Coop

HRTOS COOP 协作式实时操作系统手册

1.1 模块介绍

COOP（Cooperative Real-Time Operating System）是面向 8051 微控制器的轻量级协作式调度内核，使用 Keil C51 开发环境。

COOP 采用协作式轮转调度。任务通过主动调用 `os_schedule()` 让出 CPU，调度器按照任务 ID 的循环顺序查找下一个就绪任务，并通过修改 8051 栈上的返回地址转向目标任务入口。

内核提供任务创建、任务删除、延时等待、定时唤醒及嵌套临界区保护等基础能力。

1.1.1 设计特点

- **轻量化内核：** 采用固定任务表和就绪位图管理任务。
- **协作式调度：** 任务主动让出 CPU，不通过 Timer0 中断强制切换任务。
- **轮转选择：** 从当前任务的下一个 ID 开始搜索就绪任务。
- **延时管理：** 使用 Timer0 中断递减任务等待计数，到期后重新就绪。
- **静态资源管理：** 任务槽位固定，不提供动态任务槽位分配接口。
- **适用资源受限平台：** 面向 8051 架构及兼容的 C51 开发环境。

COOP 适用于学习协作式调度、简单嵌入式控制及资源受限系统的基础任务管理。

1.2 运行环境与配置

项目	当前配置
目标架构	8051
编译器	Keil C51
调度方式	协作式轮转调度
最大任务槽位	16
IDLE 任务 ID	0
用户任务 ID	1～15
定时器	Timer0
定时器模式	16 位定时器模式

项目	当前配置
任务就绪管理	16 位就绪位图
任务控制块	入口地址、等待计数

1.2.1 内核配置

头文件 `os.h` 中的默认配置如下：

```
#define OS_MAX_TASK    16
#define OS_IDLE_TASK   0
```

`OS_MAX_TASK` 表示任务槽位总数，包括 IDLE 槽位。因此，用户可使用的任务 ID 为 1～15，最多可注册 15 个用户任务。

任务 ID 用于标识任务，不代表任务优先级。当前 COOP 不实现优先级调度。

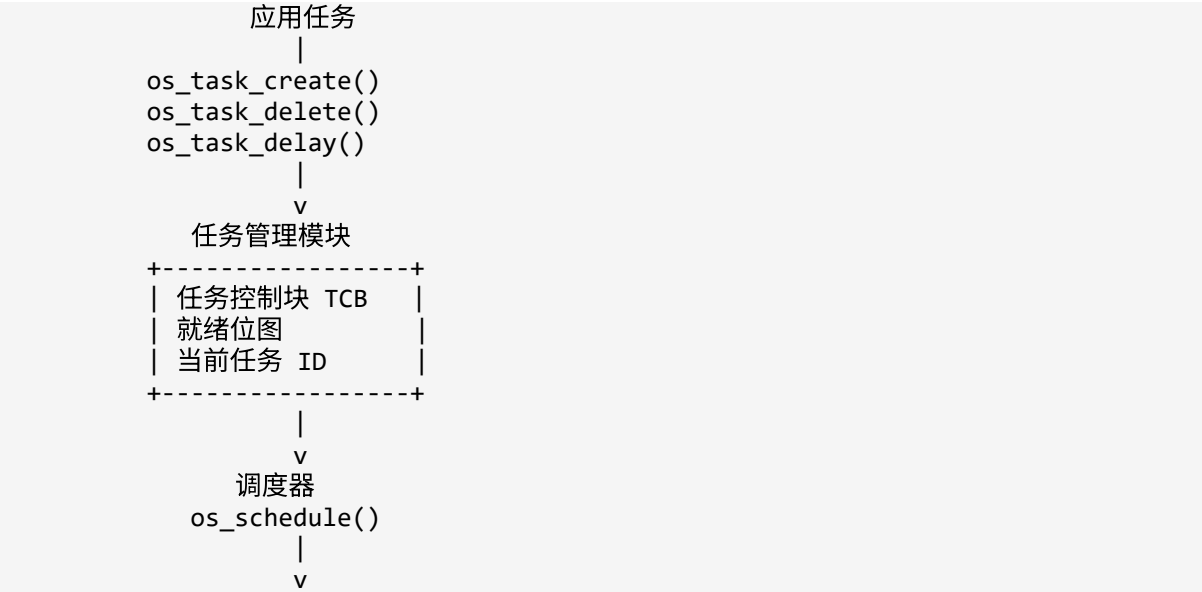
1.2.2 文件组成

文件	职责
`os.h`	基础类型、内核配置、TCB 和 API 声明
`os.c`	任务管理、就绪位图、调度器、临界区和 Timer0 延时唤醒
`main.c`	LED 轮转演示、任务注册和系统启动

