8.软件定时器

2025年10月15日 22:44

- 一、什么是软件定时器
  - FreeRTOS中的软件定时器与传统的硬件定时器不同，它借助软件的方式实现定时器的效果，FreeRTOS的软件定时器是通过系统级的时间片中断来实现计数的，当计数完成之后，系统自动执行用户定义的注册回调函数，来完成用户的功能
  - 软件定时器是一种基于操作系统“tick”实现的定时机制。它不是硬件定时器，而是由 FreeRTOS 的 **Timer Service Task（定时器守护任务 / daemon task）** 管理的一种内核对象。定时器到期后，会在守护任务上下文中调用用户注册的回调函数。
- 二、软件定时器与硬件定时器的关系
  - 在FreeRTOS中，软件定时器的实现离不开硬件定时器的帮助。FreeRTOS系统在运行的过程中需要一个时间基准,称为“系统节拍（System Tick）”，这个节拍有硬件定时器产生，软件定时器的计数是依靠硬件定时间中的系统节拍计数来实现的。所以我们可以说，软件定时器是依靠硬件定时器实现的。
  - 硬件定时器：产生时基（System Tick）
  - 软件定时器：基于Tick的逻辑封装（依靠硬件定时器）
- 三、软件定时器的关键API函数
  - 1、软件定时器的实现
  - TimerHandle_t xTimerCreate(const char * const pcTimerName, // 名称（调试用）
    const TickType_t xTimerPeriodInTicks, // 周期（Tick）
    const UBaseType_t uxAutoReload, // 是否自动重载（pdTRUE/pdFALSE）
    void * const pvTimerID, // 定时器ID
    TimerCallbackFunction_t pxCallbackFunction // 回调函数)
  - ```
    TimerHandle_t xTimerCreate( const char * const pcTimerName,
                                const TickType_t xTimerPeriodInTicks,
                                const BaseType_t xAutoReload,
                                void * const pvTimerID,
                                TimerCallbackFunction_t pxCallbackFunction )
    ```
  - **参数：**
    - pcTimerName:给定时器一个名称，主要用于调试
    - xTimerPeriodInTicks：软件定时器的周期
    - uxAutoReload：自动重装载标志，给pdTRUE表示时钟周期到期后，自动重新开始（周期定时器），给pdFALSE表示不自动重新装载
    - pvTimerID：软件定时器的标识符，用于回调函数中区分不同的定时器
    - pxCallbackFunction：定时器到期后执行的回调函数指针，指向回调函数
  - **返回值**
    - 创建成功后返回一个软件定时器的句柄
  - **函数底层的代码实现**
  - ```
    TimerHandle_t xTimerCreate( const char * const pcTimerName,
                                const TickType_t xTimerPeriodInTicks,
                                const BaseType_t xAutoReload,
                                void * const pvTimerID,
                                TimerCallbackFunction_t pxCallbackFunction )
    {
        Timer_t * pxNewTimer;

        traceENTER_xTimerCreate( pcTimerName, xTimerPeriodInTicks, xAutoReload, pvTimerID, pxCallbackFunction );

        /* MISRA Ref 11.5.1 [Malloc memory assignment] */
        /* More details at: https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/MISRA.md#rule-115 */
        /* coverity[misra_c_2012_rule_11_5_violation] */
        pxNewTimer = ( Timer_t * ) pvPortMalloc( sizeof( Timer_t ) );

        if( pxNewTimer != NULL )
        {
            /* Status is thus far zero as the timer is not created statically
             * and has not been started.  The auto-reload bit may get set in
             * prvInitialiseNewTimer. */
            pxNewTimer->ucStatus = 0x00;
            prvInitialiseNewTimer( pcTimerName, xTimerPeriodInTicks, xAutoReload, pvTimerID, pxCallbackFunction, pxNewTimer );
        }

        traceRETURN_xTimerCreate( pxNewTimer );

        return pxNewTimer;
    }

    static void prvInitialiseNewTimer( const char * const pcTimerName
    ```
    给定时器结构体Timer_t

```
    return pxNewTimer;
}
```

```
static void prvInitialiseNewTimer( const char * const pcTimerName,
                                   const TickType_t xTimerPeriodInTicks,
                                   const BaseType_t xAutoReload,
                                   void * const pvTimerID,
                                   TimerCallbackFunction_t pxCallbackFunction,
                                   Timer_t * pxNewTimer )
{
    /* 0 is not a valid value for xTimerPeriodInTicks. */
    configASSERT( ( xTimerPeriodInTicks > 0 ) );

    /* Ensure the infrastructure used by the timer service task has been
     * created/initialised. */
    prvCheckForValidListAndQueue();

    /* Initialise the timer structure members using the function
     * parameters. */
    pxNewTimer->pcTimerName = pcTimerName;
    pxNewTimer->xTimerPeriodInTicks = xTimerPeriodInTicks;
    pxNewTimer->pvTimerID = pvTimerID;
    pxNewTimer->pxCallbackFunction = pxCallbackFunction;
    vListInitialiseItem( &( pxNewTimer->xTimerListItem ) );

    if( xAutoReload != pdFALSE )
    {
        pxNewTimer->ucStatus |= ( uint8_t ) tmrSTATUS_IS_AUTORELOAD;
    }

    traceTIMER_CREATE( pxNewTimer );
}
```

-

```
static void prvCheckForValidListAndQueue( void )
{
    /* Check that the list from which active timers are referenced, and the
     * queue used to communicate with the timer service, have been
     * initialised. */
    taskENTER_CRITICAL();
    {
        if( xTimerQueue == NULL )
        {
            vListInitialise( &xActiveTimerList1 );
            vListInitialise( &xActiveTimerList2 );
            pxCurrentTimerList = &xActiveTimerList1;
            pxOverflowTimerList = &xActiveTimerList2;
            #else
            {
                xTimerQueue = xQueueCreate( ( UBaseType_t ) configTIMER_QUEUE_LENGTH, ( UBaseType_t ) sizeof( DaemonTaskMessage_t ) );
            }
            #endif /* if ( configSUPPORT_STATIC_ALLOCATION == 1 ) */
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    taskEXIT_CRITICAL();
}
```

- 我们可以看到已进入函数就是首先定义了一个软件定时器结构体，紧接着就是从栈中申请分配定时器结构体的空间，那么我们加看一下定时器结构体中有什么

```
typedef struct tmrTimerControl
{
    const char * pcTimerName;
    ListItem_t xTimerListItem;
    TickType_t xTimerPeriodInTicks;
    void * pvTimerID;
    portTIMER_CALLBACK_ATTRIBUTE TimerCallbackFunction_t pxCallbackFunction;
    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxTimerNumber;
    #endif
    uint8_t ucStatus;
} xTIMER;

/* The old xTIMER name is maintained above then typedefed to the new Timer_t
name below to enable the use of older kernel aware debuggers. */
typedef xTIMER Timer_t;
```

- 看到定时器结构体，其中包括：
  - 软件定时器的名称--pcTimerName
  - 软件定时器链表节点--xTimerListItem：FreeRTOS内部通过链表来同一管理和组织活跃的软件定时器，定时器加

入到活跃链表中便利了系统检查那些定时器到期

- □ 定时器周期--xTimerPeriodInTicks
- □ 定时器标识--pvTimerID
- □ 定时器到期要执行的回调函数--pxCallbackFunction
- □ 定时器的状态标志--ucStatus：用于标识定时器是处于活跃、休眠还是其他状态

- 为定时器结构体分配完内存之后，紧接着就是调用一个初始化软件定时器的函数prvInitialiseNewTimer（），并将函数的参数传入，我们进入到初始化函数的底层可以看到，一上来就会调用一个检查链表和队列的函数prvCheckForValidListAndQueue（），我们先不看这个函数，我们接着往下看，可以看到将传入的初始化参数全都赋值给定时器结构体中各个参数，然后就是初始化链表的函数，下面根据是否参数，来决定是否需要设置为自动重装载，看完这一些，接下来，我们来深入prvCheckForValidListAndQueue（）函数
- 已进入函数就是调用进入临界区代码，关闭中断，保证系统操作的原子性，然后判断定时器队列是否创建，如果队列未创建，则初始化两个定时器积极链表用来存放创建的软件定时器，然后就是创建定时器队列，但是注意，定时器队列并不是在创建软件定时器的时候创建的，而是在调用vTaskStartScheduler（）开启任务调度器的时候就创建了软件定时器队列

