STM32Cube高效开发教程（高级篇）

第10章 软件定时器

王维波
中国石油大学（华东）控制科学与工程学院

STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

如果有读者需要本书课件的PPT版本用于备课，可以给作者发邮件免费获取，并可加入专门的教学和技术交流QQ群

邮箱：wangwb@upc. edu. cn

第10章 软件定时器

10.1 软件定时器概述

10.2 软件定时器相关函数

10.3 软件定时器使用示例

10.1 软件定时器概述

10.1.1 软件定时器的特性

10.1.2 软件定时器相关配置

10.1.3 时间服务任务的优先级

10.1.1 软件定时器的特性

软件定时器（software timer）是FreeRTOS中的一种对象。

FreeRTOS中的软件定时器不直接使用任何硬件定时器或计数器，它依赖于系统中的时间服务任务（timer service task），或称为守护任务（daemon task）来工作。

软件定时器有一个定时周期，还有一个回调函数。根据回调函数执行的频度，有两种类型的软件定时器。

- 单次定时器（one-shot timer），回调函数被执行一次后，定时器就停止工作。
- 周期定时器（periodic timer），回调函数会被循环执行，定时器一直工作。

定时器被创建后，有休眠和运行两种状态。

（1）休眠（dormant）状态

处于休眠状态的定时器不会执行其回调函数，但是可以使用其句柄对其进行操作，例如设置周期。

定时器在以下几种情况下处于休眠状态：

- 定时器被创建后就处于休眠状态
- 单次定时器执行一次回调函数后进入休眠状态
- 定时器使用函数xTimerStop()停止后进入休眠状态

（2）运行（Running）状态

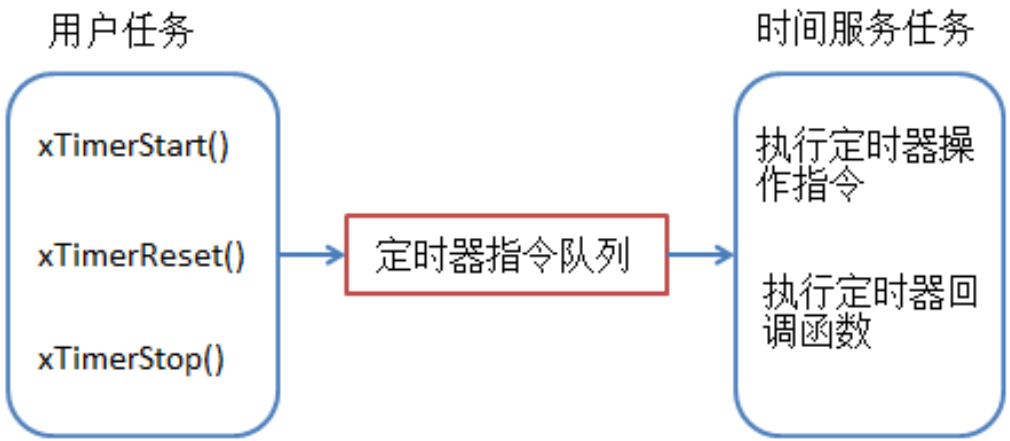
处于运行状态的定时器在流逝的时间达到其定时周期时就会执行其回调函数，不管是单次定时器，还是周期定时器。