具体工程还需要包含目标芯片对应的头文件及正确的 Keil C51 工程配置。

1.3 内核架构

COOP 的内核由任务管理、调度器和系统节拍三个主要部分构成。



```
          修改 8051 栈返回地址
                    |
                    v
              下一个任务入口

       Timer0 中断 ---> 延时计数递减
                    ---> 到期任务重新就绪
                    ---> 不执行任务调度
```

1.3.1 任务控制块

当前 TCB 定义如下：

```
typedef struct
{
    unsigned int addr;
    unsigned int wait_tick;
} OS_TCB;
```

各字段含义：

- `addr`：任务入口地址。
- `wait_tick`：任务剩余等待节拍数，归零后任务重新就绪。

任务表采用 XDATA 存储：

```
OS_TCB xdata os_task_table[OS_MAX_TASK];
```

当前 TCB 不包含任务优先级、独立任务栈指针或完整上下文保存区。

1.3.2 就绪位图

内核通过 `os_ready_bitmap` 管理任务就绪状态。

```
unsigned int os_ready_bitmap;
```

每一位对应一个任务 ID：

- 位为 1：任务就绪。
- 位为 0：任务未就绪。

例如：

```
os_ready_bitmap |= ((unsigned int)1 << id);
```

将对应任务设置为就绪状态。

```
os_ready_bitmap &= ~((unsigned int)1 << id);
```

将对应任务移出就绪集合。

这种方式能够使用一个 16 位变量表示全部 16 个任务槽位的就绪状态。

1.4 调度机制

1.4.1 协作式调度原理

COOP 不会因为定时器中断或其他任务就绪而自动抢占当前任务。任务必须主动调用 `os_schedule()`，调度器才会执行下一次任务选择。

典型流程：

```
任务 A 执行
    |
    v
调用 os_schedule()
    |
    v
从任务 A 的下一个 ID 开始搜索
    |
    v
找到下一个就绪任务 B
    |
    v
更新当前任务 ID
    |
    v
修改返回地址
    |
    v
执行任务 B 的入口
```

如果当前任务仍然就绪，但没有其他就绪任务，调度器会选择当前任务。由于切换机制会重新转向任务入口，这不等同于从原来的 C 语句位置恢复执行。

1.4.2 轮转搜索算法

`os_find_next_task()` 从 `os_current + 1` 开始，按 ID 递增顺序循环搜索，越过最大 ID 后回到 ID 0。

搜索顺序示意：

```
当前任务 ID = 3

4 -> 5 -> 6 -> ... -> 15 -> 0 -> 1 -> 2
```

找到第一个就绪任务后立即返回其 ID。

如果其他任务都未就绪：

- 当前任务仍就绪，则继续选择当前任务。
- 当前任务不就绪，则选择 ID 0 的 IDLE 任务。

这里的轮转是**按任务 ID 进行的就绪任务搜索**，不是基于优先级的调度。

1.4.3 任务切换实现

`os_schedule()` 的关键实现为：

```
next = os_find_next_task();

os_current = next;

SP = 0x32;

sp = SP - 1;
((unsigned char data *)sp)[0] =
    (unsigned char)(os_task_table[next].addr & 0xFF);

sp = SP;
((unsigned char data *)sp)[0] =
    (unsigned char)(os_task_table[next].addr >> 8);
```

其工作过程包括：

- 选择下一个就绪任务。
- 更新当前任务 ID。
- 将 SP 设置为内核预期的固定位置。
- 修改栈中保存的返回地址低字节和高字节。
- 通过函数返回流程转向目标任务入口。

