【教程】使用 STM32 测量频率和占空比的几种方法（申请置酷!）

——倾夜•陨灭星尘

这几天在论坛上面解答了好几个询问 STM32 测量频率的贴子，觉得这种需求还是存在的（示波器、电机控制等等）。而简单搜索了一下论坛，这方面的贴子有但是不全。正好今年参加比赛做过这方面的题目（最后是一等奖嘿嘿），所以把我们当时尝试过的各种方案都列出来，方便以后大家使用，也是作为一个长期在论坛的潜水党对论坛的回报。

PS:由于我们当时的题目除了测量频率之外，更麻烦的是测量占空比。而这两个的测量方法联系比较紧密，所以也一并把测量占空比的方法写出来。因为时间有限，所以并不能把所有思路都一一测试，只是写在下面作为参考，敬请谅解。

使用平台：官方 STM32F429DISCOVERY 开发板，180MHz 的主频，定时器频率 90MHz。

相关题目：

（1）测量脉冲信号频率 f_O，频率范围为 10Hz～2MHz，测量误差的绝对值不大于 0.1%。（15 分）

（2）测量脉冲信号占空比 D，测量范围为 10％～90％，测量误差的绝对值不大于 2%。（15 分）

思路一：外部中断

思路：这种方法是很容易想到的，而且对几乎所有 MCU 都适用（连 51 都可以）。方法也很简单，声明一个计数变量 TIM_cnt，每次一个上升沿/下降沿就进入一次中断，对 TIM_cnt++，然后定时统计即可。如果需要占空比，那么就另外用一个定时器统计上升沿、下降沿之间的时间即可。

缺点：缺陷显而易见，当频率提高，将会频繁进入中断，占用大量时间。而当频率超过 100kHz 时，中断程序时间甚至将超过脉冲周期，产生巨大误差。同时更重要的是，想要测量的占空比由于受到中断程序影响，误差将越来越大。

总结:我们当时第一时间就把这个方案 PASS 了，没有相关代码（这个代码也很简单）。不过，该方法在频率较低（10K 以下）时，可以拿来测量频率。在频率更低的情况下，可以拿来测占空比。

思路二：PWM 输入模式

思路：翻遍 ST 的参考手册，在定时器当中有这样一种模式：

15.3.6 PWM 输入模式

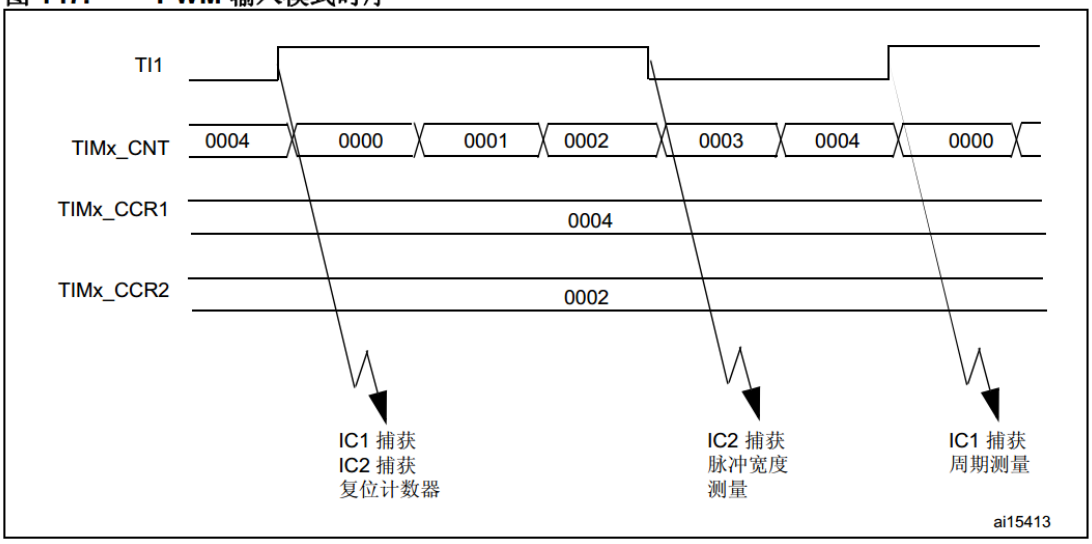
此模式是输入捕获模式的一个特例。其实现步骤与输入捕获模式基本相同，仅存在以下不同之处：

- 两个 ICx 信号被映射至同一个 TIx 输入。
- 这两个 ICx 信号在边沿处有效，但极性相反。
- 选择两个 TIxFP 信号之一作为触发输入，并将从模式控制器配置为复位模式。

例如，可通过以下步骤对应用于 TI1 的 PWM 的周期（位于 TIMx_CCR1 寄存器中）和占空比（位于 TIMx_CCR2 寄存器中）进行测量（取决于 CK_INT 频率和预分频器的值）：

