FR5080 SDK User Guide

Bluetooth dual mode SOC

2021-07 v1.2.1

www.freqchip.com

FREQCHIP 富芮坤

目录

1. 概况

本文档是 FR5080 SDK 的应用开发指导。FR5080 是单芯片 BT&BLE 双模 SOC。FR5080 SDK 是运行于 FR5080 上的软件包，包含了 BT&BLE 5.0 的完整协议栈，芯片的外设驱动，操作系统抽象层 OSAL 及 DSP 驱动。应用层可通过 FR5080 SDK 提供的接口实现与 BTDM 协议栈的交互，访问使用外设，运行自己的任务。

1.1 FR5080 软件架构

FR5080 的软件架构如下图所示。FR5080 包含了完整的 BT&BLE 5.0 双模协议栈，外设，应用层及软件开发工具包。其中蓝牙协议栈的 controller 和 host 部分及操作系统抽象层 OSAL 都是以库的形式提供，图中为灰色蓝色部分。黄色部分为 SDK，提供接口给应用层开发，绿色部分为用户自己的程序及相关的外设驱动。

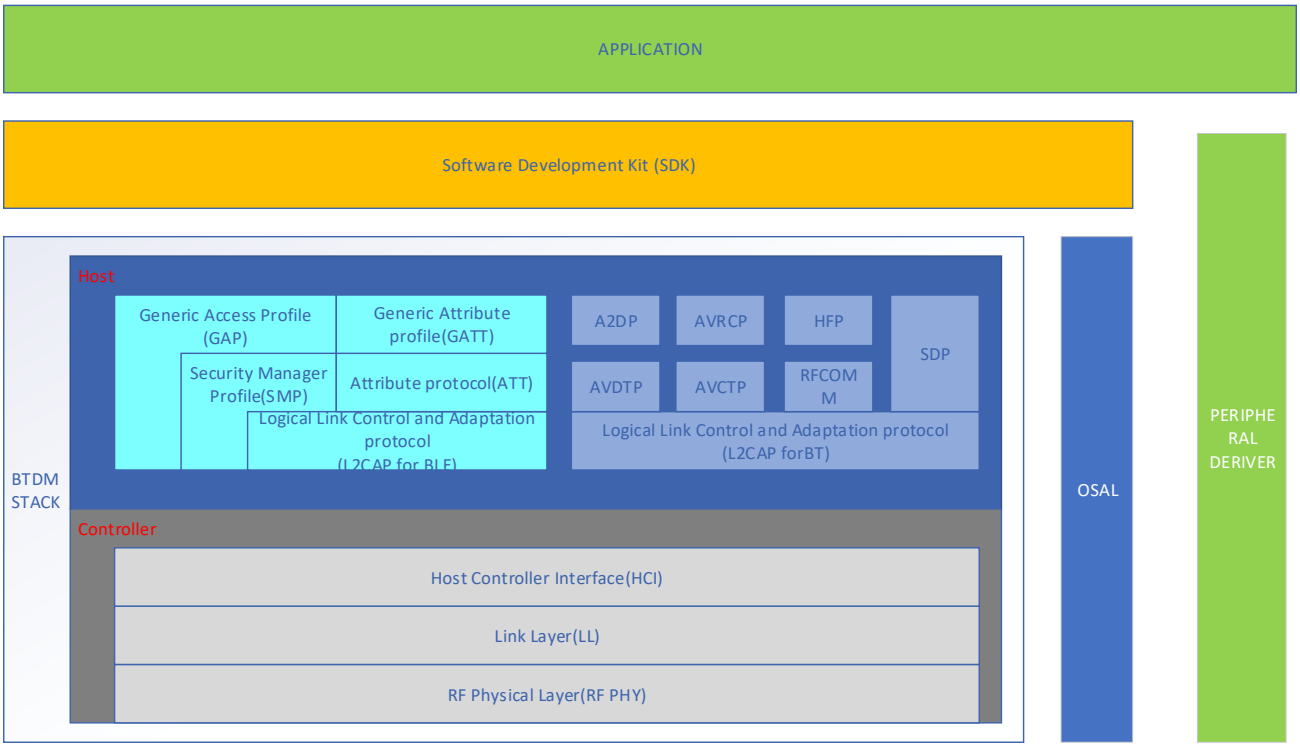


图 1.1 FR5080 软件架构

在该软件架构中，应用层通过 SDK 提供的接口实现与 BTDM 协议栈的交互，开发者可调用 BTDM 协议栈的 GAP,GATT,AVRCP,HFP,A2DP,ME 的 API 及通过注册回调函数处理相应事件

1.2 空间分配

1.2.1 地址空间

FR5080 的地址空间如下：

图 1.2 FR5080 地址空间

其中内置 480KB ROM，主要内容为启动代码、BLE 和 BT 协议栈；FLASH 空间用于存储用户程序、用户数据等；RAM 用于存储各种变量、堆栈、重新映射后的中断向量地址、对运行速度较为敏感的代码（中断响应等）等，该空间都支持低功耗的 retention 功能；外设地址空间是各种外设的地址映射，用于进行外设的配置。

1.2.2 空间分配

在 FR5080 中 FLASH 空间和 RAM 空间的分配由链接脚本指定，具体分配如下：

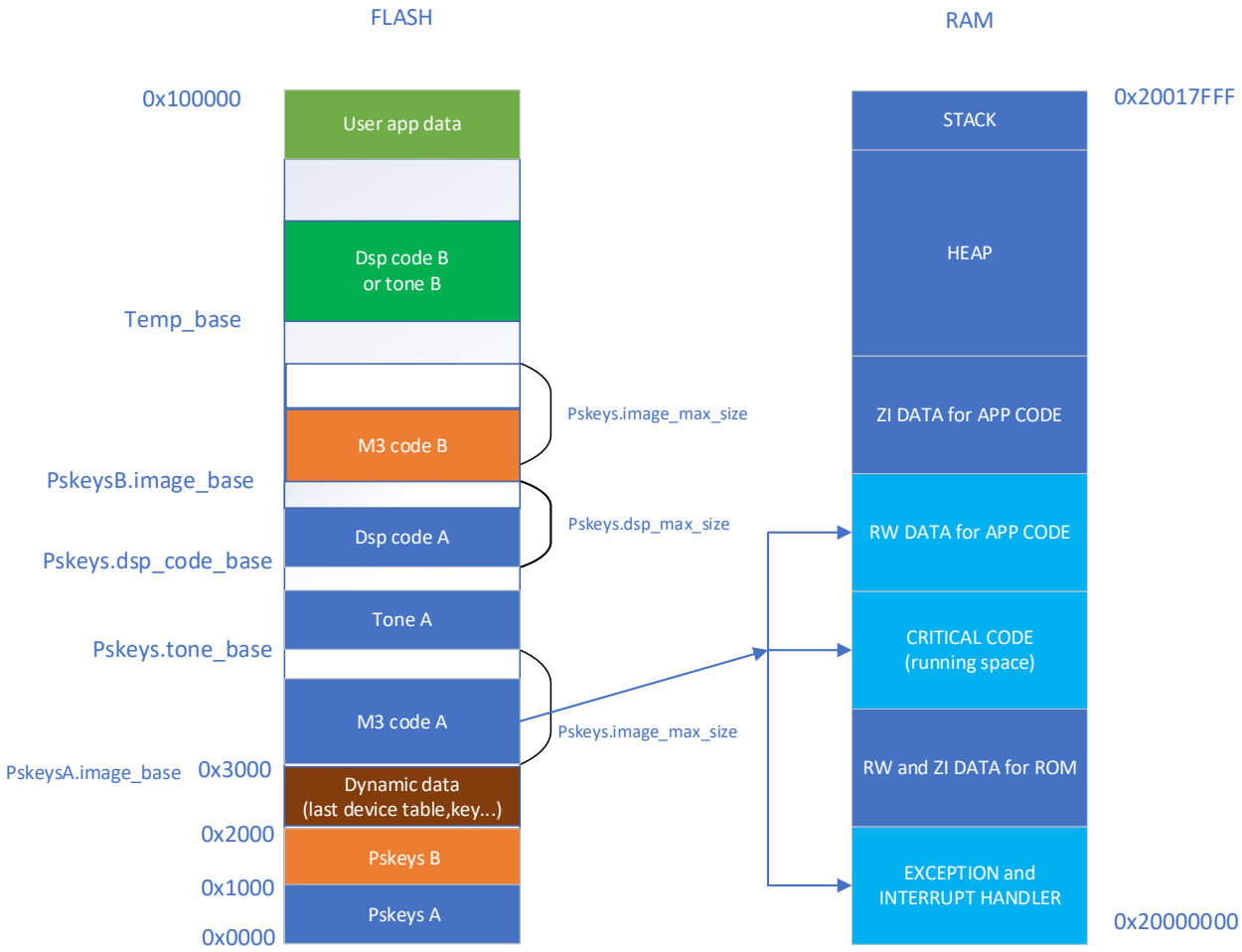


图 1.3 FR5080 flash 和 RAM 空间分配

其中 PSKEYS 存储的是配置信息；DYNAMIC DATA 存放蓝牙连接过的设备列表，包含地址，linkkey，音量等相关信息；TONE 为客户定制的提示音信息；DSP CODE 为用户 DSP 运行代码，需按照固定格式生成，MCU 在某种条件下，会将该部分代码通过 IPC 加载到 DSP；APP CODE 和 RO DATA 可以通过 XIP 被 MCU 直接访问；CRITICAL CODE 和 EXCEPTION and INTERRUPT HANDLER 为对运行时间敏感的用户代码，需要在初始化时从 flash 中搬移到 RAM 中；RW DATA 需要进行初始化；ZI 为初始值为 0 的数据段。这些操作均由 SDK 内部进行处理，用户无需做额外操作。HEAP 为动态内存分配空间，SDK 中会根据实际可用空间对内存管理单元进行初始化；STACK 为堆栈空间，生长空间由高到低，大小可由用户指定。上图中的后缀 A 和 B 分别代表 A 区和 B 区，主要是考虑 OTA 的备份，关于 OTA 的使用，可以参考下述章节。

1.3 Pskeys 配置参数

Pskeys 中保存了 FR5080 的一些配置信息，并存储在 flash 中，每次启动时，将从 Flash 中读取默认配置。其存储结构如下：

```
1.  __packed struct pskeys_t {
2.      /// 识别改配置文件为 5080 软件
3.      uint8_t reserved[4]; //'5','0','8','0'
```

```
    ///以下参数修改需要在配置工具配置
    /*****************************************************************************/
    ///以下几个参数推荐使用默认参数，fw_version 随版本更新
    uint32_t image_base;    //固件 flash 存储基地址，建议值 0x3000，若要 OTA 升级，此地址需更改到
                            //不影响升级前的 flash 空间，具体设置请参考 OTA 相关
    uint32_t image_size;        //固件大小，工具导入，自动生成
    uint8_t patch_mode; //保留，内部使用
    uint32_t fw_version;        //软件版本
    uint32_t stack_top_address; //栈顶地址，使用默认值，不建议修改
    uint16_t stack_size;            //栈长度
    uint32_t rwip_heap_base;    //堆基地址，内部自动生成
    uint32_t rwip_heap_length;  //堆长度，内部自动生成
    void * entry;               //保留，内部使用
    uint32_t dsp_code_base;     //dsp code 存放基地址
    uint32_t dsp_code_size;     //dsp code 长度
    uint32_t tone_base;         //提示音存放基地址
    uint8_t initial_qspi_bandrate;  //初始 qspi 波特率
    /*****************************************************************************/
    ///工具和 user_custom_parameters 函数中都可修改
    uint8_t tws_mode;          //是否是 BLE HCI 模式
    uint8_t localname[BT_NAME_MAX_LEN]; //蓝牙名称
    struct bdaddr_t local_bdaddr;       //蓝牙地址
    struct class_of_device_t classofdevice; //蓝牙设备类型
    uint8_t enable_profiles;        //使能相关 profile，参考 pskeys.h
    uint32_t system_options;    //使能相关 option ，参考 pskeys.h
    uint8_t stack_mode;         //选择蓝牙协议栈模式，参考 enum stack_mode_t
    /*****************************************************************************/
     //睡眠相关，内部使用，谨慎修改
    uint32_t slp_max_dur;
    uint8_t sleep_algo_dur;// uint: 312.5us, target sleep time - actual sleep time
    uint16_t twext;         //uint: us
    uint16_t twosc;         //uint: us
    uint16_t lp_cycle_sleep_delay;  //uint: lp cycle, minimum sleep time
    uint8_t sleep_delay_for_os; //uint: ms
    uint8_t handshake_to;
    /*****************************************************************************/
    ///供 APP 使用，用户可自定义复用下面参数
    uint8_t  a2dp_vol;
    uint8_t  hf_vol;
    uint8_t  tone_vol;
    struct reconnect_param_t power_on_reconnect;
    struct reconnect_param_t press_button_reconnect;
```

```
    struct reconnect_param_t link_loss_reconnect;
    uint8_t pairing_to_standby;
    uint8_t standby_to_off;
    uint8_t pmu_mode;
    uint8_t pmu_charger_poweron_sel;
    /*****************************************************************************/
    ///下面一组由配置工具导入或更改，user_custom_parameters 函数不可修改
    uint16_t tone_A_size[26];       ///A 组提示音长度列表
    uint16_t tone_B_size[26];       ///B 组提示音长度列表
    uint32_t image_max_size;    ///固件补丁最大长度
    uint32_t dsp_max_size;      ///DSP code 最大长度
    /*****************************************************************************/
     ///校验值，不要修改
     uint32_t checkword; ///0x51 0x52 0x51 0x52
    /*****************************************************************************/
    ///APP 可变数据段，lib 自动写入读出，工具和 user_custom_parameters 函数中勿写入
    struct pskeys_app_t app_data;
};
```

1.4 代码流程及入口函数

SDK 包含了四大部分，Application 部分，蓝牙协议栈部分，操作系统抽象层 OSAL 部分，还有 MCU 外设驱动部分。

整个代码结构比较简单，执行流程也很清晰易懂。SDK 的 main 函数主体入口位于 lib 库中，对于应用层以源码形式开放了一些入口，用于应用开发初始化，基本流程如下图所示：

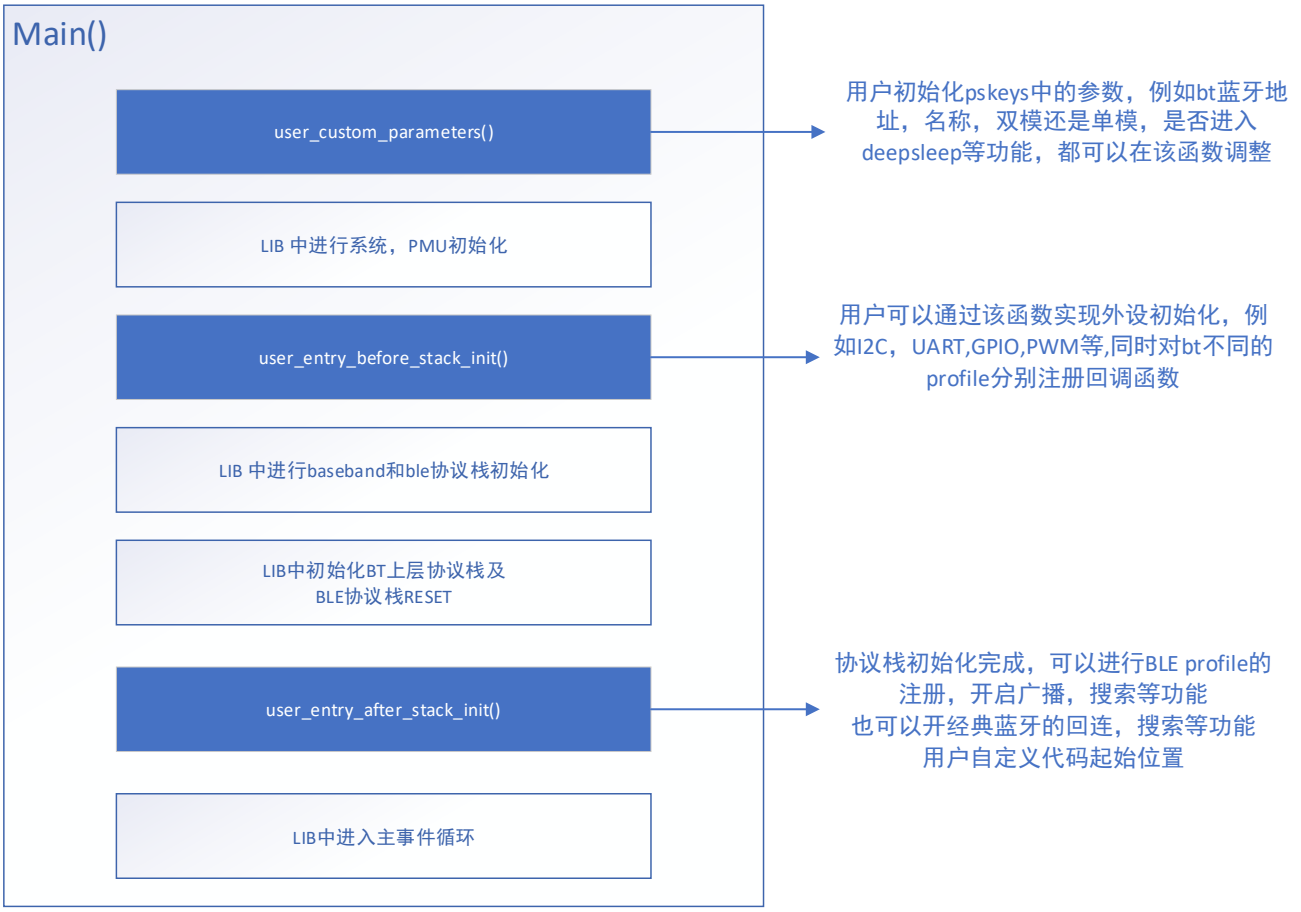


图 1.4 预留函数入口说明

1.4.1 user_custom_parameters 函数

此入口为上电后用户最早进入的函数。参考 1.3.1 节 pskeys 说明，该函数可修改部分 pskeys，修改示例如下：该函数增加了和烧录工具交互的握手次数，增加了 dsp 烧录引导程序，设置了本机蓝牙地址、名称、协议类型，系统选项等。用户可以根据实际需求进行相应的配置，pskeys 一般使用工具中配置的默认参数，不做修改或者在工具端修改。

```
1.  void user_custom_parameters(void)
2.  {
3.      /// 增加和烧录工具的握手次数
4.      retry_handshake();
5.      ///增加 dsp 部分烧录的引导程序
6.      dsp_program();
7.      ///设置 BT 蓝牙地址
8.      pskeys.local_bdaddr.addr[0] = 0x01;
9.      pskeys.local_bdaddr.addr[1] = 0x02;
10.     pskeys.local_bdaddr.addr[2] = 0x03;
11.     pskeys.local_bdaddr.addr[3] = 0x04;
```

```
    pskeys.local_bdaddr.addr[4] = 0x05;
    pskeys.local_bdaddr.addr[5] = 0x06;
    ///设置 BT 蓝牙名称
    memcpy(pskeys.localname,"FR5080_BT",strlen("FR5080_BT")+1);
    ///配置为双模模式
    pskeys.stack_mode = STACK_MODE_BTDM;
    ///使能 BT a2dp,hfp 协议，注意，如果 a2dp 协议使能，则默认同时使能 avrcp 协议
    pskeys.enable_profiles = ENABLE_PROFILE_HF|ENABLE_PROFILE_A2DP;
    ///配置系统选项，使能四线 qspi，使能 cache，使能外部唤醒，使能睡眠
    pskeys.system_options = SYSTEM_OPTION_ENABLE_QSPI_QMODE|SYSTEM_OPTION_ENABLE_CACHE
                            |SYSTEM_OPTION_EXT_WAKEUP_ENABLE|SYSTEM_OPTION_SLEEP_ENABLE;

}
```

1.4.2 user_entry_before_stack_init 函数

该函数入口可以添加对应 bt profile 的回调函数，示例如下：

```
void user_entry_before_stack_init(void)
{
    /* 初始化串口 */
    app_at_init_app();

    LOG_INFO("user_entry_before_stack_init\r\n");

    /* 注册 BT 协议回调函数 */
    bt_me_set_cb_func(bt_me_evt_func);
    bt_hf_set_cb_func(bt_hf_evt_func);
    bt_a2dp_set_cb_func(bt_a2dp_evt_func);
    bt_avrcp_set_cb_func(bt_avrcp_evt_func);
    bt_spp_set_cb_func(bt_spp_evt_func);

    /*内部参考电压校准*/
    triming_init();

    /*设置 io 电压到 3.3V*/
    pmu_set_ioldo_voltage(PMU_ALDO_VOL_3_3);
}
```

该函数初始化串口，校准内部参考电压，设置 io 电压，同时实现了注册 BT 上层 profile HFP，A2DP，AVRCP 及 BT 管理实体 ME 的回调函数。注意若使能 BT 的相关 profile，则必须注册相对应的回调函数，否则连接不上。

1.4.3 user_entry_after_stack_init 函数

user_entry_after_stack_init 为 btdm 协议栈初始化完成后，用户进行自定义行为的入口，可以进行 bond manager 的初始化，GAP 事件处理回调函数的注册，BLE 广播参数的设置，GATT service 的创建，用户 task 的创建，经典蓝牙连接，外设初始化，各个功能模块的使能等。具体使用流程参考 btdm_audio_demo 工程。

