AIC Bluetooth Stack Guide

Version	date	modify by
0.0.1	2021/11/4	liliu
0.0.2	2024/1/23	xiangyuwang

1. 概述

本文档描述如何在AIC8800/AIC8800MC/AIC8800M40开发板上运行蓝牙模块，主要是描述了蓝牙驱动、patch部分以及蓝牙应用层接口的说明。

2.平台信息确认

AIC的芯片会有不同的蓝牙模块方案来单独使用或者搭配wifi模块来使用，以及不同的蓝牙模块方案对应不同的天线数量以及射频共存配置，主要有以下几个方面。

2.1.AIC8800平台信息

路径在SDK中，plf/aic8800路径下。

```
aic8800
├── config
└── src
    ├── arch
    ├── build
    └── driver
        ├── bt
        │   ├── bt_aic8800_driver.c
        │   ├── bt_aic8800_driver.h
        │   ├── bt_aic8800_drv_config.h
        │   ├── bt_aic8800_drvif.h
        │   ├── bt_common_config.c
        │   ├── bt_common_config.h
        │   ├── bt_hci.h
        │   ├── bt_patch_table.c
        │   ├── bt_patch_table.h
        │   ├── bt_user_driver.h
        │   └── patch
        │       ├── mcu
        │       │   ├── fw_patch_table_u02.c
        │       │   ├── fw_patch_table_u02.h
        │       │   ├── fw_patch_table_u03.c
        │       │   ├── fw_patch_table_u03.h
        │       │   ├── fw_patch_u02.c
```

```
│   │   ├── fw_patch_u02.h
│   │   ├── fw_patch_u03.c
│   │   └── fw_patch_u03.h
│   ├── testmode
│   │   ├── bt_combo_aud_555.c
│   │   ├── bt_combo_aud_555.h
│   │   ├── bt_combo_lite_555.c
│   │   ├── bt_combo_lite_555.h
│   │   ├── bt_only_aud_434.c
│   │   ├── bt_only_aud_434.h
│   │   ├── bt_only_lite_555.c
│   │   └── bt_only_lite_555.h
│   └── tws
│       ├── fw_patch_table_u02.c
│       ├── fw_patch_table_u02.h
│       ├── fw_patch_table_u03.c
│       ├── fw_patch_table_u03.h
│       ├── fw_patch_u02.c
│       ├── fw_patch_u02.h
│       ├── fw_patch_u03.c
│       └── fw_patch_u03.h
```

其中patch部分是rom版本firmware的patch。分别对应了mcu、testmode、以及tws专用的fw patch。

驱动中关于天线和射频模式对应的配置信息在bt_aic8800_driver.c中。

```
/*********************
  bt driver default param config

*********************/
#define AICBT_BTMODE_DEFAULT       AICBT_BTMODE_BT_ONLY
#define AICBT_BTPORT_DEFAULT       AICBT_BTPORT_MB
#define AICBT_UART_BAUD_DEFAULT    AICBT_UART_BAUD_1_5M
#define AICBT_UART_FC_DEFAULT      AICBT_UART_FLOWCTRL_ENABLE
#define AICBT_LPM_ENABLE_DEFAULT   0
#define AICBT_TXPWR_LVL_DEFAULT    AICBT_TXPWR_LVL

#define AICBSP_HWINFO_DEFAULT      (-1)
#define AICBSP_CPMODE_DEFAULT      AICBSP_CPMODE_WORK
#define AICBSP_FWLOG_EN_DEFAULT    0
```

对于BT_MODE的定义在bt_aic8800_driver.h中。

```
#define AICBT_BTMODE_BT_ONLY_SW         0// bt only mode with switch
#define AICBT_BTMODE_BT_WIFI_COMBO      1// wifi/bt combo mode
#define AICBT_BTMODE_BT_ONLY            2// bt only mode without switch
#define AICBT_BTMODE_BT_ONLY_TEST       3// bt only test mode
#define AICBT_BTMODE_BT_WIFI_COMBO_TEST 4// wifi/bt combo test mode
#define AICBT_BTMODE_BT_ONLY_COANT      5// bt only mode with no external switch
#define AICBT_MODE_NULL                 0xFF// invalid value
```

对于AIC8800这颗芯片，一共有3种天线的选择接法。（具体天线问题需要咨询我们的硬件设计团队）

软件对应的配置信息为：

1、当AIC8800的蓝牙射频使用单独的天线时，AICBT_BTMODE_DEFAULT配置为 **AICBT_BTMODE_BT_ONLY** 。

2、当AIC8800的蓝牙共用使用wifi的射频天线时，AICBT_BTMODE_DEFAULT配置为 **AICBT_BTMODE_BT_WIFI_COMBO** 。

3、当AIC8800的蓝牙使用蓝牙自己的射频，但是通过硬件switch模块切换，与wifi的射频共用一根天线时，AICBT_BTMODE_DEFAULT配置为 **AICBT_BTMODE_BT_ONLY_SW**。

AICBT_TXPWR_LVL_DEFAULT为蓝牙默认的发射功率值。可根据情况调整默认发射功率。其余信息MCU场景下无需配置。

2.2.AIC8800MC平台信息

路径在SDK中，plf/aic8800mc路径下。与AIC8800结构基本一致，不同的是名称对应IC进行了修改。

驱动中关于天线和射频模式对应的配置信息在bt_aic8800mc_driver.c中。

```
/*********************
  bt driver default param config

*********************/
#define AICBT_BTMODE_DEFAULT       AICBT_BTMODE_BT_WIFI_COMBO
#define AICBT_BTPORT_DEFAULT       AICBT_BTPORT_MB
#define AICBT_UART_BAUD_DEFAULT    AICBT_UART_BAUD_1_5M
#define AICBT_UART_FC_DEFAULT      AICBT_UART_FLOWCTRL_ENABLE
#define AICBT_LPM_ENABLE_DEFAULT   0
#define AICBT_TXPWR_LVL_DEFAULT    AICBT_TXPWR_LVL_8800dc

#define AICBSP_HWINFO_DEFAULT      (-1)
#define AICBSP_CPMODE_DEFAULT      AICBSP_CPMODE_WORK
#define AICBSP_FWLOG_EN_DEFAULT    0
```

值得注意的是AIC8800MC平台的天线方案只有一种，对应的BT_MODE射频配置方案也仅仅只有一种。按照默认的配置为**AICBT_BTMODE_BT_WIFI_COMBO**即可，无需改动。

AICBT_TXPWR_LVL_DEFAULT为蓝牙默认的发射功率值。可根据情况调整默认发射功率。其余信息MCU场景下无需配置。

2.3.AIC8800M40平台信息

路径在SDK中，plf/aic8800m40路径下。与AIC8800结构基本一致，不同的是名称对应IC进行了修改。

驱动中关于天线和射频模式对应的配置信息在bt_aic8800m40_driver.c中。

```
/*********************
  bt driver default param config

*********************/
#define AICBT_BTMODE_DEFAULT       AICBT_BTMODE_BT_ONLY
#define AICBT_BTPORT_DEFAULT       AICBT_BTPORT_MB
#define AICBT_UART_BAUD_DEFAULT    AICBT_UART_BAUD_1_5M
#define AICBT_UART_FC_DEFAULT      AICBT_UART_FLOWCTRL_ENABLE
#define AICBT_LPM_ENABLE_DEFAULT   0
#define AICBT_TXPWR_LVL_DEFAULT    AICBT_TXPWR_LVL_8800d80

#define AICBSP_HWINFO_DEFAULT      (-1)
#define AICBSP_CPMODE_DEFAULT      AICBSP_CPMODE_WORK
#define AICBSP_FWLOG_EN_DEFAULT    0
```

