StaRT 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	`s_uint32_t` 等定义于 `sdef.h`
返回值	统一使用 `s_status`
优先级	数值越小优先级越高（0 为最高）
Tick	全局节拍，频率由 `START_TICK` 定义（Hz）

临界区	内核内部使用 `s_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
S_OK	成功	正常路径
S_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 S_TIMEOUT）	超时 / 满 / 空 / 逻辑失败
S_TIMEOUT	明确超时（预留，部分 API 尚未使用）	计划用于将来区分超时
S_BUSY	资源忙（保留）	未来互斥等

S_INVALID	参数非法	NULL / 越界 / 配置错误
S_NULL	空指针	传参为 NULL
S_DELETED	对象已删除或失效	IPC/线程已被删除
S_UNSUPPORTED	不支持命令	ctrl / 未来扩展

---

1. 启动 & 核心初始化

s_status s_start_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：S_OK 或错误码。

void rtos_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 `s_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 s_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

s_status s_thread_init(s_pthread thread, void *entry, void *stackaddr, s_uint32_t stacksize, s_int8_t priority, s_uint32_t tick)

初始化线程控制块，但不放入就绪队列。

- 参数：
- thread 线程对象指针（静态/全局存储）
- entry 线程入口函数（`void (*)(void)` 原型习惯）
- 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）
- tick 时间片长度（调度轮转基准）

- 失败条件：任一指针为空 / stacksize=0 /

  priority>=START_THREAD_PRIORITY_MAX / tick=0

s_status s_thread_startup(s_pthread thread)

将已初始化线程加入就绪队列。

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

s_status s_thread_delete(s_pthread 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 S_OK；已 DELETED 返回

  S_ERR。

void s_cleanup_defunct_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）。

s_status s_thread_restart(s_pthread thread)

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

- 返回：S_NULL/S_ERR/S_OK。

void s_thread_sleep(s_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 s_delay(s_uint32_t tick)

`s_thread_sleep` 简单封装。

void s_thread_yield(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 s_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）

s_status s_thread_ctrl(s_pthread thread, s_uint32_t cmd, void *arg)

控制/查询接口（已实现命令）：

- START_THREAD_GET_STATUS: *(s_int32_t*)arg= status
- START_THREAD_GET_PRIORITY: *(s_uint8_t*)arg= current_priority

- START_THREAD_SET_PRIORITY: *(s_uint8_t*)arg 赋值并更新 number_mask

  未支持其他命令返回 S_UNSUPPORTED。

使用示例

```
#define THREAD_STACK_SIZE 512
#define START_THREAD_PRIORITY 10

s_thread thread1;
s_thread thread2;
s_thread thread3;

s_uint8_t thread1stack[THREAD_STACK_SIZE];
s_uint8_t thread2stack[THREAD_STACK_SIZE];
s_uint8_t thread3stack[THREAD_STACK_SIZE];

s_start_init(); //初始化 start os

s_thread_init(&thread1,
          thread1entry,
          thread1stack,
          THREAD_STACK_SIZE,
          10,
          10);
s_thread_startup(&thread1);

s_thread_init(&thread2,
          thread2entry,
          thread2stack,
          THREAD_STACK_SIZE,
          12,
          10);
s_thread_startup(&thread2);

s_thread_init(&thread3,
          thread3entry,
          thread3stack,
          THREAD_STACK_SIZE,
          15,
          10);
s_thread_startup(&thread3);
s_sched_start(); //启动第一次调度

void thread1entry() //线程入口
{
    while(1)
    {
    s_printf("Thread 1\r\n");
        s_mdelay(1000);
    }
}
```

```
void thread2entry() //线程入口
{
    while(1)
    {
    s_printf("Thread 2\r\n");
        s_mdelay(1000);
    }
}

void thread3entry() //线程入口
{
    while(1)
    {
    s_printf("Thread 3\r\n");
        s_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
s_sched_init	初始化所有优先级队列与全局变量
s_sched_switch	若存在更高优先级 READY 线程则发起上下文切换
s_sched_remove_thread	从 READY 队列摘除，必要时清除位图
s_sched_insert_thread	插入 READY 队列并设置位图
s_thread_yield	同优先级轮转

---

4. 中断/CPU 相关

函数	说明
s_irq_disable	关中断返回原 PRIMASK
s_irq_enable(level)	恢复 PRIMASK
s_stack_init(entry, stack_top)	构建初始上下文（PSR/PC/LR/寄存器清零）
s_frist_switch_task(next)	首次上下文切换（历史拼写）
s_normal_switch_task(prev,next)	正常切换保存前线程栈并装载后线程栈
int __s_ffs(int v)	查找最低有效 1 位（1-based）； v=0 调用方需避免

---

5. 定时器与 Tick

函数	说明
s_timer_list_init	初始化定时器链表（level=1）
s_timer_init	初始化单个软件定时器

s_timer_start	计算 timeout_tick 并有序插入
s_timer_stop	从链表摘除
s_timer_ctrl	GET/SET 时间参数
s_tick_increase	SysTick ISR：全局 tick++ / 时间片处理 / 调用 s_timer_check
s_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
s_tick_get	获取全局 tick
s_mdelay / s_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：`ipc_parent`

