title: 【TencentOS tiny】深度源码分析（8）——软件定时器 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:19:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

软件定时器的基本概念

TencentOS tiny 的软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受硬件定时器资源限制的定时器服务，本质上软件定时器的使用相当于扩展了定时器的数量，允许创建更多的定时业务，它实现的功能与硬件定时器也是类似的。

> 硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。

软件定时器的超时处理是指：在定时时间到达之后就会自动触发一个超时，然后系统跳转到对应的函数去处理这个超时，此时，调用的函数也被称回调函数。

回调函数的执行环境可以是中断，也可以是任务，这就需要你自己在tos_config.h通过TOS_CFG_TIMER_AS_PROC宏定义选择回调函数的执行环境了。

- TOS_CFG_TIMER_AS_PROC 为 1：回调函数的执行环境是中断
- TOS_CFG_TIMER_AS_PROC 为 0：回调函数的执行环境是任务

这与硬件定时器的中断服务函数很类似，无论是在中断中还是在任务中，回调函数的处理尽可能简短，快进快出。

软件定时器在被创建之后，当经过设定的超时时间后会触发回调函数，定时精度与系统时钟的周期有关，一般可以采用SysTick作为软件定时器的时基（在m核单片机中几乎都是采用SysTick作为系统时基，而软件定时器又是基于系统时基之上）。

TencentOS tiny提供的软件定时器支持单次模式和周期模式，单次模式和周期模式的定时时间到之后都会调用软件定时器的回调函数。

- 单次模式：当用户创建了定时器并启动了定时器后，指定超时时间到达，只执行一次回调函数之后就将该定时器停止，不再重新执行。
- 周期模式：这个定时器会按照指定的定时时间循环执行回调函数，直到将定时器删除。

> 在很多应用中，可能需要一些定时器任务，硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，无法提供更多的定时器，可以采用软件定时器，由软件定时器代替硬件定时器任务。但需要注意的是软件定时器的精度是无法和硬件定时器相比的，因为在软件定时器的定时过程中是极有可能被其他中断打断，因此软件定时器更适用于对时间精度要求不高的任务。

软件定时器以tick为基本计时单位，当用户创建并启动一个软件定时器时， TencentOS tiny会根据当前系统tick与用户指定的超时时间计算出该定时器超时的时间expires，并将该定时器插入软件定时器列表。

软件定时器的数据结构

> 以下软件定时器的相关数据结构都在tos_global.c中定义

软件定时器列表

软件定时器列表用于记录系统中所有的软件定时器，这些软件定时器将按照唤醒时间升序插入软件定时器列表k_timer_ctl.list 中，它的数据类型是timer_ctl_t。

```
timer_ctl_t          k_timer_ctl = { TOS_TIME_FOREVER,
TOS_LIST_NODE(k_timer_ctl.list) };

typedef struct timer_control_st {
    k_tick_t    next_expires;
    k_list_t    list;
} timer_ctl_t;
```

- next_expires：记录下一个到期的软件定时器时间。
- list：软件定时器列表，所有的软件定时器都会被挂载到这个列表中。

软件定时器任务相关的数据结构

如果 TOS_CFG_TIMER_AS_PROC 宏定义为0，则表示使用软件定时器任务处理软件定时器的回调函数，那么此时软件定时器的回调函数执行环境为任务；反之软件定时器回调函数的处理将在中断上下文环境中。

```
k_task_t            k_timer_task;
k_stack_t           k_timer_task_stk[TOS_CFG_TIMER_TASK_STK_SIZE];
k_prio_t            const k_timer_task_prio         = TOS_CFG_TIMER_TASK_PRIO;
k_stack_t          *const k_timer_task_stk_addr     = &k_timer_task_stk[0];
size_t              const k_timer_task_stk_size     = TOS_CFG_TIMER_TASK_STK_SIZE;
```