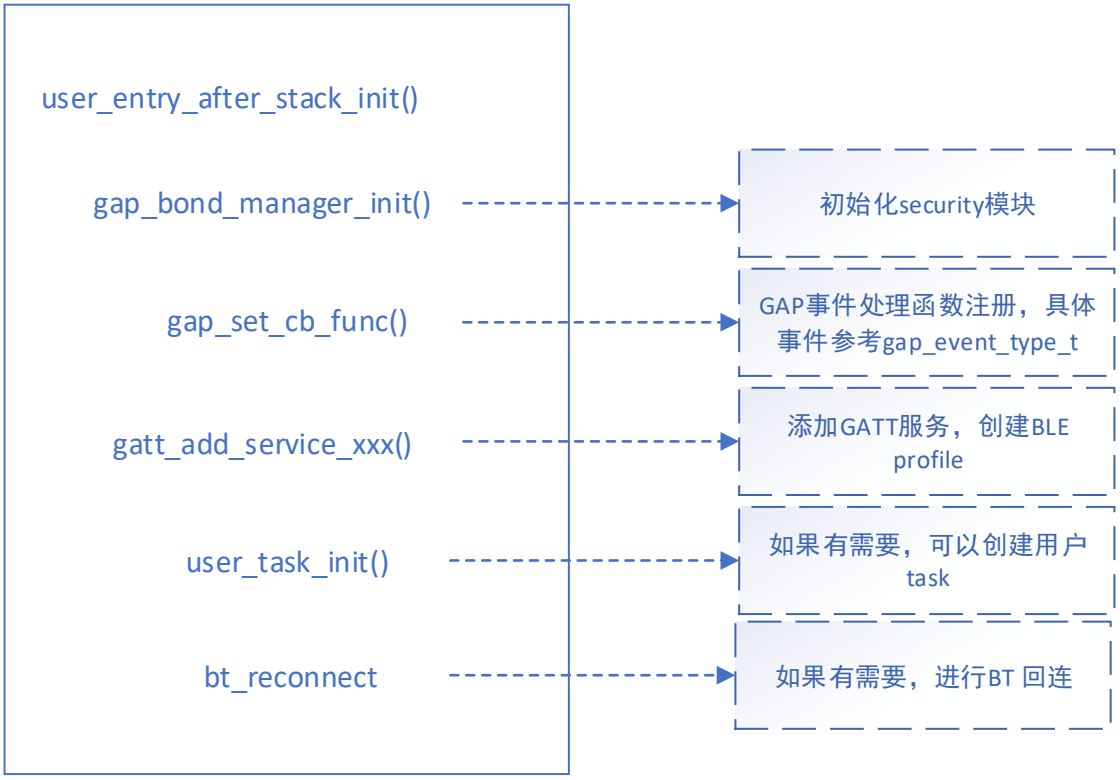


图 1.5 user_entry_after_stack_init() 函数

1.5 SDK 项目工程

SDK 以源码形式提供了多个项目工程作为参考，用户可以在这些工程上进行自己的应用开发。这些工程采用了同样的目录结构，如下图所示：

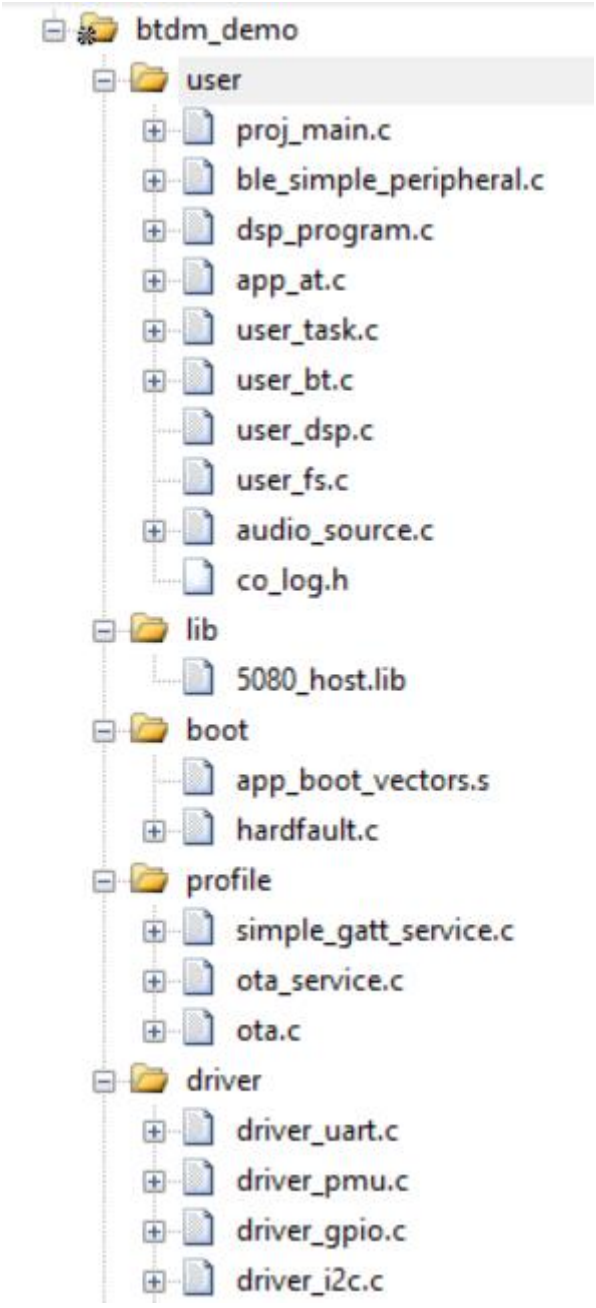


图 1.6 工程目录结构

其中 user 用于存放用户应用层的代码，自定义的 profile 等；driver 中为外设驱动；boot 中为异常向量入口和部分异常的处理函数；profile 为 sdk 提供的一些常用 profile；lib 中为封装好的库文件，其中所提供的接口在 gatt_api.h、gap_api.h，bt_api.h 等文件中。

在当前的 SDK 中提供了以下几种 sample 工程：

- Btdm 例程：参考 SDK 目录.\examples\none_evm\btdm_audio_demo
- Driver 例程：参考 SDK 目录.\examples\none_evm\btdm_drivers_demo

1.6 芯片烧录

目前芯片烧录都是基于串口的，主要有以下几个工具：

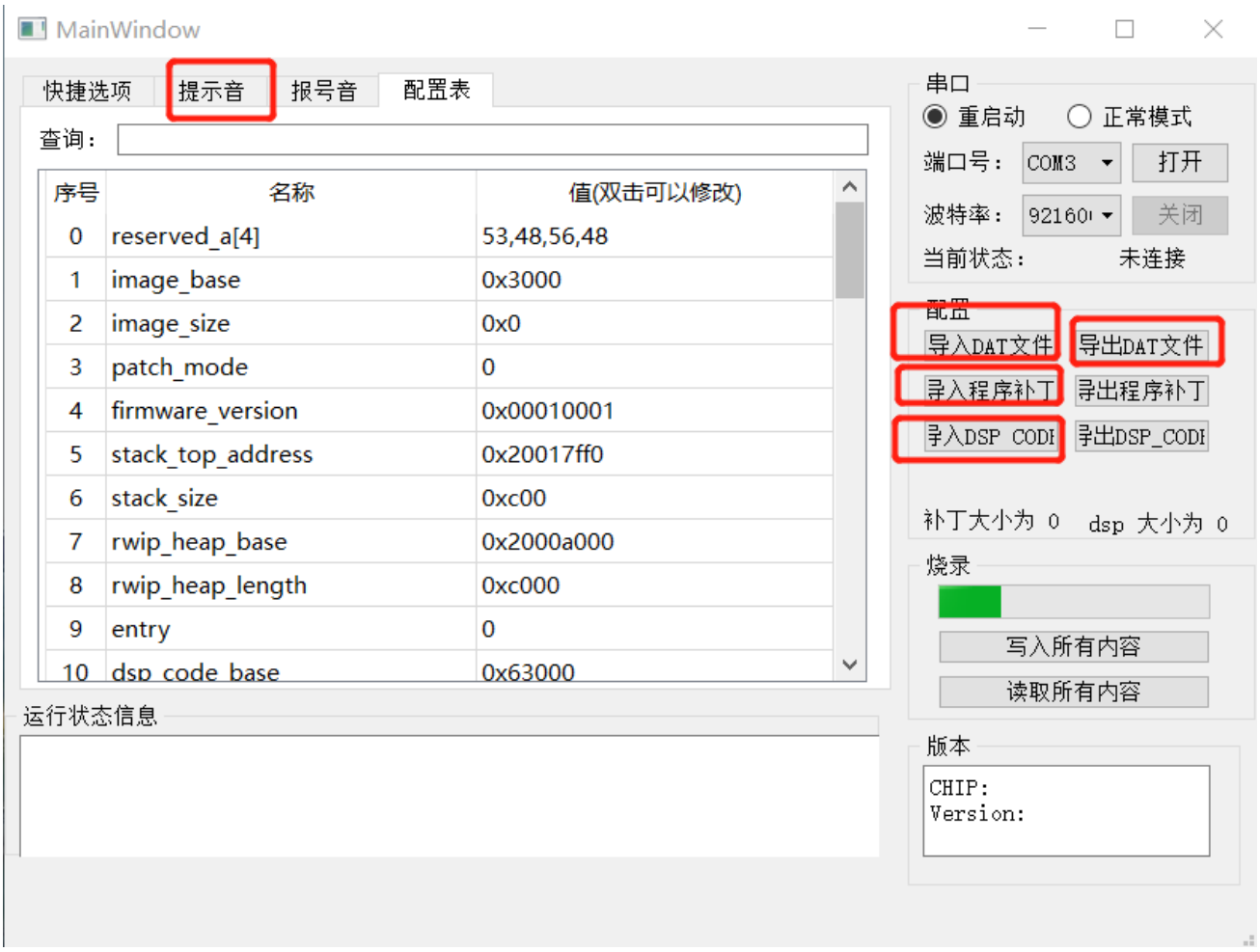
- PC 烧录工具 --- 分为 MCU 和 DSP 端两个工具，烧录单个芯片 MCU 端固件或者 DSP 端固件
- 量产 PC 烧录工具 --- PC 端接多个串口，最多可以同时烧录二十个芯片，并且支持同时烧录 MCU 端和 DSP 端固件
- 量产脱机烧录 --- 专用的脱机烧录版，支持同时烧录 MCU 端和 DSP 端固件

1.6.1 PC 烧录工具

在芯片上电时，内部 boot 程序会尝试通过串口与外部工具进行通信，在握手成功之后就可以进行烧录等后续操作。PC 烧录工具分为 MCU 烧录工具和 DSP 烧录工具。

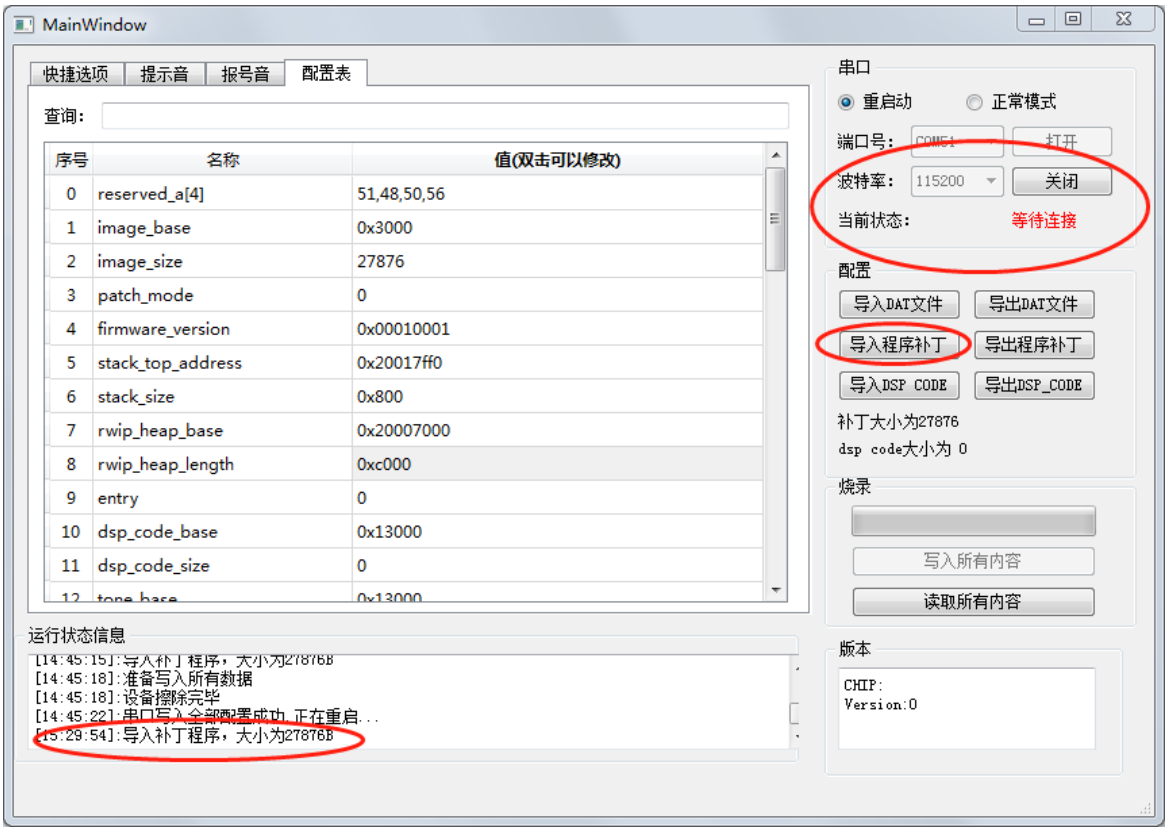
1.6.1.1 MCU 烧录工具

该烧录工具支持烧录 pskeys 配置信息，mcu 端程序，dsp 端程序和提示音。“**导入 dat 文件**“和”**导出 dat 文件**“是指将 pskeys，提示音，mcu 代码和 dsp 代码合并到同一个二进制文件进行导入导出。“**导入程序补丁**“对应的是 fr5080_sdk 用户示例中 output 文件夹下生成的 5080_basic_function.bin。”**导入 DSP CODE**“对应的是 DSP sdk 中用脚本生成的 bin 文件，若 DSP 运行的是 XIP 方式，此处导入 DSP 的启动代码 fr5080_basic_xip.bin；若 DSP 运行的是 RAM 方式，此处导入由脚本 generate_dsp_bin.py 合并生成的 fr5080_dsp_total.bin。”**提示音**“选项栏里可以导入 16 组提示音。“**配置表**”里可以更改蓝牙地址，名称等一些基础配置，有些变量不可变，请参考 1.3 节进行更改。

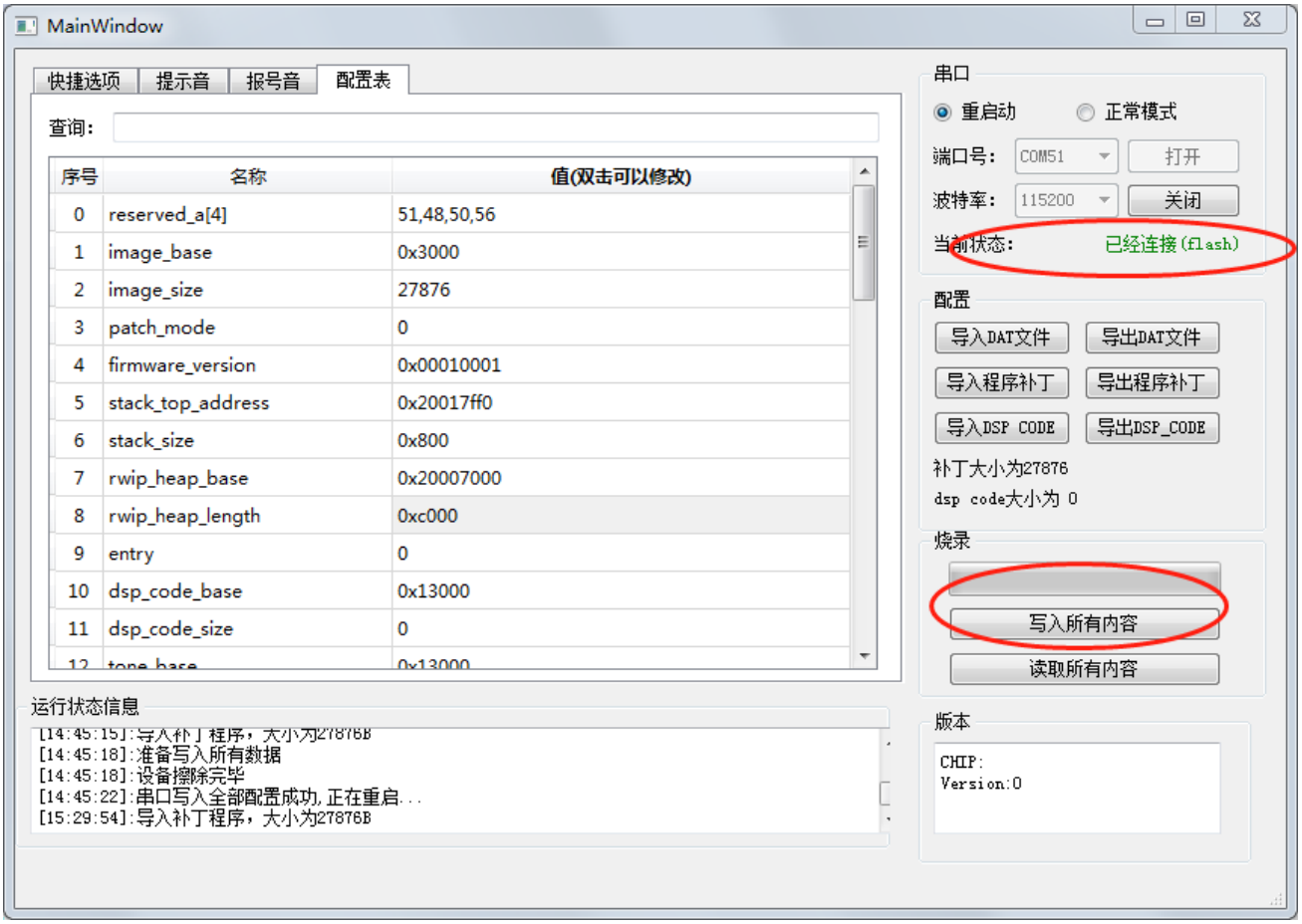


具体操作过程如下：

1. 打开 PC 端串口烧录工具，选择正确的串口号，导入 DAT 文件（或者 mcu 端程序，dsp 端程序等），然后打开串口，进入等待连接状态。



2. 将串口工具的 TX 连接到芯片 PB6（芯片端的 RX），RX 连接到芯片的 PB7（芯片端的 TX），GND 接到板子上任意地线。
3. 将芯片上电或者复位，芯片在与 PC 工具握手成功后，在工具端会显示已经连接，然后点击写入所有内容即可将程序烧录到芯片中



1.6.1.2 DSP 烧录工具

DSP 烧录工具用来烧写外挂 FLASH，适用于 XIP 方式。具体操作过程如下：

1.导入 DAT 文件，对应的是 DSP 部分脚本生成的用户代码，demo 中对应的是 fr5080_user_code_xip.bin，然后点击打开，等待芯片上电握手。

2. 将串口工具的 TX 连接到芯片 PB6（芯片端的 RX），RX 连接到芯片的 PB7（芯片端的 TX），GND 接到板子上任意地线，然后给板子上电或者复位，当前状态变为“**已连接上**“后，点击”**写入所有内容**“。若 dsp 用户代码较大，烧录的时间会比较长，请耐心等待。

3.写入成功后，当前状态会切换为“等待连接“，烧录状态显示为”写入成功“。

1.6.2 量产 PC 烧录工具

该工具可以最多接入 20 个串口，并且可以同时烧录，加大生产效率。具体操作过程如下：

1. “选择 M3 文件”对应的是 MCU 烧录工具导出的 dat 文件，“选择 DSP 文件”对应的是 DSP 端的用户程序（仅针对 XIP 方式）。当工具打开后，串口信息有变动，可以通过“**更新串口信息**”获取当前连接的串口。


Freqchip FR5080 Download Tool V2021.0709.14
烧录数据
选择M3文件
选择DSP文件
校验值： 0000
烧录数量:
烧录状态
打开所有串口
关闭所有串口
更新串口信息
COM3
COM10
1:打开
2:打开
3:打开
4:打开
5:打开
6:打开
7:打开
8:打开
9:打开
10:打开
11:打开
12:打开
13:打开
14:打开
15:打开
16:打开
17:打开
18:打开
19:打开
20:打开
关闭

2. 分别导入 M3 文件和 DSP 文件，此时”**打开**”按键会变成可点击状态，点击“**打开**“串口后，将串口工具的 TX 连接到芯片 PB6（芯片端的 RX），RX 连接到芯片的 PB7（芯片端的 TX），GND 接到板子上任意地线，之后等待芯片上电。

3. 芯片上电或者复位后，若工具握手成功，会自动烧录，烧录成功后，提示绿色“**下载成功**”字样，若烧录失败，则会提示红色“**下载失败**”。

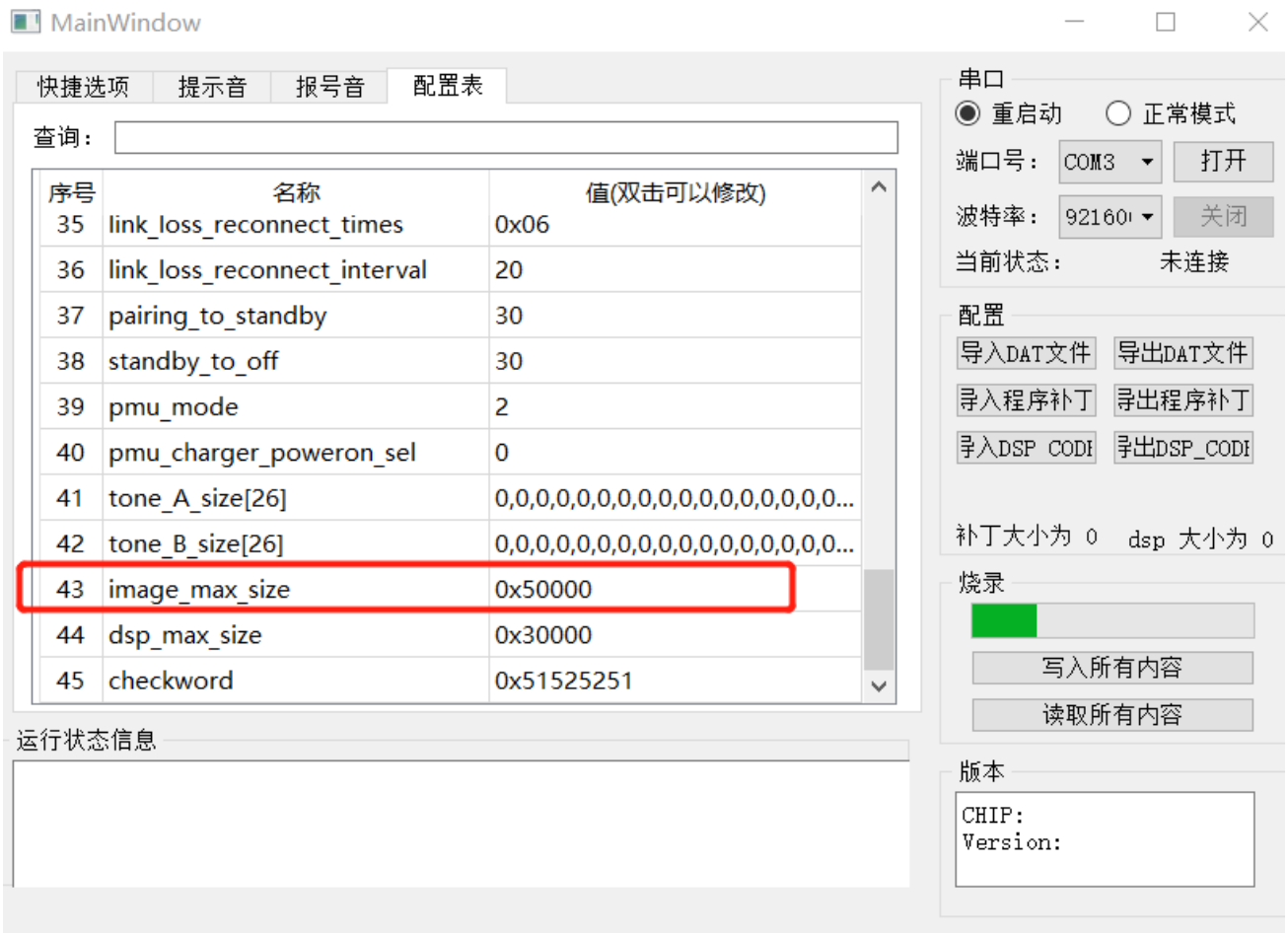
1.6.3 量产脱机烧录

FR5080 系列芯片有专用的量产脱机烧录版，支持脱机烧录，无需连接电脑，具体实施方式可以联系代理商。

1.6.4 常见烧录问题说明

1.6.4.1 导入程序补丁出现提示“导入数据有误”

需要修改 image_max_size 的大小，image_max_size 需大于实际导入程序补丁的长度，修改 image_max_size 后，dsp_code_base 和 tone_base 也需要对应的修改，若没有提示音，则不用管 tone_base, dsp_code_base 需设置为 image_base + image_max_size；若有提示音，tone_base 需设置为 image_base + image_max_size，dsp_code_base 需设置为 tone_base + tone_size（根据导入的 tone 预估，同时需要 4k 对齐）。



1.6.4.2 导入 DSP CODE 提示“导入数据有误”

出现该问题，需要去修改 dsp_max_size 大小，dsp_max_size 需要大于实际 dsp code 的大小。

1.6.4.3 芯片上电后，没有握手成功，不显示“已连接上状态”

一般是由于串口上电抖动引起，可以尝试先断开串口线，重新接好后，再次上电。若还是不行，需要检查串口是否接对，可以调换下 TX，RX 线，再次上电测试。

2. 低功耗管理

2.1 睡眠

FR5080 系列芯片在 MCU 正常工作模式，3.3V 供电情况下，工作电流在 2~5mA。为了节省电能，可以进入睡眠模式。睡眠模式下，保持 ble 或 bt 连接状态，系统的电流大致为几十 uA，可以被 GPIO，RTC 等模块唤醒，且 RAM 中数据保持，这种模式适用于正常连接状态、或者周期性广播状态等。在这种模式下，由于外设和数字部分会掉电，唤醒时 LIB 代码部分会进行现场恢复的操作，包括 MCU 状态，RF 初始化等操作，用户需要对使用到的外设重新初始化，整个流程会在很短的时间完成。

