Myos

HRTOS MyOS 轻量级调度模块说明

1.1 模块介绍

MyOS 是 HRTOS 项目提供的轻量级任务调度实验模块，面向 8051 系列单片机，用于研究任务调度、堆栈指针保存、上下文切换以及中断向量转发等机制。

MyOS 复用 HRTOS 提供的基础设施，采用相对精简的调度流程，将任务注册、任务选择、堆栈管理与中断处理结合起来。

1.1.1 设计目标

- **任务调度：** 按任务编号顺序查找可运行任务。
- **上下文管理：** 保存当前任务的堆栈指针，并切换到目标任务的堆栈。
- **中断转发：** 将不同中断映射到相应的中断服务程序。
- **机制验证：** 通过 LED 任务和外部中断示例观察任务切换及中断响应行为。

MyOS 侧重于调度机制的实现与验证，不应直接等同于具备完整任务管理、IPC 和资源管理功能的 HRTOS 完整内核。

1.2 整体架构

1.2.1 架构组成

MyOS 的主要实现可以按职责划分为应用层、调度与中断管理层，以及上下文管理层。

```
应用层
 ├─ my_main()
 ├─ 用户任务
 └─ 外部中断服务程序
          │
          ▼
调度与中断管理层
 ├─ os_kernel_tick_handler()
 ├─ os_scheduler_select_next()
 └─ os_object_register()
          │
          ▼
上下文管理层
 ├─ os_task_save_sp()
 ├─ os_context_restore()
```

```
├─ os_context_jump()
├─ os_get_sp()
└─ os_get_current_task_id()
        │
        ▼
8051 硬件
├─ CPU 堆栈与 SP
├─ Timer0
└─ 外部中断
```

以上为逻辑职责划分，不代表源码目录一定采用相同的物理分层。

1.2.2 主要文件职责

文件或函数	主要职责
`my_main.c`	配置外部中断、注册任务与中断服务程序、选择调度方式
`context_jump.c`	修改当前函数返回地址，并通过 `RET` 跳转到指定程序地址
`context_restore.c`	更新当前任务 ID、恢复目标任务的 SP，并执行 `RETI`
`task_save_sp.c`	保存指定任务的堆栈指针
`get_sp.c`	获取软件变量 `os_sp` 记录的堆栈指针值
`get_current_task_id.c`	获取当前任务 ID
`scheduler_select_next.c`	按任务编号顺序查找就绪任务
`kernel_tick_handler.c`	处理 Timer0 调度流程及其他中断的转发
`object_register.c`	注册任务入口或中断服务程序入口
`tick_stub.c`	短暂开启中断，为其他中断提供响应机会

1.3 任务编号与对象注册

MyOS 使用 `os_object_register()` 统一注册普通任务和中断服务程序。

1.3.1 普通任务注册

普通任务的外部注册编号为 `0~13`。注册时，函数会将编号加 2，再调用 HRTOS 的 `os_task_create()`。

```
os_object_register(task1, 0);
os_object_register(task2, 1);
```

外部注册编号与内部任务 ID 的映射关系如下：

外部注册编号	内部任务 ID
0	2
1	3

外部注册编号	内部任务 ID
2	4
…	…
13	15

MyOS 的调度器从内部任务 ID 2 开始回绕查找。任务 ID 0 和 1 不属于上述普通任务注册映射范围。

每个普通任务注册时，代码传入默认的函数嵌套深度参数 2。该参数的实际含义及其对堆栈分配的影响，应以 HRTOS 对应版本的任务创建实现为准。

1.3.2 中断服务程序注册

中断服务程序通过 `os_object_register()` 注册。当前代码使用的编号范围为 `14～24`，其中编号 `15` 被保留，不用于注册普通中断服务程序。

```
os_object_register(isr_ext0, 14);
os_object_register(isr_ext1, 16);
```

注册函数会将外部编号转换为中断向量表索引，并将入口地址写入 `os_interrupt_vector_table[]`。

中断编号转换逻辑需要与 `os_interrupt_id()` 的实际编号规则保持一致。由于中断向量表长度为 10，注册时必须确保转换后的索引处于有效范围内。

注意：当前 `os_object_register()` 对大于 24 的编号采用空分支处理，没有返回错误码，也没有提供注册成功或失败的状态反馈。

1.4 调度机制

1.4.1 调度器工作原理

`os_scheduler_select_next()` 根据当前任务 ID，从下一个编号开始查找就绪任务。

就绪判断条件为：

```
(OS_PROCESS_OK[i] & 1) != 0
```

该条件表示仅检查 `OS_PROCESS_OK[i]` 的最低位。

函数分两轮查找：

- 第一轮：从 `id + 1` 开始，查找到 `OS_PROCESS_MAX - 1`。
- 第二轮：如果第一轮没有找到，则从任务 ID 2 开始，查找到当前 ID。

