经典蓝牙与TWS的完整过程

系统初始化与应用启动

从`app_main.c`中的`app_main`函数开始：

```
void app_main()
{
    // ...系统初始化
    struct intent it;

    // 初始化意图结构体
    init_intent(&it);
    it.name = "earphone";
    it.action = ACTION_EARPHONE_MAIN;
    start_app(&it); // 启动耳机应用
}
```

耳机应用入口

`start_app`会调用**应用的状态机函数**。在`earphone.c`中，应用注册为：

```
REGISTER_APPLICATION(app_earphone) = {
    .name    = "earphone",
    .action  = ACTION_EARPHONE_MAIN,
    .ops     = &app_earphone_ops,
    .state   = APP_STA_DESTROY,
};
```

其中`app_earphone_ops`定义了两个操作函数：状态机函数事件处理函数：

- 整个SDK的运作都是基于这两个函数向外围扩散的。

```
static const struct application_operation app_earphone_ops = {
    .state_machine  = state_machine,
    .event_handler  = event_handler,
};
```

状态机部分

蓝牙相关初始化

在`state_machine`函数中，处理`APP_STA_START`状态：

```
case APP_STA_START:
        //进入蓝牙模式,UI退出充电状态
        ui_update_status(STATUS_EXIT_LOWPOWER);
        if (!it) {
            break;
        }
        switch (it->action) {
```

```
        case ACTION_EARPHONE_MAIN:
            /*
             * earphone 模式初始化
             */

            /*set_mode_normal 0:user normal parm  1:user bqb parm*/
            /* set_mode_dut 0:user normal parm  1:user bqb parm */
            /* 库里模式设置 set set_mode_normal=0,set_mode_dut=0 */

            extern void set_tx_mode(u8 set_mode_normal, u8 set_mode_dut);
            /* set_tx_mode(0,1);//量产程序set */

            clk_set("sys", BT_NORMAL_HZ);
            u32 sys_clk =  clk_get("sys");
            bt_pll_para(TCFG_CLOCK_OSC_HZ, sys_clk, 0, 0);

            // 蓝牙功能初始化
            bt_function_select_init();
            bredr_handle_register();//注册SPP数据处理回调等函数
            EARPHONE_STATE_INIT();//该函数完成蓝牙SPP连接的基础配置和SPP协议栈初始化
            DHFAppCommand_init();//samson added by 20250329
            btstack_init();//启动蓝牙协议栈，并开始运行蓝牙任务。这会最终触发蓝牙初始化完成事件。

            // 设置蓝牙版本
            set_bt_version(BLUETOOTH_CORE_SPEC_54);//不可见，跳转是直接填写宏定义的

            //TWS相关的参数配置
#if TCFG_USER_TWS_ENABLE
            tws_profile_init();//初始化TWS协议栈
            sys_key_event_filter_disable();//禁用按键过滤器，在TWS模式下防止主从耳冲突处理按键事件
#endif
            if (BT_MODE_IS(BT_FCC)) { //FCC进行频偏校准
                bt_ble_fre_offset_write_cbk(fre_offset_write_handle, fre);

                bt_ble_fre_offset_read_cbk(fre_offset_read_handle);

            }
            /* 按键消息使能 */
            sys_key_event_enable();
            sys_auto_shut_down_enable();//自动关机程序使能
            bt_sniff_feature_init();//蓝牙低功耗（协议栈级别的静态配置），这里是否涉及BLE协议？实现没看出来，BLE和SNIFF是两种不同的低功耗技术，分别适用于BLE和经典蓝牙场景
            sys_auto_sniff_controle(MY_SNIFF_EN, NULL);//应用层的运行时控制函数，动态开关SNIFF模式
            //bt_set_low_latency_mode(0);//added by samson on 20250426 for anc
            app_var.dev_volume = -1;
            break;
        case ACTION_A2DP_START:
#if (BT_SUPPORT_MUSIC_VOL_SYNC && CONFIG_BT_BACKGROUND_ENABLE)
            app_audio_set_volume(APP_AUDIO_STATE_MUSIC, app_var.bt_volume, 1);
            r_printf("BT_BACKGROUND RESET ACTION_A2DP_START STATE_MUSIC bt_volume=%d\n", app_var.bt_volume);
#endif
            a2dp_dec_open(0);
```

```
            break;
        case ACTION_BY_KEY_MODE:
            user_send_cmd_prepare(USER_CTRL_AVCTP_OPID_PLAY, 0, NULL);
            break;
        case ACTION_TONE_PLAY:
            STATUS *p_tone = get_tone_config();
            tone_play_index(p_tone->bt_init_ok, 1);
            break;
        case ACTION_DO_NOTHING:
            break;
        }
        break;
```

蓝牙功能选择初始化bt_function_select_init

此函数在蓝牙模块初始化阶段（如设备启动或模式切换）被调用，负责动态配置蓝牙协议栈的关键参数，确保设备在不同硬件平台、编译选项和使用场景下能够正常运行。其核心目标是：

1. **适配硬件差异**（如芯片型号、MAC 地址分配）。
2. **优化性能与功耗**（如 QoS、增强电源控制）。
3. **支持多协议功能**（如 BLE、mSBC/AAC 编码）。
4. **提供调试与测试能力**（如电量上报、SDK 版本获取）。

通过条件编译（如 `#if TCFG_USER_BLE_ENABLE`）实现功能开关，确保代码灵活性与可移植性。

`bt_function_select_init`设置蓝牙参数：

```
void bt_function_select_init()
{
    // 设置蓝牙连接参数
    __set_user_ctrl_conn_num(TCFG_BD_NUM);//设置最大连接设备数量（TCFG_BD_NUM定义）
    __set_support_msbc_flag(1);//启用mSBC编码
    __set_aac_bitrate(131 * 1000);//设置AAC编码比特率为131kbps
    __set_support_aac_flag(1);//强制启用AAC音频编码支持

    // 设置蓝牙连接超时参数
    __set_page_timeout_value(8000);//设置蓝牙页面超时时间为8000ms（回连搜索时间）
    __set_super_timeout_value(8000);// 设置超级定时器超时时间为8000ms（主机连接超时参数）

    // 配对参数设置
    __set_simple_pair_param(3, 0, 2);//设置简单配对参数

    // 其他BLE相关初始化
    #if TCFG_USER_BLE_ENABLE
    {
        u8 tmp_ble_addr[6];
        memcpy(tmp_ble_addr, (void *)bt_get_mac_addr(), 6);
        le_controller_set_mac((void *)tmp_ble_addr);
    }
    #endif

    // 设置增强功率控制
    set_bt_enhanced_power_control(1);// 启用增强型电源控制（LE Enhanced Power Control）
```

```
    // 设置延迟报告时间
    set_delay_report_time(CONFIG_A2DP_DELAY_TIME);//设置A2DP媒体数据延迟报告时间
}
```

- 设置蓝牙连接参数
    - 设置最大连接设备数量（**宏定义**）
- 设置蓝牙连接超时参数
    - 设置蓝牙页面超时时间为8000ms（回连搜索时间）
    - 设置超级定时器超时时间为8000ms（主机连接超时参数）
        - **主机回连也会超时？？？不是什么都是会回连从机吗？？？**
- 配对参数设置

蓝牙事件处理注册bredr_handle_register

从 `bredr_handle_register` 的现有代码看，它注册的是**上层业务逻辑的回调**，而非底层事件处理函数：

```
// 注册SPP数据处理回调（仅限SPP Profile）
spp_data_deal_handle_register(...);

// 注册快速测试接口（测试盒专用）
bt_fast_test_handle_register(...);

// 音量同步回调（音乐/通话音量联动）
music_vol_change_handle_register(...);

// 电量显示回调（手机端显示电量）
get_battery_value_register(...);

// DUT模式处理函数（产线测试用）
bt_dut_test_handle_register(...);
```

这些是**特定功能模块的回调**，不涉及蓝牙协议栈事件（如连接、配对、HCI状态）。

~~`bredr_handle_register` 注册蓝牙事件处理函数：~~

疑问

这一些注册的函数在什么时候被调用？在哪里可以看到调用？

比如在earphone.c中注册了耳机应用：

```
/*
 * 注册earphone模式
 */
REGISTER_APPLICATION(app_earphone) = {
    .name    = "earphone",
    .action  = ACTION_EARPHONE_MAIN,
    .ops     = &app_earphone_ops,
    .state   = APP_STA_DESTROY,
};
```

在app_main.c中的app_main中发生了调用：

```
        init_intent(&it);
        it.name = "earphone";
        it.action = ACTION_EARPHONE_MAIN;
        start_app(&it);
```

SPP数据处理回调函数 (spp_data_deal_handle_register)

```
spp_data_deal_handle_register(user_spp_data_handler);  // 在线调试模式
spp_data_deal_handle_register(spp_data_handler);       // 普通模式
```

- **调用时机**：当蓝牙SPP（Serial Port Profile）连接建立后，每当从主设备（如手机）接收到SPP数据包时
- **用途**：处理通过蓝牙SPP协议传输的串口数据
- **注释信息**：在头文件中定义为"支持串口功能的数据处理接口"
- 根据应用场景不同选择不同的处理函数：
    - 在线调试模式使用user_spp_data_handler
    - 普通模式使用spp_data_handler
- 位置呢？源码不可见？？？

蓝牙快速测试接口 (bt_fast_test_handle_register)

```
bt_fast_test_handle_register(bt_fast_test_api);
```

- **调用时机**：当蓝牙设备被测试盒连接上并进入快速测试模式时
- **用途**：用于工厂测试场景，提供蓝牙性能和功能的快速测试能力
- **注释信息**：在wireless_mic模块中注释为"被测试盒链接上进入快速测试回调"
- 此回调函数通常用于生产测试阶段，帮助验证蓝牙设备的基本功能

音乐音量同步回调函数 (music_vol_change_handle_register)

```
music_vol_change_handle_register(set_music_device_volume, phone_get_device_vol);
```

- **调用时机**：当连接的手机更改音乐播放音量时
- **用途**：实现手机与蓝牙设备之间的音量同步功能
- **注释信息**：头文件中定义为"手机更样机音乐模式的音量同步"
- 包含两个回调：
    - set_music_device_volume：接收手机音量变化并设置设备音量
    - phone_get_device_vol：获取当前设备音量值返回给手机

电量获取回调函数 (get_battery_value_register)

```
get_battery_value_register(bt_get_battery_value);
```

- **调用时机**：当连接的手机请求蓝牙设备电量信息时
- **用途**：向手机提供蓝牙设备的电量信息，用于在手机上显示

- **注释信息**：头文件中定义为"电量发送时获取电量等级的接口注册"
- 通过bt_get_battery_value函数返回电量数值，使手机能够显示蓝牙设备的电量状态

DUT测试模式处理函数 (bt_dut_test_handle_register)

```
bt_dut_test_handle_register(bt_dut_api);
```

- **调用时机**：当蓝牙设备进入DUT(Device Under Test)测试模式时
- **用途**：用于专业测试设备进行蓝牙性能和规范符合性测试
- **注释信息**：定义为"进入蓝牙DUT模式时的特定处理流程"
- 与快速测试不同，DUT模式通常用于更严格的实验室测试环境，测试设备与蓝牙标准的符合性

总结来说，这些回调函数是在SDK的蓝牙协议栈中预留的接口，允许应用层代码针对不同的蓝牙事件进行响应。bredr_handle_register函数的作用是将这些应用层回调函数注册到协议栈中，使得当特定事件发生时，协议栈能够调用这些函数来处理相应的业务逻辑。这种设计模式使协议栈和应用逻辑分离，提高了代码的模块化和可维护性。

