珠海市杰理科技股份有限公司
ZhuHai JieLi Technology Co.,Ltd

AC63_bt_data_transfer_sdk_介绍

AC63_bt_data_transfer_sdk_介绍 用户手册

Rev 0.4.0 —— 2020 年 09 月 11 日

目录

修改日志

版本	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	

Chapter 1　SDK 快速使用说明

1.1 编写目的

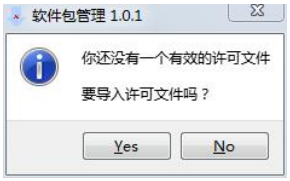
该文档主要描述 AC630N_SDK 开发包的使用方法及开发中注意的一些问题，为用户进行二次开发提供参考,其中包括：

安装包管理工具、配置工具使用、配置宏的选择以及充电仓使用注意事项

1.2 安装包管理工具

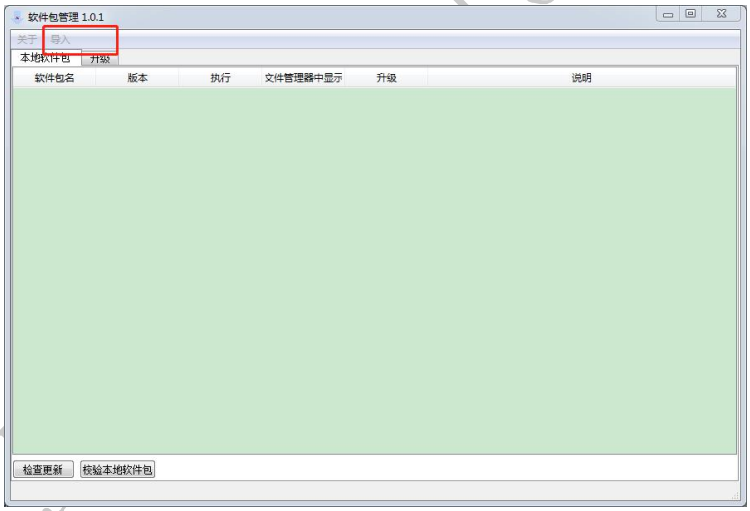
❖ 包管理工具安装

双击“jieli-sdk-tool-？.exe”安装，按照提示安装完成即可，安装完成后会提示导入 token 值，token 值需要向杰理科技公司申请，然后把申请到的 token 值导入到工具即可。如果运行的时候没有提示导入 token 值，可在“所有程序”-》“杰理工具包”-》“杰理包管理器”中导入申请到 token。会弹出如下图所示：



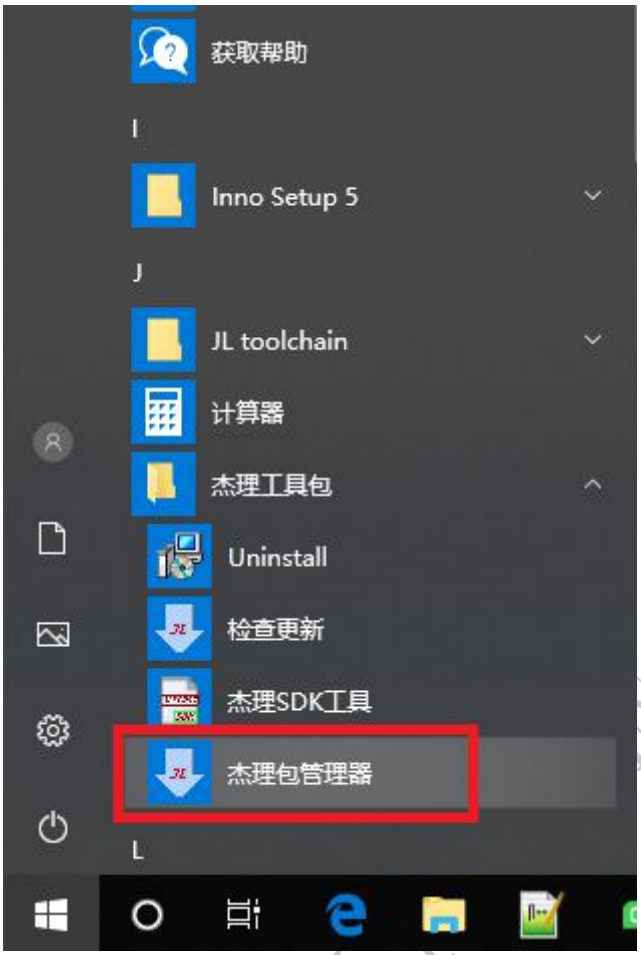
选择“Yes”后把 token 导入即可。

如果没有弹出该框，可以选择“导入”，把 token 导入即可



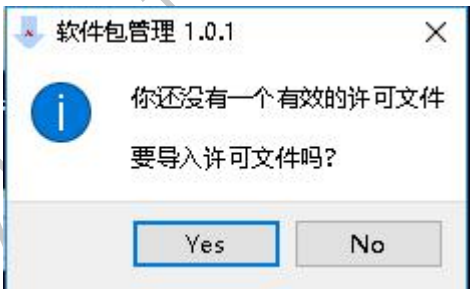
详细的安装及使用步骤如下：

1. 安装 杰理软件包 工具之后，开始菜单栏会出现对应的菜单

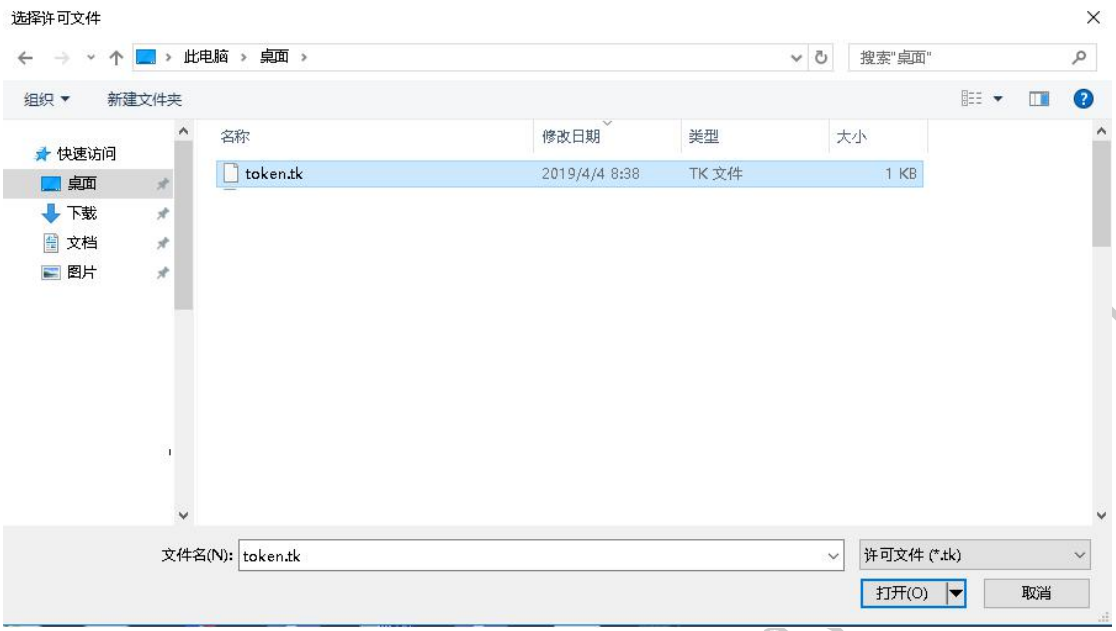


2. 导入 token 需要选择 “杰理包管理器”

