title: 【TencentOS tiny】深度源码分析（7）——事件 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-12 23:18:54 img: coverImg: password: summary: tags: - TencentOS tiny - RTOS - 操作系统 - 物联网 categories: - 操作系统 - TencentOS tiny

引言

大家在裸机编程中很可能经常用到flag这种变量，用来标志一下某个事件的发生，然后在循环中判断这些标志是否发生，如果是等待多个事件的话，还可能会if((xxx_flag)&&(xxx_flag))这样子做判断。当然，如果聪明一点的同学就会拿flag的某些位做标志，比如这个变量的第一位表示A事件，第二位表示B事件，当这两个事件都发生的时候，就判断flag&0x03的值是多少，从而判断出哪个事件发生了。

但在操作系统中又将如何实现呢？

事件

在操作系统中，事件是一种内核资源，主要用于任务与任务间、中断与任务间的同步，不提供数据传输功能！

与使用信号量同步有细微的差别：事件它可以实现一对多，多对多的同步。即一个任务可以等待多个事件的发生：可以是任意一个事件发生时唤醒任务进行事件处理；也可以是几个事件都发生后才唤醒任务进行事件处理。同样，也可以是多个任务同步多个事件。

每一个事件组只需要极少的RAM空间来保存事件旗标，一个事件（控制块）中包含了一个旗标，这个旗标的每一位表示一个“事件”，旗标存储在一个k_event_flag_t类型的变量中（名字叫flag，旗标简单理解就是事件标记变量），该变量在事件控制块中被定义，每一位代表一个事件，任务通过“逻辑与”或“逻辑或”与一个或多个事件建立关联，在事件发生时任务将被唤醒。

- 事件“逻辑或”是独立型同步，指的是任务所等待的若干事件中任意一个事件发生即可被唤醒；
- 事件“逻辑与”则是关联型同步，指的是任务所等待的若干事件中全部都发生时才被唤醒。

事件是一种实现任务间通信的机制，可用于实现任务间的同步，但事件无数据传输。多任务环境下，任务、中断之间往往需要同步操作，一个事件发生会告知等待中的任务，即形成一个任务与任务、中断与任务间的同步。

事件无排队性，即多次向任务设置同一事件(如果任务还未来得及读走)，等效于只设置一次。

此外事件可以提供一对多、多对多的同步操作。

- 一对多同步模型：一个任务等待多个事件的触发，这种情况是比较常见的；
- 多对多同步模型：多个任务等待多个事件的触发，任务可以通过设置事件位来实现事件的触发和等待操作。

事件数据结构

事件控制块

TencentOS tiny 通过事件控制块操作事件，其数据类型为k_event_t，事件控制块由多个元素组成。

- pend_obj有点类似于面向对象的继承，继承一些属性，里面有描述内核资源的类型（如互斥锁、队列、互斥量等，同时还有一个等待列表list）。
- flag是旗标，一个32位的变量，因此每个事件控制块最多只能标识32个事件发生！

```
typedef struct k_event_st {
    pend_obj_t      pend_obj;
    k_event_flag_t  flag;
} k_event_t;
```

任务控制块与事件相关的数据结构

```
typedef struct k_task_st {
        ...
    k_opt_t             opt_event_pend;     /**< 等待事件的的操作类型：
TOS_OPT_EVENT_PEND_ANY 、 TOS_OPT_EVENT_PEND_ALL */
    k_event_flag_t      flag_expect;        /**< 期待发生的事件 */
    k_event_flag_t     *flag_match;         /**< 等待到的事件（匹配的事件） */
        ...
} k_task_t;
```

与事件相关的宏定义

在tos_config.h中，配置事件开关的宏定义是TOS_CFG_EVENT_EN

```
#define TOS_CFG_EVENT_EN            1u
```

在tos_event.h中，存在一些宏定义是用于操作事件的（opt选项）：

```
// if we are pending an event, for any flag we expect is set is ok, this flag
should be passed to tos_event_pend
#define TOS_OPT_EVENT_PEND_ANY          (k_opt_t)0x0001

// if we are pending an event, must all the flag we expect is set is ok, this flag
should be passed to tos_event_pend
#define TOS_OPT_EVENT_PEND_ALL          (k_opt_t)0x0002

#define TOS_OPT_EVENT_PEND_CLR          (k_opt_t)0x0004
```

- TOS_OPT_EVENT_PEND_ANY：任务在等待任意一个事件发生，即“逻辑或”！
- TOS_OPT_EVENT_PEND_ALL：任务在等待所有事件发生，即“逻辑与”！

