本文档以 **ARM9**（三星 **2410/2440**）为平台，介绍一个多任务抢占式调度器------抢占式任务调度，提供延时,挂起,恢复任务操作。最精简化，没有加入信号量邮箱等同步通信机制。只实现一个基本任务调度器的功能。

虽然不能称为操作系统，但已体现了小型嵌入式操作系统的精髓。**OS** 代码不到 **1.5K**，核心函数只有几个，思路简单明了。比起 **UCOS**，更适合用作多任务系统原理的学习入门。对初学者来说，看 **UCOS** 的源代码很容易迷糊。

回想初学嵌入式多任务系统时，什么都不懂，**Jean J.Labrosse** 的经典之作《嵌入式实时操作系统 **uc/osII**》看得我一头雾水。事实上，使我对多任务的原理印象最深的是网上的一篇文章----《建立一个属于自己的 **AVR** 的 **RTOS**》。

学习就应该这样，循序渐进。把一步步把简单的东西弄懂了，便没有复杂的了，所谓水到渠成。

这篇文章是面对初学者的，把很多问题简化了。希望对刚接触嵌入式多任务系统的兄弟有所帮助。

必定存在不少 **bug**，欢迎指正。

# 简易多任务 OS 设计

## -------ARM9 上运行的简单多任务调度器

By lisuwei

http://group.ednchina.com/999/20520.aspx

这里提供一个简易的多任务抢占任务调度器供大家学习。虽然太简单不实用，但对理解多任务抢占式调度的原理是很有益处的。

先直观地看一下多任务系统中的代码与单任务程序（前后台系统）的区别。

*Main.c*

```
int Main(void)
{
    TargetInit();     //初始化目标板
    OSInit();         //初始化操作系统
    OSTaskCreate(Task0,&StackTask0[StackSizeTask0 - 1],PrioTask0); // 创建一个任务
    Uart_Printf("Ready to start OS\n");
    OSStart();        //运行操作系统
    return 0;         //程序不会运行至此
}

void Task0(void)
{
    TargetStart();    //设置中断向量，启动操作系统的硬件定时器中断
    Uart_Printf("Start OS\n");
    // 创建其他任务
    OSTaskCreate(Task1,&StackTask1[StackSizeTask1 - 1],PrioTask1);
    OSTaskCreate(Task2,&StackTask2[StackSizeTask2 - 1],PrioTask2);
    OSTaskCreate(Task3,&StackTask3[StackSizeTask2 - 1],PrioTask3);
    while(1)
    {
        Uart_Printf("Task0\n");
        OSTimeDly(100);       //1 秒运行一次
    }
}

void Task1(void)
{
    while(1)
    {
        Uart_Printf("Task1\n");
        OSTimeDly(300); //3 秒运行一次
    }
}
```

```
void Task2(void)
{
    while(1)
    {
        Uart_Printf("Task2\n");
        OSTaskSuspend(PrioTask2);   // 使自己进入挂起状态
    }
}

void Task3(void)
{
    while(1)
    {
        Uart_Printf("Resume Task2\n");
        OSTaskResume(PrioTask2);        // 恢复任务 2
        OSTimeDly(800);
    }
}
```

程序中创建了四个任务，任务 0 每 1 秒运行一次，任务 1 每 3 秒运行一次，任务 2 运行一次即把自己挂起，任务 3 每 8 秒运行一次并把任务 2 恢复。

在终端的运行结果如下图：

```
Download O.K.

Board is running OK
Ready to start OS
Start OS
Task0
Task1
Task2
Resume Task2
Task2
Task0
Task0
Task0
Task1
Task0
Task0
Task0
Task1
Task0
Task0
Resume Task2
Task2
Task0
Task1
Task0
```

## 什么是多任务系统？

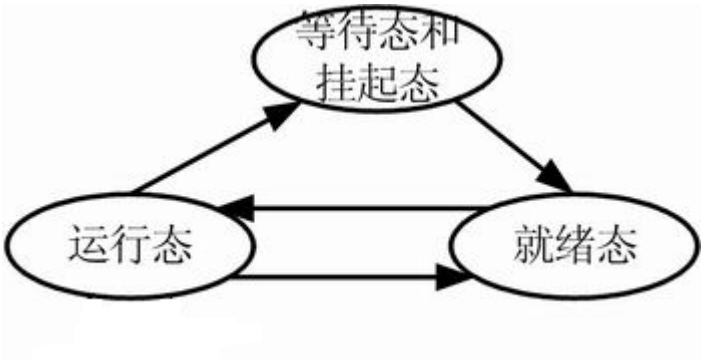
就像我们用电脑时可以同时听歌，上网，编辑文档等。在多任务系统中，可以同时执行多个并行任务，各个任务之间互相独立。通过操作系统执行任务调度而实现宏观上的“并发运行”。从宏观上不同的任务并发运行，好像每个任务都有自己的 CPU 一样。

其实在单一 CPU 的情况下，是不存在真正的多任务机制的，存在的只有不同的任务轮流使用 CPU，所以本质上还是单任务的。但由于 CPU 执行速度非常快，加上任务切换十分频繁并且切换的很快，所以我们感觉好像有很多任务同时在运行一样。这就是所谓的多任务机制。

多任务的最大好处是充分利用硬件资源，如在单任务时（大循环结构，如大部分 51 程序）遇到 delay 函数时，CPU 在空转；而在多任务系统，遇到 delay 或需等待资源时系统会自动运行下一个任务，等条件满足再回来运行先前的任务，这样就充分利用了 CPU，提高了效率。

任务有下面的特性：

- 动态性。任务并不是随时都可以运行的，而一个已经运行的任务并不能保证一直占有 CPU 直到运行完。一般有就绪态，运行态，挂起态等。
  运行态。一个运行态的任务是一个正在使用 CPU 的任务。任何时刻有且只有一个运行着的任务。
  就绪态。一个就绪态任务是可运行的，等待占有 CPU 的任务释放 CPU。
  挂起态。某些条件不满足而挂起不能运行的状态。




任务三种状态转换图

下面我们来分析代码是如何实现状态转换的。

***RTOS.h***

```
INT32U    OSRdyTbl;          /*  就绪任务表    */
```

上面定义一个 32 位变量，每一位代表一个任务，0 表示挂起状态，1 表示就绪状态。它记录了各任务的就绪与否状态，称它为就绪表。OSRdyTbl 定义为 32 位变量，对应 32 个任务。当然，定义为 64 位的话，便最多能支持 64 个任务。

