

README

1 HRTOS 系统手册

1.1 概述

本文档面向 HRTOS 内核开发者，介绍系统架构、核心机制、内部实现以及性能测试结果。

HRTOS (Hard Real-Time Operating System) 是一款面向 8051 微控制器优化的轻量级实时操作系统，专注于有限资源环境下的实时调度、高效任务切换以及低内存占用。

本文档主要用于：

- 理解 HRTOS 内核设计
- 分析核心模块实现
- 调试系统运行问题
- 进行功能扩展
- 评估系统实时性能

对于应用开发和 API 使用，请参考用户开发文档。

1.2 文档结构

本文档包含以下内容：

- 内核概述
- 系统架构
- 目录结构
- 调度器设计
- 任务管理
- IPC 机制设计
- 定时器系统
- 中断处理机制
- 内存布局
- 移植设计
- 内核启动流程
- 状态转换

- 模块依赖
- 调用流程
- 核心数据结构
- 设计原则
- 扩展指南
- 调试方法
- 性能测试

1.3 系统特性

- **目标平台**：8051 微控制器
- **编译环境**：Keil C51
- **任务模型**：标准任务 + 高速任务
- **调度方式**：基于优先级的抢占式调度
- **时间管理**：Timer0 系统 Tick
- **IPC 支持**：
 - 事件
 - 消息队列
 - 信号量
 - 互斥锁
 - 邮箱
- **内存管理**：静态分配，资源状态统一管理

1.4 快速参考

1.4.1 关键数据结构

- `OS_TCB`：任务控制结构
- `OS_RESOURCE`：统一资源对象
- 消息队列结构
- 调度状态变量

1.4.2 关键配置

- `OS_PROCESS_MAX`：标准任务数量
- `OS_FAST_TASK_MAX`：高速任务数量
- `OS_RESOURCE_MAX`：IPC 资源数量
- `OS_MSGQ_MAX`：消息队列数量

1.4.3 内存布局

示例配置：

- 系统栈区域：0x35 - 0x39
- 中断栈区域：0x3A - 0x41
- 用户栈区域：0x42 - 0x7F
- XDATA 全局区域：0x0200 - 0x03FF

1.5 系统要求

1.5.1 最低硬件要求

- 标准 8051 内核 MCU
- 片外 RAM $\geq$ 1KB
- Flash $\geq$ 8KB

1.5.2 推荐硬件配置

- STC12 / STC8 系列
- RAM $\geq$ 2KB
- Flash $\geq$ 16KB

1.5.3 开发环境

- Keil C51（主要支持环境）

1.6 使用说明

本文档适用于：

- 内核研究
- 系统调试
- 功能扩展
- 平台移植

应用开发请参考 HRTOS 文档及官方网站 hrtos.com。

Kernel Overview

2 HRTOS 内核概述

2.1 模块介绍

HRTOS（硬实时操作系统）是专为 8051 系列微控制器设计的轻量级实时操作系统。内核提供基本的 RTOS 服务，包括任务管理、进程间通信（IPC）、定时服务和中断处理，同时保持最小的内存占用，适合资源受限的嵌入式系统。

2.2 主要职责

HRTOS 内核负责：

- **任务管理**：创建、调度和管理多个并发任务
- **调度**：基于优先级的抢占式调度，可选时间片模式
- **IPC 服务**：事件标志、消息队列、互斥锁、信号量和邮箱
- **定时服务**：系统时钟生成、延时函数和运行时间跟踪
- **中断管理**：ISR 安全 API 和临界区保护
- **内存管理**：静态内存分配，使用位图跟踪栈空间
- **上下文切换**：调度期间保存和恢复任务上下文

2.3 主要文件

2.3.1 头文件（Inc/）

- ``hrtos.h``：包含所有公共 API 的主头文件
- ``config.h``：系统配置、类型定义和数据结构
- ``hrtos_internal.h``：内核内部声明和内存布局
- ``kernel.h``：内核初始化和核心 API
- ``task.h``：任务管理 API
- ``event.h``：事件同步 API
- ``interrupt.h``：中断处理和临界区 API
- ``mutex.h``：互斥锁（互斥）API
- ``semaphore.h``：信号量 API
- ``time.h``：定时和延时 API

- `wait.h`: 统一等待机制
- `mailbox.h`: 邮箱 API
- `msgq.h`: 消息队列 API

2.3.2 源文件 (Src/)

2.3.2.1 内核模块 (Src/kernel/)

- `os\_core.c`: 核心内核数据结构和内存布局
- `scheduler.c`: 调度器选择和模式切换
- `scheduler\_dispatch.c`: 任务分发机制
- `scheduler\_mode.c`: 调度模式配置
- `task\_create.c`: 任务创建和注册
- `task\_cleanup.c`: 任务清理和资源释放
- `idle\_task.c`: 空闲任务实现
- `idle\_hook\_register.c`: 空闲钩子注册
- `os\_data.c`: DATA 段初始化
- `os\_xdata.c`: XDATA 段初始化

2.3.2.2 任务模块 (Src/task/)

- `task\_delete.c`: 任务删除
- `task\_exit.c`: 任务退出
- `task\_is\_valid.c`: 任务有效性检查
- `task\_resume.c`: 任务恢复
- `task\_scheduler.c`: 任务调度器
- `task\_self.c`: 获取当前任务 ID
- `task\_state.c`: 任务状态查询
- `task\_suspend.c`: 任务挂起
- `task\_suspend\_self.c`: 自挂起
- `task\_yield.c`: 任务让出

2.3.2.3 事件模块 (Src/event/)

- `event\_delete.c`: 事件删除
- `event\_init.c`: 事件初始化
- `event\_query.c`: 事件查询
- `event\_wait.c`: 事件等待
- `event\_write.c`: 事件写入

2.3.2.4 中断模块 (Src/interrupt/)

- `enter\_critical.c`: 进入临界区
- `event\_set\_from\_isr.c`: ISR 安全的事件设置

- `exit\_critical.c`: 退出临界区
- `in\_isr.c`: ISR 上下文检查
- `interrupt\_id.c`: 中断 ID 查询
- `msgq\_rcv\_from\_isr.c`: ISR 安全的消息接收
- `msgq\_send\_from\_isr.c`: ISR 安全的消息发送
- `queue\_send\_from\_isr.c`: ISR 安全的队列发送
- `schedule\_request.c`: 调度请求
- `semaphore\_post\_from\_isr.c`: ISR 安全的信号量释放

2.3.2.5 互斥锁模块 (Src/mutex/)

- `mutex\_init.c`: 互斥锁初始化
- `mutex\_lock.c`: 具有优先级继承的互斥锁加锁
- `mutex\_unlock.c`: 互斥锁解锁

2.3.2.6 信号量模块 (Src/semaphore/)

- `sem\_init.c`: 信号量初始化
- `sem\_post.c`: 信号量释放 (V 操作)
- `sem\_wait.c`: 信号量等待 (P 操作)

2.3.2.7 定时模块 (Src/time/)

- `delay.c`: 任务延时
- `delay\_ms.c`: 毫秒延时 (忙等待)
- `ms\_to\_tick.c`: 毫秒到时钟周期转换
- `sec\_to\_tick.c`: 秒到时钟周期转换
- `tick\_config.c`: 时钟配置
- `tick\_get.c`: 获取时钟周期计数
- `uptime\_ms.c`: 获取运行时间 (毫秒)

2.3.2.8 等待模块 (Src/wait/)

- `os\_wait.c`: 统一等待核心
- `res\_init.c`: 资源初始化
- `wake\_task.c`: 任务唤醒

2.3.2.9 邮箱模块 (Src/mailbox/)

- `mailbox\_clear.c`: 邮箱清除
- `mailbox\_init.c`: 邮箱初始化
- `mailbox\_query.c`: 邮箱查询
- `mailbox\_rcv.c`: 邮箱接收
- `mailbox\_send.c`: 邮箱发送

2.3.2.10 消息队列模块 (Src/msgq/)

- `msgq\_clear.c`: 消息队列清除
- `msgq\_init.c`: 消息队列初始化
- `msgq\_recv.c`: 消息队列接收
- `msgq\_send.c`: 消息队列发送

2.3.2.11 任务优先级模块 (Src/task_priority/)

- `lock.c`: 优先级锁定
- `lock\_query.c`: 优先级锁定查询
- `task\_get\_priority.c`: 获取任务优先级
- `task\_set\_priority.c`: 设置任务优先级
- `unlock.c`: 优先级解锁

2.3.2.12 移植模块 (Src/port/)

- `cpu\_nop.c`: CPU 空操作

2.4 数据结构

2.4.1 OS_TCB (任务控制块)

位于 `config.h`, 任务控制块包含:

```
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;          /* 等待结果标志 */
    u8 wait_obj;           /* 等待对象 ID */
    u16 wait_tick;         /* 超时时钟周期 */
} OS_TCB;
```

2.4.2 OS_RESOURCE (统一 IPC 资源)

位于 `config.h`, 统一资源对象:

```
typedef struct {
    u8 value;              /* 资源值 */
    u8 owner;              /* 当前所有者 (仅互斥锁) */
    u8 wait_cnt;           /* 等待任务计数 */
    u16 wait_mask;         /* 位图等待队列 */
    u8 pending_signal;     /* ISR 信号计数 */
} OS_RESOURCE;
```

2.4.3 os_msgq_t (消息队列)

位于 `config.h`:

```
typedef struct {
    u8 *buf;               /* 缓冲区指针 */
    u8 _size;              /* 缓冲区大小 */
    u8 head;               /* 写指针 (生产者) */
    u8 tail;               /* 读指针 (消费者) */
}
```

```
    u8 count;                /* 当前计数 */  
} os_msgq_t;
```

2.5 核心函数

2.5.1 内核初始化

- `os\_interrupt\_init()`：初始化中断系统
- `os\_idle\_task()`：系统空闲任务
- `hrtos\_main()`：用户定义的主函数

2.5.2 任务管理

- `os\_task\_create()`：创建并注册任务
- `os\_task\_cleanup()`：清理任务资源
- `os\_task\_exit()`：退出当前任务
- `os\_task\_delete()`：删除任务

2.5.3 调度

- `os\_set\_scheduler()`：设置调度器模式（HRTOS/MYOS）
- `os\_scheduler\_mode\_switch()`：切换调度模式
- `os\_schedule\_request()`：请求重新调度

2.5.4 等待机制

- `os\_wait()`：统一等待核心
- `wake\_task()`：唤醒任务

2.5.5 临界区

- `os\_enter\_critical()`：进入临界区
- `os\_exit\_critical()`：退出临界区

2.6 调用关系

2.6.1 初始化流程

```
系统复位  
↓  
os_idle_task() [入口点]  
↓  
初始化 TCB 数组  
初始化资源数组  
初始化消息队列  
↓  
hrtos_main() [用户初始化]  
↓
```

```
os_task_create() 调用
↓
OS_CFG_LOCK = 1 (锁定注册)
↓
启用中断 (EA = 1)
↓
带有任务清理的空闲循环
```

2.6.2 任务创建流程

```
os_task_create(addr, id, pr, sd)
↓
参数验证
↓
栈分配 (位图搜索)
↓
栈指针计算
↓
上下文初始化
↓
任务状态设置
↓
返回成功
```

2.6.3 等待流程

```
os_wait(type, obj, tick)
↓
进入临界区
↓
设置 TCB 等待字段
↓
添加到资源等待队列 (如适用)
↓
设置任务状态为 WAIT
↓
触发调度
↓
退出临界区
↓
任务阻塞直到调用 wake_task()
```

2.7 生命周期

2.7.1 系统生命周期

1. ****启动****: 系统进入 ``os_idle_task()``
2. ****初始化****: 内核结构初始化
3. ****用户初始化****: 调用 ``hrtos_main()`` 创建任务
4. ****注册锁定****: 任务注册锁定
5. ****调度****: 启用中断, 调度器激活
6. ****运行时****: 任务基于优先级执行
7. ****空闲****: 空闲任务在其他任务未就绪时运行

2.7.2 任务生命周期

1. ****创建****: ``os_task_create()`` 分配栈并初始化 TCB
2. ****就绪****: 任务标记为可调度
3. ****运行****: 任务被调度器选中时执行
4. ****等待****: 任务在 IPC 或延时上阻塞
5. ****唤醒****: ``wake_task()`` 将任务返回到就绪状态
6. ****退出****: ``os_task_exit()`` 或 ``os_task_delete()`` 移除任务

2.8 设计原则

2.8.1 内存效率

- 仅静态分配（无动态内存）
- 基于位图的栈管理
- 紧凑的数据结构
- XDATA 段用于大型数组

2.8.2 实时性能

- 基于优先级的抢占式调度
- 快速上下文切换
- 最小中断延迟
- ISR 安全 API

2.8.3 8051 优化

- DATA 段用于频繁访问的变量
- IDATA 用于内核状态
- XDATA 用于大型数据结构
- 关键路径的汇编优化

2.8.4 统一 IPC 模型

- 所有 IPC 类型使用单一 ``OS_RESOURCE`` 结构
- 通过 ``os_wait()`` 统一等待机制
- 跨 IPC 类型的一致 API 模式

2.9 约束

- 最多 16 个标准任务
- 最多 2 个快速任务

- 最多 8 个 IPC 资源
- 最多 4 个消息队列
- 优先级级别 0-10
- 无动态内存分配
- 无递归互斥锁支持
- 栈溢出保护有限

Directory Analysis

3 HRTOS 目录分析

3.1 模块介绍

本文档全面分析了 HRTOS 项目目录结构，解释了源文件、头文件、文档和示例的组织方式。

3.2 主要职责

目录结构用于：

- 分离公共 API 和内部实现
- 按功能组织内核模块
- 提供面向用户和内部代码之间的清晰边界
- 通过硬件抽象层保持可移植性
- 支持教学和开发工作流程

3.2.1 Inc/ 目录（公共头文件）

****目的****：包含所有公共 API 声明和配置定义。此目录对用户是只读的，定义了应用程序和内核之间的接口。

****文件****：

- `hrtos.h`：包含所有其他头文件的主包含文件
- `config.h`：系统配置、类型定义和数据结构
- `kernel.h`：内核初始化和核心 API
- `task.h`：任务管理 API
- `event.h`：事件同步 API
- `interrupt.h`：中断处理和临界区 API
- `mutex.h`：互斥锁 API
- `semaphore.h`：信号量 API
- `time.h`：定时和延时 API
- `wait.h`：统一等待机制
- `mailbox.h`：邮箱 API

- `msgq.h`: 消息队列 API

****关键特性**:**

- 定义公共 API 签名
- 包含配置宏
- 暴露数据结构定义
- 无实现细节
- 应用程序代码包含安全

3.2.2 Src/ 目录 (源实现)

****目的**:** 包含按功能模块组织的所有内核实现。此目录对用户是只读的，包含实际的内核代码。

****子目录**:**

3.2.2.1 Src/kernel/ - 核心内核

****文件**:**

- `os\_core.c`: 核心数据结构、内存布局定义
- `scheduler.c`: HRTOS 和 MYOS 模式之间的调度器选择
- `scheduler\_dispatch.c`: 任务分发实现
- `scheduler\_mode.c`: 调度模式切换
- `task\_create.c`: 任务创建和注册逻辑
- `task\_cleanup.c`: 任务清理和资源释放
- `idle\_task.c`: 空闲任务实现
- `idle\_hook\_register.c`: 空闲钩子注册
- `os\_data.c`: DATA 数据搬运
- `os\_xdata.c`: XDATA 数据搬运

****职责**:**

- 系统初始化
- 任务生命周期管理
- 调度核心逻辑
- 内存布局定义
- 空闲任务管理

3.2.2.2 Src/task/ - 任务管理

****文件**:**

- `task\_delete.c`: 任务删除实现
- `task\_exit.c`: 当前任务退出
- `task\_is\_valid.c`: 任务有效性验证
- `task\_resume.c`: 恢复挂起的任务

- `task\_scheduler.c`: 任务调度逻辑
- `task\_self.c`: 获取当前任务 ID
- `task\_state.c`: 查询任务状态
- `task\_suspend.c`: 挂起任务
- `task\_suspend\_self.c`: 自挂起
- `task\_yield.c`: 让出 CPU 给其他任务

****职责**:**

- 任务状态转换
- 任务控制操作
- 优先级管理
- 状态查询

3.2.2.3 Src/event/ - 事件同步

****文件**:**

- `event\_delete.c`: 删除事件对象
- `event\_init.c`: 初始化事件
- `event\_query.c`: 查询事件状态
- `event\_wait.c`: 等待事件
- `event\_write.c`: 设置/写入事件

****职责**:**

- 事件标志管理
- 事件同步
- ISR 安全事件操作

3.2.2.4 Src/interrupt/ - 中断处理

****文件**:**

- `enter\_critical.c`: 进入临界区（禁用中断）
- `event\_set\_from\_isr.c`: ISR 安全的事件设置
- `exit\_critical.c`: 退出临界区（恢复中断）
- `in\_isr.c`: 检查是否在 ISR 上下文中
- `interrupt\_id.c`: 获取当前中断 ID
- `msgq\_rcv\_from\_isr.c`: ISR 安全的消息接收
- `msgq\_send\_from\_isr.c`: ISR 安全的消息发送
- `queue\_send\_from\_isr.c`: ISR 安全的队列发送
- `schedule\_request.c`: 请求任务重新调度
- `semaphore\_post\_from\_isr.c`: ISR 安全的信号量释放

****职责**:**

- 临界区管理
- ISR 上下文检测
- ISR 安全 IPC 操作
- 中断驱动的调度

3.2.2.5 Src/mutex/ - 互斥

****文件**:**

- `mutex\_init.c`: 初始化互斥锁
- `mutex\_lock.c`: 具有优先级继承的互斥锁获取
- `mutex\_unlock.c`: 释放互斥锁

****职责**:**

- 互斥锁状态管理
- 优先级继承实现
- 死锁预防

3.2.2.6 Src/semaphore/ - 计数信号量

****文件**:**

- `sem\_init.c`: 初始化信号量
- `sem\_post.c`: 信号量 V 操作（释放）
- `sem\_wait.c`: 信号量 P 操作（获取）

****职责**:**

- 信号量计数/资源管理
- 资源获取/释放
- 等待队列管理

3.2.2.7 Src/time/ - 定时服务

****文件**:**

- `delay.c`: 基于时钟周期的任务延时
- `delay\_ms.c`: 忙等待毫秒延时
- `ms\_to\_tick.c`: 将毫秒转换为时钟周期
- `sec\_to\_tick.c`: 将秒转换为时钟周期
- `tick\_config.c`: 配置系统时钟
- `tick\_get.c`: 获取当前时钟周期计数
- `uptime\_ms.c`: 获取系统运行时间（毫秒）

****职责**:**

- 系统时钟管理
- 时间转换工具
- 延时实现

- 运行时间跟踪

3.2.2.8 Src/wait/ - 统一等待

****文件**:**

- `os\_wait.c`: 统一等待核心实现
- `res\_init.c`: 资源初始化
- `wake\_task.c`: 任务唤醒实现

****职责**:**

- 统一等待机制
- 任务唤醒逻辑
- 资源初始化

3.2.2.9 Src/mailbox/ - 邮箱 IPC

****文件**:**

- `mailbox\_clear.c`: 清除邮箱
- `mailbox\_init.c`: 初始化邮箱
- `mailbox\_query.c`: 查询邮箱状态
- `mailbox\_recv.c`: 从邮箱接收
- `mailbox\_send.c`: 发送到邮箱

****职责**:**

- 邮箱状态管理
- 单字节消息传递
- 邮箱同步

3.2.2.10 Src/msgq/ - 消息队列

****文件**:**

- `msgq\_clear.c`: 清除消息队列
- `msgq\_init.c`: 初始化消息队列
- `msgq\_recv.c`: 从消息队列接收
- `msgq\_send.c`: 发送到消息队列

****职责**:**

- 消息队列管理
- 循环缓冲区实现
- 生产者-消费者同步

3.2.2.11 Src/task_priority/ - 优先级管理

****文件**:**

- `lock.c`: 锁定任务注册
- `lock\_query.c`: 查询优先级锁定状态

- `task\_get\_priority.c`：获取任务优先级
- `task\_set\_priority.c`：设置任务优先级
- `unlock.c`：解锁任务注册

****职责**：**

- 任务注册锁定机制
- 优先级查询和修改
- 优先级更改保护

3.2.2.12 Src/port/ - 硬件抽象

****文件**：**

- `cpu\_nop.c`：CPU 空操作指令

****职责**：**

- 硬件特定操作
- 不同 MCU 的移植层

3.3 核心函数

3.3.1 文件包含层次

1. ****应用程序层****：包含 `Inc/hrtos.h`
2. ****公共头文件层****：`hrtos.h` 包含所有其他公共头文件
3. ****实现层****：源文件包含 `hrtos.h`

3.3.2 模块初始化顺序

基于目录结构，初始化遵循：

1. 核心内核 (Src/kernel/)
2. 任务管理 (Src/task/)
3. IPC 模块 (Src/event/、Src/mutex/、Src/semaphore/ 等)
4. 定时 (Src/time/)
5. 中断处理 (Src/interrupt/)

3.4 调用关系

3.4.1 模块间依赖

```
Src/kernel/
├── 使用 → Src/wait/
├── 使用 → Src/task/
└── 使用 → Src/interrupt/

Src/task/
├── 使用 → Src/wait/
```

```
└─ 使用 → Src/task_priority/

Src/event/
└─ 使用 → Src/wait/

Src/mutex/
└─ 使用 → Src/wait/

Src/semaphore/
└─ 使用 → Src/wait/

Src/mailbox/
└─ 使用 → Src/wait/

Src/msgq/
└─ 使用 → Src/wait/

Src/interrupt/
├─ 使用 → Src/event/
├─ 使用 → Src/msgq/
└─ 使用 → Src/semaphore/

Src/time/
└─ 使用 → Src/wait/
```

3.4.2 头文件依赖图

```
hrtos.h (主入口)
├─ config.h
├─ wait.h
├─ event.h
├─ interrupt.h
├─ kernel.h
├─ mailbox.h
├─ mutex.h
├─ task.h
├─ semaphore.h
├─ msgq.h
└─ time.h
```

Scheduler Design

4 HRTOS 调度器设计

4.1 模块介绍

HRTOS 调度器负责选择在 CPU 上运行的任务。它实现了基于优先级的抢占式调度算法，具有可选的时间片模式，旨在满足 8051 微控制器的实时要求。

4.2 主要职责

调度器处理：

- 基于优先级的任务选择
- 抢占式任务切换
- 时间片管理
- 快速任务调度
- 中断级任务处理
- 调度器模式切换（HRTOS vs MYOS）

4.3 主要文件

4.3.1 源文件

- `Src/kernel/scheduler.c`：调度器模式选择
- `Src/kernel/scheduler\_dispatch.c`：任务分发机制
- `Src/kernel/scheduler\_mode.c`：调度模式配置
- `Src/kernel/task\_create.c`：任务注册（设置调度信息）
- `Src/task/task\_scheduler.c`：任务调度逻辑
- `Src/task/task\_yield.c`：自愿 CPU 让出
- `Src/interrupt/schedule\_request.c`：来自 ISR 的调度请求

4.3.2 头文件

- `Inc/kernel.h`：调度器 API 声明
- `Inc/config.h`：调度级别定义

4.4 数据结构

4.4.1 调度级别定义

位于 `Inc/config.h`:

```
/* 调度级别定义:
* 0-7    普通任务优先级
* 8      高优先级任务
* 9      中断级
* 10     中断嵌套级
*/
#define OS_SCHED_LEVEL_MAX 10
```

4.4.2 任务调度信息

位于 `Src/kernel/os\_core.c`:

```
volatile unsigned char idata OS_CURRENT_TASK; /* 当前运行任务 */
volatile unsigned char idata OS_PREV_TASK;   /* 前一个任务 */
volatile unsigned char idata OS_TIME_XY;     /* 当前时间片 */
volatile unsigned char idata OS_DISPATCH_ID; /* 分发触发源 */
volatile unsigned char xdata OS_TIME_ONCE;   /* 时间片计数 */
```

4.4.3 任务状态数组

```
volatile unsigned char xdata OS_PROCESS_OK[OS_PROCESS_MAX];
/* 格式: 位 0: 状态, 位 1-3: 优先级, 位 4-6: 栈大小 */
```

4.4.4 快速任务标志

```
volatile bit OS_KUAI_PROCESS_A; /* 快速任务 0 激活 */
volatile bit OS_KUAI_PROCESS_B; /* 快速任务 1 激活 */
```

4.5 核心函数

4.5.1 os_set_scheduler()

****位置**:** `Src/kernel/scheduler.c`

****目的**:** 在 HRTOS 和 MYOS 调度模式之间切换

****参数**:**

- `id`: 1 表示 HRTOS 模式, 0 表示 MYOS 模式

****返回**:** 成功返回 1, 失败返回 -1

****逻辑**:**

```
char os_set_scheduler(unsigned char id)
{
    if((id==0)&&(OS_INTERRUPT_T0==0)) // 切换到 MYOS
    {
        OS_INTERRUPT_T0=1;
        OS_TIME_ONCE_BACKUP=OS_TIME_ONCE;
        return 1;
    }
    else if((id==1)&&(OS_INTERRUPT_T0==1)) // 切换到 HRTOS
```

```

{
    OS_INTERRUPT_T0=0;
    OS_TIME_XY=OS_TIME_XY+OS_TIME_ONCE_BACKUP-OS_TIME_ONCE;
    OS_TIME_ONCE=OS_TIME_ONCE_BACKUP;
    return 1;
}
return -1;
}

