硬编码还是读取配置文件

`apps\earphone\user_cfg.c`：

```
#define USE_CONFIG_BIN_FILE                 0

#define USE_CONFIG_STATUS_SETTING           1                       //状态设置，包括灯状态和提示音
#define USE_CONFIG_AUDIO_SETTING            USE_CONFIG_BIN_FILE     //音频设置
#define USE_CONFIG_CHARGE_SETTING           USE_CONFIG_BIN_FILE     //充电设置
#define USE_CONFIG_KEY_SETTING              USE_CONFIG_BIN_FILE     //按键消息设置
#define USE_CONFIG_MIC_TYPE_SETTING         USE_CONFIG_BIN_FILE     //MIC类型设置
#define USE_CONFIG_LOWPOWER_V_SETTING       USE_CONFIG_BIN_FILE     //低电提示设置
#define USE_CONFIG_AUTO_OFF_SETTING         USE_CONFIG_BIN_FILE     //自动关机时间设置
#define USE_CONFIG_COMBINE_VOL_SETTING      1                     //联合音量读配置
```

`USE_CONFIG_BIN_FILE`定义的都是需要硬编码配置的。

灯效接口

`include_lib\driver\cpu\br36\asm\pwm_led.h`：

主要灯效模式

头文件中定义了多种灯效模式，通过 `pwm_led_mode_set(模式)`函数来设置：

1. **基本模式**:
    - `PWM_LED_ALL_OFF` - 全灭
    - `PWM_LED_ALL_ON` - 全亮
2. **LED0(蓝灯)控制**:
    - `PWM_LED0_ON/OFF` - 蓝灯亮/灭
    - `PWM_LED0_SLOW_FLASH` - 蓝灯慢闪
    - `PWM_LED0_FAST_FLASH` - 蓝灯快闪
    - `PWM_LED0_DOUBLE_FLASH_5S` - 蓝灯5秒内闪两下
    - `PWM_LED0_ONE_FLASH_5S` - 蓝灯5秒闪一下
    - `PWM_LED0_BREATHE` - 蓝灯呼吸效果
3. **LED1(红灯)控制**:
    - `PWM_LED1_ON/OFF` - 红灯亮/灭
    - `PWM_LED1_SLOW_FLASH` - 红灯慢闪
    - `PWM_LED1_FAST_FLASH` - 红灯快闪
    - `PWM_LED1_DOUBLE_FLASH_5S` - 红灯5秒内闪两下
    - `PWM_LED1_ONE_FLASH_5S` - 红灯5秒闪一下
    - `PWM_LED1_BREATHE` - 红灯呼吸效果
4. **组合效果**:
    - `PWM_LED0_LED1_FAST_FLASH` - 红蓝交替快闪
    - `PWM_LED0_LED1_SLOW_FLASH` - 红蓝交替慢闪
    - `PWM_LED0_LED1_BREATHE` - 红蓝交替呼吸

配置参数

在头文件开头可以配置以下参数:

基本配置:

```
#define CFG_LED0_LIGHT          100  // LED0(红灯)亮度(10-500)
#define CFG_LED1_LIGHT          100  // LED1(蓝灯)亮度(10-500)
```

闪烁参数:

```
// 快闪配置
#define CFG_SINGLE_FAST_FLASH_FREQ      500  // 快闪频率(ms)
#define CFG_SINGLE_FAST_LIGHT_TIME      100  // 快闪亮灯时间(ms)

// 慢闪配置
#define CFG_SINGLE_SLOW_FLASH_FREQ      3000  // 慢闪频率(ms)
#define CFG_SINGLE_SLOW_LIGHT_TIME      100   // 慢闪亮灯时间(ms)
```

频率（Flash Frequency）

- **定义**：两次亮灯开始之间的时间间隔
- **单位**：毫秒(ms)
- **作用**：控制LED闪烁的快慢
- 示例：
    - 500ms = 每秒闪约2次
    - 1000ms = 每秒闪1次
    - 2000ms = 每2秒闪1次

亮灯时间（Light Time）

- **定义**：每次亮灯持续的时间
- **单位**：毫秒(ms)
- **作用**：控制每次亮灯持续多久
- 示例：
    - 100ms = 亮0.1秒
    - 500ms = 亮0.5秒
    - 1000ms = 亮1秒

实际效果对比

情况1：频率=1000ms，亮灯时间=200ms

```
亮   灭    亮   灭    亮   灭
|----|-----|----|-----|----|
0   200  1000 1200  2000 2200 (ms)
```

- 每1000ms闪一次
- 每次亮200ms
- 亮的时间占20%

注意事项

亮灯时间必须小于频率（周期）时间，这是由LED控制的基本原理决定的。

为什么亮灯时间必须≤频率?

- **周期完整性**：一个完整的闪烁周期包括"亮"和"灭"两个阶段
- 数学关系：频率周期 = 亮灯时间 + 灭灯时间
- 因此：亮灯时间 ≤ 频率周期

代码中的保护机制

在`_pwm_led_one_flash_display`函数中有相关保护逻辑：

```
if (start_light_time != -1) {
    if (start_light_time >= period) {  // 如果开始亮灯时间≥周期
        led_err("start_light_time config err");
        _pwm_led_off_display(); // 直接关闭LED
        return;
    }
    // ... 其他处理 ...
}
```

这种配置会导致不可预测的行为，代码中会有保护机制将其视为错误处理。

呼吸灯参数:

```
#define CFG_LED_BREATH_TIME          1000  // 呼吸周期(ms)
#define CFG_LED0_BREATH_BRIGHT       400   // LED0呼吸亮度(0-500)
#define CFG_LED1_BREATH_BRIGHT       400   // LED1呼吸亮度(0-500)
#define CFG_LED_BREATH_BLINK_TIME    1000  // 呼吸间隔灭灯时间(ms)
```

`CFG_LED_BREATH_TIME`：

- **作用**：控制LED完成一次"灭->亮->灭"的完整呼吸效果所需的时间
- 过程：
    - 从灭(0亮度)渐变到最亮(由`CFG_LED0/1_BREATH_BRIGHT`决定)
    - 然后从最亮渐变回灭(0亮度)
- 示例：如果设置为1000ms，那么：
    - 前500ms：亮度从0%渐变到100%
    - 后500ms：亮度从100%渐变回0%