这样，可以定义两个宏，实现把任务的状态变为就绪或挂起态。

***RTOS.h***

```
/* 在就绪表中登记就绪任务 */
#define OSSetPrioRdy(prio)                    \
{                                             \
OSRdyTbl |= 0x01<<prio;                       \
}

/* 从就绪表中删除任务 */
#define OSDelPrioRdy(prio)                    \
{                                             \
OSRdyTbl &= ~(0x01<<prio);                    \
}
```

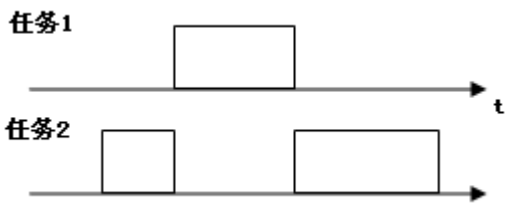
***TIPS:***

为了增加程序可移植性和直观性，代码中常使用重定义的数据类型。在 def.h 中有

```
typedef unsigned char    BOOLEAN;
typedef unsigned char    INT8U;        /* Unsigned    8 bit quantity */
typedef signed    char   INT8S;        /* Signed      8 bit quantity*/
typedef unsigned int     INT16U;       /* Unsigned 16 bit quantity */
typedef signed     int   INT16S;       /* Signed     16 bit quantity*/
typedef unsigned long    INT32U;       /* Unsigned 32 bit quantity */
typedef signed     long  INT32S;       /* Signed     32 bit quantity */
```

- 独立性。任务之间互相独立，不存在互相调用的关系。所有任务在逻辑上都是平等的。由于任务之间互相看不见，所以他们之间的信息传输就无法当面完成。这就需要各种通信机制如信号量，消息邮箱，队列等来实现。

- 并发性。由同一个处理器轮换地运行多个程序。或者说是由多个程序轮班地占用处理器这个资源。且在占用这个资源期间，并不一定能够把程序运行完毕。

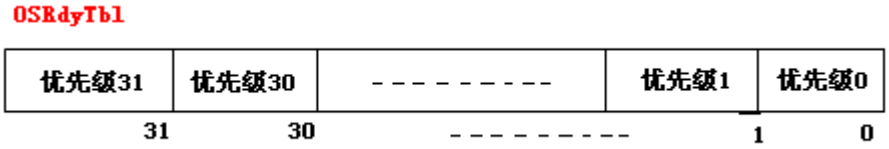


## 什么是抢占式调度？

调度的概念，通俗的说就是系统在多个任务中选择合适的任务执行。系统如何知道何时该执行哪个任务？可以为每个任务安排一个唯一的优先级别，当同时有多个任务就绪时，优先运行优先级较高的任务。

同时，任务的优先级也作为任务的唯一标识号。代码中都是对标识号来完成对任务的操作的。如 **OSDelPrioRdy(prio)，OSSetPrioRdy(prio)**等。

不同的优先级对应就绪表中的每一位。低位对应高优先级。优先级 **0** 的优先权最高，优先级 **31** 的优先权最低。




在程序中要为每一个任务分配一个唯一的优先级。

***Main.c***

```
///***************定义任务优先级**************///
#define PrioTask0          0
#define PrioTask1          1
#define PrioTask2          2
#define PrioTask3          3
```

所谓“抢占式调度”是指：一旦就绪状态中出现优先权更高的任务，便立即剥夺当前任务的运行权，把 **CPU** 分配给更高优先级的任务。这样 **CPU** 总是执行处于就绪条件下优先级最高的任务。

在程序中查找最高优先级的任务代码如下：

***RTOS.h***

```
/*  在就绪表中查找更高级的就绪任务  */
#define OSGetHighRdy()                                                    \
{                                                                         \
   for( OSNextTaskPrio = 0;                                               \
        (OSNextTaskPrio < OS_TASKS) && (!(OSRdyTbl & (0x01<<OSNextTaskPrio))); \
         OSNextTaskPrio ++ );                                             \
   OSPrioHighRdy = OSNextTaskPrio;                                        \
}
```

**TIPS:**

使用这种算法来查找最高优先级的系统严格来说不能称为实时系统。实时系统的特征是延时可预测，能够在一个规定的时间内(通常是 **ms** 级别的)对某些信号做出反应。这种算法的延时随任务的数量的改变而不同，但却是最简便的。

## 多任务系统的时间管理

与人一样，多任务系统也需要一个“心跳”来维持其正常运行，这个心跳叫做时钟节拍，通常由定时器产生一个固定周期的中断来充当，频率一般为 50-100Hz。在 TargetInit.c 文件中有下面的定时器 0 初始化函数，T0 用作系统心跳计时，产生时钟节拍。

*RTOS.h*

```
#define OS_TICKS_PER_SEC    100        /*  设置一秒内的时钟节拍数*/
```

*TargetInit.c*

```
void StartTicker(INT32U TicksPerSec)
{
    rTCFG0 = 99;                          //Prescaler0 = 99
    rTCFG1 = 0x03;                    //Select MUX input for PWM Timer0:divider=16
    rTCNTB0 = 31250 / TicksPerSec;      //设置中断频率
    rTCON |=    (1<<1);                 //Timer 0 manual update
    rTCON = 0x09;                       //Timer 0 auto reload on
                                        //Timer 0 output inverter off
                                        //清"Timer 0 manual update"
                                        //Timer 0 start */
    BIT_CLR(rINTMSK, BIT_TIMER0); // Enable WatchDog interrupts
}
```

OSTimeDly 函数就是以时钟节拍为基准来延时的。这个函数完成功能很简单，就是先挂起当起当前任务，设定其延时节拍数，然后进行任务切换，在指定的时钟节拍数到来之后，将当前任务恢复为就绪状态。任务必须通过 OSTimeDly 或 OSTaskSuspend 让出 CPU 的使用权，使更低优先级任务有机会运行。

*RTOS.c*

```
void OSTimeDly(INT32U ticks)
{
    if( ticks > 0 )                                  /*  当延时有效              */
    {
        OS_ENTER_CRITICAL();
        OSDelPrioRdy(OSPrioCur);                     /*  把任务从就绪表中删去  */
        TCB[OSPrioCur].OSTCBDly = ticks;             /*  设置任务延时节拍数     */
        OS_EXIT_CRITICAL();
        OSSched();                                   /*  重新调度                */
    }
}
```

