STM32 MCU Development

STM32 单片机开发

-- STM32 定时器使用

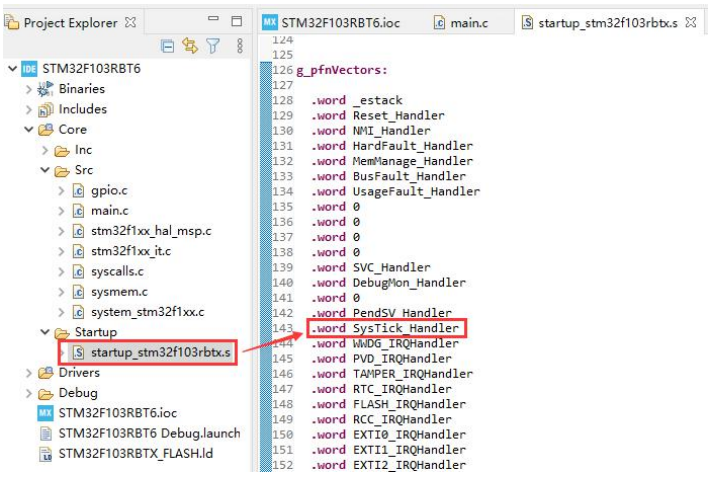
目录

1. Systick 和毫秒级延时实现.....3
1.1. Systick 工作原理.....3
1.2. 时钟树分析.....4
1.3. Systick 毫秒级延时实现.....5
2. 微秒级延时实现.....7
2.1. STM32 定时器介绍.....7
2.2. STM32CubeMX 配置.....9
2.3. 延时函数 delay_us()实现.....10
2.4. main()函数测试代码实现.....11
2.5. 运行测试.....11
3. 蜂鸣器使用.....12
3.1. 蜂鸣器介绍.....12
3.2. 蜂鸣器原理图.....13
3.3. STM32 定时器与 PWM.....13
3.4. STM32CubeMX 配置.....15
3.5. 添加蜂鸣器操作函数.....18
3.6. main()函数测试代码实现.....19
3.7. 运行测试.....19

1. Systick 和毫秒级延时实现

1.1. Systick 工作原理

Systick(系统定时器)是 ARM Cortex M3/M4 内核的一个外设，因为所有的 CM3/M4 内核的单片机都带有这个定时器，这使得软件在 CM3/M4 单片机中可以很容被移植。系统定时器一般用于单片机操作系统产生时间，维持 OS 的心跳和实现任务分时调度等，Systick 定时器是如此的重要，以至于 CM3/M4 为它专门开出一个异常类型，并且在向量表中有它的一席之地。



Systick 是一个 24 位的向下递的计数器，每当 Systick 从时钟源到来一个时钟，其值就会减 1，而一般我们将 Systick 的时钟源设置为系统时钟 HCLK(72MHz)，这样也就意味着每过 1/72M 秒 Systick 里的计数器将会减 1，当重装载数值寄存器里的值递减为 0 的时候，系统定时器就会产生一次中断，这样就有时间了。之后 CPU 自动重新装载计数器值并逐渐递减循环往复。

表8.9 SysTick控制及状态寄存器（地址：0xE000_E010）

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经数到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

表8.10 SysTick重装载数值寄存器（地址：0xE000_E014）

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时，将被重装载的值

Systick 有两个时钟源可供选择，通过配置 Systick 控制及状态寄存器实现：

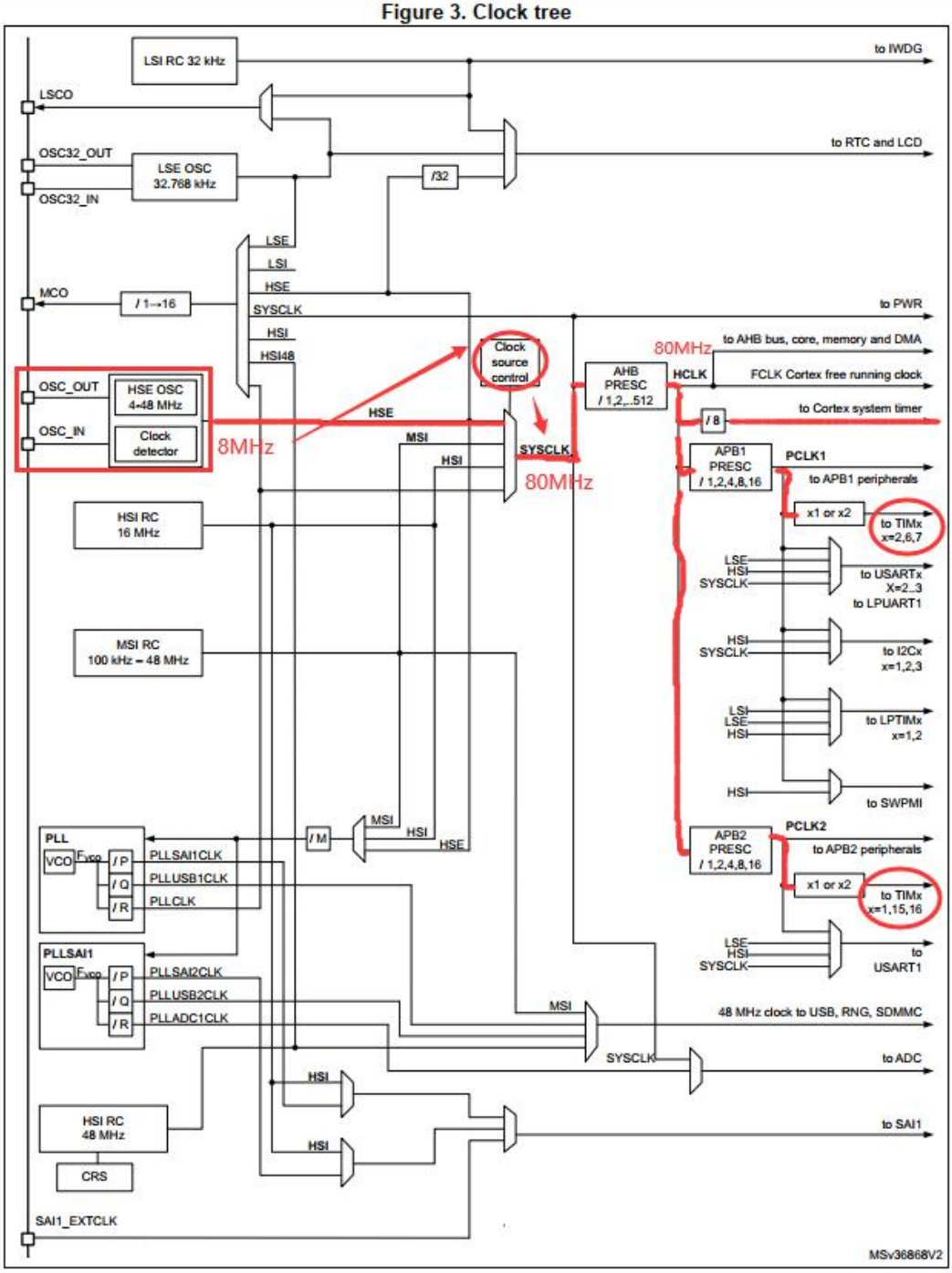
Several prescalers allow the configuration of the AHB frequency, the high speed APB (APB2) and the low speed APB (APB1) domains. The maximum frequency of the AHB and the APB2 domains is 72 MHz. The maximum allowed frequency of the APB1 domain is 36 MHz. The SDIO AHB interface is clocked with a fixed frequency equal to HCLK/2