- TOS_OPT_EVENT_PEND_CLR：清除等待到的事件旗标，可以与TOS_OPT_EVENT_PEND_ANY、TOS_OPT_EVENT_PEND_ALL混合使用（通过“|”运算符）。

除此之外还有一个枚举类型的数据结构，用于发送事件时的选项操作，可以在发送事件时清除事件旗标的其他位（即覆盖，影响其他事件），也可以保持原本旗标中的其他位（不覆盖，不影响其他事件）。

```
typedef enum opt_event_post_en {
    OPT_EVENT_POST_KEP,
    OPT_EVENT_POST_CLR,
} opt_event_post_t;
```

创建事件

系统中每个事件都有对应的事件控制块，事件控制块中包含了事件的所有信息，比如它的等待列表、它的资源类型，以及它的事件旗标值，那么可以想象一下，创建事件的本质是不是就是对事件控制块进行初始化呢？很显然就是这样子的。因为在后续对事件的操作都是通过事件控制块来操作的，如果控制块没有信息，那怎么能操作嘛~

创建事件函数是tos_event_create()，传入一个事件控制块的指针*event，除此之外还可以指定事件初始值init_flag。

事件的创建实际上就是调用pend_object_init()函数将事件控制块中的event->pend_obj成员变量进行初始化，它的资源类型被标识为PEND_TYPE_EVENT。然后将event->flag成员变量设置为事件旗标初始值init_flag。

```
__API__ k_err_t tos_event_create(k_event_t *event, k_event_flag_t init_flag)
{
    TOS_PTR_SANITY_CHECK(event);

    pend_object_init(&event->pend_obj, PEND_TYPE_EVENT);
    event->flag = init_flag;
    return K_ERR_NONE;
}
```

销毁事件

事件销毁函数是根据事件控制块直接销毁的，销毁之后事件的所有信息都会被清除，而且不能再次使用这个事件，当事件被销毁时，其等待列表中存在任务，系统有必要将这些等待这些任务唤醒，并告知任务事件已经被销毁了PEND_STATE_DESTROY。然后产生一次任务调度以切换到最高优先级任务执行。

TencentOS tiny 对事件销毁的处理流程如下：

1. 调用pend_is_nopending()函数判断一下是否有任务在等待事件
2. 如果有任务在等待事件则调用pend_wakeup_all()函数将这些任务唤醒，并且告知等待任务事件已经被销毁了（即设置任务控制块中的等待状态成员变量pend_state为PEND_STATE_DESTROY）。

3. 调用pend_object_deinit()函数将事件控制块中的内容清除，最主要的是将控制块中的资源类型设置为PEND_TYPE_NONE，这样子就无法使用这个事件了。
4. 将event->flag成员变量恢复为默认值0。
5. 进行任务调度knl_sched()

注意：如果事件控制块的RAM是由编译器静态分配的，所以即使是销毁了事件，这个内存也是没办法释放的。当然你也可以使用动态内存为事件控制块分配内存，只不过在销毁后要将这个内存释放掉，避免内存泄漏。

```
__API__ k_err_t tos_event_destroy(k_event_t *event)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(event);

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&event->pend_obj, PEND_TYPE_EVENT)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    TOS_CPU_INT_DISABLE();

    if (!pend_is_nopending(&event->pend_obj)) {
        pend_wakeup_all(&event->pend_obj, PEND_STATE_DESTROY);
    }

    pend_object_deinit(&event->pend_obj);
    event->flag = (k_event_flag_t)0u;

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}
```

等待事件

tos_event_pend()函数用于获取事件，通过这个函数，就可以知道事件旗标中的哪一位被置1，即哪一个事件发生了，然后任务可以对等待的事件指定“逻辑与”、“逻辑或”进行等待操作（opt_pend选项）。

并且这个函数实现了等待超时机制，且仅当任务等待的事件发生时，任务才能等待到事件。当事件未发生的时候，等待事件的任务会进入阻塞态，阻塞时间timeout由用户指定，在这段时间中，如果事件一直没发生，该任务将保持阻塞状态以等待事件发生。当其它任务或中断服务程序往其等待的事件旗标设置对应的标志位，该任务将自动由阻塞态转为就绪态。当任务等待的时间超过了指定的阻塞时间，即使事件还未发生，任务也会自动从阻塞态转移为就绪态。这样子很有效的体现了操作系统的实时性。