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;

    traceENTER_vTaskStartScheduler();

    #if ( configUSE_CORE_AFFINITY == 1 ) && ( configNUMBER_OF_CORES > 1 )
    {
        /* Sanity check that the UBaseType_t must have greater than or equal to
         * the number of bits as confNUMBER_OF_CORES. */
        configASSERT( ( sizeof( UBaseType_t ) * taskBITS_PER_BYTE ) >= configNUMBER_OF_CORES );
    }
    #endif /* #if ( configUSE_CORE_AFFINITY == 1 ) && ( configNUMBER_OF_CORES > 1 ) */

    xReturn = prvCreateIdleTasks();

    #if ( configUSE_TIMERS == 1 )
    {
        if( xReturn == pdPASS )
        {
            xReturn = xTimerCreateTimerTask();
        }
```

创建空闲Task

```
BaseType_t xTimerCreateTimerTask( void )
{
    BaseType_t xReturn = pdFAIL;

    traceENTER_xTimerCreateTimerTask();

    /* This function is called when the scheduler is started if
     * configUSE_TIMERS is set to 1.  Check that the infrastructure used by the
     * timer service task has been created/initialised.  If timers have already
     * been created then the initialisation will already have been performed. */
    prvCheckForValidListAndQueue();
```

创建软定队列

- 所以总结一下，创建软件定时器函数首先分配了一个定时器结构体，然后初始化定时器结构体，和定时器链表用于存放定时器，但是此时虽然会调用创建定时器队列的函数，但是并不是在此时创建队列的！

○ 2、软件定时器启动函数

○ BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);

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

○ 这是一个宏定义的函数，本质是一个定时器命令产生函数

○ **我们来看复用函数中传入的参数**：

- 1.xTimer:要启动的定时器句柄--这里就是传入一个定时器句柄的参数
- 2.xCommandID：定时器的命令ID，这里传入tmeCOMMAND_START也就是定时器启动命令，处理定时器启动命令外，还有很多其他的定时器操作命令

任务中调用

```
#define tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR    ( ( BaseType_t ) -2 )
#define tmrCOMMAND_EXECUTE_CALLBACK             ( ( BaseType_t ) -1 )
#define tmrCOMMAND_START_DONT_TRACE             ( ( BaseType_t ) 0 )
#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 )
```

ISR调用

```
#define tmrCOMMAND_STOP                         ( ( BaseType_t ) 3 )
#define tmrCOMMAND_CHANGE_PERIOD                ( ( BaseType_t ) 4 )
#define tmrCOMMAND_DELETE                       ( ( BaseType_t ) 5 )

#define tmrFIRST_FROM_ISR_COMMAND               ( ( BaseType_t ) 6 )
#define tmrCOMMAND_START_FROM_ISR               ( ( BaseType_t ) 6 )
#define tmrCOMMAND_RESET_FROM_ISR               ( ( BaseType_t ) 7 )
#define tmrCOMMAND_STOP_FROM_ISR                ( ( BaseType_t ) 8 )
#define tmrCOMMAND_CHANGE_PERIOD_FROM_ISR       ( ( BaseType_t ) 9 )
```

ISR调用

- 这就是定时器操作的全部命令，可见软件定时器不仅可以在任务中操作，还可以在ISR中断服务程序中操作

- 3.xOptionalValue：传递于命令相关的可选数值--这里传入xTaskGetTickCount（）获取当前的时钟计数值
- 4.pxHigherPriorityTaskWoken：高优先级任务唤醒标志，用于中断上下文--这里传入NULL表示不需要
- 5.xTicksToWait：阻塞等待时间--这里传入一个对于参数的等待时间

```
#define xTimerGenericCommand( xTimer, xCommandID, xOptionalValue, pxHigherPriorityTaskWoken, xTicksToWait )          \
    ( ( xCommandID ) < tmrFIRST_FROM_ISR_COMMAND ?                                                                  \
      xTimerGenericCommandFromTask( xTimer, xCommandID, xOptionalValue, pxHigherPriorityTaskWoken, xTicksToWait ) : \
      xTimerGenericCommandFromISR( xTimer, xCommandID, xOptionalValue, pxHigherPriorityTaskWoken, xTicksToWait ) )
```

- 我们可以看到定时器命令产生任务也是一个由宏定义的函数，我们根据定时器命令是在任务中调用还是在中断中调用分为FromTask和FromISR，我们来看一下带FromTask即在任务中调用的函数

```
BaseType_t xTimerGenericCommandFromTask( TimerHandle_t xTimer,
                                         const BaseType_t xCommandID,
                                         const TickType_t xOptionalValue,
                                         BaseType_t * const pxHigherPriorityTaskWoken,
                                         const TickType_t xTicksToWait )
{
    BaseType_t xReturn = pdFAIL;
    DaemonTaskMessage_t xMessage;

    ( void ) pxHigherPriorityTaskWoken;

    traceENTER_xTimerGenericCommandFromTask( xTimer, xCommandID, xOptionalValue, pxHigherPriorityTaskWoken, xTicksToWait );

    configASSERT( xTimer );

    /* Send a message to the timer service task to perform a particular action
     * on a particular timer definition. */
    if( xTimerQueue != NULL )
    {
        /* Send a command to the timer service task to start the xTimer timer. */
        xMessage.xMessageID = xCommandID;
        xMessage.u.xTimerParameters.xMessageValue = xOptionalValue;
        xMessage.u.xTimerParameters.pxTimer = xTimer;

        configASSERT( xCommandID < tmrFIRST_FROM_ISR_COMMAND );

        if( xCommandID < tmrFIRST_FROM_ISR_COMMAND )
        {
            if( xTaskGetSchedulerState() == taskSCHEDULER_RUNNING )
            {
                xReturn = xQueueSendToBack( xTimerQueue, &xMessage, xTicksToWait );
            }
            else
            {
                xReturn = xQueueSendToBack( xTimerQueue, &xMessage, tmrNO_DELAY );
            }
        }

        traceTIMER_COMMAND_SEND( xTimer, xCommandID, xOptionalValue, xReturn );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    traceRETURN_xTimerGenericCommandFromTask( xReturn );

    return xReturn;
}
```

调度器处于运行状态

- 函数一进来就是初始化返回值和传递消息队列中的部分数据结构体，下面我们观察到需要给队列中的数据结构体赋值，所以我们先来观察一下队列数据结构体

- 函数一进来就是初始化返回值和传递消息队列中的部分数据结构体，下面我们观察到需要给队列中的数据结构体赋值，所以我们先来观察一下队列数据结构体

```
typedef struct tmrTimerParameters
{
    TickType_t xMessageValue; /**< An optional value used by a subse
    Timer_t * pxTimer;        /**< The timer to which the command wi
} TimerParameter_t;

The structure that contains the two message types, along with an iden
that is used to determine which message type is valid. */
typedef struct tmrTimerQueueMessage
{
    BaseType_t xMessageID; /**< The command being sent to the timer
    union
    {
        TimerParameter_t xTimerParameters;

