第 5 章

CHAPTER 5

定时器与蜂鸣器

5.1 STM32F103 定时器概述

MCU 中的定时器实质上是一个计数器，可以对内部脉冲或者外部输入进行计数。相比传统 51 单片机，STM32F103 的定时器要完善和复杂很多，它们是专为工业控制应用量身定做的，具有延时、频率测量、PWM 输出、电机控制及编码接口等诸多功能。

STM32F103 内部集成了 8 个可编程定时器，分为基本定时器、通用定时器和高级定时器三种类型，它们都具有 16 位定时器分辨率和 16 位可编程的预分频系数，都可以产生 DMA 请求，如表 5-1 所示。

表 5-1 STM32F103 定时器分类

主 要 特 点	基本定时器	通用定时器	高级定时器
定时器	TIM6、TIM7	TIM2、TIM3、TIM4、TIM5	TIM1、TIM8
内部时钟 CK_INT 的来源	APB1 分频器	APB1 分频器	APB2 分频器
计数类型	向上	向上、向下、向上/向下	向上、向下、向上/向下
比较/捕捉通道	0	4	4
互补输出	没有	没有	有
定时器分辨率	16 位		
预分频系数	16 位(1～65536)		
产生 DMA 请求	可以		

基本定时器 TIM6/7 只能向上计数，没有外部 I/O 功能。通用定时器 TIM2/3/4/5 可以向上、向下、向上/向下计数，有四个外部 I/O，具有输入捕捉、输出比较功能。高级定时器 TIM1/8 除具有定时、输入捕捉、输出比较功能外，还具有三相电机互补输出。本章主要以基本定时器 TIM6 为例来学习 STM32F103 定时器。

5.2 基本定时器原理

STM32F103 的两个基本定时器 TIM6/7 的核心是由可编程预分频器驱动的 16 位自动

重装载计数器。在更新事件(计数器溢出)发生时,基本定时器 TIM6/7 会产生中断/DMA 请求。此外,它们还具有触发 DAC 的同步电路,如图 5-1 所示。时基单元是基本定时器最重要的组成部分,由自动重装载寄存器(TIMx_ARR)、预分频寄存器(TIMx_PSC)和计数器寄存器(TIMx_CNT)组成,在运行时可以通过软件读写这三个寄存器。基本定时器只有一个内部时钟(CK_INT),TIMxCLK 来源于 APB1 预分频器的输出,默认情况下时钟频率为 72MHz。

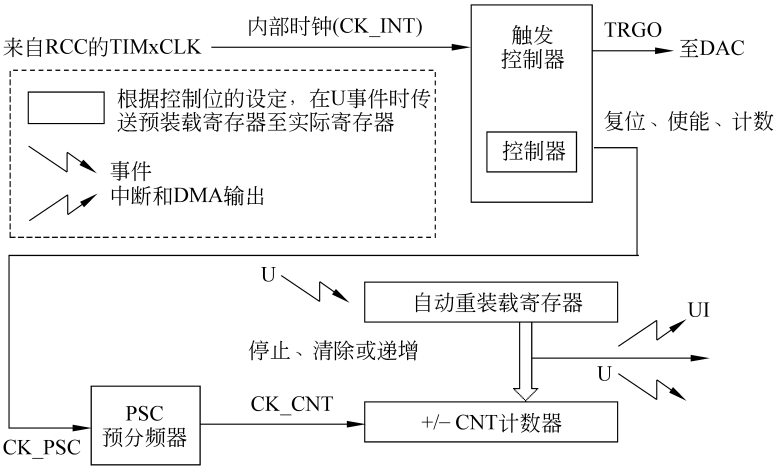

图 5-1　基本定时器的功能框图

1. 自动重装载寄存器(TIMx_ARR)

如图 5-2 所示,TIMx_ARR 在物理上实际对应了两个寄存器:一个是用户可以读写的寄存器,称为预装载寄存器;另一个是用户看不见但真正起作用的寄存器,称为影子寄存器,影子寄存器就是预装载寄存器的一份复制。自动重装载寄存器是预加载的,每次读写它时,实际上是通过读写预装载寄存器实现的。