任务获取了某个事件时，可以选择清除事件操作。

等待事件的操作不允许在中断上下文环境运行！

等待事件的过程如下：

1. 首先检测传入的参数是否正确。，注意opt_pend的选项必须存在TOS_OPT_EVENT_PEND_ALL 或者TOS_OPT_EVENT_PEND_ANY 之一，且二者不允许同时存在（互斥）。
2. 调用event_is_match()函数判断等待的事件是否已发生（即任务等待的事件与事件控制块中的旗标是否匹配）。
3. 在event_is_match()函数中会根据等待选项opt_pend是等待任意一个事件（TOS_OPT_EVENT_PEND_ANY）还是等待所有事件（TOS_OPT_EVENT_PEND_ANY）做出是否匹配的判断，如果是匹配了则返回K_TRUE，反之返回K_FALSE，同时等待到的事件通过flag_match变量返回（已发生匹配）。对于等待所有时间的选项，当且仅当所有事件都发生是才算匹配：(event & flag_expect) == flag_expect)，对于等待任意一个事件的选项，有其中一个事件发生都算匹配：(event & flag_expect)。
4. 如果事件未发生则可能会阻塞当前获取的任务，看一下用户指定的阻塞时间timeout是否为不阻塞TOS_TIME_NOWAIT，如果不阻塞则直接返回K_ERR_PEND_NOWAIT错误代码。
5. 如果调度器被锁了knl_is_sched_locked()，则无法进行等待操作，返回错误代码K_ERR_PEND_SCHED_LOCKED，毕竟需要切换任务，调度器被锁则无法切换任务。
6. 将任务控制块中关于事件的变量设置一下，即设置任务期望等待的事件k_curr_task->flag_expect，任务匹配的事件k_curr_task->flag_match，以及任务等待事件的选项k_curr_task->opt_event_pend。
7. 调用pend_task_block()函数将任务阻塞，该函数实际上就是将任务从就绪列表中移除k_rdyq.task_list_head[task_prio]，并且插入到等待列表中object->list，如果等待的时间不是永久等待TOS_TIME_FOREVER，还会将任务插入时间列表中k_tick_list，阻塞时间为timeout，然后进行一次任务调度knl_sched()。
8. 当程序能继续往下执行时，则表示任务等待到事件，又或者等待发生了超时，任务就不需要等待事件了，此时将任务控制块中的内容清空，即清空任务期望等待的事件k_curr_task->flag_expect，任务匹配的事件k_curr_task->flag_match，以及任务等待事件的选项k_curr_task->opt_event_pend，同时还调用pend_state2errno()函数获取一下任务的等待状态，看一下是哪种情况导致任务恢复运行，并且将结果返回给调用等待事件函数的任务。

注意：当等待事件的任务能从阻塞中恢复运行，也不一定是等待到事件发生，也有可能是发生了超时，因此在写程序的时候必须要判断一下等待的事件状态，如果是K_ERR_NONE则表示获取成功！

代码如下：

```
__STATIC__ int event_is_match(k_event_flag_t event, k_event_flag_t flag_expect,
k_event_flag_t *flag_match, k_opt_t opt_pend)
{
    if (opt_pend & TOS_OPT_EVENT_PEND_ALL) {
        if ((event & flag_expect) == flag_expect) {
            *flag_match = flag_expect;
            return K_TRUE;
        }
    } else if (opt_pend & TOS_OPT_EVENT_PEND_ANY) {
        if (event & flag_expect) {
            *flag_match = event & flag_expect;
            return K_TRUE;
        }
    }
    return K_FALSE;
```

```
}

__API__ k_err_t tos_event_pend(k_event_t *event, k_event_flag_t flag_expect,
k_event_flag_t *flag_match, k_tick_t timeout, k_opt_t opt_pend)
{
    TOS_CPU_CPSR_ALLOC();

    TOS_PTR_SANITY_CHECK(event);
    TOS_PTR_SANITY_CHECK(flag_match);
    TOS_IN_IRQ_CHECK();

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&event->pend_obj, PEND_TYPE_EVENT)) {
        return K_ERR_OBJ_INVALID;
    }
#endif

    if (!(opt_pend & TOS_OPT_EVENT_PEND_ALL) && !(opt_pend &
TOS_OPT_EVENT_PEND_ANY)) {
        return K_ERR_EVENT_PEND_OPT_INVALID;
    }

    if ((opt_pend & TOS_OPT_EVENT_PEND_ALL) && (opt_pend &
TOS_OPT_EVENT_PEND_ANY)) {
        return K_ERR_EVENT_PEND_OPT_INVALID;
    }

    TOS_CPU_INT_DISABLE();

    if (event_is_match(event->flag, flag_expect, flag_match, opt_pend)) {
        if (opt_pend & TOS_OPT_EVENT_PEND_CLR) { // destroy the bridge after get
across the river
            event->flag = (k_event_flag_t)0u;
        }
        TOS_CPU_INT_ENABLE();
        return K_ERR_NONE;
    }

    if (timeout == TOS_TIME_NOWAIT) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_NOWAIT;
    }

    if (knl_is_sched_locked()) {
        TOS_CPU_INT_ENABLE();
        return K_ERR_PEND_SCHED_LOCKED;
    }

    k_curr_task->flag_expect      = flag_expect;
    k_curr_task->flag_match       = flag_match;
    k_curr_task->opt_event_pend   = opt_pend;

    pend_task_block(k_curr_task, &event->pend_obj, timeout);
```

```
    TOS_CPU_INT_ENABLE();
    knl_sched();

    k_curr_task->flag_expect      = (k_event_flag_t)0u;
    k_curr_task->flag_match       = (k_event_flag_t *)K_NULL;
    k_curr_task->opt_event_pend   = (k_opt_t)0u;

    return pend_state2errno(k_curr_task->pend_state);
}
```

