StaRT RTOS 数据结构文档

本次补充：

- IPC/线程/消息队列结构字段完整描述
- 栈帧 / 上下文切换顺序示意
- 优先级位图与调度路径 ASCII 图
- 消息队列内存池布局与节点结构
- 互斥量优先级继承字段说明
- 线程状态转换更清晰的文字图
- 未来结构扩展占位

---

1. 基础类型与状态码

`sdef.h` 定义的整型别名保证与编译器一致。
`s_status` 枚举：详见 StaRT_API.md。
宏常量统一前缀：`START_`。

2. 双向循环链表：s_list

```
typedef struct list {
    struct list *next;
    struct list *prev;
} s_list, *s_plist;
```

性质：

- 所有内核队列（就绪队列、定时器、IPC 等待列表）使用该结构。
- 头节点也是一个 `s_list`，空时 next== prev == head。

相关操作：

- `s_list_init`
- `s_list_insert_after`
- `s_list_insert_before`
- `s_list_delete`
- `s_list_isempty`

使用技巧：通过 `S_LIST_ENTRY(node, type, member)` 获取父结构。

3. 线程控制块：s_thread

```
typedef struct thread {
    void      *psp;              // 线程进程栈指针 (PSP 上下文)
    void      *entry;            // 入口函数
    void      *stackaddr;        // 栈基地址（分配时的起始地址）
    s_uint32_t stacksize;        // 栈大小
```

```
    s_list     tlist;             // 就绪/挂起队列链接结点

    s_uint8_t  current_priority; // 当前优先级（可变）
    s_uint8_t  init_priority;    // 初始优先级（重启参考）
    s_uint32_t number_mask;      // 位图掩码 (1 << priority)

    s_uint32_t init_tick;        // 时间片初始值
    s_uint32_t remaining_tick;   // 剩余时间片

    s_int32_t  status;           // 线程状态（宏）
    s_timer    timer;            // 私有定时器（睡眠/超时）
} s_thread, *s_pthread;
```

状态字段

使用宏：

```
START_THREAD_INIT
START_THREAD_READY
START_THREAD_RUNNING
START_THREAD_SUSPEND
START_THREAD_TERMINATED
START_THREAD_DELETED
```

状态转换详见 API 文档表格。

位图调度结构

- `number_mask = 1U << current_priority`
- 全局 `s_thread_ready_priority_group` 组合所有 READY 线程优先级。

栈初始化

- `rtos_stack_init` 写入初始 PC=entry，LR 指向 `s_thread_exit`（避免直接 return 崩溃）。

---

4. 定时器：s_timer

```
typedef struct timer {
    s_list     row[START_TIMER_SKIP_LIST_LEVEL]; // 级别链表节点（当前
level=1）
    void     (*timeout_func)(void *p);           // 回调函数
    void      *p;                                // 回调参数
    s_uint32_t init_tick;                        // 周期或延时长度
    s_uint32_t timeout_tick;                     // 绝对到期时刻
(s_tick 基准)
} s_timer, *s_ptimer;
```

特点：

- 当前实现为单层按到期时间排序链表。
- `timeout_tick` = 安排时刻 + init_tick。
- 回调执行在 `s_tick_increase` -> `s_timer_check` 调用路径（中断上下文）。

---

5. IPC 父类：struct 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 */
};
```

为信号量/互斥量/消息队列等共享：

- `flag` == START_IPC_FLAG_FIFO / START_IPC_FLAG_PRIO
- `suspend_thread` 链表元素是 `thread.tlist`

6. 信号量：s_sem

```
typedef struct semaphore {
    struct ipc_parent parent;
    s_uint16_t count;
    s_uint16_t reserved;
} s_sem, *s_psem;
```

行为：

- `count>0` 直接获取
- `count==0` 按等待策略挂入 `parent.suspend_thread`
- 释放：有挂起线程→先自增 count 再唤醒一个；否则直接自增

限制：`count <= SEM_VALUE_MAX (0xFFFF)`。

7. 互斥量 (s_mutex)（结构定义，尚未实现逻辑）

```
typedef struct mutex {
    struct ipc_parent parent;
    s_pthread  owner;
    s_uint16_t count;              // 1=可获取 0=占用
    s_uint8_t  original_priority;  // owner 原优先级(优先级继承恢复)
    s_uint8_t  hold;               // 递归占用层次
} s_mutex, *s_pmutex;
```

补充：

- 递归上限 `MUTEX_HOLD_MAX`
- 简单优先级继承：高优先级等待时提升 owner->current_priority

- 释放最后一层时恢复 `original_priority`

---

8. 消息队列 (s_msgqueue)（尚未实现逻辑）