- k_timer_task：软件定时器任务控制块
- k_timer_task_stk：软件定时器任务栈，其大小为TOS_CFG_TIMER_TASK_STK_SIZE
- k_timer_task_prio：软件定时器任务优先级，值为TOS_CFG_TIMER_TASK_PRIO，默认值是(k_prio_t)(K_TASK_PRIO_IDLE - (k_prio_t)1u)，比空闲任务高1个数值优先级，杰杰认为这也是很低的优先级了，这样一来软件定时器的精度将更低，不过好在这个值是可以被用户自定义的，想让精度高一点就将这个软件定时器任务优先级设置得高一点就好。
- k_timer_task_stk_addr：软件定时器任务栈起始地址。
- k_timer_task_stk_size：软件定时器任务栈大小。

> 以下软件定时器的相关数据结构都在tos_timer.h中定义

软件定时器的回调函数

```
// 软件定时器的回调函数类型
typedef void (*k_timer_callback_t)(void *arg);
```

软件定时器的回调函数是一个函数指针的形式，它支持传入一个void指针类型的数据。

软件定时器控制块

每个软件定时器都有对应的软件定时器控制块，每个软件定时器控制块都包含了软件定时器的基本信息，如软件定时器的状态、软件定时器工作模式、软件定时器的周期，剩余时间，以及软件定时器回调函数等信息。

```
typedef struct k_timer_st {
#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    knl_obj_t               knl_obj;    /**< 仅为了验证,测试当前内核对象是否真的是一
个软件定时器 */
#endif

    k_timer_callback_t      cb;         /**< 时间到时回调函数 */
    void                   *cb_arg;     /**< 回调函数中传入的参数 */
    k_list_t                list;       /**< 挂载到软件定时器列表的节点 */
    k_tick_t                expires;    /**< 距离软件定时器的到期时间到期还剩多少时间
（单位为tick） */
    k_tick_t                delay;      /**< 还剩多少时间运行第一个到期的软件定时器
（的回调函数） */
    k_tick_t                period;     /**< 软件定时器的周期 */
    k_opt_t                 opt;        /**< 软件定时器的工作模式选项，可以是单次模式
TOS_OPT_TIMER_ONESHOT，也可以是周期模式TOS_OPT_TIMER_PERIODIC */
    timer_state_t           state;      /**< 软件定时器的状态 */
} k_timer_t;
```

软件定时器的工作模式

```
// if we just want the timer to run only once, this option should be passed to
tos_timer_create.
#define TOS_OPT_TIMER_ONESHOT                   (k_opt_t)(0x0001u)

// if we want the timer run periodically, this option should be passed to
tos_timer_create.
#define TOS_OPT_TIMER_PERIODIC                  (k_opt_t)(0x0002u)
```

- TOS_OPT_TIMER_ONESHOT：**单次**模式，软件定时器在超时后，只会执行一次回调函数，它的状态将被设置为TIMER_STATE_COMPLETED，不再重新执行它的回调函数，当然，用户还是可以重新启动这个单次模式的软件定时器，它并未被删除。

> 如果只希望计时器运行一次，则应将此选项传递给tos_timer_create()。

- TOS_OPT_TIMER_PERIODIC：**周期**模式，软件定时器在超时后，会执行对应的回调函数，同时根据软件定时器控制块中的period成员变量的值再重新插入软件定时器列表中，这个定时器会按照指定的定时时间循环执行（周期性执行）回调函数，直到用户将定时器删除。

> 如果我们希望计时器周期运行，则应将此选项传递给tos_timer_create()。

软件定时器的状态

定时器状态有以下几种：

```
typedef enum timer_state_en {
    TIMER_STATE_UNUSED,     /**< the timer has been destroyed */
    TIMER_STATE_STOPPED,    /**< the timer has been created but not been started,
or just be stopped(tos_timer_stop) */
    TIMER_STATE_RUNNING,    /**< the timer has been created and been started */
    TIMER_STATE_COMPLETED   /**< the timer has finished its expires, it can only
happen when the timer's opt is TOS_OPT_TIMER_ONESHOT */
} timer_state_t;
```

