AC630N_bt_data_transfer_sdk

介绍&用户手册

珠海市杰理科技股份有限公司

Zhuhai Jieli Technology Co.,LTD

修改记录

版本	更新日期	描述
0.1.0	2019 / 12 / 03	用户手册
更新：	● 建立初始版本 ● 定义文档格 ● 描述 SDK 功能	
0.2.0	2020 / 06 / 02	
	● 整合了 HID 和 SPP_AND_BLE 的 SDK 工程 ● 增加 AT+数传（SPP+BLE）描述 ● 增加 APP 升级（BLE）描述 ● 增加 BLE 主机 client 功能描述	
0.3.0	2020 / 07 / 09	
	● 修改更新部分文档说明	
0.4.0	2020 / 09 / 11	
	● AC636N 系列的 添加 MOUSE 和 DONGLE 蓝牙和 2.4G 模式 CASE 支持 ● 添加 AT 命令控制 BLE 主机 ● HID 增加自拍器、翻页器和标准键盘 CASE ● 增加支持芯片 AC635N 系列	
0.5.0	2020 / 11 / 05	
	● 添加支持 AC637N 系列芯片 ● 添加配对识别对方系统接口	
0.6.0	2020 / 12 / 15	
	● 添加 audio 部分。	
0.7.0	2021 / 1 / 8	
	● 添加 audio 使用说明。 ● Keypage 添加失败手机系统，切换描述符	
0.8.0	2021 / 3 / 1	
	● 添加支持 AC632N 系列芯片 ● 添加 GameBox 示例	
0.9.0	2021 / 4 / 2	
	● 添加蓝牙多连接示例	

	● 添加 BLE 的 IBEACON 广播示例 ● 添加 AT COM 字符串方式控制
0.9.1	2021 / 4 / 23
0.9.2	2021 / 5 / 14
1.0.0	● 非连接 2.4G 数据收发示例 ● 腾讯连接应用说明 ● 添加 AUDIO 的 MIDI 使用说明
2.0.0	2021 / 9 / 13 ● 调整代码结构，修复 apps 说明 ● 添加涂鸦协议支持 ● 添加语音遥控 hid 支持编码的示例 ● INI 配置文件生成 ● 新增 GATT 公共模块说明
2.1.0	2022 / 3 / 10 ● 增加 lighting 握手充电功能 ● 增加在线串口升级功能 ● OTA 增加可跳转 MASKROM 的升级方式 ● MESH 工程支持 AUDIO 功能
2.2.1	2021 / 11 / 23 ● 数传配置 BLE 从机可以搜索对方主机的 GATT 服务示例 ● 数传配置 BLE 主机可提供 GATT 服务给从机搜索示例 ● 数传配置 BLE 配置传输速度提速说明 ● 数传添加可使用优先级高的 ATT 发送缓存说明 ● Usb Dongle 新增支持 OTA 升级说明 ● 新增 BLE 对接 HiLink 协议 示例说明 ● 系统不可屏蔽中断支持 ● VBAT 供电和系统频率关系限制

目录

Chapter 1 SDK 快速使用说明

1.1 编写目的

该文档主要描述 AC630N_SDK 开发包的使用方法及开发中注意的一些问题，为用户进行二次开发提供参考,其中包括：

安装工具链、安装包管理工具、配置工具使用、INI 配置文件生成等注意事项。

1.2 安装工具链

1.2.1 安装包

为了进行 SDK 开发，需要安装下面两个工具：

1. 集成开发环境 Codeblocks

(1) 下载地址：http://jl-update.oss-cn-shenzhen.aliyuncs.com/codeblocks-latest.exe

2. 编译器

(2) 下载地址：http://jl-update.oss-cn-shenzhen.aliyuncs.com/jieli-windows-toolchains-latest.exe

注意：需要先安装 Codeblocks，后安装编译器；按提示安装即可。

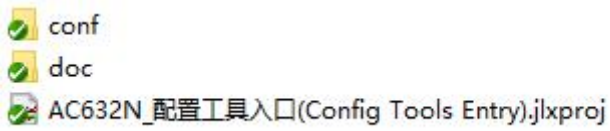
1.3 安装包管理工具

❖ 包管理工具安装

包管理工具，是用于下载杰理 SDK 需要使用的工具的工具，同时也负责打开 jlproxj 后缀的文件。即，包管理工具负责从杰理服务器端，把 SDK 所需要的最新工具，下载到当前电脑。

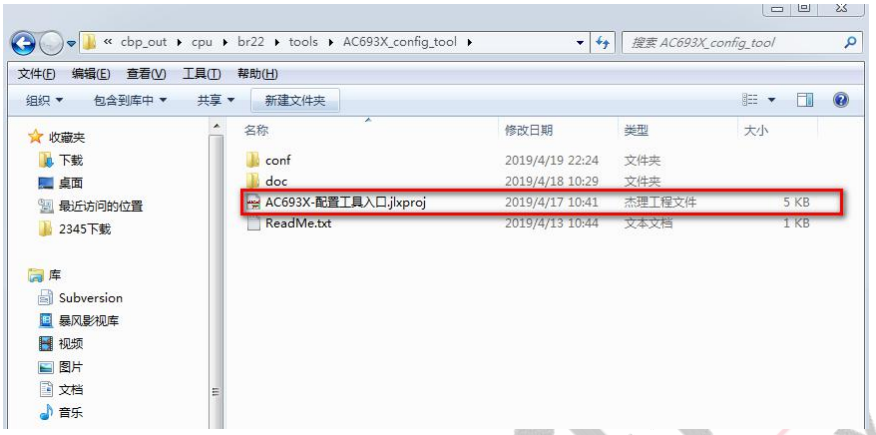
（包管理工具下载路径：http://jl-update.oss-cn-shenzhen.aliyuncs.com/jieli-pkgman-setup-latest.exe）

双击*.exe 安装，按照提示安装完成即可。安装完成后，找到文件目录即可打开双击直接打开后缀为.jlxproj 的文件。如下图所示，配置路径：sdk\cpu\bd19\tools\AC632N_config_tool

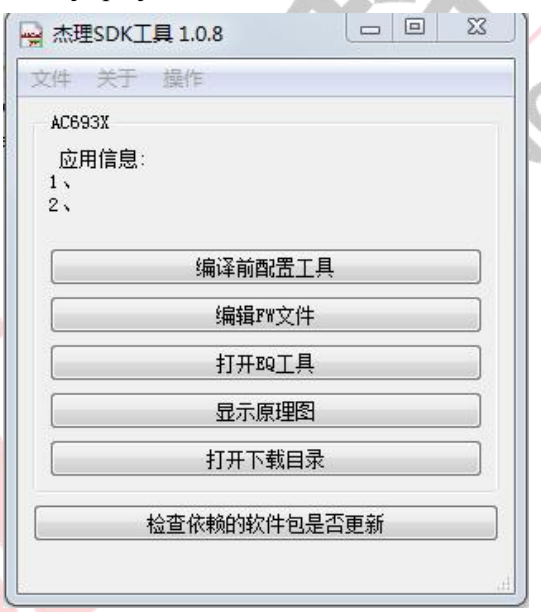


1.4 配置工具使用

1、打开 sdk 工程目录，进入 cbp_out\cpu\BD29\tools\AC630N_config_tool 目录；

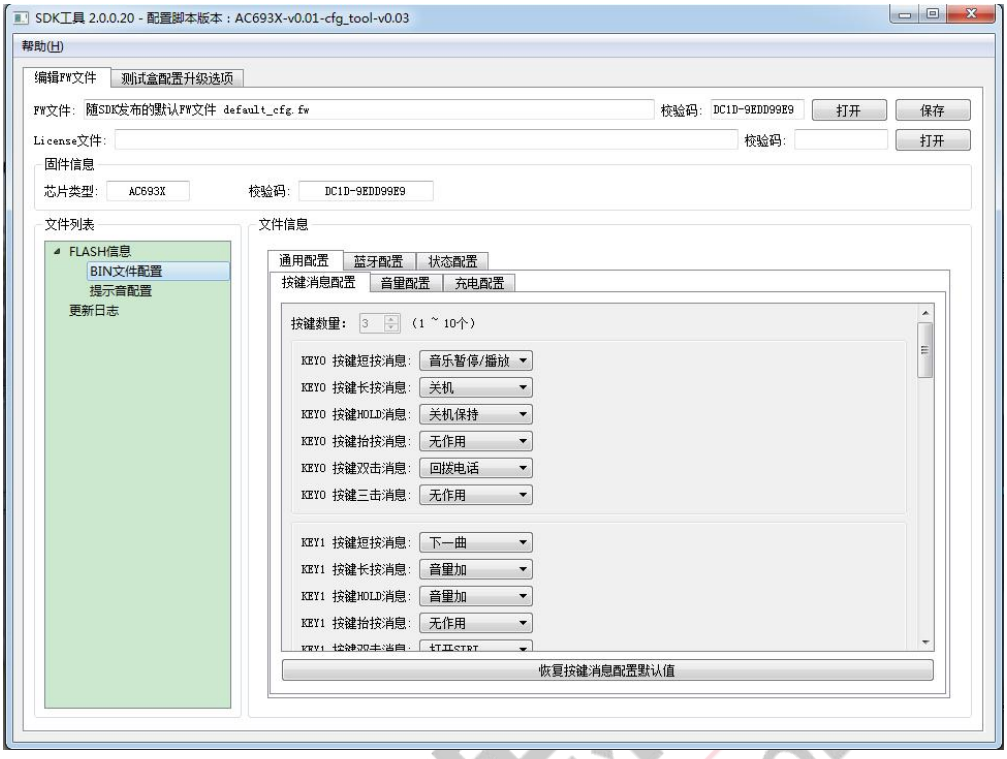


2、双击【AC630N-配置工具入口.jlxproj】，打开【杰理 SDK 工具】

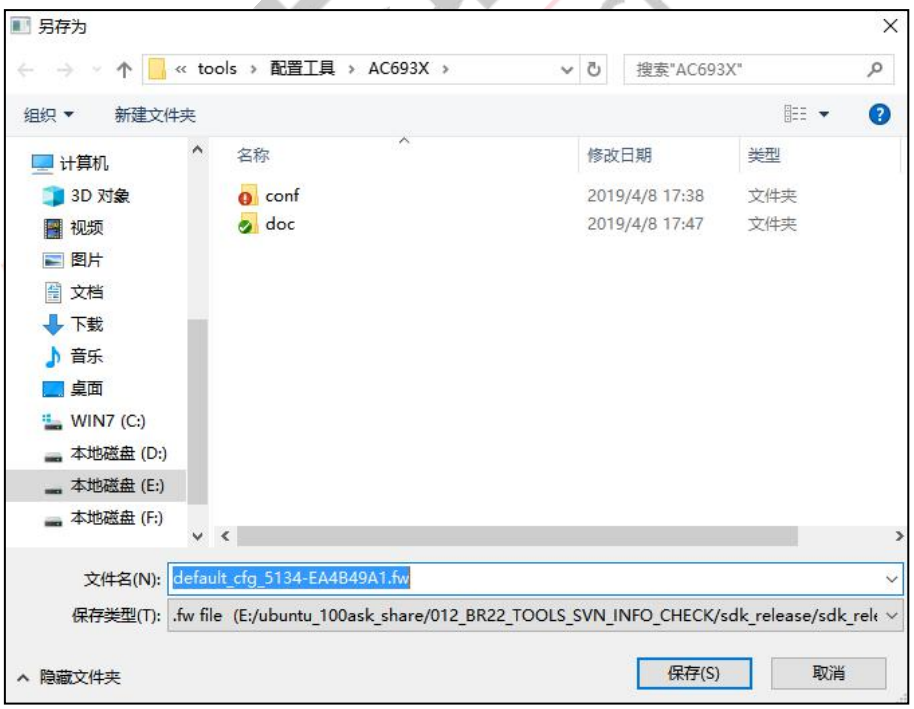


【杰理 SDK 配置工具】界面说明：

1）编辑 FW 文件：点击【编辑 FW 文件】，可对一个 FW 文件的进行编辑，修改 FW 文件的板级配置、蓝牙配置，状态配置和提示音配置，界面如下：



点击【恢复默认值】可以恢复 sdk 发布时的原始配置，配置完成后，点击【保存】，可选择保存路径保存修改过后的 fw 和 ufw 文件；



编辑 fw 文件有以下几点需要注意：

（1）板级配置的可配选项取决 cfg_tool.bin 文件，也就是说编译前配置选了什么，编辑 fw 文件就有什么配置，比如说编译前配置是选了 3 个 key，则编辑 fw 文件的时候也是只有 3 个 key 的选项。

（2）fw 版本号控制，只有工具和 fw 的版本号对得上，才能编辑对应的 fw 文件，版本号在 AC630N_config_tool\conf\entry 目录下的 user_cfg.lua 文件下修改，如下图：

```
                    设置fw版本信息
cfg:addKeyInfo("script_version", "AC693X-v0.0.3");

-------------------- 设置应用名称 --------------------
product_name = "AC693X"; --此名称将显示于配置工具入口界面

-------------------- 设置配置工具开发状态 --------------
    -- develop: 开发状态，用于开发SDK使用
    -- release: 发布状态，用于发布上传使用
```

如果版本号提示不对，请确认版本号。

（3）fw 文件制作：当 fw 文件测试没有问题后，想发布一个可编辑的 fw 文件时，把生成的 fw 文件替换 AC630N_config_tool\conf\output\default 目录下 default_cfg.fw 文件即可

（4）默认值的制作：用编译前选项配置好后，点击保存后会在 AC630N_config_tool\conf\output 生成一个 default_cfg.lua 文件，把该文件覆盖 AC630N_config_tool\conf\output\default 目录下的 default_cfg.lua 文件，到时在编辑 fw 文件的时候点击恢复默认设置，就会恢复之前设置的值。

2）显示原理图：点击【显示原理图】，可以打开一个 doc 的文件夹，可以在该文件夹下存放原理图等相关文件；

3）编译前配置：点击【编译前配置】，可以在 sdk 代码编译前进行相关配置项配置，打开界面如下：

可以在编译前配置工具中进行通用配置、蓝牙配置、提示音配置和状态配置，点击【恢复默认值】可以恢复 sdk 发布时的原始配置，配置完成后，点击【保存】保存配置，将会输出 cfg_tools.bin 到下载目录中，重新编译 sdk 代码可应用该最新配置；

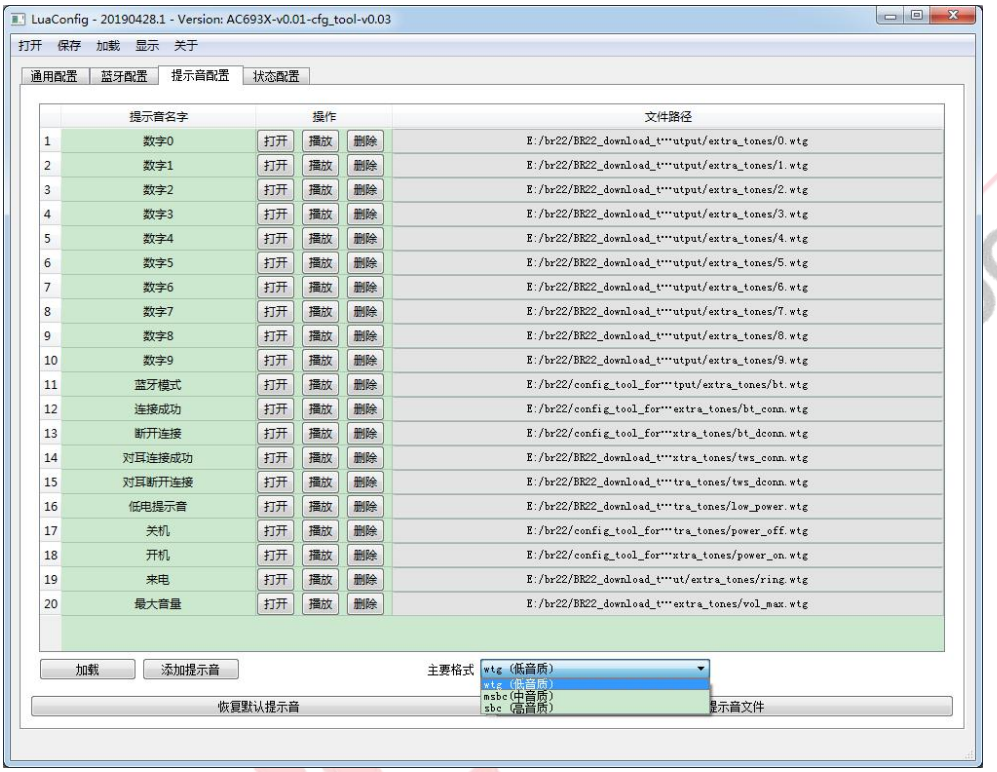
4）打开下载目录：点击【打开下载目录】，将会打开 tools 下载目录，cfg_tools.bin 将会输出到该目

录下；

5）检查依赖的软件包是否更新：点击【检查依赖的软件包是否更新】，将会查检查相关配置工具的版本更新情况，如有更新，则下载相应更新即可；

3、提示音配置

提示音设置在工具的“提示音配置”选项里面，点击后如下图所示：



默认是加载 conf\output\extra_tones 下的提示音，如果有一个新的 tone.cfg 文件，点击加载按钮可以把该提示音文件加载进来。如果想更换提示音，直接把对应的提示音文件拉到对应选项即可。提示音工具提供 3 种音质提示音格式，每个提示音均可以单独指定成不同格式，主要是看 flash 的空间。比如“蓝牙模式”提示音想音质好点，可以先选 sbc，然后把对应提示音文件拉过去即可，其他格式也是可以这样操作，每个提示音生成是什么格式取决于拉提示音文件过去的时候选的是什么格式。

注意：报号数字提示音格式一定要选同一种,原始音频数据 sbc 要用 32k 的采样率转，msbc 要用 16k 的采样率转才能保证音质。

1.5 下载目录 INI 配置文件生成

主要描述新版本 SDK(v2.0.0 及以后)对 tools 目录下 isd_config.ini 配置文件的配置和生成。

SDK 配置文件 isd_config.ini 由 isd_config_rule.c 根据对应板级配置 board_xxx_global_build_cfg.h 生成。需要注意的是如果想要修改 isd_config.ini，必须修改 board_xxx_global_build_cfg.h 或者 isd_config_rule.c，如直接修改 isd_config.ini，则每次编译修改将会**被覆盖。**

1.4.1 复位源

在 board_xxx_global_build_cfg.h

```
1.  //config long-press reset io pin,time,trigger level
2.  #if CONFIG_LP_TOUCH_KEY_EN
3.  #define CONFIG_RESET_PIN                    LDO  //io pin
4.  #define CONFIG_RESET_TIME                   04   //unit:second
5.  #define CONFIG_RESET_LEVEL                  1  //tigger level(0/1)
6.  #else
7.  #define CONFIG_RESET_PIN                    PB01 //io pin
8.  #define CONFIG_RESET_TIME                   08   //unit:second
9.  #define CONFIG_RESET_LEVEL                  0  //tigger level(0/1)
10. #endif
```

1.4.2 OTA 升级

在 board_xxx_global_build_cfg.h

```
1.  #define CONFIG_DOUBLE_BANK_ENABLE           0       //单双备份选择(若打开了改宏,FLASH 结构变为双备份结构，适用于接入第三方协议的OTA， PS: JL-OTA 同样支持双备份升级， 需要根据实际 FLASH 大小同时配置 CONFIG_FLASH_SIZE)
2.  #define CONFIG_APP_OTA_ENABLE               0       //是否支持 RCSP 升级(JL-OTA)
```

1.4.3 PID 和 VID

在 board_xxx_global_build_cfg.h

```
1.  //DON'T MODIFY THIS CONFIG EXCEPT SDK PUBLISHER
```

```
#define CONFIG_CHIP_NAME                    AC637N              //除了SDK 发布者,请不要修改
//it can be modified before first programming,but keep the same as the original version
#define CONFIG_PID                          AC637N              //烧写或强制升级之前可以修改,之后升级要保持一致
//it can be modified before first programming,but keep the same as the original version
#define CONFIG_VID                          0.01    //烧写或强制升级之前可以修改,之后升级要保持一致
```

这里需要注意，同一个产品所有固件需要保证 CHIP、PID、VID 一致，否则将无法烧写和升级

1.4.4 VM 大小

```
//with single-bank mode,actual vm size should larger this VM_LEAST_SIZE, and dual bank mode,actual vm size equals this;
#define CONFIG_VM_LEAST_SIZE                8K
```

1.4.5 uboot_debug、ota_debug.bin 打印口配置

在 board_xxx_global_build_cfg.h

```
//isd_download loader/uboot/update_loader debug io config
//#define CONFIG_UBOOT_DEBUG_PIN                PA05
//#define CONFIG_UBOOT_DEBUG_BAUD_RATE          1000000
```

1.4.6 其他配置

其他配置请参考 board_xxx_global_build_cfg.h 中注释进行配置

Chapter 2　APP 使用说明

2.1APP 概述

本章主要介绍通用蓝牙常见应用方案，让用户能从最接近目标产品的 APP 开始开发，SDK 集成了蓝牙应用 App，丰富的板级配置，外设驱动，以及蓝牙库，后续将会持续集成更多通用蓝牙方案。

2.2 APP - Bluetooth Dual-Mode SPP+BLE

2.2.1 概述

支持蓝牙双模透传传输功能。CLASSIC 蓝牙使用标准串口 SPP profile 协议，BLE 蓝牙使用自定义的 profile 协议，提供 ATT 的 WRITE、WRITE_WITHOUT_RESPONSE，NOTIFY 和 INDICATE 等属性传输收发数据。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.2.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631N_spp_and_le.cbp

（1）先配置板级 board_config.h 和对应配置文件中蓝牙双模使能

```
/*
 *  板级配置选择
 */
#define CONFIG_BOARD_AC631N_DEMO
// #define CONFIG_BOARD_AC6311_DEMO

#include "board_ac631n_demo_cfg.h"
#include "board_ac6311_demo_cfg.h"
```

//在 board_ac631n_demo_cfg.h 中配置是否打开 edr 和 ble 模块

```
#define TCFG_USER_BLE_ENABLE                    1   //BLE 功能使能
#define TCFG_USER_EDR_ENABLE                    1   //EDR 功能使能
```

（2）配置 app 选择：“app_config.h”

```
//apps example 选择,只能选 1 个,要配置对应的 board_config.h
#define CONFIG_APP_SPP_LE                   1 //SPP + LE or LE's client
#define CONFIG_APP_MULTI                    0 //蓝牙 LE 多连 + spp
#define CONFIG_APP_DONGLE                   0 //usb + 蓝牙(ble 主机),PC hid 设备
#define CONFIG_APP_CENTRAL                  0 //ble client,中心设备
```

```
#define CONFIG_APP_LL_SYNC              0 //腾讯连连
#define CONFIG_APP_BEACON               0 //蓝牙BLE ibeacon
#define CONFIG_APP_NONCONN_24G          0 //2.4G 非连接收发
#define CONFIG_APP_TUYA                 0 //涂鸦协议
#define CONFIG_APP_AT_COM               0 //AT com HEX 格式命令
#define CONFIG_APP_AT_CHAR_COM          0 //AT com 字符串格式命令
#define CONFIG_APP_IDLE                 0 //空闲任务
```

（3）配置对应 case 需求：“app_config.h”

```
#if CONFIG_APP_SPP_LE
//配置双模同名字，同地址
#define DOUBLE_BT_SAME_NAME             0 //同名字
#define DOUBLE_BT_SAME_MAC              0 //同地址
#define CONFIG_APP_SPP_LE_TO_IDLE       0 //SPP_AND_LE To IDLE Use
```

（3）蓝牙的 BLE 配置，保护 GATT 和 SM 的配置

```
//蓝牙BLE 配置
#define CONFIG_BT_GATT_COMMON_ENABLE        1 //配置使用gatt 公共模块
#define CONFIG_BT_SM_SUPPORT_ENABLE         0 //配置是否支持加密
#define CONFIG_BT_GATT_CLIENT_NUM           0 //配置主机 client 个数 (app not support)
#define CONFIG_BT_GATT_SERVER_NUM           1 //配置从机 server 个数
#define CONFIG_BT_GATT_CONNECTION_NUM       (CONFIG_BT_GATT_SERVER_NUM + CONFIG_BT_GA
TT_CLIENT_NUM) //配置连接个数
```

2.2.3 SPP 数据通信

SPP ---推荐使用手机测试工具： “蓝牙串口”

1、代码文件 spp_trans.c

2、接口说明：“spp_trans.c”

（1）SPP 模块初始化

```
1.  void transport_spp_init(void)
2.  {
3.      log_info("trans_spp_init\n");
4.      log_info("spp_file: %s", __FILE__);
5.  #if (USER_SUPPORT_PROFILE_SPP==1)
6.      spp_state = 0;
7.      spp_get_operation_table(&spp_api);
8.      spp_api->regist_recieve_cbk(0, transport_spp_recieve_cbk);
9.      spp_api->regist_state_cbk(0, transport_spp_state_cbk);
10.     spp_api->regist_wakeup_send(NULL, transport_spp_send_wakeup);
11. #endif
12.
13. #if TEST_SPP_DATA_RATE
14.     spp_timer_handle  = sys_timer_add(NULL, test_timer_handler, SPP_TIMER_MS);
15. #endif
16.
17. }
```

（2）SPP 连接和断开事件处理

```
1.  static void transport_spp_state_cbk(u8 state)
2.  {
3.      spp_state = state;
4.      switch (state) {
5.      case SPP_USER_ST_CONNECT:
6.          log_info("SPP_USER_ST_CONNECT ~~~\n");
7.
8.          break;
9.
10.     case SPP_USER_ST_DISCONN:
11.         log_info("SPP_USER_ST_DISCONN ~~~\n");
12.         spp_channel = 0;
13.
14.         break;
```

```
15.
16.     default:
17.         break;
18.     }
19.
20. }
```

（3）SPP 发送数据接口，发送前先调用接口 transport_spp_send_data_check 检查

```
1.  int transport_spp_send_data(u8 *data, u16 len)
2.  {
3.      if (spp_api) {
4.          log_info("spp_api_tx(%d) \n", len);
5.          /* log_info_hexdump(data, len); */
6.          /* clear_sniff_cnt(); */
7.          bt_comm_edr_sniff_clean();
8.          return spp_api->send_data(NULL, data, len);
9.      }
10.     return SPP_USER_ERR_SEND_FAIL;
11. }
```

（4）SPP 检查是否可以往协议栈发送数据

```
1.  int transport_spp_send_data_check(u16 len)
2.  {
3.      if (spp_api) {
4.          if (spp_api->busy_state()) {
5.              return 0;
6.          }
7.      }
8.      return 1;
9.  }
```

（5）SPP 发送完成回调，表示可以继续往协议栈发数，用来触发继续发数

```
1.  static void transport_spp_send_wakeup(void)
```

```
{
    putchar('W');
}
```

（6）SPP 接收数据接口

```
static void transport_spp_recieve_cbk(void *priv, u8 *buf, u16 len)
{
    spp_channel = (u16)priv;
    log_info("spp_api_rx(%d) \n", len);
    log_info_hexdump(buf, len);
    clear_sniff_cnt();
    ......
```

3、收发测试：“spp_trans.c”

代码已经实现收到手机的 SPP 数据后，会主动把数据回送，测试数据收发。

```
//loop send data for test
if (transport_spp_send_data_check(len)) {
    log_info("-loop send\n");
    transport_spp_send_data(buf, len);
}
```

4、串口的 UUID：“lib_profile_config.c”

串口的 UUID 默认是 16bit 的 0x1101。若要修改可自定义的 16bit 或 128bit UUID，可修改 SDP 的 S 信息结构体 `sdp_spp_service_data`，具体查看 `16it` 和 `128bit UUID` 填写示例。主要 channel id 默认是 1，不能修改。

```
#if (USER_SUPPORT_PROFILE_SPP==1)
u8 spp_profile_support = 1;
SDP_RECORD_HANDLER_REGISTER(spp_sdp_record_item) = {
    .service_record = (u8 *)sdp_spp_service_data,
    .service_record_handle = 0x00010004,
};
#endif
```

16it 和 128bit UUID 填写示例如下：

```
/*128 bit uuid:  11223344-5566-7788-aabb-8899aabbccdd  */
const u8 sdp_test_spp_service_data[96] = {
    0x36, 0x00, 0x5B, 0x09, 0x00, 0x00, 0x0A, 0x00, 0x01, 0x00, 0x04, 0x09, 0x00, 0x01, 0x36, 0x00,
    0x11, 0x1C,

    0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0xaa, 0xbb, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, //uuid128

    0x09, 0x00, 0x04, 0x36, 0x00, 0x0E, 0x36, 0x00, 0x03, 0x19, 0x01, 0x00, 0x36, 0x00,
    0x05, 0x19, 0x00, 0x03, 0x08, 0x01, 0x09, 0x00, 0x09, 0x36, 0x00, 0x17, 0x36, 0x00, 0x14, 0x1C,

    0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0xaa, 0xbb, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, //uuid128

    0x09, 0x01, 0x00, 0x09, 0x01, 0x00, 0x25, 0x06, 0x4A, 0x4C, 0x5F, 0x53, 0x50, 0x50, 0x00, 0x00,
};

//spp 16bit uuid  11 01
const u8 sdp_spp_update_service_data[70] = {
    0x36, 0x00, 0x42, 0x09, 0x00, 0x00, 0x0A, 0x00, 0x01, 0x00, 0x0B, 0x09, 0x00, 0x01, 0x36, 0x00,
    0x03, 0x19,

    0x11, 0x01, //uuid16

    0x09, 0x00, 0x04, 0x36, 0x00, 0x0E, 0x36, 0x00, 0x03, 0x19, 0x01, 0x00,
    0x36, 0x00, 0x05, 0x19, 0x00, 0x03, 0x08, 0x01, 0x09, 0x00, 0x09, 0x36, 0x00, 0x09, 0x36, 0x00,
    0x06, 0x19,

    0x11, 0x01, //uuid16

    0x09, 0x01, 0x00, 0x09, 0x01, 0x00, 0x25, 0x09, 0x4A, 0x4C, 0x5F, 0x53,
    0x50, 0x50, 0x5F, 0x55, 0x50, 0x00,
```

```
};
```

2.2.4 BLE 数据通信

 ---推荐使用手机测试工具： “nRF Connect”

1、代码文件 ble_trans.c

2、Profile 生成的 trans_profile_data 数据表放在 ble_trans_profile.h。用户可用工具 make_gatt_services（sdk 的 tools 目录下）按照“make_gatt_services 工具说明.pdf”自定义修改，重新配置 GATT 服务和属性等。

```
//////////////////////////////////////////////////////
//
// 0x0001 PRIMARY_SERVICE  1800
//
//////////////////////////////////////////////////////
0x0a, 0x00, 0x02, 0x00, 0x01, 0x00, 0x00, 0x28, 0x00, 0x18,

/* CHARACTERISTIC,  2a00, READ | WRITE | DYNAMIC, */
// 0x0002 CHARACTERISTIC 2a00 READ | WRITE | DYNAMIC
0x0d, 0x00, 0x02, 0x00, 0x02, 0x00, 0x03, 0x28, 0x0a, 0x03, 0x00, 0x00, 0x2a,
// 0x0003 VALUE 2a00 READ | WRITE | DYNAMIC
0x08, 0x00, 0x0a, 0x01, 0x03, 0x00, 0x00, 0x2a,

//////////////////////////////////////////////////////
//
// 0x0004 PRIMARY_SERVICE  ae30
//
//////////////////////////////////////////////////////
0x0a, 0x00, 0x02, 0x00, 0x04, 0x00, 0x00, 0x28, 0x30, 0xae,

```

```
/* CHARACTERISTIC,  ae01, WRITE_WITHOUT_RESPONSE | DYNAMIC, */
// 0x0005 CHARACTERISTIC ae01 WRITE_WITHOUT_RESPONSE | DYNAMIC
0x0d, 0x00, 0x02, 0x00, 0x05, 0x00, 0x03, 0x28, 0x04, 0x06, 0x00, 0x01, 0xae,
// 0x0006 VALUE ae01 WRITE_WITHOUT_RESPONSE | DYNAMIC
0x08, 0x00, 0x04, 0x01, 0x06, 0x00, 0x01, 0xae,
```

3、接口说明：“ble_trans.c”

（1）配置广播 ADV 数据

```
static int trans_make_set_adv_data(void)
{
```

（2）配置广播 RESPONE 数据

```
static int trans_make_set_rsp_data(void)
{
```

（3）配置发送缓存大小

```
#define ATT_LOCAL_PAYLOAD_SIZE    (64*2) //note: need >= 20
#define ATT_SEND_CBUF_SIZE        (512) //note: need >= 20,缓存大小可修改
```

（4）协议栈事件回调处理，主要是连接、断开等事件：“le_gatt_server.c”

```
static int trans_event_packet_handler(int event, u8 *packet, u16 size, u8 *ext_param
)
{
    /* log_info("event: %02x,size= %d\n",event,size); */
    switch (event) {
    case GATT_COMM_EVENT_CONNECTION_COMPLETE:
        log_info("connection_handle:%04x\n", little_endian_read_16(packet, 0));
        log_info("peer_address_info:");
        put_buf(&ext_param[7], 7);

        trans_con_handle = little_endian_read_16(packet, 0);
        trans_connection_update_enable = 1;

```

```
        log_info("con_interval = %d\n", little_endian_read_16(ext_param, 14 + 0));
        log_info("con_latency = %d\n", little_endian_read_16(ext_param, 14 + 2));
        log_info("cnn_timeout = %d\n", little_endian_read_16(ext_param, 14 + 4));
        break;

    case GATT_COMM_EVENT_DISCONNECT_COMPLETE:
```

（5）ATT 读事件处理：“ble_trans.c”

```
static uint16_t trans_att_read_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t offset, uint8_t *buffer, uint16_t buffer_size)
{
    uint16_t  att_value_len = 0;
    uint16_t handle = att_handle;

    log_info("read_callback,conn_handle =%04x, handle=%04x,buffer=%08x\n", connection_handle, handle, (u32)buffer);

    switch (handle) {
    case ATT_CHARACTERISTIC_2a00_01_VALUE_HANDLE: {
        char *gap_name = ble_comm_get_gap_name();
        att_value_len = strlen(gap_name);

        if ((offset >= att_value_len) || (offset + buffer_size) > att_value_len) {
            break;
        }

        if (buffer) {
            memcpy(buffer, &gap_name[offset], buffer_size);
            att_value_len = buffer_size;
```

（6）ATT 写事件处理：“ble_trans.c”

```
1.  static int trans_att_write_callback(hci_con_handle_t connection_handle, uint16_t att
    _handle, uint16_t transaction_mode, uint16_t offset, uint8_t *buffer, uint16_t buffe
    r_size)
2.  {
3.      int result = 0;
4.      u16 tmp16;
5.
6.      u16 handle = att_handle;
7.
8.      log_info("write_callback,conn_handle =%04x, handle =%04x,size =%d\n", connection
    _handle, handle, buffer_size);
9.
10.     switch (handle) {
11.
12.     case ATT_CHARACTERISTIC_2a00_01_VALUE_HANDLE:
13.         break;
```

（7）NOTIFY 和 INDICATE 发送接口，发送前调接口 app_send_user_data_check 检查

```
1.  static int app_send_user_data(u16 handle, u8 *data, u16 len, u8 handle_type)
2.  {
3.      u32 ret = APP_BLE_NO_ERROR;
4.
5.      if (!con_handle) {
6.          return APP_BLE_OPERATION_ERROR;
7.      }
8.
9.      if (!att_get_ccc_config(handle + 1)) {
10.         log_info("fail,no write ccc!!!,%04x\n", handle + 1);
11.         return APP_BLE_NO_WRITE_CCC;
12.     }
13.
14.     ret = ble_op_multi_att_send_data(con_handle, handle, data, len, handle_type);
```

