第5章 CHAPTER 5 按键与中断处理

本章将介绍嵌套向量中断控制器 NVIC 的工作原理，阐述 STM32F103ZET6 微控制器外部输入中断的工作原理，然后，以用户按键为例，详细解释 NVIC 中断的寄存器类型和库函数类型的程序设计方法。

本章的学习目标：

- 了解 NVIC 中断响应方法；
- 熟悉 GPIO 中断响应方法；
- 熟练应用寄存器或库函数进行 GPIO 中断程序设计。

5.1 NVIC 中断工作原理

嵌套向量中断控制器 NVIC 相关的中断管理工作主要有开放中断、关闭中断、设置中断请求标志、读中断请求标志、清除中断请求标志和配置中断优先级等。嵌套向量中断控制器 NVIC 的寄存器有 ISER0、ISER1、ICER0、ICER1、ISPR0、ISPR1、ICPR0、ICPR1、IABR0、IABR1、IPR0～IPR14 和 STIR，如表 5-1 所示。

表 5-1 NVIC 寄存器

序号	地 址	寄存器	名 称	描 述
1	0xE000E100	ISER0	中断开放寄存器	ISER0[0]～ISER0[31]、ISER1[0]～ISER1[27]依次对应中断号为 0～59 的中断，各位写 0 无效，写 1 开放中断
	0xE000E104	ISER1		
2	0xE000E180	ICER0	中断关闭寄存器	ICER0[0]～ICER0[31]、ICER1[0]～ICER1[27]依次对应中断号为 0～59 的中断，各位写 0 无效，写 1 关闭中断
	0xE000E184	ICER1		
3	0xE000E200	ISPR0	中断设置请求状态寄存器	ISPR0[0]～ISPR0[31]、ISPR1[0]～ISPR1[27]依次对应中断号为 0～59 的中断，各位写 0 无效，写 1 请求中断
	0xE000E204	ISPR1		
4	0xE000E280	ICPR0	中断清除请求状态寄存器	ICPR0[0]～ICPR0[31]、ICPR1[0]～ICPR1[27]依次对应中断号为 0～59 的中断，各位写 0 无效，写 1 清除中断标志
	0xE000E284	ICPR1		

续表

序号	地　址	寄存器	名　称	描　述
5	0xE000E300	IABR0	中断活跃位寄存器(只读)	IABR0[0]～IABR0[31]、IABR1[0]～IABR1[27]依次对应中断号为0～59的中断,各位读出1,表示相应中断活跃
	0xE000E304	IABR1		
6	0xE000E400～0xE000E438	IPR0～IPR14	中断优先级寄存器	共有16个优先级,优先级号从0～15,优先级号0表示的优先级最高,优先级号15表示的优先级最低
7	0xE000EF00	STIR	软件触发中断寄存器	第[8:0]位域有效,写入0～59中的某一中断号,则触发相应的中断

下面以ISER0和ISER1为例,介绍开放中断的方法。

根据表5-1,ISER0[0]～ISER0[31]对应中断号为0～31的NVIC中断,而ISER1[0]～ISER1[27]对应中断号为32～59的NVIC中断。由表2-5可知,外部中断2的中断号为8,而USART2中断的中断号为38,开放这两个中断的语句依次为:

```
ISER0 = (1uL << 8);
ISER1 = (1uL << 6);
```

设中断号为IRQn,则这两个语句也可以写为如下统一的语句形式:

```
ISER0 = 1uL << (IRQn & 0x1F);
ISER1 = 1uL << (IRQn & 0x1F);
```

上述开放中断的方法被用在CMSIS库文件中。

在CMSIS库头文件core_cm3.h中定义了NVIC中断的相关操作,这里重点介绍开放中断、关闭中断、设置中断请求标志、读中断请求标志、清除中断请求标志、设置中断优先级和获取中断优先级的函数,如程序段5-1所示。

程序段5-1　NVIC中断相关的CMSIS库函数(摘自core_cm3.h文件)

```
typedef struct
{
  __IO uint32_t ISER[8U];       //偏移地址:0x000(可读/可写) 中断设置使能寄存器
       uint32_t RESERVED0[24U];
  __IO uint32_t ICER[8U];       //偏移地址:0x080(可读/可写) 中断清除使能寄存器
       uint32_t RSERVED1[24U];
  __IO uint32_t ISPR[8U];       //偏移地址:0x100(可读/可写) 中断设置请求寄存器
       uint32_t RESERVED2[24U];
  __IO uint32_t ICPR[8U];       //偏移地址:0x180(可读/可写) 中断清除请求寄存器
       uint32_t RESERVED3[24U];
  __IO uint32_t IABR[8U];       //偏移地址:0x200(可读/可写) 中断活跃标志位寄存器
       uint32_t RESERVED4[56U];
  __IO uint8_t  IP[240U];       //偏移地址:0x300(可读/可写) 中断优先级寄存器(8位)
       uint32_t RESERVED5[644U];
  __O  uint32_t STIR;           //偏移地址:0xE00(只写) 软件触发中断寄存器
}  NVIC_Type;

#define SCS_BASE              (0xE000E000UL)
#define NVIC_BASE             (SCS_BASE +   0x0100UL)
#define NVIC                  ((NVIC_Type    * )NVIC_BASE)

```

第 1～16 行自定义结构体类型 NVIC_Type，各成员的位置与表 5-1 中各个寄存器的位置对应，再结合第 18～20 行可知，NVIC 为指向首地址 0xE000E100 的结构体指针，这样（结合表 5-1），NVIC-> ISER[0]指向的地址即为 ISER0 寄存器的地址，NVIC-> ISER[1]指向的地址即为 ISER1 寄存器的地址，以此类推，NVIC-> STIR 指向的地址即为 STIR 寄存器的地址。

```
22    __STATIC_INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)  // 开中断
23    {
24      NVIC->ISER[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F));
25    }
26
```

第 22～25 行为开放 NVIC 中断函数 NVIC_EnableIRQ，形参为 IRQn_Type 类型的变量，该自定义类型定义在 stm32f10x.h 文件中，如程序段 5-2 所示。第 24 行根据 IRQn 的值设置 ISER[0]或 ISER[1]相应的位，即开放 IRQn 对应的 NVIC 中断。

```
27    __STATIC_INLINE void NVIC_DisableIRQ(IRQn_Type IRQn) // 关中断
28    {
29      NVIC->ICER[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F));
30    }
31
```

