Overview

HRTOS 概述

1.1 系统简介

HRTOS（Hard Real-Time Operating System）是一款面向 8051 系列单片机的实时操作系统，致力于在有限的硬件资源下提供完整的多任务运行环境、实时调度机制、内核同步与通信能力，以及配套的开发文档和示例。

HRTOS 4.0 在既有内核基础上持续完善任务管理、调度与切换、中断处理、时间管理、资源同步和任务间通信等机制，面向需要在 8051 平台上构建结构化、可维护嵌入式软件的开发场景。

HRTOS 不仅关注内核本身，也重视系统的工程完整性。通过配套 API、技术文档、示例程序、驱动与通信功能，帮助开发者从底层硬件控制逐步构建具备多任务协作能力的应用系统。

1.1.1 设计目标

- **面向实时任务**：提供基于任务优先级的调度机制，并支持相应的任务切换与时间管理能力。
- **适配有限资源**：围绕 8051 的存储空间、寻址方式与处理器特性设计内核实现，兼顾功能完整性与资源占用。
- **完善任务协作**：提供任务管理、同步原语、消息传递和等待唤醒等机制，降低多任务应用的组织复杂度。
- **协调任务与中断**：提供内核中断处理机制及相应的任务协作接口，支持对响应时间敏感的嵌入式应用。
- **重视工程可维护性**：通过 API 文档、示例、驱动与通信模块，建立从内核到应用的配套开发体系。
- **持续维护与完善**：以稳定性、可理解性和长期可用性为目标，持续完善内核实现与配套资料。

1.1.2 适用场景

HRTOS 可用于具有相应硬件资源和实时需求的 8051 嵌入式项目，例如：

- 工业控制与设备状态管理

- 实时数据采集与处理
- 串行通信及通信协议实现
- 电机控制与多任务协同
- 传感器节点与嵌入式终端
- 需要任务调度、资源同步或消息传递的控制系统

实际适用性取决于具体芯片的 RAM、XDATA、程序存储空间、定时器、中断资源以及应用的时序要求。

1.2 系统架构

HRTOS 以 8051 内核运行机制为基础，将任务执行、调度决策、时间管理、中断处理与任务间协作组织为相互配合的功能模块。

1.2.1 内核组成

模块	主要职责
内核核心（Kernel）	提供内核运行所需的基础机制与调度协作接口
任务管理（Task）	管理任务状态、挂起与恢复、优先级及任务生命周期相关操作
任务调度（Scheduler）	根据内核调度规则处理任务选择与任务切换
同步机制	提供信号量、互斥锁等任务协作机制
通信机制	提供消息队列、邮箱等任务间数据传递方式
统一等待（Wait）	为相关等待操作提供统一的阻塞状态与唤醒处理接口
时间管理（Time）	提供系统节拍、任务延时、时间转换与运行时间查询等能力
中断管理（Interrupt）	配合中断处理、内核运行状态与任务调度
系统配置（Config）	管理任务数量、资源容量和时间参数等编译配置

事件机制及其他扩展功能的具体能力，应以 HRTOS 4.0 对应的 API、实现代码和模块文档为准。

1.2.2 8051 平台适配

HRTOS 面向 8051 架构设计，内核实现需要考虑该架构的寄存器组织、存储空间划分、中断机制以及不同芯片的硬件差异。

在适配具体芯片时，应重点确认：

- 内部 RAM、位寻址空间及 XDATA 的容量与使用情况
- 程序存储空间和编译后代码占用
- 定时器、中断源及相关寄存器配置
- 编译器、存储模型与芯片运行模式
- 系统配置与目标硬件资源之间的匹配关系

不同 8051 芯片在存储资源、指令执行效率和外设配置方面存在差异。因此，具体资源需求、时序表现与功能支持范围应以目标芯片、编译配置和实际测试结果为准，不宜将某一测试平台的内存地址或资源占用直接视为所有平台的固定要求。

1.3 任务管理与调度

1.3.1 多任务运行

HRTOS 为应用程序提供任务管理与任务切换机制，使开发者能够将不同功能拆分为相对独立的任务，并通过调度、等待和通信机制协调任务之间的执行关系。

任务管理相关能力包括：

- 查询任务状态与当前任务信息
- 挂起与恢复任务
- 设置任务优先级
- 主动让出执行机会
- 处理任务退出与删除
- 通过等待、唤醒和延时机制协调任务执行

具体操作的有效条件、返回值及边界行为，以任务管理 API 文档和对应实现为准。

1.3.2 优先级与任务类型

HRTOS 的调度体系包含普通任务与高速任务等不同执行路径。系统通过相应的任务状态、调度条件和任务切换机制组织任务运行。

普通任务的优先级用于表达任务之间的执行优先关系。高速任务则具有独立的调度与切换处理特点，适用于对任务响应路径有更高要求的场景。

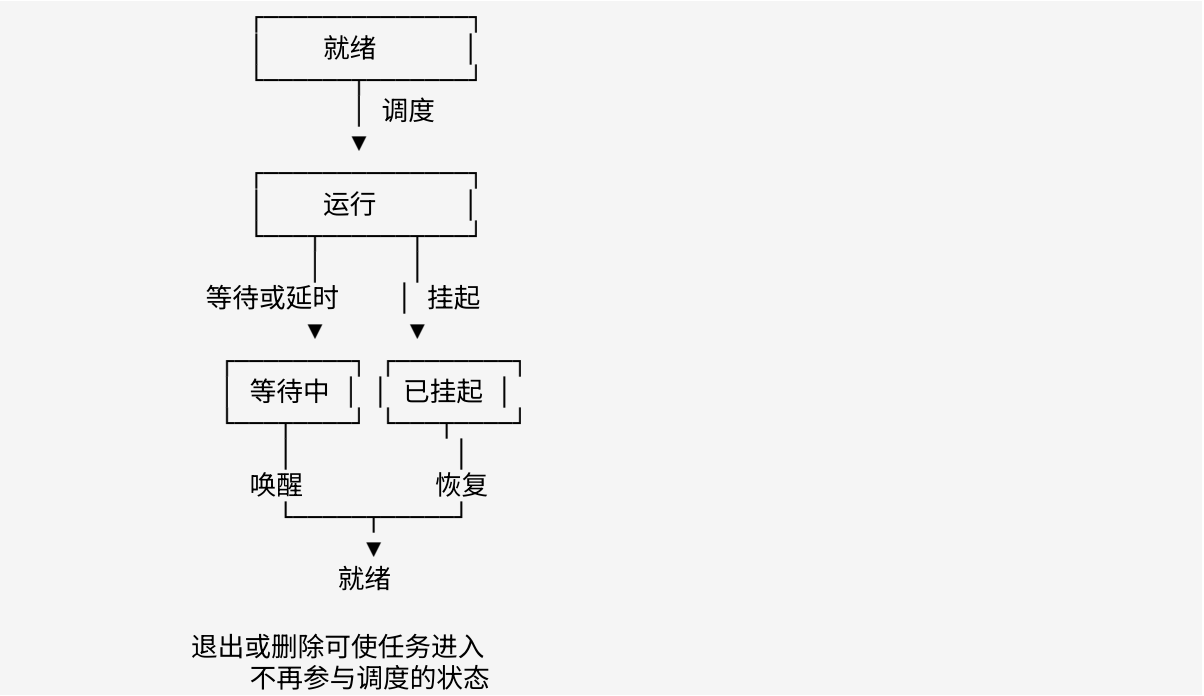
使用时需要区分以下概念：

- **任务优先级**：用于表达任务之间的调度优先关系。
- **任务类型**：决定任务使用的相应执行与切换路径。
- **时间片机制**：用于处理系统配置所涉及的节拍或调度时机。
- **中断优先级**：由处理器中断控制机制及具体中断配置决定，不应简单等同于普通任务优先级。

任务切换的实际耗时受处理器运行模式、晶振频率、编译器、切换路径和中断活动等因素影响。实时应用应结合目标硬件的测量数据评估响应能力。

1.3.3 任务状态

任务运行涉及就绪、运行、等待、挂起及退出或删除等状态。状态之间的转换由任务管理接口、等待唤醒机制和调度过程共同决定。



该图用于说明常见的任务状态关系，不代表每个 API 都允许任意状态转换。具体状态值、查询结果和转换条件应参阅任务管理模块文档。

1.4 同步与通信机制

HRTOS 提供多种任务协作方式，以适应资源共享、执行同步和数据传递等不同需求。

1.4.1 信号量

信号量用于任务同步及资源计数等场景。任务可以通过获取与释放操作协调执行顺序，具体行为取决于信号量实现和相关 API。

使用信号量时，应明确资源计数的含义、等待条件、超时处理，以及任务与中断之间的调用边界。

1.4.2 互斥锁

互斥锁用于协调多个任务对共享资源的访问，减少并发执行造成的数据冲突。

互斥锁的使用应遵循其所有权与释放规则。递归获取、优先级继承、超时行为及中断上下文限制等细节，应以当前版本的互斥锁实现为准，不能仅根据模块名称推断其支持范围。

1.4.3 消息队列

消息队列用于在任务之间传递数据，适用于生产者与消费者分离、任务解耦及异步处理等场景。

消息队列的容量、消息格式、发送与接收行为，以及队列满或空时的处理方式，由对应 API 与系统配置决定。

1.4.4 邮箱

邮箱用于任务之间的消息传递，适合数据量较小、交互方式相对简单的通信场景。

邮箱与消息队列的区别应从消息格式、存储方式、容量及 API 语义理解，具体行为以邮箱模块文档为准。

1.4.5 事件与统一等待

事件可用于表达某种状态变化或同步条件。统一等待机制则为相关内核操作提供等待状态记录与唤醒处理。

统一等待并不意味着所有内核对象都采用完全相同的数据结构，也不意味着不同模块具有相同的阻塞、超时和返回值语义。使用具体对象时，应查阅对应模块的 API 文档。

1.5 中断与任务协作

8051 应用通常需要通过中断处理定时事件、外部信号和通信收发等实时活动。HRTOS 的内核运行机制需要协调中断处理与任务调度，避免中断事件与任务状态更新之间出现不一致。

1.5.1 中断处理原则

- 在中断服务程序中优先完成必要的硬件状态处理。
- 对需要较长时间完成的工作，可根据系统接口设计交由任务处理。
- 严格遵循各 API 对中断上下文的调用限制。
- 评估临界区、嵌套中断及调度触发对响应时间的影响。
- 对涉及任务状态或共享资源的操作，确认其并发安全条件。

1.5.2 中断上下文 API

HRTOS 的部分模块可能提供专用于中断上下文的接口，例如带有 `_from_isr` 后缀的 API。此类接口的具体支持范围、允许执行的操作及唤醒行为，应以 HRTOS 4.0 的公开声明和实现为准。

不能仅凭函数名称认定所有通信与同步操作都可以在中断中调用，也不应假定中断中记录的状态一定会由某个统一后台流程自动处理。开发者应针对具体 API 核实调用约束。

1.6 时间管理

1.6.1 系统节拍

HRTOS 使用定时器相关机制提供系统时间基准与任务延时支持。当前配置和目标硬件决定实际节拍周期，系统时间精度也会受到时钟源、定时器计数方式、编译配置及中断执行情况等因素影响。

1.6.2 时间转换与运行时间

时间管理模块提供与系统节拍相关的转换和查询接口，包括：

- `os_ms_to_tick()`：将毫秒数转换为相应的 tick 数。
- `os_sec_to_tick()`：将秒数转换为相应的 tick 数。
- `os_tick_get()`：查询系统节拍计数。
- `os_uptime_ms()`：获取以毫秒表示的系统运行时间。

上述接口的取整规则、有效参数范围及精度，应以时间管理模块文档为准。tick 与毫秒之间的换算结果受实际节拍配置影响。

1.6.3 任务延时与忙等待

HRTOS 提供不同用途的延时方式：

- `os_delay()`：通过系统等待机制请求任务延时，适合需要让出处理器执行机会的任务场景。
- `os_delay_ms()`：通过忙等待方式实现延时，执行期间会消耗处理器时间，适合经过测量并确认满足要求的短时等待场景。

忙等待的实际时长受时钟频率、编译器优化及中断影响。对于具有严格时序要求的硬件操作，应使用目标硬件实测结果验证延时精度。

1.7 系统配置与资源管理

HRTOS 通过系统配置控制任务容量、资源对象数量、消息队列容量及时间参数等。配置项名称、默认值和有效范围应以当前 HRTOS 4.0 源码中的配置头文件为准。

配置系统时，应综合考虑：

- 目标芯片的内部 RAM 与 XDATA 容量
- 任务数量及每个任务所需的运行资源
- 内核控制结构与通信缓冲区的占用
- 系统节拍和延时参数
- 中断与任务并发执行时的资源使用情况
- 编译器存储模型及生成代码的实际占用

增加任务数量或通信资源容量可能带来额外的内存需求。对于资源有限的 8051 芯片，应在编译后检查内存使用报告，并在目标硬件上验证系统行为。

1.8 开发与验证

HRTOS 的工程使用不仅包括调用 API，还包括目标芯片配置、应用任务设计、硬件驱动适配和运行验证。

建议开发流程包括：

- 确认目标芯片及编译环境。
- 根据芯片资源选择合适的系统配置。
- 阅读对应模块的 API 文档。
- 从最小示例开始，逐步加入任务、同步与通信功能。
- 对关键任务的响应时间、任务切换耗时和定时精度进行实测。
- 对长时间运行、异常输入及资源边界情况进行验证。
- 在修改内核或配置后重新执行相关测试。

对于实时性要求较高的项目，应明确区分功能测试、压力测试、运行时间测量与严格的最坏执行时间分析。有限时长的测试能够提供实际运行证据，但不能单独证明所有输入和执行路径都满足严格的硬实时期限。

1.9 API 文档结构

HRTOS API 文档按功能模块组织，便于开发者从系统整体概念进入具体接口。

文档	内容
`kernel.md`	内核核心功能
`task.md`	任务管理与任务状态

文档	内容
`semaphore.md`	信号量接口与行为
`mutex.md`	互斥锁接口与使用约束
`msgq.md`	消息队列接口
`mailbox.md`	邮箱接口
`event.md`	事件机制与相关接口
`wait.md`	统一等待与唤醒机制
`time.md`	系统节拍、延时与时间转换
`interrupt.md`	中断处理与相关接口
`config.md`	系统配置与参数说明

实际文档文件名和目录结构以 HRTOS 4.0 文档仓库为准。

1.10 版本信息

- **系统名称**：HRTOS
- **当前文档对应版本**：HRTOS 4.0
- **项目定位**：面向 8051 系列单片机的实时操作系统
- **项目官网**：https://hrtos.com
- **中文介绍站**：https://hrtos.cn
- **联系邮箱**：admin@hrtos.com
- **QQ 群**：523549975
- **作者及版权所有者**：谭舒

HRTOS 持续围绕内核机制、工程配套、文档质量与实际运行验证进行完善。系统能力、硬件适配范围和实时性表现，应以对应版本的源码、正式文档、配置说明与实测结果为依据。

Config

系统配置模块

2.1 模块简介

系统配置模块定义 HRTOS 的基本数据类型、系统配置参数、任务状态、等待类型以及内核核心数据结构，为任务管理、调度、资源管理和消息队列等模块提供基础定义。

本模块主要包括：

- 基本数据类型与常用宏定义
- 系统时间及任务数量配置
- 调度级别定义
- 任务状态、等待类型与唤醒标志
- 任务控制块（TCB）
- 统一资源对象与消息队列控制块
- 内核对象数组的外部声明
- 临界区操作宏与中断退出接口

系统配置参数会影响内核对象数量、数据结构布局及相关资源占用。修改配置时，应结合内核实现、目标芯片资源和应用需求进行评估。

2.2 基本数据类型

2.2.1 类型定义

类型定义	原类型	说明
`u8`	`unsigned char`	无符号字符型
`u16`	`unsigned int`	无符号整型
`u32`	`unsigned long`	无符号长整型
`f32`	`float`	单精度浮点类型
`f64`	`double`	双精度浮点类型
`NULL`	`0`	空指针常量定义

以上类型通过预处理宏定义实现，而不是独立的类型别名声明。实际位宽及浮点类型的具体表示应以目标编译器为准。

本头文件直接将 `NULL` 定义为 `0`，使用时应注意与其他头文件中的同名定义保持兼容。

2.3 系统配置参数

2.3.1 时间配置

配置宏	默认值	说明
`OS_TIME_ONCE_DEFAULT`	`5`	默认时间片数量
`OS_TIME_T0`	`10000`	时间片相关时基配置，源码注释为单个时间片长度

`OS_TIME_T0` 的具体计时单位、定时器配置方式和实际时间长度，需要结合定时器初始化及时间管理源码确认，不能仅根据宏值直接推断实际毫秒数。

2.3.2 任务数量配置

配置宏	默认值	说明
`OS_PROCESS_MAX`	`16`	标准任务数量上限
`OS_FAST_TASK_MAX`	`2`	高速任务数量上限
`OS_MAX_TASK`	`OS_PROCESS_MAX`	标准任务数量配置的别名

标准任务与高速任务的具体存储方式、任务 ID 分配规则及调度关系，应结合任务管理和上下文切换源码确认。

2.3.3 调度级别

`OS_SCHED_LEVEL_MAX` 的默认值为 `10`。

级别	类型	说明
0–7	普通任务优先级	普通任务使用的优先级范围
8	高优先级任务	源码注释定义的高优先级任务级别
9	中断级	中断处理相关级别
10	中断嵌套级	中断嵌套处理相关级别

以上划分来自配置头文件中的注释。具体调度行为、中断嵌套条件及不同级别之间的切换规则，应以调度器和中断处理源码为准。

2.4 任务状态

HRTOS 使用以下宏定义表示任务状态。

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

这些宏用于标识任务状态。各状态的进入条件、退出条件及状态转换规则，需要结合任务管理实现理解。

2.5 等待类型与唤醒标志

2.5.1 等待类型

宏定义	值	说明
`WAIT_NONE`	`0`	无等待
`WAIT_DELAY`	`1`	延时等待
`WAIT_SEM`	`2`	信号量等待
`WAIT_MUTEX`	`3`	互斥锁等待
`WAIT_MSG_SEND`	`4`	消息队列发送等待，源码注释对应队列满的情况
`WAIT_MSG_RECV`	`5`	消息队列接收等待，源码注释对应队列空的情况
`WAIT_EVENT`	`6`	事件等待
`WAIT_MAILBOX`	`7`	邮箱等待

等待类型用于标识任务当前等待的类别。源码注释明确指出，一个任务在同一时刻只允许存在一种等待源。

具体的阻塞、超时和唤醒流程，应以对应内核模块的实现为准。

2.5.2 唤醒标志

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

这两个宏用于区分等待结束的原因。其设置时机、清除时机及对 API 返回值的影响，需要结合等待管理和对象操作代码确认。

2.5.3 无效 ID

宏定义	值	说明
`OS_INVALID_ID`	`0xFF`	无效 ID 标识

该宏提供统一的无效 ID 常量，具体适用范围应以各接口的实现为准。

2.6 任务控制块 OS_TCB

任务控制块用于保存任务的优先级、运行状态及等待信息。等待信息直接存储在任务控制块中。

2.6.1 结构定义

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

2.6.2 成员说明

成员	类型	说明
`base_prio`	`u8`	静态优先级
`cur_prio`	`u8`	动态优先级
`state`	`u8`	任务状态
`wait_type`	`u8`	当前等待类型
`wait_flag`	`u8`	等待结果标志
`wait_obj`	`u8`	等待对象 ID
`wait_tick`	`u16`	超时等待 tick

`base_prio` 和 `cur_prio` 分别记录静态优先级与动态优先级。等待相关成员共同记录任务的等待类别、等待对象、等待结果及超时信息。

动态优先级的调整策略、等待对象 ID 的编码方式，以及 `wait_tick` 的更新机制，应结合相关内核实现进一步说明。

2.7 统一资源对象 OS_RESOURCE

HRTOS 使用统一资源对象结构保存资源值、拥有者信息和等待队列信息。

2.7.1 结构定义

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

2.7.2 成员说明

成员	类型	说明
`value`	`u8`	资源值
`owner`	`u8`	当前拥有者，仅互斥锁使用
`wait_cnt`	`u8`	当前等待任务数量
`wait_mask`	`u16`	位图形式的等待队列
`pending_signal`	`u8`	源码注释注明用于 ISR 中断计数

