

目录

第一章	操作系统的概述	1
1.1	操作系统的介绍	1
1.2	操作系统的作用	1
1.3	常见的操作系统	1
第二章	μ C/OS 操作系统的概述	3
2.1	实时操作系统的介绍	3
2.1.1	前/后台系统	3
2.1.2	内核结构	3
2.1.3	可剥夺型内核	3
2.2	μ C/OS 操作系统的介绍	4
2.3	μ C/OS 操作系统的调度原则	4
2.4	μ C/OS 操作系统的程序架构	4
2.5	μ C/OS 操作系统的任务概述	4
2.5.1	任务介绍	4
2.5.2	任务切换	5
2.5.3	任务中断	5
2.5.4	任务状态	5
第三章	μ C/OS- II 工程的创建	8
3.1	μ C/OS- II 源码的获取	8
3.2	μ C/OS- II 源码的介绍	10
3.2	μ C/OS- II 的移植	11
3.2.1	裸机工程的准备	11
3.2.2	复制相关 μ C/OS- II 的移植文件	11
3.2.3	添加 μ C/OS- II 源码和移植 cpu 相关源码	12
3.2.4	添加 μ C/OS- II 源码和移植 cpu 相关头文件路径	17
3.2.5	编译纠错	17
3.2.6	修改 cpu 相关的汇编接口函数	23
3.2.7	修改 main 函数	25
第四章	μ C/OS- II 任务管理	27
4.1	μ C/OS- II 任务的创建	27
4.2	μ C/OS- II 任务相关函数	27
4.2.1	OSTaskCreate()	27
4.2.2	OSTaskCreateExt()	28

4.2.3 OSTaskDel()	30
4.2.4 OSTaskDelReq()	30
4.2.5 OSTaskSuspend()	32
4.2.6 OSTaskResume()	32
4.2.7 OSTaskChangePrio()	33
4.2.8 OSTaskStkChk()	33
4.2.9 OSTaskQuery()	34
4.2.10 OSTaskNameGet()	35
4.2.11 OSTaskNameSet()	36
4.2.12 OSStart()	36
4.3 μ C/OS- II 延时函数	37
4.3.1 OSTimeDly()	37
4.3.2 OSTimeDlyHMSM()	37
4.3.3 OSTimeDlyResume()	38
4.4 μ C/OS- II 任务管理的编程思想	38
4.5 μ C/OS- II 任务管理的编程示例	39
第五章 μ C/OS- II 信号量管理	41
5.1 μ C/OS- II 信号量简介	41
5.2 μ C/OS- II 信号量相关函数	41
5.2.1 OSSemCreate()	41
5.2.2 OSSemAccept()	41
5.2.3 OSSemPend()	42
5.2.4 OSSemPost()	43
5.2.5 OSSemDel()	44
5.2.6 OSSemQuery()	45
5.2.7 OSSemPendAbort()	46
5.2.8 OSSemSet()	46
5.3 μ C/OS- II 信号量管理编程思想	47
5.4 μ C/OS- II 信号量管理编程示例	47
第六章 μ C/OS- II 互斥信号量管理	50
6.1 μ C/OS- II 互斥信号量简介	50
6.1.1 互斥型信号量的理解	50
6.1.2 互斥信号量的组成	50
6.1.3 优先级反转的问题	50
6.2 μ C/OS- II 互斥信号量相关函数	51

6.2.1 OSMutexCreate ()	51
6.2.2 OSMutexAccept().....	52
6.2.3 OSMutexPend ().....	53
6.2.4 OSMutexPost ().....	54
6.2.5 OSMutexDel ()	54
6.2.6 OSMutexQuery ().....	55
6.3 μ C/OS- II 互斥信号量编程思想.....	56
6.4 μ C/OS- II 互斥信号量编程示例.....	57
第七章 μ C/OS- II 消息邮箱管理.....	60
7.1 μ C/OS- II 消息邮箱简介	60
7.2 μ C/OS- II 消息邮箱相关函数.....	60
7.2.1 OSMboxCreate()	60
7.2.2 OSMboxAccept()	61
7.2.3 OSMboxPend()	61
7.2.4 OSMboxPost()	62
7.2.5 OSMboxDel()	63
7.2.6 OSMboxQuery()	64
7.2.7 OSMboxPendAbort ()	65
7.2.8 OSMboxPostOpt ()	66
7.3 μ C/OS- II 消息邮箱编程思想.....	67
7.4 μ C/OS- II 消息邮箱编程示例.....	67
第八章 μ C/OS- II 消息队列管理.....	69
8.1 μ C/OS- II 消息队列简介.....	69
8.2 μ C/OS- II 消息队列相关函数.....	69
8.2.1 OSQCreate ()	69
8.2.2 OSQAccept ()	69
8.2.3 OSQPend ()	70
8.2.4 OSQPost ()	71
8.2.5 OSQDel()	72
8.2.6 OSQQuery ()	73
8.2.7 OSQFlush()	74
8.2.8 OSQPostFront()	75
8.2.9 OSQPendAbort ()	75
8.2.10 OSMboxPostOpt ()	76
8.3 μ C/OS- II 消息队列编程思想.....	77

8.4 $\mu\text{C}/\text{OS- II}$ 消息队列编程示例	78
第九章 $\mu\text{C}/\text{OS- II}$ 事件标志组管理	81
9.1 $\mu\text{C}/\text{OS- II}$ 事件标志组简介	81
9.2 $\mu\text{C}/\text{OS- II}$ 事件标志组相关函数	81
9.2.1 OSFlagCreate()	81
9.2.2 OSFlagAccept ()	82
9.2.3 OSFlagPend ()	83
9.2.4 OSFlagPost ()	84
9.2.5 OSFlagDel ()	85
9.2.6 OSFlagQuery ()	86
9.2.7 OSFlagNameGet ()	86
9.2.8 OSFlagNameSet ()	87
9.2.9 OSFlagPendGetFlagsRdy ()	88
9.3 $\mu\text{C}/\text{OS- II}$ 事件标志组编程思想	88
9.4 $\mu\text{C}/\text{OS- II}$ 事件标志组编程示例	89
第十章 $\mu\text{C}/\text{OS- II}$ 软件定时器管理	91
10.1 $\mu\text{C}/\text{OS- II}$ 软件定时器简介	91
10.1.1 软件定时器的介绍	91
10.1.2 回调函数的介绍	91
10.1.3 单次定时器	91
10.1.4 周期模式--无初始延迟	92
10.1.5 周期模式--有初始延迟	92
10.2 $\mu\text{C}/\text{OS- II}$ 软件定时器相关函数	92
10.2.1 OSTmrCreate()	92
10.2.2 OSTmrStart()	94
10.2.3 OSTmrStop()	95
10.2.4 OSTmrDel()	96
10.2.5 OSTmrStateGet()	97
10.2.6 OSTmrNameGet ()	97
10.2.7 OSTmrRemainGet ()	98
10.2.7 OSTmrSignal ()	99
10.3 $\mu\text{C}/\text{OS- II}$ 软件定时器编程思想	100
10.4 $\mu\text{C}/\text{OS- II}$ 软件定时器编程示例	100
第十一章 $\mu\text{C}/\text{OS- II}$ 内存管理	103
11.1 $\mu\text{C}/\text{OS- II}$ 内存管理的介绍	103

11.1.1 内存控制块	103
11.2 μ C/OS- II 内存管理相关函数	104
11.2.1 OSMemCreate ()	104
11.2.2 OSMemGet ()	104
11.2.3 OSMemPut ()	105
11.2.4 OSMemQuery ()	106
11.2.5 OSMemNameGet ()	107
11.2.6 OSMemNameSet ()	107
11.3 μ C/OS- II 内存管理编程思想	108
11.4 μ C/OS- II 内存管理编程示例	108
第十二章 μ C/OS- II 其他功能函数	111
12.1 μ C/OS-II 其他功能函数	111
12.1.1 OSStatInit ()	111
12.1.2 OSIntEnter ()	111
12.1.3 OSIntExit ()	112
12.1.4 OS_ENTER_CRITICAL () , OS_EXIT_CRITICAL ()	112
12.1.5 OSSchedLock ()	113
12.1.6 OSSchedUnlock ()	113
12.1.7 OSVersion ()	114
12.1.8 OSTimeGet ()	114

第一章 操作系统的概述

1.1 操作系统的介绍

操作系统（Operating System，简称 OS）是管理和控制计算机硬件与软件资源的计算机程序，是直接运行在“裸机”上的最基本的系统软件，任何其他软件都必须在操作系统的支持下才能运行。

操作系统是用户和计算机的接口，同时也是计算机硬件和其他软件的接口。操作系统的功能包括管理计算机系统的硬件、软件及数据资源，控制程序运行，改善人机界面，为其它应用软件提供支持，让计算机系统所有资源最大限度地发挥作用，提供各种形式的用户界面，使用户有一个好的工作环境，为其它软件的开发提供必要的服务和相应的接口等。实际上，用户是不用接触操作系统的，操作系统管理着计算机硬件资源，同时按照应用程序的资源请求，分配资源，如：划分 CPU 时间，内存空间的开辟，调用打印机等。

操作系统是连接硬件和软件的桥梁。

WIN7 操作系统：联想 华硕 ---硬件是不一样的，但是他们可以同时使用 WIN7、WIN10 操作系统。

1.2 操作系统的作用

1. 我们为什么要用操作系统？电脑？作用大，才会使用它作用：

- 1）方便。一台电脑，如果没有配置操作系统，那么它肯定的非常难使用。
- 2）有效。CPU（高速）跟外部设备（相对来说低速），如果没有操作系统进行管理，CPU 跟外部设备会经常处于空闲状态。工作效率降低。
- 3）可以提供软件运行的环境，WIN7 --- 装 QQ 微信---依赖于操作系统。操作系统位于应用软件和硬件之间。应用软件不能脱离操作系统而独立存在的。

2. 操作系框架，如图 1.2.1 所示。

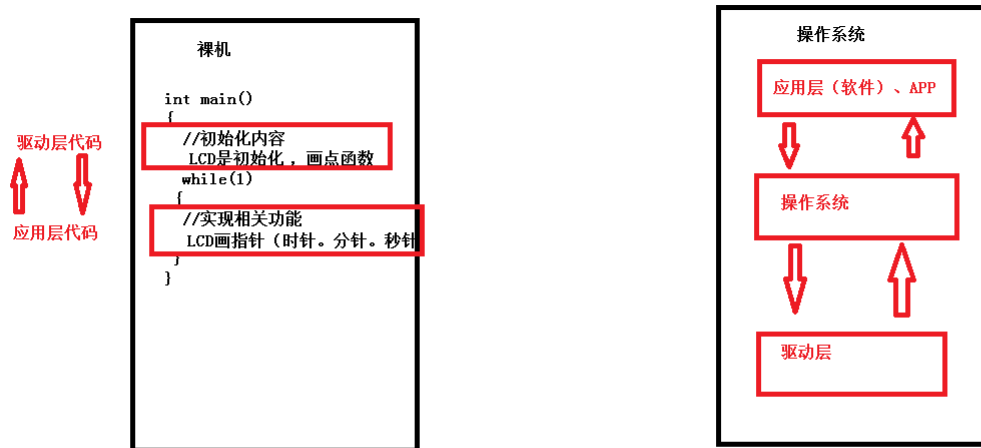


图 1.2.1

1.3 常见的操作系统

1. 常见的操作系统

win7 win8 win10 -- 电脑

安卓 IOS 赛班 --- 手机

Linux Vxworks FreeRTOS μ C/OS- -- MCU

3. 常见操作系统的分类

(1) 实时操作系统

不是给每个任务一个固定时间，任务时间长度可以自由决定，它是根据任务的优先级去执行相应的任务。

什么是任务？实现一个功能或多个功能。听音乐、看电影、上 QQ

μC/OS 、 FreeRTOS

(2) 分时操作系统

给每个任务一个固定的时间长度，当这个任务的时间到达后，这时会切换去执行下一个任务。时间不到达不进行任务的切换。

Windows --95-- 98 ,Linux ---内核 2.6 版本之前。

(3) 半实时操作系统

部分任务使用实时操作系统，部分任务使用分时操作系统。

win7-- win8 --win --10 ,Linux ---内核 2.6 版本之后。

第二章 $\mu\text{C}/\text{OS}$ 操作系统的概述

2.1 实时操作系统的介绍

2.1.1 前/后台系统

在早期嵌入式开发中，是没有操作系统这一概念的，都是直接裸机编程，比如用 51 单片机基本就没有操作系统的概念。通常把程序分为两部分：前台系统和后台系统，如图 2.1.1 所示。应用程序是一个死循环，然后在循环中调用相应的函数来实现相应的功能，这部分内容可以看成后台系统。中断服务函数处理异常事件，这部分内容就可以看前台系统。

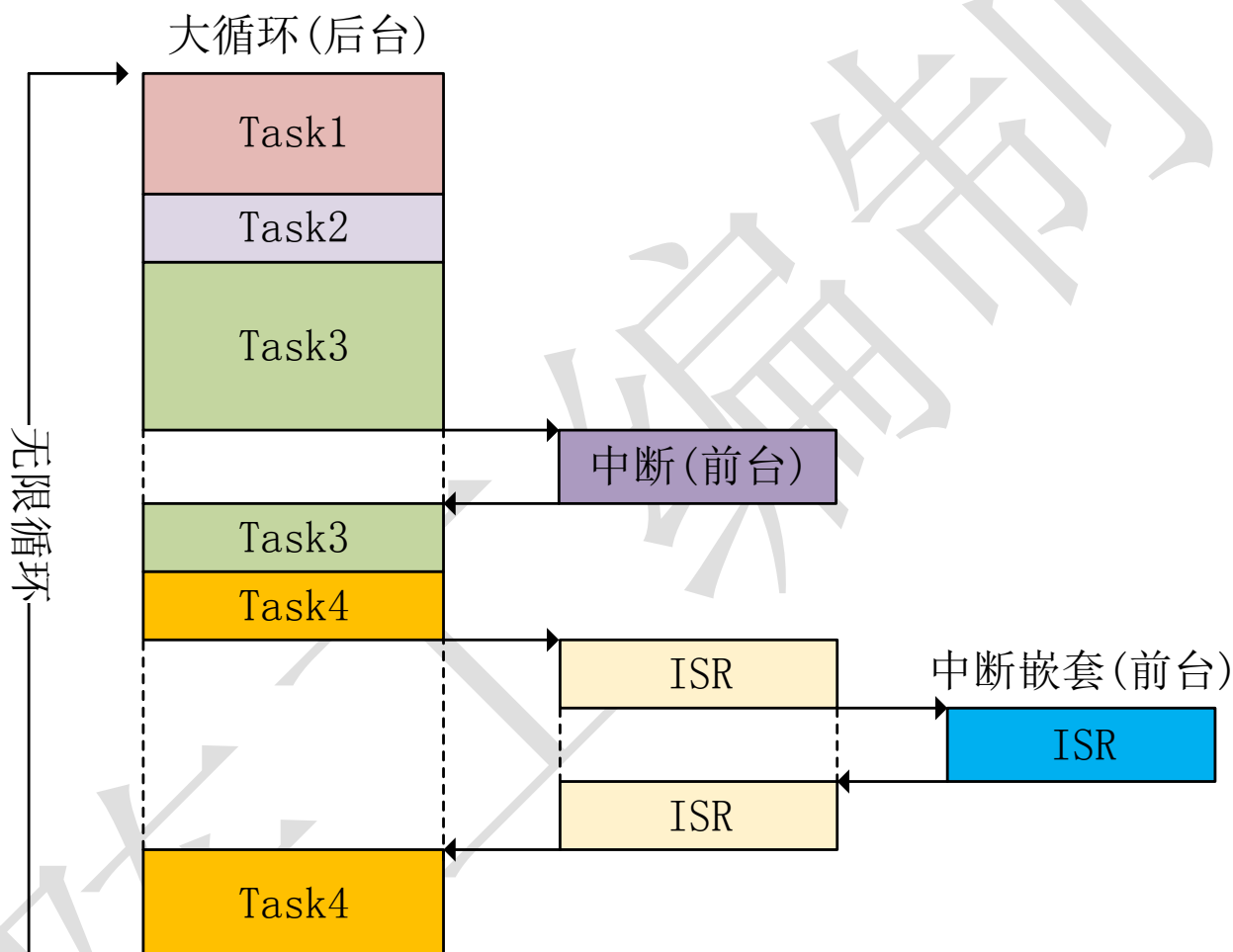


图 2.1.1

2.1.2 内核结构

实时操作系统的内核负责管理所有的任务，内核决定了运行哪个任务，何时停止当前任务切换到其他任务，这个是内核的多任务管理能力。多任务管理给人的感觉就好像芯片有多个 CPU，多任务管理实现了 CPU 资源的最大化利用，多任务管理有助于实现程序的模块化开发，能够实现复杂的实时应用。

2.1.3 可剥夺型内核

实时操作系统的内核负责管理所有的任务，内核决定了运行哪个任务，何时停止当前任务切换到其他任务，这个是内核的多任务管理能力。多任务管理给人的感觉就好像芯片有多个 CPU，多任务管理实现了 CPU 资源的最大化利用，多任务管理有助于实现程序的模块化开发，能够实现复杂的实时应用。

可剥夺内核顾名思义就是可以剥夺其他任务的 CPU 使用权，它总是运行就绪任务中的优先级最高的那个任务。 $\mu\text{C}/\text{OS}$ 的内核是可剥夺型的。

2.2 μ C/OS 操作系统的介绍

μ C/OS 是 Micrium 公司出品的实时操作系统, μ C/OS 目前有两个版本: μ C/OS-II 和 μ C/OS-III。 μ C/OS 是一种基于优先级的抢占式多任务实时操作系统, 包含了实时内核、任务管理、时间管理、任务间通信同步(信号量, 邮箱, 消息 队列)和内存管理等功能。它可以使各个任务独立工作, 互不干涉, 很容易实现准时而且无误执行, 使实时应用程序的设计和扩展变得容易, 使应用程序的设计过程大为减化。 μ C/OS 是一个完整的、可移植、可固化、可裁剪的抢占式实时多任务内核。 μ C/OS 绝大部分的代码是用 ANSI 的 C 语言编写的, 包含一小部分汇编代码, 使之可供不同架构的微处理器使用。

2.3 μ C/OS 操作系统的调度原则

调度原则: μ C/OS 任务与任务之间如何进行切换。

1. 实时操作系统的调度原则

实时操作系统每个任务都有它自己的**任务优先级**, 在 μ C/OS 中用一个编号来代替任务的优先级, 每个任务的优先级都是唯一。编号值越小, 任务的优先级就越高。当发生任务调度时, μ C/OS 操作系统会从所有准备好的任务中选取任务优先级最高的任务来执行。如有任务 1 (优先级为 5), 任务 2 (优先级为 10), 任务 3 (优先级为 15), 都准备好时, 发生了任务调度, 这里 μ C/OS 操作系统就会执行任务 1。

2. 分时操作系统的调度原则

分时操作系统给每个任务分配一个固定时间。所以分时操作系统的调度原则就是: 当正在执行的任务时间到达之后, 会切换到下一个任务。

2.4 μ C/OS 操作系统的程序架构

1. 裸机程序结构

裸机程序只有一个 main 函数, 在 main 函数中有一个死循环 while(1); 其他地方是没有 while(1)的。---裸机程序只会执行一个 while(1)里面的内容, 而不会在多个 while(1)中切换执行。

2. 操作系统程序结构

操作系统中, 也是只有一个 main 函数。在 main 函数中必须至少包含一个任务。在每个任务中都有一个 while(1)死循环。并且可以在多个任务中的 while(1)中切换执行。操作系统切换任务需要高优先级的任务释放 CPU 的使用权。低优先级的任务才能够执行。释放 100 个时钟节拍(心跳节拍)--究竟是多长时间? 一个心跳节拍时间是由我们程序员来自己设置的。可以设置一个心跳节拍为 1ms 或者 5ms

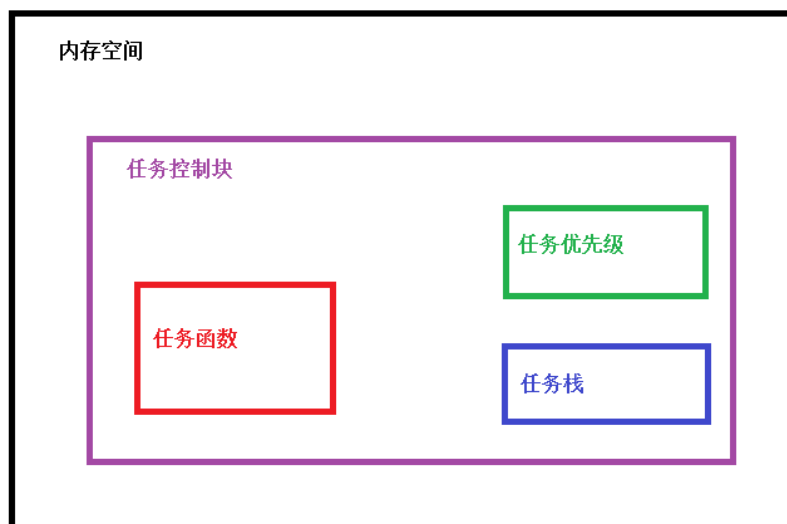
2.5 μ C/OS 操作系统的任务概述

2.5.1 任务介绍

μ C/OS 任务的组成: 即一个任务包含哪些内容。

任务由: **任务控制块**、**任务函数**、**任务优先级**、**任务堆栈** 组成。

1. 任务控制块: 任务向内存申请一段空间, 这段空间是用来执行这个任务用的, 任务控制块包含 任务函数、任务优先级和任务堆栈。



2. 任务函数：

本质是一个函数，这个函数我们叫做一个任务。函数本身是具有地址的。任务函数就是任务的入口地址。任务的功能是在函数中编写的。任务的名字一定要符合 C 语言的命名规则。

3. 任务优先级：

任务的唯一标识。每个任务都有它自己的优先级。用一种数值编号来表示（如 1，2，3）。数值编号越小，优先级越高。

4. 任务堆栈：

任务堆栈是用来存放任务信息的。如：当任务切换时，需要保存任务的状态（变量）。任务堆栈是程序员自己向内存申请的的一段内存空间。任务栈通过数组的形式来申请内存空间。如 `unsigned char arr[128];`

2.5.2 任务切换

任务切换： μ C/OS 在什么情况下会执行任务的切换？

1. 心跳节拍时间到达的时候会进行任务之间的切换。释放 100 个时钟节拍（心跳节拍）--究竟是多长时间？一个心跳节拍时间是由我们程序员来自己设置的。可以设置一个心跳节拍为 1ms 或者 5ms。在 μ C/OS 中用 Tick 来表示心跳节拍。设置的时候根据系统滴答定时器。

2. 当调用任务的调度函数时，也会进行任务之间的切换。调度函数：在 μ C/OS 中有定义：`OS_Sched()`。这个函数不需要我们程序员去调用。由 μ C/OS 操作系统自行调度的。

比如：我正在看电影，同时在上 QQ。在 QQ 的聊天界面输入信息过程中，电影是不是也在正常的播放。任务就在播放电影界面和 QQ 聊天界面来回切换。

2.5.3 任务中断

什么叫任务的中断？简单的说就是 μ C/OS 正在执行任务过程中，发生了中断。这时候中断功能还是跟裸机的中断功能一样吗？这是一样的。编写中断的功能规则也是一样的。但是，在编写中断的服务函数时，需要注意一点：

需要在编写中断服务函数的过程中添加两个函数功能：

1 个是告诉 μ C/OS 系统当前进入了中断功能，需要执行中断服务函数的内容—`OSIntEnter();`。

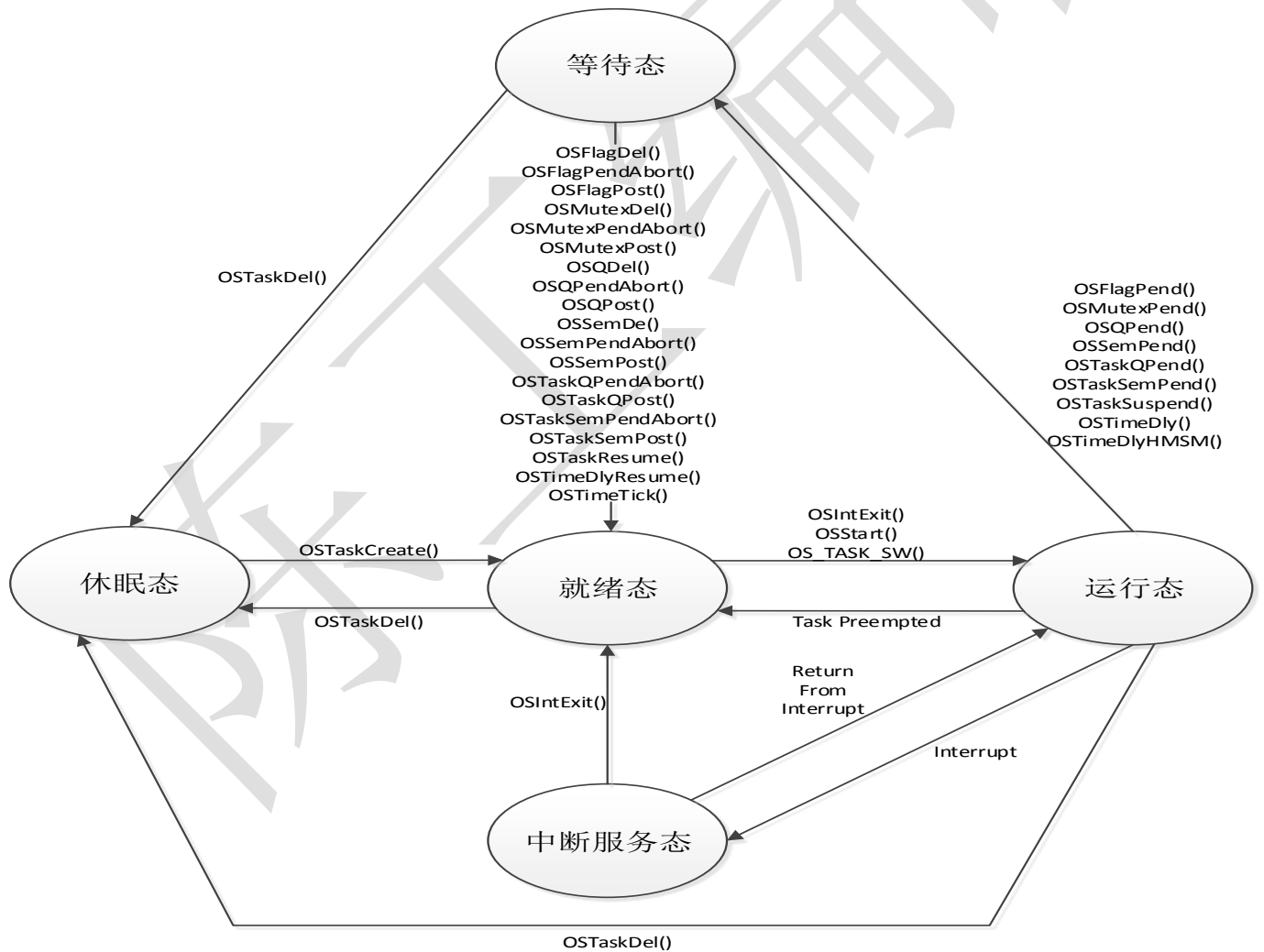
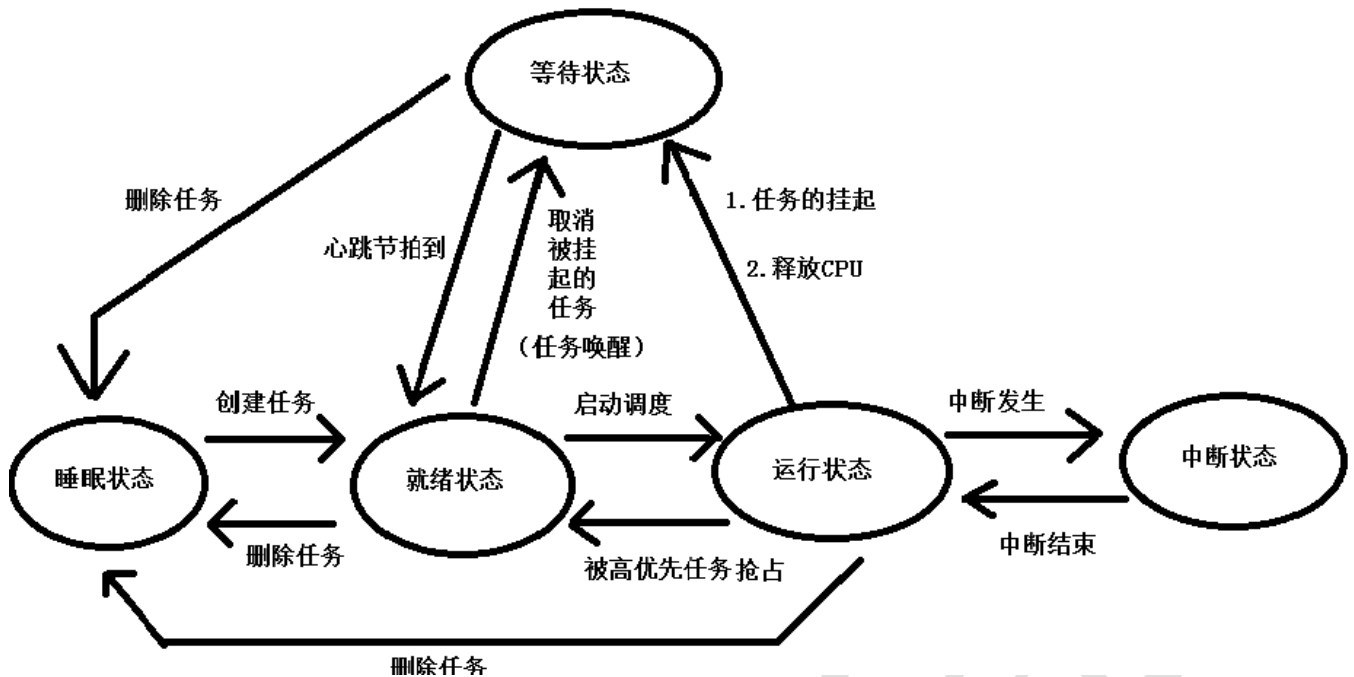
2. 在执行完成中断服务函数内容之后，需要告诉 μ C/OS 系统，当前已经执行完成中断服务函数了—`OSIntExit();`。

格式：

```
void 中断服务函数名(void)
{
    OSIntEnter();
    // 中断服务函数内容
    OSIntExit();
}
```

2.5.4 任务状态

在 μ C/OS-II 中，任务的状态有 5 种：睡眠（休眠）态、就绪态、运行态、等待态及中断服务态。



1. 休眠态

该状态下的任务指的是还没交给 $\mu\text{C}/\text{OS}$ 来管理，且驻留在程序空间（ROM 或 RAM）。把任务交给 $\mu\text{C}/\text{OS}$ 管理的话需要调用创建任务函数，`OSTaskCreate()` 或 `OSTaskCreateExt()`，后者是扩展版本。这些调用只是告诉 $\mu\text{C}/\text{OS}$ ，任务的起始地址在哪，任务建立时，用户给任务赋予的优先级是多少，任务要使用多少堆栈空间等。所以总结来

说，没有被创建前或者被销毁的任务都属于睡眠态。

2. 就绪态

当任务建立起来之后，该任务就会进入就绪态， $\mu\text{C}/\text{OS}$ 会进行任务调用，让高优先级的任务运行，即高优先级的任务能够获得 CPU 的使用权得到运行。`OSStart()` 可以启动多任务，它只能在启动时调用一次，该函数运行用户初始化代码中已经建立的、进入就绪态的优先级最高的任务。

3. 运行态

任何时候只能有一个任务处于运行态，即获得了 CPU 的使用权，该任务可以通过调用 `OSTaskDel()` 将自身返回到睡眠态，或者将另一个任务进入睡眠态。

4. 等待态

等待态顾名思义就是处于等待，等待事件的发生、时间的到来等。当处于运行态的任务调用 `OSTimeDly()` 或 `OSTimeDlyHMSM` 将自身延迟一段时间，该任务就会进入到等待态，或者调用 `OSSemPend()`、`OSQPend()` 等等待事件函数发生阻塞时也会使得任务进入等待态。而此时， $\mu\text{C}/\text{OS}$ 会执行任务切换，选择优先级最高并且处于等待状态的任务运行。而如果等待的事件发生了或者等待超时，被挂起的任务就会进入就绪态。

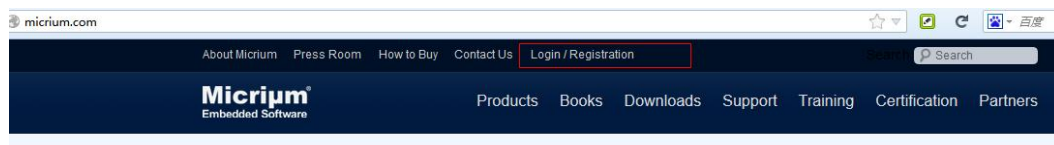
5. 中断态

中断服务态是正在运行的任务被中断了而进入的状态，响应中断时，正在执行的任务被挂起，中断服务子程序控制了 CPU 的使用权。中断服务子程序可能会报告一个或多个事件的发生，而使一个或多个任务进入就绪态。从中断服务子程序返回之前， $\mu\text{C}/\text{OS}$ 要判定，被中断的任务是否还是就绪任务态中优先级最高的。如果中断服务子程序使得另一个优先级更高的任务进入了就绪态，则新进入就绪态的这个优先级更高的任务得以运行，否则，原来被中断了的任务将继续运行。

第三章 μ C/OS- II 工程的创建

3.1 μ C/OS- II 源码的获取

1. 登陆 μ C/OS-官方网站 <http://micrium.com/>，如下所示：



2. 点击网站上方的 Login/Registration，进行登陆或注册一个新帐号。如果没有帐号要先注册一个，注册过程不再描述。我已经有帐号，直接登陆。
3. 点击后转入登陆界面，网址是 <http://micrium.com/wp-login.php>，直接输入这个进行登陆也可以，界面如下：



4. 在这个界面中输入用户名和密码，点击 Log In,即可以进入，登陆后可以下载网站上的免费资源。



Welcome to Micrium

Logged In

You have successfully logged in to the Micrium web site.

You now have access to any file in the [Download Center](#) and you can post messages in the [Micrium Discussion Forums](#).

Happy browsing!

