我详细解释这个 **SysTick轮询延时** 的核心代码，这是三种方式中最能体现硬件原理的一种：

** **整体思路：把SysTick当"永远转动的秒表"** **

不像 `DelayIRQ_Ms()` 那样用中断维护全局变量，这里**直接读取硬件计数器SysTick->VAL的值**，像看秒表一样算时间差。

---

1. 初始化函数：启动自由运行的定时器

c 复制

```c
void DelayPoll_Init(void){
    SysTick->LOAD = 0xFFFFFF;  // 重装值设为24位最大值
    SysTick->CTRL = SysTick_CTRL_ENABLE_Msk | SysTick_CTRL_CLKSOURCE_Msk;
    SysTick->VAL = 0;
}
```

寄存器配置详解：

- `LOAD = 0xFFFFFF` ：24位计数器，最大值16,777,215（约0xFFFFFF）
- `CTRL` ：控制寄存器
  - `ENABLE_Msk` = 1：使能SysTick
  - `CLKSOURCE_Msk` = 1：选择72MHz系统时钟
  - **不写** `TICKINT_Msk` = **不使能中断**（关键！）
- `VAL = 0` ：当前值清零

启动后的行为：

复制

```
72MHz时钟 → SysTick VAL寄存器
├─ 每个时钟周期 VAL 减1
├─ VAL从1减到0时，COUNTFLAG标志位置1
└─ VAL从0再减1时，自动重装为0xFFFFFF，继续循环
```

所以SysTick->VAL是一个 永远递减 、 24位回绕 **的自由运行计数器！**

** 2. **微秒延时核心：从"递减计数器"读出"递增时间"** **

c 复制

```c
void DelayPoll_us(uint32_t us){
    uint32_t start = SysTick->VAL;       // ① 记录"秒表"起始值
    uint32_t ticks = us * 72;            // ② 计算需要等待多少个tick

    while(ticks > 0){                    // ③ 循环直到攒够ticks
        // 每轮循环计算过去了多少ticks
    }
}
```

** **难点：VAL是递减的，怎么算时间差？** **

假设 `us=10` （延时10微秒）， `ticks=720` ， `start=0xFFFFFF00`

** 情况A：没有发生24位回绕 **

复制

```
时间轴:    0μs         5μs         10μs
          ──┬──────────┬──────────▶
VAL值: 0xFF00 → 0xFE48 → 0xFD90
          │         │         │
          └─start   └─current

ticks -= (start - current) = 0xFF00 - 0xFE48 = 184 = 2.56μs
```

代码实现： `if(current <= start) ticks-=(start-current);`

情况B：发生了24位回绕

复制

```
时间轴:    0μs         5μs         10μs
          ──┬──────────┬──────────▶
VAL值: 0x000010 → 0xFFFFFF → 0xFFFE70
          │           │           │
          └─start     └─回绕点    └─current

// 不能直接减，要分段算
ticks -= (start + (0xFFFFFF - current))
        = 0x10 + (0xFFFFFF - 0xFFFE70) = 160 = 2.22μs
```

代码实现： `else ticks-=(start+(0xFFFFFF-current));`

COUNTFLAG标志位：检测是否回绕

c 复制

```c
if(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk){
    ticks -= 0xFFFFFF;  // 发生回绕，直接减去一个完整周期
}
```

- `COUNTFLAG` 在 ** VAL从1→0的瞬间 ** 被硬件置1
- 一旦置1，** 读取CTRL寄存器会自动清零 **
- 所以如果读到它为1，说明在上次检查到这次检查之间发生了回绕

** 3. **毫秒延时** **

c 复制

```c
void DelayPoll_ms(uint32_t ms){
    for(uint32_t i=0; i<ms; i++){
        DelayPoll_us(1000);
    }
}
```

- ** 简单粗暴 **：循环1000次1毫秒
- ** 缺点： **有** 累积误差 **（每次调用 `DelayPoll_us` 的开销）
- ** 改进版 **：直接算 `ticks = ms * 72000` ，一次循环搞定

** **优缺点对比** **

✅ **优点：**

1. **不占用中断**：适合已经用了所有中断优先级，或不想用中断的场景
2. **精度高**：直接读硬件，不受中断延迟影响
3. **独立性强**：不依赖全局变量，可重入

❌ **缺点：**

1. **阻塞CPU**：延时期间CPU一直在空转，啥都干不了
2. **代码复杂**：要处理24位回绕，容易写错
3. **功耗高**：CPU无法休眠

总结：三种方式对比

表格 复制

方式	原理	精度	CPU占用	复杂度	适用场景
普通延时	软件循环	低	阻塞	简单	简单测试
中断延时	中断+时间戳	**极高**	非阻塞	中	**通用推荐**
轮询延时	硬件计数器	高	阻塞	高	无中断可用时

一句话：轮询延时是"硬件秒表"，中断延时是"系统时钟"，前者直接但笨，后者优雅且强大。

调试技巧：在 `DelayPoll_Init()` 后，用调试器看 `SysTick->VAL` 的值，它会以肉眼可见的速度飞快变化（每13.9ns减1），证明定时器在工作！