蓝牙事件处理的实际位置

在 `earphone.c` 中，蓝牙事件的处理由以下几个关键函数完成：

- 主事件分发函数：`event_handler`
    - **系统事件处理函数中有有关蓝牙事件 `SYS_BT_EVENT` 处理的分支**

这是整个应用层的系统事件处理入口，负责将不同类型的事件（按键、蓝牙、设备等）分发到对应的处理函数。

- **文件位置**：`c:\Users\admin\Desktop\工作\FN3\apps\earphone\earphone.c`
- **行号范围**：L2446 - L2640
- 作用：
    - 处理按键事件（`SYS_KEY_EVENT`）
    - 处理蓝牙事件（`SYS_BT_EVENT`），包括连接状态变化和底层HCI事件
    - 处理设备事件（`SYS_DEVICE_EVENT`），如充电、电源、TWS、耳检测等
    - 处理红外传感器事件（`SYS_IR_EVENT`）
    - 处理PB消息事件（`SYS_PBG_EVENT`）

```
case SYS_BT_EVENT:
        /*
         * 蓝牙事件处理
         * 包含以下几种事件类型：
         * 1. 默认处理的事件：
         *    - SYS_BT_EVENT_TYPE_CON_STATUS: 蓝牙连接状态事件，处理连接、断开等状态变化
         *    - SYS_BT_EVENT_TYPE_HCI_STATUS: 蓝牙HCI协议事件，处理底层协议通信状态
         * 2. 宏控制的事件：
         *    - TCFG_ADSP_UART_ENABLE控制: UART相关命令事件
         *    - TCFG_USER_TWS_ENABLE控制: TWS真无线立体声事件
         */
        if ((u32)event->arg == SYS_BT_EVENT_TYPE_CON_STATUS) {
            /* 处理蓝牙连接状态事件，如配对、连接、断开等 */
            bt_connction_status_event_handler(&event->u.bt);
```

```
        } else if ((u32)event->arg == SYS_BT_EVENT_TYPE_HCI_STATUS) {
            /* 处理蓝牙HCI协议事件，如命令完成、状态改变等 */
            bt_hci_event_handler(&event->u.bt);
        }
#if TCFG_ADSP_UART_ENABLE
        else if (!strcmp(event->arg, "UART")) {
            /* 处理蓝牙UART命令事件，用于UART调试接口的命令处理 */
            extern void bt_uart_command_handler(struct uart_event * event);
            bt_uart_command_handler(&event->u.uart);
        } else if (!strcmp(event->arg, "UART_CMD")) {
            /* 处理ADSP（音频数字信号处理器）UART命令事件 */
            extern void adsp_uart_command_event_handle(struct uart_cmd_event *
uart_cmd);
            adsp_uart_command_event_handle(&event->u.uart_cmd);
        }
#endif
#if TCFG_USER_TWS_ENABLE
        else if (((u32)event->arg == SYS_BT_EVENT_FROM_TWS)) {
            /*
             * 处理TWS（真无线立体声）事件
             * 用于左右耳机之间的通信和同步，如主从切换、同步状态等
             */
            bt_tws_connction_status_event_handler(&event->u.bt);
        }
#endif
        break;
```

1. 默认情况下系统可以处理两种蓝牙事件：
    - 蓝牙连接状态事件(SYS_BT_EVENT_TYPE_CON_STATUS)
    - 蓝牙HCI协议事件(SYS_BT_EVENT_TYPE_HCI_STATUS)
2. 宏控制的事件有：
    - TCFG_ADSP_UART_ENABLE宏控制：处理"UART"和"UART_CMD"两种UART相关事件
    - TCFG_USER_TWS_ENABLE宏控制：处理SYS_BT_EVENT_FROM_TWS事件，用于真无线立体声功能

这些宏控制的功能是可选的，只有在相应宏定义开启时才会编译进代码。

疑问2

这个耳机应用下的状态机函数是轮询模型呢？还是事件驱动模型？比如一个事件发生，改变了it->action的状态，状态机才会执行一次对应分支？

这个`state_machine`函数明显是一个**事件驱动模型**，而非轮询模型。理由如下：

1. **函数参数结构**：
    - 函数接收`struct intent *it`参数，这是一个事件或意图对象
    - 代码中有明显的判断`if (!it) { break; }`，说明**这个函数是在事件触发时被调用的**
2. **执行流程**：
    - 状态机根据当前的`state`和`it->action`执行相应的处理分支
        - **在什么样的状态分支中的操作分支下**

        - 比如刚刚启动耳机应用的状态：
        - 在`APP_STA_START`状态的`ACTION_EARPHONE_MAIN`动作分支下
    - 每次函数被调用时处理一个特定状态和动作组合
    - 不是连续检查所有可能的状态（这才是轮询模型的特征）
3. **状态转换机制**：
    - 代码中使用`r_printf("bt_state_machine=%d\n", state);`打印当前状态，表明**状态是由外部传入的**
    - 在**不同状态和事件组合下执行不同的功能**，比如蓝牙初始化、音频解码开启等
4. **处理逻辑**：
    - 整个函数是被动响应外部调用，而不是主动轮询检查状态
    - 只有在特定事件（如ACTION_EARPHONE_MAIN、ACTION_A2DP_START等）发生时才执行对应代码块

工作流程

1. 当系统中发生一个事件（比如按键按下、蓝牙连接等）时，某处代码会创建一个`intent`对象，设置其`action`字段
2. 然后调用此`state_machine`函数，传入当前应用状态和该intent对象
3. 状态机根据当前状态和意图动作执行相应的处理逻辑
4. 处理完成后返回错误码（如有）

例如，当系统需要进入蓝牙模式时：

- 设置`state = APP_STA_START`
- 设置`it->action = ACTION_EARPHONE_MAIN`
- 调用`state_machine(app, state, it)`
    - ```
      init_intent(&it);
         it.name = "earphone";
         it.action = ACTION_EARPHONE_MAIN;
         start_app(&it);
      ```
- 状态机执行蓝牙初始化的一系列操作

这种模式下，状态机不会持续运行或轮询检查状态变化，而是在事件发生时被调用一次，响应特定的状态转换需求。

总结：这是一个典型的**事件驱动型状态机**，它响应外部事件并执行相应的状态转换操作，而不是通过连续轮询来检查状态变化。

蓝牙协议栈初始化

`btstack_init`启动蓝牙协议栈，并开始运行蓝牙任务。这会最终触发蓝牙初始化完成事件。

- **看不到源码**

`include_lib\btstack\btstack_task.h`

```
/**
 * @brief 蓝牙协议栈核心初始化
 * 该函数负责：
```
```

```
 * 1. 初始化蓝牙协议栈核心数据结构
 * 2. 创建蓝牙任务/线程处理协议栈事件
 * 3. 初始化HCI接口（UART/USB等物理接口）
 * 4. 注册协议栈事件回调函数
 * 5. 加载蓝牙配置参数（如MAC地址、安全数据库等）
 * @return 成功返回0，失败返回错误码
 */
int btstack_init();

/**
 * @brief 蓝牙协议栈资源释放
 * 该函数负责：
 * 1. 停止所有蓝牙协议栈线程/任务
 * 2. 释放动态分配的内存资源
 * 3. 关闭并释放HCI接口
 * 4. 清理协议栈状态机
 * 5. 保存必要的持久化配置数据
 * @return 成功返回0，失败返回错误码
 */
int btstack_exit();

/**
 * @brief BQB认证测试专用线程初始化
 * 该函数用于：
 * 1. 创建专门处理BQB认证测试的线程
 * 2. 初始化测试专用的GATT服务和特征值
 * 3. 注册测试模式下的事件处理回调
 * 4. 配置协议分析仪需要的调试接口
 * 5. 设置测试所需的特殊协议参数
 */
void ble_bqb_test_thread_init(void);
```

这些函数在蓝牙协议栈中的典型调用场景：

1. **btstack_init()** 在系统启动时调用，完成蓝牙协议栈的初始化
2. **ble_bqb_test_thread_init()** 仅在工程编译为BQB认证测试模式时调用，用于创建测试专用线程
3. **btstack_exit()** 在系统关机或蓝牙功能关闭时调用，确保资源正确释放

SPP相关初始化

```
bredr_handle_register();//注册SPP数据处理回调等函数
EARPHONE_STATE_INIT();//该函数完成蓝牙SPP连接的基础配置和SPP协议栈初始化
```

`bredr_handle_register`：

```
#if (USER_SUPPORT_PROFILE_SPP==1)
#if APP_ONLINE_DEBUG
    // 注册SPP数据处理回调函数（在线调试模式）
    // 使用自定义SPP数据处理函数user_spp_data_handler
    spp_data_deal_handle_register(user_spp_data_handler);
    // 在线SPP初始化
    online_spp_init();
#else
```

```
    // 注册标准SPP数据处理回调函数
    // 使用默认的spp_data_handler处理SPP数据
    spp_data_deal_handle_register(spp_data_handler);
#endif
#endif
......
```

`EARPHONE_STATE_INIT:`

- **源码不可见**

TWS相关初始化

通过宏定义使能`TCFG_USER_TWS_ENABLE`

```
            //TWS相关的参数配置
#if TCFG_USER_TWS_ENABLE
            tws_profile_init();//初始化TWS协议栈
            sys_key_event_filter_disable();//禁用按键过滤器，在TWS模式下防止主从耳冲突处理按键事件
#endif
```

tws_profile_init作用：

- **源码不可见**
- **协议栈准备**：建立TWS连接管理框架，初始化主/从角色状态机
- **资源分配**：预分配TWS数据同步所需的内存缓冲区
- **服务注册**：注册TWS专用的事件处理通道（如角色切换、状态同步）

sys_key_event_filter_disable作用：

- **源码不可见**
- **初始化阶段保护**：在TWS协议栈完全初始化完成前，禁止任何按键事件处理
    - **因为TWS配对后，应该是一个整体，任何按键事件应该同步响应。**
    - **一前一后或者一个响应一个不响应的话就会使两个耳机状态不一致，这个TWS模式下不希望看到的**
- **防止竞争条件**：避免主从耳在初始化过程中同时尝试接管按键事件
- **确保初始化顺序**：保证TWS主从状态完全建立后再启用按键过滤器
- **没有地方调用`sys_key_event_filter_enable`很奇怪，可能内部等到TWS协议栈初始化完成后自动调用了**

避免的问题：

1. **初始化冲突** - 若未禁用按键处理：
    - 可能出现从耳在初始化过程中收到按键事件，错误地触发配对/连接操作
    - 主耳可能在建立连接前就被要求处理按键事件，导致状态不同步
2. **资源竞争** - 没有初始化完成时：
    - 主从耳可能同时尝试访问共享硬件资源（如LED控制器）
    - TWS状态未明确时处理按键可能导致错误的角色切换
3. **协议异常** - 在初始化期间：

- 未完成角色协商就响应按键，可能引发不一致的电源管理模式
- 未建立连接前触发的按键事件可能导致错误的OTA数据传输

系统主事件处理函数

处理蓝牙连接状态事件 `bt_connction_status_event_handler`

蓝牙初始化完成事件处理

在调用`btstack_init()`后，蓝牙协议栈会在后台启动并执行初始化流程。当初始化完成时，最有可能是在蓝牙协议栈内部（即SDK库中）通过系统事件通知机制发送`BT_STATUS_INIT_OK`事件。

1. 虽然我们在开源代码中没有直接找到发送该事件的确切函数，但这很可能是在SDK的封闭源码部分（如`btstack.a`或`btctrler.a`库）中实现的。
2. 根据蓝牙协议栈的标准实现方式，通常在蓝牙控制器初始化成功后，协议栈会通过事件通知机制向应用层发送初始化完成事件。
3. 在您的SDK中，可能是通过`sys_event_notify`或类似机制发送这些事件，然后被主事件循环捕获并分发到`bt_connction_status_event_handler`函数处理。

总结来说，`btstack_init()`函数启动蓝牙协议栈，但不直接触发`BT_STATUS_INIT_OK`事件。该事件是由协议栈内部在完成所有初始化步骤后异步触发的，很可能是在SDK库的内部实现中（即我们看不到源码的部分）。

但是大差不差因为`btstack_init()`后面不涉及蓝牙相关代码

当蓝牙初始化完成后，会触发`BT_STATUS_INIT_OK`事件，在`bt_connction_status_event_handler`函数中处理：

- 这个函数是在`SYS_BT_EVENT`蓝牙相关消息处理分支下的函数

代码可以分为以下几大类操作：

1. 蓝牙功能初始化和配置
    1. 设置SBC编码参数`(__set_sbc_cap_bitpool(38))`
    2. 初始化蓝牙搜索索引`(bt_init_ok_search_index())`
    3. **BLE初始化`(bt_ble_init())`蓝牙连接控制策略设置**
2. 测试模式相关操作
    1. 定频测试接口（注释掉的代码）
    2. DUT测试模式设置
    3. BQB测试初始化
3. 硬件和外设初始化
    1. 初始化按键`(EARPHONE_CUSTOM_EARPHONE_KEY_INIT())`
    2. ANC主动降噪功能初始化`(anc_poweron())`
4. 通知和状态更新
    1. UI状态更新`(ui_update_status(STATUS_BT_INIT_OK))`
    2. 充电盒状态设置`(chargestore_set_bt_init_ok(1))`

5. 连接控制逻辑
    1. 根据不同工作模式设置连接策略
    2. TWS对耳连接初始化 `(bt_tws_poweron())`
    3. 手机连接控制 `(bt_wait_phone_connect_control(1))`

默认执行的操作 以下操作是无条件执行的（默认操作）：

- 按键初始化 (EARPHONE_CUSTOM_EARPHONE_KEY_INIT())
- SBC编码参数设置 (__set_sbc_cap_bitpool(38))
- 蓝牙搜索索引初始化 (bt_init_ok_search_index())
- UI状态更新 (ui_update_status(STATUS_BT_INIT_OK))

通过宏控制的操作 以下操作是通过宏控制的：

- 测试模式相关:
    - TCFG_NORMAL_SET_DUT_MODE: 控制DUT测试模式
    - TCFG_USER_BLE_ENABLE: 控制BLE功能初始化
    - CONFIG_BT_MODE == BT_BQB: BQB测试模式控制
- 音频功能:
    - TCFG_AUDIO_DATA_EXPORT_ENABLE: 音频数据导出功能
    - TCFG_AUDIO_ANC_ENABLE: ANC降噪功能
- 设备连接:
    - TCFG_USER_TWS_ENABLE: TWS对耳功能
    - TCFG_CHARGESTORE_ENABLE || TCFG_TEST_BOX_ENABLE: 充电盒功能
- 无线麦克风:
    - TCFG_WIRELESS_MIC_ENABLE: 控制无线麦克风功能
- 其他功能:
    - TCFG_ADSP_UART_ENABLE: ADSP UART功能
    - JL_EARPHONE_APP_EN && RCSP_UPDATE_EN: RCSP升级功能
    - TCFG_BLE_DEMO_SELECT: BLE演示功能选择
    - TCFG_BD_NUM: 蓝牙设备连接数量控制

蓝牙相关操作

与蓝牙直接相关的操作包括：

- 蓝牙协议栈配置:
    - SBC编码参数配置
    - 角色切换支持设置 (lmp_hci_set_role_switch_supported(0))
- 连接管理:
    - 蓝牙初始化搜索索引 (bt_init_ok_search_index())
    - 根据模式选择连接策略:
        - BQB/PER测试模式: bt_wait_phone_connect_control(1)
        - TWS模式: bt_tws_poweron()

        - 普通模式: bt_wait_connect_and_phone_connect_switch(0)
- BLE功能:
    - BLE初始化 (bt_ble_init())
    - BLE测试模式 (ble_bqb_test_thread_init(), ble_standard_dut_test_init())
    - BLE图标状态管理
- TWS对耳功能:
    - TWS电源管理 (bt_tws_poweron())
    - TWS连接状态检测 (get_bt_tws_connect_status())
    - TWS角色切换 (tws_conn_switch_role())

这段代码展示了一个典型的蓝牙耳机方案，在蓝牙协议栈初始化完成后，根据不同的配置和工作模式，执行相应的初始化和连接策略。代码通过大量的宏控制来适应不同的产品需求和功能配置。