The RCC feeds the Cortex® System Timer (SysTick) external clock with the AHB clock (HCLK) divided by 8. The SysTick can work either with this clock or with the Cortex® clock (HCLK), configurable in the SysTick Control and Status Register. The ADCs are clocked by the clock of the High Speed domain (APB2) divided by 2, 4, 6 or 8.

The Flash memory programming interface clock (FLITFCLK) is always the HSI clock.

1.2. 时钟树分析

Figure 3. Clock tree



1.3. Systick 毫秒级延时实现

Systick 上电初始化(外部晶振还没开始工作，此时系统默认使用内部 HSI 提供时钟源)

main() -> HAL_Init() -> HAL_InitTick() -> HAL_SYSTICK_Config() -> SysTick_Config(SystemCoreClock / (1000U / uwTickFreq)

```
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
  if ((ticks - 1UL) > SysTick_LOAD_RELOAD_Msk)
  {
    return (1UL);                                                   /* Reload value impossible */
  }

  SysTick->LOAD  = (uint32_t)(ticks - 1UL);                         /* set reload register */
  NVIC_SetPriority (SysTick_IRQn, (1UL << __NVIC_PRIO_BITS) - 1UL); /* set Priority for Systick Interrupt */
  SysTick->VAL   = 0UL;                                             /* Load the SysTick Counter Value */
  SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk   |
                   SysTick_CTRL_ENABLE_Msk;                         /* Enable SysTick IRQ and SysTick Timer */
  return (0UL);                                                     /* Function successful */
}
```

外部配置好后重新初始化 Systick

main() -> SystemClock_Config() -> RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK; HAL_RCC_ClockConfig() -> HAL_InitTick() -> HAL_SYSTICK_Config() -> SysTick_Config(SystemCoreClock / (1000U / uwTickFreq)

```
898     }
899     __HAL_RCC_SYSCLK_CONFIG(RCC_ClkInitStruct->SYSCLKSource);
900
901     /* Get Start Tick */
902     tickstart = HAL_GetTick();
903
904     while (__HAL_RCC_GET_SYSCLK_SOURCE() != (RCC_ClkInitStruct->SYSCLKSource << RCC_CFGR_SWS_Pos))
905     {
906       if ((HAL_GetTick() - tickstart) > CLOCKSWITCH_TIMEOUT_VALUE)
907       {
908         return HAL_TIMEOUT;
909       }
910     }

942
943   /* Update the SystemCoreClock global variable */
944   SystemCoreClock = HAL_RCC_GetSysClockFreq() >> AHBPrescTable[(RCC->CFGR & RCC_CFGR_HPRE) >> RCC_CFGR_HPRE_Pos];
945
946   /* Configure the source of time base considering new system clocks settings*/
947   HAL_InitTick(uwTickPrio);
948
949   return HAL_OK;
950 }
```

SysTick_Config(SystemCoreClock / (1000U / uwTickFreq)

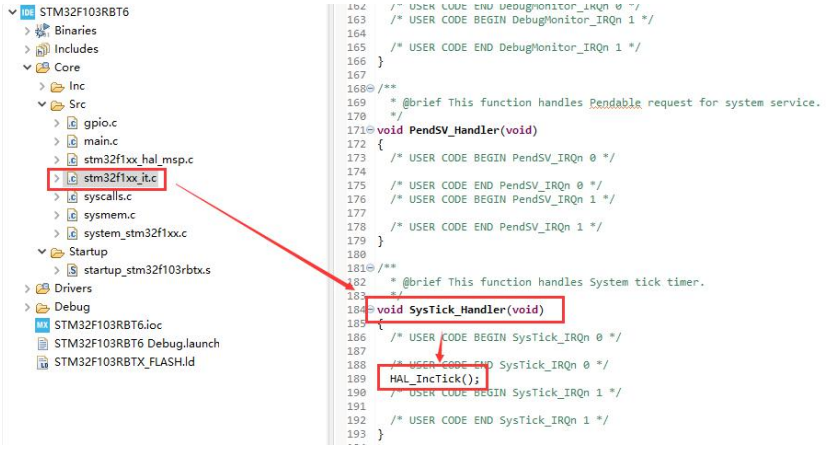
= SysTick_Config (72 000 000 / 1000U / HAL_TICK_FREQ_1KH)

= SysTick_Config (72 000) ==> SysTick->LOAD = 72000;