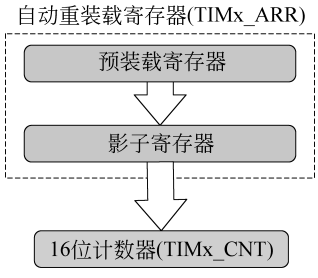

图 5-2　自动重装载寄存器组成示意图

将基本定时器的控制寄存器(TIMx_CR1)的自动重装载预装载允许位(ARPE)使能,写入预装载寄存器的内容在下一次更新事件发生后才传送到影子寄存器,否则写入自动重装载寄存器的值会被立即更新到影子寄存器。通过软件可以使能或禁止定时器更新事件的产生。

为什么要设计预装载寄存器和影子寄存器呢?因为软件不能在一个相同的时刻同时更新多个寄存器,也就不能同步多个通道的时序,若再有其他因素(如中断)影响,多个通道的时序关系就不可预知了。设计两个寄存器则可以在同一个时刻(发生更新事件时),将预装载寄存器的内容更新到所对应的真正起作用的影子寄存器中,这样就能够准确地同步多个通道的操作了。

2. 预分频器

预分频寄存器带有缓冲器,定时器运行时就能改变它的值,新值会在下一次更新事件到来时生效。预分频器可以以系数为 1～65536 的任意整数值对预分频器的输入时钟进行分频,得到新的计数器时钟 CK_CNT,计数器时钟频率的计算公式如下:

计数器时钟 f_{CK_CNT} = 计数器预分频时钟 f_{CK_PSC}/(PSC[15:0]+1)

如图 5-3 所示，预分频值 PSC[15:0]=0 时，预分频系数为 1，预分频时钟与计数器时钟相同。当预分频寄存器(TIMx_PSC)写入新的预分频值 PSC[15:0]=3 时，预分频系数并没有立即更改，而是在更新事件到来时才从 1 变为 4。

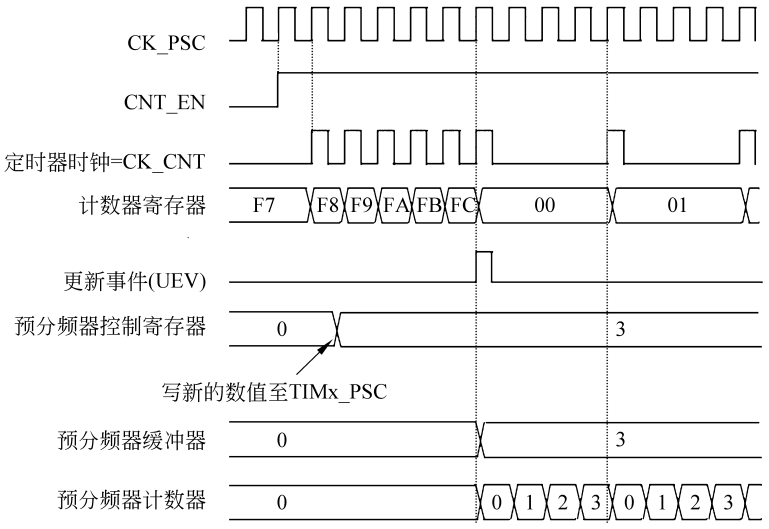

图 5-3　预分频系数从 1 变为 4 的计数器时序图

3. 计数器

TIMx_CNT 是一个带有自动重装载的 16 位累加计数器，存储了当前定时器的计数值，它的计数时钟是通过预分频器(PSC)得到，由 PSC 输出的时钟 CK_CNT 驱动。计数器(CNT)只能往上计数，最大计数值为 65535。计数器从 0 累加计数到自动重装载数值后，将产生一个计数器溢出事件并重新从 0 开始计数，每次计数器溢出时可以产生更新事件。

如图 5-4 所示，当预分频系数为 4 且寄存器 TIMx_ARR 的值为 0x36 时，每 4 个预分频时钟计数器向上计数 1 次，当计数值达到 0x0036 时发生计数溢出，计数值清零并产生更新事件，同时设置更新中断标志位。