- 选择 TIMx_CCR1 的有效输入：向 TIMx_CCMR1 寄存器中的 CC1S 位写入 01（选择 TI1）。
- 选择 TI1FP1 的有效极性（用于 TIMx_CCR1 中的捕获和计数器清零）：向 CC1P 位和 CC1NP 位写入“0”（上升沿有效）。
- 选择 TIMx_CCR2 的有效输入：向 TIMx_CCMR1 寄存器中的 CC2S 写入 10（选择 TI1）。
- 选择 TI1FP2 的有效极性（用于 TIMx_CCR2 中的捕获）：向 CC2P 位和 CC2NP 位写入“1”（下降沿有效）。
- 选择有效触发输入：向 TIMx_SMCR 寄存器中的 TS 位写入 101（选择 TI1FP1）。
- 将从模式控制器配置为复位模式：向 TIMx_SMCR 寄存器中的 SMS 位写入 100。
- 使能捕获：向 TIMx_CCER 寄存器中的 CC1E 位和 CC2E 位写入“1”。

图 147. PWM 输入模式时序



简而言之，理论上，通过这种模式，可以用硬件直接测量出频率和占空比。当时我们发现这一模式时欢欣鼓舞，以为可以一步解决这一问题，代码如下：

初始化：

```
void Tim2_PWMIC_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
  NVIC_InitTypeDef NVIC_InitStructure;
    TIM_ICInitTypeDef    TIM_ICInitStructure;
  /* TIM4 clock enable */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);

  /* GPIOB clock enable */
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
```

```
  /* TIM4 chennel2 configuration : PB.07 */
  GPIO_InitStructure.GPIO_Pin      = GPIO_Pin_7;
  GPIO_InitStructure.GPIO_Mode     = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd     = GPIO_PuPd_UP ;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

  /* Connect TIM pin to AF2 */
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource7, GPIO_AF_TIM4);

  /* Enable the TIM4 global Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

     TIM_ICInitStructure.TIM_Channel = TIM_Channel_2;
  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;
  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
  TIM_ICInitStructure.TIM_ICFilter = 0x0;

  TIM_PWMIConfig(TIM4, &TIM_ICInitStructure);

  /* Select the TIM4 Input Trigger: TI2FP2 */
  TIM_SelectInputTrigger(TIM4, TIM_TS_TI2FP2);

  /* Select the slave Mode: Reset Mode */
  TIM_SelectSlaveMode(TIM4, TIM_SlaveMode_Reset);
  TIM_SelectMasterSlaveMode(TIM4,TIM_MasterSlaveMode_Enable);

  /* TIM enable counter */
  TIM_Cmd(TIM4, ENABLE);

  /* Enable the CC2 Interrupt Request */
    TIM_ITConfig(TIM4, TIM_IT_CC2, ENABLE);

}
```

中断程序：

```
void TIM4_IRQHandler(void)
{
```

```
    /* Clear TIM4 Capture compare interrupt pending bit */
    TIM_ClearITPendingBit(TIM4, TIM_IT_CC1|TIM_IT_CC2);

    /* Get the Input Capture value */
    IC2Value = TIM_GetCapture2(TIM4);//周期

    if (IC2Value != 0)
    {
            highval[filter_cnt]=TIM_GetCapture1(TIM4);//高电平周期
            waveval[filter_cnt]=IC2Value;
            filter_cnt++;
            if(filter_cnt>=FILTER_NUM)
                filter_cnt=0;
    }
    else
    {
       DutyCycle = 0;
       Frequency = 0;
    }
}
```

主循环：

```
    while (1)
    {
            uint32_t highsum=0,wavesum=0,dutysum=0,freqsum=0;
            LCD_Clear(0);
            for(i=0;i<FILTER_NUM;i++)
            {
                highsum+=highval[i];
                wavesum+=waveval[i];
            }
            delay_ms(1);
            DutyCycle=highsum*1000/wavesum;
            Frequency=(SystemCoreClock/2*1000/wavesum);
            freq=Frequency*2.2118-47.05;//线性补偿
            sprintf(str,"DUTY:%3d\nFREQ:%.3f KHZ\n",DutyCycle,freq/1000);

            LCD_ShowString(0,200,str);
            delay_ms(100);
    }
```

但是，经过测量之后发现这种方法测试数据不稳定也不精确，数据不停跳动，且和实际值相差很大。ST 的这些功能经常有这种问题，比如定时器的编码器模式，在 0 点处频繁正负跳变时有可能会卡死。这些方法虽然省事，稳定性却不是很好。

经过线性补偿可以一定程度上减少误差（参数在不同情况下不同）：

```
freq=Frequency*2.2118-47.05;
```

这种方法无法实现要求。所以在这里我并不推荐这种方法。如果有谁能够有较好的程序，也欢迎发出来。

思路三：输入捕获

思路：一般来说，对 STM32 有一定了解的坛友们在测量频率的问题上往往都会想到利用输入捕获。首先设定为上升沿触发，当进入中断之后（rising）记录与上次中断（rising_last）之间的间隔（周期，其倒数就是频率）。再设定为下降沿，进入中断之后与上升沿时刻之差即为高电平时间(falling-rising_last)，高电平时间除周期即为占空比。