找到就绪任务后，函数将其 ID 写入 `OS_HRTOS` 并返回。

1.4.2 调度策略

MyOS 当前使用按编号顺序查找就绪任务的策略。

- 不根据任务优先级进行比较。
- 不根据任务执行时间动态调整顺序。
- 通过 `OS_PROCESS_OK[]` 判断任务是否就绪。
- 找到第一个符合条件的任务后立即返回。

因此，不宜将该实现描述为优先级抢占调度器，也不宜仅凭循环查找逻辑就保证所有异常状态下均能实现公平轮转。

1.4.3 未找到就绪任务时的行为

当前 `os_scheduler_select_next()` 在两轮查找均未命中时，不会显式设置备用任务，也不会返回专门的错误状态，而是返回 `OS_HRTOS` 当前保存的值。

因此，调度器正常运行依赖任务就绪状态和 `OS_HRTOS` 状态符合预期。若需要将其作为通用接口使用，应进一步验证无就绪任务、输入 ID 越界等边界情况。

1.5 上下文切换机制

1.5.1 保存任务堆栈指针

`os_task_save_sp()` 将全局变量 `os_sp` 保存到指定任务的堆栈记录中：

```
void os_task_save_sp(u8 id)
{
    OS_JINCHENG_SP[id] = os_sp;
}
```

`OS_JINCHENG_SP[]` 的容量由 `OS_PROCESS_MAX` 决定，当前为 16 个元素。

该函数本身没有进行 ID 越界检查，调用方必须确保传入的任务 ID 有效。

1.5.2 恢复任务上下文

`os_context_restore()` 的核心操作为：

```
OS_CURRENT_TASK = id;
SP = OS_JINCHENG_SP[id];
RETI;
```

函数先更新当前任务 ID，再将硬件堆栈指针 SP 设置为目标任务保存的值，最后执行 `RETI` 退出中断。

这里需要区分两个概念：

- **恢复 SP：** 恢复目标任务的堆栈指针。

- **恢复完整现场：** 还需要保证程序计数器、堆栈内容以及所需寄存器等状态符合目标任务的执行要求。

`RETI` 本身不会自动恢复任意一组通用寄存器。实际任务能否从被中断的位置继续执行，取决于中断入口、现场保存方式、任务初始堆栈布局和相关底层代码是否相互匹配。

1.5.3 修改返回地址进行跳转

`os_context_jump()` 通过修改当前函数的返回地址实现程序跳转：

- 保存 IE 寄存器。
- 使用两次 `POP` 弹出当前返回地址。
- 将目标地址的低字节和高字节压入堆栈。
- 恢复 IE 寄存器。
- 执行 `RET`，跳转到目标地址。

该操作依赖 8051 的堆栈行为以及编译器生成的函数调用代码。修改返回地址并不是通用的 C 语言控制流操作，必须与具体编译器、内存模型和调用约定匹配。

1.5.4 上下文切换的边界

MyOS 的上下文切换不能只根据 `os_task_save_sp()` 和 `os_context_restore()` 两个函数就认定为完整的独立任务现场切换。进行移植或修改时，应结合以下内容验证：

- 中断入口如何保存 CPU 现场。
- `os_sp` 在何处更新。
- 每个任务的初始 SP 和堆栈内容如何建立。
- 任务切换时哪些寄存器被保存和恢复。
- `RET` 与 `RETI` 对应的返回地址是否正确。
- 任务堆栈是否可能覆盖其他数据区域。

1.6 Timer0 调度与中断转发

1.6.1 内核中断处理流程

`os_kernel_tick_handler()` 通过 `os_interrupt_id()` 获取当前中断编号。

当中断编号为 1 时，代码将其作为 Timer0 内核调度中断处理：

- 清除 `TF0`。
- 将 `TH0` 和 `TL0` 清零。
- 获取当前任务 ID。
- 保存当前任务的堆栈指针。
- 选择下一个就绪任务。
- 恢复目标任务的 SP 并执行 `RETI`。

对应流程如下：

```
Timer0 中断
    ↓
os_kernel_tick_handler()
    ↓
识别中断编号 1
    ↓
清除 TF0 并重置 TH0、TL0
    ↓
读取当前任务 ID
    ↓
保存当前任务 SP
    ↓
选择目标任务
    ↓
恢复目标任务 SP
    ↓
RETI
```

