直流电机

单片机通过两个引脚输出控制信号至驱动模块，从而实现电机的正反停转控制和 PWM 调速控制。直流电机接口电路如图 1 所示，在开发板上占用 PA6/PA7 引脚。设定当 PA6=1、PA7=0 时为电机正转，当 PA6=0、PA7=1 时为电机反转，当 PA6=PA7=0 或 PA6=PA7=1 时为电机停转。另外，可以通过施加一个 PWM 脉冲波控制高电平占比来进行电机调速控制，其中 PWM 信号的占空比即为对应的电机速度。

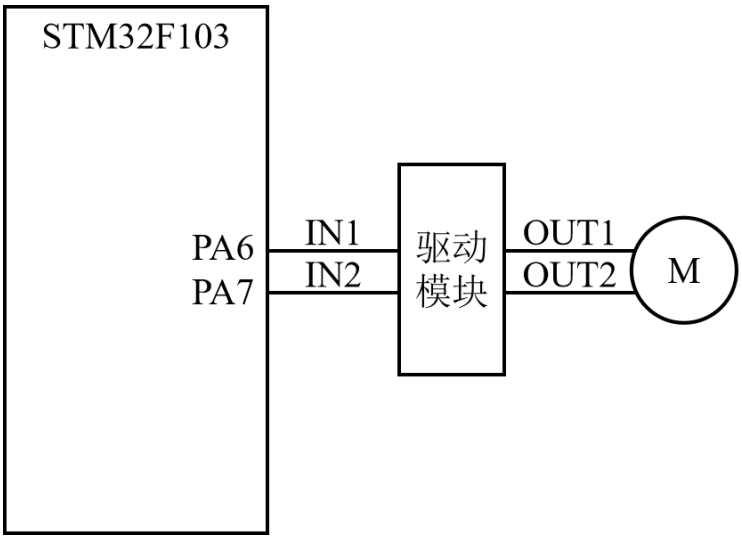


图 1 直流电机接口电路图

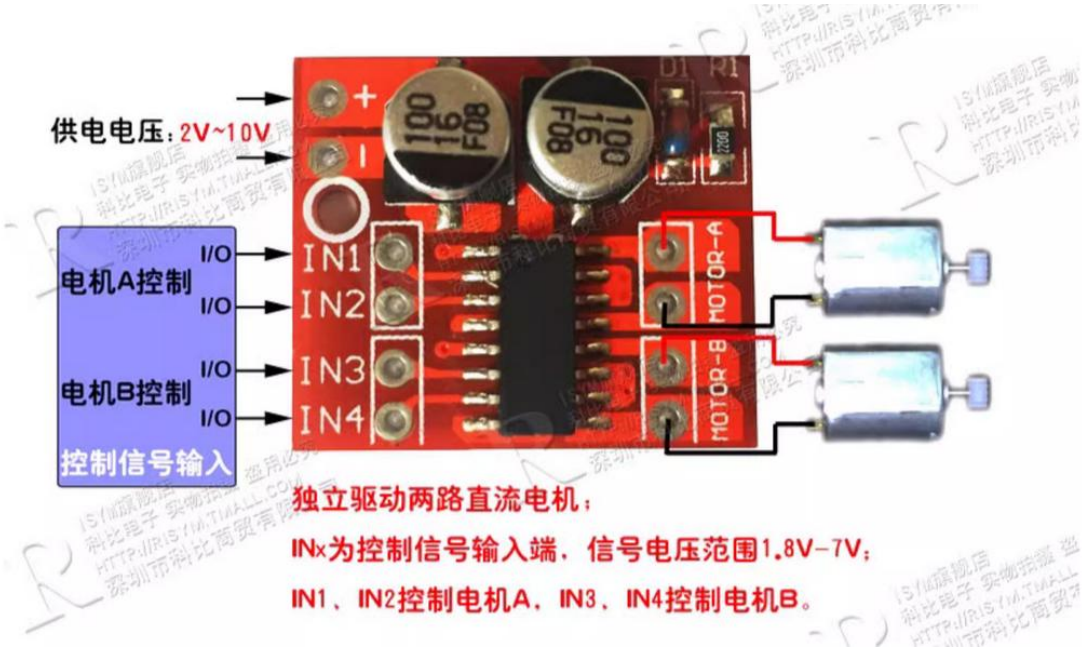


图 2 驱动模块

定义 MOTORA/MOTORB 为电机端口 PA6/PA7，根据电机转向控制的工作原理分别建立电机的正转、反转、停止函数，并声明程序中使用的所有函数。

```
/*********************************************************************************************/
/* 引脚 定义 */

#define MOTOR_GPIO_PORT                 GPIOA

#define MOTORA_GPIO_PIN                 SYS_GPIO_PIN6
#define MOTORB_GPIO_PIN                 SYS_GPIO_PIN7

#define MOTOR_GPIO_CLK_ENABLE()         do{ RCC->APB2ENR |= 1 << 2; }while(0)   /* PA口时钟使能 */

/*********************************************************************************************/


/* MOTOR端口定义 */
#define MOTORA(x)         sys_gpio_pin_set(MOTOR_GPIO_PORT, MOTORA_GPIO_PIN, x)      /* MOTORA */
#define MOTORB(x)         sys_gpio_pin_set(MOTOR_GPIO_PORT, MOTORB_GPIO_PIN, x)      /* MOTORB */


void motor_init(void);    /* 初始化 */
void fwd(void);           /* 电机正转 */
void back(void);          /* 电机反转 */
void stop(void);          /* 电机停止 */
```

motor.h

```
#ifndef __MOTOR_H
#define __MOTOR_H

#include "./SYSTEM/sys/sys.h"

/*********************************************************************************************/
/* 引脚 定义 */

#define MOTOR_GPIO_PORT                 GPIOA

#define MOTORA_GPIO_PIN                 SYS_GPIO_PIN6
#define MOTORB_GPIO_PIN                 SYS_GPIO_PIN7

#define MOTOR_GPIO_CLK_ENABLE()         do{ RCC->APB2ENR |= 1 << 2; }while(0)   /* PA 口时钟使能 */

/*********************************************************************************************/

/* MOTOR 端口定义 */
#define MOTORA(x)         sys_gpio_pin_set(MOTOR_GPIO_PORT, MOTORA_GPIO_PIN, x)      /* MOTORA */
#define MOTORB(x)         sys_gpio_pin_set(MOTOR_GPIO_PORT, MOTORB_GPIO_PIN, x)      /* MOTORB */

void motor_init(void);    /* 初始化 */
void fwd(void);           /* 电机正转 */
void back(void);          /* 电机反转 */
```

```
void stop(void);                /*  电机停止  */

#endif
```

