实验 3 基于定时器的中断实现及比较输出

实验目的：利用 STM32F407 TIM 实现定时触发中断并利用比较输出功能实现 PWM 波输出。

实验内容：

- 设置定时器参数实现定时中断功能，并编写中断服务程序。
- 利用定时器的比较输出模式实现呼吸灯功能。

3.1 定时器中断实现

本节实验介绍如何使用 STM32F4 的通用定时器实现定时并触发中断服务。

3.1.1 STM32F407 定时器简介

STM32F4 的定时器功能十分强大，有高级定时器 TIMER1 和 TIMER8，也有 TIMER2~TIMER5，TIMER9~TIMER14 等通用定时器，还有 TIMER6 和 TIMER7 等基本定时器，总共达 14 个定时器之多。本节中将使用 TIM3 的定时器中断来控制 LED 的翻转。

STM32F4 的通用定时器包含一个 16 位或 32 位自动重载计数器（CNT），该计数器由可编程预分频器（PSC）驱动。STM32F4 的通用定时器可以被用于：测量输入信号的脉冲长度(输入捕获)或者产生输出波形(输出比较和 PWM)等。 使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32F4 的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM32 的通用 TIMx (TIM2~TIM5 和 TIM9~TIM14)定时器功能包括：

1）16 位/32 位(仅 TIM2 和 TIM5)向上、向下、向上/向下自动装载计数器（TIMx_CNT），注意：TIM9~TIM14 只支持向上（递增）计数方式。

2）16 位可编程(可以实时修改)预分频器(TIMx_PSC)，计数器时钟频率的分频系数为 1～65535 之间的任意数值。

3）4 个独立通道（TIMx_CH1~4，TIM9~TIM14 最多 2 个通道），这些通道可以用来作为：

A 输入捕获

B 输出比较

C PWM 生成(边缘或中间对齐模式) ，注意：TIM9~TIM14 不支持中间对齐模式

D 单脉冲模式输出

可使用外部信号（TIMx_ETR）控制定时器和定时器互连（可以用 1 个定时器控制另外一个定时器）的同步电路。如下事件发生时产生中断/DMA（TIM9~TIM14 不支持 DMA）：

更新：计数器向上溢出/向下溢出，计数器初始化(通过软件或者内部/外部触发)；

触发事件(计数器启动、停止、初始化或者由内部/外部触发计数)；

输入捕获；

输出比较；

关于 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。自动重装载寄存器（TIMx_ARR），该寄存器在物理上实际对应着 2 个寄存器，一个是程序员可以直接操作的，另外一个是程序员看不到的，这个看不到的寄存器在《STM32F4xx 中文参考手册》里面被叫做影子寄存器。事实上真正起作用的是影子寄存器。

3.1.2 库函数配置定时器流程

实验中我们将使用定时器 TIM3 产生中断，然后在中断服务程序里翻转 LED。定时器相关的库函数主要集中在固件库文件 stm32f4xx_tim.h 和 stm32f4xx_tim.c 文件中。定时器配置步骤如下：

1. TIM3 时钟使能。

TIM3 是挂载在 APB1 上，所以通过 APB1 总线下的使能使能函数来使能 TIM3：

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE); ///使能 TIM3 时钟

2. 初始化定时器参数,设置自动重装值，分频系数，计数方式等。

voidTIM_TimeBaseInit(TIM_TypeDef*TIMx,

TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);

在库函数中，定时器的初始化参数是通过初始化函数 TIM_TimeBaseInit 实现的：

第一个参数是确定是哪个定时器，这个比较容易理解。第二个参数是定时器初始化参数结构体指针，结构体类型为 TIM_TimeBaseInitTypeDef，这个结构体的定义：

```
typedef struct
{
uint16_t TIM_Prescaler;
uint16_t TIM_CounterMode;
uint16_t TIM_Period;
uint16_t TIM_ClockDivision;
uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
```

这个结构体一共有 5 个成员变量，要说明的是，对于通用定时器只有前面四个参数有用，最后一个参数 TIM_RepetitionCounter 是高级定时器才有用的。

第一个参数 TIM_Prescaler 是用来设置分频系数的。

第二个参数 TIM_CounterMode 是用来设置计数方式，可以设置为向上计数，向下计数方式还有中央对齐计数方式，比较常用的是向上计数模式 TIM_CounterMode_Up 和向下计数模式 TIM_CounterMode_Down。