```

4.5.2 os_scheduler_mode_switch()

****位置****: `Src/kernel/scheduler\_mode.c`

****目的****: 在实时和通用调度模式之间切换

****参数****:

- `id`: 1 表示实时模式, 0 表示通用模式

4.5.3 os_schedule_request()

****位置****: `Src/interrupt/schedule\_request.c`

****目的****: 通过触发 Timer0 中断请求立即重新调度

****逻辑****:

```

char os_schedule_request(char id)
{
    id = id; /* 保留供将来使用 */
    OS_SCHED_REASON=1; /* 设置调度触发标志 */
    TF0=1; /* 触发 Timer0 中断 */
    return 1;
}

```

4.5.4 os_task_yield()

****位置****: `Src/task/task\_yield.c`

****目的****: 自愿将 CPU 让给相同优先级的其他任务

****实现****: 设置调度触发并强制上下文切换

4.6 调用关系

4.6.1 调度触发流程

```

graph TD
    A[任务调用 os_delay/os_wait] --> B[设置 OS_SCHED_REASON=1]
    B --> C[设置 TF0=1]
    C --> D[Timer0 ISR]
    D --> E[调度器分发]
    E --> F[上下文切换]

    G[ISR 调用 os_schedule_request] --> B
    H[任务调用 os_task_yield] --> B

```

4.6.2 优先级选择流程

```
graph TD
    A[调度器扫描] --> B{快速任务就绪?}
    B -->|是| C[运行快速任务]
    B -->|否| D{中断级?}
    D -->|是| E[运行中断处理程序]
    D -->|否| F[扫描标准任务]
    F --> G{任务就绪?}
    G -->|是| H[选择最高优先级]
    G -->|否| I[运行空闲任务]
    H --> J[上下文切换]
```

4.7 生命周期

4.7.1 调度器初始化

1. 系统启动进入 `os\_idle\_task()`
2. `OS\_TIME\_XY` 初始化为时间片计数
3. `OS\_CURRENT\_TASK` 设置为 0（空闲）
4. `OS\_SCHED\_REASON` 清除
5. 启用中断
6. Timer0 开始生成时钟周期

4.7.2 调度周期

1. ****时钟中断****: Timer0 ISR 触发
2. ****时间片递减****: `OS\_TIME\_XY++`
3. ****时间片检查****: 如果 `OS\_TIME\_XY == OS\_TIME\_ONCE`，触发调度
4. ****优先级扫描****: 从最高到最低优先级扫描任务
5. ****任务选择****: 选择最高优先级的就绪任务
6. ****上下文切换****: 保存当前任务，恢复选定任务
7. ****执行****: 选定的任务运行

4.7.3 快速任务调度

快速任务（优先级 8）有特殊处理：

- 使用专用上下文保存区域
- 具有独立的延时计数器（`OS\_WAIT\_DI2`）
- 可以抢占标准任务
- 最小上下文保存（5 个寄存器 vs 标准任务的 13 个）

4.8 设计原则

4.8.1 基于优先级的抢占

- 高优先级任务总是抢占低优先级任务

- 标准任务的优先级级别 0-7
- 快速任务的优先级级别 8
- 中断处理程序的优先级级别 9-10

4.8.2 时间片模式

- 可配置的时间片计数（`OS\_TIME\_ONCE`）
- 可配置的时钟周期持续时间（`OS\_TIME\_T0`）
- 相同优先级任务的时间片相等
- 可以禁用时间片以进行纯优先级调度

4.8.3 快速任务优化

- 减少上下文保存深度
- 独立的延时机制
- 专用寄存器组
- 最小调度开销

4.8.4 双调度器支持

- HRTOS 模式：基于优先级，带时间片
- MYOS 模式：替代调度算法
- 通过 `os\_set\_scheduler()` 运行时切换

4.9 约束

- 最多 10 个优先级级别
- 最多 16 个标准任务
- 最多 2 个快速任务
- 快速任务不能使用标准等待机制
- 优先级继承仅适用于互斥锁（不适用于调度器）
- 没有时间片时相同优先级内没有轮询

4.10 调度算法

4.10.1 就绪队列选择

调度器按优先级顺序扫描任务：

1. 检查中断处理程序（优先级 9-10）
2. 检查快速任务（优先级 8）
3. 扫描标准任务（优先级 1-7）

4. 回退到空闲任务（优先级 0）

4.10.2 任务状态位

``OS_PROCESS_OK[id]`` 编码：

- 位 0：就绪标志（1 = 就绪，0 = 未就绪）
- 位 1-3：优先级级别
- 位 4-6：栈大小

4.10.3 上下文切换触发

调度在以下情况下触发：

- 时间片到期（``OS_TIME_XY == OS_TIME_ONCE``）
- 任务自愿让出（``os_task_yield()``）
- 任务在 IPC 上阻塞（``os_wait()``）
- ISR 请求调度（``os_schedule_request()``）
- 任务唤醒（``wake_task()``）

Task Management

5 HRTOS 任务管理

5.1 模块介绍

任务管理是 HRTOS 的核心，负责创建、控制和管理并发任务的生命周期。它处理任务注册、状态转换、优先级管理和资源清理。

5.2 主要职责

任务管理处理：

- 任务创建和注册
- 任务状态转换（就绪、运行、等待、挂起、删除）
- 任务优先级管理
- 任务删除和清理
- 任务挂起和恢复
- 栈分配和管理
- 上下文保存/恢复协调

5.3 主要文件

5.3.1 源文件

- `Src/kernel/task\_create.c`：任务创建和注册
- `Src/kernel/task\_cleanup.c`：任务清理和资源释放
- `Src/task/task\_delete.c`：任务删除
- `Src/task/task\_exit.c`：当前任务退出
- `Src/task/task\_is\_valid.c`：任务有效性检查
- `Src/task/task\_resume.c`：恢复挂起的任务
- `Src/task/task\_scheduler.c`：任务调度逻辑
- `Src/task/task\_self.c`：获取当前任务 ID
- `Src/task/task\_state.c`：查询任务状态
- `Src/task/task\_suspend.c`：挂起任务

- `Src/task/task\_suspend\_self.c`：自挂起
- `Src/task/task\_yield.c`：让出 CPU
- `Src/task\_priority/task\_get\_priority.c`：获取任务优先级
- `Src/task\_priority/task\_set\_priority.c`：设置任务优先级
- `Src/task\_priority/lock.c`：锁定任务创建
- `Src/task\_priority/unlock.c`：解锁任务创建
- `Src/task\_priority/lock\_query.c`：查询任务创建锁定状态

5.3.2 头文件

- `Inc/task.h`：任务管理 API 声明
- `Inc/config.h`：任务配置和 TCB 定义
- `Inc/hrtos\_internal.h`：内部任务管理变量

5.4 数据结构

5.4.1 OS_TCB（任务控制块）

位于 `Inc/config.h`：

```
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;         /* 等待结果标志 */
    u8 wait_obj;          /* 等待对象 ID */
    u16 wait_tick;        /* 超时时钟周期 */
} OS_TCB;
```

5.4.2 任务状态数组

位于 `Src/kernel/os\_core.c`：

```
volatile unsigned char xdata OS_PROCESS_OK[OS_PROCESS_MAX];
/* 编码：
 * 位 0：就绪标志（1 = 可运行）
 * 位 1-3：优先级级别
 * 位 4-6：栈大小
 * 位 7：保留
 */
```

5.4.3 栈指针

```
volatile unsigned char xdata OS_JINCHENG_SP[OS_PROCESS_MAX]; /* 当前
SP */
volatile unsigned char xdata OS_JINCHENG_JILU_SP[OS_PROCESS_MAX]; /* 初
始 SP */
```

5.4.4 上下文保存区域

```
volatile unsigned char xdata OS_TASK_CONTEXT[OS_PROCESS_MAX]
[OS_JINCHENG_SHENDU];
/* 标准任务上下文：每个任务 13 个寄存器 */

volatile unsigned char xdata OS_FAST_CONTEXT[OS_FAST_TASK_MAX]
[OS_KUAI_SHENDU];
/* 快速任务上下文：每个任务 5 个寄存器 */
```

5.4.5 快速任务控制

```
volatile bit OS_KUAI_PROCESS_A; /* 快速任务 0 激活 */
volatile bit OS_KUAI_PROCESS_B; /* 快速任务 1 激活 */
volatile unsigned char xdata OS_KUAI_SP[OS_FAST_TASK_MAX];
volatile unsigned char xdata OS_SP_KUAI_BEI[OS_FAST_TASK_MAX];
volatile unsigned char xdata OS_SP_KUAI_BEI2[OS_FAST_TASK_MAX];
```

5.5 核心函数

5.5.1 os_task_create()

****位置****: `Src/kernel/task\_create.c`

****目的****: 创建并注册新任务

****参数****:

- `addr`: 任务函数地址
- `id`: 任务 ID (标准任务为 0-15, 快速任务为 16-17)
- `pr`: 优先级级别 (0-10)
- `sd`: 栈深度 (层数)

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证参数 (优先级、ID、地址)
2. 检查重复注册
3. 使用位图搜索分配栈空间
4. 计算初始栈指针
5. 将任务地址压入栈
6. 在 `OS\_PROCESS\_OK` 中设置任务状态
7. 对于快速任务: 设置激活标志和上下文
8. 对于中断处理程序: 将地址存储在中断向量中

****栈分配算法****:

```
// 基于位图的分配
for(i=0; i<(OS_USER_RAM_EXIT-OS_USER_RAM_INIT)/2; i++)
{
    x=i/8; // 位图字节索引
    OS_R0_PROTECT=i%8; // 位位置
    x=OS_MEMORY[x]&(0x80>>OS_R0_PROTECT);
    if(x==0) // 找到空闲块
```

```

{
    k++;
    if (k>=sd)
    {
        // 计数连续空闲块
        // 足够空间
        // 标记块为已使用
        for (j=i-k+1; j<=i; j++)
        {
            x=j/8;
            OS_R0_PROTECT=j%8;
            OS_MEMORY[x] |=(0x80>>OS_R0_PROTECT);
        }
        goto OS_SLAB_001_; // 成功
    }
}
else
{
    k=0;
    // 重置计数器
}
}

```

5.5.2 os_task_cleanup()

****位置****: `Src/kernel/task\_cleanup.c`

****目的****: 清理任务资源并释放栈

****参数****:

- `id`: 要清理的任务 ID

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证任务 ID
2. 检查任务是否存在
3. 计算栈块位置
4. 在位图中释放栈块
5. 清除上下文保存区域
6. 清除 `OS\_PROCESS\_OK` 中的任务状态
7. 清除 TCB 中的等待信息
8. 从资源等待队列中移除
9. 如果持有互斥锁则释放所有权
10. 将任务状态设置为 DEAD

5.5.3 任务状态函数

5.5.3.1 os_task_get_state()

****位置****: `Src/task/task\_state.c`

****目的****: 查询任务状态

****返回****:

- 0: 已删除

- 1: 运行中
- 2: 就绪
- 3: 挂起
- 4: 先前切换出

****逻辑**:**

```
if(OS_PROCESS_OK[id]==0) return 0;    // 已删除
if(id==OS_CURRENT_TASK) return 1;    // 运行中
if(id==OS_PREV_TASK) return 4;       // 前一个
if(OS_PROCESS_OK[id]&0x01) return 2;  // 就绪
return 3;                             // 挂起
```

5.5.3.2 os_task_suspend()

****位置**:** `Src/task/task\_suspend.c`

****目的**:** 挂起任务（从调度中移除）

****过程**:** 清除 `OS\_PROCESS\_OK` 中的就绪标志

5.5.3.3 os_task_resume()

****位置**:** `Src/task/task\_resume.c`

****目的**:** 恢复挂起的任务

****过程**:** 设置 `OS\_PROCESS\_OK` 中的就绪标志

5.5.3.4 os_task_delete()

****位置**:** `Src/task/task\_delete.c`

****目的**:** 完全删除任务

****过程**:** 设置删除请求标志，空闲任务将清理

5.5.3.5 os_task_exit()

****位置**:** `Src/task/task\_exit.c`

****目的**:** 当前任务自行退出

****过程**:** 设置删除请求并触发调度

5.5.4 优先级管理

5.5.4.1 os_task_set_priority()

****位置**:** `Src/task\_priority/task\_set\_priority.c`

****目的**:** 更改任务优先级

****参数**:**

- `id`: 任务 ID

- `pr`: 新优先级

****过程**:**

1. 验证优先级范围
2. 锁定优先级更改
3. 更新 `OS\_PROCESS\_OK` 中的优先级

4. 更新 TCB 优先级字段
5. 解锁优先级更改
6. 如需要则触发重新调度

5.5.4.2 os_task_get_priority()

****位置****: `Src/task\_priority/task\_get\_priority.c`

****目的****: 获取当前任务优先级

****返回****: 来自 `OS\_PROCESS\_OK` 的优先级值

5.6 调用关系

5.6.1 任务创建流程

```
graph TD
    A[os_task_create] --> B[参数验证]
    B --> C[栈分配]
    C --> D[位图搜索]
    D --> E[标记块已使用]
    E --> F[计算 SP]
    F --> G[压入任务地址]
    G --> H[设置 OS_PROCESS_OK]
    H --> I[返回成功]
```

5.6.2 任务删除流程

```
graph TD
    A[os_task_delete] --> B[设置删除请求]
    B --> C[触发调度]
    C --> D[空闲任务检测请求]
    D --> E[os_task_cleanup]
    E --> F[释放栈位图]
    E --> G[清除上下文]
    E --> H[清除 TCB]
    E --> I[释放资源]
    E --> J[设置状态 DEAD]
```

5.6.3 状态转换流程

```
stateDiagram-v2
    [*] --> 已创建: os_task_create
    已创建 --> 就绪: 初始状态
    就绪 --> 运行: 调度器选择
    运行 --> 就绪: 被抢占/让出
    运行 --> 等待: os_delay/os_wait
    运行 --> 挂起: os_task_suspend
    等待 --> 就绪: wake_task/超时
    挂起 --> 就绪: os_task_resume
    就绪 --> 已删除: os_task_delete
    运行 --> 已删除: os_task_exit
    已删除 --> [*]
```

5.7 生命周期

5.7.1 任务创建

1. **注册**：调用 ``os_task_create()``
2. **分配**：通过位图分配栈空间
3. **初始化**：设置栈指针，压入地址
4. **状态设置**：配置 ``OS_PROCESS_OK``
5. **就绪**：任务标记为可调度

5.7.2 任务执行

1. **选择**：调度器选择任务
2. **上下文恢复**：从保存区域加载上下文
3. **执行**：任务代码运行
4. **阻塞**：任务可能调用阻塞 API
5. **抢占**：高优先级任务可能抢占

5.7.3 任务删除

1. **请求**：调用 ``os_task_delete()`` 或 ``os_task_exit()``
2. **标志**：设置删除请求标志
3. **检测**：空闲任务检测请求
4. **清理**：调用 ``os_task_cleanup()``
5. **释放**：释放栈和资源
6. **最终**：任务状态设置为 DEAD

5.8 设计原则

5.8.1 静态分配

- 所有任务静态分配
- 任务创建无动态内存
- 通过位图预分配栈空间
- 最大任务数在编译时固定

5.8.2 基于优先级

- 优先级决定执行顺序
- 高优先级总是抢占低优先级
- 优先级可以在运行时更改
- 优先级继承的动态优先级

5.8.3 类型区分

- ****标准任务****：完整上下文保存，优先级 0-7
- ****快速任务****：最小上下文，优先级 8
- ****中断处理程序****：直接地址映射，优先级 9-10

5.8.4 资源清理

- 删除时自动清理
- 清除资源等待队列
- 释放互斥锁所有权
- 释放栈空间

5.9 约束

- 最多 16 个标准任务 (ID 0-15)
- 最多 2 个快速任务 (ID 16-17)
- 优先级级别限制为 0-10
- 栈深度受限 (标准任务最多 6，快速任务最多 25)
- 无递归任务创建
- 任务 0 保留给空闲任务
- 栈溢出检测有限

5.10 任务类型

5.10.1 标准任务

- ID：1-15 (ID 0 是空闲)
- 优先级：0-7
- 栈深度：1-6 层
- 上下文：13 个寄存器 (PWS、ACC、B、DPTR、寄存器组)
- 使用统一等待机制

5.10.2 快速任务

- ID：16-17
- 优先级：8
- 栈深度：1-25 层
- 上下文：5 个寄存器 (PWS、ACC、B、DPTR)
- 独立延时机制 ('OS_WAIT_DI2')
- 不能使用事件等待
- 独立工作寄存器组

5.10.3 中断处理程序

- 优先级：9-10
- 无任务 ID
- 直接地址映射
- 在中断上下文中执行
- 使用中断栈
- 调用 `os_interrupt_exit()` 退出中断

5.11 内存布局

5.11.1 栈分配

- 用户栈区域：0x42-0x7F（62 字节）
- 位图：XDATA 中 4 字节
- 块大小：每块 2 字节
- 最大块数：31 块

5.11.2 上下文保存区域

- 标准任务上下文：XDATA，每个任务 13 字节
- 快速任务上下文：XDATA，每个任务 5 字节
- 总上下文内存：208 字节（标准）+ 10 字节（快速）

6 HRTOS 事件设计

6.1 模块介绍

事件模块提供基于事件标志的任务协调轻量级同步机制。事件用于状态同步而不进行数据传输，允许任务等待特定条件发生。

6.2 主要职责

事件模块处理：

- 事件初始化和删除
- 事件标志设置（写入）
- 带超时的事件等待
- 事件状态查询
- ISR 安全事件操作

6.3 主要文件

6.3.1 源文件

- `Src/event/event\_init.c`：事件初始化
- `Src/event/event\_write.c`：事件标志设置
- `Src/event/event\_wait.c`：事件等待
- `Src/event/event\_query.c`：事件状态查询
- `Src/event/event\_delete.c`：事件删除
- `Src/interrupt/event\_set\_from\_isr.c`：ISR 安全的事件设置

6.3.2 头文件

- `Inc/event.h`：事件 API 声明
- `Inc/config.h`：事件类型定义
- `Inc/hrtos\_internal.h`：内部事件变量

6.4 数据结构

6.4.1 OS_RESOURCE (统一 IPC)

事件使用统一资源结构：

```
typedef struct {  
    u8 value;           /* 事件标志值 */  
    u8 owner;           /* 事件不使用 */  
    u8 wait_cnt;        /* 等待任务计数 */  
    u16 wait_mask;      /* 位图等待队列 */  
    u8 pending_signal;  /* ISR 信号计数 */  
} OS_RESOURCE;
```

对于事件：

- `value`：事件标志 (0 = 未设置, 非零 = 已设置)
- `wait\_mask`：等待此事件的任务位图
- `wait\_cnt`：等待任务计数

6.4.2 事件位图

位于 `Inc/hrtos\_internal.h`：

```
extern unsigned int xdata OS_EVENT_BIT;    // 事件位操作  
extern unsigned char xdata OS_TASK_EVENT[OS_PROCESS_MAX +  
OS_FAST_TASK_MAX];
```

6.5 核心函数

6.5.1 os_event_init()

****位置****：`Src/event/event\_init.c`

****目的****：初始化事件对象

****参数****：

- `obj`：资源 ID (0-7)

****返回****：成功返回 1，失败返回 -1

****过程****：

1. 验证资源 ID
2. 通过 `os\_res\_init()` 初始化资源结构
3. 将值设置为 0 (事件未设置)
4. 清除等待队列

6.5.2 os_event_write()

****位置****：`Src/event/event\_write.c`

****目的****：设置事件标志 (任务上下文)

****参数****：

- `id`：事件资源 ID

****返回****：成功返回 1，失败返回 -1

****过程**:**

1. 验证资源 ID
2. 进入临界区
3. 将事件值设置为 1
4. 唤醒所有等待任务
5. 退出临界区

6.5.3 os_event_wait()

****位置**:** `Src/event/event\_wait.c`

****目的**:** 等待事件被设置

****参数**:**

- `obj`: 事件资源 ID
- `tick`: 超时时钟周期

****返回**:** 成功返回 1, 失败返回 -1

****过程**:**

```
char os_event_wait(u8 obj, u16 tick)
{
    if(obj >= OS_RESOURCE_MAX) return -1;
    EA=0;
    /* 事件已设置, 立即返回 */
    if(OS_RES[obj].value)
    {
        return 1;
    }
    EA=1;
    /* 否则进入等待 */
    return os_wait(WAIT_EVENT, obj, tick);
}
```

6.5.4 os_event_query()

****位置**:** `Src/event/event\_query.c`

****目的**:** 查询事件状态而不阻塞

****参数**:**

- `id`: 事件资源 ID

****返回**:** 事件值 (0 = 未设置, 非零 = 已设置)

6.5.5 os_event_delete()

****位置**:** `Src/event/event\_delete.c`

****目的**:** 删除事件对象

****参数**:**

- `id`: 事件资源 ID

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证资源 ID
2. 清除资源结构
3. 以错误唤醒所有等待任务

6.5.6 os_event_set_from_isr()

****位置****: `Src/interrupt/event\_set\_from\_isr.c`

****目的****: ISR 安全的事件设置

****参数****:

- `id`: 事件资源 ID

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证资源 ID
2. 设置事件值
3. 递增挂起信号计数器
4. 触发调度请求

6.6 调用关系

6.6.1 事件等待流程

```
graph TD
    A[任务调用 os_event_wait] --> B{事件已设置?}
    B -->|是| C[立即返回]
    B -->|否| D[进入临界区]
    D --> E[调用 os_wait WAIT_EVENT]
    E --> F[设置 TCB wait_type]
    E --> G[添加到 wait_mask]
    E --> H[设置任务状态 WAIT]
    E --> I[触发调度]
    E --> J[任务阻塞]
    J --> K{事件设置或超时?}
    K -->|事件设置| L[调用 wake_task]
    K -->|超时| M[超时处理程序]
    L --> N[返回 WAIT_SIGNAL]
    M --> O[返回 WAIT_TIMEOUT]
```

6.6.2 事件设置流程

```
graph TD
    A[os_event_write] --> B[进入临界区]
    B --> C[设置 OS_RES value=1]
    C --> D{任务等待?}
    D -->|是| E[迭代 wait_mask]
    E --> F[为每个调用 wake_task]
    D -->|否| G[跳过]
```

```
F --> H[退出临界区]
G --> H
H --> I[返回成功]
```

6.6.3 ISR 事件设置流程

```
graph TD
    A[ISR 调用 os_event_set_from_isr] --> B[设置 OS_RES value=1]
    B --> C[递增 pending_signal]
    C --> D[调用 os_schedule_request]
    D --> E[触发 Timer0 中断]
    E --> F[调度器处理唤醒]
```

6.7 生命周期

6.7.1 事件生命周期

1. **初始化**：`os\_event\_init()` 创建事件对象
2. **等待**：任务调用 `os\_event\_wait()` 阻塞
3. **设置**：`os\_event\_write()` 或 ISR 设置事件标志
4. **唤醒**：所有等待任务被唤醒
5. **自动重置**：事件标志保持设置（未实现手动重置）
6. **删除**：`os\_event\_delete()` 移除事件

6.8 设计原则

6.8.1 轻量级同步

- 无数据传输，仅状态通知
- 标志的单字节值
- 基于位图的等待队列
- 最小内存开销

6.8.2 广播唤醒

- 设置事件唤醒所有等待任务
- 无单任务唤醒模式
- 适用于"事件发生"通知

6.8.3 ISR 安全

- 独立的 ISR 安全 API
- ISR 的挂起信号计数器
- 临界区保护
- 可从中断上下文安全调用

6.8.4 统一资源模型

- 与其他 IPC 使用相同的 `OS\_RESOURCE` 结构
- 通过 `os\_wait()` 统一等待机制
- 一致的 API 模式

6.9 约束

- 最多 8 个事件对象
- 无手动重置（未实现自动重置）
- 无 AND/OR 等待模式（仅单个事件）
- 仅广播唤醒（无选择性唤醒）
- 无事件数据负载
- 事件标志持续直到显式清除

6.10 使用模式

6.10.1 简单事件通知

```
// 任务 A 等待事件
os_event_wait(EVENT_ID, OS_WAIT_FOREVER);

// 任务 B 或 ISR 设置事件
os_event_write(EVENT_ID);
```

6.10.2 超时等待

```
// 带超时等待
result = os_event_wait(EVENT_ID, 100);
if (result == WAIT_TIMEOUT) {
    // 处理超时
}
```

6.10.3 ISR 通知

```
// 在 ISR 中
void external_interrupt_handler(void) {
    os_event_set_from_isr(EVENT_ID);
}
```

6.11 性能考虑

6.11.1 快速路径优化

- 事件已设置：立即返回而不阻塞
- 如果事件已设置则无上下文切换
- 仅标志修改的临界区

6.11.2 唤醒开销

- 事件设置时唤醒所有等待任务
- 每次唤醒需要上下文切换
- 如果许多任务等待可能导致惊群效应

6.11.3 内存效率

- 事件标志单字节
- 等待队列位图（2 字节）
- 无额外数据结构

6.12 与其他 IPC 的比较

6.12.1 vs 信号量

- 事件：二进制标志，广播唤醒
- 信号量：计数，每次发布单个唤醒

6.12.2 vs 消息队列

- 事件：无数据传输
- 消息队列：带队列的数据传输

6.12.3 vs 互斥锁

- 事件：无所有权，无优先级继承
- 互斥锁：所有权，优先级继承

7 HRTOS 消息队列设计

7.1 模块介绍

消息队列模块提供缓冲的任务间通信机制，用于在任务之间传递 1 字节消息。它实现具有生产者-消费者语义的循环缓冲区，支持阻塞和非阻塞模式。

7.2 主要职责

消息队列模块处理：

- 使用用户缓冲区初始化队列
- 消息发送（入队）
- 消息接收（出队）
- 队列清除
- ISR 安全消息发送
- 阻塞/非阻塞模式

7.3 主要文件

7.3.1 源文件

- `Src/msgq/msgq\_init.c`：队列初始化
- `Src/msgq/msgq\_send.c`：消息发送
- `Src/msgq/msgq\_recv.c`：消息接收
- `Src/msgq/msgq\_clear.c`：队列清除
- `Src/interrupt/msgq\_send\_from\_isr.c`：ISR 安全消息发送
- `Src/interrupt/msgq\_recv\_from\_isr.c`：ISR 安全消息接收
- `Src/interrupt/queue\_send\_from\_isr.c`：ISR 安全队列发送（遗留）

