定时器

实验目的：
1，学习使用单片机的定时器，体会定时器的好处。
2，了解中断向量表，中断服务函数，函数指针

实现功能：
在上一节蜂鸣器中用空指令延时实现驱动蜂鸣器鸣叫，这里的延时时间是不好精确控制的，那么如果我们要让 PD4 输出高 0.5s——输出低 0.5s——输出高 0.5s——输出低 0.5s 循环交替，该怎么做呢？

思考：
像之前使用逻辑分析仪得到时间那样，反复校准，可以调整最终使空指令 delay()函数实现我们要的功能。显然这是一种费力不讨好的事情，因为校准得花时间，而且一旦 0.5s 不满足要求，要改成 0.55s，那么又得重新校准，，这，想想就头大!!!
所以定时器 timer 是一个好东西，它可以实现比较精确的计时，有了 timer，像前面说的 0.5s、0.55s 的延时都能够简单就可以实现了。

基础知识讲解
1.1，对于单片机来讲，定时器是比较简洁的实现较精确计时的资源，一般单片机中都会集成定时器，stm8s103f3p6 片内有 2 个 16bit 定时器和 1 个 8bit 定时器，本实验中使用 8bit 的 timer4.
1.2, 定时器的使用如之前使用 gpio 或 beep 一样，需要先初始化，使能之后才能按设计工作，有一点不同的是，我们一般用定时器会用到定时器中断，然后在定时器中断服务函数中执行计时时间到之后需要执行的动作。
在新建 stvd 工程中有提到，stm8_interrupt_vector.c 文件中有 stm8 的中断向量表，该中断向量表保存着对应每个中断的中断服务函数的指针(地址)，如果 cpu 发生了中断(同时使能中断)，cpu 将会到中断向量表中查询获得中断服务函数指针(地址), 然后跳转执行中断服务函数，等中断服务函数执行完再回到之前被中断的地方继续执行程序。

7 中断向量映像

表10 中断向量表

中断编号	中断源	说明	从停机模式唤醒	从活跃停机模式唤醒	向量地址
	RESET	复位	是	是	0x00 8000
	TRAP	软件中断	-	-	0x00 8004
0	TLI	外部最高级中断	-	-	0x00 8008
1	AWU	从停机模式自动唤醒	-	是	0x00 800C
2	CLK	时钟控制器	-	-	0x00 8010
3	EXTI0	端口A外部中断	是(1)	是(1)	0x00 8014
4	EXTI1	端口B外部中断	是	是	0x00 8018
5	EXTI2	端口C外部中断	是	是	0x00 801C
6	EXTI3	端口D外部中断	是	是	0x00 8020
7	EXTI4	端口E外部中断	是	是	0x00 8024
8		保留	-	-	0x00 8028
9		保留	-	-	0x00 802C
10	SPI	传输结束	是	是	0x00 8030

10	SPI	传输结束	是	是	0x00 8030
11	TIM1	定时器1 更新/上溢出/下溢出/触发/刹车	-	-	0x00 8034
12	TIM1	定时器1 捕获/比较	-	-	0x00 8038
13	TIM2	定时器2 更新/上溢出	-	-	0x00 803C
14	TIM2	定时器2 捕获/比较	-	-	0x00 8040
15		保留	-	-	0x00 8044
16		保留	-	-	0x00 8048
17	UART1	发送完成	-	-	0x00 804C
18	UART1	接收寄存器数据满	-	-	0x00 8050
19	I^2C	I^2C中断	是	是	0x00 8054
20		保留	-	-	0x00 8058
21		保留	-	-	0x00 805C
22	ADC1	ADC1转换结束/模拟看门狗中断	-	-	0x00 8060
23	TIM4	定时器4 更新/上溢出	-	-	0x00 8064
24	FLASH	EOP/WR_PG_DIS	-	-	0x00 8068
保留					0x00 806C 至 0x00 807C