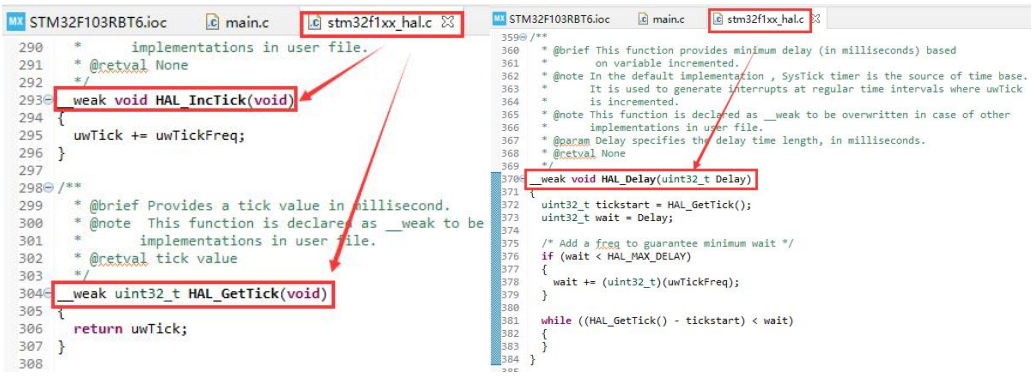
SysTick 的时钟源为 72MHz，即 1 秒钟要来 72M 次时钟脉冲；而此时重装载数值寄存器(LOAD)的值设置为 72 000，即每来 72K 个脉冲就产生一次中断；那每次中断的时间间隔就是：72K/72M=1/1000 秒=1ms。

这样我们就能够通过 Systick 定时器的中断实现毫秒级的延时了。

Systick 中断服务处理程序：



HAL_Delay()毫秒级延时实现：



2. 微秒级延时实现

HAL 库函数中有函数 HAL_Delay() 进行毫秒级的延时，但是在实际的开发中有时需要进行较为准确的微秒级别延时，如接下来我们将要实现的 DHT11 温湿度传感器采样驱动等。本章将采用一个通用定时器 TIM4 实现微秒级别的延时。

2.1. STM32 定时器介绍

STM32L433 除了通用的 Systick 定时器以外，另外还有 6 个定时器：TIM1、TIM2、TIM6、TIM7、TIM15、TIM16。TIM6、TIM7 是两个 16 位的自装载基本定时器，它们只能作定时使用，而 TIM1、TIM2、TIM15、TIM16 则是通用高级定时器，除了定时功能以外还能作 PWM 输出，接下来蜂鸣器的驱动中将会用到。

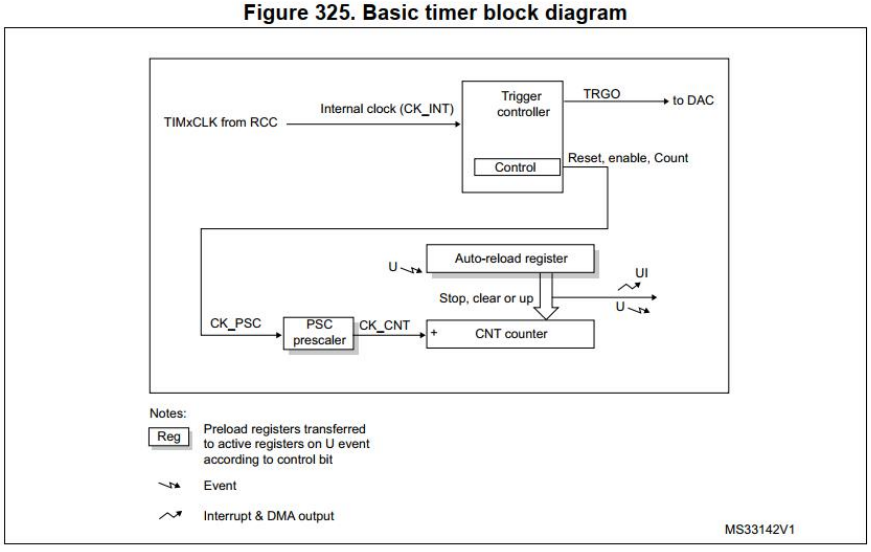
接下来我们将选择基本定时 TIM6 来实现 us 级的定时功能。

29.2 TIM6/TIM7 main features

Basic timer (TIM6/TIM7) features include:

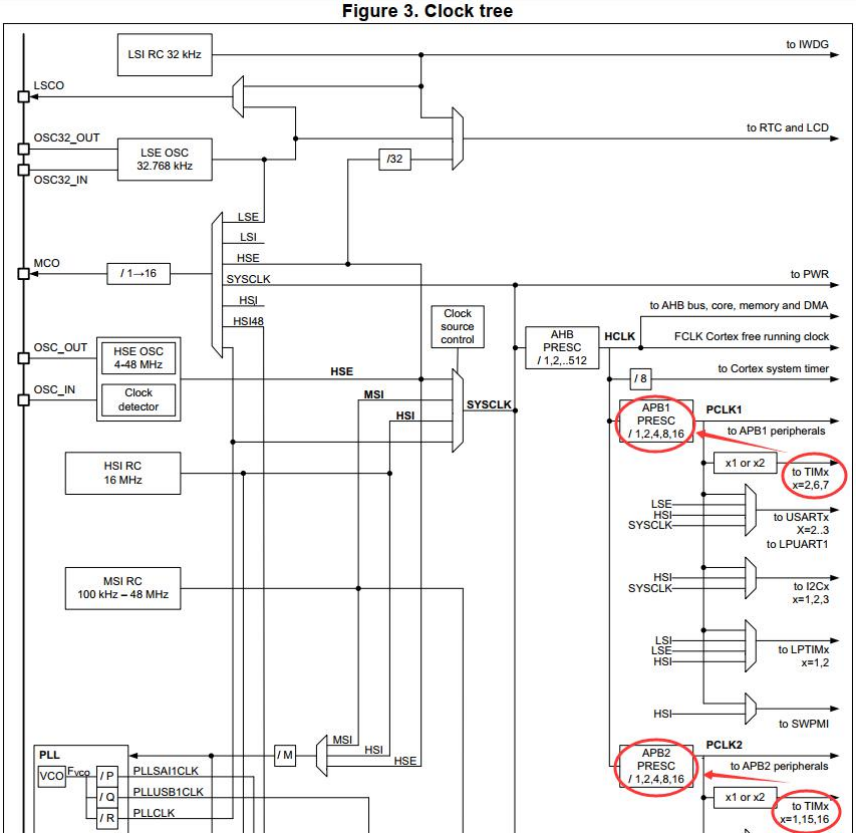
- 16-bit auto-reload upcounter
- 16-bit programmable prescaler used to divide (also “on the fly”) the counter clock frequency by any factor between 1 and 65535
- Synchronization circuit to trigger the DAC
- Interrupt/DMA generation on the update event: counter overflow