7.3.2 头文件

- `Inc/msgq.h`：消息队列 API 声明
- `Inc/config.h`：消息队列结构定义
- `Inc/hrtos\_internal.h`：内部队列变量

7.4 数据结构

7.4.1 os_msgq_t (消息队列结构)

位于 `Inc/config.h`:

```
typedef struct {  
    u8 *buf;           /* 缓冲区指针 */  
    u8 _size;          /* 缓冲区大小 */  
    u8 head;           /* 写指针 (生产者) */  
    u8 tail;           /* 读指针 (消费者) */  
    u8 count;          /* 当前消息计数 */  
} os_msgq_t;
```

7.4.2 队列数组

位于 `Src/wait/os\_wait.c`:

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

配置:

- `OS\_MSGQ\_MAX`: 4 (最大队列数)

7.5 核心函数

7.5.1 os_msgq_init()

****位置****: `Src/msgq/msgq\_init.c`

****目的****: 使用用户缓冲区初始化消息队列

****参数****:

- `q`: 队列结构指针
- `buf`: 用户缓冲区指针
- `\_size`: 缓冲区大小

****过程****:

1. 验证参数
2. 设置缓冲区指针
3. 设置缓冲区大小
4. 将 head 和 tail 初始化为 0
5. 将 count 初始化为 0

7.5.2 os_msgq_send()

****位置****: `Src/msgq/msgq\_send.c`

****目的****: 向队列发送消息

****参数****:

- `q`: 队列指针

- `obj`：资源 ID（用于等待）
- `\_data`：消息数据（1 字节）
- `nonblock`：0 = 阻塞，1 = 非阻塞
- **返回****：成功返回 1，失败返回 0
- **过程****：
 1. 检查队列是否已满
 2. 如果已满且阻塞：等待空间
 3. 如果已满且非阻塞：返回失败
 4. 在 head 位置写入数据到缓冲区
 5. 递增 head（带环绕）
 6. 递增 count
 7. 唤醒等待的接收者（如果有）

7.5.3 os_msgq_recv()

- **位置****：`Src/msgq/msgq\_recv.c`
- **目的****：从队列接收消息
- **参数****：
 - `q`：队列指针
 - `obj`：资源 ID（用于等待）
 - `\_data`：接收数据指针
 - `nonblock`：0 = 阻塞，1 = 非阻塞
- **返回****：成功返回 1，失败返回 0
- **过程****：
 1. 检查队列是否为空
 2. 如果为空且阻塞：等待数据
 3. 如果为空且非阻塞：返回失败
 4. 从 tail 位置读取缓冲区数据
 5. 递增 tail（带环绕）
 6. 递减 count
 7. 唤醒等待的发送者（如果有）

7.5.4 os_msgq_clear()

- **位置****：`Src/msgq/msgq\_clear.c`
- **目的****：清除队列中的所有消息
- **参数****：
 - `q`：队列指针

****过程**:**

1. 将 head 重置为 0
2. 将 tail 重置为 0
3. 将 count 重置为 0
4. 唤醒所有等待任务

7.5.5 os_msgq_send_from_isr()

****位置**:** `Src/interrupt/msgq\_send\_from\_isr.c`

****目的**:** ISR 安全消息发送

****参数**:**

- `q`: 队列指针
- `obj`: 资源 ID
- `\_data`: 消息数据

****返回**:** 成功返回 1, 失败返回 0

****过程**:**

1. 检查队列是否已满
2. 如果已满: 返回失败 (ISR 中不阻塞)
3. 写入数据到缓冲区
4. 更新 head 和 count
5. 触发调度请求

7.6 调用关系

7.6.1 发送流程 (阻塞)

```
graph TD
    A[os_msgq_send] --> B{队列已满?}
    B -->|否| C[写入数据到缓冲区]
    B -->|是| D{阻塞?}
    D -->|是| E[os_wait WAIT_MSG_SEND]
    D -->|否| F[返回失败]
    E --> G[任务阻塞]
    G --> H[空间可用]
    H --> I[调用 wake_task]
    I --> C
    C --> J[递增 head]
    J --> K[递增 count]
    K --> L{接收者等待?}
    L -->|是| M[wake_task 接收者]
    L -->|否| N[跳过]
    M --> O[返回成功]
    N --> O
```

7.6.2 接收流程 (阻塞)

```

graph TD
    A[os_msgq_recv] --> B{队列为空?}
    B -->|否| C[从缓冲区读取数据]
    B -->|是| D{阻塞?}
    D -->|是| E[os_wait WAIT_MSG_RECV]
    D -->|否| F[返回失败]
    E --> G[任务阻塞]
    G --> H[数据可用]
    H --> I[调用 wake_task]
    I --> C
    C --> J[递增 tail]
    J --> K[递减 count]
    K --> L{发送者等待?}
    L -->|是| M[wake_task 发送者]
    L -->|否| N[跳过]
    M --> O[返回成功]
    N --> O

```

7.6.3 ISR 发送流程

```

graph TD
    A[os_msgq_send_from_isr] --> B{队列已满?}
    B -->|否| C[写入数据]
    B -->|是| D[返回失败]
    C --> E[更新 head 和 count]
    E --> F[os_schedule_request]
    F --> G[触发 Timer0]
    G --> H[调度器处理]

```

7.7 生命周期

7.7.1 队列生命周期

1. **初始化**：`os\_msgq\_init()` 绑定缓冲区和结构
2. **发送/接收**：任务发送和接收消息
3. **满状态**：发送者阻塞（如果非阻塞则错误）
4. **空状态**：接收者阻塞（如果非阻塞则错误）
5. **清除**：`os\_msgq\_clear()` 清空队列

7.8 设计原则

7.8.1 循环缓冲区

- Head 指针用于写入（生产者）
- Tail 指针用于读取（消费者）
- 在缓冲区大小处环绕
- Count 跟踪当前消息

7.8.2 用户提供的缓冲区

- 无动态分配

- 用户管理缓冲区内存
- 队列结构仅管理指针
- 灵活的缓冲区大小

7.8.3 阻塞/非阻塞模式

- 阻塞：等待直到操作成功
- 非阻塞：立即返回状态
- 由 `nonblock` 参数控制

7.8.4 ISR 安全

- 独立的 ISR 安全发送 API
- ISR 上下文中不阻塞
- 调度请求用于唤醒
- ISR 安全接收也可用

7.8.5 统一等待

- 使用 `os\_wait()` 进行阻塞操作
- `WAIT\_MSG\_SEND` 用于发送者等待
- `WAIT\_MSG\_RECV` 用于接收者等待
- 与其他 IPC 机制一致

7.9 约束

- 最多 4 个队列对象
- 仅 1 字节消息大小
- 用户必须提供缓冲区
- 无基于优先级的消息排序
- 仅 FIFO 排序
- ISR 不能在满队列上阻塞
- 无消息过滤或选择

7.10 使用模式

7.10.1 基本生产者-消费者

```
// 生产者任务
void producer_task(void) {
    while (1) {
        u8 data = produce_data();
        os_msgq_send(&g_queue, QUEUE_RES_ID, data, 0); // 阻塞
        os_delay(10);
    }
}
```

```

    }
}

// 消费者任务
void consumer_task(void) {
    u8 data;
    while (1) {
        os_msgq_rcv(&g_queue, QUEUE_RES_ID, &data, 0); // 阻塞
        process_data(data);
    }
}

```

7.10.2 非阻塞模式

```

// 非阻塞发送
if (os_msgq_send(&g_queue, QUEUE_RES_ID, data, 1) == 0) {
    // 队列已满，处理错误
}

// 非阻塞接收
if (os_msgq_rcv(&g_queue, QUEUE_RES_ID, &data, 1) == 0) {
    // 队列为空，处理错误
}

```

7.10.3 ISR 到任务

```

// ISR
void uart_rx_isr(void) {
    u8 data = SBUF;
    os_msgq_send_from_isr(&g_queue, QUEUE_RES_ID, data);
}

// 任务
void uart_task(void) {
    u8 data;
    while (1) {
        os_msgq_rcv(&g_queue, QUEUE_RES_ID, &data, 0);
        process_uart_data(data);
    }
}

```

7.11 性能考虑

7.11.1 缓冲区大小权衡

- 较大缓冲区：较少阻塞，更多内存
- 较小缓冲区：较多阻塞，较少内存
- 最佳大小取决于生产/消费速率

7.11.2 上下文切换开销

- 阻塞操作导致上下文切换
- 非阻塞避免上下文切换但需要轮询
- ISR 发送在唤醒时触发调度

7.11.3 内存效率

- 队列结构：5 字节
- 缓冲区：用户定义大小
- 每个队列无额外内核内存

7.12 与其他 IPC 的比较

7.12.1 vs 事件

- 消息队列：带缓冲的数据传输
- 事件：无数据，仅状态通知

7.12.2 vs 邮箱

- 消息队列：多条消息，队列语义
- 邮箱：单条消息，发送时覆盖

7.12.3 vs 信号量

- 消息队列：带队列的数据传输
- 信号量：仅计数同步

Mutex Design

8 HRTOS 互斥锁设计

8.1 模块介绍

互斥锁模块提供同步机制，用于保护共享资源免受并发访问。它实现优先级继承以防止优先级反转，确保多任务系统中的实时性能。

8.2 主要职责

互斥锁模块处理：

- 互斥锁初始化
- 互斥锁获取（加锁）
- 互斥锁释放（解锁）
- 优先级继承
- 所有者跟踪
- 等待队列管理

8.3 主要文件

8.3.1 源文件

- `Src/mutex/mutex\_init.c`：互斥锁初始化
- `Src/mutex/mutex\_lock.c`：具有优先级继承的互斥锁加锁
- `Src/mutex/mutex\_unlock.c`：具有优先级恢复的互斥锁解锁

8.3.2 头文件

- `Inc/mutex.h`：互斥锁 API 声明
- `Inc/config.h`：互斥锁资源结构
- `Inc/hrtos\_internal.h`：内部互斥锁变量

8.4 数据结构

8.4.1 OS_RESOURCE（统一 IPC）

互斥锁使用统一资源结构：

```
typedef struct {  
    u8 value;           /* 互斥锁不使用 */  
    u8 owner;           /* 当前所有者任务 ID */  
    u8 wait_cnt;        /* 等待任务计数 */  
    u16 wait_mask;      /* 位图等待队列 */  
    u8 pending_signal;  /* 互斥锁不使用 */  
} OS_RESOURCE;
```

对于互斥锁：

- `owner`：当前持有锁的任务 ID
- `wait\_mask`：等待此互斥锁的任务位图
- `wait\_cnt`：等待任务计数

8.5 核心函数

8.5.1 os_mutex_init()

****位置****：`Src/mutex/mutex\_init.c`

****目的****：初始化互斥锁对象

****参数****：

- `mid`：互斥锁资源 ID (0-7)

****返回****：成功返回 1，失败返回 -1

****过程****：

1. 验证资源 ID
2. 通过 `os\_res\_init()` 初始化资源结构
3. 将所有者设置为 `OS\_INVALID\_ID`
4. 清除等待队列

8.5.2 os_mutex_lock()

****位置****：`Src/mutex/mutex\_lock.c`

****目的****：获取具有优先级继承的互斥锁

****参数****：

- `mid`：互斥锁资源 ID

****返回****：

- 1：成功获取锁
- 0：进入等待队列
- -1：参数错误或递归锁尝试

****过程****：

```
char os_mutex_lock(char mid)  
{  
    OS_RESOURCE *m;
```

```

char tid;

if (mid < 0 || mid >= OS_RESOURCE_MAX) return -1;

tid = OS_CURRENT_TASK;
m = &OS_RES[mid];
EA = 0;

/* 无递归锁支持 */
if (m->owner == tid) {
    EA = 1;
    return -1;
}

/* 互斥锁空闲，直接获取 */
if (m->owner == OS_INVALID_ID) {
    m->owner = tid;
    OS_TASK[tid].base_prio = (OS_PROCESS_OK[tid] & 14) >> 1;
    OS_TASK[tid].cur_prio = OS_TASK[tid].base_prio;
    EA = 1;
    return 1;
}

/* 优先级继承 */
OS_TASK[tid].base_prio = (OS_PROCESS_OK[tid] & 14) >> 1;
OS_TASK[tid].cur_prio = OS_TASK[tid].base_prio;

if (OS_TASK[tid].cur_prio > OS_TASK[m->owner].cur_prio) {
    OS_TASK[m->owner].cur_prio = OS_TASK[tid].cur_prio;
    OS_PROCESS_OK[m->owner] &= 0xF1;
    OS_PROCESS_OK[m->owner] |= ((OS_TASK[tid].cur_prio & 0x07) <<
1);
}

EA = 1;

/* 进入等待状态 */
os_wait(WAIT_MUTEX, mid, 0);
return 1;
}

```

8.5.3 os_mutex_unlock()

****位置**:** `Src/mutex/mutex\_unlock.c`

****目的**:** 释放互斥锁并恢复优先级

****参数**:**

- `mid`: 互斥锁资源 ID

****返回**:** 成功返回 1，失败返回 -1

****过程**:**

1. 验证资源 ID
2. 检查当前任务是否为所有者
3. 将所有者的优先级恢复到基本优先级
4. 清除所有者字段

5. 唤醒最高优先级的等待任务
6. 设置新所有者（如果有任务等待）

8.6 调用关系

8.6.1 互斥锁加锁流程

```
graph TD
    A[os_mutex_lock] --> B{互斥锁空闲?}
    B -->|是| C[设置所有者]
    B -->|否| D{当前任务所有者?}
    D -->|是| E[返回错误 - 递归]
    D -->|否| F[优先级继承]
    F --> G[提升所有者优先级]
    G --> H[os_wait WAIT_MUTEX]
    H --> I[任务阻塞]
    I --> J{互斥锁释放?}
    J -->|是| K[调用 wake_task]
    K --> L[任务获取锁]
    L --> M[返回成功]
    C --> N[返回成功]
```

8.6.2 互斥锁解锁流程

```
graph TD
    A[os_mutex_unlock] --> B{当前任务所有者?}
    B -->|否| C[返回错误]
    B -->|是| D[恢复基本优先级]
    D --> E[清除所有者字段]
    E --> F{任务等待?}
    F -->|是| G[选择最高优先级]
    F -->|否| H[返回成功]
    G --> I[wake_task 选定任务]
    I --> J[设置新所有者]
    J --> K[返回成功]
```

8.6.3 优先级继承流程

```
graph TD
    A[高优先级任务尝试加锁] --> B{互斥锁被低优先级持有?}
    B -->|否| C[正常加锁]
    B -->|是| D[提升所有者优先级]
    D --> E[更新所有者 cur_prio]
    E --> F[更新 OS_PROCESS_OK]
    F --> G[所有者抢占中优先级任务]
    G --> H[所有者释放锁]
    H --> I[所有者优先级恢复]
    I --> J[高优先级任务获取锁]
```

8.7 生命周期

8.7.1 互斥锁生命周期

1. **初始化**：`os\_mutex\_init()` 创建互斥锁
2. **获取**：任务调用 `os\_mutex\_lock()`

3. ****所有权****: 所有者字段设置为任务 ID
4. ****竞争****: 其他任务在队列中等待
5. ****优先级继承****: 如需要则提升所有者优先级
6. ****释放****: 所有者调用 ``os_mutex_unlock()``
7. ****优先级恢复****: 所有者优先级恢复
8. ****转移****: 下一个等待任务获取锁

8.8 设计原则

8.8.1 互斥

- 一次只能有一个任务持有互斥锁
- 所有者字段跟踪当前持有者
- 无递归锁定（返回错误）

8.8.2 优先级继承

- 防止优先级反转
- 高优先级等待任务提升所有者优先级
- 解锁时恢复所有者优先级
- 确保有界阻塞时间

8.8.3 无递归锁

- 递归锁尝试返回错误
- 防止递归获取导致的死锁
- 更简单的实现

8.8.4 统一资源模型

- 使用相同的 ``OS_RESOURCE`` 结构
- 通过 ``os_wait()`` 统一等待
- 与其他 IPC 一致

8.9 约束

- 最多 8 个互斥锁对象
- 无递归锁定支持
- 仅优先级继承（无天花板协议）
- 必须由拥有任务解锁
- 不能在 ISR 上下文中使用

- 锁获取无超时

8.10 优先级继承算法

8.10.1 何时提升

在以下情况下提升优先级：

1. 互斥锁已被持有
2. 等待任务的优先级高于所有者
3. 等待任务不是所有者（递归检查）

8.10.2 提升实现

```
if (OS_TASK[tid].cur_prio > OS_TASK[m->owner].cur_prio) {
    OS_TASK[m->owner].cur_prio = OS_TASK[tid].cur_prio;
    OS_PROCESS_OK[m->owner] &= 0xF1;
    OS_PROCESS_OK[m->owner] |= ((OS_TASK[tid].cur_prio & 0x07) << 1);
}
```

8.10.3 优先级恢复

解锁时：

1. 将所有者的 `cur\_prio` 恢复到 `base\_prio`
2. 用基本优先级更新 `OS\_PROCESS\_OK`
3. 所有者返回到原始优先级级别

8.11 使用模式

8.11.1 基本互斥锁保护

```
// 共享资源
u8 shared_counter = 0;

// 任务 A
void task_a(void) {
    while (1) {
        os_mutex_lock(MUTEX_ID);
        shared_counter++;
        os_mutex_unlock(MUTEX_ID);
        os_delay(10);
    }
}

// 任务 B
void task_b(void) {
    while (1) {
        os_mutex_lock(MUTEX_ID);
        shared_counter--;
        os_mutex_unlock(MUTEX_ID);
        os_delay(10);
    }
}
```

```
}
```

8.11.2 优先级继承场景

```
// 低优先级任务（优先级 2）
void low_task(void) {
    while (1) {
        os_mutex_lock(MUTEX_ID);
        // 持有锁一段时间
        process_shared_resource();
        os_mutex_unlock(MUTEX_ID);
        os_delay(100);
    }
}

// 高优先级任务（优先级 6）
void high_task(void) {
    while (1) {
        os_mutex_lock(MUTEX_ID);
        // 将 low_task 优先级提升到 6
        critical_operation();
        os_mutex_unlock(MUTEX_ID);
        os_delay(10);
    }
}

// 中优先级任务（优先级 4）
void medium_task(void) {
    while (1) {
        // 将被提升的 low_task 抢占
        background_work();
        os_delay(50);
    }
}
```

8.12 性能考虑

8.12.1 锁获取时间

- 快速路径：互斥锁空闲（无阻塞）
- 慢速路径：互斥锁被持有（阻塞 + 优先级提升）
- 优先级提升增加最小开销

8.12.2 上下文切换开销

- 锁阻塞导致上下文切换
- 优先级提升可能导致立即抢占
- 解锁导致唤醒和上下文切换

8.12.3 内存效率

- 互斥锁结构：5 字节（OS_RESOURCE 的一部分）
- 每个互斥锁无额外内存

- 等待队列的位图

8.13 与其他 IPC 的比较

8.13.1 vs 信号量

- 互斥锁：所有权，优先级继承
- 信号量：计数，无所有权

8.13.2 vs 二进制信号量

- 互斥锁：所有者必须解锁
- 二进制信号量：任何任务都可以释放

8.13.3 vs 自旋锁

- 互斥锁：阻塞，上下文切换
- 自旋锁：忙等待（HRTOS 中未实现）

8.14 常见陷阱

8.14.1 死锁预防

- 始终以一致的顺序获取锁
- 保持临界区简短
- 持有锁时不要调用阻塞 API
- 不要在 ISR 中使用互斥锁

8.14.2 优先级继承限制

- 仅防止无界优先级反转
- 不防止死锁
- 可能出现优先级提升链
- 未实现优先级天花板协议

Semaphore Design

9 HRTOS 信号量设计

9.1 模块介绍

信号量模块提供计数同步机制，用于资源管理和任务协调。它实现经典的 P（等待/获取）和 V（释放/发布）操作，支持资源计数和事件信号。

9.2 主要职责

信号量模块处理：

- 带初始计数的信号量初始化
- P 操作（等待/获取）
- V 操作（释放/发布）
- 资源计数
- 等待队列管理
- ISR 安全释放操作

9.3 主要文件

9.3.1 源文件

- `Src/semaphore/sem\_init.c`：信号量初始化
- `Src/semaphore/sem\_wait.c`：P 操作（等待）
- `Src/semaphore/sem\_post.c`：V 操作（释放）
- `Src/interrupt/semaphore\_post\_from\_isr.c`：ISR 安全释放

9.3.2 头文件

- `Inc/semaphore.h`：信号量 API 声明
- `Inc/config.h`：信号量资源结构
- `Inc/hrtos\_internal.h`：内部信号量变量

9.4 数据结构

9.4.1 OS_RESOURCE (统一 IPC)

信号量使用统一资源结构：

```
typedef struct {  
    u8 value;           /* 信号量计数 */  
    u8 owner;           /* 信号量不使用 */  
    u8 wait_cnt;        /* 等待任务计数 */  
    u16 wait_mask;      /* 位图等待队列 */  
    u8 pending_signal;  /* ISR 信号计数 */  
} OS_RESOURCE;
```

对于信号量：

- `value`：信号量计数（0 = 无资源，>0 = 可用资源）
- `wait\_mask`：等待此信号量的任务位图
- `wait\_cnt`：等待任务计数

9.5 核心函数

9.5.1 os_sem_init()

****位置****：`Src/semaphore/sem\_init.c`

****目的****：使用初始计数初始化信号量

****参数****：

- `sid`：信号量资源 ID（0-7）
- `init\_val`：初始计数值

****返回****：成功返回 1，失败返回 -1

****过程****：

1. 验证资源 ID
2. 通过 `os\_res\_init()` 初始化资源结构
3. 设置初始值
4. 清除等待队列

9.5.2 os_sem_wait()

****位置****：`Src/semaphore/sem\_wait.c`

****目的****：P 操作 - 获取信号量资源

****参数****：

- `sid`：信号量资源 ID

****返回****：

- `WAIT\_SIGNAL`：成功获取
- 任务 ID：进入等待（唤醒后返回）
- 0：参数错误

****过程****：

```

u8 os_sem_wait(u8 sid)
{
    OS_RESOURCE *s;
    u8 tid;

    if (sid >= OS_RESOURCE_MAX) return 0;

    EA = 0;
    s = &OS_RES[sid];

    if(s->value == 0) {
        EA = 1;
        tid = os_wait(WAIT_SEM, sid, 0);
    }

    /* 快速路径：信号量可用（无阻塞） */
    if(s->value > 0) {
        s->value--;          /* 消耗一个资源 */
        EA = 1;
        return WAIT_SIGNAL;  /* 成功 */
    }

    EA = 1;
    return tid;
}

```

****设计说明**：**

- 快速路径优化：如果 value > 0，立即获取而不阻塞
- 慢速路径：如果 value == 0，进入等待状态
- 双通道模型：value 计数 + wait_mask 队列

9.5.3 os_sem_post()

****位置**：** `Src/semaphore/sem\_post.c`

****目的**：** V 操作 - 释放信号量资源

****参数**：**

- `sid`：信号量资源 ID

****返回**：** 成功返回 1，失败返回 -1

****过程**：**

1. 验证资源 ID
2. 进入临界区
3. 检查是否有任务等待
4. 如果有等待：唤醒最高优先级任务
5. 如果无等待：递增 value
6. 退出临界区

9.5.4 os_semaphore_post_from_isr()

****位置**：** `Src/interrupt/semaphore\_post\_from\_isr.c`

****目的****: ISR 安全 V 操作

****参数****:

- `obj`: 信号量资源 ID

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证资源 ID
2. 递增 value 或设置挂起信号
3. 触发调度请求
4. 立即返回 (ISR 中不阻塞)

9.6 调用关系

9.6.1 信号量等待流程

```
graph TD
    A[os_sem_wait] --> B{Value > 0?}
    B -->|是| C[快速路径: 递减 Value]
    B -->|否| D[慢速路径: os_wait]
    C --> E[返回 WAIT_SIGNAL]
    D --> F[设置 TCB wait_type=WAIT_SEM]
    F --> G[添加到 wait_mask]
    G --> H[设置任务状态 WAIT]
    H --> I[触发调度]
    I --> J[任务阻塞]
    J --> K{sem_post 调用?}
    K -->|是| L[调用 wake task]
    K -->|超时| M[超时处理程序]
    L --> N[返回 WAIT_SIGNAL]
    M --> O[返回 WAIT_TIMEOUT]
```

9.6.2 信号量释放流程

```
graph TD
    A[os_sem_post] --> B[进入临界区]
    B --> C{任务等待?}
    C -->|是| D[选择最高优先级]
    C -->|否| E[递增 Value]
    D --> F[wake task 选定任务]
    F --> G[退出临界区]
    E --> G
    G --> H[返回成功]
```

