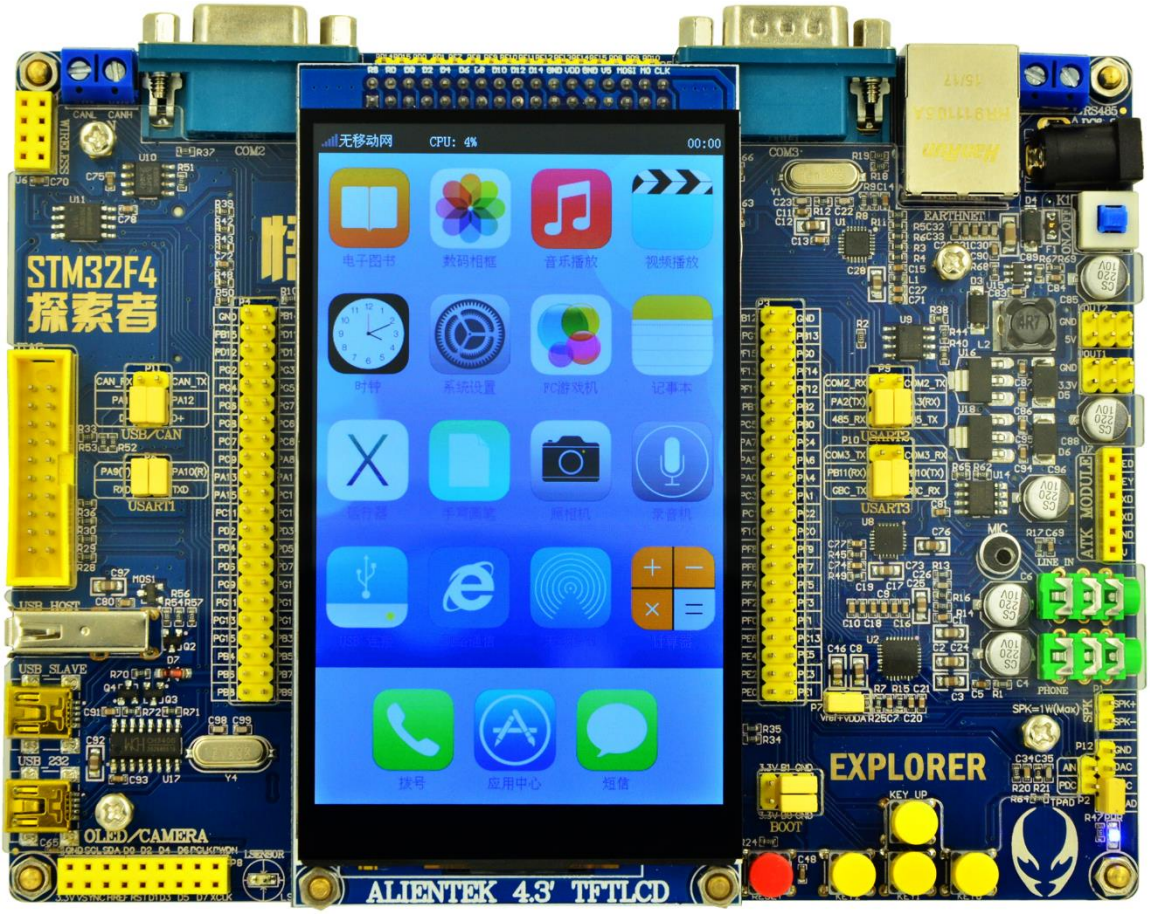
STM32F4 UCOS 开发手册 V3.0

–ALIENTEKSTM32F4 UCOSII/III 开发教程



广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛（开源电子网）：www.openedv.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com**QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友情提示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注

文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿： 第一章 UCOSII 移植 第二章 Cortex-M3/M4 基础 第三章 移植文件讲解	左忠凯	刘军	2014.11.4
V2.0	新增： 第四章 UCOSIII 移植 第五章 UCOSIII 任务管理 第六章 任务相关 API 函数使用 第七章 UCOSIII 中断和时间管理 第八章 UCOSIII 软件定时器 第九章 UCOSIII 信号量和互斥信号量 第十章 UCOSIII 消息传递 第十一章 事件标志组	左忠凯	刘军	2014.12.10
V2.1	修改： 1、修改了第四章中 led0_task 和 led1_task 两个任务中由于任务堆栈设置小而进入 hardfault 的 bug。 2、修改了第十章中关于 OSQPend()函数中参数 p_msg_size 的解释和例 10-1 的代码。	左忠凯	刘军	2015.2.27
V2.2	新增： 第十二章 存储管理	左忠凯	刘军	2015.4.24
V3.0	调整： 从第七章开始将后面的章节号做了调整。 新增： 第七章 UCOSIII 系统内部任务	左忠凯	刘军	2015.5.27

版本	版本更新说明	负责人	校审	发布日期
V3.0	新增： 第十章 UCOSIII 信号量和互斥信号量 第 10.5 节 优先级反转 第 10.6 节 优先级反转实验 第 10.7 节 互斥信号量 第 10.8 节 互斥信号量实验 第 10.9 节 任务内嵌信号量 第 10.10 节 任务内嵌信号量实验 第十一章 UCOSIII 消息传递 第 11.4 节 任务内建消息队列 第 11.5 节 任务内建消息队列实验 第十三章 同时等待多个内核对象	左忠凯	刘军	2015.5.27

目录

第一章 UCOSII 移植

在以前学习的例程中大多都不带操作系统，也就是裸奔，本教程将带领大家进入 RTOS 的世界，关于 RTOS 类操作系统有很多，本教程选取的是非常有名的 UCOS 操作系统。在使用 UCOS 之前我们要先完成 UCOS 在我们开发平台上的移植，本章我们将讲解如何在 ALIENTEK STM32F407 开发板上移植 UCOS II 操作系统，本章只是讲解如何移植，关于移植过程中使用到的文件我们会在下一章中进行详细讲解。

本章分为如下几个部分：

1.1 移植准备工作

1.2UCOS II 移植

1.3 软件设计

1.4 下载验证

1.1 移植准备工作

1) 准备基础工程

国际惯例，首先准备移植所需的基础工程，本章教程是在库函数版跑马灯实验的基础上完成的，基础工程就是跑马灯实验了。

2)UCOS II 源码

既然要移植 UCOS II，那么源码肯定是要有的，我们可以在 Micrium 官网上下载，下载地址：http://micrium.com/downloadcenter/download-results/?searchterm=mp-uc-os-ii&supported=true 下载界面如图 1.1.1 所示，在 Micrium 官网下载东西需要注册账号，我们已经下载好 UCOS II 源码放在光盘中，路径：6,软件资料→2，UCOS 学习资料→UCOSII 资料→UCOS II 源码下的 Micrium.rar 解压后就是 UCOSII 源码。

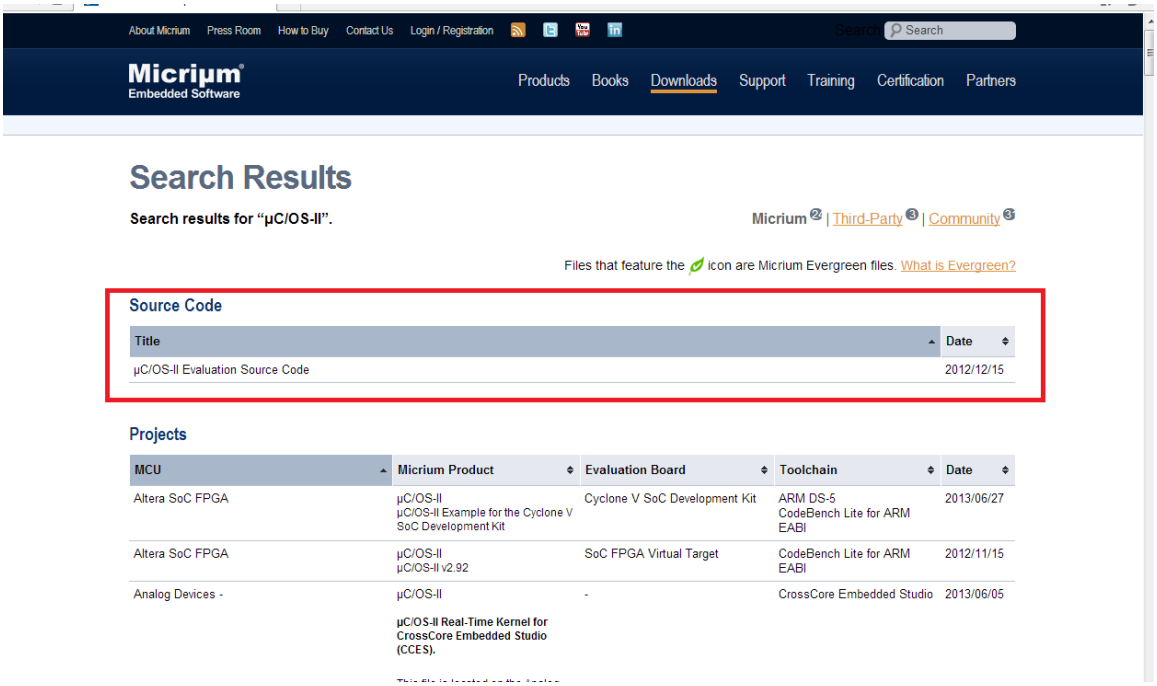


图 1.1.1 UCOS II 下载页面

有时候我们下载下来的可能是.html 结尾的文件，我们将其后缀改为.zip，然后使用解压软件解压就可以了，解压后的文件名为：Mircrium，这个就是 UCOS II 的源码文件。

我们按路径：Mircrium->Software->uCOS-II 打开文件，如图 1.1.2 所示。

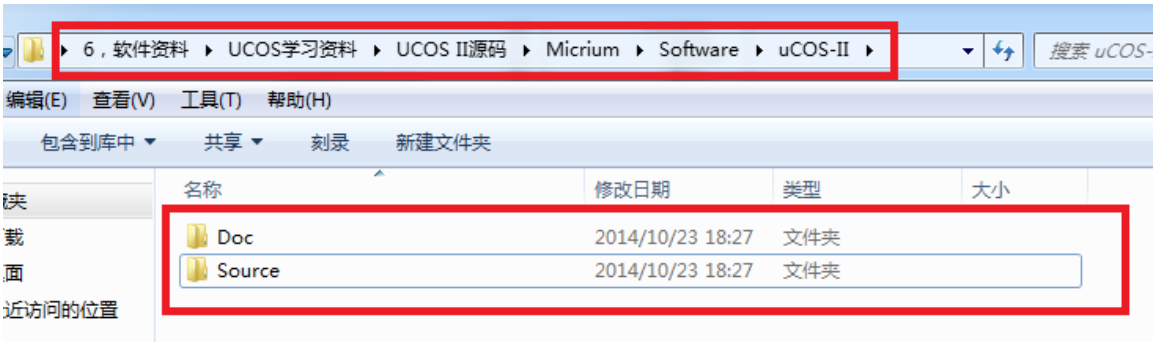


图 1.1.2 UCOS II 源码文件。

图 1.1.2 中 Doc 文件夹下面是一些关于 UCOS II 的文档，Source 文件夹就是 UCOS II 的源

码。

1.2 UCOS II 移植

1) 向工程中添加相应文件

1、建立相应文件夹

我们在工程目录下新建 UCOSII 文件夹，并在 UCOSII 文件夹另外中新建三个文件夹：CONFIG、CORE 和 PORT，如图 1.2.1 所示。

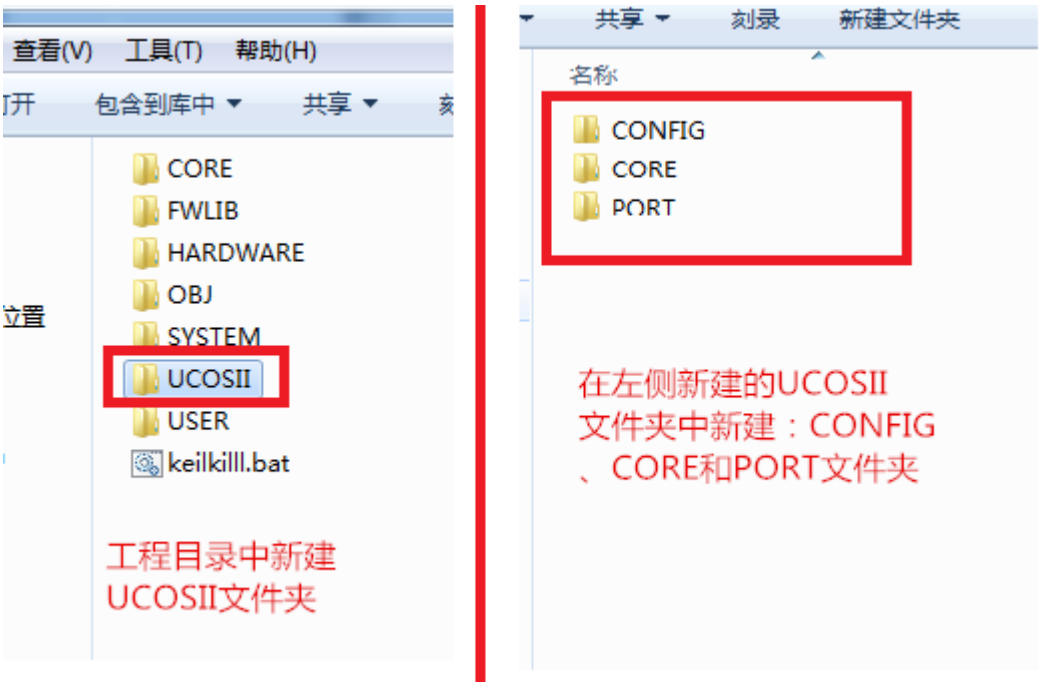


图 1.2.1 新建 UCOSII 文件夹及其内部文件夹

2、向 CORE 文件夹中添加文件

在 CORE 文件夹中我们添加 UCOSII 源码，我们打开 UCOSII 源码的 Source 文件夹，里面一共有 14 个文件，除了 os_cfg_r.h 和 os_dbg_r.c 这两个文件外我们将其他的文件都复制到我们工程中 UCOSII 文件夹中的 CORE 文件夹下，如图 1.2.2 所示。

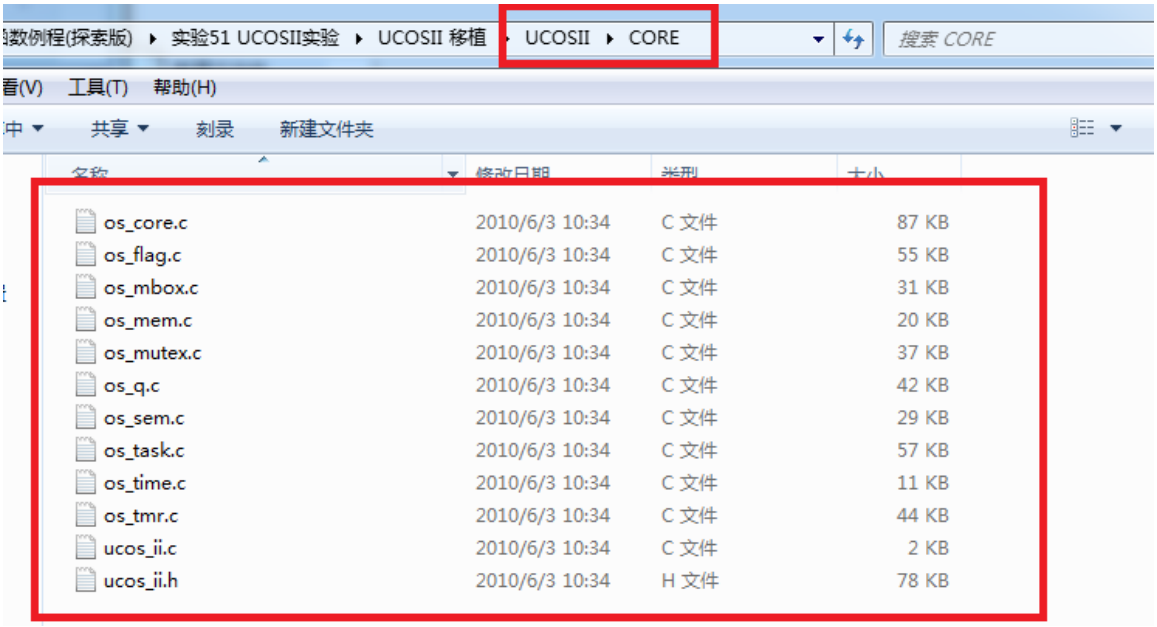


图 1.2.2 复制 UCOS 源码到 CORE 文件夹中

3、向 CONFIG 文件夹中添加文件

在 CONFIG 文件夹中有两个文件要添加：includes.h 和 os_cfg.h。这两个文件大家可以从本实验工程拷贝到自己的工程中，其中 includes.h 里面都是一些头文件，os_cfg.h 文件主要是用来配置和裁剪 UCOSII 的，将这两个文件拷贝到工程中如图 1.2.3 所示。

图 1.2.3 CONFIG 文件夹内容

4、向 PORT 文件夹中添加文件

我们需要向 PORT 文件夹中添加 5 个文件：os_cpu.h、os_cpu_a.asm、os_cpu_c.c、os_dbg.c 和 os_dgb_r.c。这五个文件可以从本实验的 PORT 文件夹中拷贝到自己的 PORT 文件夹中，拷贝完成后如图 1.2.4 所示。

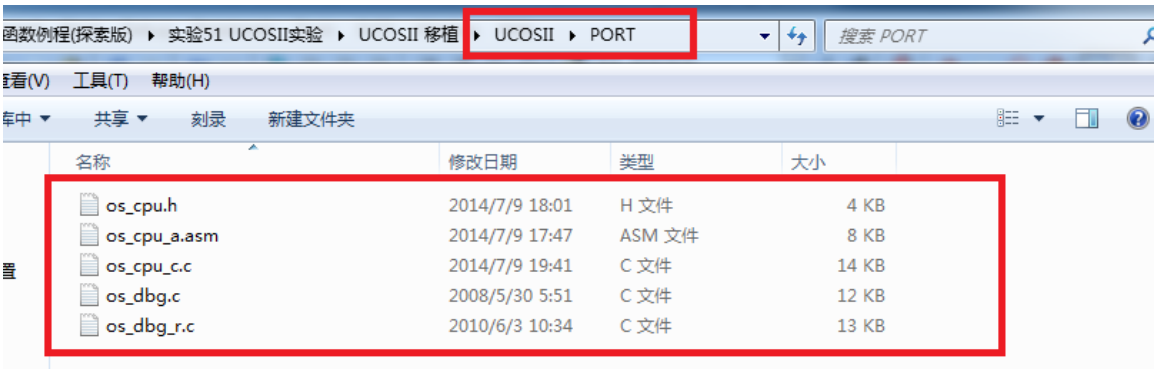


图 1.2.4 PORT 文件夹内容

4、将与 UCOSII 有关的文件添加到工程中

我们前面只是将所有的文件添加到工程目录的文件夹里面，还没有将这些文件真正的添加到工程中，我们在工程分组中建立三个分组：UCOSII-CORE、UCOSII-PORT 和 UCOSII-CONFIG。建立完成后如图 1.2.5 所示。

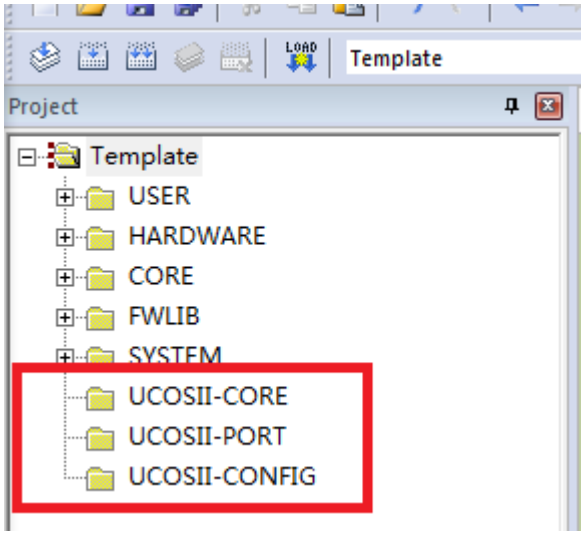


图 1.2.5 工程中添加文件

我们向 UCOSII-CORE 分组中添加 CORE 文件夹下除 ucosii.c 外的所有.c 文件，向 UCOSII-PORT 分组中添加 PORT 文件夹下的 os_cpu.h、os_cpu_a.asm 和 os_cpu_c.c 这三个文件，最后向 UCOSII-CONFIG 分组添加 CONFIG 文件夹下的 includes.h 和 os_cfg.h 这两个文件，添加完成后工程如图 1.2.6 所示。注意：不要将 ucos_ii.c 文件添加到 UCOSII-CORE 分组中!!! 否则编译以后会提示好多重复定义的错误！

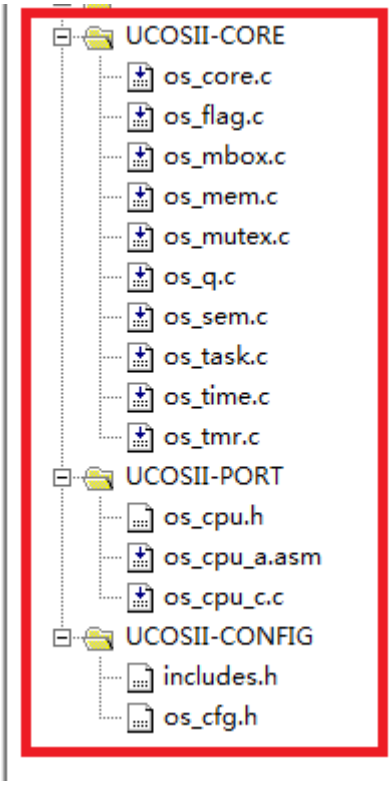


图 1.2.6 添加完成后的工程

最后添加相应的头文件路径，如图 1.2.7 所示。

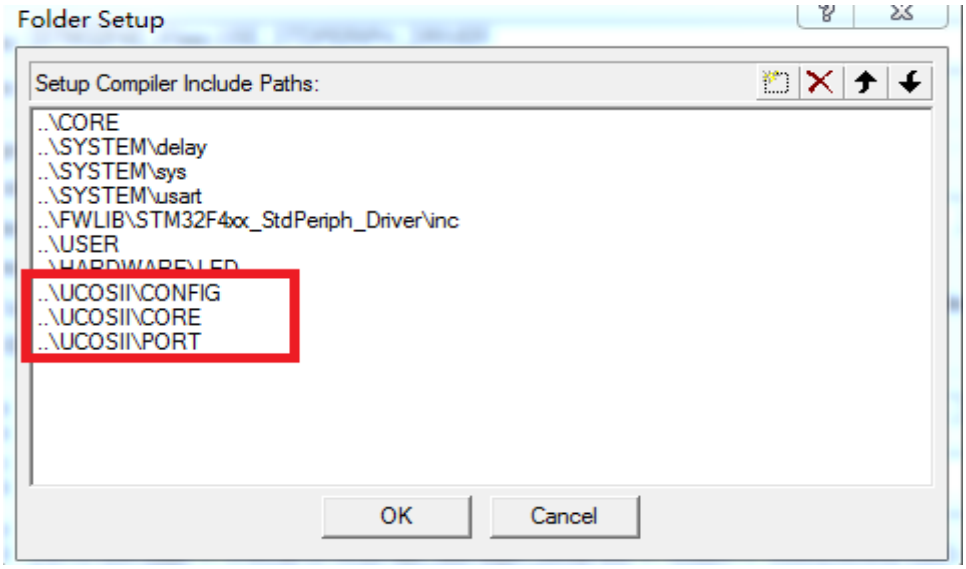


图 1.2.7 添加 UCOSII 相应的头文件

到这一步我们可以编译一下整个工程，结果提示很多错误，但是基本都是如图 1.2.8 所示的错误，提示我们不能打开“app_cfg.h”头文件。

```
.\UCOSII\CORE\ucos_ii.h(44): error:  #5: cannot open source input file "app_cfg.h": No such file or directory
 #include <app_cfg.h>
.\UCOSII\CORE\os_core.c: 0 warnings, 1 error
ompiling os_flag.c...
.\UCOSII\CORE\ucos_ii.h(44): error:  #5: cannot open source input file "app_cfg.h": No such file or directory
 #include <app_cfg.h>
.\UCOSII\CORE\os_flag.c: 0 warnings, 1 error
ompiling os_mbox.c...
.\UCOSII\CORE\ucos_ii.h(44): error:  #5: cannot open source input file "app_cfg.h": No such file or directory
 #include <app_cfg.h>
```

图 1.2.8 提示不能打开 app_cfg.h 头文件

我们可以追踪一下是在哪里出现的错误，结果发现在 ucos_ii.h 头文件中添加了 app_cfg.h 这个头文件，而这个头文件我们并没有实现，所以这里将这行代码屏蔽掉改为添加 includes.h 头文件，改完后如图 1.2.9 所示。

```
//#include <app_cfg.h> //未提供app_cfg.g头文件，屏蔽掉
#include "includes.h"  //添加includes.h头文件
#include <os_cfg.h>
#include <os_cpu.h>
```

图 1.2.9 修改头文件

修改完成后我们再编译一下整个工程，还是提示有错误，如图 1.2.10 所示，提示我们在 os_cpu_a.o 和 stm32f4xx_it.o 这两个文件中重复定义了 PendSV_Handler 这个函数。中断服务函数 PendSV_Handler 我们下一章讲解。

```
..\OBJ\Template.axf: Error: L6200E: Symbol PendSV_Handler multiply defined (by os_cpu_a.o and stm32f4xx_it.o).
Not enough information to list image symbols.
Not enough information to list the image map.
Finished: 2 information, 0 warning and 1 error messages.
"..\OBJ\Template.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

图 1.2.10 PendSV_Handler 错误

这里我们打开 stm32f4xx_it.c 文件，将中断服务函数 PendSV_Handler 屏蔽掉，屏蔽后如图 1.2.12 所示，我们再编译一下工程发现没有了错误，但是我们的移植工程还没有成功。

```
132   */
133  //void PendSV_Handler(void)
134  //{
135  //}
```

图 1.2.11 屏蔽掉 stm32f4xx_it.c 中的 PendSV 中断服务函数

5、修改 sys.h 头文件

打开我们的 sys.h 头文件，里面有一个 SYSTEM_SUPPORT_OS 的宏定义，如果定义为 0 的话不支持 OS，我们将其改为 1 支持 OS。

将 SYSTEM_SUPPORT_UCOS 定义为 1 后我们编译一下工程，发现提示如图 1.2.12 所示错误，提示我们在 stm32f4xx_it.o 和 delay.o 这两个文件中重复定义了 SysTick_Handler 这个函数，中断服务函数 SysTick_Handler 我们在下一章讲解。

```
..\OBJ\Template.axf: Error: L6200E: Symbol SysTick_Handler multiply defined (by delay.o and stm32f4xx_it.o).
Not enough information to list image symbols.
Finished: 2 information, 0 warning and 1 error messages.
"..\OBJ\Template.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

图 1.2.12 SysTick_Handler 重复定义

同样，我们将 stm32f4xx_it.c 文件中的中断服务函数 SysTick_Handler 屏蔽掉，屏蔽后如图 1.2.13 所示。

```
142  //void SysTick_Handler(void)
143  //{
144  //
145  //}
146
```

图 1.2.13 屏蔽掉中断服务函数 SysTick_Handler

屏蔽掉 SysTick_Handler 中断服务函数后我们再编译一下工程，发现没有错误，如果还有错误的话请自行根据错误类型修改。

2) 开启 FPU

因为 STM32F407 有浮点运算单元 FPU，那么我们移植完 UCOS II 以后就要测试一下是否支持浮点运算，因此要开启 FPU，开启 FPU 的过程如下。

1、打开 system_stm32f4xx.c 文件，里面有一个 SystemInit()函数，这个函数完成系统初始化,在一开始就有 FPU 设置选项，如图 1.2.14 所示，

```
417  void SystemInit(void)
418  {
419    /* FPU settings ------------------------------------------------------------*/
420    #if (__FPU_PRESENT == 1) && (__FPU_USED == 1)
421      SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access */
422    #endif
423    /* Reset the RCC clock configuration to the default reset state ------------*/
424    /* Set HSION bit */
425    RCC->CR |= (uint32_t)0x00000001;
426
427    /* Reset CFGR register */
428    RCC->CFGR = 0x00000000;
429
```

图 1.2.14 设置 FPU

从图 1.2.14 中我们可以看出如果要使用 FPU 的话__FPU_PRESENT 和__FPU_USED 要为

1，在 stm32f4xx.h 文件中已经定义了__FPU_PRESENT 为 1，但是并未定义__FPU_USED，因此我们按图 1.2.15 所示添加__FPU_USED 的定义。

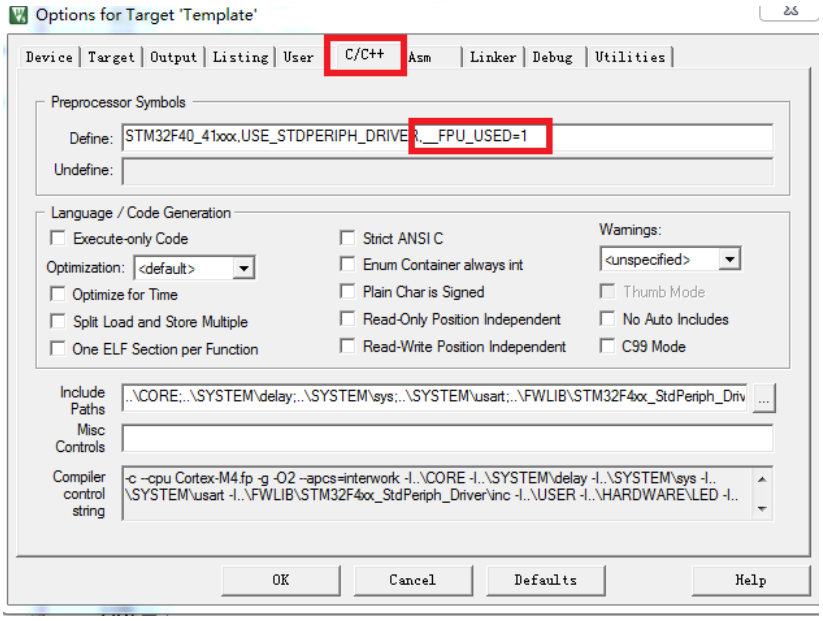


图 1.2.15 定义__FPU_USED=1

最后我们设置 keil 软件是用 FPU，如图 1.2.16 所示。

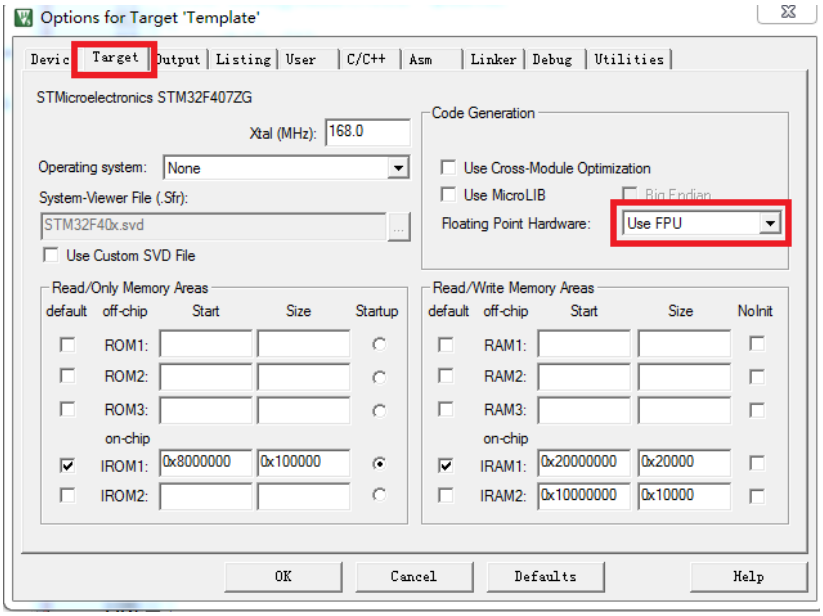


图 1.2.16 选择使用 FPU

1.3 软件设计

通过上面几节我们已经完成了 UCOS II 在 STM32F407 的移植。本节我们就编写测试程序，检验一下我们的移植是否成功，我们建立 3 个简单的任务来测试一下，这里我们还另外建立了一个 start_task 任务用来创建其他 3 个任务，main.c 里面的代码如下，完整工程详见“例 1-1 UCOSII 移植”

```
//START 任务
//设置任务优先级
```

```
#define START_TASK_PRIO             10   ///开始任务的优先级为最低
//设置任务堆栈大小
#define START_STK_SIZE              128
//任务任务堆栈
OS_STK START_TASK_STK[START_STK_SIZE];
//任务函数
void start_task(void *pdata);

//LED0 任务
//设置任务优先级
#define LED0_TASK_PRIO              7
//设置任务堆栈大小
#define LED0_STK_SIZE               64
//任务堆栈
OS_STK LED0_TASK_STK[LED0_STK_SIZE];
//任务函数
void led0_task(void *pdata);

//LED1 任务
//设置任务优先级
#define LED1_TASK_PRIO              6
//设置任务堆栈大小
#define LED1_STK_SIZE               64
//任务堆栈
OS_STK LED1_TASK_STK[LED1_STK_SIZE];
//任务函数
void led1_task(void *pdata);

//浮点测试任务
#define FLOAT_TASK_PRIO             5
//设置任务堆栈大小
#define FLOAT_STK_SIZE              128
//任务堆栈
//如果任务中使用 printf 来打印浮点数据的话一点要 8 字节对齐
__align(8) OS_STK FLOAT_TASK_STK[FLOAT_STK_SIZE];
//任务函数
void float_task(void *pdata);

int main(void)
{
    delay_init(168);            //延时初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //中断分组配置
    uart_init(115200);      //串口波特率设置
```

```
    LED_Init();    //LED 初始化

    OSInit();    //UCOS 初始化
    OSTaskCreate(start_task,(void*)0,(OS_STK*)&START_TASK_STK[START_STK_SIZE-1],\
    ST ART_TASK_PRIO); //创建开始任务
    OSStart(); //开始任务
}

//开始任务
void start_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    pdata=pdata;
    OSStatInit();    //开启统计任务

    OS_ENTER_CRITICAL();    //进入临界区(关闭中断)
    //创建 LED0 任务
    OSTaskCreate(led0_task,(void*)0,(OS_STK*)&\
        LED0_TASK_STK[LED0_STK_SIZE-1],LED0_TASK_PRIO);
    //创建 LED1 任务
    OSTaskCreate(led1_task,(void*)0,(OS_STK*)&\
        LED1_TASK_STK[LED1_STK_SIZE-1],LED1_TASK_PRIO);
    //创建浮点测试任务
    OSTaskCreate(float_task,(void*)0,(OS_STK*)&\
        FLOAT_TASK_STK[FLOAT_STK_SIZE-1],FLOAT_TASK_PRIO);
    OSTaskSuspend(START_TASK_PRIO);//挂起开始任务
    OS_EXIT_CRITICAL();    //退出临界区(开中断)
}

//LED0 任务
void led0_task(void *pdata)
{
    while(1)
    {
        LED0=0;
        delay_ms(80);
        LED0=1;
        delay_ms(400);
    };
}

//LED1 任务
void led1_task(void *pdata)
```

```
{
    while(1)
    {
        LED1=0;
        delay_ms(300);
        LED1=1;
        delay_ms(300);
    };
}

//浮点测试任务
void float_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    staticfloat float_num=0.01;
    while(1)
    {
        float_num+=0.01f;
        OS_ENTER_CRITICAL();  //进入临界区(关闭中断)
        printf("float_num 的值为: %.4f\r\n",float_num); //串口打印结果
        OS_EXIT_CRITICAL();     //退出临界区(开中断)
        delay_ms(500);
    }
}
```

从 main.c 中我们可以看出一共有 4 个任务：start_task、led0_task、led1_task 和 float_task。start_task 是用于创建其他 3 个任务的，当创建完其他 3 个任务后就会挂起的。led0_task 和 led1_task 这两个任务分别是让 LED0，LED1 闪烁的，这两个任务都很简单。最后一个 float_task 任务是用来测试 FPU 能不能正常使用，我们每 500ms 给 float_num 加一个 0.01，然后通过串口打印出来。

1.4 下载验证

编译代码后下载到开发板中，打开串口调试助手，我们可以看到 LED0 和 LED1 开始按照我们设置好的时间间隔闪烁。串口调试助手有信息输出，如图 1.4.1 所示，从图中可以看出 float_num 的值以 0.01 递增，和我们程序中设置的一样。

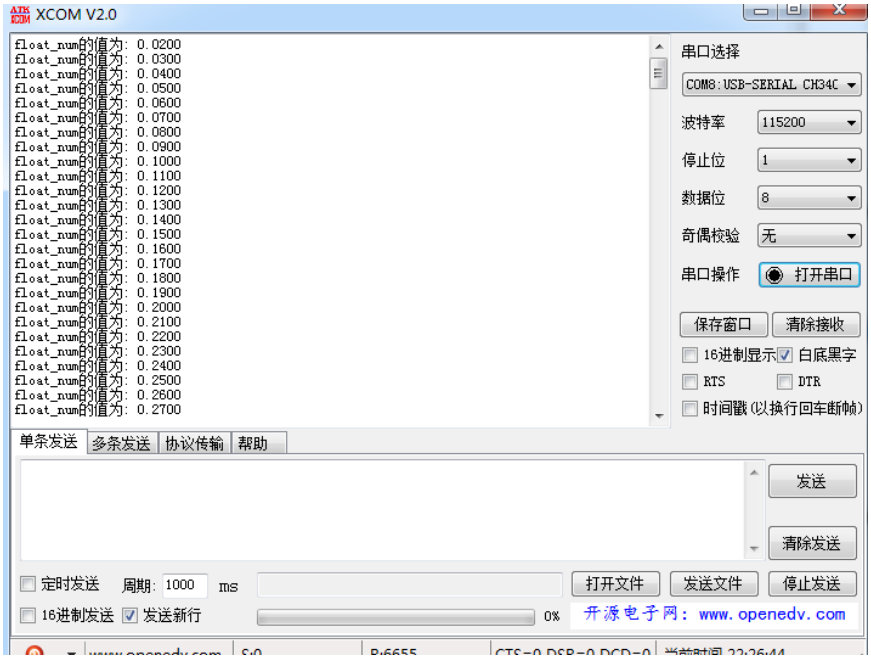


图 1.4.1 串口输出

通过上面的测试我们还不能确定是否是使用了内部 FPU，如果使用了 STM32F407 的内部 FPU 那么关于浮点的计算肯定会使用到浮点指令的，我们可以通过硬件调试来查看是否使用了 FPU 指令，我们进入调试界面，打开汇编窗口：View->Disassembly。在如图 1.4.2 所示的地方设置断点。

```
104 //浮点测试任务
105 void float_task(void *pdata)
106 {
107     OS_CPU_SR cpu_sr=0;
108     float float_num=0.01;
109     while(1)
110     {
111         float_num+=0.01f;
112         OS_ENTER_CRITICAL();     //开中断
113         printf("float_num的值为: %.4f\r\n",float_num); //串口打印结果
114         OS_EXIT_CRITICAL();      //关中断
115         delay_ms(500);
116     }
117 }
118
```

图 1.4.2 设置断点

然后我们点击全速运行，程序运行到上面设置的断点处停止，这时我们查看汇编窗口，如图 1.4.3 所示。

```
0x0800024E E01A      B            0x08000286
   111:                 float_num+=0.01f;
0x08000250 ED9F0A3E  VLDR         s0,[pc,#0xF8]
0x08000254 EE380A00  VADD.F32     s0,s16,s0
0x08000258 EEB08A40  VMOV.F32     s16,s0
```

图 1.4.3 汇编窗口

从图 1.4.3 中我们可以看出有 VLDR、VADD.F32 等指令，根据《Cortex-M4 Devices Generic User Guide》(光盘资料中：8，STM32 参考资料→Cortex-M4 Devices Generic User Guide.pdf)的 173 页我们可以知道 VLDR、VADD.F32 等指令是 FPU 指令，说明我们使用 STM32F407 内部 FPU 成功。

第二章 Cortex-M3/M4 基础

上一章中我们讲解了如何在 STM32F407 开发板上移植 UCOSII 操作系统，我们在移植操作系统的时候一定要对处理器的架构有一定的了解，本章就讲解一下 Cortex-M3/M4 的基础知识，了解了处理器的基础知识以后就能够看懂移植过程的一些重要文件，因为这些文件都是和处理器密切相关的，本章分为如下几个部分：

2.1 Cortex-M3/M4 通用寄存器
2.2 操作模式和特权级别
2.3 FPU 单元
2.4 堆栈
2.5 SVC 和 PendSV 异常

2.1Cortex-M3/M4 通用寄存器

我们首先了解一下 M3/M4 的寄存器，M4 相对于 M3 多了一个浮点单元 FPU，其他的基本和 M3 是一样的，以下内容参考自《ARM Cortex-M3 权威指南》和《Cortex-M3 与 M4 权威指南》，这两本资料的电子档在光盘下的：8，STM32 参考资料。

如我们所见， Cortex-M3/M4 系列处理器拥有通用寄存器 R0 - R15 以及一些特殊功能寄存器。 R0 - R12 是最“通用目的”的，但是绝大多数的 16 位指令只能使用 R0 - R7（低组寄存器），而 32 位的 Thumb - 2 指令则可以访问所有通用寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问，Cortex-M3/M4 的通用寄存器如图 2.1.1 所示。

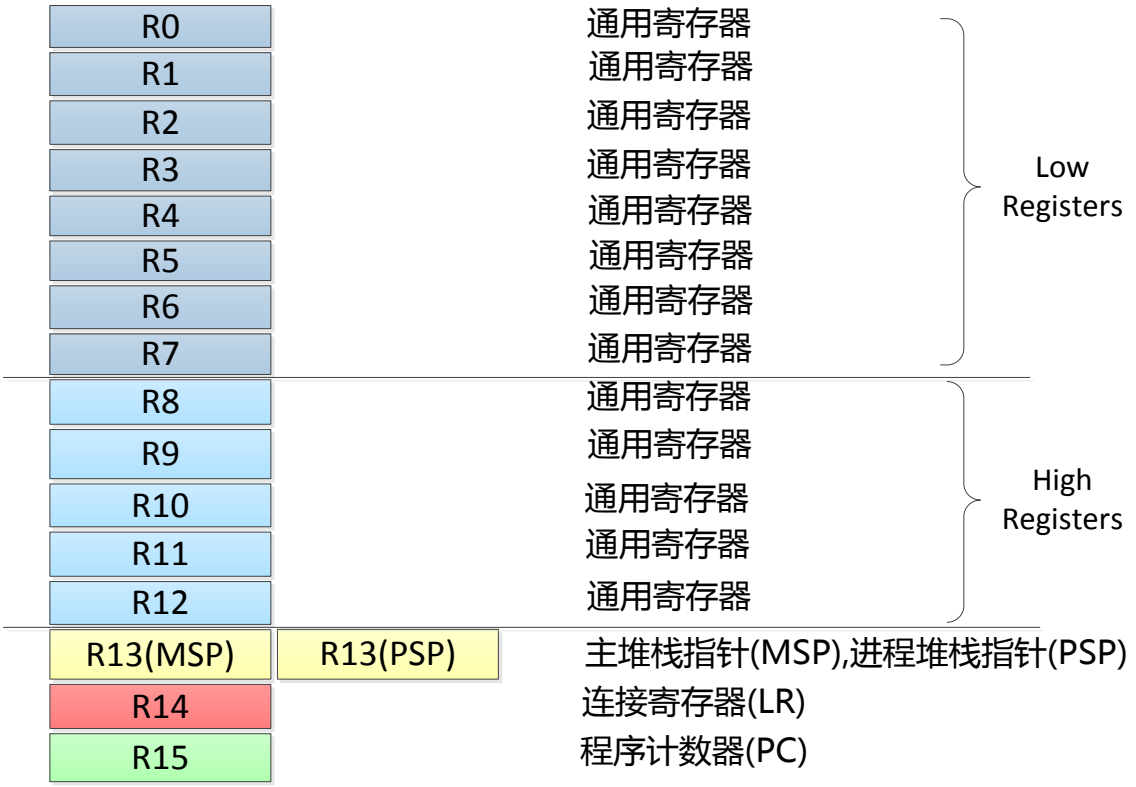


图 2.1.1 Cortex-M3/M4 通用寄存器

1)通用目的寄存器 R0-R7

R0 - R7 也被称为低组寄存器。所有指令都能访问它们。它们的字长全是 32 位，复位后的初始值是不可预料的。

2)通用目的寄存器 R8-R12

R8 - R12 也被称为高组寄存器。这是因为只有很少的 16 位 Thumb 指令能访问它们，32
位的指令则不受限制。它们也是 32 位字长，且复位后的初始值是不可预料的。

3)堆栈指针 R13

R13 是堆栈指针。在 CM3/CM4 处理器内核中共有两个堆栈指针，于是也就支持两个堆栈。当引用 R13（或写作 SP）时，你引用到的是当前正在使用的那一个，另一个必须用特殊的指令来访问（ MRS,MSR 指令）。这两个堆栈指针分别是：

- **主堆栈指针（MSP）**，或写作 SP_main。这是缺省的堆栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。
- **进程堆栈指针（PSP）**，或写作 SP_process。用于常规的应用程序代码（不处于异常服用例程中时）。要注意的是，并不是每个应用都必须用齐两个堆栈指针。简单的应用程序只

使用 MSP 就够了。堆栈指针用于访问堆栈，并且 PUSH 指令和 POP 指令默认使用 SP。

4)连接寄存器 R14

R14 是连接寄存器（LR）。在一个汇编程序中，你可以把它写作 both LR 和 R14。LR 用于在调用子程序时存储返回地址。例如，当你在使用 BL(分支并连接， Branch and Link)指令时，就自动填充 LR 的值。

尽管 PC 的 LSB 总是 0（因为代码至少是字对齐的）， LR 的 LSB 却是可读可写的。这是历史遗留的产物。在以前，由位 0 来指示 ARM/Thumb 状态。因为其它有些 ARM 处理器支持 ARM 和 Thumb 状态并存，为了方便汇编程序移植， CM3/CM4 需要允许 LSB 可读可写。

5)程序计数器 R15

R15 是程序计数器，在汇编代码中你也可以使用名字“PC”来访问它。因为 CM3/CM4 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。比如说：

```
0x1000:  MOV   R0,   PC          ; R0 = 0x1004
```

如果向 PC 中写数据，就会引起一次程序的分支（但是不更新 LR 寄存器）。 CM3/CM4 中的指令至少是半字对齐的，所以 PC 的 LSB 总是读回 0。然而，在分支时，无论是直接写 PC 的值还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即 LSB=1），用以表明这是在 Thumb 状态下执行。倘若写了 0，则视为企图转入 ARM 模式，CM3 将产生一个 fault 异
常。

6)特殊功能寄存器组

Cortex-M3/M4 有一个特殊功能寄存器组，如图 2.1.2 所示。

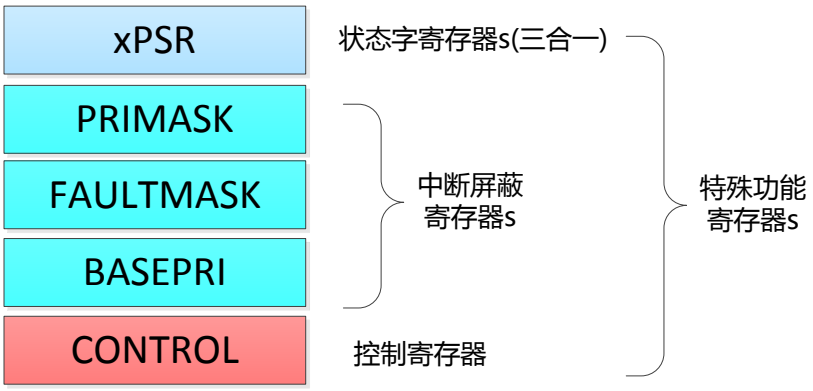


图 2.1.2 特殊功能寄存器组

Cortex‐M3/M4 中的特殊功能寄存器包括：

- 程序状态寄存器组（ PSRs 或曰 xPSR）
- 中断屏蔽寄存器组（ PRIMASK, FAULTMASK,以及 BASEPRI）
- 控制寄存器（ CONTROL）

它们只能被专用的 MSR 和 MRS 指令访问，而且它们也没有存储器地址。

```
MRS     <gp_reg>,     <special_reg>     ; 读特殊功能寄存器的值到通用寄存器
MSR     <special_reg>,  <gp_reg>        ; 写通用寄存器的值到特殊功能寄存器
```

7)程序状态寄存器(PSRs 或曰 PSR)

- 应用程序 PSR（ APSR）
- 中断号 PSR（ IPSR）
- 执行 PSR（ EP SR）

通过 MRS/MSR 指令，这 3 个 PSRs 既可以单独访问，也可以组合访问（2 个组合， 3 个组合都可以）。当使用三合一的方式访问时，应使用名字“ xPSR”或者“ PSR”。这三个寄存器的各个位的含意如表 2.1.1 所示。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

表 2.1.1 Cortex-M3/M4 程序状态寄存器(xPSR)

PRIMASK,FAULTMASK 和 BASEPRI

这三个寄存器很重要，这三个寄存器用于控制异常的使能和除能，这三个寄存器的介绍如表 2.1.2 所示

名字	功能描述
PRIMASK	这个寄存器只有一个位，当这个位置 1 时 就关掉所有可屏蔽的异常，只剩下 NMI 和硬 fault 可以响应。默认值为 0，表示没有关中断。
FAULTMASK	这个寄存器也只有一个位。当它置 1 时，只有 NMI 才能响应，所有其它的异常，包括中断和 fault，通通闭嘴。默认值为 0，表示没有关异常
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断， 0 也是缺省值。

表 2.1.2 Cortex-M3/M4 屏蔽寄存器

对于时间-关键任务而言， PRIMASK 和 BASEPRI 对于暂时关闭中断是非常重要的。而 FAULTMASK 则可以被 OS 用于暂时关闭 fault 处理机能，这种处理在某个任务崩溃时可能需要。因为在任务崩溃时，常常伴随着一大堆 faults。在系统料理“后事”时，通常不再需要响应这些 fault——人死帐清。总之 FAULTMASK 就是专门留给 OS 用的。

要访问 PRIMASK, FAULTMASK 以及 BASEPRI，同样要使用 MRS/MSR 指令,如：

```
MRS  R0,          BASEPRI       ; 读取 BASEPRI 到 R0 中
MRS  R0,          FAULTMASK     ; 同上
MRS  R0,          PRIMASK       ; 同上
MSR  BASEPRI,         R0            ; 写入 R0 到 BASEPRI 中
MSR  FAULTMASK,   R0            ; 同上
MSR  PRIMASK,         R0            ; 同上
```

只有在特权级下，才允许访问这 3 个寄存器，为了快速地开关中断，CM3/CM4 还专门设置了一条 CPS 指令，有 4 种用法，这四种用法非常重要，我们在移植 UCOS 操作系统的时候就是用这下面的方法来开关中断的。

```
CPSID    I    ;PRIMASK=1,   ; 关中断
CPSIE    I    ;PRIMASK=0,   ; 开中断
CPSID    F    ;FAULTMASK=1, ; 关异常
CPSIE    F    ;FAULTMASK=0  ; 开异常
```

8)控制寄存器（CONTROL）

CONTROL 寄存器用于定义特权级别和堆栈指针的使用，CONTROL 寄存器如表 2.1.3 所示,注意 CONTROL[2]只有 Cortex-M4 才有。

位	功能
CONTROL[31：3]	保留
CONTROL[2]	0=未使用浮点单元 1=使用浮点单元 Cortex-M4 使用这个位来决定当处理异常时是否保存浮点环境
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后缺省值） 1=选择进程堆栈指针 PSP 在线程模式下可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 Handler 模式永远都是特权级的

表 2.1.3 CONTROL 寄存器

CONTROL[2]

在 Cortex-M4 中有 FPU 单元，如果我们使用了 FPU，那么在处理异常时就需要保存 FPU 环境，此位用来指示是否需要保存浮点环境。

CONTROL[1]

在 Cortex-M3 的 handler 模式中， CONTROL[1]总是 0。在线程模式中则可以为 0 或 1。

仅当处于特权级的线程模式下，此位才可写，其它场合下禁止写此位。改变处理器的模式也有其它的方式：在异常返回时，通过修改 LR 的位 2，也能实现模式切换。

CONTROL[0]

仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位。CONTROL 寄存器也是通过 MRS 和 MSR 指令来操作的：

```
MRS      R0,          CONTROL
MSR      CONTROL,     R0
```

9)EXC_RETURN

在进入异常服务程序后， L R 的值被自动更新为特殊的 EXC_RETURN，对于 Cortex-M4 来说这是一个高 27 位全为 1 的值(M3 是高 28 位都为 1)。M4 中[4:0]位有意义，在 M3 中[3:0]有意义，并且和 M4 中的相同，EXC_RETURN 位段如表 2.1.4 所示。

注意！EXC_RETURN 的 bit4 非常重要，我们可以根据此位的值来获知硬件会对哪些寄存器进行自动压入栈和出栈处理，这个我们会在讲浮点寄存器和堆栈的时候详细讲解。

位段	描述
[31:28]	EXC_RETURN 的标识，为 0XF。
27:5	保留，全为 1。
4	堆栈类型(硬件自动压栈大小)，一个字为 32bit 0=硬件自动向堆栈中压入 26 个字 1=硬件自动向堆栈中压入 8 个字 当未使用 FPU 的时候此位总为 1。当进入异常服务程序的时候此位会被设置为 CONTROL 寄存器中 FPCA 位(bit4)的反相。例如,如果 FPCA 为 1，则此位为 0。
3	0=返回进入 Handler 模式 1=返回后进入线程模式
2	0=从主堆栈中做出栈操作，返回后使用 MSP， 1=从进程堆栈中做出栈操作，返回后使用 PSP
1	保留，必须为 0
0	0=返回 ARM 状态。 1=返回 Thumb 状态。在 CM3/CM4 中必须为 1

表 2.1.4 EXC_RETURN 位段详解

从表 2.1.4 中我们可得到 EXC_RETURN 的最终返回值共有 6 个，如表 2.1.5 所示。

使用 FPU 时的值	未使用 FPU 时的值	功能
0xFFFFFFE1	0xFFFFFFF1	返回 handler 模式
0xFFFFFFE9	0xFFFFFFF9	返回线程模式，并使用主堆栈(SP=MSP)
0xFFFFFFED	0xFFFFFFFD	返回线程模式，并使用线程堆栈(SP=PSP)

表 2.1.5 合法的 EXC_RETURN 值及其功能

2.2 操作模式和特权级别

Cortex-M3/CM4 处理器支持两种处理器的操作模式，还支持两级特权操作。两种操作模式分别为：处理者模式 (handler mode)和线程模式(thread mode)。引入两个模式的本意，是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。Cortex-M3/M4 的另一个侧面则是特权的分级——特权级和用户级。这可以提供一种存储器访问的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，如表 2.2.1 所示，这也是一个基本的安全模型。

	特权级	用户级
异常 handler 的代码	handler 模式	错误的用法
主应用程序的代码	线程模式	线程模式

表 2.2.1 Cortex-M3/M4 下的操作模式和特权级别

在 CM3/CM4 运行主应用程序时（线程模式），既可以使用特权级，也可以使用用户级；但是异常服务例程必须在特权级下执行。复位后，处理器默认进入线程模式，特权级访问。在特权级下，程序可以访问所有范围的存储器（如果有 MPU，还要在 MPU 规定的禁地之外），并且可以执行所有指令。

在特权级下的程序可以为所欲为，但也可能会把自己给玩进去——切换到用户级。一旦进入用户级，再想回来就得走“法律程序”了——用户级的程序不能简简单单地试图改写 CONTROL 寄存器就回到特权级，事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级，并且在异常服务例程执行完毕退出时，返回先前的状态。

2.3 FPU 单元

在 Coretex-M4 处理器中有一个可选的单精度 FPU 单元，我们开发板选用的 STM32F407 就有 FPU 单元，如果使能了 FPU 单元的话就可以使用它来对单精度浮点数进行计算，双精度浮点数的计算仍然要使用到 C 运行库。

2.3.1 FPU 寄存器

FPU 单元包含一系列的寄存器：

- CPACR 寄存器，在 SCB 块中
- 浮点寄存器块，S0-S31
- FPU 状态和控制寄存器：FPSCR
- 其他的一些 FPU 寄存器

1)CPACR 寄存器

通过 CPACR 寄存器来使能 FPU，CPACR 寄存器的地址为 0XE000ED88，我们也可以通过“SCB->CPACR”来访问 CPACR 寄存器，CPACR 寄存器的 bit0-bit19 和 bit24-bit31 不允许使用，为保留位，其中[20:21]为 CP10，[22:23]为 CP11。我们通过设置 CP10 和 CP11 来开启 FPU，CP10 和 CP11 设置情况如表 2.2.1 所示，注意 CP10 和 CP11 都为 2bit

Bits	设置情况
00	功能禁止，任何尝试操作将会导致用法错误(Usage fault)
01	仅特权级可用，非特权级操作将会导致用法错误(Usage fault)
10	保留
11	任何等级都可以使用

表 2.3.1 CP10 和 CP11 设置表

默认情况下 CP10 和 CP11 都为 00，如果要使用 FPU 的话需要软件设置 CPACR 来开启 FPU，通过设置 CP10 和 CP11 都为 11 来开启，实例代码如下：

```
SCB->CPACR|=0X00F00000;          //使能 FPU
```

2) 浮点寄存器块

浮点寄存器快包含 32 个 32 位的寄存器，这 32 个寄存器可以两两组合成一个 64 位的双精度寄存器，如图 2.3.1 所示。

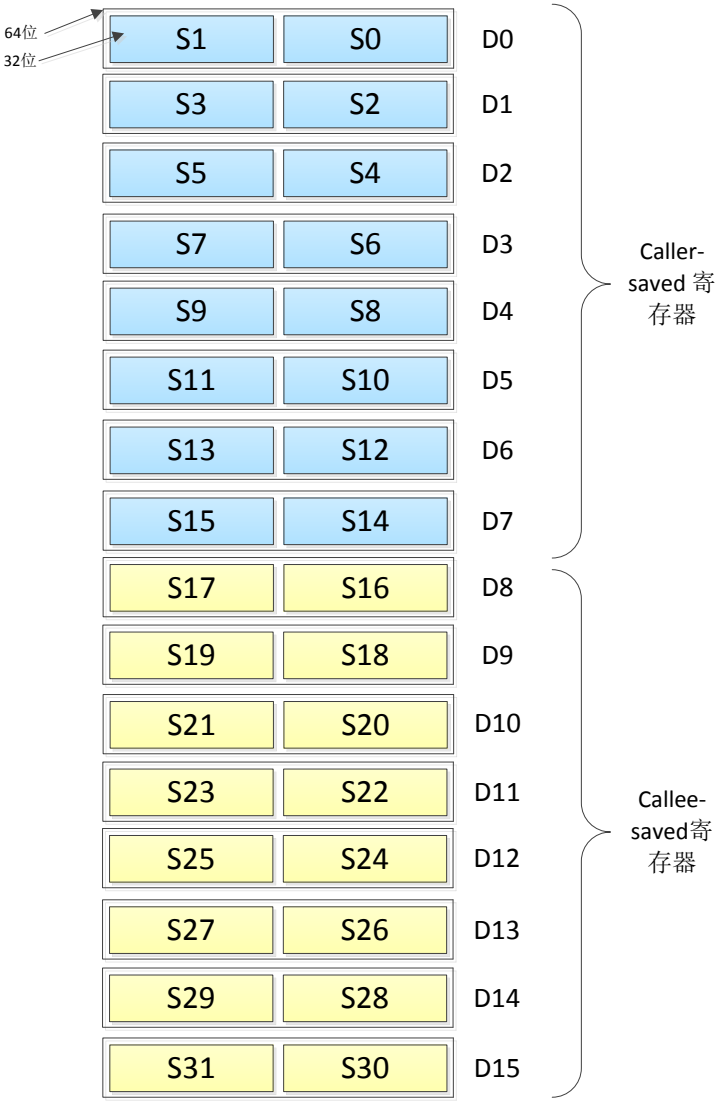


图 2.3.1 浮点寄存块

S0-S15 是 caller-saved 寄存器，如果一个应用 A 调用了另外一个应用 B，那么应用 A 在调用 B 之前一定要保存这些寄存器，因为在调用的时候这些寄存器会被改变。

S16-S31 被称为 callee-saved 寄存器，如果一个应用 A 调用应用 B，而且 B 需要大于 16 个寄存器来做计算，那么应用 B 就需要保存这些寄存器。并且在返回应用 A 的时候必须恢复这些寄存器。

2.3.2 Lazy Stacking

对于 Cortex-M4 来说 Lazy Stacking 是一个重要的特性，在使用 FPU 的情况下，不使用这个特性会在异常处理的时候消耗 29 个时钟周期，因为要将 25 个寄存器压栈，以前只需要将 8 个压栈。如果使用 Lazy Stacking 这个特性的话，那么在异常处理的时候只需要消耗 12 个时钟周期，默认情况下 Lazy Stacking 是使能的

可以看出如果在任务切换中使用 Cortex-M4 的这个特性将会极大的提高任务切换的速度，我们在移植 UCOSII 的时候就使用了这个特性。

2.4 堆栈

2.4.1 Cortex-M3/M4 堆栈操作

Cortex-M3/M4 使用的是“向下生长的满栈”模型。堆栈指针 SP 指向最后一个被压入堆栈的 32 位数值。在下一次压栈时，SP 先自减 4，再存入新的数值，如图 2.4.1 所示。

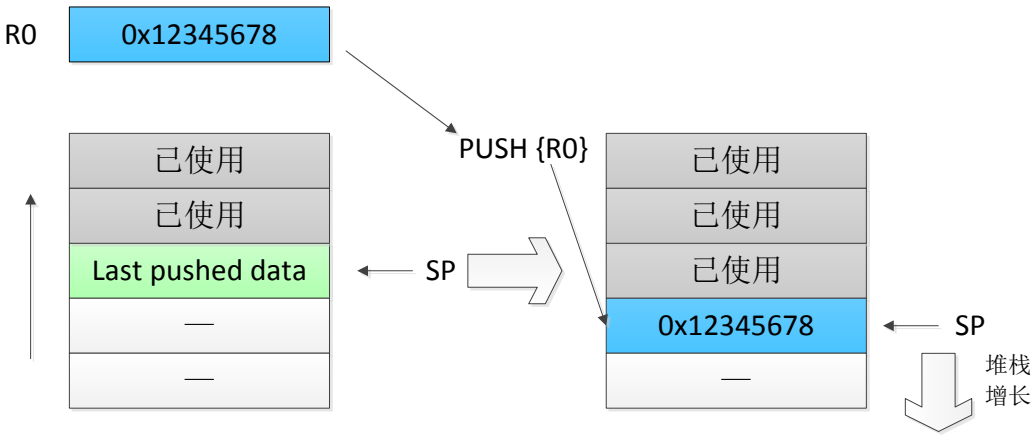


图 2.4.1 堆栈的 PUSH 操作

POP 操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4。如图 2.4.2 所示。

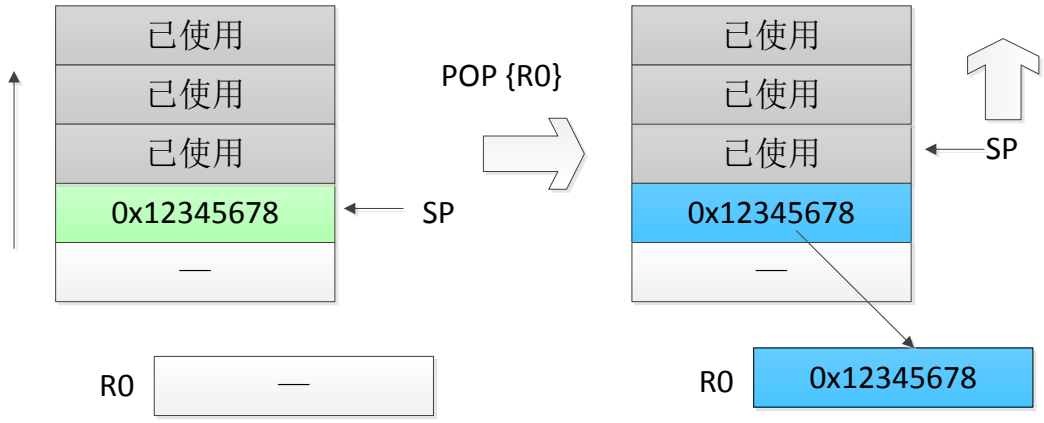


图 2.4.2 堆栈的 POP 操作

在进入 ISR 时，CM3/CM4 会自动把一些寄存器压栈，这里使用的是进入 ISR 之前使用的 SP 指针（MSP 或者是 PSP）。离开 ISR 后，只要 ISR 没有更改过 CONTROL[1],就依然使用先前的 SP 指针来执行出栈操作。