第三个参数是设置自动重载计数周期值，这在前面也已经讲解过。第四个参数是用来设置时钟分频因子，参数可设置如下所示：

```
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 5000;
TIM_TimeBaseStructure.TIM_Prescaler =7199;
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
```

针对 TIM3 初始化范例代码格式：

```
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
```

3. 设置 TIM3_DIER 允许更新中断。

因为要使用 TIM3 的更新中断，寄存器的相应位便可使能更新中断。在库函数里面定时器中断使能是通过 TIM_ITConfig 函数来实现的：

```
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState);
```

第一个参数是选择定时器号，取值为 TIM1~TIM14。

第二个参数是用来指明使能的定时器中断的类型，定时器中断的类型有很多种，包括更新中断 TIM_IT_Update，触发中断 TIM_IT_Trigger，以及输入捕获中断 TIM_IT_CC1 等等。

第三个参数是失能还是使能。

例如要使能 TIM3 的更新中断，格式为：

```
TIM_ITConfig(TIM3,TIM_IT_Update, ENABLE );
```

4. TIM3 中断优先级设置。

在定时器中断使能之后，因为要产生中断，必不可少的要设置 NVIC 相关寄存器，调用 **NVIC_Init** 函数可以实现中断优先级的设置。

5. 允许 TIM3 工作，也就是使能 TIM3。

在配置完后要开启定时器，通过 TIM3_CR1 的 CEN 位来设置。在固件库里面使能定时器的函数是通过 TIM_Cmd 函数来实现的：

void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState)

这个函数非常简单，比如要使能定时器 3，方法为：

```
TIM_Cmd(TIM3, ENABLE);      //使能 TIMx 外设
```

6. 编写中断服务函数。

最后，还要编写定时器中断服务函数，通过该函数来处理定时器产生的相关中断。在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，使用的是更新（溢出）中断，在处理完中断之后应该向 TIM3_SR 的最低位写 0，来清除该中断标志。

在固件库函数里面，用来读取中断状态寄存器的值判断中断类型的函数是：

ITStatus TIM_GetITStatus(TIM_TypeDef* TIMx, uint16_t)

该函数的作用是，判断定时器 TIMx 的中断类型 TIM_IT 是否发生中断。比如，我们要判断定时器 3 是否发生更新（溢出）中断，方法为：

```
if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET){
}
```

固件库中清除中断标志位的函数是：

void TIM_ClearITPendingBit(TIM_TypeDef* TIMx, uint16_t TIM_IT)

该函数的作用是，清除定时器 TIMx 的中断 TIM_IT 标志位。比如我们在 TIM3 的溢出中断发生后，我们要清除中断标志位，方法是：

```
TIM_ClearITPendingBit(TIM3, TIM_IT_Update );
```

固件库还提供了两个函数用来判断定时器状态以及清除定时器状态标志位的函数 TIM_GetFlagStatus 和 TIM_ClearFlag，他们的作用和前面两个函数的作用类似。只是在 TIM_GetITStatus 函数中会先判断这种中断是否使能，使能了才去判断中断标志位，而 TIM_GetFlagStatus 直接用来判断状态标志位。

通过以上几个步骤，就可以使用通用定时器的更新中断，来控制 LED 的亮灭。

3.1.3 实验步骤

配置工程过程如下：

1. 在工程目录中添加 HARDWARE 子目录，并在 HARDWARE 目录中建立 led 子目录，分别放入 led.c 和 led.h，并在工程 Group->HARDWARE 中添加 led.c，在 led.c 中按实验 2 进行 LED 的相关配置。

2. 在工程中的 HARDWARE 目录中添加 timer 子目录，并在 timer 目录中添加 timer.c 文件（包括头文件 timer.h），这两个文件需要自己编写。同时在工程 Group-> FWLIB 中添加定时器相关的固件库函数文 stm32f4xx_tim.c。