`wait_mask` 使用位图记录等待任务，bit0 对应 task0，bit1 对应 task1，以此类推，bit15 对应 task15。

该位图的定义覆盖标准任务的 16 个编号位置。实际等待任务的选择顺序、等待计数维护方式，以及 `pending_signal` 的累积与消费规则，应以资源管理和中断相关源码为准。

源码注释将信号量、互斥锁、事件及消息等待相关对象列入统一资源对象的使用说明。各类对象的具体操作流程与适用范围，需要结合相应接口实现确认。

2.8 消息队列控制块 os_msgq_t

消息队列控制块保存消息缓冲区指针、队列容量相关信息及读写状态。

2.8.1 结构定义

```
typedef struct
{
    u8 *buf;        /* 缓冲区指针 */
    u8 _size;       /* 队列容量相关字段 */
    u8 head;        /* 写指针，生产者 */
    u8 tail;        /* 读指针，消费者 */
    u8 count;       /* 当前消息数量 */
```

```
} os_msgq_t;
```

2.8.2 成员说明

成员	类型	说明
`buf`	`u8 *`	消息缓冲区指针
`_size`	`u8`	队列容量相关字段
`head`	`u8`	写指针，生产者使用
`tail`	`u8`	读指针，消费者使用
`count`	`u8`	当前消息数量

从结构定义可以确认，该控制块保存消息缓冲区及队列读写状态。消息格式、缓冲区分配方式、容量边界、指针回绕规则，以及队列满或空时的具体行为，应结合消息队列实现说明。

2.9 资源数量配置

配置宏	默认值	说明
`OS_RESOURCE_MAX`	`6`	统一资源对象数组的元素数量
`OS_MSGQ_MAX`	`3`	消息队列对象数组的元素数量

以上为当前配置头文件中的定义。修改这些参数时，应同步检查数组定义、对象创建接口、访问边界及目标芯片的可用内存。

增加对象数量会增加相应的数据存储需求，实际内存增量取决于结构体布局、编译器实现及存储空间限定。

2.10 全局对象声明

2.10.1 标准任务控制块数组

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

声明标准任务控制块数组，数组长度由 `OS_PROCESS_MAX` 决定，使用 `xdata` 存储空间限定。

2.10.2 统一资源对象数组

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

声明统一资源对象数组，元素数量由 `OS_RESOURCE_MAX` 决定。

2.10.3 消息队列对象数组

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

声明消息队列控制块数组，元素数量由 `OS_MSGQ_MAX` 决定。

以上均为外部声明，表示相应对象在其他编译单元中定义。本头文件声明了对象的类型、名称及数组长度，并未在此处定义数组本体。

2.11 临界区与中断退出接口

2.11.1 临界区操作宏

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

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

`OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()` 分别调用 `os_enter_critical()` 与 `os_exit_critical()`，用于封装临界区进入和退出操作。

临界区的具体保护方式、嵌套支持、原中断状态的保存与恢复规则，应以对应函数的实现为准。使用时应遵守内核规定的调用及配对规则。

2.11.2 中断退出接口

```
extern void os_interrupt_exit();
```

该声明提供 `os_interrupt_exit` 函数的外部接口。其适用条件、调用约束和中断退出行为应参照中断管理模块的说明。

2.12 配置修改注意事项

2.12.1 任务数量

修改 `OS_PROCESS_MAX` 或 `OS_FAST_TASK_MAX` 前，应核实任务创建接口、任务 ID 分配、任务控制块存储及上下文切换逻辑是否支持对应配置。

标准任务与高速任务可能采用不同的存储或调度处理方式，不能仅根据两个数量宏推断任务总数或所有任务的内存布局。

2.12.2 时间参数

修改 `OS_TIME_ONCE_DEFAULT` 或 `OS_TIME_T0` 前，应核实时间片初始化、定时器配置、计时单位及调度触发机制。

时间配置可能影响任务调度频率和时间相关 API 的行为，但实际影响取决于内核实现，不能只根据宏值判断系统响应时间或实时性。

2.12.3 资源数量

修改 `OS_RESOURCE_MAX` 或 `OS_MSGQ_MAX` 前，应检查对应数组定义、对象创建接口、访问边界及目标芯片可用的内部和外部 RAM。

配置的对象数量上限不代表应用程序可以无条件创建相同数量的对象，实际可用数量还受到对象管理实现和运行时约束的影响。

2.12.4 配置与实现一致性

修改配置后，应重新编译并检查相关模块，确保宏定义、数组长度、任务管理逻辑和对象访问范围保持一致。

对于优先级、中断嵌套、超时等待、资源唤醒和消息队列行为等机制性问题，应以对应实现源码和专项文档为准，避免仅凭配置定义作出结论。

Kernel

内核模块

3.1 模块简介

HRTOS 内核模块提供系统启动、任务注册、调度策略控制、任务资源回收、存储区访问、异常处理及空闲任务钩子等基础能力。

模块通过任务注册接口建立任务或中断处理入口，通过调度控制接口切换系统运行策略，并由 Idle 任务完成内核运行环境初始化、用户主程序初始化及任务清理工作。

内核模块同时提供 Slab 异常处理钩子和 Idle Hook，便于用户扩展系统异常处理与空闲阶段的后台行为。

3.2 设计目标

- 提供任务与中断处理入口的注册机制。
- 支持调度模式及 HRTOS/MyOS 调度策略切换。
- 管理任务运行栈空间的分配与回收。
- 提供 DATA 与 XDATA 存储区之间的单字节数据传送接口。
- 提供系统初始化、任务清理及空闲任务运行机制。
- 提供资源分配异常处理与 Idle Hook 扩展接口。

3.3 使用场景

- 在系统启动阶段注册任务及相关处理入口。
- 根据应用需求选择调度模式。
- 在系统初始化阶段完成用户配置。
- 在任务结束或被系统标记为死亡状态后回收任务资源。
- 在系统空闲阶段执行轻量级后台处理。
- 在 Slab 资源分配异常时执行用户定义的处理逻辑。

3.4 数据类型

本模块没有额外公开的结构体或枚举类型。

模块接口使用 `char`、`unsigned char`、`unsigned int`、`u8`、`u16` 及函数指针等类型。其中，`u8`、`u16` 为项目内部使用的类型别名。

3.5 宏定义

本模块没有在所提供的公开头文件中定义新的宏。

任务数量、调度级别、资源数量及任务栈区域等限制由内核配置和内部头文件中的相关定义决定。

3.6 API 函数列表

函数名称	功能说明
`os_dispatch_select`	设置调度器使用的中断编号
`os_task_create`	注册任务或中断处理入口
`os_interrupt_init`	MyOS 初始化及中断支持接口
`os_interrupt_exit`	中断退出处理接口
`os_memset`	内存填充接口
`os_data_to_xdata`	将 DATA 区的一个字节写入 XDATA 区
`os_xdata_to_data`	将 XDATA 区的一个字节读取到 DATA 区
`os_scheduler_mode_switch`	切换时间片模式与优先级模式
`os_set_scheduler`	切换 HRTOS/MyOS 调度策略
`hrtos_main`	用户主程序初始化入口
`os_slab_overflow`	处理 Slab 资源分配异常
`os_slab_overflow_register`	注册 Slab 异常处理钩子
`os_task_cleanup`	清理任务运行资源及相关状态
`os_idle_task`	初始化内核运行环境并执行 Idle 循环
`os_idle_hook_register`	注册 Idle Hook

其中，`os_interrupt_init()`、`os_interrupt_exit()`、`os_memset()` 和 `hrtos_main()` 在本次提供的材料中只有声明或调用信息，具体内部实现需要结合对应源码进一步确认。

3.7 调度器中断编号设置

3.7.1 os_dispatch_select

函数原型

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

功能说明

设置内核调度器使用的中断编号，并将编号保存至内部变量 `OS_DISPATCH_ID`。

参数说明

参数	类型	说明
`interrupt_id`	`unsigned char`	调度器使用的中断编号，有效范围为 0～31

返回值

返回值	含义
`1`	设置成功
`-1`	中断编号超出有效范围

调用示例

```
char result;

result = os_dispatch_select(2);

if (result == 1)
{
    /* 设置成功 */
}
```

注意事项

- 本函数设置的是调度器中断编号，不是选择或返回一个中断处理函数。
- 函数只检查编号是否大于 31，没有检查中断编号是否对应实际存在的硬件中断。
- 提供的实现没有执行临界区保护，因此并不能仅凭本函数认定其具备并发调用保护。

3.8 任务与中断入口注册

3.8.1 os_task_create

函数原型

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

功能说明

注册标准任务、高速任务或中断处理入口。函数根据优先级参数选择相应的注册路径；对于标准任务和高速任务，还会在用户栈空间区域中申请连续空闲块，并记录相应的栈指针、任务信息或上下文数据。

该函数执行的是注册与初始化工作，并不意味着调用后任务函数会立即运行。

参数说明

参数	类型	说明
`addr`	`unsigned int`	任务或处理函数入口地址
`id`	`unsigned char`	任务编号
`pr`	`unsigned char`	用于区分标准任务、高速任务及中断处理入口的优先级参数
`sd`	`unsigned char`	栈空间配置参数

返回值

返回值	含义
`1`	注册路径执行成功
`-1`	系统配置已锁定
`-2`	优先级参数超出允许范围
`-3`	任务编号超出允许范围
`-4`	入口地址为空
`-5`	标准任务的栈空间参数超出限制
`-6`	标准任务编号已注册
`-7`	高速任务的栈空间参数超出限制
`-8`	用户栈空间不足

任务类型与参数限制

按照当前提供的源码和配置定义，任务类型判断逻辑如下：

- 当 `pr < OS_SCHED_LEVEL_MAX - 2` 时，进入标准任务注册路径。
- 当 `OS_SCHED_LEVEL_MAX - 2 <= pr < OS_SCHED_LEVEL_MAX - 1` 时，进入高速任务注册路径。
- 当 `pr == OS_SCHED_LEVEL_MAX - 1` 时，设置中断处理入口地址。
- 当 `pr == OS_SCHED_LEVEL_MAX` 时，设置中断嵌套处理入口地址。
- 当 `pr > OS_SCHED_LEVEL_MAX` 时，返回 `-2`。

如果当前配置中的 `OS_SCHED_LEVEL_MAX` 为 10，则上述分支分别对应标准任务优先级参数 0～7、高速任务参数 8、中断处理入口参数 9 和中断嵌套处理入口参数 10。

标准任务的 `sd` 最大允许值为 6，高速任务的 `sd` 最大允许值为 25。源码会在申请栈空间前将 `sd` 加一，因此不能简单地将传入参数理解为最终分配的栈块数量。

调用示例

```
char result;
```

```
/* 注册一个标准任务 */
result = os_task_create(
    (unsigned int)task_func,
    0,
    3,
    5
);

if (result == 1)
{
    /* 注册成功 */
}
```

上述示例假设 `task_func` 是有效的任务入口，并且任务编号、优先级、栈空间及系统配置均符合要求。

注意事项

- `OS_CFG_LOCK` 非零时，函数拒绝注册操作。
- 标准任务注册会检查重复编号；高速任务采用独立的高速任务存储与标记机制。
- 标准任务和高速任务使用位图管理的用户栈区域，并需要足够的连续空闲空间。
- 中断处理及中断嵌套处理分支在源码中通过标签直接写入入口地址，执行路径与标准任务注册不同，不应将标准任务的全部参数检查规则直接套用到这两个分支。
- 中断处理入口的具体注册方式及其与硬件中断向量的对应关系，需要结合中断初始化和分发代码确认。

3.9 调度模式切换

3.9.1 os_scheduler_mode_switch

函数原型

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

功能说明

切换 HRTOS 的调度模式。根据源码中的参数判断及内部变量操作：

- `id = 0`：切换至时间片模式，将 `OS_RTOS_YES` 置为 0，并保存当前 `OS_TIME_ONCE`。
- `id = 1`：切换至优先级模式，将 `OS_RTOS_YES` 置为 1，调整 `OS_TIME_XY`，再恢复保存的 `OS_TIME_ONCE`。

参数说明

参数	类型	说明
`id`	`unsigned char`	`0` 表示时间片模式，`1` 表示优先级模式

返回值

返回值	含义
`1`	调度模式切换成功
`-1`	参数无效，或系统当前已处于指定模式

调用示例

```
char result;

/* 切换至优先级模式 */
result = os_scheduler_mode_switch(1);

if (result == 1)
{
    /* 切换成功 */
}
```

注意事项

- 该接口修改调度模式相关状态和时间参数。
- 切换时会调整 `OS_TIME_ONCE` 与 `OS_TIME_XY`，因此时间参数的变化应结合系统计时实现理解。
- 源码没有在该函数内部使用临界区保护。
- 建议在应用明确的控制流程中进行模式切换，并验证切换前后的调度与计时行为。

3.10 HRTOS/MyOS 调度策略切换

3.10.1 os_set_scheduler

函数原型

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

功能说明

通过修改 `OS_INTERRUPT_T0` 选择相应的调度策略，并在切换时保存或恢复 `OS_TIME_ONCE` 等时间参数。

根据当前实现：

- 当 `id == 0` 且 `OS_INTERRUPT_T0 == 0` 时，将 `OS_INTERRUPT_T0` 设为 1，并保存 `OS_TIME_ONCE`。
- 当 `id == 1` 且 `OS_INTERRUPT_T0 == 1` 时，将 `OS_INTERRUPT_T0` 设为 0，调整 `OS_TIME_XY`，并恢复保存的 `OS_TIME_ONCE`。
- 其他情况返回 `-1`。

参数说明

参数	类型	说明
`id`	`unsigned char`	调度策略切换标识，具体取值与 HRTOS/MyOS 的对应

参数	类型	说明
		关系需统一确认

返回值

返回值	含义
`1`	状态切换成功
`-1`	参数不符合当前切换条件，或当前状态已不满足切换条件

调用示例

```
char result;

result = os_set_scheduler(1);
```

调用前应先根据项目当前版本确认 `id` 与 HRTOS/MyOS 的对应关系。

注意事项

当前材料存在一处需要审核的定义不一致：

- 公开头文件注释写的是 `1 = HRTOS`、`0 = MYOS`。
- `scheduler_mode.c` 中的函数注释写的是 `0 = HRTOS`、`1 = MYOS`。
- 函数实际实现只体现 `OS_INTERRUPT_T0` 的状态转换，并未直接说明每种状态对应哪一个调度器。

因此，在确认 `OS_INTERRUPT_T0` 的初始化值及相关调度分发代码之前，文档不应武断地把参数值解释为某一确定的调度器名称。

该接口与 `os_scheduler_mode_switch()` 的用途不同：前者改变 HRTOS/MyOS 相关的调度策略状态，后者改变时间片模式与优先级模式。

3.11 DATA 与 XDATA 数据传送

3.11.1 os_data_to_xdata

函数原型

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

功能说明

读取 DATA 区指定地址的一个字节，并将该字节写入 XDATA 区的目标地址。

该函数通过汇编指令访问间接 DATA 地址及 XDATA 地址，适用于 8051 存储区之间的单字节传送。

参数说明

参数	类型	说明
`addr`	`u8`	DATA 区源地址
`addr2`	`u16`	XDATA 区目标地址

返回值

无。

调用示例

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

其中，`src_addr` 和 `dst_addr` 应替换为符合实际存储区布局的地址值。

注意事项

- 该接口传送一个字节，不是通用的多字节内存复制函数。
- 函数执行期间直接设置 `EA = 0`，完成后直接设置 `EA = 1`。
- 实现并未保存并恢复调用前的中断使能状态，因此不能将其描述为严格恢复原中断状态。
- 应确认地址对应的存储区及访问范围有效。

3.11.2 os_xdata_to_data

函数原型

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

功能说明

读取 XDATA 区指定地址的一个字节，并将其写入 DATA 区目标地址。

参数说明

参数	类型	说明
`addr2`	`u16`	XDATA 区源地址
`addr`	`u8`	DATA 区目标地址

返回值

无。

调用示例

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

注意事项

- 该接口同样只传送一个字节。
- 函数通过 `MOVX` 读取 XDATA 数据，再通过间接寻址写入 DATA 区。

- 函数结束时直接设置 `EA = 1`，没有恢复调用前的中断状态。
- 对这两个接口的中断行为进行评估时，应考虑其调用环境以及是否可能在原本关闭中断的代码路径中被调用。

3.12 中断处理接口

3.12.1 os_interrupt_init

函数原型

```
void os_interrupt_init();
```

功能说明

公开头文件将该接口描述为 MyOS 的初始化及中断支持接口。

当前提供的材料没有包含其实现，因此无法确认其具体初始化步骤、寄存器保护范围、栈切换方式及适用条件。

调用示例

```
os_interrupt_init();
```

注意事项

应以该函数的实际实现及 MyOS 的启动流程为准，确认调用时机和适用模式。

3.12.2 os_interrupt_exit

函数原型

```
void os_interrupt_exit();
```

功能说明

公开头文件将该接口定义为中断退出处理接口。

具体如何判断中断嵌套、恢复执行现场以及选择返回路径，需要结合该函数实现和相关中断分发代码确认。

调用示例

```
os_interrupt_exit();
```

注意事项

中断处理函数应遵循 HRTOS 当前版本规定的中断退出流程。由于本次未提供该函数实现，文档暂不对内部返回分支作进一步承诺。

3.13 用户主程序入口

3.13.1 hrtos_main

函数原型

```
void hrtos_main();
```

功能说明

`hrtos_main()` 是 HRTOS 运行流程中的用户主程序初始化入口。

在 `os_idle_task()` 中，内核先完成任务和资源状态的初始化，然后调用 `hrtos_main()`。该函数执行完成后，内核将 `OS_CFG_LOCK` 置为 1，随后进入 Idle 循环。

调用示例

```
void hrtos_main(void)
{
    /* 执行应用初始化 */

    /* 注册用户任务 */

    /* 完成其他启动配置 */
}
```

注意事项

- 应在该入口中完成应用所需的初始化与任务注册。
- 按当前 `os_idle_task()` 的实现，`hrtos_main()` 返回后配置锁会被启用。
- 不应假定系统进入该函数时已经完成所有用户配置。
- 用户任务和其他配置项应按照 HRTOS 的启动要求，在配置锁启用前完成注册。

3.14 任务资源清理

3.14.1 os_task_cleanup

函数原型

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

功能说明

清理指定任务的栈空间占用标记、任务上下文及相关等待状态。对于高速任务，还会清理相应的高速任务注册标记和上下文数据。

函数同时遍历系统资源，清除该任务在等待位图中的记录，并释放其作为 `owner` 持有的资源所有权标记。

参数说明

参数	类型	说明
`id`	`unsigned char`	待清理的任务编号

返回值

返回值	含义
`1`	清理成功
`-1`	任务编号无效，或对应的任务栈记录不存在

调用示例

```
char result;

/* 示例：清理任务 1 */
result = os_task_cleanup(1);
```

是否适合由应用直接调用，应结合任务生命周期管理流程决定。

清理过程

函数主要执行以下操作：

- 检查任务编号及对应的任务栈记录。
- 根据标准任务或高速任务类型计算需要回收的栈空间范围。
- 清除对应的任务注册记录和上下文数据。
- 清除高速任务的时间等待标记（适用于高速任务）。
- 清除该任务在系统资源等待位图中的记录，并递减相应的等待计数。
- 如果该任务是某项资源的 `owner`，则将资源所有者标记设为 `OS_INVALID_ID`。
- 清理任务控制块中的等待类型、等待标志、等待对象和等待时间。
- 将任务状态设置为 `READY`，恢复 `OS_HRTOS` 的原值后返回。