- TIMER_STATE_UNUSED：未使用状态。
- TIMER_STATE_STOPPED：创建了软件定时器，但此时软件定时器未启动或者处于停止状态，调用tos_timer_create()函数接口或者在软件定时器启动后调用tos_timer_stop()函数接口后，定时器将变成该状态。
- TIMER_STATE_RUNNING：软件定时器处于运行状态，在定时器被创建后调用tos_timer_start()函数接口，定时器将变成该状态，表示定时器运行时的状态。
- TIMER_STATE_COMPLETED：软件定时器已到期，只有在软件定时器的模式选择为TOS_OPT_TIMER_ONESHOT时才可能发生，表示软件定时器已经完成了。

创建软件定时器

函数

```
__API__ k_err_t tos_timer_create(k_timer_t *tmr,
                                 k_tick_t delay,
                                 k_tick_t period,
                                 k_timer_callback_t callback,
                                 void *cb_arg,
                                 k_opt_t opt);
```

参数

参数	说明（杰杰）
tmr	软件定时器控制块指针
delay	软件定时器第一次运行的延迟时间间隔
period	软件定时器的周期
callback	软件定时器的回调函数，在超时时调用（由用户自己定义）

参数	说明（杰杰）
cb_arg	用于回调函数传入的形参（void指针类型）
opt	软件定时器的工作模式（单次 / 周期）

> 杰杰觉得 delay与 period 比较有意思，就简单提一下 delay 参数与 period 参数的意义与区别：
>
> - delay参数其实是第一次运行的延迟时间间隔（即第一次调用回调函数的时间），如果软件定时器是单次模式，那么只用 delay 参数作为软件定时器的回调时间，因为软件定时器是单次工作模式的话，只会运行一次回调函数，那么就没有周期一说（period 参数将不起作用），只能是以第一次运行的延迟时间间隔作为它的回调时间。
> - period 参数则是作为软件定时器的周期性回调的时间间隔，就好比你的闹钟，每天 7 点叫你起床，但是delay参数在周期工作模式下的软件定时器也是有作用的，它是对第一次回调函数的延迟时间，举个例子：今天晚上9点的时候，你设置了一个闹钟，闹钟时间是每天早上7点的，那么在10个小时后，闹钟将叫你起床，那么这10个小时就相当于delay参数的值，因为闹钟第一次叫你起床并不是在24小时后，而在明天7点后，闹钟响了，此时闹钟将在一天后才会再响，这24小时则相当于 period 参数的值。

系统中每个软件定时器都有对应的软件定时器控制块，软件定时器控制块中包含了软件定时器的所有信息，那么可以想象一下，创建软件定时器的本质是不是就是对软件定时器控制块进行初始化呢？很显然就是这样子的。因为在后续对软件定时器的操作都是通过软件定时器控制块来操作的，如果控制块没有信息，那怎么能操作嘛~

步骤如下：

1. 判断传入的参数是否正确：软件定时器控制块不为null，回调函数不为null，如果是创建周期模式的软件定时器，那么 period 参数则不可以为0，而如果是单次模式的软件定时器，参数delay则不可以为0，无论是何种模式的软件定时器，delay 参数与 period 参数都不可以为 K_ERR_TIMER_PERIOD_FOREVER，因为这代表着软件定时器不需要运行，那还创建个锤子啊。
2. 根据传入的参数将软件定时器控制块的成员变量赋初值，软件定时器状态state被设置为 TIMER_STATE_STOPPED，expires 则被设置为0，因为还尚未启动软件定时器。
3. 调用tos_list_init()函数将软件定时器控制块中可挂载到k_tick_list列表的节点初始化。

```
__API__ k_err_t tos_timer_create(k_timer_t *tmr,
                                 k_tick_t delay,
                                 k_tick_t period,
                                 k_timer_callback_t callback,
                                 void *cb_arg,
                                 k_opt_t opt)
{
    TOS_PTR_SANITY_CHECK(tmr);
    TOS_PTR_SANITY_CHECK(callback);

    if (opt == TOS_OPT_TIMER_PERIODIC && period == (k_tick_t)0u) {
        return K_ERR_TIMER_INVALID_PERIOD;
    }

    if (opt == TOS_OPT_TIMER_ONESHOT && delay == (k_tick_t)0u) {
```

```
        return K_ERR_TIMER_INVALID_DELAY;
    }

    if (opt != TOS_OPT_TIMER_ONESHOT && opt != TOS_OPT_TIMER_PERIODIC) {
        return K_ERR_TIMER_INVALID_OPT;
    }

    if (delay == TOS_TIME_FOREVER) {
        return K_ERR_TIMER_DELAY_FOREVER;
    }

    if (period == TOS_TIME_FOREVER) {
        return K_ERR_TIMER_PERIOD_FOREVER;
    }

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    knl_object_init(&tmr->knl_obj, KNL_OBJ_TYPE_TIMER);
#endif

    tmr->state          = TIMER_STATE_STOPPED;
    tmr->delay          = delay;
    tmr->expires        = (k_tick_t)0u;
    tmr->period         = period;
    tmr->opt            = opt;
    tmr->cb             = callback;
    tmr->cb_arg         = cb_arg;
    tos_list_init(&tmr->list);
    return K_ERR_NONE;
}
```

