FSC_STOS 说明及例程

——望穿秋水 2019.4

目录

(一)整体说明

1.FSC_STOS 介绍：

<1>三种运行模式：时间片轮询、纯优先级、时间片+优先级混合。

<2>任务时间切片独立设置，不同任务可设置不同时间片。

<3>界面简洁、直观。

<4>文件总共 4 个。

<5>提供虚拟定时器、标志量、互斥量、邮箱。

<6>系统串口指令简结、可自定义。

<7>为新手设计，一看就会用的 OS。

2.支持 ARM Cortex-M0、M3、M4。支持芯片: STM32L0xx、STM32F0xx、STM32L1xx、STM32F1xx、STM32F4xx 等。

3.文档适用版本：FSC_STOS V4.8 以上。

4.关于 OS 运行模式，建议使用顺序运行模式。

<1>顺序运行模式：所有任务按顺序运行，能保证每个任务都能被执行到，为了加强实时性，可使用 OS_delayMs()控制任务的运行时间，使用 OSTaskSwitch()跳转至目标任务以增强实时性。

<2>优先级运行模式：系统只会运行处于就绪状态的最高优先级任务，给高优先级任务安排合理的系统延时才能保证其他低优先级任务能得到运行。一般来说，延时越长的任务优先级安排越高。系统任务优先级最大为任务最大数量-1。

5.系统时钟节拍固定为 1ms，系统时间切片以系统时钟节拍为基数，在此基数上任意可调。

6.任务时间切片根据任务代码量设置，一般主任务 100ms 为佳(对实时性有一定影响)，其他副任务视代码决定。

(二)说明及示例代码

(1)系统延时函数说明

<1>void delay_us(INT32U nus);
功能：微秒延时。(建议驱动中全用此延时函数)
延时范围：0 - 2^32 us。
使用范围：OS 模式、半裸机模式、准裸机模式，非中断函数及中断函数。

<2>void delay_ms(INT32U nms);
功能：毫秒延时函数。
延时范围：0 - 2^32 ms。
使用范围：OS 模式、半裸机模式，准裸机模式，非中断函数及中断函数。

<3>void OS_delayMs(volatile INT32U nms);
功能：任务毫秒延时函数。
延时范围：0 - 2^32 ms。
使用范围：OS 模式，非中断函数。

<4>void OS_delayDHMS(INT8U day,INT8U hour,INT8U min,INT8U sec);
功能：delayMs()函数按日时分秒延时。
延时范围：<=49 天 17 时 2 分 47 秒。
使用范围：OS 模式，非中断函数。

(2)任务功能函数示例代码

<1>void OSSchedLock(void)和 void OSSchedUnlock(void);
功能：锁定任务切换、解锁任务切换。
说明：任务锁定后系统将只在本任务运行，不会运行其他任务，进入半裸机模式。
任务解锁后恢复正常运行，系统将能正常切换到其他任务运行。有的代码必须一次性完成，中途不能被切换出去。代码处理完后必须调用 OSSchedUnlock()解除锁定。
例：

```
void Task2(void) //任务 2 (用作实时时间数据刷新)
{
    OSSchedLock();//锁定任务切换
    DS3231_Config();        //初使化 DS3231（主要用于检测 DS3231 是否正常）
    DS3231_SetTime(2019,3,18,1,9,32,0); //设置初使时间(如 IC 时间是准的，可不用设置)
    OSSchedUnlock();//解锁任务切换  (在初使化时，为了保证初使化不被干扰，锁定切换)
    while(1)
     {
          OSSchedLock();//锁定任务切换  (准备读取数据，锁定切换)
          DS3231_Update(&CurrentTime); //更新时间数据(从 DS3231 读取数据并保存到结
                                       //构体 CurrentTime 中)
```

```
            OSSchedUnlock();//解锁任务切换
            OS_delayMs(100);//100ms 刷新一次数据
        }
}
```

<2>void OSEnterDriver(void)和 void OSExitDriver(void);

功能：进入准裸机模式，排除 OS 对驱动的干扰，保证驱动的正常读写。

使用范围：建议只在延时较短的驱动中使用，驱动中有较大延时时，可将驱动代码以较长延时处分成若干部分，分别对若干延时较小的部分驱动代码使用准裸机模式读写操作。较长延时部分可使用 OS_delayMs()来提高代码执行效率。

