基于定时中断的时间片轮选调度

—— 成都理工大学开源协会

目录

一、中断：是谁打断了 CPU 的思路?

1.1 什么是中断?

在没有中断之前，单片机就像一个死磕型选手：

- 从 `main()` 开始，一行一行往下执行；
- 写了 `while(1)`，就在里面**一直转圈圈**；
- 除非遇到 `return` 或复位，不会“自己跳走”。

有了中断之后：

当某些“更加重要 / 更加紧急”的事情发生时，比如：

- 定时器溢出（时间到了）；
- 串口收到数据（有人跟我说话了）；
- 外部引脚电平变化（外部事件触发）；

硬件会对 CPU 说一句：**“喂，先别干你手里的活，这边有紧急任务！”**

CPU 会做三件事：

1. 暂存当前现场（PC、部分寄存器等）；
2. 跳转到对应的 **中断服务程序**（ISR）；
3. 中断处理完毕后，再回来继续执行原来的主程序。

> 类比：
>
> - 主程序好比你在写作业；
> - 中断好比你在写作业时，手机突然响了；
> - 你停下笔去接电话（ISR），接完电话再回来继续写作业。

开源出品，必属精品

1.2 本例中的中断：Timer0 每 1ms 打一次“节拍”

在下面的初始化函数中：

```
void Timer0_Init(void) // 1毫秒@12.000MHz
{
    AUXR &= 0x7F; // 定时器时钟12T模式
    TMOD &= 0xF0; // 设置定时器模式
    TL0 = 0x18;   // 设置定时初始值
    TH0 = 0xFC;   // 设置定时初始值
    TF0 = 0;      // 清除TF0标志
    TR0 = 1;      // 定时器0开始计时
    EA = 1;       // 打开总中断
    ET0 = 1;      // 使能定时器0中断
}
```

这里做了几件关键事情：

- 配置 Timer0 工作在 12T 模式；
- 通过装载 `TH0/TL0`，让它 **每 1ms 溢出一次**；
- 打开总中断 `EA` 和定时器中断 `ET0`；
- 使能计时 `TR0 = 1`。

从这一刻起：

> 系统多了一个**稳定的 1ms 心跳**。
> 每 1ms，CPU 都会被 Timer0 中断短暂“叫出去”一下。

这个“1ms 心跳”，就是我们后面要讲的**时间片**。

二、`volatile`：别让编译器“假装看不见”变量变化

2.1 问题从哪里来？

先看这几行变量定义：

```
volatile unsigned char LED_State = 0xFF;  // LED 状态
volatile unsigned char beep_state = OFF;  // 蜂鸣器状态
volatile unsigned char relay_state = OFF; // 继电器状态

volatile unsigned char tick_250ms = 0;
volatile unsigned char tick_500ms = 0;
volatile unsigned char tick_1s = 0;
```

共同点：

- 这些变量既会在 **中断函数** 中被修改，
- 又会在 **主循环** 中被读取和修改。

也就是说：
有两段完全不同的代码在“动用同一份数据”。

开源出品，必属精品

2.2 不加 `volatile` 会发生什么?

想象你是编译器，你看到这样的代码片段：

```
while (1)
{
    if (tick_1s)
    {
        // ...
    }
}
```

你很聪明，会想：

> “`tick_1s` 在 `while` 里面看起来没人改呀，
> 那我就只在一开始从内存读一次，
> 以后都用寄存器缓存的值，省时间！”

结果：

- 中断在后台把 `tick_1s` 从 0 改成了 1；
- 但主循环可能看的是“旧值”(寄存器里的 0)；
- **程序可能进不了 `if (tick_1s)` 分支**。

这就是典型的：

> “变量已经变了，但主循环假装看不见” 的问题。

2.3 `volatile` 的作用一句话

> `volatile` 告诉编译器：
> “这个变量**随时可能被别的地方改**，
> 你每次用它，都必须老老实实去内存里读 / 写，
> 不准偷偷在寄存器里藏一份。”

所以：

- 中断和主程序共享的变量 → 必须 `volatile`；
- 外设寄存器（如 `P0`, `SBUF`） → 通常也会标成 `volatile`。

三、标志位：中断“立旗”，主循环“看旗、清旗”

3.1 为什么不用中断直接干完所有事?