注意事项

- 任务编号 0 不允许清理。
- 当前实现允许的最大任务编号为 `OS_PROCESS_MAX + 1`。
- 标准任务和高速任务采用不同的栈空间及上下文记录。
- 源码中的注释提到“删除态”，但实际赋值为 `OS_TASK[id].state = READY`，因此文档应以实际赋值为准。
- 清理函数自身没有在入口处关闭中断；当前 `os_idle_task()` 在执行清理扫描时会先关闭中断。若在其他上下文调用，需要另行核对并发保护条件。
- 释放 `owner` 标记不等同于已完成所有可能的资源交接处理。若资源等待和所有权交接存在更复杂的约束，需要结合资源模块源码继续验证。

3.15 Idle 任务与空闲钩子

3.15.1 os_idle_task

函数原型

```
void os_idle_task(void);
```

功能说明

`os_idle_task()` 是内核的空闲任务入口，负责初始化任务控制块和系统资源状态、执行用户主程序初始化，以及在系统运行期间处理任务清理请求和调用 Idle Hook。

初始化阶段

函数首先关闭中断，并执行以下操作：

- 清空 Idle Hook 和 Slab 异常处理钩子指针。
- 清除调度请求状态。
- 将 `OS_CFG_LOCK` 设置为 0，允许注册配置项。
- 初始化标准任务的等待类型、状态、等待时间和等待对象。
- 清空高速任务等待标记。
- 初始化系统资源的值、所有者、等待计数、等待位图和待处理信号状态。
- 调用 `hrtos_main()` 执行用户主程序初始化。

  `hrtos_main()` 返回后，函数将 `OS_CFG_LOCK` 设置为 1，并重新开启中断。

运行阶段

随后进入永久循环，持续执行以下操作：

- 临时关闭中断，扫描标准任务编号 1 至 `OS_PROCESS_MAX - 1`。
- 对状态为 `DEAD` 的任务调用 `os_task_cleanup()`。
- 根据高速任务删除标记，清理对应的高速任务。
- 清除已经处理的高速任务删除标记。
- 重新开启中断。
- 如果已注册 Idle Hook，则调用该钩子函数。

注意事项

- Idle 任务的初始化过程是系统启动流程的重要组成部分。
- 当前实现只扫描标准任务编号 1 至 `OS_PROCESS_MAX - 1`，并不在该循环中扫描编号 0。
- Idle Hook 在永久循环中被反复调用，不应将其当作只执行一次的初始化回调。
- Idle Hook 应保持短小，并避免长时间阻塞，否则会影响 Idle 阶段的其他处理。
- 低功耗操作是否适合放入 Idle Hook，需要结合具体芯片的中断唤醒机制和系统运行要求评估。

3.15.2 os_idle_hook_register

函数原型

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

功能说明

注册 Idle Hook。当系统进入 `os_idle_task()` 的循环并执行到钩子调用位置时，内核会调用已注册的函数。

传入空指针可取消已注册的钩子。

参数说明

参数	类型	说明
`hook`	`void (*)(void)`	Idle Hook 函数指针

返回值

无。

调用示例

```
void my_idle_hook(void)
{
    /* 执行轻量级后台处理 */
}

void hrtos_main(void)
{
    os_idle_hook_register(my_idle_hook);

    /* 其他系统初始化 */
}
```

取消钩子时可传入空指针：

```
os_idle_hook_register(0);
```

注意事项

- Idle Hook 可能被频繁调用，应避免长时间执行。
- 不宜在钩子中执行阻塞式等待，否则可能导致 Idle 循环无法继续。
- 如果钩子用于低功耗管理，应确认不会影响中断响应及任务唤醒。
- 注册接口本身没有显示的临界区保护；若存在其他上下文同时修改钩子指针，应审查其同步要求。

3.16 Slab 异常处理

3.16.1 os_slab_overflow

函数原型

```
void os_slab_overflow(void);
```

功能说明

处理 Slab 资源分配异常。函数首先检查 `g_slab_overflow_hook` 是否已注册：

- 已注册钩子时，调用用户提供的异常处理函数。
- 未注册钩子时，进入无限循环。

因此，当前源码中的默认处理行为是停留在循环中，而不是主动执行系统复位。

调用示例

该函数通常由系统内部在相应异常路径中调用，应用一般不需要主动调用。

注意事项

- 异常钩子应尽可能简洁，并考虑发生异常时系统状态是否仍然可靠。
- 如果希望异常后复位或进入特定安全状态，应由用户处理函数显式实现。
- 应结合 Slab 分配相关源码确认触发该处理函数的具体条件。

3.16.2 os_slab_overflow_register

函数原型

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

功能说明

注册 Slab 资源分配异常处理钩子。注册后的函数由 `os_slab_overflow()` 在异常处理路径中调用。

传入空指针可取消钩子注册。

参数说明

参数	类型	说明
`hook`	`void (*)(void)`	用户定义的异常处理函数指针

返回值

无。

调用示例

```
void my_overflow_handler(void)
{
    /* 执行必要的异常记录或安全处理 */
}

void hrtos_main(void)
{
```

```
    os_slab_overflow_register(my_overflow_handler);
}
```

注意事项

- 应根据实际安全需求设计异常处理行为。
- 不应假设异常发生时所有系统服务都能正常使用。
- 当前注册接口直接写入函数指针，没有显示的同步保护。

3.17 内存填充接口

3.17.1 os_memset

函数原型

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

功能说明

从公开接口声明来看，该函数用于将指定内存区域填充为给定的字节值。

参数说明

参数	类型	说明
`buffer`	`void *`	目标内存地址
`c`	`unsigned char`	填充值
`count`	`unsigned int`	填充长度，通常以字节为单位

返回值

无。

调用示例

```
u8 buffer[32];

os_memset(buffer, 0, sizeof(buffer));
```

注意事项

本次材料仅包含函数声明，没有提供具体实现。其支持的存储区类型、地址空间处理方式、边界行为和与标准库 `memset()` 的兼容程度，应以实现源码为准。

3.18 使用建议

- **启动配置：** 在配置锁启用前完成任务注册及其他初始化工作。
- **调度策略：** 区分调度模式切换与 HRTOS/MyOS 策略切换，调用前确认参数定义。
- **任务栈：** 根据任务类型合理设置栈空间参数，并确认用户栈区域有足够的连续空闲块。

- **任务清理：** 通过系统既定的生命周期流程处理任务回收，避免在不明确的上下文中直接调用清理接口。
- **异常处理：** 根据产品需求注册 Slab 异常钩子，并明确异常发生后的预期行为。
- **Idle Hook：** 仅执行适合空闲上下文的轻量级处理，并验证低功耗操作与中断唤醒的兼容性。
- **存储区访问：** 使用 DATA/XDATA 传送接口时，确认地址空间和中断状态管理符合应用要求。

Task

任务管理模块

4.1 模块简介

HRTOS 任务管理模块提供任务状态查询、挂起与恢复、主动让出 CPU、任务退出与删除、优先级调整、当前任务识别，以及任务注册配置锁管理等功能。

模块通过任务编号访问标准任务和高速任务，并结合调度状态、任务运行标志、任务上下文及系统调度控制变量完成任务管理操作。

任务管理接口并不都直接完成任务切换。部分接口负责修改任务状态或调度标志，并通过触发 Timer0 中断请求内核执行后续调度；任务删除也采用请求标记与后续清理相结合的方式。

4.2 设计目标

- 提供统一的任务状态查询与有效性检查接口。
- 支持任务挂起、恢复、主动让出 CPU 和退出。
- 支持运行期间修改任务优先级，并在特定条件下进行标准任务与高速任务之间的转换。
- 提供指定下一次运行任务的调度控制接口。
- 提供当前任务编号查询。
- 管理任务注册配置锁状态。

4.3 使用场景

- 查询任务是否已注册以及当前调度状态。
- 暂停某个任务，并在条件满足后恢复该任务。
- 让当前任务主动请求调度。
- 在任务执行过程中退出当前任务，或请求删除指定任务。
- 根据系统运行需要调整任务优先级。
- 在特定调度控制场景下指定下一次运行的任务。
- 查询或控制系统任务注册配置锁。

4.4 任务编号与任务类型

本模块使用任务编号区分标准任务和高速任务。根据当前实现中的编号判断方式，任务编号范围由 `OS_PROCESS_MAX` 和 `OS_PROCESS_MAX + 1` 确定。

任务类型	编号范围	管理方式
标准任务	`0` 至 `OS_PROCESS_MAX - 1`	通过 `OS_PROCESS_OK[]`、`OS_TASK[]` 等状态信息管理
高速任务 A	`OS_PROCESS_MAX`	通过高速任务 A 对应的运行标志和上下文管理
高速任务 B	`OS_PROCESS_MAX + 1`	通过高速任务 B 对应的运行标志和上下文管理

标准任务与高速任务使用不同的状态管理路径，因此部分接口对两种任务类型的处理方式并不完全相同。

注意： 任务编号是否有效、任务是否已注册、任务是否处于就绪状态，是不同的判断条件。使用接口时应结合具体返回值及任务类型进行判断。

4.5 任务状态说明

`os_task_get_state()` 返回以下状态编码：

返回值	含义	说明
`0`	未注册或已删除	标准任务的注册状态为零，或高速任务对应的上下文标记为零
`1`	当前运行任务	任务编号与 `OS_CURRENT_TASK` 相同
`2`	就绪状态	标准任务的就绪位有效，或高速任务对应的运行标志有效
`3`	挂起状态	任务仍被识别为已注册，但当前不满足对应的就绪判断
`4`	前一任务	任务编号与 `OS_PREV_TASK` 相同
`-1`	编号无效	任务编号超出接口允许范围

上述状态是查询函数根据内核变量计算得出的结果，不应简单视为所有任务共享的独立状态字段。

特别是，返回值 `4` 表示任务编号与 `OS_PREV_TASK` 相同，并不表示任务发生了超时。返回值 `3` 也不一定能区分所有可能的等待原因。具体状态判断顺序以当前实现为准。

4.6 优先级说明

标准任务的自然优先级范围为 `0` 至 `7`，高速任务使用固定的优先级编码 `8`。

优先级	适用任务	说明
`0` 至 `7`	标准任务	通过 `OS_PROCESS_OK[]` 中的优先级位保存
`8`	高速任务	由高速任务管理机制处理

`os_task_set_priority()` 可以在标准优先级范围内修改标准任务优先级，也可以在满足条件时将任务从标准任务类型转换为高速任务类型，或从高速任务类型转换为标准任务类型。

标准任务与高速任务之间的转换涉及任务编号、堆栈指针和上下文数据的迁移，因此不能将其简单理解为修改一个优先级字段。

4.7 宏定义

本模块没有单独定义公开宏。实现中使用的任务数量、任务状态、优先级上限、调度标志和上下文相关标识符由 HRTOS 内核及配置模块提供。

4.8 API 函数列表

函数	功能
`os_task_suspend()`	挂起指定任务
`os_task_get_state()`	查询指定任务状态
`os_task_set_next()`	设置下一次调度目标
`os_task_resume()`	恢复指定任务
`os_task_exit()`	请求当前任务退出
`os_task_delete()`	请求删除指定任务
`os_task_set_priority()`	设置任务优先级或转换任务类型
`os_task_get_priority()`	查询任务优先级
`os_task_is_valid()`	查询任务注册状态
`os_task_self()`	获取当前任务编号
`os_task_yield()`	主动请求调度
`os_task_suspend_self()`	挂起当前任务

函数	功能
`os_lock()`	锁定任务注册配置
`os_unlock()`	解除任务注册配置锁
`os_lock_query()`	查询任务注册配置锁状态

4.9 API 详细说明

4.9.1 os_task_suspend

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	要挂起的任务编号

返回值

- `1`：函数完成挂起标志设置并请求调度。
- `-1`：任务编号超出允许范围。

功能说明

该函数根据任务类型修改相应的就绪或运行标志。

对于标准任务，函数通过清除 `OS_PROCESS_OK[id]` 的最低位，将其从标准任务的就绪判断中移除。

对于高速任务，函数根据任务编号清除对应高速任务的运行标志。

函数随后设置 `OS_SCHED_REASON` 并触发 Timer0 中断，请求系统重新进行调度。

使用示例

```
char result;

result = os_task_suspend(1);

if (result == 1)
{
    /* 已提交挂起操作 */
}
```

注意事项

- 当前实现没有检查任务是否已注册，也没有排除任务编号 `0`。
- 函数没有在内部使用 `EA = 0` 保护状态修改。

- 返回 `1` 表示相关标志已修改并触发调度请求，不等同于目标任务的所有资源已经完成处理。
- 如果需要挂起当前任务，建议使用 `os_task_suspend_self()`，以明确表达操作对象。

4.9.2 os_task_get_state

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	要查询的任务编号

返回值

- `0`：未注册或已删除。
- `1`：当前运行任务。
- `2`：就绪状态。
- `3`：挂起状态。
- `4`：前一任务。
- `-1`：任务编号超出允许范围。

功能说明

函数分别针对标准任务和高速任务进行判断。

对于标准任务，判断顺序为：

- 如果 `OS_PROCESS_OK[id] == 0`，返回 `0`。
- 如果任务编号等于 `OS_CURRENT_TASK`，返回 `1`。
- 如果任务编号等于 `OS_PREV_TASK`，返回 `4`。
- 如果 `OS_PROCESS_OK[id]` 的最低位为 `1`，返回 `2`。
- 其他情况返回 `3`。

对于高速任务，函数优先检查当前任务编号和前一任务编号，然后根据高速任务上下文标记及对应运行标志返回状态。

使用示例

```
char state;

state = os_task_get_state(1);

if (state == 2)
{
    /* 查询结果为就绪状态 */
}
```

注意事项

- 该函数是基于内核状态变量进行分类，并非所有状态都直接来自 `OS_TASK[id].state`。
- 返回值 `4` 是前一任务标识，不是超时状态。
- 判断结果受当前任务编号、前一任务编号和任务类型影响。
- 需要判断任务是否注册时，应结合 `os_task_is_valid()` 使用，而不是只依赖状态值。

4.9.3 os_task_set_next

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	指定的下一次调度目标编号

返回值

- `1`：成功设置下一次调度目标。
- `-1`：编号无效、目标任务不满足就绪判断，或当前调度请求槽位已被占用。

功能说明

该函数用于指定下一次调度目标，而不是直接执行完整的任务上下文切换。

函数首先检查任务编号范围及目标任务的就绪条件。标准任务通过 `OS_PROCESS_OK[id]` 的最低位判断；高速任务则分别检查对应的高速任务运行标志。

如果 `OS_JINCHENG_CUNCHU` 为 `0`，函数会：

- 将 `OS_JINCHENG_CUNCHU` 设置为 `1`。
- 将当前任务编号保存到 `OS_PREV_TASK`。
- 将目标编号写入 `OS_CURRENT_TASK`。
- 设置 `OS_SCHED_REASON`。
- 触发 Timer0 中断。

如果当前调度请求槽位已被占用，函数返回 `-1`。

使用示例

```
char result;

result = os_task_set_next(1);

if (result == 1)
{
    /* 下一次调度目标已设置 */
}
```

注意事项

- 该接口会改变内核记录的当前任务和前一任务信息，并影响后续调度路径，应谨慎使用。

- 目标任务必须满足对应的就绪判断。
- 当前实现没有在该函数中检查 `OS_JINCHENG_CUNCHU` 之外的其他调度冲突条件。
- 不应将返回 `1` 理解为目标任务已经开始执行；函数只是设置调度目标并请求调度。
- 该接口不适合代替常规的任务挂起、恢复或优先级调度机制。

4.9.4 os_task_resume

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	要恢复的任务编号

返回值

- `1`：恢复标志设置成功。
- `-1`：编号无效，或目标任务不存在。

功能说明

该函数根据任务类型恢复相应的就绪或运行标志。

对于标准任务，如果 `OS_PROCESS_OK[id] == 0`，函数返回 `-1`；否则设置最低位，使任务重新满足标准任务的就绪判断。

对于高速任务，函数先检查对应的 `OS_KUAI_SP[]` 是否为零。如果为零，则认为任务不存在并返回 `-1`；否则设置对应的高速任务运行标志。

成功后，函数设置 `OS_SCHED_REASON` 并触发 Timer0 中断。

使用示例

```
char result;

result = os_task_resume(1);

if (result == 1)
{
    /* 已请求恢复任务 */
}
```

注意事项

- 该函数没有检查目标任务是否此前确实处于挂起状态。
- 标准任务的判断条件是 `OS_PROCESS_OK[id] == 0`，不只是检查一个独立的挂起状态字段。
- 函数返回成功表示相关标志已更新并请求调度，不保证任务立即运行。
- 恢复操作不等同于重新创建任务。

4.9.5 os_task_exit

函数原型

```
void os_task_exit(void);
```

参数说明

无。

返回值

无。

功能说明

该函数用于请求当前任务退出。

函数读取 `OS_CURRENT_TASK`，然后根据任务类型修改对应的调度标志。

对于标准任务，函数清除 `OS_PROCESS_OK[id]` 的最低位，将 `OS_TASK[id].state` 设置为 `DEAD`，并清除 `wait_tick`。

对于高速任务，函数清除对应的高速任务运行标志、等待相关标志，并设置相应的删除标志。

随后，函数设置 `OS_SCHED_REASON`，恢复中断使能，并通过设置 `TF0 = 1` 请求 Timer0 中断处理后续调度。

使用示例

```
void worker_task(void)
{
    /* 执行任务工作 */

    if (/* 退出条件成立 */)
    {
        os_task_exit();
    }

    /* 退出路径之后不应依赖当前任务继续执行 */
}
```

注意事项

- 函数没有返回值，也没有通过普通返回值报告退出操作是否成功。
- 当前实现标记任务状态并触发调度，实际资源清理由系统后续流程处理。
- 不应将其描述为函数内部立即完成所有堆栈及上下文资源释放。
- 当前实现没有检查当前任务是否为任务 `0`。调用方应避免对系统空闲任务执行不符合设计预期的退出操作。
- 该函数会直接将 `EA` 置为 `0` 后再置为 `1`，不保留调用前的中断使能状态。

4.9.6 os_task_delete

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	要删除的任务编号

返回值

- `1`：删除标志设置成功。
- `-1`：任务编号无效，或尝试删除任务 `0`。

功能说明

该函数用于请求删除指定任务。

函数首先拒绝任务编号 `0`，并检查编号是否超过标准任务和高速任务所允许的范围。

对于高速任务，函数清除对应的运行标志和等待相关标志，并设置相应的高速任务删除标志。

对于标准任务，函数清除 `OS_PROCESS_OK[id]` 的最低位，将任务状态设置为 `DEAD`，并清除 `wait_tick`。

操作完成后，函数恢复中断使能并返回 `1`。

使用示例

```
char result;

result = os_task_delete(1);

if (result == 1)
{
    /* 删除请求已提交 */
}
```

注意事项

- 该函数并未在自身实现中完成全部资源释放，而是设置删除相关标志，供后续清理流程处理。
- 返回 `1` 不等于任务上下文、堆栈及相关资源已经全部释放。
- 任务 `0` 不允许通过该接口删除。
- 当前实现没有明确拒绝删除正在运行的其他任务，也没有单独检查目标任务是否已注册。
- 函数通过 `EA = 0` 和 `EA = 1` 处理临界区，但没有恢复调用前的中断使能状态。

4.9.7 os_task_set_priority

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	目标任务编号
`pr`	`unsigned char`	目标优先级，标准任务为 `0` 至 `7`，高速任务为 `8`

返回值

- `0`：标准任务优先级修改成功，或高速任务保持高速类型且无需转换。
- 正数：任务类型转换成功，并返回转换后分配的任务编号。
- `-1`：参数无效、任务状态不允许修改，或目标任务类型转换时未能分配所需槽位。

功能说明

该函数在临界区中检查任务编号、优先级范围、任务删除状态和任务类型相关条件。

对于标准任务：

- 当目标优先级处于标准范围时，函数修改 `OS_PROCESS_OK[id]` 中的优先级位，设置调度请求并触发 Timer0 中断，返回 `0`。
- 当目标优先级为高速任务优先级时，函数尝试分配一个可用的高速任务槽位，并迁移任务上下文、堆栈指针及相关状态。成功后返回新的高速任务编号。