对于AIC8800M40这颗芯片，一共有2种天线的选择接法。（具体天线问题需要咨询我们的硬件设计团队）

软件对应的配置信息为：

1、当AIC8800M40的蓝牙射频使用单独的天线时，AICBT_BTMODE_DEFAULT配置为 **AICBT_BTMODE_BT_ONLY** 。

2、当AIC8800的蓝牙共用使用wifi的射频天线时，AICBT_BTMODE_DEFAULT配置为 **AICBT_BTMODE_BT_ONLY_COANT** 。

AICBT_TXPWR_LVL_DEFAULT为蓝牙默认的发射功率值。可根据情况调整默认发射功率。其余信息MCU场景下无需配置。

3. 经典蓝牙部分介绍

注意：由于只有AIC8800系列芯片有AIC8800A芯片，自带内置codec。因此音频相关的profile以及对应的应用，只能够在AIC8800A芯片对应的平台上直接使用，AIC8800M芯片可以通过外部IIS接口，链接外部的codec芯片来进行audio部分的播放及mic操作等。AIC8800MC以及AIC8800M40等平台，暂时只能将audio部分剥离，仅使用profile以及对应的app进行单纯协议的运行，或者配合wifi等网络或者sdio/usb等接口将数据传输至外部设备或者SOC等处理。非音频相关的profile以及app可以在AIC8800/AIC8800MC/AIC8800M40全系列芯片平台上运行。

3.1 TARGET的选择

在SDK的config目录下，有AIC8800、aic8800mc、aic8800m40等3个平台的针对不同应用场景的target。

3.1.1 aic8800

在aic8800目录下，与蓝牙相关的或者有使用蓝牙作为示例的target有以下这些。

```
.aic8800
├── target_ble
├── target_ble_wifi_fhostif
├── target_bt
├── target_bt_usb
├── target_bt_wifi_audio
├── target_btdm
├── target_btdm_wifi
├── target_dwb
└── target_tws
```

其中带bt的为经典相关应用的脚本文件夹，带ble的为低功耗蓝牙相关应用的脚本文件夹，带btdm的是dual mode也就是bt/ble共存的脚本文件夹。

用户可以选择与自己项目计划匹配的target以及target里面的脚本作为基础例程进行二次开发。以target_bt为例子，他的内部结构如下：

```
.target_bt
├── build_bt.sh
├── build_bt_gsensor.sh
├── build_bt_multiple_phone.sh
├── build_bt_ota.sh
├── build_bt_source_and_ag.sh
├── build_bt_source_sink.sh
├── build_tgt.sh
├── config
```

```
│   ├── includelist.txt
│   └── sourcelist.txt
├── lib
│   └── armgcc_4_8
│       ├── libaudio_aud.a
│       ├── libaudio_lite.a
│       ├── libbt.a
│       ├── libdrv_aud_bt.a
│       └── libdrv_lite_bt.a
├── res
│   ├── 100Sin_1CH_8K_16BIT.tab
│   ├── 1KSin_1CH_8K_16BIT.tab
│   └── a2dp_source_test.tab
├── tgt_cfg
│   └── tgt_cfg_bt.h
└── tgt_cfg_hw
    ├── hw_cfg.c
    ├── hw_cfg.h
    └── hw_cfg_api.h
```

例如build_bt.sh为经典蓝牙音响或者耳机单链接的脚本文件。根据**aic_8800_sdk_应用手册**加载好编译环境后，可直接./build_bt.sh运行编译。不同的脚本文件内部有不同的宏来对应不同的功能。

3.1.2 aic8800mc

与aic8800类似。

3.1.3 aic8800m40

与aic8800类似。

3.2 经典蓝牙应用

对于经典蓝牙或者DUAL MODE蓝牙，协议栈的初始化在/modules/bt_task目录下。

```
.bt_task
├── api
│   ├── bt_task.h
│   └── bt_task_msg.h
├── sourcelist.txt
└── src
    ├── bt_task.c
    └── bt_task_msg.c
```

bt_task_init()作为bt或者dual mode的总入口函数，在rtos中被调用启动蓝牙的task。并在task中初始化bt的stack以及profile和app。

SDK中蓝牙协议栈对应的蓝牙应用在/modules/apps目录下。

```
.apps
├── api
│   ├── app_a2dp.h
│   ├── app_a2dp_source.h
│   ├── app_avrcp.h
│   ├── app_ble_audtransmit.h
│   ├── app_ble_only.h
│   ├── app_ble_queue.h
```

```
│   ├── app_ble_wakeup.h
│   ├── app_bt.h
│   ├── app_bt_queue.h
│   ├── app_hfg.h
│   ├── app_hfp.h
│   ├── app_hsp.h
│   ├── app_ota_box.h
│   ├── app_spp.h
│   ├── app_test.h
│   └── app_tws.h
├── sourcelist.txt
└── src
    ├── ble
    │   ├── app_ble_audtransmit.c
    │   ├── app_ble_console.c
    │   ├── app_ble_only.c
    │   ├── app_ble_queue.c
    │   └── app_ble_wakeup.c
    └── bt
        ├── app_a2dp.c
        ├── app_a2dp_source.c
        ├── app_avrcp.c
        ├── app_bt.c
        ├── app_bt_queue.c
        ├── app_console.c
        ├── app_hfg.c
        ├── app_hfp.c
        ├── app_hid.c
        ├── app_hsp.c
        ├── app_ota_box.c
        ├── app_spp.c
        ├── app_test.c
        └── app_tws.c
```

SDK中蓝牙的协议栈以及profile部分被封装为lib文件libbt.a，放置在各个target目录下的lib文件夹内。stack以及各个经典蓝牙的profile的API均放置在sdk的btdm/bt目录下。

```
├── bt
│   └── include
│       ├── aic_adp_api.h
│       ├── aic_adp_type.h
│       ├── aic_host_cfg.h
│       ├── bt_aon_sram.h
│       ├── bt_types_def.h
│       ├── co_errors.h
│       ├── co_types_def.h
│       └── interface
│           ├── aic_adp_a2dp.h
│           ├── aic_adp_avrcp.h
│           ├── aic_adp_dev.h
│           ├── aic_adp_hfp.h
│           ├── aic_adp_hid.h
│           ├── aic_adp_hsp.h
```

```
│       ├── aic_adp_mgr.h
│       ├── aic_adp_spp.h
│       ├── aic_adp_test.h
│       ├── aic_adp_tws.h
│       └── aic_bt_msg.h
```

3.2.1 蓝牙协议栈的启动

蓝牙协议栈使用bt_task作为在freertos里的一个基本task循环，实现了a2dp,avrcp,hfp,spp,sdp等profiles的初始化和运行，并将部分功能集成了console指令，在app_console.c文件中给外部直接在串口上调用使用。

在static RTOS_TASK_FCT(bt_task)中，蓝牙协议栈分别进行了下列动作。

```
aic_stack_init(cfg);//协议栈的初始化
app_bt_init();//应用程的初始化
ret = aic_adp_stack_config();//协议栈的配置
if(ret == TRUE){
GLOBAL_INT_DISABLE();
aic_bt_start();//协议栈配置成功后，对firmware的poweron启动
GLOBAL_INT_RESTORE();
}
bt_task_queue_notify(false);//对主循环的notify
for (;;)
{
    uint32_t msg;
    // Suspend the BT task while waiting for new events
    rtos_queue_read(app_bt_task_queue,(void *)&msg, -1, 0);
    // schedule all pending events
    aic_stack_loop();//bt stack的轮训
    handler_reg_list_poll();
    #if PLF_BLE_STACK
    aic_ble_schedule();//ble stack的轮训
    #endif
}
```