画图如下（由于修改多次使得）：

程序如下，注意由于为了减少程序复杂性使用了 32 位定时器 5（计数周期如果是 1us 时可以计数 4294s，否则如果是 16 位只能计数 65ms），如果需要在 F1 上使用则需要自行处理：

```
//定时器 5 通道 1 输入捕获配置
//arr：自动重装值(TIM2,TIM5 是 32 位的!!)
//psc：时钟预分频数
void TIM5_CH1_Cap_Init(u32 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;


    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,ENABLE);    //TIM5 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);       //使能  PORTA  时
钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0; //GPIOA0
    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_DOWN; //下拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA0

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource0,GPIO_AF_TIM5); //PA0 复用位定时器 5


    TIM_TimeBaseStructure.TIM_Prescaler=psc;   //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=arr;    //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;

    TIM_TimeBaseInit(TIM5,&TIM_TimeBaseStructure);
```

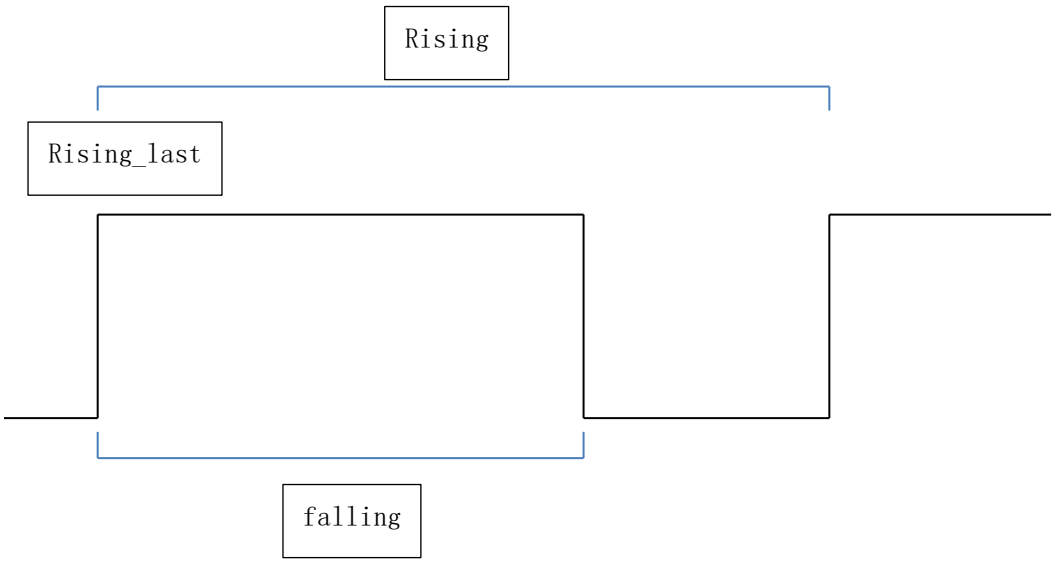
```
    //初始化 TIM5 输入捕获参数
    TIM5_ICInitStructure.TIM_Channel = TIM_Channel_1; //CC1S=01    选择输入端  IC1 映射到 TI1 上
    TIM5_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising; //上升沿捕获
    TIM5_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到 TI1 上
    TIM5_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;      //配置输入分频,不分频
    TIM5_ICInitStructure.TIM_ICFilter = 0x00;//IC1F=0000 配置输入滤波器 不滤波
    TIM_ICInit(TIM5, &TIM5_ICInitStructure);

    TIM_ITConfig(TIM5,TIM_IT_Update|TIM_IT_CC1,ENABLE);// 允许更新中断 ,允许 CC1IE 捕获中断

    TIM_Cmd(TIM5,ENABLE );      //使能定时器 5


    NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn;
      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=2;//抢占优先级
      NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;       //子优先级
      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;          //IRQ 通道使能
      NVIC_Init(&NVIC_InitStructure);  //根据指定的参数初始化 VIC 寄存器、


}
//捕获状态(对于 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
//定时器 5 中断服务程序
void TIM5_IRQHandler(void)
{

    if(TIM_GetITStatus(TIM5, TIM_IT_CC1) != RESET)//捕获 1 发生捕获事件
    {
        if(edge==RESET)//上升沿
        {
            rising=TIM5->CCR1-rising_last;
            rising_last=TIM5->CCR1;
            TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Falling); //CC1P=0 设置为上升沿捕获
            edge=SET;
        }
        else
        {
            falling=TIM5->CCR1-rising_last;
            TIM_OC1PolarityConfig(TIM5,TIM_ICPolarity_Rising); //CC1P=0 设置为上升沿捕获
            edge=RESET;
```

```
            }
        }
        TIM_ClearITPendingBit(TIM5, TIM_IT_CC1|TIM_IT_Update); //清除中断标志位
    }
```