2.2 程序运行流程

主程序的运行流程如下图：

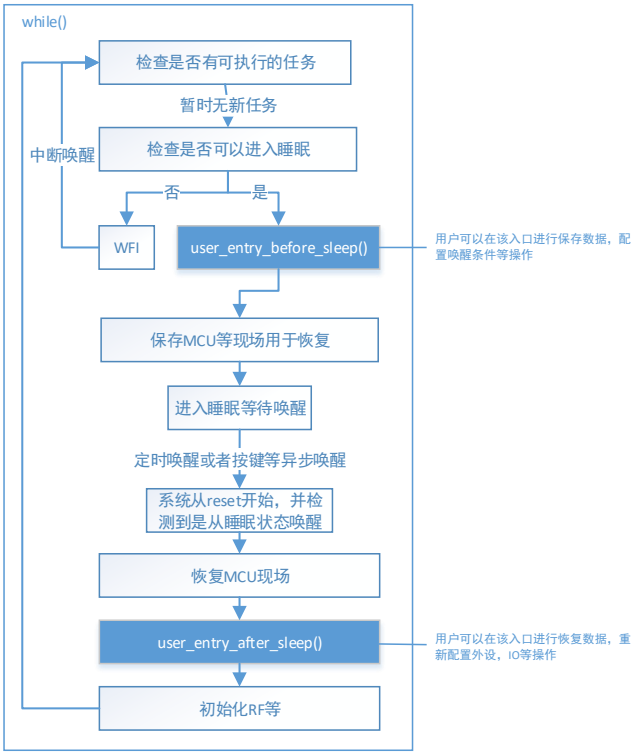


图 2.1 睡眠流程图

在该流程中用户在睡眠前和唤醒后各有一个入口可以进行自定义的操作：

1. user_entry_before_sleep

该函数在进入睡眠前被调用，用户可在里面实现控制 GPIO 的状态保持（针对 GPIO 在系统工作和睡眠状态下的控制参见外设驱动章节），配置睡眠唤醒条件等行为。

2. user_entry_after_sleep

在系统唤醒后，用户可以在该函数中重新进行外设的初始化（进入睡眠后外设的状态因为掉电都会丢失）等操作。

2.3 唤醒条件

睡眠的唤醒有同步和异步两种：同步唤醒来自一个硬件 timer，这个 timer 的设置由协议栈中代码完成，主要取决于 BT pagescan 和 inquiry sacn 间隔、BT sniff 间隔、 BLE 广播间隔、BLE 连接间隔等参数，在应用层代码中无需关注；异步主要来自于 PMU（电源管理单元）的中断信号，PMU 的中断源有：充电器插入拔出、RTC、GPIO 状态监测模块等，这些中断源可以在系统初始化时进行设置。例如：

```
pmu_set_pin_pull(GPIO_PORT_D, (1<<GPIO_BIT_4)|(1<<GPIO_BIT_5), true);
pmu_port_wakeup_func_set(GPIO_PD4|GPIO_PD5);
```

这两行代码可以配置 PMU 中的 GPIO 状态监测模块开始监测 GPIO_PD4 和 GPIO_PD5 的状态，一旦发生电平高低的变化，就可以产生 PMU 中断。如果在睡眠中产生 PMU 中断，则系统会被唤醒，唤醒后可在 PMU 的中断处理函数中进行相应的处理。

3. BT 协议栈

FR5080 SDK 中提供给用户关于 BT 相关协议主要有 ME(management entity), A2DP, AVRCP, HFP, 后续会陆续添加 SPP，HID 等协议。用户可以通过这些协议的 API 及回调函数，发送命令或处理返回事件。

3.1 ME 管理实体协议

ME 层主要负责查询附近设备，创建各种类型的连接和管理本设备被其他设备连接的入口。该部分 API 请参考.\components\btdm\include\bt_api.h。

3.1.1 BT 链接管理介绍

3.1.1.1 BT 接入状态

BT 链路层状态主要有 standby，inquiry，page，inquiry scan，page scan，connection 状态。

- Standby --- 设备默认状态，不可被发现，不可被连接，可以进入 inquiry,page,inquiry scan,page scan 状态
- Inquiry --- 扫描附近设备状态
- Page --- 创建链接状态，通常发起连接的设备为主设备，发起连接需要知道被连接设备的地址
- Inquriy scan --- 与 inquiry 状态相对应，处于可发现状态，此时设备才可以被其他设备扫描到
- Page scan --- 与 page 状态相对应，处于可连接状态，此时设备可以被其他设备连接
- Connection --- 已连接状态，主从设备时钟已同步，此状态上可以发送数据包

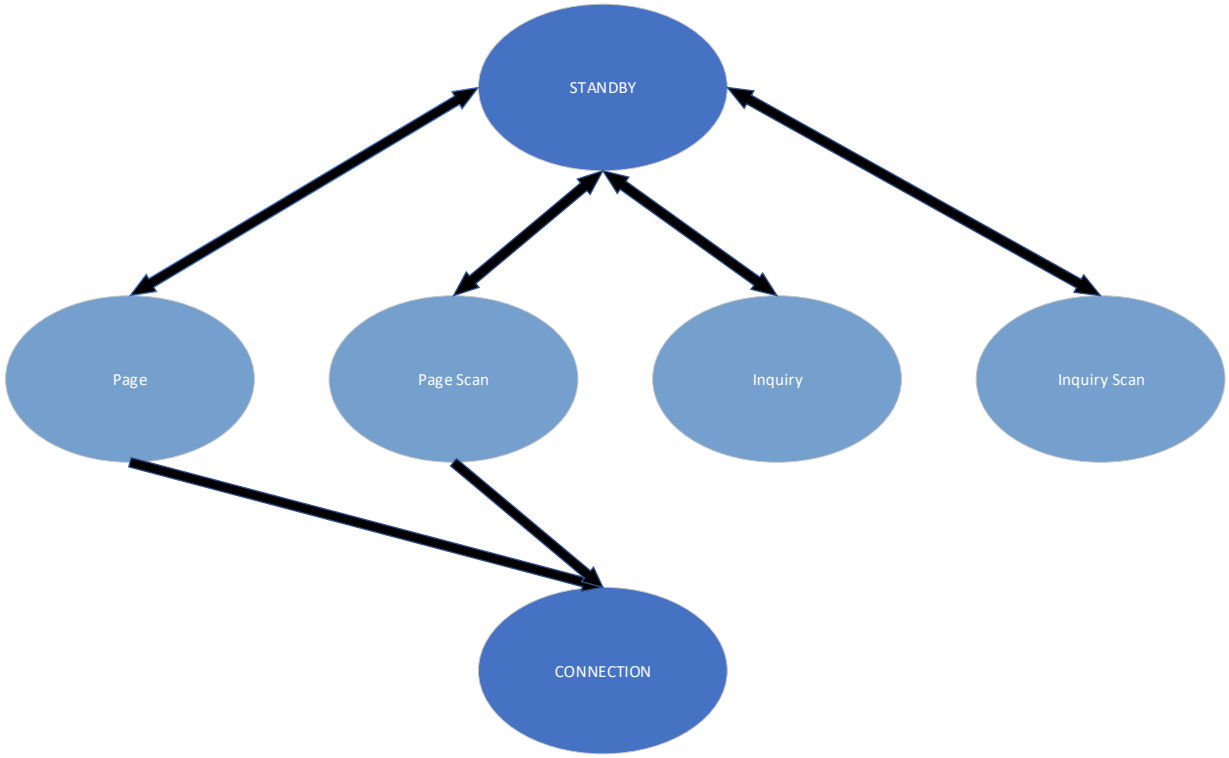


图 3.1 链路层连接状态机

3.1.1.2 BT 连接模式

当设备处于 Connection 状态时，有两种连接模式：Active 模式和 Sniff 模式。

- Active 模式 --- 此模式下，从设备在每个接收时隙都会开窗等待主设备发送数据，主设备有数据发送时，在发送时隙立即发送，没有数据发送时，按查询间隔发送查询包（POLL），主要用来保持连接和查询从设备是否有数据要发送。此模式系统无法进入睡眠模式
- Sniff 模式 --- 此模式下，从设备不需要每个接收时隙都开窗接收数据，而是按照交互好的睡眠间隔（sniff interval）才开窗接收数据，如图 3.2，在每个 sniff anchor point 才开始开窗接收数据，其他时隙系统可以进入睡眠。

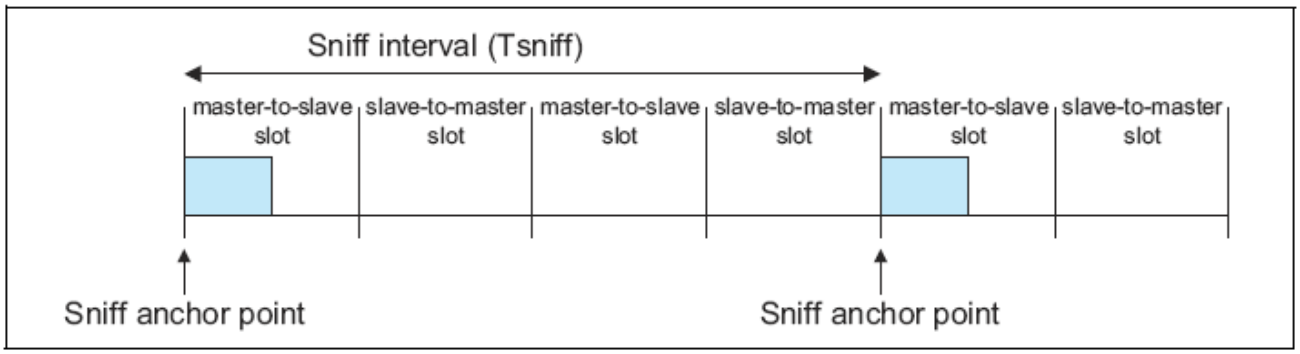


图 3.2 sniff 模式

3.1.1.3 BT 连接角色

参考章节 3.1.1.1, BT 设备分为主设备（master）和从设备（slave），一般首先发起 page 的设备为主设备，从 page scan 状态跳转到连接状态的设备为从设备。主从设备可以进行切换。

3.1.2 ME 基础流程

ME 层的基础流程如图 3.3 所示：
蓝色部分为 APP 和 LIB 间交互，橙色部分为内部交互，对用户不可见；实线部分是必须要配置和实现的，虚线部分由具体 APP 决定。利用 5080 SDK 编写 APP 代码时，若使能了 BT 协议栈，需要在 user_entry_before_btdm_init 函数中注册 ME 回调函数， 回调函数的实现可参考例程 btdm_audio_demo 中的 bt_me_evt_func 函数。注册完成后，会返回 BTEVENT_HCI_INITIALIZED 事件，表明底层 Controller 部分已初始化完毕，此后，可以发送其他有关 ME 的 API 命令，例如 bt_start_inquiry, bt_set_accessible_mode_nc 等。

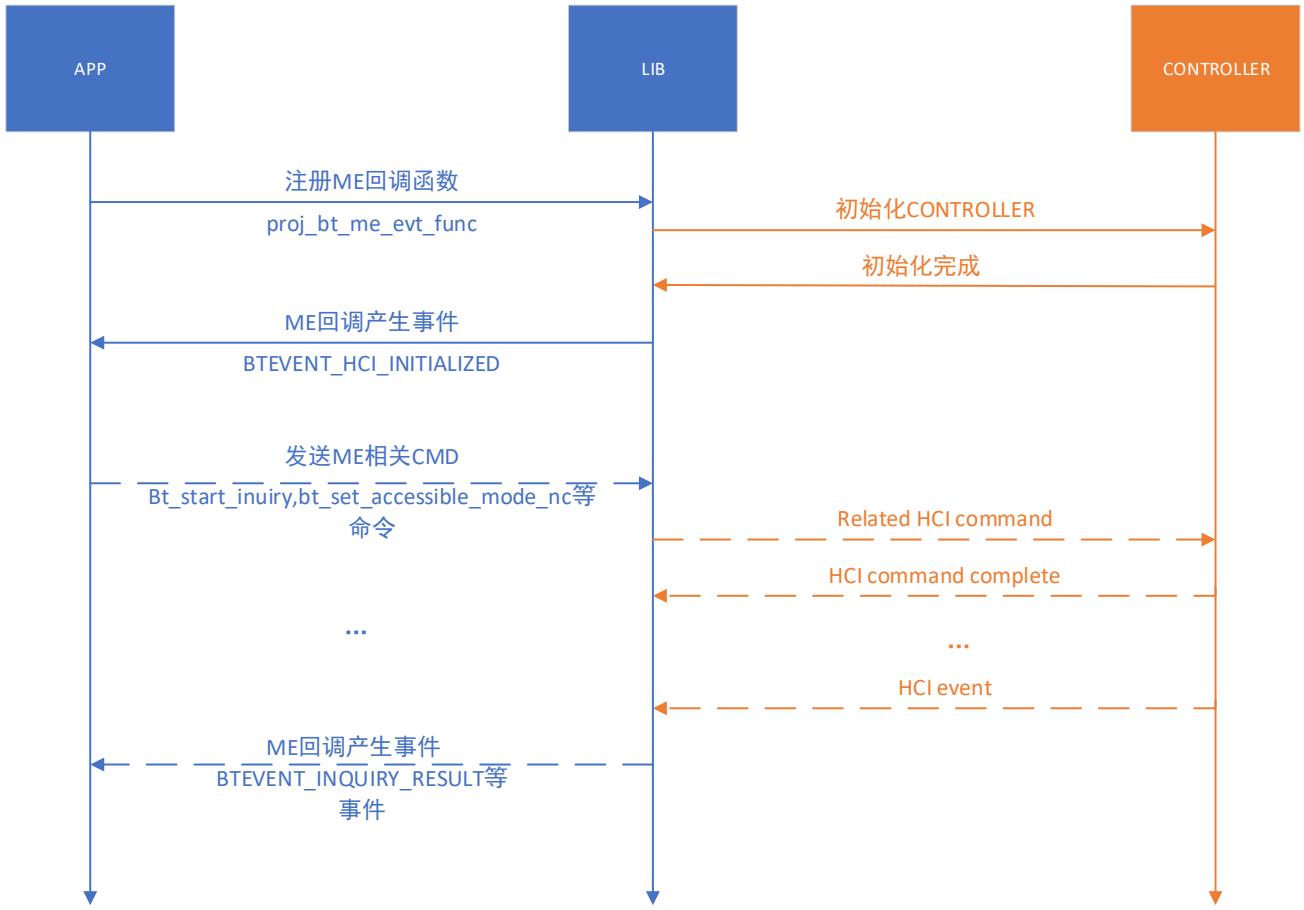


图 3.3 ME 基础流程

3.1.2.1 BT 扫描

如图 3.4 所示，当收到 BTEVENT_HCI_INITIALIZED 事件后，可发送 bt_start_inquiry，具体函数说明参考 bt_api.h。如果扫描到附近设备则返回 BTEVENT_INQUIRY_RESULT 事件，若扫描超时或者已扫描到足够多设备（阈值由 bt_start_inquiry 的参数决定），则返回 BTEVENT_INQUIRY_COMPLETE 事件。若想停止扫描，则可以发送 bt_stop_inquiry，停止后会返回 BTEVENT_INQUIRY_CANCELED 事件。

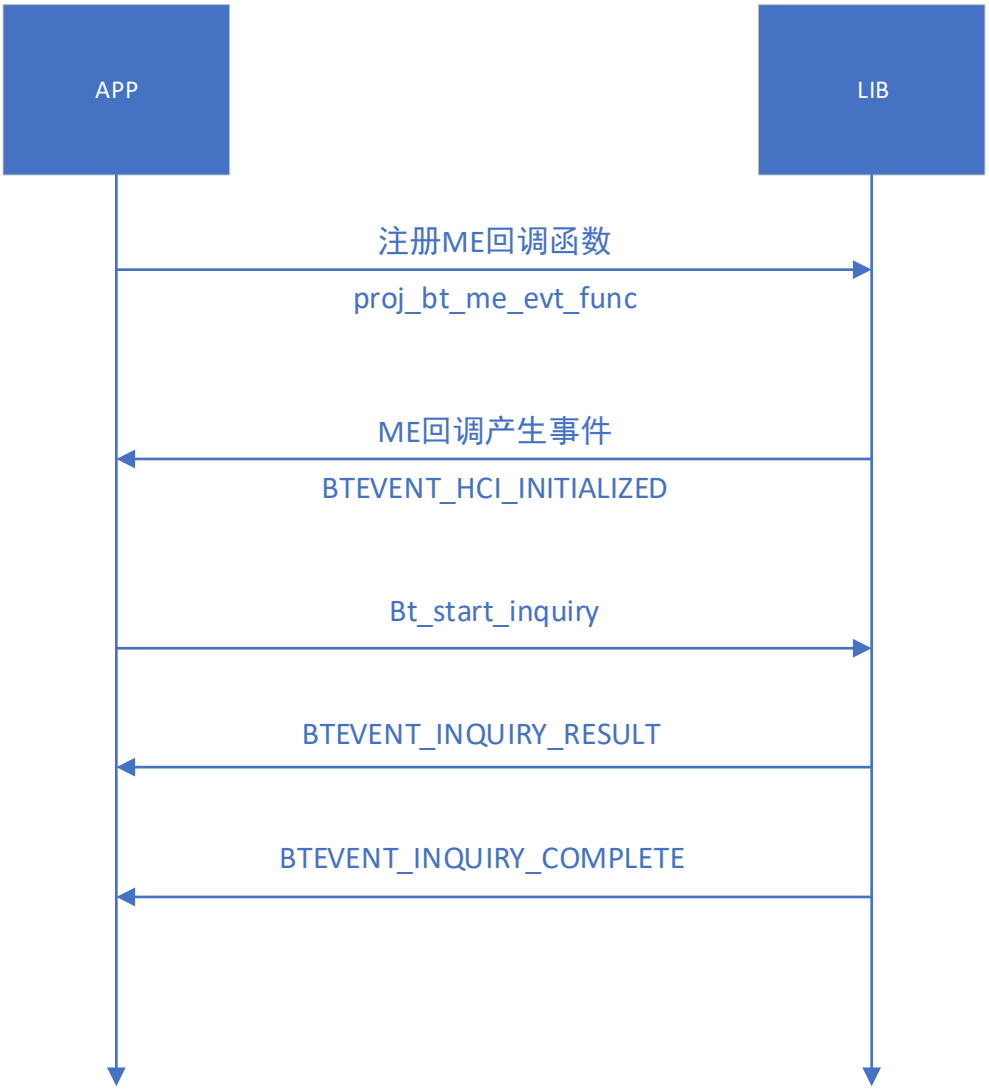


图 3.4 bt inquiry

3.1.2.2 BT 设置接入模式

如图 3.5 所示，当收到 BTEVENT_HCI_INITIALIZED 事件后，可发送 bt_set_accessible_mode_nc，bt_set_accessible_mode_c，将当前 bt 设置为是否可被发现，是否可被连接中的一组。具体函数说明及参数配置参考 bt_api.h。若 bt 接入模式发生了改变，则会产生 BTEVENT_ACCESSIBLE_CHANGE 事件。

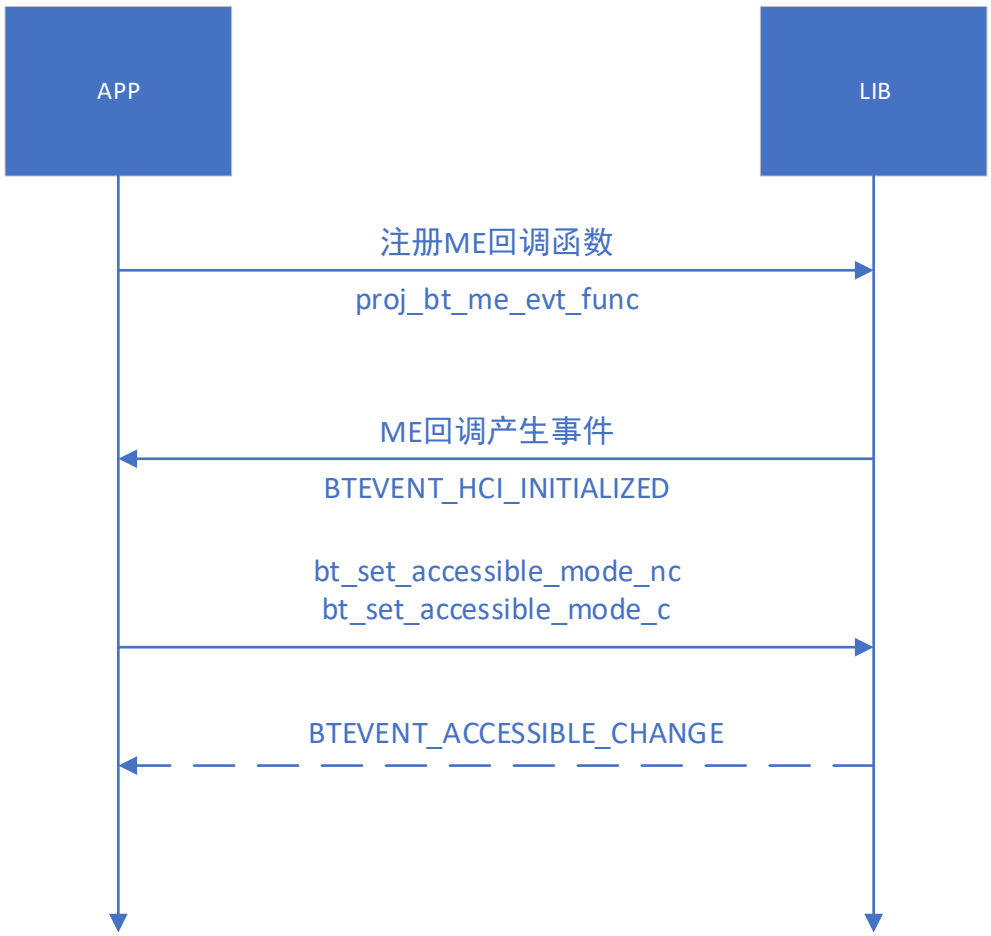


图 3.5 bt set accessible mode

3.1.2.3 BT sniff

如图 3.6 所示，当连接建立完成后（如果是 APP 端发起连接则会收到 BTEVENT_LINK_CONNECT_CNF 事件，如果是远端发起连接，则会收到 BTEVENT_LINK_CONNECT_IND 事件），APP 端可以发送 bt_start_sniff 进入 sniff 模式，若已处于 sniff 模式，则可以发送 bt_stop_sniff 退出 sniff 模式。如果连接模式有改变，则会返回 BTEVENT_MODE_CHANGE 事件。

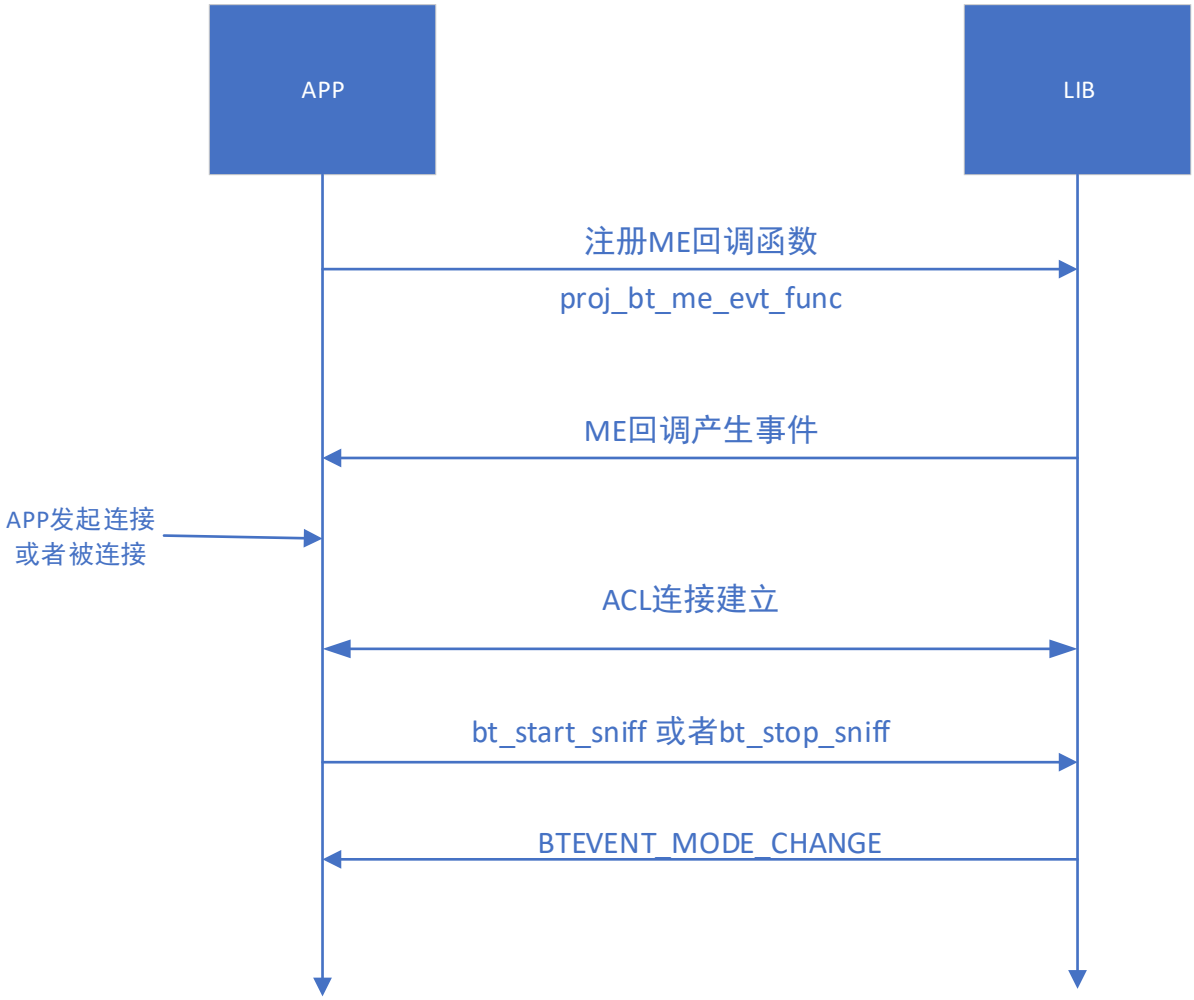


图 3.6 bt sniff

3.1.2.4 BT 主从切换

如图 3.7 所示，当连接建立完成后（如果是 APP 端发起连接则会收到 BTEVENT_LINK_CONNECT_CNF 事件，如果是远端发起连接，则会收到 BTEVENT_LINK_CONNECT_IND 事件），APP 端可以发送 bt_switch_role 切换主从设备。如果设备主从模式有改变，则会返回 BTEVENT_ROLE_CHANGE 事件。