在 T0 的中断服务函数中，依次对各个延时任务的延时节拍数减 1。若发现某个任务的延时节拍数变为 0，则把它从挂起态置为就绪态。

***RTOS.c***

```
void TickInterrupt(void)
  {
      static INT8U i;

      OSTime ++;
      //Uart_SendByte('I');
      for(i = 0; i < OS_TASKS; i++)              /*  刷新各任务时钟  */
      {
          if(TCB[i].OSTCBDly )
          {
              TCB[i].OSTCBDly --;
              if(TCB[i].OSTCBDly == 0)           /*  当任务时钟到时,必须是由定时
                                                     器减时的才行*/
              {
                  OSSetPrioRdy(i);               /*  使任务可以重新运行      */
              }
          }
      }
      rSRCPND |= BIT_TIMER0;
      rINTPND |= BIT_TIMER0;
}
```

系统自身创建了一个空闲任务，并设它为最低优先级，当系统没有任何任务就绪时，则运行这个任务，让 CPU“有事可干”。用户程序可以在这个任务中加入一些“无关紧要”的功能，如统计 CPU 使用率等。

***RTOS.c***

```
void IdleTask(void)
{
   IdleCount = 0;
   while(1)
   {
       IdleCount++;
       //Uart_Printf("IdleCount %d\n",IdleCount);
   }
}
```

**TIPS：**

不要在空闲任务中运行有可能使任务挂起的函数。空闲任务应该一直处于就绪状态。

## 如何实现多任务？

只有一个 CPU，如何在同一时间实现多个独立程序的运行？要实现多任务，条件是每个任务互相独立。人如何才能独立，有自己的私有财产。任务也一样，如果一个任务有自己的 CPU，堆栈，程序代码，数据存储区，那这个任务就是一个独立的任务。

下面我们来看看是任务是如何“独立”的。

首先是程序代码，每个任务的程序代码与函数一样，与 51 的裸奔程序一样，每个任务都是一个大循环。

```
void task ( )
{
  //initialize
  while(1)
  {
      //your code
  }
}
```

TIPS：

如果一个任务正在运行某个公共函数时（如 Printf），被另一个高优先级的任务抢占，那么当这个高优先级的任务也调用同一个公共函数时，极有可能破坏原任务的数据。因为两个任务可能共用一套数据。为了防止这种情况发生，常采用两种措施:可重入设计和互斥调用。

可重入函数中所有的变量均为局部变量，局部变量在调用时临时分配空间，所以不同的任务在不同的时刻调用该函数时，它们的同一个局部变量所分配的存储空间并不相同（任务私有栈中），互不干扰。另外，如果可重入函数调用了其他函数，则这些被调用的函数也必须是可重入函数。

互斥调用稍后说明。

然后是数据存储区，由于全局变量是系统共用的，各个任务共享，不是任务私有，所以这里的数据存储区是指任务的私有变量，如何变成私有？局部变量也。编译器是把局部变量保存在栈里的，所以好办，只要任务有个私有的栈就行。

TIPS：

临界资源是一次仅允许一个任务使用的共享资源。每个任务中访问临界资源的那段程序称为临界区。

在多任务系统中，为保障数据的可靠性和完整性，共享资源要互斥（独占）访问，所以全局变量（只读的除外）不能同时有多个任务访问，即一个任务访问的时候不能被其他任务打断。共享资源是一种临界资源。

实现互斥（独占）访问的方法有关中断，关调度，互斥信号量，计数信号量等。

定义了如下处理临界资源访问的代码，主要用于在进入临界区之前关闭中断，在退出临界区后恢复原来的中断状态。

***RTOS.h***

```
INT32U    cpu_sr;                  /*  进入临界代码区时保存 CPU 状态*/
/*  进入临界资源代码区  */
#define    OS_ENTER_CRITICAL()    (cpu_sr = OSCPUSaveSR()) /*  关总中断  */
/*  退出临界代码区  */
#define    OS_EXIT_CRITICAL()     (OSCPURestoreSR(cpu_sr)) /*  恢复原来中断状态*/
```

***RTOS_ASM.S***

```
OSCPUSaveSR
            mrs r0,CPSR          ；保存当前中断状态，参阅汇编子程序调用 ATPCS 规则,
                                 ；r0 保存传递的参数
            orr r1,r0,#NOINT     ；关闭所有中断
            msr CPSR_c,r1
            mov pc,lr

OSCPURestoreSR
            msr CPSR_c,r0        ；恢复原来的中断状态
            mov pc,lr
```

为了不影响程序当前的中断状态，先当前的 CPSR 保存起来，退出临界区后再恢复。注意 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()要成对使用。

示例：

```
OS_ENTER_CRITICAL();
Printf(…)；//临界代码
OS_EXIT_CRITICAL();
```

这样，在临界区内，只要任务不主动放弃 CPU 使用权，别的任务就没有占用 CPU 的机会，相当与独占 CPU 了。临界区的代码要尽量短，因为它会使系统响应性能降低。

私有栈的作用是存放局部变量，函数的参数，它是一个线性的空间，所以可以申请一个静态数组，把栈顶指针 SP 指向栈的数组的首元素（递增栈）或最后一个元素（递减栈）。即可打造一个人工的栈出来。

***Main.c***

```
///*******************任务堆栈大小定义************///
#define StackSizeTask0          512
///*******************建立任务堆栈****************///
INT32U StackTask0[StackSizeTask0];
```