主程序：

```
    while (1)
    {
            uint32_t highsum=0,wavesum=0,dutysum=0,freqsum=0;
            LCD_Clear(0);

            delay_ms(1);

            sprintf(str,"rise:%3d\nfall:%d\nfall-rise:%d",rising,falling,falling-rising);
            LCD_ShowString(0,100,str);
            sprintf(str,"Freq:%.2f
Hz\nDuty:%.3f\n",90000000.0/rising,(float)falling/(float)rising);//频率、占空比
            LCD_ShowString(0,200,str);
            delay_ms(100);
    }
```

注意的是，中断程序当中的变量 rising,last 因为多次修改的缘故，与名称本身含义有所区别，示意如下：



该方法尤其是在中低频（<100kHz）之下精度不错。

缺点：稍有经验的朋友们应该都能看出来，该方法仍然会带来极高的中断频率。在高频之下，首先是 CPU 时间被完全占用，此外，更重要的是，中断程序时间过长往往导致会错过一次或多次中断信号，表现就是测量值在实际值、实际值×2、实际值×3 等之间跳动。实测中，最高频率可以测到约 400kHz。

总结：该方法在低频率（<100kHz）下有着很好的精度，在考虑到其它程序的情况下，建议在 10kHz 之下使用该方法。同时，可以参考以下的改进程序减少 CPU 负载。

改进：

前述问题，限制频率提高的主要因素是过长的中断时间（一般应用情景之下，还有其它程序部分的限制）。所以进行以下改进：

1. 使用 2 个通道，一个只测量上升沿，另一个只测量下降沿。这样可以减少切换触发边沿的延迟，缺点是多用了一个 IO 口。
2. 使用寄存器，简化程序

最终程序如下：

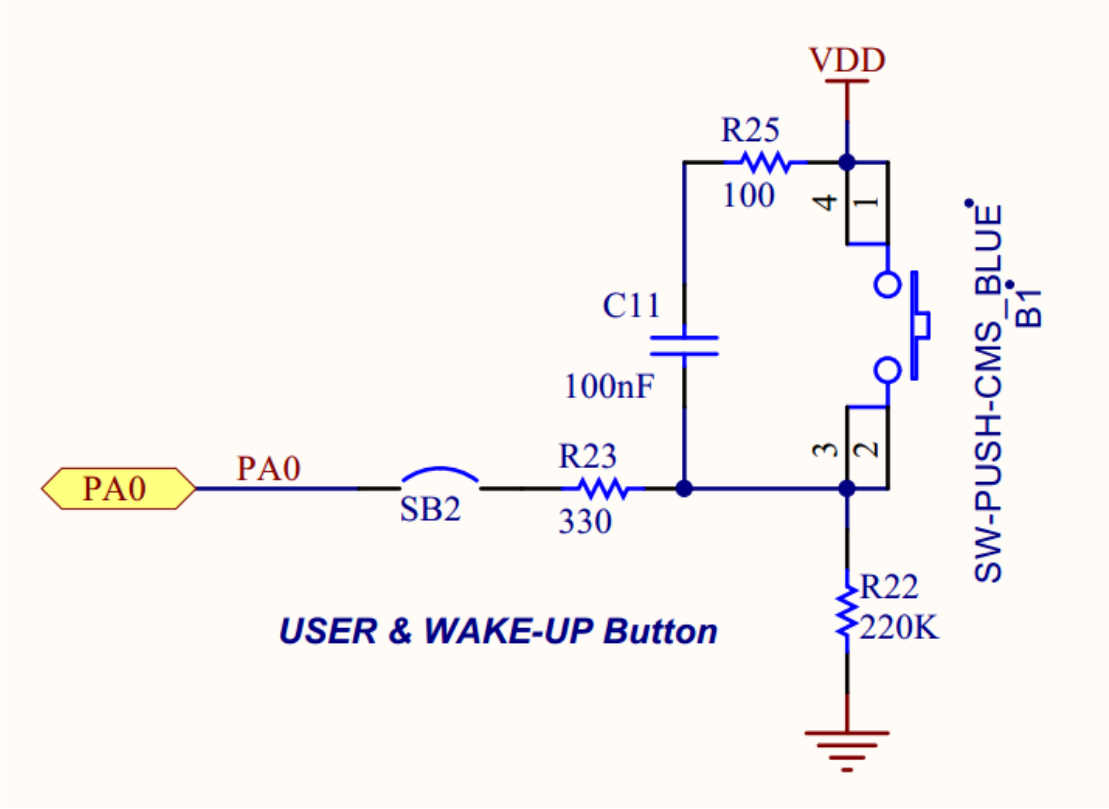
```
//TIM2_CH1->PA5
//TIM2_CH2->PB3
void TIM2_CH1_Cap_Init(u32 arr,u16 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    TIM_ICInitTypeDef    TIM_ICInitStructure;

    TIM_DeInit(TIM2);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2,ENABLE);    //TIM2 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOB,
ENABLE);       //使能 PORTA 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; //GPIOA0
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;    //速度 100MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN; //下拉
    GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA0

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3; //GPIOA0
    GPIO_Init(GPIOB,&GPIO_InitStructure); //初始化 PA0

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource5,GPIO_AF_TIM2); //PA0 复用位定时器 5
    GPIO_PinAFConfig(GPIOB,GPIO_PinSource3,GPIO_AF_TIM2); //PA0 复用位定时器 5

    TIM_TimeBaseStructure.TIM_Prescaler=psc;   //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=arr;    //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;

    TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);
    //初始化 TIM2 输入捕获参数
```

```
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1; //CC1S=01     选择输入端 IC1 映射到 TI1 上
  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;  //上升沿捕获
  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //映射到 TI1 上
  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;  //配置输入分频,不分频
  TIM_ICInitStructure.TIM_ICFilter = 0x00;//IC1F=0000 配置输入滤波器 不滤波
  TIM_ICInit(TIM2, &TIM_ICInitStructure);

    TIM_ICInitStructure.TIM_Channel = TIM_Channel_2; //CC1S=01     选择输入端 IC1 映射到 TI1 上
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Falling;    //上升沿捕获

    TIM_ICInit(TIM2, &TIM_ICInitStructure);

    TIM_ITConfig(TIM2,TIM_IT_Update|TIM_IT_CC1|TIM_IT_CC2,ENABLE);// 允许更新中断 ,允许 CC1IE 捕获中断
//    TIM2_CH1_Cap_DMAInit();
  TIM_Cmd(TIM2,ENABLE );      //使能定时器 5

  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;//抢占优先级 3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;       //子优先级 3
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //IRQ 通道使能
    NVIC_Init(&NVIC_InitStructure);  //根据指定的参数初始化 VIC 寄存器、

}
//定时器 2 中断服务程序(对于 32 位定时器来说,1us 计数器加 1,溢出时间:4294 秒)
void TIM2_IRQHandler(void)
{
    if(TIM2->SR&TIM_FLAG_CC1)//TIM_GetITStatus(TIM2, TIM_IT_CC1) != RESET)//捕获 1 发生捕获事件
    {
        rising=TIM2->CCR1-rising_last;
        rising_last=TIM2->CCR1;
        return;
    }
    if(TIM2->SR&TIM_FLAG_CC2)//TIM_GetITStatus(TIM2, TIM_IT_CC2) != RESET)
    {
        falling=TIM2->CCR2-rising_last;
        return;
    }
    TIM2->SR=0;
}
```