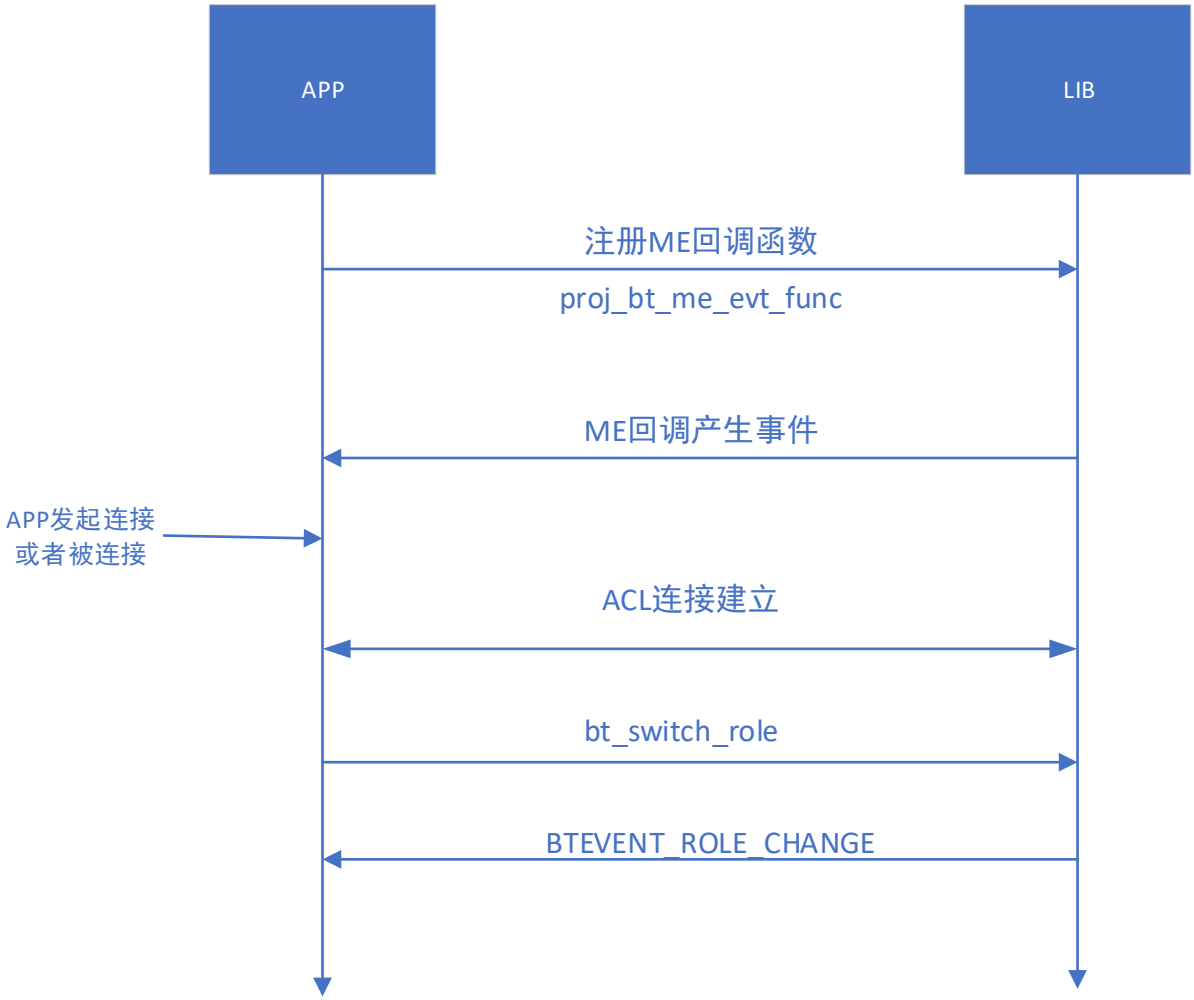


图 3.7 BT switch role

3.1.3 ME 回调函数示例

```
1.  void bt_me_evt_func(me_event_t *event)
2.  {
3.      uint8_t i;
4.      uint8_t device_index;
5.      //printf("me func = %d\r\n",event->type);
6.      switch(event->type){
7.          case BTEVENT_HCI_INITIALIZED:
8.              //蓝牙已连接情况下，将蓝牙设置成不可发现，不可连接状态
9.              bt_set_accessible_mode_c(BAM_NOT_ACCESSIBLE,NULL);
10.             //蓝牙未连接情况下，将蓝牙设置成可发现，可连接状态
11.             bt_set_accessible_mode_nc(BAM_GENERAL_ACCESSIBLE,NULL);
12.             break;
13.         case BTEVENT_INQUIRY_CANCELED:
14.
```

```
            break;
        case BTEVENT_INQUIRY_COMPLETE:
            //搜索完成后，重新搜索附近设备，搜索时间 10*1.28s，最大搜索结果限制为 10 个设备
            bt_start_inquiry(10,10);
            break;
        case BTEVENT_INQUIRY_RESULT:
            //搜索结果打印出来，此处打印的是设备地址，其他参数可参考结构体 BtInquiryResult
            printf("me bt inquiry result\r\n");
            for(i=0;i<6;i++)
            {
                printf("0x%02x ",event->param.inqResult->bdAddr.A[i]);
            }
            printf("\r\n");
            break;
        case BTEVENT_LINK_CONNECT_IND:
            ///收到连接请求，将连接信息存储到本地
            device_index = bt_find_free_dev_index();
            //查找本地空闲设备，若没有找到，则返回错误
            if(device_index == NUM_DEVICES){
                 printf("shall not be here,me_callback\r\n");
            }
            //存储本地 device 已使用标志位
            user_bt_env->dev[device_index].inUse = 1;
            //存储远端设备地址到本地
            memcpy(&user_bt_env->dev[device_index].bd,event->addr,6);
            //存储远端设备信息指针，可用来作为 bt_start_sniff,bt_switch_role 参数
            user_bt_env->dev[device_index].remDev = event->remDev;
            //将本地设备连接状态置为连接状态
            user_bt_env->dev[device_index].state = BT_CONNECTED;
              break;
        case BTEVENT_LINK_CONNECT_CNF:
            ///本地发起连接收到的确认信息，根据 event->errcode 判断是否连接成功或者失败原因
            device_index = bt_find_device(event->addr);
            //根据地址，查找对应的本地设备
            if((device_index == 0)&&(event->errCode == BEC_PAGE_TIMEOUT)){
            //若是失败原因是 page_timeout,可以再次发起连接
                bt_reconnect((enum bt_reconnect_type_t)user_bt_env->dev[device_index].
                                    reconnecting,event->addr);
            }else if(event->errCode == BEC_NO_ERROR) {
                //若连接成功，将连接信息存到本地
                user_bt_env->dev[device_index].state = BT_CONNECTED;
                user_bt_env->dev[device_index].remDev = event->remDev;
                user_bt_env->dev[device_index].reconnecting = 0;
```

```
58.                 //连接上后，需要设置断开后蓝牙的接入状态，因为本 demo 在连接时，将蓝牙接入状态设置
59.                 //为不可被发现，不可被连接
60.                   bt_set_accessible_mode_nc(BAM_GENERAL_ACCESSIBLE,NULL);
61.             }
62.             break;
63.         case BTEVENT_LINK_DISCONNECT:
64.             ///连接断开事件，将本地信息清零
65.             device_index = bt_find_device(event->addr);
66.             memset(&(user_bt_env->dev[device_index]),0,sizeof(APP_DEVICE));
67.             break;
68.         case BTEVENT_ACCESSIBLE_CHANGE:
69.             ///蓝牙接入状态改变，本 demo 将接入状态分为三种，配对，待机，空闲
70.             if(event->param.aMode == BAM_GENERAL_ACCESSIBLE){
71.                 ///配对状态，可被连接，可被发现状态
72.                 user_bt_env->access_state = ACCESS_PAIRING;
73.                 printf("\r\nII\r\n");
74.             }else if(event->param.aMode == BAM_CONNECTABLE_ONLY){
75.                 ///待机状态，可被连接，不可被发现状态
76.                 user_bt_env->access_state = ACCESS_STANDBY;
77.                 printf("\r\nIJ\r\n");
78.             }else if(event->param.aMode == BAM_NOT_ACCESSIBLE){
79.                 ///空闲状态，不可被连接，不可被发现状态
80.                 user_bt_env->access_state = ACCESS_IDLE;
81.                 printf("\r\nIK\r\n");
82.             }
83.             break;
84.         case BTEVENT_MODE_CHANGE:
85.             ///连接模式变化，ACTIVE 或者 SNIFF 模式
86.             device_index = bt_find_device(event->addr);
87.             //存储当前模式到本地
88.             user_bt_env->dev[device_index].mode = event->param.modeChange.curMode;
89.             break;
90.         case BTEVENT_ROLE_CHANGE:
91.             break;
92.         default:
93.             break;
94.     }
95. }
```

3.2 HFP 协议

HFP（Hands-free Profile），让蓝牙设备可以控制电话，如接听，挂断，拒接，拨号等功能，同时可以接收各种状态消息。HFP 定义了音频网关（AG）和免提组件（HF）两个角色，一般情况下手机是网关设备 AG，蓝牙耳机为免提设备 HF。具体协议介绍可以参考 https://www.bluetooth.com/specifications/profiles-overview/。

3.2.1 HFP 基础流程

FR5080 SDK 封装了相关 API 命令，控制 HF 发送相关命令给 AG，同时将 AG 返回的命令转换成相关事件，用户可以利用回调函数获取这些事件消息。下续章节将介绍一些基础流程，具体事件的意义及参数说明，请参考 hfp_api.h。

3.2.1.1 HFP 服务层连接建立与释放

如图 3.8 所示，APP 需先注册 hf 的回调函数 proj_bt_hf_evt_func，APP 与 LIB 通过回调函数获取事件消息，橙色部分为 LIB 与远端 AG 的交互，对用户不可见。当 AG 发起连接或者本地发起连接（调用 hf_connect）时，首先会建立 RFCOMM 连接，若是 AG 发起的连接，则 APP 会收到 HF_EVENT_SERVICE_CONNECT_REQ 事件，之后 LIB 会自动发送多条 hfp 命令，APP 会陆续收到 HF_EVENT_GATEWAY_FEATURES，HF_EVENT_CALLSETUP_IND，HF_EVENT_CALL_IND，HF_EVENT_SERVICE_IND，HF_EVENT_BATTERY_IND，HF_EVENT_GW_HOLD_FEATURES 等事件，最后收到 HF_EVENT_SERVICE_CONNECTED 事件，表明 HFP 服务层连接成功。后续所有 HFP 操作都是基于该服务层连接的基础上。当用户或者本地断开连接(hf_disconnect)后，会收到 HF_EVENT_SERVICE_DISCONNECTED 事件。

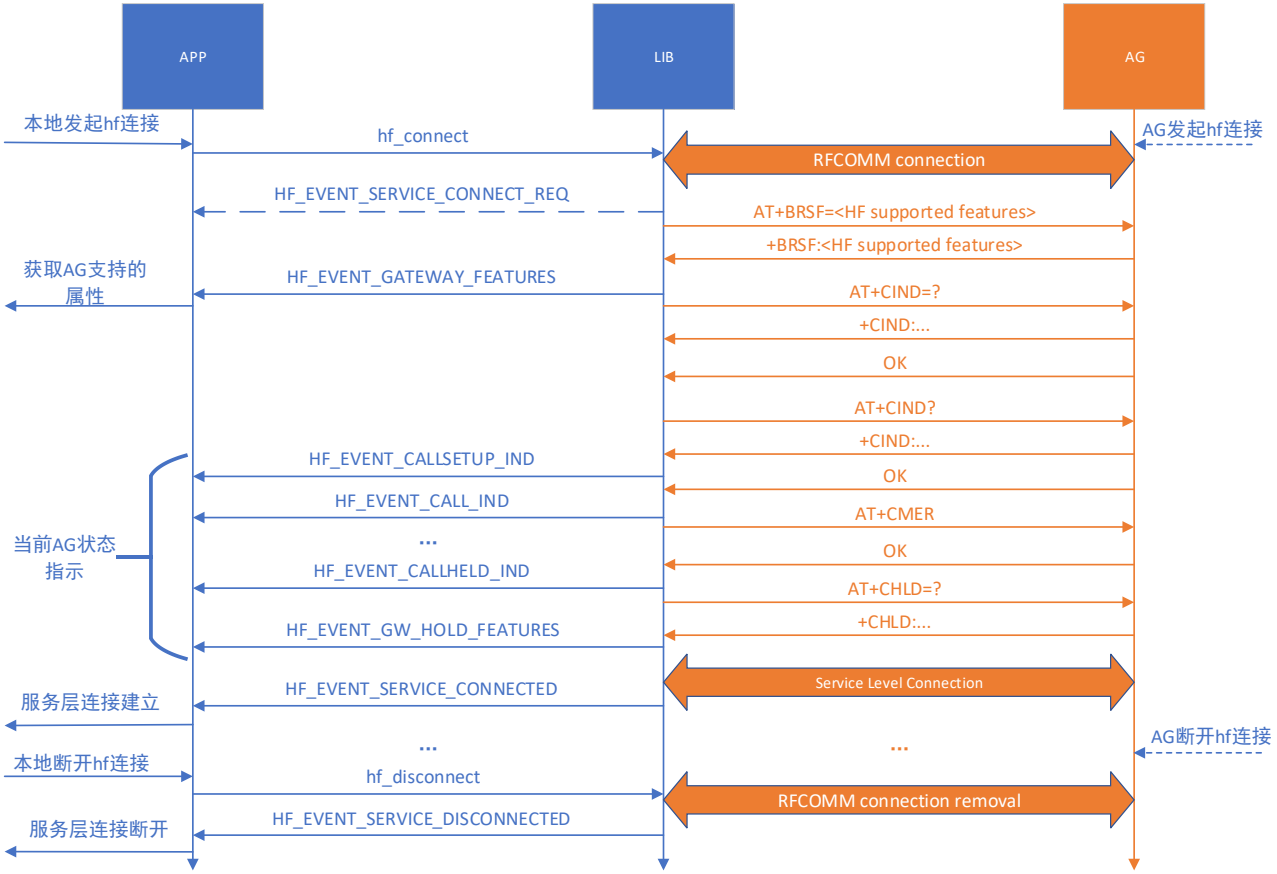


图 3.8 HFP 服务层建立流程与释放

3.2.1.2 HFP 音频连接建立与释放

如图 3.9 所示，当服务层连接已建立后，APP 调用 hf_create_audio_link 或者 AG 发起音频连接，回调函数会收到 HF_EVENT_AUDIO_CONNECTED 事件，此时通话链路已成功建立，语音数据可通过 SCO 链路进行交互。APP 可以调用 hf_disconnect_audio_link 或 AG 断开音频连接，回调函数会收到 HF_EVENT_AUDIO_DISCONNECTED 事件。

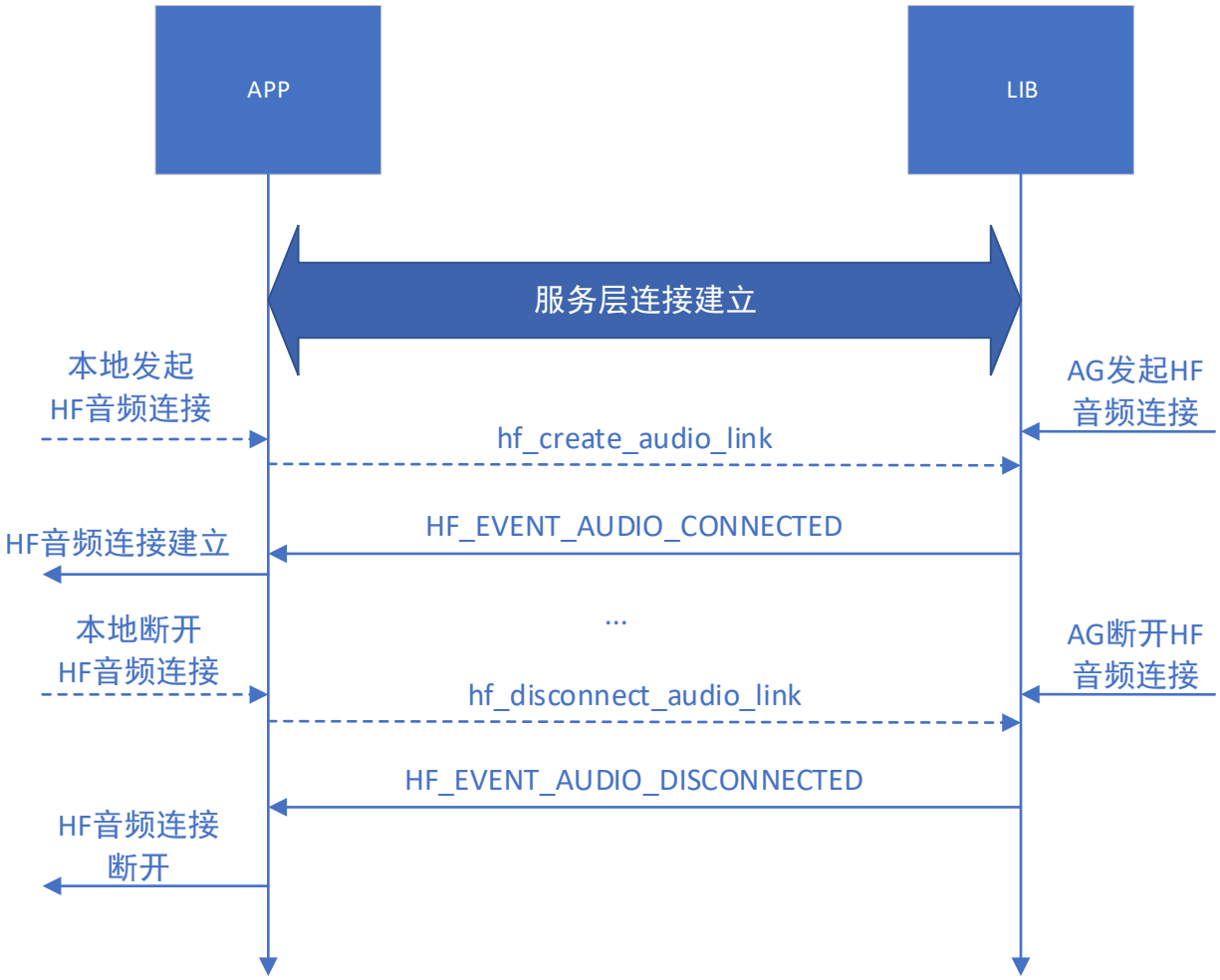


图 3.9 HFP 音频连接的建立与释放

3.2.1.3 HFP 使能相关特征

如图 3.10 所示，在服务层连接已建立后，APP 发送了 hf_enable_caller_id_notify，hf_enable_call_wait_notify，hf_disable_nrec，hf_send_at_command 等命令，若发送成功，都会有 HF_EVENT_COMMAND_COMPLETE 事件返回。图 3.10 使能了来电号码提示，多方通话，关闭手机端回声消除，降噪算法并发送了自定义 HFP 的 AT 命令。章节 3.2.2 中具体描述了如何运用自定义 AT 命令实现 iphone 手机中添加电量显示的功能。

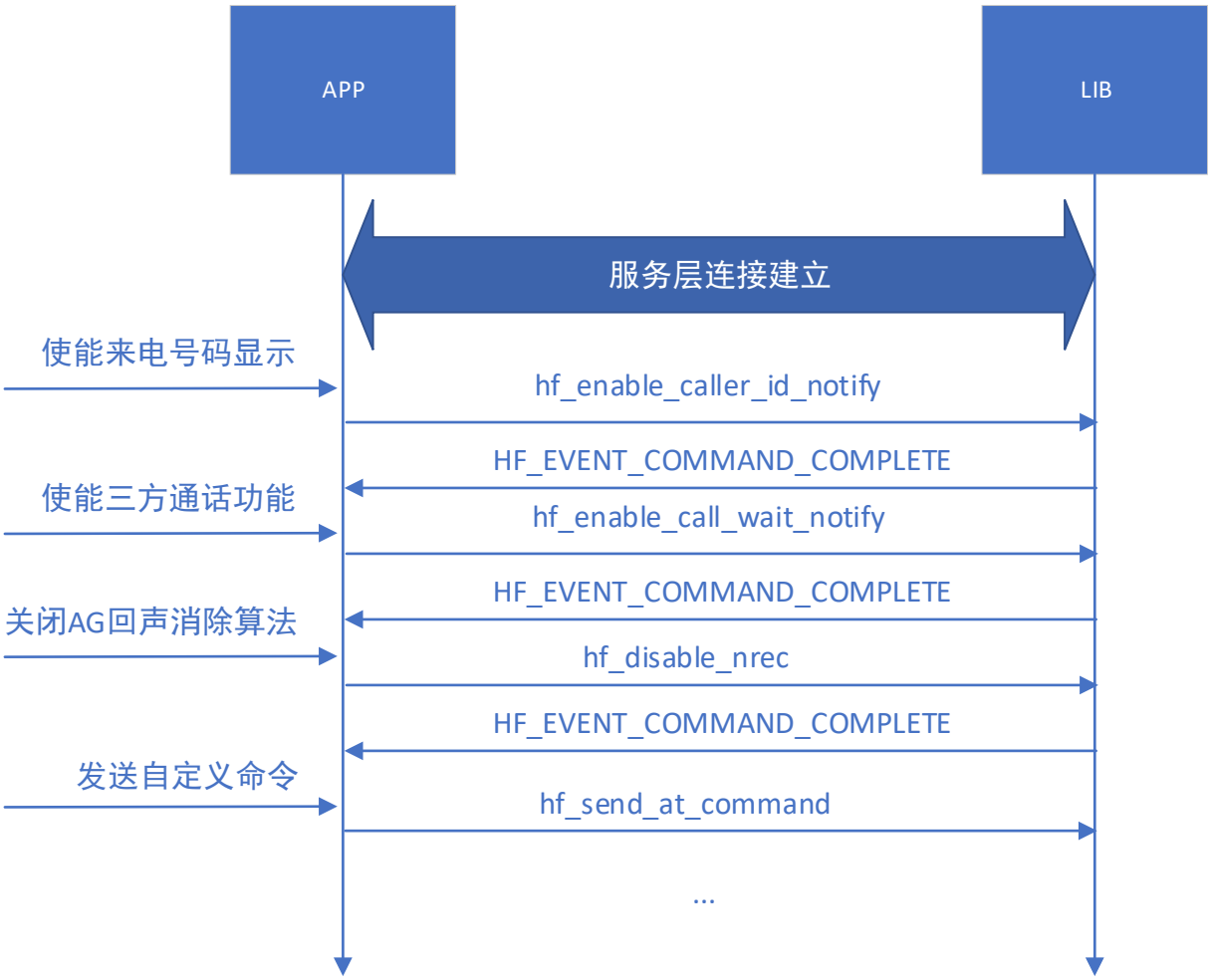


图 3.10 HFP 使能相关功能

3.2.1.4 HFP 接听电话

图 3.11 和图 3.12 描述了 APP 接通电话（是否支持带内铃音的情况下）的示例流程。带内铃声（in-band ring）指的是手机（AG）上的来电铃音传到耳机端（HF），而不只是‘嘟嘟’的提示音。图 3.11 的示例流程是支持带内铃声的，当服务层连接建立好后，有电话拨入，HF_EVENT_CALLSETUP_IND 事件提示 incoming call，同时产生 HF_EVENT_AUDIO_CONNECTED 事件表明音频连接建立，并播放手机来电提示音。之后会循环产生 HF_EVENT_RING_IND 和 HF_EVENT_CALLER_ID_NOTIFY 事件提示有电话拨进，直到 APP 发送 hf_answer_call 接听电话或者手机端接听电话（不是本示例），接通电话后，会返回 HF_EVENT_CALL_IND 和 HF_EVENT_CALLSETUP_IND 事件表明电话已接通。图 3.12 的示例流程是不支持带内铃声的，与图 3.11 相比，不用之处在于 HF_EVENT_AUDIO_CONNECTED 事件在接通电话后产生。