Figure 325. Basic timer block diagram

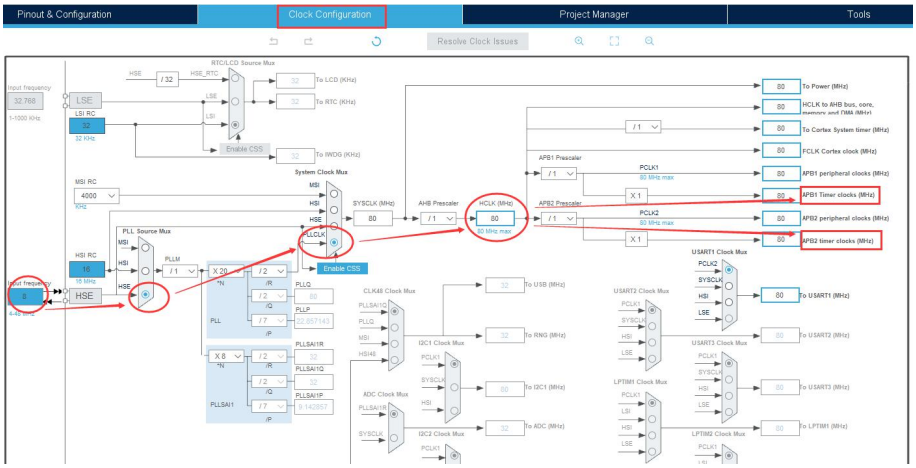


STM32L433CBT6 的芯片 datasheet 中关于时钟树的说明，我们可以看到 TIM2,6,7 连到 APB1 总线上，而 TIM1,15,16 则连到了 APB2 总线上。

Figure 3. Clock tree

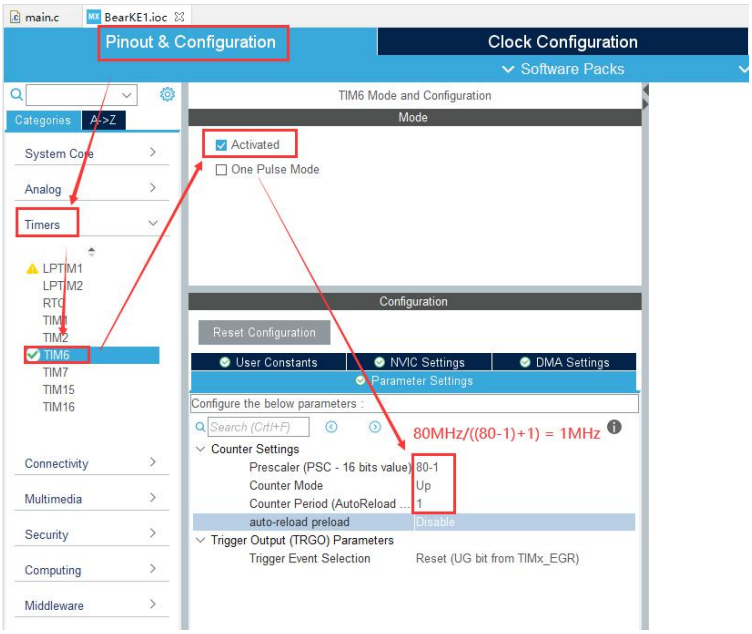


在我们的时钟树配置中，APB1 和 APB2 总线的频率都被设置为 80MHz，所以 TIM6 的输入时钟为 80MHz，接下来我们定时器的配置跟它高度相关。



2.2. STM32CubeMX 配置

微秒级的延时如果使能中断方式的话，会频繁打断 CPU 正常执行程序，大幅降低系统的工作效率，所以我们这里微秒级的延时就不使用中断模式。下面是 STM32CubeMX 的配置：



- ✓ 配置预分频：TIM6 的输入时钟为 APB1 时钟 80MHz，这个速率对定时器来说实在太快，这时需要对它做个预分频：CK_CNT = TIMxCLK/(PSC+1)=80MHz/(80-1+1)=1MHz；
- ✓ 微秒延时配置：通过修改 TIM6 定时器的 ARR(自动重装载寄存)的值，就可以配置定时器的超时时间：ARR=1/1MHz = 1us

配置好后按 Ctrl+S 重新生成代码：

```
TIM_HandleTypeDef htim6;

/* TIM6 init function */
void MX_TIM6_Init(void)
{

  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 80-1;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 1;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */

  /* USER CODE END TIM6_Init 2 */

}
```

2.3. 延时函数 delay_us()实现

修改 time.c 源文件，在合适位置添加 delay_us()函数实现：

```
  ... ...
/* USER CODE BEGIN 1 */

/* TIM6 是 16 位的计数器(2^16=65535)，这里我们 us 级延时最大到 60000 */
void delay_us(uint16_t us)
{
    /* 中断会打断微秒延时函数但计数器计数不会停,防止计数器增加到最大计数之后重新开始计数 */
    uint16_t differ = 60000-us;

    HAL_TIM_Base_Start(&htim6);

    __HAL_TIM_SET_COUNTER(&htim6, differ);

    while( differ < 60000 )
    {
        differ=__HAL_TIM_GET_COUNTER(&htim6);
    }

    HAL_TIM_Base_Stop(&htim6);
}

/* USER CODE END 1 */  ... ...
```

修改 tim.h 源文件，在合适位置添加 delay_us()函数声明：

```
/* USER CODE BEGIN Prototypes */

extern void delay_us(uint16_t us);  /* us max to 60000 */

/* USER CODE END Prototypes */
```

2.4. main()函数测试代码实现

main 函数的循环里添加测试代码，因为 us 级的延时太小我们感觉不到，所以我们在这里通过循环多次来实现 Led 每秒钟闪烁一次。

```
... ...
 /* USER CODE BEGIN WHILE */
 sysled_hearbeat();
 while (1)
 {
    {
          int     i;

          turn_led(RedLed, ON);

          for(i=0; i<100; i++)
          {
               delay_us(5000);
          }

          turn_led(RedLed, OFF);

          for(i=0; i<100; i++)
          {
               delay_us(5000);
          }
    }
   /* USER CODE END WHILE */

  /* USER CODE END 3 */
}
... ...
```