之所以改用 TIM2 是因为 TIM5 的 CH1(PA0)还是按键输入引脚。本来想来这应当也没什

么，按键不按下不就是开路嘛。但是后来发现官方开发板上还有一个 RC 滤波……

所以，当使用别人的程序之前，请一定仔细查看电路图。



这样，最高频率能够达到约 1.1MHz，是一个不小的进步。但是，其根本问题——中断太频繁——仍然存在。

解决思路也是存在的。本质上，我们实际上只需要读取 CCR1 和 CCR2 寄存器。而在内存复制过程中，面对大数据量的转移时，我们会想到什么？显然，我们很容易想到——利用 DMA。所以，我们使用输入捕获事件触发 DMA 来搬运寄存器而非触发中断即可，然后将这些数据存放在一个数组当中并循环刷新。这样，我们可以随时来查看数据并计算出频率。

这一方法我曾经尝试过，没有调出来，因为，有一个更好的方法存在。但是理论上这是没有问题的，以供参考我列出如下。

【注意：这段程序无法工作，仅供参考!!!】

```
//TIM2_CH1->DMA1_CHANNEL3_STREAM5
u32  val[FILTER_NUM]={0};
void TIM2_CH1_Cap_DMAInit(void)
{
    NVIC_InitTypeDef NVIC_InitStructure;
    DMA_InitTypeDef    DMA_InitStructure;

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1,ENABLE);//DMA1 时钟使能

  DMA_DeInit(DMA1_Stream5);

    while (DMA_GetCmdStatus(DMA1_Stream5) != DISABLE){}//等待 DMA 可配置
```

```
    /* 配置 DMA Stream */
    DMA_InitStructure.DMA_Channel = DMA_Channel_3;  //通道选择
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(TIM5->CCR1);//DMA 外设地
址
    DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)val;//DMA 存储器 0 地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;//存储器到外设模式
    DMA_InitStructure.DMA_BufferSize = FILTER_NUM;//数据传输量
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设非增量模式
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;//存储器增量模式
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;//外设数
据长度:8 位
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;//存储器数
据长度:8 位
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;// 使用普通模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;//中等优先级
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;//存储器突发单次
传输
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;//外设突发单次
传输
    DMA_Init(DMA1_Stream5, &DMA_InitStructure);//初始化 DMA Stream

      TIM_DMAConfig(TIM5,TIM_DMABase_CCR1,TIM_DMABurstLength_16Bytes);
      TIM_DMACmd(TIM5,TIM_DMA_CC1,ENABLE);
      //如果需要 DMA 中断则如下面所示
          NVIC_InitStructure.NVIC_IRQChannel = DMA1_Stream5_IRQn;
                //使能 TIM 中断
          NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;        // 抢 占
优先级
          NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;
  //子优先级
          NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
          //使能中断
      NVIC_Init(&NVIC_InitStructure);
      DMA_ITConfig(DMA1_Stream5,DMA_IT_TC,ENABLE);
      //开启 DMA 传输
      DMA_Cmd(DMA1_Stream5, ENABLE);
  }
  void DMA1_Stream5_IRQHandler(void)
  {
      DMA_ClearITPendingBit(DMA1_Stream5,DMA_IT_TCIF5);
  }
```