1.6.2 Timer0 周期

当前代码将 `TH0` 和 `TL0` 清零，但没有在该处理函数中根据预设重装值重新装载定时器。

因此，不能仅凭此函数确定固定的 Tick 周期。实际周期需要结合以下因素：

- MCU 的定时器计数模式。
- 晶振频率和定时器时钟分频。
- 中断进入及处理开销。
- HRTOS 的其他定时器初始化代码。

1.6.3 其他中断的转发

对于非 Timer0 中断，代码根据中断编号调整索引，然后执行：

```
os_context_jump(os_interrupt_vector_table[i]);
```

该操作将控制流转移到已注册的中断服务程序入口。

由于中断编号转换包含特殊处理，修改中断编号、向量表长度或 Timer0 的保留规则时，必须同步检查注册路径和转发路径，确保二者使用同一套映射关系。

1.7 API 参考

以下接口依据当前提供的 `os.h` 声明和对应函数实现整理。它们属于 MyOS 模块接口，不代表所有函数都是推荐给应用层直接调用的接口。

1.7.1 os_object_register()

功能： 注册普通任务或中断服务程序入口。

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

参数：

- `addr`：对象入口地址。
- `id`：外部注册编号。

当前约定：

- `0～13`：普通任务。
- `14～24`：中断服务程序注册范围。
- `15`：保留编号。

返回值：无。

示例：

```
os_object_register(task1, 0);
os_object_register(isr_ext0, 14);
```

1.7.2 os_scheduler_select_next()

功能： 查找下一个就绪任务。

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

参数：

- `id`：当前任务 ID。

返回值：选中的任务 ID；如果没有找到就绪任务，当前实现返回 `OS_HRTOS` 中已有的值。

该函数没有对输入 ID 进行显式越界检查。

1.7.3 os_task_save_sp()

功能： 保存指定任务的堆栈指针。

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

参数：

- `id`：内部任务 ID。

返回值：无。

1.7.4 os_context_restore()

功能： 设置当前任务 ID、恢复指定任务的 SP，并通过 `RETI` 退出中断。

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

参数：

- `id`：待恢复的任务 ID。

返回值：无。该函数在执行 `RETI` 后改变控制流，不应按普通函数调用的方式理解其后续执行过程。

1.7.5 os_context_jump()

功能： 修改当前返回地址并跳转到指定程序地址。

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

参数：

- `addr`：跳转目标地址。

返回值：无。该函数依赖底层汇编、堆栈布局及编译器调用约定。

1.7.6 os_get_sp()

功能： 获取 `os_sp` 中记录的 SP 值。

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

返回值：当前记录的堆栈指针值。

注意：该接口读取的是软件变量 `os_sp`，并非直接读取硬件 SP 寄存器。二者是否始终同步，取决于其他代码的更新逻辑。

1.7.7 os_get_current_task_id()

功能： 获取当前任务 ID。

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

返回值：`OS_CURRENT_TASK` 当前保存的值。

1.7.8 os_kernel_tick_handler()

功能： 处理 Timer0 调度中断和其他中断的转发。

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

该函数由相应的内核中断处理路径调用，不应作为普通应用函数随意调用。

1.7.9 os_tick_stub()

功能： 短暂开启中断，为其他中断提供响应机会。

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

该函数执行的操作为：

- 设置 `EA = 1`。
- 执行一段空循环。
- 设置 `EA = 0`。

注意： `os_tick_stub()` 本身不直接调用调度器，也不保证每次调用都会触发任务切换。它会强制修改总中断使能状态，因此调用前必须考虑当前中断状态及临界区要求。

1.7.10 os_set_scheduler()

功能： 选择调度方式。

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

当前 Demo 使用：

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

按照头文件注释，`0` 表示 MYOS，`1` 表示 HRTOS。具体返回值和状态切换行为应以该函数的实际实现为准。

1.8 Demo 使用说明

1.8.1 Demo 功能

示例程序定义了 6 个 LED 显示任务和 2 个外部中断服务程序。

任务通过改变 P1 端口输出值展示执行状态：

任务	LED 输出值
`task1`	0
`task2`	1
`task3`	3
`task4`	7
`task5`	15
`task0`	31

外部中断示例分别将 LED 输出设置为 0 和 255，并执行一段忙等待循环。

1.8.2 初始化流程

```
void my_main(void)
{
    /* 初始化外部中断 */
    EA = 0;
    EX0 = 1;
    EX1 = 1;
    IT0 = 1;
    IT1 = 1;

    /* 解锁系统 */
    os_unlock();

    /* 注册普通任务 */
    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);

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

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

    /* 开启总中断 */
    EA = 1;

    while (1)
    {
    }
}
```

这段代码展示了当前 Demo 的配置步骤，不应脱离 HRTOS 的初始化与调度切换实现，直接当作独立 RTOS 的完整启动代码。

1.8.3 任务编写方式

示例任务使用无限循环执行应用逻辑：

```
void task1(void)
{
    while (1)
    {
        HRTOS_LED = 0;
        os_tick_stub();
    }
}
```

