README

HRTOS 系统手册

1.1 概述

HRTOS（Hard Real-Time Operating System）是一款面向 8051 系列微控制器的实时操作系统，围绕任务调度、上下文切换、中断处理、任务同步和系统资源管理构建内核。

HRTOS 采用标准任务与高速任务相结合的任务模型，提供任务管理、优先级调度、时间片机制、统一 IPC 资源管理、消息队列、邮箱、定时服务及中断相关机制。内核结合 8051 的寄存器组、堆栈和数据存储体系进行设计，在有限的硬件资源下提供完整的任务管理与系统运行支持。

本手册面向 HRTOS 内核开发者与维护者，系统介绍内核架构、核心机制、模块实现、内部数据结构、系统运行流程、调试方法及性能测试，为内核研究、代码维护、功能扩展和平台移植提供技术依据。

应用开发、模块调用及 API 使用方法，请参考 HRTOS 用户开发文档、API 文档、模块使用手册与示例工程。

1.2 文档结构

本手册按照内核架构、核心机制、系统运行和工程维护组织，主要包括以下内容：

- **内核架构：** 内核模块划分、工程目录结构、核心数据结构及模块依赖关系。
- **调度与任务管理：** 调度器设计、任务模型、任务状态、任务创建及上下文切换。
- **IPC 与资源管理：** 事件、信号量、互斥锁、消息队列、邮箱及统一等待与唤醒机制。
- **时间与中断：** 系统时基、任务延时、中断处理、临界区保护及调度请求。
- **内存与平台移植：** DATA、IDATA、XDATA、任务栈、寄存器组及目标芯片适配。
- **系统运行流程：** 内核启动、任务注册、调度执行、等待与唤醒等关键流程。
- **内核维护与扩展：** 模块维护、功能扩展、问题定位及修改验证。
- **调试与性能测试：** 任务切换、中断响应、定时精度、内存占用及相关测试结果。

各专题围绕具体模块展开，共同构成 HRTOS 内核的设计与维护文档体系。

1.3 系统特性

项目	说明
目标平台	8051 系列微控制器
主要开发环境	Keil C51
任务模型	标准任务与高速任务
调度机制	基于优先级的任务调度，支持相应的抢占式调度与时间片机制
时间管理	基于 Timer0 的系统时基及相关时间服务
任务管理	任务创建、删除、挂起、恢复、退出及状态维护
IPC 机制	事件、消息队列、信号量、互斥锁与邮箱
等待机制	统一等待状态管理、任务唤醒及超时处理
中断机制	中断上下文管理、临界区保护及调度请求
资源管理	静态配置、统一资源状态管理及任务注册控制

HRTOS 将任务调度、IPC、时间管理和中断处理组织为相互协作的内核模块，为 8051 应用提供统一的系统运行基础。

1.4 快速参考

1.4.1 核心数据结构

名称	用途
`OS_TCB`	保存标准任务的优先级、状态和等待信息
`OS_RESOURCE`	统一资源对象，保存资源状态及等待任务相关信息
`os_msgq_t`	消息队列控制结构，管理缓冲区、读写位置及消息数量
调度器状态变量	记录当前任务、调度状态及相关调度控制信息

1.4.2 核心配置

配置项	当前配置值	用途
`OS_PROCESS_MAX`	`16`	标准任务数量配置
`OS_FAST_TASK_MAX`	`2`	高速任务数量配置
`OS_RESOURCE_MAX`	`6`	统一资源对象数量
`OS_MSGQ_MAX`	`3`	消息队列对象数量

配置项	当前配置值	用途
`OS_SCHED_LEVEL_MAX`	`10`	调度级别上限定义

普通任务使用 0–7 级优先级，高速任务对应第 8 级，中断相关执行级别为 9 和 10。

1.4.3 内存布局

HRTOS 根据 8051 的内部数据存储区、扩展数据存储区和任务栈组织内核数据。以下为一组典型配置中的内存区域示例：

内存区域	地址范围	用途
系统栈区域	`0x35-0x39`	系统相关栈空间
中断栈区域	`0x3A-0x41`	中断处理相关栈空间
用户栈区域	`0x42-0x7F`	用户任务相关栈空间
XDATA 全局区域	`0x0200-0x03FF`	扩展数据存储区中的全局数据区域

上述地址对应特定的内存配置，不代表所有 8051 芯片的通用内存布局。实际使用时，应结合目标芯片的 RAM 容量、寄存器区划分、编译器存储分配和 HRTOS 移植配置进行规划。

1.5 系统要求

1.5.1 硬件要求

HRTOS 面向 8051 系列微控制器。进行芯片选型和工程移植时，需要重点考虑以下资源：

- **指令集与执行机制：** 支持目标内核所需的 8051 指令及相关硬件机制。
- **内部 RAM：** 满足内核变量、任务栈、寄存器保存及其他运行时数据的空间需求。
- **XDATA：** 满足统一资源对象、任务控制信息及其他扩展数据的存储需求。
- **程序存储器：** 满足内核、驱动和应用程序的代码空间需求。
- **定时器与中断：** 提供系统时基及应用运行所需的中断资源。

最低资源要求与实际可用任务数量、功能配置及应用规模有关。选型时应结合目标芯片的数据手册、内核配置和编译结果评估。

1.5.2 开发环境

- **Keil C51：** HRTOS 的主要开发环境。
- **目标芯片工具链：** 用于完成芯片配置、程序编译、下载和调试。
- **硬件测试环境：** 用于验证任务调度、中断响应、时间精度、内存使用及系统稳定性。

1.5.3 移植与构建

移植 HRTOS 时，需要根据目标芯片调整相关硬件配置，确认定时器与中断设置、存储区域分配及任务栈布局，并完成编译、链接和目标硬件测试。

修改任务数量、资源数量、时间参数或内存配置后，应重新检查编译结果、存储空间占用及系统运行表现。

1.6 使用说明

本手册主要用于：

- 理解 HRTOS 的整体内核架构与模块职责。
- 阅读和分析核心源码及内部数据结构。
- 研究任务调度、上下文切换、IPC 和中断处理机制。
- 定位系统运行问题，维护现有功能并扩展内核能力。
- 了解内存布局、平台适配及性能测试方法。

不同类型的技术资料具有明确分工：

- **系统手册：** 介绍内核整体架构、核心机制和维护方法。
- **内核概述：** 介绍内核模块、数据结构、任务状态和主要调用流程。
- **API 文档：** 介绍接口定义、参数、返回值及调用约定。
- **模块使用手册：** 介绍各功能模块的配置、使用方法和注意事项。
- **示例工程：** 展示接口调用、功能组合及实际应用方式。
- **性能测试文档：** 记录测试环境、测试方法、测量结果及相关分析。

建议先阅读系统手册和内核概述，建立整体认知，再根据具体研究目标进入调度器、任务管理、IPC、时间系统、中断或内存管理等专题。

官方网站：https://hrtos.com/

Kernel Overview

HRTOS 内核概述

2.1 模块介绍

HRTOS（Hard Real-Time Operating System）是一款面向 8051 系列微控制器的实时操作系统，围绕任务调度、上下文切换、中断处理、任务同步及系统资源管理构建内核。

内核采用标准任务与高速任务相结合的任务模型，提供任务管理、统一 IPC 资源管理、消息队列、邮箱、定时服务、事件同步及中断相关机制。通过静态配置管理核心任务与资源对象，并利用统一等待模型组织任务阻塞与唤醒，适配 8051 有限的存储资源和处理能力。

本文档面向内核开发者与维护者，介绍 HRTOS 的源码组织、核心配置、数据结构、任务状态、等待机制、内核对象、核心函数及主要运行流程，为源码阅读、系统调试、功能扩展和平台维护提供参考。

2.2 主要职责

HRTOS 内核主要负责以下工作：

- **任务管理：** 创建、删除、挂起、恢复、退出及任务状态维护。
- **任务调度：** 根据任务优先级和调度模式选择执行任务，支持相应的抢占式调度与时间片机制。
- **上下文切换：** 保存和恢复任务执行环境，实现任务之间的执行权切换。
- **统一 IPC 管理：** 通过统一资源对象组织相关资源状态，支持信号量、互斥锁、事件等同步机制。
- **消息队列与邮箱：** 管理消息缓冲区、读写位置、消息数量及消息收发操作。
- **统一等待机制：** 记录任务等待类型、等待对象、等待结果及超时相关信息。
- **任务唤醒：** 根据资源状态、信号及超时条件更新任务状态，使符合运行条件的任务重新参与调度。
- **定时服务：** 提供系统时基、任务延时、Tick 查询及时间转换功能。
- **中断管理：** 提供中断状态查询、中断 ID 获取、临界区保护及调度请求接口。
- **运行时维护：** 管理空闲任务、任务退出及相关清理操作。
- **任务注册控制：** 控制是否允许继续注册新任务，并提供任务优先级查询与设置接口。

2.3 源码目录结构

HRTOS 内核源码按照功能划分为多个模块。头文件集中提供公共接口、配置定义和内部声明；源文件分别实现内核调度、任务管理、IPC、定时服务、中断处理及平台相关功能。

2.3.1 头文件（Inc/）

文件	主要职责
`hrtos.h`	公共 API 汇总入口
`config.h`	系统配置、基础类型、状态常量及核心数据结构
`hrtos_internal.h`	内核内部声明与实现相关定义
`kernel.h`	内核初始化与核心接口
`task.h`	任务管理接口
`event.h`	事件同步接口
`interrupt.h`	中断处理、临界区及相关接口
`mutex.h`	互斥锁接口
`semaphore.h`	信号量接口
`time.h`	时间管理与延时接口
`wait.h`	统一等待机制接口
`mailbox.h`	邮箱接口
`msgq.h`	消息队列接口

2.3.2 内核模块（Src/kernel/）

文件	主要职责
`os_core.c`	内核核心数据及相关定义
`scheduler.c`	调度器选择与调度处理
`scheduler_dispatch.c`	任务分发与调度执行
`scheduler_mode.c`	调度模式配置与切换处理
`task_create.c`	任务创建与注册
`task_cleanup.c`	任务清理与相关资源处理
`idle_task.c`	空闲任务实现
`idle_hook_register.c`	空闲任务钩子注册
`os_data.c`	DATA 数据搬运
`os_xdata.c`	XDATA 数据搬运

2.3.3 任务模块（Src/task/）

文件	主要职责
task_delete.c	任务删除
task_exit.c	当前任务退出
task_is_valid.c	任务有效性检查
task_resume.c	任务恢复
task_scheduler.c	任务调度相关处理
task_self.c	获取当前任务 ID
task_state.c	查询任务状态
task_suspend.c	挂起指定任务
task_suspend_self.c	当前任务自挂起
task_yield.c	当前任务主动让出执行权

2.3.4 事件模块（Src/event/）

文件	主要职责
event_delete.c	事件删除
event_init.c	事件初始化
event_query.c	事件状态查询
event_wait.c	事件等待
event_write.c	事件写入与状态更新

2.3.5 中断模块（Src/interrupt/）

文件	主要职责
enter_critical.c	进入临界区，保护关键操作
exit_critical.c	退出临界区，恢复相关保护状态
in_isr.c	检查当前是否处于 ISR 上下文
interrupt_id.c	获取当前中断 ID
event_set_from_isr.c	在 ISR 中设置事件并唤醒任务
msgq_recv_from_isr.c	在 ISR 中接收消息队列数据并处理任务唤醒
msgq_send_from_isr.c	在 ISR 中发送消息队列数据并处理任务唤醒
queue_send_from_isr.c	在 ISR 中发送邮箱消息并处理任务唤醒
schedule_request.c	请求触发调度
semaphore_post_from_isr.c	在 ISR 中释放信号量并处理任务唤醒

2.3.6 互斥锁模块（Src/mutex/）

文件	主要职责
mutex_init.c	互斥锁初始化
mutex_lock.c	互斥锁加锁及相关处理
mutex_unlock.c	互斥锁解锁及相关处理

互斥锁用于管理共享资源的互斥访问。加锁与解锁过程结合任务等待和唤醒机制，处理资源占用及任务竞争。

2.3.7 信号量模块（Src/semaphore/）

文件	主要职责
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）

信号量通过计数状态和等待机制协调任务之间的同步关系。

2.3.8 定时模块（Src/time/）

文件	主要职责
delay.c	任务延时
delay_ms.c	毫秒级延时处理
ms_to_tick.c	毫秒到 Tick 的转换
sec_to_tick.c	秒到 Tick 的转换
tick_config.c	系统时基配置
tick_get.c	获取系统 Tick 计数
uptime_ms.c	获取系统运行时间，单位为毫秒

定时模块负责系统时间相关配置、时间计数及延时接口，为任务等待和时间管理提供基础支持。

2.3.9 等待模块（Src/wait/）

文件	主要职责
os_wait.c	统一等待核心逻辑
res_init.c	统一资源初始化
wake_task.c	任务唤醒与等待状态更新

等待模块负责协调任务控制块中的等待信息与资源对象中的等待记录，为不同 IPC 类型提供共用的等待和唤醒处理机制。

2.3.10 邮箱模块（Src/mailbox/）

文件	主要职责
mailbox_clear.c	邮箱清除
mailbox_init.c	邮箱初始化
mailbox_query.c	邮箱状态查询
mailbox_recv.c	邮箱接收
mailbox_send.c	邮箱发送

邮箱模块提供消息发送、接收、查询和清除操作，用于任务之间的消息传递。

2.3.11 消息队列模块（Src/msgq/）

文件	主要职责
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

消息队列模块管理队列缓冲区、读写位置和消息数量，并通过统一等待机制处理相关阻塞与唤醒。

2.3.12 任务注册控制与优先级管理（Src/task_priority/）

文件	主要职责
lock.c	锁定任务注册，禁止继续注册新任务
lock_query.c	查询任务注册锁定状态
unlock.c	解除任务注册锁定，允许继续注册新任务
task_get_priority.c	获取任务优先级
task_set_priority.c	修改任务优先级

该目录包含两类独立功能：任务注册控制用于管理任务注册权限；任务优先级接口用于查询和调整任务优先级。任务注册锁定不等同于任务优先级锁定。

2.3.13 移植模块（Src/port/）

文件	主要职责
`cpu_nop.c`	CPU 空操作相关实现

2.4 核心配置

核心配置集中定义在 `config.h` 中，包括基础类型、时间参数、任务数量、调度级别、任务状态、等待类型及资源对象容量。

2.4.1 基础类型

```
#define u8  unsigned char
#define u16 unsigned int
#define f32 float
#define u32 unsigned long
#define f64 double
#define NULL 0
```

这些类型定义用于统一内核代码中的基础类型名称。在 Keil C51 环境中，变量的实际存储位置还受到 `data`、`idata`、`xdata` 等存储类型限定的影响。

2.4.2 时间配置

```
#define OS_TIME_ONCE_DEFAULT 5
#define OS_TIME_T0 10000
```

- `OS_TIME_ONCE_DEFAULT`：默认时间片相关配置，当前值为 `5`。
- `OS_TIME_T0`：系统定时相关配置，当前值为 `10000`。

时间片配置与系统定时配置承担不同职责。系统实际时基和调度周期由相应的定时器配置与内核处理逻辑共同决定。

2.4.3 任务数量配置

```
#define OS_PROCESS_MAX 16
#define OS_FAST_TASK_MAX 2
#define OS_MAX_TASK OS_PROCESS_MAX
```

- `OS_PROCESS_MAX`：标准任务数量配置，值为 `16`。
- `OS_FAST_TASK_MAX`：高速任务数量配置，值为 `2`。
- `OS_MAX_TASK`：`OS_PROCESS_MAX` 的别名。

标准任务和高速任务采用不同的任务管理与调度处理方式。标准任务控制块数组的容量由 `OS_PROCESS_MAX` 决定，高速任务则由专门的高速任务机制管理。

2.4.4 调度级别

```
#define OS_SCHED_LEVEL_MAX 10
```

调度级别定义如下：

级别	含义
0–7	普通任务优先级
8	高速任务级别
9	中断级
10	中断嵌套级

普通任务优先级用于任务调度；中断级别用于描述中断相关的执行层级。中断级别 `9` 和 `10` 不属于普通任务优先级的设置范围。

2.4.5 IPC 资源数量

```
#define OS_RESOURCE_MAX 6
#define OS_MSGQ_MAX 3
```

- `OS_RESOURCE_MAX`：统一资源对象数组容量，值为 `6`。
- `OS_MSGQ_MAX`：消息队列对象数组容量，值为 `3`。

事件、信号量、互斥锁等使用统一资源模型的对象通过资源对象管理相关状态。邮箱和消息队列具有各自的对象组织方式。

2.5 核心数据结构

2.5.1 OS_TCB：任务控制块

`OS_TCB` 用于保存标准任务的优先级、运行状态和等待状态等信息。

```
typedef struct {
    u8 base_prio;
    u8 cur_prio;
    u8 state;
    u8 wait_type;
    u8 wait_flag;
    u8 wait_obj;
    u16 wait_tick;
} OS_TCB;
```

字段说明：

字段	含义
`base_prio`	任务基础优先级
`cur_prio`	当前生效的任务优先级
`state`	任务状态
`wait_type`	当前等待类型
`wait_flag`	等待结果标志
`wait_obj`	当前等待对象 ID
`wait_tick`	等待超时相关 Tick 计数

`OS_TCB` 不包含独立的 `id` 字段。任务 ID 通过任务管理中的索引规则与任务控制块对应。

2.5.2 OS_RESOURCE：统一资源对象

```
typedef struct {
    u8 value;
    u8 owner;
    u8 wait_cnt;
    u16 wait_mask;
    u8 pending_signal;
} OS_RESOURCE;
```

字段说明：

字段	含义
`value`	资源相关状态值
`owner`	资源拥有者信息，主要用于需要记录所有者的资源类型
`wait_cnt`	等待任务数量相关计数
`wait_mask`	等待任务位图
`pending_signal`	ISR 相关的待处理信号计数

`wait_mask` 是一个 16 位位图，用于表示标准任务的等待情况：

- bit0 对应 task0。
- bit1 对应 task1。
- 依此类推，bit15 对应 task15。

统一资源对象集中保存资源状态与等待任务信息，为 IPC 模块复用底层等待和唤醒逻辑提供支持。不同 IPC 类型根据自身功能使用相应字段。

2.5.3 os_msgq_t：消息队列控制结构

```
typedef struct {
    u8 *buf;
    u8 _size;
    u8 head;
    u8 tail;
    u8 count;
} os_msgq_t;
```

字段说明：

字段	含义
`buf`	消息缓冲区指针
`_size`	缓冲区容量
`head`	写指针，对应生产者
`tail`	读指针，对应消费者
`count`	当前消息数量

消息队列使用缓冲区和读写位置组织消息，并通过 `count` 记录当前队列中的消息数量。发送与接收操作需要同步维护读写位置和消息计数，并处理队列的边界状态。

2.6 核心内核对象

2.6.1 任务控制块数组

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

`OS_TASK` 保存标准任务的控制信息，数组容量由 `OS_PROCESS_MAX` 决定。当前配置支持最多 16 个标准任务控制块。

高速任务采用专门的管理与上下文处理机制，不应直接将 `OS_TASK` 的容量视为高速任务数量。

2.6.2 统一资源对象数组

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

`OS_RES` 保存统一资源对象，容量由 `OS_RESOURCE_MAX` 决定。当前配置支持 6 个统一资源对象。

资源对象的初始化和分配由资源初始化及对应 IPC 模块共同管理。

2.6.3 消息队列对象数组

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

`OS_MSGQ` 保存消息队列控制结构，容量由 `OS_MSGQ_MAX` 决定。当前配置支持 3 个消息队列对象。

每个队列通过自身的缓冲区指针、容量、读写位置和消息数量管理队列数据。

2.7 任务状态与等待机制

2.7.1 任务状态

```
#define READY    0
#define WAIT     1
#define RUNNING  2
#define SUSPEND  3
#define DEAD     4
```

状态	数值	含义
`READY`	0	就绪态，具备运行条件
`WAIT`	1	等待某种条件或资源
`RUNNING`	2	正在运行

状态	数值	含义
SUSPEND	3	挂起态
DEAD	4	任务结束或删除后的状态

任务状态用于描述任务在内核中的运行阶段。任务管理模块负责执行状态变更，调度器根据任务状态判断任务是否具备运行条件。

2.7.2 等待类型

```
#define WAIT_NONE      0
#define WAIT_DELAY     1
#define WAIT_SEM       2
#define WAIT_MUTEX     3
#define WAIT_MSG_SEND  4
#define WAIT_MSG_RECV  5
#define WAIT_EVENT     6
#define WAIT_MAILBOX   7
```

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

等待类型与 `wait_obj` 配合，标识任务当前等待的资源类别及具体对象。消息队列发送等待和接收等待分别描述发送方向和接收方向的阻塞场景。

内核通过任务控制块中的等待字段和资源对象中的等待记录维护等待关系。任务等待状态发生变化时，相关等待信息需要保持一致。

2.7.3 唤醒标志

```
#define WAIT_TIMEOUT 1
#define WAIT_SIGNAL  2
```

- `WAIT_TIMEOUT`：超时唤醒标志。
- `WAIT_SIGNAL`：信号唤醒标志。

唤醒标志用于区分任务结束等待的原因。统一等待与唤醒机制负责维护相应的等待结果。

2.7.4 统一等待模型

统一等待机制通过任务控制块中的等待字段和资源对象中的等待信息协同工作。

基本处理流程如下：

1. 等待接口确定等待类型及等待对象。
2. 内核更新任务控制块中的等待信息。
3. 对需要登记资源等待信息的操作，更新对应资源的等待记录。
4. 任务进入等待状态，并在必要时触发调度。
5. 资源满足条件或等待超时后，执行相应唤醒处理。
6. 唤醒逻辑更新任务状态和等待结果，任务重新参与调度。

事件、信号量、互斥锁、邮箱和消息队列通过统一等待机制协调任务阻塞与唤醒，同时保留各自的资源操作语义。

2.8 核心函数索引

2.8.1 初始化与运行

- `os_interrupt_init()`：中断系统初始化。
- `os_idle_task()`：空闲任务实现。
- `hrtos_main()`：应用层初始化入口。

2.8.2 任务管理

- `os_task_create()`：创建并注册任务。
- `os_task_cleanup()`：执行任务清理。
- `os_task_exit()`：当前任务退出。
- `os_task_delete()`：删除任务。

2.8.3 调度控制

- `os_set_scheduler()`：设置调度器相关模式。
- `os_scheduler_mode_switch()`：切换调度模式。
- `os_schedule_request()`：请求重新调度。

2.8.4 等待与唤醒

- `os_wait()`：统一等待核心。
- `wake_task()`：任务唤醒处理。

2.8.5 临界区

- `os_enter_critical()`：进入临界区。
- `os_exit_critical()`：退出临界区。

2.9 核心调用流程

2.9.1 系统启动流程

```
系统复位
    ↓
底层启动与初始化
    ↓
进入内核启动流程
    ↓
初始化任务及资源相关结构
    ↓
调用用户初始化入口 hrtos_main()
    ↓
创建并注册应用任务
    ↓
完成启动阶段配置
    ↓
启用相应中断及调度机制
    ↓
进入正常运行阶段
```

系统启动由底层启动代码、内核初始化、用户初始化和任务调度等环节共同完成。应用在初始化阶段创建并注册任务，内核随后进入正常调度运行阶段。

2.9.2 任务创建流程

```
os_task_create(...)
    ↓
检查创建参数
    ↓
检查任务 ID 与相关资源限制
    ↓
分配或定位任务栈
    ↓
初始化任务上下文
    ↓
设置任务控制信息
    ↓
完成任务注册
    ↓
返回执行结果
```

任务创建过程负责完成参数检查、任务资源准备、上下文初始化及任务注册，使新任务具备参与调度的条件。

2.9.3 等待与唤醒流程

```
调用等待接口
    ↓
确定等待类型与对象
    ↓
记录任务等待信息
    ↓
更新相关资源等待状态
    ↓
任务进入等待状态
```

```
    ↓
请求必要的调度
    ↓
资源满足条件或等待超时
    ↓
执行唤醒处理
    ↓
任务重新参与调度
```

等待接口负责建立任务与等待对象之间的关联；唤醒处理负责更新任务状态和等待结果。任务重新具备运行条件后，由调度器根据调度规则选择执行任务。

2.10 设计原则

2.10.1 内存效率

- 使用静态配置管理核心任务与资源对象。
- 采用紧凑的数据结构控制内核管理信息的存储开销。
- 通过任务栈管理机制组织任务执行所需的栈空间。
- 使用 DATA、IDATA 和 XDATA 等 8051 存储区域组织内核数据。

2.10.2 实时调度

- 根据任务优先级和调度模式执行任务选择。
- 支持相应的任务抢占与时间片机制。
- 通过专门的上下文切换逻辑管理任务执行环境。
- 提供中断相关接口、临界区保护及调度请求机制。

调度延迟、上下文切换时间和中断响应时间应结合目标芯片、编译配置、任务负载和实测结果进行评估。

2.10.3 8051 平台适配

- 根据目标芯片的数据存储体系组织变量。
- 结合寄存器组、堆栈和中断机制管理任务执行上下文。
- 针对目标编译环境实现关键路径。
- 根据芯片资源和编译结果检查内存布局。

2.10.4 统一 IPC 模型

- 使用 `OS_RESOURCE` 统一组织相关资源状态。
- 通过 `OS_TCB` 保存任务等待信息。
- 使用等待类型、等待对象和等待位图协同管理任务阻塞与唤醒。
- 在复用底层等待机制的同时，保留不同 IPC 类型的行为语义。

2.10.5 任务注册控制与优先级管理分离

- 通过任务注册控制接口管理是否允许继续注册新任务。

- 通过优先级接口查询和设置任务优先级。
- 将任务注册权限与任务运行优先级作为两类独立功能管理。

2.11 系统约束

当前核心配置如下：

项目	当前配置
标准任务数量配置	`OS_PROCESS_MAX = 16`
高速任务数量配置	`OS_FAST_TASK_MAX = 2`
统一资源对象数量	`OS_RESOURCE_MAX = 6`
消息队列对象数量	`OS_MSGQ_MAX = 3`
普通任务优先级范围	0–7
高速任务级别	8
中断相关调度级别	9–10
任务状态	`READY`、`WAIT`、`RUNNING`、`SUSPEND`、`DEAD`
等待类型	`WAIT_NONE` 至 `WAIT_MAILBOX`

内核使用固定容量的标准任务控制块数组、统一资源对象数组和消息队列对象数组。调整任务数量、资源数量、优先级或内存相关配置时，需要同步检查相关数据结构、调度逻辑、编译结果和目标硬件运行表现。

任务等待字段、资源等待记录及任务状态之间需要保持一致。任务注册控制、任务优先级管理和任务调度分别承担不同职责，不应混淆。

2.12 内核维护与调试

维护 HRTOS 内核时，可按照问题类型定位对应模块：

- **调度问题：** 检查任务优先级、调度模式、任务状态及调度请求路径。
- **任务阻塞：** 检查等待类型、等待对象、等待标志及唤醒条件。
- **IPC 问题：** 检查资源对象状态、等待位图和等待计数的一致性。
- **中断问题：** 检查中断上下文、临界区保护及调度触发逻辑。
- **消息队列问题：** 检查缓冲区容量、读写位置、消息计数及边界处理。
- **内存问题：** 检查任务栈以及 DATA、IDATA 和 XDATA 的分配与使用情况。
- **任务注册问题：** 检查任务注册锁定状态、任务创建时机及注册接口的调用顺序。
- **实时性能问题：** 在明确测试条件的前提下测量调度延迟、上下文切换时间和中断响应时间。

内核修改应保持接口定义、任务状态、等待关系及资源管理规则的一致性，并通过功能测试和目标硬件测试验证修改结果。

本概述用于建立 HRTOS 内核的整体认知。进一步研究具体功能时，可结合对应源文件、内部头文件、API 文档及专项设计说明，深入分析模块实现与调用关系。

Directory Analysis

HRTOS 目录分析

3.1 模块介绍

HRTOS 是一款面向 8051 系列微控制器的实时操作系统。工程采用模块化源码组织方式，将公共接口、内核实现、任务管理、进程间通信（IPC）、定时服务、中断处理及平台相关代码分别组织，形成职责清晰的系统结构。

本文档介绍 HRTOS 的头文件目录、源代码目录、主要模块职责及模块间的调用关系，为开发者理解内核结构、阅读源码和进行系统维护提供参考。

3.2 主要职责

HRTOS 的目录结构主要承担以下职责：

- **接口与实现分离：** 通过头文件组织接口声明，通过源文件实现具体功能。
- **功能模块划分：** 按内核、任务、IPC、定时和中断等功能组织源码。
- **统一等待与唤醒：** 通过统一等待机制管理任务阻塞、等待状态及任务唤醒。
- **平台适配：** 集中组织处理器相关的底层实现。
- **任务管理：** 提供任务创建、注册、状态管理、挂起、恢复和删除等功能。
- **应用开发支持：** 通过公共头文件提供应用程序所需的系统接口。

3.3 Inc/ 目录：头文件

`Inc/` 目录包含系统公共接口、配置定义、核心数据结构及内核内部声明。

3.3.1 文件清单

文件	主要职责
`hrtos.h`	公共 API 汇总入口
`config.h`	系统配置、类型定义、状态常量及核心数据结构
`hrtos_internal.h`	内核内部声明及实现相关定义
`kernel.h`	内核初始化与核心接口

文件	主要职责
task.h	任务管理接口
event.h	事件同步接口
interrupt.h	中断处理与临界区接口
mutex.h	互斥锁接口
semaphore.h	信号量接口
time.h	时间管理与延时接口
wait.h	统一等待机制接口
mailbox.h	邮箱接口
msgq.h	消息队列接口

3.3.2 目录特点

- 通过头文件声明函数、数据类型及相关配置。
- 通过 config.h 统一定义内核配置、基础类型、状态常量和核心数据结构。
- 通过 hrtos.h 为应用程序提供统一的头文件入口。
- 通过各功能头文件组织任务管理、事件同步、中断处理、资源同步、时间服务及 IPC 接口。
- 通过 hrtos_internal.h 组织内核内部使用的声明和实现相关定义。

3.4 Src/ 目录：源代码实现

Src/ 目录按照内核功能划分源码，各子目录分别实现对应的系统功能。

3.4.1 Src/kernel/：核心内核

文件	主要职责
os_core.c	核心内核数据结构及相关变量定义
scheduler.c	调度器选择与调度模式处理
scheduler_dispatch.c	任务分发处理
scheduler_mode.c	调度模式相关处理
task_create.c	任务创建与注册
task_cleanup.c	任务清理与相关资源处理
idle_task.c	空闲任务实现
idle_hook_register.c	空闲钩子注册
os_data.c	DATA 存储区相关处理
os_xdata.c	XDATA 存储区相关处理

主要职责

- 组织内核核心数据与调度状态。
- 实现调度器选择及任务分发。
- 负责任务创建与注册。
- 处理任务清理及相关资源。
- 实现空闲任务与空闲钩子注册。
- 组织 DATA、XDATA 存储区相关实现。

3.4.2 Src/task/：任务管理

文件	主要职责
`task_delete.c`	任务删除
`task_exit.c`	当前任务退出
`task_is_valid.c`	任务有效性检查
`task_resume.c`	任务恢复
`task_scheduler.c`	任务调度逻辑
`task_self.c`	获取当前任务 ID
`task_state.c`	查询任务状态
`task_suspend.c`	任务挂起
`task_suspend_self.c`	当前任务自挂起
`task_yield.c`	任务主动让出执行权

主要职责

- 管理任务状态与生命周期。
- 实现任务挂起、恢复、退出和删除。
- 提供当前任务 ID 与任务状态查询。
- 处理任务主动让出 CPU 的操作。
- 配合内核调度机制完成任务状态转换。

任务优先级的获取和设置由 `Src/task_priority/` 中的相应文件负责，与本目录的任务状态管理相互区分。

3.4.3 Src/event/：事件同步

文件	主要职责
`event_delete.c`	事件删除
`event_init.c`	事件初始化
`event_query.c`	事件状态查询
`event_wait.c`	事件等待
`event_write.c`	事件写入与状态更新

主要职责

- 实现事件对象的初始化与管理。
- 提供事件状态查询和更新。
- 实现任务事件等待。
- 配合统一等待与唤醒机制完成事件同步。

3.4.4 Src/interrupt/：中断处理

文件	主要职责
`enter_critical.c`	进入临界区
`event_set_from_isr.c`	ISR 事件设置与任务唤醒
`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_recv_from_isr.c`	ISR 消息队列接收与任务唤醒
`msgq_send_from_isr.c`	ISR 消息队列发送与任务唤醒
`queue_send_from_isr.c`	ISR 邮箱发送与任务唤醒
`schedule_request.c`	调度请求
`semaphore_post_from_isr.c`	ISR 信号量释放与任务唤醒

主要职责

- 提供临界区管理接口。
- 提供 ISR 上下文检查与中断 ID 查询。
- 支持 ISR 上下文中的事件、消息队列、邮箱及信号量操作。
- 配合内核统一等待与唤醒机制处理相关任务。
- 提供中断上下文中的调度请求接口。

ISR IPC 接口负责相应的通信或同步操作及任务唤醒相关处理，内核通过统一机制完成任务唤醒。

3.4.5 Src/mutex/：互斥锁

文件	主要职责
`mutex_init.c`	互斥锁初始化
`mutex_lock.c`	互斥锁加锁
`mutex_unlock.c`	互斥锁解锁

主要职责

- 实现互斥锁初始化与状态管理。

- 提供互斥锁获取和释放接口。
- 与统一等待机制配合处理加锁等待、任务阻塞及唤醒。
- 为任务之间的共享资源访问提供互斥同步。

3.4.6 Src/semaphore/：信号量

文件	主要职责
`sem_init.c`	信号量初始化
`sem_post.c`	信号量释放
`sem_wait.c`	信号量等待

主要职责

- 实现信号量初始化与状态管理。
- 提供信号量等待和释放接口。
- 与统一等待机制配合处理任务阻塞与唤醒。
- 支持任务之间的同步及事件通知。

3.4.7 Src/time/：定时服务

文件	主要职责
`delay.c`	任务延时
`delay_ms.c`	毫秒级延时处理
`ms_to_tick.c`	毫秒到 Tick 的转换
`sec_to_tick.c`	秒到 Tick 的转换
`tick_config.c`	系统时基配置
`tick_get.c`	获取系统 Tick 计数
`uptime_ms.c`	获取系统运行时间（毫秒）

主要职责

- 管理系统时基相关功能。
- 提供毫秒、秒与 Tick 之间的时间转换。
- 实现任务延时相关功能。
- 提供系统 Tick 计数查询。
- 提供系统运行时间查询。

3.4.8 Src/wait/：统一等待

文件	主要职责
`os_wait.c`	统一等待核心逻辑
`res_init.c`	资源初始化

文件	主要职责
`wake_task.c`	任务唤醒处理

主要职责

- 实现统一等待处理。
- 管理任务等待状态及等待条件。
- 实现任务唤醒与相关状态更新。
- 提供资源初始化及等待机制所需的底层处理。

该目录是理解 HRTOS 任务阻塞、等待条件处理及任务唤醒流程的核心入口。事件、互斥锁、信号量、邮箱和消息队列等功能通过相应的等待处理与内核调度机制协同工作。

3.4.9 Src/mailbox/：邮箱 IPC

文件	主要职责
`mailbox_clear.c`	邮箱清除
`mailbox_init.c`	邮箱初始化
`mailbox_query.c`	邮箱查询
`mailbox_recv.c`	邮箱接收
`mailbox_send.c`	邮箱发送

主要职责

- 实现邮箱初始化与状态管理。
- 提供邮箱消息发送和接收接口。
- 提供邮箱状态查询与清除操作。
- 与统一等待机制配合处理相关任务阻塞与唤醒。

3.4.10 Src/msgq/：消息队列

文件	主要职责
`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

主要职责

- 实现消息队列初始化与状态管理。
- 提供消息发送和接收接口。
- 管理队列缓冲区及读写状态。

- 与统一等待机制配合处理相关任务阻塞与唤醒。

3.4.11 Src/task_priority/：任务注册控制与优先级管理

该目录包含任务注册控制和任务优先级管理两类功能。

任务注册控制

文件	主要职责
`lock.c`	锁定任务注册，禁止继续注册新任务
`lock_query.c`	查询任务注册锁定状态
`unlock.c`	解除任务注册锁定，允许继续注册新任务

任务优先级管理

文件	主要职责
`task_get_priority.c`	获取任务优先级
`task_set_priority.c`	设置任务优先级

任务注册控制用于管理系统是否允许继续注册新任务；任务优先级接口用于查询或设置任务优先级。两类功能职责独立，不应将任务注册锁定描述为优先级锁定。

3.4.12 Src/port/：平台相关实现

文件	主要职责
`cpu_nop.c`	CPU 空操作相关实现

主要职责

- 组织与目标处理器相关的底层实现。
- 为处理器相关操作提供独立的源码位置。

3.5 核心函数与接口组织

3.5.1 公共头文件入口

应用程序通过 `Inc/hrtos.h` 使用 HRTOS 公共接口：

```
#include "hrtos.h"
```

`hrtos.h` 汇总应用开发所需的相关头文件，使应用程序能够通过统一入口访问系统接口。`config.h` 提供系统配置、基础类型及核心数据结构定义，其他功能头文件分别组织对应模块的 API。

3.5.2 内核启动与任务调度

HRTOS 的启动和运行涉及内核初始化、任务创建与注册、系统时钟、中断处理及任务调度等功能。

主要功能环节包括：

- **内核数据准备：** 初始化内核运行所需的数据与状态。
- **任务创建与注册：** 建立任务及其相关调度信息。
- **资源与同步对象初始化：** 初始化应用使用的资源及 IPC 对象。
- **时钟与中断处理：** 提供系统时基和相关中断处理。
- **调度与任务执行：** 根据调度条件选择任务并执行上下文切换。
- **等待与唤醒：** 在任务阻塞条件满足后恢复任务运行资格。

3.5.3 统一等待与 IPC 的协作

事件、互斥锁、信号量、邮箱和消息队列各自提供独立的操作接口，并通过统一等待与唤醒机制协调任务阻塞和恢复。

- **事件：** 提供事件状态设置、查询与等待。
- **互斥锁：** 提供共享资源访问的互斥控制。
- **信号量：** 提供任务同步与通知。
- **邮箱：** 提供邮箱消息发送与接收。
- **消息队列：** 提供队列消息发送与接收。
- **统一等待：** 管理相关等待条件与任务唤醒。
- **调度器：** 在调度条件满足时选择后续运行任务。

3.6 模块间调用关系

HRTOS 各模块按照功能职责协同工作，形成以下主要关系：

```
任务管理
    ├── 核心内核与调度
    ├── 统一等待与唤醒
    └── 任务注册控制与优先级管理

事件模块 ──────┐
互斥锁模块 ────┤
信号量模块 ────┤
邮箱模块 ──────┼──→ 统一等待与唤醒
消息队列模块 ──┘

中断模块
    ├── ISR 事件操作
    ├── ISR 消息队列操作
    ├── ISR 邮箱操作
    ├── ISR 信号量操作
    └── 调度请求
```

```
定时模块
    └── 任务延时与时间相关处理

核心内核与调度
    ├── 任务选择与分发
    ├── 任务创建与注册
    └── 任务清理与空闲任务管理
```

3.6.1 头文件组织

公共头文件通过 `hrtos.h` 汇总，主要功能头文件包括：

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

该结构体现了 HRTOS 公共接口按功能划分的组织方式，便于应用程序根据统一入口或具体功能头文件使用相应接口。

3.7 目录维护建议

- **接口变更：** 修改公共 API 时，同步维护函数声明、实现及接口文档。
- **配置变更：** 调整任务数量、资源数量或调度参数后，检查内存占用和运行行为。
- **等待机制变更：** 检查任务控制块中的等待信息、资源等待记录及唤醒路径。
- **中断接口变更：** 检查 ISR 上下文限制、临界区处理和调度请求逻辑。
- **任务注册控制变更：** 确保锁定与解锁行为符合任务注册阶段的设计约定。
- **平台相关变更：** 结合目标芯片、编译器及存储模型进行测试。
- **文档维护：** 保持目录说明、函数职责、API 文档与当前实现一致。

3.8 总结

HRTOS 采用模块化目录结构，将核心内核、任务管理、IPC、定时服务、中断处理及平台相关实现组织在不同模块中。

`Src/kernel/` 负责核心内核与调度相关功能，`Src/task/` 管理任务生命周期，`Src/wait/` 负责统一等待与唤醒，事件、互斥锁、信号量、邮箱和消息队列模块分别提供对应的同步与通信功能，`Src/interrupt/` 提供中断上下文相关操作，`Src/task_priority/` 则负责任务注册控制和优先级管理。

通过清晰的模块划分，HRTOS 将任务调度、同步通信、时间管理及中断相关操作组织为相互协作的内核功能，为源码阅读、系统维护和应用开发提供统一的工程结构。

Kernel Startup

HRTOS 内核启动

4.1 模块介绍

HRTOS 内核启动过程负责将 8051 MCU 从初始运行环境引导至操作系统运行状态，完成内存初始化、内核变量配置、基础任务创建、Timer0 系统时基启动，以及应用初始化入口调用。

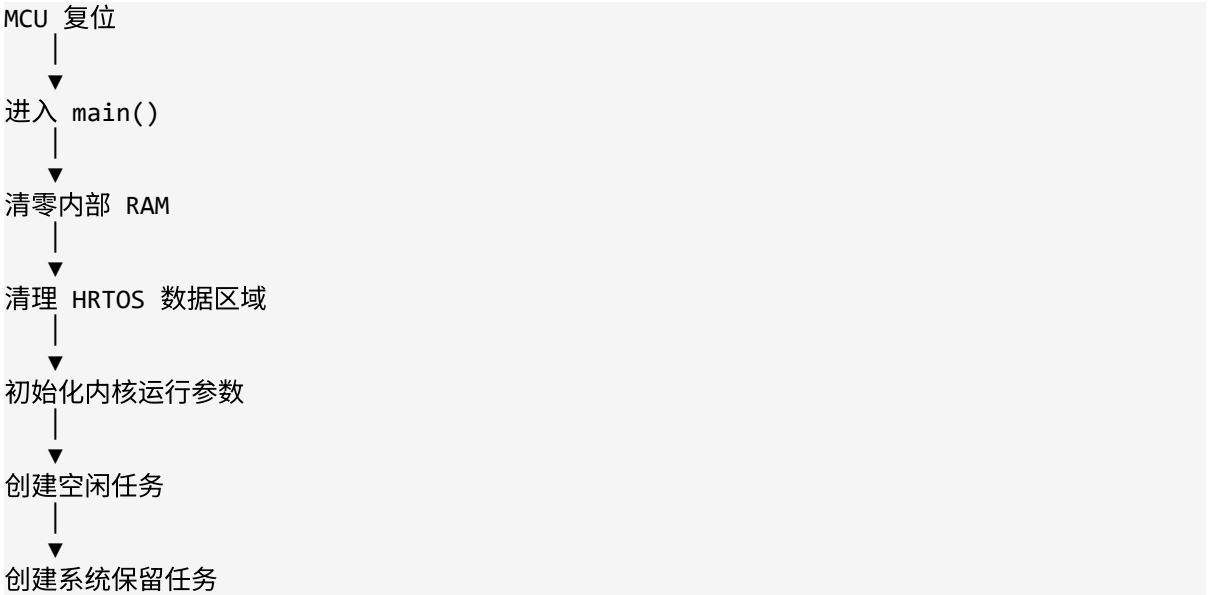
当前启动流程主要由用户工程中的 `main()` 和内核中的 `os_idle_task()` 协同完成。

`main()` 负责底层运行环境准备、内核初始参数设置、系统基础任务创建、初始任务栈设置以及 Timer0 启动。

`os_idle_task()` 负责初始化任务等待状态和资源管理状态，调用用户应用初始化入口 `hrtos_main()`，锁定系统配置，并进入永久空闲循环。

系统启动过程中需要特别注意中断状态、任务栈布局和初始化顺序。相关操作与 HRTOS 的 8051 上下文切换机制紧密关联，不应随意调整执行顺序。

4.2 启动流程



```
    │
    ▼
设置初始任务栈
    │
    ▼
装载 Timer0 初始计数值
    │
    ▼
配置 Timer0 并开启中断
    │
    ▼
调用 os_idle_task()
    │
    ▼
关闭总中断
    │
    ▼
初始化任务状态与资源状态
    │
    ▼
调用 hrtos_main()
    │
    ▼
锁定系统配置
    │
    ▼
重新开启总中断
    │
    ▼
进入永久 Idle 循环
    │
    ├── 清理已删除的普通任务
    │
    ├── 处理快速任务删除请求
    │
    ├── 执行 Idle Hook
    │
    └── 循环等待后续调度
```

需要注意，以上是源码层面的函数调用和初始化顺序。任务实际何时切换、首次进入哪个用户任务，以及中断返回时如何恢复任务现场，还取决于 HRTOS 的调度和上下文切换实现。

4.3 系统入口 main()

`main()` 是用户工程中的系统启动入口。该函数在进入 `os_idle_task()` 之前，完成底层内存处理、内核初始变量配置、基础任务创建和 Timer0 启动。

4.3.1 内部 RAM 初始化

启动代码首先执行一段嵌入式汇编：

```
MOV     R0,#7FH
CLR     A
```

```
OS_HRTOS_MAIN_INIT:
MOV     @R0,A
DJNZ    R0,OS_HRTOS_MAIN_INIT
```

执行过程如下：

- 将寄存器 `R0` 设置为 `0x7F`。
- 将累加器 `A` 清零。
- 使用 `@R0` 间接寻址方式向内部 RAM 写入零。
- 每次写入后递减 `R0`，直到循环结束。

该循环将地址 `0x7F` 至 `0x01` 的内部 RAM 单元清零。由于循环在 `R0` 递减到零时退出，地址 `0x00` 不会被这段循环清零。

因此，准确地说，这段代码清零的是内部 RAM 的 `0x01` 至 `0x7F` 地址，而不是完整的 `0x00` 至 `0x7F` 地址范围。

这属于启动阶段的底层内存操作，直接依赖 8051 内部 RAM 地址空间及寄存器使用约定。修改时应同时考虑寄存器组、启动代码和编译器生成代码的影响。

4.3.2 HRTOS 数据区域初始化

内部 RAM 清理后，启动代码执行：

```
os_memset(&OS_PROCESS_OK[0], 0, 0x01FF);
```

该调用从 `OS_PROCESS_OK[0]` 的地址开始，按 `os_memset()` 的实际接口语义清零指定长度的数据区域。

这与前面的内部 RAM 汇编清零不同：汇编代码明确操作内部 RAM，而该调用的实际目标地址空间、长度单位及覆盖范围，需要结合 `os_memset()` 的实现和变量存储声明确认。

不能仅凭这行代码就认定清零范围一定从 `OS_CURRENT_TASK` 开始，也不能在没有核实变量布局的情况下将其描述为固定的片外 RAM 地址区间。

启动时必须保证该清零范围有效，并且不会覆盖尚需保留的启动状态或其他内存对象。

4.4 内核初始参数

完成数据区域清理后，`main()` 设置内核启动阶段需要的变量。

4.4.1 初始调度状态

```
OS_DISPATCH_ID = 1;
OS_CURRENT_TASK = 0;
OS_INTERRUPT_T0 = 0;
OS_INTERRUPT_GAO = 0;
OS_RTOS_YES = 1;
```

```
OS_INTERRUPT_PROTECT = 0;
OS_TIME_ONCE = OS_TIME_ONCE_DEFAULT;
```

这些变量分别参与系统初始调度状态、当前任务标识、中断相关状态、运行模式以及时间片参数管理。

其中：

- `OS_CURRENT_TASK = 0`：将初始任务标识设置为零。
- `OS_RTOS_YES = 1`：将系统运行模式变量设置为启用状态。
- `OS_TIME_ONCE = OS_TIME_ONCE_DEFAULT`：使用配置文件提供的默认时间片计数参数。
- `OS_INTERRUPT_T0`、`OS_INTERRUPT_GAO` 和 `OS_INTERRUPT_PROTECT`：初始化相关中断状态及保护状态。

这些变量的精确语义应以各自的内核使用位置和状态转换逻辑为准，不能仅凭变量名推断完整行为。

4.4.2 快速任务删除状态

```
OS_KUAI_PROCESS_A_DEL = 0;
OS_KUAI_PROCESS_B_DEL = 0;
```

这两个变量用于记录快速任务 A、快速任务 B 的删除请求状态。

启动时将其清零，避免在初始化阶段遗留无效的删除请求。运行过程中，`os_idle_task()` 会检查这两个标志，并分别调用 `os_task_cleanup()` 处理对应快速任务的清理工作。

4.4.3 系统 Tick 计数器

```
OS_TICK_COUNT_L = 0;
OS_TICK_COUNT_H = 0;
```

两个变量分别保存累计 Tick 计数的低 16 位和高 16 位。

组合关系如下：

```
32 位系统 Tick 计数
┌──────────────────┬──────────────────┐
│   高 16 位       │    低 16 位      │
│ OS_TICK_COUNT_H  │ OS_TICK_COUNT_L  │
└──────────────────┴──────────────────┘
```

在计数逻辑正确处理低 16 位溢出的前提下，高低两个计数器可以扩展系统累计 Tick 的表示范围。其实际更新时间及溢出处理方式应以 Timer0 中断实现为准。

4.4.4 临界区嵌套计数

```
os_critical_nesting = 0;
```

将临界区嵌套状态初始化为零，使内核从预期的初始状态开始管理临界区。

该变量的作用与临界区实现方式相关，启动时应保证其初值与中断开关状态和内核临界区机制相匹配。

4.5 系统基础任务创建

4.5.1 创建空闲任务

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

该调用将 `os_idle_task()` 注册为系统空闲任务。

从当前启动代码可见，创建参数依次为任务入口、第二个参数 `0`、优先级参数 `0` 和最后一个参数 `2`。各参数的完整含义应以 `os_task_create()` 的函数声明和实现为准。

空闲任务承担系统初始化后续阶段及永久空闲循环中的工作，包括：

- 调用用户初始化入口 `hrtos_main()`。
- 锁定系统配置。
- 检查普通任务的删除状态。
- 处理快速任务删除请求。
- 执行已注册的 Idle Hook。

空闲任务的创建并不意味着它立即开始执行。实际执行时机取决于启动代码、当前任务状态以及上下文切换机制。

4.5.2 创建系统保留任务

```
os_task_create(
    (unsigned int)os_interrupt_exit, 0, 9, 0);

os_task_create(
    (unsigned int)os_interrupt_exit, 0, 10, 0);
```

启动代码还创建了两个入口指向 `os_interrupt_exit` 的系统保留任务，参数中的优先级值分别为 `9` 和 `10`。

这些任务与 HRTOS 的中断处理和退出机制有关。由于入口直接使用 `os_interrupt_exit`，其实际执行语义取决于内核如何组织任务入口、异常路径及中断上下文。