例：

```
/*********************************************************
*从 ds18b20 得到温度值
*精度：0.1C
*返回值：温度值 （-550~1250）
*********************************************************/
float DS18B20_Readdata(GPIO_TypeDef* GPIOx,uint16_t GPIO_Pin_x)
{
    u8 TL,TH,i=0,j=0,Dnum=0;
    short Temperature;
    float Temperature1;
    OSEnterDriver(); //进入驱动关键程序，进入准裸机模式
    while(i<DS18B20_NUM)
    {
        if(GPIOx==DS18B20_Sort.GPIO[i]){
        j=0;
        while(j<DS18B20_NUM){
        if(GPIO_Pin_x==DS18B20_Sort.Pin[j]){
        if(i==j) {
        Dnum=i;
        break; }}
        j++;}
        i++;}
    }
    DS18B20_Rst(GPIOx,GPIO_Pin_x);
    DS18B20_Answer_Check(GPIOx,GPIO_Pin_x);
    DS18B20_Write_Byte(GPIOx,GPIO_Pin_x,0xcc);
    DS18B20_Write_Byte(GPIOx,GPIO_Pin_x,0x44);
    DS18B20_Rst(GPIOx,GPIO_Pin_x);
    DS18B20_Answer_Check(GPIOx,GPIO_Pin_x);
    DS18B20_Write_Byte(GPIOx,GPIO_Pin_x,0xcc);// skip rom
    DS18B20_Write_Byte(GPIOx,GPIO_Pin_x,0xbe);// convert
    TL=DS18B20_Read_Byte(GPIOx,GPIO_Pin_x); // LSB
    TH=DS18B20_Read_Byte(GPIOx,GPIO_Pin_x); // MSB
    if( TH&0xfc)
```

```
    {
        Temperature=(TH<<8)|TL;
        Temperature1=(~ Temperature)+1;
        Temperature1*=0.0625;
    }
    else
    {
        Temperature1=((TH<<8)|TL)*0.0625;
    }
    Temperature1+=Deviatvalue;//校正
    if((Temperature1>-50)&&(Temperature1<150)) Ds18b20.Val[Dnum]=Temperature1;
    if(Ds18b20.Val[Dnum]>Ds18b20.ValMax[Dnum])
    Ds18b20.ValMax[Dnum]=Ds18b20.Val[Dnum];
    if(Ds18b20.Val[Dnum]<Ds18b20.ValMin[Dnum])
    Ds18b20.ValMin[Dnum]=Ds18b20.Val[Dnum];
    OSExitDriver();//退出驱动关键程序，退出准裸机模式
    return Temperature1;
}
```

<3> void OSStartRun(void)和 void OSStopRun(void);

功能：进入全裸机模式，当前任务函数等效于裸机模式下的 main 函数。

使用范围：不限。

例：

利用任务 3 作为媒介完成必须在裸机下才完成的代码或初使化。

```
void Task3(void) //任务 2
{
    OSStopRun();//进入全裸机模式
    while(1)
     {
        /***********用户功能代码***********/

        /***********************************/
        OSStartRun();
        OSTaskStateSet(Task3, TASK_PAUSING);
     }
}
```

<4>void OSSchedSwitch (void)和 void OSContextExchangeToTask(OS_TCB* tcb);

