

# FSC\_STOS 说明及例程

——望穿秋水

## 目录

(一)整体说明 .....	1
(二)示例代码 .....	1
(1)任务功能函数 .....	1
(2)FSC_STOS 系统功能 .....	3

## (一)整体说明

### 1.FSC\_STOS 介绍:

- <1> 基于时间片设计。
- <2> 界面简洁、直观。
- <3> 文件总共 4 个。
- <4> 提供：虚拟定时器、标志量、互斥量、邮箱。
- <5> 系统串口指令简洁、可自定义。
- <6> 为新手而设计，一看就会用的 OS。

2.默认为 stm32f1 系列(Cortex-M3 内核)芯片，如改为 stm32f4 系列(Cortex-M4 内核)时请把 fsc\_stos.h 里的 stm32f10x.h 改为 stm32f4 对应的头文件。

3.文档适用版本：FSC\_STOS V4.2 以上。

## (二)示例代码

### (1)任务功能函数

#### <1> void 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  
如果暂停的是自己，则会立即切换到下一任务运行，如果暂停的是别的任务则正常运行。

```
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)
{
    printf("Task2 is running\r\n");
    OS_delayMs(8000); //延时 8 秒
    OSTaskStateSet(Task1, TASK_PAUSING); //延时 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
}
}

```

## <2> INT8U OSTaskStateGet(void\* Taskx);

功能：获取指定任务的运行状态,有：

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

void Task1(void) //任务 1

```

{
    If(OSTaskStateGet(Task3)== 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)
    {
        printf("Task1 is running\r\n");
        OS_delayMs(1000); //延时
    }
}

```

## <3> void OSTaskSwitch(void\* Taskx);

//功能：立即切换到指定任务运行，并按顺序运行下去。Ps:切换任务优先级大于消息接收。

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)
    {
        printf("Task1 is running\r\n");
        OS_delayMs(1000);
    }
}

```

## <4> 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。

```

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)
    {
        printf("Task1 is running\r\n");
        OS_delayMs(1000);
    }
}

```

## (2)FSC\_STOS 系统功能

<0>此处所有资源均在 fsc\_stos.h 里配置。(数量增多会拖慢系统，请根据实际需要的数量配置)  
取值范围：1 – 65535 ,根据实标需要配置，具有高度实时性。

```

#define TIMER_SIZE      8      //系统虚拟定时器数量
#define FLAG_SIZE       8      //标志数量
#define FLAG_GROUP_SIZE 8      //标志群数量
#define MUTEX_SIZE      8      //互斥数量
#define MBOX_SIZE       8      //邮箱数量

```

### <1>系统虚拟定时器

1.定时单位时间：时间片(默认 1ms),适用于 100ms 以上不精确定时。

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);//定时初值 1000 个时间片(1000ms)
    while(1)
    {
        if(OSTimerStateGet(OSTIMER_LED))//返回 OS_TRUE 为定时完成，可省略
        {
            OSTimerReloadSet(OSTIMER_LED,1000);//重装定时值
            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_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_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); //6 秒会超时，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
    }
}

void Task3(void)
{
    while(1)
    {
        //等待 FLAG_LED,设等待超时时间为 3 秒
        if( OSFlagPend(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)
{
    printf("    OSFlag 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 15 秒
        //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(OSFLAG_GROUP_LED,15000) )
        {
            /*****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 MOTO 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 6S, 等待小于 5 秒会超时
        if( OSFlagPend(OSFLAG_LED,6000) )
        {
            /*****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 LED，在某个任务没有使用完成时，其他任务处于等待状态。

示例：

```

#define OSMutex_LED 0 //使用第 0 组互斥量，可以为其他组，最大组数在 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");//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
    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");//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
        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");//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_LED,5000); //等待 Mbox_LED，等待期间将阻塞在此处
        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/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

#### 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)PS: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。