**TIPS:**

若在任务中使用 Printf 函数，栈要设得大些，否则栈越界溢出，会有意想不到的问题。

每个任务还要有记录自己栈顶指针的变量，保存在任务控制块（TCB）中。

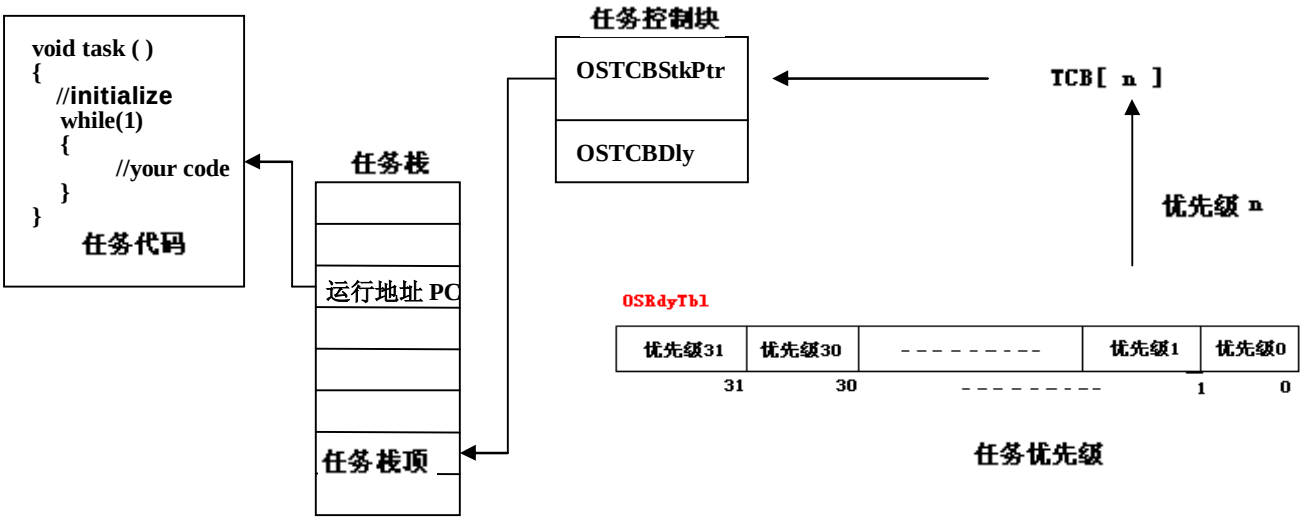
什么是任务控制块？系统中的每个任务具有一个任务控制块，任务控制块记录任务执行的环境，这里的任务控制块比较简单，只包含了任务的堆栈指针和任务延时节拍数。

***RTOS.h***

```
struct TaskCtrBlock                         /*  任务控制块数据结构  */
{
        INT32U    OSTCBStkPtr;              /*  保存任务的堆栈顶     */
        INT32U    OSTCBDly;                 /*  任务延时时钟         */
};

struct TaskCtrBlock     TCB[OS_TASKS + 1]; /*  定义任务控制块         */
```

任务控制块是任务的身份证。它把任务的程序与数据联系起来，找到它就可以得到任务的所有资源。




*程序代码、私有堆栈、任务控制块是任务的三要件*

这里把 **OSTCBStkPtr** 放在结构体的最前面是有原因的，目的是使得在汇编中访问这个变量比较容易。因为结构体的地址就是它的首元素的地址，要在汇编中访问 **OSTCBStkPtr** 这个变量，只需取得结构体的地址即可。

在 **C** 语言中不能直接实现如 **TCB. TCB[OSPrioCur].OSTCBStkPtr = SP** 的功能，只能用汇编语言完成。

定义如下指向结构体的指针变量：

***RTOS.h***

```
struct TaskCtrBlock    *p_OSTCBCur ;       /*  指向当前任务控制块的指针   */
```

在 **C** 语言中，先让 **p_OSTCBCur** 指向当前运行任务的 **TCB**。

```
p_OSTCBCur    = &TCB[OSPrioCur];          /*  目的是在汇编中引用任务的 TCB 地
                                              址取得栈顶指针  */
```

这样，运行下面的汇编代码即可把当前任务的堆顶指针取出到 **SP** 中。

***RTOS_ASM.S***

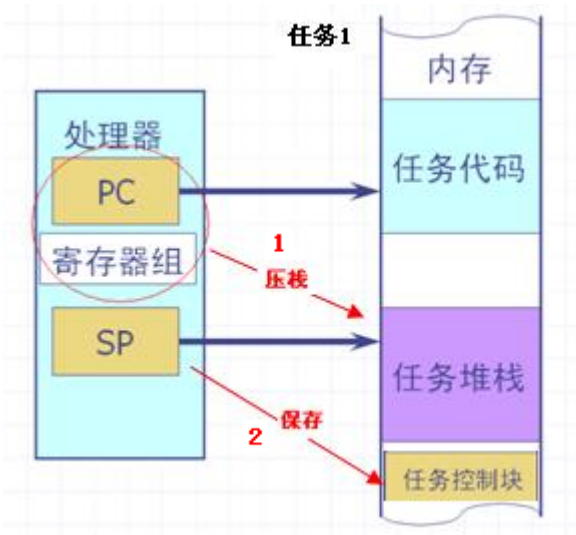
```
SaveSPToCurTcb                          ; 保存当前任务的堆顶指针到它的
        ldr r4,=p_OSTCBCur              ; 取出当前任务的 PCB 地址
        ldr r5,[r4]
        str sp,[r5]                     ; 保存当前任务的堆顶指针到它的
                                        ; TCB(因为 TaskCtrBlock 地址亦即
                                        ; OSTCBStkPtr 的地址)
```

最后来看看任务是如何“拥有”自己的 CPU 的。只有一个 CPU，各个任务共享，轮流使用。如何才能实现？我们先来看看中断的过程，当中断来临时，CPU 把当前程序的运行地址，寄存器等现场数据保存起来（一般保存在栈里），然后跳到中断服务程序执行。待执行完毕，再把先前保存的数据装回 CPU 又回到原来的程序执行。这样就实现了两个不同程序的交叉运行。

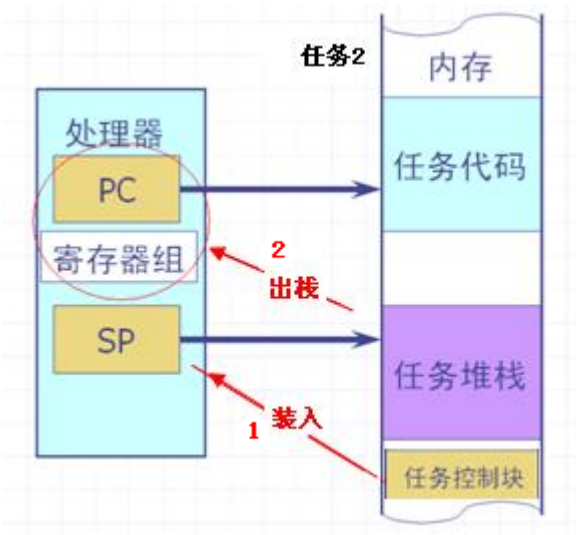
借鉴这种思想不就能实现多任务了吗！模仿中断的过程就可以实现任务切换运行。

任务切换时，把当前任务的现场数据保存在自己的任务栈里面，再把待运行的任务的数据从自己的任务栈装载到 CPU 中，改变 CPU 的 PC，SP，寄存器等。可以说，任务的切换是任务运行环境的切换。而任务的运行环境保存在任务栈中，也就是说，任务切换的关键是把任务的私有堆栈指针赋予处理器的堆栈指针 SP。