功能：任务调度切换(切换到调度器指定的任务)，后者可指定到任意任务。

说明：查询获取下一任务并触发任务切换。

使用范围：不限。

例：

```
void OSCmdRxByte(INT8U RxByte) //该任务在串口中断函数中被调用
{
#if (OS_CMD_ALL_ENABLE == 1)
    if(OS_Cmd.RXOK_Flag==OS_FALSE)
    {
        if(OS_Cmd.RX_COUNT<OS_CMD_STR_LEN-1)
            {
                OS_Cmd.RX_BUFF[OS_Cmd.RX_COUNT++]=RxByte;

    if((OS_Cmd.RX_BUFF[OS_Cmd.RX_COUNT-1]=='/')&&(OS_Cmd.RX_BUFF[OS_Cmd.RX_COUN
T-2]=='/'))
                    {
                      OS_Cmd.RXOK_Flag=OS_TRUE;
                        OSTCBTbl[1].TaskDelayMs=0;
                        if(OS_System.RuningMode>0)//模式 1 和 2
                        {
                         /*-------------------------方式一-------------------------------*/
                            OSSchedSwitch();   //主动调度
                            //OSSchedSwitchClean(); //主动调度 并清 0 时间片计数器
                        /*--------------------------------------------------------------*/
                        /*-----------------------------方式二-------------------------------*/
                          //或直接转到指定任务运行，提高实时性。
                          //跳转前可选择需要不需要把时间片计数清 0
                         // OSTimeSliceCounterReset();//时间片计数清 0
                     // OSContextExchangeToTask(&OSTCBTbl[x]);//x=2 时对应任务 1，以此类推,
                        //此跳转函数不受优先级限制，在没有中断正在响应的情况能直接切
                         //换到指定任务，如 cpu 正在响应其他中断则等待其他中断运行完
                        //成再切换。因为是底层操作函数，请慎重使用。
                        /*--------------------------------------------------------------*/
                        }
                    }
            }
        else { OS_Cmd.RX_COUNT=0; OS_Cmd.RX_BUFF[OS_Cmd.RX_COUNT]=RxByte; }
    }
#else
  RxByte=RxByte;//防止警告
#endif
}
```

<5> INT8U OSTaskStateSet(void* Taskx,INT8U TaskState);

功能：设置指定任务的运行状态,有：TASK_RUNNING-运行态， TASK_PAUSING-暂停态

在任务没有处于 OSMutex 正在使用状态下，能设置成功，返回 OS_TRUE，如处于使

用状态，分两种情况：

1. 任务设置自己，如果当前任务处于 OSMutex 状态，则设置失败，返回 OS_FALSE；
2. 设置的是其他任务，则会把当前任务切换出去，等待目标任务释放 OSMutex，释放后会再次检测是否处于释放状态，是则设置成功，否则继续等待。

例：

```
void Task2(void)      //任务 2
{
    OSTaskStateSet(Task3, TASK_PAUSING);          //暂停任务 3，等待获得运行条件
    OSTaskStateSet(Task4, TASK_PAUSING);          //暂停任务 4，等待获得运行条件
    while(1)
      {
          OSprintf("Task2 is running\r\n");
          OS_delayMs(8000);                        //延时 8 秒
          OSTaskStateSet(Task1, TASK_PAUSING);     //廷时 8 秒后使任务 1 停止运行
          OSTaskStateSet(Task3, TASK_RUNNING);     //任务 3 开始运行
          OSTaskStateSet(Task2, TASK_PAUSING);     //任务 2 完成任务后暂停自己
      }
}
```

<6> INT8U OSTaskStateGet(void* Taskx);

功能：获取指定任务的运行状态,有：

低四位：

```
TASK_CREATING          0                    //创建态
TASK_RUNNING           1                    //运行态
TASK_PAUSING           2                    //暂停态
TASK_BACKRUNNING       3                    //后台态
TASK_DELETING          4                    //删除态
```

高 4 位：

```
TASK_UNLOCK            0                    //任务无锁态
TASK_LOCK              1                    //任务锁定态
```

例：

```
void Task1(void)      //任务 1
{
    if(OSTaskStateGet(Task3)&0x0F== TASK_CREATING) //如果任务 3 为创建态
    {
       OSTaskStateSet(Task3, TASK_RUNNING); //让任务 3 变为运行态
    }
    while(1)
    {
       OSprintf("Task1 is running\r\n");
       OS_delayMs(1000); //延时
    }
```

```
}
```

<7> void OSTaskSwitch(void* Taskx);

功能：立即切换到指定任务运行，不返回。Ps:切换任务优先级大于消息接收。可利用此函数提高实时性。

使用范围：OS 模式。非中断函数(中断函数也可，但要合理使用)

```
 void Task1(void)       //任务 1
 {
    OSTaskSwitch(Task3);       //跳到任务 3 运行并从任务 3 正常运行下去。
    while(1)
    {
       OSprintf("Task1 is running\r\n");
       OS_delayMs(1000);
    }
 }
```

