

FSC-OS 手册

一、简介

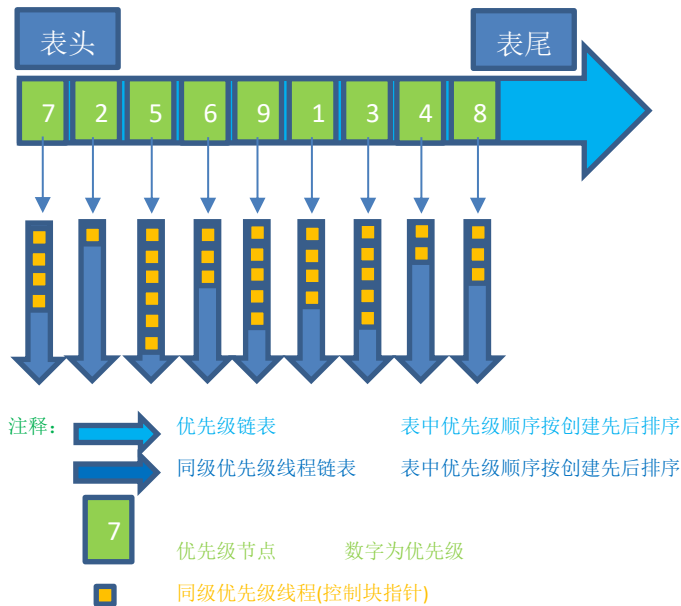
FSC-OS (@Angle_145)是基于多应用多线程的一个操作系统，线程集优先级和时间切片于一体，充分利用 CPU 资源，让程序执行更高效。目前刚完成初版，只具备内核功能。后续版本会持续更新，此手册仅适用于 V1.0.4 版本。

源码地址：

(1) Github 链接：https://github.com/Angle145/FSC_STOS.git

(2) BaiduYun 链接：<https://pan.baidu.com/s/1Ec9nZxiLMtL9obxawFt7OA> 提取码:i6k1

1. 内核简介：多线程运行规则基于优先级，同级优先级基于时间切片运行。所有创建线程均被安装到一个优先级链表中，每个链表节点为某个优先级，链表不存在相同优先级节点。每个节点也是一个链表，该链表内安装有该节点优先级的所有同级线程。例：在系统中先后创建了 33 个线程，优先级种类共有 9 种，每种优先级包含



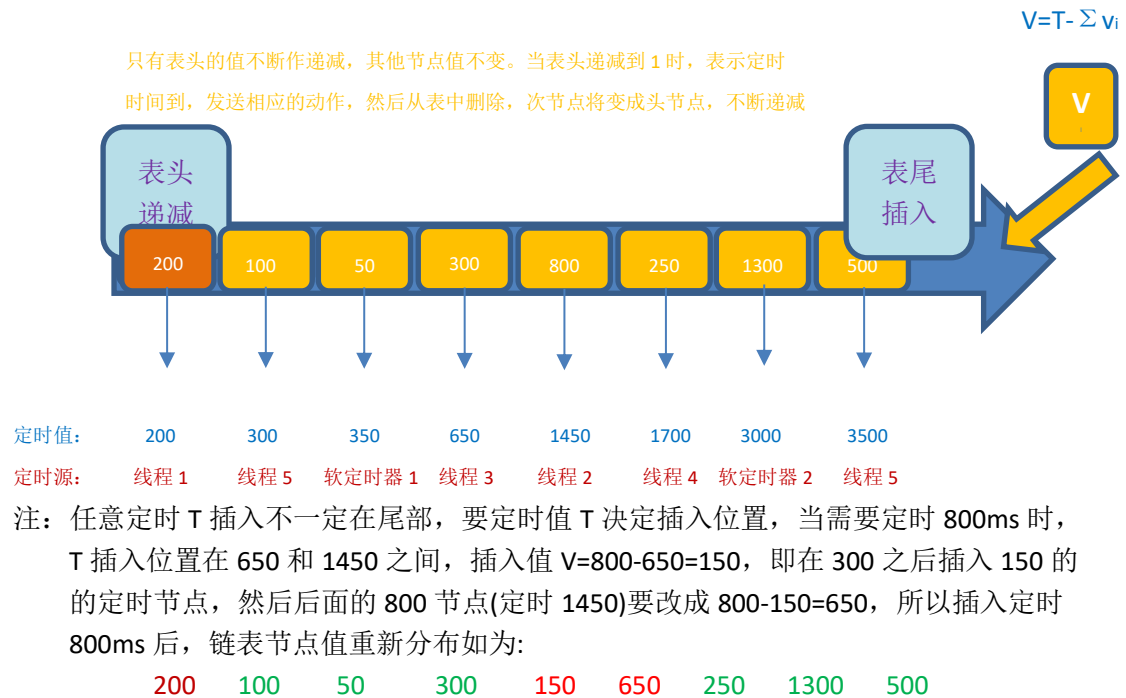
的线程数量各不同，具体如图，图中绿色为优先级链表节点，数字表示优先级，橙色为同级优先级线程，在寻找最高优先级时，先对优先级链表进行扫描，查找出最高优先级节点，再在节点链表中查找到具体线程运行(按顺序运行)。

