

# 一种计算 CPU 使用率的方法及其实现原理

## 1 前言

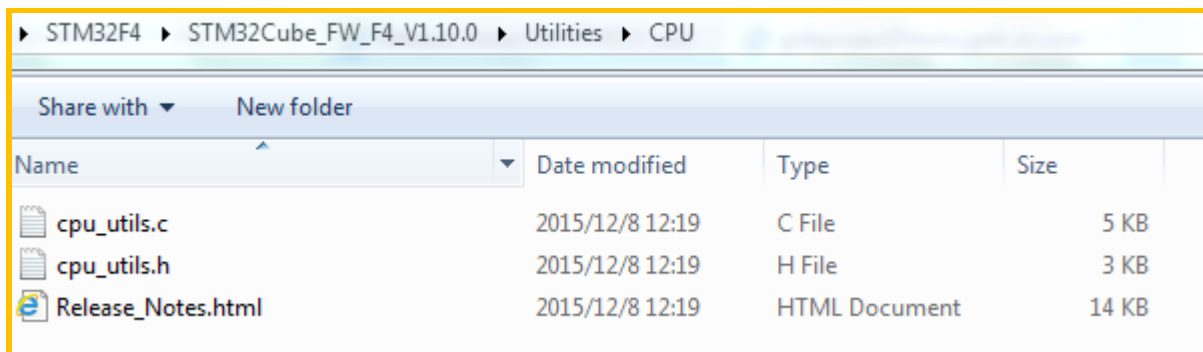
出于性能方面的考虑，有的时候，我们希望知道 CPU 的使用率为多少，进而判断此 CPU 的负载情况和对于当前运行环境是否足够“胜任”。本文将介绍一种计算 CPU 占有率的方法以及其实现原理。

## 2 移植算法

### 2.1 算法简介

此算法是基于操作系统的，理论上不限于任何操作系统，只要有任务调度就可以。本文将以 FreeRTOS 为例来介绍本算法的使用方法。

本文所介绍的算法出处为随 Cube 库一起提供的，它在 cube 库中的位置如下图所示：



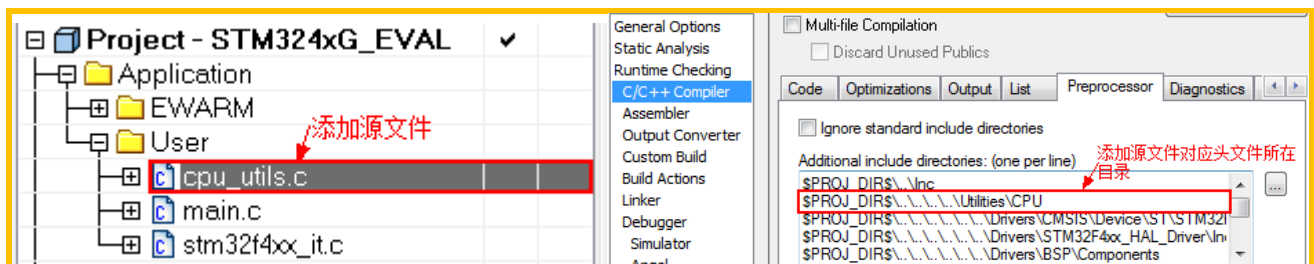
本文将以 STM32F4 为例，测试环境为 STM3240G-EVAL 评估板。

### 2.2 开始移植

本文以 CubeF4 内的示例代码工程

STM32Cube\_FW\_F4\_V1.10.0\Projects\STM324xG\_EVAL\Applications\FreeRTOS\FreeRTOS\_ThreadCreation 为例，IDE 使用 IAR。

**第一步：**使用 IAR 打开 FreeRTOS\_ThreadCreation 工程，将 cpu\_utils.c 文件添加到工程，并在工程中添加对应头文件目录：



**第二步：**打开 FreeRTOS 的配置头文件 FreeRTOSConfig.h 修改宏 configUSE\_IDLE\_HOOK 和 configUSE\_TICK\_HOOK 的值为 1：

```
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK            1           //修改此宏的值为 1
#define configUSE_TICK_HOOK            1           //修改此宏的值为 1
#define configCPU_CLOCK_HZ              ( SystemCoreClock )
#define configTICK_RATE_HZ              ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES            ( 8 )
#define configMINIMAL_STACK_SIZE        ( ( uint16_t ) 128 )
```