（8）检查是否可以往协议栈发送数据

```
1.  //收发测试，自动发送收到的数据;for test
2.  if (ble_comm_att_check_send(connection_handle, buffer_size)) {
```

（9）发送完成回调，表示可以继续往协议栈发数，用来触发继续发数

```
1.      case GATT_COMM_EVENT_CAN_SEND_NOW:
2.  #if TEST_AUDIO_DATA_UPLOAD
3.          trans_test_send_audio_data(0);
4.  #endif
5.          break;
```

4、收发测试：“ble_trans.c”

使用手机 NRF 软件，连接设备后；使能 notify 和 indicate 的 UUID (AE02 和 AE05) 的通知功能后；可以通过向 write 的 UUID (AE01 或 AE03) 发送数据；测试 UUID (AE02 或 AE05)是否收到数据。

```
1.       case ATT_CHARACTERISTIC_ae01_01_VALUE_HANDLE:
2.          log_info("\n-ae01_rx(%d):", buffer_size);
3.          put_buf(buffer, buffer_size);
4.
5.          //收发测试，自动发送收到的数据;for test
6.          if (ble_comm_att_check_send(connection_handle, buffer_size)) {
7.              log_info("-loop send1\n");
8.              ble_comm_att_send_data(connection_handle, ATT_CHARACTERISTIC_ae02_01_VAL
    UE_HANDLE, buffer, buffer_size, ATT_OP_AUTO_READ_CCC);
9.          }
10.         break;
11.
12.     case ATT_CHARACTERISTIC_ae03_01_VALUE_HANDLE:
13.         log_info("\n-ae_rx(%d):", buffer_size);
14.         put_buf(buffer, buffer_size);
15.
16.         //收发测试，自动发送收到的数据;for test
17.         if (ble_comm_att_check_send(connection_handle, buffer_size)) {
18.             log_info("-loop send2\n");
```

```
19.             ble_comm_att_send_data(connection_handle, ATT_CHARACTERISTIC_ae05_01_VAL
    UE_HANDLE, buffer, buffer_size, ATT_OP_AUTO_READ_CCC);
20.         }
21.         break;
```

2.2.5 BLE 搜索主机 Profile 服务示例

在现有的已连接的链路，执行搜索对方的 GATT 服务；操作 GATT 数据收发；目前支持指定的服务搜索，或者搜索苹果服务 ANCS 和 AMS 的支持。

```
1.  //BLE 从机扩展搜索对方服务功能,需要打开 GATT CLIENT
2.  #if CONFIG_BT_GATT_CLIENT_NUM
3.  #define TRANS_CLIENT_SEARCH_PROFILE_ENABLE  1/*配置模块搜索指定的服务*/
4.
5.  #if !TRANS_CLIENT_SEARCH_PROFILE_ENABLE && CONFIG_BT_SM_SUPPORT_ENABLE /*定制搜索
    ANCS&AMS 服务*/
6.  #define TRANS_ANCS_EN                   1/*配置搜索主机的 ANCS 服务,要开配对绑定*/
7.  #define TRANS_AMS_EN                    0/*配置搜索主机的 ANCS 服务,要开配对绑定*/
8.  #endif
9.  #endif//#if CONFIG_BT_GATT_CLIENT_NUM
```

指定服务搜索，参考文件 ble_trans_search.c

搜索苹果服务 ANCS 和 AMS 参考文件 ble_trans_search.c

2.2.6 BLE 通信提速配置

提供配置支持设备大数据传输提速，主要是支持 PDU 长包 DLE 属性（蓝牙 V4.2 版本以上，LE Data Length Extension）和 2M 包属性（蓝牙 V5.0 版本以上，LE 2M PHY）编码。需要对方也支持对应的 Feature 属性，传输提速才有效果。

```
1.  #define CONFIG_BLE_HIGH_SPEED               0 //BLE 提速模式: 使能
    DLE+2M, payload 要匹配 pdu 的包长
```

2.2.7 BLE 优先级高的 ATT 发送缓存支持

基于有需求在现有的 ATT 发送缓存的基础上，再添加新的优先级高的 ATT 发送缓存通道的使用，

SDK 已添加了对应创建新的数据通道接口（ble_api.h），如下：

```
/*********************************************************************************************/
/*!
 *  \brief      API: 配置 ATT,第二套缓存发送模块初始化(优先级比默认缓存高).
 *
 *  \function   ble_cmd_ret_e ble_op_high_att_send_init(u8 *att_ram_addr,int att_ram_size).
 *
 *  \param      [in] att_ram_addr       传入 ATT 发送模块 ram 地址，地址按 4 字节对齐.
 *  \param      [in] att_ram_size       传入 ATT 发送模块 ram 大小.
 *
 *  \return     see ble_cmd_ret_e.
 *
 *  \note       必须在原有的缓存初始化使用后，才能再初始化新的缓存使用.
 */
/*********************************************************************************************/
/* ble_cmd_ret_e ble_op_high_att_send_init(u8 *att_ram_addr,int att_ram_size) */
#define ble_op_high_att_send_init(att_ram_addr,att_ram_size)     \
 ble_user_cmd_prepare(BLE_CMD_HIGH_ATT_SEND_INIT, 2, att_ram_addr,att_ram_size)


/*********************************************************************************************/
/*!
 *  \brief      API: ATT 操作 high 缓存 handle 发送数据.
 *
 *  \function   ble_cmd_ret_e ble_op_high_att_send_data(u16 con_handle,u16 att_handle,u8 *data,u16 len, att_op_type_e att_op_type).
 *
 *  \param      [in] con_handle     连接 con_handle,range: >0.
 *  \param      [in] att_handle     att 操作 handle.
 *  \param      [in] data           数据地址.
 *  \param      [in] len            数据长度  <= cbuffer 可写入的长度.
 *  \param      [in] att_op_type    see  att_op_type_e (att.h).
 *
```

```
 * \return     see ble_cmd_ret_e.
 *
 * \note
 */
/*****************************************************************************************/
/* ble_cmd_ret_e ble_op_high_att_send_data(u16 con_handle,u16 att_handle,u8 *data,u16 len, att_op_type_e att_op_type) */
#define ble_op_high_att_send_data(con_handle,att_handle,data,len,att_op_type)     \
 ble_user_cmd_prepare(BLE_CMD_HIGH_ATT_SEND_DATA, 5, con_handle,att_handle, data, len, att_op_type)

/*****************************************************************************************/
/*!
 * \brief      ATT 操作,清 high 缓存发送的数据缓存
 *
 * \function   ble_op_high_att_clear_send_data(void).
 *
 * \return     see ble_cmd_ret_e.
 */
/*****************************************************************************************/
/* ble_cmd_ret_e ble_op_high_att_clear_send_data(void) */
#define ble_op_high_att_clear_send_data(void)     \
    ble_user_cmd_prepare(BLE_CMD_ATT_CLEAR_SEND_DATA, 1, 2)


/*****************************************************************************************/
/*!
 * \brief      API: 获取 ATT 发送模块,high 缓存 cbuffer 可写入数据的长度.
 *
 * \function   ble_cmd_ret_e ble_op_high_att_get_remain(u16 con_handle,int *remain_size_ptr).
 *
 * \param      [in]  con_handle         range: >0.
 * \param      [out] remain_size_ptr    输出可写入长度值.
 *
```

```
64.  *  \return      see ble_cmd_ret_e.
65.  */
66. /*************************************************************************************************/
67. /* ble_cmd_ret_e ble_op_high_att_get_remain(u16 con_handle,int *remain_size_ptr) */
68. #define ble_op_high_att_get_remain(con_handle,remain_size_ptr)     \
69.  ble_user_cmd_prepare(BLE_CMD_HIGH_ATT_VAILD_LEN, 2,con_handle, remain_size_ptr)
```

2.3 APP - Bluetooth Dual-Mode HID

2.3.1 概述

标准的蓝牙鼠标，支持蓝牙 CLASSIC，蓝牙 BLE 和 2.4G 模式。

蓝牙鼠标支持 windows 系统，mac 系统、安卓系统、ios 系统连接。

支持的板级：bd29、br25、br30、bd19

支持的芯片：AC631N、AC6363F、AC6369F、AC6379B、AC6379B

2.3.2 工程配置

代码工程：apps\hid\board\bd29\AC631N_hid.cbp

1、配置描述

（1）先配置板级 board_config.h 和对应配置文件中蓝牙双模使能

```
/*
 *  板级配置选择
 */
#define CONFIG_BOARD_AC631N_DEMO        // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER

#include "board_ac631n_demo_cfg.h"
```

//配置是否打开 edr 和 ble 模块

```
#define TCFG_USER_BLE_ENABLE            1   //BLE 或 2.4G 功能使能
#define TCFG_USER_EDR_ENABLE            1   //EDR 功能使能
```

（2）配置 app 选择

```
///app case 选择,只选 1,要配置对应的 board_config.h
#define CONFIG_APP_MOUSE_SINGLE                 1//单模切换#de
```

蓝牙的 BLE 配置，保护 GATT 和 SM 的配置

```
//蓝牙 BLE 配置
#define CONFIG_BT_GATT_COMMON_ENABLE        1 //配置使用 gatt 公共模块
#define CONFIG_BT_SM_SUPPORT_ENABLE         1 //配置是否支持加密
#define CONFIG_BT_GATT_CLIENT_NUM           0 //配置主机 client 个数 (app not support)
#define CONFIG_BT_GATT_SERVER_NUM           1 //配置从机 server 个数
```

```
#define CONFIG_BT_GATT_CONNECTION_NUM       (CONFIG_BT_GATT_SERVER_NUM + CONFIG_BT_GATT_CLIENT_NUM) //配置连接个数
```

（3）模式选择， 配置 BLE 或 2.4G 模式；若选择 2.4G 配对码必须跟对方的配对码一致

```
//2.4G 模式: 0---ble, 非 0---2.4G 配对码
#define CFG_RF_24G_CODE_ID          (0) //<=24bits
```

（4）支持双模 HID 设备切换处理

```
static void app_select_btmode(u8 mode)
{
    if (mode != HID_MODE_INIT) {
        if (bt_hid_mode == mode) {
            return;
        }
        bt_hid_mode = mode;
    } else {
        //init start
    }
```

（5）支持进入省电低功耗 Sleep

```
//*********************************************************************************//
//                          低功耗配置                                             //
//*********************************************************************************//
// #define TCFG_LOWPOWER_POWER_SEL          PWR_DCDC15//
#define TCFG_LOWPOWER_POWER_SEL          PWR_LDO15//
#define TCFG_LOWPOWER_BTOSC_DISABLE      0
#define TCFG_LOWPOWER_LOWPOWER_SEL       SLEEP_EN
#define TCFG_LOWPOWER_VDDIOM_LEVEL       VDDIOM_VOL_30V
#define TCFG_LOWPOWER_VDDIOW_LEVEL       VDDIOW_VOL_24V
#define TCFG_LOWPOWER_OSC_TYPE           OSC_TYPE_LRC
```

（6）支持进入软关机，可用 IO 触发唤醒

```
struct port_wakeup port0 = {
```

```
    .pullup_down_enable = ENABLE,              //配置 I/O 内部上下拉是否使能
    .edge               = FALLING_EDGE,        //唤醒方式选择,可选：上升沿\下降沿
    .attribute          = BLUETOOTH_RESUME,    //保留参数
    .iomap              = IO_PORTB_01,         //唤醒口选择
    .filter_enable      = ENABLE,
};
```

```
static void hid_set_soft_poweroff(void)
{
    log_info("hid_set_soft_poweroff\n");
    is_hid_active = 1;
```

（7）系统事件处理函数

```
static int event_handler(struct application *app, struct sys_event *event)
{
```

（8）蓝牙事件处理函数

```
static int bt_hci_event_handler(struct bt_event *bt)
{
```

（9）增加配对绑定管理

```
//2.4 开配对管理,可自己修改按键方式
#define DOUBLE_KEY_HOLD_PAIR  (1 & CFG_RF_24G_CODE_ID)  //中键+右键 长按数秒,进入 2.4G 配对模式
#define DOUBLE_KEY_HOLD_CNT       (4)  //长按中键计算次数 >= 4s
```

（10）进出低功耗函数

```
void sleep_enter_callback(u8  step)
{
    /* 此函数禁止添加打印 */
    if (step == 1) {
        putchar('<');
        APP_IO_DEBUG_1(A, 5);
```

```
        /*dac_power_off();*/
    } else {
        close_gpio();
    }
}

void sleep_exit_callback(u32 usec)
{
    putchar('>');
    APP_IO_DEBUG_0(A, 5);
}
```

可以用来统计低功耗 sleep 的时间。

2.3.3 目录结构

以鼠标 APP_MOUSE 为例子，SDK 的目录结构如图 1.2 所示。

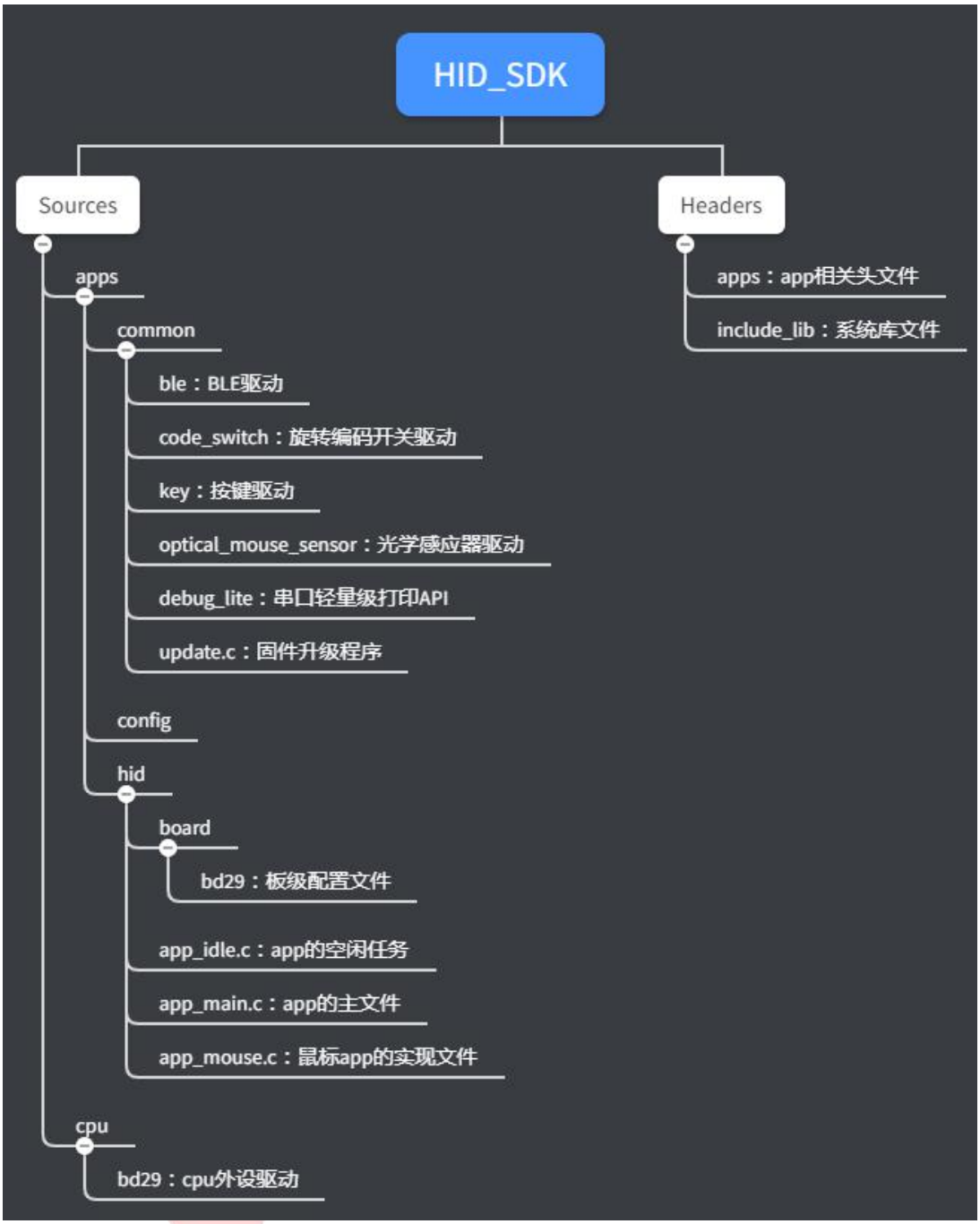


图 1.1 HID_SDK 目录结构

2.3.3 板级配置

2.3.3.1 板级方案配置

为提高开发过程的灵活性，HID_SDK 为用户提供几种不同的 CUP 配置和对应的板级方案，用户可根据具体的开发需求选择相应的方案。SDK 在上一版的基础上基于 BR23 增加标准键盘应用。

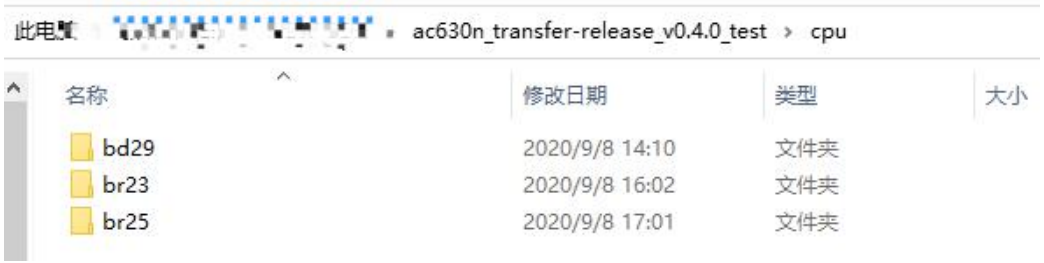


图 1.2 CPU 配置

板级方案配置文件的路径：apps/hid/board/BD29/board_config.h(hid 可替换为相应的 app 名称)。用户只需在板级方案配置文件 board_config.h 添加相应的宏定义，并包含相应的头文件，即可完成板级方案的配置。配置示例如图所示。

```
1.  /*
2.   *  板级配置选择
3.   */
4.  #define CONFIG_BOARD_AC631N_DEMO        // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER
5.
6.  #include "board_ac631n_demo_cfg.h"
```

图 1.3 板级方案配置示例

2.3.3.2 板级配置文件

板级配置文件的作用是实现相同系列不同封装的配置方案，其存放路径为：apps/hid/board/BD29(hid 替换为相应的 app 名称)。板级配置文件对应一个 C 文件和一个 H 文件。

（1）H 文件

板级配置的 H 文件包含了所有板载设备的配置信息，方便用户对具体的设备配置信息进行修改。

（2）C 文件

板级配置的 C 文件的作用是根据 H 文件包含的板载配置信息，对板载设备进行初始化。

2.3.3.3 板级初始化

系统将调用 C 文件中的 board_init()函数对板载设备进行初始化。板级初始化流程如图 1.3 所示。用户可以根据开发需求在 board_devices_init()函数中添加板载设备的初始化函数。

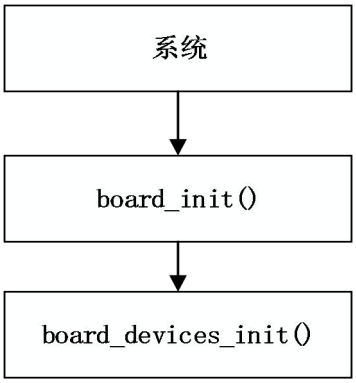


图 1.4 板级初始化流程

2.3.4 APP 开发框架

2.3.4.1 APP 总体框架

HID_SDK 为用户提供一种基于事件处理机制的 APP 开发框架，用户只需基于该框架添加需要处理的事件及事件处理函数，即可按照应用需求完成相应的开发。APP 总体框架如图 1.4 所示。

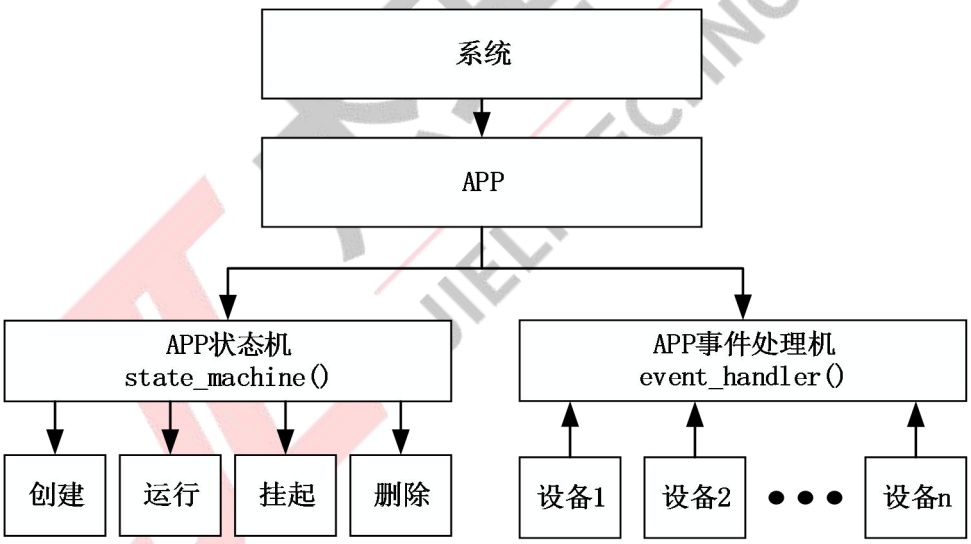


图 1.5 APP 总体框架

（1）APP 状态机

系统在运行过程中，可以通过 APP 状态机对其状态进行切换，其状态包括创建、运行、挂起、删除。

（2）APP 事件处理机

APP 是基于事件处理机制来运行的。系统在运行过程中，硬件设备的产生的数据将会以事件的形式反馈至系统的全局事件列表，系统将调度 APP 的事件处理机运行相应的事件处理函数对其进行处理。APP 的事件处理机的实现函数 apps/hid/app_mouse.c->event_handler()。

2.3.4.2 事件与事件的处理

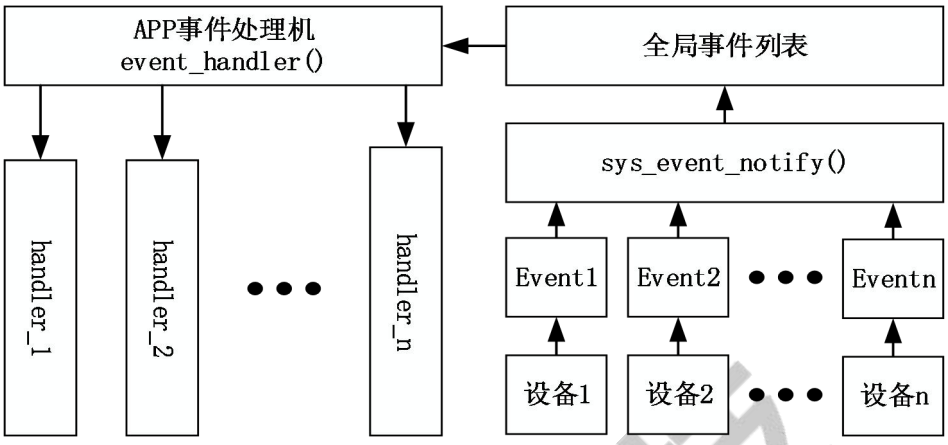


图 1.6 事件的处理过程

（1）事件的定义

HID_SDK 在 include_lib/system/event.h 中定义了事件的基本类型，如图 1.6 所示。用户可以在此基础上，根据应用需求添加自定义事件类型。

```
1.  struct sys_event {
2.      u16 type;
3.      u8 consumed;
4.      void *arg;
5.      union {
6.          struct key_event key;
7.          struct axis_event axis;
8.          struct codesw_event codesw;
```

图 1.7 事件的定义

（2）事件的产生

硬件设备产生的数据将会被打包为事件，HID_SDK 提供了一个用于发送事件的 API，用户可在 APP 中调用该 API 向全局事件列表发送事件，该 API 的使用说明如表 1-1 所示。

表 1-1 事件发送 API

函数原型	void sys_event_notify(struct sys_event *event)

功能	向全局事件列表发送数据
参数	struct sys_event *event：需要发送的事件
返回值	无

（3）事件的处理

系统通过调用 APP 的事件处理机，即 apps/hid/app_mouse.c->event_handler()函数，对 APP 中发生的事件进行处理。该函数采用 switch 选择结构，用户在不同的事件 case 下添加相应的事件处理函数即可。mouse_event_handler()函数的使用示例如图 1.7 所示。

```
1.  static int mouse_event_handler(struct application *app, struct sys_event *event)
2.  {
3.  #if (TCFG_HID_AUTO_SHUTDOWN_TIME)
4.      //重置无操作定时计数
5.      sys_timer_modify(g_auto_shutdown_timer, TCFG_HID_AUTO_SHUTDOWN_TIME * 1000);
6.  #endif
7.
8.      bt_sniff_ready_clean();
9.
10.     /* log_info("event: %s", event->arg); */
11.     switch (event->type) {
12.     case SYS_KEY_EVENT:
13.         /* log_info("Sys Key : %s", event->arg); */
14.         app_key_event_handler(event);
15.         return 0;
16.
17.     case SYS_BT_EVENT:
18.         if ((u32)event->arg == SYS_BT_EVENT_TYPE_CON_STATUS) {
19.             bt_connction_status_event_handler(&event->u.bt);
20.         } else if ((u32)event->arg == SYS_BT_EVENT_TYPE_HCI_STATUS) {
21.             bt_hci_event_handler(&event->u.bt);
22.         }
23.         return 0;
```

图 1.8 event_handler()函数的使用示例

2.3.5 按键的使用

2.3.5.1 IOKEY 的使用

（1）配置说明

IOKEY 参数在板级配置文件中（C 文件和 H 文件）进行配置，在 H 文件中可以打开 TCFG_IOKEY_ENABLE 宏和结构配置（IO 口和按键连接方式）相关参数，配置结构体参数说明如表 2-1 所示。

表 2-1 IOKEY 配置结构体说明

```
struct iokey_port {
    union key_type key_type;
    u8 connect_way;
    u8 key_value;
};
```

参数名	描述
key_type	key_type 结构依据 connect_way 而定，如果 connect_way 配置为 ONE_PORT_TO_LOW 和 ONE_PORT_TO_HIGH 只需要配置一个 IO；而如果 connect_way 配置为 DOUBLE_PORT_TO_IO，需要配置两个 IO；
connect_way	连接方式支持三种： ➢ ONE_PORT_TO_LOW：按键一个端口接地， 另一个端口接 IO； ➢ ONE_PORT_TO_HIGH：按键一个端口接 3.3v， 另一个端口接 IO； ➢ DOUBLE_PORT_TO_IO：按键两个端口接 IO，互推方式。
key_value	key_value 指的是按键值，按键被检测到被按下将发送该按键配置的 key_value 的值。

（2）配置示例

IOKEY 参数在板级配置文件中（c 文件和 h 文件）进行配置，配置示例如表 2-2 所示。

表 2-2 IOKEY 配置示例

H 文件配置

```
//iokey 配置
#define TCFG_IOKEY_ENABLE                ENABLE_THIS_MOUDLE

#define TCFG_IOKEY_POWER_ENABLE          ENABLE_THIS_MOUDLE
#define TCFG_IOKEY_POWER_CONNECT_WAY     ONE_PORT_TO_LOW  //按键一端接低电平一端接 IO
#define TCFG_IOKEY_POWER_ONE_PORT        IO_PORTB_01
```

```
#define TCFG_IOKEY_POWER_IN_PORT        NO_CONFIG_PORT
#define TCFG_IOKEY_POWER_OUT_PORT       NO_CONFIG_PORT

#define TCFG_IOKEY_PREV_ENABLE          ENABLE_THIS_MOUDLE
#define TCFG_IOKEY_PREV_CONNECT_WAY     ONE_PORT_TO_LOW  //按键一端接低电平一端接 IO
#define TCFG_IOKEY_PREV_ONE_PORT        IO_PORTB_00
#define TCFG_IOKEY_PREV_IN_PORT         NO_CONFIG_PORT
#define TCFG_IOKEY_PREV_OUT_PORT        NO_CONFIG_PORT

#define TCFG_IOKEY_NEXT_ENABLE          ENABLE_THIS_MOUDLE
#define TCFG_IOKEY_NEXT_CONNECT_WAY     ONE_PORT_TO_LOW  //按键一端接低电平一端接 IO
#define TCFG_IOKEY_NEXT_ONE_PORT        IO_PORTB_02
#define TCFG_IOKEY_NEXT_IN_PORT         NO_CONFIG_PORT
#define TCFG_IOKEY_NEXT_OUT_PORT        NO_CONFIG_PORT
```

C 文件配置

```
#if TCFG_IOKEY_ENABLE
const struct iokey_port iokey_list[] = {
    {
        .connect_way = ONE_PORT_TO_LOW,             //IO 按键的连接方式
        .key_type.one_io.port = TCFG_IOKEY_MOUSE_LK_PORT,  //IO 按键对应的引脚
        .key_value = KEY_LK_VAL,                    //按键值
    },

    {
        .connect_way = ONE_PORT_TO_LOW,             //IO 按键的连接方式
        .key_type.one_io.port = TCFG_IOKEY_MOUSE_RK_PORT,  //IO 按键对应的引脚
        .key_value = KEY_RK_VAL,                    //按键值
    },

    {
        .connect_way = ONE_PORT_TO_LOW,             //IO 按键的连接方式
```

```
17.     .key_type.one_io.port = TCFG_IOKEY_MOUSE_HK_PORT,  //IO 按键对应的引脚
18.     .key_value = KEY_HK_VAL,                           //按键值
19.  },
```

2.3.5.2 ADKEY 的使用

（1）配置说明

ADKEY 参数在板级配置文件中（c 文件和 h 文件）进行配置，如 board_ac6xxx_mouse.c 和 board_ac6xxx_mouse_cfg.h，在 h 文件中可以打开 TCFG_ADKEY_ENABLE 宏和结构配置（IO 口和按键连接方式）相关参数，配置结构体参数说明如表 2-3 所示。

表 2-3 ADKEY 配置结构体说明

```
1.  struct adkey_platform_data {
2.      u8 enable;
3.      u8 adkey_pin;
4.      u8 extern_up_en; //是否用外部上拉，1：用外部上拉， 0：用内部上拉 10K
5.      u32 ad_channel;
6.      u16 ad_value[ADKEY_MAX_NUM];
7.      u8  key_value[ADKEY_MAX_NUM];
8.  };
```

参数名	描述
enable	ADKEY 的使能选择，1 表示使能，0 表示失能
adkey_pin	AD 模块的 IO 编号
ad_channel	AD 通道编号
extern_up_en	是否使用外部上拉，1 表示使用外部上拉，0 表示使用内部上拉（10K）
ad_value[ADKEY_MAX_NUM]	按键对应的 AD 值
key_value[ADKEY_MAX_NUM]	键值，按键被检测到被按下将发送该按键配置的 key_value 的值。

（2）配置示例

ADKEY 参数在板级配置文件中（c 文件和 h 文件）进行配置，配置示例如表 2-4 所示。

表 2-4 ADKEY 配置示例

H 文件

```
1.  #define TCFG_ADKEY_ENABLE          DISABLE_THIS_MOUDLE //是否使能 AD 按键
2.  #define TCFG_ADKEY_PORT            IO_PORTB_01         //AD 按键端口(需要注意选择的 IO 口是否支持 AD 功能)
```

配置

```
3.  /*AD 通道选择，需要和 AD 按键的端口相对应:
4.      AD_CH_PA1   AD_CH_PA3   AD_CH_PA4   AD_CH_PA5
5.      AD_CH_PA9   AD_CH_PA1   AD_CH_PB1   AD_CH_PB4
6.      AD_CH_PB6   AD_CH_PB7   AD_CH_DP    AD_CH_DM
7.      AD_CH_PB2
8.  */
9.  #define TCFG_ADKEY_AD_CHANNEL           AD_CH_PB1
#define TCFG_ADKEY_EXTERN_UP_ENABLE     ENABLE_THIS_MOUDLE //是否使用外部上拉
```

C 文件配置

```
1.  #if TCFG_ADKEY_ENABLE
2.  const struct adkey_platform_data adkey_data = {
3.      .enable = TCFG_ADKEY_ENABLE,            //AD 按键使能
4.      .adkey_pin = TCFG_ADKEY_PORT,           //AD 按键对应引脚
5.      .ad_channel = TCFG_ADKEY_AD_CHANNEL,    //AD 通道值
6.      .extern_up_en = TCFG_ADKEY_EXTERN_UP_ENABLE, //是否使用外接上拉电阻
7.      .ad_value = {                   //根据电阻算出来的电压值
8.          TCFG_ADKEY_VOLTAGE0,
9.          TCFG_ADKEY_VOLTAGE1,
10.         TCFG_ADKEY_VOLTAGE2,
11.         TCFG_ADKEY_VOLTAGE3,
12.         TCFG_ADKEY_VOLTAGE4,
13.         TCFG_ADKEY_VOLTAGE5,
14.         TCFG_ADKEY_VOLTAGE6,
15.         TCFG_ADKEY_VOLTAGE7,
16.         TCFG_ADKEY_VOLTAGE8,
17.         TCFG_ADKEY_VOLTAGE9,
18.     },
19.     .key_value = {          //AD 按键各个按键的键值
20.         TCFG_ADKEY_VALUE0,
21.         TCFG_ADKEY_VALUE1,
22.         TCFG_ADKEY_VALUE2,
```

2.3.5.3 按键扫描参数配置

在 IOKEY 或者 ADKEY 使能后，按键扫描代码就会注册定时器定时扫描按键是否被按下，按键扫描参数可以在文件 apps/common/key/iokey.c 或 adkey.c 中配置，可供配置的参数表 2-5 所示。

表 2-5 按键扫描参数配置说明

```
1.  //按键驱动扫描参数列表
2.  struct key_driver_para adkey_scan_para = {
3.      .scan_time      = 10,    //按键扫描频率, 单位: ms
4.      .last_key       = NO_KEY, //上一次 get_value 按键值, 初始化为 NO_KEY;
5.      .filter_time    = 2,     //按键消抖延时;
6.      .long_time      = 75,    //按键判定长按数量
7.      .hold_time      = (75 + 15), //按键判定 HOLD 数量
8.      .click_delay_time = 20,   //按键被抬起后等待连击延时数量
9.      .key_type       = KEY_DRIVER_TYPE_AD,
10.     .get_value      = ad_get_key_value,
11. };
```

参数名	描述
scan_time	按键扫描频率，单位 ms，定时器将会按照设定的时间定时扫描 IOKEY 或者 ADKEY
last_key	默认初始化为 NO_KEY
filter_time	按键消抖时间，计算方式：fileter_time * scan_time (ms)
long_time	按键长按事件判定时间，计算方式：long_time * scan_time (ms)
hold_time	按键按住保持事件判定时间，计算方式：hold_time * scan_time (ms)
click_delay_time	按键等待连击操作延时时间，计算方式：hold_time * scan_time (ms)，注意该参数配置会影响按键灵敏度，同时也会影响连击操作的时间间隔，所以在调试过程中需要根据需要选择一个合适的参数值；