这部分属于内核内部约定，不应将其简单理解为两个普通的用户应用任务，也不应在未核实调度和中断实现前推断其完整运行路径。

4.6 初始任务栈设置

```
SP = OS_JINCHENG_JILU_SP[0] - 2;
```

该语句根据任务记录中的栈位置设置 8051 的栈指针 `SP`。

结合代码注释，该调整用于匹配初始任务进入时预期的栈布局，使首次进入初始任务时的栈状态符合 HRTOS 的上下文切换约定。

8051 的上下文切换通常与返回地址、寄存器现场及栈指针状态紧密相关。`SP` 的初始值不能脱离任务创建代码、栈初始化代码和汇编切换实现单独修改。

移植到不同编译器或不同 8051 内核实现时，应重新核对：

- 任务栈的初始布局。
- 返回地址的压入与弹出方式。
- 寄存器现场保存和恢复顺序。
- 首次进入任务时的栈指针位置。

4.7 Timer0 系统时基初始化

HRTOS 当前启动代码使用 Timer0 作为系统时间基准。

4.7.1 计算初始计数值

```
OS_T0_TH0 = (65535 - OS_TIME_T0) / 256;
OS_T0_TL0 = (65535 - OS_TIME_T0) % 256;
```

该代码将计算结果拆分为高、低字节，分别保存至 `OS_T0_TH0` 和 `OS_T0_TL0`。

随后：

```
TH0 = OS_T0_TH0;
TL0 = OS_T0_TL0;
```

将计算结果写入 Timer0 的高、低字节寄存器。

对于 16 位定时器，常见的重装值计算形式为：

$$R = 65536 - N$$

其中 N 是期望的定时器计数次数，R 是初始计数值。实际溢出周期还取决于目标芯片的定时器计数时钟、分频设置和工作模式。

当前代码使用 `65535 - OS_TIME_T0`，与常见的 `65536 - N` 形式相差一个计数单位。是否属于有意设计，应结合 HRTOS 的时间基准定义及目标芯片的实测周期确认。

4.7.2 配置并启动 Timer0

```
TMOD |= 0x01;

ET0 = 1;
TR0 = 1;
EA = 1;
```

各操作的作用如下：

- `TMOD |= 0x01`：设置 Timer0 的模式选择位，使其进入模式 1，即 16 位定时器模式。该操作不会清除 `TMOD` 的其他位。
- `ET0 = 1`：允许 Timer0 中断。
- `TR0 = 1`：启动 Timer0。
- `EA = 1`：开启 CPU 总中断允许。

Timer0 在系统运行后用于驱动系统 Tick、时间片管理及相关调度逻辑。其完整中断处理过程由 Timer0 的中断服务程序实现。

由于 Timer0 被系统用作时基，应用程序不应在没有确认内核约定的情况下擅自修改其工作模式、计数寄存器或中断配置。

4.8 进入 os_idle_task()

完成初始变量、任务和 Timer0 配置后，`main()` 直接调用：

```
os_idle_task();
```

这不是一次普通的任务创建操作，而是启动代码直接调用空闲任务函数，继续完成系统运行阶段的初始化。

在该函数执行过程中，系统先关闭总中断，完成运行状态初始化和用户应用初始化，然后重新开启总中断并进入永久循环。

如果 `os_idle_task()` 意外返回，`main()` 后续代码将继续执行。因此，正常运行情况下，空闲任务应保持永久循环，不应返回到启动函数的后续路径。

4.9 空闲任务初始化

4.9.1 关闭总中断

`os_idle_task()` 开始时执行：

```
EA = 0;
```

在初始化任务状态、资源状态以及调用 `hrtos_main()` 期间，函数保持总中断关闭，避免相关初始化过程被中断打断。

这也意味着 `hrtos_main()` 在当前实现下执行时，总中断处于关闭状态。应用初始化函数应避免依赖需要中断持续运行才能完成的同步操作，否则可能造成等待无法结束。

4.9.2 初始化 Hook 与调度状态

```
g_idle_hook = 0;
g_slab_overflow_hook = 0;

OS_SCHED_REASON = 0;
```

```
OS_CFG_LOCK = 0;
```

其中：

- `g_idle_hook`：清空 Idle Hook 函数指针。
- `g_slab_overflow_hook`：清空 Slab 溢出处理 Hook 函数指针。
- `OS_SCHED_REASON`：清除调度原因状态。
- `OS_CFG_LOCK = 0`：初始化阶段允许注册相关配置项。

这一步使系统从明确的初始状态开始运行，并允许用户在 `hrtos_main()` 中完成应用配置。

4.9.3 初始化普通任务状态

```
for (i = 0; i < OS_PROCESS_MAX; i++)
{
    OS_TASK[i].wait_type = WAIT_NONE;
    OS_TASK[i].state = READY;
    OS_TASK[i].wait_tick = 0;
    OS_TASK[i].wait_obj = OS_INVALID_ID;
}
```

循环初始化普通任务数组中的等待状态及相关字段：

- `wait_type` 设置为 `WAIT_NONE`。
- `state` 设置为 `READY`。
- `wait_tick` 清零。
- `wait_obj` 设置为 `OS_INVALID_ID`。

这些赋值用于建立统一的任务状态初值。它们并不意味着所有普通任务都会立即执行；任务是否有效、入口是否注册以及何时获得运行机会，仍取决于任务创建和调度逻辑。

该循环的范围为 `0` 至 `OS_PROCESS_MAX - 1`，不应直接据此推断快速任务的状态也在此处完成初始化。

4.9.4 初始化快速任务等待状态

```
OS_WAIT_DI2[0] = 0;
OS_WAIT_DI2[1] = 0;
```

该操作将两个相关状态项清零。其具体语义应结合快速任务等待与唤醒机制理解，不宜仅凭变量名称推断其完整含义。

4.9.5 初始化资源状态

```
for (i = 0; i < OS_RESOURCE_MAX; i++)
{
    OS_RES[i].value = 0;
    OS_RES[i].owner = OS_INVALID_ID;
    OS_RES[i].wait_cnt = 0;
    OS_RES[i].wait_mask = 0;
    OS_RES[i].pending_signal = 0;
}
```

循环初始化资源管理数组中的状态：

- `value` 清零。
- `owner` 设置为 `OS_INVALID_ID`。
- `wait_cnt` 清零。
- `wait_mask` 清零。
- `pending_signal` 清零。

这些操作为资源管理建立初始状态，避免资源所有者、等待信息或待处理信号保留未初始化值。

资源管理的具体行为仍由对应 IPC 或资源 API 决定。启动阶段的清零只负责建立初始状态，不代表所有资源已经由应用程序完成配置。

4.10 用户应用初始化 hrtos_main()

完成空闲任务内部的初始状态设置后，系统调用：

```
hrtos_main();
```

`hrtos_main()` 是用户应用的初始化入口，用于完成系统启动阶段所需的应用配置。

典型用途包括：

- 创建用户任务。
- 初始化应用使用的 IPC 对象。
- 配置驱动和应用模块。
- 注册允许在启动阶段设置的 Hook 或其他配置项。

示例：

```
void hrtos_main(void)
{
    /* 创建应用任务 */
    os_task_create(task1, ...);
    os_task_create(task2, ...);

    /* 初始化应用所需资源 */
    os_sem_init(...);
}
```

以上仅为接口使用示意，实际参数应以 HRTOS 对应 API 的定义为准。

4.10.1 初始化阶段的约束

`hrtos_main()` 执行时，`os_idle_task()` 已经关闭总中断。因此：

- 不应等待依赖 Timer0 中断才能结束的延时或同步条件。
- 不应在初始化入口中执行永久循环。
- 应避免依赖串口中断、外部中断等必须由 ISR 驱动的流程来完成阻塞式等待。

- 应在该阶段完成必要的任务创建和配置，再返回内核。

如果需要执行依赖中断的初始化流程，应重新设计初始化顺序，或在符合内核约定的阶段完成，而不能默认初始化期间中断始终开启。

4.11 锁定配置并开启中断

`hrtos_main()` 返回后，空闲任务执行：

```
OS_CFG_LOCK = 1;
EA = 1;
```

`OS_CFG_LOCK = 1` 表示启动阶段的配置窗口结束，后续是否允许注册配置项由相关接口的锁检查逻辑决定。

随后 `EA = 1` 重新开启总中断，系统进入中断驱动的正常运行阶段。

需要注意，`EA = 1` 是直接开启总中断，并不是恢复进入函数前的中断状态。该行为是当前启动流程的一部分，修改时应结合整个系统的中断管理约定进行评估。

4.12 永久空闲循环

完成初始化并重新开启总中断后，`os_idle_task()` 进入永久循环：

```
while (1)
{
    EA = 0;

    for (i = 1; i < OS_PROCESS_MAX; i++)
    {
        if (OS_TASK[i].state == DEAD)
        {
            os_task_cleanup(i);
        }
    }

    if (OS_KUAI_PROCESS_A_DEL)
    {
        os_task_cleanup(OS_PROCESS_MAX);
        OS_KUAI_PROCESS_A_DEL = 0;
    }

    if (OS_KUAI_PROCESS_B_DEL)
    {
        os_task_cleanup(OS_PROCESS_MAX + 1);
        OS_KUAI_PROCESS_B_DEL = 0;
    }

    EA = 1;

    if (g_idle_hook)
    {
```

```
        g_idle_hook();
    }
}
```

4.12.1 普通任务清理

循环从任务索引 `1` 开始，遍历到 `OS_PROCESS_MAX - 1`，检查任务状态是否为 `DEAD`。

如果任务已经进入删除状态，则调用：

```
os_task_cleanup(i);
```

进行相应的资源清理。

该循环刻意没有遍历普通任务索引 `0`。结合启动代码中任务 `0` 被用作系统空闲任务，可以看出空闲任务本身不在此处的普通任务删除扫描范围内。

4.12.2 快速任务清理

空闲循环还检查两个快速任务删除标志：

```
OS_KUAI_PROCESS_A_DEL
OS_KUAI_PROCESS_B_DEL
```

当相应标志置位时，分别调用：

```
os_task_cleanup(OS_PROCESS_MAX);
os_task_cleanup(OS_PROCESS_MAX + 1);
```

随后清除对应标志。

由此可见，快速任务删除请求通过标志交由空闲循环处理。快速任务自身的删除请求设置条件和安全时机，需要结合快速任务管理代码进一步确认。

4.12.3 Idle Hook

任务清理完成后，空闲循环重新开启总中断，并在 `g_idle_hook` 非空时调用该函数指针：

```
if (g_idle_hook)
{
    g_idle_hook();
}
```

Idle Hook 为用户提供空闲阶段的扩展入口，可用于适合该执行环境的轻量级处理。

Hook 运行在空闲任务的执行上下文中。用户应避免在其中执行长时间阻塞、永久循环或不可控的耗时操作，否则会影响空闲任务后续处理工作的及时性。

4.13 启动阶段的中断状态

当前源码的关键中断状态变化如下：

```
main() 初始化阶段
       |
```

```
        ▼
配置内核变量与任务
        │
        ▼
ET0 = 1
TR0 = 1
EA = 1
        │
        ▼
调用 os_idle_task()
        │
        ▼
EA = 0
        │
        ▼
初始化任务、资源与 Hook
        │
        ▼
调用 hrtos_main()
        │
        ▼
OS_CFG_LOCK = 1
EA = 1
        │
        ▼
进入永久空闲循环
```

这里有一个重要的实现特点：`main()` 在调用 `os_idle_task()` 之前已经执行 `EA = 1`，而 `os_idle_task()` 入口随后再次执行 `EA = 0`。因此，不能将整个启动过程描述为从头到尾始终关闭中断。

同时，Timer0 在 `main()` 中已经启动，之后才进入 `os_idle_task()`。如果 Timer0 中断恰好在这两个阶段之间发生，具体行为取决于 Timer0 ISR 对相关状态的处理方式。需要结合 ISR 源码验证该阶段是否存在依赖特定初始化顺序的边界条件。

4.14 启动约束与注意事项

4.14.1 内存初始化范围

内部 RAM 汇编清零与 `os_memset()` 清零属于不同的操作，必须分别确认其地址空间和覆盖范围。

尤其需要核对 `os_memset(&OS_PROCESS_OK[0], 0, 0x01FF)` 的目标区域是否与其他变量、栈区或特殊存储区重叠。

4.14.2 任务栈布局

`SP = OS_JINCHENG_JILU_SP[0] - 2` 与 HRTOS 的任务初始栈布局相关。修改任务创建函数、汇编切换代码或编译器设置时，必须重新验证首次进入任务和任务切换过程。

4.14.3 Timer0 独占约定

当前系统使用 Timer0 作为系统时基，并开启 Timer0 中断。应用程序不应在未经评估的情况下将 Timer0 同时用于其他定时功能，否则可能破坏系统 Tick、时间片管理和调度时序。

4.14.4 初始化期间的中断限制

`hrtos_main()` 在 `EA = 0` 的状态下执行。任何依赖中断持续运行才能完成的操作，都需要重新评估是否适合在该阶段执行。

4.14.5 配置锁定

`OS_CFG_LOCK` 在调用 `hrtos_main()` 前被设置为 `0`，返回后设置为 `1`。应用配置接口是否受该标志约束，应以对应 API 的实现为准。

4.14.6 空闲任务不能正常返回

`os_idle_task()` 的正常执行路径是永久循环。如果该函数意外返回，`main()` 将继续执行其后续代码，而不是重新启动操作系统。因此，空闲任务必须保持其设计的长期运行行为。

4.15 启动问题排查

4.15.1 系统无法进入正常运行状态

检查内部 RAM 初始化、数据区域清理、基础任务创建、初始栈指针、Timer0 配置以及中断入口是否正确。

4.15.2 用户任务未执行

检查 `hrtos_main()` 是否正常返回、任务创建接口是否成功、任务状态与优先级是否符合预期，以及 Timer0 中断和调度路径是否正常。

4.15.3 初始化阶段卡住

优先检查 `hrtos_main()` 是否调用了依赖中断的延时、同步或驱动接口。当前实现会在调用该函数前关闭总中断，因此相关等待可能无法完成。

4.15.4 任务删除后未完成清理

检查普通任务是否进入 `DEAD` 状态，以及快速任务删除标志是否正确置位、清除；同时确认空闲循环是否持续获得执行机会。

4.15.5 系统运行时序异常

检查 Timer0 的计数时钟、重载值、工作模式、ISR 执行路径及任务切换实现。仅确认 `TR0` 和 `ET0` 已设置，并不能证明系统 Tick 周期一定准确。

4.16 总结

HRTOS 的启动流程由 `main()` 与 `os_idle_task()` 协同完成。

`main()` 负责底层内存处理、内核初始变量设置、基础任务创建、初始栈设置以及 Timer0 启动。`os_idle_task()` 负责初始化任务和资源状态、执行用户初始化入口、锁定配置，并进入永久空闲循环，持续处理任务清理请求与 Idle Hook。

整个过程依赖明确的初始化顺序、中断状态约定和 8051 上下文切换机制。维护或移植启动代码时，应以实际内核实现为依据，重点验证内存覆盖范围、初始栈布局、Timer0 时基和初始化期间的中断行为。

Task Management

HRTOS 任务管理

5.1 模块介绍

任务管理模块负责标准任务的创建与注册、任务状态管理、优先级调整、任务挂起与恢复、任务删除以及任务栈空间管理，并与内核调度器协同完成任务切换。

HRTOS 支持标准任务和快速任务两种任务执行机制。标准任务使用统一的任务控制块和等待机制，快速任务使用独立的上下文保存区域及延时控制变量。中断处理程序由独立的中断管理机制控制，不属于标准任务 ID 范围。

5.2 主要职责

- 任务创建与注册
- 任务状态查询与转换
- 任务优先级查询与调整
- 任务挂起、恢复与主动让出 CPU
- 任务删除与退出
- 任务栈空间分配与释放
- 任务上下文保存与恢复
- 任务等待状态及相关资源的清理
- 快速任务的独立控制与调度协同

5.3 任务类型

5.3.1 标准任务

标准任务使用任务 ID `0-15`，最多支持 16 个标准任务。任务优先级范围为 `0-7`，使用统一任务控制块 `OS_TCB`，并通过标准任务上下文保存区域保存任务现场。

标准任务的主要控制数据包括：

- `OS_PROCESS_OK`：任务状态及配置编码
- `OS_JINCHENG_SP`：标准任务 SP 记录

- `OS_JINCHENG_JILU_SP`：标准任务 SP 备份
- `OS_TASK_CONTEXT`：标准任务上下文保存区域
- `OS_TASK`：标准任务控制块数组

  任务栈空间通过位图进行分配和管理。

5.3.2 快速任务

快速任务使用 ID `16-17`，最多支持 2 个快速任务，调度优先级为 8。

快速任务使用独立的上下文保存区域和延时控制变量，主要控制数据包括：

- `OS_KUAI_PROCESS_A`、`OS_KUAI_PROCESS_B`：快速任务控制标志
- `OS_KUAI_SP`：快速任务 SP 记录
- `OS_FAST_CONTEXT`：快速任务上下文保存区域
- `OS_WAIT_DI2`：快速任务延时控制数组
- `OS_SP_KUAI_BEI`、`OS_SP_KUAI_BEI2`：快速任务 SP 相关记录

  快速任务与标准任务采用不同的上下文保存机制，不使用标准任务的统一等待机制。

5.3.3 中断处理程序

中断处理程序由中断管理机制控制，不占用标准任务 ID `0-15` 或快速任务 ID `16-17`。

中断处理使用独立的中断栈和现场保护区域，并通过中断管理接口完成中断退出及相关调度处理。

5.4 主要文件

5.4.1 源文件

文件	主要职责
`Src/kernel/task_create.c`	创建并注册任务
`Src/kernel/task_cleanup.c`	清理任务状态及相关资源
`Src/task/task_delete.c`	发起任务删除
`Src/task/task_exit.c`	当前任务退出
`Src/task/task_is_valid.c`	检查任务 ID 是否有效
`Src/task/task_resume.c`	恢复挂起任务
`Src/task/task_scheduler.c`	任务调度相关处理
`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`	挂起指定任务
`Src/task/task_suspend_self.c`	当前任务主动挂起
`Src/task/task_yield.c`	主动让出 CPU

文件	主要职责
`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`	锁定任务注册
`Src/task_priority/unlock.c`	解除任务注册锁定
`Src/task_priority/lock_query.c`	查询任务注册锁定状态

5.4.2 头文件

文件	主要职责
`Inc/task.h`	任务管理 API 声明
`Inc/config.h`	任务配置及任务控制块定义
`Inc/hrtos_internal.h`	内核内部变量与接口声明

5.5 核心数据结构

5.5.1 OS_TCB

`OS_TCB` 是标准任务的任务控制块，定义如下：

```
typedef struct {
    u8 base_prio;   /* 基础优先级 */
    u8 cur_prio;    /* 当前优先级 */
    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;  /* 等待计时 */
} OS_TCB;
```

`OS_TCB` 不包含任务 ID 字段。任务 ID 通过任务在 `OS_TASK` 数组中的索引表示。

主要字段说明：

- `base_prio`：任务基础优先级。
- `cur_prio`：任务当前优先级，可用于记录优先级调整后的值。
- `state`：任务状态。
- `wait_type`：任务等待类型。
- `wait_flag`：等待结果标志。
- `wait_obj`：任务等待的对象 ID。
- `wait_tick`：等待计时数据。

5.5.2 任务状态控制数组

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

`OS_PROCESS_OK` 每个元素对应一个标准任务，采用位字段保存任务状态和配置：

位	含义
bit 0	状态标志
bit 1–3	优先级级别
bit 4–6	栈大小
bit 7	保留位

任务状态控制数组与 `OS_TCB` 配合使用，分别承担任务运行控制及任务状态、等待信息的管理职责。

5.5.3 栈指针记录

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

- `OS_JINCHENG_SP`：标准任务 SP 记录。
- `OS_JINCHENG_JILU_SP`：标准任务 SP 备份记录。

两组数组均按标准任务数量配置，每个任务占用一个字节的记录空间。

5.5.4 上下文保存区域

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

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

标准任务与快速任务分别使用独立的上下文保存区域。

区域	配置容量	总空间
`OS_TASK_CONTEXT`	16 × 13 字节	208 字节
`OS_FAST_CONTEXT`	2 × 5 字节	10 字节

上下文区域用于保存相应任务切换机制所需的现场数据。

5.5.5 快速任务控制数据

```
volatile bit OS_KUAI_PROCESS_A;
volatile bit OS_KUAI_PROCESS_B;

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];
```

快速任务控制标志分别对应两个快速任务。SP 相关数组保存快速任务的栈指针及相关记录。

5.6 核心函数

5.6.1 os_task_create()

位置： `Src/kernel/task_create.c`

功能： 创建并注册任务。

该接口负责检查创建参数、处理任务注册、分配所需栈空间、初始化任务运行现场，并建立任务对应的内核控制信息。

任务创建涉及以下主要步骤：

- 检查任务创建参数。
- 检查任务注册状态。
- 搜索可用栈空间。
- 更新栈空间分配位图。
- 初始化任务 SP。
- 初始化任务入口及相关上下文。
- 更新任务控制信息。
- 完成任务注册。

标准任务与快速任务采用不同的上下文保存机制，具体初始化操作由相应任务类型的处理逻辑完成。

5.6.2 os_task_cleanup()

位置： `Src/kernel/task_cleanup.c`

功能： 清理任务及其相关资源。

任务清理涉及任务状态、栈空间、上下文和等待信息等内容。任务删除后，需要完成对应资源的清理，避免已删除任务继续参与调度或残留资源占用。

清理内容包括：

- 检查待清理任务。
- 处理任务栈空间。
- 更新栈空间分配位图。
- 清理任务上下文及控制信息。
- 清理任务等待状态。
- 处理任务与内核资源之间的关联。

5.6.3 任务状态管理

位置： `Src/task/task_state.c`

功能： 查询任务状态。

任务状态查询用于获取指定任务当前的状态信息，为任务监控和应用层控制提供依据。

5.6.4 os_task_suspend()

位置： `Src/task/task_suspend.c`

功能： 挂起指定任务。

任务挂起后不再作为普通就绪任务参与调度。恢复操作由 `os_task_resume()` 完成。

5.6.5 os_task_resume()

位置： `Src/task/task_resume.c`

功能： 恢复已挂起的任务。

任务恢复后重新具备参与调度的条件，具体执行时机由调度器决定。

5.6.6 os_task_delete()

位置： `Src/task/task_delete.c`

功能： 发起任务删除。

任务删除接口负责发起删除处理。任务状态和相关资源的最终清理由内核任务清理机制完成。

5.6.7 os_task_exit()

位置： `Src/task/task_exit.c`

功能： 当前任务主动退出。

当前任务退出后，由内核调度及任务清理机制完成后续处理。

5.6.8 os_task_yield()

位置： `Src/task/task_yield.c`

功能： 当前任务主动让出 CPU。

该接口触发调度处理，使调度器有机会选择其他符合运行条件的任务。

5.6.9 任务优先级管理

设置优先级： `Src/task_priority/task_set_priority.c`

用于调整指定任务的优先级，并更新相应任务控制信息。

获取优先级： `Src/task_priority/task_get_priority.c`

用于获取指定任务的优先级信息。

任务注册锁：

`Src/task_priority/lock.c`、`Src/task_priority/unlock.c`、`Src/task_priority/lock_query.c`

用于控制和查询任务注册锁定状态。该机制用于任务注册控制，不应与任务优先级调整混为一谈。

5.7 栈空间管理

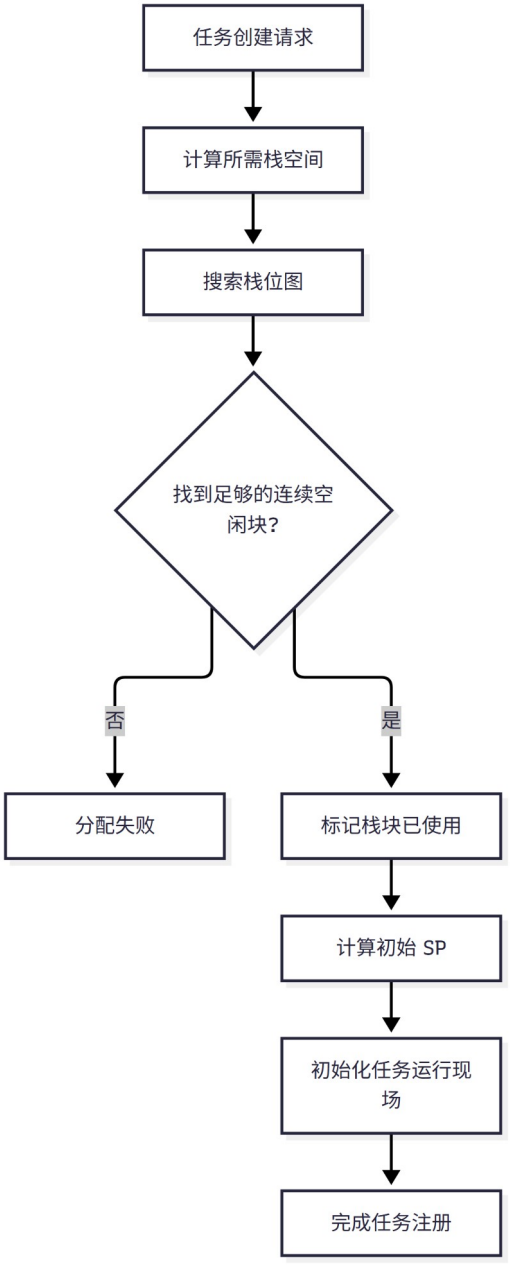
5.7.1 用户任务栈区域

当前内存布局中，用户任务栈区域为：

项目	地址或容量
起始地址	`0x42`
结束地址	`0x7F`
总容量	62 字节
分配块大小	2 字节
可分配块数	31 块
位图空间	4 字节，位于 XDATA

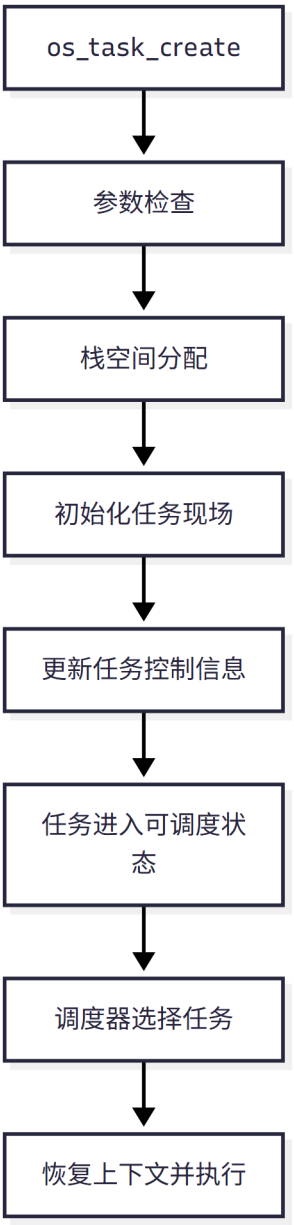
栈空间管理使用位图记录分配状态。分配时搜索连续空闲块，满足任务所需空间后标记相应块；释放时清除对应分配标记。

5.7.2 栈空间分配流程

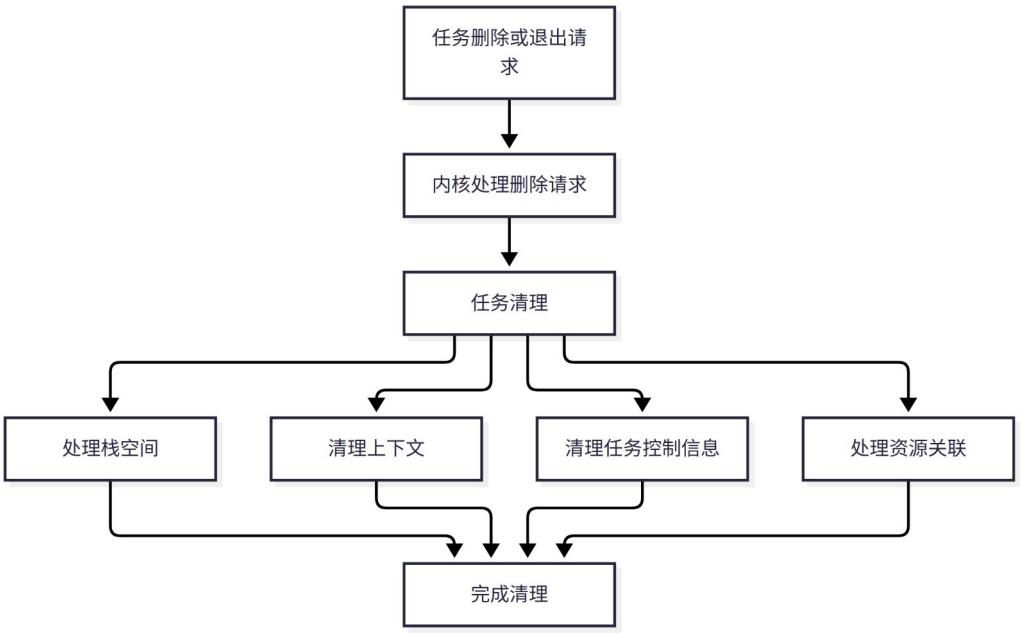

任务创建请求
计算所需栈空间
搜索栈位图
找到足够的连续空闲块?
否
是
分配失败
标记栈块已使用
计算初始 SP
初始化任务运行现场
完成任务注册

5.8 任务生命周期

5.8.1 创建与运行



5.8.2 删除与清理



5.8.3 状态转换

标准任务的生命周期涉及创建、就绪、运行、等待、挂起和删除等状态。等待任务可在等待条件满足或等待处理结束后重新进入可调度状态；挂起任务通过恢复接口重新进入可调度流程。

任务状态查询的具体返回值及内部状态编码，应与对应 API 的实现保持一致。

5.9 设计特点

5.9.1 静态资源布局

任务控制数组、任务上下文保存区域和任务栈管理数据采用固定布局，核心资源容量由编译配置决定。

5.9.2 独立上下文机制

标准任务与快速任务分别使用独立的上下文保存区域，以适配不同的任务切换路径。

5.9.3 统一任务控制

标准任务通过 `OS_TCB` 管理任务状态、优先级及等待信息，并配合 `OS_PROCESS_OK` 完成任务控制。

5.9.4 栈空间复用

任务栈空间通过位图进行分配和释放，使已清理任务占用的栈块能够重新用于后续任务注册。

5.10 系统约束

- 标准任务最多 16 个，任务 ID 为 `0-15`。
- 快速任务最多 2 个，任务 ID 为 `16-17`。
- 标准任务优先级范围为 `0-7`。
- 快速任务调度优先级为 `8`。
- 中断处理使用独立的中断管理机制。
- 标准任务和快速任务使用不同的上下文保存区域。
- 用户任务栈空间受固定 RAM 区域及位图分配结果限制。
- 任务注册锁用于控制任务注册过程。

5.11 内存布局

5.11.1 内部 RAM

地址范围	用途
`0x15`	中断嵌套临时变量 `OS_IRQ_NEST_TEMP`
`0x30-0x34`	中断保护、SP 及内核临时变量
`0x35-0x39`	系统堆栈保护区域
`0x3A-0x41`	中断堆栈区域
`0x42-0x7F`	用户任务栈区域
`0xDB-0xE0`	中断现场保护区域
`0xE1-0xE2`	Timer0 计数器相关数据
`0xE3-0xE7`	当前任务、前一任务、调度源、时间片及临界区嵌套变量

上述地址对应当前内核布局，修改固定地址、数组长度或相关编译配置时，需要同步检查是否发生内存区域重叠。

5.11.2 XDATA

系统全局数据池的固定范围为 `0x0200-0x03FF`，总容量 512 字节。当前布局中，任务管理相关区域包括：

起始地址	数据对象	容量
`0x0200`	`OS_PROCESS_OK`	16 字节

起始地址	数据对象	容量
0x0210	OS_JINCHENG_SP	16 字节
0x0220	OS_KUAI_SP	2 字节
0x0222	OS_FAST_CONTEXT	10 字节
0x022C	OS_TASK_CONTEXT	208 字节
0x02FC	OS_MEMORY	4 字节
0x0300	OS_JINCHENG_JILU_SP	16 字节
0x0336	OS_TASK	128 字节

任务管理相关数组与其他内核全局对象共同占用系统 XDATA 数据池，固定地址及偏移定义集中在内核内部配置中。

Scheduler Design

HRTOS 调度器设计

6.1 模块介绍

HRTOS 调度器负责根据任务优先级、任务状态和调度条件选择运行任务，并完成任务调度与上下文切换。

调度器支持基于优先级的抢占式调度、时间片管理、高速任务调度和中断相关调度处理，并支持 HRTOS 与 MYOS 两种调度模式的切换，以满足不同应用场景下的任务执行需求。

6.2 主要职责

- **优先级调度**：根据任务优先级选择可运行任务。
- **抢占式调度**：在满足调度条件时切换当前运行任务。
- **时间片管理**：维护时间片计数并处理时间片到期事件。
- **高速任务调度**：支持高速任务的独立调度与快速切换。
- **中断调度处理**：处理来自中断服务程序的调度请求。
- **调度模式管理**：支持 HRTOS 与 MYOS 调度模式之间的运行时切换。

6.3 主要文件

6.3.1 源文件

文件	主要职责
`Src/kernel/scheduler.c`	调度器模式选择与切换
`Src/kernel/scheduler_dispatch.c`	任务分发机制
`Src/kernel/scheduler_mode.c`	调度模式配置与切换处理
`Src/kernel/task_create.c`	任务注册及调度信息初始化
`Src/task/task_scheduler.c`	任务调度逻辑
`Src/task/task_yield.c`	任务主动让出 CPU
`Src/interrupt/schedule_request.c`	中断上下文中的调度请求

6.3.2 头文件

文件	主要职责
`Inc/kernel.h`	调度器及内核 API 声明
`Inc/config.h`	调度级别、任务数量等配置定义

6.4 数据结构

6.4.1 调度级别定义

调度级别定义于 `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
```

调度级别说明：

级别	含义
0–7	普通任务优先级
8	高速任务级别
9	中断级
10	中断嵌套级

普通任务、高速任务和中断处理具有不同的执行层级，调度器根据相应规则完成任务选择及执行切换。

6.4.2 任务调度信息

位于 `Src/kernel/os_core.c` 的调度相关变量：

```
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;       /* 当前时间片计数 */
volatile unsigned char idata OS_DISPATCH_ID;   /* 分发触发源 */
volatile unsigned char xdata OS_TIME_ONCE;     /* 时间片计数配置 */
```

变量	说明
`OS_CURRENT_TASK`	记录当前运行任务
`OS_PREV_TASK`	记录前一个运行任务
`OS_TIME_XY`	维护当前时间片相关计数
`OS_DISPATCH_ID`	记录任务分发相关信息

变量	说明
`OS_TIME_ONCE`	保存时间片计数配置

6.4.3 任务状态数组

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

`OS_PROCESS_OK[id]` 使用位字段保存任务相关状态信息：

位范围	含义
bit 0	任务状态标志
bit 1–3	任务优先级
bit 4–6	栈大小信息

6.4.4 高速任务标志

```
volatile bit OS_KUAI_PROCESS_A;  /* 高速任务 0 激活标志 */
volatile bit OS_KUAI_PROCESS_B;  /* 高速任务 1 激活标志 */
```

两个标志分别用于记录高速任务 A、B 的激活状态，配合高速任务调度机制使用。

6.5 核心函数

6.5.1 os_set_scheduler()

位置： `Src/kernel/scheduler.c`

功能： 在 HRTOS 与 MYOS 调度模式之间切换。

参数：

参数	含义
`id = 1`	切换到 HRTOS 模式
`id = 0`	切换到 MYOS 模式

返回值：

返回值	含义
`1`	切换成功
`-1`	当前条件下无法切换

实现逻辑：

```
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;
}
```

切换处理：

- 切换到 MYOS 模式时，设置 `OS_INTERRUPT_T0`，并备份当前时间片配置。
- 切换到 HRTOS 模式时，恢复相应的时间片状态与配置。
- 当请求的模式与当前模式不匹配时，返回 `-1`。

6.5.2 os_scheduler_mode_switch()

位置： `Src/kernel/scheduler_mode.c`

功能： 在实时调度模式与通用调度模式之间切换。

参数：

参数	含义
`id = 1`	实时模式
`id = 0`	通用模式

该接口用于调度模式相关控制，与 `os_set_scheduler()` 一起构成调度模式管理机制。

6.5.3 os_schedule_request()

位置： `Src/interrupt/schedule_request.c`

功能： 通过触发 Timer0 中断请求调度处理。

实现逻辑：

```
char os_schedule_request(char id)
{
    id = id;                    /* 保留参数 */
    OS_SCHED_REASON = 1;        /* 设置调度触发标志 */
    TF0 = 1;                    /* 触发 Timer0 中断 */
    return 1;
}
```

处理流程：

- 设置 `OS_SCHED_REASON`，标记存在调度请求。
- 设置 `TF0`，触发 Timer0 中断。
- 由相应的中断及调度处理流程完成后续任务调度。

该函数用于发起调度请求，任务选择与上下文切换由后续调度流程完成。

6.5.4 os_task_yield()

位置： `Src/task/task_yield.c`

功能： 主动让出当前任务的 CPU 执行机会。

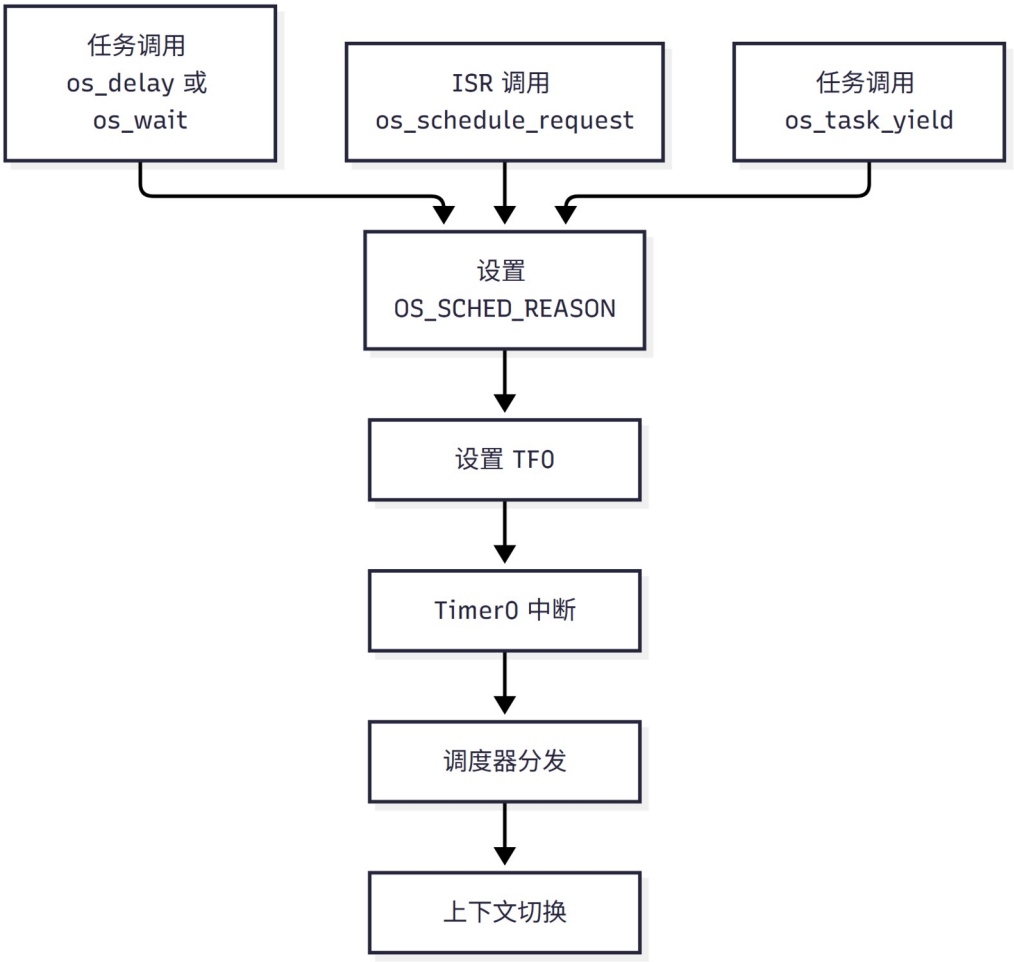
实现方式：

- 设置调度触发条件。
- 请求调度器重新选择运行任务。
- 根据调度规则执行相应的任务切换。

主动让出 CPU 后，后续运行任务由调度器依据当前任务状态和优先级规则决定。

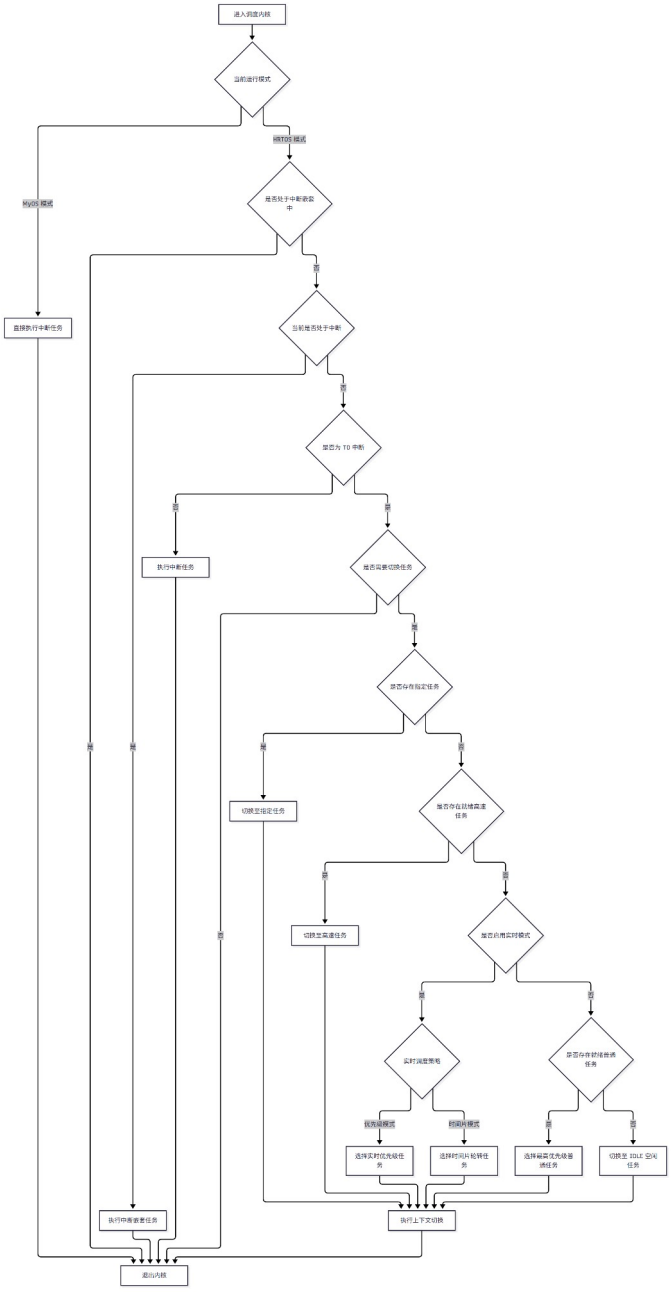
6.6 调用关系

6.6.1 调度触发流程



该流程展示任务阻塞、中断调度请求及主动让出 CPU 等操作进入调度处理的基本路径。

6.6.2 优先级选择流程



调度器根据高速任务、普通任务及空闲任务的调度规则选择后续运行对象，并在需要时完成上下文切换。

6.7 生命周期

6.7.1 调度器初始化

系统启动阶段的调度相关处理包括：

- 进入 `os_idle_task()`。
- 初始化 `OS_TIME_XY` 时间片计数。

- 将 `OS_CURRENT_TASK` 设置为 `0`，表示空闲任务。
- 清除 `OS_SCHED_REASON` 调度触发标志。
- 启用中断。
- 启动 Timer0 时钟周期。

初始化完成后，系统进入正常的任务调度与运行阶段。

6.7.2 调度周期

一次典型的时钟驱动调度过程包括：

- **时钟中断：** Timer0 ISR 触发。
- **时间片计数：** 更新 `OS_TIME_XY`。
- **时间片检查：** 判断 `OS_TIME_XY` 是否达到 `OS_TIME_ONCE`。
- **优先级扫描：** 根据调度规则检查可运行任务。
- **任务选择：** 选择符合当前调度条件的任务。
- **上下文切换：** 保存当前任务上下文并恢复目标任务上下文。
- **任务执行：** 目标任务继续运行。

时间片到期、任务阻塞、任务唤醒和主动让出 CPU 等事件也可能触发调度处理。

6.8 设计原则

6.8.1 基于优先级的抢占

- 普通任务优先级范围为 `0-7`。
- 高速任务使用级别 `8`。
- 中断级别为 `9-10`。
- 调度器根据任务优先级及当前执行条件选择运行任务。
- 高优先级任务可以在满足调度条件时抢占低优先级任务。

6.8.2 时间片模式

- 使用 `OS_TIME_ONCE` 配置时间片计数。
- 使用 `OS_TIME_T0` 配置时钟相关参数。
- 时间片机制参与任务调度与调度触发。
- 可根据系统配置使用相应的时间片调度策略。

时间片计数、硬件定时器周期和任务实际运行时间是不同概念，具体换算关系由系统时钟及定时器配置决定。

6.8.3 高速任务优化

高速任务通过专门的调度及上下文处理机制降低切换开销，相关设计包括：

- 减少上下文保存深度。

- 使用独立的延时机制。
- 使用专用寄存器组。
- 降低高速任务调度开销。

  高速任务数量上限由 `OS_FAST_TASK_MAX` 控制，当前配置为两个。

6.8.4 双调度器支持

- **HRTOS 模式：** 使用 HRTOS 对应的调度机制。
- **MYOS 模式：** 使用 MYOS 对应的调度机制。
- **运行时切换：** 通过 `os_set_scheduler()` 完成模式切换。

  模式切换涉及时间片配置及相关调度状态的调整。

6.9 约束

- 普通任务数量上限由 `OS_PROCESS_MAX` 控制，当前配置为 `16`。
- 高速任务数量上限由 `OS_FAST_TASK_MAX` 控制，当前配置为 `2`。
- 调度级别上限由 `OS_SCHED_LEVEL_MAX` 控制，当前配置为 `10`。
- 高速任务使用专门的调度机制，其等待及 IPC 接口使用范围受相应实现约束。
- 互斥锁优先级继承属于资源同步机制，不属于调度器本身的调度算法。
- 在没有时间片轮转机制参与时，同优先级任务之间的运行顺序取决于调度器的具体实现。

6.10 调度算法

6.10.1 就绪任务选择

调度器按照系统规定的执行层级和任务调度规则选择运行对象，主要包括：

- 处理中断相关执行层级。
- 检查高速任务。
- 扫描普通任务。
- 在没有其他合适任务运行时，执行空闲任务。

高速任务、普通任务和中断处理具有不同的执行机制，不能将所有级别简单视为同一类普通任务优先级。

6.10.2 任务状态位

`OS_PROCESS_OK[id]` 的编码如下：

位范围	含义
bit 0	任务状态标志
bit 1–3	优先级级别
bit 4–6	栈大小信息

该数组用于保存任务相关状态及调度信息。

6.10.3 上下文切换触发条件

调度处理可由以下事件触发：

- **时间片到期：** `OS_TIME_XY` 达到 `OS_TIME_ONCE`。
- **主动让出 CPU：** 调用 `os_task_yield()`。
- **任务阻塞：** 任务通过 `os_wait()` 等等待接口进入等待状态。
- **中断请求调度：** ISR 调用 `os_schedule_request()`。
- **任务唤醒：** `wake_task()` 等相关机制使任务重新具备运行条件。

调度请求产生后，内核根据当前执行上下文和调度条件完成任务选择，并在需要时执行上下文切换。

6.11 总结

HRTOS 调度器整合了优先级调度、时间片管理、高速任务调度、中断相关调度处理及调度模式切换等机制。

调度器通过统一的调度处理流程协调任务阻塞、唤醒、主动让出 CPU 和时钟驱动调度等场景，并配合上下文切换机制实现任务执行权的转移。其设计重点是在 8051 有限的硬件资源下，兼顾任务调度效率、实时响应能力与系统功能完整性。

State Machine

HRTOS 状态机

7.1 概述

HRTOS 的任务状态管理由任务控制块、任务就绪标志、当前任务与前一任务信息，以及等待上下文共同构成。

系统通过任务状态记录任务所处的生命周期阶段，通过就绪标志控制标准任务是否参与调度，通过等待上下文记录任务正在等待的条件及唤醒结果。高速任务采用独立的运行标志和延时计数，不完全遵循标准任务的等待处理路径。

理解 HRTOS 状态机时，需要区分以下几个概念：

- **任务内部状态**：由 `READY`、`WAIT`、`RUNNING`、`SUSPEND` 和 `DEAD` 等宏定义表示。
- **任务调度资格**：标准任务通过 `OS_PROCESS_OK[id]` 的最低位判断是否就绪。
- **任务状态查询结果**：由 `os_task_get_state()` 返回，采用独立于内部状态宏的编码。
- **等待上下文**：记录等待类型、等待对象、剩余等待时间和唤醒结果。
- **高速任务运行状态**：通过高速任务运行标志及对应的栈状态信息进行判断。

这些信息相互关联，但并不是同一个变量，也不能相互替代。

7.2 任务状态定义

HRTOS 配置文件中定义了以下任务状态：

```
#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 */
```

7.2.1 READY：就绪态

任务具备继续执行的条件，可以参与正常调度。

对于标准任务，系统使用 `OS_PROCESS_OK[id]` 的最低位表示其调度资格。最低位为 `1` 时，表示任务具有就绪资格；最低位为 `0` 时，任务不具备这一资格。

```
OS_PROCESS_OK[id] |= 1;   /* 设置就绪位 */
OS_PROCESS_OK[id] &= 0xFE; /* 清除就绪位 */
```

这两个操作只修改最低位，不应将 `OS_PROCESS_OK[id]` 的整个字节简单理解为任务状态值。

7.2.2 WAIT：等待态

任务因延时或等待某种资源、消息、事件等条件而暂时不能继续执行。

标准任务进入等待时，`os_wait()` 会记录等待上下文，并执行：

