一、STM32 中断实验

1. 实验目的

1) 掌握使用中断相关配置流程，结合数据手册，理解寄存器含义和库函数版本代码设计逻辑`
2) 了解启动文件涉及到的中断服务函数和 MCU 启动流程。

2. 实验内容

1) 以外部中断为例对中断的配置过程、中断服务函数和嵌套向量中断控制器（NVIC）的功能：中断优先级分组、中断优先级的配置、读中断请求标志、清除中断请求标志、使能中断、失能中断等做详细分析。
2) 外部中断：配置为外部中断首先需要开启 AFIO 时钟，配置 AFIO_EXTICRx 寄存器来选择 EXTIx 外部中断的输入源。对 EXTI_InitTypeDef 结构体所定义的中断线（EXTI_Line）、中断/事件（EXTI_Mode）、触发方式（EXTI_Trigger）和中断线使能（EXTI_LineCmd）变量进行赋值。对 NVIC_InitTypeDef 结构体所定义的 IRQ 通道（NVIC_IRQChannel）、抢占优先级（NVIC_IRQChannelPreemptionPriority）、子优先级（NVIC_IRQChannelSubPriority）和通道使能（NVIC_IRQChannelCmd）变量进行赋值。
3) 对中断服务函数的定义和调用，尤其是对中断线源 5-9、10-15 两组中断服务函数具体区分哪一个中断线源发生中断进行分析。

3. 程序框图

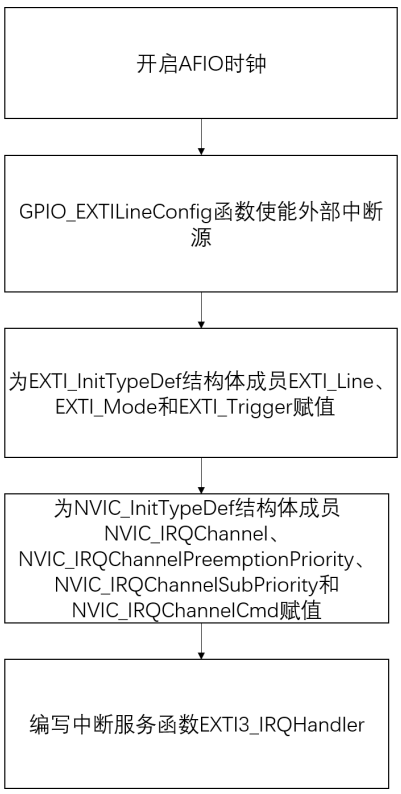

图 1 外部中断主要配置流程图

4. 主要程序

1) 外部中断以板载按键 KEY1（端口为 GPIOE.3）为例，配置为下降沿触发。主要配置代码如下。

```
1. RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE);
2. GPIO_EXTILineConfig(GPIO_PortSourceGPIOE,GPIO_PinSource3);
3. EXTI_InitStructure.EXTI_Line=EXTI_Line3;
4. EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
5. EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
6. EXTI_Init(&EXTI_InitStructure);
```

RCC_APB2PeriphClockCmd 函数在之前的报告中有过解析，此处不在赘述，其含义为开启 APB2 总线上的 AFIO 时钟，为配置 AFIO_EXTICRx 寄存器做准备。

- GPIO_EXTILineConfig 函数为配置 AFIO_EXTICRx 寄存器选择 EXTIx 外部中断的输入源。

```
1. void GPIO_EXTILineConfig(uint8_tGPIO_PortSource,uint8_tGPIO_PinSource)
    {
2.  uint32_t tmp = 0x00;
3.  assert_param(IS_GPIO_EXTI_PORT_SOURCE(GPIO_PortSource));
4.  assert_param(IS_GPIO_PIN_SOURCE(GPIO_PinSource));
5.  tmp = ((uint32_t)0x0F) << (0x04 * (GPIO_PinSource & (uint8_t)0x03));
6.  AFIO->EXTICR[GPIO_PinSource >> 0x02] &= ~tmp;
7.  AFIO->EXTICR[GPIO_PinSource >> 0x02] |= (((uint32_t)GPIO_PortSource)
   << (0x04 * (GPIO_PinSource & (uint8_t)0x03)));
8. }
```