2.3.5.4 按键事件处理

目前在 HID_SDK 中实现了的一些按键通用事件如表 2-6 所示。

表 2-6 按键的通用事件表

按键事件	说明
KEY_EVENT_CLICK	单击事件，在按键被按下经过 filter_time 时间后松开并经过 click_delay_time

	时间后如果没有被第二次按下，按键扫描函数会判定为按键单击事件并发布出去。
KEY_EVENT_LONG	长按事件，当按键被按下经过 filter_time 时间后并一直被按下，在经过 long_time 时间后按键扫描函数会判定为按键长按事件并发布出去。
KEY_EVENT_HOLD	按下保持事件，当按键被按下经过 filter_time 时间后并一直被按下，在经过 hold_time 时间后按键扫描函数会判定为按键按下保持事件并发布出去，发布完之后如果发现按键还被按下，会在经过 hold_time - long_time 时间后再次发布按下保持事件。
KEY_EVENT_UP	抬按事件，在发送完长按事件（KEY_EVENT_LONG）和按下保持事件（KEY_EVENT_HOLD）后，如果按键被松开，按键扫描函数会发布一个抬按事件。
KEY_EVENT_DOUBLE_CLICK	双击事件，在按键被按下经过 filter_time 时间后松开并在 click_delay_time 之前同一按键再次被按下，并经过 click_delay_time 之后按键没有再次被按下，按键扫描函数会发布一个双击事件。
KEY_EVENT_TRIPLE_CLICK	三击事件，在按键被按下经过 filter_time 时间后松开并在 click_delay_time 之前同一按键再次被按下，并经过 click_delay_time 之后按键再次被按下，如果往后操作重复上述连击操作，按键扫描函数都会判定为三击事件并发布出去。

按键发布消息后，在 APP 将会收到该消息，APP 可以根据该按键消息进行相关处理，APP 的 event_handler 收到的按键消息数据格式如表 2-7 所示。用户可以根据收到的按键消息进行相关处理操作。

表 2-7 按键消息数据格式

数据格式	说明
event->type	按键消息类型为 SYS_KEY_EVENT，标记为按键消息。
event->u.key.value	按键值，该值在配置 IOKEY 或者 ADKEY 中配置，标记某一个按键被按下。
event->u.key.event	按键事件类型，对应上表中所列举的按键事件。

2.3.5.5 按键拓展功能

HID_SDK 提供了一些通用按键配置和消息处理方式，如果这些通用的机制还不能满足用户的需求，用户可以通过修改配置使用按键的拓展功能。

（1）组合键功能

HID_SDK 的 IOKEY 中默认只支持单个按键的检测，用户如果需要支持组合按键，可以通过修改 IOKEY 的配置项来实现，具体实现如下：

1）在配置文件的 H 文件中打开 MULT_KEY_ENABLE 宏，并添加组合键值；

2）在配置文件的 C 文件中配置按键的重映射数据结构。

配置示例如表 2-8 所示。

表 2-8 IOKEY 的组合键配置示例

H 文件配置

```
#define MOUSE_KEY_SCAN_MODE          ENABLE_THIS_MOUDLE
#define MULT_KEY_ENABLE              ENABLE

#define KEY_LK_VAL          BIT(0)
#define KEY_RK_VAL          BIT(1)
#define KEY_HK_VAL          BIT(2)
#define KEY_LK_RK_VAL       BIT(0) | BIT(1)
#define KEY_LK_HK_VAL       BIT(0) | BIT(2)
#define KEY_RK_HK_VAL       BIT(1) | BIT(2)
#define KEY_LK_RK_HK_VAL    BIT(0) |BIT(1) | BIT(2)

#define TCFG_IOKEY_MOUSE_LK_PORT        IO_PORTB_03
#define TCFG_IOKEY_MOUSE_RK_PORT        IO_PORTB_02
#define TCFG_IOKEY_MOUSE_HK_PORT        IO_PORTB_01
#define TCFG_IOKEY_MOUSE_MK_IN_PORT     NO_CONFIG_PORT
#define TCFG_IOKEY_MOUSE_MK_OUT_PORT    NO_CONFIG_PORT
```

C 文件配置

```
const struct key_remap key_remap_table[] = {
    {
        .bit_value = BIT(KEY_LK_VAL) | BIT(KEY_RK_VAL),
        .remap_value = KEY_LK_RK_VAL,
    },

    {
        .bit_value = BIT(KEY_LK_VAL) | BIT(KEY_HK_VAL),
        .remap_value = KEY_LK_HK_VAL,
    },

```

```
    {
        .bit_value = BIT(KEY_RK_VAL) | BIT(KEY_HK_VAL),
        .remap_value = KEY_RK_HK_VAL,
    },

    {
        .bit_value = BIT(KEY_LK_VAL) | BIT(KEY_RK_VAL) | BIT(KEY_HK_VAL),
        .remap_value = KEY_LK_RK_HK_VAL,
    },
};

const struct key_remap_data iokey_remap_data = {
    .remap_num = ARRAY_SIZE(key_remap_table),
    .table = key_remap_table,
};
```

（2）按键多击事件

HID_SDK 中默认支持单击、双击事件和三击事件，用户如果需要支持更多击事件，可以修改如下文件：

1）在 event.h 文件中添加新的按键事件类型定义，如图 2.1 所示，在箭头位置可以添加其他多击事件。

```
enum {
    KEY_EVENT_CLICK,
    KEY_EVENT_LONG,
    KEY_EVENT_HOLD,
    KEY_EVENT_UP,
    KEY_EVENT_DOUBLE_CLICK,
    KEY_EVENT_TRIPLE_CLICK,
    KEY_EVENT_FOURTH_CLICK,
    KEY_EVENT_FIRTH_CLICK,
    KEY_EVENT_USER,
    KEY_EVENT_MAX,
};
```

图 2.2.1 按键事件类型定义

2）key_driver.c 中的 key_driver_scan()函数中添加任意多击事件的判断，添加位置如图 2.2 所示。

```
if (scan_para->click_delay_cnt > scan_para->click_delay_time)  //按键被抬起后延时到
{
    //TODO: 在此可以添加任意多击事件
    if (scan_para->click_cnt >= 5)
    {
        key_event = KEY_EVENT_FIRTH_CLICK;  //五击
    }
    else if (scan_para->click_cnt >= 4)
    {
        key_event = KEY_EVENT_FOURTH_CLICK;  //4 击
    }
    else if (scan_para->click_cnt >= 3)
    {
        key_event = KEY_EVENT_TRIPLE_CLICK;  //三击
    }
    else if (scan_para->click_cnt >= 2)
    {
        key_event = KEY_EVENT_DOUBLE_CLICK;  //双击
    }
    else
    {
        key_event = KEY_EVENT_CLICK;  //单击
    }
    key_value = scan_para->notify_value;
    goto _notify;
}
else   //按键抬起后等待下次延时时间未到
{
    scan_para->click_delay_cnt++;
    goto _scan_end; //按键抬起后延时时间未到, 返回
```

```
31. }
```

图 2.2.2 添加多击事件判断示例图

3）最后在 APP 层收到该多击事件进行处理即可。

（3）某些按键只响应单击事件

该功能可以通过按键值的某一 bit 进行特殊处理，由于按键值（key_value）目前用 1byte 来表示，可支持 0～255 个按键，但在大部分应用中用不到这么多按键，因此目前 HID_SDK 中使用按键值（key_value）的第 7 位进行标记，按键扫描时对标记了 bit(7)的按键不进行多击判断处理，用户如果需要应用该功能，只需要在板级配置文件中对按键值进行标记即可，如图 2.3 所示。

```
1.  {
2.      .connect_way = ONE_PORT_TO_LOW,              //IO 按键的连接方式
3.      .key_type.one_io.port = TCFG_IOKEY_MOUSE_LK_PORT,  //IO 按键对应的引脚
4.      .key_value = KEY_LK_VAL,                     //按键值
5.  },
```

图 2.2.3 添加多击事件判断示例图

2.3.6 串口的使用

串口的初始化参数在板级配置文件中（c 文件和 h 文件）进行配置，如 board_ac6xxx_mouse.c 和 board_ac6xxx_mouse_cfg.h，在 h 文件中使能 TCFG_UART0_ENABLE 宏和结构配置相关参数，在 C 文件中添加初始化数据结构，配置示例如表 2-9 所示。串口初始化完成后，用户可调用 apps/debug.c 文件中的函数进行串口打印操作。

表 2-9 串口配置示例

H 文件配置

```
1.  //*********************************************************************//
2.  //                      UART 配置                                      //
3.  //*********************************************************************//
4.  #define TCFG_UART0_ENABLE                ENABLE_THIS_MOUDLE
5.  #define TCFG_UART0_RX_PORT               NO_CONFIG_PORT
6.  #define TCFG_UART0_TX_PORT               IO_PORT_DP//IO_PORTA_05
7.  #define TCFG_UART0_BAUDRATE              1000000
```

C 文

```
1.  /************************ UART config****************************/
2.  #if TCFG_UART0_ENABLE
```

件配置

```
3.  UART0_PLATFORM_DATA_BEGIN(uart0_data)
4.  .tx_pin = TCFG_UART0_TX_PORT,                //串口打印 TX 引脚选择
5.  .rx_pin = TCFG_UART0_RX_PORT,                //串口打印 RX 引脚选择
6.  .baudrate = TCFG_UART0_BAUDRATE,             //串口波特率
7.  .flags = UART_DEBUG,                         //串口用来打印需要把改参数设置为 UART_DEBUG
8.  UART0_PLATFORM_DATA_END()
9.  #endif //TCFG_UART0_ENABLE
```

2.3.7 Mouse Report Map

Mouse Report Map 定义与 apps/common/ble/le_hogp.c 文件内，如图 2.3.1 所示。

```
1.  0x05, 0x01,     // Usage Page (Generic Desktop Ctrls)
2.  0x09, 0x02,     // Usage (Mouse)
3.  0xA1, 0x01,     // Collection (Application)
4.  0x85, 0x01,     //  Report ID (1)
5.  0x09, 0x01,     //  Usage (Pointer)
6.  0xA1, 0x00,     //  Collection (Physical)
7.  0x95, 0x05,     //  Report Count (5)
8.  0x75, 0x01,     //  Report Size (1)
9.  0x05, 0x09,     //  Usage Page (Button)
10. 0x19, 0x01,     //  Usage Minimum (0x01)
11. 0x29, 0x05,     //  Usage Maximum (0x05)
12. 0x15, 0x00,     //  Logical Minimum (0)
13. 0x25, 0x01,     //  Logical Maximum (1)
14. 0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
15. 0x95, 0x01,     //  Report Count (1)
16. 0x75, 0x03,     //  Report Size (3)
17. 0x81, 0x01,     //  Input (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
18. 0x75, 0x08,     //  Report Size (8)
19. 0x95, 0x01,     //  Report Count (1)
20. 0x05, 0x01,     //  Usage Page (Generic Desktop Ctrls)
21. 0x09, 0x38,     //  Usage (Wheel)
22. 0x15, 0x81,     //  Logical Minimum (-127)
```

```
0x25, 0x7F,       //    Logical Maximum (127)
0x81, 0x06,       //    Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x05, 0x0C,       //    Usage Page (Consumer)
0x0A, 0x38, 0x02, //    Usage (AC Pan)
0x95, 0x01,       //    Report Count (1)
0x81, 0x06,       //    Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0xC0,             //  End Collection
0x85, 0x02,       //  Report ID (2)
0x09, 0x01,       //  Usage (Consumer Control)
0xA1, 0x00,       //  Collection (Physical)
0x75, 0x0C,       //    Report Size (12)
0x95, 0x02,       //    Report Count (2)
0x05, 0x01,       //    Usage Page (Generic Desktop Ctrls)
0x09, 0x30,       //    Usage (X)
0x09, 0x31,       //    Usage (Y)
0x16, 0x01, 0xF8, //    Logical Minimum (-2047)
0x26, 0xFF, 0x07, //    Logical Maximum (2047)
0x81, 0x06,       //    Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0xC0,             //  End Collection
0xC0,             // End Collection
0x05, 0x0C,       // Usage Page (Consumer)
0x09, 0x01,       // Usage (Consumer Control)
0xA1, 0x01,       // Collection (Application)
0x85, 0x03,       //  Report ID (3)
0x15, 0x00,       //  Logical Minimum (0)
0x25, 0x01,       //  Logical Maximum (1)
0x75, 0x01,       //  Report Size (1)
0x95, 0x01,       //  Report Count (1)
0x09, 0xCD,       //  Usage (Play/Pause)
0x81, 0x06,       //  Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x0A, 0x83, 0x01, //  Usage (AL Consumer Control Configuration)
0x81, 0x06,       //  Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x09, 0xB5,       //  Usage (Scan Next Track)
```

```
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x09, 0xB6,       // Usage (Scan Previous Track)
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x09, 0xEA,       // Usage (Volume Decrement)
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x09, 0xE9,       // Usage (Volume Increment)
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x0A, 0x25, 0x02, // Usage (AC Forward)
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x0A, 0x24, 0x02, // Usage (AC Back)
0x81, 0x06,       // Input (Data,Var,Rel,No Wrap,Linear,Preferred State,No Null Position)
0x09, 0x05,       // Usage (Headphone)
0x15, 0x00,       // Logical Minimum (0)
0x26, 0xFF, 0x00, // Logical Maximum (255)
0x75, 0x08,       // Report Size (8)
0x95, 0x02,       // Report Count (2)
0xB1, 0x02,       // Feature (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position,Non-volatile)
0xC0,             // End Collection
```

图 2.3.1 Mouse Report Map

Mouse Report Map 的解析可通过在线解析工具实现，用户可根据需要对 Report Map 进行修改。

Report Map 在线解析工具地址：http://eleccelerator.com/usbdescreqparser/。

2.3.8 蓝牙鼠标 APP 总体框架

蓝牙鼠标 APP 总体框架如图 2.3.2 所示。

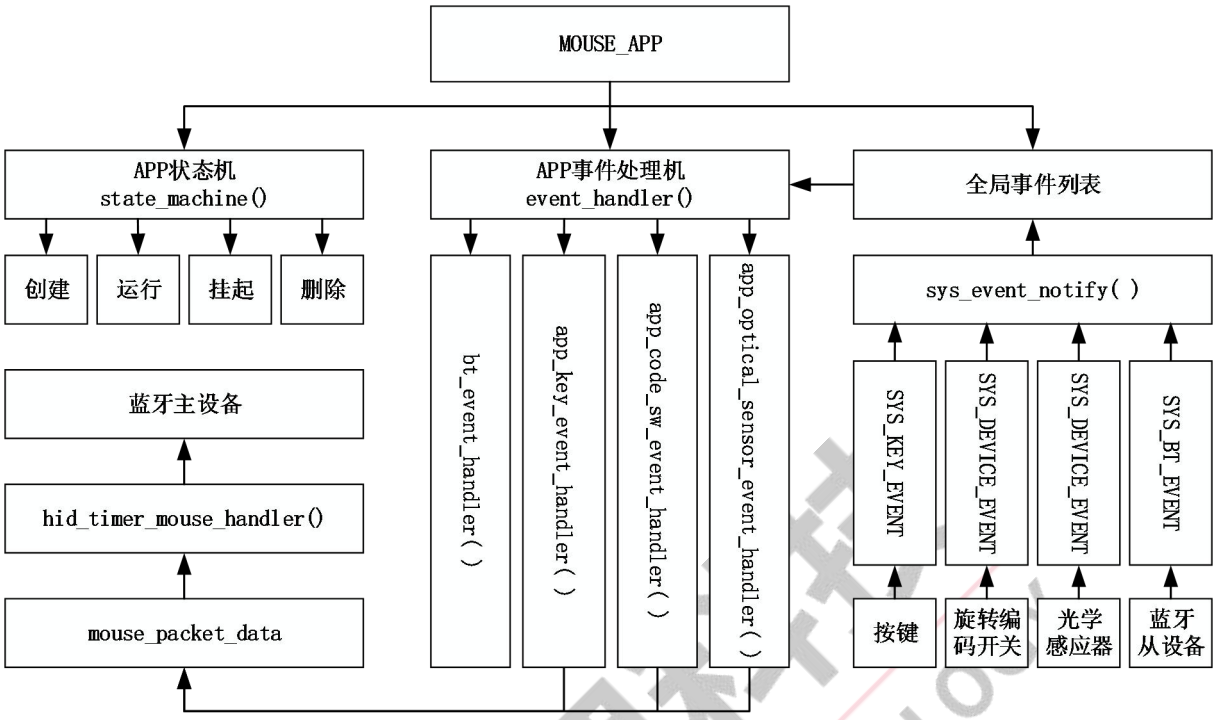


图 2.3.2 蓝牙鼠标 APP 总体框架图

（1）APP 的注册与运行

在 apps/hid/app_mouse.c 文件中包含了 APP 的注册信息，如图 3.3 所示，系统在初始化过程中将根据此信息完成该 APP 的注册。

```
/*
 * 注册 AT Module 模式
 */
REGISTER_APPLICATION(app_mouse) = {
    .name   = "mouse",
    .action = ACTION_MOUSE_MAIN,
    .ops    = &app_mouse_ops,
    .state  = APP_STA_DESTROY,
};
```

图 2.3.3 APP 的注册信息

系统初始化完成后，系统将调度 app_task 任务，该任务调用 apps/hid/app_main.c->app_main()函数，开始运行 app_mouse。

（2）事件的产生

鼠标的按键、旋转编码开关、光学感应器的数据采集在系统软件定时器的中断服务函数中完成，采集的数据将被打包为相应的事件周期性地上报至全局事件列表。

（3）事件的处理

系统将调度 APP 的事件处理机（app_mouse.c->event_handler()）依据事件类型，调用相应的事件处理函数。

app_key_event_handler()、app_code_sw_event_handler()、app_optical_sensor_event_handler()位于 apps/hid/app_mouse.c 文件，分别用以处理按键事件、旋转编码开关事件、光学感应器事件，其事件包含的数据将被填入 mouse_packet_data 中保存。

（4）数据的发送

蓝牙设备初始化时，设置了一个系统的软件定时器，用以周期性地向蓝牙主设备发送 mouse_packet_data 数据，该系统定时器的中断服务函数为：

app/common/ble/le_hogp.c->hid_timer_mouse_handler()。

2.3.9 蓝牙鼠标功耗

（1）所用光学传感器资料

PAW3205DB-TJ3T

Power Supply	**Operating voltage** **2.1V ~ 3.6V (VDD)**
Typical Operating Current (without I/O toggling)	**1.5mA @ Mouse moving (Normal)** **100uA @ Mouse not moving (Sleep1)** **15uA @ Mouse not moving (Sleep2)** **10uA @ Power down mode** *not including LED, typical value

available, normal mode and sleep1 mode. After 256 ms (typical) not moving during normal mode, the mouse sensor will enter sleep1 mode, and keep on sleep1 mode until motion detected or

After 20 sec (typical) not moving during sleep1 mode, the mouse sensor will enter sleep2 mode

（2）测试条件

1、ble 连接状态下 Interval：6*1.25 ms = 7.5ms，lantency：100；

2、Radio TX: 7.2 dBm

3、DCDC；VDDIOM 3.0V；VDDIOW 2.4V

4、VDDIO 和 VBAT 短接

（3）芯片功耗

（1、硬件不接模块；2、软件关闭所有模块）			
		无操作	软关机
2.1V	最大电流	1.338 mA	1 uA
	平均电流	139 uA	1 uA
	最小电流	30 uA	1 uA
2.6V	最大电流	852 uA	2 uA
	平均电流	110 uA	2 uA
	最小电流	33 uA	1 uA
3.0V	最大电流	965 uA	2 uA
	平均电流	110 uA	2 uA
	最小电流	33 uA	2 uA
3.3V	最大电流	813 uA	3 uA
	平均电流	113 uA	3 uA
	最小电流	35 uA	2 uA

（4）整机功耗

（1、硬件接上所有模块；2、软件使能所有模块）				
		sleep1 (无操作 256 毫秒后)	sleep2 (无操作 20.48 秒后)	软关机 (无操作 1 分钟后)
2.1V	最大电流	2.894 mA	1.253 mA	65 uA
	平均电流	230 uA	140 uA	12 uA
	最小电流	95 uA	29 uA	1 uA
2.6V	最大电流	1.008 mA	1.062 mA	63 uA
	平均电流	193 uA	125 uA	13 uA
	最小电流	111 uA	31 uA	3 uA

3.0V	最大电流	864 uA	864 uA	55 uA
	平均电流	190 uA	120 uA	13 uA
	最小电流	109 uA	37 uA	6 uA
3.3V	最大电流	858 uA	783 uA	53 uA
	平均电流	185 uA	139 uA	14 uA
	最小电流	114 uA	39 uA	7 uA

2.4 APP - Bluetooth Dual-Mode AT Moudle

2.4.1 概述

主要功能是在普通数传 BLE 和 EDR 的基础上增加了由上位机或其他 MCU 可以通过 UART 对接蓝牙芯片进行基本配置、状态获取、控制扫描、连接断开以及数据收发等操作。

AT 控制透传支持从机模式和主机模式，编译的时候只能二选一，从机模式支持双模，主机模式只支持 BLE。

定义一套串口的控制协议，具体请查看协议文档《蓝牙 AT 协议》。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.4.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631N_spp_and_le.cbp

（1）先配置板级 board_config.h 和对应配置文件中蓝牙双模使能

```
1. /*
2.  *  板级配置选择
3.  */
4. #define CONFIG_BOARD_AC631N_DEMO        // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER
5.
6. #include "board_ac631n_demo_cfg.h"
```

（2）配置对应的 board_acxxx_demo_cfg.h 文件使能 BLE 或 EDR(主机不支持 EDR)，以 board_ac630x_demo_cfg.h 为例

```
3. #define TCFG_USER_BLE_ENABLE            1  //BLE 功能使能
4. #define TCFG_USER_EDR_ENABLE            1  //EDR 功能使能
```

（3）配置 app_config.h，使能 AT

```
3. //app case 选择,只能选 1 个,要配置对应的 board_config.h
4. #define CONFIG_APP_SPP_LE                0
5. #define CONFIG_APP_AT_COM                1
6. #define CONFIG_APP_DONGLE                0
```

（4）配置 app_config.h, 选择 AT 主机或 AT 从机(二选一)

```
7.  //app case 选择,只能选 1 个,要配置对应的 board_config.h
8.  #define CONFIG_APP_AT_COM                    01//AT com HEX 格式命令
9.
10.
11.     #define TRANS_AT_COM                    0
12.     #define TRANS_AT_CLIENT                 1 //选择主机 AT
```

2.4.3 主要说明代码

代码文件	描述说明
app_at_com.c	任务主要实现，流程
at_uart.c	串口配置，数据收发
at_cmds.c	AT 协议解析处理
ble_at_com.c	从机 ble 控制实现
spp_at_com.c	spp 控制实现
ble_at_client.c	主机 ble 控制实现

2.5 APP - Bluetooth Dual-Mode Central

2.5.1 概述

Central 中心设备是使用 GATT 的 client 角色，在 SDK 中是以主机 Master 的方式实现，主动发起搜索和连接其他 BLE 设备。连接成功后遍历从机 GATT 的 Services 信息数据。最大支持 16 个 Sevices 遍历。

SDK 的例子是以杰理的数传 SDK 的 BLE 的设备中 Services 为搜索目标，用户根据需求也可自行搜索过滤其他的 Services。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.5.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631N_spp_and_le.cbp

（1）先配置板级 board_config.h 和对应配置文件中蓝牙 ble 使能

```
/*
 *  板级配置选择
 */
#define CONFIG_BOARD_AC631N_DEMO        // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER

#include "board_ac631n_demo_cfg.h"
```

//配置只支持 ble 模块

```
#define TCFG_USER_BLE_ENABLE            1  //BLE 功能使能
#define TCFG_USER_EDR_ENABLE            0  //EDR 功能使能
```

（2）配置 app 选择

```
//app case 选择,只能选 1 个,要配置对应的 board_config.h
#define CONFIG_APP_CENTRAL              1 //ble client,中心设备
```

2.5.3 主要代码说明

BLE 实现文件 ble_central.c，负责模块初始化、处理 GATT 模块事件和命令数据控制发送等。

1、配置 GATT client 的模块基本要素。

```
static const sm_cfg_t cetl_sm_init_config = {

static gatt_ctrl_t cetl_gatt_control_block = {

static const gatt_client_cfg_t central_client_init_cfg = {
    .event_packet_handler = cetl_client_event_packet_handler,
};
```

2、配置搜索的 GATT 服务以及记录搜索到的信息，支持 16bit 和 128 bit 的 UUID。

（1）配置扫描匹配的设备

```
//配置多个扫描匹配设备
static const u8 cetl_test_remoter_name1[] = "AC897N_MX(BLE)";//
static client_match_cfg_t cetl_match_device_table[] = {
#if MATCH_CONFIG_NAME
```

搜索匹配设备会发聩事件处理

```
case GATT_COMM_EVENT_SCAN_DEV_MATCH: {
        log_info("match_dev:addr_type= %d\n", packet[0]);
        put_buf(&packet[1], 6);
        if (packet[8] == 2) {
            log_info("is TEST_BOX\n");
        }
        client_match_cfg_t *match_cfg = ext_param;
        if (match_cfg) {
            log_info("match_mode: %d\n", match_cfg->create_conn_mode);
            if (match_cfg->compare_data_len) {
                put_buf(match_cfg->compare_data, match_cfg->compare_data_len);
            }
        }
    }
```

```
    break;
```

（2）配置连上后搜索的 uuid

```
//指定搜索 uuid
//指定搜索 uuid
static const target_uuid_t  jl_cetl_search_uuid_table[] = {

    // for uuid16
    // PRIMARY_SERVICE, ae30
    // CHARACTERISTIC,  ae01, WRITE_WITHOUT_RESPONSE | DYNAMIC,
    // CHARACTERISTIC,  ae02, NOTIFY,

    {
        .services_uuid16 = 0xae30,
        .characteristic_uuid16 = 0xae01,
        .opt_type = ATT_PROPERTY_WRITE_WITHOUT_RESPONSE,
    },

    {
        .services_uuid16 = 0xae30,
        .characteristic_uuid16 = 0xae02,
        .opt_type = ATT_PROPERTY_NOTIFY,
    },
```

查找到匹配的 UUID 和 handle，处理事件，并且可以记录操作 handle。

```
case GATT_COMM_EVENT_GATT_SEARCH_MATCH_UUID: {
        opt_handle_t *opt_hdl = packet;
        log_info("match:server_uuid= %04x,charactc_uuid= %04x,value_handle= %04x\n", \
                 opt_hdl->search_uuid->services_uuid16, opt_hdl->search_uuid->characteristic_uuid16, opt_hdl->value_handle);
#if cetl_TEST_WRITE_SEND_DATA
        //for test
```

```
        if (opt_hdl->search_uuid->characteristic_uuid16 == 0xae01) {
            cetl_ble_client_write_handle = opt_hdl->value_handle;
        }
#endif
    }
    break;
```

2、配置扫描和连接参数

扫描参数以 0.625ms 为单位，设置如下图：

```
#define SET_SCAN_TYPE     SCAN_ACTIVE
#define SET_SCAN_INTERVAL   48
#define SET_SCAN_WINDOW     16
```

连接参数 interval 是以 1.25ms 为单位，timeout 是以 10ms 为单位，如下图：

```
#define SET_CONN_INTERVAL   0x30
#define SET_CONN_LATENCY    0
#define SET_CONN_TIMEOUT    0x180
```

以上两组参数请慎重修改，必须按照蓝牙的协议规范来定义修改。

2.5.4 配置带 GATT 服务

配置主机可带有 GATT 服务，提供从机在连接链路上搜索；配置如下的宏为 1：

```
#define CONFIG_BT_GATT_SERVER_NUM           1 //(使能1,在主机链路上,提供服务service给对方搜索，以及操作,不创建新的连接)
```

GATT 服务处理 read 和 write 的操作详见文件 ble_central_server.c

2.6 APP - Bluetooth Dual-Mode Dongle

2.6.1 概述

蓝牙 dongle 符合 USB 和 BLE 或 EDR 传输标准，具有即插即用，方便实用的特点。它可用于蓝牙设备之间的数据传输，让电脑能够和周边的蓝牙设备进行无线连接和数据的通讯，自动发现和管理远程蓝牙设备、资源和服务，实现蓝牙设备之间的绑定和自动连接。

蓝牙 dongle 支持 BLE 和 2.4G 两种连接模式；支持连接指定蓝牙名或 mac 地址；应用示例是连接杰理的鼠标。

蓝牙 dongle 支持 EDR，支持连接指定蓝牙名或 mac 地址，连接杰理的键盘设备。

蓝牙双模 dongle 不能同时打开，会降低搜索效率。

支持的板级：br25、br30、bd19、br34

支持的芯片：AC636N、AC637N、AC632N、AC638N

2.6.2 工程配置

代码工程：apps\spp_and_le\board\br25\AC636N_spp_and_le.cbp

（1）APP 选择，配置 app_config.h

```
15. //app case  选择,只能选 1 个,要配置对应的 board_config.h
16. #define CONFIG_APP_DONGLE                    1//usb +  蓝牙(ble  主机),PC hid 设备
```

使用 EDR 模式，还必现打印如下主机搜索使能

```
17. #define EDR_EMITTER_EN                       1 //蓝牙(edr 主机)
```

（2）板级选择，配置 board_config.h。目前只有 AC6368B_DONGLE 板子支持蓝牙 dongle

```
18. //#define CONFIG_BOARD_AC636N_DEMO
19.  #define CONFIG_BOARD_AC6368B_DONGLE //CONFIG_APP_DONGLE
20. // #define CONFIG_BOARD_AC6363F_DEMO
```

（3）使能 USB 和 BLE ， 需配置 board_ac6368b_dongle_cfg.h

```
21. #define TCFG_PC_ENABLE                ENABLE_THIS_MOUDLE//PC 模块使能
22. #define TCFG_UDISK_ENABLE             DISABLE_THIS_MOUDLE//U 盘模块使能
23. #define TCFG_OTG_USB_DEV_EN           BIT(0)//USB0 = BIT(0)   USB1 = BIT(1)
```

```
#define TCFG_USER_BLE_ENABLE                    1     //BLE 或 2.4G 功能使能
#define TCFG_USER_EDR_ENABLE                    0     //EDR 功能使能
```

（4）模式选择，配置 BLE 或 2.4G 模式；若选择 2.4G 配对码必须跟对方的配对码一致

```
//2.4G 模式: 0---ble, 非 0---2.4G 配对码
#define CFG_RF_24G_CODE_ID          (0) //<=24bits
```

（5）如果选择 BLE 模式，则蓝牙 dongle 默认是按蓝牙名连接从机，需要配置连接的从机蓝牙名

```
static const u8 dongle_remoter_name1[] = "AC696X_1(BLE)";//
static const u8 dongle_remoter_name2[] = "AC630N_HID123(BLE)";// 自动连接同名的从机
```

（6）默认上电 10 秒根据信号强度 rssi 配对近距离的设备，若配对失败，停止搜索。回连已有的配对设备。

```
//dongle 上电开配对管理,若配对失败，没有配对设备，停止搜索
#define POWER_ON_PAIR_START       (1)//
#define POWER_ON_PAIR_TIME        (10000)//unit ms,持续配对搜索时间
#define DEVICE_RSSI_LEVEL         (-50)
#define POWER_ON_KEEP_SCAN        (0)//配对失败，保持一直搜索配对
```

2.6.3 主要代码说明

蓝牙 dongle 实现文件 app_dongle.c 和 ble_dg_central，负责模块初始化、处理协议栈事件和命令数据控制发送等。

（1）HID 描述符，描述为一个鼠标

```
static const u8 sHIDReportDesc[] = {
    0x05, 0x01,         // Usage Page (Generic Desktop Ctrls)
    0x09, 0x02,         // Usage (Mouse)
    0xA1, 0x01,         // Collection (Application)
    0x85, 0x01,         //     Report ID (1)
    0x09, 0x01,         //     Usage (Pointer)

```

（2）使用指定的 uuid 与从机通信，需要与从机配合，省掉了搜索 uuid 的时间

```
static const target_uuid_t   dongle_search_ble_uuid_table[] = {
    {
        .services_uuid16 = 0x1812,
        .characteristic_uuid16 = 0x2a4d,
        .opt_type = ATT_PROPERTY_NOTIFY,
    },

    {
        .services_uuid16 = 0x1812,
        .characteristic_uuid16 = 0x2a33,
        .opt_type = ATT_PROPERTY_NOTIFY,
    },

    {
        .services_uuid16 = 0x1801,
        .characteristic_uuid16 = 0x2a05,
        .opt_type = ATT_PROPERTY_INDICATE,
    },

};
```

```
/*
确定留给从机发数据的 3 个 notify handle
*/
static const u16 mouse_notify_handle[3] = {0x0027, 0x002b, 0x002f};
```

（3）用于监听蓝牙数据上报，并通过 USB 向 PC 转发蓝牙数据

```
//edr 接收设备数据
static void dongle_edr_hid_input_handler(u8 *packet, u16 size, u16 channel)
{
    log_info("hid_data_input:chl=%d,size=%d", channel, size);
```

```
5.      put_buf(packet, size);
```

```
1.  //ble 接收设备数据
2.  void dongle_ble_hid_input_handler(u8 *packet, u16 size)
3.  {
4.  #if TCFG_PC_ENABLE
5.      hid_send_data(packet, size);
6.  #endif
7.  }
```

（4）配置 BLE 的连接方式配置，可以选择通过地址或设备名等方式连接

```
1.  static const client_match_cfg_t dg_match_device_table[] = {
2.      {
3.          .create_conn_mode = BIT(CLI_CREAT_BY_NAME),
4.          .compare_data_len = sizeof(dg_test_remoter_name1) - 1, //去结束符
5.          .compare_data = dg_test_remoter_name1,
6.      },
```

2.6.4 OTA 升级功能

新增了支持使用 PC 电脑和安装手机平台升级 Usb Dongle 和 连接的远端 HID 设备固件。详细的配置和使用说明见发布包文档《Usb Dongle OTA 升级使用说明》。

2.7 APP - Bluetooth DualMode Keyboard

2.7.1 概述

本案例为基于 HID 的键盘设备，可以用来媒体播放，上下曲暂停音量的控制，支持安卓和 IOS 的双系统，并且支持 BLE 和 EDR 两种工作模式。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.7.2 工程配置

代码工程：apps\hid\board\br23\AC635N_hid.cbp

（1）先进行 APP 选择配置（apps\hid\include\app_config.h）

```
1. //app case 选择,只选 1,要配置对应的 board_config.h
2. #define CONFIG_APP_KEYBOARD                1//hid 按键 ,default case
3. #define CONFIG_APP_KEYFOB                  0//自拍器, board_ac6368a,board_6318
4. #define CONFIG_APP_MOUSE                   0//mouse,  board_mouse
5. #define CONFIG_APP_STANDARD_KEYBOARD       0//标准 HID 键盘,board_ac6351d
6. #define CONFIG_APP_KEYPAGE                 0//翻页器
```

（2）配置板级 board_config.h(apps\hid\board\bd29\board_config.h)，下图为选择 AC631n 板级，也可以选择 ac6313 板级。

```
1. /*
2.  * 板级配置选择
3.  */
4. #define CONFIG_BOARD_AC631N_DEMO   // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER

5. // #define CONFIG_BOARD_AC6313_DEMO  // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER

6. #include "board_ac631n_demo_cfg.h"
7. #include "board_ac6302a_mouse_cfg.h"
8. #include "board_ac6319a_mouse_cfg.h"
9. #include "board_ac6313_demo_cfg.h"
```

```
10. #include "board_ac6318_demo_cfg.h"
11.
12. #endif
```

//配置是否打开 edr 和 ble 模块

```
7.  #define TCFG_USER_BLE_ENABLE                1   //BLE 功能使能
8.  #define TCFG_USER_EDR_ENABLE                1   //EDR 功能使能
```