两个任务的切换过程如下：




*当前任务的运行环境保存*




*待运行的任务的数据从自己的任务栈装载到 CPU*

程序调用下面函数进行任务切换：

***RTOS.c***

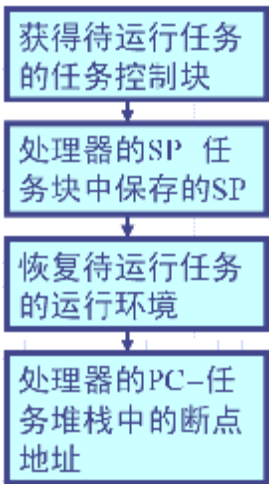
```
void OSSched (void)
{
   OS_ENTER_CRITICAL();
   OSGetHighRdy();                    /*  找出就绪表中优先级最高的任务  */
   if(OSPrioHighRdy != OSPrioCur)     /*  如果不是当前运行的任务，进行任务调度  */
   {
        p_OSTCBCur    = &TCB[OSPrioCur];    /*  目的是在汇编中引用任务的 TCB 地
                                              址取得栈顶指针  */
        p_OSTCBHighRdy = &TCB[OSPrioHighRdy];
        OSPrioCur = OSPrioHighRdy;         /*  更新 OSPrioCur */
        OS_TASK_SW();                      /*  调度任务  */
   }
   OS_EXIT_CRITICAL();
}
```

***RTOS_ASM.S***

```
OS_TASK_SW                          ; 任务级的任务切换
                                    ;
                                                    保存当前任务的运行环境
        stmfd sp!,{lr}              ; PC 入栈
        stmfd sp!,{r0-r12,lr}       ; r0-r12，lr 入栈
PUAH_PSR
        mrs r4,cpsr
        stmfd sp!,{r4}              ; cpsr 入栈
SaveSPToCurTcb                      ; 保存当前任务的堆顶指针到它的 TCB.
                                    ; TCB[OSPrioCur].OSTCBStkPtr = SP;
        ldr r4,=p_OSTCBCur          ; 取出当前任务的 PCB 地址
        ldr r5,[r4]
        str sp,[r5]                 ; 保存当前任务的堆顶指针到它的 TCB(因为
                                    ;TaskCtrBlock 地址亦即 OSTCBStkPtr 的地址)

GetHigTcbSP                         ; 取出更高优先级任务的堆顶指针到 SP，
                                    ;   SP = TCB[OSPrioCur].OSTCBStkPtr
待运行的任务的栈顶指针装载到 CPU 的 SP
        ldr r6,=p_OSTCBHighRdy      ; 取出更高优先级就绪任务的 PCB 的地址
        ldr r6,[r6]
        ldr sp,[r6]                 ; 取出更高优先级任务的堆顶指针到 SP
        b   POP_ALL                 ; 根据设定的栈结构顺序出栈
恢复待运行任务的运行环境
```

任务切换流程图如下：



## 如何建立任务？

**OSTaskCreate 函数在创建任务时调用，如下：**

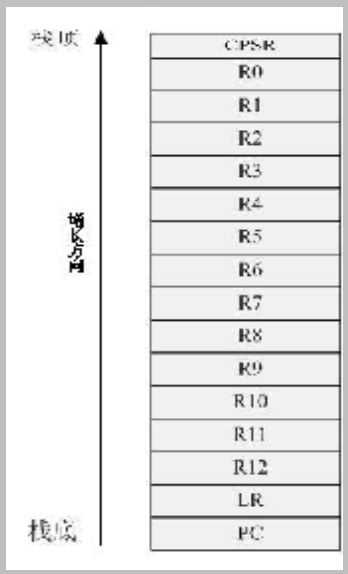
```
OSTaskCreate(Task0,&StackTask0[StackSizeTask0 - 1],PrioTask0); // 创建一个任务
```

任务未运行或被剥夺运行权后，它的运行环境以一定的结构保存在私有栈中。这个规定的顺序是十分重要的，以后任务切换时都要按这个顺序入栈出栈。

建立任务时栈结构的初始化如下，

***RTOS.c***

```
void OSTaskCreate(void (*Task)(void),INT32U *p_Stack,INT8U TaskID)
{
  if(TaskID <= OS_TASKS)
  {
      *(p_Stack)    = (INT32U)Task;                    /* pc */
      *(--p_Stack) = (INT32U)13;                       /* lr */
      *(--p_Stack) = (INT32U)12;                       /* r12*/
      *(--p_Stack) = (INT32U)11;                       /* r11*/
      *(--p_Stack) = (INT32U)10;                       /* r10*/
      *(--p_Stack) = (INT32U)9;                        /* r9 */
      *(--p_Stack) = (INT32U)8;                        /* r8 */
      *(--p_Stack) = (INT32U)7;                        /* r7 */
      *(--p_Stack) = (INT32U)6;                        /* r6 */
      *(--p_Stack) = (INT32U)5;                        /* r5 */
      *(--p_Stack) = (INT32U)4;                        /* r4 */
      *(--p_Stack) = (INT32U)3;                        /* r3 */
      *(--p_Stack) = (INT32U)2;                        /* r2 */
      *(--p_Stack) = (INT32U)1;                        /* r1 */
      *(--p_Stack) = (INT32U)0;                        /* r0 */
      *(--p_Stack) = (INT32U)(SVCMODE|0x0);            /* SREG 保存在任务栈中*/
      TCB[TaskID].OSTCBStkPtr = (INT32U)p_Stack;   /* 将人工堆栈的栈顶，保存到
                                                       堆栈的数组中*/
      TCB[TaskID].OSTCBDly = 0;                /* 初始化任务延时          */
      OSSetPrioRdy(TaskID);                    /* 在任务就绪表中登记      */
  }
  else
  {
      Uart_Printf("TaskID Error\n");
      while(1);
  }
}
```



上面函数的作用就是创建一个任务。它接收三个参数，分别是任务的入口地址，任务堆栈的首地址和任务的优先级。调用本函数后，系统会根据用户给出的参数初始化任务栈，并把栈顶指针保存到任务控制块中，在任务就绪表标记该任务为就绪状态。最后返回，这样一个任务就创建成功了。

可见，初始化后的任务堆栈空间由高到低将依次保存着 PC，LR，R12…R0，CPSR。当一个任务将要运行时，便通过取得它的堆栈指针（保存在任务控制块中）将这些寄存器出栈装入 CPU 相应的位置即可。