由数据手册可知，EXTICR[1-4]4 个寄存器分别配置 0-3、4-7、8-11、12-15 四组外部输入源，同时每个寄存器低 16 位中每 4 位一组，可选 A-G 共 7 个端口。

EXTICR[GPIO_PinSource >> 0x02]用于区分中断线源所处的具体 EXTICRx 寄存器，同时 tmp 值用于确定该中断线源在当前 EXTICRx 寄存器低 16 位中具体的 4 位，首先清除 EXTICRx 寄存器相应位，再将相应的引脚和端口置位，选择相应的 A-G 端口原理同上。

- 对 EXTI_InitStructure 结构体变量的赋值，通过调用 EXTI_Init 函数并经过程序的解析最终将对相应的寄存器进行赋值。（该部分解析穿插在代码中）

```
1. void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct)
2. { //首先对输入参数形式进行校验
3.  uint32_t tmp = 0;
```

```
4.    assert_param(IS_EXTI_MODE(EXTI_InitStruct->EXTI_Mode));
5.    assert_param(IS_EXTI_TRIGGER(EXTI_InitStruct->EXTI_Trigger));
6.    assert_param(IS_EXTI_LINE(EXTI_InitStruct->EXTI_Line));
7.    assert_param(IS_FUNCTIONAL_STATE(EXTI_InitStruct->EXTI_LineCmd));
8.    tmp = (uint32_t)EXTI_BASE;
9.    if (EXTI_InitStruct->EXTI_LineCmd != DISABLE)
10.    //如果 EXTI_LineCmd 参数有效
11.   { //对 IMR（中断）和 EMR（事件）寄存器相关位进行清除
12.     EXTI->IMR &= ~EXTI_InitStruct->EXTI_Line;
13.     EXTI->EMR &= ~EXTI_InitStruct->EXTI_Line;
14.     tmp += EXTI_InitStruct->EXTI_Mode; //将 EXTIMode_TypeDef 结构体定
    义的枚举变量与基地址 EXTI_BASE 相加，得到相应的 IMR 或 EMR 寄存器的偏移
    地址
15.     *(__IO uint32_t *) tmp |= EXTI_InitStruct->EXTI_Line; //通过指针操作，对
    IMR 或 EMR 寄存器的相应为置 1，开放中断或事件请求。
16.    //将对 RTSR（上升沿触发）和 FTSR（下降沿触发）寄存器相关位进行清除
17.     EXTI->RTSR &= ~EXTI_InitStruct->EXTI_Line;
18.     EXTI->FTSR &= ~EXTI_InitStruct->EXTI_Line;
19.     if (EXTI_InitStruct->EXTI_Trigger == EXTI_Trigger_Rising_Falling)
20.     {
21. //如果使能了上升沿和下降沿触发，则同时将 RTSR 和 FTSR 寄存器相应为置 1
22.       EXTI->RTSR |= EXTI_InitStruct->EXTI_Line;
23.       EXTI->FTSR |= EXTI_InitStruct->EXTI_Line;
24.     }
25.     else  //如果只是选择了上升沿或者下降沿触发的其中一个
26.     {
27.       tmp = (uint32_t)EXTI_BASE;
28.       tmp += EXTI_InitStruct->EXTI_Trigger;
29.       *(__IO uint32_t *) tmp |= EXTI_InitStruct->EXTI_Line;
30.      //同上述对 IMR 或 EMR 寄存器赋值原理，在基地址的基础上加上 RTSR 或
    FTSR 寄存器的偏移量并对相应位赋值。
31.     }
32.   }
33.   else  //如果失能外部中断，则对相应的 IMR（中断）和 EMR（事件）寄存器相
    关位写 0，屏蔽中断请求。
34.   {
35.     tmp += EXTI_InitStruct->EXTI_Mode;
36.     *(__IO uint32_t *) tmp &= ~EXTI_InitStruct->EXTI_Line;
```

```
37.  }
```