选择好对应的板级和 APP 后其他的设置及初始化按照默认设置，可以运行该 APP。

2.7.3 主要代码说明

（1）APP 注册运行

```
1.  #if (CONFIG_APP_MOUSE)
2.      it.name = "mouse";
3.      it.action = ACTION_MOUSE_MAIN;
4.  #elif(CONFIG_APP_KEYFOB)
5.      it.name = "keyfob";
6.      it.action = ACTION_KEYFOB;
7.  #elif(CONFIG_APP_KEYBOARD)
8.      it.name = "hid_key";

9.      start_app(&it);
10. }
```

首先在 app_main.c 函数中添加 hid_key 应用分支，然后进行应用注册。

```
1.  REGISTER_APPLICATION(app_hid) = {
2.      .name   = "hid_key",
3.      .action = ACTION_HID_MAIN,
4.      .ops    = &app_hid_ops,
5.      .state  = APP_STA_DESTROY,
6.  };
```

按照上述代码进行 APP 注册，执行配置好的 app。之后进入 APP_state_machine,根据状态机的不同状态执行不同的分支，第一次执行时进入 APP_STA_CREATE 分支，执行对应的 app_start()。开始执行 app_start()在该函数内进行时钟初始化，进行蓝牙模式选择，按键消息使能等一些初始化操作，

其中按键使能使得系统在有外部按键事件发生时及时响应，进行事件处理。

```
1.  REGISTER_LP_TARGET(app_hidkey_lp_target) = {
2.      .name = "app_hidkey_deal",
3.      .is_idle = hidkey_app_idle_query,
4.  };
5.  static const struct application_operation app_hidkey_ops = {
6.      .state_machine  = hidkey_state_machine,
7.      .event_handler  = hidkey_event_handler,
8.  };
```

键盘应用注册以后，进行以上的 app_hid_ops 进行处理。分为两个模块 hidkey_state_machine 和 hidkey_event_handler。执行流程大致如下，对应函数位于 app_keyboard.c 文件：hidkey_state_machine()--->app_start()--->sys_key_event_enable()。主要根据应用的状态进行时钟初始化，蓝牙名设置，读取配置信息，消息按键使能等配置。hidkey_event_handler(struct application *app, struct sys_event *event)--->hidkey_key_event_handler(sys_event *event)--->hidkey_key_deal_test(key_type,key_value)---->hidkey_key_deal_test(key_msg)。事件处理流程大致如上所示。hidkey_event_handler()根据传入的第二个参数事件类型，选择对应的处理分支，此处选择执行按键事件，然后调用按键事件处理函数根据事件的按键值和按键类型进行对应的事件处理。

（2）APP 事件处理机制

1 事件的产生与定义

外部事件的数据采集在系统软件定时器的中断服务函数中完成，采集的数据将被打包为相应的事件周期性地上报至全局事件列表。。

```
void sys_event_notify(struct sys_event *e);
```

此函数为事件通知函数，系统有事件发生时调用。

2 事件的处理

本案例中主要的事件处理包括连接事件处理、按键事件处理处理，事件处理函数的共同入口都是 event_handler()之后调用不同的函数实现不同类型事件的响应处理。

1.蓝牙连接事件处理

在 APP 运行以后，首先进行的蓝牙连接事件处理，进行蓝牙初始化，HID 描述符解读，蓝牙模式选择等，。调用 hidkey_event_handler(),hidkey_bt_connction_status_event_handler()函数实现蓝牙连接等事件。

2.按键事件处理

通过 hidkey_event_handler()函数中调用 hidkey_key_event_handler()函数实现对按键事件的处理。

(3)数据发送

自定义的键盘描述符如下所示：

```
1.  static const u8 hidkey_report_map[] = {
2.      0x05, 0x0C,         // Usage Page (Consumer)
3.      0x09, 0x01,         // Usage (Consumer Control)
4.      0xA1, 0x01,         // Collection (Application)
5.      0x85, 0x01,         //   Report ID (1)
6.      0x09, 0xE9,         //   Usage (Volume Increment)
7.      0x09, 0xEA,         //   Usage (Volume Decrement)
8.      0x09, 0xCD,         //   Usage (Play/Pause)
9.      0x09, 0xE2,         //   Usage (Mute)
10.     0x09, 0xB6,         //   Usage (Scan Previous Track)
11.     0x09, 0xB5,         //   Usage (Scan Next Track)
12.     0x09, 0xB3,         //   Usage (Fast Forward)
13.     0x09, 0xB4,         //   Usage (Rewind)
14.     0x15, 0x00,         //   Logical Minimum (0)
15.     0x25, 0x01,         //   Logical Maximum (1)
16.     0x75, 0x01,         //   Report Size (1)
17.     0x95, 0x10,         //   Report Count (16)
18.     0x81, 0x02,         //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Nu
    ll Position)
19.     0xC0,               // End Collection
20.     // 35 bytes
21. };
```

根据不同的按键值和按键类型，进行数据发送。部分发送函数入下：

```
1.      if (key_msg) {
2.          printf("key_msg = %02x\n", key_msg);
3.          if (bt_hid_mode == HID_MODE_EDR) {
```

```
4.              edr_hid_key_deal_test(key_msg);
5.              bt_sniff_ready_clean();
6.          } else {
7.      #if TCFG_USER_BLE_ENABLE
8.              ble_hid_key_deal_test(key_msg);
9.      #endif
10.         }
11.         return;
```

2.8 APP - Bluetooth DualMode Keyfob

2.8.1 概述

本案例主要用于蓝牙自拍器实现，进行以下配置后，打开手机蓝牙连接设备可进行对应的拍照操作。由于自拍器的使用会用到 LED 所以本案例也要对 LED 进行对应的设置，自拍器设备上电以后没有连接蓝牙之前，LED 以一定的频率闪烁，直到连接或者是进入 sleep 模式时熄灭。蓝牙连接以后 LED 熄灭，只有按键按下的时候 LED 会同时接通过，可以通过 LED 的状态来判断自拍器的工作状态。

支持板级：bd29、br25、br30、bd19

支持芯片：AC6318、AC6368A、AC6379B、AC6328A

2.8.2 工程配置

代码工程：apps\hid\board\bd19\AC632N_hid.cbp

（1）配置 app 选择(apps\hid\include\app_config.h),如下图选择对应的自拍器应用。

```
1.  //app case 选择,只选 1,要配置对应的 board_config.h
2.  #define CONFIG_APP_KEYFOB                       1//自拍器，board_ac6368a,board_6318
```

（2）先配置板级 board_config.h(apps\hid\board\bd29\board_config.h)，选择对应的开发板。

```
1.  //#define CONFIG_BOARD_AC631N_DEMO     // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER
2.  // #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
3.   #define CONFIG_BOARD_AC6318_DEMO    // CONFIG_APP_KEYFOB
4.
5.  #include "board_ac631n_demo_cfg.h"
6.  #include "board_ac6318_demo_cfg.h"
```

（3） 板级配置（board_ac6318_demo_cfg.h）

```
1.  //*********************************************************************************//
2.  //                          LED 配置                                //
3.  //*********************************************************************************//
4.  #define TCFG_PWMLED_ENABLE              ENABLE_THIS_MOUDLE      //是否支持 IO 推灯模块,bd29没有 PWM 模块
```

```
#define TCFG_PWMLED_IO_PUSH          ENABLE              //LED 使用的 IO 推灯
#define TCFG_PWMLED_IOMODE           LED_ONE_IO_MODE     //LED 模式，单 IO 还是两个 IO
#define TCFG_PWMLED_PIN              IO_PORTB_01         //LED 使用的 IO 口
```

//配置是否打开 edr 和 ble 模块

```
#define TCFG_USER_BLE_ENABLE            0   //BLE 功能使能
#define TCFG_USER_EDR_ENABLE            1   //EDR 功能使能
```

2.8.3 主要代码说明

（1）APP 注册运行

```
REGISTER_LP_TARGET(app_hid_lp_target) = {
    .name = "app_hid_deal",
    .is_idle = app_hid_idle_query,
};

static const struct application_operation app_hid_ops = {
    .state_machine  = state_machine,
    .event_handler  = event_handler,
};
/*
 * 注册 AT Module 模式
 */
REGISTER_APPLICATION(app_hid) = {
    .name   = "keyfob",
    .action = ACTION_KEYFOB,
    .ops    = &app_hid_ops,
    .state  = APP_STA_DESTROY,
};
```

按照上述代码进行 APP 注册，执行配置好的 app。之后进入 APP_state_machine,根据状态机的不同状态执行不同的分支，第一次执行时进入 APP_STA_CREATE 分支，执行对应的 app_start()。开始执行 app_start()在该函数内进行时钟初始化，进行蓝牙模式选择，按键消息使能等一些初始化操作，其中按键使能使得系统在有外部按键事件发生时及时响应，进行事件处理。

（2）APP 事件处理机制

1.事件的产生与定义

外部事件的数据采集在系统软件定时器的中断服务函数中完成，采集的数据将被打包为相应的事件周期性地上报至全局事件列表。

void sys_event_notify(**struct** sys_event *e)；

此函数为事件通知函数，系统有事件发生时调用。

2.事件的处理

本案例中主要的事件处理包括连接事件处理、按键事件处理和 LED 事件处理，事件处理函数的共同入口都是 event_handler().之后调用不同的函数实现不同类型事件的响应处理。

2.1 蓝牙连接事件处理

在 APP 运行以后，首先进行的蓝牙连接事件处理，进行蓝牙初始化，HID 描述符解读，蓝牙模式选择等，函数的第二个参数根据事件的不同，传入不同的事件类型，执行不同分支，如下图：

```
1.  static int keyfob_event_handler(struct application *app, struct sys_event *event)
2.  {
3.
4.  #if (TCFG_HID_AUTO_SHUTDOWN_TIME)
5.      //重置无操作定时计数
6.      sys_timer_modify(g_auto_shutdown_timer, TCFG_HID_AUTO_SHUTDOWN_TIME * 1000);
7.  #endif
8.
9.      bt_sniff_ready_clean();
10.
11.     /* log_info("event: %s", event->arg); */
12.     switch (event->type) {
13.     case SYS_KEY_EVENT:
14.         /* log_info("Sys Key : %s", event->arg); */
15.         app_key_event_handler(event);
16.         return 0;
```

以下图为蓝牙连接事件处理函数，进行蓝牙初始化以及模式选择。

```
1.  static int bt_connction_status_event_handler(struct bt_event *bt)
2.  {
3.
4.      log_info("----------------------bt_connction_status_event_handler %d", bt->event);
5.
```

```
    switch (bt->event) {
    case BT_STATUS_INIT_OK:
        /*
         * 蓝牙初始化完成
         */
        l
```

调用 keyfob_event_handler()，bt_connction_status_event_handler()函数实现蓝牙连接等事件。

2.2 按键事件处理和 LED 事件处理

通过调用 app_key_event_handler()函数进入按键事件处理流程，根据按键的类型和按键值进入 app_key_deal_test()和 key_value_send()函数进行事件处理。

```
static void app_key_event_handler(struct sys_event *event)
{
    /* u16 cpi = 0; */
    u8 event_type = 0;
    u8 key_value = 0;

    if (event->arg == (void *)DEVICE_EVENT_FROM_KEY) {
        event_type = event->u.key.event;
        key_value = event->u.key.value;
        printf("app_key_evnet: %d,%d\n", event_type, key_value);
        app_key_deal_test(event_type, key_value);
    }
}
```

```
static void app_key_deal_test(u8 key_type, u8 key_value)
{
    u16 key_msg = 0;

#if TCFG_USER_EDR_ENABLE
    if (!edr_hid_is_connected()) {
        if (bt_connect_phone_back_start(1)) { //回连
            return;
        }
```

```
10.  }
```

下图为 LED 工作状态部分实现函数。

```
1.  static void led_on_off(u8 state, u8 res)
2.  {
3.      /* if(led_state != state || (state == LED_KEY_HOLD)){ */
4.      if (1) { //相同状态也要更新时间
5.          u8 prev_state = led_state;
6.          log_info("led_state: %d>>>%d", led_state, state);
7.          led_state = state;
8.          led_io_flash = 0;
9.
10.
```

3.数据发送

KEYFOB 属于 HID 设备范畴，数据的定义与发送要根据 HID 设备描述符的内容进行确定，由下图的描述符可知，该描述符是一个用户自定义描述符，可以组合实现各种需要的功能，一共有两个 Input 实体描述符。其中每个功能按键对应一个 bit,一共 11bit,剩余一个 13bit 的常数输入实体，所以自定义描述符的数据包长度位 3byte.如果用户需要在自拍器的基础上增加不同按键类型的事件，可以在下面的描述符中先添加该功能，然后在按键处理函数分支进行对应的按键值和按键类型的设置，来实现对应的功能。

用户自定义的描述符组成本案例的 KEYFOB 描述符，实现对应的按键功能。

```
1.  static const u8 keyfob_report_map[] = {
2.      //通用按键
3.      0x05, 0x0C,         // Usage Page (Consumer)
4.      0x09, 0x01,         // Usage (Consumer Control)
5.      0xA1, 0x01,         // Collection (Application)
6.      0x85, 0x03,         //   Report ID (3)
7.      0x15, 0x00,         //   Logical Minimum (0)
8.      0x25, 0x01,         //   Logical Maximum (1)
9.      0x75, 0x01,         //   Report Size (1)
10.     0x95, 0x0B,         //   Report Count (11)
11.     0x0A, 0x23, 0x02,   //   Usage (AC Home)
12.     0x0A, 0x21, 0x02,   //   Usage (AC Search)
13.     0x0A, 0xB1, 0x01,   //   Usage (AL Screen Saver)
```

```
    0x09, 0xB8,         //   Usage (Eject)
    0x09, 0xB6,         //   Usage (Scan Previous Track)
    0x09, 0xCD,         //   Usage (Play/Pause)
    0x09, 0xB5,         //   Usage (Scan Next Track)
    0x09, 0xE2,         //   Usage (Mute)
    0x09, 0xEA,         //   Usage (Volume Decrement)
    0x09, 0xE9,         //   Usage (Volume Increment)
    0x09, 0x30,         //   Usage (Power)
    0x0A, 0xAE, 0x01,   //   Usage (AL Keyboard Layout)
    0x81, 0x02,         //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
    0x95, 0x01,         //   Report Count (1)
    0x75, 0x0D,         //   Report Size (13)
    0x81, 0x03,         //   Input (Const,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
    0xC0,               // End Collection

    // 119 bytes
};
```

图中 key_big_press/null 表示自定义描述符中实现音量增加的按键按下和抬起的数据包。

```
static void key_value_send(u8 key_value, u8 mode) {
    void (*hid_data_send_pt)(u8 report_id, u8 * data, u16 len) = NULL;

    if (bt_hid_mode == HID_MODE_EDR) {
#if TCFG_USER_EDR_ENABLE
        hid_data_send_pt = edr_hid_data_send;
#endif
    } else {
#if TCFG_USER_BLE_ENABLE
        hid_data_send_pt = ble_hid_data_send;
#endif
```

```
12.  }
13.
14.  if (!hid_data_send_pt) {
15.      return;
16.  }
```

2.9 APP - Bluetooth DualMode KeyPage

2.9.1 概述

本 APP 基于 HID 开发，主要用于浏览当下火爆的抖音等小视频的上下翻页、左右菜单切换、暂停等操作。首先选择需要用到的应用本案例选择，然后进行对应的支持板级选择，具体参考下文的步骤。通过软件编译下载到对应的开发板，打开手机蓝牙进行连接，进入视频浏览界面操作对应按键即可。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC638N

2.9.2 工程配置

代码工程：apps\hid\board\bd19\AC632N_hid.cbp

（1）app 配置

在工程代码中找到对应的文件(apps\hid\include\app_config.h)进行 APP 选择，本案例中选择翻页器，其结果如下图所示：

```
1. //app case 选择,只选 1,要配置对应的 board_config.h
2. #define CONFIG_APP_KEYPAGE                    1//翻页器
```

（2）板级选择

接着在文件(apps\hid\board\bd29\board_config.h)下进行对应的板级选择如下:

```
1. /*
2.  * 板级配置选择
3.  */
4. #define CONFIG_BOARD_AC631N_DEMO     // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNE
   R
5. // #define CONFIG_BOARD_AC6313_DEMO     // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURN
   ER
6.
7. #include "board_ac631n_demo_cfg.h"
8. #include "board_ac6302a_mouse_cfg.h"
9. #include "board_ac6319a_mouse_cfg.h"
10. #include "board_ac6313_demo_cfg.h"
```

```
#include "board_ac6318_demo_cfg.h"

#endif
```

该配置为选择对应的 board_ac631n_demo 板级。

//配置是否打开 edr 和 ble 模块

```
#define TCFG_USER_BLE_ENABLE                0   //BLE 功能使能
#define TCFG_USER_EDR_ENABLE                1   //EDR 功能使能
```

2.9.3 主要代码说明

(1)APP 注册(函数位于 apps/hid/app_keypage.c)

在系统进行初始化的过程中，根据以下信息进行 APP 注册。执行的大致流程为：REGISTER_APPLICATION--->state_machine--->app_start()--->sys_key_event_enable();这条流程主要进行设备的初始化设置以及一些功能使能。

REGISTER_APPLICATION--->event_handler--->app_key_event_handler()--->app_key_deal_test();这条流程在 event_handler 之下有多个 case,上述选择按键事件的处理流程进行代码流说明，主要展示按键事件发生时，程序的处理流程。

```
REGISTER_LP_TARGET(app_hid_lp_target) = {
    .name = "app_keypage",
    .is_idle = app_hid_idle_query,
};
static const struct application_operation app_hid_ops = {
    .state_machine  = state_machine,
    .event_handler  = event_handler,
};
 * 注册模式
REGISTER_APPLICATION(app_hid) = {
    .name   = "keypage",
    .action = ACTION_KEYPAGE,
    .ops    = &app_hid_ops,
    .state  = APP_STA_DESTROY,
};
```

(2) APP 状态机

状态机有 create，start，pause，resume，stop，destory 状态，根据不同的状态执行对应的分支。APP 注册后进行初始运行，进入 APP_STA_START 分支，开始 APP 运行。

```
1.  static int state_machine(struct application *app, enum app_state state, struct intent *it)
2.  {   switch (state) {
3.      case APP_STA_CREATE:
4.          break;
5.      case APP_STA_START:
6.          if (!it) {
7.              break;      }
8.          switch (it->action) {
9.          case ACTION_TOUCHSCREEN:
10.             app_start();
```

进入 app_start()函数后进行对应的初始化，时钟初始化，模式选择，低功耗初始化，以及外部事件使能。

```
1.  static void app_start()
2.  {
3.      log_info("============================================");
4.      log_info("----------------------------KEYPAGE----------------------");
5.      log_info("============================================");
6.
```

(3) APP 事件处理机制

1.事件的定义(代码位于 Headers\include_lib\system\even.h 中)

```
1.  struct sys_event {
2.      u16 type;
3.      u8 consumed;
4.      void *arg;
5.      union {
6.          struct key_event key;
7.          struct axis_event axis;
8.          struct codesw_event codesw;
```

（4）事件的产生（include_lib\system\event.h）

```
void sys_event_notify(struct sys_event *e);
```

事件通知函数,系统有事件发生时调用此函数。

（5）事件的处理(app_keypage.c)

函数执行的大致流程为：evevt_handler()--->app_key_event_handler()--->app_key_deal_test().

```
1.  static int event_handler(struct application *app, struct sys_event *event)
2.  {
3.  #if (TCFG_HID_AUTO_SHUTDOWN_TIME)
4.      //重置无操作定时计数
5.      sys_timer_modify(g_auto_shutdown_timer, TCFG_HID_AUTO_SHUTDOWN_TIME * 1000);
6.  #endif
7.
```

```
1.  static void app_key_event_handler(struct sys_event *event)
2.  {
3.      /* u16 cpi = 0; */
4.      u8 event_type = 0;
5.      u8 key_value = 0;
6.
7.      if (event->arg == (void *)DEVICE_EVENT_FROM_KEY) {
8.          event_type = event->u.key.event;
9.          key_value = event->u.key.value;
10.         printf("app_key_evnet: %d,%d\n", event_type, key_value);
11.         app_key_deal_test(event_type, key_value);
12.     }
13. }
```

```
1.  static void app_key_deal_test(u8 key_type, u8 key_value)
2.  {
3.      u16 key_msg = 0;
4.      void (*hid_data_send_pt)(u8 report_id, u8 * data, u16 len) = NULL;
5.
6.      log_info("app_key_evnet: %d,%d\n", key_type, key_value);
```

```
7.
8.  }
```

（6）APP 数据的发送

当 APP 注册运行后，有按键事件发生时，会进行对应的数据发送，由于是 HID 设备，所以数据的发送形式从对应的 HID 设备的描述符产生。用户如需要对设备进行功能自定义，可以结合 HID 官方文档对下述描述符进行修改。部分描述符如下：

```
1.  static const u8 hid_report_map[] = {
2.      // 119 bytes
3.      0x05, 0x0D,     // Usage Page (Digitizer)
4.      0x09, 0x02,     // Usage (Pen)
5.      0xA1, 0x01,     // Collection (Application)
6.      0x85, 0x01,     //  Report ID (1)
7.      0x09, 0x22,     //  Usage (Finger)
8.      0xA1, 0x02,     //  Collection (Logical)
9.      0x09, 0x42,     //  Usage (Tip Switch)
10.     0x15, 0x00,     //  Logical Minimum (0)
11.     0x25, 0x01,     //  Logical Maximum (1)
12.     0x75, 0x01,     //  Report Size (1)
13.     0x95, 0x01,     //  Report Count (1)
14.     0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
15.     0x09, 0x32,     //  Usage (In Range)
16.     0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
17.     0x95, 0x06,     //  Report Count (6)
18.     0x81, 0x03,     //  Input (Const,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
19.     0x75, 0x08,     //  Report Size (8)
20.     0x09, 0x51,     //  Usage (0x51)
21.     0x95, 0x01,     //  Report Count (1)
22.     0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
23.     0x05, 0x01,     //  Usage Page (Generic Desktop Ctrls)
1.  static const u8 pp_press[7] = {0x07,0x07,0x70,0x07,0x70,0x07,0x01};
2.  static const u8 pp_null[7]=  {0x00,0x07,0x70,0x07,0x70,0x07,0x00};
```

上图示为暂停按键对应的 HID 设备发送数据包，通过下图的 hid_data_send_pt()进行数据传输。

```
1.  log_info("point: %d,%d", point_cnt, point_len);
```

```
    if (point_cnt) {
        for (int cnt = 0; cnt < point_cnt; cnt++) {
            hid_data_send_pt(1, key_data, point_len);
            key_data += point_len;
            KEY_DELAY_TIME();
        }
    }
```

由描述符可知，设备一共有 5 个输入实体 Input，一共组成 7byte 的数据，所以对应的暂停按键数据包由 7byte 的数据组成，前 2byte 表示识别是否有触摸输入，中间 2 个 2byte 分别表示 y 坐标和 x 坐标，最后 1byte 表示 contact count，不同的按键事件对应不同的数据包，数据通过 hid_data_send_dt 函数发送至设备。对应的按键事件通过事件处理机制和数据发送实现对应的功能。

（6）增加处理公共消息

跟进识别不同的手机系统，来切换描述符。

```
static int app_common_event_handler(struct bt_event *bt)
{
    log_info("--------app_common_event_handler: %02x %02x", bt->event, bt->value);

    switch (bt->event) {
    case COMMON_EVENT_EDR_REMOTE_TYPE:
        log_info(" COMMON_EVENT_EDR_REMOTE_TYPE \n");
        connect_remote_type = bt->value;
        if (connect_remote_type == REMOTE_DEV_IOS) {
            user_hid_set_ReportMap(hid_report_map_ios, sizeof(hid_report_map_ios));

        } else {
            user_hid_set_ReportMap(hid_report_map, sizeof(hid_report_map));
        }
        break;

    case COMMON_EVENT_BLE_REMOTE_TYPE:
        log_info(" COMMON_EVENT_BLE_REMOTE_TYPE \n");
        connect_remote_type = bt->value;
```

2.10 APP - Bluetooth DualMode Standard Keyboard

2.10.1 概述

本案例主要用于标准双模蓝牙键盘的实现，设备开机之后进入配对状态，用于可以与能搜索到蓝牙 3.0、蓝牙 4.0 两种设备进行连接，连接成功之后，另一个会消失。SDK 默认支持 4 个设备连接键盘（同时只能连接一个），通过按键进行切换设备。

支持板级：br23、bd19

支持芯片：AC6351D、AC6321A

AC6321A 键盘方案，新增加了按键扫描模块 AD15N，除了处理按键扫描有点区别外，其他部分实现方式跟 AC6351D 实现方式是一致的，另外下载增加打包 ex_mcu.bin 文件，可以修改批处理 tools\download\data_trans\download.bat。

修改如下：

..\..\isd_download.exe ..\..\isd_config.ini -tonorflash -dev bd19 -boot 0x2000 -div8 -wait 300 -uboot ..\..\uboot.boot -app ..\..\app.bin ..\..\cfg_tool.bin -res ..\..\p11_code.bin ..\..\ex_mcu.bin -uboot_compress

2.10.2 工程配置

代码工程：apps\hid\board\br23\AC635N_hid.cbp

（1）配置 app 选择(apps\hid\include\app_config.h)，如下图选择对应的标准键盘应用。

```
1.  //app case 选择,只选 1,要配置对应的 board_config.h
2.  #define CONFIG_APP_STANDARD_KEYBOARD      1//标准 HID 键盘,board_ac6351d
```

（2）先配置板级 board_config.h(apps\hid\board\br23\board_config.h)，选择对应的开发板。

```
7.  #define CONFIG_BOARD_AC635N_DEMO
8.  //#define CONFIG_BOARD_AC6351D_KEYBOARD
9.
10. #include "board_ac635n_demo_cfg.h"
11. #include "board_ac6351d_keyboard_cfg.h"
```

（3） 功能配置（board_ac6351d_keyboard_cfg.h）

//配置打开 edr 或者 ble 模块

```
#define TCFG_USER_BLE_ENABLE            0  //BLE 功能使能
#define TCFG_USER_EDR_ENABLE            1  //EDR 功能使能
```

```
//*********************************************************************//
//                  矩阵按键 配置                                      //
//*********************************************************************//
#define TCFG_MATRIX_KEY_ENABLE              ENABLE_THIS_MOUDLE

//*********************************************************************//
//                    触摸板 配置                                      //
//*********************************************************************//
#define TCFG_TOUCHPAD_ENABLE                ENABLE_THIS_MOUDLE
```

（4） IO 配置（board_ac6351d_keyboard.c）

配置矩阵扫描行列 IO

```
static u32 key_row[] = {IO_PORTB_06, IO_PORTB_07, IO_PORTB_08, IO_PORTB_09, IO_PORTB_10, IO_PORTB_11,
 IO_PORTC_06, IO_PORTC_07};
static u32 key_col[] = {IO_PORTA_00, IO_PORTA_01, IO_PORTA_02, IO_PORTA_03,
IO_PORTA_04, IO_PORTA_05, IO_PORTA_06, IO_PORTA_07, \
IO_PORTA_08, IO_PORTA_09, IO_PORTA_10,  IO_PORTA_11, IO_PORTA_12, IO_PORTA_13, IO_PORTC_00, IO_PORTA_
14, IO_PORTC_01, IO_PORTA_15, IO_PORTB_05,
};
```

配置触摸板 IIC 通信接口

```
const struct soft_iic_config soft_iic_cfg[] = {
    //iic0 data
    {
        .scl = TCFG_SW_I2C0_CLK_PORT,                   //IIC CLK 脚
        .sda = TCFG_SW_I2C0_DAT_PORT,                  //IIC DAT 脚
        .delay = TCFG_SW_I2C0_DELAY_CNT,               //软件 IIC 延时参数，影响通讯时钟频率
        .io_pu = 1,                                    //是否打开上拉电阻，如果外部电路没有焊接上拉电阻需
要置 1
    },
```

```
};
```

（5） 唤醒口配置（board_ac6351d_keyboard.c）

键盘进入低功耗之后需要通过按键唤醒 cpu，635N 支持 8 个普通 IO、LVD 唤醒、LDOIN 唤醒

普通 IO 唤醒：

```
struct port_wakeup port0 = {
    .pullup_down_enable = ENABLE,                   //配置 I/O 内部上下拉是否使能
    .edge               = FALLING_EDGE,             //唤醒方式选择,可选：上升沿\下降沿
    .attribute          = BLUETOOTH_RESUME,         //保留参数
    .iomap              = IO_PORTB_06,              //唤醒口选择
    .filter_enable      = ENABLE,
};
const struct wakeup_param wk_param = {
    .port[0] = &port0,
    ...
    ...
    .port[7] = &port7,
    .sub = &sub_wkup,
    .charge = &charge_wkup,
};
```

LVD 唤醒:

lvd_extern_wakeup_enable();//要根据封装来选择是否可以使用 LVD 唤醒，6531C 封装 LVD 是 PB4

LDOIN 唤醒

LDOIN 唤醒为充电唤醒

（6） 键值的配置（app_standard_keyboard.c）

app_standard_keyboard.c 文件中定义了键盘的键值表 matrix_key_table 和 fn 键重映射键值表 fn_remap_event 还要其他按键事件 other_key_map

- matrix_key_table 定义的是标准 Keyboard 的键值，如 RCTRL、LCTRL、A、B 等..., 用户根据方案选择键芯来修改键盘键值表，对应的键值功能定义在`apps/common/usb/host/usb_hid_keys.h`
- fn_remap_event 分为两种，一种用于系统控制，如音量加键、搜索查找等，另一种为用于客户自定义的功能，如蓝牙切换等, is_user_key 为 0 表示按键为系统控制用，键值可以在 COUSTOM_CONTROL 页里找。
- Is_user_key 为 1 表示为用于自动义按键，跟标准 HID 无关，相关按键的处理再 user_key_deal 里处理。

2.10.3 主要代码说明

（1）APP 注册运行

```
1.  REGISTER_LP_TARGET(app_hid_lp_target) = {
2.      .name = "app_hid_deal",
3.      .is_idle = app_hid_idle_query,
4.  };
5.
6.  static const struct application_operation app_hid_ops = {
7.      .state_machine  = state_machine,
8.      .event_handler  = event_handler,
9.  };
10. /*
11.  * 注册 AT Module 模式
12.  */
13. REGISTER_APPLICATION(app_hid) = {
14.     .name   = "hid_key",
15.     .action = ACTION_KEYFOB,
16.     .ops    = &app_hid_ops,
17.     .state  = APP_STA_DESTROY,
18. };
```

按照上述代码进行 APP 注册，执行配置好的 app。之后进入 APP_state_machine,根据状态机的不同状态执行不同的分支，第一次执行时进入 APP_STA_CREATE 分支，执行对应的 app_start()。开始执行 app_start()在该函数内进行时钟初始化，进行蓝牙模式选择，按键消息使能等一些初始化操作，其中按键使能使得系统在有外部按键事件发生时及时响应，进行事件处理。

（2）APP 事件处理机制

1.事件的产生与定义

外部事件的数据采集在系统软件定时器的中断服务函数中完成，采集的数据将被打包为相应的事件周期性地上报至全局事件列表。

```
void sys_event_notify(struct sys_event *e);
```

此函数为事件通知函数，系统有事件发生时调用。

2.事件的处理

本案例中主要的事件处理包括连接事件处理、按键事件处理和 LED 事件处理，事件处理函数的

共同入口都是 event_handler().之后调用不同的函数实现不同类型事件的响应处理。

2.1 蓝牙连接事件处理

在 APP 运行以后，首先进行的蓝牙连接事件处理，进行蓝牙初始化，HID 描述符解读，蓝牙模式选择等，函数的第二个参数根据事件的不同，传入不同的事件类型，执行不同分支，如下图：

```
1.  static int event_handler(struct application *app, struct sys_event *event)
2.  {
3.
4.  #if (TCFG_HID_AUTO_SHUTDOWN_TIME)
5.      //重置无操作定时计数
6.      sys_timer_modify(g_auto_shutdown_timer, TCFG_HID_AUTO_SHUTDOWN_TIME * 1000);
7.  #endif
8.
9.      bt_sniff_ready_clean();
10.
11.     /* log_info("event: %s", event->arg); */
12.     switch (event->type) {
13.     case SYS_KEY_EVENT:
14.         /* log_info("Sys Key : %s", event->arg); */
15.         app_key_event_handler(event);
16.         return 0;
```

以下图为蓝牙连接事件处理函数，进行蓝牙初始化以及模式选择。

```
1.  static int bt_connction_status_event_handler(struct bt_event *bt)
2.  {
3.
4.      log_info("----------------------bt_connction_status_event_handler %d", bt->event);
5.
6.      switch (bt->event) {
7.      case BT_STATUS_INIT_OK:
8.          /*
9.           * 蓝牙初始化完成
10.          */
11.         1
```

调用 event_handler()，bt_connction_status_event_handler()函数实现蓝牙连接等事件。

2.2 按键事件处理和触摸板事件处理

通过调用 app_key_event_handler()函数进入按键事件处理流程，根据按键的类型和按键值进入 app_key_deal_test()和 key_value_send()函数进行事件处理。

```
1.  void matrix_key_map_deal(u8 *map)
2.  {
3.      u8 row, col, i = 0;
4.      static u8 fn_press = 0;
5.
6.      if (special_key_deal(map, fn_remap_key, sizeof(fn_remap_key) / sizeof(special_ke
    y), fn_remap_event, 1)) {
7.          return;
8.      }
9.
10.     if (special_key_deal(map, other_key, sizeof(other_key) / sizeof(special_key), ot
    her_key_map, 0)) {
11.         return;
12.     }
13.
14.     for (col = 0; col < COL_MAX; col++) {
15.         for (row = 0; row < ROW_MAX; row++) {
16.             if (map[col] & BIT(row)) {
17.                 full_key_array(row, col, MATRIX_KEY_SHORT);
18.             } else {
19.                 full_key_array(row, col, MATRIX_KEY_UP);
20.             }
21.         }
22.     }
23.     Phantomkey_process();
24.     send_matrix_key_report(key_status_array);
25. }
```

```
1.  void touch_pad_event_deal(struct sys_event *event)
2.  {
```

```
3.      u8 mouse_report[8] = {0};
4.      if ((event->u).touchpad.gesture_event) {
5.          //g_printf("touchpad gesture_event:0x%x\n", (event->u).touchpad.gesture_even
    t);
6.          switch ((event->u).touchpad.gesture_event) {
7.          case 0x1:
8.              mouse_report[0] |= _KEY_MOD_LMETA;
9.              mouse_report[2] = _KEY_EQUAL;
10.             hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
11.             memset(mouse_report, 0x0, 8);
12.             hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
13.             return;
14.         case 0x2:
15.             mouse_report[0] |= _KEY_MOD_LMETA;
16.             mouse_report[2] = _KEY_MINUS;
17.             hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
18.             memset(mouse_report, 0x0, 8);
19.             hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
20.             return;
21.         case 0x3:
22.             mouse_report[0] |= BIT(0);          //鼠标左键
23.             break;
24.         case 0x4:
25.             mouse_report[0] |= BIT(1);
26.             break;
27.         }
28.     }
29.     if ((event->u).touchpad.x || (event->u).touchpad.y) {
30.         mouse_report[1] = gradient_acceleration((event->u).touchpad.x);
31.         mouse_report[2] = gradient_acceleration((event->u).touchpad.y);
32.     }
33.     hid_report_send(MOUSE_POINT_REPORT_ID, mouse_report, 3);
34. }
```