```
case BT_STATUS_INIT_OK:
    // 蓝牙协议栈初始化完成
    log_info("BT_STATUS_INIT_OK\n");
#if 0
    printf("fix test rx if\n");  //定频测试用例接口
    fix_rxtx_lt_addr_set(5);
    bt_fix_txrx_api(1, fix_rx_mac_addr, 8, 5, 1);
    sys_timer_add(NULL, link_fix_rx_dump_result, 1000);
    return 0;
#endif
    // 初始化按键
    EARPHONE_CUSTOM_EARPHONE_KEY_INIT();
    /* bt_bredr_enter_dut_mode(0, 0); */
#if TCFG_NORMAL_SET_DUT_MODE
    log_info("edr set dut mode\n");
    bredr_set_dut_enble(1, 1);
    /* #if TCFG_USER_BLE_ENABLE */
    log_info("ble set dut mode\n");
    ble_standard_dut_test_init();
    /* #endif */
#endif
#if TCFG_AUDIO_DATA_EXPORT_ENABLE
#if (TCFG_AUDIO_DATA_EXPORT_ENABLE == AUDIO_DATA_EXPORT_USE_SPP)
    lmp_hci_set_role_switch_supported(0);
#endif/*AUDIO_DATA_EXPORT_USE_SPP*/
    extern int audio_capture_init();
    audio_capture_init();
#endif/*TCFG_AUDIO_DATA_EXPORT_ENABLE*/

#if TCFG_AUDIO_ANC_ENABLE
    anc_poweron();
#endif/*TCFG_AUDIO_ANC_ENABLE*/
    // 设置SBC编码参数
    __set_sbc_cap_bitpool(38);

#if (TCFG_USER_BLE_ENABLE)
    if (BT_MODE_IS(BT_BQB)) {
        ble_bqb_test_thread_init();
    } else {
```

```
#if !TCFG_WIRELESS_MIC_ENABLE
        // 初始化BLE
        bt_ble_init();
#endif
    }

#endif

#if TCFG_USER_TWS_ENABLE
    extern void pbg_demo_init(void);
    pbg_demo_init();
#endif

    // 初始化搜索索引
    bt_init_ok_search_index();
    ui_update_status(STATUS_BT_INIT_OK);
#if TCFG_CHARGESTORE_ENABLE || TCFG_TEST_BOX_ENABLE
    chargestore_set_bt_init_ok(1);
#endif


#if ((CONFIG_BT_MODE == BT_BQB)||(CONFIG_BT_MODE == BT_PER))
    // BQB测试模式或PER测试模式下，直接启用手机连接控制
    // 参数1表示开启可发现/可连接状态，允许设备被手机发现和连接
    bt_wait_phone_connect_control(1);
#else
    // 正常运行模式下的连接策略
    // 根据DAC（数模转换器）电源状态选择不同的连接方式
    if (is_dac_power_off()) {
        #if TCFG_USER_TWS_ENABLE
            // TWS双耳模式下：执行TWS专用的电源初始化流程
            // 该函数会处理双耳间的同步连接逻辑
            bt_tws_poweron();
        #else
            // 单耳模式：启动连接流程
            // 参数0表示主动连接记忆设备，发起蓝牙连接请求
            bt_wait_connect_and_phone_connect_switch(0);
        #endif
    } else {
        // DAC处于工作状态时：
        // 创建100ms后触发的定时器，用于延迟执行蓝牙连接
        // play_poweron_ok_timer是连接完成后的回调函数
        // 主要用于处理连接成功后的UI提示和状态更新
        app_var.wait_timer_do = sys_timeout_add(NULL, play_poweron_ok_timer,
100);
    }
#endif

    /*if (app_var.play_poweron_tone) {
        tone_play_index(p_tone->power_on, 1);
    }*/
    break;
    case BT_STATUS_SECOND_CONNECTED:
        // 清除当前开机记忆的设备搜索索引（针对第二个设备连接）
        clear_current_poweron_memory_search_index(0);
    case BT_STATUS_FIRST_CONNECTED:
```

```
        // 蓝牙设备连接成功通用处理流程
        // 包含首次连接（主设备）和第二个设备连接场景
        log_info("BT_STATUS_CONNECTED\n");

        // 切换电源模式到DCDC15（1.5V），3秒后生效
        earphone_change_pwr_mode(PWR_DCDC15, 3000);
        // 禁用自动关机功能（保持设备运行）
        sys_auto_shut_down_disable();

#if TCFG_ADSP_UART_ENABLE
        // 启用ADSP专用UART接口
        ADSP_UART_Init();
#endif

#if(JL_EARPHONE_APP_EN && RCSP_UPDATE_EN)
        // RCSP升级模式下的角色切换处理
        extern u8 rcsp_update_get_role_switch(void);
        if (rcsp_update_get_role_switch()) {
            // 执行TWS角色切换并禁用自动切换
            tws_conn_switch_role();
            tws_api_auto_role_switch_disable();
        }
#endif

#if(TCFG_BLE_DEMO_SELECT == DEF_BLE_DEMO_ADV)
        // BLE广播图标状态管理
        {
            u8 connet_type;
            // 根据是否为自动重连设置图标类型
            if (get_auto_connect_state(bt->args)) {
                connet_type = ICON_TYPE_RECONNECT; // 自动重连
            } else {
                connet_type = ICON_TYPE_CONNECTED; // 首次连接
            }

#if TCFG_USER_TWS_ENABLE
            // TWS模式下仅主耳显示BLE连接状态
            if (tws_api_get_role() == TWS_ROLE_MASTER) {
                bt_ble_icon_open(connet_type);
            } else {
                // 从耳可能已显示过，避免重复操作
                bt_ble_icon_close(0);
            }
#else
            // 单耳模式直接显示BLE连接状态
            bt_ble_icon_open(connet_type);
#endif
        }
#endif

#if (TCFG_BD_NUM == 2)
        // 双设备模式下管理手机连接状态
        if (get_current_poweron_memory_search_index(NULL) == 0) {
            // 当前设备为开机记忆的第一个设备，启用手机连接控制
            bt_wait_phone_connect_control(1);
        }
```

```
#endif

#if TCFG_USER_TWS_ENABLE
        // TWS模式下连接状态管理
        if (!get_bt_tws_connect_status()) {   // 对耳未连接时
            // 更新UI为蓝牙连接状态
            ui_update_status(STATUS_BT_CONN);
        }

#if TCFG_CHARGESTORE_ENABLE
        // 通知充电盒手机已连接
        chargestore_set_phone_connect();
#endif
        // 触发底层硬件状态机更新
        EARPHONE_STATE_BT_CONNECTED();

        // 如果对耳已连接则跳过后续处理
        if (bt_tws_phone_connected()) {
            break;
        }
#else
        // 单耳模式直接更新UI
        ui_update_status(STATUS_BT_CONN);

#if (TCFG_AUTO_STOP_PAGE_SCAN_TIME && TCFG_BD_NUM == 2)
        // 双设备模式下的页面扫描控制
        if (get_total_connect_dev() == 1) {   // 当前仅一个设备连接
            // 创建定时器停止页面扫描
            if (app_var.auto_stop_page_scan_timer == 0) {
                app_var.auto_stop_page_scan_timer = sys_timeout_add(NULL,
bt_close_page_scan, (TCFG_AUTO_STOP_PAGE_SCAN_TIME * 1000));
            }
        } else {
            // 多设备连接时删除定时器
            if (app_var.auto_stop_page_scan_timer) {
                sys_timeout_del(app_var.auto_stop_page_scan_timer);
                app_var.auto_stop_page_scan_timer = 0;
            }
        }
#endif   // AUTO_STOP_PAGE_SCAN_TIME
#endif