        /* Don't include xCallbackParameters if it is not going to b
         * it makes the structure (and therefore the timer queue) la
        #if ( INCLUDE_xTimerPendFunctionCall == 1 )
            CallbackParameters_t xCallbackParameters;
        #endif /* INCLUDE_xTimerPendFunctionCall */
    } u;
} DaemonTaskMessage_t;
```

  - 我们可以看到队列传递的消息结构体中包含了消息ID和一个联合体，其中包含上方定义的定时器参数结构体，说明这部分内存二者可以共用，其中定时器参数结构体中包括了要传递的消息的值和要操作的定时器的结构体，这就是初始化的消息数据结构体，我们继续往下看，然后判断定时器队列是否创建，如果创建了定时器队列，就把数据传入给队列消息结构体中，然后根据传入传入的命令ID是否小于定时器中断服务命令，如果小于中断服务命令，并且此时任务调度器的状态是运行状态时，调用队列发送函数，将命令传入到队列中。
  - 总结一下，软件定时器启动函数，初始化一个队列消息数据结构体，接着根据是否创建消息队列，将参数传递给数据结构体中，如果此时任务调度器处于运行的状态，则调用发送队列API函数，将数据命令传入队列当中，
- 3、重置软件定时器
- BaseType_t xTimerReset(TimerHandle_t xTimer, TickType_t xTicksToWait);

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

  - 这是一个宏定义的函数，本质函数和启动定时器函数复用的同一个函数
  - 这里就是只有第二参数传入的不同，启动函数传入启动命令，这里重置函数应传入重置命令--tmrCOMMAND_RESET
  - 因为其他于启动函数一样，这里就不再过多介绍了
- 4、停止软件定时器

  BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);

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

  - 这是一个宏定义函数，底层调用的函数和前面的函数是一样的
  - 因为是停止定时器，所以这里第二个参数应该个停止命令，因为底层函数相同，所以这里就不再过度叙述了
- 5、剩余的一些API函数
  - 像删除定时器函数：BaseType_t xTimerDelete(TimerHandle_t xTimer, TickType_t xTicksToWait)
    - 此函数底层同样是调用发送命令的函数，命令为删除定时器
  - 像一些在任务中调用的函数，带上FromISR后缀的函数
    - 中断服务程序中的启动定时器：
      - BaseType_t xTimerStartFromISR(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);
    - 中断服务程序中的停止定时器：
      - BaseType_t xTimerStopFromISR(TimerHandle_t xTimer, BaseType_t *pxHigherPriorityTaskWoken);
    - 等等，像这样的函数还有很多
- 上面那些写全都是向队列中发送消息的函数，那么向队列中发送完消息过后，什么时候来接收这些数据呢，又由谁来接收这些队列消息呢，接收到消息过后该如何执行呢，为了解决这些问题，接下来我们来学习一下定时器服务函数（prvTimerTask（）函数）
  - 定时器服务函数（又叫定时器守护任务）
  - 我们在Timer.c的文件中并未找到prvTimerTask（）函数，那么因为这个版本的FreeRTOS已经将其优化了，我们找到了优化过后的函数

```c
static portTASK_FUNCTION( prvTimerTask, pvParameters )
{
    TickType_t xNextExpireTime;
    BaseType_t xListWasEmpty;

    /* Just to avoid compiler warnings. */
    ( void ) pvParameters;

    #endif /* configUSE_DAEMON_TASK_STARTUP_HOOK */

    for( ; configCONTROL_INFINITE_LOOP(); )
    {
        /* Query the timers list to see if it contains any timers, and if
         * obtain the time at which the next timer will expire. */
        xNextExpireTime = prvGetNextExpireTime( &xListWasEmpty );

        /* If a timer has expired, process it.  Otherwise, block this task
         * until either a timer does expire, or a command is received. */
        prvProcessTimerOrBlockTask( xNextExpireTime, xListWasEmpty );

        /* Empty the command queue. */
        prvProcessReceivedCommands();
    }
}
```

- 这个就是定时器的守护任务的循环主体，负责监控定时器的状态，定时器到期事件和执行定时器命令
- 我们可以看到，定时器守护任务（定时器服务任务）里面实现了一个无限循环--for循环，其中给了一个宏configCONTROL_INFINITE_LOOP()来控制循环条件，在这个循环里面我们可以看到，一上来就是获取下一个二定时器的到期时间，通过调用prvGetNextExpireTime（）函数，接下来我们来看一下这个获取下一个定时器到期时间的函数的底层是如何实现的

```c
static TickType_t prvGetNextExpireTime( BaseType_t * const pxListWasEmpty )
{
    TickType_t xNextExpireTime;

    /* Timers are listed in expiry time order, with the head of the list
     * referencing the task that will expire first.  Obtain the time at which
     * the timer with the nearest expiry time will expire.  If there are no
     * active timers then just set the next expire time to 0.  That will cause
     * this task to unblock when the tick count overflows, at which point the
     * timer lists will be switched and the next expiry time can be
     * re-assessed.  */
    *pxListWasEmpty = listLIST_IS_EMPTY( pxCurrentTimerList );

    if( *pxListWasEmpty == pdFALSE )
    {
        xNextExpireTime = listGET_ITEM_VALUE_OF_HEAD_ENTRY( pxCurrentTimerList );
    }
    else
    {
        /* Ensure the task unblocks when the tick count rolls over. */
        xNextExpireTime = ( TickType_t ) 0U;
    }

    return xNextExpireTime;
}
```

- 获取下一个定时器到期时间的函数如图，开始的时候判断当前定时器链表是否为空，如果定时器不为空即有活跃定时器的时候，就调用一个宏函数来获取当前定时器链表头部的第一个定时器的到期时间，因为在定时器链表中是按照到期时间排序的；如果定时器活跃链表为空的话，就返回0；看完获取下一个定时器到期时间的函数之后，我们继续回到定时器守护任务当中，在获取了下一个定时器的到期时间后，紧接着就是调用处理定时器或阻塞任务的函数prvProcessTimerOrBlockTask（）函数，接下来我们来深入观察一下这个定时器处理和任务阻塞函数

```c
static void prvProcessTimerOrBlockTask( const TickType_t xNextExpireTime,
                                        BaseType_t xListWasEmpty )
{
    TickType_t xTimeNow;
    BaseType_t xTimerListsWereSwitched;

    vTaskSuspendAll();
    {
        /* Obtain the time now to make an assessment as to whether the timer
         * has expired or not.  If obtaining the time causes the lists to switch
         * then don't process this timer as any timers that remained in the list
         * when the lists were switched will have been processed within the
         * prvSampleTimeNow() function. */
        xTimeNow = prvSampleTimeNow( &xTimerListsWereSwitched );

        if( xTimerListsWereSwitched == pdFALSE )
        {
            /* The tick count has not overflowed, has the timer expired? */
            if( ( xListWasEmpty == pdFALSE ) && ( xNextExpireTime <= xTimeNow ) )
            {
                ( void ) xTaskResumeAll();
                prvProcessExpiredTimer( xNextExpireTime, xTimeNow );
            }
            else
            {
                /* The tick count has not overflowed, and the next expire
                 * case when the current timer list is empty. */
                if( xListWasEmpty != pdFALSE )
                {
                    /* The current timer list is empty - is the overflow list
                     * also empty? */
                    xListWasEmpty = listLIST_IS_EMPTY( pxOverflowTimerList );
                }

                vQueueWaitForMessageRestricted( xTimerQueue, ( xNextExpireTime - xTimeNow ), xListWasEmpty );

                if( xTaskResumeAll() == pdFALSE )
                {
                    taskYIELD_WITHIN_API();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        else
        {
            ( void ) xTaskResumeAll();
        }
    }
```

- 这个函数是定时器守护任务的只能调度器，根据定时器状态和时间条件，决定是立即处理到期定时器还是任务合理阻塞以节省CPU资源。我们看一下代码的实现，首先就是调用挂起任务调度器的函数，防止函数被其他任务打断，保证了操作的原子性，紧接着函数就调用prvSampleTimeNow（）函数来获取当前系统的时间并检查列表切换（节拍计数器是否发送溢出并在溢出时自动切换定时器列表），还是老规矩，我们来看一下底层代码的实现