2. 系统通用软定时器：系统内所有用到定时延时的都基于此软定时器。
系统内核统一定时插入函数：（用户不可用）

```
void os_timer_add(os_timer_type timer_id,os_u32* para,os_u32 time_ticks)
```

系统提供的软件定时器函数：（用户可用）

```
//软件定时器
os_timer* os_timer_new_create(void (*timeout_callback)(void),os_timer_mode_type tmode,os_u32 time_ticks);//创建系统
void os_timer_delete(os_timer* timer); //系统软定时器删除
void os_timer_sleep(os_timer* timer); //系统软定时器休眠
void os_timer_reload_set(os_timer* timer,os_u32 time_ticks); //设置软定时器自动重装值
void os_timer_value_set(os_timer* timer,os_u32 time_ticks); //设置软定时器定时值
os_bool os_timer_state_get(os_timer* timer); //系统软定时器定时状态获取
```



3. 线程创建:

```
//          (不变)  (线程函数名) (线程名称) (线程堆栈大小)(时间片)(优先级)      (状态)      (参数) //时间片和参数暂时没用
//          app_id  app_thread app_thread_name stk_size slice prio      state      para
os_thread_new_create( app_id, thread_01, "thread_01", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程1
os_thread_new_create( app_id, thread_02, "thread_02", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程2
os_thread_new_create( app_id, thread_03, "thread_03", 128, 10, 3, os_thread_state_readying, (void*)0 ); //创建线程3

static void thread_01(void) //APP_01的线程01
{
    while(1)
    {
        os_printf("APP_01 thread_01\r\n");
        os_thread_delay(500); //1000ms执行一次任务，这个时间越小，本任务执行的间隔越小！
    }
}

static void thread_02(void) //APP_01的线程02
{
    while(1)
    {
        os_printf("APP_01 thread_02\r\n");
        os_thread_delay(1000); //1000ms执行一次任务
    }
}

static void thread_03(void) //APP_01的线程03
{
    os_u8 arr[]="hello world!";
    while(1)
    {
```

4. 应用程序创建:

```
os_type_app_id* os_app_new_create__App_01(void) //创建APP_01
{
    os_type_app_id *app_id=os_app_new_init();
    os_app_name_set(app_id,"APP_01");
    os_app_prio_set(app_id,1);
    os_app_state_set(app_id,os_app_state_creating);

    //          (不变)   (线程函数名) (线程名称) (线程堆栈大小)(时间片)(优先级)   (状态)           (参数) //时间片和参数暂时没用
    //          app_id   app_thread app_thread_name stk_size slice prio           state           para
    os_thread_new_create( app_id,   thread_01, "thread_01", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程1
    os_thread_new_create( app_id,   thread_02, "thread_02", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程2
    os_thread_new_create( app_id,   thread_03, "thread_03", 128, 10, 3, os_thread_state_readying, (void*)0 ); //创建线程3

    os_app_new_create(app_id);
    return app_id;
}

void os_main(void)
{
    App_System=os_app_create__App_System(); //创建系统APP ( 系统自带 , 用户不可更改 )
    App_01=os_app_new_create__App_01();      //创建用户APP_01
    App_02=os_app_new_create__App_02();      //创建用户APP_02
    App_03=os_app_new_create__App_03();

    os_init_and_startup();//系统初使化并启动
}
```