        // 处理连接提示音播放
        log_info("tone status:%d\n", tone_get_status());
        if (get_call_status() == BT_CALL_HANGUP) {
            // 如果当前无通话，处理延迟断开场景
            if (app_var.phone_dly_discon_time) {
                sys_timeout_del(app_var.phone_dly_discon_time);
                app_var.phone_dly_discon_time = 0;
            } else {
                // 1.6秒后播放连接提示音
                app_var.wait_timer_do = sys_timeout_add(NULL,
play_bt_connect_dly, 1600);
            }
        }
        break;
```

1. **按键初始化** `EARPHONE_CUSTOM_EARPHONE_KEY_INIT();`
    - 初始化耳机的物理按键功能，可能涉及按键事件注册、防抖设置等。
2. **DUT模式设置** `#if TCFG_NORMAL_SET_DUT_MODE`
    - 如果启用`TCFG_NORMAL_SET_DUT_MODE`，进入DUT（Device Under Test）模式：
        - `bredr_set_dut_enble(1, 1);`：启用BR/EDR模式的DUT测试功能。
        - `ble_standard_dut_test_init();`：初始化BLE标准DUT测试环境。
3. **音频数据导出功能** `#if TCFG_AUDIO_DATA_EXPORT_ENABLE`
    - 如果启用音频数据导出功能：
        - `lmp_hci_set_role_switch_supported(0)`：禁用角色切换（仅在SPP导出模式下需要）。
        - `audio_capture_init();`：初始化音频捕获模块，用于将音频数据通过SPP或其它接口导出。
4. **ANC（主动降噪）初始化** `#if TCFG_AUDIO_ANC_ENABLE`
    - 启动ANC模块电源：`anc_poweron();`。
5. **SBC编码参数设置** `__set_sbc_cap_bitpool(38);`
    - 设置SBC编码的位池参数为38，优化音频传输质量。
6. **BLE模块初始化** `#if TCFG_USER_BLE_ENABLE`
    - 如果启用BLE支持：
        - 在BQB测试模式下初始化BLE BQB测试线程。
        - 在普通模式下初始化BLE协议栈：`bt_ble_init();`。
7. **TWS（True Wireless Stereo）初始化** `#if TCFG_USER_TWS_ENABLE`
    - 初始化TWS功能：`pbg_demo_init();`，用于双耳同步连接。
8. **搜索索引与UI更新**
    - `bt_init_ok_search_index();`：初始化蓝牙设备搜索索引，用于快速回连历史设备。
    - `ui_update_status(STATUS_BT_INIT_OK);`：更新用户界面状态为“蓝牙初始化完成”。
    - `chargestore_set_bt_init_ok(1);`：通知充电仓模块蓝牙已初始化。
9. **连接模式决策**
    - 如果处于BQB或PER测试模式：`bt_wait_phone_connect_control(1);`开启可发现可连接状态。
    - 否则根据DAC电源状态决定连接策略：
        - DAC关闭时：进入TWS模式或直接回连设备。
        - DAC开启时：延迟播放启动提示音（通过`sys_timeout_add`调用`play_poweron_ok_timer`）。
    - 策略解析：
        1. **测试模式**（`BT_BQB`/`BT_PER`）：
            - 直接开启可发现/可连接状态，用于测试场景

2. **正常模式**：
    - **DAC电源关闭时**：
        - 双耳模式使用TWS专用连接流程
        - 单耳模式直接发起主动连接
    - **DAC电源工作时**：
        - 延迟100ms后执行连接操作，主要考虑：
            - 需等待DAC电源稳定
            - 给用户更流畅的连接体验（如播放开机提示音后再连接）
3. **策略优势**：
    - 智能选择连接时机，兼顾电源管理和用户体验
    - 支持单双耳模式自动适配
    - 测试模式与正常模式分离，保证测试准确性

   这种设计既满足了测试需求，又能根据不同硬件配置（是否支持TWS）和系统状态（DAC电源状态）智能选择最合适的连接方式。

10. **开机提示音（可选）**
    - 注释代码段显示可通过 `tone_play_index` 播放开机提示音，但实际功能被条件编译控制。

疑问

其实蓝牙初始化完成消息处理分支之前，耳机应用模式的状态机中，也有按键，蓝牙，tws，ble相关的内容，他们有什么关系？一个是定义参数？一个是利用参数做初始化？

`ACTION_EARPHONE_MAIN` **(先执行)**

这是基础参数配置和底层初始化阶段，主要完成：

- 系统时钟配置 (clk_set, bt_pll_para)
- 蓝牙功能模块选择 (bt_function_select_init)
- 蓝牙协议栈初始化和启动 (btstack_init)
- 蓝牙版本设置 (set_bt_version)
- 硬件层面的功能准备

这个阶段是启动蓝牙协议栈的过程，相当于给蓝牙系统"上电"并启动运行。

`BT_STATUS_INIT_OK` **(后执行)**

这是**功能逻辑初始化**阶段，当协议栈成功初始化后触发，主要完成：

- 基于已初始化的协议栈进行功能配置
- 连接策略设置
- 用户交互层面的功能准备
- 具体业务逻辑的启动

这个阶段是在协议栈准备就绪后，**根据参数配置各种功能模块**的过程。

关键组件的关系

按键处理

- ACTION_EARPHONE_MAIN: 调用sys_key_event_enable()启用按键消息系统
- BT_STATUS_INIT_OK: 调用EARPHONE_CUSTOM_EARPHONE_KEY_INIT()初始化具体按键功能

蓝牙核心功能

ACTION_EARPHONE_MAIN:

- bt_function_select_init(): 选择蓝牙功能模块
- btstack_init(): 启动蓝牙协议栈

BT_STATUS_INIT_OK:

- 配置连接策略
- 处理连接控制逻辑

TWS对耳功能

ACTION_EARPHONE_MAIN:

- tws_profile_init(): 初始化TWS配置文件

BT_STATUS_INIT_OK:

- bt_tws_poweron(): 执行TWS电源管理和连接逻辑

BLE功能

ACTION_EARPHONE_MAIN:

- 设置蓝牙频偏校准回调

BT_STATUS_INIT_OK:

- bt_ble_init(): 初始化BLE具体功能

工作流程总结

这两段代码实际上体现了一个**分层初始化**的设计理念：

1. **第一阶段**(`ACTION_EARPHONE_MAIN`):
    - 负责**底层参数定义和协议栈启动**
    - 建立基础硬件和协议支持
    - 相当于"安装驱动程序"
2. **第二阶段**(`BT_STATUS_INIT_OK`):
    - 利用第一阶段建立的基础进行**功能初始化和业务逻辑配置**
    - 根据参数配置实际功能
    - 相当于"配置应用程序"

这种分层设计使得蓝牙初始化过程更加模块化和可维护，同时也反映了蓝牙协议栈的运行机制 - 先启动核心协议栈，然后在其基础上构建具体的应用功能。

蓝牙初始化后设置了连接策略

在`BT_STATUS_INIT_OK`分支处理中设置了连接策略，这段代码决定了设备如何与手机或其他蓝牙设备建立连接。

```
#if ((CONFIG_BT_MODE == BT_BQB)||(CONFIG_BT_MODE == BT_PER))
    // BQB测试模式或PER测试模式下，直接启用手机连接控制
    // 参数1表示开启可发现/可连接状态，允许设备被手机发现和连接
    bt_wait_phone_connect_control(1);
#else
    // 正常运行模式下的连接策略
    // 根据DAC（数模转换器）电源状态选择不同的连接方式
    if (is_dac_power_off()) {
        #if TCFG_USER_TWS_ENABLE
            //状态机部分TWS协议栈已经初始化完成
            // TWS双耳模式下：执行TWS专用的电源初始化流程
            // 该函数会处理双耳间的同步连接逻辑
            bt_tws_poweron();
        #else
            // 单耳模式：启动连接流程
            // 参数0表示主动连接记忆设备，发起蓝牙连接请求
            bt_wait_connect_and_phone_connect_switch(0);
        #endif
    } else {
        // DAC处于工作状态时：
        // 创建100ms后触发的定时器，用于延迟执行蓝牙连接
        // play_poweron_ok_timer是连接完成后的回调函数
        // 主要用于处理连接成功后的UI提示和状态更新
        app_var.wait_timer_do = sys_timeout_add(NULL, play_poweron_ok_timer,
100);
    }
#endif
```

蓝牙连接策略主要包含以下几个路径：

1. **测试模式路径**：
    - BQB测试或PER测试模式下，直接启用`bt_wait_phone_connect_control(1)`，让设备处于可发现可连接状态
2. **正常模式路径**：
    - DAC电源关闭时：
        - TWS双耳模式：执行`bt_tws_poweron()`，处理双耳间的同步连接
        - 单耳模式：执行`bt_wait_connect_and_phone_connect_switch(0)`，主动连接记忆设备
    - DAC电源开启时：
        - 延迟100ms后执行连接，创建定时器调用`play_poweron_ok_timer`
        - 这个定时后的回调函数中依旧走的是上面的两个路径

在`bt_wait_connect_and_phone_connect_switch`函数是核心的蓝牙连接函数：

```
int bt_wait_connect_and_phone_connect_switch(void *p)
{
```

```
    int ret = 0;
    int timeout = 0;
    s32 wait_timeout;

    // 当自动连接计时器归零时，结束连接尝试流程
    if (bt_user_priv_var.auto_connection_counter <= 0) {
        bt_user_priv_var.auto_connection_timer = 0;
        bt_user_priv_var.auto_connection_counter = 0;

        // 取消蓝牙页面扫描（停止寻找新设备）
        EARPHONE_STATE_CANCEL_PAGE_SCAN();
        log_info("auto_connection_counter = 0\n");
        // 发送取消页面扫描命令
        user_send_cmd_prepare(USER_CTRL_PAGE_CANCEL, 0, NULL);

        // 如果存在开机记忆的设备索引
        if (get_current_poweron_memory_search_index(NULL)) {
            // 重新初始化设备搜索索引
            bt_init_ok_search_index();
            // 递归调用自身进行下一台设备连接尝试
            return bt_wait_connect_and_phone_connect_switch(0);
        } else {
            // 进入可发现/可连接状态，等待新设备连接
            bt_wait_phone_connect_control(1);
            return 0;
        }
    }

    // 根据参数p决定连接模式：
    // p=0: 主动连接记忆设备；p!=0: 被动等待新连接
    if ((int)p == 0) {
        if (bt_user_priv_var.auto_connection_counter) {
            // 主动连接模式：设置较长的超时时间
            timeout = 5600;
            // 关闭可发现/可连接状态（专注连接指定设备）
            bt_wait_phone_connect_control(0);
            // 发送通过地址连接指定设备的命令
            user_send_cmd_prepare(USER_CTRL_START_CONNEC_VIA_ADDR,
                                  6, bt_user_priv_var.auto_connection_addr);
            ret = 1; // 返回1表示已发起连接请求
        }
    } else {
        // 被动连接模式：设置较短的超时时间
        timeout = 400;
        // 取消当前页面扫描
        user_send_cmd_prepare(USER_CTRL_PAGE_CANCEL, 0, NULL);
        // 启用可发现/可连接状态（进入配对模式）
        bt_wait_phone_connect_control(1);
    }

    // 更新剩余自动连接时间
    if (bt_user_priv_var.auto_connection_counter) {
        bt_user_priv_var.auto_connection_counter -= timeout;
        log_info("do=%d\n", bt_user_priv_var.auto_connection_counter);
    }
```

```
    // 设置新的超时定时器，再次调用本函数形成循环
    // 参数说明：(!(int)p)用于区分模式（0:主动连接 1:被动等待）
    bt_user_priv_var.auto_connection_timer = sys_timeout_add(
        (void *)(!(int)p),
        bt_wait_connect_and_phone_connect_switch,
        timeout);

