ToRTOS RTOS API 文档

本文档覆盖当前内核公开/半公开 API，列出函数原型、参数、返回值、行为、限制与典型用法。
本次补充内容：

- 补全互斥量 / 消息队列已实现接口。
- 增加错误码说明、ISR 安全性表、线程/IPC 行为细节、返回值一致性。
- 拆分 / 合并重复编号并补充未来规划与最佳实践。

0. 约定与通用说明

项	说明
基本类型	`t_uint32_t` 等定义于 `tdef.h`
返回值	统一使用 `t_status`
优先级	由 `TO_LOWER_PRIORITY_NUM_HIGHER_PRIORITY` 决定数值大小和优先级的关系
Tick	全局节拍，频率由 `TO_TICK` 定义（Hz）
临界区	内核内部使用 `t_irq_disable/enable`；外部调用无需再包裹除非做多步复合操作
线程上下文	阻塞 / 可能调度的 API 只能在线程上下文调用，不可在中断中调用
ISR 中允许	只能使用非阻塞、纯查询或唤醒型函数（见“12. ISR 可调用性表”）
定时器回调上下文	当前实现：在 `sysTick` 中（中断上下文）执行

0.1 错误码语义

状态	含义	典型来源
T_OK	成功	正常路径
T_ERR	一般错误 / 资源不足 / 超时（当前信号量与消息队列超时也用此值，后续可能改用 T_TIMEOUT）	超时 / 满 / 空 / 逻辑失败
T_TIMEOUT	明确超时（预留，部分 API 尚未使用）	计划用于将来区分超时
T_BUSY	资源忙（保留）	未来互斥等
T_INVALID	参数非法	NULL / 越界 / 配置错误
T_NULL	空指针	传参为 NULL
T_DELETED	对象已删除或失效	IPC/线程已被删除
T_UNSUPPORTED	不支持命令	ctrl / 未来扩展

1. 启动 & 核心初始化

t_status_t t_tortos_init(void)

初始化调度器、定时器链表、空闲线程并打印启动横幅（如启用）。

- 必须在创建用户线程前调用（通常放在 main 中最早阶段，硬件初始化之后）。
- 返回：T_OK 或错误码。

void t_sched_start(void)

启动首次调度，切换到最高优先级就绪线程（不返回）。应在所有初始线程 `t_thread_startup` 之后调用。

- 注意：调用后主线程（main context）不再执行普通代码。

__weak void t_start_banner(void)

- 打印启动信息。可在用户代码中重写定制输出。

2. 线程管理

t_status_t t_thread_create_static(t_thread_entry_t entry, void *stackaddr, t_uint32_t stacksize, t_int8_t priority, void *arg, t_uint32_t time_slice, t_thread_t *thread)

初始化线程控制块，但不放入就绪队列。

- 参数：
  - entry 线程入口函数（`void (*)(void *arg)` 原型）

    - stackaddr 栈空间基地址（传入首地址，内部会按栈顶初始化）
    - stacksize 栈大小
    - priority 优先级（0 最高，值越大优先级越低）
    - arg 参数传递给entry
    - time_slice 时间片长度（调度轮转基准）
    - thread 线程对象指针（静态/全局存储）
- 失败条件：任一指针为空 / stacksize=0 / priority>=TO_THREAD_PRIORITY_MAX / time_slice=0

t_status_t t_thread_create(t_thread_entry_t entry, t_uint32_t stacksize, t_int8_t priority, void *arg, t_uint32_t time_slice, t_thread_t **thread_handle)

动态创建线程，自动分配线程控制块和栈空间。

- 参数：
    - entry 线程入口函数（`void (*)(void *arg)` 原型）
    - stacksize 栈大小
    - priority 优先级（0 最高，值越大优先级越低）
    - arg 参数传递给entry
    - time_slice 时间片长度（调度轮转基准）
    - thread_handle 返回创建的线程句柄
- 失败条件：entry为空 / stacksize=0 / priority>=TO_THREAD_PRIORITY_MAX / time_slice=0 / 内存不足

t_status_t t_thread_startup(t_thread_t *thread)

将已初始化线程加入就绪队列。

- 返回：T_NULL 空指针；T_ERR 线程已被删除；T_OK 成功。
- 内部设置：current_priority、剩余时间片、状态 READY。

t_status_t t_thread_delete(t_thread_t *thread)