3.2.2 修改本地蓝牙name、btaddr、iocapabilities、class of device。

在app_bt_init中，进行了对应用和profile的注册，以及对本地蓝牙name、bt_addr、io_capabilities、class_of_ device等的初始化工作。

```
void app_bt_init(void)
{
#if FPGA == 0
    bt_factory_info_t bt_factory_info;//先进行工厂区读操作，如果工厂区烧录的有默认addr，
name等信息，讲使用工厂区的，否则使用SDK code中默认地址和名称。
    if(flash_btdm_bt_factory_read((void *)&bt_factory_info,
sizeof(bt_factory_info_t)) == INFO_READ_DONE){
        uint8_t do_copy = 0;
        uint8_t null_addr[6] = {0x00,0x00,0x00,0x00,0x00,0x00};
        uint8_t invaild_addr[6] = {0xff,0xff,0xff,0xff,0xff,0xff};
        if(memcmp(bt_factory_info.local_bt_addr,null_addr,6)
        && memcmp(bt_factory_info.local_bt_addr,invaild_addr,6)){
            memcpy(bt_addr , bt_factory_info.local_bt_addr ,6);
            do_copy = 1;
        }
        if(memcmp(bt_factory_info.local_ble_addr,null_addr,6)
```

```
        && memcmp(bt_factory_info.local_ble_addr,invaild_addr,6)){
            memcpy(ble_addr , bt_factory_info.local_ble_addr ,6);
            do_copy = 1;
        }
        if(do_copy){
            memcpy((uint8_t *)bt_local_name , (uint8_t
*)bt_factory_info.local_dev_name ,32);
        }
        if(bt_factory_info.bt_flash_erased == 0x01){
            app_bt_erased_flash();
            bt_factory_info.bt_flash_erased = 0;
            flash_btdm_bt_factory_write((void
*)&bt_factory_info,sizeof(bt_factory_info_t));
        }
    }
#endif
    TRACE("app_bt_init\n");
#if PLF_CONSOLE
    app_console_command_add();//添加console
#endif
    app_set_bt_state(BT_ACTIVE_PENDING);//设置应用层状态
    app_adp_init();//中间层初始化
    app_bt_profile_register();//profile的注册
    app_register_app_msg_handle();//应用层消息处理callback函数的注册。
    app_bt_queue_init();//应用层消息队列
    app_bt_key_register_init();//开发板按键信息处理函数注册。
    app_bt_connect_init();//经典蓝牙默认开机回连流程初始化。
    app_bt_create_powerdown_timer();//默认关机流程
#if CO_MAIN_AUTO_POWER_DOWN == 1
    co_main_batt_register_cb(app_bt_send_key);
#endif
    /* 修改 bt local name */
    aic_adp_set_bt_name((const unsigned char*)bt_local_name,
strlen(bt_local_name) + 1);
    /* 修改 class of device */
aic_adp_set_cod(BTM_COD_MAJOR_AUDIO|BTM_COD_MINOR_CONFM_HEADSET|BTM_COD_SERVICE_
AUDIO|BTM_COD_SERVICE_RENDERING);
    /* 修改 io_capabilities 默认NO_INPUT_NO_OUTPUT */
    aic_adp_set_io_capabilities(NO_INPUT_NO_OUTPUT);
    TRACE("app_bt_init  success\n");
}
```

3.2.3 可发现和可连接

在协议栈初始化完成之后，app_bt.c文件中的**app_bt_common_handle**函数会提示**AIC_ADP_STACK_INIT**成功。再此之后，可以通过**app_bt_set_scan_mode**函数来设置蓝牙的**Inquiry Scan**、**Page Scan**、或者**noscan**。

也可通过默认注册的console，来通过在串口console上敲击相关指令，来打开和关闭不同的可见性。相关code均在**app_console.c**文件中。console中调用的接口，最终也是调用了协议栈提供的**app_bt_set_scan_mode**函数来实现可见性的开关。

```
console_cmd_add("scan_en", "scan_en <scan_type> <is_dut>", 3, 3, app_scan_en);

static int app_scan_en(int argc, char * const argv[])
{
    unsigned int scan_en;
```

```
    unsigned int dut = 0;

    if (argc < 2) {
        return ERR_WRONG_ARGS;
    }

    scan_en = console_cmd_strtoul(argv[1], NULL, 0);
    if (scan_en > 0x03) {
        return ERR_WRONG_ARGS;
    }
    dut = console_cmd_strtoul(argv[2], NULL, 10);
    TRACE("scan_en %d,dut %d\n",scan_en,dut);
    app_bt_wr_scan_en(scan_en,dut);

    return ERR_NONE;
}
```

3.2.4 搜索、连接配对、断开连接远端设备等通用接口

SDK提供了搜索、连接配对、断开连接远端设备的API，并将部分接口add到了app console中，可以直接使用console调用，或者二次开发编写代码在应用中调用。

```
int app_bt_inquiry_dev(unsigned int len,unsigned int maxResp)；//len为时间范围为
                        Range: 0x01 to 0x30
                        Time = len * 1.28 s
                        Range: 1.28 to 61.44 s
                    maxResp为最大返回设备的个数，协议栈对于每次搜索返回的数据做了限制，最大暂时只有10个
int app_bt_inquiry_cancel(void)；//主动取消搜索
int app_bt_role_switch(BT_ADDR* bdaddr);//主动进行主从切换
int app_bt_disconnect_acl(BT_ADDR* bdaddr);//断开选中地址的设备
int app_bt_disconnect_all_acl(void);//断开所有链接的设备
int app_bt_set_linkpolicy(BT_ADDR* bdaddr, AppBtLinkPolicy policy);//设置设备的LinkPolicy
int app_bt_set_sniff_timer(BT_ADDR *bdaddr,AppBtSniffInfo* sniff_info,TimeT Time);//设置选中设备的sniff信息
int app_bt_stop_sniff(BT_ADDR* bdaddr);//主动退出sniff
int app_bt_connect_a2dp(BT_ADDR *bdaddr);//作为a2dp sink链接已经配对过的作为a2dp source的设备
int app_bt_source_connect_a2dp(BT_ADDR *bdaddr);//作为a2dp source链接并配对作为a2dp sink的设备
int app_bt_close_a2dp(BT_ADDR *bdaddr);//仅断开a2dp的连接
int app_bt_connect_hfg(BT_ADDR *bdaddr);//作为HFP AudioGate端链接并配对作为Handfree端的设备
int app_bt_connect_hfp(BT_ADDR *bdaddr);//作为HFP Handfree端链接已经配对过的作为AudioGate端的设备
int app_bt_disconnect_hfp(BT_ADDR *bdaddr);//仅断开hfp的连接
int app_bt_hfp_connect_sco(BT_ADDR *bdaddr);//建立sco链路
int app_bt_hfp_disconnect_sco(BT_ADDR *bdaddr);//断开sco链路
```

app_bt_inquiry_dev调用后搜索设备的结果将在app_bt.c的**app_bt_manager_handle**函数中处理。

```
    case AIC_ADP_INQUIRY_RESULT:
        {
            bt_class_of_device cod = (bt_class_of_device)aic_mgr_msg->p.inqResultP.cod;
```

```
            TRACE("APP: cod = 0x%x ,addr %x %x %x %x %x %x\n",cod ,
                    aic_mgr_msg->bdaddr.addr[0],aic_mgr_msg->bdaddr.addr[1],
                    aic_mgr_msg->bdaddr.addr[2],aic_mgr_msg->bdaddr.addr[3],
                    aic_mgr_msg->bdaddr.addr[4],aic_mgr_msg->bdaddr.addr[5]);
            if(aic_mgr_msg->p.inqResultP.inqMode ==1){
                TRACE("APP: rssi = sd\n",aic_mgr_msg->p.inqResultP.rssi);
            }
            if(aic_mgr_msg->p.inqResultP.ingMode == 2){

            }
        }
        break;
```