销毁软件定时器

软件定时器销毁函数是根据软件定时器控制块直接销毁的，销毁之后软件定时器的所有信息都会被清除，而且不能再次使用这个软件定时器，如果软件定时器处于运行状态，那么就需要将被销毁的软件定时器停止，然后再进行销毁操作。 其过程如下：

1. 判断软件定时器是否有效，然后根据软件定时器状态判断软件定时器是否创建，如果是未使用状态`TIMER_STATE_UNUSED`，则直接返回错误代码`K_ERR_TIMER_INACTIVE`。
2. 如果软件定时器状态是 运行状态 `TIMER_STATE_RUNNING`，那么调用`timer_takeoff()`函数将软件定时器停止。
3. 最后调用`timer_reset()`函数将软件定时器控制块的内容重置，主要是将软件定时器的状态设置为未使用状态`TIMER_STATE_UNUSED`，将对应的回调函数设置为`null`。

```
__API__ k_err_t tos_timer_destroy(k_timer_t *tmr)
{
    TOS_PTR_SANITY_CHECK(tmr);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
```

```
    if (!knl_object_verify(&tmr->knl_obj, KNL_OBJ_TYPE_TIMER)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    if (tmr->state == TIMER_STATE_UNUSED) {
        return K_ERR_TIMER_INACTIVE;
    }

    if (tmr->state == TIMER_STATE_RUNNING) {
        timer_takeoff(tmr);
    }

    timer_reset(tmr);
    return K_ERR_NONE;
}
```

停止软件定时器（内部函数）

在销毁软件定时器的时候提到了timer_takeoff()函数，那么就来看看这个函数具体是怎么样停止软件定时器的，其实本质上就是将软件定时器从软件定时器列表中移除。

> 注意，这个函数是内部静态函数，不是给用户使用的，它与tos_timer_stop()不同。

1. 首先通过TOS_LIST_FIRST_ENTRY宏定义将软件定时器列表k_timer_ctl.list中的第一个软件定时器取出，因为防止软件定时器列表中的第一个软件定时器被移除了，而没有重置软件定时器列表中的相关的信息，因此此时要记录一下第一个软件定时器。
2. 调用tos_list_del()将软件定时器从软件定时器列表中移除，表示中国软件定时器就被停止了，因为不知软件定时器列表中，中国软件定时器也就不会被处理。
3. 判断一下移除的软件定时器是不是第一个软件定时器，如果是，则重置相关信息。如果软件定时器列表中不存在其他软件定时器，则将软件定时器列表的下一个到期时间设置为TOS_TIME_FOREVER，反正则让软件定时器列表的下一个到期时间为第二个软件定时器。

```
__STATIC__ void timer_takeoff(k_timer_t *tmr)
{
    TOS_CPU_CPSR_ALLOC();
    k_timer_t *first, *next;

    TOS_CPU_INT_DISABLE();

    first = TOS_LIST_FIRST_ENTRY(&k_timer_ctl.list, k_timer_t, list);

    tos_list_del(&tmr->list);

    if (first == tmr) {
        // if the first guy removed, we need to refresh k_timer_ctl.next_expires
        next = TOS_LIST_FIRST_ENTRY_OR_NULL(&tmr->list, k_timer_t, list);
        if (!next) {
            // the only guy removed
```

```
            k_timer_ctl.next_expires = TOS_TIME_FOREVER;
        } else {
            k_timer_ctl.next_expires = next->expires;
        }
    }

    TOS_CPU_INT_ENABLE();
}
```