timer.c 文件代码如下：

```
//通用定时器 3 中断初始化
//arr：自动重装值。 psc：时钟预分频数
//定时器溢出时间计算方法:Tout=((arr+1)*(psc+1))/Ft us.
//Ft=定时器工作频率,单位:Mhz
//这里使用的是定时器 3
void TIM3_Init(u16 arr,    u16 psc)
{
    TIM_TimeBaseInitTypeDef    TIM_TimeBaseInitStructure;
    NVIC_InitTypeDef    NVIC_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE);
    //①使能 TIM3 时钟
    TIM_TimeBaseInitStructure.TIM_Period = arr;
    //自动重装载值
    TIM_TimeBaseInitStructure.TIM_Prescaler=psc;
    //定时器分频
    TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up;
    //向上计数模式
    TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;

    TIM_TimeBaseInit(TIM3,&TIM_TimeBaseInitStructure);
    // ②初始化定时器
    TIM3 TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE);
    //③允许定时器 3 更新中断
    NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn;
    //定时器 3 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0x01;
    //抢占优先级 1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority=0x03;
    //子优先级 3
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    // ④初始化 NVIC
    TIM_Cmd(TIM3,ENABLE);
```

```
    //⑤使能定时器 3
}
```

3. 编写定时器 3 中断服务函数，可以放入 stm32f4xx_it.c 中，其中 TIM_GetITStatus，LEDx 等需要添加相关头文件。

```
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)!=RESET;) //TIM3 溢出中断
    {
        LED1=!LED1;
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update); //清除中断标志位
}
```

中断服务函数比较简单，在每次中断后，判断 TIM3 的中断类型，如果中断类型正确，则执行 LED1 的翻转。

TIM3_Init(u16 arr, u16 psc) 函数执行流程就是上面介绍的 5 个步骤，使得 TIM3 开始工作，并开启中断。该函数的 2 个参数用来设置 TIM3 的溢出时间。

因为系统初始化 SystemInit( ) 函数里面已经初始化 APB1 的时钟为 4 分频，所以 APB1 的时钟为 42M，而从 STM32F4 的内部时钟树图得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 以及 TIM12~14 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 以及 TIM12~14 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 84M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

Tout= ((arr+1)*(psc+1))/Tclk；

其中：

Tclk：TIM3 的输入时钟频率（单位为 Mhz）。

Tout：TIM3 溢出时间（单位为 us）。

timer.h 头文件内容比较简单，自行编写即可。

最后，主函数代码如下：

```
int main(void)
{
    delay_init(168); //初始化延时函数
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
```

```
    ？？？//TIM3 初始化，参数计算后自行填写
    while(1)
    {
        ？？？//LED0 循环亮灭代码
    }
}
```

3.1.4 实验练习

在实验框架中补齐代码，分别实现 LED0 和 LED1 的不同周期闪烁。

3.2 PWM 波形输出（呼吸灯）的实现

本节实验介绍如何使用 STM32F4 的通用定时器实现比较输出即 PWM 波形输出的功能。

3.2.1 PWM 简介

脉冲宽度调制(PWM)，是英文“Pulse Width Modulation”的缩写，简称脉宽调制，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术。就是实现对脉冲宽度的控制，定时器实现 PWM 输出原理如图 3.1 所示：

