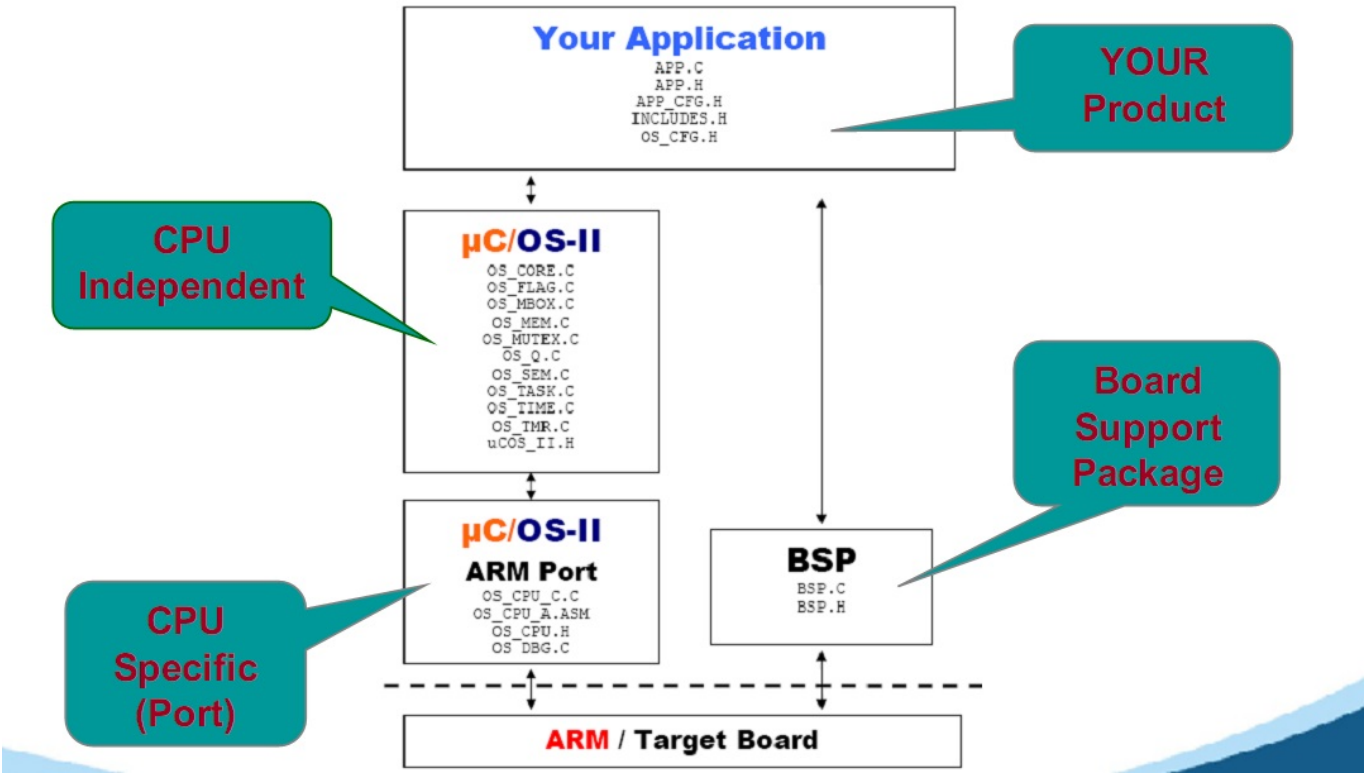


ucos ii system

文件结构

上层:

应用软件，用户代码

中层:

1. 与处理器无关代码

OS_CORE.C	系统初始化，开启多任务环境等的代码
OS_CPU_C.C	多任务栈初始化等与处理器有关的代码
OS_FLAG .C	事件标志组管理代码
OS_MBOX .C	消息邮箱管理代码
OS_mem.c	内存管理代码
OS_mutex.c	互斥型信号量管理代码
OS_q.c	消息队列管理
OS_sem.c	信号量管理代码
OS_task.c	任务管理代码
OS_time.c	事件管理代码
uCOS_II.C	包含内核的其它C语言源文件

2. 与应用程序相关配置文件

INCLUDES.H	系统的全局头文件，在所有的源码中包含
OS_CFG.H	UCOS系统的全局配置

3. 与处理器有关代码

OS_CPU.h	包含与处理器相关的常量、宏及结构体定义
OS_CPU_C.C	多任务栈初始化等与处理器有关的代码
OS_CPU_A.asm	汇编语言编写的启动任务、任务切换等四个重要函数

下层：

硬件(cpu,interupt,timer,gpio,iis...)