<8> void OSTaskSwitchBack(void* Taskx);

功能：立即切换到指定任务运行，运行完指定任务会返回当前任务继续运行。

使用范围：OS 模式。非中断函数(中断函数也可，但要合理使用)

```
   void Task1(void)       //任务 1
    {
       OSTaskSwitchBack(Task3); //跳到任务 3 运行，过一个时间切片后返回此处继续正常运行
        while(1)
        {
              OSprintf("Task1 is running\r\n");
              OS_delayMs(1000);
        }
 }
```

<9>INT16U* OSRunTimeGet(void);

//功能：获取系统累计运行时间。指针偏移从 0-6 对应毫秒、秒、分、时、日、月、年。

```
void Task2(void) //任务 2
{
      uint16_t* OSRunTimePtr;
      while(1)
       {
           OSRunTimePtr=OSRunTimeGet();
           OSprintf("系统运行时间累计:%d 年%d 月%d 日%d 时%d 分%d 秒%dms\r\n",\
                                                          *( OSRunTimePtr+6),\
                                                          *( OSRunTimePtr+5),\
                                                          *( OSRunTimePtr+4),\
                                                          *( OSRunTimePtr+3),\
```

```
                                                    *( OSRunTimePtr+2),\
                                                    *( OSRunTimePtr+1),\
                                                    *( OSRunTimePtr+0));
        OS_delayMs(1000);//为 0 时无限延时,不占 cpu 资源
    }
}
```

(2)FSC_STOS 系统功能

<0>此处所有资源均在 fsc_stos.h 里配置。(数量增多会占用更多内存资源，请按需配置)
取值范围：1 – 65535 ,根据实标需要配置，具有高度实时性。默认数量为 4。

```
#define TIMER_SIZE                  4           //系统虚拟定时器数量
#define FLAG_SIZE                   4           //标志数量
#define FLAG_GROUP_SIZE             4           //标志群数量
#define MUTEX_SIZE                  4           //互斥数量
#define MBOX_SIZE                   4           //邮箱数量
```

OSxxxxPend(x1,x2,x3)函数(互斥量除外)：x1 -等待类型 x2 -信号量成员 x3-最长等待时间
等待类型 (在 FSC_STOS.h 开头处) ：

```
#define OSFlag_BPN                              //阻塞等待新信号量
#define OSFlag_BPO                              //阻塞等待含旧信号量
#define OSFlag_BPC                              //阻塞等待响应累计信号量
#define OSFlag_NBPC                             //非阻塞等待响应累计信号量
#define OSFlag_NBPN                             //非阻塞等待新信号量

#define OSFGroup_BPN                            //阻塞等待新信号量
#define OSFGroup_NBPN                           //非阻塞等待新信号量

#define OSMBox_BPN                              //阻塞等待新信号量
#define OSMBox_BPQ                              //阻塞等待响应队列信号量
#define OSMBox_NBPN                             //非阻塞等待新信号量
#define OSMBox_NBPQ                             //非阻塞等待响应队列信号量
```

<1>系统虚拟定时器
1.定时单位时间： 1ms。 解放硬件定时器。
2.示例：

```
#define  OSTIMER_LED      0  //使用第 0 组定时器，可以为其他组，最大组数在 fsc_stos.h 里设置
void Task1(void)       //任务 1
{
    OSprintf("       ---系统虚拟定时器测试---      \r\n");
    OSTimerReloadSet (OSTIMER_LED,1000);//定时值  1000ms
    while(1)
```

```
    {
        if(OSTimerStateGet(OSTIMER_LED))//返回 OS_TRUE 为定时完成，可省略
        {
          OSTimerReloadSet (OSTIMER_LED,1000);//设置重装载定时值
          OSprintf("定时 1000ms      成功---     \r\n");
        }
    }
}
```

<2>标志量　　(用于任务间的同步)