5. 点击上面的“Downloads Center”，进入下载页面。

Browse by MCU Manufacturer



6. 点击” STMicroelectronics ”,进入 ST 公司芯片的 $\mu\text{C}/\text{OS}$ -工程下载页面。

STMicroelectronics STM32F4xx	$\mu\text{C}/\text{OS-II}$ $\mu\text{C}/\text{OS-II V2.92.11}$ $\mu\text{C}/\text{Probe V3.5.15.400}$ STM32CubeF4 Library V1.5.0	STM3240G-Eval	Atollic TrueSTUDIO V5.x IAR (EWARM) V7.x Keil MDK V5.x STM32CubeFx	2015/04/27
------------------------------	---	---------------	---	------------

Download “STM3240G-EVAL_OS2”

Download

File Type	Project
Processor	STMicroelectronics STM32F4xx
Micrium Product	$\mu\text{C}/\text{OS-II}$ $\mu\text{C}/\text{OS-II V2.92.11}$ $\mu\text{C}/\text{Probe V3.5.15.400}$ STM32CubeF4 Library V1.5.0
Version	2.92.11
Updated	April 27, 2015

7. 点击上面的下载图标后，可以下载到 $\mu\text{C}/\text{OS-II}$ 工程代码，名字是 Micrium_STM3240G-EVAL_OS2.zip，双击解压。

Micrium_STM3240G-EVAL_OS2.zip	2018/12/18 9:15	WinRAR ZIP 压缩...	1,518 KB
-------------------------------	-----------------	------------------	----------

8. 解压后得到一个文件夹 Micrium，里面有我们需要的全部文件，目录结构如下：

Examples	2015/4/27 14:00
Software	2015/4/27 14:00
README_STM3240G-EVAL_OS2.pdf	2015/4/27 14:11

打开\Micrium\Software

uC-CPU	2015/4/27 14:00	文件夹
uC-LIB	2015/4/27 14:00	文件夹
uCOS-II	2015/4/27 14:00	文件夹
uC-Serial	2015/4/27 14:00	文件夹

μC -CPU：这是和 CPU 紧密相关的文件。

μC-LCD: 这是和 LCD 相关的文件, 我们不需要使用。





μC-LIB: Micrium 公司提供的官方库文件, 如字符串操作、内存操作等函数接口。

μCOS-II: 这是关键目录文件, 我们接下来要详细分析的文件, 跟移植、使用密切相关的。

uC-Serial: 存放 μC/OS-II 基本配置文件文件。

3.2 μC/OS-II 源码的介绍

1. 打开\Micrium\Software\uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView

	os_cpu	2008/5/30 5:51	C Header File	6 KB
	os_cpu_a.asm	2008/8/22 8:49	ASM 文件	12 KB
	os_cpu_c	2008/5/30 5:51	C Source File	15 KB
	os_dbg	2008/5/30 5:51	C Source File	12 KB

os_cpu.h: 定义数据类型、处理器相关代码、函数声明。

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;      /* Unsigned  8 bit quantity */
typedef signed   char  INT8S;      /* Signed    8 bit quantity */
typedef unsigned short INT16U;     /* Unsigned 16 bit quantity */
typedef signed   short INT16S;     /* Signed   16 bit quantity */
typedef unsigned int   INT32U;     /* Unsigned 32 bit quantity */
typedef signed   int   INT32S;     /* Signed   32 bit quantity */
typedef float         FP32;        /* Single precision floating point */
typedef double        FP64;        /* Double precision floating point */

.....

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













os_cpu_a.asm: 与处理器相关 汇编代码, 主要是与任务切换相关。

os_cpu_c.c: 定义用户钩子函数, 提供扩充软件功能的接口。

os_dbg.c : 内核调试相关数据和相关函数。

以上文件不需要我们用户修改, 可以直接使用。

2. 打开 \Micrium\Software\μC/OS-II\Source

	os.h	2014/4/17 16:25	C++ Header file
	os_core.c	2014/12/11 16:59	C Source file
	os_flag.c	2014/4/17 16:25	C Source file
	os_mbox.c	2014/4/17 16:25	C Source file
	os_mem.c	2014/4/17 16:25	C Source file
	os_mutex.c	2014/4/17 16:25	C Source file
	os_q.c	2014/4/17 16:25	C Source file
	os_sem.c	2014/4/17 16:25	C Source file
	os_task.c	2014/4/17 16:25	C Source file
	os_time.c	2014/4/17 16:25	C Source file
	os_tmr.c	2014/4/17 16:25	C Source file
	ucos_ii.h	2014/4/17 16:25	C++ Header file

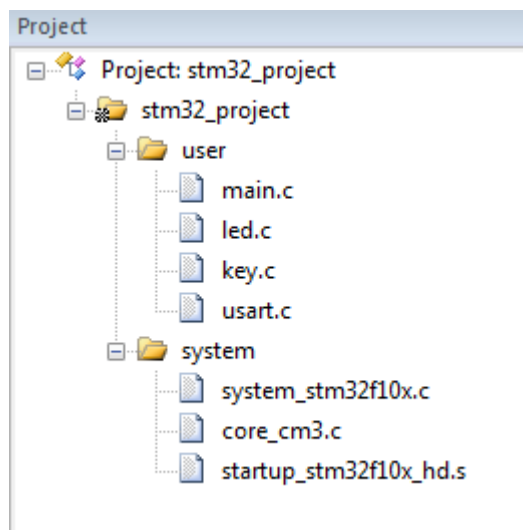
1) os_core.c: 内核数据结构管理, μC/OS-II 核心, 包含了内核的初始化, 任务切换, 事件块管理, 标志性事件组管理等功能。

- 2) os_flag.c: 标志性事件组。
 - 3) os_mbox.c: 消息邮箱。
 - 4) os_mem.c: 内存管理。
 - 5) os_mutex.c: 互斥信号量。
 - 6) os_q.c: 消息队列。
 - 7) os_sem.c: 信号量。
 - 8) os_task.c: 任务管理。
 - 9) os_tmr.c: 软件定时器。
 - 10) `ucos_ii.h`: 内部函数参数设置。
- 以文件是重要文件，在今后学习中使用到。

3.2 μ C/OS- II 的移植

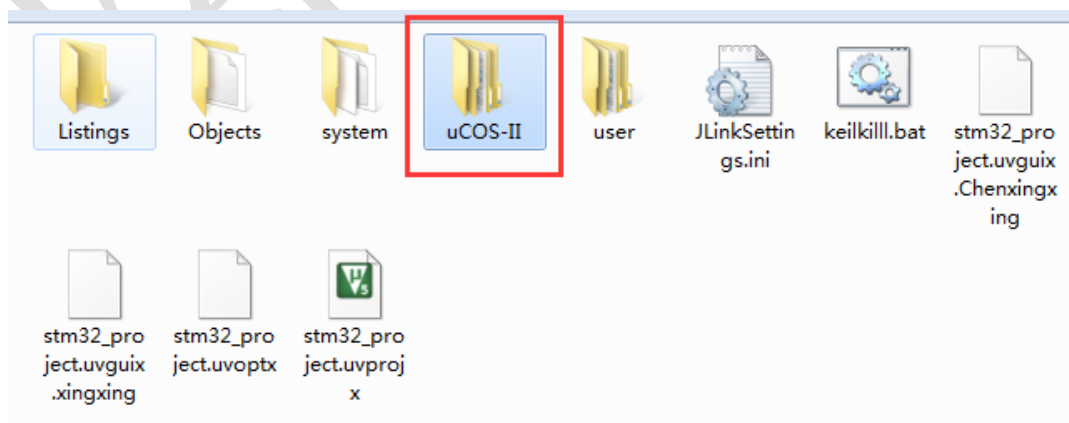
3.2.1 裸机工程的准备

找一个之前的裸机工程（这里使用的是串口工程，只实现串口基本功能的工程--stm32_project 移植前）。这个工程很普通，只是一个简单的 `urat` 工程。工程源码结构如下所示：

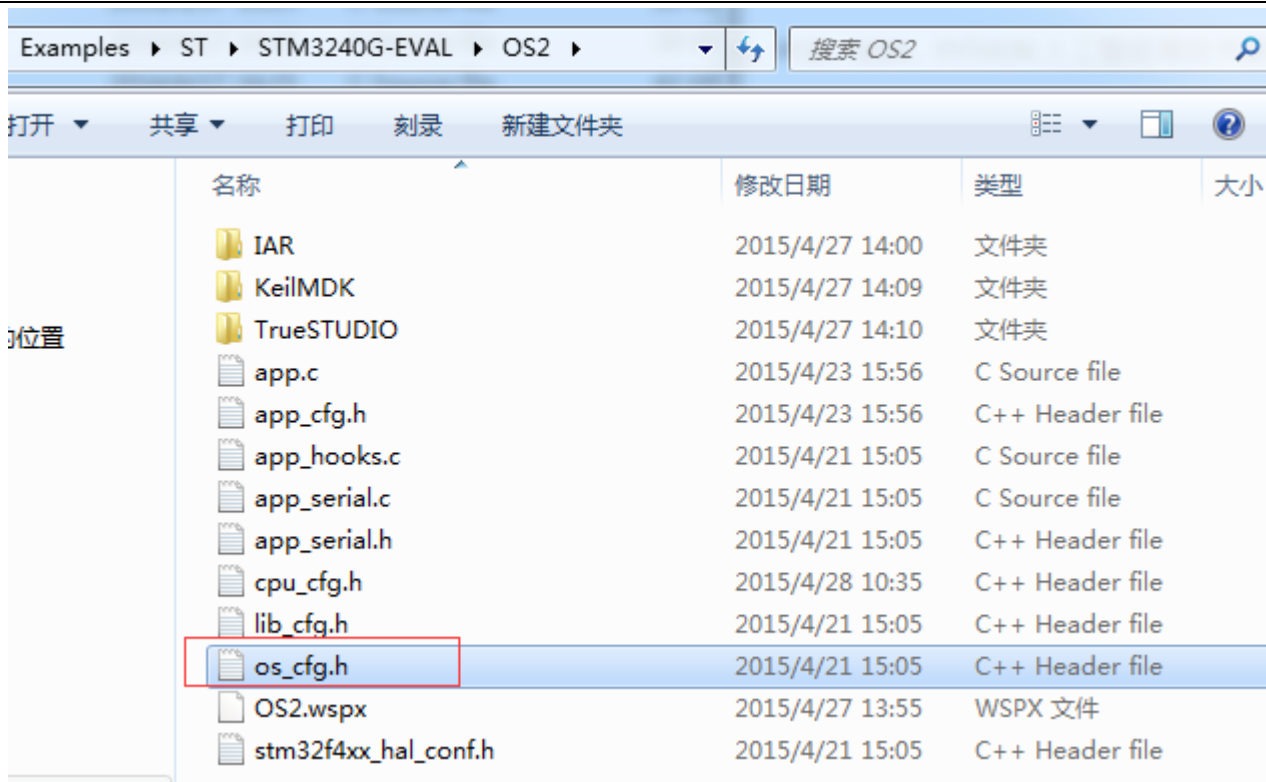


3.2.2 复制相关 μ C/OS- II 的移植文件

1. 复制 μ C/OS- II 系统源码。
打开从官方网站下载的解压好的源码工程文件夹，复制 `Micrium\Software\ μ COS-II` 文件夹到裸机工程文件夹中。



2. 复制 μ C/OS- II 配置头文件
在 `\Micrium\Examples\ST\STM3240G-EVAL\OS2` 目录中找到 `os_cfg.h` 文件，复制到裸机工程的 `stm32_project` 移植前 `\ucos-II\Source` 目录中。

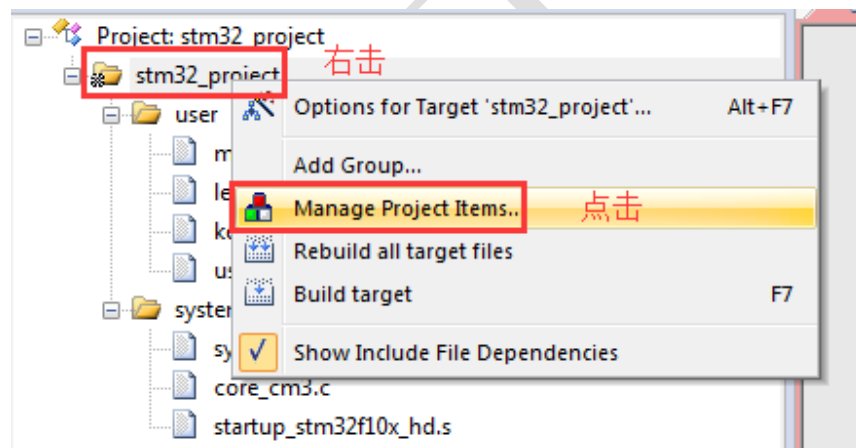


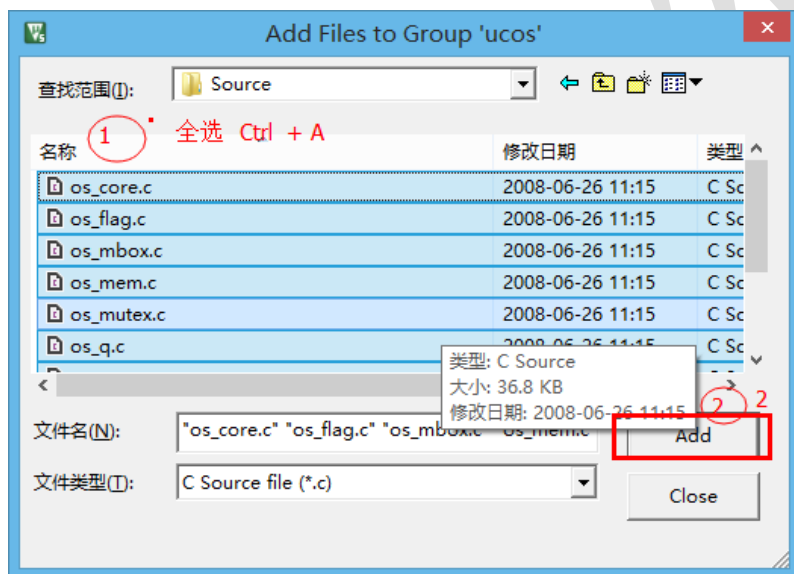
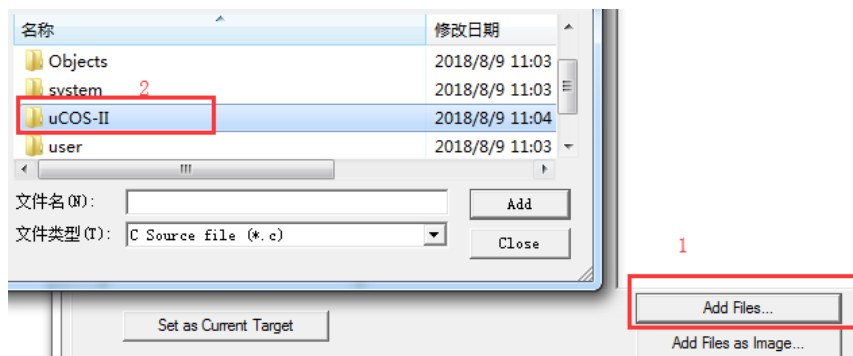
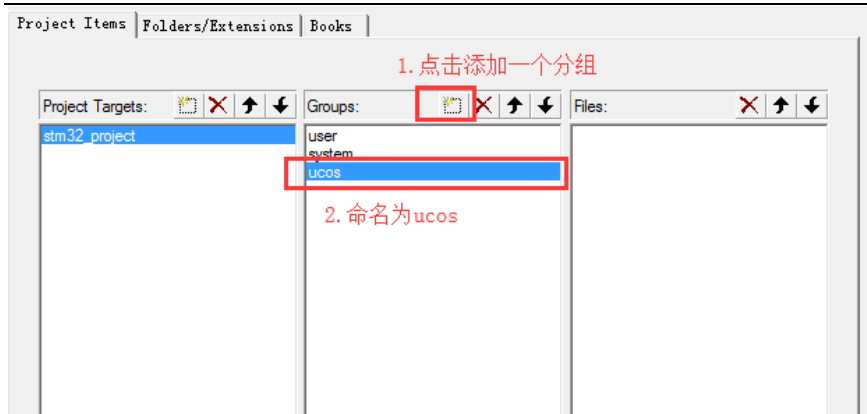
复制后，目录文件如下：

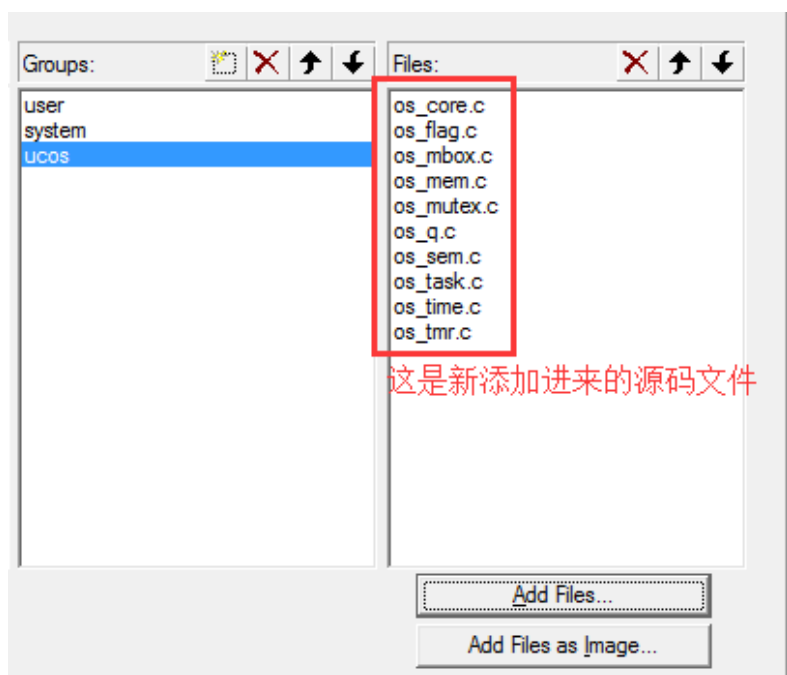
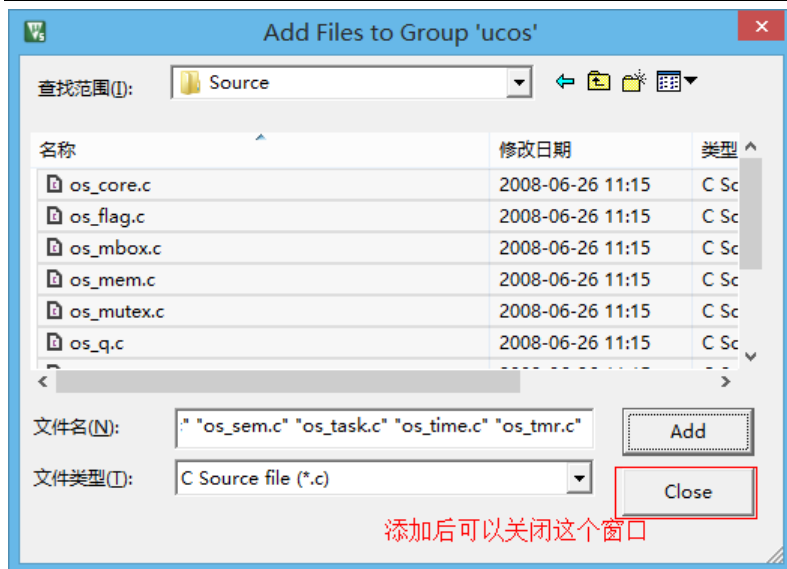
os.h	2014/4/17 16:25	C++ Header file	2 KB
os_cfg.h	2015/4/21 15:05	C++ Header file	11 KB
os_core.c	2014/12/11 16:59	C Source file	88 KB

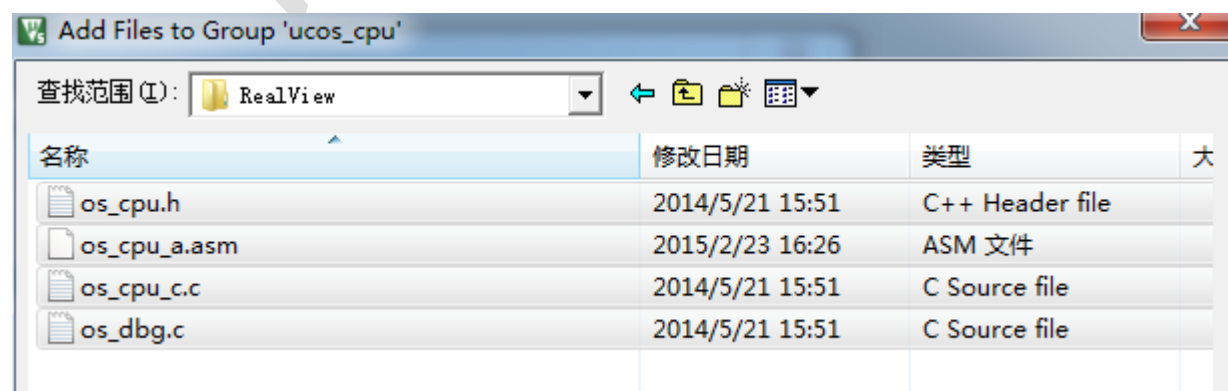
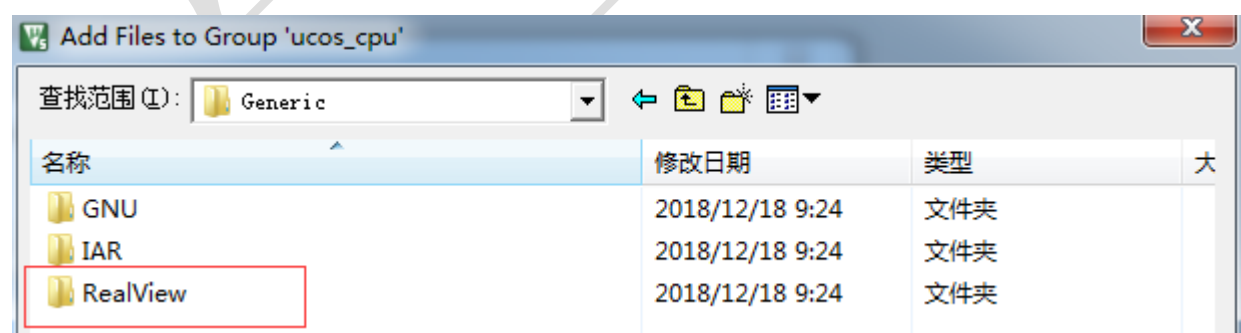
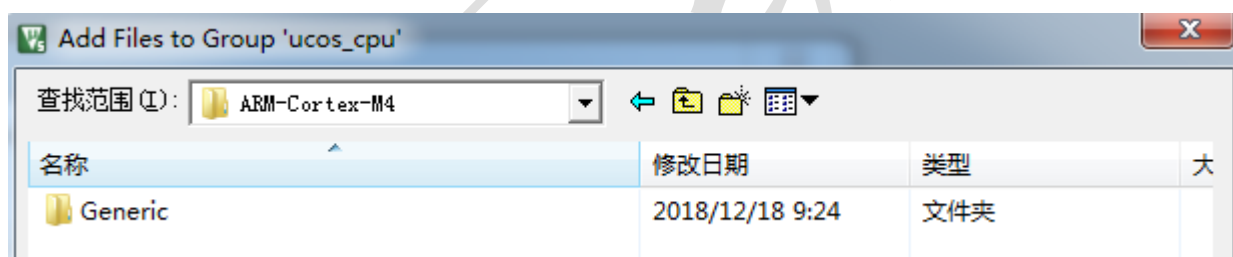
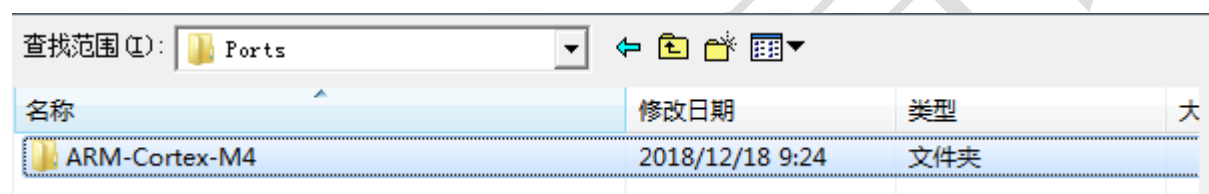
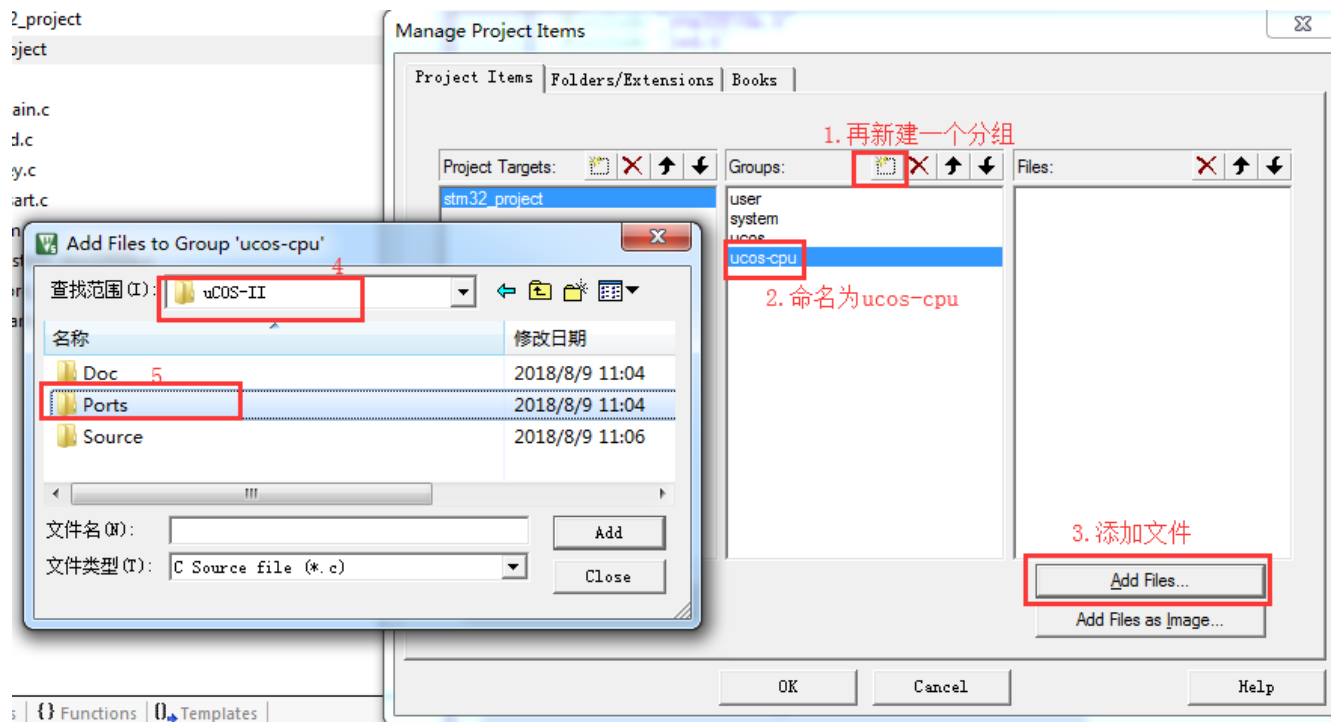
3.2.3 添加 μ C/OS- II 源码和移植 cpu 相关源码

1. 打开裸机工程，添加上一步复制过来的 μ C/OS 源码到 keil 工程中。

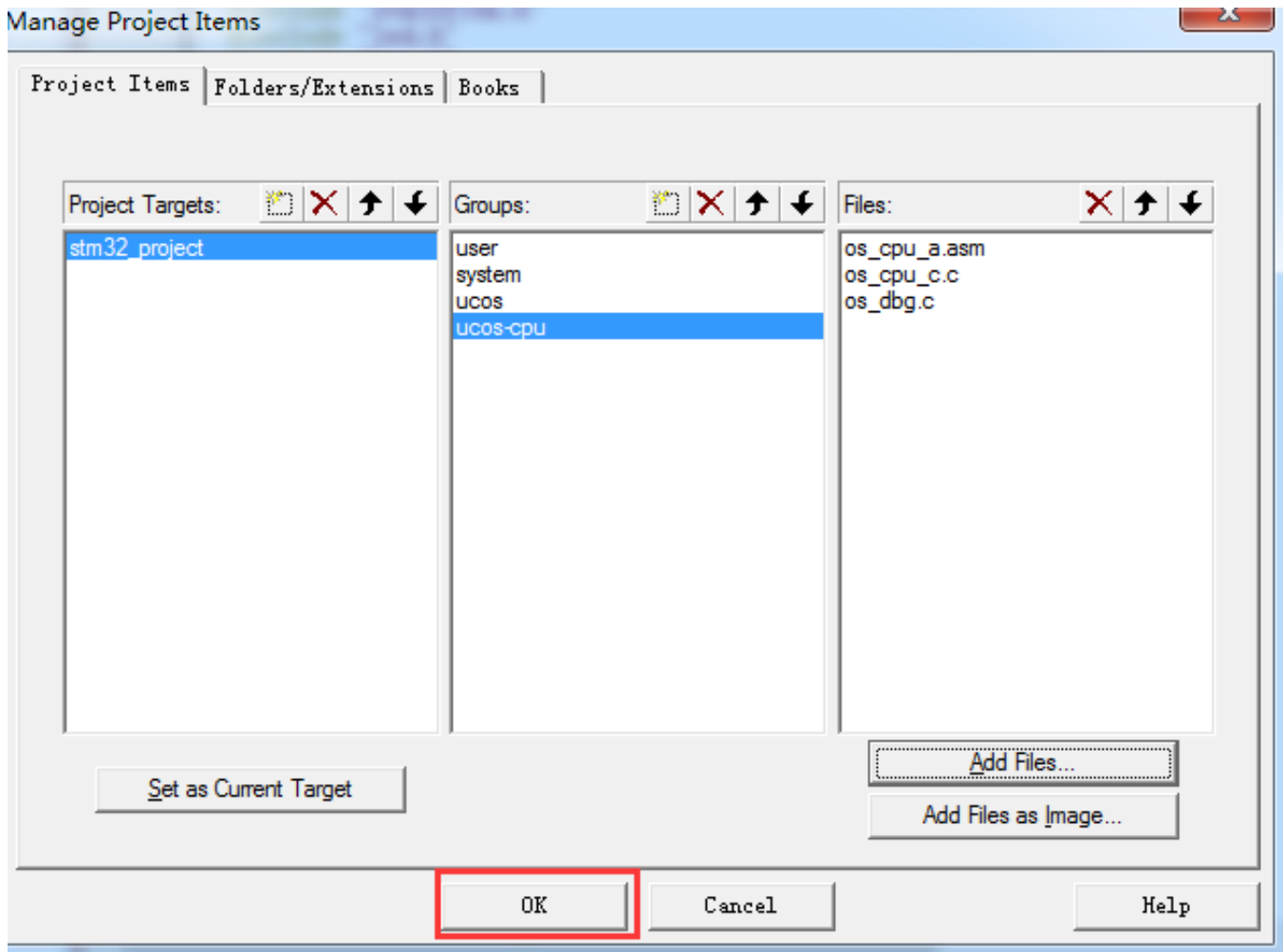




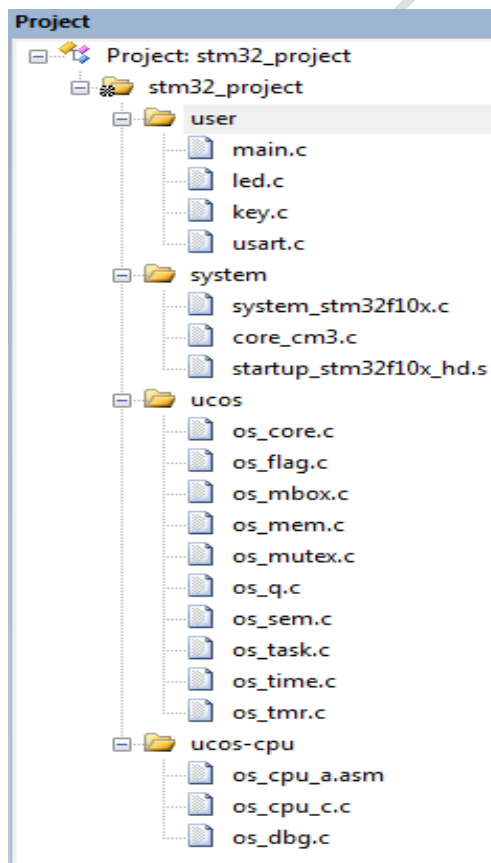




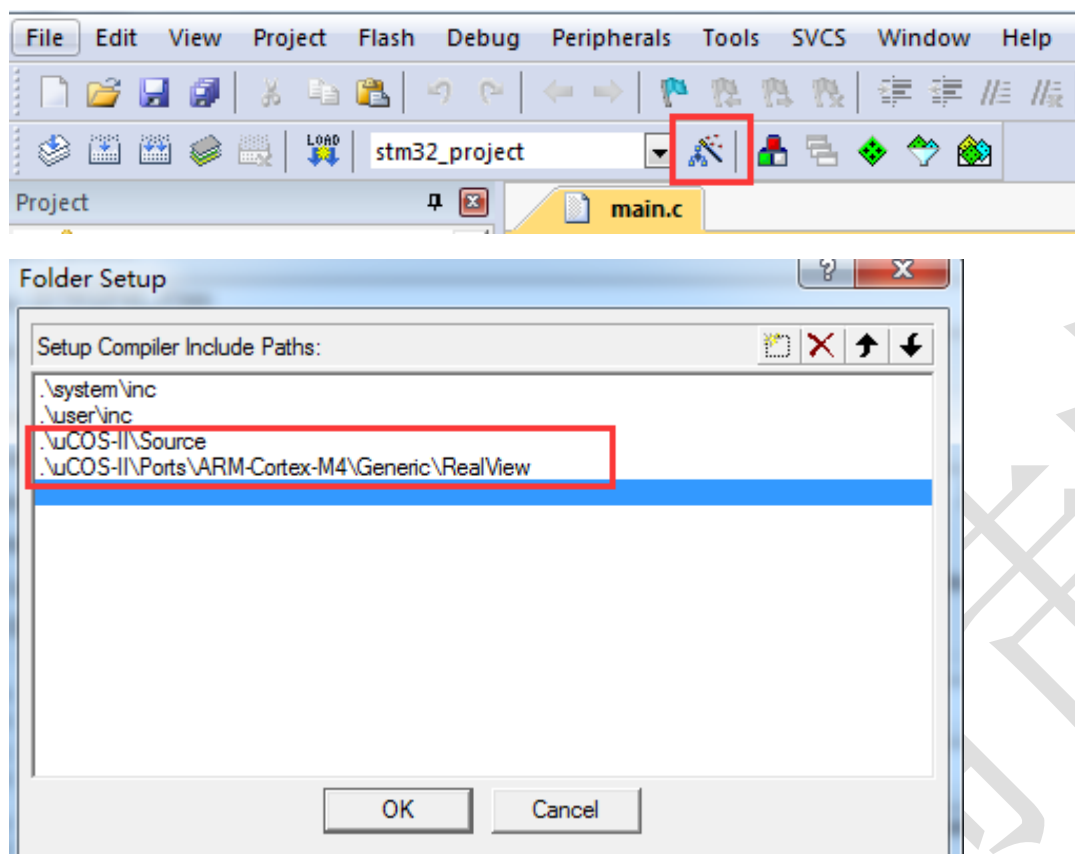
添加完成后如下图：



以下添加后工程源码结构。



3.2.4 添加 μ C/OS-II 源码和移植 cpu 相关头文件路径



3.2.5 编译纠错

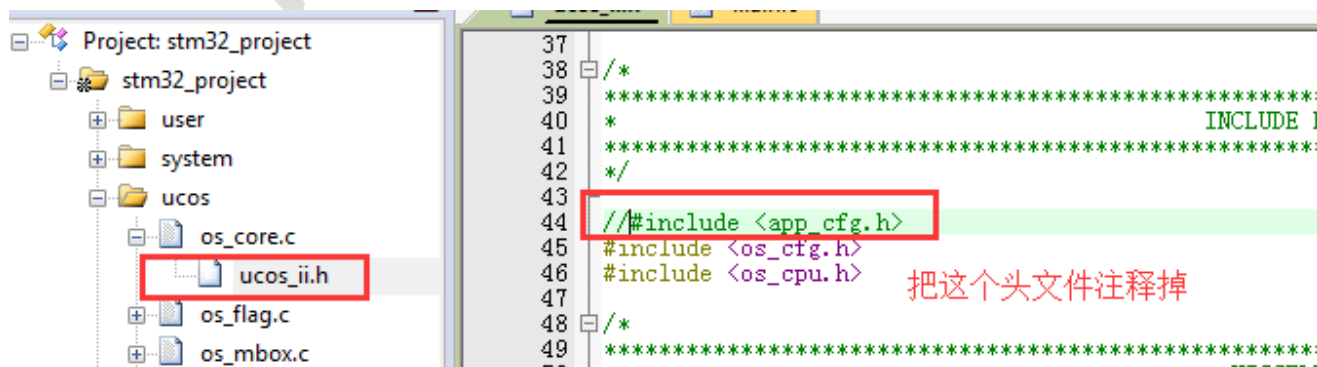
1. 编译看是否有头文件缺失。



编译结果如下，显示缺少头文件。

```
Build Output
compiling timx.c...
compiling usartx.c...
compiling usartx-dma.c...
compiling os_core.c...
.\uCOS-II\Source\ucos_ii.h(44): error: #5: cannot open source input file "app_cfg.h": No such file or directory
#include <app_cfg.h>
uCOS-II\Source\os_core.c: 0 warnings, 1 error
compiling os_flag.c...
.\uCOS-II\Source\ucos_ii.h(44): error: #5: cannot open source input file "app_cfg.h": No such file or directory
```

