嵌入式实时操作系统 uCosII 在奋斗 STM32 开发板上的应用

嵌入式实时操作系统 uCosII 是由美国工程师 Jean J.Labrosse 所创，它在中国的流行源于那本被邵贝贝引进翻译的著名书籍《嵌入式实时操作系统 uCos-II》，这本书是学习 ucosII 的宝典，虽然很厚，但理解了关键概念，再结合实际应用例程，还是很容易看懂的。uCosII 通过了美国航天管理局（FAA）的安全认证，可以用于飞机、航天器与人性命攸关的控制系统中。也就是说，用户可以放心将 uCosII 用到自己的产品中。

特点：

可移植性：uCosII 源码绝大部分是用移植性很强的 ANSI C 写的。与微处理硬件相关的部分是用汇编语言写的。 uCOS 可以在绝大多数 8 位、16 位、32 位以及 64 位处理器、微控制器及数字信号处理器（DSP）上运行。

可裁剪性：可以通过开关条件编译选项，来定义哪些 uCosII 的功能模块用于用户程序，方便控制代码运行所占用的空间及内存。

可剥夺性：uCOSII 是完全可剥夺型的实时内核，它总是运行处于就绪状态下的优先级最高的任务。

多任务：uCOSII 可以管理 64 个任务，每个任务对应一个优先级，并且是各不相同。其中 8 个任务保留给 uCOSII。用户的应用程序可以实际使用 56 个任务。

可确定性：绝大多数 uCosII 的函数调用和服务的执行时间具有可确定性，也就是说用户总是能知道函数调用与服务执行了多长时间。

任务栈： 每个任务都有自己单独的栈，uCOSII 规定每个任务有不同的栈空间。

系统服务：uCOSII 提供很多系统服务， 例如信号量、互斥信号量、事件标志、消息邮箱、消息队列、内存的申请与释放及时间管理函数等。

中断管理： 中断可以使正在执行的任务暂时挂起，中断嵌套层数可达 255 层。

应用

奋斗 STM32 开发板 MINI 及 V3 采用了 STM32F103VET6 作为板上的 MCU，内置 512K FLASH 64K SRAM。 非常适合短小精悍的 uCosII 作为操作系统。而且 ucosII 是实时操作系统，也极适合 STM32 所面对的嵌入式微控领域。奋斗板选用了已经被移植到 STM32 平台上的 ucosII2.86 源码。经过广泛测试，这个移植好的源码在 STM32 上是运行可靠的， 我们可以更加专心关注应用软件的开发。下面以奋斗板板例程《STM32 奋斗板-LED 闪烁-ucos》为实例来讲解一下 ucosII 在 STM32 下的应用。

功能要求：开发板上电后，LED1-3 会按照默认的500ms 间隔，明暗闪烁，此时可以通过串口助手SSCOM3.2 发出指令，设置LED1,LED2,LED3 的闪烁间隔时间。间隔范围是1-65535ms。可以设置任意一个LED 的闪烁间隔时间。

根据功能要求，对这个例程进行了工程策划， 选用MDK3.80a作为工程编译环境。 JLINK V8作为下载

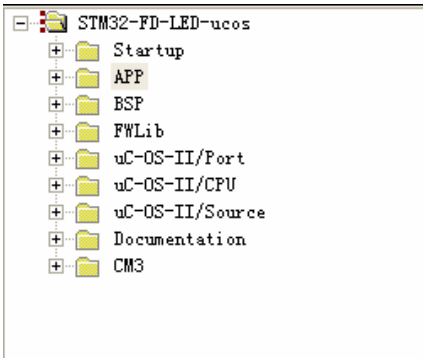
仿真器，三个LED的闪烁分别采用建立3个任务。 功能里的串口接收指令，表明例程会用到串口1中断，还需建立一个串口接收任务。再采用一个任务作为初始化时的主任务，用于建立以上的4个用户任务。 根据实时响应的重要程度，将各个任务的优先级经行了设置。