**第三步：**继续在 FreeRTOSConfig.h 头文件的末尾处添加 traceTASK\_SWITCHED\_IN 与 traceTASK\_SWITCHED\_OUT 定义：

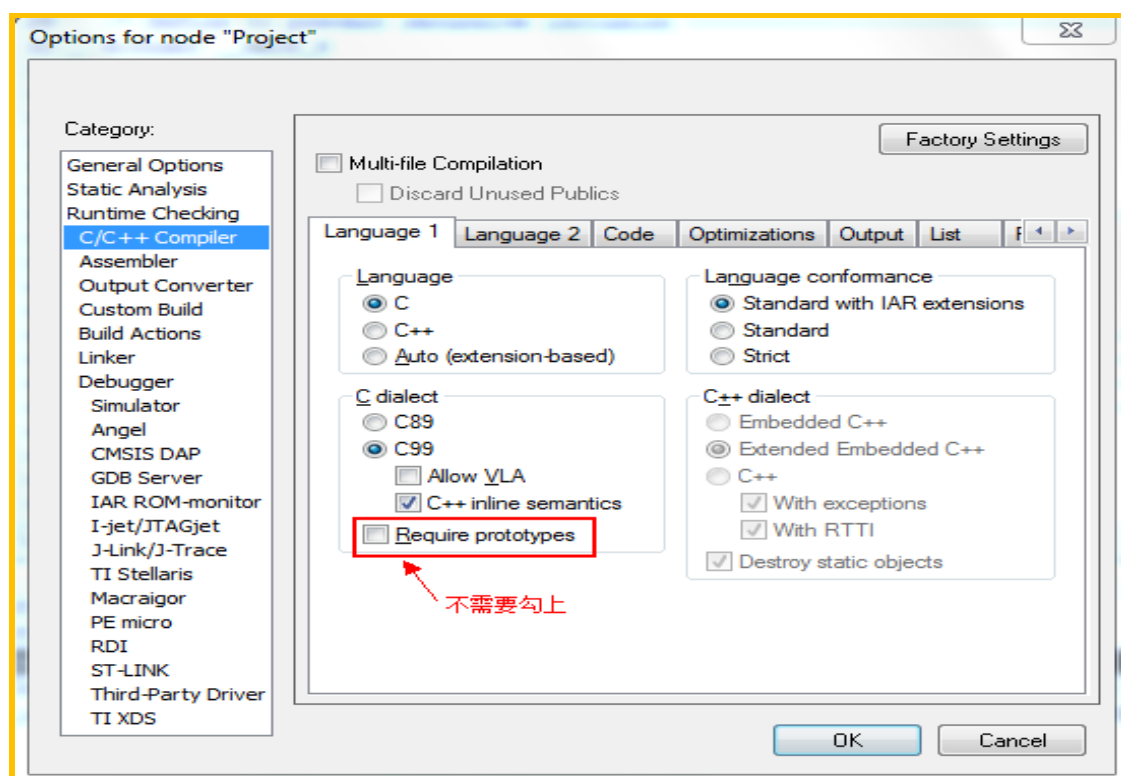
```
#define traceTASK_SWITCHED_IN()  extern void StartIdleMonitor(void); \
                                StartIdleMonitor()
#define traceTASK_SWITCHED_OUT() extern void EndIdleMonitor(void); \
                                EndIdleMonitor()
```

**第四步：**在 main.h 头文件中 include “cmsis\_os.h”

Main.h :

```
#include "stm32f4xx_hal.h"
#include "stm324xg_eval.h"
#include "cmsis_os.h"           //添加包含此头文件
//...
```

**第五步：**修改工程属性，使编译过程不需要函数原型：



**第六步：**在工程中任何用户代码处都可以调用 `osGetCPUUsage()`函数来获取当前 CPU 的使用率：

```
static void LED_Thread2(void const *argument)
{
    uint32_t count;
    uint16_t usage =0;
    (void) argument;

    for(;;)
    {
        count = osKernelSysTick() + 10000;

        /* Toggle LED2 every 500 ms for 10 s */
        while (count >= osKernelSysTick())
        {
            BSP_LED_Toggle(LED2);

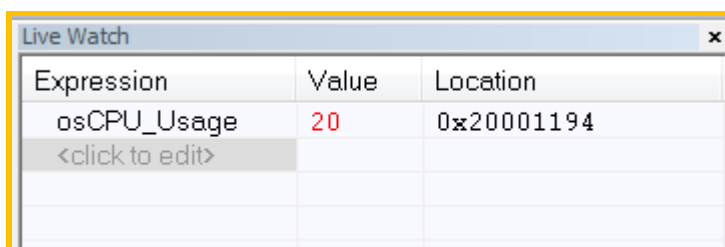
            osDelay(500);
        }
        usage =osGetCPUUsage();           //获取当前 CPU 的使用率
        /* Turn off LED2 */
        BSP_LED_Off(LED2);

        /* Resume Thread 1 */
        osThreadResume(LEDThread1Handle);

        /* Suspend Thread 2 */
        osThreadSuspend(NULL);
    }
}
```