启动软件定时器

在创建成功软件定时器的时候，软件定时器的状态从TIMER_STATE_UNUSED（未使用状态）变成TIMER_STATE_STOPPED（创建但未启动 / 停止状态），创建完成的软件定时器是未运行的，用户在需要的时候可以启动它，TencentOS tiny提供了软件定时器启动函数tos_timer_start()。启动软件定时器的本质就是将软件定时器插入软件定时器列表k_timer_ctl.list 中，既然是这样子，那么很显然需要根据软件定时器的不同状态进行不同的处理。

其实现过程如下：判断软件定时器控制块是否为null，然后判断软件定时器状态，如果为未使用状态TIMER_STATE_UNUSED则直接返回错误代码K_ERR_TIMER_INACTIVE；如果为已经运行状态TIMER_STATE_RUNNING，那么将软件定时器停止，然重新插入软件定时器列表k_timer_ctl.list中；如果是TIMER_STATE_STOPPED 或者TIMER_STATE_COMPLETED状态，则将软件定时器的状态重新设置为运行状态TIMER_STATE_RUNNING，并且插入软件定时器列表k_timer_ctl.list中。

> 注意：插入软件定时器列表的函数是timer_place()。

> tos_timer_start()函数将软件定时器控制块的period或者delay成员变量的值赋值给expires，但这个值是相对的到期时间，而不是绝对值，因此在timer_place()函数中将重新计算得出绝对的到期时间。

```
__API__ k_err_t tos_timer_start(k_timer_t *tmr)
{
    TOS_PTR_SANITY_CHECK(tmr);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!knl_object_verify(&tmr->knl_obj, KNL_OBJ_TYPE_TIMER)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    if (tmr->state == TIMER_STATE_UNUSED) {
        return K_ERR_TIMER_INACTIVE;
    }

    if (tmr->state == TIMER_STATE_RUNNING) {
        timer_takeoff(tmr);
        tmr->expires = tmr->delay;
        timer_place(tmr);
        return K_ERR_NONE;
    }
```

```
    if (tmr->state == TIMER_STATE_STOPPED ||
        tmr->state == TIMER_STATE_COMPLETED) {
        tmr->state = TIMER_STATE_RUNNING;
        if (tmr->delay == (k_tick_t)0u) {
            tmr->expires = tmr->period;
        } else {
            tmr->expires = tmr->delay;
        }
        timer_place(tmr);
        return K_ERR_NONE;
    }
    return K_ERR_TIMER_INVALID_STATE;
}
```

插入软件定时器列表

插入软件定时器列表的函数是`timer_place()`，这个函数会根据软件定时器的到期时间升序排序，然后再插入。

> 该函数是一个内部实现的静态函数

实现过程如下：

1. 根据软件定时器的到期时间`expires`（相对值） 与系统当前时间`k_tick_count`计算得出到期时间`expires`（绝对值）。

> 举个例子，闹钟将在10分钟后叫我起床（这是一个相对值）。闹钟将在当前时间（7:00）的10分钟后叫我起床，那么闹钟响的时间是7:10分，此时的时间就是绝对值。