该实现依赖 8051 的栈布局、调用过程和返回地址存放方式，以及特定的 SP 配置。

重要限制： 当前实现不是传统的独立任务栈与完整 CPU 上下文保存/恢复机制。任务切换后会重新进入目标任务的入口，而不是恢复该任务此前的任意执行位置。因此，不能假设普通局部变量会跨调度保留其值，也不能直接将标准抢占式 RTOS 的任务上下文模型套用到 COOP。

实际工程必须确保编译器生成的调用栈布局与内核实现相匹配。

1.5 任务生命周期

1.5.1 创建任务

函数原型：

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

参数：

参数	说明
`addr`	任务入口地址
`id`	任务 ID

返回值：

- `0`：创建操作成功。
- `-1`：任务 ID 超出范围，或尝试使用保留的 IDLE 槽位。

创建过程会记录任务入口、清零等待计数，并将任务设置为就绪状态。

示例：

```
os_task_create(task_led1, 1);
os_task_create(task_led2, 2);
```

注意：

- ID 0 保留给系统 IDLE 任务。
- 用户任务 ID 必须小于 `OS_MAX_TASK`。
- 当前接口不包含优先级和栈深度参数。
- 当前实现没有检查任务槽位是否已被占用，重复使用同一 ID 会覆盖对应的入口信息。
- 入口地址使用 `unsigned int` 保存，需确认目标编译器和工程配置与该地址表示方式兼容。

1.5.2 删除任务

函数原型：

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

删除操作会将任务从就绪集合中移除，并将其等待计数清零。

示例：

```
os_task_delete(5);
```

注意：

- ID 0 不能删除。
- 超出任务范围的 ID 会被忽略。
- 删除操作不会释放独立任务栈，因为当前 TCB 并未实现独立任务栈管理。
- 删除当前正在运行的任务后，应通过后续调度转向其他就绪任务；不能把删除操作理解为立即抢占或立即完成任务切换。

1.5.3 当前任务 ID

内核使用：

```
unsigned char os_current;
```

记录当前调度选择的任务 ID。

该变量是内核状态，不属于当前公开头文件声明的 API。应用程序不应随意修改它。

1.6 延时与 Timer0

1.6.1 任务延时

函数原型：

```
void os_task_delay(unsigned int tick);
```

参数 `tick` 表示系统节拍数，而不是直接表示毫秒。

使用示例：

```
os_task_delay(200);
os_schedule();
```

当任务调用 `os_task_delay()` 时：

- 将当前任务的等待计数设置为指定值。
- 将当前任务从就绪位图中移除。
- 任务调用 `os_schedule()`，主动让出 CPU。
- Timer0 中断持续递减等待计数。
- 等待计数归零后，任务重新进入就绪集合。
- 任务在后续调度中重新获得执行机会。

如果 `tick` 为 0，函数直接返回，不会将当前任务挂起。

延时函数本身不会切换任务。 如果任务延时后没有调用 `os_schedule()`，调度器不会因为这次延时操作而自动运行。

1.6.2 Timer0 中断

当前中断入口：

```
void wake_task(void) interrupt 1
```

中断服务程序负责：

- 清除 `TF0` 标志。
- 遍历任务表。
- 对非零的 `wait_tick` 递减。
- 当等待计数归零时，将任务重新设置为就绪状态。

中断处理过程不会调用 `os_schedule()`。

这种设计将定时唤醒与任务切换分离，保持协作式调度模型：**定时器负责唤醒任务，任务主动触发调度。**

1.6.3 节拍周期

当前初始化将 Timer0 配置为 16 位定时器模式：

```
TMOD &= 0xF0;
TMOD |= 0x01;

TH0 = 0x00;
TL0 = 0x00;

TF0 = 0;
ET0 = 1;
```

```
TR0 = 1;
```

该代码没有在每次中断后重新装载定时器初值。因此，不能仅凭 `os_task_delay(200)` 就认定延时为 200 ms，也不能直接将其解释为固定的 1 ms 或 10 ms 节拍。