2.4.2 双堆栈机制

我们已经知道了 CM3/CM4 的堆栈是分为两个：主堆栈和进程堆栈，CONTROL[1]决定如何选择。当 CONTROL[1]=0 时，只使用 MSP，此时用户程序和异常 handler 共享同一个堆栈。这也是复位后的缺省使用方式，如图 2.4.3 所示。

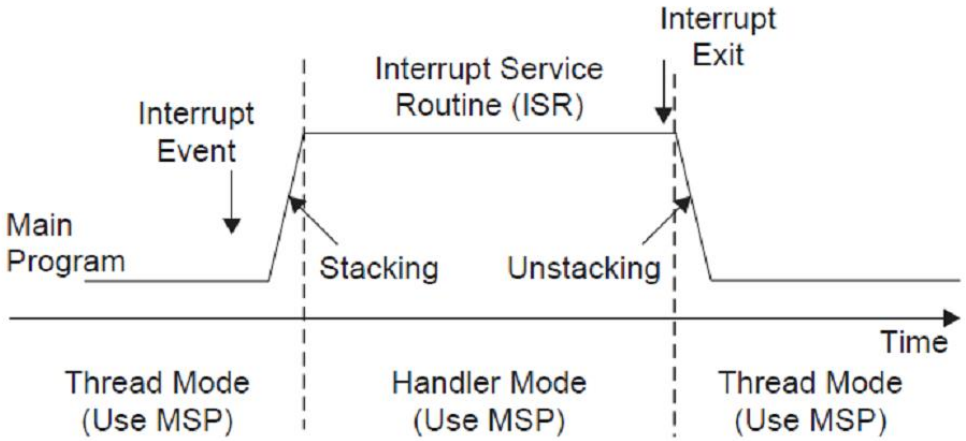


图 2.4.3 CONTROL[1]=0 时的堆栈使用情况

当 CONTROL[1]=1 时，线程模式将不再使用 PSP，而改用 MSP(handler 模式永远使用 MSP)。此时，进入异常时的自动压栈使用的是进程堆栈，进入异常 handler 后才自动改为 MSP，退出异常时切换回 PSP，并且从进程堆栈上弹出数据，如图 2.4.4 所示。

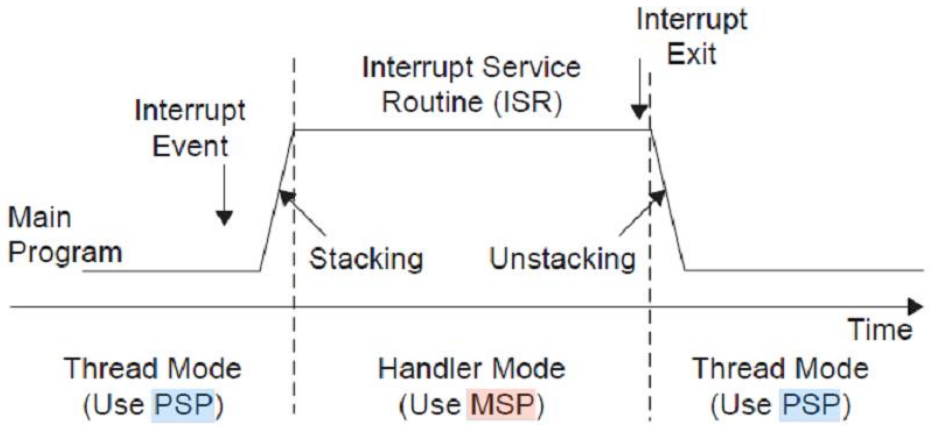


图 2.4.4 CONTROL[1]=1 时的堆栈使用情况

在特权级下，可以指定具体的堆栈指针，而不受当前使用堆栈的限制，示例代码如下：

```
MRS     R0,     MSP  ; 读取主堆栈指针到 R0
MSR     MSP,    R0   ; 写入 R0 的值到主堆栈中
MRS     R0,     PSP  ; 读取进程堆栈指针到 R0
MSR     PSP,    R0   ; 写入 R0 的值到进程堆栈中
```

通过读取 PSP 的值，OS 就能够获取用户应用程序使用的堆栈，进一步地就知道了在发生异常时，被压入寄存器的内容，而且还可以把其它寄存器进一步压栈的书写形式)。OS 还可以修改 PSP，用于实现多任务中的任务上下文切换。

2.4.3 Stack frames

在进入异常服务程序的时候会将一些数据压入堆栈中，这些数据所占用的数据块被称为 Stack frames，对于 M3 和 M4 没有使用 FPU 的时候其 Stack frames 总是 8 个字(一个字 32bit)如图 2.4.5 所示。

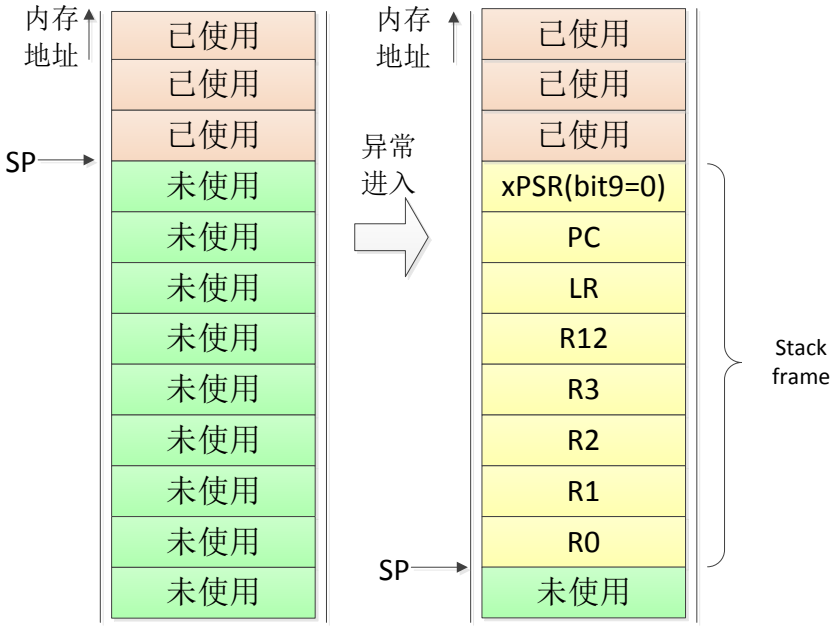


图 2.45 未使用双字对齐无 FPU 的 Stack frame

在符合 AAPCS 的应用程序中，对于响应异常时的堆栈操作是要进行双字对齐的。在 M3 或 M4 处理器中如果堆栈指针 SP 未双字对齐的话，那么就会在堆栈中自动的增加一个填充位使其双字对齐。双字对齐是一个可选项，欲使能此特性，需要把 NVIC 配置控制寄存器的 STKALIGN 置位，如下面汇编代码所演示：

```
LDR      R0,      =0xE000ED14          ; R0=NVIC CCR 的基址
LDR      R1,      [R0]
ORR.     W R1,    R1, #0x200           ; 设置 STKALIGN 位
STR      R1,      [R0]                 ; 更新 NVIC CCR
```

如果使用 C 语言，则代码如下：

```
#define NVIC_CCR ((volatile unsigned long *)(0xE000ED14))
*NVIC_CCR= *NVIC_CCR | 0x200;   //设置 STKALIGN 位
```

xPSP 寄存器的 bit9 被用来指示 SP 是否需要对齐，bit9 如果为 1 的话就需要双字对齐，如果为 0 的话就不需要双字对齐，未使用 FPU 时采用双字对齐的 Stack frame 如图 2.4.6 所示。

我们可以看到这里只是将 xPSR、PC、LR、R12、R0-R3 这 8 个寄存器自动入栈，其余的 8 个寄存器 R4-R11 就需要我们自己手动入栈了，入栈顺序不能乱了。

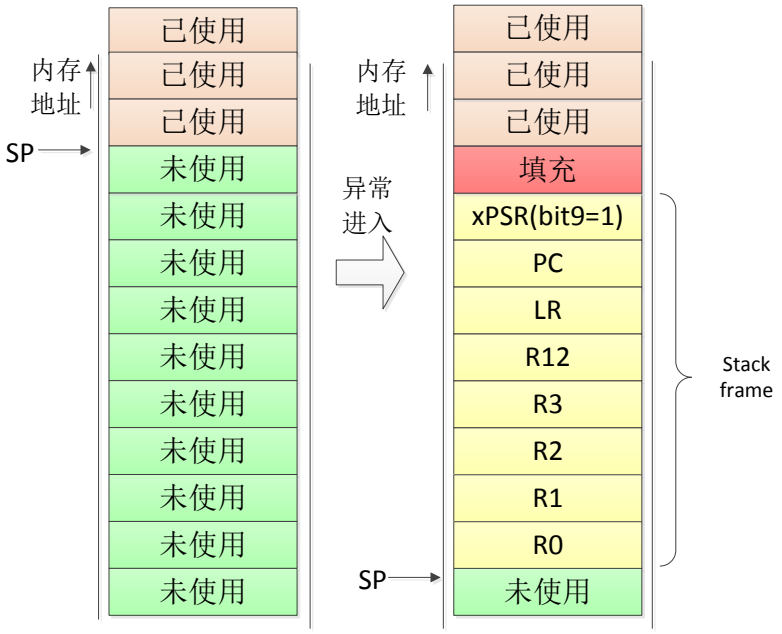


图 2.4.6 使用双字对齐无 FPU 时的 Stack frame

对于 Cortex-M4 来说因为有 FPU 单元，如果使能了 FPU 单元的话，那么 Stack frame 就会增加 S0-S15 和 FPSCR 寄存器，也可选择是否使用双字对齐，在 Cortex-M4 中默认开启了双字对齐功能，Stack frame 如图 2.4.7 所示。

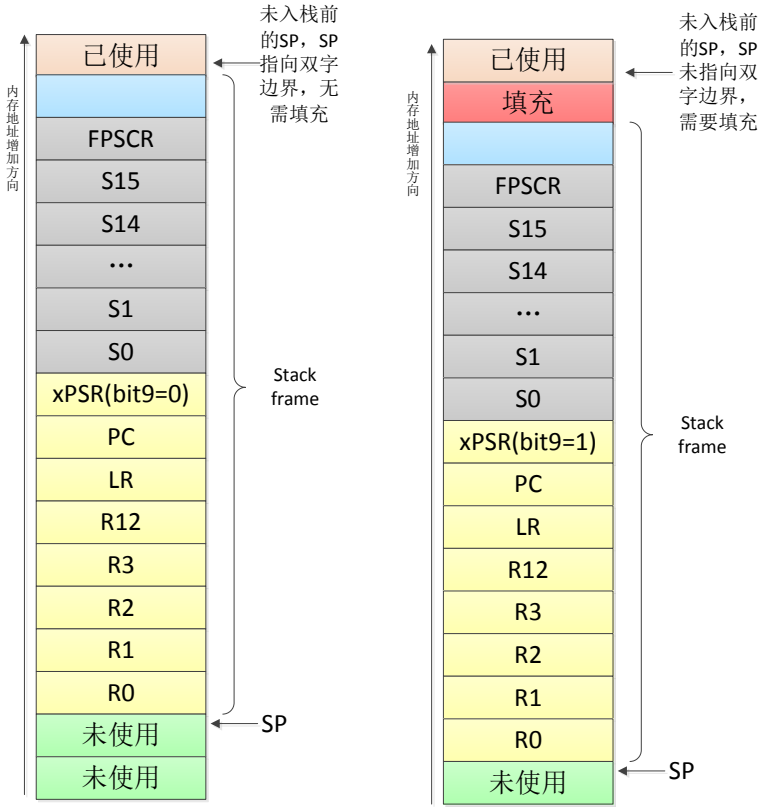


图 2.4.7 使用 FPU 时的 Stack frame

Stack frame 是在进入异常处理服务时被硬件自动入栈的，使用 FPU 的时候就会将 FPSCR、S0-S15、xPSR、PC、LR、R12、R0-R3 这 25 个寄存器自动入栈。在图 2.4.7 中我们

可以看到不管是否双字对齐在 FPSCR 的上方都会有一个蓝色的区域没有存放任何数据，也就是 Stack frame 一开始的地址是一个空区域。这一点一定要注意到，我们在修改 UCOSII 的堆栈初始化函数 OSTaskStkInit()的时侯堆栈的第一个位置写 0 就是这么来的。

我们可以看出还有 R4-R11，S16-S31 这些寄存器没有做处理，那么我们就要手动对这些寄存器做入栈和出栈处理。

2.5 SVC 和 PendSV 异常

2.5.1 SVC 异常

SVC(系统服务调用，亦简称系统调用)用于产生系统函数的调用请求。例如，操作系统不让用户程序直接访问硬件，而是通过提供一些系统服务函数，用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就会产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。这种“提出要求——得到满足”的方式，很好、很强大、很方便、很灵活、很能可持续发展。首先，它使用户程序从控制硬件的繁文缛节中解脱出来，而是由 OS 负责控制具体的硬件。第二，OS 的代码可以经过充分的测试，从而能使系统更加健壮和可靠。第三，它使用户程序无需在特权级下执行，用户程序无需承担因误操作而瘫痪整个系统的风险。第四，通过 SVC 的机制，还让用户程序变得与硬件无关，因此在开发应用程序时无需了解硬件的操作细节，从而简化了开发的难度和繁琐度，并且使应用程序跨硬件平台移植成为可能。开发应用程序唯一需要知道的就是操作系统提供的应用编程接口(API)，并且了解各个请求代号和参数表，然后就可以使用 SVC 来提出要求了。其实，严格地讲，操作硬件的工作是由设备驱动程序完成的，只是对应用程序来说，它们也是操作系统的一部分，如图 2.5.1 所示。

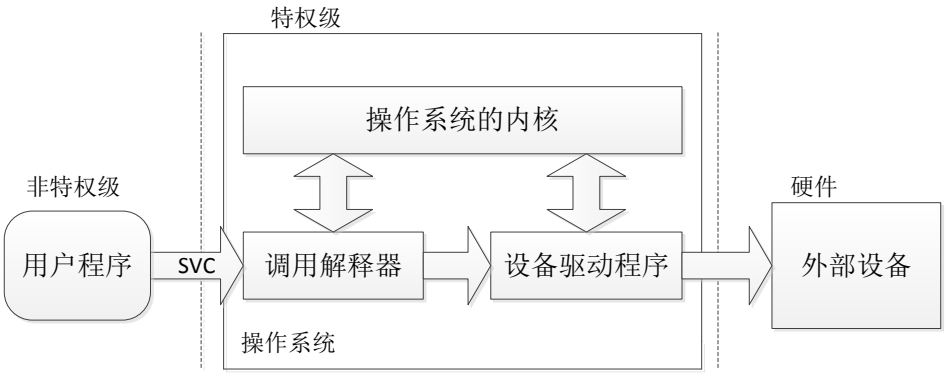


图 2.5.1 SVC 作为操作系统函数门户示意图

SVC 异常通过执行”SVC”指令来产生。该指令需要一个立即数，充当系统调用代号。SVC 异常服务例程稍后会提取出此代号，从而解释本次调用的具体要求，再调用相应的服务函数。例如，

```
SVC     0x3    ; 调用 3 号系统服务
```

在 SVC 服务例程执行后，上次执行的 SVC 指令地址可以根据自动入栈的返回地址计算出。找到了 SVC 指令后，就可以读取该 SVC 指令的机器码，从机器码中萃取出立即数，就获知了请求执行的功能代号。如果用户程序使用的是 PSP，服务例程还需要先执行 MRS Rn,PSP 指令来获取应用程序的堆栈指针。通过分析 LR 的值，可以获知在 SVC 指令执行时正在使用哪个堆栈。

在 UCOS 中并未使用 SVC 这个功能，大家了解一下就行了。

2.5.2 PendSV 异常

PendSV（可悬起的系统调用），它和 SVC 协同使用。一方面， SVC 异常是必须立即得到响应的应用程序执行 SVC 时都是希望所需的请求立即得到响应。另一方面，PendSV 则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。OS 可以利用它“缓期执行”一个异常，直到其它重要的任务完成后才执行动作。悬起 PendSV 的方法是：手工往 NVIC 的 PendSV 悬起寄存器中写 1。悬起后，如果优先级不够高，则将缓期等待执行。

PendSV 的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：

- 执行一个系统调用
- 系统滴答定时器(SYSTICK)中断，（轮转调度中需要）

让我们举个简单的例子来辅助理解。假设有这么一个系统，里面有两个就绪的任务，并且通过 SysTic k 异常启动上下文切换，如图 2.5.2 所示。

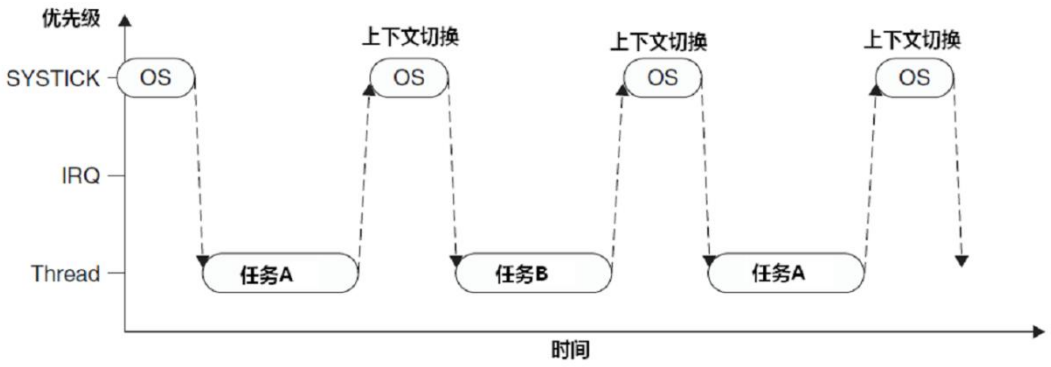


图 2.5.2 两个任务通过 SysTick 轮转调度的简单模式

图 2.5.2 是两个任务轮转调度的示意图。但若在产生 Sys Tick 异常时正在响应一个中断，则 SysTick 异常会抢占其 ISR。在这种情况下，OS 不得执行上下文切换，否则将使中断请求被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这种事。因此，在 CM3/CM4 中也是严禁没商量——如果 OS 在某中断活跃时尝试切入线程模式，将触犯用法 fault 异常，如图 2.5.3 所示。

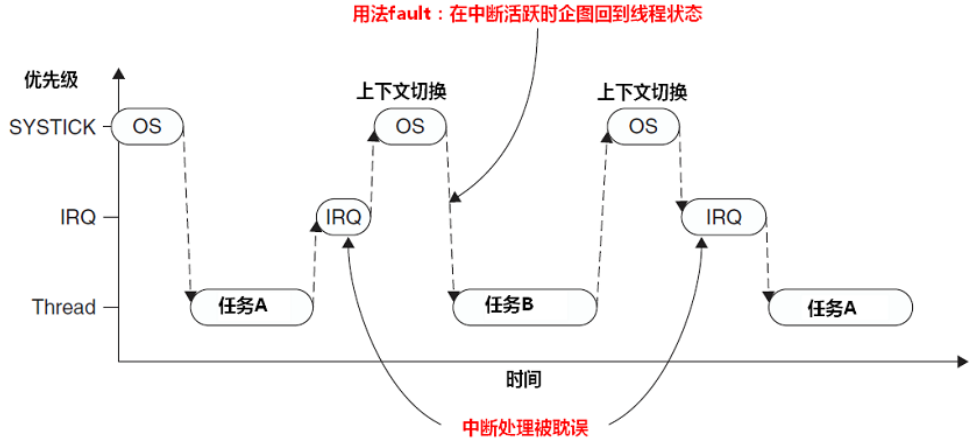


图 2.5.3 发生 IRQ 时上下文切换的问题

为解决此问题，早期的 OS 会检测当前是否有中断在活跃中，只有没有任何中断需要响应

时，才执行上下文切换（切换期间无法响应中断）。然而，这种方法的弊端在于，它可以把任务切换动作拖延很久(因为如果抢占了 IRQ，本次 SysTick 在执行后不得作上下文切换，只能等待下一次 SysTick 异常），尤其是当某中断源的频率和 Sys Tick 异常的频率比较接近时，会发生“共振”。

现在好了， Pend SV 来完美解决这个问题了。 Pend SV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。如果 OS 检测到某 IRQ 正在活动并且被 SysTick 抢占，它将悬起一个 PendSV 异常，以便缓期执行上下文切换，如图 2.5.4。

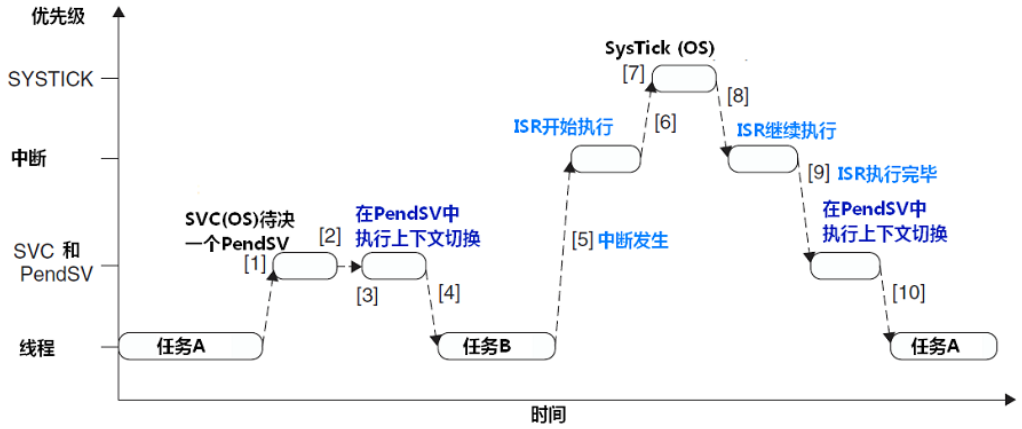


图 2.5.4 使用 PendSV 控制上下文切换

图 2.5.4 中事件的流水账记录如下：

(1) 任务 A 呼叫 SVC 来请求任务切换（例如，等待某些工作完成）
(2) OS 接收到请求，做好上下文切换的准备，并且 pend 一个 PendSV 异常。
(3) 当 CPU 退出 SVC 后，它立即进入 PendSV，从而执行上下文切换。
(4) 当 PendSV 执行完毕后，将返回到任务 B，同时进入线程模式。
(5) 发生了一个中断，并且中断服务程序开始执行
(6) 在 ISR 执行过程中，发生 SysTick 异常，并且抢占了该 ISR。
(7) OS 执行必要的操作，然后 pend 起 PendSV 异常以作好上下文切换的准备。
(8) 当 SysTick 退出后，回到先前被抢占的 ISR 中， ISR 继续执行
(9) ISR 执行完毕并退出后， PendSV 服务例程开始执行，并且在里面执行上下文切换。
(10) 当 PendSV 执行完毕后，回到任务 A，同时系统再次进入线程模式。

第三章 移植文件讲解

在第一章我们讲解了 UCOSII 在 STM32F407 开发板上的移植过程，第二章讲解了一下 Cortex-M3 和 M4 处理器的一些基础知识，本章我们就结合前两章内容讲解一下我们在 UCOSII 移植过程中的一些重要文件和我们移植 UCOSII 的过程中都做了那些工作，本章分为如下几部分：

3.1 滴答定时器 Systick
3.2 os_cpu_a.asm 文件详解
3.3 os_cpu.h 文件详解
3.4 os_cpu_c.c 文件详解

3.1 滴答定时器 SysTick

滴答定时器是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时器初值，只要不把它在 SysTick 控制以及状态寄存器中的使能位清零，就将永久不息。SysTick 的最大使命，就是定期地产生异常请求作为系统的时基。OS 都需要这种“滴答”来推动任务和时间的管理。

我们在移植 UCOSII 的过程中就要使用滴答定时器来作为系统时钟，首先就是对滴答定时器的设置，主要是设置它的定时周期，我们是在 delay_init()函数中完成滴答定时器设置的，delay_init()函数代码如下。

```
//初始化延迟函数
//当使用 ucos 的时候,此函数会初始化 ucos 的时钟节拍
//SYSTICK 的时钟固定为 HCLK 时钟的 1/8
//SYSCLK:系统时钟
void delay_init(u8 SYSCLK)
{
#if SYSTEM_SUPPORT_UCOS  //如果需要支持 OS.
    u32 reload;
#endif
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);
    fac_us=SYSCLK/8;                        //为系统时钟的 1/8
#if SYSTEM_SUPPORT_UCOS                     //如果需要支持 OS.
    reload=SYSCLK/8;                        //每秒钟的计数次数 单位为 K
    reload*=1000000/delay_tickspersec;      //根据 OS_TICKS_PER_SEC 设定溢出时间
                                            //reload 为 24 位寄存器,最大值:16777216,在
                                            //168M 下,约合 0.7989s 左右
    fac_ms=1000/delay_tickspersec;          //代表 ucos 可以延时的最少单位
    SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk;  //开启 SYSTICK 中断
    SysTick->LOAD=reload;                     //每 1/OS_TICKS_PER_SEC 秒中断一次

    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk;   //开启 SYSTICK
#else
    fac_ms=(u16)fac_us*1000;//非 ucos 下,代表每个 ms 需要的 systick 时钟数
#endif   //SYSTEM_SUPPORT_UCOS
}
```

其中红色代码部分就是在使用 UCOSII 时配置 SysTick 的代码，如果 SYSTEM_SUPPORT_UCOS 被定义了就说明使用了 UCOS，那么我们就需要配置 SysTick。首先要根据 UCOSII 中的定义的 OS_TICKS_PER_SEC 来计算出 SysTick 的装载值 reload，开启 SysTick 中断，将 reload 值写进 SysTick 的 LOAD 寄存器中，最后开启 SysTick。

开启 SysTick 后还要编写 SysTick 的中断服务函数 SysTick_Handler()，函数代码如下，同样也采用了条件编译。

```
//systick 中断服务函数,使用 ucos 时用到
void SysTick_Handler(void)
{
    if(delay_osrunning==1)              //OS 开始跑了,才执行正常的调度处理
```

```
    {
        OSIntEnter();                   //进入中断
        OSTimeTick();                   //调用 ucos 的时钟服务程序
        OSIntExit();                    //触发任务切换软中断
    }
}
```

3.2 os_cpu_a.asm 文件详解

为了方便起见，os_cpu_a.asm 文件我们分段来介绍。

```
IMPORT  OSRunning
IMPORT  OSPrioCur
IMPORT  OSPrioHighRdy
IMPORT  OSTCBCur
IMPORT  OSTCBHighRdy
IMPORT  OSIntNesting
IMPORT  OSIntExit
IMPORT  OSTaskSwHook

EXPORT  OSStartHighRdy
EXPORT  OSCtxSw
EXPORT  OSIntCtxSw
EXPORT  OS_CPU_SR_Save
EXPORT  OS_CPU_SR_Restore
EXPORT  PendSV_Handler
```

上面代码分为两部分，上半部分使用 IMPORT 来定义，下半部分使用 EXPORT 来定义。IMPORT 定义表示这是一个外部变量的标号，不是在本程序定义的；EXPORT 定义表示这些函数是在本文件中定义的，供其它文件调用。

```
NVIC_INT_CTRL    EQU    0xE000ED04     ; 中断控制寄存器
NVIC_SYSPRI2     EQU    0xE000ED22     ; 系统优先级寄存器(2)
NVIC_PENDSV_PRI  EQU    0xFFFF         ; PendSV 中断优先级和
                                       ;Systick 中断优先级都为最低(0xFF)
NVIC_PENDSVSET   EQU    0x10000000     ; 触发软件中断的值。
```

EQU 和 C 语言中的#define 一样，定义一个宏。NVIC_INT_CTRL 为中断控制寄存器，地址为 0xE000ED04；NVIC_SYSPRI14 为 PendSV 中断优先级寄存器，地址为 0xE000ED22；NVIC_PENDSV_PRI 为 PendSV 和 Systick 的中断优先级，这里为 0xFFFF，都为最低优先级；NVIC_PENDSVSET 可以触发软件中断，通过给中断控制寄存器(NVIC_INT_CTRL)的 bit28 写 1 来触发软件中断，因此 NVIC_PENDSVSET 为 0x10000000。

```
OS_CPU_SR_Save                  ;关中断
    MRS     R0, PRIMASK         ;读取 PRIMASK 到 R0,R0 为返回值
    CPSID   I                   ;PRIMASK=1,关中断(NMI 和硬件 FAULT 可以响应)
    BX      LR                  ;返回

OS_CPU_SR_Restore               ;开中断
```

```
    MSR     PRIMASK, R0    ;读取 R0 到 PRIMASK 中,R0 为参数
    BX      LR             ;返回
```

OS_CPU_SR_Save 和 OS_CPU_SR_Restore 是开关中断的汇编代码，通过给 PRIMASK 写 1 来关中断，写 0 来打开中断。这里也可是使用 CPS 指令来快速的开关中断，我们在 OS_CPU_SR_Save 中就使用了 CPSID I 来关中断。

```
OSStartHighRdy
    LDR     R4,     =NVIC_SYSPRI14          ; 设置 PendSV 的优先级
    LDR     R5,     =NVIC_PENDSV_PRI
    STR     R5,     [R4]
    MOV     R4,     #0                      ; R4=0
    MSR     PSP,    R4                      ;PSP=0
    LDR     R4,     =OSRunning
    MOV     R5,     #1                      ;R5=1
    STRB    R5,     [R4]                    ;设置 OSRunning=1
    LDR     R4,     =NVIC_INT_CTRL          ;R4=NVIC_INT_CTRL
    LDR     R5,     =NVIC_PENDSVSET         ;R5=NVIC_PENDSVSET
    STR     R5,     [R4]                    ;触发 PendSV 中断
    CPSIE   I                               ;开中断
OSStartHang
B       OSStartHang                         ;死循环，应该不会到这里的
```

OSStartHighRdy 是由 OSStart()调用，用来开启多任务的，如果多任务开启失败的话就会进入 OSStartHang。

```
OSCtxSw
        PUSH    {R4, R5}
        LDR     R4,     =NVIC_INT_CTRL      ;触发 PendSV 异常
        LDR     R5,     =NVIC_PENDSVSET
        STR     R5,     [R4]                ;向 NVIC_INT_CTRL 写入
                                            ;NVIC_PENDSVSET 触发 PendSV 中断
        POP     {R4, R5}
        BX      LR
OSIntCtxSw
        PUSH    {R4, R5}
        LDR     R4,     =NVIC_INT_CTRL
        LDR     R5,     =NVIC_PENDSVSET
        STR     R5, [R4]                    ;向 NVIC_INT_CTRL 写入
                                            ;NVIC_PENDSVSET 触发 PendSV 中断
        POP     {R4, R5}
        BX      LR
        NOP
```

OSCtxSw 和 OSIntCtxSw 这两个是用来做任务切换的，这两个看起来都是一样的，其实它们都只是触发一个 PendSV 中断，具体的切换过程在 PendSV 中断服务函数里面进行。这两个函数看起来是一样的，但是他们的意义是不同的，OSCtxSw 是任务级切换，比如从任务 A 切换到任务，OSIntCtxSw 是中断级切换，是从中断退出时切换到一个任务中，从中断切换到任务时，

CPU 的寄存器入栈工作已经完成，无需做第二次。

```
PendSV_Handler
    CPSID   I                               ;关中断，任务切换期间要关中断
    MRS     R0,     PSP                     ;R0=PSP
    CBZ     R0,     PendSV_Handler_Nosave   ; 如果 PSP 为 0 就转移          (1)
                                            ;到 PendSV_Handler_Nosave

    ;任务如果使用 FPU 的话就保存 S16-S31 寄存器
    TST     R14,    #0x10                                                   (2)
    IT      EQ
    VSTMDBEQ        R0!,    {S16-S31}

    SUBS    R0,     R0,     #0x20           ;R0-=0x20
    STM     R0,     {R4-R11}                ; 保存剩余的 R4-R11 寄存器
    LDR     R1,     =OSTCBCur               ; R1=&OSTCBCur
    LDR     R1,     [R1]                    ;R1=*R1 既 R1=OSTCBCur
    STR     R0,     [R1]                    ; *R1=R0 既 OSTCBCur=SP

PendSV_Handler_Nosave
    PUSH    {R14}                           ; 保存 R14 的值，后面要调用函数
    LDR     R0,      =OSTaskSwHook          ; R0=&OSTaskSwHook
    BLX     R0                              ;调用 OSTaskSwHook()
    POP     {R14}                           ;恢复 R14

    LDR     R0,     =OSPrioCur              ;R0=&OSPrioCur
    LDR     R1,     =OSPrioHighRdy          ;R1=&OSPrioHighRdy
    LDRB    R2,     [R1]                    ;R2=*R1 既 R2=OSPrioHighRdy
    STRB    R2,     [R0]                    ;*R0=R2  既  OSPrioCur=OSPrioHighRdy

    LDR     R0,     =OSTCBCur               ;R0=&OSTCBCur
    LDR     R1,     =OSTCBHighRdy           ;R1=&OSTCBHighRdy
    LDR     R2,     [R1]                    ;R2=*R1 既 R2=OSTCBHighRdy
    STR     R2,     [R0]                    ;*R0=R2 既 OSTCBCur=OSTCBHighRdy

    LDR     R0,     [R2]                    ;R0=*R2,既 R0=OSTCBHighRdy，    (3)
                                            ;R0 是新任务的 SP
    LDM     R0,     {R4-R11}                ;从堆栈中恢复 R4-R11
    ADDS    R0,     R0,  #0x20              ;R0+=20

    ;任务如果使用 FPU 的话就将 S16-S31 从堆栈中恢复出来
    TST     R14, #0x10                                                      (4)
    IT      EQ
    VLDMIAEQ  R0!,      {S16-S31}
```

```
MSR     PSP,    R0              ; PSP=R0,用新任务的 SP 加载 PSP
ORR     LR,     LR,     #0x04   ;确保 LR 的位 2 为 1，返回后使用进程堆栈(5)
CPSIE   I                       ;开中断
BX      LR                      ;中断返回
NOP
END
```

上面的汇编代码才是真正的任务切换程序，在每行代码后都有详细的注释，为了更好的理解我们对代码中打标号的地方重点讲解一下。

(1)如果 PSP 为 0 的话说明是第一次做任务切换，而任务创建的时候会调用堆栈初始化函数 OSTaskStkInit()来初始化堆栈，在初始化的过程中已经做了入栈处理，所以这里就不需要在做入栈处理，直接跳转到 PendSV_Handler_Nosave。

(2)我们前面讲过 EXC_RETURN 的 bit4 用来表示是否使用 FPU,所以我们通过判断 R14 的 bit4 来决定是否将 S16-S31 寄存器做入栈处理。

(3)此时 SP 指向的就是要运行的最高优先级的任务。

(4)同(2)一样。

(5)因为进入中断使用的是 MSP，而退出中断的时候使用的是 PSP，因此这里需要将 LR 的位 2 置 1。

在第一章 UCOSII 移植的时候说过要屏蔽掉 stm32f4xx_it.c 文件中的 PendSV_Handler()中断服务函数，原因就是我们在 os_cpu_a.asm 中重新定义了 PendSV_Handler()函数，我们有时候在移植的时候可能会发现有 OS_CPU_PendSVHandler 这样的函数,其实这是官方移植使用到的,具体作用和 PendSV_Handler 一样都是 PendSV 的中断服务函数，不过 ST 官方启动文件 startup_stm32f40_41xxx.s 中定义的 PendSV 中断服务函数为 PendSV_Handler，所以说如果要使用 OS_CPU_PendSVHandler 作为 PendSV 的中断服务函数就需要修改启动文件 startup_stm32f40_41xxx.s 中的中断向量表。

3.3 os_cpu.h 文件详解

```
typedef unsigned char   BOOLEAN;
typedef unsigned char   INT8U;          //无符号 8 位数
typedef signed    char  INT8S;          //有符号 8 位数
typedef unsigned short  INT16U;         //无符号 16 位数
typedef signed    short INT16S;         //有符号 16 位数
typedef unsigned int    INT32U;         //无符号 32 位数
typedef signed    int   INT32S;         //有符号 32 位数
typedef float           FP32;           //单精度浮点数
typedef double          FP64;           //双精度浮点数

//STM32 是 32 位位宽的,这里 OS_STK 和 OS_CPU_SR 都应该为 32 位数据类型
typedef unsigned int    OS_STK;         //OS_STK 为 32 位数据，也就是 4 字节
typedef unsigned int    OS_CPU_SR;      //默认的 CPU 状态寄存器大小 32 位
```

上面代码主要是定义了一些数据类型，在这里我们一定要注意 OS_STK 这个数据类型，我们在定义任务堆栈的时候就是定义为 OS_STK 类型的，这是一个 32 位的数据类型，按字节来算的话实际堆栈大小是我们定义的 4 倍。

```
//定义栈的增长方向.
//CM3 中,栈是由高地址向低地址增长的,所以 OS_STK_GROWTH 设置为 1
#define    OS_STK_GROWTH              1       //堆栈增长方向
//任务切换宏,由汇编实现.
#define    OS_TASK_SW()               OSCtxSw()

//OS_CRITICAL_METHOD = 1 :直接使用处理器的开关中断指令来实现宏
//OS_CRITICAL_METHOD = 2 :利用堆栈保存和恢复 CPU 的状态
//OS_CRITICAL_METHOD = 3 :利用编译器扩展功能获得程序状态字，保存在局部变量 cpu_sr

#define    OS_CRITICAL_METHOD     3         //进入临界段的方法

#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
```

上面代码中我们定义了堆栈的增长方向，任务级切换的宏定义 OS_TASK_SW。如果 OS_CRITICAL_METHOD 被定义为 3 的话那么进出临界段的宏定义分别为 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()，他们都是由汇编编写的。

3.4 os_cpu_c.c 文件详解

os_cpu_c.c 文件里面主要定义了几钩子函数，这里我们就不具体讲解这些钩子函数了，我们主要来看一下 OSTaskStkInit()这个函数，OSTaskStkInit()是堆栈初始化函数，函数代码如下。

```
OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;
(void)opt;              //opt 未使用
stk   = ptos;           // Load stack pointer
#if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *(--stk) = (INT32U)0x00000000L;      //No Name Register
    *(--stk) = (INT32U)0x00001000L;      //FPSCR
    *(--stk) = (INT32U)0x00000015L;      //S15
    *(--stk) = (INT32U)0x00000014L;      //S14
    *(--stk) = (INT32U)0x00000013L;      //S13
    *(--stk) = (INT32U)0x00000012L;      //S12
    *(--stk) = (INT32U)0x00000011L;      //S11
    *(--stk) = (INT32U)0x00000010L;      //S10
    *(--stk) = (INT32U)0x00000009L;      //S9
    *(--stk) = (INT32U)0x00000008L;      //S8
    *(--stk) = (INT32U)0x00000007L;      //S7
    *(--stk) = (INT32U)0x00000006L;      //S6
    *(--stk) = (INT32U)0x00000005L;      //S5
    *(--stk) = (INT32U)0x00000004L;      //S4
```

```
    *(--stk) = (INT32U)0x00000003L;      //S3
    *(--stk) = (INT32U)0x00000002L;      //S2
    *(--stk) = (INT32U)0x00000001L;      //S1
    *(--stk) = (INT32U)0x00000000L;      //S0
#endif
    *(--stk)    = (INT32U)0x01000000L;   //xPSP
    *(--stk)    = (INT32U)task;          //PC
    *(--stk)    = (INT32U)OS_TaskReturn; //R14
    *(--stk)    =(INT32U)0x12121212L;    //R12
    *(--stk)    = (INT32U)0x03030303L;   //R3
    *(--stk)    = (INT32U)0x02020202L;   //R2
    *(--stk)    = (INT32U)0x01010101L;   //R1
    *(--stk)    = (INT32U)p_arg;         //R0

#if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *(--stk) = (INT32U)0x00000031L;      //S31
    *(--stk) = (INT32U)0x00000030L;      //S30
    *(--stk) = (INT32U)0x00000029L;      //S29
    *(--stk) = (INT32U)0x00000028L;      //S28
    *(--stk) = (INT32U)0x00000027L;      //S27
    *(--stk) = (INT32U)0x00000026L;      //S26
    *(--stk) = (INT32U)0x00000025L;      //S25
    *(--stk) = (INT32U)0x00000024L;      //S24
    *(--stk) = (INT32U)0x00000023L;      //S23
    *(--stk) = (INT32U)0x00000022L;      //S22
    *(--stk) = (INT32U)0x00000021L;      //S21
    *(--stk) = (INT32U)0x00000020L;      //S20
    *(--stk) = (INT32U)0x00000019L;      //S19
    *(--stk) = (INT32U)0x00000018L;      //S18
    *(--stk) = (INT32U)0x00000017L;      //S17
    *(--stk) = (INT32U)0x00000016L;      //S16
#endif
    *(--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);
}
```

堆栈初始化函数 OSTaskStkInit()是由任务创建函数 OSTaskCreate()和 OSTaskCreateExt()这

两个函数调用的，用于在创建任务的时候初始堆栈，从上面的代码中可以看出就是在任务堆栈中保存寄存器的值。如果使用 FPU 的话我们就要保存 FPU 寄存器，否则只保存通用寄存器。

这里一定要注意入栈顺序，我们前面讲过如果开启 FPU 并且使用了 Lazy Stacking 特性的话(Cortex-M4 默认是开启了的)就会将 FPSCR、S0-S15、xPSR、PC、LR、R12、R0-R3 这些寄存器自动入栈，这几个寄存器的入栈顺序我们在将 Stack frame 的时候已经讲了，可以看到在 OSTaskStkInit()函数中就是按照这个顺序入栈的。剩下的 S16-S31 和 R4-R11 就需要我们手动入栈了。至此，移植文件的讲解就完成了，关于这几个文件的详细内容，大家要参考源码的。

第四章 UCOSIII 移植

在 2009 年 Micrium 推出了 UCOSIII，相对于 UCOSII 性能有了进一步的提升，支持时间片轮转调度，极短的关中断事件等。本章我们就讲解如何在 STM32F407 开发板上移植 UCOSIII 操作系统。

4.1 UCOSIII 简介
4.2 移植准备工作
4.3 UCOS III 移植
4.4 软件设计
4.5 下载验证

4.1 UCOSIII 简介

UCOSIII 是一个可裁剪、可固化、可剥夺的多任务系统，没有任务数目的限制，是 UCOS 的第三代内核，UCOSIII 有以下几个重要的特性：

可剥夺多任务管理：UCOSIII 和 UCOSII 一样都属于可剥夺的多任务内核，总是执行当前就绪的最高优先级任务。

同优先级任务的时间片轮转调度：这个是 UCOSIII 和 UCOSII 一个比较大的区别,UCOSIII 允许一个任务优先级被多个任务使用，当这个优先级处于最高就绪态的时候，UCOSIII 就会轮流调度处于这个优先级的所有任务，让每个任务运行一段由用户指定的时间长度，叫做时间片。

极短的关中断时间：UCOSIII 可以采用锁定内核调度的方式而不是关中断的方式来保护临界段代码，这样就可以将关中断的时间降到最低，使得 UCOSIII 能够非常快速的响应中断请求。

任务数目不受限制：UCOSIII 本身是没有任务数目限制的，但是从实际应用角度考虑，任务数目会受到 CPU 所使用的存储空间的限制，包括代码空间和数据空间。

优先级数量不受限制：UCOSIII 支持无限多的任务优先级。

内核对象数目不受限制：UCOSIII 允许定义任意数目的内核对象。内核对象指任务、信号量、互斥信号量、事件标志组、消息队列、定时器和存储快等。

软件定时器：用户可以任意定义“单次”和“周期”型定时器，定时器是一个递减计数器，递减到零就会执行预先定义好的操作。每个定时器都可以指定所需操作，周期型定时器在递减到零时会执行指定操作，并自动重置计数器值。

同时等待多个内核对象：UCOSIII 允许一个任务同时等待多个事件。也就是说，一个任务能够挂起在多个信号量或消息队列上，当其中任何一个等待的事件发生时，等待任务就会被唤醒。

直接向任务发送信号：UCOSIII 允许中断或任务直接给另一个任务发送信号，避免创建和使用诸如信号量或事件标志等内核对象作为向其他任务发送信号的中介，该特性有效地提高了系统性能。

直接向任务发送消息：UCOSIII 允许中断或任务直接给另一个任务发送消息，避免创建和使用消息队列作为中介。

任务寄存器：每个任务都可以设定若干个“任务寄存器”，任务寄存器和 CPU 硬件寄存器是不同的，主要用来保存各个任务的错误信息，ID 识别信息，中断关闭时间的测量结果等。

任务级时钟节拍处理：UCOSIII 的时钟节拍是通过一个专门任务完成的，定时中断仅触发该任务。将延迟处理和超时判断放在任务级代码完成，能极大地减少中断延迟时间。

防止死锁：所有 UCOSIII 的“等待”功能都提供了超时检测机制，有效地避免了死锁。

时间戳：UCOSIII 需要一个 16 位或 32 位的自由运行计数器（时基计数器）来实现时间测量，在系统运行时，可以通过读取该计数器来测量某一个事件的时间信息。例如，当 ISR 给任务发送消息时，会自动读取该计数器的数值并将其附加在消息中。当任务读取消息时，可得到该消息携带的时标，这样，再通过读取当前的时标，并计算两个时标的差值，就可以确定传递这条消息所花费的确切时间。

UCOS、UCOSII、UCOSIII 之间的特性比较如表 4.1.1 所示。

特性	UCOS	UCOSII	UCOSIII
年份	1992	1998	2009
源代码	√	√	√
可剥夺型任务调度	√	√	√
最大任务数目	64	255	无限制
优先级相同的任务数目	1	1	无限制
时间片轮转调度	×	×	√
信号量	√	√	√
互斥信号量	×	√	√(可嵌套)
事件标志	×	√	√
消息邮箱	√	√	不再需要
消息队列	√	√	√
固定大小的存储管理	×	√	√
直接向任务发送信号	×	×	√
无需调度的发送机制	无	无	可选
直接向任务发送消息	×	×	√
软件定时器	×	√	√
任务挂起/恢复	×	√	√(可嵌套)
防止死锁	√	√	√
可裁剪	√	√	√
代码量	3K-8K	6K-26K	6K-24K
数据量	1K+	1K+	1K+
代码可固化	√	√	√
运行时可配置	×	×	√
编译时可配置	√	√	√
支持内核对象的 ASCII 命名	×	√	√
同时等待多个内核对象	×	√	√
任务寄存器	×	√	√
内置性能测试	×	基本	增强
用户可定义的介入函数	×	√	√
“POST”操作可加时间戳	×	×	√
内核察觉式调试	×	√	√
用汇编语言优化的调度器	×	×	√
捕获退出的任务	×	×	√
任务级时钟节拍处理	×	×	√
系统服务函数的数目	~20	~90	~70

表 4.1.1 各个版本 UCOS 的特性列表

4.2 移植准备工作

4.2.1 准备基础工程

同移植 UCOSII 一样，首先准备移植所需的基础工程，本章教程同样是在库函数版跑马灯实验的基础上完成的，基础工就是跑马灯实验了。

4.2.2 UCOSIII 源码

我们移植 UCOSIII 肯定需要 UCOSIII 源码了，这里我们需要两个文件：一个是 UCOSIII 的源码，一个是 Micrium 官方在 STM32F4xx 上移植好的工程文件。UCOSIII 源码下载地址为：http://micrium.com/downloadcenter/download-results/?searchterm=mp-uc-os-iii-1&supported=true，下载界面如图 4.2.1 所示，这个是 3.03 版本的 UCOSIII，这里我们已经下载好放在光盘中，路径：6、软件资料->UCOS 学习资料->UCOSIII 3.03。

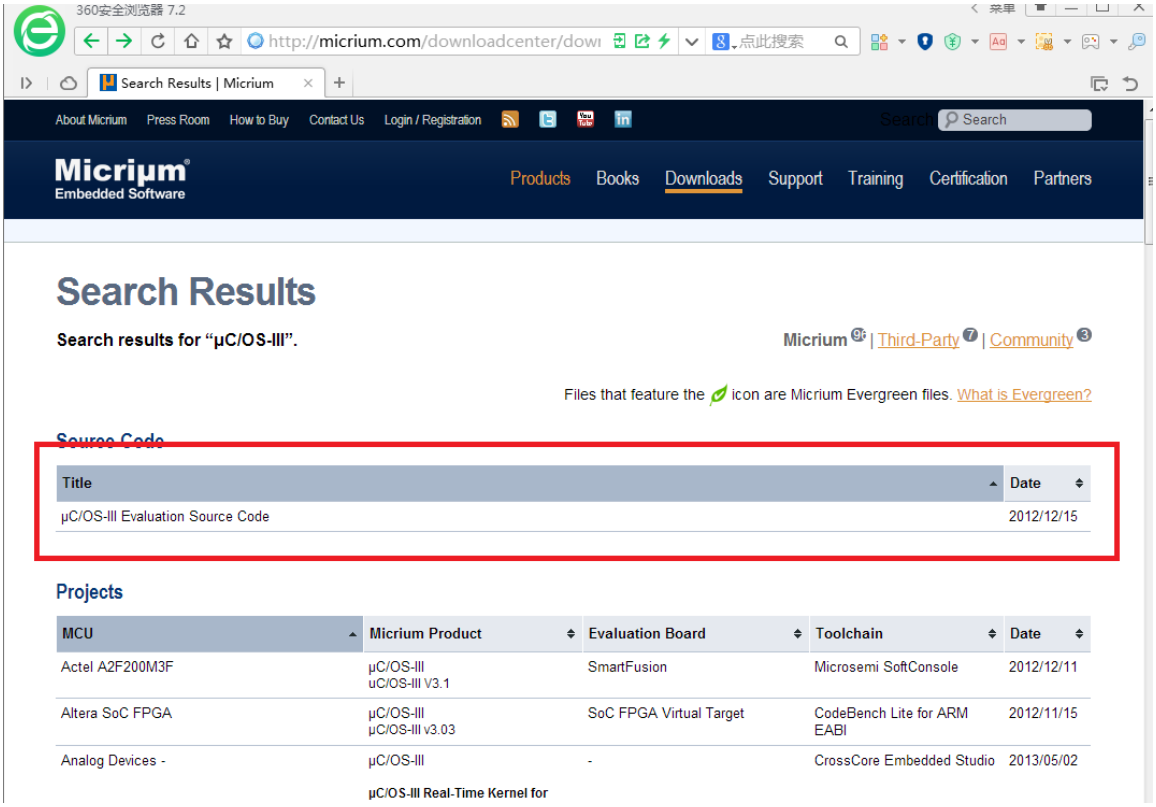


图 4.2.1 UCOSIII 源码下载界面

我们打开下载好的 UCOSIII 3.03 版本的源码，如图 4.2.2 所示，我们可以看到有 5 个文件，我们这里只关心 Source 文件夹里面的文件，这个文件夹里面的就是 UCOSIII 3.03 的源码。

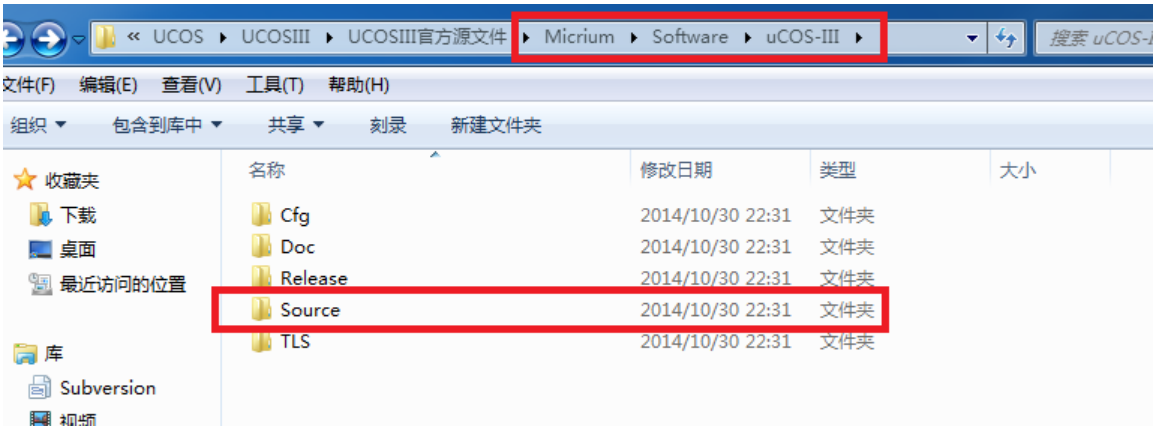


图 4.2.2 UCOSIII 3.03 源码

接着我们下载 Micrium 官方在 STM32F4xx 上移植好的 UCOSIII 工程，下载地址：http://micrium.com/downloadcenter/download-results/?searchterm=hm-stmicroelectronics&supported=true，下载界面如图 4.2.3 所示，我们已经下载好放到光盘中，路径：6、软件资料->UCOS 学习资料->UCOSIII 3.04。

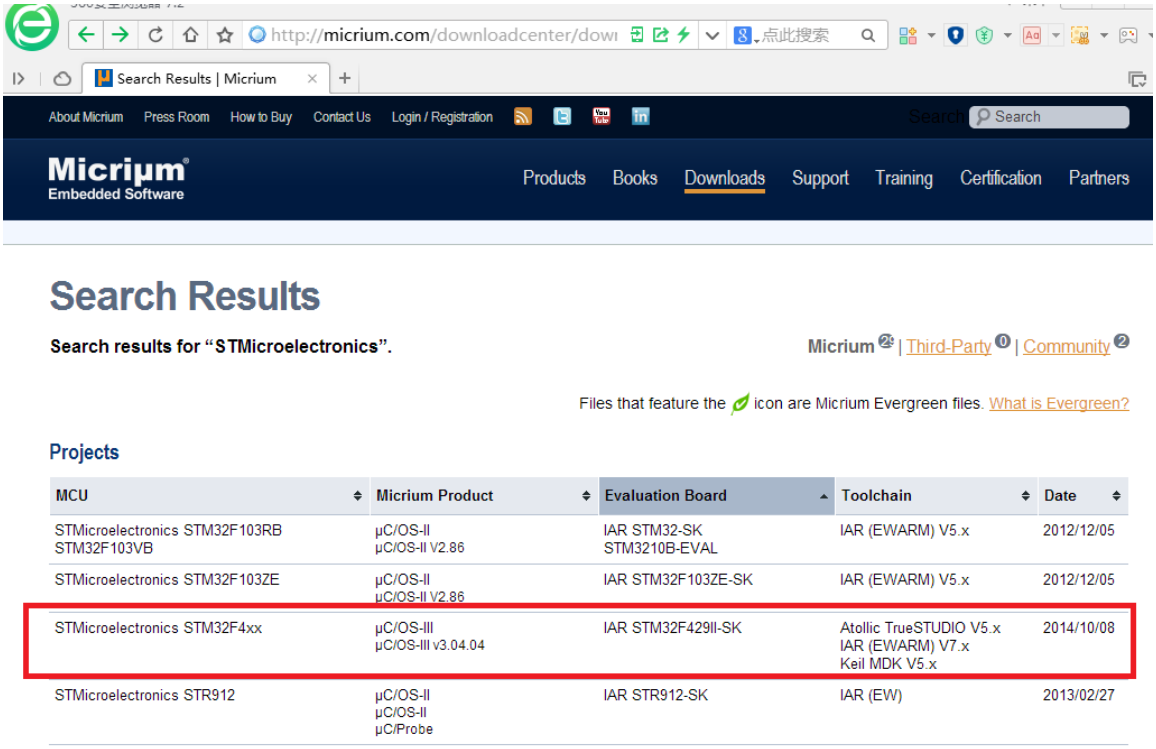


图 4.2.3Micrium 官方移植工程

需要注意一下两点！！！！

1、从图 4.2.2 可以看出 Micrium 官方是在 STM32F429 上移植的，并且 UCOSIII 的版本是 3.04。从第一章 UCOSII 的移植教程中我们可以看出有一些中间文件需要我们来实现。UCOSIII 移植也是一样的，既然 Micrium 已经在 STM32F4 上移植好了 UCOSIII，那么为了方便，这些中间文件我们就直接使用 Micrium 已经编写好的，Micrium 官方虽然是在 STM32F429 上移植的，但是完全可以应用在 STM32F407 上！

2、我们在移植的过程中会将 Micrium 官方使用的 3.04 版本的 UCOSIII 用 3.03 版本替换掉。我在移植 3.04 版本 UCOSIII 的时候遇到了这样一个问题：一旦调用

OSStatTaskCPUUsageInit()函数就会进入 hardfault，如果这时选择-O1 或者-O2 优化的话就没有问题，如果选择-O0 优化的话就会出现这种问题，开发环境使用的 MDK 5.11A，不知是 KEIL 问题还是 UCOSIII 3.04 版本的问题，所以为了保险起见我们使用 3.03 版本的 UCOSIII。另外，目前 UCOSIII 的资料基本都是基于 UCOSIII 3.03 版本的，所以这也是我们选择 3.03 版本 UCOSIII 的另一个主要原因。如果一定要使用 UCOSIII 3.04 的话，使用 KEIL 时一定要选择-O1 或者-O2 优化。

我们打开 Micrium 官方移植好的工程，也就是我们下载下来的 UCOSIII 3.04 源码，打开后如图 4.2.4 所示。

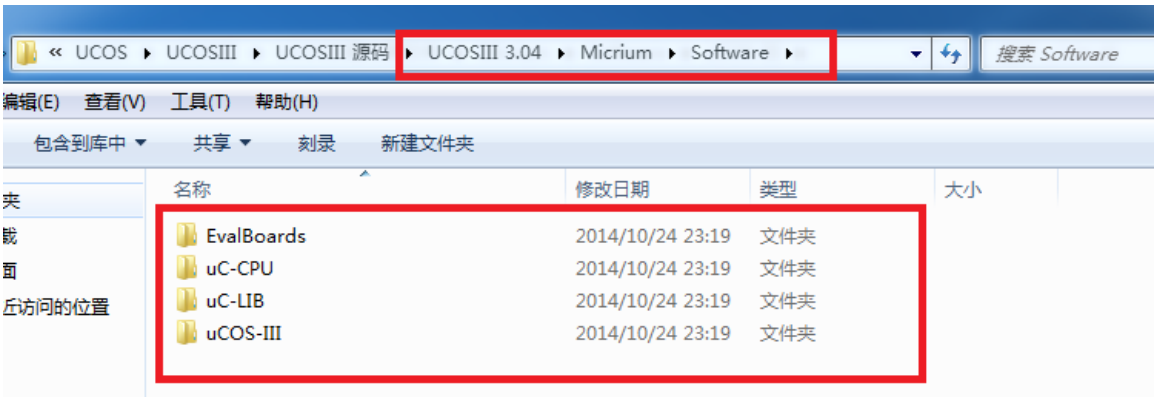


图 4.2.4 Micrium 官方移植工程

在图 4.2.4 中有四个文件夹：EvalBoards、uC-CPU、uC-LIB 和 uCOS-III，这四个文件的内容如下：

1、EvalBoards 文件夹

这个文件里面就是关于 STM32F429 的工程文件，我们是在 STM32F407 上移植的，我们打开这个文件如图 4.2.5 所示。

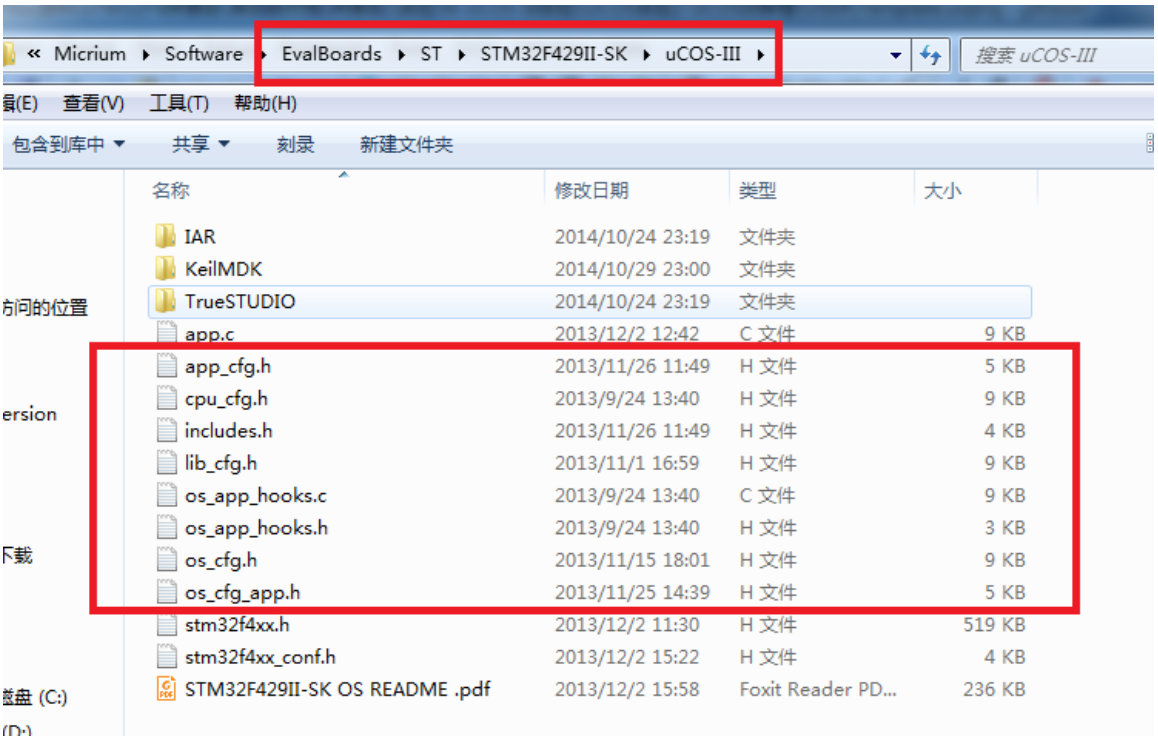


图 4.2.5 EvalBoards 文件

在图 4.2.5 中红框圈起来的是我们移植时候需要添加到我们的工程中的文件，一共有 8 个文件。

2、uC-CPU 文件夹

这个文件里面是与 CPU 相关的代码，有下面几个文件：

1) cpu_core.c 文件

该文件包含了适用于所有 CPU 架构的 C 代码。该文件包含了用来测量中断关闭事件的函数(中断关闭和打开分别由 CPU_CRITICAL_ENTER()和 CPU_CRITICAL_EXIT()两个宏实现)，还包含一个可模仿前导码零计算的函数(以防止 CPU 不提供这样的指令)，以及一些其他的函数。

2) cpu_core.h 文件

包含 cpu_core.c 中函数的原型声明，以及用来测量中断关闭时间变量的定义。

3) cpu_def.h 文件

包含 uC/CPU 模块使用的各种#define 常量。

4) cpu.h 文件

包含了一些类型的定义，使 UCOSIII 和其他模块可与 CPU 架构和编译器字宽度无关。在该文件中用户能够找到 CPU_INT16U、CPU_INT32U、CPU_FP32 等数据类型的定义。该文件还指定了 CPU 使用的是大端模式还是小端模式，定义了 UCOSIII 使用的 CPU_STK 数据类型，定义了 CPU_CRITICAL_ENTER()和 CPU_CRITICAL_EXTI()，还包括一些与 CPU 架构相关的函数的声明。

5) cpu_a.asm 文件

该文件包含了一些用汇编语言编写的函数，可用来开中断和关中断，计算前导零(如果 CPU 支持这条指令)，以及其他一些只能用汇编语言编写的与 CPU 相关的函数，这个文件中的函数可以从 C 代码中调用。

6) cpu_c.c 文件

包含了一些基于特定 CPU 架构但为了可移植而用 C 语言编写的函数 C 代码，作为一个普通原则，除非汇编语言能显著提高性能，否则尽量用 C 语言编写函数。

注意，上面的 cpu.h、cpu_a.asm 和 cpu_c.c 这三个文件，是在 uC-CPU 文件夹中 ARM-Cortex-M4 文件夹下的，我们打开 ARM-Cortex-M4 文件如图 4.2.6 所示。