stm8_interrupt_vector.c 文件中注册的中断向量表：

```
struct interrupt_vector const _vectab[] = {
  {0x82, (interrupt_handler_t)_stext}, /* RESET */
  {0x82, (interrupt_handler_t)TRAP_IRQHandler}, /* TRAP - Software interrupt */
    {0x82, (interrupt_handler_t)TLI_IRQHandler}, /* irq0 - External Top Level interrupt (TLI) */
    {0x82, (interrupt_handler_t)AWU_IRQHandler}, /* irq1 - Auto Wake Up from Halt interrupt */
    {0x82, (interrupt_handler_t)CLK_IRQHandler}, /* irq2 - Clock Controller interrupt */
  {0x82, (interrupt_handler_t)EXTI_PORTA_IRQHandler}, /* irq3 - External interrupt 0 (GPIOA) */
    {0x82, (interrupt_handler_t)EXTI_PORTB_IRQHandler}, /* irq4 - External interrupt 1 (GPIOB) */
    {0x82, (interrupt_handler_t)EXTI_PORTC_IRQHandler}, /* irq5 - External interrupt 2 (GPIOC) */
    {0x82, (interrupt_handler_t)EXTI_PORTD_IRQHandler}, /* irq6 - External interrupt 3 (GPIOD) */
    {0x82, (interrupt_handler_t)EXTI_PORTE_IRQHandler}, /* irq7 - External interrupt 4 (GPIOE) */

#if defined (STM8S208) || defined (STM8AF52Ax)
    {0x82, (interrupt_handler_t)CAN_RX_IRQHandler}, /* irq8 - CAN Rx interrupt */
    {0x82, (interrupt_handler_t)CAN_TX_IRQHandler}, /* irq9 - CAN Tx/ER/SC interrupt */
#elif defined (STM8S903)
    {0x82, (interrupt_handler_t)EXTI_PORTF_IRQHandler}, /* irq8 - External interrupt 5 (GPIOF) */
    {0x82, (interrupt_handler_t)NonHandledInterrupt}, /* irq9 - Reserved */
#else /*STM8S207, STM8S105 or STM8AF62Ax or STM8AF626x*/
    {0x82, (interrupt_handler_t)NonHandledInterrupt}, /* irq8 - Reserved */
    {0x82, (interrupt_handler_t)NonHandledInterrupt}, /* irq9 - Reserved */
#endif /* STM8S208 or STM8AF52Ax */
    {0x82, (interrupt_handler_t)SPI_IRQHandler}, /* irq10 - SPI End of transfer interrupt */
    {0x82, (interrupt_handler_t)TIM1_UPD_OVF_TRG_BRK_IRQHandler}, /* irq11 - TIM1 Update/Overflow/Trigger/Br
    {0x82, (interrupt_handler_t)TIM1_CAP_COM_IRQHandler}, /* irq12 - TIM1 Capture/Compare interrupt */

#ifdef STM8S903
  {0x82, (interrupt_handler_t)TIM5_UPD_OVF_BRK_TRG_IRQHandler}, /* irq13 - TIM5 Update/Overflow/Break/Trigge
  {0x82, (interrupt_handler_t)TIM5_CAP_COM_IRQHandler}, /* irq14 - TIM5 Capture/Compare interrupt */

#else /*STM8S208, STM8S207, STM8S105 or STM8S103 or STM8AF62Ax or STM8AF52Ax or STM8AF626x*/
  {0x82, (interrupt_handler_t)TIM2_UPD_OVF_BRK_IRQHandler}, /* irq13 - TIM2 Update/Overflow/Break interrupt
  {0x82, (interrupt_handler_t)TIM2_CAP_COM_IRQHandler}, /* irq14 - TIM2 Capture/Compare interrupt */
#endif /*STM8S903*/
```

例如本实验将要用到的的定时器 4，按照手册中的中断向量表（上图），定时器 4 的中断号为 23，中断向量地址为 0x008064，在 stm8_interrupt_vector.c 中可以找到中断号为 23 的定义：

```
#ifdef STM8S903
    {0x82, (interrupt_handler_t)TIM6_UPD_OVF_TRG_IRQHandler}, /* irq23 - TIM6 Update/Overflow/Trigger interrupt */
#else
    {0x82, (interrupt_handler_t)TIM4_UPD_OVF_IRQHandler}, /* irq23 - TIM4 Update/Overflow interrupt */
#endif /*STM8S903*/
```

