

## 下载代码

stm32 标准外设库是 stm32 全系列芯片的外设驱动，有了它可以大大加速我们开发 stm32。

首先从 st 公司的网站下载最新的 stm32 标准外设库，写本文时最新的版本是 V3.5.0。

解压该 zip 文件，得到如下文件夹和文件

```
STM32F10x_StdPeriph_Lib_V3.5.0\  
_htmresc  
Libraries  
Project  
Utilities  
Release_Notes.html  
stm32f10x_stdperiph_lib_um.chm
```

其中 **Libraries** 包含库的源代码，**Project** 包含 stm32 各个外设的使用范例和一个工程模板，**Utilities** 是使用 st 公司评估板的例子，**stm32f10x\_stdperiph\_lib\_um.chm** 教我们怎么用标准外设库。

## 工程目录结构

既然准备使用 32 位单片机，应该是个不小项目，因此工程目录也应该做个规划。这里我推荐一下我所使用的目录结构。假设工程名字叫 **template**，建一个名为 **template** 的文件夹，该目录下有个 3 个固定文件夹 **doc**，**src**，**include**，**doc** 用来存放工程相关的资料文件，**src** 放源代码，在 **src** 下每个功能模块一个文件夹，**include** 放各个模块都要使用的公共头文件。**output** 放编译输出文件，内含两个子文件夹 **obj** 和 **list**。

```
template\  
doc  
src  
include  
output\obj  
    \list
```

## 整理库代码

由于 Libraries 下的 CMSIS 文件夹中很多代码是和编译器及芯片相关的，导致文件夹多且深度大，不利于工程维护，实际上一个项目往往是用固定的编译器和芯片，因此有必要对库进行整理。

在 src 下建立 libstm32 目录

1. 把 Libraries\STM32F10x\_StdPeriph\_Driver\下的内容拷贝到 libstm32 目录下

2. 在 libstm32 目录下建立 cmsis 文件夹，把 Libraries\CMSIS\CM3\CoreSupport\下的 core\_cm3.c, core\_cm3.h; Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\下的 stm32f10x.h, system\_stm32f10x.c, system\_stm32f10x.h 拷贝到 cmsis 文件夹中。

3. 根据你所选的芯片类型，将 Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm\下对应的启动文件拷贝到 cmsis 文件夹中。这里我拷贝的是 startup\_stm32f10x\_hd.s（大容量型 stm32 芯片的启动文件）。

下面对该库文件做个简单介绍：

Libraries\STM32F10x\_StdPeriph\_Driver\下的内容很好理解就是 stm32 的各个外设模块驱动代码。

misc.h 和 misc.c 是和 CM3 内核有关的 NVIC 和 SysTick 的驱动代码。

Libraries\CMSIS 下是什么呢？cmsis 英文全称:Cortex Microcontroller Software Interface Standard, 是 Cortex 系列处理器硬件抽象层，可以理解为 cortex 内核的软件接口。

core\_cm3.c, core\_cm3.h

它们的目录名为 CoreSupport, 说明这两个文件是 CM3 内核支撑文件，其他使用 CM3 内核的芯片也可以用，不一定是 stm32。这两个文件用来获取设置 CM3 内核，配置一些内核寄存器。

stm32f10x.h, system\_stm32f10x.c, system\_stm32f10x.h 和 startup\_stm32f10x\_hd.s 在 DeviceSupport 目录下，说明这几个文件是和具体的芯片有关的，也就是 stm32 芯片的支撑文件。其中 stm32f10x.h 是标准外设库的入口，使用标准外设库的代码中必须包含该头文件。system\_stm32f10x.c, system\_stm32f10x.h 这两个文件提供函数用来初始化 stm32 芯片，配置 PLL、系统时钟和内置 flash 接口。startup\_stm32f10x\_hd.s 是大容量型 stm32 芯片的启动文件。

## 建立工程

使用 keil MDK（我使用 4.12 版）在 `template` 目录下建立工程，工程名为 `template`。选一个 `stm32` 系列的芯片，哪一个都无所谓（我选的是 `STM32F101RC`，因为我的板子就是用这个芯片），接下来要注意的是当弹出是否拷贝启动代码到工程文件夹时要选 `No`，因为标准外设库里已经有启动代码了。

将 UV4 中 `project window` 里的顶层目录名改为 `template`，并将第一个 `group` 名改为 `libstm32`。把 `libstm32` 目录下所有 `.c` 和 `.s` 文件加载到工程里的 `libstm32`。

在 `src` 下建立一个 `init` 目录用来放置系统初始化代码。把 `Project\STM32F10x_StdPeriph_Template\` 下的 `stm32f10x_it.c` 拷贝到 `init` 文件夹中，`stm32f10x_it.h`，`stm32f10x_conf.h` 拷贝到 `include` 文件夹中。

`stm32f10x_it.c`，`stm32f10x_it.h` 是中断服务程序文件。`stm32f10x_conf.h` 是标准外设库的配置文件，对于工程中不需要的外设，可以注释掉里面的包含的头文件。这里我建议先仅留下 `stm32f10x_gpio.h`，`stm32f10x_rcc.h`，`misc.h`，用到什么再打开什么，这样编译起来快一点，当然也可都留着。

### 使用 `stm32` 标准外设库

事实上，`stm32` 标准外设库的使用在 `stm32f10x_stdperiph_lib_um.chm` 中的 `How to use the Library` 一节中已有说明，下面我把其中的步骤罗列一下：

1. 根据所选芯片，把 `Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm` 中的启动代码加到工程中，这一步在上面已经做过了。
2. 在 `stm32f10x.h` 的 66-73 行，根据所选芯片类型，去掉相应注释，这里我去掉 `STM32F10X_HD` 行的注释（大容量型 `stm32` 芯片）。
3. 去掉 105 行的 `USE_STDPERIPH_DRIVER` 注释，启用 `stm32` 标准外设库。
4. 在 `system_stm32f10x.c` 的 110-115 行，根据所选芯片主频，去掉相应注释，默认 `SYSCLK_FREQ_72MHz` 注释已去掉，如果你的芯片主频是 72MHz，就不用做修改了，这里我的芯片是 36MHz，注释 `SYSCLK_FREQ_72MHz`，去掉 `SYSCLK_FREQ_36MHz` 注释。

## 跑马灯程序

现在可以使用 **stm32** 标准外设库了，下面以一个简单的跑马灯程序说明。  
在 **init** 目录下建立 **main.c** 作为系统入口。

在 **src** 下建立一个 **bsp** 目录用来放置板级支持代码，建立 **led.c**，**led.h**。  
代码如下：

**led.h**

```
#ifndef _LED_H_
#define _LED_H_

#include <stdint.h>

#define LED_0      0
#define LED_1      1
#define LED_2      2

void led_init(void);
void led_on(uint32_t n);
void led_off(uint32_t n);

#endif
```