motor.h

motor.c

```
#include "./BSP/MOTOR/motor.h"

/*  电机初始化  */
void motor_init(void)
{
    MOTOR_GPIO_CLK_ENABLE(); /* MOTOR 时钟使能  */

    sys_gpio_set(MOTOR_GPIO_PORT, MOTORA_GPIO_PIN,
                 SYS_GPIO_MODE_OUT,          SYS_GPIO_OTYPE_PP,
SYS_GPIO_SPEED_MID, SYS_GPIO_PUPD_PU);     /* MOTORA 引脚模式设置
*/

    sys_gpio_set(MOTOR_GPIO_PORT, MOTORB_GPIO_PIN,
                 SYS_GPIO_MODE_OUT,          SYS_GPIO_OTYPE_PP,
SYS_GPIO_SPEED_MID, SYS_GPIO_PUPD_PU);     /* MOTORB 引脚模式设置
*/

    MOTORA(1);
    MOTORB(1);      /*  电机停止  */
}

/*  电机正转  */
void fwd(void)
{
      MOTORA(0);
    MOTORB(1);
}

/*  电机反转  */
void back(void)
{
      MOTORA(1);
    MOTORB(0);
}
```

```
/* 电机停止 */
void stop(void)
{
    MOTORA(1);
    MOTORB(1);
}
```

motor.c

直流电机 PWM 调速

直流电机调速控制通过施加一个 PWM 脉冲波控制高电平占比来进行，具体可以设定一个工作周期（短时间），并在单个工作周期内按照所要求的占空比正/反转和停止相应的时长即可。

例如要求电机以 50%占空比速度正转，可以设定一个工作周期为 20ms，在单个工作周期内使电机正转 10ms，停止 10ms，如此电机正转时长占总时长的 50%，即可实现电机以 50%占空比的速度工作。相应程序如下：

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/MOTOR/motor.h"      //使用电机模块需要包含电机头文件

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    motor_init();                   //使用电机模块需要电机初始化
    while(1)
    {
        fwd();                      //电机正转
        delay_ms(10);               //延时 10ms
        stop();                     //电机停止
        delay_ms(10);               //延时 10ms
    }
}
```

蜂鸣器

蜂鸣器由 NPN 型三极管和蜂鸣器组成，单片机输出一个信号使得三极管处于导通或断开的状态，继而控制蜂鸣器发声或关闭。蜂鸣器接口电路如图 1 所示，在开发板上占用 **PB8** 引脚。当 PB8=0 时蜂鸣器关闭，PB8=1 时蜂鸣器发声。

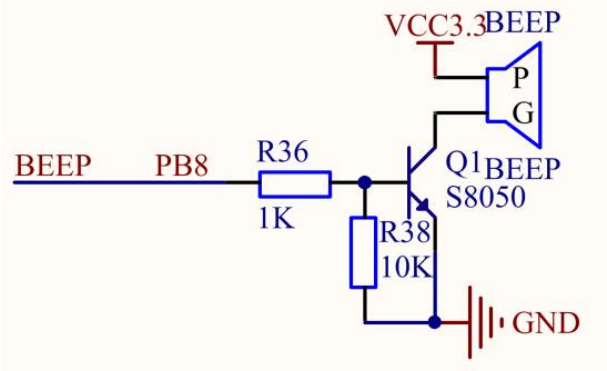


图 1 蜂鸣器接口电路图

蜂鸣器使用：

```
/* 蜂鸣器控制 */
#define BEEP(x)         sys_gpio_pin_set(BEEP_GPIO_PORT, BEEP_GPIO_PIN, x)

/* BEEP取反定义 */
#define BEEP_TOGGLE()   do{ BEEP_GPIO_PORT->ODR ^= BEEP_GPIO_PIN; }while(0)     /* BEEP = !BEEP */
```

使用蜂鸣器直接对端口进行赋值即可，BEEP(0)即表示对 PB8 赋低电平，此时蜂鸣器关闭，BEEP(1)即表示对 PB8 赋高电平，此时蜂鸣器发声。蜂鸣器取反使用 BEEP=!BEEP 或 BEEP_TOGGLE()。（对应命名在 beep.h 中查看）

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/BEEP/beep.h"        //使用蜂鸣器模块需要包含 beep 头文件

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    beep_init();                    //使用蜂鸣器模块需要 beep 初始化
    while(1)
    {
        BEEP(1);                    //蜂鸣器发声
        BEEP(0);                    //蜂鸣器关闭
        BEEP=!BEEP;                 //蜂鸣器取反
        BEEP_TOGGLE();              //蜂鸣器取反
```

```
        }
    }
```

按键

每个按键各连接一个 I/O 接口，当按键按下或松开时，对应的 I/O 接口会表现为相应的高低电平（依电路而定）。因此，单片机通过判断 I/O 接口的电平状态，能够识别对应按键是否按下或松开。按键接口电路如图 1 所示，在开发板上分别占用 **PA0、PE3、PE4** 引脚。其中按键 KEY0 和 KEY1 均为按下时低电平，松开时高电平，按键 KEY_UP 则为按下时高电平，松开时低电平。

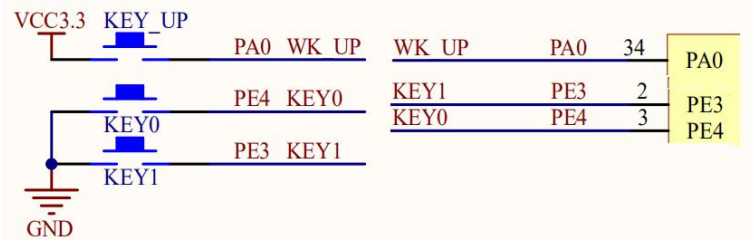


图 1 按键接口电路图

此外，开发板使用的按键为机械触点按键。由于机械触点的弹性振动，这种按键在闭合和断开的瞬间会产生抖动，抖动时间取决于按键的机械特性，一般为 5~10ms。按键抖动会造成单片机对操作次数的判断错误，需要进行消抖处理。常用的按键消抖方法是使用软件延时消抖，具体见 key_scan(0)按键检测函数。

