级（但不能设置为负数）

通过应用中断和复位控制寄存器（Application Interrupt and Reset Control Register，AIRCR）的Bits[10:8]（PRIGROUP）将优先级分组。分组决定每个可编程中断的PRI_n的Bits[7:0]的高低位分配，从而影响抢占优先和子优先级的级数

PRIGROUP	抢占优先级位	子优先级位	抢占优先级级数	子优先级级数
0	[7:1]	[0]	128	2
1	[7:2]	[1:0]	64	4
2	[7:3]	[2:0]	32	8
3	[7:4]	[3:0]	16	16
4	[7:5]	[4:0]	8	32
5	[7:6]	[5:0]	4	64
6	[7]	[6:0]	2	128
7	None	[7:0]	1	256

这里解释一下我们的抢占优先级，和子优先级，当我们发生中断时，肯定会有一个轻重缓急，所以就会有等级之分，当我们同时发生多个中断时，抢占优先级数字越小的优先级越高，优先级越高的就会有限触发，甚至还会打断低优先级的中断

这样的话，那我们的子优先级又是干嘛的呢？

当两个中断的抢占优先级相同时，即这两个中断将没有嵌套关系，当一个中断到来后，若此时CPU正在处理另一个中断，则这个后到来的中断就要等到前一个中断处理函数处理完毕后才能被处理，当这两个中断同时到达，则中断控制器会根据它们的子优先级决定先处理哪个

如果两个中断的优先级都设置为一样了，那么谁先触发的就谁先执行；如果是同时触发的，那么就根据中断异常表的位置（靠前）来决定谁先执行

而我们的STM32并不是完整的吧M3内核的中断系统全部移植过来了，而是进行了裁剪，下面我们来看看我们的STM32的中断

STM32 的中断和优先级

Cortex-M3设计有256种中断，但大多数MCU都用不到这么多中断，比如STM32F103系列就只有70种异常和中断，其中前10个是系统异常，后面60个是外部中断

STM32F103的异常和中断，基于Cortex-M3修改而来，前面的系统异常部分几乎没有变化，外部中断则对应不同的外设。同样，STM32F103也继承了Cortex-M3的中断优先级规则，因为中断少了很多，中断优先级也用不了那么多，只使用了PRI_n的Bits[7:0]中的Bits[7:4]设置优先级

PRIGROUP	抢占优先级位	子优先级位	抢占优先级级数	子优先级级数
3	[7:4]	None	16	1
4	[7:5]	[4]	8	2
5	[7:6]	[5:4]	4	4
6	[7]	[6:4]	2	8
7	None	[7:4]	1	16