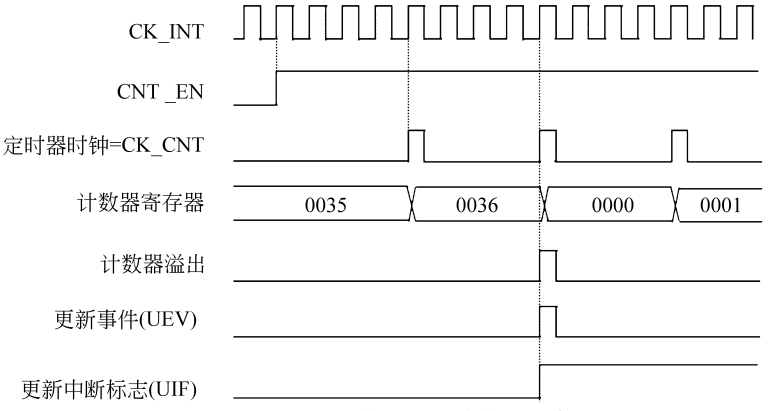

图 5-4　预分频系数为 4 时计数器时序图

寄存器 TIMx_CR1 中的计数器使能位(CEN)用于使能或关闭计数器计数。

4. 定时器的定时时间计算

在时钟 CK_CNT 的驱动下，计数器计一个数的时间是 1/CK_CNT＝1/(T_{CLK}/(PSC＋1))，产生一次中断的时间为(1/CK_CNT)(ARR＋1)，因此定时器的定时时间为

$$(1/CK\_CNT)(ARR+1)=((PSC+1)(ARR+1))/T_{CLK}$$

其中，PSC 为预分频器的值(为 0～65535)，ARR 为自动重装载寄存器的值(为 0～65535)，T_{CLK}为输入时钟频率。

5.3　定时器的 HAL 库用法

5.3.1　定时器寄存器结构体 TIM_TypeDef

结构体数据类型 TIM_TypeDef 将与定时器相关的寄存器进行了封装，该结构体与定时器的宏都是在文件 stm32f103xe.h 中定义的，代码如下：

```
typedef struct
{
    __IO uint32_t CR1;                          /* TIM 控制寄存器 1 */
    __IO uint32_t CR2;                          /* TIM 控制寄存器 2 */
    __IO uint32_t SMCR;                         /* TIM 从机模式控制寄存器 */
    __IO uint32_t DIER;                         /* TIM DMA/中断使能寄存器 */
    __IO uint32_t SR;                           /* TIM 状态寄存器 */
    __IO uint32_t EGR;                          /* TIM 事件生成寄存器 */
    __IO uint32_t CCMR1;                        /* TIM 捕获/比较模式寄存器 1 */
    __IO uint32_t CCMR2;                        /* TIM 捕获/比较模式寄存器 2 */
    __IO uint32_t CCER;                         /* TIM 捕获/比较使能寄存器 */
    __IO uint32_t CNT;                          /* TIM 计数器寄存器 */
    __IO uint32_t PSC;                          /* TIM 预分频器寄存器 */
    __IO uint32_t ARR;                          /* TIM 自动重加载寄存器 */
    __IO uint32_t RCR;                          /* TIM 重复计数器寄存器 */
    __IO uint32_t CCR1;                         /* TIM 捕获/比较寄存器 1 */
    __IO uint32_t CCR2;                         /* TIM 捕获/比较寄存器 2 */
    __IO uint32_t CCR3;                         /* TIM 捕获/比较寄存器 3 */
    __IO uint32_t CCR4;                         /* TIM 捕获/比较寄存器 4 */
    __IO uint32_t BDTR;                         /* TIM 中断和停滞时间寄存器 */
    __IO uint32_t DCR;                          /* TIM DMA 控制寄存器 */
    __IO uint32_t DMAR;                         /* TIM DMA 完整传输寄存器地址 */
    __IO uint32_t OR;                           /* TIM 选项寄存器 */
} TIM_TypeDef;

#define APB1PERIPH_BASE PERIPH_BASE             /* 总线基地址 */
#define APB2PERIPH_BASE (PERIPH_BASE + 0x00010000UL)
#define AHBPERIPH_BASE (PERIPH_BASE + 0x00020000UL)

#define TIM1_BASE (APB2PERIPH_BASE + 0x00002C00UL)
#define TIM2_BASE (APB1PERIPH_BASE + 0x00000000UL)
#define TIM3_BASE (APB1PERIPH_BASE + 0x00000400UL)
#define TIM4_BASE (APB1PERIPH_BASE + 0x00000800UL)
```

```
#define TIM5_BASE (APB1PERIPH_BASE + 0x00000C00UL)
#define TIM6_BASE (APB1PERIPH_BASE + 0x00001000UL)
#define TIM7_BASE (APB1PERIPH_BASE + 0x00001400UL)
#define TIM8_BASE (APB2PERIPH_BASE + 0x00003400UL)

#define TIM1 ((TIM_TypeDef *)TIM1_BASE)
#define TIM2 ((TIM_TypeDef *)TIM2_BASE)
#define TIM3 ((TIM_TypeDef *)TIM3_BASE)
#define TIM4 ((TIM_TypeDef *)TIM4_BASE)
#define TIM5 ((TIM_TypeDef *)TIM5_BASE)
#define TIM6 ((TIM_TypeDef *)TIM6_BASE)
#define TIM7 ((TIM_TypeDef *)TIM7_BASE)
#define TIM8 ((TIM_TypeDef *)TIM8_BASE)
```