`CFG_LED_BREATH_BLINK_TIME`：

- **作用**：在一次完整的呼吸周期完成后，LED保持熄灭状态的时间

- 过程：
    - 在完成"灭->亮->灭"的呼吸效果后
    - LED会保持熄灭状态，持续这个参数设置的时间
    - 然后才开始下一个呼吸周期

完整周期

一个完整的LED呼吸循环包括：

1. 呼吸阶段(`CFG_LED_BREATH_TIME`)：
    - 灭 -> 亮 -> 灭
2. 灭灯延时阶段(`CFG_LED_BREATH_BLINK_TIME`)：
    - 保持灭灯状态

硬件配置:

```
// 双IO模式支持(取消注释启用)
//#define PWM_LED_TWO_IO_SUPPORT
#define PWM_LED_TWO_IO_CONNECT      1  // 0:LED共地, 1:LED共VCC

// 三IO模式支持(取消注释启用)
//#define PWM_LED_THREE_IO_SUPPORT
```

注意事项

1. 默认LED0是蓝灯，低电平亮；LED1是红灯，高电平亮
2. 使用前需要正确配置IO模式和对应的GPIO引脚
3. 呼吸灯效果需要PWM支持
4. 修改配置参数后需要重新编译程序

灯效的调用链

SDK 启动后进入耳机模式时，灯效相关的函数调用链：

1.初始化阶段:

- 系统启动时，会调用 `board_init()` 函数
- 在 `board_init()` 中会调用 `board_devices_init()`
- `board_devices_init()`中会调用`pwm_led_init(&pwm_led_data)`初始化 LED 驱动
    - 像STM32一样，底层的硬件配置模块的使能。

2.LED 配置:

- `include_lib\system\user_cfg.h`定义状态数据结构的声明与定义
- LED 的默认配置在`status_config`结构体中定义，包含了各种状态下的 LED 显示模式
- 例如:
    - 充电开始: PWM_LED1_ON (红灯常亮)
    - 充电完成: PWM_LED0_ON (蓝灯常亮)
    - 开机: PWM_LED0_ON (蓝灯常亮)
    - 低电量: PWM_LED1_SLOW_FLASH (红灯慢闪)
    - 蓝牙初始化完成: PWM_LED0_LED1_SLOW_FLASH (红蓝灯交替慢闪)
    - 蓝牙连接成功: PWM_LED0_ONE_FLASH_5S (蓝灯5秒闪一次)
    - 蓝牙断开: PWM_LED0_LED1_FAST_FLASH (红蓝灯交替快闪)

```
//status bin结构体
typedef struct __STATUS {
    u8 charge_start;    //开始充电
    u8 charge_full;     //充电完成
    u8 power_on;        //开机
    u8 power_off;       //关机
    u8 lowpower;        //低电
    u8 max_vol;         //最大音量
    u8 phone_in;        //来电
    u8 phone_out;       //去电
    u8 phone_activ;     //通话中
    u8 bt_init_ok;      //蓝牙初始化完成
    u8 bt_connect_ok;   //蓝牙连接成功
    u8 bt_disconnect;   //蓝牙断开
    u8 tws_connect_ok;  //TWS连接成功
    u8 tws_disconnect;  //TWS蓝牙断开
} _GNU_PACKED_ STATUS;

typedef struct __STATUS_CONFIG {
    u8 sw;
    STATUS led;    //led status
```

```
    STATUS tone;    //tone status
} _GNU_PACKED_ STATUS_CONFIG;
```

- 把所有的状态使用结构体封装，初始化两个结构体变量，用作LED显示和tone提示音的标识。这个两个结构体变量共同组成一个状态配置结构体。定义了各种状态的led和tone。

具体定义各状态的led以及提示音是在各板级文件中初始化的：

apps\earphone\board\br36\board_ac700n_demo.c

- **初始化为结构体变量**

```
/*各个状态下默认的闪灯方式和提示音设置，如果USER_CFG中设置了
USE_CONFIG_STATUS_SETTING为1，则会从配置文件读取对应的配置来填充改结构体*/
STATUS_CONFIG status_config = {
    //灯状态设置
    .led = {
        .charge_start  = PWM_LED1_ON,
        .charge_full   = PWM_LED0_ON,
        .power_on      = PWM_LED0_ON,
        .power_off     = PWM_LED1_FLASH_THREE,
        .lowpower      = PWM_LED1_SLOW_FLASH,
        .max_vol       = PWM_LED_NULL,
        .phone_in      = PWM_LED_NULL,
        .phone_out     = PWM_LED_NULL,
        .phone_activ   = PWM_LED_NULL,
        .bt_init_ok    = PWM_LED0_LED1_SLOW_FLASH,
        .bt_connect_ok = PWM_LED0_ONE_FLASH_5S,
        .bt_disconnect = PWM_LED0_LED1_FAST_FLASH,
        .tws_connect_ok = PWM_LED0_LED1_FAST_FLASH,
        .tws_disconnect = PWM_LED0_LED1_SLOW_FLASH,
    },
    //提示音设置
    .tone = {
        .charge_start  = IDEX_TONE_NONE,
        .charge_full   = IDEX_TONE_NONE,
        .power_on      = IDEX_TONE_POWER_ON,
        .power_off     = IDEX_TONE_POWER_OFF,
        .lowpower      = IDEX_TONE_LOW_POWER,
        .max_vol       = IDEX_TONE_MAX_VOL,
        .phone_in      = IDEX_TONE_NONE,
        .phone_out     = IDEX_TONE_NONE,
        .phone_activ   = IDEX_TONE_NONE,
        .bt_init_ok    = IDEX_TONE_BT_MODE,
```

```
        .bt_connect_ok = IDEX_TONE_BT_CONN,
        .bt_disconnect = IDEX_TONE_BT_DISCONN,
        .tws_connect_ok    = IDEX_TONE_TWS_CONN,
        .tws_disconnect    = IDEX_TONE_TWS_DISCONN,
    }
};

#define __this (&status_config)
```

- **疑问：灯效应该是读取配置文件的啊？但是配置工具根本不生效。提示音倒是可以。**
- 宏的作用

3.状态更新:

- 当系统状态变化时，会调用 `ui_update_status(status)` 函数
- 该函数会将状态写入循环缓冲区，并调用 `ui_manage_scan(NULL)` 处理状态变化

4.LED 控制:

- `ui_manage_scan()` 是 LED 控制的核心函数
- 根据当前状态，调用`pwm_led_mode_set()`设置对应的 LED 显示模式
- 例如：
    - 当状态为`STATUS_POWERON`时，设置`PWM_LED0_ON`(蓝灯常亮)
    - 当状态为`STATUS_LOWPOWER`时，设置`PWM_LED1_SLOW_FLASH`(红灯慢闪)

PWM 驱动:

- `pwm_led_mode_set()`函数会根据传入的模式参数，配置 PWM 输出，控制 LED 的亮灭和闪烁

主要函数调用关系

```
board_init()
  └── board_devices_init()
      └── pwm_led_init(&pwm_led_data)  // 初始化 LED 驱动

// 当系统状态变化时
ui_update_status(status)  // 更新状态
  └── ui_manage_scan(NULL)  // 处理状态变化
      └── pwm_led_mode_set(mode)  // 设置 LED 显示模式
          └── 配置 PWM 输出，控制 LED
```

ui_update_status函数

```
// 更新UI状态函数，用于处理设备状态变化时的LED灯效更新
// @param status 传入的新状态值（如STATUS_POWERON/STATUS_POWEROFF等）
void ui_update_status(u8 status)
{
    // 获取LED配置信息指针，通过this宏获取板级文件中各状态led的配置信息
    STATUS *p_led = get_led_config();

    // 如果 UI 子系统未初始化（缓冲区未配置），先执行初始化
    if (!sys_ui_var.ui_init_flag) {    // 更新UI状态之前需先初始化ui_cbuf
        // 未初始化则执行初始化
        ui_manage_init();// 确保循环缓冲区可用
    }

    // 记录状态更新日志（用于调试追踪）
    log_info("update ui status :%d", status);

    // 特殊状态处理：当状态为开机/关机且对应LED模式为三闪时
    //p_led->power_on这个是利用this宏获取板级文件中各状态led的配置信息
    //PWM_LED1_FLASH_THREE为自定义状态，不能通过pmd_led_mode去设置
    if ((status == STATUS_POWERON && p_led->power_on == PWM_LED0_FLASH_THREE)
||
        (status == STATUS_POWEROFF && p_led->power_off ==
PWM_LED1_FLASH_THREE)) {
        // 设置LED闪烁次数为7次（包含特殊模式需要的三次闪烁）
        log_info(">>>set ui_flash_cnt");
        sys_ui_var.ui_flash_cnt = 7;
    }
```

```
    // 将新状态写入循环缓冲区（供后续处理使用）
    // cbuf_write()函数用于将数据写入循环缓冲区，参数为缓冲区指针、数据指针和数据长度
    cbuf_write(&(sys_ui_var.ui_cbuf), &status, 1);

    // 注释掉的定时器代码：原计划使用定时器触发扫描，现采用直接触发方式
    /* if (!sys_ui_var.sys_ui_timer) { */
    /*     sys_ui_var.sys_ui_timer = usr_timeout_add(NULL, ui_manage_scan,
10); */
    /* } */

    // 如果当前处于特殊闪烁阶段（1-6次之间），延迟实际处理
    if (sys_ui_var.ui_flash_cnt >= 1 && sys_ui_var.ui_flash_cnt <= 6) {
        return;
    }

    // 最终触发UI状态扫描处理（立即执行状态更新）
    ui_manage_scan(NULL);
}
```

这个函数的主要作用是将新的设备状态添加到环形缓冲区中，并触发UI状态更新。

1. **状态记录**：
    - 首先，函数将新状态写入环形缓冲区`sys_ui_var.ui_cbuf`中
    - 这是一个典型的生产者-消费者模式，这里负责生产状态数据
2. **特殊状态处理**：
    - 对于开机/关机状态，如果配置为三闪模式，会设置`ui_flash_cnt = 7`
    - 这个计数器用于控制LED的闪烁次数
3. **状态处理触发**：
    - 函数最后会调用`ui_manage_scan(NULL)`来处理状态更新
    - 这是一个直接调用，而不是通过定时器（注释掉的代码显示之前可能是通过定时器触发的）
4. **闪烁处理**：
    - 如果当前处于闪烁过程中（`ui_flash_cnt`在1-6之间），会直接返回，不进行状态处理
    - 这是为了确保闪烁效果能够完成，不被新的状态更新打断

关于你提到的"没有往下层传递状态"的问题，实际上状态是通过以下方式传递的：

1. 状态首先被写入环形缓冲区：`cbuf_write(&(sys_ui_var.ui_cbuf), &status, 1)`
2. 然后通过`ui_manage_scan(NULL)`函数从缓冲区读取并处理这些状态

这种设计有以下几个优点：

- 解耦状态更新和状态处理
- 允许在状态处理繁忙时缓冲状态更新
- 可以灵活控制状态处理的时机

疑问

`apps\earphone\board\br36\board_ac700n_demo.c`

在上面得知板级文件中已经确定好了各状态的LED灯效，为什么还要利用全局变量的环形缓冲区呢？直接传递状态设置不就可以了？

使用环形缓冲区（ring buffer）处理状态更新有以下几个重要原因，这些原因在嵌入式系统中特别重要：

1. **状态去抖动（Debouncing）**：
    - 在嵌入式设备中，状态可能会在短时间内频繁变化（比如开关机、连接/断开等）
    - 使用缓冲区可以平滑处理这些快速变化的状态，避免LED闪烁过于频繁或不稳定
2. **状态优先级处理**:
    - 某些状态可能比其他状态更重要（如低电量警告需要优先显示）
    - 缓冲区允许系统在适当的时候处理状态，确保重要状态得到及时响应
3. **状态覆盖处理**:
    - 当新状态到来时，如果当前状态正在处理中（如LED正在执行闪烁效果）
    - 缓冲区可以保存新状态，等待当前状态处理完成后再处理下一个
4. **线程/中断安全**:
    - 在中断上下文中，直接操作LED可能不安全或不可靠
    - 使用缓冲区可以在主循环中安全地处理状态更新
5. **状态序列化处理**:
    - 某些状态需要按特定顺序执行（如开机动画、关机动画）
    - 缓冲区可以确保这些状态按正确的顺序处理

6. **资源管理**:
    - 直接操作LED可能涉及硬件资源冲突
    - 集中处理可以更好地管理资源，避免冲突
7. **调试和日志记录**:
    - 通过缓冲区可以更容易地记录和追踪状态变化
    - 有助于调试和问题排查