按键使用：

首先调用 key_scan(0)函数获取键值，通过判断键值确认按键是否按下。当获取键值为 KEY0_PRES 时，表明 KEY0 按键按下；当获取键值为 KEY1_PRES 时，表明 KEY1 按键按下；当获取键值为 WKUP_PRES 时，表明 KEY_UP 按键按下。（对应键值在 key_scan(0)函数中查看）

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/KEY/key.h"          //使用按键模块需要包含按键头文件
uint8_t key;

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    key_init();                     //使用按键模块需要按键初始化
    while(1)
```

```
    {
        key=key_scan(0);                //获取键值
        if(key==KEY0_PRES)              //KEY0 按键按下
        {
        }
        if(key==KEY1_PRES)              //KEY1 按键按下
        {
        }
        if(key==WKUP_PRES)              //KEY_UP 按键按下
        {
        }
    }
}
```

LED

LED 接口电路如图 1 所示，在开发板上分别占用 **PB5 和 PE5** 引脚，其中 LED0 为红灯，LED1 为绿灯。单片机输出一个信号控制 LED 亮灭，以 LED0 为例，当 PB5=0 时红灯亮，当 PB5=1 时红灯灭。

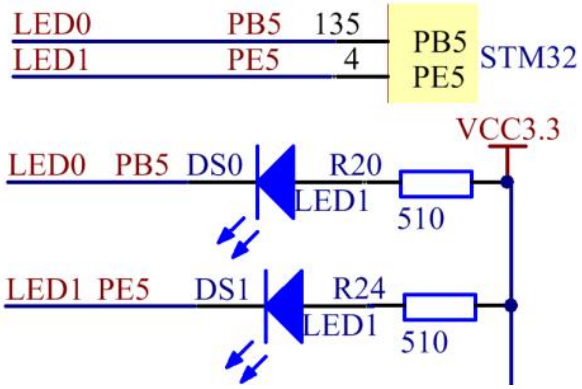


图 1 LED 接口电路图

LED 使用：

```
/* LED端口定义 */
#define LED0(x)         sys_gpio_pin_set(LED0_GPIO_PORT, LED0_GPIO_PIN, x)      /* LED0 */
#define LED1(x)         sys_gpio_pin_set(LED1_GPIO_PORT, LED1_GPIO_PIN, x)      /* LED1 */

/* LED取反定义 */
#define LED0_TOGGLE()   do{ LED0_GPIO_PORT->ODR ^= LED0_GPIO_PIN; }while(0)      /* LED0 = !LED0 */
#define LED1_TOGGLE()   do{ LED1_GPIO_PORT->ODR ^= LED1_GPIO_PIN; }while(0)      /* LED1 = !LED1 */
```

使用 LED 直接对端口进行赋值即可，以 LED0 为例，LED0(0)即表示对 PB5 赋低电平，此时红灯亮，LED0(1)即表示对 PB5 赋高电平，此时红灯灭。LED 取反使用 LED0=!LED0 和 LED1=!LED1 或 LED0_TOGGLE()和 LED1_TOGGLE()。（对应命名在 led.h 中查看）

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/LED/led.h"          //使用 led 模块需要包含 led 头文件

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    led_init();                     //使用 led 模块需要 led 初始化
    while(1)
    {
```

```
        LED0(0);                    //红灯亮
        LED1(0);                    //绿灯亮
        LED0(1);                    //红灯灭
        LED1(1);                    //绿灯灭
        LED0=!LED0；                //红灯取反
        LED1=!LED1；                //绿灯取反
        LED0_TOGGLE()；             //红灯取反
        LED1_TOGGLE()；             //红灯取反
    }
}
```

LCD

液晶显示调用函数如表 1 所示。（具体可使用函数在 lcd.h 和 lcd.c 中查看）

```
/**************************************************************************************************/
/* 函数申明 */

void lcd_wr_data(volatile uint16_t data);           /* LCD写数据 */
void lcd_wr_regno(volatile uint16_t regno);         /* LCD写寄存器编号/地址 */
void lcd_write_reg(uint16_t regno, uint16_t data);  /* LCD写寄存器的值 */


void lcd_init(void);                        /* 初始化LCD */
void lcd_display_on(void);                  /* 开显示 */
void lcd_display_off(void);                 /* 关显示 */
void lcd_scan_dir(uint8_t dir);             /* 设置屏扫描方向 */
void lcd_display_dir(uint8_t dir);          /* 设置屏幕显示方向 */
void lcd_ssd_backlight_set(uint8_t pwm);    /* SSD1963 背光控制 */

void lcd_write_ram_prepare(void);           /* 准备写GRAM */
void lcd_set_cursor(uint16_t x, uint16_t y);    /* 设置光标 */
uint32_t lcd_read_point(uint16_t x, uint16_t y);/* 读点(32位颜色,兼容LTDC) */
void lcd_draw_point(uint16_t x, uint16_t y, uint32_t color);/* 画点(32位颜色,兼容LTDC) */

void lcd_clear(uint16_t color);     /* LCD清屏 */
void lcd_fill_circle(uint16_t x, uint16_t y, uint16_t r, uint16_t color);                   /* 填充实心圆 */
void lcd_draw_circle(uint16_t x0, uint16_t y0, uint8_t r, uint16_t color);                  /* 画圆 */
void lcd_draw_hline(uint16_t x, uint16_t y, uint16_t len, uint16_t color);                  /* 画水平线 */
void lcd_set_window(uint16_t sx, uint16_t sy, uint16_t width, uint16_t height);             /* 设置窗口 */
void lcd_fill(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, uint32_t color);          /* 纯色填充矩形(32位颜色,兼容LTDC) */
void lcd_color_fill(uint16_t sx, uint16_t sy, uint16_t ex, uint16_t ey, uint16_t *color);   /* 彩色填充矩形 */
void lcd_draw_line(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t color);     /* 画直线 */
void lcd_draw_rectangle(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t color);/* 画矩形 */


void lcd_show_char(uint16_t x, uint16_t y, char chr, uint8_t size, uint8_t mode, uint16_t color);                    /* 显示一个字符 */
void lcd_show_num(uint16_t x, uint16_t y, uint32_t num, uint8_t len, uint8_t size, uint16_t color);                  /* 显示数字 */
void lcd_show_xnum(uint16_t x, uint16_t y, uint32_t num, uint8_t len, uint8_t size, uint8_t mode, uint16_t color);   /* 扩展显示数字 */
void lcd_show_string(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t size, char *p, uint16_t color);   /* 显示字符串 */
```