初始状态的堆栈要与任务切换后的堆栈保存结构相一致，因为任务被创建后并不是直接就获得执行的，而是通过 OSStartHighRdy（）或 OSSched（）函数进行调度分配，满足执行条件后才能获得执行的。为了使这个调度简单一致，把栈结构统一。

**TIPS:**

为方便处理，所有任务运行在管理模式下。

系统通过运行 OSStartHighRdy 来开始第一个任务。

该函数是在多任务系统启动后，负责从第一个任务的 TCB 控制块中获得该任务的堆栈指针 SP，通过 SP 依次将 CPU 现场恢复。这时系统就将控制权交给该任务，直到该任务被阻塞或者被其它任务抢占 CPU。该函数仅仅在多任务启动时被执行一次，用来启动第一个运行的任务。

***RTOS_ASM.S***

```
OSStartHighRdy
        ldr r4,=p_OSTCBHighRdy    ; 取出最高优先级就绪任务的 PCB 的地址
        ldr r4,[r4]               ; 取得任务的栈顶指针(因为 TaskCtrBlock 地址
                                  ; 亦即 OSTCBStkPtr 的地址)
        ldr sp,[r4]               ; 任务的栈顶指针赋给 SP
        b    POP_ALL              ; 根据设定的栈结构顺序出栈
```

***RTOS_ASM.S***

```
; 根据设定的栈结构顺序出栈
POP_ALL
        ldmfd sp!,{r4}            ; psr 出栈
        msr CPSR_cxsf,r4
        ldmfd sp!,{r0-r12,lr,pc}  ; r0-r12,lr,pc 出栈
```

## 如何实现抢占式调度？

基于任务优先级的抢占式调度，也就是最高优先级的任务一旦处于就绪状态，则立即抢占正在运行的低优先级任务的处理器资源。

为了保证 CPU 总是执行处于就绪条件下优先级最高的任务，每当任务状态改变后，即判断当前运行的任务是否是就绪任务中优先级最高的，否则进行任务切换。

任务状态会在什么时候发生改变呢？有下面两种情况：

- 高优先级的任务因为需要某种资源或延时，主动请求挂起，让出处理器，此时将调度就绪状态的低优先级任务获得执行，这种调度称为任务级的切换。
  如任务执行 OSTimeDly（）或 OSTaskSuspend（）把自身挂起就属于这种。
- 高优先级的任务因为时钟节拍到来，或在中断处理结束后，内核发现更高优先级任务获得了执行条件(如延时的时钟到时)，则在中断后直接切换到更高优先级任务执行。这种调度也称为中断级的切换。

第一种任务级的切换已经分析过了，这里来分析中断级的任务切换。

在目标板初始化函数 TargetInit 中有

***TargetInit.c***

```
pISR_IRQ = (INT32U)ASM_IRQHandler;
```

把 ASM_IRQHandler 的地址赋给中断向量服务程序入口，只要发生中断，系统便跳到 ASM_IRQHandler 函数处执行，它统管了系统的所有中断。

***RTOS_ASM.S***

```
ASM_IRQHandler                  ; 中断入口地址，在中断向量表初始化时被设置
        sub lr,lr,#4            ; 计算中断返回地址
        stmfd sp!,{r0-r3,r12,lr}; 保护现场,此时处于中断模式下，sp 指向中
                                ; 断模式下的堆栈.不能进行任务切换(各任务
                                ; 堆栈处在管理模式堆栈).
                                ; R4-R11 装的是局部变量，在进行函数跳转时，
                                ; 编译器它会自动保护它们，
                                ; 即 C 语言函数返回时，寄存器 R4-R11、SP
                                ; 不会改变，无需人为保护
        bl OSIntEnter
        bl C_IRQHandler         ; 调用 c 语言的中断处理程序
        bl OSIntExit            ; 判断中断后是否有更高优先级的任务进入就
                                ; 绪而需要进行任务切换
        ldr r0,=OSIntCtxSwFlag      ;if(OSIntCtxSwFlag == 1) OSIntCtxSw()
        ldr r1,[r0]
        cmp r1,#1
        beq OSIntCtxSw          ; 有更高优先级的任务进入了就绪状态，则跳
                                ; 转到 OSIntCtxSw 进行中断级的任务切换
        ldmfd sp!,{r0-r3,r12,pc}^ ; 不进行任务切换，出栈返回被中断的任务。
                                ; 寄存器出栈同时将 spsr_irq 的值复制到
                                ; CPSR，实现模式切换
```

**TIPS：**

在 ADS 编译器中，“__irq”专门用来声明 IRQ 中断服务程序，如果用“__irq”来声明一个函数，那么表示该函数为一个 IRQ 中断服务程序，汇编时编译器便会自动在该函数体内加入中断现场保护和恢复的代码。

但在这里，中断由系统管理，中断服务函数不能使用“__irq”关键字进行声明。

在保存返回地址和现场数据后（注意是保存在中断模式下的栈中），随即跳到 OSIntEnter 去执行。OSIntEnter()是为了实现中断嵌套而调用的函数，OSIntNesting 记录的是中断嵌套层数，每进入一次中断，便把 OSIntNesting 的值加 1，在退出中断时，若 OSIntNesting 不为 0，则说明中断仍未完成，要待所有的中断完成后才能返回到任务中。

```
RTOS.c
void    OSIntEnter (void)
{
        if (OSIntNesting < 255)
            {
                OSIntNesting++;
            }
}
```

随即进入 C 语言中断入口程序 C_IRQHandler，它通过判断 INTOFFSET 寄存器来识别当前发生的中断，然后跳转到相应的中断服务函数。

```
RTOS.c
void C_IRQHandler(void)
{

      (* ((void(*)()) (*((U32 *)((int)&pISR_EINT0 + (rINTOFFSET<<2))))) ();

}
```

执行完中断服务程序后，不会马上退出返回先前的任务，它将 **OSIntNesting** 的值减 **1**，当 **OSIntNesting** 的值为 **0** 时，表示所有的嵌套中断都完成了。可以返回原来的任务或进行中断级任务切换。这时判断之前运行的任务是否仍然是最高优先级的就绪任务，如果中断让更高优先级的任务就绪，则需要切换任务，通过置位 **OSIntCtxSwFlag** 变量来标记。