接下来打开 μ C/OS-II.h 文件，把#include <app_cfg.h>语句注释。如下所示：



修改后，保存，再进行编译，查看编译情况。

```
compiling os_cpu.c...
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c(53): error: #5: cannot open source input file "lib_def.h": No such file or directory
#include <lib_def.h>
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c: 0 warnings, 1 error
compiling os_dbg.c...
".\Objects\stm32f4_projcet.axf" - 1 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:09
```

```
46 /*
47 *****
48 *                               INCLUDE FILES
49 *****
50 */
51
52 #include <ucos_ii.h>
53 // #include <lib_def.h>
54
```

编译后，还有报了一堆的错误，错误信息如下：

```
*--p_stk = (CPU_STK)0x40A00000u; /* S5 */
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c(348): error: #65: expected a ";"
*--p_stk = (CPU_STK)0x40800000u; /* S4 */
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c: 0 warnings, 30 errors
".\Objects\stm32f4_projcet.axf" - 30 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:00
```

```
.....
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c(348): error: #65: expected a ";"
*--p_stk = (CPU_STK)0x40800000u; /* S4 */
*/
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu.c: 0 warnings, 30 errors
".\Objects\stm32f4_projcet.axf" - 30 Error(s), 0 Warning(s).
Target not created.
Build Time Elapsed: 00:00:00
```

双击打开错误信息，把错误代码注释。

```
// #if (OS_CPU_ARM_FP_EN == DEF_ENABLED)
//     if ((opt & OS_TASK_OPT_SAVE_FP) != (INT16U)0) {
//         *--p_stk = (CPU_STK)0x02000000u; /* FPSCR */
//     }
//     /* Initialize S0-S31 floating point registers */
//     *--p_stk = (CPU_STK)0x41F80000u; /* S31 */
//     *--p_stk = (CPU_STK)0x41F00000u; /* S30 */
//     *--p_stk = (CPU_STK)0x41E80000u; /* S29 */
//     *--p_stk = (CPU_STK)0x41E00000u; /* S28 */
//     *--p_stk = (CPU_STK)0x41D80000u; /* S27 */
//     *--p_stk = (CPU_STK)0x41D00000u; /* S26 */
//     *--p_stk = (CPU_STK)0x41C80000u; /* S25
```

```
*/
//      *--p_stk = (CPU_STK)0x41C00000u;          /* S24
*/
//      *--p_stk = (CPU_STK)0x41B80000u;          /* S23
*/
//      *--p_stk = (CPU_STK)0x41B00000u;          /* S22
*/
//      *--p_stk = (CPU_STK)0x41A80000u;          /* S21
*/
//      *--p_stk = (CPU_STK)0x41A00000u;          /* S20
*/
//      *--p_stk = (CPU_STK)0x41980000u;          /* S19
*/
//      *--p_stk = (CPU_STK)0x41900000u;          /* S18
*/
//      *--p_stk = (CPU_STK)0x41880000u;          /* S17
*/
//      *--p_stk = (CPU_STK)0x41800000u;          /* S16
*/
//      *--p_stk = (CPU_STK)0x41700000u;          /* S15
*/
//      *--p_stk = (CPU_STK)0x41600000u;          /* S14
*/
//      *--p_stk = (CPU_STK)0x41500000u;          /* S13
*/
//      *--p_stk = (CPU_STK)0x41400000u;          /* S12
*/
//      *--p_stk = (CPU_STK)0x41300000u;          /* S11
*/
//      *--p_stk = (CPU_STK)0x41200000u;          /* S10
*/
//      *--p_stk = (CPU_STK)0x41100000u;          /* S9
*/
//      *--p_stk = (CPU_STK)0x41000000u;          /* S8
*/
//      *--p_stk = (CPU_STK)0x40E00000u;          /* S7
*/
//      *--p_stk = (CPU_STK)0x40C00000u;          /* S6
*/
//      *--p_stk = (CPU_STK)0x40A00000u;          /* S5
*/
//      *--p_stk = (CPU_STK)0x40800000u;          /* S4
*/
//      *--p_stk = (CPU_STK)0x40400000u;          /* S3
*/
//      *--p_stk = (CPU_STK)0x40000000u;          /* S2
*/
//      *--p_stk = (CPU_STK)0x3F800000u;          /* S1
```



```

*/
//          *--p_stk = (CPU_STK)0x00000000u;          /* S0
*/
//  }
//endif

```

然后编译:

```

Build Output
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskCreateHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskDelHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskIdleHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskReturnHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskStatHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TaskSwHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol App_TimeTickHook (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol DEF_BIT_FIELD (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol DEF_BIT_MASK (referred from os_cpu_c.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 10 error messages.
".\Objects\stm32f4_projcet.axf" - 10 Error(s), 2 Warning(s).
Target not created

```

很明显,上面这几个错误都是由于 `os_cpu_os.c` 文件中引用了未定义的函数导致的,这种错误有两处情况,一种是函数有实现,但是没有声明,另外一种函数根本没有实现。直接使用了,也就是说我们复制文件少了某些文件。但是这些函数从名字上看是属于用户自定义的钩子函数,这种函数对我们来说是不需要的,所以我们打开 `os_cpu_os.c` 文件进行修改。

打开 `os_cpu_c.c` 文件后,按照上面编译报错的顺序找到错误的地方。

```

void OSTaskCreateHook(OS_TCB *ptcb)
{
    #if OS_APP_HOOKS_EN > 0
        App_TaskCreateHook(ptcb);
    #else
        (void)ptcb;          /* Prevent compiler warning          */
    #endif
}

#if OS_CPU_HOOKS_EN > 0
void OSTaskDelHook(OS_TCB *ptcb)
{
    #if OS_APP_HOOKS_EN > 0
        App_TaskDelHook(ptcb);
    #else
        (void)ptcb;          /* Prevent compiler warning          */
    #endif
}
#endif

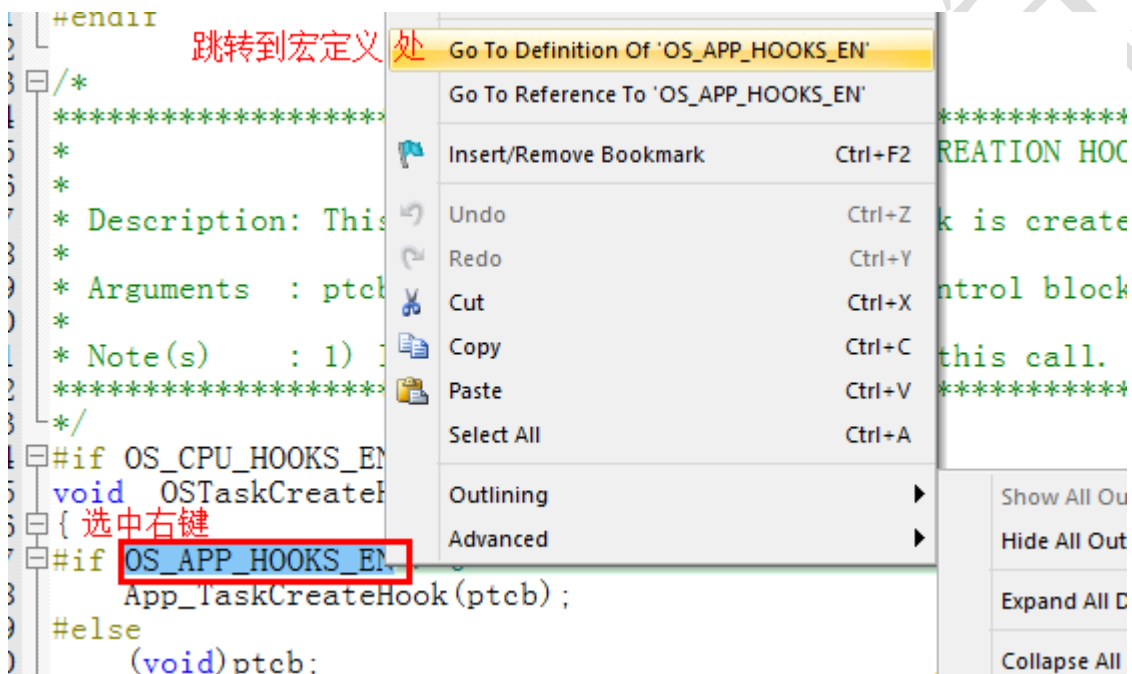
#if OS_CPU_HOOKS_EN > 0 && OS_VERSION >= 251
void OSTaskIdleHook(void)
{
    #if OS_APP_HOOKS_EN > 0
        App_TaskIdleHook();
    #endif
}

```

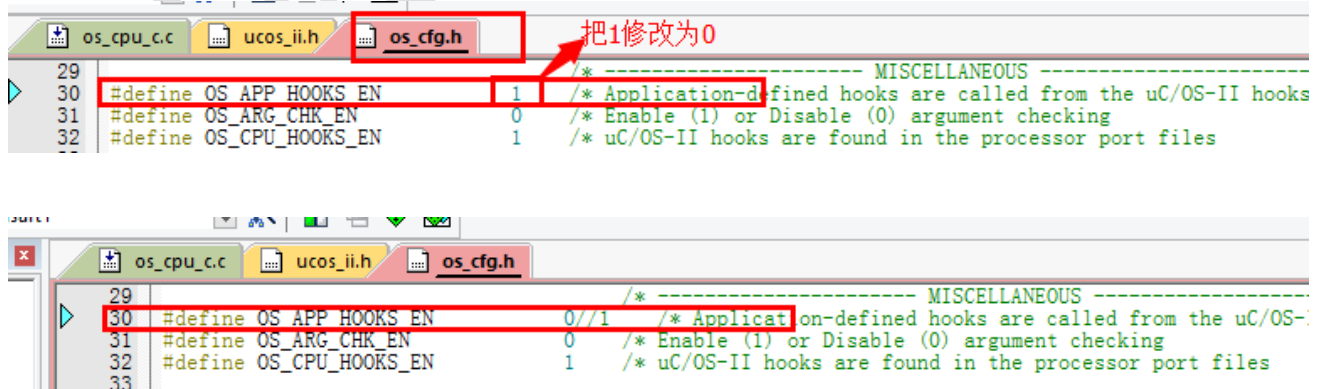
```
#endif
}
#endif

#if OS_CPU_HOOKS_EN > 0
void OSTaskStatHook(void)
{
    #if OS_APP_HOOKS_EN > 0
        App_TaskStatHook();
    #endif
}
#endif
```

以上几个地方代码都有一个共同点,所报错的函数都是当宏 OS_APP_HOOKS_EN 大小 0 时成立,这样明白了,也就是如果我们不需要这个函数,直接把 OS_APP_HOOKS_EN 设置为 0,再编译就可以了,或都把这些函数直接删除也可以。此处选择修改 OS_APP_HOOKS_EN 宏为 0。



这样可以直接跳转到宏的定义的地方,如下所示:



之后编译:

```
Build Output
    prio &= DEF_BIT_FIELD(24, 0);
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu_c.c(498): warning: #223-D: function "DEF_BIT_MASK" declared implicitly
    prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);
uCOS-II\Ports\ARM-Cortex-M4\Generic\RealView\os_cpu_c.c: 2 warnings, 0 errors
compiling os_dbg.c...
linking...
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol DEF_BIT_FIELD (referred from os_cpu_c.o).
.\Objects\stm32f4_projcet.axf: Error: L6218E: Undefined symbol DEF_BIT_MASK (referred from os_cpu_c.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 2 error messages.
".\Objects\stm32f4_projcet.axf" - 2 Error(s), 2 Warning(s).
Target not created.
```

```
88 void OS_CPU_SysTickInit (INT32U cnts)
89 {
90     INT32U prio;
91
92
93     OS_CPU_CM4_NVIC_ST_RELOAD = cnts - 1u;
94
95
96     prio = OS_CPU_CM4_NVIC_SHPRI3;          /* Set SysTick handler prio.
97     prio &= DEF_BIT_FIELD(24, 0);
98     prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);
99
100     OS_CPU_CM4_NVIC_SHPRI3 = prio;
101 }
```

此函数是操作系统使用的时钟初始化，我们可以更改为自己的时钟初始化函数。更改后如下

```
#include "stm32f4xx.h"
void OS_CPU_SysTickInit (INT32U cnts)
{
    if ((cnts - 1) > SysTick_LOAD_RELOAD_Msk) return; /* Reload value impossible */
    SysTick->LOAD = cnts - 1; /* set reload register */
    NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1); /* set Priority for SysTick Interrupt */
    SysTick->VAL = 0; /* Load the SysTick Counter Value */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk |
                    SysTick_CTRL_ENABLE_Msk; /* Enable SysTick IRQ and SysTick Timer */
    return; /* Function successful */
}

#include "stm32f4xx.h"
void OS_CPU_SysTickInit (INT32U cnts)
{
    // INT32U prio;

    // OS_CPU_CM4_NVIC_ST_RELOAD = cnts - 1u;
    /* Set SysTick handler prio. */
    // prio = OS_CPU_CM4_NVIC_SHPRI3;
    // prio &= DEF_BIT_FIELD(24, 0);
    // prio |= DEF_BIT_MASK(OS_CPU_CFG_SYSTICK_PRIO, 24);
    // OS_CPU_CM4_NVIC_SHPRI3 = prio;

    // /* Enable timer. */
    // OS_CPU_CM4_NVIC_ST_CTRL |= OS_CPU_CM4_NVIC_ST_CTRL_CLK_SRC |
    // OS_CPU_CM4_NVIC_ST_CTRL_ENABLE;
    // /* Enable timer interrupt. */
    // OS_CPU_CM4_NVIC_ST_CTRL |= OS_CPU_CM4_NVIC_ST_CTRL_INTEN;

    if ((cnts - 1) > SysTick_LOAD_RELOAD_Msk) return; /* Reload value impossible */
}
```

```

SysTick->LOAD    = cnts - 1;                                /* set reload register */
NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1); /* set Priority for SysTick Interrupt */
SysTick->VAL      = 0;                                       /* Load the SysTick Counter Value */
SysTick->CTRL     = SysTick_CTRL_CLKSOURCE_Msk |
                  SysTick_CTRL_TICKINT_Msk |
                  SysTick_CTRL_ENABLE_Msk;                  /* Enable
SysTick IRQ and SysTick Timer */
return;                                                     /* Function successful */
}

```

修改后，再编译。

这次编译结果完成没有问题。如下所示：

```

*** Using Compiler 'V5.05 update 1 (build 106)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target 'STM32F4'
compiling os_cpu_c.c...
linking...
Program Size: Code=2628 RO-data=424 RW-data=8 ZI-data=1048
FromELF: creating hex file...
".\Objects\stm32f4_projcet.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:03

```

3.2.6 修改 cpu 相关的汇编接口函数

经过大量的修改，终于编译成功，但是，到现在为止，移植工作还没有完成，接下来继续进行这艰苦的工作。在 os_cpu_c.c 实现了 systick 中断服务函数代码。如下：

```

/*
*****
*****
*
*                               SYS TICK HANDLER
*
* Description: Handle the system tick (SysTick) interrupt, which is used to generate the uC/OS-II tick
*              interrupt.
*
* Arguments   : None.
*
* Note(s)     : 1) This function MUST be placed on entry 15 of the Cortex-M3 vector table.
*****
*****
*/

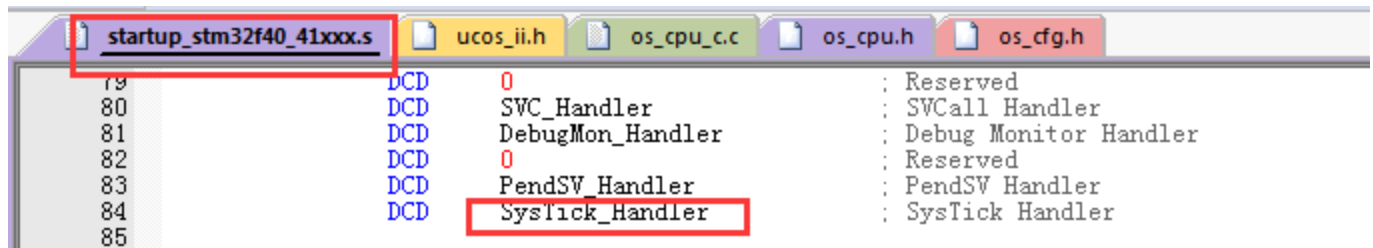
void OS_CPU_SysTickHandler (void)
{
    OS_CPU_SR  cpu_sr;
    OS_ENTER_CRITICAL();                /* Tell uC/OS-II that we are starting an ISR */
    OSIntNesting++;
    OS_EXIT_CRITICAL();

    OSTimeTick();                       /* Call uC/OS-II's OSTimeTick() */
}

```

```
OSIntExit(); /* Tell uC/OS-II that we are leaving the ISR */
}
```

注意观察，函数是 OS_CPU_SysTickHandler，而我们工程的 systick 中断服务函数名不是这个，在启动代码文件 startup_stm32f40_41xxx.s 上，systick 中断程序的名字是 SysTick_Handler，如下所示：



所以，要把这个函数名进行修改，修改 os_cpu_c.c 或 启动代码文件 startup_stm32f40_41xxx.s 的函数名都可以。此处，我选择修改启动代码文件 startup_stm32f40_41xxx.s 中的函数名字为 **OS_CPU_SysTickHandler**。先不着修改这个，还有一个函数名也要做修改，启动代码 startup_stm32f40_41xxx.s 中的 PendSV_Handler 这个异常在 M4 中也是用来实现操作系统任务切换功能的，而这个代码的中断服务程序在 μC/OS-官方下载的工程文件中也已经实现了，在 os_cpu_a.asm 文件中，如下所示：

OS_CPU_PendSVHandler

```
CPSID    I                      ; Prevent interruption during context switch
MRS      R0, PSP                ; PSP is process stack pointer
CBZ      R0, OS_CPU_PendSVHandler_nosave ; Skip register save the first time

SUBS     R0, R0, #0x20          ; Save remaining regs r4-11 on process stack
STM      R0, {R4-R11}
LDR      R1, =OSTCBCur          ; OSTCBCur->OSTCBStkPtr = SP;
LDR      R1, [R1]
STR      R0, [R1]              ; R0 is SP of process being switched out
                                   ; At this point, entire context of process has been saved
```

OS_CPU_PendSVHandler_nosave

```
PUSH     {R14}                  ; Save LR exc_return value
LDR      R0, =OSTaskSwHook      ; OSTaskSwHook();
BLX      R0
POP      {R14}
LDR      R0, =OSPrioCur        ; OSPrioCur = OSPrioHighRdy;
LDR      R1, =OSPrioHighRdy
LDRB     R2, [R1]
STRB     R2, [R0]
LDR      R0, =OSTCBCur          ; OSTCBCur = OSTCBHighRdy;
LDR      R1, =OSTCBHighRdy
LDR      R2, [R1]
STR      R2, [R0]

LDR      R0, [R2]              ; R0 is new process SP; SP = OSTCBHighRdy->OSTCBStkPtr;
LDM      R0, {R4-R11}          ; Restore r4-11 from new process stack
ADDS     R0, R0, #0x20
MSR      PSP, R0                ; Load PSP with new process SP
ORR      LR, LR, #0x04          ; Ensure exception return uses process stack
CPSIE    I
BX       LR                    ; Exception return will restore remaining context
END
```

所以接下来我们把启动代码 startup_stm32f40_41xxx.s 文件中的两个异常处理函数名修改,修改后如下所示:

```

61
62
63 ; Vector Table Mapped to Address 0 at Reset
64         AREA      RESET, DATA, READONLY
65         EXPORT    __Vectors
66         EXPORT    __Vectors_End
67         EXPORT    __Vectors_Size  导入外部函数名
68         IMPORT    OS_CPU_SysTickHandler      ;by chenxingxing
69         IMPORT    OS_CPU_PendSVHandler      ;by chenxingxing

```

```

79 DCD      0           ; Reserved
80 DCD      0           ; Reserved
81 DCD      SVC_Handler ; SVCcall Handler
82 DCD      DebugMon_Handler ; Debug Monitor Handler
83 DCD      0           ; Reserved
84 ; DCD      PendSV_Handler ; PendSV Handler
85 ; DCD      SysTick_Handler ; SysTick Handler 更忙原来的函数名
86 DCD      OS_CPU_PendSVHandler ; PendSV Handler modifi by chenxingxing
87 DCD      OS_CPU_SysTickHandler ; SysTick Handler modifi by chenxingxing

```

修改完成后编译,看是否有有问题。

```

Build Output

*** Using Compiler 'V5.05 update 1 (build 106)', folder: 'D:\Keil_v5\ARM\ARMCC\Bin'
Build target 'STM32F4'
assembling startup_stm32f40_41xxx.s...
linking...
Program Size: Code=3320 RO-data=680 RW-data=88 ZI-data=4920
FromELF: creating hex file...
".\Objects\stm32f4_projcet.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:02

```

编译没有报错,到此,μC/OS-II 移植工作已基本完成了。

3.2.7 修改 main 函数

1. 添加 μC/OS-II 头文件

```

main.c
1 #include "stm32f10x.h"
2 #include "led.h"
3 #include "key.h"
4 #include "usart.h"
5 #include "ucos_ii.h" 添加ucos所用到的头文件
6 /*

```

2. 添加 μC/OS-II 相关功能函数

```

#include "stm32f4xx.h"
#include "led.h"
#include "key.h"
#include "usart.h"
#include "string.h"
#include "ucos_ii.h"

#define SystemFrequency 168000000

```

```
/* 用户任务 */

#define START_TASK_PRIO 8          //定义 Start 的优先级
#define StartStkLengh 128         //定义 Start 的堆栈长度
OS_STK StartStk[StartStkLengh];   //定义 Start 的堆栈
void start_task(void *pdata);
int main ()
{

    LED_Init(); //LED 初始化
    KEY_Init(); //按键初始化
    usart1_init(9600); //串口初始化，波特率为 9600
    OS_CPU_SysTickInit(SystemFrequency / OS_TICKS_PER_SEC); //系统滴答初始化
    OSInit(); //OS 初始化
    OSTaskCreate(start_task, NULL, &StartStk[StartStkLengh - 1], START_TASK_PRIO); //创建一个名为 start_task 的任务
    OSStart(); //开启 os, 开始调度, 跳到已经就绪优先级最高的任务中
    while(1)
    {

    }

}

void start_task(void *pdata)
{
    pdata=pdata;
    while (1)
    {
        LED1=!LED1;
        OSTimeDly(100); //释放 CPU 的使用权
    }

}
```

第四章 μ C/OS- II 任务管理

4.1 μ C/OS- II 任务的创建

μ C/OS- II 操作系统由任务组成, μ C/OS- II 提供了一些任务函数给用户使用。我们用户可以直接调用 API 函数来实现任务的创建。

在 μ C/OS- II 中, 跟任务相关的函数在 `os_task.c` 文件中有相关的函数。

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

1. 最先肯定是要调用 `CPU_Init()` 初始化 μ C/OS- II。

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, 再在调用 `OSTaskCreate()` 函数创建任务。

3. 最后调用 `OSStart()` 函数开启 μ C/OS- II。

4.2 μ C/OS- II 任务相关函数

4.2.1 `OSTaskCreate()`

函数原型:

```
INT8U OSTaskCreate (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio)
```

函数功能:

创建一个新的 μ C/OS 任务。注意: 任务的建立可以在多任务启动之前, 也可以在正在执行任务中创建。但是, 不能在中断服务函数中创建任务。每个任务都必须要有有一个死循环。

函数参数:

`void (*task)(void *p_arg)` : 这是一个函数指针的定义, 指向任务函数。

`void *p_arg` : 指向一个数据结构, 此结构是用来在创建任务的时候, 向任务函数传递的参数。

`OS_STK *ptos` : 指向任务堆栈栈顶的指针。任务堆栈是用来保存局部变量, 函数参数, 返回地址及任务被中断时的 CPU 寄存器内容。

`INT8U prio` : 任务的优先级。每个任务都必须有一个唯一的优先级作为任务的标识。数值越小, 优先级越高。

返回值:

`OSTaskCreate()` 返回值为如下之一:

- 1) `OS_ERR_NONE` : 任务创建成功
- 2) `OS_PRIO_EXIT` : 此优先用的任务已经存在
- 3) `OS_ERR_PRIO_INVALID` : 设置的优先级大于 `OS_LOWEST_PRIO`
- 4) `OS_ERR_TASK_CREATE_ISR` : 在中断服务函数中创建任务

说明:

用户程序中不建议使用优先级 0, 1, 2, 3, 以及 `OS_LOWEST_PRIO-3`, `OS_LOWEST_PRIO-2`, `OS_LOWEST_PRIO-1`, `OS_LOWEST_PRIO`。这些优先级 μ C/OS 系统保留, 其余的 56 个优先级提供给应用程序。

示例:

```
#define Start_Task_Pro 10          //定义 Start 任务的优先级
#define Start_Stk_Size 128        //定义 Start 任务栈的大小
void Start_Task(void *pdata);      //起始任务的声明
OS_STK Start_Task_Stk[Start_Stk_Size]; //定义 Start 任务的栈
int main(void)
{
    OSInit();                      //OS 初始化
    OS_CPU_SysTickInit();          //OS 系统时钟初始化
    OSTaskCreate(Start_Task, NULL, &Start_Task_Stk[Start_Stk_Size-1], Start_Task_Pro); //创建 start 任务
```



```

    OSStart();                //开启 OS,开启调度, 执行任务优先最高的任务。
}

void Start_Task(void *pdata)
{
    pdata=pdata; //防止编译出警告, pdata 没有使用
    while(1)
    {
        /*任务代码*/
        OSTimeDly(1000); //延时 1000ms
    }
}

```

4.2.2 OSTaskCreateExt ()

函数原型:

```

INT8U  OSTaskCreateExt ( void    (*task)(void *p_arg),
                        void      *p_arg,
                        OS_STK    *ptos,
                        INT8U      prio,
                        INT16U     id,
                        OS_STK    *pbos,
                        INT32U     stk_size,
                        void      *pext,
                        INT16U     opt)

```

函数功能:

建立一个新任务。任务的建立可以在多任务环境启动之前,也可以在正在运行的任务中建立,但中断处理程序中不能建立新任务。用 OSTaskCreateExt()来创建任务会更加的灵活,但会增加一些额外的开销。

函数参数:

void (*task)(void *p_arg) : 这是一个函数指针的定义,指向任务函数。

void *p_arg : 指向一个数据结构,此结构是用来在创建任务的时候,向任务函数传递的参数。

OS_STK *ptos : 指向任务堆栈栈顶的指针。任务堆栈是用来保存局部变量,函数参数,返回地址及任务被中断时的 CPU 寄存器内容。

INT8U prio : 任务的优先级。每个任务都必须有一个唯一的优先级作为任务的标识。数值越小,优先级越高。

INT16U id : 是任务的标识,目前这个参数没有实际的用途,但保留在 OSTaskCreateExt () 中供今后扩展,应用程序中可设置 id 与优先级相同。

OS_STK *pbos : 指向堆栈底端的指针。如果初始化常量 OS_STK_GROWTH 设为 1,堆栈被设为从内存高地址向低地址增长。此时 pbos 应该指向任务堆栈空间的最低地址。反之,如果 OS_STK_GROWTH 设为 0,堆栈将从低地址向高地址增长。pbos 应该指向堆栈空间的最高地址。参数 pbos 用于堆栈检测函数 OSTaskStkChk ()。

INT32U stk_size : 指定任务堆栈的大小。其单位由 OS_STK 定义:当 OS_STK 的类型定义为 INT8U、INT16U、INT32U 的时候,stk_size 的单位分别为字节(8 位)、字(16 位)和双字(32 位)。

void *pext : 用户定义数据结构的指针,可作为 TCB 的扩展。例如,当任务切换时,用户定义的数据结构中可存放浮点寄存器的数值,任务运行时间,任务切入次数等信息。

INT16U opt : 存放与任务相关的操作信息。opt 的低 8 位由 uC/OS 保留,用户不能使用。用户可以使用 opt 的高 8 位。每一种操作由 opt 中的一位或几位指定,当相应的位被置位时,表示选择某种操作。当前的 μC/OS- 版本支持下列操作:

OS_TASK_OPT_STK_CHK: 决定是否进行任务栈检查。

OS_TASK_OPT_STK_CLR: 决定是否清空堆栈。

OS_TASK_OPT_SAVE_FP: 决定是否保存浮点寄存器的数值。此项操作仅当处理器有浮点硬件时有效。保存操作由硬件相关的代码完成。

返回值:

OSTaskCreateExt ()返回值为如下之一:

- 1) OS_ERR_NONE : 任务创建成功
- 2) OS_PRIO_EXIT : 此优先用的任务已经存在
- 3) OS_ERR_PRIO_INVALID : 设置的优先级大于 OS_LOWEST_PRIO
- 4) OS_ERR_TASK_CREATE_ISR : 在中断服务函数中创建任务

说明:

用户程序中不建议使用优先级 0, 1, 2, 3, 以及 OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1, OS_LOWEST_PRIO。这些优先级 uC/OS 系统保留, 其余的 56 个优先级提供给应用程序。

示例:

```
#define Start_Task_Pro 10          //定义 Start 任务的优先级
#define Start_Stk_Size 128        //定义 Start 任务栈的大小
void Start_Task(void *pdata);      //起始任务的声明
OS_STK Start_Task_Stk[Start_Stk_Size]; //定义 Start 任务的栈
int main(void)
{
    OSInit();                      //OS 初始化
    OS_CPU_SysTickInit();          //OS 系统时钟初始化
    OSTaskCreateExt( Start_Task,    //任务函数
                    NULL,           //传递给任务参数的实参
                    &Start_Task_Stk[Start_Stk_Size-1], //栈顶的地址
                    Start_Task_Pro, //任务优先级
                    Start_Task_Pro, //任务 id
                    &Start_Task_Stk[0], //栈底的地址
                    Start_Stk_Size,   //任务栈的大小
                    "my first μC/OS- task", //用户自定义
                    OS_TASK_OPT_STK_CHK //允许任务栈检查
                );
    OSStart();                     //开启 OS,开启调度, 执行任务优先最高的任务。
}

void Start_Task(void *pdata)
{
    pdata=pdata; //防止编译出警告, pdata 没有使用
    while(1)
    {
        /*任务代码*/
        OSTimeDly(1000); //延时 1000ms
    }
}
```

4.2.3 OSTaskDel()

函数原型:

INT8U OSTaskDel (INT8U prio)

函数功能:

删除一个指定优先级的任务。一个任务可以把自己的优先级传给 OSTaskDel () 函数，从而把自己删除。如果一个任务不知道自己优先级，也可以传递 OS_PRIO_SELF。被删除的任务将回到睡眠状态。一个任务也可以删除其他优先级的任务。当任务被删除之后，只能通过 OSTaskCreate () 或 OSTaskCreateExt () 重新创建。

函数参数:

INT8U prio : 要删除的任务优先级，也可能是参数 OS_PRIO_SELF 。

返回值:

- 1) OS_ERR_NONE : 成功删除任务
- 2) OS_ERR_TASK_DEL_IDLE : 错误操作，试图删除空闲任务
- 3) OS_ERR_PRIO_INVALID : 优先级大于 OS_LOWEST_PRIO
- 4) OS_ERR_TASK_DEL : 指定的任务已被删除
- 5) OS_ERR_TASK_NOT_EXIST : 指定的任务不存在。
- 6) OS_ERR_TASK_DEL_ISR : 错误操作，试图在中断服务函数中删除任务