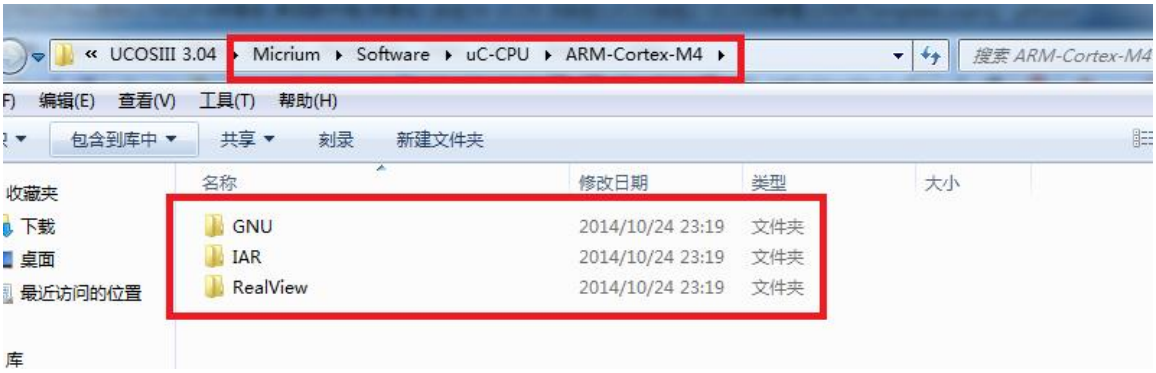


图 4.2.6 ARM-Cortex-M4 文件夹

从图 4.2.6 中可以看出一共有三个文件：GNU、IAR、RealView，这三个文件中都有 cpu.h、cpu_a.asm 和 cpu_c.c 这三个文件。我们使用的是 KEIL，所以我们在移植 UCOSIII 的时候选择 RealView 中的文件。在我们接下来的讲解中会看到同样的设计，根据不同的编译平台有不同的处理。

3、uC-LIB 文件

uC-LIB 是由一些可移植并且与编译器无关的函数组成，UCOS III 不使用 uC-LIB 中的函数，

但是 UCOS III 和 uC-CPU 假定 lib_def.h 是存在的，uC-LIB 包含以下几个文件：

1) lib_ascii.h 和 lib_ascii.c 文件

提供 ASCII_ToLower()、ASCII_ToUpper()、ASCII_IsAlpha()和 ASCII_IsDig()等函数，它们可以分别替代标准库函数 tolower()、toupper()、isalpha()和 isdigit()等。

2) lib_def.h 文件

定义了许多常量，如 RTUE/FALSE、YES/NO、ENABLE/DISABLE，以及各种进制的常量。但是，该文件中所有#define 常量都以 DEF_打头，所以上述常量的名字实际上为 DEF_TRUE/DEF_FALSE、DEF_YES/DEF_NO、DEF_ENABLE/DEF_DISABLE 等。该文件还为常用数学计算定义了宏。

3) lib_math.h 和 lib_math.c 文件

包含了 Math_Rand()、Math_SetRand()等函数的源代码，可用来替代标准库函数 rand()、srand()。

4) lib_mem.c 和 lib_mem.h 文件

包含了 Mem_Clr()、Mem_Set()、Mem_Copy()和 Mem_Cmp()等函数的源代码，可用来替代标准库函数 memclr()、memset()、memcpy()和 memcmp()等。

5) lib_str.c 和 lib_str.h 文件

包含了 Str_Lenr()、Str_Copy()和 Str_Cmp()等函数的源代码，可用于替代标准库函数 srtlen()、strcpy()和 strcmp()等。

6) lib_mem_a.asm 文件

包含了 lib_mem.c 函数的汇编优化版。

4、uCOS-III 文件

这个文件夹中有两个文件 Ports 和 Sourec，Ports 文件为与 CPU 平台有关的文件，Source 文件夹里面为 UCOSIII 3.04 的源码，我们打开 Source 文件夹如图 4.2.7 所示。

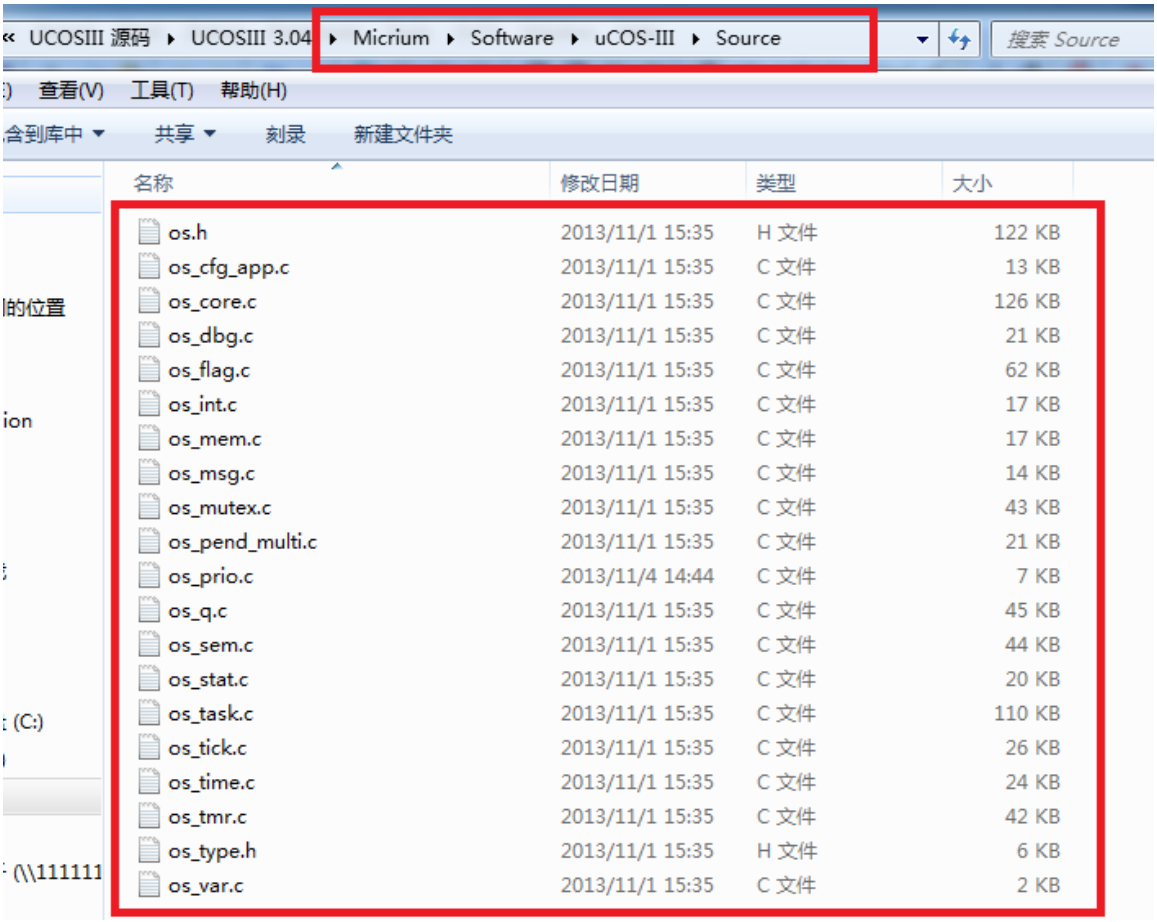


图 4.2.7 UCOSIII 3.04 源码文件

UCOSIII 3.04 和 UCOSIII 3.03 源码的文件都是一样的，不同的是各个文件里面的有些函数做了修改，UCOSIII 源码各个文件内容如表 4.2.1 所示。

文件	描述
os.h	包含 UCOSIII 的主要头文件，声明了常量、宏、全局变量、函数原型等
os_cfg_app.c	根据 os_cfg_app.h 中的宏定义声明变量和数组
os_core.c	UCOSIII 的内核功能模块
os_dbg.c	包含内核调试或 uC/Probe 使用的常量的声明
os_flag.c	包含事件标志的管理代码。
os_int.c	包含中断处理任务的代码。
os_mem.c	包含 UCOSIII 固定大小的存储分区的管理代码。
os_msg.c	包含消息处理的代码。
os_mutex.c	包含互斥型信号量的管理代码
os_pend_multi.c	包含允许任务同时等待多个信号量或多个消息队列的代码。
os_prio.c	包含位映射表的管理代码，用于追踪那些已经就绪的任务。
os_q.c	包含消息队列的管理代码。
os_sem.c	包含信号量的管理代码。
os_stat.c	包含统计任务的代码。

os_task.c	包含任务的管理代码。
os_tick.c	包含可管理正在延时和超时等待的任务的代码。
os_time.c	包含可是任务延时一段时间的代码。
os_tmr.c	包含软件定时器的管理代码。
os_type.h	包含 UCOSIII 数据类型的声明。
os_var.c	包含 UCOSIII 的全局变量。

表 4.2.1 UCOSIII 源码文件解释

4.3 UCOS III 移植

4.3.1 向工程中添加相应的文件

1) 新建相应的文件夹

在工程目录中新建一个 UCOSIII 文件夹，然后将我们下载的 Micrium 官方移植工程中的 uC-CPU、uC-LIB 和 UCOS-III 这三个文件复制到工程中，如图 4.3.1 所示。

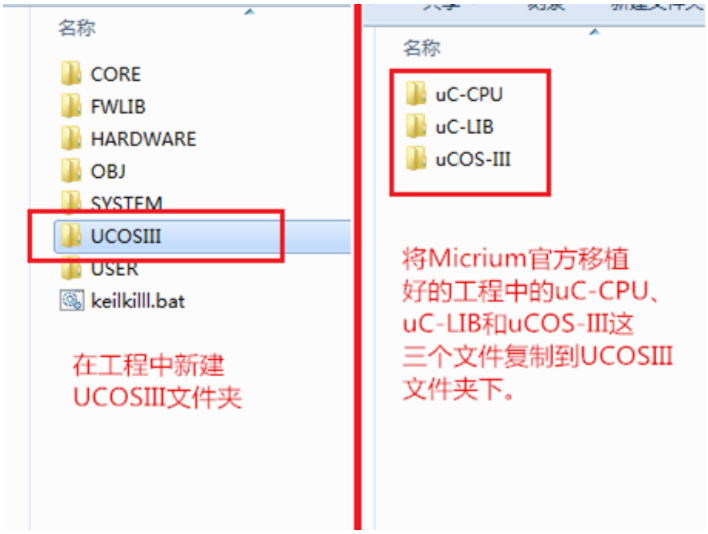


图 4.3.1 新建 UCOSIII 文件夹并添加相应文件

我们还需要在 UCOIII 文件中再新建两个文件：UCOS_BSP 和 UCOS_CONFIG，如图 4.3.2 所所示。

图 4.3.2 新建 UCOS_BSP 和 UCOS_CONFIG 两个文件

2) 向 UCOS_CONFIG 添加文件

复制 Micrium 官方移植好的工程中的相关文件到 UCOS_CONFIG 文件夹下，这些文件如图 4.3.3 所示，这些文件在 Micrium 官方移植工程中的路径为：

Micrium->Software->EvalBoards->ST->STM32F429II-SK->uCOS-III

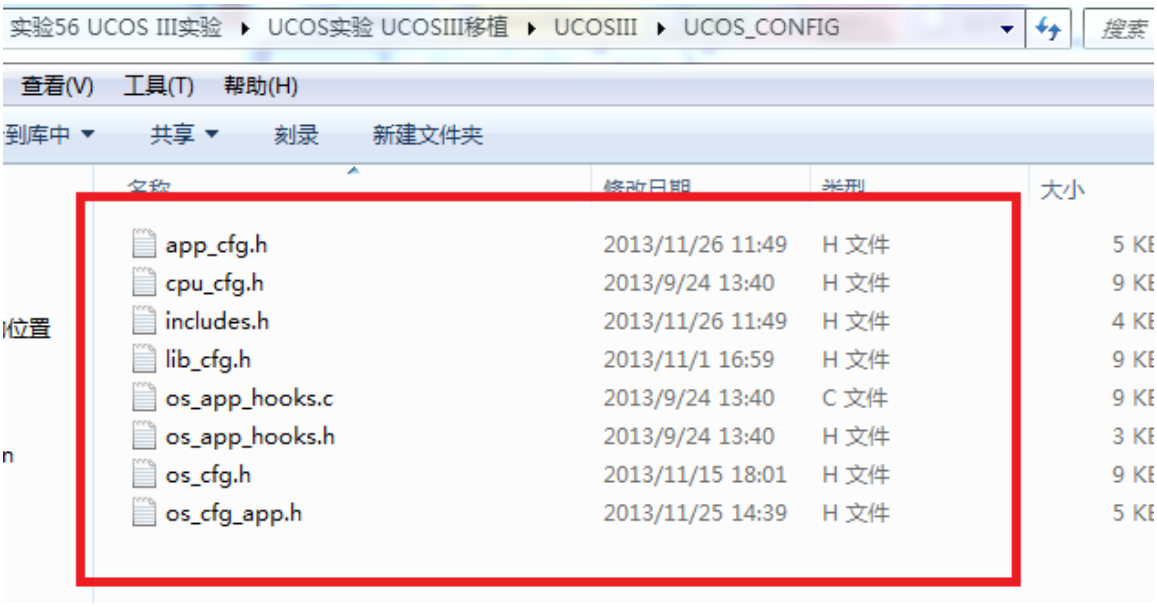


图 4.3.3 UCOS_CONFIG 中的文件

3) 向 UCOS_BSP 添加文件

同样复制 Micrium 官方移植好的工程中的相关文件到 UCOS_BSP 文件下，需要复制的文件如图 4.3.4 所示，这些文件在 Micrium 官方移植工程中的路径为：

Micrium->Software->EvalBoards->ST->STM32F429II-SK->BSP

图 4.3.4 UCOS_BSP 中的文件

4) 向工程中添加分组

我们在上面已经准备好了所需的文件，我们还要将这些文件添加到我们的工程中，我们在 KEIL 工程中新建如图 4.3.5 所示的分组。

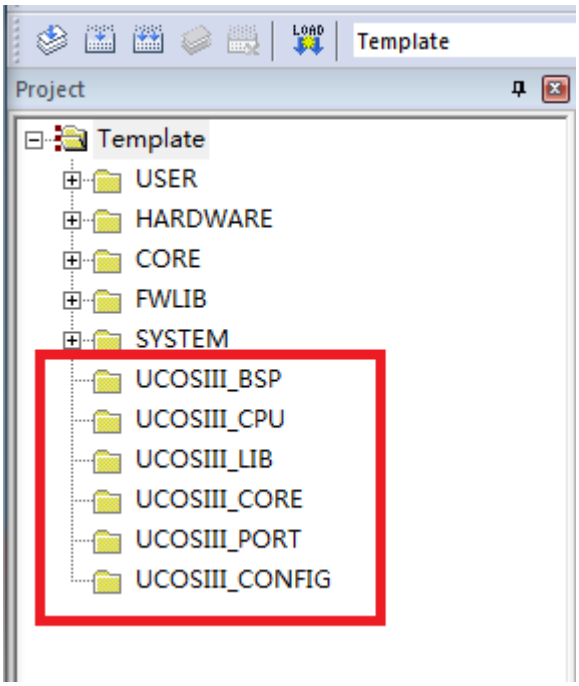


图 4.3.5 在工程中建立新分组

工程中分组建立完成以后我们就需要向新建的各个分组中添加文件了，按照如图 4.3.6 所示向各个分组中添加文件。

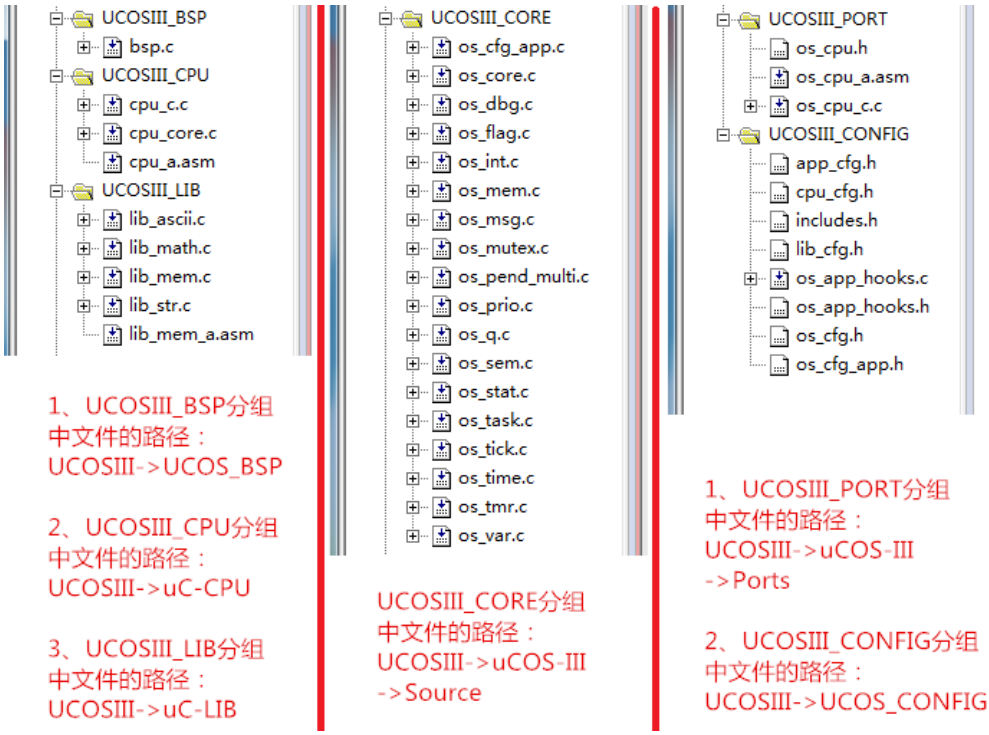


图 4.3.6 向 KEIL 工程中添加文件

向各个分组添加完文件以后我们还需要添加相应的头文件路径，按照图 4.3.7 所示添加头文件路径。

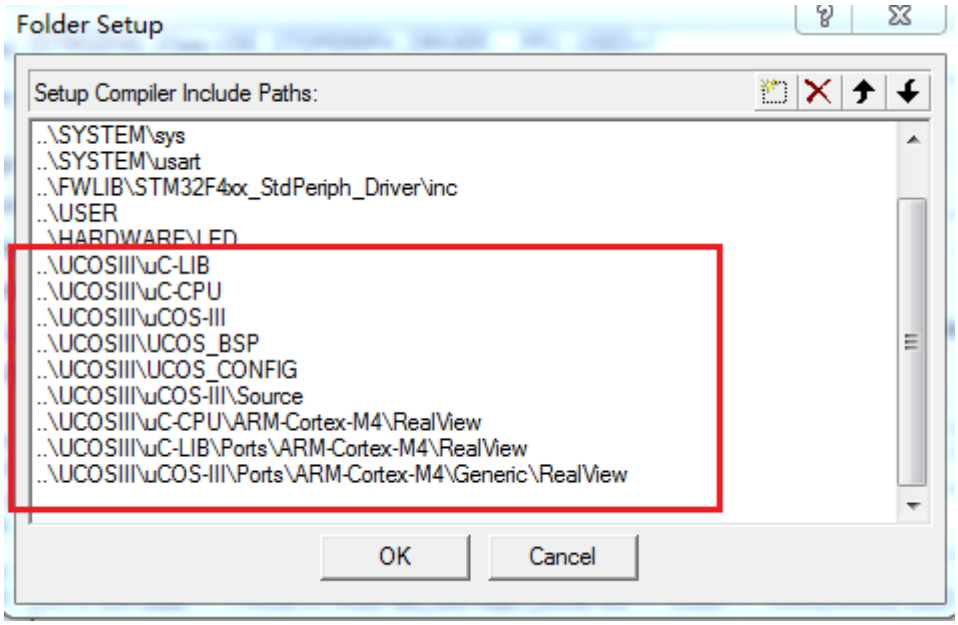


图 4.3.7 添加相应头文件

到这一步我们编译一下工程，会有如图 4.3.8 所示的错误提示，提示我们在 bsp.c 文件中 BSP_IntInit()和 BSP_PeriphEn()这两个函数未定义，这里我们先不管这两个错误，稍后我们会讲解的。

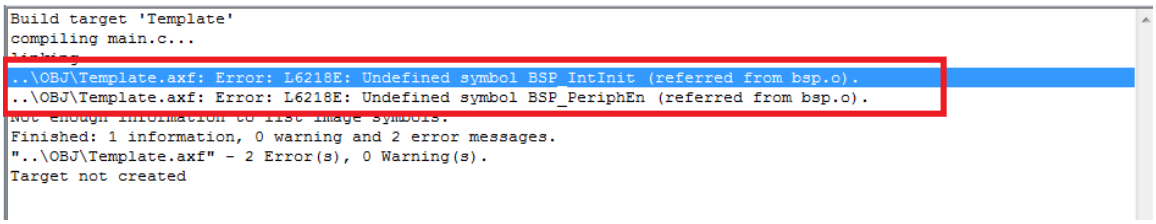


图 4.3.8 编译器提示错误信息

4.3.2 修改 bsp.c 和 bsp.h 文件

我们打开 bsp.c 文件，里面有很多的代码我们只需要其中的很少一部分关于 DWT 的代码，因此我们要做相应的修改，在修改之前我们稍微讲解一下 Cortex-M3/M4 的跟踪组件。

在 CM3/CM4 中有 3 种跟踪源：ETM、ITM 和 DWT，要想使用 ETM、ITM 和 DWT 的话要将 DEMCR 寄存器的 TRCENA 位(bit24)置 1，DEMCR 寄存器地址为：0XE000EDFC，在我们的光盘中 STM32 参考资料下的《Cortex-M3 与 M4 权威指南》(英文名为《The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, 3rd Edition》)的 501 页有详细的讲解，感兴趣的朋友可以自行查阅一下。在 DWT 组件中有一个 CYCCNT 寄存器，这个寄存器用来对时钟周期计数，我们可以使用这个寄存器来测量执行某个任务所花费的时间。

DWT 组件有多个寄存器，我们这里只使用 DWT 的控制寄存器 CTRL、CYCCNT 寄存器，CTRL 寄存器地址为 0XE0001000、CYCCNT 寄存器地址为 0XE0001004。如果我们要使用时钟计数功能需要将 CTRL 寄存器的 bit0 置 1。

这里大家可以直接使用“例 4-1 UCOSIII 移植”实验中的 bsp.c 文件，修改后的 bsp.c 文件如下，这里为了方便讲解我们删掉了一些函数的英文注释，改为中文注释，在我们提供的例程中并没有添加这些函数的中文注释。

```
#define     BSP_MODULE
```

```
#include   <bsp.h>

#define   BSP_REG_DEM_CR          (*(CPU_REG32 *)0xE000EDFC)  //DEMCR 寄存器
#define   BSP_REG_DWT_CR          (*(CPU_REG32 *)0xE0001000)   //DWT 控制寄存器
#define   BSP_REG_DWT_CYCCNT (*(CPU_REG32 *)0xE0001004)   //DWT  时钟计数寄存器

#define   BSP_REG_DBGMCU_CR   (*(CPU_REG32 *)0xE0042004)

//DEMCR 寄存器的第 24 位,如果要使用 DWT ETM ITM 和 TPIU 的话 DEMCR 寄存器的第 24
//位置 1
#define   BSP_BIT_DEM_CR_TRCENA           DEF_BIT_24
//DWTCR 寄存器的第 0 位,当为 1 的时候使能 CYCCNT 计数器,使用 CYCCNT 之前应当先初始
//化
#define   BSP_BIT_DWT_CR_CYCCNTENA         DEF_BIT_00

//获取系统时钟
//返回值：系统时钟 HCLK
CPU_INT32U   BSP_CPU_ClkFreq (void)
{
    RCC_ClocksTypeDef   rcc_clocks;
    RCC_GetClocksFreq(&rcc_clocks);          //获取各个时钟频率
    return ((CPU_INT32U)rcc_clocks.HCLK_Frequency);   //返回 HCLK 时钟频率
}

//此函数用来开启和初始化 CPU 的时间戳定时器，其实就是使能 DWT 和 CYCCNT，并且初始
//化 CYCCNT 为 0。
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
void   CPU_TS_TmrInit (void)
{
    CPU_INT32U   fclk_freq;
    fclk_freq = BSP_CPU_ClkFreq();
    BSP_REG_DEM_CR       |= (CPU_INT32U)BSP_BIT_DEM_CR_TRCENA; //使能 DWT
    BSP_REG_DWT_CYCCNT   = (CPU_INT32U)0u;      //初始化 CYCCNT 寄存器
    BSP_REG_DWT_CR |= (CPU_INT32U)BSP_BIT_DWT_CR_CYCCNTENA;//开启 CYCCNT
    CPU_TS_TmrFreqSet((CPU_TS_TMR_FREQ)fclk_freq);
}
#endif

//此函数其实就是获取 CYCCNT 中的值
#if (CPU_CFG_TS_TMR_EN == DEF_ENABLED)
CPU_TS_TMR   CPU_TS_TmrRd (void)
{
    CPU_TS_TMR   ts_tmr_cnts;
```

```
    ts_tmr_cnts = (CPU_TS_TMR)BSP_REG_DWT_CYCCNT;
    return (ts_tmr_cnts);
}
#endif

//下面这两个函数：CPU_TS32_to_uSec()和 CPU_TS64_to_uSec()是用来将读取到的时钟周期数
//转换为 us。
#if (CPU_CFG_TS_32_EN == DEF_ENABLED)
CPU_INT64U  CPU_TS32_to_uSec (CPU_TS32  ts_cnts)
{
    CPU_INT64U  ts_us;
    CPU_INT64U  fclk_freq;
    fclk_freq = BSP_CPU_ClkFreq();
    ts_us     = ts_cnts / (fclk_freq / DEF_TIME_NBR_uS_PER_SEC);
    return (ts_us);
}
#endif

#if (CPU_CFG_TS_64_EN == DEF_ENABLED)
CPU_INT64U  CPU_TS64_to_uSec (CPU_TS64  ts_cnts)
{
    CPU_INT64U  ts_us;
    CPU_INT64U  fclk_freq;
    fclk_freq = BSP_CPU_ClkFreq();
    ts_us     = ts_cnts / (fclk_freq / DEF_TIME_NBR_uS_PER_SEC);
    return (ts_us);
}
#endif
```

我们还要对 bsp.h 文件做相应的修改，修改后的 bsp.h 的文件如下所示，bsp.h 文件很简单，就是添加一些头文件。

```
#ifndef  BSP_PRESENT
#define  BSP_PRESENT

#ifdef   BSP_MODULE
#define  BSP_EXT
#else
#define  BSP_EXT  extern
#endif

#include  <stdio.h>
#include  <stdarg.h>
#include  <cpu.h>
#include  <cpu_core.h>
```

```
#include   <lib_def.h>
#include   <lib_ascii.h>
#include   <stm32f4xx_conf.h>

#endif
```

4.3.3 修改 os_cpu_a.asm 文件

os_cpu_a.asm 文件和我们第一章 UCOSII 移植中的 os_cpu_a.asm 文件基本一致，大家根据“例 1-1 UCOSII 移植”实验中的 os_cpu_a.asm 文件来修改自己工程中的 os_cpu_a.asm 文件，我们在第三章中对于这个文件已经做了非常详细的讲解，因此这里就不再讲解了。

修改完 os_cpu_a.asm 文件后我们编译一下工程提示如图 4.3.9 所示错误，提示我们在 os_cpu_c.c 使用到的 OS_CPU_FP_Reg_Pop()和 OS_CPU_FP_Reg_Push()这两个函数未定义，这个错误我们会在下一小节讲解 os_cpu_c.c 文件修改的时候讲到。

```
linking...
..\OBJ\Template.axf: Error: L6218E: Undefined symbol OS_CPU_FP_Reg_Pop (referred from os_cpu_c.o).
..\OBJ\Template.axf: Error: L6218E: Undefined symbol OS_CPU_FP_Reg_Push (referred from os_cpu_c.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 2 error messages.
"..\OBJ\Template.axf" - 2 Error(s), 0 Warning(s).
Target not created
```

图 4.3.8 编译提示错误

4.3.4 修改 os_cpu_c.c 文件

首先我们在 os_cpu_c.c 文件开始部分添加 includes.h 头文件，如图 4.3.9 所示。

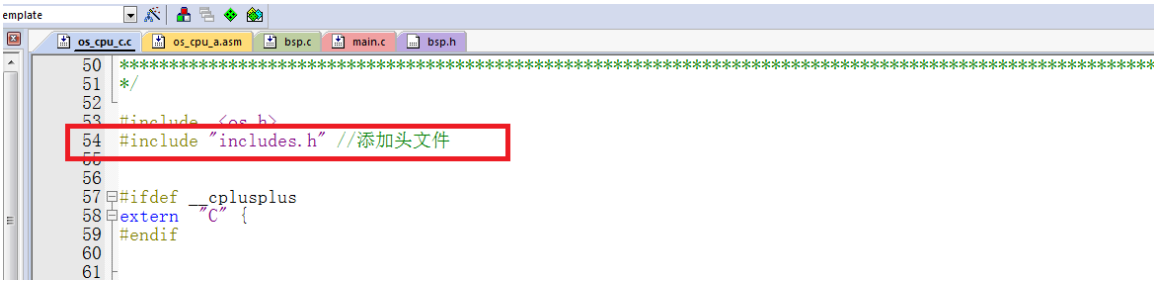


图 4.3.9 添加 includes.h 头文件

在上一小节中我们知道修改完 os_cpu_a.asm 文件后有两个错误，我们需要修改 os_cpu_c.c 文件来消除这两个错误。我们找到在 os_cpu_c.c 文件中的 OSTaskSwHook()函数使用到这两个函数，这两个函数分别用来对 FPU 寄存器进行出栈和入栈处理的，我们对于 FPU 寄存器的入栈和出栈有相应的处理，这里不使用这两个函数，因此把 OSTaskSwHook()函数中关于 OS_CPU_FP_Reg_Pop()和 OS_CPU_FP_Reg_Push()这两个函数的代码注销掉，如图 4.3.10 所示。

```
321 void  OSTaskSwHook (void)
322 {
323 #if OS_CFG_TASK_PROFILE_EN > 0u
324     CPU_TS  ts;
325 #endif
326 #ifdef  CPU_CFG_INT_DIS_MEAS_EN
327     CPU_TS  int_dis_time;
328 #endif
329
330
331 #if (OS_CPU_ARM_FP_EN == DEF_ENABLED)
332 //     if ((OSTCBCurPtr->Opt & OS_OPT_TASK_SAVE_FP) != (OS_OPT)0) {
333 //         OS_CPU_FP_Reg_Push(OSTCBCurPtr->StkPtr);
334 //     }
335
336 //     if ((OSTCBHighRdyPtr->Opt & OS_OPT_TASK_SAVE_FP) != (OS_OPT)0) {
337 //         OS_CPU_FP_Reg_Pop(OSTCBHighRdyPtr->StkPtr);
338 //     }
339 #endif
340
341 #if OS_CFG_APP_HOOKS_EN > 0u
342     if (OS_AppTaskSwHookPtr != (OS_APP_HOOK_VOID)0) {
343         (*OS_AppTaskSwHookPtr)();
344     }
345 #endif
346
```

如图 4.3.10 注销掉相应函数调用的代码

注销掉 OSTaskSwHook()函数中调用的 OS_CPU_FP_Reg_Pop()和 OS_CPU_FP_Reg_Push()这两个函数后再次编译一下工程，应该就没有错误了。

同移植 UCOSII 一样，我们还需要修改堆栈初始化函数 OSTaskStkInit()，修改后的 OSTaskStkInit()函数如下所示。这个函数基本上和我们移植 UCOSII 时 os_cpu_c.c 文件中的 OSTaskStkInit()函数一样，具体含意参考我们第三章关于 UCOSII 中 os_cpu_c.c 文件的讲解。

```
CPU_STK   *OSTaskStkInit (OS_TASK_PTR     p_task,
                          void            *p_arg,
                          CPU_STK         *p_stk_base,
                          CPU_STK         *p_stk_limit,
                          CPU_STK_SIZE    stk_size,
                          OS_OPT          opt)
{
    CPU_STK     *p_stk;
    (void)opt;
    p_stk = &p_stk_base[stk_size];
    p_stk = (CPU_STK *)((CPU_STK)(p_stk) & 0xFFFFFFF8);     //堆栈 8 字节对齐

 #if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *(--p_stk) = (CPU_STK)0x00000000u;
    *(--p_stk) = (CPU_STK)0x00001000u; //FPSCR
    *(--p_stk) = (CPU_STK)0x00000015u; //s15
    *(--p_stk) = (CPU_STK)0x00000014u; //s14
    *(--p_stk) = (CPU_STK)0x00000013u; //s13
    *(--p_stk) = (CPU_STK)0x00000012u; //s12
    *(--p_stk) = (CPU_STK)0x00000011u; //s11
```

```
    *(--p_stk) = (CPU_STK)0x00000010u; //s10
    *(--p_stk) = (CPU_STK)0x00000009u; //s9
    *(--p_stk) = (CPU_STK)0x00000008u; //s8
    *(--p_stk) = (CPU_STK)0x00000007u; //s7
    *(--p_stk) = (CPU_STK)0x00000006u; //s6
    *(--p_stk) = (CPU_STK)0x00000005u; //s5
    *(--p_stk) = (CPU_STK)0x00000004u; //s4
    *(--p_stk) = (CPU_STK)0x00000003u; //s3
    *(--p_stk) = (CPU_STK)0x00000002u; //s2
    *(--p_stk) = (CPU_STK)0x00000001u; //s1
    *(--p_stk) = (CPU_STK)0x00000000u; //s0
#endif

    *(--p_stk) = (CPU_STK)0x01000000u;      //xPSR
    *(--p_stk) = (CPU_STK)p_task;           // Entry Point
    *(--p_stk) = (CPU_STK)OS_TaskReturn;    //R14 (LR)
    *(--p_stk) = (CPU_STK)0x12121212u;      // R12
    *(--p_stk) = (CPU_STK)0x03030303u;      // R3
    *(--p_stk) = (CPU_STK)0x02020202u;      // R2
    *(--p_stk) = (CPU_STK)p_stk_limit;      // R1
    *(--p_stk) = (CPU_STK)p_arg;            //R0

#if (__FPU_PRESENT==1)&&(__FPU_USED==1)
    *(--p_stk) = (CPU_STK)0x00000031u; //s31
    *(--p_stk) = (CPU_STK)0x00000030u; //s30
    *(--p_stk) = (CPU_STK)0x00000029u; //s29
    *(--p_stk) = (CPU_STK)0x00000028u; //s28
    *(--p_stk) = (CPU_STK)0x00000027u; //s27
    *(--p_stk) = (CPU_STK)0x00000026u; //s26
    *(--p_stk) = (CPU_STK)0x00000025u; //s25
    *(--p_stk) = (CPU_STK)0x00000024u; //s24
    *(--p_stk) = (CPU_STK)0x00000023u; //s23
    *(--p_stk) = (CPU_STK)0x00000022u; //s22
    *(--p_stk) = (CPU_STK)0x00000021u; //s21
    *(--p_stk) = (CPU_STK)0x00000020u; //s20
    *(--p_stk) = (CPU_STK)0x00000019u; //s19
    *(--p_stk) = (CPU_STK)0x00000018u; //s18
    *(--p_stk) = (CPU_STK)0x00000017u; //s17
    *(--p_stk) = (CPU_STK)0x00000016u; //s16
#endif
    *(--p_stk) = (CPU_STK)0x11111111u;      //R11
    *(--p_stk) = (CPU_STK)0x10101010u;      //R10
    *(--p_stk) = (CPU_STK)0x09090909u;      //R9
```

```
    *(--p_stk) = (CPU_STK)0x08080808u;      // R8
    *(--p_stk) = (CPU_STK)0x07070707u;      //R7
    *(--p_stk) = (CPU_STK)0x06060606u;      //R6
    *(--p_stk) = (CPU_STK)0x05050505u;      // R5
    *(--p_stk) = (CPU_STK)0x04040404u;      //R4
    return (p_stk);
}
```

4.3.5 修改 os_cfg_app.h

我们还需要修改 os_cfg_app.h 文件，os_cfg_app.h 主要是对 UCOSIII 内部一些系统任务的配置，如任务优先级、任务堆栈、UCOSIII 的系统时钟节拍等等，os_cfg_app.h 文件都是一些宏定义，比较简单，文件代码如下。

```
#ifndef OS_CFG_APP_H
#define OS_CFG_APP_H

#define   OS_CFG_MSG_POOL_SIZE          100u      //消息数量
#define   OS_CFG_ISR_STK_SIZE           128u      //中断任务堆栈大小
//任务堆栈深度，比如当定义为 10 时表示当任务堆栈剩余百分之 10%时就说明堆栈为空
#define   OS_CFG_TASK_STK_LIMIT_PCT_EMPTY   10u

//**********************空闲任务**************************
#define   OS_CFG_IDLE_TASK_STK_SIZE     128u      //空任务堆栈大小

//**********************中断服务管理任务******************
#define   OS_CFG_INT_Q_SIZE             10u       //中断队列大小
#define   OS_CFG_INT_Q_TASK_STK_SIZE    128u      //中断服务管理任务大小

//**********************统计任务**************************
//统计任务优先级，倒数第二个优先级，最后一个优先级是空闲任务的。
#define   OS_CFG_STAT_TASK_PRIO      (OS_CFG_PRIO_MAX-2u)
// OS_CFG_STAT_TASK_RATE_HZ 用于统计任务计算 CPU 使用率，统计任务通过统计在
//(1/ OS_CFG_STAT_TASK_RATE_HZ)秒的时间内 OSStatTaskCtr 能够达到的最大值来得到
//CPU 使用率，OS_CFG_STAT_TASK_RATE_HZ 值应该在 1~10Hz
#define   OS_CFG_STAT_TASK_RATE_HZ      10u
#define   OS_CFG_STAT_TASK_STK_SIZE     128u      //统计任务堆栈

//********************时钟节拍任务************************
#define   OS_CFG_TICK_RATE_HZ           200u      //系统时钟节拍频率
//时钟节拍任务，一般设置一个相对较高的优先级
#define   OS_CFG_TICK_TASK_PRIO         1u
#define   OS_CFG_TICK_TASK_STK_SIZE     128u      //时钟节拍任务堆栈大小
#define   OS_CFG_TICK_WHEEL_SIZE        17u       //时钟节拍列表大小
```

```
//************************定时任务************************
#define   OS_CFG_TMR_TASK_PRIO          2u      //定时任务优先级
#define   OS_CFG_TMR_TASK_RATE_HZ       100u    //定时频率，一般为 100Hz
#define   OS_CFG_TMR_TASK_STK_SIZE      128u    //定时任务堆栈大小
#define   OS_CFG_TMR_WHEEL_SIZE         17u     //定时器列表数量
#endif
```

在 UCOSIII 中有五个系统任务：空闲任务、时钟节拍任务、统计任务、定时任务和中断服务管理任务，在系统初始化的时候至少要创建两个任务：空闲任务和时钟节拍任务。空闲任务的优先级应该为最低 OS_CFG_PRIO_MAX-1，如果使用中断服务管理任务的话那么中断服务管理任务的优先级应该为最高 0。其他 3 个任务的优先级用户可以自行设置，本手册中我们将统计任务设置为 OS_CFG_PRIO_MAX-2，既倒数第二个优先级；时钟节拍任务也需要一个高优先级,我们将优先级 1 分配给时钟节拍任务；将优先级 2 分配给定时器任务。所以优先级 0、1、2、OS_CFG_PRIO_MAX-2 和 OS_CFG_PRIO_MAX-1 这 5 个优先级用户应用程序是不能使用的!

4.3.6 修改 SYSTEM 文件夹

1) 修改 sys.h 文件

首先我们要修改 sys.h 文件中宏定义 SYSTEM_SUPPORT_OS，我们将其定义为 1，定义系统文件夹支持 OS，如下所示。

```
//0,不支持 os
//1,支持 os
#define SYSTEM_SUPPORT_OS       1           //定义系统文件夹支持 OS
```

4.4 软件设计

移植完成后就需要编写测试软件测试我们移植是否正确，我们建立 3 个任务，其中两个任务分别用于 LED0 和 LED1 闪烁，另外一个任务用于测试浮点计算，软件代码如下。

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "led.h"
#include "includes.h"

//任务优先级
#define START_TASK_PRIO      3
//任务堆栈大小
#define START_STK_SIZE       512
//任务控制块
OS_TCB StartTaskTCB;
//任务堆栈
CPU_STK START_TASK_STK[START_STK_SIZE];
//任务函数
void start_task(void *p_arg);

//任务优先级
```

```
#define LED0_TASK_PRIO      4
//任务堆栈大小
#define LED0_STK_SIZE       128
//任务控制块
OS_TCB Led0TaskTCB;
//任务堆栈
CPU_STK LED0_TASK_STK[LED0_STK_SIZE];
void led0_task(void *p_arg);

//任务优先级
#define LED1_TASK_PRIO      5
//任务堆栈大小
#define LED1_STK_SIZE       128
//任务控制块
OS_TCB Led1TaskTCB;
//任务堆栈
CPU_STK LED1_TASK_STK[LED1_STK_SIZE];
//任务函数
void led1_task(void *p_arg);

//任务优先级
#define FLOAT_TASK_PRIO     6
//任务堆栈大小
#define FLOAT_STK_SIZE      128
//任务控制块
OS_TCB FloatTaskTCB;
//任务堆栈
__align(8) CPU_STK    FLOAT_TASK_STK[FLOAT_STK_SIZE];
//任务函数
void float_task(void *p_arg);

int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);            //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);          //串口初始化
    INTX_DISABLE();             //关中断,防止滴答定时器对外设初始化的打扰
    LED_Init();                 //LED 初始化
    INTX_ENABLE();              //开中断
```

```
    OSInit(&err);       //初始化 UCOSIII
    OS_CRITICAL_ENTER();//进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB*          )&StartTaskTCB,         //任务控制块
                 (CPU_CHAR*        )"start task",  //任务名字
                 (OS_TASK_PTR      )start_task,          //任务函数
                 (void*            )0,                   //传递给任务函数的参数
                 (OS_PRIO          )START_TASK_PRIO,       //任务优先级
                 (CPU_STK*         )&START_TASK_STK[0],  //任务堆栈基地址
                 (CPU_STK_SIZE     )START_STK_SIZE/10,     //任务堆栈深度限位
                 (CPU_STK_SIZE     )START_STK_SIZE,        //任务堆栈大小
                 (OS_MSG_QTY       )0,                     //任务内部消息队列能够接收
                                                           //的最大消息数目,为 0 时禁止
                                                           //接收消息
                 (OS_TICK          )0,                     //当使能时间片轮转时的时间
                                                           //片长度,为 0 时为默认长度。
                 (void*            )0,                     //用户补充的存储区
                 (OS_OPT           )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*          )&err);                 //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);                //开启 UCOSIII
    while(1);
}

//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
   OSStatTaskCPUUsageInit(&err);    //统计任务
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN          //如果使能了测量中断关闭时间
    CPU_IntDisMeasMaxCurReset();
#endif

#if   OS_CFG_SCHED_ROUND_ROBIN_EN  //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
```

```
#endif

    OS_CRITICAL_ENTER();  //进入临界区
    //创建 LED0 任务
    OSTaskCreate((OS_TCB*          )&Led0TaskTCB,
                 (CPU_CHAR*        )"led0 task",
                 (OS_TASK_PTR      )led0_task,
                 (void*            )0,
                 (OS_PRIO          )LED0_TASK_PRIO,
                 (CPU_STK*         )&LED0_TASK_STK[0],
                 (CPU_STK_SIZE     )LED0_STK_SIZE/10,
                 (CPU_STK_SIZE     )LED0_STK_SIZE,
                 (OS_MSG_QTY       )0,
                 (OS_TICK          )0,
                 (void*            )0,
                 (OS_OPT           )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*          )&err);
    //创建 LED1 任务
    OSTaskCreate((OS_TCB*          )&Led1TaskTCB,
                 (CPU_CHAR*        )"led1 task",
                 (OS_TASK_PTR      )led1_task,
                 (void*            )0,
                 (OS_PRIO          )LED1_TASK_PRIO,
                 (CPU_STK*         )&LED1_TASK_STK[0],
                 (CPU_STK_SIZE     )LED1_STK_SIZE/10,
                 (CPU_STK_SIZE     )LED1_STK_SIZE,
                 (OS_MSG_QTY       )0,
                 (OS_TICK          )0,
                 (void*            )0,
                 (OS_OPT           )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*          )&err);
    //创建浮点测试任务
    OSTaskCreate((OS_TCB*          )&FloatTaskTCB,
                 (CPU_CHAR*        )"float test task",
                 (OS_TASK_PTR      )float_task,
                 (void*            )0,
                 (OS_PRIO          )FLOAT_TASK_PRIO,
                 (CPU_STK*         )&FLOAT_TASK_STK[0],
                 (CPU_STK_SIZE     )FLOAT_STK_SIZE/10,
                 (CPU_STK_SIZE     )FLOAT_STK_SIZE,
                 (OS_MSG_QTY       )0,
                 (OS_TICK          )0,
                 (void*            )0,
```

```
                    (OS_OPT         )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                    (OS_ERR*        )&err);
    OS_TaskSuspend((OS_TCB*)&StartTaskTCB,&err);        //挂起开始任务
    OS_CRITICAL_EXIT();     //进入临界区
}

//led0 任务函数
void led0_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED0=0;
        OSTimeDlyHMSM(0,0,0,200,OS_OPT_TIME_HMSM_STRICT,&err); //延时 200ms
        LED0=1;
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

//led1 任务函数
void led1_task(void *p_arg)
{
    OS_ERR err;
    p_arg = p_arg;
    while(1)
    {
        LED1=~LED1;
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_HMSM_STRICT,&err); //延时 500ms
    }
}

//浮点测试任务
void float_task(void *p_arg)
{
    CPU_SR_ALLOC();
    static float float_num=0.01;
    while(1)
    {
        float_num+=0.01f;
        OS_CRITICAL_ENTER();   //进入临界区
        printf("float_num 的值为: %.4f\r\n",float_num);
        OS_CRITICAL_EXIT();     //退出临界区
```

```
        delay_ms(500);                  //延时 500ms
    }
}
```

测试代码非常简单，没什么要说的，关于 UCOSIII 的具体使用方法在以后的章节会讲到，led0_task 任务用于让 LED0 闪烁，led1_task 用于让 LED1 闪烁，float_task 任务用于测试浮点计算，用来验证 UCOSIII 的 FPU 是否移植成功，和我们第一章移植 UCOSII 时的软件设计基本一致的。

4.5 下载验证

代码编译完成后我们就可以下载到 STM32F407 开发板中，下载进去以后我们可以看到 LED0 开始闪烁，灭的时间比亮的时间长，因为我们设置的是亮 200ms 灭 500ms；LED1 均匀闪烁，我们设置亮 500ms 灭 500ms。我们打开串口调试助手，串口调试助手接收到数据，如图 4.5.1 所示。

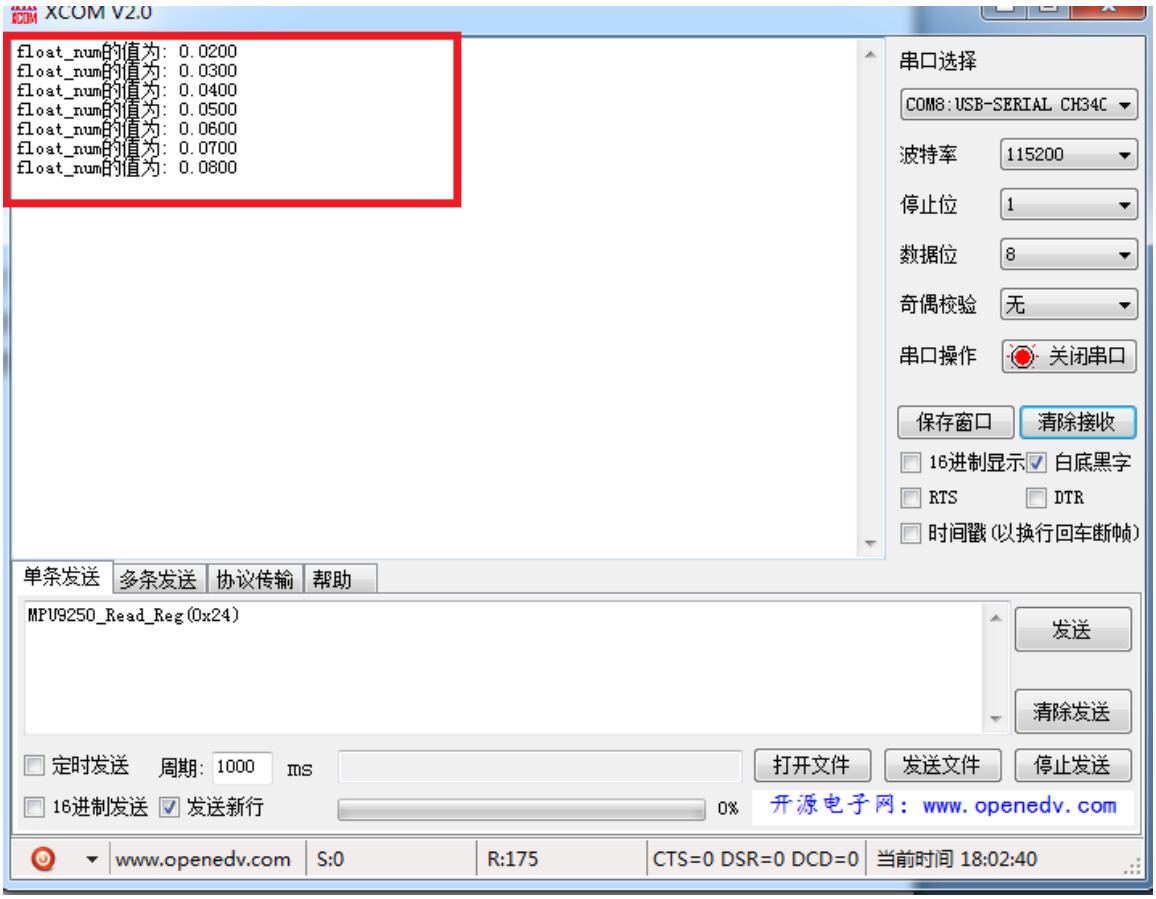


图 4.5.1 串口提示助手输出数据

从图 4.5.1 中可以看出串口调试助手接收到数据，float_num 的值在增加，每次加 0.01。这和我们在程序中设置的每次增加 0.01 相符，由此可见移植的 UCOSIII 支持 FPU，UCOSIII 移植成功。

第五章 UCOSIII 任务管理

多任务操作系统最主要的就是对任务的管理，包括任务的创建、挂起、删除和调度等，因此对于 UCOSIII 操作系统中任务管理的理解就显得尤为重要。本章我们就讲解 UCOSIII 中的任务管理，本章分为如下几个部分：

5.1 UCOSIII 启动和初始化
5.2 任务状态
5.3 任务控制块
5.4 任务堆栈
5.5 任务就绪表
5.3 任务调度和切换

5.1 UCOSIII 启动和初始化

在使用 UCOSIII 的时候我们要按照一定的顺序初始化并打开 UCOSIII，我们可以按照下面的顺序：

1、 最先肯定是要调用 OSInit()初始化 UCOSIII。

2、创建任务，一般我们在 main()函数中只创建一个 start_task 任务，其他任务都在 start_task 任务中创建，在调用 OSTaskCreate()函数创建任务的时候一定要调用 OS_CRITICAL_ENTER()函数进入临界区，任务创建完以后调用 OS_CRITICAL_EXIT()函数退出临界区。

3、最后调用 OSStart()函数开启 UCOSIII。

我们打开“例 4-1 UCOSIII 移植”实验工程的 main()函数，代码如下：

```
int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);            //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);          //串口初始化
    LED_Init();                 //LED 初始化

    OSInit(&err);               //初始化 UCOSIII
    OS_CRITICAL_ENTER();        //进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB*       )&StartTaskTCB,     //任务控制块
                 (CPU_CHAR*     )"start task",      //任务名字
                 (OS_TASK_PTR   )start_task,        //任务函数
                 (void*         )0,                 //传递给任务函数的参数
                 (OS_PRIO       )START_TASK_PRIO,       //任务优先级
                 (CPU_STK*      )&START_TASK_STK[0],    //任务堆栈基地址
                 (CPU_STK_SIZE  )START_STK_SIZE/10,     //任务堆栈深度限位
                 (CPU_STK_SIZE  )START_STK_SIZE,        //任务堆栈大小
                 (OS_MSG_QTY    )0,                     //任务内部消息队列能够接收
                                                        //的最大消息数目,为 0 时禁止
                                                        //接收消息
                 (OS_TICK       )0,                     //当使能时间片轮转时的时间
                                                        //片长度，为 0 时为默认长度
                 (void*         )0,                     //用户补充的存储区
                 (OS_OPT        )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*       )&err);                 //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);          //开启 UCOSIII
    while(1);
}
```

从上面代码可以看出我们就是按照前面提到的步骤来使用 UCOSIII，首先是 OSInit()初始

化 UCOSIII，然后创建一个 start_task()任务，最后调用 OSStart()函数开启 UCOSIII。

注意：我们在调用 OSStart()开启 UCOSIII 之前一定要至少创建一个任务，其实我们在调用 OSInit()函数初始化 UCOSIII 的时候已经创建了一个空闲任务。

5.2 任务状态

UCOSIII 支持的是单核 CPU，不支持多核 CPU，这样在某一时刻只有一个任务会获得 CPU 使用权进入运行态，其他的任务就会进入其他状态，UCOSIII 中的任务有多个状态，如下表 5.2.1 所示。

任务状态	描述
休眠态	休眠态就是任务只是以任务函数的方式存在，只是存储区中的一段代码，并未用 OSTaskCreate()函数创建这个任务，不受 UCOSIII 管理的。
就绪态	任务在就绪表中已经登记，等待获取 CPU 使用权。
运行态	正在运行的任务就处于运行态。
等待态	正在运行的任务需要等待某一个事件，比如信号量、消息、事件标志组等，就会暂时让出 CPU 使用权，进入等待事件状态。
中断服务态	一个正在执行的任务被中断打断，CPU 转而执行中断服务程序，这时这个任务就会被挂起，进入中断服务态。

在 UCOSIII 中任务可以在这 5 个状态中转换，转换关系如图 5.2.1 所示。

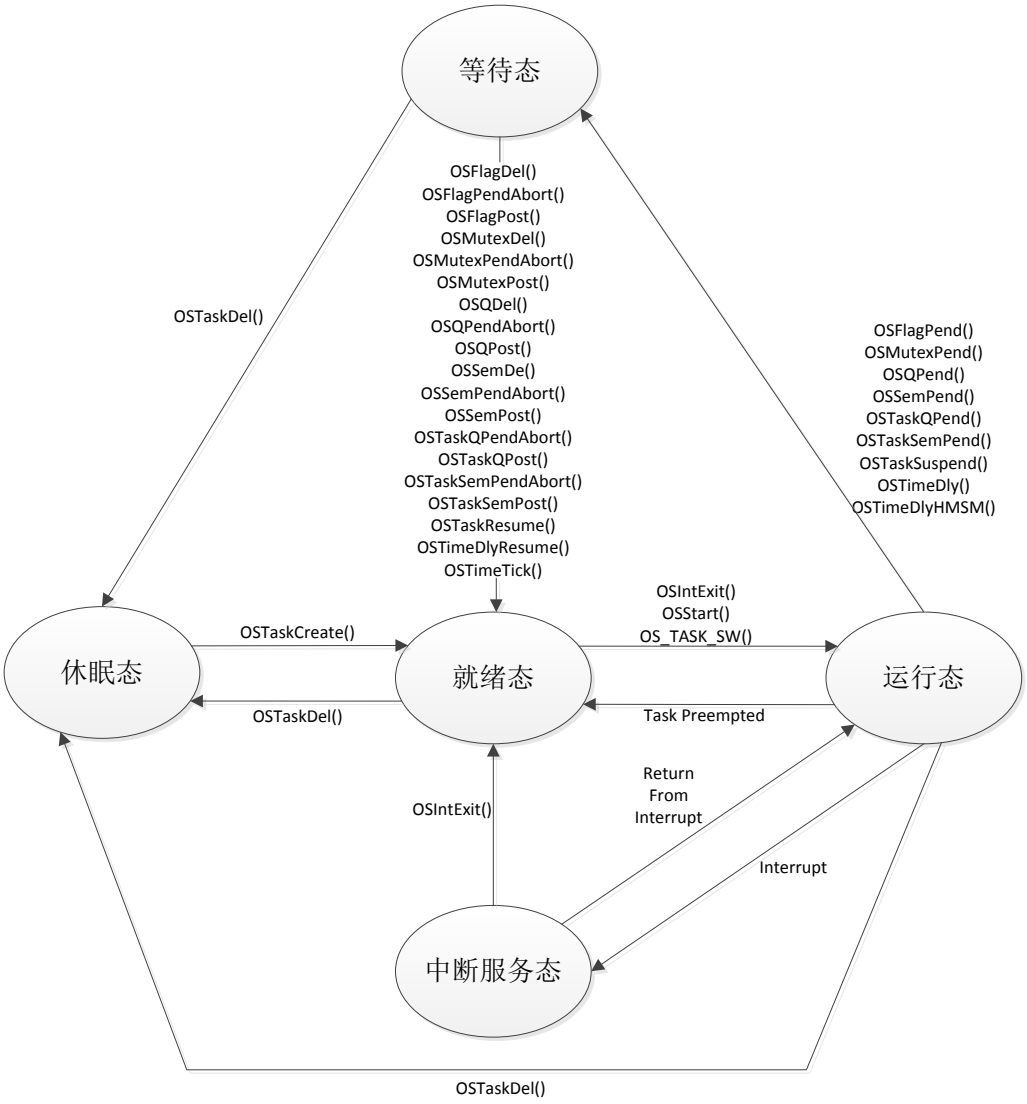


图 5.2.1 UCOSIII 任务状态转换图

5.3 任务控制块

在学习 UCSOII 的时候我们知道有个重要的数据结构：任务控制块 OS_TCB，在 UCOSIII 中也有任务控制块 OS_TCB。任务控制块 TCB 用来保存任务的信息，我们使用 OSTaskCreate()函数来创建任务的时候就会给任务分配一个任务控制块。任务控制块是一个结构体，这个结构体如下，这里我们取掉了条件编译语句。

```
struct os_tcb {
    CPU_STK         *StkPtr;      //指向当前任务堆栈的栈顶
    void            *ExtPtr;      //指向用户可定义的数据区
    CPU_STK         *StkLimitPtr; //可指向任务堆栈中的某个位置
    OS_TCB          *NextPtr;     //NexPtr 和 PrevPtr 用于在任务就绪表建立 OS_TCB
    OS_TCB          *PrevPtr;     //双向链表
    OS_TCB          *TickNextPtr; // TickNextPtr 和 TickPrevPtr 可把正在延时或在指定
时
    OS_TCB          *TickPrevPtr; //间内等待某个事件的任务的 OS_TCB 构成双向链表
```

```
    OS_TICK_SPOKE       *TickSpokePtr; //通过该指针可知道该任务在时钟节拍轮的那个
                                       //spoke 上
    CPU_CHAR            *NamePtr;      //任务名
    CPU_STK             *StkBasePtr;   //任务堆栈基地址
    OS_TASK_PTR         TaskEntryAddr;//任务代码入口地址
    void                *TaskEntryArg; //传递给任务的参数
    OS_PEND_DATA        *PendDataTblPtr;//指向一个表，包含有任务等待的所有事件对象的
                                        //信息
    OS_STATE            PendOn;         //任务正在等待的事件的类型
    OS_STATUS           PendStatus;     //任务等待的结果
    OS_STATE            TaskState;      //任务的当前状态
    OS_PRIO             Prio;           //任务优先级
    CPU_STK_SIZE        StkSize;        //任务堆栈大小
    OS_OPT              Opt;            //保存调用 OSTaskCreat()创建任务时的可选参数
                                        //options 的值
    OS_OBJ_QTY          PendDataTblEntries; //任务同时等待的事件对象的数目
    CPU_TS              TS;             //存储事件发生时的时间戳
    OS_SEM_CTR          SemCtr;         //任务内建的计数型信号量的计数值
    OS_TICK             TickCtrPrev;    //存储 OSTickCtr 之前的数值
    OS_TICK             TickCtrMatch;   //任务等待延时结束时，当 TickCtrMatch 和
                                        //OSTickCtr
                                        //的数值相匹配时，任务延时结束
    OS_TICK             TickRemain;     //任务还要等待延时的节拍数
    OS_TICK             TimeQuanta;     // TimeQuanta 和 TimeQuantaCtr 与时间片有关
    OS_TICK             TimeQuantaCtr;
    void                *MsgPtr;        //指向任务接收到的消息
    OS_MSG_SIZE         MsgSize;        //任务接收到消息的长度
    OS_MSG_Q            MsgQ;           //UCOSIII 允许任务或 ISR 向任务直接发送消
息，                                        //MsgQ 就为这个消息队列
    CPU_TS              MsgQPendTime;   //记录一条消息到达所花费的时间
    CPU_TS              MsgQPendTimeMax; //记录一条消息到达所花费的最长时间
    OS_REG              RegTbl[OS_CFG_TASK_REG_TBL_SIZE]; //寄存器表，和 CPU 寄
                                                          //存器不同
    OS_FLAGS            FlagsPend;      //任务正在等待的事件的标志位
    OS_FLAGS            FlagsRdy;       //任务在等待的事件标志中有哪些已经就绪
    OS_OPT              FlagsOpt;       //任务等待事件标志组时的等待类型
    OS_NESTING_CTR      SuspendCtr;     //任务被挂起的次数
    OS_CPU_USAGE        CPUUsage;       //CPU 使用率
    OS_CPU_USAGE        CPUUsageMax;    //CPU 使用率峰值
    OS_CTX_SW_CTR       CtxSwCtr;       //任务执行的频繁程度
    CPU_TS              CyclesDelta;    //改成员被调试器或运行监视器利用
    CPU_TS              CyclesStart;    //任务已经占用 CPU 多长时间
    OS_CYCLES           CyclesTotal;    //表示一个任务总的执行时间
```

```
    OS_CYCLES           CyclesTotalPrev;
    CPU_TS              SemPendTime;          //记录信号量发送所花费的时间
    CPU_TS              SemPendTimeMax;       //记录信号量发送到一个任务所花费的最长
                                              //时间
    CPU_STK_SIZE        StkUsed;              //任务堆栈使用量
    CPU_STK_SIZE        StkFree;              //任务堆栈剩余量
    CPU_TS              IntDisTimeMax;        //该成员记录任务的最大中断关闭时间
    CPU_TS              SchedLockTimeMax;     //该成员记录锁定调度器的最长时间
    OS_TCB              *DbgPrevPtr;          //下面 3 个成语变量用于调试
    OS_TCB              *DbgNextPtr;
    CPU_CHAR            *DbgNamePtr;
};
```

从上面的 os_tcb 结构体中可以看出 UCOSIII 的任务控制块要比 UCOSII 的要复杂的多，这也间接的说明了 UCOSIII 要比 UCOSII 功能要强大得多。

5.4 任务堆栈

在 UCOSIII 中任务堆栈是一个非常重要的概念，任务堆栈用来在切换任务和调用其它函数的时候保存现场，因此每个任务都应该有自己的堆栈，我们可以按照下面的步骤创建一个堆栈：

1、定义一个 CPU_STK 变量，在 UCOSIII 中用 CPU_STK 数据类型来定义任务堆栈，CPU_STK 在 cpu.h 中有定义，其实 CPU_STK 就是 CPU_INT32U，可以看出一个 CPU_STK 变量为 4 字节，因此任务的实际堆栈大小应该为我们定义的 4 倍。下面代码就是我们定义了一个任务堆栈 TASK_STK，堆栈大小为 64*4=256 字节。

```
CPU_STK     TASK_STK[64];       //定义一个任务堆栈
```

我们可以使用下面的方法定义一个堆栈，这样代码比较清晰，我们所有例程都使用下面的方法定义堆栈。

```
#define             TASK_STK_SIZE               64  //任务堆栈大小
CPU_STK             TASK_STK[LED1_STK_SIZE];  //任务堆栈
```

我们使用 OSTaskCreat()函数创建任务的时候就可以把创建的堆栈传递给任务，如下红色字体所示将创建的堆栈传递给任务，将堆栈的基地址传递给 OSTaskCreate()函数的参数 p_stk_base，将堆栈深度传递给参数 stk_limit，堆栈深度通常为堆栈大小的十分之一，主要用来检测堆栈是否为空，将堆栈大小传递给参数 stk_size。

```
OSTaskCreate((OS_TCB*       )&StartTaskTCB,         //任务控制块
             (CPU_CHAR*     )"start task",          //任务名字
             (OS_TASK_PTR   )start_task,            //任务函数
             (void*         )0,                     //传递给任务函数的参数
             (OS_PRIO       )START_TASK_PRIO,       //任务优先级
             (CPU_STK*      )&TASK_STK[0],          //任务堆栈基地址
             (CPU_STK_SIZE  )TASK_STK_SIZE/10,      //任务堆栈深度限位
             (CPU_STK_SIZE  )TASK_STK_SIZE,         //任务堆栈大小
             (OS_MSG_QTY    )0,
             (OS_TICK       )0,
             (void*         )0,                     //用户补充的存储区
```