9.6.3 ISR 释放流程

```
graph TD
    A[os_semaphore_post_from_isr] --> B[递增 Value 或 pending_signal]
    B --> C[os schedule_request]
    C --> D[触发 Timer0 中断]
    D --> E[调度器处理唤醒]
```

9.7 生命周期

9.7.1 信号量生命周期

1. ****初始化****: ``os_sem_init()`` 设置初始计数
2. ****获取****: 任务调用 ``os_sem_wait()`` (P 操作)
3. ****计数****: 每次成功等待时 value 递减
4. ****阻塞****: value 达到 0 时任务阻塞
5. ****释放****: 任务调用 ``os_sem_post()`` (V 操作)
6. ****唤醒****: 等待任务被唤醒
7. ****重用****: 信号量可以重复使用

9.8 设计原则

9.8.1 计数语义

- Value 表示可用资源计数
- P 操作递减 (获取)
- V 操作递增 (释放)
- Value 可以 >1 (多个资源)

9.8.2 快速路径优化

- 进入临界区前检查 value
- 如果 value > 0, 立即获取
- 避免上下文切换开销
- 对性能至关重要

9.8.3 双通道模型

- 通道 1: ``value`` 计数器 (快速路径)
- 通道 2: ``wait_mask`` 队列 (慢速路径)
- 互斥操作
- 防止竞争条件

9.8.4 ISR 安全

- 独立的 ISR 安全释放 API
- ISR 上下文中不阻塞
- ISR 的挂起信号计数器
- 唤醒的调度请求

9.8.5 统一资源模型

- 使用相同的 ``OS_RESOURCE`` 结构

- 通过 `os\_wait()` 统一等待
- 与其他 IPC 一致

9.9 约束

- 最多 8 个信号量对象
- Value 限制为 8 位 (0-255)
- 等待无超时 (无限等待)
- 无优先级继承
- 无所有权跟踪
- 不能用于互斥 (改用互斥锁)

9.10 使用模式

9.10.1 资源计数

```
// 初始化 3 个资源
os_sem_init(SEM_ID, 3);

// 任务 A
void task_a(void) {
    while (1) {
        os_sem_wait(SEM_ID);
        use_resource();
        os_sem_post(SEM_ID);
        os_delay(10);
    }
}

// 任务 B
void task_b(void) {
    while (1) {
        os_sem_wait(SEM_ID);
        use_resource();
        os_sem_post(SEM_ID);
        os_delay(10);
    }
}
```

9.10.2 二进制信号量 (信号)

```
// 初始化为二进制信号量 (0 或 1)
os_sem_init(SEM_ID, 0);

// 任务 A - 生产者
void producer_task(void) {
    while (1) {
        produce_data();
        os_sem_post(SEM_ID); // 信号
        os_delay(10);
    }
}
```

```
// 任务 B - 消费者
void consumer_task(void) {
    while (1) {
        os_sem_wait(SEM_ID); // 等待信号
        consume_data();
    }
}
```

9.10.3 ISR 信号

```
// ISR
void external_interrupt_handler(void) {
    os_semaphore_post_from_isr(SEM_ID);
}

// 任务
void interrupt_task(void) {
    while (1) {
        os_sem_wait(SEM_ID);
        handle_interrupt_event();
    }
}
```

9.11 性能考虑

9.11.1 快速路径优势

- 资源可用时无上下文切换
- 无竞争情况的最小开销
- 对高频操作至关重要

9.11.2 阻塞开销

- value == 0 时上下文切换
- 释放时的唤醒开销
- 基于优先级的唤醒选择

9.11.3 内存效率

- 信号量结构：5 字节（OS_RESOURCE 的一部分）
- Value：1 字节
- 等待队列：2 字节（位图）

9.12 与其他 IPC 的比较

9.12.1 vs 互斥锁

- 信号量：计数，无所有权
- 互斥锁：二进制，所有权，优先级继承

9.12.2 vs 事件

- 信号量：计数，资源跟踪
- 事件：二进制标志，无计数

9.12.3 vs 消息队列

- 信号量：计数同步
- 消息队列：带缓冲的数据传输

9.13 常见陷阱

9.13.1 资源泄漏

- 成功等待后始终释放
- 使用错误处理确保释放
- 不要在错误路径中忘记释放

9.13.2 优先级反转

- 信号量没有优先级继承
- 高优先级任务可能被低优先级任务阻塞
- 如果优先级反转是问题，使用互斥锁

9.13.3 过度使用

- 不要使用信号量进行互斥
- 使用互斥锁进行独占访问
- 使用信号量进行资源计数或信号

Timer System

10 HRTOS 定时器系统

10.1 模块介绍

定时器系统为 HRTOS 提供基于时间的服务，包括系统时钟生成、任务延时、时间转换工具和运行时间跟踪。它基于 8051 微控制器的 Timer0，是内核中所有时间相关操作的基础。

10.2 主要职责

定时器系统处理：

- 通过 Timer0 生成系统时钟
- 时间片管理
- 任务延时实现
- 时钟到时间单位的转换
- 系统运行时间跟踪
- 定时器配置

10.3 主要文件

10.3.1 源文件

- `Src/time/tick\_config.c`：定时器配置
- `Src/time/delay.c`：基于时钟的任务延时
- `Src/time/delay\_ms.c`：忙等待毫秒延时
- `Src/time/ms\_to\_tick.c`：毫秒到时钟周期转换
- `Src/time/sec\_to\_tick.c`：秒到时钟周期转换
- `Src/time/tick\_get.c`：获取当前时钟周期计数
- `Src/time/uptime\_ms.c`：获取系统运行时间（毫秒）

10.3.2 头文件

- `Inc/time.h`：定时器 API 声明
- `Inc/config.h`：定时器配置常量

- `Inc/hrtos\_internal.h`: 内部定时器变量

10.4 数据结构

10.4.1 定时器配置常量

位于 `Inc/config.h`:

```
#define OS_TIME_ONCE_DEFAULT 5          // 默认时间片计数
#define OS_TIME_T0 10000                // 单个时间片长度 (Timer0 计数)
```

10.4.2 内部定时器变量

位于 `Src/kernel/os\_core.c`:

```
volatile unsigned char idata OS_TIME_XY;          // 当前时间片计数器
volatile unsigned char idata OS_T0_TH0;           // Timer0 高字节
volatile unsigned char idata OS_T0_TL0;           // Timer0 低字节
volatile unsigned char xdata OS_TIME_ONCE;        // 时间片计数
volatile unsigned char xdata OS_TIME_ONCE_BACKUP; // 模式切换备份
```

10.4.3 时钟周期计数器

位于 `Inc/hrtos\_internal.h`:

```
extern unsigned int xdata OS_TICK_COUNT_L;         // 时钟周期计数低 16 位
extern unsigned int xdata OS_TICK_COUNT_H;         // 时钟周期计数高 16 位
```

10.4.4 快速任务延时计数器

```
extern unsigned int xdata OS_WAIT_DI2[OS_FAST_TASK_MAX]; // 快速任务延时
```

10.5 核心函数

10.5.1 os_tick_config()

****位置****: `Src/time/tick\_config.c`

****目的****: 配置系统定时器参数

****参数****:

- `slice\_count`: 每个调度周期的时间片数

- `tick\_value`: 每个时钟周期的 Timer0 计数值

****返回****: 成功返回 1, 失败返回 -1

****过程****:

1. 验证参数
2. 将 `OS\_TIME\_ONCE` 设置为 slice_count
3. 将 Timer0 重载值设置为 tick_value
4. 重置当前时间片计数器

10.5.2 os_delay()

****位置****: `Src/time/delay.c`

****目的****: 延迟当前任务指定的时钟周期数

****参数****:

- `tick`: 延迟时长（时钟周期）

****过程****:

```
void os_delay(u16 tick)
{
    if(tick != 0) {
        if(OS_CURRENT_TASK < OS_PROCESS_MAX+2) {
            os_wait(WAIT_DELAY, OS_INVALID_ID, tick);
        }
    }
}
```

****设计说明****:

- 使用统一等待机制
- 类型: `WAIT\_DELAY`
- 无资源对象（使用 `OS\_INVALID\_ID`）
- 任务阻塞直到时钟周期到期

10.5.3 os_delay_ms()

****位置****: `Src/time/delay\_ms.c`

****目的****: 短硬件定时的忙等待延迟

****参数****:

- `ms`: 延迟（毫秒）

****特性****:

- 忙等待（不让出 CPU）
- 仅用于短硬件定时
- 不触发调度
- 需要精确定时而不进行任务切换时使用

10.5.4 os_ms_to_tick()

****位置****: `Src/time/ms\_to\_tick.c`

****目的****: 将毫秒转换为时钟周期

****参数****:

- `ms`: 毫秒

****返回****: 等效的时钟周期计数

****计算****: 基于 `OS\_TIME\_T0` 和 CPU 时钟频率

10.5.5 os_sec_to_tick()

****位置****: `Src/time/sec\_to\_tick.c`

****目的****: 将秒转换为时钟周期

****参数****:

- `sec`: 秒

****返回****: 等效的时钟周期计数

10.5.6 os_tick_get()

****位置****: `Src/time/tick\_get.c`

****目的****: 获取当前系统时钟周期计数

****参数****:

- `high`: 存储高 16 位的指针

- `low`: 存储低 16 位的指针

****过程****:

1. 进入临界区
2. 读取 `OS\_TICK\_COUNT\_L` 和 `OS\_TICK\_COUNT\_H`
3. 退出临界区
4. 通过指针返回值

10.5.7 os_uptime_ms()

****位置****: `Src/time/uptime\_ms.c`

****目的****: 获取系统运行时间（毫秒）

****返回****: 运行时间作为 32 位毫秒计数

****过程****:

1. 读取时钟周期计数器
2. 基于时钟周期持续时间转换为毫秒
3. 返回 32 位值

10.6 调用关系

10.6.1 定时器中断流程

```
graph TD
    A[Timer0 溢出] --> B[ISR 入口]
    B --> C[递减 OS_TIME_XY]
    C --> D{OS_TIME_XY == 0?}
    D -->|是| E[设置 OS_SCHED_REASON=1]
    D -->|否| F[继续]
    E --> G[重载 Timer0]
    F --> G
    G --> H[递增 OS_TICK_COUNT]
```

```

H --> I{快速任务延时?}
I -->|是| J[递减 OS_WAIT_DI2]
I -->|否| K[跳过]
J --> L{延时到期?}
L -->|是| M[唤醒快速任务]
L -->|否| N[继续]
M --> N
N --> O{标准任务延时?}
O -->|是| P[递减 wait_tick]
O -->|否| Q[跳过]
P --> R{延时到期?}
R -->|是| S[调用 wake_task]
R -->|否| T[继续]
S --> T
T --> U[ISR 退出]
U --> V{需要调度?}
V -->|是| W[上下文切换]
V -->|否| X[返回任务]

```

10.6.2 任务延时流程

```

graph TD
    A[任务调用 os_delay] --> B[os_wait WAIT_DELAY]
    B --> C[设置 TCB wait_tick]
    C --> D[设置任务状态 WAIT]
    D --> E[触发调度]
    E --> F[任务阻塞]
    F --> G[定时器中断]
    G --> H[递减 wait_tick]
    H --> I{wait_tick == 0?}
    I -->|是| J[调用 wake_task]
    I -->|否| K[继续阻塞]
    J --> L[任务就绪]
    L --> M[调度器选择任务]
    M --> N[任务恢复]

```

10.7 生命周期

10.7.1 定时器初始化

1. 系统启动
2. `os\_idle\_task()` 初始化定时器变量
3. 用户调用 `os\_tick\_config()` (可选)
4. Timer0 中断启用
5. 定时器开始生成时钟周期

10.7.2 时钟周期生成周期

1. Timer0 计数到溢出
2. Timer0 ISR 触发
3. 时间片计数器递减
4. 时钟周期计数器递增

5. 检查并递减任务延时
6. 如需要则触发调度
7. 重载 Timer0
8. ISR 退出

10.7.3 任务延时生命周期

1. 任务调用 ``os_delay(tick)``
2. ``wait_tick`` 设置为 tick 值
3. 任务进入 WAIT 状态
4. 每个时钟周期中断递减 ``wait_tick``
5. 当 ``wait_tick`` 达到 0 时，任务唤醒
6. 任务返回到 READY 状态
7. 调度器选择任务运行

10.8 设计原则

10.8.1 基于 Timer0

- 使用 8051 Timer0 生成时钟周期
- 可配置的时钟周期持续时间
- 硬件中断驱动
- 最小 CPU 开销

10.8.2 时间片调度

- ``OS_TIME_ONCE``: 每个调度周期的时间片数
- ``OS_TIME_XY``: 当前时间片计数器
- 当计数器达到 0 时触发调度
- 支持时间片和纯优先级模式

10.8.3 统一等待

- 延时使用相同的 ``os_wait()`` 机制
- 延时使用 ``WAIT_DELAY`` 类型
- 与 IPC 等待一致
- 简化实现

10.8.4 快速任务独立定时

- 快速任务使用 ``OS_WAIT_DI2`` 数组
- 独立于标准任务延时

- 快速任务的最小开销
- 快速任务无 TCB wait_tick

10.8.5 32 位时钟周期计数器

- 32 位总时钟周期计数 (16+16 位)
- 支持长运行时间
- 在合理时间内防止溢出
- 通过临界区原子访问

10.9 约束

- 所有定时使用单个定时器 (Timer0)
- 配置后时钟周期持续时间固定
- 最大延时受限于 16 位 wait_tick
- 无高分辨率定时器
- 无定时器回调机制
- 忙等待延迟仅用于短持续时间

10.10 定时器配置

10.10.1 默认配置

```
#define OS_TIME_ONCE_DEFAULT 5      // 每周期 5 个时间片
#define OS_TIME_T0 10000           // Timer0 计数值
```

10.10.2 自定义配置

```
// 配置 10ms 时钟周期，每周期 10 个时间片
os_tick_config(10, 10000);
```

10.10.3 时钟周期持续时间计算

时钟周期持续时间取决于：

- CPU 时钟频率
- Timer0 预分频器 (如果有)
- Timer0 重载值

12MHz CPU 示例：

- Timer0 以 1MHz 计数 (12/12)
- 重载值 10000 = 10ms 时钟周期
- 10 个时间片 = 100ms 调度周期

10.11 使用模式

10.11.1 基本任务延时

```
void periodic_task(void) {
    while (1) {
        do_work();
        os_delay(100); // 延迟 100 个时钟周期
    }
}
```

10.11.2 毫秒转换

```
void task_with_ms_delay(void) {
    u16 ticks = os_ms_to_tick(500); // 500ms
    while (1) {
        do_work();
        os_delay(ticks);
    }
}
```

10.11.3 运行时间跟踪

```
void monitor_task(void) {
    u32 uptime;
    while (1) {
        uptime = os_uptime_ms();
        display_uptime(uptime);
        os_delay(1000);
    }
}
```

10.11.4 硬件忙等待

```
void init hardware(void) {
    // 硬件稳定的短延迟
    os_delay_ms(10);
}
```

10.12 性能考虑

10.12.1 中断开销

- 定时器中断频率决定开销
- 更高频率 = 更精确的定时，更多开销
- 在精度和 CPU 负载之间平衡

10.12.2 延迟精度

- 时钟周期粒度决定最小延迟
- 不能延迟少于一个时钟周期
- 使用 `os\_delay\_ms()` 获得亚时钟周期精度（忙等待）

10.12.3 上下文切换开销

- 任务延时导致上下文切换
- 唤醒需要调度
- 多个任务同时到期导致多次唤醒

10.13 与其他 RTOS 定时器的比较

10.13.1 vs 硬件定时器

- HRTOS：单个系统定时器，软件延时
- 硬件定时器：多个独立定时器，硬件事件

10.13.2 vs 虚拟定时器

- HRTOS：无虚拟定时器回调
- 虚拟定时器：每个任务的定时器回调

10.13.3 vs 高分辨率定时器

- HRTOS：基于时钟周期，分辨率有限
- 高分辨率：微秒/纳秒精度

Interrupt Design

11 HRTOS 中断设计

11.1 模块介绍

中断模块为 HRTOS 提供中断管理服务，包括临界区保护、ISR 上下文检测和进程间通信的 ISR 安全 API。它确保中断处理程序和内核调度器之间的安全交互。

11.2 主要职责

中断模块处理：

- 临界区管理（进入/退出）
- ISR 上下文检测
- ISR 安全 IPC 操作
- 中断 ID 识别
- 来自 ISR 的调度请求
- 中断嵌套支持

11.3 主要文件

11.3.1 源文件

- `Src/interrupt/enter\_critical.c`：进入临界区
- `Src/interrupt/exit\_critical.c`：退出临界区
- `Src/interrupt/in\_isr.c`：检查 ISR 上下文
- `Src/interrupt/interrupt\_id.c`：获取中断 ID
- `Src/interrupt/schedule\_request.c`：从 ISR 请求调度
- `Src/interrupt/event\_set\_from\_isr.c`：ISR 安全的事件设置
- `Src/interrupt/msgq\_send\_from\_isr.c`：ISR 安全的消息发送
- `Src/interrupt/msgq\_recv\_from\_isr.c`：ISR 安全的消息接收
- `Src/interrupt/queue\_send\_from\_isr.c`：ISR 安全的队列发送（遗留）
- `Src/interrupt/semaphore\_post\_from\_isr.c`：ISR 安全的信号量释放

11.3.2 头文件

- `Inc/interrupt.h`：中断 API 声明
- `Inc/config.h`：临界区宏
- `Inc/hrtos\_internal.h`：内部中断变量

11.4 数据结构

11.4.1 临界区变量

位于 `Inc/hrtos\_internal.h`：

```
extern bit os_int_status;           // 进入时保存的中断状态
extern unsigned char idata os_critical_nesting; // 嵌套计数器
```

11.4.2 中断控制变量

位于 `Src/kernel/os\_core.c`：

```
volatile bit OS_ISR_FLAG;           // ISR 发生标志
volatile bit OS_INTERRUPT_GAO;      // 高优先级中断标志
volatile bit OS_INTERRUPT_GAO_BIAO; // 高优先级中断指示器
volatile unsigned char data OS_INTERRUPT_PROTECT; // 中断 ID 保护
volatile unsigned char data OS_INT_BAO[6]; // 中断上下文保存 (ACC、B、DPH、DPL、PSW、EA)
```

11.4.3 中断地址向量

```
volatile unsigned char xdata OS_INTERRUPT_ADDR[2]; // 中断处理程序地址
volatile unsigned char xdata OS_INTERRUPT_ADDR2[2]; // 嵌套中断处理程序地址
```

11.5 核心函数

11.5.1 os_enter_critical()

****位置****：`Src/interrupt/enter\_critical.c`

****目的****：进入临界区（禁用中断）

****过程****：

```
void os_enter_critical()
{
    if(os_critical_nesting == 0) {
        os_int_status = EA; // 保存当前中断状态
        EA = 0;             // 禁用中断
    }
    os_critical_nesting++; // 递增嵌套计数器
}
```

****设计说明****：

- 支持嵌套临界区
- 仅在首次进入时禁用中断
- 在最终退出时恢复原始状态
- 防止中断重入问题

11.5.2 os_exit_critical()

****位置****: `Src/interrupt/exit\_critical.c`

****目的****: 退出临界区（恢复中断）

****过程****:

```
void os_exit_critical()
{
    os_critical_nesting--;          // 递减嵌套计数器
    if(os_critical_nesting == 0) {
        EA = os_int_status;        // 恢复中断状态
    }
}
```

****设计说明****:

- 仅在嵌套达到 0 时启用中断
- 恢复原始中断状态
- 对嵌套临界区安全

11.5.3 os_in_isr()

****位置****: `Src/interrupt/in\_isr.c`

****目的****: 检查当前是否在 ISR 上下文中执行

****返回****:

- 1: 在 ISR 上下文中
- 0: 在任务上下文中

****实现****: 检查内部 ISR 标志

11.5.4 os_interrupt_id()

****位置****: `Src/interrupt/interrupt\_id.c`

****目的****: 获取当前中断 ID

****返回****: 中断号（0、2-31，基于自然进入顺序）

****实现****: 读取中断向量或内部保护寄存器

11.5.5 os_schedule_request()

****位置****: `Src/interrupt/schedule\_request.c`

****目的****: 从 ISR 请求任务重新调度

****参数****:

- `id`: 保留参数

****返回****: 成功返回 1

****过程****:

```
char os_schedule_request(char id)
{
    id = id;                      // 保留供将来使用
}
```

```

    OS_SCHED_REASON = 1;          // 设置调度触发标志
    TF0 = 1;                      // 触发 Timer0 中断
    return 1;
}

```

****设计说明**:**

- 不直接调用调度器
- 通过 Timer0 中断触发调度
- 对 ISR 上下文安全
- 确保调度在安全点发生

11.5.6 ISR 安全 IPC 函数

11.5.6.1 os_event_set_from_isr()

****位置**:** `Src/interrupt/event\_set\_from\_isr.c`

****目的**:** 从 ISR 设置事件标志

****参数**:**

- `id`: 事件资源 ID

****返回**:** 成功返回 1, 失败返回 -1

****过程**:**

1. 验证资源 ID
2. 设置事件值
3. 递增挂起信号计数器
4. 调用 `os\_schedule\_request()`

11.5.6.2 os_semaphore_post_from_isr()

****位置**:** `Src/interrupt/semaphore\_post\_from\_isr.c`

****目的**:** 从 ISR 释放信号量

****参数**:**

- `obj`: 信号量资源 ID

****返回**:** 成功返回 1, 失败返回 -1

****过程**:**

1. 验证资源 ID
2. 递增信号量值或挂起信号
3. 调用 `os\_schedule\_request()`

11.5.6.3 os_msgq_send_from_isr()

****位置**:** `Src/interrupt/msgq\_send\_from\_isr.c`

****目的**:** 从 ISR 发送消息

****参数**:**

- `q`: 队列指针

- `obj`：资源 ID
- `\_data`：消息数据
- **返回****：成功返回 1，失败返回 0（队列已满）
- **过程****：
 1. 检查队列是否已满
 2. 如果已满，返回失败（ISR 中不阻塞）
 3. 写入数据到队列
 4. 更新队列指针
 5. 调用 `os\_schedule\_request()`

11.6 调用关系

11.6.1 临界区流程

```
graph TD
    A[任务调用 os_enter_critical] --> B{嵌套 == 0?}
    B -->|是| C[保存 EA 到 os_int_status]
    B -->|否| D[跳过保存]
    C --> E[设置 EA = 0]
    D --> E
    E --> F[递增 os_critical_nesting]
    F --> G[临界代码]
    G --> H[任务调用 os_exit_critical]
    H --> I[递减 os_critical_nesting]
    I --> J{嵌套 == 0?}
    J -->|是| K[从 os_int_status 恢复 EA]
    J -->|否| L[跳过恢复]
    K --> M[返回]
    L --> M
```

11.6.2 ISR 到任务通知流程

```
graph TD
    flowchart TD
    A[硬件中断] --> B[ISR 入口]
    B --> C[读取硬件状态]
    C --> D[调用 ISR 安全 API]
    D --> E{API 类型?}
    E -->|事件| F[os_event_set_from_isr]
    E -->|信号量| G[os_semaphore_post_from_isr]
    E -->|消息| H[os_msgq_send_from_isr]
    F --> I[更新资源状态]
    G --> I
    H --> I
    I --> J[设置 OS_SCHED_REASON]
    J --> K[ISR 退出]
    K --> L[内核中断退出处理]
    L --> M{检查 OS_SCHED_REASON}
    M -->|需要调度| N[执行任务唤醒与调度]
    M -->|无需调度| O[恢复被中断任务]
    N --> P[切换到就绪任务]
    O --> Q[任务处理事件]
```

11.7 生命周期

11.7.1 临界区生命周期

1. ****进入****: 调用 ``os_enter_critical()``
2. ****嵌套检查****: 如果是首次进入, 保存中断状态
3. ****禁用****: 设置 ``EA = 0`` (如果是首次进入)
4. ****递增****: 增加嵌套计数器
5. ****临界代码****: 执行受保护代码
6. ****退出****: 调用 ``os_exit_critical()``
7. ****递减****: 减少嵌套计数器
8. ****恢复****: 如果嵌套达到 0, 恢复中断状态

11.7.2 ISR 执行生命周期

1. ****中断****: 发生硬件中断
2. ****上下文保存****: 硬件保存上下文
3. ****ISR 入口****: ISR 执行
4. ****ISR 安全 API****: 可能调用 ISR 安全函数
5. ****调度请求****: 可能请求重新调度
6. ****ISR 退出****: ISR 返回
7. ****上下文恢复****: 硬件恢复上下文
8. ****调度****: 如果请求, 调度器运行

11.8 设计原则

11.8.1 嵌套临界区

- 支持嵌套临界区
- 嵌套计数器跟踪深度
- 仅在首次进入时禁用
- 仅在最终退出时启用
- 防止过早启用中断

11.8.2 ISR 安全

- 独立的 ISR 安全 API
- ISR 中无阻塞操作
- 调度请求而非直接调度
- 最小 ISR 执行时间

11.8.3 延迟调度

- ISR 不直接调用调度器
- 使用 Timer0 中断作为安全点
- 防止重入问题
- 确保状态一致

11.8.4 临界区宏

```
#define OS_ENTER_CRITICAL()    do{ os_enter_critical(); }while(0)
#define OS_EXIT_CRITICAL()    do{ os_exit_critical(); }while(0)
```