所以定时器的中断服务函数指针为：(interrupt_handler_t)TIM4_UPD_OVF_IRQHandler，对应的函数实现在 stm8s_it.c 中：

```
INTERRUPT_HANDLER(TIM4_UPD_OVF_IRQHandler, 23)
{
 /* In order to detect unexpected events during development,
    it is recommended to set a breakpoint on the following instruction.
 */

}
```

程序一般是在 main()函数中执行（一般是 while()循环函数中），一旦发生定时器 4 中断，cpu 就会根据 stm8_interrupt_vector.c 内中断向量表的定义找到中断服务函数指针(interrupt_handler_t)TIM4_UPD_OVF_IRQHandle ， 然后通过指针跳转执行函数 INTERRUPT_HANDLER(TIM4_UPD_OVF_IRQHandler, 23)，等中断服务函数执行完毕再回到原来被中断的地方(while()循环中)继续执行原来的程序。

下面进入正文：
大家应该都想到了用定时器来实现精确计时，下面就是我用定时器 4 实现的让 PA1 (led)输出高 0.5s——输出低 0.5s——输出高 0.5s——输出低 0.5s 循环交替，请看代码：

Stm8s_it.c:

```
INTERRUPT_HANDLER(TIM4_UPD_OVF_IRQHandler, 23)
{
 /* In order to detect unexpected events during development,
    it is recommended to set a breakpoint on the following instruction.
 */
   static u16 cnt = 0;   // 局部静态变量，函数退出空间不回收

   cnt++;
   if(cnt >= 5000) // 0.1ms *5000 = 0.5s
   {
       cnt = 0;
       GPIO_WriteReverse(GPIOA, GPIO_PIN_1); // PA1 输出电平翻转
   }

   TIM4_ClearITPendingBit(TIM4_IT_UPDATE);   //  清除中断标志位
}
```

Main.c:

```c
/**
* author : tianyx
* email  : zzztyx55@sina.com
* qq     : 609421258
* github : https://github.com/zzztyx55
*/
/* Private defines -----------------------------------------------------------*/
/* Private function prototypes -----------------------------------------------*/
/* Private functions ---------------------------------------------------------*/
void delay(u32 dly)
{
    u32 i;
    for(i = 0; i < dly; i++)
        ; //空指令
}

void clk_config(void)
{
    // HSI 时钟 16MHz
    CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);  // HSI时钟预分频, 分频系数1
    CLK_SYSCLKConfig(CLK_PRESCALER_HSIDIV1);  // 系统时钟配置, HSI, 分频系数1
    CLK_HSICmd(ENABLE);   // 使能HSI
    while(RESET == CLK_GetFlagStatus(CLK_FLAG_HSIRDY));  // 等待HSI ready
}

/*
* stm8s103f3p6 的定时器4使用的是 系统时钟
* 本程序中将系统时钟配置为HSI = 16MHz
*/
void Init_Timer4(void)
{
    // 配置定时器4的预分频值16分频和计数值100
    // 定时器计数到100后就会产生溢出中断, 中断完之后又重新从0开始计数
    // time4 定时中断周期T = 100*(16/16MHz) = 0.1ms
    // 所以中断服务函数就会每0.1ms被调用一次
    TIM4_TimeBaseInit(TIM4_PRESCALER_16, 100);
     /* Clear TIM4 update flag */
    TIM4_ClearFlag(TIM4_FLAG_UPDATE);
     /* Enable update interrupt */
    TIM4_ITConfig(TIM4_IT_UPDATE, ENABLE);  // 使能time4溢出更新中断
    TIM4_Cmd(ENABLE);  // 使能timer4
}

/**
* author : tianyx
* email  : zzztyx55@sina.com
* qq     : 609421258
* github : https://github.com/zzztyx55
*/
void main(void)
{
    // 关中断
    disableInterrupts();
    // 系统时钟配置
    clk_config();
    // init: PA1 - led, output high
    GPIO_DeInit(GPIOA);
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);

    Init_Timer4();  // 初始化并使能time4
    // 开中断
    enableInterrupts();

    /* Infinite loop */
    // while循环不做实际任务, 仅仅是防止程序退出, 进行空循环
    // 实际的任务在timer4 的中断服务函数中进行, 见 stm8s_it.c 中 TIM4_UPD_OVF_IRQHand
    while (1)
    {
    }
} « end main »
```