嵌套向量中断控制器（NVIC）在中断控制中起着关键的作用，是 Cortex-M3 核心的一部分，该部分代码配置包括对中断通道的使能（与上述外部中断线源使能有区别），对每一个中断设置优先级，定义当多个不同中断同时发生时 CPU 处理的中断的规则。

```
1.  NVIC_InitStructure.NVIC_IRQChannel = EXTI3_IRQn;
2.  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;
3.  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x01;
4.  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
5.  NVIC_Init(&NVIC_InitStructure);
```

- 对于中断优先级的设置，在配置优先级大小之前，需先进行优先级分组（分为抢占优先级和子优先级）调用 NVIC_PriorityGroupConfig 函数，对应用中断和复位控制寄存器（SCB_AIRCR [10：8]）赋值。（对 SCB_AIRCR [10：8]赋值的同时需保持 SCB_AIRCR [31：16]为 0x05FA，否则写入会被忽略——PM0056 Programming manual）

```
1.  void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)
2.  {
3.    assert_param(IS_NVIC_PRIORITY_GROUP(NVIC_PriorityGroup));
4.    /* Set the PRIGROUP[10:8] bits according to NVIC_PriorityGroup value */
5.    SCB->AIRCR = AIRCR_VECTKEY_MASK | NVIC_PriorityGroup;
6.  }
```

在确定了优先级分组后，NVIC_IPRx 寄存器对应的每组可屏蔽中断配置位的高 4 位对优先级的分配结果如下：

组	AIRCR[10：8]	bit[7：4]分配情况	分配结果
0	111	0： 4	0 位抢占优先级， 4 位响应优先级
1	110	1： 3	1 位抢占优先级， 3 位响应优先级
2	101	2： 2	2 位抢占优先级， 2 位响应优先级
3	100	3： 1	3 位抢占优先级， 1 位响应优先级
4	011	4： 0	4 位抢占优先级， 0 位响应优先级

抢占优先级的级别大于子优先级，数值越小代表的优先级越高。

- NVIC_Init 函数解析（该部分解析穿插在代码中）

```
1.  void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
2.  {   //检测输入参数的正确性
3.   uint32_t tmppriority = 0x00, tmppre = 0x00, tmpsub = 0x0F;
4.   assert_param(IS_FUNCTIONAL_STATE(NVIC_InitStruct->NVIC_IRQChannel
    Cmd));
5.   assert_param(IS_NVIC_PREEMPTION_PRIORITY(NVIC_InitStruct->NVIC_IR
    QChannelPreemptionPriority));
6.   assert_param(IS_NVIC_SUB_PRIORITY(NVIC_InitStruct->NVIC_IRQChannel
    SubPriority));
7.    if (NVIC_InitStruct->NVIC_IRQChannelCmd != DISABLE)
8.   {
9.      tmppriority = (0x700 - ((SCB->AIRCR) & (uint32_t)0x700))>> 0x08;
10.    tmppre = (0x4 - tmppriority);  //判断抢占优先级位数
11.    tmpsub = tmpsub >> tmppriority;  //判断子优先级位数
12.    tmppriority = (uint32_t)NVIC_InitStruct->NVIC_IRQChannelPreemptionPr
    iority << tmppre; //配置对应通道 8 位高 4 位中的抢占优先级
13.   tmppriority |=  NVIC_InitStruct->NVIC_IRQChannelSubPriority & tmpsub;
      //配置对应通道 8 位高 4 位中的子优先级
14.  tmppriority = tmppriority << 0x04; //将抢占优先级和子优先级提升至对应通
    道 8 位的高 4 位
15.    NVIC->IP[NVIC_InitStruct->NVIC_IRQChannel] = tmppriority;
16.   //将优先级的配置写入 NVIC_IP 寄存器
17.    NVIC->ISER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
18.     (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
      }   //配置 NVIC_ISER 寄存器，使能相应 IRQ 通道
19.  else
20.  {
21.    NVIC->ICER[NVIC_InitStruct->NVIC_IRQChannel >> 0x05] =
22.     (uint32_t)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (uint8_t)0x1F);
      }   //NVIC_ICER 为中断失能寄存器，对相应为置 1，则所选择的通道失能。
23. }
```