中断里当然可以啥都干，比如：

- 收到串口数据，立刻做复杂协议解析；
- 计时到了，立刻跑一长串业务逻辑。

问题是：

- 中断执行太久，会 **占用 CPU**，影响其他正常工作；
- 优先级更高的中断可能被延迟；
- 程序结构变乱，不好维护、不好扩展。

开源出品，必属精品

3.2 更优雅的做法：事件标志（Flag）

我们可以规定：

1. **中断只做两件事：**
   - 做少量简单的计时 / 判断；
   - 设置一个“事件标志”(Flag)。
2. **主循环做三件事：**
   - 不停轮询这些标志；
   - 看到哪个标志被置 1，就执行对应任务；
   - 做完之后清除标志。

示意结构：

```
// 中断服务函数
void xxx_isr(void)
{
    // 简单逻辑
    flag_X = 1;   // 告诉主循环：事件 X 发生了
}

// 主循环
while (1)
{
    if (flag_X)
    {
        flag_X = 0;  // 清标志
        // 真正执行任务 X
    }
}
```

3.3 本例中的标志位

本例中的这几个变量就是典型的“事件标志”：

```
volatile unsigned char tick_250ms = 0;
volatile unsigned char tick_500ms = 0;
volatile unsigned char tick_1s = 0;
```

- `tick_250ms = 1`：表示 **“250ms 周期任务到了”**；
- `tick_500ms = 1`：表示 500ms 周期任务到了；
- `tick_1s = 1`：表示 1s 周期任务到了。

谁来置位？ 中断。
谁来处理？ 主循环。
谁来清零？ 主循环。

> 这是一种非常常见、好用、可扩展的设计模式：
> **“中断立旗，主循环看旗干活”**。

开源出品，必属精品

四、时间片轮选调度：一个 CPU 如何“轮着伺候大家”？

4.1 问题场景

我们有三个周期任务：

1. LED 动画：每 **250ms** 更新一次；
2. 继电器：每 **500ms** 切换一次；
3. 蜂鸣器：每 **1000ms** 切换一次。

但 CPU 只有一个核心，一次只能做一件事。
我们希望它看起来像：

- LED 在规律地跑马灯；
- 继电器在规律地吸合和释放；
- 蜂鸣器在规律地响、停。

这就是一个小型的“多任务调度”问题。

4.2 步骤拆解：时间片 + 轮询

1. **构造统一时间片：**
   用 Timer0 产生 **1ms 中断** → 系统心跳。
2. **用计数器“分配节奏”：**

   在中断里：

   ```
   static unsigned int acc250 = 0, acc500 = 0, acc1000 = 0;

   acc250++;
   acc500++;
   acc1000++;
   ```

   - `acc250` 达到 250 → 250ms 到了；
   - `acc500` 达到 500 → 500ms 到了；
   - `acc1000` 达到 1000 → 1s 到了。
3. **到点就“举旗”：**

   ```
   if (acc250 >= 250)
   {
       acc250 = 0;
       tick_250ms = 1;
   }
   ```

   同理还有 `tick_500ms`、`tick_1s`。
4. **主循环轮流问一圈（轮选）：**

开源出品，必属精品

```
while (1)
{
    if (tick_1s)    { 处理 1s 任务;    }
    if (tick_500ms) { 处理 500ms 任务; }
    if (tick_250ms) { 处理 250ms 任务; }
}
```

- 永远按这个顺序反复循环；
- 哪个 `tick_xxx` 被置位，就执行对应那段代码（一小段）；
- 执行完记得清零标志位。

> 这就是一个最朴素的 **“基于时间片的轮询（轮转）调度”**：
>
> - 时间片由 1ms 定时中断提供；
> - 不同任务有不同的周期；
> - 主循环轮流扫描，谁到时间就服务谁。