**led.c**

```
#include "stm32f10x.h"
#include "led.h"

void led_init(void) {
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

```

void led_on(uint32_t n) {
    switch (n) {
        case LED_0:
            GPIO_SetBits(GPIOC, GPIO_Pin_6);
            break;
        case LED_1:
            GPIO_SetBits(GPIOC, GPIO_Pin_7);
            break;
        case LED_2:
            GPIO_SetBits(GPIOC, GPIO_Pin_8);
            break;
        default:
            break;
    }
}

```

```

void led_off(uint32_t n){
    switch (n) {
        case LED_0:
            GPIO_ResetBits(GPIOC, GPIO_Pin_6);
            break;
        case LED_1:
            GPIO_ResetBits(GPIOC, GPIO_Pin_7);
            break;
        case LED_2:
            GPIO_ResetBits(GPIOC, GPIO_Pin_8);
            break;
        default:
            break;
    }
}

```

main.c

```

#include "led.h"
static void delay(uint32_t ms){
    uint32_t count = 8000;
    while (ms--) {
        while (count--);
        count = 8000;
    }
}

```

```

int main(void){
    led_init();
    for (;;) {
        led_on(LED_0);
        led_off(LED_1);
        led_off(LED_2);
        delay(1000);

        led_off(LED_0);
        led_on(LED_1);
        led_off(LED_2);
        delay(1000);