例如，在你的代码中，当设备关机时（`POWER_OFF`状态），LED会执行三次闪烁效果（`PWM_LED1_FLASH_THREE`）。如果在这期间有其他状态变化，使用缓冲区可以确保闪烁效果完成后再处理新状态，避免LED显示混乱。

这种设计模式在嵌入式系统中很常见，特别是在处理用户界面和状态机时，它可以提高系统的可靠性和响应性。

板级文件的led配置获取

```
/*各个状态下默认的闪灯方式和提示音设置，如果USER_CFG中设置了
USE_CONFIG_STATUS_SETTING为1，则会从配置文件读取对应的配置来填充改结构体*/
STATUS_CONFIG status_config = {
    //灯状态设置
    .led = {
        .charge_start  = PWM_LED1_ON,
        .charge_full   = PWM_LED0_ON,
        .power_on      = PWM_LED0_ON,
        .power_off     = PWM_LED1_FLASH_THREE,
        .lowpower      = PWM_LED1_SLOW_FLASH,
        .max_vol       = PWM_LED_NULL,
        .phone_in      = PWM_LED_NULL,
        .phone_out     = PWM_LED_NULL,
        .phone_activ   = PWM_LED_NULL,
        .bt_init_ok    = PWM_LED0_LED1_SLOW_FLASH,
        .bt_connect_ok = PWM_LED0_ONE_FLASH_5S,
        .bt_disconnect = PWM_LED0_LED1_FAST_FLASH,
        .tws_connect_ok = PWM_LED0_LED1_FAST_FLASH,
        .tws_disconnect = PWM_LED0_LED1_SLOW_FLASH,
    },
    //提示音设置
    .tone = {
        .charge_start  = IDEX_TONE_NONE,
        .charge_full   = IDEX_TONE_NONE,
        .power_on      = IDEX_TONE_POWER_ON,
```

```
        .power_off     = IDEX_TONE_POWER_OFF,
        .lowpower      = IDEX_TONE_LOW_POWER,
        .max_vol       = IDEX_TONE_MAX_VOL,
        .phone_in      = IDEX_TONE_NONE,
        .phone_out     = IDEX_TONE_NONE,
        .phone_activ   = IDEX_TONE_NONE,
        .bt_init_ok    = IDEX_TONE_BT_MODE,
        .bt_connect_ok = IDEX_TONE_BT_CONN,
        .bt_disconnect = IDEX_TONE_BT_DISCONN,
        .tws_connect_ok    = IDEX_TONE_TWS_CONN,
        .tws_disconnect    = IDEX_TONE_TWS_DISCONN,
    }
};

#define __this (&status_config)
```

- **宏的作用**：
    - 将`__this`定义为指向`status_config`结构体的指针
    - 这样在代码的其他地方使用`__this`就相当于使用`&status_config`
- **使用原理**：
    - 这是一种在C语言中模拟面向对象编程中`this` 指针的方式
    - 通过这种方式，可以方便地访问和修改`status_config`结构体中的成员
- **实际应用**：
    - 在代码的其他地方，可以直接使用`__this->led.power_on` 来访问或修改电源开启时的LED效果
- **优势**：
    - 代码更简洁：不需要每次都写完整的`status_config.led.xxx`
    - 便于维护：如果以后需要更改结构体变量名，只需要修改宏定义即可
    - 提高可读性：`__this`更符合面向对象的编程习惯
- **注意事项**：
    - 使用前需要确保`status_config`已经被正确定义和初始化
    - 避免在头文件中定义这样的宏，可能会造成命名冲突
    - 在C++中通常使用类的`this` 指针，但在C中需要自己实现

这种设计模式在嵌入式系统中很常见，特别是在需要频繁访问配置参数的场景中。通过这种方式，可以方便地管理和访问设备的各项配置参数。

```
// 获取LED配置信息指针
STATUS *p_led = get_led_config();

STATUS *get_led_config(void)
{
    return &(__this->led);
}
```

1. **获取配置指针**:
    - `get_led_config()`函数返回一个指向`STATUS`结构体的指针
    - 这个结构体包含了所有状态对应的LED效果配置
2. **与板级配置的关联**:
    - 在`board_ac700n_demo.c`中，我们看到了`status_config` 结构体的定义
    - `get_led_config()`很可能是返回了`&status_config.led`的地址
    - 这样`p_led`就指向了板级配置中定义的LED效果配置
3. **使用方式**:
    - 后续代码可以通过`p_led->power_on`、 `p_led->power_off`等方式
    - 直接访问板级配置中定义的LED效果
4. **设计优势**:
    - 将硬件相关的配置（板级）与业务逻辑（UI管理）解耦
    - 方便不同硬件平台通过实现不同的 `get_led_config()` 来支持不同的LED配置
    - 提高了代码的可移植性和可维护性
5. **与 `__this` 宏的关系**:
    - 如果`get_led_config()` 内部使用了`__this`宏
    - 那么它实际上就是返回了`&__this->led` 的地址
    - 这样就把板级配置和UI管理模块连接起来了

调用`ui_update_status`的时机

蓝牙相关状态更新

蓝牙初始化

- **位置**: `earphone.c` 中的 `bt_connction_status_event_handler` 函数

- **调用**: `ui_update_status(STATUS_BT_INIT_OK)`
- **场景**: 蓝牙初始化完成时调用

蓝牙连接状态

- **位置**: `earphone.c` 中的 `bt_connction_status_event_handler` 函数
- **调用**:
    - `ui_update_status(STATUS_BT_CONN)` - 蓝牙连接成功
    - `ui_update_status(STATUS_BT_DISCONN)` - 蓝牙断开连接
    - `ui_update_status(STATUS_A2DP_START)` - A2DP 音频流开始
    - `ui_update_status(STATUS_A2DP_STOP)` - A2DP 音频流停止

电话状态

- **位置**: `earphone.c` 中的 `bt_connction_status_event_handler` 函数
- 调用:
    - `ui_update_status(STATUS_PHONE_INCOME)` - 有来电
    - `ui_update_status(STATUS_PHONE_OUT)` - 挂断电话
    - `ui_update_status(STATUS_PHONE_ACTIV)` - 电话激活

电源管理相关

充电状态

- **位置**: `app_charge.c` 中的各个充电处理函数
- 调用:
    - `ui_update_status(STATUS_CHARGE_START)` - 开始充电
    - `ui_update_status(STATUS_CHARGE_FULL)` - 充电完成
    - `ui_update_status(STATUS_CHARGE_CLOSE)` - 充电关闭
    - `ui_update_status(STATUS_CHARGE_ERR)` - 充电错误
    - `ui_update_status(STATUS_CHARGE_LDO5V_OFF)` - LDO5V 关闭