```
OS_TASK[tid].state = WAIT;
OS_PROCESS_OK[tid] &= 0xFE;
```

任务被唤醒后，`wake_task()` 会设置唤醒结果、清理等待上下文、将任务状态设为 `READY`，并恢复其就绪资格。

高速任务的延时等待有所不同：它不通过上述标准任务等待上下文建立延时，而是清除对应的高速任务运行标志，并将等待时间写入 `OS_WAIT_DI2[]`。

7.2.3 RUNNING：运行态

`RUNNING` 表示任务正在运行。不过，当前提供的 `os_task_get_state()` 并不是直接检查 `OS_TASK[id].state == RUNNING`，而是根据任务编号、当前任务编号、前一任务编号，以及相应的任务运行信息判断查询结果。

因此，不能仅通过检查 TCB 中的 `state` 字段，就认定能够完整还原 `os_task_get_state()` 的查询结果。

7.2.4 SUSPEND：挂起态

`SUSPEND` 是配置文件中定义的任务状态。状态查询函数在标准任务不满足运行、前一任务和就绪条件时，会返回表示挂起的查询结果。

但目前提供的源码没有包含专门的挂起与恢复函数，因此不能仅凭这些文件确定挂起状态的全部进入条件、恢复条件，以及挂起与等待状态之间的交互规则。

7.2.5 DEAD：删除态

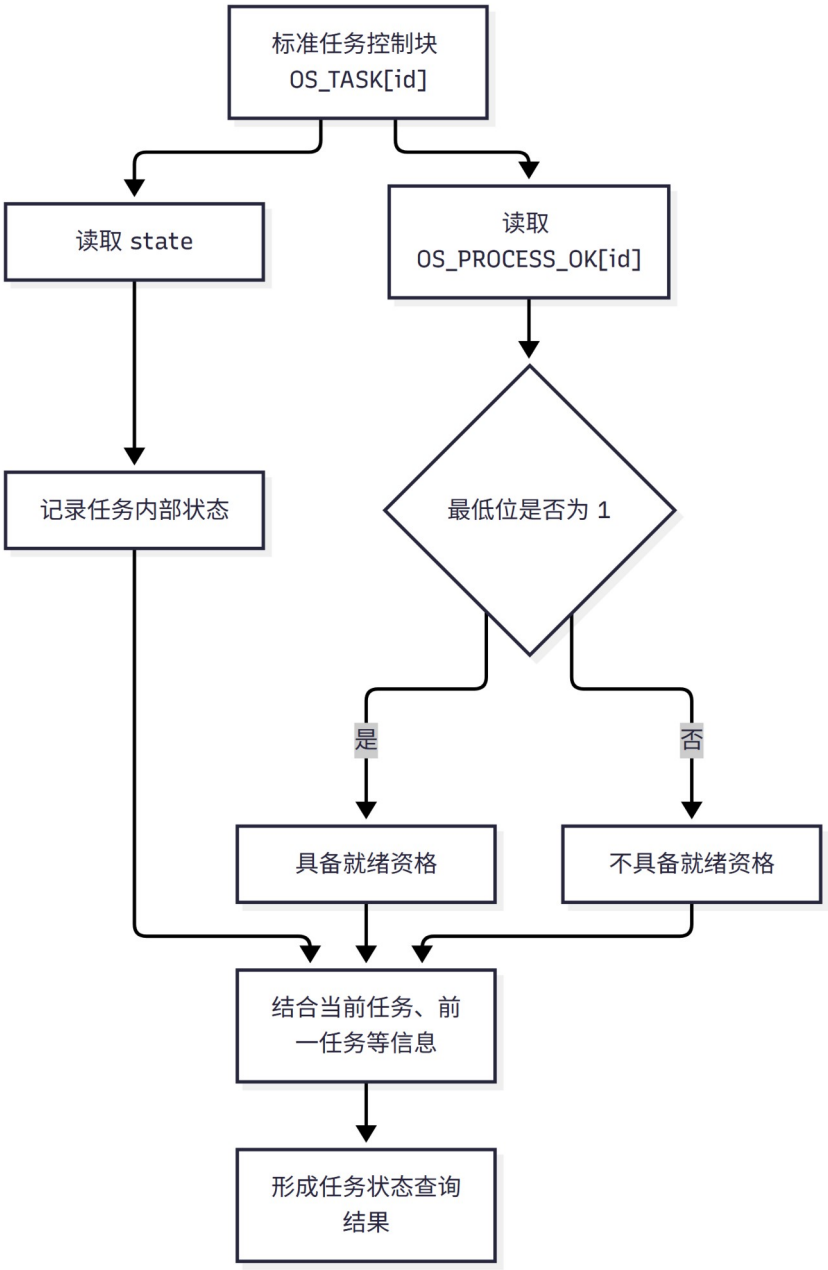
`DEAD` 表示任务处于删除状态。

`wake_task()` 在清理等待对象信息后，会检查任务的就绪标志和内部状态。如果 `OS_PROCESS_OK[tid] == 0` 或任务状态为 `DEAD`，函数便提前返回，不再执行后续的唤醒操作。

需要注意，`wake_task()` 并不负责完整的任务删除流程。任务删除、栈回收以及任务槽位重新利用的完整机制，还需要结合任务创建、删除和清理相关源码进行分析。

7.3 任务状态与调度资格

标准任务的内部状态和就绪标志需要结合起来理解。



这张图表示的是状态信息的判读关系，并不代表所有状态转换都由一个统一函数完成。

在当前实现中：

- `os_wait()` 负责设置标准任务的等待上下文、将其状态设为 `WAIT`，并清除就绪位。
- `wake_task()` 负责清理等待上下文，将符合唤醒条件的任务恢复为 `READY`，并设置就绪位。
- `os_task_get_state()` 根据不同任务类型和运行信息返回状态查询结果。

因此，任务状态、调度资格和查询结果应当分别记录、分别理解。

7.4 任务状态查询接口

7.4.1 函数原型

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

该接口根据任务编号查询任务状态。

7.4.2 参数说明

参数	说明
`id`	任务编号。标准任务编号为 `0` 至 `15`，高速任务编号为 `16` 和 `17`。

当任务编号大于 `OS_PROCESS_MAX + 1` 时，函数返回 `-1`。

7.4.3 返回值

返回值	含义
`0`	任务已删除或尚未创建
`1`	当前正在运行的任务
`2`	就绪状态
`3`	挂起状态
`4`	最近一次被切出的任务
`-1`	任务编号超出允许范围

这里的返回值是查询接口的专用编码，不能直接与 `READY`、`WAIT`、`RUNNING`、`SUSPEND`、`DEAD` 的宏定义进行数值比较。

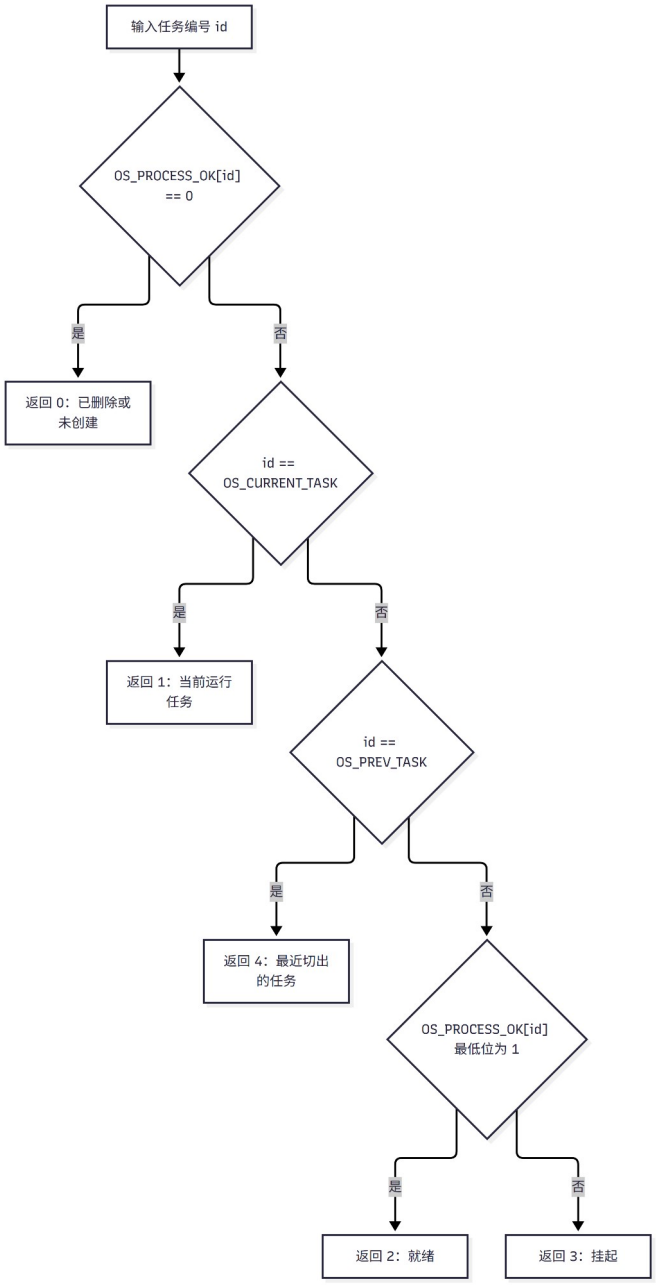
例如：

```
#define READY 0
#define RUNNING 2
```

虽然 `READY` 的内部编码为 `0`，但 `os_task_get_state()` 返回 `2` 表示就绪，而不是运行。这是两套不同的编码体系。

7.4.4 标准任务的查询流程

对于编号小于 `OS_PROCESS_MAX` 的标准任务，函数按照以下顺序判断：



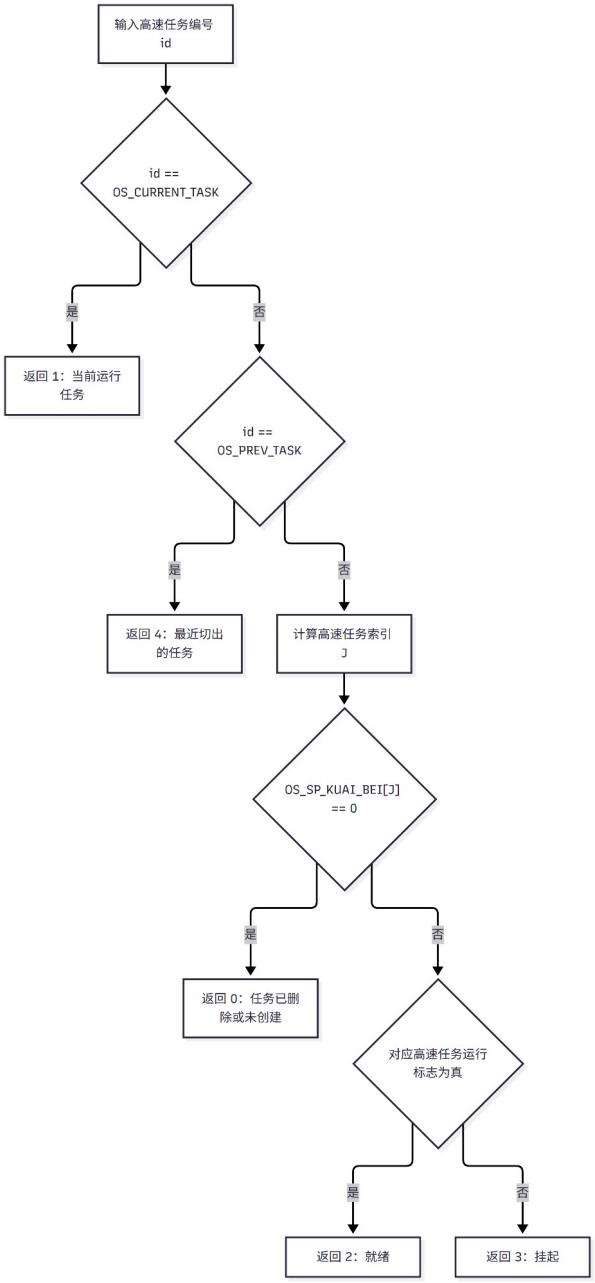
该判断顺序具有实际意义。

当多个条件同时成立时，函数优先返回代码中先检查的条件。例如，标准任务的就绪标志为零时，即使其编号与当前任务编号相同，函数也会优先返回 `0`。

7.4.5 高速任务的查询流程

高速任务的编号范围为 `OS_PROCESS_MAX` 至 `OS_PROCESS_MAX + 1`，即当前配置下的 `16` 和 `17`。

函数先检查该任务是否为当前任务或前一任务，然后根据高速任务的栈状态及运行标志判断其余状态。



高速任务 A 使用 `OS_KUAI_PROCESS_A`，高速任务 B 使用 `OS_KUAI_PROCESS_B`。

需要注意，当前函数把高速任务的非运行状态归类为就绪或挂起结果，但这不意味着高速任务与标准任务具有完全相同的 TCB 状态管理机制。

7.4.6 使用注意事项

`os_task_get_state()` 的返回类型为 `char`，而无效编号使用 `-1` 表示。若要将其作为跨编译器、跨编译选项的稳定 API，应确认目标编译器中 `char` 的有符号性，必要时改用明确的有符号返回类型。

此外，该函数没有对每个合法编号对应的底层状态进行完整性校验。调用方应传入有效任务编号，并将返回结果视为查询时的状态快照。

7.5 等待类型

HRTOS 使用 `wait_type` 区分任务正在等待的条件。

```
#define WAIT_NONE      0
#define WAIT_DELAY     1
#define WAIT_SEM       2
#define WAIT_MUTEX     3
#define WAIT_MSG_SEND  4
#define WAIT_MSG_RECV  5
#define WAIT_EVENT     6
#define WAIT_MAILBOX   7
```

各类型的含义如下：

等待类型	含义
`WAIT_NONE`	当前没有等待
`WAIT_DELAY`	延时等待
`WAIT_SEM`	等待信号量
`WAIT_MUTEX`	等待互斥锁
`WAIT_MSG_SEND`	消息队列发送等待，通常与队列满有关
`WAIT_MSG_RECV`	消息队列接收等待，通常与队列空有关
`WAIT_EVENT`	等待事件
`WAIT_MAILBOX`	等待邮箱

这些宏定义表明系统为上述等待场景预留了类型标识，但具体类型是否被每个对应模块正确使用，仍应结合信号量、互斥锁、消息队列、事件和邮箱等实现进行确认。

7.6 等待上下文

配置文件中的 `OS_TCB` 包含以下等待相关字段：

```
u8  wait_type;
u8  wait_flag;
u8  wait_obj;
u16 wait_tick;
```

这些字段共同记录任务的等待信息。

字段	作用
`wait_type`	等待类型
`wait_flag`	等待结果或唤醒原因
`wait_obj`	等待对象编号

字段	作用
`wait_tick`	等待超时计数

其中，`wait_tick` 的实际含义需要结合系统 Tick 更新代码确认。`os_wait()` 将调用方传入的 `tick` 写入该字段，但仅凭当前文件不能确认每次 Tick 如何递减该字段、如何处理计数到期，以及是否存在特殊的零值或边界约定。

7.6.1 等待结果

```
#define WAIT_TIMEOUT  1
#define WAIT_SIGNAL   2
```

返回值	含义
`WAIT_TIMEOUT`	因超时而结束等待
`WAIT_SIGNAL`	因信号或相应等待条件满足而结束等待
`0`	等待参数无效，或当前任务已经存在等待状态

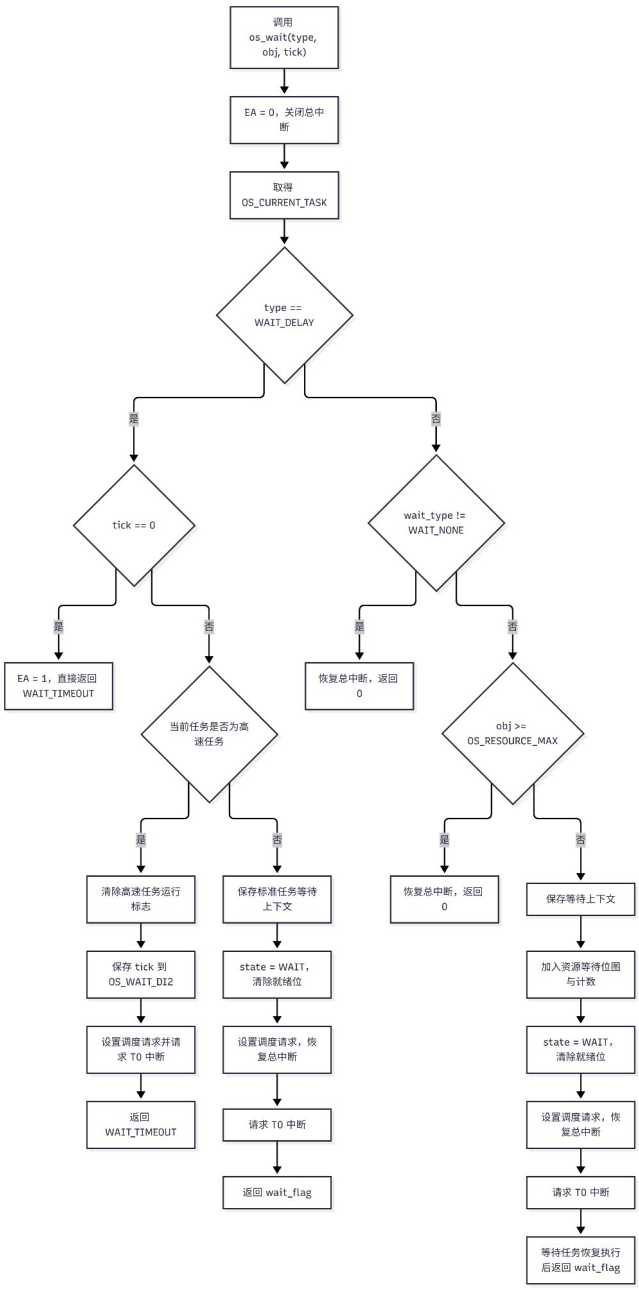
`WAIT_TIMEOUT` 和 `WAIT_SIGNAL` 是等待结果标志，不是任务状态。

例如，一个任务可以先处于 `WAIT`，在超时后由唤醒路径设置 `wait_flag = WAIT_TIMEOUT`，再恢复为 `READY`。任务重新获得执行机会后，`os_wait()` 返回等待结果。

7.7 标准任务的等待流程

标准任务调用 `os_wait()` 后，系统首先关闭总中断，取得当前任务编号，并根据等待类型进入相应处理路径。

对于非延时等待，函数还会检查当前任务是否已经存在等待类型，并验证等待对象编号是否超出资源槽位范围。



该流程图依据当前提供的 `wait.c` 整理。需要特别注意，函数中对延时等待和非延时等待的处理并不完全相同。

7.7.1 延时等待的处理

当 `type == WAIT_DELAY` 时，函数首先判断 `tick` 是否为零。

如果 `tick == 0`，函数恢复总中断并立即返回 `WAIT_TIMEOUT`，不会真正建立等待状态。

如果当前任务是标准任务，函数会写入等待上下文，将状态设为 `WAIT`，清除就绪位，设置调度请求并请求 Timer0 中断。函数随后返回 `OS_TASK[tid].wait_flag`。

如果当前任务是高速任务，则进入独立处理路径：

- 高速任务 A 清除 `OS_KUAI_PROCESS_A`，并将等待时间写入 `OS_WAIT_DI2[0]`。
- 高速任务 B 清除 `OS_KUAI_PROCESS_B`，并将等待时间写入 `OS_WAIT_DI2[1]`。
- 设置调度请求，并请求 Timer0 中断。

这一分支没有通过标准任务的 `OS_TASK[tid].state = WAIT`、等待位图及等待计数来建立延时等待。

7.7.2 资源等待的处理

对于非 `WAIT_DELAY` 类型，函数首先判断当前任务的 `wait_type` 是否为 `WAIT_NONE`。若不是，说明任务已经记录了等待类型，函数会返回 `0`。

随后检查：

```
if(obj >= OS_RESOURCE_MAX)
{
    EA = 1;
    return 0;
}
```

在当前配置下，`OS_RESOURCE_MAX` 为 `6`，因此资源对象编号必须小于 `6`。

通过检查后，函数保存等待上下文。如果类型不是 `WAIT_DELAY`，且对象编号不等于 `OS_INVALID_ID`，则执行：

```
OS_RES[obj].wait_mask |= ((u16)1 << tid);
OS_RES[obj].wait_cnt++;
```

其中：

- `wait_mask` 以位图形式记录等待任务。
- `wait_cnt` 记录等待计数。

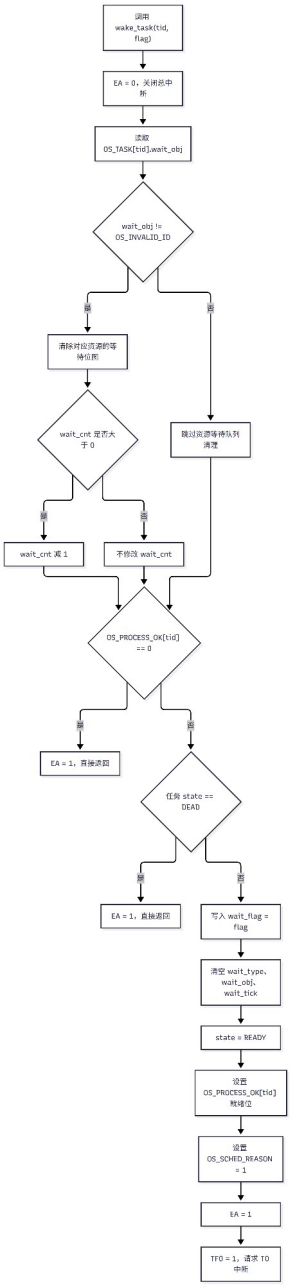
接着将任务状态设为 `WAIT`，清除其就绪位，并请求调度。

实现边界： 当前代码没有显式验证 `type` 是否属于 `WAIT_SEM` 至 `WAIT_MAILBOX` 等已定义类型。它主要根据 `WAIT_DELAY` 分支、当前等待状态及对象编号进行处理。因此，调用方应传入模块约定的合法等待类型，不应将此函数视为具有完整参数校验的通用接口。

7.8 任务唤醒流程

`wake_task()` 负责处理指定任务的等待清理、唤醒结果记录和就绪恢复。

函数的实际执行顺序如下：



7.8.1 等待队列清理

函数先读取 `OS_TASK[tid].wait_obj`。如果该值不等于 `OS_INVALID_ID`，便清除对应资源的等待位图，并在 `wait_cnt` 大于零时将其减一。

这里的清理操作发生在删除状态检查之前。因此，即使后续判断发现任务已删除或状态为 `DEAD`，前面的等待队列清理也已经执行。

当前实现没有在访问 `OS_RES[obj]` 之前显式检查 `obj < OS_RESOURCE_MAX`，也没有在函数入口检查 `tid` 是否有效。因此，该函数依赖调用方传入有效任务编号，并保证等待对象信息合法。

7.8.2 删除状态检查

等待队列清理后，函数依次检查：

```
if(OS_PROCESS_OK[tid] == 0)
{
    EA = 1;
    return;
}

if(OS_TASK[tid].state == DEAD)
{
    EA = 1;
    return;
}
```

如果任一条件成立，函数就恢复总中断并返回，不再恢复任务就绪资格。

这意味着唤醒操作并不保证任何传入任务都会被恢复为就绪态。

7.8.3 恢复任务就绪资格

通过删除状态检查后，函数依次执行：

```
OS_TASK[tid].wait_flag = flag;

OS_TASK[tid].wait_type = WAIT_NONE;
OS_TASK[tid].wait_obj  = OS_INVALID_ID;
OS_TASK[tid].wait_tick = 0;

OS_TASK[tid].state = READY;
OS_PROCESS_OK[tid] |= 1;

OS_SCHED_REASON = 1;
```

这一过程将等待结果写入任务控制块，清除等待上下文，将任务状态恢复为 `READY`，并重新设置就绪位。

最后，函数恢复总中断并设置 `TF0 = 1`，请求 Timer0 中断，以触发后续的系统处理。

需要注意，`wake_task()` 本身不检查任务是否确实处于 `WAIT` 状态，也没有检查该任务是否已经被唤醒。因此，调用方应当确保调用时机和任务编号符合相应模块的约定。

7.9 延时等待中的对象编号注意事项

`os_wait()` 的注释说明，延时等待不关联具体资源对象。但当前实现中，标准任务的延时分支仍然执行：

```
OS_TASK[tid].wait_obj = obj;
```

另一方面，`wake_task()` 只检查：

```
if(obj != OS_INVALID_ID)
```

便可能访问 `OS_RES[obj]`，而没有进一步验证对象编号是否属于有效资源范围。

因此，按照当前实现，如果标准任务延时等待时传入了非 `OS_INVALID_ID` 的对象编号，后续唤醒路径可能把这个编号当作资源对象访问。

调用约定建议： 在当前代码不变的情况下，纯延时等待应传入 `OS_INVALID_ID` 作为对象编号，并确认所有调用点遵守这一约定。更稳妥的实现方式是在延时分支中明确将 `wait_obj` 设置为 `OS_INVALID_ID`，并在 `wake_task()` 中增加任务编号和资源对象编号的边界检查。

以上是根据现有代码发现的实现边界，不代表所有现有调用点都已经触发该问题。

7.10 高速任务的延时状态管理

HRTOS 配置了两个高速任务：

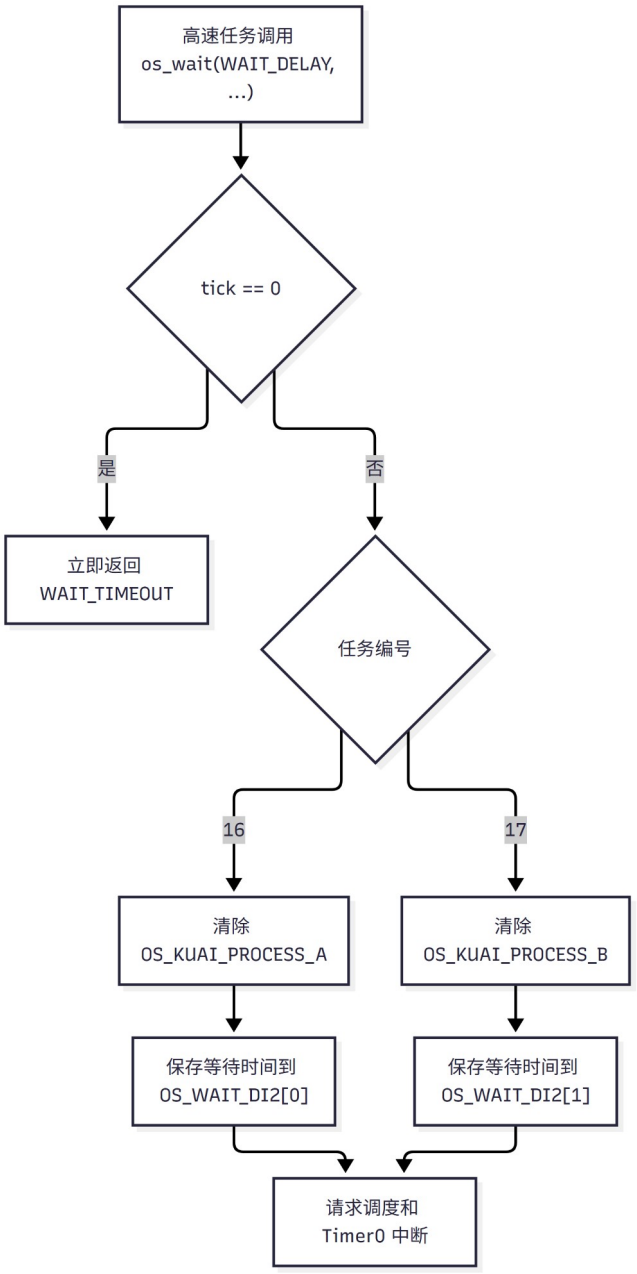
```
#define OS_PROCESS_MAX    16
#define OS_FAST_TASK_MAX  2
```

标准任务编号为 `0` 至 `15`，高速任务编号为 `16` 和 `17`。

在 `os_wait()` 的延时分支中，高速任务不使用标准任务的等待位图和等待计数，而是使用独立的变量：

高速任务	运行标志	延时计数
高速任务 A	`OS_KUAI_PROCESS_A`	`OS_WAIT_DI2[0]`
高速任务 B	`OS_KUAI_PROCESS_B`	`OS_WAIT_DI2[1]`

高速任务延时处理的核心流程是：



高速任务延时到期后如何恢复运行，需要结合 Timer0 中断处理及高速任务调度代码进一步确认。不能仅凭 `os_wait()` 推断其完整的计时与恢复过程。

7.11 状态一致性与调试

排查任务状态问题时，应同时检查任务控制块、调度资格和等待上下文，而不是只看单一字段。

7.11.1 标准任务

建议关注以下变量：

- `OS_TASK[id].state`：任务内部状态。
- `OS_PROCESS_OK[id]` 的最低位：任务是否具有就绪资格。
- `OS_TASK[id].wait_type`：是否记录了等待类型。
- `OS_TASK[id].wait_obj`：等待对象编号。
- `OS_TASK[id].wait_tick`：等待计数。
- `OS_TASK[id].wait_flag`：等待结果。
- `OS_CURRENT_TASK`：当前任务编号。
- `OS_PREV_TASK`：前一任务编号。

7.11.2 资源等待

建议同时检查：

- `OS_RES[obj].wait_mask`：资源等待位图。
- `OS_RES[obj].wait_cnt`：资源等待计数。
- 任务控制块中对应的 `wait_type` 和 `wait_obj`。
- 唤醒路径是否对同一个等待记录进行了重复清理。

`wait_mask` 与 `wait_cnt` 应当保持逻辑一致，但当前 `wake_task()` 会在 `wait_cnt` 大于零时直接递减计数，并未确认被清除的任务位原先是否确实存在。因此，如果唤醒路径被重复调用，计数和位图可能出现不一致，需要结合调用方排查。

7.11.3 状态查询

调试日志应区分内部状态宏与 `os_task_get_state()` 的返回值。例如：

```
/* 内部状态 */
OS_TASK[id].state

/* 对外查询结果 */
os_task_get_state(id)
```

两者采用不同的编码和判断逻辑，不能用同一套数字标签解释。

7.12 实现边界

根据当前提供的 `task_get_state.c`、`wait.c`、`wake_task.c` 和配置文件，可以确认：

- 任务状态宏、等待类型和等待结果标志的具体定义。
- 标准任务的等待上下文设置及唤醒清理流程。
- 标准任务就绪位的设置和清除方式。
- 高速任务延时使用独立运行标志和延时计数。
- 状态查询接口对标准任务与高速任务采用不同的判断路径。
- `wake_task()` 会在删除状态检查前清理可能关联的资源等待记录。

目前尚不能仅凭这些文件完整确认：

- 所有任务状态转换的完整集合。
- 任务挂起、恢复、删除及重新创建的完整流程。
- 每种 IPC 对象的等待和唤醒策略。
- Tick 中断如何递减等待时间及处理超时。
- 高速任务延时到期后的具体恢复流程。
- 调度器如何使用 `OS_SCHED_REASON`、`OS_CURRENT_TASK` 和 `OS_PREV_TASK` 完成实际切换。
- 所有状态变更是否都由临界区保护。

这些内容应当以相应的任务管理、调度器、Timer0 中断及 IPC 模块源码为依据补充，而不应仅凭状态宏或接口注释推断。

7.13 总结

HRTOS 的任务状态管理不是简单地依赖一个状态字段，而是结合任务内部状态、调度资格、当前任务信息和等待上下文共同完成。

标准任务通过统一等待上下文管理延时和资源等待，并在唤醒后恢复就绪资格；高速任务则对延时采用独立的处理路径。`os_task_get_state()` 为调用方提供统一的查询入口，但其返回编码与内部状态宏不同。

在系统手册中，应始终区分**内部状态、调度资格、查询结果和等待结果**。这既有助于准确理解内核行为，也能避免调试时因不同编码体系混淆而得出错误结论。

Wait Design

HRTOS 任务等待与唤醒模块

8.1 模块概述

任务等待与唤醒模块是 HRTOS 内核任务状态管理与调度控制的基础模块，负责统一处理任务阻塞、等待关系登记、唤醒结果记录、等待上下文清理及任务就绪状态恢复。

在实时操作系统中，任务并不总是持续执行。当任务需要等待指定时间，或者等待信号量、互斥锁、邮箱、消息队列及事件等条件满足时，内核需要暂时取消该任务的正常调度资格，并在相应条件满足或等待超时后恢复任务运行。

HRTOS 通过 `os_wait()` 和 `wake_task()` 将等待处理与唤醒处理分离：

- `os_wait()`：记录任务等待信息，将任务设置为等待状态，并请求内核进行任务调度。
- `wake_task()`：清理任务等待记录，设置唤醒结果，将任务恢复为就绪状态，并请求内核重新调度。
- `os_res_init()`：初始化资源控制块，为资源等待关系管理提供基础状态。

该模块不直接负责信号量计数、互斥锁所有权或邮箱数据收发等具体业务逻辑，而是为这些上层功能提供统一的任务等待与唤醒机制。

8.2 模块组成

源文件	核心职责
`wait.c`	统一处理任务等待请求、等待上下文记录、资源等待登记及任务阻塞
`wake_task.c`	清理任务等待状态、记录唤醒结果、恢复任务就绪状态
`res_init.c`	初始化资源控制块及其等待记录

8.3 核心数据与状态

8.3.1 任务等待上下文

`os_wait()` 使用任务控制块中的等待字段记录当前任务的等待信息。

字段	作用
wait_type	标记任务当前的等待类型
wait_obj	记录关联的等待对象编号
wait_tick	记录等待时间参数，以系统 Tick 为单位
wait_flag	记录任务的等待或唤醒结果
state	标记任务当前的运行状态

任务进入等待状态时，内核写入相应的等待信息，并将 state 设置为 WAIT。任务被唤醒时，内核清除等待上下文，将状态恢复为 READY。

其中，wait_flag 用于向任务提供等待结果。根据 os_wait() 中的注释，主要结果包括：

- WAIT_TIMEOUT：等待超时。
- WAIT_SIGNAL：因资源或其他唤醒操作而恢复运行。
- 0：等待参数无效，或当前任务已经处于等待状态等异常情况。

具体 API 的最终返回值还取决于各上层接口自身的处理逻辑。

8.3.2 资源控制块

os_res_init() 对 OS_RESOURCE 控制块执行以下初始化：

字段	初始化值	说明
value	0	资源初始值
owner	OS_INVALID_ID	初始时无有效拥有者
wait_cnt	0	等待任务计数清零
wait_mask	0	清除所有资源等待位

OS_RESOURCE 还包含 pending_signal 字段，但当前提供的 os_res_init() 并未对其赋值。因此，不能仅凭该函数认定资源控制块的所有字段都会被完整初始化；需要结合资源创建和其他初始化代码确认该字段的初始状态。

8.3.3 资源等待位图

HRTOS 使用 wait_mask 记录等待资源的标准任务。每一位对应一个任务编号，置位表示该任务已登记为该资源的等待者。

```
OS_RES[obj].wait_mask |= ((u16)1 << tid);
```

任务进入资源等待路径时，os_wait() 设置对应位，并递增 wait_cnt。

任务被唤醒时，wake_task() 根据任务记录的 wait_obj 清除对应等待位，并在 wait_cnt 非零时递减计数。

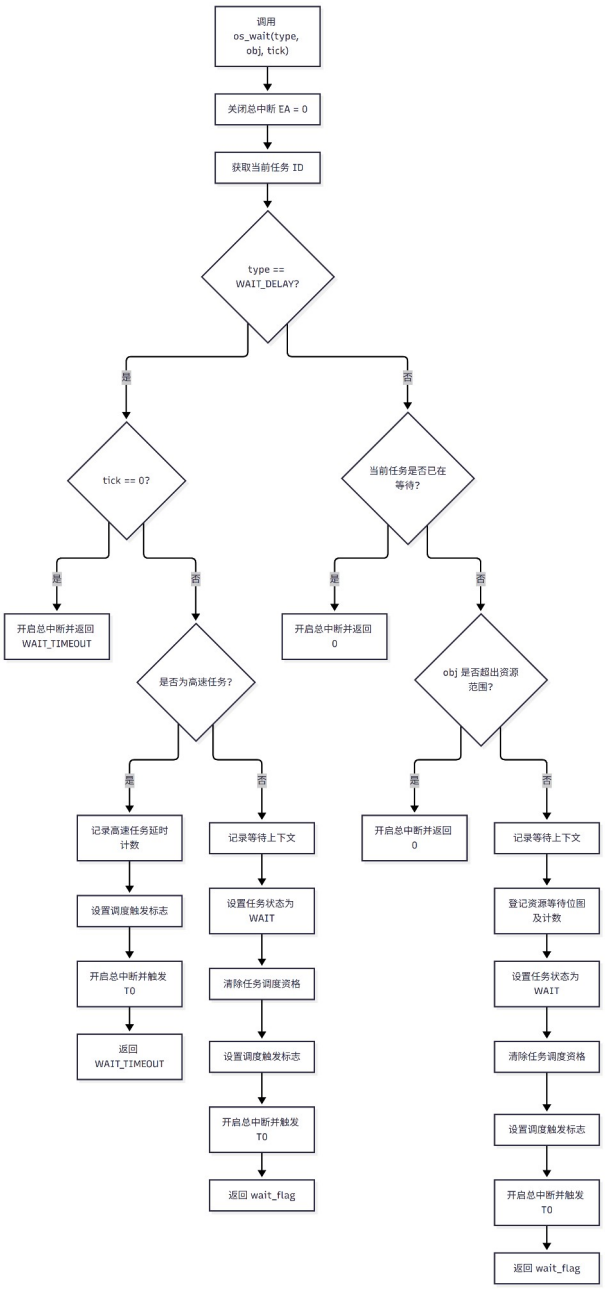
这种位图机制能够以紧凑的数据结构维护任务与资源之间的等待关系，避免为每个等待任务单独建立链表节点。

需要注意，当前 `wait_mask` 为 16 位，而 `os_wait()` 中资源等待登记使用当前任务编号 `tid` 进行移位。该资源等待路径主要面向标准任务；高速任务使用独立的延时等待处理路径。

8.4 任务等待机制

8.4.1 等待处理流程

`os_wait(type, obj, tick)` 是统一等待入口。函数首先进入临界区并读取当前任务编号，然后根据等待类型执行相应处理。



8.4.2 普通任务的阻塞过程

对于标准任务，等待处理主要包含以下步骤：

1. **检查当前等待状态。** 如果 `OS_TASK[tid].wait_type != WAIT_NONE`，函数认为当前任务已经处于等待状态，开启总中断后返回 `0`，不重复登记等待。
2. **检查等待对象。** 对于非 `WAIT_DELAY` 类型的等待，检查 `obj` 是否小于 `OS_RESOURCE_MAX`。超出资源范围时返回 `0`。
3. **记录等待上下文。** 写入 `wait_type`、`wait_obj`、`wait_tick`，并将 `wait_flag` 清零。
4. **登记资源等待关系。** 对于非延时等待且 `obj != OS_INVALID_ID` 的情况，将当前任务对应的位写入资源 `wait_mask`，并递增 `wait_cnt`。
5. **设置任务阻塞状态。** 将任务状态设为 `WAIT`，同时清除 `OS_PROCESS_OK[tid]` 的 bit0，取消该任务的正常调度资格。

6. **请求调度。** 设置 `OS_SCHED_REASON = 1`，开启总中断并设置 `TF0 = 1`，请求通过 Timer0 中断路径进行调度处理。

任务进入等待状态后，`os_wait()` 最后返回 `OS_TASK[tid].wait_flag`。该函数本身并不包含等待循环，而是通过任务状态变化和调度机制使当前任务暂时退出执行；任务恢复运行后，再从原调用位置继续执行并取得返回值。

8.4.3 延时等待的特殊处理

当 `type == WAIT_DELAY` 时，`os_wait()` 走独立的延时处理分支。

对于 `tick == 0` 的请求，函数直接开启总中断并返回 `WAIT_TIMEOUT`，不执行阻塞操作。

对于标准任务，函数记录等待上下文，将任务状态设为 `WAIT`，清除其正常调度资格，并请求调度。

对于高速任务，函数不使用标准任务的统一等待上下文，而是根据任务编号分别处理：

- `tid == OS_PROCESS_MAX`：清零 `OS_KUAI_PROCESS_A`，将 `tick` 写入 `OS_WAIT_DI2[0]`。
- `tid == OS_PROCESS_MAX + 1`：清零 `OS_KUAI_PROCESS_B`，将 `tick` 写入 `OS_WAIT_DI2[1]`。

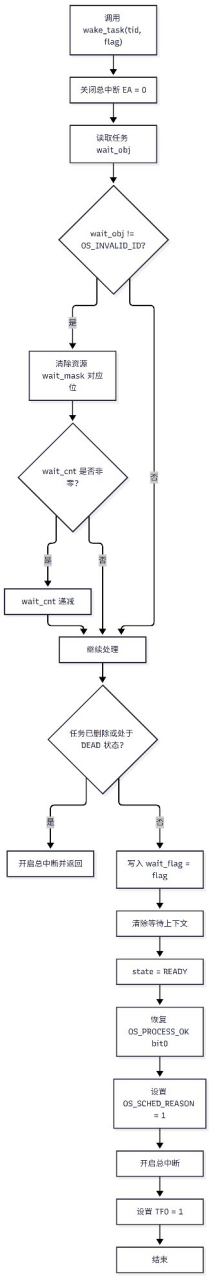
随后设置调度触发标志、开启总中断并触发 Timer0 中断。该路径不会像标准任务的延时路径一样填写 `OS_TASK[tid]` 的等待字段，也不会将高速任务加入标准资源等待位图。

需要注意：高速任务分支中，`EA = 1` 和 `TF0 = 1` 之后立即返回 `WAIT_TIMEOUT`，因此这个返回值不能简单解释为实际延时时间已经结束。高速任务的延时行为需要结合 `OS_WAIT_DI2` 的更新逻辑及调度实现理解。

8.5 任务唤醒机制

8.5.1 唤醒处理流程

`wake_task(tid, flag)` 用于唤醒指定任务。调用时，函数根据任务记录的等待对象清理资源等待关系，检查任务是否已删除或处于删除准备状态，然后写入唤醒结果并恢复任务就绪状态。



8.5.2 唤醒处理的关键步骤

清理资源等待记录

函数首先读取任务的 `wait_obj`。如果该字段不是 `OS_INVALID_ID`，则清除该任务在对应资源 `wait_mask` 中的位，并在 `wait_cnt` 非零时递减计数。

这一步用于解除任务与原等待资源之间的登记关系。资源等待记录的清理发生在任务删除状态检查之前。

检查任务状态

如果 `OS_PROCESS_OK[tid] == 0`，函数将其视为已删除任务；如果任务状态为 `DEAD`，则视为处于删除准备状态。两种情况下均开启总中断并直接返回，不再执行正常唤醒流程。

记录唤醒结果并清理上下文

对于可以正常唤醒的任务，函数将 `flag` 写入 `wait_flag`，然后将以下字段恢复为无等待状态：

- `wait_type = WAIT_NONE`
- `wait_obj = OS_INVALID_ID`
- `wait_tick = 0`

`wait_flag` 不会在该步骤中清零，因此可以保留此次唤醒的结果，供任务恢复执行后读取。

恢复任务就绪状态

函数将任务状态设置为 `READY`，并执行：

```
OS_PROCESS_OK[tid] |= 1;
```

该操作恢复 `OS_PROCESS_OK` 的 bit0，即重新赋予任务正常调度资格。随后设置 `OS_SCHED_REASON = 1`，开启总中断并通过 `TF0 = 1` 请求调度处理。

需要注意，恢复任务就绪状态不等于立即执行该任务。实际运行哪个任务仍由 HRTOS 的调度机制决定。

8.6 资源初始化机制

`os_res_init(OS_RESOURCE *r)` 负责初始化资源控制块的主要状态：

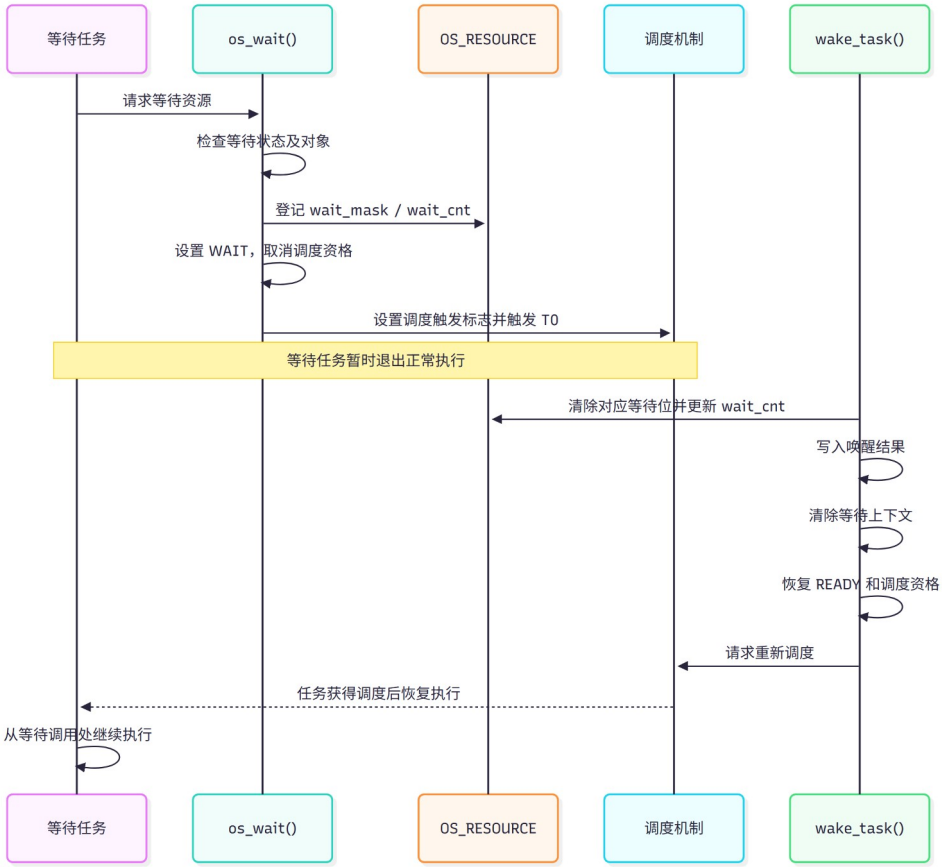
```
r->value = 0;
r->owner = OS_INVALID_ID;
r->wait_cnt = 0;
r->wait_mask = 0;
```

该函数不接收资源编号，而是接收资源控制块指针，由调用方确定要初始化的资源。

初始化后，资源值归零、拥有者失效，等待计数和等待位图均清空。该函数本身没有关闭或恢复总中断，也没有初始化 `pending_signal` 字段。因此，调用时机及完整初始化要求应由资源创建流程保证。

8.7 等待与唤醒的协作关系

任务等待和唤醒不是两个独立的操作，而是通过任务控制块和资源等待位图形成完整的状态管理过程。



该协作关系体现了模块的两个核心目标：

- **状态一致性：** 通过等待上下文和资源等待位图记录任务当前的等待关系，并在唤醒时清理对应记录。
- **调度协作：** 通过任务状态、`OS_PROCESS_OK`、`OS_SCHED_REASON` 和 Timer0 触发机制，将阻塞与唤醒操作交给内核调度流程处理。

8.8 设计特点

8.8.1 统一等待入口

不同内核功能可以通过 `os_wait()` 进入统一等待机制，减少各模块重复实现任务阻塞状态管理的需要。等待类型用于区分延时等待和资源等待，等待对象用于关联相应资源。

8.8.2 位图式等待关系管理

资源控制块使用 `wait_mask` 记录等待任务，使用 `wait_cnt` 维护等待数量。标准任务进入资源等待路径时登记对应位，唤醒时清除对应位。

8.8.3 等待结果显式传递

内核使用 `wait_flag` 保存唤醒结果，使任务在恢复运行后能够获取等待结束的原因。上层 API 可以根据自身语义进一步处理这一结果。

8.8.4 与调度机制解耦

`os_wait()` 和 `wake_task()` 负责更新任务状态并请求调度，而不是在函数内部直接选择并运行下一个任务。任务切换由内核调度机制完成。

8.8.5 高速任务独立处理延时

高速任务的延时请求通过专用状态变量和计数数组处理，不直接进入标准任务的资源等待登记流程。这种分支体现了高速任务与标准任务在等待管理上的实现差异。

8.9 使用注意事项

- **等待对象必须有效。** 非延时等待的 `obj` 需要处于有效资源编号范围内；`os_wait()` 对超范围对象返回 `0`。
- **重复等待会被拒绝。** 当前任务的 `wait_type` 不为 `WAIT_NONE` 时，`os_wait()` 不会重复登记等待。
- **唤醒结果需要正确解释。** `WAIT_SIGNAL` 和 `WAIT_TIMEOUT` 表示不同的唤醒原因，但上层 API 的最终返回值可能经过额外处理。
- **唤醒不等于资源获取成功。** 任务被唤醒只说明等待状态结束；是否成功取得资源，应由具体资源 API 的实现决定。
- **资源等待位图与计数必须保持一致。** `wake_task()` 按任务记录的 `wait_obj` 清除等待位，并在计数非零时递减；调用方需要保证任务等待上下文与资源记录相匹配。
- **删除状态会影响唤醒处理。** 已删除或处于 `DEAD` 状态的任务不会执行正常的唤醒状态恢复流程。
- **资源初始化需要完整性保证。** `os_res_init()` 未初始化 `pending_signal`，完整初始化应由资源创建流程结合实际使用情况保证。
- **总中断状态需要结合接口约定理解。** 当前实现使用 `EA = 0` 进入临界区、使用 `EA = 1` 重新开启总中断，并非保存和恢复调用前的 EA 状态。调用环境应遵循该内核的中断管理约定。

8.10 小结

HRTOS 任务等待与唤醒模块通过 `os_wait()`、`wake_task()` 和 `os_res_init()`，建立了从等待登记、任务阻塞、唤醒结果记录到任务重新就绪的基础流程。它将任务状态管理与具体资源操作分离，为信号量、互斥锁、邮箱、消息队列及其他等待机制提供公共内核支持，并通过调度触发机制与 HRTOS 的任务调度流程协同工作。

Event Design

HRTOS 事件模块

9.1 模块介绍

事件模块提供基于事件标志的任务同步机制，用于通知任务某个条件已经发生。

事件不承担数据传输功能。任务可以通过事件等待接口等待指定事件，也可以通过事件设置接口通知等待任务。事件对象使用 HRTOS 统一资源结构，并接入统一等待与唤醒机制。

事件模块支持任务上下文和中断上下文中的事件设置。

9.2 主要职责

- 事件对象初始化与删除
- 事件标志设置
- 事件状态查询
- 带超时的事件等待
- 等待任务管理与唤醒
- 中断上下文中的事件设置
- 与内核统一等待机制协同工作

9.3 主要文件

9.3.1 源文件

文件	主要职责
`Src/event/event_init.c`	初始化事件对象
`Src/event/event_write.c`	设置事件标志
`Src/event/event_wait.c`	等待事件发生
`Src/event/event_query.c`	查询事件状态
`Src/event/event_delete.c`	删除事件对象
`Src/interrupt/event_set_from_isr.c`	在中断上下文中设置事件

9.3.2 头文件

文件	主要职责
`Inc/event.h`	事件 API 声明
`Inc/config.h`	内核配置与相关类型定义
`Inc/hrtos_internal.h`	内核内部事件变量及接口声明

9.4 数据结构

9.4.1 OS_RESOURCE

事件对象使用统一资源结构 `OS_RESOURCE`：