3.数据发送

KEYBOARD 属于 HID 设备范畴，数据的定义与发送要根据 HID 设备描述符的内容进行确定，由下图的描述符可知，该描述符是一个用户自定义描述符，由 KeyBoard、Consumer Control 和 Mouse 组成， Keyboard 主要实现普通按键的功能，Consumer Control 实现多媒体系统控制， Mouse 实现触摸板功能。

```
0x05, 0x01,                 // Usage Page (Generic Desktop Ctrls)
0x09, 0x06,                 // Usage (Keyboard)
0xA1, 0x01,                 // Collection (Application)
0x85, KEYBOARD_REPORT_ID,//     Report ID (1)
0x05, 0x07,                 //     Usage Page (Kbrd/Keypad)
0x19, 0xE0,                 //     Usage Minimum (0xE0)
0x29, 0xE7,                 //     Usage Maximum (0xE7)
0x15, 0x00,                 //     Logical Minimum (0)
0x25, 0x01,                 //     Logical Maximum (1)
0x75, 0x01,                 //     Report Size (1)
0x95, 0x08,                 //     Report Count (8)
0x81, 0x02,                 //     Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x95, 0x01,                 //     Report Count (1)
0x75, 0x08,                 //     Report Size (8)
0x81, 0x01,                 //     Input (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x95, 0x03,                 //     Report Count (3)
0x75, 0x01,                 //     Report Size (1)
0x05, 0x08,                 //     Usage Page (LEDs)
0x19, 0x01,                 //     Usage Minimum (Num Lock)
0x29, 0x03,                 //     Usage Maximum (Scroll Lock)
0x91, 0x02,                 //     Output (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position,Non-volatile)
0x95, 0x05,                 //     Report Count (5)
0x75, 0x01,                 //     Report Size (1)
0x91, 0x01,                 //     Output (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Position,Non-volatile)
0x95, 0x06,                 //     Report Count (6)
0x75, 0x08,                 //     Report Size (8)
0x15, 0x00,                 //     Logical Minimum (0)
0x26, 0xFF, 0x00,   //     Logical Maximum (255)
0x05, 0x07,                 //     Usage Page (Kbrd/Keypad)
```

```
    0x19, 0x00,             //      Usage Minimum (0x00)
    0x2A, 0xFF, 0x00,   //      Usage Maximum (0xFF)
    0x81, 0x00,             //      Input (Data,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
    0xC0,                       // End Collection
    0x05, 0x0C,             // Usage Page (Consumer)
    0x09, 0x01,             // Usage (Consumer Control)
    0xA1, 0x01,             // Collection (Application)
    0x85, COUSTOM_CONTROL_REPORT_ID,//      Report ID (3)
    0x75, 0x10,             //      Report Size (16)
    0x95, 0x01,             //      Report Count (1)
    0x15, 0x00,             //      Logical Minimum (0)
    0x26, 0x8C, 0x02,   //      Logical Maximum (652)
    0x19, 0x00,             //      Usage Minimum (Unassigned)
    0x2A, 0x8C, 0x02,   //      Usage Maximum (AC Send)
    0x81, 0x00,             //      Input (Data,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
    0xC0,                       // End Collection
//
    // Dummy mouse collection starts here
    //
    0x05, 0x01,             // USAGE_PAGE (Generic Desktop)
    0x09, 0x02,             // USAGE (Mouse)
    0xa1, 0x01,             // COLLECTION (Application)
    0x85, MOUSE_POINT_REPORT_ID,        //      REPORT_ID (Mouse)
    0x09, 0x01,             //      USAGE (Pointer)
    0xa1, 0x00,             //      COLLECTION (Physical)
    0x05, 0x09,             //          USAGE_PAGE (Button)
    0x19, 0x01,             //          USAGE_MINIMUM (Button 1)
    0x29, 0x02,             //          USAGE_MAXIMUM (Button 2)
    0x15, 0x00,             //          LOGICAL_MINIMUM (0)
    0x25, 0x01,             //          LOGICAL_MAXIMUM (1)
    0x75, 0x01,             //          REPORT_SIZE (1)
    0x95, 0x02,             //          REPORT_COUNT (2)
    0x81, 0x02,             //          INPUT (Data,Var,Abs)
```

```
0x95, 0x06,                    //        REPORT_COUNT (6)
0x81, 0x03,                    //        INPUT (Cnst,Var,Abs)
0x05, 0x01,                    //        USAGE_PAGE (Generic Desktop)
0x09, 0x30,                    //        USAGE (X)
0x09, 0x31,                    //        USAGE (Y)
0x15, 0x81,                    //        LOGICAL_MINIMUM (-127)
0x25, 0x7f,                    //        LOGICAL_MAXIMUM (127)
0x75, 0x08,                    //        REPORT_SIZE (8)
0x95, 0x02,                    //        REPORT_COUNT (2)
0x81, 0x06,                    //        INPUT (Data,Var,Rel)
0xc0,                          //    END_COLLECTION
0xc0                           // END_COLLECTION
```

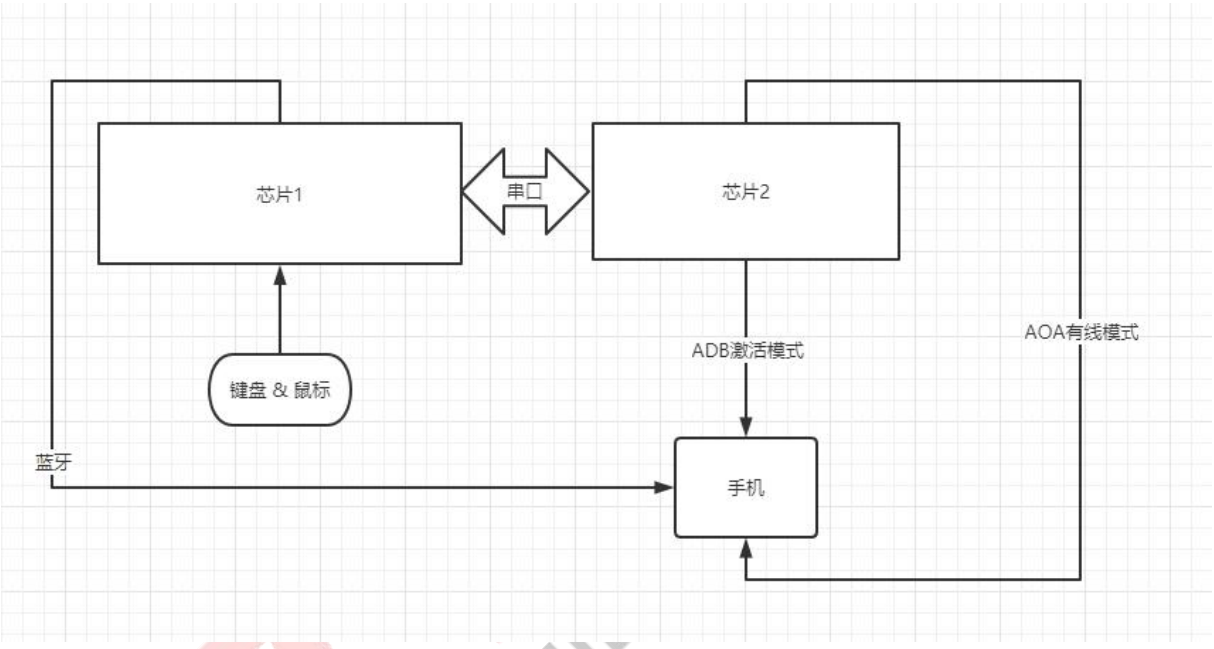
HID 数据发送接口，根据当前连接模式来发送 HID report。

```
1.  void hid_report_send(u8 report_id, u8 *data, u16 len)
2.  {
3.      if (bt_hid_mode == HID_MODE_EDR) {
4.  #if TCFG_USER_EDR_ENABLE
5.          edr_hid_data_send(report_id, data, len);
6.  #endif
7.      } else {
8.  #if TCFG_USER_BLE_ENABLE
9.          ble_hid_data_send(report_id, data, len);
10. #endif
11.     }
12. }
```

2.11 APP - Gamebox 吃鸡王座 mode

2.11.1 概述

本案例主要用于吃鸡王座的实现，开手机蓝牙连接设备，将键盘鼠标分别接到 usb 0 usb1。 枚举成功后键鼠的灯会亮。 也可使用 AOA 有线模式连接安卓手机，此时需要两个芯片，使用一个双 usb 芯片解析键鼠，通过串口发送键鼠数据给另一个芯片,系统框图如下



如果不需要有线模式，可以省略芯片 2，增加一个模拟开关用于激活 MTK 平台的手机。

吃鸡王座 SDK。支持默认布局的和平精英游戏。

按键映射如下：

F1 连发模式开关

F2 压枪开关

F3 加快开枪速度

F4 降低开枪速度

F5 增加压枪力度

F6 降低压枪力度

支持板级：bd19 、bd29 、br23

支持芯片：AC6311、AC6321A

2.11.2 工程配置

代码工程：apps\hid\board\bd19\AC632N_hid.cbp

（1）配置 app 选择(apps\hid\include\app_config.h),如下图选择对应的标准键盘应用。

```
3.  //app case 选择,只选 1,要配置对应的 board_config.h
4.  #define CONFIG_APP_KEYBOARD                0//hid 按键 ,default case
5.  #define CONFIG_APP_KEYFOB                  0//自拍器, board_ac6368a,board_6318
6.  #define CONFIG_APP_MOUSE                   0//mouse, board_mouse
7.  #define CONFIG_APP_STANDARD_KEYBOARD       0//标准 HID 键盘,board_ac6351d
8.  #define CONFIG_APP_KEYPAGE                 0//翻页器
9.  #define CONFIG_APP_GAMEBOX                 1//吃鸡王座
```

（2）先配置板级 board_config.h(apps\hid\board\bd19\board_config.h)，选择对应的开发板。

```
12. #define CONFIG_BOARD_AC635N_DEMO
13. //#define CONFIG_BOARD_AC6351D_KEYBOARD
14.
15. #include "board_ac635n_demo_cfg.h"
```

```
16. #include "board_ac6351d_keyboard_cfg.h"
```

（3） 功能配置（board_ac632n_demo_cfg.h）

```
10. //*********************************************************************//
11.     USB 配置
12. #define TCFG_PC_ENABLE                  DISABLE_THIS_MOUDLE //PC 模块使能
13. #define TCFG_UDISK_ENABLE               DISABLE_THIS_MOUDLE //U 盘模块使能
14. #define TCFG_HID_HOST_ENABLE            ENABLE_THIS_MOUDLE  //游戏盒子模式
15. #define TCFG_ADB_ENABLE                 ENABLE_THIS_MOUDLE
16. #define TCFG_AOA_ENABLE                 ENABLE_THIS_MOUDLE
```

//配置打开 ble 模块

```
15. #define TCFG_USER_BLE_ENABLE            1   //BLE 功能使能
16. #define TCFG_USER_EDR_ENABLE            0   //EDR 功能使能
```

2.11.3 主要代码说明

1.事件的处理

本案例中主要的事件处理包括 usb 拔插，MTK 普通 adb 激活模式手机拔插事件。鼠标移动点击，键盘按键

2.1 usb 事件处理

```
1.  static void usb_event_handler(struct sys_event *event, void *priv)
2.  {
3.      const char *usb_msg;
4.      usb_dev usb_id;
5.
6.      switch ((u32)event->arg) {
7.      case DEVICE_EVENT_FROM_OTG:
8.          usb_msg = (const char *)event->u.dev.value;
9.          usb_id = usb_msg[2] - '0';
10.
11.         log_debug("usb event : %d DEVICE_EVENT_FROM_OTG %s",
```

```
12.                 event->u.dev.event, usb_msg);
13.         if (usb_msg[0] == 'h') {
14.             if (event->u.dev.event == DEVICE_EVENT_IN) {
15.                 log_info("usb %c online", usb_msg[2]);
16.                 if (usb_host_mount(usb_id, 3, 20, 250)) {
17.                     usb_h_force_reset(usb_id);
18.                     usb_otg_suspend(usb_id, OTG_UNINSTALL);
19.                     usb_otg_resume(usb_id);
20.                 }
21.             } else if (event->u.dev.event == DEVICE_EVENT_OUT) {
22.                 log_info("usb %c offline", usb_msg[2]);
23.                 set_phone_connect_status(0);
24.                 usb_host_unmount(usb_id);
25.             }
26.         } else if (usb_msg[0] == 's') {
27. #if TCFG_PC_ENABLE
28.             if (event->u.dev.event == DEVICE_EVENT_IN) {
29.                 usb_start(usb_id);
30.             } else {
31.                 usb_stop(usb_id);
32.             }
33. #endif
34.         }
35.         break;
36.     case DEVICE_EVENT_FROM_USB_HOST:
37.         log_debug("host_event %x", event->u.dev.event);
38.         if ((event->u.dev.event == DEVICE_EVENT_IN) ||
39.             (event->u.dev.event == DEVICE_EVENT_CHANGE)) {
40.                     int err = os_taskq_post_msg(TASK_NAME, 2, DEVICE_EVENT_IN,
    event->u.dev.value);
41.             if (err) {
42.                 r_printf("err %x ", err);
43.             }
```

```
44.          } else if (event->u.dev.event == DEVICE_EVENT_OUT) {
45.              log_error("device out %x", event->u.dev.value);
46.          }
47.          break;
48.      }
49.  }
```

以下鼠标处理函数，

```
1.  void mouse_route(const struct mouse_data_t *p)
2.  {
3.      if (get_run_mode() != UART_MODE) {
4.          if (mouse_filter((void *)p) == 0) {
5.              return;
6.          }
7.      }
8.      /* log_info("btn: %x x-y %d %d wheel %d ac_pan %d",  */
9.      /*         p->btn, p->x, p->y, p->wheel, p->ac_pan); */
10.     switch (get_run_mode()) {
11.     case UART_MODE ://在 USB 中断函数调用
12.         send2uart(MOUSE_POINT_MODE, p);
13.         break;
14.     case BT_MODE ://在 uart 中断 或者 usb 中断函数调用
15.     case USB_MODE://在串口中断调用
16.         if (is_mouse_point_mode) {
17.             send2phone(MOUSE_POINT_MODE, p);
18.         } else {
19.             mouse_mapping(p);
20.             send2phone(TOUCH_SCREEN_MODE, p);
21.         }
22.         break;
23.     case MAPPING_MODE:
24.         send2phone(MOUSE_POINT_MODE + 1, p);
25.         break;
```

```
    default :
        log_info("btn: %x x-y %d %d wheel %d ac_pan %d",
                 p->btn, p->x, p->y, p->wheel, p->ac_pan);
        break;
    }
}
```

处理中间键消息， 指针模式，触摸模式切换。

2.2 键盘数据处理

```
void keyboard_route(const u8 *p)
{
    /* log_info("keyboard:"); */
    /* printf_buf(p, 8); */
    if (keyboard_filter((struct keyboard_data_t *)p) == 0) {
        return;
    }

    switch (get_run_mode()) {
    case UART_MODE ://在 USB 中断函数调用
        send2uart(KEYBOARD_MODE, p);
        break;
    case BT_MODE ://在 uart 接收事件 或者 usb 中断函数调用
    case USB_MODE://在串口事件调用
        key_mapping((const void *)p);
        send2phone(TOUCH_SCREEN_MODE, p);
        break;
    case MAPPING_MODE:
        send2phone(KEYBOARD_MODE, p);
        break;
    default :
        printf_buf((u8 *)p, 8);
        break;
```

```
24.     }
25. }
```

3. **Adb 激活 MTK 手机**

是跑激活过程的脚本， active.bash

APP_ACTIVITY_PATH 安卓激活页面

```
1.  #define      APP_ACTIVITY_PATH    "com.zh-jieli.gmaeCenter/com.zh-jieli.gameCenter.activity.guide.SplashActivity\n"
2.  #define      APP_WEBSITE          "http://www.zh-jieli.com\n"
3.  #define      APP_BASH_IN_PATH     "/sdcard/jilei/active.bash"
4.  #define      APP_BASH_OUT_PATH    "/data/local/tmp/active.bash"
5.
6.  u32 adb_game_active()
7.  {
8.      log_info("%s() %d\n", __func__, __LINE__);
9.      u32 max_len = adb.max_len;;
10.     u8 *adb_buffer = adb.buffer;
11.     //1，启动 app
12.     adb_ex_cmd("am start -n " APP_ACTIVITY_PATH, adb_buffer, max_len);
13.     puts((char *)adb_buffer);
14.     //查找 Error 字符串，如果找到跳转网页下载 app，否则执行 adb 指令
15.     if (strstr((const char *)adb_buffer, "Error") != NULL) {
16.         adb_ex_cmd("am start -a android.intent.action.VIEW -d " APP_WEBSITE, adb_buffer, max_len);
17.         puts((char *)adb_buffer);
18.     } else {
19.         adb_ex_cmd("dd if=" APP_BASH_IN_PATH " of=" APP_BASH_OUT_PATH "\n", adb_buffer, max_len);
20.         puts((char *)adb_buffer);
21.         adb_ex_cmd("chown shell " APP_BASH_OUT_PATH";chmod 777 "APP_BASH_OUT_PATH "\n", adb_buffer, max_len);
22.         puts((char *)adb_buffer);
```

```
23.         adb_ex_cmd("trap \"\" HUP;sh "APP_BASH_OUT_PATH "&\n", adb_buffer, max_len);

24.         puts((char *)adb_buffer);
25.     }
26.
27.     return 0;
28. }
```

4. 按键映射

在这个函数里面处理按键映射。

void key_mapping(const struct keyboard_data_t *k)

2.12 APP - IBEACON

2.12.1 概述

蓝牙 beacon 即蓝牙信标，是通过 BLE 广播特定数据格式的广播包，接收终端通过扫描获取 BLE 广播包信息，再根据协议进行解析。接收终端和蓝牙 beacon 之间的通信，不需要建立蓝牙连接。目前的应用：1.蓝牙信标室内定位，具有简单，低功耗，手机兼容性好的优点。2.消息推送，见于大型商场的推销活动等。3.考勤打卡，身份识别。

支持板级：bd19、br23、br25、bd29、br30

支持芯片：AC632N、AC635N、AC636N、AC631N、AC637N

2.12.2 工程配置

代码工程：apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

1.在 app_config.文件中进行 IBEACON 应用的配置。

```
//app case 选择,只能选 1 个,要配置对应的 board_config.h
#define CONFIG_APP_BEACON                   1 //蓝牙 BLE ibeacon
```

2.12.3 主要代码说明

1.配置应用 case 之后，根据定义好的数据格式，制作信标数据包，该函数位于 le_beacon.c 文件。

```
static u8 make_beacon_packet(u8 *buf, void *packet, u8 packet_type, u8 *web)
{
    switch (packet_type) {
    case IBEACON_PACKET:
    case EDDYSTONE_UID_PACKET:
    case EDDYSTONE_TLM_PACKET:
        memcpy(buf, (u8 *)packet, packet_type);
        break;
    case EDDYSTONE_EID_PACKET:
        memcpy(buf, (u8 *)packet, packet_type);
        break;
    case EDDYSTONE_ETLM_PACKET:
        memcpy(buf, (u8 *)packet, packet_type);
```

```
        break;
    case EDDYSTONE_URL_PACKET:
        packet_type = make_eddystone_url_adv_packet(buf, packet, web);
        break;
    }
    return packet_type;
}
```

2.例如下面是 eddystone_etlm 的数据格式，将该数据制作成数据包后，通过 make_set_adv_data()将数据发送出去，该函数同样位于 le_beacon.c 文件中，同时信标的广播类型应是广播不连接的 ADV_NONCONN_IND 类型。

```
static const EDDYSTONE_ETLM eddystone_etlm_adv_packet = {
    .length = 0x03,
    .ad_type1 = 0x03,
    .complete_list_uuid = 0xabcd,
    .length_last = 0x15,
    .ad_type2 = 0x16,
    .eddystone_uuid = 0xfeaa,
    .frametype = 0x20,
    .tlm_version = 0x01,
    .etml = {
        0, 0, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0
    },          //12 字节加密数据
    .random = 1,                //随机数,要与加密时用到的随机数相同
    .check = 2,                 //AES-EAX 计算出来的校验和
};
```

3.将数据以广播包的形式发送出去，客户端通过扫描获取到信标发出去的数据。

```
static int make_set_adv_data(void)
{
    u8 offset = 0;
```

```
71.     u8 *buf = adv_data;
72.   /* offset += make_eir_packet_val(&buf[offset], offset, HCI_EIR_DATATYPE_FLAGS, 0x06, 1);
    */offset+=make_beacon_packet(&buf[offset],&eddystone_etlm_adv_packet,EDDYSTONE_ETLM_PACKET,
    NULL);
73.   offset+=make_beacon_packet(&buf[offset],&eddystone_eid_adv_packet,EDDYSTONE_EID_PACKET,
    NULL); */
74.     //offset+=make_beacon_packet(&buf[offset],&eddystone_url_adv_packet,EDDYSTONE_URL_PACKET,
    "https://fanyi.baidu.com/");
75.     //offset+=make_beacon_packet(&buf[offset],&eddystone_tlm_adv_packet,EDDYSTONE_TLM_PACKET,
    NULL);
76.     //offset+=make_beacon_packet(&buf[offset],&eddystone_uid_adv_packet,EDDYSTONE_UID_PACKET,
    NULL);
77.  // offset += make_beacon_packet(&buf[offset], &ibeacon_adv_packet, IBEACON_PACKET, NULL);
78.     if (offset > ADV_RSP_PACKET_MAX) {
79.         puts("***adv_data overflow!!!!!!\n");
80.         return -1;
81.     }
```

2.13 APP - Bluetooth Multi connections

2.13.1 概述

支持蓝牙双模透传传输数传，支持蓝牙 LE 多连接功能。CLASSIC 蓝牙使用标准串口 SPP profile 协议，支持可发现搜索连接功能。蓝牙 LE 目前支持 GAP 1 主 1 从，或者 2 主的角色等应用。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

注意不同芯片可以使用 RAM 的空间有差异，有可能会影响性能。

2.13.2 工程配置

代码工程：apps\spp_and_le\board\br30\AC637N_spp_and_le.cbp

（1）app 配置

在工程代码中找到对应的文件(apps\hid\include\app_config.h)进行 APP 选择，本案例中选择翻页器，其结果如下图所示：

```
1. //app case 选择,只能选 1 个,要配置对应的 board_config.h
2. #define CONFIG_APP_MULTI                0 //蓝牙 LE 多连 + spp
```

配置多机个数和加密

```
1. //蓝牙 BLE 配置
2. #define CONFIG_BT_GATT_COMMON_ENABLE       1 //配置使用 gatt 公共模块
3. #define CONFIG_BT_SM_SUPPORT_ENABLE        0 //配置是否支持加密
4. #define CONFIG_BT_GATT_CLIENT_NUM          1 //配置主机 client 个
   数 (app not support)
5. #define CONFIG_BT_GATT_SERVER_NUM          1 //配置从机 server 个数
6. #define CONFIG_BT_GATT_CONNECTION_NUM      (CONFIG_BT_GATT_SERVER_NUM +
   CONFIG_BT_GATT_CLIENT_NUM) //配置连接个数
```

（2）板级选择

接着在文件(apps\hid\board\bd30\board_config.h)下进行对应的板级选择如下:

```
1. /*
```

```
 *  板级配置选择
 */

#define CONFIG_BOARD_AC637N_DEMO
// #define CONFIG_BOARD_AC6373B_DEMO
// #define CONFIG_BOARD_AC6376F_DEMO
// #define CONFIG_BOARD_AC6379B_DEMO

#include "board_ac637n_demo_cfg.h"
#include "board_ac6373b_demo_cfg.h"
#include "board_ac6376f_demo_cfg.h"
#include "board_ac6379b_demo_cfg.h"
#endif
```

//对应的板级头文件，配置是否打开 edr 和 ble 模块

```
#define TCFG_USER_BLE_ENABLE                1   //BLE 功能使能
#define TCFG_USER_EDR_ENABLE                0   //EDR 功能使能
```

2.13.3 主要代码说明

1、主任务处理文件 apps/hid/app_multi_conn.c

(1)APP 注册处理(函数位于 apps/hid/app_multi_conn.c)

在系统进行初始化的过程中，根据以下信息进行 APP 注册。执行的大致流程为：REGISTER_APPLICATION--->state_machine--->app_start()--->sys_key_event_enable();这条流程主要进行设备的初始化设置以及一些功能使能。

REGISTER_APPLICATION--->event_handler--->app_key_event_handler()--->app_key_deal_test();这条流程在 event_handler 之下有多个 case,上述选择按键事件的处理流程进行代码流说明，主要展示按键事件发生时，程序的处理流程。

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

```
/*
 * 注册 AT Module 模式
 */
REGISTER_APPLICATION(app_multi) = {
    .name  = "multi_conn",
    .action = ACTION_MULTI_MAIN,
    .ops  = &app_multi_ops,
    .state  = APP_STA_DESTROY,
};

//-----------------------
//system check go sleep is ok
static u8 app_state_idle_query(void)
{
    return !is_app_active;
}

REGISTER_LP_TARGET(app_state_lp_target) = {
    .name = "app_state_deal",
    .is_idle = app_state_idle_query,
};
```

(2) APP 状态机

状态机有 create，start，pause，resume，stop，destory 状态，根据不同的状态执行对应的分支。APP 注册后进行初始运行，进入 APP_STA_START 分支，开始 APP 运行。

```
static int state_machine(struct application *app, enum app_state state, struct intent *it)
{   switch (state) {
    case APP_STA_CREATE:
        break;
    case APP_STA_START:
        if (!it) {
            break;      }
        switch (it->action) {
```

```
9.          case ACTION_MULTI_MAIN:
10.             app_start();
```

进入 app_start()函数后进行对应的初始化，时钟初始化，模式选择，蓝牙初始，低功耗初始化，以及外部事件使能。

```
1.  static void app_start()
2.  {
3.      log_info("==========================================");
4.      log_info("----------------------multi_conn demo----------------------");
5.      log_info("==========================================");
6.
```

(3) APP 事件处理机制

1.事件的定义(代码位于 Headers\include_lib\system\even.h 中)

```
1.  struct sys_event {
2.      u16 type;
3.      u8 consumed;
4.      void *arg;
5.      union {
6.          struct key_event key;
7.          struct axis_event axis;
8.          struct codesw_event codesw;
```

（4）事件的产生（include_lib\system\event.h）

```
void sys_event_notify(struct sys_event *e);
```

事件通知函数,系统有事件发生时调用此函数。

（5）事件的处理(app_mutil.c)

函数执行的大致流程为：evevt_handler()--->app_key_event_handler()--->app_key_deal_test().

```
1.  static int event_handler(struct application *app, struct sys_event *event)
2.
```

```
1.  static void app_key_event_handler(struct sys_event *event)
```

```
1.  static void app_key_deal_test(u8 key_type, u8 key_value)
```

2、**LE 公共处理** ble_multi_conn.c

配置 Gatt common 模块初始化，蓝牙初始化等操作。

3、**LE 的 GATT server 实现** ble_multi_peripheral.c

配置 Gatt Server 端的广播、连接、事件处理等。

4、**LE 的 GATT client 实现** ble_multi_central.c

配置 Gatt Cleint 端的搜索、连接、事件处理等。

2.14 APP - Bluetooth Dual-Mode AT Moudle (char)

2.14.1 概述

主要功能是在普通数传 BLE 和 EDR 的基础上增加了由上位机或其他 MCU 可以通过 UART 对接蓝牙芯片进行基本配置、状态获取、控制扫描、连接断开以及数据收发等操作。

AT 控制透传主从多机模式。

定义一套串口的控制协议，具体请查看协议文档《蓝牙 AT_CHAR 协议》。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

注意不同芯片可以使用 RAM 的空间有差异，有可能会影响性能。

2.14.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631N_spp_and_le.cbp

（1）先配置板级 board_config.h 和对应配置文件中蓝牙双模使能

```
/*
 *  板级配置选择
 */
#define CONFIG_BOARD_AC631N_DEMO        // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER

#include "board_ac631n_demo_cfg.h"
```

（2）配置对应的 board_acxxx_demo_cfg.h 文件使能 BLE 以 board_ac630x_demo_cfg.h 为例

```
#define TCFG_USER_BLE_ENABLE                1  //BLE 功能使能
#define TCFG_USER_EDR_ENABLE                0  //EDR 功能使能
```

（3）配置 app_config.h，选择 CONFIG_APP_AT_CHAR_COM

```
#define CONFIG_APP_AT_CHAR_COM                  1//AT com 字符串格式命令
```

2.14.3 主要说明代码<at_char_cmds.c>

1.命令包头

```
static const char at_head_at_cmd[]      = "AT+";
```

```
static const char at_head_at_chl[]        = "AT>";

 static const str_info_t at_head_str_table[] = {
      INPUT_STR_INFO(STR_ID_HEAD_AT_CMD, at_head_at_cmd),
      INPUT_STR_INFO(STR_ID_HEAD_AT_CHL, at_head_at_chl),
};

```

2.命令类型:

```
static const char at_str_gver[]           = "GVER";
static const char at_str_gcfgver[]        = "GCFGVER";
static const char at_str_name[]           = "NAME";
static const char at_str_lbdaddr[]        = "LBDADDR";
static const char at_str_baud[]           = "BAUD";

static const char at_str_adv[]            = "ADV";
static const char at_str_advparam[]       = "ADVPARAM";
static const char at_str_advdata[]        = "ADVDATA";
static const char at_str_srdata[]         = "SRDATA";
static const char at_str_connparam[]      = "CONNPARAM";

static const char at_str_scan[]           = "SCAN";
static const char at_str_targetuuid[]     = "TARGETUUID";
static const char at_str_conn[]           = "CONN";
static const char at_str_disc[]           = "DISC";
static const char at_str_ota[]            = "OTA";

static const str_info_t at_cmd_str_table[] = {
      INPUT_STR_INFO(STR_ID_GVER, at_str_gver),
      INPUT_STR_INFO(STR_ID_GCFGVER, at_str_gcfgver),
      INPUT_STR_INFO(STR_ID_NAME, at_str_name),
      INPUT_STR_INFO(STR_ID_LBDADDR, at_str_lbdaddr),
      INPUT_STR_INFO(STR_ID_BAUD, at_str_baud),
```

```
61.
62.     INPUT_STR_INFO(STR_ID_ADV, at_str_adv),
63.     INPUT_STR_INFO(STR_ID_ADVPARAM, at_str_advparam),
64.     INPUT_STR_INFO(STR_ID_ADVDATA, at_str_advdata),
65.     INPUT_STR_INFO(STR_ID_SRDATA, at_str_srdata),
66.     INPUT_STR_INFO(STR_ID_CONNPARAM, at_str_connparam),
67.
68.     INPUT_STR_INFO(STR_ID_SCAN, at_str_scan),
69.     INPUT_STR_INFO(STR_ID_TARGETUUID, at_str_targetuuid),
70.     INPUT_STR_INFO(STR_ID_CONN, at_str_conn),
71.     INPUT_STR_INFO(STR_ID_DISC, at_str_disc),
72.     INPUT_STR_INFO(STR_ID_OTA, at_str_ota),
73.
74. //    INPUT_STR_INFO(, ),
75. //    INPUT_STR_INFO(, ),
76. };
```

3. 串口收数中断,

进入收数中断之前会进行初步校验，数据结尾不是'\r'，则无法唤醒中断

```
77. static void at_packet_handler(u8 *packet, int size)
```

4. AT 命令解析

```
78.             /*比较数据包头*/
79.     str_p=at_check_match_string(parse_pt,
    parse_size,at_head_str_table,sizeof(at_head_str_table));
80.     if (!str_p)
81.     {
82.         log_info("###1unknow at_head:%s", packet);
83.         AT_STRING_SEND(at_str_err);
84.         return;
85.     }
86.     parse_pt    += str_p->str_len;
```

```
87.         parse_size -= str_p->str_len;
88.
89.         /*普通命令*/
90.         if(str_p->str_id == STR_ID_HEAD_AT_CMD)
91.         {
92.             /*比较命令*/
93.             str_p=at_check_match_string(parse_pt,
    parse_size,at_cmd_str_table,sizeof(at_cmd_str_table));
94.             if (!str_p)
95.             {
96.                 log_info("###2unknow at_cmd:%s", packet);
97.                 AT_STRING_SEND(at_str_err);
98.                 return;
99.             }
100.
101.              parse_pt += str_p->str_len;
102.              parse_size -= str_p->str_len;
103.              /*判断当前是命令类型,查询或设置命令*/
104.              if(parse_pt[0] == '=')
105.              {
106.                  operator_type = AT_CMD_OPT_SET;
107.              }
108.              else if(parse_pt[0] == '?')
109.              {
110.                  operator_type = AT_CMD_OPT_GET;
111.              }
112.              parse_pt++;
113.          }
114.
115.          /*通道切换命令*/
116.          else if(str_p->str_id == STR_ID_HEAD_AT_CHL)
117.          {
118.              operator_type = AT_CMD_OPT_SET;
```

```
119.        }
120.
121. //      if(operator_type == AT_CMD_OPT_NULL)
122. //      {
123. //          AT_STRING_SEND(at_str_err);
124. //          log_info("###3unknow operator_type:%s", packet);
125. //          return;
126. //      }
127.
128.
129.        log_info("str_id:%d", str_p->str_id);
130.
131.        /*解析并返回命令参数 par*/
132.        par = parse_param_split(parse_pt,',','\r');
133.
134.        log_info("\n par->data: %s",par->data);
135.
136.        /*命令处理与响应*/
137.        switch (str_p->str_id)
138.        {
139.            case STR_ID_HEAD_AT_CHL:
140.                log_info("STR_ID_HEAD_AT_CHL\n");
141.            break;
142.        .....................................
```

5.命令的参数获取与遍历

```
143. par = parse_param_split(parse_pt,',','\r');
144. /*parameter
145. *packet:  参数指针
146. split_char:  参数之间的间隔符，一般是’,’
147. end_char:   参数的结束符，一般是’\r’
148. */
149. static at_param_t *parse_param_split(const u8 *packet,u8 split_char,u8 end_char)
```

```
150. {
151.     u8 char1;
152.     int i = 0;
153.     at_param_t *par = parse_buffer;
154.
155.     if (*packet == end_char) {
156.         return NULL;
157.     }
158.
159.     log_info("%s:%s",__FUNCTION__,packet);
160.
161.     par->len = 0;
162.
163.     while (1) {
164.         char1 = packet[i++];
165.         if (char1 == end_char) {
166.             par->data[par->len] = 0;
167.             par->next_offset = 0;
168.             break;
169.         } else if (char1 == split_char) {
170.             par->data[par->len] = 0;
171.             par->len++;
172.             par->next_offset = &par->data[par->len] - parse_buffer;
173.
174.             //init next par
175.             par = &par->data[par->len];
176.             par->len = 0;
177.         } else {
178.             par->data[par->len++] = char1;
179.         }
180.
181.         if (&par->data[par->len] - parse_buffer >= PARSE_BUFFER_SIZE) {
182.             log_error("parse_buffer over");
```

```
                par->next_offset = 0;
                break;
            }
        }
        return (void *)parse_buffer;
}
```

当有多个参数是，需要遍历获取，以连接参数为例

```
        while (par) {   //遍历所有参数
            log_info("len=%d,par:%s", par->len, par->data);
            conn_param[i] = func_char_to_dec(par->data, '\0');   //遍历获取连接参数
            if (par->next_offset) {
                par = AT_PARAM_NEXT_P(par);
            } else {
                break;
            }
            i++;
        }
```

6.默认信息配置

```
#define G_VERSION "BR30_2021_01_31"
#define CONFIG_VERSION "2021_02_04"   /*版本信息*/
u32 uart_baud = 115200;   /*默认波特率*/
char dev_name_default[] = "jl_test";   /*默认 dev name*/
```

7.在代码中添加新的命令(以查询、设置波特率为例)

