

## Coop

# 1 HRTOS Coop 模块

## 1.1 模块介绍

Coop 是 HRTOS 提供的轻量级协作式调度示例，用于展示任务主动让出 CPU 的调度模型。

### 1.1.1 什么是协作式调度

协作式调度（Cooperative Scheduling）是一种任务调度机制，其中任务必须主动释放 CPU 控制权，调度器不会强制剥夺正在运行任务的 CPU 使用权。任务在完成当前工作或需要等待时，通过调用调度函数主动让出 CPU，调度器再选择下一个就绪任务执行。

### 1.1.2 为什么需要协作式调度

协作式调度具有以下特点：

- **任务主动释放 CPU**：任务通过显式调用调度函数来切换到其他任务
- **调度行为简单**：无需复杂的上下文保存和恢复机制
- **实现成本低**：代码量小，资源占用少
- **适合资源受限系统**：适用于 RAM 和 Flash 资源极其有限的 MCU

这种调度模型特别适合：

- 简单的嵌入式控制系统
- 学习 RTOS 调度原理
- 不需要严格实时性的应用场景
- 资源极低的 8051 等 MCU 平台

---

## 1.2 Coop 架构

Coop 模块采用简洁的三层架构：



## 1.2.1 文件职责

- \*\*os.c\*\*** - 调度核心实现
    - 任务管理（任务表维护）
    - 调度器核心逻辑
    - 任务切换实现
    - 临界区保护
    - 延时唤醒机制
  - \*\*os.h\*\*** - 公开接口定义
    - 任务控制块（TCB）定义
    - 系统常量定义
    - 公开 API 声明
  - \*\*main.c\*\*** - 使用示例
    - 多任务 LED 控制演示
    - 任务创建和调度流程
    - `os_wait()` 延时使用示例
- 

## 1.3 协作式调度原理

### 1.3.1 任务执行流程

```
任务 A 运行
↓
主动让出 CPU (调用 os_schedule)
↓
调度器选择任务 B
↓
任务 B 运行
```

在 Coop 中，任务切换完全由任务自身控制。当一个任务调用 `os_schedule()` 时，调度器会从就绪位图中选择下一个任务执行。

### 1.3.2 与抢占式调度的区别

- | 项目    | 协作式调度          | 抢占式调度          |
|-------|----------------|----------------|
| 切换时机  | 任务主动让出 CPU     | 定时器中断或高优先级任务抢占 |
| 实现复杂度 | 低（简单轮询）        | 高（需完整上下文保存/恢复） |
| 响应能力  | 依赖任务主动让出       | 实时响应，可保证截止时间   |
| 应用场景  | 简单控制、学习、资源受限系统 | 复杂实时系统、多优先级任务  |
| 任务公平性 | 依赖任务自觉性        | 调度器保证公平性       |
-

## 1.4 调度流程

### 1.4.1 完整调度流程

#### 1.4.1.1 系统初始化

```
os_init();
```

初始化操作包括：

- 清空就绪位图 `os\_ready\_bitmap = 0`
- 设置当前任务 ID `os\_current = 0`
- 初始化临界区计数器
- 配置定时器中断 (Timer0)
- 清空所有任务的等待计数器

#### 1.4.1.2 创建任务

```
os_task_create(task_led0, 0, 0, 0);
```

创建任务时：

- 在任务表中记录任务 ID 和入口地址
- 将任务 ID 对应的位设置为就绪状态
- 任务进入就绪队列

#### 1.4.1.3 任务运行

任务开始执行，执行具体的应用逻辑（如 LED 控制）。

#### 1.4.1.4 主动切换

```
os_schedule();
```

任务完成当前工作后，主动调用调度函数：

- 调度器从当前任务 ID 的下一个位置开始轮询
- 在就绪位图中查找第一个就绪任务
- 找到后更新 `os\_current`
- 通过修改栈指针 (SP) 和返回地址实现任务切换

#### 1.4.1.5 下一个任务执行

调度器将 PC 指针指向新任务的入口地址，新任务开始运行。

### 1.4.2 调度算法

Coop 采用简单的轮询调度算法：

- 从 `os\_current + 1` 开始遍历任务表
- 找到第一个就绪的任务（位图对应位为 1）
- 如果遍历完所有任务仍未找到，则回到 ID 为 0 的空闲任务
- 这种方式保证了所有就绪任务的公平执行

---

## 1.5 API 参考

### 1.5.1 os\_init

**\*\*功能\*\***：系统初始化，配置调度器和定时器

**\*\*函数原型\*\***：

```
void os_init(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*示例\*\***：

```
os_init();  
---
```

### 1.5.2 os\_task\_create

**\*\*功能\*\***：创建任务，将任务加入就绪队列

**\*\*函数原型\*\***：

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

**\*\*参数\*\***：

- `addr`：任务入口地址（函数指针）

- `id`：任务 ID（0-15）

- `pr`：优先级（当前版本未使用）

- `sd`：栈深度（当前版本未使用）

**\*\*返回值\*\***：

- 0：创建成功

**\*\*示例\*\***：

```
os_task_create(task_led0, 0, 0, 0);  
---
```

### 1.5.3 os\_schedule

**\*\*功能\*\***：任务调度，主动让出 CPU 给下一个就绪任务

**\*\*函数原型\*\***：

```
void os_schedule(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*示例\*\***：

```
void task_led0() {  
    P1 = 1;  
    busy_delay();  
    os_schedule(); // 主动让出 CPU  
}  
---
```

### 1.5.4 os\_task\_delete

**\*\*功能\*\***：删除指定任务，从就绪队列中移除

**\*\*函数原型\*\***：

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

**\*\*参数\*\***：

- `id`：要删除的任务 ID

**\*\*返回值\*\***：无

**\*\*注意\*\***：ID 为 0 的任务（空闲任务）不能删除

**\*\*示例\*\***：

```
os_task_delete(5); // 删除 ID 为 5 的任务  
---
```

### 1.5.5 os\_wait

**\*\*功能\*\***：任务延时，将当前任务挂起指定时间

**\*\*函数原型\*\***：

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

**\*\*参数\*\***：

- `type`：等待类型（当前版本未使用）

- `obj`：等待对象（当前版本未使用）

- `tick`：延时的 tick 数

**\*\*返回值\*\***：

- 0：成功

**\*\*说明\*\***：

- 调用后任务从就绪队列中移除

- 定时器中断会递减等待计数

- 计数归零后任务自动重新就绪

**\*\*示例\*\***：

```
os_wait(0, 0, 200); // 延时 200 个 tick  
os_schedule();  
---
```

### 1.5.6 os\_enter\_critical

**\*\*功能\*\***：进入临界区，关闭中断

**\*\*函数原型\*\***：

```
void os_enter_critical();
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*说明\*\***：支持嵌套调用，通过计数器管理

---

### 1.5.7 os\_exit\_critical

**\*\*功能\*\***：退出临界区，恢复中断

**\*\*函数原型\*\***：

```
void os_exit_critical();
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*说明\*\***：当临界区计数器归零时才真正开启中断

---

### 1.5.8 os\_alloc\_slot

**\*\*功能\*\***：查找空闲任务槽位（当前版本已注释）

**\*\*函数原型\*\***：

```
unsigned char os_alloc_slot(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：

- 空闲槽位 ID

- 0xFF：无空闲槽位

**\*\*注意\*\***：该函数在当前实现中被注释，任务 ID 由用户手动指定

---

## 1.6 Demo 使用说明

### 1.6.1 Demo 功能

示例程序展示了 15 个 LED 控制任务的协作式调度：

- 每个任务控制 P1 端口的一个 LED
- 任务轮流执行，产生 LED 流水灯效果
- task\_led13 演示了 `os\_wait()` 延时功能

### 1.6.2 任务定义

```
void task_led0() { P1 = 1; busy_delay(); os_schedule(); }
void task_led1() { P1 = 2; busy_delay(); os_schedule(); }
void task_led2() { P1 = 4; busy_delay(); os_schedule(); }
// ... 更多任务
void task_led13() { P1 = 2; busy_delay(); os_wait(0, 0, 200);
os_schedule(); }
```

每个任务执行三个步骤：

1. 设置 P1 端口值（点亮对应 LED）
2. 执行延时（busy\_delay）

3. 调用 `os\_schedule()` 让出 CPU

### 1.6.3 初始化流程

```
int main(void)
{
    SP = 0x30;           // 设置栈指针

    os_init();           // 系统初始化

    // 创建 15 个任务
    os_task_create(task_led0, 0, 0, 0);
    os_task_create(task_led1, 1, 0, 0);
    // ... 创建其他任务
    os_task_create(task_led14, 14, 0, 0);

    os_schedule();       // 启动调度

    while(1);           // 防止退出
}
```

### 1.6.4 运行效果

程序运行后：

- 15 个任务按创建顺序轮流执行
- P1 端口的 LED 依次点亮，形成流水灯效果
- task\_led13 在执行后会延时 200 个 tick 再重新就绪
- 由于任务主动让出 CPU，所有任务公平地分享 CPU 时间

---

## 1.7 Coop 使用场景

### 1.7.1 适合的场景

- **简单控制系统**：如 LED 控制、简单按键扫描
- **资源极低 MCU**：8051 等 RAM 和 Flash 资源极其有限的平台
- **学习 RTOS 原理**：理解任务调度、上下文切换的基本概念
- **无需复杂抢占的应用**：任务执行时间可控，不需要严格实时性
- **原型验证**：快速验证多任务协作逻辑

### 1.7.2 不适合的场景

- **高实时性复杂系统**：需要严格保证任务响应时间
- **多优先级任务**：需要优先级抢占调度
- **任务阻塞等待**：需要复杂的信号量、消息队列等 IPC 机制
- **长时间计算任务**：可能导致其他任务长时间得不到执行

---

## 1.8 Coop 与 HRTOS 关系

### 1.8.1 Coop 定位

Coop 是 HRTOS 提供的简化版调度模型，具有以下特点：

- 简化的协作式调度机制
- 用于理解任务协作和调度基本原理
- 代码量小，易于学习和移植
- 适合作为 RTOS 入门的起点

### 1.8.2 HRTOS 完整功能

完整的 HRTOS 系统提供更强大的功能：

- **\*\*完整抢占式调度\*\***：支持时间片轮转和优先级抢占
- **\*\*任务阻塞机制\*\***：支持任务挂起、延时等待
- **\*\*IPC 通信\*\***：信号量、消息队列、事件标志
- **\*\*Shell 命令\*\***：提供交互式命令行接口
- **\*\*Modbus 协议\*\***：支持工业通信协议
- **\*\*更丰富的 API\*\***：提供完整的 RTOS 服务接口

### 1.8.3 学习路径

建议学习路径：

1. 先学习 Coop 模块，理解基本的任务调度概念
2. 理解协作式调度的优缺点
3. 再学习完整 HRTOS 的抢占式调度
4. 掌握优先级、阻塞、IPC 等高级特性

Coop 是理解 RTOS 思想的入口模块，为学习完整 RTOS 打下基础。

---

## 1.9 常见问题

### 1.9.1 任务无法切换

**\*\*原因\*\***：任务中忘记调用 ``os_schedule()``

**\*\*解决\*\***：确保每个任务在适当位置调用 ``os_schedule()`` 主动让出 CPU