第 27～30 行为关闭 NVIC 中断函数 NVIC_DisableIRQ，形参为 IRQn_Type 类型的变量。第 29 行根据 IRQn 的值向 ICER[0]或 ICER[1]相应的位写入 1，关闭 IRQn 对应的 NVIC 中断。

```
32    __STATIC_INLINE void NVIC_SetPendingIRQ(IRQn_Type IRQn)// 中断请求
33    {
34      NVIC->ISPR[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F));
35    }
36
```

第 32～35 行为设置中断请求标志的函数 NVIC_SetPendingIRQ，形参为 IRQn_Type 类型的变量。第 34 行根据 IRQn 的值向 ISPR[0]或 ISPR[1]相应的位写入 1，设置 IRQn 对应的 NVIC 中断请求标志，使该 NVIC 中断处于请求态。

```
37    __STATIC_INLINE uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)  // 处于请求态时返回 1
38    {                                                            //否则返回 0
39      return((uint32_t) ((NVIC->ISPR[(uint32_t)(IRQn) >> 5] & (1 << ((uint32_t)(IRQn) &
        0x1F)))?1:0));
40    }
41
```

第 37～40 行为获取 NVIC 中断请求状态的函数 NVIC_GetPendingIRQ，形参为 IRQn_Type 类型的变量。第 39 行根据 IRQn 的值读出它对应的 ISPR[0]或 ISPR[1]的位，如果 IRQn 中断处于请求态，则返回 1；否则返回 0。

```
42    __STATIC_INLINE void NVIC_ClearPendingIRQ(IRQn_Type IRQn) // 清除中断请求标志
43    {
44      NVIC->ICPR[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F));
45    }
46
```

第42～45行为清除NVIC中断请求标志的函数NVIC_ClearPendingIRQ,形参为IRQn_Type类型的变量。第44行根据IRQn的值向ICPR[0]或ICPR[1]相应的位写入1,清除IRQn对应的NVIC中断标志。

```
47    __STATIC_INLINE void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
48    {
49      if((int32_t)IRQn < 0)
50      { //为 Cortex-M3 系统异常:MemManage,BusFault,UsageFault,SVC,DebugMon 设定优先级
51        SCB->SHP[((uint32_t)(IRQn) & 0xF)-4] = ((priority << (8 - __NVIC_PRIO_BITS)) & 0xff);
52      }
53      else
54      { // 为中断: IRQn,n = 0～59 设定优先级
55        NVIC->IP[(uint32_t)(IRQn)] = ((priority << (8 - __NVIC_PRIO_BITS)) & 0xff);
56      }
57    }
```

第47～57行为设置异常和中断优先级的函数NVIC_SetPriority,形参有两个:(1)IRQn_Type类型的变量IRQn为中断号;(2)无符号32位整型变量priority为设置的优先级号数值。第49～52行设置中断号为－12～－1的异常的优先级号;第53～56行设置中断号为0～59的中断的优先级号。这里的“__NVIC_PRIO_BITS”为宏定义的常数4,因此,priority的取值为0～15。注意:中断优先级号小的中断具有较高的优先级;如果有多个中断被设置为相同的优先级,则中断号小的中断优先级高。

```
58    __STATIC_INLINE uint32_t NVIC_GetPriority(IRQn_Type IRQn)
59    {
60      if ((int32_t)(IRQn) < 0)
61      {
62        return(((uint32_t)SCB->SHP[(((uint32_t) IRQn) & 0xFuL)-4UL] >> (8U - __NVIC_
PRIO_BITS)));
63      }
64      else
65      {
66         return(((uint32_t)NVIC->IP[((uint32_t) IRQn)] >> (8U - __NVIC_PRIO_BITS)));
67      }
68    }
```

第58～68行为获取异常和中断优先级的函数NVIC_GetPriority,形参为IRQn_Type类型的变量IRQn,返回值为中断的优先级号。对于中断号小于0的异常,第60～63行获取异常的优先级号;否则,第65～67行获取中断号为0～59的NVIC中断的优先级号。

程序段5-1中,第47～68行的代码需要访问中断优先级寄存器IPR0～IPR14,这些寄存器的结构如图5-1所示。

由图5-1可知,每个IPR寄存器用于设置4个NVIC中断的优先级,32位的IPR寄存器的4字节的低4位均无效,只有高4位有效,故可以设置的优先级号为0～15。根据图5-1,如果设置EXTI2中断的优先级号为10,则需要将IPR2的第[7:4]位域设为10。当两个中断具有不同的优先级号时,优先级号小的中断优先级高;当两个中断具有相同的优先级号时,中断号小的中断优先级高。

可配置优先级的异常的优先级号由3个系统手柄优先级寄存器(SHPR1～SHPR3)设置,其地址依次为0xE000ED18、0xE000ED1C和0xE000ED20,如表5-2所示。

位号	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
IPR0	RTC	TAMPER	PVD	WWDG
IPR1	EXTI1	EXTI0	RCC	FLASH
IPR2	DMA1_Ch1	EXTI4	EXTI3	EXTI2
IPR3	DMA1_Ch5	DMA1_Ch4	DMA1_Ch3	DMA1_Ch2
IPR4	USB_HP	ADC1_2	DMA1_Ch7	DMA1_Ch6
IPR5	EXTI9_5	CAN_SCE	CAN_RX1	USB_LP
IPR6	TIM1_CC	TIM1_TRG	TIM1_UP	TIM1_BRK
IPR7	I2C1_EV	TIM4	TIM3	TIM2
IPR8	SPI1	I2C2_ER	I2C2_EV	I2C1_ER
IPR9	USART3	USART2	USART1	SPI2
IPR10	TIM8_BRK	USBWakeUp	RTCAlarm	EXTI15_10
IPR11	ADC3	TIM8_CC	TIM8_TRG	TIM8_UP
IPR12	SPI3	TIM5	SDIO	FSMC
IPR13	TIM7	TIM6	UART5	UART4
IPR14	DMA2_Ch4_5	DMA2_Ch3	DMA2_Ch2	DMA2_Ch1

图 5-1　中断优先级配置寄存器

表 5-2　异常号 4～15 的优先级配置寄存器

序号	异常号	异常名称	位　域	配置名称	寄　存　器
1	4	MemManage	[7:0]	PRI_4	SHPR1
2	5	BusFault	[15:8]	PRI_5	
3	6	UsageFault	[23:16]	PRI_6	
4	7	保留	[31:24]	PRI_7	
5	8	保留	[7:0]	PRI_8	SHPR2
6	9	保留	[15:8]	PRI_9	
7	10	保留	[23:16]	PRI_10	
8	11	SVCall	[31:24]	PRI_11	
9	12	Debug Monitor	[7:0]	PRI_12	SHPR3
10	13	保留	[15:8]	PRI_13	
11	14	PendSV	[23:16]	PRI_14	
12	15	SysTick	[31:24]	PRI_15	