电源状态

- **位置**: `app_power_manage.c` 和 `earphone.c`
- **调用**:
    - `ui_update_status(STATUS_POWERON)` - 开机（在 `app_main.c` 中）

- `ui_update_status(STATUS_POWEROFF)` - 关机
- `ui_update_status(STATUS_LOWPOWER)` - 低电量
- `ui_update_status(STATUS_EXIT_LOWPOWER)` - 退出低电量模式

TWS 相关

TWS 连接状态

- **位置**: `bt_tws.c` 中的 `bt_tws_connction_status_event_handler` 和 `led_state_sync_handler` 函数
- **调用**:
  - `ui_update_status(STATUS_BT_TWS_CONN)` - TWS 连接成功
  - `ui_update_status(STATUS_BT_TWS_DISCONN)` - TWS 断开连接

其他状态

DUT 模式

- **位置**: `earphone.c` 中的 `bt_bredr_enter_dut_mode` 函数
- **调用**: `ui_update_status(STATUS_DUT_MODE)`

无线麦克风模式

- **位置**:`wireless_mic/bt/bt_status_event.c`
- **调用**:
  - `ui_update_status(STATUS_PHONE_INCOME)`
  - `ui_update_status(STATUS_PHONE_OUT)`
  - `ui_update_status(STATUS_PHONE_ACTIV)`

关键状态机处理

在 `state_machine` 函数中，也有状态更新：

- `ui_update_status(STATUS_EXIT_LOWPOWER)` - 退出低功耗模式

总结

`ui_update_status` 函数在耳机模式中被广泛用于更新各种系统状态，主要包括：

1. **蓝牙连接状态**：初始化、连接、断开等
2. **电话状态**：来电、挂断、通话中等
3. **充电状态**：开始充电、充电完成、充电错误等
4. **电源状态**：开机、关机、低电量等
5. **TWS 状态**：TWS 连接、断开等

这些状态更新会触发 LED 灯效的相应变化，通过 `ui_manage_scan` 函数根据当前状态设置 LED 的显示模式。

`ui_manage_init` 函数

```
// 如果 UI 子系统未初始化（缓冲区未配置），先执行初始化
if (!sys_ui_var.ui_init_flag) {    // 更新UI状态之前需先初始化ui_cbuf
    // 未初始化则执行初始化
    ui_manage_init();// 确保循环缓冲区可用
}

// 初始化 UI 管理模块的循环缓冲区
int  ui_manage_init(void)
{
    // 检查是否已初始化过，还要再检查一次？怕中途被修改掉？
    if (!sys_ui_var.ui_init_flag) {
        // 初始化循环缓冲区，用于存储待处理的 UI 状态事件
        cbuf_init(&(sys_ui_var.ui_cbuf), &(sys_ui_var.ui_buf), UI_MANAGE_BUF);
        sys_ui_var.ui_init_flag = 1;// 标记为已初始化
    }
    return 0;
}
```

在 `ui_update_status` 函数中，初始化检查是必要的，因为：

1. **防止未初始化的缓冲区访问**：
   `ui_cbuf` 是一个循环缓冲区，用于暂存设备状态变化事件（如开机/关机/蓝牙连接等）。如果未初始化就直接调用 `cbuf_write` 写入数据，会导致非法内存访问或数据损坏。

2. **确保状态机的正确性**：
   初始化过程分配了缓冲区的内存空间并设置读写指针。未初始化时直接操作缓冲区可能导致状态机逻辑混乱（如读写指针越界、数据覆盖等）。
3. **单次初始化保证**：
   通过 `ui_init_flag` 标志确保初始化只执行一次，避免重复初始化导致的资源浪费或竞态条件。

环形缓冲区结构体变量

apps\earphone\ui_manage.c

```
typedef struct _ui_var {
    u8 ui_init_flag;
    u8 other_status;
    u8 power_status;
    u8 current_status;
    volatile u8 ui_flash_cnt;
    cbuffer_t ui_cbuf;
    u8 ui_buf[UI_MANAGE_BUF];
    int sys_ui_timer;
} ui_var;

//初始化为变量，并给电源状态成员赋值
static ui_var sys_ui_var = {.power_status =  STATUS_NORMAL_POWER};
```

环形缓冲区相关结构体：

在 `ui_var` 结构体中，有两个关键成员用于实现环形缓冲区：

```
typedef struct _ui_var {
    // ... 其他成员 ...
    cbuffer_t ui_cbuf;         // 环形缓冲区控制结构
    u8 ui_buf[UI_MANAGE_BUF]; // 实际存储数据的缓冲区
    // ... 其他成员 ...
} ui_var;
```

其中：

- `ui_buf[UI_MANAGE_BUF]` 是实际存储数据的静态数组，大小由 `UI_MANAGE_BUF` 定义（在代码中定义为8）
- `ui_cbuf` 是 `cbuffer_t` 类型的控制结构，用于管理环形缓冲区的读写位置

初始化过程：

- **相当于传递`ui_var`结构体缓冲区相关成员地址，接收分配后环形缓冲区的起始地址。**

```
cbuf_init(&(sys_ui_var.ui_cbuf), &(sys_ui_var.ui_buf), UI_MANAGE_BUF);
```

这行代码做了以下工作：

1. 传入缓冲区控制结构`ui_cbuf`的地址
2. 传入实际存储数据的缓冲区`ui_buf` 的地址
3. 传入缓冲区大小`UI_MANAGE_BUF` （8字节）

环形缓冲区的工作原理

虽然我们看不到 `cbuffer_t`和 `cbuf_init`的具体实现，但典型的环形缓冲区实现通常包含以下字段：

```
typedef struct {
    u8 *buffer;     // 指向数据缓冲区的指针
    u16 size;       // 缓冲区总大小
    u16 read_pos;   // 读取位置
    u16 write_pos;  // 写入位置
    u16 count;      // 当前存储的数据量
} cbuffer_t;
```

为什么需要双重检查 `ui_init_flag`

```
if (!sys_ui_var.ui_init_flag) {
    cbuf_init(&(sys_ui_var.ui_cbuf), &(sys_ui_var.ui_buf), UI_MANAGE_BUF);
    sys_ui_var.ui_init_flag = 1;
}
```