```
void my_task() {  
    // 任务逻辑  
    do_something();  
    os_schedule(); // 必须调用  
}
```

---

## 1.9.2 忘记主动释放 CPU

**\*\*现象\*\***：某个任务一直运行，其他任务得不到执行

**\*\*原因\*\***：任务执行时间过长或循环中未调用调度函数

**\*\*解决\*\***：在长时间循环中定期调用 `os\_schedule()`

```
void long_task() {  
    for(int i = 0; i < 1000; i++) {  
        process_item(i);  
        if(i % 10 == 0) {  
            os_schedule(); // 定期让出 CPU  
        }  
    }  
}
```

---

## 1.9.3 调度异常

**\*\*现象\*\***：程序跑飞或栈溢出

**\*\*原因\*\***：

- 栈指针设置不当
- 任务 ID 冲突
- 未正确初始化系统

**\*\*解决\*\***：

- 确保 `os\_init()` 在创建任务前调用
- 检查任务 ID 是否唯一
- 合理设置栈指针 SP

---

## 1.9.4 任务阻塞问题

**\*\*现象\*\***：调用 `os\_wait()` 后任务不再执行

**\*\*原因\*\***：

- 定时器未正确配置
- 延时时间过长
- 忘记在 `os\_wait()` 后调用 `os\_schedule()`

**\*\*解决\*\***：

```
os_wait(0, 0, 100); // 延时  
os_schedule();      // 必须调用调度函数
```

---

## 1.9.5 任务 ID 管理

**\*\*问题\*\***：如何管理任务 ID 避免冲突

**\*\*建议\*\***：

- 使用宏定义或枚举管理任务 ID
- 建立任务 ID 到任务的映射表
- 避免动态分配 ID，使用静态分配

```
#define TASK_LED0    0
#define TASK_LED1    1
#define TASK_SENSOR  2

os_task_create(task_led0, TASK_LED0, 0, 0);
os_task_create(task_led1, TASK_LED1, 0, 0);
os_task_create(task_sensor, TASK_SENSOR, 0, 0);
---
```

## 1.9.6 临界区使用

**\*\*问题\*\*：**何时需要使用临界区保护

**\*\*建议\*\*：**

- 在访问共享全局变量时使用
- 在修改就绪位图时使用
- 保持临界区尽可能短

```
os_enter_critical();
// 临界区代码
os_ready_bitmap |= (1 << id);
os_exit_critical();
```

## Modbus

# 2 HRTOS Modbus RTU 模块使用说明

## 2.1 模块简介

### 2.1.1 Modbus RTU 简介

Modbus RTU 是一种广泛应用于工业自动化领域的串行通信协议，采用主从模式通信，通过 CRC16 校验确保数据传输可靠性。该协议简单高效，适用于资源受限的嵌入式系统。

### 2.1.2 HRTOS Modbus 模块作用

HRTOS Modbus RTU 模块为 HRTOS 实时操作系统提供完整的 Modbus 从站功能实现，基于任务、事件和中断机制设计，充分利用 HRTOS 的多任务调度能力，实现高效的串行通信处理。

### 2.1.3 支持的应用场景

- 工业传感器数据采集
- PLC 设备通信
- 智能仪表接口
- 嵌入式设备远程控制
- 51 MCU 应用开发

### 2.1.4 模块特点

- 基于 HRTOS 任务调度和事件同步机制
- 支持中断驱动接收，降低 CPU 占用
- 完整的帧超时检测机制
- 支持多种 Modbus 功能码
- 资源占用小，适合 51 MCU 等资源受限平台
- 清晰的分层架构，易于移植和扩展

---

## 2.2 模块架构

## 2.2.1 整体结构

```
应用层
|
Modbus 任务 (modbus_task)
|
Modbus 协议处理 (modbus_slave_process)
|
帧管理 (modbus_frame)
|
寄存器管理 (modbus_register)
|
CRC 校验 (modbus_crc)
|
UART 接口 (modbus_port)
|
BSP 硬件抽象层 (bsp_uart)
```

## 2.2.2 文件职责说明

### 2.2.2.1 modbus.c

- Modbus 核心模块
- 提供 `modbus\_init()` 初始化接口
- 实现 `modbus\_process()` 周期处理函数
- 实现 `modbus\_frame\_parse()` 帧解析功能
- 负责地址、CRC、数据长度校验

### 2.2.2.2 modbus\_slave.c

- Modbus 从站协议处理
- 实现功能码分发和响应生成
- 支持功能码：0x01（读线圈）、0x03（读保持寄存器）、0x05（写单个线圈）、0x06（写单个寄存器）、0x10（写多个寄存器）
- 处理设备地址匹配
- 自动添加 CRC 并发送响应

### 2.2.2.3 modbus\_frame.c

- 帧接收和管理模块
- 实现 `modbus\_frame\_receive()` 逐字节接收
- 实现 `modbus\_frame\_timeout()` 帧超时检测（3.5 字符时间）
- 管理 `modbus\_rx\_buf` 接收缓冲区
- 通过事件机制通知处理任务

### 2.2.2.4 modbus\_register.c

- 寄存器和线圈管理模块
- 实现 `modbus\_reg\_read()` / `modbus\_reg\_write()` 保持寄存器读写
- 实现 `modbus\_coil\_read()` / `modbus\_coil\_write()` 线圈读写
- 使用静态数组存储寄存器和线圈数据

- 提供地址范围保护

#### 2.2.2.5 modbus\_crc.c

- CRC16 校验模块
- 实现 `modbus\_crc16()` 函数
- 采用 Modbus 标准 CRC16 算法（多项式 0xA001）
- 用于帧校验和响应生成

#### 2.2.2.6 modbus\_port.c

- 硬件端口抽象层
- 实现 `modbus\_port\_init()` 端口初始化
- 实现 `modbus\_port\_send()` 数据发送
- 调用 BSP 层 UART 接口

---

## 2.3 HRTOS 集成方式

### 2.3.1 任务结构

#### 2.3.1.1 modbus\_task()

- \*\*作用\*\*：**Modbus 主处理任务
- 初始化 Modbus 模块
  - 等待接收事件（`MODBUS\_EVENT\_RX`）
  - 调用 `modbus\_process()` 处理协议帧
  - 删除事件标志，循环等待

#### 2.3.1.2 modbus\_timer\_task()

- \*\*作用\*\*：**定时超时检测任务
- 周期执行（`os\_delay\_ms(1)`）
  - 调用 `modbus\_frame\_timeout()` 检测帧接收超时
  - 实现 3.5 字符时间超时判断

#### 2.3.1.3 uart\_isr()

- \*\*作用\*\*：**UART 中断服务程序
- 响应 UART 接收中断
  - 读取接收到的字节
  - 调用 `modbus\_frame\_receive()` 存入接收缓冲区
  - 处理发送完成中断

### 2.3.2 完整数据流程

UART 接收中断

↓

```

uart_isr() 读取字节
↓
modbus_frame_receive(ch) 存入缓冲区
↓
modbus_timer_task() 周期检测超时
↓
超时后触发 MODBUS_EVENT_RX 事件
↓
modbus_task() 被唤醒
↓
modbus_process() 处理
↓
modbus_frame_parse() 解析帧 (地址、CRC、长度)
↓
modbus_slave_process() 根据功能码处理
↓
modbus_reg_read/write() 或 modbus_coil_read/write()
↓
modbus_slave_send() 生成响应并添加 CRC
↓
modbus_port_send() 通过 UART 发送响应
---

```

## 2.4 快速开始

### 2.4.1 最小使用流程

根据 `main.c` 中的官方 Demo，完整的使用流程如下：

#### 2.4.1.1 步骤 1：UART 初始化

```
bsp_uart_init();
```

#### 2.4.1.2 步骤 2：事件初始化

```
os_event_init(MODBUS_EVENT_RX);
```

#### 2.4.1.3 步骤 3：创建 Modbus 任务

```

// 创建定时器任务 (用于超时检测)
os_task_create(
    modbus_timer_task,
    2,           // 任务 ID
    2,           // 优先级
    6            // 堆栈大小
);

// 创建 Modbus 主处理任务
os_task_create(
    modbus_task,
    1,           // 任务 ID
    3,           // 优先级
    6            // 堆栈大小
);

// 创建 UART 中断任务
os_task_create(
    uart_isr,
    0,           // 任务 ID
    9,           // 优先级 (最高)
    0            // 堆栈大小 (中断任务)
);

```

```
);
```

#### 2.4.1.4 步骤 4：启动系统

```
void hrtos_main(void)
{
    os_slab_overflow_register(modbus_overflow);
    bsp_uart_init();
    os_event_init(MODBUS_EVENT_RX);

    os_task_create(modbus_timer_task, 2, 2, 6);
    os_task_create(modbus_task, 1, 3, 6);
    os_task_create(uart_isr, 0, 9, 0);
}
```

#### 2.4.2 完整示例代码

以下代码来自当前工程 `Src/main.c`：

```
void modbus_timer_task(void)
{
    while(1)
    {
        os_delay_ms(1);
        modbus_frame_timeout();
    }
}

void modbus_task(void)
{
    modbus_init();

    while(1)
    {
        os_event_wait(MODBUS_EVENT_RX, 0);
        modbus_process();
        os_event_delete(MODBUS_EVENT_RX);
    }
}

void uart_isr(void)
{
    unsigned char ch;
    char id;

    id = os_interrupt_id();
    if(id != 4)
    {
        os_interrupt_exit();
    }

    if(RI)
    {
        RI = 0;
        ch = SBUF;
        modbus_frame_receive(ch);
    }

    if(TI)
    {
        TI = 0;
    }
}
```

```

        os_interrupt_exit();
    }

void hrtos_main(void)
{
    os_slab_overflow_register(modbus_overflow);
    bsp_uart_init();
    os_event_init(MODBUS_EVENT_RX);

    os_task_create(modbus_timer_task, 2, 2, 6);
    os_task_create(modbus_task, 1, 3, 6);
    os_task_create(uart_isr, 0, 9, 0);
}

