第5章 外部中断的原理与应用

CHAPTER 5

外部中断(EXTernal Interrupt,EXTI)通过GPIO引脚输入外部中断请求信号(上升沿或下降沿),引发中断——CPU在正常执行程序的过程中,由于某个事件的原因,暂停CPU正在执行的程序,转去执行处理该事件的中断服务程序,中断服务程序执行完后,再返回刚才暂停的位置,继续执行刚才被暂停的程序,这个过程叫作“中断”。

5.1 中断的概念

5.1.1 中断的作用

最初中断技术引入计算机系统,是为了消除快速CPU和慢速外设之间进行数据传输时的矛盾。因为慢速外设准备好数据需要很长时间,如果不使用中断技术,就会使快速CPU长时间等待,降低了CPU的效率。而使用中断技术后,在慢速外设准备数据期间,快速CPU可以处理其他事务,待慢速外设准备好数据后,向CPU发出中断请求。CPU收到慢速外设的中断请求后,暂停正在执行的程序,转去接收慢速外设的数据,接收完数据后,再返回刚才暂停处,继续执行刚才暂停的程序,这就提高了CPU的效率。除此之外,中断还具有以下几个功能。

(1) 可以实现多个外设同时工作,提高了效率。

(2) 可以实现实时处理,对采集的信息进行实时处理。

(3) 可以实现故障处理。由于故障是随机事件,事先无法预测,因而中断技术是故障处理的有效方法。

5.1.2 中断的常见术语

(1) 中断源:可以引起中断的事件或设备称为“中断源”。根据中断源不同,中断可分为3类:由计算机本身的硬件异常引起的中断,称为内部异常中断;由CPU执行中断指令引起的中断,称为软件中断或软中断;由外部设备(输入设备/输出设备)请求引起的中断,称为硬件中断或外部中断。

(2) 中断请求、中断响应、中断服务、中断返回:中断请求就是中断源对CPU发出处理中断的要求;中断响应就是CPU转去执行中断服务程序;中断服务就是CPU执行中断服务程序的过程;中断返回就是CPU执行完中断服务程序后,返回响应中断时暂停的位置。

一个完整的中断处理过程包含了中断请求、中断响应、中断服务和中断返回。

(3) 中断服务程序和中断向量：处理中断的程序叫中断服务程序。中断服务程序的入口(起始)地址叫中断向量。

(4) 中断的优先级：当多个中断源同时发出中断请求时，需要设置一个优先权等级以决定 CPU 响应中断请求(执行对应的中断服务程序)的先后顺序，这个优先权等级就是中断的优先级。

(5) 中断嵌套：一个低中断优先级的中断在执行过程中，可以被高中断优先级的中断打断——CPU 暂停正在执行的低优先级中断，转去执行高优先级的中断，高优先级中断执行完后，返回刚才暂停处继续执行低优先级中断，这个过程叫中断嵌套。

(6) 中断系统：实现中断处理功能的软件、硬件系统。

5.2　NVIC 中断管理

Cortex-M3 内核支持 256 个中断，其中包含了 16 个内核中断和 240 个外部中断，并且具有 256 级的可编程中断设置。STM32F103 系列有 60 个可屏蔽中断(在 STM32F107 系列才有 68 个)，这么多个中断是如何管理的呢？

5.2.1　抢占优先级和响应优先级

STM32 的中断优先级有两种：一种是“抢占优先级”，另一种是“响应优先级”。优先级数值越小，优先级越高。抢占优先级具有“抢占”的属性，即“高抢占优先级”的中断可以打断“低抢占优先级”的中断。而响应优先级只有“响应”的属性，即这种优先级只影响哪个中断优先被响应。但要注意，由于抢占优先级可以打断其他中断，所以哪个中断先被响应是“抢占优先级”起决定作用，因为即使一个中断因为高的响应优先级而先被响应(CPU 先去执行该中断的处理程序)了，也会因为低抢占优先级而被其他中断打断，实际还是先处理了高抢占优先级的中断。

