

1 系统配置模块

1.1 模块简介

系统配置模块定义 HRTOS 的所有系统配置参数、数据类型和核心数据结构。该模块是整个系统的基础，决定了系统的资源分配、任务数量、调度策略等关键参数。

1.1.1 设计目的

- 定义系统基本数据类型
- 配置系统资源参数
- 定义任务状态和等待类型
- 定义核心数据结构
- 提供系统配置接口

1.1.2 使用场景

- 系统定制和裁剪
- 资源分配配置
- 任务数量调整
- 调度策略配置
- 应用程序开发参考

1.2 数据类型

1.2.1 基本类型

类型定义	原类型	说明
---	---	---
u8	unsigned char	8 位无符号整数
u16	unsigned int	16 位无符号整数
u32	unsigned long	32 位无符号整数
f32	float	32 位浮点数
f64	double	64 位浮点数
NULL	0	空指针

1.2.2 OS_TCB

任务控制块结构：

```
typedef struct {
    u8 id; /* 任务 ID */
    u8 base_prio; /* 静态优先级 */
    u8 cur_prio; /* 动态优先级 */
    u8 state; /* 任务状态 */
    u8 wait_type; /* 等待类型 */
    u8 wait_flag; /* 等待结果 */
    u8 wait_obj; /* 等待对象 ID */
    u16 wait_tick; /* 超时等待 tick */
} OS_TCB;
```

| 成员 | 类型 | 说明 |

---|---|---

| id | u8 | 任务 ID |

| base_prio | u8 | 静态优先级 |

| cur_prio | u8 | 动态优先级 |

| state | u8 | 任务状态 |

| wait_type | u8 | 等待类型 |

| wait_flag | u8 | 等待结果 |

| wait_obj | u8 | 等待对象 ID |

| wait_tick | u16 | 超时等待 tick |

1.2.3 OS_RESOURCE

统一资源对象结构：

```
typedef struct {
    u8 value; /* 资源值 */
    u8 owner; /* 当前拥有者 */
    u8 wait_cnt; /* 当前等待任务数量 */
    u16 wait_mask; /* 位图等待队列 */
    u8 pending_signal; /* ISR 中断计数 */
} OS_RESOURCE;
```

| 成员 | 类型 | 说明 |

---|---|---

| value | u8 | 资源值 |

| owner | u8 | 当前拥有者（互斥锁专用） |

| wait_cnt | u8 | 当前等待任务数量 |

| wait_mask | u16 | 位图等待队列 |

| pending_signal | u8 | ISR 中断计数 |

1.2.4 os_msgq_t

消息队列控制块结构：

```
typedef struct {
```

```
    u8 *buf;          /* 环形消息缓冲区指针 */
    u8 _size;         /* 队列最大容量 */
    u8 head;          /* 写指针（生产者） */
    u8 tail;          /* 读指针（消费者） */
    u8 count;         /* 当前消息数量 */
} os_msgq_t;
```

成员	类型	说明
buf	u8*	环形消息缓冲区指针
_size	u8	队列最大容量
head	u8	写指针（生产者）
tail	u8	读指针（消费者）
count	u8	当前消息数量

1.3 宏定义

1.3.1 系统配置宏

宏定义	默认值	功能说明
OS_TIME_ONCE_DEFAULT	5	时间片数量
OS_TIME_T0	10000	单个时间片长度

1.3.2 任务配置宏

宏定义	默认值	功能说明
OS_PROCESS_MAX	16	标准任务数量最大值
OS_FAST_TASK_MAX	2	高速任务数量最大值
OS_SCHED_LEVEL_MAX	10	调度级别最大值
OS_MAX_TASK	16	最大任务数（等于 OS_PROCESS_MAX）

1.3.3 资源配置宏

宏定义	默认值	功能说明
OS_RESOURCE_MAX	8	统一资源对象最大数量
OS_MSGQ_MAX	4	消息队列对象最大数量

1.3.4 任务状态宏

宏定义	值	说明

READY	0	就绪态
WAIT	1	等待态
RUNNING	2	运行态
SUSPEND	3	挂起态
DEAD	4	删除态

1.3.5 等待类型宏

宏定义	值	说明
WAIT_NONE	0	无等待
WAIT_DELAY	1	延时等待
WAIT_SEM	2	信号量等待
WAIT_MUTEX	3	互斥锁等待
WAIT_MSG_SEND	4	消息队列发送等待
WAIT_MSG_RECV	5	消息队列接收等待
WAIT_EVENT	6	事件等待
WAIT_MAILBOX	7	邮箱等待

1.3.6 唤醒标志宏

宏定义	值	说明
WAIT_TIMEOUT	1	超时唤醒
WAIT_SIGNAL	2	信号唤醒

1.3.7 其他宏

宏定义	值	说明
OS_INVALID_ID	0xFF	无效 ID
OS_ENTER_CRITICAL()	do{ os_enter_critical(); }while(0)	进入临界区宏
OS_EXIT_CRITICAL()	do{ os_exit_critical(); }while(0)	退出临界区宏

1.4 全局变量

1.4.1 任务控制块数组

```
extern OS_TCB xdata OS_TASK[OS_PROCESS_MAX];
```

任务控制块数组，存储所有任务的状态信息。

1.4.2 统一资源对象数组

```
extern xdata OS_RESOURCE OS_RES[OS_RESOURCE_MAX];
```

统一资源对象数组，用于信号量、互斥锁、事件、邮箱等。

1.4.3 消息队列对象数组

```
extern xdata os_msgq_t OS_MSGQ[OS_MSGQ_MAX];
```

消息队列对象数组，存储所有消息队列的控制信息。

1.4.4 时间片备份

```
extern unsigned char xdata OS_TIME_ONCE_BACKUP;
```

配合 HRTOS 与 MYOS 调度策略切换的时间片备份。

1.5 调度级别定义

HRTOS 采用 10 级优先级设计：

| 优先级范围 | 类型 | 说明 |

|---|---|---|

| 0-7 | 普通任务 | 标准任务优先级，数值越大优先级越高 |

| 8 | 高优先级任务 | 高速任务，响应更快 |

| 9 | 中断级 | 中断处理函数优先级 |

| 10 | 中断嵌套级 | 高优先级中断嵌套处理 |

1.6 配置说明

1.6.1 任务数量配置

- **OS_PROCESS_MAX**：标准任务最大数量，默认为 16
- **OS_FAST_TASK_MAX**：高速任务最大数量，默认为 2
- 根据实际应用需求调整，影响内存占用

1.6.2 时间配置

- **OS_TIME_ONCE_DEFAULT**：时间片数量，默认为 5
- **OS_TIME_T0**：单个时间片长度，默认为 10000
- 影响调度精度和系统响应时间

1.6.3 资源配置

- **OS_RESOURCE_MAX**：统一资源对象最大数量，默认为 8
- **OS_MSGQ_MAX**：消息队列对象最大数量，默认为 4
- 根据应用需求调整，影响内存占用

1.6.4 内存布局

系统采用精心设计的内存布局：

- ****DATA 区****：内核临时寄存器
- ****IDATA 高区****：中断现场保护、调度核心变量
- ****XDATA 区****：任务控制块、资源对象、消息队列

1.7 配置建议

1.7.1 任务数量配置

- 根据实际任务需求设置 OS_PROCESS_MAX
- 高速任务数量通常较少（1-2 个）
- 过多任务会增加内存开销和调度开销

1.7.2 时间片配置

- 实时性要求高的应用：减少时间片数量
- 公平性要求高的应用：增加时间片数量
- 时间片长度影响系统响应精度

1.7.3 资源配置

- 根据同步和通信需求设置 OS_RESOURCE_MAX
- 根据消息传递需求设置 OS_MSGQ_MAX
- 合理配置可优化内存使用

1.8 注意事项

- 修改配置后需要重新编译系统
- 配置参数影响系统性能和内存占用
- 建议根据实际应用需求进行裁剪
- 修改任务数量需要相应调整内存布局

2 事件模块

2.1 模块简介

事件模块提供一对多任务同步机制，允许多个任务等待同一个事件。事件触发后会唤醒所有等待的任务，适用于需要广播通知的场景。

2.1.1 设计目的

- 提供一对多任务同步机制
- 支持事件广播通知
- 实现任务间条件同步
- 采用统一资源模型管理

2.1.2 使用场景

- 事件广播通知
- 多任务同步
- 条件等待
- 状态变化通知

2.2 数据类型

本模块使用 config.h 中定义的 OS_RESOURCE 结构。

2.3 宏定义

本模块无公开宏定义。

2.4 API 函数列表

函数名称	功能说明
---	---
os_event_init	事件初始化
os_event_write	设置事件

os_event_wait	等待事件
os_event_query	查询事件
os_event_delete	删除事件

2.5 API 详细说明

2.5.1 os_event_init

函数原型：

```
char os_event_init(u8 obj);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| obj | u8 | 事件 ID（资源对象编号） |

返回值：

- 1：初始化成功
- -1：参数错误

功能说明：

初始化指定事件。函数会：

1. 检查事件 ID 是否合法
2. 调用 os_res_init() 初始化资源对象
3. 清空事件状态
4. 清空等待队列

事件采用统一资源模型：

- value：事件状态（0=未触发，1=已触发）
- wait_mask：等待该事件的任务位图

调用示例：

```
char result = os_event_init(0);  
if (result == 1) {  
    // 初始化成功  
}
```

注意事项：

- 事件 ID 必须在 0 到 OS_RESOURCE_MAX-1 范围内
- 初始化应在系统启动或单任务环境调用
- 初始化后事件为未触发状态

2.5.2 os_event_write

函数原型：

```
char os_event_write(u8 obj);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

obj	u8	事件 ID
-----	----	-------

返回值：

- 1：设置成功

- -1：参数错误

功能说明：

设置指定事件，触发事件通知。该函数是 ISR-safe 和 task-safe 的（靠临界区保证）。

****事件触发：****

- 设置事件状态（OS_RES[obj].value = 1）

- 如果有等待任务，唤醒所有等待该事件的任务

- 遍历等待队列，对每个等待任务调用 wake_task()