```
typedef struct {
    u8 value;
    u8 owner;
    u8 wait_cnt;
    u16 wait_mask;
    u8 pending_signal;
} OS_RESOURCE;
```

各字段在事件场景中的用途如下：

字段	作用
`value`	事件标志值，0 表示未设置，非零表示已设置
`owner`	统一资源结构中的所有者字段，事件不使用资源所有权机制
`wait_cnt`	等待任务计数
`wait_mask`	等待任务位图
`pending_signal`	中断上下文使用的挂起信号计数

`wait_mask` 使用 16 位位图表示标准任务的等待状态，位 0 对应任务 0，位 15 对应任务 15。

9.4.2 事件相关变量

事件模块涉及以下内部变量：

```
extern unsigned int xdata OS_EVENT_BIT;
extern unsigned char xdata OS_TASK_EVENT[OS_PROCESS_MAX + OS_FAST_TASK_MAX];
```

- `OS_EVENT_BIT`：事件相关的位操作变量。
- `OS_TASK_EVENT`：按标准任务和快速任务数量配置的事件相关任务数据。

这些变量属于内核内部数据，不应与 `OS_RESOURCE.value` 表示的单个事件对象标志混为一谈。

9.5 核心函数

9.5.1 os_event_init()

位置： `Src/event/event_init.c`

功能： 初始化事件对象。

参数：

- `obj`：事件对象对应的资源 ID。

返回值：

- `1`：成功。
- `-1`：失败。

初始化过程包括检查资源 ID、初始化统一资源结构、清零事件标志，以及初始化相关等待信息。

事件对象使用统一资源池。当前 `OS_RESOURCE_MAX` 配置为 6，事件对象与其他使用 `OS_RESOURCE` 的 IPC 对象共享该资源池，因此事件可用数量取决于资源池配置和对象分配情况。

9.5.2 os_event_write()

位置： `Src/event/event_write.c`

功能： 在任务上下文中设置事件标志。

参数：

- `id`：事件对象对应的资源 ID。

返回值：

- `1`：成功。
- `-1`：失败。

事件设置操作将事件对象的 `value` 设为非零状态，并处理等待该事件的任务。事件设置采用持续有效的标志语义，事件被设置后不会因任务读取或等待返回而自动清除。

9.5.3 os_event_wait()

功能： 等待指定事件触发。

函数原型：

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

参数说明：

- `obj`：事件资源槽位编号，范围为 `0` 至 `OS_RESOURCE_MAX - 1`。
- `tick`：最大等待时间，以系统节拍数为单位，具体超时行为遵循内核等待机制。

返回值：

- `1`：事件已经触发，直接返回。
- `-1`：事件资源槽位编号无效。
- 其他结果：由 `os_wait()` 返回，具体含义遵循内核等待机制。

执行逻辑：

- 参数检查通过后，关闭总中断，检查事件状态。
- 如果事件已触发，恢复总中断并立即返回 `1`。
- 如果事件未触发，恢复总中断，调用 `os_wait(WAIT_EVENT, obj, tick)` 进入事件等待流程。

注意事项：

事件已触发时，函数不会进入等待流程；事件未触发时，任务的等待及超时处理由内核等待机制负责。

9.5.4 os_event_query()

位置： `Src/event/event_query.c`

功能： 查询事件状态，不主动阻塞当前任务。

参数：

- `id`：事件对象对应的资源 ID。

返回值：

- `0`：事件未设置。
- 非零：事件已设置。

该接口用于检查事件标志当前的状态。

9.5.5 os_event_delete()

位置： `Src/event/event_delete.c`

功能： 删除事件对象。

参数：

- `id`：事件对象对应的资源 ID。

返回值：

- `1`：成功。
- `-1`：失败。

删除操作负责清理事件对象及其相关等待信息。对于正在等待该事件的任务，内核需要按照事件删除机制处理其等待状态。

9.5.6 os_event_set_from_isr()

位置： `Src/interrupt/event_set_from_isr.c`

功能： 在中断上下文中设置事件。

参数：

- `id`：事件对象对应的资源 ID。

返回值：

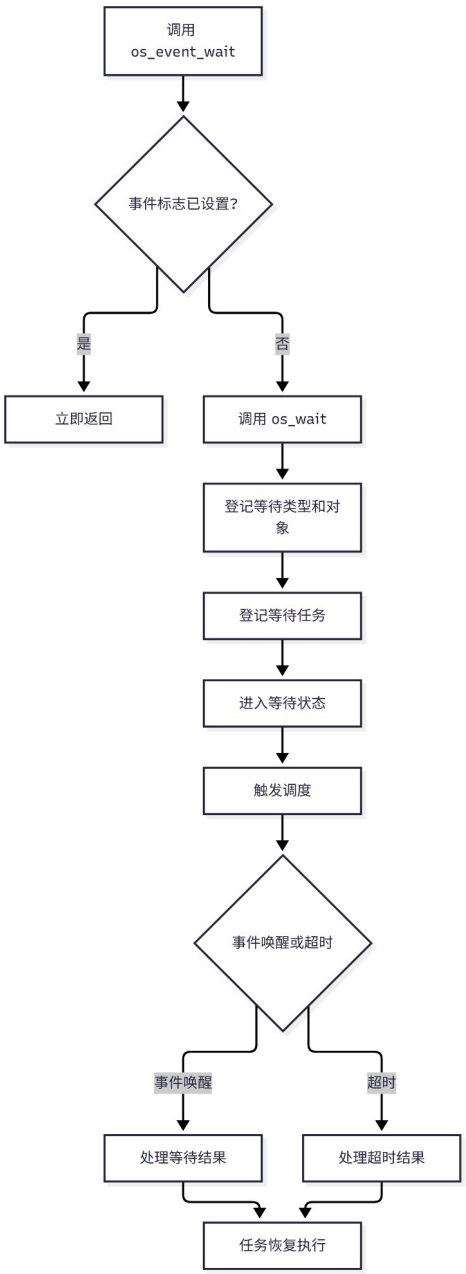
- `1`：成功。
- `-1`：失败。

该接口用于中断服务程序通知事件发生。其处理路径与普通任务上下文中的事件设置不同，涉及事件标志、挂起信号计数及调度请求等内核机制。

中断服务程序应使用专门的 ISR 接口执行相应操作，避免将普通任务上下文中的处理流程直接套用到中断上下文。

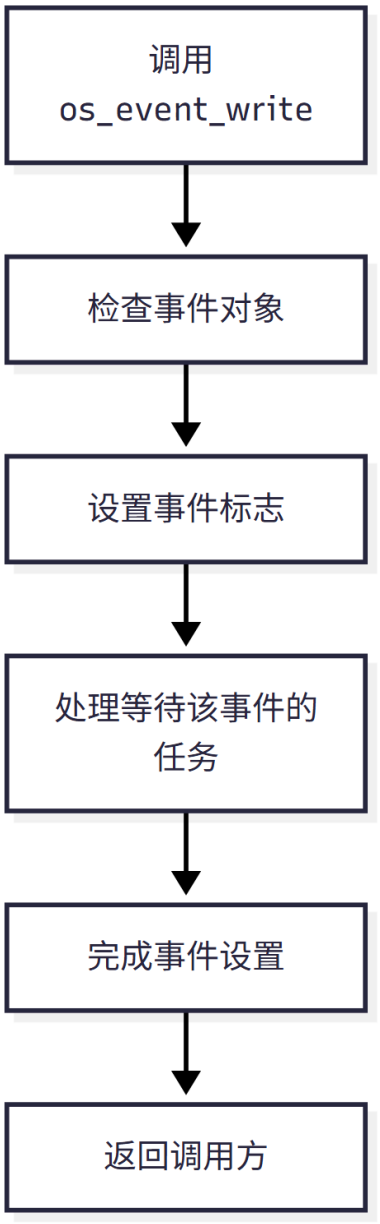
9.6 调用关系

9.6.1 事件等待流程



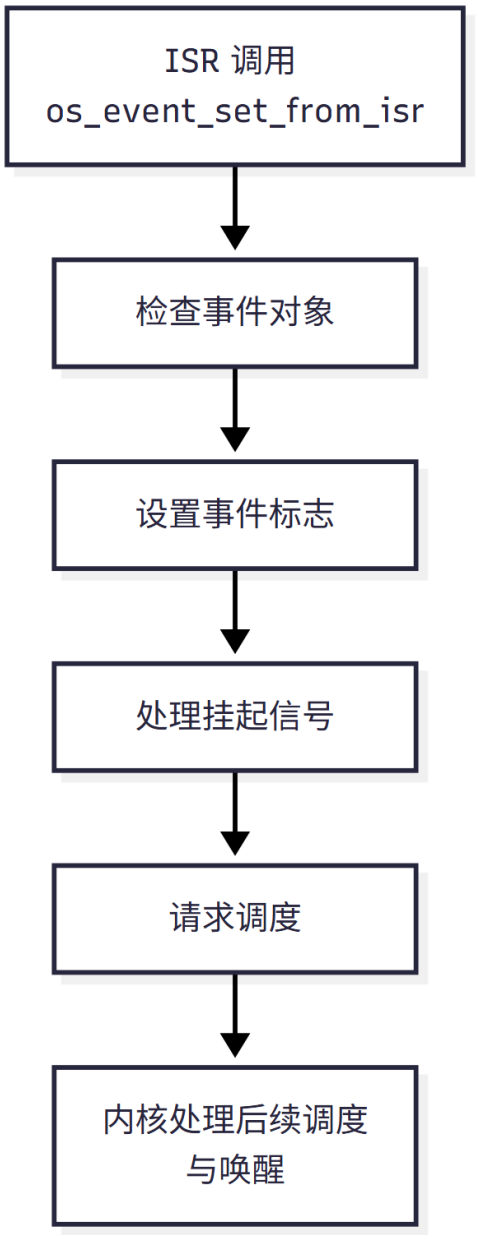
事件已经设置时，任务可通过快速路径直接返回；事件尚未设置时，任务进入统一等待机制，由内核负责等待状态管理、唤醒和超时处理。

9.6.2 任务上下文中的事件设置



事件设置后，等待该事件的任务按照事件唤醒机制进行处理。事件标志保持设置状态，不因唤醒任务而自动清除。

9.6.3 中断上下文中的事件设置



ISR 事件设置通过专用接口完成，并与内核调度机制协同工作。

9.7 事件生命周期

9.7.1 初始化

调用 `os_event_init()` 初始化事件对象，建立事件标志及相关资源状态。

9.7.2 等待

任务调用 `os_event_wait()` 检查事件状态。事件已设置时可立即返回；事件未设置时进入统一等待机制。

9.7.3 设置

任务调用 `os_event_write()`，或中断服务程序调用 `os_event_set_from_isr()`，将事件标志设置为有效状态。

9.7.4 唤醒

事件设置后，内核根据事件等待机制处理相关等待任务，使符合唤醒条件的任务重新参与调度。

9.7.5 保持

事件标志采用持续有效的状态语义。事件被设置后，标志保持有效，直到通过明确的事件管理操作改变其状态。

9.7.6 删除

调用 `os_event_delete()` 删除事件对象，并处理相应的资源和等待状态。

9.8 设计特点

9.8.1 轻量级同步

事件以标志状态表达事件是否发生，不携带数据负载，适合任务间的状态通知与条件同步。

9.8.2 统一等待机制

事件等待通过 `os_wait()` 接入内核统一等待体系，使用等待类型、等待对象和等待计时信息管理阻塞与恢复。

9.8.3 位图等待管理

统一资源结构通过 `wait_mask` 记录标准任务的等待状态，避免为每个事件对象单独建立复杂的等待队列结构。

9.8.4 广播式事件语义

事件用于通知某个条件已经发生，不承担信号量计数或消息队列数据传递功能。具体等待任务的唤醒范围由事件设置接口的实现决定。

9.8.5 中断上下文支持

事件模块提供独立的 ISR 设置接口，用于将中断中发生的事件通知给内核，并配合调度请求完成后续处理。

9.9 系统约束

- 事件对象使用统一资源池，当前资源池容量由 `OS_RESOURCE_MAX` 决定。
- 当前 `OS_RESOURCE_MAX` 配置为 6，事件与其他统一资源对象共享该容量。
- 事件不携带数据负载。
- 事件标志采用持续有效语义，不因等待返回而自动清除。
- 事件等待通过统一等待机制处理阻塞和超时。
- 中断上下文中的事件设置应使用 `os_event_set_from_isr()`。
- 事件等待、唤醒和删除操作应遵循统一资源管理规则。

9.10 使用示例

9.10.1 事件通知

```
/* 任务 A：等待事件 */
result = os_event_wait(EVENT_ID, OS_WAIT_FOREVER);

/* 任务 B：设置事件 */
os_event_write(EVENT_ID);
```

任务 A 等待指定事件；任务 B 设置事件标志后，内核按照事件等待机制处理相关任务。

9.10.2 带超时的事件等待

```
result = os_event_wait(EVENT_ID, 100);

if (result == WAIT_TIMEOUT)
{
    /* 处理超时 */
}
```

该示例展示带超时等待的使用形式。等待结果常量及其与 API 返回值的对应关系，应与 HRTOS 的等待接口定义一致。

9.10.3 中断事件通知

```
void external_interrupt_handler(void)
{
```

```
    os_event_set_from_isr(EVENT_ID);
}
```

中断服务程序通过专用接口设置事件，使中断中发生的事件能够进入内核的通知与调度处理流程。

9.11 性能特点

9.11.1 事件已设置时的快速路径

当事件标志已经有效时，等待接口可以直接返回，不必进入任务阻塞与恢复流程。

9.11.2 等待与唤醒开销

事件未设置时，任务需要经过统一等待机制。事件设置时，内核还需要处理相关等待任务，因此唤醒开销会随等待任务数量及实际处理路径变化。

9.11.3 内存开销

事件使用统一 `OS_RESOURCE` 对象和现有等待位图，不需要为每个事件单独维护独立的任务链表。

9.12 与其他 IPC 的区别

IPC 类型	主要用途	核心语义
事件	状态通知与条件同步	事件标志
信号量	资源计数与任务同步	计数及等待处理
消息队列	任务间数据传递	消息缓存与队列
互斥锁	共享资源互斥访问	所有权与互斥控制

事件适合表达“某件事已经发生”，不适合承载数据，也不能替代互斥锁的资源所有权管理。

Semaphore Design

HRTOS 信号量设计

10.1 模块介绍

HRTOS 信号量模块提供基于计数的任务同步机制，用于协调多个任务之间的执行关系，以及管理具有多个可用实例的共享资源。

信号量采用经典的 P/V 操作模型：

- **P 操作：** 通过 `os_sem_wait()` 获取信号量。计数大于 0 时，消耗一个计数；计数为 0 时，进入等待流程。
- **V 操作：** 通过 `os_sem_post()` 释放信号量。有任务等待时，优先唤醒一个等待任务；没有等待任务时，增加信号量计数。

信号量不记录资源所有者，也不提供互斥锁的所有权检查和优先级继承机制。

10.2 主要职责

信号量模块主要负责：

- 初始化信号量及其初始计数。
- 提供信号量获取与释放操作。
- 管理可用资源计数。
- 维护等待任务位图。
- 在释放信号量时，根据等待任务的基础优先级选择唤醒对象。

10.3 主要文件

10.3.1 源文件

- `Src/semaphore/semaphore_init.c`：信号量初始化。
- `Src/semaphore/semaphore_wait.c`：信号量等待与获取。
- `Src/semaphore/semaphore_post.c`：信号量释放与等待任务唤醒。

10.3.2 头文件

- `hrtos_internal.h`：内部类型、资源结构及相关内核接口声明。

10.4 数据结构

10.4.1 OS_RESOURCE 统一资源结构

信号量复用 HRTOS 的统一资源结构：

```
typedef struct {
    u8  value;
    u8  owner;
    u8  wait_cnt;
    u16 wait_mask;
    u8  pending_signal;
} OS_RESOURCE;
```

各字段在信号量模块中的用途如下：

字段	用途
`value`	信号量计数，表示当前可直接获取的计数值。
`owner`	信号量不使用所有权机制，初始化时设置为 `OS_INVALID_ID`。
`wait_cnt`	初始化时清零，当前提供的信号量实现未使用该字段维护等待数量。
`wait_mask`	等待任务位图，用于记录等待该信号量的标准任务。
`pending_signal`	属于统一资源结构；当前提供的三个信号量函数没有直接使用该字段。

`wait_mask` 为 16 位位图，标准任务 ID 0–15 对应 bit 0–15。

当前 `OS_RESOURCE_MAX` 为 6。信号量与其他使用 `OS_RESOURCE` 的 IPC 对象共享资源池，因此可创建的信号量数量取决于资源槽位的整体分配情况，并非独立拥有 6 个专用槽位。

10.5 核心函数

10.5.1 os_sem_init()

函数原型

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

功能

初始化指定资源槽位中的信号量，设置初始计数并清空相关等待状态。

参数

- `sid`：信号量使用的资源槽位编号，有效范围为 `0` 至 `OS_RESOURCE_MAX - 1`。
- `init_val`：信号量初始计数，类型为 `u8`，范围为 0–255。

返回值

- `1`：初始化成功。
- `-1`：资源槽位编号无效。

实现过程

1. 检查 `sid` 是否超出资源池范围。
2. 获取对应的 `OS_RESOURCE` 结构。
3. 将 `value` 设置为 `init_val`。
4. 将 `owner` 设置为 `OS_INVALID_ID`。
5. 清零 `wait_mask` 和 `wait_cnt`。

该函数直接初始化结构字段，不调用 `os_res_init()`。调用方应在系统初始化阶段，或能够确保没有相关任务正在使用该信号量时，执行初始化操作。

10.5.2 os_sem_wait()

函数原型

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

功能

执行 P 操作，获取一个信号量计数。当计数为 0 时，调用统一等待接口进入等待流程。

参数

- `sid`：信号量资源槽位编号。

返回值

- `WAIT_SIGNAL`：计数大于 0 时，函数递减计数并返回该值。
- `0`：资源槽位编号无效。
- 其他返回值：取决于实际等待路径及 `os_wait()` 的返回结果。

实现过程

1. 检查资源槽位编号是否合法。
2. 关闭总中断，获取对应的资源结构。
3. 如果 `value == 0`，先更新当前任务的 `base_prio`，随后重新开启总中断并调用 `os_wait(WAIT_SEM, sid, 0)`。
4. 检查 `value` 是否大于 0；若大于 0，则递减计数、开启总中断并返回 `WAIT_SIGNAL`。
5. 如果未进入上述快速返回路径，则开启总中断并返回等待接口保存的结果。

设计特点

- 计数可用时，直接递减并返回，无须调用 `os_wait()`。
- 计数为 0 时，进入统一任务等待流程。
- 等待调用的超时参数为 `0`，具体等待与唤醒语义还取决于内核等待机制的实现。
- 本函数的返回值应结合实际执行路径判断，不宜仅凭“函数已经返回”就认定获取成功。

使用注意

由于该函数在 `value == 0` 的分支中调用 `os_wait()` 后仍会继续检查 `value`，其具体返回行为与内核等待机制存在关联。调用方应依据实际的返回值约定处理结果，不应将所有非零返回值一概视为获取成功。

10.5.3 os_sem_post()

函数原型

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

功能

执行 V 操作，释放一个信号量。若有等待任务，则选择一个任务唤醒；否则增加信号量计数。

参数

- `sid`：信号量资源槽位编号。

返回值

- `1`：检测到等待任务并执行唤醒处理路径。
- `0`：没有等待任务，释放操作正常结束。
- `-1`：资源槽位编号无效。

实现过程

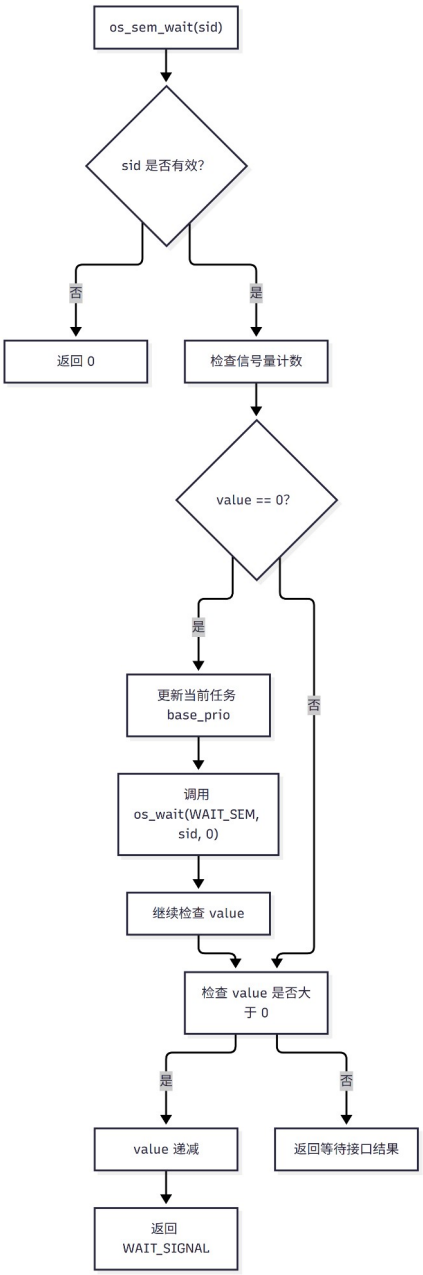
1. 检查资源槽位编号。
2. 关闭总中断，进入临界处理区。
3. 检查 `wait_mask`。
4. 如果存在等待任务，遍历标准任务位图，比较等待任务的 `base_prio`，选择基础优先级最高的任务。
5. 如果找到有效任务，调用 `wake_task(next, WAIT_SIGNAL)`。
6. 有等待任务时，不增加 `value`，随后开启总中断并返回 `1`。
7. 没有等待任务时，若 `value < 255`，则将计数加 1；达到 255 时保持不变。随后开启总中断并返回 `0`。

设计特点

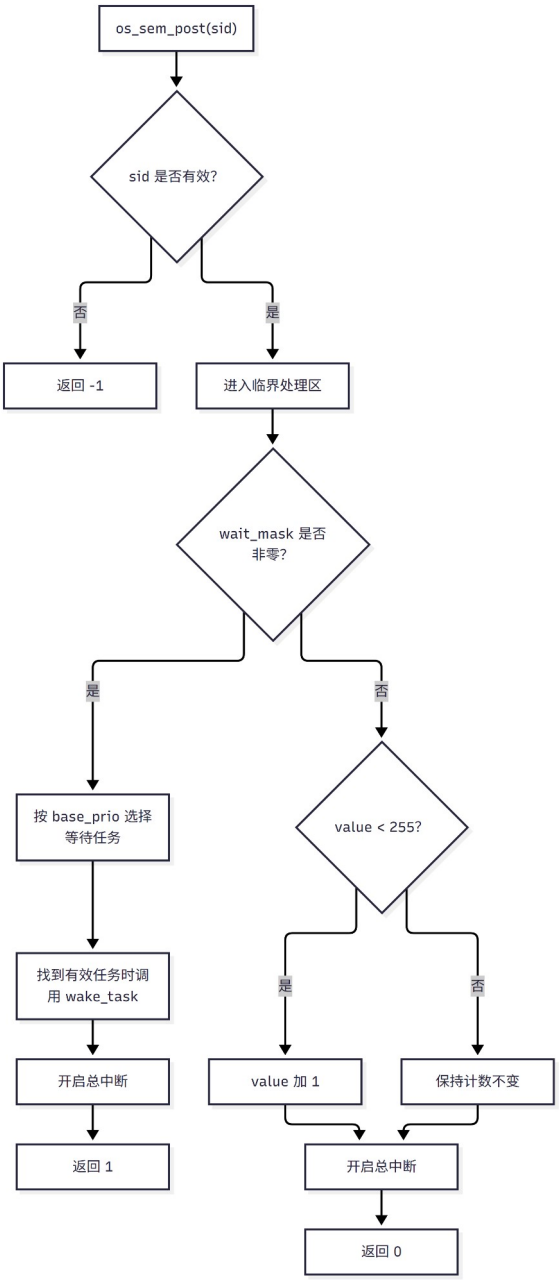
- 采用先处理等待任务、再累加计数的释放策略。
- 多个任务等待时，按 `base_prio` 选择优先级最高的任务。
- 当没有等待任务时，计数最大为 255，不会发生 8 位计数器溢出。
- 当 `wait_mask` 非零但遍历后没有找到有效任务时，当前实现仍走返回 `1` 的路径；该返回值表示执行了等待任务处理分支，并不保证一定有有效任务被唤醒。

10.6 调用流程

10.6.1 信号量获取流程



10.6.2 信号量释放流程



10.7 生命周期

一个典型的信号量使用过程如下：

1. **初始化：** 使用 `os_sem_init()` 设置初始计数。
2. **获取：** 任务调用 `os_sem_wait()` 尝试获取计数。
3. **等待：** 当计数为 0 时，进入内核等待流程。
4. **释放：** 任务调用 `os_sem_post()` 释放信号量。
5. **唤醒或累加：** 存在等待任务时执行唤醒处理；没有等待任务时增加计数。
6. **持续使用：** 在资源池和任务生命周期允许的范围内重复使用。

10.8 设计原则

10.8.1 计数语义

- `value` 表示当前信号量计数。
- 获取操作在可用路径中递减计数。
- 释放操作在没有等待任务时递增计数。
- 计数范围为 0–255。

10.8.2 等待任务选择

释放信号量时，当前实现遍历 `wait_mask`，以等待任务的 `base_prio` 作为比较依据，选择基础优先级最高的任务。

这与互斥锁的优先级继承逻辑不同：信号量不会因为某个任务等待而修改持有者优先级，也不记录资源所有权。

10.8.3 统一资源模型

- 信号量使用统一 `OS_RESOURCE` 资源结构。
- 通过 `os_wait()` 接入内核统一等待机制。
- 等待任务通过 `wait_mask` 进行标记。
- 信号量与其他 IPC 对象共享资源槽位。

10.9 使用约束

- 信号量资源槽位数量由 `OS_RESOURCE_MAX` 和其他 IPC 对象的使用情况共同决定。
- 初始计数和运行计数均使用 `u8`，范围为 0–255。
- `os_sem_wait()` 的等待参数为 `0`，不提供独立的有限等待时间参数。
- 信号量不记录所有者，不支持互斥锁式的所有权检查。
- 信号量没有优先级继承机制。
- 不应将信号量直接等同于互斥锁；需要独占访问和所有权管理时，应使用互斥锁。

10.10 使用模式

10.10.1 资源计数

当系统存在多个可用资源实例时，可以使用计数信号量协调访问。

```
#define SEM_RESOURCE_ID  0

void semaphore_init(void)
{
    os_sem_init(SEM_RESOURCE_ID, 3);
}
```

```
void task_a(void)
{
    while (1)
    {
        if (os_sem_wait(SEM_RESOURCE_ID) == WAIT_SIGNAL)
        {
            use_resource();
            os_sem_post(SEM_RESOURCE_ID);
        }

        os_delay(10);
    }
}
```

示例中的成功判断仅适用于 `WAIT_SIGNAL` 确实表示获取成功的路径。对于阻塞路径，应结合实际 `os_wait()` 返回约定确认任务是否成功取得信号量，再执行资源操作。

10.10.2 二进制信号量

将初始计数设置为 0，可用于表达“当前没有可获取信号”的初始状态。

```
#define SEM_EVENT_ID  1

void semaphore_init(void)
{
    os_sem_init(SEM_EVENT_ID, 0);
}

void producer_task(void)
{
    while (1)
    {
        produce_data();
        os_sem_post(SEM_EVENT_ID);
        os_delay(10);
    }
}

void consumer_task(void)
{
    while (1)
    {
        if (os_sem_wait(SEM_EVENT_ID) == WAIT_SIGNAL)
        {
            consume_data();
        }
    }
}
```

这里展示的是基本使用模式。实际应用应确认阻塞路径的返回值语义，并根据生产者与消费者之间的同步需求设计错误处理。

10.11 性能考虑

10.11.1 可用计数路径

当 `value > 0` 时，获取操作直接递减计数并返回 `WAIT_SIGNAL`，不需要调用 `os_wait()`。这是信号量的直接获取路径。

10.11.2 等待与唤醒路径

当 `value == 0` 时，获取操作进入统一等待流程；释放操作在检测到等待任务后，会进行优先级比较并执行唤醒处理。该路径涉及任务等待和唤醒机制，开销高于直接获取路径。

10.11.3 内存占用

信号量复用 `OS_RESOURCE`，不额外定义独立的信号量控制块。

在当前 Keil C51 配置下，`OS_RESOURCE` 结构由三个 8 位字段、一个 16 位字段和另一个 8 位字段组成，合计 6 字节。实际结构大小以当前编译器布局和工程配置为准。

10.12 与其他 IPC 的比较

10.12.1 信号量与互斥锁

- **信号量：** 计数同步，不记录所有者。
- **互斥锁：** 用于独占访问，记录所有者，并具有相应的优先级继承处理。

10.12.2 信号量与事件

- **信号量：** 通过计数表示可获取的信号或资源数量。
- **事件：** 通常通过事件状态或标志表达某个条件是否发生。

10.12.3 信号量与消息队列

- **信号量：** 主要用于计数和任务同步。
- **消息队列：** 用于在任务之间传递带有内容的数据。

10.13 常见注意事项

10.13.1 正确处理获取结果

调用 `os_sem_wait()` 后，应根据实际返回值约定判断是否成功获取信号量，不能将任意非零值直接当作成功。

10.13.2 配对使用获取与释放

对计数信号量而言，获取与释放次数应符合应用的资源管理逻辑。多释放可能使计数提前达到上限，多获取则可能导致任务等待。

10.13.3 区分同步与互斥

信号量适合资源计数和任务间同步，但不具备互斥锁的所有权检查与优先级继承机制。需要保护独占共享资源时，应根据实际场景选择合适的同步原语。

Mutex Design

HRTOS 互斥锁设计

11.1 模块介绍

互斥锁模块用于保护任务之间共享的资源，确保同一时刻只有一个任务持有指定互斥锁。

HRTOS 互斥锁使用统一 IPC 资源结构管理锁的所有者和等待任务，并支持优先级继承：当较高优先级任务等待较低优先级任务持有的锁时，内核可以提升锁所有者的当前优先级，降低优先级反转的影响。

互斥锁不支持递归获取，获取操作不提供超时参数。

11.2 主要职责

互斥锁模块负责：

- 初始化互斥锁资源
- 获取和释放互斥锁
- 记录当前锁所有者
- 管理互斥锁等待任务
- 根据等待任务的优先级执行优先级继承
- 解锁时恢复当前所有者的基础优先级
- 在解锁时选择等待队列中当前优先级最高的任务，并转移锁所有权

11.3 主要文件

11.3.1 源文件

- `Src/mutex/mutex_init.c`：互斥锁初始化
- `Src/mutex/mutex_lock.c`：互斥锁获取及优先级继承
- `Src/mutex/mutex_unlock.c`：互斥锁释放、所有权转移及优先级恢复

11.3.2 头文件

- `Inc/mutex.h`：互斥锁 API 声明
- `Inc/config.h`：统一 IPC 资源结构及相关配置
- `Inc/hrtos_internal.h`：内核内部变量和接口声明

11.4 数据结构

11.4.1 OS_RESOURCE

互斥锁使用统一 IPC 资源结构 `OS_RESOURCE`，而不是独立的互斥锁结构。

```
typedef struct {
    u8 value;
    u8 owner;
    u8 wait_cnt;
    u16 wait_mask;
    u8 pending_signal;
} OS_RESOURCE;
```

互斥锁相关字段说明：

- `owner`：当前持有互斥锁的任务 ID；`OS_INVALID_ID` 表示当前没有所有者。
- `wait_mask`：等待该互斥锁的标准任务位图，bit0–bit15 对应标准任务 0–15。
- `wait_cnt`：统一资源结构中的等待计数。
- `value`：互斥锁初始化时设置为 `1`；当前加锁和解锁实现主要通过 `owner` 判断锁的所有权。
- `pending_signal`：统一资源结构中的通用字段，当前所示互斥锁加锁和解锁代码不直接使用该字段。

11.4.2 资源配置

当前配置：

- `OS_RESOURCE_MAX = 6`：统一 IPC 资源槽位数量上限。
- 互斥锁与其他 IPC 对象共享统一资源池。

因此，互斥锁可使用的资源槽位受统一资源池限制，不能将其单独描述为最多支持 8 个互斥锁。实际可用数量还取决于其他 IPC 对象占用的资源槽位。

11.5 核心函数

11.5.1 os_mutex_init()

位置： `Src/mutex/mutex_init.c`

功能： 初始化指定互斥锁资源。

函数原型：

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

参数：

- `mid`：互斥锁资源槽位编号，范围为 `0` 至 `OS_RESOURCE_MAX - 1`。

返回值：

- `1`：初始化成功。
- `-1`：资源槽位编号无效。

处理过程：

- 检查资源槽位编号。
- 获取对应的 `OS_RES[mid]`。
- 将 `value` 设置为 `1`。
- 将 `owner` 设置为 `OS_INVALID_ID`。
- 将 `wait_mask` 和 `wait_cnt` 清零。

该函数直接初始化统一资源结构的相关字段，不调用 `os_res_init()`。

11.5.2 os_mutex_lock()

位置： `Src/mutex/mutex_lock.c`

功能： 获取指定互斥锁；如果锁已被其他任务持有，则根据优先级继承逻辑处理，并进入内核等待流程。

函数原型：

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

参数：

- `mid`：互斥锁资源槽位编号。

返回值：

- `1`：直接获取互斥锁成功。
- `-1`：资源槽位编号无效，或当前任务已是该锁的所有者。
- 其他结果：锁被占用时，直接返回 `os_wait(WAIT_MUTEX, mid, 0)` 的结果。

处理过程：

- 检查资源槽位编号。
- 获取当前任务 ID，并定位对应的资源结构。
- 关闭总中断，进入临界处理流程。
- 如果当前任务已经是锁的所有者，则恢复总中断并返回 `-1`，不支持递归获取。
- 如果 `owner == OS_INVALID_ID`，将当前任务设为所有者，并根据 `OS_PROCESS_OK` 恢复其基础优先级，随后返回 `1`。
- 如果锁已被其他任务持有，则根据 `OS_PROCESS_OK` 更新当前任务的基础优先级和当前优先级。

- 比较等待任务与锁所有者的当前优先级；当等待任务优先级更高时，提升所有者的当前优先级，并同步更新 `OS_PROCESS_OK` 中的优先级位。
- 恢复总中断后，调用 `os_wait(WAIT_MUTEX, mid, 0)` 进入等待流程，并返回该调用的结果。

注意事项：

锁被占用时，`os_mutex_lock()` 的返回值由 `os_wait()` 决定，不能统一描述为固定返回 `0` 或 `1`。调用者应结合实际内核等待机制理解返回结果。

11.5.3 os_mutex_unlock()

位置： `Src/mutex/mutex_unlock.c`

功能： 释放指定互斥锁，恢复当前所有者的基础优先级；如果存在有效等待任务，则将锁的所有权转移给其中当前优先级最高的任务，并唤醒该任务。

函数原型：

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

参数：

- `mid`：互斥锁资源槽位编号。

返回值：

- `1`：锁的所有权已转移给选定的等待任务。
- `0`：没有有效等待任务，或等待位图中没有找到有效任务。
- `-1`：资源槽位编号无效，或当前任务不是锁的所有者。

处理过程：

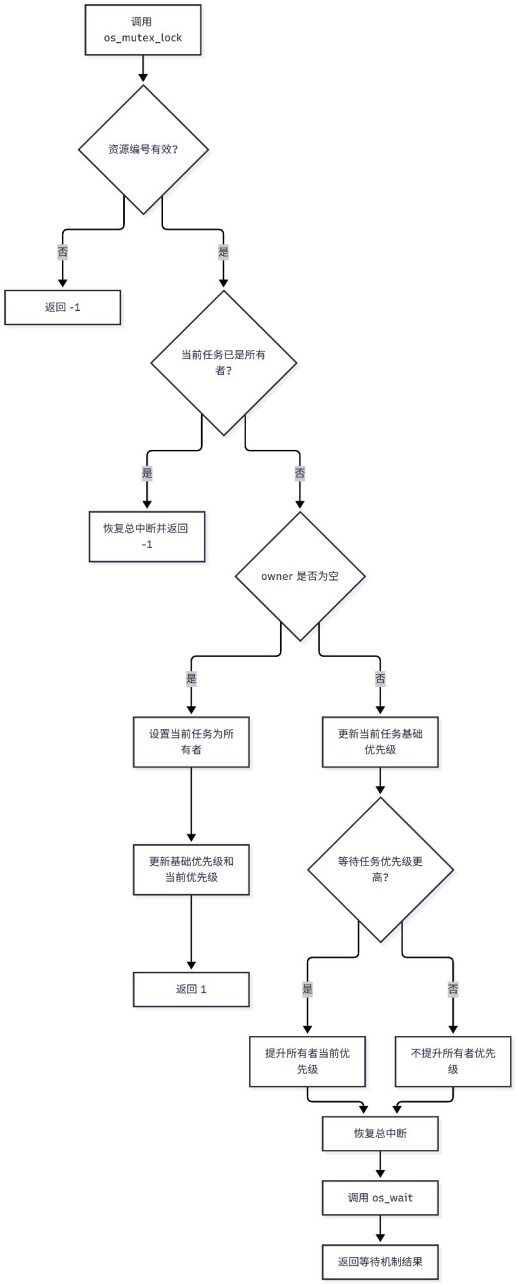
- 检查资源槽位编号。
- 获取当前任务 ID 和对应的资源结构。
- 关闭总中断，进入临界处理流程。
- 如果当前任务不是锁的所有者，恢复总中断并返回 `-1`。
- 将当前所有者的 `cur_prio` 恢复为 `base_prio`，并同步更新 `OS_PROCESS_OK` 中的优先级位。
- 检查 `wait_mask`。
- 如果存在等待位，则遍历标准任务 0–15，选择位图中 `cur_prio` 最高的任务。
- 如果未找到有效任务，则清空等待位图、将所有者设为 `OS_INVALID_ID`，恢复总中断并返回 `0`。
- 如果找到有效任务，先将 `owner` 设置为该任务 ID，再调用 `wake_task(next, WAIT_SIGNAL)` 唤醒该任务，最后恢复总中断并返回 `1`。
- 如果没有等待任务，则将 `owner` 设置为 `OS_INVALID_ID`，恢复总中断并返回 `0`。

重要说明：

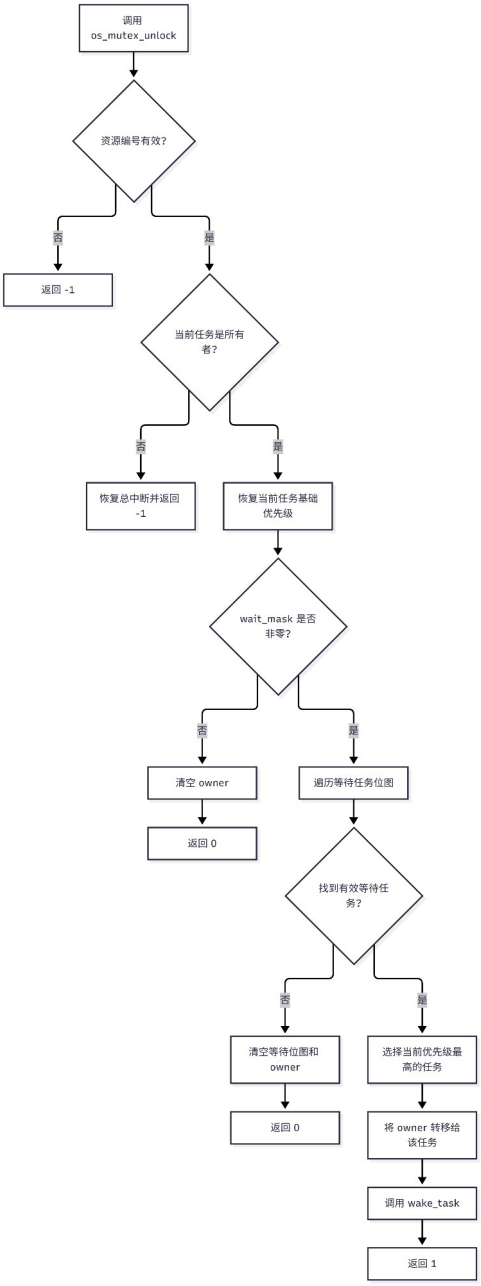
所有权转移发生在唤醒调用之前。因此，解锁路径并不是简单地清空所有者字段，再让等待任务竞争锁；它会先将锁指定给选中的任务。

11.6 调用关系

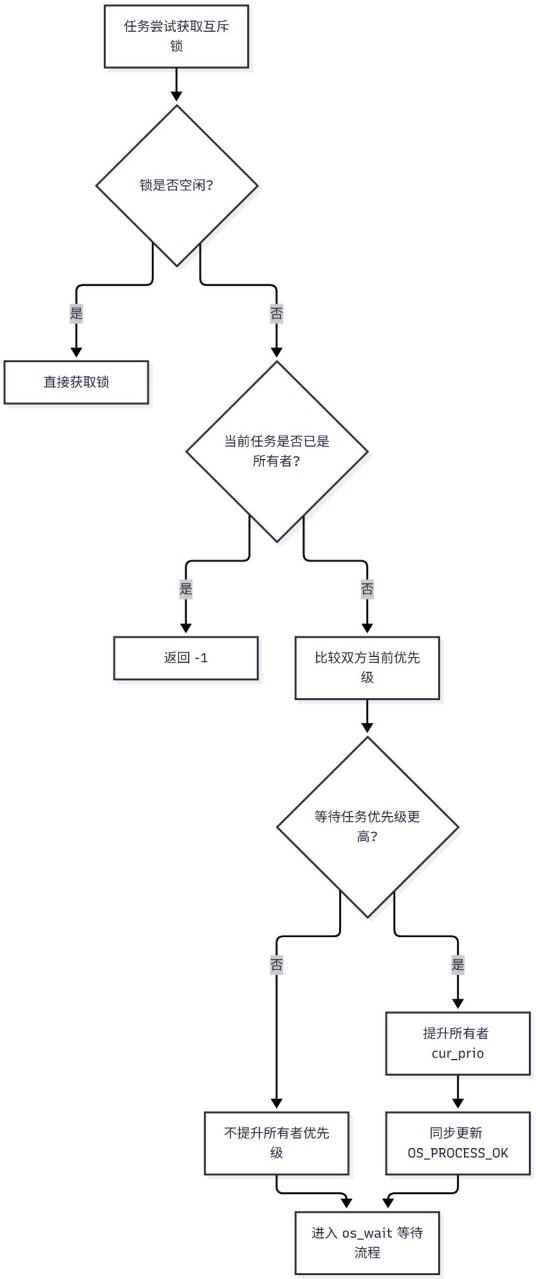
11.6.1 互斥锁获取流程



11.6.2 互斥锁释放流程



11.6.3 优先级继承流程



以上流程描述当前实现中的主要处理路径。实际任务切换时机仍由内核调度机制决定。

11.7 生命周期

11.7.1 互斥锁生命周期

- **初始化：** 调用 `os_mutex_init()`，清空所有者和等待状态。
- **获取：** 任务调用 `os_mutex_lock()`。
- **直接获取：** 锁空闲时，当前任务成为所有者。
- **竞争等待：** 锁被其他任务持有时，当前任务根据优先级继承逻辑处理后进入等待流程。

- **释放：** 所有者调用 `os_mutex_unlock()`。
- **优先级恢复：** 当前所有者的当前优先级恢复到基础优先级。
- **所有权转移：** 如果存在有效等待任务，选择其中当前优先级最高的任务并转移锁所有权。
- **恢复空闲：** 没有有效等待任务时，将所有者设置为 `OS_INVALID_ID`。

11.8 设计原则

11.8.1 互斥与所有权

- 使用 `owner` 记录当前锁所有者。
- 当前所有者才能成功释放互斥锁。
- 不允许同一任务递归获取同一把锁。
- 所有权转移与等待任务唤醒由解锁流程完成。

11.8.2 优先级继承

- 当锁被其他任务持有时，比较等待任务与锁所有者的当前优先级。
- 当等待任务优先级更高时，提升所有者的当前优先级。
- 将优先级更新同步到 `OS_PROCESS_OK` 对应的优先级位。
- 解锁时，将当前所有者的当前优先级恢复到基础优先级。
  当前实现不应被描述为完整的多锁优先级继承链或优先级天花板协议。

11.8.3 统一资源模型

- 互斥锁使用 `OS_RESOURCE` 统一资源结构。
- 通过 `OS_RES[mid]` 访问对应资源。
- 通过 `os_wait()` 进入互斥锁等待流程。
- 通过 `wake_task()` 唤醒选定的等待任务。

11.9 约束

- 互斥锁共享统一 IPC 资源池，当前资源槽位上限为 6。
- 不支持递归获取。
- 只有当前所有者可以释放互斥锁。
- 获取接口不提供超时参数。
- 不应在 ISR 上下文中使用互斥锁 API。
- 当前实现包含优先级继承和解锁时的基础优先级恢复。
- 不应据此宣称支持完整的优先级继承链或优先级天花板协议。

11.10 优先级继承算法

11.10.1 优先级提升条件

在锁已被其他任务持有时：

- 当前任务不是锁的所有者。
- 当前任务的优先级高于锁所有者的当前优先级。

满足条件时，执行优先级提升。

11.10.2 优先级提升实现

```
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);
}
```

该代码将锁所有者的当前优先级提升到等待任务的当前优先级，并更新 `OS_PROCESS_OK` 中对应的优先级位。

11.10.3 优先级恢复

解锁时：

- 将当前所有者的 `cur_prio` 恢复为 `base_prio`。
- 清除 `OS_PROCESS_OK` 中原有的优先级位。
- 将基础优先级写回对应的优先级位。

该恢复操作按照当前实现直接执行，不会在此处重新遍历当前任务持有的其他互斥锁来计算应保留的继承优先级。

11.11 使用模式

11.11.1 基本互斥锁保护

```
#define MUTEX_ID  0

u8 shared_counter = 0;

void task_a(void)
{
    while (1)
    {
        if (os_mutex_lock(MUTEX_ID) >= 0)
        {
            shared_counter++;
            os_mutex_unlock(MUTEX_ID);
```

```
        }

        os_delay(10);
    }
}

void task_b(void)
{
    while (1)
    {
        if (os_mutex_lock(MUTEX_ID) >= 0)
        {
            shared_counter--;
            os_mutex_unlock(MUTEX_ID);
        }

        os_delay(10);
    }
}
```

示例假定互斥锁已经初始化，且两个任务均为合法的标准任务。调用者应根据实际等待机制确认加锁返回值，并确保只在成功持有锁时执行解锁。

11.11.2 优先级继承场景

假设存在三个任务：

- 低优先级任务：优先级 2，持有互斥锁。
- 中优先级任务：优先级 4，执行普通工作。
- 高优先级任务：优先级 6，尝试获取同一把互斥锁。

当高优先级任务等待低优先级任务持有的锁时，若满足优先级继承条件，低优先级任务的当前优先级将提升至 6。这样可以降低中优先级任务持续抢占锁所有者、延长高优先级任务等待时间的风险。

低优先级任务释放锁时，其当前优先级会按照当前实现恢复到基础优先级；随后，解锁流程将锁所有权转移给选中的等待任务。

11.12 性能考虑

11.12.1 锁获取路径

- **空闲路径：** 检查所有者状态，设置所有者并更新优先级。
- **竞争路径：** 执行所有者检查、优先级比较及必要的优先级更新，然后进入内核等待流程。

  竞争路径的额外开销取决于等待状态、优先级更新和调度过程。

11.12.2 解锁路径

- 检查调用者是否为所有者。

- 恢复当前所有者的基础优先级。
- 根据等待位图选择任务。
- 有有效等待任务时，执行所有权转移并调用 `wake_task()`。
  等待任务较多时，需要遍历标准任务位图以选择当前优先级最高的等待任务。

11.12.3 内存效率

- 互斥锁不使用独立的控制结构。
- 每个互斥锁使用一个统一 `OS_RESOURCE` 槽位。
- `OS_RESOURCE` 的结构大小为 7 字节，其中部分字段供其他 IPC 机制使用。
- 互斥锁使用的资源槽位与其他 IPC 对象共享，实际资源占用取决于整个系统的 IPC 配置。

11.13 与其他 IPC 机制的比较

11.13.1 互斥锁与信号量

- **互斥锁：** 具有所有权，只有所有者才能释放，并包含优先级继承处理。
- **信号量：** 主要用于同步或资源计数，是否具有所有权取决于具体实现。

11.13.2 互斥锁与二进制信号量

- **互斥锁：** 通过 `owner` 管理持有者，并限制释放者身份。
- **二进制信号量：** 通常用于事件同步，不必采用所有者机制。

11.13.3 互斥锁与自旋锁

- **互斥锁：** 通过等待机制处理锁竞争。
- **自旋锁：** 通过持续检查锁状态等待资源释放；HRTOS 当前互斥锁模块不采用这种方式。

11.14 常见注意事项

11.14.1 死锁风险

- 多把锁应尽可能按照统一顺序获取。
- 持锁期间应尽量缩短共享资源的访问时间。
- 避免持锁调用可能导致任务长期阻塞的 API。
- 不应在 ISR 上下文中使用互斥锁。

11.14.2 优先级继承边界

- 优先级继承用于缓解优先级反转，不等同于消除所有阻塞。
- 优先级继承不能防止死锁。
- 当前实现的解锁恢复逻辑直接将当前所有者恢复到基础优先级。
- 不应假定当前实现能够完整处理多把锁之间的优先级继承链。
- 实际最坏阻塞时间仍取决于临界区长度、任务行为、调度机制及其他系统约束。

Queue Design

HRTOS 消息队列设计

12.1 模块介绍

消息队列模块提供带缓冲区的任务间通信机制，用于在任务之间传递 1 字节消息。模块采用循环缓冲区管理消息，支持阻塞和非阻塞收发，并提供中断上下文使用的消息队列接口。

12.2 主要职责

消息队列模块负责：

- 使用用户提供的缓冲区初始化队列
- 消息发送与接收
- 队列清除
- 阻塞与非阻塞收发
- 中断上下文中的消息发送与接收

12.3 主要文件

12.3.1 源文件

- `Src/msgq/msgq_init.c`：消息队列初始化
- `Src/msgq/msgq_send.c`：消息发送
- `Src/msgq/msgq_recv.c`：消息接收
- `Src/msgq/msgq_clear.c`：消息队列清除
- `Src/interrupt/msgq_send_from_isr.c`：中断上下文消息发送
- `Src/interrupt/msgq_recv_from_isr.c`：中断上下文消息接收

12.3.2 头文件

- `Inc/msgq.h`：消息队列 API 声明
- `Inc/config.h`：消息队列结构定义及相关配置
- `Inc/hrtos_internal.h`：内核内部变量及接口声明

12.4 数据结构

12.4.1 os_msgq_t

消息队列控制结构定义于 `Inc/config.h`。