        led_off(LED_0);
        led_off(LED_1);
        led_on(LED_2);
        delay(1000);
    }
}

```

在 project 中建立 init, bsp 组，并将各种代码加入。在工程的 Options 中，c/c++ 选项卡的 Include Paths 中添加.\include; .\src\libstm32\cmsis; .\src\libstm32\inc; .\src\bsp;。

Output 选项卡 Select Folder for Objects 中选.\output\obj。

Listing 选项卡 Select Folder for Listings 中选.\output\list。

Debug 选项卡选 use ULINK Cortex Debugger, Run to main()打钩，这一步大家可以根据自己手上的仿真器做不同选择。编译运行。

## ucosii在stm32 上的移植详解

虽然目前网上已经有不少关于 ucosii 在 stm32 上的移植版本，包括 micrium 也有官方移植版本。但这些版本具体是怎么移植出来的，又该怎么基于移植好的 ucosii 开发应用软件，网上介绍的并不多。这里介绍一下我的移植经历，希望对大家有所帮助。

我的移植基本上是从零开始的。首先想要做好移植，有两方面的内容是必须要了解。1.目标芯片；2.ucosii 内核原理。

虽然我们移植的目标芯片是 stm32，但操作系统的移植基本是针对 Cortex-M3 内核（以下简称 CM3）而言的，所以我们只需了解 CM3 内核就好了。stm32 芯片就是 CM3 内核加上各种各样的外设。

怎么才能了解 CM3 呢？看一本书<<ARM Cortex-M3 权威指南>>（宋岩译，网上多的很）就好了，很多同学可能想，看完这本书移植的新鲜劲都没了，因此我把该书和移植有关的章节都列了出来，并对其中的重点内容进行介绍，我数了数相关章节还不到 100 页，就这点内容，总要看了吧。

相关章节如下：

### chapter2 Cortex-M3 概览

#### 2.1 - 2.9

主要了解 Cortex-M3 的概貌。刚开始看时不用追求全部理解，后面会有详细介绍，很多内容多看几遍就明白。其中 2.8 指令集，只要了解，CM3 只使用 thumb2 就 ok 了。

### chapter3 Cortex-M3 基础

#### 3.1 寄存器组

R0-R12: 通用寄存器

R13: 堆栈寄存器

有两个，MSP 和 PSP，同时只能看见一个  
引用 R13 时，引用的是正在使用的那个

MSP: 可用于异常服务和应用程序

PSP: 只能用于应用程序

系统复位后，用的堆栈指针是 MSP。

R14: 连接寄存器，又名 LR，存储返回地址

R15: 程序计数寄存器，又名 PC

#### 3.2 特殊功能寄存器

程序状态字寄存器组（PSRs）

中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）

控制寄存器（CONTROL）

程序状态字寄存器组（PSRs）分为

应用程序 PSR（APSR）

中断号 PSR（IPSR）

执行 PSR（EPSR）

每个都是 32 位，由于这 3 个寄存器有效位是错开的，因此可以组合访问。

中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI），这三个寄存器用于控制异常的使能和除能。

控制寄存器（CONTROL）它有两个作用：

1. 定义特权级别
2. 选择当前使用哪个堆栈指针

### 3.3 操作模式和特权级别

操作模式：处理者模式和线程模式

异常处理：处理者模式

主程序：线程模式

ucosii 不区分特权级和用户级，程序始终工作在特权级

这两个堆栈指针的切换是全自动的，就在出入异常服务例程时由硬件处理。

### 3.4 - 3.7

没什么好讲的，需要看。

### 3.8 复位序列

0x00000000 MSP 初值

0x00000004 PC 初值 复位向量

## chapter7 异常

### 7.1 异常类型

分为系统异常(编号 1-15)和外部中断(大于 15)

### 7.2 优先级

CM3 支持 3 个固定的高优先级和多达 256 级的可编程优先级。

在 NVIC 中，每个中断都有一个优先级配置寄存器（1 个 byte），用来配置该中断的优先级。但该寄存器并不是每个位都被使用，不同制造商生产的芯片不相同，譬如 stm32 使用 4 位，也就是说 stm32 支持 16 个可编程优先级（参考：chapter9）。

注意该寄存器是以 MSB 对齐的，因此 stm32 每个中断的优先级配置寄存器 7:4 位有效，3:0 位无效。

对于优先级，CM3 又分为抢占优先级和亚优先级，

NVIC 中的应用程序中断及复位控制寄存器(AIRCR)的优先级分组(10:8)描述了如何划分抢占优先级和亚优先级。

什么意思？以 stm32 为例，优先级配置寄存器不是 7:4 位有效吗，如果 AIRCR 中的优先级分组值为 4，则优先级配置寄存器的 7:5 位确定抢占优先级，位 4 确定亚优先级。此时所有中断有 8 个抢占优先级，每个抢占优先级有 2 个亚优先级。

抢占优先级高的中断可以抢占抢占优先级低的中断，即抢占优先级决定了中断是否可以嵌套。

相同抢占优先级的中断不能嵌套，但当抢占优先级相同的异常有不只一个到来时，就优先响应亚优先级最高的异常。

参考附录 D

表 D.9 中断优先级寄存器阵列 0xE000\_E400 - 0xE000\_E4EF 共 240 个。

表 D.16 系统异常优先级寄存器 0xE000\_ED18 - 0xE000\_ED23 共 12 个。

优先级相同，看中断号，中断号小的优先。



### 7.3 向量表

初始在 0x00000000 处，可以通过向量表偏移量寄存器(VTOR)（地址：0xE000\_ED08）更改，一般无需更改。

### 7.4 中断输入及挂起行为

需要看。

### 7.5 Fault 异常

可不看。

### 7.6 SVC 和 PendSV

#### SVC

SVC 主要用在分特权级和用户级的操作系统，ucosii 不区分特权级和用户级，可以不管这个东西。

这里说点题外话，一开始我很奇怪为什么会提供这种中断，因为这种中断一般都是用在大型的操作系统上，如 linux 系统上，可 CM3 又不提供 MMU，应该是无法移植 linux 系统。后来我才知道 uclinux 是针对没有 MMU 的嵌入式系统而设计的，不过还是很怀疑有人会在像 stm32 这种芯片上用 uclinux。

#### PendSV

PendSV 中断主要做上下文切换，也就是任务切换，是 ucosii 移植过程中最重要的中断。

主要有两点：

1. PendSV 中断是手工往 NVIC 的 PendSV 悬起寄存器中写 1 产生的(由 OS 写)。

2. PendSV 中断优先级必须设为最低。

在讲移植代码时会介绍具体是如何做的。

对于 7.6 的 PendSV 部分应认真研读一下。

### chapter8 NVIC 与中断控制

NVIC 负责芯片的中断管理，它和 CM3 内核紧密相关。

如果对于 CM3 中断配置不是很了解，可以看看 8.1, 8.2, 8.3, 8.4 节。

8.7 节讲述了 SysTick 定时器，需要看。

### chapter9 中断的具体行为

#### 9.1 中断 / 异常的响应序列

当 CM3 开始响应一个中断时

1. xPSR, PC, LR, R12 以及 R3 - R0 入栈

2. 取向量

3. 选择堆栈指针 MSP/PSP，更新堆栈指针 SP，更新连接寄存器 LR，更新程序计数器 PC

对移植 ucosii 来说，需要注意 1,3

#### 9.2 异常返回

在 CM3 中，进入中断时，LR 寄存器的值会被自动更新。9.6 节对更新后的值进行说明。这里统称 EXC\_RETURN。返回时通过把 EXC\_RETURN 往 PC 里写来识别返回动作的。因为 EXC\_RETURN 是一个特殊值，所以对于 CM3，汇编语言就不需要类似 reti 这种指令，而用 C 语言开发时，不需要特殊编译器命令指示一个函

数为中断服务程序。实际上，中断服务程序如果是 c 代码编写，汇编成汇编代码，函数结尾一般是 `reti`。

### 9.3 嵌套的中断

只要注意：中断嵌套不能过深即可。

### 9.4 和 9.5

这两节说明 **CM3** 对中断的响应能力大大提高了，主要是硬件机制的改进。但对移植来说，并不需要关注。

### 9.6 异常返回值

对不同状态进入中断时，**LR** 寄存器的值进行说明，需要看。这里有一点需要注意，该点在讲移植代码时再介绍。

### 9.7 和 9.8

对移植来说，并不需要关注。

## chapter10 Cortex-M3 的低层编程

这一章仅需关注 10.2 节，因为对移植来说汇编与 **C** 的接口是必须面对的。

### 10.2 汇编与 C 的接口

有两点需要知道：

1. 当主调函数需要传递参数（实参）时，它们使用 **R0 - R3**。其中 **R0** 传递第一个，**R1** 传递第 2 个.....在返回时，把返回值写到 **R0** 中。

2. 在函数中，用汇编写代码时，**R0-R3, R12** 可以随便使用，而使用 **R4 - R11**，则必须先 **PUSH**，后 **POP**。

以上内容和移植多少都有些关系，刚开始看，可能不太明白，多看几遍就好了。

## ucosii在stm32 上的移植详解 2

在详解 1 中主要讲了移植需要用到的 CM3 内核知识,本文讲一讲 ucosii 的原理和代码组成。ucosii 最经典的学习资料莫过于邵贝贝老师的<<嵌入式实时操作系统 uc/os-ii(第 2 版)>>,我想这本书对学 ucosii 已经足够了,因为他把 ucosii V2.55 代码都讲了一遍。移植前应该好好看看此书。

下面说说我对 ucosii 的理解。应该说 ucosii 这个内核还是比较简单的,基本可以分为任务调度,任务同步和内存管理三个部分。

### 任务调度

ucosii 为保证实时性,给每个任务分配一个不同的优先级。当发生任务切换时,总是切换到就绪的最高优先级任务。有 2 种情况会发生任务切换。

- 1.任务等待资源就绪或自我延时;
- 2.退出中断;

情况 1 可以理解为任务主动放弃 cpu 的使用权。

情况 2 可以理解为中断后,某种资源可能就绪了,需要任务切换。

需要注意的是 SysTick 中断,这个中断是 os 的“心跳”,必须得有。这样就使得 cpu 会发生周期性地做任务切换。由于 ucosii 不支持时间片轮转调度,因此在该中断中必须做的工作仅有 os 的时间管理。也就是调用 OSTimeTick()。

### 任务同步

任务同步和大多数操作系统的做法差不多,如果学过操作系统或是有多线程编程经验的话,应该很好理解。无非是任务 A 因为某个资源未就绪,就放弃 cpu 使用权,等任务 B 或是中断使该资源就绪,当再次任务进行切换时如果任务 A 优先级最高,则任务 A 继续执行。具体怎么实现就看邵老师的书吧。

### 内存管理

ucosii 的内存管理比较简单,就不说了。

下面看看 ucosii 代码组成:

os\_core.c 是 ucosii 的核心,它包含了内核初始化,任务切换,事件块管理等,其中事件块是各个同步量(这里我把互斥量,信号量,邮箱,队列统称为同步量,不是很科学,图个方便。事件标志组不是以事件块为基础的,不过原理也差不多)的基础。

os\_task.c 任务管理代码。

os\_flag.c

os\_mbox.c

os\_mutex.c

os\_q.c

os\_sem.c 各个同步量管理代码。

os\_mem.c 内存管理代码。

os\_time.c 时间管理代码，主要做各种延时。

os\_tmr.c

定时器管理代码，这部分代码是从 V2.81 版才开始有的，邵老师的书讲的是 V2.55 版的代码，是没有这部分内容的。如果前面的代码都理解的话，这部分代码也是不难理解的。一个定时器大体由 3 部分组成：定时时间，回调函数和属性。当定时时间到了的话，就进行一次回调函数的处理，定时器属性说明定时器是周期性的定时还是只做一次定时。如果用户使能了 OS\_TMR\_EN，ucosii 会在内部创建一个定时器任务，负责处理各个定时器。这个任务一般应该由硬件定时器的中断函数中调用 OSTmrSignal() 去激活。所以从本质上说 os\_tmr.c 中的定时器是由一个硬件定时器分化出来的。

默认情况下是由 SysTick 中断里通过 OSTimeTickHook() 去激活定时器任务的。

## 移植相关文件

os\_cpu.h: 进行数据类型定义，处理器相关代码和几个函数原型。

os\_cpu\_c.c: 定义一些用户 hook 函数。

os\_cpu\_a.asm: 移植需要用汇编代码完成的函数，主要就是任务切换函数。

os\_dbg.c: 内核调试相关数据和函数，可以不改。

ucosii 内核就介绍到这里。

### ucosii在stm32 上的移植详解 3

移植详解 1 和 2 中主要讲了移植需要用到的基础知识，本文则对具体的移植过程进行介绍。

首先从 micrium 网站上下载官方移植版本（编译器使用 ARM/Keil 的，V2.86 版本，V2.85 有问题）。

下载地址：<http://micrium.com/page/downloads/ports/st/stm32>

解压缩后得到如下文件夹和文件：

Micrium\  
AppNotes  
Licensing  
Software  
ReadMe.pdf

AppNotes 包含 ucosii 移植说明文件。这两个文件中我们仅需关心 Micrium\AppNotes\AN1xxx-RTOS\AN1018-uCOS-II-Cortex-M3\AN-1018.pdf。因为这个文件对 ucosii 在 CM3 内核移植过程中需要修改的代码进行了说明。

Licensing 包含 ucosii 使用许可证。

Software 下有好几个文件夹，在本文的移植中仅需关心 uCOS-II 即可。

CPU: stm32 标准外设库

EvalBoards: micrium 官方评估板相关代码

uc-CPU: 基于 micrium 官方评估板的 ucosii 移植代码

uC-LCD: micrium 官方评估板 LCD 驱动代码

uc-LIB: micrium 官方的一个库代码

uCOS-II: ucosii 源代码

uC-Probe: 和 uC-Probe 相关代码

ReadMe.pdf 就不说了。

好了，官方的东西介绍完了，该我们自己建立工程着手移植了。关于建立工程，并使用stm32 标准外设库在我之前的文章《stm32 标准外设库使用详解》已有介绍，这里请大家下载其中模板代码（<http://download.csdn.net/source/3448543>），本文的移植是基于这个工程的。

建立文件夹

template\src\ucosii  
template\src\ucosii\src  
template\src\ucosii\port;

把 Micrium\Software\uCOS-II\Source 下的文件拷贝至 template\src\ucosii\src;  
把 Micrium\Software\uCOS-II\Ports\ARM-Cortex-M3\Generic\RealView 下的文件拷贝至 template\src\ucosii\port;

ucosii\src 下的代码是 ucosii 中无需修改部分

ucosii\port 下的代码是移植时需要修改的。为防止对源码的误改动造成移植失败，可以把 ucosii\src 下的代码文件设为只读。

这里根据 AN-1018.pdf 和移植详解 1、2 中介绍的移植基础知识，对 ucosii\port 下的代码解释一下。

os\_cpu.h