思路四：使用外部时钟计数器

这种方法是我这几天回答问题时推荐的方法。思路是配置两个定时器，定时器 a 设置为外部时钟计数器模式，定时器 b 设置为定时器（比如 50ms 溢出一次，也可以用软件定时器），然后定时器 b 中断函数中统计定时器 a 在这段时间内的增量，简单计算即可。

代码：

```
//TIM7->100ms
//TIM2_CH2->PB3
void TIM_Cnt_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef    TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    TIM_DeInit(TIM2);
    TIM_DeInit(TIM7);

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2|RCC_APB1Periph_TIM7,ENABLE);
//TIM2 时钟使能
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);        //使能 PORTA 时
钟
//IO
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3; //GPIOA0
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;    //速度 100MHz
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; //下拉
    GPIO_Init(GPIOB,&GPIO_InitStructure); //初始化 PA0

    GPIO_PinAFConfig(GPIOB,GPIO_PinSource3,GPIO_AF_TIM2); //PA0 复用位定时器 5
//TIM2 配置
    TIM_TimeBaseStructure.TIM_Prescaler=0;   //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=0xFFFFFFFF;    //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);

    TIM_TIxExternalClockConfig(TIM2,TIM_TIxExternalCLK1Source_TI2,TIM_ICPolarity_Risi
ng,0);//外部时钟源
//TIM7   100ms
    TIM_TimeBaseStructure.TIM_Prescaler=18000-1;   //定时器分频
    TIM_TimeBaseStructure.TIM_CounterMode=TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseStructure.TIM_Period=1000-1;    //自动重装载值
    TIM_TimeBaseStructure.TIM_ClockDivision=TIM_CKD_DIV1;
    TIM_TimeBaseInit(TIM7,&TIM_TimeBaseStructure);
```

```
//中断
   NVIC_InitStructure.NVIC_IRQChannel = TIM7_IRQn;
      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;//抢占优先级 3
      NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;        //子优先级 3
      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;              //IRQ 通道使能
      NVIC_Init(&NVIC_InitStructure);  //根据指定的参数初始化 VIC 寄存器、
      TIM_ITConfig(TIM7,TIM_IT_Update,ENABLE);//允许更新中断 ,允许 CC1IE 捕获中断


      TIM_Cmd(TIM7,ENABLE );    //使能定时器 5
   TIM_Cmd(TIM2,ENABLE );       //使能定时器 5
}
u32 TIM7_LastCnt;
//频率为 TIM_ExtCntFreq
void TIM7_IRQHandler(void)
{
      char str[32];
      TIM_ExtCntFreq=(TIM2->CNT-TIM7_LastCnt)*SAMPLE_PERIOD;// SAMPLE_PERIOD 为
采样周期 0.1s
      sprintf(str,"%3.3f",TIM_ExtCntFreq/1000.0);//必须加这一句，莫明其妙
      TIM7_LastCnt=TIM2->CNT;
      TIM_ClearITPendingBit(TIM7,TIM_IT_Update);
}
```

缺点：

1. 无法测量占空比，高频的占空比测量方法见下文。
2. 在频率较低的情况下，测量精度不如思路 3（因为测量周期为 100ms，此时如果脉冲周期是 200ms……）。
3. 输入幅值必须超过 3V 。如果不够或者超出，需要加入前置放大器。

总结：这种方法精度很高，实测在 2MHz 之下误差为 30Hz 也就是 0.0015%（由中断服务程序引发，可以使用线性补偿修正），在 25MHz 之下也是误差 30Hz 左右（没法达到更高的原因是波形发生器的最大输出频率是 25MHz^_^）。同时，从根本上解决了中断频率过高的问题。而由于低频的问题，建议：在低频时，或者加大采样间隔（更改 TIM7 的周期），或者采用思路 3 的输入捕获。

此外，还有一个莫名其妙的问题就是，中断当中如果不加入 sprintf(str,"%3.3f",TIM_ExtCntFreq/1000.0)这一句，TIM_ExtCntFreq 就始终为 0 。我猜测是优化的问题，但是加入 volatile 也没有用，时间不够就没有理睬了。

