第 5 章

CHAPTER 5

外部中断的原理与应用

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,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 中断的概述

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,再返回刚才的暂停处,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：处理中断的程序叫作中断服务程序。中断服务程序的入口（起始）地址叫作中断向量。

（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，这个过程叫作中断嵌套。

（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 系列单片机有 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，见表 5.1。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 系列单片机一共有 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，用多少位来设置抢占优先级的等级，多少位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

续表

分　　组	IP[7:4]分配情况
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。因为改变优先级分组,会造成程序中设置好的中断的优先级都跟着发生变化,进而引起程序执行错误。

5.2.3　NVIC 中断管理相关 HAL 函数

在 stm32f1xx_hal_cortex.h 中有 NVIC 相关的 HAL 库函数声明,初始化时主要用 4 个函数。

5.2.3.1　函数 HAL_NVIC_SetPriorityGrouping()

表 5.2 描述了函数 HAL_NVIC_SetPriorityGrouping()。

表 5.2　HAL_NVIC_SetPriorityGrouping()的函数描述表

函数名	HAL_NVIC_SetPriorityGrouping
函数原型	void HAL_NVIC_SetPriorityGrouping (uint32_t PriorityGroup)
功能描述	设置优先级分组(抢占优先级和子优先级)
输入参数	PriorityGroup：取值 NVIC_PRIORITYGROUP_x (x = 0 ～ 4)，NVIC_PRIORITYGROUP_0：0 位用于设置抢占优先级，4 位用于设置子优先级。其余分组以此类推
输出参数	无
返回值	无
先决条件	无
被调用函数	无

5.2.3.2　函数 HAL_NVIC_SetPriority()

表 5.3 描述了函数 HAL_NVIC_SetPriority()。

表 5.3　HAL_NVIC_SetPriority()的函数描述表

函数名	HAL_NVIC_SetPriority
函数原型	void HAL_NVIC_SetPriority (IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority)
功能描述	设置中断优先级
输入参数 1	IRQn:中断请求名,相关定义见 stm32f103xb.h 文件
输入参数 2	PreemptPriority：中断请求 IRQn 的抢占优先级,取值范围 0～15,值越小,优先级越高
输入参数 3	SubPriority：中断请求 IRQn 的子优先级,取值范围 0～15,值越小,优先级越高
输出参数	无
返回值	无
先决条件	无
被调用函数	无

5.2.3.3 函数 HAL_NVIC_EnableIRQ()/HAL_NVIC_DisableIRQ()

表 5.4 描述了函数 HAL_NVIC_EnableIRQ()/HAL_NVIC_DisableIRQ ()。

表 5.4 HAL_NVIC_EnableIRQ()/HAL_NVIC_DisableIRQ()的函数描述表

函数名	HAL_NVIC_EnableIRQ/HAL_NVIC_DisableIRQ
函数原型	void HAL_NVIC_EnableIRQ (IRQn_Type IRQn)/ void HAL_NVIC_DisableIRQ (IRQn_Type IRQn)
功能描述	使能/失能指定中断，在此函数之前须调用 NVIC_PriorityGroupConfig()，以确保中断优先级正确设置
输入参数	IRQn：中断请求名，相关定义见 stm32f103xb.h 文件
输出参数	无
返回值	无
先决条件	无
被调用函数	无

5.2.3.4 函数 HAL_NVIC_SystemReset()

表 5.5 描述了函数 HAL_NVIC_SystemReset()。

表 5.5 HAL_NVIC_SystemReset()的函数描述表

函数名	HAL_NVIC_SystemReset
函数原型	void HAL_NVIC_SystemReset (void)
功能描述	用于软件复位 MCU
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

5.3 EXTI 外部中断

5.3.1 中断请求信号的输入引脚

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引脚。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 端口的输入中断，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 引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。

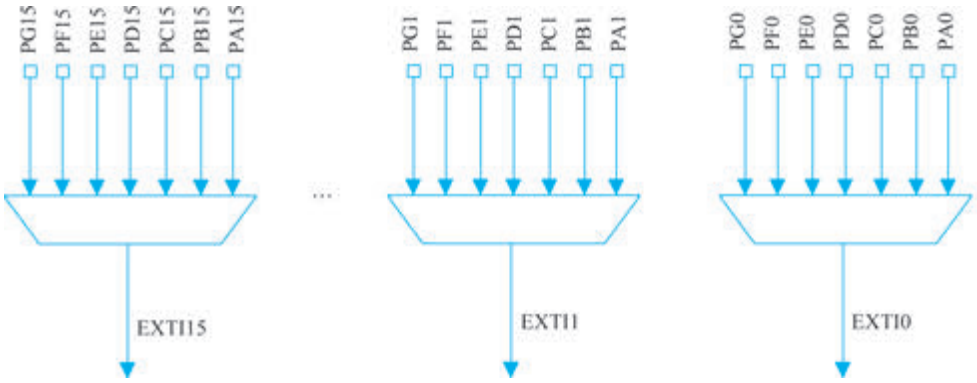

图 5.1　GPIO 引脚和外部中断线的映射关系

5.3.2　EXTI 线对应的中断函数

I/O 端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 个中断函数，见表 5.6。

表 5.6　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_stm32f103xb.s 这样的启动文件中找到。