2.5. 运行测试

源码编译、烧录及运行，观察红色 Led 灯的状态变化，看是否是 1 秒钟闪烁一次。

3. 蜂鸣器使用

3.1. 蜂鸣器介绍

蜂鸣器(beep/buzzer)是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、报警器、电子玩具、汽车电子设备、等电子产品中作发声器件，使用蜂鸣器来做提示或报警，比如按键按下、开始工作、工作结束或是故障等。

蜂鸣器主要分为压电式蜂鸣器和电磁式蜂鸣器两种类型：

1. 压电式蜂鸣器： 压电式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成，有的压电式蜂鸣器外壳上还装有发光二极管。多谐振荡器由晶体管或集成电路构成。当接通电源后（1.5~15V 直流工作电压）,多谐振荡器起振,输出 1.5~2.5kHZ 的音频信号，阻抗匹配器推动压电蜂鸣片发声。压电蜂鸣片由锆钛酸铅或铌镁酸铅压电陶瓷材料制成，在陶瓷片的两面镀上银电极经极化和老化处理后，再与黄铜片或不锈钢片粘在一起。

2. 电磁式蜂鸣器：电磁式蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

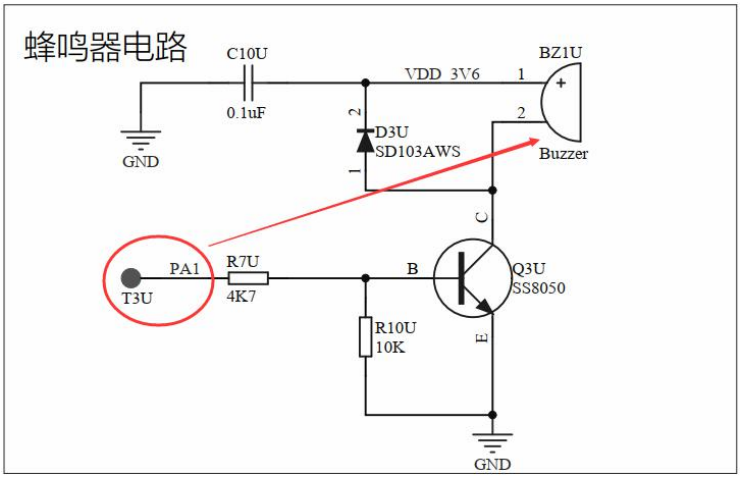
一般蜂鸣器在工作时需要给其输入一个某个频率的震荡源(PWM 方波)才会发声，这时候就分有源蜂鸣器与无源蜂鸣器两种，这里的“源”是指震荡源而不是指电源：

- 有源蜂鸣器内部带震荡源，所以只要一通电(给高电平或低电平)就会叫，即电平触发；
- 而无源蜂鸣器内部不带震荡源，直流信号无法令其鸣叫，必须用 2K~5K 的方波去驱动它；

有源蜂鸣器往往比无源的贵，因为里面内含振荡源，只要一通电就发声，但发生频率固定，音色单一；无源蜂鸣器内部不含振荡源，内部结构相当于电磁场扬声器，必须给他输入一定频率的信号才能发声，但这个频率软件编程可控，这样就可以做出“多来米发索拉西”等不同音色的效果。

3.2. 蜂鸣器原理图

下图是 BearKE1 开发板原理图上蜂鸣器的电路图，由下图可知它受 PA1 管脚的控制。



在该开发板上我们采用的是无源蜂鸣器，因为人耳能听到的频率范围在 20Hz--20kHz 之间，此时我们只需要使用 PA1 管脚的定时器 PWM 功能输出功能，输出一个 2.4KHz 左右的 PWM 方波蜂鸣器就会发声了，通过调整 PWM 的频率就可以调整蜂鸣器的音色了。

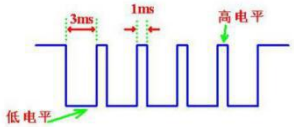
3.3. STM32 定时器与 PWM

假设你想通过 5V 电源控制灯的亮度，有个传统办法就是串联一个可调电阻，通过改变电阻，灯的亮度就会改变。还有一个办法，就是 PWM 调节。它不用串联电阻，而是串联一个开关。假设在 1s 内有 0.5s 打开、另外 0.5s 关闭，这样持续下去，灯就会闪烁，只是这时候频率(1Hz)太慢人感受得到不舒服。

如果我们把频率调高一点到 1KHz，这时候是 1ms 内 0.5ms 开、0.5ms 灭，那么此时灯的闪烁频率就很高。我们知道，闪烁频率超过一定值，人眼就会感觉不到。所以，这时你看不到灯的闪烁，只看到灯的亮度只有原来的一半，我们看到的呼吸灯就是这个工作原理。

PWM(Pulse Width Modulation，脉冲宽度调制)是通过对一系列脉冲的宽度进行调制，输出所需要的波形（包含形状以及幅值）。在 PWM 里有两个重要的专业术语：

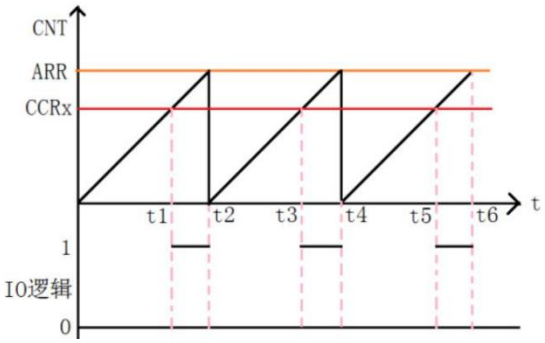
- 频率是指 1 秒内高低电平周期性出现的次数；
- 占空比就是指在一个周期内，信号处于高电平的时间占据整个信号周期的百分比，例如下图所示的 PWM 波形占空比为 25%，而方波则是指占空比为 50%的 PWM 波形。



占空比为25%信号

PWM 在很多场合都会使用，我们经常见到的就是交流调光电路，也可以说是无级调速，高电平占多一点，也就是占空比大一点亮度就亮一点，占空比小一点亮度就没有那么亮，前提是 PWM 的频率要大于我们人眼识别频率，要不然会出现闪烁现象。除了在调光电路应用，还有在直流斩波电路、蜂鸣器驱动、电机驱动、逆变电路、加湿机雾化量等都会有应用。