定时器在以下几种状态下处于运行状态：

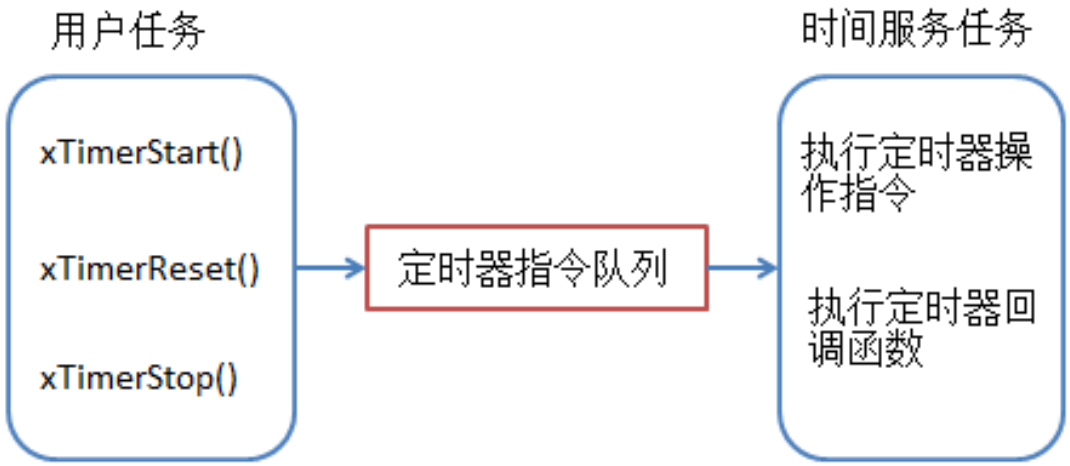
- 使用函数xTimerStart()启动后，定时器进入运行状态
- 定时器在运行状态时，被执行函数xTimerReset()复位起始时间后依然处于运行状态

软件定时器的各种操作实际上是在系统的时间服务任务里完成的。时间服务任务是FreeRTOS自动创建的一个任务。

在用户任务里执行的各种指令都是通过一个队列发送给时间服务任务的，这个队列称为定时器指令队列。时间服务任务读取定时器指令队列里的指令，然后执行相应的操作。



时间服务任务和定时器指令队列是FreeRTOS自动创建的，其操作都是内核实现的。



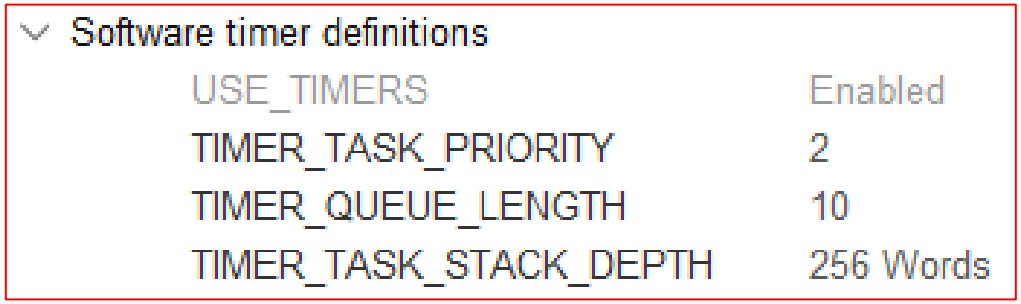
时间服务任务还在定时时间到达时执行定时器回调函数。

由于FreeRTOS里的延时功能是由时间服务函数实现的，所以在定时器的回调函数里不能出现使系统进入阻塞状态的函数，如vTaskDelay()、vTaskDelayUntil()等。

回调函数里可以调用等待信号量、事件组等对象的函数，但是等待的节拍数必须设置为0。

10.1.2 软件定时器相关配置

Software timer definitions组参数，默认设置如图所示



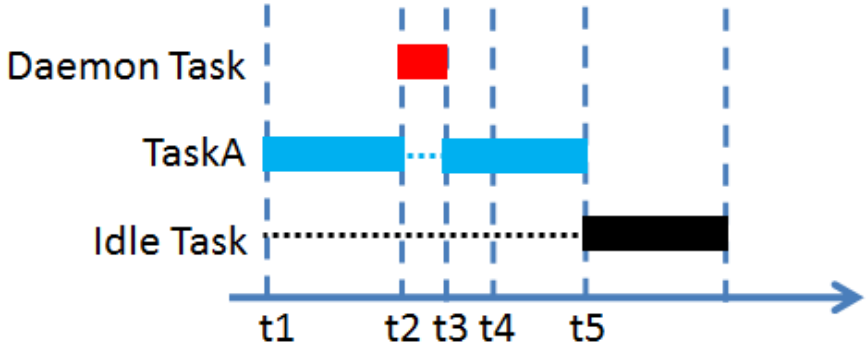
- USE_TIMERS，是否使用软件定时器，默认是Enabled，且不可修改。
- TIMER_TASK_PRIORITY，时间服务任务的优先级，默认值是2，比空闲任务的优先级高
- TIMER_QUEUE_LENGTH，定时器指令队列的长度
- TIMER_TASK_STACK_DEPTH，时间服务任务的栈空间大小，默认值256个字