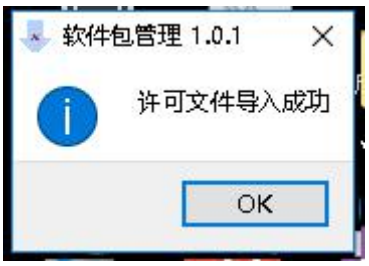
3. 如果是第一次打开，将会提示没有许可文件，如下图：



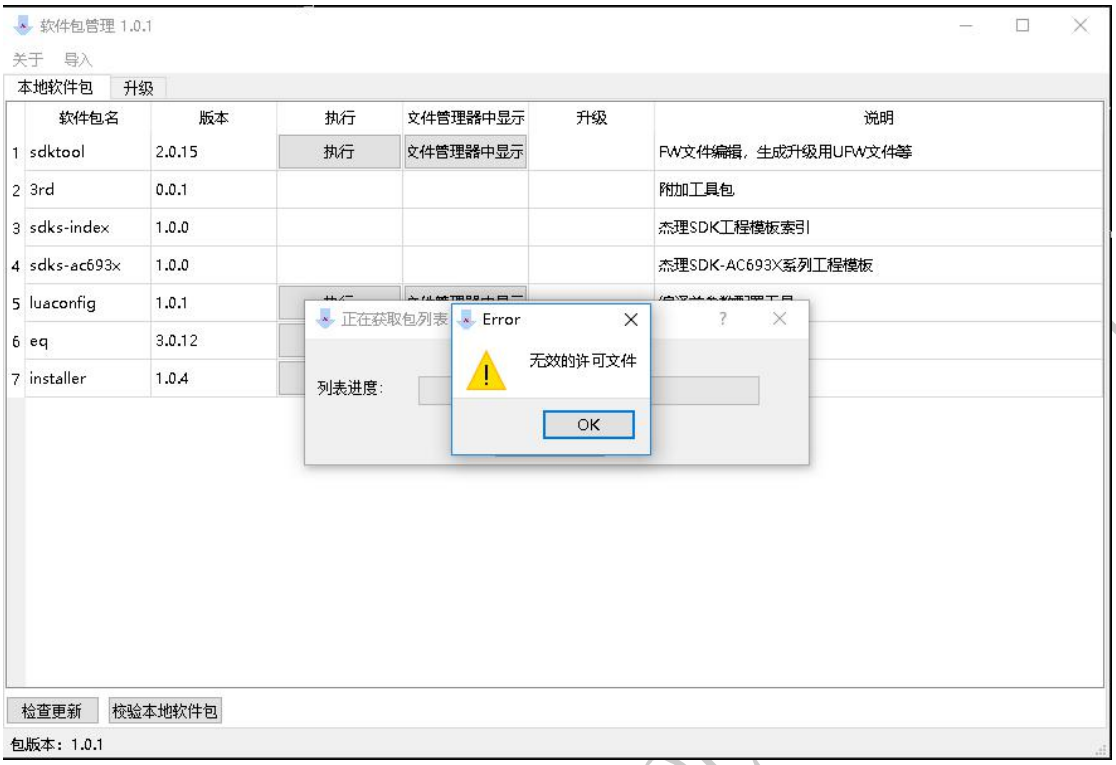
4 选择“Yes”之后，弹出对话框选择许可文件（.tk 结尾），如下图：



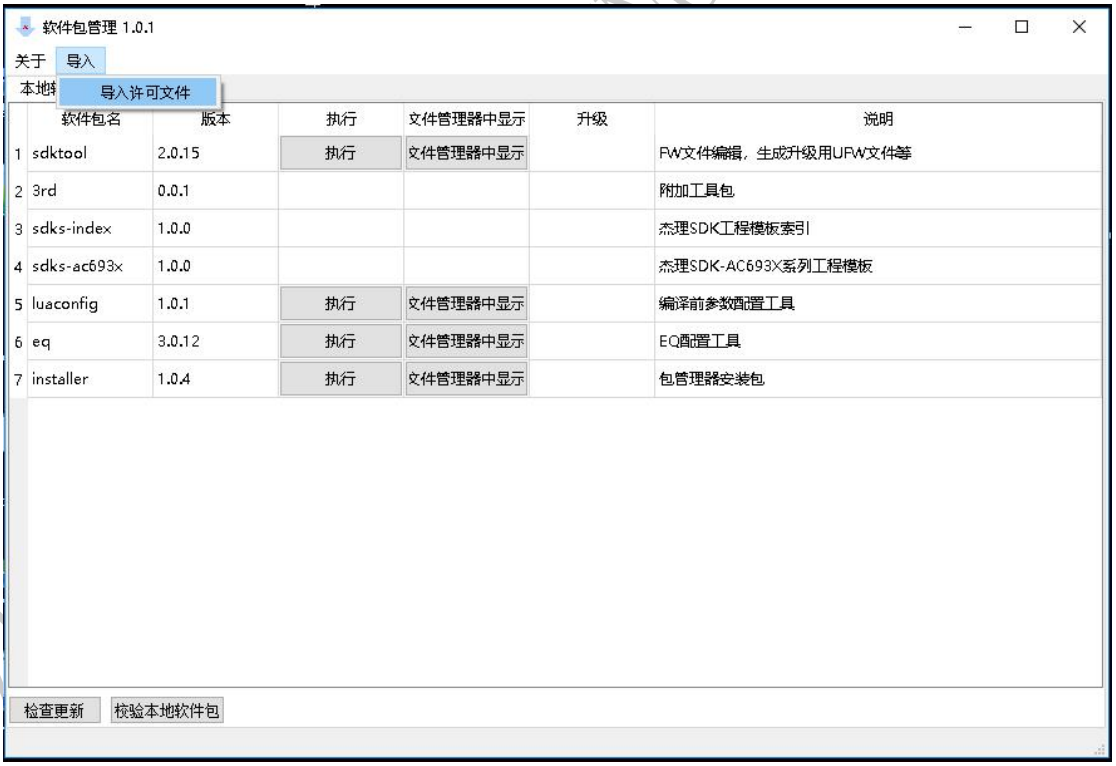
5.在选择正确的许可文件之后，将会有如下提示：



如果有如下提示，则说明许可文件无效。



许可文件无效之后，需要自己手动重新导入许可文件，如下图所示：



许可文件有过期时间，过期后，打开软件的时候有如下图所示提示，并将要求用户提供新的许可文件

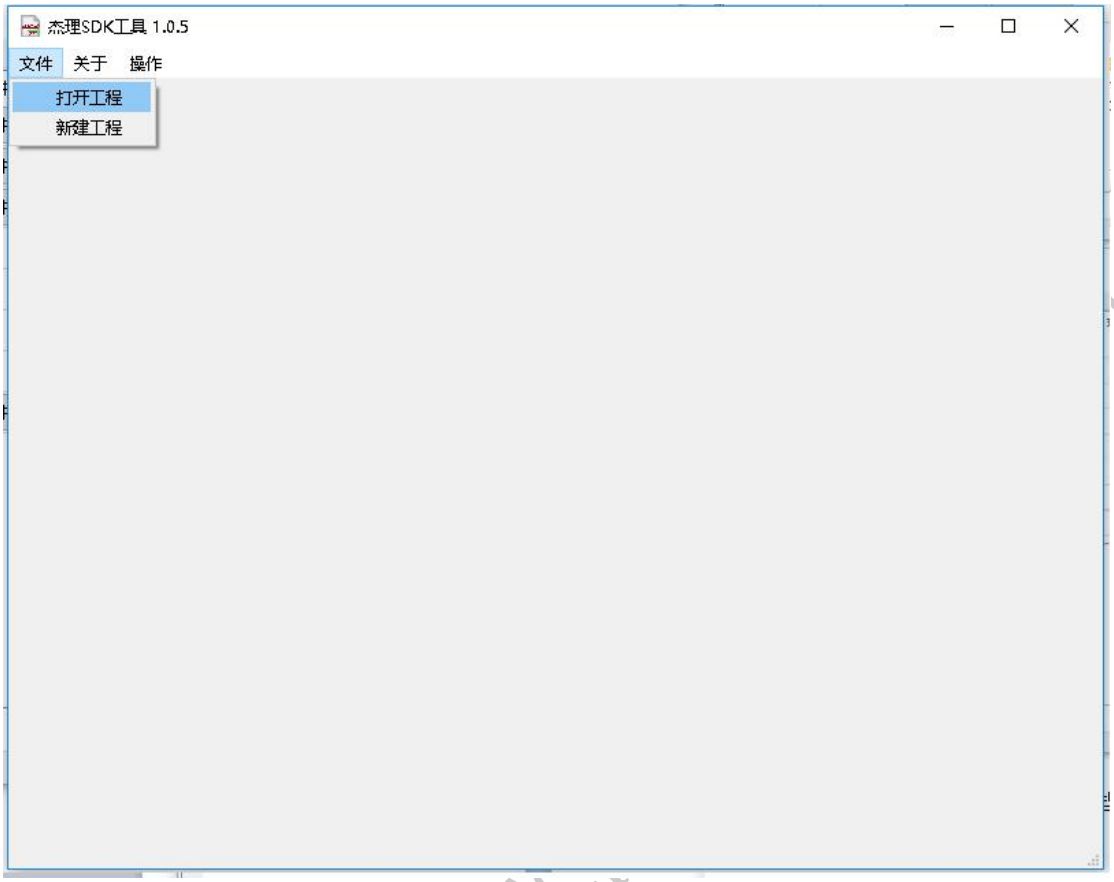
许可文件过期之后，会自动把新的许可文件发送到特定邮箱（用户第一次申请的时候提供的邮箱地址），目前暂定为每个月的 1 号过期并发送新的许可文件到邮箱中。

许可文件过期后，将不能更新或者从服务器上下载工具以及其他文件（包括新建工程使用的模板）。但是已经下载了的工具可以继续使用。

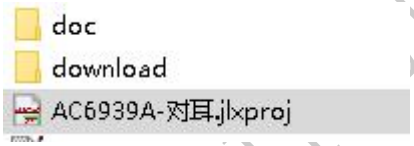
使用方式：杰理包管理器界面如下图所示：

* 并不是说有包都有执行按钮。这是因为有一些包里面没有可执行程序。

* 杰理 sdk 工具打开如下图所示：



也可以直接双击打开 jlxproj 文件，如下图所示：



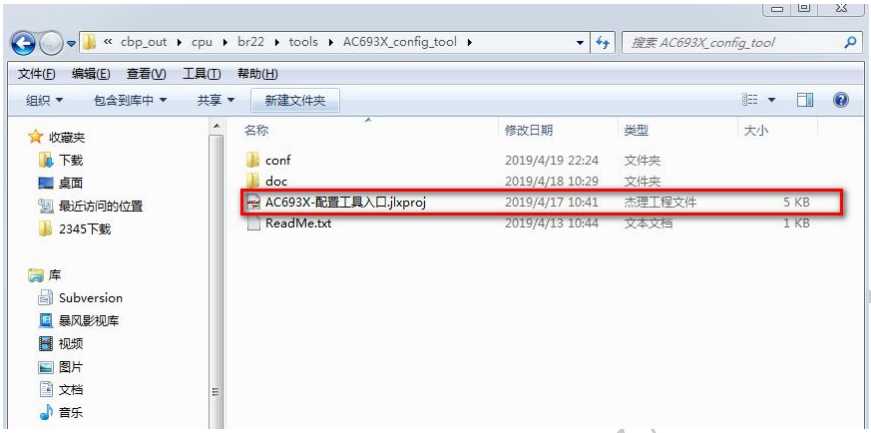
安装软件包：

也可以单独选择特定的包安装。

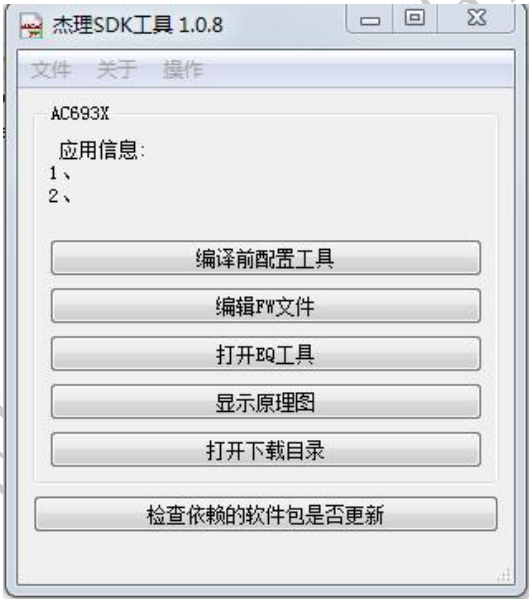
如果下载安装过的软件被破坏了、或者出现了缺失。可以点击“校验”来校验是否出现缺失或者破损，并从服务器中下载安装对应的正确文件。