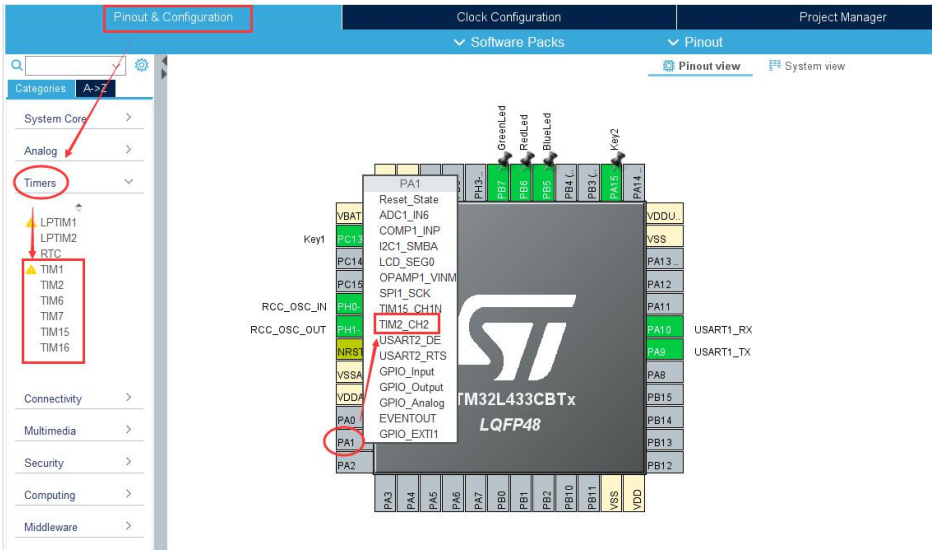
STM32 的定时器除了 TIM6 和 TIM7（基本定时器）之外，其他的定时器都可以产生 PWM 输出。其中，高级定时器 TIM1、TIM8 可以同时产生 7 路 PWM 输出，而通用定时器可以同时产生 4 路 PWM 输出，这样 STM32 最多可以同时产生 30 路 PWM 输出。其工作原理如下图所示：



在 STM32 的通用定时器中有两个重要的寄存器：

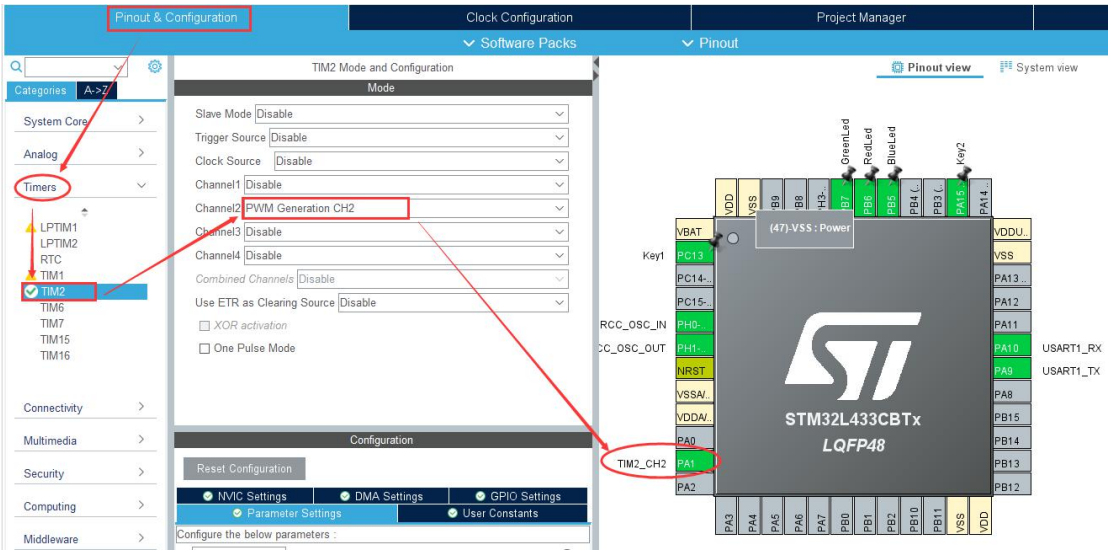
- ARR(Auto-reload register，自动重装载寄存器)是个 16 位的寄存器，这里面装着计数器能计数的最大数值。如果使能了中断，当计数到达这个值时定时器就产生溢出中断；
- CCR(Capture/Compare Registers，捕获/比较寄存器)，CCR 与定时器的值进行比较：如果 CCR 大则输出低电平，反之则输出高电平。通过改变 CCR 值可以改变 PWM 的占空比；

如下图所示，在 STM32L433 这颗芯片里总共提供了 6 个定时器，其中蜂鸣器连接的管脚 PA1 可以作为 TIM2_CH2 使用，这时我们要让蜂鸣器工作只需要配置使能 TIM2 定时器的 CH2 通道输出 PWM 即可。