一共是这五种划分情况，具体的概念大概就是这样，我们在下一小节中将会教大家如何去配置我们的NVIC

由于中断的内容较多，我们这里就先讲解我们最常用到的外部中断，其他中断我们碰到了再去讲解使用方法

外部中断：

外部中断相信小伙伴们在51当中就有接触，根据触发方式可以分为上升沿触发，下降沿触发以及上下沿都出发的模式

2.中断的使用

和前面一样，我们打算通过一个例子为大家展示我们的外部中断，对！没错，还是用点灯实验

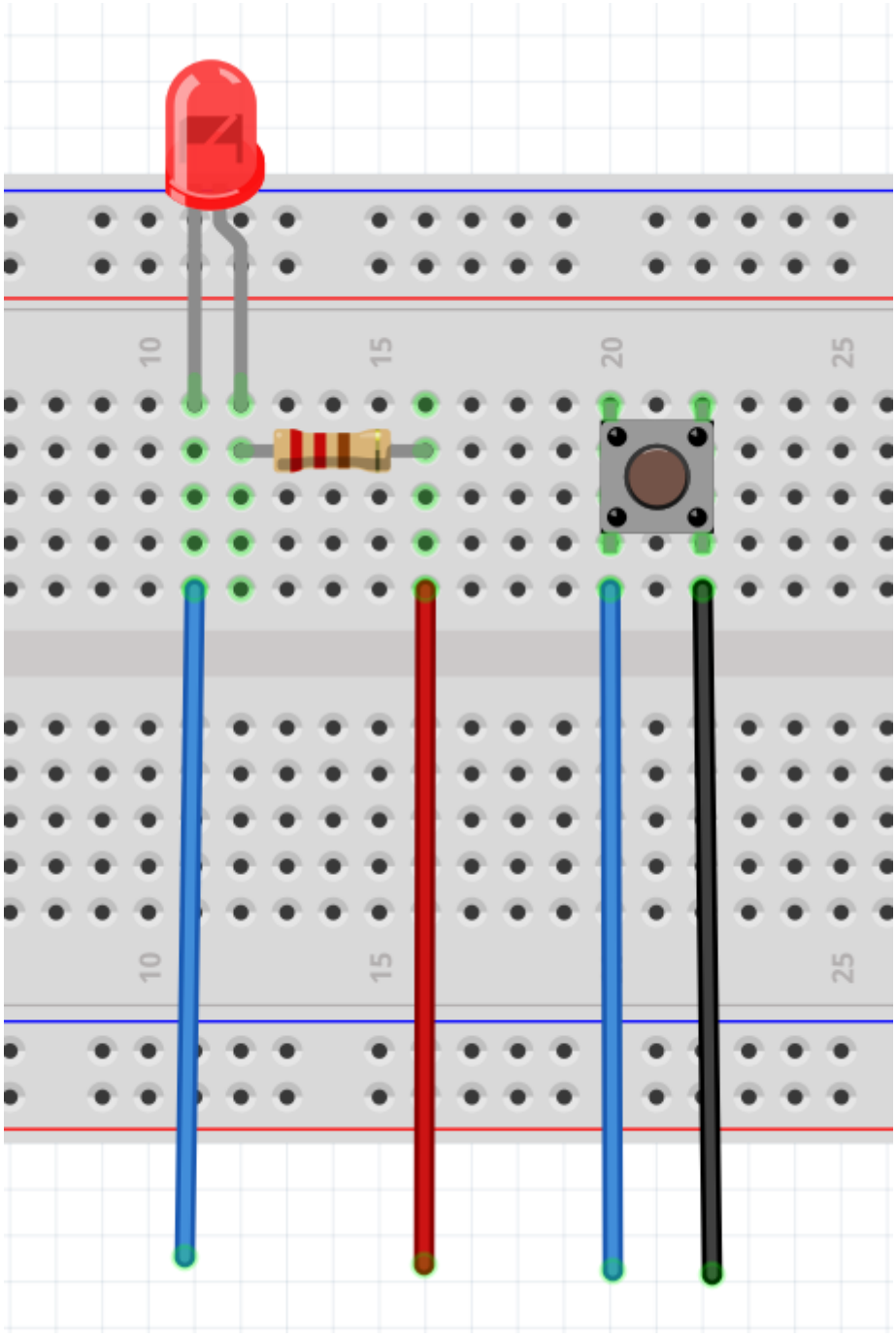
项目：

外部中断控制LED亮灭

材料:

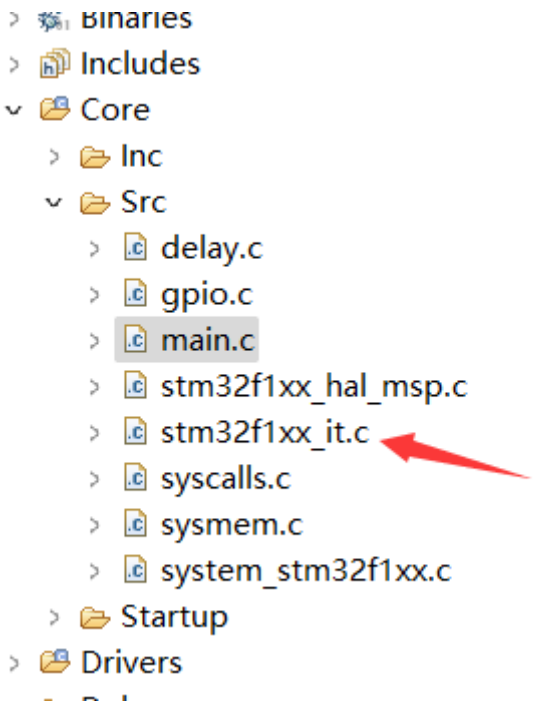
面包板，LED小灯泡一个，1K电阻一个，独立按键一个，导线若干

原理图:

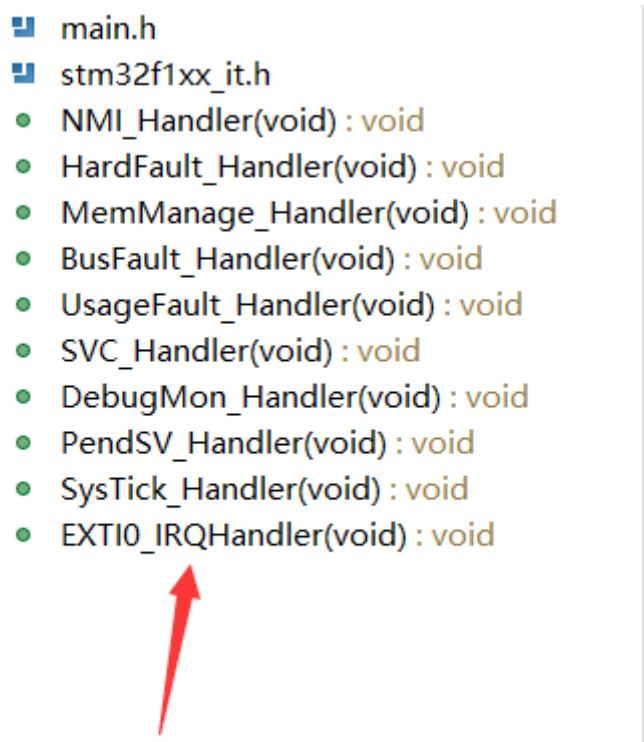


思路分析:

首先我们需要知道我们的中断函数是在哪里



我们点开stm32f1xx_it.c文件，然后就可以在右侧大纲看到，目前我们用到的中断都在里面了



点击最下面的EXTI0_IRQHandler();，这个函数就是我们的外部中断0，也就是我们按键按下出发的中断函数，然后我们来仔细看一下代码

```
204 void EXTI0_IRQHandler(void)
205 {
206   /* USER CODE BEGIN EXTI0_IRQn 0 */
207
208   /* USER CODE END EXTI0_IRQn 0 */
209   HAL_GPIO_EXTI_IRQHandler(KEY_Pin);
210   /* USER CODE BEGIN EXTI0_IRQn 1 */
211
212   /* USER CODE END EXTI0_IRQn 1 */
213 }
```

可以看到，他是直接执行了我们的HAL_GPIO_EXTI_IRQHandler(KEY_Pin);函数，这个是个我们的HAL库封装好的函数，我们转跳过去看看具体内容

```
546 void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
547 {
548   /* EXTI line interrupt detected */
549   if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
550   {
551     __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
552     HAL_GPIO_EXTI_Callback(GPIO_Pin);
553   }
554 }
```

可以看到，函数中除了一些判断，比较显眼的就是我标注的这一行，HAL_GPIO_EXTI_Callback(GPIO_Pin);我们再次转跳过去，看看具体内容

```
561 __weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
562 {
563   /* Prevent unused argument(s) compilation warning */
564   UNUSED(GPIO_Pin);
565   /* NOTE: This function Should not be modified, when the callback is needed,
566            the HAL_GPIO_EXTI_Callback could be implemented in the user file
567    */
568 }
```

可以看到，这个函数前面有一个___weak，相信C语言基础较好的小伙伴们肯定就知道了，这个是弱函数，需要我们去重下写，如果不重新写的话他就会默认使用弱函数，其实我们的库函数里有很多的弱函数在默默的维持着程序的正常运行，这里就不拓展说下去了，知道弱函数概念就好了

总结一下，我们的按键触发中断，会执行我们的EXTIo_IRQHandler();-->HAL_GPIO_EXTI_IRQHandler(KEY_Pin);

-->HAL_GPIO_EXTI_Callback(GPIO_Pin);

然后我们只需要重写HAL_GPIO_EXTI_Callback(GPIO_Pin);写入我们中断要执行的指令就好了

代码书写：

我们讲HAL_GPIO_EXTI_Callback(GPIO_Pin);函数在main.c中重写一下

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
    if(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0){
        HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, 1-
HAL_GPIO_ReadPin(LED_GPIO_Port,LED_Pin));
    }
    while(HAL_GPIO_ReadPin(KEY_GPIO_Port,KEY_Pin)==0);
}
```

要注意的是我们需要放在主函数的外部，当中断时他就会转跳过来执行内部的内容哦

函数的内容几乎和我们上节课的没什么区别，只是把我们的HAL_Delay(20);给去掉了，因为我们的HAL_Delay();函数不能在我们的中断中使用，具体原因这里不展开讲解，后续课程会慢慢涉及到

好了，我们的项目代码就算完成了，编译一下，然后烧录

可以看到，我们的实验现象和上期完全一样，但是我们主函数中是一句代码都没有书写的，好了本章内容结束，下章见！