内核结构

ucos的内核机构可以从以下的代码可以看出,应用支持10个事件控制块,5个事件标志组,5个内存区块,4个队列控制块和20个任务,最低优先级为63,任务堆栈大小都为128等等,这些都是可以在OS_CFG.H中自行定义的.

```
#define OS_LOWEST_PRIO           63u   /* Defines the lowest priority that can be assigned ...         */
                                       /* ... MUST NEVER be higher than 254!                           */

#define OS_MAX_EVENTS            10u   /* Max. number of event control blocks in your application      */
#define OS_MAX_FLAGS              5u   /* Max. number of Event Flag Groups    in your application      */
#define OS_MAX_MEM_PART           5u   /* Max. number of memory partitions                             */
#define OS_MAX_QS                 4u   /* Max. number of queue control blocks in your application      */
#define OS_MAX_TASKS             20u   /* Max. number of tasks in your application, MUST be >= 2       */

#define OS_SCHED_LOCK_EN          1u   /* Include code for OSSchedLock() and OSSchedUnlock()           */

#define OS_TICK_STEP_EN           1u   /* Enable tick stepping feature for uC/OS-View                  */
#define OS_TICKS_PER_SEC        100u   /* Set the number of ticks in one second                        */


                                       /* --------------------- TASK STACK SIZE ---------------------- */
#define OS_TASK_TMR_STK_SIZE    128u   /* Timer      task stack size (# of OS_STK wide entries)        */
#define OS_TASK_STAT_STK_SIZE   128u   /* Statistics task stack size (# of OS_STK wide entries)        */
#define OS_TASK_IDLE_STK_SIZE   128u   /* Idle       task stack size (# of OS_STK wide entries)        */
```

临界段

处理器处理临界代码都必须先关中断，再处理临界代码，然后再开中断。关中断时间对实时系统的实时响应很重要。所以是实时系统的一个很重要的指标。uCOS使用两个宏（在OS_CPU.h中定义。注：没个CPU都有自己的OS_CPU.h）。这两个宏分别为OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL()关闭中断和打开中断. 列:

```
void function(void)
{
    OS_ENTER_CRITICAL(); //关闭中断
    /*uCOS II 临界代码段*/
    OS_EXIT_CRITICAL(); //打开中断
}
```

注:在ODTimeDel()之类的函数调用的时候不能关闭中断，不然应用程序会死机.

任务

> 任务由任务控制块,任务堆栈和任务代码三部分构成.系统通过任务控制块来感知和控制任务,任务堆栈主要用来保存和恢复断点,任务代码是一个超循环,它描述了任务的执行过程.

> 通常任务是一个无限循环。函数没有返回值。任务完成以后可以自我删除。(注意：删除不是任务代码删除了，只是这个任务不会再执行了；即使调用了OSTaskDel()这个任务也不会有返回值).

```
                                        /* ---------------------- TASK MANAGEMENT ----------------------- */
#define OS_TASK_CHANGE_PRIO_EN    1u    /*     Include code for OSTaskChangePrio()                       */
#define OS_TASK_CREATE_EN         1u    /*     Include code for OSTaskCreate()                           */
#define OS_TASK_CREATE_EXT_EN     1u    /*     Include code for OSTaskCreateExt()                        */
#define OS_TASK_DEL_EN            1u    /*     Include code for OSTaskDel()                              */
#define OS_TASK_NAME_EN           1u    /*     Enable task names                                         */
#define OS_TASK_PROFILE_EN        1u    /*     Include variables in OS_TCB for profiling                 */
#define OS_TASK_QUERY_EN          1u    /*     Include code for OSTaskQuery()                            */
#define OS_TASK_REG_TBL_SIZE      1u    /*     Size of task variables array (#of INT32U entries)         */
#define OS_TASK_STAT_EN           1u    /*     Enable (1) or Disable(0) the statistics task              */
#define OS_TASK_STAT_STK_CHK_EN   1u    /*     Check task stacks from statistic task                     */
#define OS_TASK_SUSPEND_EN        1u    /*     Include code for OSTaskSuspend() and OSTaskResume()       */
#define OS_TASK_SW_HOOK_EN        1u    /*     Include code for OSTaskSwHook()                           */
```

任务创建:

可以使用OSTaskCreat()或者OSTaskCreatExt()创建. 这两个函数负责分配任务控制块和任务堆栈,并初始化他们,然后把任务控制块,任务堆栈和任务代码关联起来成为一个完整的任务.
uCOS II 可以管理的任务可以达到64个，但是建议不要使用前四个优先级的任务和后四个优先级的任务.

```
int  main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);      /* Clear the screen                         */

    OSInit();                                              /* Initialize uC/OS-II                      */

    RandomSem    = OSSemCreate(1);                         /* Random number semaphore                  */

    OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();                                             /* Start multitasking                       */
    return 0;
}
```