该段代码配置主要包括对中断优先级的设置，和对相应中断通道的使能。是中断配置过程中较为重要的一部分。

中断服务函数

```
1.  void EXTI3_IRQHandler(void)
2.  {
```

```
    delay_ms(10);
    if(KEY1==0)
        LED1=!LED1;
    EXTI_ClearITPendingBit(EXTI_Line3);
}
```

该段代码为中断服务函数，当所定义的中断发生时，内核会将当前状态保存，立即执行中断服务函数内容，并在其执行完以后自动返回之前所保存的状态，继续执行。

EXTI_ClearITPendingBit 函数的意义为对 EXTI_PR 寄存器的相应位写 1，清除中断标志位，否则将持续进入该中断。针对 NVIC 的寄存器组 IABR 同样表示中断激活标志位，但其为只读寄存器，若读取到某一位为 1，则表示该位所对应的中断正在被执行，在中断执行完之后由硬件自动清零。

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line)
{
  assert_param(IS_EXTI_LINE(EXTI_Line));
  EXTI->PR = EXTI_Line;
}
```

值得注意的是，针对外部中断，响应外部中断的服务函数将中断线源 5-9 和中断线源 10-15 分为两组，分别由中断服务函数 EXTI9_5_IRQHandler 和 EXTI15_10_IRQHandler 响应。当 EXTI9_5_IRQHandler 中断函数执行时，可在内部执行 EXTI_GetFlagStatus 或 EXTI_GetITStatus 函数用于确定具体的中断源。但是两个函数具有细微的差别。

```
FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line)
{
  FlagStatus bitstatus = RESET;
  assert_param(IS_GET_EXTI_LINE(EXTI_Line));
  if ((EXTI->PR & EXTI_Line) != (uint32_t)RESET)
    bitstatus = SET;
  else
    bitstatus = RESET;
  return bitstatus;
}
```

```
ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)
{
  ITStatus bitstatus = RESET;
```

```
4.    uint32_t enablestatus = 0;
5.    assert_param(IS_GET_EXTI_LINE(EXTI_Line));
6.    enablestatus =  EXTI->IMR & EXTI_Line;
7.    if (((EXTI->PR & EXTI_Line) != (uint32_t)RESET) && (enablestatus != (uint3
      2_t)RESET))
8.      bitstatus = SET;
9.    else
10.     bitstatus = RESET;
11.   return bitstatus;
12. }
```

通过分析可知，EXTI_GetFlagStatus 函数只是纯粹读取中断标志位的状态，而 EXTI_GetITStatu 除了读取中断标志位，还查看 EXT_IMR 寄存器是否对该中断进行屏蔽，在中断挂起且没有屏蔽的情况下就会响应中断。当中断线源 10-15 发生中断时，区分中断源原理同上。

各中断服务函数的定义均在启动文件 startup_stm32f10x_hd.s 定义。中断向量表在系统启动初期由硬件进行设置地址。如下图所示（只截取了部分）：此处内容和启动过程紧密相连。