10.1.3 时间服务任务的优先级

时间服务任务执行定时器指令队列中的定时器操作指令，或定时器的回调函数。时间服务任务的优先级由参数configTIMER_TASK_PRIORITY设定，至少要高于空闲任务的优先级，默认值为2。

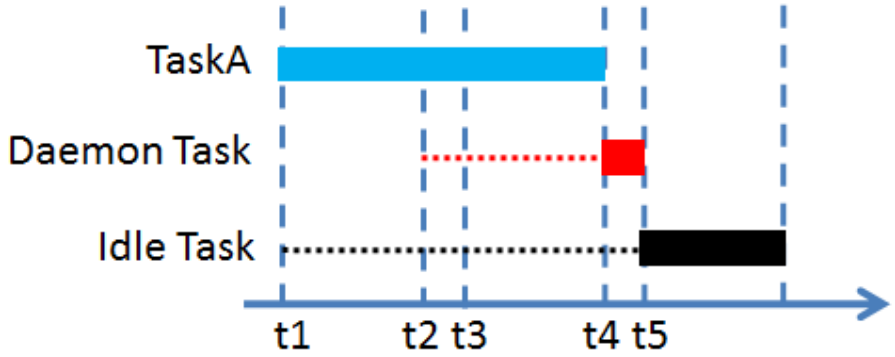
使用定时器的用户任务的优先级可能高于或低于时间服务任务的优先级，那么时间服务任务执行定时器操作指令的时机是不同的。

假设系统中只有一个用户任务TaskA操作定时器，其优先级低于时间服务任务（Daemon Task）的优先级，在任务TaskA中执行一个xTimerStart()指令的时序如图所示