---
```

## 2.5 配置说明

### 2.5.1 配置文件

所有配置项定义在 `Inc/modbus\_config.h` 中。

### 2.5.2 配置项列表

| 配置宏                                     | 默认值   | 说明                 |
|-----------------------------------------|-------|--------------------|
| -----                                   | ----- | -----              |
| <b>**MODBUS_SLAVE_ADDR**</b>            | 1     | Modbus 从机地址        |
| <b>**MODBUS_RX_BUF_SIZE**</b>           | 68    | 接收缓冲区大小（字节）        |
| <b>**MODBUS_TX_BUF_SIZE**</b>           | 32    | 发送缓冲区大小（字节）        |
| <b>**MODBUS_REG_MAX**</b>               | 32    | 保持寄存器最大数量          |
| <b>**MODBUS_FUNC_READ_HOLDING_REG**</b> | 1     | 开启功能码 0x03（读保持寄存器） |
| <b>**MODBUS_FUNC_WRITE_SINGLE_REG**</b> | 1     | 开启功能码 0x06（写单个寄存器） |
| <b>**MODBUS_FUNC_WRITE_MULTI_REG**</b>  | 1     | 开启功能码 0x10（写多个寄存器） |
| <b>**MODBUS_COIL_MAX**</b>              | 8     | 线圈最大数量             |

### 2.5.3 配置说明

#### 2.5.3.1 从机地址配置

```
#define MODBUS_SLAVE_ADDR 1
```

设置 Modbus 从站地址，支持广播地址 0。

#### 2.5.3.2 缓冲区配置

```
#define MODBUS_RX_BUF_SIZE 68
#define MODBUS_TX_BUF_SIZE 32
```

根据实际通信数据量和系统内存调整。

### 2.5.3.3 寄存器配置

```
#define MODBUS_REG_MAX 32
```

定义保持寄存器数量，地址范围 0 ~ MODBUS\_REG\_MAX-1。

### 2.5.3.4 功能码配置

```
#define MODBUS_FUNC_READ_HOLDING_REG 1
```

```
#define MODBUS_FUNC_WRITE_SINGLE_REG 1
```

```
#define MODBUS_FUNC_WRITE_MULTI_REG 1
```

设置为 1 启用对应功能码，设置为 0 禁用。

### 2.5.3.5 线圈配置

```
#define MODBUS_COIL_MAX 8
```

定义线圈数量，1 bit 表示一个线圈。

---

## 2.6 API 参考

### 2.6.1 modbus.h

#### 2.6.1.1 modbus\_init()

**\*\*功能\*\***：初始化 Modbus 协议模块

**\*\*函数原型\*\***：

```
void modbus_init(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
void modbus_task(void)
{
    modbus_init();
    while(1)
    {
        os_event_wait(MODBUS_EVENT_RX, 0);
        modbus_process();
        os_event_delete(MODBUS_EVENT_RX);
    }
}
```

#### 2.6.1.2 modbus\_process()

**\*\*功能\*\***：Modbus 周期处理，由任务周期调用

**\*\*函数原型\*\***：

```
void modbus_process(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
os_event_wait(MODBUS_EVENT_RX, 0);
modbus_process();
```

#### 2.6.1.3 modbus\_coil\_read()

**\*\*功能\*\*：**读取线圈状态

**\*\*函数原型\*\*：**

```
unsigned char modbus_coil_read(unsigned int addr);
```

**\*\*参数\*\*：**

- `addr`：线圈地址

**\*\*返回值\*\*：**

- 0：线圈 OFF

- 1：线圈 ON

**\*\*使用示例\*\*：**

```
unsigned char coil_status = modbus_coil_read(0);
```

#### 2.6.1.4 modbus\_coil\_write()

**\*\*功能\*\*：**写入线圈状态

**\*\*函数原型\*\*：**

```
void modbus_coil_write(unsigned int addr, unsigned char value);
```

**\*\*参数\*\*：**

- `addr`：线圈地址

- `value`：线圈值（0=OFF, 1=ON）

**\*\*返回值\*\*：**无

**\*\*使用示例\*\*：**

```
modbus_coil_write(0, 1); // 设置线圈 0 为 ON
```

#### 2.6.1.5 modbus\_frame\_timeout()

**\*\*功能\*\*：**帧超时检测，在定时任务中周期调用

**\*\*函数原型\*\*：**

```
void modbus_frame_timeout(void);
```

**\*\*参数\*\*：**无

**\*\*返回值\*\*：**无

**\*\*使用示例\*\*：**

```
void modbus_timer_task(void)
{
    while(1)
    {
        os_delay_ms(1);
        modbus_frame_timeout();
    }
}
```

---

### 2.6.2 modbus\_frame.h

#### 2.6.2.1 modbus\_frame\_init()

**\*\*功能\*\*：**初始化帧处理模块

**\*\*函数原型\*\*：**

```
void modbus_frame_init(void);
```

**\*\*参数\*\*：**无

**\*\*返回值\*\*：**无

**\*\*使用示例\*\*：**

```
modbus_frame_init();
```

#### 2.6.2.2 modbus\_frame\_receive()

**\*\*功能\*\*：**接收一个字节数据

**\*\*函数原型\*\*：**

```
unsigned char modbus_frame_receive(unsigned char ch);
```

**\*\*参数\*\*：**

- `ch`：接收到的字节

**\*\*返回值\*\*：**

- 0：未完成

- 1：收到完整帧

**\*\*使用示例\*\*：**

```
void uart_isr(void)
{
    if(RI)
    {
        RI = 0;
        ch = SBUF;
        modbus_frame_receive(ch);
    }
}
```

#### 2.6.2.3 modbus\_frame\_parse()

**\*\*功能\*\*：**解析接收到的数据帧

**\*\*函数原型\*\*：**

```
unsigned char modbus_frame_parse(void);
```

**\*\*参数\*\*：**无

**\*\*返回值\*\*：**

- 1：解析成功

- 0：解析失败（CRC 错误、长度错误等）

**\*\*使用示例\*\*：**

```
if(modbus_frame_parse())
{
    modbus_slave_process(modbus_frame_get());
}
```

#### 2.6.2.4 modbus\_frame\_get()

**\*\*功能\*\*：**获取当前数据帧

**\*\*函数原型\*\*：**

```
modbus_frame_t *modbus_frame_get(void);
```

**\*\*参数\*\*：**无

**\*\*返回值\*\***：指向当前帧结构的指针

**\*\*使用示例\*\***：

```
modbus_frame_t *frame = modbus_frame_get();  
---
```

## 2.6.3 modbus\_port.h

### 2.6.3.1 modbus\_port\_init()

**\*\*功能\*\***：Modbus 端口初始化

**\*\*函数原型\*\***：

```
void modbus_port_init(void);
```

**\*\*参数\*\***：无

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
modbus_port_init();
```

### 2.6.3.2 modbus\_port\_send()

**\*\*功能\*\***：发送 Modbus 数据

**\*\*函数原型\*\***：

```
void modbus_port_send(unsigned char *buf, unsigned int len);
```

**\*\*参数\*\***：

- `buf`：数据缓冲区指针

- `len`：数据长度

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
unsigned char data[] = {0x01, 0x03, 0x00, 0x00};  
modbus_port_send(data, 4);
```

### 2.6.3.3 modbus\_port\_recv()

**\*\*功能\*\***：接收一个字节

**\*\*函数原型\*\***：

```
unsigned char modbus_port_recv(unsigned char *ch);
```

**\*\*参数\*\***：

- `ch`：接收字节指针

**\*\*返回值\*\***：接收状态

**\*\*使用示例\*\***：

```
unsigned char ch;  
if(modbus_port_recv(&ch))  
{  
    // 处理接收到的字节  
}
```

### 2.6.3.4 modbus\_port\_rx\_byte()

**\*\*功能\*\***：处理接收字节

**\*\*函数原型\*\*:**

```
unsigned char modbus_port_rx_byte(unsigned char ch);
```

**\*\*参数\*\*:**

- `ch`: 接收到的字节

**\*\*返回值\*\*:** 处理状态

**\*\*使用示例\*\*:**

```
modbus_port_rx_byte(ch);
```

---

## 2.6.4 modbus\_register.h

### 2.6.4.1 modbus\_reg\_read()

**\*\*功能\*\*:** 读取保持寄存器

**\*\*函数原型\*\*:**

```
unsigned int modbus_reg_read(unsigned char addr);
```

**\*\*参数\*\*:**

- `addr`: 寄存器地址

**\*\*返回值\*\*:** 寄存器值

**\*\*使用示例\*\*:**

```
unsigned int value = modbus_reg_read(0);
```

### 2.6.4.2 modbus\_reg\_write()

**\*\*功能\*\*:** 写入保持寄存器

**\*\*函数原型\*\*:**

```
void modbus_reg_write(unsigned char addr, unsigned int value);
```

**\*\*参数\*\*:**

- `addr`: 寄存器地址

- `value`: 写入值

**\*\*返回值\*\*:** 无

**\*\*使用示例\*\*:**

```
modbus_reg_write(0, 0x1234);
```

---

## 2.6.5 modbus\_slave.h

### 2.6.5.1 modbus\_slave\_process()

**\*\*功能\*\*:** Modbus 从站请求处理

**\*\*函数原型\*\*:**

```
void modbus_slave_process(modbus_frame_t *frame);
```

**\*\*参数\*\*:**

- `frame`: 指向 Modbus 帧结构的指针

**\*\*返回值\*\*:** 无

**\*\*使用示例\*\*:**

```
if(modbus_frame_parse())
{
    modbus_slave_process(modbus_frame_get());
}
```

---

## 2.6.6 modbus\_crc.h

### 2.6.6.1 modbus\_crc16()

**\*\*功能\*\*:** 计算 Modbus CRC16 校验值

**\*\*函数原型\*\*:**

```
unsigned int modbus_crc16(unsigned char *data1, unsigned int len);
```

**\*\*参数\*\*:**

- `data1`: 数据缓冲区指针

- `len`: 数据长度

**\*\*返回值\*\*:** CRC16 校验值

**\*\*使用示例\*\*:**

```
unsigned char data[] = {0x01, 0x03, 0x00, 0x00};
unsigned int crc = modbus_crc16(data, 4);
```

---

## 2.7 Modbus 寄存器管理

### 2.7.1 寄存器定义

Modbus 模块使用静态数组存储保持寄存器:

```
static unsigned int holding_reg[MODBUS_REG_MAX];
```

寄存器地址范围: 0 ~ MODBUS\_REG\_MAX-1 (默认 0 ~ 31)

### 2.7.2 添加方式

寄存器在编译时通过配置宏确定数量, 无需动态添加。用户可通过 `modbus\_reg\_write()` 预设寄存器初始值。

### 2.7.3 数据读取

使用 `modbus\_reg\_read()` 读取寄存器值:

```
unsigned int value = modbus_reg_read(addr);
```

- 地址超出范围返回 0

- 返回值为 16 位无符号整数

### 2.7.4 数据写入

使用 `modbus\_reg\_write()` 写入寄存器值:

```
modbus_reg_write(addr, value);
```

- 地址超出范围不执行写入

- 写入值为 16 位无符号整数

### 2.7.5 用户扩展方法

如需扩展寄存器功能（如关联实际硬件），可修改 `modbus\_register.c` 中的读写函数：

```
unsigned int modbus_reg_read(unsigned char addr)
{
    if(addr >= MODBUS_REG_MAX)
    {
        return 0;
    }

    // 用户自定义：读取实际硬件值
    // holding_reg[addr] = read_hardware(addr);

    return holding_reg[addr];
}

void modbus_reg_write(unsigned char addr, unsigned int value)
{
    if(addr < MODBUS_REG_MAX)
    {
        holding_reg[addr] = value;

        // 用户自定义：写入实际硬件
        // write_hardware(addr, value);
    }
}

---
```

## 2.8 Modbus Slave 功能

### 2.8.1 从机初始化

从机初始化通过 `modbus\_init()` 完成，包括：

- 端口初始化（`modbus\_port\_init()`）
- 帧处理初始化（`modbus\_frame\_init()`）
- 设置初始化标志

### 2.8.2 请求处理

从机请求处理流程：

1. UART 中断接收数据
2. 帧超时检测触发事件
3. `modbus\_process()` 被调用
4. `modbus\_frame\_parse()` 解析帧（地址、CRC 校验）
5. `modbus\_slave\_process()` 根据功能码分发处理

### 2.8.3 数据响应流程

1. 根据功能码调用对应处理函数
2. 读取或写入寄存器/线圈
3. 生成响应帧（地址 + 功能码 + 数据）
4. 计算 CRC16 并附加到帧尾
5. 通过 `modbus\_port\_send()` 发送响应

## 2.8.4 支持功能码

| 功能码                     | 名称                       | 说明      | 支持状态 |
|-------------------------|--------------------------|---------|------|
| ----- ----- ----- ----- |                          |         |      |
| 0x01                    | Read Coils               | 读取线圈    | 支持   |
| 0x03                    | Read Holding Registers   | 读取保持寄存器 | 支持   |
| 0x05                    | Write Single Coil        | 写单个线圈   | 支持   |
| 0x06                    | Write Single Register    | 写单个寄存器  | 支持   |
| 0x10                    | Write Multiple Registers | 写多个寄存器  | 支持   |
| ---                     |                          |         |      |

## 2.9 移植说明

### 2.9.1 需要修改的文件

移植到不同 MCU 平台主要修改以下文件：

#### 2.9.1.1 modbus\_port.c

修改 UART 硬件接口调用：

```
void modbus_port_init(void)
{
    // 替换为您的 UART 初始化代码
    // bsp_uart_init();
}

void modbus_port_send(unsigned char *buf, unsigned int len)
{
    unsigned int i;
    for (i = 0; i < len; i++)
    {
        // 替换为您的 UART 发送函数
        // bsp_uart_send_byte(buf[i]);
    }
}
```

#### 2.9.1.2 BSP/bsp.c

实现 UART 硬件抽象层：

```
void bsp_uart_init(void)
{
    // 配置 UART 参数（波特率、数据位、停止位等）
    // 使能 UART 接收中断
}
```

```
void bsp_uart_send_byte(unsigned char dat)
{
    // 发送单字节数据
    // 等待发送完成
}
```

## 2.9.2 UART 发送

根据目标 MCU 的 UART 寄存器实现发送函数：

```
void bsp_uart_send_byte(unsigned char dat)
{
    // 写入 UART 数据寄存器
    UART_DR = dat;

    // 等待发送完成
    while(!(UART_SR & UART_SR_TXE));
}
```

## 2.9.3 UART 接收

在中断服务程序中读取接收到的字节：

```
void uart_isr(void)
{
    unsigned char ch;

    // 检查接收中断标志
    if(UART_SR & UART_SR_RXNE)
    {
        ch = UART_DR;
        modbus_frame_receive(ch);
    }

    // 退出中断
    os_interrupt_exit();
}
```

## 2.9.4 中断处理

确保 UART 接收中断已使能，并正确配置中断优先级：

```
void bsp_uart_init(void)
{
    // UART 配置
    // ...

    // 使能接收中断
    UART_CR |= UART_CR_RXNEIE;

    // 配置中断优先级
    NVIC_SetPriority(UART_IRQn, priority);

    // 使能中断
    NVIC_EnableIRQ(UART_IRQn);
}
```

## 2.9.5 定时检测

Modbus RTU 要求 3.5 字符时间的帧间隔检测。根据波特率计算定时周期：

```
void modbus_timer_task(void)
{
    while(1)
    {
        // 根据波特率调整延时
        // 例如 9600bps, 1 字符时间约 1ms
        // 3.5 字符时间约 3.5ms
        os_delay_ms(1); // 调整为合适的值
        modbus_frame_timeout();
    }
}
```

---

## 2.10 完整 Demo 说明

### 2.10.1 Demo 功能

官方 Demo 实现了完整的 Modbus RTU 从站功能，包括：

- UART 串口通信（9600bps）
- Modbus 帧接收和解析
- 支持功能码 0x01、0x03、0x05、0x06、0x10
- 寄存器和线圈读写
- CRC16 校验
- HRTOS 任务调度集成

### 2.10.2 硬件环境

- MCU：51 系列（如 STC89C52）
- UART：串口 1
- 波特率：9600bps（定时器 1 模式 2，重装值 0xFD）
- 中断：UART 接收中断

### 2.10.3 软件结构

```
hrtos_main()
├── bsp_uart_init()           // UART 初始化
├── os_event_init()          // 事件初始化
├── os_task_create()         // 创建定时器任务
├── os_task_create()         // 创建 Modbus 任务
└── os_task_create()         // 创建 UART 中断任务
```

### 2.10.4 任务列表

| 任务名称 | 优先级 | 堆栈大小 | 功能 |

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

| uart\_isr | 9 | 0 | UART 中断服务 |

| modbus\_timer\_task | 2 | 6 | 帧超时检测 |  
| modbus\_task | 3 | 6 | Modbus 协议处理 |

## 2.10.5 运行流程

1. 系统启动, 调用 `hrtos\_main()`
2. 初始化 UART 和事件
3. 创建三个任务
4. UART 中断任务接收数据
5. 定时器任务检测超时
6. Modbus 任务处理协议帧
7. 响应数据通过 UART 发送

## 2.10.6 关键代码

### 2.10.6.1 UART 配置 (BSP/bsp.c)

```
void bsp_uart_init(void)
{
    SCON = 0x50;           // 8 位数据, 可变波特率
    TMOD &= 0x0F;
    TMOD |= 0x20;          // 定时器 1 模式 2
    TL1 = 0xFD;            // 9600bps
    TH1 = 0xFD;
    ET1 = 0;               // 禁止定时器中断
    TR1 = 1;               // 定时器 1 开始计时
    EA = 1;
    ES = 1;                // 使能串口中断
}
```

### 2.10.6.2 中断接收 (Src/main.c)

```
void uart_isr(void)
{
    unsigned char ch;
    char id;

    id = os_interrupt_id();
    if(id != 4)
    {
        os_interrupt_exit();
    }

    if(RI)
    {
        RI = 0;
        ch = SBUF;
        modbus_frame_receive(ch);
    }

    if(TI)
    {
        TI = 0;
    }

    os_interrupt_exit();
}
```

```
}
```

### 2.10.6.3 任务创建 (Src/main.c)

```
void hrtos_main(void)
{
    os_slab_overflow_register(modbus_overflow);
    bsp_uart_init();
    os_event_init(MODBUS_EVENT_RX);