这里OSTaskCreate创建了一个任务TeskTask1,传给任务的参数为空(void *)0,栈顶地址为&TaskStartStk[TASK_STK_SIZE - 1],优先级为0,创建第一个任务前需要先初始化系统OSInit(),创建完任务后就可以调用OSStart()开始多任务.

```
/*
*********************************************************************************************************
*                                          First Task (startup task)
*********************************************************************************************************
*/
void TestTask1(void *pdata)
{   printf("%4u: ***** Test Task 1 First call *****\n", OSTime);

#if OS_TASK_STAT_EN > 0
    OSStatInit();                                                       //Initialize the statistics task
#endif

    OSTaskCreate(TestTask2, (void *) 22, &TestTaskStk2[TASK_STK_SIZE], 22);      //Create 3 other tasks
    OSTaskCreate(TestTask3, (void *) 33, &TestTaskStk3[TASK_STK_SIZE], 33);
    OSTaskCreate(TestTask4, (void *) 10, &TestTaskStk4[TASK_STK_SIZE], 10);

    while (1)
    {   printf("%4u: ***** Test Task 11 *****\n", OSTime);

#ifdef SUSPEND_RESUME
        OSTaskSuspend(OS_PRIO_SELF);                                    //Calling sequence -->OSTaskSwHook-->OSCtxSw
#else
        OSTimeDly(1);                                                   //Calling sequence -->OSTaskSwHook-->OSCtxSw
#endif
    }
} ? end TestTask1 ?
```

TeskTask1首先初始化统计任务,然后依次创建三个任务:TestTask2,TestTask3,TestTask4.然后进入while循环.任务永远不会退出,但可以通过OSTimeDly或者OSTaskSuspend挂起.

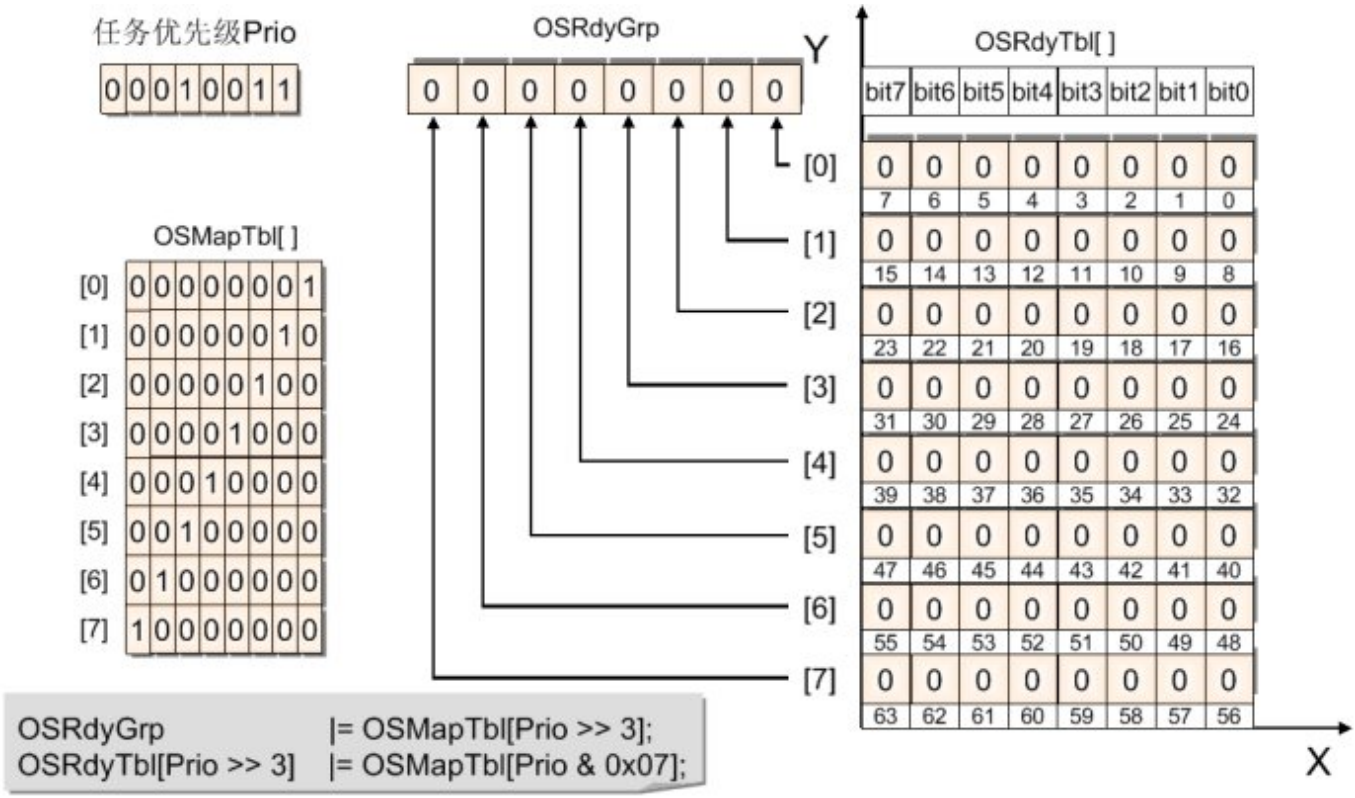
任务状态:

1．睡眠：驻留在ROM或者RAM中，系统还没有管理，只有通过`OSTaskCreat()`或`OSCreatExt()`创建之后才能使得系统管理任务．
2．就绪：任务一旦建立就进入了就绪态，等待运行．
3．等待：可以调用`OSTimeDel()`或者`OSTimeDlyHMSM()`使得任务进入等待状态。一直等待函数中定义的延时时间到了，这两个函数会强制执行任务转换，让下一个优先级更高的任务进入就绪态的任务运行．
4．运行：当前任务正在执行．
5．中断：当前正在执行的任务被中断，进入中断服务态，响应中断时该任务被挂起。中断服务子程序占有了CPU的使用权．

任务控制块(TCB)：

系统会根据OS_MAX_TASKS + OS_N_SYS_TASKS确定任务控制表OSTCBTbl[]的大小并初始化为空. 重要的数据结构,一旦任务建立了，任务控制块OS_TCB将被赋值，ucosii用它保存任务的状态，用来恢复任务. 任务建立的时候，OS_TCBs就被初始化了. 关键的结构体变量:
1. **OSTCBStkPtr**: 当前任务栈顶的指针.是OS_TCB数据结构中唯一的一个能用汇编语言来处置的变量（在任务切换段的代码中）,把OSTCBStkPtr放在数据结构的最前面，使得从汇编语言中处理这个变量时较为容易.
2. **OSTCBExtPtr**: 指向用户定义的任务控制块扩展.用户可以扩展任务控制块而不必修改μC/OS-II的源代码.
3. **OSTCBStkBottom**: 指向任务栈底的指针.函数OSTaskStkChk()(用于堆栈检验)要用到变量OSTCBStkBottom,在运行中检验栈空间的使用情况。用户可以用它来确定任务实际需要的栈空间.这个功能只有当用户在任务建立时允许使用OSTaskCreateExt()函数时才能实现。这就要求用户将OS_TASK_CREATE_EXT_EN设为1以便允许该功能.
4. **OSTCBStkSize**: 存有栈中可容纳的指针元数目，而不是用字节表示的栈容量总数.

任务就绪表：



每个就绪的任务都放在任务就绪表中。就绪表中有两个变量，OSRdyGrp和OSRdyTbl[OS_RDY_TBL_SIZE],在

OSRdyGrp中任务按照优先级分组，8个任务为一组。OSRdyGrp中的每位表示8组任务中每一组是否有进入就绪态的任务,任务就绪，OSRdyTbl[OS_RDY_TBL_SIZE]中相应元素中的相应位也被置1.OSRdyTbl[OS_RDY_TBL_SIZE]数组有多大取决于OS_LOWSET_PRIO.当应用程序的数目比较少的时候可以降低OS_LOWSET_PRIO，可以降低系统对RAM（数据空间）的需求. 就绪表的定义如下,大小由最低优先级确定.

```
#define  OS_RDY_TBL_SIZE   ((OS_LOWEST_PRIO) / 16u + 1u)/* Size of ready table                      */
```

系统通过OSRdyGrp和OSRdyTbl[OSUnMapTbl[OSRdyGrp]]来确定就绪表中的最高优先级任务.

```
y             = OSUnMapTbl[OSRdyGrp];
OSPrioHighRdy = (INT8U)((y << 3u) + OSUnMapTbl[OSRdyTbl[y]]);
```

优先级判定表OSUnMapTbl[]定义如下:

```
/*
*********************************************************************************************************
*                                     PRIORITY RESOLUTION TABLE
*
* Note: Index into table is bit pattern to resolve highest priority
*       Indexed value corresponds to highest priority bit position (i.e. 0..7)
*********************************************************************************************************
*/

INT8U  const  OSUnMapTbl[256] = {
    0u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x00 to 0x0F                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x10 to 0x1F                  */
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x20 to 0x2F                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x30 to 0x3F                  */
    6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x40 to 0x4F                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x50 to 0x5F                  */
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x60 to 0x6F                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x70 to 0x7F                  */
    7u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x80 to 0x8F                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x90 to 0x9F                  */
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xA0 to 0xAF                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xB0 to 0xBF                  */
    6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xC0 to 0xCF                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xD0 to 0xDF                  */
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xE0 to 0xEF                  */
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u  /* 0xF0 to 0xFF                  */
};
```

任务调度:

> 根据任务优先级,UCOS总是让就绪表中任务优先级最高的任务先执行.
> 任务的调度由函数：OSSched()完成。中断级的调用由另一个函数：OSIntExt()完成。

任务切换:

> 任务调度器决定哪个任务该运行了，然后由IS_TASK_SW()函数做任务切换.OS_TASK_SW()是一个宏调用。含有处理器的软中断指令.
> 任务切换的核心工作是任务堆栈指针的切换.