- 安全的宏包装
- Do-while(0) 用于正确的语句分组
- 一致的 API 使用

11.9 约束

- 最大嵌套深度受限于 8 位计数器
- ISR 不能调用阻塞 API
- ISR 不能使用互斥锁
- ISR 必须简短（无复杂逻辑）
- 仅通过 Timer0 进行调度请求
- 内核中无中断优先级配置

11.10 ISR 规则

11.10.1 ISR 中允许

- 读取硬件寄存器
- 设置事件标志
- 释放信号量
- 发送消息（非阻塞）
- 设置简单状态标志
- 调用 `os\_schedule\_request()`

11.10.2 ISR 中禁止

- 调用阻塞 API（`os\_delay`、`os\_wait` 等）
- 使用互斥锁
- 接收消息
- 长时间计算
- 动态内存分配

- 递归调用内核

11.10.3 ISR 设计模式

```
void example_isr(void) interrupt VECTOR
{
    // 1. 读取硬件状态（最小化）
    u8 status = HW_STATUS_REG;

    // 2. 清除中断源
    HW_CLEAR_REG = status;

    // 3. 通知任务（ISR 安全 API）
    if (status & EVENT_FLAG) {
        os_event_set_from_isr(EVENT_ID);
    }

    // 4. 快速退出
}
```

11.11 性能考虑

11.11.1 临界区开销

- 最小开销（保存/恢复 EA）
- 嵌套增加小的计数器递增/递减
- 临界区中无上下文切换

11.11.2 ISR 延迟

- 临界区增加 ISR 延迟
- 保持临界区简短
- 嵌套增加最大延迟
- 在保护和响应性之间平衡

11.11.3 调度请求开销

- 用于调度的 Timer0 中断
- 调度器运行前的小延迟
- 安全但不立即
- 对大多数应用可接受

11.12 中断优先级级别

11.12.1 8051 中断优先级

HRTOS 使用 8051 的自然中断优先级：

- 外部中断 0（最高）

- Timer0
- 外部中断 1
- Timer1
- 串行口（最低）

11.12.2 内核中断级别

```
/* 调度级别定义：  
* 0-7    普通任务优先级  
* 8      高优先级任务  
* 9      中断级  
* 10     中断嵌套级  
*/  
#define OS_SCHED_LEVEL_MAX 10  
- 优先级 9：中断处理程序任务  
- 优先级 10：嵌套中断处理程序  
- 高于标准和快速任务
```

11.13 与其他 RTOS 的比较

11.13.1 vs 嵌套中断

- HRTOS：有限的嵌套支持
- 其他 RTOS：完全的嵌套中断支持

11.13.2 vs 中断锁定

- HRTOS：临界区（禁用所有中断）
- 其他 RTOS：中断屏蔽（选择性禁用）

11.13.3 vs ISR 延迟处理

- HRTOS：基于 Timer0 的延迟调度
- 其他 RTOS：软件中断或直接调度

Memory Layout

12 HRTOS 内存布局

12.1 模块介绍

HRTOS 专为内存资源有限的 8051 微控制器设计。内存布局经过精心组织，在满足 8051 架构约束的同时最大化效率，8051 架构有三个主要内存段：DATA、IDATA 和 XDATA。

12.2 主要职责

内存布局处理：

- 频繁访问变量的 DATA 段分配
- 内核状态和关键变量的 IDATA 段
- 大型数据结构的 XDATA 段
- 任务栈管理
- 上下文保存区域
- 栈分配的内存位图
- 内存溢出保护

12.3 主要文件

12.3.1 源文件

- `Src/kernel/os\_core.c`：内存布局定义和变量放置
- `Src/kernel/os\_data.c`：DATA 段初始化
- `Src/kernel/os\_xdata.c`：XDATA 段初始化

12.3.2 头文件

- `Inc/config.h`：栈配置和内存常量
- `Inc/hrtos\_internal.h`：内部内存布局定义

12.4 数据结构

12.4.1 内存配置常量

位于 `Inc/hrtos\_internal.h`:

```
/* 栈配置 */
#define OS_SP_KER 5 /* 系统栈大小 */
#define OS_SP_INIT 53 /* 系统初始 SP 值 */
#define OS_APP_STACK_MAX 0x80 /* 栈溢出阈值 */
#define OS_USER_RAM_INIT 66 /* 用户栈起始 */
#define OS_USER_RAM_EXIT 127 /* 用户栈结束 */
#define OS_SP_INTERRUPT_RETURN 58 /* 中断返回 SP */
#define OS_KUAI_SHENDU 5 /* 快速任务上下文深度 */
#define OS_JINCHENG_SHENDU 13 /* 标准任务上下文深度 */

/* 内存管理 */
#define OS_MEMORY_ADMINISTRATION 4 /* 内存管理大小 */
```

12.4.2 DATA 段布局 (0x20-0x34)

位于 `Src/kernel/os\_core.c`:

```
/* 0x30 - 中断 ID 保护 */
volatile unsigned char data OS_INTERRUPT_PROTECT_at_ 0x30;

/* 0x31 - R0 寄存器保护 */
volatile unsigned char data OS_R0_PROTECT_at_ 0x31;

/* 0x32 - 通用工作变量 */
volatile unsigned char data os_sp_at_ 0x32;

/* 0x33 - 内核临时变量 */
volatile unsigned char data OS_HRTOS_at_ 0x33;

/* 0x34 - 中断保护寄存器 */
volatile unsigned char data OS_HRTOS_2_at_ 0x34;
```

12.4.3 栈区域 (DATA 段)

```
/* 0x35-0x39 - 系统栈 (5 字节) */
unsigned char data OS_HRTOS_DUIZHAN_OS[OS_SP_KER] _at_ 0x35;

/* 0x3A-0x41 - 中断栈 (8 字节) */
unsigned char data OS_HRTOS_DUIZHAN_INT[8] _at_ 0x3a;

/* 0x42-0x7F - 用户任务栈 (62 字节) */
unsigned char data OS_HRTOS_DUIZHAN_APP[62] _at_ 0x42;
```

12.4.4 IDATA 段布局 (0xE0-0xE7)

```
/* 0xE7 - 临界区嵌套 */
volatile unsigned char idata os_critical_nesting_at_ 0xE7;

/* 0xE6 - 时间片控制 */
volatile unsigned char idata OS_TIME_XY_at_ 0xE6;

/* 0xE5 - 分发触发源 */
volatile unsigned char idata OS_DISPATCH_ID_at_ 0xE5;

/* 0xE4 - 前一个任务 */
volatile unsigned char idata OS_PREV_TASK_at_ 0xE4;
```

```

/* 0xE3 - 当前任务 */
volatile unsigned char idata OS_CURRENT_TASK _at_ 0xE3;

/* 0xE2 - Timer0 TH0 */
volatile unsigned char idata OS_T0_TH0 _at_ 0xE2;

/* 0xE1 - Timer0 TL0 */
volatile unsigned char idata OS_T0_TL0 _at_ 0xE1;

/* 0xDB-0xE0 - 中断上下文保存 (6 字节) */
volatile unsigned char idata OS_INT_BAO[6] _at_ 0xDB;

```

12.4.5 XDATA 段布局 (0x0200-0x03FF)

基地址：0x0200，总大小：512 字节

12.4.5.1 偏移映射

```

#define OS_SYS_GLOBAL_BASE    0x0200
#define OS_SYS_GLOBAL_SIZE    512

/* 任务状态 / 控制 */
#define OS_OFF_PROCESS_OK      0x0000 /* 任务状态数组 (16B) */
#define OS_OFF_JINCHENG_SP     0x0010 /* 标准任务 SP (16B) */
#define OS_OFF_KUAI_SP        0x0020 /* 快速任务 SP (2B) */

/* 上下文保存区域 */
#define OS_OFF_FAST_CONTEXT    0x0030 /* 快速任务上下文 (10B) */
#define OS_OFF_TASK_CONTEXT    0x0040 /* 标准任务上下文 (208B) */

/* 内存 / 栈管理 */
#define OS_OFF_MEMORY          0x0110 /* 栈位图 (4B) */
#define OS_OFF_JINCHENG_JILU_SP 0x0120 /* SP 备份 (16B) */

/* 时间 / 调度器 */
#define OS_OFF_TIME_ONCE       0x0130 /* 时间片 (1B) */
#define OS_OFF_WAIT_DI2        0x0131 /* 快速任务延时 (4B) */

/* 中断控制 */
#define OS_OFF_INTERRUPT_ADDR   0x0135 /* 中断地址 (2B) */
#define OS_OFF_INTERRUPT_ADDR2 0x0137 /* 中断地址 2 (2B) */

/* 事件 / 时钟系统 */
#define OS_OFF_TICK_L          0x0139 /* 时钟低 16 位 (2B) */
#define OS_OFF_TICK_H          0x013B /* 时钟高 16 位 (2B) */
#define OS_OFF_EVENT_BIT       0x013D /* 事件位图 (2B) */

/* 内存保护 / 调试 */
#define OS_OFF_STACK_WARN      0x013F /* 栈警告 (1B) */
#define OS_OFF_KUAI_BEI        0x0140 /* 快速任务 SP 起始 (2B) */
#define OS_OFF_KUAI_BEI2       0x0142 /* 快速任务 SP 大小 (2B) */
#define OS_OFF_TASK_EVENT      0x0144 /* 任务事件标志 (18B) */

/* 时间备份 */
#define OS_OFF_TIME_BACKUP      0x0156 /* 时间片备份 (1B) */

```

12.4.5.2 XDATA 变量放置

```

/* 任务状态 / 控制 */
volatile unsigned char xdata OS_PROCESS_OK[OS_PROCESS_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_PROCESS_OK);

```

```

volatile unsigned char xdata OS_JINCHENG_SP[OS_PROCESS_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_JINCHENG_SP);

volatile unsigned char xdata OS_KUAI_SP[OS_FAST_TASK_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_KUAI_SP);

/* 上下文保存区域 */
volatile unsigned char xdata OS_FAST_CONTEXT[OS_FAST_TASK_MAX]
[OS_KUAI_SHENDU]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_FAST_CONTEXT);

volatile unsigned char xdata OS_TASK_CONTEXT[OS_PROCESS_MAX]
[OS_JINCHENG_SHENDU]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TASK_CONTEXT);

/* 内存 / 栈管理 */
volatile unsigned char xdata OS_MEMORY[OS_MEMORY_ADMINISTRATION]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_MEMORY);

volatile unsigned char xdata OS_JINCHENG_JILU_SP[OS_PROCESS_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_JINCHENG_JILU_SP);

/* 时间 / 调度器 */
volatile unsigned char xdata OS_TIME_ONCE
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TIME_ONCE);

volatile unsigned int xdata OS_WAIT_DI2[OS_FAST_TASK_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_WAIT_DI2);

/* 中断控制 */
volatile unsigned char xdata OS_INTERRUPT_ADDR[2]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_INTERRUPT_ADDR);

volatile unsigned char xdata OS_INTERRUPT_ADDR2[2]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_INTERRUPT_ADDR2);

/* 事件 / 时钟系统 */
volatile unsigned int xdata OS_TICK_COUNT_L
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TICK_L);

volatile unsigned int xdata OS_TICK_COUNT_H
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TICK_H);

volatile unsigned int xdata OS_EVENT_BIT
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_EVENT_BIT);

/* 内存保护 / 调试 */
volatile unsigned char xdata OS_INSIDE_STACK_USE_MAXIMUM
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_STACK_WARN);

volatile unsigned char xdata OS_SP_KUAI_BEI[OS_FAST_TASK_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_KUAI_BEI);

volatile unsigned char xdata OS_SP_KUAI_BEI2[OS_FAST_TASK_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_KUAI_BEI2);

volatile unsigned char xdata OS_TASK_EVENT[OS_PROCESS_MAX +
OS_FAST_TASK_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TASK_EVENT);

```

```

/* 时间备份 */
volatile unsigned char xdata OS_TIME_ONCE_BACKUP
    _at_ (OS_SYS_GLOBAL_BASE + OS_OFF_TIME_BACKUP);

```

12.5 核心函数

12.5.1 栈分配（位图）

位于 `Src/kernel/task\_create.c`:

```

// 基于位图的栈分配
for(i=0; i<(OS_USER_RAM_EXIT-OS_USER_RAM_INIT)/2; i++)
{
    x=i/8;                // 位图字节索引
    OS_R0_PROTECT=i%8;    // 位位置
    x=OS_MEMORY[x]&(0x80>>OS_R0_PROTECT);
    if(x==0)              // 空闲块
    {
        k++;              // 计数连续空闲
        if(k>=sd)         // 足够空间
        {
            // 标记为已使用
            for(j=i-k+1; j<=i; j++)
            {
                x=j/8;
                OS_R0_PROTECT=j%8;
                OS_MEMORY[x]|=(0x80>>OS_R0_PROTECT);
            }
        }
    }
    else
    {
        k=0;              // 重置计数器
    }
}

```

12.5.2 栈释放

位于 `Src/kernel/task\_cleanup.c`:

```

// 释放栈块
for(; OS_HRTOS<m; OS_HRTOS++)
{
    os_sp=OS_HRTOS>>3;
    OS_R0_PROTECT=OS_HRTOS&7;
    j=(~(0x80>>OS_R0_PROTECT));
    OS_MEMORY[os_sp]&=j;
}

```

12.6 内存映射摘要

12.6.1 DATA 段（0x00-0x7F）

地址范围	大小	用途
0x00-0x1F	32 字节	寄存器组（R0-R7）

0x20-0x2F 16 字节 可位寻址区域
0x30-0x34 5 字节 内核临时变量
0x35-0x39 5 字节 系统栈
0x3A-0x41 8 字节 中断栈
0x42-0x7F 62 字节 用户任务栈区域

12.6.2 IDATA 段 (0x80-0xFF)

地址范围 大小 用途
----- ----- -----
0x80-0xFF 128 字节 上 128 字节 (内部 RAM)
0xE0-0xE7 8 字节 内核状态变量
0xE8-0xFF 24 字节 保留

12.6.3 XDATA 段 (0x0000-0xFFFF)

地址范围 大小 用途
----- ----- -----
0x0000-0x01FF 512 字节 应用程序使用
0x0200-0x03FF 512 字节 HRTOS 全局池
0x0400+ - 应用程序使用

12.7 设计原则

12.7.1 8051 内存架构

- **DATA**: 直接寻址，最快访问
- **IDATA**: 间接寻址，上 128 字节
- **XDATA**: 外部内存，更大但更慢

12.7.2 访问频率优化

- 频繁访问的变量在 DATA/IDATA
- 大型数组在 XDATA
- 关键状态在 IDATA (0xE0-0xE7)

12.7.3 栈管理

- 基于位图的分配
- 2 字节块
- 用户栈: 0x42-0x7F (62 字节 = 31 块)
- 系统栈和中断栈分离

12.7.4 上下文保存优化

- 快速任务：5 个寄存器（最小）
- 标准任务：13 个寄存器（完整）
- 位于 XDATA 以提高空间效率

12.8 约束

- 总用户栈：62 字节
- 最大任务栈：受可用块限制
- XDATA 池：总共 512 字节
- 无动态内存分配
- 栈溢出检测有限
- 不处理内存碎片（静态分配）

12.9 内存使用摘要

12.9.1 DATA 段使用

- 内核变量：5 字节（0x30-0x34）
- 系统栈：5 字节（0x35-0x39）
- 中断栈：8 字节（0x3A-0x41）
- 用户栈：62 字节（0x42-0x7F）
- 总计：80 字节

12.9.2 IDATA 段使用

- 内核状态：13 字节（0xDB-0xE7）
- 总计：13 字节

12.9.3 XDATA 段使用

- 任务状态：16 字节
- 栈指针：18 字节
- 上下文保存：218 字节
- 内存位图：4 字节
- 定时：11 字节
- 中断：4 字节
- 事件/时钟：6 字节
- 调试：24 字节
- 总计：~301 字节（512 字节中）

12.10 移植考虑

12.10.1 对于不同的 8051 变体

- 根据可用 RAM 调整栈大小
- 如需要修改 XDATA 基地址
- 为不同内存大小更新偏移映射
- 为不同架构调整上下文保存深度

12.10.2 内存配置

```
// 为不同目标调整这些
#define OS_USER_RAM_INIT          66
#define OS_USER_RAM_EXIT          127
#define OS_SYS_GLOBAL_BASE        0x0200
#define OS_SYS_GLOBAL_SIZE        512
```

13 HRTOS 移植层

13.1 模块介绍

移植层用于隔离硬件相关实现与 HRTOS 内核核心代码，使系统能够适配不同 8051

系列 MCU。

HRTOS 当前主要面向 8051

架构优化，移植工作主要集中在硬件相关配置、定时器、中断以及编译环境适配。

13.2 主要职责

移植层主要负责：

- 硬件寄存器定义
- 定时器配置
- 中断入口配置
- 临界区控制
- 特定编译器适配
- 硬件相关初始化

上下文切换、任务栈布局等核心机制依赖具体 CPU

架构和编译器实现，需要根据目标平台进行验证。

13.3 主要文件

13.3.1 头文件

- ``Inc/hrtos.h``：系统公共接口及硬件相关定义
- ``Inc/config.h``：系统配置

13.3.2 源文件

- ``Src/port/``：硬件相关移植代码

具体文件结构根据版本和目标平台调整。

13.4 硬件依赖

13.4.1 寄存器定义

HRTOS 在 8051 平台通常依赖编译器提供的寄存器定义：

```
#include <reg52.h>
```

用于访问：

- 特殊功能寄存器
- I/O 端口
- 定时器寄存器
- 中断控制寄存器

不同 MCU 型号需要使用对应头文件。

13.5 定时器适配

HRTOS 需要一个硬件定时资源用于系统时基。

默认情况下使用 Timer0。

移植时需要根据目标芯片调整：

- 工作模式
- 重载值
- 中断入口
- 时钟频率

示例：

```
void timer0_init(void)
{
    // 根据目标芯片配置 Timer0
}
```

定时器配置完成后，需要验证：

- 系统节拍准确性
- 延时功能
- 任务调度周期

13.6 中断适配

移植时需要确认：

- 中断向量定义
- 中断开启方式
- 临界区实现方式

中断相关代码必须保证：

- ISR 执行时间短

- 不进行阻塞操作
- 正确处理调度请求

13.7 内存配置

不同 MCU 的 RAM 和 XDATA 资源不同，需要根据实际硬件调整。

需要关注：

- 内部 RAM 使用
- 系统变量区域
- 任务栈空间
- 外部数据空间

配置修改后需要重新验证：

- 任务数量
- 栈空间
- IPC 资源
- 长时间运行稳定性

13.8 移植流程

13.8.1 确认目标硬件

分析：

- CPU 架构
- 时钟频率
- RAM 容量
- 定时器资源
- 中断资源

13.8.2 配置硬件环境

完成：

- 寄存器定义
- 时钟配置
- 定时器配置
- 中断配置

13.8.3 调整系统参数

根据资源情况调整：

- 内存布局

- 任务数量
- 栈大小
- IPC 数量

13.8.4 功能验证

测试：

- 系统启动
- 任务创建
- 任务切换
- 延时功能
- IPC 功能
- 中断响应

13.9 8051 平台适配

13.9.1 标准 8051

主要特点：

- 资源有限
- 内部 RAM 较小
- 外设资源较少

需要重点关注：

- 内存使用
- 栈空间
- 代码大小

13.9.2 增强 8051

可能提供：

- 更多 RAM
- 更多定时器
- 更高主频
- 更多外设

可以根据资源情况调整系统配置。

13.10 移植约束

13.10.1 硬件约束

目标硬件需要具备：

- 可用定时器资源
- 中断支持
- 足够 RAM
- 满足系统运行需求的存储空间

13.10.2 实时性约束

移植后需要保证：

- 调度周期稳定
- 中断响应可预测
- 任务切换正常

13.10.3 兼容性约束

移植代码应：

- 与内核保持隔离
- 减少核心代码修改
- 保持接口一致

13.11 移植测试

13.11.1 基础测试

验证：

- 系统启动
- 第一个任务运行
- 调度正常

13.11.2 时间测试

验证：

- 系统节拍
- 延时精度
- 时间片周期

13.11.3 稳定性测试

验证：

- 多任务运行
- IPC 使用
- 长时间运行

13.12 当前限制

- 当前主要针对 8051 架构优化
- 不提供通用跨架构 HAL
- 不支持多核处理
- 不支持 MPU 等硬件保护机制

13.13 未来优化方向

未来可以根据实际需求：

- 增加更完善的硬件抽象
- 优化移植接口
- 增加更多 MCU 支持
- 完善驱动扩展方式

13.14 总结

HRTOS 移植层的目标是在保持内核稳定性的基础上隔离硬件差异。

当前版本优先保证 8051

平台的可靠性和易理解性，移植到其他平台时应以实际硬件验证结果为准。

Kernel Startup

14 HRTOS 内核启动

14.1 模块介绍

内核启动负责完成 HRTOS 从系统复位到进入运行状态的初始化过程。

启动流程主要包括：

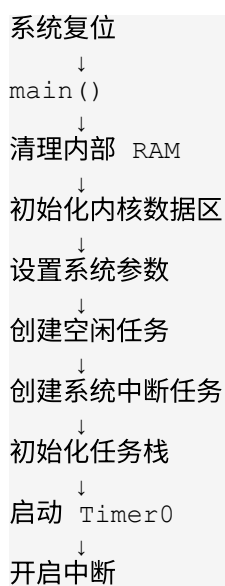
- 清理系统运行环境
- 初始化内核数据区域
- 配置系统参数
- 创建系统基础任务
- 启动 Timer0 系统时基
- 进入空闲任务
- 等待中断触发调度

HRTOS 当前启动流程基于 8051

架构设计，系统初始化完成后首先进入空闲任务，由定时器中断和任务调度机制驱动后续运行。

14.2 启动流程概述

系统启动过程：



↓
进入 `os_idle_task()`
↓
等待调度运行用户任务

14.3 系统入口

14.3.1 `main()`

HRTOS 启动入口为用户工程中的 ``main()`` 函数。

主要职责：

- 清理 MCU 初始状态
- 初始化 HRTOS 内核变量
- 创建系统任务
- 配置系统定时器
- 开启中断
- 进入空闲任务

14.4 内存初始化

14.4.1 内部 RAM 清理

系统启动首先清理 8051 内部 RAM。

目的：

- 保证变量初始状态确定
- 避免复位后的随机数据影响系统运行

典型流程：

设置地址指针
↓
清零累加器
↓
循环清除内部 RAM

14.4.2 内核数据区初始化

内部 RAM 初始化完成后，清理 HRTOS 使用的数据区域。

包括：

- 任务状态
- 调度变量
- IPC 资源
- 系统计数器

14.5 内核参数初始化

启动过程中需要设置核心运行参数。

主要包括：

14.5.1 当前任务

```
OS_CURRENT_TASK = 0;
```

系统初始运行任务编号为 0。

14.5.2 系统模式

```
OS_RTOS_YES = 1;
```

表示启用 RTOS 调度模式。

14.5.3 调度状态

初始化：

- 调度原因
- 中断状态
- 时间片计数

14.5.4 Tick 计数器

初始化系统运行时间统计：

- 低 16 位计数器
- 高 16 位计数器

用于记录系统运行时间。

14.6 系统任务创建

14.6.1 空闲任务

系统首先创建空闲任务：

```
os_task_create(os_idle_task, 0, 0, 2);
```

空闲任务负责：

- 系统入口
- 空闲处理
- 任务资源清理

14.6.2 系统中断任务

创建系统保留任务：

- 中断退出处理任务
- 高优先级中断相关任务

用于支持系统内部调度流程。

14.7 定时器初始化

14.7.1 Timer0 配置

HRTOS 使用 Timer0 作为系统时间基准。

初始化内容：

- 设置重载值
- 配置工作模式
- 开启 Timer0 中断
- 启动定时器

示例：

```
TMOD |= 0x01;
ET0 = 1;
TR0 = 1;
```

Timer0 负责：

- 系统 Tick
- 时间片管理
- 调度触发

14.8 进入空闲任务

完成初始化后：

```
os_idle_task();
```

系统正式进入 RTOS 运行状态。

空闲任务是 HRTOS 启动后的第一个执行任务。

其职责：

- 完成运行阶段初始化
- 调用用户初始化入口
- 执行空闲循环

14.9 空闲任务运行流程

```
进入 os_idle_task
↓
初始化运行状态
↓
调用 hrtos_main()
↓
用户创建应用任务
↓
开启系统运行
↓
进入空闲循环
```

14.10 hrtos_main()

14.10.1 作用

`hrtos\_main()` 是用户应用初始化入口。

用于：

- 创建用户任务
- 初始化 IPC
- 配置应用模块
- 完成启动阶段配置

示例：

```
void hrtos_main(void)
{
    os_task_create(task1, ...);
    os_task_create(task2, ...);

    os_sem_init(...);
}
```