任务名		优先级
APP_TASK_START_PRIO	2	主任务
Task_Com1_PRIO	4	COM1通信任务
Task_Led1_PRIO	7	LED1 闪烁任务
Task_Led2_PRIO	8	LED2 闪烁任务
Task_Led3_PRIO	9	LED3 闪烁任务

为了兼顾实时效率及 CPU 的负荷。将 ucosII 的时钟节拍设置为 10ms， ucosII 需要提供周期性信号源，用于实现时间延时和确认超时，时钟节拍的含义就是任务和任务之间最短切换时间。这个节拍也不能设置的非常短，会造成 CPU 负荷过大，会造成任务执行兼顾不周。 某些高优先级任务总是在执行，有些低优先级任务得不到执行。 但节拍也不能设置的非常长，这会造成任务执行的实时性变差。 一般 10-100ms 就可以了。

下面分析一下这个程序的结构。

打开工程，可以在工程结构栏看到这个例程的工程结构（如下图）



Startup 组项：

包含了适用于 STM32F103 高容量系列的启动文件。 这是程序的执行的入口文件。在上电启动时，主要完成了对堆栈的初始设置，设置中断向量表，以及跳转到最终指向 main（）函数的 C 库。

APP 组项：

App.c 里包含了任务的建立、各任务的原型以及 ucosII 内核的启动。

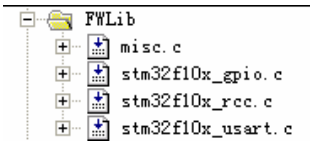
Stm32f10x_it.c 里包含了各个中断服务程序。在这个例程中，只用到了两个中断，一个是 systick 中断，一个是串口 1 中断。 Systick 中断为 ucosII 内核提供了 10ms 的时钟节拍。

BSP 组项：

Com.c 包含了串口 1 的初始化。

Bsp.c 包含了对所用到外设的初始化。

FWLIB 组项：



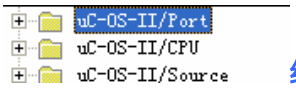
这个组项里包含了例程所用的到的 STM32 的各外设固件库。

Misc.c 是和中断设置有关系的固件库

Stm32f10x_gpio.c　是和通用端口有关系的库

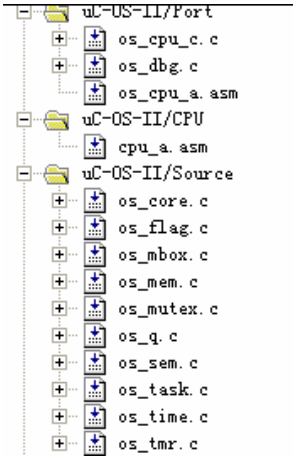
Stm32f10x_rcc.c　是和外设时钟有关系的库

Stm32f10x_usart.c 是和串口有关系的库

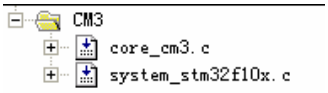


组项：

以下的文件都是 ucosII2.86 源码。 这些文件已经移植到 STM32 平台下，可以直接包含到工程里使用。



CM3 组项：



Core_cm3.c 包含了 Coretex-M3 内核的外设访问层源文件。

System_stm32f10x.c 包含了和 Coretex-M3 系统时钟有关系的外设访问层源文件。

例程所用到的 uCosII 概念解释：

任务：

任务通常是一个无限的循环， 返回参数必须定义为 void。 当任务开始执行时，会有一个参数传递给用户任务代码。 uCosII 可以管理 64 个任务， 其中系统保留了 8 个任务。开放给用户的有 56 个任务，每个任务的优先级都不同， 任务的优先级号越低，任务的优先级越高，在这个版本的 uCosII 中，任务的优先级号就是任务编号。

任务的状态一定是以下 5 种之一：

睡眠态：就是任务没有交给 ucosII 调度的，也就是没用经过建立的任务。 只保存在存储器空间里。

就绪态：任务一旦建立，任务就进入就绪态。任务可以通过调用 OSTasjDel()返回到睡眠态。

运行态：任何时刻只能有一个任务处于运行态。

等待状态：正在运行的任务可以通过调用以下 2 个函数之一，将自身延迟一段时间。 这 2 个函数是 OSTimeDly（）或 OSTimeDlyHMSM（）。这个任务于是进入等待状态，一直到函数中定义的延迟时间到。