程序段 5-2　自定义枚举类型 IRQn_Type(摘自 stm32f10x.h 文件)

```
typedef enum IRQn
{
// Cortex-M3 处理器异常号
   NonMaskableInt_IRQn       = -14,    // 不可屏蔽中断
   MemoryManagement_IRQn     = -12,    // 4 Cortex-M3 存储器管理异常
   BusFault_IRQn             = -11,    // 5 Cortex-M3 总线出错异常
   UsageFault_IRQn           = -10,    // 6 Cortex-M3 Usage Fault 异常
   SVCall_IRQn               = -5,     // 11 Cortex-M3 SV 调用异常
   DebugMonitor_IRQn         = -4,     // 12 Cortex-M3 调试监测器异常
  PendSV_IRQn                = -2,     // 14 Cortex-M3 请求 SV 中断
  SysTick_IRQn               = -1,     // 15 Cortex-M3 系统节拍定时中断

// STM32 中断号
  WWDG_IRQn                  = 0,      // 加窗看门狗中断
  PVD_IRQn                   = 1,      // PVD 通过 EXTI 侦测中断
  TAMPER_IRQn                = 2,      // Tamper 中断
  RTC_IRQn                   = 3,      // RTC 中断
  FLASH_IRQn                 = 4,      // Flash 中断
  RCC_IRQn                   = 5,      // RCC 中断
  EXTI0_IRQn                 = 6,      //外部中断 0
  EXTI1_IRQn                 = 7,      //外部中断 1
  EXTI2_IRQn                 = 8,      //外部中断 2
  EXTI3_IRQn                 = 9,      //外部中断 3
  EXTI4_IRQn                 = 10,     //外部中断 4
  DMA1_Channel1_IRQn         = 11,     // DMA1 通道 1 中断
  DMA1_Channel2_IRQn         = 12,     // DMA1 通道 2 中断
  DMA1_Channel3_IRQn         = 13,     // DMA1 通道 3 中断
  DMA1_Channel4_IRQn         = 14,     // DMA1 通道 4 中断
  DMA1_Channel5_IRQn         = 15,     // DMA1 通道 5 中断
  DMA1_Channel6_IRQn         = 16,     // DMA1 通道 6 中断
  DMA1_Channel7_IRQn         = 17,     // DMA1 通道 7 中断
  ADC1_2_IRQn                = 18,     // ADC1 和 ADC2 中断
  USB_HP_CAN1_TX_IRQn        = 19,     // USB 设备高优先级中断或 CAN1 发送中断
  USB_LP_CAN1_RX0_IRQn       = 20,     //USB 设备低优先中断或 CAN1 接收 0 中断
  CAN1_RX1_IRQn              = 21,     // CAN1 接收 1 中断
  CAN1_SCE_IRQn              = 22,     // CAN1 SCE 中断
  EXTI9_5_IRQn               = 23,     //外部中断 5～外部中断 9
  TIM1_BRK_IRQn              = 24,     // TIM1 终止中断
  TIM1_UP_IRQn               = 25,     // TIM1 更新中断
  TIM1_TRG_COM_IRQn          = 26,     // TIM1 触发补偿中断
  TIM1_CC_IRQn               = 27,     // TIM1 捕获比较中断
  TIM2_IRQn                  = 28,     // TIM2 中断
  TIM3_IRQn                  = 29,     // TIM3 中断
  TIM4_IRQn                  = 30,     // TIM4 中断
  I2C1_EV_IRQn               = 31,     // I2C1 Event 中断
  I2C1_ER_IRQn               = 32,     // I2C1 Error 中断
  I2C2_EV_IRQn               = 33,     // I2C2 Event 中断
  I2C2_ER_IRQn               = 34,     // I2C2 Error 中断
  SPI1_IRQn                  = 35,     // SPI1 中断
  SPI2_IRQn                  = 36,     // SPI2 中断
  USART1_IRQn                = 37,     // USART1 中断
  USART2_IRQn                = 38,     // USART2 中断
```

```
53      USART3_IRQn                 = 39,      // USART3 中断
54      EXTI15_10_IRQn              = 40,      //外部中断 10～外部中断 15
55      RTCAlarm_IRQn               = 41,      // RTC 报警中断(借助外部中断)
56      USBWakeUp_IRQn              = 42,      // USB 设备唤醒中断(借助外部中断)
57      TIM8_BRK_IRQn               = 43,      // TIM8 终止中断
58      TIM8_UP_IRQn                = 44,      // TIM8 更新中断
59      TIM8_TRG_COM_IRQn           = 45,      // TIM8 触发补偿中断
60      TIM8_CC_IRQn                = 46,      // TIM8 捕获比较中断
61      ADC3_IRQn                   = 47,      // ADC3 中断
62      FSMC_IRQn                   = 48,      // FSMC 中断
63      SDIO_IRQn                   = 49,      // SDIO 中断
64      TIM5_IRQn                   = 50,      // TIM5 中断
65      SPI3_IRQn                   = 51,      // SPI3 中断
66      UART4_IRQn                  = 52,      // UART4 中断
67      UART5_IRQn                  = 53,      // UART5 中断
68      TIM6_IRQn                   = 54,      // TIM6 中断
69      TIM7_IRQn                   = 55,      // TIM7 中断
70      DMA2_Channel1_IRQn          = 56,      // DMA2 通道 1 中断
71      DMA2_Channel2_IRQn          = 57,      // DMA2 通道 2 中断
72      DMA2_Channel3_IRQn          = 58,      // DMA2 通道 3 中断
73      DMA2_Channel4_5_IRQn        = 59       // DMA2 通道 4 和通道 5 中断
74    } IRQn_Type;
```

上述代码对应表 2-5，由于 IRQn_Type 为指定了成员值的枚举类型，因此，可以用强制类型转换将 IRQn_Type 类型的变量转化为整型。例如，程序段 5-1 第 24 行的(uint32_t)(IRQn)，就是将 IRQn 转化为无符号 32 位整型。如果 IRQn 为 EXTI2，则(uint32_t)(IRQn)为 8。

现在，就可以直接调用 CMSIS 库中关于中断的函数实现对 NVIC 中断的管理了。例如，关闭 EXTI2 中断、开放 EXTI2 中断和清除 EXTI2 中断标志位的语句依次为：

```
NVIC_DisableIRQ(EXTI2_IRQn);
NVIC_EnableIRQ(EXTI2_IRQn);
NVIC_ClearPendingIRQ(EXTI2_IRQn);
```