1.3 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；

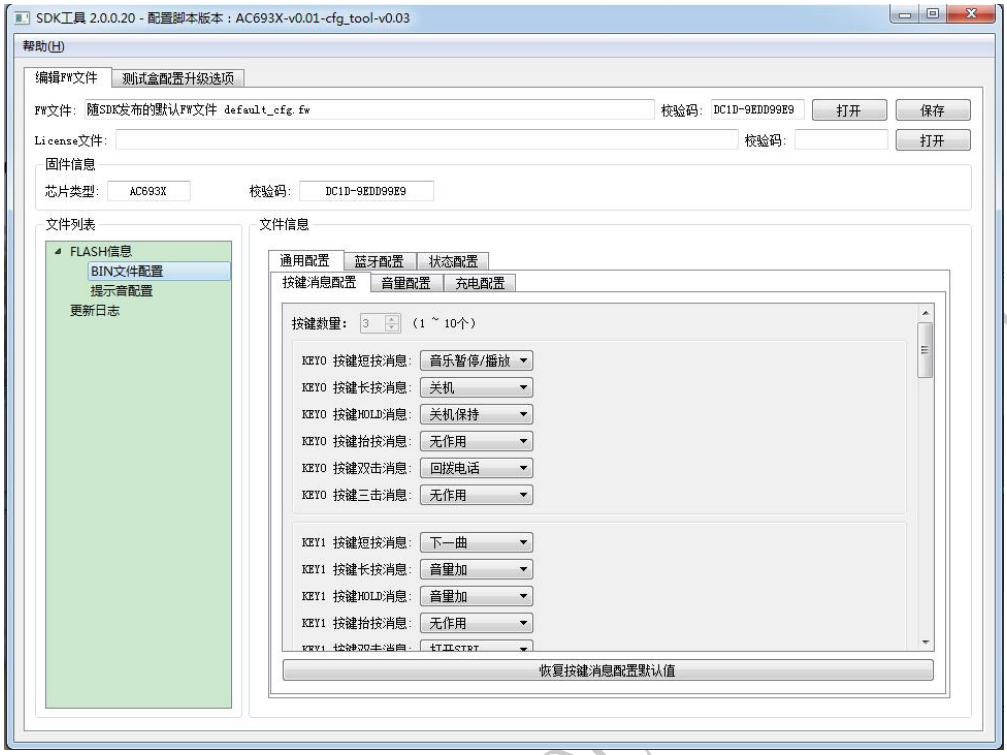


2、双击【AC630N-配置工具入口.jlxproj】，打开【杰理 SDK 工具】

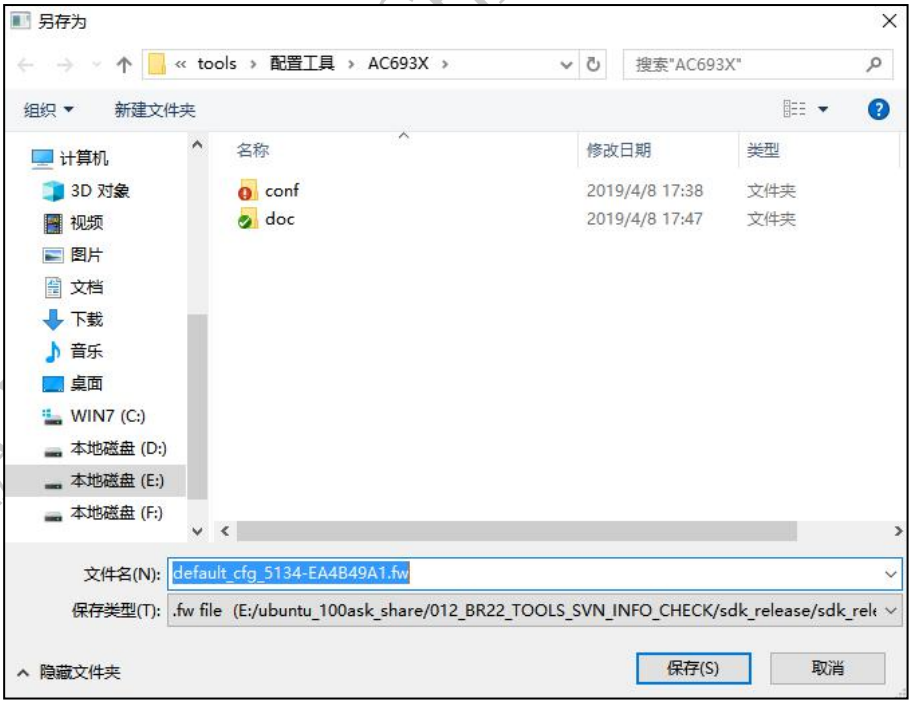


【杰理 SDK 配置工具】界面说明：

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：

```
1                          设置fw版本信息
2   cfg:addKeyInfo("script_version", "AC693X-v0.0.3");
3
4   -------------------- 设置应用名称 --------------------
5   product_name = "AC693X"; --此名称将显示于配置工具入口界面
6
7   -------------------- 设置配置工具开发状态 --------------
8       -- develop: 开发状态，用于开发SDK使用
9       -- release: 发布状态，用于发布上传使用
```

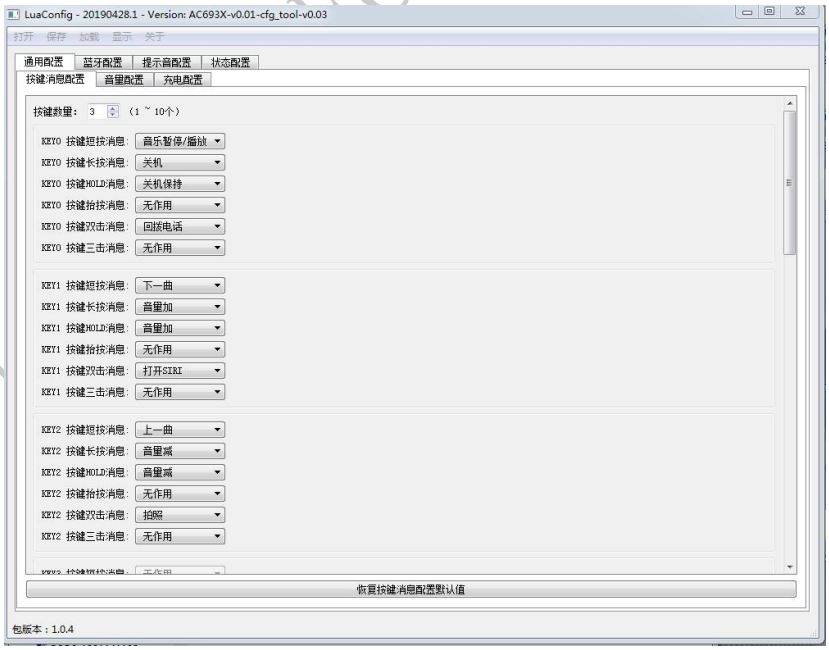
如果版本号提示不对，请确认版本号。

（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：



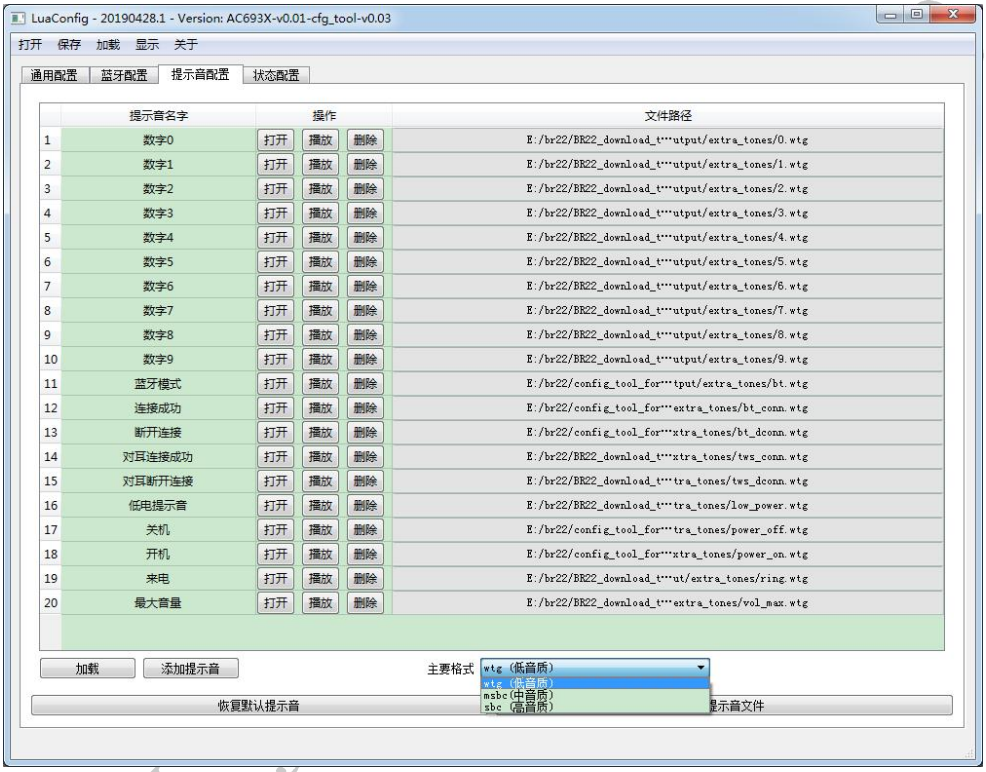
可以在编译前配置工具中进行通用配置、蓝牙配置、提示音配置和状态配置，点击【恢复默认值】可以恢复 sdk 发布时的原始配置，配置完成后，点击【保存】保存配置，将会输出 cfg_tools.bin 到下载目录中，重新编译 sdk 代码可应用该最新配置；

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。

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

支持的芯片： AC631N、AC636N、AC635N

2.2.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631X_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_AC630X_DEMO
// #define CONFIG_BOARD_AC6311_DEMO
// #define CONFIG_BOARD_AC6313_DEMO
// #define CONFIG_BOARD_AC6318_DEMO
// #define CONFIG_BOARD_AC6319_DEMO

#include "board_ac630x_demo_cfg.h"
#include "board_ac6311_demo_cfg.h"
#include "board_ac6313_demo_cfg.h"
#include "board_ac6318_demo_cfg.h"
#include "board_ac6319_demo_cfg.h"
```

```
#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

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

2.2.3 SPP 数据通信

---推荐测试工具

1、代码文件 spp_trans_data.c

2、接口说明

（1）SPP 模块初始化

```
1.  void transport_spp_init(void)
2.  {
3.      spp_state = 0;
4.      spp_get_operation_table(&spp_api);
5.      spp_api->regist_recieve_cbk(0, transport_spp_recieve_cbk);
6.      spp_api->regist_state_cbk(0, transport_spp_state_cbk);
7.      spp_api->regist_wakeup_send(NULL, transport_spp_send_wakeup);
8.
```

（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.          break;
```

```
8.
9.      case SPP_USER_ST_DISCONN:
10.         log_info("SPP_USER_ST_DISCONN ~~~\n");
11.         break;
12.     default:
13.         break;
14.     }
15. }
```

（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.          return spp_api->send_data(NULL, data, len);
8.      }
9.      return SPP_USER_ERR_SEND_FAIL;
10. }
```

（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)
2.  {
3.      putchar('W');
4.  }
```

（6）SPP 接收数据接口

```
1.  static void transport_spp_recieve_cbk(void *priv, u8 *buf, u16 len)
2.  {
3.      log_info("spp_api_rx(%d) \n", len);
4.      log_info_hexdump(buf, len);
5.      clear_sniff_cnt();
6.          ...
```

3、收发测试

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。

```
1.  //loop send data for test
2.  if (transport_spp_send_data_check(len)) {
3.      transport_spp_send_data(buf, len);
4.  }
```

4、串口的 UUID

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。

```
1.  #if (USER_SUPPORT_PROFILE_SPP==1)
2.  u8 spp_profile_support = 1;
3.  SDP_RECORD_HANDLER_REGISTER(spp_sdp_record_item) = {
4.      .service_record = (u8 *)sdp_spp_service_data,
5.      .service_record_handle = 0x00010004,
6.  };
7.  #endif
```

```
/*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, 0x02, 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, 0x08, 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

---推荐测试工具

1、代码文件 le_trans_data.c

2、Profile 生成的 profile_data 数据表放在 le_trans_data.h。用户可用工具 make_gatt_services（sdk 的 tools 目录下）自定义修改，重新配置 GATT 服务和属性等。

```
static const uint8_t profile_data[] = {
    ////////////////////////////////////////////////////
    // 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

（1）配置广播 ADV 数据

