一、问题引入

在 按键实验中，我们需要在while(1) 中不断的查询 S1 ~ S4对应的GPIO引脚的电平状态

从而判断按键是否按下或弹起

这是因为CPU不知道用户什么时候会按下按键

为了放置没有检测到，就必须一直检测

像这样，检测外部事件的方式我们称之为**轮询**

轮询天生就有缺陷：

（1）使程序流程阻塞在循环中

（2）浪费CPU

（3）轮询有时间差，响应不及时

现在的方案：一旦事件发生，就马上通知我

=> 当按键按下时，外设立即主动的通知CPU

=> 中断

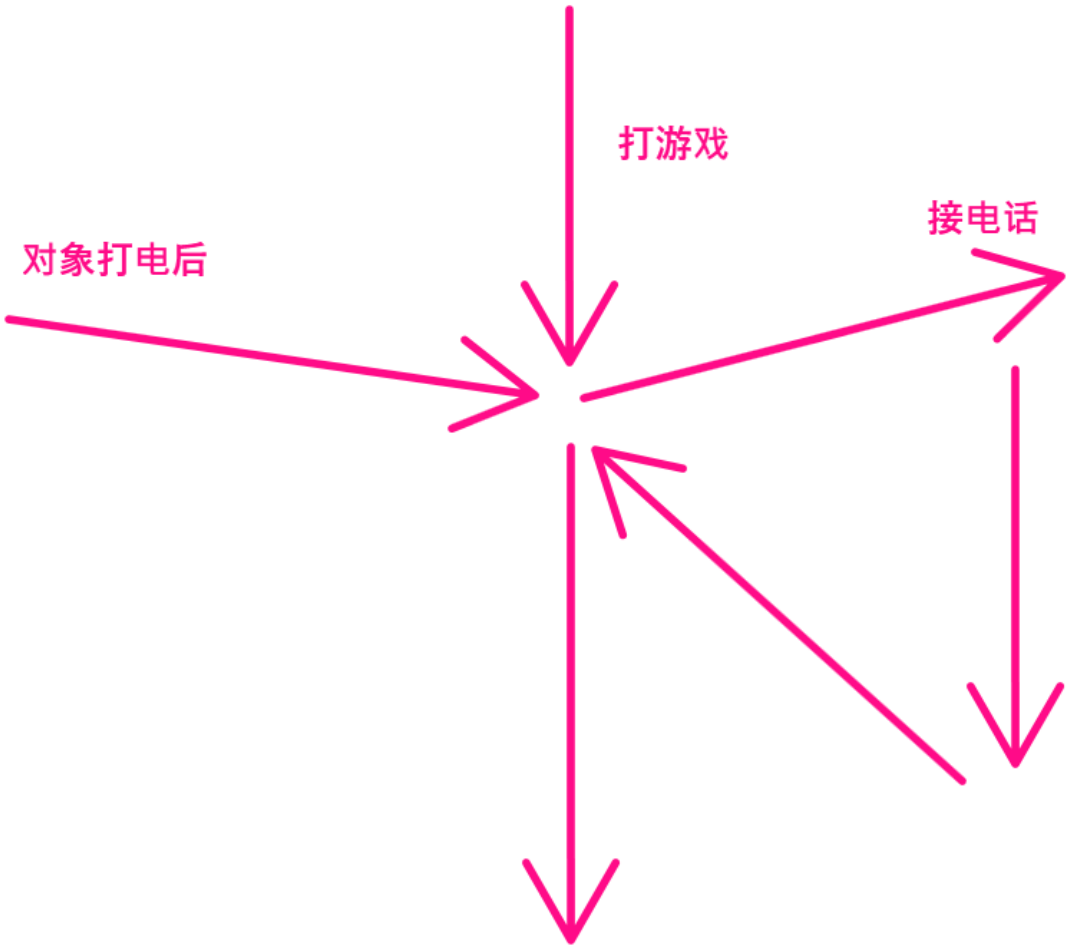
二、中断

中断一般定义为打断CPU指令指向顺序的事件

CPU在执行当前程序的过程中，由于某种随机出现的`外设请求`或`CPU异常`

使CPU暂停当前正在执行的程序，去执行相应的服务处理程序

当服务处理程序执行完毕后，CPU再返回到暂停处，继续执行原来的程序



引起中断产生的设备，称之为**中断源**

三、STM32F4xx 中断机制

Cortex M4 处理器有两种模式（工作模式）：

（1）线程模式（Thread Mode）　　正常模式

（2）处理模式（Handler Mode）　　异常模式 / 中断模式

当CPU的INR引脚收到一个信号

CPU会自动切换到 处理模式 去执行相应的中断服务程序

做完后，CPU会回到线程模式继续执行原来的程序

Q：中断有很多种，当中断产生后，CPU如何知道是哪个设备产生的?