```c
static TickType_t prvSampleTimeNow( BaseType_t * const pxTimerListsWereSwitched )
{
    TickType_t xTimeNow;
    PRIVILEGED_DATA static TickType_t xLastTime = ( TickType_t ) 0U;

    xTimeNow = xTaskGetTickCount();

    if( xTimeNow < xLastTime )
    {
        prvSwitchTimerLists();
        *pxTimerListsWereSwitched = pdTRUE;
    }
    else
    {
        *pxTimerListsWereSwitched = pdFALSE;
    }

    xLastTime = xTimeNow;

    return xTimeNow;
}
```

- 我们可以看到底层函数首先初始话了一个上一次计数的时间，然后函数获取了当前的计数时间，紧接着将二者进行判断，然后完成不同的情况，如果当前计数时间小于上一次计数的时间，此时说明计数器发送了溢出，我们需要进行定时器列表的切换，我们调用prvSwitchTimerLists（）来切换定时器列表，并将参数pxTimerListsWereSwirched定时器列表切换标志置为pdTRUE，表示定时器进行了列表切换，否者将参数置为pdFALSE，然后将当前获取的计数时间赋值给上一次获取的计数时间，为下次调用函数做好准备，最后返回当前的时间。回到定时器处理和任务阻塞函数当中，我们根据定时器切换列表标志的值，进行不同的操作
- 如果定时器列表未进行切换，说明计数节拍并未发生溢出，此时又有两种不同的情况
  - 情况一：如果此时定时器链表不为空并且下一个定时器到期时间<=获取的当前时间，此时我们调用恢复任务调度器的函数，开启任务调度，并且调用prvProcessExpiredTimer（）处理到期定时器函数，将下一个定时器到期的时间和当前时间全都传入，我们来看一下，底层处理到期定时器函数的实现

```
static void prvProcessExpiredTimer( const TickType_t xNextExpireTime,
                                    const TickType_t xTimeNow )
{
    /* MISRA Ref 11.5.3 [Void pointer assignment] */
    /* More details at: https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/MISRA.md#rule-115 */
    /* coverity[misra_c_2012_rule_11_5_violation] */
    Timer_t * const pxTimer = ( Timer_t * ) listGET_OWNER_OF_HEAD_ENTRY( pxCurrentTimerList );

    /* Remove the timer from the list of active timers.  A check has already
     * been performed to ensure the list is not empty. */

    ( void ) uxListRemove( &( pxTimer->xTimerListItem ) );

    /* If the timer is an auto-reload timer then calculate the next
     * expiry time and re-insert the timer in the list of active timers. */
    if( ( pxTimer->ucStatus & tmrSTATUS_IS_AUTORELOAD ) != 0U )
    {
        prvReloadTimer( pxTimer, xNextExpireTime, xTimeNow );
    }
    else
    {
        pxTimer->ucStatus &= ( ( uint8_t ) ~tmrSTATUS_IS_ACTIVE );
    }

    /* Call the timer callback. */
    traceTIMER_EXPIRED( pxTimer );
    pxTimer->pxCallbackFunction( ( TimerHandle_t ) pxTimer );
}
```

- 这就是处理到期定时器函数的底层代码，我们可以看到，首先一上来我们就调用宏函数获取到了定时器链表的头部定时器（到期的定时器），并将定时器从链表中移除，此时我们将取出来的定时器观察器定时器结构体中的数据，如果定时器结构体中定时器的状态设置为了自动重装载模式，那么我们将调用prvRelooadTimer（）自动重装载定时器函数，来处理到期的定时器，接下来我们深入底层代码来观察一下

```
static void prvReloadTimer( Timer_t * const pxTimer,
                            TickType_t xExpiredTime,
                            const TickType_t xTimeNow )
{
    /* Insert the timer into the appropriate list for the next expiry time.
     * If the next expiry time has already passed, advance the expiry time,
     * call the callback function, and try again. */
    while( prvInsertTimerInActiveList( pxTimer, ( xExpiredTime + pxTimer->xTimerPeriodInTicks ), xTimeNow, xExpiredTime ) != pdFALSE )
    {
        /* Advance the expiry time. */
        xExpiredTime += pxTimer->xTimerPeriodInTicks;

        /* Call the timer callback. */
        traceTIMER_EXPIRED( pxTimer );
        pxTimer->pxCallbackFunction( ( TimerHandle_t ) pxTimer );
    }
}
```

- 看到函数底层代码的实现，我们可以看到，一进入函数，就要进行一个while循环的条件判断，我们看到判断条件是一个将定时器插入到活跃链表当中是否成功，prvInsertTimerInActiveList（）函数，是将一个定时器加入到定时器活页链表当中的API函数，还是老样子，我们来看一下，底层代码的实现

```c
static BaseType_t prvInsertTimerInActiveList( Timer_t * const pxTimer,
                                              const TickType_t xNextExpiryTime,
                                              const TickType_t xTimeNow,
                                              const TickType_t xCommandTime )
{
    BaseType_t xProcessTimerNow = pdFALSE;

    listSET_LIST_ITEM_VALUE( &( pxTimer->xTimerListItem ), xNextExpiryTime );
    listSET_LIST_ITEM_OWNER( &( pxTimer->xTimerListItem ), pxTimer );

    if( xNextExpiryTime <= xTimeNow )
    {
        /* Has the expiry time elapsed between the command to start/reset a
         * timer was issued, and the time the command was processed? */
        if( ( ( TickType_t ) ( xTimeNow - xCommandTime ) ) >= pxTimer->xTimerPeriodInTicks )
        {
            /* The time between a command being issued and the command being
             * processed actually exceeds the timers period.  */
            xProcessTimerNow = pdTRUE;
        }
        else
        {
            vListInsert( pxOverflowTimerList, &( pxTimer->xTimerListItem ) );
        }
    }
```

```c
    else
    {
        if( ( xTimeNow < xCommandTime ) && ( xNextExpiryTime >= xCommandTime ) )
        {
            /* If, since the command was issued, the tick count has overflowed
             * but the expiry time has not, then the timer must have already passed
             * its expiry time and should be processed immediately. */
            xProcessTimerNow = pdTRUE;
        }
        else
        {
            vListInsert( pxCurrentTimerList, &( pxTimer->xTimerListItem ) );
        }
    }

    return xProcessTimerNow;
}
```

- 这个函数负责智能的将定时器插入到正确的活动列表中，同时检查各种时间边界，确定是否有定时器需要立即处理；
- 我们来看一下传入的四个参数是什么
  - pxTimer：要操作的定时器
  - xNextExpiryTime：传入定时器的下一个到期时间
  - xTimeNow：获取的当前系统时间，用于判断定时器是否已经到期
  - xCommandTime：定时器命令发出时间，用于判断命令处理是否存在延迟
- 看完了每个参数代表的含义，接下来我们正式进入代码的查看，已进入函数先初始化一个用来表示是否需要立即处理定时器的标志xProcessTimerNow，紧接着就是调用了两个宏定义的函数listtSET_LISTT_ITEM_VALUE来设置定时器链表节点的值为定时器下一次到期时间xNextExpiryTime，然后调用listSET_LIST_ITEM_OWNER来设置定时器链表节点对象为当前定时器pxTimer；然后就是判断定时器是否到期（xNextExpiryTime<=xTimeNow是否成立）--
  - 为什么定时器下一次到期时间小于等于当前系统时间，就说明定时器已经到期或者刚好到期，我们假设定时器A的下一次到期时间是201个系统节拍时，假如此时系统的节拍结束已经到201了或者是已经超过201了，是不是就说明刚刚好到定时器A的到期时间或者是已经过了定时器A的到期时间，说明了定时器A已经到期了
- --在定时器到期情况下，此时会进行下一步的判断，从发生定时器操作命令的时间到现在的时间间隔是否已经>=定时器的计数周期了又分为以下两种情况
  - 一、如果时间间隔大于等于计数周期，说明处理命令的时间已经超过了定时器的周期了，说明此时定时器需要立即处理，此时将定时器立即处理标志位xProcessTimerNow置为pdTRUE，表示定时器需要立即处理
  - 二、如果时间间隔小于技术周期，说明虽然定时器已经到期了，但是命令处理的延迟在定时器的周期范围内，此时调用vListInsert（）函数，将定时器链表节点插入到pxOverflowTimerList（溢出定时器链表）中，等待后续的处理。既然提到了插入链表函数（list.c），那么我们必然要看一下函数底层代码的实现，来好好学习一下啊，话不多说，我们直接开整