```
static int make_set_adv_data(void)
{
```

（2）配置广播 RESPONE 数据

```
static int make_set_rsp_data(void)
{
```

（3）配置发送缓存大小

```
#define ATT_LOCAL_PAYLOAD_SIZE    (200)      //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

```
/* LISTING_START(packetHandler): Packet Handler */
static void cbk_packet_handler(uint8_t packet_type, uint16_t channel, uint8_t *packet, uint16_t size)
{
    int mtu;
    u32 tmp;
    u8 status;
    switch (packet_type) {
    case HCI_EVENT_PACKET:
        switch (hci_event_packet_get_type(packet)) {
        /* case DAEMON_EVENT_HCI_PACKET_SENT: */
        /* break; */
        case ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE:
```

```
13.         log_info("ATT_EVENT_HANDLE_VALUE_INDICATION_COMPLETE\n");
14.     case ATT_EVENT_CAN_SEND_NOW:
15.         can_send_now_wakeup();
16.         break;
17.
18.     case HCI_EVENT_LE_META:
19.         switch (hci_event_le_meta_get_subevent_code(packet)) {
20.         case HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE:
```

（5）ATT 读事件处理

```
1.  static uint16_t att_read_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t offset, uint8_t
    *buffer, uint16_t buffer_size)
2.  {
3.      uint16_t  att_value_len = 0;
4.      uint16_t handle = att_handle;
5.      log_info("read_callback, handle= 0x%04x,buffer= %08x\n", handle, (u32)buffer);
6.      switch (handle) {
7.      case ATT_CHARACTERISTIC_2a00_01_VALUE_HANDLE:
8.          att_value_len = gap_device_name_len;
9.
```

（6）ATT 写事件处理

```
1.  static int 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 buffer_size)
2.  {
3.      int result = 0;
4.      u16 tmp16;
5.      u16 handle = att_handle;
6.      log_info("write_callback, handle= 0x%04x,size = %d\n", handle, buffer_size);
7.
8.      switch (handle) {
9.      case ATT_CHARACTERISTIC_2a00_01_VALUE_HANDLE:
```

```
10.     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.      if (!con_handle) {
5.          return APP_BLE_OPERATION_ERROR;
6.      }
7.      if (!att_get_ccc_config(handle + 1)) {
8.          log_info("fail,no write ccc!!!,%04x\n", handle + 1);
9.          return APP_BLE_NO_WRITE_CCC;
10.     }
11.     ret = ble_user_cmd_prepare(BLE_CMD_ATT_SEND_DATA, 4, handle, data, len, handle_type);
```

（8）检查是否可以往协议栈发送数据

```
1.  static int app_send_user_data_check(u16 len)
2.  {
3.      u32 buf_space = get_buffer_vaild_len(0);
4.      if (len <= buf_space) {
5.          return 1;
6.      }
7.      return 0;
8.  }
```

（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.  static void can_send_now_wakeup(void)
2.  {
3.      /* putchar('E'); */
4.      if (ble_resume_send_wakeup) {
5.          ble_resume_send_wakeup();
6.      }
```

调用示例

```
1.  //收发测试，自动发送收到的数据;for test
2.  if (app_send_user_data_check(buffer_size)) {
3.  app_send_user_data(ATT_CHARACTERISTIC_ae02_01_VALUE_HANDLE, buffer, buffer_size, ATT_OP_AUTO_READ_CCC);
4.  }
```

4、收发测试

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.      printf("\n-ae01_rx(%d):", buffer_size);
3.      printf_buf(buffer, buffer_size);
4.      //收发测试，自动发送收到的数据;for test
5.      if (app_send_user_data_check(buffer_size)) {
6.          app_send_user_data(ATT_CHARACTERISTIC_ae02_01_VALUE_HANDLE, buffer, buffer_size, ATT_OP_AUTO_READ_CCC);
7.      }
8.
9.  #if TEST_SEND_DATA_RATE
10.     if ((buffer[0] == 'A') && (buffer[1] == 'F')) {
11.         test_data_start = 1;//start
12.     } else if ((buffer[0] == 'A') && (buffer[1] == 'A')) {
13.         test_data_start = 0;//stop
14.     }
15. #endif
16.     break;
17.
18.  case ATT_CHARACTERISTIC_ae03_01_VALUE_HANDLE:
19.     printf("\n-ae_rx(%d):", buffer_size);
20.     printf_buf(buffer, buffer_size);
```

```
21.     //收发测试，自动发送收到的数据;for test
22.     if (app_send_user_data_check(buffer_size)) {
23.         app_send_user_data(ATT_CHARACTERISTIC_ae05_01_VALUE_HANDLE, buffer, buffer_size, ATT_O
    P_AUTO_READ_CCC);
24.     }
25.     break;
```


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

支持的芯片：AC6302A、AC6319A、AC6363F、AC6369F

2.3.2 工程配置

代码工程：apps\hid\board\bd29\AC631X_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
//#define CONFIG_BOARD_AC6313_DEMO      // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER
 //#define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
//#define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
#define CONFIG_BOARD_AC6318_DEMO     // CONFIG_APP_KEYFOB

#include "board_ac631n_demo_cfg.h"
#include "board_ac6302a_mouse_cfg.h"
#include "board_ac6319a_mouse_cfg.h"
#include "board_ac6313_demo_cfg.h"
#include "board_ac6318_demo_cfg.h"

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

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

（2）配置 app 选择

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

（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

```
5.  //2.4G 模式: 0---ble, 非 0---2.4G 配对码
6.  #define CFG_RF_24G_CODE_ID          (0) //<=24bits
```

（4）支持双模 HID 设备切换处理

```
1.  static void app_select_btmode(u8 mode)
2.  {
3.      if (mode != HID_MODE_INIT) {
4.          if (bt_hid_mode == mode) {
5.              return;
6.          }
7.          bt_hid_mode = mode;
8.      } else {
9.          //init start
10.     }
```

（5）支持进入省电低功耗 Sleep

```
1.  //*********************************************************************************//
2.  //                          低功耗配置                                             //
3.  //*********************************************************************************//
4.  // #define TCFG_LOWPOWER_POWER_SEL          PWR_DCDC15//
5.  #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)
{
```

2、经典蓝牙 EDR 模式的 HID 接口列表

（1）代码文件 hid_user.c

（2）主要接口说明

接口	备注说明
user_hid_set_icon	配置显示的图标
user_hid_set_ReportMap	配置描述符 report 表

user_hid_init	模块初始化
user_hid_exit	模块初退出
user_hid_enable	模块开关使能
user_hid_disconnect	断开连接
user_hid_msg_handler	协议栈事件处理
user_hid_send_data	发送数据接口
user_hid_send_ok_callback	协议栈发送完成回调，用来触发继续发数

3、低功耗蓝牙 BLE 模式的 HID 接口列表

（1）代码文件 le_hogp.c

（2）hogp 的 profile 的数据表放在 le_hogp.h；用户可用工具 make_gatt_services 自定义修改,重新配置 GATT 服务和属性等。

（3）主要接口说明

接口	备注说明
le_hogp_set_icon	配置显示图标
le_hogp_set_ReportMap	配置描述符 report 表
bt_ble_init	模块初始化
bt_ble_exit	模块初退出
ble_module_enable	模块开关使能
ble_disconnect	断开连接
cbk_packet_handler	协议栈事件处理
cbk_sm_packet_handler	配对加密事件处理
advertisements_setup_init	广播参数
make_set_adv_data	Adv 包数据组建
make_set_rsp_data	Rsp 包数据组建
set_adv_enable	广播开关
check_connetion_updata_deal	连接参数调整流程
att_read_callback	ATT 读事件处理
att_write_callback	ATT 写事件处理
app_send_user_data	发送数据接口
app_send_user_data_check	检查是否可以往协议栈发送数据

can_send_now_wakeup	协议栈发送完成回调，用来触发继续发数

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。

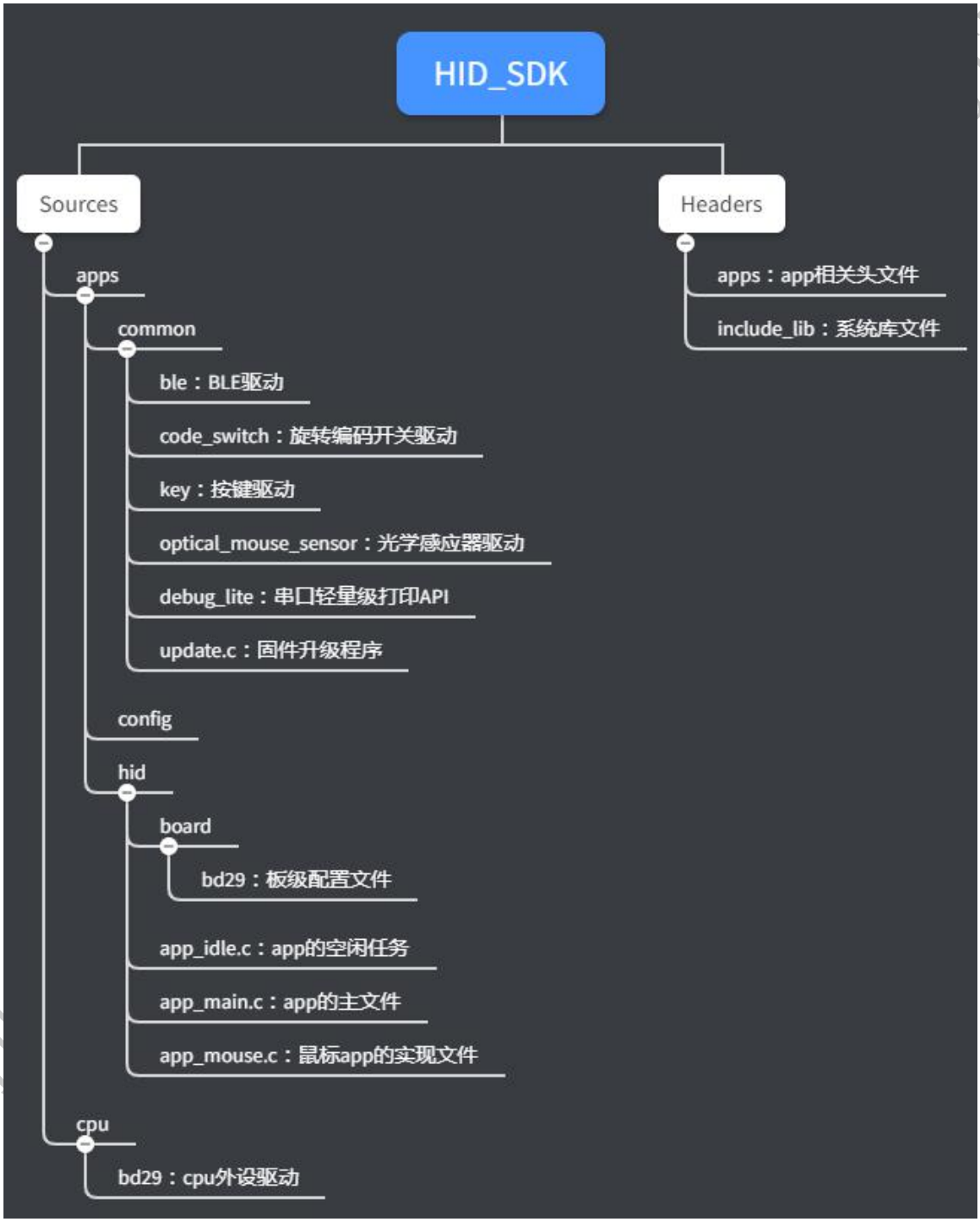


图 1.1 HID_SDK 目录结构

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。

```
/*
 * 板级配置选择
 */
// #define CONFIG_BOARD_AC630X_DEMO
#define CONFIG_BOARD_AC6302A_MOUSE
// #define CONFIG_BOARD_AC6319A_MOUSE
// #define CONFIG_BOARD_AC6313_DEMO
// #define CONFIG_BOARD_AC6318_DEMO
```