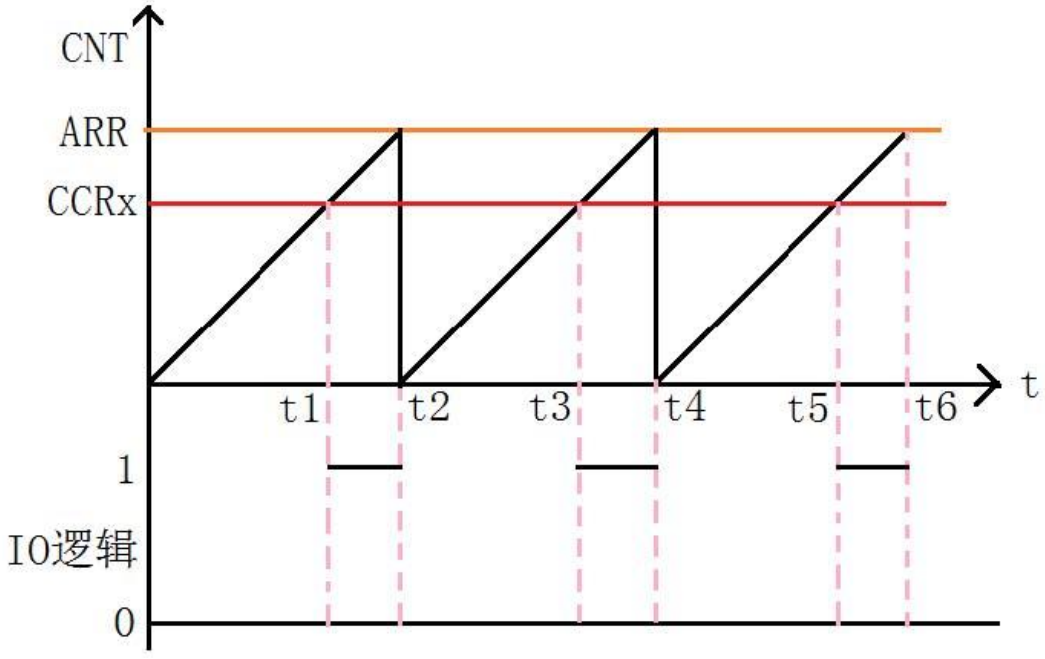


图 3.1 定时器实现 PWM 输出原理图

假定定时器工作在向上计数 PWM 模式，且当 CNT<CCRx 时，输出 0，当 CNT>=CCRx 时输出 1。那么就可以得到如上的 PWM 示意图：当 CNT 值小于 CCRx 的时候，IO 输出低电平(0)，当 CNT 值大于等于 CCRx 的时候，IO 输出高电平(1)，当 CNT 达到 ARR 值的时候，重新归零，然后重新向上计数，依次循环。改变 CCRx 的值，就可以改变 PWM 输出的占空比，改变 ARR 的值，就可以改变 PWM 输出的频率，这就是 PWM 输出的原理。

STM32F4 的定时器除了 TIM6 和 7。其他的定时器都可以用来产生 PWM 输出。其中高级定时器 TIM1 和 TIM8 可以同时产生多达 7 路的 PWM 输出。而通用定时器也能同时产生多达 4 路的 PWM 输出，实验中使用 TIM2 的 CH3 产生一路 PWM 输出（呼吸灯连接 GPIO 中的 B10，而 B10 可以复用为 TIM2 的 CH3）。

要使 STM32F4 的通用定时器 TIMx 产生 PWM 输出，除了基本的寄存器外，还会用到 3 个寄存器，来控制 PWM。这三个寄存器分别是：捕获/比较模式寄存器（TIMx_CCMR1/2）、捕获/比较使能寄存器（TIMx_CCER）、捕获/比较寄存器（TIMx_CCR1~4）。

3.2.2 库函数配置定时器流程

PWM 实际跟上一节一样使用的是定时器的功能，所以相关的函数设置同样在库函数文件 stm32f4xx_tim.h 和 stm32f4xx_tim.c 文件中。

（1）开启 TIM2 和 GPIO 时钟。

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

（2）配置 GPIO B10 为复用 TIM2 输出。

```
GPIO_PinAFConfig(GPIOB,GPIO_PinSource10,GPIO_AF_TIM2);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;            //复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz; //速度 100MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;          //推挽复用输出
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;            //上拉
GPIO_Init(GPIOB,&GPIO_InitStructure);
```

（3）初始化 TIM2，设置 TIM2 的 ARR 和 PSC 等参数。

```
TIM_TimeBaseStructure.TIM_Prescaler=psc;
TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up;
TIM_TimeBaseStructure.TIM_Period=arr;
TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);
```

（4）设置 TIM2_CH3 的 PWM 模式，使能 TIM2 的 CH3 输出。

```
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High ;
TIM_OC3Init(TIM2, &TIM_OCInitStructure);
```

使能 TIM2。

```
TIM_Cmd(TIM2, ENABLE);
```

修改 TIM2_CCR3 来控制占空比。

最后，在经过以上设置之后，PWM 其实已经开始输出了，只是其占空比和频率都是固定的，而通过修改 TIM2_CCR3 则可以控制 CH3 的输出占空比，进而控制 LED 的亮度。

在库函数中，修改 TIM2_CCR3 占空比的函数是：

```
void TIM_SetCompare3(TIM_TypeDef* TIMx, uint16_t Compare2)；
```