**第七步：**编译并运行测试

在调试状态下使用 Live Watch 窗口监控全部变量 `osCPU_Usage` 的值：



Expression	Value	Location
osCPU_Usage	20	0x20001194
<click to edit>		

`osCPU_Usage` 是在 `cpu_utils.c` 文件中定义的全局变量，表示当前 CPU 的使用率，是个动态值，由上图可以，CPU 使用率的动态值为 20%。实际在代码中是按第六步中调用 `osGetCPUUsage()`函数来获取当前 CPU 的使用率的。

至此，算法使用方法介绍完毕。

### 3 算法实现原理分析

操作系统运行时是不断在不同的任务间进行切换，而驱动这一调度过程是通过系统 tick 来驱动的，即每产生一次系统 tick 则检查一下当前正在运行的任务的环境判断是否需要切换任务，即调度，如果需要，则触发 PendSV，通过在 PendSV 中断调用 vTaskSwitchContext() 函数来实现任务的调度。而本文所要讲述的 CPU 使用率算法是通过在一定时间内（1000 个时间片内），计算空闲任务所占用的时间片总量，100 减去空闲任务所占百分比则为工作任务所占百分比，即 CPU 使用率。

```
void vApplicationIdleHook(void)
{
    if( xIdleHandle == NULL )
    {
        /* Store the handle to the idle task. */
        xIdleHandle = xTaskGetCurrentTaskHandle();    //记录空闲任务的句柄
    }
}
```

此函数为空闲任务钩子函数，每次当切换到空闲任务时就会运行此钩子函数，它的作用就是记录当前空闲任务的句柄并保存到全局变量 xIdleHandle。

```
void vApplicationTickHook (void)
{
    static int tick = 0;

    if(tick ++ > CALCULATION_PERIOD)    //每 1000 个 tick,刷新一次 CPU 使用率
    {
        tick = 0;

        if(osCPU_TotalIdleTime > 1000)
        {
            osCPU_TotalIdleTime = 1000;
        }
        osCPU_Usage = (100 - (osCPU_TotalIdleTime * 100) / CALCULATION_PERIOD); //这行代码就是
CPU 使用率的具体计算方法了
        osCPU_TotalIdleTime = 0;
    }
}
```

此函数为操作系统的 tick 钩子函数，即每次产生系统 tick 中断都会进入到此钩子函数。此钩子函数实际上就是具体计算 CPU 使用率的算法了。osCPU\_TotalIdleTime 是一个全局变量，表示在 1000 个 tick 时间内空闲任务总共占用的时间片，CALCULATION\_PERIOD 宏的值为 1000，即每 1000 个 tick 时间内重新计算一次 CPU 的使用率。

下面两个函数就是如何计算 osCPU\_TotalIdleTime 的：

```
void StartIdleMonitor (void)
{
    if( xTaskGetCurrentTaskHandle() == xIdleHandle ) //如果是切入到空闲任务
    {
        osCPU_IdleStartTime = xTaskGetTickCountFromISR(); //记录切入到空闲任务的时间点
    }
}
```

```
}  
void EndIdleMonitor (void)  
{  
    if( xTaskGetCurrentTaskHandle() == xIdleHandle ) //如果是从空闲任务切出  
    {  
        /* Store the handle to the idle task. */  
        osCPU_IdleSpentTime = xTaskGetTickCountFromISR() - osCPU_IdleStartTime; //计算此次空闲  
        任务花费多长时间  
        osCPU_TotalIdleTime += osCPU_IdleSpentTime; //空闲任务所占时间进行累加  
    }  
}
```

这两个函数是调度器钩子函数，在调度器进行任务切进和切出时分别回调，**StartIdleMonitor()**函数记录切换到空闲任务时的时间点，**EndIdleMonitor()**则在推出空闲任务时计算此次空闲任务花费多长时间，并累加到 **osCPU\_TotalIdleTime**，即空闲任务总共占用的时间片。

```
uint16_t osGetCPUUsage (void)  
{  
    return (uint16_t)osCPU_Usage;           //直接返回全局变量 osCPU_Usage，即 CPU 使用率  
}
```

全局变量 **osCPU\_Usage** 保存的就是 **CPU** 的使用率，它是在操作系统的 **tick** 钩子函数中每隔 1000 个 **tick** 就被重新计算一次。

## 4 结论

通过此方法可以很好的用来评估 **STM23 MCU** 的运行性能。

### 重要通知 - 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。