实际中断周期取决于晶振频率、定时器时钟分频、目标芯片的计时模式及溢出行为。需要根据目标硬件配置进行计算和实测。

1.7 系统初始化

函数原型：

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

初始化过程包括：

- 清空就绪位图。
- 将当前任务 ID 设置为 0。
- 重置临界区嵌套计数器。
- 清空任务表中的入口地址和等待计数。
- 设置 IDLE 任务入口。
- 配置 Timer0 为 16 位模式。
- 清除 Timer0 溢出标志并开启其中断。
- 启动 Timer0。
- 开启总中断。

基本使用流程：

```
void main(void)
{
    SP = 0x30;

    os_init();

    os_task_create(task_led1, 1);
    os_task_create(task_led2, 2);

    os_schedule();

    while (1);
}
```

上述代码用于说明初始化和启动顺序。由于调度器依赖特定的栈布局，实际工程必须遵守对应的 SP 配置要求。

1.8 临界区

1.8.1 进入临界区

函数原型：

```
void os_enter_critical(void);
```

当前实现：

```
EA = 0;
os_en_cr_count++;
```

该函数关闭总中断，并递增临界区嵌套计数器。

1.8.2 退出临界区

函数原型：

```
void os_exit_critical(void);
```

当前实现会递减非零的嵌套计数器。当计数器归零时，执行：

```
EA = 1;
```

这使得内层临界区退出时不会提前开启中断。

注意： 当前实现没有保存进入临界区之前的 EA 状态。当最外层退出时，它会直接开启总中断。因此，不能将它视为能够自动恢复任意先前中断状态的通用临界区实现。

使用时应遵守配对调用原则：

```
os_enter_critical();

/* 需要保护的短操作 */

os_exit_critical();
```

避免在临界区内执行耗时操作。

1.9 LED 轮转示例

1.9.1 示例目标

示例通过多个任务轮流写入 P1 端口，演示协作式任务切换。

LED 输出序列：

```
01 -> 02 -> 04 -> 08 -> 10 -> 20 -> 40 -> 80
80 -> 40 -> 20 -> 10 -> 08 -> 04 -> 02 -> 01
```

每个任务执行端口写入、演示延时及主动调度操作。

1.9.2 任务示例

```
#define LED_PORT P1

static void task_led0(void)
{
    LED_PORT = 0x01;
    demo_delay();
    os_schedule();
}
```

```
static void task_led1(void)
{
    LED_PORT = 0x02;
    demo_delay();
    os_schedule();
}
```

其他任务采用类似结构，写入不同的端口值。

这里的 `demo_delay()` 是普通忙等待函数，用于拉开 LED 输出间隔，不是内核提供的延时机制。

1.9.3 带延时的任务

```
static void task_led13(void)
{
    LED_PORT = 0x02;

    demo_delay();

    os_task_delay(200);

    os_schedule();
}
```

任务在设置延时后移出就绪集合，再通过 `os_schedule()` 让出 CPU。等待计数归零后，该任务重新就绪。

1.9.4 任务注册

示例使用 ID 1～15 注册 15 个用户任务：

```
os_task_create(task_led1,  1);
os_task_create(task_led2,  2);
os_task_create(task_led3,  3);
/* 其余任务依次注册 */
os_task_create(task_led14, 14);
os_task_create(task_led0,  15);
```

ID 0 由内核保留给 IDLE，不能用于注册用户任务。

由于任务切换通过入口地址重新开始执行，示例任务不应依赖跨调度保留的普通局部变量。需要保存的状态应采用合适的持久存储方式，例如静态变量或全局变量。

1.10 API 参考

API	功能	返回值
`os_init()`	初始化内核和 Timer0	无
`os_task_create(addr, id)`	注册任务并将其设置为就绪	成功为 0，失败为 -1
`os_task_delete(id)`	删除任务的就绪状态	无
`os_task_delay(tick)`	设置当前任务的等待计数	无

API	功能	返回值
`os_schedule()`	执行任务选择与切换	无
`os_enter_critical()`	进入嵌套临界区	无
`os_exit_critical()`	退出嵌套临界区	无