下面以 TIM6 为例说明这些宏定义的作用，对 TIM6 结构体变量的操作就等于对 TIM6 寄存器的操作，如访问 TIM6 的 CR1 寄存器可以这样操作：TIM6->CR1 &= ~(1<<0)(实现关闭 TIM6 的计数功能)。

从定时器基地址的宏定义可以看出 TIM1 和 TIM8 挂接在 APB2 总线上，其他定时器则挂接在 APB1 总线上。

5.3.2 定时器句柄结构体 TIM_HandleTypeDef

HAL 库在 TIM_TypeDef 的基础上封装了一个结构体数据类型 TIM_HandleTypeDef，该结构体也可以称为定时器的句柄(handle)，定义如下：

```
typedef struct
{
    TIM_TypeDef              * Instance;  /* 寄存器基地址 */
    TIM_Base_InitTypeDef       Init;      /* TIM 时基初始化所需参数 */
    HAL_TIM_ActiveChannel      Channel;   /* 活动通道 */
    DMA_HandleTypeDef          * hdma[7]; /* DMA 处理器数组 */

    HAL_LockTypeDef            Lock;      /* 锁定对象 */
    __IO HAL_TIM_StateTypeDef  State;     /* TIM 的运行状态 */

#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
    …                                     /* 回调函数,略 */
#endif                                    /* USE_HAL_TIM_REGISTER_CALLBACKS */
} TIM_HandleTypeDef;
```

这里重点介绍一下该结构体的前四个成员，其他参数主要是 HAL 库内部使用的。

(1) 成员 Instance 是定时器寄存器的实例化指针，方便操作寄存器，比如使能计数器可以这样操作：SET_BIT(htim6->Instance->CR1, TIM_CR1_CEN)。

(2) 成员 Init 是用户接触最多的，用于配置定时器的基本参数，它的结构体数据类型定义如下：

```
typedef struct
{
```

```
    uint32_t Prescaler;      /* 设置定时器预分频器值,范围是 0x0000 到 0xFFFF */
    uint32_t CounterMode;    /* 计数模式:向上计数模式、向下计数模式和中心对齐模式 */
    uint32_t Period;         /* 设置定时器周期,范围是 0x0000 到 0xFFFF */
    /* 设置定时器时钟频率 CK_INT 与数字滤波器所使用的采样时钟之间的预分频系数 */
    uint32_t ClockDivision;
    uint32_t RepetitionCounter;     /* 设置重复计数器寄存器的值,用在高级定时器中 */
    /* 设置自动重装载寄存器是更新事件产生时写入有效,还是立即写入有效 */
    uint32_t AutoReloadPreload;
} TIM_Base_InitTypeDef;
```