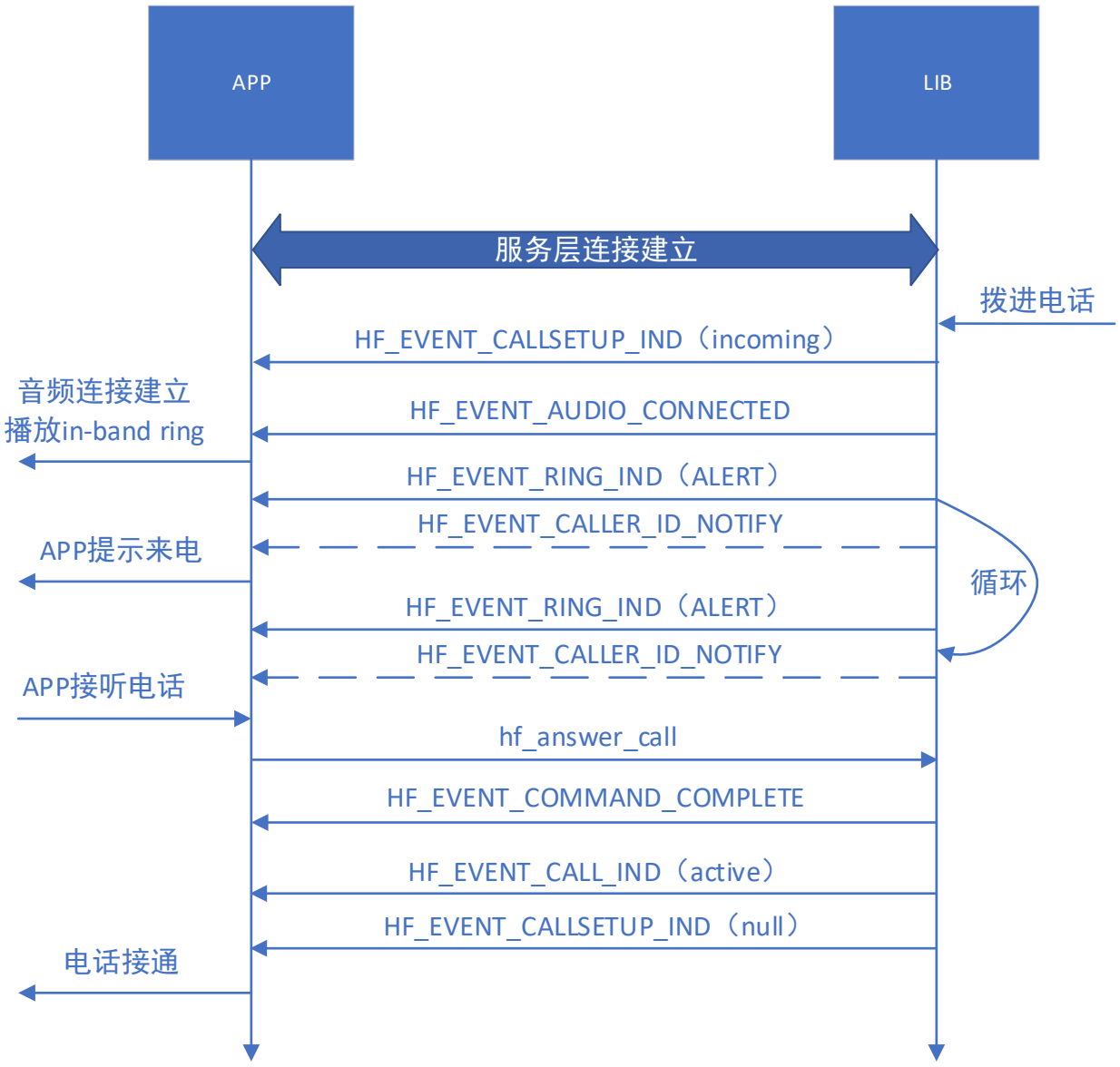


图 3.11 接听电话（支持带内铃声）

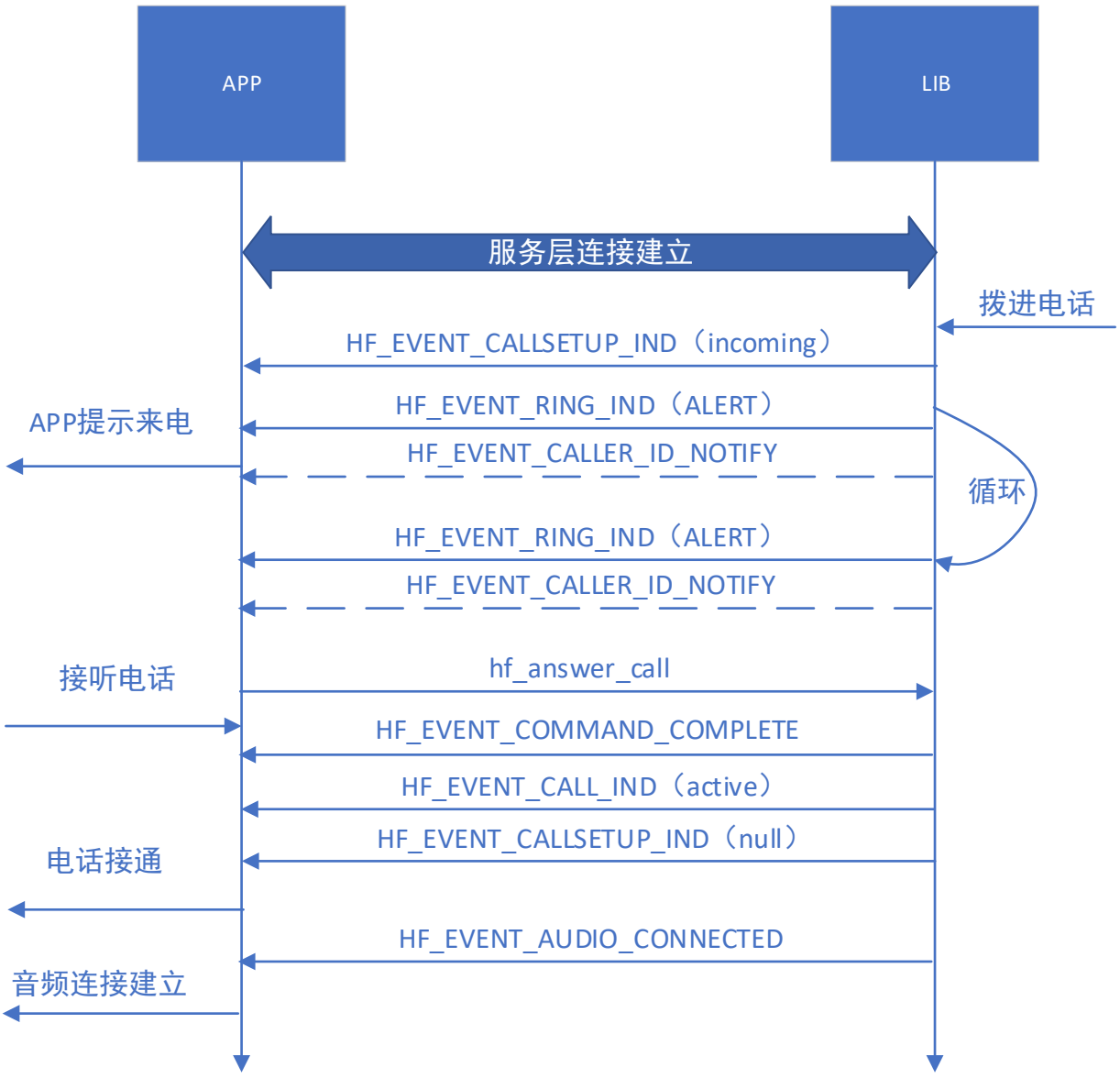


图 3.12 接听电话（不支持带内铃音）

3.2.1.5 HFP 拒接电话

如图 3.13 所示，当有电话拨进时，APP 可以发送 hf_hang_up 挂断电话，此时会返回 HF_EVENT_CALLSETUP_IND 事件，表示通话已结束。当手机端主动拒接时，也会返回该事件。另外，hf_hang_up 命令可以控制挂断电话，可以在正在播出电话时或者正在通话的时候发送该命令。

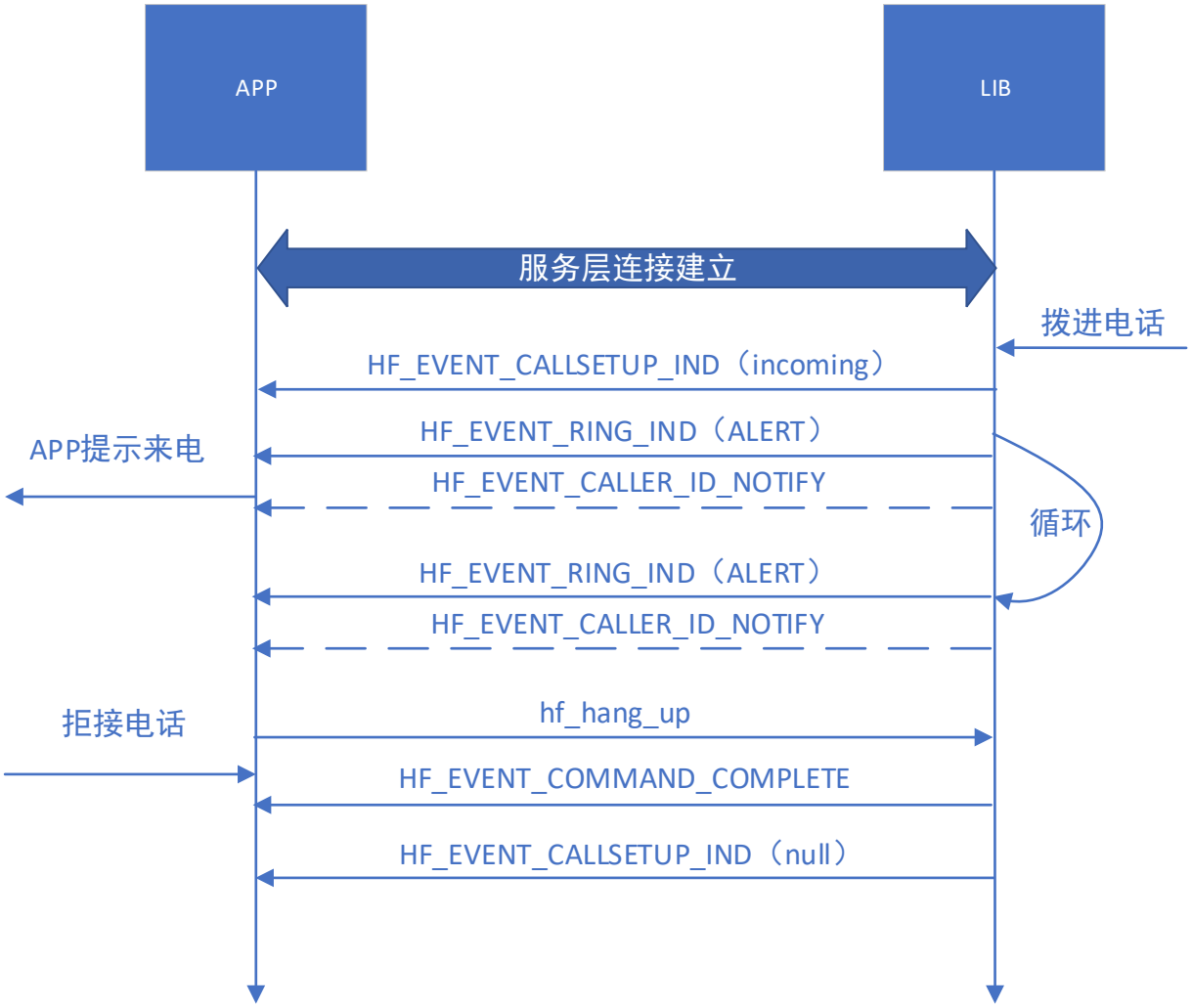


图 3.13 拒接电话

3.2.1.6 HFP 回拨电话

如图 3.14 所示，APP 发起 hf_redial 后，成功发送后，会收到 HF_EVENT_CALLSETUP_IND 和 HF_EVENT_AUDIO_CONNECTED 事件，提示音频连接建立，之后会再次收到 HF_EVENT_CALLSETUP_IND，表明远端手机已经收到来电，远端接通电话后，返回 HF_EVENT_CALL_IND 和 HF_EVENT_CALLSETUP_IND，表明电话接通。

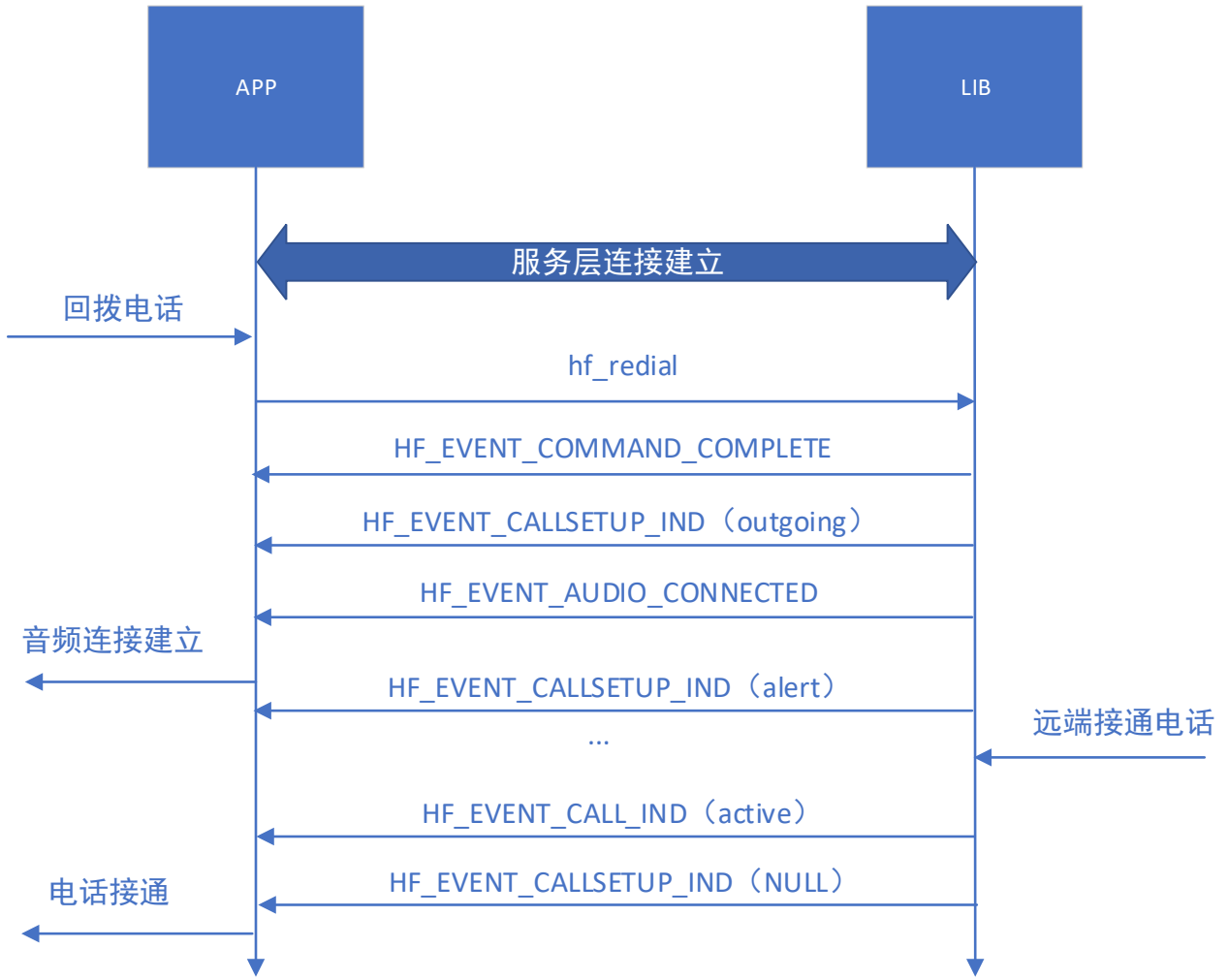


图 3.14 回拨电话

3.2.1.7 HFP 激活与释放 AG 语音助手

如图 3.15 所示，当服务层连接建立好后，APP 可以发送 hf_enable_voice_recognition（true）激活手机端语音助手，激活成功，会返回 HF_EVENT_AUDIO_CONNECTED 事件 ，表明音频连接建立，此时手机语音助手已被调出；当不需要语音助手时，可以发送 hf_enable_voice_recognition（fasle）退出，或者等超时自动退出，成功退出返回 HF_EVENT_AUDIO_DISCONNECTED 事件。

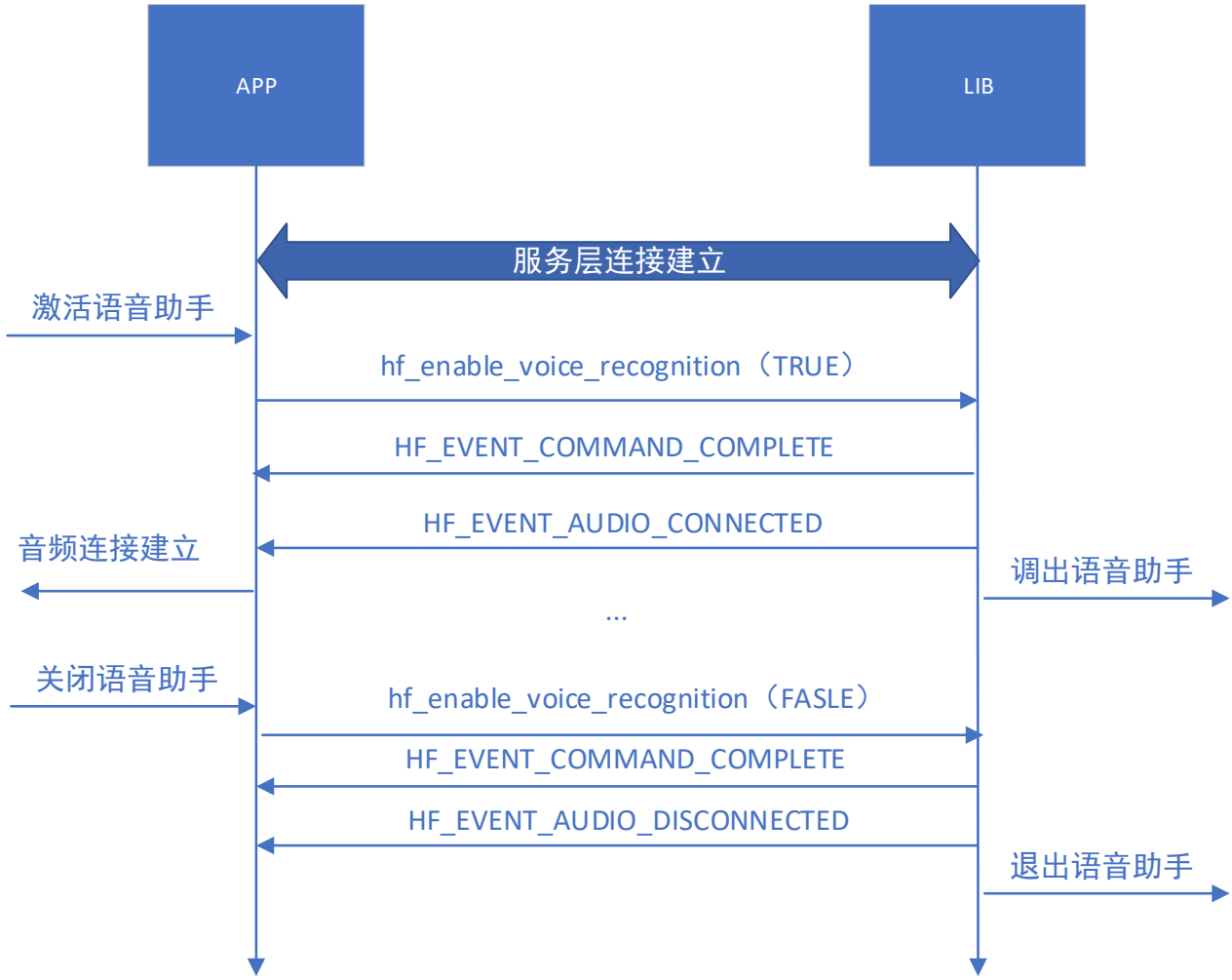


图 3.15 语音助手打开与关闭

3.2.2 HFP 回调函数示例

```
1.  void bt_hf_evt_func(hf_event_t *event)
2.  {
3.      uint8_t i = 0;
4.      uint8_t deviceNum = NUM_DEVICES;
5.      const char *callerid_ptr;
6.      uint8_t *buf;
7.      BtStatus status;
8.
9.      ///查询本地设备列表中，改地址对应的设备序号
10.     deviceNum = bt_find_device(event->addr);
11.     //printf("hf func:event=%d\r\n",event->type);
12.     switch(event->type){
13.         ///收到 AG 发起的 HFP 连接请求
14.         case HF_EVENT_SERVICE_CONNECT_REQ:
```

```
            ///将 responder 标志位置 TRUE
            user_bt_env->dev[deviceNum].responder = TRUE;
        break;
        ///HFP 服务层连接建立完成
        case HF_EVENT_SERVICE_CONNECTED:
        {
            printf("hf profile connected,devnum=%d\r\n",deviceNum);
            ///保存 HFP 连接信息
            user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_HF_CONNECTED;
            user_bt_env->dev[deviceNum].hf_chan = event->chan;
            ///若 responder 标志位没有被置起，表示是 HF 端主动连接，此时需要根据 pskeys 及状态看
            ///是否需要建立 A2DP 连接
            if((user_bt_env->dev[deviceNum].responder == FALSE)
                &&((user_bt_env->dev[deviceNum].conFlags&LINK_STATUS_AV_CONNECTED)==0)
                &&(pskeys.enable_profiles & ENABLE_PROFILE_A2DP)){
                a2dp_open_stream(AVDTP_STRM_ENDPOINT_SRC, event->addr);
            }
            ///使能来电显示
            status = hf_enable_caller_id_notify(event->chan, TRUE);
            //printf("status= %d\r\n",status);
            ///使能三方通话
            status = hf_enable_call_wait_notify(event->chan, TRUE);
            ///不使能远端 NREC 算法
            status = hf_disable_nrec(user_bt_env->dev[deviceNum].hf_chan);
            ///使能耳机电池电量显示
            buf = (uint8_t *)os_malloc(sizeof("AT+XAPL=AAAA-1111_01,10"));
            if(buf != NULL) {
                memcpy(buf, "AT+XAPL=AAAA-1111_01,10", sizeof("AT+XAPL=AAAA-1111_01,10"));
                hf_send_at_command(event->chan, (const uint8_t *)buf);
            }
            ///查询本地所有连接是否都已成功连接上，若是，则保存连接信息到 flash
            app_bt_check_conn(deviceNum);
        }
        break;
        ///HFP 服务层连接断开
        case HF_EVENT_SERVICE_DISCONNECTED:
        {
            printf("hf disconnected,devnum=%d,err=%d\r\n",deviceNum,event->errCode);
            ///保存相应信息到本地设备
            user_bt_env->dev[deviceNum].responder = FALSE;
            user_bt_env->dev[deviceNum].conFlags &= ~LINK_STATUS_HF_CONNECTED;
            user_bt_env->dev[deviceNum].hf_chan = event->chan;