当前版本**没有实现或公开**以下旧手册中的接口：

- `os_wait()`
- `os_alloc_slot()`
- 带优先级和栈深度参数的四参数 `os_task_create()`

请以当前 `os.h` 中实际声明的接口为准。

1.11 使用限制与注意事项

1.11.1 不支持抢占式调度

COOP 不会因为某个任务优先级更高而抢占当前任务，也不会在 Timer0 中断中执行任务切换。

如果任务长时间不调用 `os_schedule()`，其他就绪任务可能长时间得不到执行机会。

1.11.2 不支持独立任务栈与完整上下文恢复

当前内核没有为每个任务维护独立栈指针，也没有保存和恢复完整的 CPU 上下文。

调度器直接修改固定栈位置上的返回地址。因此，任务入口、调用深度、编译器生成代码和 SP 布局都必须满足实现要求。

不要将该机制直接用于依赖任意挂起位置恢复、深层调用状态保持或独立任务栈的应用。

1.11.3 延时单位由硬件节拍决定

`os_task_delay()` 接收的是节拍数。必须先确认 Timer0 的实际中断周期，才能换算延时时间。

1.11.4 任务 ID 由应用静态分配

创建任务时应为每个用户任务分配唯一 ID。当前实现没有检查重复 ID，也没有独立的任务创建状态标志。

1.11.5 临界区需要正确配对

每次 `os_enter_critical()` 都应有对应的 `os_exit_critical()`。同时，应注意当前实现会在最外层退出时直接开启总中断。

1.11.6 目标芯片与工程配置必须匹配

当前内核依赖 8051 的栈和返回地址机制，并通过 `reg52.h` 使用传统 8051 寄存器定义。移植到其他兼容芯片时，应验证寄存器定义、Timer0 行为、堆栈布局及编译器兼容性。

1.12 COOP 与 HRTOS 的关系

COOP 与完整 HRTOS 的主要区别在于调度模型、上下文管理及内核服务范围。

对比项目	COOP	HRTOS 完整内核
调度模型	协作式轮转	基于优先级的抢占式调度
任务切换	主动调用调度函数	依据内核调度机制执行
优先级调度	不支持	支持
任务延时	Timer0 计数，到期重新就绪	由完整内核时间与等待机制管理
IPC	当前实现未提供	提供多种同步通信机制
任务上下文	修改固定栈位置的返回地址	依完整内核的上下文切换实现
资源定位	精简的协作式任务管理	更完整的实时操作系统服务

COOP 适合用于理解任务注册、就绪状态、轮转搜索和主动让出 CPU 等基本概念。若应用需要优先级抢占、复杂阻塞机制、IPC 或更完整的实时任务管理，应根据目标硬件资源选择合适的完整内核。

1.13 常见问题

1.13.1 为什么只有一个任务持续运行?

检查该任务是否在适当位置调用 `os_schedule()`，以及其他任务是否处于就绪状态。

协作式调度不会强制中断一个持续运行且不让出 CPU 的任务。

1.13.2 调用 os_task_delay() 后，为什么任务没有立即切换?

`os_task_delay()` 只设置等待计数并清除当前任务的就绪状态。任务需要进一步调用 `os_schedule()`，调度器才会选择下一个任务。

1.13.3 为什么延时 200 个 tick 不一定是 200 ms?

因为 tick 是定时器中断次数，而不是固定的毫秒单位。当前代码没有配置明确的周期重装值，需要根据目标芯片的计时条件计算实际节拍。

1.13.4 为什么不建议在任务里依赖普通局部变量保存状态?

当前调度器通过修改栈上的返回地址重新进入任务入口，不会像标准独立上下文切换机制那样恢复任务此前的执行位置。需要跨调度保留的状态，应放在适当的持久存储位置。

1.13.5 为什么任务 ID 0 不能创建?

ID 0 是内核保留的 IDLE 槽位，由 `os_init()` 设置其入口地址。用户任务应使用 ID 1～15。

1.13.6 为什么不建议直接复制到其他 8051 芯片?

不同芯片、编译器配置和代码生成方式可能影响栈布局、定时器行为和寄存器定义。移植前应确认调度器的返回地址修改方式与目标工程兼容。

总结

COOP 是一个以固定任务表、就绪位图和主动调度为核心的 8051 协作式调度内核。它通过 Timer0 管理延时唤醒，并通过修改栈上的返回地址实现任务入口切换。

理解其协作式调度特征、固定栈布局和任务入口重入行为，是正确使用与维护当前实现的关键。