（1）添加波特率枚举成员

```
enum {
    STR_ID_NULL = 0,
    STR_ID_HEAD_AT_CMD,
    STR_ID_HEAD_AT_CHL,

    STR_ID_OK = 0x10,
```

```
    STR_ID_ERROR,

    STR_ID_GVER = 0x20,
    STR_ID_GCFGVER,
    STR_ID_NAME,
    STR_ID_LBDADDR,
    STR_ID_BAUD,        /*波特率成员*/

    STR_ID_ADV,
    STR_ID_ADVPARAM,
    STR_ID_ADVDATA,
    STR_ID_SRDATA,
    STR_ID_CONNPARAM,

    STR_ID_SCAN,
    STR_ID_TARGETUUID,
    STR_ID_CONN,
    STR_ID_DISC,
    STR_ID_OTA,
//  STR_ID_,
//  STR_ID_,
};
```

(2)添加波特率命令类型

```
static const char at_str_gver[]          = "GVER";
static const char at_str_gcfgver[]       = "GCFGVER";
static const char at_str_name[]          = "NAME";
static const char at_str_lbdaddr[]       = "LBDADDR";
static const char at_str_baud[]          = "BAUD"; /*波特率命令类型*/

static const char at_str_adv[]           = "ADV";
static const char at_str_advparam[]      = "ADVPARAM";
```

```
239. static const char at_str_advdata[]        = "ADVDATA";
240. static const char at_str_srdata[]         = "SRDATA";
241. static const char at_str_connparam[]     = "CONNPARAM";
242.
243. static const char at_str_scan[]            = "SCAN";
244. static const char at_str_targetuuid[]    = "TARGETUUID";
245. static const char at_str_conn[]            = "CONN";
246. static const char at_str_disc[]            = "DISC";
247. static const char at_str_ota[]             = "OTA";
```

(3)添加命令到命令列表

```
248. static const str_info_t at_cmd_str_table[] = {
249.         INPUT_STR_INFO(STR_ID_GVER, at_str_gver),
250.         INPUT_STR_INFO(STR_ID_GCFGVER, at_str_gcfgver),
251.         INPUT_STR_INFO(STR_ID_NAME, at_str_name),
252.         INPUT_STR_INFO(STR_ID_LBDADDR, at_str_lbdaddr),
253.         INPUT_STR_INFO(STR_ID_BAUD, at_str_baud),   /*波特率命令*/
254.
255.         INPUT_STR_INFO(STR_ID_ADV, at_str_adv),
256.         INPUT_STR_INFO(STR_ID_ADVPARAM, at_str_advparam),
257.         INPUT_STR_INFO(STR_ID_ADVDATA, at_str_advdata),
258.         INPUT_STR_INFO(STR_ID_SRDATA, at_str_srdata),
259.         INPUT_STR_INFO(STR_ID_CONNPARAM, at_str_connparam),
260.
261.         INPUT_STR_INFO(STR_ID_SCAN, at_str_scan),
262.         INPUT_STR_INFO(STR_ID_TARGETUUID, at_str_targetuuid),
263.         INPUT_STR_INFO(STR_ID_CONN, at_str_conn),
264.         INPUT_STR_INFO(STR_ID_DISC, at_str_disc),
265.         INPUT_STR_INFO(STR_ID_OTA, at_str_ota),
266.
267. //      INPUT_STR_INFO(, ),
268. //      INPUT_STR_INFO(, ),
269. };
```

(4) 在 at_packet_handler 函数中添加命令的处理与响应

```
270.            case STR_ID_BAUD:
271.                log_info("STR_ID_BAUD\n");
272.                {
273.                    if(operator_type == AT_CMD_OPT_SET) //设置波特率
274.                    {
275.                        uart_baud = func_char_to_dec(par->data, '\0');
276.                        if(uart_baud==9600||uart_baud==19200||uart_baud==38400||uart_baud==115200||
277.                            uart_baud==230400||uart_baud==460800||uart_baud==921600)
278.                        {
279.                            AT_STRING_SEND("OK"); /*返回响应*/
280.                            ct_uart_init(uart_baud);
281.                        }
282.                        else{     //TODO 返回错误码
283.
284.
285.                        }
286.                    }
287.                    else{                                   //读取波特率
288.
289.                        sprintf( buf, "+BAUD:%d", uart_baud);
290.                        at_cmd_send(buf, strlen(buf)); /*返回波特率数据*/
291.                        AT_STRING_SEND("OK"); /*返回响应*/
292.                    }
293.                }
294.            break;
```

8.串口发数 api，用于发送响应信息

```
295. /*
296. parameter
297. packet: 数据包
```

```
298. size: 数据长度
299. */
300. at_uart_send_packet(const u8 *packet, int size);
```

用于回复带”\r\n”的响应

```
301. void at_cmd_send(const u8 *packet, int size)
302. {
303.     at_uart_send_packet(at_str_enter, 2);
304.     at_uart_send_packet(packet, size);
305.     at_uart_send_packet(at_str_enter, 2);
306. }
```

2.15 APP - Nonconn_24G

2.15.1 概述

主要功能是在蓝牙 BLE 架构基础上自定义不可见的 2.4G 非连接模式数据传输示例。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.15.2 工程配置

代码工程：apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

（1）配置 app 选择(apps\spp_and_le\include\app_config.h)，如下图选择对应的应用示例

```
1. //app case 选择,只能选 1 个,要配置对应的 board_config.h
1. #define CONFIG_APP_NONCONN_24G           1 //2.4G 非连接收发
```

（2）先配置板级 board_config.h(apps\spp_and_le\board\brxx\board_config.h)，选择对应的开发板，可以使用默认的板级

```
1. #define CONFIG_BOARD_AC632N_DEMO
2. // #define CONFIG_BOARD_AC6321A_DEMO
```

只需要使能 BLE 就可以了

```
1. #define TCFG_USER_BLE_ENABLE                    1    //BLE 功能使能
2. #define TCFG_USER_EDR_ENABLE                    0    //EDR 功能使能
```

2.15.3 数据收发模块

实现代码文件在 ble_24g_deal.c

(1)主要配置宏如下：

```
1. //-----------------------------------------------------
2. #define CFG_RF_24G_CODE_ID    0x13 // 24g 识别码(24bit),发送接收都要匹配
3. //-----------------------------------------------------
4. //配置收发角色
5. #define CONFIG_TX_MODE_ENABLE     1 //发射器
6. #define CONFIG_RX_MODE_ENABLE     0 //接收器
7. //-----------------------------------------------------
```

```
//TX 发送配置
#define TX_DATA_COUNT               3  //发送次数,决定 os_time_dly 多久
#define TX_DATA_INTERVAL            20 //发送间隔>=20ms

#define ADV_INTERVAL_VAL            ADV_SCAN_MS(TX_DATA_INTERVAL)//
#define RSP_TX_HEAD                 0xff
//-------------------------------------------------------
//RX 接收配置
//搜索类型
#define SET_SCAN_TYPE       SCAN_ACTIVE
//搜索 周期大小
#define SET_SCAN_INTERVAL   ADV_SCAN_MS(200)//
//搜索 窗口大小
#define SET_SCAN_WINDOW     ADV_SCAN_MS(200)//
```

（2）发射器发送接口

```
//发送数据, len support max is 60
int ble_tx_send_data(const u8 *data, u8 len)
```

（3）接收器接收接口

```
void ble_rx_data_handle(const u8 *data, u8 len)
```

2.16 APP - CONN_24G

2.16.1 概述

主要功能是在蓝牙 BLE 架构基础上自定义不可见的 2.4G 连接模式数据传输示例。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.16.2 工程配置

代码工程：apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

（1）配置 app 选择(apps\spp_and_le\include\app_config.h)，如下图选择对应的应用示例

```
2.  //app case 选择,只能选 1 个,要配置对应的 board_config.h
2.  #define CONFIG_APP_CONN_24G          1 //基于 BLE 的 2.4g,板级只需要开 BLE
```

（2）先配置板级 board_config.h(apps\spp_and_le\board\brxx\board_config.h)，选择对应的开发板，可以使用默认的板级

```
1.  #define CONFIG_BOARD_AC632N_DEMO
2.  // #define CONFIG_BOARD_AC6321A_DEMO
3.  // #define CONFIG_BOARD_AC6323A_DEMO
4.  // #define CONFIG_BOARD_AC6328A_DEMO
5.  // #define CONFIG_BOARD_AC6328B_DONGLE
6.  // #define CONFIG_BOARD_AC6329B_DEMO
7.  // #define CONFIG_BOARD_AC6329C_DEMO
8.  // #define CONFIG_BOARD_AC6329E_DEMO
9.  // #define CONFIG_BOARD_AC6329F_DEMO
```

只需要使能 BLE 就可以了

```
3.  #define TCFG_USER_BLE_ENABLE                    1   //BLE 功能使能
4.  #define TCFG_USER_EDR_ENABLE                    0   //EDR 功能使能
```

（3）设置 2.4G ID(apps\spp_and_le\examples\conn_24g\app_conn_24g.c)：修改 CFG_RF_24G_CODE_ID 为 0x23 即可

```
1.  //2.4G 模式: 0---ble, 非 0---2.4G 配对码
2.  /* #define CFG_RF_24G_CODE_ID          (0) //<=24bits */
```

```
3. #define CFG_RF_24G_CODE_ID          (0x23) //<=24bits
```

2.16.3 设置 2.4G 物理层

设置物理层可设置传输数率，目前蓝牙协议包含的物理层，我司都是支持的（包含 1M、2M、CODED S2、CODED S8），具体设置如下：

设置为 1M 物理层收发（1M 不需要设置 S2/S8，所以 SELECT_CODED_S2_OR_S8 随意设置）：

```
1. //选择物理层
2. #define  SELECT_PHY                 CONN_SET_1M_PHY//1M:CONN_SET_1M_PHY 2M:CONN_SET_
   2M_PHY CODED:CONN_SET_CODED_PHY
3. //选择 CODED 类型:S2 or S8
4. #define  SELECT_CODED_S2_OR_S8      CONN_SET_PHY_OPTIONS_S2//S2:CONN_SET_PHY_OPTIONS
   _S2 S8:CONN_SET_PHY_OPTIONS_S8
```

设置为 2M 物理层收发（2M 不需要设置 S2/S8，所以 SELECT_CODED_S2_OR_S8 随意设置）：

```
1. //选择物理层
2. #define  SELECT_PHY                 CONN_SET_2M_PHY//1M:CONN_SET_1M_PHY 2M:CONN_SET_
   2M_PHY CODED:CONN_SET_CODED_PHY
3. //选择 CODED 类型:S2 or S8
4. #define  SELECT_CODED_S2_OR_S8      CONN_SET_PHY_OPTIONS_S2//S2:CONN_SET_PHY_OPTIONS
   _S2 S8:CONN_SET_PHY_OPTIONS_S8
```

设置为 CODED S2 物理层收发：

```
1. //选择物理层
2. #define  SELECT_PHY                 CONN_SET_CODED_PHY//1M:CONN_SET_1M_PHY 2M:CONN_S
   ET_2M_PHY CODED:CONN_SET_CODED_PHY
3. //选择 CODED 类型:S2 or S8
4. #define  SELECT_CODED_S2_OR_S8      CONN_SET_PHY_OPTIONS_S2//S2:CONN_SET_PHY_OPTIONS
   _S2 S8:CONN_SET_PHY_OPTIONS_S8
```

设置为 CODED S8 物理层收发：

```
1. //选择物理层
2. #define  SELECT_PHY                 CONN_SET_CODED_PHY//1M:CONN_SET_1M_PHY 2M:CONN_S
   ET_2M_PHY CODED:CONN_SET_CODED_PHY
3. //选择 CODED 类型:S2 or S8
```

```
#define  SELECT_CODED_S2_OR_S8      CONN_SET_PHY_OPTIONS_S8//S2:CONN_SET_PHY_OPTIONS_S2 S8:CONN_SET_PHY_OPTIONS_S8
```

2.16.4 数据发送模块

（1）在(app_conn_24g.c)中打开 CONN_24G_KEEP_SEND_EN 使能（数据发送使能）：

```
#define  CONN_24G_KEEP_SEND_EN      1   //just for 2.4gtest keep data
```

代码会在连接之后打开发送测试、在断开连接关闭发送测试：

```
    case BLE_ST_CREATE_CONN:
        /*
         * 蓝牙设备已经连接允许主机发数据
         */
#if CONN_24G_KEEP_SEND_EN && CONFIG_BT_GATT_CLIENT_NUM
        if (conn_24g_phy_test_timer_id == 0) {
            log_info("OPEN CONN_24G_KEEP_SEND_EN\n");
            conn_24g_phy_test_timer_id = sys_timer_add(NULL, conn_24g_phy_test, 200);
        }
```

```
    case BLE_ST_DISCONN:
        /*
         * 蓝牙断开连接,主机设置关闭数据发送
         */
#if CONN_24G_KEEP_SEND_EN && CONFIG_BT_GATT_CLIENT_NUM
        log_info("CLOSE CONN_24G_KEEP_SEND_EN\n");
        sys_timeout_del(conn_24g_phy_test_timer_id);
        conn_24g_phy_test_timer_id = 0;
#endif
```

2.16.5 绑定、解绑使用

（1）绑定：

原理：普通的蓝牙设备，从机连接一个设备之后，会记住对端的地址，但是下次还是可以和新

的设备进行连接；这是因为没有做绑定的原因。由于考虑到客户可能需要绑定功能，于是加入；从机连接一个设备之后，保存配对信息，当重新广播时若从机有配对信息则从机将广播设置为定向广播（定向广播具有定向连接性）。主机连接一个从机设备之后，保存从机配对信息，当重新扫描时若主机有配对信息则主机只扫描具有定向广播的从机，并且只连接具有与主机记录配对信息相同的从机

从机设置(ble_24g_server.c)：有配对信息设置为定向广播。

```
1.  //配置定向广播信息
2.  conn_24g_server_adv_config.adv_type = ADV_DIRECT_IND_LOW;
3.  conn_24g_server_adv_config.adv_interval = PAIR_DIRECT_LOW_ADV_INTERVAL;
4.  log_info("===DIRECT_ADV address:");
5.  put_buf(conn_24g_server_adv_config.direct_address_info, 7);
```

主机设置(ble_24g_client.c)：连接后只识别定向广播，并且重新传入搜索配置

```
1.  //连接后识别定向广播,不然会导致从机一断开连接,主机马上又连接
2.  ble_gatt_client_set_search_config(&conn_24g_central_bond_config);
```

（2）解绑：

原理：有绑定肯定就有解绑需求，本设计设置 AD key 0 长按抬起解绑。将主从机的配对信息删除即可完成解绑，解绑之后打开主从机的扫描和广播为下次绑定做好准备。

主机(app_conn_24g.c)：删除配对信息并设置 scan：

```
1.  memset(&client_pair_bond_info[0], 0, 8);
2.  conn_24g_pair_vm_do(client_pair_bond_info, sizeof(client_pair_bond_info), 1);
3.  log_info("clear client pear info!");
4.  put_buf(&client_pair_bond_info, 8);
5.  ble_gatt_client_scan_enable(0);
6.  ble_gatt_client_disconnect_all();
7.  conn_24g_central_init();
8.  ble_gatt_client_scan_enable(1);
```

从机(app_conn_24g.c)：删除配对信息并设置 adv：

```
1.  memset(&pair_bond_info[0], 0, 8);
2.  conn_24g_pair_vm_do(pair_bond_info, sizeof(pair_bond_info), 1);
3.  log_info("clear server pear info!");
4.  put_buf(&pair_bond_info, 8);
5.  //关闭模块重新打开
```

```
ble_gatt_server_adv_enable(0);
ble_gatt_server_disconnect_all();
/* ble_comm_disconnect(0x50); */
conn_24g_server_init();
ble_gatt_server_adv_enable(1);
```

2.17 APP - Tecent LL

2.17.1 概述

本案例用于实现腾讯连连协议，使用腾讯连连微信小程序与设备连接后可以对设备进行控制。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.17.2 工程配置

代码工程：apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

（2）配置 app 选择(apps\spp_and_le\include\app_config.h)，如下图选择对应的应用示例。

```
1. //app case 选择,只能选 1 个,要配置对应的 board_config.h
2. #define CONFIG_APP_LL_SYNC                0 //腾讯连连/
```

（3）配置板级蓝牙设置（apps\spp_and_le\board\brxx\board_acxxxx_demo.cfg）,只开 BLE 不开 EDR

```
1. //*************************************************************//
2. //                      蓝牙配置                                 //
3. //*************************************************************//
4. #define TCFG_USER_TWS_ENABLE                    0   //tws 功能使能
5. #define TCFG_USER_BLE_ENABLE                    1   //BLE 功能使能
6. #define TCFG_USER_EDR_ENABLE                    0   //EDR 功能使能
```

2.17.3 模块开发

详细参考 《腾讯连连开发文档》。

2.18 APP - TUYA

2.18.1 概述

本案例用于实现涂鸦协议，使用涂鸦智能或者涂鸦云测 APP 与设备连接后可以对设备进行控制。

支持的板级： bd29、br25、br23、br30、bd19、br34

支持的芯片： AC631N、AC636N、AC635N、AC637N、AC632N、AC638N

2.18.2 工程配置

（1）代码工程：apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

（2）配置 app 选择(apps\spp_and_le\include\app_config.h)，如下图选择对应的应用示例。

```
3.  //app case 选择,只能选 1 个,要配置对应的 board_config.h
4.  #define CONFIG_APP_TUYA                 0 //涂鸦协议/
```

（3）配置板级蓝牙设置（apps\spp_and_le\board\brxx\board_acxxxx_demo.cfg）,只开 BLE 不开 EDR

```
7.  //*************************************************************//
8.  //                      蓝牙配置                                //
9.  //*************************************************************//
10. #define TCFG_USER_TWS_ENABLE                    0   //tws 功能使能
11. #define TCFG_USER_BLE_ENABLE                    1   //BLE 功能使能
12. #define TCFG_USER_EDR_ENABLE                    0   //EDR 功能使能
```

2.18.3 模块开发

详细参考 《涂鸦协议开发文档》。

2.19APP - Voice remote control

2.19.1 概述

本案例用于 hid 语音遥控功能传送功能，支持的编码格式。

IMA ADPCM、Speex、Opus、SBC、mSBC、LC3

支持的板级： br30、br34

支持的芯片： AC637N、AC638N

2.19.2 工程配置

（1）代码工程：apps\hid\board\bdxx\AC63xN_hid.cbp

（2）配置 app 选择(apps\spp_and_le\include\app_config.h)，如下图选择对应的应用示例。

```
5.  //app case 选择,只能选 1 个,要配置对应的 board_config.h
6.  #define CONFIG_APP_REMOTE_CONTROL          0//语音遥控
```

（3）配置板级蓝牙设置（apps\spp_and_le\board\brxx\board_acxxxx_demo.cfg）,只开 BLE 不开 EDR

```
13. //*************************************************************//
14. //                      蓝牙配置                                 //
15. //*************************************************************//
16. #define TCFG_USER_TWS_ENABLE                    0   //tws 功能使能
17. #define TCFG_USER_BLE_ENABLE                    1   //BLE 功能使能
18. #define TCFG_USER_EDR_ENABLE                    0   //EDR 功能使能
```

2.19.3 模块开发

示例是在原来 app_keyboard 的示例上，新增支持编码应用。编码格式支持 IMA ADPCM、Speex、Opus、SBC、mSBC、LC3 等格式。

通用编码接口在 audio_codec_demo.c 文件里面，Audio 编码功能的配置使用，参考（5.2 Audio 使用） 里面的 **<通用编码接口的使用>** 章节说明。

2.20 GATT COMMON

2.20.1 概述

主要描述新版本 SDK(v2.0.0 及以后)在 COMMON 目录新增 GATT 公共模块，主要处理 GATT 层跟蓝牙协议栈的功能信息交换，GATT 跟 apps 层的功能信息交换；有效使 apps 不需要太依赖协议栈接口的直接调用，只需要关注 GATT 接口模块配置以及控制就可以。

2.20.2 代码说明

目录路径：apps\common\third_party_profile\jieli\gatt_common

文件信息：

```
1. le_gatt_client.c
2. le_gatt_common.c
3. le_gatt_common.h
4. le_gatt_server.c
```

模块主要由 app_config.h 里面的 gatt 配置管理，如下：

```
1. //蓝牙 BLE 配置
2. #define CONFIG_BT_GATT_COMMON_ENABLE       1 //配置使用 gatt 公共模块
3. #define CONFIG_BT_SM_SUPPORT_ENABLE        0 //配置是否支持加密
4. #define CONFIG_BT_GATT_CLIENT_NUM          0 //配置主机 client 个数 (app not support)
5. #define CONFIG_BT_GATT_SERVER_NUM          1 //配置从机 server 个数
6. #define CONFIG_BT_GATT_CONNECTION_NUM      (CONFIG_BT_GATT_SERVER_NUM + CONFIG_BT_GA
   TT_CLIENT_NUM) //配置连接个数
```

GATT 主从角色各自单独最多支持 8 个，若同时使用，连接最多支持调链路。

le_gatt_common.c

------主要执行蓝牙协议栈的初始化驱动，包含 GATT、ATT 和 SM 等配置初始化，驱动 GATT server 和 GATT client 模块实现，分发两个模块的消息处理，支持多机 GATT 多机连接管理。提供一些 GATT 通过接口调用，例如 ATT 数据发送，蓝牙链路断开等。

提供的接口列表，具体接口定义看代码注释说明：

```
1. //common
```

```
u32  ble_comm_cbuffer_vaild_len(u16 conn_handle);
int ble_comm_att_send_data(u16 conn_handle, u16 att_handle, u8 *data, u16 len, att_op_type_e op_type);
bool ble_comm_att_check_send(u16 conn_handle, u16 pre_send_len);
const char *ble_comm_get_gap_name(void);
int ble_comm_disconnect(u16 conn_handle);
u8 ble_comm_dev_get_handle_state(u16 handle, u8 role);
void ble_comm_dev_set_handle_state(u16 handle, u8 role, u8 state);
void ble_comm_register_state_cbk(void (*cbk)(u16 handle, u8 state));
s8 ble_comm_dev_get_index(u16 handle, u8 role);
s8 ble_comm_dev_get_idle_index(u8 role);
u8 ble_comm_dev_get_handle_role(u16 handle);
u16 ble_comm_dev_get_handle(u8 index, u8 role);
void ble_comm_set_config_name(const char *name_p, u8 add_ext_name);
void ble_comm_init(const gatt_ctrl_t *control_blk);
void ble_comm_exit(void);
void ble_comm_module_enable(u8 en);
int ble_comm_set_connection_data_length(u16 conn_handle, u16 tx_octets, u16 tx_time);
int ble_comm_set_connection_data_phy(u16 conn_handle, u8 tx_phy, u8 rx_phy);
```

le_gatt_server.c

------主要执行 GATT server 角色的操作，例如配置 profile，开广播，连接参数更新流程、ota 通用操作，协议栈的事件处理，多机机制管理等。apps 层只需执行配置操作，就可以实现驱动模块实现基本功能。

提供的接口列表，具体接口定义看代码注释说明：

```
//server
void ble_gatt_server_init(gatt_server_cfg_t *server_cfg);
void ble_gatt_server_exit(void);
ble_state_e ble_gatt_server_get_work_state(void);
ble_state_e ble_gatt_server_get_connect_state(u16 conn_handle);
int ble_gatt_server_adv_enable(u32 en);
void ble_gatt_server_module_enable(u8 en);
```

```
8.  void ble_gatt_server_disconnect_all(void);
9.  int  ble_gatt_server_connetion_update_request(u16 conn_handle, const struct conn_update_param_t *update_table, u16 table_count);
10. int ble_gatt_server_characteristic_ccc_set(u16 conn_handle, u16 att_handle, u16 ccc_config);
11. u16 ble_gatt_server_characteristic_ccc_get(u16 conn_handle, u16 att_handle);
12. void ble_gatt_server_set_update_send(u16 conn_handle, u16 att_handle, u8 att_handle_type);
13. void ble_gatt_server_receive_update_data(void *priv, void *buf, u16 len);
14. void ble_gatt_server_set_adv_config(adv_cfg_t *adv_cfg);
15. void ble_gatt_server_set_profile(const u8 *profile_table, u16 size);
```

le_gatt_client.c

------主要执行 GATT client 角色的操作，例如配置 scan 参数，匹配设备扫描连接，匹配 profile 搜索连接，协议栈的事件处理；多机机制管理等。apps 层只需执行配置操作，就可以实现驱动模块实现基本功能。

提供的接口列表，具体接口定义看代码注释说明：

```
1.  //client
2.  void ble_gatt_client_init(gatt_client_cfg_t *client_cfg);
3.  void ble_gatt_client_exit(void);
4.  void ble_gatt_client_set_scan_config(scan_conn_cfg_t *scan_conn_cfg);
5.  void ble_gatt_client_set_search_config(gatt_search_cfg_t *gatt_search_cfg);
6.  ble_state_e ble_gatt_client_get_work_state(void);
7.  ble_state_e ble_gatt_client_get_connect_state(u16 conn_handle);
8.  int ble_gatt_client_create_connection_request(u8 *address, u8 addr_type, int mode);
9.  int ble_gatt_client_create_connection_cannel(void);
10. int ble_gatt_client_scan_enable(u32 en);
11. void ble_gatt_client_module_enable(u8 en);
12. void ble_gatt_client_disconnect_all(void);
```

le_gatt_common.1

------模块配置结构定义，提供接口定义。

2.21 HiLink

2.21.1 概述

本示例实现华为 HiLink 协议套餐 4 (harmony connect suit4) 方式接入，实例以智能灯的形态展现，支持智慧生活 APP 连接和控制蓝牙设备；可控制开关灯，可在 APP 端执行 OTA 升级设备功能。

支持通用芯片系列

2.21.2 工程配置

（1） 代码工程： apps\spp_and_le\board\bdxx\AC63xN_spp_and_le.cbp

（2） 配置 app 选择(apps\spp_and_le\include\app_config.h)， 选择对应的应用示例

```
1. //app case 选择,只能选 1 个,要配置对应的 board_config.h
2. #define CONFIG_APP_HILINK                    1 //华为协议
```

（3）配置板级蓝牙设置（apps\spp_and_le\board\brxx\board_acxxxx_demo.cfg），只开 BLE 不开 EDR

```
1. //*********************************************************************//
2. //                                蓝牙配置                             //
3. //*********************************************************************//
4. #define TCFG_USER_TWS_ENABLE                 0   //tws 功能使能
5. #define TCFG_USER_BLE_ENABLE                 1   //BLE 功能使能
6. #define TCFG_USER_EDR_ENABLE                 0   //EDR 功能使能
```

2.21.3 模块开发

参考《HiLink 协议开发说明》

Chapter 3　SIG Mesh 使用说明

3.1 概述

遵守蓝牙 SIG Mesh 协议，基于蓝牙 5 ble 实现网内节点间通讯，具体功能如下：

- 全节点类型支持(Relay/Proxy/Friend/Low Power)；
- 支持以 PB-GATT 方式入网(手机 APP 配网，支持"nRF Mesh"安卓和苹果最新版本)；
- 支持以 PB-ADV 方式入网（“天猫精灵”配网）；
- 支持设备上电自配网(不用网关就可实现设备在同一网内，支持加密 key 自定义)；
- 节点 relay/beacon 功能可修改；
- 节点地址可自定义；
- 节点信息断电保存；
- 节点发布(Publish)和订阅(Subscribe)地址可修改；
- 支持节点 Reset 为未配网设备；
- 支持蓝牙 SIG 既有 Models 和用户自定义 Vendor Models。

支持的板级： bd29、br25、br30、bd19、br34、br23

支持的芯片： AC631N、AC636N、AC637N、AC632N、AC638N

3.2 工程配置

代码工程：apps\mesh\board\bd29\AC631N_mesh.cbp

```
▾ mesh/
  ▾ api/
    feature_correct.h
    mesh_config_common.c
    model_api.c
    model_api.h
  ▾ board/
    ▾ bd29/
      board_ac630x_demo.c
      board_ac630x_demo_cfg.h
      board_ac6311_demo.c
      board_ac6311_demo_cfg.h
      board_ac6313_demo.c
      board_ac6313_demo_cfg.h
      board_ac6318_demo.c
      board_ac6318_demo_cfg.h
      board_ac6319_demo.c
      board_ac6319_demo_cfg.h
      board_config.h
  ▾ examples/
    generic_onoff_client.c
    generic_onoff_server.c
    vendor_client.c
    vendor_server.c
    AliGenie_socket.c
```

在 api/model_api.h 下，通过配置 CONFIG_MESH_MODEL 选择相应例子，SDK 提供了 5 个应用实例。默认选择 SIG_MESH_GENERIC_ONOFF_CLIENT，即位于 examples/generic_onoff_client.c 下的例子。

```
4.  //< Detail in "MshMDLv1.0.1"
5.  #define SIG_MESH_GENERIC_ONOFF_CLIENT      0 // examples/generic_onoff_client.c
6.  #define SIG_MESH_GENERIC_ONOFF_SERVER      1 // examples/generic_onoff_server.c
7.  #define SIG_MESH_VENDOR_CLIENT             2 // examples/vendor_client.c
8.  #define SIG_MESH_VENDOR_SERVER             3 // examples/vendor_server.c
9.  #define SIG_MESH_ALIGENIE_SOCKET           4 // examples/AliGenie_socket.c
10. // more...
11.
12. //< Config which example will use in <examples>
13. #define CONFIG_MESH_MODEL                  SIG_MESH_GENERIC_ONOFF_CLIENT
```

3.2.2 Mesh 配置

在 api/mesh_config_common.c 下，可以自由配置网络和节点特性，例如 LPN/Friend 节点特性、Proxy 下配网前后广播 interval、节点信息传递时广播 interval 和 duration 等。

如下举例了节点信息传递时广播 interval 和 duration 的配置，和 PB-GATT 下配网前后广播 interval 的配置。

```
/**
* @brief Config adv bearer hardware param when node send messages
*/
/*-------------------------------------------------------------*/
const u16 config_bt_mesh_node_msg_adv_interval = ADV_SCAN_UNIT(10); // unit: ms
const u16 config_bt_mesh_node_msg_adv_duration = 100; // unit: ms

/**
* @brief Config proxy connectable adv hardware param
*/
/*-------------------------------------------------------------*/
const u16 config_bt_mesh_proxy_unprovision_adv_interval = ADV_SCAN_UNIT(30); // unit: ms
const u16 config_bt_mesh_proxy_pre_node_adv_interval = ADV_SCAN_UNIT(10); // unit: ms
_WEAK_
const u16 config_bt_mesh_proxy_node_adv_interval = ADV_SCAN_UNIT(300); // unit: ms
```

注意：在常量前加上“_WEAK_”的声明，代表这个常量可以在其它文件重定义。如果想某一配置私有化到某一实例，应该在该文件下在该配置前加上“_WEAK_”声明，并在所在实例文件里重定义该常量配置。

3.2.3 board 配置

在 board/xxxx/board_config.h 下，可以根据不同封装选择不同 board，以 AC63X 为例，默认选择 CONFIG_BOARD_AC630X_DEMO 作为目标板。

```
4.  /*
5.  * 板级配置选择
6.  */
7.  #define CONFIG_BOARD_AC630X_DEMO
8.  // #define CONFIG_BOARD_AC6311_DEMO
9.  // #define CONFIG_BOARD_AC6313_DEMO
10. // #define CONFIG_BOARD_AC6318_DEMO
11. // #define CONFIG_BOARD_AC6319_DEMO
12.
13. #include "board_ac630x_demo_cfg.h"
14. #include "board_ac6311_demo_cfg.h"
15. #include "board_ac6313_demo_cfg.h"
16. #include "board_ac6318_demo_cfg.h"
17. #include "board_ac6319_demo_cfg.h"
```

3.3 应用实例

3.3.1 SIG Generic OnOff Client

1、简介

该实例通过手机“nRF Mesh”进行配网

```
-> 设备名称：OnOff_cli
-> Node Features：Proxy + Relay
-> Authentication 方式：NO OOB
-> Elements 个数：1
-> Model：Configuration Server + Generic On Off Client
```

2、实际操作

1) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA3 作为 AD 按键

设备名称为“OnOff_cli”，MAC 地址为“11:22:33:44:55:66”

```
-> api/model_api.h
#define CONFIG_MESH_MODEL              SIG_MESH_GENERIC_ONOFF_CLIENT
-> board/xxxx/board_xxxx_demo_cfg.h
#define TCFG_UART0_TX_PORT             IO_PORT_DP
#define TCFG_UART0_BAUDRATE            1000000
#define TCFG_ADKEY_ENABLE              ENABLE_THIS_MOUDLE //是否使能 AD 按键
#define TCFG_ADKEY_PORT                IO_PORTA_03 //注意选择的 IO 口是否支持 AD 功能
#define TCFG_ADKEY_AD_CHANNEL          AD_CH_PA3
-> examples/generic_onoff_client.c
#define BLE_DEV_NAME                   'O', 'n', 'O', 'f', 'f', '_', 'c', 'l', 'i'
#define CUR_DEVICE_MAC_ADDR            0x112233445566
```

对于 MAC 地址，如果想不同设备在第一次上电时使用随机值，可以按照以下操作，将 NULL 传入 bt_mac_addr_set 函数

如果想用配置工具配置 MAC 地址，应不调用 bt_mac_addr_set 函数

```
-> examples/generic_onoff_client.c
void bt_ble_init(void)
{
    u8 bt_addr[6] = {MAC_TO_LITTLE_ENDIAN(CUR_DEVICE_MAC_ADDR)};

    bt_mac_addr_set(NULL);

    mesh_setup(mesh_init);
}
```

2) .编译工程并下载到目标板，接好串口，接好 AD 按键，上电或者复位设备

3) .使用手机 APP“nRF Mesh”进行配网，详细操作请 ->点击这里<-

(该动图位于该文档同级目录，如点击无效请手动打开“Generic_On_Off_Client.gif”)