- 事件成立全局唤醒

****并发安全：****

- 使用 EA = 0/EA = 1 保护临界区

- 防止中断与任务同时修改事件状态

调用示例：

```
char result = os_event_write(0);
if (result == 1) {
    // 事件设置成功
}
```

注意事项：

- 会唤醒所有等待该事件的任务

- 事件触发后保持触发状态

- 支持中断安全版本 os_event_set_from_isr

- 可能触发任务调度

- 多次调用不会重复唤醒

2.5.3 os_event_wait

函数原型：

```
char os_event_wait(u8 obj, u16 tick);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

obj	u8	事件 ID
-----	----	-------

tick	u16	超时时间（tick 数）
------	-----	--------------

返回值：

- 1：事件已触发或被唤醒
- -1：参数错误

功能说明：

等待指定事件触发，支持超时机制。

****事件已触发时：****

- 直接返回 1（不阻塞）

****事件未触发时：****

- 调用 `os_wait(WAIT_EVENT, obj, tick)` 进入等待
- 支持超时等待
- 被唤醒后返回等待结果

调用示例：

```
char result = os_event_wait(0, 100);  
if (result == 1) {  
    // 事件触发或被唤醒  
}
```

注意事项：

- 支持超时机制
- 可能阻塞当前任务
- 事件触发后会唤醒所有等待任务
- `tick = 0` 表示无限等待

2.5.4 os_event_query

函数原型：

```
char os_event_query(u8 obj);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

obj	u8	事件 ID
-----	----	-------

返回值：

- 查询结果

功能说明：

查询指定事件的状态，非阻塞操作。

调用示例：

```
char state = os_event_query(0);
```

注意事项：

- 非阻塞操作
- 不会改变事件状态

- 不会触发任务调度

2.5.5 os_event_delete

函数原型：

```
char os_event_delete(u8 obj);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

obj	u8	事件 ID
-----	----	-------

返回值：

- 删除结果

功能说明：

删除指定事件，释放相关资源。

调用示例：

```
char result = os_event_delete(0);
```

注意事项：

- 会清空事件状态
- 会清空等待队列
- 谨慎使用，可能影响等待任务

Interrupt

3 中断管理模块

3.1 模块简介

中断管理模块提供中断相关的控制功能，包括临界区保护、中断环境判断、中断安全 API 等。该模块确保中断与任务间的安全通信和操作。

3.1.1 设计目的

- 提供临界区保护机制
- 支持中断环境判断
- 提供中断安全 API
- 实现中断与任务间安全通信
- 支持中断嵌套

3.1.2 使用场景

- 临界区保护
- 中断服务程序
- 中断与任务通信
- 中断安全资源操作
- 中断环境判断

3.2 数据类型

本模块无特殊数据类型定义。

3.3 宏定义

本模块使用 config.h 中定义的临界区宏：

| 宏定义 | 功能说明 |

---|---

| OS_ENTER_CRITICAL() | 进入临界区（关中断） |

| OS_EXIT_CRITICAL() | 退出临界区（恢复中断） |

3.4 API 函数列表

函数名称	功能说明
---	---
os_interrupt_id	获取中断编号
os_enter_critical	进入临界区（关中断）
os_exit_critical	退出临界区（恢复中断）
os_in_isr	判断当前运行环境
os_queue_send_from_isr	中断安全邮箱发送
os_schedule_request	请求系统调度
os_event_set_from_isr	中断安全事件设置
os_semaphore_post_from_isr	中断安全信号量释放
os_msgq_send_from_isr	中断安全消息队列发送

3.5 API 详细说明

3.5.1 os_interrupt_id

函数原型：

```
char os_interrupt_id();
```

参数说明：

无

返回值：

- 中断编号（0, 2-31 根据自然入口顺序排列）

功能说明：

获取当前中断编号。工作在寄存器 3，返回中断编号。中断编号根据自然入口顺序排列。

调用示例：

```
char irq_id = os_interrupt_id();
```

注意事项：

- 必须在中断上下文中调用
- 返回值对应 51 单片机中断向量
- 用于中断服务程序识别

3.5.2 os_enter_critical

函数原型：

```
void os_enter_critical();
```

参数说明：

无

返回值：

无

功能说明：

关中断，保护运行不被打断。支持中断嵌套，使用嵌套计数器管理：

1. 检查嵌套计数 (os_critical_nesting)
2. 如果是第一层 (os_critical_nesting == 0) :
 - 保存当前中断状态 (os_int_status = EA)
 - 关中断 (EA = 0)
3. 嵌套计数加 1 (os_critical_nesting++)

调用示例：

```
os_enter_critical();  
// 临界区代码  
os_exit_critical();
```

注意事项：

- 必须与 os_exit_critical 配对使用
- 支持嵌套调用
- 嵌套层数有限制
- 临界区代码应尽可能简短
- 可以使用宏 OS_ENTER_CRITICAL()

3.5.3 os_exit_critical

函数原型：

```
void os_exit_critical();
```

参数说明：

无

返回值：

无

功能说明：

恢复中断，退出临界区。支持中断嵌套，使用嵌套计数器管理：

1. 嵌套计数减 1 (os_critical_nesting--)
2. 如果是最后一层 (os_critical_nesting == 0) :
 - 恢复中断状态 (EA = os_int_status)

调用示例：

```
os_enter_critical();  
// 临界区代码  
os_exit_critical();
```

注意事项：

- 必须与 os_enter_critical 配对使用

- 支持嵌套调用
- 不能单独调用（未配对）
- 可以使用宏 OS_EXIT_CRITICAL()

3.5.4 os_in_isr

函数原型：

```
char os_in_isr(void);
```

参数说明：

无

返回值：

- 1：当前在中断中运行
- 0：当前在任务中运行

功能说明：

判断当前代码是在任务里运行，还是在中断里运行。通过检查 OS_ISR_FLAG 标志判断。

调用示例：

```
if (os_in_isr()) {
    // 当前在中断中
} else {
    // 当前在任务中
}
```

注意事项：

- 用于区分运行环境
- 中断安全 API 内部使用
- 返回值基于系统标志

3.5.5 os_queue_send_from_isr

函数原型：

```
char os_queue_send_from_isr(u8 id, u8 _data);
```

参数说明：

参数	类型	说明
id	u8	邮箱编号
_data	u8	写入数据

id | u8 | 邮箱编号 |

_data | u8 | 写入数据 |

返回值：

- 发送结果

功能说明：

中断安全邮箱发送。在中断中向邮箱发送数据，确保中断环境下的安全操作。

调用示例：

```
char result = os_queue_send_from_isr(0, 0x55);
```

注意事项:

- 必须在中断上下文中调用
- 采用延迟唤醒机制
- 触发调度请求
- 与 os_mailbox_send 对应

3.5.6 os_schedule_request

函数原型:

```
char os_schedule_request(char id);
```

参数说明:

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

id	char	调度参数
----	------	------

返回值:

- 调度请求结果

功能说明:

通知系统重新调度。在中断中请求系统进行任务调度。

调用示例:

```
os_schedule_request(1);
```

注意事项:

- 通常在中断服务程序结束时调用
- 触发调度器检查
- 可能导致任务切换

3.5.7 os_event_set_from_isr

函数原型:

```
char os_event_set_from_isr(u8 id);
```

参数说明:

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

id	u8	事件 ID
----	----	-------

返回值:

- 设置结果

功能说明:

中断里设置事件标志, 通知任务某个条件成立了。中断安全的事件触发操作。

调用示例:

```
char result = os_event_set_from_isr(0);
```

注意事项：

- 必须在中断上下文中调用
- 采用延迟唤醒机制
- 与 `os_event_write` 对应
- 会触发调度请求

3.5.8 `os_semaphore_post_from_isr`

函数原型：

```
char os_semaphore_post_from_isr(u8 obj);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

obj	u8	资源对象 ID
-----	----	---------

返回值：

- 1：成功
- -1：参数错误

功能说明：

中断里释放/发送一个信号量，通知某个等待信号量的任务继续运行。采用延迟唤醒机制：

****有等待任务时：****

- 仅记录事件 (`pending_signal++`)
- 触发调度 (`OS_SCHED_REASON = 1`)
- 实际唤醒由内核统一处理

****无等待任务时：****

- 计数+1 (`value++`)

调用示例：

```
char result = os_semaphore_post_from_isr(0);
```

注意事项：

- 必须在中断上下文中调用
- 采用延迟唤醒机制，避免中断中复杂操作
- 与 `os_sem_post` 对应
- 会触发调度请求

3.5.9 `os_msgq_send_from_isr`

函数原型：

```
char os_msgq_send_from_isr(os_msgq_t *q, u8 obj, u8 _data);
```

参数说明：

| 参数 | 类型 | 说明 |

|---|---|---|

| q | os_msgq_t* | 消息队列控制块 |

| obj | u8 | 资源对象 ID |

| _data | u8 | 待发送数据 |

返回值：

- 发送结果

功能说明：

中断安全消息队列发送。在中断中向消息队列发送数据，确保中断环境下的安全操作。

调用示例：

```
char result = os_msgq_send_from_isr(&my_queue, 0, 0x55);
```

注意事项：

- 必须在中断上下文中调用
- 采用延迟唤醒机制
- 与 os_msgq_send 对应
- 会触发调度请求

4 内核模块

4.1 模块简介

内核模块是 HRTOS 的核心组件，提供系统初始化、任务创建、调度控制、内存管理等基础功能。该模块负责系统的启动、中断处理、任务调度等关键操作。

4.1.1 设计目的

- 提供系统初始化和启动功能
- 实现任务注册和创建机制
- 控制调度模式切换
- 提供系统级钩子函数
- 管理内存操作

4.1.2 使用场景

- 系统启动时初始化
- 创建用户任务
- 切换调度模式
- 注册系统钩子函数
- 系统级异常处理

4.2 数据类型

本模块无特殊数据类型定义。

4.3 宏定义

本模块无公开宏定义。

4.4 API 函数列表

函数名称	功能说明
---	---

os_dispatch_select	调度器选择中断号
os_task_create	任务创建
os_interrupt_init	中断初始化
os_interrupt_exit	中断退出
os_memset	内存填充
os_data_to_xdata	数据区到扩展数据区拷贝
os_xdata_to_data	扩展数据区到数据区拷贝
os_scheduler_mode_switch	调度模式切换
os_set_scheduler	调度器选择
hrtos_main	HRTOS 默认用户主函数
os_slab_overflow	系统溢出处理
os_slab_overflow_register	注册溢出钩子函数
os_task_cleanup	系统任务清理
os_idle_task	系统 idle 任务
os_idle_hook_register	注册 idle 钩子函数