对于高速任务：

- 当目标优先级处于标准范围时，函数尝试申请一个可用的标准任务编号，将高速任务上下文和堆栈信息迁移到标准任务结构中。成功后返回新的标准任务编号。
- 当目标优先级仍为高速任务优先级时，函数返回 `0`，不执行任务类型转换。

任务类型转换需要可用的目标任务槽位，因此可能因资源不足而失败。

使用示例

```
char result;

result = os_task_set_priority(1, 5);

if (result == 0)
{
    /* 标准任务优先级修改成功 */
}
else if (result > 0)
{
    /* 任务类型转换成功，result 为新任务编号 */
}
else
{
```

```
    /* 修改失败 */
}
```

注意事项

- 优先级参数的有效范围由 `OS_SCHED_LEVEL_MAX` 决定；按当前实现，允许的最高输入值为 `OS_SCHED_LEVEL_MAX - 2`。
- 标准任务优先级直接修改与标准任务、高速任务之间的类型转换是不同操作。
- 类型转换涉及任务编号和上下文迁移，调用方必须根据返回值更新对任务编号的引用。
- 任务 `0` 不允许修改优先级。
- 当前实现对高速任务还有额外的上下文标记和等待标志检查，因此并非任何高速任务都能在任意状态下修改优先级。
- 该函数直接使用 `EA = 0` 和 `EA = 1` 管理临界区，不会恢复调用前的中断使能状态。

4.9.8 os_task_get_priority

函数原型

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

参数说明

参数	类型	说明
`id`	`unsigned char`	要查询的任务编号

返回值

- 标准优先级：通常为 `0` 至 `7`。
- 高速优先级：当前实现返回 `OS_SCHED_LEVEL_MAX - 2`。
- `-1`：任务编号超出允许范围。

功能说明

该函数通过任务编号判断任务类型。

对于满足 `id < OS_PROCESS_MAX - 1` 的任务，函数从 `OS_PROCESS_OK[id]` 中提取优先级位并返回。

对于其他合法编号，函数直接返回 `OS_SCHED_LEVEL_MAX - 2`，即高速任务优先级编码。

使用示例

```
char prio;

prio = os_task_get_priority(1);
```

注意事项

- 当前实现的标准任务判断条件是 `id < OS_PROCESS_MAX - 1`，而不是 `id < OS_PROCESS_MAX`。
- 因此，编号为 `OS_PROCESS_MAX - 1` 的最后一个标准任务不会按标准任务路径提取优先级，而会进入高速任务优先级返回路径。

- 这是当前实现中需要重点核对的边界条件。正式文档不应掩盖这一实现差异，修正源码后应同步更新接口说明。
- 当前函数不会检查任务是否已注册或处于有效状态。

4.9.9 os_task_is_valid

函数原型

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

参数说明

参数	类型	说明
`id`	`u8`	要检查的任务编号

返回值

- `1`：任务被识别为已注册。
- `0`：任务未注册。
- `-1`：任务编号超出允许范围。

功能说明

该函数检查指定编号对应的任务注册标记。

对于标准任务，函数通过 `OS_PROCESS_OK[id]` 是否非零判断注册状态。

对于两个高速任务，函数分别检查 `OS_SP_KUAI_BEI[]` 中对应的上下文标记。

使用示例

```
signed char valid;

valid = os_task_is_valid(1);

if (valid == 1)
{
    /* 任务被识别为已注册 */
}
```

注意事项

- 对标准任务而言，只要 `OS_PROCESS_OK[id]` 非零，函数就返回 `1`；它并不要求任务当前处于就绪状态。
- 该接口检查的是实现定义的注册标记，不等同于全面验证任务当前能否运行。
- 需要判断任务是否就绪时，应使用 `os_task_get_state()` 等相关接口辅助判断。

4.9.10 os_task_self

函数原型

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

参数说明

无。

返回值

当前内核记录的任务编号，即 `OS_CURRENT_TASK`。

功能说明

函数直接返回 `OS_CURRENT_TASK`，用于获取系统当前记录的运行任务编号。

使用示例

```
u8 my_id;

my_id = os_task_self();
```

注意事项

- 函数只返回当前任务编号，不检查该编号对应的任务是否仍然有效。
- 在任务编号发生变化或执行任务类型转换的场景中，应结合具体操作的返回值处理任务编号。

4.9.11 os_task_yield

函数原型

```
void os_task_yield(void);
```

参数说明

无。

返回值

无。

功能说明

函数设置 `OS_SCHED_REASON = 1`，然后通过 `TF0 = 1` 请求 Timer0 中断，以触发系统调度。

使用示例

```
void worker_task(void)
{
    while (1)
    {
        /* 执行一段工作 */

        os_task_yield();
    }
}
```

注意事项

- 当前实现只设置调度请求并触发 Timer0 中断，没有直接修改标准任务的就绪位。

- 该函数不保证调用后一定切换到另一个任务；最终运行结果取决于调度器的处理逻辑。
- 不能仅凭函数名称认定其一定会重置时间片或强制切换任务。

4.9.12 os_task_suspend_self

函数原型

```
void os_task_suspend_self(void);
```

参数说明

无。

返回值

无。

功能说明

函数读取 `OS_CURRENT_TASK`，根据当前任务类型清除相应的就绪或运行标志，然后设置调度请求并触发 Timer0 中断。

对于标准任务，函数清除 `OS_PROCESS_OK[id]` 的最低位。

对于高速任务，函数清除对应的高速任务运行标志。

使用示例

```
void worker_task(void)
{
    /* 执行任务工作 */

    os_task_suspend_self();

    /* 后续代码是否执行取决于任务何时恢复并再次运行 */
}
```

注意事项

- 该函数不会直接清除任务注册标记，也不会执行任务删除。
- 恢复任务需要其他执行路径调用 `os_task_resume()`，并且目标任务必须仍满足恢复条件。
- 当前实现没有在函数内使用 `EA = 0` 保护相关标志修改。
- 函数不保证立即切换完成，实际切换由后续调度处理执行。

4.9.13 os_lock

函数原型

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

参数说明

无。

返回值

无。

功能说明

函数将 `OS_CFG_LOCK` 设置为 `1`，表示系统任务注册配置进入锁定状态。

使用示例

```
os_lock();

/* 此处进行需要保持注册配置锁定的操作 */
```

注意事项

- 该接口修改的是任务注册配置锁，不是用于保护普通共享变量的通用互斥锁。
- 当前函数本身只设置配置锁标志。
- 具体哪些任务注册接口会检查该标志，需要结合相应的任务注册实现确认。

4.9.14 os_unlock

函数原型

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

参数说明

无。

返回值

无。

功能说明

函数将 `OS_CFG_LOCK` 设置为 `0`，解除任务注册配置锁定状态。

使用示例

```
os_unlock();
```

注意事项

- 该接口只修改注册配置锁标志。
- 它不负责恢复任务状态，也不负责解锁互斥锁或其他同步资源。
- 如果系统要求运行期间禁止继续注册任务，应结合注册接口对 `OS_CFG_LOCK` 的检查逻辑使用。

4.9.15 os_lock_query

函数原型

```
char os_lock_query(void);
```

参数说明

无。

返回值

- `1`：注册配置处于锁定状态。
- `0`：注册配置未锁定。

功能说明

函数读取 `OS_CFG_LOCK`，根据其当前值返回锁状态。

使用示例

```
char locked;

locked = os_lock_query();

if (locked == 1)
{
    /* 注册配置已锁定 */
}
```

注意事项

- 查询结果反映的是系统注册配置锁状态。
- 它不代表 CPU 中断已关闭，也不代表任务调度已停止。
- 该接口不会改变锁状态。

4.10 使用建议

- **区分状态、注册与就绪。** 任务已注册并不意味着它当前可运行；应根据操作目的选择状态查询或注册检查接口。
- **谨慎使用指定下一次调度目标的接口。** `os_task_set_next()` 会直接更新内核记录的当前任务和前一任务编号，使用前应确认目标任务状态及调度上下文。
- **正确处理删除流程。** `os_task_delete()` 和 `os_task_exit()` 会设置删除相关标志，实际资源清理由后续内核流程完成。
- **修改优先级后更新任务编号引用。** 标准任务与高速任务之间转换时，返回值可能是新的任务编号。
- **不要将注册配置锁当作通用互斥锁。** `os_lock()` 和 `os_unlock()` 操作的是 `OS_CFG_LOCK`。
- **注意任务编号边界。** 尤其是最后一个标准任务与两个高速任务的编号边界，需与当前源码及配置保持一致。
- **避免依赖未经确认的状态语义。** `os_task_get_state()` 的返回值是通过多项内核变量计算得到的，使用时应以实际实现为准。

Time

时间管理模块

5.1 模块简介

HRTOS 时间管理模块提供系统 Tick 计数、任务延时、时间单位转换、系统运行时间查询以及 Timer0 定时参数配置等功能。

模块通过 Timer0 维护系统时间计数，并通过 `os_wait()` 将任务延时接入内核等待机制。对于短时间、必须持续占用处理器的硬件时序等待，模块还提供忙等待函数 `os_delay_ms()`。

时间管理模块主要包含两种延时方式：

- **忙等待延时**：持续占用 CPU，不主动让出处理器，适用于短时间硬件时序控制。
- **任务延时**：通过内核等待机制等待指定数量的 Tick，适用于任务中的常规延时。

5.2 设计目标

- 提供系统 Tick 累计计数。
- 支持基于 Tick 的任务延时。
- 提供毫秒、秒与 Tick 数之间的换算接口。
- 支持系统累计运行时间查询。
- 支持配置系统 Tick 周期相关参数。
- 为硬件时序控制和任务级时间管理提供不同的接口。

5.3 适用场景

- 任务周期性执行与延时等待。
- 系统运行时间统计。
- 基于 Tick 的时间参数换算。
- 外设初始化、短脉冲控制等硬件时序等待。
- 系统定时参数配置与时间行为测试。

5.4 时间单位与配置

本模块使用系统配置和内部状态维护时间参数。常见配置项包括：

配置项	说明
`OS_TIME_ONCE_DEFAULT`	系统时间片相关的默认配置值，具体作用以系统配置及调度实现为准
`OS_TIME_T0`	Timer0 定时参数相关配置项，具体含义以系统配置和初始化代码为准
`OS_TIME_ONCE`	当前配置的时间片计数参数，由 `os_tick_config()` 设置
`OS_T0_TH0`、`OS_T0_TL0`	用于计算 Tick 时间的 Timer0 重载值相关变量
`OS_TICK_COUNT_H`、`OS_TICK_COUNT_L`	系统累计 Tick 计数的高、低 16 位

需要注意，源码中的 `tick_value` 参数会经过重载值转换。调用者传入的值不是直接写入 Timer0 的最终重载值，而是先计算：

```
65536 - tick_value
```

因此，理解配置参数时应区分传入的计数值与最终写入的 Timer0 重载值。

5.5 API 函数列表

函数	功能
`os_delay_ms()`	毫秒级忙等待延时
`os_ms_to_tick()`	毫秒转换为 Tick 数
`os_sec_to_tick()`	秒转换为 Tick 数
`os_tick_config()`	配置系统 Tick 参数
`os_tick_get()`	读取系统累计 Tick 计数
`os_uptime_ms()`	获取系统累计运行时间的毫秒估算值
`os_delay()`	通过内核等待机制延时指定 Tick 数

5.6 API 详细说明

5.6.1 os_delay_ms

函数原型

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

功能说明

执行毫秒级忙等待。函数通过循环消耗处理器执行时间，不调用内核等待接口，也不主动请求任务调度。

参数

参数	说明
`ms`	忙等待循环次数对应的延时参数，类型为 `unsigned char`

返回值

无。

调用示例

```
os_delay_ms(10);
```

使用注意

- 函数执行期间持续占用 CPU。
- 不会主动释放处理器供其他任务运行。
- 实际延时受 CPU 时钟、指令执行时间、编译器优化及中断干扰等因素影响。
- 参数表示函数内部循环次数所对应的延时量，不应将其视为跨时钟频率、跨编译配置都严格准确的毫秒计时。
- 适用于短时间硬件时序等待，不适合替代任务延时接口。

5.6.2 os_ms_to_tick

函数原型

```
u16 os_ms_to_tick(u16 ms);
```

功能说明

将输入的毫秒数换算为系统 Tick 数。

函数根据 `OS_T0_TH0` 和 `OS_T0_TL0` 计算当前 Tick 对应的计数值，再将其除以 1000，得到以整数毫秒表示的 Tick 时间。若计算结果小于 1 毫秒，则按 1 毫秒进行换算。

返回值使用向上取整的整数除法：

```
(ms + tick_ms - 1) / tick_ms
```

参数

参数	说明
`ms`	待换算的毫秒数

返回值

返回换算得到的 Tick 数。

调用示例

```
u16 ticks = os_ms_to_tick(100);
```

使用注意

- 换算结果依赖当前 Timer0 重载值。
- 函数使用整数运算，Tick 时间先换算为整数毫秒，存在精度损失。
- 该函数不是基于完整硬件时钟模型的高精度时间换算工具。
- 当输入值与中间计算结果接近 `u16` 的取值上限时，应注意整数运算溢出的可能性。

5.6.3 os_sec_to_tick

函数原型

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

功能说明

将输入的秒数转换为系统 Tick 数。

函数首先根据 Timer0 重载值计算单个 Tick 对应的整数毫秒数，再通过 `1000 / tick_ms` 计算每秒对应的 Tick 数，最后乘以输入秒数。

参数

参数	说明
`sec`	待换算的秒数，类型为 `u8`

返回值

返回换算得到的 Tick 数。

调用示例

```
u16 ticks = os_sec_to_tick(5);
```

使用注意

- 换算结果依赖当前 Timer0 配置。
- 计算过程使用整数除法，可能产生精度损失。
- 当 Tick 周期不能整除 1000 毫秒时，计算得到的每秒 Tick 数可能与实际时间存在偏差。
- `sec` 的取值范围受 `u8` 类型限制，返回结果还应考虑 `u16` 的表示范围。

5.6.4 os_tick_config

函数原型

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

功能说明

配置系统时间片相关参数及 Timer0 重载值，并将系统累计 Tick 计数清零。

函数执行过程如下：

- 检查 `slice_count` 是否为 0；若为 0，返回 `-1`。
- 将 `slice_count` 写入 `OS_TIME_ONCE`。
- 检查 `tick_value` 是否大于 1000；若不满足，返回 `-1`。
- 清零 `OS_TICK_COUNT_H` 和 `OS_TICK_COUNT_L`。
- 根据 `65536 - tick_value` 计算重载值，并写入 `OS_T0_TH0`、`OS_T0_TL0`。
- 配置成功时返回 `0`。

参数

参数	说明
`slice_count`	写入 `OS_TIME_ONCE` 的时间片相关计数参数，不能为 0
`tick_value`	用于计算 Timer0 重载值的计数参数，必须大于 1000

返回值

返回值	含义
`0`	配置成功
`-1`	参数检查失败

调用示例

```
char result = os_tick_config(5, 10000);

if (result == 0)
{
    /* 配置成功 */
}
```

使用注意

- `tick_value` 必须大于 1000，且受 `u16` 类型范围限制。
- 该函数写入 `OS_TIME_ONCE` 的操作发生在 `tick_value` 合法性检查之前。因此，如果 `slice_count` 合法而 `tick_value` 不合法，函数虽然返回 `-1`，`OS_TIME_ONCE` 仍然会被修改。
- 配置成功时会清零累计 Tick 计数，调用后系统累计时间计数不再保留原值。
- 函数没有在内部停止 Timer0，也没有对整个配置过程进行显式的中断保护。应结合系统初始化顺序和实际运行状态选择调用时机。
- 参数的实际计时意义取决于 Timer0 工作模式、计数时钟及相关初始化配置。不能仅凭此函数保证实际时间精度。

5.6.5 os_tick_get

函数原型

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

功能说明

读取系统累计 Tick 计数，并分别通过两个指针返回高 16 位和低 16 位。

组合方式如下：

```
u32 total_ticks = ((u32)high_value << 16) | low_value;
```

参数

参数	说明
`high`	接收系统 Tick 计数高 16 位的指针
`low`	接收系统 Tick 计数低 16 位的指针

返回值

无。

调用示例

```
u16 high;
u16 low;

os_tick_get(&high, &low);

u32 total_ticks = ((u32)high << 16) | low;
```

使用注意

- 两个计数值分别保存在 `OS_TICK_COUNT_H` 和 `OS_TICK_COUNT_L` 中。
- 函数在读取时关闭 Timer0 中断使能 `ET0`，读取完成后将 `ET0` 设置为 1。
- 当前实现没有保存并恢复调用前的 `ET0` 状态。如果调用前 Timer0 中断已关闭，函数返回后仍会将其打开。
- 函数没有检查输出指针是否为空，调用者应传入有效的指针。
- 该函数只读取累计 Tick 计数，不直接返回毫秒或微秒。

5.6.6 os_uptime_ms

函数原型

```
u32 os_uptime_ms(void);
```

功能说明

根据系统累计 Tick 计数和当前 Timer0 重载值，估算系统累计运行时间，并以毫秒为单位返回。

函数首先读取高、低 16 位 Tick 计数，组合为 32 位累计计数；随后根据 Timer0 重载值计算单个 Tick 对应的整数毫秒数，最后将累计 Tick 数乘以该整数毫秒数。

核心计算关系为：

```
total_tick = ((u32)tick_h << 16) | tick_l;

tick_us = 65536 - tick_value;
tick_ms = tick_us / 1000;

if (tick_ms == 0)
{
    tick_ms = 1;
}

uptime_ms = total_tick * tick_ms;
```

以上为计算逻辑的示意表达，实际执行过程以函数源码为准。

参数

无。

返回值

返回 `u32` 类型的系统累计运行时间估算值，单位为毫秒。

调用示例

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

使用注意

- 返回值基于累计 Tick 数估算，不是直接读取独立的毫秒硬件计时器。
- 计算时使用整数除法；当单个 Tick 不能整除 1000 微秒时，会产生精度损失。
- 当计算出的 Tick 毫秒数为 0 时，函数按 1 毫秒处理。
- 读取 Tick 计数时，函数关闭 `ET0`，随后将 `ET0` 设置为 1，没有恢复调用前的中断使能状态。
- Tick 计数读取与 Timer0 重载值读取不是一个完整的受保护操作。如果运行期间重新配置定时参数，计数与换算参数可能不一致。
- `u32` 返回值具有有限范围，长时间运行时需要考虑计数回绕风险。
- 当前实现没有使用浮点运算。

5.6.7 os_delay

函数原型

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

功能说明

让当前任务通过内核等待机制延时指定数量的 Tick。

当 `tick` 为 0 时，函数直接返回。当 `tick` 不为 0 时，函数先检查当前任务 ID 是否小于 `OS_PROCESS_MAX + 2`。满足条件时，设置 Timer0 计数寄存器 `TL0` 和 `TH0`，随后调用：

```
os_wait(WAIT_DELAY, OS_INVALID_ID, tick);
```

由内核等待机制处理当前任务的延时等待。

参数

参数	说明
`tick`	请求延时的系统 Tick 数

返回值

无。

调用示例

```
os_delay(100);
```

使用注意

- `tick == 0` 时不会进入等待流程。
- 对于当前任务 ID 小于 `OS_PROCESS_MAX + 2` 的情况，函数会调用内核等待接口。
- 如果当前任务 ID 不满足检查条件，函数不会调用 `os_wait()`。
- 函数会直接修改 `TL0` 和 `TH0`，因此延时调用与 Timer0 计时行为之间的具体关系需要结合系统定时器中断代码确认。
- 实际等待时间取决于系统 Tick 配置、内核等待机制以及调度行为。
- 该函数的意图是执行任务级延时，而不是忙等待。其是否释放处理器、何时重新进入就绪状态，以及高速任务的具体处理方式，应以 `os_wait()` 和对应的唤醒实现为准。