```
void vListInsert( List_t * const pxList,
                  ListItem_t * const pxNewListItem )
{
    ListItem_t * pxIterator;
    const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
    traceENTER_vListInsert( pxList, pxNewListItem );
    listTEST_LIST_INTEGRITY( pxList );
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
    if( xValueOfInsertion == portMAX_DELAY )
    {
        pxIterator = pxList->xListEnd.pxPrevious;
    }
    else
    {
        for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
        {
            /* There is nothing to do here, just iterating to the wanted
             * insertion position. */
        }
    }
    pxNewListItem->pxNext = pxIterator->pxNext;
    pxNewListItem->pxNext->pxPrevious = pxNewListItem;
    pxNewListItem->pxPrevious = pxIterator;
    pxIterator->pxNext = pxNewListItem;
    pxNewListItem->pxContainer = pxList;
    ( pxList->uxNumberOfItems ) = ( UBaseType_t ) ( pxList->uxNumberOfItems + 1U );
    traceRETURN_vListInsert();
}
```

- 这个函数是FreeRTOS中链表操作的核心函数，它是将新链表节点按值排序插入到链表中的核心函数，它实现了双向链表的智能插入算法，确保了链表项按照想xItemValue升序排列。
  - 函数参数
    - pxList：要插入到目标链表
    - pxNewListItem：要插入到新链表节点指针
  - 看完了函数的作用和参数，那么接下来我们正式取观察一下函数内部的实现，函数开始初始化了两个局部变量，分别是--
    - pxIterator：用于遍历链表的迭代器指针
    - xValueOfInsertion：获取新节点的“排序值”（如定时器的排序值xItemValue），后续根据该值确定插入位置
  - --初始化过后，就是调用list链表中两个宏定义函数来检查链表和节点的完整性，然后根据获取的新节点的值来和portMAX_Delay做判断，我们知道portMAX_DELAY是最大延时的意思，这里表示的是最大值（无限大），这段代码的意思就是，如果获取的新节点的值比链表中任一节点的值都大，那我们就把新节点插入到链表的末尾；
  - 否则我们就对这个新节点进行普通值插入，此时首先会经历一个空的for循环，这个for循环并不做事，只是用来找到新节点的插入位置的，我们看到，for循环的第一个参数将迭代器指针指向了链表的末尾节点，说明从链表的末尾开始执行遍历。我们看到for循环的循环条件，链表迭代器指针指向的下一个节点的值<=我们传入的新节点的值，此时循环会继续后移迭代器，直到pxIterator找到了一个节点值，此节点值>新节点值，此时就会将迭代器的指针会停在“第一个节点值>新节点值”的位置前一个节点，即新节点应插入到pxIterator和pxIterator->pxNext之间
  - 找到了新节点应该插入到链表中的位置过后，下一步就是将新节点插入到链表当中，这也是操作链表的核心所在，我们来看一下，Free RTOS是如何将新节点插入到链表中的，
    - 首先我们要设置新节点（要插入的节点）的后继指针（新节点后面那个节点）指向迭代器的下一个节点（就是前面for循环中遍历到的pxItemator->pxNext，即第一个比新节点值大的节点位置），设置完后一个节点的位置之后，接下来就是设置新节点的位置
    - 更新后继节点的前驱指针，就是上一步我们设置那个节点的前一个节点位置，此位置就是我们用来放置新节点的位置，我们并不可以直接让新节点指向这个位置，我们可以借组后一个节点的位置，让此节点的前一个位置指向传入的新节点即可，此时就把新节点设置到了链表当中合适的位置上了，因为改变了链表中插入新节点所在的位置，我们需要把新节点所在位置的前一个节点也确定下来
    - 设置新节点的前驱指针，就是新节点的前一个节点位置，我们让它指向我们之前for循环中遍历的迭代器的位置因为之前for循环中我们查找新节点的位置的时候，就是找到的迭代器和迭代器下一个节点之间的位置，所以，新节点的前一个节点应该指向迭代器，为了确保节点的位置的固定和连续性，所以，我们让迭代器的后一个节点指向新节点所在的位置
    - 更新迭代器的后继指针，让它指向新节点所在的位置
    - 经过上述一系列的操作之后，我们就完成了链表中插入新节点的核心功能了
    - 最后我们在更新一下列表的状态，我们设置一下新节点的容器指针让它指向我们存放的链表，然后把链表中的节点数目计数+1
  - 这就是插入链表函数的底层代码的全部实现了，那么我们回到prvInsertTimerInActiveList（）函数中的对应位置继续往下接着看一下，上面我们说到，如果定时器命令的操作时间到现在时间间隔小于定时器的计数周期的话，我们就把调用函数将定时器插入到pxOverflowTimerList（溢出定时器链表）中。这些情况都是都是在定时器到期的情况下才会执行的，那么如果定时器没到期呢，我们看到上面的代码也会分为两种情况--
    - 一、当系统获取到的现在的时间<命令发生的时间（xTimeNow<xCommandTime）并且下一次到期的时间>=命令发生的时间（xNextExpiryTime>=xCommandTime），此种情况说明系统节拍已经溢出，并且定时器已经到期了，需要立即处理，所以将定时器立即处理标志位xProcessTimerNow置为pdTRUE
      - 为什么说xTimeNow<xCommandTime&&xNextExpiryTime>=xCommandTime就说明定时器已经到期

了，并且因为系统节拍也溢出，需要马上将到期的定时器给处理调，以免下一次定时器到期覆盖之前的到期；因为系统节拍是循环计数的，即计数到达最大值之后，重新从0开始计数时钟节拍，所以如果当前时间xTimeNow当前时间<xCommandTime命令时间的话，说明系统节拍已经溢出重新计数了，要不然命令的时间因该<=当前时间的，在系统节拍溢出计数的情况下，定时器下一次到期的时间>=定时器操作命令的时间的话说明到期时间在命令时间之后，这样解释可能有点模糊，我们用一个时间线来解释一下

- ◇ 假设命令在时间T1=xCommandTime发生（此时系统节拍未溢出）
- ◇ 定时器的到期时间设为T2=xNextExpiryTime（T2>=T1,即“在命令时间之后到期”）
- ◇ 节拍溢出之后，当前的时间T_now=xTimeNow（因为已经发生溢出，计数节拍回0了，所以T_now<T1）

□ 二、如果不满足定时器立即处理的条件的话，此时就调用插入链表的函数，将定时器中的链表节点插入到活跃定时器链表pxCurrentTimerList中去，关于怎么将定时器中的链表节点插入到定时器链表中的，上面已经详细讲过了，这里就不再赘述了

- 函数执行到最后会返回定时器立即处理标志位下xProcessTimerNow
- 看完prvInsertTimerInActiveList（）函数，我们回到prvRelooadTimer（）中去继续往下看看下面的代码，while循环中的判断条件就是看一下定时器立即处理标志位xProcessTimerNow是否被置为pdTRUE了，表示定时器的到期时间已经过期了，那么将会执行while循环中的代码，此时会将执行xExpiredTime+=pxTimer->xTimerPeriodInTicks这句代码，这表示将下一次定时器到期再次加上一个定时器在周期，那么此时定时器就会产生一个新的“下一个定时器到期时间”（因为进入到while循环中的定时器都是“下一个到期时间已经过期了”的，需要立刻处理的定时器），我们让定时器加上一个周期之后，此时这个过期的到期时间就会从已经过期变成新的到期时间了，紧接着就调用用于自定义的回调函数pxCallbackFunction（）
  - 总结下来就是计算下一次的到期时间（本次到期时间+一个周期，xExpiredTime+=pxTimer->xTimerPeriodInTicks（prvInsertTimerInActiveList（）函数参数中的）），然后尝试将定时器插入到活跃链表中；如果下一次到期时间已经过期，那么就推进到期时间（加上一个周期xExpiredTime+=pxTimer->xTimerPeriodInTicks（这是while循环中的）），然后调用定时器回调函数，执行用户定义的回调函数。然后再次尝试将定时器插入定时器活跃链表中，直到到期时间合法为止，这就是重装载定时器的函数。
- 回到prvProcessExpiredTimer（）中，看完了prvRelooadTimer（）重装载函数，我们接着往下看，则是设置了自动重装载标志位的，如果每设置自动重装载，那么就会将定时器结构体中的状态位设置为“非活跃”（清除活跃标志），不再参与计时，然后下面就是调用定时器回调函数，来执行用户自定义的代码，这就是prvProcessExpiredTimer（）处理到期定时器函数中的全部内容，我们来总结一下
  - 函数获取到到期的定时器的操作句柄之后，将定时器从活跃定时器链表中移除，根据定时器是否设置了自动重装载标志位，来决定定时器是进入到自动重装载函数中，将定时器的到期时间设置到合法范围，并将定时器重新插入到活跃链表中，还是将定时器从活跃链表中移除后，清除定时器结构体中的活跃标志位，不再参与计时，最后调用定时器到期执行的回调函数。
- 回到prvProcessTimerOrBlockTask（）函数，我们继续往下看，前面看到了如果定时器活跃链表不为空，且定时器已经到期的情况下，会调用prvProcessExpiredTimer（）处理到期定时器函数，那么如果没有定时器到期或者定时器活跃链表为空的情况下（走else分支），此时会进一步判断定时器溢出链表是否为空（ xListWasEmpty = **listLIST_IS_EMPTY**(pxOverflowTimerList) ），紧接着就是调用一个阻塞等待定时器消息队列的函数vQueueWaiForMessageRestricted（）函数，函数参数传入了定时器消息队列的句柄，阻塞等待时间，以及定时器链表的状态（是否为空），既然谈到了这个消息队列的等待函数，那必须是深入了解一下，话不多说，我们直接开干