```
#ifndef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif

typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;
typedef signed char INT8S;
typedef unsigned short INT16U;
typedef signed short INT16S;
typedef unsigned int INT32U;
typedef signed int INT32S;
typedef float FP32;
typedef double FP64;
就不解释了。
typedef unsigned int OS_STK;
typedef unsigned int OS_CPU_SR;
```

因为 CM3 是 32 位宽的，所以 OS\_STK（堆栈的数据类型）被类型重定义为 unsigned int。

因为 CM3 的状态寄存器（xPSR）是 32 位宽的，因此 OS\_CPU\_SR 被类型重定义为 unsigned int。OS\_CPU\_SR 是在 OS\_CRITICAL\_METHOD 方法 3 中保存 cpu 状态寄存器用的。在 CM3 中，移植 OS\_ENTER\_CRITICAL(), OS\_EXIT\_CRITICAL()选方法 3 是最合适的。

```
#define OS_CRITICAL_METHOD 3
```

```
#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() {cpu_sr = OS_CPU_SR_Save();}
#define OS_EXIT_CRITICAL() {OS_CPU_SR_Restore(cpu_sr);}
#endif
```

具体定义宏 `OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()`，其中 `OS_CPU_SR_Save()` 和 `OS_CPU_SR_Restore()` 是用汇编代码写的，代码在 `os_cpu_a.asm` 中，到时再解释。

```
#define OS_STK_GROWTH 1
```

`CM3` 中，栈是由高地址向低地址增长的，因此 `OS_STK_GROWTH` 定义为 1。

```
#define OS_TASK_SW() OSCtxSw()
```

定义任务切换宏，`OSCtxSw()` 是用汇编代码写的，代码在 `os_cpu_a.asm` 中，到时再解释。

```
#if OS_CRITICAL_METHOD == 3
OS_CPU_SR OS_CPU_SR_Save(void);
void OS_CPU_SR_Restore(OS_CPU_SR cpu_sr);
#endif

void OSCtxSw(void);
void OSIntCtxSw(void);
void OSStartHighRdy(void);
void OS_CPU_PendSVHandler(void);
```