任务调度器代码的设计,使得他的运行时间与系统中的任务数无关,从而使它满足了实时系统的要求.

注:根据任务是否有自己的私有空间,人们把任务分为进程与线程,有私有空间的任务叫进程,而没有私有空间的任务叫线程.ucosii没有为任务分配私有空间,因此ucosii中的所有任务都属于线程.这也是ucosii的缺陷之一,linux系统有进程.

调度

上锁，开锁

空闲任务

优先级最低，必须存在

统计任务

优先级次低，选择
每秒运行一次计算CPU利用率，精度1%。

中断

时钟节拍

提供周期性信号源，用于时间延迟和确认超时。 ###流程

```
OSInit();//ucos的初始化,空闲任务，统计任务，系统变量及数据结构
OSTaskCreate();//任务创建
OSStart();//ucos的启动,必须建立一个任务
```

时间管理

```
                                    /* ---------------------- TIME MANAGEMENT ----------------------- */
#define OS_TIME_DLY_HMSM_EN         1u   /*     Include code for OSTimeDlyHMSM()                       */
#define OS_TIME_DLY_RESUME_EN       1u   /*     Include code for OSTimeDlyResume()                     */
#define OS_TIME_GET_SET_EN          1u   /*     Include code for OSTimeGet() and OSTimeSet()           */
#define OS_TIME_TICK_HOOK_EN        1u   /*     Include code for OSTimeTickHook()                      */
```

时间管理的内容在代码os_time.c中，包含操作系统时间的设置及获取，对任务的延时，任务按分秒延时，取消任务的延时共5个系统调用。时间管理的最主要功能就是对任务进行延时。 时间管理中最重要的数据结构就是全局变量OSTime，OSTime的值就是操作系统的时间，它的定义在uC/OS-II的头文件ucos_ii.h中:

```
#if OS_TIME_GET_SET_EN > 0u
OS_EXT  volatile  INT32U  OSTime;            /* Current value of system time (in ticks)       */
#endif
```

其中关键字volatile总是与优化有关,volatile意味着禁止对变量进行优化.因为OSTime的值是易变的，加了关键字volatile后，不会被编译器优化，每次取值都会直接在内存中对该变量的地址取值，从而保证不会因为编译器优化而产生错误的结果.
OSTime在操作系统初始化时被设置为0。
时间管理中使用的另一个重要的数据结构就是任务控制块，任务控制块有一项是OSTCBDly，标志这个任务延时的时间。这个时间是以两次时钟中断间隔的时间为单位的。另外，对任务的延时实际上阻塞了任务，因此要对就绪表和就绪组等数据结构进行相关的操作.
时间的设置和获取都是关于OSTime的赋值，代码比较简单，如下所示:

```
#if OS_TIME_GET_SET_EN > 0u
INT32U OSTimeGet (void)
{
    INT32U    ticks;
#if OS_CRITICAL_METHOD == 3u                 /* Allocate storage for CPU status register         */
    OS_CPU_SR  cpu_sr = 0u;
#endif

    OS_ENTER_CRITICAL();
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}
#endif
```

需要注意的是，对OSTime的操作一定要使用临界区。时间设置函数将参数ticks的值赋值给OSTime，这两个函数并不常用.

任务延时函数OSTimeDly用于阻塞任务一定时间，这个时间以参数的形式给出。如果这个参数的值是N，那么在N个时间片(时钟滴答)之后，任务才能回到就绪态获得继续运行的机会。如果参数的值是0，就不会阻塞任务。任务延时函数OSTimeDly的代码如下所示:

```
void OSTimeDly (INT32U ticks)
{
    INT8U     y;
#if OS_CRITICAL_METHOD == 3u                 /* Allocate storage for CPU status register         */
    OS_CPU_SR  cpu_sr = 0u;
#endif

    if (OSIntNesting > 0u)
    {                /* See if trying to call from an ISR               */
        return;
    }
    if (OSLockNesting > 0u)
    {                /* See if called with scheduler locked             */
        return;
    }
    if (ticks > 0u)
    {                        /* 0 means no delay!                                  */
        OS_ENTER_CRITICAL();
        y          = OSTCBCur->OSTCBY;       /* Delay current task                                 */
        OSRdyTbl[y] &= (OS_PRIO)~OSTCBCur->OSTCBBitX;
        if (OSRdyTbl[y] == 0u)
        {
            OSRdyGrp &= (OS_PRIO)~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;            /* Load ticks in TCB                                  */
        OS_EXIT_CRITICAL();
        OS_Sched();                          /* Find next task to run!                             */
    }
} ?  end OSTimeDly ?
```