14.10.2 注意事项

`hrtos\_main()` 应：

- 不执行长时间阻塞操作
- 不进行无限循环
- 快速完成初始化

14.11 调度启动

系统启动完成后：

1. Timer0 产生系统中断
2. 内核更新 Tick
3. 判断是否需要调度
4. 选择最高优先级任务
5. 完成任务切换

任务开始正式运行。

14.12 启动阶段划分

14.12.1 硬件初始化阶段

主要工作：

- CPU 复位

- RAM 清理
- 基础寄存器初始化

14.12.2 内核初始化阶段

主要工作：

- 初始化内核变量
- 初始化任务环境
- 初始化资源

14.12.3 系统启动阶段

主要工作：

- 创建系统任务
- 启动 Timer0
- 开启中断

14.12.4 运行阶段

主要工作：

- 空闲任务运行
- Timer0 驱动调度
- 用户任务执行

14.13 启动约束

启动过程中：

- 初始化期间避免中断干扰
- 任务创建需要在系统启动阶段完成
- Timer0 配置必须正确
- 用户初始化不能长时间占用 CPU

14.14 启动调试

14.14.1 系统无法运行

检查：

- main 是否正常执行
- RAM 初始化是否完成
- Timer0 是否启动
- 总中断是否开启

14.14.2 用户任务不执行

检查：

- 任务创建返回值
- 任务优先级
- 任务状态
- 调度是否触发

14.14.3 系统异常复位

检查：

- 栈空间
- 内存越界
- 中断嵌套
- 用户任务运行时间

14.15 总结

HRTOS 启动过程采用简单明确的初始化流程：

硬件环境准备 → 内核初始化 → 创建基础任务 → 启动定时器 → 进入空闲任务 → 调度运行。

该设计保持启动过程可控、结构清晰，符合轻量级 8051 实时系统的设计目标。

State Machine

15 HRTOS 状态机

15.1 模块介绍

HRTOS 使用状态机来管理任务状态、IPC 资源状态和系统模式。本文档描述 HRTOS 中的各种状态机、它们的转换以及触发状态更改的事件。

15.2 主要职责

状态机处理：

- 任务生命周期状态
- IPC 资源状态
- 等待状态类型
- 系统模式状态
- 调度原因状态

15.3 主要文件

15.3.1 源文件

- `Src/task/task\_state.c`：任务状态查询
- `Src/wait/os\_wait.c`：等待状态管理
- `Src/wait/wake\_task.c`：唤醒时的状态转换
- `Src/kernel/scheduler.c`：调度器模式状态

15.3.2 头文件

- `Inc/config.h`：状态定义和常量
- `Inc/hrtos\_internal.h`：内部状态变量

15.4 数据结构

15.4.1 任务状态

位于 `Inc/config.h`：

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

15.4.2 等待类型

```
#define WAIT_NONE    0    /* 无等待 */
#define WAIT_DELAY    1    /* 延时等待 */
#define WAIT_SEM      2    /* 信号量等待 */
#define WAIT_MUTEX    3    /* 互斥锁等待 */
#define WAIT_MSG_SEND 4    /* 消息队列发送等待 (队列满) */
#define WAIT_MSG_RECV 5    /* 消息队列接收等待 (队列空) */
#define WAIT_EVENT    6    /* 事件等待 */
#define WAIT_MAILBOX  7    /* 邮箱等待 */
```

15.4.3 等待标志

```
#define WAIT_TIMEOUT 1    /* 超时唤醒 */
#define WAIT_SIGNAL  2    /* 信号唤醒 */
```

15.4.4 TCB 状态字段

```
typedef struct {
    u8 state;          /* 任务状态 */
    u8 wait_type;      /* 等待类型 */
    u8 wait_flag;      /* 等待结果标志 */
    u8 wait_obj;       /* 等待对象 ID */
    u16 wait_tick;     /* 超时时钟周期 */
} OS_TCB;
```

15.5 核心状态机

15.5.1 任务状态机

15.5.1.1 状态图

```
stateDiagram-v2
    [*] --> DEAD: Initial
    DEAD --> READY: os_task_create
    READY --> RUNNING: Scheduler selects
    RUNNING --> READY: Preempted/Yield
    RUNNING --> WAIT: os_delay/os_wait
    RUNNING --> SUSPEND: os_task_suspend
    RUNNING --> DEAD: os_task_exit
    WAIT --> READY: wake_task/timeout
    SUSPEND --> READY: os_task_resume
    SUSPEND --> DEAD: os_task_delete
    READY --> DEAD: os_task_delete
    DEAD --> [*]
```

15.5.1.2 状态描述

- **DEAD (删除态) ****
- 任务不存在或已被删除
 - 无栈分配

- 不参与调度
- 所有任务的初始状态
- **READY (就绪态) ****
- 任务准备运行
- 栈已分配
- 等待调度器选择
- `OS\_PROCESS\_OK` 中设置就绪位
- **RUNNING (运行态) ****
- 任务当前正在执行
- `OS\_CURRENT\_TASK == task\_id`
- 拥有 CPU 控制
- 可能随时被抢占
- **WAIT (等待态) ****
- 任务在 IPC 或延时上阻塞
- 等待资源或超时
- 不参与调度
- `OS\_PROCESS\_OK` 中清除就绪位
- **SUSPEND (挂起态) ****
- 任务被显式挂起
- 不参与调度
- 可以恢复到就绪状态
- `OS\_PROCESS\_OK` 中清除就绪位

15.5.1.3 状态转换

| 从 | 到 | 触发 | 函数 |

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

| DEAD | READY | `os\_task\_create()` | 任务创建 |

| READY | RUNNING | 调度器 | 任务被选中 |

| RUNNING | READY | 抢占/让出 | 更高优先级任务或 `os\_task\_yield()` |

| RUNNING | WAIT | `os\_delay()`/`os\_wait()` | 阻塞操作 |

| RUNNING | SUSPEND | `os\_task\_suspend()` | 显式挂起 |

| RUNNING | DEAD | `os\_task\_exit()` | 任务退出 |

| WAIT | READY | `wake\_task()` | 资源可用 |

| WAIT | READY | 超时 | 定时器时钟周期到期 |

| SUSPEND | READY | `os\_task\_resume()` | 恢复任务 |

| READY | DEAD | `os\_task\_delete()` | 删除任务 |

| SUSPEND | DEAD | `os\_task\_delete()` | 删除挂起任务 |

15.5.2 等待类型状态机

15.5.2.1 等待类型图

```
stateDiagram-v2
    [*] --> WAIT_NONE: Initial
    WAIT_NONE --> WAIT_DELAY: os_delay
    WAIT_NONE --> WAIT_SEM: os_sem_wait
    WAIT_NONE --> WAIT_MUTEX: os_mutex_lock
    WAIT_NONE --> WAIT_MSG_SEND: os_msg_send full
    WAIT_NONE --> WAIT_MSG_RECV: os_msg_recv empty
    WAIT_NONE --> WAIT_EVENT: os_event_wait
    WAIT_NONE --> WAIT_MAILBOX: os_mailbox_recv
    WAIT_DELAY --> WAIT_NONE: wake_task/timeout
    WAIT_SEM --> WAIT_NONE: wake_task/timeout
    WAIT_MUTEX --> WAIT_NONE: wake_task/timeout
    WAIT_MSG_SEND --> WAIT_NONE: wake_task/timeout
    WAIT_MSG_RECV --> WAIT_NONE: wake_task/timeout
    WAIT_EVENT --> WAIT_NONE: wake_task/timeout
    WAIT_MAILBOX --> WAIT_NONE: wake_task/timeout
```

15.5.2.2 等待类型描述

****WAIT_NONE (无等待) ****

- 任务未等待
- 正常执行状态
- 可以被调度

****WAIT_DELAY (延时等待) ****

- 等待时间过去
- `wait\_tick` 倒计时
- 无资源对象

- 超时时自动唤醒

****WAIT_SEM (信号量等待) ****

- 等待信号量
- `wait\_obj` = 信号量 ID
- 在 `os\_sem\_post()` 时唤醒
- 超时时唤醒

****WAIT_MUTEX (互斥锁等待) ****

- 等待互斥锁
- `wait\_obj` = 互斥锁 ID
- 优先级继承激活
- 在 `os\_mutex\_unlock()` 时唤醒

****WAIT_MSG_SEND (消息发送等待) ****

- 等待队列空间

- `wait\_obj` = 队列资源 ID
- 空间可用时唤醒
- 超时时唤醒
- **WAIT_MSG_RECV (消息接收等待) ****
- 等待消息
- `wait\_obj` = 队列资源 ID
- 消息可用时唤醒
- 超时时唤醒
- **WAIT_EVENT (事件等待) ****
- 等待事件标志
- `wait\_obj` = 事件 ID
- 在 `os\_event\_write()` 时唤醒
- 超时时唤醒
- **WAIT_MAILBOX (邮箱等待) ****
- 等待邮箱数据
- `wait\_obj` = 邮箱 ID
- 数据可用时唤醒
- 超时时唤醒

15.5.3 等待标志状态机

15.5.3.1 等待标志图

```
stateDiagram-v2
    [*] --> 0: Initial
    0 --> WAIT_TIMEOUT: Timeout expires
    0 --> WAIT_SIGNAL: Resource available
    WAIT_TIMEOUT --> [*]
    WAIT_SIGNAL --> [*]
```

15.5.3.2 等待标志描述

- **0 (初始) ****
- 尚未唤醒
- 任务仍在等待
- **WAIT_TIMEOUT (超时唤醒) ****
- 由于超时而唤醒
- `wait\_tick` 达到 0
- 资源不可用
- **WAIT_SIGNAL (信号唤醒) ****
- 由于资源而唤醒
- 资源变为可用

- 未达到超时

15.5.4 资源状态机

15.5.4.1 资源状态图

```
stateDiagram-v2
    [*] --> Uninitialized
    Uninitialized --> Idle: os_res_init
    Idle --> Owned: Mutex lock
    Idle --> Available: Semaphore post
    Available --> Idle: Semaphore wait
    Owned --> Idle: Mutex unlock
    Idle --> Idle: Event write/clear
    Idle --> [*]: Resource delete
```

15.5.4.2 资源状态描述

****Uninitialized****

- 资源未初始化
- 无法使用

****Idle****

- 资源已初始化但未使用
- 无所有者（对于互斥锁）
- 值可能为 0 或非零

****Owned（仅互斥锁）****

- 互斥锁被任务持有
- 所有者字段设置为任务 ID
- 其他任务可能等待

****Available（仅信号量）****

- 信号量有可用资源
- 值 > 0
- 任务可以无需等待获取

15.5.5 调度器模式状态机

15.5.5.1 调度器模式图

```
stateDiagram-v2
    [*] --> HRTOS: Default
    HRTOS --> MYOS: os_set_scheduler(0)
    MYOS --> HRTOS: os_set_scheduler(1)
```

15.5.5.2 模式描述

****HRTOS 模式****

- 基于优先级的调度
- 时间片支持
- 互斥锁优先级继承

- 默认模式
- **MYOS 模式**
- 替代调度算法
- 基于时间片
- 无优先级继承
- 遗留模式

15.6 状态转换函数

15.6.1 wake_task()

****位置****: `Src/wait/wake\_task.c`

****目的****: 将任务从 WAIT 转换到 READY

****过程****:

```
void wake_task(u8 tid, u8 flag)
{
    u8 obj;
    obj = OS_TASK[tid].wait_obj;

    // 清除等待掩码
    if(obj != OS_INVALID_ID)
    {
        OS_RES[obj].wait_mask &= ~((u16)1 << tid);
        if(OS_RES[obj].wait_cnt)
        {
            OS_RES[obj].wait_cnt--;
        }
    }

    // 设置唤醒结果
    OS_TASK[tid].wait_flag = flag;

    // 清除等待上下文
    OS_TASK[tid].wait_type = WAIT_NONE;
    OS_TASK[tid].wait_obj = OS_INVALID_ID;
    OS_TASK[tid].wait_tick = 0;

    // 设置就绪状态
    OS_TASK[tid].state = READY;

    // 允许调度
    OS_PROCESS_OK[tid] |= 1;
    OS_SCHED_REASON = 1;
    TF0 = 1;
}
```

15.6.2 os_wait()

****位置****: `Src/wait/os\_wait.c`

****目的****: 将任务从 READY/RUNNING 转换到 WAIT

****过程****:

1. 设置 TCB 等待字段
2. 添加到资源等待队列
3. 将任务状态设置为 WAIT
4. 清除就绪位
5. 触发调度

15.7 状态查询

15.7.1 os_task_get_state()

****位置****: `Src/task/task\_state.c`

****目的****: 查询任务状态

****返回****:

- 0: 已删除
- 1: 运行中
- 2: 就绪
- 3: 挂起
- 4: 之前切换出

15.8 设计原则

15.8.1 状态一致性

- 所有状态更改在临界区中
- 原子状态转换
- 无中间无效状态

15.8.2 状态封装

- 状态存储在 TCB 中
- 仅通过函数访问
- 内部状态不暴露

15.8.3 确定性转换

- 每个转换都有清晰的触发
- 无自发状态更改
- 可预测的转换

15.9 约束

- 任务不能同时处于多个状态
- 等待类型和等待标志必须一致
- 状态转换必须是原子的
- 无直接状态操作（使用函数）

15.10 状态机调试

15.10.1 常见问题

1. ****任务卡在 WAIT****：检查唤醒机制
2. ****无效状态转换****：检查临界区
3. ****状态不一致****：检查并发访问
4. ****缺少唤醒****：检查资源操作

15.10.2 调试技术

- 监视 TCB 状态字段
- 跟踪状态转换
- 检查 wait_mask 一致性
- 验证调度器触发

Module Dependencies

16 HRTOS 模块依赖

16.1 模块介绍

HRTOS 由多个相互连接的模块组成。理解这些模块之间的依赖关系对于维护、调试和扩展内核至关重要。本文档描述了所有 HRTOS 模块之间的依赖关系。

16.2 主要职责

模块依赖描述：

- 哪些模块依赖于哪些
- 避免循环依赖
- 分层架构
- 初始化顺序
- API 使用边界

16.3 主要文件

16.3.1 模块结构

Src/	
— kernel/	(核心内核 - 不依赖于其他 Src 模块)
— wait/	(等待机制 - 被大多数模块使用)
— task/	(任务管理 - 依赖于 wait、task_priority)
— task_priority/	(优先级管理 - 依赖于 task)
— event/	(事件 - 依赖于 wait)
— mutex/	(互斥锁 - 依赖于 wait)
— semaphore/	(信号量 - 依赖于 wait)
— mailbox/	(邮箱 - 依赖于 wait)
— msgq/	(消息队列 - 依赖于 wait)
— time/	(时间 - 依赖于 wait)
— interrupt/	(中断 - 依赖于 event、msgq、semaphore)
— port/	(移植 - 无依赖)

16.4 依赖图

16.4.1 高级依赖图

graph TD

```

A[应用程序] --> B[公共 API]
B --> C[内核模块]
B --> D[任务模块]
B --> E[IPC 模块]
B --> F[时间模块]
B --> G[中断模块]

C --> H[等待模块]
D --> H
D --> I[任务优先级]
E --> H
F --> H
G --> E
G --> H

I --> D

J[移植模块] -. -> C
J -. -> D
J -. -> E
J -. -> F
J -. -> G

```

16.4.2 详细模块依赖

16.4.2.1 内核模块

****依赖**：**

- 等待模块（用于任务清理）
- 任务模块（用于任务操作）
- 中断模块（用于调度）

****被依赖**：**

- 所有其他模块（通过初始化）
- 应用程序代码

****特性**：**

- 具有最小依赖的核心模块
- 提供基本服务
- 首先初始化

16.4.2.2 等待模块

****依赖**：**

- 无（自包含）

****被依赖**：**

- 任务模块
- 事件模块
- 互斥锁模块
- 信号量模块
- 邮箱模块

- 消息队列模块
- 时间模块
- 内核模块
- **特性****:
- 中央等待机制
- 被所有阻塞操作使用
- 无循环依赖

16.4.2.3 任务模块

- **依赖****:
- 等待模块（用于阻塞操作）
- 任务优先级模块（用于优先级管理）
- **被依赖****:
- 内核模块
- 应用程序代码
- **特性****:
- 依赖于 wait 进行状态转换
- 依赖于 priority 进行优先级更改
- 核心功能

16.4.2.4 任务优先级模块

- **依赖****:
- 任务模块（用于任务操作）
- **被依赖****:
- 任务模块（通过设计解决循环依赖）
- **特性****:
- 与任务模块紧密耦合
- 管理优先级锁定
- 无其他依赖

16.4.2.5 事件模块

- **依赖****:
- 等待模块（用于事件等待）
- **被依赖****:
- 中断模块（用于 ISR 安全操作）
- 应用程序代码
- **特性****:
- 对 wait 的简单依赖

- 中断模块中的 ISR 安全版本

16.4.2.6 互斥锁模块

****依赖**:**

- 等待模块（用于互斥锁等待）

****被依赖**:**

- 应用程序代码

****特性**:**

- 依赖于 wait 进行阻塞
- 实现优先级继承
- 无 ISR 操作

16.4.2.7 信号量模块

****依赖**:**

- 等待模块（用于信号量等待）

****被依赖**:**

- 中断模块（用于 ISR 安全操作）
- 应用程序代码

****特性**:**

- 依赖于 wait 进行阻塞
- 中断模块中的 ISR 安全版本

16.4.2.8 邮箱模块

****依赖**:**

- 等待模块（用于邮箱等待）

****被依赖**:**

- 应用程序代码

****特性**:**

- 对 wait 的简单依赖
- 无 ISR 操作

16.4.2.9 消息队列模块

****依赖**:**

- 等待模块（用于队列操作）

****被依赖**:**

- 中断模块（用于 ISR 安全操作）
- 应用程序代码

****特性**:**

- 依赖于 wait 进行阻塞

- 中断模块中的 ISR 安全版本

16.4.2.10 时间模块

****依赖**:**

- 等待模块（用于延时操作）

****被依赖**:**

- 应用程序代码

****特性**:**

- 依赖于 wait 进行任务延时
- 定时器配置独立

16.4.2.11 中断模块

****依赖**:**

- 事件模块（用于 ISR 安全事件设置）
- 消息队列模块（用于 ISR 安全发送）
- 信号量模块（用于 ISR 安全释放）
- 等待模块（用于调度请求）

****被依赖**:**

- 应用程序代码
- 内核模块（用于调度）

****特性**:**

- 依赖于多个 IPC 模块
- 提供 ISR 安全包装器
- 无循环依赖

16.4.2.12 移植模块

****依赖**:**

- 无

****被依赖**:**

- 所有模块（通过硬件抽象）

****特性**:**

- 硬件抽象层
- 可选依赖
- 平台特定

16.5 依赖规则

16.5.1 无循环依赖

HRTOS 通过以下方式避免循环依赖：

1. ****分层架构****: 清晰的分层防止循环
2. ****中央等待模块****: 所有阻塞通过 wait
3. ****ISR 分离****: ISR 安全函数在单独模块中
4. ****优先级模块****: 紧密耦合但设计为避免循环

16.5.2 依赖方向

依赖单向流动:

```
HRTOS 内核
↑
等待
↑
任务、IPC、时间、中断
↑
应用程序 (顶层)
```

16.6 模块接口依赖

16.6.1 头文件依赖

```
hrtos.h (主入口)
├── config.h
├── wait.h
├── event.h
├── interrupt.h
├── kernel.h
├── mailbox.h
├── mutex.h
├── task.h
├── semaphore.h
├── msgq.h
└── time.h
```

16.7 API 使用依赖

16.7.1 任务模块 API 被...使用

- ****内核****: ``os_task_create()``、``os_task_cleanup()``
- ****应用程序****: 所有任务 API

16.7.2 等待模块 API 被...使用

- ****任务****: ``os_wait()`` (通过延时间接)
- ****事件****: ``os_wait()`` 用于事件等待
- ****互斥锁****: ``os_wait()`` 用于互斥锁等待
- ****信号量****: ``os_wait()`` 用于信号量等待
- ****邮箱****: ``os_wait()`` 用于邮箱接收
- ****消息队列****: ``os_wait()`` 用于队列操作

- **时间**: `os\_wait()` 用于延时

Call Flow

17 HRTOS 调用流程

17.1 模块介绍

调用流程描述了 HRTOS 中的执行路径，展示了函数在各种操作期间如何相互调用。理解调用流程对于调试、性能分析和系统理解至关重要。

17.2 主要职责

调用流程文档涵盖：

- 任务创建和删除流程
- 调度和上下文切换流程
- IPC 操作流程
- 中断处理流程
- 定时器和延时流程

17.3 主要文件

17.3.1 关键流程文件

- `Src/kernel/task\_create.c`：任务创建流程
- `Src/kernel/task\_cleanup.c`：任务清理流程
- `Src/wait/os\_wait.c`：等待流程
- `Src/wait/wake\_task.c`：唤醒流程
- `Src/interrupt/schedule\_request.c`：调度触发流程
- `Src/kernel/scheduler.c`：调度器选择流程

17.4 任务创建流程

17.4.1 完整流程图

```
sequenceDiagram
    participant App as 应用程序
    participant TC as task create
    participant BM as 位图管理器
```

```
participant SP as 栈指针
participant TCB as TCB 设置
participant Sched as 调度器

App->>TC: os_task_create(addr, id, pr, sd)
TC->>TC: 验证参数
TC->>TC: 检查优先级范围
TC->>TC: 检查重复
TC->>BM: 搜索位图中的空闲块
BM->>BM: 查找连续空闲块
BM->>TC: 返回块起始
TC->>TC: 计算初始 SP
TC->>TC: 检查栈溢出
TC->>SP: 将任务地址压入栈
TC->>TCB: 设置 OS_PROCESS_OK
TC->>TCB: 设置 OS_JINCHENG_SP
TC->>TCB: 设置 OS_JINCHENG_JILU_SP
TC->>Sched: 设置就绪标志
TC-->>App: 返回成功
```

17.4.2 详细步骤

1. ****参数验证****
 - 检查优先级范围 (0-10)
 - 检查任务 ID 有效性
 - 检查地址不为 NULL
 - 检查重复注册
2. ****栈分配****
 - 搜索位图中的空闲块
 - 查找匹配大小的连续块
 - 在位图中标记块为已使用
 - 计算初始栈指针
3. ****栈初始化****
 - 保存当前 SP
 - 将 SP 设置为任务栈
 - 压入任务地址 (低字节)
 - 压入任务地址 (高字节)
 - 恢复 SP
4. ****TCB 设置****
 - 在 `OS\_PROCESS\_OK` 中设置优先级
 - 设置就绪标志
 - 记录栈指针
 - 记录初始栈指针
 - 对于快速任务: 设置上下文和标志

5. **完成**

- 返回成功 (1)
- 任务现在准备好调度

17.5 任务删除流程

17.5.1 完整流程图

```
sequenceDiagram
    participant App as 应用程序
    participant TD as task delete
    participant Idle as 空闲任务
    participant TC as task cleanup
    participant BM as 位图管理器
    participant Res as 资源管理器
    participant Sched as 调度器

    App->>TD: os_task_delete(id)
    TD->>TD: 设置删除请求标志
    TD->>Sched: 触发调度
    Sched->>Idle: 上下文切换到空闲
    Idle->>Idle: 检查删除请求
    Idle->>TC: os_task_cleanup(id)
    TC->>TC: 验证任务 ID
    TC->>TC: 计算栈块
    TC->>BM: 释放位图块
    TC->>TC: 清除上下文保存区域
    TC->>TC: 清除 OS_PROCESS_OK
    TC->>Res: 从等待队列中移除
    TC->>Res: 释放互斥锁所有权
    TC->>TC: 清除 TCB 等待字段
    TC->>TC: 设置状态为 DEAD
    TC-->>Idle: 返回成功
    Idle->>Idle: 继续空闲循环
```

17.5.2 详细步骤

1. **删除请求**
 - 设置 `OS\_EVENT\_DELETE\_REQ` 标志
 - 触发调度
 - 任务继续直到被抢占
2. **空闲任务检测**
 - 空闲任务扫描删除请求
 - 为每个请求调用 `os\_task\_cleanup()`
3. **栈释放**
 - 计算栈块位置
 - 清除位图中的位
 - 释放栈空间
4. **上下文清理**

- 清除上下文保存区域
- 清除栈指针
- 清除任务状态
- 5. ****资源清理****
 - 从所有等待队列中移除
 - 释放互斥锁所有权
 - 清除挂起操作
- 6. ****TCB 清理****
 - 清除 wait_type
 - 清除 wait_flag
 - 清除 wait_obj
 - 清除 wait_tick
 - 设置状态为 DEAD

17.6 调度流程

17.6.1 完整流程图

```
sequenceDiagram
    participant Task as 运行任务
    participant Wait as os_wait
    participant TCB as TCB 更新
    participant Res as 资源队列
    participant ISR as Timer0 ISR
    participant Sched as 调度器
    participant Next as 下一个任务

    Task->>Wait: os_wait(type, obj, tick)
    Wait->>Wait: 进入临界区
    Wait->>TCB: 设置 wait_type、wait_obj、wait_tick
    Wait->>TCB: 设置状态为 WAIT
    Wait->>Res: 添加到 wait_mask (如适用)
    Wait->>TCB: 清除就绪标志
    Wait->>ISR: 设置 OS_SCHED_REASON=1、TF0=1
    Wait->>Wait: 退出临界区
    Wait-->>Task: 返回 (任务阻塞)

    Note over Task,ISR: 发生上下文切换

    ISR->>ISR: Timer0 溢出
    ISR->>ISR: 检查 OS_SCHED_REASON
    ISR->>Sched: 调用调度器
    Sched->>Sched: 扫描就绪任务
    Sched->>Sched: 选择最高优先级
    Sched->>Next: 上下文切换到下一个任务
    Next->>Next: 执行任务代码
```