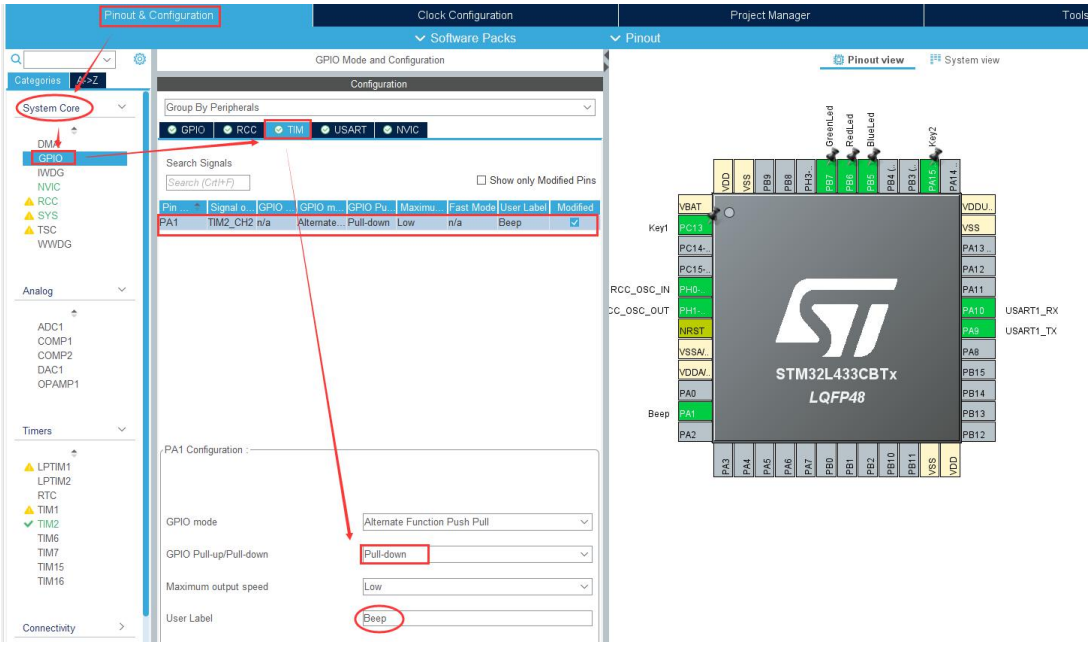


3.4. STM32CubeMX 配置

首先配置使能 TIM2，让 Channel2 输出 PWM。这时 PA1 管脚的状态就被设置成 TIM2_CH2 功能模式了。

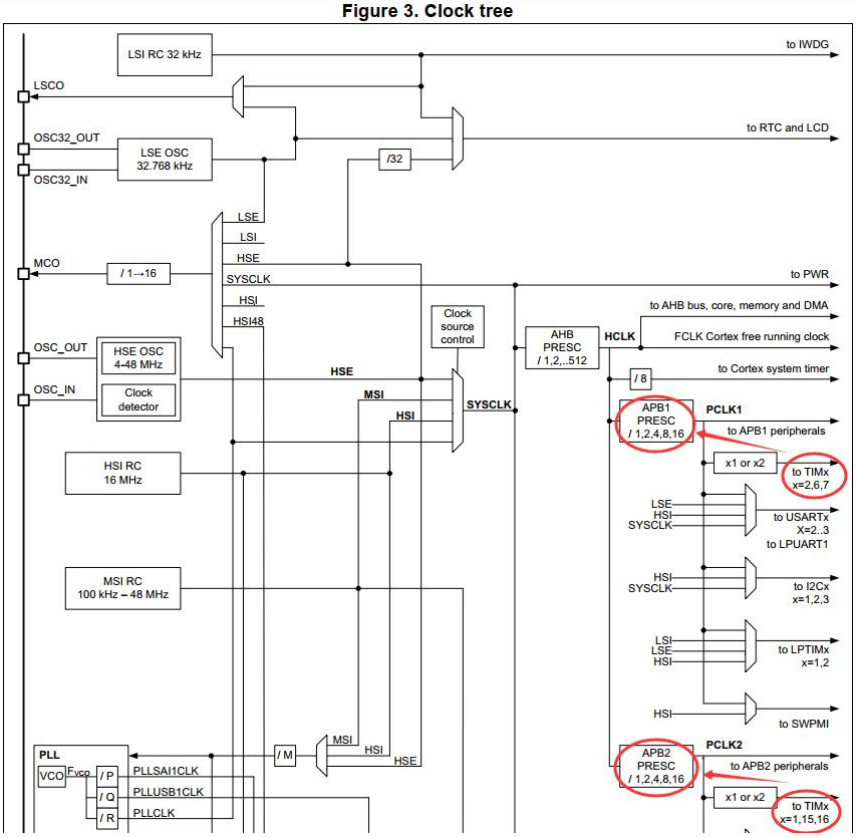


因为 PWM 会输出高低电平，接着设置 GPIO 下拉防止蜂鸣器误发声音。

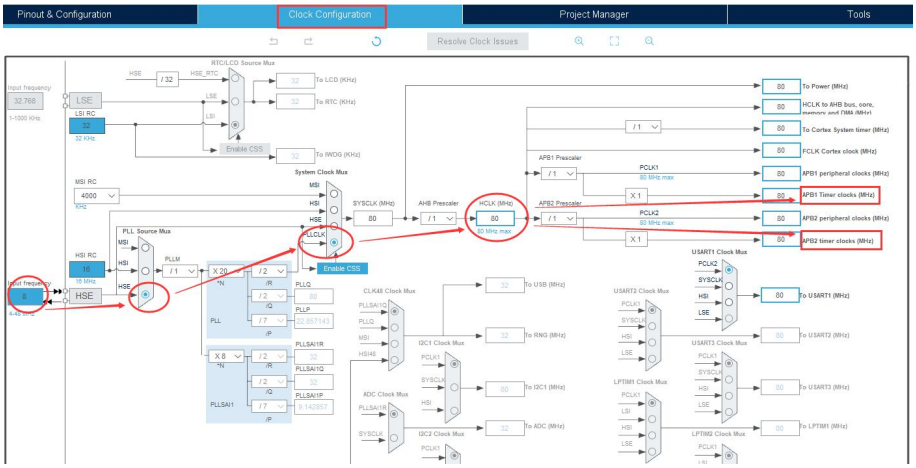


STM32L433CBT6 的芯片 datasheet 中关于时钟树的说明，我们可以看到 TIM2,6,7 连到 APB1 总线上，而 TIM1,15,16 则连到了 APB2 总线上。