代码清晰,OSLockNesting是调度锁，也就是说，如果OSLockNesting>0，那么不允许进行任务调度.因为任务延时的时候要中止当前任务的执行，所以要进行调度，因此在调度锁有效的情况下是不能执行任务延时的.如果延时时间大于0，那么就要进行一次任务调度，将当前的任务的就绪标志取消，也就是对就绪表和就绪组的相关操作.之后延时时间赋值给任务块的OSTCBDly项以对延时计数。操作系统在每个时钟中断都要对每个OSTCBDly大于0的任务的OSTCBDly进行减1操作和进行任务调度，那么当任务的延时时间到了的时候(OSTCBDly为0)就可以恢复到就绪态.

注:如果将任务延时1个时间片，调用OSTimeDly(1)，会不会产生正确的结果呢？回答是否定的。这是因为任务在调用时间延时函数的时候可能已经马上就要发生时间中断了，那么设置OSTCBDly的值为1，想延时10ms，然后系统切换到一个新的任务运行。在可能极短的时间，如0.5ms的时候就进入时钟中断服务程序，立刻将OSTCBDly的值减到0了。调度器在调度的时候就会恢复这个才延时了0.5ms的任务。可见，OSTimeDly的误差最大应该是1个时间片的长度，OSTCBDly(1)不会刚好延时10ms，如果真的需要延时一个时间片，最好调用OSTCBDly(2).

任务延时函数OSTimeDly用于将任务阻塞一段时间，这个时间是以时间片为单位的。如果想以时、分、秒、毫秒为单位进行任务延时，需要调用以分秒作为单位的任务延时函数`OSTimeDlyHMSM`.

任务在延时之后，进入阻塞态。当延时时间到了就从阻塞态恢复到就绪态，可以被操作系统调度执行。**但是**，并非回到就绪态就只有这么一种可能，因为即便任务的延时时间没到，还是可以通过函数`OSTimeDlyResume`恢

复该任务到就绪态的. 另外，OSTimeDlyResume也不仅仅能恢复使用OSTimeDly或OSTimeDlyHMSM而延时的任务。对于因等待事件发生而阻塞的，并且设置了超时(timeout)时间的任务，也可以使用OSTimeDlyResume来恢复。对这些任务使用了OSTimeDlyResume，就好像已经等待超时了一样。
但是，对于，采用OSTaskSuspend挂起的任务，是不允许采用OSTimeDlyResume来恢复的。

```
#if OS_TIME_DLY_RESUME_EN > 0u
INT8U OSTimeDlyResume (INT8U prio)
{
    OS_TCB    *ptcb;
#if OS_CRITICAL_METHOD == 3u                          /* Storage for CPU status register      */
    OS_CPU_SR  cpu_sr = 0u;
#endif

    if (prio >= OS_LOWEST_PRIO)
    {
        return (OS_ERR_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];                        /* Make sure that task exist            */
    if (ptcb == (OS_TCB *)0)
    {
        OS_EXIT_CRITICAL();
        return (OS_ERR_TASK_NOT_EXIST);                   /* The task does not exist            */
    }
    if (ptcb == OS_TCB_RESERVED)
    {
        OS_EXIT_CRITICAL();
        return (OS_ERR_TASK_NOT_EXIST);                   /* The task does not exist            */
    }
    if (ptcb->OSTCBDly == 0u)
    {                                                 /* See if task is delayed             */
        OS_EXIT_CRITICAL();
        return (OS_ERR_TIME_NOT_DLY);                 /* Indicate that task was not delayed */
    }
```

代码中一个非常重要的数据结构就是任务块的OSTCBStat，如下所示：

位：	7	6	5	4	3	2	1	0
	请求多事件	未用	请求事件标志组	请求互斥信号量	挂起	请求队列	请求邮箱	请求信号量

宏定义如下:

```
/*
*********************************************************************************************************
*                                  TASK STATUS (Bit definition for OSTCBStat)
*********************************************************************************************************
*/
#define OS_STAT_RDY           0x00u  /* Ready to run                                        */
#define OS_STAT_SEM           0x01u  /* Pending on semaphore                                */
#define OS_STAT_MBOX          0x02u  /* Pending on mailbox                                  */
#define OS_STAT_Q             0x04u  /* Pending on queue                                    */
#define OS_STAT_SUSPEND       0x08u  /* Task is suspended                                   */
#define OS_STAT_MUTEX         0x10u  /* Pending on mutual exclusion semaphore               */
#define OS_STAT_FLAG          0x20u  /* Pending on event flag group                         */
#define OS_STAT_MULTI         0x80u  /* Pending on multiple events                          */

#define OS_STAT_PEND_ANY      (OS_STAT_SEM | OS_STAT_MBOX | OS_STAT_Q | OS_STAT_MUTEX | OS_STAT_FLAG)
```