```
void vQueueWaitForMessageRestricted( QueueHandle_t xQueue,
                                     TickType_t xTicksToWait,
                                     const BaseType_t xWaitIndefinitely )
{
    Queue_t * const pxQueue = xQueue;

    traceENTER_vQueueWaitForMessageRestricted( xQueue, xTicksToWait, xWaitIndefinitely );
    prvLockQueue( pxQueue );

    if( pxQueue->uxMessagesWaiting == ( UBaseType_t ) 0U )
    {
        /* There is nothing in the queue, block for the specified period. */
        vTaskPlaceOnEventListRestricted( &( pxQueue->xTasksWaitingToReceive ), xTicksToWait, xWaitIndefinitely );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    prvUnlockQueue( pxQueue );

    traceRETURN_vQueueWaitForMessageRestricted();
}
```

- 这是FreeRTOS内核中为内核代码设计的受限队列等待函数，它实现了高效的队列等待机制，专门用于定时器守护任务等内核组件的阻塞等待，此函数的作用是负责将调用任务放置到队列的等待接收列表中，实现了高效的阻塞等待机制，但不会立即阻塞任务，而是在调度器解锁时才真正阻塞。函数一开始就是将队列句柄转换为内部队列结构的指针，方便后续操作。然后函数调用一个我们之前学习队列时从来没有见到过的函数--队列加锁函数prvLockQueue（），将队列传入之后，队列会被加上一个“锁”，此锁会防止多任务同时操作队列，导致数据竞争（效果上类似于之前的进入临界区代

码），因为类似于进入临界区代码，所以我们就不在深入底层代码学习了，紧接着判断队列中的等待的消息数量是否为空，如果为空，就会调用挂起任务函数vTaskPlaceOnEventListRestricted（）函数，这个函数时整个队列等待函数中的核心部分，所以必须要深入学习一底层代码的实现

```
void vTaskPlaceOnEventListRestricted( List_t * const pxEventList,
                                      TickType_t xTicksToWait,
                                      const BaseType_t xWaitIndefinitely )
{
    traceENTER_vTaskPlaceOnEventListRestricted( pxEventList, xTicksToWait, xWaitIndefinitely );

    configASSERT( pxEventList );
    listINSERT_END( pxEventList, &( pxCurrentTCB->xEventListItem ) );

    /* If the task should block indefinitely then set the block time to a
     * value that will be recognised as an indefinite delay inside the
     * prvAddCurrentTaskToDelayedList() function. */
    if( xWaitIndefinitely != pdFALSE )
    {
        xTicksToWait = portMAX_DELAY;
    }

    traceTASK_DELAY_UNTIL( ( xTickCount + xTicksToWait ) );
    prvAddCurrentTaskToDelayedList( xTicksToWait, xWaitIndefinitely );

    traceRETURN_vTaskPlaceOnEventListRestricted();
}
```

- 通过函数的名字我们可以知道，此函数是用于将当前任务加入到事件列表中，并进行任务的延时或者等待。接下来我们来看一下底层函数的代码，函数开始时调用一个链表的宏函数（listINSERT_END（）函数），将当前任务的链表节点添加到事件链表的末尾，紧接着就是判断任务是否需要无限等待，根据xWaitIndefinitely标志位决定，如果xWaitIndefinitely为pdTRUE，就把等待时间设置为portMAX_DELAY最大等待延时，以实现无限等待，如果不满足条件，就调用把任务放入阻塞链表的函数（prvAddCurrentTaskToDelayedList（）函数），把传入的任务从就绪态任务链表转移到阻塞态任务链表中。这就是这个函数底层代码的全部内容了

- 回到vQueueWaiForMessageRestricted（）函数中，我们继续往下看，如果消息队列中的信息数量不为0，我们就调用函数将队列做标记，然后调用“解锁”函数（prvUnlockQueue（）与我们前面的"上锁"函数prvLockQueue（）配套使用，也就是上锁的次数要和解锁的次数对应才可以，用法上和调用临界区和退出临界区类似），这就是这个函数的全部代码实现
- 回到prvProcessTimerOrBlockTask（）函数，我们继续往下观看，最后函数调用恢复任务调度器的函数，这就是此函数底层的全部代码
- 我们回到定时器守护任务的代码，看完定时器处理和任务阻塞函数的代码之后，我们来看定时器守护任务中最重要的代码，也就是我们定时器处理接收到命令的函数prvProcessReceivedCommands（）函数，话不多说，我们直接开整，看看函数底层代码如何实现的

```
static void prvProcessReceivedCommands( void )
{
    DaemonTaskMessage_t xMessage = { 0 };
    Timer_t * pxTimer;
    BaseType_t xTimerListsWereSwitched;
    TickType_t xTimeNow;

    while( xQueueReceive( xTimerQueue, &xMessage, tmrNO_DELAY ) != pdFAIL )
    {
        /* Commands that are positive are timer commands rather than pended
         * function calls. */
        if( xMessage.xMessageID >= ( BaseType_t ) 0 )
        {
            /* The messages uses the xTimerParameters member to work on a
             * software timer. */
            pxTimer = xMessage.u.xTimerParameters.pxTimer;

            if( listIS_CONTAINED_WITHIN( NULL, &( pxTimer->xTimerListItem ) ) == pdFALSE )
            {
                /* The timer is in a list, remove it. */
                ( void ) uxListRemove( &( pxTimer->xTimerListItem ) );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            traceTIMER_COMMAND_RECEIVED( pxTimer, xMessage.xMessageID, xMessage.u.xTimerParameters.xMessageValue );
```