17.6.2 详细步骤

1. ****阻塞操作****
 - 任务调用阻塞 API
 - API 调用 `os\_wait()`
 - 进入临界区
2. ****等待设置****
 - 设置 TCB 等待字段
 - 添加到资源等待队列
 - 设置任务状态为 WAIT
 - 清除就绪标志
3. ****调度触发****
 - 设置 `OS\_SCHED\_REASON = 1`
 - 设置 `TF0 = 1` (触发 Timer0)
 - 退出临界区
4. ****上下文切换****
 - Timer0 ISR 触发
 - 调度器扫描任务
 - 选择最高优先级的就绪任务
 - 发生上下文切换
5. ****任务执行****
 - 下一个任务运行
 - 前一个任务阻塞

17.7 唤醒流程

17.7.1 完整流程图

```
sequenceDiagram
    participant ISR as ISR/Post
    participant Res as 资源
    participant Wake as wake_task
    participant TCB as TCB 更新
    participant Sched as 调度器
    participant Task as 阻塞任务

    ISR->>Res: Post 操作 (sem_post、mutex_unlock 等)
    Res->>Res: 更新资源状态
    Res->>Wake: wake_task(tid, flag)
    Wake->>Res: 从 wait_mask 中移除
    Wake->>Res: 递减 wait_cnt
    Wake->>TCB: 设置 wait_flag
    Wake->>TCB: 清除 wait_type
    Wake->>TCB: 清除 wait_obj
    Wake->>TCB: 清除 wait_tick
    Wake->>TCB: 设置状态为 READY
```

```
Wake->>TCB: 设置就绪标志
Wake->>Sched: 设置 OS_SCHED_REASON=1、TF0=1
Wake-->>Res: 返回
```

Note over Res,Task: 发生上下文切换

```
Sched->>Task: 任务被选中
Task->>Task: 恢复执行
Task->>Task: 检查 wait_flag
Task->>Task: 从 os_wait 返回
```

17.7.2 详细步骤

1. ****资源操作****
 - ISR 或任务执行 post 操作
 - 资源状态更新
 - 识别等待任务
2. ****唤醒任务调用****
 - 调用 `wake\_task(tid, flag)`
 - 临界区保护
3. ****队列清理****
 - 从 wait_mask 中移除任务
 - 递减 wait_cnt
 - 清除资源关联
4. ****TCB 更新****
 - 设置 wait_flag (TIMEOUT 或 SIGNAL)
 - 清除 wait_type
 - 清除 wait_obj
 - 清除 wait_tick
 - 设置状态为 READY
 - 在 `OS\_PROCESS\_OK` 中设置就绪标志
5. ****调度触发****
 - 设置 `OS\_SCHED\_REASON = 1`
 - 设置 `TF0 = 1`
 - 任务将被调度

17.8 中断处理流程

17.8.1 完整流程图

```
sequenceDiagram
    participant HW as 硬件
    participant ISR as ISR 处理程序
```

```

participant API as ISR 安全 API
participant Res as 资源
participant Sched as 调度器
participant Task as 等待任务

HW->>ISR: 发生中断
ISR->>ISR: 保存上下文
ISR->>API: 调用 ISR 安全 API (event_set_from_isr 等)
API->>Res: 更新资源值
API->>Res: 递增 pending_signal
API->>Sched: os_schedule_request()
Sched->>Sched: 设置 OS_SCHED_REASON=1
Sched->>Sched: 设置 TF0=1
API-->>ISR: 返回
ISR->>ISR: 恢复上下文
ISR-->>HW: 返回

Note over HW,Task: Timer0 中断触发

Sched->>Task: 唤醒任务
Task->>Task: 处理事件

```

17.8.2 详细步骤

1. ****中断入口****
 - 发生硬件中断
 - 硬件保存上下文
 - ISR 处理程序执行
2. ****ISR 安全 API 调用****
 - 事件设置、信号量释放或消息发送
 - 资源值更新
 - 挂起信号递增
3. ****调度请求****
 - 调用 `os\_schedule\_request()`
 - 设置调度触发
 - 触发 Timer0 中断
4. ****ISR 退出****
 - 上下文恢复
 - 从中断返回
5. ****延迟调度****
 - Timer0 中断触发
 - 调度器处理唤醒
 - 任务恢复

17.9 延时流程

17.9.1 完整流程图

```
sequenceDiagram
    participant Task as 任务
    participant Delay as os_delay
    participant Wait as os_wait
    participant TCB as TCB 更新
    participant Timer as Timer0 ISR
    participant Tick as 时钟处理程序
    participant Wake as wake_task

    Task->>Delay: os_delay(tick)
    Delay->>Wait: os_wait(WAIT_DELAY, INVALID_ID, tick)
    Wait->>Wait: 进入临界区
    Wait->>TCB: 设置 wait_type=WAIT_DELAY
    Wait->>TCB: 设置 wait_tick=tick
    Wait->>TCB: 设置状态=WAIT
    Wait->>TCB: 清除就绪标志
    Wait->>Timer: 设置 OS_SCHED_REASON=1、TF0=1
    Wait->>Wait: 退出临界区
    Wait-->>Delay: 返回
    Delay-->>Task: 返回 (任务阻塞)

    Note over Task,Timer: 任务阻塞，定时器中断

    Timer->>Tick: 时钟周期发生
    Tick->>TCB: 递减 wait_tick
    Tick->>Tick: wait_tick == 0?
    Tick->>Wake: 是，调用 wake_task
    Wake->>TCB: 设置状态=READY
    Wake->>TCB: 设置就绪标志
    Wake->>Timer: 设置 OS_SCHED_REASON=1
    Wake-->>Tick: 返回
    Tick-->>Timer: 返回

    Note over Timer,Task: 调度器选择任务

    Task->>Task: 恢复执行
```

17.9.2 详细步骤

1. ****延时调用****
 - 任务调用 `os\_delay(tick)`
 - 调用 `os\_wait(WAIT\_DELAY, INVALID\_ID, tick)`
2. ****等待设置****
 - 设置 wait_type 为 WAIT_DELAY
 - 设置 wait_tick 为 tick 值
 - 无资源对象 (INVALID_ID)
 - 设置状态为 WAIT
 - 清除就绪标志
3. ****调度触发****
 - 设置调度触发

- 任务阻塞
- 4. ****时钟周期处理****
 - 每次 Timer0 中断
 - 递减 wait_tick
 - 检查 wait_tick == 0
- 5. ****唤醒****
 - 当 wait_tick 达到 0
 - 使用 WAIT_TIMEOUT 调用 `wake\_task()`
 - 任务返回到 READY 状态
- 6. ****任务恢复****
 - 调度器选择任务
 - 任务恢复执行
 - 从 `os\_delay()` 返回

17.10 互斥锁加锁流程

17.10.1 完整流程图

```
sequenceDiagram
    participant Task as 任务
    participant Lock as mutex_lock
    participant Res as 资源
    participant Prio as 优先级继承
    participant Wait as os_wait
    participant TCB as TCB_更新

    Task->>Lock: os_mutex_lock(mid)
    Lock->>Lock: 验证参数
    Lock->>Res: 检查所有者
    Res-->>Lock: 所有者状态
    Lock->>Lock: 当前任务所有者?
    Lock-->>Task: 是, 返回错误 (无递归)

    alt 互斥锁空闲
        Lock->>Res: 设置所有者 = 当前任务
        Lock->>TCB: 设置 base_prio
        Lock->>TCB: 设置 cur_prio
        Lock-->>Task: 返回成功
    else 互斥锁被持有
        Lock->>Prio: 检查优先级
        Prio->>TCB: 获取当前任务优先级
        Prio->>TCB: 获取所有者优先级
        alt 等待任务优先级更高
            Prio->>TCB: 提升所有者 cur_prio
            Prio->>TCB: 更新 OS_PROCESS_OK
        end
        Lock->>Wait: os_wait(WAIT_MUTEX, mid, 0)
        Wait->>TCB: 设置 wait_type=WAIT_MUTEX
        Wait->>Res: 添加到 wait_mask
        Wait->>TCB: 设置状态=WAIT
```

```
Lock-->>Task: 返回（进入等待队列）
end
```

17.10.2 详细步骤

1. ****加锁请求****
 - 任务调用 `os\_mutex\_lock(mid)`
 - 验证互斥锁 ID
2. ****递归检查****
 - 检查当前任务是否为所有者
 - 如果是，返回错误（无递归锁）
3. ****空闲互斥锁****
 - 如果互斥锁空闲（所有者 = INVALID_ID）
 - 将所有者设置为当前任务
 - 设置任务优先级
 - 立即返回成功
4. ****被持有的互斥锁****
 - 如果互斥锁被另一个任务持有
 - 检查优先级继承
 - 如果等待任务优先级更高：
 - 提升所有者优先级
 - 更新所有者的 `OS\_PROCESS\_OK`
 - 调用 `os\_wait(WAIT\_MUTEX, mid, 0)`
 - 添加到等待队列
 - 返回（任务阻塞）

17.11 信号量等待流程

17.11.1 完整流程图

```
sequenceDiagram
    participant Task as 任务
    participant Wait as sem_wait
    participant Res as 资源
    participant WaitCore as os_wait
    participant TCB as TCB 更新

    Task->>Wait: os_sem_wait(sid)
    Wait->>Res: 检查值
    Res-->>Wait: 值状态

    alt value > 0 (快速路径)
        Wait->>Res: 递减值
        Wait-->>Task: 返回 WAIT_SIGNAL
    end
```

```

else value == 0 (慢速路径)
    Wait->>WaitCore: os_wait(WAIT_SEM, sid, 0)
    WaitCore->>TCB: 设置 wait_type=WAIT_SEM
    WaitCore->>Res: 添加到 wait_mask
    WaitCore->>TCB: 设置状态=WAIT
    WaitCore->>TCB: 清除就绪标志
    WaitCore-->>Wait: 返回 (阻塞)
    Wait-->>Task: 返回任务 ID
end

```

17.11.2 详细步骤

1. ****信号量等待****
 - 任务调用 `os\_sem\_wait(sid)`
 - 进入临界区
2. ****快速路径检查****
 - 检查 `value > 0`
 - 如果是:
 - 递减值
 - 立即返回 `WAIT_SIGNAL`
 - 无阻塞, 无上下文切换
3. ****慢速路径****
 - 如果 `value == 0`
 - 调用 `os\_wait(WAIT\_SEM, sid, 0)`
 - 设置 `wait_type`
 - 添加到 `wait_mask`
 - 设置状态为 `WAIT`
 - 任务阻塞
4. ****唤醒****
 - 当调用 `os\_sem\_post()` 时
 - 使用 `WAIT_SIGNAL` 唤醒任务
 - 从 `os\_sem\_wait()` 返回

17.12 消息队列发送流程

17.12.1 完整流程图

```

sequenceDiagram
    participant Task as 任务
    participant Send as msgq_send
    participant Queue as 队列结构
    participant Wait as os_wait
    participant Wake as wake_task

```

```

Task->>Send: os_msgq_send(q, obj, data, nonblock)
Send->>Queue: 检查计数与大小
Queue-->>Send: 队列状态

alt 队列未满
    Send->>Queue: 在 head 写入数据
    Send->>Queue: 递增 head
    Send->>Queue: 递增 count
    Send->>Queue: 接收者等待?
    alt 是
        Send->>Wake: wake_task(receiver_id)
        Wake->>Queue: 从 wait_mask 中移除
        Wake->>Queue: 设置状态=READY
    end
    Send-->>Task: 返回成功
else 队列已满
    alt nonblock == 1
        Send-->>Task: 返回失败
    else nonblock == 0
        Send->>Wait: os_wait(WAIT_MSG_SEND, obj, 0)
        Wait->>Queue: 添加到 wait_mask
        Wait->>Queue: 设置状态=WAIT
        Send-->>Task: 返回 (阻塞)
    end
end
end

```

17.12.2 详细步骤

1. ****发送请求****
 - 任务调用 `os\_msgq\_send()`
 - 检查队列状态
2. ****队列未满****
 - 在 head 处将数据写入缓冲区
 - 递增 head (带环绕)
 - 递增 count
 - 检查接收者是否等待
 - 如果是, 唤醒接收者
 - 返回成功
3. ****队列已满****
 - 检查 nonblock 标志
 - 如果非阻塞: 返回失败
 - 如果阻塞:
 - 调用 `os\_wait(WAIT\_MSG\_SEND, obj, 0)`
 - 添加到发送者等待队列
 - 任务阻塞
4. ****唤醒****
 - 当接收者消费消息时

- 空间变为可用
- 发送者被唤醒
- 发送操作完成

17.13 调用流程摘要

17.13.1 常见模式

1. ****阻塞操作****: 所有阻塞操作通过 ``os_wait()``
2. ****唤醒****: 所有唤醒通过 ``wake_task()``
3. ****调度****: 通过 Timer0 中断触发调度
4. ****临界区****: 所有状态更改由临界区保护

17.13.2 性能考虑

- ****快速路径****: 许多操作具有快速路径优化
- ****上下文切换****: 阻塞操作导致上下文切换
- ****临界区****: 临界区中的时间最小
- ****ISR 安全****: ISR 操作延迟到任务上下文

17.13.3 调试调用流程

- ****跟踪函数调用****: 在关键函数处添加断点
- ****监视状态****: 观察 TCB 和资源状态更改
- ****检查触发****: 验证调度触发
- ****验证转换****: 确保状态转换正确

18 HRTOS 数据结构

18.1 模块介绍

HRTOS 使用一组精心设计的数据结构来管理任务、IPC 资源、定时和系统状态。本文档提供了内核中使用的所有主要数据结构的综合参考。

18.2 主要职责

数据结构处理：

- 任务控制和状态
- IPC 资源管理
- 消息队列缓冲
- 内存分配跟踪
- 定时和时钟周期管理
- 中断上下文保存

18.3 数据结构参考

18.3.1 OS_TCB（任务控制块）

****位置**：** `Inc/config.h`

****目的**：** 存储有关任务的所有信息

****定义**：**

```
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;          /* 等待结果标志 */  
    u8 wait_obj;           /* 等待对象 ID */  
    u16 wait_tick;         /* 超时时钟周期 */  
} OS_TCB;
```

****字段描述**：**

- `id`：唯一任务标识符（0-17）
- `base\_prio`：创建时分配的原始优先级

- `cur\_prio`: 当前优先级（可能被优先级继承提升）
- `state`: 当前任务状态（READY、WAIT、RUNNING、SUSPEND、DEAD）
- `wait\_type`: 等待类型（WAIT_DELAY、WAIT_SEM 等）
- `wait\_flag`: 唤醒原因（WAIT_TIMEOUT、WAIT_SIGNAL）
- `wait\_obj`: 正在等待的资源 ID
- `wait\_tick`: 超时前的剩余时钟周期
- **数组****: XDATA 中的 `OS\_TASK[OS\_PROCESS\_MAX]`
- **大小****: 每个 TCB 9 字节
- **总大小****: 144 字节（16 个任务 × 9 字节）

18.3.2 OS_RESOURCE（统一 IPC 资源）

- **位置****: `Inc/config.h`
- **目的****: 所有 IPC 类型（信号量、互斥锁、事件、邮箱）的统一结构
- **定义****:

```
typedef struct {
    u8 value;           /* 资源值 */
    u8 owner;           /* 当前所有者（仅互斥锁） */
    u8 wait_cnt;        /* 等待任务计数 */
    u16 wait_mask;      /* 位图等待队列 */
    u8 pending_signal;  /* ISR 信号计数 */
} OS_RESOURCE;
```

- **字段描述****:
 - `value`: 资源特定值
 - 信号量: 可用资源计数
 - 事件: 事件标志（0/1）
 - 互斥锁: 未使用
 - 邮箱: 邮箱值
 - `owner`: 持有互斥锁的任务 ID（如果空闲则为 OS_INVALID_ID）
 - `wait\_cnt`: 等待此资源的任务数
 - `wait\_mask`: 等待任务的位图（位 0 = 任务 0，位 1 = 任务 1 等）
 - `pending\_signal`: ISR 信号计数（用于 ISR 安全操作）
- **数组****: XDATA 中的 `OS\_RES[OS\_RESOURCE\_MAX]`
- **大小****: 每个资源 6 字节
- **总大小****: 48 字节（8 个资源 × 6 字节）

18.3.3 os_msgq_t（消息队列）

- **位置****: `Inc/config.h`
- **目的****: 消息队列的循环缓冲区
- **定义****:

```
typedef struct {
    u8 *buf;           /* 缓冲区指针 */
    u8 _size;          /* 缓冲区大小 */
    u8 head;           /* 写指针 (生产者) */
    u8 tail;           /* 读指针 (消费者) */
    u8 count;          /* 当前消息计数 */
} os_msgq_t;
```

****字段描述**:**

- `buf`: 指向用户提供的缓冲区的指针

- `\_size`: 缓冲区容量

- `head`: 下一条消息将写入的索引

- `tail`: 下一条消息将读取的索引

- `count`: 队列中当前的消息数

****数组**:** XDATA 中的 `OS\_MSGQ[OS\_MSGQ\_MAX]`

****大小**:** 每个队列 5 字节

****总大小**:** 20 字节 (4 个队列 × 5 字节)

****注意**:** 实际消息缓冲区由用户提供, 不计入内核内存

18.3.4 OS_PROCESS_OK (任务状态数组)

****位置**:** `Src/kernel/os\_core.c`

****目的**:** 编码任务状态、优先级和栈大小

****定义**:**

```
volatile unsigned char xdata OS_PROCESS_OK[OS_PROCESS_MAX];
```

****编码**:**

- 位 0: 就绪标志 (1 = 准备运行, 0 = 未就绪)

- 位 1-3: 优先级级别 (0-7)

- 位 4-7: 栈大小 (层)

****示例**:**

- `0x01`: 就绪, 优先级 0, 栈大小 0

- `0x1F`: 就绪, 优先级 7, 栈大小 1

- `0x00`: 未就绪 (已删除或挂起)

****大小**:** 每个任务 1 字节

****总大小**:** 16 字节

18.3.5 栈指针

****位置**:** `Src/kernel/os\_core.c`

****目的**:** 跟踪每个任务的栈指针

18.3.5.1 OS_JINCHENG_SP (标准任务栈指针)

```
volatile unsigned char xdata OS_JINCHENG_SP[OS_PROCESS_MAX];
```

****目的**:** 每个标准任务的当前栈指针

****大小****: 每个任务 1 字节

****总大小****: 16 字节

18.3.5.2 OS_JINCHENG_JILU_SP (初始栈指针)

```
volatile unsigned char xdata OS_JINCHENG_JILU_SP[OS_PROCESS_MAX];
```

****目的****: 用于清理的初始栈指针

****大小****: 每个任务 1 字节

****总大小****: 16 字节

18.3.5.3 OS_KUAI_SP (快速任务栈指针)

```
volatile unsigned char xdata OS_KUAI_SP[OS_FAST_TASK_MAX];
```

****目的****: 快速任务的当前栈指针

****大小****: 每个快速任务 1 字节

****总大小****: 2 字节

18.3.6 上下文保存区域

****位置****: `Src/kernel/os\_core.c`

****目的****: 上下文切换期间保存任务上下文

18.3.6.1 OS_TASK_CONTEXT (标准任务上下文)

```
volatile unsigned char xdata OS_TASK_CONTEXT[OS_PROCESS_MAX]  
[OS_JINCHENG_SHENDU];
```

****大小****: 每个任务 13 字节

****内容****: PWS、ACC、B、DPTR、寄存器组

****总大小****: 208 字节 (16 个任务 × 13 字节)

18.3.6.2 OS_FAST_CONTEXT (快速任务上下文)

```
volatile unsigned char xdata OS_FAST_CONTEXT[OS_FAST_TASK_MAX]  
[OS_KUAI_SHENDU];
```

****大小****: 每个快速任务 5 字节

****内容****: PWS、ACC、B、DPTR

****总大小****: 10 字节 (2 个任务 × 5 字节)

18.3.7 内存位图

****位置****: `Src/kernel/os\_core.c`

****目的****: 栈分配的位图

****定义****:

```
volatile unsigned char xdata OS_MEMORY[OS_MEMORY_ADMINISTRATION];
```

****大小****: 4 字节 (32 位)

****用途****:

- 每位代表一个 2 字节栈块

- 位 1 = 块已分配

- 位 0 = 块空闲

- 支持最多 31 个块（总共 62 字节）

18.3.8 定时变量

****位置****: `Src/kernel/os\_core.c` 和 `Inc/hrtos\_internal.h`

18.3.8.1 OS_TIME_XY（时间片计数器）

```
volatile unsigned char idata OS_TIME_XY;
```

****目的****: 当前时间片计数器

****大小****: 1 字节

18.3.8.2 OS_TIME_ONCE（时间片计数）

```
volatile unsigned char xdata OS_TIME_ONCE;
```

****目的****: 每个调度周期的时间片数

****大小****: 1 字节

18.3.8.3 OS_TICK_COUNT_L/H（时钟周期计数器）

```
volatile unsigned int xdata OS_TICK_COUNT_L;
```

```
volatile unsigned int xdata OS_TICK_COUNT_H;
```

****目的****: 32 位系统时钟周期计数器

****大小****: 总共 4 字节（2 × 2 字节）

18.3.8.4 OS_WAIT_DI2（快速任务延时）

```
volatile unsigned int xdata OS_WAIT_DI2[OS_FAST_TASK_MAX];
```

****目的****: 快速任务的延時計數器

****大小****: 总共 4 字节（2 × 2 字节）

18.3.9 中断变量

****位置****: `Src/kernel/os\_core.c`

18.3.9.1 OS_INTERRUPT_ADDR（中断处理程序地址）

```
volatile unsigned char xdata OS_INTERRUPT_ADDR[2];
```

****目的****: 中断处理程序地址（优先级 9）

****大小****: 2 字节

18.3.9.2 OS_INTERRUPT_ADDR2（嵌套中断处理程序地址）

```
volatile unsigned char xdata OS_INTERRUPT_ADDR2[2];
```

****目的****: 嵌套中断处理程序地址（优先级 10）

****大小****: 2 字节

18.3.9.3 OS_INT_BAO（中断上下文保存）

```
volatile unsigned char idata OS_INT_BAO[6];
```

****目的****: 中断期间保存 ACC、B、DPH、DPL、PSW、EA

****大小****: 6 字节

18.3.10 临界区变量

****位置****: `Inc/hrtos\_internal.h`

18.3.10.1 os_critical_nesting（嵌套计数器）

```
volatile unsigned char idata os_critical_nesting;
```

****目的****: 嵌套临界区的计数

****大小****: 1 字节

18.3.10.2 os_int_status (保存的中断状态)

```
volatile bit os_int_status;
```

****目的****: 临界区入口时保存的 EA 状态

****大小****: 1 位

18.3.11 调度器变量

****位置****: `Src/kernel/os\_core.c`

18.3.11.1 OS_CURRENT_TASK (当前任务 ID)

```
volatile unsigned char idata OS_CURRENT_TASK;
```

****目的****: 当前运行任务的 ID

****大小****: 1 字节

18.3.11.2 OS_PREV_TASK (前一个任务 ID)

```
volatile unsigned char idata OS_PREV_TASK;
```

****目的****: 前一个运行任务的 ID

****大小****: 1 字节

18.3.11.3 OS_DISPATCH_ID (分发触发)

```
volatile unsigned char idata OS_DISPATCH_ID;
```

****目的****: 分发触发的来源

****大小****: 1 字节

18.3.11.4 OS_SCHED_REASON (调度触发)

```
volatile bit OS_SCHED_REASON;
```

****目的****: 指示需要调度的标志

****大小****: 1 位

18.3.12 快速任务控制

****位置****: `Src/kernel/os\_core.c`

18.3.12.1 OS_KUAI_PROCESS_A/B (快速任务标志)

```
volatile bit OS_KUAI_PROCESS_A;
```

```
volatile bit OS_KUAI_PROCESS_B;
```

****目的****: 快速任务的激活标志

****大小****: 2 位

18.3.12.2 OS_SP_KUAI_BEI (快速任务 SP 起始)

```
volatile unsigned char xdata OS_SP_KUAI_BEI[OS_FAST_TASK_MAX];
```

****目的****: 快速任务的初始栈指针

****大小****: 2 字节

18.3.12.3 OS_SP_KUAI_BEI2 (快速任务栈大小)

```
volatile unsigned char xdata OS_SP_KUAI_BEI2[OS_FAST_TASK_MAX];
```

****目的****: 快速任务的栈大小

****大小****: 2 字节

18.3.13 事件变量

****位置****: `Inc/hrtos\_internal.h`

18.3.13.1 OS_EVENT_BIT (事件位图)

```
volatile unsigned int xdata OS_EVENT_BIT;
```

****目的****: 事件位操作

****大小****: 2 字节

18.3.13.2 OS_TASK_EVENT (任务事件标志)

```
volatile unsigned char xdata OS_TASK_EVENT[OS_PROCESS_MAX + OS_FAST_TASK_MAX];
```

****目的****: 用于任务删除的事件标志

****大小****: 18 字节

18.3.14 栈区域

****位置****: `Src/kernel/os\_core.c`

18.3.14.1 OS_HRTOS_DUIZHAN_OS (系统栈)

```
unsigned char data OS_HRTOS_DUIZHAN_OS[OS_SP_KER] _at_ 0x35;
```

****目的****: 内核操作的系统栈

****大小****: 5 字节

****地址****: 0x35-0x39

18.3.14.2 OS_HRTOS_DUIZHAN_INT (中断栈)

```
unsigned char data OS_HRTOS_DUIZHAN_INT[8] _at_ 0x3a;
```

****目的****: 中断处理程序的栈

****大小****: 8 字节

****地址****: 0x3A-0x41

18.3.14.3 OS_HRTOS_DUIZHAN_APP (用户栈)

```
unsigned char data OS_HRTOS_DUIZHAN_APP[62] _at_ 0x42;
```

****目的****: 用户任务的栈区域

****大小****: 62 字节

****地址****: 0x42-0x7F

18.3.15 内核临时变量

****位置****: `Src/kernel/os\_core.c`

18.3.15.1 DATA 段变量

```
volatile unsigned char data OS_INTERRUPT_PROTECT _at_ 0x30;  
volatile unsigned char data OS_R0_PROTECT _at_ 0x31;  
volatile unsigned char data os_sp _at_ 0x32;  
volatile unsigned char data OS_HRTOS _at_ 0x33;  
volatile unsigned char data OS_HRTOS_2 _at_ 0x34;
```