3.2.5 a2dp相关

source或者sink的选择以及初始化时的注册，均在app_bt.c文件中**app_bt_init**的初始化流程中**app_bt_profile_register**、**app_adp_init**、**app_register_app_msg_handle**函数里。

app_bt_profile_register：注册profile。

app_adp_init：注册profile对应的中间层。

app_register_app_msg_handle：注册app层向中间层的massage callback函数。

```
void app_bt_profile_register(void)
{
#if APP_SUPPORT_A2DP_SOURCE == 1
    aic_adp_a2dp_register_source();
#endif
#if APP_SUPPORT_A2DP_CP == 1
    aic_adp_a2dp_support_cb();
#endif
#if APP_SUPPORT_A2DP_SBC == 1
    aic_adp_a2dp_register_sink_sbc();
#endif
#if APP_SUPPORT_A2DP_AAC == 1
    aic_adp_a2dp_register_sink_aac();
#endif
#if APP_SUPPORT_AVRCP == 1
    aic_adp_avrcp_register();
#endif
#if APP_SUPPORT_HFP == 1
    aic_adp_hfp_register();
#endif
#if APP_SUPPORT_HSP == 1
    aic_adp_hsp_register();
#endif
#if APP_SUPPORT_HID == 1
    aic_adp_hid_register();
#endif
}

void app_adp_init(void)
{
#if APP_SUPPORT_A2DP_SOURCE == 1
    aic_adp_a2dp_source_init();
    app_a2dp_source_init();
#endif
```

```
#if APP_SUPPORT_A2DP_SBC == 1 || APP_SUPPORT_A2DP_AAC == 1
    aic_adp_a2dp_init();
#endif
#if APP_SUPPORT_AVRCP == 1
    aic_adp_avrcp_init();
#endif
#if APP_SUPPORT_HFP == 1
    #if APP_SUPPORT_HFG == 1
    aic_adp_hfp_init(HANDSFREE_AG);
    #else
    aic_adp_hfp_init(HANDSFREE);
    #endif
#endif
#if APP_SUPPORT_HSP == 1
    aic_adp_hsp_init();
#endif
#if APP_SUPPORT_SPP == 1
    app_spp_init();
    aic_adp_spp_init();
#endif
#if APP_SUPPORT_TWS == 1
    aic_adp_tws_init();
    app_tws_init();
#endif

#if SCO_CVSD_PLC_TEST == 1 || PLF_BT_OTA == 1
    aic_adp_test_init();
#if APP_SUPPORT_OTA_BOX
    app_ota_init();
#endif
#endif
    aic_adp_mgr_init();
}

void app_register_app_msg_handle(void)
{
    aic_adp_register_app_msg_handle(AIC_COMMON ,app_bt_common_handle);
    aic_adp_register_app_msg_handle(AIC_MGR    ,app_bt_manager_handle);
#if APP_SUPPORT_A2DP_SBC == 1 || APP_SUPPORT_A2DP_AAC == 1 ||
APP_SUPPORT_A2DP_SOURCE == 1
    aic_adp_register_app_msg_handle(AIC_A2DP   ,app_a2dp_msg_handle);
#endif
#if APP_SUPPORT_AVRCP == 1
    aic_adp_register_app_msg_handle(AIC_AVRCP  ,app_avrcp_msg_handle);
#endif
#if APP_SUPPORT_HFP == 1
    aic_adp_register_app_msg_handle(AIC_HFP    ,app_hfp_msg_handle);
#endif
#if APP_SUPPORT_HSP == 1
    aic_adp_register_app_msg_handle(AIC_HSP    ,app_hsp_msg_handle);
#endif
#if APP_SUPPORT_HID == 1
    aic_adp_register_app_msg_handle(AIC_HID    ,app_hid_msg_handle);
#endif
#if APP_SUPPORT_SPP == 1
    aic_adp_register_app_msg_handle(AIC_SPP    ,app_spp_msg_handle);
#endif
#if SCO_CVSD_PLC_TEST == 1
```

```
    aic_adp_register_app_msg_handle(AIC_TEST   ,app_test_msg_handle);
#endif
#if APP_SUPPORT_TWS == 1
    aic_adp_register_app_msg_handle(AIC_TWS    ,app_tws_msg_handle);
#endif
#if PLF_BT_OTA == 1
    aic_adp_register_app_msg_handle(AIC_TEST    ,app_ota_msg_handle);
#endif
}
```

当a2dp作为sink时，不论用户使用宏控制支持SBC或者AAC编解码时，采样率协议栈均支持44.1K以及48K。

当a2dp作为source时，协议栈默认仅支持48K采样率作为stream EP的capability。

a2dp sink与source的发起连接方式参考3.2.4章节。

当作为source时，可以使用app_bt_inquiry_dev搜索到其他蓝牙的设备，然后调用int app_bt_source_connect_a2dp(BT_ADDR *bdaddr)发起链接、配对、sdp discover、a2dp链接。

当作为sink时，仅支持已经被其他手机等设备链接绑定过之后，才能使用int app_bt_connect_a2dp(BT_ADDR *bdaddr)接口对已经绑定过的设备进行回连操作。对没有绑定过的new device addr发起链接会直接返回ERROR，提示未识别的设备。

不论作为source还是sink，链接结果以及控制指令的反馈，均在注册在中间层的**void app_a2dp_msg_handle(AIC_EVENT *event)**函数中回显。

```
void app_a2dp_msg_handle(AIC_EVENT *event)
{
    AIC_ADP_A2DP_EVENT *aic_a2dp_msg = (AIC_ADP_A2DP_EVENT *)event->Param;
    app_a2dp_state_machine(event);

    switch(event->EventId) {
        case AIC_ADP_A2DP_STREAM_OPEN:
            {
                if(aic_a2dp_msg->role == BT_A2DP_SOURCE){
#if APP_SUPPORT_A2DP_SOURCE == 1
#endif
                }
                app_handle_a2dp_connect_status(event);
            }
            break;
        case AIC_ADP_A2DP_STREAM_STARTED:
            {
                if(aic_a2dp_msg->role == BT_A2DP_SOURCE){
#if APP_SUPPORT_A2DP_SOURCE == 1
#endif
                }else{
                }
            }
            break;
        case AIC_ADP_A2DP_STREAM_SUSPENDED:
            {
                if(aic_a2dp_msg->role == BT_A2DP_SINK){
                }
            }
            break;
        case AIC_ADP_A2DP_STREAM_CLOSED:
```

```
            {
                if(aic_a2dp_msg->role == BT_A2DP_SINK){
                }
                app_handle_a2dp_connect_status(event);
            }
            break;

        case AIC_ADP_A2DP_STREAM_PACKET_SENT:
            if(aic_a2dp_msg->role == BT_A2DP_SOURCE){
#if APP_SUPPORT_A2DP_SOURCE == 1
#endif
            }
            break;
        case AIC_ADP_A2DP_STREAM_DATA_IND:
            break;
        default:
            break;
    }
}
```

当a2dp作为sink时，收到的未解压的数据依靠**AIC_ADP_A2DP_STREAM_DATA_IND**来传递。

当a2dp作为source时，依靠**int app_bt_a2dp_start(BT_ADDR* bdaddr,uint32_t on)**来发送AVDTP的START与SUSPEND指令给与他链接的sink设备，从而控制对方音频codec的开关。通过**Status_BTDef aic_adp_a2dp_send_sbc_data(BT_ADDR bdaddr,U8 * data,U16 dataLen,U16 frameSize);**发送压缩好的音频数据给对端设备。