4.5 API 详细说明

4.5.1 os_dispatch_select

函数原型：

```
char os_dispatch_select(unsigned char interrupt_id);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

interrupt_id	unsigned char	中断编号
--------------	---------------	------

返回值：

- 返回调度器选择的中断号

功能说明：

调度器选择中断号，用于系统中断分发机制。该函数根据传入的中断 ID 选择相应的处理路径。

调用示例：

```
char id = os_dispatch_select(2);
```

注意事项：

- 需要在中断上下文中调用
- 中断编号必须在合法范围内

4.5.2 os_task_create

函数原型:

```
char os_task_create(unsigned int addr, unsigned char id, unsigned char pr, unsigned char sd);
```

参数说明:

| 参数 | 类型 | 说明 |

---|---|---

| addr | unsigned int | 任务函数地址 |

| id | unsigned char | 任务编号 |

| pr | unsigned char | 任务优先级 |

| sd | unsigned char | 任务堆栈层数 |

返回值:

- 1: 创建成功

- -1: 创建失败 (参数错误、资源不足、重复注册)

功能说明:

创建并注册一个新任务。函数会:

1. 参数合法性检查 (优先级、任务编号、地址)
2. 堆栈空间分配 (使用位图管理)
3. 任务控制块初始化
4. 根据任务类型 (标准任务/高速任务/中断处理) 进行不同处理
5. 设置任务初始状态为就绪态

标准任务优先级范围: 0-7

高速任务优先级: 8

中断处理优先级: 9-10

调用示例:

```
// 创建标准任务
char result = os_task_create((unsigned int)task_func, 0, 3, 5);
if (result == 1) {
    // 任务创建成功
}
```

注意事项:

- 任务编号不能超过 OS_PROCESS_MAX+1
- 优先级不能超过 OS_SCHED_LEVEL_MAX
- 堆栈层数有限制 (标准任务≤6, 高速任务≤25)
- 不能重复注册同一任务编号
- 需要在系统初始化阶段调用

4.5.3 os_interrupt_init

函数原型:

```
void os_interrupt_init();
```

参数说明：

无

返回值：

无

功能说明：

MyOS 的初始化及中断支持。该函数负责：

1. 保存当前堆栈指针
2. 初始化系统堆栈
3. 保护现场寄存器
4. 跳转到中断服务程序

调用示例：

```
os_interrupt_init();
```

注意事项：

- 仅在 MyOS 模式下使用
- 需要在系统启动时调用

4.5.4 os_interrupt_exit

函数原型：

```
void os_interrupt_exit();
```

参数说明：

无

返回值：

无

功能说明：

中断退出处理。根据当前中断状态选择正确的返回路径：

- 普通中断返回
- 中断嵌套返回
- MyOS 模式返回

调用示例：

```
os_interrupt_exit();
```

注意事项：

- 必须在中断服务程序结束时调用
- 会根据 OS_ISR_FLAG 判断当前是否在中断中

4.5.5 os_memset

函数原型：

```
void os_memset(void *buffer, unsigned char c, unsigned int count);
```

参数说明：

参数	类型	说明
buffer	void*	目标缓冲区地址
c	unsigned char	填充值
count	unsigned int	填充长度

返回值：

无

功能说明：

将指定缓冲区填充为指定值。类似标准库的 memset 函数。

调用示例：

```
u8 buffer[32];  
os_memset(buffer, 0, 32);
```

注意事项：

- 需要确保缓冲区有足够空间
- count 参数以字节为单位

4.5.6 os_data_to_xdata

函数原型：

```
void os_data_to_xdata(u8 addr, u16 addr2);
```

参数说明：

参数	类型	说明
addr	u8	源地址（DATA 区）
addr2	u16	目标地址（XDATA 区）

返回值：

无

功能说明：

将数据从 DATA 区拷贝到 XDATA 区。

调用示例：

```
os_data_to_xdata(src_addr, dst_addr);
```

注意事项：

- 地址必须在合法范围内
- 用于 51 单片机特殊存储区操作

4.5.7 os_xdata_to_data

函数原型：

```
void os_xdata_to_data(u16 addr2, u8 addr);
```

参数说明:

参数	类型	说明
addr2	u16	源地址 (XDATA 区)
addr	u8	目标地址 (DATA 区)

返回值:

无

功能说明:

将数据从 XDATA 区拷贝到 DATA 区。

调用示例:

```
os_xdata_to_data(src_addr, dst_addr);
```

注意事项:

- 地址必须在合法范围内
- 用于 51 单片机特殊存储区操作

4.5.8 os_scheduler_mode_switch

函数原型:

```
char os_scheduler_mode_switch(unsigned char id);
```

参数说明:

参数	类型	说明
id	unsigned char	调度模式标识

返回值:

- 调度模式切换结果

功能说明:

切换调度器模式:

- id=1: 实时模式 (基于优先级)
- id=0: 通用模式 (基于时间片)

调用示例:

```
os_scheduler_mode_switch(1); // 切换到实时模式
```

注意事项:

- 模式切换会影响调度行为
- 建议在系统初始化时确定调度模式

4.5.9 os_set_scheduler

函数原型:

```
char os_set_scheduler(unsigned char id);
```

参数说明:

| 参数 | 类型 | 说明 |

|---|---|---|

| id | unsigned char | 调度器标识 |

返回值:

- 调度器设置结果

功能说明:

选择调度器类型:

- id=1: HRTOS 调度器

- id=0: MYOS 调度器

调用示例:

```
os_set_scheduler(1); // 使用 HRTOS 调度器
```

注意事项:

- 需要在系统启动时调用

- 一旦设置不建议频繁切换

4.5.10 hrtos_main

函数原型:

```
void hrtos_main();
```

参数说明:

无

返回值:

无

功能说明:

HRTOS 默认的用户主函数。系统启动后会跳转到该函数执行用户代码。

调用示例:

```
// 用户实现该函数
void hrtos_main() {
    // 用户初始化代码
    // 创建任务
    // 启动调度
}
```

注意事项:

- 用户必须实现该函数

- 该函数是系统启动后的入口点

4.5.11 os_slab_overflow

函数原型:

```
void os_slab_overflow();
```

参数说明:

无

返回值：

无

功能说明：

系统发生堆栈溢出时调用的处理函数。默认实现可能导致系统复位。

调用示例：

```
// 通常由系统自动调用
```

注意事项：

- 可以通过 `os_slab_overflow_register` 注册自定义处理函数
- 发生溢出表示严重错误

4.5.12 `os_slab_overflow_register`

函数原型：

```
void os_slab_overflow_register(void (*hook)(void));
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

hook	void (*)(void)	溢出处理钩子函数指针
------	----------------	------------

返回值：

无

功能说明：

注册堆栈溢出时的用户处理函数。当系统检测到堆栈溢出时，会调用注册的钩子函数。

调用示例：

```
void my_overflow_handler() {  
    // 用户自定义溢出处理  
}  
  
os_slab_overflow_register(my_overflow_handler);
```

注意事项：

- 钩子函数应尽可能简单
- 避免在钩子函数中调用复杂操作

4.5.13 `os_task_cleanup`

函数原型：

```
char os_task_cleanup(unsigned char id);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

| id | unsigned char | 任务编号 |

返回值：

- 清理结果

功能说明：

系统任务清理，释放任务占用的资源并清理相关数据结构。

调用示例：

```
os_task_cleanup(0);
```

注意事项：

- 通常由系统内部调用

- 用户一般不需要直接调用

4.5.14 os_idle_task

函数原型：

```
void os_idle_task();
```

参数说明：

无

返回值：

无

功能说明：

系统 idle 任务，当没有其他就绪任务时运行。可以执行低优先级后台任务或进入低功耗模式。

调用示例：

```
// 系统自动调用
```

注意事项：

- 可以通过 os_idle_hook_register 注册自定义 idle 钩子

- idle 任务优先级最低

4.5.15 os_idle_hook_register

函数原型：

```
void os_idle_hook_register(void (*hook)(void));
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| hook | void (*)(void) | idle 钩子函数指针 |

返回值：

无

功能说明：

注册 idle 任务钩子函数。当系统进入 idle 任务时，会调用注册的钩子函数。

调用示例：

```
void my_idle_hook() {  
    // 自定义 idle 处理  
    // 可以进入低功耗模式  
}  
  
os_idle_hook_register(my_idle_hook);
```

注意事项：

- 钩子函数会被频繁调用
- 避免在钩子函数中执行耗时操作
- 钩子函数中不能调用阻塞 API

5 邮箱模块

5.1 模块简介

邮箱模块提供单字节消息传递机制，适合简单的信号通知和数据传递。HRTOS 邮箱采用统一资源模型实现，支持任务间快速通信。

5.1.1 设计目的

- 提供简单的单字节消息传递
- 支持任务间快速通信
- 实现事件通知机制
- 采用统一资源模型管理

5.1.2 使用场景

- 简单信号通知
- 单字节数据传递
- 事件标志传递
- 任务同步

5.2 数据类型

本模块使用 config.h 中定义的 OS_RESOURCE 结构。

5.3 宏定义

本模块无公开宏定义。

5.4 API 函数列表

函数名称	功能说明
---	---
os_mailbox_init	邮箱初始化
os_mailbox_send	邮箱发送

os_mailbox_receive	邮箱接收
os_mailbox_query	邮箱查询
os_mailbox_clear	邮箱清除

5.5 API 详细说明

5.5.1 os_mailbox_init

函数原型：

```
void os_mailbox_init(u8 mid);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| mid | u8 | 邮箱 ID（资源对象编号） |

返回值：

无

功能说明：

初始化指定邮箱。函数会：

1. 检查邮箱 ID 是否合法
2. 清空邮箱内容（value = 0）
3. 清空资源拥有者（owner = OS_INVALID_ID）
4. 清空等待队列（wait_mask）
5. 清空等待计数（wait_cnt）

邮箱采用统一资源模型：

- value：邮箱数据（0=空，非 0=有数据）
- wait_mask：等待该邮箱的任务位图

调用示例：