将线程移出调度，并放入待删除链表，状态置 TERMINATED，等待 idle 清理。

- 可重复调用：若已 TERMINATED 返回 T_OK；已 DELETED 返回 T_ERR。

void t_cleanup_waiting_termination_threads(void)

由 idle 线程周期调用，遍历待删除链表，标记线程为 DELETED 并摘链。当前不释放栈与控制块（假设静态分配）。

t_status_t t_thread_restart(t_thread_t *thread)

仅在线程已被 `t_cleanup_waiting_termination_threads` 处理成 DELETED 后使用；重建栈上下文并重新 startup。

- 返回：T_NULL/T_ERR/T_OK。

void t_thread_sleep(t_uint32_t tick)

当前线程阻塞指定 tick。内部：

1. 移出就绪队列
2. 状态= SUSPEND
3. 配置其专属定时器启动
4. 触发调度

- 不返回状态；tick==0 等价于立即让出（但仍走定时器路径，建议调用 yield）。

void t_delay(t_uint32_t tick)

`t_thread_sleep` 简单封装。

void t_thread_rotate_same_prio(void)

协作式让出 CPU：将当前线程插入其优先级就绪链表尾部并触发调度；若本优先级只有该线程则直接返回。

void t_thread_exit(void)

线程主动结束（用于在线程函数 return 前安全退出）。流程：

- 删除自身（终止 + 加入待删除链表）
- 立即调度到下一线程（不返回）

t_status_t t_thread_ctrl(t_thread_t *thread, t_uint32_t cmd, void *arg)

控制/查询接口（已实现命令）：

- TO_THREAD_GET_STATUS: *(t_int32_t)*arg= status
- TO_THREAD_GET_PRIORITY: *(t_uint8_t)*arg= current_priority
- TO_THREAD_SET_PRIORITY: *(t_uint8_t)*arg 赋值并更新 number_mask
  未支持其他命令返回 T_UNSUPPORTED。

使用示例

```
#define THREAD_STACK_SIZE 512
#define THREAD_PRIORITY 10

t_thread thread1;
t_thread thread2;
t_thread thread3;

t_uint8_t thread1stack[THREAD_STACK_SIZE];
t_uint8_t thread2stack[THREAD_STACK_SIZE];
t_uint8_t thread3stack[THREAD_STACK_SIZE];

t_start_init(); //初始化start os

t_thread_create_static(thread1entry,
          thread1stack,
          THREAD_STACK_SIZE,
          10,
```

```
          NULL,
          10,
          &thread1);
t_thread_startup(&thread1);

t_thread_create_static(thread2entry,
          thread2stack,
          THREAD_STACK_SIZE,
          12,
          NULL,
          10,
          &thread2);
t_thread_startup(&thread2);

t_thread_create_static(thread3entry,
          thread3stack,
          THREAD_STACK_SIZE,
          15,
          NULL,
          10,
          &thread3);
t_thread_startup(&thread3);
t_sched_start(); //启动第一次调度

void thread1entry(void *arg) //线程入口
{
    while(1)
    {
    t_printf("Thread 1\r\n");
        t_mdelay(1000);
    }
}

void thread2entry(void *arg) //线程入口
{
    while(1)
    {
    t_printf("Thread 2\r\n");
        t_mdelay(1000);
    }
}

void thread3entry(void *arg) //线程入口
{
    while(1)
    {
    t_printf("Thread 3\r\n");
        t_mdelay(1000);
    }
}
```

3. 调度器内部接口（应用层尽量不要直接调用）

函数	说明
t_sched_init	初始化所有优先级队列与全局变量
t_sched_switch	若存在更高优先级 READY 线程则发起上下文切换
t_sched_remove_thread	从 READY 队列摘除，必要时清除位图
t_sched_insert_thread	插入 READY 队列并设置位图
t_thread_rotate_same_prio	同优先级轮转

4. 中断/CPU 相关

函数	说明
t_irq_disable	关中断返回原 PRIMASK
t_irq_enable(level)	恢复 PRIMASK
t_stack_init(entry, stack_top, arg)	构建初始上下文（PSR/PC/LR/寄存器清零）
t_first_switch_task(next)	首次上下文切换（历史拼写）
t_normal_switch_task(prev,next)	正常切换保存前线程栈并装载后线程栈
int **t_ffs(int v) 或** t_fls(int v)	查找最低/最高有效 1 位（1-based）；v=0 调用方需避免