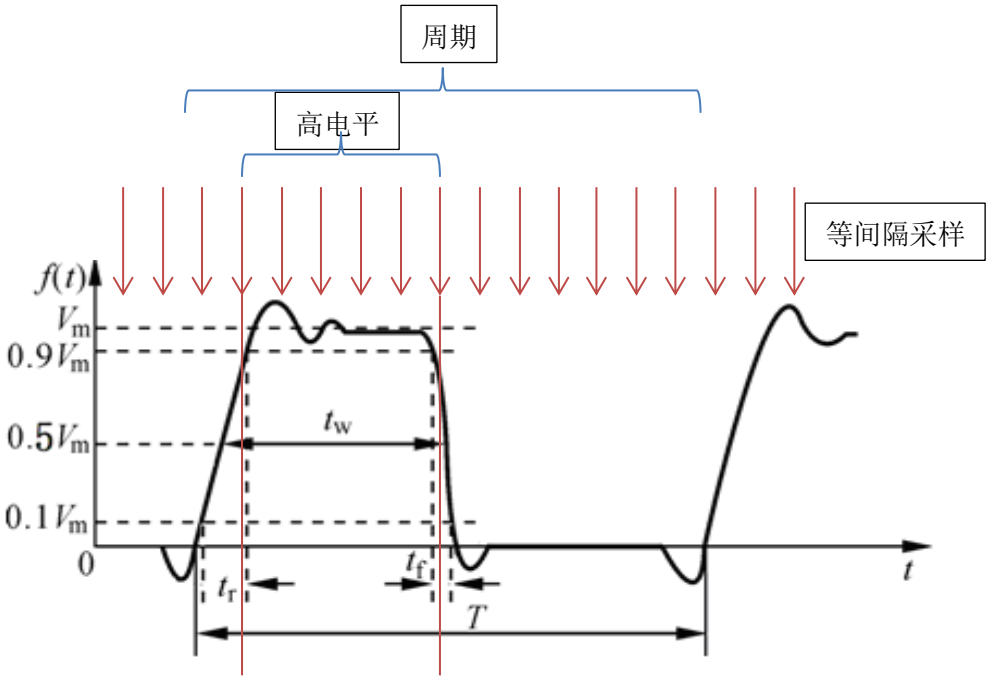
思路五：ADC 采样测量（概率测量法）

一般的高端示波器，测量频率即是这种方法。简而言之，高速采样一系列数据，然后通过频谱分析（例如快速傅里叶变换 FFT），获得频率。F4 有着 FPU 和 DSP 指令，计算速度上可以接受。但是 ADC 的采样频率远远达不到。官方手册上声明，在三通道交替采样+DMA 之下，最高可以达到 8.4M 的采样率。然而，根据香农采样定理，采样频率至少要达到信号的 2 倍。2M 信号和 8.4M 的采样率，即使能够计算，误差也无法接受。所以，ADC 采样是无法测量频率特别是高频频率的。

但是，无法测量频率，却可以测量占空比，乃至超调量和上升时间（信号从 10%幅值上升到 90%的时间）！原理也很简单，大学概率课上都说过这个概率基本原理：

$$\lim_{n\to\infty}\frac{N_A}{n}=P(A)$$

当采样数 n 趋于无穷时，事件 A 的概率即趋近于统计的频率。所以，当采样数越大，则采样到的高电平占样本总数的频率即趋近于概率——占空比！



因此，基本思路即是等间隔（速度无所谓，但必须是保证等概率采样）采样，并将这些数据存入一个数组，反复刷新。这样，可以在任意时间对数组中数据进行统计，获得占空比数据。