例如，假定设置中断 3(RTC 中断)的抢占优先级为 2，响应优先级为 1。中断 6(外部中断 0)的抢占优先级为 3，响应优先级为 0。中断 7(外部中断 1)的抢占优先级为 2，响应优先级为 0。那么这 3 个中断的优先级顺序为：中断 7 >中断 3 >中断 6。

5.2.2　中断优先级分组

中断分为 5 组，如表 5.1 所示。STM32 使用中断优先级控制的寄存器组 IP[240](全称是 Interrupt Priority Register)设置中断的优先级，每个中断使用一个寄存器来确定优先级。STM32F103 系列一共有 60 个可屏蔽中断，使用 IP[59]～IP[0]。每个 IP 寄存器的高 4 位用来设置抢占和响应优先级的等级，低 4 位没有用到。那么在这高 4 位中，用多少位设置抢占优先级的等级，多少位用来设置响应优先级的等级？这是由“中断优先级分组”决定的。

表 5.1 中断优先级分组表

分 组	IP[7:4]分配情况
0	0 位抢占优先级,4 位响应优先级
1	1 位抢占优先级,3 位响应优先级
2	2 位抢占优先级,2 位响应优先级
3	3 位抢占优先级,1 位响应优先级
4	4 位抢占优先级,0 位响应优先级

当设置中断优先级分组为 2 组时,在 IP 寄存器中,抢占优先级和响应优先级都是 2 位来设置,因此,两个优先级等级范围都为 0～3,那么编程设置时,两个优先级等级都不能超过 3。

另外要注意,一旦中断优先级分组设置好了,也就意味着 IP 寄存器中设置抢占和响应优先级的位数分配好了,在程序中就不要再随意改动了。因为优先级分组一改,会造成程序中设置好的中断的优先级都跟着起变化,进而引起程序执行错误。

5.2.3 NVIC 中断管理相关函数

在 misc.h 中与 NVIC 相关的库函数声明的函数主要有两个:优先级分组的函数 NVIC_PriorityGroupConfig()和设置中断优先级的函数 NVIC_Init()。示例代码如下:

```
NVIC_InitTypeDef  NVIC_InitStruct;
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
NVIC_InitStruct.NVIC_IRQChannel = TIM3_IRQn;
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1;
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 2;
NVIC_Init(&NVIC_InitStruct);
```

其中,中断优先级分组被设置为分组 2,NVIC_InitStruct.NVIC_IRQChannel 的值是中断请求的名字,在 stm32f10x.h 文件中定义了所有中断请求的名字。

- VIC_InitStruct.NVIC_IRQChannelPreemptionPriority 指抢占优先级的等级。
- VIC_InitStruct.NVIC_IRQChannelSubPriority 指响应优先级的等级。

5.3 EXTI 外部中断

5.3.1 中断请求信号的输入脚

STM32 的每个 GPIO 引脚都可以复用为 EXTI 的外部中断请求信号输入脚。这里的复用是指 STM32 上电复位后引脚的功能不是 EXTI 输入,而是 GPIO 功能,但可以通过程序设置成 EXTI 输入功能。STM32 的中断控制器支持 19 个外部中断/事件请求,其中 0～15 外部中断/事件请求对应外部 I/O 口的输入中断。16 连接到 PVD 输出。17 连接到 RTC 闹钟事件。18 连接到 USB 唤醒事件。即 STM32 的外部中断线只有 0～15 共 16 根。那么 I/O 脚是如何和外部中断线对应起来的呢?

由图 5.1 可以看到,GPIO 引脚和 EXTI 线的映射关系为:GPIOx.0 映射到 EXTI0,GPIOx.1 映射到 EXTI1,……,GPIOx.15 映射到 EXTI15。