```
void OS_CPU_SysTickHandler(void);
void OS_CPU_SysTickInit(void);
INT32U OS_CPU_SysTickClkFreq(void);
```

申明几个函数，这里要注意最后三个函数需要注释掉，为什么呢？

`OS_CPU_SysTickHandler()` 定义在 `os_cpu.c.c` 中，是 `SysTick` 中断的中断处理函数，而 `stm32f10x_it.c`，中已经有该中断函数的定义 `SysTick_Handler()`，这里也就不需要了，是不是很奇怪官方移植版为什么会这样弄吧，后面我会解释的。

`OS_CPU_SysTickInit()` 定义在 `os_cpu.c.c` 中，用于初始化 `SysTick` 定时器，它依赖于 `OS_CPU_SysTickClkFreq()`，而此函数我们自己会实现，所以注释掉。

`OS_CPU_SysTickClkFreq()` 定义在 `BSP.C` (`Micrium\Software\EvalBoards`) 中，而本文移植中并未用到 `BSP.C`，后面我们会自己实现，因此可以把它注释掉。

os\_cpu\_c.c

ucosii 移植时需要我们写 10 个相当简单的 C 函数。

OSInitHookBegin()

OSInitHookEnd()

OSTaskCreateHook()

OSTaskDelHook()

OSTaskIdleHook()

OSTaskStatHook()

OSTaskStkInit()

OSTaskSwHook()

OSTCBInitHook()

OSTimeTickHook()

这些函数除了 OSTaskStkInit(), 都是一些 hook 函数。这些 hook 函数如果不使用的话, 都不会用上, 也都比较简单, 看看就应该明白了, 所以就不介绍。

下面就说一说 OSTaskStkInit()。说之前还是得先说一下任务切换, 因为初始化任务堆栈, 是为任务切换服务的。代码在正常运行时, 一行一行往下执行, 怎么才能跑到另一个任务 (即函数) 执行呢? 首先大家可以回想一下中断过程, 当中断发生时, 原来函数执行的地方 (程序计数器 PC、处理器状态寄存器及所有通用寄存器, 即当前代码的现场) 被保存到栈里面去了, 然后开始取中断向量, 跑到中断函数里面执行。执行完了呢, 想回到原来函数执行的地方, 该怎么办呢, 只要把栈中保存的原来函数执行的信息恢复即可 (把栈中保存的代码现场重新赋给 cpu 的各个寄存器), 一切就都回去了, 好像什么事都没发生一样。这个过程大家应该都比较熟悉, 任务切换和这有什么关系, 试想一下, 如果有 3 个函数 foo1(), foo2(), foo3() 像是刚被中断, 现场保存到栈里面去了, 而中断返回时做点手脚 (调度程序的作用), 想回哪个回哪个, 是不是就做了函数 (任务) 切换了。看到这里应该有点明白 OSTaskStkInit() 的作用了吧, 它被任务创建函数调用, 所以要在开始时, 在栈中作出该任务好像刚被中断一样的假象。(关于任务切换的原理邵老师书中的 3.06 节有介绍)。

那么中断后栈中是个什么情形呢, <<ARM Cortex-M3 权威指南>>中 9.1.1 有介绍, xPSR, PC, LR, R12, R3-R0 被自动保存到栈中的, R11-R4 如果需要保存, 只能手工保存。因此 OSTaskStkInit() 的工作就是在任务自己的栈中保存 cpu 的所有寄存器。这些值里 R1-R12 都没什么意义, 这里用相应的数字代号 (如 R1 用 0x01010101) 主要是方便调试。



其他几个：

xPSR = 0x01000000L, xPSR T 位（第 24 位）置 1，否则第一次执行任务时 Fault,

PC 肯定得指向任务入口，

R14 = 0xFFFFFFFFEL, 最低 4 位为 E，是一个非法值，主要目的是不让使用 R14，即任务是不能返回的。

R0 用于传递任务函数的参数，因此等于 p\_arg。

```
OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK
*ptos, INT16U opt) {
    OS_STK *stk;
    (void)opt;          /* 'opt' is not used, prevent warning */
    stk    = ptos;      /* Load stack pointer */
    /* Registers stacked as if auto-saved on exception */
    *(stk)  = (INT32U)0x01000000L; /* xPSR */
    *(--stk) = (INT32U)task;      /* Entry Point */
    /* R14 (LR) (init value will cause fault if ever used) */
    *(--stk) = (INT32U)0xFFFFFFFFEL;
    *(--stk) = (INT32U)0x12121212L; /* R12 */
    *(--stk) = (INT32U)0x03030303L; /* R3 */
    *(--stk) = (INT32U)0x02020202L; /* R2 */
    *(--stk) = (INT32U)0x01010101L; /* R1 */
    *(--stk) = (INT32U)p_arg;      /* R0 : argument */
    /* Remaining registers saved on process stack */
    *(--stk) = (INT32U)0x11111111L; /* R11 */
    *(--stk) = (INT32U)0x10101010L; /* R10 */
    *(--stk) = (INT32U)0x09090909L; /* R9 */
    *(--stk) = (INT32U)0x08080808L; /* R8 */
    *(--stk) = (INT32U)0x07070707L; /* R7 */
    *(--stk) = (INT32U)0x06060606L; /* R6 */
    *(--stk) = (INT32U)0x05050505L; /* R5 */
    *(--stk) = (INT32U)0x04040404L; /* R4 */
    return (stk);
}
```

把 OS\_CPU\_SysTickHandler(), OS\_CPU\_SysTickInit()注释掉。

```
#define OS_CPU_CM3_NVIC_ST_CTRL  (*((volatile INT32U *)0xE000E010))
#define OS_CPU_CM3_NVIC_ST_RELOAD  (*((volatile INT32U
*)0xE000E014))
#define OS_CPU_CM3_NVIC_ST_CURRENT  (*((volatile INT32U
*)0xE000E018))
#define OS_CPU_CM3_NVIC_ST_CAL  (*((volatile INT32U *)0xE000E01C))
#define OS_CPU_CM3_NVIC_ST_CTRL_COUNT  0x00010000
#define OS_CPU_CM3_NVIC_ST_CTRL_CLK_SRC  0x00000004
#define OS_CPU_CM3_NVIC_ST_CTRL_INTEN  0x00000002
#define OS_CPU_CM3_NVIC_ST_CTRL_ENABLE  0x00000001
```

把上面这些宏定义也注释掉，因为它们都用于 OS\_CPU\_SysTickHandler(), OS\_CPU\_SysTickInit()。

os\_cpu\_a.asm

这个文件包含着必须用汇编写的代码。

```
EXTERN OSRunning    ; External references
EXTERN OSPrioCur
EXTERN OSPrioHighRdy
EXTERN OSTCBCur
EXTERN OSTCBHighRdy
EXTERN OSIntNesting
EXTERN OSIntExit
EXTERN OSTaskSwHook
```

申明这些变量是在其他文件定义的，本文件只做引用（有几个好像并未引用，不过没有关系）。

```
EXPORT OS_CPU_SR_Save    ; Functions declared in this file
EXPORT OS_CPU_SR_Restore
EXPORT OSStartHighRdy
EXPORT OSCtxSw
EXPORT OSIntCtxSw
EXPORT OS_CPU_PendSVHandler
```

申明这些函数是在本文件中定义的。