```
                (OS_OPT          )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                (OS_ERR*         )&err);                    //存放该函数错误时的返回值
```

创建任务的时候会初始化任务的堆栈，我们需要提前将 CPU 的寄存器保存在任务堆栈中，完成这个任务的是 OSTaskStkInit()函数，这个函数大家应该不陌生的，我们在移植 UCOSIII 的时候专门讲解过这个函数，用户不能调用这个函数，这个函数是被 OSTaskCreate()函数在创建任务的时候调用的。

5.5 任务就绪表

UCOSIII 中将已经就绪的任务放到任务就绪表里，任务就绪表有两部分：优先级位映射表 OSPrioTbl[]和就绪任务列表 OSRdyList[]。

5.5.1 优先级位映射表

当某一个任务就绪以后就会将优先级位映射表中相应的位置 1，优先级位映射表如图 5.5.1 所示，该表元素的位宽度可以是 8 位，16 位或 32 位，根据 CPU_DATA(见 cpu.h)的不同而不同，在 STM32F407 中我们定义 CPU_DATA 为 CPU_INT32U 类型的，即 32 位宽。UCOSIII 中任务数目由宏 OS_CFG_PRIO_MAX 配置的(见 os_cfg.h)。

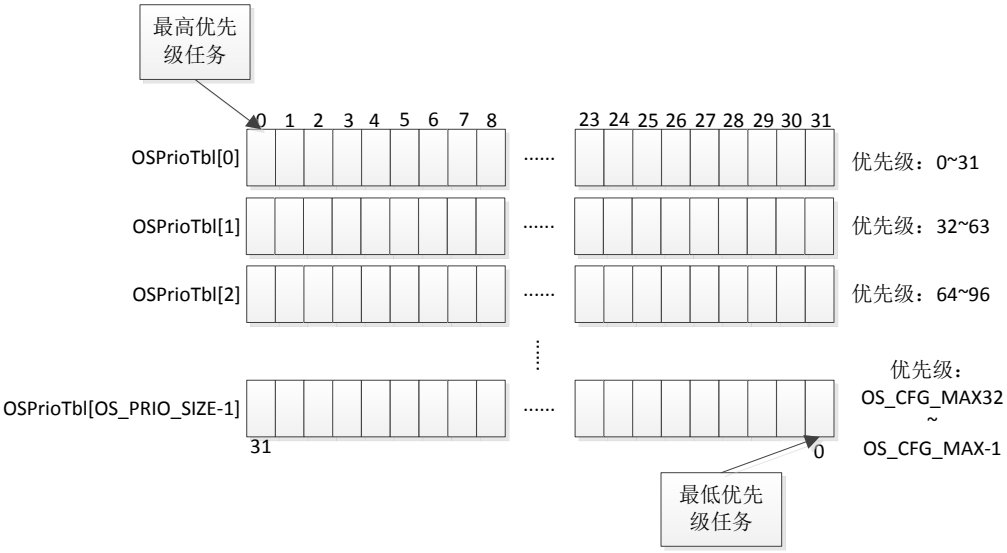


图 5.5.1 优先级位映射表

在图 5.5.1 中从左到右优先级逐渐降低，但是每个 OSPrioTbl[]数组的元素最低位在右，最高为在左边，比如 OSPrioTbl[0]的 bit31 为最高优先级 0，bit0 为优先级 31。之所以这样做主要是为了支持一条特殊的指令“计算前导零(CLZ)”,使用这条指令可以快速的找到最高优先级任务。

有关于优先级的操作有 3 个函数：OS_PrioGetHighest()、OS_PrioInsert()和 OS_PrioRemove()。分别为获取就绪表中最高优先级任务、将某个任务在就绪表中相对应的位置 1 和将某个任务在就绪表中相对应的位清零，OS_PrioGetHighest()函数代码如下：

```
OS_PRIO    OS_PrioGetHighest (void)
{
    CPU_DATA    *p_tbl;
    OS_PRIO      prio;
```

```
    prio    = (OS_PRIO)0;
    p_tbl = &OSPrioTbl[0];              //从 OSPrioTbl[0]开始扫描映射表，一直到遇到非零项
    while (*p_tbl == (CPU_DATA)0) {
//当数组 OSPrioTbl[]中的某个元素为 0 的时候 ，就继续扫描下一个数组元素，prio 加
//DEF_INT_CPU_NBR_BITS 位，根据 CPU_DATA 长度的不同 DEF_INT_CPU_NBR_BITS 值不
//同，我们定义 CPU_DATA 为 32 位，那么 DEF_INT_CPU_NBR_BITS 就为 32，prio 就加 32。
        prio += DEF_INT_CPU_NBR_BITS;
        p_tbl++; //p_tbl 加一，继续寻找 OSPrioTbl[]数组的下一个元素。
    }
    //一旦找到一个非零项，在加上该项的前导零数量就找到了最高优先级任务了。
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);
    return (prio);
}
```

从 OS_PrioGetHighest()函数可以看出，计算前导零我们使用了函数 CPU_CntLeadZeros()，这个函数是由汇编编写的，在 cpu_a.asm 文件中，代码如下：

```
CPU_CntLeadZeros
        CLZ         R0, R0                          ; 计算前导零
        BX          LR
```

函数 OS_PrioInsert()和 OS_PrioRemove()分别为将指定优先级任务相对应的优先级映射表中的位置 1 和清零，这两个函数代码如下：

```
//将参数 prio 对应的优先级映射表中的位置 1。
void   OS_PrioInsert (OS_PRIO   prio)
{
    CPU_DATA   bit;
    CPU_DATA   bit_nbr;
    OS_PRIO    ix;

    ix                  = prio / DEF_INT_CPU_NBR_BITS;
    bit_nbr             = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u);
    bit                 = 1u;
    bit                 <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;
    OSPrioTbl[ix] |= bit;
}

//将参数 prio 对应的优先级映射表中的位清零
void   OS_PrioRemove (OS_PRIO   prio)
{
    CPU_DATA   bit;
    CPU_DATA   bit_nbr;
    OS_PRIO    ix;

    ix                  = prio / DEF_INT_CPU_NBR_BITS;
    bit_nbr             = (CPU_DATA)prio & (DEF_INT_CPU_NBR_BITS - 1u);
```

```
    bit                 = 1u;
    bit                 <<= (DEF_INT_CPU_NBR_BITS - 1u) - bit_nbr;
    OSPrioTbl[ix] &= ~bit;
}
```

5.5.2 就绪任务列表

上一小节详细的讲解了优先级位映射表 OSPrioTbl[]，这个表主要是用来标记哪些任务就绪了的，这一节我们要讲的就绪任务列表 OSRdyList[]是用来记录每一个优先级下所有就绪的任务，OSRdyList[]在 os.h 中有定义，数组元素的类型为 OS_RDY_LIST，OS_RDY_LIST 为一个结构体，结构体定义如下：

```
struct   os_rdy_list {
    OS_TCB          *HeadPtr;     //用于创建链表，指向链表头
    OS_TCB          *TailPtr;     //用于创建链表，指向链表尾
    OS_OBJ_QTY      NbrEntries;   //此优先级下的任务数量
};
```

UCOSIII 支持时间片轮转调度，因此在一个优先级下会有多个任务，那么我们就要对这些任务做一个管理，这里使用 OSRdyList[]数组管理这些任务。OSRdyList[]数组中的每个元素对应一个优先级，比如 OSRdyList[0]就用来管理优先级 0 下的所有任务。OSRdyList[0]为 OS_RDY_LIST 类型，从上面 OS_RDY_LIST 结构体可以看到成员变量：HeadPtr 和 TailPtr 分别指向 OS_TCB，我们知道 OS_TCB 是可以用来构造链表的，因此同一个优先级下的所有任务是通过链表来管理的，HeadPtr 和 TailPtr 分别指向这个链表的头和尾，NbrEntries 用来记录此优先级下的任务数量，图 5.5.2 表示了优先级 4 现在有 3 个任务时候的就绪任务列表。

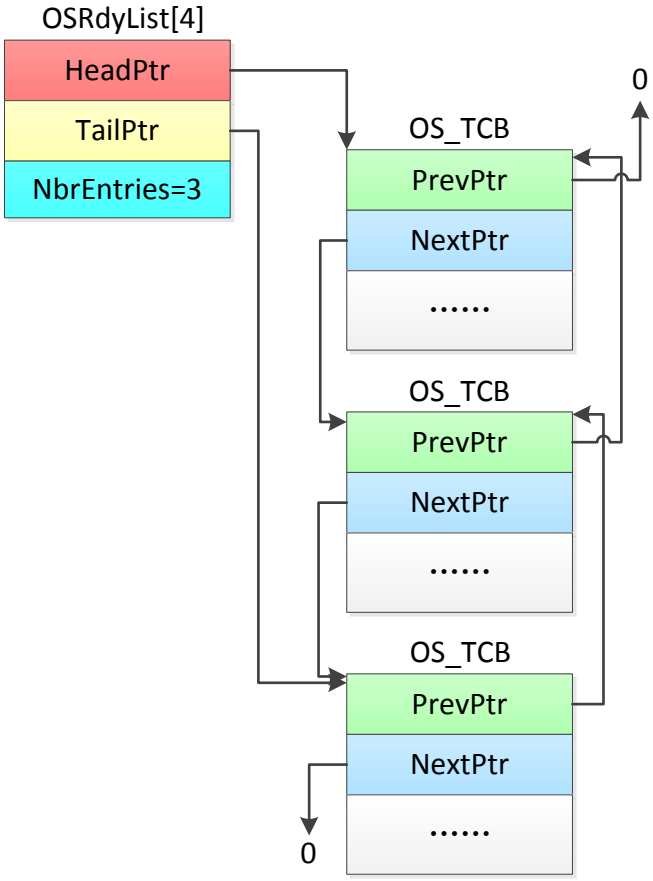


图 5.5.2 优先级 4 就绪任务列表

图 5.5.2 中展示了在优先级 4 下面有 3 个任务，这三个任务组成一个链表，OSRdyList[4]的 HeadPtr 指向链表头，TailPtr 指向链表尾，NbrEntries 为 3，表示一共有 3 个任务。注意有些优先级只能有一个任务，比如 UCOSIII 自带的 5 个系统任务：空闲任务 OS_IdleTask()、时钟节拍任务 OS_TickTask()、统计任务 OS_StatTask、定时任务 OS_TmrTask()和中断服务管理任务 OS_IntQTask()。

针对任务就绪列表的操作有以下 6 个函数，如表 5.5.1 所示，这些函数都在 os_core.c 这个文件中，这几个函数是 UCOSIII 内部使用的，用户程序不能使用。

函数	描述
OS_RdyListInit()	由 OSInit()调用用来初始化并清空任务就绪列表
OS_RdyListInsertHead()	向某一优先级下的任务双向链表头部添加一个任务控制块 TCB
OS_RdyListInsertTail()	向某一优先级下的任务双向链表尾部添加一个任务控制块 TCB
OS_RdyListRemove()	将任务控制块 TCB 从任务就绪列表中删除
OS_RdyListInsertTail()	将一个任务控制块 TCB 从双向链表的头部移到尾部
OS_RdyListInsert()	在就绪表中添加一个任务控制块 TCB

表 5.5.1 任务就绪表操作函数

5.6 任务调度和切换

5.6.1 可剥夺型调度

任务调度和切换就是让就绪表中优先级最高的任务获得 CPU 的使用权，UCOSIII 是可剥夺

型，抢占式的，可以抢了低优先级任务的 CPU 使用权，任务的调度是由一个叫做任务调度器的东西来完成的，任务调度器有两种：一种是任务级调度器，一种是中断级调度器。

1、任务级调度器

任务级调度器为 OSSched()，OSSched()函数代码在 os_core.c 文件中，如下所示：

```
void    OSSched (void)
{
    CPU_SR_ALLOC();
    // OSSched()为任务级调度器，如果是在中断服务函数中不能使用！
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                    (1)
        return;
    }
    //调度器是否上锁
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {              (2)
        return;
    }

    CPU_INT_DIS();                                                (3)
    OSPrioHighRdy     = OS_PrioGetHighest();                      (4)
    OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;           (5)
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {                         (6)
        CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
    OSTCBHighRdyPtr->CtxSwCtr++;
#endif
    OSTaskCtxSwCtr++;
#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
    OS_TLS_TaskSw();
#endif

    OS_TASK_SW();                                                 (7)
    CPU_INT_EN();                                                 (8)
}
```

(1) 检查 OSSched()函数是否在中断服务函数中调用，因为 OSSched()为任务级调度函数，因此不能用于中断级任务调度。

(2) 检查调度器是否加锁，很明显，如果任务调度器加锁了就不能做任务调度和切换的！

(3) 关中断

(4) 获取任务就绪表中就绪了的最高优先级任务，OSPrioHighRdy 用来保存当前就绪表中就绪了的最高优先级。

(5) 我们需要获取下一次任务切换是要运行的任务，因为 UCOSIII 的一个优先级下可以有多个任务，所以我们需要在这些任务中挑出任务切换后要运行的任务，在这里可以看出获取的

是就绪任务列表中的第一个任务，OSTCBHighRdyPtr 指向将要切换任务的 OS_TCB。

(6) 判断要运行的任务是否是正在运行的任务，如果是的话就不需要做任务切换，OSTCBCurPtr 指向正在执行的任务的 OS_TCB。

(7) 执行任务切换！

(8) 开中断

在 OSSched()中真正执行任务切换的是宏 OS_TASK_SW()(在 os_cpu.h 中定义)，宏 OS_TASK_SW()就是函数 OSCtxSW()，OSCtxSW()是 os_cpu_a.asm 中用汇编写的一段代码，OSCtxSW()要做的就是将当前任务的 CPU 寄存器的值保存在任务堆栈中，也就是保存现场，保存完当前任务的现场后将新任务的 OS_TCB 中保存的任务堆栈指针的值加载到 CPU 的堆栈指针寄存器中，最后还要从新任务的堆栈中恢复 CPU 寄存器的值。

2、中断级调度器

中断级调度器为 OSIntExit()，代码如下，调用 OSIntExit()时，中断应该是关闭的。

```
void    OSIntExit (void)
{
        CPU_SR_ALLOC()
    if (OSRunning != OS_STATE_OS_RUNNING) {                    (1)
        return;
    }

        CPU_INT_DIS();
    if (OSIntNestingCtr == (OS_NESTING_CTR)0) {                (2)
        CPU_INT_EN();
        return;
    }
    OSIntNestingCtr--;                                         (3)
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                 (4)
        CPU_INT_EN();
        return;
    }

    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {           (5)
    CPU_INT_EN();
        return;
    }

    OSPrioHighRdy     = OS_PrioGetHighest();                   (6)
    OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
    CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
```

```
    OSTCBHighRdyPtr->CtxSwCtr++;
    #endif
    OSTaskCtxSwCtr++;
#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
    OS_TLS_TaskSw();
#endif

    OSIntCtxSw();                                    (7)
    CPU_INT_EN();                                    (8)
}
```

(1) 判断 UCOSIII 是否运行，如果 UCOSIII 未运行的话就直接跳出。

(2) OSIntNestingCtr 为中断嵌套计数器，进入中断服务函数后我们要调用 OSIntEnter()函数，在这个函数中会将 OSIntNestingCtr 加 1，用来记录中断嵌套的次数。这里检查 OSIntNestingCtr 是否为 0，确保在退出中断服务函数时调用 OSIntExit()后不会等于负数。

(3) OSIntNestingCtr 减 1，因为 OSIntExit()是在退出中断服务函数时调用的，因此中断嵌套计数器要减 1。

(4) 如果 OSIntNestingCtr 还大于 0，说明还有其他的中断发生，那么就跳回到中断服务程序中，不需要做任务切换。

(5) 检查调度器是否加锁，如果加锁的话就直接跳出，不需要做任务切换。

(6) 接下来的 5 行程序和任务级调度器 OSSechd()是一样的，从 OSRdyList[]中取出最高优先级任务的控制块 OS_TCB。

(7) 调用中断级任务切换函数 OSIntCtxSW()。

(8) 开中断

在中断级调度器中真正完成任务切换的就是中断级任务切换函数 OSIntCtxSW()，与任务级切换函数 OSCtxSW()不同的是，由于进入中断的时候现场已经保存过了，所以 OSIntCtxSW()不需要像 OSCtxSW()一样先保存当前任务现场，只需要做 OSCtxSW()的后半部分工作，也就是从将要执行的任务堆栈中恢复 CPU 寄存器的值。

5.6.2 时间片轮转调度

前面多次提到 UCOSIII 支持多个任务同时拥有一个优先级，要使用这个功能我们需要定义 OS_CFG_SCHED_ROUND_ROBIN_EN 为 1，这些任务的调度是一个值得考虑的问题，不过这不是我们要考虑的，貌似说了一句废话。在 UCOSIII 中允许一个任务运行一段时间(时间片)后让出 CPU 的使用权，让拥有同优先级的下一个任务运行，这种任务调度方法就是时间片轮转调度。图 5.6.1 展示了运行在同一优先级下的执行时间图，在优先级 N 下有 3 个就绪的任务，我们将时间片划分为 4 个时钟节拍。

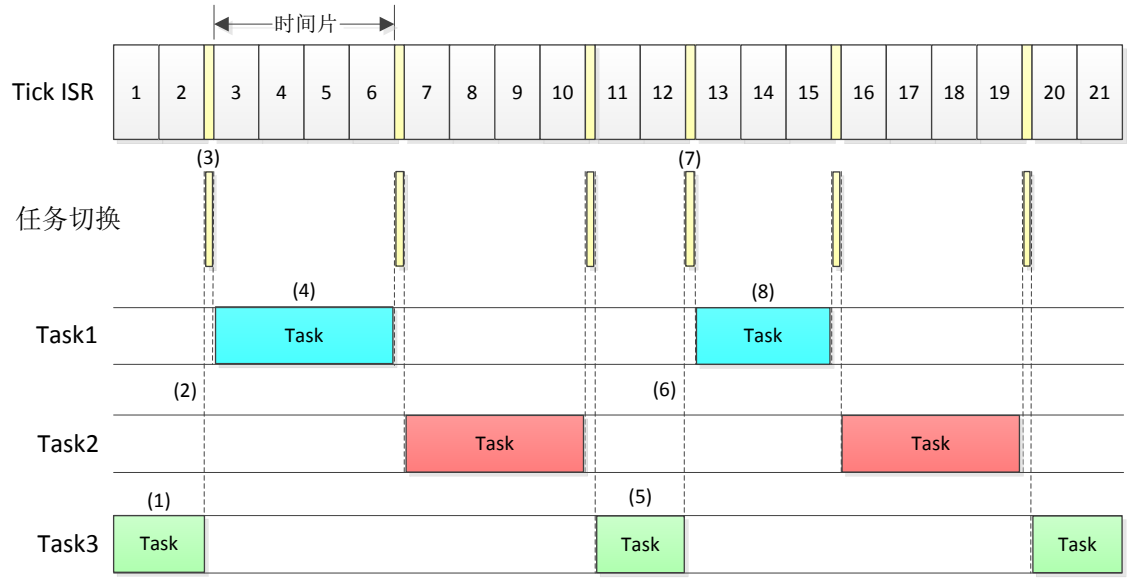


图 5.6.1 轮转调度

(1) 任务 3 正在运行，这时一个时钟节拍中断发生，但是任务 3 的时间片还没完成。

(2) 任务 3 的时钟片用完。

(3) UCOSIII 切换到任务 1，任务 1 是优先级 N 下的下一个就绪任务。

(4) 任务 1 连续运行至时间片用完。

(5) 任务 3 运行。

(6) 任务 3 调用 OSSchedRoundRobinYield()(在 os_core.c 文件中定义)函数放弃剩余的时间片，从而使优先级 X 下的下一个就绪的任务运行。

(7) UCOSIII 切换到任务 1。

(8) 任务 1 执行完其时间片

我们前面讲了任务级调度器和中断级调度器，这里我们要讲解的肯定是时间片轮转调度器，如果当前任务的时间片已经运行完，但是同一优先级下有多个任务，那么 UCOSIII 就会切换到该优先级对应的下一个任务，通过调用 OS_SchedRoundRobin()函数来完成，这个函数由 OSTimeTick()或者 OS_IntQTask()调用，函数代码如下。

```
void    OS_SchedRoundRobin (OS_RDY_LIST    *p_rdy_list)
{
    OS_TCB     *p_tcb;
        CPU_SR_ALLOC();

    if (OSSchedRoundRobinEn != DEF_TRUE) {                    (1)
        return;
    }

    CPU_CRITICAL_ENTER();
    p_tcb = p_rdy_list->HeadPtr;                              (2)
    if (p_tcb == (OS_TCB *)0) {                               (3)
            CPU_CRITICAL_EXIT();
        return;
```

```
    }

    if (p_tcb == &OSIdleTaskTCB) {                                  (4)
            CPU_CRITICAL_EXIT();
        return;
    }

    if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {                        (5)
        p_tcb->TimeQuantaCtr--;
    }

    if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {                        (6)
        CPU_CRITICAL_EXIT();
        return;
    }

    if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {                   (7)
        CPU_CRITICAL_EXIT();
        return;
    }

    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {                (8)
        CPU_CRITICAL_EXIT();
        return;
    }
    OS_RdyListMoveHeadToTail(p_rdy_list);                           (9)
    p_tcb = p_rdy_list->HeadPtr;                                    (10)
    if (p_tcb->TimeQuanta == (OS_TICK)0) {                          (11)
    p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
    } else {
        p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;                   (12)
    }
    CPU_CRITICAL_EXIT();
}
```

(1) 首先检查时间片轮转调度是否允许。要允许时间片轮转调度的话需要使用 OSSchedRoundRobinCfg ()函数。

(2) 获取某一优先级下就绪任务列表中的第一个任务。

(3) 如果 p_tcb 为 0 ，说明没有任务就绪那就直接返回

(4) 如果 p_tcb 为空闲任务的 TCB 那么也就直接返回

(5) 任务控制块 OS_TCB 中的 TimeQuantaCtr 字段表示当前任务的时间片还剩多少，在这里 TimeQuantaCtr 减 1。

(6) 在经过(5)将 TimeQuantaCtr 减 1 以后，判断这时的 TimeQuantaCtr 是否大于 0，如果大于 0 说明任务的时间片还没用完，那么就不能进行任务切换，直接返回。

(7) 前面说过就绪任务列表中的 NbrEntries 字段表示某一优先级下的任务数量，这里判断 NbrEntries 是否小于 2，如果任务数小于 2 就不需要做任务切换，直接返回。

(8) 判断调度器是否上锁，如果上锁的话就直接返回。

(9) 当执行到这一步的时候说明当前任务的时间片已经用完，将当前任务的 OS_TCB 从双向链表头移到链表尾。

(10) 获取新的双向链表头，也就是下一个要执行的任务。

(11) 我们要为下一个要执行的任务装载时间片值，一般我们在新建任务的时候会指定的，这个指定的值被存放在任务控制块 OS_TCB 的 TimeQuanta 字段中，这里我们判断 TimeQuanta 是否为 0，如果为 0 的话那么任务剩余的时间片 TimeQuantaCtr 就使用默认值 OSSchedRoundRobinDfltTimeQuanta，如果我们使能了 UCOSIII 的时间片轮转调度功能的话，OSSchedRoundRobinDfltTimeQuanta 在我们调用 OSInit()函数初始化 UCOSIII 的时候就会被初始化为 OSCfg_TickRate_Hz / 10u ，比如 OSCfg_TickRate_Hz 为 200 的话那么默认的时间片就为 20。

(12) 如果 TimeQuanta 不等于 0 ，也就是说我们定义了任务的时间片，那么 TimeQuantaCtr 就等于 TimeQuanta，也就是我们设置的时间片值。

通过上面的讲解我们可以清晰的看到，如果某一优先级下有多个任务话，这些任务是如何被调度和运行的，每次任务切换后运行的都是处于就绪任务列表 OSRdyList[]链表头的任务，当这个任务的时间片用完后这个任务就会被放到链表尾，然后再运行新的链表头的任务。

第六章 任务相关 API 函数使用

在上一章中我们讲解了 UCOSIII 的任务管理，我们学习就是为了使用，这一节我们就讲解一下 UCOSIII 如何创建任务和任务相关函数的使用，本章分为如下几个部分：

6.1 任务创建和删除实验

6.2 任务挂起和恢复实验

6.3 时间片轮转调度实验

6.1 任务创建和删除实验

6.1.1 OSTaskCreate()函数

UCOSIII 是多任务系统，那么肯定要能创建任务，创建任务就是将任务控制块、任务堆栈、任务代码等联系在一起，并且初始化任务控制块的相应字段。在 UCOSIII 中我们通过函数 OSTaskCreate()来创建任务，OSTaskCreate()函数原型如下(在 os_task.c 中有定义)。调用 OSTaskCreat()创建一个任务以后，刚创建的任务就会进入就绪态，注意！不能在中断服务程序中调用 OSTaskCreat()函数来创建任务。

```
void  OSTaskCreate (OS_TCB          *p_tcb,
                    CPU_CHAR        *p_name,
                    OS_TASK_PTR     p_task,
                    void            *p_arg,
                    OS_PRIO         prio,
                    CPU_STK         *p_stk_base,
                    CPU_STK_SIZE    stk_limit,
                    CPU_STK_SIZE    stk_size,
                    OS_MSG_QTY      q_size,
                    OS_TICK         time_quanta,
                    void            *p_ext,
                    OS_OPT          opt,
                    OS_ERR          *p_err)
```

***p_tcb：** 指向任务的任务控制块 OS_TCB。

***p_name：** 指向任务的名字，我们可以给每个任务取一个名字

p_task： 执行任务代码，也就是任务函数名字

***p_arg：** 传递给任务的参数

prio： 任务优先级，数值越低优先级越高，用户不能使用系统任务使用的那些优先 级！

***p_stk_base：** 指向任务堆栈的基地址。

stk_limit： 任务堆栈的堆栈深度，用来检测和确保堆栈不溢出。

stk_size： 任务堆栈大小

q_size： UCOSIII 中每个任务都有一个可选的内部消息队列，我们要定义宏 OS_CFG_TASK_Q_EN>0，这是才会使用这个内部消息队列。

time_quanta： 在使能时间片轮转调度时用来设置任务的时间片长度，默认值为时钟节拍除以 10。

***p_ext：** 指向用户补充的存储区。

opt： 包含任务的特定选项，有如下选项可以设置。

OS_OPT_TASK_NONE	表示没有任何选项
OS_OPT_TASK_STK_CHK	指定是否允许检测该任务的堆栈
OS_OPT_TASK_STK_CLR	指定是否清除该任务的堆栈
OS_OPT_TASK_SAVE_FP	指定是否存储浮点寄存器，CPU 需要有浮点运算硬件并且有专用代码保存浮点寄存器。

***p_err：** 用来保存调用该函数后返回的错误码。

6.1.2 OSTaskDel()函数

OSTaskDel()函数用来删除任务，当一个任务不需要运行的话，我们就可以将其删除掉，删除任务不是说删除任务代码，而是 UCOSIII 不再管理这个任务，在有些应用中我们只需要某个任务只运行一次，运行完成后就将其删除掉，比如外设初始化任务，OSTaskDel()函数原型如下：

```
void    OSTaskDel (OS_TCB    *p_tcb,
                   OS_ERR    *p_err)
```

***p_tcb：** 指向要删除的任务 TCB，也可以传递一个 NULL 指针来删除调用 OSTaskDel()函数的任务自身。

***p_err：** 指向一个变量用来保存调用 OSTaskDel()函数后返回的错误码。

虽然 UCOSIII 允许用户在系统运行的时候来删除任务，但是应该尽量的避免这样的操作，如果多个任务使用同一个共享资源，这个时候任务 A 正在使用这个共享资源，如果删除了任务 A，这个资源并没有得到释放，那么其他任务就得不到这个共享资源的使用权，会出现各种奇怪的结果。

我们调用 OSTaskDel()删除一个任务后，这个任务的任务堆栈、OS_TCB 所占用的内存并没有释放掉，因此我们可以利用他们用于其他的任务，当然我们也可以使用内存管理的方法给任务堆栈和 OS_TCB 分配内存，这样当我们删除掉某个任务后我们就可以使用内存释放函数将这个任务的任务堆栈和 OS_TCB 所占用的内存空间释放掉。

6.1.3 实验程序设计

例 6-1： 设计 3 个任务，任务 A 用于创建其他任务，创建完成以后就删除掉自身，任务 B 和任务 C 在 LCD 上有各自的运行区域，每隔 1s 他们都会切换一次各自运行区域的背景颜色，而且显示各自的运行次数，任务 B 运行 5 次以后删除掉任务 C，两个任务运行的过程中还要通过串口打印各自的运行次数，当任务 B 删除掉任务 C 以后也要通过串口打印提示信息。

答： 任务代码如下，完整工程请参考 “例 6-1 UCOSIII 任务创建和删除”。

```
#define START_TASK_PRIO          3          //start_task 任务优先级
#define START_STK_SIZE           128        // start_task 任务堆栈大小
OS_TCB StartTaskTCB;                        // start_task 任务控制块
CPU_STK START_TASK_STK[START_STK_SIZE]; // start_task 任务堆栈
void start_task(void *p_arg);               // start_task 任务函数

#define TASK1_TASK_PRIO          4          // task1_task 任务优先级
#define TASK1_STK_SIZE           64         // task1_task 任务堆栈大小
OS_TCB Task1_TaskTCB;                       // task1_task 任务控制块
CPU_STK TASK1_TASK_STK[TASK1_STK_SIZE]; // start_task 任务堆栈
void task1_task(void *p_arg);               //task1_task 任务函数

#define TASK2_TASK_PRIO          5          // task2_task 任务优先级
#define TASK2_STK_SIZE           64         // task2_task 任务堆栈大小
OS_TCB Task2_TaskTCB;                       // task2_task 任务控制块
CPU_STK TASK2_TASK_STK[TASK2_STK_SIZE]; // task2_task 任务堆栈
void task2_task(void *p_arg);               // task2_task 任务函数
```

```
//LCD 刷屏时使用的颜色
int lcd_discolor[14]={   WHITE, BLACK, BLUE,   BRED,
                         GRED,   GBLUE, RED,    MAGENTA,
                         GREEN, CYAN,   YELLOW,BROWN,
                         BRRED, GRAY };

//主函数
int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);          //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);        //串口初始化

    LED_Init();               //LED 初始化
    LCD_Init();               //LCD 初始化
    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,30,200,16,16,"UCOSIII Examp 6-1");
    LCD_ShowString(30,50,200,16,16,"Task Creat and Del");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2014/11/25");

    OSInit(&err);                   //初始化 UCOSIII
    OS_CRITICAL_ENTER();  //进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB*       )&StartTaskTCB, //任务控制块          (1)
                 (CPU_CHAR*     )"start task",      //任务名字
                 (OS_TASK_PTR )start_task,        //任务函数
                 (void*         )0,                    //传递给任务函数的参数
                 (OS_PRIO       )START_TASK_PRIO,      //任务优先级
                 (CPU_STK*      )&START_TASK_STK[0], //任务堆栈基地址
                 (CPU_STK_SIZE)START_STK_SIZE/10,    //任务堆栈深度限位
                 (CPU_STK_SIZE)START_STK_SIZE,       //任务堆栈大小
                 (OS_MSG_QTY )0,                     //任务内部消息队列能够接收
                                                     //的最大消息数目,为 0 时禁止
                                                     //接收消息
                 (OS_TICK       )0,                  //当使能时间片轮转时的时间
                                                     //片长度，为 0 时为默认长度
                 (void*         )0,                  //用户补充的存储区
```

```
                (OS_OPT        )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                                                        //任务选项
                (OS_ERR*       )&err);                  //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);          //开启 UCOSIII
}


//开始任务任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    OS_CRITICAL_ENTER();   //进入临界区
    //创建 TASK1 任务
    OSTaskCreate((OS_TCB*       )&Task1_TaskTCB,                    (2)
                 (CPU_CHAR*     )"Task1 task",
                 (OS_TASK_PTR )task1_task,
                 (void*         )0,
                 (OS_PRIO       )TASK1_TASK_PRIO,
                 (CPU_STK*      )&TASK1_TASK_STK[0],
                 (CPU_STK_SIZE)TASK1_STK_SIZE/10,
                 (CPU_STK_SIZE)TASK1_STK_SIZE,
                 (OS_MSG_QTY )0,
                 (OS_TICK       )0,
                 (void*         )0,
                (OS_OPT        )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*       )&err);

    //创建 TASK2 任务
    OSTaskCreate((OS_TCB*       )&Task2_TaskTCB,                    (3)
                 (CPU_CHAR*     )"task2 task",
                 (OS_TASK_PTR )task2_task,
                 (void*         )0,
                 (OS_PRIO       )TASK2_TASK_PRIO,
                 (CPU_STK*      )&TASK2_TASK_STK[0],
                 (CPU_STK_SIZE)TASK2_STK_SIZE/10,
                 (CPU_STK_SIZE)TASK2_STK_SIZE,
                 (OS_MSG_QTY )0,
                 (OS_TICK       )0,
                 (void*         )0,
```

```
                    (OS_OPT        )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                    (OS_ERR*       )&err);
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err); //删除 start_task 任务自身                    (4)
}


//task1 任务函数
void task1_task(void *p_arg)
{
    u8 task1_num=0;
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    OS_CRITICAL_EXIT();
    while(1)
    {
        task1_num++;     //任务执 1 行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0= ~LED0;
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        if(task1_num==5)
        {
            //任务 1 执行 5 此后删除掉任务 2
            OSTaskDel((OS_TCB*)&Task2_TaskTCB,&err);                    (5)
            printf("任务 1 删除了任务 2!\r\n");
        }
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]); //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80);     //显示任务执行次数
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_HMSM_STRICT,&err); //延时 1s    (6)
    }
}

//task2 任务函数
void task2_task(void *p_arg)
{
    u8 task2_num=0;
```

```
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Task2 Run:000");
    OS_CRITICAL_EXIT();
    while(1)
    {
        task2_num++;     //任务 2 执行次数加 1 注意 task1_num2 加到 255 的时候会清零!!
        LED1=~LED1;
        printf("任务 2 已经执行：%d 次\r\n",task2_num);
        LCD_ShowxNum(206,111,task2_num,3,16,0x80);  //显示任务执行次数
        LCD_Fill(126,131,233,313,lcd_discolor[13-task2_num%14]); //填充区域
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_HMSM_STRICT,&err); //延时 1s
    }
}
```

(1) 创建开始任务 start_task，start_task 任务用来创建另外两个任务：task1_task 和 task2_task。

(2) 开始任务 start_task 中用来创建任务 1：task1_task。

(3) 开始任务 start_task 中用来创建任务 2：task2_task。

(4) 开始任务 start_task 只是用来创建任务 task1_task 和 task2_task，那么这个任务肯定只需要执行一次，两个任务创建完成以后就可以删除掉 start_task 任务了，这里我们使用 OSTaskDel()函数删除掉任务自身，这里传递给 OSTaskDel()函数参数 p_tcb 的值为 0，表示删除掉任务自身。

(5) 根据要求我们在任务 1 执行 5 次后由任务 1 删除掉任务 2，这里通过调用 OSTaskDel()函数删除掉任务 2，注意这时我们传递给 OSTaskDel()中参数 p_tcb 的值为任务 2 的任务控制块 Task2_TaskTCB 的地址，因此这里我们用了取址符号“&”。

(6) 调用函数 OSTimeDlyHMSM()延时 1s，调用 OSTimeDlyHMSM()函数以后就会发起一个任务切换。

6.1.4 程序运行结果分析

程序编译完成后下载到开发板中查看结果和我们要求的是否一致，下载代码后我们看到任务 1 和任务 2 开始运行，根据串口调试助手输出信息我们可以看到任务 1 运行 5 次后删除了任务任务 2，任务 2 停止运行，LCD 显示如图 6.1.1。

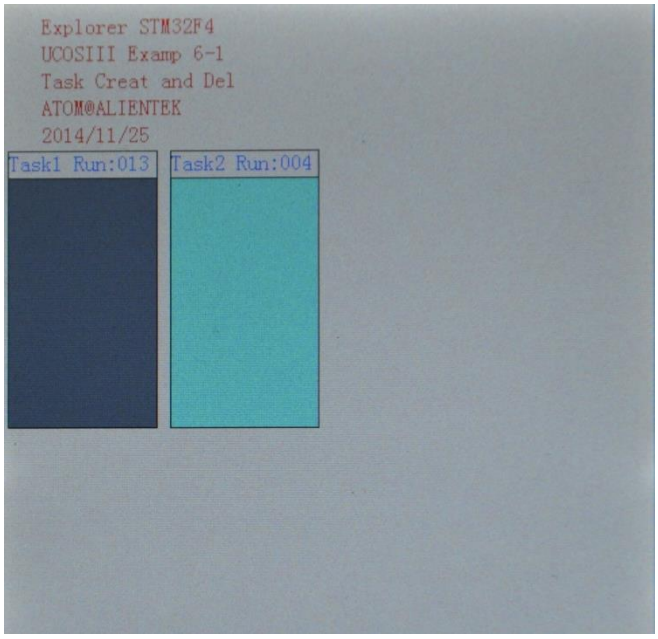


图 6.1.1 LCD 显示效果图

左边方框是任务 1 的运行区域，右边方框是任务 2 的运行区域，可以看出此时任务 1 运行了 13 此，而任务 2 运行了 4 次就停止了，我们在来看一下串口调试助手的输出信息，如图 6.1.2 所示。

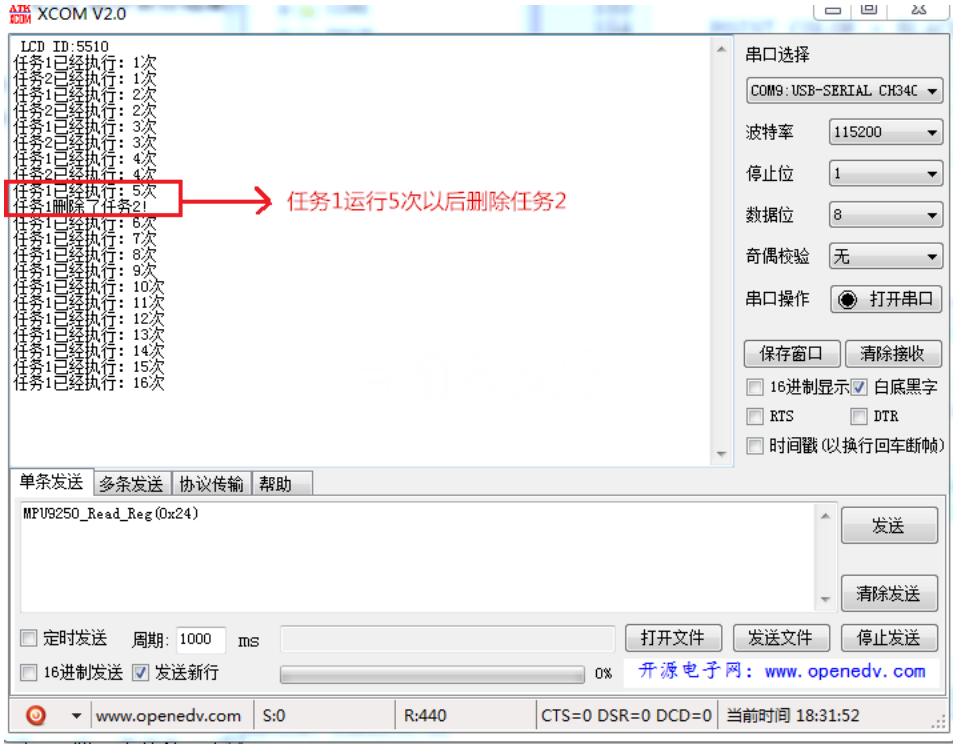


图 6.1.2 串口调试助手输出信息。

从图 6.1.2 中我们可以看出，一开始任务 1 和任务 2 同时运行，因为任务 1 的优先级比任务 2 的优先级高，所以我们看到任务 1 先输出信息，然后再是任务 2。当任务 1 运行 5 次以后删除掉了任务 2，以后只有任务 1 单独运行。

6.2 任务挂起和恢复实验

6.2.1 OSTaskSuspend()函数

有时候有些任务因为某些原因需要暂停运行，但是以后还要运行，因此我们就不能删除掉任务，这里我们可以使用 OSTaskSuspend()函数挂起这个任务，以后再恢复运行，函数 OSTaskSuspend()的原型如下：

```
void    OSTaskSuspend (OS_TCB    *p_tcb,OS_ERR    *p_err)
```

***p_tcb ：** 指向需要挂起的任务的 OS_TCB，可以通过指向一个 NULL 指针将调用该函数的任务挂起。

***p_err：** 指向一个变量，用来保存该函数的错误码。

我们可以多次调用 OSTaskSuspend ()函数来挂起一个任务，因此我们需要调用同样次数的 OSTaskResume()函数才可以恢复被挂起的任务，这一点非常重要。

6.2.2 OSTaskResume()函数

OSTaskResume()函数用来恢复被 OSTaskSuspend()函数挂起的任务，OSTaskResume()函数是唯一能恢复被挂起任务的函数。如果被挂起的任务还在等待别的内核对象，比如事件标志组、信号量、互斥信号量、消息队列等，即使使用 OSTaskResume()函数恢复了被挂起的任务，该任务也不一定能立即运行，该任务还是要等相应的内核对象，只有等到内核对象后才可以继续运行，OSTaskResume()函数原型如下：

```
void    OSTaskResume (OS_TCB    *p_tcb,OS_ERR    *p_err)
```

***p_tcb ：** 指向需要解挂的任务的 OS_TCB，指向一个 NULL 指针是无效的，因为该任务正在运行，不需要解挂。

***p_err：** 指向一个变量，用来保存该函数的错误码。

6.2.3 实验程序设计

例 6-2：本实验是在例 6-1 的基础上完成的，本实验同样设计了 3 个任务，任务 A 用于创建其他任务，创建完成以后就删除掉自身，任务 B 和任务 C 在 LCD 上有各自的运行区域，每隔 1s 他们都会切换一次各自运行区域的背景颜色，而且显示各自的运行次数，任务 B 运行 5 次以后挂起任务 C，当任务 B 运行 10 次以后重新恢复任务 C，两个任务运行的过程中还要通过串口打印各自的运行次数，当任务 B 挂起和恢复任务 C 以后也要通过串口打印提示信息。

答：程序部分代码如下，完整工程请参考我们“例 6-2 UCOSIII 任务挂起和恢复”

```
//task1 任务函数
void task1_task(void *p_arg)
{
    u8 task1_num=0;
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
```

```
    LCD_DrawLine(5,130,115,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"Task1 Run:000");
    OS_CRITICAL_EXIT();
    while(1)
    {
        task1_num++; //任务 1 执行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LED0= ~LED0;
        printf("任务 1 已经执行：%d 次\r\n",task1_num);
        if(task1_num==5)
        {
            //任务 1 执行 5 次后挂起任务 2
            OSTaskSuspend((OS_TCB*)&Task2_TaskTCB,&err);          (1)
            printf("任务 1 挂起了任务 2!\r\n");
        }
        if(task1_num==10)
        {
            //任务 1 运行 10 次后恢复任务 2
            OSTaskResume((OS_TCB*)&Task2_TaskTCB,&err);           (2)
            printf("任务 1 恢复了任务 2!\r\n");
        }
        LCD_Fill(6,131,114,313,lcd_discolor[task1_num%14]); //填充区域
        LCD_ShowxNum(86,111,task1_num,3,16,0x80); //显示任务执行次数
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_HMSM_STRICT,&err); //延时 1s

    }
}
```

这里我们只列出了任务 1 任务函数，任务 2 和其他代码和例 6-1 一样。

(1) 根据要求任务 1 运行 5 次后调用 OSTaskSuspend()函数挂起任务 2。

(2) 当任务 1 运行到第 10 次就调用函数 OSTaskResume()函数解挂任务 2。

6.2.4 程序运行结果分析

程序编译完成后下载到开发板中查看结果和我们要求的是否一致，当任务 1 运行大于 5 次小于 10 次的时候 LCD 显示如图 6.2.1 所示。

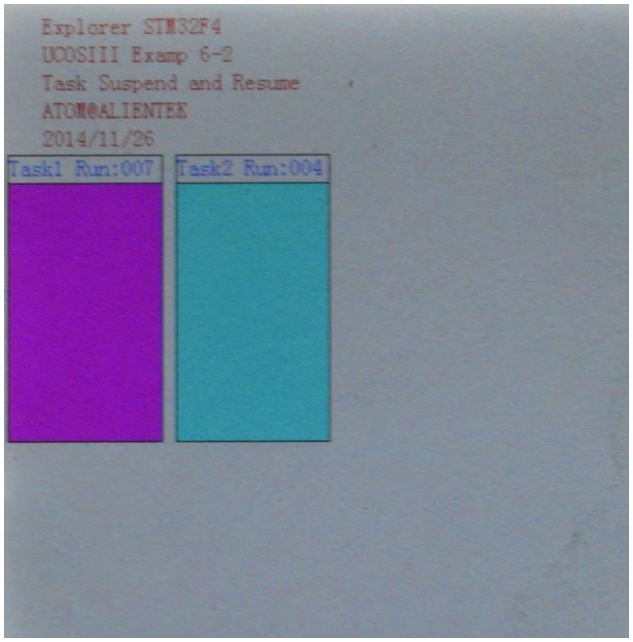


图 6.2.1 任务 2 被挂起

从图 6.2.1 中我们可以看出，任务 1 在继续运行，此时已经运行了 7 次，而任务 2 只运行了 4 次就停止了，说明任务 1 挂起了任务 2。因为任务 1 的优先级比任务 2 的优先级高，所以在任务 1 运行第 5 次的时候直接挂起了任务 2，这样任务 2 本来已经就绪要运行第 5 次了，但是由于被挂起了，所有就不能运行，因此显示任务 2 运行只运行了 4 次！当任务 1 运行到第 10 次以后就会恢复任务 2，如图 6.2.2 所示。

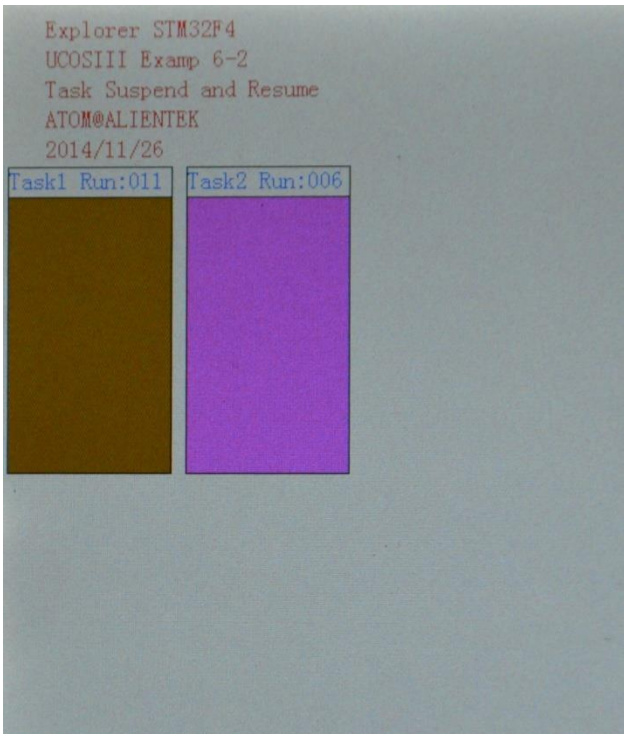


图 6.2.2 任务 2 恢复

从图 6.2.2 中可以看出任务 1 运行了 11 次，因此肯定恢复了任务 2，所以任务 2 可以正常运行，此时任务 2 运行了 6 次，相比任务 1 少了 5 次，我们在来看一下串口调试助手输出的信

息，如图 6.2.3 所示。

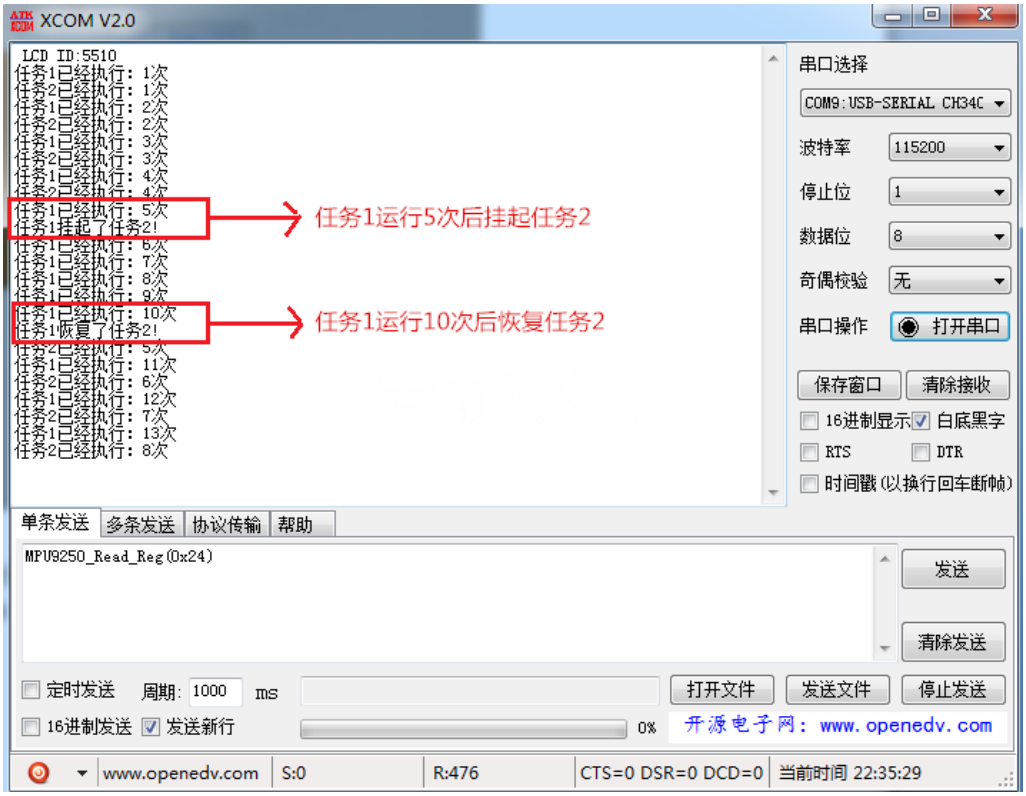


图 6.2.3 串口调试助手输出信息

从串口调试助手中我们更能清晰的看出任务 1 挂起和恢复任务 2 的过程！最后任务 1 和任务 2 都可以运行，因为任务 2 被挂起了 5 个“轮回”，因此最后任务 1 的运行次数比任务 2 多 5 次。

6.3 时间片轮转调度实验

我们说过 UCOSIII 是支持多个任务拥有同一个优先级的，这些任务采用时间片轮转调度方法进行任务调度。在 os_cfg.h 文件中有个宏 OS_CFG_SCHED_ROUND_ROBIN_EN，我们要想使用时间片轮转调度就需要将 OS_CFG_SCHED_ROUND_ROBIN_EN 定义为 1，这样 UCOSIII 中有关时间片轮转调度的代码才会被编译，否则不能使用时间片轮转调度，这点特别重要！

6.3.1 OSSchedRoundRobinCfg()函数

OSSchedRoundRobinCfg()函数用来使能或失能 UCOSIII 的时间片轮转调度功能，如果我们要使用时间片轮转调度功能的话不仅要将宏 OS_CFG_SCHED_ROUND_ROBIN_EN 定义为 1，还需要调用 OSSchedRoundRobinCfg()函数来使能 UCOSIII，OSSchedRoundRobinCfg()函数原型如下。

```
void  OSSchedRoundRobinCfg ( CPU_BOOLEAN   en,
                             OS_TICK       dflt_time_quanta,
                             OS_ERR        *p_err)
```

en： 用于设置打开或关闭时间片轮转调度机制，如果为 DEF_ENABLED 表示打开时间片轮转调度，为 DEF_DISABLED 表示关闭时间片轮转调度。

dflt_time_quanta： 设置默认的时间片长度，就是系统时钟节拍个数，比如我们设置系统时钟频率 OSCfg_TickRate_Hz 为 200Hz，那么每个时钟节拍就是 5ms。当

我们设置 dflt_time_quanta 为 n 时，时间片长度就是(5*n)ms 长，如果我们设置 dflt_time_quanta 为 0 的话，UCOSIII 就会使用系统默认的时间片长度：OSCfg_TickRate_Hz / 10，比如如果 OSCfg_TickRate_Hz 为 200，那么时间片长度为：200/10*5=100ms。

***p_err：** 保存调用此函数后返回的错误码

6.3.2 OSSchedRoundRobinYield()函数

当一个任务想放弃本次时间片，把 CPU 的使用权让给同优先级下的另外一个任务就可以使用 OSSchedRoundRobinYield()函数，函数原型如下：

```
void    OSSchedRoundRobinYield (OS_ERR    *p_err)
```

***p_err：** 用来保存函数调用后返回的错误码。

OS_ERR_NONE	调用成功
OS_ERR_ROUND_ROBIN_1	当前优先级下没有其他就绪任务
OS_ERR_ROUND_ROBIN_DISABLED	未使能时间片轮转调度功能
OS_ERR_YIELD_ISR	在中断调用了本函数。

我们在调用这个后函数遇到最多的错误就是 OS_ERR_ROUND_ROBIN_1，也就是当前优先级下没有就绪任务了。

6.3.3 实验程序设计

例 6-3 ：本实验同样设计了 3 个任务，任务 A 用于创建其他任务，创建完成以后就删除掉自身，任务 B 和任务 C 拥有同样的优先级，这两个任务采用时间片轮转调度，两个任务都是通过串口打印一些数据，然后在 LCD 上显示任务的运行次数。可以通过串口输出的信息情况来观察时间片轮转调度的运行。

答：实验关键代码如下，实验完整工程见“例 6-3 UCOSIII 时间片轮转调度”

为了测试时间片轮转调度，因此这里需要将两个任务的优先级设置为一样的

```
#define TASK1_TASK_PRIO          4          //任务 1 优先级
#define TASK2_TASK_PRIO          4          //任务 2 优先级
```

因为要使用时间片轮转调度功能，那么 start_task 任务函数中在创建其他两个测试任务的时候就需要进行相应的设置，比如开启时间片轮转调度功能，创建任务的时候还需要设置每个任务的时间片数量，start_task 任务函数如下：

```
//开始任务任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

#if   OS_CFG_SCHED_ROUND_ROBIN_EN    //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);                    (1)
#endif

    OS_CRITICAL_ENTER();   //进入临界区
```

```
    //创建 TASK1 任务
    OSTaskCreate((OS_TCB *          )&Task1_TaskTCB,
                 (CPU_CHAR*         )"Task1 task",
                 (OS_TASK_PTR       )task1_task,
                 (void*             )0,
                 (OS_PRIO           )TASK1_TASK_PRIO,
                 (CPU_STK*          )&TASK1_TASK_STK[0],
                 (CPU_STK_SIZE      )TASK1_STK_SIZE/10,
                 (CPU_STK_SIZE      )TASK1_STK_SIZE,
                 (OS_MSG_QTY        )0,
                 (OS_TICK           )3,   //3 个时间片，既 3*5=15ms                (2)
                 (void *            )0,
                 (OS_OPT            )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*           )&err);

    //创建 TASK2 任务
    OSTaskCreate((OS_TCB *          )&Task2_TaskTCB,
                 (CPU_CHAR*         )"task2 task",
                 (OS_TASK_PTR       )task2_task,
                 (void*             )0,
                 (OS_PRIO           )TASK2_TASK_PRIO,
                 (CPU_STK *         )&TASK2_TASK_STK[0],
                 (CPU_STK_SIZE      )TASK2_STK_SIZE/10,
                 (CPU_STK_SIZE      )TASK2_STK_SIZE,
                 (OS_MSG_QTY        )0,
                 (OS_TICK           )3,   //3 个时间片，既 3*5=15ms                (3)
                 (void *            )0,
                 (OS_OPT            )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*           )&err);
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err);   //删除 start_task 任务自身
}
```

(1) 这里我们使能时间片轮转调度机制，只有在宏 OS_CFG_SCHED_ROUND_ROBIN_EN 定义为 1 的时候，也就是允许使用时间片轮转调度的时候我们才调用 OSSchedRoundRobinCfg()函数。

(2) 任务 task1_task 的时间片长度为 3，也就是 3*5=15ms。

(3) 任务 task2_task 的时间片长度也为 3。

任务 1 和任务 2 的任务函数代码如下：

```
//task1 任务函数
void task1_task(void *p_arg)
{
    u8 i,task1_num=0;
    OS_ERR err;
```

```
    CPU_SR_ALLOC();
    p_arg = p_arg;

    POINT_COLOR = RED;
    LCD_ShowString(30,130,110,16,16,"Task1 Run:000");
    POINT_COLOR = BLUE;
    while(1)
    {
        task1_num++; //任务 1 执行次数加 1  注意 task1_num1 加到 255 的时候会清零!!
        LCD_ShowxNum(110,130,task1_num,3,16,0x80);     //显示任务执行次数
        for(i=0;i<5;i++) printf("Task1:01234\r\n");                    (1)
        LED0 = ~LED0;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_HMSM_STRICT,&err); //延时 1s

    }
}

//task2 任务函数
void task2_task(void *p_arg)
{
    u8 i,task2_num=0;
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    POINT_COLOR = RED;
    LCD_ShowString(30,150,110,16,16,"Task2 Run:000");
    POINT_COLOR = BLUE;
    while(1)
    {
        task2_num++; //任务 2 执行次数加 1  注意 task1_num2 加到 255 的时候会清零!!
        LCD_ShowxNum(110,150,task2_num,3,16,0x80);   //显示任务执行次数
        for(i=0;i<5;i++) printf("Task2:56789\r\n");                    (2)
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_HMSM_STRICT,&err); //延时 1s
    }
}
```

(1) 任务 1 中通过串口打印 5 次“Task1:01234\r\n”这个字符串，方便观察一个任务还未运行完但是时间片用完被其他任务抢夺 CPU 使用权。

(2) 和任务 1 一样，不过为了区分与任务 1 的区别这里通过串口打印字符串“Task2:56789\r\n”。

6.3.4 实验程序运行结果

代码编译完成后下载到开发板中观察和分析实验现象，程序运行过程中 LCD 显示如图 6.3.1 所示。

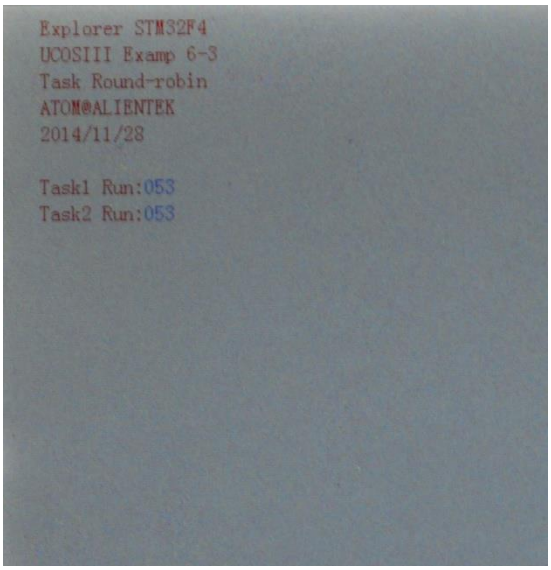


图 6.3.1 LCD 显示

从图 6.3.1 中我们可以看到任务 1 和任务 2 各自都运行了 53 次，说明时间片轮转调度起作用了，因为任务 1 和任务 2 拥有相同的优先级，如果时间片轮转调度没有起作用的话肯定会出错的。但是我们从图 6.3.1 中我们并不能看出时间片轮转调度执行的细节，这时候我们就需要观察串口输出了，串口输出如图 6.3.2 所示。

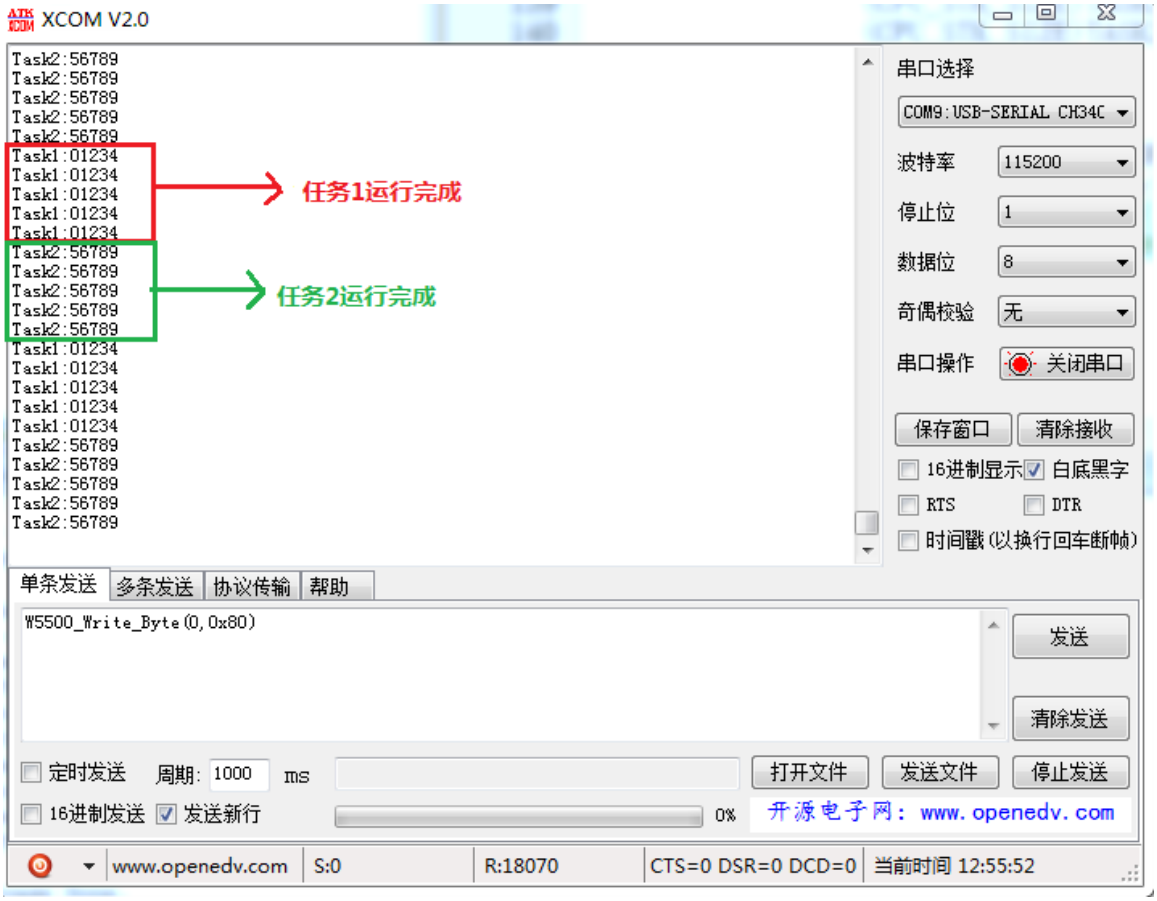


图 6.3.2 串口调试助手输出信息

我们在任务 1 中通过串口循环输出 5 次字符串“Task1:01234”，任务 2 中通过串口循环输出 5 次字符串“Task2:56789”。通过图 6.3.2 可以看出任务 1 和任务 2 运行正常，说明时间片轮转调度正常。

第七章 UCOSIII 系统内部任务

在 UCOSII 中我们知道有两个系统任务：统计任务和空闲任务，在 UCOSIII 中系统内部任务扩展到了 5 个，本章我们就详细的讲解一下 UCOSIII 中的这 5 个系统任务，本章分为如下几部分：

7.1 空闲任务
7.2 时钟节拍任务
7.3 统计任务
7.4 定时任务
7.5 中断服务管理任务
7.6 钩子函数

7.1 空闲任务

我们首先来看一下空闲任务：OS_IdleTask()，在 os_core.c 文件中定义。任务 OS_IdleTask()是必须创建的，不过不需要手动创建，在调用 OS_Init()初始化 UCOS 的时候就会被创建。打开 OS_Init()函数，可以看到，在 OS_Init()中调用了函数 OS_IdleTaskInit()，打开函数 OS_IdleTaskInit()，函数代码如下：

```
void  OS_IdleTaskInit (OS_ERR  *p_err)
{
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

    OSIdleTaskCtr = (OS_IDLE_CTR)0;                                    (1)