3.2.3 实验步骤

配置工程过程如下：

（1）工程配置及文件添加与实验 3.1 类似，不再赘述。添加 pwm.c 和 pwm.h 文件

（2）pwm.c 中添加定时器初始化函数。

```
void TIM2_PWM_Init(u32 arr, u32 psc)
{
```

```
GPIO_InitTypeDef GPIO_InitStructure;
TIM_TimeBaseInitTypeDef   TIM_TimeBaseStructure;
TIM_OCInitTypeDef   TIM_OCInitStructure;

RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
GPIO_PinAFConfig(GPIOB,GPIO_PinSource10,GPIO_AF_TIM2);


......//配置 GPIO B10，需配置为 GPIO_Mode_AF 输出复用

......//配置 TIM_TimeBaseStructure，同 3.1

TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);


 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
 TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
 TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High ;
 TIM_OC3Init(TIM2, &TIM_OCInitStructure);
 TIM_Cmd(TIM2, ENABLE);

}
```

（3）添加主程序中代码

首先包含相关头文件，定义控制变量；

```
 delay_init(168);
 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
TIM2_PWM_Init(3000,0);

 while(1)
{
     delay_ms(1);
      ？？？//修改比较值，修改占空比
```

```
        TIM_SetCompare3(TIM2,  pwm);          //pwm 即为 CCRx
    }
```

3.2.4 实验练习

在实验 3.2 的框架中补齐代码，通过修改占空比输出 PWM 波形并实现 LED 呼吸灯。

实验 4 基于 EXTI 的外部中断

实验目的：利用 STM32F407 EXTI 实现外部触发中断并调用中断服务函数。

实验内容：

- 设置 EXTI 实现外部中断功能，并编写中断服务程序。

4.1 STM32F407 的外部中断

ARM Cortex-M3/M4 内核的中断系统很庞大，但是 STM32 并没有使用内核的所有资源，目前 STM32(非互联型)可支持 60 个中断通道，互联型支持 68 个。STM32 支持 20 个外部中断源，包括：16 个 GPIO 输入中断；PVD 输出；RTC 闹钟事件；USB 唤醒事件；以太网唤醒事件(只适用于互联型产品)。本节讨论的是 16 个 GPIO 输入中断。

GPIO 输入中断虽然有 16 个输入通道，但是只占用了 7 个中断向量。EXTI0~EXTI4 各占用一个中断向量，EXTI5~9 共用一个，EXTI10~15 共用一个。所以在编程的时候 EXTI5~9 将共用一个中断函数，EXTI10~15 共用一个中断函数。这 16 个外部中断和 GPIO 的映射如图 4.1 所示。

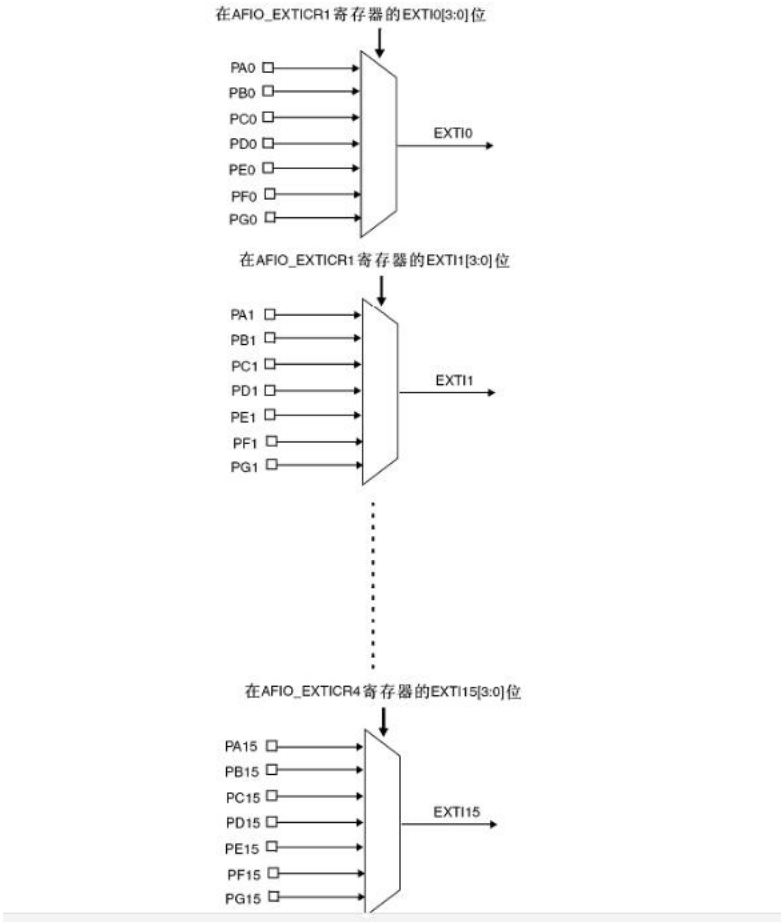
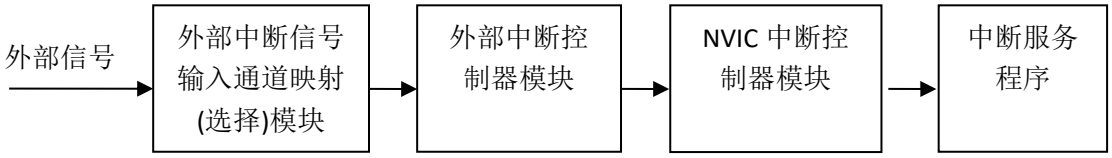