2. 通过for循环`TOS_LIST_FOR_EACH`找到合适的位置插入软件定时器列表，此时插入软件定时器列表安装到期时间升序插入。

3. 找到合适的位置后，调用`tos_list_add_tail()`函数将软件定时器插入软件定时器列表。

4. 如果插入的软件定时器是唯一定时器列表中的第一个，那么相应的，下一个到期时间就是这个软件定时器的到期时间，将到期时间更新： `k_timer_ctl.next_expires = tmr->expires`。如果`TOS_CFG_TIMER_AS_PROC` 宏定义为0，则判断一下软件定时器任务是否处于睡眠状态，如果是则调用`tos_task_delay_abort()`函数恢复软件定时器任务运行，以便于更新它休眠的时间，因为此时是需要更新软件定时器任务睡眠的时间的，毕竟第一个软件定时器到期时间已经改变了。

5. 如果软件定时器任务处于挂起状态，表示并没有软件定时器在工作，现在插入了软件定时器，需要调用`tos_task_resume()`函数将软件定时器任务唤醒。

> 关于唤醒软件定时器任务是为了什么，我们在后续讲解

```
__STATIC__ void timer_place(k_timer_t *tmr)
{
    TOS_CPU_CPSR_ALLOC();
    k_list_t *curr;
```

```
    k_timer_t *iter = K_NULL;

    TOS_CPU_INT_DISABLE();

    tmr->expires += k_tick_count;

    TOS_LIST_FOR_EACH(curr, &k_timer_ctl.list) {
        iter = TOS_LIST_ENTRY(curr, k_timer_t, list);
        if (tmr->expires < iter->expires) {
            break;
        }
    }
    tos_list_add_tail(&tmr->list, curr);

    if (k_timer_ctl.list.next == &tmr->list) {
        // we are the first guy now
        k_timer_ctl.next_expires = tmr->expires;

#if TOS_CFG_TIMER_AS_PROC == 0u
        if (task_state_is_sleeping(&k_timer_task)) {
            tos_task_delay_abort(&k_timer_task);
        }
#endif
    }

#if TOS_CFG_TIMER_AS_PROC == 0u
    if (task_state_is_suspended(&k_timer_task)) {
        tos_task_resume(&k_timer_task);
    }
#endif

    TOS_CPU_INT_ENABLE();
}
```

停止软件定时器（外部函数）

> 在前文也提及停止软件定时器，但是那个timer_takeoff()函数是内部函数，而tos_timer_stop()函数是外部函数，可以被用户使用。

停止软件定时器的本质也是调用timer_takeoff()函数将软件定时器从软件定时器列表中移除，但是在调用这个函数之前还好做一些相关的判断，这样能保证系统的稳定性。

1. 对软件定时器控制块检测，如果软件定时器控制块为null，则直接返回错误代码。
2. 如果软件定时器状态为未使用状态TIMER_STATE_UNUSED，则直接返回错误代码K_ERR_TIMER_INACTIVE。
3. 如果软件定时器状态为TIMER_STATE_COMPLETED 或者是TIMER_STATE_STOPPED，则不需要停止软件定时器，因为这个软件定时器是未启动的。则直接返回错误代码K_ERR_TIMER_STOPPED。

4. 如果软件定时器状态为TIMER_STATE_RUNNING，就将软件定时器状态设置为停止状态TIMER_STATE_STOPPED，并且调用timer_takeoff()函数将软件定时器从软件定时器列表中移除。

```
__API__ k_err_t tos_timer_stop(k_timer_t *tmr)
{
    TOS_PTR_SANITY_CHECK(tmr);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!knl_object_verify(&tmr->knl_obj, KNL_OBJ_TYPE_TIMER)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    if (tmr->state == TIMER_STATE_UNUSED) {
        return K_ERR_TIMER_INACTIVE;
    }

    if (tmr->state == TIMER_STATE_COMPLETED ||
        tmr->state == TIMER_STATE_STOPPED) {
        return K_ERR_TIMER_STOPPED;
    }

    if (tmr->state == TIMER_STATE_RUNNING) {
        tmr->state = TIMER_STATE_STOPPED;
        timer_takeoff(tmr);
    }

    return K_ERR_NONE;
}
```