示例:

```
void TaskX(void *pdata)
{
    INT8U err;
    while(1)
    {
        err = OSTaskDel(10); /* 删除优先级为 10 的任务*/
        if (err == OS_NO_ERR)
        {
            /* 任务被删除 */
        }
    }
}
```

4.2.4 OSTaskDelReq()

函数原型:

INT8U OSTaskDelReq (INT8U prio)

函数功能:

函数请求一个任务删除自身。通常 OSTaskDelReq () 用于删除一个占有系统资源的任务（例如任务建立了信号量）。对于此类任务，在删除任务之前应当先释放任务占用的系统资源。具体的做法是：在需要被删除的任务中调用 OSTaskDelReq () 检测是否有其他任务的删除请求，如果有，则释放自身占用的资源，然后调用 OSTaskDel () 删除自身。例如，假设任务 5 要删除任务 10，而任务 10 占有系统资源，此时任务 5 不能直接调用 OSTaskDel (10) 删除任务 10，而应该调用 OSTaskDelReq (10) 向任务 10 发送删除请求。在任务 10 中调用 OSTaskDelReq (OS_PRIO_SELF)，并检测返回值。如果返回 OS_TASK_DEL_REQ，则表明有来自其他任务的删除请求，此时任务 10 应该先释放资源，然后调用 OSTaskDel (OS_PRIO_SELF) 删除自己。任务 5 可以循环调用 OSTaskDelReq (10) 并检测返回值，如果返回 OS_TASK_NOT_EXIST，表明任务 10 已经成功删除。

函数参数:

INT8U prio : 要删除的任务优先级。如果参数为 OS_PRIO_SELF，则表示调用函数的任务正在查询是否有来自其他任务的删除请求。

返回值:

- 1) OS_ERR_NONE : 删除请求已经被任务记录
- 2) OS_ERR_TASK_NOT_EXIST: 指定的任务不存。发送删除请求的任务可以等待此返回值, 看删除是否成功。
- 3) OS_ERR_TASK_DEL : 指定的任务已被删除
- 4) OS_ERR_TASK_DEL_IDLE: 错误操作, 试图删除空闲任务
- 5) OS_ERR_PRIO_INVALID : 参数指定的优先级大于 OS_LOWEST_PRIO 或没有设定 OS_PRIO_SELF 的值。
- 6) OS_ERR_TASK_DEL_REQ: 当前任务收到来自其他任务的删除请求。

示例:

```
void Led_Task(void *pdata)
{
    u8 i;
    pdata=pdata;
    while (1)
    {
        for(i=0;i<5;i++)
        {
            printf("1\r\n");
            OSTimeDly(1000);
        }
        if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ)
        {
            /* 释放任务占用的系统资源 */
            /* 释放动态分配的内存 */
            OSTaskDel(OS_PRIO_SELF);
        }
        OSTimeDly(100);
    }
}

void Key_Task(void *pdata)
{
    INT8U err;
    pdata=pdata;
    while (1)
    {
        if(KEY_Scan(0))//按键按下
        {
            err = OSTaskDelReq(Led_Task_Pro); /* 请求任务优先级为 Led_Task_Pro 的任务 */
            if (err == OS_NO_ERR)
            {
                err = OSTaskDelReq(Led_Task_Pro);
                while (err != OS_TASK_NOT_EXIST)
                {
                    OSTimeDly(1); /* 等待任务删除 */
                }
                /* 任务优先级为 Led_Task_Pro 已被删除 */
            }
        }
    }
}
```

```
    }  
    OSTimeDly(100);  
}  
}
```

4.2.5 OSTaskSuspend ()

函数原型:

```
INT8U OSTaskSuspend (INT8U prio)
```

函数功能:

无条件挂起一个任务，调用此函数也可以挂起自己。如果一个任务被挂起了，那么这个任务就会处于等待状态，不参与任务的调度。说明此任务不会再次被执行。当任务被挂起后，操作系统会重新进行任务的调度，运行下一个优先级最高的任务。如果想再次运行被挂起的任务，可以用唤醒任务函数 `OSTaskResume ()` 进行任务的唤醒。

函数形参:

要挂起的任务优先级，也可能是参数 `OS_PRIO_SELF`，如果是 `OS_PRIO_SELF`，说明任务挂起的是自己。

返回值:

- 1) `OS_ERR_NONE` : 任务被成功挂起
- 2) `OS_ERR_TASK_SUSPEND_IDLE` : 错误操作，试图挂起空闲任务
- 3) `OS_ERR_PRIO_INVALID` : 优先级大于等于 `OS_LOWEST_PRIO`
- 4) `OS_ERR_TASK_SUSPEND_PRIO` : 试图挂起的任务不存在
- 5) `OS_ERR_TASK_NOT_EXISTS` : 指定的任务没有创建或已被删除。

示例:

```
void TaskX(void *pdata)  
{  
    INT8U err;  
    while(1)  
    {  
        err = OSTaskSuspend(OS_PRIO_SELF); /* 挂起当前任务 */  
        /* 当其他任务唤醒被挂起任务时，任务可继续运行 */  
    }  
}
```

4.2.6 OSTaskResume ()

函数原型:

```
INT8U OSTaskResume (INT8U prio)
```

函数功能:

唤醒 `OSTaskSuspend ()` 函数挂起的任务。

函数参数:

唤醒任务的优先级。

返回值:

- 1) `OS_ERR_NONE` : 任务被成功被唤醒
- 2) `OS_ERR_PRIO_INVALID` : 优先级大于等于 `OS_LOWEST_PRIO`
- 3) `OS_ERR_TASK_RESUME_PRIO` : 试图唤醒的任务不存在
- 4) `OS_ERR_TASK_NOT_EXIST` : 指定的任务没有创建或已被删除。
- 5) `OS_ERR_TASK_NOT_SUSPENDED`: 任务不在挂起状态

示例:

```
void TaskX(void *pdata)
```

```

{
    INT8U err;
    while(1)
    {
        err = OSTaskResume(10); /* 唤醒优先级为 10 的任务 */
        if (err == OS_NO_ERR)
        {
            /* 任务被唤醒 */
        }
    }
}

```

4.2.7 OSTaskChangePrio ()

函数原型

INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)

函数功能:

改变一个任务的优先级。

函数参数:

INT8U oldprio : 任务原来的优先级

INT8U newprio: 任务新的优先级

返回值:

- 1) OS_ERR_NONE : 成功设置
- 2) OS_ERR_PRIO_INVALID : 优先级大于等于 OS_LOWEST_PRIO
- 3) OS_ERR_PRIO_EXIST : 新的优先级已经被设置
- 4) OS_ERR_PRIO : 原来的优先级不存在
- 5) OS_ERR_TASK_NOT_EXIST: 指定的任务没有创建或已被删除。

```

void TaskX(void *data)
{
    INT8U err;
    while(1)
    {
        err = OSTaskChangePrio(10, 15);
    }
}

```

4.2.8 OSTaskStkChk()

函数原型:

INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *p_stk_data)

函数功能:

OSTaskStkChk () 检查任务堆栈状态, 计算指定任务堆栈中的未用空间和已用空间。

使用 OSTaskStkChk () 函数要求所检查的任务是被 OSTaskCreateExt () 函数建立的, 且 opt

参数中 OS_TASK_OPT_STK_CHK 操作项打开。

计算堆栈未用空间的方法是从堆栈底端向顶端逐个字节比较, 检查堆栈中 0 的个数, 直到一个非 0 的数值出现。这种方法的前提是堆栈建立时已经全部清零。要实现清零操作, 需要在任务建立初始化堆栈时设置 OS_TASK_OPT_STK_CLR 为 1。如果应用程序在初始化时已经将全部 RAM 清零, 且不进行任务删除操作, 也可以设置 OS_TASK_OPT_STK_CLR 为 0, 这将加快 OSTaskCreateExt () 函数的执行速度。

函数参数:

INT8U prio : 要获取堆栈信息的任务优先级, 也可以指定参数 OS_PRIO_SELF, 获取调用任务本身的信息。

OS_STK_DATA *p_stk_data: 指向一个类型为 OS_STK_DATA 的数据结构, 其中包含如下信息:

INT32U OSFree; /* 堆栈中未使用的字节数 */

INT32U OSUsed; /* 堆栈中已使用的字节数 */

返回值:

- 1) OS_ERR_NONE: 函数调用成功
- 2) OS_ERR_PRIO_INVALID: 参数指定的优先级大于 OS_LOWEST_PRIO, 或未指定 OS_PRIO_SELF。
- 3) OS_ERR_TASK_NOT_EXIST: 指定的任务不存在
- 4) OS_ERR_TASK_OPT: 任务用 OSTaskCreateExt() 函数建立的时候没有指定 OS_TASK_OPT_STK_CHK 操作, 或者任务是用 OSTaskCreate() 函数建立的。
- 5) OS_ERR_PDATA_NULL: 数据指针为空。

示例:

```
void Task (void *pdata)
{
    OS_STK_DATA stk_data;
    INT32U stk_size;
    while(1)
    {
        err = OSTaskStkChk(10, &stk_data);
        if (err == OS_NO_ERR)
        {
            stk_size = stk_data.OSFree + stk_data.OSUsed;
        }
    }
}
```

4.2.9 OSTaskQuery()

函数原型:

INT8U OSTaskQuery (INT8U prio, OS_TCB *p_task_data)

函数功能:

任务的相关信息。应用程序必须分配的 OS_TCB 数据结构来获得所需的任务控制块的信息。您的副本包含了 OS_TCB 结构体的各个成员。在访问 OS_TCB 结构的时候, 特别是 OSTCBNext 和 OSTCBPrev 的内容, 它们指向下一个和以前 OS_TCB 的任务链, 所以必须小心使用。

函数参数:

INT8U prio : 要获取数据的任务优先级。可以通过指定 OS_PRIO_SELF 调用任务本身的信息。

OS_TCB *p_task_data: 是指向 OS_TCB, 其包含任务控制块的拷贝信息。

返回值:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_PRIO_INVALID: 指定的优先级比 OS_LOWEST_PRIO 高。
- 3) OS_ERR_PRIO: 试图获得一个无效的任务信息。
- 4) OS_ERR_TASK_NOT_EXIST: 任务被分配到一互斥 PIP。
- 5) OS_ERR_PDATA_NULL: p_task_data 是一个空指针。

说明:

在任务控制块的字段取决于以下配置选项 (参见 OS_CFG.H):

- 1) OS_TASK_CREATE_EN
- 2) OS_Q_EN
- 3) OS_FLAG_EN
- 4) OS_MBOX_EN
- 5) OS_SEM_EN

6) OS_TASK_DEL_EN

示例:

```
void Task (void *p_arg)
{
    OS_TCB task_data;
    INT8U err;
    void *pext;
    INT8U status;
    p_arg= p_arg;
    while(1)
    {
        //...用户代码
        err = OSTaskQuery(OS_PRIO_SELF, &task_data);
        if (err == OS_ERR_NONE)
        {
            pext = task_data.OSTCBExtPtr; /* Get TCB extension pointer */
            status = task_data.OSTCBStat; /* Get task status */
            //...用户代码
        }
        //...用户代码
    }
}
```

4.2.10 OSTaskNameGet()

函数原型:

```
INT8U OSTaskNameGet (INT8U prio, INT8U *pname, INT8U *perr)
```

函数功能:

获取指定优先级任务的名称。

函数参数:

INT8U prio : 是你将要从中获取从名字的任务的优先级。如果指定 OS_PRIO_SELF, 你会获得当前任务的名称。

INT8U *pname: 指向包含该任务的名称的 ASCII 字符串。

INT8U *perr : 指向错误代码变量的指针, 其返回值如下:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_TASK_NOT_EXIST: 指定的任务没有创建或已被删除。
- 3) OS_ERR_PRIO_INVALID: 指定了一个无效的优先级。
- 4) OS_ERR_PNAME_NULL: pname 你传递一个 NULL 指针。
- 5) OS_ERR_NAME_GET_ISR: 尝试从 ISR 此功能, 不允许。

返回值:

指向 pname 的 ASCII 字符串的大小; 如果为 0, 说明遇到错误。

示例:

```
INT8U *EngineTaskName; //存放名字字符串指针
void Task (void *p_arg)
{
    INT8U err;
    INT8U size; //存放任务名的长度
    p_arg= p_arg;
```



```
while(1)
{
    //...用户代码
    size = OSTaskNameGet(OS_PRIO_SELF, &EngineTaskName, &err);
}
}
```

4.2.11 OSTaskNameSet()

函数原型:

```
void OSTaskNameSet (INT8U prio, INT8U *pname, INT8U *perr)
```

函数功能:

设置任务名称。

函数参数:

INT8U prio: 将要设置名字的任务的优先级。如果指定 OS_PRIO_SELF, 将设置当前任务的名称。

INT8U *pname: 指向包含任务名称的 ASCII 字符串。

INT8U *perr: 指向错误代码变量的指针, 其返回值如下:

- 1) OS_ERR_NONE: 设置成功。
- 2) OS_ERR_TASK_NOT_EXIST: 指定的任务没有创建或已被删除。
- 3) OS_ERR_PNAME_NULL: pname 指向 NULL。
- 4) OS_ERR_NAME_SET_ISR: 在 ISR 使用本函数, 不允许。
- 5) OS_ERR_PRIO_INVALID: 指定了无效的优先级。

返回值:

无。

示例:

```
void Task (void *p_arg)
{
    INT8U err;
    p_arg= p_arg;
    while(1)
    {
        OSTaskNameSet(OS_PRIO_SELF,"Engine Task", &err);//表示设置任务本身
        //...用户代码
    }
}
```

4.2.12 OSStart ()

函数原型:

```
void OSStart (void)
```

函数功能:

启动 uC/OS-II 的多任务环境。

函数参数:

无。

返回值:

无。

说明:

- 1) 在调用 OSStart()之前必须先调用 OSInit ()。
- 2) 在用户程序中 OSStart()只能被调用一次。第二次调用 OSStart()将不进行任何操作。

示例:

```
void main (void)
{
    //.....用户代码
    OSInit(); // 初始化 uC/OS-II
    //.....用户代码 // 最少创建一个任务
    OSStart(); //启动多任务内核
}
```

4.3 μ C/OS- II 延时函数

4.3.1 OSTimeDly()

函数原型:

```
void OSTimeDly (INT16U ticks)
```

函数功能:

将一个任务延时若干个时钟节拍。如果延时时间大于 0，系统将立即进行任务调度。延时时间的长度可从 0 到 65535 个时钟节拍。延时时间 0 表示不进行延时，函数将立即返回调用者。

函数形参:

INT16U ticks : 心跳节拍，一个心跳节拍的时间，延时时间从 0 到 65536。一个心跳节拍的时间有多长时间，它是由 os_cfg.h 中的 OS_TICKS_PER_SEC 来决定。假设 #define OS_TICKS_PER_SEC 1000 。意思是把 1 秒的时间分为 1000 个心跳节拍，所以一个心跳节拍的时间为 1ms。

返回值:

无。

示例:

```
void TaskX(void *pdata)
{
    while(1)
    {
        OSTimeDly(10); /* 任务延时 10 个时钟节拍 */
    }
}
```

4.3.2 OSTimeDlyHMSM()

函数原型:

```
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U ms)
```

函数功能:

将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。所以使用 OSTimeDlyHMSM() 比 OSTimeDly() 更方便。调用 OSTimeDlyHMSM() 后，如果延时时间不为 0，系统将立即进行任务调度。

函数参数:

INT8U hours: 为延时小时数，范围从 0-255。

INT8U minutes: 为延时分钟数，范围从 0-59。

INT8U seconds: 为延时秒数，范围从 0-59。

INT8U ms: 为延时毫秒数，范围从 0-999。

返回值:

OS_ERR_NONE: 函数调用成功。

OS_ERR_TIME_INVALID_MINUTES: 参数错误，分钟数大于 59。

OS_ERR_TIME_INVALID_SECONDS: 参数错误，秒数大于 59。

OS_ERR_TIME_INVALID_MS: 参数错误，毫秒数大于 999。

OS_ERR_TIME_ZERO_DLY: 四个参数全为 0。

OS_ERR_TIME_DLY_ISR : 错误操作, 在中断服务函数中使用。

说明:

OSTimeDlyHMSM (0, 0, 0, 0) 表示不进行延时操作, 而立即返回调用者。另外, 如果延时总时间超过 65535 个时钟节拍, 将不能用 OSTimeDlyResume () 函数终止延时并唤醒任务。

示例:

```
void TaskX(void *pdata)
{
    while(1)
    {
        OSTimeDlyHMSM(0, 0, 0, 100); /* 任务延时 100 毫秒 */
    }
}
```

4.3.3 OSTimeDlyResume()

函数原型:

INT8U OSTimeDlyResume (INT8U prio)

函数功能:

唤醒一个用 OSTimeDly () 或 OSTimeDlyHMSM () 函数延时的任务。

函数参数:

INT8U prio 为指定要唤醒任务的优先级。

返回值:

- 1) OS_ERR_NONE: 函数调用成功。
- 2) OS_PRIO_INVALID: 参数指定的优先级大于 OS_LOWEST_PRIO。
- 3) OS_TIME_NOT_DLY: 要唤醒的任务不在延时状态。
- 4) OS_TASK_NOT_EXIST: 指定的任务不存在。

说明:

用户不应该用 OSTimeDlyResume () 去唤醒一个设置了等待超时操作, 并且正在等待事件发生的任务。操作的结果是使该任务结束等待, 除非的确希望这么做。OSTimeDlyResume() 函数不能唤醒一个用 OSTimeDlyHMSM () 延时, 且延时时间总计超过 65535 个时钟节拍的 task。例如, 如果系统时钟为 100Hz, OSTimeDlyResume () 不能唤醒延时 OSTimeDlyHMSM (0, 10, 55, 350) 或更长时间的任务。

示例:

```
void TaskX(void *pdata)
{
    INT8U err;
    pdata = pdata;
    while(1)
    {
        err = OSTimeDlyResume(10); /* 唤醒优先级为 10 的任务 */
        if (err == OS_NO_ERR)
        {
            /* 任务被唤醒 */
        }
    }
}
```

4.4μC/OS- II 任务管理的编程思想

1. 任务的创建与删除

设计 3 个任务, 任务 Start 用于创建其他任务, 任务 LED 在执行 LED 灯的闪烁, 并记录任务执行的次数, 通过

串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键按下时删除任务 LED。

2. 任务的挂起与唤醒

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行 Led 灯的闪烁，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键 up 键按下时挂起任务 LED，当按键 down 键按下时挂起唤醒任务 LED。

3. 任务优先级的改变

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行 Led 灯的闪烁，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键按下改变任务 LED 的优先级，并在串口上显示改变成功还是失败。

4.5μC/OS- II 任务管理的编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、任务的创建

设计一个任务，在 main.c 中创建主任务 Start_Task，并且在里面执行 Led 灯的闪烁，并记录任务执行的次数，通过串口显示任务的执行次数。Start_Task 任务函数在 uc0s-task.c 中，代码如下：

```
//创建 start 任务
OSTaskCreate(
    Start_Task, //任务函数
    NULL,      //传递给任务函数的实参
    &Start_Task_Stk[Start_Task_Size-1], //任务堆栈栈顶的地址
    Start_Task_Pro //任务优先级
);
//start 任务执行的内容
void Start_Task(void *p_arg)
{
    INT8U cnt=0; //记录任务执行的次数
    p_arg=p_arg; //防止编译时出现警告，警告变量 p_arg 没有使用。
    while(1)
    {
        LED1=!LED1;
        cnt++; //任务次数自增
        printf("任务执行的次数为 : %d 次\r\n",cnt);
        OSTimeDly(1000); //延时 1000 个心跳节拍
    }
}
```

2、任务的挂起与唤醒

设计 3 个任务，主任务 Start 用于创建其他任务,任务 LED 在执行 Led 灯的闪烁，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键 up 键按下时挂起任务 LED，当按键 down 键按下时挂起唤醒任务 LED。代码如下：

```
//任务 1 的任务函数
void Led_Task(void *p_arg)
{
    INT8U cnt=0; //记录任务执行的次数
    p_arg=p_arg;
    while(1)
    {
        LED1=!LED1;
        cnt++; //任务次数自增
    }
}
```

```
printf("任务执行的次数为 : %d 次\r\n", cnt);
OSTimeDly(1000); //延时 1000 个心跳节拍
/*
if(cnt == 5)
{
    //OSTaskSuspend(Led_Task_Pro); //挂起任务优先级为 Led_Task_Pro 的任务
    OSTaskSuspend(OS_PRIO_SELF); //挂起自己
}*/

}

}

//任务 2 的任务函数
void Key_Task(void *p_arg)
{
    u8 key;
    p_arg = p_arg;
    while (1)
    {
        key = KEY_Scan(0); //按键扫描
        if(key == up_press) //up 这个按键下
        {
            OSTaskSuspend(Led_Task_Pro); //挂起任务优先级为 Led_Task_Pro 的任务
        }
        else if (key == down_press) //down 按键按下
        {
            OSTaskResume(Led_Task_Pro); //唤醒任务优先级为 Led_Task_Pro 的任务
        }
        OSTimeDly(100);
    }
}
```

第五章 $\mu\text{C}/\text{OS-II}$ 信号量管理

5.1 $\mu\text{C}/\text{OS-II}$ 信号量简介

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。

信号量用于控制对共享资源的保护，但是现在基本用来做任务同步用。

要想获取资源的任务必须执行“等待”操作，如果该资源对应的信号量有效值大于 1，则任务可以获得该资源，任务继续运行。如果该信号量的有效值为 0，则任务加入等待信号量的任务表中。如果等待时间超过某一个设定值，该信号量仍然没有被释放掉，则等待信号量的任务就进入就绪态，如果将等待时间设置为 0 的话任务就将一直等待该信号量

信号量通常分为两种：二进制信号量和计数型信号量。

二进制信号量只能取 0 和 1 两个值，计数型信号量的信号量值大于 1，计数型信号量的范围为 0~65535。

二值信号量用于那些一次只能一个任务使用的资源，比如 I/O 设备，打印机等，计数型信号量用于某些资源可以同时被几个任务所使用，比如一个缓存池有 10 个缓存块，那么同时最多可以支持 10 个任务来使用内存池。

5.2 $\mu\text{C}/\text{OS-II}$ 信号量相关函数

5.2.1 OSSemCreate()

函数原型：

```
OS_EVENT *OSSemCreate (INT16U cnt)
```

函数功能：

建立并初始化一个信号量。

函数参数：

INT16U cnt：建立的信号量的初始值，可以取 0 到 65535 之间的任何值。

返回值：

(void *)0：空指针，表示没有可用的事件控制块。

非(void *)0：指向分配给所建立的消息邮箱的事件控制块的指针。

说明：

必须先建立信号量，然后才能使用，并且不能在中断中创建信号量。

示例：

```
OS_EVENT *DispSem;
int main(void)
{
    OSInit();           /* 初始化 OS */
    OS_CPU_SysTickInit(); /* 初始化 OS 时钟 */
    DispSem = OSSemCreate(1); /* 建立显示设备的信号量 */
    //用户代码
    OSTStart();         /* 启动多任务内核 */
}
```

5.2.2 OSSemAccept()

函数原型：

```
INT16U OSSemAccept (OS_EVENT *pevent)
```

函数功能：

无等待的查看是否有信号量，查看设备是否就绪或事件是否发生。不同于 OSSemPend() 函数，如果设备没有就绪，OSSemAccept() 函数并不挂起任务。中断可以调用该函数来查询信号量。

函数参数:

OS_EVENT * pevent 是指向需要查询的设备的信号量。当建立信号量时, 该指针返回到用户程序。(参考 **OSSemCreate ()** 函数)。

返回值:

大于 0 : 说明设备就绪, 这个值被返回调用者, 设备信号量的值减一。

等于 0 : 说明设备没有就绪。

说明:

必须先建立信号量, 然后才能使用。

示例:

```
OS_EVENT *DispSem;
void Task (void *pdata)
{
    INT16U value;
    pdata = pdata;
    while(1)
    {
        value = OSSemAccept(DispSem); /*查看设备是否就绪或事件是否发生 */
        if (value > 0)
        {
            /* 就绪, 执行处理代码 */
        }
    }
}
```

5.2.3 OSSemPend ()

函数原型

```
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
```

函数功能

挂起任务等待信号量。此函数用于任务试图取得设备的使用权, 任务需要和其他任务或中断同步, 任务需要等待特定事件的发生的场合。如果任务调用 **OSSemPend()**函数时, 信号量的值大于零, **OSSemPend()**函数递减该值。如果调用时信号量等于零, **OSSemPend()**函数将任务加入该信号量的等待队列。**OSSemPend()**函数挂起当前任务直到其他的任务或中断置起信号量(把信号值设置为>0 值) 或 超出等待的预期时间 **timeout**。如果在预期的时钟节拍内信号量被置起, **uc/OS-II** 默认最高优先级的任务取得信号量恢复执行。一个被 **OSTaskSuspend()** 函数挂起的任务也可以接受信号量, 但这个任务将一直保持挂起状态直到通过调用 **OSTaskResume()**函数恢复任务的运行。

函数参数:

OS_EVENT *pevent : 指向信号量的指针。该指针的值在建立该信号量时可以得到。(参考 **OSSemCreate ()** 函数)。

INT16U timeout : 允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的信号量时恢复运行状态。如果该值为零表示任务将持续的等待信号量。最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的, 可能存在一个时钟节拍的误差, 因为只有在一个时钟节拍结束后才会减少定义的等待超时时钟节拍。

INT8U *perr : 指向包含错误码的变量的指针。**OSSemPend ()** 函数返回的错误码可能为下述几种:

- 1) **OS_ERR_NONE** :信号量不为 0, 任务得到信号量, 可以运行。
- 2) **OS_ERR_TIMEOUT** :信号量未在指定的超时周期内置起。
- 3) **OS_ERR_PEND_ABORT** :等待中止。是由于另一个任务或 ISR 通过调用 **OSSemPendAbort ()**。
- 4) **OS_ERR_EVENT_TYPE pevent** :不是指向一个信号量。
- 5) **OS_ERR_PEND_LOCKED** :在调用本函数时, 调度器被锁定。
- 6) **OS_ERR_PEND_ISR** :尝试从一个 ISR (中断程序) 调用 **OSSemPend ()**, 不允许。

7) `OS_ERR_PEVENT_NULL`: `pevent` 传入的是一个空指针 `NULL`。这个选项依赖于 `OS_ARG_CHK_EN` 这个宏, 在 `os_cfg.h` 中有定义, 要检测参数是否空, 则需要打开 `OS_ARG_CHK_EN` 这个宏。

返回值:

无。

说明:

必须先建立信号量, 然后才能使用。

示例:

```
OS_EVENT *DispSem;
void DispTask(void *pdata)
{
    INT8U err;
    pdata = pdata;
    while(1)
    {
        //用户代码
        OSSemPend(DispSem, 0, &err);
        /*只有信号量置起, 该任务才能执行*/
        //用户代码
    }
}
```

5.2.4 OSSemPost()

函数原型:

```
INT8U  OSSemPost (OS_EVENT *pevent)
```

函数功能:

发出一个信号量, 把信号值加 1。如果指定的信号量是零或大于零, `OSSemPost()` 函数递增该信号量。如果有任何任务在等待信号量, 最高优先级的任务将得到信号量并进入就绪状态, 任务调度函数将进行任务调度。

函数参数:

`OS_EVENT *pevent` : 指向信号量的指针。

返回值:

- 1) `OS_ERR_NONE` : 信号量成功置起。
- 2) `OS_ERR_SEM_OVF` : 信号计数溢出。
- 3) `OS_ERR_EVENT_TYPE pevent` : 不是指向一个消息邮箱。
- 4) `OS_ERR_PEVENT_NULL pevent` : 传入的是一个空指针。

说明:

必须先建立信号量, 然后才能使用。

示例:

```
OS_EVENT *DispSem;
void TaskX(void *pdata)
{
    INT8U err;
    pdata = pdata;
    while(1)
    {
        err = OSSemPost(DispSem);
        if (err == OS_ERR_NONE)
        {
```



```

        /* 信号量置起 */
    }
    else
    {
        /* 信号量溢出 */
    }
}
}

```

5.2.5 OSSemDel()

函数原型:

```
OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

删除一个信号量。使用这个函数必须小心，要检测 OSSemPend 函数的错误码情况。

函数参数:

OS_EVENT *pevent : 指向信号量的指针。

INT8U opt : 选择指定要删除信号量类型，有以下两种可选:

OS_DEL_NO_PEND: 只有当不存在任务删除挂起时才删除，否则不删除。

OS_DEL_ALWAYS: 直接删除。

INT8U *perr 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 信号量已经成功删除。
- 2) OS_ERR_DEL_ISR : 尝试从一个 ISR 删除一个信号量
- 3) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。
- 4) OS_ERR_EVENT_TYPE : pevent 不是指向一个信号量。
- 5) OS_ERR_INVALID_OPT : 传入选项参数(opt)有误。
- 6) OS_ERR_TASK_WAITING : 一个或多个任务正在等待信号量。

返回值:

(OS_EVENT *)0 : 空指针，说明信号量被成功删除。

其他 : 需要根据错误代码进行检查。

说明:

必须先建立信号量，然后才能使用。

示例:

```

OS_EVENT *DispSem;
void Task (void *p_arg)
{
    INT8U err;
    p_arg= p_arg;
    while(1)
    {
        //用户代码
        DispSem = OSSemDel(DispSem, OS_DEL_ALWAYS, &err);
        if (DispSem == (OS_EVENT *)0)
        {
            /* Semaphore has been deleted */
        }
        //用户代码
    }
}

```

}

5.2.6 OSSemQuery()

函数原型:

```
INT8U OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *p_sem_data)
```

函数功能:

获取一个信号量的相关信息。使用 OSSemQuery() 之前, 应用程序需要先创立类型为 OS_SEM_DATA 的数据结构, 用来保存从信号量的事件控制块中取得的数据。使用 OSSemQuery() 可以得知是否有, 以及有多少任务位于信号量的任务等待队列中 (通过查询 OS_EventTbl[] 域), 还可以获取信号量的标识号码。

函数参数:

OS_EVENT *pevent : 指向信号量的指针。

OS_SEM_DATA *p_sem_data : 指向 OS_SEM_DATA 数据结构的指针, 该数据结构定义如下:

```
typedef struct os_sem_data {
    INT16U  OSCnt;                      /* Semaphore count */
#ifdef OS_LOWEST_PRIO <= 63
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
    INT8U   OSEventGrp;                  /* Group corresponding to tasks waiting for event to occur */
#else
    INT16U  OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
    INT16U  OSEventGrp;                  /* Group corresponding to tasks waiting for event to occur */
#endif
} OS_SEM_DATA;
```

返回值:

- 1) OS_ERR_NONE : 获取成功。
- 2) OS_ERR_EVENT_TYPE : pevent 不是指向一个信号量。
- 3) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。
- 4) OS_ERR_PDATA_NULL : p_sem_data 传入的是一个空指针。

说明:

必须先建立信号量, 然后才能使用。

示例:

```
OS_EVENT *DispSem;
void Task(void *p_arg)
{
    OS_SEM_DATA sem_data;
    INT8U err;
    while(1)
    {
        //用户代码
        err = OSSemQuery(DispSem, &sem_data);
        if (err == OS_ERR_NONE)
        {
            /* Examine sem_data */
            //用户代码
        }
        //用户代码
    }
}
```

5.2.7 OSSemPendAbort()

函数原型:

```
INT8U OSSemPendAbort (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

使正在等待该信号量的任务取消挂起, 不再等待。取消任务挂起后, $\mu\text{C}/\text{OS-II}$ 会自动执行调度。若应用中不需其在取消任务后实现调度, 可在上述两种方式后或上操作宏 `OS_OPT_POST_NO_SCHED`。

函数参数:

`OS_EVENT *pevent` : 指向信号量的指针。

`INT8U opt` : 指定要中止信号量等待的类型。有以下两种选择:

`OS_PEND_OPT_NONE`: 只使正在等待该信号量的最高优先级任务取消挂起。

`OS_PEND_OPT_BROADCAST`: 使所有正在等待该信号量的任务取消挂起。

`INT8U *perr` : 指向错误代码变量的指针, 其返回值如下:

- 1) `OS_ERR_NONE` 没有任务在等待该信号量。
- 2) `OS_ERR_EVENT_TYPE pevent` 不是指向一个信号量。
- 3) `OS_ERR_PEND_ABORT` 至少一个任务等待的信号量已经退出等待。
- 4) `OS_ERR_PEVENT_NULL pevent` 传入的是一个空指针。

返回值:

大于 0 : 返回被本函数取消挂起的正在等待信号量的任务数。

等于 0 : 表示没有任务在等待该信号量, 因此该功能没作用。

说明:

必须先建立信号量, 然后才能使用。

示例:

```
OS_EVENT * psem;
void CommTaskRx (void *p_arg)
{
    INT8U err,ret;
    (void)p_arg;
    while(1)
    {
        //用户代码
        ret = OSSemPendAbort (psem, OS_PEND_OPT_NONE, &err);
        if(ret == 0)
        {
            //用户代码
        }
        else
        {
            //用户代码
        }
    }
}
```

5.2.8 OSSemSet()

函数原型:

```
void OSSemSet (OS_EVENT *pevent, INT16U cnt, INT8U *perr)
```

函数功能:

改变当前信号量的计数值。当信号量被用于表示某事件发生了多少次的情况下会使用。

函数参数:

OS_EVENT *pevent : 指向信号量的指针。

INT16U cnt : 期望的数值。

INT8U *perr : 指向错误代码变量的指针, 其返回值如下:

- 1) OS_ERR_NONE 计数值改变成功。
- 2) OS_ERR_EVENT_TYPE pevent 不是一个指向信号量的指针。
- 3) OS_ERR_PEVENT_NULL pevent 传入的是一个空指针。
- 4) OS_ERR_TASK_WAITING 存放任务在等待信号量。

返回值:

无。

说明:

- 1) 必须先建立信号量, 然后使用。
- 2) 这个函数只能修改当信号量>0 的情况, 也就是没有任务等待信号量的情况。

示例:

```
OS_EVENT *SignalSem;
void Task (void *p_arg)
{
    INT8U err;
    p_arg= p_arg;
    while(1)
    {
        OSSemSet(SignalSem, 0, &err); /* Reset the semaphore count */
        //用户代码
    }
}
```