3, 程序中使用了定时器 4 的库函数，需要将库文件 stm8s_tim4.c 添加到工程中，
编译调试，效果：led 亮 0.5s——灭 0.5s——亮 0.5s——灭 0.5s 循环交替

4，上一讲中卖了个关子，说怎么样实现精确的延时，同时又要少让 cpu 做无用功。
现在我们知道了怎么样实现精确的计时，但是怎么样做到精确的延时，同时又少做无用功呢？

现在假如说有这样一个需求：
Led 每隔 10s 点亮 0.5s，每当按键按下就鸣叫蜂鸣器 0.5s。

假如代码写成下面这样：

```
/**
* author : tianyx
* email  : zzztyx55@sina.com
* qq     : 609421258
* github : https://github.com/zzztyx55
*/
void main(void)
{
    // 关中断
    disableInterrupts();
    // 系统时钟配置
    clk_config();

    GPIO_DeInit(GPIOA);
    GPIO_DeInit(GPIOB);
    GPIO_DeInit(GPIOC);
    GPIO_DeInit(GPIOD);

    // init: PA1 - led, output high
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);
    GPIO_WriteHigh(GPIOA, GPIO_PIN_1); // turn off led
    // PB4, key input
    GPIO_Init(GPIOB, GPIO_PIN_4, GPIO_MODE_IN_FL_NO_IT);

    time_0_5s_upate_flag = 0;
    led_cnt = 0;
    beep_turn_on_flag = 0;

    // 初始化并使能time4
    Init_Timer4();
    // 初始化Beep
    init_beep();

    delay(100);

    // 开中断
    enableInterrupts();

    /* Infinite loop */
    while (1)
    {
        GPIO_WriteLow(GPIOA, GPIO_PIN_1); // turn on led
        delay_0_5s();
        GPIO_WriteHigh(GPIOA, GPIO_PIN_1); // turn off led
        delay_10s();

        if(0 == GPIO_ReadInputPin(GPIOB, GPIO_PIN_4))  // PB4拉低 --> 按键按
        {
            BEEP_Cmd(ENABLE);       // 启动蜂鸣器
            delay_0_5s();
            BEEP_Cmd(DISABLE);      // 关闭蜂鸣器
        }
    }
} « end main »

#ifdef USE_FULL_ASSERT
```

看设计要求和代码流程，似乎是这么样，如果把程序编译烧写到开发板上测试就会发现：经常出现按下了按键，蜂鸣器却没有鸣叫；而蜂鸣器鸣叫的那个循环，发现 led 的一个周期不止 10.5s。
再来看看程序，为什么会这样？
我们从 while 循环的开头开始看，mcu 会先控制 led 亮，然后做 0.5s 的无用功，然后关 led，然后做 10s 无用功，然后去检测是否有按键按下，如果按下了，启动蜂鸣器，在做 0.5s 无用功，然后关闭蜂鸣器，再循环回去。
实际上我们不可能保证我们按下按键的瞬间正好程序执行到检测按键的代码处，而在检测按键之外的循环中，有 10.5s 程序在做无用功，我们不可能按着按键 10.5s 以确保等程序来检测按键吧；接着分析，一旦检测到了按键，循环时间就是 0.5s+10s+0.5s = 11s，所以那个周期对于 led 来说就不满足需要“Led 每隔 10s 点亮 0.5s”。

从上面一个简单的例子就可以明白：
类似上面的空指令实现的延时，我们要尽可能的少用，能不用就不用。

有同学会发现，笔者在 while()循环之前，经常也会有一个 delay 延时，对于这个延时，其实就无所谓了，用空指令延时也好，用其他你能想到的任何方式实现的延时也好，因为这个只是程序启动初始化时才会有的延时，而 mcu 的主要工作都是在死循环里面。
有同学又会说，那初始化的延时不要可不可以？理论上是可以的，但是一般我们会写一个短延时。因为有的 mcu 或者接在 mcu 的外设上的模块会需要一点时间来稳定状态，当然这个一般都是根据实际的情况来调整初始化流程，不可以一概而论。