```
NVIC_INT_CTRL EQU 0xE000ED04 ;中断控制及状态寄存器 ICSR 的地址
NVIC_SYSPRI14 EQU 0xE000ED22 ;PendSV 优先级寄存器的地址
```

NVIC\_PENDSV\_PRI EQU 0xFF ;PendSV 中断的优先级为 255（最低）

NVIC\_PENDSVSET EQU 0x10000000 ;位 28 为 1

定义几个常量，类似 C 语言中的#define 预处理指令。

OS\_CPU\_SR\_Save

MRS R0, PRIMASK ;读取 PRIMASK 到 R0 中，R0 为返回值

CPSID 1 ;PRIMASK=1，关中断（NMI 和硬 fault 可以响应）

BX LR ;返回

OS\_CPU\_SR\_Restore

MSR PRIMASK, R0 ;读取 R0 到 PRIMASK 中，R0 为参数

BX LR ;返回

OSStartHighRdy()由 OSStart()调用，用来启动最高优先级任务，当然任务必须在 OSStart()前已被创建。

OSStartHighRdy

;设置 PendSV 中断的优先级 #1

LDR R0, =NVIC\_SYSPRI14 ;R0 = NVIC\_SYSPRI14

LDR R1, =NVIC\_PENDSV\_PRI ;R1 = NVIC\_PENDSV\_PRI

STRB R1, [R0];\*(uint8\_t\*)NVIC\_SYSPRI14 = NVIC\_PENDSV\_PRI

;设置 PSP 为 0 #2

MOVS R0, #0 ;R0 = 0

MSR PSP, R0 ;PSP = R0

;设置 OSRunning 为 TRUE

LDR R0, =OSRunning ;R0 = OSRunning

MOVS R1, #1 ;R1 = 1

STRB R1, [R0];OSRunning = 1

;触发 PendSV 中断 #3

LDR R0, =NVIC\_INT\_CTRL ;R0 = NVIC\_INT\_CTRL

LDR R1, =NVIC\_PENDSVSET ;R1 = NVIC\_PENDSVSET

STR R1, [R0];\*(uint32\_t\*)NVIC\_INT\_CTRL = NVIC\_PENDSVSET

CPSIE 1 ;开中断

OSStartHang ;死循环，应该不会到这里

B OSStartHang

#1.PendSV 中断的优先级应该为最低优先级，原因在<<ARM Cortex-M3 权威指南>>的 7.6 节已有说明。

#2.PSP 设置为 0，是告诉具体的任务切换程序（OS\_CPU\_PendSVHandler()），这是第一次任务切换。做过切换后 PSP 就不会为 0 了，后面会看到。

#3.往中断控制及状态寄存器 ICSR(0xE00ED04)第 28 位写 1 即可产生 PendSV 中断。这个<<ARM Cortex-M3 权威指南>>8.4.5 其它异常的配置寄存器有说明。

当一个任务放弃 cpu 的使用权，就会调用 OS\_TASK\_SW()宏，而 OS\_TASK\_SW()就是 OSCtxSw()。OSCtxSw()应该做任务切换。但是在 CM3 中，所有任务切换都被放到 PendSV 的中断处理函数中去做了，因此 OSCtxSw()只需简单的触发 PendSV 中断即可。OS\_TASK\_SW()是由 OS\_Sched()调用。

```
void OS_Sched(void)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr = 0;
    #endif

    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) {
        if (OSLockNesting == 0) {
            OS_SchedNew();
            if (OSPrioHighRdy != OSPrioCur) {
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            #if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++;
            #endif

            OSCtxSwCtr++;
            OS_TASK_SW(); /* 触发 PendSV 中断 */
        }
    }
}

/* 一旦开中断，PendSV 中断函数会执行（当然要等更高优先级中断处理完） */
OS_EXIT_CRITICAL();
}
```

## OSCtxSw

;触发 PendSV 中断

```
LDR    R0, =NVIC_INT_CTRL    ;R0 = NVIC_INT_CTRL
```

```
LDR    R1, =NVIC_PENDSVSET    ;R1 = NVIC_PENDSVSET
```

```
STR    R1, [R0]                ;*(uint32_t *)NVIC_INT_CTRL = NVIC_PENDSVSET
```

```
BX     LR                      ;返回
```

当一个中断处理函数退出时，OSIntExit()会被调用来决定是否优先级更高的任务需要执行。如果有OSIntExit()对调用OSIntCtxSw()做任务切换。

## OSIntCtxSw

;触发 PendSV 中断

```
LDR    R0, =NVIC_INT_CTRL
```

```
LDR    R1, =NVIC_PENDSVSET
```

```
STR    R1, [R0]
```

```
BX     LR
```

看到这里有些同学可能奇怪怎么OSCtxSw()和OSIntCtxSw()完全一样，事实上，这两个函数的意义是不一样的，OSCtxSw()做的是任务之间的切换，如任务A因为等待某个资源或是做延时切换到任务B，而OSIntCtxSw()则是中断退出时，由中断状态切换到另一个任务。由中断切换到任务时，CPU寄存器入栈的工作已经做完了，所以无需做第二次了（参考邵老师书的3.10节）。这里只不过由于CM3的特殊机制导致了在这两个函数中只要做触发PendSV中断即可，具体切换由PendSV中断来处理。

前面已经说过真正的任务切换是在PendSV中断处理函数里做的，由于CM3在中断时会有有一半的寄存器自动保存到任务堆栈里，所以在PendSV中断处理函数中只需保存R4-R11并调节堆栈指针即可。

PendSV中断处理函数伪代码如下：

```
OS_CPU_PendSVHandler()
```

```
{
```

```
    if (PSP != NULL) {
```

```
        Save R4-R11 onto task stack;
```

```
        OSTCBCur->OSTCBStkPtr = SP;
```

```
    }
```

```
    OSTaskSwHook();
```

```
    OSPrioCur = OSPrioHighRdy;
```

```
    OSTCBCur = OSTCBHighRdy;
```

```

    PSP = OSTCBHighRdy->OSTCBStkPtr;
    Restore R4-R11 from new task stack;
    Return from exception;
}

```

**OS\_CPU\_PendSVHandler** ;xPSR, PC, LR, R12, R0-R3 已自动保存

```

CPSID    I                ;任务切换期间需要关中断
MRS      R0, PSP          ;R0 = PSP
;如果 PSP == 0, 跳到 OS_CPU_PendSVHandler_nosave 执行 #1
CBZ      R0, OS_CPU_PendSVHandler_nosave
;保存 R4-R11 到任务堆栈
SUBS     R0, R0, #0x20     ;R0 -= 0x20
STM      R0, {R4-R11}     ;保存 R4-R11 到任务堆栈
;OSTCBCur->OSTCBStkPtr = SP;
LDR      R1, =OSTCBCur    ;R1 = &OSTCBCur
LDR      R1, [R1]         ;R1 = *R1 (R1 = OSTCBCur)
STR      R0, [R1]         ;*R1 = R0 (*OSTCBCur = SP) #2

```

**OS\_CPU\_PendSVHandler\_nosave**