正在运行的任务可能在等待某一事件的发生，可以通过调用以下函数之一实现： OSFlagPend() ，OSSemPend(),OSMutexPend(), OSMboxPend()或 OSQPend()。如果某事件并未发生，调用上述函数的任务就进入了等待状态，直到等待的事件发生了。当任务因等待事件被挂起时，下一个优先级最高的就绪任务就得到了 CPU 的使用权。当时间发生了或等待延时超时时，被挂起的任务就进入就绪态。

中断服务态：正在运行的任务是可以被中断的，被中断的任务于是进入了中断服务态，响应中断时，正在执行的任务被挂起，中断服务程序控制了 CPU 的使用权。从中断服务程序返回前，uCosII 要判定被中断的任务是否是目前就绪任务里优先级最高的，如果不是，就执行优先级最高的那个任务。 如果是，就执行被中断的这个任务。

消息邮箱：这是 uCosII 中的一种通信机制，可以使一个任务或者中断服务程序向另一个任务发送一个指针型的变量，通常该指针指向了一个包含了消息的特定数据结构。 在例程中需要建立邮箱，用到了函数 OSMboxCreate(), 串口 1 中断服务程序用到了向邮箱发送一则消息的函数 OSMboxPost(), 串口接收任务用到了等待邮箱中消息的函数 OSMboxPend()。

程序流程讲解：

经过启动文件的初始化，首先在 C 代码里，执行 main（）， 在启动 ucosII 内核前先禁止 CPU 的中断-CPU_IntDis()，防止启动过程中系统崩溃，然后对 ucosII 内核进行初始化 OSInit()，以上两个函数都不用特意修改，在 STM32 平台上已经被移植好了，对板子上的一些用到的外设进行初始化设置 BSP_Init()， 这个函数包含了对系统时钟的设置 RCC_Configuration（），将系统时钟设置为 72MHz，对 LED 闪烁控制端口进行设置 GPIO_Configuration()，对中断源进行配置 NVIC_Configuration()， 对串口 1 进行配置 USART_Config(USART1,115200)，板子上的外设初始化完毕后，默认 3 个 LED 的闪烁间隔分别是 500ms、500ms、500ms, 通过串口 1 格式化输出例程信息，

```
USART_OUT(USART1,"****(C) COPYRIGHT 2010 奋斗嵌入式开发工作室 *******\r\n");
USART_OUT(USART1,"*                                                  *\r\n");
USART_OUT(USART1,"*            奋斗版 STM32 开发板 LED 闪烁实验          *\r\n");
USART_OUT(USART1,"*                                                  *\r\n");
USART_OUT(USART1,"*                 基于 uCOSII2.86                  *\r\n");
USART_OUT(USART1,"*                                                  *\r\n");
USART_OUT(USART1,"*LED1 闪烁间隔：1ms--65535ms   指令 L1 1F--L1 65535F *\r\n");
USART_OUT(USART1,"*LED2 闪烁间隔：1ms--65535ms   指令 L2 1F--L1 65535F *\r\n");
USART_OUT(USART1,"*LED3 闪烁间隔：1ms--65535ms   指令 L3 1F--L1 65535F *\r\n");
USART_OUT(USART1,"*                                                  *\r\n");
USART_OUT(USART1,"*   奋斗 STM32 论坛：ourstm.5d6d.com                 *\r\n");
USART_OUT(USART1,"*                                                  *\r\n");
USART_OUT(USART1,"****************************************************\r\n");
USART_OUT(USART1,"\r\n");
USART_OUT(USART1,"\r\n");
```

建立主任务，该任务是为了在内核启动后，建立另外 4 个用户任务。

```
os_err = OSTaskCreate((void (*) (void *)) App_TaskStart,     //指向任务代码的指针
                      (void *) 0,                            //任务开始执行时，传递给任务的参数的指针
                      (OS_STK *) &App_TaskStartStk[APP_TASK_START_STK_SIZE - 1], //分配给任务的堆栈的栈顶指针    从顶向下递减
                      (INT8U) APP_TASK_START_PRIO);          //分配给任务的优先级
```