(3) 成员 Channel 用来设置活跃通道,取值范围为 HAL_TIM_ACTIVE_CHANNEL_1～HAL_TIM_ACTIVE_CHANNEL_4。

(4) 成员 hdma 用来关联 DMA,用于定时器的 DMA 功能。

5.3.3　TIM 相关 HAL 库函数

TIM 的操作函数及宏定义在文件 stm32f1xx_hal_tim.c 和 stm32f1xx_hal_tim.h 中。

1. 函数 HAL_TIM_Base_Init

函数 HAL_TIM_Base_Init 说明如表 5-2 所示。

表 5-2　函数 HAL_TIM_Base_Init 说明

函数原型	HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef * htim)
功能描述	根据 htim 中设定的参数初始化 TIM 的时基单元,并初始化关联的句柄
输入参数	htim：时基句柄指针
返回值	HAL 状态
应用示例	TIM_HandleTypeDef　htim6;　　/* 定时器句柄 */ htim6.Instance = TIM6;　　/* 通用定时器 6 */ htim6.Init.Prescaler = 72 - 1;　　/* 预分频系数 */ htim6.Init.CounterMode = TIM_COUNTERMODE_UP;　　/* 向上计数 */ htim6.Init.Period = 1000 - 1;　　/* 自动重装载值 */ /* 自动重装载寄存器是更新事件产生时写入有效 */ htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE; HAL_TIM_Base_Init(&htim6);

2. 函数 HAL_TIM_Base_MspInit

函数 HAL_TIM_Base_MspInit 说明如表 5-3 所示。

表 5-3　函数 HAL_TIM_Base_MspInit 说明

函数原型	void HAL_TIM_Base_MspInit(TIM_HandleTypeDef * htim_base)
功能描述	定时器时基 Msp 初始化,配置与 MCU 有关的时钟使能以及中断优先级
输入参数	htim：TIM 句柄
应用示例	HAL_TIM_Base_MspInit(&htim6);

3. 函数 HAL_TIM_Base_Start_IT

函数 HAL_TIM_Base_Start_IT 说明如表 5-4 所示。

表 5-4 函数 HAL_TIM_Base_Start_IT 说明

函数原型	HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef * htim)
功能描述	使能定时器更新中断(TIM_IT_UPDATE)和使能定时器
输入参数	htim：TIM 句柄
返回值	HAL 状态
应用示例	HAL_TIM_Base_Start_IT(&htim6)；

4. 函数 HAL_TIMEx_MasterConfigSynchronization

函数 HAL_TIMEx_MasterConfigSynchronization 说明如表 5-5 所示。

表 5-5 函数 HAL_TIMEx_MasterConfigSynchronization 说明

函数原型	HAL_StatusTypeDef HAL_TIMEx_MasterConfigSynchronization(TIM_HandleTypeDef * htim，TIM_MasterConfigTypeDef * sMasterConfig)
功能描述	配置 TIM 在主模式下工作
输入参数	htim：TIM 句柄
	sMasterConfig：指向 TIM_MasterConfigTypeDef 结构的指针，包含所选触发器输出 TRGO 和主/从模式
返回值	HAL 状态
应用示例	TIM_MasterConfigTypeDef sMasterConfig = {0}； / * TIMx_EGR 寄存器中的 UG 位用作触发输出 * / sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET； / * 禁止主/从模式 * / sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE； HAL_TIMEx_MasterConfigSynchronization(&htim6，&sMasterConfig)；

5.4 基本定时器应用示例

本示例使用定时器中断方式实现 LED1 每 1000ms 闪烁一次，以延时的方式实现 LED8 每 1000ms 闪烁一次。

5.4.1 STM32CubeMX 工程配置

定时器的时钟 T_{CLK}(即内部时钟 CK_INT)是 HCLK 经过 APB1 预分频器后提供的，如果 APB1 预分频系数为 1，则频率不变，否则频率乘以 2。如图 5-5 所示，APB1 的预分频系数是/2，所以定时器时钟 $T_{CLK}=36MHz\times2=72MHz$。