***RTOS.c***

```
void OSIntExit(void)
{
    OS_ENTER_CRITICAL();
    if (OSIntNesting > 0)
    {
        OSIntNesting --;
    }
    if (OSIntNesting == 0)                  /*  退出所有中断才能进行任务切换*/
    {
        OSGetHighRdy();                     /*  找出就绪表中优先级最高的任务  */
        if(OSPrioHighRdy != OSPrioCur)
        {
            p_OSTCBCur = &TCB[OSPrioCur];     /*  目的是在汇编中引用
                                                任务的 TCB 地址取得栈顶指针  */
            p_OSTCBHighRdy = &TCB[OSPrioHighRdy];
            OSPrioCur = OSPrioHighRdy;          /*  更新 OSPrioCur */
            OSIntCtxSwFlag = TRUE;              /*  进行中断级任务调度  */
        }
     }
     OS_EXIT_CRITICAL();
}
```

最后对 **OSIntCtxSwFlag** 标志进行判断，若 **OSIntCtxSwFlag** 置位则执行 **OSIntCtxSw** 函数准备任务切换，否则退出中断返回先前的任务。

OSIntCtxSw 函数是中断后需任务切换时的跳转程序，由它转到中断级任务切换程序。为什么需要它？.

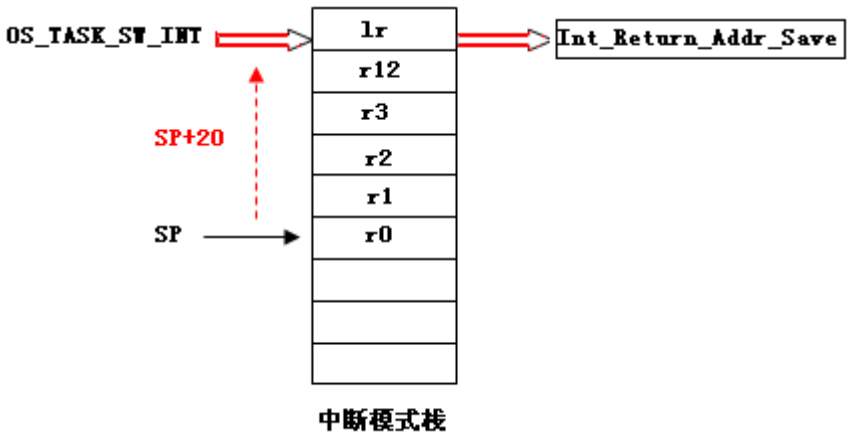
若中断后需任务切换，则中断结束后不返回先前的任务，而去执行中断级任务切换函数 OS_TASK_SW_INT。这是正确的。但此时仍处于中断模式下，SP 指向中断模式下的堆栈.，不能进行任务切换(各任务堆栈处在管理模式堆栈)。所以要待退出中断后才进行任务切换。如何保证退出中断后马上运行 OS_TASK_SW_INT？这得做点手脚。方法是把中断返回地址换成 OS_TASK_SW_INT 函数的地址。因为中断返回地址保存了在中断模式的堆栈中(中断时先保存在中断模式的 lr 中，之后压栈)，这里把它用一个变量 Int_Return_Addr_Save 保存下来，待后面任务切换时再把它取出压入先前任务的堆栈中(管理模式下)。这样，中断返回时，出栈的 PC 值就是 OS_TASK_SW_INT 的地址。

***RTOS_ASM.S***

```
OSIntCtxSw                              ; 把中断的返回地址保存到
                                        ; Int_Return_Addr_Save 变量中
        ldr r0,=OSIntCtxSwFlag          ; 清 OSIntCtxSwFlag 标志位
        mov r1,#0
        str r1,[r0]                     ; OSIntCtxSwFlag = 0

        add sp,sp,#20                   ; 调整 SP,使之指向之前入栈的 lr(中断模
                                        ; 式下的 lr 保存的是中断返回地址),
        ldr r0,[sp]                     ; lr -> r0
        ldr r1,=Int_Return_Addr_Save    ; &Int_Return_Addr_Save -> r1
        str r0,[r1]                     ; lr -> Int_Return_Addr_Save

        ldr r0,=OS_TASK_SW_INT
        str r0,[sp]                 ; OS_TASK_SW_INT -> lr
        sub sp,sp,#20               ; 调整 SP 回到栈顶
        ldmfd sp!,{r0-r3,r12,pc}^   ; 退出中断后跳到 OS_TASK_SW 而不返回
```




*OSIntCtxSw 函数完成的功能*

这样，执行 ldmfd sp!,{r0-r3,r12,pc}^后，PC 便指向 OS_TASK_SW_INT 函数，执行中断级任务切换。此时的栈为进入中断之前运行的任务的栈，要把这个任务的运行环境保存到栈中。之后的代码便与任务级的切换的一样。

***RTOS_ASM.S***

*把保存在 Int_Return_Addr_Save 中的地址取出来和其它寄存器一起入栈*

```
OS_TASK_SW_INT                  ; 中断级任务切换,中断服务时使用另外
                                ; 的堆栈，所以要回到管理模式中才进行任务切换
        sub sp,sp,#4            ; 为 PC 保留位置
        stmfd sp!,{r0-r12,lr}   ; r0-r12，lr 入栈
        ldr r0,=Int_Return_Addr_Save; 取出进入中断之前保存的 PC
        ldr r0,[r0]             ; Int_Return_Addr_Save -> r0
        add sp,sp,#56           ; 调整 SP
        stmfd sp,{r0}           ; r0(PC)入栈
        sub sp,sp,#56           ; 调整 SP 回到栈顶
        b   PUAH_PSR
PUAH_PSR
        mrs r4,cpsr
        stmfd sp!,{r4}              ; cpsr 入栈
```

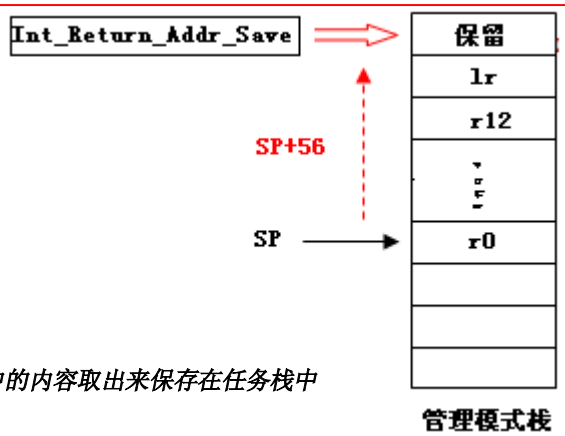
*保存当前任务的堆顶指针到它的 TCB*

```
SaveSPToCurTcb                  ; TCB[OSPrioCur].OSTCBStkPtr = SP;
        ldr r4,=p_OSTCBCur      ; 取出当前任务的 PCB 地址
        ldr r5,[r4]
        str sp,[r5]             ; 保存当前任务的堆顶指针到它的 TCB(因为
                                ;TaskCtrBlock 地址亦即 OSTCBStkPtr 的地址)
```