```
typedef struct {
    u8 *buf;        /* 用户缓冲区指针 */
    u8 _size;       /* 缓冲区大小 */
    u8 head;        /* 写入位置 */
    u8 tail;        /* 读取位置 */
    u8 count;       /* 当前消息数量 */
} os_msgq_t;
```

成员说明：

- `buf`：指向用户提供的消息缓冲区。
- `_size`：缓冲区容量。
- `head`：下一条消息的写入位置。
- `tail`：下一条消息的读取位置。
- `count`：当前队列中的消息数量。

12.4.2 队列数组

消息队列控制块数组：

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

相关配置：

- `OS_MSGQ_MAX`：最大消息队列数量，当前配置为 3。
- `OS_RESOURCE_MAX`：统一 IPC 资源数量上限，当前配置为 6。

消息队列数量与统一 IPC 资源数量是不同的配置，不能混为一谈。

12.4.3 内存占用

在当前 Keil C51 配置下，`u8 *buf` 占用 3 字节，其余 4 个成员各占 1 字节。

成员	大小
`buf`	3 字节
`_size`	1 字节
`head`	1 字节
`tail`	1 字节
`count`	1 字节
单个控制块	**7 字节**

当前配置下，3 个消息队列控制块共占用 21 字节，不包含用户提供的消息缓冲区。

12.5 核心函数

12.5.1 os_msgq_init()

位置： `Src/msgq/msgq_init.c`

功能： 将消息队列控制结构与用户缓冲区关联，并初始化队列状态。

参数：

- `q`：消息队列控制结构指针。
- `buf`：用户提供的消息缓冲区指针。
- `_size`：缓冲区容量。

处理过程：

- 保存缓冲区指针。
- 保存缓冲区容量。
- 将 `head` 和 `tail` 初始化为 0。
- 将 `count` 初始化为 0。

调用者应确保控制结构、缓冲区指针和缓冲区容量有效，并为队列提供足够的存储空间。

12.5.2 os_msgq_send()

位置： `Src/msgq/msgq_send.c`

功能： 向指定消息队列发送一条消息。

参数：

- `q`：消息队列控制结构指针。
- `obj`：关联的 IPC 资源编号，用于等待与唤醒。
- `_data`：待发送的 1 字节消息。
- `nonblock`：收发模式，`0` 表示阻塞，`1` 表示非阻塞。

返回值：

- `1`：发送成功。
- `0`：发送失败。

处理过程：

- 检查队列是否有可用空间。
- 队列已满时，根据 `nonblock` 决定立即返回失败或进入等待流程。
- 空间可用时，将消息写入 `head` 指向的位置。
- 更新 `head` 和 `count`。
- 按内核实现处理相关等待任务的唤醒。

阻塞模式下，调用者应结合等待机制的返回结果判断后续操作，不应假定等待结束就必然意味着发送成功。

12.5.3 os_msgq_recv()

位置： `Src/msgq/msgq_recv.c`

功能： 从指定消息队列接收一条消息。

参数：

- `q`：消息队列控制结构指针。
- `obj`：关联的 IPC 资源编号，用于等待与唤醒。
- `_data`：接收消息的存储地址。
- `nonblock`：收发模式，`0` 表示阻塞，`1` 表示非阻塞。

返回值：

- `1`：接收成功。
- `0`：接收失败。

处理过程：

- 检查队列中是否存在消息。
- 队列为空时，根据 `nonblock` 决定立即返回失败或进入等待流程。
- 有消息可读取时，从 `tail` 指向的位置读取数据。
- 更新 `tail` 和 `count`。
- 按内核实现处理相关等待任务的唤醒。

12.5.4 os_msgq_clear()

位置： `Src/msgq/msgq_clear.c`

功能： 清除队列当前保存的消息。

参数：

- `q`：消息队列控制结构指针。

处理过程：

- 将 `head` 重置为 0。
- 将 `tail` 重置为 0。
- 将 `count` 重置为 0。

清除操作用于丢弃队列中已有的消息。不要将其视为自动解除所有等待任务阻塞状态的操作；等待任务的处理应以实际内核实现为准。

12.5.5 os_msgq_send_from_isr()

位置： `Src/interrupt/msgq_send_from_isr.c`

功能： 在中断服务程序中发送消息。

参数：

- `q`：消息队列控制结构指针。
- `obj`：关联的 IPC 资源编号。
- `_data`：待发送的 1 字节消息。

返回值：

- `1`：发送成功。
- `0`：发送失败。

处理原则：

- 中断上下文中不应因队列已满而阻塞等待。
- 队列有空间时，写入消息并更新队列状态。
- 队列已满时，按接口约定返回失败。
- 与任务唤醒和调度相关的行为以实际 ISR 实现为准。

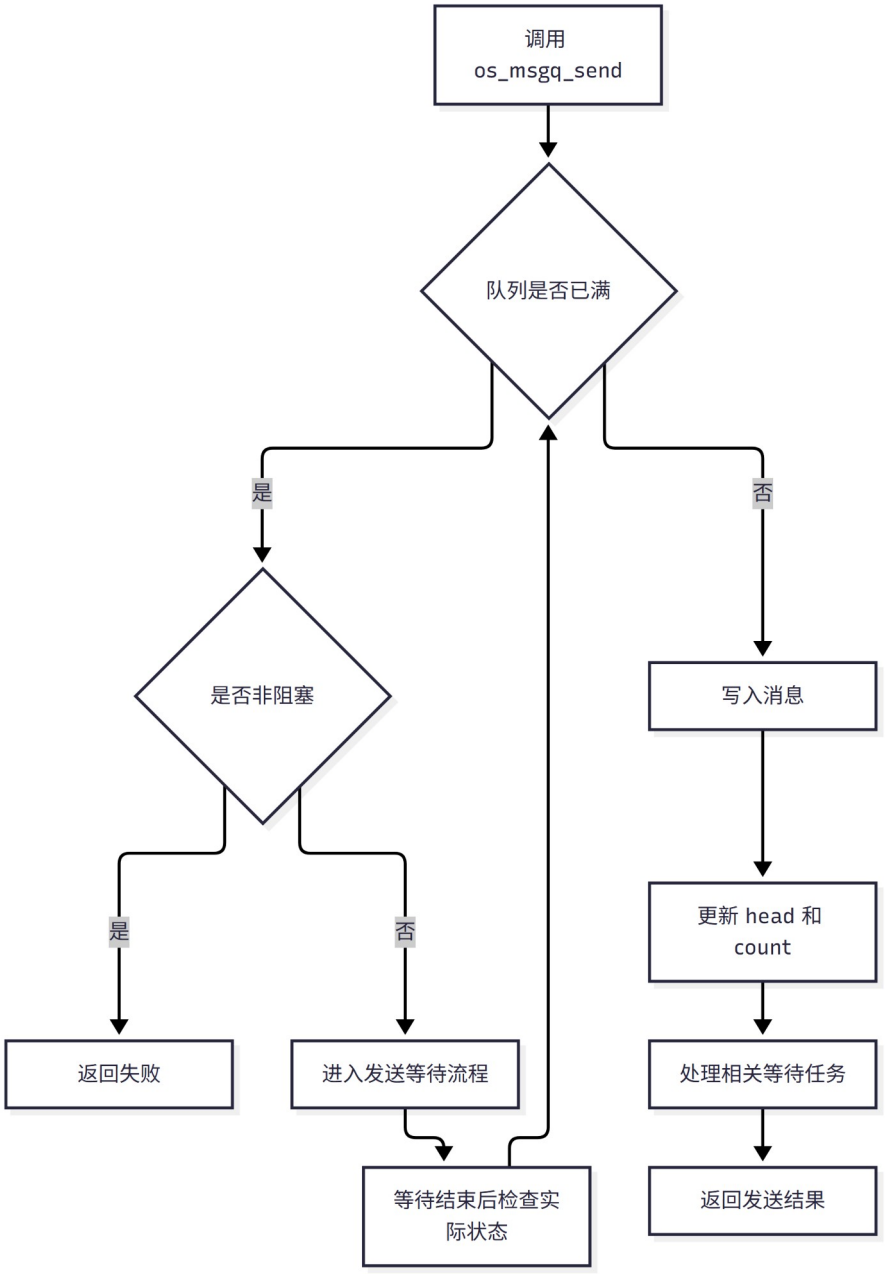
12.5.6 os_msgq_recv_from_isr()

位置： `Src/interrupt/msgq_recv_from_isr.c`

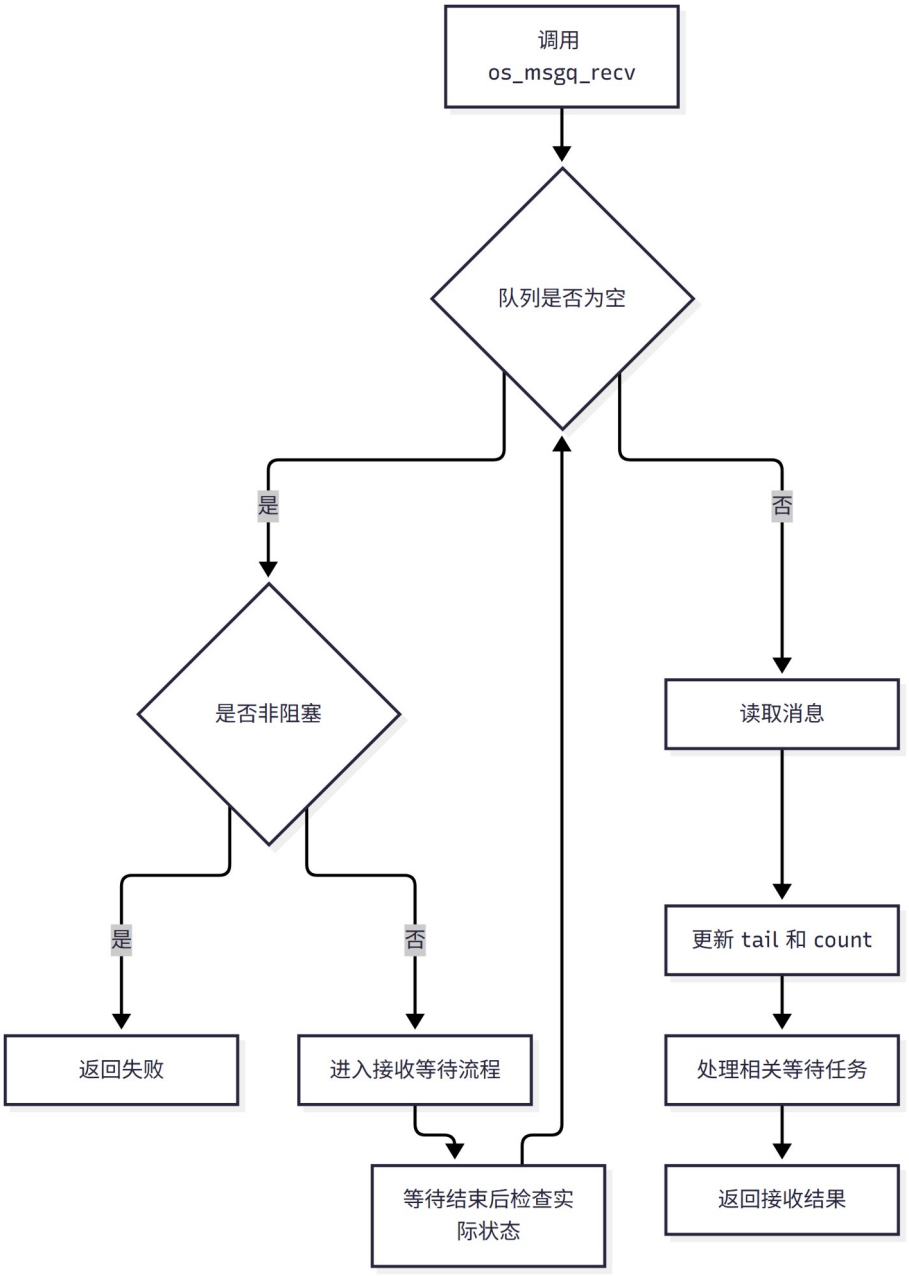
该接口用于中断上下文中的消息接收。其具体参数、返回值、队列状态处理和中断安全约束，应以该接口的实际实现及头文件声明为准。

12.6 调用关系

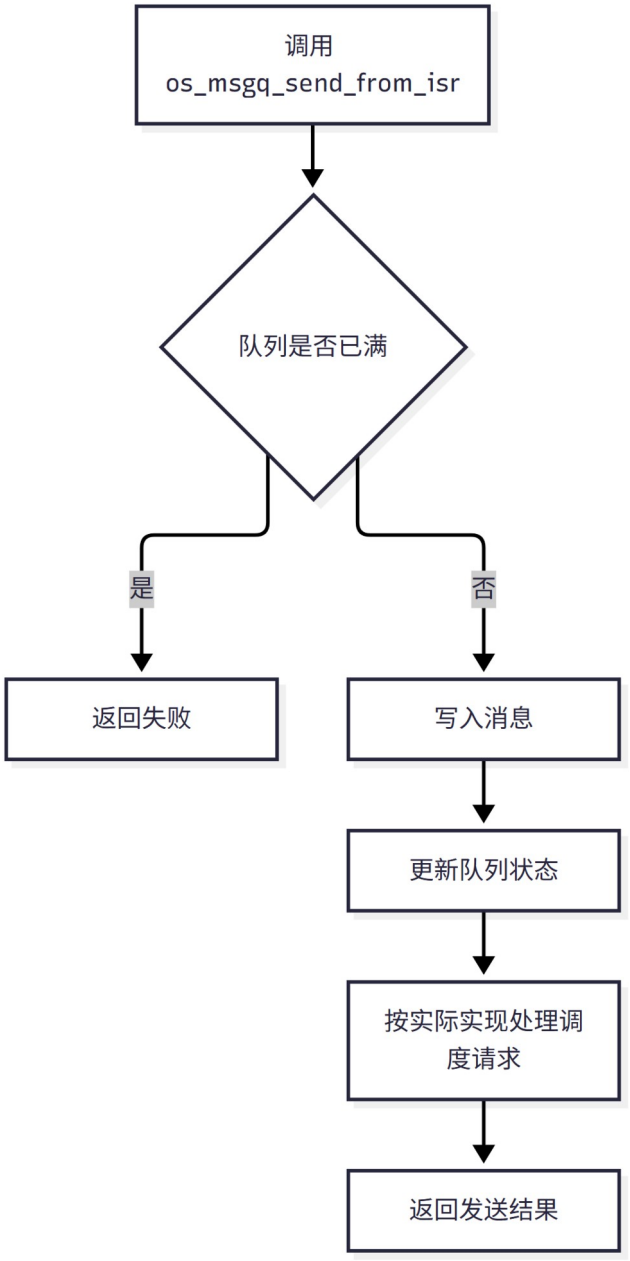
12.6.1 消息发送流程



12.6.2 消息接收流程


调用
os_msgq_recv
队列是否为空
是
否
是否非阻塞
是
否
读取消息
更新 tail 和 count
返回失败
进入接收等待流程
处理相关等待任务
等待结束后检查实
际状态
返回接收结果

12.6.3 中断发送流程



以上流程图用于说明主要处理路径。具体等待、唤醒及调度行为以对应源文件为准。

12.7 生命周期

12.7.1 队列生命周期

- **初始化：** 调用 `os_msgq_init()`，关联控制结构与用户缓冲区。
- **发送：** 调用发送接口，将消息写入循环缓冲区。
- **接收：** 调用接收接口，从循环缓冲区读取消息。
- **队列已满：** 发送操作根据阻塞模式处理。

- **队列为空：** 接收操作根据阻塞模式处理。
- **清除：** 调用 `os_msgq_clear()`，清空队列当前保存的消息。

12.8 设计原则

12.8.1 循环缓冲区

- `head` 指向下一条消息的写入位置。
- `tail` 指向下一条消息的读取位置。
- 指针在缓冲区末尾处循环回绕。
- `count` 记录当前消息数量。

12.8.2 用户提供缓冲区

- 消息存储空间由用户提供。
- 队列模块不依赖动态内存分配。
- 控制结构负责管理缓冲区指针、容量和读写状态。
- 缓冲区的有效性与生命周期由调用者负责。

12.8.3 阻塞与非阻塞

- **阻塞模式：** 当队列条件不满足时，可通过内核等待机制等待。
- **非阻塞模式：** 当队列条件不满足时，立即返回失败。
- 具体等待结果及唤醒行为遵循内核实现。

12.8.4 中断安全

- 提供独立的中断上下文消息队列接口。
- 中断上下文中不能因队列条件不满足而执行阻塞等待。
- 消息入队、出队及调度请求必须符合对应接口的实现约束。
- 不应仅凭接口名称推断所有中断上下文操作均安全。

12.8.5 统一等待机制

消息队列通过内核等待机制处理需要阻塞的收发操作。相关等待类型包括：

- `WAIT_MSG_SEND`：消息发送等待。
- `WAIT_MSG_RECV`：消息接收等待。

具体等待条件、唤醒时机及等待结果以实际内核代码为准。

12.9 约束

- 当前最多配置 3 个消息队列控制块。
- 每条消息为 1 字节。
- 消息缓冲区由用户提供。

- 消息按照 FIFO 顺序处理。
- 不提供基于消息优先级的排序。
- 中断上下文不能在队列已满或为空时阻塞等待。
- 不提供内置消息过滤或按条件选择消息的功能。

12.10 使用模式

以下示例用于说明典型调用方式。`g_queue`、`QUEUE_RES_ID`、`produce_data()` 和 `process_data()` 需要由应用程序自行定义；缓冲区应在队列初始化前准备好。

12.10.1 基本生产者—消费者

```
/* 用户提供的缓冲区 */
u8 g_queue_buf[16];

/* 消息队列控制结构 */
os_msgq_t g_queue;

/* 生产者任务 */
void producer_task(void)
{
    while (1)
    {
        u8 data = produce_data();

        if (os_msgq_send(&g_queue, QUEUE_RES_ID, data, 0) == 0)
        {
            /* 根据应用需求处理发送失败 */
        }

        os_delay(10);
    }
}

/* 消费者任务 */
void consumer_task(void)
{
    u8 data;

    while (1)
    {
        if (os_msgq_recv(&g_queue, QUEUE_RES_ID, &data, 0) == 1)
        {
            process_data(data);
        }
    }
}
```

12.10.2 非阻塞模式

```
/* 非阻塞发送 */
if (os_msgq_send(&g_queue, QUEUE_RES_ID, data, 1) == 0)
{
```

```
    /* 队列已满或发送失败 */
}

/* 非阻塞接收 */
if (os_msgq_recv(&g_queue, QUEUE_RES_ID, &data, 1) == 0)
{
    /* 队列为空或接收失败 */
}
```

12.10.3 中断到任务

```
/* 中断服务程序 */
void uart_rx_isr(void)
{
    u8 data = SBUF;

    if (os_msgq_send_from_isr(&g_queue, QUEUE_RES_ID, data) == 0)
    {
        /* 队列满时按应用需求处理 */
    }
}

/* 接收任务 */
void uart_task(void)
{
    u8 data;

    while (1)
    {
        if (os_msgq_recv(&g_queue, QUEUE_RES_ID, &data, 0) == 1)
        {
            process_uart_data(data);
        }
    }
}
```

应用程序应根据实际 API 声明确认返回值含义，并确保中断服务程序调用的接口满足当前内核的中断安全约束。

12.11 性能考虑

12.11.1 缓冲区容量

- 容量较大时，可以容纳更多暂存消息，但会占用更多 RAM。
- 容量较小时，内存占用较少，但更容易出现队列已满的情况。
- 容量应结合消息产生速率、消费速率及允许的最大积压量确定。

12.11.2 上下文切换

- 阻塞操作可能导致任务进入等待状态并触发调度。
- 非阻塞操作不因队列条件不满足而等待，但应用程序需要自行处理失败结果。
- 中断上下文中的消息操作应遵循对应接口的调度与唤醒规则。

12.11.3 内存效率

- 单个 `os_msgq_t` 控制结构占用 7 字节。
- 当前 3 个控制结构共占用 21 字节。
- 用户消息缓冲区的内存占用由缓冲区容量决定。
- 上述控制结构占用不包含其他 IPC 资源、任务控制块及内核等待机制的开销。

12.12 与其他 IPC 机制的比较

12.12.1 消息队列与事件

- **消息队列：** 在缓冲区中保存并传递具体数据。
- **事件：** 用于表示事件状态及进行任务同步，不承担多条数据的缓存传输。

12.12.2 消息队列与邮箱

- **消息队列：** 支持多条消息按 FIFO 顺序排队。
- **邮箱：** 通常用于传递单条消息，具体覆盖规则取决于邮箱实现。

12.12.3 消息队列与信号量

- **消息队列：** 兼顾数据传输与任务同步。
- **信号量：** 主要用于同步或资源计数，不负责保存消息内容。

Mailbox Design

HRTOS 邮箱设计

13.1 模块介绍

HRTOS 邮箱模块用于在任务之间传递单个数据，提供邮箱初始化、数据发送、数据接收、数据查询和邮箱清除等操作。

邮箱使用统一的 `OS_RESOURCE` 资源结构保存数据。其中，`value` 字段用于存储邮箱当前的数据，`wait_mask` 用于记录等待邮箱的任务。

邮箱采用单槽存储方式：发送操作将数据写入邮箱，接收操作读取数据并将邮箱清空。当邮箱为空时，接收任务可以通过统一等待机制进入等待流程。

13.2 主要职责

邮箱模块主要负责：

- 初始化邮箱资源。
- 向邮箱发送数据。
- 从邮箱接收数据。
- 查询邮箱当前保存的数据。
- 清除邮箱中的数据。
- 在发送数据时，检查等待队列并唤醒等待任务。

13.3 主要文件

13.3.1 源文件

- `Src/mailbox/mailbox_init.c`：邮箱初始化。
- `Src/mailbox/mailbox_send.c`：邮箱数据发送。
- `Src/mailbox/mailbox_receive.c`：邮箱数据接收。
- `Src/mailbox/mailbox_query.c`：查询邮箱当前数据。
- `Src/mailbox/mailbox_clear.c`：清除邮箱数据。

13.3.2 头文件

- `hrtos_internal.h`：内部资源结构、任务状态及相关内核接口声明。

13.4 数据结构

13.4.1 OS_RESOURCE 统一资源结构

邮箱复用 HRTOS 的统一资源结构：

```
typedef struct {
    u8  value;
    u8  owner;
    u8  wait_cnt;
    u16 wait_mask;
    u8  pending_signal;
} OS_RESOURCE;
```

各字段在邮箱模块中的用途如下：

字段	用途
`value`	保存邮箱当前的数据。值为 `0` 时，接收接口将邮箱视为空。
`owner`	邮箱不使用资源所有者机制，初始化时设置为 `OS_INVALID_ID`。
`wait_cnt`	初始化时清零，当前提供的邮箱代码没有使用该字段维护等待数量。
`wait_mask`	等待任务位图，用于查找等待接收邮箱数据的任务。
`pending_signal`	统一资源结构中的字段，当前提供的邮箱函数没有直接使用。

当前 `OS_RESOURCE_MAX` 为 6。邮箱与其他使用 `OS_RESOURCE` 的 IPC 对象共享资源池，可使用的邮箱数量取决于整体资源槽位的分配情况。

13.5 核心函数

13.5.1 os_mailbox_init()

函数原型

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

功能

初始化指定邮箱，将邮箱设置为空状态，并清除相关等待状态。

参数

- `mid`：邮箱资源槽位编号，有效范围为 `0` 至 `OS_RESOURCE_MAX - 1`。

返回值

无。

实现过程

1. 检查邮箱资源槽位编号。
2. 如果编号无效，直接返回。
3. 获取对应的 `OS_RESOURCE` 结构。
4. 将 `value` 设置为 `0`，表示邮箱为空。
5. 将 `owner` 设置为 `OS_INVALID_ID`。
6. 清零 `wait_mask` 和 `wait_cnt`。

该函数直接设置资源结构的相关字段，不调用其他资源初始化接口。

初始化操作应安排在邮箱投入使用之前，避免清除正在使用的邮箱数据或等待状态。

13.5.2 os_mailbox_send()

函数原型

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

功能

向指定邮箱写入一个数据，并在检测到等待任务时尝试唤醒其中一个任务。

参数

- `mid`：邮箱资源槽位编号。
- `content`：待发送的数据，类型为 `u8`，取值范围为 0–255。

返回值

- `1`：邮箱发送处理完成。
- `-1`：邮箱资源槽位编号无效。

实现过程

1. 检查邮箱资源槽位编号。
2. 关闭总中断，进入临界处理区。
3. 将 `content` 写入邮箱的 `value` 字段。
4. 检查 `wait_mask` 是否非零。
5. 如果存在等待任务，从任务位图中按任务 ID 从小到大的顺序查找第一个置位的任务。
6. 找到任务后，调用 `wake_task(next, WAIT_SIGNAL)`。
7. 开启总中断并返回 `1`。

如果没有等待任务，数据会保留在 `value` 字段中，供后续接收或查询操作使用。

设计特点

- 每次发送都会直接覆盖 `value`，不检查邮箱原先是否已有未读取的数据。
- 存在等待任务时，按任务 ID 从小到大的顺序查找第一个置位任务，而不是按照任务优先级选择。
- 当 `wait_mask` 非零但未找到有效任务时，当前实现仍返回 `1`。
- 返回 `1` 代表发送处理路径完成，不代表数据一定已经被接收任务读取。

13.5.3 os_mailbox_receive()

函数原型

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

功能

从指定邮箱读取数据。邮箱当前有数据时，读取并清空邮箱；邮箱为空时，调用统一等待接口进入等待流程。

参数

- `mid`：邮箱资源槽位编号。

返回值

- `-1`：邮箱资源槽位编号无效。
- 其他值：当前实现读取或返回的邮箱数据。

实现过程

1. 检查邮箱资源槽位编号。
2. 关闭总中断，获取对应的邮箱资源结构。
3. 如果 `value == 0`，开启总中断并调用 `os_wait(WAIT_MAILBOX, mid, 0)`。
4. 随后检查 `value` 是否非零。
5. 如果有数据，将数据保存到局部变量 `_data`，清空 `value`，开启总中断并返回 `_data`。
6. 如果此时 `value` 仍为零，则开启总中断并返回当前 `value`。

设计特点

- 邮箱有数据时，直接读取并清空邮箱。
- 邮箱为空时，调用 `os_wait()` 进入等待流程。
- 等待调用的超时参数为 `0`。
- 当前实现没有检查 `os_wait()` 的返回值。
- 等待调用之后，函数继续检查 `value`，并不会无条件地将等待返回值转换为接收成功。

使用注意

当前实现中，`value == 0` 同时表示邮箱为空，也可能表示发送的数据本身为 `0`。因此，`content` 为 `0` 时，邮箱无法仅通过 `value` 区分“邮箱为空”和“邮箱保存了数据 0”。

此外，阻塞路径的最终返回行为取决于 `os_wait()` 的实际执行机制及邮箱数据在等待期间的变化。调用方不应仅凭任务从等待状态恢复，就认定已经成功接收到有效数据。

13.5.4 os_mailbox_query()

函数原型

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

功能

查询指定邮箱当前保存的数据，不主动修改邮箱状态。

参数

- `mid`：邮箱资源槽位编号。

返回值

- `-1`：邮箱资源槽位编号无效。
- 其他值：当前 `OS_RESOURCE.value` 的值。

实现过程

1. 检查邮箱资源槽位编号。
2. 如果编号无效，返回 `-1`。
3. 否则直接返回 `OS_RES[mid].value`。

该接口只读取当前值，不清空邮箱，也不执行等待操作。

由于返回值 `0` 同时可能表示邮箱为空或邮箱中保存的数据为 `0`，因此不能单凭查询结果准确判断这两种状态。

13.5.5 os_mailbox_clear()

函数原型

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

功能

清除指定邮箱当前保存的数据，将 `value` 设置为 `0`。

参数

- `mid`：邮箱资源槽位编号。

返回值

- `1`：清除操作完成。
- `-1`：邮箱资源槽位编号无效。

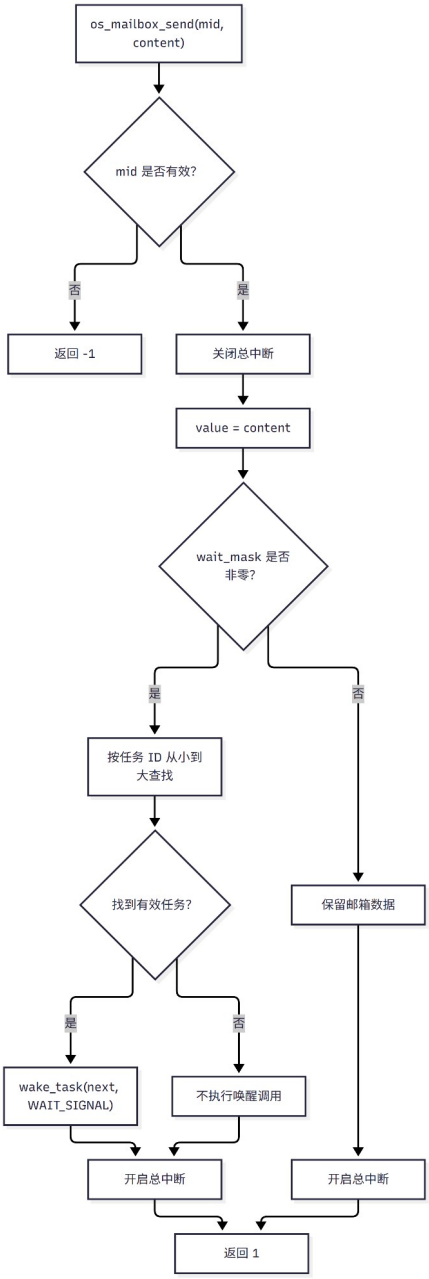
实现过程

1. 检查邮箱资源槽位编号。
2. 获取对应的 `OS_RESOURCE` 结构。
3. 关闭总中断。
4. 将 `value` 设置为 `0`。
5. 开启总中断并返回 `1`。

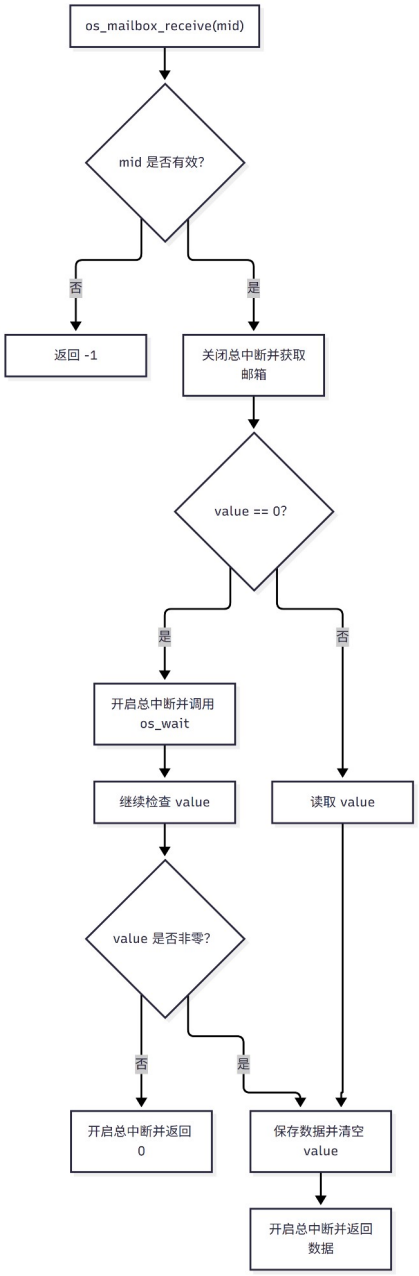
该函数只清除邮箱数据，不修改 `wait_mask`、`wait_cnt` 或 `owner`，也不负责唤醒等待任务。

13.6 调用流程

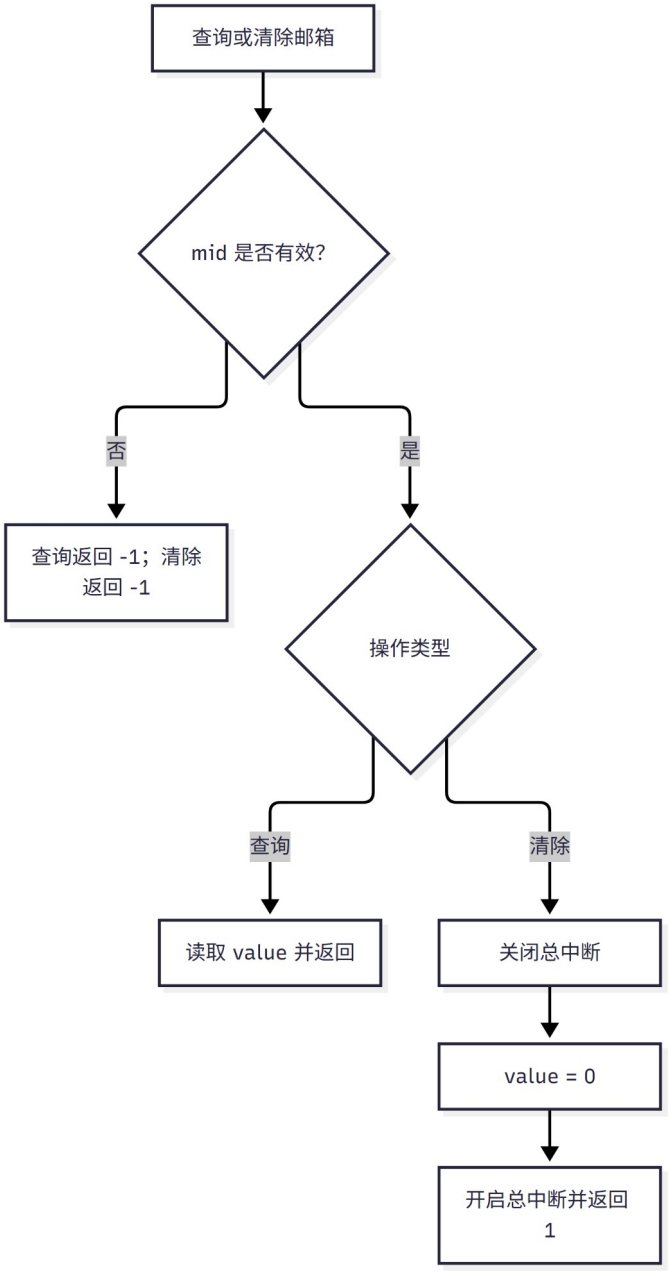
13.6.1 邮箱发送流程



13.6.2 邮箱接收流程


os_mailbox_receive(mid)
mid 是否有效?
否
是
返回 -1
关闭总中断并获取邮箱
value == 0?
是
否
开启总中断并调用 os_wait
继续检查 value
读取 value
value 是否非零?
否
是
开启总中断并返回 0
保存数据并清空 value
开启总中断并返回数据

13.6.3 邮箱查询与清除流程



13.7 生命周期

13.7.1 邮箱初始化

1. 系统初始化阶段调用 `os_mailbox_init()`。
2. 邮箱数据设置为 `0`。
3. 所有者字段设置为 `OS_INVALID_ID`。
4. 等待位图和等待计数清零。
5. 邮箱进入可供任务使用的初始状态。

13.7.2 邮箱发送与接收

1. 发送任务调用 `os_mailbox_send()`。
2. 发送函数将数据写入邮箱。
3. 如果检测到等待任务，按任务 ID 顺序选择一个任务并执行唤醒操作。
4. 接收任务调用 `os_mailbox_receive()`。
5. 邮箱数据非零时，接收函数读取数据并清空邮箱。
6. 邮箱数据为零时，接收函数进入统一等待流程。

13.7.3 邮箱查询与清除

- `os_mailbox_query()` 用于读取当前邮箱值，不改变邮箱内容。
- `os_mailbox_clear()` 用于清空当前邮箱值，不负责清除等待队列。

13.8 设计原则

13.8.1 单槽数据存储

邮箱通过单个 `u8 value` 字段保存数据，不包含独立的多条消息缓冲区。

因此，连续发送多个数据时，后一次发送会覆盖前一次保存的值。若应用需要缓存多条消息，应使用消息队列。

13.8.2 统一资源模型

邮箱复用 `OS_RESOURCE`，通过 `wait_mask` 关联等待任务，并使用 `os_wait()` 接入统一任务等待机制。

13.8.3 非优先级排序唤醒

邮箱发送函数按任务 ID 从小到大的顺序查找等待任务，不按照任务基础优先级或当前优先级排序。

13.8.4 查询与清除分离

- 查询操作仅读取当前值。
- 清除操作只将当前值设为 `0`。
- 接收操作在读取非零数据后清空邮箱。

三种接口的语义不同，使用时应根据实际目的进行选择。

13.9 使用约束

- 邮箱与其他 IPC 对象共享 `OS_RESOURCE` 资源池。
- 邮箱数据使用 `u8` 保存，范围为 0–255。

- 当前实现使用 `value == 0` 表示邮箱为空，因此数据 `0` 无法被可靠地区分为有效消息或空状态。
- 邮箱不具备消息队列的多条消息缓存能力。
- 发送操作会覆盖原有数据，不检查邮箱是否已有未读取内容。
- `os_mailbox_query()` 不修改邮箱状态，也不提供等待功能。
- `os_mailbox_clear()` 不清除等待位图，不负责唤醒等待任务。
- 邮箱不使用资源所有者机制。

13.10 使用模式

13.10.1 基本邮箱通信

以下示例展示邮箱的基本初始化、发送和接收方式。示例使用非零数据，以避免当前实现中 `0` 值与空状态无法区分的问题。

```
#define MAILBOX_ID  0

void mailbox_setup(void)
{
    os_mailbox_init(MAILBOX_ID);
}

void producer_task(void)
{
    u8 data = 1;

    while (1)
    {
        os_mailbox_send(MAILBOX_ID, data);
        os_delay(10);
    }
}

void consumer_task(void)
{
    int data;

    while (1)
    {
        data = os_mailbox_receive(MAILBOX_ID);

        if (data > 0)
        {
            process_data((u8)data);
        }
    }
}
```

该示例用于说明 API 的基本调用方式。实际应用还需要结合当前阻塞接收实现，确认任务恢复后的数据有效性，并处理发送覆盖、接收时序等问题。

13.10.2 查询邮箱数据

```
int data;

data = os_mailbox_query(MAILBOX_ID);

if (data > 0)
{
    /* 根据当前邮箱数据执行处理 */
}
```

查询操作不会消费数据，后续接收操作仍可能读取相同数据，除非其他操作修改了邮箱内容。

13.10.3 清空邮箱

```
if (os_mailbox_clear(MAILBOX_ID) == 1)
{
    /* 清除操作完成 */
}
```

该操作仅清空当前数据，不会清除等待任务位图。

13.11 性能考虑

13.11.1 直接读写

邮箱数据存储在 `OS_RESOURCE.value` 中，不涉及独立的数据缓冲区管理。发送操作直接写入一个字节，查询操作直接读取当前值。

13.11.2 等待任务查找

发送函数在 `wait_mask` 非零时，按任务 ID 顺序遍历 `OS_MAX_TASK` 范围内的任务位图，找到第一个置位任务后执行唤醒操作。

13.11.3 内存占用

邮箱不单独分配消息缓冲区，而是复用统一资源结构。邮箱的资源开销主要来自共享的 `OS_RESOURCE` 结构，不应将整个结构的大小误认为邮箱独立新增的内存开销。

13.12 与其他 IPC 的比较

13.12.1 邮箱与消息队列

- **邮箱：** 通过一个 `value` 字段保存单个数据，后续发送可能覆盖先前数据。
- **消息队列：** 使用缓冲区管理多条消息，适合连续的数据传递。

13.12.2 邮箱与信号量

- **邮箱：** 主要用于传递一个数据值。
- **信号量：** 主要用于计数同步和协调任务执行。

13.12.3 邮箱与互斥锁

- **邮箱：** 不记录资源所有者，重点在于数据传递。
- **互斥锁：** 用于共享资源的独占访问，具有所有权管理。

13.13 常见注意事项

13.13.1 避免发送覆盖

邮箱只保存一个当前值。如果发送速度高于接收速度，后续数据可能覆盖尚未读取的数据。需要保留连续消息时，应使用消息队列。

13.13.2 避免使用零值作为有效数据

当前实现使用 `value == 0` 判断邮箱为空。因此，发送数据 `0` 时，接收接口可能将其视为空状态。应避免将 `0` 作为有效邮箱数据，或在后续版本中增加独立的有效状态标志。

13.13.3 正确理解接收返回值

当邮箱为空时，接收函数调用 `os_wait()`，随后继续检查邮箱值。其返回结果依赖等待机制及数据变化，调用方应确认返回值确实代表有效数据后再处理。

13.13.4 区分清除与重新初始化

`os_mailbox_clear()` 只将 `value` 清零，而 `os_mailbox_init()` 还会重置所有者字段和等待相关字段。两者不能视为完全相同的操作。

Timer System

HRTOS 定时器系统手册

14.1 模块概述

HRTOS 定时器系统为内核提供统一的时间基准，负责系统 Tick 计数、任务延时、时间单位转换以及系统运行时间统计。该模块以 8051 的 Timer0 为基础，将硬件定时中断与内核调度、任务等待和唤醒机制连接起来，是 HRTOS 时间管理功能的重要组成部分。

定时器系统不仅用于周期性触发任务调度，还承担标准任务的延时管理和快速任务的独立延时计数。通过统一的系统时间基准，应用程序可以实现周期性任务、延时执行、运行时间监测等功能，而不必在每个任务中单独维护软件计时逻辑。

HRTOS 同时提供两种不同的延时方式：

- **系统 Tick 延时**：通过 `os_delay()` 使当前任务进入等待状态，将 CPU 让给其他可运行任务，并在延时到期后重新参与调度。
- **忙等待延时**：通过 `os_delay_ms()` 在当前执行路径中循环等待，期间持续占用 CPU，不主动让出处理器。

两种方式分别适用于任务级时间管理和短时间硬件时序控制，使用时应根据实时性要求和 CPU 占用情况进行选择。

14.2 模块职责

定时器系统主要负责以下工作：

- 基于 Timer0 生成周期性系统时钟中断。
- 管理系统 Tick 累计计数和时间片计数。
- 为标准任务提供基于 Tick 的阻塞延时。
- 维护快速任务的独立延时计数。
- 将毫秒、秒等时间单位转换为系统 Tick 数。
- 提供系统累计运行时间查询接口。
- 提供系统定时参数配置接口。
- 配合内核等待与唤醒机制，实现延时到期后的任务恢复。

14.3 文件组织

14.3.1 源文件

文件	主要职责
`Src/time/tick_config.c`	配置系统时间片参数和 Timer0 重载值
`Src/time/delay.c`	实现基于系统 Tick 的任务延时
`Src/time/delay_ms.c`	实现毫秒级忙等待延时
`Src/time/ms_to_tick.c`	将毫秒转换为系统 Tick 数
`Src/time/sec_to_tick.c`	将秒转换为系统 Tick 数
`Src/time/tick_get.c`	读取系统 Tick 累计计数
`Src/time/uptime_ms.c`	获取系统累计运行时间，单位为毫秒

14.3.2 相关头文件

文件	主要用途
`Inc/time.h`	对外提供时间管理 API 声明
`Inc/config.h`	定义默认时间片参数及相关配置常量
`Inc/hrtos_internal.h`	声明内核内部时间变量及相关接口

14.4 时间管理模型

HRTOS 的时间管理包含几个相互关联但职责不同的概念。

14.4.1 系统 Tick

系统 Tick 是内核用于计算时间和管理任务延时的基本计数单位。Timer0 按照配置产生周期性中断，每次有效的系统 Tick 推进都会更新累计计数，并检查相应的延时条件。

系统 Tick 的实际时间长度由 Timer0 重载值、定时器工作模式、定时器计数时钟以及具体的 8051 芯片配置共同决定。

需要注意，Tick 是软件层面的时间单位，并不天然等于一毫秒。只有在硬件时钟配置和软件换算关系成立时，才能将 Tick 换算为对应的实际时间。

14.4.2 时间片

时间片用于控制内核周期性调度的触发间隔。

系统通过 `OS_TIME_ONCE` 和 `OS_TIME_XY` 维护时间片相关状态：

- `OS_TIME_ONCE`：配置的时间片计数参数。

- `OS_TIME_XY`：当前时间片倒计数状态。
- `OS_SCHED_REASON`：内核调度相关状态之一，用于记录调度触发原因。

系统 Tick 与时间片不是同一个概念。一个时间片可以由多个 Tick 组成，时间片计数达到调度条件后，内核根据当前调度模式决定是否进行任务切换。

14.4.3 任务延时

任务延时以系统 Tick 为基础，通过内核统一等待机制实现。

标准任务调用 `os_delay()` 后，内核将其作为 `WAIT_DELAY` 类型的等待处理，并将延时计数与任务状态关联。任务在等待期间不参与普通就绪任务的执行竞争；延时到期后，内核将任务恢复到可调度状态。

快速任务的延时计数采用独立的 `OS_WAIT_DI2` 数组，不依赖标准任务的普通 TCB 延时字段。这种设计使快速任务可以使用独立的延时管理路径。

14.4.4 系统运行时间

系统运行时间由累计 Tick 数与当前配置的 Tick 时间共同换算得到。

HRTOS 通过 `os_uptime_ms()` 提供毫秒单位的运行时间查询接口。该接口返回 32 位无符号整数，适合用于运行时间显示、周期性监测和应用程序的时间统计。

需要注意，当前实现以整数毫秒表示单个 Tick 的时间，因此查询结果的精度受到整数换算方式的限制。

14.5 内部数据与配置

14.5.1 默认配置参数

相关配置位于 `Inc/config.h`。

```
#define OS_TIME_ONCE_DEFAULT 5
#define OS_TIME_T0 10000
```

`OS_TIME_ONCE_DEFAULT` 用于提供默认时间片计数配置，`OS_TIME_T0` 用于提供默认 Timer0 计数参数。

`OS_TIME_T0` 的数值本身并不能直接证明实际 Tick 周期为多少毫秒。实际周期还取决于定时器采用的计数时钟以及硬件工作模式。

14.5.2 时间片状态变量

```
volatile unsigned char idata OS_TIME_XY;
volatile unsigned char idata OS_T0_TH0;
volatile unsigned char idata OS_T0_TL0;
```

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

各变量的作用如下：

变量	说明
`OS_TIME_XY`	当前时间片计数状态
`OS_T0_TH0`	保存的 Timer0 高字节参数
`OS_T0_TL0`	保存的 Timer0 低字节参数
`OS_TIME_ONCE`	当前时间片计数配置
`OS_TIME_ONCE_BACKUP`	模式切换过程中使用的时间片备份状态

这些变量用于维护时间片调度和定时器配置状态。具体更新时机应结合内核初始化、Timer0 中断处理及调度模式切换代码理解。

14.5.3 系统 Tick 累计计数器

```
extern unsigned int xdata OS_TICK_COUNT_L;
extern unsigned int xdata OS_TICK_COUNT_H;
```

系统 Tick 使用两个 16 位变量保存累计计数：

- `OS_TICK_COUNT_L`：低 16 位。
- `OS_TICK_COUNT_H`：高 16 位。

组合后得到 32 位累计 Tick 数。

该计数方式适合 8051 资源受限的运行环境。读取两个部分时，需要避免 Timer0 中断在读取过程中更新计数器，否则可能得到不一致的组合值。

14.5.4 快速任务延时计数器

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

`OS_WAIT_DI2` 为快速任务提供独立的延时计数存储空间。

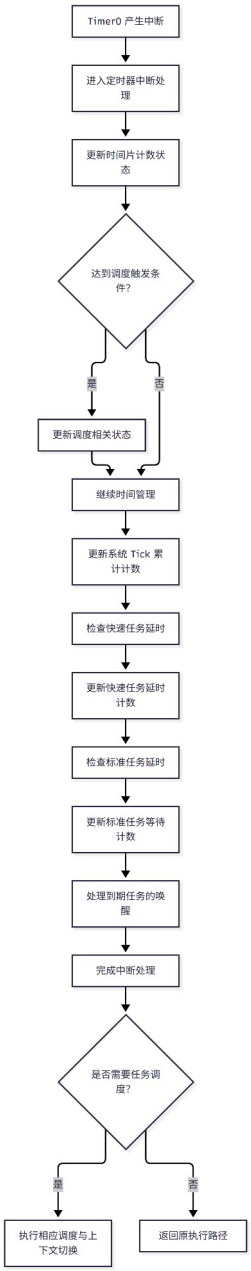
快速任务的延时管理与标准任务的 `wait_tick` 机制不同。分析快速任务的延时行为时，应以快速任务对应的计数更新、到期判断和唤醒实现为准，不能直接套用标准任务的 TCB 延时流程。

14.6 Timer0 定时机制

Timer0 为系统时间管理提供周期性硬件中断。每次中断发生后，内核通过定时器相关处理推进系统时间，并检查需要执行的调度和延时管理操作。

14.6.1 定时器中断处理流程

以下流程用于说明定时器系统与内核时间管理之间的主要关系。具体执行顺序、重载操作以及调度标志的更新细节，应以实际 Timer0 中断处理源码为准。



该机制将硬件定时事件与软件任务管理联系起来。定时器中断负责推进时间并识别到期条件，任务调度则根据内核状态和调度策略决定接下来运行哪个任务。

需要区分的是，发生 Timer0 中断不代表每次都会切换任务。是否切换取决于调度条件、当前运行状态及内核的具体调度实现。

14.7 核心 API

14.7.1 os_tick_config()

源文件： `Src/time/tick_config.c`

功能： 配置系统时间片计数参数、Timer0 重载参数，并清零系统 Tick 累计计数器。

函数声明：

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

参数说明：

参数	说明
`slice_count`	时间片计数配置，不允许为零
`tick_value`	Timer0 计数参数，必须大于 1000

返回值：

返回值	含义
`0`	配置成功
`-1`	参数无效

14.7.1.1 执行过程

函数首先检查 `slice_count` 是否为零。如果为零，直接返回 `-1`。

通过第一项检查后，函数将 `slice_count` 写入 `OS_TIME_ONCE`，随后检查 `tick_value` 是否大于 1000。

当 `tick_value` 满足要求时，函数执行以下操作：

- 将系统 Tick 累计计数的高 16 位清零。
- 将系统 Tick 累计计数的低 16 位清零。
- 根据输入的计数参数计算 Timer0 重载值。
- 将计算后的高、低字节分别写入 `OS_T0_TH0` 和 `OS_T0_TL0`。
- 返回 `0`，表示配置成功。

重载值的计算方式为：