    return ret;
}
```

参数为0的执行流程

1.**初始进入函数**

- 当参数为0时，表示要主动连接记忆设备。

2.检查自动连接计数器

- 首先检查`auto_connection_counter`是否小于等于0
    - 如果是，表示已经没有回连时间了，进入超时处理
    - 如果否，继续执行主动连接流程

3.主动连接流程

- 因为参数p=0，进入主动连接分支
- 检查`auto_connection_counter`是否大于0
    - 如果是，设置超时时间为5600ms
    - 关闭可发现可连接状态：`bt_wait_phone_connect_control(0)`
    - 发送连接指定地址的命令：`user_send_cmd_prepare(USER_CTRL_S)`
    - 减少`auto_connection_counter`的值（减去5600）
    - 创建新的定时器，超时后以参数`(void *)(!(int)0)` = `(void *)1`再次调用本函数
    - 返回1（表示已发起连接）

4.超时后再次进入（参数为1）

- 当第3步的定时器超时，会以参数1再次调用本函数
- 因为参数p!=0，进入被动连接分支
- 设置超时时间为400ms
- 取消当前页面扫描
- 启用可发现可连接状态：`bt_wait_phone_connect_control(1)`
- 减少`auto_connection_counter`的值（减去400）
- 创建新的定时器，超时后以参数`(void *)(!(int)1)` = `(void *)0`再次调用本函数
- 返回0

5.超时后再次进入（参数为0）

- 重复第3步和第4步，**在主动连接和被动连接之间交替**
- 这个过程会不断减少`auto_connection_counter`的值

6.连接尝试结束处理

- 当`auto_connection_counter`减到小于等于0时：
    - 重置`auto_connection_timer = 0`和`auto_connection_counter = 0`
    - 取消页面扫描
    - 检查是否还有其他记忆设备可尝试连接：`get_current_poweron_memory_search_index(NULL)`
        - 如果有，重置搜索索引并再次调用`bt_wait_connect_and_phone_connect_switch(0)`尝试下一个设备
        - 如果没有，调用`bt_wait_phone_connect_control(1)`进入可发现可连接状态，并返回0

原理解释

这个函数的核心原理是：

1. **交替尝试策略**：在主动连接（寻找指定设备）和被动等待（允许被发现和连接）之间交替
2. 时间控制机制：通过`auto_connection_counter`控制总的尝试时间
3. **递归调用链**：函数通过定时器和递归调用自身形成循环，直到连接成功或尝试时间耗尽
4. **多设备尝试**：如果一个设备连接失败，会尝试连接下一个记忆设备

函数是否会结束?

函数确实会结束，但通过定时器和递归调用形成了一个"循环"机制：

1. 单次调用会结束，但会创建定时器再次调用自身
2. 当`auto_connection_counter`耗尽且没有更多记忆设备时，函数才会真正"结束循环"
3. **如果连接成功，蓝牙协议栈会触发连接事件，可能会在事件处理函数中取消这个定时器**

所以从用户角度看，函数会持续工作直到：

- 成功连接到某个设备
- 尝试了所有记忆设备后进入可发现可连接状态
- 被其他事件中断（如手动取消、关机等）

这种设计确保了设备能够尽可能地恢复之前的连接，同时又不会无限期地尝试连接不可用的设备。

非TWS模式蓝牙耳机开机的蓝牙连接流程

1. **初始调用**
    - 蓝牙初始化完成后，调用`bt_wait_connect_and_phone_connect_switch(0)`开始回连
    - 参数0表示主动尝试连接记忆设备
2. **回连时间机制**
    - 系统使用`auto_connection_counter`变量控制总回连时间
    - 这是**所有记忆设备共享的总时间**，而非每台设备单独计时
    - 每次尝试连接会减少这个计数器：主动连接减5600ms，被动等待减400ms
3. **每台记忆设备的回连尝试**
    - 系统会使用记忆设备索引（通过`get_current_poweron_memory_search_index`获取）

        - 对每个记忆设备进行尝试连接
        - 没有明确设置每台设备的尝试次数，而是由总时间控制
            - 每一个主动连接的时间呗5600
        - 一个设备连接失败后，会尝试下一个记忆设备（如果总时间允许）
    4. **回连策略**
        - 采用交替策略：主动连接(5600ms) → 被动等待(400ms) → 主动连接 → ...
        - **主动连接时关闭可发现可连接状态，专注连接指定设备**
            - **回连过程中，主要是主动连接但是也可以被其他设备连接**
        - **被动等待时开启可发现可连接状态，短暂允许被连接**
    5. **回连不成功后的处理**
        - 当所有记忆设备都尝试连接失败，或总时间耗尽：
        - 系统会调用`bt_wait_phone_connect_control(1)`
        - 这会将耳机设置为**可发现可连接状态**
        - 此时耳机可以被新的设备发现并配对
    6. **回连流程结束**
        - 回连成功：蓝牙事件会处理连接成功，停止后续尝试
        - 回连失败：尝试所有记忆设备后，进入可发现可连接状态
        - 程序不会"死循环"尝试连接，会有明确的结束条件

要点总结

- 回连时间是所有设备共享的，不是每台设备单独计时
- 系统会尝试连接所有记忆设备，直到连接成功或时间耗尽
- 最终，如果回连不成功，耳机会进入可发现可连接状态，等待新连接
- 这种设计既保证了尝试恢复已有连接，又不会无限期阻止新连接的建立

这种机制确保了耳机能优先尝试连接用户熟悉的设备，同时在必要时也能与新设备配对，提供了良好的用户体验。

TWS模式的流程

DAC电源关闭时：

- TWS双耳模式：执行`bt_tws_poweron()`，处理双耳间的同步连接

```
/*
 * 开机tws初始化
 */
/*
 * TWS(真无线立体声)设备初始化连接函数
 * 该函数在蓝牙协议栈初始化完成后调用，负责处理TWS设备的连接策略
 * 主要功能：
 * 1. 处理DUT测试模式特殊场景
 * 2. 初始化TWS连接状态
 * 3. 处理已配对设备的连接流程
 * 4. 实现未配对设备的配对策略
 * 5. 配置音频传输通道参数
 */
```

```
u8 s_ucLowlatency_mode = 0;//added by samson 在开机的时候，设置低延迟不发送通知；
__BANK_INIT_ENTRY
int bt_tws_poweron()
{
    int err;
    u8 addr[6];
    char channel;

    // 如果处于充电盒DUT测试模式，启用可发现/可连接状态并返回
    if (chargestore_get_ex_enter_dut_flag()) {
        log_info("tws poweron enter dut case\n");
        printf("%s:%d-----samson--------------\n", __func__, __LINE__);
        user_send_cmd_prepare(USER_CTRL_WRITE_SCAN_ENABLE, 0, NULL);
        user_send_cmd_prepare(USER_CTRL_WRITE_CONN_ENABLE, 0, NULL);
        return 0;
    }
   printf("%s:%d-----samson--------------\n", __func__, __LINE__);
#if TCFG_AUDIO_ANC_ENABLE
    /*支持ANC训练快速连接，不连接tws*/
#if TCFG_ANC_BOX_ENABLE
    if (ancbox_get_status())
#else
    if (0)
#endif/*TCFG_ANC_BOX_ENABLE*/
    {
        printf("%s:%d-----samson--------------\n", __func__, __LINE__);
        EARPHONE_STATE_SET_PAGE_SCAN_ENABLE();
        user_send_cmd_prepare(USER_CTRL_WRITE_SCAN_ENABLE, 0, NULL);
        user_send_cmd_prepare(USER_CTRL_WRITE_CONN_ENABLE, 0, NULL);
        return 0;
    } else {
        /*tws配对前，先关闭可发现可连接，不影响tws配对状态*/
        //user_send_cmd_prepare(USER_CTRL_WRITE_SCAN_DISABLE, 0, NULL);
        //user_send_cmd_prepare(USER_CTRL_WRITE_CONN_DISABLE, 0, NULL);
    }
#endif/*TCFG_AUDIO_ANC_ENABLE*/

    gtws.tws_dly_discon_time = 0;// 清除延迟断开定时器
    gtws.state = BT_TWS_POWER_ON;// 设置当前状态为电源开启
    gtws.connect_time = 0;// 重置连接时间

    // 根据配置设置TWS配对模式
#if CONFIG_TWS_PAIR_BY_BOTH_SIDES
    bt_set_pair_code_en(0);// 禁用配对码功能
#endif
    lmp_hci_write_page_timeout(12000);// 设置页面超时时间为12秒（TWS连接超时阈值）

    // 根据A2DP游戏模式配置自动角色切换功能
#if A2DP_PLAY_AUTO_ROLE_SWITCH_ENABLE
    tws_api_auto_role_switch_enable();// 启用自动主从角色切换
#else
    tws_api_auto_role_switch_disable();// 禁用自动主从角色切换
#endif

    // 根据ESCO电源平衡配置启用相应功能
#if ESCO_AUTO_POWER_BALANCE_ROLE_SWITCH_ENABLE
```

```
    extern void tws_api_esco_power_balance_role_switch(u8 enable);

 tws_api_esco_power_balance_role_switch(ESCO_AUTO_POWER_BALANCE_ROLE_SWITCH_ENABL
E);
#endif

    int result = 0;
    // 尝试获取已存储的对耳设备地址
    err = tws_get_sibling_addr(addr, &result);
    if (err == 0) {
        /*
         * 获取到对耳地址，表示设备已配对过
         * 以下步骤处理已配对设备的连接流程
         */
        printf("\n ---------have tws info----------\n");
        gtws.state |= BT_TWS_PAIRED;// 设置已配对状态标志

        EARPHONE_STATE_TWS_INIT(1);// 通知底层已进入TWS初始化阶段

        // 设置对耳设备地址
        tws_api_set_sibling_addr(addr);

        // 根据硬件配置选择音频传输通道
        if (set_channel_by_code_or_res() == 0) {
            channel = bt_tws_get_local_channel();
            tws_api_set_local_channel(channel);// 设置本地通道标识
        }

        // 特殊情况处理：测试盒模式跳过连接过程
#if TCFG_TEST_BOX_ENABLE
        if (chargestore_get_testbox_status()) {
            // 测试盒模式特殊处理
        } else
#endif
        {
#ifdef CONFIG_NEW_BREDR_ENABLE
            /* if (bt_tws_get_local_channel() == 'L') { */
            /* syscfg_read(CFG_TWS_LOCAL_ADDR, addr, 6); */
            /* } */
            // 新版EDR配置：生成快速连接地址
            u8 local_addr[6];
            syscfg_read(CFG_TWS_LOCAL_ADDR, local_addr, 6);
            for (int i = 0; i < 6; i++) {
                addr[i] += local_addr[i];// 组合生成快速连接地址
            }
            tws_api_set_quick_connect_addr(addr);
#endif
            // 开始连接对耳设备，设置连接超时时间
            bt_tws_connect_sibling(CONFIG_TWS_CONNECT_SIBLING_TIMEOUT);
        }

    } else {
        // 未找到对耳地址，表示设备未配对
        printf("\n ---------no tws info----------\n");

        EARPHONE_STATE_TWS_INIT(0);// 通知底层未进入TWS初始化阶段
```

```
        // no_tws for test
        //EARPHONE_STATE_TWS_INIT(1);
        //EARPHONE_STATE_SET_PAGE_SCAN_ENABLE();

        gtws.state |= BT_TWS_UNPAIRED;

        // 根据配置设置本地通道标识
        if (set_channel_by_code_or_res() == 0) {
            tws_api_set_local_channel('U');// 设置为未指定通道
        }
#if TCFG_TEST_BOX_ENABLE
        if (chargestore_get_testbox_status()) {
            // 测试盒模式特殊处理
        } else
#endif
        {

#if CONFIG_TWS_PAIR_MODE == CONFIG_TWS_PAIR_BY_AUTO
            /*
             * 自动配对模式:
             * 1. 生成自动配对码
             * 2. 开始连接对耳设备
             */
#ifdef CONFIG_NEW_BREDR_ENABLE
            tws_api_set_quick_connect_addr(tws_set_auto_pair_code());// 设置自动生
成的配对码
#else
            tws_auto_pair_enable = 1;// 启用自动配对功能
            tws_le_acc_generation_init();// 初始化LE地址生成
#endif
            bt_tws_connect_sibling(6);// 开始连接对耳设备

#else

// 其他配对模式处理（如按键配对）
#if CONFIG_TWS_PAIR_MODE == CONFIG_TWS_PAIR_BY_CLICK
            if (result == VM_INDEX_ERR) {
                printf("tws_get_sibling_addr index_err and
bt_tws_start_search_sibling =%d\n", result);
                // 存储索引错误时启动搜索
                tws_search_sibling();
            } else
#endif
            {
                /*
                 * 未配对状态下的连接策略:
                 * 1. 切换连接和可发现状态
                 * 2. 启动连接流程
                 */
                bt_tws_connect_and_connectable_switch();
                /*tws_api_create_connection(0);*/
                /*bt_tws_search_sibling_and_pair();*/

            }
#endif
```

```
        }
    }

#ifndef CONFIG_NEW_BREDR_ENABLE
    tws_remote_state_clear();// 清除远程设备状态信息
#endif