lcd_init()为液晶显示初始化函数，在使用液晶显示之前需要进行初始化设置；

lcd_show_num()为液晶显示数字函数，用于显示 1 位数字；

lcd_show_string()为液晶显示字符串函数，用于显示 1 组字符串，并用双引号标注显示内容，也可以首先定义所需的显示内容为 code 型字符串数组，然后调用显示。

调用相关函数显示即可，其中 3.5 寸屏尺寸为 320*480，字体大小有 12/16/24/32 四种尺寸，以 12 尺寸为例，每个字符大小为 12 像素高 6 像素宽。

函数名称	函数说明
lcd_init()	初始化
lcd_show_num(uint16_t x, uint16_t y, uint32_t num, uint8_t len, uint8_t size, uint16_t color)	显示数字（高位为 0 不显示） 其中 x、y 为起点坐标，num 为具体数值，len 为数字的位数，size 为字体大小，color 为显示颜色
lcd_show_string(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint8_t size, char *p, uint16_t color)	显示字符串 其中 x、y 为起点坐标，width、height 为区域大小，size 为字体大小，*p 为字符串起始地址，或不加*直接给出字符串，双引号标注，color 为显示颜色

#include "./SYSTEM/sys/sys.h"

```
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/LCD/lcd.h"          //使用 LCD 模块需要包含 LCD 头文件
uint8_t number=10;

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    lcd_init();                     //使用 LCD 模块需要 LCD 初始化
    while(1)
    {
        lcd_show_string(30,40,200,24,24,"Wo Ai Danpianji",RED);
//从屏幕横向 30 纵向 40 的位置开始显示，字符串长度不超过 200 像素，高度为 24 像素，字体大小为 24，显示 Wo Ai Danpianji 字符串并用双引号标注，颜色为红色
        lcd_show_num(30,70,number,2,24,RED);
//从屏幕横向 30 纵向 70 的位置开始显示，显示数字参数为 number，数字位数为 2 位数，字体大小为 24，颜色为红色
    }
}
```

外部中断

中断系统是单片机的重要组成部分，其中断响应和处理过程如图 1 所示。当中断源发出的中断请求被允许时，单片机会暂时中止执行当前程序，转而执行中断服务程序处理中断服务请求，处理结束后再回到原来中止程序位置，继续执行被中断的程序。由于中断系统工作方式的实时特性，常用于实现实时控制、故障处理、紧急停止等需要快速响应和及时处理的控制功能，大大提高了单片机的工作效率。

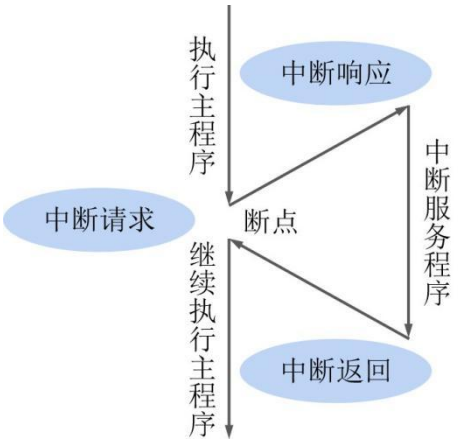


图 1 中断响应和处理过程示意图

其中外部中断是由外部信号产生的中断请求，STM32 所有引脚均可作为外部中断使用，触发方式为跳沿触发方式。本门课程可以选用 KEY0、KEY1、KEY_UP 按键作为外部中断触发。

当某一按键作为外部中断使用时，需要在 extix_init()中保留该按键的外部中断设置，并注释其余按键的外部中断设置。以 KEY0 按键作为外部中断触发，其余按键作为按键使用为例：

```
void extix_init(void)
{
    key_init();
    sys_nvic_ex_config(KEY0_INT_GPIO_PORT,    KEY0_INT_GPIO_PIN,
SYS_GPIO_FTIR);    /* KEY0 配置为下降沿触发中断 */
//      sys_nvic_ex_config(KEY1_INT_GPIO_PORT, KEY1_INT_GPIO_PIN,
SYS_GPIO_FTIR);    /* KEY1 配置为下降沿触发中断 */
//      sys_nvic_ex_config(WKUP_INT_GPIO_PORT, WKUP_INT_GPIO_PIN,
SYS_GPIO_RTIR);    /* WKUP 配置为上升沿触发中断 */

    sys_nvic_init( 0, 2, KEY0_INT_IRQn, 2); /* 抢占 0，子优先级 2，组 2 */
```

```
//      sys_nvic_init( 1, 2, KEY1_INT_IRQn, 2); /*  抢占 1，子优先级 2，组 2 */
//     sys_nvic_init( 3, 2, WKUP_INT_IRQn, 2); /*  抢占 3，子优先级 2，组 2 */
}
```

中断优先级拓展：

STM32 的中断向量具有两个属性，一个为抢占属性，一个为响应属性，其属性编号越小，表明它的优先级别越高。先看抢占再看响应。

当外部中断发出中断请求时，单片机将会响应该中断请求并自动跳转执行中断服务程序。中断服务程序不需要进行声明，也不需要进行调用，其建立的一般形式为：

void EXTIx_IRQHandler(void)

其中 x 为中断号，用于识别不同的中断源并指向对应的中断地址，IRQHandler 则表示该函数为中断函数。KEY0 按键占用 PE4 引脚，可以建立中断服务程序为 void EXTI4_IRQHandler(void)，KEY1 按键占用 PE3 引脚，可以建立中断服务程序为 void EXTI3_IRQHandler(void)，KEY_UP 按键占用 PA0 引脚 KEY0，可以建立中断服务程序为 void EXTI0_IRQHandler(void)。