5. 定时器与 Tick

函数	说明
t_timer_list_init	初始化定时器链表（level=1）
t_timer_init	初始化单个软件定时器
t_timer_start	计算 timeout_tick 并有序插入
t_timer_stop	从链表摘除
t_timer_ctrl	GET/SET 时间参数
t_tick_increase	SysTick ISR：全局 tick++ / 时间片处理 / 调用 t_timer_check
t_timer_check	把到期定时器移至临时表并执行回调
timeout_function	线程睡眠专用回调：标 READY + 触发调度
t_tick_get	获取全局 tick
t_mdelay / t_tick_from_ms	毫秒封装

注意：回调在中断执行；避免调用阻塞 API。

6. IPC 基础

结构：`t_ipc_t`

```
typedef struct {
    t_ipc_type_t   type;          /* IPC type */
    union {
        t_queue_pointers_t queue;     /* Used for queue */
        t_sema_data_t   sema;      /* Used for semaphore/mutex */
    } u;
    t_list_t     wait_list;      /* Thread wait list */
    t_uint16_t   msg_waiting;    /* Current item count or resource count */
    t_uint16_t   length;         /* Max number of items or max count */
    t_uint16_t   item_size;      /* Size of each item */
    t_uint8_t    status;         /* 1=valid, 0=deleted */
    t_uint8_t    mode;           /* FIFO / PRIO */
#if (TO_USING_STATIC_ALLOCATION && TO_USING_DYNAMIC_ALLOCATION)
    t_uint8_t   is_static_allocated;
#endif
} t_ipc_t;
```

函数	语义
t_ipc_delete	删除IPC对象，唤醒所有等待线程。

7. 信号量 Semaphore

函数	语义
t_sema_create_static / t_sema_create	初始化 count、挂起队列、策略
t_ipc_delete	唤醒全部等待者并失效对象
t_sema_recv	获取资源或阻塞（支持无限/有限超时/非阻塞）
t_sema_send	释放资源并按策略唤醒一个等待者

当前超时返回 T_ERR（后续可区分 T_TIMEOUT）。

使用示例

```
t_ipc_t sem1;

t_sema_create_static(1, 0, TO_IPC_FLAG_FIFO, &sem1); // 初始化信号量，初始值为0，FIFO方式

void thread1entry(void *arg)
{
  while(1)
  {
    if(T_OK !=t_sema_recv(&sem1, TO_WAITING_FOREVER)) // 等待信号量
    {
      t_printf("Thread 1: Waiting for semaphore...\r\n");
    }
```

```
    j++;
    if(j==255)
    j=0;
    t_printf("Thread 1: j = %d\r\n", j);
    t_mdelay(500);
  }
}

void thread2entry(void *arg)
{
  while(1)
  {
    i++;
    if(i==30){
      t_printf("Thread 2: delete semaphore...\r\n");
      t_ipc_delete(&sem1); // 删除信号量
    }
    if(i==255){
      i=0;
    }
    t_printf("Thread 2: i = %d\r\n", i);
    t_mdelay(1000);
  }
}

void thread3entry(void *arg)
{
  while(1)
  {
    T_DEBUG_LOG(TO_DEBUG_INFO, "Thread 3: k = %d\n", k);
    if(T_OK ==t_sema_send(&sem1)) // 释放信号量
    {
      t_printf("Thread 3: release semaphore...\r\n");
    }
    T_DEBUG_LOG(TO_DEBUG_INFO, "sem1.msg_waiting = %d\n", sem1.msg_waiting);
    k++;
    if(k==255){
      k=0;
    }
    t_delay(600);
  }
}
```

---

8. 互斥量 Mutex（已具备简单优先级继承）

普通互斥量

函数	说明
T_MUTEX_CREATE_STATIC(mode, mutex) / T_MUTEX_CREATE(mode, mutex_handle)	初始化普通互斥量，可设排队策略
T_MUTEX_DELETE(mutex)	唤醒等待者并恢复所有者原优先级
T_MUTEX_ACQUIRE(mutex, timeout)	获取互斥量；当高优先级等待低优先级持有者时提高持有者优先级（简单继承）
T_MUTEX_RELEASE(mutex)	释放互斥量并恢复优先级