```

;调用 OSTaskSwHook()
PUSH     {R14}            ;保存 R14, 因为后面要调用函数
LDR      R0, =OSTaskSwHook ;R0 = &OSTaskSwHook
BLX      R0               ;调用 OSTaskSwHook()
POP      {R14}            ;恢复 R14
;OSPrioCur = OSPrioHighRdy;
LDR      R0, =OSPrioCur   ;R0 = &OSPrioCur
LDR      R1, =OSPrioHighRdy ;R1 = &OSPrioHighRdy
LDRB     R2, [R1]         ;R2 = *R1 (R2 = OSPrioHighRdy)
STRB     R2, [R0]         ;*R0 = R2 (OSPrioCur = OSPrioHighRdy)
;OSTCBCur = OSTCBHighRdy;
LDR      R0, =OSTCBCur     ;R0 = &OSTCBCur
LDR      R1, =OSTCBHighRdy ;R1 = &OSTCBHighRdy
LDR      R2, [R1]         ;R2 = *R1 (R2 = OSTCBHighRdy)
STR      R2, [R0]         ;*R0 = R2 (OSTCBCur = OSTCBHighRdy)

```

LDR R0, [R2] ;R0 = \*R2 (R0 = OSTCBHighRdy), 此时 R0 是新任务的 SP

;SP = OSTCBHighRdy->OSTCBStkPtr #3

LDM R0, {R4-R11} ;从任务堆栈 SP 恢复 R4-R11

ADDs R0, R0, #0x20 ;R0 += 0x20

MSR PSP, R0 ;PSP = R0, 用新任务的 SP 加载 PSP

ORR LR, LR, #0x04 ;确保 LR 位 2 为 1, 返回后使用进程堆栈 #4

CPSIE I ;开中断

BX LR ;中断返回

END

#1 如果 PSP == 0, 说明 OSStartHighRdy()启动后第一次做任务切换, 而任务刚创建时 R4-R11 已经保存在堆栈中了, 所以不需要再保存一次了。

#2 OSTCBStkPtr 是任务控制块结构体的第一个变量, 所以\*OSTCBCur = SP(不是很科学)就是 OSTCBCur->OSTCBStkPtr = SP;

#3 和#2 类似。

#4 因为在中断处理函数中使用的是 MSP, 所以在返回任务后必须使用 PSP, 所以 LR 位 2 必须为 1。

os\_dbg.c

用于系统调试, 可以不管。

需要修改的代码就介绍到这里, 如果还有不明白之处, 就再看看 AN-1018.pdf, 邵老师的书和<<ARM Cortex-M3 权威指南>>。

## ucosii在stm32 上的移植详解 4

详解 3 中有一个问题还没解释，就是 `stm32f10x_it.c` 中已经有 `SysTick` 中断函数的定义 `SysTick_Handler()`，为什么官方版非要弄个 `OS_CPU_SysTickHandler()`。答案就在启动文件上，一般我们自己开发基于 `stm32` 芯片的软件，都会使用标准外设库 `CMSIS` 中提供的启动文件，而官方移植的启动文件却是自己写的，在两个文件 `init.s`，`vectors.s` 中

(`Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK`)。`init.s` 负责进入 `main()`，`vectors.s` 设置中断向量。`OS_CPU_SysTickHandler` 和 `OS_CPU_PendSVHandler` 就是在 `vectors.s` 中被设置的。

我的移植是使用标准外设库 `CMSIS` 中 `startup_stm32f10x_hd.s` 作为启动文件的，那该怎么在这个文件中设置 `OS_CPU_SysTickHandler` 呢，事实上在 `startup_stm32f10x_hd.s` 文件中，`PendSV` 中断向量名为 `PendSV_Handler`，所以只需用 `OS_CPU_PendSVHandler` 把所有出现 `PendSV_Handler` 的地方替换掉就可以了。

那么为什么 `OS_CPU_SysTickHandler` 不用这种方式处理呢，这样也就不用注释 `os_cpu.c` 中的 `OS_CPU_SysTickHandler()`，这主要是基于两个原因：

1. `startup_stm32f10x_hd.s` 尽量少该，能不改就不改。

2. 如果保留 `OS_CPU_SysTickHandler()`，在以后开发过程中，改动 `OS_CPU_SysTickHandler()` 中的内容可能性是非常大的，如果一不小心将该文件其他部分改了造成了问题，这个 `bug` 就非常难查了，所以我一般移植好后就把 `ucosii` 的这些文件设置为只读。

对于上面的原因 1，一开始移植时，我曾做过在 `PendSV_Handler()` 中调用 `OS_CPU_PendSVHandler()`，后来发现这样不行，这是为什么呢？问题出在 `LR` 寄存器上。

```
PendSV_Handler()
{
    OS_CPU_PendSVHandler();
}
```

汇编出来的代码会是这样：

```
PendSV_Handler PROC
    PUSH    {r4,lr}
    BL      OS_CPU_PendSVHandler
    POP     {r4,pc}
ENDP
```



这样在进入 OS\_CPU\_PendSVHandler 之后，LR 寄存器中存放的是指令 POP {r4,pc}的地址+1。在 OS\_CPU\_PendSVHandler 中的 ORR LR, LR, #0x04 就不会起作用，也就无法使用 PSP，移植因此失败。其实在 AN-1018.pdf 的 3.04.06 中也有强调 OS\_CPU\_PendSVHandler 必须被放置在中断向量表中。一开始我也没注意。

到这里移植的大部分工作都做完了，下面剩下的就是把工程配置好，SysTick 中断处理好。

在工程中建立 ucossii 组，把 ucossii 下的文件都加进该组。这里别忘了把 os\_cpu\_a.asm 加入。

在工程的 Options 中，c/c++选项卡的 Include Paths 中添加.\src\ucossii\src;.\src\ucossii\port。

编译工程，会发现缺少 app\_cfg.h 和 os\_cfg.h 文件，app\_cfg.h 是用来配置应用程序的，主要是任务的优先级和堆栈大小，中断优先级等信息。目前还没有基于 ucossii 开发应用程序，所以只需在 include 文件夹中创建一个空的 app\_cfg.h 文件即可。os\_cfg.h 是用来配置 ucossii 系统的。拷贝

Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK\OS-Probe\os\_cfg.h 到 template\include，对其做如下修改：

```
#define OS_APP_HOOKS_EN      0
#define OS_DEBUG_EN          0
#define OS_EVENT_MULTI_EN    0
#define OS_SCHED_LOCK_EN     0
#define OS_TICK_STEP_EN      0
#define OS_TASK_CHANGE_PRIO_EN 0
#define OS_TASK_QUERY_EN      0
#define OS_TASK_STAT_EN       0
#define OS_TASK_STAT_STK_CHK_EN 0
#define OS_TASK_SUSPEND_EN    0
#define OS_FLAG_EN            0
#define OS_MBOX_EN            0
#define OS_TIME_DLY_HMSM_EN   0
#define OS_TIME_DLY_RESUME_EN 0
#define OS_TIME_GET_SET_EN     0
#define OS_TIME_TICK_HOOK_EN  0
```

所做的修改主要是把一些功能给去掉，减少内核大小，也利于调试。等移植完成后，如果需要该功能，再做开启。

接下来就剩下处理好 SysTick 中断和启动任务了。SysTick 是系统的“心跳”，本质上来说就是一个定时器。先把原来 main.c 中的内容删除，添加如下代码：