5. 创建软件定时器

```

static void thread_02(void) //APP_01的线程02
{
    //例程：软件定时器
    //功能：两个定时器，一个定时打印输出，一个定时删除上一个定时器
    os_timer* timer_print;
    os_timer* timer_delte;
    timer_print=os_timer_new_create(5000);//初使化创建定时器timer_print，定时5000ms
    os_timer_relaod_set(timer_print,5000);//设为循环定时，循环定时5000ms
    timer_delte=os_timer_new_create(25000);//初使化创建定时器timer_delte，定时25000ms
    while(1)
    {
        if(os_timer_state_get(timer_print)==os_true)//获取定时器timer_print定时状态
        {
            os_printf("Hello World!---- \r\n");
        }
        if(os_timer_state_get(timer_delte)==os_true)//获取定时器timer_delte定时状态
        {
            os_timer_sleep(timer_print);//休眠定时器timer_print
            os_timer_sleep(timer_delte);//休眠定时器timer_delte本身
        }
        os_printf("APP_01 thread_02\r\n");
        os_thread_delay(1000); //1000ms执行一次任务
    }
}

```

二、文件结构

	Conf.h	系统配置文件
	Global.c	
	Global.h	存放全局变量的文件
	List.c	
	List.h	链表文件，系统核心文件之一
	Mem.c	
	Mem.h	内存管理文件
	Method.c	
	Method.h	系统API函数，系统提供给用户的所有可用函数
	OS.asm	
	OS.c	内核实现文件
	OS.h	
	OS_main.c	系统主函数(入口函数)
	Shell.c	
	Shell.h	系统指令交互文件
	Signal.c	
	Signal.h	系统信号量、同步、通讯文件
	Type.c	
	Type.h	系统变量类型定义文件

三、系统使用具体介绍

(1) os_main()函数

```

void os_main(void)
{
    App_system=os_app_create__App_system(); //创建系统APP (系统自带,用户不可更改)
    App_01=os_app_new_create__App_01();      //创建用户APP_01
    App_02=os_app_new_create__App_02();      //创建用户APP_02
    App_03=os_app_new_create__App_03();

    os_init_and_startup();//系统初始化并启动
}

```

1.os_main()函数用于创建 App,创建 Appr 的函数名称格式为 app_new_create__XXX(void)

2.App_system、App_01、App_02、App_03 为全局变量,其实体在 global.c/.h 中。

Global.c 中创建

```

#include "global.h"//全局变量

/*-----

os_type_app_id *App_system;
os_type_app_id *App_01;
os_type_app_id *App_02;
os_type_app_id *App_03;

```

Global.h 中声明

```

#ifndef _OS_GLOBAL_
#define _OS_GLOBAL_
#include "type.h"
/*-----

extern os_type_app_id *App_system;
extern os_type_app_id *App_01;
extern os_type_app_id *App_02;
extern os_type_app_id *App_03;