如图 5-6 所示，在 Timers 中选择基本定时器 TIM6，勾选激活(Activated)复选框，在 Parameter Settings 中配置 TIM6 的参数。时钟预分频系数(Prescaler)设置为 72－1，计数

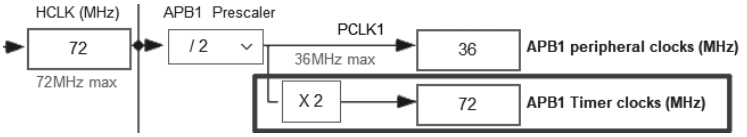

图 5-5　TIM6 时钟配置

模式(Counter Mode)为向上计数(Up),自动重装载寄存器(ARR)设置为 1000－1,根据定时器的溢出时间公式,定时周期 $T_{out}=((71+1)\times(999+1))/72MHz=1ms$。使能自动重装载,禁止触发输出。

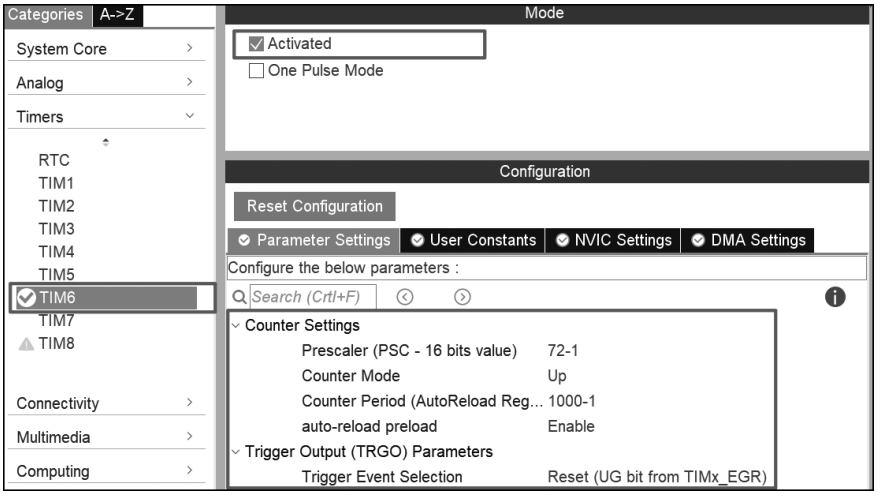

图 5-6　TIM6 参数配置

如图 5-7 所示,配置 NVIC,使能 TIM6 全局中断,抢占优先级(Preemption Priority)和响应优先级(Sub Priority)均采用默认值 0。

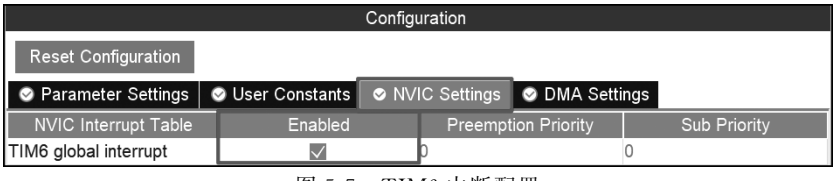

图 5-7　TIM6 中断配置

5.4.2　定时器配置与中断服务函数

1. 定时器配置函数

函数 MX_TIM6_Init 初始化 TIM6 的基本参数,配置定时时间为 1ms,代码如下:

```
TIM_HandleTypeDef htim6;

static void MX_TIM6_Init(void)
{
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    htim6.Instance = TIM6;                                    /* 通用定时器 6 */
```

```
    htim6.Init.Prescaler = 72 - 1;                    /* 预分频系数 */
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;      /* 向上计数器 */
    htim6.Init.Period = 1000 - 1;                     /* 定时器计数周期值 */
    /* 使能自动重装载 */
    htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
    {
        Error_Handler();
    }

    /* 主/从模式配置 */
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    /* 禁止主/从模式 */
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim6,
        &sMasterConfig) != HAL_OK)
    {
        Error_Handler();
    }
}
```