```
#include "ucos_ii.h"
#include "stm32f10x.h"
static OS_STK startup_task_stk[STARTUP_TASK_STK_SIZE];
static void systick_init(void)
{
    RCC_ClocksTypeDef rcc_clocks;
    RCC_GetClocksFreq(&rcc_clocks);
    SysTick_Config(rcc_clocks.HCLK_Frequency / OS_TICKS_PER_SEC);
}
static void startup_task(void *p_arg)
{
    systick_init();    /* Initialize the SysTick. */
#ifdef OS_TASK_STAT_EN
    OSStatInit();      /* Determine CPU capacity. */
#endif
    /* TODO: create application tasks here */

    OSTaskDel(OS_PRIO_SELF);
}

int main(void)
{
    OSInit();
    OSTaskCreate(startup_task, (void *)0,
        &startup_task_stk[STARTUP_TASK_STK_SIZE - 1],
        STARTUP_TASK_PRIO);
    OSStart();
    return 0;
}
```

systick\_init()用来初始化并启动 SysTick 定时器。

RCC\_GetClocksFreq()用来获取系统时钟。

SysTick\_Config()初始化并使能 SysTick 定时器。

这里要注意的是 `OS_TICKS_PER_SEC`，它是每秒钟的 `ticks` 数，如果为 1000，就是 1s 中 1000 个 `ticks`，也就是说 1ms 就会产生一个 `SysTick` 中断。系统的时间片为 1ms。

在邵老师的书中 3.11 节已有明确说明，必须在调用 `OSStart()` 之后，才能开启时钟节拍器（`SysTick`）。一般会把它放在第一个任务（启动任务）中。

`startup_task()` 用来创建其他应用任务，创建完其他任务后，就会自己删除自己。

文件中的 `STARTUP_TASK_STK_SIZE`，`STARTUP_TASK_PRIO` 需要在 `app_cfg.h` 中定义。代码如下：

```
/* task priority */
#define STARTUP_TASK_PRIO          4
/* task stack size */
#define STARTUP_TASK_STK_SIZE      80
```

在 `stm32f10x_it.c` 中，还需要添加 `SysTick` 中断的处理代码：

```
void SysTick_Handler(void)
{
    OSIntEnter();
    OSTimeTick();
    OSIntExit();
}
```

这个代码是仿照 `OS_CPU_SysTickHandler()` 中代码的，在邵老师书的 3.11 节亦有说明。这里就不解释。

至此 `ucosii` 在 `stm32` 上的移植已全部完成。

## ucosii在stm32 上的移植详解 5

详解 1-4 把移植过程都已经介绍了。接下来的工作是验证移植是否 ok 以及如何基于移植好的 ucosii 开发应用程序。前一个问题可以说是后一个问题的特殊情况，一般我们会创建两个简单的任务，看看任务切换是否成功来验证移植是否 ok，因为任务切换可以说是 ucosii 最核心的功能。

任务代码(main.c):

```
static void task1(void *p_arg)
{
    for (;;)
    {
        led_on(LED_0);
        OSTimeDly(500);
        led_off(LED_0);
        OSTimeDly(500);
    }
}

static void task2(void *p_arg)
{
    for (;;)
    {
        led_on(LED_1);
        OSTimeDly(500);
        led_off(LED_1);
        OSTimeDly(500);
    }
}
```

在 startup\_task()创建任务:

```
err = OSTaskCreate(task1, (void *)0,
                   &task1_stk[TASK1_STK_SIZE-1], TASK1_PRIO);
err = OSTaskCreate(task2, (void *)0,
                   &task2_stk[TASK2_STK_SIZE-1], TASK2_PRIO);
```

把任务的堆栈大小和优先级写入 `app_cfg.h`，定义任务堆栈，编译调试。

在任务中打断点，用模拟器调试可以发现已经可以做任务切换了。如果有板子，烧到板子中运行，可以看到两个灯会以 **1Hz** 的频率闪烁。

可以认为移植初步成功，内核其他功能有待在应用中继续验证。

如何基于移植好的 `ucosii` 开发应用程序呢？

开发应用程序大部分都是为了处理或控制一个真实的物理系统，而真实的物理系统往往都是模拟系统，为了方便计算机处理，首先需要对系统做离散化处理。针对 `ucosii`，离散化过程是通过系统“心跳”（**SysTick**）来实现的。一般应用程序都有多个任务（不多任务谁用 `ucosii` 啊），任务可以分为周期任务和非周期任务。周期任务是周期性循环地处理事情的任务，而非周期任务一般是某个条件触发才执行的任务。这里有一个问题，**SysTick** 的时间是多少合适。**SysTick** 的时间一般取周期性任务中周期最短的时间值。譬如说，系统里有 3 个周期性任务：系统主任务（如处理 `pid` 等，任务周期 **4ms**），键盘扫描任务（任务周期 **16ms**），通信任务（任务周期 **128ms**），**SysTick** 时间就取 **4ms**。当然在 **SysTick** 时间较小时，要注意系统负荷问题，这时最好测一下 `cpu` 使用率及各个任务的时间等。

周期性任务的开发套路是怎么样的呢？看看定时器任务的做法就知道了，代码在 `os_tmr.c`。首先在 `OSTmr_Init()` 中初始化 `OSTmrSemSignal`，然后 `OSTmr_Task()` 任务会一直等待 `OSTmrSemSignal`，等到 `OSTmrSemSignal` 后去处理各个定时器。那么谁在释放 `OSTmrSemSignal` 呢？`OSTmrSignal()`，这个函数要求放在一定频率的时钟中断里，默认是在 **SysTick** 中断中（如果使能 `OS_TIME_TICK_HOOK_EN`）。好了，现在我们可以总结总结周期性任务的一般套路了。

首先在任务初始化函数中初始化一个信号量（一般会用信号量），伪代码如下：

```
void task_init(void)
{
    task_sem = OSSemCreate(0);
}
在任务中等待信号量
void task (void *p_arg)
{
    for (;;)
    {
        OSSemPend(task_sem, 0, &err);
```

```
        /* TODO: task handle here */  
    }  
}
```

周期性的释放信号量

`OSSemPost(task_sem);`

对于上面所说系统主任务，`OSSemPost(task_sem)`可以放在 `SysTick_Handler()` 中。所以一般来说 `OS_CPU_SysTickHandler()` 改动的可能性是非常大的。

非周期任务的开发套路又是怎样的呢？其实和周期性任务是差不多的，只是信号量不是周期性地释放，而是按需释放。

其他内核功能就不多介绍了，大家按需使用，不是很难。

本文代码：<http://download.csdn.net/source/3472653>

该移植代码在我自己开发的一个小玩意上已得到一段时间的验证，未发现问题。但由于水平所限，并不敢保证该移植是没有任何问题的，殷切希望大家批评指正。