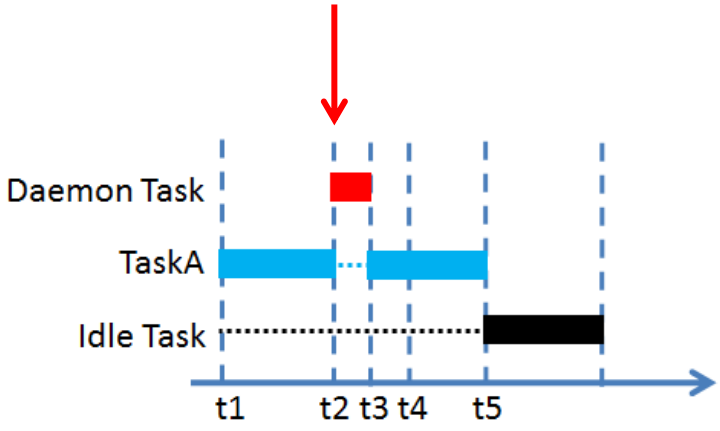


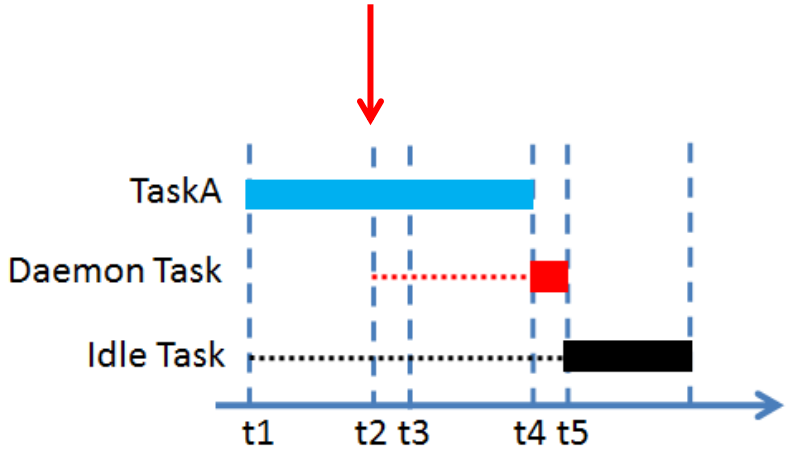
- 时刻t2，任务TaskA调用函数xTimerStart()，实际上是向定时器指令队列写入指令，这会使时间服务任务退出阻塞状态。因为其优先级高于用户任务TaskA，它会抢占执行
- 时刻t3，时间服务任务处理完TaskA发送到队列中的定时器操作指令后重新进入阻塞状态

如果用户任务TaskA的优先级高于时间服务任务



- t2时刻，任务TaskA调用函数xTimerStart()，向定时器指令队列发送指令。由于TaskA的优先级高于时间服务任务，时间服务任务接收到队列指令后也不能抢占CPU进入运行状态，只能进入就绪状态
- t4时刻，任务TaskA进入阻塞状态，时间服务任务进入运行状态，处理定时器指令队列里的指令
- t5时刻，时间服务任务处理完指令后进入阻塞状态





从两种情况可以看到，时间服务任务处理定时器指令队列中的指令的时机是不同的。

但是，不管是哪种情况，定时器的起始时间是从发送“启动定时器”指令到队列开始计算的，也就是调用xTimerStart()函数或xTimerReset()函数的的时刻开始计算，而不是从时间服务任务执行相应指令的时刻开始计算。例如，在右图中，定时器的启动时刻是t2，而不是t4。

10.2 软件定时器相关函数

10.2.1 相关函数概述

10.2.2 部分函数详解

10.2.1 相关函数概述

定时器相关的函数在文件timers.h和timers.c中定义和实现，在用户任务程序中可以调用的常用函数见下表。

分组	函数	功能
创建和删除	xTimerCreate()	创建一个定时器，动态分配内存
	xTimerCreateStatic()	创建一个定时器，静态分配内存
	xTimerDelete()	删除一个定时器
启动、停止和复位	xTimerStart()	启动一个定时器
	xTimerStartFromISR()	xTimerStart()的ISR版本
	xTimerStop()	停止一个定时器
	xTimerStopFromISR()	xTimerStop()的ISR版本
	xTimerReset()	复位一个定时器，重新设置定时器的起始时间
	xTimerResetFromISR()	xTimerReset()的ISR版本

定时器常用函数（续表）

分组	函数	功能
查询和设置参数	pcTimerGetName()	返回定时器的字符串名称
	vTimerSetTimerID()	设置定时器ID
	pvTimerGetTimerID()	获取定时器ID
	xTimerChangePeriod()	设置定时器周期，周期用节拍数表示
	xTimerChangePeriodFromISR()	xTimerChangePeriod()的ISr版本
	xTimerGetPeriod()	返回定时器的定时周期，单位是节拍数
	xTimerIsTimerActive()	查询一个定时器是否处于活动状态
	xTimerGetExpiryTime()	返回定时器还需多少个节拍数就到期

10.2.2 部分函数详解

1. 创建定时器

函数xTimerCreate()以动态分配内存方式创建定时器

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,
                           const TickType_t xTimerPeriodInTicks,
                           const UBaseType_t uxAutoReload,
                           void * const pvTimerID,
                           TimerCallbackFunction_t pxCallbackFunction );
```

- pcTimerName，是定时器的字串名称。
- xTimerPeriodInTicks，是定时器周期，用节拍数表示，可以使用函数pdMS_TO_TICKS()将毫秒时间转换为节拍数。
- uxAutoReload，定时器的类型，pdTRUE表示周期定时器，pdFALSE表示单次定时器。定时器的类型在创建后无法更改。

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,
                           const TickType_t xTimerPeriodInTicks,
                           const UBaseType_t uxAutoReload,
                           void * const pvTimerID,
                           TimerCallbackFunction_t pxCallbackFunction );
```