```
os_mailbox_init(0); // 初始化邮箱 0
```

注意事项：

- 邮箱 ID 必须在 0 到 OS_RESOURCE_MAX-1 范围内
- 初始化应在系统启动或单任务环境调用
- 初始化后邮箱为空

5.5.2 os_mailbox_send

函数原型：

```
char os_mailbox_send(u8 mid, u8 content);
```

参数说明：

| 参数 | 类型 | 说明 |

|---|---|---|

| mid | u8 | 邮箱 ID |

| content | u8 | 待发送数据 |

返回值：

- 1：发送成功

- -1：参数错误

功能说明：

向指定邮箱发送一个字节数据。

****有等待接收任务时：****

- 写入数据 (m->value = content)

- 查找等待队列中的任务

- 调用 wake_task() 唤醒任务

- 返回 1

****无等待任务时：****

- 写入数据 (m->value = content)

- 数据保存在邮箱中

- 返回 1

邮箱语义：

- 写入数据

- 如果有等待接收任务，直接唤醒

- 否则保存数据供后续接收

调用示例：

```
char result = os_mailbox_send(0, 0x55);
if (result == 1) {
    // 发送成功
}
```

注意事项：

- 邮箱只能保存一个字节数据

- 新数据会覆盖旧数据

- 支持中断安全版本 os_queue_send_from_isr

- 可能触发任务调度

5.5.3 os_mailbox_receive

函数原型：

```
int os_mailbox_receive(u8 mid);
```

参数说明：

| 参数 | 类型 | 说明 |

|---|---|---|

| mid | u8 | 邮箱 ID |

返回值：

- 接收到的数据（非负数）
- -1：参数错误

功能说明：

从指定邮箱接收一个字节数据。

****邮箱有数据时（快路径）：****

- 读取数据（_data = m->value）
- 清空邮箱（m->value = 0）
- 返回数据

****邮箱无数据时（慢路径）：****

- 调用 os_wait(WAIT_MAILBOX, mid, 0) 进入等待
- 被唤醒后重新读取数据
- 返回数据

邮箱语义：

- 如果有数据，直接取走
- 若无数据，进入等待状态

调用示例：

```
int data = os_mailbox_receive(0);
if (data >= 0) {
    // 接收成功，data 包含接收到的数据
}
```

注意事项：

- 接收后邮箱变为空
- 可能阻塞当前任务
- 多次接收会等待新数据
- 返回值为 int 类型，需检查是否为 -1

5.5.4 os_mailbox_query

函数原型：

```
int os_mailbox_query(u8 mid);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| mid | u8 | 邮箱 ID |

返回值：

- 邮箱中的数据（非阻塞查询）
- -1：参数错误

功能说明：

非阻塞查询指定邮箱中的数据。不会改变邮箱状态，也不会阻塞当前任务。

调用示例：

```
int data = os_mailbox_query(0);  
if (data >= 0) {  
    // 邮箱中有数据  
}
```

注意事项：

- 非阻塞操作
- 不会清空邮箱
- 不会触发任务调度
- 适合轮询场景

5.5.5 os_mailbox_clear

函数原型：

```
char os_mailbox_clear(u8 mid);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

mid	u8	邮箱 ID
-----	----	-------

返回值：

- 清除结果

功能说明：

清除指定邮箱中的数据，重置邮箱为空状态。

调用示例：

```
char result = os_mailbox_clear(0);
```

注意事项：

- 会丢弃邮箱中的数据
- 不会唤醒等待任务
- 谨慎使用，可能导致数据丢失

Msgq

6 消息队列模块

6.1 模块简介

消息队列模块提供基于环形缓冲区的消息传递机制，支持生产者-消费者模型。消息队列允许任务间传递单字节数据，支持阻塞和非阻塞两种模式。

6.1.1 设计目的

- 提供任务间数据传输通道
- 支持生产者-消费者模型
- 实现缓冲机制，平滑生产消费速度差异
- 支持阻塞和非阻塞操作

6.1.2 使用场景

- 任务间数据传输
- 事件通知
- 数据缓冲
- 生产者-消费者同步

6.2 数据类型

6.2.1 os_msgq_t

消息队列控制块结构：

```
typedef struct {  
    u8 *buf;           // 环形消息缓冲区指针  
    u8 _size;          // 队列最大容量  
    u8 head;           // 写指针（生产者位置）  
    u8 tail;           // 读指针（消费者位置）  
    u8 count;          // 当前队列中的消息数量  
} os_msgq_t;
```

成员	类型	说明
--- --- ---		
buf	u8*	环形消息缓冲区指针
_size	u8	队列最大容量

head u8 写指针（生产者位置）
tail u8 读指针（消费者位置）
count u8 当前队列中的消息数量

6.3 宏定义

本模块无公开宏定义。

6.4 API 函数列表

函数名称 功能说明
--- ---
os_msgq_init 消息队列初始化
os_msgq_send 发送消息
os_msgq_recv 接收消息
os_msgq_clear 清空消息队列

6.5 API 详细说明

6.5.1 os_msgq_init

函数原型：

```
void os_msgq_init(os_msgq_t *q, u8 *buf, u8 _size);
```

参数说明：

参数 类型 说明
--- --- ---
q os_msgq_t* 消息队列控制块指针
buf u8* 环形消息缓冲区
_size u8 队列最大容量

返回值：

无

功能说明：

初始化消息队列，将消息队列对象与缓冲区绑定，并设置初始状态：

1. 关联消息缓冲区（q->buf = buf）
2. 设置队列容量（q->_size = _size）
3. 初始化队头索引（q->head = 0）
4. 初始化队尾索引（q->tail = 0）
5. 清空消息数量（q->count = 0）

调用示例：

```
u8 msg_buffer[16];
os_msgq_t my_queue;

os_msgq_init(&my_queue, msg_buffer, 16);
```

注意事项:

- 缓冲区必须足够大
- _size 不能超过缓冲区实际大小
- 初始化后队列为空
- 必须在发送/接收前初始化

6.5.2 os_msgq_send

函数原型:

```
char os_msgq_send(os_msgq_t *q, u8 obj, u8 _data, char nonblock);
```

参数说明:

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID (用于等待队列管理)
_data	u8	待发送数据
nonblock	char	非阻塞标志 (0=阻塞, 非 0=非阻塞)

返回值:

- 1: 发送成功
- 0: 非阻塞模式下队列满
- -1: 等待失败

功能说明:

向消息队列发送一个字节消息。

****队列未满时: ****

- 将数据写入队尾 (q->buf[q->head] = _data)
- 更新写指针 (q->head = (q->head + 1) % q->_size)
- 消息数量加 1 (q->count++)
- 如果有等待接收的任务, 唤醒一个
- 返回 1

****队列已满时: ****

- 非阻塞模式: 直接返回 0
- 阻塞模式: 调用 os_wait(WAIT_MSG_SEND, obj, 0) 进入等待
- 返回等待结果

****唤醒接收任务: ****

- 遍历等待队列, 查找等待接收的任务

- 调用 wake_task() 唤醒任务

调用示例：

```
// 阻塞发送
char result = os_msgq_send(&my_queue, 0, 0x55, 0);

// 非阻塞发送
result = os_msgq_send(&my_queue, 0, 0x55, 1);
if (result == 1) {
    // 发送成功
}
```

注意事项：

- 队列满时会阻塞（除非非阻塞模式）
- 使用环形缓冲区，自动循环
- obj 参数用于统一资源模型管理
- 可能阻塞当前任务
- 支持中断安全版本 os_msgq_send_from_isr

6.5.3 os_msgq_recv

函数原型：

```
char os_msgq_recv(os_msgq_t *q, u8 obj, u8 *_data, char nonblock);
```

参数说明：

参数	类型	说明
q	os_msgq_t*	消息队列控制块
obj	u8	资源对象 ID（用于等待队列管理）
_data	u8*	接收数据存储地址
nonblock	char	非阻塞标志（0=阻塞，非 0=非阻塞）

返回值：

- 1：接收成功
- 0：非阻塞模式下队列空
- -1：等待失败

功能说明：

从消息队列接收一个字节消息。

****队列非空时：****

- 从队头读取消息（*_data = q->buf[q->tail]）
- 更新读指针（q->tail = (q->tail + 1) % q->_size）
- 消息数量减 1（q->count--）
- 如果有等待发送的任务，唤醒一个
- 返回 1

****队列为空时：****

- 非阻塞模式：直接返回 0
- 阻塞模式：调用 `os_wait(WAIT_MSG_RECV, obj, 0)` 进入等待
- 返回等待结果

****唤醒发送任务：****

- 遍历等待队列，查找等待发送的任务
- 调用 `wake_task()` 唤醒任务

调用示例：

```
u8 data;
// 阻塞接收
char result = os_msgq_recv(&my_queue, 0, &data, 0);

// 非阻塞接收
result = os_msgq_recv(&my_queue, 0, &data, 1);
if (result == 1) {
    // 接收成功，data 包含接收到的数据
}
```

注意事项：

- 队列空时会阻塞（除非非阻塞模式）
- `_data` 指针必须有效
- `obj` 参数用于统一资源模型管理
- 可能阻塞当前任务
- 支持中断安全版本 `os_msgq_recv_from_isr`

6.5.4 `os_msgq_clear`

函数原型：

```
void os_msgq_clear(os_msgq_t *q);
```

参数说明：

| 参数 | 类型 | 说明 |

| --- | --- | --- |

| `q` | `os_msgq_t*` | 消息队列控制块 |

返回值：

无

功能说明：

清空消息队列中的所有消息，重置队列状态。

调用示例：