****目的****: 内核操作的临时变量

****大小****: 5 字节

****地址****: 0x30-0x34

18.3.16 系统标志

****位置****: `Src/kernel/os\_core.c`

```
volatile bit OS_KUAI_PROCESS_A;  
volatile bit OS_KUAI_PROCESS_B;  
volatile bit OS_JINCHENG_CUNCHU;  
volatile bit OS_SCHED_REASON;  
volatile bit OS_ISR_FLAG;  
volatile bit OS_INTERRUPT_T0;  
volatile bit OS_RTOS_YES;  
volatile bit OS_a;  
volatile bit OS_b;  
volatile bit OS_INTERRUPT_GAO;  
volatile bit OS_INTERRUPT_GAO_BIAO;  
volatile bit os_int_status;  
volatile bit OS_CFG_LOCK;
```

****目的****: 各种系统标志和状态指示器

****大小****: 12 位

18.4 内存使用摘要

18.4.1 DATA 段 (0x20-0x7F)

结构	大小	地址
内核临时变量	5 字节	0x30-0x34
系统栈	5 字节	0x35-0x39
中断栈	8 字节	0x3A-0x41
用户栈	62 字节	0x42-0x7F
总计	**80 字节**	

18.4.2 IDATA 段 (0xE0-0xE7)

结构	大小	地址
OS_CURRENT_TASK	1 字节	0xE3
OS_PREV_TASK	1 字节	0xE4
OS_DISPATCH_ID	1 字节	0xE5
OS_TIME_XY	1 字节	0xE6
os_critical_nesting	1 字节	0xE7
OS_T0_TH0	1 字节	0xE2

OS_T0_TL0 1 字节 0xE1
OS_INT_BAO 6 字节 0xDB-0xE0
总计 **13 字节**

18.4.3 XDATA 段 (0x0200-0x03FF)

结构 大小 偏移
----- ----- -----
OS_PROCESS_OK 16 字节 0x0000
OS_JINCHENG_SP 16 字节 0x0010
OS_KUAI_SP 2 字节 0x0020
OS_FAST_CONTEXT 10 字节 0x0030
OS_TASK_CONTEXT 208 字节 0x0040
OS_MEMORY 4 字节 0x0110
OS_JINCHENG_JILU_SP 16 字节 0x0120
OS_TIME_ONCE 1 字节 0x0130
OS_WAIT_DI2 4 字节 0x0131
OS_INTERRUPT_ADDR 2 字节 0x0135
OS_INTERRUPT_ADDR2 2 字节 0x0137
OS_TICK_COUNT_L 2 字节 0x0139
OS_TICK_COUNT_H 2 字节 0x013B
OS_EVENT_BIT 2 字节 0x013D
OS_INSIDE_STACK_USE_MAXIMUM 1 字节 0x013F
OS_SP_KUAI_BEI 2 字节 0x0140
OS_SP_KUAI_BEI2 2 字节 0x0142
OS_TASK_EVENT 18 字节 0x0144
OS_TIME_ONCE_BACKUP 1 字节 0x0156
OS_TASK (TCB 数组) 144 字节 动态
OS_RES (资源数组) 48 字节 动态
OS_MSGQ (队列数组) 20 字节 动态
总计 **~520 字节**

18.5 数据结构设计原则

18.5.1 紧凑设计

- 最小化内存使用
- 适当时使用位字段

- 高效打包数据
- 避免填充

18.5.2 静态分配

- 所有结构静态分配
- 无动态内存
- 可预测的内存使用
- 无碎片

18.5.3 统一模型

- 多种 IPC 类型的单一结构
- 一致的 API 模式
- 简化实现
- 减少代码大小

18.5.4 访问优化

- 频繁访问的变量在 DATA/IDATA
- 大数组在 XDATA
- 关键状态在快速内存
- 最小化 XDATA 访问

18.6 数据结构约束

- 最多 16 个标准任务
- 最多 2 个快速任务
- 最多 8 个 IPC 资源
- 最多 4 个消息队列
- 栈限制为 62 字节
- 无动态分配
- 运行时无数据结构扩展

Design Principles

19 HRTOS 设计原则

19.1 模块介绍

HRTOS 基于特定的架构原则设计，以实现其实时性能、低内存占用和适合 8051 微控制器的目标。本文档阐述了这些原则并解释了它们如何指导内核的设计。

19.2 主要职责

设计原则指导：

- 架构决策
- 内存管理策略
- API 设计模式
- 性能优化
- 移植性考虑

19.3 核心设计原则

19.3.1 极简主义

****原则****：在提供基本 RTOS 功能的同时，保持内核尽可能小和简单。

****实现****：

- 仅包含基本功能
- 无不必要的抽象
- 紧凑的数据结构
- 最小的代码大小

****好处****：

- 适合有限的内存
- 更容易理解
- 更快的执行
- 更低的维护负担

****权衡****：

- 功能集有限
- 灵活性较差
- 配置选项较少

19.3.2 静态分配

****原则****：所有内存在编译时静态分配；无动态内存分配。

****实现****：

- 任务、资源、队列的固定大小数组
- 基于位图的栈分配
- 预分配的上下文保存区域
- 无 malloc/free

****好处****：

- 可预测的内存使用
- 无碎片
- 确定性行为
- 更简单的实现

****权衡****：

- 固定的最大限制
- 如果未充分利用则内存浪费
- 无法适应运行时需求

19.3.3 基于优先级的调度

****原则****：任务基于优先级调度；高优先级任务总是抢占低优先级任务。

****实现****：

- 10 个优先级级别（0-10）
- 抢占式调度
- 相同优先级的可选时间片
- 互斥锁的优先级继承

****好处****：

- 确定性时序
- 实时保证
- 可预测的行为
- 有界的响应时间

****权衡****：

- 可能发生优先级反转（通过继承缓解）
- 需要仔细的优先级分配

- 低优先级任务可能饥饿

19.3.4 统一 IPC 模型

****原则****：所有 IPC 机制使用统一的资源结构和等待机制。

****实现****：

- 单一 `OS\_RESOURCE` 结构
- 统一的 `os\_wait()` 函数
- 一致的 API 模式
- 基于位图的等待队列

****好处****：

- 减少代码大小
- 一致的行为
- 更容易维护
- 简化

****权衡****：

- 每个 IPC 类型的灵活性较低
- 某些 IPC 功能未优化
- 共享限制

19.3.5 8051 优化

****原则****：专门针对 8051 架构及其内存模型进行优化。

****实现****：

- DATA 段用于频繁访问的变量
- IDATA 用于内核状态
- XDATA 用于大型数组
- 关键路径的汇编优化
- 快速任务的寄存器组使用

****好处****：

- 目标平台上的最大性能
- 高效的内存使用
- 最小开销
- 硬件特定优化

****权衡****：

- 可移植性较差
- 绑定到 8051 架构
- 更难移植到其他平台

19.3.6 实时焦点

****原则****：为实时性能设计，具有有界的执行时间。

****实现****：

- 确定性调度
- 有界的上下文切换时间
- 最小的中断延迟
- 无无界操作

****好处****：

- 可预测的时序
- 适合实时应用程序
- 有界的最坏情况执行时间
- 可靠的时序保证

****权衡****：

- 可能牺牲平均情况性能
- 功能复杂度限制
- 需要仔细实现

19.3.7 ISR 安全

****原则****：为中断场景提供专用安全 API，在保证实时性的同时避免 ISR 中直接执行复杂内核操作。

****实现****：

- 提供独立的 ISR 安全函数
- ISR 中禁止阻塞操作
- ISR 中仅执行必要的资源状态更新
- 通过设置调度请求标志，由内核在中断退出阶段统一判断是否调度
- 保持最小 ISR 执行时间

****好处****：

- 安全可靠的中断处理机制
- 避免 ISR 中发生优先级反转
- 调度时机明确，可预测性高
- 简化中断服务程序设计
- 避免 ISR 内直接调用调度器带来的复杂性

****权衡****：

- ISR 操作受限，仅支持非阻塞 API
- ISR 中不能直接执行任务切换
- 被唤醒任务需要等待中断退出阶段完成调度判断

19.3.8 临界区保护

****原则****：使用嵌套中断禁用的临界区保护共享数据。

****实现****：

- ``os_enter_critical()` / `os_exit_critical()``
- 嵌套计数器支持
- 保存/恢复中断状态
- 最小临界区持续时间

****好处****：

- 简单实现
- 有效保护
- 对嵌套调用安全
- 低开销

****权衡****：

- 增加中断延迟
- 非选择性（禁用所有中断）
- 无基于优先级的屏蔽

19.3.9 分层架构

****原则****：将内核组织在具有清晰依赖关系和无循环依赖的层中。

****实现****：

- 底层的移植层
- 移植层之上的等待层
- 等待层之上的功能模块
- 顶层的应用程序
- 清晰的 API 边界

****好处****：

- 可维护的结构
- 容易理解
- 促进移植
- 简化测试

****权衡****：

- 层之间的一些耦合
- 可能增加间接性
- 未严格执行

19.3.10 教学导向

****原则****：设计为可理解且适合教授 RTOS 概念。

****实现****：

- 清晰的 API 命名
- 简单的数据结构
- 记录良好的代码
- 示例程序

****好处****：

- 容易学习
- 良好的教育价值
- 清晰的概念
- 可接近的代码

****权衡****：

- 可能牺牲一些优化
- 比生产 RTOS 更简单
- 功能有限

19.4 内存设计原则

19.4.1 内存分段

****原则****：根据访问模式适当使用 8051 内存段。

****实现****：

- DATA：频繁访问、小变量
- IDATA：内核状态、关键变量
- XDATA：大型数组、较少访问

****理由****：

- DATA 最快但有限
- IDATA 是间接的但仍然快
- XDATA 较慢但丰富

19.4.2 栈管理

****原则****：具有溢出检测的基于位图的栈分配。

****实现****：

- 2 字节块
- 位图跟踪
- 带重置的溢出检测
- 系统、中断、用户的独立栈

****理由**：**

- 高效分配
- 容易释放
- 安全检查
- 清晰分离

19.4.3 上下文保存优化

****原则**：**最小化快速任务的上下文保存大小。

****实现**：**

- 快速任务：5 个寄存器
- 标准任务：13 个寄存器
- 专用上下文区域
- 寄存器组使用

****理由**：**

- 快速任务更快的上下文切换
- 快速任务减少内存
- 速度和完整性之间的权衡

19.5 API 设计原则

19.5.1 一致性

****原则**：**所有 API 遵循一致的命名和行为模式。

****实现**：**

- 所有 API 的 `os\_XXX` 前缀
- 一致的参数排序
- 一致的返回值语义
- 一致的错误处理

****理由**：**

- 容易学习
- 可预测的行为
- 减少认知负担

19.5.2 简单性

****原则**：**保持 API 简单，参数最少。

****实现**：**

- 整数 ID 代替指针

- 小参数计数
 - 简单返回值
 - 无复杂结构
- **理由**：**
- 更容易使用
 - 不易出错
 - 适合 8051

19.5.3 阻塞与非阻塞

****原则**：** 在适当的地方提供阻塞和非阻塞模式。

****实现**：**

- 队列的 `nonblock` 参数
- 等待的超时支持
- 非阻塞的立即返回
- 清晰的文档

****理由**：**

- 不同用例的灵活性
- 避免死锁
- 响应式设计

19.6 性能原则

19.6.1 快速路径优化

****原则**：** 通过避免开销的快速路径优化常见情况。

****实现**：**

- 信号量快速路径（值 > 0）
- 事件快速路径（已设置）
- 尽可能立即返回
- 最小临界区时间

****理由**：**

- 常见情况快
- 减少开销
- 更好的平均性能
- 维持实时保证

19.6.2 最小上下文切换

****原则****：通过精心设计最小化上下文切换开销。

****实现****：

- 小上下文保存区域
- 快速任务优化
- 高效的寄存器使用
- 关键路径的汇编

****理由****：

- 更快的调度
- 更低开销
- 更好的实时响应
- 减少 CPU 浪费

19.6.3 确定性执行

****原则****：确保所有操作的有界执行时间。

****实现****：

- 无无界循环
- 固定大小的数据结构
- 无动态分配
- 有界等待队列

****理由****：

- 可预测的时序
- 实时保证
- 更容易分析
- 可靠的行为

19.7 可移植性原则

19.7.1 硬件抽象

****原则****：在移植层中隔离硬件特定代码。

****实现****：

- 用于硬件操作的移植模块
- 寄存器定义分离
- 可配置的内存布局
- 定时器抽象

****理由****：

- 更容易移植

- 清晰的硬件边界
- 平台特定优化
- 可维护代码

19.7.2 基于配置

****原则****：使用编译时配置处理平台差异。

****实现****：

- 配置常量
- 条件编译
- 内存布局参数
- 定时器配置

****理由****：

- 无运行时开销
- 编译时优化
- 清晰配置
- 容易适应

19.8 安全原则

19.8.1 栈溢出保护

****原则****：检测并防止栈溢出。

****实现****：

- 栈使用跟踪
- 溢出检测
- 严重溢出时系统重置
- 警告阈值

****理由****：

- 防止内存损坏
- 系统稳定性
- 调试辅助
- 安全机制

19.8.2 参数验证

****原则****：验证内核函数的所有参数。

****实现****：

- 范围检查

- 空指针检查
 - 重复注册检查
 - 状态验证
- **理由**:**
- 防止崩溃
 - 早期检测错误
 - 系统稳定性
 - 调试辅助

19.9 权衡摘要

原则	好处	权衡
-----	-----	-----
极简主义	小尺寸、简单	功能有限
静态分配	可预测、无碎片	固定限制、浪费
基于优先级	确定性、实时	优先级反转、饥饿
统一 IPC	小代码、一致	灵活性较低
8051 优化	最大性能	可移植性较差
实时焦点	有界时序	可能牺牲平均性能
ISR 安全	安全中断	ISR 操作受限
临界区	简单保护	增加延迟
分层架构	可维护	一些耦合
教学导向	容易学习	比生产简单

19.10 设计反模式（避免）

19.10.1 反模式：动态内存分配

- **避免原因**:**
- 8051 上的碎片
 - 不可预测的时序
 - 复杂实现
 - 不适合实时

19.10.2 反模式：复杂数据结构

- **避免原因**:**
- 内存太多
 - 复杂代码

- 难以调试
- 8051 上慢

19.10.3 反模式：无限队列

****避免原因**：**

- 无界内存
- 不可预测的时序
- 复杂管理
- 不适合实时

19.10.4 反模式：轮询调度

****避免原因**：**

- 确定性较差
- 难以保证时序
- 不适合硬实时

19.10.5 反模式：递归锁

****避免原因**：**

- 复杂实现
- 潜在死锁
- 难以调试
- 对教学不必要

19.11 未来设计考虑

19.11.1 潜在增强

1. ****增强安全机制****
 - 根据不同硬件平台，支持可选的内存保护和运行安全机制
 - 提升系统可靠性
2. ****多定时器支持****
 - 支持多个硬件定时资源
 - 提供更灵活的时间管理方案
3. ****同步机制增强****
 - 支持优先级上限协议等高级同步方案
 - 进一步降低复杂实时场景下的优先级反转风险
4. ****事件组机制****
 - 支持 AND/OR 条件组合

- 提供更丰富的任务同步方式

5. ****内存管理增强****

- 增加内存池等可选机制
- 提供更灵活的动态资源管理能力

19.11.2 设计演进

在保持核心设计原则的基础上，未来版本可以：

- 增加经过验证的可选高级功能
- 持续优化硬件抽象层
- 增强调试与诊断支持
- 提供更多配置选项
- 持续完善开发文档与示例

所有功能演进应以提升实际应用价值为目标，同时保持系统轻量、高效、易理解的核心特点。

20 HRTOS 扩展指南

20.1 模块介绍

本指南用于说明 HRTOS

功能扩展的方法和设计原则，帮助开发人员在保持系统稳定性、兼容性和实时性的基础上增加新功能。

扩展应遵循：

- 保持现有接口兼容
- 遵循已有架构设计
- 控制资源占用
- 完成功能验证
- 同步更新文档

20.2 扩展原则

20.2.1 兼容性

扩展功能时应避免破坏已有 API 和运行行为。

推荐：

- 新增接口，而不是修改已有接口
- 保持数据结构稳定
- 记录必要的兼容性变化

20.2.2 最小化影响

新增功能应尽量独立：

- 减少对核心内核修改
- 避免增加不必要的依赖
- 控制代码复杂度

20.2.3 统一设计模式

扩展模块应遵循 HRTOS 现有设计方式：

- 使用统一资源模型
- 使用统一等待机制
- 遵循 API 命名规则
- 保持分层架构

20.3 扩展方向

20.3.1 添加新的 IPC 机制

新增 IPC 前应确认现有机制无法满足需求。

设计时需要考虑：

- 资源结构
- 等待方式
- 唤醒流程
- ISR 安全需求

扩展流程：

- 定义资源类型
- 实现初始化接口
- 实现等待和释放操作
- 添加公共 API
- 增加测试验证

新增 IPC 应遵循现有等待模型，通过统一等待机制实现任务阻塞和唤醒。

20.3.2 添加系统服务

系统服务用于提供内核级公共功能。

扩展步骤：

- 定义服务接口
- 实现服务逻辑
- 与内核初始化流程连接
- 添加使用文档和示例

服务设计应避免直接依赖具体应用逻辑。

20.3.3 扩展现有模块

扩展已有模块时：

- 分析当前实现
- 确认扩展位置
- 保持 API 风格一致

- 验证原有功能

优先采用增加接口的方式扩展，而不是修改已有行为。

20.3.4 添加新的 API

新增 API 时应明确：

- 参数定义
- 返回值
- 使用条件
- 错误处理方式

API 应：

- 使用统一命名规则
- 包含必要参数检查
- 更新 API 文档

20.3.5 配置扩展

新增配置项时：

- 使用统一配置文件管理
- 提供合理默认值
- 说明配置影响
- 验证资源变化

配置修改不应破坏默认运行环境。

20.3.6 移植到新硬件

移植需要关注：

- CPU 架构
- 寄存器资源
- 定时器配置
- 中断系统
- 内存布局

移植流程：

- 分析目标硬件
- 实现硬件适配层
- 配置系统时钟
- 实现定时器中断
- 测试内核功能

20.4 扩展约束

20.4.1 内存约束

扩展功能需要评估：

- RAM 占用
- XDATA 使用
- 任务栈需求
- 全局变量增加

避免因为功能增加影响系统稳定运行。

20.4.2 性能约束

扩展必须保证：

- 有界执行时间
- 可预测响应时间
- 最小中断影响
- 合理资源消耗

20.4.3 实时性约束

实时系统扩展不能只关注功能，还需要考虑：

- 调度影响
- 中断延迟
- 上下文切换开销

20.5 扩展测试

20.5.1 功能测试

验证：

- 基本功能
- 参数边界
- 错误处理
- 异常情况

20.5.2 集成测试

验证：

- 与任务系统协同
- 与 IPC 协同
- 与中断协同

- 与现有模块兼容

20.5.3 性能测试

测试：

- 执行时间
- 内存变化
- 调度影响
- 长时间运行稳定性

20.5.4 回归测试

确保：

- 原有功能正常
- API 行为不变
- 系统稳定运行

20.6 扩展检查清单

提交扩展前确认：

- ☐ 遵循 HRTOS 编码规范
- ☐ 使用统一设计模式
- ☐ 完成参数检查
- ☐ 完成测试验证
- ☐ 更新相关文档
- ☐ 评估内存影响
- ☐ 评估实时性能影响
- ☐ 保持兼容性

20.7 常见扩展问题

20.7.1 破坏兼容性

问题：

修改已有接口导致旧程序无法运行。

解决：

优先新增接口，避免修改已有行为。

20.7.2 忽略资源限制

问题：

新增功能超过 MCU 资源能力。

解决：

实现前评估内存和性能影响。

20.7.3 缺少同步保护

问题：

多个任务访问共享资源导致异常。

解决：

使用临界区或系统同步机制保护。

20.7.4 未进行充分测试

问题：

功能可以运行，但存在隐藏问题。

解决：

进行功能测试、压力测试和长期运行测试。

20.8 文档要求

扩展完成后需要同步更新：

- API 文档
- 使用示例
- 架构说明
- 配置说明
- 测试记录

20.9 总结

HRTOS 扩展应以稳定性和可维护性为核心。

新增功能不是简单增加代码，而是在保持轻量、高效、易理解的基础上持续完善系统能力。

Debugging

21 HRTOS 调试指南

21.1 模块介绍

本指南用于帮助开发者定位 HRTOS 系统运行过程中遇到的问题，包括任务运行异常、调度异常、时间精度问题、中断问题以及资源同步问题。

调试过程应遵循：

- 从简单现象入手
- 使用可测量的方法定位问题
- 避免引入额外影响系统实时性的调试代码
- 修改后进行充分验证

21.2 调试工具

21.2.1 串口输出调试

21.2.1.1 目的

通过串口输出关键运行信息，辅助分析系统状态。

21.2.1.2 适用场景

- 任务创建结果检查
- 状态变量观察
- 参数配置确认
- 错误定位

21.2.1.3 注意事项

串口输出会增加系统执行时间，可能影响实时性能。

在实时性敏感代码中，应避免大量打印。

建议：

- 调试阶段使用
- 测试完成后关闭或减少输出

21.2.2 逻辑分析仪

21.2.2.1 目的

分析 HRTOS 运行时序，验证实时性能。

21.2.2.2 可测量内容

- 任务切换时间
- 中断响应时间
- 时间片周期
- 延时精度
- GPIO 时序变化

21.2.2.3 使用方法

通过 GPIO 输出标志信号：

```
GPIO = 1;  
  
// 执行需要测量的代码  
  
GPIO = 0;
```

使用逻辑分析仪观察高低电平持续时间，即可获得执行时间。

21.3 任务调试

21.3.1 任务未运行

21.3.1.1 常见现象

- 任务创建成功但没有执行
- 其他任务运行正常
- 系统没有明显错误

21.3.1.2 检查步骤

- 检查任务创建结果
- 检查任务状态
- 检查任务优先级
- 检查任务阻塞条件

21.4 调度调试

21.4.1 调度异常检查

21.4.1.1 检查内容

- 时间片配置
- 调度请求
- 当前任务状态
- 等待任务状态

21.5 时间性能测试

21.5.1 时间片测试

验证系统时间片周期是否准确。

21.5.2 延时精度测试

验证任务延时功能准确性。

21.5.3 任务切换时间测试

测量上下文切换开销。

21.6 中断调试

21.6.1 中断未响应

检查：

- 中断是否开启
- 中断入口是否正确
- 中断源配置是否正确
- ISR 是否正常退出

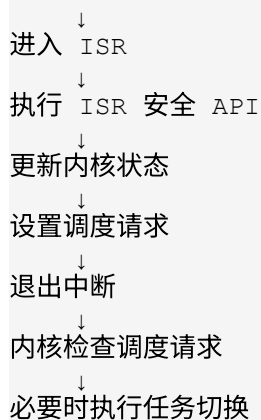
21.6.2 ISR 注意事项

ISR 中应：

- 保持执行时间短
- 避免阻塞
- 使用 ISR 安全 API
- 不直接执行复杂任务操作

HRTOS 中断处理流程：

硬件中断



21.7 栈调试

21.7.1 栈溢出问题

常见现象：

- 系统异常复位
- 变量异常变化
- 任务运行不稳定

检查：

- 任务栈空间
- 函数调用深度
- 局部变量使用
- 中断嵌套情况

21.8 IPC 调试

21.8.1 信号量问题

检查：

- 信号量是否正确创建
- 等待任务是否能够唤醒
- 释放操作是否执行

21.8.2 互斥锁问题

检查：

- 是否正确释放锁
- 是否存在资源竞争
- 是否长时间占用资源

21.8.3 消息通信问题

检查：

- 消息发送状态
- 接收任务状态
- 消息缓冲区状态

21.9 常见问题排查

| 问题 | 可能原因 | 检查方向 |

| - | - | - |

系统无响应	中断异常、调度停止	检查 Timer0、中断状态
任务不运行	创建失败、阻塞等待	检查任务状态
调度异常	时间配置错误	检查调度触发
栈溢出	栈空间不足	检查栈使用
中断异常	ISR 配置错误	检查中断入口
IPC 无响应	资源状态异常	检查同步对象

21.10 调试最佳实践

21.10.1 分阶段验证

建议：

- 单任务运行
- 多任务调度
- 增加 IPC
- 增加中断
- 长时间运行测试

21.10.2 保持测试可重复

记录：

- 硬件环境
- 编译配置
- 测试程序
- 测试时间

21.10.3 修改后回归测试

每次修改内核后：

- 重新编译
- 执行基础测试
- 执行长时间运行测试

确保系统稳定。

21.11 总结

HRTOS 调试应以实际运行数据为依据，通过串口、逻辑分析仪和硬件测试手段定位问题。

由于实时系统对时间敏感，调试过程中应尽量减少对系统运行环境的影响，优先采用低侵入性的测试方法。