软件定时器的处理（在中断上下文环境）

我们知道，TencentOS tiny的软件定时器是可以在中断上下文环境来处理回调函数的，因此当软件定时器到期后，会在tos_tick_handler()函数中调用timer_update()来处理软件定时器。这个函数在每次tick中断到来的时候都会判断一下是否有软件定时器到期，如果有则去处理它。 过程如下：

1. 判断软件定时器的下一个到期时间k_timer_ctl.next_expires是否小于k_tick_count，如果是小于则表示还未到期，直接退出。

2. 反之则表示到期，此时要遍历软件定时器列表，找到所有到期的软件定时器，并处理他们。

> 因为有可能不只是一个软件定时器到期，很可能有多个定时器到期。当然啦，当软件定时器没到期的时候就会退出遍历。

3. 到期后的处理就是：调用timer_takeoff()函数将到期的软件定时器停止，如果是周期工作的定时器就调用timer_place()函数将它重新插入软件定时器列表中（它到期的相对时间就是软件定时器的周期值：tmr->expires = tmr->period）；如果是单次工作模式的软件定时器，就仅将软件定时器状态设置为TIMER_STATE_COMPLETED。

4. 调用软件定时器的回调函数处理相关的工作：`(*tmr->cb)(tmr->cb_arg)`

```
__KERNEL__ void timer_update(void)
{
    k_timer_t *tmr;
    k_list_t *curr, *next;

    if (k_timer_ctl.next_expires < k_tick_count) {
        return;
    }

    tos_knl_sched_lock();

    TOS_LIST_FOR_EACH_SAFE(curr, next, &k_timer_ctl.list) {
        tmr = TOS_LIST_ENTRY(curr, k_timer_t, list);
        if (tmr->expires > k_tick_count) {
            break;
        }

        // time's up
        timer_takeoff(tmr);

        if (tmr->opt == TOS_OPT_TIMER_PERIODIC) {
            tmr->expires = tmr->period;
            timer_place(tmr);
        } else {
            tmr->state = TIMER_STATE_COMPLETED;
        }

        (*tmr->cb)(tmr->cb_arg);
    }

    tos_knl_sched_unlock();
}
```

软件定时器的处理（在任务上下文环境）

> 关于使用软件定时器任务处理回调函数（即回调函数执行的上下文环境是任务），则必须打开 `TOS_CFG_TIMER_AS_PROC` 宏定义。

创建软件定时器任务

既然是软件定时器任务，那么就必须创建软件定时器任务，那么这个任务将在`timer_init()`函数中被创建，它是一个内核调用的函数，在内核初始化时就被调用（在`tos_knl_init()`函数中调用）。

> 创建软件定时器任务也是跟创建其他任务没啥差别，都是通过`tos_task_create()`函数创建，软件定时器任务控制块、任务主体、优先级、任务栈起始地址与大小等都在前面的数据结构中指定了，任务的名字为"timer"。

```
__KERNEL__ k_err_t timer_init(void)
{
#if TOS_CFG_TIMER_AS_PROC > 0u
    return K_ERR_NONE;
#else
    return tos_task_create(&k_timer_task,
                                                "timer",
                                                timer_task_entry,
                                                K_NULL,
                                                k_timer_task_prio,
                                                k_timer_task_stk_addr,
                                                k_timer_task_stk_size,
                                                0);
#endif
}
```

软件定时器任务主体

软件定时器任务的主体也是一个while (K_TRUE)循环，在循环中处理对应的事情。

1. 调用timer_next_expires_get()函数获取软件定时器列表中的下一个到期时间，并且更新 next_expires 的值。

> 注意：这里的时间已经在函数内部转换为相对到期时间，比如10分钟后闹钟叫我起床，而不是7:10分闹钟叫我起床）

2. 根据next_expires的值，判断一下软件定时器任务应该休眠多久，在多久后到期时才唤醒软件定时器任务并且处理回调函数。也就是说，软件定时器任务在软件定时器没有到期的时候是不会被唤醒的，都是处于休眠状态，调用tos_task_delay()函数将任务进入休眠状态，此时任务会被挂载到系统的延时（时基）列表中。