5.2　GPIO 外部输入中断

根据寄存器 AFIO_EXTICR1～AFIO_EXTICR4(见表 4-3)，可从 GPIO 口中选择 16 个引脚配置为 16 个外部中断的输入端，如图 5-2 所示。

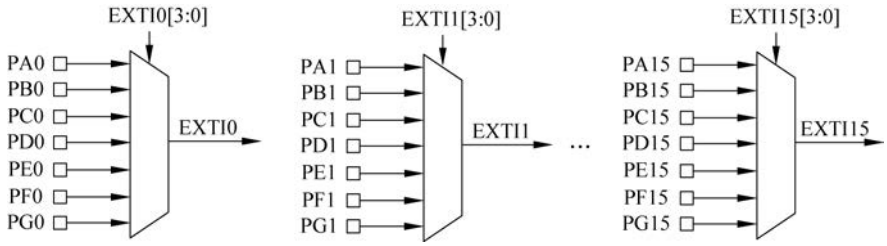

图 5-2　外部中断输入端引脚配置方法

EXTI 模块共有 19 根线路，除了外部中断 EXTI0～EXTI15 外，还有 EXTI16、EXTI17 和 EXTI18，这三根线路分别与 PVD 输出、RTC 报警事件和 USB 唤醒事件相连接。EXTI 模块共有 6 个寄存器，即中断屏蔽寄存器 EXTI_IMR、事件屏蔽寄存器 EXTI_EMR、上升沿

触发选择寄存器 EXTI_RTSR、下降沿触发选择寄存器 EXTI_FTSR、软件触发事件寄存器 EXTI_SWIER 和中断请求寄存器 EXTI_PR。EXTI 模块寄存器的基地址为 0x4001 0400，下面详细介绍各个寄存器的情况。

中断屏蔽寄存器 EXTI_IMR 的偏移地址为 0x0，复位值为 0x0，只有第[18:0]位有效，第 i 位对应 EXTIi，清零表示屏蔽该线路上的中断请求，置 1 表示打开该线路上的中断请求。

事件屏蔽寄存器 EXTI_EMR 的偏移地址为 0x04，复位值为 0x0，只有第[18:0]位有效，第 i 位对应 EXTIi，清零表示屏蔽该线路上的事件请求，置 1 表示打开该线路上的事件请求。

上升沿触发选择寄存器 EXTI_RTSR 的偏移地址为 0x08，复位值为 0x0，只有第[18:0]位有效，第 i 位的名称为 TRi，清零表示关闭上升沿触发中断或事件，置 1 表示打开上升沿触发中断或事件。

下降沿触发选择寄存器 EXTI_FTSR 的偏移地址为 0x0C，复位值为 0x0，只有第[18:0]位有效，第 i 位的名称也记为 TRi，清零表示关闭下降沿触发中断或事件，置 1 表示打开下降沿触发中断或事件。

软件触发事件寄存器 EXTI_SWIER 的偏移地址为 0x10，复位值为 0x0，只有第[18:0]位有效，第 i 位记为 SWIERi，当 EXTIi 中断有效且 SWIERi＝0 时，向 SWIERi 中写入 1，将触发中断请求。

中断请求寄存器 EXTI_PR 的偏移地址为 0x14，只有第[18:0]位有效，第 i 位称为 PRi，如果第 i 个中断触发了，则 PRi 自动置 1，向 PRi 写入 1 清零该位，同时清零 SWIERi 位中的 1。

5.3 用户按键中断实例

结合图 3-11 和图 3-6 可知，STM32F103ZET6 微控制器的 PE2、PE3 和 PE4 分别与网络标号 KEY2、KEY1 和 KEY0 相连接；结合图 3-13 和图 3-3 可知，PB8 与网络标号 BEEP 相连接，控制蜂鸣器的开启与关闭。

本节拟设计工程，实现如下功能：

(1) KEY2 按键作为外部中断 EXTI2 输入端，当按下 KEY2 按键时 LED0 灯点亮；

(2) KEY1 按键作为外部中断 EXTI3 输入端，当按下 KEY1 按键时 LED1 灯熄灭；

(3) KEY0 按键作为外部中断 EXTI4 输入端，当按下 KEY0 按键时，如果蜂鸣器原来是开启的，则关闭蜂鸣器；否则，开启蜂鸣器。


视频讲解

5.3.1 寄存器类型工程实例

在工程 01 的基础上，新建“工程 03”，保存在目录“D:\STM32F103ZET6 工程\工程 03”下，此时的工程 03 与工程 01 完全相同。现在，修改 main.c 和 includes.h 文件，并新建 bsp.c、bsp.h、beep.c、beep.h、key.c、key.h、exti.c 和 exti.h 文件(新建的文件均保存在目录“D:\STM32F103ZET6 工程\工程 03\BSP”下)，然后，将 bsp.c、beep.c、key.c 和 exti.c 文件添加到“BSP”分组下，建设好的工程如图 5-3 所示。