```
typedef struct {
    struct ipc_parent parent;
    void      *msg_pool;
    s_uint16_t msg_size;      // 对齐后单条有效载荷
    s_uint16_t max_msgs;
    s_uint16_t index;         // 当前队列消息数
    void      *msg_queue_head;
    void      *msg_queue_tail;
    void      *msg_queue_free;
    s_list     suspend_sender_thread; // 发送方等待
} s_msgqueue, *s_pmsgqueue;

struct s_mq_message {
    struct s_mq_message *next;
    // payload 紧随其后
};
```

内存布局（池）：

```
+-----------+---------+   (node0)
| next ptr  | payload |
+-----------+---------+   (node1)
| next ptr  | payload |
...
```

初始化构建自由链表：LIFO 形式 → 分配 O(1)。

---

9. 全局调度相关

```
s_pthread  s_current_thread;
s_uint8_t  s_current_priority;
s_list     s_thread_priority_table[START_THREAD_PRIORITY_MAX];
s_uint32_t s_thread_ready_priority_group;
s_list     s_thread_defunct_list;
```

```
volatile s_uint32_t s_tick;
```

位图：

```
s_thread_ready_priority_group
bit i = 1 ↔ priority i 有至少一个 READY 线程
```

查找：

```
highest = __s_ffs(bitmap) - 1;
```

---

10. 调度结构与位图优先级查找

READY 队列（示例 0...3）：

```
prio0: [HEAD] <-> T0a <-> T0b <-> [HEAD]
prio1: [HEAD] <-> T1a <-> [HEAD]
prio2: [HEAD] <-> (empty)
prio3: [HEAD] <-> T3a <-> T3b <-> T3c <-> [HEAD]
```

此时位图：

```
bitmap: b00011011 (LSB=prio0)
```

上下文切换触发来源：

- 新线程变 READY 且优先级高于当前
- 当前线程时间片耗尽（轮转）
- 阻塞 / 删除 / 退出当前线程
- 显式 yield

11. 线程状态转换文字图

```
INIT --startup--> READY --dispatch--> RUNNING
 ^                   ^  ^               |
 |                   |  |               |
 |               (timeout/IPC release)  |
 |                   |  |               v
 |<--restart-- DELETED  |          (sleep / IPC wait)
 |                      |               |
 |               READY <--- SUSPEND <----+
 |
 +-- idle cleanup -- TERMINATED <- (delete / exit from RUNNING)
          |
          +--> DELETED --(restart)--> INIT
```

12. 定时器链表示例

```
(now) tick=100
[HEAD] -> (timeout=120) -> (130) -> (210) -> ...
插入 timeout=125 node:
遍历到 130 停止，插入其前
```

比较采用：

```
(s_int32_t)(current_tick - node->timeout_tick) >= 0
```

以处理 tick 回绕。

13. 上下文切换关键变量

变量	说明
s_prev_thread_sp_p	上一线程 PSP 存放位置地址
s_next_thread_sp_p	下一线程 PSP 存放位置地址
s_interrupt_flag	触发 PendSV 标志（防止重复配置）

---

14. 一致性策略

场景	顺序
阻塞	关中断 → 从 READY 移除 → 状态=SUSPEND → 加入等待队列 → 开中断 → 调度
唤醒	关中断 → 从等待队列移除 → 状态=READY → 插入 READY → 开中断
删除线程	从 READY 移除 → 停止私有定时器 → 状态=TERMINATED → 入 defunct
Idle 清理	遍历 defunct → 状态=DELETED → 摘链

---

15. 关键宏与可配置项（摘要）

宏	作用
START_THREAD_PRIORITY_MAX	优先级数量
START_TICK	Tick 频率 Hz
START_TIMER_SKIP_LIST_LEVEL	定时器层级（当前=1）
START_IDLE_STACK_SIZE	Idle 栈大小
START_USING_SEMAPHORE / MUTEX / MESSAGEQUEUE / IPC	子系统开关
START_DEBUG	启用调试输出
S_PRINTF_BUF_SIZE	printf 临时缓冲

---

16. 变更记录（文档）

- 2025-08-26 增补：互斥量/消息队列字段说明、上下文切换流程、位图调度示意、未来扩展占位。
- 2025-08-24 初版结构描述。

如发现与源码不符，请以源码为准并提交 Issue。

Happy hacking with StaRT!