CPU如何知道到哪里去执行中断服务程序?

A：**中断向量表!**

1. 中断向量表

STM32F4xx 给每一个中断事件，一个编号，这个编号就是**中断号**

产生某个中断时，这个中断号就保存在 IPSR（中断程序状态寄存器）

CPU预定义了一个数组，遇到xxx中断，就会跳转到数组元素对应的地址执行

这个数组就是 **中断向量表**

```
__vectors[i]     保存的就是 i号中断的中断服务程序地址
```

也就是说，当 i号中断产生，CPU就会__vectors[i] 所表示的地址中去执行中断服务程序

2. 中断服务函数

CPU寻找的中断服务程序的地址

就是中断服务函数的地址

```
XXX_IRHandler

XXX 表示中断源，可以通过启动文件中的中断向量表查看

CORE/startup_stm32f40xx.s
```

四、STM32F4xx 中断逻辑

一个中断如果要被CPU响应，必须经过两个阶段：

（1）中断源控制

产生中断的设备，本身可以屏蔽自身的中断，所有需要配置中断源

如果中断源控制器开启中断，当中断产生的时候，中断控制器就会向它的上一级（NVIC）发送一个中断请求信号

（2）NVIC（中断控制器）

它时负责管理芯片上所有的中断（异常），所有的中断都需要通过它，然后才能到达CPU

CPU响应中断很简单，就是到指定的位置执行用户的中断服务函数

五、外部中断

STM32F4xx 的GPIO可以当作一个中断源

外部中断 / 事件控制器包含多达 23个用于产生事件/中断请求的边沿检测器：

每根输入线都可以单独进行配置

以选择类型（中断或事件）和相应的触发事件（上升沿触发、下降沿触发或边沿触发）

每跟输入线还可以单独屏蔽：挂起寄存器用于保存中断请求线的状态

外部中断有23个用于产生中断请求的边沿检测器

`EXTI0、EXTI1 .. EXTI15 ..`

但是只有 `EXTI0 ~ EXTI15` 才是针对GPIO

我们一共有144个GPIO口，针对GPIO外部中断只有16个

怎么分配?

```
EXTI0
    PA0
    PB0
    ..
    PI0

EXTI1
    PA1
    PB1
    ...
    PI1

...

eg:
    S1 -- PA0 -- EXTI0
    S2 -- PE2 -- EXTI2
```

六、代码实现

任何中断都需要通过 `中断源配置` 和 `NVIC配置`，所以中断的编程可以分为三个步骤

以外部中断为例:

1. 中断源配置

step1：GPIO配置

```
1. 使能GPIO时钟
2. 配置GPIO为输入模式
```

step2：SYSCFG配置

```
1. 使能SYSCFG时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
2. 选择具体的GPIO引脚作为 EXTIx 的外部中断的输入引脚
    void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx,
                               uint8_t EXTI_PinSourcex)
        @EXTI_PortSourceGPIOx       选定 GPIO分组
                        EXTI_PortSourceGPIOA
                        EXTI_PortSourceGPIOB
                        ...
                        EXTI_PortSourceGPIOI
        @EXTI_PinSourcex
                        EXTI_PinSource0
                        EXTI_PinSource1
                        ...
                        EXTI_PinSource15
    eg:
        // S1 -- PA0 -- EXTI0
        SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
```

step3：配置外部中断控制器

```
void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct)
    @EXTI_InitStruct        结构体指针
                            指向的空间保存了外部中断控制器的所有配置信息
    typedef struct
{
  uint32_t EXTI_Line;// 指定要初始化的外部中断口线
    EXTI_Line0
    EXTI_Line1
    ...
    EXTI_Line15

  EXTIMode_TypeDef EXTI_Mode;// 指定模式
    EXTI_Mode_Interrupt // 中断模式
    EXTI_Mode_Event // 事件模式

  EXTITrigger_TypeDef EXTI_Trigger;// 指定触发方式
    EXTI_Trigger_Rising// 上升沿触发
    EXTI_Trigger_Falling// 下降沿触发
    EXTI_Trigger_Rising_Falling// 双边沿触发

  FunctionalState EXTI_LineCmd;// 使能 或者 失能
    ENABLE
    DISABLE
}EXTI_InitTypeDef;

使用步骤和GPIO_Init() 一样
    step1：定义结构体变量
    step2：给成员变量赋值
    step3：调用函数完成初始化
```