这种模式称为"双重检查锁定"（Double-Checked Locking），虽然在这个简单实现中没有使用互斥锁，但它的目的是：

1. 避免重复初始化
2. 确保在多任务环境下（虽然这里可能没有）的线程安全
3. 提高性能，因为大多数情况下只需要检查一次标志位

环形缓冲区的工作流程

1. **写入数据**（在 `ui_update_status` 中）：

```
cbuf_write(&(sys_ui_var.ui_cbuf), &status, 1);
```

2. **读取数据**（可能在 `ui_manage_scan` 中）：

```
u8   get_ui_status(u8 *status)
{
    return cbuf_read(&(sys_ui_var.ui_cbuf), status, 1);
}
```

为什么使用环形缓冲区

1. **解耦生产者和消费者**：状态更新（生产者）和状态处理（消费者）可以异步进行
    a. 比如播放歌曲有LED灯效，来电也有LED灯效。
        i. 在听歌中突然来电，这时应该显示来电LED灯效，通话完成后，应该恢复播歌灯效
    b. 比如有LED灯效优先级
        i. 播歌时显示播歌LED灯效，但是突然到了低电状态，会显示低电LED灯效，低电提示音播报完成后，歌曲恢复，那么应该显示播歌LED灯效，估计播歌被存到了环形缓冲区中了。先显示优先级高的LED灯效。
2. **防止数据丢失**：在消费者处理不及时时，可以缓冲一定量的状态更新
3. **提高响应性**：生产者可以快速返回，不需要等待状态处理完成

缓冲区大小选择

`UI_MANAGE_BUF` 被定义为8，这意味着：

- 最多可以缓冲8个状态更新
- 这是一个折衷的选择，既不会占用太多内存，又能应对短时间内的状态爆发
- 如果状态更新速度持续超过处理速度，仍然可能导致状态丢失

这种实现在嵌入式系统中非常常见，特别是在处理实时事件和状态更新的场景中。

`ui_manage_scan` 函数

```
void ui_manage_scan(void *priv)
{
    // 1. 获取LED配置信息指针
    // 通过get_led_config()函数获取板级配置中定义的状态-LED映射表
    STATUS *p_led = get_led_config();
```

```
    // 2. 重置定时器标记
    // 清除当前定时器ID，表示当前定时任务已完成
    sys_ui_var.sys_ui_timer = 0;

    // 3. 打印调试信息
    // 输出当前闪烁计数器和UI状态，用于调试
    log_info("ui_flash_cnt:%d cur_ui_status:%d", sys_ui_var.ui_flash_cnt,
sys_ui_var.current_status);

    // 4. 状态更新条件判断
    // 当没有闪烁任务(ui_flash_cnt=0)或闪烁任务完成(ui_flash_cnt=7)时，才更新状态
    //从这里就能看出闪烁任务的优先级了！
    if (sys_ui_var.ui_flash_cnt == 0 || sys_ui_var.ui_flash_cnt == 7) {
        // 4.1 从环形缓冲区获取新状态
        // get_ui_status()会从环形缓冲区读取下一个状态
        if (get_ui_status(&sys_ui_var.current_status)) {
            log_info("---samson-----changed--ui_flash_cnt:%d
cur_ui_status:%d",
            sys_ui_var.ui_flash_cnt, sys_ui_var.current_status);

            // 4.2 状态分类处理
            // 将状态分为电源相关状态和其他状态
            if (sys_ui_var.current_status >= STATUS_CHARGE_START &&
                sys_ui_var.current_status <= STATUS_NORMAL_POWER) {
                // 电源相关状态（充电、低电量等）
                sys_ui_var.power_status = sys_ui_var.current_status;
            } else {
                // 其他状态（蓝牙连接、通话等）
                sys_ui_var.other_status = sys_ui_var.current_status;
            }
        }
    }

    // 5. 输出当前状态信息（调试用）
    log_info("---samson-----changed--power_status:%d other_status:%d",
           sys_ui_var.power_status, sys_ui_var.other_status);

    // 6. 定时任务调度
    if (sys_ui_var.ui_flash_cnt) {
        // 6.1 处理闪烁效果
        // 如果当前有闪烁任务，递减计数器
        sys_ui_var.ui_flash_cnt--;
        // 300ms后再次执行本函数，实现闪烁效果
```

```
        sys_ui_var.sys_ui_timer = usr_timeout_add(NULL, ui_manage_scan, 300,
1);
    } else if (get_ui_status_len()) {
        // 6.2 处理缓冲区内其他状态
        // 如果环形缓冲区中还有未处理的状态，100ms后继续处理
        sys_ui_var.sys_ui_timer = usr_timeout_add(NULL, ui_manage_scan, 100,
1);
    }

    // 7. 从机设备处理（已注释掉）
    // 如果是TWS从设备，直接关闭LED并返回
    #if 0
    if (tws_api_get_role() == TWS_ROLE_SLAVE) {
        #if (LED_IO_CHOOSE == 1)
            pwm_led_mode_set(PWM_LED_ALL_OFF);
        #elif (LED_IO_CHOOSE == 2)
            led_status_set(ALL_OFF);
        #endif
        return;
    }
    #endif

    // 8. 状态处理优先级判断
    // 如果当前不是关机状态，且电源状态不是正常供电状态，则优先处理电源相关状态
    if (sys_ui_var.other_status != STATUS_POWEROFF && sys_ui_var.power_status
!= STATUS_NORMAL_POWER) {  //关机的状态优先级要高于电源状态
        switch (sys_ui_var.power_status) {
                //各电源状态处理
        }

        // 后续会处理其他状态...
        switch (sys_ui_var.other_status) {
                //其他状态处理
        }
```

状态流转流程

1. **状态产生**：
    - 系统各个模块（如蓝牙、电源管理等）通过 `ui_update_status()` 函数将状态写入环形缓冲区

2. **状态获取**：
    - 在 `ui_manage_scan()` 函数开头，通过 `get_ui_status(&sys_ui_var.current_status)` 从环形缓冲区读取最新状态
    - 读取后更新到`current_status`，并根据状态类型保存到`power_status`或`other_status`
3. **状态处理**：
    - 根据`power_status`和`other_status` 的值，进入对应的 case 分支
    - 每个 case 分支调用`pwm_led_mode_set()`或`led_status_set()`设置对应的 LED 效果