```
tick_value = 65535 - tick_value;
tick_value++;
```

等价于在 16 位计数范围内计算：

```
reload_value = 65536 - tick_value;
```

其中 `reload_value` 表示写入定时器的重载值，而函数参数 `tick_value` 表示预期的计数长度。

14.7.1.2 使用注意事项

当前实现存在一个需要明确的行为：`OS_TIME_ONCE` 在 `tick_value` 完整校验之前就已被修改。因此，当 `slice_count` 合法但 `tick_value` 无效时，函数虽然返回 `-1`，时间片配置却已经发生变化。

另外，该函数清零的是系统 Tick 累计计数器，并没有在所提供的函数代码中显式重置 `OS_TIME_XY`。如果需要确认配置后的完整时间片初始状态，应继续检查定时器初始化与调度状态维护代码。

14.7.2 os_delay()

源文件： `Src/time/delay.c`

功能： 使当前标准任务等待指定数量的系统 Tick。

函数声明：

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

参数说明：

参数	说明
`tick`	延时 Tick 数

函数没有返回值。

14.7.2.1 执行过程

当 `tick` 为零时，函数直接返回，不执行延时操作。

当 `tick` 不为零时，函数首先检查当前任务编号是否满足：

```
OS_CURRENT_TASK < OS_PROCESS_MAX + 2
```

通过检查后，函数对 Timer0 的低、高字节寄存器执行赋值：

```
TL0 = 0x66;
TH0 = 0xFC;
```

随后调用统一等待接口：

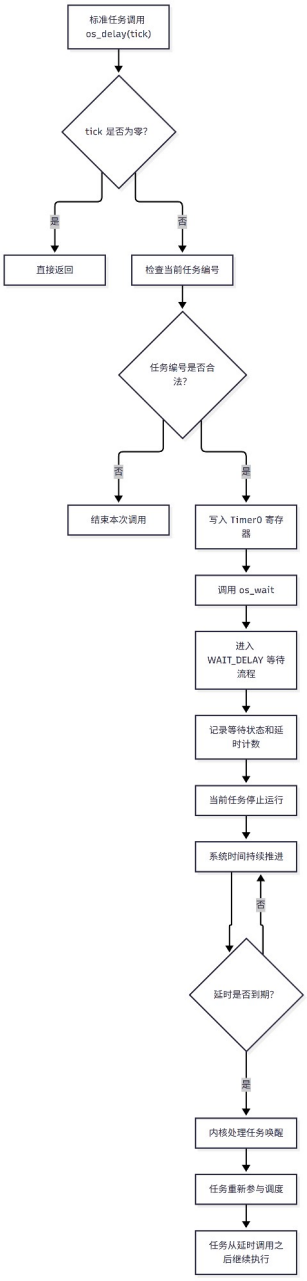
```
os_wait(WAIT_DELAY, OS_INVALID_ID, tick);
```

其中：

- `WAIT_DELAY` 表示延时等待类型。
- `OS_INVALID_ID` 表示该等待不关联具体的资源对象。
- `tick` 表示请求的延时计数。

通过统一等待机制，延时处理可以与其他任务等待操作共享内核的状态管理和唤醒流程。

14.7.2.2 任务延时流程



图中对等待状态及唤醒过程的描述，是对 `os_wait()` 与定时器时间管理之间关系的概括。任务状态变更、计数初始化和到期处理的具体细节，以对应内核源码为准。

14.7.2.3 设计特点

`os_delay()` 不是简单地执行循环等待，而是通过内核等待机制暂停当前任务的执行。

在等待期间，其他满足调度条件的任务可以继续运行，因此这种延时方式适合周期性任务、后台处理任务及需要主动让出 CPU 的应用场景。

14.7.2.4 使用示例

```
void periodic_task(void)
{
```

```
    while (1)
    {
        do_work();

        /* 等待 100 个系统 Tick */
        os_delay(100);
    }
}
```

这里的 `100` 表示 100 个系统 Tick，而不是无条件等于 100 毫秒。实际延时还与 Tick 周期、任务调度和唤醒时机有关。

14.7.3 os_delay_ms()

源文件： `Src/time/delay_ms.c`

功能： 通过软件循环实现毫秒级忙等待延时。

函数声明：

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

参数说明：

参数	说明
`ms`	请求的延时时间，单位为毫秒

函数没有返回值。

14.7.3.1 实现机制

函数使用嵌套循环执行延时：

```
void os_delay_ms(unsigned char ms)
{
    u8 i, j;

    while (ms--)
    {
        i = 3;
        j = 172;

        do
        {
            while (--j)
            {
                ;
            }
        }
        while (--i);
    }
}
```

每次外层循环执行一组固定的计数循环，通过消耗 CPU 执行时间实现软件延时。

该函数不调用 `os_wait()`，也不通过任务等待状态实现延时，因此不会主动让出 CPU 或主动触发任务调度。

14.7.3.2 适用场景

- 硬件初始化过程中需要短暂等待。
- 外设操作要求在当前执行路径中插入短延时。
- 某些硬件时序要求不适合通过任务阻塞实现。

14.7.3.3 使用注意事项

忙等待期间，当前执行路径持续占用 CPU。即使其他任务已经就绪，也不会因为该函数本身而主动发生任务切换。

实际延时时长取决于 CPU 工作频率、指令执行周期、编译器优化和生成的机器代码。循环参数本身不能保证在所有芯片、时钟和编译选项下都精确对应一毫秒。

因此，该函数适用于短时间等待，不应替代常规任务延时。对有严格时序要求的硬件操作，应通过实际测量或指令级分析确认延时精度。

14.7.4 os_ms_to_tick()

源文件： `Src/time/ms_to_tick.c`

功能： 将毫秒数转换为系统 Tick 数。

函数声明：

```
u16 os_ms_to_tick(u16 ms);
```

参数说明：

参数	说明
`ms`	待转换的时间，单位为毫秒

返回值为转换后的 Tick 数。

14.7.4.1 换算机制

函数首先根据保存的 Timer0 高、低字节重建重载值：

```
tick_value = OS_T0_TH0;
tick_value = tick_value << 8;
tick_value += OS_T0_TL0;
```

然后计算单个 Tick 对应的计数长度：

```
tick_time_us = 65536 - tick_value;
```

接着通过整数除法将计数长度换算为毫秒：

```
tick_ms = tick_time_us / 1000;
```

如果换算结果为零，则将 `tick_ms` 设置为 1。

最后使用向上取整的整数除法：

```
return (ms + tick_ms - 1) / tick_ms;
```

其数学含义为：

$$ticks = \left\lceil \frac{ms}{tick_ms} \right\rceil$$

在 `tick_ms` 为正数且计算未发生整数溢出的前提下，该算法可以保证转换结果不会因为除法截断而少于按整数毫秒计算所需的 Tick 数。

14.7.4.2 使用示例

```
u16 ticks;

ticks = os_ms_to_tick(500);
os_delay(ticks);
```

该示例先将 500 毫秒转换为 Tick 数，再调用 `os_delay()` 执行任务延时。

需要注意，转换精度受到整数毫秒换算的限制。当实际 Tick 周期不能被 1 毫秒整除时，当前实现会截去不足一毫秒的部分，结果可能与真实硬件时间存在偏差。

此外，`ms`、`tick_ms` 和中间表达式均受到 16 位整数范围的约束。对于接近类型上限的输入，应注意中间加法溢出的可能性。

14.7.5 os_sec_to_tick()

源文件： `Src/time/sec_to_tick.c`

功能： 将秒数转换为系统 Tick 数。

函数声明：

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

参数说明：

参数	说明
`sec`	待转换的秒数，类型为 8 位无符号整数

返回值为换算后的 Tick 数。

14.7.5.1 换算机制

函数先根据 Timer0 重载参数计算单个 Tick 对应的毫秒数：

```
tick_value = OS_T0_TH0;
tick_value <<= 8;
tick_value += OS_T0_TL0;

tick_ms = (65536 - tick_value) / 1000;
```

如果 `tick_ms` 为零，则设置为 1。

随后计算每秒对应的 Tick 数：

```
tick_per_sec = 1000 / tick_ms;
```

最后将每秒 Tick 数乘以输入秒数：

```
return tick_per_sec * sec;
```

其基本换算关系为：

$$ticks = / \frac{1000}{tick_ms} / \times sec$$

与 `os_ms_to_tick()` 不同，当前秒转换函数先计算每秒 Tick 数，再进行整数乘法，没有采用向上取整的方式。因此，在 Tick 周期无法整除一秒的情况下，转换结果可能低于实际所需的 Tick 数。

此外，函数输入为 `u8`，返回值为 `u16`。对于较大的秒数或者较短的 Tick 周期，乘法结果可能超出 16 位无符号整数的表示范围。

14.7.5.2 使用示例

```
u16 ticks;

ticks = os_sec_to_tick(5);
os_delay(ticks);
```

该示例将 5 秒换算为 Tick 数，再让当前任务进入相应的延时等待。

实际应用中，应根据系统 Tick 周期及整数换算精度判断是否适合直接使用该接口。

14.7.6 os_tick_get()

源文件： `Src/time/tick_get.c`

功能： 读取系统 Tick 累计计数的高、低 16 位。

函数声明：

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

参数说明：

参数	说明
`high`	用于接收 Tick 累计计数的高 16 位
`low`	用于接收 Tick 累计计数的低 16 位

函数没有返回值，结果通过指针参数输出。

14.7.6.1 执行过程

函数先关闭 Timer0 中断使能：

```
ET0 = 0;
```

随后读取高、低计数部分：

```
*high = OS_TICK_COUNT_H;
*low  = OS_TICK_COUNT_L;
```

最后重新设置：

```
ET0 = 1;
```

通过在读取过程中暂时禁止 Timer0 中断，可以避免该中断在两个计数部分读取之间修改累计计数，从而降低读取结果不一致的风险。

14.7.6.2 使用注意事项

当前实现直接在读取完成后将 `ET0` 置为 1，并没有保存调用前的中断使能状态。因此，它并非严格意义上的“保存并恢复原中断状态”。

如果调用该函数之前 `ET0` 已经为 0，函数返回前仍会将其置为 1。对于需要保持既有中断状态的调用环境，应注意这一实现细节。

此外，调用者需要提供有效的 `high` 和 `low` 指针。

14.7.7 os_uptime_ms()

源文件： `Src/time/uptime_ms.c`

功能： 获取系统累计运行时间，单位为毫秒。

函数声明：

```
u32 os_uptime_ms(void);
```

返回值为 32 位无符号整数，表示根据系统累计 Tick 数和当前 Tick 毫秒长度换算得到的运行时间。

14.7.7.1 执行过程

函数首先暂时关闭 Timer0 中断：

```
ET0 = 0;

tick_h = OS_TICK_COUNT_H;
tick_l = OS_TICK_COUNT_L;

ET0 = 1;
```

随后将高、低 16 位组合为 32 位累计 Tick 数：

```
total_tick = ((u32)tick_h << 16) | tick_l;
```

函数再根据 Timer0 重载参数计算单个 Tick 对应的计数长度，并换算为整数毫秒：

```
tick_value = OS_T0_TH0;
tick_value <<= 8;
tick_value += OS_T0_TL0;

tick_us = 65536 - tick_value;
tick_ms = tick_us / 1000;
```

```
if (tick_ms == 0)
{
    tick_ms = 1;
}
```

最后计算累计运行时间：

```
return total_tick * tick_ms;
```

其换算关系为：

$$uptime_ms = total_tick \times tick_ms$$

14.7.7.2 精度与范围

该函数提供的是基于累计 Tick 数计算的运行时间，而不是直接读取独立的高精度硬件时钟。

由于单个 Tick 的时间采用整数毫秒表示，实际 Tick 周期中不足一毫秒的部分不会保留在换算结果中。因此，运行时间越长，整数换算产生的累计偏差越值得关注。

返回值为 `u32`，当累计运行时间超出 32 位无符号整数的表示范围时，结果将发生整数回绕。

与 `os_tick_get()` 一样，当前实现没有保存调用前的 `ET0` 状态，而是在读取结束后直接将其置为 1。调用环境需要考虑这一点。

14.8 系统时间与任务调度的关系

定时器系统与调度器之间存在明确的职责分工。

Timer0 提供周期性硬件事件；时间管理代码维护累计 Tick 和等待计数；内核根据延时到期情况恢复任务状态；调度器则决定后续由哪个满足运行条件的任务获得 CPU。

14.8.1 延时到期处理

标准任务的延时请求通过 `os_wait()` 进入统一等待机制。任务等待期间，系统时间继续推进，内核根据其等待计数判断是否到期。

当延时到期时，任务被恢复为可调度状态。任务何时真正继续执行，取决于调度器的选择和当前系统中的其他可运行任务。

因此，`os_delay(tick)` 应理解为请求至少经过相应 Tick 计数的任务延时，而不应将其视为具有绝对精确结束时刻的硬件定时器。

14.8.2 时间片触发与任务切换

时间片计数用于产生周期性调度条件，但时间片到期不等同于必然发生任务切换。

在时间片调度模式下，时间片到期可以成为调度触发条件；在纯优先级调度等其他模式下，任务是否切换还受到相应调度规则的影响。

具体的任务切换行为，应结合 HRTOS 调度器的模式定义和调度触发实现理解。

14.9 初始化与配置流程

定时器系统需要与内核启动流程协同工作。

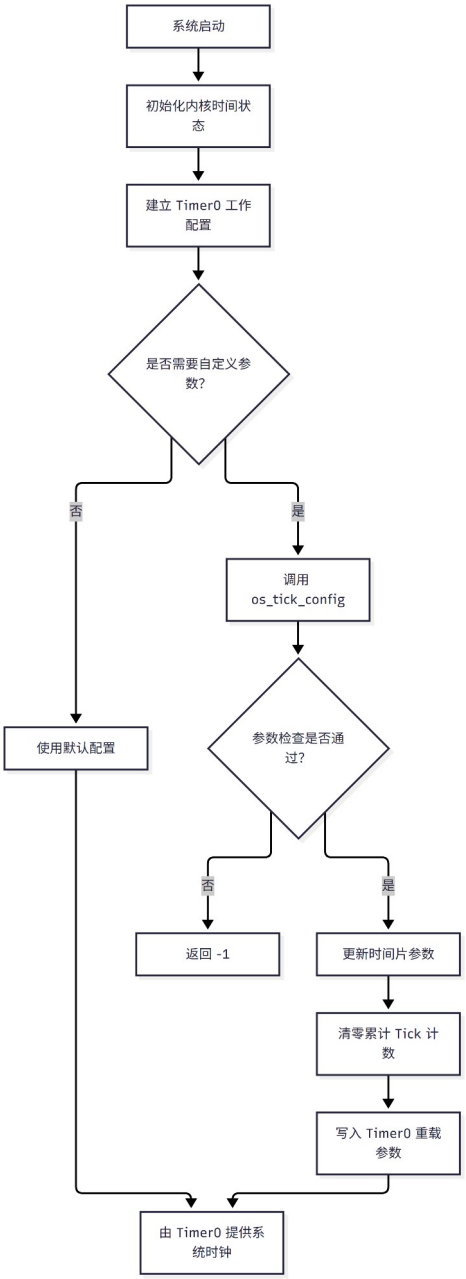
14.9.1 定时器初始化

系统启动时，内核初始化与时间管理有关的状态，并根据实际配置建立 Timer0 工作环境。

用户如需调整默认时间片计数或 Timer0 重载参数，可以调用 `os_tick_config()`。配置成功后，函数会更新相应参数并清零累计 Tick 计数器。

Timer0 的启动、中断使能、工作模式配置和实际寄存器初始化，还取决于系统启动代码及目标芯片的硬件配置。

14.9.2 配置流程



需要注意，`os_tick_config()` 本身的执行内容不等同于完整的 Timer0 硬件初始化过程。不能仅凭该函数就推断 Timer0 的所有工作模式、启动位和中断使能状态均已完成配置。

14.10 使用模式

14.10.1 周期性任务

对于需要定期执行工作的任务，可以在完成一次工作后调用 `os_delay()`，避免使用长时间忙等待占用 CPU。

```
void periodic_task(void)
```

```
{
    while (1)
    {
        do_work();

        os_delay(100);
    }
}
```

示例中的延时参数以系统 Tick 为单位。若任务要求严格的固定周期，还需要考虑任务执行时间、调度延迟和 Tick 周期带来的误差。

14.10.2 毫秒时间转换

当应用需求以毫秒表达时，可以先使用 `os_ms_to_tick()` 进行换算，再通过 `os_delay()` 等待。

```
void task_with_ms_delay(void)
{
    u16 ticks;

    ticks = os_ms_to_tick(500);

    while (1)
    {
        do_work();

        os_delay(ticks);
    }
}
```

该方式可以让应用代码以毫秒描述目标延时，但实际精度仍受系统 Tick 周期和整数换算方式限制。

14.10.3 运行时间监测

`os_uptime_ms()` 可以用于读取系统累计运行时间，并将结果提供给显示、日志或监测逻辑。

```
void monitor_task(void)
{
    u32 uptime;

    while (1)
    {
        uptime = os_uptime_ms();

        display_uptime(uptime);

        os_delay(1000);
    }
}
```

此处的 `os_delay(1000)` 表示等待 1000 个系统 Tick，而不是固定等待 1000 毫秒。如果目标是约一秒的任务间隔，可以根据当前 Tick 配置使用时间转换接口。

14.10.4 短时间硬件等待

在硬件初始化或者外设操作过程中，如果需要在当前执行路径中插入短延时，可以使用忙等待接口。

```
void init_hardware(void)
{
    /* 为硬件稳定预留短暂时间 */
    os_delay_ms(10);
}
```

使用该接口时，应确认忙等待期间占用 CPU 不会影响其他任务及中断处理的时序要求。

14.11 设计特点与实现边界

14.11.1 基于 Timer0 的统一时间基准

HRTOS 使用 Timer0 为系统提供周期性时间事件，并以软件计数维护系统 Tick 和相关时间状态。

统一时间基准使任务延时、时间片管理和运行时间统计可以共享同一套时间参数，减少不同功能之间的计时配置差异。

该设计也意味着系统时间管理与 Timer0 的配置存在依赖关系。修改定时器参数时，需要同时考虑任务延时、调度周期和时间转换函数的行为。

14.11.2 统一等待机制

标准任务延时通过 `os_wait()` 进入统一等待流程，采用 `WAIT_DELAY` 类型，并使用 `OS_INVALID_ID` 表示不关联具体等待对象。

这种实现将任务延时纳入内核统一的等待和唤醒管理体系，而不是为普通延时单独建立一套完整的任务状态管理流程。

14.11.3 快速任务独立延时

快速任务通过 `OS_WAIT_DI2` 数组管理延时计数，与标准任务的普通延时状态分开维护。

这种分离有利于保持快速任务管理路径的独立性，但其实际响应时间和执行开销仍取决于快速任务对应的中断处理、计数更新与唤醒实现。

14.11.4 32 位累计 Tick

系统将累计 Tick 拆分为高、低两个 16 位变量，以适应 8051 的数据处理特点。

该方案扩展了计数范围，同时要求读取时注意中断并发问题。当前接口通过暂时关闭 Timer0 中断读取两个计数部分，但没有保存并恢复原有的 `ET0` 状态。

14.11.5 忙等待与任务延时并存

HRTOS 同时提供任务级阻塞延时和软件忙等待延时，分别面向不同使用场景。

任务级延时适合常规应用逻辑，可以让出 CPU；忙等待适合短时间硬件时序操作，但会持续消耗处理器资源。两者不能简单互相替代。

14.12 配置约束与注意事项

14.12.1 Tick 周期与硬件计时关系

实际 Tick 周期由目标芯片的定时器工作模式、计数时钟和重载值共同决定。

在配置系统 Tick 前，应确认目标芯片的 Timer0 时钟来源与计数规则，不能直接将重载参数解释为毫秒数。

14.12.2 时间换算精度

当前毫秒和秒转换接口均使用整数毫秒作为中间单位。

当 Tick 周期不能以整数毫秒准确表示时，换算结果可能产生误差。特别是 `os_sec_to_tick()` 采用每秒 Tick 数的整数除法，不能保证所有配置下都向上取整。

对于需要严格控制实际等待时长的场景，应先验证目标硬件下的 Tick 周期及换算误差。

14.12.3 延时范围

`os_delay()` 使用 `u16` 表示延时 Tick 数，单次输入范围受到 16 位无符号整数的限制。

毫秒和秒转换函数也受到输入类型、返回类型及中间运算范围限制。较大的时间值可能出现整数溢出或回绕。

14.12.4 中断使能状态

`os_tick_get()` 和 `os_uptime_ms()` 均直接操作 `ET0`，当前实现会在读取结束后将其设置为 1，而不是恢复调用前的状态。

如果函数可能在 Timer0 中断原本关闭的环境中调用，应特别关注这一行为。

14.12.5 Timer0 寄存器访问

当前 `os_delay()` 在调用 `os_wait()` 前写入 `TL0` 和 `TH0`。这一操作属于现有实现的一部分，分析其对定时器运行的影响时，需要结合 Timer0 的实际工作模式及内核相关代码进行确认。

14.12.6 忙等待的 CPU 占用

`os_delay_ms()` 不会主动触发任务调度，因此不适用于长时间等待或需要主动让出 CPU 的常规任务逻辑。

其延时精度也依赖实际生成的机器代码，不能仅凭函数参数保证严格的毫秒级精度。

14.13 性能与实时性考虑

定时器系统的性能主要受到 Tick 中断频率、单次中断处理开销、延时任务数量以及任务切换开销的共同影响。

14.13.1 中断频率

在其他条件相同的情况下，较短的 Tick 周期会提高定时器中断频率，从而增加系统的中断处理开销。

Tick 周期越短并不意味着整个系统必然越快。应根据任务延时粒度、调度需求和处理器负载确定合适的参数。

14.13.2 延时精度

`os_delay()` 的时间粒度受系统 Tick 周期限制，并受到中断处理、任务唤醒及调度等待的影响。

`os_delay_ms()` 使用忙等待方式，可以在不主动让出 CPU 的情况下执行短时间延时，但它的精度取决于执行环境，不能据此直接认定其在所有配置下都具有亚毫秒精度。

14.13.3 任务切换开销

任务进入延时等待后，内核需要完成等待状态管理，并在到期时处理唤醒和后续调度。

因此，任务延时的实际返回时刻不仅取决于计数器何时到期，还与任务恢复运行的时机有关。对于具有明确截止时间的任务，应将这些因素纳入系统时序分析。

14.14 与其他定时方案的区别

14.14.1 相对于独立硬件定时器

HRTOS 的系统 Tick 用于内核统一时间管理，主要服务于任务延时、时间片调度和运行时间统计。

独立硬件定时器则可以用于特定外设的周期事件、输入捕获、输出比较或其他硬件时序功能。两者的职责不同，不能将系统 Tick 等同于所有外设定时能力。

14.14.2 相对于软件定时器回调机制

当前提供的时间管理接口以 Tick 计数和任务延时为主要功能，不应据此认为系统已经提供了完整的软件定时器回调框架。

需要周期性执行的应用逻辑，可以由任务结合 `os_delay()` 实现；更复杂的定时器对象、回调注册与取消机制，则需要以其他实际模块的实现为准。

14.14.3 相对于高分辨率定时机制

当前时间转换接口将 Tick 计数换算为整数毫秒，因此其时间表示精度受到该算法的限制。

如果应用需要更细粒度的时间测量或更精确的硬件事件控制，应根据目标芯片的计时资源和系统架构选择合适的实现方式，而不应将当前毫秒换算结果直接视为高分辨率时间戳。

14.15 模块总结

HRTOS 定时器系统围绕 Timer0 建立统一的时间管理基础，将系统 Tick、时间片计数、任务延时、快速任务延时以及累计运行时间统计连接起来。

其中，`os_delay()` 负责通过内核等待机制实现任务级延时，`os_delay_ms()` 提供不主动让出 CPU 的软件忙等待，`os_ms_to_tick()` 和 `os_sec_to_tick()` 提供时间单位转换，`os_tick_get()` 与 `os_uptime_ms()` 则分别用于获取累计 Tick 和毫秒单位的运行时间。

该模块的设计重点是共享时间基准、复用内核等待机制，并在有限的 8051 资源条件下完成基础时间管理。实际使用时，应重点关注目标芯片的 Timer0 配置、Tick 换算精度、整数运算范围及中断使能状态等实现边界。

Interrupt Design

HRTOS 中断设计

15.1 模块介绍

HRTOS 中断模块负责管理中断相关的内核操作，为临界区保护、ISR 上下文识别、中断 ID 获取、中断嵌套以及中断环境下的进程间通信提供基础支持。

中断处理程序与普通任务运行在不同的执行上下文中。ISR 需要快速完成硬件状态处理，并在必要时通知内核中的任务。如果中断处理程序直接执行不适合中断上下文的操作，可能导致任务调度异常、内核状态不一致或中断响应延迟。

因此，HRTOS 通过独立的中断接口、调度请求机制和中断相关状态管理，协调硬件中断处理与内核任务调度。

中断模块的设计重点是保护内核共享状态、识别当前执行上下文、提供适用于 ISR 的通知接口，并将需要执行的任务调度交由内核处理。

15.2 主要职责

15.2.1 临界区管理

提供临界区进入和退出接口，用于保护需要避免中断并发访问的共享数据。

15.2.2 ISR 上下文识别

提供 ISR 上下文检测能力，使内核或应用程序能够区分中断处理环境与普通任务环境。

15.2.3 中断标识管理

提供中断 ID 查询接口，用于识别当前中断来源，为中断分发、调试和特定中断处理逻辑提供依据。

15.2.4 ISR 环境下的进程间通信

提供适用于中断上下文的事件设置、消息发送、信号量释放以及邮箱发送等操作，支持 ISR 向任务传递通知或数据。

15.2.5 调度请求

允许 ISR 请求内核执行后续调度处理。调度请求与直接调用任务调度器不同，其目的是将调度操作交由内核规定的处理路径完成。

15.2.6 中断嵌套支持

配合内核中断状态、优先级和上下文管理机制，支持 HRTOS 设计中规定的嵌套中断处理流程。

15.3 主要文件

15.3.1 源文件

文件	主要职责
`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`	检查当前是否处于 ISR 上下文
`Src/interrupt/interrupt_id.c`	获取当前中断 ID
`Src/interrupt/schedule_request.c`	从 ISR 请求内核调度
`Src/interrupt/event_set_from_isr.c`	从 ISR 设置事件
`Src/interrupt/msgq_send_from_isr.c`	从 ISR 发送消息
`Src/interrupt/msgq_recv_from_isr.c`	ISR 相关的消息接收实现，具体调用约束以接口实现为准
`Src/interrupt/queue_send_from_isr.c`	ISR 环境下的邮箱发送与任务唤醒
`Src/interrupt/semaphore_post_from_isr.c`	从 ISR 释放信号量

其中，`queue_send_from_isr.c` 应描述为 ISR 环境下的邮箱发送与任务唤醒，不应将其误述为通用的 ISR 安全队列发送接口。

15.3.2 头文件

文件	主要职责
`Inc/interrupt.h`	中断相关 API 声明
`Inc/config.h`	临界区相关宏及配置
`Inc/hrtos_internal.h`	内核内部中断状态和变量声明

15.4 内部数据结构

15.4.1 临界区状态变量

临界区相关变量在 `Inc/hrtos_internal.h` 中声明。

```
extern bit os_int_status;
extern unsigned char idata os_critical_nesting;
```

`os_int_status` 用于保存首次进入临界区时的中断使能状态。

`os_critical_nesting` 用于记录临界区嵌套深度。嵌套计数使多层调用能够共享同一套临界区保护机制，避免内部调用提前恢复中断。

由于这两个变量属于内核共享状态，使用临界区接口时应遵循成对进入和退出的原则，并确保嵌套关系正确。

15.4.2 中断控制变量

以下变量位于 `Src/kernel/os_core.c`，用于中断状态、上下文保护或中断处理流程中的内部控制。

```
volatile bit OS_ISR_FLAG;
volatile bit OS_INTERRUPT_GAO;
volatile bit OS_INTERRUPT_GAO_BIAO;
volatile unsigned char data OS_INTERRUPT_PROTECT;
volatile unsigned char data OS_INT_BAO[6];
```

这些变量涉及 ISR 状态标记、高优先级中断处理标记、中断 ID 保护以及中断上下文信息保存等用途。

`OS_INT_BAO` 的注释所列内容包括 ACC、B、DPH、DPL、PSW 和 EA。实际保存、恢复顺序以及各变量的使用条件，应以对应的汇编代码和内核实现为准。

`volatile` 用于提示编译器，这些变量可能在当前代码路径之外被修改或观察。但它本身不提供互斥保护，也不能代替正确的中断同步机制。

15.4.3 中断处理地址

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

这两个数组用于保存中断处理相关的地址信息，分别对应普通中断处理和嵌套中断处理所涉及的地址。

具体地址的编码格式、写入时机、读取方式以及使用条件，应结合中断入口和嵌套中断处理代码理解。

15.5 核心函数

15.5.1 os_enter_critical()

源文件： `Src/interrupt/enter_critical.c`

功能： 进入临界区，保护需要避免中断并发访问的内核或应用程序共享状态。

其基本处理逻辑如下：

```
void os_enter_critical()
{
    if(os_critical_nesting == 0) {
        os_int_status = EA;
        EA = 0;
    }

    os_critical_nesting++;
}
```

首次进入临界区时，函数保存当前 EA 状态，并关闭总中断使能。再次进入时，只增加嵌套计数，不重复覆盖原始中断状态。

这种设计允许多个内部函数在嵌套调用过程中使用临界区保护，而不必各自独立保存和恢复中断状态。

设计特点

- 通过嵌套计数支持多层临界区调用。
- 仅在首次进入时保存中断状态并关闭中断。
- 保留进入临界区之前的 EA 状态。
- 避免内层临界区提前恢复中断。

使用约束

临界区进入和退出必须匹配。不能在未正确进入的情况下调用退出接口，也不能让嵌套计数因调用错误而失去一致性。

15.5.2 os_exit_critical()

源文件： `Src/interrupt/exit_critical.c`

功能： 退出临界区，并在最外层临界区结束时恢复保存的中断状态。

其基本处理逻辑如下：

```
void os_exit_critical()
{
    os_critical_nesting--;

    if(os_critical_nesting == 0) {
        EA = os_int_status;
```

```
    }
}
```

退出操作首先减少嵌套计数。当计数归零时，恢复首次进入临界区时保存的 EA 状态。

恢复的是原始状态，而不是无条件将 EA 设置为 1。因此，如果进入临界区之前总中断使能已经关闭，退出后仍应保持关闭状态。

设计特点

- 仅在最外层临界区退出时恢复中断状态。
- 保持嵌套调用之间的保护关系。
- 避免临界区结束后错误改变调用前的中断状态。

使用约束

必须保证进入和退出次数匹配。若在计数为零时错误调用退出接口，计数可能发生下溢，导致后续临界区行为异常。具体防护行为应以实际实现为准。

15.5.3 os_in_isr()

源文件： `Src/interrupt/in_isr.c`

功能： 检查当前执行环境是否属于 ISR 上下文。

该接口用于区分普通任务执行路径与中断执行路径，可为接口调用约束、调试检查以及上下文相关的处理逻辑提供依据。

返回值

- `1`：当前处于 ISR 上下文。
- `0`：当前不处于 ISR 上下文。

具体的 ISR 状态判定方式由内部标志及其维护逻辑决定。使用该接口时，应结合 HRTOS 对中断入口、嵌套中断和中断退出的实现理解其语义。

15.5.4 os_interrupt_id()

源文件： `Src/interrupt/interrupt_id.c`

功能： 获取当前中断 ID。

该接口用于识别当前中断来源，适用于需要根据中断编号执行不同处理逻辑的场景。

中断 ID 的具体取值由 HRTOS 的中断入口和编号映射规则决定。使用时应以当前版本源码中的实际映射为准，不能直接将传统 8051 的硬件向量顺序等同于 HRTOS 的中断 ID。

在当前实现的相关测试中，外部中断 INT0 对应 ID `0`，INT1 对应 ID `2`。其他中断源的编号应依据实际实现确认。

15.5.5 os_schedule_request()

源文件： `Src/interrupt/schedule_request.c`

功能： 从 ISR 请求内核执行后续调度处理。

接口的基本逻辑如下：

```
char os_schedule_request(char id)
{
    id = id;
    OS_SCHED_REASON = 1;
    TF0 = 1;

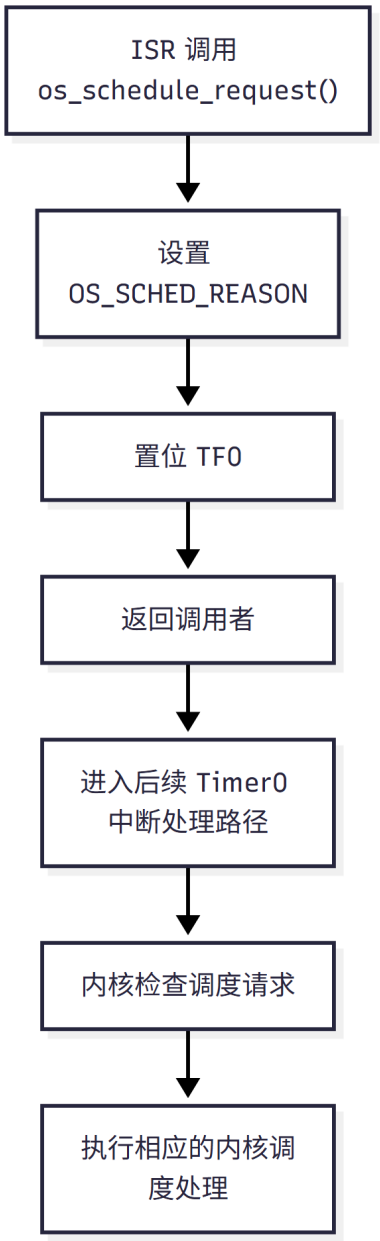
    return 1;
}
```

参数 `id` 为保留参数，当前实现中不参与实际调度决策。

函数通过设置调度触发标志，并置位 Timer0 溢出标志，触发后续的 Timer0 中断处理路径。

这种方式将调度请求与实际调度过程分离，使内核能够在规定的中断处理路径中完成后续操作，而不是在当前 ISR 中直接调用调度器。

处理流程



设计特点

- 调度请求与实际调度执行分离。
- 使用 Timer0 触发后续处理。
- 避免在调用接口的位置直接执行完整调度流程。
- 为中断环境下的任务通知和调度请求提供统一入口。

该机制并不意味着调度一定在当前硬件 ISR 返回后立即发生。实际执行时机取决于中断优先级、当前中断状态、Timer0 配置及内核中断处理流程。

15.6 ISR 环境下的进程间通信

15.6.1 os_event_set_from_isr()

源文件： `Src/interrupt/event_set_from_isr.c`

功能： 在 ISR 环境下设置事件，使任务能够通过相应的事件机制获知中断产生的状态变化。

参数

`id`：事件资源 ID。

返回值

按当前接口约定，成功返回 `1`，失败返回 `-1`。

处理逻辑

函数首先检查事件资源 ID 是否有效，然后更新对应的事件状态，并根据具体实现处理相关的挂起信号或唤醒请求。需要执行后续调度时，通过 `os_schedule_request()` 请求内核处理。

事件设置接口用于传递状态通知。若多个事件可能在任务处理前连续发生，应用程序应根据事件机制的实际语义设计处理逻辑，避免错误假定每一次事件都对应独立的计数记录。

15.6.2 os_semaphore_post_from_isr()

源文件： `Src/interrupt/semaphore_post_from_isr.c`

功能： 在 ISR 环境下释放信号量，用于通知任务某项操作已经完成，或某个资源已具备继续处理的条件。

参数

`obj`：信号量资源 ID。

返回值

按当前接口约定，成功返回 `1`，失败返回 `-1`。

处理逻辑

函数检查信号量资源是否合法，然后根据信号量当前状态更新其计数或相关挂起信号。当操作需要引起任务调度时，通过调度请求机制交由内核进一步处理。

信号量释放与任务实际运行之间存在调度过程。调用成功表示接口接受了相应操作，不应据此假定被通知任务已在 ISR 返回前执行。

15.6.3 os_msgq_send_from_isr()

源文件： `Src/interrupt/msgq_send_from_isr.c`

功能： 在 ISR 环境下向消息队列发送消息。

参数

- `q`：消息队列指针。
- `obj`：相关资源 ID。
- `_data`：待发送的消息数据。

返回值

按当前接口约定，发送成功返回 `1`；发送失败或队列已满时返回 `0`。

处理逻辑

函数根据消息队列当前状态判断是否可以发送消息。如果队列已满，则返回失败，不在 ISR 中等待队列空间。如果队列可用，则写入消息、更新队列相关状态，并根据实际实现请求后续调度处理。

使用注意事项

- 调用前应确保队列对象有效。
- ISR 中不应等待队列空间。
- 应明确消息队列满载时的处理策略。
- 如果消息不能丢失，应通过队列容量规划、硬件状态处理或其他应用层机制保证数据完整性。

15.6.4 os_msgq_recv_from_isr()

源文件： `Src/interrupt/msgq_recv_from_isr.c`

该文件属于 ISR 相关的消息接收实现。

消息接收与消息发送具有不同的上下文约束。不能仅根据文件名就认定该接口可以在所有 ISR 中安全调用，也不能在未核对实现的情况下推断其是否阻塞、是否修改调度状态或适用于哪些中断场景。

使用该接口前，应检查实际函数声明、返回值、队列空状态处理及相关调用约束，并据此补充面向应用程序的具体说明。

15.6.5 queue_send_from_isr.c

源文件： `Src/interrupt/queue_send_from_isr.c`

功能： ISR 环境下的邮箱发送与任务唤醒。

该文件涉及邮箱通信和等待任务的唤醒处理，不应将其描述为通用的 ISR 安全队列发送接口。

在中断环境下发送邮箱消息时，内核需要依据邮箱状态及等待关系处理相应的通信操作。若存在等待该邮箱的任务，还需要按照实际等待对象和内核调度机制执行唤醒处理。

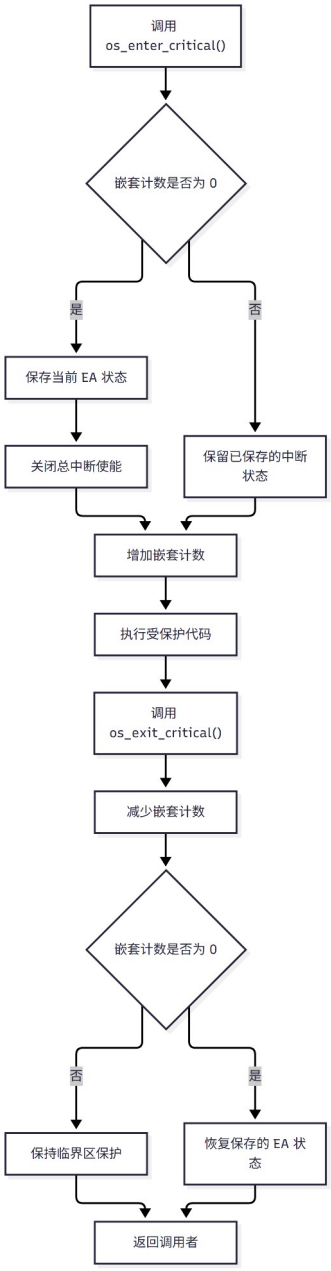
邮箱发送、等待任务唤醒和后续任务调度是相互关联但不同的处理阶段。接口完成消息发送或唤醒操作，并不等于被唤醒任务立即开始运行。

具体的参数、返回值、消息存储方式和唤醒条件，应以该文件的函数实现及相关邮箱数据结构为准。

15.7 调用关系

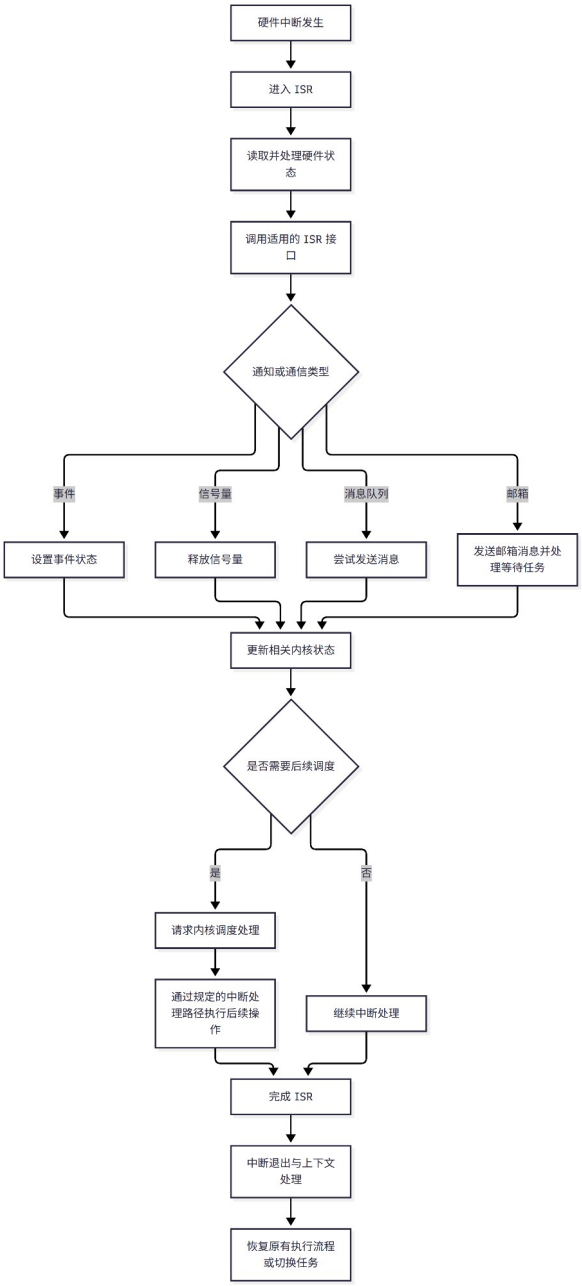
15.7.1 临界区处理流程

临界区采用嵌套计数管理。首次进入时保存原始中断状态并关闭总中断使能；内层进入只增加计数；退出时逐层减少计数；只有计数归零时才恢复原始中断状态。



15.7.2 ISR 通知与任务调度流程

ISR 安全接口负责在中断环境下完成允许的状态更新或通信操作。若需要执行后续任务调度，接口可以请求内核通过调度处理路径继续执行。



实际调度行为由内核实现决定。上图展示的是逻辑关系，并不意味着所有 ISR 接口都会无条件触发调度，也不意味着每次中断退出都会发生任务切换。

15.8 生命周期

15.8.1 临界区生命周期

临界区的完整生命周期由进入、嵌套保护、受保护代码执行以及退出组成。

- **进入临界区**：调用 `os_enter_critical()`。
- **检查嵌套状态**：判断当前是否为首次进入。
- **保存中断状态**：首次进入时保存当前 EA 状态。

- **关闭中断**：首次进入时关闭总中断使能。
- **执行受保护代码**：在临界区内访问需要保护的共享状态。
- **退出临界区**：调用 `os_exit_critical()`。
- **更新嵌套状态**：递减嵌套计数。
- **恢复中断状态**：嵌套计数归零时恢复进入前保存的状态。

15.8.2 ISR 执行生命周期

ISR 的执行涉及硬件中断响应、处理程序执行以及内核对中断退出和调度请求的处理。

- **中断产生**：外设或处理器中断源产生中断请求。
- **进入中断处理程序**：处理器根据硬件配置转入相应中断入口。
- **处理硬件状态**：读取状态寄存器并按芯片要求处理或清除中断请求。
- **执行必要的通知操作**：根据需要调用适用的 ISR 相关接口。
- **提出调度请求**：当操作需要后续内核调度处理时，调用相应调度请求接口。
- **完成 ISR 处理**：完成当前中断源所需的处理。
- **执行内核中断退出流程**：由 HRTOS 的中断处理机制完成必要的状态处理。
- **继续原执行流程或切换任务**：根据中断状态和调度条件决定后续执行路径。

需要区分硬件中断返回与 HRTOS 内核调度处理。具体的上下文保存、恢复和任务切换顺序，应以目标芯片上的实际中断入口及内核代码为准。

15.9 设计原则

15.9.1 嵌套临界区

HRTOS 使用嵌套计数管理临界区的进入与退出。

- 首次进入时保存原始中断状态。
- 嵌套进入时不重复覆盖原始状态。
- 内层退出时不提前恢复中断。
- 最外层退出时恢复原始中断状态。
- 所有进入与退出操作必须保持匹配。

临界区的实际安全性还取决于共享状态的访问方式、调用上下文以及编译器和处理器对相关操作的实现。

15.9.2 ISR 安全

ISR 相关接口应遵循中断上下文约束。

- 避免在 ISR 中执行阻塞操作。
- 避免在 ISR 中等待不可立即获得的资源。
- 只调用明确支持当前中断上下文的接口。

- 将复杂业务处理交给普通任务完成。
- 控制 ISR 执行时间，减少对其他中断和任务的影响。

“ISR 相关”或函数名包含 `_from_isr`，并不意味着可以忽略参数有效性、资源状态及具体接口的调用约束。

15.9.3 延迟调度

HRTOS 提供独立的调度请求接口，用于将调度请求交由内核处理。

当前 `os_schedule_request()` 实现通过设置 `OS_SCHED_REASON` 并置位 `TF0` 请求 Timer0 中断处理路径。

这种方式将请求发起与后续调度处理分离，但具体的响应时机仍取决于当前中断状态、Timer0 的配置以及内核的中断处理逻辑。

15.9.4 临界区宏

`Inc/config.h` 中的临界区宏用于统一调用临界区接口。

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

使用 `do { ... } while (0)` 可以让宏在语法上表现为单条语句，降低其嵌入条件语句或其他代码块时产生的语法问题。

调用宏时，仍需确保进入和退出操作匹配，并遵循临界区接口的调用约束。

15.10 约束与注意事项

15.10.1 临界区约束

- 嵌套计数使用 8 位变量，其可表示范围有限。
- 临界区进入与退出必须配对。
- 临界区内不应执行不必要的耗时操作。
- 关闭总中断使能会影响可响应中断的时机。
- 嵌套计数的有效范围和异常处理能力，应以实际实现为准。

15.10.2 ISR 调用约束

- ISR 不应调用阻塞式延时或等待接口。
- 不应在 ISR 中等待互斥资源。
- 不应在 ISR 中执行长时间计算。
- 不应在 ISR 中进行不可控的动态内存操作。
- 不应在未确认接口约束前调用普通任务接口。
- 消息队列发送等操作应采用非阻塞处理方式。

- 消息接收接口的 ISR 使用条件必须根据具体实现确认。

15.10.3 调度约束

- ISR 不应自行绕过内核调度流程。
- 调度请求接口用于发起请求，不等于立即完成任务切换。
- 调度行为取决于内核状态及相应中断处理流程。
- Timer0 的使用方式应与系统时基和调度配置保持一致。

15.10.4 中断嵌套约束

嵌套中断涉及硬件优先级、内核中断状态和上下文保护机制。不能仅根据任务优先级或中断 ID 推断硬件中断一定可以抢占另一中断。

具体的嵌套条件、允许的调用范围以及退出要求，应以 HRTOS 中断入口、退出处理和目标芯片的中断配置为准。

15.11 ISR 编程规范

15.11.1 ISR 中适合执行的操作

- 读取必要的硬件寄存器。
- 按芯片要求处理或清除中断源。
- 更新简单的状态标志。
- 调用明确支持 ISR 上下文的事件接口。
- 调用明确支持 ISR 上下文的信号量接口。
- 调用符合约束的非阻塞消息发送接口。
- 调用适用的邮箱发送与任务唤醒接口。
- 在需要时请求内核进行后续调度处理。

上述操作是否适用，仍需结合对应接口的具体实现和调用条件判断。

15.11.2 ISR 中应避免的操作

- 调用阻塞式延时或等待接口。
- 等待消息、信号量或其他资源变为可用。
- 使用需要阻塞等待的互斥机制。
- 执行耗时且无法预测的业务逻辑。
- 执行未经确认可在 ISR 中使用的动态内存操作。
- 递归调用内核接口。
- 在未确认安全性的情况下直接调用任务调度器。

15.11.3 ISR 编程示例

以下示例用于说明常见的中断处理结构，实际寄存器名称、清除方式、事件 ID 和中断声明应根据目标芯片及项目配置调整。

```
void example_isr(void) interrupt VECTOR
{
    u8 status;

    /* 读取硬件状态 */
    status = HW_STATUS_REG;

    /* 按芯片要求处理或清除中断源 */
    HW_CLEAR_REG = status;

    /* 使用适用的 ISR 接口通知任务 */
    if(status & EVENT_FLAG) {
        os_event_set_from_isr(EVENT_ID);
    }

    /* 完成本次中断处理 */
}
```

该示例不表示所有芯片都采用相同的中断标志清除方式，也不表示事件接口在所有配置下都会触发任务切换。具体行为应根据硬件手册及 HRTOS 接口实现确定。

15.12 性能考虑

15.12.1 临界区开销

临界区涉及中断状态保存、总中断使能控制以及嵌套计数更新。

首次进入和最终退出需要处理保存与恢复逻辑；嵌套进入和退出则涉及计数更新。实际开销取决于编译器生成的指令、目标芯片执行模式及具体代码路径。

15.12.2 中断响应延迟

关闭总中断使能期间，依赖 EA 控制的中断可能无法立即进入处理程序。因此，临界区持续时间会影响中断响应时机。

设计时应控制临界区长度，避免在临界区内执行长时间循环、等待操作或不必要的函数调用。

15.12.3 调度请求开销

基于 Timer0 的调度请求涉及调度标志更新和中断触发操作，后续还需要经过相应的中断处理路径。

这种机制能够将请求发起与调度处理分离，但会产生额外的中断处理开销。实际延迟应通过目标硬件测量，不宜仅凭函数结构推断。

15.12.4 ISR 执行时间

ISR 的执行时间不仅影响当前中断源的处理延迟，也会影响其他中断的响应和任务的运行时机。

对实时性要求较高的应用，应尽可能将复杂处理放在任务上下文中执行，并通过事件、信号量、消息队列或邮箱等机制传递处理请求。

15.13 中断优先级与调度级别

15.13.1 8051 硬件中断优先级

传统 8051 系列处理器通常包含外部中断、定时器中断和串行口中断等中断源。具体中断源及其硬件优先级由芯片型号和中断配置决定。

不能将所有 8051 兼容芯片的中断优先级固定描述为同一顺序。部分增强型 8051 芯片还提供扩展中断源、额外优先级或特殊中断控制功能。

因此，实际中断优先级应以目标芯片的数据手册和项目配置为准。

15.13.2 HRTOS 内核调度级别

HRTOS 使用内核调度级别区分普通任务、高优先级执行路径以及中断相关处理级别。