    os_task_create(modbus_timer_task, 2, 2, 6);
    os_task_create(modbus_task, 1, 3, 6);
    os_task_create(uart_isr, 0, 9, 0);
}
```

---

## 2.11 常见问题

### 2.11.1 无法通信

**\*\*可能原因\*\*:**

- 波特率不匹配
- 从机地址配置错误
- UART 硬件连接问题
- 中断未正确使能

**\*\*解决方法\*\*:**

- 检查主从机波特率是否一致
- 确认 `MODBUS\_SLAVE\_ADDR` 配置正确
- 检查 TX/RX 引脚连接
- 确认 UART 中断已使能 (`ES=1`)

### 2.11.2 CRC 错误

**\*\*可能原因\*\*:**

- 数据传输错误
- 噪声干扰
- 波特率偏差

**\*\*解决方法\*\*:**

- 检查通信线路质量
- 降低波特率
- 增加硬件滤波
- 检查 CRC 计算实现

### 2.11.3 地址错误

**\*\*可能原因\*\*:**

- 寄存器地址超出范围

- 线圈地址超出范围

- 从机地址不匹配

**\*\*解决方法\*\*:**

- 确认访问地址在 `MODBUS\_REG\_MAX` 范围内

- 确认线圈地址在 `MODBUS\_COIL\_MAX` 范围内

- 检查主站发送的从机地址

#### **2.11.4 UART 配置错误**

**\*\*可能原因\*\*:**

- 定时器配置错误

- 波特率计算错误

- 中断优先级冲突

**\*\*解决方法\*\*:**

- 检查定时器模式和重装值

- 根据晶振频率计算正确的波特率

- 调整中断优先级

#### **2.11.5 超时问题**

**\*\*可能原因\*\*:**

- 定时任务周期不合适

- 帧超时阈值配置错误

- 任务调度延迟

**\*\*解决方法\*\*:**

- 调整 `modbus\_timer\_task()` 的延时周期

- 修改 `MODBUS\_FRAME\_TIMEOUT` 配置

- 检查 HRTOS 任务调度配置

#### **2.11.6 数据格式错误**

**\*\*可能原因\*\*:**

- 字节序错误

- 数据长度不匹配

- 功能码不支持

**\*\*解决方法\*\*:**

- Modbus RTU 使用大端序（高字节在前）

- 检查请求帧的数据长度字段

- 确认主站使用的功能码已启用

---

## 2.12 附录

### 2.12.1 A. 数据帧结构

#### 2.12.1.1 请求帧格式

|      |     |    |       |       |
|------|-----|----|-------|-------|
| 从机地址 | 功能码 | 数据 | CRC_L | CRC_H |
|------|-----|----|-------|-------|

#### 2.12.1.2 响应帧格式

|      |     |    |       |       |
|------|-----|----|-------|-------|
| 从机地址 | 功能码 | 数据 | CRC_L | CRC_H |
|------|-----|----|-------|-------|

### 2.12.2 B. 功能码示例

#### 2.12.2.1 0x03 读保持寄存器

**\*\*请求\*\*:** `01 03 00 00 00 02 CRC`

- 地址: 01

- 功能码: 03

- 起始地址: 0000

- 寄存器数量: 0002

**\*\*响应\*\*:** `01 03 04 12 34 56 78 CRC`

- 地址: 01

- 功能码: 03

- 字节数: 04

- 数据: 1234 5678

#### 2.12.2.2 0x06 写单个寄存器

**\*\*请求\*\*:** `01 06 00 10 00 64 CRC`

- 地址: 01

- 功能码: 06

- 寄存器地址: 0010

- 写入值: 0064

**\*\*响应\*\*:** `01 06 00 10 00 64 CRC`

- 原样返回请求内容

### 2.12.3 C. 资源占用

| 资源类型 | 占用 |
|------|----|
|------|----|

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

|     |                     |
|-----|---------------------|
| RAM | 约 150 字节（缓冲区 + 寄存器） |
|-----|---------------------|

|     |            |
|-----|------------|
| ROM | 约 3 KB（代码） |
|-----|------------|

| 任务数量 | 3 个 |

| 事件数量 | 1 个 |

---

**\*\*更新日期\*\***: 2026-07-23

**\*\*适用版本\*\***: HRTOS Modbus RTU 模块

## Myos

# 3 HRTOS Myos 轻量级调度模块说明

## 3.1 模块介绍

Myos 是 HRTOS 提供的轻量级任务调度实验模块，用于展示任务切换、上下文保存和调度机制。

### 3.1.1 为什么设计 Myos

Myos 旨在以最简化的方式展示 RTOS 最核心的机制，帮助开发者理解任务调度的基本原理。通过剥离完整内核的复杂功能，Myos 专注于以下核心概念：

- **任务管理**：如何创建和管理多个任务
- **调度选择**：如何在任务之间进行切换
- **上下文保存**：如何保存当前任务的执行状态
- **上下文恢复**：如何恢复任务的执行现场

### 3.1.2 Myos 解决的问题

Myos 通过简化的实现解决了以下问题：

1. **多任务并发执行**：在单核 51 单片机上实现多任务轮转调度
2. **任务状态保存**：在中断发生时保存当前任务的堆栈和寄存器状态
3. **任务切换**：在不同任务之间平滑切换，保证每个任务都能获得执行机会
4. **中断处理**：统一管理外部中断和系统时钟中断

### 3.1.3 与完整 HRTOS 内核的关系

Myos 是 HRTOS 完整内核的简化版本，专注于调度核心机制的学习和验证。HRTOS 完整内核在 Myos 的基础上扩展了更多功能，如任务间通信（IPC）、事件等待、Shell 命令行、Modbus 协议栈等。Myos 为理解这些高级功能奠定了基础。

---

## 3.2 Myos 整体架构

Myos 采用分层架构设计，从应用层到硬件抽象层逐层封装，确保各层职责清晰。

### 3.2.1 架构层次

应用层 (App)  
↓  
调度层 (Kernel)  
↓  
任务上下文管理 (HAL)  
↓  
硬件抽象层 (HAL)

## 3.2.2 文件职责说明

### 3.2.2.1 App 层

**\*\*my\_main.c\*\***

- 应用程序入口
- 任务定义和注册
- 中断服务程序定义
- 系统初始化配置

### 3.2.2.2 HAL 层

**\*\*context\_jump.c\*\***

- 实现上下文跳转功能
- 修改函数返回地址，实现程序跳转

**\*\*context\_restore.c\*\***

- 恢复指定任务的上下文
- 恢复堆栈指针并退出中断

**\*\*task\_save\_sp.c\*\***

- 保存当前任务的堆栈指针
- 为任务切换准备数据

**\*\*get\_sp.c\*\***

- 获取当前堆栈指针值
- 用于堆栈溢出检测

**\*\*get\_current\_task\_id.c\*\***

- 获取当前运行任务的 ID
- 用于调度决策

**\*\*scheduler\_select\_next.c\*\***

- 选择下一个可运行任务
- 实现轮转调度策略

### 3.2.2.3 Kernel 层

**\*\*kernel\_tick\_handler.c\*\***

- 系统时钟中断处理入口
- 协调整个调度流程

- 管理中断向量表
- \*\*object\_register.c\*\***
- 注册任务或中断服务程序
- 管理任务和中断的映射关系
- \*\*tick\_stub.c\*\***
- 系统 Tick 占位函数
- 为中断响应提供时间窗口
- 

### 3.3 调度工作流程

Myos 的调度流程由定时器 T0 中断触发，通过保存当前任务状态、选择下一任务、恢复目标任务状态来实现任务切换。

#### 3.3.1 调度流程描述

##### 1. **\*\*Tick 产生\*\***

定时器 T0 溢出产生中断，触发 `os\_kernel\_tick\_handler()` 函数。

##### 2. **\*\*调度触发\*\***

内核检测到中断编号为 1（Timer0），启动调度流程。

##### 3. **\*\*选择下一任务\*\***

调用 `os\_scheduler\_select\_next()` 从当前任务的下一个任务开始循环查找，返回第一个处于就绪状态的任务 ID。

##### 4. **\*\*保存当前任务上下文\*\***

调用 `os\_task\_save\_sp()` 将当前堆栈指针保存到任务堆栈记录中。

##### 5. **\*\*恢复目标任务上下文\*\***

调用 `os\_context\_restore()` 恢复目标任务的堆栈指针，并通过 RETI 指令退出中断，CPU 自动恢复任务现场。

##### 6. **\*\*继续执行\*\***

目标任务从中断点继续执行。

#### 3.3.2 流程图



```
↓
恢复目标任务上下文 (os_context_restore)
↓
RETI 退出中断
↓
目标任务继续执行
---
```

## 3.4 上下文切换机制

HAL 层负责实现底层的上下文切换功能，确保任务切换时 CPU 状态的正确保存和恢复。

### 3.4.1 为什么需要保存 CPU 状态

在多任务系统中，每个任务都有独立的执行上下文，包括：

- 堆栈指针 (SP)
- 程序计数器 (PC)
- 通用寄存器
- 状态寄存器

当任务切换时，必须保存当前任务的这些状态，否则任务恢复时无法正确继续执行。

### 3.4.2 为什么需要恢复任务现场

恢复任务现场是保存的逆过程，通过恢复之前保存的堆栈指针和寄存器状态，使任务能够从中断点继续执行，就像从未被打断一样。

### 3.4.3 上下文切换在 RTOS 中的作用

上下文切换是 RTOS 实现多任务并发的核心机制：

- **时间片轮转**：每个任务执行一段时间后切换到下一个任务
- **优先级调度**：高优先级任务就绪时切换到高优先级任务
- **中断响应**：中断处理完成后切换回被中断的任务

### 3.4.4 HAL 层核心函数

#### 3.4.4.1 os\_context\_jump

修改当前函数返回地址，使 RET 指令返回到指定地址，实现程序上下文跳转。

#### 3.4.4.2 os\_context\_restore

恢复任务堆栈指针，并通过 RETI 指令退出中断，CPU 自动恢复任务现场。

#### 3.4.4.3 os\_task\_save\_sp

将当前堆栈指针保存到指定任务的堆栈记录中，为任务切换准备数据。

---

## 3.5 任务调度机制

Myos 采用简单的轮转调度策略，通过 `os_scheduler_select_next()` 函数实现任务选择。

### 3.5.1 任务选择方式

调度器从当前任务的下一个任务开始循环查找，返回第一个处于就绪状态的任务 ID。如果到达任务列表末尾仍未找到，则回绕到任务 2 继续查找。

### 3.5.2 调度策略

- **轮转调度**：任务按照固定顺序轮流执行
- **就绪检测**：通过 `OS_PROCESS_OK[id] & 1` 判断任务是否就绪
- **循环查找**：确保所有就绪任务都能获得执行机会

### 3.5.3 任务切换条件

任务切换在以下条件下触发：

1. 定时器 T0 中断产生
2. 当前任务堆栈指针正常（未溢出）
3. 存在至少一个就绪任务

### 3.5.4 调度器实现特点

- **简单高效**：不涉及优先级、时间片等复杂概念
- **确定性**：任务执行顺序可预测
- **资源占用低**：仅需少量 RAM 和 CPU 资源

---

## 3.6 API 参考

本节列出 Myos 提供的公开接口，所有接口定义在 `inc/os.h` 中。

### 3.6.1 os\_set\_scheduler

**功能**：设置调度器类型

**函数原型**：

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

**参数**：

- `id`：调度器类型，1 表示 HRTOS，0 表示 MYOS

**返回值**：设置结果

**示例**：

```
os_set_scheduler(0); // 选择 MYOS 调度方式
```

---

### 3.6.2 os\_task\_save\_sp

**\*\*功能\*\***：保存任务堆栈指针

**\*\*函数原型\*\***：

```
void os_task_save_sp(u8 id);
```

**\*\*参数\*\***：

- `id`：任务 ID

**\*\*返回值\*\***：无

**\*\*示例\*\***：

```
os_task_save_sp(current_task_id);  
---
```

### 3.6.3 os\_context\_jump

**\*\*功能\*\***：跳转到指定程序地址执行

**\*\*函数原型\*\***：

```
void os_context_jump(u16 addr);
```

**\*\*参数\*\***：

- `addr`：跳转目标地址

**\*\*返回值\*\***：无

**\*\*示例\*\***：

```
os_context_jump(task_address);  
---
```

### 3.6.4 os\_scheduler\_select\_next

**\*\*功能\*\***：选择下一个可运行任务

**\*\*函数原型\*\***：

```
u8 os_scheduler_select_next(u8 id);
```

**\*\*参数\*\***：

- `id`：当前任务 ID

**\*\*返回值\*\***：下一个可运行任务 ID

**\*\*示例\*\***：

```
next_task = os_scheduler_select_next(current_task_id);  
---
```

### 3.6.5 os\_context\_restore

**\*\*功能\*\***：恢复指定任务上下文

**\*\*函数原型\*\***：

```
void os_context_restore(u8 id);
```

**\*\*参数\*\***：

- `id`：任务 ID

**\*\*返回值\*\***: 无

**\*\*示例\*\***:

```
os_context_restore(next_task_id);  
---
```

### 3.6.6 os\_get\_sp

**\*\*功能\*\***: 获取当前堆栈指针值

**\*\*函数原型\*\***:

```
u8 os_get_sp(void);
```

**\*\*参数\*\***: 无

**\*\*返回值\*\***: 当前堆栈指针

**\*\*示例\*\***:

```
u8 sp = os_get_sp();  
---
```

### 3.6.7 os\_get\_current\_task\_id

**\*\*功能\*\***: 获取当前任务 ID

**\*\*函数原型\*\***:

```
char os_get_current_task_id(void);
```

**\*\*参数\*\***: 无

**\*\*返回值\*\***: 当前正在运行的任务 ID

**\*\*示例\*\***:

```
u8 task_id = os_get_current_task_id();  
---
```

### 3.6.8 os\_tick\_stub

**\*\*功能\*\***: 系统 Tick 占位函数

**\*\*函数原型\*\***:

```
void os_tick_stub(void);
```

**\*\*参数\*\***: 无

**\*\*返回值\*\***: 无

**\*\*示例\*\***:

```
while (1) {  
    // 任务逻辑  
    os_tick_stub(); // 让出 CPU, 允许任务切换  
}  
---
```

### 3.6.9 os\_object\_register

**\*\*功能\*\***: 注册任务或中断服务程序

**\*\*函数原型\*\***:

```
void os_object_register(u16 addr, u8 id);
```

**\*\*参数\*\*:**

- `addr`: 对象入口地址

- `id`: 对象编号

- 0~13: 注册普通任务

- 14~24: 注册中断服务程序

- 15: 保留给 Timer0 内核调度器

**\*\*返回值\*\*:** 无

**\*\*示例\*\*:**

```
os_object_register(task1, 0);    // 注册任务 1
os_object_register(isr_ext0, 14); // 注册外部中断 0
---
```

### 3.6.10 os\_kernel\_tick\_handler

**\*\*功能\*\*:** MYOS 系统时钟及中断入口

**\*\*函数原型\*\*:**

```
void os_kernel_tick_handler(void);
```

**\*\*参数\*\*:** 无

**\*\*返回值\*\*:** 无

**\*\*示例\*\*:**

```
// 由定时器 T0 中断自动调用, 无需手动调用
---
```

## 3.7 Demo 使用说明

本节基于 `App/my\_main.c` 介绍 Myos 的使用方法。

### 3.7.1 Demo 功能

Demo 创建了 6 个 LED 显示任务和 2 个外部中断服务程序, 通过 LED 显示不同任务和中断的执行状态。

### 3.7.2 初始化流程

```
void my_main(void)
{
    /* 1. 初始化中断 */
    EA = 0;           // 关闭总中断
    EX0 = 1;          // 允许外部中断 0
    EX1 = 1;          // 允许外部中断 1
    IT0 = 1;          // 外部中断 0 下降沿触发
    IT1 = 1;          // 外部中断 1 下降沿触发

    /* 2. 解锁系统 */
    os_unlock();

    /* 3. 注册任务 */
}
```

```

os_object_register(task1, 0);
os_object_register(task2, 1);
os_object_register(task3, 2);
os_object_register(task4, 3);
os_object_register(task5, 4);
os_object_register(task0, 5);

/* 4. 注册中断服务程序 */
os_object_register(isr_ext0, 14);
os_object_register(isr_ext1, 16);

/* 5. 选择 MYOS 调度方式 */
os_set_scheduler(0);

/* 6. 开启总中断 */
EA = 1;

/* 7. 主循环 */
while (1) {
    // 主任务空闲，由调度器控制任务切换
}
}

```

### 3.7.3 任务创建方式

每个任务都是一个无限循环函数，通过 `os\_tick\_stub()` 让出 CPU：

```

void task1(void)
{
    while (1)
    {
        HRTOS_LED = 0;          // 任务逻辑
        os_tick_stub();          // 让出 CPU
    }
}

```

### 3.7.4 运行效果

- 6 个任务轮流执行，LED 显示不同的值（0、1、3、7、15、31）
- 外部中断触发时，LED 显示对应的值（0 或 255）
- 任务切换由定时器 T0 中断自动触发

---

## 3.8 Myos 与 HRTOS 关系

Myos 和 HRTOS 完整内核是不同层次的产品，各有其适用场景。

### 3.8.1 Myos 特点

- **轻量级**：代码量小，资源占用低
- **专注核心**：仅实现任务调度和上下文切换
- **学习导向**：适理解 RTOS 基本原理
- **验证平台**：用于验证调度算法和硬件适配

### 3.8.2 HRTOS 完整内核特点

- **\*\*功能完整\*\***：支持完整的任务管理、IPC、事件等待
- **\*\*扩展丰富\*\***：集成 Shell 命令行、Modbus 协议栈等
- **\*\*生产就绪\*\***：适合实际项目应用
- **\*\*生态完善\*\***：提供丰富的中间件和驱动支持

### 3.8.3 关系总结

Myos 是 HRTOS 完整内核的核心子集，专注于调度机制的学习和验证。掌握 Myos 后，可以更容易理解 HRTOS 完整内核的设计思想和实现细节。

---

## 3.9 使用场景

Myos 适合以下场景：

### 3.9.1 学习 RTOS 原理

- 理解任务调度基本概念
- 学习上下文切换机制
- 掌握中断处理流程

### 3.9.2 理解任务切换

- 观察多任务轮转执行
- 分析堆栈变化过程
- 验证调度算法正确性

### 3.9.3 裁剪型嵌入式项目

- 资源受限的 51 单片机项目
- 仅需基本调度功能的应用
- 快速原型验证

---

## 3.10 常见问题

### 3.10.1 任务无法切换

**\*\*可能原因\*\***：

1. 定时器 T0 未正确初始化
2. 中断未开启 (EA = 0)
3. 调度器类型未设置为 MYOS

**\*\*解决方法\*\*:**

- 检查定时器 T0 配置
- 确认 `EA = 1` 已执行
- 调用 `os\_set\_scheduler(0)` 设置为 MYOS

---

### 3.10.2 上下文异常

**\*\*可能原因\*\*:**

1. 堆栈指针保存错误
2. 任务堆栈溢出
3. 中断嵌套过深

**\*\*解决方法\*\*:**

- 检查 `os\_task\_save\_sp()` 调用时机
- 增大任务堆栈大小
- 减少中断嵌套深度

---

### 3.10.3 Tick 异常

**\*\*可能原因\*\*:**

1. 定时器 T0 配置错误
2. 中断向量表未正确设置
3. `os\_tick\_stub()` 未调用

**\*\*解决方法\*\*:**

- 检查定时器 T0 初始化代码
- 确认中断向量表正确注册
- 在任务循环中调用 `os\_tick\_stub()`

---

### 3.10.4 栈问题

**\*\*可能原因\*\*:**

1. 堆栈溢出 (SP > 127)
2. 任务堆栈分配不足
3. 函数嵌套过深

**\*\*解决方法\*\*:**

- 检查 `os\_get\_sp()` 返回值
- 增大任务堆栈大小

- 减少函数嵌套深度

---

## 3.11 附录

### 3.11.1 相关资源

- HRTOS 官方文档
- HRTOS 源码仓库
- 51 单片机开发指南

Shell

## 4 HRTOS Shell 模块使用说明

### 4.1 模块介绍

HRTOS Shell 模块为嵌入式实时操作系统提供轻量级命令行交互接口，通过 UART 串口实现人机交互。

#### 4.1.1 为什么 RTOS 需要 Shell

在嵌入式系统开发中，Shell 提供了便捷的系统调试和监控手段：

- **\*\*系统调试\*\***：无需重新烧录程序即可查看系统状态
- **\*\*状态查看\*\***：实时监控任务、信号量、消息队列等内核对象
- **\*\*参数控制\*\***：动态调整系统参数和配置
- **\*\*功能测试\*\***：快速验证系统功能模块

#### 4.1.2 HRTOS Shell 应用场景

Shell 模块特别适用于以下场景：

- 开发阶段的系统调试
- 现场故障诊断
- 系统参数动态配置
- 用户交互界面
- 自动化测试脚本执行

---

### 4.2 Shell 模块架构

Shell 模块采用分层架构设计，各层职责清晰：



## 4.2.1 文件职责说明

### 4.2.1.1 核心文件

- **\*\*shell.c\*\***: Shell 核心逻辑，包含初始化、输入处理、命令解析
- **\*\*shell\_cmd.c\*\***: 命令表定义和命令执行调度
- **\*\*shell\_port.c\*\***: 硬件移植接口，实现字符输入输出

### 4.2.1.2 Shell 子目录

- **\*\*init.c\*\***: Shell 初始化
- **\*\*input.c\*\***: UART 中断输入处理
- **\*\*main.c\*\***: Shell 任务主循环
- **\*\*parse\_line.c\*\***: 命令行解析，生成 argc/argv 参数
- **\*\*printf.c\*\***: 格式化输出函数
- **\*\*prompt.c\*\***: 命令提示符显示
- **\*\*shell.c\*\***: Shell 周期处理
- **\*\*shell\_port.c\*\***: 移植接口实现

### 4.2.1.3 Cmd 子目录

- **\*\*execute.c\*\***: 命令执行逻辑和字符串比较
- **\*\*help.c\*\***: help 命令实现
- **\*\*task.c\*\***: task 命令实现（任务管理）
- **\*\*msg.c\*\***: msg 命令实现（消息队列管理）
- **\*\*wait.c\*\***: wait 命令实现（等待状态查看）
- **\*\*version.c\*\***: version 命令实现（版本信息）
- **\*\*shell\_cmd.c\*\***: 命令表定义

---

## 4.3 HRTOS Shell 运行流程

结合 main.c，Shell 的启动和运行流程如下：

### 4.3.1 Shell 初始化

```
void hrtos_main(void)
{
    shell_init();           // 初始化 Shell
    bsp_uart_init();        // 初始化 UART
    os_event_init(SHELL_EVENT_RX); // 初始化 Shell 事件

    // 创建 Shell 任务
    os_task_create(shell_task, 1, 1, 6);

    // 创建 UART 中断任务
```

```
    os_task_create(uart_isr, 0, 9, 0);  
}
```

### 4.3.2 UART 初始化

UART 配置在 `bsp\_uart\_init()` 中完成：

```
void bsp_uart_init(void)  
{  
    SCON = 0x50;           // 8 位数据，可变波特率  
    TMOD &= 0x0F;          // 设置定时器模式  
    TMOD |= 0x20;          // 定时器 1 模式 2  
    TL1 = 0xFD;            // 设置定时初始值  
    TH1 = 0xFD;            // 设置定时初始值  
    ET1 = 0;               // 禁止定时器中断  
    TR1 = 1;               // 定时器 1 开始计时  
    EA = 1;                // 开总中断  
    ES = 1;                // 开串口中断  
}
```

### 4.3.3 输入接收

UART 中断接收字符：

```
void uart_isr(void) // interrupt 4  
{  
    char ch;  
  
    if (RI)  
    {  
        ch = SBUF;  
        RI = 0;  
  