不论是source还是sink，均使用**int app_bt_close_a2dp(BT_ADDR *bdaddr)**来断开a2dp协议的链接。

a2dp使用了经过中间层封装的**aic_adp_a2dp.h**中的协议栈API来进行最终与协议栈的沟通。

3.2.6 hfp相关

1）Handsfree

当作为Handsfree端设备时。我们需要打开Inquiry scan与Page scan。使手机能够搜索并链接到HFP的profile。也仅当设备作为HF端被手机等Audio Gate设备链接过之后。我们才能使用**int app_bt_connect_hfp(BT_ADDR *bdaddr);**接口对已经链接过的设备的地址进行回连操作。当手机正常与我们设备HFP链接上之后，会在注册的**void app_hfp_msg_handle(AIC_EVENT *event)**函数中处理HFP massage的回显以及默认的一系列AT指令的动作。

```
void app_hfp_msg_handle(AIC_EVENT *event)
{
    AIC_ADP_HFP_EVENT *aic_hfp_msg = (AIC_ADP_HFP_EVENT *)event->Param;
    app_hfp_state_machine(event);

    switch(event->EventId)
    {
        case AIC_ADP_HFP_CONNECTED:
            TRACE("APP:app_hfp_connect , result = %d\n",aic_hfp_msg-
>p.conP.reason);

            if(aic_hfp_msg->p.conP.reason == BT_NO_ERROR){
                if(aic_hfp_msg->role == HANDSFREE){
                    #if APP_SUPPORT_NREC_OFF == 1
                    aic_adp_hfp_disable_NREC(aic_hfp_msg->bdaddr);
                    #endif
```

```
                    aic_adp_hfp_set_battery_level(7);
                    aic_adp_hfp_send_volume(aic_hfp_msg->bdaddr
                    ,app_get_hfp_volume(&aic_hfp_msg->bdaddr));
                    app_hfp_siri_report(aic_hfp_msg->bdaddr);
                    app_hfp_send_at_cclk(aic_hfp_msg->bdaddr);
                }else{
                #if APP_SUPPORT_HFG
                #endif
                }
            }

app_bt_handler_register(HANDLER_REG_3,aic_adp_hfp_battery_report_proc);
            app_handle_hfp_connect_status(event);
            break;
        case AIC_ADP_HFP_DISCONNECTED:
            {
                uint8_t discReason = (uint8_t)aic_hfp_msg->p.discP.reason;
                TRACE("APP:app_hfp_disconnect , reason = %d\n",discReason);
                if(aic_hfp_msg->role == HANDSFREE){
                    app_hfp_stop();
                }else{
                    #if APP_SUPPORT_HFG
                    #endif
                }
                app_bt_handler_register(HANDLER_REG_3,NULL);
                app_handle_hfp_connect_status(event);
            }
            break;
        case AIC_ADP_HFP_AUDIO_CONNECTED:
            TRACE("APP:hfp_sco_connect\n");
            #if defined(HFP_1_6_ENABLE)
            TRACE("APP:hfp_codec_type = %d\n",aic_hfp_msg-
>p.scoP.negotiated_codec);
            #endif
            if(aic_hfp_msg->role == HANDSFREE){
                aic_adp_hfp_send_volume(aic_hfp_msg->bdaddr
                ,app_get_hfp_volume(&aic_hfp_msg->bdaddr));
            }
            if(aic_hfp_msg->p.scoP.reason == BT_NO_ERROR)
                app_hfp_play(aic_hfp_msg->bdaddr);
            break;
        case AIC_ADP_HFP_AUDIO_DISCONNECTED:
            app_hfp_stop();
            break;
        case AIC_ADP_HFP_AUDIO_DATA_SENT:
            break;
        case AIC_ADP_HFP_AUDIO_DATA_IND:
            {
                AIC_ADP_HFP_AUDIO_DATA *audiodata =
                (AIC_ADP_HFP_AUDIO_DATA   *)aic_hfp_msg->p.dataindP.buff;
                AIC_ADP_HFP_AUDIO_DATA sendaudiodata;
                AppReceive_ScoData(audiodata->data ,audiodata->len);
                #if SCO_LOOPBACK_TEST == 1
                memcpy(tmpdata.buffer, audiodata->data ,audiodata->len);
                tmpdata.len = audiodata->len;
                sendaudiodata.data = tmpdata.buffer;
                sendaudiodata.len = tmpdata.len;
                TRACE("APP:sco send data = 0x%x ,len =
```

```
                %d\n",sendaudiodata.data,sendaudiodata.len);
                if(sendaudiodata.len)
                    aic_adp_hfp_send_sco_data(aic_hfp_msg-
>bdaddr,&sendaudiodata);
                #else
                uint32_t idx = hf_sendbuff.idx % HF_SENDBUFF_MEMPOOL_NUM;
                AppPrepareSend_ScoData(&hf_sendbuff.mempool[idx].buffer[0],
                audiodata->len);
                hf_sendbuff.mempool[idx].len = audiodata->len;
                sendaudiodata.data = hf_sendbuff.mempool[idx].buffer;
                sendaudiodata.len = hf_sendbuff.mempool[idx].len;
                if(sendaudiodata.len)
                    aic_adp_hfp_send_sco_data(aic_hfp_msg-
>bdaddr,&sendaudiodata);
                hf_sendbuff.idx++;
                #endif
            }
            break;
        case AIC_ADP_HFP_CALL_STATUS_IND:
            {
                bt_hfp_adp_call_status call_state = aic_hfp_msg->p.callP.callS;
                if(call_state == BT_CALL_STATUS_ACTIVE){
                    //stop playing internal audio of (ring and phone call
number),
                    and then start sco playing
                }
            }
            break;
        case AIC_ADP_HFP_CALLSETUP_STATUS_IND:
            {
                bt_hfp_adp_call_status call_state = aic_hfp_msg-
>p.callsetupP.callS;
                TRACE("APP:call_setup_state = %d\n",call_state);
            }
            break;
        case AIC_ADP_HFP_CALLHOLD_STATUS_IND:
            {
                bt_hfp_adp_callhold_status hfp_call_hold =
                aic_hfp_msg->p.callholdP.callholdS;
                TRACE("APP:hfp_call_hold = %d\n",hfp_call_hold);
            }
            break;
        case AIC_ADP_HFP_RING_IND:

            break;
        case AIC_ADP_HFP_SIRI_STATUS:
            TRACE("APP: siri status = %d\n",aic_hfp_msg->p.siriP.siriS);
            break;
        case AIC_ADP_HFP_CURRENT_CALL_NUM:

            break;
        case AIC_ADP_HFP_SPEAKER_VOLUME:
            {
                U8 volume = (U8)aic_hfp_msg->p.volP.volume;
                TRACE("APP:HFP_SPEAKER_VOLUME = %d\n",volume);
                app_set_hfp_volume(&aic_hfp_msg->bdaddr,volume);
                #if APP_SUPPORT_TWS == 0
```

```
                app_bt_local_volume_handle(1,app_get_hfp_volume(&aic_hfp_msg-
>bdaddr));
                #endif
            }
            break;
        default:
            break;
    }
}
```

设备通过**case AIC_ADP_HFP_AUDIO_DATA_IND:**消息来处理收到的SCO数据，以及通过**Status_BTDef aic_adp_hfp_send_sco_data(BT_ADDR bdaddr , AIC_ADP_HFP_AUDIO_DATA *buffer);**来发送本地的SCO数据到协议栈。