5.3 μ C/OS- II 信号量管理编程思想

1. 任务的创建与删除

设计 3 个任务, 任务 Start 用于创建其他任务, 任务 LED 在执行 Led 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 KEY 功能是扫描按键, 当按键按下时删除信号量。

2. 任务的等待与发送

设计 3 个任务, 任务 Start 用于创建其他任务, 任务 LED 在等待获取信号量执行 Led 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 KEY 功能是扫描按键, 当按键按下时发送一个信号量。

3. 任务的检查

设计 3 个任务, 任务 Start 用于创建其他任务, 任务 LED 在等待获取信号量执行 Led 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 KEY 功能是扫描按键, 当按键按下时获取信号量的相关信息。

5.4 μ C/OS- II 信号量管理编程示例

本实验部分源码如下, 完整的工程详见工程示例代码:

1、在 uc0s-task.c 中首先定义一个信号量, 如下:

```
OS_EVENT *key_sem; //信号量事件控制块
```

2、在 main.c 中创建任务 Start_Task(), 然后在 uc0s-task.c 开始任务 Start_Task()中调用 OSSemCreate()函数创建一个信号量, 代码如下:

```
void Start_Task(void *p_arg)
{
    p_arg=p_arg;
    key_sem=OSSemCreate(5); //创建一个信号量, 信号量的值为 5.
    if(key_sem != 0)
    {
```

```
printf("成功创建一个信号量\r\n");
}
OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
while (1)
{
    OSTimeDly(1000);
}
}
```

3、任务的等待与发送

在 uc0s-task.c 中设计 3 个任务,任务 Start 用于创建其他任务,任务 LED 在等待获取信号量执行 Led 灯的闪烁,并记录任务执行的次数,通过串口显示任务执行的次数。任务 KEY 功能是扫描按键,当按键按下时发送一个信号量。

//任务 1 的任务函数

```
void Led_Task(void *p_arg)
{
    INT8U cnt=0;        //记录任任务得到信号量的次数
    INT8U cnt_buf[10];
    INT8U perr;
    p_arg=p_arg;
    while (1)
    {
        OSSemPend(key_sem,0,&perr);////无阻塞的等待一个信号量
        if(perr == OS_ERR_NONE) //得到信号量
        {
            LED1=!LED1;
            cnt++;          //任务得到信号量次数自增
            printf("任务得到信号量的次数为 : %d 次\r\n",cnt);
            //LCD 显示
            sprintf((char*)cnt_buf,"get sem cnt : %d",cnt);
            LCD_Show_String(30,130,200,16,BLUE,WHITE,cnt_buf);
        }
        printf("hello\r\n");
        OSTimeDly(500);
    }
}
```

//任务 2 的任务函数

```
void Key_Task(void *p_arg)
{
    INT8U res;
    p_arg=p_arg;
    while (1)
    {
        if(KEY_Scan(0))
        {
            res=OSSemPost(key_sem);//发送一个信号量
            if(res == OS_ERR_NONE)
```

```
{  
    printf("成功发送信号量\r\n");  
    LCD_Show_String(30,150,200,16,BLUE,WHITE,"post sem OK");  
}  
}  
    OSTimeDly(10);  
}  
}
```

第六章 $\mu\text{C}/\text{OS-II}$ 互斥信号量管理

6.1 $\mu\text{C}/\text{OS-II}$ 互斥信号量简介

6.1.1 互斥型信号量的理解

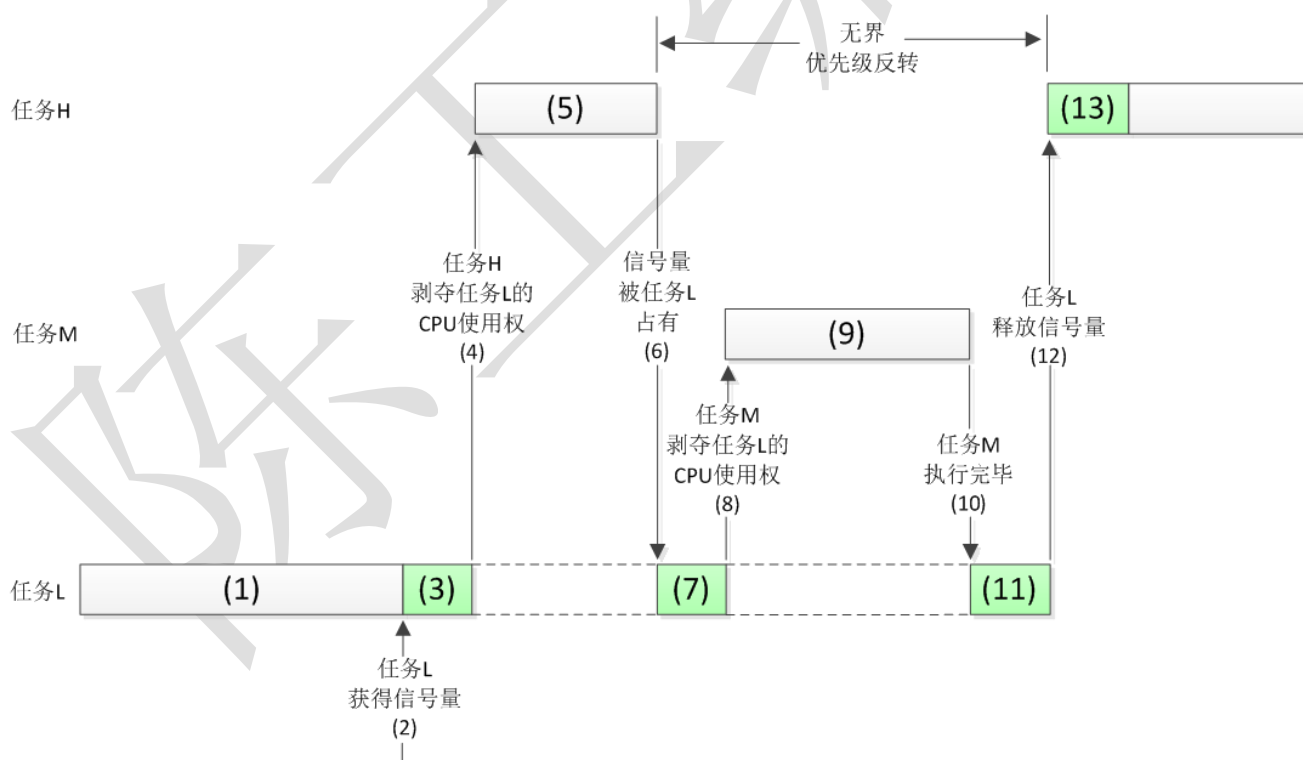
互斥型信号量首先是二值信号量,实现对共享资源的独占式处理,其次互斥型信号量可以在应用程序代码中用于降解优先级的反转问题,这个是它和普通信号量的最本质的区别。

6.1.2 互斥信号量的组成

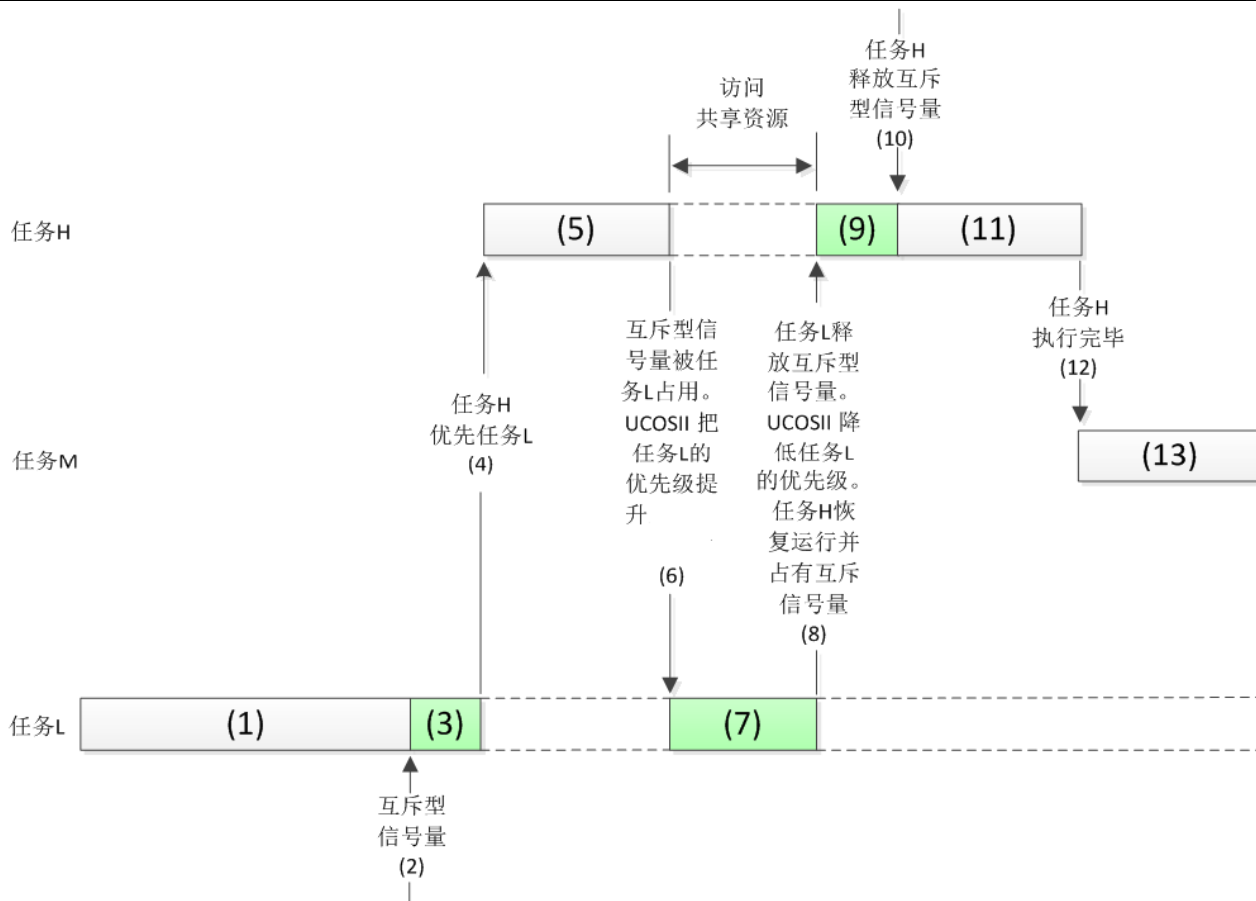
- ◆一个标志,指示 mutex 是否可以使用(0 或 1)
- ◆一个优先级,一旦高优先级的任务需要这个 mutex,赋予给占有 mutex 的任务。
- ◆一个等待该 mutex 的任务列表

6.1.3 优先级反转的问题

假设现在有三个任务分别是 Task1 (任务 H), Task2 (任务 M), Task3 (任务 L), 优先级从大到小。程序在运行过程中, Task1 和 Task2 处于挂起 (pend) 的状态, 等待某个事件的发生。这样优先级最低的 Task3 首先申请到了 mutex 并开始同共享资源打交道, 过了一会, Task1 等待的事件出现了, 那么此时 Task1 就需要使用共享资源了, 于是申请 mutex (OSMutexPend)。在这种情况下, OSMutexPend 注意到高优先级的任务要使用这个共享资源, 于是将 Task3 的优先级升到比 Task1 更高的级别, 并强制任务调度回到 Task3。而 Task1 只好继续挂起, 等待共享资源被释放。同时 Task3 继续运行, 直到 Task3 调用 OSMutexPost(), 释放 Mutex。在调用 OSMutexPost() 时, OSMutexPost() 会将 Task3 的优先级恢复到原来的水平, 同时还会注意到有个高优先级的任务 Task1 需要这个 Mutex, 于是将这个 Mutex 交给 Task1, 并做任务切换。



为了避免优先级反转这个问题, $\mu\text{C}/\text{OS-II}$ 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。



6.2 μ C/OS- II 互斥信号量相关函数

6.2.1 OSMutexCreate ()

函数原型:

```
OS_EVENT *OSMutexCreate (INT8U prio, INT8U *perr)
```

函数功能:

建立并初始化一个互斥型信号量。互斥型信号量的使用与信号量的使用差不多，但还需要一个较高的空闲优先级，这个级别要比使用这个互斥型信号量的所有任务优先级都高（数字更小）。

函数参数:

INT8U prio: 较高的空闲优先级，用于任务提权。

INT8U *perr : 指向一个用于保存错误代码的变量。可能为以下几种:

- 1) OS_ERR_NONE: 创建成功。
- 2) OS_ERR_PEVENT_NULL: pevent 是一个空指针。
- 3) OS_ERR_PRIO_EXIST: 指定的优先级已有任务存在。
- 4) OS_ERR_PEND_ISR: 从 ISR 调用 OSMutexCreate (), 不允许。
- 5) OS_ERR_PRIO_INVALID: 指定的优先级比 OS_LOWEST_PRIO(系统最低优先级)高。

返回值:

等于(void *)0 : 没有事件控制块可用，返回 NULL 指针。

不等于(void *)0: 一个指向分配给互斥事件控制块。

说明:

1) 在使用之前必须先创建一互斥。

2) 您必须确保 PRIO 具有比任何使用互斥访问资源的任务更高的优先级。例如，如果优先级 20， 25 的三个任务，和 30 将要使用互斥，那么 PRIO 的值必须比 20 低， 同时指定的值必须没有任务占用。

示例:

```
OS_EVENT *DispMutex;
int main (void)
```



```
{
    INT8U err;
    //用户代码
    OSInit(); /* Initialize  $\mu$ C/OS-II */
    //用户代码
    DispMutex = OSMutexCreate(18, &err); /* Create Display Mutex */
    //用户代码
    OSStart(); /* Start Multitasking */
}
```

6.2.2 OSMutexAccept()

函数原型:

```
BOOLEAN OSMutexAccept (OS_EVENT *pevent, INT8U *perr)
```

函数功能:

无等待请求互斥型信号量。

函数参数:

OS_EVENT *pevent: 指向要请求的互斥信号量的指针。当建立互斥信号量时，该指针返回到用户程序。

INT8U *perr: 指向一个用于保存错误代码的变量。可能为以下几种:

- 1) **OS_ERR_NONE:** 请求成功。
- 2) **OS_ERR_EVENT_TYPE:** pevent 不是指向一个互斥信号量。
- 3) **OS_ERR_PEVENT_NULL:** pevent 是一个空指针。
- 4) **OS_ERR_PEND_ISR:** 从 ISR 调用 OSMutexAccept(), 不允许。
- 5) **OS_ERR_PIP_LOWER:** 拥有互斥任务的优先级比设置 PIP 优先级高。

返回值:

OS_TRUE: 互斥信号量可用。

OS_FALSE: 互斥信号量被另一个任务拥有，不可用。

说明:

- 1) 在使用之前必须先创建一互斥信号量。
- 2) 此功能不能被 ISR 调用。
- 3) 通过 OSMutexAccept()成功请求互斥量后,当共享资源使用完成时必须调用 OSMutexPost()来释放互斥锁。

示例:

```
OS_EVENT *DispMutex;
void Task (void *p_arg)
{
    INT8U err;
    BOOLEAN test;
    p_arg=p_arg;
    while(1)
    {
        test = OSMutexAccept(DispMutex, &err);
        if(test == OS_TRUE)
        {
            /* 请求互斥信号量成功 */
        }
        else
        {
            /*请求互斥信号量失败*/
        }
    }
}
```

```
}  
//...用户代码  
}  
}
```

6.2.3 OSMutexPend ()

函数原型:

```
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
```

函数功能:

用于任务试图取得设备的使用权, 任务需要和其他任务或中断同步, 任务需要等待特定事件的发生的场合。如果任务调用 OSMutexPend()函数时, 如果互斥信号量被其他任务持有, OSMutexPend()函数挂起当前任务直到其他的任务或中断置起信号量或超出等待的预期时间。并且, 该互斥信号量的任务的持有者会把自己的优先级提升到一个比较高的优先级 (在创建互斥量时指定), 来保证他不被其他任务中止, 来确保在等待该信号的高优先级任务能尽快得到互斥信号。如果在预期的时钟节拍内信号量被置起, uC/OS-II 默认最高优先级的任务取得信号量恢复执行。一个被 OSTaskSuspend() 函数挂起的任务也可以接受互斥信号量, 但这个任务将一直保持挂起状态直到通过调用 OSTaskResume()函数恢复任务的运行。

函数参数:

OS_EVENT *pevent: 指向要请求的互斥信号量的指针。当建立互斥信号量时, 该指针返回到用户程序。

INT16U timeout: 任务等待超时周期, 为 0 时表示无限等待; 其它, 递减到 0 时任务恢复执行。

INT8U *perr: 指向一个用于保存错误代码的变量。可能为以下几种:

- 1) OS_ERR_NONE: 请求成功。
- 2) OS_ERR_TIMEOUT: 互斥量不在指定的超时时间内可用。
- 3) OS_ERR_PEND_ABORT: OSMutexPend()被另一个任务中止。
- 4) OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。
- 5) OS_ERR_PEVENT_NULL: pevent 是一个空指针。
- 6) OS_ERR_PEND_ISR: 从 ISR 调用 OSMutexPend(), 不允许。
- 7) OS_ERR_PIP_LOWER: 拥有互斥任务的优先级比设置 PIP 优先级高。

返回值:

无。

说明:

必须先建立互斥信号量, 然后再使用。不允许从中断调用该函数。

示例:

```
OS_EVENT *DispMutex;  
void DispTask (void *p_arg)  
{  
    INT8U err;  
    p_arg= p_arg;  
    while(1)  
    {  
        //...用户代码  
        OSMutexPend(DispMutex, 0, &err);/*申请互斥信号量*/  
        //...用户代码  
        /*共享设备 (资源) 代码; */  
        //...用户代码  
        OSMutexPost(DispMutex);/*释放互斥信号量*/  
        //...用户代码
```

```
}  
}
```

6.2.4 OSMutexPost ()

函数原型:

```
INT8U OSMutexPost (OS_EVENT *pevent)
```

函数功能:

释放一个互斥型信号量。

函数参数:

OS_EVENT *pevent: 指向一个互斥型信号量的指针。

返回值:

- 1) OS_ERR_NONE: 指定互斥信号量被释放。
- 2) OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。
- 3) OS_ERR_PEVENT_NULL: pevent 传入的是一个空指针。
- 4) OS_ERR_PEND_ISR: 从 ISR 调用 OSMutexPend (), 不允许。
- 5) OS_ERR_PIP_LOWER: 拥有互斥任务的优先级比设置 PIP 优先级高。
- 6) OS_ERR_NOT_MUTEX_OWNER: 互斥信号量并没有被占用。

说明:

必须先建立互斥信号量, 然后再使用。不允许从中断调用该函数。

示例:

```
OS_EVENT *DispMutex;  
void TaskX (void *p_arg)  
{  
    INT8U err;  
    p_arg= p_arg;  
    while(1)  
    {  
        //...用户代码  
        err = OSMutexPost(DispMutex);  
        if(err == OS_ERR_NONE)  
        {  
            /*释放互斥信号量成功*/  
        }  
        //...用户代码  
    }  
}
```

6.2.5 OSMutexDel ()

函数原型:

```
OS_EVENT *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

用来删除一个互斥信号量。使用这个函数是很危险的, 因为多个任务可能是依赖于互斥信号量而运行的。所以在删除互斥信号量之前, 必须先删除所有访问互斥信号量的任务。

函数参数:

OS_EVENT *pevent: 指向要请求的互斥信号量的指针。当建立互斥信号量时, 该指针返回到用户程序。

INT8U opt: 选择指定要删除消息邮箱类型:

- 1) OS_DEL_NO_PEND: 只有当不存在悬而未决的任务删除时才删除, 否则不删除。

2) OS_DEL_ALWAYS: 直接删除。

INT8U *perr: 指向一个用于保存错误代码的变量。可能为以下几种:

- 1) OS_ERR_NONE: 互斥信号量删除成功。
- 2) OS_ERR_DEL_ISR: 试图从 ISR 删除一个互斥信号量。
- 3) OS_ERR_INVALID_OPT: 指定操作参数有误。
- 4) OS_ERR_TASK_WAITING: 一个或多个任务正在等待互斥信号量同时指定了 OS_DEL_NO_PEND。
- 5) OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。
- 6) OS_ERR_PEVENT_NULL: pevent 是一个空指针。

返回值:

(OS_EVENT *)0 : 如果互斥锁被成功删除返回 NULL 指针。

其他 : 删除失败, 需要检查错误代码以确定原因。

说明:

无。

示例:

```
OS_EVENT *DispMutex;
void Task (void *p_arg)
{
    INT8U err;
    p_arg= p_arg;
    while (1)
    {
        //...用户代码
        DispMutex = OSMutexDel(DispMutex, OS_DEL_ALWAYS, &err);
        if (DispMutex == (OS_EVENT *)0)
        {
            /* 互斥信号量删除成功*/
        }
        //...用户代码
    }
}
```

6.2.6 OSMutexQuery ()

函数原型:

```
INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *p_mutex_data)
```

函数功能:

获取一个互斥信号量的相关信息。

函数参数:

OS_EVENT *pevent: 指向一个互斥型信号量的指针。

OS_MUTEX_DATA *p_mutex_data: 存放查询到的状态信息, 其结构如下。

```
typedef struct os_mutex_data {
    #if OS_LOWEST_PRIO <= 63
        INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
        INT8U   OSEventGrp; /* Group corresponding to tasks waiting for event to occur */
    #else
        INT16U  OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */
        INT16U  OSEventGrp; /* Group corresponding to tasks waiting for event to occur */
    #endif
}
```

```

    BOOLEAN OSValue;                /* Mutex value (OS_FALSE = used, OS_TRUE = available) */
    INT8U   OSOwnerPrio;             /* Mutex owner's task priority or 0xFF if no owner */
    INT8U   OSMutexPIP;              /* Priority Inheritance Priority or 0xFF if no owner */
} OS_MUTEX_DATA;

```

返回值:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_EVENT_TYPE: pevent 不是指向一个互斥信号量。
- 3) OS_ERR_PEVENT_NULL: pevent 传入的是一个空指针。
- 4) OS_ERR_QUERY_ISR: 从 ISR 调用 OSMutexQuery (), 不允许。
- 5) OS_ERR_PDATA_NULL: p_mutex_data 传入的是一个空指针。

说明:

必须先建立互斥信号量, 然后再使用。不允许从中断调用该函数。

示例:

```

OS_EVENT *DispMutex;
void Task (void *p_arg)
{
    OS_MUTEX_DATA mutex_data;
    INT8U err;
    INT8U highest; /* Highest priority task waiting on mutex */
    INT8U x;
    INT8U y;
    p_arg= p_arg;
    while(1)
    {
        //...用户代码
        err = OSMutexQuery(DispMutex, &mutex_data);
        if (err == OS_ERR_NONE)
        {
            /* Examine Mutex data */
        }
        //...用户代码
    }
}

```

6.3 μ C/OS-II 互斥信号量编程思想

1. 任务的创建与删除

设计 4 个任务, 任务 Start 用于创建其他任务, 任务 LED 在执行 Led1 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 LED1 在执行 Led2 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 KEY 功能是扫描按键, 当按键按下时删除互斥信号量。

2. 任务的等待与发送

设计 4 个任务, 任务 Start 用于创建其他任务, 任务 LED 在等待获取互斥信号量执行 Led1 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 LED1 在等待获取互斥信号量执行 Led2 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 KEY 功能是扫描按键。

3. 任务的检查

设计 4 个任务, 任务 Start 用于创建其他任务, 任务 LED 在等待获取互斥信号量执行 Led1 灯的闪烁, 并记录任务执行的次数, 通过串口显示任务执行的次数。任务 LED1 在等待获取互斥信号量执行 Led2 灯的闪烁, 并记录

任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键按下时获取互斥信号量的相关信息。

6.4 μ C/OS- II 互斥信号量编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、在 uc0s-task.c 中首先定义一个互斥信号量，如下：

```
OS_EVENT *key_mutex; //互斥信号量事件控制块
```

2、在 main.c 中创建任务 Start_Task(), 然后在 uc0s-task.c 开始任务 Start_Task()中调用 OSMutexCreate()函数创建一个互斥信号量，代码如下：

```
void Start_Task(void *p_arg)
{
    INT8U perr;
    p_arg=p_arg;
    key_mutex = OSMutexCreate(5,&perr);//创建一个互斥信号量
    if(perr == OS_ERR_NONE)
    {
        printf("成功创建一个互斥信号量\r\n");
    }
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Task_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Led2_Task,NULL, &Led2_Task_Stk[Led2_Task_Size - 1],Led2_Task_Pro); //创建一个名为 Led2_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Task_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
    while (1)
    {
        OSTimeDly(1000);
    }
}
```

3、任务的等待与发送

在 uc0s-task.c 中设计 4 个任务，任务 Start 用于创建其他任务,任务 LED 在等待获取互斥信号量执行 Led1 灯的闪烁，并记录任务执行的次数，通过串口显示任务执行的次数。任务 LED1 在等待获取互斥信号量执行 Led2 灯的闪烁，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键。

//任务 1 的代码函数

```
void Led_Task(void *p_arg)
{
    INT8U cnt=0; //记录任任务得到信号量的次数
    INT8U perr;
    INT8U cnt_buf[10];
    p_arg=p_arg;
    while (1)
    {
        OSMutexPend (key_mutex,0,&perr);//阻塞的等待一个互斥信号量
        if(perr == OS_ERR_NONE) //得到互斥信号量
        {
            LED1=!LED1;
            cnt++; //任务得到互斥信号量次数自增
            printf("LED1 任务得到互斥信号量的次数为 : %d 次\r\n",cnt);
            //LCD 显示
            sprintf((char*)cnt_buf,"led1 get mutex cnt : %d",cnt);
        }
    }
}
```

```
        LCD_Show_String(30,130,200,16,BLUE,WHITE,cnt_buf);
    }
    OSMutexPost(key_mutex);
    OSTimeDly(500);
}

//任务 2 的任务函数
void Led2_Task(void *p_arg)
{
    INT8U cnt=0;      //记录任任务得到信号量的次数
    INT8U perr;
    INT8U cnt_buf[50];
    p_arg=p_arg;
    while (1)
    {
        OSMutexPend (key_mutex,0,&perr);//阻塞的等待一个互斥信号量
        if(perr == OS_ERR_NONE) //得到互斥信号量
        {
            LED2=!LED2;
            cnt++;      //任务得到互斥信号量次数自增
            printf("LED2 任务得到互斥信号量的次数为 : %d 次\r\n",cnt);
            //LCD 显示
            sprintf((char*)cnt_buf,"led2 get mutex cnt : %d",cnt);
            LCD_Show_String(30,150,200,16,BLUE,WHITE,cnt_buf);
        }
        //OSMutexPost(key_mutex);
        OSTimeDly(500);
    }
}

//任务 3 的任务函数
void Key_Task(void *p_arg)
{
    u8 key ;
    p_arg=p_arg;
    while (1)
    {
        key=KEY_Scan(0); //按键扫描
        if(key==up_press)//up 这个按键下
        {

        }
        else if (key==down_press) //down 按键按下
        {

        }
        OSTimeDly(100);
    }
}
```

}

第七章 $\mu\text{C}/\text{OS-II}$ 消息邮箱管理

7.1 $\mu\text{C}/\text{OS-II}$ 消息邮箱简介

邮箱顾名思义就是用于通信的，日常生活中邮箱中的内容一般是信件。在操作系统中也是通过邮箱来管理任务间的通讯与同步，但是必须注意的是，系统中的邮箱中的内容并不是信件本身，而是指向消息内容的地址！这个指针是 `void` 类型的，可以指向任何类型的数据结构。因此，邮箱所发送的信息范围更宽，可以容纳下任何长度的数据。

一个任务或者中断服务程序有时候需要和另一个任务交流信息，这个就是消息传递的过程就叫做任务间通信，任务间的消息传递可以通过 2 种途径：一是通过全局变量，二是通过发布消息。

使用全局变量的时候每个任务或者中断服务程序都必须保证其对全局变量的独占访问。消息也可以通过消息队列作为中介发布给任务。

什么是消息？

消息包含一下几个部分：指向数据的指针，数据的长度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域或者甚至是一个函数。消息的内容必须一直保持可见性，可见性是指代表消息的变量必须在接收消息的任务代码范围内有效。这是因为发布的数据采用的是指针传递，也就是引用传递，并不是值传递。也就是说，发布的消息本身并不产生拷贝，我们可以使用动态内存分配的方式来给消息分配一个内存块，或者，也可以传递一个指向全局变量、全局数据结构、全局数组或者函数的指针。

消息邮箱是一个指针型变量。消息邮箱发送的是一个地址。

7.2 $\mu\text{C}/\text{OS-II}$ 消息邮箱相关函数

7.2.1 `OSMboxCreate()`

函数原型：

```
OS_EVENT *OSMboxCreate (void *pmsg)
```

函数功能：

建立并初始化一个消息邮箱。消息邮箱允许任务或中断向其他一个或几个任务发送消息。

函数参数：

`void *pmsg`：参数用来初始化建立的消息邮箱。如果该指针不为空，则建立的消息邮箱将含有消息。

返回值：

`(OS_EVENT *)0`：如果没有可用的事件控制块，返回空指针。

说明：

无。

示例：

```
OS_EVENT *CommMbox; /*定义邮箱指针*/
void main(void)
{
    //...用户代码
    //...用户代码
    OSInit(); /* 初始化  $\mu\text{C}/\text{OS-II}$  */
    //...用户代码
    //...用户代码
    CommMbox = OSMboxCreate((void *)0); /* 建立消息邮箱 */
    OSStart(); /* 启动多任务内核 */
}
```

7.2.2 OSMboxAccept()

函数原型:

```
void *OSMboxAccept (OS_EVENT *pevent)
```

函数功能:

函数查看指定的消息邮箱是否有需要的消息。不同于 OSMboxPend () 函数, 如果没有需要的消息, OSMboxAccept () 函数并不挂起任务。如果消息已经到达, 该消息被传递到用户任务并且从消息邮箱中清除。通常中断调用该函数, 因为中断不允许挂起等待消息。

函数参数:

OS_EVENT *pevent: 是指向需要查看的消息邮箱的指针。当建立消息邮箱时, 该指针返回到用户程序。

返回值:

等于(void *)0 : 如果消息邮箱没有消息或 pevent 传入为 NULL,或其他不是指向消息邮箱的指针, 返回 NULL 指针。

不等于(void *)0: 如果消息已经到达, 返回指向该消息的指针;

说明:

必须先建立消息邮箱, 然后使用。

示例:

```
OS_EVENT *CommMbox;
void Task (void *pdata)
{
    void *msg;
    pdata = pdata;
    while(1)
    {
        msg = OSMboxAccept(CommMbox); /* 检查消息邮箱是否有消息 */
        if (msg != (void *)0) /* 如果 msg 不为空则有消息 */
        {
            //...用户代码 /* 处理消息 */
        }
        else
        {
            //... 用户代码 /*没有消息 */
        }
    }
}
```

7.2.3 OSMboxPend()

函数原型:

```
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
```

函数功能:

用于任务等待消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量, 在不同的程序中消息的使用也可能不同。如果调用 OSMboxPend()函数时消息邮箱已经存在需要的消息, 那么该消息被返回给 OSMboxPend()的调用者, 消息邮箱中清除该消息。如果调用 OSMboxPend()函数时消息邮箱中没有需要的消息, OSMboxPend()函数挂起当前任务直到得到需要的消息或超出定义等待超时的时间。如果同时有多个任务等待同一个消息, μC/OS-II 默认最高优先级的任务取得消息并且任务恢复执行。一个由 OSTaskSuspend()函数挂起的任务也可以接受消息, 但这个任务将一直保持挂起状态直到通过调用 OSTaskResume()函数后恢复任务的运行。

函数参数:

OS_EVENT *pevent: 指向消息邮箱的指针。

INT16U timeout: 任务等待超时周期, 为 0 时表示无限等待; 其它, 递减到 0 时任务恢复执行。

INT8U *perr: 指向错误代码变量的指针, 其返回值如下:

- 1) **OS_ERR_NONE** : 接收到消息。
- 2) **OS_ERR_TIMEOUT**: 未在指定的超时周期内收到的消息。
- 3) **OS_ERR_PEND_ABORT** : 等待中止。是由于另一个任务或通过 ISR 调用 **OSMboxPendAbort()**。
- 4) **OS_ERR_EVENT_TYPE**: **pevent** 不是指向一个消息邮箱。
- 5) **OS_ERR_PEND_LOCKED** : 在调用本函数时, 调度器被锁定。
- 6) **OS_ERR_PEND_ISR** : 尝试从一个 ISR 调用 **OSMboxPend()**, 不允许。
- 7) **OS_ERR_PEVENT_NULL**: **pevent** 传入的是一个空指针。

返回值:

等于 **(void *)0** : 如果未在指定的超时期限内收到消息, 返回的消息是一个 **NULL** 指针, ***perr** 值为 **OS_ERR_TIMEOUT**。

不等于 **(void *)0**: 返回由任何一个任务或中断服务程序发送的消息, 以及 ***perr** 值为 **OS_ERR_NONE**。

说明:

- 1) 必须先创建邮箱在使用本函数。
- 2) 不允许在中断函数中调用本函数。

示例:

```
OS_EVENT *CommMbox;
void CommTask(void *p_arg)
{
    INT8U err;
    void *pmsg;
    p_arg= p_arg;
    while(1)
    {
        //...用户代码
        pmsg = OSMboxPend(CommMbox,0,&err);
        if (err == OS_ERR_NONE)
        {
            /*处理接收到的消息*/
        }
        else
        {
            /*接收消息失败, 检查相关错误代码以查明原因*/
        }
        //...用户代码
    }
}
```

7.2.4 OSMboxPost()

函数原型:

```
INT8U OSMboxPost(OS_EVENT *pevent, void *pmsg)
```

函数功能:

函数通过消息邮箱向任务发送消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同。如果消息邮箱中已经存在消息, 返回错误码说明消息邮箱已满, **OSMboxPost()**函数立即返回调用者, 消息也没有能够发到消息邮箱。如果有任何任务在等待消息邮箱的消息, 最高优先级的任务将得到这个消息。如果等待

消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将得到消息而恢复执行，也就是说，发生了一次任务切换。

函数参数：

OS_EVENT *pevent: 指向消息邮箱的指针。

void *pmsg: 是即将发送给任务的消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针。

返回值：

- 1) **OS_ERR_NONE** : 发送成功。
- 2) **OS_ERR_MBOX_FULL** : 邮箱已满。
- 3) **OS_ERR_EVENT_TYPE** : **pevent** 不是指向一个消息邮箱。
- 4) **OS_ERR_PEVENT_NULL** : **pevent** 传入的是一个空指针。
- 5) **OS_ERR_POST_NULL_PTR** : 发送的内容为 **NULL**(空)，不允许。

说明：

- 1) 必须先建立消息邮箱，然后使用。
- 2) 不允许传递一个空指针，因为这意味着消息邮箱为空。

示例：