```
struct ipc_parent
{
    s_uint8_t status;              /**< 1 = valid, 0 = deleted */
    s_uint8_t flag;                /**< Queueing policy / mode */
    s_list    suspend_thread;      /**< Waiting thread list head */
};
```

s_status s_ipc_suspend(s_list *list, s_pthread thread,

s_uint8_t flag)

线程挂起到 IPC 等待链表。
flag：

- FIFO：链表尾
- PRIO：按优先级插入（数值小 → 高）

s_status s_ipc_list_resume_all(s_list *list)

唤醒给定挂起链表全部线程（标 READY 并入就绪队列）。不立即切换；调用者可随后 `s_sched_switch()`。

---

7. 信号量 Semaphore

结构：`s_sem`

```
typedef struct semaphore
{
    struct ipc_parent parent; /**< Base IPC header */
    s_uint16_t count;         /**< Current resource count */
    s_uint16_t reserved;      /**< Reserved (alignment/extension) */
} s_sem, *s_psem;
```

函数	语义
s_sem_init	初始化 count、挂起队列、策略
s_sem_delete	唤醒全部等待者并失效对象

s_sem_take	获取资源或阻塞（支持无限/有限超时/非阻塞）
s_sem_release	释放资源并按策略唤醒一个等待者

当前超时返回 S_ERR（后续可区分 S_TIMEOUT）。

使用示例

```
s_sem sem1;

s_sem_init(&sem1, 0, START_IPC_FLAG_FIFO); // 初始化信号量，初始值为 0，FIFO 方式

void thread1entry()
{
  while(1)
  {
    if(S_OK !=s_sem_take(&sem1, START_WAITING_FOREVER)) // 等待信号量
    {
      s_printf("Thread 1: Waiting for semaphore...\r\n");
    }
    j++;
    if(j==255)
    j=0;
    s_printf("Thread 1: j = %d\r\n", j);
    s_mdelay(500);
  }
}

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

void thread3entry()
{
  while(1)
  {
    S_DEBUG_LOG(START_DEBUG_INFO, "Thread 3: k = %d\n", k);
    if(S_OK ==s_sem_release(&sem1)) // 释放信号量
```

```
    {
      s_printf("Thread 3: release semaphore...\r\n");
    }
    S_DEBUG_LOG(START_DEBUG_INFO, "sem1.count = %d\n", sem1.count);
    k++;
    if(k==255){
      k=0;
    }
    s_delay(600);
  }
}
```

---

8. 互斥量 Mutex（已具备递归与简单优先级继承雏形）

结构：`s_mutex`