递归互斥量

函数	说明
T_MUTEX_RECURSIVE_CREATE_STATIC(mode, mutex) / T_MUTEX_RECURSIVE_CREATE(mode, mutex_handle)	初始化递归互斥量，可设排队策略
T_MUTEX_RECURSIVE_DELETE(mutex)	唤醒等待者并恢复所有者原优先级
T_MUTEX_RECURSIVE_ACQUIRE(mutex, timeout)	获取递归互斥量，支持同一线程多次获取；优先级继承
T_MUTEX_RECURSIVE_RELEASE(mutex)	递归计数减，归零时释放并恢复优先级

注意：继承恢复依赖 `original_prio` 保存；同时无完整链式继承与死锁检测。

使用示例

```
t_ipc_t mutex1;

T_MUTEX_RECURSIVE_CREATE_STATIC(TO_IPC_FLAG_FIFO, &mutex1);

/*优先级关系：thread1 < thread2 < thread3 */
void thread1entry(void *arg) /* High priority (等待互斥量) */
{
    static int phase = 0;
    while (1)
    {
        if (phase == 0)
        {
            t_mdelay(100); /* 先让低优先级线程获取互斥量制造反转 */
            /* Lines 359-366 omitted */
            }
        }
        else
        {
            /* 后续简单演示重复获取/释放 */
            /* Lines 371-375 omitted */
            t_mdelay(600);
        }
        j++;
        if (j == 255) j = 0;
```

```
        t_mdelay(50);
    }
}

void thread2entry(void *arg) /* Medium priority (制造 CPU 干扰) */
{
    while (1)
    {
        i++;
        if (i % 50 == 0)
            t_printf("MED  : running i=%d\n", i);
        if (i == 255) i = 0;
        /* 不使用互斥量，纯粹占用时间片 */
        t_mdelay(40);
    }
}

void thread3entry(void *arg) /* Low priority (先获取互斥量并长时间占用) */
{
    static int once = 0;
    t_uint8_t base_prio_saved = 0;
    while (1)
    {
        if (once == 0)
        {
            if (T_OK == T_MUTEX_ACQUIRE(&mutex1, TO_WAITING_FOREVER))
            /* Lines 405-428 omitted */
            }
        }
        else
        {
            /* 之后偶尔再占用一下，验证递归外正常路径 */
            /* Lines 433-438 omitted */
            t_mdelay(200);
        }

        k++;
        if (k == 255) k = 0;
        t_mdelay(10);
    }
}
```

9. 消息队列 Message Queue

已实现：初始化、删除、阻塞/非阻塞发送、紧急发送（头部插入）、阻塞接收。
内部：固定大小消息节点 + 单链自由链表 + FIFO 队列。
超时时间使用线程私有定时器；超时亦返回 T_ERR（计划区分 T_TIMEOUT）。

函数	说明
t_queue_create_static / t_queue_create	初始化池 / 构建自由链表
t_ipc_delete	唤醒所有收发等待者并失效对象
t_queue_send	阻塞发送（池满时挂起）
t_queue_send	非阻塞（池满返回 T_ERR）
t_queue_send	头部插入（高优先级消费）
t_queue_recv	阻塞 / 非阻塞接收

使用示例

```
typedef struct
{
    t_uint8_t data[4];
} msg_t;

#define MSG_QUEUE_SIZE 10
#define MSG_POOL_SIZE  sizeof(msg_t) * MSG_QUEUE_SIZE + sizeof(void*) *
MSG_QUEUE_SIZE // 10条消息

t_ipc_t msgqueue1;

t_queue_create_static(msgpool, MSG_QUEUE_SIZE, sizeof(msg_t), TO_IPC_FLAG_FIFO,
&msgqueue1);

void thread1entry(void *arg)
{
  msg_t msg;
    while(1)
    {
    if(T_OK !=t_queue_recv(&msgqueue1, &msg, TO_WAITING_FOREVER)) // 等待消息队列
    {
      t_printf("Thread 1: Waiting for message...\r\n");
    }

    t_printf("Thread 1: received data[0] = %d\r\n", msg.data[0]);

        t_mdelay(600);
    }
}

void thread2entry(void *arg)
{
  msg_t msg;
    while(1)
    {
        i++;
    msg.data[0] = i;
    t_printf("Thread 2: urgent data[0] = %d\r\n", msg.data[0]);
    t_queue_send(&msgqueue1, &msg, TO_WAITING_FOREVER); // 紧急发送
        if(i==30){
      /* Lines 527-528 omitted */
```

```
      t_ipc_delete(&msgqueue1); // 删除消息队列
    }
        if(i==255){
            /* Lines 531-533 omitted */
        t_mdelay(900);
    }
}

void thread3entry(void *arg)
{
  msg_t msg;
  t_status_t err;
  while(1)
  {
    k++;
    if(k==255){
      k=0;
    }
    msg.data[0] = k;
    t_printf("Thread 3: send data[0] = %d\r\n", msg.data[0]);
    err = t_queue_send(&msgqueue1, &msg, TO_WAITING_NO);
    if( err == T_OK )
    {
      /* Lines 552-557 omitted */
    t_delay(300);
  }
}
```