同时SDK默认设置了一些虚拟的按键KEY，来作为模拟真实耳机或者音响使用HFP时的场景，来发送HFP作为handsfree端时的各种AT指令，从而控制手机等AudioGate设备，触发一系列对应AT指令的流程。这些虚拟的KEY在初始化时注册的**BOOL app_hfp_key_handle(uint32_t key)**函数中，里面关于虚拟KEY值的消息，客户可以根据源码与硬件按键绑定，或者使用其他网络数据或者串口等模拟键值来进行绑定发送。

```
BOOL app_hfp_key_handle(uint32_t key)
{
        BOOL ret = FALSE;
        APP_DEVLIST * hfp_devinfo = NULL;
        hfp_devinfo = aic_adp_get_hfp_current_devinfo();
        if(hfp_devinfo == NULL){
            return ret;
        }

        switch(key){
            case APP_KEY_PLAY|APP_KEY_PRESS:
                {
                    TRACE("APP:hfp state %d ,call status %d\n",
                    hfp_devinfo->hfp_state,hfp_devinfo->hfp_call_status);
                    if(hfp_devinfo->hfp_state == HFP_ADP_STATE_CONNECTED){
                        if(hfp_devinfo->hfp_call_status == BT_CALL_STATUS_ACTIVE
||
                           hfp_devinfo->hfp_call_status ==
BT_CALL_STATUS_OUTGOING ||
                           hfp_devinfo->hfp_call_status == BT_CALL_STATUS_ALERT
||
                           aic_adp_hfp_get_voicerecognition_state(hfp_devinfo-
>bdaddr)){
                                aic_adp_hfp_call_release(hfp_devinfo->bdaddr);
                                ret = TRUE;
                        }else if(hfp_devinfo->hfp_call_status ==
BT_CALL_STATUS_INCOMING)                        {
                                aic_adp_hfp_call_answer(hfp_devinfo->bdaddr);
                                ret = TRUE;
                        }
                    }
                }
                break;
            case APP_KEY_VOLADD|APP_KEY_PRESS:
            case APP_KEY_VOLADD|APP_KEY_HOLD:
            case APP_KEY_VOLSUB|APP_KEY_PRESS:
            case APP_KEY_VOLSUB|APP_KEY_HOLD:
```

```
                TRACE("APP:hfp state %d ,call status %d\n",
                hfp_devinfo->hfp_state,hfp_devinfo->hfp_call_status);
                if(hfp_devinfo->hfp_call_status != BT_CALL_STATUS_NONE){
                    aic_adp_hfp_send_volume(hfp_devinfo->bdaddr
                    ,app_get_hfp_volume(&hfp_devinfo->bdaddr));
                    ret = TRUE;
                }
                break;
            case APP_KEY_PLAY|APP_KEY_DOUBLE_CLICK:
                if(hfp_devinfo->hfp_call_status == BT_CALL_STATUS_INCOMING)
                {
                    aic_adp_hfp_call_release(hfp_devinfo->bdaddr);
                    ret = TRUE;
                }
                else if(hfp_devinfo->hfp_call_status == BT_CALL_STATUS_NONE)
                {
                    aic_adp_hfp_call_redial(hfp_devinfo->bdaddr);
                    ret = TRUE;
                }
                break;
            case APP_KEY_PLAY|APP_KEY_HOLD:
                if(hfp_devinfo->hfp_call_status == BT_CALL_STATUS_INCOMING)
                {
                    aic_adp_hfp_call_release(hfp_devinfo->bdaddr);
                    ret = TRUE;
                }
                else if(hfp_devinfo->hfp_call_status == BT_CALL_STATUS_NONE)
                {
                    app_hfp_siri_voiceRecognition(hfp_devinfo->bdaddr);
                    ret = TRUE;
                }
                break;
            default:
                break;
        }
        return ret;
}
```

hfp使用了经过中间层封装的**aic_adp_hfp.h**中的协议栈API来进行最终与协议栈的沟通。例如上述例子中的：

aic_adp_hfp_call_release、 **aic_adp_hfp_call_answer** 、 **aic_adp_hfp_send_volume**

aic_adp_hfp_call_redial、 **app_hfp_siri_voiceRecognition**等等API接口。

2} AudioGate

作为AudioGate端时，可以通过**app_bt_inquiry_dev**来搜索想要链接的设备，然后调用**int app_bt_connect_hfg(BT_ADDR *bdaddr)**来发起连接。链接成功后，可以发送**int app_bt_hfp_connect_sco(BT_ADDR *bdaddr)**以及**int app_bt_hfp_disconnect_sco(BT_ADDR *bdaddr)**来主动建立SCO链路或者断开SCO链路。当然也可以由对端handsfree设备发送AT指令，设备作为AG端收到AT指令后自动建立SCO或者断开SCO链路。

与HF端一样，设备通过**case AIC_ADP_HFP_AUDIO_DATA_IND:**消息来处理收到的SCO数据，以及通过**Status_BTDef aic_adp_hfp_send_sco_data(BT_ADDR bdaddr , AIC_ADP_HFP_AUDIO_DATA *buffer);**来发送本地的SCO数据到对端设备。

4. 低功耗蓝牙部分介绍

- stop adv: `app_ble_adv_stop`(停止adv)
- update adv interval: `app_ble_adv_param_update` (如果adv正在进行，需要先停止adv,再更改参数，改完参数后会自动开始adv)
- update adv data : `app_ble_adv_data_update` (如果adv正在进行，无需停止adv)
- restart adv: `app_b1e_adv_start` (停止adv后再次开始adv)

4.1 connection api

- disconnect: `app_ble_disconnect`(断开当前连接)
- update con param: `app_ble_update_con_params`(更新连接参数)

4.2 ble enable/disable

- ble enable: `ble_task_init`
- ble disable: `ble_task_deinit`

4.3 batt set/get

- get bstt level: `app_batt_get_lvl` (获取本地电池信息,batt_min, batt_max是工作的最小和最大电压，需要根据实际需求修改batt_min,batt_max)
- end batt level : `app_batt_send_lvl` (通过ble发送电池信息)

4.4 app_callbacks

- app init: `app_on_init` (profile对应的app init函数)
- app_add_svc: `app_on_add_svc` (添加profile service instance)
- pp_enable_profile: `app_on_enable_prf`(使能profile)
- app_connection: `app_on_connection` (连接建立时的callbak函数)
- app_disconnect: `app_on_disconnect` (断开连接时的callbak函数)
- app_con_param_update: `app_on_update_params_request` (收到连接参数改变时的callback函数)
- app_adv_status: `app_on_adv_status` (adv操作(set adv data/param, enable/disable adv)完成时的callback函数)

4.5 MSG

app.c ble通用应用方法实现，包括adv, connection
app_task.c 协议栈发给应用层消息处理
`KE_MSG_HANDLER_TAB(appm)` 中的msg是协议栈发给应用层的消息,需要关注的msg的含义分别如下：

- `KE_MSG_DEFAULT_HANDLER`: 协议栈发给profile对应APP层的消息，需要对应的profile中的msg handler来处理。
- `GAPM_PROFILE_ADDED_IND`: profile添加完成
- `GAPM_ACTIVITY_CREATE_IND`:adv创建完成
- `GAPM_ACTIVITY_STOP_IND`: adv停止
- `GAPM_CMP_EVT`: 一些通用设置操作完成，比如：set adv data/param等
- `GAPM_CONNECTION_REQ_IND`: 连接建立完成
- `GAPC_APRAM_UPDATA_REQ_IND`: 连接参数更新
- `GAPC_DISCONNECT_IND`: 连接断开

profile.c profile创建，销毁等
profile_task.c 是协议栈和用户的桥梁，与协议栈和profile_app通过MSG进行数据及状态交互
app_profile.c 与profile_task.c通过MSG进行交互，处理用户数据

4.6 添加自定义profile

需要创建的文件分别是：**pfofile.c** 及 **pfofile_task.c**, **app_profile.c**及对应的.h文件。(*可参考udf(User Define Profile)部分*)

profile:

profile.c负责profile的创建，销毁等操作
profie_task.c负责协议栈和用户层的数据转发，处理。
app_pfofile.c负责处理数据

- pfofile.c:
  需要关注结构体 `prf_task_cbs`

  ```
  /// Profile task callbacks.
  struct prf_task_cbs
  {
      /// Initialization callback
      prf_init_fnct    init;  //init profile: create att database, init
  profile task handler
      /// Destroy profile callback
      prf_destroy_fnct destroy;  //free memory
      /// Connection callback
      prf_create_fnct  create;   //connection callback
      /// Disconnection callbac
      prf_cleanup_fnct cleanup;  //Disconnection callback
  };
  ```