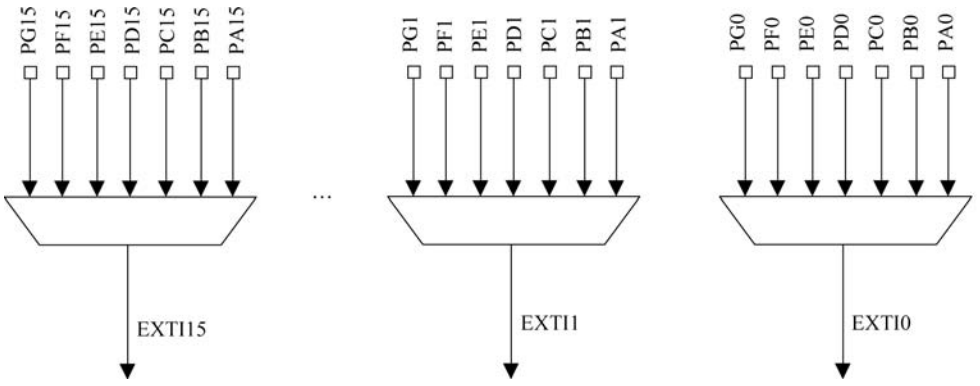

图 5.1　GPIO 脚和外部中断线的映射关系

5.3.2　EXTI 线对应的中断函数

I/O 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，如表 5.2 所示。

表 5.2　中断线与中断函数名对应表

EXTI 线	中断函数名
EXTI0	EXTI0_IRQHandler
EXTI1	EXTI1_IRQHandler
EXTI2	EXTI2_IRQHandler
EXTI3	EXTI3_IRQHandler
EXTI4	EXTI4_IRQHandler
EXTI9_5	EXTI9_5_IRQHandler
EXTI15_10	EXTI15_10_IRQHandler

注意：所有中断函数名都可以在 startup_stm32f10x_md.s 这样的启动文件中找到。

5.4　EXTI 的常用库函数

5.4.1　函数 EXTI_Init()

函数 EXTI_Init()在 stm32f10x_gpio.h 文件中声明，具体描述如表 5.3 所示。

表 5.3　EXTI_Init()函数描述表

函数名	EXTI_Init
函数原型	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”，了解该参数允许的取值范围的更多内容
输出参数	无
返回值	无
先决条件	无
被调用函数	无

EXTI_InitTypeDef 在文件 stm32f10x_exti.h 中定义：

```
typedef struct
{
  u32 EXTI_Line;
  EXTIMode_TypeDef EXTI_Mode;
  EXTIrigger_TypeDef EXTI_Trigger;
  FunctionalState EXTI_LineCmd;
} EXTI_InitTypeDef;
```

参数 EXTI_Mode 设置了被使能线路的模式，参数值如表 5.4 所示。

表 5.4　EXTI_Mode 取值

EXTI_Mode 的值	描　　述
EXTI_Mode_Event	设置 EXTI 线路为事件请求
EXTI_Mode_Interrupt	设置 EXTI 线路为中断请求

参数 EXTI_Trigger 设置了被使能线路的触发边沿，参数值如表 5.5 所示。

表 5.5　EXTI_Trigger 取值

EXTI_Trigger 的值	描　　述
EXTI_Trigger_Falling	设置输入线路下降沿为中断请求
EXTI_Trigger_Rising	设置输入线路上升沿为中断请求
EXTI_Trigger_Rising_Falling	设置输入线路上升沿和下降沿为中断请求

相关参数的定义实例代码如下：

```
EXTI_InitStruct.EXTI_Line = EXTI_Line11|EXTI_Line13;
EXTI_InitStruct.EXTI_LineCmd = ENABLE;
EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_Init(&EXTI_InitStruct);
```

5.4.2　函数 GPIO_EXTILineConfig()

这个函数的声明在 stm32f10x_gpio.h 文件中，具体描述如表 5.6 所示。

表 5.6　GPIO_EXTILineConfig()函数描述表

函数名	GPIO_EXTILineConfig
函数原型	void GPIO_EXTILineConfig(u8 GPIO_PortSource,u8 GPIO_PinSource)
功能描述	选择 GPIO 引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 参阅“Section：GPIO_PortSource”，了解该参数允许的取值范围的更多内容
输入参数 2	EXTI_PinSource：待设置的外部中断线路 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

函数调用实例代码如下：

```
/* Selects PB.8 as EXTI Line 8 */
GPIO_EXTILineConfig(GPIO_PortSource_GPIOB, GPIO_PinSource8);
```