```
#define OS_SCHED_LEVEL_MAX 10
```

当前内核设计中，调度级别的概念包括普通任务优先级、高优先级任务以及中断和嵌套中断相关级别。

需要注意，**内核调度级别不等同于 8051 硬件中断优先级**。两者作用于不同的控制机制，不能直接互换或简单对应。各级别的准确取值和语义，应以当前版本内核定义及实际调度实现为准。

15.14 与其他 RTOS 的设计差异

不同 RTOS 在临界区、中断嵌套和 ISR 延迟处理方面采用的策略可能不同，具体取决于处理器架构、实时性目标和内核设计。

15.14.1 临界区与中断屏蔽

HRTOS 的临界区接口通过保存 EA 状态并关闭总中断使能来实现相应的保护。

部分其他 RTOS 或处理器平台可能采用选择性中断屏蔽、优先级阈值或专用中断屏蔽寄存器。不同方案在保护范围、响应延迟和实现复杂度上存在差异。

15.14.2 中断嵌套

中断嵌套能力取决于硬件中断优先级配置、中断入口实现和内核上下文管理方式。

评价嵌套中断机制时，应关注允许的嵌套条件、上下文保存与恢复、共享状态保护和异常退出处理，而不宜仅以“支持”或“不支持”概括。

15.14.3 ISR 延迟调度

HRTOS 提供通过调度请求接口发起后续内核处理的机制，当前实现通过 Timer0 触发相应处理路径。

其他 RTOS 可能使用软件中断、专用调度中断或其他架构相关机制。不同方案的实际性能应结合处理器架构、中断配置和测量结果比较，不能仅根据机制名称判断优劣。

15.15 总结

HRTOS 中断模块围绕临界区保护、ISR 上下文识别、中断 ID 管理、中断嵌套以及中断环境下的通信与调度请求展开。

临界区机制通过嵌套计数维护中断状态的保存与恢复；ISR 相关接口为事件通知、信号量释放、消息发送以及邮箱通信提供支持；调度请求机制则将调度请求与后续内核处理分离。

在应用设计中，应明确区分硬件中断优先级与内核调度级别，正确匹配临界区进入与退出，并严格遵循各接口的调用上下文约束。

对于邮箱发送、任务唤醒、嵌套中断和调度时机等实现细节，应以 HRTOS 当前版本源码和目标硬件测试结果为准，确保系统行为与实际实现一致。

Memory Layout

HRTOS 内存布局

16.1 模块介绍

HRTOS 采用固定地址布局管理 8051 内部 RAM 和 XDATA 全局内存池。内部 RAM 主要用于内核临时变量、系统栈、中断栈、用户任务栈以及调度状态；XDATA 全局内存池用于保存任务状态、任务上下文、栈指针记录、时间数据、任务控制块、资源对象和消息队列对象。

本文以当前源码中的绝对地址声明、偏移宏、数组长度及结构体定义为依据，说明实际内存分配范围。仅存在偏移宏或注释、没有实际存储对象声明的区域，不计入已分配内存。

固定地址布局的正确性取决于以下条件：

- 对象起始地址与偏移宏一致。
- 数组长度与后续对象的起始地址相匹配。
- 结构体实际大小与固定地址间距一致。
- 内存区域不发生地址重叠。
- 目标芯片具有相应的 RAM 容量和寻址能力。

16.2 主要文件

文件	主要职责
`Src/kernel/os_core.c`	内核固定地址变量及 XDATA 对象的定义
`Src/kernel/os_data.c`	DATA 区域相关初始化
`Src/kernel/os_xdata.c`	XDATA 区域相关初始化
`Src/kernel/task_create.c`	任务创建及用户栈分配
`Src/kernel/task_cleanup.c`	任务清理及用户栈释放
`Inc/config.h`	任务数量、栈深度及内核容量配置
`Inc/hrtos_internal.h`	内核变量、任务控制块和内核对象结构定义

16.3 内存空间概览

HRTOS 当前源码涉及以下存储区域。

存储区域	地址范围	用途
内部 RAM 低地址区	`0x00-0x7F`	寄存器组、位寻址 RAM、内核变量及栈
内部 RAM 高地址区	`0x80-0xFF`	通过 IDATA 间接寻址访问的内部 RAM，具体取决于芯片
XDATA 全局内存池	`0x0200-0x03FF`	任务上下文、控制块及内核对象

DATA 和 IDATA 是 8051 的不同寻址方式，并不意味着它们必然对应两块独立的物理 RAM。高地址 IDATA 是否可用，取决于具体芯片的内部 RAM 实现。

XDATA 全局池的规划容量为 512 字节。实际已分配对象的大小、地址间隙和剩余空间，应分别统计，不能仅凭全局池的结束地址推断已使用容量。

16.4 内部 RAM 布局

16.4.1 固定地址内核变量

起始地址	结束地址	大小	变量
`0x15`	`0x15`	1 字节	`OS_IRQ_NEST_TEMP`
`0x30`	`0x30`	1 字节	`OS_INTERRUPT_PROTECT`
`0x31`	`0x31`	1 字节	`OS_R0_PROTECT`
`0x32`	`0x32`	1 字节	`os_sp`
`0x33`	`0x33`	1 字节	`OS_HRTOS`
`0x34`	`0x34`	1 字节	`OS_HRTOS_2`

以上地址由源码中的 `_at_` 声明直接指定。

`OS_INTERRUPT_PROTECT` 保存中断编号相关信息；`OS_R0_PROTECT` 用于 R0 保护；`os_sp`、`OS_HRTOS` 和 `OS_HRTOS_2` 用于各自实现路径中的内核临时状态。

`OS_IRQ_NEST_TEMP` 位于 `0x15`，不属于 `0x30-0x34` 连续变量区。

16.4.2 系统栈、中断栈与用户任务栈

当前源码定义如下：

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

```
unsigned char data OS_HRTOS_DUIZHAN_INT[8] _at_ 0x3A;

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

在 `OS_SP_KER = 5` 的配置下，内存范围如下。

栈区域	起始地址	结束地址	容量
系统栈	`0x35`	`0x39`	5 字节
中断栈	`0x3A`	`0x41`	8 字节
用户任务栈	`0x42`	`0x7F`	62 字节

用户任务栈覆盖十进制地址 `66-127`，共 62 字节。`OS_USER_RAM_EXIT = 128` 是不包含在有效范围内的结束边界。

按照每个栈块 2 字节计算，用户任务栈可划分为 31 个栈块。实际可分配给单个任务的栈空间由任务创建参数和栈分配算法决定。

注意：源码中的注释与数组实际长度存在不一致的地方。本文以 `_at_` 地址和数组长度为准，而不是以旧注释中的起止地址为准。

16.4.3 IDATA 中断保护及调度状态

当前源码将以下变量固定在 IDATA 高地址区域。

起始地址	结束地址	大小	变量
`0xDB`	`0xE0`	6 字节	`OS_INT_BAO`
`0xE1`	`0xE1`	1 字节	`OS_T0_TL0`
`0xE2`	`0xE2`	1 字节	`OS_T0_TH0`
`0xE3`	`0xE3`	1 字节	`OS_CURRENT_TASK`
`0xE4`	`0xE4`	1 字节	`OS_PREV_TASK`
`0xE5`	`0xE5`	1 字节	`OS_DISPATCH_ID`
`0xE6`	`0xE6`	1 字节	`OS_TIME_XY`
`0xE7`	`0xE7`	1 字节	`os_critical_nesting`

`OS_INT_BAO` 占用 `0xDB-0xE0` 六个连续字节，后续变量从 `0xE1` 开始，整个固定地址区域共占用 13 字节。

各变量用途如下：

- `OS_INT_BAO`：中断现场保护数据。
- `OS_T0_TL0`、`OS_T0_TH0`：Timer0 相关保存数据。
- `OS_CURRENT_TASK`：当前任务 ID。
- `OS_PREV_TASK`：上一个任务 ID。

- `OS_DISPATCH_ID`：调度触发源。
- `OS_TIME_XY`：时间片控制状态。
- `os_critical_nesting`：临界区嵌套计数。

寄存器保存顺序及恢复行为由中断入口、汇编上下文保存代码和恢复流程决定。

16.4.4 BIT 类型内核标志

源码声明了以下 `volatile bit` 变量：

```
volatile bit OS_KUAI_PROCESS_A;
volatile bit OS_KUAI_PROCESS_B;
volatile bit OS_KUAI_PROCESS_A_DEL;
volatile bit OS_KUAI_PROCESS_B_DEL;

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;
```

这些变量用于高速任务生命周期、调度请求、中断状态、内核模式及配置保护等功能。

它们没有在当前代码中通过独立的 `_at_` 地址逐一指定位置。因此，不能根据声明顺序为它们编造具体 RAM 地址，也不能直接把变量数量换算成某个连续 DATA 地址范围。

实际位存储分配应以 Keil C51 编译结果和链接映射为准。

16.5 XDATA 全局内存池

16.5.1 基本配置

```
#define OS_SYS_GLOBAL_BASE  0x0200
#define OS_SYS_GLOBAL_SIZE  512
#define OS_SYS_GLOBAL_END   (OS_SYS_GLOBAL_BASE + OS_SYS_GLOBAL_SIZE - 1)
```

项目	数值
基地址	`0x0200`
总容量	512 字节
最后一个有效地址	`0x03FF`
完整地址范围	`0x0200-0x03FF`

每个 XDATA 对象的起始地址按下式计算：

```
对象起始地址 = OS_SYS_GLOBAL_BASE + 对象偏移
```

全局池的容量为 512 字节，并不意味着其中每个地址都已经被对象占用。

16.5.2 实际固定地址对象映射

以下表格按当前源码中的偏移宏、固定地址声明和对象长度整理。表内结束地址均包含在对象范围内。

起始地址	结束地址	大小	对象
0x0200	0x020F	16 字节	OS_PROCESS_OK
0x0210	0x021F	16 字节	OS_JINCHENG_SP
0x0220	0x0221	2 字节	OS_KUAI_SP
0x0222	0x022B	10 字节	OS_FAST_CONTEXT
0x022C	0x02FB	208 字节	OS_TASK_CONTEXT
0x02FC	0x02FF	4 字节	OS_MEMORY
0x0300	0x030F	16 字节	OS_JINCHENG_JILU_SP
0x0310	0x0310	1 字节	OS_TIME_ONCE
0x0311	0x0314	4 字节	OS_WAIT_DI2
0x0315	0x0316	2 字节	OS_INTERRUPT_ADDR
0x0317	0x0318	2 字节	OS_INTERRUPT_ADDR2
0x0319	0x031A	2 字节	OS_TICK_COUNT_L
0x031B	0x031C	2 字节	OS_TICK_COUNT_H
0x031F	0x0320	2 字节	OS_SP_KUAI_BEI
0x0321	0x0322	2 字节	OS_SP_KUAI_BEI2
0x0324	0x0324	1 字节	OS_TIME_ONCE_BACKUP
0x0325	0x03A4	128 字节	OS_TASK
0x03A5	0x03C8	36 字节	OS_RES
0x03C9	0x03DD	21 字节	OS_MSGQ

这张表仅包含源码中实际声明的固定地址对象。`OS_EVENT_BIT`、`OS_STACK_WARN` 和 `OS_TASK_EVENT` 没有列为已分配对象，原因见第 6 节。

16.5.3 实际地址间隙

当前对象映射中存在以下未分配地址：

起始地址	结束地址	容量	说明
0x031D	0x031E	2 字节	没有对应的固定地址

起始地址	结束地址	容量	说明
			对象声明
0x0323	0x0323	1 字节	没有对应的固定地址对象声明

因此，实际对象从 0x0200 连续布局至 0x031C，随后在 0x031D-0x031E 留有间隙；0x031F-0x0322 为高速任务栈指针备份；0x0323 为单字节间隙；0x0324 为时间备份，之后是任务控制块、资源对象及消息队列对象。

16.6 未实际分配的事件及栈警告区域

当前源码定义了以下偏移宏：

```
#define OS_OFF_EVENT_BIT   0x011D
#define OS_OFF_STACK_WARN  0x011E
#define OS_OFF_TASK_EVENT  0x0123
```

这些宏本身只定义地址偏移，不会自动分配内存。

16.6.1 OS_EVENT_BIT

其规划地址为：

```
基地址：0x0200
偏移量：0x011D
计算地址：0x031D
```

当前所提供的源码没有出现对应的固定地址对象声明。因此，0x031D 不能直接标记为已经分配给 OS_EVENT_BIT 的地址。

16.6.2 OS_STACK_WARN

其规划地址为：

```
基地址：0x0200
偏移量：0x011E
计算地址：0x031E
```

当前所提供的源码没有出现对应的固定地址对象声明。因此，0x031E 当前属于未分配地址。

16.6.3 OS_TASK_EVENT

其规划起始地址为：

```
基地址：0x0200
偏移量：0x0123
计算地址：0x0323
```

源码没有提供对应的固定地址数组声明，但后续对象已经占用：

对象	实际地址范围
`OS_TIME_ONCE_BACKUP`	`0x0324-0x0324`
`OS_TASK`	`0x0325-0x03A4`

如果在 `0x0323` 声明 18 字节数组，其地址范围将是 `0x0323-0x0334`，会覆盖 `OS_TIME_ONCE_BACKUP`，并与 `OS_TASK` 的 `0x0325-0x0334` 范围重叠。

因此，**当前布局不能将 `OS_TASK_EVENT` 作为从 `0x0323` 开始的 18 字节独立对象使用。** 在重新确定地址并同步修改所有引用之前，应将该区域视为未分配的规划区域。

16.7 任务状态与上下文

16.7.1 任务状态数组

```
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);
```

当前配置：

```
#define OS_PROCESS_MAX   16
#define OS_FAST_TASK_MAX 2
```

对象	容量	用途
`OS_PROCESS_OK`	16 字节	普通任务启动状态及相关编码
`OS_JINCHENG_SP`	16 字节	普通任务栈指针记录
`OS_KUAI_SP`	2 字节	高速任务栈指针记录

`OS_PROCESS_OK` 不应简单理解为布尔数组。其位字段及编码方式必须与任务创建、等待、唤醒和调度实现保持一致。

16.7.2 任务上下文保存区域

```
#define OS_OFF_FAST_CONTEXT 0x0022
#define OS_OFF_TASK_CONTEXT 0x002C

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);
```

当前配置：

```
#define OS_KUAI_SHENDU      5
#define OS_JINCHENG_SHENDU 13
```

高速任务上下文：

```
2 × 5 = 10 字节
```

普通任务上下文：

```
16 × 13 = 208 字节
```

两个上下文数组合计 218 字节。

上下文数组保存任务切换所需的运行现场数据；用户任务栈则提供任务执行期间的栈空间。两者用途不同，不能将上下文容量当作任务栈容量。

16.7.3 栈位图与栈指针备份

```
#define OS_OFF_MEMORY           0x00FC
#define OS_OFF_JINCHENG_JILU_SP 0x0100
```

对象	地址范围	容量	用途
`OS_MEMORY`	`0x02FC-0x02FF`	4 字节	用户栈分配位图
`OS_JINCHENG_JILU_SP`	`0x0300-0x030F`	16 字节	普通任务 SP 备份
`OS_SP_KUAI_BEI`	`0x031F-0x0320`	2 字节	高速任务 SP 起始地址记录
`OS_SP_KUAI_BEI2`	`0x0321-0x0322`	2 字节	高速任务 SP 大小记录

`OS_MEMORY` 共 32 位。按照每个栈块 2 字节计算，用户任务栈需要 31 个栈块，对应 31 个有效位；其余位的具体使用应以分配实现为准。

栈分配和释放必须采用一致的栈块索引规则，确保位图记录与实际栈地址对应。

16.8 时间、中断及调度数据

16.8.1 时间和高速任务延时

对象	地址范围	容量	用途
`OS_TIME_ONCE`	`0x0310-0x0310`	1 字节	时间片相关数据
`OS_WAIT_DI2`	`0x0311-0x0314`	4 字节	两个高速任务的延时数据
`OS_TIME_ONCE_BACKUP`	`0x0324-0x0324`	1 字节	配合 MyOS 保存时间片相关状态

`OS_WAIT_DI2` 的类型为 `unsigned int[2]`。在每个 `unsigned int` 占用 2 字节的当前布局下，总计 4 字节。

16.8.2 中断地址记录

对象	地址范围	容量
OS_INTERRUPT_ADDR	0x0315-0x0316	2 字节
OS_INTERRUPT_ADDR2	0x0317-0x0318	2 字节

两个对象分别提供中断处理相关的地址记录空间。具体编码方式及其在中断入口和恢复流程中的用途，应以实际中断处理代码为准。

16.8.3 系统时钟计数

对象	地址范围	容量
OS_TICK_COUNT_L	0x0319-0x031A	2 字节
OS_TICK_COUNT_H	0x031B-0x031C	2 字节

每个计数变量为 `unsigned int`，两个变量合计 4 字节。变量名称表示高、低计数部分，但其组合顺序、进位处理和读取一致性应由计时实现确定。

8051 为 8 位处理器，多字节计数器的并发读写需要考虑中断更新造成的不一致风险。

16.9 任务控制块

16.9.1 固定地址与结构定义

```
volatile OS_TCB xdata OS_TASK[OS_PROCESS_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + 0x0125);
```

对应结构体：

```
typedef struct
{
    u8 base_prio;
    u8 cur_prio;
    u8 state;

    u8 wait_type;
    u8 wait_flag;
    u8 wait_obj;

    u16 wait_tick;
} OS_TCB;
```

当前布局要求每个 `OS_TCB` 占用 8 字节。16 个任务控制块共占用 128 字节。

项目	数值
起始地址	0x0325
结束地址	0x03A4
对象数量	16

项目	数值
单个对象大小	8 字节
总容量	128 字节

任务控制块保存基本优先级、当前优先级、任务状态及统一等待信息。

等待字段包括等待类型、等待结果、等待对象 ID 和等待 tick。任务上下文、栈指针及用户栈空间记录保存在其他区域。

16.10 统一资源对象

16.10.1 固定地址与结构定义

```
volatile OS_RESOURCE xdata OS_RES[OS_RESOURCE_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + 0x01A5);
```

当前配置：

```
#define OS_RESOURCE_MAX 6
```

结构体：

```
typedef struct
{
    u8 value;
    u8 owner;

    u8 wait_cnt;
    u16 wait_mask;

    u8 pending_signal;
} OS_RESOURCE;
```

在当前 C51 编译布局中，每个资源对象按 6 字节计算，6 个对象合计 36 字节。

项目	数值
起始地址	`0x03A5`
结束地址	`0x03C8`
对象数量	6
单个对象大小	6 字节
总容量	36 字节

字段用途：

- `value`：资源状态或计数。
- `owner`：需要所有者信息时记录所有者。
- `wait_cnt`：等待任务数量。
- `wait_mask`：等待任务位图。
- `pending_signal`：待处理信号计数。

`wait_mask` 为 16 位，可以对应普通任务 ID `0-15`。等待数量与等待位图必须在任务等待、唤醒及清理过程中保持一致。

16.11 消息队列对象

16.11.1 固定地址与结构定义

```
volatile os_msgq_t xdata OS_MSGQ[OS_MSGQ_MAX]
    _at_ (OS_SYS_GLOBAL_BASE + 0x01C9);
```

当前配置：

```
#define OS_MSGQ_MAX 3
```

结构体：

```
typedef struct
{
    u8 *buf;
    u8 _size;
    u8 head;
    u8 tail;
    u8 count;
} os_msgq_t;
```

在 XDATA 指针占用 3 字节、其余字段各占 1 字节且没有额外填充的当前布局下，每个对象占用 7 字节，3 个对象合计 21 字节。

项目	数值
起始地址	`0x03C9`
结束地址	`0x03DD`
对象数量	3
单个对象大小	7 字节
总容量	21 字节

消息队列对象保存缓冲区指针、容量、读写位置和当前消息数量。实际消息缓冲区由 `buf` 指向，其内存不包含在这 21 字节的控制对象容量中。

16.12 XDATA 内存容量统计

16.12.1 实际对象容量

按当前声明的数组长度和结构体大小统计：

对象组	占用字节
任务状态及栈指针记录	34
高速任务与普通任务上下文	218
用户栈位图	4

对象组	占用字节
时间、延时、中断及 tick 数据	11
高速任务栈指针备份	4
时间备份	1
普通任务控制块	128
统一资源对象	36
消息队列控制对象	21
实际对象总计	**457**

上述分组中的 11 字节由 `OS_TIME_ONCE` 1 字节、`OS_WAIT_DI2` 4 字节、两个中断地址数组 4 字节及两个 tick 计数变量 4 字节构成，合计应为 13 字节；因此，正式统计应采用逐对象求和结果，而不能直接沿用未经复核的分组数字。

按全部实际对象逐项求和，正确的已分配容量为 **475 字节**。

16.12.2 地址范围与剩余空间

项目	数值
全局池总容量	512 字节
实际对象容量	475 字节
未分配容量	37 字节
最后一个已分配对象结束地址	`0x03DD`
末尾连续空闲区	`0x03DE-0x03FF`，34 字节
内部地址间隙	3 字节

内部地址间隙由 `0x031D-0x031E` 的 2 字节和 `0x0323` 的 1 字节构成。

因此，当前全局池中共有 475 字节被实际对象占用，37 字节未分配。其中 34 字节位于池末尾，另有 3 字节位于内部地址间隙。

以上统计不包含应用程序独立提供的消息缓冲区、其他模块自行分配的 XDATA 数据，以及编译器未按固定地址放置的变量。

16.13 设计约束

16.13.1 固定地址约束

- 每个固定地址对象必须与其偏移宏及数组长度一致。
- 不得将只有偏移宏、没有实际声明的对象计入已分配容量。
- 不得让数组范围与后续固定地址对象重叠。

- 修改结构体字段、数组长度或配置参数后，必须重新计算全部后续地址。
- 最终布局应通过 Keil C51 编译结果和链接映射验证。

16.13.2 栈管理约束

- 系统栈、中断栈和用户任务栈必须保持独立。
- 用户栈有效范围为 `0x42-0x7F`。
- 用户栈按 2 字节一块计算，共有 31 个栈块。
- 栈分配与释放必须采用一致的位图索引规则。
- 实际栈使用情况还应结合任务创建参数和运行时栈深度分析。

16.13.3 任务及内核对象约束

- 普通任务状态、任务控制块、任务上下文和栈记录承担不同职责。
- 高速任务与普通任务使用不同的上下文和栈管理数据。
- 资源等待位图、等待计数和任务控制块等待字段必须保持一致。
- 消息队列控制对象与消息数据缓冲区应分别统计。
- 任务删除时必须同步清理任务状态、栈分配记录及等待对象关联信息。

16.13.4 芯片与编译器约束

移植到其他 8051 器件时，应确认：

- 内部 RAM 容量及高地址 IDATA 的实际可用性。
- XDATA 容量及固定地址的可访问性。
- C51 存储类型、指针宽度和结构体实际大小。
- 中断入口、寄存器保护及上下文恢复代码。
- 固定地址对象是否与最终链接映射一致。

16.14 模块总结

HRTOS 当前内存布局采用固定地址管理内核变量、系统栈、中断栈、用户任务栈、任务上下文及主要内核对象。

内部 RAM 中，系统栈占用 `0x35-0x39`，中断栈占用 `0x3A-0x41`，用户任务栈占用 `0x42-0x7F`。IDATA 中断保护与调度状态占用 `0xDB-0xE7`。

XDATA 全局池的规划范围为 `0x0200-0x03FF`，总容量 512 字节。按当前源码实际声明的固定地址对象统计，已分配容量为 475 字节，未分配容量为 37 字节。

本布局以当前源码为依据。任何涉及对象地址、对象大小或内存容量的修改，都必须同步检查固定地址声明、偏移宏、结构体定义及最终链接映射，确保实际内存分配不存在冲突。

Call Flow

HRTOS 调用流程

17.1 文档概述

HRTOS 内核通过任务注册、任务状态管理、等待与唤醒、调度请求以及任务资源回收等机制，组织系统的运行过程。

本文围绕内核关键调用链，介绍系统初始化、任务创建、任务清理、任务等待、任务唤醒和调度请求的执行流程，并说明普通任务与高速任务在等待处理和资源管理方面的区别。

本文涉及的核心机制包括：

- **系统初始化与配置管理**：初始化任务状态、等待信息及系统资源，并在用户初始化完成后锁定系统配置。
- **任务注册与运行栈管理**：为标准任务和高速任务分配运行栈，并初始化入口地址及相关上下文信息。
- **任务等待与唤醒**：通过任务控制信息记录等待条件、等待对象、超时计数及唤醒结果。
- **调度请求**：通过调度请求标志和 Timer0 中断触发任务重新调度。
- **任务清理与资源回收**：释放任务占用的运行栈空间，清除等待关系和互斥资源所有权，并重置任务状态。
- **Idle Hook**：在空闲任务循环中提供用户扩展入口。

17.2 核心文件与职责

文件	主要职责
`idle_task.c`	初始化运行环境、调用用户初始化入口、锁定配置、执行空闲循环及任务清理
`task_create.c`	注册标准任务、高速任务及中断处理函数，分配运行栈空间
`task_cleanup.c`	回收任务运行栈空间、清除资源等待关系、释放互斥资源所有权
`wait.c`	设置任务等待信息、建立资源等待关系、阻塞任务并触发调度
`wake_task.c`	清除等待信息、设置唤醒结果、恢复任务

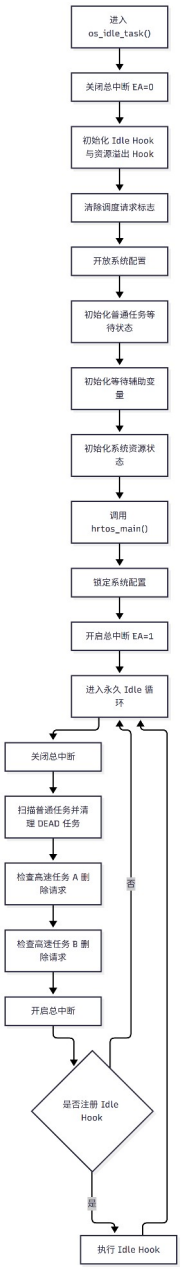
文件	主要职责
	就绪状态并触发调度
`schedule_request.c`	设置调度请求标志并触发 Timer0 中断

17.3 系统初始化与运行入口

17.3.1 初始化流程

HRTOS 通过 `os_idle_task()` 初始化内核运行环境。初始化阶段首先关闭总中断，随后重置空闲钩子、资源溢出钩子、调度请求标志及配置锁，并初始化普通任务的等待状态和系统资源状态。

初始化完成后，系统调用 `hrtos_main()` 执行用户侧的系统配置和应用初始化。该阶段结束后，内核设置配置锁，禁止继续注册配置项，随后开启总中断并进入永久空闲循环。



17.3.2 初始化阶段的状态设置

`os_idle_task()` 对以下状态进行初始化：

1. 将 `g_idle_hook` 和 `g_slab_overflow_hook` 置空。
2. 将 `OS_SCHED_REASON` 清零，清除已有调度请求标志。
3. 将 `OS_CFG_LOCK` 置为 `0`，开放系统配置。
4. 将普通任务的 `wait_type` 初始化为 `WAIT_NONE`，`state` 初始化为 `READY`，`wait_tick` 清零，`wait_obj` 设置为 `OS_INVALID_ID`。
5. 清零 `OS_WAIT_DI2` 中两个高速任务的延时计数。
6. 将各系统资源的 `value`、`wait_cnt`、`wait_mask` 和 `pending_signal` 清零，并将 `owner` 设置为 `OS_INVALID_ID`。

这里的普通任务初始化循环覆盖任务编号 `0` 至 `OS_PROCESS_MAX - 1`。两个高速任务的运行状态和运行栈信息由其独立的注册及管理路径处理。

17.3.3 配置阶段与运行阶段

HRTOS 通过 `OS_CFG_LOCK` 区分系统配置阶段和正常运行阶段。

阶段	`OS_CFG_LOCK`	运行行为
用户初始化阶段	`0`	允许执行任务注册等配置操作
正常运行阶段	`1`	禁止通过 `os_task_create()` 继续注册任务或中断处理函数

`hrtos_main()` 是系统配置与应用初始化的入口。该函数返回后，内核锁定配置、开启总中断，并开始执行 Idle 循环。

17.3.4 Idle 循环与 Idle Hook

Idle 循环主要承担两项职责：

- 检查并处理任务删除请求。
- 调用用户注册的 `g_idle_hook`。

每轮循环首先关闭总中断，执行任务清理检查，然后重新开启总中断。只有在 `g_idle_hook` 非空时，才调用该钩子函数。

Idle Hook 在开启总中断之后执行，因此它运行期间系统中断处于开放状态。用户实现该钩子时，应合理控制执行时间，避免长时间占用空闲任务的执行机会。

17.4 任务创建流程

17.4.1 创建入口

`os_task_create(addr, id, pr, sd)` 用于注册标准任务、高速任务，以及设置中断处理函数和中断嵌套处理函数的入口地址。

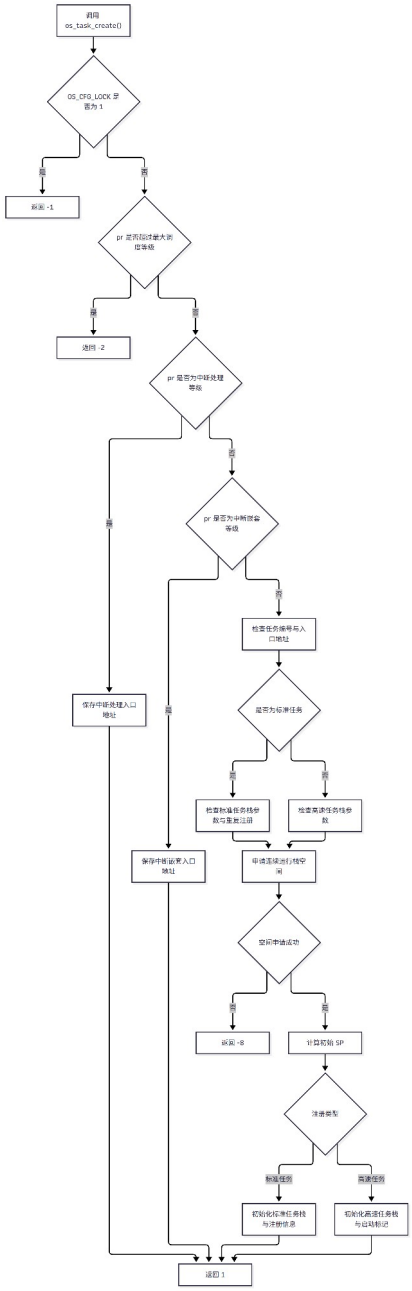
参数含义如下：

参数	含义
`addr`	任务或处理函数入口地址
`id`	任务编号
`pr`	优先级及注册类型选择参数
`sd`	运行栈空间参数

函数通过 `pr` 判断注册类型。标准任务和高速任务需要分配运行栈空间；中断处理函数和中断嵌套处理函数则分别保存入口地址。

17.4.2 注册类型选择

函数首先检查配置锁和优先级参数，再根据 `pr` 选择对应的注册路径。



中断处理函数和中断嵌套处理函数的分支在任务编号及入口地址检查之前执行。标准任务和高速任务则继续进行任务编号、入口地址及栈参数检查。

17.4.3 运行栈空间申请

标准任务与高速任务共用基于 `OS_MEMORY` 位图的运行栈空间申请机制。

申请过程如下：

1. 将参数 `sd` 加一，得到本次申请所需的空间块数。
2. 从用户运行栈区域的起始位置开始扫描 `OS_MEMORY`。
3. 根据位图中的对应位判断当前空间块是否已被占用。
4. 连续遇到空闲块时，累积连续空闲块数量。
5. 当连续空闲块数量满足申请要求时，将对应位图全部标记为占用。
6. 根据分配位置计算初始栈指针 `SP`。
7. 初始化对应任务类型的运行栈记录及入口地址。

如果扫描过程中发现可用空间不足，函数返回 `-8`。

位图以每字节八个标志位管理空间块。分配时使用 `0x80 >> OS_R0_PROTECT` 定位相应位，并将成功分配的连续空间标记为已占用。

17.4.4 标准任务注册

标准任务的注册路径要求：

- `sd` 不大于 `6`。
- `OS_PROCESS_OK[id]` 为 `0`，即该编号未处于已注册状态。
- 运行栈空间申请成功。

注册成功后，函数更新以下信息：

- `OS_JINCHENG_SP[id]`：记录任务栈指针相关信息。
- `OS_JINCHENG_JILU_SP[id]`：记录初始栈指针。
- `OS_PROCESS_OK[id]`：记录优先级、启动标记和栈空间大小等信息。

随后，函数暂存当前系统栈指针，将 `SP` 切换到新任务的栈位置，把入口地址的低字节和高字节依次压入该任务的栈中，最后恢复原来的 `SP`。

这一步为任务后续的运行入口建立初始栈内容。

17.4.5 高速任务注册

高速任务的注册路径要求 `sd` 不大于 `25`。运行栈申请成功后，函数通过 `id %= OS_FAST_TASK_MAX` 将编号映射到高速任务槽位。

随后更新高速任务的栈指针、初始栈指针和栈空间记录，并根据任务槽位设置相应的启动标记：

- 槽位 `0` 对应 `OS_KUAI_PROCESS_A`。

- 其他槽位对应 `OS_KUAI_PROCESS_B`。

同时，函数初始化 `OS_FAST_CONTEXT` 中的相关寄存器组信息，并将任务入口地址压入高速任务的运行栈。

17.4.6 中断处理函数注册

当 `pr` 等于 `OS_SCHED_LEVEL_MAX - 1` 时，函数将入口地址的低字节和高字节分别保存到 `OS_INTERRUPT_ADDR[0]` 和 `OS_INTERRUPT_ADDR[1]`。

当 `pr` 等于 `OS_SCHED_LEVEL_MAX` 时，函数将入口地址保存到 `OS_INTERRUPT_ADDR2` 对应的两个字节中。

这两条路径保存中断处理入口地址，不执行标准任务或高速任务的运行栈分配流程。

17.4.7 创建操作的返回值

返回值	含义
`1`	注册成功
`-1`	系统配置已锁定
`-2`	优先级参数超过最大调度等级
`-3`	任务编号无效
`-4`	任务入口地址为空
`-5`	标准任务栈参数无效
`-6`	标准任务重复注册
`-7`	高速任务栈参数无效
`-8`	用户运行栈空间不足

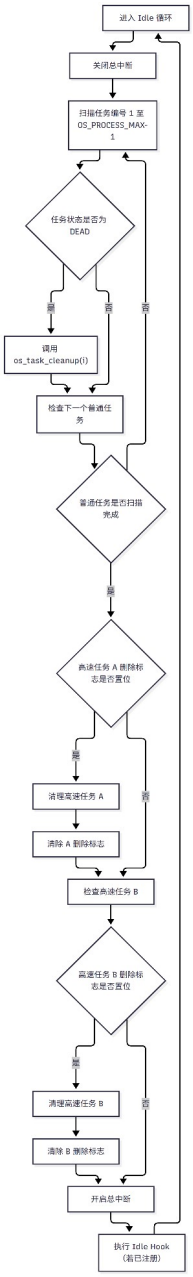
17.5 任务删除与清理流程

17.5.1 删除请求与实际清理

HRTOS 将任务删除后的资源回收放在 `os_task_cleanup()` 中执行。Idle 循环负责检查普通任务的 `DEAD` 状态，以及两个高速任务各自的删除请求标志。

普通任务与高速任务采用不同的删除请求表示方式：

- 普通任务通过 `OS_TASK[i].state == DEAD` 进入清理路径。
- 高速任务 A 通过 `OS_KUAI_PROCESS_A_DEL` 标记删除请求。
- 高速任务 B 通过 `OS_KUAI_PROCESS_B_DEL` 标记删除请求。



17.5.2 清理入口与合法性检查

`os_task_cleanup(id)` 根据任务编号选择相应的资源回收路径。

函数首先拒绝以下请求：

- `id` 大于 `OS_PROCESS_MAX + 1`。
- `id` 等于 `0`。

对于标准任务，函数要求 `OS_JINCHENG_JILU_SP[id]` 非零；对于两个高速任务，分别要求对应的 `OS_SP_KUAI_BEI` 记录非零。若相应运行栈记录不存在，则返回 `-1`。

合法任务进入清理流程后，函数暂存当前 `OS_HRTOS` 值，待资源回收完成后恢复该值。

17.5.3 运行栈与上下文清理

标准任务与高速任务分别使用独立的栈记录和上下文数组。

标准任务清理过程包括：

1. 调整栈记录中的兼容性计数。
2. 根据栈记录和用户 RAM 起始地址计算待释放空间范围。
3. 清除任务注册记录和初始栈指针记录。
4. 将 `OS_TASK_CONTEXT[id]` 对应的上下文内容清零。

高速任务清理过程则分别处理高速任务 A 或 B：

1. 调整对应高速任务的栈空间记录。
2. 根据初始栈指针计算待释放空间范围。
3. 清除对应高速任务的启动标记。
4. 清除高速任务栈指针及栈空间记录。
5. 将对应的 `OS_FAST_CONTEXT` 内容清零。

完成对应类型的记录清理后，函数根据计算出的空间范围遍历 `OS_MEMORY`，清除对应位图中的占用标记，从而释放运行栈空间。

17.5.4 等待关系与资源所有权清理

任务清理不仅回收运行栈，还要移除该任务在系统资源中的关联信息。

函数遍历全部系统资源，对每个资源执行两项检查。

第一项：清除等待位图。

如果资源的 `wait_mask` 中包含当前任务的位，则清除该位；如果 `wait_cnt` 非零，同时将等待计数减一。

第二项：释放互斥资源所有权。

如果资源的 `owner` 等于待清理任务编号，则将 `owner` 设置为 `OS_INVALID_ID`。

这一处理确保任务退出后，不再以等待者或资源所有者的身份保留在资源管理记录中。

17.5.5 清理任务等待状态

资源关系处理完成后，函数统一重置任务控制块中的等待字段：

字段	清理后的值
`wait_type`	`WAIT_NONE`
`wait_flag`	`WAIT_TIMEOUT`
`wait_obj`	`OS_INVALID_ID`
`wait_tick`	`0`

字段	清理后的值
`state`	`READY`

对于高速任务，函数还会清零对应的 `OS_WAIT_DI2` 延时计数，防止遗留的延时状态再次激活该任务。

最后，函数恢复进入清理流程前保存的 `OS_HRTOS`，并返回 `1` 表示清理成功。

需要注意，`os_task_cleanup()` 完成的是任务资源与运行状态的重置；函数最终将任务状态设置为 `READY`，而不是 `DEAD`。

17.6 任务等待流程

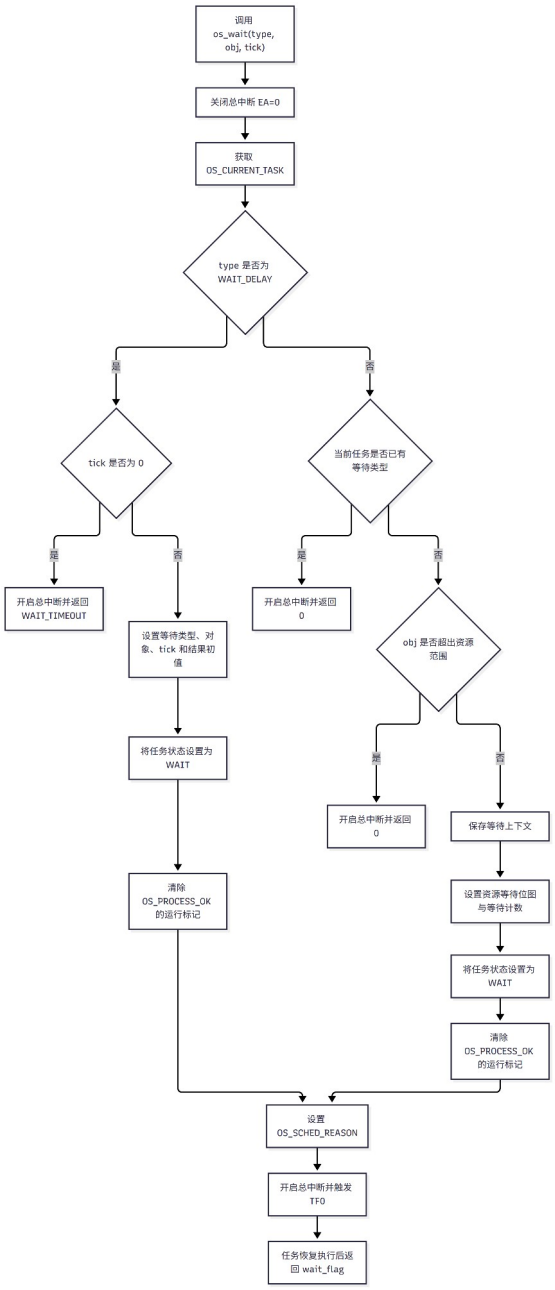
17.6.1 等待机制概述

`os_wait(type, obj, tick)` 是 HRTOS 统一的任务等待入口。函数通过等待类型、等待对象和等待时长描述任务的等待条件，并将相应信息保存到任务控制块中。

对于普通任务，等待操作主要包括：

1. 关闭总中断，进入临界区。
2. 获取当前任务编号。
3. 检查等待参数及当前等待状态。
4. 保存等待类型、等待对象、等待时长和等待结果初值。
5. 必要时将任务加入资源等待位图。
6. 将任务状态设置为 `WAIT`，清除其运行资格标记。
7. 设置调度请求标志，开启总中断并触发 Timer0 中断。
8. 在任务后续恢复执行时，返回任务控制块中的等待结果。

17.6.2 普通任务等待流程



17.6.3 等待状态与资源等待位图

普通任务进入等待状态时，`os_wait()` 将以下字段写入 `OS_TASK[tid]`：

- `wait_type`：等待类型。
- `wait_obj`：等待对象编号。
- `wait_tick`：等待超时计数。
- `wait_flag`：等待结果，进入等待时初始化为 `0`。
- `state`：任务状态设置为 `WAIT`。

对于非 `WAIT_DELAY` 类型的等待，如果 `obj` 不等于 `OS_INVALID_ID`，函数会在对应资源的 `wait_mask` 中设置当前任务的位，并将 `wait_cnt` 加一。

纯时间等待不需要进入资源等待位图。资源等待则通过对象编号建立任务与系统资源之间的关联关系。

17.6.4 等待期间的调度触发

任务进入等待状态后，函数通过以下操作请求任务重新调度：

```
OS_SCHED_REASON = 1;
EA = 1;
TF0 = 1;
```

`OS_SCHED_REASON` 用于记录调度请求，`TF0` 用于触发 Timer0 中断。`os_wait()` 本身不直接调用调度器。

等待函数在设置等待状态后，继续返回 `OS_TASK[tid].wait_flag`。任务实际恢复执行时，会从该调用点继续执行并取得等待结果。

17.6.5 等待结果

在普通任务的等待路径中，等待结果保存在 `wait_flag` 中。

返回值	含义
`WAIT_TIMEOUT`	等待超时
`WAIT_SIGNAL`	通过资源或事件正常唤醒
`0`	等待参数无效，或当前任务已经处于等待状态

对于 `WAIT_DELAY` 且 `tick == 0` 的调用，函数直接返回 `WAIT_TIMEOUT`，不建立等待状态。

17.6.6 高速任务的延时处理

当等待类型为 `WAIT_DELAY` 时，函数会单独判断当前任务是否属于高速任务。高速任务不使用普通任务的统一等待字段和资源等待位图来处理延时，而是使用独立的启动标记和 `OS_WAIT_DI2` 计数。

- 高速任务 A：清除 `OS_KUAI_PROCESS_A`，将延时时长写入 `OS_WAIT_DI2[0]`。
- 高速任务 B：清除 `OS_KUAI_PROCESS_B`，将延时时长写入 `OS_WAIT_DI2[1]`。

随后设置调度请求标志、开启总中断并触发 Timer0 中断。该分支在完成上述操作后直接返回 `WAIT_TIMEOUT`，延时计数由高速任务的独立管理路径处理。

17.7 任务唤醒流程

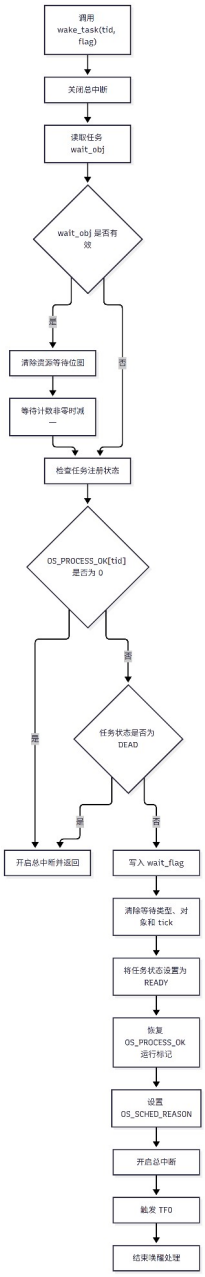
17.7.1 唤醒入口

`wake_task(tid, flag)` 用于唤醒指定的普通任务，并将唤醒原因写入任务控制块。

函数首先关闭总中断，读取任务原先关联的等待对象，并尝试清除该任务在对应资源中的等待位图及等待计数。随后检查任务是否已经删除或处于删除准备状态。

如果任务可以继续唤醒处理，函数将等待结果设置为调用者提供的 `flag`，清除等待上下文，将任务恢复为就绪状态，并请求系统重新调度。

17.7.2 唤醒流程图



17.7.3 清理等待上下文

对于允许唤醒的任务，`wake_task()` 会执行以下操作：

1. 将 `wait_flag` 设置为传入的 `flag`。
2. 将 `wait_type` 设置为 `WAIT_NONE`。
3. 将 `wait_obj` 设置为 `OS_INVALID_ID`。
4. 将 `wait_tick` 清零。
5. 将 `state` 设置为 `READY`。
6. 通过 `OS_PROCESS_OK[tid] |= 1` 恢复任务的运行资格标记。

随后，函数设置 `OS_SCHED_REASON`，开启总中断并将 `TF0` 置位，触发后续调度处理。

17.7.4 已删除任务的处理

当 `OS_PROCESS_OK[tid] == 0` 或 `OS_TASK[tid].state == DEAD` 时，函数不会继续修改任务的等待结果和就绪状态，而是开启总中断并直接返回。

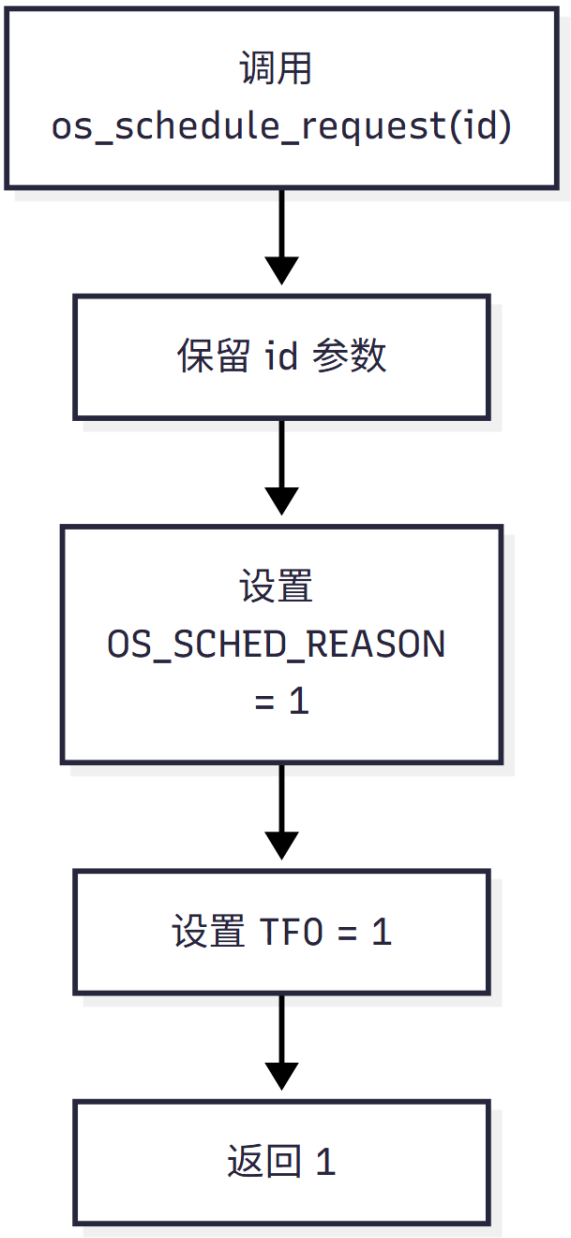
在这两种情况下，函数前面已经执行的资源等待位图检查仍然会发生。因此，唤醒入口不仅检查任务状态，还会先处理与该任务相关的等待记录。

17.8 调度请求流程

17.8.1 调度请求接口

`os_schedule_request(id)` 用于通知系统重新进行任务调度。

该函数保留 `id` 参数供后续扩展使用，当前实现不根据该参数选择调度目标。函数的主要操作是设置调度请求标志并触发 Timer0 中断。



17.8.2 调度请求与任务切换的关系

HRTOS 将调度请求与调度执行分开处理。

阶段	主要操作	对应接口或机制
发起请求	设置调度请求标志	`OS_SCHED_REASON = 1`
触发处理	置位 Timer0 中断标志	`TF0 = 1`
执行调度	在中断处理链路中进行任务调度	Timer0 中断处理路径

`os_wait()`、`wake_task()` 和 `os_schedule_request()` 均采用这一请求方式，不在函数内部直接调用调度器。这样，任务等待、任务唤醒和其他调度请求可以通过统一的中断触发机制进入后续调度流程。

17.9 任务延时、唤醒与清理之间的关系

任务延时、任务唤醒和任务清理都会影响任务的运行资格或等待状态，但它们承担不同的职责。

操作	主要状态变化	后续处理
普通任务进入等待	`state = WAIT`，清除运行资格标记	请求调度
普通任务被唤醒	`state = READY`，恢复运行资格标记	请求调度
普通任务进入删除准备状态	`state = DEAD`	Idle 循环检查并调用清理函数
任务资源清理	清除等待关系、释放运行栈、重置控制块	恢复任务记录的初始状态
高速任务延时	清除相应高速任务启动标记，记录独立延时计数	通过高速任务专用管理路径处理

普通任务的等待和唤醒主要围绕任务控制块及资源等待位图展开；高速任务的延时则使用独立的计数与启动标记。任务清理负责解除任务与资源之间的关联，避免已退出任务继续保留等待记录或互斥资源所有权。

17.10 核心执行链总结

HRTOS 的核心运行过程可以归纳为以下几条链路：

1. **系统启动链路**：`os_idle_task()` 初始化内核状态，调用 `hrtos_main()` 完成用户配置，然后锁定配置并进入 Idle 循环。
2. **任务注册链路**：`os_task_create()` 检查配置与注册参数，为标准任务或高速任务分配运行栈，初始化任务运行记录；中断处理函数注册路径则保存对应入口地址。
3. **任务等待链路**：`os_wait()` 保存等待条件，必要时建立资源等待关系，将普通任务设置为等待状态，并通过 Timer0 中断请求调度。
4. **任务唤醒链路**：`wake_task()` 清理等待关系，设置唤醒结果，恢复任务就绪状态，并通过 Timer0 中断请求调度。
5. **任务清理链路**：Idle 循环检查删除状态或删除请求，调用 `os_task_cleanup()` 释放运行栈空间、清除资源关联并重置任务记录。
6. **调度请求链路**：调度请求接口设置 `OS_SCHED_REASON` 并触发 `TF0`，由后续中断处理链路执行调度。