主任务的任务名为 App_TaskStart, 主任务有自己的堆栈， 堆栈尺寸为 APP_TASK_START_STK_SIZE*4

（字节）， 然后执行 ucosII 内部函数 OSTimeSet(0)，将节拍计数器清 0，节拍计数器范围是 0-4294967295，对于节拍频率 100hz 时， 每隔 497 天就重新计数， 调用内部函数 OSStart()，启动 ucosII 内核， 此时 ucosII 内核开始运行。对任务表进行监视，主任务因为已经处于就绪状态，于是开始执行主任务 App_TaskStart()，uCOSII 的任务结构规定必须为无返回的结构，也就是无限循环模式。

```
static   void App_TaskStart(void* p_arg)
{
    (void) p_arg;                          //这个语句是防止没有引用的参数会编译错误或警告
    OS_CPU_SysTickInit();                  //时钟节拍初始化为 100Hz （10ms）
#if (OS_TASK_STAT_EN > 0)                  //使能 ucos 的统计任务
    //----统计任务初始化函数
    OSStatInit();
#endif
App_TaskCreate();                          //建立其他 4 个用户任务
while (1)
{
    //1 秒一次循环                         //也可采用挂起
     OSTimeDlyHMSM(0, 0,1, 0);
    }
}


static   void App_TaskCreate(void)
{
    Com1_MBOX=OSMboxCreate((void *) 0)          //建立串口 1 中断的消息邮箱
    //串口 1 接收及发送任务-------------------------------------------------------
    OSTaskCreateExt(Task_Com1,
                    void *)0,                                          //任务开始执行时，传递给任务的参数的指针
                    (OS_STK *)&Task_Com1Stk[Task_Com1_STK_SIZE-1],     //分配给任务的堆栈的栈顶指针   从顶向下递减
                     Task_Com1_PRIO,                                   //分配给任务的优先级
                     Task_Com1_PRIO,                                   //预备给以后版本的特殊标识符，在现行版本同任务优先级
                     (OS_STK *)&Task_Com1Stk[0],                       //指向任务堆栈栈底的指针，用于堆栈的检验
                     Task_Com1_STK_SIZE,                               //指定堆栈的容量，用于堆栈的检验
                     (void *)0,                                        //指向用户附加的数据域的指针，用来扩展任务的任务控制块
                     OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR);   //选项，指定是否允许堆栈检验，是否将堆栈清 0,任务是否要进行浮点运算等等。
    //LED1 闪烁任务---------------------------------------------------
    OSTaskCreateExt(Task_Led1,
                    (void *)0,
                    (OS_STK *)&Task_Led1Stk[Task_Led1_STK_SIZE-1],
                    Task_Led1_PRIO,
                    Task_Led1_PRIO,
                    (OS_STK *)&Task_Led1Stk[0],
                     Task_Led1_STK_SIZE,
                     (void *)0,
                    OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR);
```

```
    //LED2 闪烁任务----------------------------------------------------
    OSTaskCreateExt(Task_Led2,
                    (void *)0,
                    (OS_STK *)&Task_Led2Stk[Task_Led2_STK_SIZE-1],
                    Task_Led2_PRIO,
                    Task_Led2_PRIO,
                    (OS_STK *)&Task_Led2Stk[0],
                    Task_Led2_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR);

    //LED3 闪烁任务----------------------------------------------------
    OSTaskCreateExt(Task_Led3,(void *)0,
                    (OS_STK *)&Task_Led3Stk[Task_Led3_STK_SIZE-1],
                    Task_Led3_PRIO,
                    Task_Led3_PRIO,
                    (OS_STK *)&Task_Led3Stk[0],
                    Task_Led3_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR);
}
```

4 个用户任务各自有自己的堆栈空间，为了接收串口的信息事件，建立了一个信息邮箱，来传递串口进来的信息。

```
//LED1 闪烁任务----------------------------------
static   void Task_Led1(void* p_arg)
{
    (void) p_arg;
    while (1)
    {
        LED_LED1_ON();
        OSTimeDlyHMSM(0, 0, 0, milsec1);
        LED_LED1_OFF();
        OSTimeDlyHMSM(0, 0, 0, milsec1);
    }
}
```