(1)单个标志量

```
#define OSFLAG_LED   0 //使用第 0 组标志量，可以为其他组，最大组数在 fsc_stos.h 里设置
void Task1(void)
{
    OSprintf("          OSFlag Test \r\n");
    while(1)
    {
        //等待 FLAG_LED,设等待超时时间为 5  秒
        if( OSFlagPend(OSFlag_BPN,OSFLAG_LED,5000) )
        {
            /******正常处理代码******/

            /************************/
            OSprintf("LED1 ON    正常打开     \r\n");
        }
        else
        {
            /******超时处理代码******/

           /************************/
           OSprintf("LED1 等待超时    \r\n");
        }
       //如不需要超时功能，超时时间设为 0 即可，以上代码可简化为:
       //OSFlagPend(OSFlag_BPN,OSFLAG_LED,0); //原地等待直到接收到  FLAG_LED  为止
       // printf("LED1 ON    正常打开     \r\n");
    }
}
void Task2(void)
{
    while(1)
    {
      OS_delayMs(4000); //实验时修改此延时值可看到 LED1  和  LED2  超时与不超时效果
      OSFlagPost(OSFLAG_LED);       //发送 FLAG_LED
    }
}
```

```
void Task3(void)
{
    while(1)
    {
        //等待 FLAG_LED,设等待超时时间为 3  秒
        if( OSFlagPend(OSFlag_BPN,OSFLAG_LED,3000) )
        {
            /******正常处理代码******/

            /************************/
            OSprintf("LED2 ON    正常打开     \r\n");
        }
        else
        {
            /******超时处理代码******/

            /************************/
            OSprintf("LED2 等待超时    \r\n");
        }
    }
}
```

(2)标志量群(多个标志量)

说明：标志量群成员所有成员都收到消息时，标志量群等待才会通过。

```
#define OSFLAG_LED       0   //第 0 组标志量(将  OSFlag 标志加入标志群成为其成员)
#define OSFLAG_MOTO    1   //第 1 组标志量(成员实体在  OSFlag 中，即为上面的单个标志量)
#define OSFLAG_TEMP       2          //第  2 组标志量
#define OSFLAG_GROUP_LED       0 //使用第 0 组标志量群
void Task1(void)
{
    OSprintf("          OSFGroup Test \r\n");
    OSFlagAddToGroup(OSFLAG_GROUP_LED, OSFLAG_LED);//LED  标志量加入标志量群
    OSFlagAddToGroup(OSFLAG_GROUP_LED, OSFLAG_MOTO); //MOTO  标志量加入标志量群
    OSFlagAddToGroup(OSFLAG_GROUP_LED,  OSFLAG_TEMP);  //TEMP  标志量加入标志量群
    while(1)
    {
     //等待 OSFLAG_GROUP_LED,设群等待超时时间为 10 秒,修改此值看超时效果。
     //标志量群等待，当 LED、MOTO、TEMP 三个标志量都发送完成时，群等待才完成接收
      if( OSFlagGroupPend(OSFGroup_BPN,OSFLAG_GROUP_LED,10000) )
       {
         /******正常处理代码******/

         /************************/
```

```
          OSprintf("LED1 ON    正常打开     \r\n");
        }
        else
        {
          /******超时处理代码******/

          /************************/
          OSprintf("LED1 等待超时    \r\n");
        }
    }
}
void Task2(void)
{
    while(1)
    {
      OS_delayMs(3000); //实验时通过修改此延时值可看到超时与不超时效果
      OSFlagPost(OSFLAG_LED);       //发送 FLAG_LED (LED 共计 5 秒发送一次)
      OS_delayMs(2000);
      OSFlagPost(OSFLAG_TEMP);       //发送  TEMP 标志量
    }
}
void Task3(void)
{
    while(1)
    {
      OS_delayMs(1000);
      OSFlagPost(OSFLAG_MOTO);       //发送 MOTO 标志量
      //等待 FLAG_LED,   等待超时时间为 1S+4S=5S，等待小于 5  秒会超时
      if( OSFlagPend(OSFlag_BPN,OSFLAG_LED,4000) )
      {
        /******正常处理代码******/

        /************************/
        OSprintf("LED2 ON    正常打开     \r\n");
        }
        else
        {
          /******超时处理代码******/

          /************************/
          OSprintf("LED2 等待超时    \r\n");
      }
    }
}
```