  需要实现pfofile创建销毁等对应的回调函数，如下所示:
  `profile_init`
  `profile_destory`
  `profile_create`
  `profile_cleanup`
  定义 `profile_itf`
  接口函数 `profile_prf_its_get` ，并添加到 `prf_itf_get` 函数中
  添加属性表 `profile_att_db`
- pfofile.h
  添加服务及特征值的定义
- profile_task.c
  实现消息处理相关的函数

用户层:

- app_profile.c
  主要添加函数如下:
  `app_profile_init` 初始化 `app_profile_env` (需要把此函数添加到 `ble_user_app_init_cb` 函数中)
  `app_profile_add_profile` 发送 MSG(`GAPM_PROFILE_TASK_ADD_CMD`) 给协议栈，添加profile(需要把此函数添加到 `app_add_svc_func_list` 中)
  `app_profile_enable_prf` 发送 MSG(`PROFILE_ENABLE_REQ`) 给profile task，使能pfofile(需要把此函数添加到 `ble_user_enable_prf_cb` 函数中)
  定义消息处理函数，处理profile发来的msg、data
  定义用户数据发送接收处理函数，处理数据

- 其他头文件及编译
  rwip_task.h中添加 `TASK_ID_PROFILE`
  rwprf_config.h 中添加宏控制：`BLE_PROFILE`
  btdm文件夹中的**config/includelist.txt sourcelist.txt**中添加profile及app_profile到编译目录。

自定义profile应用例程udf(User Define Profile)（可用于客户参考，自行添加自定义profile）：

- udf例程分为client端和server端：

  Client：

  udfc.c：　　//btdm\ble\ble_profiles\udf\udfc\src

  udfc.h

  udfc_task.c：　　//btdm\ble\ble_profiles\udf\udfc\src

  udfc_task.h

  app_udfc.c:　　//btdm\ble\ble_app\app_udfc

  app_udfc.h

  Server:

  udfs.c:　　//btdm\ble\ble_profiles\udf\udfs\src

  udfs.h

  udfs_task.c　　//btdm\ble\ble_profiles\udf\udfs\src

  udfs_task.h

  app_udfs.c　　//btdm\ble\ble_app\app_udfs

  app_udfs.h

  并由应用层文件调用基础接口：

  app_ble_only.c　　//modules\apps\src\ble

  app_ble_only.h

  - Server:
    - udfs.c提供Attributes Database以及协议栈注册/注销接口，并将自定义的UUID等database注册到协议栈ATT描述中，用于其他设备检索。

```
/*
 * UDF CMD PROFILE ATTRIBUTES
 ****************************************************************************************
 */
#define udfs_service_uuid_128_content               {0xfb,0x34,0x9b,0x5f,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0x01,0xfe,0x00,0x00}
#define udfs_wr_uuid_128_content                    {0xfb,0x34,0x9b,0x5f,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0x02,0xfe,0x00,0x00}
#define udfs_rd_uuid_128_content                    {0xfb,0x34,0x9b,0x5f,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0x03,0xfe,0x00,0x00}
#define udfs_ntf_uuid_128_content                   {0xfb,0x34,0x9b,0x5f,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0x04,0xfe,0x00,0x00}
#define udfs_ind_uuid_128_content                   {0xfb,0x34,0x9b,0x5f,0x80,0x00,0x00,0x80,0x00,0x10,0x00,0x00,0x05,0xfe,0x00,0x00}

#define ATT_DECL_PRIMARY_SERVICE_UUID           {0x00, 0x28}
#define ATT_DECL_CHARACTERISTIC_UUID            {0x03, 0x28}
#define ATT_DESC_CLIENT_CHAR_CFG_UUID           {0x02, 0x29}

static const uint8_t UDFS_SERVICE_UUID_128[ATT_UUID_128_LEN]   = udfs_service_uuid_128_content;

/// Full UDFS_SERVICE Database Description - Used to add attributes into the database
const struct attm_desc_128 udfs_att_db[UDFS_IDX_NB] =
{
    // Service Declaration
    [UDFS_IDX_SVC]                  =   {ATT_DECL_PRIMARY_SERVICE_UUID, PERM(RD, ENABLE), 0, 0},

    //  Write/ Write noresponce Characteristic Declaration
    [UDFS_IDX_WR_CHAR]              =   {ATT_DECL_CHARACTERISTIC_UUID, PERM(RD, ENABLE), 0, 0},
    //  Write/ Write noresponce Characteristic Value
    [UDFS_IDX_WR_VAL]               =   {udfs_wr_uuid_128_content,
                                         PERM(WRITE_REQ, ENABLE)|PERM(WRITE_COMMAND, ENABLE),
                                         PERM(RI, ENABLE)|PERM_VAL(UUID_LEN, PERM_UUID_128), UDFS_MAX_LEN},

    // Data Read Characteristic Declaration
    [UDFS_IDX_RD_CHAR]              =   {ATT_DECL_CHARACTERISTIC_UUID, PERM(RD, ENABLE), 0, 0},
    // Data Read Characteristic Value
    [UDFS_IDX_RD_VAL]               =   {udfs_rd_uuid_128_content,
                                         PERM(RD, ENABLE),
                                         PERM(RI, ENABLE)|PERM_VAL(UUID_LEN, PERM_UUID_128), UDFS_MAX_LEN},

    // Notification Characteristic Declaration
    [UDFS_IDX_NTF_CHAR]             =   {ATT_DECL_CHARACTERISTIC_UUID, PERM(RD, ENABLE), 0, 0},
    // Notification Characteristic Value
    [UDFS_IDX_NTF_VAL]              =   {udfs_ntf_uuid_128_content, PERM(NTF, ENABLE)|PERM(RD, ENABLE),
                                         PERM(RI, ENABLE)|PERM_VAL(UUID_LEN, PERM_UUID_128), UDFS_MAX_LEN},
    // Notification Characteristic - Client Characteristic Configuration Descriptor
    [UDFS_IDX_NTF_CFG]              =   {ATT_DESC_CLIENT_CHAR_CFG_UUID, PERM(RD, ENABLE)|PERM(WRITE_REQ, ENABLE), 0, 0},


    // Indication Characteristic Declaration
    [UDFS_IDX_IND_CHAR]             =   {ATT_DECL_CHARACTERISTIC_UUID, PERM(RD, ENABLE), 0, 0},
    // Indication Characteristic Value
    [UDFS_IDX_IND_VAL]              =   {udfs_ind_uuid_128_content, PERM(IND, ENABLE)|PERM(RD, ENABLE),
                                         PERM(RI, ENABLE)|PERM_VAL(UUID_LEN, PERM_UUID_128), UDFS_MAX_LEN},
    // Indication Characteristic - Client Characteristic Configuration Descriptor
    [UDFS_IDX_IND_CFG]              =   {ATT_DESC_CLIENT_CHAR_CFG_UUID, PERM(RD, ENABLE)|PERM(WRITE_REQ, ENABLE), 0, 0},
};
```

为了方便user理解和二次开发，我们分别定义了4个不同的UUID，来作为四种不同properties的用例，分别是

//Write/Write noresponce Characteristic Declaration

udfs_wr_uuid_128_content：PERM(WRITE_REQ, ENABLE)|PERM(WRITE_COMMAND, ENABLE)

//Data Read Characteristic Declaration

udfs_rd_uuid_128_content：PERM(RD, ENABLE)

//Notification Characteristic Declaration

udfs_ntf_uuid_128_content：PERM(NTF, ENABLE)