下面笔者提供一个不会出现上面空指令延时导致的两个问题的示例：

```
INTERRUPT_HANDLER(TIM4_UPD_OVF_IRQHandler, 23)
{
 /* In order to detect unexpected events during development,
    it is recommended to set a breakpoint on the following instruction.
 */
   static u16 cnt = 0;   // 局部静态变量，函数退出空间不回收

   cnt++;
   if(cnt >= 5000) // 0.1ms *5000 = 0.5s
   {
       cnt = 0;
       //GPIO_WriteReverse(GPIOA, GPIO_PIN_1); // PA1 输出电平翻转
       time_0_5s_upate_flag = 1;
   }

   TIM4_ClearITPendingBit(TIM4_IT_UPDATE);   //  清除中断标志位
}
```

```c
/**
* author : tianyx
* email  : zzztyx55@sina.com
* qq     : 609421258
* github : https://github.com/zzztyx55
*/
void main(void)
{
    // 关中断
    disableInterrupts();
    // 系统时钟配置
    clk_config();

    GPIO_DeInit(GPIOA);
    GPIO_DeInit(GPIOB);
    GPIO_DeInit(GPIOC);
    GPIO_DeInit(GPIOD);

    // init: PA1 - led, output high
    GPIO_Init(GPIOA, GPIO_PIN_1, GPIO_MODE_OUT_PP_HIGH_SLOW);
    GPIO_WriteHigh(GPIOA, GPIO_PIN_1); // turn off led
    // PB4, key input
    GPIO_Init(GPIOB, GPIO_PIN_4, GPIO_MODE_IN_FL_NO_IT);

    time_0_5s_upate_flag = 0;
    led_cnt = 0;
    beep_turn_on_flag = 0;

    // 初始化并使能time4
    Init_Timer4();
    // 初始化Beep
    init_beep();

    delay(100);

    // 开中断
    enableInterrupts();


    /* Infinite loop */
    while (1)
    {
        if(time_0_5s_upate_flag)  // 定时器0.5s计时标志
        {
            time_0_5s_upate_flag = 0; // clear time flag

            led_cnt += 1;
            if(led_cnt == 20)  // 10s = 0.5s*20
            {
                GPIO_WriteLow(GPIOA, GPIO_PIN_1); // turn on led
            }
            else if(led_cnt == 21)  // 10.5s = 0.5s*21，即在10.5s时刻关闭led
            {
                GPIO_WriteHigh(GPIOA, GPIO_PIN_1); // turn off led
                led_cnt = 0; // clear cnt
            }

            if(beep_turn_on_flag == 1)  // 蜂鸣器鸣叫标志被置1
            {
                beep_turn_on_flag = 2;  // 标志置2
                BEEP_Cmd(ENABLE);       // 启动蜂鸣器
            }
            else if(beep_turn_on_flag == 2) // 蜂鸣器鸣叫标志被置2
            {
                beep_turn_on_flag = 0;  // clear
                BEEP_Cmd(DISABLE);      // 关闭蜂鸣器
            }

        } « end if time_0_5s_upate_flag »

        if(0 == GPIO_ReadInputPin(GPIOB, GPIO_PIN_4))  // PB4拉低 --> 按键按下
            beep_turn_on_flag = 1;      // 蜂鸣器鸣叫标志置1
    } « end while 1 »
} « end main »

#ifdef USE_FULL_ASSERT

/**
  * @brief  Reports the name of the source file and the source line number
```

在定时器的中断服务函数中，每次计时到0.5s就将变量置1：

```
time_0_5s_upate_flag = 1;
```

在while循环中检查这个变量是否置1，置1表示0.5s，进去判断是否满足led的时刻，满足就点亮或者关闭led，判断是否需要启动或者关闭蜂鸣器并做响应的动作；与检查变量置1并行的是检查是否有按键按下。

在这里所有的检查，动作都是能够快速执行完毕的，如果某个时刻正在执行点亮led，突然按下了按键，由于按键动作都是至少ms级，所以等按键松开之前，程序已经执行到检测按键处了。

这里的示例只是为了说明空指令延时不好，对于按键检测，实际上是不能这么简单处理的，后面会讲到按键检测。