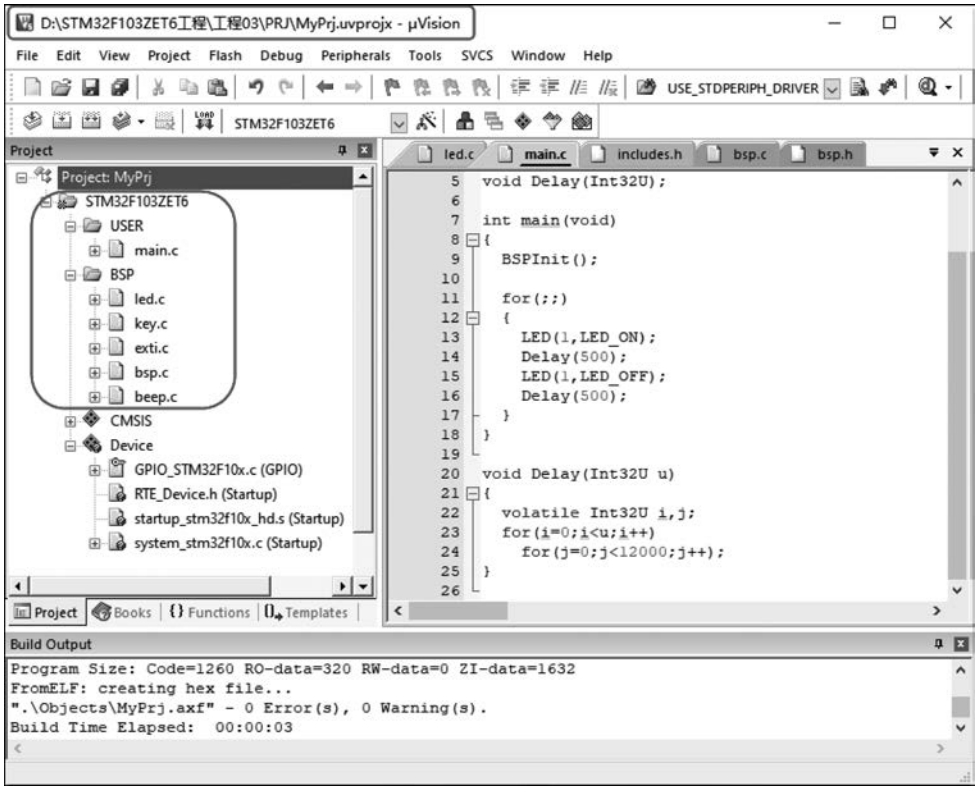

图 5-3　工程 03 工作窗口

修改后的 main. c 和 includes. h 文件如程序段 5-3 和程序段 5-4 所示，新创建的文件 bsp. c、bsp. h、beep. c、beep. h、key. c、key. h、exti. c 和 exti. h 分别如程序段 5-5～程序段 5-12 所示。

程序段 5-3　文件 main. c

```
1   //Filename: main.c
2
3   #include "includes.h"
4
5   void Delay(Int32U);
6
7   int main(void)
8   {
9     BSPInit();
10
11    for(;;)
12    {
13        LED(1,LED_ON);
14        Delay(500);
15        LED(1,LED_OFF);
16        Delay(500);
17    }
18  }
19
20  void Delay(Int32U u)
```

```
21    {
22      volatile Int32U i,j;
23      for(i=0;i<u;i++)
24          for(j=0;j<12000;j++);
25    }
```

对比程序段 4-7,这里的主程序文件 main.c 中,第 9 行调用 BSPInit 函数实现外设的初始化,该函数在程序段 5-5 中介绍;在第 11～17 行的无限循环体中,实现 LED1 灯的循环闪烁功能。

程序段 5-4　文件 includes.h

```
1     //Filename: includes.h
2
3     #include "stm32f10x.h"
4
5     #include "vartypes.h"
6     #include "bsp.h"
7     #include "led.h"
8     #include "key.h"
9     #include "exti.h"
10    #include "beep.h"
```

文件 includes.h 为总的包括头文件,包括了工程 03 中全部的用户程序头文件,如第 5～10 行所示。

程序段 5-5　文件 bsp.c

```
1     //Filename: bsp.c
2
3     #include "includes.h"
4
5     void BSPInit(void)
6     {
7       LEDInit();
8       KEYInit();
9       EXTIKeyInit();
10      BEEPInit();
11    }
```

文件 bsp.c 只有一个函数 BSPInit,在该函数中调用了 LED 初始化函数 LEDInit、按键初始化函数 KEYInit、外部输入中断初始化函数 EXTIKeyInit 和蜂鸣器初始化函数 BEEPInit,如第 7～10 行所示,因此,BSPInit 函数是总的系统初始化函数。

程序段 5-6　文件 bsp.h

```
1     //Filename: bsp.h
2
3     #ifndef  _BSP_H
4     #define  _BSP_H
5
6     void BSPInit(void);
7
8     #endif
```

文件 bsp.h 是文件 bsp.c 对应的头文件,用于声明 bsp.c 中实现的函数。这里第 6 行

声明了函数 BSPInit，该函数定义在文件 bsp.c 中。

程序段 5-7　文件 beep.c

```
1    //Filename: beep.c
2
3    #include "includes.h"
4
5    void BEEPInit(void)
6    {
7      RCC->APB2ENR |= (1uL<<3);    //打开 PB 时钟源
8      GPIOB->CRH |= (1uL<<0);
9      GPIOB->CRH &= ~(7uL<<1);     //PB8 口工作在推挽输出模式下(10MHz)
10
11     GPIOB->ODR &= ~(1uL<<8);     //关闭蜂鸣器
12   }
13
14   void BEEP(void)
15   {
16     GPIOB->ODR ^= (1uL<<8);
17   }
```

文件 beep.c 用于驱动蜂鸣器，包括蜂鸣器初始化函数 BEEPInit 和蜂鸣器工作函数 BEEP。由于 PB8 口与蜂鸣器控制端相连接，所以在初始化函数 BEEPInit 中，应首先打开 PB 口的时钟源(第 7 行)，接着配置 PB8 口工作在推挽输出模式下(第 8～9 行)，然后，使 PB8 输出低电平(第 11 行)，即关闭蜂鸣器。第 14～17 行的函数 BEEP 中，只有第 16 行所示的一条语句，即调用 BEEP 函数时，如果 PB8 原来输出低电平，则输出高电平(蜂鸣器鸣叫)；如果 PB8 原来输出高电平，则输出低电平(关闭蜂鸣器)。

程序段 5-8　文件 beep.h

```
1    //Filename: beep.h
2
3    #ifndef  _BEEP_H
4    #define  _BEEP_H
5
6    void BEEPInit(void);
7    void BEEP(void);
8
9    #endif
```

文件 beep.h 中声明了文件 beep.c 中定义的函数 BEEPInit 和 BEEP。

程序段 5-9　文件 key.c

```
1    //Filename: key.c
2
3    #include "includes.h"
4
5    void KEYInit(void)
6    {
7      RCC->APB2ENR |= (1uL<<6);
8      RCC->APB2ENR |= (1uL<<0);
9
10     GPIOE->CRL &= ~((7uL<<8) | (7uL<<12) | (7uL<<16));
11     GPIOE->CRL |= (1uL<<11) | (1uL<<15) | (1uL<<19);
```

```
12
13      GPIOE->ODR |= (7uL<<2);
14    }
```

文件 key.c 包含了 KEYInit 函数，3 个按键 KEY0～KEY2 依次占用了 PE4、PE3 和 PE2 口，所以第 7 行打开 PE 口的时钟源，第 8 行打开 AFIO 口的时钟源(参考图 3-7)。第 10～11 行配置 PE2～PE4 口为带上拉的输入口，第 13 行使 PE2～PE4 口均输出高电平，相当于为 3 个按键 KEY2、KEY1 和 KEY0 提供上拉电平。

程序段 5-10　文件 key.h

```
1    //Filename: key.h
2
3    #ifndef  _KEY_H
4    #define  _KEY_H
5
6    void KEYInit(void);
7
8    #endif
```