> 注意：如果next_expires的值为TOS_TIME_FOREVER，则不是休眠而是直接挂起，因为挂起状态的任务对调度器而言是不可见的，这样子的处理效率更高~挂起任务的函数是tos_task_suspend()。

3. 任务如果被唤醒了，或者被恢复运行了，则表明软件定时器到期了或者有新的软件定时器插入列表了，那么在唤醒之后就要判断一下是哪种情况，如果是到期了则处理对应的回调函数：首先调用 timer_takeoff()函数将到期的软件定时器停止，如果是周期工作的定时器就调用timer_place()函数将它重新插入软件定时器列表中（它到期的相对时间就是软件定时器的周期值：tmr->expires = tmr->period）；如果是单次工作模式的软件定时器，就仅将软件定时器状态设置为 TIMER_STATE_COMPLETED。（这里也是会遍历软件定时器列表以处理所有到期的软件定时器）

4. 最后将调用软件定时器的回调函数处理相关的工作：(*tmr->cb)(tmr->cb_arg)。

5. 如果定时器还未到期，并且软件定时器任务被唤醒了，那么就表示有新的软件定时器插入列表了，此时要更新一下任务的睡眠时间，因为软件定时器任务主体是一个while循环，还是会回到 timer_next_expires_get()函数中重新获取下一个唤醒任务的时间的。

> 注意：软件定时器的处理都是在锁调度器中处理的，就是为了避免其他任务打扰回调函数的执行。

```
__STATIC__ void timer_task_entry(void *arg)
{
    k_timer_t *tmr;
    k_list_t *curr, *next;
    k_tick_t next_expires;

    arg = arg; // make compiler happy
    while (K_TRUE) {
        next_expires = timer_next_expires_get();
        if (next_expires == TOS_TIME_FOREVER) {
            tos_task_suspend(K_NULL);
        } else if (next_expires > (k_tick_t)0u) {
            tos_task_delay(next_expires);
        }

        tos_knl_sched_lock();

        TOS_LIST_FOR_EACH_SAFE(curr, next, &k_timer_ctl.list) {
            tmr = TOS_LIST_ENTRY(curr, k_timer_t, list);
            if (tmr->expires > k_tick_count) { // not yet
                break;
            }

            // time's up
            timer_takeoff(tmr);

            if (tmr->opt == TOS_OPT_TIMER_PERIODIC) {
                tmr->expires = tmr->period;
                timer_place(tmr);
            } else {
                tmr->state = TIMER_STATE_COMPLETED;
            }

            (*tmr->cb)(tmr->cb_arg);
        }

        tos_knl_sched_unlock();
    }
}
```

获取软件定时器下一个到期时间

`timer_next_expires_get()`就是用于获取软件定时器下一个到期时间，如果软件定时器到期时间是`TOS_TIME_FOREVER`，就返回`TOS_TIME_FOREVER`，如果下一个到期时间小于`k_tick_count`则直接返回0，表示已经到期了，可以直接处理它，而如果是其他值，则需要减去`k_tick_count`，将其转变为相对值，因为调用这个函数就是为了知道任务能休眠多少时间。

> 打个比方，我7点醒来了，而7:10分的闹钟才会响，那么我就能再睡10分钟，就是这个道理。

```
__KERNEL__ k_tick_t timer_next_expires_get(void)
{
    TOS_CPU_CPSR_ALLOC();
    k_tick_t next_expires;

    TOS_CPU_INT_DISABLE();

    if (k_timer_ctl.next_expires == TOS_TIME_FOREVER) {
        next_expires = TOS_TIME_FOREVER;
    } else if (k_timer_ctl.next_expires <= k_tick_count) {
        next_expires = (k_tick_t)0u;
    } else {
        next_expires = k_timer_ctl.next_expires - k_tick_count;
    }

    TOS_CPU_INT_ENABLE();
    return next_expires;
}
```

喜欢就关注我吧！

相关代码可以在公众号后台回复“19”获取。