        // 回车处理  
        if ((ch == '\r') || (ch == '\n'))  
        {  
            shell_line[shell_pos] = '\0';  
            if (shell_pos > 0)  
            {  
                os_event_set_from_isr(SHELL_EVENT_RX);  
            }  
        }  
        // 退格处理  
        else if ((ch == '\b') || (ch == 0x7F))  
        {  
            if (shell_pos > 0)  
            {  
                shell_pos--;  
            }  
        }  
        // 普通字符  
        else if (shell_pos < (SHELL_MAX_LINE_LEN - 1))  
        {  
            shell_line[shell_pos++] = ch;  
        }  
    }  
}
```

### 4.3.4 命令解析

Shell 任务周期处理：

```
static void shell_task(void)
{
    shell_init();

    while(1)
    {
        shell_process();
        os_delay(10);
    }
}
```

### 4.3.5 命令执行

```
void shell_process(void)
{
    os_event_wait(SHELL_EVENT_RX, 0);

    shell_parse_line(shell_line);

    shell_pos = 0;
    shell_line[0] = '\0';

    os_event_delete(SHELL_EVENT_RX);

    shell_prompt();
}

---
```

## 4.4 快速开始

### 4.4.1 最小使用示例

#### 4.4.1.1 初始化 Shell

```
#include "shell.h"

void hrtos_main(void)
{
    // 初始化 Shell
    shell_init();

    // 初始化 UART
    bsp_uart_init();

    // 初始化 Shell 事件
    os_event_init(SHELL_EVENT_RX);
}
```

#### 4.4.1.2 创建 Shell 任务

```
static void shell_task(void)
{
    shell_init();

    while(1)
    {
        shell_process();
        os_delay(10);
    }
}
```

```

    }
}

void hrtos_main(void)
{
    shell_init();
    bsp_uart_init();
    os_event_init(SHELL_EVENT_RX);

    // 创建 Shell 任务
    os_task_create(
        shell_task,
        1,          // 任务 ID
        1,          // 优先级
        6           // 堆栈大小
    );
}

```

#### 4.4.1.3 UART 配置

UART 配置在 BSP 层完成，根据目标 MCU 修改 `bsp\_uart\_init()`：

```

void bsp_uart_init(void)
{
    SCON = 0x50;          // 8 位数据，可变波特率
    TMOD &= 0x0F;
    TMOD |= 0x20;         // 定时器 1 模式 2
    TL1 = 0xFD;           // 波特率 9600
    TH1 = 0xFD;
    ET1 = 0;
    TR1 = 1;
    EA = 1;
    ES = 1;
}

```

#### 4.4.1.4 启动运行

编译烧录后，通过串口终端（如 SecureCRT、Putty）连接：

- 波特率：9600
- 数据位：8
- 停止位：1
- 校验位：无

连接成功后，将看到提示符：

```

HRTOS>
---
```

## 4.5 Shell 配置说明

### 4.5.1 宏配置

| 名称                 | 类型 | 说明            |
|--------------------|----|---------------|
| SHELL_MAX_LINE_LEN | 宏  | 命令行最大长度，默认 32 |
| SHELL_MAX_ARGC     | 宏  | 最大命令参数个数，默认 8 |

|                                      |
|--------------------------------------|
| SHELL_EVENT_RX   宏   Shell 事件标志，默认 1 |
| SHELL_CMD_COUNT   宏   命令表命令数量，默认 5   |

## 4.5.2 全局变量

|                                    |
|------------------------------------|
| 名称   类型   说明                       |
| ----- ----- -----                  |
| shell_line   char[]   Shell 输入缓冲区  |
| shell_pos   unsigned char   当前输入位置 |

## 4.5.3 配置建议

根据实际需求调整配置：

- 增加命令行长度：修改 `SHELL\_MAX\_LINE\_LEN`
- 增加参数个数：修改 `SHELL\_MAX\_ARGC`
- 修改事件标志：修改 `SHELL\_EVENT\_RX`

---

## 4.6 Shell API 参考

### 4.6.1 shell\_init

**\*\*功能说明\*\***：初始化 Shell 模块

**\*\*函数原型\*\***：

```
void shell_init(void);
```

**\*\*参数说明\*\***：无

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
void hrtos_main(void)
{
    shell_init();
    bsp_uart_init();
    os_event_init(SHELL_EVENT_RX);
}
```

---

### 4.6.2 shell\_process

**\*\*功能说明\*\***：Shell 周期处理函数，在 Shell 任务中循环调用

**\*\*函数原型\*\***：

```
void shell_process(void);
```

**\*\*参数说明\*\***：无

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
static void shell_task(void)
{
    shell_init();

    while(1)
    {
        shell_process();
        os_delay(10);
    }
}

---
```

### 4.6.3 shell\_putc

**\*\*功能说明\*\***：输出一个字符到串口

**\*\*函数原型\*\***：

```
void shell_putc(char ch);
```

**\*\*参数说明\*\***：

- `ch`：要输出的字符

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
shell_putc('A');

---
```

### 4.6.4 shell\_puts

**\*\*功能说明\*\***：输出字符串到串口

**\*\*函数原型\*\***：

```
void shell_puts(const char *str);
```

**\*\*参数说明\*\***：

- `str`：要输出的字符串

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
shell_puts("Hello HRTOS\r\n");

---
```

### 4.6.5 shell\_printf

**\*\*功能说明\*\***：格式化输出，支持 %d、%u、%X、%s、%c 格式

**\*\*函数原型\*\***：

```
void shell_printf(const char *fmt, ...);
```

**\*\*参数说明\*\***：

- `fmt`：格式化字符串

- `...`：可变参数

**\*\*返回值\*\***：无

**\*\*使用示例\*\***：

```
shell_printf("Task ID: %d, Priority: %d\r\n", task_id, priority);
shell_printf("Value: 0x%x\r\n", value);
---
```

#### 4.6.6 shell\_prompt

**\*\*功能说明\*\***: 显示 Shell 命令提示符

**\*\*函数原型\*\***:

```
void shell_prompt(void);
```

**\*\*参数说明\*\***: 无

**\*\*返回值\*\***: 无

**\*\*使用示例\*\***:

```
shell_prompt(); // 输出 "HRTOS> "
---
```

#### 4.6.7 shell\_parse\_line

**\*\*功能说明\*\***: 解析命令行，将输入字符串分割为 argc/argv 格式

**\*\*函数原型\*\***:

```
void shell_parse_line(char *line);
```

**\*\*参数说明\*\***:

- `line`: 命令行字符串

**\*\*返回值\*\***: 无

**\*\*使用示例\*\***:

```
char line[] = "task list";
shell_parse_line(line);
---
```

#### 4.6.8 shell\_cmd\_execute

**\*\*功能说明\*\***: 执行 Shell 命令

**\*\*函数原型\*\***:

```
void shell_cmd_execute(int argc, char *argv[]);
```

**\*\*参数说明\*\***:

- `argc`: 参数个数

- `argv`: 参数列表

**\*\*返回值\*\***: 无

**\*\*使用示例\*\***:

```
char *argv[] = {"task", "list"};
shell_cmd_execute(2, argv);
---
```

### 4.7 内置命令说明

### 4.7.1 help 命令

**\*\*功能\*\***：显示所有可用命令列表

**\*\*使用方法\*\***：

```
help
```

**\*\*参数\*\***：无

**\*\*示例\*\***：

```
HRTOS> help
Available commands:
  help
  version
  task
  wait
  msg
```

---

### 4.7.2 version 命令

**\*\*功能\*\***：显示系统版本信息

**\*\*使用方法\*\***：

```
version
```

**\*\*参数\*\***：无

**\*\*示例\*\***：

```
HRTOS> version
HRTOS Version 1.0.0
```

---

### 4.7.3 task 命令

**\*\*功能\*\***：任务管理，包括查看任务列表、挂起/恢复任务、修改优先级

**\*\*使用方法\*\***：

|                         |         |
|-------------------------|---------|
| task list               | 查看任务状态  |
| task suspend <id>       | 挂起任务    |
| task resume <id>        | 恢复任务    |
| task priority <id> <pr> | 修改任务优先级 |

**\*\*参数\*\***：

- `id`：任务 ID (0-15)

- `pr`：优先级 (0-8)

**\*\*示例\*\***：

```
HRTOS> task list
ID STATE PRIO
0  1     1
1  1     2
2  1     3

HRTOS> task suspend 1
suspend ok

HRTOS> task resume 1
```

```
resume ok

HRTOS> task priority 1 5
priority ok

---
```

#### 4.7.4 wait 命令

**\*\*功能\*\***：查看阻塞任务列表

**\*\*使用方法\*\***：

```
wait list
```

**\*\*参数\*\***：无

**\*\*示例\*\***：

```
HRTOS> wait list
ID  TYPE      OBJ  TICK  FLAG
1   SEM      0    100   0
2  MSG_RECV  1     50   1
```

**\*\*输出说明\*\***：

- `ID`：任务 ID
- `TYPE`：等待类型（DELAY、SEM、MUTEX、MSG\_SEND、MSG\_RECV、EVENT、MAILBOX）
- `OBJ`：等待对象 ID
- `TICK`：等待超时时间
- `FLAG`：等待标志

---

#### 4.7.5 msg 命令

**\*\*功能\*\***：查看消息队列信息

**\*\*使用方法\*\***：

```
msg list
```

**\*\*参数\*\***：无

**\*\*示例\*\***：

```
HRTOS> msg list
ID  SIZE  COUNT  HEAD  TAIL
0   32    5     0     3
1   64   10     2     8
```

**\*\*输出说明\*\***：

- `ID`：消息队列 ID
- `SIZE`：队列大小
- `COUNT`：当前消息数量
- `HEAD`：队列头位置
- `TAIL`：队列尾位置

---

## 4.8 自定义命令扩展

### 4.8.1 命令注册方式

在 `Src/Cmd/shell\_cmd.c` 中的命令表添加新命令：

```
const shell_cmd_t shell_cmd_table[] =
{
    { "help",      shell_cmd_help,      "show command list"      },
    { "version",   shell_cmd_version,   "show system version"    },
    { "task",      shell_cmd_task,      "task management"        },
    { "wait",      shell_cmd_wait,      "show waiting tasks"     },
    { "msg",       shell_cmd_msg,       "message management"     },
    { "mycmd",     shell_cmd_mycmd,     "my custom command"     } // 新增命令
};
```

### 4.8.2 命令名称定义

命令名称需满足：

- 小写字母
- 不含特殊字符
- 简洁明了

### 4.8.3 参数处理方式

命令函数原型：