文件 key.h 中声明了 KEYInit 函数，该函数位于 key.c 文件中，用于初始化 3 个按键 KEY0～KEY2。

程序段 5-11　文件 exti.c

```
1    //Filename: exti.c
2
3    #include "includes.h"
4
5    void EXTIKeyInit(void)
6    {
7      AFIO->EXTICR[0] |= (1uL<<2)<<8;
8      AFIO->EXTICR[0] &= ~(((3uL<<0) | (1uL<<3))<<8);
9      AFIO->EXTICR[0] |= (1uL<<2)<<12;
10     AFIO->EXTICR[0] &= ~(((3uL<<0) | (1uL<<3))<<12);
11     AFIO->EXTICR[1] |= (1uL<<2)<<0;
12     AFIO->EXTICR[1] &= ~(((3uL<<0) | (1uL<<3))<<0);
13
```

这里，第 7～8 行将 PE2 口(KEY2)用作外部中断 EXTI2 输入，第 9～10 行将 PE3 口(KEY1)作为外部中断 EXTI3 输入，第 11～12 行将 PE4 口(KEY0)作为外部中断 EXTI4 输入。参考表 4-3 可知，将 PE2 口用作外部中断 EXTI2 输入，即将 EXTI2[3:0]设置为 4；同理，将 PE3 口用作外部中断 EXTI3 输入，即将 EXTI3[3:0]设置为 4，将 PE4 口用作外部中断 EXTI4 输入，即将 EXTI4[3:0]设置为 4。由于 EXTI2[3:0]位于寄存器 AFIO_EXTICR1(即第 7 行的 AFIO->EXTICR[0])的第[11:8]位，所以第 7～8 行中的配置字中出现了“<<8”；同理，可理解第 9～12 行的配置字的含义。

```
14      EXTI->IMR |= (7uL<<2);
15      EXTI->FTSR |= (7uL<<2);
16
```

第 14 行打开外部中断 EXTI2、EXTI3 和 EXTI4，第 15 行配置这 3 个外部中断为下降沿触发。

```
17      NVIC_EnableIRQ(EXTI2_IRQn);
18      NVIC_EnableIRQ(EXTI3_IRQn);
19      NVIC_EnableIRQ(EXTI4_IRQn);
20      NVIC_SetPriority(EXTI2_IRQn,5);
21      NVIC_SetPriority(EXTI3_IRQn,6);
22      NVIC_SetPriority(EXTI4_IRQn,7);
23    }
24
```

第17～19行调用CMSIS库函数NVIC_EnableIRQ开放中断EXTI2、EXTI3和EXTI4。结合第14行可知，外部中断的开放需要两步，首先要配置EXTI模块中的EXTI_IMR寄存器使外部中断的线路有效，然后，还要开放外部中断对应的NVIC中断。第20～22行依次配置NVIC中断EXTI2、EXTI3和EXTI4的优先级号为5、6和7。

因此，第5～23行的函数EXTIKeyInit实现的作用如下：

(1) 将外部按键对应的引脚配置为中断功能。

(2) 开放EXTI模块中的这些外部输入中断。

(3) 配置这些外部输入中断均为下降沿触发类型。

(4) 通过NVIC中断控制器开放这些NVIC中断。

(5) 配置这些外部输入中断的优先级，共有16级，优先级号取值范围为0～15。

```
25    void EXTI2_IRQHandler()
26    {
27      LED(0,LED_ON);
28      EXTI->PR = (1uL<<2);
29      NVIC_ClearPendingIRQ(EXTI2_IRQn);
30    }
31
```

第25～31行为外部中断EXTI2的中断服务函数，函数名必须为EXTI2_IRQHandler(参考2.6节)，当按键KEY2被按下后，将进入该函数执行。第27行点亮LED0灯，第28行清除EXTI2中断标志位，第29行清除NVIC寄存器中EXTI2中断对应的标志位。

```
32    void EXTI3_IRQHandler()
33    {
34      LED(0,LED_OFF);
35      EXTI->PR = (1uL<<3);
36      NVIC_ClearPendingIRQ(EXTI3_IRQn);
37    }
38
```

第32～38行为外部中断EXTI3的中断服务函数，函数名必须为EXTI3_IRQHandler，当按键KEY1被按下后，将进入该函数执行。第34行熄灭LED0灯，第35行清除EXTI3中断标志位，第36行清除NVIC寄存器中EXTI3中断对应的标志位。

```
39    void EXTI4_IRQHandler()
40    {
41      BEEP();
42      EXTI->PR = (1uL<<4);
43      NVIC_ClearPendingIRQ(EXTI4_IRQn);
44    }
```

第 39～44 行为外部中断 EXTI4 的中断服务函数，函数名必须为 EXTI4_IRQHandler，当按键 KEY0 被按下后，将进入该函数执行。第 41 行调用 BEEP 函数，如果蜂鸣器原来是关闭的，则打开蜂鸣器；否则，关闭蜂鸣器。第 42 行清除 EXTI4 中断标志位，第 43 行清除 NVIC 寄存器中 EXTI4 中断对应的标志位。

程序段 5-12　文件 exti.h

```
1    //Filename: exti.h
2
3    #ifndef  _EXTI_H
4    #define  _EXTI_H
5
6    void EXTIKeyInit(void);
7
8    #endif
```

文件 exti.h 中声明了函数 EXTIKeyInit，该函数定义在 exti.c 中，用于初始化外部输入中断。注意，文件 exti.c 中的 3 个中断服务函数 EXTI2_IRQHandler、EXTI3_IRQHandler 和 EXTI4_IRQHandler 均无须声明，因为这些中断服务函数不是由主函数或其他函数调用执行的，而是由硬件的中断系统被触发相应的中断后自动调用的。