因此,如果一个任务只是设置了延时，那么该任务块的OSTCBStat的值应该是0，也就是OS_STAT_RDY.被设置了延时的任务和就绪任务的区别在于，就绪任务的控制块的OSTCBDly的值一定是0，而设置了延时的任务的OSTCBDly的值一定不是0. 如果一个任务在等待一个或多个事件的发生，那么该任务的控制块的0、1、2、4、5位必然有1位或多位不为0.也就是ptcb->OSTCBStat&OS_STAT_PEND_ANY的值不为0.这是在判断任务是不是在等待事件的发生。等待事件发生的任务可能设置了超时，也可能没有设置超时，如果没有设置超时那么就会在下面所示的代码返回：
![ostcbdly]./res/cvwl210.png)
所以不会被恢复到就绪态。设置了超时的OSTCBDly的值大于0，先将OSTCBDly的值置位为0，然后使用ptcb->OSTCBStat&=~OS_STAT_PEND_ANY，所以的5种事件等待标志全部强制清0，不再等待了。
另外，还需要判断OSTCBStat的位3挂起标志。因为被挂起的任务必须用也只能用OSTaskResume来恢复。
OS_STAT_SUSPEND的值是0x08，ptcb->OSTCBStat&OS_STAT_SUSPEND是将STCBStat的位3挂起标志位单独取

出来了，判断它是不是0，如果是0，那么就不是被挂起的任务，否则就是被挂起的任务。对于挂起的任务只能处理到这里，对于其他的任务就开始对就绪表和就绪组进行处理，恢复任务到就绪态，然后执行任务调度。

事件管理

主要函数

功能	信号量	互斥信号量	事件标志组	消息邮箱	消息队列
建立事件	OSSemCreate	OSMutexCreate	OSFlagCreate	OSMboxCreate	OSQCreate
删除事件	OSSemDel	OSMutexDel	OSFlagDel	OSMboxDel	OSQDel
等待事件	OSSemPend	OSMutexPend	OSFlagPend	OSMboxPend	OSQPend
发送事件	OSSemPost	OSMutexPost	OSFlagPost	OSMboxPost	OSQPost
无等待获得事件	OSSemAccept	OSMutexAccept	OSFlagAccept	OSMboxAccept	OSQAccept
查询事件状态	OSSemQuery	OSMutexQuery	OSFlagQuery	OSMboxQuery	OSQQuery