五、示例代码 + 算法思想解析

下面是完整代码（含注释），后面按模块拆解说明。

```
#include <STC15F2K60S2.H>
#include "hardware.h"
#include "intrins.h"

volatile unsigned char LED_State = 0xFF;  // LED状态在ISR中使用，需volatile
volatile unsigned char beep_state = OFF;  // 蜂鸣器状态在ISR中使用，需volatile
volatile unsigned char relay_state = OFF; // 继电器状态在ISR中使用，需volatile

// 使用事件标志替代多字节计数器，避免主循环与ISR的非原子读写
volatile unsigned char tick_250ms = 0;
volatile unsigned char tick_500ms = 0;
volatile unsigned char tick_1s = 0;

void Timer0_Init(void) // 1毫秒@12.000MHz
{
    AUXR &= 0x7F; // 定时器时钟12T模式
    TMOD &= 0xF0; // 设置定时器模式
    TL0 = 0x18;   // 设置定时初始值
    TH0 = 0xFC;   // 设置定时初始值
    TF0 = 0;      // 清除TF0标志
    TR0 = 1;      // 定时器0开始计时
    EA = 1;       // 打开总中断
    ET0 = 1;      // 使能定时器0中断
}

unsigned char led_time = 0;
unsigned char led_flag = 0;

void main(void)
{
    Timer0_Init();
    while (1)
    {
```

```
        // 1s事件：切换蜂鸣器
        if (tick_1s)
        {
            tick_1s = 0;
            beep_state = !beep_state; // 简化切换
        }

        // 500ms事件：切换继电器
        if (tick_500ms)
        {
            tick_500ms = 0;
            relay_state = !relay_state; // 简化切换
        }

        // 250ms事件：更新LED动画
        if (tick_250ms)
        {
            tick_250ms = 0;
            if (++led_time >= 8)
            {
                led_time = 0;
                led_flag = !led_flag;
            }
            if (led_flag)
                LED_State |= ~(0xFF >> led_time); // 依次熄灭LED
            else
                LED_State &= (0xFF << led_time); // 依次点亮LED
        }
    }
}

void Timer0_Isr(void) interrupt 1 // 中断程序
{
    // 用静态毫秒累加器触发事件标志，避免主循环读写多字节计数器
    static unsigned int acc250 = 0, acc500 = 0, acc1000 = 0;

    acc250++;
    acc500++;
    acc1000++;

    if (acc250 >= 250)
    {
        acc250 = 0;
        tick_250ms = 1;
    }
    if (acc500 >= 500)
    {
        acc500 = 0;
        tick_500ms = 1;
    }
    if (acc1000 >= 1000)
    {
        acc1000 = 0;
        tick_1s = 1;
    }

    LED_Control(LED_State);
```

开源出品，必属精品

```
    BEEP_Relay_Control(beep_state, relay_state);
}
```

5.1 全局变量区：任务状态 + 事件标志

```
volatile unsigned char LED_State = 0xFF;
volatile unsigned char beep_state = OFF;
volatile unsigned char relay_state = OFF;
```

- 这三个代表“逻辑层面的意图”：
  - LED 当前应呈现什么样的 8 位模式；
  - 蜂鸣器当前是开还是关；
  - 继电器当前是吸合还是释放。
- **谁改它们？** 主循环里的任务逻辑；
- **谁用它们？** 中断中的 `LED_Control()` / `BEEP_Relay_Control()`，用于真正输出到硬件。

```
volatile unsigned char tick_250ms = 0;
volatile unsigned char tick_500ms = 0;
volatile unsigned char tick_1s = 0;
```

- 这三个是时间事件的“就绪标志位”：
  - “250ms 任务已到点”；
  - “500ms 任务已到点”；
  - “1s 任务已到点”。

中断置位，主循环清零并处理。

5.2 `Timer0_Init`：搭好“时间基准的舞台”

```
void Timer0_Init(void)
{
    AUXR &= 0x7F; // 定时器时钟12T模式
    TMOD &= 0xF0; // 设置定时器模式
    TL0 = 0x18;
    TH0 = 0xFC;
    TF0 = 0;
    TR0 = 1;
    EA = 1;
    ET0 = 1;
}
```