工程 03 的工作流程如图 5-4 所示。

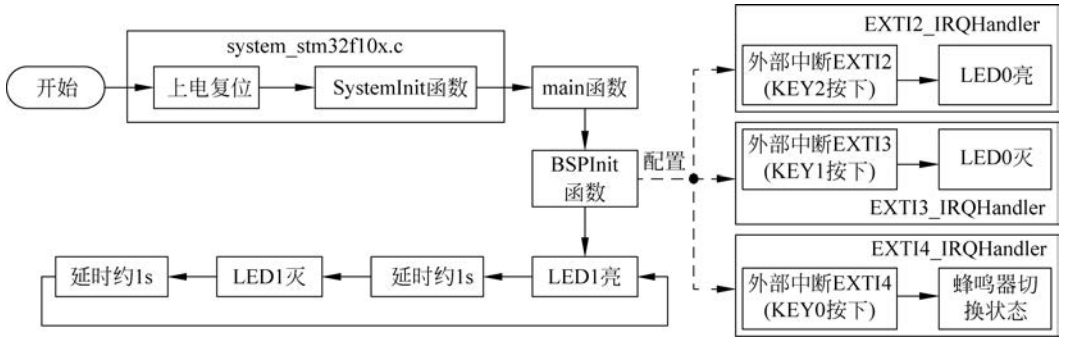

图 5-4　工程 03 的工作流程

由图 5-4 可知，工程 03 运行到主函数 main 后，执行 BSPInit 函数初始化 LED 灯、按键、蜂鸣器和外部中断等外设，然后进行无限循环体，执行 LED1 灯的循环闪烁功能。工程 03 中有 3 个中断服务函数，当按键 KEY02 被按下时，执行 EXTI2_IRQHandler 函数，点亮 LED0 灯；当按键 KEY1 被按下时，执行 EXTI3_IRQHandler 函数，熄灭 LED0 灯；当按键 KEY0 被按下时，执行 EXTI4_IRQHandler 函数，使蜂鸣器切换工作状态。


视频讲解

5.3.2　库函数类型工程实例

本节讨论的工程与 5.3.1 节的工程 03 实现的功能完全相同，这里使用库函数方式进行工程设计。在工程 02 的基础上，新建“工程 04”，保存在目录“D:\STM32F103ZET6 工程\工程 04”下，此时的工程 04 与工程 02 完全相同，需要做的修改如下：

(1) 修改文件 main.c 和 includes.h。

(2) 新建文件 bsp.c、bsp.h、key.c、key.h、beep.c、beep.h、exti.c 和 exti.h，新建的文件均保存在目录“D:\STM32F103ZET6 工程\工程 04\BSP”下。

（3）将 bsp.c、key.c、beep.c 和 exti.c 文件添加到工程管理器的“BSP”分组下。

（4）将位于目录“D:\STM32F103ZET6 工程\工程 04\STM32F10x_FWLib\src”下的库文件 stm32f10x_exti.c 添加到工程管理器的“LIB”分组下。

建设好的工程 04 如图 5-5 所示。

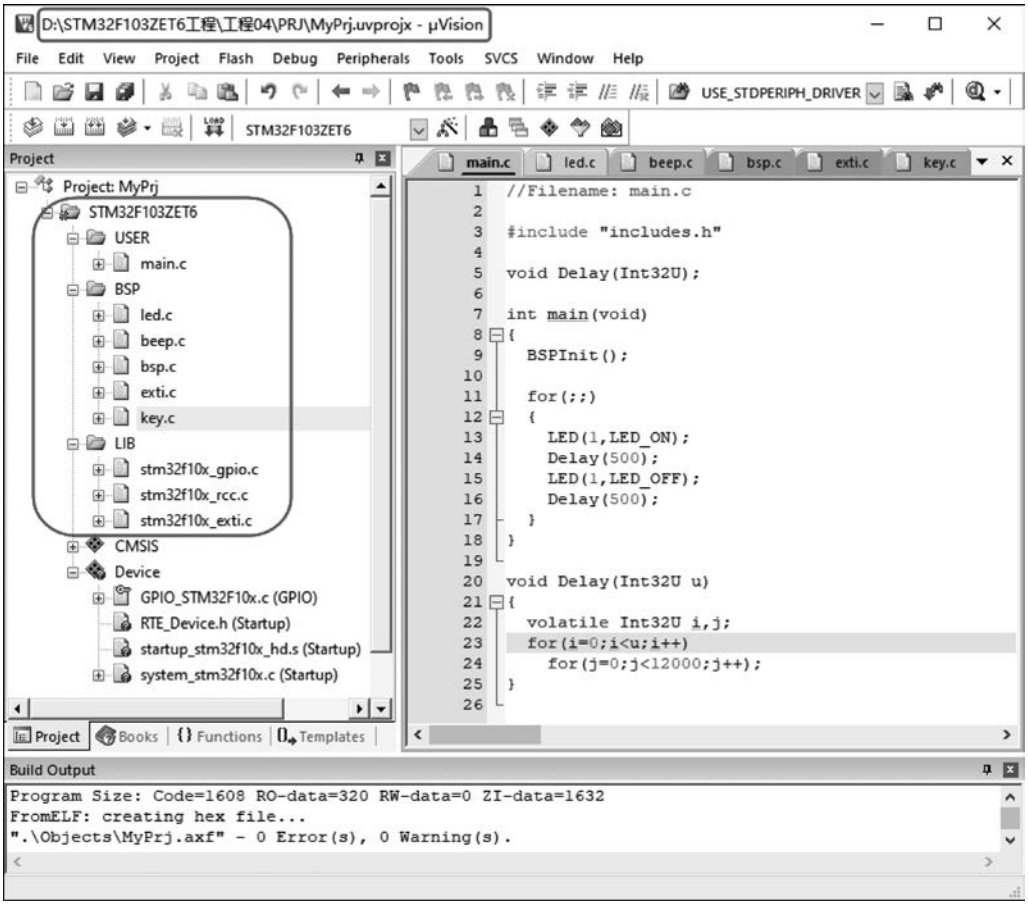

图 5-5　工程 04 工作窗口

工程 04 中的文件 vartypes.h、led.c 和 led.h 与工程 02 中的同名文件源代码相同；工程 04 中的文件 main.c、includes.h、bsp.c、bsp.h、exti.h、key.h、beep.h 与工程 03 中的同名文件的内容相同。下面介绍其余文件的内容，即 beep.c、key.c 和 exti.c 文件的内容，如程序段 5-13～程序段 5-15 所示。

程序段 5-13　文件 beep.c

```
1    //Filename: beep.c
2
3    #include "includes.h"
4
5    void BEEPInit(void)
6    {
7      GPIO_InitTypeDef g;
8      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
9
```

```
10      g.GPIO_Pin = GPIO_Pin_8;
11      g.GPIO_Mode = GPIO_Mode_Out_PP;
12      g.GPIO_Speed = GPIO_Speed_10MHz;
13      GPIO_Init(GPIOB, &g);
14
15      GPIO_ResetBits(GPIOB,GPIO_Pin_8);
16    }
17
```

第5～16行为蜂鸣器的初始化函数。蜂鸣器的控制端为PB8口，第8行调用库函数RCC_APB2PeriphClockCmd打开PB口的时钟源；第10～12行为配置PB8口的属性为推挽输出且工作频率为10MHz，第13行调用库函数GPIO_Init初始化PB8口。

```
18    void BEEP(void)
19    {
20      Int08U v;
21      v = GPIO_ReadOutputDataBit(GPIOB,GPIO_Pin_8);
22      if(v == 1)
23          GPIO_ResetBits(GPIOB,GPIO_Pin_8);
24      else
25          GPIO_SetBits(GPIOB,GPIO_Pin_8);
26    }
```