5.4　EXTI 的常用 HAL 库函数

当发生 EXTI 中断后，系统自动进入 5.3.2 小节介绍的中断函数运行。以下两个 HAL 库函数在用户自行调用时才会运行，它们的函数声明在 stm32f1xx_hal_gpio.h 文件中。

5.4.1　函数 HAL_GPIO_EXTI_IRQHandler()

表 5.7 描述了函数 HAL_GPIO_EXTI_IRQHandler()。

表 5.7　HAL_GPIO_EXTI_IRQHandler()的函数描述表

函数名	HAL_GPIO_EXTI_IRQHandler
函数原型	void HAL_GPIO_EXTI_IRQHandler (uint16_t GPIO_Pin)
功能描述	当 GPIO_Pin 引脚上有外部中断请求时，调用 HAL_GPIO_EXTI_Callback()
输入参数	GPIO_Pin：用作外部中断请求输入线的 GPIO 引脚，取值 GPIO_PIN_x(x=0～15)
输出参数	无
返回值	无
先决条件	无
被调用函数	HAL_GPIO_EXTI_Callback()

5.4.2 函数 HAL_GPIO_EXTI_Callback()

函数 HAL_GPIO_EXTI_Callback()的描述见表 5.8。

表 5.8 HAL_GPIO_EXTI_Callback()函数描述表

函数名	HAL_GPIO_EXTI_Callback
函数原型	__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
功能描述	当 GPIO_Pin 引脚上有外部中断请求时，供用户自行编程完成功能，不能自动运行，需用户调用才会运行
输入参数	GPIO_Pin：用作外部中断请求输入线的 GPIO 引脚，取值 GPIO_PIN_x(x=0～15)
输出参数	无
返回值	无
先决条件	无
被调用函数	无

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) 设置 NVIC 中断优先级分组、中断优先级、使能对应的 EXTI 中断。
(4) 编写 EXTI 中断服务函数。