```
__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
```

图 2 中断向量表

Cortex-M3 内核规定中断向量表中第一个 32 位数据内容为栈顶地址，第二个 32 位数据内容则是复位中断向量的入口地址。以 Flash 启动模式为例（起始地址为 0x08000000），首先对堆栈进行开辟，将 SP 指向 0x20000500，之后 CPU 自动从第二个 32 位数据（0x08000004）中取出复位中断向量的入口地址(0x08000851，PC寄存器LSB为1表示使用使用Thumb-2指令集，LSB为0时表示试图进入ARM指令集模式，但是M3内核不支持ARM指令集，则会产生错误中断。)，PC 就跳转到中断服务程序 Reset_Handler（此处硬件自动将地址对齐至 0x08000850）。 Reset_Handler 函数对系统时钟初始化后跳转到用户 main 函数。

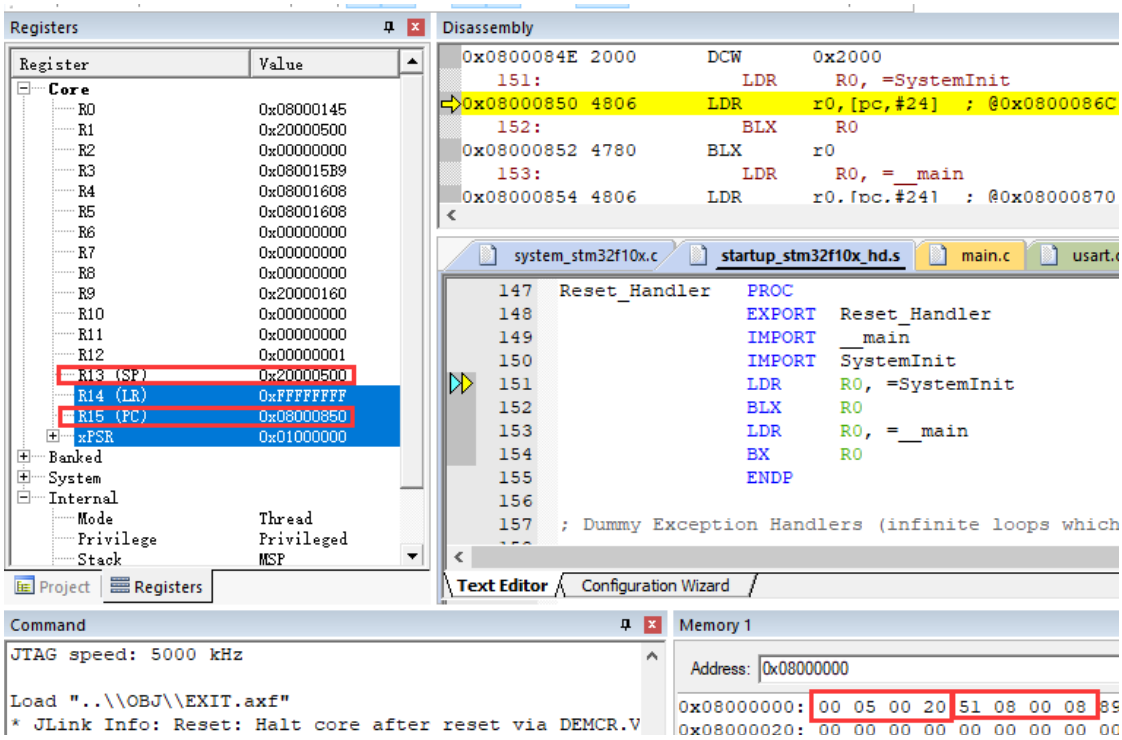

图 3 memory 值

启动文件中关于中断服务函数的定义：