第18～26行为BEEP函数。第21行调用库函数GPIO_ReadOutputDataBit读出PB8口的输出状态，如果为高电平(第22行为真)，则第23行调用GPIO_ResetBits函数清零PB8口，即关闭蜂鸣器；如果为低电平(第24行的情况)，则第25行调用GPIO_SetBits函数置位PB8口，即打开蜂鸣器。因此，调用一次BEEP函数，就使得蜂鸣器切换一次工作状态。

程序段 5-14　文件 key.c

```
1     //Filename: key.c
2
3     #include "includes.h"
4
5     void KEYInit(void)
6     {
7       GPIO_InitTypeDef g;
8       RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE | RCC_APB2Periph_AFIO, ENABLE);
9
10      g.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3 | GPIO_Pin_4;
11      g.GPIO_Mode = GPIO_Mode_IPU;
12      g.GPIO_Speed = GPIO_Speed_10MHz;
13      GPIO_Init(GPIOE, &g);
14
15      GPIO_SetBits(GPIOE,GPIO_Pin_2 | GPIO_Pin_3 | GPIO_Pin_4);
16    }
```

第5～16行为KEYInit函数。按键KEY0～KEY2占用PE4、PE3和PE2口，同时需要开启AFIO功能，所以，第8行调用库函数RCC_APB2PeriphClockCmd开启PE口和AFIO口的时钟源。第10～13行初始化PE2～PE4为上拉有效的输入口，且工作频率为10MHz。第15行使PE2～PE4口输出高电平，相当于为PE2～PE4提供强上拉功能。

程序段 5-15　文件 exti.c

```
1   //Filename: exti.c
2
3   #include "includes.h"
4
5   void EXTIKeyInit(void)
6   {
7     EXTI_InitTypeDef   e;
8
9     GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource2); //PE2 为 EXTI2
10    GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource3); //PE3 为 EXTI3
11    GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource4); //PE4 为 EXTI4
12
13    e.EXTI_Line = EXTI_Line2 | EXTI_Line3 | EXTI_Line4;
14    e.EXTI_Mode = EXTI_Mode_Interrupt;
15    e.EXTI_Trigger = EXTI_Trigger_Falling;
16    e.EXTI_LineCmd = ENABLE;
17    EXTI_Init(&e);
18
19    NVIC_EnableIRQ(EXTI2_IRQn);     //开放外部中断 EXTI2,EXTI3,EXTI4
20    NVIC_EnableIRQ(EXTI3_IRQn);
21    NVIC_EnableIRQ(EXTI4_IRQn);
22    NVIC_SetPriority(EXTI2_IRQn,5);
23    NVIC_SetPriority(EXTI3_IRQn,6);
24    NVIC_SetPriority(EXTI4_IRQn,7);
25  }
26
```

第 5～25 行为外部输入中断的初始化函数 EXTIKeyInit。第 9～11 行调用库函数 GPIO_EXTILineConfig 依次将 PE2、PE3 和 PE4 配置为外部中断 EXTI2、EXTI3 和 EXTI4。第 13～17 行配置外部输入中断 EXTI2、EXTI3 和 EXTI4 工作在中断模式，且为下降沿触发方式。第 19～21 行为调用 CMSIS 库函数 NVIC_EnableIRQ 开放外部中断 EXTI2、EXTI3 和 EXTI4，第 22～24 行调用 CMSIS 库函数 NVIC_SetPriority 配置外部中断 EXTI2、EXTI3 和 EXTI4 的优先级号依次为 5、6 和 7。

```
27  void EXTI2_IRQHandler()
28  {
29    LED(0,LED_ON);
30    EXTI_ClearFlag(EXTI_Line2);
31    NVIC_ClearPendingIRQ(EXTI2_IRQn);
32  }
33
34  void EXTI3_IRQHandler()
35  {
36    LED(0,LED_OFF);
37    EXTI_ClearFlag(EXTI_Line3);
38    NVIC_ClearPendingIRQ(EXTI3_IRQn);
39  }
40
41  void EXTI4_IRQHandler()
42  {
43    BEEP();
```

```
44        EXTI_ClearFlag(EXTI_Line4);
45        NVIC_ClearPendingIRQ(EXTI4_IRQn);
46    }
```

对比程序段 5-11，这里的 3 个中断服务函数中，清除中断标志使用了库函数 EXTI_ClearFlag，如第 30、37 和 44 行所示，依次清零 EXTI2、EXTI3 和 EXTI4 的中断请求标志。

在上述库函数方式的工程程序中，新出现了结构体类型 EXIT_InitTypeDef 和新的库函数 GPIO_EXTILineConfig、EXTI_Init 和 EXTI_ClearFlag，它们均被定义在 stm32f10x_exti.c 或 stm32f10x_exti.h 中，因此需要将 stm32f10x_exti.c 添加到工程管理器的“LIB”分组中。关于这些新结构体类型和新库函数的具体描述请参考 STM32F10x 库函数手册，或直接从头文件 stm32f10x_exti.h 中查看。

5.4 本章小结

本章介绍了嵌套向量中断控制器 NVIC 的工作原理和 GPIO 口作为外部中断的程序设计方法，然后以按键控制为例，讨论了下降沿触发中断的方法，并给出了寄存器类型和库函数类型的工程程序。外部中断是 STM32F103ZET6 微控制器响应外部异步事件的唯一方式，中断的处理能力也是反映 STM32F103ZET6 的性能和灵活性的重要指标。建议读者在学习本章内容后，仔细阅读库函数手册和文件 stm32f10x_exti.c 与 stm32f10x_exti.h，充分掌握新出现的库函数的用法，并设计下降沿触发中断的工程程序。第 6 章介绍定时器时还将继续使用 NVIC 中断。

习题

1. 阐述与中断控制相关的操作及其 CMSIS 库函数。
2. 结合本章内容，说明 GPIO 外部输入中断的响应处理方法。
3. 编写寄存器类型工程，实现对按键的中断输入响应，用 LED 灯状态反映按键的按下或弹开。
4. 编写库函数类型工程，实现对按键的中断输入响应，用 LED 灯状态反映按键的按下或弹开。
5. 说明将 PD12 配置为外部中断输入 EXTI12 的方法。
6. 简要描述中断优先级的配置方法。