10. 打印与调试

函数	说明
t_printf	轻量格式化输出（非线程安全）
T_DEBUG_LOG	条件编译日志宏（INFO/WARN/ERR）

11. 线程状态机

状态	说明	进入	退出
INIT	已初始化未调度	t_thread_create_static	startup
READY	可运行	startup/超时/IPC释放	被调度 / 阻塞 / 删除
RUNNING	正在执行	调度器切换	时间片到/阻塞/删除
SUSPEND	等待事件/定时器/IPC	sleep / recv 阻塞	事件满足/超时
TERMINATED	待清理	delete / exit	idle 清理
DELETED	资源已回收（控制块保留）	idle 清理	restart

12. ISR 可调用性表

API	ISR 可用	说明
t_tick_increase	是	典型 SysTick
t_tick_get	是	只读
t_printf	视实现	若使用阻塞 UART 需谨慎
t_sema_send	否(当前)	内部可能调度；若需支持需改为延迟调度
t_sema_recv	否	可能阻塞
t_mutex_send_base / t_mutex_recv_base	否	可能阻塞或调度
t_queue_send / t_queue_recv	否	可能阻塞
t_timer_start / t_timer_stop	否(建议线程)	需短临界区；若需支持 ISR 可局部裁剪
t_thread_* (除查询)	否	涉及调度/阻塞
t_ffs / t_fls	是	纯计算
t_irq_disable / t_irq_enable	是	底层操作

13. 典型使用流程（简要）

```
hw_init();
t_start_init();

t_thread_create_static(entry1, stack1, sizeof(stack1), 5, NULL, 10, &t1);
t_thread_startup(&t1);

t_thread_create_static(entry2, stack2, sizeof(stack2), 6, NULL, 10, &t2);
t_thread_startup(&t2);

t_sched_start(); /* 不返回 */
```

SysTick:

```
void SysTick_Handler(void) {
    t_tick_increase();
}
```

14. 设计要点与局限

方面	当前实现	局限 / 未来
调度	位图 + O(1) 取最高优先级	无优先级动态调整
时间片	固定每线程 time_slice	暂无自适应/统计
定时器	单层有序链表 O(n) 插入	计划：多层 / 小根堆
IPC	信号量/互斥量/消息队列	未支持事件集/管道
优先级继承	简单单层	缺少链式、动态反转处理
内存	静态/手工分配	未集成堆/内存池
调试	简单日志	缺少断言/统计/水位线
安全	依赖正确使用	未检测栈溢出

15. 最佳实践

- 线程栈大小留裕量（建议 > 256B 简单任务）。
- 避免在回调（中断上下文）中长时间计算；仅设置标志或唤醒线程。
- 统一封装驱动中断 → 线程通知：中断里投放 semaphore 或 queue（将来提供 _from_isr 变体）。
- 定期在空闲线程中加入轻量监控（如统计 RUNNING 次数、检测 READY 队列一致性）。

16. 未来扩展计划

功能	优先级
区分 T_TIMEOUT	高
Mutex 完整优先级继承	高
Tickless 低功耗	中
事件标志组	中
消息队列零拷贝优化	中
栈使用水位线	中
单元测试 / 仿真（QEMU）	中
统计（CPU 使用率 / 上下文切换计数）	低

17. 变更记录（文档）

如发现描述与实现不一致，请以源码为准并提交 Issue。

Happy hacking with ToRTOS!