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。

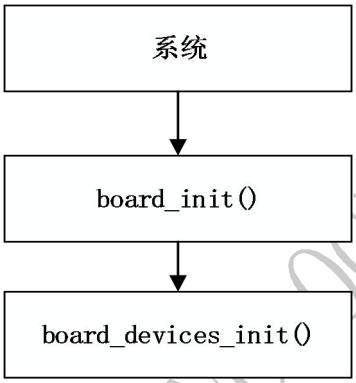


图 1.4 板级初始化流程

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。

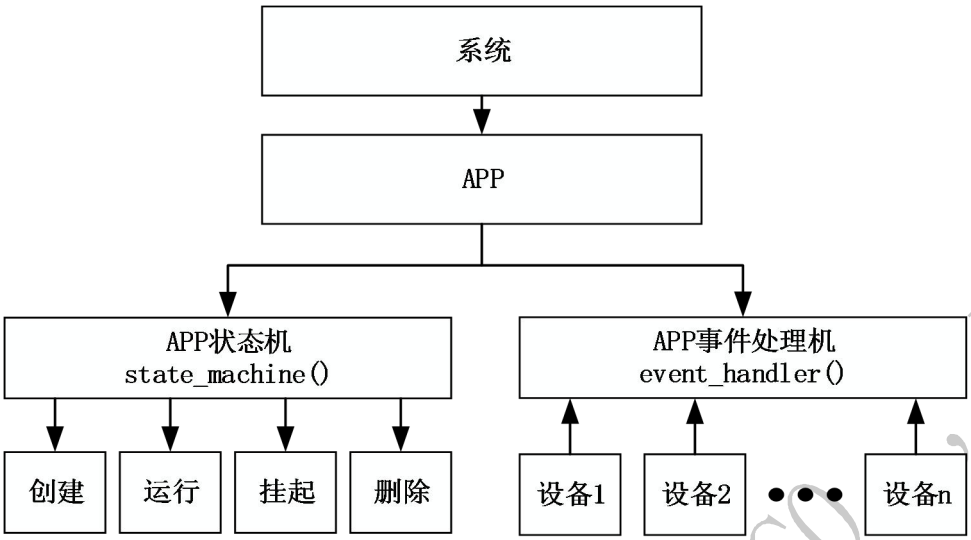


图 1.5 APP 总体框架

（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

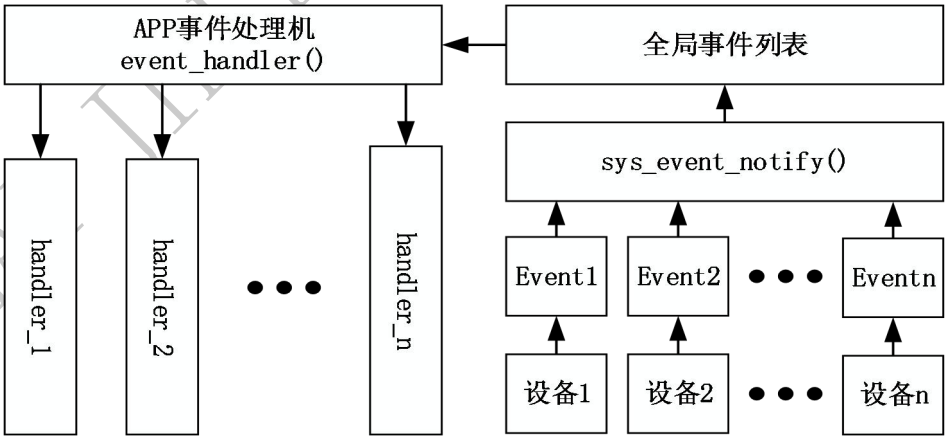


图 1.6 事件的处理过程

（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。event_handler()函数的使用示例如图 1.7 所示。

```
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.
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

```
1.  struct iokey_port {
```

```
2.    union key_type key_type;
3.    u8 connect_way;
4.    u8 key_value;
5.  };
```

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

```
1.  //iokey 配置
2.  #define TCFG_IOKEY_ENABLE                ENABLE_THIS_MOUDLE
3.
4.  #define TCFG_IOKEY_POWER_ENABLE          ENABLE_THIS_MOUDLE
5.  #define TCFG_IOKEY_POWER_CONNECT_WAY     ONE_PORT_TO_LOW  //按键一端接低电平一
    端接 IO
6.  #define TCFG_IOKEY_POWER_ONE_PORT        IO_PORTB_01
7.  #define TCFG_IOKEY_POWER_IN_PORT         NO_CONFIG_PORT
8.  #define TCFG_IOKEY_POWER_OUT_PORT        NO_CONFIG_PORT
9.
10. #define TCFG_IOKEY_PREV_ENABLE           ENABLE_THIS_MOUDLE
11. #define TCFG_IOKEY_PREV_CONNECT_WAY      ONE_PORT_TO_LOW  //按键一端接低电平一端
    接 IO
12. #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
        .key_type.one_io.port = TCFG_IOKEY_MOUSE_HK_PORT,  //IO 按键对应的引脚
        .key_value = KEY_HK_VAL,                     //按键值
    },
```

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

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

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 文件配置

```
#define TCFG_ADKEY_ENABLE                DISABLE_THIS_MOUDLE //是否使能 AD 按键
#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)
/*AD 通道选择，需要和 AD 按键的端口相对应:
   AD_CH_PA1   AD_CH_PA3   AD_CH_PA4   AD_CH_PA5
   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;
```

```
26. }
27. else   //按键抬起后等待下次延时时间未到
28. {
29.     scan_para->click_delay_cnt++;
30.     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	1. //***// 2. // UART 配置 // 3. //***//

配置

```
#define TCFG_UART0_ENABLE                   ENABLE_THIS_MOUDLE
#define TCFG_UART0_RX_PORT                  NO_CONFIG_PORT
#define TCFG_UART0_TX_PORT                  IO_PORT_DP//IO_PORTA_05
#define TCFG_UART0_BAUDRATE                 1000000
```

C 文件配置

```
/************************ UART config*****************************/
#if TCFG_UART0_ENABLE
UART0_PLATFORM_DATA_BEGIN(uart0_data)
.tx_pin = TCFG_UART0_TX_PORT,                //串口打印 TX 引脚选择
 .rx_pin = TCFG_UART0_RX_PORT,               //串口打印 RX 引脚选择
  .baudrate = TCFG_UART0_BAUDRATE,           //串口波特率
  .flags = UART_DEBUG,                       //串口用来打印需要把改参数设置为 UART_DEBUG
   UART0_PLATFORM_DATA_END()
#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。

```
0x05, 0x01,     // Usage Page (Generic Desktop Ctrls)
    0x09, 0x02,     // Usage (Mouse)
    0xA1, 0x01,     // Collection (Application)
    0x85, 0x01,     //   Report ID (1)
    0x09, 0x01,     //   Usage (Pointer)
    0xA1, 0x00,     //   Collection (Physical)
    0x95, 0x05,     //     Report Count (5)
    0x75, 0x01,     //     Report Size (1)
    0x05, 0x09,     //     Usage Page (Button)
    0x19, 0x01,     //     Usage Minimum (0x01)
    0x29, 0x05,     //     Usage Maximum (0x05)
    0x15, 0x00,     //     Logical Minimum (0)
    0x25, 0x01,     //     Logical Maximum (1)
    0x81, 0x02,     //     Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
```

```
0x95, 0x01,         //  Report Count (1)
0x75, 0x03,         //  Report Size (3)
0x81, 0x01,         //  Input (Const,Array,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x75, 0x08,         //  Report Size (8)
0x95, 0x01,         //  Report Count (1)
0x05, 0x01,         //  Usage Page (Generic Desktop Ctrls)
0x09, 0x38,         //  Usage (Wheel)
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。

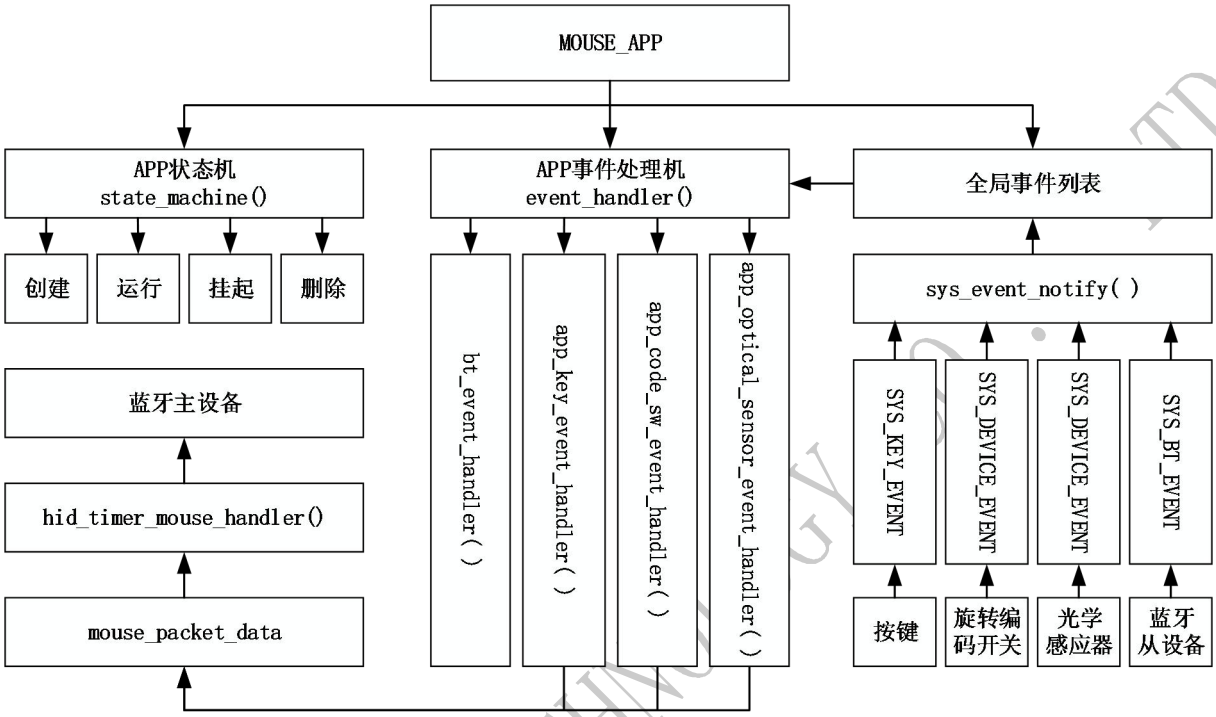


图 2.3.2 蓝牙鼠标 APP 总体框架图

（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。

```
1.  /*
2.   * 注册 AT Module 模式
3.   */
4.  REGISTER_APPLICATION(app_mouse) = {
5.      .name   = "mouse",
6.      .action = ACTION_MOUSE_MAIN,
7.      .ops    = &app_mouse_ops,
8.      .state  = APP_STA_DESTROY,
9.  };
```