```
os_msgq_clear(&my_queue);
```

注意事项：

- 会丢弃队列中所有未处理消息
- 重置读写指针

- 不会唤醒等待任务
- 谨慎使用，可能导致数据丢失

7 互斥锁模块

7.1 模块简介

互斥锁模块提供互斥访问共享资源的机制，支持优先级继承以防止优先级反转问题。HRTOS 互斥锁采用统一资源模型实现，确保同一时间只有一个任务能访问受保护资源。

7.1.1 设计目的

- 提供互斥访问共享资源的机制
- 支持优先级继承，防止优先级反转
- 确保资源访问的排他性
- 提供死锁预防（不支持递归锁）

7.1.2 使用场景

- 保护共享数据结构
- 保护硬件资源访问
- 保护临界区代码
- 任务间互斥控制

7.2 数据类型

本模块使用 config.h 中定义的 OS_RESOURCE 结构。

7.3 宏定义

本模块无公开宏定义。

7.4 API 函数列表

函数名称	功能说明
---	---
os_mutex_init	互斥锁初始化
os_mutex_lock	获取互斥锁

| os_mutex_unlock | 释放互斥锁 |

7.5 API 详细说明

7.5.1 os_mutex_init

函数原型：

```
char os_mutex_init(u8 mid);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| mid | u8 | 互斥锁 ID（资源对象编号） |

返回值：

- 1：初始化成功
- -1：参数错误（ID 超出范围）

功能说明：

初始化指定互斥锁。函数会：

1. 检查互斥锁 ID 是否合法
2. 设置互斥锁为可用状态（value = 1）
3. 清空资源拥有者（owner = OS_INVALID_ID）
4. 清空等待队列（wait_mask）
5. 清空等待计数（wait_cnt）

互斥锁采用统一资源模型：

- value：互斥锁状态（1=空闲，0=占用）
- owner：当前拥有者任务 ID
- wait_mask：等待该互斥锁的任务位图

调用示例：

```
char result = os_mutex_init(0); // 初始化互斥锁 0
if (result == 1) {
    // 初始化成功
}
```

注意事项：

- 互斥锁 ID 必须在 0 到 OS_RESOURCE_MAX-1 范围内
- 初始化应在系统启动或单任务环境调用
- 互斥锁被使用后严禁重新初始化
- 不支持递归锁

7.5.2 os_mutex_lock

函数原型：

```
char os_mutex_lock(u8 mid);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

mid	u8	互斥锁 ID
-----	----	--------

返回值：

- 1：成功获取锁或进入等待队列
- -1：参数错误或递归锁错误

功能说明：

尝试获取指定互斥锁。支持优先级继承机制：

****互斥锁空闲时：****

- 直接获取锁（owner = 当前任务 ID）
- 更新任务优先级为基础优先级
- 返回 1

****互斥锁被占用时：****

- 检查是否为递归锁（当前任务已持有）
- 若非递归，执行优先级继承：
 - 如果等待任务优先级 > 拥有者优先级
 - 提升拥有者优先级到等待任务优先级
 - 同步到 PROCESS_OK 寄存器
- 进入互斥锁等待状态（os_wait(WAIT_MUTEX, mid, 0)）
- 返回 1

****递归锁检查：****

- 如果当前任务已持有该锁
- 直接返回 -1（不支持递归锁）

优先级继承机制：

- 防止高优先级任务被低优先级任务间接阻塞
- 提升拥有者优先级到等待任务中的最高优先级
- 释放锁时恢复原始优先级

调用示例：

```
char result = os_mutex_lock(0);
if (result == 1) {
    // 成功获取互斥锁
    // 访问共享资源
    os_mutex_unlock(0); // 释放互斥锁
}
```

注意事项：

- 不支持递归锁

- 必须与 os_mutex_unlock 配对使用
- 可能阻塞当前任务
- 优先级继承仅在等待时触发
- 获取锁后必须及时释放

7.5.3 os_mutex_unlock

函数原型：

```
char os_mutex_unlock(u8 mid);
```

参数说明：

参数	类型	说明
mid	u8	互斥锁 ID

---	---	---
-----	-----	-----

mid	u8	互斥锁 ID
-----	----	--------

返回值：

- 1：释放成功且有等待任务被唤醒
- 0：释放成功且无等待任务
- -1：释放失败（非法 ID 或非拥有者）

功能说明：

释放指定互斥锁。函数会：

1. 检查互斥锁 ID 是否合法
2. 检查当前任务是否为互斥锁拥有者
3. 恢复任务原始优先级（取消优先级继承）
4. 同步优先级到 PROCESS_OK 寄存器

****有等待任务时：****

- 在等待队列中寻找最高优先级任务
- 转移互斥锁所有权给该任务
- 调用 wake_task() 唤醒任务
- 返回 1

****无等待任务时：****

- 互斥锁恢复为空闲状态（owner = OS_INVALID_ID）
- 返回 0

****非拥有者释放：****

- 返回 -1（拒绝释放）

优先级恢复：

- 释放互斥锁后，取消优先级继承状态
- 恢复任务到 base_prio
- 同步到调度器

调用示例：

```
char result = os_mutex_unlock(0);  
if (result == 1) {  
    // 有任务被唤醒  
}
```

注意事项：

- 只有互斥锁拥有者才能释放
- 必须与 os_mutex_lock 配对使用
- 会恢复任务原始优先级
- 可能触发任务调度
- 释放后互斥锁变为空闲状态

Overview

8 HRTOS 概述

8.1 系统简介

HRTOS (Hard Real-Time Operating System) 是一款专为 51 单片机设计的硬实时操作系统，版本 V3.30。系统采用优先级抢占式调度策略，支持多任务并发执行，提供完整的内核同步与通信机制。

8.1.1 设计目的

- **硬实时性保证**：通过优先级抢占和时间片轮转双重调度机制，确保关键任务及时响应
- **资源高效利用**：针对 51 单片机有限的内存资源，采用紧凑的内存布局和高效的数据结构
- **统一资源模型**：信号量、互斥锁、事件、邮箱、消息队列采用统一的等待队列管理
- **中断安全设计**：提供完整的中断安全 API，支持中断与任务间的安全通信

8.1.2 使用场景

- 工业控制系统
- 嵌入式实时数据采集
- 通信协议栈
- 电机控制
- 传感器网络节点

8.2 系统架构

8.2.1 内核组成

HRTOS 内核由以下核心模块组成：

- **任务管理** (Task)：任务创建、删除、挂起、恢复、优先级管理
- **调度器** (Scheduler)：优先级抢占和时间片轮转调度
- **同步机制**：信号量 (Semaphore)、互斥锁 (Mutex)
- **通信机制**：消息队列 (Message Queue)、邮箱 (Mailbox)、事件 (Event)

- ****时间管理**** (Time)：系统 tick、延时、时间转换
- ****中断管理**** (Interrupt)：临界区保护、中断安全 API
- ****统一等待**** (Wait)：统一的任务阻塞与唤醒机制

8.2.2 内存布局

系统采用精心设计的内存布局，充分利用 51 单片机的寻址空间：

- ****DATA 区 (0x20-0x34) ****：内核临时寄存器
- ****IDATA 高区 (0xDB-0xE7) ****：中断现场保护、调度核心变量
- ****XDATA 区 (0x0200 起始) ****：任务控制块、资源对象、消息队列

8.3 任务调度机制

8.3.1 调度模式

HRTOS 支持两种调度模式：

1. ****优先级抢占模式**** (实时模式)
 - 高优先级任务优先执行
 - 同优先级任务时间片轮转
 - 适合硬实时应用
2. ****时间片轮转模式**** (通用模式)
 - 所有任务按时间片轮流执行
 - 公平性好，实时性略低
 - 适合通用多任务应用

8.3.2 优先级设计

系统采用 10 级优先级设计：

优先级范围	类型	说明
--- --- ---		
0-7	普通任务	标准任务优先级，数值越大优先级越高
8	高优先级任务	高速任务，响应更快
9	中断级	中断处理函数优先级
10	中断嵌套级	高优先级中断嵌套处理

8.3.3 任务生命周期

任务状态转换：

创建 → 就绪 → 运行 → 挂起/等待 → 就绪

↓
删除

- ****就绪态**** (READY)：任务已准备好，等待调度

- ****运行态**** (RUNNING)：任务正在执行
- ****等待态**** (WAIT)：任务等待资源或超时
- ****挂起态**** (SUSPEND)：任务被暂停，不参与调度
- ****删除态**** (DEAD)：任务已删除

8.4 统一等待机制

HRTOS 采用统一的等待机制管理所有阻塞操作：

8.4.1 等待类型

等待类型	宏定义	说明
无等待	WAIT_NONE	任务不处于等待状态
延时等待	WAIT_DELAY	任务延时指定 tick
信号量等待	WAIT_SEM	等待信号量资源
互斥锁等待	WAIT_MUTEX	等待互斥锁
消息发送等待	WAIT_MSG_SEND	消息队列满时等待
消息接收等待	WAIT_MSG_RECV	消息队列空时等待
事件等待	WAIT_EVENT	等待事件触发
邮箱等待	WAIT_MAILBOX	等待邮箱数据

8.4.2 唤醒标志

唤醒标志	宏定义	说明
超时唤醒	WAIT_TIMEOUT	等待超时后被唤醒
信号唤醒	WAIT_SIGNAL	资源可用或事件触发被唤醒

8.4.3 统一资源模型

所有内核对象（信号量、互斥锁、事件、邮箱、消息队列）使用统一的资源结构：