以下是代码，使用了三通道 8 位 ADC+DMA。理论上，采用查询法也是可以的。

```
//ADC1-CH13-PC3
//DMA2-CH0-STREAM0
  #define ADCx                    ADC1
  #define ADC_CHANNEL             ADC_Channel_13
  #define ADCx_CLK                RCC_APB2Periph_ADC1
  #define ADCx_CHANNEL_GPIO_CLK   RCC_AHB1Periph_GPIOC
  #define GPIO_PIN                GPIO_Pin_3
  #define GPIO_PORT               GPIOC
  #define DMA_CHANNELx            DMA_Channel_0
  #define DMA_STREAMx             DMA2_Stream0
  #define                                                 ADCx_DR_ADDRESS
((uint32_t)&(ADCx->DR))//((uint32_t)0x4001224C)
void ADC_DMAInit(void)
{
  ADC_InitTypeDef          ADC_InitStructure;
  ADC_CommonInitTypeDef ADC_CommonInitStructure;
  DMA_InitTypeDef          DMA_InitStructure;
```

```
    GPIO_InitTypeDef          GPIO_InitStructure;

    /* Enable ADCx, DMA and GPIO clocks ****************************************/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);
    RCC_AHB1PeriphClockCmd(ADCx_CHANNEL_GPIO_CLK, ENABLE);
    RCC_APB2PeriphClockCmd(ADCx_CLK, ENABLE);


    /* DMA2 Stream0 channel2 configuration **************************************/
    DMA_InitStructure.DMA_Channel = DMA_CHANNELx;
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)ADCx_DR_ADDRESS;
    DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)&(ADC_DATAPOOL[0]);
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
    DMA_InitStructure.DMA_BufferSize = ADC_POOLSIZE;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
    DMA_Init(DMA_STREAMx, &DMA_InitStructure);
    DMA_Cmd(DMA_STREAMx, ENABLE);

    /* Configure ADC3 Channel7 pin as analog input ******************************/
    GPIO_InitStructure.GPIO_Pin = GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
    GPIO_Init(GPIO_PORT, &GPIO_InitStructure);

    /*                    ADC                    Common                    Init
************************************************************/
    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
    ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_20Cycles;
    ADC_CommonInit(&ADC_CommonInitStructure);

    /*                                ADC3                                Init
************************************************************/
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_8b;
```

```
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
  ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfConversion = 1;
  ADC_Init(ADCx, &ADC_InitStructure);

  /* ADC3 regular channel7 configuration *************************************/
  ADC_RegularChannelConfig(ADCx, ADC_CHANNEL, 1, ADC_SampleTime_480Cycles);

 /* Enable DMA request after last transfer (Single-ADC mode) */
  ADC_DMARequestAfterLastTransferCmd(ADCx, ENABLE);

  /* Enable ADC3 DMA */
  ADC_DMACmd(ADCx, ENABLE);

  /* Enable ADC3 */
  ADC_Cmd(ADCx, ENABLE);
}
```

主程序：

```
for(j=0;j<ADC_POOLSIZE;j++)
{
    if(ADC_DATAPOOL[j]>0x01)
        posicnt++;
}
duty=100*posicnt/(float)(ADC_POOLSIZE)+0.1f;//线性补偿
```

缺点：

1. 精度低：实测 2MHz 下误差约 1.3%，低频时无法统计（比如，频率 10Hz，而 ADC 采样时间 50ms。这时如果采样时间中刚好全是高电平，占空比为 1……）。
2. 内存占用大：数据池大小为 65536，占用了 64KB 内存 。
3. 有响应延迟：测量出来的是“平均占空比”而非“瞬时占空比”。由于我测试时使用的是波形发生器，输出波形相当稳定（1W+的价格毕竟是有它的道理的……），实际应用当中一般不能够达到这样的水平，势必带来响应延迟（准确说应该是采样系统积分惯性越大）。
4. 幅值过低（0.3V）无法测量，过高则超过 ADC 允许最大值。所以必须视情况使用不同的前置放大器。

实际上使用时如何取舍，就需要看实际情况了。毕竟，这只是低成本下的解决方案而已。

综上，对这几种方法做一个总结：

外部中断：编写容易，通用性强。缺点是中断进入频繁，误差大。

PWM 输入：全硬件完成，CPU 负载小，编写容易。缺点是不稳定，误差大。

输入捕获：可达到约 400kHz。低频精度高，10Hz 可达到 0.01%以下，400kHz 也有 3%。

缺点是中断频繁，无法测量高频，幅值必须在 3.3~5V 之间。

外部时钟计数器（首选）：可达到非常高的频率（理论上应当是 90MHz）和非常低的误差（2MHz 下为 0.0015%且可线性补偿）。缺点是低频精度较低，同样幅值必须在 3.3~5V 之间。

ADC 采样频率测量法：难以测量频率，高频下对占空比、上升时间有可以接受的测量精度（2MHz 下约 1.3%），低频下无法测量。幅值 0.3~3.3V，加入前置放大则幅值随意。

ADC 采样频谱分析：高端示波器专用，STM32 弃疗。

我采用的方法是：首先 ADC 测量幅值并据此改变前置放大器放大倍数，调整幅值为 3.3V ，同时测量得到参考占空比。而后使用外部时钟计数器测量得到频率，如果较高（>10000）则确认为频率数据，同时 ADC 测量占空比确认为占空比数据。否则再使用输入捕获方法测量得到频率、占空比数据。

对于各个方法存在的线性误差，使用了线性补偿来提高精度。一般情况下，使用存储在 ROM 中的数据作为参数，当需要校正时，采用如下校正思路：

波形发生器生成一些预设参数波形（例如 10Hz，10%；100K，50%；2M，90%……），在不同区间内多次测量得到数据，随后以原始数据为 x，真实数据为 y，去除异常数据之后，做 y=f(x)的线性回归，并取相关系数最高的作为新的参数，同时存储在 ROM 当中。

我认为，我的这篇文章，应当是很全面了。当然，限于水平，存在着未完善和不正确的地方，也欢迎指正。

转载请注明作者——倾夜·陨灭星尘或 openedv 开源电子网的 yyx112358