5.7 使用建议

5.7.1 选择合适的延时接口

使用需求	建议接口	说明
短时间硬件时序等待	`os_delay_ms()`	忙等待，占用 CPU
按 Tick 延时当前任务	`os_delay()`	通过内核等待机制处理
毫秒参数转换	`os_ms_to_tick()`	返回估算的 Tick 数
秒参数转换	`os_sec_to_tick()`	返回估算的 Tick 数
读取累计 Tick	`os_tick_get()`	获取高、低 16 位
查询累计运行时间	`os_uptime_ms()`	返回基于 Tick 的毫秒估算值
配置系统定时参数	`os_tick_config()`	修改时间片相关参数及 Timer0 重载值

5.7.2 配置与精度注意事项

时间管理模块的多个接口使用 Timer0 重载值进行整数换算。由于代码采用整数除法，转换结果和累计运行时间均可能存在量化误差。

如果需要更高的时间精度，应结合实际晶振频率、Timer0 工作模式、定时器中断实现和调度开销进行验证，不应将本模块返回的毫秒数直接视为严格精确的墙钟时间。

在系统运行期间修改 Tick 配置时，还应确认累计计数是否需要保留、Timer0 是否正在运行，以及其他时间相关模块是否依赖原有配置。

5.8 使用说明

- **系统 Tick**：时间接口的换算精度与实际行为取决于系统 Tick 配置及定时器运行情况。
- **忙等待延时**：`os_delay_ms()` 通过软件循环实现延时，会持续占用 CPU，实际延时受处理器运行频率、编译器优化及中断执行等因素影响。
- **时间单位转换**：`os_ms_to_tick()` 和 `os_sec_to_tick()` 根据系统 Tick 配置进行整数换算，转换结果可能存在取整误差。
- **Tick 配置**：调用 `os_tick_config()` 时，应确保传入的参数符合接口规定的有效范围。
- **中断状态**：使用 `os_tick_get()` 和 `os_uptime_ms()` 时，应注意接口对 Timer0 中断使能状态的处理，避免与应用程序自身的中断管理逻辑产生冲突。
- **任务延时**：`os_delay()` 通过内核等待机制实现基于 Tick 的任务延时，其实际行为取决于系统定时器、内核等待与任务唤醒机制的协同工作。

Wait

统一等待模块

6.1 模块简介

HRTOS 统一等待模块为任务延时和资源同步提供统一的等待状态管理机制。模块包含等待入口、资源对象初始化和任务唤醒接口，负责记录任务的等待类型、等待对象、等待计时参数及唤醒结果，并通过调度触发机制请求内核重新调度。

模块主要由以下接口构成：

- `os_wait()`：统一等待入口，处理延时等待和关联资源对象的等待。
- `os_res_init()`：初始化统一资源对象的基础状态。
- `wake_task()`：清理任务等待信息，并在允许唤醒时恢复其就绪资格。

标准任务和高速任务的延时处理路径有所不同。标准任务通过任务控制块记录等待状态；高速任务使用独立的运行标志和等待计数，不进入标准任务的统一等待处理路径。

6.2 设计目标

- 统一管理任务的等待状态与等待上下文。
- 为延时、信号量、互斥锁、消息及事件等机制提供公共等待基础。
- 使用资源等待位图记录关联资源的等待任务。
- 保存等待结果，使任务唤醒后能够获取相应状态。
- 通过调度请求与 Timer0 中断触发机制衔接内核调度。

6.3 适用场景

- 任务延时与超时等待。
- 信号量、互斥锁等同步对象的等待。
- 消息队列、邮箱及事件相关的等待。
- 任务等待状态的统一维护。
- 资源释放或其他事件发生后的任务唤醒。

具体资源机制如何选择等待者、如何判断资源条件满足，以及超时由何处检测，需要结合对应资源模块和系统定时处理代码理解。

6.4 核心数据结构

6.4.1 OS_RESOURCE

`os_res_init()` 接收 `OS_RESOURCE` 类型的指针，并初始化该对象中的以下字段：

成员	初始化值	用途
`value`	`0`	资源相关数值
`owner`	`OS_INVALID_ID`	当前拥有者标识
`wait_cnt`	`0`	等待任务计数
`wait_mask`	`0`	等待任务位图

当前提供的 `os_res_init()` 实现没有初始化 `pending_signal` 字段。若该字段属于实际使用的资源结构体，初始化时是否需要清零，应结合对应中断信号处理逻辑确认。

资源结构体的完整定义应以 `hrtos.h` 或项目实际配置头文件为准，不应仅根据本模块的函数实现推断全部成员。

6.4.2 任务等待上下文

对于标准任务，`os_wait()` 通过 `OS_TASK[tid]` 保存等待信息：

字段	用途
`wait_type`	当前等待类型
`wait_obj`	关联的等待对象编号
`wait_tick`	等待计时参数
`wait_flag`	等待结果
`state`	任务状态

当任务进入等待时，函数将任务状态设为 `WAIT`，并清除 `OS_PROCESS_OK[tid]` 的最低位，使该任务不再具有正常调度资格。

任务被成功唤醒时，`wake_task()` 清理等待上下文、将状态设置为 `READY`，并重新置位相应的调度资格位。

6.5 等待类型与返回标志

本模块通过 `type` 参数区分等待类型。项目中的等待类型名称包括：

名称	含义
`WAIT_NONE`	当前没有等待
`WAIT_DELAY`	延时等待

名称	含义
WAIT_SEM	信号量等待
WAIT_MUTEX	互斥锁等待
WAIT_MSG_SEND	消息发送等待
WAIT_MSG_RECV	消息接收等待
WAIT_EVENT	事件等待
WAIT_MAILBOX	邮箱等待

上述名称及其具体数值应以项目实际配置为准。

os_wait() 的返回值和 wake_task() 的 flag 参数使用项目定义的等待结果标志。常见名称包括：

名称	含义
WAIT_TIMEOUT	超时结果
WAIT_SIGNAL	信号或正常唤醒结果

这些标志的具体数值应以项目头文件中的定义为准。

6.6 API 函数列表

函数	功能
os_wait()	统一等待入口
os_res_init()	初始化资源对象
wake_task()	清理等待上下文并尝试唤醒任务

6.7 API 详细说明

6.7.1 os_wait

函数原型

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

功能说明

使当前任务进入指定的等待流程。函数根据等待类型执行不同处理：延时等待使用专门路径；其他等待类型会检查当前任务是否已有等待状态，并验证资源对象编号，然后保存等待信息、更新资源等待位图，并请求调度。

参数

参数	类型	说明
`type`	`u8`	等待类型
`obj`	`u8`	等待对象编号；纯延时等待不依赖资源对象
`tick`	`u16`	等待计时参数，单位为系统 Tick

返回值

返回 `u8` 类型的结果。具体返回行为如下：

情况	返回行为
延时等待且 `tick == 0`	返回 `WAIT_TIMEOUT`
高速任务执行非零延时等待	设置相应高速任务运行标志和等待计数后，返回 `WAIT_TIMEOUT`
当前任务已有等待状态，且本次不是延时等待	返回 `0`
非延时等待的对象编号超出 `OS_RESOURCE_MAX` 范围	返回 `0`
标准任务完成等待状态登记	返回当前 `wait_flag` 字段的值

需要特别注意：标准任务登记等待后，函数在触发调度之前将 `wait_flag` 初始化为 `0`，随后返回该字段。因此，不能仅根据函数原型或注释，就认定所有调用都会在任务实际恢复执行后返回最终的 `WAIT_TIMEOUT` 或 `WAIT_SIGNAL`。

标准任务等待流程

对于标准任务，非零延时和资源等待均会记录任务等待上下文，并清除任务的正常调度资格位。资源等待还会在关联对象有效且不为 `OS_INVALID_ID` 时更新等待位图和等待计数。

主要处理过程如下：

- 保存等待类型、对象、计时参数，并将 `wait_flag` 清零。
- 对关联资源对象设置对应任务的等待位。
- 更新任务状态为 `WAIT`。
- 清除 `OS_PROCESS_OK[tid]` 的最低位。
- 设置 `OS_SCHED_REASON` 并置位 `TF0`，请求系统进入调度流程。

  对于 `WAIT_DELAY`，代码不会将任务加入资源等待位图。

高速任务延时流程

当当前任务 ID 大于或等于 `OS_PROCESS_MAX` 时，函数进入高速任务专用分支。

- ID 等于 `OS_PROCESS_MAX` 时，清除高速任务 A 的运行标志，并将 `tick` 写入 `OS_WAIT_DI2[0]`。
- ID 等于 `OS_PROCESS_MAX + 1` 时，清除高速任务 B 的运行标志，并将 `tick` 写入 `OS_WAIT_DI2[1]`。
- 两种路径均设置调度请求并触发 Timer0 中断，然后返回 `WAIT_TIMEOUT`。

这一路径不会写入标准任务的 `wait_type`、`wait_obj`、`wait_tick` 和 `wait_flag`，也不会进入标准任务的资源等待位图处理流程。

调用示例

```
/* 标准任务延时：具体行为取决于内核等待与唤醒实现 */
os_wait(WAIT_DELAY, OS_INVALID_ID, 10);

/* 等待资源编号 0：必须确保该资源已正确初始化 */
u8 result = os_wait(WAIT_SEM, 0, 10);
```

示例仅展示接口调用方式。资源等待是否成功、何时唤醒以及结果如何解释，应以相应资源模块和等待结果处理代码为准。

使用注意

- 当前实现通过 `EA = 0` 进入临界区，并在多个返回路径中设置 `EA = 1`，没有保存并恢复调用前的全局中断使能状态。
- 对非延时等待，函数会拒绝已经具有非 `WAIT_NONE` 等待类型的任务。
- 对非延时等待，函数会检查 `obj` 是否小于 `OS_RESOURCE_MAX`，但不会在本函数中验证等待类型是否属于已定义类型。
- `WAIT_DELAY` 使用独立路径，不执行相同的重复等待检查。
- 对标准任务，函数返回的是登记等待时读取的 `wait_flag`，而不是通过当前函数实现了一个显式的“挂起后再恢复执行”的同步等待循环。
- 超时扫描、资源条件检查和唤醒者选择并不由这段代码完整实现，需要继续检查对应内核代码。

6.7.2 os_res_init

函数原型

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

功能说明

初始化指定资源对象的基础状态，使其资源数值、拥有者和等待记录回到初始值。

参数

参数	类型	说明
`r`	`OS_RESOURCE *`	待初始化的资源对象指针

返回值

无。

调用示例

```
OS_RESOURCE my_resource;

os_res_init(&my_resource);
```

执行行为

函数依次执行以下操作：

- 将 `value` 设为 `0`。
- 将 `owner` 设为 `OS_INVALID_ID`。
- 将 `wait_cnt` 设为 `0`。
- 将 `wait_mask` 设为 `0`。

使用注意

- 调用前必须确保 `r` 指向有效、可写的资源对象。
- 函数不会检查空指针。
- 当前实现没有初始化 `pending_signal` 等其他可能存在的成员。
- 资源初始化应在资源投入使用前完成，避免清除正在使用中的等待状态。

6.7.3 wake_task

函数原型

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

功能说明

清理指定任务关联的等待记录，并在任务仍符合唤醒条件时设置唤醒结果、恢复就绪状态以及触发调度。

参数

参数	类型	说明
`tid`	`u8`	待处理的任务编号
`flag`	`u8`	写入任务等待结果的标志

返回值

无。

执行流程

函数首先关闭全局中断使能，并读取目标任务的 `wait_obj`。

如果 `wait_obj` 不等于 `OS_INVALID_ID`，函数会：

- 清除对应资源等待位图中属于该任务的位。
- 在 `wait_cnt` 非零时将其减一。

  随后，函数检查目标任务是否已经删除或处于 `DEAD` 状态：

- 如果 `OS_PROCESS_OK[tid] == 0`，则恢复全局中断使能并返回。
- 如果 `OS_TASK[tid].state == DEAD`，则恢复全局中断使能并返回。

  若任务通过上述检查，函数会：

- 将 `flag` 写入 `wait_flag`。
- 将 `wait_type` 设为 `WAIT_NONE`。
- 将 `wait_obj` 设为 `OS_INVALID_ID`。
- 将 `wait_tick` 设为 `0`。
- 将任务状态设为 `READY`。
- 设置 `OS_PROCESS_OK[tid]` 的最低位。
- 设置 `OS_SCHED_REASON`，恢复全局中断使能，并置位 `TF0` 请求调度。

  调用示例

```
/* 示例：由内核或相应同步机制在合适的上下文中调用 */
wake_task(task_id, WAIT_SIGNAL);
```

此示例中的 `task_id` 应是有效的任务编号，调用时机和 `flag` 的含义应与相应等待机制保持一致。

使用注意

- 函数没有在入口检查 `tid` 是否小于任务总数。调用者必须确保任务编号有效，否则访问任务控制块或资源数组可能越界。
- 函数会先尝试清理资源等待记录，然后才检查任务是否已删除或处于 `DEAD` 状态。因此，即使最终不执行唤醒，等待位图和等待计数也可能已经被修改。
- 函数没有验证目标任务当前是否确实处于等待状态。
- `flag` 会被直接写入 `wait_flag`，函数本身不会验证它是否为有效等待结果。
- 函数通过 `EA = 0` 和 `EA = 1` 控制中断使能，不会恢复调用前的中断状态。
- `wake_task()` 负责处理等待上下文和恢复就绪资格，但资源条件是否满足、应唤醒哪个任务等策略应由调用它的资源机制负责。

6.8 使用建议

6.8.1 延时与资源等待分开理解

`WAIT_DELAY` 表示纯时间等待，不需要关联资源等待位图。信号量、互斥锁、消息、邮箱和事件等等待则可能关联资源对象，具体由对应模块传入参数。

6.8.2 不要混淆等待登记与最终唤醒

调用 `os_wait()` 并不意味着所有路径都会同步返回最终的等待结果。标准任务路径先登记等待信息并请求调度；高速任务延时路径则采用独立的运行标志和等待计数。应结合调度器、定时器处理及具体资源模块理解完整执行过程。

6.8.3 保证任务编号和资源编号有效

`wake_task()` 不检查任务编号边界；`os_res_init()` 不检查指针有效性。使用这些接口时，应由调用者确保参数合法，并保证资源对象在整个等待期间有效。

6.8.4 注意中断状态管理

`os_wait()` 和 `wake_task()` 均会直接将全局中断使能设置为关闭或开启状态，没有完整保存和恢复进入函数前的中断状态。若这些函数可能在中断已关闭的上下文中调用，应特别审核相关路径。

6.9 使用说明

- **等待机制**：统一等待模块负责记录任务等待状态并配合唤醒处理，具体的资源获取、释放与同步规则由相应功能模块决定。
- **任务类型**：标准任务与高速任务采用不同的等待处理路径，具体行为应以对应任务类型的接口约定为准。
- **等待与返回值**：`os_wait()` 的返回值应结合任务调度、上下文恢复及调用模块的处理逻辑理解。调用相关接口时，应遵循对应模块规定的等待与唤醒语义。
- **任务唤醒**：调用 `wake_task()` 时，调用方必须确保任务编号有效，并正确处理任务当前状态及其等待关系。
- **资源初始化**：`os_res_init()` 初始化资源对象中由该函数明确设置的字段。其他字段的初始化要求应以资源结构定义及相关模块实现为准。
- **中断状态**：等待与唤醒相关操作应遵循系统的中断管理约定，避免破坏调用前的中断状态或影响内核调度。

Semaphore

信号量模块

7.1 模块简介

HRTOS 信号量模块提供基于计数值的任务同步机制，通过 P/V 操作管理可用资源数量，并支持多个任务等待同一信号量。

信号量基于系统统一资源表 `OS_RES[]` 实现，使用资源槽位中的 `value` 保存计数，使用 `wait_mask` 记录等待任务。释放信号量时，如果存在等待任务，模块会尝试选择一个优先级较高的任务并唤醒；如果没有等待任务，则增加信号量计数。

信号量不使用资源拥有者机制，适用于任务同步、资源数量管理和事件协调等场景。它与互斥锁的主要区别是：信号量关注计数与同步，不以任务所有权作为获取和释放的基本条件。

7.2 设计目标

- 提供任务间的 P/V 同步操作。
- 使用计数值管理可用资源数量。
- 支持多个任务等待同一信号量。
- 在信号量可用时提供直接获取路径。
- 复用 HRTOS 统一资源表和任务等待机制。

7.3 使用场景

- **资源计数管理：** 限制同时使用某类资源的任务数量。
- **任务间同步：** 协调生产者与消费者等不同任务的执行。
- **事件协调：** 通过信号量通知其他任务继续执行。
- **有限资源访问：** 使用计数信号量表示可用资源数量。

需要注意，信号量本身不记录资源由哪个任务获取，也不提供与互斥锁相同的所有权约束。

7.4 数据结构

本模块复用 `hrtos_internal.h` 中定义的 `OS_RESOURCE` 资源结构，并通过 `OS_RES[sid]` 访问指定信号量。

字段	用途
`value`	信号量计数，表示当前记录的可用资源数量
`owner`	兼容统一资源结构，初始化时设为 `OS_INVALID_ID`；信号量逻辑不使用所有权机制
`wait_mask`	等待该资源的任务位图
`wait_cnt`	兼容字段，初始化时清零；当前所提供的信号量实现未使用该字段进行计数管理

信号量编号 `sid` 对应统一资源表中的槽位，因此必须满足：

```
sid < OS_RESOURCE_MAX
```

7.5 宏定义

本模块没有单独定义公开宏。实现中使用的 `OS_RESOURCE_MAX`、`OS_INVALID_ID`、`WAIT_SEM`、`WAIT_SIGNAL`、`OS_PROCESS_MAX` 等标识符由 HRTOS 的其他配置或内核模块提供。

7.6 API 函数列表

函数	功能
`os_sem_init()`	初始化指定信号量
`os_sem_wait()`	等待或获取一个信号量
`os_sem_post()`	释放信号量，或唤醒等待任务

7.7 API 详细说明

7.7.1 os_sem_init

函数原型

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

参数说明

参数	类型	说明
`sid`	`u8`	信号量编号，对应资源表槽

参数	类型	说明
		位
`init_val`	`u8`	信号量初始计数值

返回值

- `1`：初始化成功。
- `-1`：信号量编号超出有效范围。

功能说明

初始化指定的信号量，主要执行以下操作：

- 检查 `sid` 是否小于 `OS_RESOURCE_MAX`。
- 将 `value` 设置为传入的初始计数值。
- 将 `owner` 设置为 `OS_INVALID_ID`。
- 清空 `wait_mask`。
- 将 `wait_cnt` 设为 `0`。

使用示例

```
char result;

result = os_sem_init(0, 2);

if (result == 1)
{
    /* 信号量 0 初始化成功 */
}
```

注意事项

- 初始化会直接覆盖指定资源槽位中的相关字段。
- 实现没有使用 `EA` 进行临界区保护，因此应在合适的初始化阶段调用，避免与正在运行的任务或等待操作并发访问同一资源。
- 不应在仍有任务等待该信号量时随意重新初始化，否则可能破坏现有等待关系。
- 当前实现将 `wait_cnt` 清零，但信号量的等待任务选择主要依赖 `wait_mask`。

7.7.2 os_sem_wait

函数原型

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

参数说明

参数	类型	说明
`sid`	`u8`	信号量编号

返回值

- `0`：信号量编号无效。
- `WAIT_SIGNAL`：计数获取路径返回的成功标识。
- 其他值：可能来自 `os_wait()` 的返回结果，具体含义取决于内核等待机制的实现。

功能说明

该函数用于执行信号量 P 操作。源码首先检查信号量编号，然后进入计数检查流程。

当计数大于零时，函数将计数减一，并返回 `WAIT_SIGNAL`。

当计数为零时，函数会保存当前任务的基础优先级，重新开启中断，再调用：

```
tid = os_wait(WAIT_SEM, sid, 0);
```

等待内核处理任务阻塞和后续唤醒。`WAIT_SEM` 用于标识信号量等待类型，最后一个参数传入 `0`。

从当前实现看，`os_wait()` 返回后，函数还会再次检查信号量计数：如果计数大于零，则减一并返回 `WAIT_SIGNAL`；否则返回 `tid`，即保存的 `os_wait()` 返回值。