    OSTaskCreate((OS_TCB *          )&OSIdleTaskTCB,
                 (CPU_CHAR *        )((void *)"uC/OS-III Idle Task"),
                 (OS_TASK_PTR       )OS_IdleTask,
                 (void *            )0,
                 (OS_PRIO           )(OS_CFG_PRIO_MAX - 1u),
                 (CPU_STK *         )OSCfg_IdleTaskStkBasePtr,
                 (CPU_STK_SIZE      )OSCfg_IdleTaskStkLimit,
                 (CPU_STK_SIZE      )OSCfg_IdleTaskStkSize,
                 (OS_MSG_QTY        )0u,
                 (OS_TICK           )0u,
                 (void *            )0,
                 (OS_OPT            )(OS_OPT_TASK_STK_CHK | \
                                      OS_OPT_TASK_STK_CLR | OS_OPT_TASK_NO_TLS),
                 (OS_ERR *          )p_err);
}
```

(1)、OSIdleTaskCtr 在文件 os.h 中定义，是一个 32 位无符号整型变量。这里将 OSIdleTaskCtr 清零。

从上面的代码可以看出，函数 OS_IdleTaskInit()很简单，只是调用了 OSTaskCreate()来创建一个任务，这个任务就是空闲任务。任务优先级为 OS_CFG_PRIO_MAX – 1，OS_CFG_PRIO_MAX 是一个宏，在文件 os_cfg.h 中定义，OS_CFG_PRIO_MAX 定义了 UCOSIII 可用的任务数。前面我们说过 UCOSIII 的任务数是无数的，但是在实际使用中考虑到硬件资源(ROM 和 RAM)等因素，不可能真的使用无数的任务，在 UCOSIII 中可以使用宏 OS_CFG_PRIO_MAX 来定义可使用的任务数，默认情况下 OS_CFG_PRIO_MAX 为 64。空闲任务优先级为 OS_CFG_PRIO_MAX-1，说明空闲任务的优先级为最低的。

空闲任务堆栈大小为 OSCfg_IdleTaskStkSize，OSCfg_IdleTaskStkSize 也是一个宏，在 os_cfg_app.c 文件中定义，默认为 128，则空闲任务堆栈默认为 128*4=512 字节。

空闲任务的任务函数为任务函数为 OS_IdleTask()，OS_IdleTask()函数代码如下：

```
void    OS_IdleTask (void    *p_arg)
{
    CPU_SR_ALLOC();

    p_arg = p_arg;
    while (DEF_ON) {
        CPU_CRITICAL_ENTER();                    (1)
        OSIdleTaskCtr++;                         (2)
#if OS_CFG_STAT_TASK_EN > 0u                     (3)
        OSStatTaskCtr++;                         (4)
#endif
        CPU_CRITICAL_EXIT();                     (5)

        OSIdleTaskHook();                        (6)
    }
}
```

(1)和(5)、临界段代码保护，后续章节会讲解到，这里不做讲解。

(2)、OSIdleTaskCtr 加一，每进入一次空闲任务，OSIdleTaskCtr 就加一。我们可以通过查看 OSIdleTaskCtr 变量的递增速度来判断 CPU 执行应用任务的繁忙程度，如果递增的快的话说明应用任务花费时间少，很快就执行完了。

(3)、宏 OS_CFG_STAT_TASK_EN 大于 0 说明开启了统计任务。

(4)、OSStatTaskCtr 默认也是一个 32 位的无符号整形变量，在文件 os.h 中定义。这里将 OSStatTaskCtr 加一，统计任务中用到 OSStatTaskCtr，用来统计 CPU 的使用率，后面讲解统计任务的时候在详细的讲解 OSStatTaskCtr。

(6)、OSIdleTaskHook()叫做钩子函数，我们可以在钩子函数中干一些其他的事情，下一小节我们将详解钩子函数。

7.2 时钟节拍任务

我们在看另一个必须创建的任务时钟节拍任务：OS_Ticktask()，在 OS_Init()中调用了一个函数 OS_TickTaskInit()，函数代码如下：

```
void    OS_TickTaskInit (OS_ERR    *p_err)
{
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

    OSTickCtr             = (OS_TICK)0u;
    OSTickTaskTimeMax = (CPU_TS)0u;
```

```
    OS_TickListInit();

    if (OSCfg_TickTaskStkBasePtr == (CPU_STK *)0) {
        *p_err = OS_ERR_TICK_STK_INVALID;
         return;
    }

    if (OSCfg_TickTaskStkSize < OSCfg_StkSizeMin) {
        *p_err = OS_ERR_TICK_STK_SIZE_INVALID;
         return;
    }

    if (OSCfg_TickTaskPrio >= (OS_CFG_PRIO_MAX - 1u)) {
         *p_err = OS_ERR_TICK_PRIO_INVALID;
         return;
    }

    OSTaskCreate((OS_TCB *          )&OSTickTaskTCB,
                 (CPU_CHAR*         )((void *)"uC/OS-III Tick Task"),
                 (OS_TASK_PTR       )OS_TickTask,
                 (void*             )0,
                 (OS_PRIO           )OSCfg_TickTaskPrio,
                 (CPU_STK*          )OSCfg_TickTaskStkBasePtr,
                 (CPU_STK_SIZE      )OSCfg_TickTaskStkLimit,
                 (CPU_STK_SIZE      )OSCfg_TickTaskStkSize,
                 (OS_MSG_QTY        )0u,
                 (OS_TICK           )0u,
                 (void*             )0,
                 (OS_OPT            )(OS_OPT_TASK_STK_CHK |\
                                      OS_OPT_TASK_STK_CLR | OS_OPT_TASK_NO_TLS),
                 (OS_ERR        *)p_err);
}
```

可以看到在函数 OS_TickTaskInit()的最后调用 OSTaskCreate()来创建了一个任务，任务函数为 OS_TickTask()，所以说时钟节拍任务是 UCOSIII 必须创建的，同样，不需要我们手工创建。时钟节拍任务的任务优先级为 OSCfg_TickTaskPrio，时钟节拍任务的优先级尽可能的高一点，ALIENTEk 默认设置时钟节拍任务的任务优先级为 1。

时钟节拍任务的作用是跟踪正在延时的任务，以及在指定时间内等待某个内核对象的任务，OS_TickTask()任务函数代码如下：

```
void   OS_TickTask (void   *p_arg)
{
    OS_ERR    err;
    CPU_TS    ts;
```

```
    p_arg = p_arg;
    while (DEF_ON) {
        (void)OSTaskSemPend((OS_TICK      )0,                                    (1)
                            (OS_OPT       )OS_OPT_PEND_BLOCKING,
                            (CPU_TS *     )&ts,
                            (OS_ERR *     )&err);
        if (err == OS_ERR_NONE) {
            if (OSRunning == OS_STATE_OS_RUNNING) {
                OS_TickListUpdate();                                             (2)
            }
        }
    }
}
```

(1)、请求信号量，OSTaskSemPend()是请求任务内建信号量的，信号量会在 OSTimeTick()中 POST，这里的信号量是用来做任务同步的，关于信号量的知识会在后续章节中讲解。前面我们讲过 OSTimeTick()会在滴答定时器中断服务函数中调用。

(2)、信号量请求成功的话就调用函数 OS_TickListUpdate()函数，在了解 OS_TickListUpdate()函数之前我们先来了解时钟节拍任务中一个重要的概念：时钟节拍列表。

时钟节拍列表是由一个数据表 OSCfg_TickWheel(在 os_cfg_app.c 中定义)和一个计数器 OSTickCtr 组成，表 OSCfg_TickWheel 是一个数组，数组元素个数由宏 OS_CFG_TICK_WHEEL_SIZE 定义，宏 OS_CFG_TICK_WHEEL_SIZE 在 os_cfg_app.h 中定义了。表 OSCfg_TickWheel 中的元素为 os_tick_spoke 类型的，os_tick_spoke 是一个结构体，结构体定义如下：

```
struct   os_tick_spoke {
    OS_TCB                  *FirstPtr;
    OS_OBJ_QTY              NbrEntries;
    OS_OBJ_QTY              NbrEntriesMax;
};
```

FirstPtr：指针变量，在表头上并属于该表。

NbrEntries：表示在该表项上等待的任务的数目。

NbrEntriesMax：表示在该表项上等待的任务的最大数目。

在使用时钟节拍列表时需要先初始化时钟节拍列表，在 OS_TickTaskInit()函数中会调用 OS_TickListInit 来初始化时钟节拍列表，函数 OS_TickListInit()代码如下：

```
void   OS_TickListInit (void)
{
    OS_TICK_SPOKE_IX     i;
    OS_TICK_SPOKE        *p_spoke;



    for (i = 0u; i < OSCfg_TickWheelSize; i++) {
        p_spoke                    = (OS_TICK_SPOKE *)&OSCfg_TickWheel[i];
```

```
        p_spoke->FirstPtr          = (OS_TCB          *)0;
        p_spoke->NbrEntries        = (OS_OBJ_QTY       )0u;
        p_spoke->NbrEntriesMax = (OS_OBJ_QTY       )0u;
    }
}
```

初始化时钟节拍列表十分简单，就是将 OSCfg_TickWheel[]中的每个变量都清零，初始完成以后的时钟节拍列表如表 7.2.1 所示。

OSCfg_TickWheel[]			
变量序号	FirstPtr 值	NbrEntries 值	NbrEntriesMax 值
[0]	0	0	0
[1]	0	0	0
[2]	0	0	0
[3]	0	0	0
•••	•••	•••	•••
[OS_CFG_TICK-WHEEL_SIZE-2]	0	0	0
[OS_CFG_TICK-WHEEL_SIZE-1]	0	0	0

表 7.2.1 初始化后的时钟节拍列表

7.3 统计任务

在 UCOSIII 中统计任务可用来统计 CPU 的使用率、各个任务的 CPU 使用率和各任务的堆栈使用情况，默认情况下统计任务是不会创建的，如果要使能统计任务的话需要将宏 OS_CFG_STAT_TASK_EN 置 1，宏 OS_CFG_STAT_TASK_EN 在 os_cfg.h 文件中有定义。当我们将宏 OS_CFG_STAT_TASK_EN 置 1 以后，OSinit()函数中有关统计任务的代码就可以编译了额。OS_StatTaskInit() 函数用来创建统计任务，统计任务的优先级通过宏 OS_CFG_STAT_TASK_PRIO 设置，ALIENTEK 将统计任务的优先级设置为 OS_CFG_PRIO_MAX-2，也就是倒数第二。

如果要使用统计任务的话就需要在 main()函数创建的第一个也是唯一一个应用任务中调用 OSStatTaskCPUUsageInit()函数。注意在 OSStart()之前只能创建一个任务，在我们提供的所有例程中，在 main()函数中只创建了一个任务，就是 start_task()开始任务，start_task()函数示例代码如下：

```
//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
```

```
    OSStatTaskCPUUsageInit(&err);    //统计任务
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN          //如果使能了测量中断关闭时间
    CPU_IntDisMeasMaxCurReset();
#endif

#if   OS_CFG_SCHED_ROUND_ROBIN_EN   //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
#endif

    OS_CRITICAL_ENTER();   //进入临界区
     •••
    创建其他任务
     ••••
    OS_TaskSuspend((OS_TCB*)&StartTaskTCB,&err);        //挂起开始任务
    OS_CRITICAL_EXIT();      //进入临界区
}
```

从上面代码中可以看出最先调用了函数 OSStatTaskCPUUsageInit()，创建其他任务只能在 OSStatTaskCPUUsageInit()函数之后。CPU 的总的使用率会保存在变量 OSStatTaskCPUUsage 中，我们可以通过读取这个值来获取 CPU 的使用率。从 V3.03.00 版本起，CPU 的使用率用一个 0~10000 之间的整数表示(对应 0.00~100.00%)，在这之前的版本,CPU 使用率是 0~100 之间的整数表示。

如果将宏 OS_CFG_STAT_TASK_STK_CHK_EN 置 1 的话表示检查任务堆栈使用情况，那么统计任务就会调用 OSTaskStkChk()函数来计算所有已创建任务的堆栈使用量，并将检测结果写入到每个任务的 OS_TCB 中的 StkFree 和 StkUsed 中。

7.4 定时任务

UCOSIII 提供软件定时器功能，定时任务是可选的，将宏 OS_CFG_TMR_EN 设置为 1 就会使能定时任务，在 OSInit()中将会调用函数 OS_TmrInit()来创建定时任务。定时任务的优先级通过宏 OS_CFG_TMR_TASK_PRIO 定义，ALIENTEK 默认将定时器任务优先级设置为 2。

7.5 中断服务管理任务

当把 os_cfg.h 文件中的宏 OS_CFG_ISR_POST_DEFERRED_EN 置 1 就会使能中断服务管理任务，UCOSIII 会创建一个名为 OS_IntQTask()的任务，该任务负责“延迟”在 ISR 中调用的系统 post 服务函数的行为。中断服务管理任务的任务优先级永远是最高的，为 0！

在 UCOS 中可以通过关闭中断和任务调度器上锁两种方式来管理临界段代码(有关临界段代码保护下一章会详细讲解)，如果采用后一种，即调度器上锁的方式来管理临界段代码的话，那么在中断服务函数中调用的“post”类函数就不允许操作诸如任务就绪表、等待表等系统内部数据结构。

当 ISR(中断服务函数)调用 UCOSIII 提供的“post”函数时，要发送的数据和发送的目的地都会存入一个特别的缓冲队列中，当所有嵌套的 ISR 都执行完成以后 UCOSIII 会做任务切换，

运行中断服务管理任务，该任务会把缓存队列中存放的信息重发给相应的任务。这样做的好处就是可以减少中断关闭的时间，否则，在 ISR 中还需要把任务从等待列表中删除，并把任务放入就绪表，以及做一些其他的耗时操作。

7.6 钩子函数

7.6.1 空闲任务钩子函数

上一小节中我们简单的提了一下空闲任务的钩子函数 OSIdleTaskHook()，本节我们以空闲任务的钩子函数 OSIdleTaskHook()为例来学习一下钩子函数，函数 OSIdleTaskHook()代码如下：

```
void  OSIdleTaskHook (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppIdleTaskHookPtr)();
    }
#endif
}
```

从上面的函数代码中可以看出要使用空闲任务钩子函数的话需要将宏 OS_CFG_APP_HOOKS_EN 置 1，即允许使用空闲任务的钩子函数。当时使能空闲任务的钩子函数以后每次进入空闲任务就会调用指针 OS_AppIdleTaskHookPtr 所指向的函数。OS_AppIdleTaskHookPtr 是何方神圣？打开 os_app_hooks.c 文件，在文件中有个函数 App_OS_SetAllHooks()，函数代码如下：

```
void  App_OS_SetAllHooks (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    CPU_SR_ALLOC();

    CPU_CRITICAL_ENTER();
    OS_AppTaskCreateHookPtr = App_OS_TaskCreateHook;
    OS_AppTaskDelHookPtr    = App_OS_TaskDelHook;
    OS_AppTaskReturnHookPtr = App_OS_TaskReturnHook;

    OS_AppIdleTaskHookPtr   = App_OS_IdleTaskHook;
    OS_AppStatTaskHookPtr   = App_OS_StatTaskHook;
    OS_AppTaskSwHookPtr     = App_OS_TaskSwHook;
    OS_AppTimeTickHookPtr   = App_OS_TimeTickHook;
    CPU_CRITICAL_EXIT();
#endif
}
```

红色代码显示将 App_OS_IdleTaskHook 复制给 OS_AppIdleTaskHookPtr。那么问题来了，OS_AppIdleTaskHookPtr 又是何方神圣？我们仔细查看 os_app_hooks.c 文件，会发现 App_OS_IdleTaskHook 是一个函数，代码如下：

```
void  App_OS_IdleTaskHook (void)
```

```
{


}
```

到这里我们基本就懂了空闲任务的钩子函数 OSIdleTaskHook()是怎么工作了，在 OSIdleTaskHook 中最终调用的是函数 App_OS_IdleTaskHook()，也就是说如果我们要想在空闲任务的钩子函数中做一些其他处理的话就需要将程序代码写在 App_OS_IdleTaskHook()函数中。

注意!在空闲任务的钩子函数中不能调用任何可以是空闲进入等待态的代码,原因很简单，CPU 总是在不停的运行，需要一直工作，不能让 CPU 停下来，哪怕是执行一些对应用没有任何用的代码，比如简单的将一个变量加一。在 UCOS 中为了让 CPU 一直工作，在所有应用任务都进入等待态的时候 CPU 会执行空闲任务，我们可以从空闲任务的任务函数 OS_IdleTask()看出，在 OS_IdleTask()中没有任何可以让空闲任务进入等待态的代码。如果在 OS_IdleTask()中有可以让空闲任务进入等待态的代码的话有可能会在同一时刻所有任务(应用任务和空闲任务)同时进入等待态，此时 CPU 就会无所事事了，所以在空闲任务的钩子函数 OSIdleTaskHook()中不能出现可以让空闲任务进入等待态的代码！这点很重要，一定要谨记！

7.6.2 实验程序设计

例 7-1：在例 4-1 的基础上完成本次实验，当空闲任务每执行 50000 就通过串口打印字符串“Idle Task Running 50000 times!”,因为要使用到串口，为了防止打扰，删除掉例 4-1 中的浮点测试任务。实验步骤如下：

(1)、将宏 OS_CFG_APP_HOOKS_EN 定义为 1，使能钩子函数。

(2)、调用 App_OS_SetAllHooks()函数设置所有的钩子函数使用的 app 函数，我们在开始任务中 start_task()中使用条件编译语句来设置，代码如下：

```
#if OS_CFG_APP_HOOKS_EN                    //使用钩子函数
    App_OS_SetAllHooks();
#endif
```

当 OS_CFG_APP_HOOKS_EN 大于 1 的话就说明要使用钩子函数，那么就会编译函数 App_OS_SetAllHooks()。记得在 main.c 文件中添加头文件 os_app_hooks.h。

(2)、编写钩子函数的内容，也就是在函数 App_OS_IdleTaskHook()中编写我们需要的功能代码，代码如下：

```
void   App_OS_IdleTaskHook (void)
{
    static int num;
    num++;
    if(num%1000==0)
    {
        printf("Idle Task Running 10 times!\r\n");
    }
}
```

7.6.3 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，可以看到 LED0 和 LED1 闪烁，打开串口调试助手，如图 7.3.1 所示。

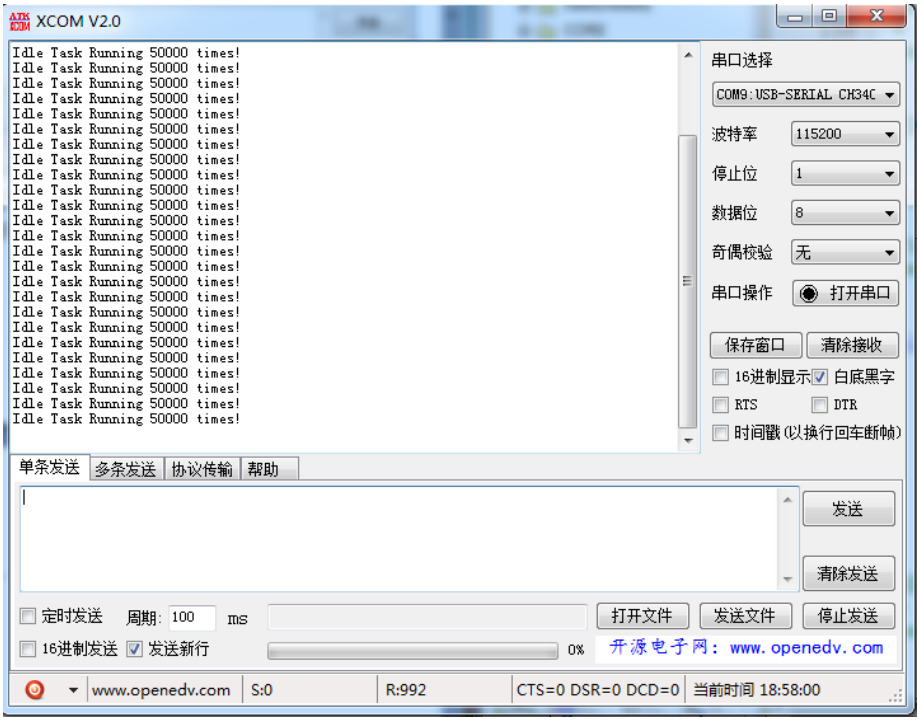


图 7.6.1 空闲任务钩子函数发送数据

我们可以看到串口调试助手接收到字符串“Idle Task Running 50000 times!”说明钩子函数运行正常。

7.6.4 其他任务钩子函数

UCOSIII 中一共有 8 个钩子函数，出了上面讲的空闲任务的钩子函数以外，还有 7 个，分别为： OSInitHook()、OSStatTaskHook()，OSTaskCreateHook()、OSTaskDelHook()、OSTaskReturnHook()、OSTaskSwHook()和 OSTimeTickHook()这些钩子函数的使用方法和空闲任务的钩子函数的使用方法类似，这里就不详解了，大家感兴趣的话可以自行查看一下这些钩子函数。

第八章 UCOSIII 中断和时间管理

本章我们讲解一下 UCOSIII 的中断处理和时间管理，在使用 UCOS 操作系统的时候我们对于中断服务程序的处理就要做一点修改，这个和我们不使用操作系统的时候是不同的。我们在对某些任务做延时的时候会使用到一些延时函数，本章我们就讲解一下这两个知识点，本章分为以下几个部分：

8.1 中断管理

8.2 时间管理

8.1 中断管理

8.1.1 UCOSIII 中断处理过程

在 STM32 中是支持中断的，中断是一个硬件机制，主要用来向 CPU 通知一个异步事件发生了，这时 CPU 就会将当前 CPU 寄存器值入栈，然后转而执行中断服务程序，在 CPU 执行中断服务程序的时候有可能有更高优先级的任务就绪，那么当退出中断服务程序的时候，CPU 就会直接执行这个高优先级的任务。

UCOSIII 是支持中断嵌套的，既高优先级的中断可以打断低优先级的中断，在 UCOSIII 中使用 OSIntNestingCtr 来记录中断嵌套次数，最大支持 250 级的中断嵌套，每进入一次中断服务函数 OSIntNestingCtr 就会加 1，当退出中断服务函数的时候 OSIntNestingCtr 就会减 1。

我们在编写 UCOSIII 的中断服务程序的时候需要使用到两个函数 OSIntEnter()和 OSIntExit()，OSIntExit()函数我们前面已经讲过了是中断级任务调度器，OSIntEnter()的函数代码如下：

```
void    OSIntEnter (void)
{
    if (OSRunning != OS_STATE_OS_RUNNING) {              //判断 UCOSIII 是否运行，
        return;
    }
    if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {       //判断中断嵌套次数是否大于 250
        return;
    }
    OSIntNestingCtr++;                                   //中断嵌套次数加 1
}
```

从上面代码中我们可以看出 OSIntEnter()函数其实就是将 OSIntNestingCtr 进行简单的加一操作，用来中断嵌套的次数而已。

那么我们在 UCOSIII 环境中如何编写中断服务函数呢？我们按照下面所示代码编写中断服务函数：

```
void XXX_Handler(void)                                                    (1)
{
    OSIntEnter();              //进入中断                                 (2)

    用户自行编写的中断服务程序；      //这部分就是我们的中断服务程序      (3)

    OSIntExit();               //触发任务切换软中断                       (4)
}
```

(1) 中断服务程序，XXX 为不同中断源的中断函数名字。

(2) 首先调用 OSIntEnter()函数来标记进入中断服务函数，并且记录中断嵌套次数。

(3) 这部分就是我们需要自行编写的中断服务程序了，也就是我们平时不使用 UCOSIII 时的中断服务程序。

(4) 退出中断服务函数的时候调用 OSIntExit()，发起一次中断级任务切换。

8.1.2 直接发布和延迟发布

相比 UCOSII，UCOSIII 对从中断发布消息或者信号的处理有两种模式：直接发布和延迟发布两种方式。我们可以通过宏 OS_CFG_ISR_POST_DEFERRED_EN 来选择使用直接发布还是延迟发布。宏 OS_CFG_ISR_POST_DEFERRED_EN 在 os_cfg.h 文件中有定义，当定义为 0 时使用直接发布模式，定义为 1 的时候使用延迟发布模式。不管使用那种方式，我们的应用程序不需要做出任何的修改，编译器会根据不同的设置编译相应的代码。

1、直接发布

在 UCOSII 中使用的就是直接发布，直接发布如图 8.1.1 所示。

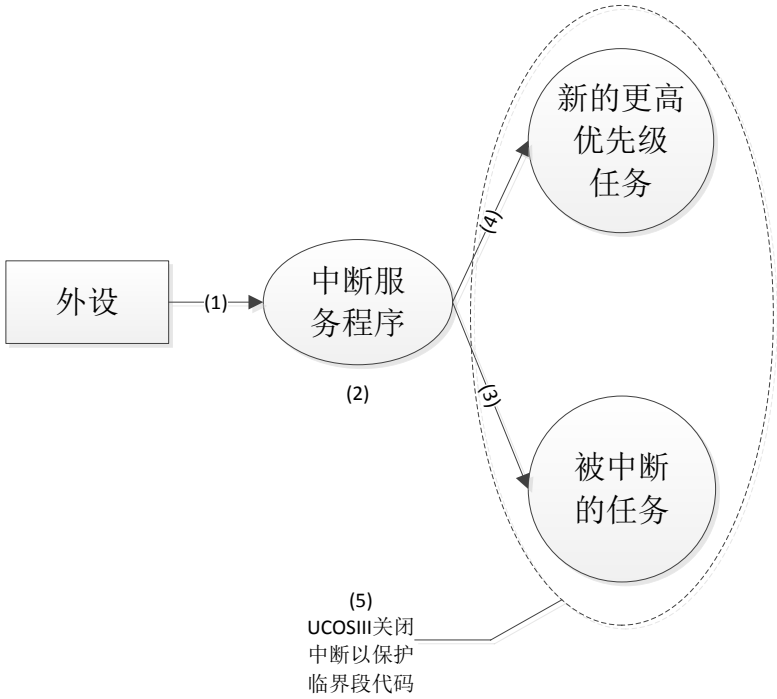


图 8.1.1 直接发布模式

(1) 外设产生中断请求。

(2) 中断服务程序开始运行，该中断服务程序中可能会包含有发送信号量、消息、事件标志组等事件。那么等待这个事件的任务的优先级要么比当前被中断的任务高，要么比其低。

(3) 如果中断对应的事件使得某个比被中断的任务优先级低的任务进入就绪态，则中断退出后仍恢复运行被中断的任务。

(4) 如果中断对应的事件使得某个比被中断的任务优先级更高的任务进入就绪态，则 UCOSIII 将进行任务切换，中断服务程序推出后就执行更高优先级的任务。

(5) 如果使用直接发布模式的话，则 UCOSIII 必须关中断以保护临界段代码，防止中断处理程序访问这些临界段代码。

使用直接发布模式的话，UCOSIII 会对临界段代码采用关闭中断的保护措施，这样就会延长中断的响应时间。虽然 UCOSIII 已经采用了所有可能的措施来降低中断关闭时间，但仍然有一些复杂的功能会使得中断关闭相对较长的时间。

2、延迟发布

当设置宏 OS_CFG_ISR_POST_DEFERRED_EN 为 1 的时候，UCOSIII 不是通过关中断，而是通过给任务调度器上锁的方法来保护临界段代码，在延迟发布模式下基本不存在关闭中断的情况，延迟发布如图 8.1.2 所示。

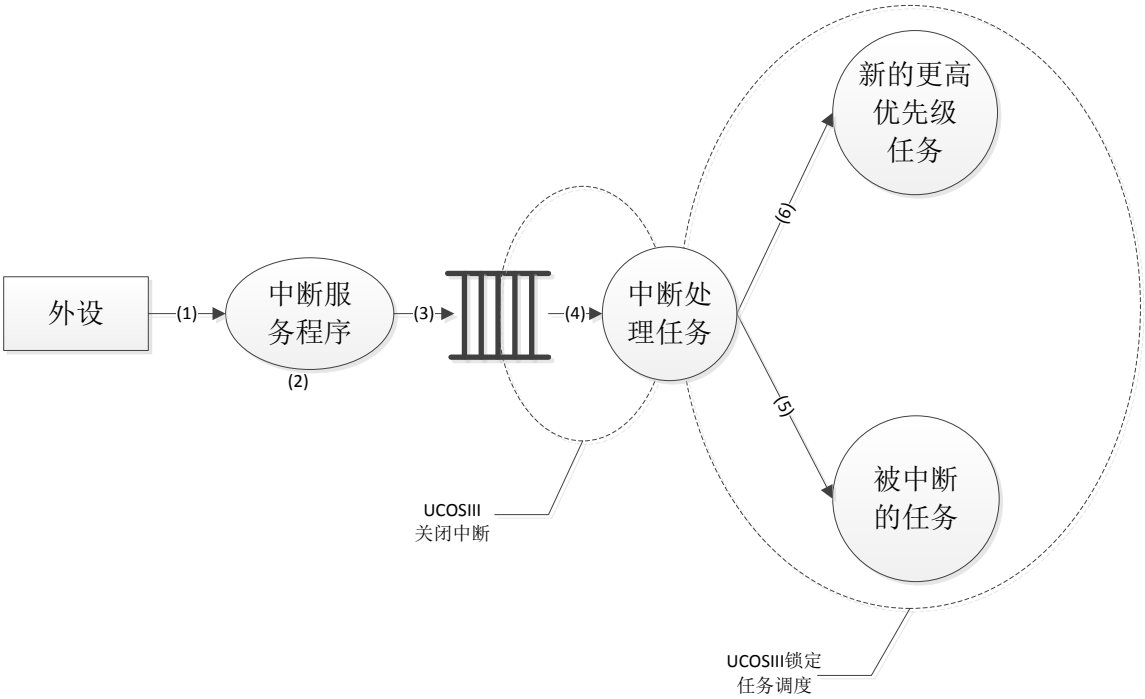


图 8.1.2 延迟发布模式

(1) 外设产生中断请求。

(2) 中断服务程序开始运行，该中断服务程序中可能会包含有发送信号量、消息、事件标志组等事件。那么等待这个事件的任务的优先级要么比当前被中断的任务高，要么比其低。

(3) 中断服务程序通过调用系统的发布服务函数向任务发布消息或信号，在延迟发布模式下，这个过程不是直接进行发布操作，而是将这个发布函数调用和相应的参数写入到专用队列中，该队列称为中断队列。然后使中断队列处理任务进入就绪态,这个任务是 UCOSIII 的内部任务，并且具有最高优先级(0)。

(4) 中断服务程序处理结束时，UCOSIII 切换执行中断队列处理任务，该任务从中断队列中提取出发布函数调用信息，此时仍需要关闭中断以防止中断服务程序同时对中断队列进行访问。中断队列处理任务提取出发布函数调用的信息后重新开中断，锁定任务调度器，然后进行发布函数调用，相当于发布函数调用一直是在任务级代码中进行的，这样本来应该在临界段中处理的代码就被放到了任务级完成。

(5) 中断队列处理任务将中断队列处理完，将自身挂起，并重新启动任务调度来运行处于最高优先级的就绪任务。如果原先被中断的任务仍然是最高优先级的就绪任务，则 UCOSIII 恢复运行这个任务。

(6) 由于中断队列处理任务的发布操作使得更重要的任务进入就绪态，内核将切换到更高优先级的任务运行。

在使用延迟发布模式额外增加的操作都是为了避免使用关中断来保护临界段代码。这些额外增加的操作仅包括将发布调用及其参数复制到中断队列中、从中断队列提取发布调用和相关参数以及一次额外的任务切换。

3、直接发布和延迟发布的对比

直接发布模式下，UCOSIII 通过关闭中断来保护临界段代码。延迟发布模式下，UCOSIII 通过锁定任务调度来保护临界段代码。

在延迟发布模式下，UCOSIII 在访问中断队列时，仍然需要关闭中断，但这个时间是非常短的。

如果应用中存在非常快速的中断请求源，则当 UCOSIII 在直接发布模式下的中断关闭时间不能满足要求的时候，可以使用延迟发布模式来降低中断关闭时间。

8.1.3 OSTimeTick()函数

就像人的心脏一样，UCOSIII 需要一个系统时钟节拍，作为系统心跳，这个时钟我们一般都使用 MCU 的硬件定时器。Cortex-M 内核提供了一个定时器用于产生系统时钟节拍，这个定时器就是 Systick。UCOSIII 通过时钟节拍来对任务进行整个节拍的延迟，并为等待事件的任务提供超时判断。时钟节拍中断必须调用 OSTimeTick()函数，我们使用 Systick 来为系统提供时钟，因此在 Systick 的中断服务程序中就必须调用 OSTimeTick()，函数代码如下：

```
void    OSTimeTick (void)
{
    OS_ERR    err;
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    CPU_TS    ts;
#endif
    OSTimeTickHook();                                                    (1)
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    t=OS_TS_GET();
    OS_IntQPost((OS_OBJ_TYPE     ) OS_OBJ_TYPE_TICK,                          (2)
                (void*           )&OSRdyList[OSPrioCur],
                (void*           ) 0,
                (OS_MSG_SIZE     ) 0u,
                (OS_FLAGS        ) 0u,
                (OS_OPT          ) 0u,
                (CPU_TS          ) ts,
                (OS_ERR*         )&err);
#else
   (void)OSTaskSemPost((OS_TCB*)&OSTickTaskTCB,                          (3)
                       (OS_OPT    ) OS_OPT_POST_NONE,
                       (OS_ERR *)&err);
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);                           (4)
#endif
#if OS_CFG_TMR_EN > 0u
    OSTmrUpdateCtr--;
    if (OSTmrUpdateCtr == (OS_CTR)0u) {
        OSTmrUpdateCtr = OSTmrUpdateCnt;
        OSTaskSemPost((OS_TCB*)&OSTmrTaskTCB,                            (5)
                      (OS_OPT    ) OS_OPT_POST_NONE,
                      (OS_ERR *)&err);
    }
#endif
#endif
```

```
}
```

(1) 时钟节拍中断服务程序中首先会调用钩子函数 OSTimeTickHook()，这个函数中用户可以放置一些代码。

(2) 如果使用了延迟发布模式，则 UCOSIII 读取当前的时间戳信息，并在中断队列中放入发布函数调用请求和相关参数，延迟向时钟节拍任务发信号的操作。然后，中断队列处理任务根据中断队列向时钟节拍任务发信号。

(3) 向时钟节拍任务(OS_TickTask())发送一个信号量。

(4) 如果 UCOSIII 使用了时间片轮转调度机制，判断当前任务分配的运行时间片是否已经用完。

(5) 如果使用定时器的话就向定时器任务(OS_TmrTask())发送信号量。

8.1.4 临界段代码保护

有一些代码我们需要保证其完成运行，不能被打断，这些不能被打断的代码就是临界段代码，也叫临界区。我们在进入临界段代码的时候使用宏 OS_CRITICAL_ENTER()，退出临界区的时候使用宏 OS_CRITICAL_EXIT()或者 OS_CRITICAL_EXIT_NO_SCHED()。

当宏 OS_CFG_ISR_POST_DEFERRED_EN 定义为 0 的时候，进入临界区的时候 UCOSIII 会使用关中断的方式，退出临界区以后重新打开中断。当 OS_CFG_ISR_POST_DEFERRED_EN 定义为 1 的时候进入临界区前是给调度器上锁，并在退出临界区的时候给调度器解锁。进入和退出临界段的宏在 os.h 文件中有定义，代码如下：

```
//采用调度器加锁的方式保护临界段代码区
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u                                    (1)
#define  OS_CRITICAL_ENTER()                                    \       (2)
         do {                                                   \
             CPU_CRITICAL_ENTER();                              \
             OSSchedLockNestingCtr++;                           \
             if (OSSchedLockNestingCtr == 1u) {                 \
                 OS_SCHED_LOCK_TIME_MEAS_START();               \
             }                                                  \
             CPU_CRITICAL_EXIT();                               \
         } while (0)
#define  OS_CRITICAL_EXIT()                                     \       (3)
         do {                                                   \
             CPU_CRITICAL_ENTER();                              \
             OSSchedLockNestingCtr--;                           \
             if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {  \
                 OS_SCHED_LOCK_TIME_MEAS_STOP();                \
                 if (OSIntQNbrEntries > (OS_OBJ_QTY)0) {        \
                     CPU_CRITICAL_EXIT();                       \
                     OS_Sched0();                               \
                 } else {                                       \
                     CPU_CRITICAL_EXIT();                       \
                 }                                              \
             } else {                                           \
```

```
                CPU_CRITICAL_EXIT();                    \
            }                                           \
        } while (0)

#define  OS_CRITICAL_EXIT_NO_SCHED()                    \                    (4)
        do {                                            \
            CPU_CRITICAL_ENTER();                       \
            OSSchedLockNestingCtr--;                    \
            if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {   \
                OS_SCHED_LOCK_TIME_MEAS_STOP();         \
            }                                           \
            CPU_CRITICAL_EXIT();                        \
        } while (0)
#else                                                                        (5)
//采用关中断的方式保护临界代码区
#define  OS_CRITICAL_ENTER()                CPU_CRITICAL_ENTER()
#define  OS_CRITICAL_EXIT()                 CPU_CRITICAL_EXIT()
#define  OS_CRITICAL_EXIT_NO_SCHED()        CPU_CRITICAL_EXIT()
```

(1) 如果宏 OS_CFG_ISR_POST_DEFERRED_EN 大于 0，那么就采用调度器上锁的方式来保护临界段代码。

(2) 采用调度器加锁的方式保护临界代码区，因为 OSSchedLockNestingCtr 是全局变量，我们在访问全局资源的时候一定要加保护，这里使用宏 CPU_CRITICAL_ENTER()来保护 OSSchedLockNestingCtr，当给 OSSchedLockNestingCtr 加 1，也就是调度器上锁以后再调用宏 CPU_CRITICAL_EXIT()退出中断。注意这里仅仅是因为要操作全局资源 OSSchedLockNestingCtr 才会关闭和打开中断，真正对于临界段代码的保护还是采用的调度器加锁的方式！

(3) 退出临界段，调度器解锁，其实就是对 OSSchedLockNestingCtr 做减一操作。

(4) 也是退出临界段，不过使用这个宏的话在退出临界段的时候不会进行任务调度。

(5) 如果宏 OS_CFG_ISR_POST_DEFERRED_EN 等于 0 的话，说明对于临界段代码的保护采用的是关闭中断的方式。这里又有两个宏 CPU_CRITICAL_ENTER 和 CPU_CRITICAL_EXIT，这两个宏最终调用的还是函数 CPU_SR_Save()和 CPU_SR_Restore()，这两个函数我们前面介绍过，就是使用汇编实现的关闭和打开中断，在 cpu_a.asm 文件中有定义。

8.2 时间管理

8.2.1 OSTimeDly()函数

当我们需要对一个任务进行延时操作的时候就可以使用这个函数，函数原型如下。

```
void  OSTimeDly (OS_TICK     dly,OS_OPT      opt,OS_ERR     *p_err)
```

dly： 指定延时的时间长度，这里单位为时间节拍数。

opt： 指定延迟使用的选项，有四种选项。

OS_OPT_TIME_DLY	相对模式
OS_OPT_TIME_TIMEOUT	和 OS_OPT_TIME_DLY 一样
OS_OPT_TIME_MATCH	绝对模式

OS_OPT_TIME_PERIODIC 周期模式

p_err： 指向调用该函数后返回的错误码

“相对模式”在系统负荷较重时有可能延时会少一个节拍，甚至偶尔差多个节拍，在周期模式下，任务仍然可能会被推迟执行，但它总会和预期的“匹配值”同步。因此，推荐使用“周期模式”来实现长时间运行的周期性延时。

“绝对模式”可以用来在上电后指定的时间执行具体的动作，比如可以规定，上电 N 秒后关闭某个外设。

8.2.2 OSTimeDlyHMSM()函数

我们也可调用 OSTimeDlyHMSM()函数来更加直观的来对某个任务延时，OSTimeDlyHMSM()函数原型如下：

```
void  OSTimeDlyHMSM (CPU_INT16U   hours,        //需要延时的小时数
                     CPU_INT16U   minutes,      //需要延时的分钟数
                     CPU_INT16U   seconds,      //需要延时的秒数
                     CPU_INT32U   milli,        //需要延时的毫秒数
                     OS_OPT       opt,          //选项
                     OS_ERR       *p_err)
```

hours

minutes

seconds

milli： 前面这四个参数用来设置需要延时的时间，使用的是：小时、分钟、秒和毫秒这种格式，这个就比较直观了，这个延时最小单位和我们设置的时钟节拍频率有关，比如我们设置时钟节拍频率 OS_CFG_TICK_RATE_HZ 为 200 的话，那么最小延时单位就是 5ms。

opt： 相比 OSTimeDly()函数多了两个选项 OS_OPT_TIME_HMSM_STRICT 和 OS_OPT_TIME_HMSM_NON_STRICT，其他四个选项都一样的。

使用 OS_OPT_TIME_HMSM_NON_STRICT 选项的话将会检查延时参数，hours 的范围应该是 0~99，minutes 的范围应该是 0~59，seconds 的范围为 0~59，milli 的范围为 0~999。

使用 OS_OPT_TIME_HMSM_NON_STRICT 选项的话，hours 的范围为 0~999，minutes 的范围为 0~9999，seconds 的范围为 0~65535，mili 的范围为 0~4294967259。

p_err： 调用此函数后返回的错误码

8.2.3 其他有关时间函数

1、OSTimeDlyResume()函数

一个任务可以通过调用这个函数来“解救”那些因为调用了 OSTimeDly()或者 OSTimeDlyHMSM()函数而进入等待态的任务，函数原型如下：

```
void  OSTimeDlyResume (OS_TCB   *p_tcb,OS_ERR   *p_err)
```

p_tcb： 需要恢复的任务的任务控制块。

p_err： 指向调用这个函数后返回的错误码。

2、OSTimeGet()和 OSTimeSet()函数

OSTimeGet()函数用来获取当前时钟节拍计数器的值。OSTimeSet()函数可以设置当前时钟节拍计数器的值，这个函数谨慎使用。

第九章 UCOSIII 软件定时器

在学习单片机的时候会使用定时器来做很多定时任务，这个定时器是单片机自带的，也就是硬件定时器，在 UCOSIII 中提供了软件定时器，我们可以使用这些软件定时器完成一些功能，本章我们就讲解一下 UCOSIII 的软件定时器，本章分为以下几个部分。

9.1 定时器工作模式

9.2 UCOSIII 定时器实验

9.1 定时器工作模式

定时器其实就是一个递减计数器，当计数器递减到 0 的时候就会触发一个动作，这个动作就是回调函数，当定时器计时完成时就会自动的调用这个回调函数。因此我们可以使用这个回调函数来完成一些设计。比如，定时 10 秒后打开某个外设等等，在回调函数中应避免任何可以阻塞或者删除定时任务的函数。如果要使用定时器的话需要将宏 OS_CFG_TMR_DEL_EN 定义为 1。定时器的分辨率由我们定义的系统节拍频率 OS_CFG_TICK_RATE_HZ 决定，比如我们定义为 200，系统时钟周期就是 5ms，定时器的最小分辨率肯定就是 5ms。但是定时器的实际分辨率是通过宏 OS_CFG_TMR_TASK_RATE_HZ 定义的，这个值绝对不能大于 OS_CFG_TICK_RATE_HZ。比如我们定义 OS_CFG_TMR_TASK_RATE_HZ 为 100，则定时器的时间分辨率为 10ms。有关 UCOSIII 定时器的函数都在 os_tmr.c 文件中。

9.1.1 创建一个定时器

如果我们要使用定时器，肯定需要先创建一个定时器，使用 OSTmrCreate()函数来创建一个定时器，这个函数也用来确定定时器的运行模式，OSTmrCreate()函数原型如下：

```
void  OSTmrCreate (OS_TMR                *p_tmr,
                   CPU_CHAR              *p_name,
                   OS_TICK               dly,
                   OS_TICK               period,
                   OS_OPT                opt,
                   OS_TMR_CALLBACK_PTR   p_callback,
                   void                  *p_callback_arg,
                   OS_ERR                *p_err)
```

p_tmr： 指向定时器的指针，宏 OS_TMR 是一个结构体。
p_name: 定时器名称。
dly: 初始化定时器的延迟值。
period: 重复周期。
opt: 定时器运行选项，这里有两个模式可以选择。
OS_OPT_TMR_ONE_SHOT 单次定时器
OS_OPT_TMR_PERIODIC 周期定时器
p_callback: 指向回调函数的名字。
p_callback_arg: 回调函数的参数。
p_err: 调用此函数以后返回的错误码。

9.1.2 单次定时器

使用 OSTmrCreate()函数创建定时器时把参数 opt 设置为 OS_OPT_TMR_ONE_SHOT，就是创建的单次定时器。创建一个单次定时器以后，我们一旦调用 OSTmrStart()函数定时器就会从创建时定义的 dly 开始倒计数，直到减为 0 调用回调函数。如图 9.1.1 所示。

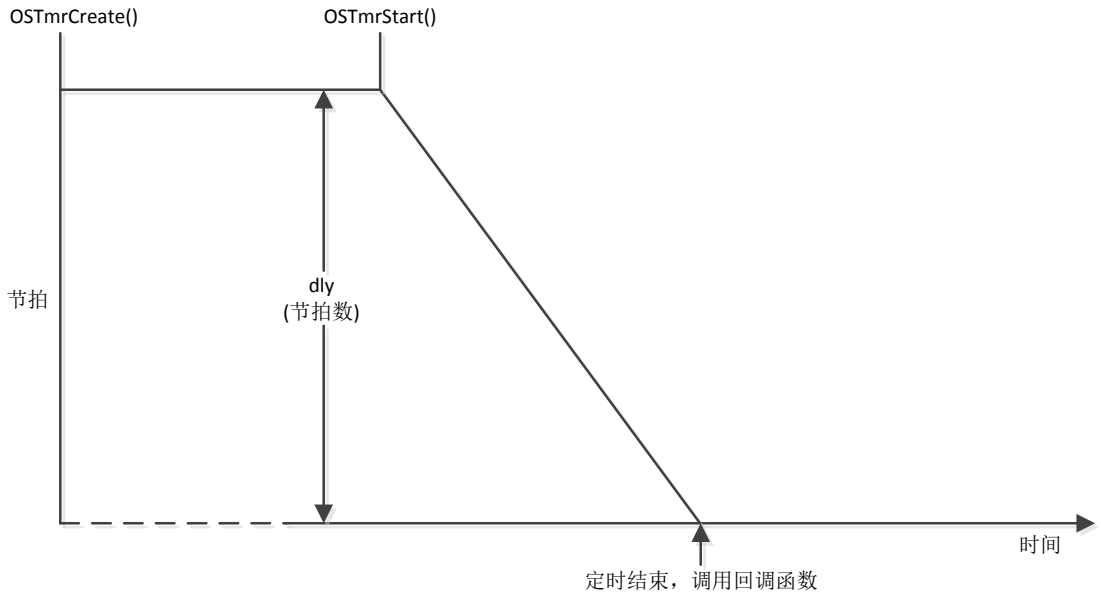


图 9.1.1 单次定时器

图 7.3.1 展示了单次定时器在调用 OSTmrStart()函数后开始倒计数，将 dly 减为 0 后调用回调函数的过程，到这里定时器就停止运行，不再做任何事情了，我们可以调用 OSTmrStop()函数来删除这个运行完成的定时器。其实我们也可以重新调用 OSTmrStart()函数来开启一个已经运行完成的定时器，通过调用 OSTmrStart()函数来重新触发单次定时器，如果图 9.1.2 所示。

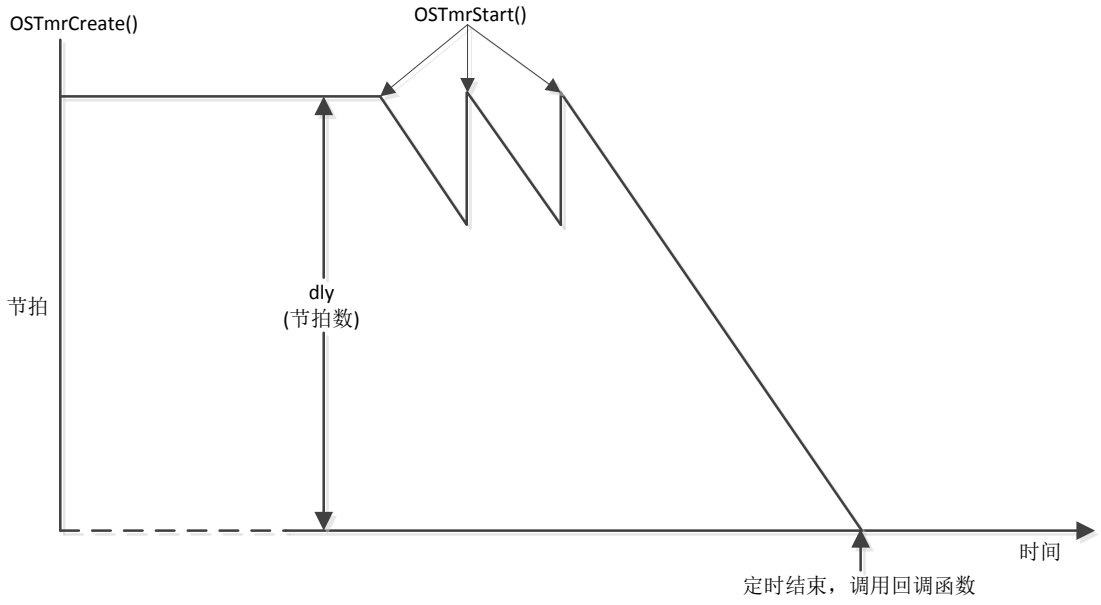


图 9.1.2 重新触发一次单次定时器

9.1.3 周期定时器(无初始化延迟)

使用 OSTmrCreate()函数创建定时器时把参数 opt 设置为 OS_OPT_TMR_PERIODIC，就是创建的周期定时器。 当定时器倒计数完成后，定时器就会调用回调函数，并且重置计数器开始下一轮的定时，就这样一直循环下去。如果使用 OSTmrCreate()函数创建定时器的时候，参数 dly 为 0 的话，那么定时器在每个周期开始时计数器的初值就为 period，如图 9.1.3 所示。

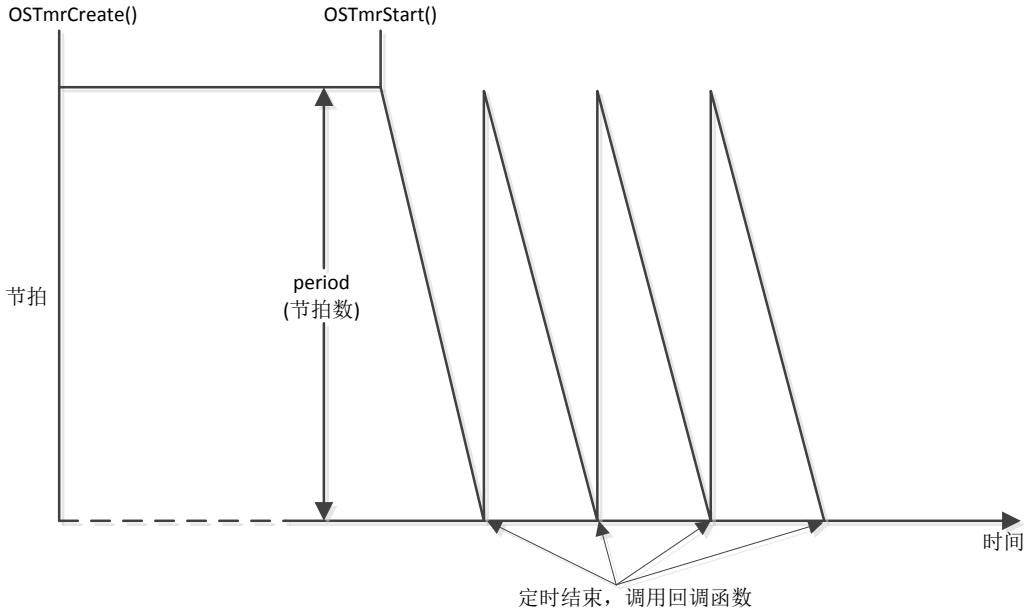


图 9.1.3 周期定时器 (dly=0,period>0)

9.1.4 周期定时器(有初始化延迟)

在创建定时器的时候也可以创建带有初始化延时的，初始化延时就是 OSTmrCreate()函数中的参数 dly 就是初始化延迟，定时器的第一个周期就是 dly。当第一个周期完成后就是用参数 period 作为周期值，调用 OSTmrStart()函数开启有初始化延时的定时器，如图 9.1.4 所示。

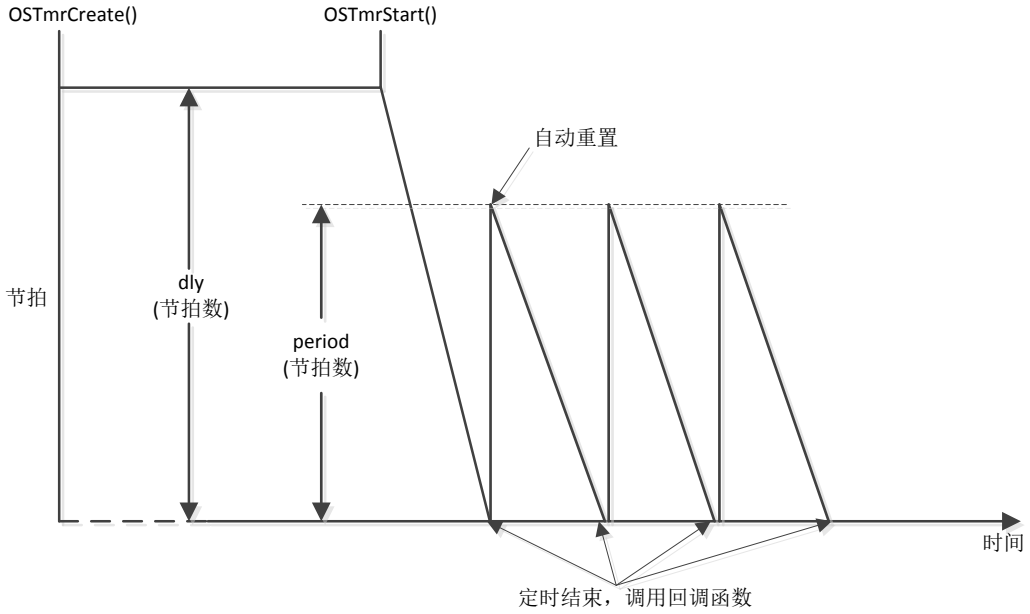


图 9.1.4 周期定时器(dly>0,period>0)

9.2 UCOSIII 定时器实验

9.2.1 实验程序设计

例 9-1：本实验新建两个任务：任务 A 和任务 B，任务 A 用于创建两个定时器：定时器 1 和定时器 2，任务 A 还创建了另外一个任务 B。其中定时器 1 为周期定时器，初始延时为 200ms，以后的定时器周期为 1000ms，定时器 2 为单次定时器，延时为 2000ms。

任务 B 作为按键检测任务，当 KEY_UP 键按下的时候，打开定时器 1；当 KEY0 按下的时候打开定时器 2；当 KEY1 按下的时候，同时关闭定时器 1 和 2；任务 B 还用来控制 LED0,使其闪烁，提示系统正在运行。

定时器 1 定时完成以后调用回调函数刷新其工作区域的背景，并且在 LCD 上显示定时器 1 运行的次数。定时器 2 定时完成后也调用其回调函数来刷新其工作区域的背景，并且显示运行次数，由于定时器 2 是单次定时器，我们通过串口打印来观察单次定时器的运行情况。

答：实验关键代码如下，实验完整工程见“例 9-1 UCOSIII 软件定时器实验”，这里主要讲解 main.c 文件。

首先是要定义两个定时器，OS_TMR 是一个结构体，代码如下：

```
OS_TMR      tmr1;           //定时器 1
OS_TMR      tmr2;           //定时器 2
```

接下来我们看一下 main 函数，main 函数比较简单，代码如下：

```
//主函数
int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);  //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);    //串口初始化

    LED_Init();             //LED 初始化
    LCD_Init();             //LCD 初始化
    KEY_Init();             //按键初始化

    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,30,200,16,16,"UCOSIII Examp 8-1");
    LCD_ShowString(30,50,200,16,16,"KEY_UP:Start Tmr1");
    LCD_ShowString(30,70,200,16,16,"KEY0:Start Tmr2");
    LCD_ShowString(30,90,200,16,16,"KEY1:Stop Tmr1 and Tmr2");

    LCD_DrawLine(0,108,239,108);        //画线
    LCD_DrawLine(119,108,119,319);      //画线

    POINT_COLOR = BLACK;
```

```
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线

    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);     //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,       "TIMER1:000");
    LCD_ShowString(126,111,110,16,16,"TIMER2:000");

    OSInit(&err);                  //初始化 UCOSIII
    OS_CRITICAL_ENTER();  //进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB *          )&StartTaskTCB,
                 (CPU_CHAR*         )"start task",
                 (OS_TASK_PTR       )start_task,
                 (void*             )0,
                 (OS_PRIO           )START_TASK_PRIO,
                 (CPU_STK *         )&START_TASK_STK[0],
                 (CPU_STK_SIZE      )START_STK_SIZE/10,
                 (CPU_STK_SIZE      )START_STK_SIZE,
                 (OS_MSG_QTY        )0,
                 (OS_TICK           )0,
                 (void *            )0,
                 (OS_OPT            )OS_OPT_TASK_STK_CHK|\
                                     OS_OPT_TASK_STK_CLR,
                 (OS_ERR *          )&err);
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);          //开启 UCOSIII
}
```

在 main 函数中我们主要完成了外设的初始化，在 LCD 上显示一些提示信息，绘制定时器 1 和定时器 2 的工作区域等。在 main 函数中我们还调用 OSTaskCreate()函数创建了一个 start_task 任务。start_task 任务函数如下。

```
//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
   OSStatTaskCPUUsageInit(&err);   //统计任务
#endif
```

```
#ifdef CPU_CFG_INT_DIS_MEAS_EN          //如果使能了测量中断关闭时间
    CPU_IntDisMeasMaxCurReset();
#endif

#if   OS_CFG_SCHED_ROUND_ROBIN_EN   //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
#endif
        //创建定时器 1
    OSTmrCreate((OS_TMR*        )&tmr1,        //定时器 1                    (1)
                (CPU_CHAR*      )"tmr1",       //定时器名字
                (OS_TICK        )20,           //20*10=200ms
                (OS_TICK        )100,          //100*10=1000ms
                (OS_OPT         )OS_OPT_TMR_PERIODIC, //周期模式
                (OS_TMR_CALLBACK_PTR)tmr1_callback,    //定时器 1 回调函数
                (void*          )0,            //参数为 0
                (OS_ERR*        )&err);        //返回的错误码
    //创建定时器 2
    OSTmrCreate((OS_TMR*        )&tmr2,                                    (2)
                (CPU_CHAR*      )"tmr2",
                (OS_TICK        )200,          //200*10=2000ms
                (OS_TICK        )0,
                (OS_OPT         )OS_OPT_TMR_ONE_SHOT,  //单次定时器
                (OS_TMR_CALLBACK_PTR)tmr2_callback,//定时器 2 回调函数
                (void*          )0,
                (OS_ERR*        )&err);

    OS_CRITICAL_ENTER();   //进入临界区
    //创建 TASK1 任务
    OSTaskCreate((OS_TCB *          )&Task1_TaskTCB,                       (3)
                 (CPU_CHAR*         )"Task1 task",
                 (OS_TASK_PTR       )task1_task,
                 (void*             )0,
                 (OS_PRIO           )TASK1_TASK_PRIO,
                 (CPU_STK*          )&TASK1_TASK_STK[0],
                 (CPU_STK_SIZE      )TASK1_STK_SIZE/10,
                 (CPU_STK_SIZE      )TASK1_STK_SIZE,
                 (OS_MSG_QTY        )0,
                 (OS_TICK           )0,
                 (void*             )0,
                 (OS_OPT            )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR*           )&err);
```

```
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err);  //删除 start_task 任务自身
}
```

(1) 这里我们使用 OSTmrCreate()函数创建一个软件定时器 1，定时器 1 为周期定时器，初始延时为 200ms，周期为 1000ms，定时器 1 对应的回调函数为 tmr1_callback()。

(2) 创建定时器 2，定时器 2 为单次定时器，初始延时为 2000ms，定时器 2 的回调函数为 tmr2_callback()。

(3) 使用 OSTaskCreate()函数创建任务 1。

定时器 1、定时器 2 的回调函数，以及任务 1 的任务函数如下。

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    u8 key,num;
    OS_ERR err;
    while(1)
    {
        key = KEY_Scan(0);
        switch(key)
        {
            case WKUP_PRES:       //当 key_up 按下的话打开定时器 1
                OSTmrStart(&tmr1,&err);   //开启定时器 1
                printf("开启定时器 1\r\n");
                break;
            case KEY0_PRES:       //当 key0 按下的话打开定时器 2
                OSTmrStart(&tmr2,&err);   //开启定时器 2
                printf("开启定时器 2\r\n");
                break;
            case KEY1_PRES:       //当 key1 按下话就关闭定时器
                OSTmrStop(&tmr1,OS_OPT_TMR_NONE,0,&err);  //关闭定时器 1
                OSTmrStop(&tmr2,OS_OPT_TMR_NONE,0,&err);  //关闭定时器 2
                printf("关闭定时器 1 和 2\r\n");
                break;
        }
        num++;
        if(num==50) //每 500msLED0 闪烁一次
        {
            num = 0;
            LED0 = ~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);    //延时 10ms
    }
}
```

```
//定时器 1 的回调函数
void tmr1_callback(void *p_tmr, void *p_arg)
{
    static u8 tmr1_num=0;
    LCD_ShowxNum(62,111,tmr1_num,3,16,0x80); //显示定时器 1 的执行次数
    LCD_Fill(6,131,114,313,lcd_discolor[tmr1_num%14]);//填充区域
    tmr1_num++;        //定时器 1 执行次数加 1
}

//定时器 2 的回调函数
void tmr2_callback(void *p_tmr,void *p_arg)
{
    static u8 tmr2_num = 0;
    tmr2_num++;        //定时器 2 执行次数加 1
    LCD_ShowxNum(182,111,tmr2_num,3,16,0x80);   //显示定时器 2 执行次数
    LCD_Fill(126,131,233,313,lcd_discolor[tmr2_num%14]); //填充区域
    LED1 = ~LED1;
    printf("定时器 2 运行结束\r\n");
}
```

这三个函数比较简单，后面都有注释的，这里就不一一解释了，不过一定要注意的是：在定时器的回调函数里面一定要注意避免使用任何可以阻塞或者删除掉定时器任务的函数！

9.2.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，一开始 LCD 如图 9.2.1 所示。定时器 1 和定时器 2 都没有打开，只有 LED0 在闪烁，提示系统正在运行。

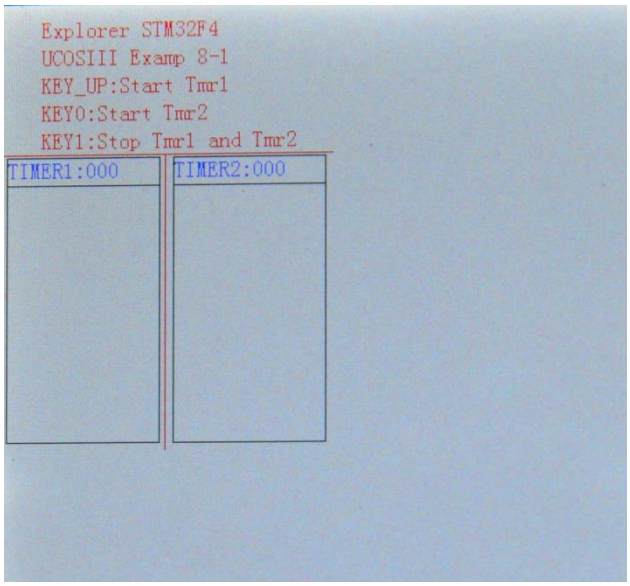


图 9.2.1 上电以后 LCD 界面

从图 9.2.1 中可以看到，此时定时器 1 和定时器 2 都没有开启，定时器 1 和 2 的工作区域背景都是白色的，并且两个定时器的运行次数都为 0，当我们按下 KEY_UP 的时候定时器 1 开始运行，此时 LCD 如图 9.2.2 所示。

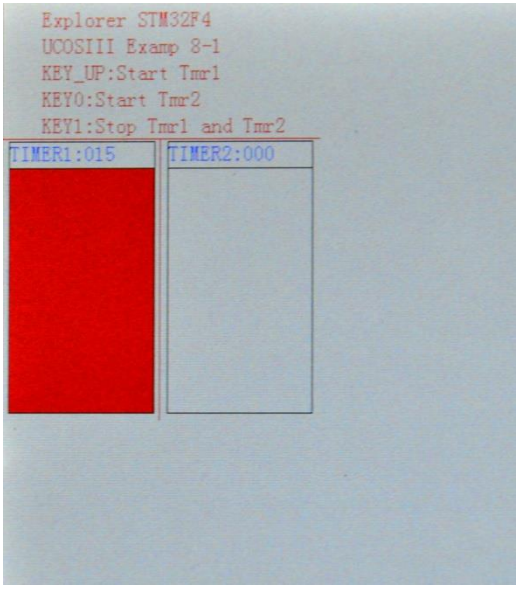


图 9.2.2 启动定时器 1 后的 LCD 界面

从图 9.2.2 中可以看出此时定时器 1 运行了 15 次，而定时器 2 为 0，因为我们根本就没开启定时器 2 ，这里要注意到的是：当我们按下 KEY_UP 键后，左边的区域并没有立即刷新为其他颜色，这是因为我们按下 KEY_UP 键后，定时器 1 开始运行，直到运行完初始化延迟时间 200ms 后才会调用定时器 1 的回调函数刷新左边区域的背景颜色，只有初始化延时为 200ms，以后的周期就是 1000ms。