- pvTimerID，为定时器设置的一个ID。当一个回调函数被多个定时器使用时，可以通过这个ID来区分是哪个定时器。
- pxCallbackFunction，是回调函数的名称。类型TimerCallbackFunction_t是回调函数类型指针，它的定义是

```
typedef void (*TimerCallbackFunction_t)( TimerHandle_t xTimer );
```

所以，回调函数有固定的输入参数定义，传递的参数xTimer就是定时器的句柄。

2. 设置定时周期

在创建定时器后，可以使用函数xTimerChangePeriod()在休眠状态或运行状态下修改定时器的定时周期

```
#define xTimerChangePeriod( xTimer, xNewPeriod, xTicksToWait )
        xTimerGenericCommand( ( xTimer ), tmrCOMMAND_CHANGE_PERIOD,
    ( xNewPeriod ), NULL, ( xTicksToWait ) )
```

- xTimer，是定时器的句柄
- xNewPeriod，是设置的新的定时周期（节拍数）
- xTicksToWait，是发到定时器指令队列时等待的节拍数

函数的返回值为pdTRUE或pdFALSE。pdFALSE表示在等待超时后指令还没有发送给定时器指令队列，pdTRUE表示指令成功发送到了定时器指令队列。

xTimerChangePeriod()调用了xTimerGenericCommand()，这是向定时器指令队列发送指令的通用函数，xTimerStart()、xTimerStop()等函数也是调用此此函，只是传递的参数不同

```
BaseType_t xTimerGenericCommand( TimerHandle_t xTimer, const
    BaseType_t xCommandID, const TickType_t xOptionalValue, BaseType_t *
    const pxHigherPriorityTaskWoken, const TickType_t xTicksToWait );
```

- xTimer，是所操作的定时器的句柄。
- xCommandID，是执行的定时器操作指令ID，是一些宏定义常数，部分指令ID定义如下

```
#define tmrCOMMAND_START                ( ( BaseType_t ) 1 )
#define tmrCOMMAND_RESET                ( ( BaseType_t ) 2 )
#define tmrCOMMAND_STOP                 ( ( BaseType_t ) 3 )
#define tmrCOMMAND_CHANGE_PERIOD        ( ( BaseType_t ) 4 )
#define tmrCOMMAND_DELETE               ( ( BaseType_t ) 5 )
```

```
BaseType_t xTimerGenericCommand( TimerHandle_t xTimer, const
    BaseType_t xCommandID, const TickType_t xOptionalValue, BaseType_t *
    const pxHigherPriorityTaskWoken, const TickType_t xTicksToWait );
```

- xOptionalValue，指令的参数，例如xTimerChangePeriod()调用xTimerGenericCommand()时传递xNewPeriod作为指令的参数
- pxHigherPriorityTaskWoken，是一个BaseType_t * 类型的指针类型变量，是一个返回数据，表示执行完函数后是否需要进行上下文切换，这个参数在ISR版本的函数里用到
- xTicksToWait，是执行函数需要等待的节拍数，也就是向队列写入数据时等待的节拍数

3. 查询定时器定时周期

函数xTimerGetPeriod()返回定时器的定时周期，返回值是用节拍数表示的定时器定时周期。

```
TickType_t xTimerGetPeriod( TimerHandle_t xTimer );
```

4. 查询定时器是否处于运行状态

```
BaseType_t xTimerIsTimerActive( TimerHandle_t xTimer );
```

函数若返回pdTRUE表示定时器处于运行状态，否则就是处于休眠状态。

5. 启动定时器

启动定时器使用函数xTimerStart()或xTimerStartFromISR()，启动定时器也是发送指令到定时器指令队列。

```
#define xTimerStart( xTimer, xTicksToWait )
        xTimerGenericCommand( ( xTimer ), tmrCOMMAND_START,
    ( xTaskGetTickCount() ), NULL, ( xTicksToWait ) )
```