```

```
58.         }
59.         break;
60.         ///HFP 音频连接成功建立
61.         case HF_EVENT_AUDIO_CONNECTED:
62.         {
63.             ///置位相应状态
64.             user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_SCO_CONNECTED;
65.         }
66.         break;
67.         ///HFP 音频连接断开
68.         case HF_EVENT_AUDIO_DISCONNECTED:
69.         {
70.             ///置位相应状态
71.             user_bt_env->dev[deviceNum].conFlags &= ~LINK_STATUS_SCO_CONNECTED;
72.         }
73.         break;
74.         ///HFP 通话状态上报
75.         case HF_EVENT_CALL_IND:
76.             user_bt_env->dev[deviceNum].active  = event->param.call;
77.         break;
78.         ///HFP 通话建立状态上报
79.         case HF_EVENT_CALLSETUP_IND:
80.             user_bt_env->dev[deviceNum].setup_state = event->param.callSetup;
81.         break;
82.         ///来电号码提示
83.         case HF_EVENT_CALLER_ID_NOTIFY:
84.             ///log 打印来电号码
85.             printf("caller id: ");
86.             callerid_ptr = event->param.callerIdParms->number;
87.             i = 0;
88.             while(callerid_ptr[i] != '\0'){
89.                 printf("%c",callerid_ptr[i]);
90.                 i++;
91.             }
92.             printf("\r\n");
93.         break;
94.         ///HFP AT 命令发送完成
95.         case HF_EVENT_COMMAND_COMPLETE:
96.             ///检测是否是自定义 AT 命令发送完成
97.             if(event->param.command->type == HF_COMMAND_SEND_AT_COMMAND){
98.                 if(memcmp((uint8_t *)event->param.command->parms[0], "AT+XAPL=AAAA-1111_0
99.                   1,10", sizeof("AT+XAPL=AAAA-1111_01,10"))==0){
100.                    buf = (uint8_t *)os_malloc(24);
```

```
                    if(buf != NULL) {
                        ///显示电量
                        memcpy(buf, "AT+IPHONEACCEV=1,1,9", 24);
                        hf_send_at_command(event->chan, (const uint8_t *)buf);
                    }
                }
                ///注意释放发送自定义 At 命令时所分配的内存
                os_free((uint8 *)event->param.command->parms[0]);
            }
        break;

        default:
            break;
    }
}
```

3.3 A2DP 协议

A2DP(Advanced Audio Distribution Profile) 即蓝牙音频传输模型协议，是一种使用蓝牙非同步传输信道方式（ACL），传输高质量音乐文件数据的协议。A2DP 分为 SOURCE（SRC）和 SINK（SNK）两种角色，SRC 将数据音源传送给 SNK，一般情况下手机是 SRC，耳机为 SNK。FR5080 中 A2DP 协议支持 SRC 和 SNK 两种角色，可以接收手机 SRC 传过来的音乐，也可以将本地音乐传送给其他耳机 SNK。注意一次连接中只能是一种角色，可以在连接断开后，切换角色。

3.3.1 A2DP 基础流程

下续章节描述了 A2DP 的一些连接过程，及播放暂停，SRC 和 SNK 切换相关流程，具体函数定义及参数说明，请参考 a2dp_api.h。

3.3.1.1 A2DP 连接建立过程

如图 3.16 和图 3.17 所示，FR5080 SDK 对于 A2DP 连接进行了简化，首先 LIB 进行了 A2DP 的 Stream 注册，配置好本地支持的音频编码格式，通常是 SBC 和 AAC，当远端设备发起连接时，会收到 A2DP_EVENT_STREAM_OPEN_IND 事件，该事件带有远端设备支持的音频编码格式及参数，对比本地和远端参数，选择合适的编码格式，并利用 a2dp_open_stream_rsp 返回消息。当远端设备收到 response 后，会检测 response 返回的参数是 A2DP_ERR_NO_ERROR，则 APP 会收到 A2DP_EVENT_STREAM_OPEN，表明 A2DP 连接成功。若是本地 APP 发起 A2DP 连接，若远端回复了正确的 response，则也会收到 A2DP_EVENT_STREAM_OPEN，表明 A2DP 连接成功。

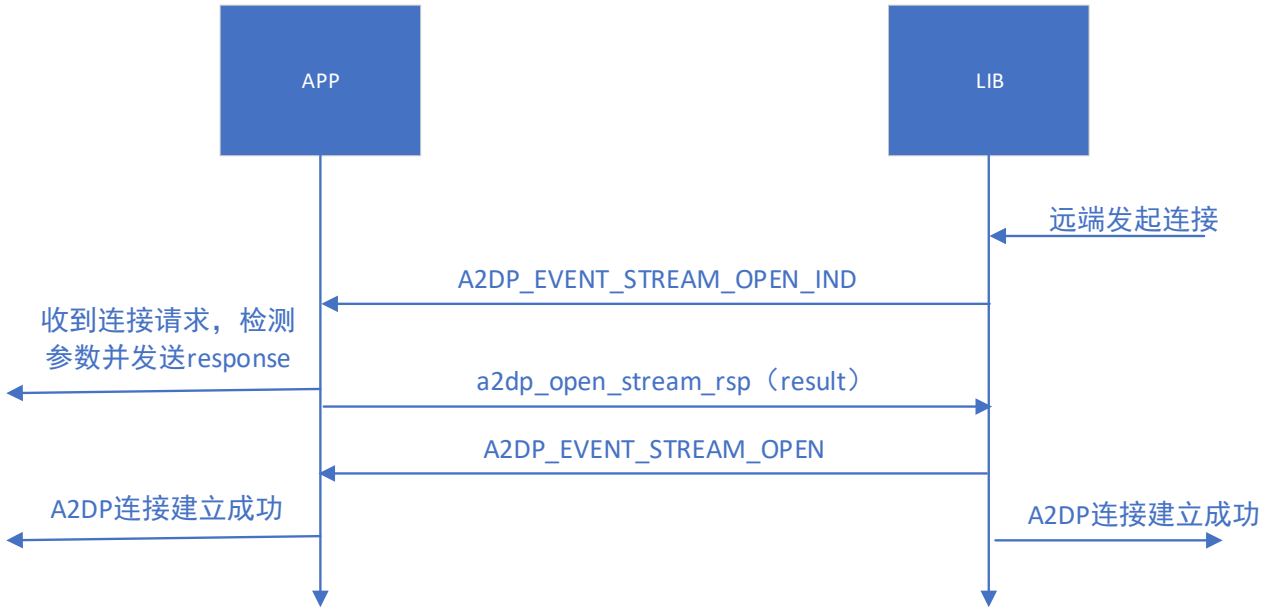


图 3.16 远端发起 A2DP 连接

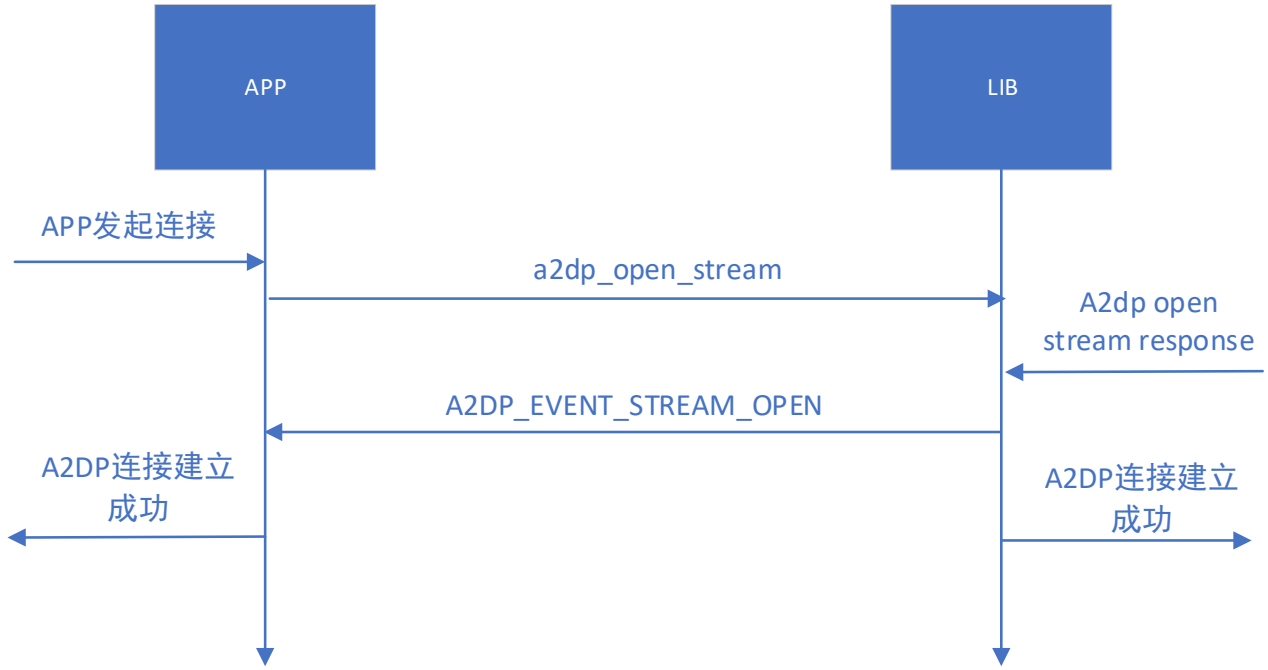


图 3.17 本地发起 A2DP 连接

3.3.1.2 A2DP 流传输开始与暂停

图 3.18 和图 3.19 描述本地和远端发起流传输及暂停流传输的简单流程。一般情况下，发起流传输的设备都是 SRC，当本地发起流传输时，调用 a2dp_start_stream，若成功打开，则返回 A2DP_EVENT_STREAM_STARTED 事件，之后本地可以调用 a2dp_stream_send_sbc_packet 发送数据包，成功发送完后返回 A2DP_EVENT_STREAM_SBC_PACKET
_SENT 事件，当 SRC 端音频数据播放完成后或者收到暂停信号，本地发送 a2dp_suspend_stream，成功发送则返回 A2DP_EVENT_STREAM_SUSPENDED，此时音频数据流暂停。若是远端 SRC 发起流传输时，首先会收到

A2DP_EVENT_STREAM_START_IND 事件，本地需要回复 a2dp_start_stream_rsp，远端收到后返回 A2DP_EVENT_STREAM_STARTED 事件，表明流数据传输开始。若此后收到 A2DP_EVENT_STREAM_SUSPENDED 事件，表明流数据传输结束或者暂停。

注意：流数据传输的开始与暂停，一般有 SRC 发起；一般控制播放暂停，建议使用 AVRCP 协议中 API，A2DP 中的 A2DP_EVENT_STREAM_STARTED 和 A2DP_EVENT_STREAM_SUSPENDED 事件仅作为 A2DP 状态指示。

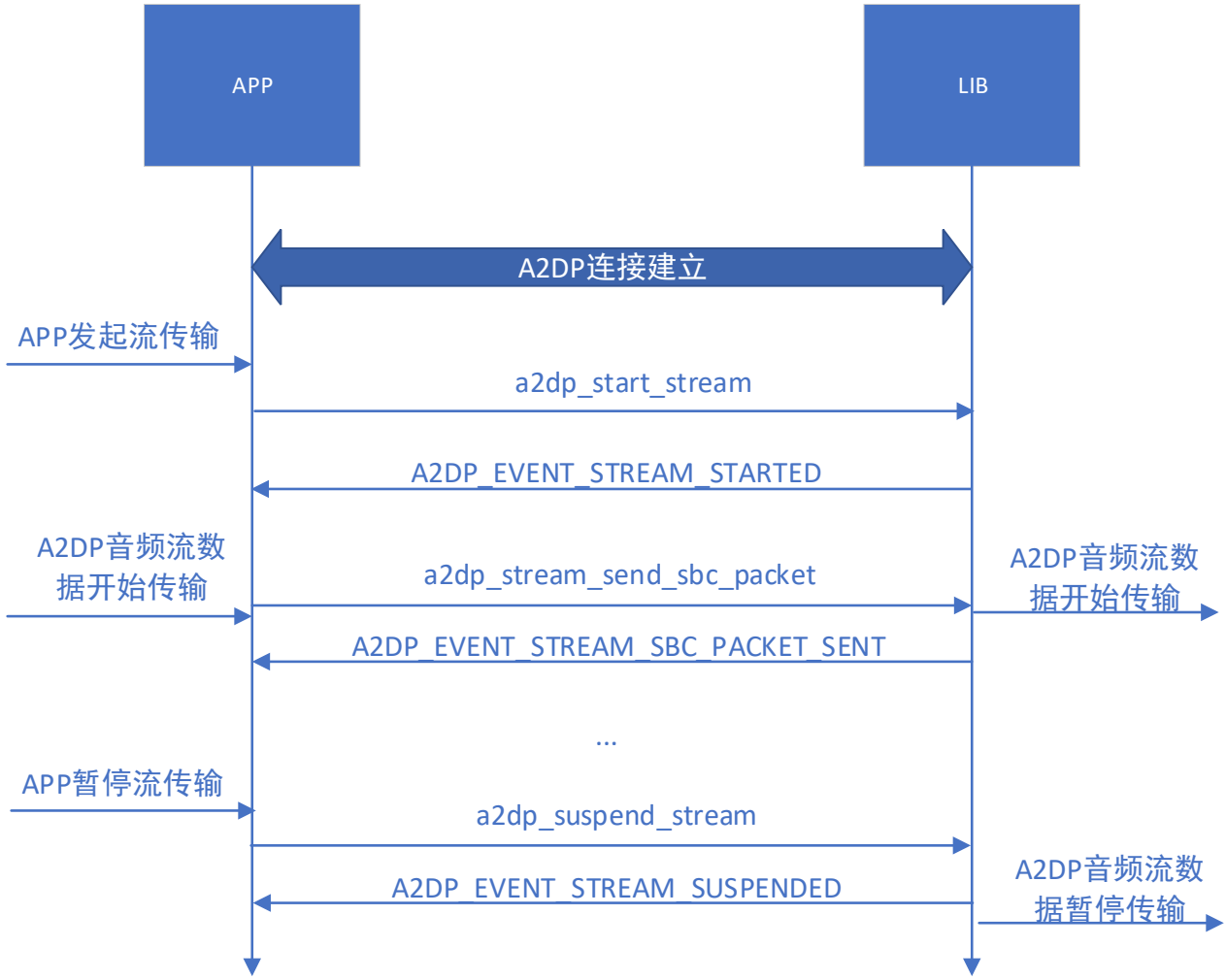


图 3.18 本地 SRC 发起 A2DP 流传输及暂停

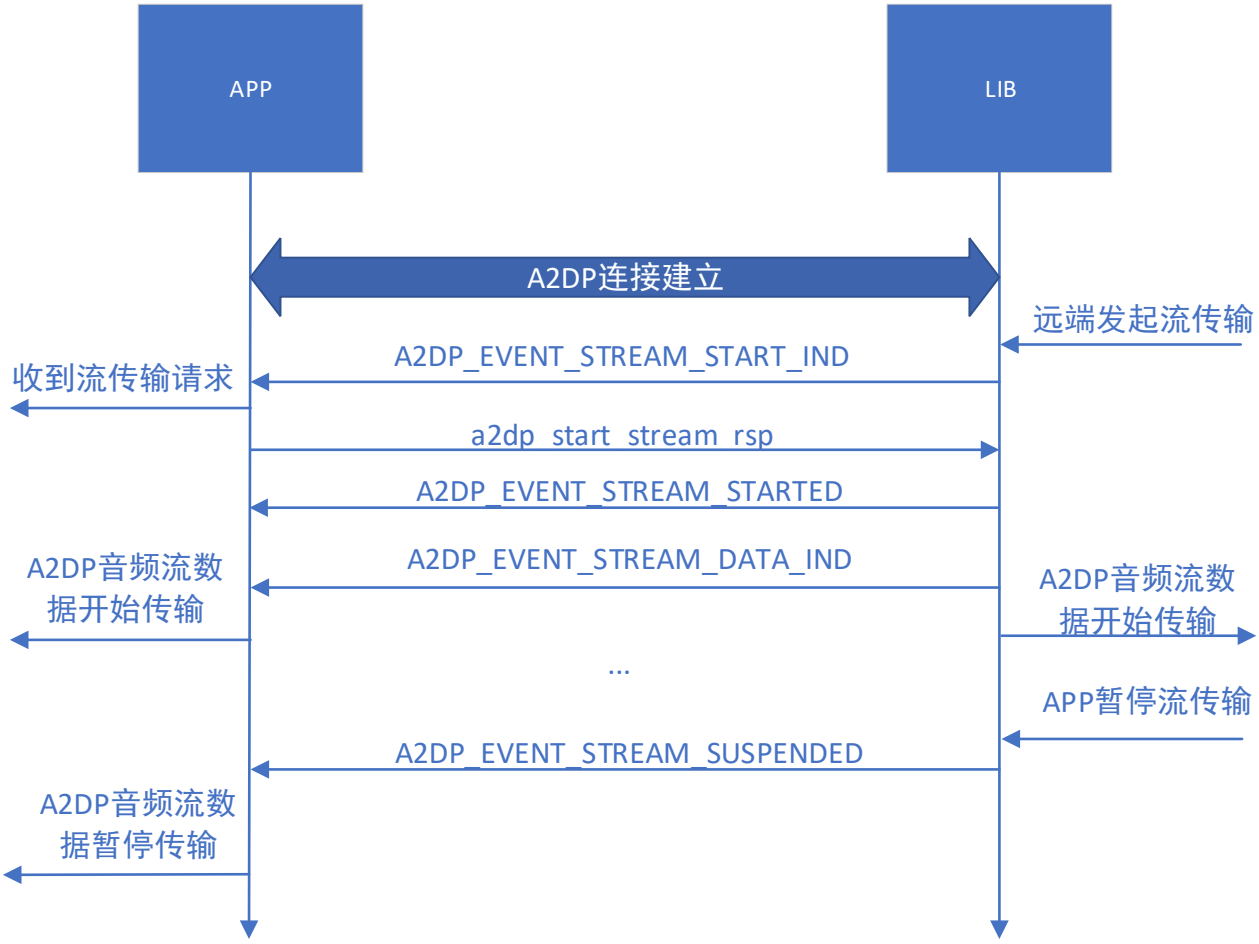


图 3.19 远端 SRC 发起 A2DP 流传输及暂停

3.3.1.3 A2DP 断开连接

如图 3.20 所示，远端和本地都可以发起断开连接，本地断开时，需发送 a2dp_close_stream，成功关闭后，会返回 A2DP_EVENT_STREAM_CLOSED 事件，部分设备也会产生 A2DP_EVENT_STREAM_IDLE 事件

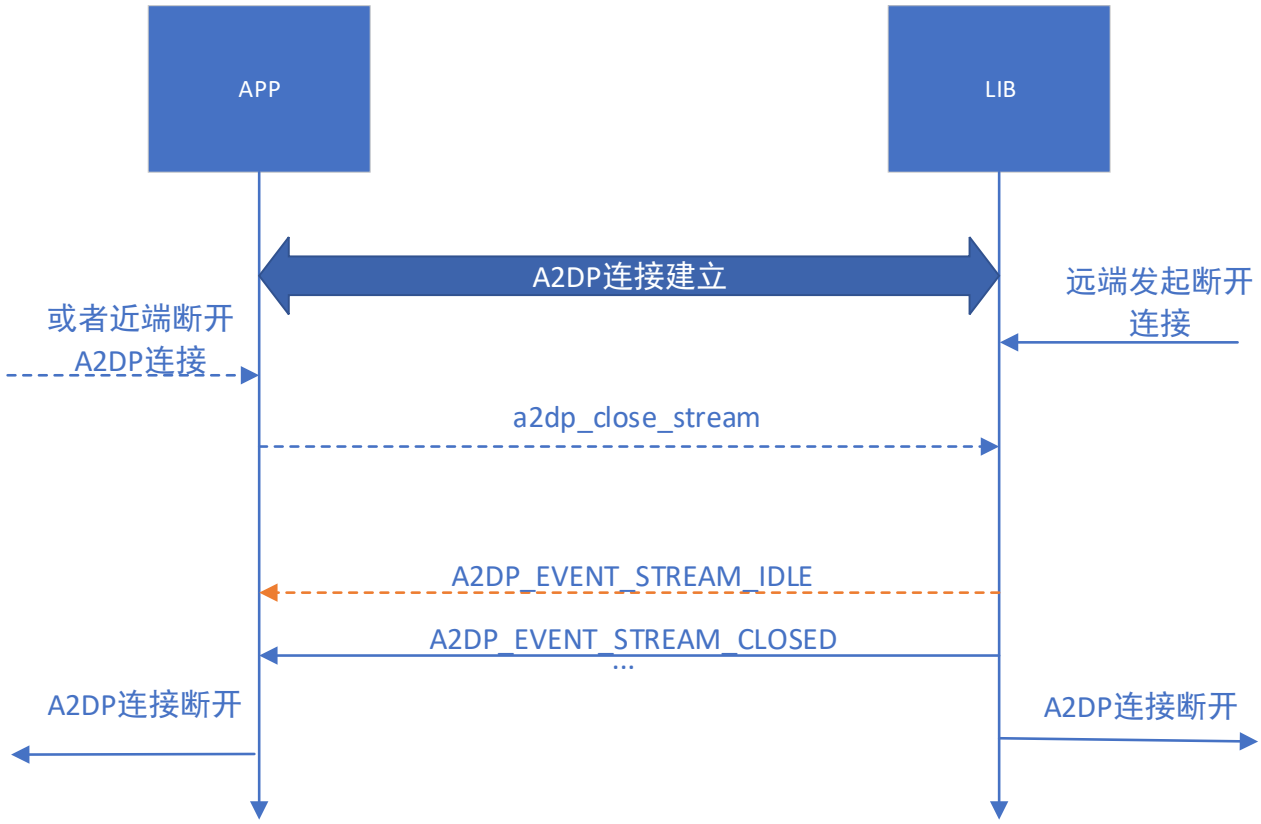


图 3.20 A2DP 断开连接

3.3.1.4 A2DP SRC 和 SNK 切换

FR5080 上电后 A2DP 角色默认为 SNK，在空闲状态时（配对或者待机状态），可以通过调用 bt_set_a2dp_type 更改 A2DP 角色，利用 bt_get_a2dp_type 可以获取当前的 A2DP 角色。