5.5　EXTI 的应用实例

5.5.1　EXTI 的初始化步骤

(1) 使能 EXTI 线所在的 GPIO 时钟和 AFIO 复用时钟。
(2) 初始化 EXTI 线所在的 GPIO 的输入输出模式。
(3) 将 GPIO 脚映射到对应的 EXTI 线上。
(4) 设置 NVIC 优先级分组，初始化 NVIC。
(5) 初始化 EXTI。

5.5.2　EXTI 应用实例

【例 5.1】 实现图 5.2 所示按键控制 LED 灯发光花样。

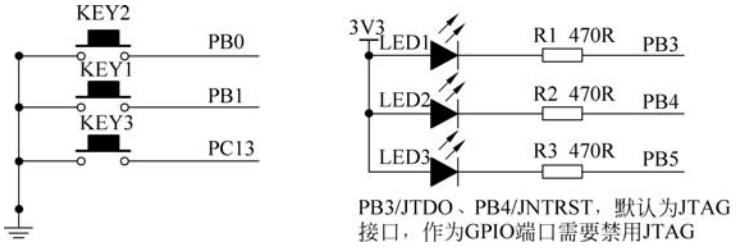

图 5.2　例 5.1 程序对应电路图

```
#include "stm32f10x.h"
#include "exti.h"
#include "led.h"
#define bitband(A,n) *((volatile u32 *)((A&0xf0000000)+0x2000000+((((A&0xfffff)<<3)+
n)<<2)))
#define GPIOB_ODR_Addr (GPIOB_BASE+0x0C)
#define PBout(n) bitband(GPIOB_ODR_Addr,n)
void exti_init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct;
 EXTI_InitTypeDef EXTI_InitStruct;
 NVIC_InitTypeDef NVIC_InitStruct;

 RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO|RCC_APB2Periph_GPIOB
  |RCC_APB2Periph_GPIOC,ENABLE);

 GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IPU;
 GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0|GPIO_Pin_1;
 GPIO_Init(GPIOB,&GPIO_InitStruct);
```

```
 GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IPU;
 GPIO_InitStruct.GPIO_Pin = GPIO_Pin_13;
 GPIO_Init(GPIOC,&GPIO_InitStruct);

 //GPIO_EXTILineConfig 函数声明在 stm32f10x_gpio.h 中
 GPIO_EXTILineConfig(GPIO_PortSourceGPIOB,GPIO_PinSource0);
 GPIO_EXTILineConfig(GPIO_PortSourceGPIOB,GPIO_PinSource1);
 GPIO_EXTILineConfig(GPIO_PortSourceGPIOC,GPIO_PinSource13);

 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);        //NVIC 相关函数声明在 misc.h 中
 NVIC_InitStruct.NVIC_IRQChannel = EXTI0_IRQn;          //中断请求名在 stm32f10x.h 中
 NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
 NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0;
 NVIC_InitStruct.NVIC_IRQChannelSubPriority = 2;
 NVIC_Init(&NVIC_InitStruct);

 NVIC_InitStruct.NVIC_IRQChannel = EXTI1_IRQn;
 NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
 NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1; //抢占优先级
 NVIC_InitStruct.NVIC_IRQChannelSubPriority = 2;        //响应优先级
 NVIC_Init(&NVIC_InitStruct);

 NVIC_InitStruct.NVIC_IRQChannel = EXTI15_10_IRQn;
 NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
 NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 2;
 NVIC_InitStruct.NVIC_IRQChannelSubPriority = 2;
 NVIC_Init(&NVIC_InitStruct);

 EXTI_InitStruct.EXTI_Line = EXTI_Line0|EXTI_Line1;
 EXTI_InitStruct.EXTI_LineCmd = ENABLE;
 EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
 EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling;
 EXTI_Init(&EXTI_InitStruct);

 EXTI_InitStruct.EXTI_Line = EXTI_Line13;
 EXTI_InitStruct.EXTI_LineCmd = ENABLE;
 EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
 EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling;
 EXTI_Init(&EXTI_InitStruct);

}

void EXTI0_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line0) == SET)
    water_led_register();
  EXTI_ClearITPendingBit(EXTI_Line0);
}

void EXTI1_IRQHandler(void)
{
```

```
  if(EXTI_GetITStatus(EXTI_Line1) == SET)
    blink_register();
  EXTI_ClearITPendingBit(EXTI_Line1);
}

void EXTI15_10_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line13) == SET)
    centerflower_bitband();
  EXTI_ClearITPendingBit(EXTI_Line13);
}

void Delay(u32 count)
{
    u32 i = 0;
    for(; i < count; i++);
}

/*************************************************************************