```
xTimeNow = prvSampleTimeNow( &xTimerListsWereSwitched );
switch( xMessage.xMessageID )
{
    case tmrCOMMAND_START:
    case tmrCOMMAND_START_FROM_ISR:
    case tmrCOMMAND_RESET:
    case tmrCOMMAND_RESET_FROM_ISR:
        /* Start or restart a timer. */
        pxTimer->ucStatus |= ( uint8_t ) tmrSTATUS_IS_ACTIVE;

        if( prvInsertTimerInActiveList( pxTimer, xMessage.u.xTimerParameters.xMessageValue + pxTimer->xTimerPeriodInTicks, xTimeNow, xMessage.u.xTimerParameters.xMessageValue ) != pdFALSE )
        {
            /* The timer expired before it was added to the active
             * timer list.  Process it now. */
            if( ( pxTimer->ucStatus & tmrSTATUS_IS_AUTORELOAD ) != 0U )
            {
                prvReloadTimer( pxTimer, xMessage.u.xTimerParameters.xMessageValue + pxTimer->xTimerPeriodInTicks, xTimeNow );
            }
            else
            {
                pxTimer->ucStatus &= ( ( uint8_t ) ~tmrSTATUS_IS_ACTIVE );
            }

            /* Call the timer callback. */
            traceTIMER_EXPIRED( pxTimer );
            pxTimer->pxCallbackFunction( ( TimerHandle_t ) pxTimer );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        break;

    case tmrCOMMAND_STOP:
    case tmrCOMMAND_STOP_FROM_ISR:
        /* The timer has already been removed from the active list. */
        pxTimer->ucStatus &= ( ( uint8_t ) ~tmrSTATUS_IS_ACTIVE );
        break;

    case tmrCOMMAND_CHANGE_PERIOD:
    case tmrCOMMAND_CHANGE_PERIOD_FROM_ISR:
        pxTimer->ucStatus |= ( uint8_t ) tmrSTATUS_IS_ACTIVE;
        pxTimer->xTimerPeriodInTicks = xMessage.u.xTimerParameters.xMessageValue;
        configASSERT( ( pxTimer->xTimerPeriodInTicks > 0 ) );

        /* The new period does not really have a reference, and can
         * be longer or shorter than the old one.  The command time is
         * therefore set to the current time, and as the period cannot
         * be zero the next expiry time can only be in the future,
         * meaning (unlike for the xTimerStart() case above) there is
         * no fail case that needs to be handled here. */
        ( void ) prvInsertTimerInActiveList( pxTimer, ( xTimeNow + pxTimer->xTimerPeriodInTicks ), xTimeNow, xTimeNow );
        break;

    case tmrCOMMAND_DELETE:
        #if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
        {
            /* The timer has already been removed from the active list,
             * just free up the memory if the memory was dynamically
             * allocated. */
            if( ( pxTimer->ucStatus & tmrSTATUS_IS_STATICALLY_ALLOCATED ) == ( uint8_t ) 0 )
            {
                vPortFree( pxTimer );
            }
            else
            {
                pxTimer->ucStatus &= ( ( uint8_t ) ~tmrSTATUS_IS_ACTIVE );
            }
        }
        #endif /* configSUPPORT_DYNAMIC_ALLOCATION */
        break;
    default:
        /* Don't expect to get here. */
        break;
}
```

- 这就是定时器命令接收处理函数的底层代码，接下来我们一步步分析一下，首先函数一开始就初始化了一个 DaemonTaskMessage_t 类型的结构体，这个结构体我们上面已经讲过了，但是为了真正理解这个重要函数，在这里我们再讲一遍

```
and xCallbackParameterStype respectively. */
typedef struct tmrTimerParameters
{
    TickType_t xMessageValue; /**< An optional value used by a subse
    Timer_t * pxTimer;        /**< The timer to which the command wi
} TimerParameter_t;

The structure that contains the two message types, along with an iden
that is used to determine which message type is valid. */
typedef struct tmrTimerQueueMessage
{
    BaseType_t xMessageID; /**< The command being sent to the timer
    union
    {
        TimerParameter_t xTimerParameters;

        /* Don't include xCallbackParameters if it is not going to b
         * it makes the structure (and therefore the timer queue) la
        #if ( INCLUDE_xTimerPendFunctionCall == 1 )
```

```
and xCallbackParameters type respectively. */
typedef struct tmrTimerParameters
{
    TickType_t xMessageValue; /**< An optional value used by a subse
    Timer_t * pxTimer;        /**< The timer to which the command wi
} TimerParameter_t;

The structure that contains the two message types, along with an iden
that is used to determine which message type is valid. */
typedef struct tmrTimerQueueMessage
{
    BaseType_t xMessageID; /**< The command being sent to the timer
    union
    {
        TimerParameter_t xTimerParameters;

        /* Don't include xCallbackParameters if it is not going to b
         * it makes the structure (and therefore the timer queue) la
        #if ( INCLUDE_xTimerPendFunctionCall == 1 )
            CallbackParameters_t xCallbackParameters;
        #endif /* INCLUDE_xTimerPendFunctionCall */
    } u;
} DaemonTaskMessage_t;
```

- 这个结构体里面包含了一个消息ID和一个参数联合体，前面定义结构体的时候讲结构体初始化为0，说明将消息ID初始化为0，然后又初始化了一个定时器结构体和两个局部变量，然后进入一个循环，我们来看一下循环条件，判断队列接收消息函数是否接收到了消息，看看函数的参数，其中传入了定时器消息队列句柄、传入了消息ID的地址，第三个参数很关键，它是专门用于定时器无延时等待的宏定义，这就说明循环条件是看不带延时的消息接收队列是否真的接收到了数据，如果接收到了数据，就进入循环中的代码去处理接收到的命令，否则就一直判断while循环的条件，确保一有命令传入，可以及时的得到处理
- 进入while循环首先判断接收到消息的ID号是否是>=0的，为什么要进行这个判断呢，那么我们就需要看一下定时器命令ID是如何定义的

```
#define tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR    ( ( BaseType_t ) -2 )
#define tmrCOMMAND_EXECUTE_CALLBACK             ( ( BaseType_t ) -1 )
#define tmrCOMMAND_START_DONT_TRACE             ( ( BaseType_t ) 0 )
#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 )