使用示例

```
u8 result;

result = os_sem_wait(0);

if (result == WAIT_SIGNAL)
{
    /* 按照返回值判断获取结果 */
}
```

注意事项

- 当信号量计数为零时，该函数可能进入任务等待流程。
- 当前 API 没有提供独立的超时参数，不能仅根据本函数的接口承诺超时等待能力。
- 不要直接把所有非 `WAIT_SIGNAL` 的返回值都解释为超时；其含义需要结合 `os_wait()` 的实现确认。
- 当前代码在调用 `os_wait()` 前已经执行 `EA = 1`。计数检查与等待登记之间是否存在竞争窗口，需要结合内核等待实现进一步验证。
- 如果唤醒后计数仍为零，函数会返回 `os_wait()` 的结果，而不是无条件返回 `WAIT_SIGNAL`。因此，调用方应在确认内核返回语义后，再确定可靠的成功判定方式。

7.7.3 os_sem_post

函数原型

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

参数说明

参数	类型	说明
`sid`	`u8`	信号量编号

返回值

- `1`：资源的等待位图非零，函数执行了等待任务处理分支。
- `0`：没有等待任务，函数完成计数处理。
- `-1`：信号量编号无效。

需要注意，源码在等待位图非零时会尝试选择任务并调用 `wake_task()`。即使最终没有选出有效任务，该分支仍然返回 `1`，所以该返回值不应直接理解为“已确认成功唤醒一个任务”。

功能说明

该函数用于执行信号量 V 操作。源码首先检查编号，然后通过 `EA = 0` 进入临界区。

如果 `wait_mask` 非零，函数遍历任务位图，选择 `OS_TASK[i].base_prio` 最大的等待任务，并调用：

```
wake_task(next, WAIT_SIGNAL);
```

此分支不增加信号量计数，而是尝试直接唤醒等待任务。

如果没有等待任务，函数检查当前计数是否小于 `255`。只有在小于 `255` 时才执行 `value++`，从而避免计数超过一个字节能够表示的最大值。随后函数恢复中断并返回 `0`。

使用示例

```
char result;

result = os_sem_post(0);

if (result == 1)
{
    /* 存在等待任务，已执行唤醒处理分支 */
}
else if (result == 0)
{
    /* 没有等待任务，已完成计数处理 */
}
else
{
    /* 信号量编号无效 */
}
```

注意事项

- 有等待任务时，当前实现优先执行唤醒分支，而不是增加计数。
- 没有等待任务时，计数最多累加到 `255`；达到上限后再次调用不会继续增加计数。
- 任务选择依据是 `base_prio`，不是 `cur_prio`。
- 当前源码没有提供独立的 `os_sem_post_from_isr()` 接口，不能据此宣称本模块具备专用的中断释放 API。
- 函数通过 `EA = 0` 和 `EA = 1` 保护临界区，但没有保存并恢复调用前的中断使能状态。

7.8 使用建议

- **合理设置初始计数。** 初始计数值应与需要管理的资源数量或同步场景相匹配。
- **区分信号量与互斥锁。** 信号量不跟踪持有者，不应将其视为具备完整所有权检查的互斥锁。
- **谨慎处理等待返回值。** 应结合 `os_wait()` 和 `wake_task()` 的具体实现确认阻塞、唤醒及返回值的对应关系。
- **避免重复初始化活动中的信号量。** 重新初始化会清空等待位图，可能影响已有等待任务。
- **注意计数上限。** `value` 的实际计数累加上限为 `255`。
- **评估临界区使用方式。** 当前实现直接将 `EA` 置为 `0`，结束时再置为 `1`，调用前若中断已经关闭，函数返回时也会将其打开。
- **不要默认支持中断上下文调用。** 应在确认内核调度、等待和唤醒机制的约束后，再确定各接口的可调用上下文。

7.9 使用说明

- **计数机制**：信号量通过计数值管理资源的可用状态，并配合统一等待机制处理任务等待与唤醒。
- **资源获取**：使用信号量时，应根据接口定义处理获取结果，并遵循对应的等待与唤醒约定。
- **资源释放**：释放信号量时，应确保调用符合模块规定的使用方式，避免因不恰当的调用影响任务调度。
- **并发使用**：在多个任务或中断共同访问信号量的场景中，应结合系统的调度机制和中断管理规则设计同步流程。
- **中断上下文**：不要默认所有信号量接口均可在中断服务程序中安全调用；应以接口明确规定的调用上下文为准。

Mutex

互斥锁模块

8.1 模块简介

互斥锁模块用于协调多个任务对共享资源的访问，采用统一资源模型记录互斥锁的拥有者、等待任务及等待状态。

模块提供互斥锁初始化、获取和释放接口。获取已被其他任务持有的互斥锁时，当前任务通过 `os_wait()` 进入等待流程；实现还包含优先级继承逻辑，尝试提高锁拥有者的当前优先级，降低高优先级任务因低优先级任务持锁而受到的阻塞。

当前实现不支持递归获取同一把互斥锁。

8.2 设计目标

- 为共享资源提供互斥访问机制。
- 通过资源拥有者标识记录锁的持有关系。
- 支持等待任务管理与锁释放后的任务唤醒。
- 在发生锁竞争时尝试实施优先级继承。

8.3 使用场景

- 保护多个任务共同访问的共享数据。
- 协调对共享硬件资源的访问。
- 保护需要互斥执行的临界代码。
- 管理任务之间存在竞争关系的资源。

8.4 数据类型

本模块使用 `hrtos_internal.h` 中定义的 `OS_RESOURCE` 结构，通过 `OS_RES[mid]` 访问互斥锁对应的资源槽位。

主要使用以下成员：

成员	功能说明
value	初始化时设为 1，但当前锁获取和释放逻辑主要依据 owner 判断锁状态
owner	当前互斥锁拥有者的任务 ID；OS_INVALID_ID 表示没有拥有者
wait_mask	等待该资源的任务位图
wait_cnt	等待任务计数，初始化时清零

任务优先级相关字段：

字段	功能说明
base_prio	任务基础优先级
cur_prio	任务当前优先级，可在锁竞争时被调整

互斥锁的获取和释放逻辑会同时操作任务优先级字段与 OS_PROCESS_OK 中的优先级编码。其编码方式及调度效果应以系统任务创建和调度代码为准。

8.5 宏定义

本模块没有在公开头文件中定义专用宏。

8.6 API 函数列表

函数名称	功能说明
os_mutex_init()	初始化互斥锁资源
os_mutex_lock()	获取互斥锁，必要时进入等待流程
os_mutex_unlock()	释放互斥锁，并在存在等待任务时尝试转移所有权

8.7 API 详细说明

8.7.1 os_mutex_init

函数原型：

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

参数说明：

参数	类型	说明
mid	u8	互斥锁对应的资源槽位编号

返回值：

返回值	说明
`1`	初始化成功
`-1`	资源槽位编号无效

功能说明：

初始化指定的互斥锁资源。函数将资源的 `value` 设置为 `1`，将 `owner` 设置为 `OS_INVALID_ID`，并清空等待位图与等待计数。

调用示例：

```
char result;

result = os_mutex_init(0);

if (result == 1)
{
    /* 初始化成功 */
}
```

注意事项：

- `mid` 必须小于 `OS_RESOURCE_MAX`。
- 应在系统启动阶段或能够确保资源未被并发使用的环境中初始化。
- 该函数会覆盖拥有者和等待状态，不应在互斥锁正在使用时重新初始化。
- 当前实现没有关闭总中断或采用其他显式同步措施，初始化时应确保不存在并发访问。
- 函数形参在实现中声明为 `char`，公开头文件中声明为 `u8`。建议统一两处声明与定义的类型，避免接口声明不一致。

8.7.2 os_mutex_lock

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	互斥锁对应的资源槽位编号

返回值：

返回值	说明
`1`	当前任务直接获取锁
`-1`	编号无效，或当前任务已经持有该锁
其他值	锁被占用时，返回 `os_wait()` 的结果

功能说明：

尝试获取指定互斥锁。函数首先检查资源编号，然后读取当前任务 ID，并关闭总中断进入临界区。

互斥锁已由当前任务持有时

函数恢复总中断并返回 `-1`，因此当前实现不支持递归获取。

互斥锁没有拥有者时

函数将当前任务 ID 写入 `owner`，根据 `OS_PROCESS_OK` 中的优先级编码重新设置当前任务的 `base_prio` 和 `cur_prio`，恢复总中断后返回 `1`。

互斥锁已由其他任务持有时

函数根据当前任务的基础优先级和锁拥有者的当前优先级执行优先级比较。如果等待任务的基础优先级高于锁拥有者的当前优先级，则提高拥有者的 `cur_prio`，并尝试同步更新 `OS_PROCESS_OK` 中的优先级编码。

完成上述处理后，函数恢复总中断，并调用：

```
os_wait(WAIT_MUTEX, mid, 0);
```

该调用将互斥锁等待交由内核等待机制处理，函数最终返回 `os_wait()` 的返回值。

调用示例：

```
char result;

result = os_mutex_lock(0);

if (result == 1)
{
    /* 当前调用直接获取锁 */
    /* 访问受保护的共享资源 */

    os_mutex_unlock(0);
}
```

注意事项：

- 返回 `1` 在当前实现中明确对应直接获取锁的路径。锁竞争路径返回的是 `os_wait()` 的结果，不能仅根据返回值为非零就断言当前任务已经获得互斥锁。
- 锁竞争时的优先级提升发生在调用 `os_wait()` 之前。
- 当前实现没有在 `os_mutex_lock()` 内部循环重试获取锁，也没有在等待返回后自行验证当前任务是否已经成为锁拥有者。
- 优先级继承逻辑只比较当前等待任务与锁拥有者的优先级；多个锁、多个等待者之间的完整继承与恢复策略，还需要结合其他内核代码确认。
- 函数直接设置 `EA = 0` 和 `EA = 1`，没有保存并恢复调用前的总中断使能状态。

8.7.3 os_mutex_unlock

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	互斥锁对应的资源槽位编号

返回值：

返回值	说明
`1`	所有权已转移给选中的等待任务，并调用唤醒接口
`0`	释放流程完成，未保留有效的等待任务所有权
`-1`	编号无效，或当前任务不是锁拥有者

功能说明：

释放指定互斥锁。调用时，函数首先检查编号，然后确认当前任务是否为锁拥有者。非拥有者调用时返回 `-1`。

验证拥有者身份后，函数将当前任务的 `cur_prio` 恢复为 `base_prio`，并同步更新 `OS_PROCESS_OK` 中的优先级编码。

存在等待任务时

函数遍历 `wait_mask` 中的任务位，选择 `OS_TASK[i].cur_prio` 最高的任务。选定任务后，将互斥锁的 `owner` 设置为该任务 ID，并调用：

```
wake_task(next, WAIT_SIGNAL);
```

随后恢复总中断并返回 `1`。

如果等待位图非零，但遍历后没有选出有效任务，函数会清空 `wait_mask`、将 `owner` 设为 `OS_INVALID_ID`，恢复总中断并返回 `0`。

没有等待任务时

函数将 `owner` 设置为 `OS_INVALID_ID`，恢复总中断并返回 `0`。

调用示例：

```
char result;

result = os_mutex_unlock(0);
```

```
if (result == 1)
{
    /* 锁所有权已转移，并调用了等待任务唤醒接口 */
}
else if (result == 0)
{
    /* 释放流程完成，未转移给有效等待任务 */
}
```

注意事项：

- 只有当前锁拥有者才能释放互斥锁。
- 当前实现会在释放时恢复当前任务的基础优先级。
- 选定等待任务后，函数先更新 `owner`，再调用 `wake_task()`。因此，所有权转移和任务唤醒的实际配合效果需要结合 `os_wait()` 与 `wake_task()` 的实现验证。
- 选择等待任务时，代码遍历范围为 `0` 到 `OS_PROCESS_MAX - 1`，并依据 `cur_prio` 选择最高优先级任务；当多个任务优先级相同时，优先选择编号较小的任务。
- 函数没有在释放时显式验证候选任务当前是否仍处于 `WAIT_MUTEX` 状态，而是依赖 `wait_mask` 的正确性。
- 函数直接设置 `EA = 0` 和 `EA = 1`，没有保存并恢复调用前的总中断使能状态。

8.8 使用建议

- 在系统启动阶段完成互斥锁初始化，并避免在运行过程中重复初始化正在使用的锁。
- 获取锁后应尽量缩短持锁时间，避免长时间占用共享资源。
- 对锁竞争路径，应根据实际 `os_wait()` 返回约定编写调用逻辑，不应将任意非零返回值都视为获取锁成功。
- 只有成功持有互斥锁的任务才应调用释放接口。
- 多把锁共同使用时，应在应用层规划统一的加锁顺序，降低循环等待风险。
- 对实时性要求较高的任务，应评估锁竞争导致的阻塞时间，以及优先级继承与恢复策略的实际效果。

8.9 使用说明

- **锁的使用**：互斥锁用于协调多个任务对共享资源的访问。访问共享资源前应获取互斥锁，并在使用完成后及时释放。
- **任务所有权**：互斥锁具有任务所有权约束，应由持有该锁的任务执行正常释放操作，避免错误释放或破坏锁状态。
- **竞争与等待**：当互斥锁已被其他任务持有时，后续任务的处理行为应遵循接口规定的等待机制。使用时应结合任务调度与等待唤醒机制理解接口返回结果。
- **优先级继承**：优先级继承用于缓解优先级反转。使用互斥锁时，应注意任务优先级可能随锁竞争发生变化，不应假定任务优先级在运行期间始终保持不变。
- **多锁场景**：当任务同时持有多个互斥锁时，应合理安排加锁与解锁顺序，并避免形成循环等待。
- **中断与临界区**：互斥锁主要用于任务间同步，不应默认其接口可在中断服务程序中安全调用。相关操作应遵循系统统一的中断管理约定。

Event

事件模块

9.1 模块简介

HRTOS 事件模块提供基于事件状态的任务同步机制。任务可以等待指定事件，其他任务或中断处理代码可以通过事件触发接口设置事件状态，并唤醒等待该事件的任务。

事件模块基于统一资源对象 `OS_RESOURCE` 实现，使用资源对象的 `value` 字段记录事件状态，并通过 `wait_mask` 和 `wait_cnt` 管理等待任务。

模块提供事件初始化、事件触发、事件等待、状态查询和事件删除接口。

9.1.1 设计特点

- **事件状态保持：** 事件触发后，`value` 被设置为 `1`，后续查询或等待操作可以检查该状态。
- **多任务唤醒：** 触发事件时，遍历等待位图，对其中标记的任务调用唤醒接口。
- **支持超时等待：** 事件尚未触发时，通过统一等待接口进入事件等待流程。
- **统一资源管理：** 事件使用 HRTOS 的统一资源对象，不单独定义事件控制块结构。
- **接口分离：** 提供独立的初始化、触发、等待、查询和删除函数。

9.1.2 适用场景

- 多个任务等待同一个条件成立
- 任务之间进行状态变化通知
- 任务等待外部事件或中断产生的通知
- 需要查询某个事件当前是否已经触发的场景

9.2 数据结构

事件模块使用系统配置模块定义的 `OS_RESOURCE` 结构。

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

```
} OS_RESOURCE;
```

事件模块主要使用以下成员：

成员	作用
`value`	记录事件状态，事件未触发时为 `0`，触发后设置为 `1`
`wait_cnt`	记录等待任务数量
`wait_mask`	使用位图标记等待任务

事件对象编号对应统一资源对象数组 `OS_RES` 的下标，有效范围为 `0` 至 `OS_RESOURCE_MAX - 1`。当前配置中，`OS_RESOURCE_MAX` 为 `6`。

事件模块不单独定义公开的事件 ID 类型或事件专用控制块。

9.3 API 函数列表

函数	功能
`os_event_init()`	初始化指定事件资源
`os_event_write()`	触发事件并尝试唤醒等待任务
`os_event_wait()`	等待事件触发
`os_event_query()`	查询事件触发状态
`os_event_delete()`	清除事件触发状态

9.4 事件初始化 os_event_init

9.4.1 函数原型

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

9.4.2 参数说明

参数	类型	说明
`obj`	`u8`	事件资源槽位编号

9.4.3 返回值

返回值	说明
`1`	初始化成功
`-1`	事件资源槽位编号无效

9.4.4 功能说明

初始化指定的事件资源。函数首先检查 `obj` 是否小于 `OS_RESOURCE_MAX`，通过检查后进入中断关闭状态，调用 `os_res_init(&OS_RES[obj])` 初始化对应的统一资源对象，随后重新开启中断并返回成功。

```
char result = os_event_init(0);

if (result == 1)
{
    /* 事件初始化成功 */
}
```

9.4.5 注意事项

- `obj` 必须小于 `OS_RESOURCE_MAX`。
- 初始化函数会调用统一资源初始化接口 `os_res_init()`。
- 资源对象各成员的具体初始化值，应以 `os_res_init()` 的实现为准。
- 初始化操作涉及中断状态修改，使用时应结合系统中断管理规则评估调用环境。

9.5 事件触发 os_event_write

9.5.1 函数原型

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

9.5.2 参数说明

参数	类型	说明
`obj`	`u8`	事件资源槽位编号

9.5.3 返回值

返回值	说明
`1`	事件触发操作完成
`-1`	事件资源槽位编号无效

9.5.4 功能说明

触发指定事件，并遍历等待位图，尝试唤醒等待该事件的任务。

函数执行过程如下：

- 检查事件资源槽位编号是否合法。
- 将 `EA` 设置为 `0`。
- 将 `OS_RES[obj].value` 设置为 `1`，表示事件已触发。

- 当 `wait_cnt` 大于 `0` 时，遍历标准任务编号范围。
- 检查 `wait_mask` 中对应任务的位是否置位。
- 对位图中标记的任务调用 `wake_task(i, WAIT_SIGNAL)`。
- 将 `EA` 设置为 `1`，然后返回成功。

```
char result = os_event_write(0);

if (result == 1)
{
    /* 事件触发操作完成 */
}
```

9.5.5 事件触发与状态保持

事件触发后，`value` 被设置为 `1`。根据当前实现，`os_event_write()` 本身不会在触发后自动将该值恢复为 `0`。

因此，事件状态具有保持特征：即使某个任务在事件触发之后才调用 `os_event_wait()`，只要事件状态仍为 `1`，该任务也会直接返回，不进入等待流程。

事件触发后，函数会遍历等待位图并对标记的任务调用唤醒接口。实际成功唤醒的任务数量及其后续调度行为，取决于 `wake_task()` 的具体实现和当时的任务状态。

9.5.6 中断与并发注意事项

当前实现通过直接操作 `EA` 控制中断状态，而不是通过临界区封装宏完成保护。

需要特别注意：

- 函数退出时直接执行 `EA = 1`，并非恢复进入函数前保存的中断使能状态。
- 从中断服务程序调用时，应结合具体中断嵌套规则和内核中断管理机制评估安全性。
- 从任务上下文调用时，也应评估与其他中断及内核操作之间的并发关系。
- 不能仅根据函数注释中的 ISR-safe 描述，就认定该接口在所有中断上下文中都安全。
  在确认临界区和中断退出机制之前，不应将该函数描述为无条件安全的中断调用接口。

9.6 事件等待 os_event_wait

9.6.1 函数原型

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

9.6.2 参数说明

参数	类型	说明
`obj`	`u8`	事件资源槽位编号
`tick`	`u16`	最大等待时间参数，以系统

参数	类型	说明
		节拍为单位

9.6.3 返回值

返回值	说明
`1`	调用时事件已经触发，函数直接返回
`-1`	事件资源槽位编号无效
其他返回值	由统一等待接口 `os_wait()` 返回

9.6.4 功能说明

等待指定事件触发。函数先检查事件资源槽位编号，再短暂关闭中断并检查事件状态。

当 `OS_RES[obj].value` 非零时，函数重新开启中断并直接返回 `1`，不进入等待状态。

当事件尚未触发时，函数重新开启中断，然后调用：

```
return os_wait(WAIT_EVENT, obj, tick);
```

由统一等待接口处理后续的阻塞、超时和唤醒流程。

```
char result = os_event_wait(0, 100);

if (result == 1)
{
    /* 调用时事件已经触发 */
}
else if (result == -1)
{
    /* 事件资源槽位编号无效 */
}
else
{
    /* 根据统一等待接口的返回规则处理 */
}
```