GPIO 初始化函数步骤:
1.使能 PB 时钟、AFIO 时钟
2.失能 JTAG/SWD 下载功能以恢复 PB3,PB4 的 IO 功能
3.初始化 PB3,PB4,PB5 为推挽输出
*************************************************************************/
void led_init_register(void)
{
 RCC->APB2ENR |= (0x01|(0x01 << 3));
 AFIO->MAPR = (0x4 << 24);
 GPIOB->CRL |= (0x01 << 12)|(0x01 << 16)|(0x01 << 20);

}

void led_init_libfunc(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB|RCC_APB2Periph_AFIO,ENABLE);
  GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable,ENABLE);

  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_10MHz;
  GPIO_Init(GPIOB,&GPIO_InitStruct);
}

void water_led_register(void)
{
   char i = 4;
  //使用寄存器控制 GPIOB 输出
      while(i!= 0)
        {
```

```
            GPIOB->ODR = 0xffff;
            GPIOB->ODR &= ~(0x01<<3);
            Delay(3000000);

            GPIOB->ODR = 0xffff;
            GPIOB->ODR &= ~(0x01<<4);
            Delay(3000000);

            GPIOB->ODR = 0xffff;
            GPIOB->ODR &= ~(0x01<<5);
            Delay(3000000);

            i--;
          }
}

void blink_register(void)
{
  char i=4;
  while(i!=0)
        {
          GPIOB->BRR = (0x01<<3)|(0x01<<4)|(0x01<<5);
          Delay(3000000);

          GPIOB->BSRR = (0x01<<3)|(0x01<<4)|(0x01<<5);
          Delay(3000000);

          GPIOB->BSRR = ((0x01<<3)|(0x01<<4)|(0x01<<5))<<16;
          Delay(3000000);

          GPIOB->BSRR = (0x01<<3)|(0x01<<4)|(0x01<<5);
          Delay(3000000);

          i--;
        }

}

void blink2water_libfunc(void)
{
  char i=4;
  //使用库函数控制 GPIOB 输出
      while(i!=0)
        {
          GPIO_SetBits(GPIOB,GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5);
          Delay(3000000);

          GPIO_ResetBits(GPIOB,GPIO_Pin_3);
          Delay(3000000);

          GPIO_SetBits(GPIOB,GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5);
```

```
            Delay(3000000);

            GPIO_ResetBits(GPIOB,GPIO_Pin_3);
            Delay(3000000);

            GPIO_WriteBit(GPIOB,GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5,Bit_SET);
            Delay(3000000);

            GPIO_WriteBit(GPIOB,GPIO_Pin_4,Bit_RESET);
            Delay(3000000);

            GPIO_WriteBit(GPIOB,GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5,Bit_SET);
            Delay(3000000);

            GPIO_WriteBit(GPIOB,GPIO_Pin_4,Bit_RESET);
            Delay(3000000);

            GPIO_Write(GPIOB,0xffff);
            Delay(3000000);

            GPIO_Write(GPIOB,~(0x01<<5));
            Delay(3000000);

            GPIO_Write(GPIOB,0xffff);
            Delay(3000000);

            GPIO_Write(GPIOB,~(0x01<<5));
            Delay(3000000);

            i--;
          }
}

void centerflower_bitband(void)
{
  char i=4;
  //使用位带操作控制 GPIOB 输出
          while(i!=0)
            {
               PBout(3)=1;
               PBout(4)=1;
               PBout(5)=1;
               Delay(3000000);

               PBout(3)=1;
               PBout(4)=0;
               PBout(5)=1;
               Delay(3000000);

               PBout(3)=0;
               PBout(4)=1;
```

```
            PBout(5) = 0;
            Delay(3000000);

            i--;
        }
}


int main(void)
{
   led_init_libfunc();
   exti_init();

  while(1)
  blink2water_libfunc();
}
```