闪烁任务是通过精确延时来是任务就绪及挂起的。 函数原型可以参考 ucosII 手册， 最大精度是 10ms，延时 105ms， 其实是延时了 110ms。

```
//++++++++++++COM1 处理任务++++++++++++++++++++++++++
static   void Task_Com1(void *p_arg){
    INT8U err;
    unsigned char * msg;
    (void)p_arg;
    while(1){
```

```
    msg=(unsigned char *)OSMboxPend(Com1_MBOX,0,&err);          //等待串口接收指令成功的邮箱信息
    if(msg[0]=='L'&&msg[1]==0x31){
          milsec1=atoi(&msg[3]);                        //LED1 的延时毫秒   (mini and V3)
        USART_OUT(USART1,"\r\n");
        USART_OUT(USART1,"LED1: %d ms 间隔闪烁",milsec1);
    }
    else if(msg[0]=='L'&&msg[1]==0x32){
       milsec2=atoi(&msg[3]);                             //LED2 的延时毫秒   (only V3)
       USART_OUT(USART1,"\r\n");
       USART_OUT(USART1,"LED2: %d ms 间隔闪烁",milsec2);
    }
    else if(msg[0]=='L'&&msg[1]==0x33){
       milsec3=atoi(&msg[3]);                             //LED3 的延时毫秒   (only V3)
       USART_OUT(USART1,"\r\n");
       USART_OUT(USART1,"LED3: %d ms 间隔闪烁",milsec3);
  }
}
```

msg=(unsigned char *)OSMboxPend(Com1_MBOX,0,&err); 这条语句是等待消息邮箱有信息传递过来， 一旦接收到到改变 LED 闪烁延时的指令，串口中断例程就通过 OSMboxPost(Com1_MBOX,(void *)&msg); 将接收到的信息用消息邮箱传递给串口接收指令任务， 串口接收任务接收到指令后，对指令进行解析， 以获得各 LED 的闪烁间隔时间。 Milsec1、Milsec2、Milsec3 作为全局变量传递给各个 LED 闪烁任务。 LED 闪烁任务在延时等待结束后，会用新的延时值重新运行。

以下是串口 1 接收中断例程

```
void USART1_IRQHandler(void)
{      unsigned int i;
      unsigned char msg[50];
      OS_CPU_SR    cpu_sr;
      OS_ENTER_CRITICAL();   //保存全局中断标志,关总中断// Tell uC/OS-II that we are starting an ISR
      OSIntNesting++;               //这个是中断嵌套标志，
      OS_EXIT_CRITICAL();      //恢复全局中断标志

        if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)                  //判断读寄存器是否非空
        {
             // Read one byte from the receive data register
             msg[RxCounter1++]= USART_ReceiveData(USART1);     //将读寄存器的数据缓存到接收缓冲区里
            if(msg[RxCounter1-1]=='L'){msg[0]='L'; RxCounter1=1;}        //判断起始标志
            if(msg[RxCounter1-1]=='F')                                   //判断结束标志是否是"F"
            {
                for(i=0; i< RxCounter1; i++){
                TxBuffer1[i]=msg[i];                                    //将接收缓冲器的数据转到发送缓冲区，准备转发
            }
            TxBuffer1[RxCounter1]=0;                                     //接收缓冲区终止符
            RxCounter1=0;
            OSMboxPost(Com1_MBOX,(void *)&msg);                          //发送消息邮箱
```

```
        }
        if(USART_GetITStatus(USART1, USART_IT_TXE) != RESET)                    //
        {
            USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
        }
        OSIntExit();    //在 os_core.c 文件里定义,如果有更高优先级的任务就绪了,则执行一次任务切换
}
```

以上的例程流程完成了一个支持 3 个 LED 灯根据串口的指令，来变换闪烁延时。

2011 年 5 月 16 日　西安

奋斗嵌入式开发工作室