定时器的起始时间是从指令正确发送到指令队列时刻开始计算的，而不是从时间服务函数执行指令时刻开始计算的。

6. 停止定时器

执行函数xTimerStop()可以使处于运行状态的定时器停止，进入休眠状态。其定义如下：

```
#define xTimerStop( xTimer, xTicksToWait )
    xTimerGenericCommand( ( xTimer ), tmrCOMMAND_STOP, 0U, NULL,
( xTicksToWait ) )
```

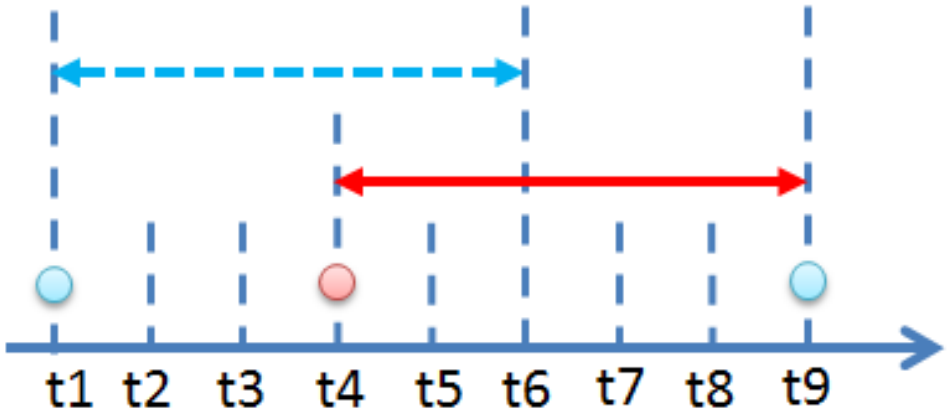
7. 复位定时器

- 若定时器处于休眠状态，xTimerReset()的与xTimerStart()的作用完全相同
- 若定时器处于运行状态，则xTimerReset()会重置定时器的起始时刻为当前时刻

函数xTimerReset()的定义

```
#define xTimerReset( xTimer, xTicksToWait )
            xTimerGenericCommand( ( xTimer ), tmrCOMMAND_RESET,
    ( xTaskGetTickCount() ), NULL, ( xTicksToWait ) )
```

对一个定时周期为5的定时器使用xTimerStart()和xTimerReset()函数的运行时序如图所示。



- 在t1时刻启动了定时器，定时器的预期过期时刻在t6
- 在t4时刻对定时器复位，定时器的过期时刻重新计算，变为t9
- 在t9时刻定时时间到，执行定时器的回调函数

10.3 软件定时器使用示例

10.3.1 示例功能与CubeMX项目设置

10.3.2 程序功能实现

10.3.1 示例功能和CubeMX项目设置

本节的示例Demo10_1SoftTimer演示软件定时器的使用：

- 创建一个周期定时器Timer_Periodic，定时周期为1秒，使LED1闪烁
- 创建一个单次定时器Timer_Once，定时周期为5秒，到期时使LED2熄灭
- 创建一个任务，在任务中检测KeyRight键，按下时使Timer_Once复位

（1）LED和按键的GPIO设置

本示例中用到LED1、LED2和KeyRight，与两个LED连接的引脚初始输出设置为低电平，以使两个LED点亮。

Pin Name	GPIO mode	GPIO Pull-up/Pull-down	GPIO output level	User Label
PE2	Input mode	Pull-up	n/a	KeyRight
PF9	Output Push Pull	No pull-up and no pull-down	Low	LED1
PF10	Output Push Pull	No pull-up and no pull-down	Low	LED2

（2）FreeRTOS的设置

创建1个任务Task_Main

Tasks

Task Name	Entry Function	Priority	Stack Size (Words)	Allocation
Task_Main	AppTask_Main	osPriorityNormal	128	Dynamic