按下 KEY0，开启定时器 2，等待 2000ms 后右边矩形的背景刷新为其他颜色，如图 9.2.3 所示。由于定时器 2 我们配置为单次模式，从按下按键开始等待定时器 2 计数器减到 0，就会调用一次回调函数，然后定时器 2 停止运行，除非我们再一次打开定时器 2，我们再按一下 KEY0 打开定时器 2，等待 2000ms 后右边矩形背景又被刷新为其他颜色，说明定时器 2 回调函数再一次被调用。

图 9.2.3 开启定时器 2 后的 LCD 界面。

按下 KEY1 会同时关闭定时器 1 和定时器 2，虽然定时器 2 为单次定时器，每次执行完毕后会自行关闭，但是我们这里还是会通过调用 OSTmrStop()函数来关闭定时器 2。

我们再观察串口调试助手输出的信息，如图 9.2.4 所示。我们操作的时候串口调试助手就会接收到相应的信息，大家对照这源码自行分析串口调试助手显示的信息。

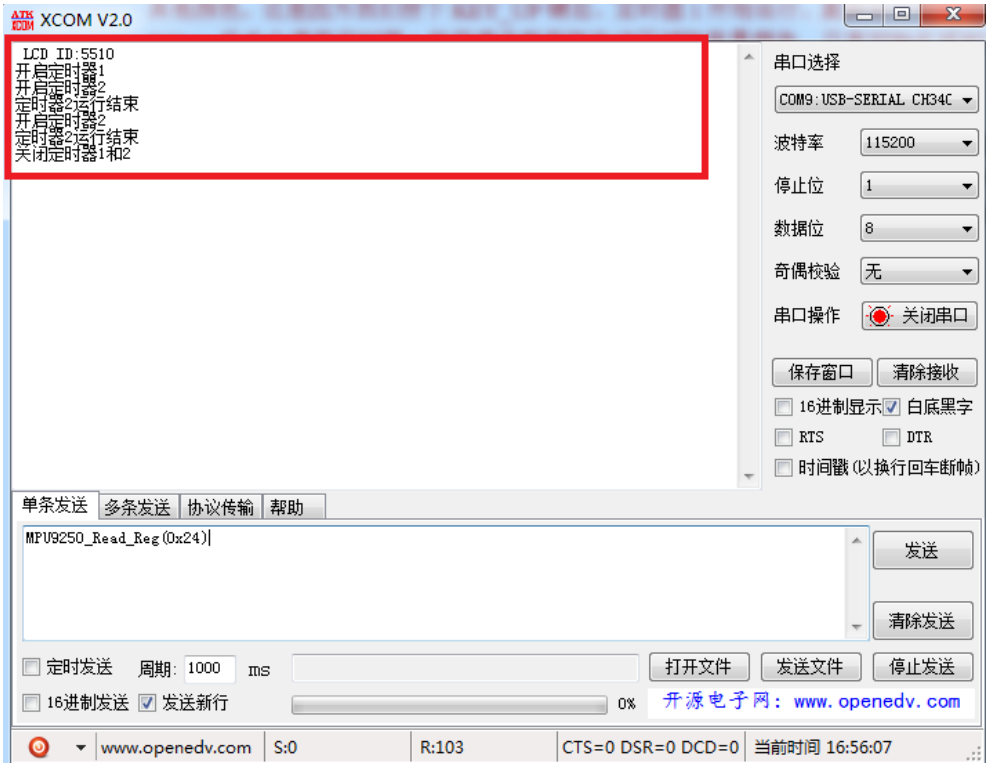


图 9.2.4 串口调试助手接收到的信息

第十章 UCOSIII 信号量和互斥信号量

在 UCOSIII 中有可能会有多个任务会访问共享资源，因此信号量最早用来控制任务存取共享资源，现在信号量也被用来实现任务间的同步以及任务和 ISR 间同步。在可剥夺的内核中，当任务独占式使用共享资源的时候，会出现低优先级的任务先于高优先级任务运行的现象，这个现象被称为优先级反转，为了解决优先级反转这个问题，UCOSIII 引入了互斥信号量这个概念。本章我们就来讲解一下 UCOSIII 的信号量和互斥信号量，本章分为如下几个部分。

10.1 信号量
10.2 优先级反转
10.3 互斥信号量
10.4 直接访问共享资源区实验
10.5 使用信号量访问共享资源区实验
10.6 任务同步实验

10.1 信号量

信号量像是一种上锁机制，代码必须获得对应的钥匙才能继续执行，一旦获得了钥匙，也就意味着该任务具有进入被锁部分代码的权限。一旦执行至被锁代码段，则任务一直等待，直到对应被锁部分代码的钥匙被再次释放才能继续执行。

信号量分为两种：二进制信号量与计数型信号量，二进制信号量只能取 0 和 1 两个值，计数型信号量不止可以取 2 个值，在共享资源中只有任何可以使用信号量，中断服务程序则不能使用。

1、二进制信号量

某一资源对应的信号量为 1 的时候，那么就可以使用这一资源，如果对应资源的信号量为 0，那么等待该信号量的任务就会被放进等待信号量的任务表中。在等待信号量的时候也可以设置超时，如果超过设定的时间任务没有等到信号量的话那么该任务就会进入就绪态。任务以“发信号”的方式操作信号量。可以看出如果一个信号量为二进制信号量的话，一次只能一个任务使用共享资源。

2、计数型信号量

有时候我们需要可以同时有多个任务访问共享资源，这个时候二进制信号量就不能使用了，计数型信号量就是用来解决这个问题的。比如某一个信号量初始化值为 10，那么只有前 10 个请求该信号量的任务可以使用共享资源，以后的任务需要等待前 10 个任务释放掉信号量。每当有任务请求信号量的时候，信号量的值就会减 1，直到减为 0。当有任务释放掉信号量的时候信号量的值就会加 1。

有关信号量的 API 函数如表 10.1.1 所示。

函数	描述
OSSemCreate()	创建一个信号量
OSSemDel()	删除一个信号量
OSSemPend()	等待一个信号量
OSSemPendAbort()	取消等待
OSSemPost()	释放一个信号量
OSSemSet()	强制设置一个信号量的值

表 10.1.1 信号量 API 函数

10.1.1 创建信号量

要想使用信号量，肯定需要先创建一个信号量，我们使用函数 OSSemCreate()来创建信号量，函数原型如下：

```
void  OSSemCreate ( OS_SEM        *p_sem,
                    CPU_CHAR      *p_name,
                    OS_SEM_CTR    cnt,
                    OS_ERR        *p_err)
```

p_sem： 指向信号量控制块，我们需要按照如下所示方式定义一个全局信号量，并将这个信号量的指针传递给函数 OSSemCreate()。

```
OS_SEM  TestSem;
```

p_name: 指向信号量的名字。

cnt： 设置信号量的初始值，如果此值为 1，代表此信号量为二进制信号量，如果大于 1 的话就代表此信号量为计数型信号量。

p_err： 保存调用此函数后的返回的错误码。

10.1.2 请求信号量

当一个任务需要独占式的访问某个特定的系统资源时，需要与其他任务或中断服务程序同步，或者需要等待某个事件的发生，应该调用函数 OSSemPend()，函数原型如下：

```
OS_SEM_CTR    OSSemPend ( OS_SEM      *p_sem,
                          OS_TICK     timeout,
                          OS_OPT      opt,
                          CPU_TS      *p_ts,
                          OS_ERR      *p_err)
```

p_sem： 指向一个信号量的指针。

timeout： 指定等待信号量的超时时间(时钟节拍数)，如果在指定时间内没有等到信号量则允许任务恢复执行。如果指定时间为 0 的话任务就会一直等待下去，直到等到信号量。

opt： 用于设置是否使用阻塞模式，有下面两个选项。

OS_OPT_PEND_BLOCKING 指定信号量无效时，任务挂起以等待信号量。

OS_OPT_PEND_NON_BLOCKING 信号量无效时，任务直接返回。

p_ts： 指向一个时间戳，用来记录接收到信号量的时刻，如果给这个参数赋值 NULL，则说明用户没有要求时间戳。

p_err： 保存调用本函数后返回的错误码。

10.1.3 发送信号量

任务获得信号量以后就可以访问共享资源了，在任务访问完共享资源以后必须释放信号量，释放信号量也叫发送信号量，使用函数 OSSemPost()发送信号量。如果没有任务在等待该信号量的话则 OSSemPost()函数只是简单的将信号量加 1，然后返回到调用该函数的任务中继续运行。如果有一个或者多个任务在等待这个信号量，则优先级最高的任务将获得这个信号量，然后由调度器来判定刚获得信号量的任务是否为系统中优先级最高的就绪任务，如果是，则系统将进行任务切换，运行这个就绪任务，OSSemPost()函数原型如下：

```
OS_SEM_CTR    OSSemPost ( OS_SEM    *p_sem,
                          OS_OPT    opt,
                          OS_ERR    *p_err)
```

p_sem： 指向一个信号量的指针

opt： 用来选择信号量发送的方式。

OS_OPT_POST_1 仅向等待该信号量的优先级最高的任务发送信号量。

OS_OPT_POST_ALL 向等待该信号量的所有任务发送信号量。

OS_OPT_POST_NO_SCHED 该选项禁止在本函数内执行任务调度操作。即使该函数使得更高优先级的任务结束挂起进入就绪状态，也不会执行任务调度，而是会在其他后续函数中完成任务调度。

p_err： 用来保存调用此函数后返回的错误码

10.2 直接访问共享资源区实验

我们前面提过信号量主要用于访问共享资源和进行任务同步，这里我们先做一个直接访问共享资源的实验，看看会带来什么后果。

10.2.1 实验程序设计

例 10-1：创建 3 个任务，任务 A 用于创建其他两个任务，任务 A 执行一次后就会被删除掉。任务 B 和任务 C 都可以访问作为共享资源 D，任务 B 和 C 对于共享资源 D 是直接访问的，观察直接访问共享资源会造成什么要的后果。

答：本实验部分源码如下，完整的工程详见：例 10-1 UCOSIII 直接访问共享资源。

首先是共享资源，这里我们设置一个数组为共享资源，任务 1 和任务 2 都可以访问这个共享资源

```
u8 share_resource[30];     //共享资源区
```

开始任务 start_task()就是创建两个任务，很简单，这里就不讲解了，我们来看一下任务 1 和任务 2 的任务函数，任务函数代码如下。

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    OS_ERR err;
    u8 task1_str[]="First task Running!";
    while(1)
    {
        printf("\r\n 任务 1:\r\n");
        LCD_Fill(0,110,239,319,CYAN);
        memcpy(share_resource,task1_str,sizeof(task1_str)); //向共享资源区拷贝数据   (1)
        delay_ms(200);
        printf("%s\r\n",share_resource);      //串口输出共享资源区数据
        LED0 = ~LED0;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);     //延时 1s
    }
}

//任务 2 的任务函数
void task2_task(void *p_arg)
{
    u8 i=0;
    OS_ERR err;
    u8 task2_str[]="Second task Running!";
    while(1)
    {
        printf("\r\n 任务 2:\r\n");
        LCD_Fill(0,110,239,319,BROWN);
        memcpy(share_resource,task2_str,sizeof(task2_str));//向共享资源区拷贝数据   (2)
        delay_ms(200);
        printf("%s\r\n",share_resource);      //串口输出共享资源区数据
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);     //延时 1s
    }
```

```
}
```

(1) 任务 1 向共享资源区拷贝数据“First task Running!”，然后延时 200ms，通过串口输出拷贝到共享资源区中的数据，既输出“First task Running!”。

(2) 任务 2 也向共享资源区中拷贝数据“Second task Running!”，同样延时 200ms，并通过串口拷贝到共享资源区中的数据，既输出“Second task Running!”。

任务 1 和任务 2 都使用了共享资源 share_resource，我们在任务 1 和任务 2 中都是使用了函数 delay_ms()进行延时，delay_ms()函数是会引起任务切换的，而我们对于共享资源的访问没有进行任何的保护，势必会造成意想不到的结果发生，我们观察一下程序的运行结果就知道了。

10.2.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，这里我们借助串口调试助手来分析一下任务 1 和任务 2 对于共享资源的使用情况，串口调试助手输出的信息如图 10.2.1 所示。

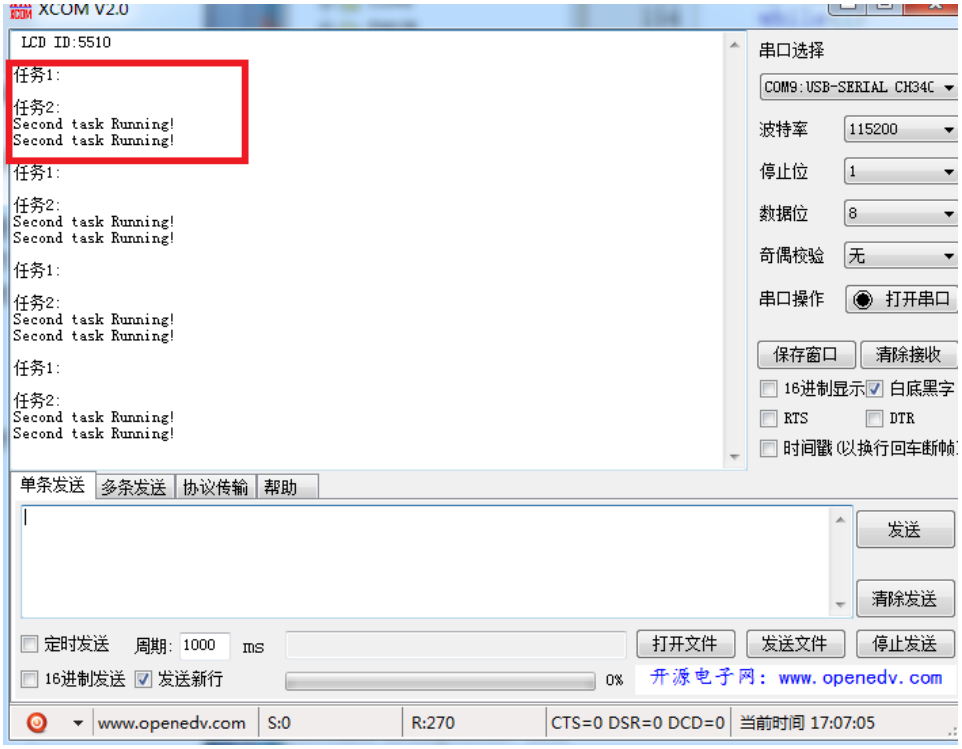


图 10.2.1 串口输出信息

从图 10.2.1 中可以看出，系统并没有按照我们想要的方式输出信息，我们想要的输出像下面的一样。

```
任务 1：
First task Running!

任务 2：
Second task Running!
```

但是现在的输出确实下面这样的：

```
任务 1:

任务 2:
Second task Running!
```

```
Second task Running!
```

我们分析一下源码，在任务 1 向 share_resource 拷贝数据“First task Running!”以后就因为 delay_ms()函数系统进行了任务切换。任务 2 开始运行，这时任务 2 又向 share_resource 拷贝了数据“Second task Running!”，任务 2 也因为 delay_ms()函数发生任务切换，任务 1 接着运行，但是这是 share_resource 已经被被修改为 “Second task Running!”因此输出就会和我们预计的不一样了，从而导致错误的发生，这个就是多任务共享资源区带来的问题！所以在任务访问共享资源区的时候我们要对其进行保护。下面我们展示一下使用信号量来保护共享资源区。

10.3 使用信号量访问共享资源区实验

在例 10-1 中我们对于 share_resource 的访问并没有进行保护，从而导致了错误的发生，这一节我们使用信号量来进行共享资源区的访问。

10.3.1 实验程序设计

例 10-2：在例 10-1 的基础上使用信号量来访问共享资源区。

答：本实验部分源码如下，完整的工程详见：例 10-2 UCOSIII 使用信号量共享资源。

这里我们要定义一个信号量，如下。

```
OS_SEM MY_SEM;          //定义一个信号量，用于访问共享资源
```

在开始任务 start_task()中调用 OSSemCreate()函数创建一个信号量，代码如下：

```
//创建一个信号量
OSSemCreate ((OS_SEM*        )&MY_SEM,      //指向信号量
             (CPU_CHAR*      )"MY_SEM",     //信号量名字
             (OS_SEM_CTR     )1,            //信号量值为 1
             (OS_ERR*        )&err);
```

任务 1 和任务 2 的代码如下：

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    OS_ERR err;
    u8 task1_str[]="First task Running!";
    while(1)
    {
        printf("\r\n 任务 1:\r\n");
        LCD_Fill(0,110,239,319,CYAN);
        OSSemPend(&MY_SEM,0,OS_OPT_PEND_BLOCKING,0,&err); //请求信号量(1)
        memcpy(share_resource,task1_str,sizeof(task1_str)); //向共享资源区拷贝数据
        delay_ms(200);
        printf("%s\r\n",share_resource);    //串口输出共享资源区数据
        OSSemPost (&MY_SEM,OS_OPT_POST_1,&err);      //发送信号量              (2)
        LED0 = ~LED0;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);     //延时 1s
    }
}
```

```
//任务 2 的任务函数
void task2_task(void *p_arg)
{
    OS_ERR err;
    u8 task2_str[]="Second task Running!";
    while(1)
    {
        printf("\r\n 任务 2:\r\n");
        LCD_Fill(0,110,239,319,BROWN);
        OSSemPend(&MY_SEM,0,OS_OPT_PEND_BLOCKING,0,&err); //请求信号量(3)
        memcpy(share_resource,task2_str,sizeof(task2_str));  //向共享资源区拷贝数据
        delay_ms(200);
        printf("%s\r\n",share_resource);   //串口输出共享资源区数据
        OSSemPost (&MY_SEM,OS_OPT_POST_1,&err);      //发送信号量          (4)
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);     //延时 1s
    }
}
```

(1) 任务 1 要访问共享资源 share_resource，因此调用函数 OSSemPend()来请求信号量。
(2) 任务 1 使用完共享资源 share_resource，调用 OSSemPost()函数释放信号量。
(3) 同(1)
(4) 同(2)

10. 3.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，这里我们借助串口调试助手来分析一下任务 1 和任务 2 对于共享资源的使用情况，串口调试助手输出的信息如图 10.3.1 所示。

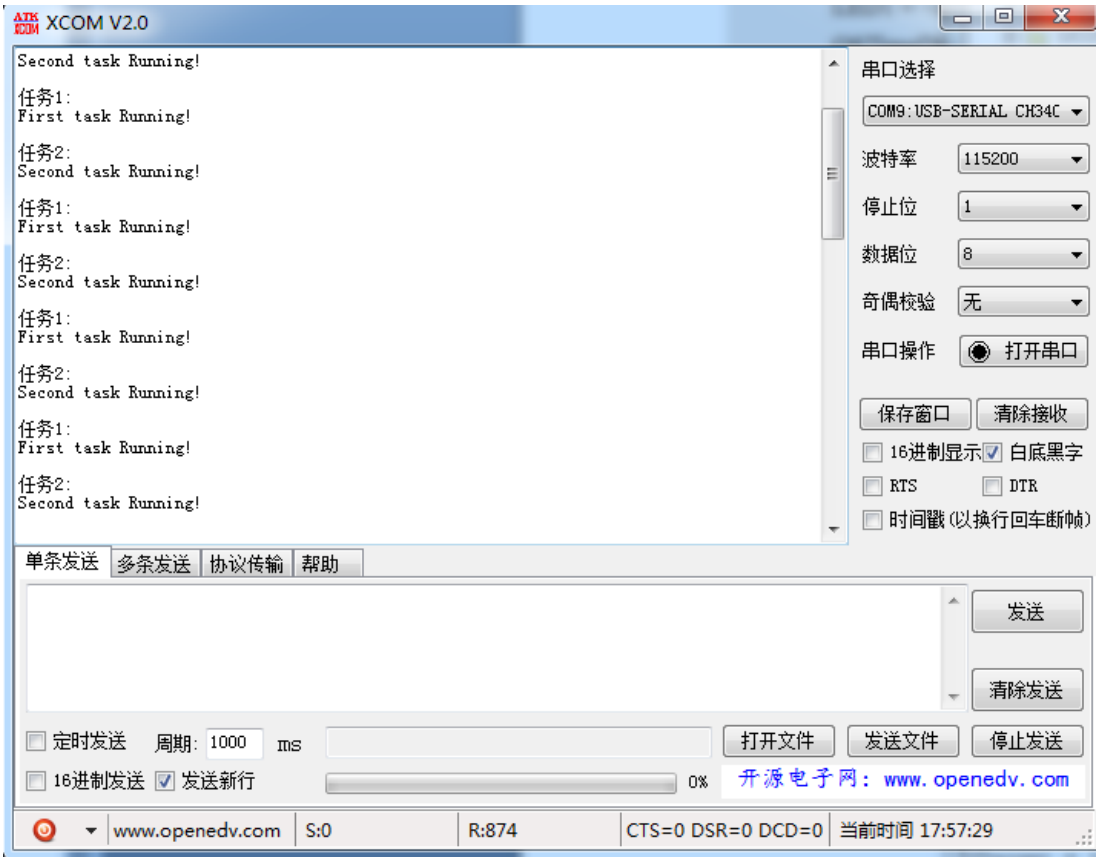


图 10.3.1 串口输出信息。

从图 10.3.1 中可以看出，串口按照我们设定的来输出信息，共享资源区并没有被其他任务随意修改！

10.4 任务同步实验

信号量现在更多的被用来实现任务的同步以及任务和 ISR 间的同步，信号量用于任务同步如图 10.4.1 所示。

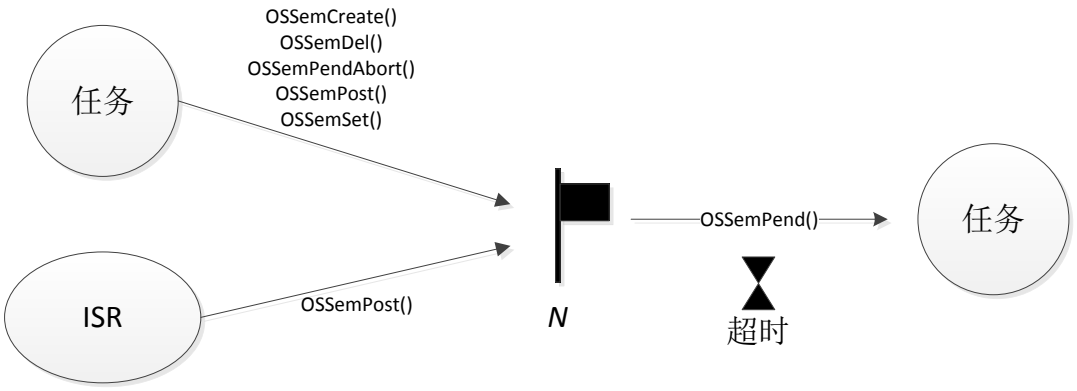


图 10.4.1 信号量用于任务同步

图 10.4.1 中用一个小旗子代表信号量，小旗子旁边的数值 N 为信号量计数值， 表示发布信号量的次数累积值，ISR 可以多次发布信号量，发布的次数会记录为 N。一般情况下，N 的初始值是 0，表示事件还没有发生过。在初始化时，也可以将 N 的初值设为大于零的某个值，

来表示初始情况下有多少信号量可用。

等待信号量的任务旁边的小沙漏表示等待任务可以设定超时时间。超时的意思是该任务只会等待一定时间的信号量，如果在这段时间内没有等到信号量，UCOSIII 就会将任务置于就绪表中，并返回错误码。

10.4.1 实验程序设计

例 10-3：创建 3 个任务，任务 A 用于创建其他两个任务和一个初始值为 0 的信号量，任务 C 必须征得任务 B 的同意才能执行一次操作。

答：这个问题显然是一个任务同步的问题，在两个任务之间设置一个初值为 0 的信号量来实现两个任务的合作。任务 B 通过发信号量表示同意与否，任务 C 一直请求信号量，当信号量大于 1 的时候任务 C 才能执行接下来的操作。本实验部分源码如下，完整的工程详见：例 10-3 UCOSIII 使用信号量进行任务同步。

定义一个信号量，用于任务同步。

```
OS_SEM      SYNC_SEM;              //定义一个信号量，用于任务同步
```

我们还需要调用函数 OSSemCreate()创建一个信号量，这个信号量的初始值为 0，如下。

```
//创建一个信号量
OSSemCreate ((OS_SEM*        )&SYNC_SEM,
             (CPU_CHAR*      )"SYNC_SEM",
             (OS_SEM_CTR     )0,
             (OS_ERR*        )&err);
```

任务 1 和任务 2 是我们本次实验的重点，这两个任务的任务函数如下。

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    u8 key;
    OS_ERR err;
    while(1)
    {
        key = KEY_Scan(0);    //扫描按键
        if(key==WKUP_PRES)
        {
            OSSemPost(&SYNC_SEM,OS_OPT_POST_1,&err);//发送信号量      (1)
            LCD_ShowxNum(150,111,SYNC_SEM.Ctr,3,16,0); //显示信号量值      (2)
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}

//任务 2 的任务函数
void task2_task(void *p_arg)
{
    u8 num;
    OS_ERR err;
```

```
    while(1)
    {
        //请求信号量
        OSSemPend(&SYNC_SEM,0,OS_OPT_PEND_BLOCKING,0,&err);        (3)
        num++;
        LCD_ShowxNum(150,111,SYNC_SEM.Ctr,3,16,0);          //显示信号量值
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);          //刷屏
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);     //延时 1s
    }
}
```

(1) 当 KEY_UP 键按下的时候调用 OSSemPost()函数发送一次信号量。

(2) 信号量 SYNC_SEM 的字段 Ctr 用来记录信号量值，我们每调用一次 OSSemPost()函数 Ctr 字段就会加一，这里我们将 Ctr 的值显示在 LCD 上，来观察 Ctr 的变化。

(3) 任务 2 请求信号量 SYNC_SEM，如果请求到信号量的话就会执行任务 2 下面的代码，如果没有请求到的话就会一直阻塞函数。当调用函数 OSSemPend()请求信号量成功的话，SYNC_SEM 的字段 Ctr 就会减一，直到为 0。在任务 2 中我们也将信号量 SYNC_SEM 的字段 Ctr 显示在 LCD 上，观察其变化。

10.4.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，由于我们新建的信号量 SYNC_SEM 的初始值为 0，因此在开机以后任务 2 会由于请求不到信号量而阻塞，此时 LCD 显示如图 10.4.1 所示。

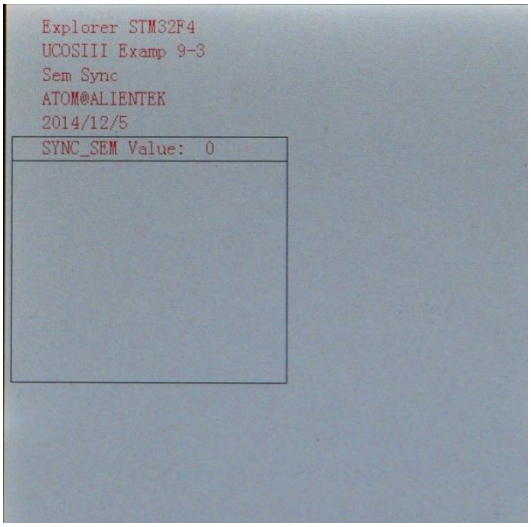


图 10.4.1 开机 LCD 显示

从图 10.4.1 中我们可以看出，由于信号量 SYNC_SEM 的初始值为 0，因此 SYNC_SEM 的信号量值显示为 0，并且任务 2 阻塞。当我们按下 KEY_UP 键以后就会发送信号量，SYNC_SEM 的值就会变化(增加)，我们多按几次 KEY_UP 键，如图 10.4.2 所示。

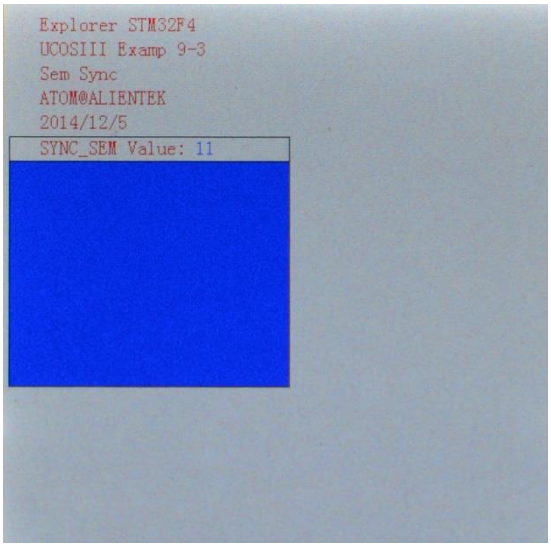


图 10.4.2 发送多次信号量后的 LCD 显示

从图 10.4.2 中可以看出，此时信号量 SYNC_SEM 的值为 11，说明任务 2 可以请求 11 次信号量 SYNC_SEM。任务 2 每隔 1s 就会请求一起信号量 SYNC_SEM，直到 SYNC_SEM 的信号量值为 0，由于任务 2 请求不到信号量了，因此任务 2 就会阻塞，此时 LCD 如图 10.4.3 所示。

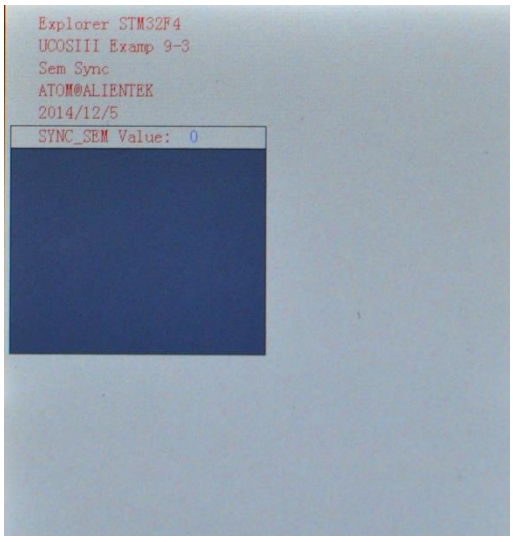


图 10.4.3 信号量减小为 0

信号量值减小到 0，任务 2 阻塞。当我们再次按下 KEY_UP 的时候，任务 2 又会接着“运行”，大家可以自行尝试一次。

10.5 优先级反转

优先级反转在可剥夺内核中是非常常见的，在实时系统中不允许出现这种现象，这样会破坏任务的预期顺序，可能会导致严重的后果，图 10.5.1 就是一个优先级反转的例子。

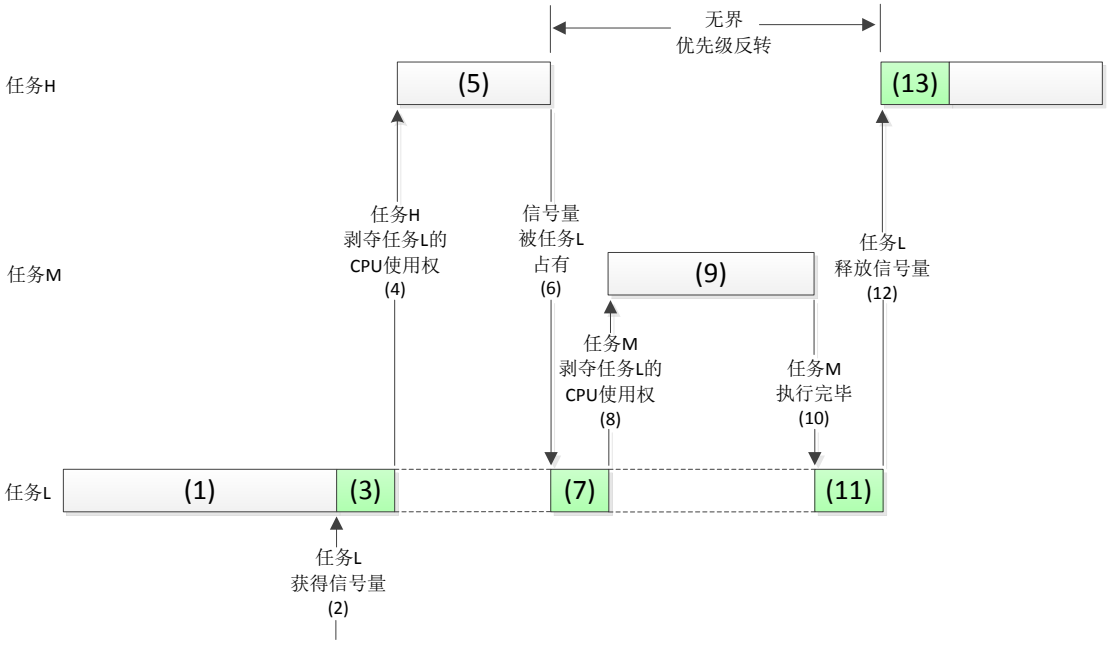


图 10.5.1 优先级反转示意图

(1) 任务 H 和任务 M 处于挂起状态，等待某一事件的发生，任务 L 正在运行。

(2) 某一时刻任务 L 想要访问共享资源，在此之前它必须先获得对应该资源的信号量。

(3) 任务 L 获得信号量并开始使用该共享资源。

(4) 由于任务 H 优先级高，它等待的事件发生后便剥夺了任务 L 的 CPU 使用权。

(5) 任务 H 开始运行。

(6) 任务 H 运行过程中也要使用任务 L 正在使用着的资源，由于该资源的信号量还被任务 L 占用着，任务 H 只能进入挂起状态，等待任务 L 释放该信号量。

(7) 任务 L 继续运行。

(8) 由于任务 M 的优先级高于任务 L，当任务 M 等待的事件发生后，任务 M 剥夺了任务 L 的 CPU 使用权。

(9) 任务 M 处理该处理的事。

(10) 任务 M 执行完毕后，将 CPU 使用权归还给任务 L。

(11) 任务 L 继续运行。

(12) 最终任务 L 完成所有的工作并释放了信号量，到此为止，由于实时内核知道有个高优先级的任务在等待这个信号量，故内核做任务切换。

(13) 任务 H 得到该信号量并接着运行。

在这种情况下，任务 H 的优先级实际上降到了任务 L 的优先级水平。因为任务 H 要一直等待直到任务 L 释放其占用的那个共享资源。由于任务 M 剥夺了任务 L 的 CPU 使用权，使得任务 H 的情况更加恶化，这样就相当于任务 M 的优先级高于任务 H，导致优先级反转。

10.6 优先级反转实验

10.6.1 实验程序设计

例 10-4：创建 4 个任务，任务 A 用于创建 B、C 和 D 这三个任务，A 还创建了一个初始值为 1 的信号量 TEST_SEM，任务 B 和 D 都请求信号量 TEST_SEM，其中任务优先级从高到底分别为：B、C、D。

答：本实验部分源码如下，完整的工程详见：例 10-4 UCOSIII 优先级反转。

首先肯定要先定义一个信号量，代码如下：

```
OS_SEM      TEST_SEM;       //定义一个信号量
```

在任务 start_task 中我们创建了一个初始值为 1 的信号量 TEST_SEM，代码如下：

```
//创建一个信号量
OSSemCreate ((OS_SEM*       )&TEST_SEM,
             (CPU_CHAR*     )"TEST_SEM",
             (OS_SEM_CTR    )1,                 //信号量初始值为 1
             (OS_ERR*       )&err);
```

high_task、middle_task 和 low_task 这三个任务函数是本次实验的重点，这三个任务函数的代码如下：

```
//高优先级任务的任务函数
void high_task(void *p_arg)
{
    u8 num;
    OS_ERR err;

    CPU_SR_ALLOC();
    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"High Task");
    OS_CRITICAL_EXIT();
    while(1)
    {
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_PERIODIC,&err);          //延时 500ms
        num++;
        printf("high task Pend Sem\r\n");
        OSSemPend(&TEST_SEM,0,OS_OPT_PEND_BLOCKING,0,&err); //请求信号量(1)
        printf("high task Running!\r\n");
        LCD_Fill(6,131,114,313,lcd_discolor[num%14]); //填充区域
        LED1 = ~LED1;
        OSSemPost(&TEST_SEM,OS_OPT_POST_1,&err);                     //释放信号量(2)
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_PERIODIC,&err);          //延时 500ms
    }
}

//中等优先级任务的任务函数
void middle_task(void *p_arg)
{
    u8 num;
```

```
    OS_ERR err;
    CPU_SR_ALLOC();

    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Middle Task");
    OS_CRITICAL_EXIT();
    while(1)
    {
        num++;
        printf("middle task Running!\r\n");
        LCD_Fill(126,131,233,313,lcd_discolor[13-num%14]); //填充区域
        LED0 = ~LED0;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);   //延时 1s
    }
}

//低优先级任务的任务函数
void low_task(void *p_arg)
{
    static u32 times;
    OS_ERR err;
    while(1)
    {
        OSSemPend(&TEST_SEM,0,OS_OPT_PEND_BLOCKING,0,&err); //请求信号量(3)
        printf("low task Running!\r\n");
        for(times=0;times<20000000;times++)                                    (4)
        {
            OSSched();                                          //发起任务调度
        }
        OSSemPost(&TEST_SEM,OS_OPT_POST_1,&err);        //释放信号量        (5)
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);   //延时 1s
    }
}
```

(1)、high_task 任务中请求信号量 TEST_SEM。

(2)、high_task 任务中释放信号量 TEST_SEM。

(3)、low_task 任务中请求信号量 TEST_SEM。

(4)、这里用来模拟 low_task 任务长时间占用信号量 TEST_SEM。

(5)、low_task 任务中释放信号量 TEST_SEM。

10.6.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，下载完以后 LCD 显示如图 10.6.1 所示。

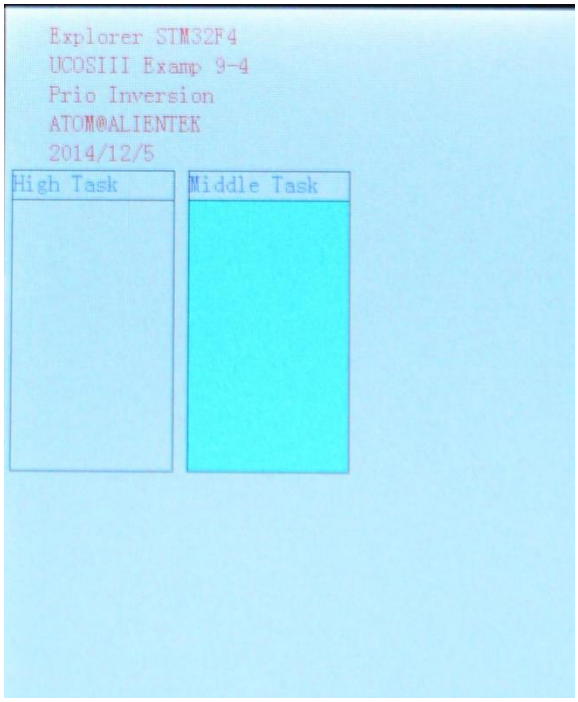


图 10.6.1 LCD 显示

从 LCD 上不容易看出优先级反转的现象，我们可以通过串口很方便的观察优先级反转，串口输出如图 10.6.2 所示。

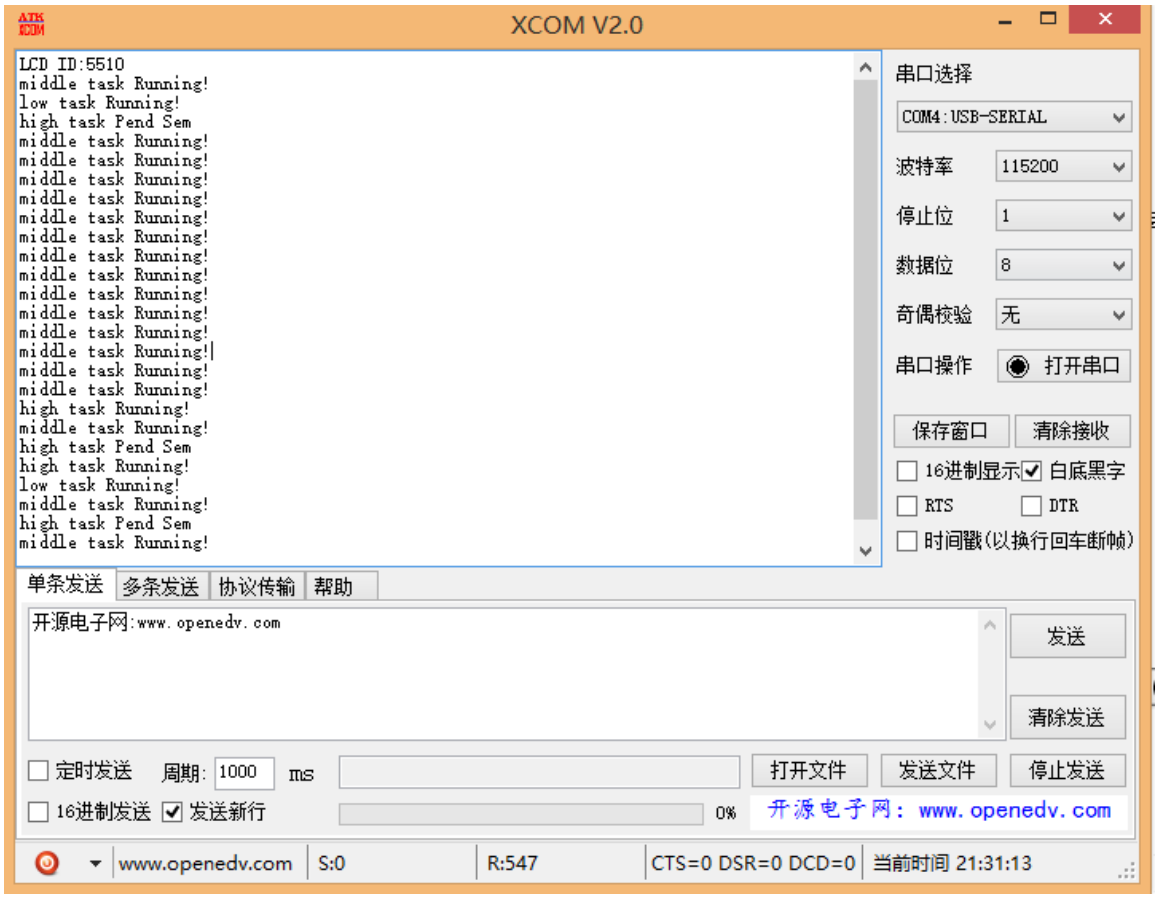


图 10.6.2 串口输出

为了方便分析，我们将串口输出复制出来，如下：

```
LCD ID:5510
middle task Running!
low task Running!                    (1)
high task Pend Sem                   (2)
middle task Running!                 (3)
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
middle task Running!
high task Running!                   (4)
```

```
middle task Running!
high task Pend Sem
high task Running!
low task Running!
middle task Running!
high task Pend Sem
```

(1)、low_task 任务获得下信号量 TEST_SEM 开始运行。

(2)、high_task 请求信号量 TEST_SEM，但是此时信号量 TEST_SEM 被任务 low_task 占用着，因此 high_task 就要一直等待，直到 low_task 任务释放信号量 TEST_SEM。

(3)、由于 high_task 没有请求到信号量 TEST_SEM，只能一直等待，红色部分代码中 high_task 没有运行，而 middle_task 一直在运行，给人的感觉就是 middle_task 的任务优先级高于 high_task。但是事实上 high_task 任务的任务优先级是高于 middle_task 的，这个就是优先级反转！

(4)、high_task 任务因为获得信号量 TEST_SEM 而运行

从例-4 中可以看出，当一个低优先级任务和一个高优先级任务同时使用同一个信号量，而系统中还有其他中等优先级任务时。如果低优先级任务获得了信号量，那么高优先级的任务就会处于等待状态，但是，中等优先级的任务可以打断低优先级任务而先于高优先级任务运行(此时高优先级的任务在等待信号量 ，所以不能运行)，这是就出现了优先级反转的现象。

10.7 互斥信号量

为了避免优先级反转这个问题，UCOSIII 支持一种特殊的二进制信号量：互斥信号量，用它可以解决优先级反转问题，如图 10.7.1 所示。

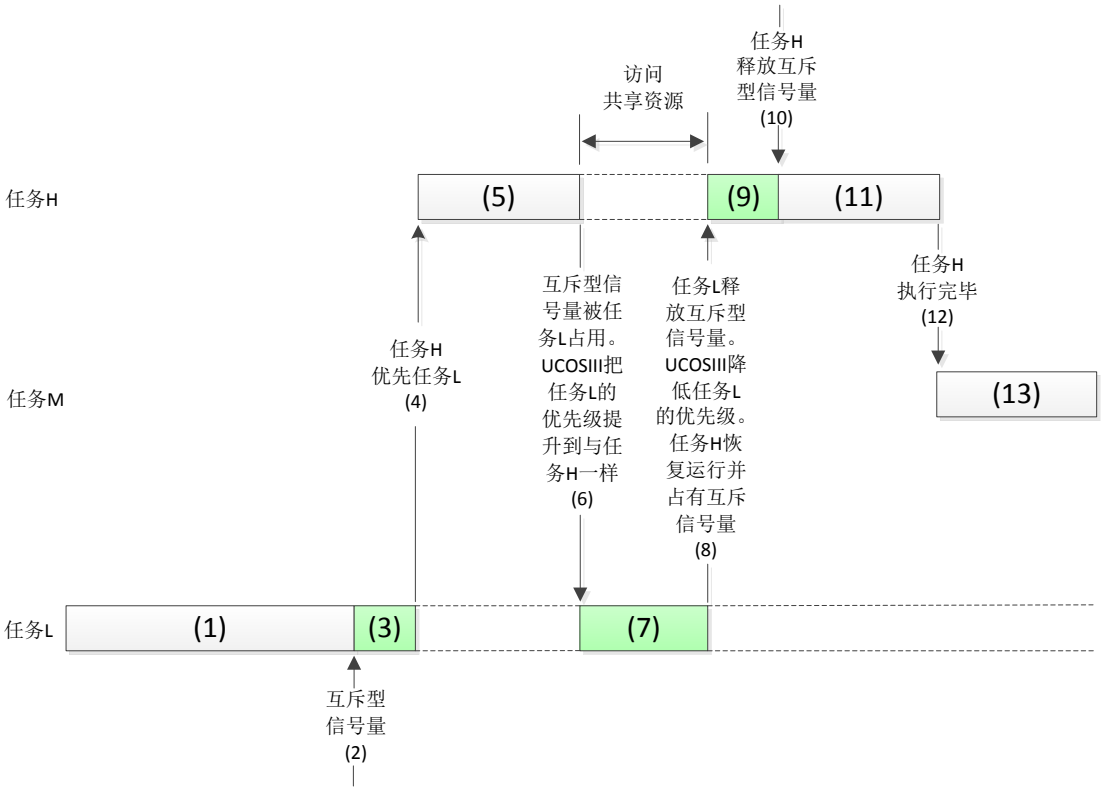


图 10.7.1 使用互斥型信号量访问共享资源

(1) 任务 H 与任务 M 处于挂起状态，等待某一事件的发生，任务 L 正在运行中。

(2) 某一时刻任务 L 想要访问共享资源，在此之前它必须先获得对应资源的互斥型信号量。

(3) 任务 L 获得互斥型信号量并开始使用该共享资源。

(4) 由于任务 H 优先级高，它等待的事件发生后便剥夺了任务 L 的 CPU 使用权。

(5) 任务 H 开始运行。

(6) 任务 H 运行过程中也要使用任务 L 在使用的资源，考虑到任务 L 正在占用着资源，UCOSIII 会将任务 L 的优先级升至同任务 H 一样，使得任务 L 能继续执行而不被其他中等优先级的任务打断。

(7) 任务 L 以任务 H 的优先级继续运行，注意此时任务 H 并没有运行，因为任务 H 在等待任务 L 释放掉互斥信号量。

(8) 任务 L 完成所有的任务，并释放掉互斥型信号量，UCOSIII 会自动将任务 L 的优先级恢复到提升之前的值，然后 UCOSIII 会将互斥型信号量给正在等待着的任务 H。

(9) 任务 H 获得互斥信号量开始执行。

(10) 任务 H 不再需要访问共享资源，于是释放掉互斥型信号量。

(11) 由于没有更高优先级的任务需要执行，所以任务 H 继续执行。

(12) 任务 H 完成所有工作，并等待某一事件发生，此时 UCOSIII 开始运行在任务 H 或者任务 L 运行过程中已经就绪的任务 M。

(13) 任务 M 继续执行。

注意！只有任务才能使用互斥信号量(中断服务程序则不可以)，UCOSIII 允许用户嵌套使用互斥型信号量，一旦一个任务获得了一个互斥型信号量，则该任务最多可以对该互斥型信号量嵌套使用 250 次，当然该任务只有释放相同的次数才能真正释放这个互斥型信号量。

与普通信号量一样，对于互斥信号量也可以进行许多操作，如表 10.7.1 所示，文件 os_mutex.c 是关于互斥信号量的。

函数	描述
OSMutexCreate()	创建一个互斥信号量
OSMutexDel()	删除一个互斥型信号量
OSMutexPend()	等待一个互斥型信号量
OSMutexPendAbort()	取消等待
OSMutexPost()	释放一个互斥型信号量

表 10.7.1 互斥型信号量操作 API 函数

10.7.1 创建互斥型信号量

创建互斥信号量使用函数 OSMutexCreate()，函数原型如下：

```
void  OSMutexCreate (OS_MUTEX   *p_mutex,
                     CPU_CHAR   *p_name,
                     OS_ERR     *p_err)
```

p_mutex： 指向互斥型信号量控制块。互斥型信号量必须有用户应用程序进行实际分配，可以使用如下所示代码。

```
OS_MUTEX  MyMutex;
```

p_name： 互斥信号量的名字

p_err： 调用此函数后返回的错误码。

10.7.2 请求互斥型信号量

当一个任务需要对资源进行独占式访问的时候就可以使用函数 OSMutexPend()，如果该互斥信号量正在被其他的任务使用，那么 UCOSIII 就会将请求这个互斥信号量的任务放置在这个互斥信号量的等待表中。任务会一直等待，直到这个互斥信号量被释放掉，或者设定的超时时间到达为止。如果在设定的超时时间到达之前信号量被释放，UCOSIII 将会恢复所有等待这个信号量的任务中优先级最高的任务。

注意！如果占用该互斥信号量的任务比当前申请该互斥信号量的任务优先级低的话，OSMutexPend()函数会将占用该互斥信号量的任务的优先级提升到和当前申请任务的优先级一样。当占用该互斥信号量的任务释放掉该互斥信号量以后，恢复到之前的优先级。OSMutexPend()函数原型如下：

```
void  OSMutexPend (OS_MUTEX   *p_mutex,
                   OS_TICK    timeout,
                   OS_OPT     opt,
                   CPU_TS     *p_ts,
                   OS_ERR     *p_err)
```

p_mutex： 指向互斥信号量。

timeout： 指定等待互斥信号量的超时时间（时钟节拍数），如果在指定的时间内互斥信号量没有释放，则允许任务恢复执行。该值设置为 0 的话，表示任务将会一直等待下去，直到信号量被释放掉。

opt： 用于选择是否使用阻塞模式。

OS_OPT_PEND_BLOCKING 指定互斥信号量被占用时，任务挂起等待该互斥信号量。

OS_OPT_PEND_NON_BLOCKING 指定当互斥信号量被占用时，直接返回任务。

注意！当设置为 OS_OPT_PEND_NON_BLOCKING，是 timeout 参数就没有意义了，应该设置为 0。

p_ts： 指向一个时间戳，记录发送、终止或删除互斥信号量的时刻。

p_err： 用于保存调用此函数后返回的错误码。

10.7.3 发送互斥信号量

我们可以通过调用函数 OSMutexPost()来释放互斥型信号量，只有之前调用过函数 OSMutexPend()获取互斥信号量，才需要调用 OSMutexPost()函数来释放这个互斥信号量，函数原型如下：

```
void  OSMutexPost (OS_MUTEX   *p_mutex,
                   OS_OPT     opt,
                   OS_ERR     *p_err)
```

p_mutex: 指向互斥信号量。

opt: 用来指定是否进行任务调度操作

OS_OPT_POST_NONE 不指定特定的选项

OS_OPT_POST_NO_SCHED 禁止在本函数内执行任务调度操作。

p_err: 用来保存调用此函数返回的错误码。

10.8 互斥信号量实验

10.8.1 实验程序设计

在 10.6 节中的例 10-4 中由于使用了信号量导致了优先级反转发生，本节中我们将信号量换成互斥信号量，本实验部分源码如下，完整的工程详见：例 10-5 UCOSIII 互斥信号量。

首先肯定要定义一个互斥信号量：

```
OS_MUTEX  TEST_MUTEX;          //定义一个互斥信号量
```

在 start_task 任务中创建一个互斥信号量 TEST_MUTES。

```
//创建一个互斥信号量
OSMutexCreate((OS_MUTEX*   )&TEST_MUTEX,
              (CPU_CHAR*   )"TEST_MUTEX",
              (OS_ERR*     )&err);
```

high_task、middle_task 和 low_task 这三个任务的任务函数如下：

```
//高优先级任务的任务函数
void high_task(void *p_arg)
{
    u8 num;
    OS_ERR err;

    CPU_SR_ALLOC();
    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(5,110,115,314);  //画一个矩形
    LCD_DrawLine(5,130,115,130);       //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(6,111,110,16,16,"High Task");
    OS_CRITICAL_EXIT();
    while(1)
    {
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_PERIODIC,&err);    //延时 500ms
        num++;
        printf("high task Pend Sem\r\n");
        OSMutexPend (&TEST_MUTEX,0,\            //请求互斥信号量                    (1)
                     OS_OPT_PEND_BLOCKING,0,&err);
        printf("high task Running!\r\n");
        LCD_Fill(6,131,114,313,lcd_discolor[num%14]);                    //填充区域
        LED1 = ~LED1;
        OSMutexPost(&TEST_MUTEX,OS_OPT_POST_NONE,&err);//释放互斥信号量 (2)
        OSTimeDlyHMSM(0,0,0,500,OS_OPT_TIME_PERIODIC,&err);  //延时 500ms
    }
}

//中等优先级任务的任务函数
```

```
void middle_task(void *p_arg)
{
    u8 num;
    OS_ERR err;
    CPU_SR_ALLOC();

    POINT_COLOR = BLACK;
    OS_CRITICAL_ENTER();
    LCD_DrawRectangle(125,110,234,314); //画一个矩形
    LCD_DrawLine(125,130,234,130);          //画线
    POINT_COLOR = BLUE;
    LCD_ShowString(126,111,110,16,16,"Middle Task");
    OS_CRITICAL_EXIT();
    while(1)
    {
        num++;
        printf("middle task Running!\r\n");
        LCD_Fill(126,131,233,313,lcd_discolor[13-num%14]); //填充区域
        LED0 = ~LED0;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);    //延时 1s
    }
}

//低优先级任务的任务函数
void low_task(void *p_arg)
{
    static u32 times;
    OS_ERR err;
    while(1)
    {
        OSMutexPend (&TEST_MUTEX,0,\                          //请求互斥信号量  (3)
                     OS_OPT_PEND_BLOCKING,0,&err);
        printf("low task Running!\r\n");
        for(times=0;times<20000000;times++)                                    (4)
        {
            OSSched();                                                  //发起任务调度
        }
        OSMutexPost(&TEST_MUTEX,\                                              (5)
                    OS_OPT_POST_NONE,&err);//释放互斥信号量
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);      //延时 1s
    }
}
```

(1)、high_task 任务中请求互斥信号量 TEST_MUTEX。

(2)、high_task 任务中释放互斥信号量 TEST_MUTEX。
(3)、low_task 任务中请求会互斥信号量 TEST_MUTEX。
(4)、这里用来模拟 low_task 任务长时间占用互斥信号量 TEST_MUTEX。
(5)、low_task 任务中释放互斥信号量 TEST_MUTEX。

10.8.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，下载完以后 LCD 显示如图 10.8.1 所示。

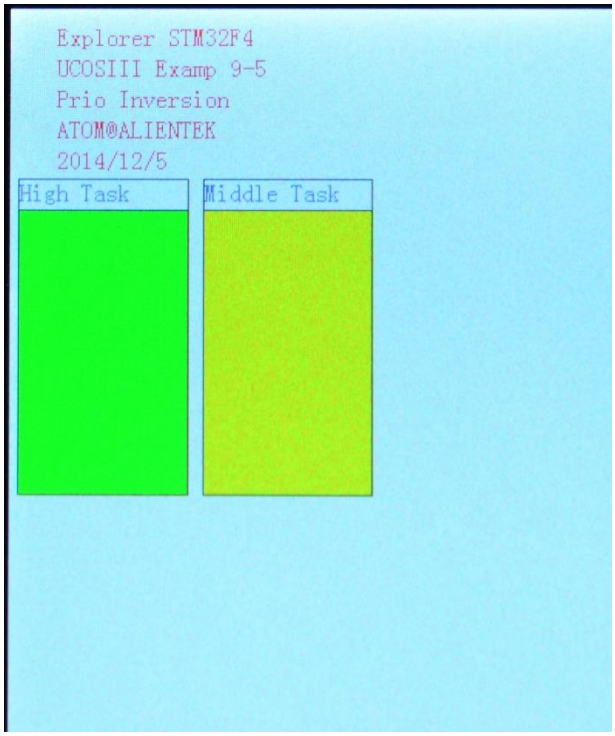


图 10.8.1 LCD 显示结果

再来看串口输出信息，如图 10.8.2 所示：

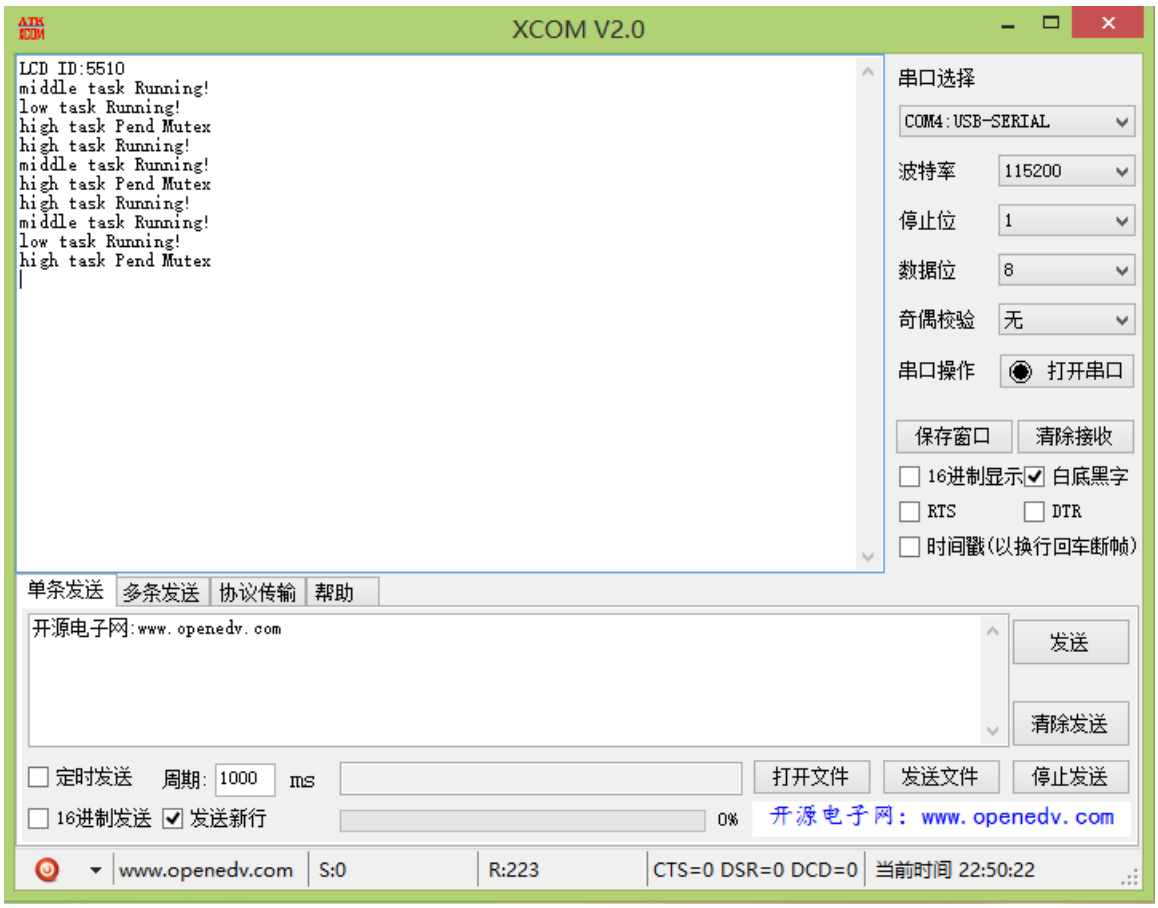


图 10.8.2 串口输出

为了方便分析，我们将串口调试助手中的数据复制下来，如下：

```
LCD ID:5510
middle task Running!                    (1)
low task Running!                       (2)
high task Pend Mutex                    (3)
high task Running!                      (4)
middle task Running!
high task Pend Mutex
high task Running!
middle task Running!
low task Running!
high task Pend Mutex
```

(1)、middle_task 任务运行。

(2)、low_task 获得互斥信号量运行。

(3)、high_task 请求信号量，在这里会等待一段时间，等待 low_task 任务释放互斥信号量。但是 middle_task 不会运行，因为由于 low_task 正在使用互斥信号量，所以 low_task 任务优先级暂时提升到了一个高优先级(比 middle_task 任务优先级高)，所以 middle_task 任务不能在打断 low_task 任务的运行了！

(4)、high_task 任务获得互斥信号量而运行。

从上面的分析可以看出互斥信号量有效的抑制了优先级反转现象的发生。

10.9 任务内嵌信号量

前面我们使用信号量时都需要先创建一个信号量，不过在 UCOSIII 中每个任务都有自己的内嵌的信号量，这种功能不仅能够简化代码，而且比使用独立的信号量更有效。任务信号量是直接内嵌在 UCOSIII 中的，任务信号量相关代码在 os_task.c 中。任务内嵌信号量相关函数如表 10.9.1 所示：

函数名	描述
OSTaskSemPend()	等待任务信号量
OSTaskSemPendAbort()	取消等待任务信号量
OSTaskSemPost()	发布任务信号量
OSTaskSemSet()	强行设置任务信号量计数

表 10.9.1 任务信号量相关函数

10.9.1 等待任务信号量

等待任务内嵌信号量使用函数 OSTaskSemPend()，OStaskSemPend()允许一个任务等待由其他任务或者 ISR 直接发送的信号，使用过程基本和独立的信号量相同，OSTaskSemPend()函数原型如下：

```
OS_SEM_CTR  OSTaskSemPend ( OS_TICK      timeout,
                            OS_OPT       opt,
                            CPU_TS       *p_ts,
                            OS_ERR       *p_err)
```

timeout： 如果在指定的节拍数内没有收到信号量任务就会因为等待超时而恢复运行，如果 timeout 为 0 的话任务就会一直等待，直到收到信号量。

opt: 用于选择是否使用阻塞模式。

OS_OPT_PEND_BLOCKING 指定互斥信号量被占用时，任务挂起等待该互斥信号量。

OS_OPT_PEND_NON_BLOCKING 指定当互斥信号量被占用时，直接返回任务。

注意！当设置为 OS_OPT_PEND_NON_BLOCKING，是 timeout 参数就没有意义了，应该设置为 0。

p_ts： 指向一个时间戳，记录发送、终止或删除互斥信号量的时刻。

P_err: 调用此函数后返回的错误码。

10.9.2 发布任务信号量

OSTaskSemPost()可以通过一个任务的内置信号量向某个任务发送一个信号量，函数原型如下：