*待运行的任务的栈顶指针装载到 CPU 的 SP*

```
GetHigTcbSP                     ; 取出更高优先级任务的堆顶指针到 SP，
                                ;   SP = TCB[OSPrioCur].OSTCBStkPtr
        ldr r6,=p_OSTCBHighRdy  ; 取出更高优先级就绪任务的 PCB 的地址
        ldr r6,[r6]
        ldr sp,[r6]             ; 取出更高优先级任务的堆顶指针到 SP
        b   POP_ALL             ; 根据设定的栈结构顺序出栈
```

*恢复待运行任务的运行环境*




*把保存在 Int_Return_Addr_Save 中的内容取出来保存在任务栈中*

## 挂起/恢复任务

- 挂起任务

通过 OSTaskSuspend()可以主动挂起一个任务。OSTaskSuspend()会把任务从任务就绪表中移出，最后重新启动系统调度。这个函数可以挂起任务本身也可以挂起其他任务。

```
RTOS.c
void OSTaskSuspend(INT8U prio)
{
   OS_ENTER_CRITICAL();
   TCB[prio].OSTCBDly = 0;
   OSDelPrioRdy(prio);                  /* 从任务就绪表上去除标志位 */
   OS_EXIT_CRITICAL();

   if(OSPrioCur == prio)                /* 当要挂起的任务为当前任务  */
   {
        OSSched();                      /* 重新调度                  */
   }
}
```

- 恢复任务（OSTaskResume()）

可以让被 OSTaskSuspend 或 OSTimeDly 挂起的任务恢复就绪态，然后进行任务调度。

```
RTOS.c
void OSTaskResume(INT8U prio)
{
   OS_ENTER_CRITICAL();
   OSSetPrioRdy(prio);              /* 从任务就绪表上重置标志位 */
   TCB[prio].OSTCBDly = 0;          /* 将时间计时设为 0,延时到  */
   OS_EXIT_CRITICAL();

   if(OSPrioCur > prio)         /* 当前任务的优先级低于重置位的任务的优先级 */
   {
        OSSched();              /* 重新调度 */
   }
}
```

## 一些技巧

### l 总头文件“includes.h”

***Includes.h***

```
#ifndef __INCLUDES_H__
#define __INCLUDES_H__

#include "def.h"
#include "option.h"
#include "2440addr.h"
#include "2440lib.h"
#include "2440slib.h"
#include "RTOS.h"
#include "TargetInit.h"
#endif
```

***RTOS.c***

```
#include   "Includes.h"
……..
```

***Main.c***

```
#include   "Includes.h"
……..
```

“Includes.h”是主函数文件中唯一包含的头文件。每个.C 文件都通过“includes.h”包含了所有的.H 文件。主函数文件中用到的头文件(通用的和面向硬件对象的头文件)、宏定义、函数声明都放在这个总头文件中。这样就无须考虑每个.C 文件应该包含哪些相应的头文件，它们通过“Includes.h”全部包含进来了。

注意用条件编译（#ifndef -- #endif）避免总头文件中的头文件重复包含。

- 全局变量的定义：

***RTOS.h***

```
#ifdef     CPP_GLOBALS
#define    EXTERN
#else
#define    EXTERN    extern
#endif

EXTERN    INT32U    StackIdle[StackSizeIdle];        /*  建立空闲任务栈              */
EXTERN    INT32U    OSRdyTbl;                        /*  就绪任务表                  */
……..
```

***RTOS.c***

```
#define     CPP_GLOBALS
#include    "Includes.h"
……..
```

一般程序中的全局变量在.C 文件定义，并在它的相应头文件中通过 extern 关键字声明变量。因此，全局变量必须在 .C 和 .H 文件中定义。这种重复的定义很容易导致错误。而上面的方法只需用在头文件中定义一次。

因为.C 文件会包含.H 文件，所以可以统一在.H 文件中定义，在定义变量时加上 EXTERN 关键字，EXTERN 的解释随宏定义 CPP_GLOBALS 的不同而不同（extern 或“空”）。若 CPP_GLOBALS 未被宏定义，则表明当前的文件是.H 文件，EXTERN 被编译器解释为 extern 关键字，这是表明它是外部变量，只作为变量的外部引用不分配空间；若 CPP_GLOBALS 被宏定义，则表明当前的文件是.C 文件，EXTERN 被编译器解释为空，为之变量分配空间。这样，只需在相应的.C 文件中加上#define CPP_GLOBALS 即可。通过这种办法，使得全局变量只需在头文件中定义一次就可以了，避免了不少麻烦。

## 写在最后

如果你都把上面的内容都消化了，那么恭喜你，你可以很轻松地去看 UCOS 的源代码了（当然你要有一定的数据结构基础）,原理基本上是一样的。

关于多任务系统的应用程序的编写，建议查阅周航慈老师的《基于嵌入式实时操作系统的程序设计技术》，很好的一本书。

摘录网上的一篇文章的一段作为结尾：

“ 工程师在选用多任务操作系统前要先看看自己的项目是不是真需要用操作系统！如果你的任务可折分性较差，折分后的各个任务之间有 N 多的同步问题和复用资源问题，那么算了，我觉得你还是不要用多任务操作系统，或者将这些功能都放在一个任务里面，不要有事没事就觉得多任务好！多任务是用降低实时性来换取软件开发的独立性，不要被实时多任务操作系统的实时两个字骗了，这个实时只是相对于其它非实时性多任务操作系统来讲的，实时性最高的当然是你自己编写的单任务程序。”

--------实时多任务操作系统之我见

http://www.avrw.com/article/art_101_4031.htm

参考资料：

建立一个属于自己的 AVR 的 RTOS

--------- http://www.21ic.com/news/html/80/show11677.htm

嵌入式实时操作系统 uc/osII ----- Jean J.Labrosse

嵌入式实时操作系统μC\OS-Ⅱ原理及应用 ----- 任哲

基于嵌入式实时操作系统的程序设计技术 ----- 周航慈

By Lisuwei

Lisuweizhai@126.com

2008.12