发送事件

TencentOS tiny 提供两个函数发送事件，分别是：tos_event_post()与tos_event_post_keep()，两个函数本质上都是调用同一个函数event_do_post()去实现发送事件的操作的，只不过选项是不同而已，使用tos_event_post()函数会覆盖写入指定的事件，可能影响其他已发生的事件，而tos_event_post_keep()函数则可以保持其他事件位不改变的同时发生事件，在实际情况中后者更常用。

此函数用于将已发生的事件写入事件旗标中指定的位，当对应的位被置1之后，等待事件的任务将可能被恢复，此时需要遍历等待在事件对象上的事件等待列表，判断是否有任务期望的事件与当前事件旗标的值匹配，如果有，则唤醒该任务。

简单来说，就是设置自己定义的事件标志位为1，并且看看有没有任务在等待这个事件，有的话就唤醒它。

TencentOS tiny 中设计的很好的地方就是简单与低耦合，这两个api接口本质上都是调用event_do_post()函数去发生事件，只是通过opt_post参数不同选择不同的处理方法。

在event_do_post()函数中的处理也是非常简单明了的，其执行思路如下：

1. 首先判断一下发生事件的方式opt_post，如果是OPT_EVENT_POST_KEP则采用或运算“|”写入事件旗标，否则直接赋值。
2. 使用TOS_LIST_FOR_EACH_SAFE遍历等待在事件对象上的事件等待列表，通过event_is_match()函数判断是否有任务期望的事件与当前事件旗标的值匹配，如果有则调用pend_task_wakeup()函数唤醒对应的任务。
3. 如果唤醒的等待任务指定了清除对应的事件，那么将清除事件的旗标值。
4. 最后进行一次任务调度knl_sched()。

```
__STATIC__ k_err_t event_do_post(k_event_t *event, k_event_flag_t flag,
opt_event_post_t opt_post)
{
    TOS_CPU_CPSR_ALLOC();
    k_task_t *task;
    k_list_t *curr, *next;

#if TOS_CFG_OBJECT_VERIFY_EN > 0u
    if (!pend_object_verify(&event->pend_obj, PEND_TYPE_EVENT)) {
        return K_ERR_OBJ_INVALID;
    }
#endif
```

```
    if (opt_post == OPT_EVENT_POST_KEP) {
        event->flag |= flag;
    } else {
        event->flag = flag;
    }

    TOS_CPU_INT_DISABLE();

    TOS_LIST_FOR_EACH_SAFE(curr, next, &event->pend_obj.list) {
        task = TOS_LIST_ENTRY(curr, k_task_t, pend_list);

        if (event_is_match(event->flag, task->flag_expect, task->flag_match, task->opt_event_pend)) {
            pend_task_wakeup(TOS_LIST_ENTRY(curr, k_task_t, pend_list), PEND_STATE_POST);

            // if anyone pending the event has set the TOS_OPT_EVENT_PEND_CLR, then no wakeup for the others pendig for the event.
            if (task->opt_event_pend & TOS_OPT_EVENT_PEND_CLR) {
                event->flag = (k_event_flag_t)0u;
                break;
            }
        }
    }

    TOS_CPU_INT_ENABLE();
    knl_sched();

    return K_ERR_NONE;
}

__API__ k_err_t tos_event_post(k_event_t *event, k_event_flag_t flag)
{
    TOS_PTR_SANITY_CHECK(event);

    return event_do_post(event, flag, OPT_EVENT_POST_CLR);
}

__API__ k_err_t tos_event_post_keep(k_event_t *event, k_event_flag_t flag)
{
    TOS_PTR_SANITY_CHECK(event);

    return event_do_post(event, flag, OPT_EVENT_POST_KEP);
}
```

喜欢就关注我吧！

相关代码可以在公众号后台回复“19”获取。