3.3.2 A2DP 回调函数示例

```
1.  void bt_a2dp_evt_func(a2dp_event_t *event)
2.  {
3.      uint8_t deviceNum = NUM_DEVICES;
4.      uint8_t error = A2DP_ERR_NO_ERROR;
5.      AvdtpCodec       *codec;
6.      uint8_t               *elements;
7.      uint8_t               *reqElements;
8.      ///查询本地设备列表中，该地址对应的设备序号
9.      deviceNum = bt_find_device(event->addr);
10.     if(deviceNum >= NUM_DEVICES){
11.         printf("av func: event=%d,erorr device num.....\r\n",event->event);
12.         return;
13.     }
14.     //printf("av func:%d\r\n",event->event);
```

```
    switch(event->event){
        ///收到远端的 a2dp 连接请求
        case A2DP_EVENT_STREAM_OPEN_IND:
            ///远端请求 codec 类型为 sbc
            if (AVDTP_CODEC_TYPE_SBC == event->codec->codecType) {
                ///获取当前 stream 注册的 codec 类型
                codec = a2dp_get_registered_codec(event->stream);
                elements = codec->elements;
                reqElements = event->codec->elements;

                ///判断请求的参数是否与本地注册的参数冲突
                if (!(reqElements[0] & (elements[0] & 0xF0))) {
                    error = A2DP_ERR_NOT_SUPPORTED_SAMP_FREQ;
                } else if (!(reqElements[0] & (elements[0] & 0x0F))) {
                    error = A2DP_ERR_NOT_SUPPORTED_CHANNEL_MODE;
                } else if (!(reqElements[1] & (elements[1] & 0x0C))) {
                    error = A2DP_ERR_NOT_SUPPORTED_SUBBANDS;
                } else if (!(reqElements[1] & (elements[1] & 0x03))) {
                    error = A2DP_ERR_NOT_SUPPORTED_ALLOC_METHOD;
                } else if (reqElements[2] < elements[2]) {
                    error = A2DP_ERR_NOT_SUPPORTED_MIN_BITPOOL_VALUE;
                } else if (reqElements[3] > elements[3]) {
                    error = A2DP_ERR_NOT_SUPPORTED_MAX_BITPOOL_VALUE;
                }
                ///保存请求的 sbc 的采样率
                codec->freq = (reqElements[0] & 0xF0) >> 5;
                ///回复 response 给远端
                a2dp_open_stream_rsp(event->stream, error, 0);
            }
            ///远端请求 codec 类型为 aac
            else if(AVDTP_CODEC_TYPE_MPEG2_4_AAC== event->codec->codecType) {
                ///获取当前 stream 注册的 codec 类型
                codec = a2dp_get_registered_codec(event->stream);
                elements = codec->elements;
                reqElements = event->codec->elements;

                ///保存请求的 sbc 的采样率
                if(reqElements[1] & 0x01) {
                    ///44.1k
                    codec->freq = 1;
                }
                else if(reqElements[2] & 0x80) {
                    ///48k
```

```
                    codec->freq = 0;
                }
                ///回复 response 给远端
                a2dp_open_stream_rsp(event->stream, error, 0);
            }
            else {
                ///不支持的 codec 类型，拒绝连接请求
                a2dp_open_stream_rsp(event->stream, AVRCP_ERR_UNKNOWN_ERROR, 0);
            }
            break;
        ///A2DP stream 成功打开
        case A2DP_EVENT_STREAM_OPEN:
            ///保存状态信息及 stream 到本地
            user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_AV_CONNECTED;
            user_bt_env->dev[deviceNum].pstream = event->stream;

            ///查询本地所有连接是否都已成功连接上，若是，则保存连接信息到 flash
            app_bt_check_conn(deviceNum);
            ///若 AVRCP 连接还没有建立，发起 avrcp 连接
            if((user_bt_env->dev[deviceNum].conFlags&LINK_STATUS_AVC_CONNECTED) == 0){
                avrcp_connect(event->addr);

            }
            printf("av profile connected\r\n");
            break;
        ///A2DP stream 关闭
        case A2DP_EVENT_STREAM_CLOSED:
            ///保存状态信息到本地
            printf("av profile disconnected\r\n");
            user_bt_env->dev[deviceNum].conFlags &= (~LINK_STATUS_AV_CONNECTED);
            user_bt_env->dev[deviceNum].pstream = NULL;
            break;
        ///A2DP stream 空闲
        case A2DP_EVENT_STREAM_IDLE:
            ///保存状态信息到本地
            user_bt_env->dev[deviceNum].conFlags &= (~LINK_STATUS_AV_CONNECTED);
            user_bt_env->dev[deviceNum].pstream = NULL;
            break;
        ///A2DP stream 开始传输请求
        case A2DP_EVENT_STREAM_START_IND:
            ///回复 response
            a2dp_start_stream_rsp(user_bt_env->dev[deviceNum].pstream , A2DP_ERR_NO_ERROR);
```

```
            break;
        ///A2DP stream 开始传输
        case A2DP_EVENT_STREAM_STARTED:
            ///保存状态信息到本地
            user_bt_env->current_playing_index = deviceNum;
            user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_MEDIA_PLAYING;
            break;
        ///A2DP stream 暂停
        case A2DP_EVENT_STREAM_SUSPENDED:
            user_bt_env->dev[deviceNum].conFlags &= (~LINK_STATUS_MEDIA_PLAYING);
            break;
        ///A2DP stream sbc 数据包发送完毕
        case A2DP_EVENT_STREAM_SBC_PACKET_SENT:
            break;
        ///A2DP 媒体数据
        case A2DP_EVENT_STREAM_DATA_IND:
            break;
        default:
            printf("av event %d\r\n",event->event);
            break;
    }
}
```

3.4 AVRCP 协议

AVRCP（AUDIO/VIDEO REMOTE CONTROL PROFILE）协议定义了蓝牙设备音频/视频远程控制的特征和流程，确保不同蓝牙设备之间音视频传输控制的兼容。一般包括暂停，播放，上一曲，下一曲，音量控制等远程控制操作。

AVRCP 定义了两种角色，CT（CONTROLLER）和 TG（TARGET）,一般情况下，CT 就是可以发送命令控制 TG 的设备，如蓝牙耳机；TG 可以接收 CT 发过来的指令，同时也能通知 CT 一些状态的改变，如手机。一种设备可以同时支持两种角色，如 FR5080，既可以做为 CT，发送控制指令，控制手机播放暂停，也同时做为 TG，用来控制手机音量。AVRCP 定义了四大类别（Categories），

3.4.1 AVRCP 基础流程

下续章节描述了 FR5080 SDK 中一些基本流程，例如 AVRCP 连接建立和断开，事件查询与注册，音量控制等。其中定义的一些 API 及事件描述可参考 avrcp_api.h。

3.4.1.1 AVRCP 连接建立与断开

如图 3.21 所示，FR5080 中，APP 和远端手机都可以发起 AVRCP 连接，当 APP 发起 AVRCP 连接时，调用 avrcp_connect,发送成功后，会收到 AVRCP_EVENT_CONNECT 事件，表明连接成功；若是远端发起 AVRCP 连接，

会先收到 AVRCP_EVENT_CONNECT_IND 事件，此时 APP 需要回复 avrcp_connect_rsp 给远端，接受或者拒绝改连接。连接成功也会收到 AVRCP_EVENT_CONNECT 事件。连接建立完成后，APP 和远端都可以发起断开连接操作，APP 断开连接调用 avrcp_disconnect，成功断开会收到 AVRCP_EVENT_DISCONNECT 事件。

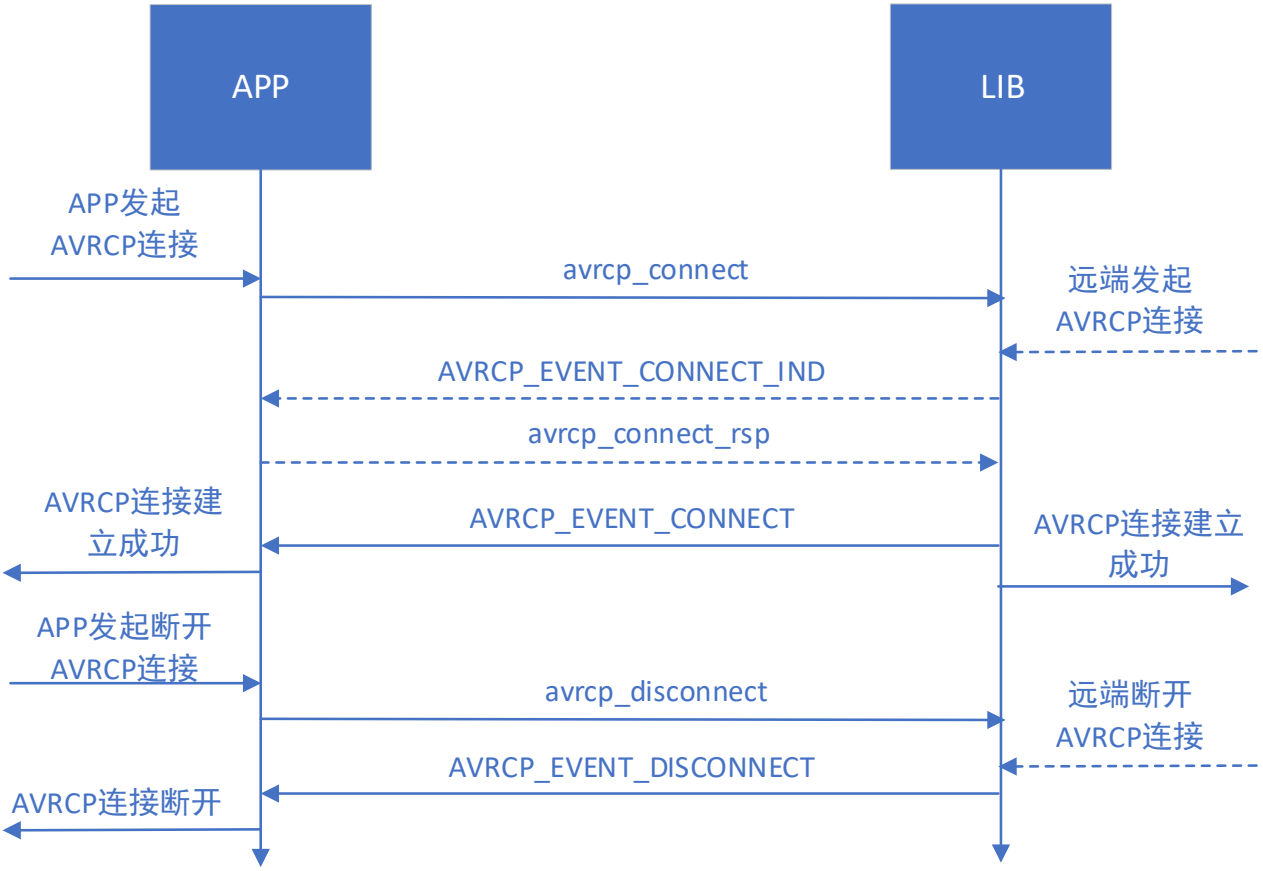


图 3.21 AVRCP 连接建立与断开

3.4.1.2 AVRCP 属性查询及注册

如图 3.22 所示，当 AVRCP 连接建立完成后，本地 APP 可以查询远端 TG 支持的属性能力（capabilities），APP 根据应用选择注册通知。本例中，通过 avrcp_ct_get_capabilities 查询 TG 的属性，成功发送返回 AVRCP_EVENT_ADV_
TX_DONE 事件，远端同时会发送其支持的属性信息，以 AVRCP_EVENT_ADV_RESPONSE 事件方式传给 APP，APP 发现远端支持播放状态改变消息，发送 avrcp_ct_register_notification 注册该消息（AVRCP_ENABLE_PLAY_STATUS
_CHANGED），若发送成功，返回 AVRCP_EVENT_ADV_TX_DONE 事件，之后会收到 AVRCP_EVENT_ADV_RESPONSE 事件，返回当前 TG 的播放状态初始值。当 TG 开始播放音乐时，会产生 AVRCP_EVENT_ADV_NOTIFY 通知本地播放状态改变，此后本地若想继续监测 TG 的播放状态改变，需再次注册该消息。

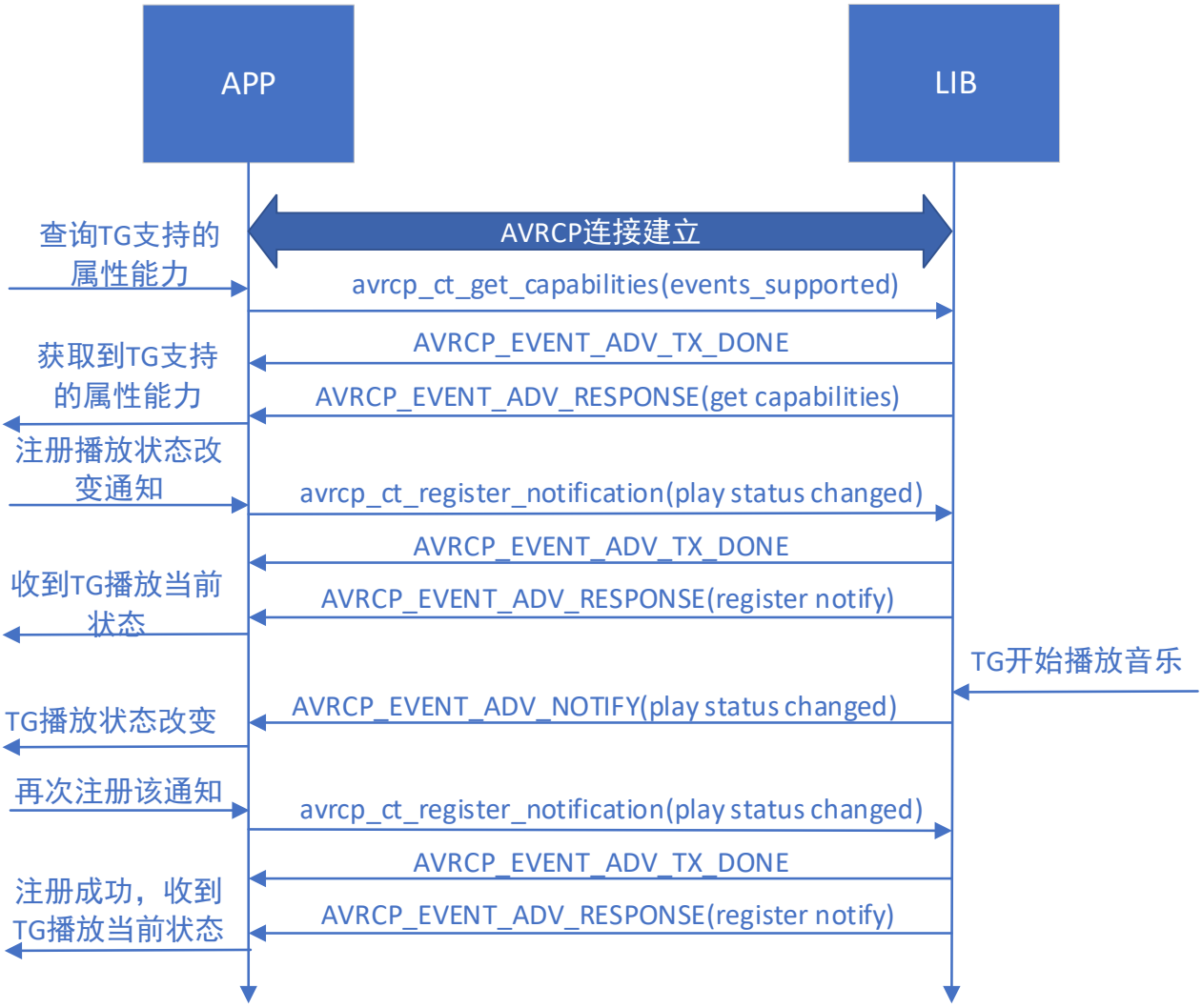


图 3.22 AVRCP 属性查询及注册

3.4.1.3 AVRCP 常用控制命令

FR5080 支持常用的 AVRCP 控制命令，例如：播放（AVRCP_POP_PLAY），暂停（AVRCP_POP_PAUSE），上一曲（AVRCP_POP_BACKWARD），下一曲（AVRCP_POP_FORWARD），快进（AVRCP_POP_FAST_FORWARD），快退（AVRCP_POP_REWIND）等。用户可以调用 avrcp_set_panel_key 发送这些控制命令。如图 3.23 所示，APP 发送完 avrcp_set_panel_key 按下命令，会收到 AVRCP_EVENT_PANEL_CNF 事件，注意对于播放，暂停，上一曲，下一曲命令，应立即继续利用 avrcp_set_panel_key 发送对应的释放命令，此时会再次收到会收到 AVRCP_EVENT_PANEL
_CNF 事件；对于快进，快退命令，发送完 avrcp_set_panel_key 按下命令后，立即生效，若要停止，则调用 avrcp_set_panel_key 发送对应的释放命令。

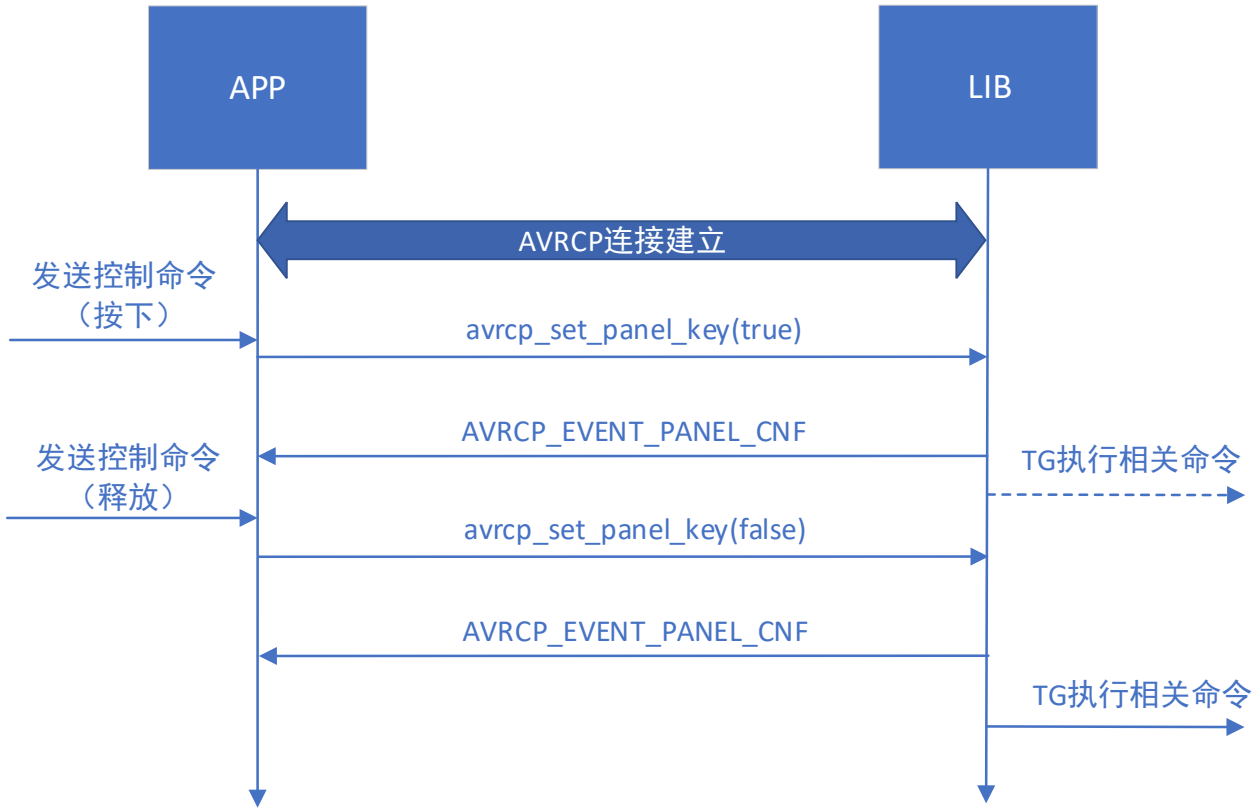


图 3.23 AVRCP 发送控制命令

3.4.1.4 AVRCP 音量控制

FR5080 中可以同步控制手机的音量（需要手机支持）。在 AVRCP 连接建立完成好后，本地可以发送 avrcp_tg_set_absolute_volume 命令，将本地音量同步到手机 AG 端；手机端也可以通过 AVRCP_EVENT_ADV_INFO（AVRCP_OP_SET_ABSOLUTE_VOLUME）事件控制本地音量。

3.4.2 AVRCP 回调函数示例

```
1.  void bt_avrcp_evt_func(avrcp_event_t *event)
2.  {
3.      uint8_t deviceNum;
4.
5.      ///查询本地设备列表中，该地址对应的设备序号
6.      deviceNum = bt_find_device(event->addr);
7.      if(deviceNum >= NUM_DEVICES){
8.          printf("avrcp func: event=%d,erorr device num.....\r\n",event->event);
9.      }
10.     switch(event->event){
11.         ///收到远端连接请求
12.         case AVRCP_EVENT_CONNECT_IND:
13.         {
```

```
            ///接收该连接并发送 response 给远端
            avrcp_connect_rsp(event->chnl, true);
        }
        break;
        ///AVRCP 连接成功建立
        case AVRCP_EVENT_CONNECT:
        {
            ///存储本地状态信息
            user_bt_env->dev[deviceNum].conFlags|= LINK_STATUS_AVC_CONNECTED;
            user_bt_env->dev[deviceNum].rcp_chan = event->chnl;
            ///查询本地所有连接是否都已成功连接上，若是，则保存连接信息到 flash
            app_bt_check_conn(deviceNum);

            ///查询远端 TG 支持的 event
            avrcp_ct_get_capabilities(event->chnl, AVRCP_CAPABILITY_EVENTS_SUPPORTED);
            ///设置本地 TG 支持的 event
            avrcp_tg_set_event_mask(event->chnl, AVRCP_ENABLE_VOLUME_CHANGED);
            printf("avrcp profile connected...\r\n");
        }
        break;
        ///AVRCP 连接断开
        case AVRCP_EVENT_DISCONNECT:
            ///保存状态信息
            user_bt_env->dev[deviceNum].conFlags &= (~LINK_STATUS_AVC_CONNECTED);
            user_bt_env->dev[deviceNum].rcp_chan = NULL;
            printf("avrcp profile disconnected...\r\n");
            break;
        ///控制命令远端确认消息
        case AVRCP_EVENT_PANEL_CNF:
            ///被确认消息是名利‘按下’事件
            if(event->param.panelCnf.press == TRUE){
                ///根据不同命令，自动发送对应的‘释放’命令
                switch(event->param.panelCnf.operation) {
                    case AVRCP_POP_PAUSE:
                        avrcp_set_panel_key(event->chnl, AVRCP_POP_PAUSE, FALSE);
                        break;
                    case AVRCP_POP_PLAY:
                        avrcp_set_panel_key(event->chnl, AVRCP_POP_PLAY, FALSE);
                        break;
                    case AVRCP_POP_FORWARD:
                        avrcp_set_panel_key(event->chnl, AVRCP_POP_FORWARD, FALSE);
                        break;
                    case AVRCP_POP_BACKWARD:
```

```
                        avrcp_set_panel_key(event->chnl, AVRCP_POP_BACKWARD, FALSE);
                        break;
                    case AVRCP_POP_STOP:
                        avrcp_set_panel_key(event->chnl, AVRCP_POP_STOP, FALSE);
                        break;
                }
            }
            break;
        ///接收远端 TG 的控制信息，通常只有音量信息
        case AVRCP_EVENT_ADV_INFO:
            if(event->advOp == AVRCP_OP_SET_ABSOLUTE_VOLUME) {
                printf("SET_ABSOLUTE_VOLUME is %d.\r\n", event->param.adv.info.volume);
            }
            break;
        ///本地 ADV 命令成功发送
        case AVRCP_EVENT_ADV_TX_DONE:
            break;
        ///回复本地查询及注册命令事件
        case AVRCP_EVENT_ADV_RESPONSE:
            ///本地查询 TG 支持事件，返回消息
            if((event->advOp == AVRCP_OP_GET_CAPABILITIES)
              &&(event->param.adv.rsp.capability.type == AVRCP_CAPABILITY_EVENTS_SUPPORTED)
              &&(event->param.adv.rsp.capability.info.eventMask & AVRCP_ENABLE_PLAY_STATUS_
             CHANGED)){
                ///注册播放状态改变事件，当 TG 播放状态改变时，产生 AVRCP_EVENT_ADV_NOTIFY 事件
                avrcp_ct_register_notification(event->chnl, AVRCP_ENABLE_PLAY_STATUS_
                                               CHANGED, 0);
            }
            ///本地注册事件，返回的当前状态
            if(event->advOp == AVRCP_OP_REGISTER_NOTIFY) {
                ///播放状态改变通知的当前状态
                if(event->param.adv.notify.event == AVRCP_EID_MEDIA_STATUS_CHANGED) {
                    if((event->param.adv.notify.p.mediaStatus == AVRCP_MEDIA_STOPPED)
                        ||(event->param.adv.notify.p.mediaStatus == AVRCP_MEDIA_PAUSED)) {
                        ///保存状态到本地，音乐暂停状态比 A2DP 中的 suspend 事件更早产生，
                        ///有利于 APP 状态判断
                        user_bt_env->dev[deviceNum].conFlags &= ~LINK_STATUS_MEDIA_PLAYING
                    }
                    else {
                        user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_MEDIA_PLAYING;
                        user_bt_env->current_playing_index = deviceNum;
                    }
                }
```

```
            }
            break;
        ///注册的事件通知返回
        case AVRCP_EVENT_ADV_NOTIFY:
            ///播放状态改变通知
            if(event->param.adv.notify.event == AVRCP_EID_MEDIA_STATUS_CHANGED) {
                if((event->param.adv.notify.p.mediaStatus == AVRCP_MEDIA_STOPPED)
                    ||(event->param.adv.notify.p.mediaStatus == AVRCP_MEDIA_PAUSED)) {
                    user_bt_env->dev[deviceNum].conFlags &= ~LINK_STATUS_MEDIA_PLAYING;
                }
                else {
                    user_bt_env->dev[deviceNum].conFlags |= LINK_STATUS_MEDIA_PLAYING;
                    user_bt_env->current_playing_index = deviceNum;
                }
                ///重新注册播放状态改变通知消息
                avrcp_ct_register_notification(event->chnl, AVRCP_ENABLE_PLAY_STATUS_
                                                CHANGED,0);
            }
            break;
        default:
            printf("other avrcp event %d\r\n",event->event);
            break;
    }
}
```