状态处理特点

状态优先级：

```
if (sys_ui_var.other_status != STATUS_POWEROFF && sys_ui_var.power_status != STATUS_NORMAL_POWER)
```

- 电源相关状态（如低电量、充电等）有较高优先级
- 关机状态`STATUS_POWEROFF`有最高优先级

LED 控制方式：

- 通过`#if (LED_IO_CHOOSE == 1)`和`#elif (LED_IO_CHOOSE == 2)` 支持不同的 LED 控制方式
- `LED_IO_CHOOSE == 1`：使用 PWM 控制 LED
- `LED_IO_CHOOSE == 2`：使用 GPIO 控制 LED
- 调用`pwm_led_mode_set`或者`led_status_set`设置对应灯效

特殊效果处理：

- 闪烁效果通过定时器实现
- 例如关机时的三闪效果：

```
if (p_led->power_off == PWM_LED1_FLASH_THREE) {
    if (sys_ui_var.ui_flash_cnt) {
        if (sys_ui_var.ui_flash_cnt % 2) {
            led_status_set(ALL_OFF);
        } else {
            led_status_set(LED1_ON);
        }
    }
}
```

典型状态处理示例

开机状态:

```
case STATUS_POWERON:
    log_info("[STATUS_POWERON]\n");
    user_poweron_flag = 1;
    pwm_led_mode_set(PWM_LED0_ON);
    // 启动定时器处理开机动画
    if (!poweron_led_cnt) {
        poweron_led_cnt = sys_timeout_add(NULL, poweron_led_deal, 1000);
    }
    break;
```

蓝牙连接状态:

```
case STATUS_BT_CONN:
    log_info("[STATUS_BT_CONN]\n");
    pwm_led_mode_set(PWM_LED_ALL_OFF);  // 连接成功后关闭 LED
    // 启动定时器检查手机连接状态
    if(!get_phone_timer) {
        get_phone_timer = sys_timeout_add(NULL, get_phone_timer_func, 1000);
    }
    break;
```

TWS 连接状态:

```
case STATUS_BT_TWS_CONN:
    log_info("[STATUS_BT_TWS_CONN]\n");
    if (bt_tws_get_local_channel() == 'L') {
        pwm_led_mode_set(PWM_LED1_SLOW_FLASH);  // 左耳慢闪
    } else {
        pwm_led_mode_set(PWM_LED0_SLOW_FLASH);  // 右耳慢闪
    }
    break;
```

状态处理流程总结

1. 从环形缓冲区读取最新状态
2. 更新当前状态到`current_status`和对应的状态变量
3. 根据状态优先级决定处理哪个状态
4. 调用对应的 LED 控制函数设置 LED 效果
5. 如果需要特殊效果（如闪烁），启动定时器处理

这种设计使得系统能够灵活地处理各种状态，并且可以方便地扩展新的状态和对应的 LED 效果。

灯效更新函数在什么地方被调用

系统启动和电源管理

`app_main.c - check_power_on_key`:

```
ui_update_status(STATUS_POWERON);  // 系统上电初始化
```

`app_power_manage.c - app_power_event_handler`:

```
ui_update_status(STATUS_EXIT_LOWPOWER);  // 退出低功耗模式
```

蓝牙连接状态

`earphone.c - bt_connction_status_event_handler`:

```
ui_update_status(STATUS_BT_INIT_OK);    // 蓝牙初始化完成
ui_update_status(STATUS_BT_CONN);       // 蓝牙已连接
ui_update_status(STATUS_BT_DISCONN);    // 蓝牙断开连接
```

电话状态

earphone.c - bt_connction_status_event_handler:

```
ui_update_status(STATUS_PHONE_INCOME);  // 来电
ui_update_status(STATUS_PHONE_OUT);     // 去电
ui_update_status(STATUS_PHONE_ACTIV);   // 通话中
```

音乐播放状态

earphone.c - bt_connction_status_event_handler:

```
ui_update_status(STATUS_A2DP_START);    // 音乐开始播放
ui_update_status(STATUS_A2DP_STOP);     // 音乐停止
```

充电状态

app_charge.c:

```
ui_update_status(STATUS_CHARGE_START);     // 开始充电
ui_update_status(STATUS_CHARGE_FULL);      // 充电完成
ui_update_status(STATUS_CHARGE_CLOSE);     // 充电关闭
ui_update_status(STATUS_CHARGE_ERR);       // 充电错误
ui_update_status(STATUS_CHARGE_LDO5V_OFF); // 5V电源关闭
```

TWS连接状态

bt_tws.c:

```
ui_update_status(STATUS_BT_TWS_CONN);     // TWS已连接
ui_update_status(STATUS_BT_TWS_DISCONN);  // TWS断开连接
```

系统关机

```
earphone.c - sys_enter_soft_poweroff:
```

调用顺序总结

1. 系统启动时：`STATUS_POWERON`
2. 蓝牙初始化：`STATUS_BT_INIT_OK`
3. 蓝牙连接/断开：`STATUS_BT_CONN`/`STATUS_BT_DISCONN`
4. 电话相关：
    - 来电：`STATUS_PHONE_INCOME`
    - 去电：`STATUS_PHONE_OUT`
    - 通话中：`STATUS_PHONE_ACTIV`
5. 音乐播放：
    - 开始播放：`STATUS_A2DP_START`
    - 停止播放：`STATUS_A2DP_STOP`
6. 充电状态：
    - 开始充电：`STATUS_CHARGE_START`
    - 充电完成：`STATUS_CHARGE_FULL`
    - 充电错误：`STATUS_CHARGE_ERR`
7. TWS连接：
    - TWS连接：`STATUS_BT_TWS_CONN`
    - TWS断开：`STATUS_BT_TWS_DISCONN`
8. 系统关机：`STATUS_POWEROFF`

注意事项

1. 有些调用被注释掉了（如`//ui_update_status`），这些是不活跃的调用。
2. 在`bt_connction_status_event_handler`函数中有一些条件调用，具体执行哪个状态更新取决于当前的蓝牙状态。