设计两个定时器，设计好的定时器如图所示

Timers

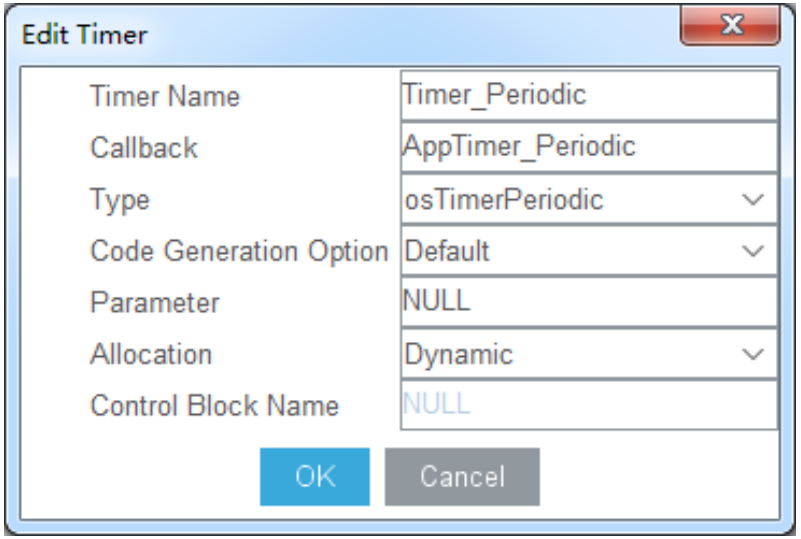
Timer Name	Callback	Type	Allocation	Code Ge...	Parameter	Control Block ...
Timer_Periodic	AppTimer_Periodic	osTimerPeriodic	Dynamic	Default	NULL	NULL
Timer_Once	AppTimer_Once	osTimerOnce	Dynamic	Default	NULL	NULL

Add Delete

定时器的主要参数是回调函数名称和定时器类型，定时器类型有两种：

- osTimeOnce表示单次定时器
- osTimerPeriodic表示周期定时器

如果以静态分配内存方式创建定时器，还需要设置控制块名称。



10.3.2 程序功能实现

1.主程序

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();                    //LED, KeyRight引脚GPIO初始化
  MX_FSMC_Init();

  /* USER CODE BEGIN 2 */
  TFTLCD_Init();                     //LCD 初始化
  LCD_ShowString(10, 10, (uint8_t *)"Demo9_1:Soft Timer");
  LCD_ShowString(10, 30, (uint8_t *)"LED1 toggle each 1sec");
  LCD_ShowString(10, 50, (uint8_t *)"LED2 will be off in 5sec");
  LCD_ShowString(10, 70, (uint8_t *)"Press KeyRight to reset");
  LCD_ShowString(10, 90, (uint8_t *)" or restart Timer_Once");
  /* USER CODE END 2 */

  osKernelInitialize();
  MX_FREERTOS_Init();
  osKernelStart();
}
```

2. FreeRTOS对象初始化

函数MX_FREERTOS_Init() 创建在CubeMX中设计的任务和两个定时器。初始代码还包括两个定时器的回调函数代码框架。

```
/* 任务 Task_Main 相关定义*/
osThreadId_t Task_MainHandle;            //任务Task_Main的句柄
const osThreadAttr_t Task_Main_attributes = {      //任务属性
                .name = "Task_Main",
                .priority = (osPriority_t) osPriorityNormal,
                .stack_size = 128 * 4
};

/* 定时器 Timer_Periodic 相关定义*/
osTimerId_t Timer_PeriodicHandle;        //定时器Timer_Periodic的句柄
const osTimerAttr_t Timer_Periodic_attributes = {   //定时器属性
                .name = "Timer_Periodic"
};

/* 定时器Timer_Once 相关定义*/
osTimerId_t Timer_OnceHandle;            //定时器Timer_Once的句柄
const osTimerAttr_t Timer_Once_attributes = {      //定时器属性
                .name = "Timer_Once"
};
```

函数MX_FREERTOS_Init() 创建任务和两个定时器。