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

支持的芯片： AC631N、AC636N、AC635N

2.4.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631X_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_AC630X_DEMO
// #define CONFIG_BOARD_AC6311_DEMO
// #define CONFIG_BOARD_AC6313_DEMO
// #define CONFIG_BOARD_AC6318_DEMO
// #define CONFIG_BOARD_AC6319_DEMO
```

（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

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

（3）配置 app_config.h，使能 AT

```
//app case 选择,只能选 1 个,要配置对应的 board_config.h
```

```
#define CONFIG_APP_SPP_LE                   0
#define CONFIG_APP_AT_COM                   1
#define CONFIG_APP_DONGLE                   0
```

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

```
//app case 选择,只能选 1 个,要配置对应的 board_config.h
    #define TRANS_AT_COM                    0
    #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
le_at_com.c	从机 ble 控制实现
spp_at_com.c	spp 控制实现
le_at_client.c	主机 ble 控制实现

2.5 APP - Bluetooth Dual-Mode Client

2.5.1 概述

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

支持的芯片： AC631N、AC636N、AC635N

2.5.2 工程配置

代码工程：apps\spp_and_le\board\bd29\AC631X_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_AC630X_DEMO
// #define CONFIG_BOARD_AC6311_DEMO
// #define CONFIG_BOARD_AC6313_DEMO
// #define CONFIG_BOARD_AC6318_DEMO
// #define CONFIG_BOARD_AC6319_DEMO

#include "board_ac630x_demo_cfg.h"
#include "board_ac6311_demo_cfg.h"
#include "board_ac6313_demo_cfg.h"
#include "board_ac6318_demo_cfg.h"
#include "board_ac6319_demo_cfg.h"
```

```
#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 case 选择,只能选 1 个,要配置对应的 board_config.h
#define CONFIG_APP_SPP_LE                   1
#define CONFIG_APP_AT_COM                   0
#define CONFIG_APP_DONGLE                   0
```

```
//配置对应的 APP 的蓝牙功能
#if CONFIG_APP_SPP_LE
#define TRANS_DATA_EN                       0 //蓝牙双模透传
#define TRANS_CLIENT_EN                     1 //蓝牙(ble 主机)透传
```

2.5.3 主要代码说明

BLE 实现文件 le_client_demo.c，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、主要接口说明

接口	备注说明
bt_ble_init	模块初始化
bt_ble_exit	模块初退出
ble_module_enable	模块开关使能
client_disconnect	断开连接
cbk_packet_handler	协议栈事件处理
cbk_sm_packet_handler	配对加密事件处理
scanning_setup_init	扫描参数配置
client_report_adv_data	扫描到的广播信息处理
bt_ble_scan_enable	扫描开关
client_create_connection	创建连接监听
client_create_connection_cannel	取消连接监听
client_search_profile_start	启动搜索从机的 GATT 服务
user_client_report_search_result	回调搜索 GATT 服务的结果

check_target_uuid_match	检查搜索到的 GATT 服务是否匹配目标
do_operate_search_handle	搜索完成后，处理搜索到的目标 handle。 例如：对 handle 的读操作，对 handle 写使能通知功能等
user_client_report_data_callback	接收到从机的数据，例如 notify 和 indicate 数据
app_send_user_data	发送数据接口
app_send_user_data_check	检查是否可以往协议栈发送数据
l2cap_connection_update_request_just	是否接受从机连接参数的调整求，接受后协议栈会自动更新参数
client_send_conn_param_update	主动更新连接参数调整
connection_update_complete_success	连接参数调整成功
get_buffer_vaild_len	获取发送 buf 可写如长度
can_send_now_wakeup	协议栈发送完成回调，用来触发继续发数
client_write_send	向从机写数据，需要等待从机回复应答
client_write_without_respond_send	向从机写数据，不需要从回复应答
client_read_value_send	向从机发起读数据，只能读到<=MTU 的长度数据
client_read_long_value_send	向从机发起读数据，一次读完所有数据

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）配置搜索的 services

```
typedef struct {
    uint16_t services_uuid16;
    uint16_t characteristic_uuid16;
    uint8_t services_uuid128[16];
    uint8_t characteristic_uuid128[16];
    uint16_t opt_type;
} target_uuid_t;

#if TRANS_CLIENT_EN
//指定搜索 uuid
static const target_uuid_t  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,
},
```

（2）记录搜索匹配的 handle 等信息

```
#define SEARCH_UUID_MAX   (sizeof(search_uuid_table)/sizeof(target_uuid_t))

typedef struct {
    target_uuid_t *search_uuid;
    uint16_t value_handle;
    /* uint8_t  properties; */
} opt_handle_t;

//搜索操作记录的 handle
#define OPT_HANDLE_MAX   16
static opt_handle_t opt_handle_table[OPT_HANDLE_MAX];
static u8 opt_handle_used_cnt;

typedef struct {
    uint16_t read_handle;
```

```
    uint16_t read_long_handle;
    uint16_t write_handle;
    uint16_t write_no_respond;
    uint16_t notify_handle;
    uint16_t indicate_handle;
} target_hdl_t;

//记录 handle 使用
static target_hdl_t target_handle;
```

查找到匹配的 UUID 和 handle，会执行对 handle 的读写使能通知功能等操作，如下图。

```
//操作 handle，完成 write ccc
static void do_operate_search_handle(void)
{
    u16 tmp_16;
    u16 i, cur_opt_type;
    opt_handle_t *opt_hdl_pt;

    log_info("find target_handle:");
    log_info_hexdump(&target_handle, sizeof(target_hdl_t));

    if (0 == opt_handle_used_cnt) {
        return;
    }
```

3、配值发起连接的条件

创建连接可根据名字、地址、厂家自定义标识等匹配发起连接对应的设备。用户可修改创建连接条件，如下图是可选择的条件：

```
enum {
    CLI_CREAT_BY_ADDRESS = 0,//指定地址创建连接
    CLI_CREAT_BY_NAME,//指定设备名称创建连接
```

```
    CLI_CREAT_BY_TAG,//匹配厂家标识创建连接
    CLI_CREAT_BY_LAST_SCAN,
};
```

```
static const char user_tag_string[] = {0xd6, 0x05, 'j', 'i', 'e', 'l', 'i' };
static const u8 create_conn_mode = BIT(CLI_CREAT_BY_NAME);// BIT(CLI_CREAT_BY_ADDRESS) | BIT(CLI_CREAT_BY_NAME)
static const u8 create_conn_remoter[6] = {0x11, 0x22, 0x33, 0x88, 0x88, 0x88};
```

4、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。

2.6 APP - Bluetooth BLE Dongle

2.6.1 概述

蓝牙 dongle 符合 USB 和 BLE 传输标准，具有即插即用，方便实用的特点。它可用于 BLE 设备之间的数据传输，让电脑能够和周边的 BLE 设备进行无线连接和数据的通讯，自动发现和管理远程 BLE 设备、资源和服务，实现 BLE 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。蓝牙 dongle 支持连接指定蓝牙名或 mac 地址。

支持的板级：br25

支持的芯片：AC636N

2.6.2 工程配置

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

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

```
//#define CONFIG_BOARD_AC636N_DEMO
 #define CONFIG_BOARD_AC6368B_DONGLE //CONFIG_APP_DONGLE
// #define CONFIG_BOARD_AC6363F_DEMO
// #define CONFIG_BOARD_AC6366C_DEMO
// #define CONFIG_BOARD_AC6368A_DEMO
// #define CONFIG_BOARD_AC6369F_DEMO
// #define CONFIG_BOARD_AC6369C_DEMO
```

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

```
#define TCFG_PC_ENABLE                  ENABLE_THIS_MOUDLE//PC 模块使能
#define TCFG_UDISK_ENABLE               DISABLE_THIS_MOUDLE//U 盘模块使能
#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
```

（3）APP 选择，配置 app_config.h

```
30. //app case  选择,只能选 1 个,要配置对应的 board_config.h
31. #define CONFIG_APP_SPP_LE                    0
32. #define CONFIG_APP_AT_COM                    0
33. #define CONFIG_APP_DONGLE                    1
```

（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

```
34. //2.4G 模式: 0---ble, 非 0---2.4G 配对码
35. #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

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

2.6.3 主要代码说明

蓝牙 dongle 实现文件 dongle.c，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

```
26. static const u8 sHIDReportDesc[] = {
27.     0x05, 0x01,         // Usage Page (Generic Desktop Ctrls)
28.     0x09, 0x02,         // Usage (Mouse)
29.     0xA1, 0x01,          // Collection (Application)
30.     0x85, 0x01,         //     Report ID (1)
31.     0x09, 0x01,         //     Usage (Pointer)
32.
```

（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

```
33. static const target_uuid_t   dongle_search_ble_uuid_table[] = {
34.     {
35.         .services_uuid16 = 0x1812,
36.         .characteristic_uuid16 = 0x2a4d,
37.         .opt_type = ATT_PROPERTY_NOTIFY,
38.     },
```

```
39.
40.     {
41.         .services_uuid16 = 0x1812,
42.         .characteristic_uuid16 = 0x2a33,
43.         .opt_type = ATT_PROPERTY_NOTIFY,
44.     },
45.
46.     {
47.         .services_uuid16 = 0x1801,
48.         .characteristic_uuid16 = 0x2a05,
49.         .opt_type = ATT_PROPERTY_INDICATE,
50.     },
51.
52. };
```

```
1. /*
2. 确定留给从机发数据的 3 个 notify handle
3. */
4. static const u16 mouse_notify_handle[3] = {0x0027, 0x002b, 0x002f};
```

（3）用于监听从机 notify 数据，并通过 USB 向 PC 转发蓝牙数据

```
1. static void ble_report_data_deal(att_data_report_t *report_data, target_uuid_t *sear
   ch_uuid)
2. {
3.     log_info_hexdump(report_data->blob, report_data->blob_length);
4.
5.     switch (report_data->packet_type) {
6.     case GATT_EVENT_NOTIFICATION: { //notify
7.         u8 packet[4];
8.
```