这些机制共同构成 HRTOS 的任务生命周期管理框架：任务先完成注册，再根据运行需求进入就绪或等待状态；等待条件满足后，任务恢复就绪资格；任务进入删除处理后，Idle 循环负责回收运行资源并清理相关状态。

Porting Layer

HRTOS 移植层

18.1 模块介绍

HRTOS 移植层用于封装与目标 MCU、硬件寄存器及编译环境相关的底层实现，减少硬件差异对上层模块的影响。

当前提供的移植代码主要面向 STC12C5A60S2 和 STC8H8U64K，已实现串口初始化及 RAM 区字符串输出。串口初始化负责配置串口工作模式、波特率发生器及相关中断控制，字符串输出接口用于向串口发送以 `\0` 结尾的字符串。

需要注意，串口移植代码只是 HRTOS 整体硬件适配的一部分。系统时基、任务调度、中断管理、上下文切换和内存布局等功能，仍需结合内核实现、目标芯片和编译环境进行配置与验证，不能仅凭串口移植代码判断其完整移植状态。

18.2 当前支持的平台

目标平台	已实现功能	主要硬件差异
STC12C5A60S2	串口初始化、RAM 区字符串输出	使用 Timer1 模式 2 产生波特率，采用 12T 配置方式下常见的定时器设置
STC8H8U64K	串口初始化、RAM 区字符串输出	通过 AUXR 配置 Timer1 为 1T 计数，并选择 Timer1 作为串口波特率发生器

具体波特率取决于系统时钟、定时器时钟、重载值及芯片的串口分频配置。移植时应以目标芯片的数据手册和实际测量结果为准。

18.3 主要接口

18.3.1 bsp_uart_init()

初始化串口及其波特率发生器。

当前两个平台均将串口配置为模式 1，即 8 位 UART、可变波特率，并通过 Timer1 提供波特率时钟。两个平台的定时器配置存在差异，不能直接互换初始化代码。

STC12C5A60S2 的主要配置：

- 将 Timer1 设置为模式 2，即 8 位自动重装模式。
- 将 `TH1` 和 `TL1` 设置为 `0xFD`。
- 禁止 Timer1 中断并启动 Timer1。
- 设置 `EA = 1`、`ES = 1`。

STC8H8U64K 的主要配置：

- 通过 `AUXR` 将 Timer1 配置为 1T 计数模式。
- 通过 `AUXR` 选择串口 1 使用 Timer1 作为波特率发生器。
- 将 `TH1` 设置为 `0xFE`、`TL1` 设置为 `0xE0`。
- 禁止 Timer1 中断并启动 Timer1。
- 设置 `ES = 0`。

STC8H8U64K 的代码没有显式设置 `EA`，也没有在初始化函数中配置 `TMOD` 的 Timer1 工作模式。因此，相关寄存器的最终状态还可能受启动代码及此前的硬件初始化影响。若后续要将该实现作为独立、可复用的 BSP，建议结合芯片手册确认这些配置是否完整。

18.3.2 bsp_uart_send_str_ram(const char *s)

发送位于 RAM 区的字符串，直到遇到字符串结束符 `\0`。

接口实现采用轮询方式发送每个字节：

- 将当前字符写入 `SBUF`。
- 等待发送完成标志 `TI` 置位。
- 清除 `TI`，继续发送下一个字符。
- 遇到 `\0` 后结束发送。

该接口为阻塞式输出。发送期间，调用任务或函数需要等待串口完成当前字符串的发送，因此不适合直接用于要求严格执行时间上限的路径。尤其应避免在不允许阻塞的中断服务程序中调用。

调用方应传入有效的、以 `\0` 结尾的 RAM 字符串指针。该接口没有提供空指针检查，也没有实现超时退出机制；如果发送完成标志无法正常置位，函数可能一直等待。

18.4 串口配置差异

18.4.1 STC12C5A60S2

```
SCON = 0x50;
```

```
TMOD &= 0x0F;
TMOD |= 0x20;

TL1 = 0xFD;
TH1 = 0xFD;

ET1 = 0;
TR1 = 1;

EA = 1;
ES = 1;
```

该配置将 Timer1 设置为模式 2，并使用 `0xFD` 作为自动重装值。实际波特率仍取决于系统时钟及芯片的定时器、串口时钟配置。

18.4.2 STC8H8U64K

```
SCON = 0x50;

TMOD &= 0x0F;

AUXR |= 0x40;
AUXR &= 0xFE;

TL1 = 0xE0;
TH1 = 0xFE;

ET1 = 0;
TR1 = 1;

ES = 0;
```

该配置通过 `AUXR` 设置 Timer1 的 1T 计数方式，并选择 Timer1 作为串口 1 的波特率发生器。由于代码没有显式设置 Timer1 的 `TMOD` 模式位，实际工作模式应结合 STC8H8U64K 的寄存器定义和初始化环境确认。

两个平台的寄存器配置服务于各自的硬件环境。新增芯片时，应单独确认串口模式、波特率时钟来源、定时器工作模式、中断控制及相关寄存器的复位状态，不应只替换定时器重载值。

18.5 移植层与内核的边界

移植层负责处理目标硬件与运行环境相关的差异；内核负责任务管理、调度、等待与唤醒、IPC 等操作系统机制。

当前提供的串口代码不实现以下功能：

- Timer0 系统时基及系统 Tick 管理。
- 任务调度与上下文切换。

- 任务栈布局和寄存器现场保存。
- 内核临界区及通用中断管理。
- 内存分配、任务资源和 IPC 管理。

上述功能属于 HRTOS 整体运行机制的一部分，其实现可能涉及其他内核源文件、配置文件或特定编译器约定。移植到新平台时，应以对应版本的实际源码为依据，明确哪些功能已经实现、哪些功能需要调整。

18.6 新平台适配要点

18.6.1 硬件与编译环境

确认目标 MCU 的 8051 内核实现、系统时钟、定时器时钟方式、RAM 与 XDATA 容量，以及编译器对特殊功能寄存器和中断函数的支持情况。

18.6.2 串口适配

根据目标芯片重新确认串口工作模式、波特率发生器、定时器模式、重载值和中断控制方式，并验证实际通信波特率。

如果目标平台采用不同的寄存器布局或专用波特率发生器，应为其提供独立的硬件适配实现，避免直接复用不兼容的寄存器配置。

18.6.3 内核适配

核对系统时基、中断入口、临界区、上下文切换、任务栈与内存布局等实现。上述内容必须与实际内核代码及目标编译环境保持一致，不能仅通过串口输出正常来判定内核已经完成移植。

18.6.4 功能验证

完成硬件配置后，应分别验证串口输出、系统启动、任务调度、延时、IPC、中断响应和长时间运行稳定性。串口输出测试只能证明相应的串口路径可以工作，不能替代内核功能验证。

18.7 当前限制

- 当前展示的串口移植实现面向 STC12C5A60S2 和 STC8H8U64K。
- 串口字符串输出采用轮询方式，属于阻塞式接口。
- 当前接口仅提供 RAM 区字符串输出，没有展示代码存储区字符串输出或通用串口收发接口。
- 当前提供的代码不足以证明所有 8051 系列 MCU 均可直接移植。

- 不同芯片的时钟体系、寄存器定义、内存资源和编译器约定可能存在差异，需要分别验证。

18.8 总结

HRTOS 当前提供的移植代码主要用于 STC12C5A60S2 和 STC8H8U64K 平台的串口初始化与 RAM 区字符串输出。两个平台通过不同的 Timer1 和 AUXR 配置实现串口波特率发生，体现了硬件相关代码需要按目标芯片分别适配的特点。

进行新平台移植时，应以实际源码和芯片手册为依据，区分串口等外设适配与内核运行机制适配，并通过实际硬件测试确认功能和时序表现。保持硬件差异与内核机制之间的边界清晰，有助于后续维护、扩展和问题定位。

Design Principles

HRTOS 设计原则

19.1 模块介绍

HRTOS 面向 8051 系列微控制器设计，以轻量化内核为基础，围绕任务调度、中断管理、资源同步、栈空间管理和应用接口构建完整的实时操作系统运行环境。

与直接将通用操作系统架构移植到 8 位微控制器上的设计思路不同，HRTOS 充分考虑 8051 的寄存器组、内部 RAM、外部 RAM、指令执行特征及中断机制，将内核调度、上下文保存和内存布局作为整体进行设计。

HRTOS 的设计不仅关注内核本身的代码规模和运行开销，也重视系统接口的一致性、模块之间的协作关系，以及开发者在实际工程中的使用体验。随着版本迭代，内核机制、驱动接口、通信功能、Shell、示例程序和技术文档持续完善，逐步形成面向 8051 应用开发的完整软件体系。

本文档介绍 HRTOS 的核心设计原则，以及这些原则在内核架构、内存管理、接口设计和性能优化中的体现。

19.2 主要职责

设计原则主要用于指导以下方面：

- **内核架构**：明确任务管理、调度、中断和资源管理之间的职责边界。
- **内存管理**：根据 8051 的存储空间特点规划变量、任务栈和上下文区域。
- **实时调度**：通过优先级机制、调度请求和任务状态管理组织任务执行。
- **中断处理**：控制中断路径上的内核操作，协调中断处理与任务调度。
- **API 设计**：提供结构清晰、行为一致的内核接口。
- **性能优化**：围绕上下文切换、任务唤醒和关键执行路径持续改进。
- **工程集成**：通过驱动、通信模块、Shell 和示例降低系统接入成本。
- **持续维护**：在保持架构稳定的前提下，逐步完善功能、修复缺陷并补充验证资料。

19.3 核心设计原则

19.3.1 1. 轻量化内核与完整功能并重

设计原则

在有限的存储资源下，以紧凑的内核实现覆盖实际应用所需的任务调度、延时等待、资源同步和任务间通信等基础能力，并通过外围模块扩展系统功能。

设计体现

- 使用紧凑的数据结构管理任务及其运行状态。
- 采用固定规模的任务、资源和消息队列管理结构。
- 将内核机制与驱动、通信协议、Shell 等功能模块分离。
- 避免为通用性引入不必要的抽象层。
- 根据 8051 的资源条件控制功能实现的内存成本。

设计目标

使系统既能适应资源有限的 8051 芯片，又能为实际工程提供超出单纯任务切换的完整软件支持。内核保持精简，应用能力则通过模块化扩展逐步增强。

19.3.2 2. 面向 8051 架构进行原生优化

设计原则

以内核运行于 8051 系列处理器为前提，将处理器寄存器、存储空间和中断机制直接纳入架构设计，而不是将其他处理器上的实现方式简单照搬过来。

设计体现

- 根据 DATA、IDATA 和 XDATA 的特性安排内核变量与管理数组。
- 利用 8051 寄存器组机制支持不同任务执行路径。
- 针对普通任务与高速任务设计不同的上下文保存策略。
- 结合定时器中断和调度请求机制组织任务切换。
- 根据具体芯片的 RAM 容量、定时器资源和编译工具链调整移植配置。

设计目标

让内核的数据布局、任务切换和中断处理方式与目标处理器相匹配，在有限硬件资源内取得性能、内存占用和功能完整性之间的平衡。

19.3.3 3. 分层任务机制与差异化上下文管理

设计原则

根据任务执行需求区分不同的任务机制，并针对不同执行路径采用相应的上下文管理方式。

设计体现

- 普通任务采用标准任务上下文管理机制。
- 高速任务使用独立的任务管理记录及上下文保存区域。
- 利用专用寄存器组减少部分任务切换过程中的现场保存开销。
- 通过统一的调度请求机制衔接任务运行状态变化与调度处理。
- 根据任务类型分别管理栈指针及相关运行记录。

设计目标

不要求所有任务都承担相同的上下文保存成本，而是根据任务机制的差异进行针对性优化，使系统能够兼顾常规任务管理能力与对切换开销较敏感的执行场景。

19.3.4 4. 以优先级为核心的任务调度

设计原则

以任务优先级作为调度决策的重要依据，通过统一的调度入口协调任务状态变化和执行权转移。

设计体现

- 区分普通任务与高速任务的执行机制。
- 通过任务状态记录和任务控制块管理任务运行资格。
- 通过调度请求标志及定时器中断触发后续调度处理。
- 支持任务等待、唤醒和延时等状态转换。
- 根据配置及调度机制处理时间片相关行为。

设计目标

使任务执行顺序具有明确的调度依据，降低任务间运行关系的复杂度，并为实时应用提供可分析、可测试的任务管理基础。

任务优先级决定调度中的相对执行次序，但系统的实际响应时间仍与中断屏蔽时间、任务切换开销、临界区长度和应用程序执行行为有关。

19.3.5 5. 固定内存布局与可预测的资源占用

设计原则

优先采用固定规模的数据结构和明确的内存规划，减少运行期间的内存管理复杂度。

设计体现

- 任务控制块、资源对象和消息队列采用固定数量的管理结构。
- 内核全局数据按预设偏移组织在指定的 XDATA 区域。
- 通过位图记录任务栈空间的分配状态。
- 按任务类型划分栈指针记录和上下文保存区域。
- 通过编译期配置控制任务数量、资源数量和队列规模。

设计目标

让内核的主要内存开销能够通过配置和布局进行核算，便于评估目标芯片的资源容量，降低动态内存管理带来的不确定性。

固定规模也意味着系统容量受到编译期配置限制。任务数量、栈空间及其他内核对象的配置，需要与目标芯片的实际存储资源相匹配。

19.3.6 6. 统一的等待与资源管理模型

设计原则

通过统一的任务等待状态和资源管理结构，协调不同等待对象与任务状态之间的关系。

设计体现

- 使用统一的任务等待接口处理相关等待请求。
- 通过任务控制块记录等待类型、等待对象、等待标志及超时信息。
- 通过资源对象维护资源状态、所有者及等待任务信息。
- 使用等待位图记录普通任务与资源对象之间的等待关系。
- 通过任务唤醒流程更新等待记录和任务运行状态。

设计目标

减少不同等待机制之间重复的状态管理逻辑，使任务阻塞、资源释放和任务唤醒具有一致的处理路径，并便于对相关状态转换进行维护和测试。

19.3.7 7. 中断处理与任务调度相分离

设计原则

明确中断服务程序与任务调度之间的职责边界，避免在中断处理过程中直接执行不适合中断上下文的阻塞操作。

设计体现

- 提供专门面向中断场景的接口或处理路径。
- 中断处理期间避免调用需要等待任务运行才能完成的阻塞操作。
- 通过调度请求标志记录需要进一步处理的调度事件。
- 利用中断退出及相应的内核处理流程协调任务切换。

- 根据 8051 的中断优先级与嵌套机制管理中断执行过程。

设计目标

使中断服务程序尽量聚焦于事件响应和必要的状态更新，将任务执行权的协调交由内核调度机制处理，从而降低中断路径与任务管理逻辑之间的耦合。

19.3.8 8. 临界区与共享状态保护

设计原则

对任务和中断可能共同访问的内核状态实施必要的保护，保证关键状态更新过程的一致性。

设计体现

- 使用临界区接口保护需要原子化处理的操作。
- 通过嵌套计数或相应状态记录管理临界区进入与退出。
- 在修改任务状态、资源状态和调度相关数据时考虑中断并发访问。
- 在退出保护区域时恢复相应的中断状态。
- 控制关键区域的执行范围，避免不必要地延长中断响应延迟。

设计目标

在 8051 的单核执行环境下，协调任务上下文与中断上下文对共享数据的访问，减少因状态更新交错而产生的内核异常。

19.3.9 9. 明确的模块边界与接口约束

设计原则

按照功能职责组织内核与外围模块，保持模块之间的调用关系清晰，减少重复实现和不必要的交叉依赖。

设计体现

- 将任务调度、等待唤醒、资源管理和上下文切换等职责分别组织。
- 通过统一的内部接口连接相关内核模块。
- 将硬件相关实现与上层功能逻辑适当分离。
- 将驱动、通信协议和 Shell 等能力作为外围功能进行组织。
- 通过模块文档说明功能职责、接口用法及相互依赖关系。

设计目标

使内核结构更容易理解和维护，同时允许外围功能在不改变核心任务管理机制的情况下持续扩展。

19.3.10 10. 以实际工程验证驱动持续优化

设计原则

内核优化以可复现的测试、实际硬件表现和问题定位结果为依据，而不是仅以代码长度或理论上的执行效率作为评价标准。

设计体现

- 对任务切换、延时精度和中断响应等关键路径进行实测。
- 分别测试不同任务机制及调度触发路径的性能表现。
- 对长时间运行、任务切换和资源状态转换进行压力测试。
- 对发现的问题进行定位、修复和回归验证。
- 将稳定的测试结果、性能数据和使用限制整理为技术文档。

设计目标

通过持续测试逐步掌握内核的实际运行特征，使性能改进能够得到验证，缺陷修复能够得到复查，版本演进能够建立在已有工程成果之上。

19.4 内存设计原则

19.4.1 内存分区与访问效率

HRTOS 根据 8051 的存储空间组织方式安排不同类型的数据，兼顾访问效率、容量限制和数据管理需求。

- **DATA**：用于适合直接寻址、需要频繁访问的变量。
- **IDATA**：用于适合间接寻址的内部 RAM 数据。
- **XDATA**：用于存放任务管理数组、上下文记录及其他需要较大空间的数据。

不同存储区的具体使用方式由编译器、芯片型号及内核配置共同决定。内存布局需要与启动代码、栈使用方式及任务切换机制保持一致。

19.4.2 栈空间管理

HRTOS 对任务栈和内核使用的栈空间进行规划，并通过相应的栈指针记录和管理数据跟踪栈空间的使用情况。

设计重点包括：

- 根据任务类型设置相应的栈管理结构。
- 采用固定粒度的空间记录管理用户任务栈分配。
- 通过位图记录栈空间的占用状态。
- 在任务创建和清理过程中维护栈空间的分配记录。

- 结合目标芯片的 RAM 容量确定栈空间配置。

栈空间容量应根据任务调用深度、局部变量、中断嵌套和上下文保存需求综合评估。栈检查及异常处理的具体行为取决于相应配置和实现。

19.4.3 上下文保存与恢复

上下文管理直接影响任务切换的执行开销，也是 HRTOS 面向 8051 进行优化的重要组成部分。

HRTOS 根据普通任务与高速任务的不同执行机制，分别组织上下文记录及相关栈指针信息。通过合理使用寄存器组和专用保存区域，减少部分执行路径上的现场保存与恢复工作。

上下文保存范围必须满足相应任务机制的运行要求。优化重点是在保证任务状态正确恢复的前提下，控制上下文大小和切换成本。

19.5 API 设计原则

19.5.1 接口一致性

内核 API 尽量采用统一的命名风格、参数约定和返回值表达方式，使任务管理、等待、资源同步及通信功能具有相对一致的使用体验。

接口文档应明确说明：

- 函数用途和适用场景。
- 参数含义及有效范围。
- 返回值及其对应的处理结果。
- 是否允许在中断上下文中调用。
- 是否可能阻塞，以及相关超时行为。
- 对任务状态、资源状态或系统调度产生的影响。

19.5.2 轻量化接口

接口设计以明确表达操作意图为目标，避免引入与目标平台资源条件不匹配的复杂抽象。

在满足功能需求的前提下，优先采用简洁的参数形式、清晰的状态约定和易于理解的调用方式。对于确实需要额外配置的功能，通过独立参数、配置项或专门接口进行表达。

19.5.3 阻塞、非阻塞与超时语义

涉及等待对象或消息队列的接口，应明确区分立即返回、阻塞等待和超时返回等行为。

不同模式需要具有清晰的完成条件和返回语义，使应用程序能够根据任务优先级、响应需求及资源使用方式选择适当的调用形式。

具体支持的模式以各接口的实际定义为准。

19.6 性能优化原则

19.6.1 优化关键执行路径

优先关注对实时响应和任务执行效率影响较大的路径，包括：

- 任务上下文保存与恢复。
- 调度请求的产生和处理。
- 等待任务的唤醒。
- 中断进入、退出及相关状态处理。
- 常见资源操作的快速处理路径。

在满足正确性要求的前提下，通过减少不必要的状态操作、控制临界区长度和优化上下文管理降低执行开销。

19.6.2 差异化性能设计

不同任务机制具有不同的执行需求，内核不必为所有任务采用完全相同的处理方式。

HRTOS 通过普通任务与高速任务的差异化管理，针对部分对切换开销敏感的场景进行优化。相关性能应结合实际芯片、编译配置、调度模式和测试条件进行评估。

19.6.3 以测量结果评价性能

性能数据应尽可能明确测试平台、时钟频率、编译条件、测试方法和测量范围。

任务切换耗时、中断响应延迟和延时精度属于不同的性能指标，不能相互替代。单项测试结果也不直接等同于整个系统的最坏情况执行时间。

通过持续积累可复现的测试数据，建立内核性能变化的比较依据。

19.7 可移植性原则

19.7.1 硬件相关实现隔离

HRTOS 的内核机制与具体芯片的硬件资源之间需要保持明确的接口关系。定时器配置、寄存器定义、中断向量和底层初始化等内容，应根据目标芯片的特点进行适配。

19.7.2 配置驱动的移植

不同 8051 芯片在 RAM 容量、定时器、中断机制、寄存器组及编译器支持方面存在差异。移植时需要综合检查：

- 内部 RAM 与外部 RAM 的容量及地址布局。
- 内核和用户栈的空间分配。
- 定时器时钟及中断周期。
- 中断优先级与嵌套行为。
- 汇编代码和编译器扩展的兼容性。
- 任务数量及其他内核对象的配置限制。

移植的重点不仅是代码能够编译通过，还包括任务切换、时序行为和内存使用能够在目标硬件上正确运行。

19.8 可靠性与维护原则

19.8.1 状态一致性

任务创建、等待、唤醒、删除和清理等操作会涉及多个内核数据结构。相关流程需要保持任务状态、等待信息、栈空间记录和资源等待记录之间的一致性。

19.8.2 边界条件处理

内核接口需要针对其定义范围内的参数和状态进行检查，包括任务标识、优先级、栈空间参数、资源索引以及重复注册等情况。

具体检查范围由各接口的设计要求决定。文档应清晰说明参数约束和异常返回行为，帮助应用开发者正确使用接口。

19.8.3 回归测试与版本维护

内核功能调整不仅需要验证新增或修改的路径，也应关注对现有功能的影响。

持续维护工作包括：

- 修复已确认的内核缺陷。
- 验证任务状态转换及资源管理流程。
- 复测关键性能指标。
- 检查不同配置下的内存占用。
- 更新 API、模块、移植和测试文档。
- 保持示例程序与公开接口的一致性。

HRTOS 的维护目标是在持续完善功能的同时，尽可能保持已有接口、架构约定和应用代码的稳定性。

19.9 设计权衡

HRTOS 的架构选择围绕 8051 平台的资源条件与实际应用需求展开，各项设计之间存在相互影响。

设计方向	主要收益	主要约束
轻量化内核	控制代码规模和运行开销	内核功能需要合理取舍
固定规模管理结构	内存占用清晰、容量便于核算	运行容量受编译配置限制
差异化任务机制	针对不同执行路径优化开销	需要分别管理相应的上下文
统一等待模型	减少重复状态管理逻辑	不同等待对象仍有各自的语义
8051 原生优化	充分利用目标处理器的特点	依赖相应的处理器和工具链特性
中断与调度分离	明确中断处理和任务切换的职责	任务响应还取决于中断退出及调度条件
固定内存布局	便于分析内核数据占用	地址规划需要与配置保持一致
持续测试与优化	以实测结果指导改进	测试结论受平台和覆盖范围限制

19.10 后续演进方向

HRTOS 的后续演进以现有内核架构为基础，优先完善已有功能的稳定性、文档质量、测试覆盖和工程集成能力。

19.10.1 内核与性能

- 持续验证任务切换、延时和中断响应等关键路径。
- 完善任务栈使用情况及相关异常的检测能力。
- 优化资源等待、唤醒和状态管理流程。
- 在保持现有机制清晰的前提下评估必要的性能改进。

19.10.2 功能与生态

- 完善 Shell、Modbus RTU 等现有功能模块的文档与示例。
- 持续丰富驱动接口及常用外围组件。
- 完善模块依赖、内存布局、API 和移植说明。

- 提高示例工程的完整性与可复现性。

19.10.3 工程质量

- 扩充不同任务组合及边界条件下的测试。
- 持续整理实际硬件测试数据。
- 加强内核实现、接口文档和示例代码之间的一致性。
- 对已验证稳定的功能保持谨慎修改，降低版本演进对现有应用的影响。

HRTOS 将继续围绕轻量化内核、8051 原生优化和完整工程生态推进建设。设计演进以实际需求、验证结果和长期维护价值为依据，在扩展系统能力的同时，保持内核结构清晰、资源使用可控以及现有功能的稳定性。

Extension Guide

HRTOS 扩展指南

20.1 模块介绍

HRTOS 扩展指南用于说明如何在现有系统架构上增加功能、扩展接口、完善外围模块或适配目标硬件。

HRTOS 的扩展工作分为内核扩展、现有模块扩展、公共 API 扩展、外围组件开发和硬件适配等类型。不同类型的扩展涉及不同的内核机制、资源配置及验证要求，应根据实际修改范围确定实施方案。

扩展工作的基本要求是：

- 明确扩展目标及其所属模块。
- 了解相关内核机制和现有接口。
- 评估代码、RAM、XDATA 及任务栈的变化。
- 验证新增功能与现有功能之间的协作关系。
- 同步维护接口说明、使用示例和相关技术文档。

20.2 扩展类型

20.2.1 内核功能扩展

内核扩展涉及任务管理、调度、等待唤醒、资源管理、中断处理、上下文切换及内存管理等机制。

此类修改可能影响多个模块之间的协作关系。实施前应梳理相关数据结构、调用路径和状态转换，并明确修改对任务执行及中断处理的影响。

20.2.2 现有模块扩展

现有模块扩展包括为任务管理、资源同步、消息队列、Shell、Modbus RTU 等功能增加接口或完善已有能力。

优先沿用模块现有的接口形式、配置方式和错误处理约定，使新增功能与模块原有能力保持一致。

20.2.3 公共 API 扩展

新增公共 API 时，应明确函数的用途、参数、返回值、调用条件和相关状态变化。

如果新接口涉及阻塞等待、任务唤醒或中断上下文，还应说明相应的调用限制和执行行为。

20.2.4 外围组件扩展

外围组件包括设备驱动、通信协议及面向应用的辅助功能。

这类功能应尽可能通过公开接口与内核协作，避免在外围模块中重复实现任务调度、等待唤醒等已有内核机制。

20.2.5 硬件适配

硬件适配涉及目标芯片的存储空间、定时器、中断向量、寄存器组和编译工具链等内容。

HRTOS 面向 8051 架构设计，适配不同 8051 芯片时，需要验证底层硬件差异与现有上下文管理、内存布局及调度机制的兼容性。若目标平台不再属于 8051 架构，则需要重新评估处理器相关实现，而不仅是替换硬件寄存器定义。

20.3 扩展实施流程

20.3.1 1. 分析现有实现

开始修改前，先确定功能所属模块及其依赖关系。

重点检查：

- 现有功能是否已经提供相应接口。
- 相关模块的数据结构和状态定义。
- 调用路径及模块间依赖。
- 是否涉及任务状态、资源状态或中断状态。
- 现有配置对新增功能的限制。

对于内核功能，应同时关注功能入口、状态更新、异常返回和资源清理等相关流程，避免只修改单个函数而遗漏关联路径。

20.3.2 2. 确定扩展方案

根据功能需求确定修改范围，并优先选择影响范围明确的实现方式。

- 能通过外围模块实现的功能，尽量不修改内核。
- 能通过新增接口实现的需求，优先评估新增接口方案。
- 需要修改已有行为时，明确受影响的调用方和兼容性变化。
- 涉及多个模块时，先确定各模块的职责及接口关系。
- 涉及底层机制时，明确其对现有任务切换和中断处理的影响。

20.3.3 3. 实现功能

实现过程中应沿用相关模块已有的数据组织方式、接口约定和错误处理模式。

对于涉及共享数据的操作，应考虑任务与中断上下文之间的并发访问；对于涉及等待和唤醒的功能，应保持相关任务状态、等待记录及资源状态之间的一致性。

新增配置项时，应明确默认值、有效范围及其对内存布局和功能行为的影响。

20.3.4 4. 完成功能验证

功能实现后，应先验证基本调用流程，再检查边界条件和异常路径。

根据扩展类型，选择适当的功能测试、集成测试、性能测试及长时间运行测试。涉及内核调度或中断机制的修改，还应验证对原有任务执行行为的影响。

20.3.5 5. 更新配套资料

扩展完成后，同步更新受到影响的技术资料，包括：

- API 文档及参数说明。
- 模块说明和依赖关系。
- 配置项及内存布局说明。
- 使用示例和测试程序。
- 性能测试结果及适用条件。
- 必要的版本变更记录。

20.4 各类功能的扩展方法

20.4.1 新增 IPC 机制

新增进程间通信机制前，应先确认现有资源同步、消息队列和等待接口是否能够满足需求。

设计时需要明确：

- 通信对象的数据结构及容量。
- 发送、接收、等待或释放操作的语义。
- 是否支持阻塞、非阻塞及超时行为。
- 等待任务的记录方式。
- 任务唤醒及资源状态更新流程。
- 是否需要提供中断上下文接口。

如果新机制能够使用现有等待模型，应优先复用相关任务状态管理和唤醒流程；如果其语义或性能要求不同，则应明确独立实现的必要性及其与现有内核的接口关系。

新增 IPC 后，应重点验证多任务竞争、等待超时、资源释放、任务退出及异常路径。

20.4.2 新增系统服务

系统服务用于向应用或其他模块提供公共功能。

扩展时应明确服务的调用接口、内部依赖、初始化要求及资源使用方式。

如果服务需要在系统启动阶段初始化，应确定其初始化时机和依赖顺序；如果服务运行期间维护状态数据，应明确其访问方式及并发保护要求。

系统服务应保持清晰的功能边界，避免将特定应用的业务逻辑直接耦合到内核。

20.4.3 扩展现有模块

扩展已有模块时，应先确认原有接口的行为及模块内部的状态管理方式，再确定新增功能的实现位置。

修改过程中应重点检查：

- 原有调用路径是否发生变化。
- 新增状态是否影响原有状态转换。
- 错误返回及资源释放是否完整。
- 新旧接口是否具有一致的使用约定。
- 原有功能是否需要增加回归测试。

如果确实需要改变已有接口或行为，应在相应文档中明确说明变化及其影响范围。

20.4.4 新增公共 API

公共 API 应具备明确、稳定的接口定义。

每个新增接口至少需要说明：

项目	说明内容
功能	接口的用途及适用场景

项目	说明内容
参数	类型、含义、有效范围及输入要求
返回值	正常结果、失败条件及对应返回值
调用条件	允许的任务或中断上下文
执行行为	是否阻塞、是否触发调度或改变任务状态
资源影响	对内存、任务栈及相关内核对象的影响
使用示例	典型调用方式及必要的错误处理

新增 API 后，应同步更新公开头文件、API 文档及相关示例。

20.4.5 新增配置项

配置项应具有明确的用途、取值范围和默认行为。

涉及任务数量、资源数量、消息队列容量、栈空间或内存布局的配置，需要检查其与固定数组大小、内存偏移及目标芯片容量之间的关系。

配置变化后，应重新检查受影响的数据占用及功能限制，避免出现配置值与实际存储空间不匹配的情况。

20.4.6 适配新的 8051 芯片

适配新芯片时，应先对比现有目标平台与新平台的硬件差异。

重点检查：

- CPU 内核及指令执行特性。
- 内部 RAM、扩展 RAM 及可用地址范围。
- 定时器时钟、计数方式和中断周期。
- 中断向量、优先级和嵌套行为。
- 寄存器组数量及使用方式。
- 编译器、汇编语法和链接配置。
- 内核栈、任务栈及上下文保存区域。

适配完成后，应从基础启动、定时器运行、任务创建、任务切换、中断处理、延时精度和长时间运行等方面逐步验证。

如果硬件差异影响现有上下文切换或内存布局，应针对相关机制进行适配，而不是仅以编译通过作为完成标准。

20.5 扩展验证

20.5.1 功能验证

检查新增功能的正常执行流程、参数边界、失败返回和资源清理行为。

对于具有状态转换的功能，还应检查重复调用、调用顺序变化及异常退出等情况。

20.5.2 集成验证

检查新增功能与相关模块之间的协作关系，包括：

- 与任务调度和任务状态管理的协作。
- 与等待、唤醒及资源管理机制的协作。
- 与中断处理和临界区保护的协作。
- 与现有 API 和外围组件的兼容性。
- 与不同配置及目标芯片的适配情况。

20.5.3 性能与资源验证

根据功能特点评估其对系统运行的影响，重点包括：

- 代码空间及 RAM、XDATA 占用变化。
- 任务栈和上下文空间需求。
- 关键操作的执行耗时。
- 调度响应和中断处理开销。
- 长时间运行时的稳定性。

性能测试结果应注明测试条件和测量范围。对于尚未进行最坏情况分析的功能，不应仅凭平均耗时或单次测试结果推断其最坏情况执行时间。

20.5.4 回归验证

内核或公共接口发生变化后，应重新验证相关原有功能。

回归范围应覆盖受修改影响的模块及其关联路径。对于涉及任务状态、资源管理、上下文切换和中断处理的改动，应重点检查原有执行流程是否受到影响。

20.6 扩展检查清单

扩展完成后，可根据修改范围逐项检查：

- [ ] 扩展目标及模块边界明确。
- [ ] 已检查现有接口和模块依赖。
- [ ] 新增接口的参数与返回值定义完整。

- [ ] 任务状态、资源状态及异常路径处理完整。
- [ ] 已评估代码空间、RAM、XDATA 和栈空间变化。
- [ ] 已评估调度及中断处理方面的影响。
- [ ] 已完成适用的功能与集成测试。
- [ ] 已完成必要的性能或长时间运行测试。
- [ ] 已检查相关原有功能。
- [ ] 已更新 API 文档、示例及相关模块说明。

20.7 常见扩展问题

20.7.1 修改接口后影响已有应用

修改已有 API 的参数、返回值或执行行为，可能导致原有应用需要调整。

处理方式：优先评估新增接口或兼容性适配方案；如果必须改变原有行为，应明确变更范围，并更新相关文档及示例。

20.7.2 新增功能超出芯片资源限制

新增数组、上下文记录或缓冲区，可能使内存布局超出目标芯片的实际容量。

处理方式：在实现阶段核算新增数据的空间需求，并检查静态地址、数组长度、栈空间和其他内核对象之间的关系。

20.7.3 任务与中断上下文访问冲突

任务和中断服务程序可能访问相同的状态变量或资源对象。如果缺少适当的同步保护，可能产生状态不一致。

处理方式：明确共享数据的访问路径，依据操作要求使用适当的临界区或内核同步机制，并验证相关中断路径。

20.7.4 新增功能影响任务响应

新增操作可能增加临界区长度、任务切换开销或中断处理时间。

处理方式：识别受影响的执行路径，对关键操作进行测量，并根据结果调整实现方式。

20.7.5 功能正常但异常路径不完整

功能在常规场景下可以正常运行，并不意味着超时、资源不足、任务退出或重复调用等情况下也能正确处理。

处理方式：根据接口定义补充边界条件和异常路径测试，重点检查状态恢复及资源释放。

20.8 文档与维护要求

扩展工作完成后，应根据实际修改内容更新相应资料：

- **API 文档**：记录接口用途、参数、返回值及调用限制。
- **模块文档**：说明新增功能、内部职责及模块依赖。
- **内存文档**：更新受影响的数据结构、空间占用和地址布局。
- **配置文档**：记录新增配置项及其影响。
- **示例程序**：提供能够体现接口正确用法的示例。
- **测试记录**：记录测试平台、测试条件、结果和已知限制。
- **版本记录**：说明影响兼容性或改变既有行为的修改。

20.9 总结

HRTOS 的扩展应以现有内核机制为基础，根据功能类型明确修改范围，并充分考虑 8051 平台的内存约束、任务调度和中断处理特点。

通过合理划分模块职责、复用已有机制、评估资源开销和开展针对性验证，可以在保持内核结构清晰的同时，持续完善 HRTOS 的功能体系与工程支持能力。

Debugging

HRTOS 调试指南

21.1 模块介绍

HRTOS 调试指南用于指导开发者定位系统运行过程中出现的异常，涵盖任务创建与运行、任务调度、延时与时间片、中断处理、资源同步、消息通信以及栈空间等方面。

HRTOS 的调试应结合内核运行状态、实际硬件表现和可复现的测试现象进行。对于无法直接观察的问题，可通过串口输出、DEBUG 信息、GPIO 标记及逻辑分析仪等手段逐步缩小排查范围。

调试工作主要包括：

- **现象确认**：明确异常发生的条件、出现频率和影响范围。
- **状态检查**：检查任务状态、等待信息、资源状态及调度相关变量。
- **执行路径分析**：定位异常发生前后的函数调用和中断处理流程。
- **硬件测量**：通过 GPIO、串口及逻辑分析仪观察实际运行行为。
- **问题复现与验证**：通过重复测试确认问题原因，并验证修改结果。

21.2 调试工具与方法

21.2.1 串口输出

串口输出适合观察系统启动过程、接口返回值、关键状态和错误发生的位置。

常见用途包括：

- 检查任务创建及初始化结果。
- 输出关键 API 的返回值。
- 观察任务运行、等待和唤醒的执行顺序。
- 确认定时器及相关配置。
- 记录异常发生前后的关键事件。

调试输出应尽量包含明确的事件标识和必要的状态信息，避免连续输出大量无关内容。

串口发送会消耗 CPU 时间，并可能改变任务调度时序。对于中断响应、任务切换和短时间执行路径，不宜仅依靠串口打印测量耗时。必要时可先记录关键事件，再在非实时敏感路径中统一输出。

21.2.2 DEBUG 信息

DEBUG 机制用于辅助观察内核运行状态和定位异常。具体可观察的变量、事件和错误信息取决于 HRTOS 实际提供的调试接口及编译配置。

使用 DEBUG 信息时，应重点关注：

- 输出信息对应的执行阶段。
- 相关变量的有效范围和实际含义。
- 调试输出是否改变了执行路径。
- 调试配置是否影响代码大小、内存占用或执行时序。

DEBUG 信息可以帮助缩小排查范围，但不能代替对实际调用流程和状态变化的检查。

21.2.3 GPIO 与逻辑分析仪

GPIO 标记配合逻辑分析仪，适合观察任务执行区间、中断处理区间和其他需要测量的硬件时序。

基本测量方式如下：

```
GPIO = 1;

/* 执行需要观察的代码 */

GPIO = 0;
```

逻辑分析仪测得的高电平宽度表示两次 GPIO 操作之间的时间间隔。测量结果还会受到 GPIO 操作开销、函数调用、中断干扰及编译优化等因素影响。

使用时应明确测量起点和终点，并确保测试代码不会改变待测机制的关键行为。

适用场景包括：

- 观察任务切换相关时序。
- 测量指定代码段的执行时间。
- 检查定时器周期及周期性事件。
- 观察中断处理过程。
- 对比不同调度配置下的运行表现。

具体性能测试方法、测试条件和实测数据统一参见《HRTOS 性能测试》相关文档。

21.3 任务调试

21.3.1 任务创建失败

任务创建失败时，首先检查创建接口的返回值，再根据对应错误检查参数及系统配置。

排查内容包括：

- 任务 ID 是否在有效范围内。
- 任务入口地址是否正确。
- 优先级是否符合接口要求。
- 栈空间参数是否有效。
- 是否存在重复注册。
- 当前内存配置是否能够满足栈空间分配要求。
- 系统配置锁是否影响任务创建。

如果创建接口返回成功但任务没有按预期执行，应继续检查任务状态、启动标志、调度条件及任务入口函数，而不是仅凭返回值判断任务已经正常运行。

21.3.2 任务创建成功但未运行

建议按以下顺序排查：

1. **确认任务入口**：检查入口地址和初始化过程是否正确。
2. **检查任务状态**：确认任务是否处于可运行状态，是否已经进入等待状态。
3. **检查启动状态**：核对任务启动标志及相关状态记录。
4. **检查优先级**：确认优先级设置是否符合预期，以及是否存在持续运行的更高优先级任务。
5. **检查等待条件**：确认任务是否正在等待延时、资源或其他事件。
6. **检查调度路径**：确认调度请求能够产生，并且后续调度处理正常执行。

如果其他任务能够正常运行，应进一步比较异常任务与正常任务的创建参数、状态记录和执行路径，缩小问题范围。

21.3.3 任务运行异常

如果任务能够启动，但出现重复执行、提前退出或运行顺序不符合预期，应检查：

- 任务入口函数及其循环逻辑。
- 任务状态是否被意外修改。
- 等待与唤醒操作的调用顺序。
- 是否存在越界访问或共享变量竞争。
- 是否有中断服务程序修改相关状态。
- 栈空间和上下文保存是否正常。

对间歇性异常，应记录异常发生前的关键事件，并尽量构造能够重复触发问题的最小测试程序。

21.4 调度调试

21.4.1 调度未触发

HRTOS 通过调度请求机制衔接内核状态变化与后续调度处理。调度未按预期发生时，应检查整个触发链路。

排查内容包括：

- 调度请求是否在预期操作中产生。
- `OS_SCHED_REASON` 等调度相关状态是否符合预期。
- Timer0 是否正常运行。
- 中断是否被关闭或长时间屏蔽。
- 当前任务和候选任务的状态是否正确。
- 中断退出及后续调度处理是否正常执行。

应区分“调度请求未产生”“调度请求已产生但未被处理”和“调度已执行但目标任务不符合预期”等不同情况。

21.4.2 任务切换顺序异常

任务切换顺序不符合预期时，应结合任务优先级、任务状态和调度模式进行分析。

重点检查：

- 任务优先级配置。
- 当前任务是否满足继续运行的条件。
- 高优先级任务是否处于可运行状态。
- 时间片相关配置及其触发条件。
- 等待任务的唤醒时机。
- 任务上下文保存与恢复过程。

任务切换顺序应依据实际调度规则判断，不能仅凭任务创建顺序或串口输出先后判断调度是否正确。

21.4.3 调度异常与性能问题的区别

如果任务执行顺序错误、任务无法恢复运行或调度请求丢失，应优先进行功能性排查。

如果任务顺序正确，但切换耗时、延时精度或响应时间不符合预期，则应进一步使用 GPIO 和逻辑分析仪等手段进行时序测量。

性能测量的具体步骤和指标定义参见《HRTOS 性能测试》相关文档。

21.5 延时与时间片调试

21.5.1 延时不准确

延时行为异常时，应检查：

- Timer0 的时钟及定时器配置。
- 定时器中断是否按照预期运行。
- 延时参数及其单位。
- 等待状态设置与任务唤醒流程。
- 测量起点、终点及测量工具的误差。

还应区分定时器周期、任务延时请求和任务实际恢复执行的时间。任务从等待状态转为可运行状态，并不意味着它一定在同一时刻立即获得 CPU。

21.5.2 时间片行为异常

检查时间片行为时，应确认当前使用的调度模式及其相关配置，并观察定时器中断、调度请求和任务切换之间的关系。

如果时间片周期本身不正确，应从定时器配置和中断处理入手；如果时间片周期正常但任务切换行为不符合预期，则应继续检查任务状态及调度条件。

21.5.3 时间测量注意事项

时间片周期、任务延时精度、任务切换耗时和中断响应时间属于不同指标。

测量时应明确测试区间，并记录目标芯片、系统时钟、编译配置、调度模式和测试方法。不要将单次测量值直接视为所有运行条件下的固定性能。

21.6 中断调试

21.6.1 中断未响应

中断没有按预期执行时，可按以下顺序排查：

1. 确认对应中断源已正确配置。
2. 检查全局中断及相应中断使能状态。
3. 检查中断向量和入口地址。
4. 检查中断优先级及嵌套配置。
5. 检查中断服务程序是否进入。
6. 检查中断退出流程是否正常完成。

如果中断可以进入但任务没有按预期恢复运行，应继续检查中断处理期间产生的调度请求及中断退出后的调度处理。

21.6.2 ISR 执行异常

中断服务程序应尽量缩短执行路径，避免执行阻塞操作，并遵循对应接口的中断上下文使用要求。

重点检查：

- 是否调用了不允许在 ISR 中使用的接口。
- 是否存在过长的中断处理过程。
- 是否错误修改任务或资源状态。
- 是否正确处理嵌套中断和退出流程。
- 是否产生了预期的调度请求。

21.6.3 HRTOS 中断与调度流程

一般任务调度相关的中断处理关系可概括为：

```
硬件中断发生
        ↓
进入对应中断处理流程
        ↓
执行必要的中断处理
        ↓
更新相关状态或产生调度请求
        ↓
完成中断退出流程
        ↓
内核处理相应调度请求
        ↓
根据任务状态及调度规则决定是否切换任务
```

实际执行路径取决于具体中断类型及相关内核机制。并非所有中断都会触发任务切换，也并非所有中断处理都会唤醒任务。

21.7 栈与内存调试

21.7.1 栈空间异常

栈空间异常可能表现为系统运行不稳定、局部变量异常、返回地址异常或意外复位，但这些现象也可能由其他内存问题引起。

排查时应检查：

- 任务栈的配置和实际分配情况。
- 函数调用层级及局部变量占用。

- 中断嵌套导致的额外栈需求。
- 上下文保存和恢复相关的栈指针记录。
- 任务创建和删除过程中栈空间记录的更新。
- 是否存在数组越界或非法内存访问。

应结合目标芯片的内部 RAM、XDATA 布局及实际栈使用方式分析问题，不宜只根据异常现象直接判定为栈溢出。

21.7.2 内存状态异常

如果任务状态、资源状态或其他变量出现非预期变化，应检查相关内存是否被越界访问，以及是否存在多个执行路径同时修改同一数据的问题。

对于固定地址分配的内核数据，还应核对数组长度、地址偏移和占用范围，确认不同数据区域之间不存在意外重叠。

具体地址及空间分配参见《HRTOS 内存布局》。

21.8 IPC 与资源同步调试

21.8.1 等待任务无法唤醒

任务等待后没有按预期恢复运行时，应依次检查：

- 等待接口的返回值。
- 任务控制块中的等待类型和等待对象。
- 资源对象的等待记录。
- 释放、通知或超时处理是否实际发生。
- 唤醒流程是否正确更新任务状态。
- 调度请求是否产生并得到后续处理。

如果任务已被标记为可运行，但没有实际执行，应进一步检查调度条件，而不是继续重复执行资源释放操作。

21.8.2 资源状态异常

资源同步出现异常时，应检查资源对象的当前状态、所有者信息、等待任务记录以及对应 API 的调用顺序。

重点关注：

- 资源初始化是否正确。
- 获取与释放操作是否匹配。
- 等待任务是否与实际等待对象对应。

- 资源释放时等待记录是否正确更新。
- 任务退出或删除后是否遗留相关资源状态。

具体检查内容应与所使用的资源类型和 API 语义一致。

21.8.3 消息队列异常

消息发送或接收行为异常时，应检查：

- 队列初始化及缓冲区配置。
- 队列容量和当前消息数量。
- 消息发送与接收接口的返回值。
- 队列满、队列空等边界条件。
- 发送方与接收方的任务状态。
- 多任务访问时的同步保护。

对于消息内容异常，还应检查消息长度、缓冲区边界以及消息生产和消费双方对数据格式的约定。

21.9 常见问题排查表

问题现象	优先检查内容	推荐方法
系统无响应	Timer0、中断状态、调度请求	检查关键状态并观察 GPIO
任务未运行	创建结果、任务状态、等待条件	检查返回值及任务状态记录
调度顺序异常	优先级、运行资格、调度模式	记录任务状态变化
延时不准确	定时器配置、唤醒流程	逻辑分析仪测量
中断无响应	中断使能、向量、入口及退出流程	GPIO 标记与状态检查
任务无法唤醒	等待对象、资源记录、唤醒流程	跟踪等待与唤醒路径
资源状态异常	获取、释放及清理流程	检查资源状态变化
运行中出现异常数据	越界访问、共享状态、栈空间	检查内存布局并缩小复现范围
性能下降	调试输出、临界区、切换路径	对照测试条件进行时序测量

21.10 调试与性能测试的配合

调试和性能测试的目的不同，但可以互相补充。

调试阶段的重点是确认执行路径、状态变化和异常来源；性能测试阶段的重点是采用明确的测量方法，量化执行耗时、延时精度和响应表现。

建议按照以下顺序进行：

1. 使用 DEBUG 信息或串口输出确认异常现象。
2. 根据任务状态、资源状态和中断流程缩小问题范围。
3. 对涉及时间的异常使用 GPIO 和逻辑分析仪进行测量。
4. 修复问题后重新执行原有复现测试。
5. 对受影响的功能和性能指标进行回归验证。

如果调试输出改变了原有时序，应减少输出或改用硬件标记重新测量。性能数据应记录对应的编译配置、硬件环境及测试条件。

详细性能指标、测试程序及实测结果参见《HRTOS 性能测试》。

21.11 调试最佳实践

21.11.1 分阶段增加系统复杂度

建议先验证基本启动和单任务运行，再逐步加入多任务调度、延时、资源同步、中断及外围模块。

每个阶段都应确认新增功能正常后，再进入下一阶段。这样能够减少多个问题同时出现时的定位难度。

21.11.2 保持问题可复现

记录以下信息：

- 目标芯片及硬件环境。
- 系统时钟与编译配置。
- 任务、资源及相关配置参数。
- 触发问题的操作顺序。
- 异常现象及出现条件。
- DEBUG 输出和测量结果。

对于偶发问题，应尽可能保留触发条件，并逐步减少与问题无关的任务和外围功能。

21.11.3 修改后进行回归验证

内核代码或配置发生变化后，应重新编译，并针对修改涉及的模块执行相应测试。

涉及任务调度、等待唤醒、资源管理或中断处理的修改，还应检查相关联的执行路径。对于长期运行问题，应根据风险和修改范围安排持续运行测试。

21.11.4 区分功能异常与测量误差

观测到的时间变化可能来自真实的内核行为，也可能来自串口输出、GPIO 操作、编译优化或其他中断干扰。

排查时应先确认测量方法和测量区间，再判断是否存在内核功能异常，避免将测量误差误认为系统缺陷。

21.12 总结

HRTOS 调试应从可复现的现象出发，结合内核状态、实际调用流程和硬件测量逐步定位问题。

串口与 DEBUG 信息适合观察执行过程，任务状态及资源记录有助于分析内核逻辑，GPIO 和逻辑分析仪则适合验证实际时序。对于涉及时间性能的问题，应结合独立的性能测试方法和测试记录进行判断。

通过规范的问题复现、针对性的状态检查和修改后的回归验证，可以逐步提高 HRTOS 问题定位的效率，并为内核的持续维护提供可靠依据。