//added by samson on 20250502 这个在上面会截断开机提示音，放到这里等开机提示音完成后再进行
// 延迟执行低延迟模式设置（等待开机提示音播放完成）
    #ifdef CONFIG_A2DP_GAME_MODE_ENABLE
    extern void bt_set_low_latency_mode(int enable);
     os_time_dly(100); //延迟是等开机提示音结束，延时100ms确保开机提示音播放完成
    /* tws_api_auto_role_switch_disable(); */
     bt_set_low_latency_mode(0);//开机默认低延时模式设置 added by samson on
20240426，默认关闭低延迟模式
#endif
    return 0;
}
```

TWS连接策略函数干了什么？具体什么样的连接策略？

- 耳机有配对记录，就会直接tws连接
- 没有配对记录的话，根据配对模式进入不同的连接策略
    - 自动配对
    - 手动配对
- 未配对状态下的连接策略
    - 定时切换切换连接和可发现状态

耳机与手机蓝牙连接成功

非TWS经典蓝牙连接成功路径

之前的流程

蓝牙协议栈初始化完成后，进入`BT_STATUS_INIT_OK`非TWS模式会调用`bt_wait_connect_and_phone_connect_switch(0);`进入记忆设备回连操作。假如回连设备成功的话，会进入`BT_STATUS_SECOND_CONNECTED`或者`case BT_STATUS_FIRST_CONNECTED:`分支吗？两者有什么区别？

双设备模式概念，如果不开启`TCFG_BD_NUM`双设备模式的话，耳机回连成功手机蓝牙根本不会走`BT_STATUS_SECOND_CONNECTED`分支。

单设备模式下蓝牙连接成功后做了什么通用操作？哪一些是使用宏控制的？

通用操作（无条件执行）：

- **电源模式切换**
    - 调用`earphone_change_pwr_mode(PWR_DCDC15, 3000)`将电源模式切换为DCDC15（1.5V），3秒后生效，以降低功耗。
- **禁用自动关机**
    - 调用`sys_auto_shut_down_disable()`禁用自动关机功能，确保设备持续运行。
- **UI状态更新**

    - 调用 `ui_update_status(STATUS_BT_CONN)` 更新用户界面状态为蓝牙连接成功。
    - **这个状态更新后不知道会不会触发灯效更新？？？**
- **延迟断开与提示音播放**
    - 如果当前无通话且未启用延迟断开，则启动定时器：
    - ```
      app_var.wait_timer_do = sys_timeout_add(NULL,
      play_bt_connect_dly, 1600);
      ```
    - 1.6秒后播放连接提示音（`play_bt_connect_dly` 回调触发）。

宏管理操作：

- **BLE图标控制**（宏 `TCFG_BLE_DEMO_SELECT`）
    - **作用**：区分自动重连和首次连接，设置对应的BLE图标。
- **充电盒通知**（宏 `TCFG_CHARGESTORE_ENABLE`）
    - **作用**：充电盒状态同步，如指示灯或电量上报。
- **双设备模式检查**（宏 `TCFG_BD_NUM == 2`）
    - **单设备模式下不执行**：该宏仅在双设备模式下启用。

处理TWS事件 `bt_tws_connction_status_event_handler`

TWS连接成功路径

之前流程

进入 `bt_tws_connction_status_event_handler` 的 `TWS_EVENT_CONNECTED` 分支，**TWS连接成功的话，会连接手机吗？具体在什么位置？调用了什么函数？**

当TWS连接成功后，系统会执行以下核心操作，重点关注与手机蓝牙连接相关的处理：

配对信息同步

```
// 更新对耳设备地址和公共地址到配置存储
if (memcmp(addr[0], addr[2], 6)) {
    syscfg_write(CFG_TWS_REMOTE_ADDR, addr[2], 6);
}
if (memcmp(addr[1], addr[3], 6)) {
    syscfg_write(CFG_TWS_COMMON_ADDR, addr[3], 6);
}
```

- 将对耳设备地址和公共地址写入配置存储，确保后续连接时可快速识别
- 触发 `bt_update_mac_addr` 更新本地蓝牙MAC地址

主从角色处理

```
// 主耳处理手机连接逻辑
if (role == TWS_ROLE_MASTER) {
    if (!phone_link_connection) {
        // 同步LED状态
        bt_tws_led_state_sync(SYNC_LED_STA_TWS_CONN, 400);
    } else {
        bt_tws_led_state_sync(SYNC_LED_STA_PHONE_CONN, 400);
```
```

```
    }

    // 同步播放连接提示音
    if (!get_bt_tws_discon_dly_state() &&
        (get_call_status() == BT_CALL_HANGUP) &&
        !(state & TWS_STA_SBC_OPEN)) {
        bt_tws_play_tone_at_same_time(SYNC_TONE_TWS_CONNECTED, 400);
    }
}
```

- 主耳负责协调手机连接状态
- 根据手机连接状态同步LED显示和提示音

手机连接触发机制

```
// 没有手机连接的情况下启动连接流程
if (phone_link_connection == 0) {
    if (role == TWS_ROLE_MASTER) {
        bt_tws_connect_and_connectable_switch();  // 触发连接手机流程
    }
    sys_auto_shut_down_disable();
    sys_auto_shut_down_enable();
}
```

- 主耳检测到无手机连接时：
    1. 调用`bt_tws_connect_and_connectable_switch`切换连接模式
    2. 重置自动关机定时器
    3. 进入可连接状态搜索手机

连接状态同步

```
// 通知应用层连接状态变化
DhfAppCommand_DataNotify(EM_COMM_CONN_STATUS_CHANGE_DEVICE_SEND);
```

- 通过事件总线广播连接状态变更
- 触发上层应用逻辑处理连接状态更新

关键函数分析

```
static void bt_tws_connect_and_connectable_switch()
{
    bt_tws_delete_pair_timer();
    printf("%s:%d-----samson--------------\n", __func__, __LINE__);
#if TCFG_AUDIO_ANC_ENABLE
#if TCFG_ANC_BOX_ENABLE
    if (ancbox_get_status()) {
        printf("%s:%d-----samson--------------\n", __func__, __LINE__);
        EARPHONE_STATE_SET_PAGE_SCAN_ENABLE();
        user_send_cmd_prepare(USER_CTRL_WRITE_SCAN_ENABLE, 0, NULL);
        user_send_cmd_prepare(USER_CTRL_WRITE_CONN_ENABLE, 0, NULL);
        return;
    }
#endif
```

```
#endif

    if (tws_api_get_role() == TWS_ROLE_MASTER) {
        connect_and_connectable_switch(0);
    }
}
```

定时器管理

```
// 清除延迟断开定时器
if (gtws.tws_dly_discon_time) {
    sys_timeout_del(gtws.tws_dly_discon_time);
    gtws.tws_dly_discon_time = 0;
}
```

- 清除TWS连接延迟断开定时器
- 防止连接成功后误触发断开操作

电源管理

```
// 调整电源模式
pwm_led_clk_set(PWM_LED_CLK_BTOSC_24M);
```

- 切换LED时钟源到蓝牙高频时钟
- 确保连接状态下的稳定供电

功能模块同步

```
// 同步功能模块状态
chargestore_sync_chg_level();  // 同步充电舱电量
bt_tws_sync_volume();          // 同步音量
```

- 保持双耳间关键状态同步
- 为后续手机连接做准备

连接保持机制

```
// 管理自动关机定时器
sys_auto_shut_down_disable();
sys_auto_shut_down_enable();
```

状态标志更新

```
// 更新状态标志位
gtws.state &= ~BT_TWS_DISCON_DLY_TIMEOUT;
gtws.state &= ~BT_TWS_DISCON_DLY_TIMEOUT_NO_CONN;
```

- 清除所有延迟断开标志
- 确保进入稳定连接状态

总结手机连接相关处理流程：

1. 主耳检测手机连接状态
2. 若未连接则触发连接流程：
    - 进入可连接模式
    - 启动自动回连定时器
    - 设置页面扫描参数
3. 同步连接状态到从耳设备
4. 维持连接状态下的电源管理和定时器控制

这些操作确保在TWS连接成功后，设备能正确处理与手机的连接状态，保持双耳同步并提供良好的用户体验。

TWS连接成功后，手机蓝牙未连接时的操作

```
static void bt_tws_connect_and_connectable_switch()
{
    // 删除现有的配对定时器，防止重复触发配对流程
    bt_tws_delete_pair_timer();

    // 调试日志：打印函数名和行号，用于跟踪代码执行
    printf("%s:%d-----samson--------------\n", __func__, __LINE__);

#if TCFG_AUDIO_ANC_ENABLE
#if TCFG_ANC_BOX_ENABLE
    // 如果启用ANC（主动降噪）功能且处于ANC盒子模式
    if (ancbox_get_status()) {
        // 调试日志：标记进入ANC模式的特殊情况处理
        printf("%s:%d-----samson--------------\n", __func__, __LINE__);

        // 启用页面扫描（使设备可被发现）
        EARPHONE_STATE_SET_PAGE_SCAN_ENABLE();

        // 发送命令：开启可发现模式
        user_send_cmd_prepare(USER_CTRL_WRITE_SCAN_ENABLE, 0, NULL);

        // 发送命令：开启可连接模式
        user_send_cmd_prepare(USER_CTRL_WRITE_CONN_ENABLE, 0, NULL);

        // ANC盒子模式下直接返回，不执行后续逻辑
        return;
    }
#endif
#endif

    // 仅主耳设备执行以下连接流程
    if (tws_api_get_role() == TWS_ROLE_MASTER) {
        // 进入连接状态机切换
        // 参数0表示使用默认的连接策略
        connect_and_connectable_switch(0);
    }
}
```

1. **定时器管理**
   `bt_tws_delete_pair_timer()` 用于清除之前的配对定时器，避免在TWS连接成功后重复触发配对逻辑。
2. **调试信息**
   `printf` 语句用于记录代码执行路径，帮助调试时确认函数调用栈。
3. **ANC特殊处理**
   - 当启用ANC功能且设备处于ANC盒子模式时，强制设备保持可发现/可连接状态。
   - 这种设计确保在ANC调试模式下，手机可以随时连接设备。
4. **主从角色区分**
   - 只有主耳设备（`TWS_ROLE_MASTER`）执行连接手机的逻辑
   - 从耳设备直接忽略，避免双耳同时尝试连接手机导致冲突
     - TWS连接后，视为同一个设备，主耳进行外部连接操作并同步给从耳
5. **连接状态机切换**
   `connect_and_connectable_switch(0)` 函数执行以下操作：
   - 进入可连接状态，准备与手机配对
   - 根据参数选择不同的连接策略（0表示标准连接模式）
   - 包含自动回连记忆设备、搜索新设备等子状态