```
typedef struct mutex
{
    struct ipc_parent parent;       /**< Base IPC header */
    s_pthread          owner;       /**< Owning thread */
    s_uint16_t         count;       /**< Availability (1 free, 0
taken) */
    s_uint8_t          original_priority; /**< Owner original
priority before inheritance */
    s_uint8_t          hold;        /**< Recursive acquisition depth
*/
} s_mutex, *s_pmutex;
```

函数	说明
s_mutex_init	初始化，可设排队策略
s_mutex_delete	唤醒等待者并恢复所有者原优先级
s_mutex_take	支持递归；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）

s_mutex_release	递归计数减，归零时转移或释放并恢复优先级

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

使用示例

```
s_mutex mutex1;

s_mutex_init(&mutex1,START_IPC_FLAG_FIFO);

/*优先级关系：thread1 < thread2 < thread3 */
void thread1entry() /* High priority (等待互斥量) */
{
    static int phase = 0;
    while (1)
    {
        if (phase == 0)
        {
            s_mdelay(100); /* 先让低优先级线程获取互斥量制造反转 */
            s_printf("HIGH : try take mutex\n");
            if (S_OK == s_mutex_take(&mutex1, START_WAITING_FOREVER))
            {
                s_printf("HIGH : got mutex (after inheritance)
j=%d\n", j);
                s_mutex_release(&mutex1);
                s_printf("HIGH : released mutex\n");
                phase = 1;
            }
        }
        else
        {
            /* 后续简单演示重复获取/释放 */
            if (S_OK == s_mutex_take(&mutex1, START_WAITING_FOREVER))
            {
                s_mutex_release(&mutex1);
            }
            s_mdelay(600);
        }
        j++;
        if (j == 255) j = 0;
        s_mdelay(50);
    }
}

void thread2entry() /* Medium priority (制造 CPU 干扰) */
{
    while (1)
    {
        i++;
        if (i % 50 == 0)
            s_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 */
        s_mdelay(40);
    }
}

void thread3entry() /* Low priority (先获取互斥量并长时间占用) */
{
    static int once = 0;
    s_uint8_t base_prio_saved = 0;
    while (1)
    {
        if (once == 0)
        {
            if (S_OK == s_mutex_take(&mutex1, START_WAITING_FOREVER))
            {
                base_prio_saved = s_thread_get()->current_priority;
                s_printf("LOW  : took mutex, do long work (base
prio=%d)\n",
                         base_prio_saved);

                /* 模拟长任务分多段，期间高优先级会等待，触发优先级继承 */
                for (int seg = 0; seg < 5; seg++)
                {
                    s_mdelay(120); /* 每段持有 */
                    s_thread *self = s_thread_get();
                    if (self)
                    {
                        if (self->current_priority != base_prio_saved)
                            s_printf("LOW  : inherited priority -> %d
(seg=%d)\n",
                                     self->current_priority, seg);
                    }
                }

                s_printf("LOW  : releasing mutex\n");
                s_mutex_release(&mutex1);
                s_printf("LOW  : released mutex (should drop back to
prio=%d)\n",
                         base_prio_saved);
                once = 1;
            }
        }
        else
        {
            /* 之后偶尔再占用一下，验证递归外正常路径 */
            if (S_OK == s_mutex_take(&mutex1, START_WAITING_FOREVER))
            {
                s_mdelay(30);
                s_mutex_release(&mutex1);
            }
            s_mdelay(200);
        }

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

---

9. 消息队列 Message Queue

结构：`s_msgqueue`

```
typedef struct
{
    struct ipc_parent parent;          /**< Base IPC header */
    void             *msg_pool;        /**< Raw pool base */
    s_uint16_t        msg_size;        /**< Aligned payload size */
    s_uint16_t        max_msgs;        /**< Maximum storable messages
*/
    s_uint16_t        index;           /**< Current queued count */
    void             *msg_queue_head;  /**< FIFO head (linked nodes)
*/
    void             *msg_queue_tail;  /**< FIFO tail */
    void             *msg_queue_free;  /**< Free node stack head */
    s_list            suspend_sender_thread; /**< Sender wait list */
} s_msgqueue, *s_pmsgqueue;
```