```
void MX_FREERTOS_Init(void)
{
    /* 创建定时器 Timer_Periodic */
    Timer_PeriodicHandle = osTimerNew (AppTimer_Periodic, osTimerPeriodic,
                                       NULL, &Timer_Periodic_attributes);

    /* 创建定时器 Timer_Once */
    Timer_OnceHandle = osTimerNew (AppTimer_Once, osTimerOnce,
                                   NULL, &Timer_Once_attributes);

    /* 创建任务Task_Main */
    Task_MainHandle = osThreadNew(AppTask_Main, NULL,
    &Task_Main_attributes);
}
```

一个osTimerAttr_t类型的结构体变量定义定时器的属性

```
typedef struct {
      const char          *name;      //定时器的字符串名称
      uint32_t        attr_bits;      //属性位
      void            *cb_mem;        //控制块存储空间
      uint32_t        cb_size;        //控制块大小
} osTimerAttr_t;
```

创建定时器使用了CMSIS 的标准接口函数osTimerNew()

```
osTimerId_t osTimerNew (osTimerFunc_t func, osTimerType_t type, void
      *argument, const osTimerAttr_t *attr);
```

osTimerNew()创建的定时器的周期自动设置为1，所以还需要调用函数xTimerChangePeriod()设置定时器的周期。

创建后的定时器处于休眠状态，需要用xTimerStart()函数启动定时器。

3. 任务和定时器的功能代码

任务Task_Main的代码在进入无限for循环之前设置了两个定时器的周期，并启动了两个定时器。

```
uint32_t  counter=0;    //计数变量

void AppTask_Main(void *argument)   //任务Task_Main的任务函数
{
        xTimerChangePeriod( Timer_PeriodicHandle, pdMS_TO_TICKS(1000), portMAX_DELAY);
        xTimerChangePeriod( Timer_OnceHandle, pdMS_TO_TICKS(5000), portMAX_DELAY);
        xTimerStart( Timer_PeriodicHandle,portMAX_DELAY);
        xTimerStart( Timer_OnceHandle,portMAX_DELAY);

        for(;;)
        {
          GPIO_PinState keyState=HAL_GPIO_ReadPin(GPIOE, GPIO_PIN_2); //PE2=KeyRight
          if (keyState==GPIO_PIN_RESET)  //KeyRight按下
          {
             counter=0;    //计数变量清零
             if (xTimerIsTimerActive(Timer_OnceHandle)==pdFALSE)  //休眠状态
                 HAL_GPIO_WritePin(GPIOF, GPIO_PIN_10, GPIO_PIN_RESET);  //使LED2亮
             xTimerReset(Timer_OnceHandle,portMAX_DELAY); //定时器Timer_Once复位
          }
          vTaskDelay(10);
        }
}
```

Timer_Periodic是周期定时器，它使计数值counter加1，使LED1闪烁。

Timer_Once是单次定时器，到期后使LED2熄灭。

```
/* 定时器Timer_Periodic 回调函数 */
void AppTimer_Periodic( void *argument)
{
    LED1_Toggle();      //LED1闪烁
    counter++;          //计数值加1
    LCD_ShowUintX(50,200,counter,4);
}

/* 定时器Timer_Once 回调函数 */
void AppTimer_Once( void *argument)
{
    LED2_Toggle();
}
```

程序编译后下载到开发板上运行测试，运行时会发现：

- 系统复位后，LED1闪烁，LCD上显示计数秒数。如果不按下KeyRight键，计数到5之后LED2熄灭，表示单次定时器Timer_Once的回调函数被执行。
- 如果在LED2熄灭之前按下KeyRight键，会从0开始重新计数到5之后LED2才熄灭，这说明xTimerReset()的工作特点。
- 如果LED2已经熄灭了，按KeyRight键，LED2会重新点亮，计数5秒后又熄灭。说明定时器处于休眠状态时，函数xTimerReset()的功能与xTimerStart()一样。

练习任务

1. 看教材，练习本章的示例。

2. 使用软件定时器，定时周期500ms，在回调函数里通过ADC1的IN5通道采集数据，在LCD上显示。