`os_tick_stub()` 提供短暂的中断响应窗口。任务能否按预期切换，仍取决于 Timer0 是否正确配置、内核调度路径是否正常运行，以及任务上下文是否初始化正确。

1.8.4 外部中断示例

```
void isr_ext0(void)
{
    u16 i;

    HRTOS_LED = 0;

    i = 20000;
    while (i--)
    {
    }
}
```

该示例通过改变 LED 输出并执行忙等待，便于观察外部中断处理行为。实际应用中应控制中断服务程序的执行时间，避免长时间占用 CPU 或影响其他中断的响应。

1.9 使用注意事项

1.9.1 任务 ID 与数组边界

`OS_PROCESS_MAX` 为 16，相关任务状态数组也以该值定义容量。调用 `os_task_save_sp()`、`os_context_restore()` 等函数时，应保证任务 ID 在有效范围内。

当前部分底层接口没有显式的参数越界检查。

1.9.2 堆栈安全

内核处理函数包含如下检查：

```
if (os_get_sp() > 127)
{
    while (1);
}
```

这会在记录的 `os_sp` 大于 127 时进入死循环。

该检查只能说明代码设置了一个基于 `os_sp` 的阈值判断；它并不能单独证明所有任务堆栈都受到完整保护，也不能保证检测到所有堆栈越界情形。

1.9.3 中断使能状态

`os_tick_stub()` 会直接设置 `EA`，先开启再关闭总中断，并不保存和恢复原始中断状态。调用时需要确保不会破坏上层临界区或其他中断管理逻辑。

1.9.4 编译器与芯片适配

MyOS 使用 8051 特定寄存器和内嵌汇编，涉及 `SP`、`IE`、`RET`、`RETI`、`POP` 和 `PUSH` 等操作。

移植到不同的 8051 兼容芯片或修改编译器设置时，应重新验证：

- 汇编语法和调用约定。
- 函数入口地址与指针类型。
- 中断向量编号。
- 定时器时钟配置。
- 任务堆栈布局与现场恢复。

1.9.5 调度行为验证

建议至少验证以下情形：

- 多个任务均就绪时的选择顺序。
- 仅有一个任务就绪时的行为。
- 没有任务就绪时的行为。

- 当前任务 ID 位于边界值时的回绕逻辑。
- Timer0 与外部中断同时发生时的响应。
- 多次任务切换后 SP 与任务现场是否保持正确。

1.10 MyOS 与 HRTOS 完整内核的关系

MyOS 与 HRTOS 完整内核共享项目基础设施，但二者的能力范围不能简单视为相同。

MyOS 当前展示的重点是任务选择、堆栈指针保存与恢复，以及中断转发机制。HRTOS 完整内核则提供更丰富的任务管理和系统功能，具体能力以对应版本的源码与 API 文档为准。

MyOS 适合用于理解底层调度实现、验证 8051 上下文切换机制，以及研究任务和中断之间的关系。对于实际工程选型，应根据功能需求、资源约束、可靠性要求和经过验证的硬件测试结果进行判断。

1.11 常见问题

1.11.1 任务没有按预期切换

建议检查：

- Timer0 是否正确配置并产生中断。
- `os_set_scheduler(0)` 是否执行成功。
- `OS_PROCESS_OK[]` 的就绪状态是否符合预期。
- 当前任务 ID 是否有效。
- 目标任务的堆栈指针和初始现场是否正确。

1.11.2 中断发生后程序异常

建议检查：

- 中断编号与向量表索引的转换。
- `os_interrupt_vector_table[]` 中的入口地址。
- `os_context_jump()` 修改返回地址的堆栈假设。
- `RET` 与 `RETI` 的使用场景。
- 中断嵌套和总中断使能状态。

1.11.3 堆栈检测触发

检查 `os_sp` 的更新方式、任务堆栈初始化、中断嵌套深度和函数调用深度。不要仅依据 `os_get_sp()` 的单个返回值，就认定所有堆栈使用情况都已得到检测。

1.11.4 Tick 周期不符合预期

检查 Timer0 的工作模式、晶振频率、定时器计数方式以及中断处理流程。当前处理函数清零 `TH0` 和 `TL0`，但不能仅凭这一操作推断实际 Tick 周期。

总结： MyOS 是 HRTOS 项目中的轻量级调度实验模块，核心实现集中在任务选择、堆栈指针管理、上下文跳转与中断转发。阅读和使用时，应以实际源码及目标硬件验证结果为准，尤其需要关注任务初始现场、堆栈布局、中断编号映射和定时器配置。