```

(2) app_new_create__XXX(void)函数

(1)里面的步骤是固定的,按以下图中步骤创建即可。

```

os_type_app_id* os_app_new_create__App_01(void) //创建APP_01
{
    os_type_app_id *app_id=os_app_new_init();
    os_app_name_set(app_id,"APP_01");
    os_app_prio_set(app_id,1);
    os_app_state_set(app_id,os_app_state_creating);

    //          (不变) (线程函数名) (线程名称) (线程堆栈大小)(时间片)(优先级) (状态) (参数) //时间片和参数暂时没用
    //          app_id app_thread app_thread_name stk_size slice prio state para
    os_thread_new_create( app_id, thread_01, "thread_01", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程1
    os_thread_new_create( app_id, thread_02, "thread_02", 128, 10, 2, os_thread_state_readying, (void*)0 ); //创建线程2
    os_thread_new_create( app_id, thread_03, "thread_03", 128, 10, 3, os_thread_state_readying, (void*)0 ); //创建线程3

    os_app_new_create(app_id);
    return app_id;
}

```

线程数量需要多少个就创建多少个。

(3) os_thread_new_create ()函数

参数 1: app_id

参数 2: 线程名, 可任意命名, 但要和线程函数名一致

参数 3: 线程字符串名, 用于系统信息显示

参数 4: 线程堆栈, 一般不小于 32

参数 5: 时间片, 一般 1-100

参数 6: 线程运行优先级, 越大优先级越高

参数 7: 线程状态

参数 8: 系统参数, 为(void*)0

```
//          (不变)  (线程函数名) (线程名称) (线程堆栈大小)(时间片)(优先级)  (状态)          (参数) ,
//          app_id  app_thread app_thread_name stk_size slice  prio      state      para
os_thread_new_create( app_id,  thread_01,  "thread_01",  128,  10,  2,  os_thread_state_readying,  (void*)0 );
```

(3) thread_XX (void)函数

线程函数,如同裸机中的 main()函数用法一致。以下的 while(1)内的代码为示例代码,使用时删除。系统提供延时函数 os_thread_delay(u32 t), 延时时间单位为系统时钟, 系统时钟默认是 1ms。

```
static void thread_01(void) //APP_01的线程01
{
    while(1)
    {
        os_printf("APP_01 thread_01\r\n");
        os_thread_delay(500); //1000ms执行一次任务, 这个时间越小, 本任务执行的间隔越小!
    }
}
```

四、系统配置

Conf.h:

1. 内存池是系统所有内存开销的基础, 适当设置大些。
2. 时间切片设为 1-10 较为合理

```
/******os参数定义*****/
#define OS_MEMORYPOOL_SIZE  4*1024*4    //内存池大小,单位: Byte
#define THREAD_TIME_SLICE  1            //线程时间切片,单位: 1ms
#define APP_TIME_SLICE     10           //app时间切片,单位: 1ms
#define THREAD_NAME_LEN    32
#define APP_NAME_LEN       32
/******/
```

五、交互指令系统

Shell.c/.h:

1. 指令分为全局指令和系统指令。
2. 全局指令是任何时间都响应的指令, 目前有:

```
//全局指令
char cmd_help[]={ "cmd/help/" }; //指令帮助
char cmd_enter[]={ "cmd/enter/" }; //进入指令系统
char cmd_exit[]={ "cmd/exit/" }; //退出指令系统
```

3.系统指令(受限指令)是进入指令系统后才能响应的指令，目前有：

```
//系统指令
char cmd_osinformation[]={ "osinformation//"}; //查看系统状态
char cmd_hardreset[]={ "hardreset//"}; //硬件重启
char cmd_oston[]={ "oston//"}; //系统关闭
char cmd_osoff[]={ "osoff//"}; //系统打开
```

3. 指令格式：

/为分隔符，//为结束符

全局指令：cmd/+功能名+//

系统指令：功能名+//

4.自定义指令

在 shell.c 开头处创建相应字符串指令。

在 os_shell_handle_process()函数中添加相应的代码即可实现自定义指令。

如：添加 app_01 的控制指令

```
//系统指令
char cmd_osinformation[]={ "osinformation//"}; //查看系统状态
char cmd_hardreset[]={ "hardreset//"}; //硬件重启
char cmd_oston[]={ "oston//"}; //系统关闭
char cmd_osoff[]={ "osoff//"}; //系统打开
char cmd_app1_on[]={ "app1_on//"}; //app1打开
char cmd_app1_off[]={ "app1_off//"}; //app1关闭
```