<2>互斥量

说明：多个任务共同使用同一资源，该资源任意时刻只能被一个任务使用，在该任务没有使用完成时其他任务不能使用，其他任务只能等待该任务使用完成才能竞争获取使用权。

竞争机制：Order 模式下，使用完毕的任务下一顺序任务获得。

Prio 模式下，优先级高的获得。

示例：

```
#define OSMutex_LED 0     //使用第 0 组互斥量，可以为其他组，最大组数 fsc_stos.h 里设置
void Task1(void) //任务  1
{
    OSprintf(" OSMutex Test \r\n");
    while(1)
    {
        OSMutexPend(OSMutex_LED,0); //等待进入资源独占区
                    /*----------------------开始独占资源----------------*/
 /*开始使用资源，其他同样在等待 OSMutex_LED 的任务将处于等待本任务处理完成状态*/
        OSprintf("LED1 ON -Task1 using start\r\n");//printf  使用串口  1  发送字符串
        delay_ms(1000);
        OS_delayMs(1000); //3 个延时函数模拟处理资源
        delay_ms(1000);
        OSprintf("LED1 OFF - Task1 using end \r\n\r\n ");//printf  使用串口  1  发送字符串
        OSMutexPost(OSMutex_LED);//资源使用完成，Post  释放 OSMutex_LED 标号资源
                    /*----------------------结束独占资源----------------*/
        OS_delayMs(8000);//调用  OS_delayMs  把任务切出去让其他任务使用资源
    }
}
void Task2(void) //任务  2
{
    while(1)
    {
        OSMutexPend(OSMutex_LED,0);//等待进入资源独占区
                    /*----------------------开始独占资源----------------*/
 /*开始使用资源，其他同样在等待 OSMutex_LED 的任务将处于等待本任务处理完成状态*/
        OSprintf("LED2 ON - Task2 using start \r\n");//printf  使用串口  1  发送字符串
        delay_ms(2000);
        OS_delayMs(1000); //3 个延时函数模拟处理资源
        delay_ms(1000);
        OSprintf("LED2 OFF - Task2 using end \r\n\r\n ");//printf  使用串口  1  发送字符串
        OSMutexPost(OSMutex_LED); //资源使用完成，Post  释放 OSMutex_LED 标号资源
                    /*----------------------结束独占资源----------------*/
```

```
            OS_delayMs(5000);//调用  OS_delayMs  把任务切出去让其他任务使用资源
        }
}
void Task3(void) //任务  3
{
          while(1)
          {
            OSMutexPend(OSMutex_LED,0);//等待进入资源独占区
                              /*------------------------开始独占资源-----------------*/
 /*开始使用资源，其他同样在等待 OSMutex_LED 的任务将处于等待本任务处理完成状态*/
            OSprintf("LED3 ON -Task3 using      start\r\n");//printf  使用串口  1  发送字符串
            delay_ms(1000);
            OS_delayMs(1000); //3 个延时函数模拟处理资源
            delay_ms(1000);
          OSprintf("LED3 OFF - Task3 using end \r\n\r\n ");//printf  使用串口  1  发送字符串
          OSMutexPost(OSMutex_LED); //资源使用完成，Post  释放 OSMutex_LED 标号资源
                          /*------------------------结束独占资源-----------------*/
/*结束使用资源，其他同样在等待 OSMutex_LED 的任务将竞争获得使用权*/
          OS_delayMs(1000);//调用  OS_delayMs  把任务切出去让其他任务使用资源
          }
}
```

<3>邮箱

```
#define OSMbox_LED 0 //使用第  0  组邮箱，可以为其他组，最大组数在  fsc_stos.h  里设置
void Task1(void) //任务  1
{
   uint8_t* str;
   OSprintf(" OSMbox Test \r\n");
   while(1)
   {
     str =(uint8_t*)OSMboxPend(OSMBox_BPN,OSMbox_LED,5000); //等待 Mbox_LED，阻塞等待
     if( str== (uint8_t*)0 ) { OSprintf("OSMbox  等待超时  \r\n");}
     else { OSprintf("OSMbox received: %s \r\n " ,str);}
  }
}
void Task2(void) //任务  2
{
   uint8_t str[20]="hellow world!";
   while(1)
   {
      OS_delayMs(2000);
```

```
        OSMboxPost(OSMbox_LED, (uint8_t*)str); //发送    Mbox_LED
    }
}
```