9.6.5 等待行为

事件等待接口包含两条不同的执行路径：

- **事件已触发：** 直接返回 `1`，不进入任务等待流程。
- **事件未触发：** 调用 `os_wait()`，进入统一的事件等待流程。

在等待过程中，事件触发可能通过 `wake_task()` 唤醒任务；等待超时则由统一等待机制处理。

`tick == 0` 是否表示无限等待，以及超时唤醒和事件唤醒分别返回什么结果，需要结合 `os_wait()` 和任务等待管理源码确认，不能仅凭本函数定义确定。

9.6.6 注意事项

- 该函数可能使当前任务进入阻塞状态。
- 等待时间以系统节拍为单位，而不是直接以毫秒为单位。
- 事件触发状态保持为 `1` 时，后续调用可以直接返回。
- 调用方应根据统一等待接口的实际返回约定处理超时和其他结果。

9.7 事件状态查询 os_event_query

9.7.1 函数原型

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

9.7.2 参数说明

参数	类型	说明
`obj`	`u8`	事件资源槽位编号

9.7.3 返回值

返回值	说明
`1`	事件已触发
`0`	事件未触发
`-1`	事件资源槽位编号无效

9.7.4 功能说明

查询指定事件当前的触发状态。

函数先检查 `obj` 是否有效，然后读取对应资源对象的 `value` 字段：非零时返回 `1`，否则返回 `0`。

```
char state = os_event_query(0);

if (state == 1)
{
    /* 事件已触发 */
}
else if (state == 0)
{
    /* 事件未触发 */
}
else
{
    /* 事件资源槽位编号无效 */
}
```

9.7.5 注意事项

- 该函数不主动修改事件状态。
- 该函数不会主动调用任务唤醒接口。
- 返回值为 `0` 时，只能说明读取时事件状态为零。
- 查询操作本身不代表事件状态在函数返回后仍然保持不变。

9.8 事件删除 os_event_delete

9.8.1 函数原型

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

9.8.2 参数说明

参数	类型	说明
`obj`	`u8`	事件资源槽位编号

9.8.3 返回值

返回值	说明
`1`	清除操作完成
`-1`	事件资源槽位编号无效

9.8.4 功能说明

当前实现检查资源槽位编号后，将指定资源对象的 `value` 字段设置为 `0`。

```
char result = os_event_delete(0);

if (result == 1)
{
    /* 事件状态清除操作完成 */
}
```

需要注意，当前实现并没有调用资源释放函数，也没有主动清空 `wait_mask`、`wait_cnt` 或唤醒等待任务。因此，该函数实际执行的是清除事件状态，而不是完整销毁资源对象。

9.8.5 注意事项

- 调用后，事件状态被设置为未触发。
- 等待队列成员不会由本函数主动清理。
- 本函数不会主动唤醒等待任务。
- 不应将其描述为释放资源或清空等待队列的完整删除操作。

如果应用需要真正销毁事件对象，必须结合统一资源管理机制设计相应流程，避免残留等待状态影响后续使用。

9.9 使用流程

事件模块的典型使用流程是先初始化事件，再由任务等待事件，由事件产生方触发事件，最后根据需要查询或清除事件状态。

```
/* 初始化事件 */
os_event_init(0);

/* 等待事件，最多等待指定的系统节拍数 */
char result = os_event_wait(0, 100);

/* 由事件产生方触发事件 */
os_event_write(0);

/* 查询事件状态 */
char state = os_event_query(0);

/* 清除事件触发状态 */
os_event_delete(0);
```

以上代码仅用于说明接口调用关系。实际多任务程序中，应由适当的任务或中断处理路径执行相应操作，并根据函数返回值进行错误处理。

9.10 使用说明

- **资源编号**：事件对象使用统一资源数组中的资源槽位，使用前应确保事件编号有效，并符合系统资源配置要求。
- **事件状态**：当前实现采用保持型事件状态。事件触发后，状态不会自动消费，具体状态变化应遵循事件接口的定义。
- **事件触发**：触发事件时，系统会处理等待该事件的任务，并通过统一任务唤醒机制通知符合条件的等待任务。
- **等待与超时**：事件等待的唤醒结果和超时行为由统一等待机制协同管理，调用时应遵循相应接口的返回值约定。
- **初始化与删除**：事件初始化和删除应遵循接口规定的生命周期管理方式。删除事件前，应确保不存在仍依赖该事件的任务或其他操作。
- **中断上下文**：事件接口的中断调用限制取决于具体实现。除非接口明确保证，否则不应默认其可在中断服务程序或嵌套中断环境中安全调用。

Mailbox

邮箱模块

10.1 模块简介

邮箱模块提供基于统一资源模型的单字节消息传递机制，可用于任务间的数据传递、简单事件通知和任务同步。

邮箱使用 `OS_RES` 数组中的资源槽位存储数据，并通过等待任务位图记录等待该邮箱的任务。模块提供邮箱初始化、数据发送、数据接收、状态查询和内容清除等接口。

当前实现属于**单槽位邮箱**：每个邮箱资源只能保存一个字节的数据，不具备消息队列的缓存能力。发送新数据时，邮箱中的原有数据可能被覆盖。

10.2 设计目标

- 提供轻量级的单字节消息传递机制。
- 复用 HRTOS 统一资源结构，减少独立资源管理开销。
- 支持任务等待与发送后的等待任务唤醒。
- 提供阻塞接收和非阻塞查询两种访问方式。

10.3 使用场景

- 任务之间传递简单数据。
- 任务之间进行事件通知。
- 保存单个待处理消息。
- 配合任务等待机制实现基本同步。

10.4 数据类型

本模块使用 `hrtos_internal.h` 中定义的 `OS_RESOURCE` 结构，并通过 `OS_RES[mid]` 访问对应资源。

主要使用的成员如下：

成员	功能
`value`	保存邮箱数据，当前实现同时使用 `0` 表示空邮箱
`owner`	资源所有者字段，邮箱初始化时设置为 `OS_INVALID_ID`
`wait_mask`	等待该资源的任务位图
`wait_cnt`	等待任务计数，初始化时清零

重要限制： 当前实现使用 `value == 0` 判断邮箱为空，因此不能可靠地区分“邮箱为空”和“邮箱中存放了 `0x00`”。如果需要传递完整的 `0x00`～`0xFF` 数据范围，应考虑增加独立的邮箱有效标志，或调整邮箱状态表示方式。

10.5 宏定义

本模块没有在公开头文件中定义专用宏。

邮箱编号的有效范围为：

```
0 <= mid < OS_RESOURCE_MAX
```

其中 `OS_RESOURCE_MAX` 表示系统资源槽位数量。

10.6 API 函数列表

函数名称	功能说明
`os_mailbox_init()`	初始化邮箱资源
`os_mailbox_send()`	向邮箱写入数据并尝试唤醒等待任务
`os_mailbox_receive()`	接收邮箱数据，邮箱为空时尝试进入等待
`os_mailbox_query()`	查询邮箱当前保存的值
`os_mailbox_clear()`	清除邮箱当前保存的值

10.7 API 详细说明

10.7.1 os_mailbox_init

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	邮箱对应的资源槽位编号

返回值：

无。

功能说明：

初始化指定邮箱资源。传入的编号超出 `OS_RESOURCE_MAX` 时，函数直接返回，不执行初始化。

初始化过程包括：

- 将 `value` 设置为 `0`。
- 将 `owner` 设置为 `OS_INVALID_ID`。
- 将 `wait_mask` 清零。
- 将 `wait_cnt` 清零。

调用示例：

```
os_mailbox_init(0);
```

注意事项：

- `mid` 必须小于 `OS_RESOURCE_MAX`。
- 应在系统启动阶段或能够确保资源未被并发使用的环境中完成初始化。
- 初始化会清空等待位图及等待计数，不应在邮箱正在使用时随意调用。
- 初始化后，邮箱的 `value` 为 `0`，当前实现将其视为空状态。

10.7.2 os_mailbox_send

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	邮箱对应的资源槽位编号
`content`	`u8`	待写入邮箱的数据

返回值：

返回值	说明
`1`	邮箱写入操作完成
`-1`	邮箱编号无效

功能说明：

向指定邮箱写入一个字节的数据。函数先关闭总中断，再将 `content` 写入邮箱的 `value` 字段。

如果 `wait_mask` 非零，函数从任务位图中查找第一个置位的任务编号，并调用 `wake_task(next, WAIT_SIGNAL)` 尝试唤醒该任务。随后恢复总中断并返回 `1`。

如果没有等待任务，则保留写入的数据，恢复总中断并返回 `1`。

调用示例：

```
char result;

result = os_mailbox_send(0, 0x55);

if (result == 1)
{
    /* 邮箱写入操作完成 */
}
```

注意事项：

- 邮箱只有一个数据存储槽位，发送新数据会直接覆盖旧数据。
- 返回 `1` 代表函数完成了写入及相应的唤醒处理流程，并不代表接收任务已经成功处理数据。
- 当前实现没有检查邮箱中是否已有未接收的数据。
- 发送内容为 `0` 时，接收接口会将邮箱判断为空，因此 `0x00` 不能作为普通有效消息可靠传递。
- 函数通过直接设置 `EA = 0` 和 `EA = 1` 控制总中断，未保存并恢复调用前的中断使能状态。调用前若总中断原本关闭，函数返回前仍会将其打开。
- 当前提供的实现没有专门的 ISR 发送接口，不能据此认定该 API 可在任意中断上下文中安全调用。

10.7.3 os_mailbox_receive

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	邮箱对应的资源槽位编号

返回值：

返回值	说明
`-1`	邮箱编号无效
其他值	函数返回的邮箱数据或等待后的读取结果，具体行为受当前实现及等待机制影响

功能说明：

从指定邮箱读取数据。邮箱编号无效时，函数直接返回 `-1`。

函数关闭总中断后检查邮箱的 `value`：

- 当 `value == 0` 时，先恢复总中断，再调用 `os_wait(WAIT_MAILBOX, mid, 0)`，尝试进入指定邮箱的等待流程。
- 当 `value != 0` 时，将数据复制到局部变量 `_data`，清空邮箱内容，恢复总中断并返回读取的数据。

在调用 `os_wait()` 后，当前实现会继续检查 `m->value`。如果此时 `value != 0`，函数返回局部读取路径中的数据；如果仍为 `0`，函数最终返回当前的 `m->value`。

调用示例：

```
int data;

data = os_mailbox_receive(0);

if (data >= 0)
{
    /* 根据实际协议处理数据 */
}
```

注意事项：

- 邮箱有非零数据时，读取后会清空邮箱。
- 邮箱为空时，函数会尝试调用 `os_wait()`，但能否正确阻塞、唤醒并取得数据，需要结合 `os_wait()`、`wake_task()` 和调度器的实现验证。
- 当前代码没有在 `os_wait()` 返回后明确检查唤醒结果，也没有在该路径中重新执行与快路径相同的完整取数及清空操作。
- 当等待返回后邮箱仍为空时，函数会返回 `0`，而不是明确的等待失败状态。
- 由于 `0` 同时表示空邮箱，返回值无法可靠地区分有效的零值数据与空状态。
- 当前实现的接收慢路径存在需要进一步审查的地方，文档不应直接承诺“阻塞接收一定能够返回正确数据”。

10.7.4 os_mailbox_query

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	邮箱对应的资源槽位编号

返回值：

返回值	说明
`-1`	邮箱编号无效
其他值	当前 `value` 字段的值

功能说明：

读取指定邮箱的当前 `value`，不清除邮箱内容，也不主动调用等待或唤醒接口。

调用示例：

```
int data;

data = os_mailbox_query(0);

if (data > 0)
{
    /* 当前邮箱保存了非零值 */
}
```

注意事项：

- 查询操作本身不修改邮箱状态。
- 返回 `0` 不能证明邮箱中存在一个有效的零值消息；按照当前实现，`0` 同时表示空状态。
- 该函数直接读取共享资源成员，没有显式关闭总中断或采用其他同步措施。并发访问时的可见性和一致性应结合具体调用环境评估。

10.7.5 os_mailbox_clear

函数原型：

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

参数说明：

参数	类型	说明
`mid`	`u8`	邮箱对应的资源槽位编号

返回值：

返回值	说明
`1`	清除操作完成
`-1`	邮箱编号无效

功能说明：

清除指定邮箱当前保存的值。编号有效时，函数关闭总中断，将 `OS_RES[mid].value` 设置为 `0`，随后重新打开总中断并返回 `1`。

调用示例：

```
char result;

result = os_mailbox_clear(0);
```

注意事项：

- 清除操作会丢弃当前保存的数据。
- 该函数只修改 `value`，不会清除 `wait_mask`、`wait_cnt` 或修改 `owner`。
- 当前实现没有主动唤醒等待任务。
- 函数使用直接设置 `EA = 0`、`EA = 1` 的方式控制总中断，未恢复调用前的中断使能状态。

10.8 使用建议

- 邮箱适合保存单个待处理数据，不适合需要缓存多条消息的场景。
- 如果发送方可能连续发送多条消息，应评估覆盖风险，必要时使用消息队列等机制。
- 当前邮箱实现不适合可靠传递 `0x00`，在修正空状态表示之前，应明确限制消息数据范围。
- 在将邮箱用于关键任务同步前，应结合等待、唤醒和调度器源码验证完整的阻塞与唤醒流程。
- 在任务或中断上下文中调用相关接口时，应确认其执行上下文要求，以及总中断使能状态的处理方式。

10.9 使用说明

- **单消息存储**：邮箱用于保存单条消息，不提供多消息队列或消息缓存功能。连续发送消息时，新消息可能覆盖尚未接收的旧消息。
- **消息值限制**：当前实现使用 `value == 0` 表示邮箱为空，因此不应将零值作为需要可靠传递的有效消息。
- **发送与接收**：发送和接收操作通过统一资源管理及任务等待机制协同工作。使用时应遵循接口规定的调用顺序，并正确处理返回结果。
- **等待与唤醒**：邮箱操作可能涉及等待任务的唤醒，具体行为取决于统一等待与任务调度机制。
- **初始化时机**：初始化会重置邮箱状态及相关等待信息。应在邮箱尚未投入使用时完成初始化，避免影响正在等待或使用该邮箱的任务。
- **并发访问**：多个任务共享邮箱时，应遵循系统的任务调度与同步约定，不应默认查询、发送和清空操作具备完整的并发保护。
- **中断状态**：使用邮箱接口时，应注意其对中断使能状态的影响；在中断服务程序或临界区中调用前，应确认接口符合相应的调用约束。

Msgq

消息队列模块

11.1 模块简介

消息队列模块提供基于环形缓冲区的单字节消息传递机制，支持消息缓存、阻塞收发和非阻塞收发。

消息队列通过 `os_msgq_t` 控制块管理缓冲区地址、队列容量、读写索引及当前消息数量。发送操作将数据写入队列，接收操作从队列中读取数据。队列已满或为空时，可根据调用参数选择立即返回或调用内核等待接口。

与单槽位邮箱相比，消息队列可以缓存多条消息，适合生产者与消费者之间存在处理速度差异的场景。

11.2 设计目标

- 提供轻量级的任务间消息传递机制。
- 使用环形缓冲区缓存多条单字节消息。
- 支持阻塞与非阻塞两种收发模式。
- 通过统一资源模型中的等待位图和任务唤醒机制协调收发任务。

11.3 使用场景

- 任务间传递字节数据。
- 缓存传感器采样结果或串口接收数据。
- 解耦数据生产任务与数据处理任务。
- 缓冲突发消息，平滑生产与消费速度差异。

11.4 数据类型

11.4.1 os_msgq_t

消息队列控制块类型为 `os_msgq_t`。当前实现使用以下成员：

成员	功能说明
`buf`	环形缓冲区首地址
`_size`	缓冲区容量，即可保存的消息数量上限
`head`	写入索引
`tail`	读取索引
`count`	当前队列中的消息数量

`head` 和 `tail` 均以 `_size` 为模循环递增。`count` 用于区分队列为空和队列已满的状态，因此该实现可以使用全部 `_size` 个缓冲区字节，不需要额外保留一个空槽位。

11.5 宏定义

本模块没有在公开头文件中定义专用宏。

11.6 API 函数列表

函数名称	功能说明
`os_msgq_init()`	初始化消息队列并绑定缓冲区
`os_msgq_send()`	向消息队列发送一个字节
`os_msgq_recv()`	从消息队列接收一个字节
`os_msgq_clear()`	清空队列状态

11.7 API 详细说明

11.7.1 os_msgq_init

函数原型：

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

参数说明：

参数	类型	说明
`q`	`os_msgq_t *`	消息队列控制块指针
`buf`	`u8 *`	用户提供的环形缓冲区地址
`_size`	`u8`	队列容量，单位为字节消息数

返回值：

无。

功能说明：

将消息队列控制块与指定缓冲区关联，并初始化队列状态：

- `q->buf = buf`：保存缓冲区地址。
- `q->_size = _size`：设置队列容量。
- `q->head = 0`：将写入索引复位。
- `q->tail = 0`：将读取索引复位。
- `q->count = 0`：将消息数量清零。

调用示例：

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

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

注意事项：

- `q` 和 `buf` 必须指向有效内存。
- 缓冲区实际容量必须不小于 `_size`。
- 当前实现没有检查空指针，也没有检查 `_size` 是否为零。
- `_size` 为零时，收发路径中的取模运算可能产生错误，因此必须传入非零容量。
- 初始化会重置控制块状态，不应在队列仍被其他任务使用时随意重新初始化。
- 初始化只设置控制块，不会清除缓冲区中原有的字节内容；这些内容在 `count` 清零后不属于有效消息。

11.7.2 os_msgq_send

函数原型：

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

参数说明：

参数	类型	说明
`q`	`os_msgq_t *`	消息队列控制块指针
`obj`	`u8`	用于等待与唤醒管理的资源对象编号
`_data`	`u8`	待发送的单字节数据
`nonblock`	`char`	非零表示非阻塞模式，零表示阻塞模式

返回值：

返回值	说明
`1`	消息写入成功
`0`	非阻塞模式下队列已满

返回值	说明
其他值	阻塞路径中 `os_wait()` 返回的结果

功能说明：

向指定消息队列写入一个字节的数据。函数通过 `q->count` 判断队列是否已满。

队列未满时

- 将 `_data` 写入 `q->buf[q->head]`。
- 更新写入索引：`q->head = (q->head + 1) % q->_size`。
- 将消息数量加一。
- 检查 `OS_RES[obj].wait_mask`，查找等待该对象且等待类型为 `WAIT_MSG_RECV` 的任务。
- 找到任务后，调用 `wake_task(next, WAIT_SIGNAL)` 尝试唤醒。
- 恢复总中断并返回 `1`。

队列已满时

- 非阻塞模式：恢复总中断并返回 `0`。
- 阻塞模式：恢复总中断后调用 `os_wait(WAIT_MSG_SEND, obj, 0)`，并将等待接口的返回值保存在局部变量中。
- 如果等待返回后队列出现可用空间，函数继续执行写入流程；如果队列仍然已满，则返回等待接口的结果。

调用示例：

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

os_msgq_init(&my_queue, msg_buffer, sizeof(msg_buffer));