```
void shell_cmd_mycmd(int argc, char *argv[]);
```

参数说明：

- `argc`：参数个数（包括命令本身）
- `argv`：参数数组，argv[0] 为命令名

### 4.8.4 执行函数编写

完整示例：

```
void shell_cmd_mycmd(int argc, char *argv[])
{
    // 检查参数个数
    if (argc < 2)
    {
        shell_puts("Usage:\r\n");
        shell_puts(" mycmd <param>\r\n");
        return;
    }

    // 处理参数
    if (shell_strcmp(argv[1], "on") == 0)
    {
        shell_puts("Turn ON\r\n");
        // 执行开启操作
    }
    else if (shell_strcmp(argv[1], "off") == 0)
    {

```

```

        shell_puts("Turn OFF\r\n");
        // 执行关闭操作
    }
    else
    {
        shell_puts("Unknown parameter\r\n");
    }
}

```

## 4.8.5 完整示例

添加一个 LED 控制命令：

```

// 在 shell_cmd.c 中添加命令表项
{ "led",      shell_cmd_led,      "LED control"      }

// 在单独文件中实现命令函数
void shell_cmd_led(int argc, char *argv[])
{
    u8 led_id;

    if (argc < 3)
    {
        shell_puts("Usage:\r\n");
        shell_puts("  led <id> <on|off>\r\n");
        return;
    }

    led_id = shell_atoi(argv[1]);

    if (shell_strcmp(argv[2], "on") == 0)
    {
        // 点亮 LED
        shell_printf("LED %d ON\r\n", led_id);
    }
    else if (shell_strcmp(argv[2], "off") == 0)
    {
        // 熄灭 LED
        shell_printf("LED %d OFF\r\n", led_id);
    }
    else
    {
        shell_puts("Invalid command\r\n");
    }
}

```

使用示例：

```

HRTOS> led 1 on
LED 1 ON

HRTOS> led 1 off
LED 1 OFF

```

---

## 4.9 Shell 移植说明

### 4.9.1 移植文件

移植到不同 MCU 需要修改以下文件：

- **\*\*shell\_port.c\*\***: 字符输入输出接口
- **\*\*bsp\_uart\_init()\*\***: UART 初始化
- **\*\*uart\_isr()\*\***: UART 中断处理

## 4.9.2 字符输入接口

在 UART 中断中调用 Shell 输入处理:

```
void uart_isr(void)
{
    char ch;

    if (RI)
    {
        ch = SBUF;
        RI = 0;

        // 回车处理
        if ((ch == '\r') || (ch == '\n'))
        {
            shell_line[shell_pos] = '\0';
            if (shell_pos > 0)
            {
                os_event_set_from_isr(SHELL_EVENT_RX);
            }
        }
        // 退格处理
        else if ((ch == '\b') || (ch == 0x7F))
        {
            if (shell_pos > 0)
            {
                shell_pos--;
            }
        }
        // 普通字符
        else if (shell_pos < (SHELL_MAX_LINE_LEN - 1))
        {
            shell_line[shell_pos++] = ch;
        }
    }
}
```

## 4.9.3 字符输出接口

实现 `shell\_putc()` 和 `shell\_puts()`:

```
void shell_putc(char ch)
{
    bsp_uart_send_byte((unsigned char)ch);
}

void shell_puts(const char *str)
{
    bsp_uart_send_str_ram(str);
}
```

## 4.9.4 UART 适配

根据目标 MCU 修改 `bsp\_uart\_init()`：

```
void bsp_uart_init(void)
{
    // 配置 UART 参数
    // 设置波特率
    // 使能中断
}
```

### 4.9.5 移植步骤

1. 实现底层 UART 驱动
2. 实现 `bsp\_uart\_init()`
3. 实现 `bsp\_uart\_send\_byte()`
4. 实现 `bsp\_uart\_send\_str\_ram()`
5. 修改 UART 中断处理函数
6. 调整波特率和中断优先级

---

## 4.10 HRTOS 调试应用示例

### 4.10.1 查看任务状态

使用 `task list` 命令查看所有任务：

```
HRTOS> task list
ID STATE PRIO
0  1     1
1  1     2
2  1     3
```

**\*\*应用场景\*\*：**

- 确认任务是否正确创建
- 检查任务优先级配置
- 监控任务运行状态

### 4.10.2 查看消息队列

使用 `msg list` 命令查看消息队列状态：

```
HRTOS> msg list
ID  SIZE  COUNT  HEAD  TAIL
0   32    5     0     3
```

**\*\*应用场景\*\*：**

- 监控消息队列使用情况
- 检测消息积压
- 调试消息传递问题

### 4.10.3 查看等待状态

使用 `wait list` 命令查看阻塞任务：

```
HRTOS> wait list
ID  TYPE      OBJ  TICK  FLAG
1   SEM      0    100   0
2   MSG_RECV 1    50    1
```

**\*\*应用场景\*\*：**

- 诊断死锁问题
- 查看任务等待原因
- 监控系统资源竞争

#### 4.10.4 任务控制

动态控制任务运行：

```
HRTOS> task suspend 2
suspend ok

HRTOS> task resume 2
resume ok

HRTOS> task priority 2 5
priority ok
```

**\*\*应用场景\*\*：**

- 临时挂起调试任务
- 动态调整任务优先级
- 测试任务调度策略

#### 4.10.5 查看版本信息

```
HRTOS> version
HRTOS Version 1.0.0
```

**\*\*应用场景\*\*：**

- 确认固件版本
- 现场版本确认
- 问题追溯

---

### 4.11 Demo 说明

#### 4.11.1 硬件环境

Demo 基于 51 单片机：

- MCU：8051 系列
- 外设：UART 串口
- 波特率：9600

#### 4.11.2 软件结构

Demo 包含以下任务：

1. **\*\*Shell 任务\*\***：处理命令行输入
2. **\*\*UART 中断任务\*\***：接收串口数据
3. **\*\*Demo 任务 1\*\***：LED 闪烁测试
4. **\*\*Demo 任务 2\*\***：LED 闪烁测试

### 4.11.3 任务结构

```
void hrtos_main(void)
{
    shell_init();
    bsp_uart_init();
    os_event_init(SHELL_EVENT_RX);

    // Shell 任务
    os_task_create(shell_task, 1, 1, 6);

    // UART 中断任务
    os_task_create(uart_isr, 0, 9, 0);

    // Demo 任务
    os_task_create(demo_task1, 2, 1, 2);
    os_task_create(demo_task2, 3, 2, 2);
}
```

### 4.11.4 运行效果

系统启动后，通过串口终端连接，显示提示符：

```
HRTOS>
```

可以输入命令进行交互：

```
HRTOS> help
```

Available commands:

```
help
version
task
wait
msg
```

```
HRTOS> task list
```

```
ID STATE PRIO
```

```
0  1      1
1  1      2
2  1      1
3  1      2
```

```
HRTOS> version
```

```
HRTOS Version 1.0.0
```

### 4.11.5 操作示例

1. 连接串口终端（9600, 8N1）
2. 看到 `HRTOS>` 提示符
3. 输入 `help` 查看命令列表

4. 输入 `task list` 查看任务状态

5. 输入 `version` 查看版本信息

---

## 4.12 常见问题

### 4.12.1 无法输入字符

**\*\*可能原因\*\*:**

- UART 未正确初始化
- 波特率不匹配
- 中断未使能

**\*\*解决方法\*\*:**

- 检查 `bsp\_uart\_init()` 配置
- 确认终端波特率设置为 9600
- 检查 EA、ES 中断使能位

### 4.12.2 命令无响应

**\*\*可能原因\*\*:**

- Shell 任务未创建
- 事件未初始化
- 命令解析错误

**\*\*解决方法\*\*:**

- 确认 `os\_task\_create()` 创建了 Shell 任务
- 检查 `os\_event\_init(SHELL\_EVENT\_RX)` 调用
- 查看命令拼写是否正确

### 4.12.3 UART 乱码

**\*\*可能原因\*\*:**

- 波特率不匹配
- 定时器配置错误
- 晶振频率不匹配

**\*\*解决方法\*\*:**

- 确认终端和 MCU 波特率一致
- 检查定时器重装值 (TL1、TH1)
- 根据实际晶振频率调整定时器值

### 4.12.4 命令无法识别

**\*\*可能原因\*\*:**

- 命令拼写错误
- 命令未注册
- 命令表未更新

**\*\*解决方法\*\*:**

- 使用 `help` 查看可用命令
- 检查命令表是否包含该命令
- 确认命令名称拼写正确（小写）

#### 4.12.5 参数错误

**\*\*可能原因\*\*:**

- 参数个数不足
- 参数类型错误
- 参数超出范围

**\*\*解决方法\*\*:**

- 使用命令的 help 功能查看用法
- 检查参数个数是否正确
- 确认参数值在有效范围内

#### 4.12.6 Shell 提示符不显示

**\*\*可能原因\*\*:**

- `shell\_init()` 未调用
- `shell\_prompt()` 未执行
- 输出函数未实现

**\*\*解决方法\*\*:**

- 确认在 `hrtos\_main()` 中调用了 `shell\_init()`
- 检查 `shell\_prompt()` 函数实现
- 验证 `shell\_puts()` 输出功能

---

### 4.13 附录

#### 4.13.1 A. 命令速查表

| 命令    | 功能     | 示例     |
|-------|--------|--------|
| ----- | -----  | -----  |
| help  | 显示命令列表 | `help` |

|               |        |                     |
|---------------|--------|---------------------|
| version       | 显示版本信息 | `version`           |
| task list     | 查看任务状态 | `task list`         |
| task suspend  | 挂起任务   | `task suspend 1`    |
| task resume   | 恢复任务   | `task resume 1`     |
| task priority | 修改优先级  | `task priority 1 5` |
| wait list     | 查看阻塞任务 | `wait list`         |
| msg list      | 查看消息队列 | `msg list`          |

### 4.13.2 B. 配置参数速查表

|                    |       |            |
|--------------------|-------|------------|
| 配置项                | 默认值   | 说明         |
| -----              | ----- | -----      |
| SHELL_MAX_LINE_LEN | 32    | 命令行最大长度    |
| SHELL_MAX_ARGC     | 8     | 最大参数个数     |
| SHELL_EVENT_RX     | 1     | Shell 事件标志 |
| SHELL_CMD_COUNT    | 5     | 命令数量       |

### 4.13.3 C. 错误码说明

Shell 模块使用字符串返回错误信息：

- `Unknown command`：命令不存在
- `Invalid task id`：任务 ID 无效
- `Invalid priority`：优先级无效
- `suspend failed`：挂起失败
- `resume failed`：恢复失败
- `priority failed`：优先级设置失败

---

**\*\*更新日期\*\***：2026-07-23

**\*\*适用版本\*\***：HRTOS 3.32+