```
OS_SEM_CTR  OSTaskSemPost (OS_TCB     *p_tcb,
                           OS_OPT     opt,
                           OS_ERR     *p_err)
```

p_tcb： 指向要用信号通知的任务的 TCB，当设置为 NULL 的时候可以向自己发送信量。

opt: 用来指定是否进行任务调度操作

OS_OPT_POST_NONE 不指定特定的选项

OS_OPT_POST_NO_SCHED　　禁止在本函数内执行任务调度操作。

p_err:　　调用此函数后返回的错误码。

10.10 任务内嵌信号量实验

10.10.1 实验程序设计

例 10-6：创建 3 个任务，任务 start_task 用于创建其他两个任务，任务 task1_task 主要用于扫描按键，当检测到 KWY_UP 按下以后就向任务 task2_task 发送一个任务信号量。任务 task2_task 请求任务信号量，当请求到任务信号量的时候就更新一次屏幕指定区域的背景颜色。

答：这个问题显然也是一个任务同步的问题，task1_task 任务和 task2_task 任务之间使用 task2_task 任务内嵌的信号量来做同步，完整的工程详见：例 10-6 UCOSIII 任务内嵌信号量。

由于我们使用任务内嵌的信号量所以就不需要创建信号量，task1_task 和 task2_task 两个任务的任务函数如下：

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    u8 key;
    u8 num;
    OS_ERR err;
    while(1)
    {
        key = KEY_Scan(0);    //扫描按键
        if(key==WKUP_PRES)
        {
            OSTaskSemPost(&Task2_TaskTCB,                    (1)
            OS_OPT_POST_NONE,&err); //使用系统内建信号量向任务 task2 发送信号量
            LCD_ShowxNum(150,111,Task2_TaskTCB.SemCtr,3,16,0);          //显示信号量值
        }
        num++;
        if(num==50)
        {
            num=0;
            LED0=~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);          //延时 10ms
    }
}

//任务 2 的任务函数
void task2_task(void *p_arg)
{
    u8 num;
    OS_ERR err;
```

```
    while(1)
    {
        OSTaskSemPend(0,OS_OPT_PEND_BLOCKING,\                    (2)
                      0,&err);  //请求任务内建的信号量
        num++;
        LCD_ShowxNum(150,111,Task2_TaskTCB.SemCtr,3,16,0);    //显示任务内建信号量值
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);          //刷屏
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);    //延时 1s
    }
}
```

(1)、调用函数 OSTaskSemPost()向任务 task2_task 发送一个任务信号量。

(2)、任务 task2_task 一直请求任务信号量。

10.10.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，由于任务 task2_task 内嵌信号量初始值为 0，因此在开机以后任务 task2_task 会由于请求不到信号量而阻塞，此时 LCD 显示如图 10.10.1 所示。

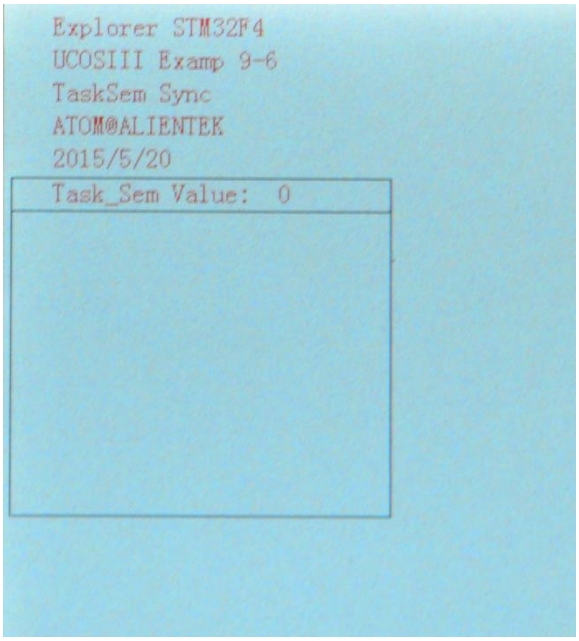


图 10.10.1 开机 LCD 显示

当我们按下 KEY_UP 键以后就会发送信号量，task2_task 任务内嵌信号量的值就会变化(增加)，我们多按几次 KEY_UP 键，如图 10.10.2 所示。

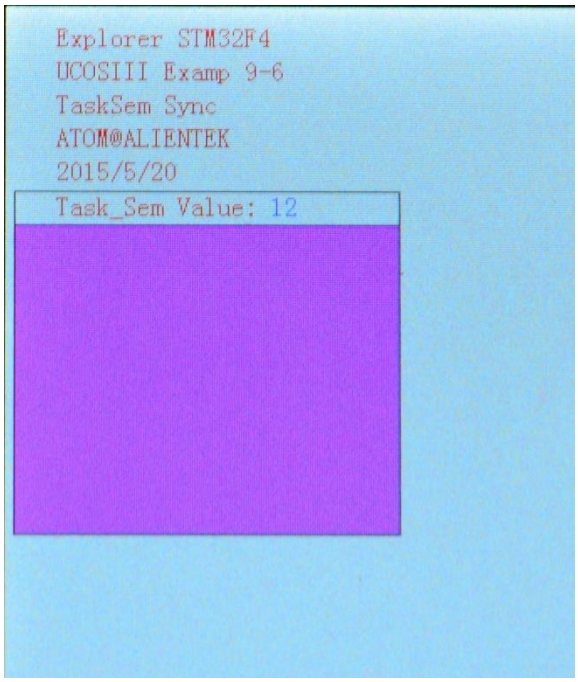


图 10.10.2 发送多次信号量后的 LCD 显示

从图 10.10.2 中可以看出，此时 task2_task 任务的内嵌信号量的值为 12，说明 task2_task 任务可以请求 12 次任务内嵌的信号量。任务 task2_task 每隔 1s 就会请求一起内嵌信号量，直到任务内嵌的信号量值为 0，由于 task2_task 请求不到信号量了，因此 task2_task 就会阻塞，此时 LCD 如图 10.10.3 所示。

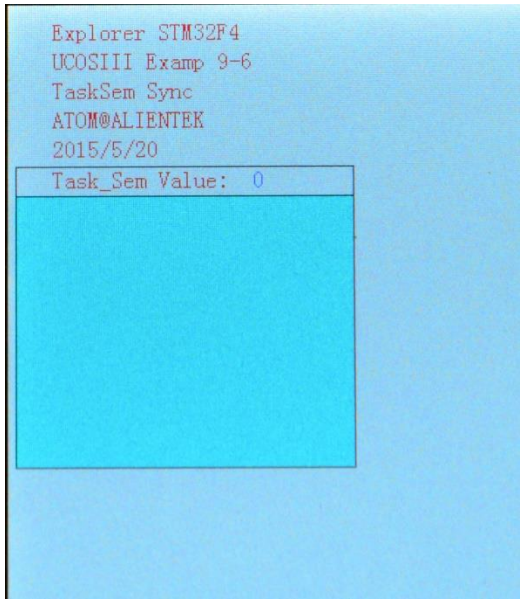


图 10.10.3 信号量减小为 0

task2_task 内嵌信号量值减小到 0，任务 task2_task 阻塞。当我们再次按下 KEY_UP 的时候，task2_task 又会接着“运行”，大家可以自行尝试一次。

第十一章 UCOSIII 消息传递

有时候一个任务要和另外一个或者几个任务进行“交流”，这个“交流”就是消息的传递，也称之为任务间通信，在 UCOSIII 中消息可以通过消息队列作为中介发布给任务，也可以直接发布给任务，本章我们就讲解一个 UCOSIII 中的消息传递，本章分为如下几部分。

11.1 消息队列

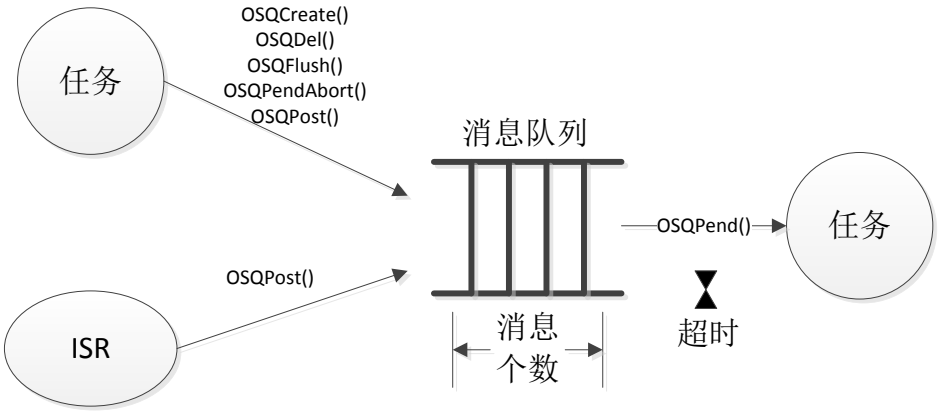
11.2 消息队列相关函数

11.3 消息队列实验

11.1 消息队列

消息一般包含：指向数据的指针，表明数据长度的变量和记录消息发布时刻的时间戳，指针指向的可以是一块数据区或者甚至是一个函数，消息的内容必须一直保持可见性，因为发布数据采用的是引用传递是指针传递而不是值传递，也就说，发布的数据本身不产生数据拷贝。

在 UCOSII 中有消息邮箱和消息队列，但是在 UCOSIII 中只有消息队列。消息队列是由用户创建的内核对象，数量不限制，图 11.1.1 展示了用户可以对消息队列进行的操作。



如图 11.1.1 消息队列相关操作

从图 11.1.1 中可以看出，中断服务程序只能使用 OSQPost()函数！在 UCOSIII 中对于消息队列的读取既可以采用先进先出(FIFO)的方式，也可以采用后进先出(LIFO)的方式。当任务或者中断服务程序需要向任务发送一条紧急消息时 LIFO 的机制就非常有用了。采用后进先出的方式，发布的消息会绕过其他所有的已经位于消息队列中的消息而最先传递给任务。

图 11.1.1 中接收消息的任务旁边的小沙漏表示任务可以指定一个超时时间，如果任务在这段时间内没有接收到消息的话就会唤醒任务，并且返回一个错误码告诉 UCOSIII 超时，任务是因为接收消息超时而被唤醒的，不是因为接收到了消息。如果将这个超时时间指定为 0 的话，那么任务就会一直等待下去，直到接收到消息。

消息队列中有一个列表，记录了所有正在等待获得消息的任务，如图 11.1.2 所示为多个任务可以在一个消息队列中等待，当一则消息被发布到队列中时，最高优先级的等待任务将获得该消息，发布方也可以向消息队列中所有等待的任务广播一则消息。

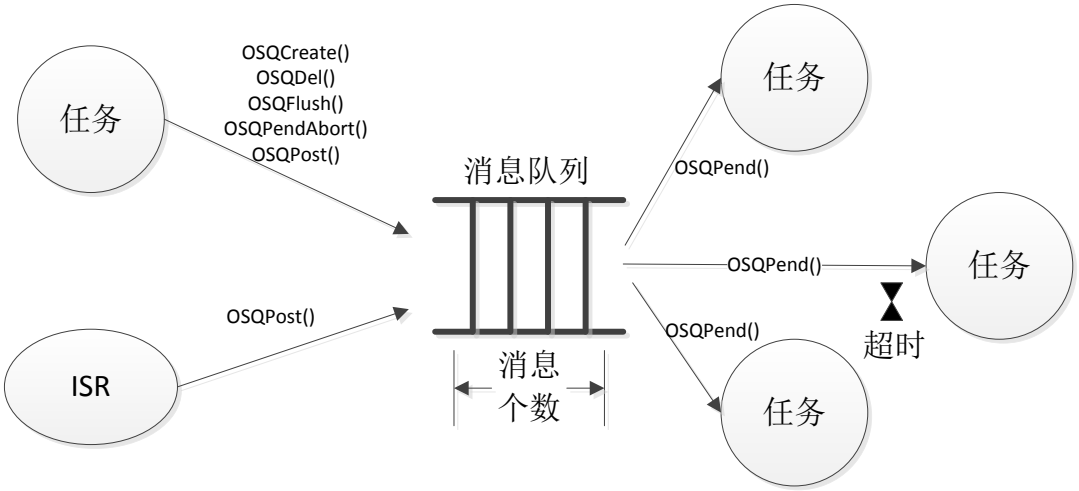


图 11.1.2 多个任务在等待一个消息队列

11.2 消息队列相关函数

有关消息队列的 API 函数如下表 11.2.1 所示。

函数	描述
OSQCreate()	创建一个消息队列
OSQDel()	删除一个消息队列
OSQFlush()	清空一个消息队列
OSQPend()	等待消息队列
OSQPendAbort()	取消等待消息队列
OSQPost()	向消息队列发送一条消息

我们常用的关于消息队列的函数其实只有三个，创建消息队列函数 OSQCreate()，向消息队列发送消息函数 OSQPost()和等待消息队列函数 OSQPend()。

11.2.1 创建消息队列

OSQCreate()函数用来创建一个消息队列，消息队列使得任务或者中断服务程序可以向一个或者多个任务发送消息，函数原型如下。

```
void    OSQCreate (OS_Q            *p_q,
                   CPU_CHAR        *p_name,
                   OS_MSG_QTY      max_qty,
                   OS_ERR          *p_err)
```

p_q： 指向一个消息队列，消息队列的存储空间必须由应用程序分配，我们采用如下的语句定义一个消息队列。

```
OS_Q    Msg_Que;
```

p_name： 消息队列的名字。

max_qty： 指定消息队列的长度，必须大于 0。当然，如果 OS_MSGs 缓冲池中没有足够多的 OS_MSGs 可用，那么发送消息将会失败，并且返回相应的错误码，指明当前没有 可用的 OS_MSGs

p_err： 保存调用此函数后返回的错误码

11.2.2 等待消息队列

当一个任务想要从消息队列中接收一个消息的话就需要使用函数 OSQPend()。当任务调用这个函数的时候，如果消息队列中有至少一个消息时，这些消息就会返回给函数调用者。函数原型如下：

```
void    *OSQPend (OS_Q             *p_q,
                  OS_TICK          timeout,
                  OS_OPT           opt,
                  OS_MSG_SIZE      *p_msg_size,
                  CPU_TS           *p_ts,
                  OS_ERR           *p_err)
```

p_q： 指向一个消息队列。

timeout： 等待消息的超时时间，如果在指定的时间没有接收到消息的话，任务就会被唤醒，接着运行。这个参数也可以设置为 0，表示任务将一直等待下去，直到接收到消息。

opt： 用来选择是否使用阻塞模式，有两个选项可以选择。

OS_OPT_PEND_BLOCKING　如果没有任何消息存在的话就阻塞任务，一直等待，直到接收到消息。

OS_OPT_PEND_NON_BLOCKING　如果消息队列没有任何消息的话任务就直接返回。

p_msg_size: 指向一个变量用来表示接收到的消息长度(字节数)。

p_ts: 指向一个时间戳，表明什么时候接收到消息。如果这个指针被赋值为 NULL 的话，说明用户没有要求时间戳。

p_err: 用来保存调用此函数后返回的错误码。

如果消息队列中没有任何消息，并且参数 opt 为 OS_OPT_PEND_NON_BLOCKING 时，那么调用 OSQPend()函数的任务就会被挂起，直到接收到消息或者超时。如果有消息发送给消息队列，但是同时有多个任务在等待这个消息，那么 UCOSIII 将恢复等待中的最高优先级的任务。

11.2.3 向消息队列发送消息

可以通过函数 OSQPost()向消息队列发送消息，如果消息队列是满的，则函数 OSQPost()就会立刻返回，并且返回一个特定的错误代码，函数原型如下：

```
void  OSQPost (OS_Q          *p_q,
               void          *p_void,
               OS_MSG_SIZE   msg_size,
               OS_OPT        opt,
               OS_ERR        *p_err)
```

如果有多个任务在等待消息队列的话，那么优先级最高的任务将获得这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，则系统会执行任务调度，等待消息的任务立即恢复运行，而发送消息的任务被挂起。可以通过 opt 设置消息队列是 FIFO 还是 LIFO。

如果有多个任务在等待消息队列的消息，则 OSQPost()函数可以设置仅将消息发送给等待任务中优先级最高的任务(opt 设置为 OS_OPT_POST_FIF 或者 OS_OPT_POST_LIFO)，也可以将消息发送给所有等待的任务(opt 设置为 OS_OPT_POST_ALL)。如果 opt 设置为 OS_OPT_POST_NO_SCHED，则在发送完消息后，会进行任务调度。

p_q: 指向一个消息队列。

p_void: 指向实际发送的内容，p_void 是一个执行 void 类型的指针，其具体含义由用户程序的决定。

msg_size: 设定消息的大小，单位为字节数。

opt: 用来选择消息发送操作的类型，基本的类型可以有下面四种。

OS_OPT_POST_ALL　将消息发送给所有等待该消息队列的任务，需要和选项 OS_OPT_POST_FIFO 或者 OS_OPT_POST_LIFO 配合使用。

OS_OPT_POST_FIFO　待发送消息保存在消息队列的末尾

OS_OPT_POST_LIFO　待发送的消息保存在消息队列的开头

OS_OPT_POST_NO_SCHED　禁止在本函数内执行任务调度。

我们可以使用上面四种基本类型来组合出其他几种类型，如下：

OS_OPT_POST_FIFO + OS_OPT_POST_ALL

OS_OPT_POST_LIFO + OS_OPT_POST_ALL

OS_OPT_POST_FIFO + OS_OPT_POST_NO_SCHED

OS_OPT_POST_LIFO + OS_OPT_POST_NO_SCHED

OS_OPT_POST_FIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED
OS_OPT_POST_LIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED

p_err： 用来保存调用此函数后返回的错误码。

11.3 消息队列实验

11.3.1 实验程序设计

例 11-1：设计一个应用程序，该程序有 4 任务、两个消息队列和一个定时器。任务 start_task 用于创建其他 3 个任务。main_task 任务为主任务，用于检测按键，并且将按键的值通过消息队列 KEY_Msg 发送给任务 Keyprocess_task，main_task 任务还用于检测消息队列 DATA_Msg 的总大小和剩余空间大小，并且控制 LED0 的闪烁。Keyprocess_task 任务获取 KEY_Msg 内的消息，根据不同的消息做出相应的处理。

定时器 1 的回调函数 tmr1_callback 通过消息队列 DATA_Msg 将定时器 1 的运行次数作为信息发送给任务 msgdis_task，任务 msgdis_task 将 DATA_Msg 中的消息显示在 LCD 上。

答：实验关键代码如下，实验完整工程见“例 11-1 UCOSIII 消息传递”。

首先应该定义两个消息队列和一个定时器以及相关的宏，代码如下。

```
//////////////////////消息队列/////////////////////////////
#define KEYMSG_Q_NUM        1    //按键消息队列的数量
#define DATAMSG_Q_NUM       4    //发送数据的消息队列的数量
OS_Q KEY_Msg;                    //定义一个消息队列，用于按键消息传递，模拟消息邮箱
OS_Q DATA_Msg;                   //定义一个消息队列，用于发送数据
//////////////////////定时器/////////////////////////////
u8 tmr1sta=0;       //标记定时器的工作状态
OS_TMR tmr1;        //定义一个定时器
```

结构体 OS_Q 用来描述消息队列，在 OS_Q 中有个字段 MsgQ，MsgQ 也是一个结构体，MsgQ 中的字段 NbrEntriesSize 和 NbrEntries 用来记录消息队列总大小和已经使用了的消息队列大小，两者之差就是消息队列剩余的空间大小。函数 check_msg_queue()就是用来检测消息队列 DATA_Msg 的总空间大小和剩余空间大小的，函数代码如下。

```
//查询 DATA_Msg 消息队列中的总队列数量和剩余队列数量
void check_msg_queue(u8 *p)
{
    u8 msgq_remain_size;  //消息队列剩余大小
    msgq_remain_size = DATA_Msg.MsgQ.NbrEntriesSize-DATA_Msg.MsgQ.NbrEntries;
    p = mymalloc(SRAMIN,20); //申请内存
    //显示 DATA_Msg 消息队列总的大小
    sprintf((char*)p,"Total Size:%d",DATA_Msg.MsgQ.NbrEntriesSize);
    LCD_ShowString(10,190,100,16,16,p);
    sprintf((char*)p,"Remain Size:%d",msgq_remain_size);    //显示 DATA_Msg 剩余大小
    LCD_ShowString(10,230,100,16,16,p);
    myfree(SRAMIN,p);    //释放内存
}
```

前面虽然定义了 2 个消息队列和 1 个定时器，但是此时还不能用，我们需要调用 OSQCreate()和 OSTmrCreate()这两个函数来创建消息队列和定时器，在任务 start_task 中我们来创建这两个

消息队列和定时器，代码如下。

```
//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
   OSStatTaskCPUUsageInit(&err);  //统计任务
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN          //如果使能了测量中断关闭时间
    CPU_IntDisMeasMaxCurReset();
#endif

#if  OS_CFG_SCHED_ROUND_ROBIN_EN  //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
#endif
    OS_CRITICAL_ENTER();  //进入临界区
    //创建消息队列 KEY_Msg
    OSQCreate ( (OS_Q*          )&KEY_Msg,//消息队列                    (1)
                (CPU_CHAR*      )"KEY Msg",//消息队列名称
                (OS_MSG_QTY     )KEYMSG_Q_NUM,  //消息队列长度，这里设置为 1
                (OS_ERR*        )&err);       //错误码
    //创建消息队列 DATA_Msg
    OSQCreate ( (OS_Q*          )&DATA_Msg,                             (2)
                (CPU_CHAR*      )"DATA Msg",
                (OS_MSG_QTY     )DATAMSG_Q_NUM,
                (OS_ERR*        )&err);
    //创建定时器 1
    OSTmrCreate((OS_TMR*        )&tmr1,       //定时器 1                  (3)
                (CPU_CHAR*      )"tmr1",      //定时器名字
                (OS_TICK        )0,           //0ms
                (OS_TICK        )50,           //50*10=500ms
                (OS_OPT         )OS_OPT_TMR_PERIODIC, //周期模式
                (OS_TMR_CALLBACK_PTR)tmr1_callback,//定时器 1 回调函数
                (void*          )0,           //参数为 0
                (OS_ERR*        )&err);       //返回的错误码
     ******
    //为了省篇幅，此处省略掉了创建任务的代码。
```

```
    ......
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err);  //删除 start_task 任务自身
}
```

(1) 调用函数 OSCreate()创建一个消息队列 KEY_Msg，KEY_Msg 队列长度为 1，我们用来模拟 UCOSII 中的消息邮箱。

(2) 调用函数 OSCreate()创建一个消息队列 DATA_Msg，队列长度为 4。

(3) 调用函数 OSTmrCreate()创建一个定时器 tmr1，tmr1 为周期定时器，定时周期为 500ms。

```
//定时器 1 的回调函数
void tmr1_callback(void *p_tmr,void *p_arg)
{
    u8 *pbuf;
    static u8 msg_num;
    OS_ERR err;
    pbuf = mymalloc(SRAMIN,10);  //申请 10 个字节
    if(pbuf)  //申请内存成功
    {
        msg_num++;
        sprintf((char*)pbuf,"ALIENTEK %d",msg_num);
        //发送消息
        OSQPost((OS_Q*          )&DATA_Msg,
                (void*          )pbuf,
                (OS_MSG_SIZE    )10,
                (OS_OPT         )OS_OPT_POST_FIFO,
                (OS_ERR*        )&err);
        if(err != OS_ERR_NONE)
        {
            myfree(SRAMIN,pbuf); //释放内存
            OSTmrStop(&tmr1,OS_OPT_TMR_NONE,0,&err); //停止定时器 1
            tmr1sta = !tmr1sta;
            LCD_ShowString(10,150,100,16,16,"TMR1 STOP! ");
        }
    }
}
//主任务的任务函数
void main_task(void *p_arg)
{
    u8 key,num;
    OS_ERR err;
    u8 *p;
    while(1)
    {
        key = KEY_Scan(0);  //扫描按键
```

```
        if(key)
        {
            //发送消息
            OSQPost((OS_Q*          )&KEY_Msg,
                    (void*          )&key,
                    (OS_MSG_SIZE    )1,
                    (OS_OPT         )OS_OPT_POST_FIFO,
                    (OS_ERR*        &err);
        }
        num++;
        if(num%10==0) check_msg_queue(p);//检查 DATA_Msg 消息队列的容量
        if(num==50)
        {
            num=0;
            LED0 = ~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}
//按键处理任务的任务函数
void Keyprocess_task(void *p_arg)
{
    u8 num;
    u8 *key;
    OS_MSG_SIZE size;
    OS_ERR err;
    while(1)
    {
        //请求消息 KEY_Msg
        key=OSQPend((OS_Q*          )&KEY_Msg,
                    (OS_TICK        )0,
                    (OS_OPT         )OS_OPT_PEND_BLOCKING,
                    (OS_MSG_SIZE*   )&size,
                    (CPU_TS*        )0,
                    (OS_ERR*        )&err);
        switch(*key)
        {
            case WKUP_PRES:          //KEY_UP 控制 LED1
                LED1 = ~LED1;
                break;
            case KEY2_PRES:      //KEY2 控制蜂鸣器
                BEEP = ~BEEP;
                break;
```

```
            case KEY0_PRES:       //KEY0 刷新 LCD 背景
                num++;
                LCD_Fill(126,111,233,313,lcd_discolor[num%14]);
                break;
            case KEY1_PRES:       //KEY1 控制定时器 1
                tmr1sta = !tmr1sta;
                if(tmr1sta)
                {
                    OSTmrStart(&tmr1,&err);
                    LCD_ShowString(10,150,100,16,16,"TMR1 START!");
                }
                else
                {
                    OSTmrStop(&tmr1,OS_OPT_TMR_NONE,0,&err); //停止定时器 1
                    LCD_ShowString(10,150,100,16,16,"TMR1 STOP! ");
                }
                break;
        }
    }
}
//显示消息队列中的消息
void msgdis_task(void *p_arg)
{
    u8 *p;
    OS_MSG_SIZE size;
    OS_ERR err;
    while(1)
    {
        //请求消息
        p=OSQPend((OS_Q*            )&DATA_Msg,
                  (OS_TICK          )0,
                  (OS_OPT           )OS_OPT_PEND_BLOCKING,
                  (OS_MSG_SIZE*     )&size;
                  (CPU_TS*          )0,
                  (OS_ERR*          )&err);
        LCD_ShowString(5,270,100,16,16,p);
        myfree(SRAMIN,p);     //释放内存
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err); //延时 1s
    }
}
```

上面有四个函数：tmr1_callback()、main_task()、Keyprocess_task()和 msgdis_task()，这四个函数分别为定时器 1 的回调函数、主任务的任务函数、按键处理任务的任务函数和显示任务的任务函数。

tmr1_callback()函数是定时器 1 的回调函数，在 start_task 任务中我们是创建了一个定时器 tmr1 的，tmr1 是一个周期定时器，定时周期为 500ms。在 tmr1 的回调函数 tmr1_callback()中通过函数 OSQPost()向消息队列 DATA_Msg 发送消息，这里向消息队列发送数据采用的是 FIFO 方式，当发送失败的话就释放相应的内存并关闭定时器。

main_task()函数为主任务的任务函数，在这个函数中我们不断的扫描按键的键值，然后将键值发到消息队列 KEY_Msg 中，这里向消息队列发送数据采用的是 FIFO 方式。main_task()任务还要每隔 100ms 检测一次消息队列 DATA_Msg 总的大小和剩余大小并显示在 LCD 上，最后还要控制 LED0 的闪烁，提示系统正在运行。

Keyprocess_task()为按键处理任务，在 main_task 任务中我们将按键值发送到了消息队列 KEY_Msg 中，在本函数中我们调用 OSQPend()函数从消息队列 KEY_Msg 中获取消息，也就是按键值，然后根据不同的键值做出相应的处理。KEY_UP 控制 LED1；KEY2 控制蜂鸣器；KEY0 用来控制刷新 LCD 右下部分的背景颜色；KEY1 控制 tmr1 的开关。

msgdis_task()通过调用 OSQPend()函数获得消息队列 DATA_Msg 中的数据，并将获得到的消息显示在 LCD 上。

11.3.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，此时的 LCD 初始界面如图 11.3.1 所示。

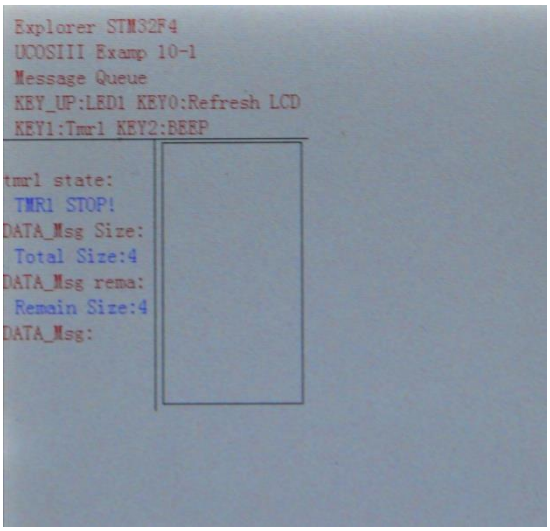


图 11.3.1 LCD 初始界面

从 11.3.1 中可以看出 DATA_Msg 的总大小为 4，这个和我们创建 DATA_Msg 消息队列时设置的一样。由于此时定时器 1 并没有启动，所以消息队列 DATA_Msg 的剩余大小也为 4，右下部分的方框部分的 LCD 背景为白色，当我们按下 KEY0 键的时候就会刷新右下方框中的背景，如图 11.3.2 中所示。

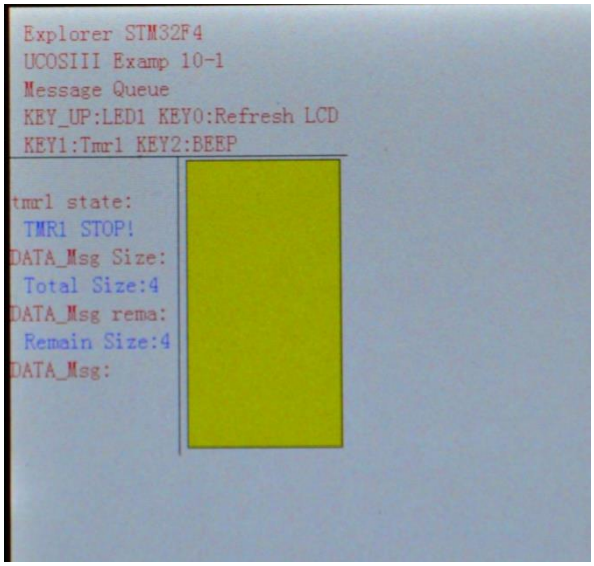


图 11.3.2 按下 KEY0 后的 LCD 界面

从图 11.3.2 中可以看出当按下 KEY0 以后右下部分的 LCD 背景就会被刷新为其他颜色(这里的黄色为多次按下 KEY0 后的效果)。按下 KEY1 键开启定时器 1，那么定时器 1 的回调函数就会每隔 500ms 向消息队列 DATA_Msg 中发送一条消息，如图 11.3.3 所示。

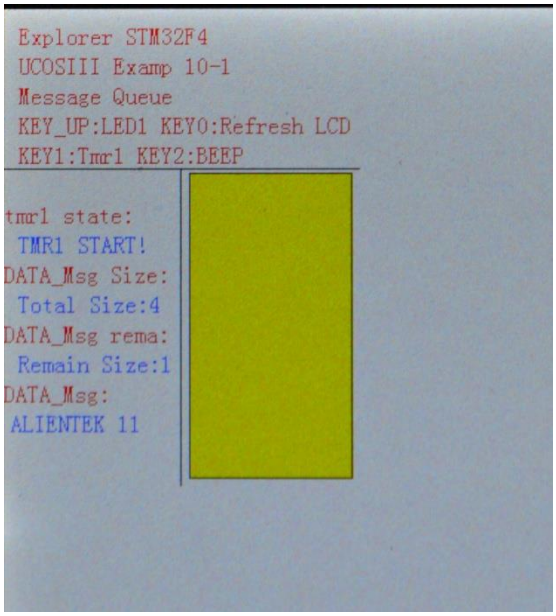


图 11.3.3 开启定时器 1

如图 11.3.3 所示，此时消息队列 DATA_Msg 还剩下 1 个可用空间，定时器 1 还会一直向 DATA_Msg 发送消息，直到消息队列 DATA_Msg 满了，就会关闭定时器 1，停止发送，如图 11.3.4 所示。

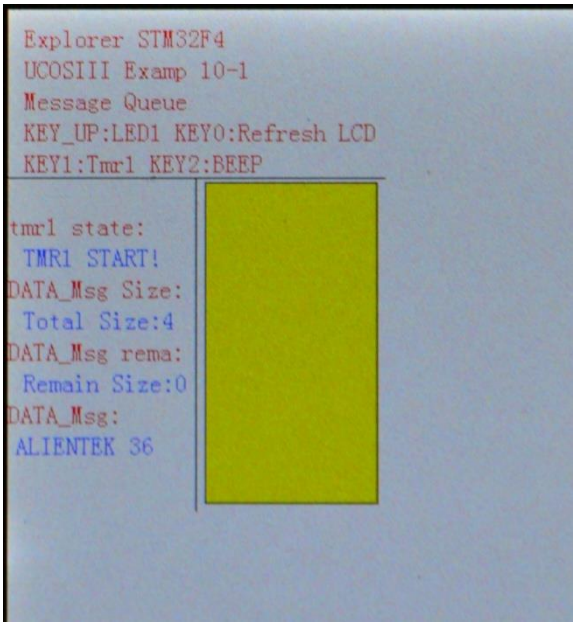


图 11.3.4 消息队列 DATA_Msg 剩余空间为 0

从图 11.3.4 中可以看出此时消息队列 DATA_Msg 剩余空间为 0，那么定时器 1 回调函数中再次调用函数 OSQPost()向消息队列 DATA_Msg 中发送数据的话就会发送失败，此时 err 就为 OS_ERR_MSG_POOL_EMPTY，提示消息队列空了，err 不等于 OS_ERR_NONE，那么就会关闭定时器 1，停止向 DATA_Msg 中发送数据，除非再次手动开启定时器 1，也就是按下 KEY1 键。注意！在图 11.3.4 中显示消息队列 DATA_Msg 的剩余空间为 0，但是 tmr1 还是开启状态，这里并没有错，是因为此时关闭定时器 1 的程序还没来得及执行(这个瞬间不好拍啊！！！)，大家仔细观察 LCD 会发现这个关闭的瞬间。

大家可能会注意到，消息队列 DATA_Msg 中剩余空间为 0，定时器 1 关闭了，但是此时任务 msgdis_task 还是能够接收到消息，并且在 LCD 上显示，而且消息队列 DATA_Msg 的剩余空间大小会增大，直到等于 DATA_Msg 总的大小才会停止，如图 11.3.5 所示。

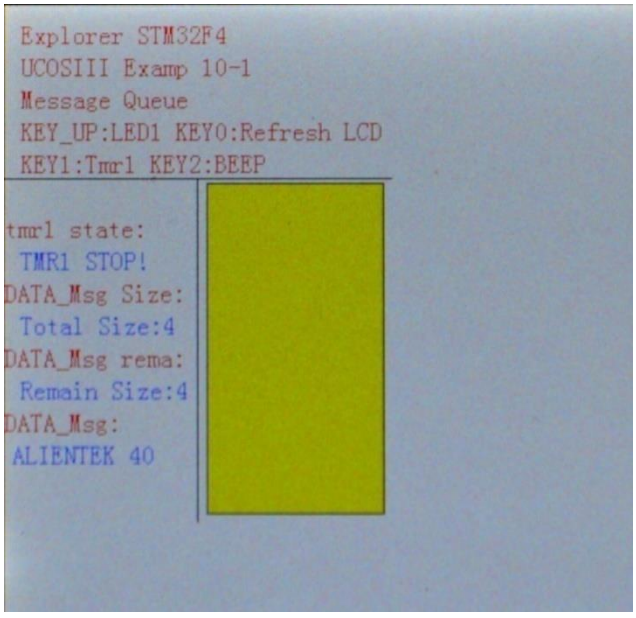


图 11.3.5 msgdis_task 任务停止运行

这是因为虽然定时器 1 关闭了，没有数据发送到消息队列 DATA_Msg 中，但是此时

DATA_Msg 中还有没有处理掉的数据，因此 msgdis_task 任务还会一直运行，直到处理完 DATA_Msg 中的所有数据，每处理掉一个数据，DATA_Msg 的剩余大小就会加 1，当处理完所有的数据，DATA_Msg 剩余大小肯定就会等于总大小了。

按下 KEY1 键会改变 LED1 的状态，按下 KEY2 键会开关蜂鸣器，大家可以尝试操作一下，至于消息队列 KEY_Msg 就非常简单了，大家自行分析一下。

11.4 任务内建消息队列

和任务信号量一样，UCOSIII 中每个任务也都有其内建消息队列，这样的话用户就不需要使用外部的消息队列就可直接向任务发布消息，这个特性不仅简化了代码，而且比使用外部消息队列更加有效，任务内建消息队列相关函数在文件 os_task.c 中。消息任务内建消息队列是可选项，如果要使用任务内建消息队列的话宏 OS_CFG_TASK_Q_EN 必须置 1。任务内嵌消息队列相关函数如表 11.4.1 所示：

函数名	描述
OSTaskQPend()	等待消息
OSTaskQPendAbort()	取消等待消息
OSTaskQPost()	向任务发送一条消息
OSTaskQFlush()	清空任务的消息队列

表 11.4.1 任务内建消息队列相关函数

11.4.1 等待任务内建消息

函数 OSTaskQPend()用来请求消息，该函数让任务直接接收从其他任务或者 ISR 中发送来的消息，不需要经过中间的消息队列，函数原型如下：

```
void   *OSTaskQPend (OS_TICK        timeout,
                     OS_OPT         opt,
                     OS_MSG_SIZE    *p_msg_size,
                     CPU_TS         *p_ts,
                     OS_ERR         *p_err)
```

timeout： 等待消息的超时时间，如果在指定的时间没有接收到消息的话，任务就会被唤醒，接着运行。这个参数也可以设置为 0，表示任务将一直等待下去，直到接收到消息。

opt: 用来选择是否使用阻塞模式，有两个选项可以选择。

OS_OPT_PEND_BLOCKING 如果没有任何消息存在的话就阻塞任务，一直等待，直到接收到消息。

OS_OPT_PEND_NON_BLOCKING 如果消息队列没有任何消息的话任务就直接返回。

p_msg_size: 指向存放消息大小的变量。

p_ts: 指向一个时间戳，表明什么时候接收到消息。如果这个指针被赋值为 NULL 的话，说明用户没有要求时间戳。

p_err: 用来保存调用此函数后返回的错误码。

11.4.2 发送任务内建消息

函数 OSTaskQPost()可以通过一个任务的内建消息队列向这个任务发送一条消息，同外置的消息队列一样，一条消息就是一个指针，函数原型如下：

```
void    OSTaskQPost (OS_TCB          *p_tcb,
                     void            *p_void,
                     OS_MSG_SIZE     msg_size,
                     OS_OPT          opt,
                     OS_ERR          *p_err)
```

p_tcb: 指向接收消息的任务的 TCB，可以通过指定一个 NULL 指针或该函数调用者的 TCB 地址来向该函数的调用者自己发送一条消息。

p_void: 发送给一个任务的消息。

msg_size: 指定发送的消息的大小(字节数)。

opt: 指定发送操作的类型，LIFO 和 FIFO 只能二选一。

OS_OPT_POST_FIFO 待发送消息保存在消息队列的末尾

OS_OPT_POST_LIFO 待发送的消息保存在消息队列的开头

上面两个选项可以搭配下面这个选项一起使用。

OS_OPT_POST_NO_SCHED 指定该选项时，在发送后不会进行任务调度，因此，该函数的调用者还可以继续运行。

p_err: 用来保存调用此函数后返回的错误码。

11.5 任务内建消息队列实验

11.5.1 实验程序设计

例 11-2：设计一个应用程序，该程序有 3 任务和一个定时器。任务 start_task 用于创建其他 2 个任务。main_task 任务为主任务，用于检测按键，当检测到按键 KWY_UP 被按下的时候就开启或关闭定时器 1， main_task 任务还用于检测 msgdis_task 任务的内建消息队列的总大小和剩余空间大小，并且控制 LED0 的闪烁。

定时器 1 的回调函数 tmr1_callback 通过任务 msgdis_task 内建的消息队列将定时器 1 的运行次数作为信息发送给任务 msgdis_task，任务 msgdis_task 将自带的队列中的消息显示在 LCD 上。

答：实验关键代码如下，实验完整工程见“例 11-2 UCOSIII 任务内建消息队列”。

既然要使用任务 msgdis_task 的内建消息队列，那么在创建任务 msgdis_task 的时候就需要指定内建消息队列的大小，大小通过一个宏来设定：

```
#define TASK_Q_NUM 4          //任务内建消息队列的长度
```

由于 msgdis_task 任务是在 start_task 任务中创建的，所以我们来看一下 start_task 的任务函数，如下:

```
//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

    CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
   OSStatTaskCPUUsageInit(&err);  //统计任务
```

```
#endif

#ifdef CPU_CFG_INT_DIS_MEAS_EN          //如果使能了测量中断关闭时间
    CPU_IntDisMeasMaxCurReset();
#endif

#if   OS_CFG_SCHED_ROUND_ROBIN_EN   //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
#endif

    OS_CRITICAL_ENTER();  //进入临界区
    //创建定时器 1
    OSTmrCreate((OS_TMR*          )&tmr1,       //定时器 1
                (CPU_CHAR *       )"tmr1",      //定时器名字
                (OS_TICK          )0,           //0ms
                (OS_TICK          )50,           //50*10=500ms
                (OS_OPT           )OS_OPT_TMR_PERIODIC, //周期模式
                (OS_TMR_CALLBACK_PTR)tmr1_callback,//定时器 1 回调函数
                (void*            )0,           //参数为 0
                (OS_ERR*          )&err);       //返回的错误码
    //创建主任务
    OSTaskCreate((OS_TCB *          )&Main_TaskTCB,
                 (CPU_CHAR*         )"Main task",
                 (OS_TASK_PTR       )main_task,
                 (void*             )0,
                 (OS_PRIO           )MAIN_TASK_PRIO,
                 (CPU_STK*          )&MAIN_TASK_STK[0],
                 (CPU_STK_SIZE      )MAIN_STK_SIZE/10,
                 (CPU_STK_SIZE      )MAIN_STK_SIZE,
                 (OS_MSG_QTY        )0,
                 (OS_TICK           )0,
                 (void*             )0,
                 (OS_OPT            )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                 (OS_ERR *          )&err);
    //创建 MSGDIS 任务
    OSTaskCreate((OS_TCB *          )&Msgdis_TaskTCB,
                 (CPU_CHAR*         )"Msgdis task",
                 (OS_TASK_PTR       )msgdis_task,
                 (void*             )0,
                 (OS_PRIO           )MSGDIS_TASK_PRIO,
                 (CPU_STK*          )&MSGDIS_TASK_STK[0],
                 (CPU_STK_SIZE      )MSGDIS_STK_SIZE/10,
```

```
                    (CPU_STK_SIZE     )MSGDIS_STK_SIZE,
                    (OS_MSG_QTY       )TASK_Q_NUM,                (1)
                    (OS_TICK          )0,
                    (void*            )0,
                    (OS_OPT           )OS_OPT_TASK_STK_CHK|\
                                       OS_OPT_TASK_STK_CLR,
                    (OS_ERR *         )&err);
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err);  //删除 start_task 任务自身
}
```

(1)、由于要使用任务 msgdis_task 的内建消息队列，因此在创建任务 msgdis_task 的时候我们需要设置任务 msgdis_task 内建消息队列的大小，也就是函数 OSTaskCreate()的参数 q_size；

我们在来看一下软件定时器 1 的回调函数、任务函数 main_task 和 msgdis_task。

```
//定时器 1 的回调函数
void tmr1_callback(void *p_tmr,void *p_arg)
{
    u8 *pbuf;
    static u8 msg_num;
    OS_ERR err;
    pbuf = mymalloc(SRAMIN,10);  //申请 10 个字节
    if(pbuf)  //申请内存成功
    {
        msg_num++;
        sprintf((char*)pbuf,"ALIENTEK %d",msg_num);
        //发送消息
        OSTaskQPost((OS_TCB*        )&Msgdis_TaskTCB,  //向任务 Msgdis 发送消息(1)
                    (void*          )pbuf,
                    (OS_MSG_SIZE    )10,
                    (OS_OPT         )OS_OPT_POST_FIFO,
                    (OS_ERR*        )&err);
        if(err != OS_ERR_NONE)
        {
            myfree(SRAMIN,pbuf); //释放内存
            OSTmrStop(&tmr1,OS_OPT_TMR_NONE,0,&err); //停止定时器 1
            tmr1sta = !tmr1sta;
            LCD_ShowString(40,150,100,16,16,"TMR1 STOP! ");
        }
    }
}

//主任务的任务函数
void main_task(void *p_arg)
{
```

```
    u8 key,num;
    OS_ERR err;
    u8 *p;
    while(1)
    {
        key = KEY_Scan(0);   //扫描按键
        if(key==WKUP_PRES)
        {
            tmr1sta = !tmr1sta;
            if(tmr1sta)
            {
                OSTmrStart(&tmr1,&err);
                LCD_ShowString(40,150,100,16,16,"TMR1 START!");
            }
            else
            {
                OSTmrStop(&tmr1,OS_OPT_TMR_NONE,0,&err); //停止定时器 1
                LCD_ShowString(40,150,100,16,16,"TMR1 STOP! ");
            }
        }
        num++;
        if(num%10==0) check_msg_queue(p);//检查 DATA_Msg 消息队列的容量
        if(num==50)
        {
            num=0;
            LED0 = ~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}

//显示消息队列中的消息
void msgdis_task(void *p_arg)
{
    u8 *p;
    OS_MSG_SIZE size;
    OS_ERR err;
    while(1)
    {
        //请求消息
        p=OSTaskQPend((OS_TICK          )0,                                    (2)
                      (OS_OPT           )OS_OPT_PEND_BLOCKING,
                      (OS_MSG_SIZE*     )&size,
```

```
                        (CPU_TS*        )0,
                        (OS_ERR*        )&err );
        LCD_ShowString(40,270,100,16,16,p);
        myfree(SRAMIN,p);     //释放内存
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err); //延时 1s
    }
}
```

(1)、定时器 1 回调函数中向任务 msgdis_task 的内建消息队列中发送消息。

(2)、任务 msgdis_task 从自身自带的消息队列中请求消息。

11.5.2 实验程序运行结果

代码编译完成下载到开发板中观察和分析实验现象，此时的 LCD 初始界面如图 11.5.1 所示。

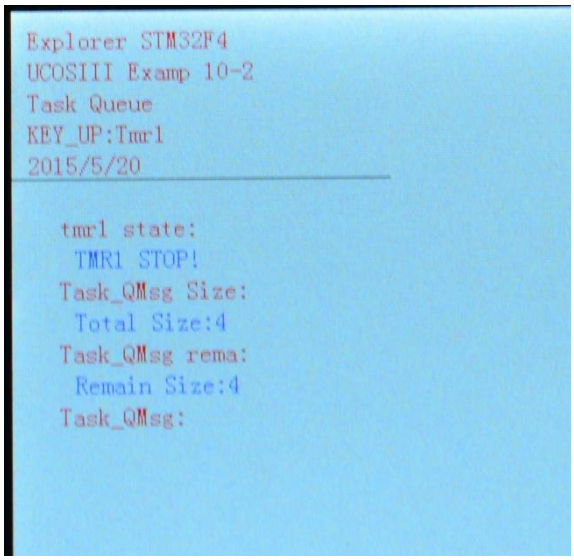


图 11.5.1 LCD 初始界面

从 11.5.1 中可以看出任务 msgdis_task 的内建消息队列的总大小为 4，这个和我们设置的一样。由于此时定时器 1 并没有启动，所以内建消息队列的剩余大小也为 4。 按下 KEY_UP 键开启定时器 1，那么定时器 1 的回调函数就会每隔 500ms 向消息队列任务 msgdis_task 的内建消息队列中发送一条消息，如图 11.5.2 所示。

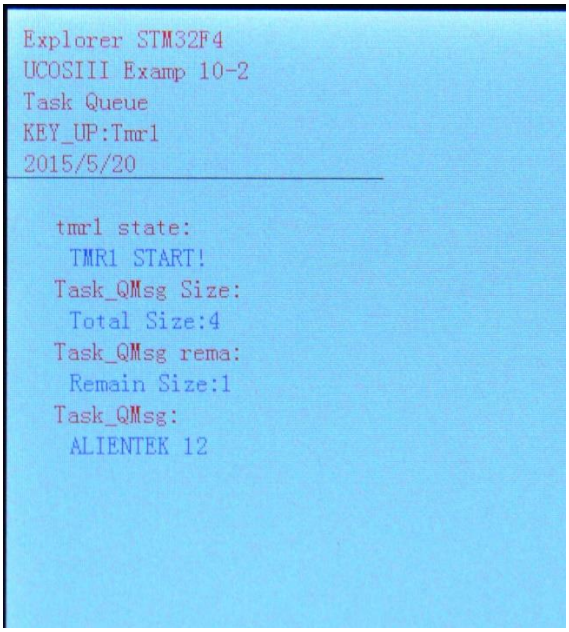


图 11.5.2 开启定时器 1

如图 11.5.2 所示，此时消息队列还剩下 1 个可用空间，定时器 1 还会一直向消息队列发送消息，直到消息队列满了，就会关闭定时器 1，停止发送，如图 11.5.3 所示。

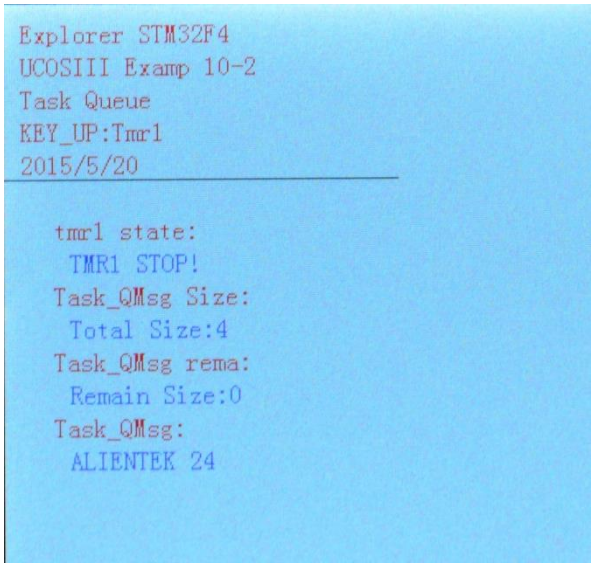


图 11.5.3 任务内建消息队列剩余空间为 0

从图 11.5.3 中可以看出此时任务内建消息队列剩余空间为 0，那么定时器 1 回调函数中再次调用函数 OSTaskQPost()向任务 msgdis_task 内建的消息队列发送数据的话就会发送失败，此时 err 就为 OS_ERR_MSG_POOL_EMPTY，提示消息队列空了，err 不等于 OS_ERR_NONE，那么就会关闭定时器 1，停止向消息队列中发送数据，除非再次手动开启定时器 1，也就是按下 KEY1 键。

大家可能会注意到，消息队列中剩余空间为 0，定时器 1 关闭了，但是此时任务 msgdis_task 还是能够接收到消息，并且在 LCD 上显示，而且任务 msgdis_task 的内建消息队列的剩余空间大小会增大，直到恢复为总大小才会停止，如图 11.5.4 所示。

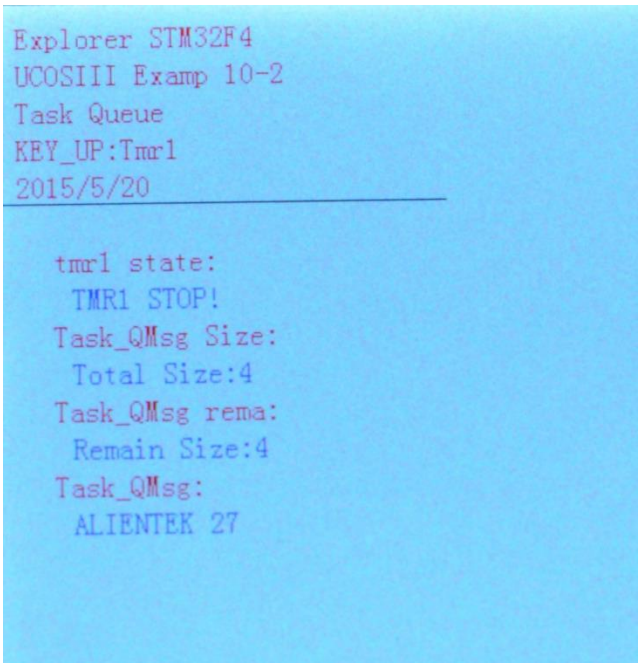


图 11.5.4 msgdis_task 任务停止运行

这是因为虽然定时器 1 关闭了，没有数据发送到消息队列中，但是此时消息队列中还有没有处理掉的数据，因此 msgdis_task 任务还会一直运行，直到处理完消息队列中的所有数据，每处理掉一个数据，任务 msgdis_task 内建消息队列的空闲大小就会加 1，当处理完所有的数据，空闲大小肯定就会等于总大小了。

第十二章 事件标志组

前面我们讲过可以使用信号量来完成任务同步，这里我们再讲解一下另外一种任务同步的方法，就是事件标志组，事件标志组用来解决一个任务和多个事件之间的同步，本章分为以下几个部分。

12.1 事件标志组
12.2 事件标志组相关函数
12.3 事件标志组实验

12.1 事件标志组

有时候一个任务可能需要和多个事件同步，这个时候就需要使用事件标志组。事件标志组与任务之间有两种同步机制：“或”同步和“与”同步，当任何一个事件发生，任务都被同步的同步机制是“或”同步；需要所有的事件都发生任务才会被同步的同步机制是“与”同步，这两种同步机制如图 12.1.1 所示。

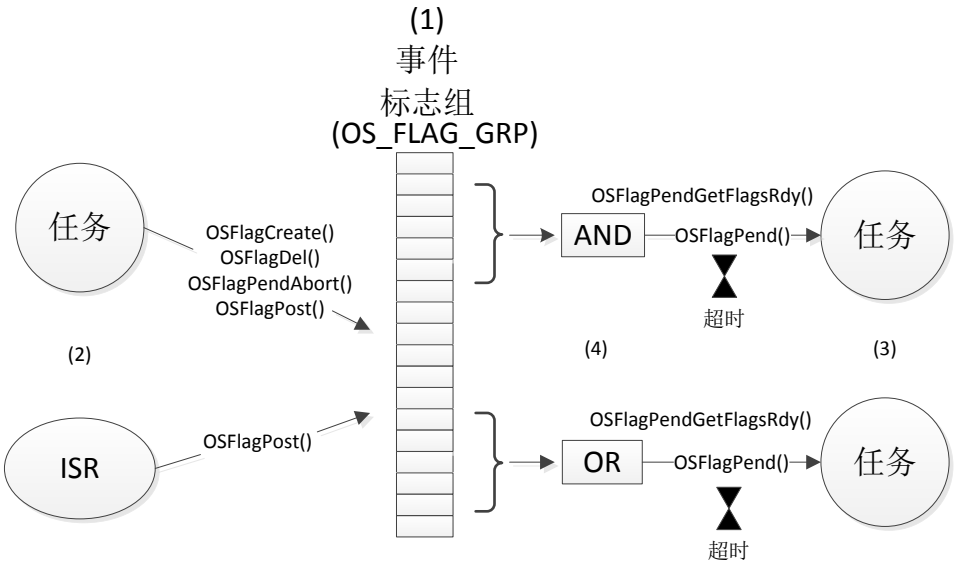


图 12.1.1 事件标志组

(1) 在 UCOSIII 中事件标志组是 OS_FLAG_GRP，在 os.h 文件中有定义，事件标志组中也包含了一串任务，这些任务都在等待着事件标志组中的部分(或全部)事件标志被置 1 或被清零，在使用之前，必须创建事件标志组。

(2) 任务和 ISR(中断服务程序)都可以发布事件标志，但是，只有任务可以创建、删除事件标志组以及取消其他任务对事件标志组的等待。

(3) 任务可以通过调用函数 OSFlagPend()等待事件标志组中的任意个事件标志，调用函数 OSFlagPend()的时候可以设置一个超时时间，如果过了超时时间请求的事件还没有被发布，那么任务就会重新进入就绪态。

(4) 我们可以设置同步机制为“或”同步还是“与”同步。

UCOSIII 中关于事件标志组的 API 函数如表 12.1.1 所示，一般情况下我们只使用 OSFlagCreate()、OSFlagPend()和 OSFlagPost()这三个函数。

函数	描述
OSFlagCreate()	创建事件标志组
OSFlagDel()	删除事件标志组
OSFlagPend()	等待事件标志组
OSFlagPendAbort()	取消等待事件标志组
OSFlagPendGetFlagsRdy()	获取使任务就绪的事件标志
OSFlagPost()	向事件标志组发布标志

表 12.1.1 事件标志组 API 函数

12.2 事件标志组相关函数

12.2.1 创建事件标志组

在使用事件标志组之前，需要调用函数 OSFlagCreate()创建一个事件标志组，OSFlagCreate()函数原型如下。

```
void    OSFlagCreate ( OS_FLAG_GRP     *p_grp,
                       CPU_CHAR        *p_name,
                       OS_FLAGS        flags,
                       OS_ERR          *p_err)
```

p_grp： 指向事件标志组，事件标志组的存储空间需要应用程序进行实际分配，我们可以按照下面的例子来定义一个事件标志组。

OS_FLAG_GRP EventFlag;

p_name： 事件标志组的名字。

flags： 定义事件标志组的初始值。

p_err： 用来保存调用此函数后返回的错误码。

12.2.2 等待事件标志组

等待一个事件标志组需要调用函数 OSFlagPend()，函数原型如下。

```
OS_FLAGS    OSFlagPend ( OS_FLAG_GRP    *p_grp,
                         OS_FLAGS       flags,
                         OS_TICK        timeout,
                         OS_OPT         opt,
                         CPU_TS         *p_ts,
                         OS_ERR         *p_err)
```

OSFlagPend()允许将事件标志组里事件标志的“与或”组合状态设置成任务的等待条件。任务等待的条件可以是标志组里任意一个标志置位或清零，也可以是所有事件标志都置位或清零。如果任务等待的事件标志组不满足设置的条件，那么该任务被置位挂起状态，直到等待的事件标志组满足条件、指定的超时时间到、事件标志被删除或另一个任务终止了该任务的挂起状态。

p_grp： 指向事件标志组。

flags： bit 序列，任务需要等待事件标志组的哪个位就把这个序列对应的位置 1，根据设置这个序列可以是 8bit、16bit 或者 32 比他。比如任务需要等待时间标志组的 bit0 和 bit1 时(无论是等待置位还是清零)，flag 是的值就为 0X03。

timeout： 指定等待事件标志组的超时时间(时钟节拍数)，如果在指定的超时时间内所等待的一个或多个事件没有发生，那么任务恢复运行。如果此值设置为 0，则任务就将一直等待下去，直到一个或多个事件发生。

opt： 决定任务等待的条件是所有标志置位、所有标志清零、任意一个标志置位还是任意一个标志清零，具体的定义如下。

OS_OPT_PEND_FLAG_CLR_ALL 等待事件标志组所有的位清零

OS_OPT_PEND_FLAG_CLR_ANY 等待事件标志组中任意一个标志清零

OS_OPT_PEND_FLAG_SET_ALL 等待事件标志组中所有的位置位

OS_OPT_PEND_FLAG_SET_ANY 等待事件标志组中任意一个标志置位

调用上面四个选项的时候还可以搭配下面三个选项。

OS_OPT_PEND_FLAG_CONSUME 用来设置是否继续保留该事件标志的状态。
OS_OPT_PEND_NON_BLOCKING 标志组不满足条件时不挂起任务。
OS_OPT_PEND_BLOCKING 标志组不满足条件时挂起任务。

这里应该注意选项 OS_OPT_PEND_FLAG_CONSUME 的使用方法，如果我们希望任务等待事件标志组的任意一个标志置位，并在满足条件后将对应的标志清零那么就可以搭配使用选项 OS_OPT_PEND_FLAG_CONSUME。

p_ts: 指向一个时间戳，记录了发送、终止和删除事件标志组的时刻，如果为这个指针赋值 NULL，则函数的调用者将不会收到时间戳。

p_err: 用来保存调用此函数后返回的错误码。

12.2.3 向事件标志组发布标志

调用函数 OSFlagPost()可以对事件标志组进行置位或清零，函数原型如下。

```
OS_FLAGS    OSFlagPost (  OS_FLAG_GRP    *p_grp,
                          OS_FLAGS       flags,
                          OS_OPT         opt,
                          OS_ERR         *p_err)
```