5.5.2 EXTI 应用实例

例 5.1：使用 EXTI 实现图 5.2 所示按键控制 LED 发光。

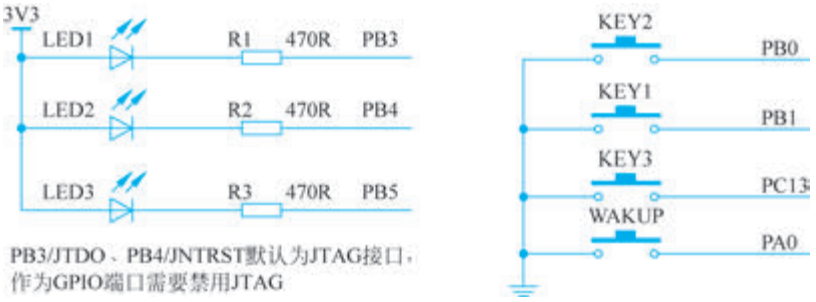

图 5.2 例 5.1 图

解：几个主要函数代码如下：

```
#include "main.h"
#include "gpio.h"
#include "exti.h"
void SystemClock_Config(void);
int main(void)
{
  HAL_Init();
```

```
  SystemClock_Config();
  MX_GPIO_Init();
  exti_init();

  while (1)
  {
      /* 流水灯重复 5 次,具体代码实现见 91 页 */
      GPIO_RUNNING(5);
  }
}

void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* 使能 GPIO 时钟 */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOD_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
    /* 使能 AFIO 时钟 */
    __HAL_RCC_AFIO_CLK_ENABLE();

    /* 引脚重映射关闭 JTAG,开启 SW-DP,以启用 PB3、PB4 的 I/O 功能 */
  __HAL_AFIO_REMAP_SWJ_NOJTAG();

  /* 设置引脚 PC13 输入输出模式 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*设置引脚 PB0、PB1 输入输出模式 */
  GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /*设置引脚 PB3、PB4、PB5 输入输出模式*/
  GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* 设置 PB3、PB4、PB5 的初值 */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5,
                    GPIO_PIN_RESET);
}
void exti_init(void)
{
    /*(1) 使能 EXTI 线所在的 GPIO 时钟和 AFIO 复用时钟
        在 main()函数调用 HAL_Init()和 MX_GPIO_Init()函数时完成,
        具体见例程"EX04 EXTI_MX"(可在清华大学出版社官网下载)
```

```
      *(2) 初始化 EXTI 线所在的 GPIO 的输入/输出模式,在 main()函数调用 MX_GPIO_Init()
         函数时完成
      *(3) 设置 NVIC 中断优先级分组、中断优先级、使能对应的 EXTI 中断
         NVIC 中断优先级分组在 main()函数调用 HAL_Init()函数时完成,分组为 4
         这里只是完成设置 NVIC 中断优先级并使能对应的 EXTI 中断
    /* 因中断优先级分组为 4,所以只有抢占优先级范围为 0～15,无子优先级
     */
    HAL_NVIC_SetPriority(EXTI0_IRQn,2,0);
    HAL_NVIC_EnableIRQ(EXTI0_IRQn);

    HAL_NVIC_SetPriority(EXTI1_IRQn,3,0);
    HAL_NVIC_EnableIRQ(EXTI1_IRQn);

    HAL_NVIC_SetPriority(EXTI15_10_IRQn,4,0);
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    /* 只写了 PB0 或 PB1 按键按下产生中断请求的情况,PC13 按下时,由
      EXTI15_10_IRQHandler ()直接处理,没有调用 HAL_GPIO_EXTI_Callback ()
     */
    if(GPIO_Pin == GPIO_PIN_0)
    {
      /* 3 个灯亮灭 5 次,具体代码实现见 91 页 */
      GPIO_BLINK(5);
    }
  else if(GPIO_Pin == GPIO_PIN_1)
    {
        /* 中间向两边亮 5 次,具体代码实现见 91 页 */
        GPIO_CENTER_FLOWING(5);
    }
}

void EXTI0_IRQHandler(void)
{
    /* 中断服务函数中调用 HAL_GPIO_EXTI_IRQHandler()举例
      在库函数 HAL_GPIO_EXTI_IRQHandler()中会调用 HAL_GPIO_EXTI_Callback()
     */
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}

void EXTI1_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_1);
}

void EXTI15_10_IRQHandler(void)
{
    /* 中断服务函数中也可直接写用户代码,
     * 不调用 HAL_GPIO_EXTI_IRQHandler() */
if (__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_13) != 0x00u)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
```

```
    /* 3 个灯依次闪烁 2 次,重复 5 次 ,具体代码实现见后页 */
    GPIO_RUNNINGBLINK(5);
  }
}
/* HAL_GPIO_EXTI_IRQHandler()是 HAL 库提供的库函数 */
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  /* 若检测到 EXTI 线上有中断请求 */
  if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != 0x00u)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
/* 3 个灯亮灭 num 次 */
void GPIO_BLINK(unsigned char num)
{
    unsigned char i;
    for(i = 0;i < num;i++)
    {
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        ,GPIO_PIN_SET);
      HAL_Delay(300);
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        ,GPIO_PIN_RESET);
      HAL_Delay(300);
    }
}
/* 流水灯重复 num 次 */
void GPIO_RUNNING(unsigned char num)
{
    unsigned char i;
    for(i = 0;i < num;i++)
    {
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        ,GPIO_PIN_SET);
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3,GPIO_PIN_RESET);
      HAL_Delay(300); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        ,GPIO_PIN_SET);
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4,GPIO_PIN_RESET);
      HAL_Delay(300); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                        ,GPIO_PIN_SET);
      HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET);
      HAL_Delay(300);
    }
}
/* 中间向两边亮 num 次 */
void GPIO_CENTER_FLOWING(unsigned char num)
{
  unsigned char i;
  for(i = 0;i < num;i++)
  {
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                      ,GPIO_PIN_SET);
```

```
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4,GPIO_PIN_RESET);
    HAL_Delay(300); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_5,GPIO_PIN_RESET);
    HAL_Delay(300);
  }
}
/* 3 个灯依次闪烁 2 次,循环 num 次 */
void GPIO_RUNNINGBLINK(unsigned char num)
{
  unsigned char i;
  for(i = 0;i < num;i++)
  {
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(300);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3,GPIO_PIN_RESET);
    HAL_Delay(200);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(200);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3,GPIO_PIN_RESET);
    HAL_Delay(200); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(300);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4,GPIO_PIN_RESET);
    HAL_Delay(200); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(200);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_4,GPIO_PIN_RESET);
    HAL_Delay(200); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(300);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET);
    HAL_Delay(200); HAL_GPIO_WritePin(GPIOB,GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5
                              ,GPIO_PIN_SET);
    HAL_Delay(200);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_5,GPIO_PIN_RESET);
    HAL_Delay(200);
  }
}
```