connect_and_connectable_switch函数

```
/*
 * TWS连接后的手机连接状态机切换函数
 * 管理回连手机、可发现/可连接模式、TWS连接保持等核心状态切换
 * TWS 回连手机、开可发现可连接、回连已配对设备、搜索tws设备4个状态间定时切换函数
 */
static void connect_and_connectable_switch(void *_sw)
{
    // 当前状态标记（0-回连手机，1-等待配对，2-特殊模式）
    int timeout = 0;
    int sw = (int)_sw;
    int end_sw = 1;

    // 清除旧的配对定时器（确保单次定时器触发）
    gtws.pair_timer = 0;

    // 调试日志：显示当前状态机状态、TWS状态、剩余自动连接时间
    log_info("switch: %d, state = %x, %d\n", sw, gtws.state,
             bt_user_priv_var.auto_connection_counter);

    if (sw == 0) {  // 状态0：回连手机模式
__again:
        if (bt_user_priv_var.auto_connection_counter > 0) {  // 有剩余自动连接时间
            // 已连接TWS设备时使用较长超时
            if (gtws.state & BT_TWS_SIBLING_CONNECTED) {
                timeout = 8000;  // 8秒超时
            } else {  // 未连接TWS时使用随机超时
#ifdef CONFIG_TWS_BY_CLICK_PAIR_WITHOUT_PAIR
                timeout = 2000 + (rand32() % 4 + 1) * 500;  // 2-4秒随机超时
#else
```

```
                timeout = 4000 + (rand32() % 4 + 1) * 500;  // 4-6秒随机超时
#endif
            }
            // 更新剩余自动连接时间
            bt_user_priv_var.auto_connection_counter -= timeout ;

            // 取消之前的配对/连接操作
            tws_api_cancle_wait_pair();
            tws_api_cancle_create_connection();

            // 发起指定地址的手机连接请求
            user_send_cmd_prepare(USER_CTRL_START_CONNEC_VIA_ADDR, 6,
                                  bt_user_priv_var.auto_connection_addr);
        } else {  // 自动连接时间耗尽
            bt_user_priv_var.auto_connection_counter = 0;

            // 关闭页面扫描（停止主动搜索设备）
            EARPHONE_STATE_CANCEL_PAGE_SCAN();

            // 开机回连逻辑
            if (gtws.state & BT_TWS_POWER_ON) {
                // 获取下一个记忆设备地址
                if (get_current_poweron_memory_search_index(NULL)) {
                    bt_init_ok_search_index();  // 初始化记忆设备搜索
                    goto __again;  // 递归尝试连接下一个设备
                }
                gtws.state &= ~BT_TWS_POWER_ON;  // 清除开机标志
            }

            // TWS已连接时的处理
            if (gtws.state & BT_TWS_SIBLING_CONNECTED) {
                // 进入可配对状态等待手机连接
                tws_api_wait_pair_by_code(bt_get_tws_device_indicate(NULL),
                                          bt_get_local_name(), 0);

                // 启动SNIFF低功耗模式检测
                tws_sniff_controle_check_enable();
                return;
            }

            // 按键配对模式下的未配对状态处理
#if CONFIG_TWS_PAIR_MODE == CONFIG_TWS_PAIR_BY_CLICK
            if (gtws.state & BT_TWS_UNPAIRED) {
                // 进入可配对状态等待用户配对
                tws_api_wait_pair_by_code(bt_get_tws_device_indicate(NULL),
                                          bt_get_local_name(), 0);
                return;
            }
#endif
            sw = 1;  // 切换到状态1
        }
    }

    if (sw == 1) {  // 状态1：等待配对模式
        // 进入可配对状态，允许手机发现并连接
        tws_api_wait_pair_by_code(bt_get_tws_device_indicate(NULL),
```

```
                                bt_get_local_name(), 0);

        // 仅当未连接TWS设备时继续执行
        if (!(gtws.state & BT_TWS_SIBLING_CONNECTED)) {
            // 非未配对状态时的特殊处理
            if (!(gtws.state & BT_TWS_UNPAIRED)) {
#ifdef CONFIG_TWS_AUTO_PAIR_WITHOUT_UNPAIR
                end_sw = 2;  // 允许进入状态2的自动配对模式
#endif
            } else {
#if CONFIG_TWS_PAIR_MODE == CONFIG_TWS_PAIR_BY_CLICK
                goto __set_time;  // 跳过创建连接直接设置定时
#endif
            }
            // 创建新的TWS连接尝试
            tws_api_create_connection(0);
        }
__set_time:
        // 设置随机超时时间（连接状态保持或模式切换）
#ifdef CONFIG_TWS_BY_CLICK_PAIR_WITHOUT_PAIR
        timeout = 500 + (rand32() % 4 + 1) * 500;  // 0.5-2.5秒随机超时
#else
        timeout = 2000 + (rand32() % 4 + 1) * 500;  // 2-4秒随机超时
#endif

        // 启用页面扫描（使设备可被发现）
        EARPHONE_STATE_SET_PAGE_SCAN_ENABLE();

        // 特殊退出条件判断
        if (bt_user_priv_var.auto_connection_counter <= 0 && end_sw == 1) {
            if (gtws.state & BT_TWS_SIBLING_CONNECTED) {
                // 已连接TWS时启用SNIFF模式
                tws_sniff_controle_check_enable();
            }
            return;  // 直接返回不重启定时器
        }

    } else if (sw == 2) {  // 状态2：特殊配对模式（需配置启用）
        u8 addr[6];

        // 配置特殊配对模式的超时时间
#ifdef CONFIG_TWS_BY_CLICK_PAIR_WITHOUT_PAIR
        timeout = 500 + (rand32() % 4 + 1) * 500;
        // 维持可配对状态
        tws_api_wait_pair_by_code(bt_get_tws_device_indicate(NULL),
                                  bt_get_local_name(), 0);
#else
        timeout = 2000 + (rand32() % 4 + 1) * 500;

        // 快速连接模式：
        memcpy(addr, tws_api_get_quick_connect_addr(), 6);  // 保存当前地址
        // 设置自动生成的配对码
        tws_api_set_quick_connect_addr(tws_set_auto_pair_code());
        // 发起TWS连接请求
        tws_api_create_connection(0);
        // 恢复原始地址（不影响后续连接）
```

```
        tws_api_set_quick_connect_addr(addr);
#endif
    }

    // 状态递增并循环（状态0->1->2->0...）
    if (++sw > end_sw) {
        sw = 0;
    }

    // 设置新的定时器，超时后继续切换状态
    gtws.pair_timer = sys_timeout_add((void *)sw, connect_and_connectable_switch,
timeout);
}
```

1. **状态机设计**：
    - 使用`sw`参数维护状态流转（0→1→2→0循环）
    - 不同状态对应不同连接策略
2. **连接策略**：
    - 状态0：优先回连记忆手机
    - 状态1：进入可配对模式等待连接
    - 状态2：特殊配对逻辑（需配置启用）
3. **超时机制**：
    - 使用随机超时防止连接冲突
    - 超时时间随状态变化（4-6秒 → 2-4秒 → 0.5-2.5秒）
4. **TWS状态协同**：
    - 保持TWS连接时启用SNIFF低功耗模式
    - 自动切换连接模式以保持双耳同步
5. **定时器驱动**：
    - 通过`sys_timeout_add`实现持续状态轮询
    - 每次超时后自动切换到下一个状态

该函数实现了TWS连接成功后持续尝试连接手机的智能状态切换机制，通过多状态轮询确保设备在不同场景下都能以最优方式保持连接能力。

断开连接

手机的主动连接请求

当手机发起连接请求时，会触发HCI事件，在`bt_hci_event_handler`函数中处理：

```
int bt_hci_event_handler(struct bt_event *bt)
{
    switch (bt->event) {
    case HCI_EVENT_CONNECTION_REQUEST:
        // 接受连接请求
        user_send_cmd_prepare(USER_CTRL_ACCEPT_CONN, 1, bt->args);
        break;

    case HCI_EVENT_CONNECTION_COMPLETE:
        // 连接完成事件处理
        bt_hci_event_connection(bt);
        break;

    // ...其他事件处理
    }

    // 处理连接状态事件
    bt_connction_status_event_handler(bt);

    return 0;
}
```

bt_hci_event_handler 函数是蓝牙协议栈的底层事件处理函数，主要负责处理蓝牙主机控制器接口（HCI）触发的各种事件，其核心功能包括：

厂商自定义事件处理

- 远程测试事件
    - HCI_EVENT_VENDOR_REMOTE_TEST:
        - 当设备因远端测试断开连接时，清除测试状态并复位系统（用于产测场景）。
        - 在经典蓝牙模式下连接时，禁用BLE广播；非TWS连接时关闭TWS功能。

标准HCI事件处理

通过`switch`分支处理以下核心事件：

- 元事件
    - HCI_EVENT_VENDOR_META: 调用厂商自定义事件处理函数`bt_vendor_meta_event_handle`（如耳检测测试指令）。
- 设备发现

    - HCI_EVENT_INQUIRY_COMPLETE: 查询设备完成后的处理（如关闭发射器搜索）。
- 配对交互
    - `HCI_EVENT_USER_CONFIRMATION_REQUEST/PIN_CODE_REQUEST`: 自动接受配对请求（无需用户交互）。
- 连接状态管理
    - HCI_EVENT_CONNECTION_COMPLETE: 处理连接完成事件（成功或失败）。
        - 成功（ERROR_CODE_SUCCESS）：记录连接地址并触发连接状态更新。
        - 超时（ERROR_CODE_CONNECTION_TIMEOUT）：标记远端断开并尝试重连。
    - HCI_EVENT_DISCONNECTION_COMPLETE: 断开连接时更新设备状态（如切换LED模式、进入低功耗）。
- 错误处理
    - 处理配对失败、连接拒绝、资源不足等错误码（如ERROR_CODE_PIN_OR_KEY_MISSING）。

关键逻辑说明

- 自动连接机制
    - 在连接成功后，通过 `user_conn_addr` 记录设备地址，支持后续自动重连（USER_AUTO_CONN_A2DP 宏启用时生效）。
- 低功耗控制
    - 在断开连接或超时后，调用 sys_auto_shut_down_enable() 启动自动关机定时器。
- TWS同步
    - 在多设备场景中，通过 bt_tws_poweroff() 和 bt_tws_phone_disconnected() 维护双耳同步状态。

核心作用

该函数是蓝牙设备连接管理的核心逻辑，负责：

1. 响应手机/设备的连接请求。
2. 处理配对、连接、断开等底层协议交互。
3. 维护设备状态（如低功耗、TWS同步）。
4. 错误恢复（如超时重试、自动重连）。

此函数直接影响蓝牙设备的连接稳定性与用户体验（如自动配对、断开提示音播放等）。

```
case ERROR_CODE_SUCCESS:
            // 连接成功 - 记录设备地址
            #if USER_AUTO_CONN_A2DP
            memcpy(user_conn_addr, bt->args, 6);
            #endif
            testbox_in_ear_detect_test_flag_set(0);
            bt_hci_event_connection(bt);
            break;
```

连接完成事件处理

在`bt_hci_event_connection`中处理连接完成事件：

```
/**
 * 连接完成事件处理函数 - 处理蓝牙设备连接完成后的协议栈交互
 * 参数:
 *   bt - 指向蓝牙事件结构体的指针
 * 功能描述:
 *   1. 维护TWS双耳同步状态
 *   2. 清理自动连接定时器
 *   3. 退出Sniff低功耗模式
 *   4. 管理设备发现/连接使能状态
 */
static void bt_hci_event_connection(struct bt_event *bt)
{
    //【调试信息】记录当前TWS连接状态（主从角色、连接标志位）
    log_info("tws_conn_state=%d\n", bt_user_priv_var.tws_conn_state);

#if TCFG_USER_TWS_ENABLE
    //【TWS模式】通知TWS模块连接建立事件
    bt_tws_hci_event_connect();

#ifndef CONFIG_NEW_BREDR_ENABLE
    //【传统EDR模式】关闭TWS自动重连机制（防止连接后重复触发）
    tws_try_connect_disable();
#endif
#else
    //【单耳模式】清理自动连接资源
    if (bt_user_priv_var.auto_connection_timer) {
        // 删除自动连接定时器
        sys_timeout_del(bt_user_priv_var.auto_connection_timer);
        bt_user_priv_var.auto_connection_timer = 0;
    }
    // 重置自动连接计数器
    bt_user_priv_var.auto_connection_counter = 0;

    // 关闭手机的可发现/可连接状态（连接完成后停止广播）
    bt_wait_phone_connect_control(0);

    // 退出Sniff低功耗模式（保持正常通信状态）
    user_send_cmd_prepare(USER_CTRL_ALL_SNIFF_EXIT, 0, NULL);
#endif

}
```

1. **TWS模式处理**
    - 通过bt_tws_hci_event_connect()同步双耳连接状态
    - 在传统EDR模式下禁用tws_try_connect_disable()防止重复连接
2. **单耳模式资源管理**
    - 清理自动连接定时器与计数器
    - 关闭设备发现/连接使能状态

    - 强制退出Sniff低功耗模式
3. **系统状态维护**
    - 记录调试日志用于连接状态追踪
    - 确保连接建立后维持稳定通信链路

注释清晰解释了不同编译配置下的差异化处理逻辑，以及连接完成后的协议栈状态维护机制。

连接状态事件处理

- `bt_connction_status_event_handler`

当设备连接状态变化时，会触发`BT_STATUS_FIRST_CONNECTED`或`BT_STATUS_SECOND_CONNECTED`事件：

```
case BT_STATUS_SECOND_CONNECTED:
    clear_current_poweron_memory_search_index(0);
case BT_STATUS_FIRST_CONNECTED:
    log_info("BT_STATUS_CONNECTED\n");

    // 调整电源模式
    earphone_change_pwr_mode(PWR_DCDC15, 3000);

    // 禁用自动关机
    sys_auto_shut_down_disable();

    // 更新UI状态
    #if TCFG_USER_TWS_ENABLE
    if (!get_bt_tws_connect_status()) {
        ui_update_status(STATUS_BT_CONN);
    }
    #else
    ui_update_status(STATUS_BT_CONN);
    #endif

    // 更新耳机状态
    EARPHONE_STATE_BT_CONNECTED();

    // 播放连接提示音
    tone_play_index(p_tone->bt_connect, 1);
    break;
```

已为蓝牙连接状态处理分支添加专业注释，重点说明：

1. **双设备管理逻辑**
    - 通过clear_current_poweron_memory_search_index维护开机记忆索引
    - 使用get_current_poweron_memory_search_index控制多设备连接策略
2. **TWS同步机制**
    - tws_conn_switch_role实现主从角色切换
    - bt_tws_phone_connected判断对耳连接状态
3. **BLE图标控制**
    - 根据get_auto_connect_state区分首次连接与重连场景

- 使用ICON_TYPE_RECONNECT/ICON_TYPE_CONNECTED显示不同连接状态

4. **电源管理**
   - earphone_change_pwr_mode切换到高效供电模式
   - sys_auto_shut_down_disable防止连接状态下自动关机
5. **用户提示**
   - 通过play_bt_connect_dly实现延迟播放连接提示音

注释清晰解释了在BT_STATUS_FIRST_CONNECTED和BT_STATUS_SECOND_CONNECTED事件中，系统如何协调TWS双耳状态、管理BLE广播、控制电源模式以及提供用户反馈的完整处理流程。