- 1ms 一次中断 = **1ms 时间片**；
- 所有任务的周期（250/500/1000ms）都是这一个时间片的整数倍；
- 后面所有调度行为都建立在这个“心跳”基础上。

开源出品，必属精品

5.3 中断服务函数：软件定时器 + 立事件标志

```
static unsigned int acc250 = 0, acc500 = 0, acc1000 = 0;

acc250++;
acc500++;
acc1000++;
```

- `static` 确保变量在每次中断之间不会丢失；
- 三个累加器分别对应三个周期任务。

```
if (acc250 >= 250)
{
    acc250 = 0;
    tick_250ms = 1;
}
```

- 250 次 1ms 中断 → 250ms；
- 重置计数器，置 `tick_250ms = 1`，表示："250ms 任务可以执行一次了"。

同理：

- `acc500` 达 500 → `tick_500ms = 1`；
- `acc1000` 达 1000 → `tick_1s = 1`。

> 注意：
> **中断这里只是"告诉主循环：到点啦！"**
> 没有真正去做 LED 动画运算，也没有直接对状态机进行复杂操作。

最后两句：

```
LED_Control(LED_State);
BEEP_Relay_Control(beep_state, relay_state);
```

- 以统一节奏（1ms）刷新输出，
- 确保硬件状态与逻辑变量保持同步。

5.4 主循环：简单但高效的"轮选调度器"

```
while (1)
{
    // 1s事件：切换蜂鸣器
    if (tick_1s)
    {
        tick_1s = 0;
        beep_state = !beep_state;
    }

    // 500ms事件：切换继电器
    if (tick_500ms)
    {
        tick_500ms = 0;
```

开源出品，必属精品

```
        relay_state = !relay_state;
    }

    // 250ms事件：更新LED动画
    if (tick_250ms)
    {
        tick_250ms = 0;
        if (++led_time >= 8)
        {
            led_time = 0;
            led_flag = !led_flag;
        }
        if (led_flag)
            LED_State |= ~(0xFF >> led_time); // 依次熄灭LED
        else
            LED_State &= (0xFF << led_time); // 依次点亮LED
    }
}
```

特点：

1. **结构非常清晰：**
   - 先看 1s 任务；
   - 再看 500ms 任务；
   - 再看 250ms 任务；
   - 然后回到循环开头，继续下一轮。
2. **每个任务都非常短：**
   - 蜂鸣器：翻转一个状态位；
   - 继电器：翻转一个状态位；
   - LED：状态机前进一步，更新一次 `LED_State`。
3. **多任务的“错觉”：**
   - 虽然 CPU 是一个一个执行的；
   - 但因为轮询速度很快，人眼感觉就是“同时在动”。

六、全文小结与课堂思考题

6.1 全文小结

1. **中断（Interrupt）**
   - Timer0 被配置为每 1ms 产生一次中断；
   - 中断服务函数为系统提供统一时间片，相当于整个系统的“节拍器”。
2. `volatile` **的作用**
   - 中断与主程序共享的变量必须加 `volatile`；
   - 防止编译器优化导致“变量明明变了，主循环却看不见”。
3. **标志位（事件驱动）**
   - 中断只负责“立旗”（设置标志位），不做复杂逻辑；
   - 主循环反复“看旗”，谁的旗举起来就处理谁的任务，再把旗放下。

开源出品，必属精品

4. **时间片轮选调度**
    - 用 1ms 定时中断作为时间片；
    - 用累加计数实现不同周期（250/500/1000ms）的“软件定时器”；
    - 主循环轮流检查各任务标志，谁到时间就执行谁；
    - 每次执行任务只做一小步，形成“伪并行”的多任务系统。
5. **工程意义**
    - 避免长延时 `delay` 导致的阻塞；
    - 结构清晰，易于扩展其他周期任务（再加一个 `tick_xxx` 和计数器即可）；
    - 是从“延时式编程”向“事件驱动 / RTOS 思维”过渡的重要练习。

开源出品，必属精品