图 4.1 GPIO 和中断线的映射关系图

可以看出：EXTI0 通道只能选择 PA0，PB0，PC0，PD0，PE0，PF0，PG0 这 7 个引脚中的一个，而 EXTI1 只能选择名称为 Px1 的 7 个引脚中的一个(x 可取 A~G 中的一个)，依此类推 EXTI2 只能选择 Px2 引脚……。

4.2 库函数配置 EXTI 流程

外部中断 EXTI 和 NVIC 相关寄存器及各字段配置可以参考用户手册，这里不再赘述。可以总结为 STM32 外部中断要经过 3 个部分模块设置处理，然后才进入到中断服务程序的处理，其框图如下：



通过上面的介绍和分析，可以对 STM32 的外部中断有了清楚的了解，在编

程的时候按照其信号路径，根据需要设置相关的寄存器即可。使用库函数配置外部中断的步骤如下：

（1）使能 IO 口时钟，初始化 IO 口为输入

要使用 IO 口作为中断输入，所以要使能相应的 IO 口时钟，以及初始化相应的 IO 口为输入模式，具体的使用方法与第二章 GPIO 是一致的，这里就不做过多讲解。

（2）开启 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

配置 GPIO 与中断线的映射关系，首先需要打开 SYSCFG 时钟。

RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

//使能 SYSCFG 时钟 ，一定要注意，只要我们使用到外部中断，就必须打开 SYSCFG 时钟，而 STM32F4 与 STM32F1 不同，STM32F1 使用的是：

RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE); //使能复用时钟

接下来配置 GPIO 与中断线的映射关系。在库函数中，配置 GPIO 与中断线的映射关系的函数 SYSCFG_EXTILineConfig( )来实现的：

void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex);

该函数将 GPIO 端口与中断线映射起来，使用示例是：

SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);

//GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

//这是以前的 F103 所用的配置函数

将中断线 0 与 GPIOA 映射起来，那么很显然是 GPIOA.0 与 EXTI0 中断线连接了。设置好中断 线映射之后，接下来就要设置该中断线上中断的初始化参数了。

（3）初始化线上中断，设置触发条件等。

中断线上中断的初始化是通过函数 EXTI_Init()实现的。EXTI_Init()函数的定义是：

void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct); 下面我们用一个使用范例来说明这个函数的使用：

EXTI_InitTypeDef EXTI_InitStructure;

```
EXTI_InitStructure.EXTI_Line=EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;  //中断事件
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);      //初始化外设 EXTI 寄存器
```

上面的例子设置中断线 4 上的中断为下降沿触发。STM32 的外设的初始化都是通过结构体来设置初始值的，这里就不再讲解结构体初始化的过程了，结构体 EXTI_InitTypeDef 的成员变量：

```
typedef struct
{
    uint32_t EXTI_Line;
    EXTIMode_TypeDef EXTI_Mode;
    EXTITrigger_TypeDef EXTI_Trigger;
    FunctionalState EXTI_LineCmd;
}EXTI_InitTypeDef;
```

第一个参数是中断线的标号，对于外部中断，取值范围为 EXTI_Line0 ~EXTI_Line15。这个在上面已经讲过中断线的概念。也就是说，这个函数配置的是某个中断线上的中断参数。第二个参数是中断模式，可选值为中断 EXTI_Mode_Interrupt 和事件 EXTI_Mode_Event。第三个参数是触发方式，可以是下降沿触发 EXTI_Trigger_Falling，上升沿触发 EXTI_Trigger_Rising，或者任意电平（上升沿和下降沿）触发 EXTI_Trigger_Rising_Falling，最后一个参数就是使能中断线了。