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 S_ERR（计划区分 S_TIMEOUT）。

函数	说明
s_msgqueue_init	初始化池 / 构建自由链表
s_msgqueue_delete	唤醒所有收发等待者并失效对象
s_msgqueue_send_wait	阻塞发送（池满时挂起）

s_msgqueue_send	非阻塞（池满返回 S_ERR）
s_msgqueue_urgent	头部插入（高优先级消费）
s_msgqueue_recv	阻塞 / 非阻塞接收

使用示例

```
typedef struct
{
    s_uint8_t data[4];
} msg_t;

#define MSG_QUEUE_SIZE 10
#define MSG_POOL_SIZE
START_MSGQ_POOL_SIZE(sizeof(msg_t),MSG_QUEUE_SIZE) // 10 条消息

s_msgqueue msgqueue1;

s_msgqueue_init(&msgqueue1,
                  msgpool,
                  sizeof(msg_t),
                  MSG_POOL_SIZE,
                  START_IPC_FLAG_FIFO);

void thread1entry()
{
  msg_t msg;
    while(1)
    {
    if(S_OK !=s_msgqueue_recv(&msgqueue1, &msg, sizeof(msg),
START_WAITING_FOREVER)) // 等待消息队列
    {
      s_printf("Thread 1: Waiting for message...\r\n");
    }

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

        s_mdelay(600);
    }
}

void thread2entry()
{
  msg_t msg;
    while(1)
    {
        i++;
    msg.data[0] = i;
    s_printf("Thread 2: urgent data[0] = %d\r\n", msg.data[0]);
```

```
    s_msgqueue_urgent(&msgqueue1, &msg, sizeof(msg));
        if(i==30){
      s_printf("Thread 2: delete message queue...\r\n");
      s_msgqueue_delete(&msgqueue1); // 删除消息队列
    }
        if(i==255){
            i=0;
        }
        s_mdelay(900);
    }
}

void thread3entry()
{
  msg_t msg;
  s_status err;
  while(1)
  {
    k++;
    if(k==255){
      k=0;
    }
    msg.data[0] = k;
    s_printf("Thread 3: send data[0] = %d\r\n", msg.data[0]);
    err = s_msgqueue_send(&msgqueue1, &msg, sizeof(msg));
    if( err == S_OK )
    {
      s_printf("Thread 3: err =%d\r\n", err);
    }
    else{
      s_printf("Thread 3: failed to send data[0] = %d, err = %d\r\n",
msg.data[0], err);
    }
    s_delay(300);
  }
}
```

---

10. 打印与调试

函数	说明
s_printf	轻量格式化输出（非线程安全）
s_vsnprintf	内部缓冲生成

s_putc (weak)	单字符发送（用户重写）
S_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	s_thread_init	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 / take 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
s_tick_increase	是	典型 SysTick
s_tick_get	是	只读
s_printf / s_putc	视实现	若使用阻塞 UART 需谨慎
s_sem_release	否(当前)	内部可能调度；若需支持需改为延迟调度
s_sem_take	否	可能阻塞
s_mutex_take/release	否	可能阻塞或调度
s_msgqueue_send/recv	否	可能阻塞
s_timer_start/stop	否(建议线程)	需短临界区；若需支持 ISR 可局部裁剪
s_thread_* (除查询)	否	涉及调度/阻塞
__s_ffs	是	纯计算
s_irq_disable/enable	是	底层操作

13. 典型使用流程（简要）

```
hw_init();
s_start_init();

s_thread_init(&t1, entry1, stack1, sizeof(stack1), 5, 10);
s_thread_startup(&t1);

s_thread_init(&t2, entry2, stack2, sizeof(stack2), 6, 10);
s_thread_startup(&t2);

s_sched_start(); /* 不返回 */
```

SysTick:

```
void SysTick_Handler(void) {
    s_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 init_tick	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 msgqueue（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 S_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）

- 2025-08-26 补充：互斥量/消息队列 API、错误码表、ISR 调用表、状态机统一、修正函数名。

（后续增量请追加）

如发现描述与实现不一致，请以源码为准并提交 Issue。

Happy hacking with StaRT!