/* 非阻塞发送 */
result = os_msgq_send(&my_queue, 0, 0x55, 1);

if (result == 1)
{
    /* 消息已写入队列 */
}
```

注意事项：

- `obj` 不仅是普通参数，还用于访问 `OS_RES[obj]` 并管理等待任务。必须确保该编号有效，并与对应消息队列的等待管理约定一致。
- 当前函数没有验证 `q`、`q->buf` 或 `obj` 的有效性。
- 队列满时，阻塞模式能否正确恢复执行并完成发送，取决于 `os_wait()`、`wake_task()` 和调度器的实现。
- 该函数通过直接设置 `EA = 0` 和 `EA = 1` 控制总中断，没有保存并恢复调用前的中断使能状态。

- 当前提供的源码没有定义专用的 `os_msgq_send_from_isr()` 接口，因此不应将该函数描述为具有独立的 ISR 安全版本。

11.7.3 os_msgq_recv

函数原型：

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

参数说明：

参数	类型	说明
`q`	`os_msgq_t *`	消息队列控制块指针
`obj`	`u8`	用于等待与唤醒管理的资源对象编号
`_data`	`u8 *`	接收数据的存储地址
`nonblock`	`char`	非零表示非阻塞模式，零表示阻塞模式

返回值：

返回值	说明
`1`	消息读取成功
`0`	非阻塞模式下队列为空
其他值	阻塞路径中 `os_wait()` 返回的结果

功能说明：

从消息队列中读取一个字节的数据。函数通过 `q->count` 判断队列是否为空。

队列非空时

- 从 `q->buf[q->tail]` 读取队头消息，并写入 `*_data`。
- 更新读取索引：`q->tail = (q->tail + 1) % q->_size`。
- 将消息数量减一。
- 检查 `OS_RES[obj].wait_mask`，查找等待该对象且等待类型为 `WAIT_MSG_SEND` 的任务。
- 找到任务后，调用 `wake_task(next, WAIT_SIGNAL)` 尝试唤醒。
- 恢复总中断并返回 `1`。

队列为空时

- 非阻塞模式：恢复总中断并返回 `0`。
- 阻塞模式：恢复总中断后调用 `os_wait(WAIT_MSG_RECV, obj, 0)`，并保存等待接口的返回值。
- 如果等待返回后队列中已有消息，函数继续执行读取流程；如果队列仍为空，则返回等待接口的结果。

调用示例：

```
u8 msg_buffer[16];
os_msgq_t my_queue;
u8 data;
char result;

os_msgq_init(&my_queue, msg_buffer, sizeof(msg_buffer));

/* 非阻塞接收 */
result = os_msgq_recv(&my_queue, 0, &data, 1);

if (result == 1)
{
    /* data 为接收到的消息 */
}
```

注意事项：

- `_data` 必须指向可写内存，当前实现没有检查该指针是否有效。
- `obj` 必须与该消息队列的等待管理约定一致。
- 当前阻塞路径将 `os_wait()` 的返回值保存在局部变量中；只有等待返回后队列仍有消息时，才会进入实际读取分支。
- 当前函数没有在读取前显式验证 `obj < OS_RESOURCE_MAX`。
- 函数通过直接设置 `EA = 0` 和 `EA = 1` 控制总中断，没有恢复调用前的中断使能状态。
- 当前提供的源码没有定义专用的 `os_msgq_recv_from_isr()` 接口，不应据此承诺该函数可安全用于任意中断上下文。

11.7.4 os_msgq_clear

函数原型：

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

参数说明：

参数	类型	说明
`q`	`os_msgq_t *`	消息队列控制块指针

返回值：

无。

功能说明：

将消息队列的运行状态复位：

- `q->head = 0`。
- `q->tail = 0`。
- `q->count = 0`。

调用示例：

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

注意事项：

- 清空后，队列中的所有未读取消息均被视为无效。
- 函数只修改控制块，不擦除缓冲区内存中的实际字节。
- 当前实现不会主动唤醒等待发送或等待接收的任务。
- 函数没有检查 `q` 是否为空指针。
- 函数通过直接设置 `EA = 0` 和 `EA = 1` 控制总中断，没有恢复调用前的中断使能状态。

11.8 使用建议

- 根据最大突发消息量和消费者处理速度选择缓冲区容量。
- 在调用初始化接口前，为控制块和缓冲区分配足够且生命周期稳定的内存。
- 不要使用零容量队列。
- 多任务并发访问同一个队列时，应保证等待对象编号、资源等待位图与队列实例之间的对应关系正确。
- 对实时性要求较高的路径，可根据业务特点使用非阻塞模式，避免任务因队列状态进入等待。
- 若需要在中断中传递数据，应确认内核提供的专用中断安全接口及其调用约束；不要直接假定普通收发函数适用于 ISR。

11.9 使用说明

- **初始化配置**：初始化消息队列时，应确保控制块和缓冲区指针有效，并设置符合实际存储空间的队列容量。
- **资源编号**：发送和接收操作应使用有效的队列资源编号，并遵循项目中消息队列接口的定义。
- **队列容量**：消息队列的存储能力受配置容量限制。使用时应根据生产者与消费者的处理速度合理设置容量，并正确处理发送或接收操作的返回结果。
- **阻塞与等待**：阻塞式收发依赖统一等待、任务唤醒和调度机制。调用时应遵循接口规定的等待条件与返回值约定。
- **清空操作**：清空操作用于重置队列中的消息状态，不应默认其会自动取消或唤醒正在等待的任务。执行前应确认队列及相关任务处于适当状态。
- **并发访问**：多个任务共享消息队列时，应结合系统调度和同步机制设计收发流程，避免未经协调的并发操作影响队列状态。
- **中断上下文**：除非接口明确保证，否则不要默认消息队列接口适用于中断服务程序或嵌套中断环境。

Interrupt

中断管理模块

12.1 模块简介

HRTOS 中断管理模块提供中断编号查询、临界区保护、运行环境判断、中断触发调度，以及中断上下文与任务之间的通信接口。

模块通过临界区嵌套计数和中断状态保存机制管理临界区，通过调度请求标志及资源状态记录，支持中断环境下的邮箱、事件、信号量和消息队列操作。

模块主要包含以下功能：

- **中断识别**：获取当前中断编号，判断代码是否运行在中断环境。
- **临界区保护**：进入临界区时保存原有中断使能状态，并在适当时机恢复。
- **调度请求**：设置调度请求标志，必要时触发 Timer0 中断。
- **中断通信**：提供邮箱、事件、信号量和消息队列的中断发送或释放接口。

12.2 设计与使用场景

中断管理模块适用于以下场景：

- 在共享数据访问期间保护关键代码。
- 在中断服务程序中识别当前中断来源。
- 在中断中通知任务处理新数据或事件。
- 在中断中发送消息队列数据。
- 在需要系统重新检查调度条件时提交调度请求。

需要注意，中断上下文中的接口不应执行阻塞等待。具体接口是否可以安全用于所有中断场景，还取决于中断嵌套方式、共享资源访问规则和内核调度实现。

12.3 相关宏与内部状态

12.3.1 临界区宏

宏或接口	功能
`OS_ENTER_CRITICAL()`	进入临界区，具体展开行为取决于宏定义
`OS_EXIT_CRITICAL()`	退出临界区，具体展开行为取决于宏定义
`os_enter_critical()`	进入临界区并管理嵌套计数
`os_exit_critical()`	退出临界区，并在嵌套计数归零时恢复保存的中断状态

本文所提供的实现直接展示了 `os_enter_critical()` 和 `os_exit_critical()` 的行为。宏是否直接映射到这两个函数，应以实际配置文件中的定义为准。

12.3.2 相关内部状态

状态	功能
`EA`	8051 全局中断使能控制位
`os_critical_nesting`	临界区嵌套计数
`os_int_status`	保存进入最外层临界区前的中断使能状态
`OS_INTERRUPT_PROTECT`	`os_interrupt_id()` 返回的中断编号状态
`OS_ISR_FLAG`	`os_in_isr()` 查询的中断环境标志
`OS_SCHED_REASON`	调度请求标志
`TF0`	Timer0 溢出标志；`os_schedule_request()` 通过设置该标志触发 Timer0 中断处理

12.4 API 函数列表

函数	功能
`os_interrupt_id()`	获取当前中断编号
`os_enter_critical()`	进入临界区
`os_exit_critical()`	退出临界区
`os_in_isr()`	判断当前是否处于中断环境
`os_queue_send_from_isr()`	在中断环境向指定资源写入数据并提交调度请求
`os_schedule_request()`	设置调度请求标志并触发 Timer0 中断
`os_event_set_from_isr()`	在中断环境设置事件标志并记录待处理信号
`os_semaphore_post_from_isr()`	在中断环境释放信号量
`os_msgq_send_from_isr()`	在中断环境向消息队列发送一个字节

12.5 API 详细说明

12.5.1 os_interrupt_id

函数原型

```
char os_interrupt_id();
```

参数

无。

返回值

返回 `OS_INTERRUPT_PROTECT` 的当前值。接口注释约定中断编号为 `0`、`2～31`，并按照自然中断入口顺序排列。

功能说明

查询当前中断编号。实现本身只返回内部状态 `OS_INTERRUPT_PROTECT`，并不在该函数中计算或转换中断向量编号。

调用示例

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

注意事项

- 该接口适合在需要识别中断来源的代码中使用。
- 编号与具体中断入口的对应关系，应结合中断入口配置及内核实现确认。
- 如果在任务上下文中调用，其返回值是否具有有效的中断来源含义，不能仅凭该函数实现确定。

12.5.2 os_enter_critical

函数原型

```
void os_enter_critical();
```

参数与返回值

无参数，无返回值。

功能说明

进入临界区，通过嵌套计数支持多层临界区调用。

当 `os_critical_nesting == 0` 时，函数首先保存当前 `EA` 状态到 `os_int_status`，然后将 `EA` 置零。之后每次调用均增加嵌套计数。

其核心逻辑如下：

```
if (os_critical_nesting == 0)
{
    os_int_status = EA;
    EA = 0;
}

os_critical_nesting++;
```

调用示例

```
os_enter_critical();

/* 受保护的关键操作 */

os_exit_critical();
```

注意事项

- 每次进入临界区，都必须与对应的 `os_exit_critical()` 配对。
- 临界区支持嵌套；只有最外层进入时才保存中断状态并关闭全局中断。
- 临界区内的操作应尽量简短，以控制中断屏蔽时间。
- 不要在未核实配对关系的情况下随意调用退出接口。

12.5.3 os_exit_critical

函数原型

```
void os_exit_critical();
```

参数与返回值

无参数，无返回值。

功能说明

退出临界区，减少嵌套计数。当 `os_critical_nesting > 0` 时，函数才会执行减计数操作；当计数减至零时，将 `EA` 恢复为 `os_int_status` 保存的状态。

核心逻辑如下：

```
if (os_critical_nesting > 0)
{
    os_critical_nesting--;

    if (os_critical_nesting == 0)
    {
        EA = os_int_status;
    }
}
```

如果嵌套计数已经为零，调用不会执行任何操作。

注意事项

- 应与 `os_enter_critical()` 配对使用。
- 只有退出最外层临界区时才恢复保存的中断状态。
- 恢复的是进入最外层临界区之前的状态，而不是无条件打开中断。
- 不正确的配对可能破坏后续临界区的保护行为。

12.5.4 os_in_isr

函数原型

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

参数

无。

返回值

返回值	含义
`1`	`OS_ISR_FLAG` 当前为非零，即系统标志指示处于中断环境
`0`	`OS_ISR_FLAG` 当前为零，即系统标志指示处于任务环境

功能说明

查询内核维护的中断环境标志：

```
char os_in_isr(void)
{
    return OS_ISR_FLAG;
}
```

该函数本身不检测硬件中断状态，而是直接返回内核标志。

调用示例

```
if (os_in_isr())
{
    /* 中断环境下的处理 */
}
else
{
    /* 任务环境下的处理 */
}
```

注意事项

- 返回结果依赖于内核对 `OS_ISR_FLAG` 的维护。
- 不应仅凭该函数就推断所有中断嵌套场景下的行为；相关标志更新时机应结合内核中断入口与退出代码确认。

12.5.5 os_queue_send_from_isr

函数原型

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

参数

参数	类型	说明
`id`	`u8`	资源槽位编号
`_data`	`u8`	待写入的数据

返回值

返回值	含义
`1`	写入成功
`-1`	资源编号无效

功能说明

该接口检查资源编号后，直接更新对应资源的 `value`，增加 `pending_signal`，并设置调度请求标志。

核心操作如下：

```
OS_RES[id].value = _data;
OS_RES[id].pending_signal++;
OS_SCHED_REASON = 1;
```

该实现不会在函数内部遍历等待任务，也不会直接调用任务唤醒函数。

调用示例

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

注意事项

- 资源编号必须小于 `OS_RESOURCE_MAX`。
- 接口实际操作的是 `OS_RES[id]` 资源槽位；数据以单个字节形式写入。
- 连续调用会覆盖 `value`，同时分别增加 `pending_signal`。因此，不能将该接口直接视为能够保存任意数量消息的 FIFO 邮箱。
- 函数没有自行关闭全局中断，也没有调用 `os_schedule_request()`。
- 数据写入和调度标志设置后的具体任务唤醒行为，取决于内核对 `pending_signal` 和 `OS_SCHED_REASON` 的处理。

12.5.6 os_schedule_request

函数原型

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

参数

参数	类型	说明
`id`	`char`	当前实现保留的参数，不参

参数	类型	说明
		与调度判断

返回值

当前实现正常执行时返回 `1`。

功能说明

设置调度请求标志，并通过设置 Timer0 溢出标志请求进入 Timer0 中断处理：

```
id = id;
OS_SCHED_REASON = 1;
TF0 = 1;

return 1;
```

该接口本身不直接调用调度器，也不直接切换任务。

调用示例

```
os_schedule_request(0);
```

注意事项

- `id` 当前没有实际功能，只是保留参数。
- 设置 `TF0` 是请求 Timer0 中断处理的方式，实际执行时机取决于中断使能状态和内核中断处理流程。
- 函数没有自行保存和恢复 `EA`。
- 不应仅根据此函数就断言调用后会立即发生任务切换。

12.5.7 os_event_set_from_isr

函数原型

```
char os_event_set_from_isr(char obj);
```

参数

参数	类型	说明
`obj`	`char`	事件资源槽位编号

返回值

返回值	含义
`1`	事件设置操作完成
`-1`	资源编号无效

功能说明

该接口将指定资源的事件值设置为 `1`，将 `pending_signal` 设置为当前 `wait_cnt`，并设置调度请求标志。

核心操作如下：

```
OS_RES[obj].value = 1;
OS_RES[obj].pending_signal = OS_RES[obj].wait_cnt;
OS_SCHED_REASON = 1;
```

调用示例

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

注意事项

- 资源编号必须小于 `OS_RESOURCE_MAX`。
- 事件值被设置为 `1`，形成保持状态；该函数本身不会清除事件值。
- `pending_signal` 是被赋值为 `wait_cnt`，而不是执行自增操作。
- 该函数没有直接遍历等待任务或调用唤醒函数。
- 具体有多少任务被唤醒，以及等待任务如何处理，应以相关内核调度与资源处理代码为准。
- 函数没有自行关闭或恢复全局中断。

12.5.8 os_semaphore_post_from_isr

函数原型

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

参数

参数	类型	说明
`obj`	`u8`	信号量对应的资源槽位编号

返回值

返回值	含义
`1`	操作完成
`-1`	资源编号无效

功能说明

该接口根据资源当前的等待任务数量，采取不同处理方式。

存在等待任务时

当 `OS_RES[obj].wait_cnt > 0` 时，函数增加待处理信号计数，并设置调度请求标志：

```
OS_RES[obj].pending_signal++;
OS_SCHED_REASON = 1;
```

此分支不会直接增加 `value`，也不会直接唤醒任务。

没有等待任务时

当 `wait_cnt == 0` 时，如果资源值小于 `255`，则将其加一：

```
if (OS_RES[obj].value < 255)
{
    OS_RES[obj].value++;
}
```

因此，资源值的最大值为 `255`，达到上限后不会继续增加。

调用示例

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

注意事项

- 资源编号必须小于 `OS_RESOURCE_MAX`。
- 有等待任务时，函数只记录待处理信号并提交调度请求，后续行为由内核处理。
- 没有等待任务时，资源值最多增加至 `255`。
- 函数没有自行关闭或恢复全局中断。
- 该实现的实际语义应结合信号量等待接口及内核对 `pending_signal` 的处理共同理解。

12.5.9 os_msgq_send_from_isr

函数原型

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

参数

参数	类型	说明
`q`	`os_msgq_t *`	消息队列控制块指针
`obj`	`u8`	与消息队列等待处理相关联的资源槽位编号
`_data`	`u8`	待发送的单字节数据

返回值

返回值	含义
`1`	数据写入消息队列成功
`-1`	队列指针为空、资源编号无效或队列已满

功能说明

该接口在中断环境中向环形消息队列写入一个字节。它依次检查队列指针、资源编号和队列容量；任一检查失败，立即返回 `-1`。

当队列有空闲空间时，函数执行以下操作：

```
q->buf[q->head] = _data;
q->head = (u8)((q->head + 1) % q->_size);
q->count++;
```

随后，如果对应资源的 `wait_mask` 非零，函数会扫描任务数组，查找第一个 `wait_type == WAIT_MSG_RECV` 的任务。找到后设置调度请求标志、增加该资源的 `pending_signal`，然后结束扫描。

调用示例

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

注意事项

- `q` 不能为 `NULL`。
- `obj` 必须小于 `OS_RESOURCE_MAX`。
- 队列已满时，函数立即返回，不执行阻塞等待。
- 该函数会更新环形队列的 `head` 和 `count`，并在匹配条件满足时提交调度请求。
- 当前实现中的任务扫描条件只检查 `OS_TASK[i].wait_type == WAIT_MSG_RECV`，并没有同时检查该任务是否属于 `OS_RES[obj].wait_mask` 对应的等待集合。因此，不能据此保证被扫描到的任务一定在等待当前消息队列。
- 当前代码只记录待处理信号并提交调度请求，不直接调用任务唤醒函数。
- 实现采用无锁操作，并在注释中假设单中断上下文。其并发安全边界取决于实际中断嵌套、任务访问方式及消息队列使用约束。

12.6 临界区使用建议

临界区用于保护需要避免中断并发访问的关键操作。建议保持进入与退出严格配对，并尽量缩短关闭中断的时间。

```
os_enter_critical();

/* 访问需要保护的共享状态 */

os_exit_critical();
```

如果使用 `OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()` 宏，应确认其定义与当前实现一致。

12.7 中断与任务通信的使用原则

中断服务程序通常需要快速完成必要处理，再将后续工作交由任务执行。根据本模块提供的实现，可以采用以下原则：

- **单字节资源写入**：使用 `os_queue_send_from_isr()`，但要注意它覆盖资源值，并非 FIFO 队列。
- **事件通知**：使用 `os_event_set_from_isr()` 设置事件状态。
- **信号量释放**：使用 `os_semaphore_post_from_isr()` 记录释放信号或增加资源计数。

- **消息队列发送**：使用 `os_msgq_send_from_isr()` 将数据写入消息队列。
- **调度请求**：需要请求内核重新检查调度条件时，可使用 `os_schedule_request()`。

这些接口并不都采用相同的中断保护方式，也并非全部直接唤醒任务。调用者应根据具体接口的返回值和实现语义处理异常情况。

12.8 使用说明

- **中断上下文**：`_from_isr` 接口用于在中断服务程序中执行相应的资源操作。调用前应确认接口的适用范围，并遵循系统规定的中断优先级与嵌套规则。
- **调度请求**：`os_schedule_request()` 用于请求内核处理调度，不代表调用后立即发生任务切换。实际切换时机取决于中断处理流程和内核调度机制。
- **资源操作差异**：不同的 ISR 接口分别面向消息队列、事件、信号量等资源，其状态更新和任务唤醒行为可能不同，应按照各自的接口约定使用。
- **任务唤醒与调度**：中断中的资源操作可能使等待任务具备继续运行的条件，但任务何时恢复执行取决于内核的唤醒与调度流程。
- **中断状态查询**：`os_interrupt_id()` 用于获取当前中断标识，`os_in_isr()` 用于查询当前是否处于中断上下文。使用这些接口时，应遵循系统对中断状态和调用上下文的定义。
- **并发与共享资源**：中断服务程序与任务可能共同访问内核资源。设计相关逻辑时，应遵循系统的中断管理和同步约定，不应默认所有 ISR 接口均具备适用于任意嵌套场景的完整并发保护。