```
1. EXTI3_IRQHandler PROC
2.         EXPORT  EXTI3_IRQHandler        [WEAK]
3.         B       .
4.         ENDP
```

该段代码表示当在其余文件定义了 EXTI3_IRQHandler 函数，中断发生时则执行用户所定义的中断函数，若没有定义，则进入死循环。

二、SEP8000 时钟门控测试

1. 实验目的

1) 验证 PMU 模块在设计范围内对其余各模块时钟门控功能正常与否；

2) 结合 PMU 数据手册，熟悉 PMU 功能。

2. 实验内容

PMU 为时钟与功耗管理模块单元，主要控制各个模块的时钟，当某一模块不在工作状态时可以选择关闭其时钟，以达到降低部分功耗的要求。

根据数据手册 CLKGTCFG 寄存器的描述，复位后该寄存器各位为 1，各模块时钟均打开，可由图 4 可知，在未配置 CLKGTCFG 寄存器之前，各个模块时钟信号均有波形。通过对 CLKGTCFG 寄存器的有效位分别置 0（PMU_CLKGTCFG = ~(1<<4);根据数据手册依次对 PMU_CLKGTCF 值相应位赋 0），观察各模块时钟是否按照验证代码能够正常关闭和开启。

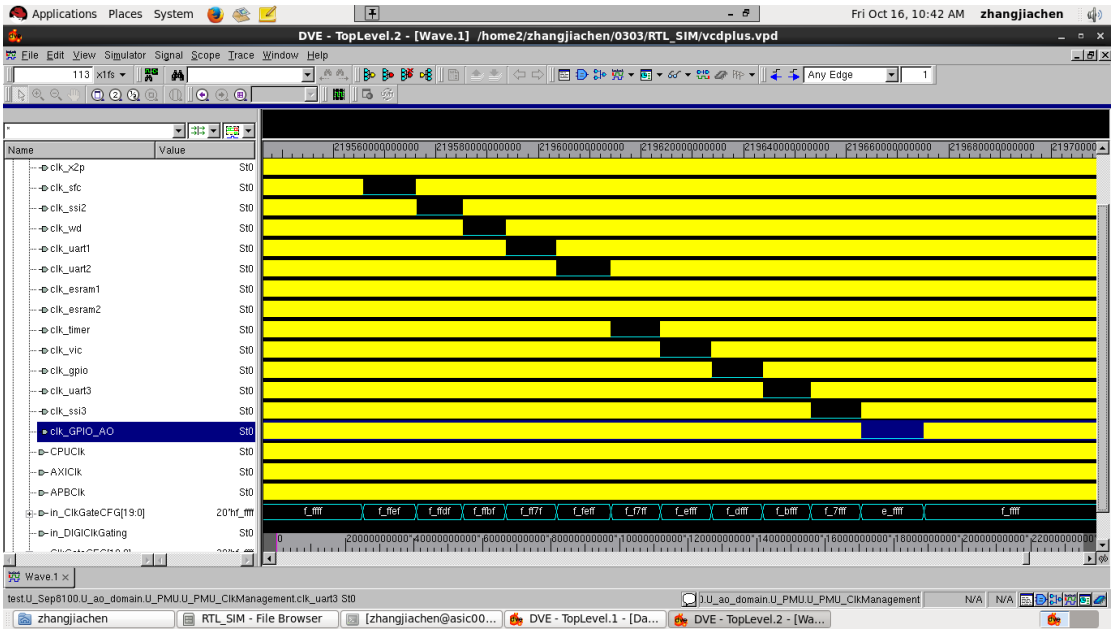

图 4 各模块时钟信号波形图

经验证当信号线 in_ClkGateCFG[19:0]为 f_ffef 时，clk_sfc 信号线为 0；in_ClkGateCFG[19:0]为 f_ffdf 时，clk_ssi2 信号线为 0；in_ClkGateCFG[19:0]为 f_ffbf 时，clk_wd 信号线为 0；in_ClkGateCFG[19:0]为 f_ff7f 时，clk_uart1 信号线为 0；in_ClkGateCFG[19:0]为 f_feff 时，clk_ uart1 信号线为 0；in_ClkGateCFG[19:0]为 f_f7ff 时，clk_ timer 信号线为 0；in_ClkGateCFG[19:0]为 f_efff 时，clk_ vic 信号线为 0；in_ClkGateCFG[19:0]为 f_dfff 时，clk_ gpio 信号线为 0；in_ClkGateCFG[19:0]为 f_bfff 时，clk_ uart3 信号线为 0；in_ClkGateCFG[19:0]为 f_7fff 时，clk_ ssi3 信号线为 0；in_ClkGateCFG[19:0]为 e_ffff 时，clk_ GPIO_AO 信号线为 0；

各个时钟的门控信号均有效，在为有效位写 0 后，相应的模块时钟即可关闭，写 1 后模块时钟开启。根据 PMU_ClkManagement 模块 RTL 代码可知 clk_esram1 和 clk_esram2 时钟不受 in_ClkGateCFG[19:0]影响，始终为开启状态。