```
typedef struct {
    u8 value;           /* 资源值 */
    u8 owner;           /* 当前拥有者（互斥锁专用） */
    u8 wait_cnt;        /* 当前等待任务数量 */
    u16 wait_mask;      /* 位图等待队列 */
    u8 pending_signal;  /* ISR 中断计数 */
} OS_RESOURCE;
```

等待队列使用位图管理：

- bit0 对应 task0
- bit1 对应 task1
- ...

- bit15 对应 task15

8.5 同步机制

8.5.1 信号量 (Semaphore)

- 用于资源计数和任务同步
- 支持 P 操作 (wait) 和 V 操作 (post)
- 采用双通道模型: value 计数 + wait_mask 队列

8.5.2 互斥锁 (Mutex)

- 用于互斥访问共享资源
- 支持优先级继承, 防止优先级反转
- 不支持递归锁

8.6 通信机制

8.6.1 消息队列 (Message Queue)

- 环形缓冲区实现
- 支持阻塞/非阻塞模式
- 生产者-消费者模型

8.6.2 邮箱 (Mailbox)

- 单字节消息传递
- 适合简单信号通知
- 统一资源模型实现

8.6.3 事件 (Event)

- 一对多任务同步
- 事件触发后唤醒所有等待任务
- 支持超时等待

8.7 中断安全 API

HRTOS 提供完整的中断安全 API, 确保中断与任务间的安全通信:

- ``os_semaphore_post_from_isr()``: 中断中释放信号量
- ``os_event_set_from_isr()``: 中断中设置事件
- ``os_queue_send_from_isr()``: 中断中发送邮箱消息
- ``os_msgq_send_from_isr()``: 中断中发送消息队列

中断安全 API 采用延迟唤醒机制：

- 中断中仅记录 pending_signal
- 实际唤醒由内核调度器统一处理
- 避免中断中复杂操作

8.8 时间管理

8.8.1 系统 Tick

- 基于 Timer0 实现
- 32 位计数器（高 16 位 + 低 16 位）
- 可配置时间片数量和单个时间片长度

8.8.2 时间转换

- ``os_ms_to_tick()``：毫秒转 tick
- ``os_sec_to_tick()``：秒转 tick
- ``os_uptime_ms()``：获取系统运行时间（毫秒）

8.8.3 延时函数

- ``os_delay()``：任务延时（基于 tick，释放 CPU）
- ``os_delay_ms()``：忙等待延时（占用 CPU，用于硬件时序）

8.9 系统配置

系统通过 ``config.h`` 进行配置：

8.9.1 任务配置

- ``OS_PROCESS_MAX``：标准任务最大数量（默认 16）
- ``OS_FAST_TASK_MAX``：高速任务最大数量（默认 2）

8.9.2 时间配置

- ``OS_TIME_ONCE_DEFAULT``：时间片数量（默认 5）
- ``OS_TIME_T0``：单个时间片长度（默认 10000）

8.9.3 资源配置

- ``OS_RESOURCE_MAX``：统一资源对象最大数量（默认 8）
- ``OS_MSGQ_MAX``：消息队列最大数量（默认 4）

8.10 API 模块列表

HRTOS API 按功能模块组织：

- **kernel.md**：内核核心功能
- **task.md**：任务管理
- **semaphore.md**：信号量
- **mutex.md**：互斥锁
- **msgq.md**：消息队列
- **mailbox.md**：邮箱
- **event.md**：事件
- **wait.md**：统一等待
- **time.md**：时间管理
- **interrupt.md**：中断管理
- **config.md**：系统配置

8.11 版本信息

- **系统名称**：HRTOS
- **版本号**：V3.30
- **更新日期**：2026-6-15
- **联系邮箱**：admin@hrtos.com
- **官网**：hrtos.com
- **QQ 群**：523549975
- **版权所有**：谭舒

9 信号量模块

9.1 模块简介

信号量模块提供经典的 P/V 操作同步机制，用于任务间同步和资源计数。HRTOS 采用统一资源模型实现信号量，支持资源计数和等待队列管理。

9.1.1 设计目的

- 提供任务间同步机制
- 实现资源计数管理
- 支持多任务等待同一资源
- 采用双通道模型优化性能

9.1.2 使用场景

- 资源访问控制
- 任务同步
- 生产者-消费者模型
- 事件通知

9.2 数据类型

本模块使用 config.h 中定义的 OS_RESOURCE 结构。

9.3 宏定义

本模块无公开宏定义。

9.4 API 函数列表

函数名称	功能说明
---	---
os_sem_init	信号量初始化
os_sem_wait	信号量 P 操作（等待/获取）

| os_sem_post | 信号量 V 操作（释放/发送） |

9.5 API 详细说明

9.5.1 os_sem_init

函数原型：

```
char os_sem_init(u8 sid, u8 init_val);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| sid | u8 | 信号量 ID（资源对象编号） |

| init_val | u8 | 信号量初始计数值 |

返回值：

- 1：初始化成功
- -1：参数错误（ID 超出范围）

功能说明：

初始化指定信号量。函数会：

1. 检查信号量 ID 是否合法
2. 设置信号量初始计数值
3. 清空资源拥有者（信号量不使用 owner 字段）
4. 清空等待队列（wait_mask）
5. 清空等待计数（wait_cnt）

信号量采用统一资源模型：

- value：可用资源计数
- wait_mask：等待该资源的任务位图

调用示例：

```
char result = os_sem_init(0, 2); // 初始化信号量 0，初始计数为 2
if (result == 1) {
    // 初始化成功
}
```

注意事项：

- 信号量 ID 必须在 0 到 OS_RESOURCE_MAX-1 范围内
- 初始化应在系统启动或单任务环境调用
- 不涉及临界区保护（假定未运行调度器）
- 初始计数值表示可用资源数量

9.5.2 os_sem_wait

函数原型：

```
u8 os_sem_wait(u8 sid);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| sid | u8 | 信号量 ID |

返回值：

- WAIT_SIGNAL：成功获取信号量
- WAIT_TIMEOUT：等待超时（如果设置了超时）
- 0：参数错误或等待失败

功能说明：

信号量 P 操作（等待/获取）。采用双通道同步模型：

****快路径（无阻塞）：****

- value > 0 时直接消费资源（value--）
- 返回 WAIT_SIGNAL
- 避免进入等待队列，减少调度开销

****慢路径（阻塞）：****

- value == 0 时进入统一等待机制
- 调用 os_wait(WAIT_SEM, sid, 0)
- 任务进入 WAIT 状态
- 挂入 wait_mask 等待队列
- 等待 os_sem_post 唤醒

并发安全设计：

- 使用 EA = 0/EA = 1 保护临界区
- 防止 tick 中断修改 value
- 防止 sem_post 与 sem_wait 同时竞争资源
- 保证 value-- 与状态判断原子性

调用示例：

```
u8 result = os_sem_wait(0);
if (result == WAIT_SIGNAL) {
    // 成功获取信号量
    // 执行临界区代码
    os_sem_post(0);    // 释放信号量
}
```

注意事项：

- 信号量 ID 必须合法
- value 不可为负（只减不加负逻辑）
- WAIT_SEM 是统一调度入口类型
- 可能阻塞当前任务

- 获取信号量后必须对应释放

9.5.3 os_sem_post

函数原型：

```
char os_sem_post(u8 sid);
```

参数说明：

参数	类型	说明
sid	u8	信号量 ID

返回值：

- 1：有等待任务被唤醒
- 0：无等待任务，信号量计数增加
- -1：参数错误

功能说明：

信号量 V 操作（释放/发送）。核心规则：

****有等待任务时：****

- 不增加 value
- 直接唤醒最高优先级等待任务
- 等价于"资源直接交付给等待任务"

****无等待任务时：****

- value++（资源累积）
- 表示信号量可用

调度策略：

- wait_mask 使用 bitmap 管理等待任务
- 查找等待队列中 cur_prio 最高的任务
- 调用 wake_task() 唤醒任务

并发安全：

- 使用 EA = 0/EA = 1 保护临界区
- 防止 tick 中断同时修改 wait_mask
- 防止调度器与唤醒逻辑竞争资源状态

调用示例：

```
char result = os_sem_post(0);  
if (result == 1) {  
    // 有任务被唤醒  
}
```

注意事项：

- value 只在"无等待任务"时累加
- wait_mask 为唯一等待状态来源

- 必须与 `os_sem_wait` 配对使用
- 唤醒任务会设置 `wait_flag = WAIT_SIGNAL`
- 支持中断安全版本 `os_semaphore_post_from_isr`

10 任务管理模块

10.1 模块简介

任务管理模块提供完整的任务生命周期管理功能，包括任务创建、删除、挂起、恢复、优先级调整等操作。该模块是 HRTOS 的核心组件之一，负责管理多任务并发执行。

10.1.1 设计目的

- 提供任务状态管理功能
- 支持任务优先级动态调整
- 实现任务挂起和恢复机制
- 提供任务查询和控制接口
- 支持任务锁机制

10.1.2 使用场景

- 动态创建和删除任务
- 任务优先级调整
- 任务暂停和恢复
- 任务状态查询
- 任务间同步控制

10.2 数据类型

本模块无特殊数据类型定义，使用 config.h 中定义的 OS_TCB 结构。

10.3 宏定义

本模块无公开宏定义。

10.4 API 函数列表

函数名称	功能说明
---	---

	os_task_suspend		挂起任务	
	os_task_get_state		获取任务状态	
	os_task_set_next		指定下一个运行任务	
	os_task_resume		恢复挂起任务	
	os_task_exit		退出当前任务	
	os_task_delete		删除任务	
	os_task_set_priority		设置任务优先级	
	os_task_get_priority		获取任务优先级	
	os_task_is_valid		检查任务是否有效	
	os_task_self		获取当前任务 ID	
	os_task_yield		让出 CPU	
	os_task_suspend_self		主动挂起当前任务	
	os_lock		任务加锁	
	os_unlock		任务解锁	
	os_lock_query		查询锁状态	

10.5 API 详细说明

10.5.1 os_task_suspend

函数原型：

```
char os_task_suspend(unsigned char id);
```

参数说明：

	参数		类型		说明	
--	----	--	----	--	----	--

	---		---		---	
--	-----	--	-----	--	-----	--

	id		unsigned char		任务编号	
--	----	--	---------------	--	------	--

返回值：

- 1：挂起成功

- -1：参数错误

功能说明：

将指定任务的状态变为挂起状态。挂起的任务暂停运行，不参与调度，直到被恢复。

调用示例：

```
char result = os_task_suspend(0);
if (result == 1) {
    // 任务 0 已挂起
}
```

注意事项：

- 任务编号必须在合法范围内

- 挂起当前任务需要使用 `os_task_suspend_self`
- 挂起的任务不会占用 CPU 资源

10.5.2 `os_task_get_state`

函数原型：