//Indication Characteristic Declaration

udfs_ind_uuid_128_content：PERM(IND, ENABLE)

当客户需要对其中一组自定义UUID做多个properties操作时，只需要将对应的权限或入value来进行配置即可使自定义UUID有对应的property。

server端profile的添加主要接口和接口如下。

```
const struct prf_task_cbs udfs_itf =
{
   (prf_init_fnct) udfs_init,
   udfs_destroy,
   udfs_create,
   udfs_cleanup,
};

const struct prf_task_cbs* udfs_prf_itf_get(void)
{
  return &udfs_itf;
}
```

udfs_prf_itf_get需要添加在**prf_user.c**中的prf_itf_get函数中，结合在rwip_task.h注册的TASK ID使用。

自定义添加时需注意对应task的ID以及状态值。

- udfs_task.c提供作为server端时，创建对应task之后，收到client端相关gatt指令write/read等操作时的处理函数以及作为server发送notification和indication等相关基础指令。
- app_udfs.c 作为对上面两个profile文件的应用，提供向kernel注册task的调用流程，以及从app task（app_udfs.c 注册在app task）向udfs task（profile task）发送指令以及接收数据等用户层面API接口。

  void app_udfs_update_rd_value(uint8_t* data, uint32_t length);

  void app_udfs_send_notification(uint8_t* data, uint32_t length);

  void app_udfs_send_indication(uint8_t* data, uint32_t length);

  当需要自定义多个UUID均有同样的主动操作时，如两个UUID都有app_udfs_send_notification需求，此时需要User扩展一下此函数，将不同UUID对应的handle也作为写入参数引入其中。

  另外提供应用层数据处理的注册函数，方便与其他模块配合使用。
- app_ble_only.c作为对app_udfs.c中API接口与协议栈的接入流程使用。

- Cilent：

  与Server端接口类似。实现GATT协议中Cilent端支持的检索服务、write（cmd）、read，接收indication以及notification等相关操作。应用层也与Server端类似，app_ble_only.c实现接入协议栈。

  在app_udfc.c中，重点关注几个function。

```
/*
 * MESSAGE HANDLERS
 ****************************************************************************************
 */
void app_udfc_print_val_prop(uint8_t prop)
{
    if(prop & ATT_CHAR_PROP_BCAST){
        TRACE("Value property : Boardcast Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_RD){
        TRACE("Value property : Read Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_WR_NO_RESP){
        TRACE("Value property : Write Without Response Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_WR){
        TRACE("Value property : Write Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_NTF){
        TRACE("Value property : Notify Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_IND){
        TRACE("Value property : Indicate Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_AUTH){
        TRACE("Value property : Authenticated Signed Writes Permitted\n");
    }
    if(prop & ATT_CHAR_PROP_EXT_PROP){
        TRACE("Value property : Extended Properties Permitted\n");
    }
}«end app_udfc_print_val_prop»

static int app_udfc_enable_rsp_handler(ke_msg_id_t const msgid,
                                       struct udfc_enable_rsp *param,
                                       ke_task_id_t const dest_id,
                                       ke_task_id_t const src_id)
{
    TRACE("%s ,%x\n\n", __func__,param->status);
    for(uint8_t svc_idx = 0; svc_idx < param->ats->nb_svc; svc_idx++){
        struct prf_disc_sdp_svc_info *scv_info = param->ats->scv_info[svc_idx];
        TRACE("service start_hdl %d, end_hdl %d, uuid_len %d uuid: ",scv_info->start_hdl,scv_info->end_hdl,scv_info->uuid_len);
        for(uint8_t i = 0; i < scv_info->uuid_len; i++){
            TRACE("0x%x ",scv_info->uuid[i]);
        }
        TRACE("\n");
        for(uint8_t i = 0; i < scv_info->num_of_char; i++){
            TRACE("char_hdl %d, val_hdl %d, val_prop 0x%x, val_uuid_len %d, val_uuid:",
            scv_info->info[i].char_hdl,
            scv_info->info[i].val_hdl,
            scv_info->info[i].val_prop,
            scv_info->info[i].val_uuid_len);
            for(uint8_t j = 0; j < scv_info->info[i].val_uuid_len; j++){
                TRACE("0x%x ",scv_info->info[i].val_uuid[j]);
            }
            TRACE("\n");
            app_udfc_print_val_prop(scv_info->info[i].val_prop);
            if(scv_info->info[i].desc_hdl){
                TRACE("desc_hdl %d, desc_uuid_len %d, desc_uuid:",
                scv_info->info[i].desc_hdl,
                scv_info->info[i].desc_uuid_len);
                for(uint8_t j = 0; j < scv_info->info[i].desc_uuid_len; j++){
                    TRACE("0x%x ",scv_info->info[i].desc_uuid[j]);
                }
                TRACE("\n");
            }
        }«end for uint8_t i=0;i<scv_info...»
    }«end for uint8_t svc_idx=0;svc_...»
    return (KE_MSG_CONSUMED);
}«end app_udfc_enable_rsp_handler»
```

当作为client端主动链接server端后，会自动discover server端全部的database。扫描成功后会将全部的database值在app_udfc_enable_rsp_handler函数中打印出来。User可以通过打印或者参数中带有的数组，来保存需要使用到的UUID对应的handle以及properties。方便后续调用API来对选中的UUID进行对应的操作。

//Write操作

void app_udfc_wr_req(uint16_t handle, uint8_t* data, uint32_t length);

//Write no response操作

void app_udfc_wr_nores_req(uint16_t handle, uint8_t* data, uint32_t length);

//Read操作

void app_udfc_rd_req(uint16_t handle);

//Notification enable/disable操作

void app_udfc_ntf_cfg_req(uint16_t handle, bool ntf_en);

//Indication enable/disable操作

void app_udfc_ind_cfg_req(uint16_t handle, bool ind_en);

在app_ble_console.c中，我们将对应Udf server以及client的操作，都做成了console接口，方便User快速使用用例，并更快的了解运作流程。User可以在我们的UDF代码基础上直接二次开发，或者仿照UDF的写法重新添加新的profile以及对应的app来进行开发。

```
#if BLE_APP_UDFC
    console_cmd_add("ble_conn", "ble connect   eg: ble_conn <addr_type> <addr0> <addr1> <addr2> <addr3> <addr4> <addr5>", 1, 8, app_ble_connect);
    console_cmd_add("ble_conn_cancel", "ble connect   eg: ble_conn_cancel", 1, 1, app_ble_connect_cancel);
    console_cmd_add("ble_wr", "ble write   eg: ble_wr <handle> <len> <data0> <data1> ....", 1, 200, app_ble_write);
    console_cmd_add("ble_wr_nores", "ble write no response   eg: ble_wr_nores <handle> <len> <data0> <data1> ....", 1, 200, app_ble_write_no_response);
    console_cmd_add("ble_rd", "ble read   eg:  ble_rd <handle>", 1, 2, app_ble_read);
    console_cmd_add("ble_ntf_cfg", "ble set notification cfg   eg: ble_ntf_cfg <handle> <1/0>", 1, 3, app_ble_notification_cfg);
    console_cmd_add("ble_ind_cfg", "ble set indication cfg   eg: ble_ind_cfg <handle> <1/0>", 1, 3, app_ble_indication_cfg);
#endif
#if BLE_APP_UDFS
    console_cmd_add("ble_updata_rdval", "ble update read value   eg: ble_updata_rdval <len> <data0> <data1> ....", 1, 200, app_ble_update_read_value);
    console_cmd_add("ble_sd_ntf", "ble send notification   eg: ble_sd_ntf <len> <data0> <data1> ....", 1, 200, app_ble_send_notification);
    console_cmd_add("ble_sd_ind", "ble send indication   eg: ble_sd_ind <len> <data0> <data1> ....", 1, 200, app_ble_send_indication);
#endif
```