相应程序如下：

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/EXTI/exti.h"        //使用外部中断需要包含相应头文件

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);         //三个系统初始化都先放着
    extix_init();                   //使用外部中断需要相应初始化
    while(1)
    {
    }
}

//KEY0  外部中断服务程序
void KEY0_INT_IRQHandler(void)
{
     delay_ms(20);                  /*  消抖  */
```

```
        EXTI->PR = KEY0_INT_GPIO_PIN;      /* 清除 KEY0 所在中断线的中断标志位 */

        if (KEY0 == 0)
        {
        }
    }

    //KEY1 外部中断服务程序
    void KEY1_INT_IRQHandler(void)
    {
        delay_ms(20);                          /* 消抖 */
        EXTI->PR = KEY1_INT_GPIO_PIN;      /* 清除 KEY1 所在中断线的中断标志位 */

        if (KEY1 == 0)
        {
        }
    }

    //WK_UP 外部中断服务程序
    void WKUP_INT_IRQHandler(void)
    {
        delay_ms(20);                          /* 消抖 */
        EXTI->PR = WKUP_INT_GPIO_PIN;      /* 清除 WKUP 所在中断线的中断标志位 */

        if (WK_UP == 1)
        {
        }
    }
```

使用外部中断存在一些注意事项和使用技巧，总结如下：（1）当外部中断发出的中断请求被允许时，单片机将会响应该中断请求并自动跳转执行中断服务程序，因此中断触发按键不需要进行检测，中断服务程序也不需要进行声明和调用，进行中断初始化配置并建立中断服务程序即可；（2）为了快速处理中断请求以便继续响应其他请求，通常情况下中断服务程序根据功能要求仅改变标志位数值，回到系统主程序再根据标志位数值执行相应系统输出。

串口通信

本门课程使用的单片机包含 3 个 USART（通用同步异步收发器）和 2 个 UART（通用异步收发器），使用 2 个数据寄存器 DR（发送 TDR，接收 RDR）接收和发送数据，接收信息和发送结束时会置相应标志位（接收信息 RXNE，发送完成 TC），其中发送结束置标志位会产生中断请求，如果中断请求被允许的话，单片机将会响应该中断请求并自动跳转执行中断服务程序。

单片机发送

调用 printf()函数即可，需要发送的字符串使用双引号标注，参数使用%d 指代。以发送字符串 Wo Ai Danpianji,10 为例：

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
Uint8_t number=10;

void main()
{
    sys_stm32_clock_init(9);
```

```
    delay_init(72);
    usart_init(72, 115200);              //三个系统初始化都先放着
    while(1)
    {
        printf("Wo Ai Danpianji,%d",number);
    }
}
```

单片机接收

```
extern uint8_t  g_usart_rx_buf[USART_REC_LEN];  /* 接收缓冲,最大USART_REC_LEN个字节.末字节为换行符 */
extern uint16_t g_usart_rx_sta;                 /* 接收状态标记 */
```

g_usart_rx_sta 为接收状态标记，为 16 为寄存器。其中第 0~13 位为接收到的有效字节数目，第 14 位为接收到 0x0d 标记位，第 15 位为接收完成标志位。通过判断 g_usart_rx_sta 的最高位是否为 1 确认单片机是否接收到信息，接受信息结束后需要手动将 g_usart_rx_sta 清零。

```
/*  接收状态
 *  bit15,      接收完成标志
 *  bit14,      接收到0x0d
 *  bit13~0,    接收到的有效字节数目
*/
```

g_usart_rx_buf[]为接收缓冲数组，用于存放接收信息，最大可存放 200 个字节。如果 g_usart_rx_sta 的最高位为 1，即单片机接收到信息，从 g_usart_rx_buf[]当中读取使用。需要注意的是单片机接收到的信息为 ASCII 码，需要进行转换（+48）或以字符形式使用（单引号标注）。

```
/* 接收缓冲, 最大USART_REC_LEN个字节. */
uint8_t g_usart_rx_buf[USART_REC_LEN];
```

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);              //三个系统初始化都先放着
    while(1)
    {
        if(g_usart_rx_sta & 0x8000)   //最高位为 1，单片机接收到信息
        {
            if(g_usart_rx_buf[0]=='a') //接收到的信息为 a
            {
```