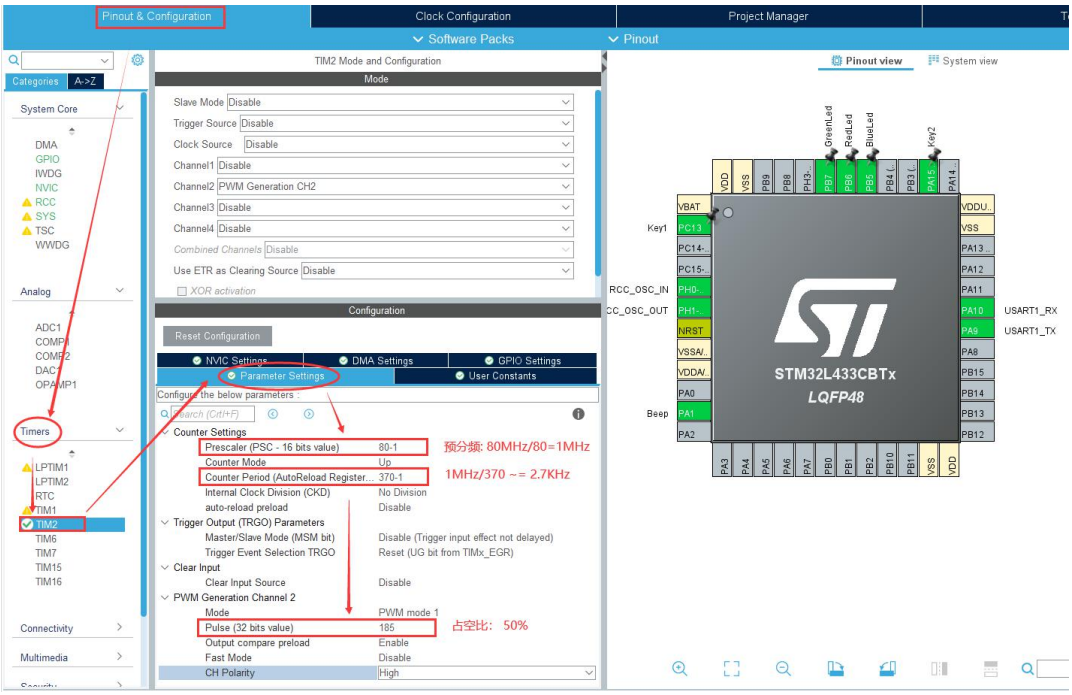
Figure 3. Clock tree



在我们的时钟树配置中，APB1 和 APB2 总线的频率都被设置为 80MHz，所以 TIM2 的输入时钟为 80MHz，接下来我们要生成的 PWM 方波频率跟它高度相关。

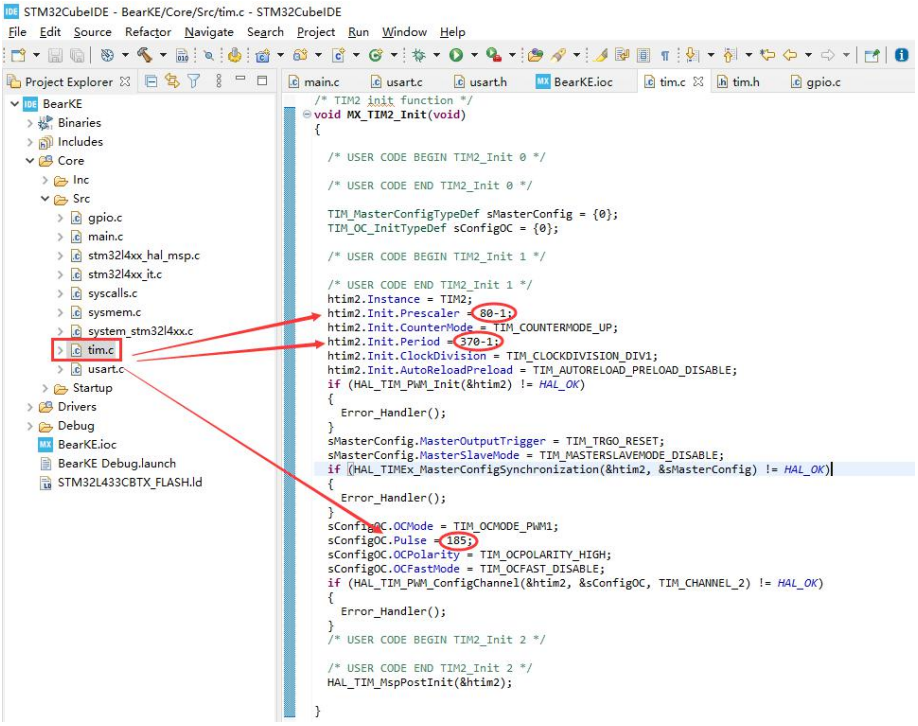


蜂鸣器发声的响亮(音色)与 PWM 输出的频率相关，其工作频率在 2.7KHz 左右，频率越高声音越脆。接下来我们设置其频率和占空比：



- ✓ 配置预分频：TIM2 的输入时钟为 APB1 时钟 80MHz，这个速率对定时器来说实在太快，这时需要对它做个预分频：CK_CNT = TIMxCLK/(PSC+1)=80MHz/(80-1+1)=1MHz；
- ✓ 配置 PWM 时钟：通过 TIM2 定时器的 ARR(自动重装载寄存)的值可以调整 PWM 的输出频率，这里想让蜂鸣器工作在 2.7KHz，则：ARR=1MHz/2700 ~= 370
- ✓ 配置占空比： 这里我们设置占空比为 50%，则 Pulse=370/2=185

任意位置按 Ctrl+S 将开始自动生成代码。



3.5. 添加蜂鸣器操作函数

修改 tim.c 文件，添加蜂鸣器操作函数 beep_start()实现，其中 times 为蜂鸣器响几次，而 interval 为响、停的时间间隔：

```
... ...
/* USER CODE BEGIN 1 */
void beep_start(uint8_t times, uint16_t interval)
{
    while( times-- )
    {
        /* Start buzzer */
        if (HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2) != HAL_OK)
        {
            /* Starting Error */
            Error_Handler();
        }

        HAL_Delay(interval);

        /* Stop buzzer */
        if (HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_2) != HAL_OK)
```

```
        {
            /* Starting Error */
            Error_Handler();
        }

        HAL_Delay(interval);
    }
}
/* USER CODE END 1 */
... ...
```

修改 tim.h，添加 蜂鸣器操作函数的声明：

```
... ...
/* USER CODE BEGIN Prototypes */
extern void delay_us(uint16_t us);  /* us max to 60000 */
extern void beep_start(uint8_t times, uint16_t interval);
/* USER CODE END Prototypes */
... ...
```

3.6. main()函数测试代码实现

修改 main.c 文件，添加 printf()函数调用：

```
... ...
int main(void)
{
 ... ...
  /* USER CODE BEGIN WHILE */
  sysled_hearbeat();
  beep_start(2, 300);
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

3.7. 运行测试

编译、烧录并重新运行程序，系统重启时就会听到蜂鸣器鸣响 2 次。