（4）配置中断分组（NVIC），并使能中断。

设置好中断线和 GPIO 映射关系，然后又设置好了中断的触发模式等初始化参数。既然是外部中断，涉及到中断还要设置 NVIC 中断优先级，设置中断线 2 的中断优先级。

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;
//使能按键外部中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;
//抢占优先级 2
```

NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;

//响应优先级 2

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

//使能外部中断通道

NVIC_Init(&NVIC_InitStructure);

//中断优先级分组初始化

（5）编写中断服务函数。

配置完中断优先级之后，接着要做的就是编写中断服务函数。中断服务函数的名字是事先有定义的。这里需要说明一下，STM32F4 的 IO 口外部中断函数只有 7 个，分别为：

EXPORT EXTI0_IRQHandler

EXPORT EXTI1_IRQHandler

EXPORT EXTI2_IRQHandler

EXPORT EXTI3_IRQHandler

EXPORT EXTI4_IRQHandler

EXPORT EXTI9_5_IRQHandler

EXPORT EXTI15_10_IRQHandler

这些函数在系统启动文件中定义，名称不能有任何变化，使用时直接使用即可。

中断线 0-4 每个中断线对应一个中断函数，中断线 5-9 共用中断函数 EXTI9_5_IRQHandler，中断线 10-15 共用中断函数 EXTI15_10_IRQHandler。在编写中断服务函数的时候会经常使用到两个函数，第一个函数是判断某个中断线上的中断是否发生（标志位是否置位）： ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)； 这个函数一般使用在中断服务函数的开头判断中断是否发生。另一个函数是清除某个中断线上的中断标志位：

void EXTI_ClearITPendingBit(uint32_t EXTI_Line)；

这个函数一般应用在中断服务函数结束之前，清除中断标志位。 常用的中断服务函数格式为：

void EXTI3_IRQHandler(void)

{

```
    if(EXTI_GetITStatus(EXTI_Line3)!=RESET)
    //判断某个线上的中断是否发生
    {
        ..... //中断操作
        EXTI_ClearITPendingBit(EXTI_Line3);
        //清除 LINE 上的中断标志位
    }
}
```

固件库还提供了两个函数用来判断外部中断状态以及清除外部状态标志位的函数 EXTI_GetFlagStatus 和 EXTI_ClearFlag，他们的作用和前面两个函数的作用类似。只是在 EXTI_GetITStatus 函数中会先判断这种中断是否使能，使能了才去判断中断标志位，而 EXTI_GetFlagStatus 直接用来判断状态标志位。使用 IO 口外部中断的一般步骤：

1）使能 IO 口时钟，初始化 IO 口为输入。

2）使能 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

3）初始化线上中断，设置触发条件等。

4）配置中断分组（NVIC），并使能中断。

5）编写中断服务函数。

通过以上几个步骤的设置，就可以正常使用外部中断了。

4.3 实验步骤

配置工程过程如下：

（1）在工程目录中添加 HARDWARE 子目录，并在 HARDWARE 目录中建立 exti 子目录，分别放入文件 exti.c 和 exti.h，并在工程分组 Group 中的 HARDWARE 下添加 exti.c，同理加入 led 和 key，并进行相应的工程配置。

（2）在 exti.c 中按照 4.2 流程进行 EXTI 的相关配置，同时还需要引入了外部中断相关的固件库函数文 stm32f4xx_exti.c 和 tm32f4xx_syscfg.c。

exti.c 文件代码如下：

```
void EXTIX_Init(void)
{
```

```
    NVIC_InitTypeDef     NVIC_InitStructure;
    EXTI_InitTypeDef     EXTI_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource0);

    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; //上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;

    EXTI_Init( &EXTI_InitStructure );

    //中断 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel=EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02；
    NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

（3）编写 LED 与 KEY 相关代码。

（4）编写中断服务函数。

主函数代码如下：

```
//包含相关头文件
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    delay_init(168);
    LED_Init();
    KEY_Init();
    EXTIX_Init();
    LED0=0;
    while(1)
```

```
        {

        }
    }
```

4.4 实验练习

在实验 4.3 的框架中补齐代码，以 SW0（E0）做为外部中断输入引脚控制 LED 的亮灭。