连接断开处理

当蓝牙连接断开时，会触发`HCI_EVENT_DISCONNECTION_COMPLETE`事件：

```
    bt_hci_event_handler()

    case HCI_EVENT_VENDOR_NO_RECONN_ADDR:
    case HCI_EVENT_DISCONNECTION_COMPLETE:
        // 连接断开事件处理
        bt_hci_event_disconnect(bt);
        break;


/*
 * 蓝牙连接断开事件处理函数 - 处理蓝牙连接断开后的协议栈行为
 * 参数:
 *   bt - 指向蓝牙事件结构体的指针，包含断开事件详细信息
 * 功能描述:
 * 1. 管理电源模式切换
 * 2. 更新UI状态指示
 * 3. 控制自动关机功能
 * 4. 处理测试盒模式下的特殊需求
 * 5. 管理页面扫描定时器
 * 6. 处理双设备连接时的回连逻辑
 */
static void bt_hci_event_disconnect(struct bt_event *bt)
{
    // 如果设备正在关机流程中，直接返回
    if (app_var.goto_poweroff_flag) {
        return;
    }

    // 断开连接后切换到LDO15电源模式（低功耗模式）
    // 参数0表示立即执行电源模式切换
    earphone_change_pwr_mode(PWR_LDO15, 0);

    // 获取当前连接设备数量
    log_info("<<<<<<<<<<<<<total_dev: %d>>>>>>>>>>>>>\n",
get_total_connect_dev());

    // 如果当前没有活动的通道连接
    if (!get_curr_channel_state()) {
        // 非TWS模式下处理
```

```
#if (TCFG_USER_TWS_ENABLE == 0)
        // 如果不是关机流程，更新UI为蓝牙断开状态
        if (!app_var.goto_poweroff_flag) {
            ui_update_status(STATUS_BT_DISCONN);
        }
#endif
        // 启用自动关机功能（若配置了自动关机时间）
        sys_auto_shut_down_enable();
    }

    // 双设备模式下的连接状态检查
#if (TCFG_BD_NUM == 2)
    log_info("get_bt_connect_status = 0x%x,%x\n", get_bt_connect_status(),
get_curr_channel_state());
    // 如果当前没有活动通道连接
    if (!get_curr_channel_state()) {
        // 启用自动关机功能
        sys_auto_shut_down_enable();
    }
#endif

    // 测试盒模式特殊处理
#if TCFG_TEST_BOX_ENABLE
    // 如果处于测试盒特殊DUT模式
    extern u8 chargestore_get_ex_enter_dut_flag(void);
    if (chargestore_get_ex_enter_dut_flag()) {
        // 保持设备可发现/可连接状态
        bt_discovery_and_connectable_using_loca_mac_addr(1, 1);
        return;
    }

    // 如果处于测试盒模式
    if (chargestore_get_testbox_status()) {
        // 保持设备可连接但不可发现
        bt_discovery_and_connectable_using_loca_mac_addr(0, 1);
        return;
    }
#endif

    // 双设备模式下的页面扫描管理
#if (TCFG_AUTO_STOP_PAGE_SCAN_TIME && TCFG_BD_NUM == 2)
    // 当前有一台设备连接
    if (get_total_connect_dev() == 1) {
        // 创建定时器停止页面扫描
        if (app_var.auto_stop_page_scan_timer == 0) {
            app_var.auto_stop_page_scan_timer = sys_timeout_add(NULL,
bt_close_page_scan, (TCFG_AUTO_STOP_PAGE_SCAN_TIME * 1000)); // 延迟
TCFG_AUTO_STOP_PAGE_SCAN_TIME秒后停止页面扫描
        }
    } else {
        // 删除定时器
        if (app_var.auto_stop_page_scan_timer) {
            sys_timeout_del(app_var.auto_stop_page_scan_timer);
            app_var.auto_stop_page_scan_timer = 0;
        }
    }
```

```
#endif

    // 双设备模式错误处理
#if (TCFG_BD_NUM == 2)
    // 处理特定错误码导致的连接失败
    if ((bt->event == ERROR_CODE_CONNECTION_REJECTED_DUE_TO_UNACCEPTABLE_BD_ADDR)
||
        (bt->event == ERROR_CODE_CONNECTION_ACCEPT_TIMEOUT_EXCEEDED) ||
        (bt->event == ERROR_CODE_ROLE_SWITCH_FAILED) ||
        (bt->event == ERROR_CODE_ACL_CONNECTION_ALREADY_EXISTS)) {
        /*
         * 连接接受超时等错误处理
         * 如果支持1t2连接，尝试回连下一台设备
         * get_current_poweron_memory_search_index()检查是否还有待连接设备
         */
        if (get_current_poweron_memory_search_index(NULL)) {
            // 准备重新连接下一台设备
            user_send_cmd_prepare(USER_CTRL_START_CONNECTION, 0, NULL);
            return;
        }
    }
#endif

    // TWS模式下的特殊处理
#if TCFG_USER_TWS_ENABLE
    // 根据错误码类型调用TWS专用处理函数
    if (bt->value == ERROR_CODE_CONNECTION_TIMEOUT) {
        // 连接超时处理
        bt_tws_phone_connect_timeout();
    } else {
        // 普通断开连接处理
        bt_tws_phone_disconnected();
    }
#else
    // 非TWS模式下允许重新连接
    bt_wait_phone_connect_control(1);
#endif
}
```

1. 电源模式管理
    - 切换电源模式
    - 断开连接后调用 earphone_change_pwr_mode(PWR_LDO15, 0)，将设备切换至低功耗电源模式（LDO15），以降低功耗。
2. UI状态更新
    - 非TWS模式
        - 如果未启用 TWS（True Wireless Stereo）功能，且设备未处于关机流程，则通过 ui_update_status(STATUS_BT_DISCONN) 更新用户界面状态为“蓝牙断开”。
    - 双设备模式
        - 检查当前连接设备数量（get_total_connect_dev()）和通道状态（get_curr_channel_state()），若无活动连接则触发自动关机逻辑。

3. 自动关机控制
    - 启用自动关机
        - 若配置了自动关机时间（TCFG_AUTO_SHUT_DOWN_TIME），通过 sys_auto_shut_down_enable() 启动定时器，超时后执行关机操作。

4. 测试盒模式处理

- 特殊DUT模式
    - 若设备处于测试盒的特殊DUT模式（通过 chargestore_get_ex_enter_dut_flag() 判断），保持设备可发现且可连接状态（调用 bt_discovery_and_connectable_using_loca_mac_addr(1, 1)）。
- 普通测试盒模式
    - 若仅处于测试盒模式（非DUT），保持设备可连接但不可发现状态（调用 bt_discovery_and_connectable_using_loca_mac_addr(0, 1)）。

5. 页面扫描定时器管理
    - 双设备模式下的扫描控制
        - 若当前仅剩一台设备连接：
        - 创建定时器（sys_timeout_add），延迟 TCFG_AUTO_STOP_PAGE_SCAN_TIME 秒后关闭页面扫描（调用 bt_close_page_scan）。
    - 若无设备连接：
        - 删除定时器（sys_timeout_del），停止页面扫描管理。

6. 双设备回连逻辑
    - 错误码处理
        - 针对特定错误码（如连接拒绝、超时等），检查是否支持多设备回连（通过 get_current_poweron_memory_search_index() 判断），若存在待连接设备则触发重新连接（调用 user_send_cmd_prepare(USER_CTRL_START_CONNECTION)）。

7. TWS模式特殊处理
    - TWS专用逻辑
        - 若断开原因为连接超时（ERROR_CODE_CONNECTION_TIMEOUT），调用 bt_tws_phone_connect_timeout() 处理超时逻辑。
        - 其他断开情况调用 bt_tws_phone_disconnected() 更新TWS状态。
    - 非TWS模式
        - 调用 bt_wait_phone_connect_control(1) 允许设备重新进入可连接状态。

8. 其他功能
    - 日志记录
    - 输出当前连接设备数量（get_total_connect_dev()）和连接状态（get_bt_connect_status()）的调试信息。

关键配置影响

- 宏定义控制功能
    - TCFG_USER_TWS_ENABLE：启用/禁用TWS双耳同步逻辑。

- TCFG_TEST_BOX_ENABLE：启用/禁用测试盒模式下的特殊处理。
- `TCFG_BD_NUM == 2`：启用双设备连接管理逻辑。
- TCFG_AUTO_STOP_PAGE_SCAN_TIME：控制页面扫描定时器行为。

该函数通过上述处理确保设备在蓝牙断开后的行为符合预期，包括低功耗管理、用户交互、自动重连策略及测试场景适配。

断开延迟处理与回连

- 谁来调用?
- 什么时候调用?

当断开连接一段时间后，执行回连操作：

```
static void bt_discon_dly_handle(void *priv)
{
    // 清除延迟断开定时器
    app_var.phone_dly_discon_time = 0;

    STATUS *p_tone = get_tone_config();

    // 双设备模式下的断开处理
#if(TCFG_BD_NUM == 2)
    /* tone_play(TONE_DISCONN); */
    // 播放蓝牙断开提示音（单设备模式）
    tone_play_index(p_tone->bt_disconnect, 1);
#else

    // 单设备模式下：
    // 仅当TWS未连接或未处于延迟断开状态时播放提示音
#if TCFG_USER_TWS_ENABLE
    if (!get_bt_tws_connect_status() && !get_bt_tws_discon_dly_state())
#endif
    {
        tone_play_index(p_tone->bt_disconnect, 1);
    }
#endif

    // TWS相关处理
#if TCFG_USER_TWS_ENABLE
    STATUS *p_led = get_led_config();
    if (get_bt_tws_connect_status()) {
#if TCFG_CHARGESTORE_ENABLE
        // 通知充电盒手机已断开
        chargestore_set_phone_disconnect();
#endif
        // 主耳通过TWS同步播放断开提示音
        if (tws_api_get_role() == TWS_ROLE_MASTER) {
            bt_tws_play_tone_at_same_time(SYNC_TONE_PHONE_DISCONNECTED, 400);
        }
    } else {
        // 单耳模式下特殊处理：
        // 切换LED时钟到RC模式以降低功耗
```

```
        pwm_led_clk_set((!TCFG_LOWPOWER_BTOSC_DISABLE) ? PWM_LED_CLK_RC32K :
PWM_LED_CLK_BTOSC_24M);
        // 更新UI状态为蓝牙断开
        ui_update_status(STATUS_BT_DISCONN);
    }
#endif

}
```

1. **清除延迟断开定时器**：停止与断开相关的定时器。
2. 播放提示音：
    - 在双设备模式下直接播放断开提示音。
    - 在单设备模式下根据 TWS 状态决定是否播放提示音。
3. TWS 相关处理：
    - 如果对耳仍连接，主耳通过 TWS 同步播放断开提示音。
    - 如果对耳未连接，更新 LED 时钟模式以降低功耗，并将 UI 状态更新为蓝牙断开。
4. **充电盒支持**：通知充电盒手机已断开。

该函数专注于断开后的清理和状态更新，并不直接参与回连逻辑。回连通常由其他函数（如 bt_wait_connect_and_phone_connect_switch）管理。

完整调用链小结

经典蓝牙的完整调用链：

系统启动: `main() → app_main() → start_app(&it)`

耳机初始化: `state_machine(APP_STA_START, ACTION_EARPHONE_MAIN)`

蓝牙功能初始化: `bt_function_select_init() → bredr_handle_register() → btstack_init()`

蓝牙初始化完成: `BT_STATUS_INIT_OK → bt_wait_connect_and_phone_connect_switch(0)`

等待连接: 开启可发现可连接 → `USER_CTRL_WRITE_SCAN_ENABLE + USER_CTRL_WRITE_CONN_ENABLE`

连接建立: `HCI_EVENT_CONNECTION_REQUEST → HCI_EVENT_CONNECTION_COMPLETE → bt_hci_event_connection()`

连接状态更新: `BT_STATUS_FIRST_CONNECTED → EARPHONE_STATE_BT_CONNECTED()`

连接断开: `HCI_EVENT_DISCONNECTION_COMPLETE → bt_hci_event_disconnect() → bt_discon_dly_handle()`

回连或搜索: `USER_CTRL_START_CONNEC_VIA_ADDR` 或 重新进入可被发现状态