a. 创建字符串指令，

```
void os_shell_handle_process(void)
{
    . . . . .
    XXXXXX
    ...
    XXXXXX
    /*****
    if(os_shell_compare(cmd_osinformation)==0) { os_information_process(); }
    if(os_shell_compare(cmd_hardreset)==0) { os_hard_reset_tips(); os_hard_reset(); }
    if(os_shell_compare(cmd_oston)==0) { os_on_tips(); os_on(); }
    if(os_shell_compare(cmd_osoff)==0) { os_off_tips(); os_off(); }
    if(os_shell_compare(cmd_app1_on)==0) { os_shell_printf("app1打开\r\n"); os_app_open(App_01); }
    if(os_shell_compare(cmd_app1_off)==0) { os_shell_printf("app1关闭\r\n"); os_app_close(App_01); }
    *****/
}
```

b. 在 void os_shell_handle_process(void)函数中加入功能代码。注：在 shell 中显示打印字符串只能使用 os_shell_printf()函数，用法和 printf 一致。

六、系统使用教程

(1) 创建一个 APP 的流程:

- 1> Target 新建一个文件夹命名为 APP_xx 名。
- 2> 新建两个空白文件，保存到上面文件夹。
- 3> 文件分别命名为 APP_xx.c、APP_xx.h。
- 4> 复制 APP_01.c 的内容到 APP_xx.c
- 5> 复制 APP_01.h 的内容到 APP_xx.h
- 6> 修改 APP_xx.c 中 #include "app_01.h" 为 #include "app_xx.h"
- 7> 修改 APP_xx.h 中 #ifndef _OS_APP_01_ #define _OS_APP_01_ 为 #ifndef _OS_APP_XX_ #define _OS_APP_XX_
- 8> 修改 APP_xx.c/APP_xx.h 中 os_type_app_id* os_app_new_create__App_01(void) 为 os_type_app_id* os_app_new_create__App_xx(void)
- 9> APP_xx.c 中的线程数量可根据需要来定，多余的可以删除。
- 10> APP_xx.c 中，每创建一个线程，就必须写一个对应线程名字的线程



```
13 os_app_state_set(app_id,os_app_state_creating);
14
15 // (不变) (线程函数名) (线程名称) (线程堆栈大小)(时间片)(优先级) (状态)
16 // app_id app_thread app_thread_name stk_size slice prio state
17 os_thread_new_create( app_id, thread_01, "thread_01", 128, 10, 2, os_thread_state_ready
18 os_thread_new_create( app_id, thread_02, "thread_02", 128, 10, 2, os_thread_state_ready
19 os_thread_new_create( app_id, thread_03, "thread_03", 128, 10, 3, os_thread_state_ready
20
21 os_app_new_init(app_id);
22 return app_id;
23 }
24
25 static void thread_01(void) //APP_01的线程01
26 {
27 while(1)
28 {
29 os_printf("APP_01 thread_01\r\n");
30 os_thread_delay(25000); //1000ms执行一次任务，这个时间越小，本任务执行的间隔越小！
31 }
32 }
33 }
```

把创建 APP 的函数名 os_app_new_create__App_xx() 复制到 os_main.c 中的 os_main() 函数中。

- 11> 创建好了就可以在线程函数 while(1) 内写代码了。
- 12> 用户初始化的代码可以放在 while(1) 前面，也可以放在 os_main.c 中的

os_user_init()函数中。

```
void os_user_init(void) //用户代码初始化函数 (用户所有初始化函数均放在此函数内, 初始化函数禁止调用os_thread_delay(
){
    USART1_Config(115200); //串口1初始化
    printf("-@OS Inside\r\n");

    os_mbox_01=os_mbox_create();
}
void os_main(void)
{
    App_System=os_app_new_create__App_System(); //创建系统APP (系统自带, 用户不可更改)
    App_01=os_app_new_create__App_01(); //创建用户APP_01
    App_02=os_app_new_create__App_02(); //创建用户APP_02

    os_init_and_startup();//系统初始化并启动
}
```