配网完成后节点结构如下：

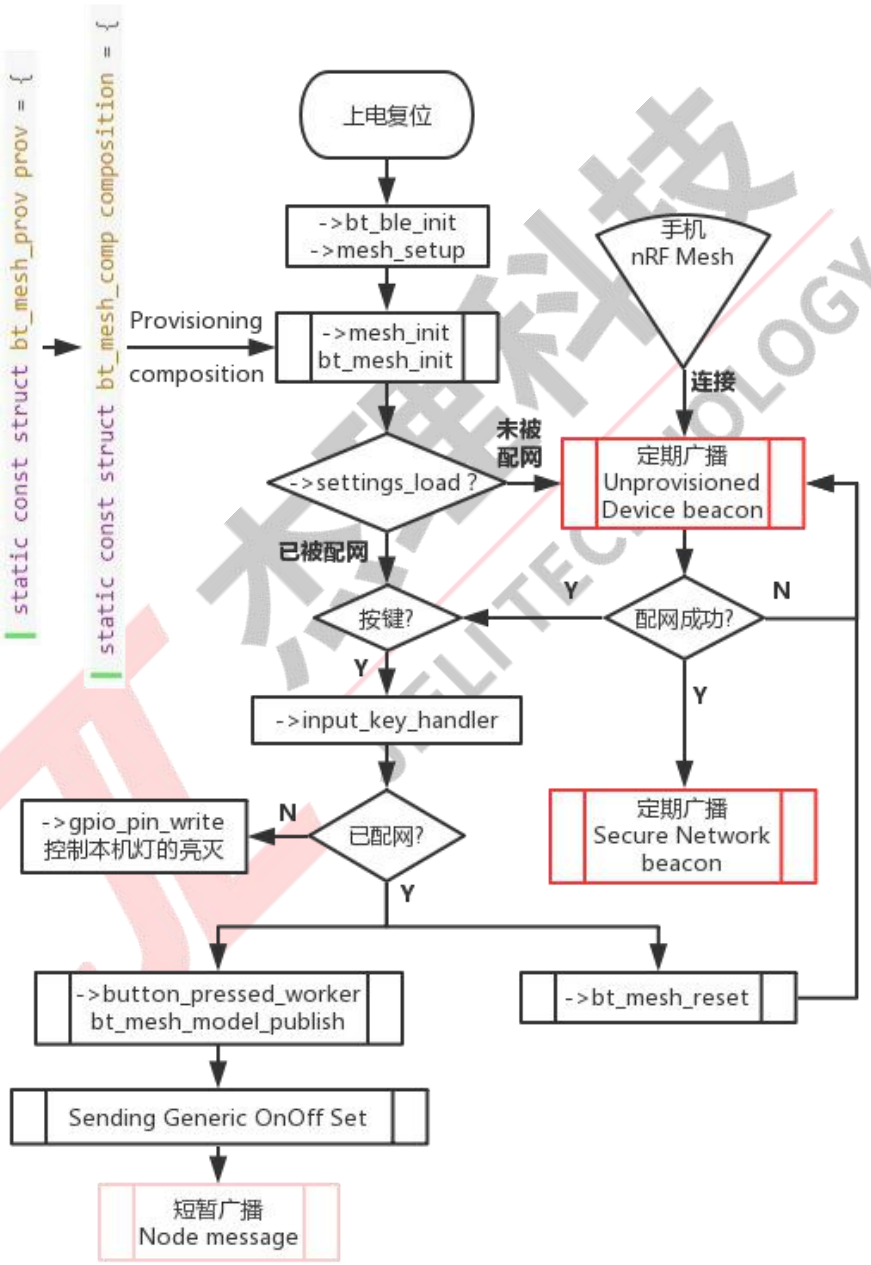
```
▾ Elements
  ▾ Element
```

```
Configuration Server  #SIG Model ID: 0x0000
Generic On Off Client  #SIG Model ID: 0x1001
```

4) .此时按下按键，就能将开关信息 Publish 到 Group 地址 0xC000 了，如果结合下一小节 SIG Generic OnOff Server，就能控制这个 server 设备 led 灯的亮和灭了

3、代码解读

1) .Client 运作流程图



3.3.2 SIG Generic OnOff Server

1、简介

该实例通过手机“nRF Mesh”进行配网

```
-> 设备名称：OnOff_srv
-> Node Features：Proxy + Relay
-> Authentication 方式：NO OOB
-> Elements 个数：1
-> Model：Configuration Server + Generic On Off Server
```

2、实际操作

1) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA1 控制 LED 灯

设备名称为“OnOff_srv”，MAC 地址为“22:22:33:44:55:66”

```
-> api/model_api.h
#define CONFIG_MESH_MODEL              SIG_MESH_GENERIC_ONOFF_SERVER
-> board/xxxx/board_xxxx_demo_cfg.h
#define TCFG_UART0_TX_PORT             IO_PORT_DP
#define TCFG_UART0_BAUDRATE            1000000
-> examples/generic_onoff_server.c
#define BLE_DEV_NAME                   'O', 'n', 'O', 'f', 'f', '_', 's', 'r', 'v'
#define CUR_DEVICE_MAC_ADDR            0x222233445566
const u8 led_use_port[] = {
  IO_PORTA_01,
};
```

对于 MAC 地址，如果想不同设备在第一次上电时使用随机值，可以按照以下操作，将 NULL 传入 bt_mac_addr_set 函数

如果想用配置工具配置 MAC 地址，应不调用 bt_mac_addr_set 函数

```
-> examples/generic_onoff_server.c
void bt_ble_init(void)
{
    u8 bt_addr[6] = {MAC_TO_LITTLE_ENDIAN(CUR_DEVICE_MAC_ADDR)};

    bt_mac_addr_set(NULL);

    mesh_setup(mesh_init);
}
```

2) .编译工程并下载到目标板，接好串口，接好演示用 LED 灯，上电或者复位设备

3) .使用手机 APP“nRF Mesh”进行配网，详细操作请 ->点击这里<-

(该动图位于该文档同级目录，如点击无效请手动打开“Generic_On_Off_Server.gif”)

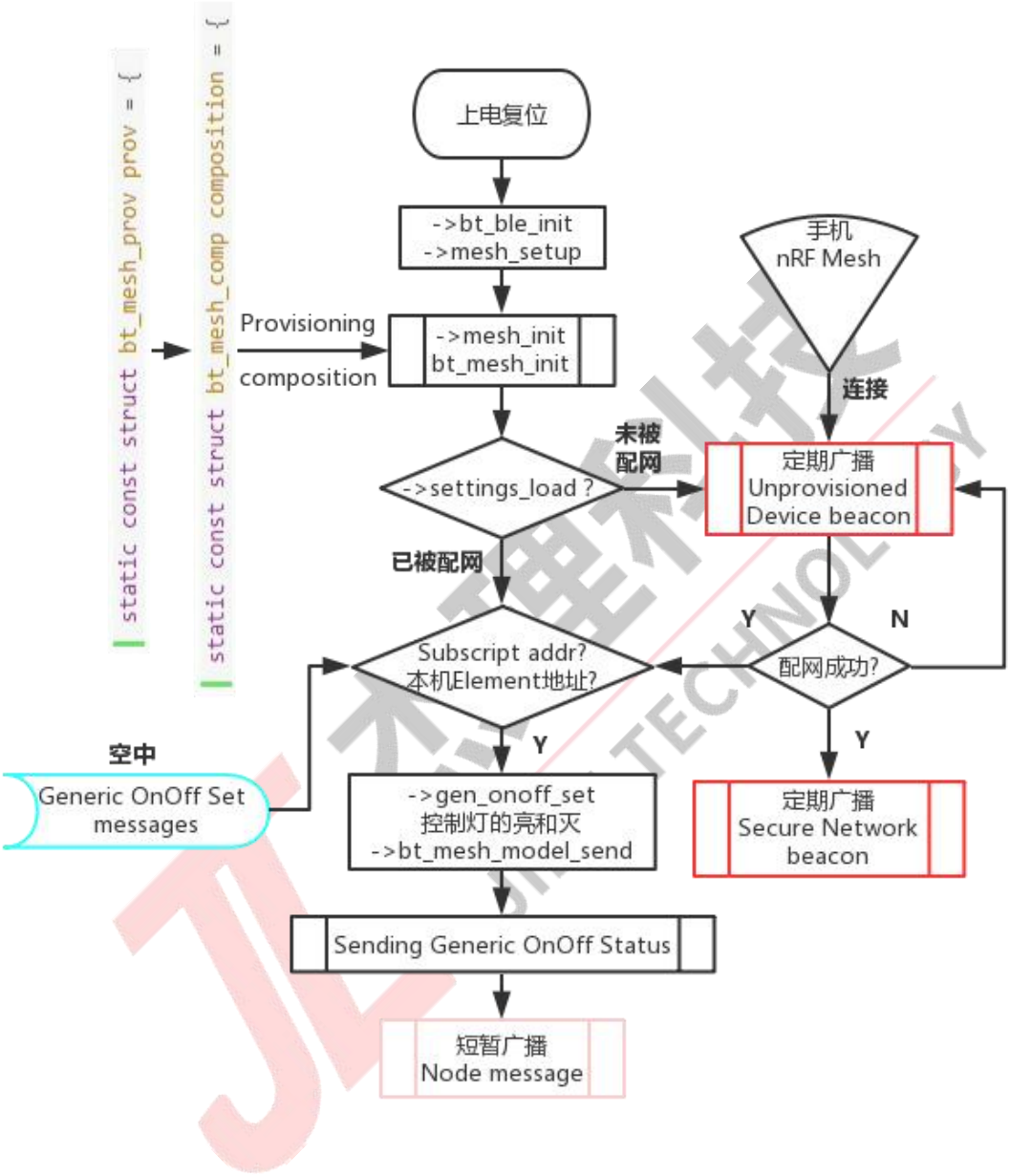
配网完成后节点结构如下：

```
▾ Elements
  ▾ Element
    Configuration Server  #SIG Model ID: 0x0000
    Generic On Off Server  #SIG Model ID: 0x1000
```

4) .结合上一小节 SIG Generic OnOff Client，此时如果 Client 设备按下按键，那么本机的 LED 灯就会 亮或者灭了

3、代码解读

1) .Server 运作流程图



3.3.3 SIG AliGenie Socket

1、简介

该实例按照阿里巴巴“*IoT 开放平台*”关于“*天猫精灵蓝牙 mesh 软件基础规范*”，根据“*硬件品类规范*”描述自己为一个“*插座*”，通过“天猫精灵”语音输入进行发现连接(配网)和控制设备。

2、实际操作

1) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA1 控制 LED 灯(模拟插座的开和关的操作)

设备名称为“AG-Socket”

三元组（MAC 地址、ProductID、Secret）在天猫精灵开发者网站申请

```
-> api/model_api.h
#define CONFIG_MESH_MODEL              SIG_MESH_ALIGENIE_SOCKET
-> board/xxxx/board_xxxx_demo_cfg.h
#define TCFG_UART0_TX_PORT             IO_PORT_DP
#define TCFG_UART0_BAUDRATE            1000000
-> examples/AliGenie_socket.c
#define BLE_DEV_NAME                   'A', 'G', '-', 'S', 'o', 'c', 'k', 'e', 't'
//< 三元组(本例以个人名义申请的插座类三元组)
#define CUR_DEVICE_MAC_ADDR            0x28fa7a42bf0d
#define PRODUCT_ID                     12623
#define DEVICE_SECRET                  "753053e923f30c9f0bc4405cf13ebda6"
const u8 led_use_port[] = {
    IO_PORTA_01,
};
```

对于 MAC 地址，本例中一定要按照三元组里面的 MAC 地址传入到 bt_mac_addr_set 函数里

```
-> examples/AliGenie_socket.c
void bt_ble_init(void)
{
    u8 bt_addr[6] = {MAC_TO_LITTLE_ENDIAN(CUR_DEVICE_MAC_ADDR)};

    bt_mac_addr_set(bt_addr);

    mesh_setup(mesh_init);
}
```

2) .编译工程并下载到目标板，接好串口，接好演示用 LED 灯，上电或者复位设备

3) .天猫精灵连接到互联网上

①. 上电“天猫精灵”，长按设备上的语音按键，让设备进入待连接状态

②. 手机应用商店下载“天猫精灵”APP，APP 上个人中心登陆

③. 打开手机“WLAN”，将“天猫精灵”通过手机热点连接到互联网上

详细操作请 ->点击这里<-

(该动图位于该文档同级目录，如点击无效请手动打开“AliGenie_connect.gif”)

4) .通过天猫精灵进行配网和控制

①. 配网对话

用户：“天猫精灵，搜索设备”

天猫精灵：“发现一个智能插座，是否连接”

用户：“连接”

天猫精灵：“连接成功。。。 。。。”

②. 语音控制插座命令(可通过“*IoT 开放平台*”添加自定义语音命令)

命令：“天猫精灵，打开插座” 效果：开发板上 LED 灯打开

命令：“天猫精灵，关闭插座” 效果：开发板上 LED 灯关闭

3、代码解读

1) .配网

关键在于如何设置在天猫精灵开发者网站申请下来的三元组

①. 天猫精灵开发者网站申请三元组，并填到下面文件相应宏定义处

例如申请到的三元组如下：

Product ID(十进制)	Device Secret	Mac 地址
12623	753053e923f30c9f0bc4405cf13ebda6	28fa7a42bf0d

则按下面规则填写，MAC 前要加上 0x，Secret 要用双引号包住

```
-> examples/AliGenie_socket.c
//< 三元组(本例以个人名义申请的插座类三元组)
#define CUR_DEVICE_MAC_ADDR          0x28fa7a42bf0d
#define PRODUCT_ID                   12623
#define DEVICE_SECRET                "753053e923f30c9f0bc4405cf13ebda6"
```

②. 建立 Element 和 Model

按照*插座软件规范*，要建立一个 element，两个 model

Element	Model	属性名称
Primary	Generic On/Off Server 0x1000	开关
Primary	Vendor Model 0x01A80000	故障上报/ 定时控制开关

相应代码操作如下：

- 结构体 elements 注册了一个 primary element = SIG root_models + Vendor_server_models

- 结构体 root_models = Cfg_Server + Generic_OnOff_Server
- 结构体 vendor_server_models = Vendor_Client_Model + Vendor_Server_Model

```
//< Basic_Cfg_Server + Generic_OnOff_Server
static struct bt_mesh_model root_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_SRV, gen_onoff_srv_op, &gen_onoff_pub_srv, &onof
f_state[0]),
};

//< Vendor_Client + Vendor_Server
static struct bt_mesh_model vendor_server_models[] = {
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_CLI, NULL, NULL, NULL),
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_SRV, vendor_srv_op, NULL,
&onoff_state[0]),
};

//< Only primary element
static struct bt_mesh_elem elements[] = {
    BT_MESH_ELEM(0, root_models, vendor_server_models),  // primary element
    // second element
    // ... ...
};
```

2) .用户数据处理

①. SIG Generic OnOff Server 回调

结构体 root_models 里的 Generic_OnOff_Server 注册了回调 gen_onoff_srv_op 来对用户数据进行处理

当收到 BT_MESH_MODEL_OP_GEN_ONOFF_GET 等注册消息时，就会调用 gen_onoff_get 等对应的回调函数进行用户数据处理

```
static const struct bt_mesh_model_op gen_onoff_srv_op[] = {
    { BT_MESH_MODEL_OP_GEN_ONOFF_GET, 0, gen_onoff_get },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET, 2, gen_onoff_set },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK, 2, gen_onoff_set_unack },
    BT_MESH_MODEL_OP_END,
};
```

②. Vendor Model 回调

结构体 vendor_srv_op 里的 Vendor_Server_Model 注册了回调 vendor_srv_op 来对用户数据进行处理

当收到 VENDOR_MSG_ATTR_GET 等注册消息时，就会调用 vendor_attr_get 等对应的回 调函数进行用户数据处理

```
static const struct bt_mesh_model_op vendor_srv_op[] = {
    { VENDOR_MSG_ATTR_GET, ACCESS_OP_SIZE, vendor_attr_get },
    { VENDOR_MSG_ATTR_SET, ACCESS_OP_SIZE, vendor_attr_set },
    BT_MESH_MODEL_OP_END,
};
```

3.3.4 SIG Vendor Client

1、简介

该实例会自动进行配网

```
1. -> 设备名称：Vd_cli
2. -> Node Features:Proxy
3. -> Authentication 方式：NO OOB
4. -> Elements 个数：1
5. ->Root model：Configuration Server + Configuration Client
6. ->Vendor model: Vendor_Client_Model
```

2、实际操作

1) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA3 作为 AD 按键端口，将 PA3 与 ADKEY 相连

设备名称为“Vd_cli” , “Node”地址为“0x0001”,“Group”地址为“0xc000”

设备名称为“OnOff_cli”，MAC 地址为“11:22:33:44:55:66”

```
1. -> api/model_api.h
2. #define CONFIG_MESH_MODEL            SIG_MESH_VENDOR_CLIENT
3. > board/xxxx/board_xxxx_demo_cfg.h
4. #define TCFG_UART0_TX_PORT           IO_PORT_DP
5. #define TCFG_UART0_BAUDRATE          1000000
6. -> examples/generic_onoff_server.c
7. #define BLE_DEV_NAME                 'V', 'd', '_', 'c', 'l', 'i'
8. #define CUR_DEVICE_MAC_ADDR          0x222233445566
9. const u8 led_use_port[] = {
10.     IO_PORTA_01,
11. };
```

2) .编译工程并下载到目标板上，接好串口，接好 AD 按键，上电或者复位设备

3) .此时按下按键，就能把开关信息 Publish 到 Group 地址 0xC000 了，结合下一小节 SIG Vendor Client，就可以控制 server 设备上的 LED 灯的亮灭了

3、代码解读

1) .配网流程

1. 配置节点信息与元素组成

```
int bt_mesh_init(const struct bt_mesh_prov *prov,
                 const struct bt_mesh_comp *comp);
    parameters:
        prov  节点的配置信息
            本实例中的 prov： static const struct bt_mesh_prov prov = {
                .uuid = dev_uuid,
                .output_size = 0,
                .output_actions = 0,
                .output_number = 0,
                .complete = prov_complete,
                .reset = prov_reset,
                };
        comp  节点的元素组成
            本实例中的 comp： static const struct bt_mesh_comp composition = {
                .cid = BT_COMP_ID_LF,
                .elem = elements,
                .elem_count = ARRAY_SIZE(elements),
                };
    return 值:
        int 型，返回 0 代表成功，其他代表出错
```

2. 配置节点地址与网络通信密钥

```
int bt_mesh_provision(const u8_t net_key[16], u16_t net_idx,
                      u8_t flags, u32_t iv_index, u16_t addr,
                      const u8_t dev_key[16]);
    parameters:
        addr            节点地址
        net_key         网络密钥，用于保护网络层的通信
        net_idx         net_key 网络密钥的索引
        dev_key         设备密钥，用于保护节点和配置客户端之间的通信。
    return_value:
        int 型，返回 0 代表成功，其他代表出错
```

3. 为节点添加 app_key

添加的 AppKey 必须与 NetKey 成对使用，AppKey 用于对接收和发送的消息进行身份验证和加密

```
int bt_mesh_cfg_app_key_add(u16_t net_idx, u16_t addr, u16_t key_net_idx,
                            u16_t key_app_idx, const u8_t app_key[16],
                            u8_t *status);
    parameters:
        addr            节点地址
        app_key         应用密钥，用于保护上层传输层的通信
        key_app_idx     app_key 的索引
    return_value:
        int 型
```

4. 小结

本实例中配置的 net_key/dev_key/app_key 必须与 server 的相同否则无法正常进行通信

2） .model 配置

1. 将 model 绑定到该节点添加的 app_key

将应用密钥与元素的模型绑定在一起

```
int bt_mesh_cfg_mod_app_bind_vnd(u16_t net_idx, u16_t addr, u16_t elem_addr,
                                 u16_t mod_app_idx, u16_t mod_id, u16_t cid,
                                 u8_t *status);
    parameters:
        addr            节点地址
        elem_addr       配置为该模型的元素地址
        mod_id          模型的标识
                        本例中的 mod_id：BT_MESH_VENDOR_MODEL_ID_CLI
        key_app_idx     app_key 的索引
        cid             公司标识符
    return_value:
        int 型
```

2. 为 model 添加 publish 行为

配置模型的发布状态

```
int bt_mesh_cfg_mod_pub_set_vnd(u16_t net_idx, u16_t addr, u16_t elem_addr,
                                u16_t mod_id, u16_t cid,
```

```
                struct bt_mesh_cfg_mod_pub *pub, u8_t *status);
    parameters:
        addr      本节点的地址，即 node_addr
        elem_addr  配置为 pub 中的发送属性的元素地址
        pub       pub 结构体存放 publish 行为的相关属性
                本实例中的 pub 结构体：
                struct bt_mesh_cfg_mod_pub pub;
                pub.addr = dst_addr;          //publish 的目的地址
                pub.app_idx = app_idx;        //app_key 的索引
                pub.cred_flag = 0;            //friendship 的凭证标志
                pub.ttl = 7;                  //生命周期，决定能被 relay 的次数
                pub.period = 0;               //定期状态发布的周期
                pub.transmit = 0;             //每次 publish msg 的重传次数（高 3 位）+ 重传输之间 50ms 的步骤数（低 5 位）

        status      请求消息的状态
```

3) .节点行为

1. client 的 Publish 行为

当按键按下时进入 input_key_handle 函数，该函数会获取按键的键值和按键的状态，进行按键处理

```
void input_key_handler(u8 key_status, u8 key_number)
```

根据按键的不同状态 input_key_handle 函数会将相应的状态信息（点击为 1，长按为 0）通过结构体 struct _switch *sw 传入 client_publish 函数

```
static void client_publish(struct switch *sw)
```

client_publish 函数会对要发送的 msg 进行处理，接下来会详细介绍 client_publish 中主要函数的作用及 msg 的构成

首先 client_publish 函数根据 key_number 获取存在 mod_cli_sw 结构体里相应的 vendor_model，这个 vendor_model 的发送属性在上一节的 bt_mesh_cfg_mod_pub_set_vnd 函数里已经进行了配置之后使用 bt_mesh_model_msg_init 函数初始化一个结构体 msg 并存入 3 字节的操作码

```
void bt_mesh_model_msg_init(struct net_buf_simple *msg, u32_t opcode);
```

接下来会使用 buffer_add_u8_at_tail 函数添加一个状态值到 msg 尾部，用于 server 端控制 LED 状态

```
u8 *buffer_add_u8_at_tail(void *buf, u8 val);
```

然后使用 buffer_memset 函数将 msg 中剩下的空的部分用 0x02 填满，msg 剩下的空的部分可以也可以由用户自定义内容

```
1. void *buffer_memset(struct net_buf_simple *buf, u8 val, u32 len);
```

之后就用 bt_mesh_model_publish 把包含 msg 的信息发送出去

```
1. int bt_mesh_model_publish(struct bt_mesh_model *model);
```

2. client 的回调函数

结构体 vendor_client_models 里的 BT_MESH_VENDOR_MODEL_ID_CLI 注册了 vendor_cli_op 来进行数据处理

当收到对面的 ack msg 并匹配上 BT_MESH_VENDOR_MODEL_OP_STATUS 时就会调用 vendor_status 回调函数对数据进行处理

```
1. static const struct bt_mesh_model_op vendor_cli_op[] = {
2. {
3.   BT_MESH_VENDOR_MODEL_OP_STATUS, ACCESS_OP_SIZE, vendor_status },
4.   BT_MESH_MODEL_OP_END,
5. };
```

vendor_status 函数的功能：显示作为 ack msg 的发送方 server 端的地址及其所受到 client 控制的情况

```
1. static void vendor_status(struct bt_mesh_model *model,
2.                 struct bt_mesh_msg_ctx *ctx,
3.                 struct net_buf_simple *buf)
```

3.3.5 SIG Vendor Server

1、简介

该实例会自动进行配网

```
1. ->设备名称：Vd_srv
2. ->Node Features:Proxy
3. ->Authentication 方式：NO OOB
4. ->Elements 个数：1
5. ->Root model：Configuration Server + Configuration Client
```

```
6.  ->Vendor model: Vendor_Client_Model
```

2、实际操作

1) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA3 作为 AD 按键端口，将 PA1 与 LED 相连

设备名称为“Vd_srv” , “Node”地址为“0x0002”,“Group”地址为“0xc000”

```
1.  -> api/model_api.h
2.  #define CONFIG_MESH_MODEL            SIG_MESH_VENDOR_SERVER
3.  > board/xxxx/board_xxxx_demo_cfg.h
4.  #define TCFG_UART0_TX_PORT           IO_PORT_DP
5.  #define TCFG_UART0_BAUDRATE          1000000
6.  -> examples/generic_onoff_server.c
7.  #define BLE_DEV_NAME                 'V', 'd', '_', 's', r', 'v'
8.  #define CUR_DEVICE_MAC_ADDR          0x222233445566
9.  const u8 led_use_port[] = {
10.     IO_PORTA_01,
11. };
```

2) .编译工程并下载到目标板上，接好串口，接好 LED 灯，上电或者复位设备

3) .此时若 client 端按下按键，server 设备上的 LED 灯就会根据按键的点击还是长按而点亮或熄灭了

3、代码解读

1) .配网流程

1. 配置节点的信息与元素组成

```
1.  int bt_mesh_init(const struct bt_mesh_prov *prov,
2.                   const struct bt_mesh_comp *comp);
3.  parameters:
4.          prov 节点的配置信息
5.              本实例中的 prov： static const struct bt_mesh_prov prov = {
6.                      .uuid = dev_uuid,
7.                      .output_size = 0,
8.                      .output_actions = 0,
9.                      .output_number = 0,
10.                     .complete = prov_complete,
```

```
                .reset = prov_reset,
                };
    comp 节点的元素组成
        本实例中的 comp： static const struct bt_mesh_comp composition = {
                .cid = BT_COMP_ID_LF,
                .elem = elements,
                .elem_count = ARRAY_SIZE(elements),
                };
return 值:
    int 型，返回 0 代表成功，其他代表出错
```

2. 配置节点地址与网络通信密钥

```
int bt_mesh_provision(const u8_t net_key[16], u16_t net_idx,
            u8_t flags, u32_t iv_index, u16_t addr,
            const u8_t dev_key[16]);
parameters:
    addr        节点地址
    net_key     网络密钥，用于保护网络层的通信
    net_idx     net_key 的索引
    dev_key     设备密钥，用于保护节点和配置客户端之间的通信。
return_value:
    int 型，返回 0 代表成功，其他代表出错
```

3. 为节点添加 app_key

添加的 AppKey 必须与 NetKey 成对使用，AppKey 用于对接收和发送的消息进行身份验证和加密

```
int bt_mesh_cfg_app_key_add(u16_t net_idx, u16_t addr, u16_t key_net_idx,
            u16_t key_app_idx, const u8_t app_key[16],
            u8_t *status);
parameters:
    addr            节点地址
    key_app_idx     app_key 的索引
    app_key         应用密钥，用于保护上层传输层的通信
    status          请求消息的状态
```

```
9.  return_value:
10.     int 型
```

4. 小结

本实例中配置的 net_key/dev_key/app_key 必须与 server 的相同否则无法正常进行通信

2) .model 配置

1. 将 model 绑定到该节点添加的 app_key

将应用密钥与元素的模式绑定在一起

```
1.  int bt_mesh_cfg_mod_app_bind_vnd(u16_t net_idx, u16_t addr, u16_t elem_addr,
2.                  u16_t mod_app_idx, u16_t mod_id, u16_t cid,
3.                  u8_t *status);
4.  parameters:
5.      addr        节点地址
6.      elem_addr   配置为该模型的元素地址
7.      mod_id      模型的标识
8.                  本例中的 mod_id：BT_MESH_VENDOR_MODEL_ID_CLI
9.      key_app_idx app_key 的索引
10.     cid         公司标识符
11.     status      请求消息的状态
12. return_value:
13.     int 型
```

2. 为 model 添加 Subscription 地址

配置模型的订阅地址，用于接收 client 端 publish 的 msg

```
1.  int bt_mesh_cfg_mod_sub_add_vnd(u16_t net_idx, u16_t addr, u16_t elem_addr,
2.                  u16_t sub_addr, u16_t mod_id, u16_t cid,
3.                  u8_t *status);
4.  parameters:
5.      addr        本节点的地址
6.      elem_addr   元素地址，此处填入订阅控制信息对应元素的地址
7.      sub_addr    订阅的地址
8.      mod_id      模型的标识
9.      status      请求消息的状态
10. return_value:
```

```
11.        int 型
```

3) .节点行为

1. server 的回调函数

结构体 vendor_client_models 里的 BT_MESH_VENDOR_MODEL_ID_CLI 注册了 vendor_cli_op 来进行数据处理

当收到对面的 ack 并匹配上 BT_MESH_VENDOR_MODEL_OP_STATUS 时就会调用 vendor_set 回调函数对数据进行处理

```
1.  static void vendor_set(struct bt_mesh_model *model,
2.                        struct bt_mesh_msg_ctx *ctx,
3.                        struct net_buf_simple *buf)
```

vendor_set 函数中使用 buffer_pull_u8_from_head 函数提取 msg 中存放的 led 状态信息

```
1.  u8 buffer_pull_u8_from_head(void *buf);
```

之后使用 gpio_pin_write 函数进行点灯或者灭灯操作

```
1.  void gpio_pin_write(u8_t led_index, u8_t onoff)
2.  parameters:
3.          led_index    GPIO 口的索引
4.          onoff        LED 的亮灭状态
```

进行点灯操作后要组织信息作为 ack 反馈给 client 端，首先使用 bt_mesh_model_msg_init 函数初始化一个 net_buf_simple 类型对象 ack_msg 并在其中添加 3 字节的操作码

```
1.  void bt_mesh_model_msg_init(struct net_buf_simple *msg, u32_t opcode)
```

之后使用 buffer_add_u8_at_tail 函数在 ack_msg 函数的尾部添加 LED 的状态

```
1.  u8 *buffer_add_u8_at_tail(void *buf, u8 val);
```

然后使用 buffer_memset 函数将 ack_msg 填满

```
1.  void *buffer_memset(struct net_buf_simple *buf, u8 val, u32 len);
```

最后就使用 bt_mesh_model_send 函数将反馈信息发送给 client 端

```
1.  int bt_mesh_model_send(struct bt_mesh_model *model,
2.                         struct bt_mesh_msg_ctx *ctx,
3.                         struct net_buf_simple *msg,
4.                         const struct bt_mesh_send_cb *cb,
5.                         void *cb_data);
6.  parameters:
7.          model       发送信息所属的模型
8.          ctx         消息的环境信息，包括通信的密钥，生命周期，远端地址等
9.          msg         反馈信息 ack_msg
10.         cb          可选的消息发送的回调，此实例为空
11.         cb_data     要传递给回调的用户数据，此实例为空
```

3.3.6 SIG AliGenie Light

1、简介

该实例按照阿里巴巴“*IoT 开放平台*”关于“*天猫精灵蓝牙 mesh 软件基础规范*”，根据“*硬件品类规范*”描述自己为一个“*灯*”，通过“天猫精灵”语音输入进行发现连接(配网)和控制设备。

2、实际操作

2) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA1 控制 LED 灯(模拟插座的开和关的操作)

设备名称为“AG-Socket”

三元组（MAC 地址、ProductID、Secret）在天猫精灵开发者网站申请

```
-> api/model_api.h
#define CONFIG_MESH_MODEL               SIG_MESH_ALIGENIE_SOCKET
-> board/xxxx/board_xxxx_demo_cfg.h
#define TCFG_UART0_TX_PORT              IO_PORT_DP
#define TCFG_UART0_BAUDRATE             1000000
-> examples/AliGenie_socket.c
#define BLE_DEV_NAME                    'A', 'G', '-', 'L', 'i', 'g', 'h', 't'
//< 三元组(本例以个人名义申请的插座类三元组)
```

```
#define CUR_DEVICE_MAC_ADDR          0x18146c110001
#define PRODUCT_ID                   7218909
#define DEVICE_SECRET                "aab00b61998063e62f98ff04c9a787d4"
const u8 led_use_port[] = {
   IO_PORTA_01,
};
```

对于 MAC 地址，本例中一定要按照三元组里面的 MAC 地址传入到 bt_mac_addr_set 函数里

```
-> examples/AliGenie_socket.c
void bt_ble_init(void)
{
      u8 bt_addr[6] = {MAC_TO_LITTLE_ENDIAN(CUR_DEVICE_MAC_ADDR)};

      bt_mac_addr_set(bt_addr);

     mesh_setup(mesh_init);
}
```

4) .编译工程并下载到目标板，接好串口，接好演示用 LED 灯，上电或者复位设备

5) .天猫精灵连接到互联网上

①. 上电“天猫精灵”，长按设备上的语音按键，让设备进入待连接状态

②. 手机应用商店下载“天猫精灵”APP，APP 上个人中心登陆

③. 打开手机“WLAN”，将“天猫精灵”通过手机热点连接到互联网上

详细操作请 ->点击这里<-

(该动图位于该文档同级目录，如点击无效请手动打开“AliGenie_connect.gif”)

5) .通过天猫精灵进行配网和控制

①. 配网对话

用户：“天猫精灵，搜索设备”

天猫精灵：“发现一个智能插座，是否连接”

用户：“连接”

天猫精灵：“连接成功。。。 。。。”

②. 语音控制插座命令(可通过“*IoT 开放平台*”添加自定义语音命令)

命令：“天猫精灵，开灯” 效果：开发板上 LED 灯打开

命令：“天猫精灵，灯的亮度调到 50” 效果：开发板上 LED 灯亮度为 50%

命令：“天猫精灵，关灯” 效果：开发板上 LED 灯关闭

3、代码解读

2) .配网

关键在于如何设置在天猫精灵开发者网站申请下来的三元组

①. 天猫精灵开发者网站申请三元组，并填到下面文件相应宏定义处

例如申请到的三元组如下：

Product ID(十进制)	Device Secret	Mac 地址

7218909	aab00b61998063e62f98ff04c9a787d4	18146c110001

则按下面规则填写，MAC 前要加上 0x，Secret 要用双引号包住

```
-> examples/AliGenie_socket.c
//< 三元组(本例以个人名义申请的插座类三元组)
#define CUR_DEVICE_MAC_ADDR          0x18146c110001
#define PRODUCT_ID                   7218909
#define DEVICE_SECRET                "aab00b61998063e62f98ff04c9a787d4"
```

②. 建立 Element 和 Model

按照*灯具件规范*，要建立一个 element，三个 model

Element	Model	属性名称
Primary	Generic On/Off Server 0x1000	开关
Primary	Light_Lightness_Server 0x1300	亮度
Primary	Vendor Model 0x01A80000	故障上报/ 定时控制开关

相应代码操作如下：

- 结构体 elements 注册了一个 primary element = SIG root_models + Vendor_server_models
- 结构体 root_models = Cfg_Server + Generic_OnOff_Server + Light_Lightness_Server
- 结构体 vendor_server_models = Vendor_Client_Model + Vendor_Server_Model

```
//< Basic_Cfg_Server + Generic_OnOff_Server
static struct bt_mesh_model root_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_SRV, gen_onoff_srv_op, &gen_onoff_pub_srv, &onof
f_state[0]),
BT_MESH_MODEL(BT_MESH_MODEL_ID_LIGHT_LIGHTNESS_SRV, light_lightness_srv_op,
&gen_onoff_pub_srv, &light),
};


//< Vendor_Client + Vendor_Server
static struct bt_mesh_model vendor_server_models[] = {
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_CLI, NULL, NULL, NULL),
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_SRV, vendor_srv_op, NULL,
&onoff_state[0]),
};

//< Only primary element
static struct bt_mesh_elem elements[] = {
    BT_MESH_ELEM(0, root_models, vendor_server_models),  // primary element
    // second element
```

```
    // ... ...
};
```

3) .用户数据处理

①. SIG Generic OnOff Server 回调

结构体 root_models 里的 Generic_OnOff_Server 注册了回调 gen_onoff_srv_op 来对用户数据进行处理

当收到 BT_MESH_MODEL_OP_GEN_ONOFF_GET 等注册消息时，就会调用 gen_onoff_get 等对应的回调函数进行用户数据处理

```
static const struct bt_mesh_model_op gen_onoff_srv_op[] = {
    { BT_MESH_MODEL_OP_GEN_ONOFF_GET, 0, gen_onoff_get },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET, 2, gen_onoff_set },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK, 2, gen_onoff_set_unack },
    BT_MESH_MODEL_OP_END,
};
```

②. Vendor Model 回调

结构体 vendor_server_models 里的 Vendor_Server_Model 注册了回调 vendor_srv_op 来对用户数据进行处理

当收到 VENDOR_MSG_ATTR_GET 等注册消息时，就会调用 vendor_attr_get 等对应的回调函数进行用户数据处理

```
static const struct bt_mesh_model_op vendor_srv_op[] = {
    { VENDOR_MSG_ATTR_GET, ACCESS_OP_SIZE, vendor_attr_get },
    { VENDOR_MSG_ATTR_SET, ACCESS_OP_SIZE, vendor_attr_set },
    BT_MESH_MODEL_OP_END,
};
```

③. Light_Lightness_Server 回调

结构体 vendor_server_models 里的 Light_Lightness_Server 注册了回调 vendor_srv_op 来对用户数据进行处理