```
OS_EVENT *CommMbox;
INT8U CommRxBuf[100];
void CommTaskRx (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        err = OSMboxPost(CommMbox, (void *)CommRxBuf);
        //...用户代码
    }
}
```

7.2.5 OSMboxDel()

函数原型：

OS_EVENT *OSMboxDel (OS_EVENT *pevent, INT8U opt, INT8U *perr)

函数功能：

用于删除一个消息邮箱。使用这个函数是很危险的，因为多个任务可能是依赖于消息邮箱而运行的。所以在删除消息邮箱之前，必须先删除所有访问消息邮箱的任务。使用这个函数，要特别小心，删除后，**OSMboxAccept()** 函数并不知道预期的邮箱已被删除。**OSMboxPend()** 函数申请消息也会失败，所以必须要检查它的返回值。

函数参数：

OS_EVENT *pevent: 指向消息邮箱的指针。

INT8U opt: 选择指定要删除消息邮箱类型：

- 1) **OS_DEL_NO_PEND**: 只有当不存任务等待该消息邮箱时才删除，否则不删除。
- 2) **OS_DEL_ALWAYS**: 不管有多少任务在等待该邮箱，都直接删除。当使用这一项删除消息邮箱，则所有等待该消息邮箱的任务都会被唤醒，并且返回 **OS_ERR_PEND_ABORT** 错误码。

INT8U *perr: 指向错误代码变量的指针，其返回值如下：

- 1) **OS_ERR_NONE** : 消息邮箱已经成功删除。
- 2) **OS_ERR_DEL_ISR** : 尝试从一个 **ISR** 删除一个消息邮箱。
- 3) **OS_ERR_PEVENT_NULL** : **pevent** 传入的是一个空指针。

- 4) OS_ERR_EVENT_TYPE : pevent 不是指向一个消息邮箱。
- 5) OS_ERR_INVALID_OPT : 传入选项参数(opt)有误。
- 6) OS_ERR_TASK_WAITING : 一个或多个任务正在等待消息。

返回值:

(OS_EVENT *)0 : 如果消息邮箱被成功删除, 则返回 NULL 指针。

其他 : 删除失败, 需要检查错误代码以确定原因。

说明:

无。

示例:

```
OS_EVENT *DispMbox;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while (1)
    {
        //...用户代码
        DispMbox = OSMboxDel(DispMbox, OS_DEL_ALWAYS, &err);
        if (DispMbox == (OS_EVENT *)0)
        {
            /* 消息邮箱被成功删除*/
        }
        //...用户代码
    }
}
```

7.2.6 OSMboxQuery()

函数原型:

```
INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *p_mbox_data)
```

函数功能:

用来取得消息邮箱的相关信息。用户程序必须分配一个 OS_MBOX_DATA 的数据结构, 该结构用来从消息邮箱的事件控制块接收数据。通过调用 OSMboxQuery()函数可以知道任务是否在等待消息以及有多少个任务在等待消息, 还可以检查消息邮箱现在的消息。

函数参数:

OS_EVENT *pevent: 指向消息邮箱的指针。

OS_MBOX_DATA *p_mbox_data: 是指向 OS_MBOX_DATA 数据结构的指针, 该数据结构包含下述成员:

```
void *OSMsg; /* 消息邮箱中消息*/
INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /*消息邮箱等待队列的任务链表*/
INT8U OSEventGrp;
```

返回值:

- 1) OS_ERR_NONE : 调用成功。
- 2) OS_ERR_EVENT_TYPE : pevent 不是指向一个消息邮箱。
- 3) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。
- 4) OS_ERR_PNAME_NULL : p_mbox_data 传入的是一个空指针。

说明:

无。

示例:

```
OS_EVENT *CommMbox;
void Task (void *p_arg)
{
    OS_MBOX_DATA mbox_data;
    INT8U err;
    p_arg= p_arg;
    while(1)
    {
        //...用户代码
        err = OSMboxQuery(CommMbox, &mbox_data);
        if (err == OS_ERR_NONE)
        {
            /*邮箱包含的相关信息获取成功*/
        }
    }
}
```

7.2.7 OSMboxPendAbort ()

函数原型:

```
INT8U OSMboxPendAbort (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

用来使在等待该邮箱消息的任务取消挂起。取消任务挂起后，uC/OS-II 会自动执行调度。若应用中不需其在取消任务后实现调度，可在上述两种方式后或上操作宏 OS_OPT_POST_NO_SCHED。

函数参数:

OS_EVENT *pevent: 指向消息邮箱的指针。

INT8U opt: 指定要中止消息邮箱等待的类型。

- 1) OS_PEND_OPT_NONE: 只中止最高优先级任务中在等待消息的邮箱。
- 2) OS_PEND_OPT_BROADCAST: 中止所有任务中在等待消息的邮箱。

INT8U *perr: 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 没有任务在等待该邮箱。
- 2) OS_ERR_EVENT_TYPE : pevent 不是指向一个消息邮箱。
- 3) OS_ERR_PEND_ABORT : 至少一个任务等待的消息邮箱已经退出等待。
- 4) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。

返回值:

等于 0: 表示没有任务在等待邮箱消息，因此该功能没作用。

不等于 0: 返回被本函数中止的正在等待该邮箱消息的任务数。

说明:

无。

示例:

```
OS_EVENT *CommMbox;
void CommTask(void *p_arg)
{
    INT8U err;
    INT8U nbr_tasks;
    p_arg= p_arg;
    while(1)
    {
```

```

//...用户代码
nbr_tasks = OSMboxPendAbort(CommMbox, OS_PEND_OPT_BROADCAST, &err);
if (err == OS_ERR_NONE)
{
    /* 没有任务在等待该邮箱 */
}
else if(err == OS_ERR_PEND_ABORT)
{
    /*所有在等待该消息邮箱的任务退出等待 */
}
//...用户代码
}
}

```

7.2.8 OSMboxPostOpt ()

函数原型:

```
INT8U OSMboxPostOpt (OS_EVENT *pevent, void *pmsg, INT8U opt)
```

函数功能:

功能跟 OSMboxPost()类似，不同的是有多个发送选项可选。

函数参数:

OS_EVENT *pevent: 指向消息邮箱的指针。

void *pmsg: 是即将发送给任务的消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针，因为这意味着消息邮箱为空。

INT8U opt: 发送选项，有以下几种:

- 1) OS_POST_OPT_NONE: 消息只发给等待邮箱消息的任务中优先级最高的任务。
- 2) OS_POST_OPT_BROADCAST: 消息发给所有等待邮箱消息的任务,所有等待该消息邮箱的任务都会获得消息，进入就绪状态。
- 3) OS_POST_OPT_NO_SCHED: 消息被提交到邮箱，但不马上调用调度器进行任务切换。选项是可以叠加的，例如： OS_POST_OPT_BROADCAST | OS_POST_OPT_NO_SCHED。

返回值:

- 1) OS_ERR_NONE : 发送成功。
- 2) OS_ERR_MBOX_FULL : 邮箱已满。
- 3) OS_ERR_EVENT_TYPE : pevent 不是指向一个消息邮箱。
- 4) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。
- 5) OS_ERR_POST_NULL_PTR : 发送的内容为 NULL(空)，不允许。

说明:

- 1) 必须先建立消息邮箱，然后使用。
- 2) 不允许传递一个空指针，因为这意味着消息邮箱为空。
- 3) 如果需要使用此功能，并希望减少代码空间，可以禁用代码生成 OSMboxPost()，因为 OSMboxPostOpt()可以模拟 OSMboxPost()。

示例:

```

OS_EVENT *CommMbox;
INT8U CommRxBuf[100];
void CommTaskRx (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
}

```

```
while(1)
{
    //...用户代码
    err = OSMboxPostOpt(CommMbox,
        (void *)CommRxBuf,
        OS_POST_OPT_BROADCAST);
    //...用户代码
}
```

7.3 μ C/OS- II 消息邮箱编程思想

1. 任务的创建与删除

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行 Led 灯亮灯，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键按下时删除任务 LED。

2. 任务的等待与发送

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息邮箱发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键按下时发送消息。

3. 任务的检查

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息邮箱发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键 up 键按下时发送消息，当按键 down 键按下时检查是否在等待消息以及有多少个任务在等待消息，还可以检查消息邮箱现在的消息。

7.4 μ C/OS- II 消息邮箱编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、在 uclos-task.c 中首先定义一个消息邮箱，如下：

```
OS_EVENT *key_mbox; //定义事件控制块指针 定义消息邮箱的指针
```

2、在 main.c 中创建任务 Start_Task(), 然后在 uclos-task.c 开始任务 Start_Task()中调用 OSMboxCreate()函数创建一个消息邮箱，代码如下：

```
void Start_Task(void *p_arg)
{
    p_arg=p_arg;
    //key_mbox=OSMboxCreate(NULL); //创建一个消息邮箱，邮箱中没有消息
    key_mbox=OSMboxCreate("Hello");//创建一个消息邮箱，邮箱中的消息为 LEDON
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
    while (1)
    {
        OSTimeDly(1000);
    }
}
```

3、任务的等待与发送

在 uclos-task.c 中设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息邮箱发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键按下时发送消息。

//任务 1 的代码函数

```
void Led_Task(void *pdata)
{
    INT8U perr;
    char *mbox_msg;
```



```
pdata=pdata;
while (1)
{
    //mbox_msg = OSMboxPend(key_mbox,100,&perr);//等待一个消息邮箱
    mbox_msg = OSMboxPend(key_mbox,0,&perr);//阻塞等待一个消息邮箱
    if(perr == OS_ERR_NONE)
    {
        if(strcmp(mbox_msg,"LEDON")==0)           //比较获取到的消息邮箱内容
        {
            printf("XYD\r\n");
        }
        else if(strcmp(mbox_msg,"LEDOFF")==0)      //比较获取到的消息邮箱内容
        {
            printf("welcome\r\n");
        }
    }
    printf("hello\r\n");
    OSTimeDly(500);
}
}

//任务 2 的任务函数
void Key_Task(void *pdata)
{
    INT8U err;
    pdata=pdata;
    while (1)
    {
        if(KEY_Scan(0))
        {
            err=OSMboxPost(key_mbox,"LEDOFF");
            if(err == OS_ERR_NONE)
            {
                printf("成功发送消息邮箱\r\n");
            }
        }
        OSTimeDly(10);
    }
}
```

第八章 μ C/OS- II 消息队列管理

8.1 μ C/OS- II 消息队列简介

消息的集合---存放多条消息。

FIFO

应用消息队列传输信息就类似于水管，一头流进（发送消息）一头流出（接收消息）。这也是它的缺点。 μ C/OS-II 补充了一个插队函数 OSQPostFront()。

消息队列是多个邮箱的数组，可以看做是个指针数组，任务之间可以按照一定顺序以指针定义的变量来传递，即是发送一个个指针给任务，任务获得指针，来处理指向的变量。这个方式有先进先出，先进后出。

一个邮箱只能传递一个指针，而队列可传递多个。

8.2 μ C/OS- II 消息队列相关函数

8.2.1 OSQCreate ()

函数原型：

```
OS_EVENT *OSQCreate(void **start, INT16U size)
```

函数功能：

函数建立一个消息队列。任务或中断可以通过消息队列向其他一个或多个任务发送消息。消息的含义是和具体的应用密切相关的。

函数参数：

void **start: 是消息内存区的基地址，消息内存区是一个指针数组。

INT16U size: 是消息内存区的大小。

返回值：

(OS_EVENT *)0 : 如果没有可用的事件控制块，返回空指针。

不等于 0: 返回一个指向消息队列事件控制块的指针。

说明：

无。

示例：

```
OS_EVENT *CommQ;  
void *CommMsg[10];  
void main (void)  
{  
    OSInit(); /* Initialize  $\mu$ C/OS-II */  
    //...用户代码  
    CommQ = OSQCreate(&CommMsg[0], 10); /* Create COMM Q */  
    //...用户代码  
    OSStart(); /* Start Multitasking */  
}
```

8.2.2 OSQAccept ()

函数原型：

```
void *OSQAccept(OS_EVENT *pevent, INT8U *perr)
```

函数功能：

函数检查消息队列中是否已经有需要的消息。不同于 OSQPend() 函数，如果没有需要的消息，OSQAccept() 函数并不挂起任务。如果消息已经到达，该消息被传递到用户任务。通常中断调用该函数，因为中断不允许挂起等待消息。

函数参数：

OS_EVENT *pevent: 是指向需要查看的消息队列的指针。当建立消息邮箱时，该指针返回到用户程序。

INT8U *perr: 指向一个用于保存错误代码的变量。可能为以下几种：

- 1) **OS_ERR_NONE:** 请求成功。
- 2) **OS_ERR_EVENT_TYPE:** **pevent** 不是指向一个消息队列。
- 3) **OS_ERR_PEVENT_NULL:** **pevent** 是一个空指针。
- 4) **OS_ERR_Q_EMPTY:** 队列不包含任何信息。

返回值：

等于(**void ***)0：如果消息队列没有消息，返回空指针。

不等于(**void ***)0：如果消息已经到达，返回指向该消息的指针。

说明：

必须先建立消息队列，然后使用。

示例：

```
OS_EVENT *CommQ;
void Task (void *p_arg)
{
    void *pmsg;
    p_arg=p_arg;
    while(1)
    {
        pmsg = OSQAccept(CommQ); /* Check queue for a message */
        if (pmsg != (void *)0)
        {
            . /* Message received, process */
        }
        else
        {
            . /* Message not received, do .. */
            . /* .. something else */
        }
        //...用户代码
    }
}
```

8.2.3 OSQPend ()

函数原型：

```
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
```

函数功能：

函数用于任务等待队列中的消息。消息通过中断或另外的任务发送给需要的任务。消息是一个以指针定义的变量，在不同的程序中消息的使用也可能不同。如果调用 **OSQPend()**函数时队列中已经存在需要的消息，那么该消息被返回给 **OSQPend()**函数的调用者，队列中清除该消息。如果调用 **OSQPend()**函数时队列中没有需要的消息，**OSQPend()**函数挂起当前任务直到得到需要的消息或超出定义的超时时间。如果同时有多个任务等待同一个消息，**uC/OS-ii** 默认最高优先级的任务取得消息并且任务恢复执行。一个由 **OSTaskSuspend()**函数挂起的任务也可以接受消息，但这个任务将一直保持挂起状态直到通过调用 **OSTaskResume()**函数恢复任务的运行。

函数参数：

OS_EVENT *pevent: 指向消息队列的指针。当建立消息队列时，该指针返回到用户程序。

INT16U timeout: 任务等待超时周期，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

INT8U *perr: 指向错误代码变量的指针，其返回值如下：

- 1) OS_ERR_NONE : 接收到消息。
- 2) OS_ERR_TIMEOUT: 未在指定的超时周期内收到的消息。
- 3) OS_ERR_PEND_ABORT : 等待中止。是由于另一个任务或通过 ISR 调用 OSQPendAbort()。
- 4) OS_ERR_EVENT_TYPE : pevent 不是指向一个消息邮箱。
- 5) OS_ERR_PEND_LOCKED : 在调用本函数时, 调度器被锁定。
- 6) OS_ERR_PEND_ISR : 尝试从一个 ISR 调用 OSMboxPend (), 不允许。
- 7) OS_ERR_PEVENT_NULL: pevent 传入的是一个空指针。

返回值:

无。

说明:

必须先创建消息队列后, 才能使用。

示例:

```
OS_EVENT *CommQ;  
void CommTask(void *p_arg)  
{  
    INT8U err;  
    void *pmsg;  
    p_arg=p_arg;  
    while(1)  
    {  
        //...用户代码  
        pmsg = OSQPend(CommQ, 100, &err);  
        if (err == OS_ERR_NONE)  
        {  
            /* Message received within 100 ticks! */  
        }  
        else  
        {  
            /* Message not received, must have timed out */  
        }  
        //...用户代码  
    }  
}
```

8.2.4 OSQPost ()

函数原型:

```
INT8U OSQPost (OS_EVENT *pevent, void *pmsg)
```

函数功能:

函数通过消息队列向任务发送消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同。如果队列中已经存满消息, 返回错误码;OSQPost()函数立即返回调用者, 消息也没有能够发到队列。如果有任何任务在等待队列中的消息, 最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高, 那么高优先级的任务将得到消息而恢复执行, 也就是说, 发生了一次任务切换。消息队列是先入先出(FIFO)机制的, 先进入队列的消息先被传递给任务。

函数参数:

OS_EVENT *pevent: 指向消息队列的指针。

void *pmsg: 是即将发送给任务的消息。消息是一个指针长度的变量, 在不同的程序中消息的使用也可能不同。不允许传递一个空指针。

返回值:

- 1) OS_ERR_NONE : 发送成功。
- 2) OS_ERR_Q_FULL : 消息队列已满。
- 3) OS_ERR_EVENT_TYPE: pevent 不是指向消息队列的指针。
- 4) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。

说明:

必须先建立消息队列, 然后使用。不允许传递一个空指针。

示例:

```
OS_EVENT *CommQ;
INT8U CommRxBuf[100];
void CommTaskRx(void *pdata)
{
    INT8U err;
    pdata = pdata;
    for (;;)
    {
        err = OSQPost(CommQ, (void *)&CommRxBuf[0]);
        if (err == OS_ERR_NONE)
        {
            /* 将消息放入消息队列 */
        }
        else
        {
            /* 消息队列已满 */
        }
    }
}
```

8.2.5 OSQDel()

函数原型:

```
OS_EVENT *OSQDel(OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

用于删除不再使用的消息队列, 使用该函数时需非常谨慎, 最好是先删除与该消息队列相关的所有任务, 或者事先解除那些任务与该消息队列间的联系, 不然会导致这些任务的失效。如果失效的任务是系统或应用中的关键任务, 后果将不堪设想, 因此, 使用该函数时应当非常慎重。

如果删除该队列, 所有的任务挂起的队列将准备就绪, 因此你必须小心应用程序中队列用于互斥的情况, 因为资源将不再由队列看守。

此调用可能会禁止中断很长一段时间。中断禁止时间成正比任务等待队列的数目。

使用这个函数, 如果选项是 OS_DEL_ALWAYS, OSQAccept()函数并不知道预期的邮箱已被删除。

OSQPend()函数申请消息也会失败返回 OS_ERR_PEND_ABORT, 所以必须要检查它的返回值。

函数参数:

OS_EVENT *pevent: 指向消息队列的指针。当建立消息队列时, 该指针返回到用户程序。

INT8U opt: 选择指定要删除消息邮箱类型:

- 1) OS_DEL_NO_PEND: 只有当不存在悬而未决的任务删除时才删除, 否则不删除。
- 2) OS_DEL_ALWAYS: 不管是否有任务等待消息都直接删除。

INT8U *perr: 指向一个用于保存错误代码的变量。可能为以下几种:

- 1) OS_ERR_NONE: 消息队列删除成功。

- 2) OS_ERR_DEL_ISR: 试图从 ISR 删除一个消息队列。
- 3) OS_ERR_INVALID_OPT: 指定操作参数有误。
- 4) OS_ERR_TASK_WAITING: 一个或多个任务正在等待消息队列同时指定了 OS_DEL_NO_PEND。
- 5) OS_ERR_EVENT_TYPE: pevent 不是指向一个消息队列。
- 6) OS_ERR_PEVENT_NULL: pevent 是一个空指针。

返回值:

(OS_EVENT *)0 : 如果消息邮箱被成功删除, 则返回 NULL 指针。

其他 : 删除失败, 需要检查错误代码以确定原因。

说明:

无。

示例:

```
OS_EVENT *DispQ;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while (1)
    {
        //...用户代码
        DispQ = OSQDel(DispQ, OS_DEL_ALWAYS, &err);
        if (DispQ == (OS_EVENT *)0)
        {
            /* 消息队列删除成功 */
        }
        //...用户代码
    }
}
```

8.2.6 OSQQuery ()

函数原型:

```
INT8U OSQQuery (OS_EVENT *pevent, OS_Q_DATA *p_q_data)
```

函数功能:

函数用来取得消息队列的相关信息。用户程序必须建立一个 OS_Q_DATA 的数据结构, 该结构用来保存从消息队列的事件控制块得到的数据。通过调用 OSQQuery () 函数可以知道任务是否在等待消息、有多少个任务在等待消息、队列中有多少消息以及消息队列可以容纳的消息数。 OSQQuery () 函数还可以得到即将被传递给任务的的信息。

函数参数:

OS_EVENT *pevent: 指向消息队列的指针。

OS_Q_DATA *p_q_data: 指向 OS_Q_DATA 的一个数据结构, 它包含下列字段:

void *OSMsg; /* Pointer to next message to be extracted from queue */

INT16U OSNMsgs; /* Number of messages in message queue */

INT16U OSQSize; /* Size of message queue */

OS_PRIO OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur */

OS_PRIO OSEventGrp; /* Group corresponding to tasks waiting for event to occur */

返回值:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_EVENT_TYPE: pevent 不是指向一个消息队列。

3) OS_ERR_PEVENT_NULL: pevent 是一个空指针。

4) OS_ERR_PDATA_NULL: p_q_data 是一个空指针。

说明:

必须先创建消息队列, 才能使用本函数。

示例:

```
OS_EVENT *CommQ;
void Task (void *p_arg)
{
    OS_Q_DATA qdata;
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        err = OSQQuery(CommQ, &qdata);
        if (err == OS_ERR_NONE)
        {
            /* 'qdata' can be examined! */
        }
        //...用户代码
    }
}
```

8.2.7 OSQFlush()

函数原型:

```
INT8U OSQFlush (OS_EVENT *pevent)
```

函数功能:

函数清空消息队列并且忽略发送往队列的所有消息。不管队列中是否有消息, 这个函数的执行时间都是相同的。

函数参数:

OS_EVENT *pevent: 指向消息队列的指针。当建立消息队列时, 该指针返回到用户程序。

返回值:

1) OS_ERR_NONE: 清空成功。

2) OS_ERR_EVENT_TYPE: pevent 不是指向一个消息队列。

3) OS_ERR_PEVENT_NULL: pevent 是一个空指针。

说明:

必须先创建消息队列后, 才能使用。

示例:

```
OS_EVENT *CommQ;
void main(void)
{
    INT8U err;
    OSInit(); /* Initialize µC/OS-II */
    //...用户代码
    err = OSQFlush(CommQ);
    //...用户代码
    OSStart(); /* Start Multitasking */
}
```

}

8.2.8 OSQPostFront()

函数原型:

```
INT8U OSQPostFront (OS_EVENT *pevent, void *pmsg)
```

函数功能:

函数通过消息队列向任务发送消息。OSQPostFront() 函数和 OSQPost() 函数非常相似, 不同之处在于 OSQPostFront() 函数将发送的消息插到消息队列的最前端。也就是说, OSQPostFront() 函数使得消息队列按照后入先出(LIFO)的方式工作, 而不是先入先出(FIFO)。如果队列中已经存满消息, 返回错误码。OSQPost() 函数立即返回调用者, 消息也没能发到队列。如果有任何任务在等待队列中的消息, 最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高, 那么高优先级的任务将得到消息而恢复执行, 也就是说, 发生了一次任务切换。

函数参数:

OS_EVENT *pevent: 指向消息队列的指针。

void *pmsg: 是即将发送给任务的消息。不允许传递一个空指针。

返回值:

- 1) OS_ERR_NONE : 发送成功。
- 2) OS_ERR_Q_FULL : 消息队列已满。
- 3) OS_ERR_EVENT_TYPE : pevent 不是指向消息队列的指针。
- 4) OS_ERR_PEVENT_NULL : pevent 传入的是一个空指针。

说明:

必须先建立消息队列, 然后使用。不允许传递一个空指针。

示例:

```
OS_EVENT *CommQ;
INT8U CommRxBuf[100];
void CommTaskRx(void *pdata)
{
    INT8U err;
    pdata = pdata;
    for (;;)
    {
        err = OSQPostFront (CommQ, (void *)&CommRxBuf[0]);
        if (err == OS_ERR_NONE)
        {
            /* 将消息放入消息队列 */
        }
        else
        {
            /* 消息队列已满 */
        }
    }
}
```

8.2.9 OSQPendAbort ()

函数原型:

```
INT8U OSQPendAbort (OS_EVENT *pevent, INT8U opt, INT8U *perr)
```

函数功能:

用来使正在等待该消息队列的任务取消挂起。取消任务挂起后, uC/OS-II 会自动执行调度。若应用中不需

其在取消任务后实现调度，可在上述两种方式后或上操作宏 `OS_OPT_POST_NO_SCHED`。

函数参数：

`OS_EVENT *pevent`：指向消息队列的指针。

`INT8U opt`：指定要中止消息邮箱等待的类型。

1) `OS_PEND_OPT_NONE`：只使正在等待该队列中消息的最高优先级任务取消挂起。

2) `OS_PEND_OPT_BROADCAST`：使所有在等待该队列消息的任务取消挂起。

`INT8U *perr`：

1) `OS_ERR_NONE`：没有任务在等待该消息队列。

2) `OS_ERR_EVENT_TYPE`：`pevent` 不是指向一个消息队列。

3) `OS_ERR_PEND_ABORT`：至少一个任务等待的消息队列已经退出等待。

4) `OS_ERR_PEVENT_NULL`：`pevent` 传入的是一个空指针。

返回值：

等于 0：表示没有任务在等待队列中的消息，因此该功能没作用。

不等于 0：返回被本函数取消挂起的正在等待队列中消息的任务数。

说明：

必须先创建消息队列后，才能使用。

示例：

```
OS_EVENT *CommQ;
void CommTask(void *p_arg)
{
    INT8U err;
    INT8U nbr_tasks;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        nbr_tasks = OSQPendAbort(CommQ, OS_PEND_OPT_BROADCAST, &err);
        if (err == OS_ERR_NONE)
        {
            /* 没有任务在等待该消息队列 */
        }
        else if(err == OS_ERR_PEND_ABORT)
        {
            /*所有在等待该消息队列的任务退出等待 */
        }
        //...用户代码
    }
}
```

8.2.10 OSMboxPostOpt ()

函数原型：

```
INT8U OSMboxPostOpt (OS_EVENT *pevent, void *pmsg, INT8U opt)
```

函数功能：

用于通过一个队列发送一个消息给一个任务。消息是一个指针大小可变，并且其用途是应用特定的。如果消息队列已满，错误代码返回，表示队列已满。 `OSQPostOpt()`然后立即返回到其调用者，并且消息不放置在队列中。如果有任何任务是在队列中等待消息， `OSQPostOpt()`中既可以发布消息的优先级最高的任务队列（`OPT` 设置为 `OS_POST_OPT_NONE`）或所有任务在队列中等待（`OPT` 设置为 `OS_POST_OPT_BROADCAST`）等。 `OSQPostOpt()`

实际上可以代替 `OSQPost()`和 `OSQPostFront()`，因为你通过选项参数指定的操作模式，选择。这样做可以让您减少所需的 $\mu C/OS-II$ 的代码空间。

函数参数：

`OS_EVENT *pevent`：指向消息邮箱的指针。

`void *pmsg`：是即将发送给任务的消息。消息是一个指针长度的变量，在不同的程序中消息的使用也可能不同。不允许传递一个空指针。

`INT8U opt`：发送选项参数，有以下几种：

- 1) `OS_POST_OPT_NONE`：发送成功。
- 2) `OS_POST_OPT_BROADCAST`：等待队列中的所有任务。
- 3) `OS_POST_OPT_NO_SCHED`：调度程序将不会被调用。

返回值：

- 1) `OS_ERR_NONE`：发送成功。
- 2) `OS_ERR_Q_FULL`：消息队列已满。
- 3) `OS_ERR_EVENT_TYPE`：`pevent` 不是指向消息队列的指针。
- 4) `OS_ERR_PEVENT_NULL`：`pevent` 传入的是一个空指针。
- 5) `OS_ERR_POST_NULL_PTR`：发送的内容为 `NULL`(空)，不允许。

说明：

- 1) 在使用之前必须先创建一队列。
- 2) 如果需要使用此功能，并希望减少代码空间，可以禁止代码生成 `OSQPost()`和 `OSQPostFront()`。在 `OS_CFG.H` 设置 `OS_Q_POST_EN` 为 0、`OS_Q_POST_FRONT_EN` 为 0，因为 `OSQPostOpt()`可以模拟这两种功能。
- 3) `OSQPostOpt()` 的执行时间取决于任务在等待队列中的数量，如果你发送选项选择 `OS_POST_OPT_BROADCAST`。
- 4) 当同时提交多个消息时，可先失能提交后的调度，待到最后一个消息被提交时，再使能调度。从而避免了需提交几个消息就得执行几次调度的尴尬局面，提高了系统的实时性。

示例：

```
OS_EVENT *CommQ;
INT8U CommRxBuf[100];
void CommRxTask (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        //...用户代码
        err = OSQPostOpt( CommQ,
                        (void *)&CommRxBuf[0],
                        OS_POST_OPT_BROADCAST);

        //...用户代码
        //...用户代码
    }
}
```

8.3 $\mu C/OS-II$ 消息队列编程思想

1. 任务的创建与删除

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行 Led 灯亮灯，并记录任务执行的次数，通过串口显示任务执行的次数。任务 KEY 功能是扫描按键，当按键按下时删除消息队列。

2. 任务的等待与发送

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息队列发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键按下时发送消息队列的消息。

3. 任务的检查

设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息队列发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键 down 键按下时发送消息，当按键 right 键按下时等待消息队列，当按键 left 键按下时，还可以检查消息队列的相关信息。

8.4 μ C/OS- II 消息队列编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、在 ucos-task.c 中首先定义一个消息队列，如下：

```
OS_EVENT *CommQ;           //定义事件控制块指针 定义消息邮箱的指针
//定义一个指针数组,用于存放消息队列的内容
void * CommMsg[5]={ "LED1ON", "LED2ON", "LED3ON", "LED4ON", "ALLOFF"};
```

2、在 main.c 中创建任务 Start_Task(), 然后在 ucos-task.c 开始任务 Start_Task()中调用 OSQCreate()函数创建一个消息队列，代码如下：

```
void Start_Task(void *p_arg)
{
    p_arg=p_arg;
    CommQ=OSQCreate(&CommMsg[0],5);//创建一个消息队列，里面可以存放 5 则消息的内容
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
    while (1)
    {
        //向消息队列中发送消息
        OSQPost(CommQ,CommMsg[0]); //发送第一则消息
        OSQPost(CommQ,CommMsg[1]); //发送第二则消息
        OSQPost(CommQ,CommMsg[2]); //发送第三则消息
        OSQPost(CommQ,CommMsg[3]); //发送第四则消息
        OSQPost(CommQ,CommMsg[4]); //发送第五则消息
        OSTaskSuspend(Start_Task_Pro);//把任务挂起
    }
}
```

3、任务的等待与发送

在 ucos-task.c 中设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待消息队列发送的消息，并记录任务执行的发送的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键按下时发送消息队列的消息。

//任务 1 的任务函数

```
void Led_Task(void *p_arg)
{
    INT8U err;
    char *q_msg;
    p_arg=p_arg;
    while (1)
    {
        q_msg = OSQPend( CommQ,0,&err);
        if(err == OS_ERR_NONE)
        {
```

```
LCD_Show_String(30,130,200,16,BLUE,WHITE,"post msg:");
LCD_Show_String(30+9*8,130,200,16,BLUE,WHITE,(u8*)q_msg);
printf("get post msg:%s\r\n",q_msg);
if(strcmp(q_msg,"LED1ON")==0)
{
    LED1=0;
}

else if(strcmp(q_msg,"LED2ON")==0)
{
    LED2=0;
}

else if(strcmp(q_msg,"LED3ON")==0)
{
    LED3=0;
}

else if(strcmp(q_msg,"LED4ON")==0)
{
    LED4=0;
}

else if(strcmp(q_msg,"ALLOFF")==0)
{
    LED1=1;
    LED2=1;
    LED3=1;
    LED4=1;
}
}
OSTimeDly (1000);
}
```

//任务 2 的任务函数

void Key_Task(void *p_arg)

```
{
    u8 key ;
    p_arg=p_arg;
    while (1)
    {
        key=KEY_Scan(0); //按键扫描
        switch(key)
        {
            case up_press:      OSQPost(CommQ ,"LED1ON");break;
            case down_press:    OSQPost(CommQ ,"LED2ON");break;
            case right_press:   OSQPost(CommQ ,"LED3ON");break;
```

```
        case left_press:      OSQPost(CommQ,"LED4ON");break;
        default:
            break;
    }
    OSTimeDly(100);
}
}
```

第九章 μ C/OS- II 事件标志组管理

9.1 μ C/OS- II 事件标志组简介

在信号量和互斥信号量的管理中，任务请求资源，如果资源未被占用就继续需运行，否则只能阻塞，等待资源释放事件的发生。这种事件是单一的事件，如果任务要等待多个事件的发生，或者多个事件中某一个事件的发生就可以继续运行，那么就应该采用事件标志组管理。事件标志组管理的条件组合可以是多个事件都发生（与关系），也可以是多个事件中有任何一个事件发生（或关系）。

有时候一个任务需要与多个事件同步（多个条件成立时才执行任务，如双击鼠标运行），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。

“或”同步：等待多个事件时，任何一个事件发生，任务都被同步，这个就称为“或”同步。

“与”同步：当所有的事件都发生时任务才被同步，这种同步机制被称为“与”同步。

事件标志组由 2 部分组成：一是保存各事件状态的标志位；二是等待这些标志位置位或清除的任务列表。可以用 8 位、16 位或 32 位的序列表示事件标志组，每一位表示一个事件的发生。要使系统支持事件标志组功能，需要在 OS_CFG.H 文件中打开 OS_FLAG_EN 选项。

9.2 μ C/OS- II 事件标志组相关函数

9.2.1 OSFlagCreate()

函数原型：

```
OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags, INT8U *perr)
```

函数功能：

用于创建和初始化一个事件标志组。

函数参数：

OS_FLAGS flags：包含在事件标志组中存储的初始值。

INT8U *perr：指向错误代码变量的指针，其返回值如下：

1) OS_ERR_NONE：事件标志组创建成功。

2) OS_ERR_CREATE_ISR：程序尝试从一个 ISR 创建事件标志组。

3) OS_ERR_FLAG_GRP_DEPLETED：没有更多可用的事件标志组，此时需要在 os_cfg.H 增加 os_max_flags 数量以解决本错误。

返回值：

(OS_FLAG_GRP *)0：如果没有可用的事件标志控制块，返回空指针。

不等于 0：指向分配给所建立的事件标志组的事件标志控制块的指针。

说明：

无。

示例：