(2) 线程内的编程

1> 线系统提供 3 个延时函数供用户选择

```
//延时函数
void delay_us(os_u32 us); //微秒延时函数(阻塞, 不调度)
void delay_ms(os_u32 ms); //毫秒延时函数(阻塞, 不调度)
void os_thread_delay(os_u32 time_ticks); //线程延时函数(阻塞, 调度)
```

2> 使用 os_thread_delay()函数时, 在延时期间线程会释放 cpu 使用权直到延时完成。

3> 线程切换锁定和解锁函数

```
//线程切换锁定函数
void os_thread_sched_lock(void); //线程切换锁定函数
void os_thread_sched_unlock (void); //线程切换解锁函数
```

用户代码中, 有时候代码必须要一次性完成, 中途不能被系统切换出去, 可以使用这个函数来锁定和释放线程切换。

4> 系统软件定时器

当要求定时精度不高时, 可以使用软件定时器来定时, 可以减少对硬件的依赖。让编程回归纯代码本质, 同时也提高可移植性。

```
//软件定时器
os_timer* os_timer_new_create(void (*timeout_callback)(void), os_timer_mode_type tmode, os_u32 time_ticks); //创建系统
void os_timer_delete(os_timer* timer); //系统软定时器删除
void os_timer_sleep(os_timer* timer); //系统软定时器休眠
void os_timer_reload_set(os_timer* timer, os_u32 time_ticks); //设置软定时器自动重装值
void os_timer_value_set(os_timer* timer, os_u32 time_ticks); //设置软定时器定时值
os_bool os_timer_state_get(os_timer* timer); //系统软定时器定时状态获取
```

定时器的简单例程: 回调函数形式

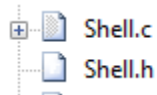
在 global.c/.h 中创建全局变量定时器 timer_printf、timer_delete

```

static void timer_print_callback(void) //timer_print定时完成回调函数
{
    os_printf("You are so good!---- \r\n");
}
static void timer_delte_callback(void) //timer_delte定时完成回调函数
{
    os_timer_delete(timer_print);//删除定时器timer_print
    os_timer_delete(timer_delte);//删除定时器timer_delte本身
}
static void thread_02(void) //APP_01的线程02
{
    //例程：软件定时器
    //功能：两个定时器，一个定时打印输出，一个定时删除上一个定时器
    timer_print=os_timer_new_create(timer_print_callback,timer_mode_type__cycle,5000);//初始化创建定时器timer_print，定
    timer_delte=os_timer_new_create(timer_delte_callback,timer_mode_type__once,25000);//初始化创建定时器timer_delte，定
    while(1)
    {
        os_printf("APP_01 thread_02\r\n");
        os_thread_delay(1000); //1000ms执行一次任务
    }
}