#define tmrFIRST_FROM_ISR_COMMAND               ( ( BaseType_t ) 6 )
#define tmrCOMMAND_START_FROM_ISR               ( ( BaseType_t ) 6 )
#define tmrCOMMAND_RESET_FROM_ISR               ( ( BaseType_t ) 7 )
#define tmrCOMMAND_STOP_FROM_ISR                ( ( BaseType_t ) 8 )
#define tmrCOMMAND_CHANGE_PERIOD_FROM_ISR       ( ( BaseType_t ) 9 )
```

  - 这是定时器命令的ID展示，前面的时候我已经讲过了，我们可以看到只有两个命令是小于0的

tmrCOMMAND_EXECUTE_CALLBACK_FROM_ISR	-2	从中断服务函数（ISR）中触发定时器回调执行（仅在某些特殊场景下使用）。
tmrCOMMAND_EXECUTE_CALLBACK	-1	不启动或停止定时器，而是**直接执行一个回调函数**。常用于定时器守护任务直接执行回调操作。

  - 我们可以看到小于0的情况分别是在中断服务程序中调用执行回调函数命令和在任务中执行回调函数命令，总之，发生执行回调函数代码，就不符合条件，如果不符合条件，就不会继续执行下面的代码
- 如果消息ID满足条件，那么会继续执行下面的代码，首先是从消息中获取软件定时器，然后就调用链表中的内置宏定义函数，来检查一下定时器是否在链表当中，如果定时器在链表当中，就调用链表移除函数（uxListRemove（））将定时器从链表当中移除掉，然后函数就调用获取当前系统时间的函数（prvSampleTimeNow（）函数，前面已经学习过了），并将值赋值给前面定义的局部变量
- 然后就进入到了这个函数最最最核心的代码部分，函数就是根据这里的代码去处理传入的命令的，我们看到这是一个switch语句，其中switch中传入的要进行比对的参数是队列中的消息ID，也就是传入的命令，进入到switch中，首先我们可以看到有四个case条件判断连在了一起，这四个分别是，定时器启动命令（Start）、在中断服务程序中定时器启动命令（Start_From_ISR）、定时器复位命令（Reset）、在中断服务程序中复位定时器命令（Reset_From_ISR），学过了c语言可以知道，如果case语句后面什么都不跟，那么它就会继续向下执行，说明这四个定时器命令处理过程是相同的，进入到这四个命令的处理代码中
  - 首先，将定时器结构体状态位置为活跃状态，然后调用将定时器插入到活跃链表函数（prvInsertTimerInActiveList（）函数（我们前面详细学习过）），看到参数，第一个参数是将要加入链表的定时器传入，第二个参数是（xMessage.u.xTimerParameters.xMessageValue + pxTimer->xTimerPeriodInTicks）表示定时器下一次到期时间（定时器命令的发出时间+定时器的一个周期的时间），第三个参数传入的是前面获取到的当前系统的时间，第四个参数是xMessage.u.xTimerParameters.xMessageValue 也就是定时器命令的发出时间，这是函数的参数，这里要判断是否成功将定时器插入到活跃定时器链表当中，如果成功插入，就会进入到if中执行下面的代

码，进一步判断定时器的状态是否设置了自动重装载标志，如果设置了重装载标志位，那么就会调用执行定时器重装载函数（prvReloadTimer（）函数），如果定时器不是重装载定时器，那么就取消定时器的活跃标志位，然后执行定时器的定义的回调函数，最后执行Break，结束case语句

- 我们看到下面两个定时器命令处理的case语句，分别是定时器停止（Stop）、在中断服务程序中停止定时器（Stop_From_ISR）,进入case语句，我们直接将定时器中的状态标志位中的活跃定时器标志位清除，然后Break结束case语句对这两个命令的处理
- 接下来两个定时器命令分别是修改定时器周期命令（Change_Period）、和在中断服务程序中修改定时器周期的命令（Change_Period_From_ISR），看到处理过程，首先就是将定时器中的状态标志位设置为活跃定时器，然后将定时器结构体中的定时器周期值改变，将消息结构体中的值赋值给定时器的周期，达到修改定时器周期的目的，然后再调用将定时器插入到活跃链表的函数（prvInsertTimerInActiveList（）），传入的参数分别是要修改的定时器、定时器下一次的到期时间（因为修改了定时器的周期，此时传入的是新的定时器周期）、获取的当前系统的时间，最后一个也是给的当前系统的时间，因为是此时修改的定时器的周期，最后Break结束case语句
- 最后我们看到最后一个命令--删除定时器命令（Delete），进入到case语句中，首先会判断宏是否开启，这个条件编译是判断是否开启了动态内存分配，因为如果开启了动态内存分配，到最后删除定时器的时候，内存是会被系统回收的，进入下一步判断，如果定时器的状态&上静态分配内存标志位如果为0，即不是静态分配的，是动态分配的内存，我们就进入if，调用释放内存函数vPortFree（），并将要释放的定时器传入。否则如果是静态分配的内存，那么我们直接取消定时器的活跃状态标志位，最后break结束case语句

- 这就是定时器守护任务接收并处理命令底层的全部代码，我们可以看到函数底层就是在一个while循环中不停的判断是否又消息命令传入，一有命令传入，函数就紧急对命令进行处理，关于处理命令的部分，则是借助了一个switch语句来完成的，根据传入命令的ID的不同，进入不同的命令处理程序

- 以上就是定时器守护任务的全部内容，我们从创建定时器，到向定时器中写入命令，再到系统底层是如何去处理这些命令的和角度去讲述了定时器的内部机制，相信看完了这些，你已经对定时器又相当深厚的理解

四、软件定时器的实现核心是什么

- 我们知道了软件定时器的计数是依靠系统时钟计数完成，软件定时器是依靠软件来实现完成的，那么说明软件定时器并不是系统初始时刻就存在的，而是需要自己创建的。那么我手动完成后软件定时器它是怎么进行操作的呢
- 这里就要提到软件定时器的核心操作
  - **系统节拍（System Tick）**
    - 上面我们已经详细说明了软件定时器的计数依靠硬件定时器产生的系统时钟计数，这是软件定时器最基础的功能实现
  - **定时器控制结构体Timer_t**
    - 定时器的所有状态和属性都是通过定时器结构体（Timer_t）来控制的，因为我没有深入了解软件定时器，这里就不深入解释了，下面我们会对软件定时器结构体做出解读
  - **定时器链表（xActiveTimerList）**
    - 系统中的所有运行状态的软件定时器都会存入这个定时器活跃链表中，链表中的定时器按照距离超时时间的大小依次排列，便捷了系统检查定时器是否已经超时
  - **软件定时器消息队列（xTimerQueue）**
    - 软件定时器在创建完成之后并不会自动启动，而是需要手动开启软件定时器，那么我们就需要手动向系统发送通知，告诉系统我们要开启软件定时器了，那么如何向系统发送，即可以确保消息发送成功，又可以保证系统一定可以接收到消息呢，此时我们可以想到利用队列和系统通信来完成这一操作，这就是系统级消息队列。
    - 因为Free RTOS是多任务实时系统，用户可在多个任务或者中断上下文中同时操作定时器，前面学习队列可以知道，队列的发送和接收都和数据缓冲区有关，正是借助这一点，如果缓冲区已满或者空的话，可以进入阻塞，如果没有中间队列的消息传递，直接在多个上下文修改定时器的状态，会导致数据竞争，利用队列来通信，避免了丢失或者读出错误信息破坏整个系统的运行，安全性高
    - 因为要经过消息队列传递给系统，系统需要从队列中读取数据，所以操作软件定时器不会直接修改软件定时器的状态，而是将操作封装成“命令”发送到队列，定时器服务任务独占队列读取权，按顺序处理命令，确保每个命令对定时器的修改都是原子且有序的，避免了多上下文冲突
  - **回调函数**
    - 软件定时器在到期之后就是触发用户自定的任务回调程序，这是定时器实现用户功能的关键
  - **定时器守护任务（也叫定时器服务任务）**
    - 这个非常非常重要，因为它是系统接收和处理软件定时器命令的关键，消息队列中的数据都是向此任务传递的，所有的命令也是又该任务接收的，接收到命令之后，也是在该函数中执行的，定时器的重装载，定时器的链表的插入等等都与这个函数相关

五、软件定时器中常见的坑

- **1、不要在回调函数中长时间**
  - 回调函数在守护任务运行时，阻塞会阻塞所有的定时器命令处理。若需要长任务，回调应xTaskCreate/xTaskNotify/xQueueSend给工作任务，尽快返回
- **2、回调任务内可以调用大多数FreeRTOS中的API**

        - 但是注意：回调任务的优先级=守护任务的优先级，API的使用应该考虑优先级/死锁问题
    - **3、FromISR版本的限制**
        - FromISRAPI不能执行阻塞调用，且传入的pxHigherPriorityTaskWoken需要正确处理以触发上下文
    - **4、tick精度和溢出**
        - 定时器以tick为单位，不是实时精确的微秒级。Tick频率越高精度越高，但是CPU负担也越大
- 六、软件定时器的运行环境和回调上下文
    - 所有软件定时器的回调函数都在定时器服务任务（也叫守护任务）上下文中执行，而不是在创建它的任务或ISR中执行
    - 回调不能阻塞（不可以调用会阻塞的API），否则会卡住定时器服务任务（守护任务），从而影响所有软件定时器
    - 若要在其他任务上下文处理较长工作，回调通常会通知某个工作任务（比如通过队列、信号量、任务通知）来完成真正耗时的工作
- 七、内核实现关键点复述
    - 命令通过消息队列传递到软件定时器守护任务（xTimerQueue）
    - 守护任务维护两个“活动”链表
    - 每个定时器有关listItem，List按照到期时间排序，头部即下一个到期项
    - 插入、删除、重载通过内部函数（如prvInsertTimerInActiveList（）、prvReloadTimer（）、uxListRemove（））操作链表
    - 回调在守护任务中执行，auto-reload（周期定时器）的会在回调函数后重新插入链表，one-shot（单次定时器）的会清除active标志