文件 stm32f1xx_hal_msp.c 中，函数 HAL_TIM_Base_MspInit 实现了 TIM6 的时钟使能、中断优先级和中断使能配置，代码如下：

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef * htim_base)
{
    if (htim_base->Instance == TIM6)
    {
        __HAL_RCC_TIM6_CLK_ENABLE();                  /* 使能 TIM6 时钟 */

        HAL_NVIC_SetPriority(TIM6_IRQn, 0, 0);       /* 设置 TIM6 中断优先级 */
        HAL_NVIC_EnableIRQ(TIM6_IRQn);                /* TIM6 中断使能 */
    }
}
```

函数 HAL_TIM_Base_MspInit 名称中的 Msp 是 MCU specific package(MCU 的具体方案)的缩写，理解 Msp 函数的运行逻辑是搞懂 HAL 库的关键，下面来分析一下该函数的运行逻辑。

HAL 库初始化 TIM6 的流程：MX_TIM6_Init()→HAL_TIM_Base_Init()→HAL_TIM_Base_MspInit()。库文件 stm32f1xx_hal_tim.c 中默认的 HAL_TIM_Base_MspInit 是一个弱函数，没有任何实际的控制逻辑，因此函数 HAL_TIM_Base_Init 优先调用在文件 stm32f1xx_hal_msp.c 中重定义的函数 HAL_TIM_Base_MspInit。MX_TIM6_Init 初始化与 MCU 无关的 TIM6 配置，HAL_TIM_Base_MspInit 初始化与 MCU 相关的 TIM6 配置，这样做的好处是，当移植代码到其他 MCU 平台时，只需要修改函数 HAL_TIM_Base_MspInit 中的内容，而不需要修改函数 MX_TIM6_Init 中的内容。可见，相对于标准库，因为有 Msp 函数，HAL 库具备非常强的代码移植性。

与之对应，复位函数 HAL_TIM_Base_MspDeIni 实现了与函数 HAL_TIM_Base_MspInit 相反的操作，用于重置 TIM，它在函数 HAL_TIM_Base_DeInit 中被调用，也是个弱函数。

2. TIM 中断服务函数

在文件 stm32f1xx_it.c 中生成了 TIM6 的中断服务函数 TIM6_IRQHandler，它调用了定时器公用中断处理函数 HAL_TIM_IRQHandler，代码如下：

```
void TIM6_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim6);
}
```

HAL 库函数 HAL_TIM_IRQHandler 封装了定时器的中断处理过程：通过判断中断标志位确定中断来源（更新中断）后，会自动清除该中断标志位，然后调用定时器更新中断回调函数 HAL_TIM_PeriodElapsedCallback，HAL 库在文件 stm32f1xx_hal_tim.c 中定义了该弱回调函数，代码如下：

```
__weak void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef * htim)
{
    UNUSED(htim);                                    /* 防报错的定义 */
    /* 这个函数不应该被改变，如果需要使用回调函数，请重新在用户文件中实现该函数 */
}
```

为了实现用户功能，需要重定义该回调函数，这样做的好处是不需要去理会中断服务函数怎么实现，只需要重写这个回调函数并添加用户功能代码即可。更便利的是，当同时有多个定时器中断时，HAL 库自动将它们的服务函数规整到一起并调用同一个回调函数，也就是无论几个中断，只需要重写一个回调函数并判断传入的定时器编号即可。

5.4.3　用户代码

在文件 main.c 中重定义回调函数 HAL_TIM_PeriodElapsedCallback，实现每 1s 将 LED1 的状态翻转一次，定时时间为 1ms，定时器中断 1000 次的时间才是 1s，具体的代码如下：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef * htim)
{
    static uint32_t time = 0;

    if (htim->Instance == TIM6)                              /* 定时器 6 */
    {
        time++;                                              /* 每 1ms 累加 1 次 */
        if (time == 1000)                                    /* 1s 时间到 */
        {
            HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);    /* 翻转 LED1 */
            time = 0;
        }
    }
}
```