```

(3) 系统指令的应用



1> 指令使用

在串口助手发送 cmd/enter// ， 然后出现

```

cmd/enter//
cmd->

```

cmd->后，发送 osinformation// ， 查看系统运行信息。

```
cmd/enter//
cmd->osinformation//
+++++系统状态信息+++++
FSC-OS @Angle145
运行模式 : oder+prio
时钟节拍 : 1 ms
CPU总占用率: 96.2%
内存容量 : 16384Byte
已用内存 : 16365Byte
运行时间: 0年0月1日 0时51分19秒 696ms
系统关闭倒计时: 0年0月0日 0时0分0秒
CPU最大占用率: 96.2%
内存碎片 : 0Byte 0.0%
剩余内存 : 19Byte

+-----+
| 占用率 | 延时ms | 使用栈 | 空闲栈 | 百分比 | 时间片 | 优先级 | 状态 | 线程名 | 程序名 |
| CPU   | DlyTime | Used   | Free   | Per    | TimeSlice | Prio   | State | ThreadName | AppName |
+-----+
| 3.8%  | 0       | 183    | 201    | 47.7%  | 0         | 0      | 0:0   |            | APP_SYSTEM |
| 3.8%  | 0       | 47     | 81     | 36.7%  | 1         | 0      | 0:2   | thread_idle |            |
| 0.0%  | 0       | 136    | 120    | 53.1%  | 1         | -1     | 0:2   | thread_manager |            |
+-----+
| 0.1%  | 0       | 147    | 237    | 38.3%  | 0         | 1      | 0:0   |            | APP_01 |
| 0.1%  | 25000   | 47     | 81     | 36.7%  | 10        | 3      | 0:3   | thread_01   |            |
| 0.0%  | 0       | 47     | 81     | 36.7%  | 10        | 3      | 0:2   | thread_02   |            |
| 0.0%  | 3000    | 53     | 75     | 41.4%  | 10        | 4      | 0:3   | thread_03   |            |
+-----+
| 0.0%  | 0       | 148    | 236    | 38.5%  | 0         | 2      | 0:0   |            | APP_02 |
| 0.0%  | 4000    | 47     | 81     | 36.7%  | 10        | 5      | 0:3   | thread_01   |            |
| 0.0%  | 8000    | 47     | 81     | 36.7%  | 10        | 5      | 0:3   | thread_02   |            |
| 0.0%  | 0       | 54     | 74     | 42.2%  | 10        | 7      | 0:2   | thread_03   |            |
+-----+
cmd->
```

2> 指令自定义添加

步骤一：在 shell.c 开头处新建一条自定义命令字符串

```
1 #include "shell.h"
2
3 //全局指令
4 char cmd_help[]={"cmd/help//"}; //指令帮助
5 char cmd_enter[]={"cmd/enter//"}; //进入指令系统
6 char cmd_exit[]={"cmd/exit//"}; //退出指令系统
7
8 //系统指令
9 char cmd_osinformation[]={"osinformation//"}; //查看系统状态
10 char cmd_hardreset[]={"hardreset//"}; //硬件重启
11 char cmd_oston[]={"oston//"}; //系统关闭
12 char cmd_osoff[]={"osoff//"}; //系统打开
13 char cmd_app_close[]={"app_close//"};
14
15 OS_SHELL_CMD os_shell_cmd;
16 OS_SHELL_TIME os_shell_time;
17
18 void os_shell_input(os_u8 rx_byte)
19 {
20     if(os_shell_cmd.bool_rx_ok==os_false)
21     {
```

往下拉找到

`void os_shell_handle_process(void)`

函数，在该函数里

找到如下图位置，加入响应命令的代码。

```

456                                     );
457                                     OSDisp_osoff_delaytime(); }
458 #endif
459 /*****
460 if(os_shell_compare(cmd_osinformation)==0) { os_information_process(); }
461 if(os_shell_compare(cmd_hardreset)==0) { os_hard_reset_tips(); os_hard_reset(); }
462 if(os_shell_compare(cmd_oston)==0) { os_on_tips(); os_on(); }
463 if(os_shell_compare(cmd_osoff)==0) { os_off_tips(); os_off(); }
464 // if(os_shell_compare(cmd_runmode_order)==0) { OS_System.RuningMode=0; OSDisp_runmode_order(); }
465 // if(os_shell_compare(cmd_runmode_prio)==0) { OS_System.RuningMode=1; OSDisp_runmode_prio(); }
466 // if(os_shell_compare(cmd_runmode_order_prio)==0){ OS_System.RuningMode=2; OSDisp_runmode_order_prio(); }
467 if(os_shell_compare(cmd_app_close)==0) { /*执行APP关闭的函数*/ }
468
469 /*****
470 os_shell_rx_buff_clean();
471 os_shell_printf("%cmd->");
472 }
473 }
474 os_shell_cmd.rx_counter=0;
475 os_shell_cmd.bool_rx_ok=os_false;
476 }

```

其中 if(os_shell_compare(命令)==0) { }这个格式是固定的。