一般情况下，需要进行置位或者清零的标志由一个掩码确定（参数 flags）。OSFlagPost()修改完事件标志后，将检查并使那些等待条件已经满足的任务进入就绪态。该函数可以对已经置位或清零的标志进行重复置位和清零操作。

p_grp: 指向事件标志组。

flags: 决定对哪些位清零和置位，当 opt 参数为 OS_OPT_POST_FLAG_SET 的时，参数 flags 中置位的位就会在事件标志组中对应的位也将被置位。当 opt 为 OS_OPT_POST_FLAG_CLR 的时候参数 flags 中置位的位在事件标志组中对应的位将被清零。

opt: 决定对标志位的操作，有两种选项。
OS_OPT_POST_FLAG_SET 对标志位进行置位操作
OS_OPT_POST_FLAG_CLR 对标志位进行清零操作

p_err: 保存调用此函数后返回的错误码。

12.3 时间标志组实验

12.3.1 实验程序设计

例 12-1：设计一个程序，只有按下 KEY0 和 KEY1(不需要同时按下)时任务 flagsprocess_task 任务才能执行。

答：分析上面的程序设计要求，我们可以使用事件标志组来实现，按下 KEY0 和 KEY1 作为两个不同的事件，只有这两个事件同时发生了才能执行任务 flagsprocess_task。实验代码如下，实验完整工程见“例 12-1 UCOSIII 事件标志组”。

```
#include "sys.h"
#include "delay.h"
#include "usart.h"
#include "led.h"
#include "lcd.h"
#include "sram.h"
```

```
#include "malloc.h"
#include "beep.h"
#include "key.h"
#include "includes.h"

//任务优先级
#define START_TASK_PRIO      3
//任务堆栈大小
#define START_STK_SIZE       128
//任务控制块
OS_TCB StartTaskTCB;
//任务堆栈
CPU_STK START_TASK_STK[START_STK_SIZE];
//任务函数
void start_task(void *p_arg);

//任务优先级
#define MAIN_TASK_PRIO       4
//任务堆栈大小
#define MAIN_STK_SIZE        128
//任务控制块
OS_TCB Main_TaskTCB;
//任务堆栈
CPU_STK MAIN_TASK_STK[MAIN_STK_SIZE];
void main_task(void *p_arg);

//任务优先级
#define FLAGSPROCESS_TASK_PRIO     5
//任务堆栈大小
#define FLAGSPROCESS_STK_SIZE      128
//任务控制块
OS_TCB Flagsprocess_TaskTCB;
//任务堆栈
CPU_STK FLAGSPROCESS_TASK_STK[FLAGSPROCESS_STK_SIZE];
//任务函数
void flagsprocess_task(void *p_arg);

//LCD 刷屏时使用的颜色
int lcd_discolor[14]={   WHITE, BLACK, BLUE,   BRED,
                         GRED,  GBLUE, RED,    MAGENTA,
                         GREEN, CYAN,  YELLOW,BROWN,
                         BRRED, GRAY };
```

```
//////////////////////事件标志组/////////////////////////////
#define KEY0_FLAG          0x01
#define KEY1_FLAG          0x02
#define KEYFLAGS_VALUE     0X00
OS_FLAG_GRP    EventFlags;          //定义一个事件标志组

//加载主界面
void ucos_load_main_ui(void)
{
    POINT_COLOR = RED;
    LCD_ShowString(30,10,200,16,16,"Explorer STM32F4");
    LCD_ShowString(30,30,200,16,16,"UCOSIII Examp 12-1");
    LCD_ShowString(30,50,200,16,16,"Event Flags");
    LCD_ShowString(30,70,200,16,16,"ATOM@ALIENTEK");
    LCD_ShowString(30,90,200,16,16,"2014/12/9");
    POINT_COLOR = BLACK;
    LCD_DrawRectangle(5,130,234,314);  //画矩形
    POINT_COLOR = BLUE;
    LCD_ShowString(30,110,220,16,16,"Event Flags Value:0");
}

//主函数
int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(168);           //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);         //串口初始化
    LED_Init();                //LED 初始化
    LCD_Init();                //LCD 初始化
    KEY_Init();                //按键初始化
    BEEP_Init();               //初始化蜂鸣器
    FSMC_SRAM_Init();    //初始化 SRAM
    my_mem_init(SRAMIN); //初始化内部 RAM
    ucos_load_main_ui();       //加载主 UI

    OSInit(&err);              //初始化 UCOSIII
    OS_CRITICAL_ENTER();   //进入临界区
    //创建开始任务
    OSTaskCreate((OS_TCB *          )&StartTaskTCB,              //任务控制块
                 (CPU_CHAR*     )"start task",                   //任务名字
```

```
                (OS_TASK_PTR  )start_task,                  //任务函数
                (void*        )0,                           //传递给任务函数的参数
                (OS_PRIO      )START_TASK_PRIO,             //任务优先级
                (CPU_STK*     )&START_TASK_STK[0],          //任务堆栈基地址
                (CPU_STK_SIZE )START_STK_SIZE/10,           //任务堆栈深度限位
                (CPU_STK_SIZE )START_STK_SIZE,              //任务堆栈大小
                (OS_MSG_QTY   )0,                           //任务内部消息队列能够接收
                                                            //的最大消息数目,为 0 时禁止
                                                            //接收消息
                (OS_TICK      )0,                           //当使能时间片轮转时的时间
                                                            //片长度，为 0 时为默认长度
                (void*        )0,                           //用户补充的存储区
                (OS_OPT       )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                (OS_ERR*      )&err);                       //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);          //开启 UCOSIII
}


//开始任务函数
void start_task(void *p_arg)
{
    OS_ERR err;
    CPU_SR_ALLOC();
    p_arg = p_arg;

#if   OS_CFG_SCHED_ROUND_ROBIN_EN   //当使用时间片轮转的时候
     //使能时间片轮转调度功能,时间片长度为 1 个系统时钟节拍，既 1*5=5ms
    OSSchedRoundRobinCfg(DEF_ENABLED,1,&err);
#endif

    OS_CRITICAL_ENTER();   //进入临界区
    //创建一个事件标志组
    OSFlagCreate((OS_FLAG_GRP* )&EventFlags,            //指向事件标志组
                 (CPU_CHAR*    )"Event Flags",          //名字
                 (OS_FLAGS     )KEYFLAGS_VALUE,         //事件标志组初始值
                 (OS_ERR*      )&err);                  //错误码
    //创建主任务
    OSTaskCreate((OS_TCB *        )&Main_TaskTCB,
                 (CPU_CHAR*    )"Main task",
                 (OS_TASK_PTR  )main_task,
                 (void*        )0,
                 (OS_PRIO      )MAIN_TASK_PRIO,
```

```
                    (CPU_STK*       )&MAIN_TASK_STK[0],
                    (CPU_STK_SIZE )MAIN_STK_SIZE/10,
                    (CPU_STK_SIZE )MAIN_STK_SIZE,
                    (OS_MSG_QTY   )0,
                    (OS_TICK        )0,
                    (void*          )0,
                    (OS_OPT         )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                    (OS_ERR*        )&err);
    //创建 MSGDIS 任务
    OSTaskCreate((OS_TCB*           )&Flagsprocess_TaskTCB,
                    (CPU_CHAR*      )"Flagsprocess task",
                    (OS_TASK_PTR  )flagsprocess_task,
                    (void*          )0,
                    (OS_PRIO        )FLAGSPROCESS_TASK_PRIO,
                    (CPU_STK*       )&FLAGSPROCESS_TASK_STK[0],
                    (CPU_STK_SIZE )FLAGSPROCESS_STK_SIZE/10,
                    (CPU_STK_SIZE )FLAGSPROCESS_STK_SIZE,
                    (OS_MSG_QTY   )0,
                    (OS_TICK        )0,
                    (void*          )0,
                    (OS_OPT         )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                    (OS_ERR*        )&err);
    OS_CRITICAL_EXIT();     //退出临界区
    OSTaskDel((OS_TCB*)0,&err);   //删除 start_task 任务自身
}

//主任务的任务函数
void main_task(void *p_arg)
{
    u8 key,num;
    OS_FLAGS flags_num;
    OS_ERR err;
    while(1)
    {
        key = KEY_Scan(0);   //扫描按键
        if(key == KEY0_PRES)
        {
            //向事件标志组 EventFlags 发送标志
            flags_num=OSFlagPost((OS_FLAG_GRP*)&EventFlags,
                                 (OS_FLAGS    )KEY0_FLAG,
                                 (OS_OPT      )OS_OPT_POST_FLAG_SET,
                                 (OS_ERR*     )&err);
            LCD_ShowxNum(174,110,flags_num,1,16,0);
```

```
            printf("事件标志组 EventFlags 的值:%d\r\n",flags_num);
        }
        else if(key == KEY1_PRES)
        {
            //向事件标志组 EventFlags 发送标志
            flags_num=OSFlagPost((OS_FLAG_GRP*)&EventFlags,
                                 (OS_FLAGS    )KEY1_FLAG,
                                 (OS_OPT      )OS_OPT_POST_FLAG_SET,
                                 (OS_ERR*     )&err);
            LCD_ShowxNum(174,110,flags_num,1,16,0);
            printf("事件标志组 EventFlags 的值:%d\r\n",flags_num);
        }
        num++;
        if(num==50)
        {
            num=0;
            LED0 = ~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}

//事件标志组处理任务
void flagsprocess_task(void *p_arg)
{
    u8 num;
    OS_ERR err;
    while(1)
    {
        //等待事件标志组
        OSFlagPend((OS_FLAG_GRP*)&EventFlags,
                   (OS_FLAGS    )KEY0_FLAG+KEY1_FLAG,
                   (OS_TICK     )0,
                   (OS_OPT
        )OS_OPT_PEND_FLAG_SET_ALL+OS_OPT_PEND_FLAG_CONSUME,
                   (CPU_TS*     )0,
                   (OS_ERR*         )&err);
        num++;
        LED1 = ~LED1;
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);
        printf("事件标志组 EventFlags 的值:%d\r\n",EventFlags.Flags);
        LCD_ShowxNum(174,110,EventFlags.Flags,1,16,0);
    }
```

```
}
```

首先需要定义事件标志组，并且定义了 KEY0 和 KEY1 的掩码以及事件标志组的初始值。函数 ucos_load_main_ui()为主界面，主要是在 LCD 上显示一些提示信息。main()函数为主函数，和前面的实验一样，主要是初始化外设和 UCOSIII，并且新建一个 start_task 任务。

start_task 任务里面创建了一个事件标志组 EventFlags 和两个任务：main_task 和 flagsprocess_task。

main_task()函数为主任务的任务函数，函数里面主要是获取按键值，如果按下 KEY0 的话就调用 OSFlagPost()向事件标志组 EventFlags 发布标志 KEY0_FLAG，如果按下 KEY1 的话就向事件标志组 EventFlags 发布 KEY1_FLAG。 在 main_task()函数中每调用一次 OSFlagPost()就在 LCD 上显示事件标志组 EventFlags 的当前值并且通过串口输出这个值，我们可以通过这个值的变化来观察事件标志组各个事件产生的过程。

flagsprocess_task()函数为事件标志组处理任务的任务函数，在这个函数中我们一直等待事件标志组 Eventflags 中相应的事件发生，当等待的事件发生的话就刷新 LCD 下方方框的背景颜色，并且控制 LED1 反转。

12.3.2 实验程序结果分析

代码编译完成下载到开发板中观察和分析实验现象，代码刚下进去后 LCD 显示如图 12.3.1 所示。

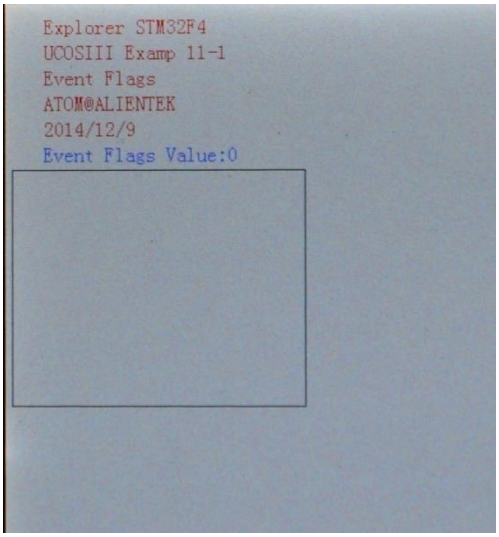


图 12.3.1 LCD 初始界面

从图 12.3.1 中可以看出此时事件标志组 EventFlags 为 0，没有任何事件发生，因此 LCD 下方方框内的背景颜色为白色的，当我们按下 KEY0 的时候，LCD 界面就如图 12.3.2 所示。

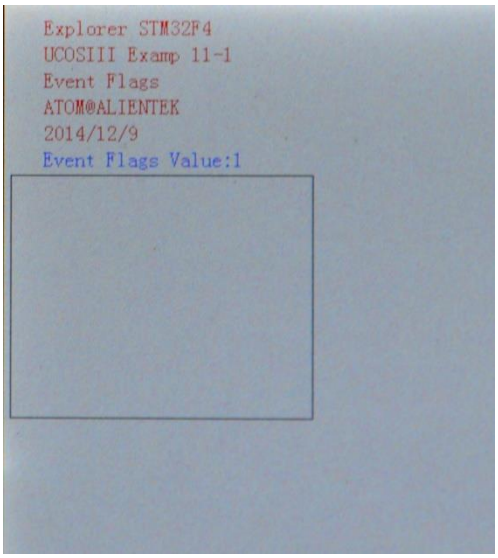


图 12.3.2 按下 KEY0 后的 LCD 界面

从图 12.3.2 中可以看出，此时事件标志组 EventFlags 的值为 1，因为我们按下 KEY0 的话 EventFlags 的 bit0 就会置 1，但是此时 KEY1 还没有按下因此下方方框内的背景还是为白色。此时我们再按下 KEY1 键，LCD 界面如图 12.3.3 所示。

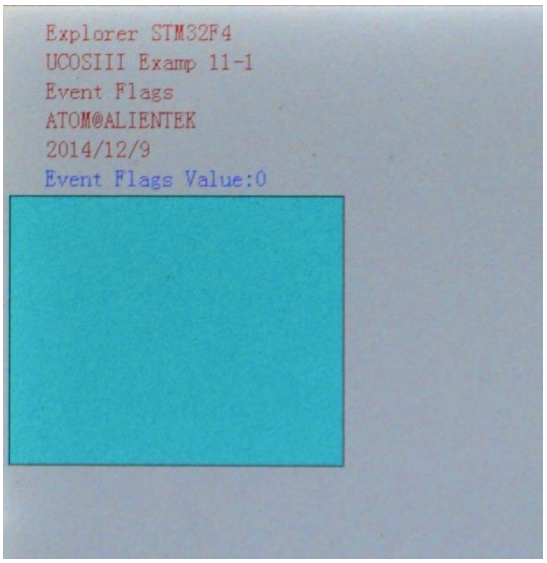


图 12.3.3 按下 KEY1 后的 LCD 界面

从图 12.3.3 中可以看出此时 EventFlags 的值为 0，这是因为我们按下 KEY1 后，任务 flagsprocess_task 等待的事件都发生了就会刷新一次下方方框内背景，并且 LED1 会反转。我们在调用函数 OSFlagPend()的时候设定了参数 opt 为 OS_OPT_PEND_FLAG_SET_ALL 和 OS_OPT_PEND_FLAG_CONSUME，因此会清除相应的标志的，因此此时的事件标志组的值就为 0 了。

注意！其实当我们按下 KEY1 的一瞬间事件标志组 EventFlags 的值应该为 3 的，但是很快会被任务 flagsprocess_task 刷新掉显示为 0，这个过程非常快，在 LCD 上是看不出来的，不过我们可以通过串口调试助手来观察，如图 12.3.4 所示。

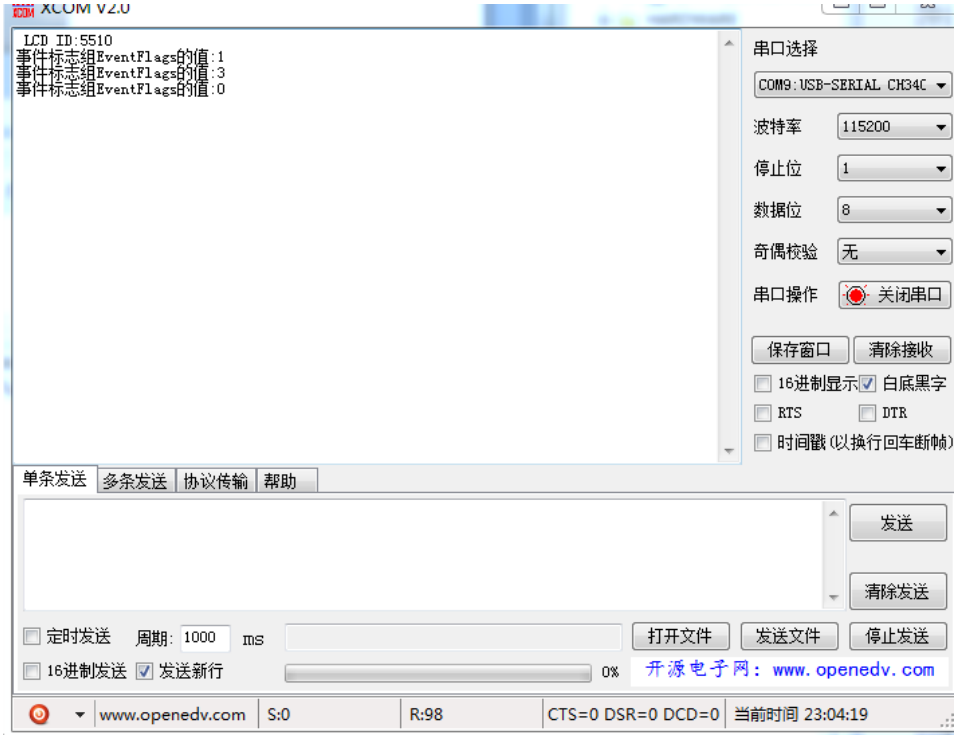


图 12.3.4 串口调试助手

从图 12.3.4 中的串口调试助手可以看出，当我们按下 KEY0 的时候 EventFlags 为 0，接着按下 KEY1 那么 EventFlags 就肯定为 3，这样任务 flagsprocess_task 等待的事件都发生了，任务 flagsprocess_task 得到执行 EventFlags 马上变为了 0。

第十三章 同时等待多个内核对象

在前几章中我们讲解了任务如何等待单个对象，比如信号量、互斥信号量、消息队列和时间标志组等。本章我们就讲解一下 UCOSIII 如何同时等待多个内核对象，在 UCOSIII 中只支持同时等待多个信号量和消息队列，不支持同时等待多个事件标志组和互斥信号量，本章分为如下几部分：

13.1 同时等待多个内核对象

13.2 OSPendMulti()函数

13.3 同时等待多个内核对象实验

13.1 同时等待多个内核对象

UCOSIII 中一个任务可以同时等待任意数量的信号量或者消息队列，当只要等到其中的任意一个的时候就会导致该任务进入就绪态，如图 13.1.1 所示。

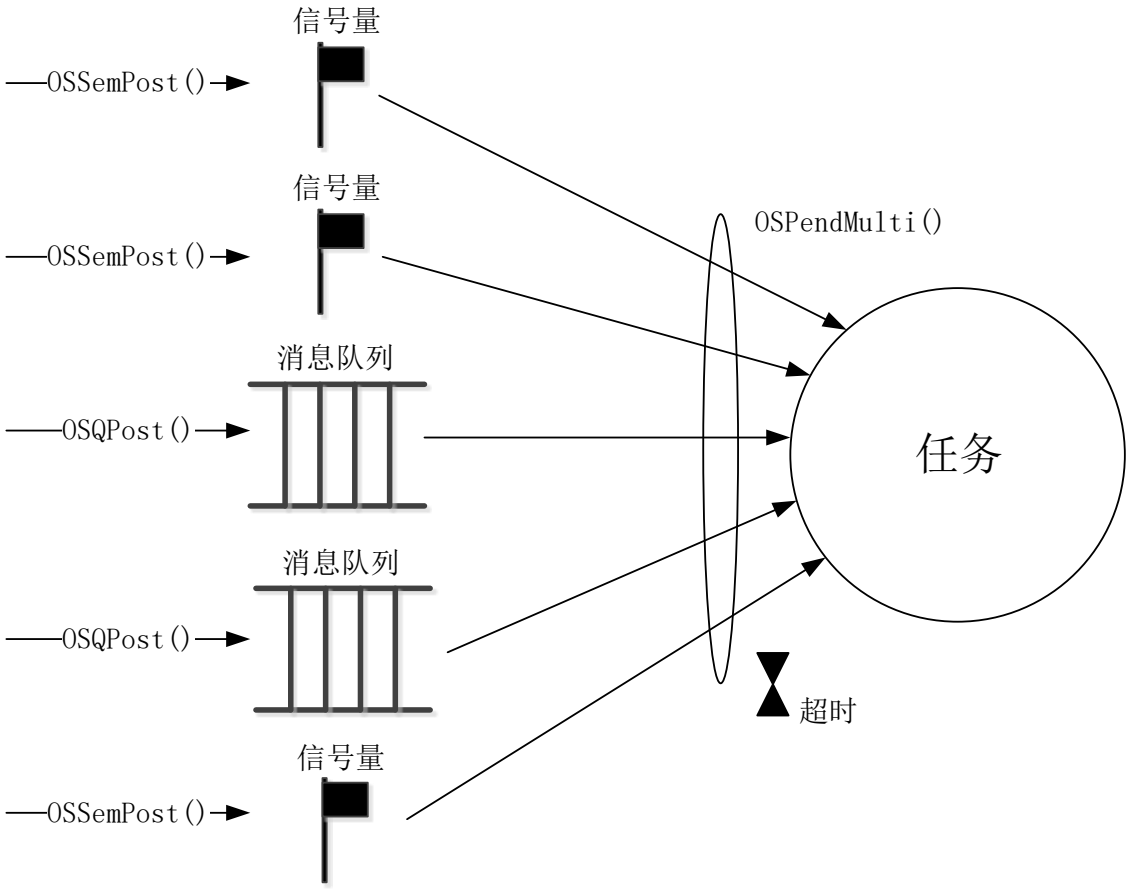


图 13.1.1 任务等待多个内核对象

在图 13.1.1 中任务通过调用函数 OSPendMulti()来等待多个内核对象，我们可以设定一个等待超时值，如果在指定的时间内没有一个内核对象被发布，那么将返回一个错误码，表示等待超时。

13.2 OSPendMulti()函数

函数 OSPendMulti()用来等待多个内核对象，调用 OSPendMulti()时，如果这些对象中有多个可用，则所有可用的信号量和消息都将返回给调用者，如果没有任何对象可用，则 OSPendMulti()将挂起当前任务，直到以下任一情况发生：

1、对象变为可用。

2、到达设定的超时时间。

3、一个或多个任务被删除或被终止。

4、一个或多个对象被删除。

如果一个对象变为可用，并且有多个任务在等待这个对象，则 UCOSIII 将恢复优先级最高的那个任务，函数 OSPendMulti()原型如下：

```
OS_OBJ_QTY    OSPendMulti ( OS_PEND_DATA    *p_pend_data_tbl,
                            OS_OBJ_QTY        tbl_size,
```

```
                        OS_TICK         timeout,
                        OS_OPT          opt,
                        OS_ERR          *p_err)
```

p_pend_data_tbl: 指向 OS_PEND_DATA 表的指针，调用者通过该表来查询函数的调用结果。调用该函数的时候首先必须初始化 OS_PEND_DATA 表中的每个元素的 PendObjPtr，使得各个指针指向被等待的对象。

tbl_siae: 表 p_pend_data_tbl 的大小，也就是所等待的内核对象数量。

timeout: 设定一个等待超时值(时钟节拍数)，用来设置任务等待对象发送的时间，如果为 0，表示这个任务将一直等待下去，直到对象被发送。

opt: 来选择是否使用阻塞模式，有两个选项可以选择。

OS_OPT_PEND_BLOCKING 如果没有任何消息存在的话就阻塞任务，一直等待，直到接收到消息。

OS_OPT_PEND_NON_BLOCKING 如果消息队列没有任何消息的话任务就直接返回。

p_err: 用来保存调用此函数后返回的错误码。

13.3 同时等待多个内核对象实验

13.3.1 实验程序设计

例 13-1：设计一个应用程序，该程序有 3 任务、2 个信号量和 1 个消息队列。任务 A 用于创建其他 2 个任务、2 个信号量和 1 个消息队列。B 任务为任务 1，用于检测按键，当检测到按键 KEY1 被按下就发送信号量 1，当 KEY2 被按下就发送信号量 2，当 KEY_UP 被按下就发送消息队列，任务 1 还用来控制 LED0 的闪烁。

任务 C 调用函数 OSPendMulti()来同时等待 2 个信号量和 1 个消息队列。

答：实验部分源码如下，实验完整工程见“例 13-1 UCOSIII 同时等待多个内核对象”。

```
//任务 1 的任务函数
void task1_task(void *p_arg)
{
    u8 key;
    OS_ERR err;
    u8 num;
    u8 *pbuf;
    static u8 msg_num;
    pbuf=mymalloc(SRAMIN,10);     //申请内存
    while(1)
    {
        key = KEY_Scan(0);   //扫描按键
        switch(key)
        {
            case KEY1_PRES:
                OSSemPost(&Test_Sem1,OS_OPT_POST_1,&err);//发送信号量 1          (1)
                break;
            case KEY0_PRES:
```

```
                OSSemPost(&Test_Sem2,OS_OPT_POST_1,&err);//发送信号量 2          (2)
            case WKUP_PRES:
                msg_num++;
                sprintf((char*)pbuf,"ALIENTEK %d",msg_num);

                //发送消息
                OSQPost((OS_Q*          )&Test_Q,                               (3)
                        (void*          )pbuf,
                        (OS_MSG_SIZE    )10,
                        (OS_OPT         )OS_OPT_POST_FIFO,
                        (OS_ERR*        )&err);
                break;
        }
        num++;
        if(num==50)
        {
            num=0;
            LED0=~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}

//等待多个内核对象的任务函数
void multi_task(void *p_arg)
{
    u8 num;
    OS_ERR err;
    OS_OBJ_QTY index;
    OS_PEND_DATA pend_multi_tbl[CORE_OBJ_NUM];                                  (4)

    pend_multi_tbl[0].PendObjPtr=(OS_PEND_OBJ*)&Test_Sem1;                      (5)
    pend_multi_tbl[1].PendObjPtr=(OS_PEND_OBJ*)&Test_Sem2;
    pend_multi_tbl[2].PendObjPtr=(OS_PEND_OBJ*)&Test_Q;

    while(1)
    {
        index=OSPendMulti((OS_PEND_DATA*    )pend_multi_tbl,                    (6)
                          (OS_OBJ_QTY       )CORE_OBJ_NUM,     //内核数量
                          (OS_TICK          )0,
                          (OS_OPT           )OS_OPT_PEND_BLOCKING,
                          (OS_ERR*          )&err);
        LCD_ShowNum(147,111,index,1,16);                        //显示当前有几个内核对
```

```
象准备好了
        num++;
        LCD_Fill(6,131,233,313,lcd_discolor[num%14]);          //刷屏
        LED1 = ~LED1;
        OSTimeDlyHMSM(0,0,1,0,OS_OPT_TIME_PERIODIC,&err);    //延时 1s
    }
}
```

(1)、发送信号量 Test_Sem1。

(2)、发送信号量 Test_Sem2。

(3)、发送消息队列 Test_Q。

(4)、定义一个 OS_PEND_DATA 类型的数组 pend_multi_tbl[]，数组大小为内核对象数量。

(5)、在函数 OSPendMulti()调用 pend_multi_tbl[]数组之前我们必须初始化数组中的每个元素，让数组中的每个元素的 PendObjPtr 指向被等待的内核对象。

(6)、调用函数 OSPendMulti()同时等待多个内核对象。

10.3.2 实验程序结果分析

代码编译完成下载到开发板中观察和分析实验现象，代码刚下进去后 LCD 显示如图 13.3.1 所示。

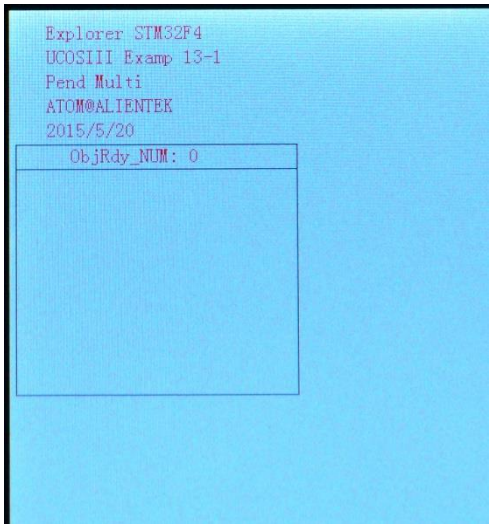


图 13.3.1 LCD 显示图

当我们按下 KEY0、KEY1 和 KEY_UP 其中的一个或者多个按键的时候(不是同时按下)，任务 multi_task 就会请求到一个或多个内核对象，此时 LCD 指定的区域的背景颜色就会改变，并且会显示此时请求到了的内核对象数目，如图 13.3.2 所示。

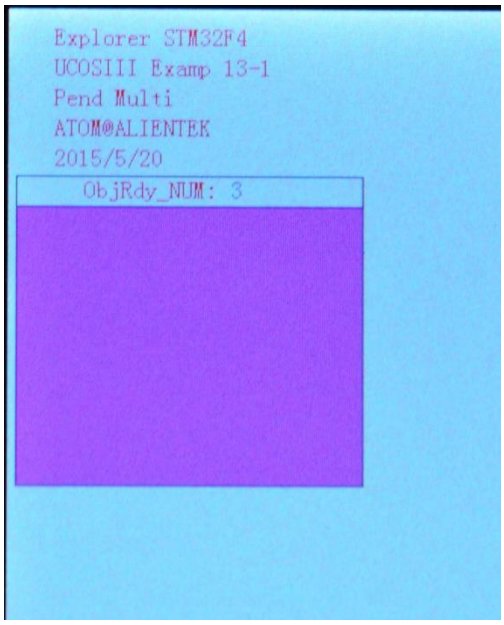


图 13.3.2 请求到多个内核对象

从图 13.3.2 中可以看出此时 ObjRdy_NUM 为 3，说明同时有 3 个内核对象请求到。

第十四章 存储管理

作为一个操作系统，内存管理是其必备的功能，在 UCOSIII 中也有内存管理模块，使用内存管理模块可以动态的分配和释放内存，这样可以高效的使用“昂贵”的内存资源，本章我们就学习一下 UCOSIII 的内存管理，本章分为如下几部分：

14.1 内存管理简介
14.2 存储区创建
14.3 存储块的使用
14.4 存储管理实验

14.1 存管理简介

内存管理是一个操作系统必备的系统模块，我们在用 VC++或者 Visual Studio 学习 C 语言的时候会使用 malloc()和 free()这两个函数来申请和释放内存。我们在使用 Keil MDK 编写 STM32 程序的时候就可以使用 malloc()和 free()，但是不建议这么用，这样的操作会将原来大块内存逐渐的分割成很多个小块内存，产生大量的内存碎片，最终导致应用不能申请到大小合适的连续内存。

UCOSIII 提供了自己的动态内存方案，UCOIII 将存储空间分成区和块，一个存储区有数个固定大小的库组成，如图 14.1.1 所示。

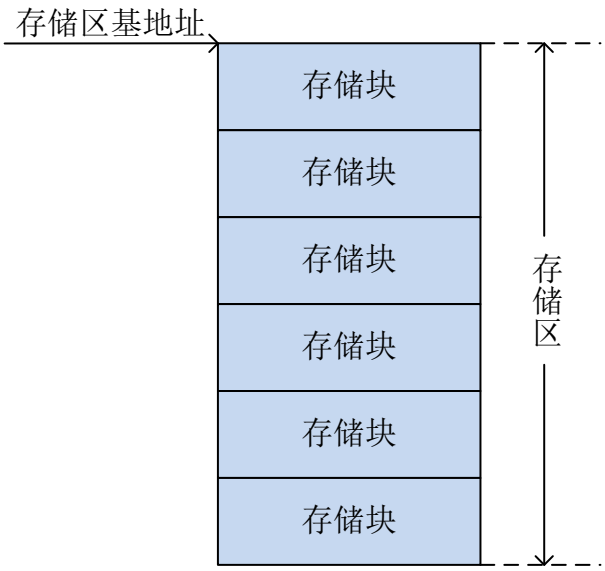


图 14.1.1 存储区和存储块

一般存储区是固定的，在程序中可以用数组来表示一个存储区，比如 u8 buffer[20][10]就表示一个有 20 个存储块，每个存储块 10 字节的存储区。如果我们定义的存储区在程序运行期间都不会被删除掉，一直有效，那么存储区内存也可以使用 malloc()来分配。在创建存储区以后应用程序就可以获得固定大小的存储块。

在实际使用中我们可以根据应用程序对内存需求的不同建立多个存储区，每个存储区中有不同大小、不同数量的存储块，应用程序可以根据所需内存不同从不同的存储区中申请内存使用，使用完以后在释放到相应的存储区中。

14.2 存储区创建

在使用内存管理之前首先要创建存储区，在创建存储区之前我们先了解一个重要的结构体，存储区控制块：OS_MEM，结构体 OS_MEM 如下，取掉了与调试有关的变量：

```
struct os_mem
{
    OS_OBJ_TYPE         Type;           //类型，必须为 OS_OBJ_TYPE_MEM
    void                *AddrPtr;       //指向存储区起始地址
    CPU_CHAR            *NamePtr;       //指向存储区名字
    void                *FreeListPtr;   //指向空闲存储块
    OS_MEM_SIZE          BlkSize;       //存储区中存储块大小，单位：字节
```

```
    OS_MEM_QTY          NbrMax;          //存储区中总的存储块数
    OS_MEM_QTY          NbrFree;         //存储区中空闲存储块数
};
```

创建存储区使用函数 OSMemCreate()，函数原型如下：

```
void   OSMemCreate (OS_MEM          *p_mem,
                    CPU_CHAR        *p_name,
                    void            *p_addr,
                    OS_MEM_QTY       n_blks,
                    OS_MEM_SIZE     blk_size,
                    OS_ERR          *p_err)
```

OSMemCreate()函数用于创建一个存储区，函数参数如下：

p_mem： 指向存储区控制块地址，一般有用户程序定义一个 OS_MEM 结构体。

p_name： 指向存储区的名字，我们可以给存储区取一个名字。

p_addr： 存储区所有存储空间基地址。

n_blks： 存储区中存储块个数。

blk_size： 存储块大小。

p_err： 返回的错误码。

我们在来看一下 OSMemCreate()函数的具体过程，函数源码如下：

```
void   OSMemCreate (OS_MEM          *p_mem,
                    CPU_CHAR        *p_name,
                    void            *p_addr,
                    OS_MEM_QTY       n_blks,
                    OS_MEM_SIZE     blk_size,
                    OS_ERR          *p_err)
{
#if OS_CFG_ARG_CHK_EN > 0u
    CPU_DATA        align_msk;
#endif
    OS_MEM_QTY      i;
    OS_MEM_QTY      loops;
    CPU_INT08U      *p_blk;
    void            **p_link;
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == DEF_TRUE) {
```

```
        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;
         return;
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {                    (1)
         *p_err = OS_ERR_MEM_CREATE_ISR;
         return;
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    if (p_addr == (void *)0) {                                    (2)
         *p_err     = OS_ERR_MEM_INVALID_P_ADDR;
         return;
    }
    if (n_blks < (OS_MEM_QTY)2) {                                 (3)
        *p_err = OS_ERR_MEM_INVALID_BLKS;
         return;
    }
    if (blk_size < sizeof(void *)) {                              (4)
         *p_err = OS_ERR_MEM_INVALID_SIZE;
         return;
    }
    align_msk = sizeof(void *) - 1u;
    if (align_msk > 0) {
         if (((CPU_ADDR)p_addr & align_msk) != 0u){               (5)
             *p_err = OS_ERR_MEM_INVALID_P_ADDR;
              return;
         }
         if ((blk_size & align_msk) != 0u) {                      (6)
             *p_err = OS_ERR_MEM_INVALID_SIZE;
              return;
         }
    }
#endif

    p_link = (void **)p_addr;                                     (7)
    p_blk  = (CPU_INT08U *)p_addr;                                (8)
    loops  = n_blks - 1u;                                         (9)
    for (i = 0u; i < loops; i++) {                                (10)
         p_blk +=  blk_size;                                      (11)
```

```
            *p_link = (void    *)p_blk;                                    (12)
             p_link = (void **)(void *)p_blk;                              (13)
        }                                                                  (14)
    *p_link= (void *)0;                                                    (15)

    OS_CRITICAL_ENTER();
    p_mem->Type             = OS_OBJ_TYPE_MEM;                             (16)
    p_mem->NamePtr          = p_name;                                      (17)
    p_mem->AddrPtr          = p_addr;                                      (18)
    p_mem->FreeListPtr = p_addr;                                           (19)
    p_mem->NbrFree          = n_blks;                                      (20)
    p_mem->NbrMax           = n_blks;                                      (21)
    p_mem->BlkSize          = blk_size;                                    (22)
#if OS_CFG_DBG_EN > 0u
    OS_MemDbgListAdd(p_mem);
#endif

    OSMemQty++;                                                            (23)

    OS_CRITICAL_EXIT_NO_SCHED();
    *p_err = OS_ERR_NONE;
}
```

(1)、判断 OSIntNestingCtr 是否大于 0，如果大于 0 的话说明在中断中调用函数 OSMemCreate()来创建存储区，如果是在中断中调用函数 OSMemCreate()将会返回错误码：OS_ERR_MEM_CREATE_ISR，说明中断中是不能调用函数 OSMemCreate()来创建存储区的。

(2)、存储区空间基地址不能为 0，否则地址无效，返回错误码：OS_ERR_MEM_INVALID_P_ADDR。

(3)、存储区中存储块数量 n_blks 最少为 2。

(4)、每个存储块大小不小于一个指针大小，在 Keil MDK 中一个指针大小为 4 字节，因此每个存储块大小要大于 4 字节。

(5)、判断存储区空间基地址是否 4 字节对齐，存储区的存储空间基地址必须 4 字节对齐。

(6)、存储区中每个存储块的大小要是 4 的倍数！

(7)~(14)、将存储区中的存储块连成一个空闲存储块链表。每个存储块中保存着下一个存储块的地址，因此在(4)中规定了每个存储块要大于 4 个字节，因为每个存储块至少要保存一个地址，而一个地址恰恰为 4 字节。

(15)、空闲存储块链表中最后一个存储块指向 0。

(16)~(22)、初始化结构体 OS_MEM 中的成员变量。

(23)、创建存储区成功以后，记录存储区数量的全局变量 OSMemQty 加一

调用函数 OSMemCreate()创建好的存储区如图 14.2.1 所示。

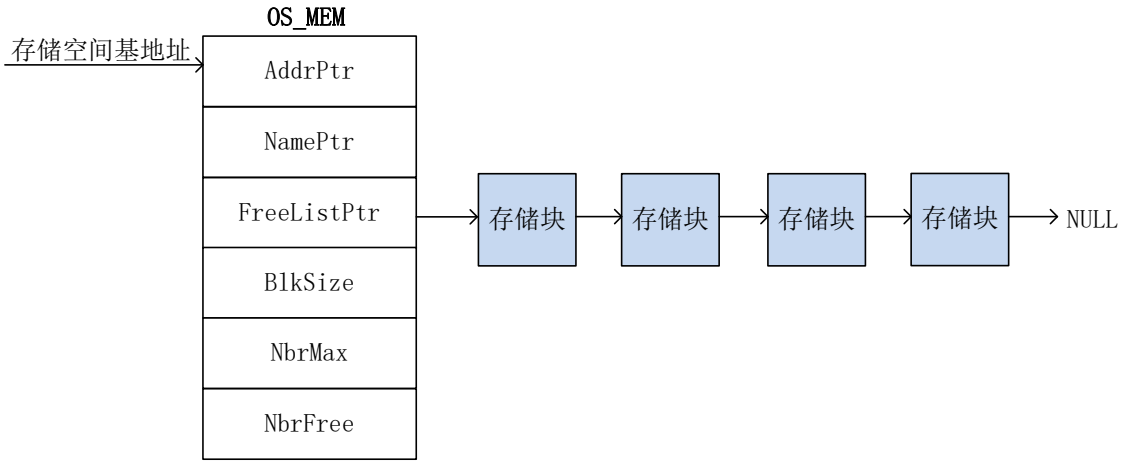


图 14.2.1 创建好的存储分区

14.3 存储块的使用

调用函数 OSMemCreate()创建好存储区以后我们就可以使用创建好的存储块了。

14.3.1 内存申请

使用函数 OSMemGet()来获取存储块，函数原型如下：

```
void    *OSMemGet (OS_MEM     *p_mem,
                   OS_ERR     *p_err)
```

函数 OSMemGet()用来从指定的存储区中获取存储块共应用使用。

p_mem： 要使用的存储区。

p_err： 返回的错误码。

返回值： 获取到的存储块地址。

OSMemGet()函数源码如下：

```
void    *OSMemGet (OS_MEM     *p_mem,
                   OS_ERR     *p_err)
{
    void       *p_blk;
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((void *)0);
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    if (p_mem == (OS_MEM *)0) {                                    (1)
        *p_err    = OS_ERR_MEM_INVALID_P_MEM;
        return ((void *)0);
```

```
    }

    CPU_CRITICAL_ENTER();
    if (p_mem->NbrFree == (OS_MEM_QTY)0) {                    (2)
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_MEM_NO_FREE_BLKS;
        return ((void *)0);
    }
    p_blk = p_mem->FreeListPtr;                               (3)
    p_mem->FreeListPtr = *(void **)p_blk;                     (4)
    p_mem->NbrFree--;                                         (5)
    CPU_CRITICAL_EXIT();
   *p_err = OS_ERR_NONE;
    return (p_blk);                                           (6)
}
```

(1)、判断存储区 p_mem 是否存在，不存在的话，就返货错误码：OS_ERR_MEM_INVALID_P_MEM。

(2)、判断存储区中是否还有空闲的存储块，没有的话就返回错误码：OS_ERR_MEM_NO_FREE_BLKS。

(3)、如果还有剩余存储块的话就取出空闲存储块链表中第一个存储块供用户程序使用。

(4)、在(3)中我们已经用掉了空闲存储块链表中的第一个存储块，那么空闲存储块链表头 FreeListPtr 就需要更新，指向下一个存储块。

(5)、因为应用程序申请了一个存储块，所以剩余存储块数量减一。

(6)、向应用程序返回申请到的存储块。

从上面的分析中我们可以看出 UCOSIII 自带的内存管理函数的局限性，每次申请内存的时候用户要先估计所申请的内存是否会超过存储区中存储块的大小。比如我们创建了一个有 10 个存储块，每个存储块大小为 100 字节的存储区 buffer。这时我们应用程序需要申请一个 10 字节的内存，那么就可以使用函数 OSMemGet()从存储区 buffer 中申请一个存储块。但是每个存储块有 100 个字节，但是应用程序只使用其中的 10 个字节，剩余的 90 个字节就浪费掉了，为了减少浪费我们可以创建一个每个存储块为 10 字节的存储区，这样就不会有内费了。

但是，问题又来了，假设我们在程序的其他地方需要申请一个 150 字节的内存，但是存储区 buffer 的每个存储块只有 100 字节，显然存储区 buffer 不能满足程序的需求。有读者就会问可不可以在存储区中连续申请两个 100 字节的存储块，这样就有 200 字节的内存供应用程序使用了？想法是好想法，但是通过阅读函数 OSMemGet()发现并没有提供这样的功能，OSMemGet()函数在申请内存的时候每次只取指定存储区的一个存储块！如果想申请 150 字节的内存就必须再新建一个每个存储块至少有 150 字节的存储区。

通过上面的分析可以看出 UCOSIII 的内存管理很粗糙，不灵活，并不能申请指定大小的内存块。使用过 ALIENTEK 的 STM32 开发板的用户就会知道 ALIENTEK 也实现了内存的动态使用，可以申请任意大小的内存空间，使用起来十分方便。

14.3.2 内存释放

上一小节讲解了内存的申请，这节就讲解一下内存释放，在 UCOSIII 中内存的释放可以使用函数 OSMemPut()来完成，函数原型如下：

```
void    OSMemPut (OS_MEM      *p_mem,
                  void        *p_blk,
                  OS_ERR      *p_err)
```

函数 OSMemPut()用于释放内存，将申请到的存储块还给指定的存储区。

p_mem： 指向存储区控制块，也就是要接收存储块的那个存储区。

p_blk： 指向存储块，要归还的存储块。

p_err： 错误码。

我们再来看一下 OSMemPut()函数的源码，源码如下：

```
void    OSMemPut (OS_MEM      *p_mem,
                  void        *p_blk,
                  OS_ERR      *p_err)
{
    CPU_SR_ALLOC();


#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    if (p_mem == (OS_MEM *)0) {                                    (1)
        *p_err    = OS_ERR_MEM_INVALID_P_MEM;
        return;
    }
    if (p_blk == (void *)0) {                                      (2)
        *p_err    = OS_ERR_MEM_INVALID_P_BLK;
        return;
    }
#endif

    CPU_CRITICAL_ENTER();
    if (p_mem->NbrFree >= p_mem->NbrMax) {                         (3)
        CPU_CRITICAL_EXIT();
       *p_err = OS_ERR_MEM_FULL;
        return;
    }
    *(void **)p_blk = p_mem->FreeListPtr;                          (4)
    p_mem->FreeListPtr = p_blk;                                    (5)
    p_mem->NbrFree++;                                              (6)
    CPU_CRITICAL_EXIT();
```

```
    *p_err = OS_ERR_NONE;
}
```

(1)、检查存储区是否存在，不存在的话就返回错误码：OS_ERR_MEM_INVALID_P_MEM。

(2)、检查存储块是否存在，不存在的话就返回错误码：OS_ERR_MEM_INVALID_P_BLK。

(3)、检查存储区空闲存储块数量是否大于存储区总存储块数量，要是大于的话那肯定是见鬼了，如果大于的话就返回错误码：OS_ERR_MEM_FULL。

(4)、将要归还的存储块添加到空闲存储块链表中，这里是添加到表头的位置，前面我们说了存储块中会保存下一个空闲存储块的地址，所以这里向存储块 p_blk 中写入下一个空闲存储块地址。

(5)、将归还的存储块添加到空闲存储链表中以后需要更新一下 FreeListPtr。

(6)、存储区中可用存储块数量加一。

14.4 存储管理实验

14.4.1 实验程序设计

例 14-1：设计一个程序，创建两个存储区，这两个存储区分别创建 STM32 的内部 RAM 和外部 SRAM 中，通过开发板上的按键来申请和释放内存。

答：分析上面程序设计要求，我们创建两个存储区 INTERNAL_MEM 和 EXTERNAL_MEM，INTERNAL_MEM 使用 STM32 内部 RAM，EXTERNAL_MEM 使用外部的 SRAM。按下 KEY_UP 键从 INTERNAL_MEM 中申请一块内存，按下 KEY1 将申请到的内存释放给 INTERNAL_MEM。按下 KEY2 键从 EXTERNAL_MEM 中申请一块内存，按下 KEY0 将内存释放给 EXTERNAL_MEM。实验完整工程见“例 14-1 UCOSIII 内存管理”。

首先定义存储区的存储空间，定义如下：

```
OS_MEM INTERNAL_MEM;                        //定义一个存储区
#define INTERNAL_MEM_NUM       5            //存储区中存储块数量
#define INTERNAL_MEMBLOCK_SIZE   100        //每个存储块大小由于一个指针变量占用 4
                                            //字节所以块的大小一定要为 4 的倍数而且
                                           //必须大于一个指针变量(4 字节)占用的空间,
                                            //否则的话存储块创建不成功

//存储区的内存池，使用内部 RAM
__align(4) CPU_INT08U Internal_RamMemp[INTERNAL_MEM_NUM]\
                                    [INTERNAL_MEMBLOCK_SIZE];

OS_MEM EXTERNAL_MEM;                        //定义一个存储区
#define EXTRENNAL_MEM_NUM        5          //存储区中存储块数量
#define EXTERNAL_MEMBLOCK_SIZE  100         //每个存储块大小由于一个指针变量占用 4
                                            //字节所以块的大小一定要为 4 的倍数而且
                                           //必须大于一个指针变量(4 字节)占用的空间,
                                            //否则的话存储块创建不成功

//存储区的内存池，使用外部 SRAM
__align(32) volatile CPU_INT08U External_RamMemp[EXTRENNAL_MEM_NUM]\
```

```
                [EXTERNAL_MEMBLOCK_SIZE]  __attribute__((at(0X68000000)));
```

上面我们定义了两个存储区：INTERNAL_MEM 和 EXTERNAL_MEM，另外还定义了 2 个这两个存储区的存储空间。

打开 main 函数，在 main 函数中创建了两个存储区，代码如下：

```
//主函数
int main(void)
{
    OS_ERR err;
    CPU_SR_ALLOC();

    delay_init(72);   //时钟初始化
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//中断分组配置
    uart_init(115200);    //串口初始化
    LED_Init();           //LED 初始化
    LCD_Init();           //LCD 初始化
    KEY_Init();           //按键初始化
    BEEP_Init();        //初始化蜂鸣器
    FSMC_SRAM_Init();   //初始化 SRAM
    ucos_load_main_ui();//加载主 UI1

    OSInit(&err);             //初始化 UCOSIII
    OS_CRITICAL_ENTER();  //进入临界区
    //创建一个存储分区
    OSMemCreate((OS_MEM*     )&INTERNAL_MEM,          //存储区控制块    (1)
                (CPU_CHAR*   )"Internal Mem",         //存储区名字
                (void*       )&Internal_RamMemp[0][0],//存储空间基地址
                (OS_MEM_QTY  )INTERNAL_MEM_NUM,       //存储块数量
                (OS_MEM_SIZE )INTERNAL_MEMBLOCK_SIZE,//存储块大小
                (OS_ERR*     )&err);                  //错误码
    //创建一个存储分区
    OSMemCreate((OS_MEM*     )&EXTERNAL_MEM,                          (2)
                (CPU_CHAR*   )"External Mem",
                (void*       )&External_RamMemp[0][0],
                (OS_MEM_QTY  )EXTRENNAL_MEM_NUM,
                (OS_MEM_SIZE )EXTERNAL_MEMBLOCK_SIZE,
                (OS_ERR*     )&err);
    //创建开始任务
    OSTaskCreate((OS_TCB*       )&StartTaskTCB,          //任务控制块
                 (CPU_CHAR*     )"start task",           //任务名字
                 (OS_TASK_PTR   )start_task,             //任务函数
                 (void*         )0,                      //传递给任务函数的参数
                 (OS_PRIO       )START_TASK_PRIO,        //任务优先级
                 (CPU_STK*      )&START_TASK_STK[0],     //任务堆栈基地址
```

```
                    (CPU_STK_SIZE )START_STK_SIZE/10,       //任务堆栈深度限位
                    (CPU_STK_SIZE )START_STK_SIZE,          //任务堆栈大小
                    (OS_MSG_QTY   )0,                       //任务内部消息队列能够接收
                                                            //的最大消息数目,为 0 时禁止
                                                            //接收消息
                    (OS_TICK      )0,                       //当使能时间片轮转时的时间
                                                            //片长度,为 0 时为默认长度,
                    (void*        )0,                       //用户补充的存储区
                    (OS_OPT       )OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR,
                    (OS_ERR *     )&err);                   //存放该函数错误时的返回值
    OS_CRITICAL_EXIT();     //退出临界区
    OSStart(&err);          //开启 UCOSIII
}
```

(1)、创建存储区 INTERNAL_MEM，存储区使用 STM32 内部 RAM，共有 5 个存储块，每个存储块大小为 100 字节。

(2)、创建存储区 EXTERNAL_MEM，存储区使用 STM32 外部 SRAM，共有 5 个存储块，每个存储块大小为 100 字节。

另外还有 2 个任务函数：main_task()和 memmanage_task()，任务函数代码如下：

```
//主任务的任务函数
void main_task(void *p_arg)
{
    u8 key,num;
    static u8 internal_memget_num;
    static u8 external_memget_num;
    CPU_INT08U *internal_buf;
    CPU_INT08U *external_buf;
    OS_ERR err;
    while(1)
    {
        key = KEY_Scan(0);              //扫描按键                          (1)
        switch(key)
        {
            case WKUP_PRES:             //按下 KEY_UP 键
                internal_buf=OSMemGet((OS_MEM*     )&INTERNAL_MEM, (2)
                                      (OS_ERR*     )&err);
                printf("internal_buf 内存申请之后的地址为:%#x\r\n",(u32)(internal_buf));
                if(err == OS_ERR_NONE)          //内存申请成功              (3)
                {
                    LCD_ShowString(30,180,200,16,16,"Memory Get success!    ");
                    internal_memget_num++;
                    POINT_COLOR = BLUE;
                    sprintf((char*)internal_buf,"INTERNAL_MEM Use %d times",\
                    internal_memget_num);
```

```
            LCD_ShowString(30,196,200,16,16,internal_buf);
            POINT_COLOR = RED;
        }
        if(err == OS_ERR_MEM_NO_FREE_BLKS)      //内存块不足
        {
            LCD_ShowString(30,180,200,16,16,"INTERNAL_MEM Empty!    ");
        }
        break;
    case KEY1_PRES:
        if(internal_buf != NULL)            //internal_buf 不为空就释放内存
        {
            OSMemPut((OS_MEM*    )&INTERNAL_MEM, //释放内存     (4)
                     (void*      )internal_buf,
                     (OS_ERR*    )&err);
            printf("internal_buf 内存释放之后的地址为:%#x\r\n",(u32)(internal_buf));
            LCD_ShowString(30,180,200,16,16,"Memory Put success!     ");
        }
        break;
    case KEY2_PRES:
        external_buf=OSMemGet((OS_MEM*    )&EXTERNAL_MEM, (5)
                              (OS_ERR*    )&err);
        printf("external_buf 内存申请之后的地址为:%#x\r\n",(u32)(external_buf));
        if(err == OS_ERR_NONE) //内存申请成功
        {
            LCD_ShowString(30,260,200,16,16,"Memory Get success!    ");
            external_memget_num++;
            POINT_COLOR = BLUE;
            sprintf((char*)external_buf,"EXTERNAL_MEM Use %d times",\
                    external_memget_num);
            LCD_ShowString(30,276,200,16,16,external_buf);
            POINT_COLOR = RED;
        }
        if(err == OS_ERR_MEM_NO_FREE_BLKS) //内存块不足
        {
            LCD_ShowString(30,260,200,16,16,"EXTERNAL_MEM Empty!    ");
        }
        break;
    case KEY0_PRES:
        if(external_buf != NULL)     //external_buf 不为空就释放内存
        {
            OSMemPut((OS_MEM*    )&EXTERNAL_MEM,     //释放内存(6)
                     (void*      )external_buf,
                     (OS_ERR*    )&err);
```

```
                printf("external_buf 内存释放之后的地址为:%#x\r\n",(u32)(external_buf));
                LCD_ShowString(30,260,200,16,16,"Memory Put success!      ");
            }
            break;
        }

        num++;
        if(num==50)
        {
            num=0;
            LED0 = ~LED0;
        }
        OSTimeDlyHMSM(0,0,0,10,OS_OPT_TIME_PERIODIC,&err);     //延时 10ms
    }
}

//内存管理任务
void memmanage_task(void *p_arg)
{
    OS_ERR err;
    LCD_ShowString(5,164,200,16,16,"Total:    Remain:");
    LCD_ShowString(5,244,200,16,16,"Total:    Remain:");
    while(1)
    {
        POINT_COLOR = BLUE;
        LCD_ShowxNum(53,164,INTERNAL_MEM.NbrMax,1,16,0);
        LCD_ShowxNum(125,164,INTERNAL_MEM.NbrFree,1,16,0);
        LCD_ShowxNum(53,244,EXTERNAL_MEM.NbrMax,1,16,0);
        LCD_ShowxNum(125,244,EXTERNAL_MEM.NbrFree,1,16,0);
        POINT_COLOR = RED;
        OSTimeDlyHMSM(0,0,0,100,OS_OPT_TIME_PERIODIC,&err);//延时 100ms
    }
}
```

(1)、调用函数 KEY_Scan()获取按键值。

(2)、按下 KEY_UP 按键的话调用函数 OSMemGet()从 INTERNAL_MEM 中申请一块内存，指针 internal_buf 指向申请到的内存。

(3)、内存申请成功以后打印出 internal_buf 的地址，申请到内存以后可以使用这段内存，在这段内存中写入字符串指定的字符串。

(4)、按下 KEY1 键释放内存，并且打印出释放内存后的 internal_buf 地址。

(5)、同(2)作用相同，只不过是从存储区 EXTERNAL_MEM 中申请内存。

(6)、同(3)作用相同。

memmanage_task()任务函数用于检测存储区 INTERNAL_MEM 和 EXTERNAL_MEM 的总存储块数和空闲存储块数量，并且显示到 LCD 上。

14.4.2 实验程序结果分析

代码编译完成下载到开发板中观察和分析实验现象，代码刚下进去后 LCD 显示如图 14.4.1 所示。

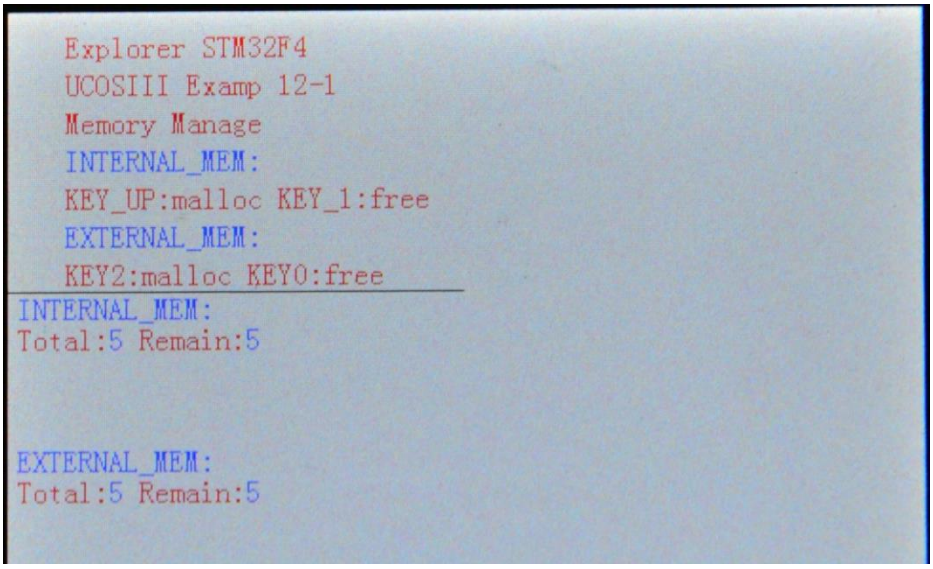


图 14.4.1 LCD 界面

从图 14.4.1 可以看出，此时 INTERNAL_MEM 和 EXTERNAL_MEM 两个存储区都有 5 个存储块，此时还剩余 5 个。按下 KEY_UP 和 KEY2 键即可申请内存，申请成功以后如图 14.4.3 所示。串口调试助手打印出 internal_buf 和 external_buf 申请到的地址，如图 14.4.2 所示。

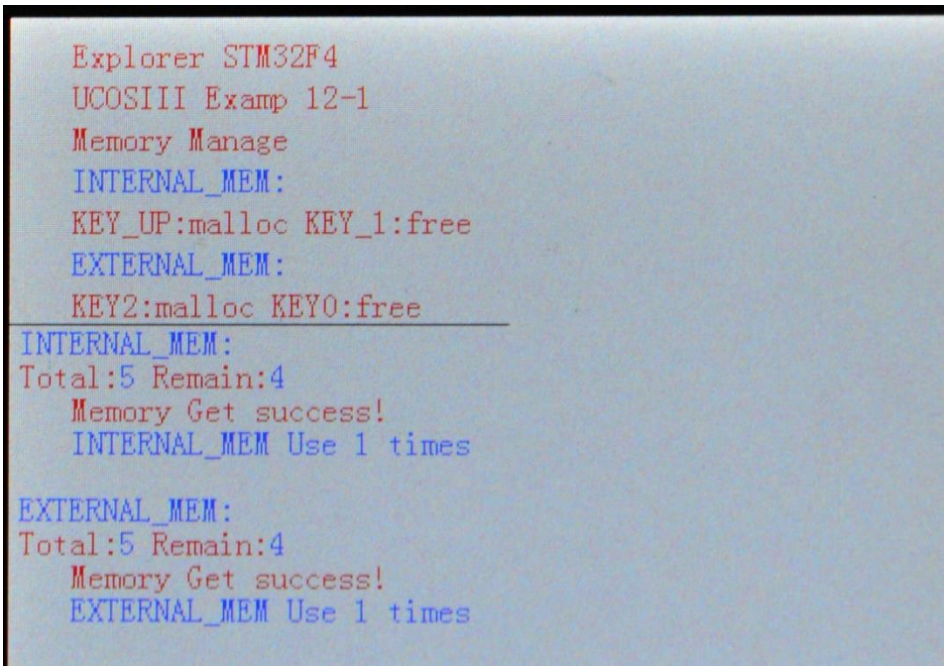


图 14.4.3 申请内存

从图 14.4.3 中可以看出，此时分别从 INTERNAL_MEM 和 EXTERNAL_MEM 这两个存储区中申请了一个存储块，此时都还剩下 4 个存储块可用。

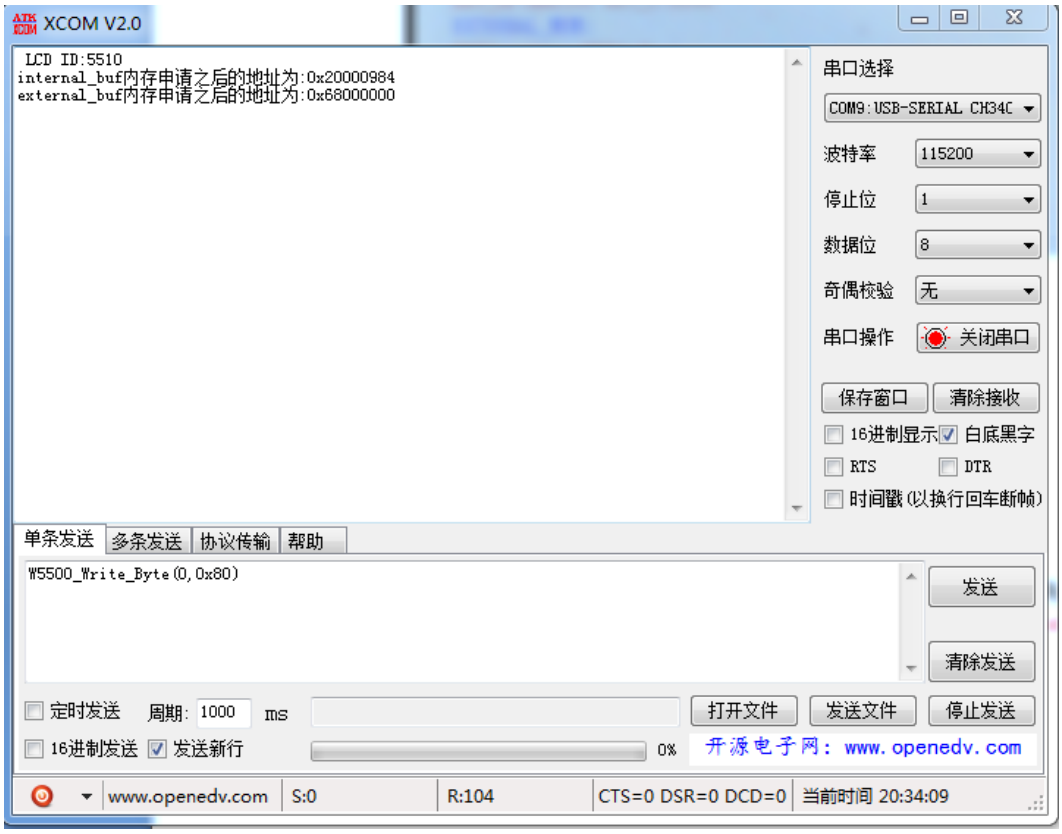


图 14.4.2 串口调助手输出信息

内存申请成功以后 internal_buf 的地址为 0X20000984，external_buf 的地址为 0X68000000。可以看出 internal_buf 是在 STM32 内部 RAM 中，external_buf 是在 STM32 外扩的 SRAM 中。按下 KEY1 和 KEY0 即可释放内存。如图 14.4.4 所示。

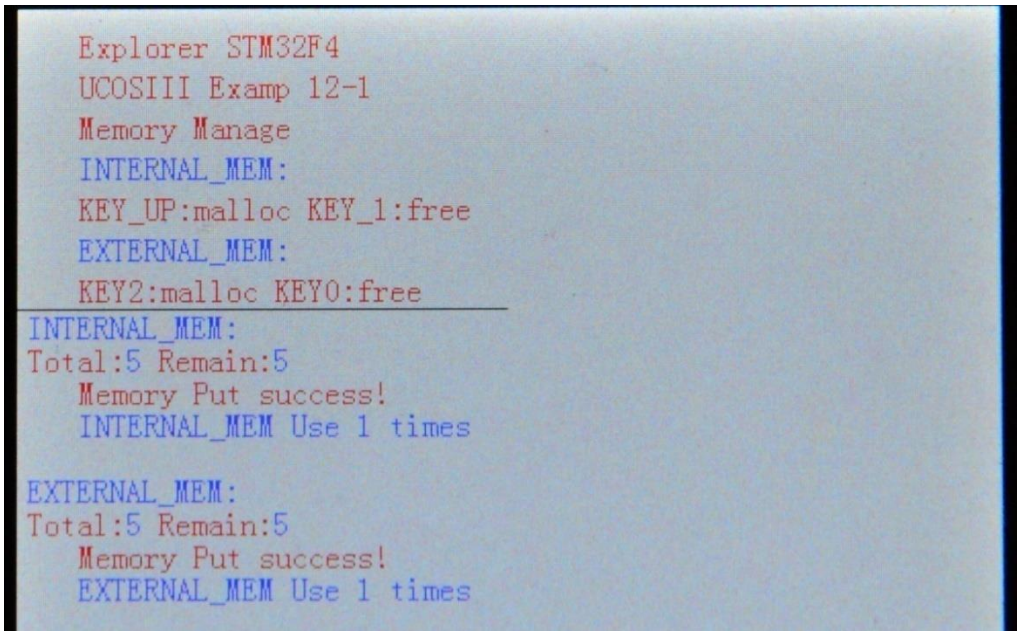


图 14.4.4 释放内存

从图 14.4.4 可以看出当释放内存以后，INTERNAL_MEM 和 EXTERNAL_MEM 这两个存储区的空闲存储块又恢复到了 5 个。

如果我们一直申请内存不释放的话，当申请 5 次以后存储区就会变空，导致再也不能申请到内存，此时 LCD 如图 14.4.5 所示。注意这里是一直给 internal_buf 和 external_buf 申请内存，会导致内存泄漏！

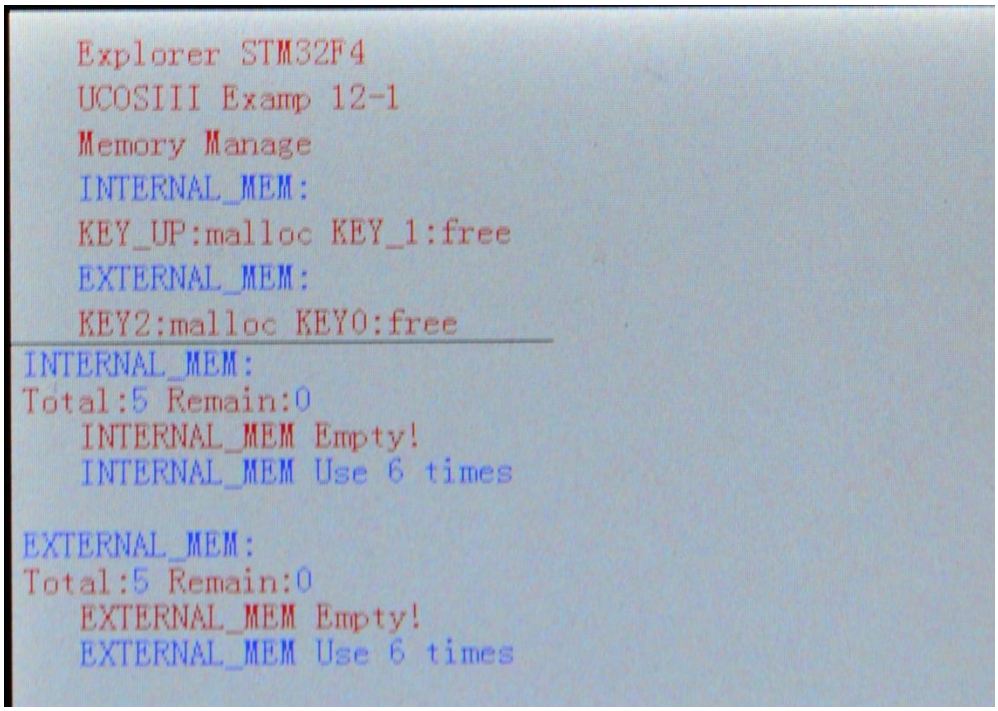


图 14.4.5 存储区空