```
OS_FLAG_GRP *EngineStatus; /*创建事件标志组指针*/
```

```
void main (void)
```

```
{
```

```
    INT8U err; /*存放错误代码变量*/
```

```
    //...用户代码
```

```
    OSInit(); /* 初始化  $\mu$  C/OS-II */
```

```
    //...用户代码
```

```
    //...用户代码
```

```
    /* 创建一个事件标志组并初始化 */
```

```
    EngineStatus = OSFlagCreate(0x00, &err);
```

```
    //...用户代码
```

```
//...用户代码
OSStart(); /*启动多任务*/
}
```

9.2.2 OSFlagAccept ()

函数原型:

```
OS_FLAGS OSFlagAccept (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT8U *perr)
```

函数功能:

函数查看指定的事件标志组的组合状态。 OSFlagAccept () 函数并不挂起任务。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

OS_FLAGS flags: 期望的标志位组合值, 如: 0x80,0x11 等等。

INT8U wait_type: 指定的检查类型, 有以下几种选项:

- 1) OS_FLAG_WAIT_CLR_ALL(AND): 期望的所有的事件标志位都清零时 (与关系) 有效。
- 2) OS_FLAG_WAIT_CLR_ANY(OR): 期望的所有的事件标志位有一个清零时 (或关系) 有效。
- 3) OS_FLAG_WAIT_SET_ALL(AND): 期望的所有的事件标志位都置位时 (与关系) 有效。
- 4) OS_FLAG_WAIT_SET_ANY(OR): 期望的所有的事件标志位有一个置位时 (或关系) 有效。

INT8U *perr: 指向错误代码变量的指针, 其返回值如下:

- 1) OS_ERR_NONE : 成功获取到指定标志。
- 2) OS_ERR_EVENT_TYPE pgrp : 不是指向一个事件标志组。
- 3) OS_ERR_FLAG_WAIT_TYPE : 没有指定一个合适的 wait_type 类型。
- 4) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针, 可能事件标志组没有被创建。
- 5) OS_ERR_FLAG_NOT_RDY : 等待所需的标志不可用。

返回值:

在 OSFlagAccept () 调用成功时返回事件标志组当前的标志。

说明:

本函数在成功获取到期望的标志位后, 并不会清掉或设置相关标志位。如果想在成功获取到期望的标志位后清除相关标志位, 则可以在原有 wait_type 类型的基础上加上 OS_FLAG_CONSUME 项。如: (OS_FLAG_WAIT_SET_ANY | OS_FLAG_CONSUME)。

示例:

```
/*LED2 任务*/
void led2_task(void *pdata)
{
    u8 err;
    OS_FLAGS value;
    pdata=pdata; /*防止编译器优化警告*/
    while(1)
    {
        /*查询标志*/
        value = OSFlagAccept(FlagStat,(1<<2) | (1<<1), OS_FLAG_WAIT_SET_AND,&err);
        if(err == OS_ERR_NONE) /*成功获取标志*/
        {
            LED2=0;
            OSTimeDly(60); /*延时 60 个时钟节拍*/
            LED2=1;
            OSTimeDly(100); /*延时 100 个时钟节拍*/
        }
    }
}
```

```

    sprintf((char *)str,"value: 0x%x",value);
    lcd_show_str(35,60,240,16,str,16,RED,WHITE,0);/*实时显示 value 值*/
    OSTimeDly(20);/*延时 20 个时钟节拍*/
}
}

```

9.2.3 OSFlagPend ()

函数原型:

```
OS_FLAGS OSFlagPend (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U wait_type, INT16U timeout, INT8U *perr)
```

函数功能:

用来在任务中等待事件标志组中条件的组合（置位或清零）。如果调用任务的条件不可用，那么调用本函数的任务将被挂起，直到所需的条件满足或指定的时间超时任务才恢复执行。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

OS_FLAGS flags: 包含在事件标志组中存储的初始值。

INT8U wait_type: 指定的检查类型，有以下几种选项:

- 1) OS_FLAG_WAIT_CLR_ALL(AND): 期望的所有的事件标志位都清零时（与关系）有效。
- 2) OS_FLAG_WAIT_CLR_ANY(OR): 期望的所有的事件标志位有一个清零时（或关系）有效。
- 3) OS_FLAG_WAIT_SET_ALL(AND): 期望的所有的事件标志位都置位时（与关系）有效。
- 4) OS_FLAG_WAIT_SET_ANY(OR): 期望的所有的事件标志位有一个置位时（或关系）有效。

INT16U timeout: 任务等待超时值，为 0 时表示无限等待；其它，递减到 0 时任务恢复执行。

INT8U *perr: 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 成功获取到指定标志。
- 2) OS_ERR_EVENT_TYPE pgrp : 不是指向一个事件标志组。
- 3) OS_ERR_FLAG_WAIT_TYPE : 没有指定一个合适的 wait_type 类型。
- 4) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针，可能事件标志组没有被创建。
- 5) OS_ERR_TIMEOUT : 未在指定的超时周期内获取到期望标志位。
- 6) OS_ERR_PEND_ISR : 尝试从一个 ISR 调用 OSFlagPend(), 不允许。

返回值:

使任务准备就绪的标志（或 0），如果没有一个标志是准备好的，或是出现错误。

说明:

- 1) 必须先创建邮箱在使用本函数。
- 2) 不允许在中断函数中调用本函数。
- 3) 本函数在成功获取到期望的标志位后，并不会清掉或设置相关标志位。如果想在成功获取到期望的标志位后清除相关标志位，则可以在原有 wait_type 类型的基础上加上 OS_FLAG_CONSUME 项。如：(OS_FLAG_WAIT_CLR_AND | OS_FLAG_CONSUME)。

示例:

```

/*LED3 任务*/
void led3_task(void *pdata)
{
    u8 err;
    pdata=pdata;/*防止编译器优化警告*/
    while(1)
    {
        OSFlagPend(FlagStat,0x10 | 0x02, OS_FLAG_WAIT_SET_AND,0,&err);
        LED3=0;
        OSTimeDly(50);/*延时 50 个时钟节拍*/
    }
}

```



```
LED3=1;
OSTimeDly(120);/*延时 120 个时钟节拍*/
}
}
```

9.2.4 OSFlagPost ()

函数原型:

```
OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *perr)
```

函数功能:

根据位掩码 (flags) 置位或者清零事件标志组中的对应位。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

OS_FLAGS flags: 包含在事件标志组中存储的初始值。

INT8U opt: 指定的操作类型, 有以下 2 种:

1) OS_FLAG_SET: 置位。

2) OS_FLAG_CLR: 清零。

INT8U *perr: 指向错误代码变量的指针, 其返回值如下:

1) OS_ERR_NONE : 设置成功

2) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针, 可能事件标志组没有被创建。

3) OS_ERR_EVENT_TYPE : pgrp 不是指向一个事件标志组。

4) OS_ERR_FLAG_INVALID_OPT : opt 指定了一个无效的选项。

返回值:

事件标志组中新的标志状态。

说明:

无。

示例:

```
/*按键任务*/
void key_task(void *pada)
{
    u8 kdat,err;
    pada=pada;/*防止编译器警告*/
    while(1)
    {
        kdat = KEY_Scan(0);
        switch(kdat)
        {
            case Up_KEY: OSFlagPost(FlagStat, 0xffff, OS_FLAG_CLR, &err);/*清零所有位*/
                break;
            case Down_KEY: OSFlagPost(FlagStat, (1<<1), OS_FLAG_SET, &err);/*第 1 位置位*/
                break;
            case Left_KEY: OSFlagPost(FlagStat, (1<<2), OS_FLAG_SET, &err);/*第 2 位置位*/
                break;
            case Right_KEY: OSFlagPost(FlagStat, (1<<3), OS_FLAG_SET, &err);/*第 3 位置位*/
                break;
            default:
                break;
        }
    }
}
```

```

    OSTimeDly(20);/*延时 20 个时钟节拍*/
}
}

```

9.2.5 OSFlagDel ()

函数原型:

```
OS_FLAG_GRP *OSFlagDel (OS_FLAG_GRP *pgrp, INT8U opt, INT8U *perr)
```

函数功能:

用于删除一个事件标志组。使用这个函数是很危险的，因为多个任务可能是依赖于事件标志组而运行的。所以在删除事件标志组之前，必须先删除所有访问事件标志组的任务。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

INT8U opt: 选择指定要删除事件标志组类型:

- 1) OS_DEL_NO_PEND: 只有当不存在因等待事件标志组而挂起的任务删除时才删除，否则不删除。
- 2) OS_DEL_ALWAYS: 直接删除。

INT8U *perr: 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 事件标志组已经删除。
- 2) OS_ERR_DEL_ISR : 尝试从一个 ISR 删除一个事件标志组。
- 3) OS_ERR_FLAG_INVALID_PGRP : PGRP 传入的是一个空指针。
- 4) OS_ERR_EVENT_TYPE : PGRP 不是指向一个事件标志组。
- 5) OS_ERR_INVALID_OPT : 传入选项参数(opt)有误。
- 6) OS_ERR_TASK_WAITING : 一个或多个任务正在等待事件标志组量。

返回值:

(OS_FLAG_GRP *)0 : 如果事件标志组被成功删除，则返回空指针。

其他 : 删除失败，需要检查错误代码以确定原因。

说明:

- 1) 必须先创建邮箱在使用本函数。
- 2) 不允许在中断函数中调用本函数。

示例:

```

OS_FLAG_GRP *EngineStatusFlags;
void Task (void *p_arg)
{
    INT8U err;
    OS_FLAG_GRP *pgrp;
    p_arg=p_arg;
    while (1)
    {
        //...用户代码
        //...用户代码
        pgrp = OSFlagDel(EngineStatusFlags, OS_DEL_ALWAYS, &err);
        if (pgrp == (OS_FLAG_GRP *)0)
        {
            /* 事件标志组已成功删除*/
        }
        //...用户代码
        //...用户代码
    }
}

```

```
}
```

9.2.6 OSFlagQuery ()

函数原型:

```
OS_FLAGS OSFlagQuery (OS_FLAG_GRP *pgrp, INT8U *perr)
```

函数功能:

被用于获得一组事件标志的当前值。此时，这个函数不返回任务等待事件标志组的列表。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

INT8U *perr: 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 获取成功。
- 2) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针，可能事件标志组没有被创建。
- 3) OS_ERR_EVENT_TYPE : pgrp 不是指向一个事件标志组。

返回值:

事件标志组中的标志状态。

说明:

无。

示例:

```
OS_FLAG_GRP *EngineStatusFlags;
void Task (void *p_arg)
{
    OS_FLAGS flags;
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        //...用户代码
        flags = OSFlagQuery(EngineStatusFlags, &err);/*获取当前事件标志组标志位*/
        //...用户代码
        //...用户代码
    }
}
```

9.2.7 OSFlagNameGet ()

函数原型:

```
INT8U OSFlagNameGet (OS_FLAG_GRP *pgrp, INT8U *pname, INT8U *perr)
```

函数功能:

可以获取你分配给事件标志组的名称。

函数参数:

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

INT8U *pname: 是一个指向指针的指针的事件标志组的名称。

INT8U *perr: 指向错误代码变量的指针，其返回值如下:

- 1) OS_ERR_NONE : 获取成功
- 2) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针，可能事件标志组没有被创建。
- 3) OS_ERR_PNAME_NULL: pname 指向 NULL。
- 4) OS_ERR_EVENT_TYPE : pgrp 不是指向一个事件标志组。

返回值:

指向 PNAME 的 ASCII 字符串的大小；如果为 0，说明遇到错误。

说明：

无。

示例：

```
INT8U *EngineStatusName; /*事件标志组名称指针*/
OS_FLAG_GRP *EngineStatusFlags; /*事件标志组指针*/
void Task (void *p_arg)
{
    INT8U err;
    INT8U size;
    p_arg=p_arg;
    while(1)
    {
        size = OSFlagNameGet(EngineStatusFlags,
        &EngineStatusName,
        &err); /*获取事件标志组名称*/
        //...用户代码
        //...用户代码
    }
}
```

9.2.8 OSFlagNameSet ()

函数原型：

```
void OSFlagNameSet (OS_FLAG_GRP *pgrp, INT8U *pname, INT8U *perr)
```

函数功能：

可以设置事件标志组的名称。

函数参数：

OS_FLAG_GRP *pgrp: 指向事件标志组的指针。

INT8U *pname: 指向包含该事件标志组的名称的 ASCII 字符串。

INT8U *perr: 指向错误代码变量的指针，其返回值如下：

- 1) OS_ERR_NONE : 设置成功
- 2) OS_ERR_FLAG_INVALID_PGRP : pgrp 传入的是一个空指针，可能事件标志组没有被创建。
- 3) OS_ERR_PNAME_NULL : pname 指向 NULL。
- 4) OS_ERR_EVENT_TYPE: pgrp 不是指向一个事件标志组。
- 5) OS_ERR_NAME_SET_ISR: 在 ISR 使用本函数，不允许。

返回值：

无。

说明：

无。

示例：

```
OS_FLAG_GRP *EngineStatus; /*事件标志组指针*/
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
```

```
/*设置事件标志组名称*/
OSFlagNameSet(EngineStatus, "Engine Status Flags", &err);
//...用户代码
//...用户代码
}
}
```

9.2.9 OSFlagPendGetFlagsRdy ()

函数原型:

OS_FLAGS OSFlagPendGetFlagsRdy (void)

函数功能:

用于获取当前使任务就绪的标志。

函数参数:

无。

返回值:

使任务就绪的标志。

示例:

```
/*LED3 任务*/
void led3_task(void *pdata)
{
    u8 err;
    OS_FLAGS flags;
    pdata= pdata; /*防止编译器优化警告*/
    while(1)
    {
        OSFlagPend(FlagStat,(1<<4) | (1<<1), OS_FLAG_WAIT_SET_AND,100,&err);
        if(err == OS_ERR_NONE) /*成功获取到期望的标志*/
        {
            flags = OSFlagPendGetFlagsRdy(); /*获取使任务就绪的标志*/
            printf("FlagsRdy: 0x%x",flags);
            LED3=0;
            OSTimeDly(50); /*延时 50 个时钟节拍*/
            LED3=1;
            OSTimeDly(120); /*延时 120 个时钟节拍*/
        }
    }
}
```

9.3 μ C/OS- II 事件标志组编程思想

1. 任务的创建与删除

设计 3 个任务,任务 Start 用于创建其他任务,任务 LED 在执行事件标志组显示内容,并记录任务执行的次数,通过串口显示任务执行的内容。任务 KEY 功能是扫描按键,当按键按下时删除事件标志组。

2. 任务的等待与发送

设计 3 个任务,任务 Start 用于创建其他任务,任务 LED 在执行等待事件标志组,并记录任务执行的内容,通过串口显示任务执行的内容。任务 KEY 功能是扫描按键,当按键按下时发送事件标志组标志。

3. 任务的检查

设计 3 个任务,任务 Start 用于创建其他任务,任务 LED 在执行等待事件标志组,并记录任务执行的内容,通

过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键 up 和 down 键按下时发送事件标志组标志，当按键 left 键按下时删除事件标志组，当按键 right 键按下时，还可以检查事件标志组的当前值。

9.4 μ C/OS- II 事件标志组编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、在 uc0s-task.c 中首先定义一个事件标志组指针，如下：

OS_FLAG_GRP *EventFlags; //创建事件标志组指针

2、在 main.c 中创建任务 Start_Task(), 然后在 uc0s-task.c 开始任务 Start_Task()中调用 OSFlagCreate()函数创建一个事件标志组，代码如下：

```
void Start_Task(void *p_arg)
{
    INT8U perr;
    p_arg=p_arg;
    EventFlags=OSFlagCreate(0x00,&perr);//创建一个事件标志组，初始值为 0x00
    if(perr== OS_ERR_NONE)
    {
        printf("成功创建一个事件标志组\r\n");
    }
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
    while (1)
    {
        OSTaskSuspend(Start_Task_Pro); //把任务挂起
    }
}
```

3、任务的等待与发送

在 uc0s-task.c 中设计 3 个任务，任务 Start 用于创建其他任务,任务 LED 在执行等待事件标志组，并记录任务执行的内容，通过串口显示任务执行的内容。任务 KEY 功能是扫描按键，当按键按下时发送事件标志组标志。

//任务 1 的任务函数

void Led_Task(void *pdata)

```
{
    INT8U err;
    pdata=pdata;
    while (1)
    {

        //OSFlagPend(EngineStatus,0x01+0x02, OS_FLAG_WAIT_SET_ALL,0,&err);//阻塞的等待一个标志性事件组,
        不清标志
        OSFlagPend(EngineStatus,0x01+0x02, OS_FLAG_WAIT_SET_ALL+OS_FLAG_CONSUME,0,&err);//阻塞的等待
        一个标志性事件组,清标志
        printf("事件标志组组 EngineStatus 的值的值:%d\r\n",EngineStatus->OSFlagFlags);
        OSTimeDly (1000);
    }
}
```

//任务 2 的任务函数

void Key_Task(void *pdata)

```
{
```

```
u8 key;
INT8U perr;
OS_FLAGS res;
pdata=pdata;
while (1)
{
    key = KEY_Scan(0);
    switch(key)
    {
        case 1:    //当 up 键按下,向事件标志组 EventFlags 发送标志
                    res=OSFlagPost(EngineStatus,0x01,OS_FLAG_SET,&perr);
                    printf("key1err:%d\r\n",perr);
                    printf("事件标志 EngineStatus:%d\r\n",res);

                    break;
        case 2:    //当 down 键按下,向事件标志组 EventFlags 发送标志
                    res=OSFlagPost(EngineStatus,0x02,OS_FLAG_SET,&perr);
                    printf("key2err:%d\r\n",perr);
                    printf("事件标志 EngineStatus:%d\r\n",res);

                    break;
    }
    OSTimeDly(10);
}
}
```

第十章 $\mu\text{C}/\text{OS-II}$ 软件定时器管理

10.1 $\mu\text{C}/\text{OS-II}$ 软件定时器简介

10.1.1 软件定时器的介绍

定时器本质是递减计数器，当计数器减到零时可以触发某种动作的执行，这个动作通过回调函数来实现。当定时器计时完成时，定义的回调函数就会被立即调用，应用程序可以有任意数量的定时器， $\mu\text{C}/\text{OS-II}$ 中定时器的时间分辨率由一个宏 `OS_TMR_CFG_TICKS_PER_SEC`，单位为 HZ，默认为 10Hz。

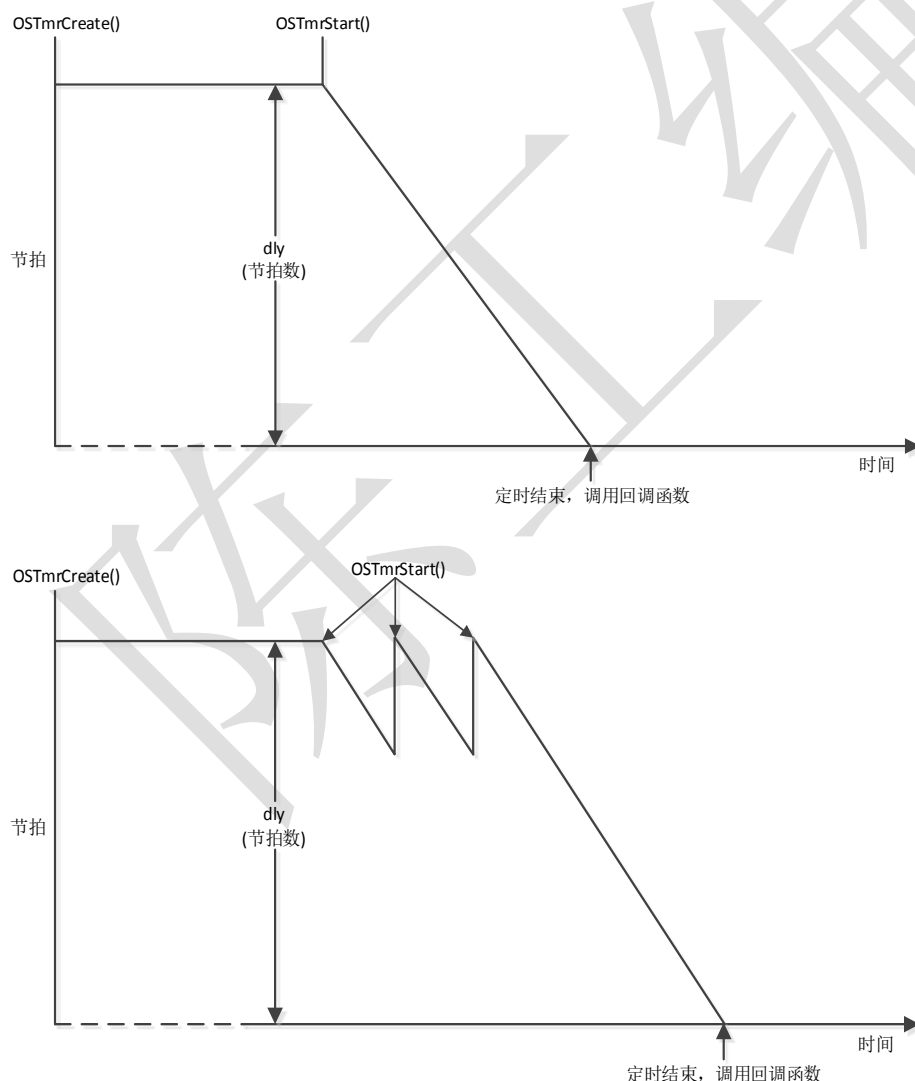
注意！一定要避免在回调函数中使用阻塞调用或者可以阻塞或删除定时器任务的函数。

10.1.2 回调函数的介绍

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方法直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

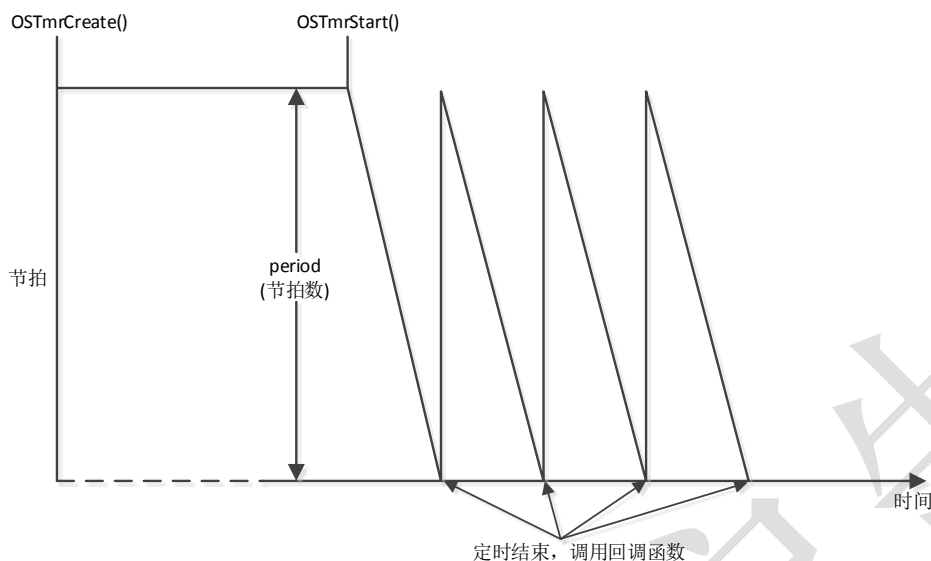
10.1.3 单次定时器

单次定时器从初始值(也就是 `OSTmrCreate()` 函数中的参数 `dly`)开始倒数，直到为 0 调用回调并停止。单次定时器的定时器只执行一次。



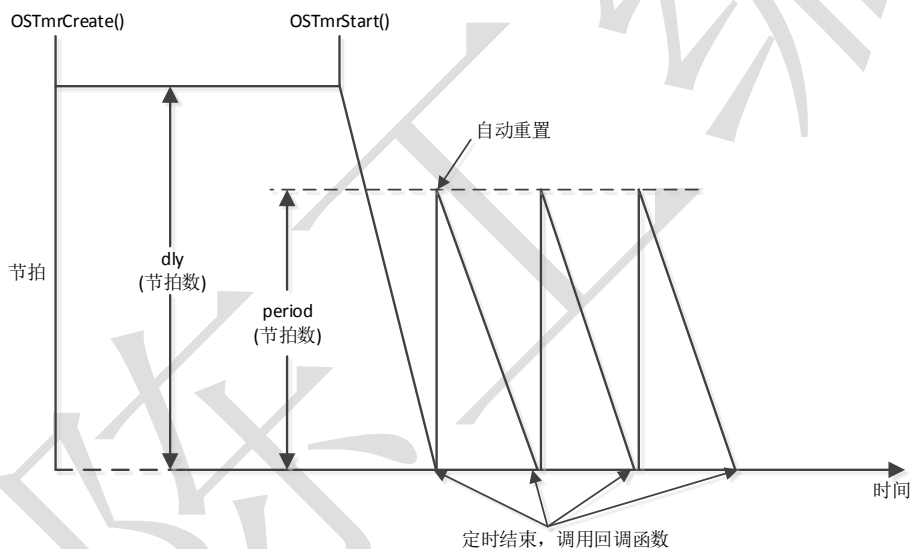
10.1.4 周期模式--无初始延迟

创建定时器的时候我们可以设定为周期模式，当倒计时完成后，定时器调用回调函数，并重置计数器重新开始计时，一直循环性下去。如果在调用函数 `OSTmrCreate()` 创建周期定时器时让参数 `dly` 为 0，那么定时器每个周期就是 `period`。



10.1.5 周期模式--有初始延迟

周期定时器也可以设定为带初始延迟时间的运行模式，使用函数 `OSTmrCreate()` 参数 `dly` 来确定第一个周期，以后的每个周期开始时将计数器值重置为 `period`。



10.2 μ C/OS- II 软件定时器相关函数

10.2.1 `OSTmrCreate()`

函数原型：

```
OS_TMR *OSTmrCreate (INT32U          dly,
                     INT32U          period,
                     INT8U           opt,
                     OS_TMR_CALLBACK callback,
                     void            *callback_arg,
                     INT8U           *pname,
                     INT8U           *perr)
```

函数功能：

用于创建一个定时器。定时器可以被配置为周期性运行或单次运行。当倒计时（设定的定时值）到 0，可选的“回调”函数会被执行。回调函数用于发信号告诉任务定时器时间到或者执行任何其他功能。但是建议保持回调函数可能短。使用时必须调用 `OSTmrStart()` 来启动计时器。如果配置了定时器单次计时模式，定时时间到，需要调用 `OSTmrStart()` 来重新触发定时器。如果不打算重新触发它，或者不使用定时器了，可以调用 `OSTmrDel()` 来删除的定时器。如果使用单次计时模式，可以在回调函数里调用 `OSTmrDel()` 删除定时器。使用软件定时器时，必须指定本任务的优先级（`OS_TASK_TMR_PRIO`），一般设置为最高优先级。

函数参数：

INT32U dly: 指定使用的计时器初始延迟时间。

- 1) 在单次触发模式时，这是单次定时的时间。
- 2) 在周期模式下，这是最初定时器进入周期模式之前的延时。
- 3) 延时单位取决于你多久调用一次 `OSTmrSignal()`。如果 `OS_TMR_CFG_TICKS_PER_SEC` 设置为 10，则 `DLY` 指定延迟时间为 1/10 秒（也就是 100ms）。`OS_TMR_CFG_TICKS_PER_SEC` 设置为多少，就是 1/`OS_TMR_CFG_TICKS_PER_SEC` 秒。需要注意的是在创建时的定时器不能启动。

INT32U period: 指定计时器周期性定时的时间量。如果设置为 0，那么将使用单次模式。

INT8U opt: 1) `OS_TMR_OPT_PERIODIC`: 指定周期性定时模式。

2) `OS_TMR_OPT_ONE_SHOT`: 指定单次定时模式。

OS_TMR_CALLBACK callback: 1) 指定一个回调函数的地址（可选），这个函数将被调用，处理定时结束的相关事项，回调函数必须

声明如下：

```
void MyCallback (void *ptmr, void *callback_arg);
```

2) 如果不需要回调函数，可以传入一个空指针。

void *callback_arg: 是传递给回调函数参数，如果回调函数不需要参数，可以是一个空指针。

INT8U *pname: 指向一个 ASCII 字符串，用于设置定时器名称。通过调用 `ostmrnameget()` 获取这个名称。

INT8U *perr: 指向存放错误代码的指针，并且可以是以下之一：

- 1) `OS_ERR_NONE`: 定时器已成功创建。
- 2) `OS_ERR_TMR_INVALID_DLY`: 在单次延时模式下指定了 0 延时。
- 3) `OS_ERR_TMR_INVALID_PERIOD`: 在周期性延时模式下指定了 0 延时。。
- 4) `OS_ERR_TMR_INVALID_OPT`: 指定的延时模式有误。
- 5) `OS_ERR_TMR_ISR`: 在中断函数中调用本函数，不允许。
- 6) `OS_ERR_TMR_NON_AVAIL`: 当你不能启动一个计时器时，你会得到这个错误，因为所有的定时器元素（即对象）已经被分配。
- 7) `OS_ERR_TMR_NAME_TOO_LONG`: 设置的定时器的名字太长，必须比 `OS_TMR_CFG_NAME_SIZE` 小。

返回值：

(OS_TMR *)0：如果没有可用的时间控制块，返回空指针。

不等于 0：返回指向分配给所建立的定时器的时间控制块的指针。

说明：

- 1) 建议检查返回值，以确保计时器成功创建。
- 2) 不能从一个中断服务程序调用这个函数。
- 3) 请注意，在创建时不启动计时器。开始计时时必须调用 `ostmrstart()` 启动计时器。

示例：

```
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
```

```

CloseDoorTmr = OSTmrCreate( 10,
                             100,
                             OS_TMR_OPT_PERIODIC,
                             DoorCloseFnct,
                             (void *)0,
                             "Door Close",
                             &err);

if (err == OS_ERR_NONE)
{
    /* 定时器创建成功，但还没启动*/
}
}
}

```

10.2.2 OSTmrStart()

函数原型:

```

BOOLEAN OSTmrStart (OS_TMR *ptmr,
                    INT8U *perr)

```

函数功能:

启动（或重新启动）计时器的倒计时过程。要启动的计时器必须是先前已经建立的。

函数参数:

OS_TMR *ptmr: 指向想要启动的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种:

- 1) OS_ERR_NONE: 启动成功。
- 2) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 3) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
- 4) OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数，不允许。
- 5) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值:

OS_TRUE: 定时器启动成功。

OS_FALSE: 操作有误。

说明:

- 1) 建议检查函数的返回值。
- 2) 不能在中断函数调用本函数。
- 3) 要启动的计时器必须是先前已经建立的。

示例:

```

OS_TMR *CloseDoorTmr;
BOOLEAN status;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        status = OSTmrStart(CloseDoorTmr,&err);
        if (err == OS_ERR_NONE)
        {

```

```

        /* Timer was started */
    }
}
}

```

10.2.3 OSTmrStop()

函数原型:

```

BOOLEAN OSTmrStop(OS_TMR *ptmr,
                  INT8U opt,
                  void *callback_arg,
                  INT8U *perr)

```

函数功能:

用于停止（即中止）定时器。

函数参数:

OS_TMR *ptmr: 指向想要停止的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

INT8U opt: 停止后的要做什么事情:

- 1) OS_TMR_OPT_NONE: 直接停止，不做任何其他处理。
- 2) OS_TMR_OPT_CALLBACK: 停止，用初始化的参数执行一次回调函数。
- 3) OS_TMR_OPT_CALLBACK_ARG: 停止，用新的参数执行一次回调函数。

void *callback_arg: 新的回调函数参数，如果回调函数不需要参数，可以是一个空指针。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种:

- 1) OS_ERR_NONE: 计时器已停止。
- 2) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 3) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
- 4) OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数，不允许。
- 5) OS_ERR_TMR_INVALID_OPT: 指定了无效选项。
- 6) OS_ERR_TMR_STOPPED: 试图停止一个已停止计时器。
- 7) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。
- 8) OS_ERR_TMR_NO_CALLBACK: 选项中指定了调用回调函数，但没有传入回调函数相关参数。

返回值:

- 1) OS_TRUE: 定时器停止成功。
- 2) OS_FALSE: 操作有误。

说明:

- 1) 建议检查函数的返回值。
- 2) 不能在中断函数调用本函数。
- 3) 要停止的计时器必须是先前已经建立的。

示例:

```

OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        OSTmrStop(CloseDoorTmr,
                  OS_TMR_OPT_CALLBACK,
                  (void *)0,

```

```
&err);
if (err == OS_ERR_NONE || err == OS_ERR_TMR_STOPPED)
{
    /* Timer was stopped ... */
    /* ... callback was called only if timer was running */
}
}
```

10.2.4 OSTmrDel()

函数原型:

```
BOOLEAN OSTmrDel(OS_TMR *ptmr,
                  INT8U *perr)
```

函数功能:

允许你删除一个定时器。如果一个计时器正在运行，它先会被停止，然后被删除。如果定时器已经到了，并且停止不再计时，那么将直接被删除。它是由你来删除未使用的计时器。如果删除一个计时器，你必须不能再使用它。

函数参数:

OS_TMR *ptmr: 指向想要删除的定时器。该指针的值为在创建定时器时返回的值（见 OSTmrCreate（））。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种:

- 1) OS_ERR_NONE: 定时器被成功删除。
- 2) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 3) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
- 4) OS_ERR_TMR_ISR: 在 ISR 中调用本函数，不允许。
- 5) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值:

OS_TRUE: 定时器已删除。

OS_FALSE: 出现错误。

说明:

- 1) 建议检查返回值，以确本函数的执行结构是有效的。
- 2) 不能调用从 ISR 本函数。
- 3) 如果你删除了一个定时器，必须不能再引用它。

示例:

```
OS_TMR *CloseDoorTmr;
void Task(void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        CloseDoorTmr = OSTmrDel(CloseDoorTmr,
                                &err);
        if (err == OS_ERR_NONE)
        {
            /* Timer was deleted ... DO NOT reference it anymore! */
        }
    }
}
```

}

10.2.5 OSTmrStateGet()

函数原型:

```
INT8U OSTmrStateGet (OS_TMR *ptmr,
                    INT8U *perr)
```

函数功能:

用来获取计时器的当前状态。

函数参数:

OS_TMR *ptmr: 指向想要获取信息的定时器。该指针的值为在创建定时器时返回的值。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种:

- 1) OS_ERR_NONE: 启动成功。
- 2) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 3) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
- 4) OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数, 不允许。
- 5) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值:

定时器的状态, 有以下几种:

- 1) OS_TMR_STATE_UNUSED: 计时器尚未创建。
- 2) OS_TMR_STATE_STOPPED: 定时器已创建, 但尚未开始或已经停止计时。
- 3) OS_TMR_STATE_COMPLETED: 定时器在单次模式, 已经完成了延时。
- 4) OS_TMR_STATE_RUNNING: 当前定时器正在运行。

说明:

- 1) 建议检查函数的返回值。
- 2) 不能在中断函数调用本函数。

示例:

```
INT8U CloseDoorTmrState;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        CloseDoorTmrState = OSTmrStateGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE)
        {
            /* Call was successful */
        }
    }
}
```

10.2.6 OSTmrNameGet ()

函数原型:

```
INT8U OSTmrNameGet (OS_TMR *ptmr,
                    INT8U *pdest,
                    INT8U *perr)
```

函数功能:

用于获取定时器的名称。

函数参数：

OS_TMR *ptmr: 指向想要获取名称的定时器。该指针的值为在创建定时器时返回的值。

INT8U *pdest: 指向一个要存放 ASCII 字符串内存区的首地址。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种：

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_TMR_INVALID_DEST: pname 传入一个空指针。
- 3) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 4) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。
- 5) OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数，不允许。
- 6) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值：

计数器名称的字符长度。

说明：

- 1) 建议检查函数的返回值。
- 2) 不能在中断函数调用本函数。

示例：

```
INT8U *CloseDoorTmrName;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        OSTmrNameGet(CloseDoorTmr, &CloseDoorTmrName, &err);
        if (err == OS_ERR_NONE)
        {
            /* CloseDoorTmrName points to the name of the timer */
        }
    }
}
```

10.2.7 OSTmrRemainGet ()

函数原型：

```
INT32U OSTmrRemainGet (OS_TMR *ptmr,
                      INT8U *perr)
```

函数功能：

可以获取指定定时器的剩余（定时）的时间。该返回值表示的时间取决于 OS_TMR_CFG_TICKS_PER_SEC 的配置。如果 OS_TMR_CFG_TICKS_PER_SEC 设置为 10，那么返回的剩下的真实时间为：返回时间*100ms。如果定时器超时，返回值为 0。

函数参数：

OS_TMR *ptmr: 指向想要获取剩余时间的定时器。该指针的值为在创建定时器时返回的值。

INT8U *perr: 指向存放错误代码的变量。其值为以下几种：

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_TMR_INVALID: ptmr 传入一个空指针。
- 3) OS_ERR_TMR_INVALID_TYPE: ptmr 不是指向一个计时器。

4) OS_ERR_NAME_GET_ISR: 在 ISR 中调用本函数, 不允许。

5) OS_ERR_TMR_INACTIVE: ptmr 指向一个已被删除或没有创建一个定时器。

返回值:

返回指定的计时器的剩余时间。如果指定了无效计时器或者定时器时间已到, 返回的值将是 0。

说明:

- 1) 建议检查函数的返回值。
- 2) 不能在中断函数调用本函数。

示例:

```
INT32U TimeRemainToCloseDoor;
OS_TMR *CloseDoorTmr;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    for (;;)
    {
        TimeRemainToCloseDoor = OSTmrRemainGet(CloseDoorTmr, &err);
        if (err == OS_ERR_NONE)
        {
            /* Call was successful */
        }
    }
}
```

10.2.7 OSTmrSignal ()

函数原型:

```
INT8U OSTmrSignal (void)
```

函数功能:

通过任务或中断服务程序被调用, 用以更新计时器。通常情况下, OSTmrSignal()将由 OSTimeTickHook()按照 OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC 的时间来调用。换句话说, 如果 OS_CFG.H 中的 OS_TICKS_PER_SEC 设置为 1000, 那么你应该每 10 或 100 个滴答中断 (100 赫兹或 10 赫兹) 调用 OSTmrSignal()。一般情况下, 建议 100ms 调用一次 OSTmrSignal()更新计时器, 因为更高的计时器速率, 会使您的系统开销增大。

函数参数:

无。

返回值:

返回 OS_ERR_NONE 表明调用成功。

说明:

本函数一般由系统中断调用, 用户代码尽量不要调用。

示例:

```
#if OS_TMR_EN > 0
static INT16U OSTmrTickCtr = 0;
#endif
void OSTimeTickHook (void)
{
    #if OS_TMR_EN > 0
        OSTmrTickCtr++;
    }
```



```
if (OSTmrTickCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC))
{
    OSTmrTickCtr = 0;
    OSTmrSignal();
}
#endif
}
```

10.3 μ C/OS- II 软件定时器编程思想

1. 任务的创建与删除

设计 2 个任务，任务 Start 用于创建其他任务并且创建软件定时器。任务 KEY 功能是扫描按键，当按键按下时删除软件定时器。

2. 软件定时器单次模式

设计 2 个任务，任务 Start 用于创建其他任务并且创建软件定时器。任务 KEY 功能是扫描按键，当按键 up 键按下时开启软件定时器，当按键 down 键按下时关闭软件定时器。

3. 软件定时器循环模式

设计 2 个任务，任务 Start 用于创建其他任务并且创建软件定时器。任务 KEY 功能是扫描按键，当按键 up 键按下时开启软件定时器，当按键 down 键按下时关闭软件定时器。

4. 软件定时器的当前状态

设计 2 个任务，任务 Start 用于创建其他任务并且创建软件定时器。任务 KEY 功能是扫描按键，当按键 up 键按下时开启定时器，当按键 down 键按下时关闭定时器，当按键 left 键按下时删除定时器，当按键 right 键按下时获取定时器还剩多少时钟节拍。

10.4 μ C/OS- II 软件定时器编程示例

本实验部分源码如下，完整的工程详见工程示例代码：

1、在 uc0s-task.c 中首先定义一个软件定时器指针，如下：

```
OS_TMR *mytmr; //软件定时器指针
```

2、在 main.c 中创建任务 Start_Task()，然后在 uc0s-task.c 开始任务 Start_Task()中调用 OSTmrCreate()函数创建一个软件定时器，代码如下：

//单次模式

```
void Start_Task(void *p_arg)
```

```
{
    INT8U perr;
    p_arg=p_arg;
    //创建软件定时器。
    mytmr=OSTmrCreate( 20, //设置定时时间 20*节拍时间 =20*100ms =2000ms
                      0, //单次模式不使用此形参
                      OS_TMR_OPT_ONE_SHOT,//单次模式
                      MyCallback_Tmr, //回调函数
                      NULL, //传递给回调函数的实参
                      "my tmr", //软件定时器名字
                      &perr);
    if(perr == OS_ERR_NONE)
    {
        printf("成功创建软件定时器\r\n");
    }
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
}
```

```

while (1)
{
    OSTaskSuspend(Start_Task_Pro); //把任务挂起
}
}

//循环模式
void Start_Task(void *p_arg)
{
    INT8U perr;
    p_arg=p_arg;
    //创建软件定时器,周期模式, 有初始延时
    mytmr=OSTmrCreate( 50, //第一次的定时时间 --初始延时 50*时钟节拍=50*100
=5000ms
20, //以后每次的定时时间 20*时钟节拍=20*100 =2000ms
OS_TMR_OPT_PERIODIC,//周期模式
MyCallback_Tmr, //回调函数
NULL, //传递给回调函数的实参
"my tmr", //软件定时器名字
&perr);

if(perr == OS_ERR_NONE)
{
    printf("成功创建软件定时器\r\n");
}
OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
while (1)
{
    OSTaskSuspend(Start_Task_Pro); //把任务挂起
}
}

```

3、软件定时器的开启和关闭

在 ucosp-task.c 中设计 2 个任务，任务 Start 用于创建其他任务并且创建软件定时器。任务 KEY 功能是扫描按键，当按键 up 键按下时开启软件定时器，当按键 down 键按下时关闭软件定时器。

//任务函数

```

void Key_Task(void *p_arg)
{
    u8 key;
    INT8U perr;
    p_arg=p_arg;
    while (1)
    {
        key = KEY_Scan(0);
        switch(key)
        {
            case up_press: //当 up 键按下,开启定时器
                OSTmrStart(mytmr,&perr);
                if(perr ==OS_ERR_NONE)

```

```
        {
            printf("开启定时器\r\n");
            LCD_Show_String(30,130,280,16,BLUE,WHITE,"Start tmr OK");
        }
        break;
    case down_press: //当 down 按下,关闭定时器
        OSTmrStop(mytmr,OS_TMR_OPT_NONE,0,&perr);
        if(perr == OS_ERR_NONE)//这里不会执行,因为设置的是单次模式,执行了后
系统自动关闭了。

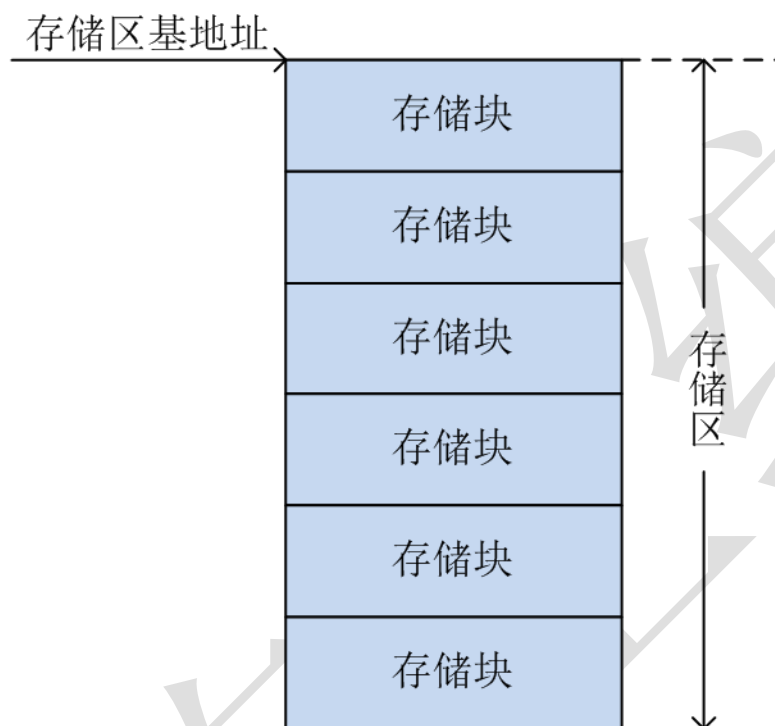
        {
            printf("关闭定时器\r\n");
            LCD_Show_String(30,130,280,16,BLUE,WHITE,"Stop tmr OK");
        }
        break;
    }
    OSTimeDly(10);
}
}
```

第十一章 $\mu\text{C}/\text{OS-II}$ 内存管理

11.1 $\mu\text{C}/\text{OS-II}$ 内存管理的介绍

在标准 C 中, 用户可以使用 `malloc()` 函数动态的申请内存空间, 用 `free()` 函数释放 `malloc()` 函数申请到的内存空间。但是, 如果用户在 $\mu\text{C}/\text{OS-II}$ 操作系统中多次调用 `malloc()` 函数来申请内存空间的话, 就有可能把原来很大的一块内存分割成了很多小块的内存空间, 并且这些内存空间还不是连续的, 也就是常说的内存碎片。如果工程中存在太多的内存碎片, 那么到最后将会导致连一小块内存都分配不到了。另外, 由于内存管理算法的原因, `malloc()` 和 `free()` 函数执行时间是不确定的。

在 $\mu\text{C}/\text{OS-II}$ 中, 操作系统把连续的大块内存按分区来管理。每个分区中包含有整数个大小相同的内存块。利用这种机制, $\mu\text{C}/\text{OS-II}$ 对 `malloc()` 和 `free()` 函数进行了改进, 使得它们可以分配和释放固定大小的内存块。这样一来, `malloc()` 和 `free()` 函数的执行时间也是固定的了。如下图所示:



一般情况下, 存储区是固定的, 所以, 用户可以在程序中可以用数组来表示一个存储区, 比如 `unsigned char mem_buffer[50][32]`, 就可以表示一个拥有 50 个存储块, 每个存储块 32 个字节的存储区

11.1.1 内存控制块

为了方便内存管理, $\mu\text{C}/\text{OS-II}$ 中使用内存控制块来表示内存区, 内存控制块定义为一个类型为 `OS_MEM` 的结构体。此结构体在 `μcos_ii.h` 中定义:

```
typedef struct os_mem {          /* MEMORY CONTROL BLOCK */
    void *OSMemAddr;            /* Pointer to beginning of memory partition */
    void *OSMemFreeList;        /* Pointer to list of free memory blocks */
    INT32U OSMemBlkSize;        /* Size (in bytes) of each block of memory */
    INT32U OSMemNBlks;          /* Total number of blocks in this partition */
    INT32U OSMemNFree;          /* Number of memory blocks remaining in this partition */
#ifdef OS_MEM_NAME_SIZE > 1
    INT8U OSMemName[OS_MEM_NAME_SIZE]; /* Memory partition name */
#endif
};
```

```
} OS_MEM
```

成员说明:

OSMemAddr: 指向内存分区中的基地址的指针, 在创建内存分区时被初始化, 之后无法被改变。

OSMemFreeList: 指向下一个空闲的内存控制块。

OSMemBlkSize: 内存控制块的大小, 在创建内存分区时初始化。

OSMemNBlks: 控制块的数量, 在创建内存分区时初始化。

OSMemNFree: 当前空闲的内存控制块。

11.2 μ C/OS-II 内存管理相关函数

11.2.1 OSMemCreate()

函数原型:

```
OS_MEM *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *perr)
```

函数功能:

函数建立并初始化一块内存区。一块内存区包含指定数目的大小确定的内存块。程序可以包含这些内存块并在用完后释放回内存区。

函数参数:

void *addr: 建立的内存区的起始地址。内存区可以使用静态数组或在初始化时使用 **malloc()** 函数建立。

INT32U nblks: 需要的内存块的数目。每一个内存区最少需要定义两个内存块。

INT32U blksize: 每个内存块的大小, 最少应该能够容纳一个指针。

INT8U *perr: 是指向包含错误码的变量的指针。OSMemCreate() 函数返回的错误码可能为下述几种:

- 1) **OS_ERR_NONE** : 成功建立内存区。
- 2) **OS_ERR_MEM_INVALID_ADDR**: 指定一个无效的地址(空指针)或分区没有正确对齐。
- 3) **OS_MEM_INVALID_PART** : 没有空闲的内存区。
- 4) **OS_MEM_INVALID_BLKS** : 没有为每一个内存区建立至少两个内存块。
- 5) **OS_MEM_INVALID_SIZE** : 内存块大小不足以容纳一个指针变量。

返回值:

等于(**OS_MEM ***)0: 如果没有空闲内存区, 返回空指针。

不等于(**OS_MEM ***)0: 返回指向内存区控制块的指针。

说明:

必须首先建立内存区, 然后使用。

示例:

```
OS_MEM *CommMem;
INT32U CommBuf[16][32];
void main(void)
{
    INT8U err;
    OSInit(); /* Initialize  $\mu$ C/OS-II */
    //...用户代码
    CommMem = OSMemCreate(&CommBuf[0][0], 16, 32 * sizeof(INT32U), &err);
    //...用户代码
    OSStart(); /* Start Multitasking */
}
```

11.2.2 OSMemGet ()

函数原型:

```
void *OSMemGet(OS_MEM *pmem, INT8U *perr)
```

函数功能:

函数用于从内存区分配一个内存块。用户程序必须知道所建立的内存块的大小，同时用户程序必须在使用完内存块后释放内存块。可以多次调用 **OSMemGet ()** 函数。

函数参数：

OS_MEM *pmem：是指向内存区控制块的指针，可以从 **OSMemCreate ()** 函数返回得到。

INT8U *perr：是指向包含错误码的变量的指针。返回的错误码可能为下述几种：

- 1) **OS_ERR_NONE**：成功得到一个内存块。
- 2) **OS_MEM_NO_FREE_BLK**：内存区已经没有空间分配给内存块。

返回值：

返回指向内存区控制块的指针。如果没有空闲的内存区，返回空指针。

说明：

必须首先建立内存区，然后使用。

示例：

```
OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U *pmsg;
    p_arg=p_arg;
    while(1)
    {
        pmsg = OSMemGet(CommMem, &err);
        if (pmsg != (INT8U *)0)
        {
            /*内存申请成功*/
        }
        //...用户代码
    }
}
```

11.2.3 OSMemPut ()

函数原型：

INT8U OSMemPut (OS_MEM *pmem, void *pblk)

函数功能：

函数释放一个内存块，内存块必须释放回原先申请的内存区。

函数参数：

OS_MEM *pmem：是指向内存区控制块的指针，可以从 **OSMemCreate ()** 函数 返回得到。

void *pblk：是指向将被释放的内存块的指针。

返回值：

- 1) **OS_ERR_NONE**：释放成功。
- 2) **OS_ERR_MEM_FULL**：释放的空间比申请的多。
- 3) **OS_ERR_MEM_INVALID_PMEM**：pmem 传递了一个空指针。
- 4) **OS_ERR_MEM_INVALID_PBLK**：pblk 传递了一个空指针。

说明：

必须首先建立内存区，然后使用。

示例：

```
OS_MEM *CommMem;
INT8U *CommMsg;
void Task (void *p_arg)
```

```

{
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        err = OSMemPut(CommMem, (void *)CommMsg);/*释放内存块*/
        if (err == OS_ERR_NONE)
        {
            /* 内存块释放成功 */
        }
        //...用户代码
    }
}

```

11.2.4 OSMemQuery ()

函数原型:

```
INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *p_mem_data)
```

函数功能:

函数得到内存区的信息。该函数返回 OS_MEM 结构包含的信息,但使用了一个新的 OS_MEM_DATA 的数据结构。OS_MEM_DATA 数据结构还包含了正被使用的内存块数目的域。

函数参数:

OS_MEM *pmem: 是指向内存区控制块的指针, 可以从 OSMemCreate () 函数 返回得到。

OS_MEM_DATA *p_mem_data: 是指向 OS_MEM_DATA 数据结构的指针, 该数据结构包含了以下的域:

Void OSAddr; /*指向内存区起始地址的指针*/

Void OSFreeList; /*指向空闲内存块列表起始地址的指针*/

INT32U OSBlkSize; /*每个内存块的大小*/

INT32U OSNBkls; /*该内存区的内存块总数*/

INT32U OSNFree; /*空闲的内存块数目*/

INT32U OSNUsed; /*使用的内存块数目*/

返回值:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_MEM_INVALID_PMEM: pmem 传递了一个空指针。
- 3) OS_ERR_MEM_INVALID_PBLK: pblk 传递了一个空指针。

说明:

必须首先建立内存区, 然后使用。

示例:

```

OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U err;
    OS_MEM_DATA mem_data;
    p_arg=p_arg;
    while(1)
    {
        //...用户代码
        err = OSMemQuery(CommMem, &mem_data);/*获取内存分区信息*/
        //...用户代码
    }
}

```

```
}  
}
```

11.2.5 OSMemNameGet ()

函数原型:

```
INT8U OSMemNameGet (OS_MEM *pmem, INT8U *pname, INT8U *perr)
```

函数功能:

获取存储分区的名称。

函数参数:

OS_MEM *pmem: 指向存储器分区。

INT8U *pname: 指向一个要存放 ASCII 字符串内存区的首地址。

INT8U *perr: 指向包含错误码的变量的指针, 并且可以是任何以下的:

- 1) OS_ERR_NONE: 获取成功。
- 2) OS_ERR_INVALID_PMEM: pmem 传递一个 NULL 指针。
- 3) OS_ERR_PNAME_NULL: pname 传递一个 NULL 指针。
- 4) OS_ERR_NAME_GET_ISR: 尝试从 ISR 使用本功能, 不允许。

返回值:

获取到的 pname 字符串的长度或者 0。

说明:

必须首先建立内存区, 然后使用。

示例:

```
OS_MEM *CommMem;  
INT8U *CommMemName;  
void Task (void *pdata)  
{  
    INT8U err;  
    INT8U size;  
    pdata = pdata;  
    while (;;)   
    {  
        /*获取内存分区名称*/  
        size = OSMemNameGet(CommMem, &CommMemName, &err);  
        //...用户代码  
    }  
}
```

11.2.6 OSMemNameSet ()

函数原型:

```
void OSMemNameSet (OS_MEM *pmem, INT8U *pname, INT8U *perr)
```

函数功能:

设置存储分区名称。

函数参数:

OS_MEM *pmem: 指向存储器分区。

INT8U *pname: 指向一个 ASCII 字符串的首地址。

INT8U *perr: 指向包含错误码的变量的指针, 并且可以是任何以下的:

- 1) OS_ERR_NONE: 设置成功。
- 2) OS_ERR_INVALID_PMEM: pmem 传递一个 NULL 指针。
- 3) OS_ERR_PNAME_NULL: pname 传递一个 NULL 指针。

4) OS_ERR_MEM_NAME_TOO_LONG: 设置的内存分区名称长度太长。

5) OS_ERR_NAME_GET_ISR: 尝试从 ISR 使用本功能, 不允许。

返回值:

无。

说明:

必须首先建立内存区, 然后使用。

示例:

```
OS_MEM *CommMem;
void Task (void *p_arg)
{
    INT8U err;
    p_arg=p_arg;
    while(1)
    {
        OSMemNameSet(CommMem, "Comm. Buffer", &err);
        //...用户代码
    }
}
```

11.3 μ C/OS- II 内存管理编程思想

1. 任务的创建与申请

设计 2 个任务, 任务 Start 用于创建其他任务并且创建内存分区。任务 KEY 功能是扫描按键, 当按键按下时申请一个内存块。

2. 任务的释放与当前信息

设计 2 个任务, 任务 Start 用于创建其他任务并且创建内存分区。任务 KEY 功能是扫描按键, 当按键按下 up 键时申请一个内存块, 当按键按下 down 键时释放一个内存块, 当按键按下 right 键时获取内存块的当前信息。

11.4 μ C/OS- II 内存管理编程示例

本实验部分源码如下, 完整的工程详见工程示例代码:

1、在 uc0s-task.c 中首先定义一个内存控制块, 如下:

```
OS_MEM *osmen; //内存控制块指针
INT8U mem_buffer[10][32]; //内存控制块
```

2、在 main.c 中创建任务 Start_Task(), 然后在 uc0s-task.c 开始任务 Start_Task()中调用 OSMemCreate()函数创建一个内存块, 代码如下:

```
//创建一个内存分区
void Start_Task(void *p_arg)
{
    INT8U perr;
    p_arg=p_arg;
    //创建一个内存分区, 有 10 个内存块, 每个内存块的大小为 32 个字节
    osmen=OSMemCreate(mem_buffer,10,32,&perr);
    if(perr == OS_ERR_NONE)
    {
        printf("成功创建内存分区\r\n");
    }
    OSTaskCreate(Led_Task,NULL, &Led_Task_Stk[Led_Stk_Size - 1],Led_Task_Pro); //创建一个名为 Led_Task 的任务
    OSTaskCreate(Key_Task,NULL, &Key_Task_Stk[Key_Stk_Size - 1],Key_Task_Pro); //创建一个名为 Key_Task 的任务
```

```
while (1)
{
    OSTaskSuspend(Start_Task_Pro); //把任务挂起
}
}
```

3、任务的申请与释放

在 ucos-task.c 中设计 2 个任务, 任务 Start 用于创建其他任务并且创建内存分区。任务 KEY 功能是扫描按键, 当按键按下 up 键时申请一个内存块, 当按键按下 down 键时释放一个内存块, 当按键按下 right 键时获取内存块的当前信息。

//任务函数

```
static INT8U memget_num=0;
u8 memget_num_buf[30];
void Key_Task(void *p_arg)
{
    u8 key;
    INT8U perr;
    INT8U *mem;
    OS_MEM_DATA mem_data;
    INT8U OSBlkSize_Buf[30]={0};
    INT8U OSNBlks_Buf[30]={0};
    INT8U OSNFree_Buf[30]={0};
    INT8U OSNUsed_Buf[30]={0};
    p_arg=p_arg;
    while (1)
    {
        key = KEY_Scan(0);
        switch(key)
        {
            case up_press: //当 up 键按下,申请一个内存块
                mem=OSMemGet(osmen,&perr);
                if(perr == OS_ERR_NONE)
                {
                    printf("成功申请到内存空间, 其地址为 0x%x\r\n",(u32)mem);
                    LCD_Show_String(30,150,280,16,BLUE,WHITE," Get Memory OK");
                    memget_num++;
                    printf("内存申请的次数为:%d 次\r\n",memget_num);
                    sprintf((char*)memget_num_buf,"Memory Use cnt : %d",memget_num);
                    LCD_Show_String(30,170,280,16,BLUE,WHITE,memget_num_buf);
                }
                if(perr ==OS_ERR_MEM_NO_FREE_BKLS)
                {
                    printf("内存不足\r\n");
                    LCD_Show_String(30,190,280,16,BLUE,WHITE,"there is not enough memory");
                }
                break;
            case down_press: //当 down 按下
                if(mem != NULL )
                {
```

```

        perr=OSMemPut(osmen,mem);
        if(perr == OS_ERR_NONE)
        {
            printf("成功释放内存空间，其地址为 0x%x\r\n",(u32)mem);
            LCD_Show_String(30,150,280,16,BLUE,WHITE," Put Memory OK");
            memget_num--;
            printf("内存申请的次数为:%d 次\r\n",memget_num);
            sprintf((char*)memget_num_buf,"Memory Use cnt : %d",memget_num);
            LCD_Show_String(30,170,280,16,BLUE,WHITE,memget_num_buf);
        }
    }
    break;
case right_press:
    if(mem != NULL )
    {
        perr=OSMemQuery(osmen,&mem_data);
        if(perr == OS_ERR_NONE)
        {
            printf("内存区的起始地址为: 0x%x\r\n",(u32)mem_data.OSAddr);
            printf("内存区的空闲区起始地址为: 0x%x\r\n",(u32)mem_data.OSFreeList);
            printf("内存块的大小为: %d \r\n",(u32)mem_data.OSBlkSize);
            printf("内存块的总数量为: %d \r\n",(u32)mem_data.OSNBlks);
            printf("内存块的空闲数量为: %d \r\n",(u32)mem_data.OSNFree);
            printf("内存块的已用数量为: %d \r\n",(u32)mem_data.OSNUSED);

            //LCD 显示
            sprintf((char*)OSBlkSize_Buf,"OSBlkSize :%d",mem_data.OSBlkSize);
            sprintf((char*)OSNBlks_Buf, "OSNBlks    :%d",mem_data.OSNBlks);
            sprintf((char*)OSNFree_Buf, "OSNFree    :%d",mem_data.OSNFree);
            sprintf((char*)OSNUSED_Buf, "OSNUSED    :%d",mem_data.OSNUSED);

            LCD_Show_String(30,210,280,16,BLUE,WHITE,OSBlkSize_Buf);
            LCD_Show_String(30,230,280,16,BLUE,WHITE,OSNBlks_Buf);
            LCD_Show_String(30,250,280,16,BLUE,WHITE,OSNFree_Buf);
            LCD_Show_String(30,270,280,16,BLUE,WHITE,OSNUSED_Buf);
        }
    }
    break;
}
OSTimeDly(10);
}
}

```

第十二章 μ C/OS-II 其他功能函数

12.1 μ C/OS-II 其他功能函数

12.1.1 OSStatInit ()

函数原型:

```
void OSStatInit (void)
```

函数功能:

统计任务初始化。

OSStatInit()获取当系统中没有其他任务运行时, 32 位计数器所能达到的最大值。OSStatInit()的调用时机是多任务环境已经启动, 且系统中只有一个任务在运行。也就是说, 该函数只能在第一个被建立并运行的任务中调用。

在使用本函数时, OS_TASK_STAT_EN 必须置为 1。这时候 uCOS-II 会自动创建一个统计任务 OSTaskStat()。这个统计任务每秒计算一次 CPU 在单位时间内被使用的时间, 并把计算结果以百分比的形式存放在变量 OSCPUUsage 中, 以便应用程序通过访问它来了解 CPU 的利用率。

函数参数:

无。

返回值:

无。

说明:

如果用户应用程序要使用统计任务, 则必须把定义在系统头文件 OS_CFG.H 中的系统配置常量 OS_TASK_STAT_EN 设置为 1, 并且必须在创建统计任务之前调用函数 OSStatInit()对统计任务进行初始化。

示例:

```
void main (void)
{
    //.....用户代码
    OSInit(); // 初始化 uC/OS-II
    //.....用户代码 // 最少创建一个任务
    OSStart(); //启动多任务内核
}
```

12.1.2 OSIntEnter ()

函数原型:

```
void OSIntEnter (void)
```

函数功能:

通知 uC/OS-II 一个中断处理函数正在执行, 这有助于 uC/OS-II 掌握中断嵌套的情况。OSIntEnter()

函数通常和 OSIntExit()函数联合使用。

函数参数:

无。

返回值:

无。

说明:

- 1) 在任务级不能调用该函数。
- 2) 如果用户使用的微处理器有存储器直接加 1 的单条指令的话, 将全程变量 OSIntNesting 直接加 1, 可以避免调用函数所带来的额外的开销!

3) 如果用户使用的微处理器没有这样的指令, 必须先将 `OSIntNesting` 读入寄存器, 再将寄存器加 1, 然后再写回到变量 `OSIntNesting` 中去, 就不如调用 `OSIntEnter()`。 `OSIntNesting` 是共享资源。 `OSIntEnter()` 把上述三条指令用开中断、关中断保护起来, 以保证处理 `OSIntNesting` 时的排它性。直接给 `OSIntNesting` 加 1 比调用 `OSIntEnter()` 快得多, 可能时, 直接加 1 更好。当心的是, 在有些情况下, 从 `OSIntEnter()` 返回时, 会把中断开了。这种情况, 在调用 `OSIntEnter()` 之前要先清中断源, 否则中断将连续反复打入, 用户应用程序就会崩溃!

示例:

```
void USART1_IRQHandler(void)
{
    OSIntEnter();
    //用户代码
    OSIntExit ()
}
```

12.1.3 OSIntExit ()

函数原型:

```
void OSIntExit (void)
```

函数功能:

通知 uC/OS-II 一个中断服务已执行完毕, 这有助于 uC/OS-II 掌握中断嵌套的情况。通常 `OSIntExit()` 和 `OSIntEnter()` 联合使用。当最后一层嵌套的中断执行完毕后, 如果有更高优先级的任务准备就绪, uC/OS-II 会调用任务调度函数, 在这种情况下, 中断返回到更高优先级的任务而不是被中断了的任务。

函数参数:

无。

返回值:

无。

说明:

在任务级不能调用该函数。并且即使没有调用 `OSIntEnter()` 而是使用直接递增 `OSIntNesting` 的方法, 也必须调用 `OSIntExit()` 函数。

示例:

```
void USART1_IRQHandler(void)
{
    OSIntEnter();
    //用户代码
    OSIntExit ()
}
```

12.1.4 OS_ENTER_CRITICAL (), OS_EXIT_CRITICAL ()

函数原型:

无。

函数功能:

`OS_ENTER_CRITICAL()` 和 `OS_EXIT_CRITICAL()` 为定义的宏, 用来关闭、打开 CPU 的中断。

函数参数:

`OS_ENTER_CRITICAL()` 进入临界区, 禁止被中断打断。

`OS_EXIT_CRITICAL()` 退出临界区, 允许被中断打断。

返回值:

无。

说明:

OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()必须成对使用。

示例:

```
void TaskX(void *pdata)
{
    for (;;) {
        OS_ENTER_CRITICAL(); /* 关闭中断 */
        //... 用户代码 /* 进入核心代码 */
        OS_EXIT_CRITICAL(); /* 打开中断 */
    }
}
```

12.1.5 OSSchedLock ()

函数原型:

```
void OSSchedLock (void)
```

函数功能:

函数停止任务调度，只有使用配对的函数 OSSchedUnlock()才能重新开始内核的任务调度。调用 OSSchedLock()函数的任务独占 CPU，不管有没有其他高优先级的就绪任务。在这种情况下，中断仍然可以被接受和执行（中断必须允许）。OSSchedLock()函数和 OSSchedUnlock()函数必须配对使用。uC/OS-II 可以支持多达 254 层的 OSSchedLock()函数嵌套，必须调用同样次数的 OSSchedUnlock()函数才能恢复任务调度。

函数参数:

无。

返回值:

无。

说明:

任务调用了 OSSchedLock () 函数后，决不能再调用可能导致当前任务挂起的系统函数：OSTimeDly ()，OSTimeDlyHMSM ()，OSSemPend ()，OSMboxPend ()，OSQPend ()。因为任务调度已经被禁止，其他任务不能运行，这会导致系统死锁。

示例:

```
void TaskX(void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSSchedLock(); /* 停止任务调度 */
        . /* 不允许被打断的执行代码 */
        OSSchedUnlock(); /* 恢复任务调度 */
    }
}
```

12.1.6 OSSchedUnlock ()

函数原型:

```
void OSSchedUnlock (void)
```

函数功能:

在调用了 OSSchedLock () 函数后，OSSchedUnlock () 函数恢复任务调度。

函数参数:

无。

返回值:

无。

说明:

任务调用了 OSSchedLock () 函数后, 决不能再调用可能导致当前任务挂起的系统函数: OSTimeDly (), OSTimeDlyHMSM (), OSSemPend (), OSMboxPend (), OSQPend ()。因为任务调度已经被禁止, 其他任务不能运行, 这会导致系统死锁。

示例:

```
void TaskX(void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSSchedLock(); /* 停止任务调度 */
        /* 不允许被打断的执行代码 */
        OSSchedUnlock(); /* 恢复任务调度 */
    }
}
```

12.1.7 OSVersion ()

函数原型:

INT16U OSVersion (void)

函数功能:

获取当前 uC/OS-II 的版本号。

函数参数:

无。

返回值:

返回的是版本号*100 。

说明:

无。

示例:

```
void TaskX(void *pdata)
{
    INT16U os_version;
    for (;;) {
        os_version = OSVersion(); /* 获取 uC/OS-II's 的版本 */
    }
}
```

12.1.8 OSTimeGet()

函数原型:

INT32U OSTimeGet(void);

函数功能:

获取当前系统时钟节拍数值

函数参数:

无。

返回值:

当前时钟计数 (时钟节拍数)。

说明:

无。

示例:

```
void TaskX(void *pdata)
{
    INT32U clk;
    While(1)
    {
        clk = OSTimeGet(); /* 获取当前系统时钟的值 */
    }
}
```