```
                }
                else if(g_usart_rx_buf[0]=='1')    //接收到的信息为 1
                {
                }
                g_usart_rx_sta =0;                 //标志位手动清零
            }
        }
    }
```

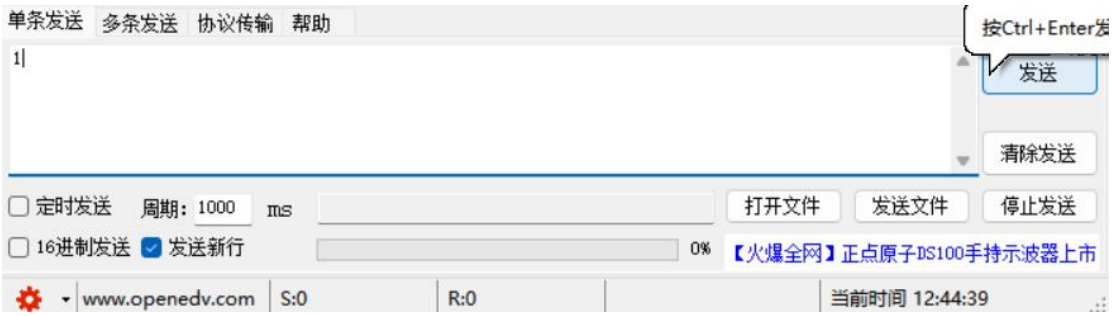
串口助手

选择串口号

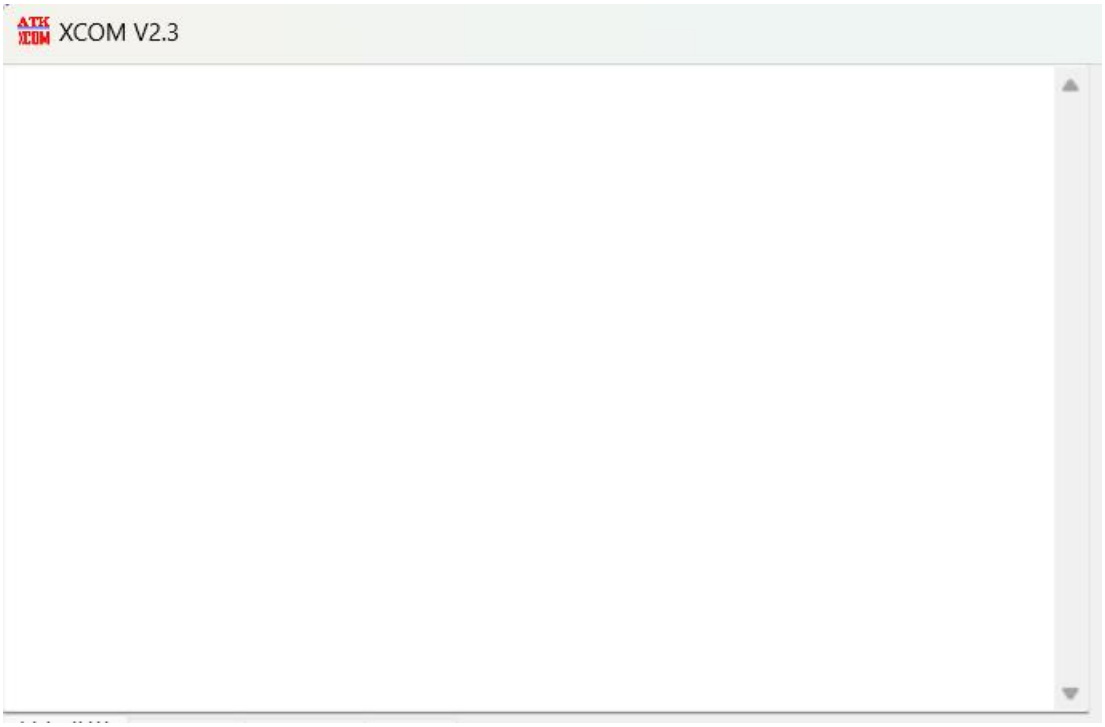
选择波特率

打开串口

发送消息


单条发送
多条发送
协议传输
帮助
1
按Ctrl+Enter发
发送
清除发送
定时发送
周期: 1000
ms
打开文件
发送文件
停止发送
16进制发送
发送新行
0%
【火爆全网】正点原子DS100手持示波器上市
www.openedv.com
S:0
R:0
当前时间 12:44:39

接收信息


XCOM V2.3

定时器

本门课程使用的单片机包含 2 个基本定时器 TIM6、TIM7，4 个通用定时器 TIM2、TIM3、TIM4、TIM5，2 个高级定时器 TIM1、TIM8，课程使用通用定时器即可，功能包括定时中断、PWM 输出、输入捕获、脉冲计数。

修改 gtim.h 文件使用 TIM4 做定时中断、TIM3 做 PWM 输出（PA6、PA7）、TIM5 做输入捕获（KEY_UP）、TIM2 做脉冲计数（KEY_UP）。

定时中断

定时器本质上是对脉冲信号进行计数，其中定时是对内部时钟信号进行计数，设置 arr 和 psc 初值之后，每隔一定时间溢出时会产生中断请求触发中断响应进行中断处理。同样地，定时器中断服务程序也不需要进行声明和调用，进行定时器初始化配置并建立对应的中断服务程序即可。

定时器打开后停止：

```
GTIM_TIMX_INT->CR1 &= 0xFE;
GTIM_TIMX_CNT->CR1 &= 0xFE;
```

定时器停止后打开：

```
GTIM_TIMX_INT->CR1 |= 1 << 0;
GTIM_TIMX_CNT->CR1 |= 1 << 0;
```

定时器定时初始化：

```
gtim_timx_int_init(10000-1, 7200-1);
```

其中 arr=10000-1，psc=7200-1，定时计算公式为(arr+1)*(psc+1)/72MHz，此处为(9999+1)*(7199+1)/72*10^6=1s，即每隔 1s 进入一次定时器中断服务程序。

```
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"   //三个系统头文件都先包含
#include "./BSP/TIMER/gtim.h"       //使用通用定时器需要包含相应头文件

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);             //三个系统初始化都先放着
    gtim_timx_int_init(10000-1, 7200-1);//不用定时器不开中断
    while(1)
```

```
    {
    }
}
void GTIM_TIMX_INT_IRQHandler(void)
{
    if (GTIM_TIMX_INT->SR & 0X0001)     /* 溢出中断 */
    {
    }
    GTIM_TIMX_INT->SR &= ~(1 << 0); /* 清除中断标志位 */
}
```

多个时间定时

1、需要定时几个时间就设置几个参数，每个参数之间互不干扰，适用于不同模块的时间定时。以 BEEP 响 1 秒，LED1 亮 2 秒为例：

```
#include "./BSP/TIMER/gtim.h"
uint8_t count_1s,count_2s;

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);                 //三个系统初始化都先放着
    gtim_timx_int_init(10000-1, 7200-1);
    while(1)
    {
    }
}
void GTIM_TIMX_INT_IRQHandler(void)
{
    if (GTIM_TIMX_INT->SR & 0X0001)     /* 溢出中断 */
    {
        if(++count_1s==1)
        {
            BEEP=0;
            count_1s=0;
        }
        if(++count_2s==2)
        {
            LED1=1;
            count_2s=0;
        }
    }
    GTIM_TIMX_INT->SR &= ~(1 << 0); /* 清除中断标志位 */
```

```
}
```

2、将多个时间以 if else if 的并列形式处理，适用于具有一定时间序列的功能处理。以 LED0 亮 1 秒灭 2 秒，LED1 亮 1 秒灭 2 秒，循环往复，为例：

```
#include "./BSP/TIMER/gtim.h"
uint8_t count;