```
char os_task_get_state(unsigned char id);
```

参数说明：

参数	类型	说明
---	---	---
id	unsigned char	任务编号

---	---	---
-----	-----	-----

id	unsigned char	任务编号
----	---------------	------

返回值：

- 0：删除态 (DEAD)
- 1：运行态 (RUNNING)
- 2：就绪态 (READY)
- 3：挂起态 (SUSPEND)
- 4：超时态 (上一个结束的任务)
- -1：参数错误

功能说明：

查询指定任务的当前状态。根据任务控制块和系统状态寄存器判断任务状态。

调用示例：

```
char state = os_task_get_state(0);
if (state == 2) {
    // 任务处于就绪态
}
```

注意事项：

- 任务编号必须在合法范围内
- 返回值与 `config.h` 中定义的状态宏对应

10.5.3 `os_task_set_next`

函数原型：

```
char os_task_set_next(unsigned char id);
```

参数说明：

参数	类型	说明
---	---	---
id	unsigned char	任务编号

---	---	---
-----	-----	-----

id	unsigned char	任务编号
----	---------------	------

返回值：

- 1：设置成功
- -1：参数错误

功能说明：

指定下一个运行的任务，终止当前调度并强行切换到指定任务。该函数会强制触发任务切换。

调用示例：

```
os_task_set_next(1); // 强制切换到任务 1
```

注意事项：

- 会立即触发任务切换
- 指定的任务必须处于就绪态
- 谨慎使用，可能影响实时性

10.5.4 os_task_resume

函数原型：

```
char os_task_resume(unsigned char id);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

id	unsigned char	任务编号
----	---------------	------

返回值：

- 1：恢复成功
- -1：参数错误

功能说明：

将指定任务从挂起状态恢复为就绪状态，使其重新参与调度。

调用示例：

```
char result = os_task_resume(0);  
if (result == 1) {  
    // 任务 0 已恢复  
}
```

注意事项：

- 任务必须处于挂起态才能恢复
- 恢复后的任务会进入就绪队列
- 可能立即触发调度

10.5.5 os_task_exit

函数原型：

```
void os_task_exit();
```

参数说明：

无

返回值：

无

功能说明：

立即退出当前任务。任务会从系统中删除，释放相关资源。

调用示例：

```
void task_func() {  
    // 任务逻辑  
    if (error_condition) {  
        os_task_exit(); // 退出任务  
    }  
}
```

注意事项：

- 调用后任务立即终止
- 会释放任务占用的堆栈资源
- 不能在任务外部调用

10.5.6 os_task_delete

函数原型：

```
char os_task_delete(unsigned char id);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

id	unsigned char	任务编号
----	---------------	------

返回值：

- 1：删除成功
- -1：参数错误

功能说明：

彻底删除指定任务，从任务队列中移除并释放内存资源。

调用示例：

```
char result = os_task_delete(0);  
if (result == 1) {  
    // 任务 0 已删除  
}
```

注意事项：

- 不能删除当前运行的任务
- 删除后任务编号可以重新使用
- 会释放任务堆栈资源

10.5.7 os_task_set_priority

函数原型：

```
char os_task_set_priority(unsigned char id, unsigned char pr);
```

参数说明：

参数	类型	说明
----	----	----

---|---|---

| id | unsigned char | 任务编号 |

| pr | unsigned char | 新优先级（0-7 自然优先级，8 高优先级） |

返回值：

- 1：设置成功

- -1：参数错误

功能说明：

修改指定任务的优先级。优先级范围：

- 0-7：普通任务优先级（数值越大优先级越高）

- 8：高优先级任务

调用示例：

```
char result = os_task_set_priority(0, 5); // 将任务 0 优先级设为 5
```

注意事项：

- 优先级必须在合法范围内

- 修改优先级可能立即触发调度

- 高优先级任务会抢占低优先级任务

10.5.8 os_task_get_priority

函数原型：

```
char os_task_get_priority(unsigned char id);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| id | unsigned char | 任务编号 |

返回值：

- 任务优先级（0-7 自然优先级，8 高优先级）

- -1：参数错误

功能说明：

查询指定任务的当前优先级。

调用示例：

```
char prio = os_task_get_priority(0);
```

注意事项：

- 任务编号必须在合法范围内

- 返回值反映任务的静态优先级

10.5.9 os_task_is_valid

函数原型：

```
signed char os_task_is_valid(u8 id);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

id	u8	任务编号
----	----	------

返回值：

- -1：参数异常
- 0：任务未注册
- 1：任务已注册

功能说明：

检查指定任务是否已注册/存在。用于验证任务编号的有效性。

调用示例：

```
signed char valid = os_task_is_valid(0);
if (valid == 1) {
    // 任务 0 已注册
}
```

注意事项：

- 常用于参数检查
- 可以避免对无效任务进行操作

10.5.10 os_task_self

函数原型：

```
u8 os_task_self(void);
```

参数说明：

无

返回值：

- 当前任务的 ID

功能说明：

获取当前运行任务的编号。

调用示例：

```
u8 my_id = os_task_self();
```

注意事项：

- 只能在任务上下文中调用
- 返回值可用于任务识别

10.5.11 os_task_yield

函数原型：

```
void os_task_yield();
```

参数说明：

无

返回值：

无

功能说明：

当前任务从运行态转为就绪态，主动让出 CPU 并触发调度。适用于需要公平共享 CPU 的场景。

调用示例：

```
void task_func() {
    while (1) {
        // 任务逻辑
        os_task_yield(); // 让出 CPU
    }
}
```

注意事项：

- 任务会重新进入就绪队列
- 可能立即被重新调度
- 不会改变任务优先级

10.5.12 os_task_suspend_self

函数原型：

```
void os_task_suspend_self();
```

参数说明：

无

返回值：

无

功能说明：

当前任务主动挂起自己，让出 CPU 并触发调度。挂起后需要其他任务调用 `os_task_resume` 才能恢复。

调用示例：

```
void task_func() {
    // 任务逻辑
    os_task_suspend_self(); // 主动挂起
}
```

注意事项：

- 只能在任务上下文中调用
- 挂起后需要其他任务恢复
- 谨慎使用，可能导致任务无法恢复

10.5.13 os_lock

函数原型：

```
void os_lock(void);
```

参数说明：

无

返回值：

无

功能说明：

对注册任务进行加锁，防止任务被修改或删除。

调用示例：

```
os_lock();  
// 临界区代码  
os_unlock();
```

注意事项：

- 必须与 os_unlock 配对使用
- 避免长时间持有锁
- 可能影响系统调度

10.5.14 os_unlock

函数原型：

```
void os_unlock(void);
```

参数说明：

无

返回值：

无

功能说明：

解除注册任务的锁状态，允许任务被修改或删除。

调用示例：

```
os_lock();  
// 临界区代码  
os_unlock();
```

注意事项：

- 必须与 os_lock 配对使用
- 解锁未加锁的任务可能导致错误

10.5.15 os_lock_query

函数原型：

```
char os_lock_query();
```

参数说明：

无

返回值：

- 1：锁定状态

- 0: 未锁定

功能说明:

查询当前任务的锁状态。

调用示例:

```
char locked = os_lock_query();  
if (locked == 1){  
    // 任务处于锁定状态  
}
```

注意事项:

- 用于检查锁状态
- 可以避免重复加锁

11 时间管理模块

11.1 模块简介

时间管理模块提供系统时间管理功能，包括系统 tick 计数、任务延时、时间转换等。该模块基于 Timer0 实现，提供精确的时间管理能力。

11.1.1 设计目的

- 提供系统 tick 计数
- 实现任务延时功能
- 支持时间单位转换
- 提供系统运行时间查询
- 配置系统定时器参数

11.1.2 使用场景

- 任务延时
- 定时操作
- 时间测量
- 系统运行时间统计
- 硬件时序等待

11.2 数据类型

本模块无特殊数据类型定义。

11.3 宏定义

本模块使用 config.h 中定义的时间配置宏：

| 宏定义 | 默认值 | 功能说明 |

---|---|---

| OS_TIME_ONCE_DEFAULT | 5 | 时间片数量 |

| OS_TIME_T0 | 10000 | 单个时间片长度 |

11.4 API 函数列表

函数名称	功能说明
---	---
os_delay_ms	忙等待延时（毫秒）
os_ms_to_tick	毫秒转 tick
os_sec_to_tick	秒转 tick
os_tick_config	系统定时器配置
os_tick_get	获取系统 tick 计数
os_uptime_ms	获取系统运行时间（毫秒）
os_delay	任务延时（tick）

11.5 API 详细说明

11.5.1 os_delay_ms

函数原型：

```
void os_delay_ms(unsigned char ms);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

ms	unsigned char	延时毫秒数
----	---------------	-------

返回值：

无

功能说明：

忙等待延时函数，占用 CPU，不释放处理器。仅用于短时间硬件时序等待，不适合任务延时。

调用示例：

```
os_delay_ms(10); // 忙等待 10 毫秒
```

注意事项：

- 占用 CPU，不释放处理器
- 仅用于短时间硬件时序等待
- 不适合任务延时
- 任务延时应使用 os_delay()

11.5.2 os_ms_to_tick

函数原型：

```
ul6 os_ms_to_tick(ul6 ms);
```

参数说明：

| 参数 | 类型 | 说明 |

---|---|---

| ms | u16 | 毫秒数 |

返回值:

- 对应的 tick 数

功能说明:

将毫秒数转换为系统 tick 数。根据当前定时器配置进行换算。

调用示例:

```
u16 ticks = os_ms_to_tick(100); // 将 100 毫秒转换为 tick
```

注意事项:

- 转换结果依赖于当前定时器配置

- 用于时间参数换算

11.5.3 os_sec_to_tick

函数原型:

```
u16 os_sec_to_tick(u8 sec);
```

参数说明:

| 参数 | 类型 | 说明 |

---|---|---

| sec | u8 | 秒数 |

返回值:

- 对应的 tick 数

功能说明:

将秒数转换为系统 tick 数。根据当前定时器配置进行换算。

调用示例:

```
u16 ticks = os_sec_to_tick(5); // 将 5 秒转换为 tick
```

注意事项:

- 转换结果依赖于当前定时器配置

- 用于时间参数换算

11.5.4 os_tick_config

函数原型:

```
char os_tick_config(u8 slice_count, u16 tick_value);
```

参数说明:

| 参数 | 类型 | 说明 |

---|---|---

| slice_count | u8 | 时间片数量 |

| tick_value | u16 | 单个时间片的大小 |

返回值：

- 0：配置成功

- -1：参数错误

功能说明：

系统定时器时间设置。配置 Timer0 的重装值和时间片数量：

1. 清零系统时钟计数（高 16 位和低 16 位）
2. 设置时间片数量（OS_TIME_ONCE = slice_count）
3. 计算 Timer0 重装值（65535 - tick_value + 1）
4. 检查重装值是否合法（必须 > 1000）
5. 设置 Timer0 高字节和低字节（OS_T0_TH0、OS_T0_TL0）

调用示例：

```
char result = os_tick_config(5, 10000);
if (result == 0) {
    // 配置成功
}
```

注意事项：

- tick_value 不能为 0
- 计算后的重装值必须 > 1000
- 应在系统启动时调用
- 影响所有时间相关函数

11.5.5 os_tick_get

函数原型：

```
void os_tick_get(u16 *high, u16 *low);
```

参数说明：

| 参数 | 类型 | 说明 |

| --- | --- | --- |

| high | u16* | 返回 tick 高 16 位 |

| low | u16* | 返回 tick 低 16 位 |

返回值：

无

功能说明：

获取系统 tick 计数。系统 tick 由高 16 位和低 16 位组成 32 位计数器。

调用示例：

```
u16 high, low;
os_tick_get(&high, &low);
u32 total_ticks = ((u32)high << 16) | low;
```

注意事项：

- 32 位计数器，低 16 位溢出时高 16 位加 1
- 用于精确时间测量
- 读取时建议关中断保证原子性

11.5.6 os_uptime_ms

函数原型：

```
u32 os_uptime_ms();
```

参数说明：

无

返回值：

- 系统累计运行时间（毫秒）

功能说明：

获取系统累计运行时间（毫秒）。函数会：

1. 读取 32 位 tick 计数（高 16 位 + 低 16 位）
2. 获取单 tick 对应的时间（微秒）
3. 转换为毫秒
4. 返回总运行时间

计算方法：

- $\text{tick_us} = 65536 - \text{tick_value}$ （Timer0 重装值）
- $\text{tick_ms} = \text{tick_us} / 1000$
- $\text{total_ms} = \text{total_tick} * \text{tick_ms}$

调用示例：

```
u32 uptime = os_uptime_ms();
```

注意事项：

- 基于 Timer0 配置计算
- 使用浮点运算减少误差
- 返回值为 32 位毫秒数
- 读取时关中断保证原子性

11.5.7 os_delay

函数原型：

```
void os_delay(u16 tick);
```

参数说明：

参数	类型	说明
tick	u16	延时的 tick 数

---	---	---
-----	-----	-----

tick	u16	延时的 tick 数
------	-----	------------

返回值：

无

功能说明：

任务延时函数，基于系统 tick 计时。当前任务进入 WAIT_DELAY 状态，到期后自动唤醒。

1. tick = 0 时直接返回
2. 检查当前任务合法性
3. 调用 os_wait(WAIT_DELAY, OS_INVALID_ID, tick) 进入等待
4. 到期后被内核唤醒

调用示例：

```
os_delay(100); // 延时 100 个 tick
```

注意事项：

- 会阻塞当前任务
- 释放 CPU，其他任务可运行
- 基于系统 tick 计时
- tick = 0 立即返回
- 高速任务和标准任务处理方式不同

Wait

12 统一等待模块

12.1 模块简介

统一等待模块是 HRTOS 的核心组件，提供统一的任务阻塞与唤醒机制。所有阻塞操作（延时、信号量、互斥锁、消息队列、事件、邮箱）都通过该模块实现。

12.1.1 设计目的

- 提供统一的任务阻塞接口
- 简化同步机制实现
- 统一管理等待队列
- 支持超时机制
- 实现任务唤醒逻辑

12.1.2 使用场景

- 任务延时等待
- 资源等待（信号量、互斥锁）
- 消息等待（消息队列、邮箱）
- 事件等待
- 所有需要任务阻塞的场景

12.2 数据类型

12.2.1 OS_RESOURCE

统一资源对象结构（在 config.h 中定义）：

```
typedef struct {
    u8 value;           /* 资源值 */
    u8 owner;           /* 当前拥有者（互斥锁专用） */
    u8 wait_cnt;        /* 当前等待任务数量 */
    u16 wait_mask;      /* 位图等待队列 */
    u8 pending_signal; /* ISR 中断计数 */
} OS_RESOURCE;
```

成员	类型	说明
---	---	---

value u8 资源值
owner u8 当前拥有者（互斥锁专用）
wait_cnt u8 当前等待任务数量
wait_mask u16 位图等待队列
pending_signal u8 ISR 中断计数

12.3 宏定义

本模块使用 config.h 中定义的等待类型和唤醒标志宏：

12.3.1 等待类型

宏定义 值 说明
--- --- ---
WAIT_NONE 0 无等待
WAIT_DELAY 1 延时等待
WAIT_SEM 2 信号量等待
WAIT_MUTEX 3 互斥锁等待
WAIT_MSG_SEND 4 消息队列发送等待
WAIT_MSG_RECV 5 消息队列接收等待
WAIT_EVENT 6 事件等待
WAIT_MAILBOX 7 邮箱等待

12.3.2 唤醒标志

宏定义 值 说明
--- --- ---
WAIT_TIMEOUT 1 超时唤醒
WAIT_SIGNAL 2 信号唤醒

12.4 API 函数列表

函数名称 功能说明
--- ---
os_wait 统一等待核心
os_res_init 资源对象初始化
wake_task 唤醒任务

12.5 API 详细说明

12.5.1 os_wait

函数原型:

```
u8 os_wait(u8 type, u8 obj, u16 tick);
```

参数说明:

| 参数 | 类型 | 说明 |

---|---|---

| type | u8 | 等待类型 (WAIT_*) |

| obj | u8 | 等待对象 ID (资源编号) |

| tick | u16 | 超时时间 (tick 数) |

返回值:

- WAIT_TIMEOUT: 超时唤醒
- WAIT_SIGNAL: 资源/事件唤醒
- 0: 等待失败

功能说明:

统一等待核心, 所有阻塞操作统一进入该函数。支持的等待类型:

1. 延时等待 (WAIT_DELAY)
2. 信号量等待 (WAIT_SEM)
3. 互斥锁等待 (WAIT_MUTEX)
4. 消息发送等待 (WAIT_MSG_SEND)
5. 消息接收等待 (WAIT_MSG_RECV)
6. 事件等待 (WAIT_EVENT)
7. 邮箱等待 (WAIT_MAILBOX)

****延时等待处理: ****

- tick = 0 时直接返回 WAIT_TIMEOUT
- 高速任务独立处理, 不进入统一等待队列
- 标准任务设置等待信息并触发调度

****资源等待处理: ****

- 检查任务是否已处于等待状态 (不允许重复等待)
- 检查资源 ID 是否合法
- 设置等待信息 (wait_type、wait_obj、wait_tick、wait_flag)
- 加入资源等待队列 (wait_mask)
- 阻塞当前任务 (state = WAIT)
- 触发任务切换

****等待队列管理: ****

- 使用位图管理等待任务 (wait_mask)

- bit0 对应 task0, bit1 对应 task1, ...
- WAIT_DELAY 属于纯时间等待, 不进入资源等待队列
- **任务切换:**
- 设置 OS_SCHED_REASON = 1 触发调度
- 设置 TF0 = 1 触发 T0 中断
- 任务被唤醒后返回等待结果

调用示例:

```
// 延时等待
u8 result = os_wait(WAIT_DELAY, OS_INVALID_ID, 100);

// 信号量等待
result = os_wait(WAIT_SEM, 0, 0);

// 事件等待 (带超时)
result = os_wait(WAIT_EVENT, 0, 100);
```

注意事项:

- 不允许重复进入等待
- 资源 ID 必须合法 (非延时等待)
- 会阻塞当前任务
- 被唤醒后返回等待结果
- 高速任务延时独立处理

12.5.2 os_res_init

函数原型:

```
void os_res_init(OS_RESOURCE *r);
```

参数说明:

参数	类型	说明
r	OS_RESOURCE*	资源对象指针

返回值:

无

功能说明:

初始化资源对象, 清空所有字段:

1. 清空资源值 (r->value = 0)

2. 清空拥有者 (r->owner = OS_INVALID_ID)

3. 清空等待计数 (r->wait_cnt = 0)

4. 清空等待位图 (r->wait_mask = 0)

调用示例:

```
OS_RESOURCE my_res;
os_res_init(&my_res);
```

注意事项：

- 用于初始化统一资源对象
- 信号量、互斥锁、事件、邮箱都使用该函数
- 初始化后资源为空闲状态

12.5.3 wake_task

函数原型：

```
void wake_task(u8 tid, u8 flag);
```

参数说明：

参数	类型	说明
----	----	----

---	---	---
-----	-----	-----

tid	u8	任务编号
-----	----	------

flag	u8	唤醒标志 (WAIT_TIMEOUT/WAIT_SIGNAL)
------	----	---------------------------------

返回值：

无

功能说明：

唤醒指定任务，执行以下操作：

1. 保存等待对象 (obj = OS_TASK[tid].wait_obj)
2. 清理等待位图 (OS_RES[obj].wait_mask &= ~((u16)1 << tid))
3. 减少等待计数 (OS_RES[obj].wait_cnt--)
4. 设置唤醒结果 (OS_TASK[tid].wait_flag = flag)
5. 清理等待上下文 (wait_type、wait_obj、wait_tick)
6. 设置就绪态 (OS_TASK[tid].state = READY)
7. 允许调度 (OS_PROCESS_OK[tid] |= 1)
8. 触发调度 (OS_SCHED_REASON = 1, TF0 = 1)

调用示例：

```
wake_task(0, WAIT_SIGNAL); // 唤醒任务 0，信号唤醒
```

注意事项：

- 任务必须处于等待状态
- 会清理任务的所有等待信息
- 会触发任务调度
- flag 必须是合法的唤醒标志