3. 在`app_charge.c`中有多个地方会更新充电状态，这些状态会根据充电过程的不同阶段被调用。

这个函数在整个SDK中被广泛用于更新设备的LED状态，反映了设备的各种工作状态。

流程图说明

mermaid源码：

```
flowchart TD
    %% 开始节点
    START([开始]) --> INIT[系统初始化]

    %% 系统初始化
    INIT --> BT_INIT[蓝牙初始化]
    BT_INIT --> BT_OK[STATUS_BT_INIT_OK]

    %% 状态监听循环
    BT_OK --> MONITOR{系统状态监听}

    %% 充电状态检测（最高优先级）
    MONITOR -->|检测到充电| CHARGE_CHECK{充电状态}
    CHARGE_CHECK -->|开始充电| CHARGE_START[STATUS_CHARGE_START]
    CHARGE_CHECK -->|充电错误| CHARGE_ERR[STATUS_CHARGE_ERR]
    CHARGE_CHECK -->|充电完成| CHARGE_FULL[STATUS_CHARGE_FULL]

    %% TWS连接流程（第二优先级）
    MONITOR -->|TWS事件| TWS_CHECK{TWS连接状态}
    TWS_CHECK -->|连接中| TWS_CONNECTING[STATUS_BT_TWS_CONN_ING]
    TWS_CHECK -->|连接成功| TWS_CONN[STATUS_BT_TWS_CONN]
    TWS_CHECK -->|连接失败| TWS_DISCONN[STATUS_BT_TWS_DISCONN]

    %% TWS角色判断
    TWS_CONN --> ROLE_CHECK{检查TWS角色}
    ROLE_CHECK -->|主耳| MASTER_CONNECT[连接手机蓝牙]
    ROLE_CHECK -->|从耳| SLAVE_WAIT[等待主耳同步]

    %% 蓝牙连接状态（第三优先级）
    MASTER_CONNECT --> BT_STATUS{蓝牙连接状态}
    MONITOR -->|蓝牙事件| BT_STATUS
```

```
    BT_STATUS -->|连接成功| BT_CONN[STATUS_BT_CONN]
    BT_STATUS -->|连接失败| BT_DISCONN[STATUS_BT_DISCONN]

    %% 电话状态处理（第四优先级）
    BT_CONN --> PHONE_CHECK{电话状态检测}
    PHONE_CHECK -->|来电| PHONE_INCOME[STATUS_PHONE_INCOME]
    PHONE_CHECK -->|通话中| PHONE_ACTIVE[STATUS_PHONE_ACTIV]
    PHONE_CHECK -->|去电| PHONE_OUT[STATUS_PHONE_OUT]

    %% 音乐播放状态（第五优先级）
    BT_CONN --> MUSIC_CHECK{音乐状态检测}
    MUSIC_CHECK -->|开始播放| A2DP_START[STATUS_A2DP_START]
    MUSIC_CHECK -->|停止播放| A2DP_STOP[STATUS_A2DP_STOP]

    %% 特殊状态
    MONITOR -->|开关机事件| POWER_EVENT[开机/关机三闪效果]

    %% 状态处理核心流程
    CHARGE_START --> UI_UPDATE[ui_update_status函数]
    CHARGE_ERR --> UI_UPDATE
    CHARGE_FULL --> UI_UPDATE
    TWS_CONNECTING --> UI_UPDATE
    TWS_CONN --> UI_UPDATE
    TWS_DISCONN --> UI_UPDATE
    BT_CONN --> UI_UPDATE
    BT_DISCONN --> UI_UPDATE
    PHONE_INCOME --> UI_UPDATE
    PHONE_ACTIVE --> UI_UPDATE
    PHONE_OUT --> UI_UPDATE
    A2DP_START --> UI_UPDATE
    A2DP_STOP --> UI_UPDATE
    POWER_EVENT --> UI_UPDATE
    SLAVE_WAIT --> UI_UPDATE

    %% 状态处理机制
    UI_UPDATE --> CBUF[存入ui_cbuf环形缓冲区]
    CBUF --> SCAN[ui_manage_scan处理]
    SCAN --> PRIORITY{状态优先级判断}

    %% 优先级处理
    PRIORITY -->|优先级1| CHARGE_PROCESS[处理充电状态]
    PRIORITY -->|优先级2| TWS_PROCESS[处理TWS状态]
    PRIORITY -->|优先级3| BT_PROCESS[处理蓝牙状态]
```

```
    PRIORITY -->|优先级4| PHONE_PROCESS[处理电话状态]
    PRIORITY -->|优先级5| MUSIC_PROCESS[处理音乐状态]

    %% LED同步机制
    TWS_PROCESS --> SYNC_CHECK{需要双耳同步?}
    SYNC_CHECK -->|是| ANCHOR_SYNC[sniff_achor_point_hook同步]
    SYNC_CHECK -->|否| LED_CONTROL[LED灯效控制]
    ANCHOR_SYNC --> LED_CONTROL

    CHARGE_PROCESS --> LED_CONTROL
    BT_PROCESS --> LED_CONTROL
    PHONE_PROCESS --> LED_CONTROL
    MUSIC_PROCESS --> LED_CONTROL

    %% 循环监听
    LED_CONTROL --> MONITOR

    %% 样式定义
    classDef startEnd fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef chargeStatus fill:#ffebee,stroke:#c62828,stroke-width:2px
    classDef twsStatus fill:#e3f2fd,stroke:#1565c0,stroke-width:2px
    classDef btStatus fill:#e8f5e8,stroke:#2e7d32,stroke-width:2px
    classDef phoneStatus fill:#fff3e0,stroke:#ef6c00,stroke-width:2px
    classDef musicStatus fill:#f3e5f5,stroke:#7b1fa2,stroke-width:2px
    classDef processStatus fill:#fffde7,stroke:#f57f17,stroke-width:2px
    classDef decisionStyle fill:#fce4ec,stroke:#ad1457,stroke-width:2px

    class START startEnd
    class CHARGE_START,CHARGE_ERR,CHARGE_FULL,CHARGE_PROCESS chargeStatus
    class
TWS_CONNECTING,TWS_CONN,TWS_DISCONN,TWS_PROCESS,SYNC_CHECK,ANCHOR_SYNC
twsStatus
    class BT_CONN,BT_DISCONN,BT_PROCESS,BT_OK btStatus
    class PHONE_INCOME,PHONE_ACTIVE,PHONE_OUT,PHONE_PROCESS phoneStatus
    class A2DP_START,A2DP_STOP,MUSIC_PROCESS musicStatus
    class UI_UPDATE,CBUF,SCAN,LED_CONTROL processStatus
    class
MONITOR,CHARGE_CHECK,TWS_CHECK,BT_STATUS,PHONE_CHECK,MUSIC_CHECK,ROLE_CHECK,PR
IORITY decisionStyle
```