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);                 //三个系统初始化都先放着
    gtim_timx_int_init(10000-1, 7200-1);
    LED0(0);
    while(1)
    {
    }
}

void GTIM_TIMX_INT_IRQHandler(void)
{
    if (GTIM_TIMX_INT->SR & 0X0001)     /* 溢出中断 */
    {
        count++;
        if(count==1)
        {
            LED0(1);
        }
        else if(count==3)
        {
            LED1(0);
        }
        else if(count==4)
        {
            LED1(1);
        }
        else if(count==6)
        {
            LED0(0);
            count=0;
        }
    }
    GTIM_TIMX_INT->SR &= ~(1 << 0); /* 清除中断标志位 */
}
```

3、将多个时间以参数嵌套的形式处理，适用于定时不同量级的时间处理。以 BEEP 响 1 秒，LED0 以 100ms 的间隔闪烁 4 秒熄灭为例：

```
#include "./BSP/TIMER/gtim.h"
uint8_t count_100ms,count_1s;
uint8_t LED0_flag;

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);                 //三个系统初始化都先放着
    gtim_timx_int_init(1000-1, 7200-1);
    BEEP(1);
    LED0_flag=1;
    while(1)
    {
    }
}

void GTIM_TIMX_INT_IRQHandler(void)
{
    if (GTIM_TIMX_INT->SR & 0X0001)     /* 溢出中断 */
    {
        count_100ms++;
        if(LED0_flag==1)LED0=!LED0;
        if(count_100ms==10)
        {
            count_100ms=0;
            BEEP(0);
            count_1s++;
            if(count_1s==4)
            {
                count_1s=0;
                LED0(1);
                LED0_flag=0;
            }
        }
    }
    GTIM_TIMX_INT->SR &= ~(1 << 0); /* 清除中断标志位 */
}
```

另外，当定时器持续工作时，定时开始需要清零一系列累加参数或开关定时器以得到精确的定时时间。

PWM 输出

使用 TIM3 没有重映像下的 CH1 和 CH2 进行 PWM 输出。

复用功能	TIM3_REMAP[1:0] = 00 (没有重映像)	TIM3_REMAP[1:0] = 10 (部分重映像)	TIM3_REMAP[1:0] = 11 (完全重映像)(1)
TIM3_CH1	PA6	PB4	PC6
TIM3_CH2	PA7	PB5	PC7
TIM3_CH3	PB0		PC8
TIM3_CH4	PB1		PC9

```
/* TIMX PWM输出定义
 * 这里输出的PWM控制MOTOR
 * 默认是针对TIM2~TIM5
 * 注意: 通过修改这几个宏定义,可以支持TIM1~TIM8任意一个定时器,任意一个IO口输出PWM
 */
#define GTIM_TIMX_PWM_CHY_GPIO_PORT          GPIOA
#define GTIM_TIMX_PWM_CH1_GPIO_PIN           SYS_GPIO_PIN6
#define GTIM_TIMX_PWM_CH2_GPIO_PIN           SYS_GPIO_PIN7
#define GTIM_TIMX_PWM_CHY_GPIO_CLK_ENABLE() do{ RCC->APB2ENR |= 1 << 2; }while(0)   /* PA口时钟使能 */

/* TIMX REMAP设置
 * 因为我们MOTOR接在PA6和PA7上, 必须通过开启TIM3的没有重映射功能, 才能将TIM3_CH1和TIM3_CH2输出到PA6和PA7上
 * 因此, 必须实现GTIM_TIMX_PWM_CHY_GPIO_REMAP, 通过sys_gpio_remap_set函数设置重映射
 * 对那些使用默认设置的定时器PWM输出脚, 不用设置重映射, 是不需要该函数的!
 */
#define GTIM_TIMX_PWM_CHY_GPIO_REMAP()       sys_gpio_remap_set(10, 2, 0)           /* 通道REMAP设置, 该函数不是必须的, 根据需要实现 */

#define GTIM_TIMX_PWM                        TIM3
#define GTIM_TIMX_PWM_CH1                    1                          /* 通道1,  1<= Y <=4 */
#define GTIM_TIMX_PWM_CH2                    2                          /* 通道2,  1<= Y <=4 */
#define GTIM_TIMX_PWM_CH1_CCRX               TIM3->CCR1                 /* 通道1的输出比较寄存器 */
#define GTIM_TIMX_PWM_CH2_CCRX               TIM3->CCR2                 /* 通道2的输出比较寄存器 */
#define GTIM_TIMX_PWM_CHY_CLK_ENABLE()       do{ RCC->APB1ENR |= 1 << 1; }while(0)   /* TIM3 时钟使能 */
```

使用定时器 TIM3 的通道 1 和通道 2，对 PA6 和 PA7 引脚进行 PWM 输出。定时器 TIM3 单个时间周期即为电机输出的单个时间周期，分频值即为单个时间周期的划分值。

例如设定电机周期为 10ms，调速范围为 0-100，则设定定时器 TIM3 单个时间周期为 10ms，每个时间周期分为 100 份（arr=99），即 gtim_timx_pwm_chy_init (100-1,7200-1);。另外设定 GTIM_TIMX_PWM_CH1_CCRX 为 TIM3->CCR1，对应 PA6 引脚，GTIM_TIMX_PWM_CH2_CCRX 为 TIM3->CCR2，对应 PA7 引脚，通过设置 GTIM_TIMX_PWM_CH1_CCRX 和 GTIM_TIMX_PWM_CH2_CCRX 参数即可对 PA6 或 PA7 输出对应占空比的 PWM 波形，实现电机正反转调速。

```
#include "./BSP/TIMER/gtim.h"

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);                 //三个系统初始化都先放着
    gtim_timx_pwm_chy_init(100-1, 7200-1);       //10ms 分 100 份
    while(1)
    {
        GTIM_TIMX_PWM_CH1_CCRX=0;
```

```
            GTIM_TIMX_PWM_CH2_CCRX=0;       //电机停止

            GTIM_TIMX_PWM_CH1_CCRX=50;
            GTIM_TIMX_PWM_CH2_CCRX=0;       //电机以 50 速度正转

            GTIM_TIMX_PWM_CH1_CCRX=0;
            GTIM_TIMX_PWM_CH2_CCRX=50;      //电机以 50 速度反转
        }
    }
```

输入捕获