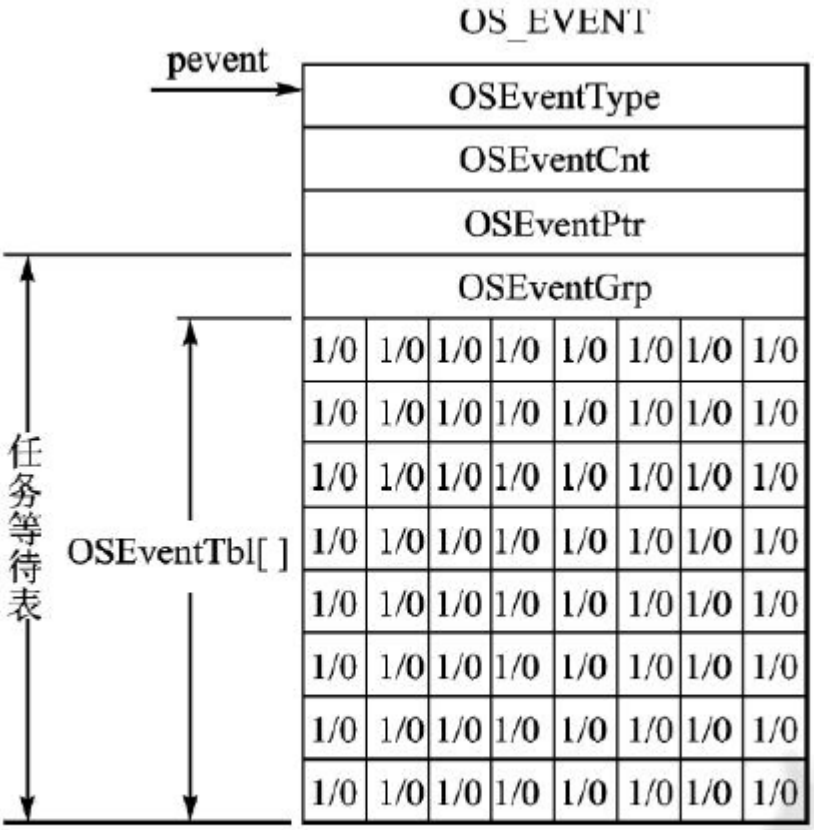


图 5－9 事件控制块 ECB 结构

事件控制块结构体(UCOS_II.H)

```
/*
*********************************************************************************************************
*                                          EVENT CONTROL BLOCK
*********************************************************************************************************
*/

#if OS_LOWEST_PRIO <= 63u
typedef  INT8U    OS_PRIO;
#else
typedef  INT16U   OS_PRIO;
#endif

#if (OS_EVENT_EN) && (OS_MAX_EVENTS > 0u)
typedef struct os_event {
    INT8U    OSEventType;                    /* Type of event control block (see OS_EVENT_TYPE_xxxx)    */
    void    *OSEventPtr;                     /* Pointer to message or queue structure                   */
    INT16U   OSEventCnt;                     /* Semaphore Count (not used if other EVENT type)          */
    OS_PRIO  OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
    OS_PRIO  OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                */

#if OS_EVENT_NAME_EN > 0u
    INT8U   *OSEventName;
#endif
} OS_EVENT;
#endif

#if (OS_EVENT_EN) && (OS_MAX_EVENTS > 0u)
OS_EXT  OS_EVENT         *OSEventFreeList;          /* Pointer to list of free EVENT control blocks    */
OS_EXT  OS_EVENT          OSEventTbl[OS_MAX_EVENTS];/* Table of EVENT control blocks                   */
#endif
```

ucos系统默认定义了**OS_MAX_EVENTS**这么多个事件控制块，**OSEventFreeList**为一个单项链表。同TCB,此时初始化的控制块没有与任何的具体事件相关联。

信号量管理

信号量数据结构

```
/*
*********************************************************************************************************
*                                          SEMAPHORE DATA
*********************************************************************************************************
*/

#if OS_SEM_EN > 0u
typedef struct os_sem_data {
    INT16U  OSCnt;                           /* Semaphore count                                         */
    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 */
} OS_SEM_DATA;
#endif
```

互斥信号量数据结构

```
/*
*********************************************************************************************************
*                                   MUTUAL EXCLUSION SEMAPHORE DATA
*********************************************************************************************************
*/

#if OS_MUTEX_EN > 0u
typedef struct os_mutex_data {
    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 */
    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;
#endif
```

消息管理

消息邮箱数据结构

```
/*
*********************************************************************************************************
*                                        MESSAGE MAILBOX DATA
*********************************************************************************************************
*/

#if OS_MBOX_EN > 0u
typedef struct os_mbox_data {
    void   *OSMsg;                          /* Pointer to message in mailbox                            */
    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  */
} OS_MBOX_DATA;
#endif
```

消息队列数据结构

```
/*
*********************************************************************************************************
*                                        MESSAGE QUEUE DATA
*********************************************************************************************************
*/

#if OS_Q_EN > 0u
typedef struct os_q {                   /* QUEUE CONTROL BLOCK                                          */
    struct os_q   *OSQPtr;              /* Link to next queue control block in list of free blocks      */
    void         **OSQStart;            /* Pointer to start of queue data                               */
    void         **OSQEnd;              /* Pointer to end   of queue data                               */
    void         **OSQIn;               /* Pointer to where next message will be inserted  in   the Q   */
    void         **OSQOut;              /* Pointer to where next message will be extracted from the Q   */
    INT16U         OSQSize;             /* Size of queue (maximum number of entries)                    */
    INT16U         OSQEntries;          /* Current number of entries in the queue                       */
} OS_Q;


typedef struct os_q_data {
    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      */
} OS_Q_DATA;
#endif
```

内存管理

内存块结构体

```
/*
*********************************************************************************************************
*                                      MEMORY PARTITION DATA STRUCTURES
*********************************************************************************************************
*/

#if (OS_MEM_EN > 0u) && (OS_MAX_MEM_PART > 0u)
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      */
#if OS_MEM_NAME_EN > 0u
    INT8U  *OSMemName;                     /* Memory partition name                                    */
#endif
} OS_MEM;


typedef struct os_mem_data {
    void   *OSAddr;                    /* Pointer to the beginning address of the memory partition     */
    void   *OSFreeList;                /* Pointer to the beginning of the free list of memory blocks   */
    INT32U  OSBlkSize;                 /* Size (in bytes) of each memory block                         */
    INT32U  OSNBlks;                   /* Total number of blocks in the partition                      */
    INT32U  OSNFree;                   /* Number of memory blocks free                                 */
    INT32U  OSNUsed;                   /* Number of memory blocks used                                 */
} OS_MEM_DATA;
#endif
```

ucos ii的移植

ucos ii给实时系统开发提供一个简易的框架，其实现基本的系统管理功能。

开发者需要做的是根据自己的平台修改和剪裁。

需要设计中修改的如处理器的位数，开中断和关中断实现等。

调通基本的系统，就可以在上面根据需要开发更多的功能。