（4）连接从机的方式配置，可以选择通过地址或设备名等方式连接

```
static const client_match_cfg_t match_dev01 = {
    .create_conn_mode = BIT(CLI_CREAT_BY_NAME),                 //连接从机设备的方式:有地址连接,设备名连接,厂家标识连接
    .compare_data_len = sizeof(dongle_remoter_name1) - 1,    //去结束符(连接内容长度)
    .compare_data = dongle_remoter_name1,                       //根据连接方式,填内容
    .bonding_flag = 0,//不绑定                                    //是否配对,如果配对的话，下次就会直接连接
};
```

（5）添加从机的链接方式，将配置好的 client_match_cfg_t 结构体挂载到 client_conn_cfg_t 结构体上

```
static const client_conn_cfg_t dongle_conn_config = {
    .match_dev_cfg[0] = &match_dev01,
    .match_dev_cfg[1] = &match_dev02,
    .match_dev_cfg[2] = NULL,                                   //需要链接的从机设备,
    .report_data_callback = ble_report_data_deal,
    .search_uuid_cnt = 0, //配置不搜索 profile，加快回连速度
    /* .search_uuid_cnt = (sizeof(dongle_search_ble_uuid_table) / sizeof(target_uuid_t)), */
    /* .search_uuid_table = dongle_search_ble_uuid_table, */
    .security_en = 1,
    .event_callback = dongle_event_callback,
};
```

```
NFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURNER
// #define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
// #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
// #define CONFIG_BOARD_AC6318_DEMO    // CONFIG_APP_KEYFOB

#include "board_ac631n_demo_cfg.h"
#include "board_ac6302a_mouse_cfg.h"
#include "board_ac6319a_mouse_cfg.h"
#include "board_ac6313_demo_cfg.h"
```

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

该配置为选择对应的 board_ac631n_demo 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.  // #define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
7.  // #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
8.  // #define CONFIG_BOARD_AC6318_DEMO     // CONFIG_APP_KEYFOB
9.
10. #include "board_ac631n_demo_cfg.h"
11. #include "board_ac6302a_mouse_cfg.h"
12. #include "board_ac6319a_mouse_cfg.h"
13. #include "board_ac6313_demo_cfg.h"
14. #include "board_ac6318_demo_cfg.h"
15.
16. #endif
```

该配置选择对应的 CONFIG_BOARD_AC6313_DEMO 板级，上述两块板级都可以运行翻页器的应用 。

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

支持的芯片： AC631N、AC636N、AC635N

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. // #define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
7. // #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
8.  //#define CONFIG_BOARD_AC6318_DEMO    // CONFIG_APP_KEYFOB
```

```
#include "board_ac631n_demo_cfg.h"
#include "board_ac6302a_mouse_cfg.h"
#include "board_ac6319a_mouse_cfg.h"
#include "board_ac6313_demo_cfg.h"
#include "board_ac6318_demo_cfg.h"

#endif
```

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

```
#if (CONFIG_APP_MOUSE)
    it.name = "mouse";
    it.action = ACTION_MOUSE_MAIN;
#elif(CONFIG_APP_KEYFOB)
    it.name = "keyfob";
    it.action = ACTION_KEYFOB;
#elif(CONFIG_APP_KEYBOARD)
    it.name = "hid_key";


    start_app(&it);
}
```

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。

```
REGISTER_APPLICATION(app_hid) = {
    .name   = "hid_key",
    .action = ACTION_HID_MAIN,
    .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。

```
1.  REGISTER_LP_TARGET(app_hid_lp_target) = {
2.      .name = "app_hid_deal",
3.      .is_idle = app_hid_idle_query,
4.  };
5.  static const struct application_operation app_hid_ops = {
6.      .state_machine  = state_machine,
7.      .event_handler  = 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 state_machine 和 event_handler。执行流程大致如下，对应函数位于 app_keyboard.c 文件：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。

event_handler(struct application *app, struct sys_event *event)--->app_key_event_handler(sys_event *event)--->app_key_deal_test(key_type,key_value)---->ble_hid_key_deal_test(key_msg)。事件处理流程大致如上所示。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 event_handler(),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 event_handler()函数中调用 app_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.  define KEYBOARD_REPORT_MAP \
2.      USAGE_PAGE(CONSUMER_PAGE),          \
3.      USAGE(CONSUMER_CONTROL),            \
4.      COLLECTION(APPLICATION),            \
5.      REPORT_ID(1),     \
6.      USAGE(VOLUME_INC),                  \
7.      USAGE(VOLUME_DEC),                  \
8.      USAGE(PLAY_PAUSE),                  \
9.      USAGE(MUTE),                        \
10.     USAGE(SCAN_PREV_TRACK),             \
11.     USAGE(SCAN_NEXT_TRACK),             \
12.     USAGE(FAST_FORWARD),                \
13.     USAGE(REWIND),                      \
14.     LOGICAL_MIN(0),                     \
15.     LOGICAL_MAX(1),                     \
16.     REPORT_SIZE(1),                     \
17.     REPORT_COUNT(16),                   \
18.     INPUT(0x02),                        \
19.     END_COLLECTION,                     \
20.
21.
22. static const u8 hid_report_map[] = {KEYBOARD_REPORT_MAP};
```

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

支持芯片：AC6318、AC6368A

2.8.2 工程配置

代码工程：apps\hid\board\bd29\AC631x_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_KEYBOARD                0//hid 按键 ,default case
3.  #define CONFIG_APP_KEYFOB                  1//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）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_TURN
    ER
2.  // #define CONFIG_BOARD_AC6313_DEMO     // CONFIG_APP_KEYBOARD,CONFIG_APP_PAGE_TURN
    ER
3.  // #define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
4.  // #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
5.  #define CONFIG_BOARD_AC6318_DEMO    // CONFIG_APP_KEYFOB
6.
7.  #include "board_ac631n_demo_cfg.h"
```

```
#include "board_ac6302a_mouse_cfg.h"
#include "board_ac6319a_mouse_cfg.h"
#include "board_ac6313_demo_cfg.h"
#include "board_ac6318_demo_cfg.h"
```

（3） LED 配置（board_ac6318_demo_cfg.h）

```
//*********************************************************************************//
//                                LED 配置                                         //
//*********************************************************************************//
#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
```

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",
```

```
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 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。

```
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.
5.  #if TCFG_USER_EDR_ENABLE
6.      if (!edr_hid_is_connected()) {
7.          if (bt_connect_phone_back_start(1)) { //回连
8.              return;
9.          }
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.      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, 0x0B,       //   Report Count (11)
0x0A, 0x23, 0x02, //   Usage (AC Home)
0x0A, 0x21, 0x02, //   Usage (AC Search)
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 Nulitio
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
```

```
static const u8 key_a_big_press[3] = {0x00,0x02,0x00};
static const u8 key_a_big_null[3] =    {0x00, 0x00, 0x00};
```

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;

```

```
4.      if (bt_hid_mode == HID_MODE_EDR) {
5.  #if TCFG_USER_EDR_ENABLE
6.          hid_data_send_pt = edr_hid_data_send;
7.  #endif
8.      } else {
9.  #if TCFG_USER_BLE_ENABLE
10.         hid_data_send_pt = ble_hid_data_send;
11. #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

支持的芯片： AC631N、AC636N、AC635N

2.9.2 工程配置

代码工程：apps\hid\board\bd29\AC631X_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_KEYBOARD                0//hid 按键,default case
3. #define CONFIG_APP_KEYFOB                  0//自拍器
4. #define CONFIG_APP_MOUSE                   0//mouse
5. #define CONFIG_APP_STANDARD_KEYBOARD       0//标准 HID 键盘
6. #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
```

```
// #define CONFIG_BOARD_AC6302A_MOUSE  // CONFIG_APP_MOUSE
// #define CONFIG_BOARD_AC6319A_MOUSE  // CONFIG_APP_MOUSE
// #define CONFIG_BOARD_AC6318_DEMO    // CONFIG_APP_KEYFOB

#include "board_ac631n_demo_cfg.h"
#include "board_ac6302a_mouse_cfg.h"
#include "board_ac6319a_mouse_cfg.h"
#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。

2.9.3 主要代码说明

(1)APP 注册(函数位于 app/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。

```
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) {
        case ACTION_TOUCHSCREEN:
            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。

```
static void app_start()
{


    log_info("==========================================");
    log_info("----------------------------KEYPAGE----------------------");
    log_info("==========================================");

```

(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)

```
struct sys_event {
```

```
    u16 type;
    u8 consumed;
    void *arg;
    union {
        struct key_event key;
        struct axis_event axis;
        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().

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

#if (TCFG_HID_AUTO_SHUTDOWN_TIME)
    //重置无操作定时计数
    sys_timer_modify(g_auto_shutdown_timer, TCFG_HID_AUTO_SHUTDOWN_TIME * 1000);
#endif

```

```
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;
    void (*hid_data_send_pt)(u8 report_id, u8 * data, u16 len) = NULL;

    log_info("app_key_evnet: %d,%d\n", key_type, key_value);

    }
```

（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：

```
static const u8 hid_report_map[] = {
    // 119 bytes
    0x05, 0x0D,     // Usage Page (Digitizer)
    0x09, 0x02,     // Usage (Pen)
    0xA1, 0x01,     // Collection (Application)
    0x85, 0x01,     //  Report ID (1)
    0x09, 0x22,     //  Usage (Finger)
    0xA1, 0x02,     //  Collection (Logical)
    0x09, 0x42,     //   Usage (Tip Switch)
    0x15, 0x00,     //   Logical Minimum (0)
    0x25, 0x01,     //   Logical Maximum (1)
    0x75, 0x01,     //   Report Size (1)
    0x95, 0x01,     //   Report Count (1)
    0x81, 0x02,     //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
    0x09, 0x32,     //   Usage (In Range)
```

```
0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x95, 0x06,     //  Report Count (6)
0x81, 0x03,     //  Input (Const,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x75, 0x08,     //  Report Size (8)
0x09, 0x51,     //  Usage (0x51)
0x95, 0x01,     //  Report Count (1)
0x81, 0x02,     //  Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No Null Position)
0x05, 0x01,     //  Usage Page (Generic Desktop Ctrls)
static const u8 pp_press[7] = {0x07,0x07,0x70,0x07,0x70,0x07,0x01};
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。

```
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。

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，用于可以搜索到蓝牙名 3.0、蓝牙名 4.0 两个设备，选择任意一个进行连接，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

支持芯片：AC6351D

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_KEYBOARD                0//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       1//标准 HID 键盘,board_ac6351d
6.  #define CONFIG_APP_KEYPAGE                 0//翻页器
```