```
/* TIMX 输入捕获定义
 * 这里的输入捕获使用定时器TIM5_CH1,捕获WK_UP按键的输入
 * 默认是针对TIM2~TIM5.
 * 注意: 通过修改这几个宏定义,可以支持TIM1~TIM8任意一个定时器,任意一个IO口做输入捕获
 *       特别要注意:默认用的PA0,设置的是下拉输入!如果改其他IO,对应的上下拉方式也得改!
 */
#define GTIM_TIMX_CAP_CHY_GPIO_PORT             GPIOA
#define GTIM_TIMX_CAP_CHY_GPIO_PIN              SYS_GPIO_PIN0
#define GTIM_TIMX_CAP_CHY_GPIO_CLK_ENABLE()     do{ RCC->APB1ENR |= 1 << 2; }while(0)   /* PA口时钟使能 */

#define GTIM_TIMX_CAP                           TIM5
#define GTIM_TIMX_CAP_IRQn                      TIM5_IRQn
#define GTIM_TIMX_CAP_IRQHandler                TIM5_IRQHandler
#define GTIM_TIMX_CAP_CHY                       1                       /* 通道Y,  1<= Y <=4 */
#define GTIM_TIMX_CAP_CHY_CCRX                  TIM5->CCR1              /* 通道Y的输出比较寄存器 */
#define GTIM_TIMX_CAP_CHY_CLK_ENABLE()          do{ RCC->APB1ENR |= 1 << 3; }while(0)    /* TIM5 时钟使能 */
```

使用定时器 TIM5 的通道 1，对 KEY_UP 按键进行输入捕获。定时器 TIM5 单个时间周期在不分频情况下即为输入捕获计数的单个时间周期，例如 gtim_timx_cap_chy_init(0XFFFF, 7200-1);即设定定时器 TIM5 对 KEY_UP 按键输入捕获单次计数为 0.1ms。

```
/* 输入捕获状态(g_timxchy_cap_sta)
 * [7]  :0,没有成功的捕获;1,成功捕获到一次.
 * [6]  :0,还没捕获到高电平;1,已经捕获到高电平了.
 * [5:0]:捕获高电平后溢出的次数,最多溢出63次,所以最长捕获值 = 63*65536 + 65535 = 4194303
 *       注意:为了通用,我们默认ARR和CCRy都是16位寄存器,对于32位的定时器(如:TIM5),也只按16位使用
 *       按1us的计数频率,最长溢出时间为:4194303 us, 约4.19秒
 */
uint8_t g_timxchy_cap_sta = 0;  /* 输入捕获状态 */
uint32_t g_timxchy_cap_val =0 ; /* 输入捕获值 */
```

g_timxchy_cap_sta 为输入捕获状态标记，为 8 位寄存器。其中第 0~5 位为溢出次数，第 6 位为捕获到高电平标志位，第 7 位为成功捕获标志位。通过判断 g_timxchy_cap_sta 的最高位是否为 1 确认是否成功捕获到一次高电平，得到时间后需要手动将 g_timxchy_cap_sta 清零。

g_timxchy_cap_val 为输入捕获值，用于存放最后一次时间周期输入捕获的数值，g_timxchy_cap_val = GTIM_TIMX_CAP_CHY_CCRX。

```
#include "./BSP/TIMER/gtim.h"

extern uint8_t    g_timxchy_cap_sta;   //输入捕获状态
```

```
extern uint16_t g_timxchy_cap_val;    //输入捕获值

void main()
{
    sys_stm32_clock_init(9);
    delay_init(72);
    usart_init(72, 115200);              //三个系统初始化都先放着
    gtim_timx_cap_chy_init(0XFFFF, 7200-1);      //0.1ms
    while(1)
    {
        if (g_timxchy_cap_sta & 0X80)     //成功捕获到了一次高电平
        {
        temp = g_timxchy_cap_sta & 0X3F;    //获取溢出次数
        temp *= 65536;                      //得到溢出时间
        temp += g_timxchy_cap_val; //加最后一次时间得到总的高电平时间
        g_timxchy_cap_sta = 0;              //开启下一次捕获
        }
    }
}
```

脉冲计数

```
/* TIMX 输入计数定义
* 这里的输入计数使用定时器TIM2_CH1,捕获WK_UP按键的输入
* 默认是针对TIM2~TIM5, 只有CH1和CH2通道可以用做输入计数, CH3/CH4不支持!
* 注意: 通过修改这几个宏定义,可以支持TIM1~TIM8任意一个定时器,CH1/CH2对应IO口做输入计数
*       特别要注意:默认用的PA0,设置的是下拉输入!如果改其他IO,对应的上下拉方式也得改!
*/
#define GTIM_TIMX_CNT_CHY_GPIO_PORT            GPIOA
#define GTIM_TIMX_CNT_CHY_GPIO_PIN             SYS_GPIO_PIN0
#define GTIM_TIMX_CNT_CHY_GPIO_CLK_ENABLE()    do{ RCC->APB1ENR |= 1 << 0; }while(0)    /* PA口时钟使能 */

#define GTIM_TIMX_CNT                          TIM2
#define GTIM_TIMX_CNT_IRQn                     TIM2_IRQn
#define GTIM_TIMX_CNT_IRQHandler               TIM2_IRQHandler
#define GTIM_TIMX_CNT_CHY                      1                              /* 通道Y,  1<= Y <=2 */
#define GTIM_TIMX_CNT_CHY_CLK_ENABLE()         do{ RCC->APB1ENR |= 1 << 0; }while(0)    /* TIM2 时钟使能 */
```

定时器本质上是对脉冲信号进行计数，如果说定时是对内部时钟信号进行计数，计数则是对外部输入信号进行计数。这里使用定时器 TIM2 的通道 1，对 KEY_UP 按键进行脉冲计数。

```
void gtim_timx_cnt_chy_init(uint16_t psc);                  /* 通用定时器 脉冲计数初始化函数 */
uint32_t gtim_timx_cnt_chy_get_count(void);                 /* 通用定时器 获取脉冲计数 */
void gtim_timx_cnt_chy_restart(void);                       /* 通用定时器 重启计数器 */
```

脉冲计数包含三个函数，分别为脉冲计数初始化函数、获取计数值函数、重启计数器函数，其中初始化 psc 可以选择 0，即不分频：gtim_timx_cnt_chy_init(0)。使用时调用 gtim_timx_cnt_chy_get_count()获取计数值，清零时调用 gtim_timx_cnt_chy_restart()重启计数器即可。