当收到 BT_MESH_MODEL_OP_LIGHT_LIGHTNESS_GET 等注册消息时，就会调用 vendor_attr_get 等对应的回调函数进行用户数据处理

```
static const struct bt_mesh_model_op light_lightness_srv_op[] = {
    { BT_MESH_MODEL_OP_LIGHT_LIGHTNESS_GET, 0 ,lightness_get },
    { BT_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET, 0, lightness_set },
    { BT_MESH_MODEL_OP_LIGHT_LIGHTNESS_SET_UNACK, 0, lightness_set_unack },
    BT_MESH_MODEL_OP_END,
};
```

3.3.7 SIG AliGenie Fan

1、简介

该实例按照阿里巴巴“*IoT 开放平台*”关于“*天猫精灵蓝牙 mesh 软件基础规范*”，根据“*硬件品类规范*”描述自己为一个“*风扇*”，通过“天猫精灵”语音输入进行发现连接(配网)和控制设备。

2、实际操作

3) .基本配置

使用 USB DP 作为串口 debug 脚，波特率为 1000000

使用 PA1 控制 LED 灯(模拟插座的开和关的操作)

设备名称为“AG-Socket”

三元组（MAC 地址、ProductID、Secret）在天猫精灵开发者网站申请

```
-> api/model_api.h
#define CONFIG_MESH_MODEL              SIG_MESH_ALIGENIE_SOCKET
-> board/xxxx/board_xxxx_demo_cfg.h
#define TCFG_UART0_TX_PORT             IO_PORT_DP
#define TCFG_UART0_BAUDRATE            1000000
-> examples/AliGenie_socket.c
#define BLE_DEV_NAME                   'A', 'G', '-', 'F', 'a', 'n
//< 三元组(本例以个人名义申请的插座类三元组)
#define CUR_DEVICE_MAC_ADDR            0x27fa7af002a0
#define PRODUCT_ID                     7809508
#define DEVICE_SECRET                  "d2729d5f3898079fa7b697c76a7bfe8e"
const u8 led_use_port[] = {
   IO_PORTA_01,
};
```

对于 MAC 地址，本例中一定要按照三元组里面的 MAC 地址传入到 bt_mac_addr_set 函数里

```
-> examples/AliGenie_socket.c
void bt_ble_init(void)
{
      u8 bt_addr[6] = {MAC_TO_LITTLE_ENDIAN(CUR_DEVICE_MAC_ADDR)};

      bt_mac_addr_set(bt_addr);

     mesh_setup(mesh_init);
}
```

6) .编译工程并下载到目标板，接好串口，接好演示用 LED 灯（用于指示风扇开关状态和风速），上电或者复位设备

7) .天猫精灵连接到互联网上

①. 上电“天猫精灵”，长按设备上的语音按键，让设备进入待连接状态

②. 手机应用商店下载“天猫精灵”APP，APP 上个人中心登陆

③. 打开手机“WLAN”，将“天猫精灵”通过手机热点连接到互联网上

详细操作请 ->点击这里<-

(该动图位于该文档同级目录，如点击无效请手动打开“AliGenie_connect.gif”)

6) .通过天猫精灵进行配网和控制

①. 配网对话

用户：“天猫精灵，搜索设备”

天猫精灵：“发现一个风扇，是否连接”

用户：“连接”

天猫精灵：“连接成功。。。 。。。”

②. 语音控制插座命令(可通过“*IoT 开放平台*”添加自定义语音命令)

命令：“天猫精灵，打开风扇” 效果：开发板上 LED 灯打开

命令：“天猫精灵，风扇调到 2 挡” 效果：开发板上 LED 灯亮度改变

命令：“天猫精灵，关闭风扇” 效果：开发板上 LED 灯关闭

3、代码解读

3) .配网

关键在于如何设置在天猫精灵开发者网站申请下来的三元组

①. 天猫精灵开发者网站申请三元组，并填到下面文件相应宏定义处

例如申请到的三元组如下：

Product ID（十进制）	Device Secret	Mac 地址
7809508	d2729d5f3898079fa7b697c76a7bfe8e	27fa7af002a0

则按下面规则填写，MAC 前要加上 0x，Secret 要用双引号包住

```
-> examples/AliGenie  socket.c
//< 三元组(本例以个人名义申请的插座类三元组)
#define CUR_DEVICE_MAC_ADDR          0x27fa7af002a0
#define PRODUCT_ID                   7809508
#define DEVICE_SECRET                "d2729d5f3898079fa7b697c76a7bfe8e"
```

②. 建立 Element 和 Model

按照*插座软件规范*，要建立一个 element，两个 model

Element	Model	属性名称
Primary	Generic On/Off Server 0x1000	开关
Primary	Vendor Model 0x01A80000	调整风速档位/ 定时控制开关

相应代码操作如下：

- 结构体 elements 注册了一个 primary element = SIG root_models + Vendor_server_models
- 结构体 root_models = Cfg_Server + Generic_OnOff_Server
- 结构体 vendor_server_models = Vendor_Client_Model + Vendor_Server_Model

```
//< Basic_Cfg_Server + Generic_OnOff_Server
static struct bt_mesh_model root_models[] = {
    BT_MESH_MODEL_CFG_SRV(&cfg_srv),
    BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_SRV, gen_onoff_srv_op, &gen_onoff_pub_srv, &onof
f_state[0]),
};

//< Vendor_Client + Vendor_Server
static struct bt_mesh_model vendor_server_models[] = {
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_CLI, NULL, NULL, NULL),
```

```
    BT_MESH_MODEL_VND(BT_COMP_ID_LF, BT_MESH_VENDOR_MODEL_ID_SRV, vendor_srv_op, NULL,
&onoff_state[0]),
};

//< Only primary element
static struct bt_mesh_elem elements[] = {
    BT_MESH_ELEM(0, root_models, vendor_server_models),  // primary element
    // second element
    // ... ...
};
```

4) .用户数据处理

①. SIG Generic OnOff Server 回调

结构体 root_models 里的 Generic_OnOff_Server 注册了回调 gen_onoff_srv_op 来对用户数据进行处理

当收到 BT_MESH_MODEL_OP_GEN_ONOFF_GET 等注册消息时，就会调用 gen_onoff_get 等对应的回调函数进行用户数据处理

```
static const struct bt_mesh_model_op gen_onoff_srv_op[] = {
    { BT_MESH_MODEL_OP_GEN_ONOFF_GET, 0, gen_onoff_get },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET, 2, gen_onoff_set },
    { BT_MESH_MODEL_OP_GEN_ONOFF_SET_UNACK, 2, gen_onoff_set_unack },
    BT_MESH_MODEL_OP_END,
};
```

②. Vendor Model 回调

结构体 vendor_srv_op 里的 Vendor_Server_Model 注册了回调 vendor_srv_op 来对用户数据进行处理

当收到 VENDOR_MSG_ATTR_GET 等注册消息时，就会调用 vendor_attr_get 等对应的回 调函数进行用户数据处理

```
static const struct bt_mesh_model_op vendor_srv_op[] = {
    { VENDOR_MSG_ATTR_GET, ACCESS_OP_SIZE, vendor_attr_get },
    { VENDOR_MSG_ATTR_SET, ACCESS_OP_SIZE, vendor_attr_set },
    BT_MESH_MODEL_OP_END,
};
```

Chapter 4 OTA 使用说明

4.1 概述

1.测试盒 OTA 升级介绍

AC630N 默认支持通过杰理蓝牙测试盒进行 BLE 或者 EDR 链路的 OTA 升级，方便客户在开发阶段对不方便有线升级的样机进行固件更新，或者在量产阶段进行批量升级。有关杰理蓝牙测试盒的使用及相关升级操作说明，详见文档“AC690x_1T2 测试盒使用说明.pdf”。

另外升级可使用直接跳转到 MASK 接口的升级方式，避免芯片掉电 IO 状态改变，适用于芯片电源由 IO 口 KEEP 住的方案，需要注意检查跳转前是否将使用 DMA 的硬件模块全部关闭。

在对应板级配置文件 xxx_global_build_cfg.h 中打开 CONFIG_UPDATE_JUMP_TO_MASK 配置：

```
2.  #define CONFIG_UPDATE_JUMP_TO_MASK                1     //配置升级到 loader 的方式 0 为直接 reset,1 为跳转(适用于芯片电源由 IO 口 KEEP 住的方案,需要注意检查跳转前是否将使用 DMA 的硬件模块全部关闭)
```

2. APP OTA 升级介绍

AC630N 可选支持 APP OTA 升级，SDK 提供通过 JL_RCSP 协议与 APP 交互完成 OTA 的 demo 流程。客户可以直接参考 JL_RCSP 协议相关文档和手机 APP OTA 外接库说明，将 APP OTA 功能集成到客户自家 APP 中。APP OTA 功能方便对已市场的产品进行远程固件推送升级，以此修复已知问题或支持新功能。

另外升级可使用直接跳转到 MASKROM 接口的升级方式，避免芯片掉电 IO 状态改变，适用于芯片电源由 IO 口 KEEP 住的方案，需要注意检查跳转前是否将使用 DMA 的硬件模块全部关闭。在对应板级配置文件 xxx_global_build_cfg.h 中打开 CONFIG_UPDATE_JUMP_TO_MASK 配置。

注意：目前 APP 升级支持 BLE 模式，EDR 暂不支持。

3.自定义单/双线串口升级介绍

AC630N 可选择支持自定义单/双线任意串口升级功能，对于单线串口升级，SDK 是通过发送数据时将单一 IO 口对其设置为发送状态，在数据发送完后在对其设置回接收状态这样的策略实现单线串口通信。当 SDK 接收到上位机发送符合杰理串口升级规范协议的数据后，就会进入升级流程。最新版本的烧写器可作为这类升级的上位机。目前支持的上位机是 1 拖 8 烧录工具。

4.2 OTA - APP 升级(BLE)

1、SDK 工程相关配置

1.1 在对应板级配置文件 xxx_global_build_cfg.h 中打开 CONFIG_APP_OTA_ENABLE 配置

```
1. /* Following Macros Affect Periods Of Both Code Compiling And Post-build */
2.
3. #define CONFIG_DOUBLE_BANK_ENABLE                0       //单双备份选择(若打开了改宏,FLASH 结构变为双备份结构，适用于接入第三方协议的OTA， PS: JL-OTA 同样支持双备份升级， 需要根据实际 FLASH 大小同时配置 CONFIG_FLASH_SIZE)
4. #define CONFIG_APP_OTA_ENABLE                    1       //是否支持 RCSP 升级(JL-OTA)
5.
6. #define CONFIG_UPDATE_JUMP_TO_MASK               0    //配置升级到 loader 的方式 0 为直接 reset,1 为跳转(适用于芯片电源由 IO 口 KEEP 住的方案,需要注意检查跳转前是否将使用 DMA 的硬件模块全部关闭)
```

1.2 生成的升级文件为 update.ufw，将其放在手机 APP 对应的文件目录中，连接蓝牙，选择文件后点击开始升级即可。

2、手机端工具

2.1 安卓端开发说明：详见 tools 目录下 Android_杰理 OTA 外接库开发说明

2.2 IOS 端开发说明： 详见 tools 目录下 IOS_杰理 OTA 外接库开发说明

4.2 自定义单/双线串口升级介绍

1、SDK 工程相关配置

1.1 在 app_config.h 中打开 USER_UART_UPDATE_ENABLE 配置

```
1.  #define USER_UART_UPDATE_ENABLE                    0       //是否支持自定义串口升级
```

1.2 在 app_config.h 中，对收/发 IO 进行配置，如果收/发都配置为同一 IO 口，则当前就是单线串口升级。

```
2.  #define UART_UPDATE_RX_PORT                    IO_PORTA_02       //设置 PA2 为接收口
3.  #define UART_UPDATE_TX_PORT                    IO_PORTA_03       //设置 PA3 为接收口
```

1.3 如果样机之间需要进行串口升级的话，则需要配置当前的角色，即配置主机/从机，假如配置为主机则需要发送符合 《杰理串口升级规范》协议的数据让从机进行升级流程。AC63N 默认是配置为从机。

```
#define UART_UPDATE_ROLE            UART_UPDATE_SLAVE       //主机对应的宏是 UART_UPDATE_MASTER
```

1.4 SDK 目前支持的上位机是 1 拖 8 烧录工具(V3.1.8 以上版本)。

Chapter 5 AUDIO 功能

5.1 概述

HID、SPP_AND_LE 和 MESH 新添加了 AUDIO 的实现示例代码，需要使用 AUDIO 功能要在板级配置使能 TCFG_AUDIO_ENABLE ，如下图：

```
1. //支持 Audio 功能，才能使能 DAC/ADC 模块
2. #ifdef CONFIG_LITE_AUDIO
3. #define TCFG_AUDIO_ENABLE                   ENABLE
4. #if TCFG_AUDIO_ENABLE
5. #undef TCFG_AUDIO_ADC_ENABLE
6. #undef TCFG_AUDIO_DAC_ENABLE
7. #define TCFG_AUDIO_ADC_ENABLE               ENABLE_THIS_MOUDLE
8. #define TCFG_AUDIO_DAC_ENABLE               ENABLE_THIS_MOUDLE
```

支持板级：br23、br25、br30、br34

支持芯片：AC635N、AC636N、AC673N、AC638N

5.2 Audio 的使用

1、DAC 硬件输出参数配置

在板级配置文件里面有如下配置，

```
1.  /*
2.  DAC 硬件上的连接方式,可选的配置:
3.      DAC_OUTPUT_MONO_L                左声道
4.      DAC_OUTPUT_MONO_R                右声道
5.      DAC_OUTPUT_LR                    立体声
6.      DAC_OUTPUT_MONO_LR_DIFF          单声道差分输出
7.  */
8.  #define TCFG_AUDIO_DAC_CONNECT_MODE          DAC_OUTPUT_MONO_LR_DIFF
```

需要根据具体的硬件接法，配置 TCFG_AUDIO_DAC_CONNECT_MODE 宏。

2、MIC 配置和使用

2.1 配置说明

在每个 board.c 文件里都有配置 mic 参数的结构体，如下图所示：

```
1.  struct adc_platform_data adc_data = {
2.      .mic_channel     = TCFG_AUDIO_ADC_MIC_CHA,                    //MIC 通道选择，对于
    693x，MIC 只有一个通道，固定选择右声道
3.  /*MIC LDO 电流档位设置:
4.      0:0.625ua    1:1.25ua    2:1.875ua    3:2.5ua*/
5.      .mic_ldo_isel    = TCFG_AUDIO_ADC_LDO_SEL,
6.  /*MIC 是否省隔直电容:
7.      0: 不省电容  1: 省电容 */
8.  #if ((TCFG_AUDIO_DAC_CONNECT_MODE == DAC_OUTPUT_FRONT_LR_REAR_LR) || (TCFG_AUDIO_DAC
    _CONNECT_MODE == DAC_OUTPUT_DUAL_LR_DIFF))
9.      .mic_capless     = 0,//四声道与双声道差分使用，不省电容接法
10. #else
11.     .mic_capless     = 0,
12. #endif
13. /*MIC 免电容方案需要设置，影响 MIC 的偏置电压
```

```
14.     21:1.18K    20:1.42K    19:1.55K    18:1.99K    17:2.2K     16:2.4K     15:2.6K
        14:2.91K    13:3.05K    12:3.5K     11:3.73K
15.     10:3.91K    9:4.41K     8:5.0K      7:5.6K      6:6K        5:6.5K      4:7K
        3:7.6K      2:8.0K      1:8.5K              */
16.     .mic_bias_res   = 16,
17. /*MIC LDO 电压档位设置,也会影响 MIC 的偏置电压
18.     0:2.3v  1:2.5v  2:2.7v  3:3.0v */
19.     .mic_ldo_vsel  = 2,
20. /*MIC 电容隔直模式使用内部 mic 偏置*/
21.     .mic_bias_inside = 1,
22. /*保持内部 mic 偏置输出*/
23.     .mic_bias_keep = 0,
24.
25.     // ladc 通道
26.     .ladc_num = ARRAY_SIZE(ladc_list),
27.     .ladc = ladc_list,
28. };
```

主要关注以下变量：

1）mic_capless：0：选用不省电容模式 1：选用省电容模式

2）mic_bias_res：选用省电容模式的时候才有效，mic 的上拉偏置电阻，选择范围为：

1:16K 2:7.5K 3:5.1K 4:6.8K 5:4.7K 6:3.5K 7:2.9K 8:3K 9:2.5K 10:2.1K 11:1.9K 12:2K 13:1.8K 14:1.6K 15:1.5K 16:1K 31:0.6K

3）mic_ldo_vsel：mic_ldo 的偏置电压，与偏置电阻共同决定 mic 的偏置，选择范围为：0:2.3v 1:2.5v 2:2.7v 3:3.0v

4）mic_bias_inside：mic 外部电容隔直，芯片内部提供偏置电压，当 mic_bias_inside=1，可以正常使用 mic_bias_res 和 mic_ldo_vsel

2.2 自动校准 MIC 偏置电压

使用省电容模式时，可在 app_config.h 配置 TCFG_MC_BIAS_AUTO_ADJUST，选择 MIC 的自动校准模式，自动选择对应的 MIC 偏置电阻和偏置电压。注意：不省电容无法校准。配置如下图：

```
1.  /*
2.   *省电容 mic 偏置电压自动调整(因为校准需要时间，所以有不同的方式)
3.   *1、烧完程序（完全更新，包括配置区）开机校准一次
```

```
*2、上电复位的时候都校准,即断电重新上电就会校准是否有偏差(默认)
*3、每次开机都校准，不管有没有断过电，即校准流程每次都跑
*/
#define MC_BIAS_ADJUST_DISABLE      0   //省电容 mic 偏置校准关闭
#define MC_BIAS_ADJUST_ONE          1   //省电容 mic 偏置只校准一次（跟 dac trim 一样）
#define  MC_BIAS_ADJUST_POWER_ON        2    // 省电容 mic 偏置每次上电复位都校准(Power_On_Reset)
#define MC_BIAS_ADJUST_ALWAYS       3   //省电容 mic 偏置每次开机都校准(包括上电复位和其他复位)
#define TCFG_MC_BIAS_AUTO_ADJUST    MC_BIAS_ADJUST_POWER_ON
#define TCFG_MC_CONVERGE_TRACE      0   //省电容 mic 收敛值跟踪
```

2.3 Mic 的使用示例

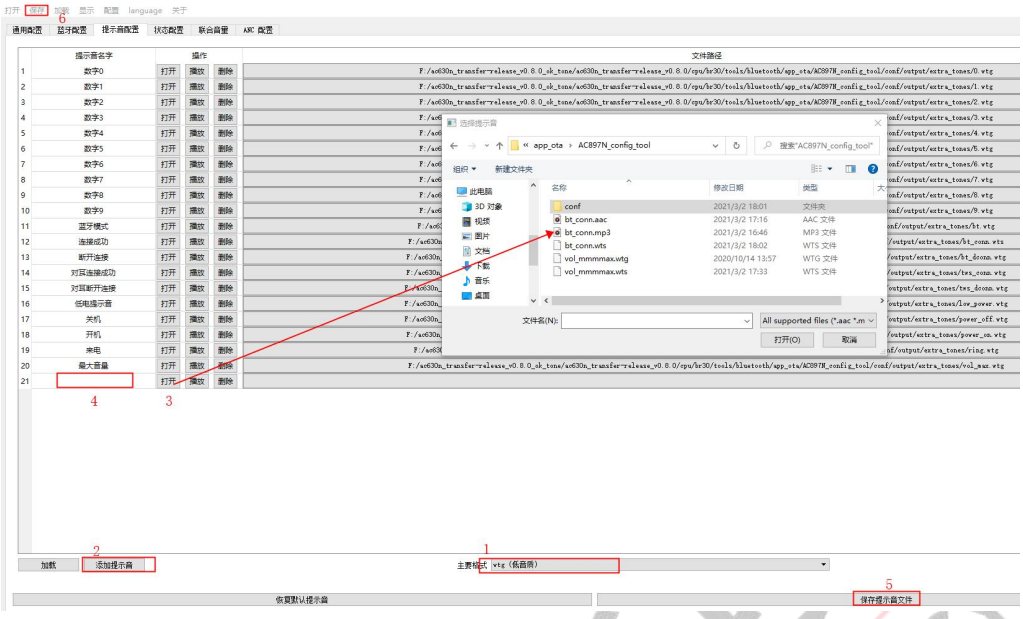
可调用 audio_adc_open_demo(void)函数输出 mic 的声音，示例如下图：

```
if (key_type == KEY_EVENT_LONG && key_value == TCFG_ADKEY_VALUE0) {
    printf(">>>key0:open mic\n");
    //br23/25 mic test
    extern int audio_adc_open_demo(void);
    audio_adc_open_demo();
    //br30 mic test
    /* extern void audio_adc_mic_demo(u8 mic_idx, u8 gain, u8 mic_2_dac); */
    /* audio_adc_mic_demo(1, 1, 1); */
}
```

3、提示音的使用

3.1 提示音文件配置

1.打开 SDK 对应的 cpu\brxx\tools\ACxxxN_config_tool，进入配置工具入口--->选择编译前配置工具--->提示音配置。



2.打开以上界面按步骤添加自己需要的*.mp3 格式的源文件，转换成需要的主要格式。要注意文件的路径，SDK 中默认的路径可能和本地保存的路径不同，要改成 SDK 当前的绝对路径。

3.在 ota 的目录 download.bat 下载项中添加 tone.cfg 配置选项。

4. 播放 sin\wtg 提示音，要在板级配置文件里面使能 TCFG_DEC_G729_ENABLE 和 TCFG_DEC_PCM_ENABLE 两个宏，如下图所示：

```
1.  #if TCFG_AUDIO_ENABLE
2.  #undef TCFG_AUDIO_ADC_ENABLE
3.  #undef TCFG_AUDIO_DAC_ENABLE
4.  #define TCFG_AUDIO_ADC_ENABLE              ENABLE_THIS_MOUDLE
5.  #define TCFG_AUDIO_DAC_ENABLE              ENABLE_THIS_MOUDLE
6.  #define TCFG_DEC_G729_ENABLE               ENABLE
7.  #define TCFG_DEC_PCM_ENABLE                ENABLE
8.  #define TCFG_ENC_OPUS_ENABLE               DISABLE
9.  #define TCFG_ENC_SPEEX_ENABLE              DISABLE
10. #define TCFG_LINEIN_LR_CH                  AUDIO_LIN0_LR
11. #else
12. #define TCFG_DEC_PCM_ENABLE                DISABLE
13. #endif/*TCFG_AUDIO_ENABLE*/
```

并在 tone_player.h 文件中定义文件名和路径：文件名和配置工具中第 4 步添加的提示音的名字要一致。

```
14.#define TONE_NUM_0              TONE_RES_ROOT_PATH"tone/0.*"
```

3.2 提示音使用示例

可以调用 tone_play()播放提示音，使用示例如下图：

```
if (key_type == KEY_EVENT_LONG && key_value == TCFG_ADKEY_VALUE2) {
    printf(">>>key1:tone_play_test\n");
    //br23/25 tone play test
    /* tone_play_by_path(TONE_NORMAL, 1); */
    /* tone_play_by_path(TONE_BT_CONN, 1); */
    //br30 tone play test
    tone_play(TONE_NUM_0, 1);
    /* tone_play(TONE_SIN_NORMAL, 1); */

}
```

4、opus\speex 编码的使用

4.1 配置说明

opus\speex 编码模块是对 mic 的数据进行编码，使用给功能，需要在板级配置文件里面使能 TCFG_ENC_OPUS_ENABLE 和 TCFG_ENC_SPEEX_ENABLE 这两个宏，配置如下图所示：

```
#if TCFG_AUDIO_ENABLE
#undef TCFG_AUDIO_ADC_ENABLE
#undef TCFG_AUDIO_DAC_ENABLE
#define TCFG_AUDIO_ADC_ENABLE              ENABLE_THIS_MOUDLE
#define TCFG_AUDIO_DAC_ENABLE              ENABLE_THIS_MOUDLE
#define TCFG_DEC_G729_ENABLE               ENABLE
#define TCFG_DEC_PCM_ENABLE                ENABLE
#define TCFG_ENC_OPUS_ENABLE               ENABLE
#define TCFG_ENC_SPEEX_ENABLE              ENABLE
#define TCFG_LINEIN_LR_CH                  AUDIO_LIN0_LR
#else
#define TCFG_DEC_PCM_ENABLE                DISABLE
#endif/*TCFG_AUDIO_ENABLE*/
```

4.2 opus\speex 编码示例

```
int audio_mic_enc_open(int (*mic_output)(void *priv, void *buf, int len), u32 code_type);
```

对 mic 的数据进行 opus\speex 编码使用 audio_mic_enc_open()函数，参数 mic_output 为编码后数据输出的函数，code_type 为要进行编码的类型，可选 AUDIO_CODING_OPUS 和 AUDIO_CODING_SPEEX，使用示例如下图：

```
1. /*encode test*/
2. extern int audio_mic_enc_open(int (*mic_output)(void *priv, void *buf, int len), u32
   code_type);
3. audio_mic_enc_open(mic_enc_output_data, AUDIO_CODING_OPUS);//opus encode test
4. /* audio_mic_enc_open(mic_enc_output_data, AUDIO_CODING_SPEEX);//speex encode test *
```

5、 通用编码接口的使用

5.1 配置说明

通用编码接口在 audio_codec_demo.c 文件里面，使用该接口需要在板级配置文件里面使能 ENC_DEMO_EN 这个宏，配置如下图所示：

```
#define  ENC_DEMO_EN                                    ENABLE;
```

5.2 要创建一个编码只需要调用如下的 audio_demo_enc_open 函数

```
int audio_demo_enc_open(int (*demo_output)(void *priv, void *buf, int len), u32 code_type, u8
ai_type)
```

第一个参数为外部注册的编码输出回调，注册此回调，在此回调中即可得到编码的后数据，第二个参数为编码类型，根据传入参数选择对应编码,可选的编码有 OPUS 编码，SPEEX 编码，ADPCM 编码，LC3 编码，SBC 编码和 MSBC 编码,创建对应编码器需分别在板级文件中使能 TCFG_ENC_OPUS_ENABLE,TCFG_ENC_SPEEX_ENABLE,TCFG_ENC_ADPCM_ENABLE TCFG_ENC_SBC_ENABLE,TCFG_ENC_SBC_ENABLE，MSBC 编码默认打开，没有宏控制

```
#define TCFG_ENC_OPUS_ENABLE                ENABLE
#define TCFG_ENC_SPEEX_ENABLE               ENABLE
#define TCFG_ENC_LC3_ENABLE                 ENABLE
#define TCFG_ENC_ADPCM_ENABLE               ENABLE
#define TCFG_ENC_SBC_ENABLE                 ENABLE
```

第三个参数是 speex 编码的参数，根据需要传 0 或者传 1 即可，不同的编码参数 修改均在 audio_demo_enc_open 函数内部修改 fmt 结构体的值即可，以 adpcm 参数为例，如下

```
    case AUDIO_CODING_WAV:
    fmt.sample_rate = 16000;
```

```
    fmt.bit_rate = 1024;  //blockSize,可配成 256/512/1024/2048
    fmt.channel = 2;
    fmt.coding_type = AUDIO_CODING_WAV;
    break;
```

根据实际编码参数修改 fmt 结构体 成员的值即可

audio_demo_enc_open 函数里默认打开了个定时器，定时器向解码器中写入需要编码的源数据，定时器执行如下函数：

```
static void demo_frame_test_time_fundc(void *parm)
```

在此函数中写入源数据即可，代码默认写入正弦波数据去编码

5.3 关闭编码只需要调用如下的 audio_demo_enc_close 函数

```
int  audio_demo_enc_close()
```

5.3 Audio_MIDI 的使用

5.3.1 文件下载配置

1、在 SDK\cpu\br23\tools 下添加对应 midi.bin 文件,在同级目录的 download.bat 下添加下载项：

```
isd_download.exe -tonorflash -dev br23 -boot 0x12000 -div8 -wait 300 -uboot uboot.boot -app app.bin
cfg_tool.bin -res midi.bin %1
```

在 download.bat 同级目录的 isd_config.ini 文件添加：

```
INTERNAL_DIR_ALIGN=0X2; //flash 内目录里的文件起始地址 4 对齐
```

2、在对应的 SDK\apps\spp_and_le\board\br23\board_ac635n_demo_cfg.h 文件中，使能 AUDIO 功能：

```
#define TCFG_AUDIO_ENABLE                       1//DISABLE
#define AUDIO_MIDI_CTRL_CONFIG                  1 //midi 电子琴接口使能 ,开这个宏要关掉低功耗使能
```

关闭低功耗模式：

```
#define TCFG_LOWPOWER_LOWPOWER_SEL              0//SLEEP_EN                     //SNIFF 状态
下芯片是否进入 powerdown#if TCFG_USER_BLE_ENABLE
```

3、在 SDK/cpu/br23/audio_dec/audio_dec_midi_ctrl.c 中做如下设置：具体函数内容以 SDK 为准。

```
  void midi_paly_test(u32 key)
{
    static u8 open_close = 0;
    static u8 change_prog = 0;
    static u8 note_on_off = 0;
    switch (key) {
    case KEY_IR_NUM_0:
        if (!open_close) {
            /* midi_ctrl_dec_open(16000);//启动 midi key */
            //midi_ctrl_dec_open(16000, "storage/sd0/C/MIDI.bin\0");//启动 midi key   SD 卡
            midi_ctrl_dec_open(16000,SDFILE_RES_ROOT_PATH"MIDI.bin\0");//启动 midi key  系统

        } else {
            midi_ctrl_dec_close();//关闭 midi key
        }
```

```
            open_close = !open_close;
            break;
        case KEY_IR_NUM_1:
            if (!change_prog) {
                midi_ctrl_set_porg(0, 0);//设置 0 号乐器，音轨 0
            } else {
                midi_ctrl_set_porg(22, 0);//设置 22 号乐器，音轨 0
            }
            change_prog = !change_prog;
            break;
        case KEY_IR_NUM_2:
            if (!note_on_off) {
                //模拟按键 57、58、59、60、61、62,以力度 127，通道 0，按下测试
                ...
            } else
                //模拟按键 57、58、59、60、61、62 松开测试
               ...
            }
            note_on_off = !note_on_off;
            break;
        default:
            break;
        }
}
```

4、在 app_spp_and_le.c 种调用 midi_play_test()函数播放对应文件：

```
static void app_key_event_handler(struct sys_event *event)
{
        ......
#if TCFG_AUDIO_ENABLE
if (event_type == KEY_EVENT_CLICK && key_value == TCFG_ADKEY_VALUE0) {
            /*midi test*/
        printf(">>>key0:open midi\n");
```

```
        midi_paly_test(KEY_IR_NUM_0);
        }
if (event_type == KEY_EVENT_CLICK && key_value == TCFG_ADKEY_VALUE1) {

            printf(">>>key0:set    midi\n");
            midi_paly_test(KEY_IR_NUM_1);

        }
        if (event_type == KEY_EVENT_CLICK && key_value == TCFG_ADKEY_VALUE2)
            printf(">>>key2:play    midi\n");
            midi_paly_test(KEY_IR_NUM_2);
        }
```

Chapter 6 lighting 握手充电功能

6.1 概述

支持可以使用 lighting 线给设备充电。当 lighting 插入后，先和 lighting 线握手使其输出强电，再开启内置充电功能，实现对设备的充电。Lighting 充电支持普通苹果线充电和苹果快充线充电，且支持正反插供电，具体供电多少 mA 和不同板级有关。

6.2 工程配置

1、SDK 工程相关配置

1.1 在板级配置文件 board_xxx_demo_cfg.h 中打开充电配置使能、是否支持 lighting 握手协议，另外可自定义协议通信的 IO 口配置，使能配置如下：

```
1.  //是否支持芯片内置充电
2.  #define TCFG_CHARGE_ENABLE              ENABLE_THIS_MOUDLE
3.  //是否支持开机充电
4.  #define TCFG_CHARGE_POWERON_ENABLE      DISABLE
5.  //是否支持拔出充电自动开机功能
6.  #define TCFG_CHARGE_OFF_POWERON_NE      DISABLE
7.  //是否支持 lighting 握手协议
8.  #define TCFG_HANDSHAKE_ENABLE           ENABLE
9.  #define TCFG_HANDSHAKE_IO_DATA1         IO_PORTB_02//握手 IO 靠近 lighting 座子中间的
10. #define TCFG_HANDSHAKE_IO_DATA2         IO_PORTB_07//握手 IO 在 lighting 座子边上的
```

1.2 主要代码说明：

1.2.1 lighting 充电初始化(handshake_app_start)、以及判断是否充电(get_charge_online_flag())设置不同的供电方式均在 board_xxx_demo_cfg.c 的 board_init 里面，对应的代码如下：

```
1.  void board_init()
2.  {
3.  #if TCFG_CHARGE_ENABLE && TCFG_HANDSHAKE_ENABLE
4.      if(get_charge_online_flag()){
5.          handshake_app_start(0, NULL);
6.      }
7.  #endif
8.
9.   if(get_charge_online_flag()) {
10.   power_set_mode(PWR_LDO15);
11.  } else {
12.   power_set_mode(TCFG_LOWPOWER_POWER_SEL);
13.  }
14. }
```

Chapter 7 SDK 通用功能说明

7.1 系统不可屏蔽中断

7.1.1 功能使用说明

不可屏蔽中断，是指在系统或者用户程序使用关中断接口，如 local_irq_disable() 关闭系统中断后，如果某个外设中断设置为不可屏蔽中断，则该中断依然可以响应，不受系统中断开关的影响，一般用于外设中断响应实时性要求非常高的场景。

目前支持的芯片有 AC631N、AC632N、AC636N。

发布的 sdk 默认关闭该功能，如果用户需要开启该功能，可以修改文件：config/lib_system_config.c 中的配置变量为 1

```
1. //================================================//
2. //        不可屏蔽中断使能配置(UNMASK_IRQ)        //
3. //================================================//
4. const int CONFIG_CPU_UNMASK_IRQ_ENABLE = 0;
```

在该变量使能后，用户可以通过不可屏蔽中断设置接口将某个外设设置为不可屏蔽中断，接口使用示例：

```
1. request_irq(7, 3, timer3_isr, 0); //注册中断号为 7 的外设中断向量函数，优先级为 3
2. irq_unmask_set(7, 0); //将中断号为 7 的外设中断设置为不可屏蔽中断，第二个参数默认为 0，指定
   cpu0。
```

注意事项：

由于不可屏蔽中断可以在系统或者用户在保护临界区的时候还能正常响应，因此不可屏蔽中断函数体需要满足如下要求：

1、中断函数代码需要放 RAM 中执行，不可定位到 Flash。

2、中断函数体应该尽可能的简单，只用于控制功能（如配置寄存器），切勿包含运算密度高的流程。

3、中断函数体不能调用操作系统接口，如有调用其他用户函数接口，该接口需要满足上述 1、2 点要求。

7.2 VBAT 供电和系统时钟频率关系限制

AC63 芯片系列主要提供的是宽压的 FALSH 使用和低功耗技术应用设计，不同的供电电压芯片只能跑不同的时钟频率；供电电压和系统时钟频率是有对应的关系限制，请注意使用，详见下表：

1、芯片 VBAT 和 VDDIO 没有接一起，VDDIO 跟 VBAT 大概有 0.3v 的电压差值

	系统时钟跑的频率范围
VBAT 供电范围（2.1v~3.0v）	24M~64M
VBAT 供电（3.0v 以上）	24~96M

2、芯片 VBAT 和 VDDIO 引脚短接一起

	系统时钟跑的频率范围
VBAT 供电范围（1.8v~2.7v）	24M~64M
VBAT 供电（2.7v 以上）	24~96M