（2）先配置板级 board_config.h(apps\hid\board\br23\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_ac6351d_keyboard_cfg.h）

```
1.  //*************************************************************//
2.  //                    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_key), fn_remap_event, 1)) {
7.          return;
8.      }
9.
10.     if (special_key_deal(map, other_key, sizeof(other_key) / sizeof(special_key), other_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);
```

```
}
```

```
void touch_pad_event_deal(struct sys_event *event)
{
    u8 mouse_report[8] = {0};
    if ((event->u).touchpad.gesture_event) {
        //g_printf("touchpad gesture_event:0x%x\n", (event->u).touchpad.gesture_event);
        switch ((event->u).touchpad.gesture_event) {
        case 0x1:
            mouse_report[0] |= _KEY_MOD_LMETA;
            mouse_report[2] = _KEY_EQUAL;
            hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
            memset(mouse_report, 0x0, 8);
            hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
            return;
        case 0x2:
            mouse_report[0] |= _KEY_MOD_LMETA;
            mouse_report[2] = _KEY_MINUS;
            hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
            memset(mouse_report, 0x0, 8);
            hid_report_send(KEYBOARD_REPORT_ID, mouse_report, 8);
            return;
        case 0x3:
            mouse_report[0] |= BIT(0);          //鼠标左键
            break;
        case 0x4:
            mouse_report[0] |= BIT(1);
            break;
        }
    }
    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
```

```
    } else {
#if TCFG_USER_BLE_ENABLE
        ble_hid_data_send(report_id, data, len);
#endif
    }
}
```


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。

3.2 工程配置

代码工程：apps\mesh\board\bd29\AC631X_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”)

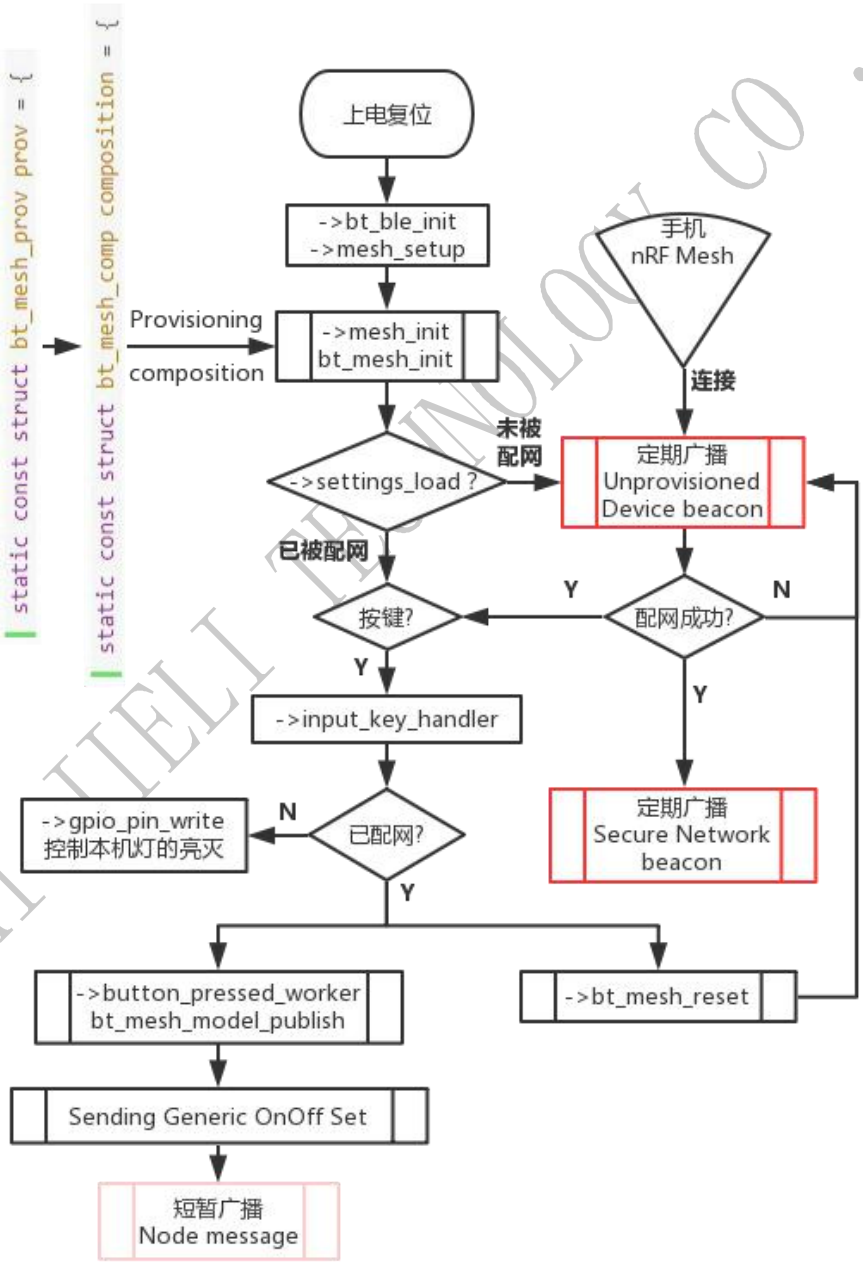
配网完成后节点结构如下：

```
▾ 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”)

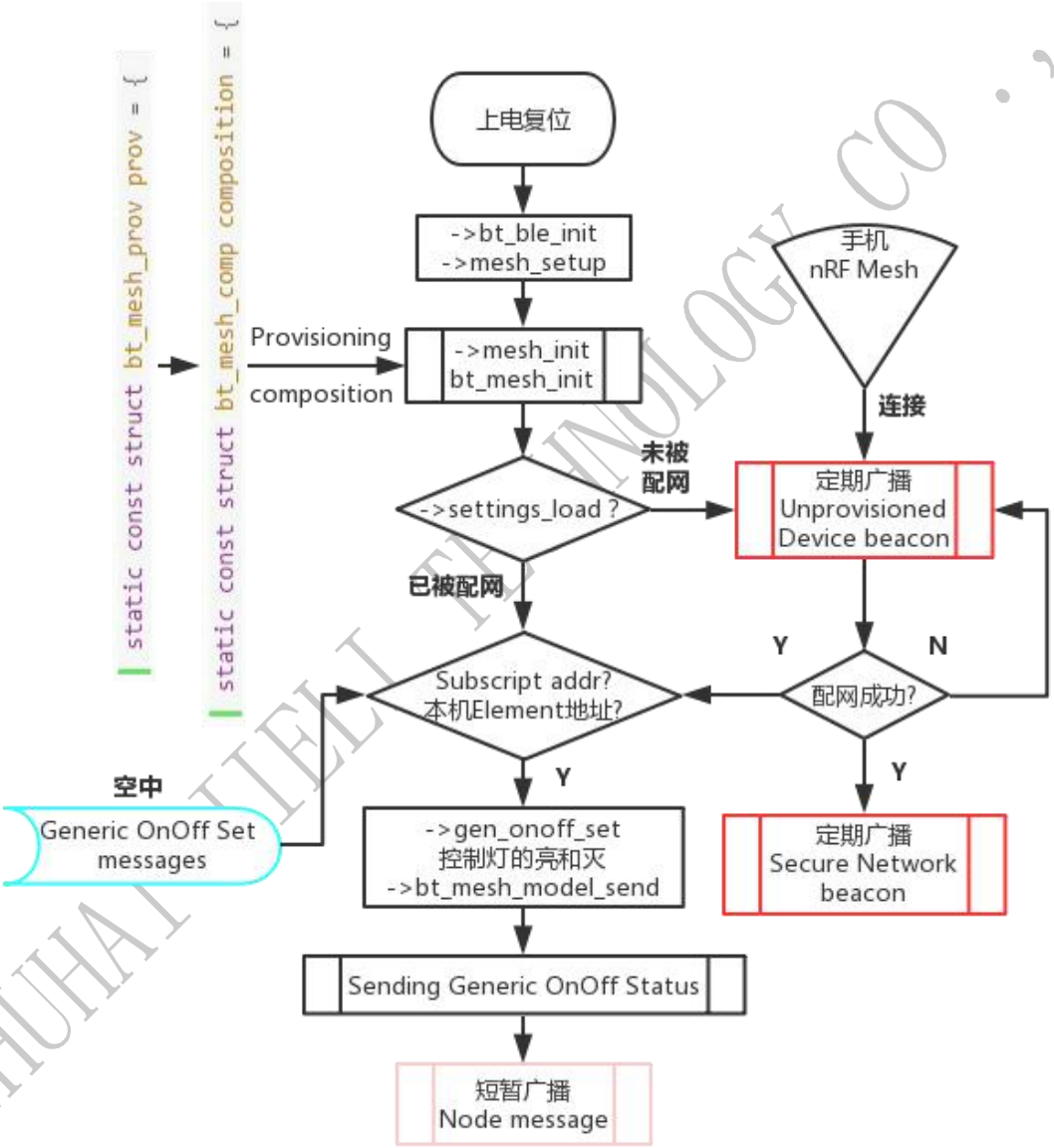
配网完成后节点结构如下：

```
▾ 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。
```

```
9.      return_value:
10.         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

```
1.  int bt_mesh_cfg_app_key_add(u16_t net_idx, u16_t addr, u16_t key_net_idx,
2.                              u16_t key_app_idx, const u8_t app_key[16],
3.                              u8_t *status);
4.      parameters:
5.          addr            节点地址
6.          app_key         应用密钥，用于保护上层传输层的通信
7.          key_app_idx     app_key 的索引
8.      return_value:
9.          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

```
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.     return_value:
12.         int 型
```

2. 为 model 添加 publish 行为

配置模型的发布状态

```
1.  int bt_mesh_cfg_mod_pub_set_vnd(u16_t net_idx, u16_t addr, u16_t elem_addr,
2.                                  u16_t mod_id, u16_t cid,
3.                                  struct bt_mesh_cfg_mod_pub *pub, u8_t *status);
4.      parameters:
5.              addr      本节点的地址，即 node_addr
6.              elem_addr  配置为 pub 中的发送属性的元素地址
7.              pub       pub 结构体存放 publish 行为的相关属性
8.                        本实例中的 pub 结构体：
9.                        struct bt_mesh_cfg_mod_pub pub;
10.                       pub.addr = dst_addr;          //publish 的目的地址
11.                       pub.app_idx = app_idx;        //app_key 的索引
12.                       pub.cred_flag = 0;            //friendship 的凭证标志
13.                       pub.ttl = 7;                  //生命周期，决定能被 relay 的次数
14.                       pub.period = 0;               //定期状态发布的周期
15.                       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）
16.
17.             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

```
1.  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

```
1.  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

```
1. 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

```
1. 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)
```

3.3.4 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

```
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 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
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式绑定在一起

```
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
    status          请求消息的状态
return_value:
    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
```

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”。

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。

4.2 OTA - APP 升级(BLE)

1、SDK 工程相关配置

1.1 在 app_config.h 打开相关的宏定义：RCSP_BTMATE_EN、RCSP_UPDATE_EN

```
1. //需要 app(BLE)升级要开一下宏定义
2. #define RCSP_BTMATE_EN                    1
3. #define RCSP_UPDATE_EN                    1
4. #define UPDATE_MD5_ENABLE                 0
```

1.2 打开 APP 升级，需要修改 ini 的话需要在 cpu\bd29\tools\bluetooth\app_ota 下修改，如果未打开 APP 升级，则修改 cpu\bd29\tools\bluetooth\standard 下的 ini 配置。对应生成的升级文件 ufw 也在对应的目录下

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