<4>系统串口指令

1>在 fsc_stos.c 文件开头处可自定义指令，支持中文或英文

2>指令为任意字符串，必须以//结尾。可自定义，保证指令以//结尾即可，如"打开任务 1//"。

3>已集成指令：

cmd/Task1/open//	打开任务 1
cmd/Task2/open//	打开任务 2
cmd/Task3/open//	打开任务 3
cmd/Task4/open//	打开任务 4
cmd/Task5/open//	打开任务 5
cmd/Task1/close//	关闭任务 1
cmd/Task2/close//	关闭任务 2
cmd/Task3/close//	关闭任务 3
cmd/Task4/close//	关闭任务 4
cmd/Task5/close//	关闭任务 5
cmd/Task1/prio=%d//	%d 表示整型数字,例：cmd/Task1/prio=8//
cmd/Task2/prio=%d//	
cmd/Task3/prio=%d//	
cmd/Task4/prio=%d//	
cmd/Task5/prio=%d//	
cmd/Task1/timeslice=%d//	设置任务 1 时间切片时间
cmd/Task2/timeslice=%d//	设置任务 2 时间切片时间
cmd/Task3/timeslice=%d//	设置任务 3 时间切片时间
cmd/Task4/timeslice=%d//	设置任务 4 时间切片时间
cmd/Task5/timeslice=%d//	设置任务 5 时间切片时间
cmd/setsystime=%d/%d/%d/%d.%d.%d//	设置系统时间：年/月/日/时.分.秒 (需要用户提供时间支持，如内部或外部 RTC)
cmd/setosofftime=%d/%d/%d/%d.%d.%d//	设置系统关闭时间　年/月/日/时.分.秒，
cmd/osmanage//	查看系统状态信息
cmd/runmode/order//	顺序运行模式
cmd/runmode/prio //	优先级运行模式
cmd/runmode/order+prio//	时间片+优先级运行模式
cmd/oson//	系统关闭
cmd/osoff//	系统打开
cmd/hardreset//	硬件重启

4>自定义新指令方法：

(1) 指令匹配有关函数介绍：

CompareCmd(“指令字符串”)： 比较接收到的指令和”指令字符串”是否一致，返回 0:一致 ，返回 1：不一致。

CompareCmdDig(“指令字符串”)：比较接收到的指令和”指令字符串”在’=’号前的字符是否一致，返回 0:一致,返回 1：不一致。

GrabCmdDig(n)： 提取接收指令中的第 n 个数字(从 0 开始)，如 cmd/led1/time=25//指令中包含 2 个数字 1 和 25，我们需要提取的有用数字是 25，则 GrabCmdDig(1)返回的值为 25。如果数字是小数，请分开成整数部分和小数部分分别读取两个数，在程序中通过合成成一个小数。目前不支持负数。小数如 cmd/ time=12.25s// ,在程序中：

```
float time;
time= GrabCmdDig(0)+ GrabCmdDig(1)/100.0。
```

(2) LED0、LED1 简单控制指令实现例程：

a.在 FSC_STOS.c 开头处添加自定义新指令。

```
/*----------------自定义指令区--------------*/
char cmd_led0_on[]={"cmd/led0/on//"}; //LED0 亮
char cmd_led0_off[]={"cmd/led0/off//"}; //LED0 灭
char cmd_led1_on[]={"cmd/led1/on//"}; //LED1 亮
char cmd_led1_off[]={"cmd/led1/off//"}; //LED1 灭

char cmd_led0_blink[]={"cmd/led0/blink/time=%d//"};//LED0 闪烁时间设置
char cmd_led1_blink[]={"cmd/led1/blink/time=%d//"};//LED1 闪烁时间设置
/*----------------------------------------*/
```

b.在 FSC_STOS.c 下拉找到 OS_TaskManage()任务函数，在函数中用户自定义指令区域中添加功能代码。

```
/*********************用户自定义指令区*******************/
if(CompareCmd(cmd_led0_on)==0)          LED0=0; //低电平亮
if(CompareCmd(cmd_led0_off)==0)         LED0=1; //高电平灭
if(CompareCmd(cmd_led1_on)==0)          LED1=0; //低电平亮
if(CompareCmd(cmd_led1_off)==0)         LED1=1; //高电平灭

if(CompareCmdDig(cmd_led0_blink)==0)    LED0_Blink_Time= GrabCmdDig(1);
if(CompareCmdDig(cmd_led1_blink)==0)    LED1_Blink_Time= GrabCmdDig(1);
/*****************************************************/
```