2. NVIC配置

主要配置优先级，以及是否使能中断

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
    @NVIC_InitStruct          结构体指针
                              指向的空间保存了NVIC的配置信息

typedef struct
{
  uint8_t NVIC_IRQChannel;// 指定中断通道（中断号）
    EXTI0_IRQn
    EXTI1_IRQn
    ...
    EXTI4_IRQn
    EXTI9_5_IRQn
    EXTI15_10_IRQn

  uint8_t NVIC_IRQChannelPreemptionPriority;// 抢占优先级，决定是否能抢占
    抢占优先级高的可以打断低的中断处理函数
    0 ~ 15 数字越小，优先级越高

  uint8_t NVIC_IRQChannelSubPriority;// 响应优先级，同时来中断，决定先指向谁
    0 ~ 15 数组越小，优先级越高

  FunctionalState NVIC_IRQChannelCmd;// 使能 或者 失能
    ENABLE
    DISABLE
} NVIC_InitTypeDef;


使用步骤和GPIO_Init() 一样
    step1：定义结构体变量
    step2：给成员变量赋值
    step3：调用函数完成初始化
```

抢占优先级和响应优先级共用4个bits，具体占多少?

```
void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)
// 写在NVIC配置之前，整个程序只允许配置一次
// 建议：写在main函数的第一行
    @NVIC_PriorityGroup      直接填写 NVIC_PriorityGroup_2
```

3. 中断服务函数的编写

```
void PPP_IRQHandler(void)
{

}

// PPP 表示中断源，可以通过启动文件的中断向量表查看 CORE/startup_stm32f40xx.s
```

```
ITStatus EXTI_GetITStatus(uint32_t EXTI_Line);
```

```
// 功能：获取指定中断是否产生
    @EXTI_Line      指定要检测的那个中断的中断请求线
        EXTI_Line0
        EXTI_Line1
        ...
        EXTI_Line15
    @return     SET     产生了中断
                RESET   没产生中断

void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
// 功能：清除指定中断的中断请求
    @EXTI_Line      指定要清除的那个中断的中断请求线
        EXTI_Line0
        EXTI_Line1
        ...
        EXTI_Line15


eg:
    void EXTI0_IRQHandler()
    {
        if(EXTI_GetITStatus(EXTI_Line0) == SET)
        {
            // 证明EXTI0上产生了中断

            // 操作（产生中断后要做的事情）

            // ...

            EXTI_ClearITPendingBit(EXTI_Line0);// 清除中断请求
        }
    }
```

七、练习

使用中断方式，实现按键控灯

```
按下S1控制D1状态反转（使用中断）

void s1_int_init()
{
    // 中断源配置
        // 配置gpio
        // 配置syscfg
        // 配置外部中断控制器

    // NVIC配置
}

// 中断服务函数
void EXTI0_IRQHandler()
{
    if(EXTI_GetITStatus(EXTI_Line0) == SET)
    {
```

```
        // 证明EXTI0上产生了中断

        // 操作（产生中断后要做的事情）

        // ...

        EXTI_ClearITPendingBit(EXTI_Line0);// 清除中断请求
    }
}

void led_gpio_init()
{
    // PF9 PF10 PE13 PE14
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);

    GPIO_InitTypeDef g;
    g.GPIO_Pin = GPIO_Pin_9;
    g.GPIO_Mode = GPIO_Mode_OUT;
    g.GPIO_Speed = GPIO_Speed_50MHz;
    g.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(GPIOF, &g);

    g.GPIO_Pin = GPIO_Pin_10;
    GPIO_Init(GPIOF, &g);

    g.GPIO_Pin = GPIO_Pin_13;
    GPIO_Init(GPIOE, &g);

    g.GPIO_Pin = GPIO_Pin_14;
    GPIO_Init(GPIOE, &g);

    GPIO_SetBits(GPIOF, GPIO_Pin_9|GPIO_Pin_10);
    GPIO_SetBits(GPIOE, GPIO_Pin_13|GPIO_Pin_14);
}

int main()
{
    // 相关初始化
    led_gpio_init();
    s1_int_init();

    while(1);// 卡住程序
}
```