提示：其中,LED 的定义和 LED0_Blink_Time 和 LED1_Blink_Time 变量可用 extern 从 APP.c 全局量中声明过来，在 FSC_STOS.c 开头处。

```
/*--------------其他文件变量或函数声明区--------------*/
  #define LED0 PA1
  #define LED1 PC13
  extern uint32_t      LED0_Blink_Time;
  extern uint32_t      LED1_Blink_Time;
/*----------------------------------------------------------------------*/
```

参考总结：

<1>特点：　站在 OS 小白角度编写，尽可能让小白一看就会用的 OS：

(1)　整个 OS 只有 4 个文件(3 个内核文件+1 个 APP .c 文件)。

(2)　精炼短小，内存占用小，任务切换效率高。

<2>概述：　(1)　多任务基于时间切片运行，时间片默认为 1ms,可设置。

(2)　提供虚拟定时器(数量任意)。

(3)　提供标志量、互斥量、邮箱用于任务之间协调同步(直接使用，无需申请)。

(4)　提供串口系统指令，命令任务的运行和关闭以及其他功能(更多功能待开发)。

(5)　提供毫秒延时函数 delay_ms()和微秒延时函数 delay_us()。

(6)　提供 OS_delayMs()延时函数，具有特殊延时机制，使代码运行效率更高效。

<3>小贴士：

(1)任务堆栈大小要看任务的中间运算数据数量和局部变量大小而定，一般不小于 64。

(2)推荐用户在任务最后调用 OS_delayMs()延时函数来明确任务的确定运行周期：

* OS_delayMs()延时的时间即为任务的运行周期。

*调用 OS_delayMs()，任务会自动退出运行，同时系统会切换到其他任务运行，

*延时结束时系统会立即切换回该任务运行。

*没有调用 OS_delayMs()的任务按顺序运行，但会随时可能被其他任务抢断运行。

*多个调用 OS_delayMs()的任务同时延时结束时，先创建的任务优先运行。

(3)运行于优先级模式时，如任务不调用 OS_delayMs()或 OSPend(信号量等待)则其他优先级低于该任务的任务将不会得到运行权而停止运行。

(4)裸机工程加入 FSC_STOS 后，用户禁止使用 systick 定时器。其他定时器正常使用。

(5)有关 FSC_STOS 的所有参数均在 fsc_stos.h 里设置。

(6)在任务中驱动外接模块时，推荐在读取前调用 OSSchedLock()来禁止任务切换，读取完闭后调用 OSSchedUnlock()来解锁任务切换，这样才能保证在读取过程中不被切换打断读取，从而保证读取成功。

(7)在外接模块或其他内置外设驱动中，驱动程序中必须使用 delay_us()及 delay_ms()进行延时，并且在 Task 任务函数中调用或其他地方调用模块驱动函数时，必须在驱动函数前先调用 OSSchedLock()来锁定任务切换，仿止任务切换出去从而打断驱动的正常读写。当调用完成驱动函数时，必须再调用 OSSchedUnlock()来解锁任务切换。

例：

```
void Task2(void) //任务 2     用作实时时间数据刷新任务
{
   OSSchedLock();//初使化模块前锁住任务切换
   DS3231_Config();//模块初使化
   DS3231_SetTime(2019,3,18,1,9,32,0); //设置初使化数据
   OSSchedUnlock();//解锁任务切换
   while(1)
   {
      OSSchedLock();//调用模块驱动函数前锁住任务切换
      DS3231_Update(&CurrentTime); //调用模块驱动函数更新模块数据
      OSSchedUnlock();//解锁任务切换
      OS_delayMs(100);//100ms 读取一次模块数据
      }
 }
```