4. BLE 协议栈

SDK 里面包含了完整的协议栈，虽然 controller 和 host 部分是以库的形式提供，但给出了接口丰富的 API 提供给上层应用开发调用。Profile 则是以源码的形式提供。

4.1 通用访问规范（GAP）

通用访问规范 GAP（Generic Access Profile）定义了蓝牙设备之间如何发现以及建立安全/非安全连接。

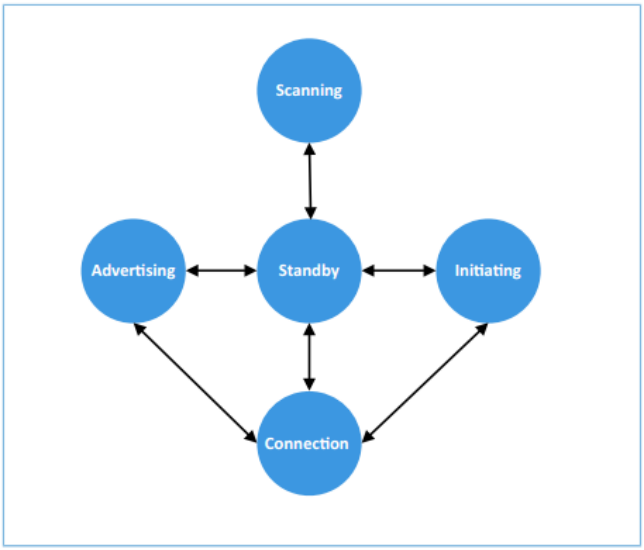
4.1.1 GAP 角色

GAP 定义了四种设备角色：广播者（Broadcaster）、观察者（Observer）、中央设备（Central）、以及外围设备（Peripheral），一个设备可以同时支持多种角色：

- 广播者（Broadcaster）：以广播事件向外发送广播数据，可以在没有接收机的条件下工作。
- 观察者（Observer）：从广播者设备那里接收到广播数据，可以在没有发射机的条件下工作。
- 中央设备（Central）：向另一个设备发起物理连接，链路层从发起态转换为连接态。必须同时具备接收机和发射机。
- 外围设备（Peripheral）：接收另一个设备发起的物理连接，链路层从广播态转换为连接态。必须同时具备接收机和发射机

4.1.2 GAP 访问模式和设备流程

GAP 用于控制设备在多种工作模式间进行切换，包括：设备发现，连接建立，连接终止，设备参数配置等。下图展示了链路层在 GAP 的控制下在个状态之间进行切换的过程：



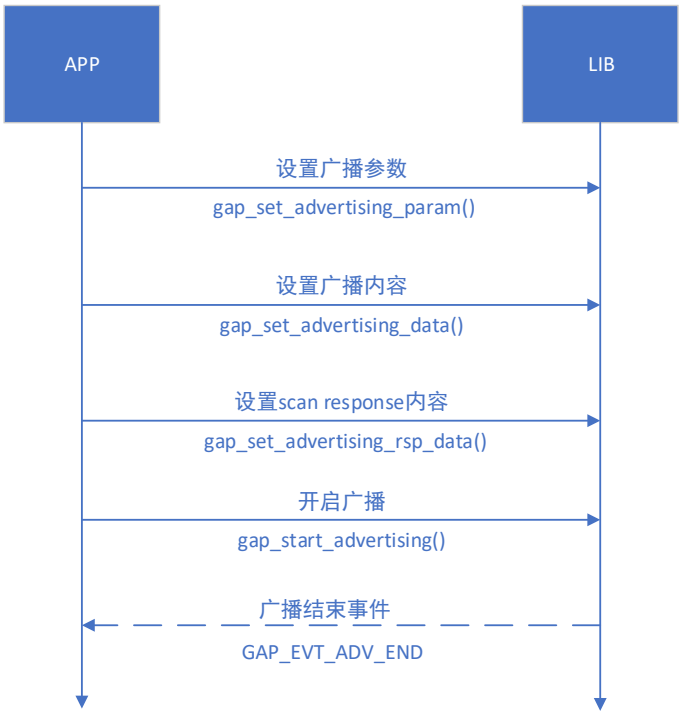
其中各状态描述如下：

- 就绪态（Standby State）：设备上电后处于初始的待机状态。
- 广播态（Advertising State）：设备向外广播特定的数据，以便让发起连接的设备发现这一广播设备。广播数据中包含广播地址及其他广播信息（比如设备名称等）。

- 扫描态（Scanning State）：设备接收广播数据，并向可扫描广播设备发送扫描请求。广播设备收到扫描请求后会回复一个扫描响应数据。这个过程被称作设备发现（Device Discovery）。
- 发起态（Initiating State）：进入发起态的设备必须指定一个想要对其发起连接的对端设备地址。如果收到的广播包中的广播者地址跟指定的对端地址匹配，发起态设备将向广播设备发送一个连接请求。连接请求数据包中包含一些指定的连接参数信息（具体参考 4.1.3.2 连接参数介绍）。
- 连接态（Connection State）：连接建立的时候，处于广播态的设备将作为 Slave 转至连接态，处于发起态的设备将作为 Master 转至为连接态。

4.1.3 开启传统广播流程

本节介绍设备作为外围设备时开启传统广播的流程，用于程序与协议栈库之间的交互流程如下图所示。



1. 设置广播参数：

```
gap_adv_param_t adv_param;
// 设置广播为非定向可连接广播
adv_param.adv_mode = GAP_ADV_MODE_UNDIRECT;
adv_param.adv_addr_type = GAP_ADDR_TYPE_PUBLIC;
// 在 37/38/39 三个信道上均有广播数据
adv_param.adv_chnl_map = GAP_ADV_CHAN_ALL;
adv_param.adv_filt_policy = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
// 广播间隔设定为 0x40*625 = 40ms
adv_param.adv_intv_min = 0x40;
adv_param.adv_intv_max = 0x40;
gap_set_advertising_param(&adv_param);
```

2. 设置广播数据，这一过程为可选，另外 adv_param.adv_mode 为 GAP_ADV_MODE_DIRECT 和 GAP_ADV_MODE_HDC_DIRECT 时设置广播数据不起作用。：

```
uint8_t adv_data[0x1C];
uint8_t *pos;
uint8_t adv_data_len = 0;
pos = &adv_data[0];
uint8_t manufacturer_value[] = {0x00,0x00};
*pos++ = sizeof(manufacturer_value) + 1;
*pos++  = '\xff';
memcpy(pos, manufacturer_value, sizeof(manufacturer_value));
pos += sizeof(manufacturer_value);

uint16_t uuid_value = 0x1812;
*pos++ = sizeof(uuid_value) + 1;
*pos++  = '\x03';
memcpy(pos, (uint8_t *)&uuid_value, sizeof(uuid_value));
pos += sizeof(uuid_value);
adv_data_len = ((uint32_t)pos - (uint32_t)(&adv_data[0]));
gap_set_advertising_data(adv_data,adv_data_len );
```

3. 设置扫描响应数据（可选），adv_mode 为 GAP_ADV_MODE_DIRECT、GAP_ADV_MODE_NON_CONN_NON_SCAN、GAP_ADV_MODE_HDC_DIRECT 和 GAP_ADV_MODE_BEACON 时，不需设置扫描响应数据，其他情况则需设置：

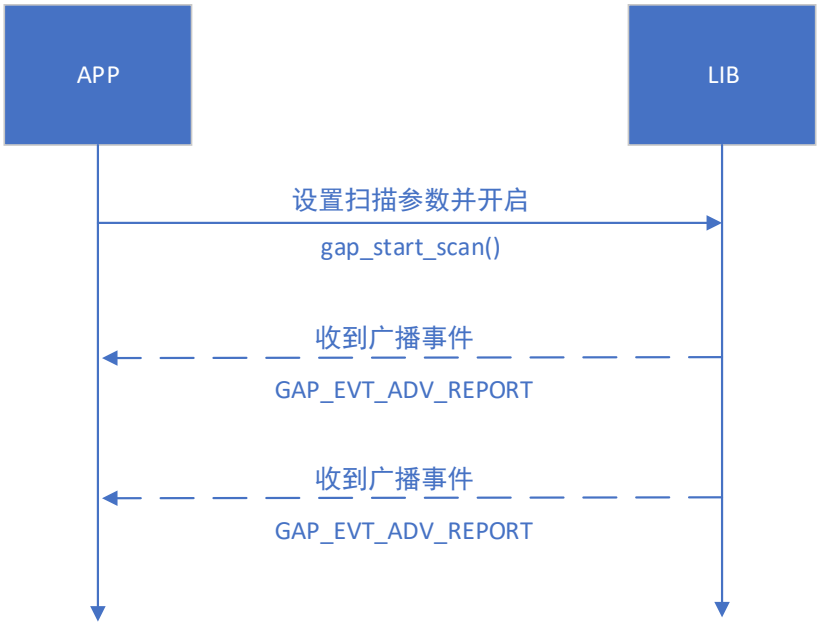
```
uint8_t scan_rsp_data[0x1F];
uint8_t scan_rsp_data_len = 0;

uint8_t local_name[] = "8010H_ADV";
uint8_t local_name_len = sizeof(local_name);
pos = &scan_rsp_data[0];
*pos++ = local_name_len + 1;  //pos len;  (payload + type)
*pos++  = '\x09';   //pos: type
memcpy(pos, local_name, local_name_len);
pos += local_name_len;
scan_rsp_data_len = ((uint32_t)pos - (uint32_t)(&scan_rsp_data[0]));
gap_set_advertising_rsp_data(scan_rsp_data,scan_rsp_data_len );
```

4. 开启广播，调用 gap_start_advertising 函数开启广播时需要用户传入参数 duration，设置广播持续时间。广播持续时间到后，广播停止，协议栈会上传 GAP EVENT：GAP_EVT_ADV_END。
5. 广播停止，调用 gap_stop_advertising 函数停止广播，广播停止后，协议栈会上传 GAP EVENT：GAP_EVT_ADV_END。如果用户希望更新广播参数，则需要调用停止广播函数，在广播停止的事件后面

重新调用 gap_set_advertising_data、gap_set_advertising_data 和 gap_set_advertising_rsp_data 配置广播参数，设置广播数据和扫描回复数据，然后再调用 gap_start_advertising 函数开启广播。

4.1.4 开启传统扫描流程

本节介绍在设备作为中央设备时开启传统扫描的工作流程，用户程序下发开启扫描指令，参数为扫描参数，在接收到广播信息后协议栈库会向上产生 GAP_EVT_ADV_REPORT 事件，如下图所示：



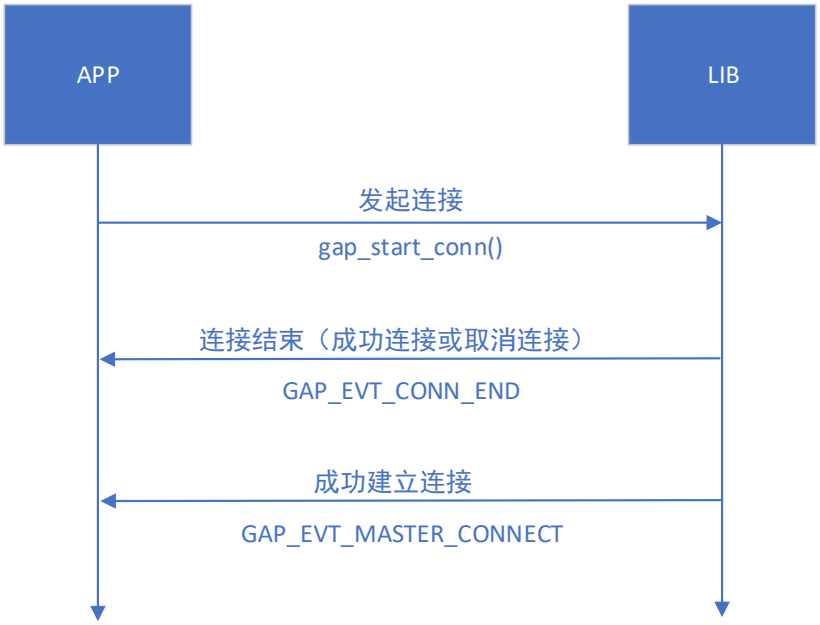
1. 设置扫描参数并开启扫描

```
gap_scan_param_t scan_param;
// 设置普通扫描模式
scan_param.scan_mode = GAP_SCAN_MODE_GEN_DISC;
scan_param.dup_filt_pol = 0;
// 设置扫描间隔，单位 625us
scan_param.scan_intv = 32;
// 设置扫描窗口，单位 625us
scan_param.scan_window = 20;
// 设置扫描持续时间
scan_param.duration = 1500;
gap_start_scan(&scan_param);
```

2. 停止扫描：调用函数 gap_stop_scan 可以停止正在进行的扫描。扫描停止后，协议栈会上传 GAP EVENT：GAP_EVT_SCAN_END。

4.1.5 发起传统连接流程

本节介绍在设备作为中央设备时发起连接的流程。通过扫描获取对方的地址和地址类型后，就可以发起对该设备的连接，应用层与 BLE Stack 之间的交互流程如下图所示：



成功建立连接时应用层会受到 GAP_EVT_CONN_END 和 GAP_EVT_MASTER_CONNECT 事件，用户可以调用 gap_stop_conn 取消当前的连接操作，协议栈库会产生超时事件 GAP_EVT_CONN_END。建立连接采用的函数为：

```
1.  void gap_start_conn(mac_addr_t *addr, // 从机地址
2.                          uint8_t addr_type, // 从机地址类型
3.                          uint16_t min_itvl, // 最小连接间隔，单位为 1.25ms
4.                          uint16_t max_itvl, // 最大连接间隔，单位为 1.25ms
5.                          uint16_t slv_latency, // 从机延迟
6.                          uint16_t timeout) // 连接监听超时时间，单位为 10ms
```

4.1.6 注册接收 GAP 消息

用户通过函数 gap_set_cb_func 向协议栈库注册用于接收 GAP 事件的回调函数，用户程序需要在协议栈初始化好之后调用该函数进行注册：

```
1.  gap_set_cb_func(app_gap_evt_cb);
```

其中回调函数举例如下：

```
1.  void app_gap_evt_cb(gap_event_t *p_event)
2.  {
3.      switch(p_event->type)
4.      {
```

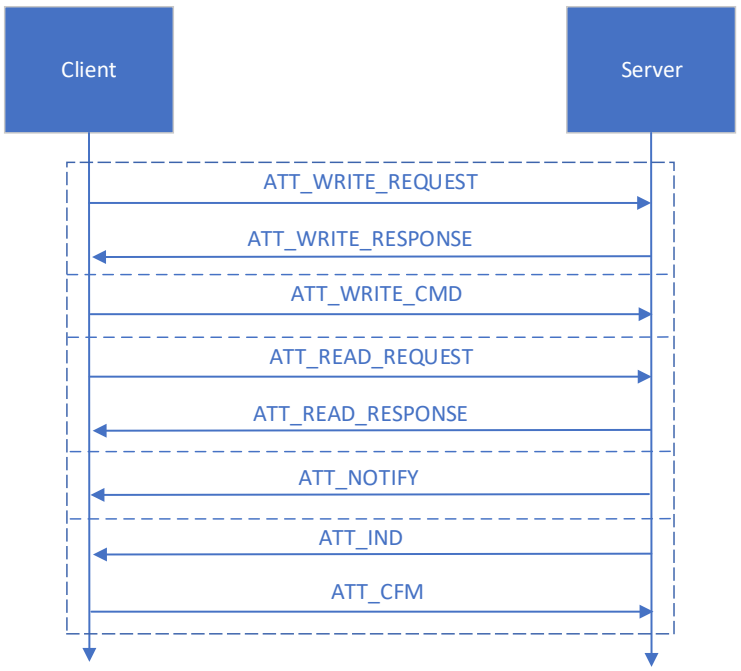
```
        // 扫描到空中广播信息
        case GAP_EVT_ADV_END:
            break;
        // 所有 service 注册完成
        case GAP_EVT_ALL_SVC_ADDED:
            break;
        // 建立好一条连接
        case GAP_EVT_CONNECT:
            break;
        // 连接断开
        case GAP_EVT_DISCONNECT:
            break;
        // 连接参数更新
        case GAP_EVT_LINK_PARAM_UPDATE:
            break;
        ...
    }
}
```

4.2 通用属性规范（GATT）

通用属性规范 GATT（Generic Attribute Profile）用于两个连接设备之间的数据通信。在低功耗蓝牙 GATT 层中，数据以对特征（Characteristic）进行读写的形式进行传输。

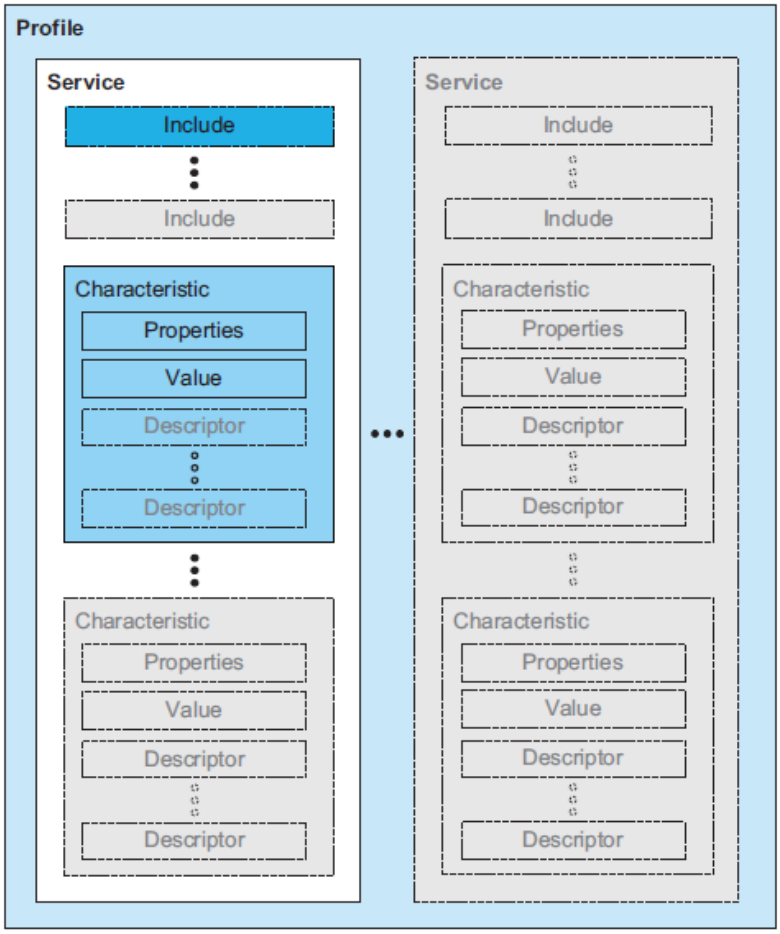
4.2.1 GATT 角色

定义服务和特征的一端称为服务端，接收来自客户端的读写操作，服务端通过通知和指示的方式向客户端发送数据。GATT 服务端和 GATT 客户端与设备角色没有直接关系，但是一般情况下外围设备为服务端，中央设备为客户端。常用的 GATT 操作模式如下图所示：



4.2.2 GATT Profile 层级

一个设备内包含一个或多个 profile，一个 profile 可能会包含一个或多个 service，一个 service 由一个或多个特征。GATT 规定了交换 profile 文件数据的层级结构，如图所示：



4.2.2.1 属性（Attribute）

在低功耗蓝牙中，特征可以看作是一组属性信息的集合，包括特征声明、特征值以及特征描述符。实际上，在低功耗蓝牙 GATT 层中，数据以属性的形式进行交互。属性主要包括以下几个部分：

- 属性句柄：句柄是属性在 GATT 属性表中的索引，每个属性都有唯一的句柄，该句柄为 16 比特数值，对特征值的读写等操作都采用该句柄作为标识；
- 属性类型：表示数据代表的事物，通常是 Bluetooth SIG 规定或由用户自定义的 UUID（通用唯一标识符）；
- 属性值：属性的数据值；
- 属性权限：规定了访问属性所需要的权限。

4.2.2.2 特征（Characteristic）

一个典型的特征由以下属性组成：

- 特征声明：描述特征的性质、存储位置（句柄）、类型；
- 特征值：特征的数据值；
- 特征描述符：描述特征的附加信息或配置。

4.2.2.3 服务（Service）

GATT 服务是一系列特征的集合，例如，心率服务包括心率测量特征和身体位置特征等。一个 profile 包含一个或多个服务。

下面列举一些常见的服务，这两个服务在 SDK 中默认存在：

- 通用访问服务（Generic Access Service）：该服务包括设备及访问信息等，比如设备名称、供应商标志、产品标志等。该服务定义的特征有：设备名称（Device Name）、外观（Appearance）、外设优先连接参数（Peripheral Preferred Connection Parameters）等；
- 通用属性服务（Generic Attribute Service）：该服务主要用于服务端通知所连接的客户端其提供的服务发生了变化。该服务包含 Service changed 特征。

4.2.3 服务端

4.2.3.1 添加 profile

添加 profile 就是创建服务的过程，用户程序需要在协议栈初始化好之后进行服务的添加注册，在所有的服务添加完成之后，协议栈库会产生 GAP_EVT_ALL_SVC_ADDED 消息，用户程序可以在这里进行开启广播等后续操作。添加服务方法如下：

```
1.  // 特征值定义表
2.  simple_profile_svc.p_att_tb = simple_profile_att_table;
3.  // 特征值个数
4.  simple_profile_svc.att_nb = SP_IDX_NB;
5.  // GATT 读写等事件的回调函数
6.  simple_profile_svc.gatt_msg_handler = sp_gatt_msg_handler;
7.
8.  // 向协议栈库注册添加服务
9.  sp_svc_id = gatt_add_service(&simple_profile_svc);
```

特征值表如何定义，下面是一个范例，包含了普通可读写属性、普通可通知属性、需要鉴权才可访问的属性、具有 128 bits-UUID 的属性等。

```
1.  const gatt_attribute_t simple_profile_att_table[SP_IDX_NB] =
2.  {
3.      // 定义一个 service，UUID 存放在 sp_svc_uuid 中
4.      [SP_IDX_SERVICE]                    =   {
5.                  { UUID_SIZE_2, UUID16_ARR(GATT_PRIMARY_SERVICE_UUID) },
6.                  GATT_PROP_READ,
7.                  UUID_SIZE_16,
8.                  (uint8_t*)sp_svc_uuid,
9.              },
10.     // 定义一个特征
11.     [SP_IDX_CHAR1_DECLARATION]          =   {
```

```
                { UUID_SIZE_2, UUID16_ARR(GATT_CHARACTER_UUID) },
                GATT_PROP_READ,
                0,
                NULL,
            },
    // 定义特征的值，属性为可读可写，该属性具有 128bits UUID，数据长度为 SP_CHAR1_VALUE_LEN
    [SP_IDX_CHAR1_VALUE]                =   {
                { UUID_SIZE_16, (SP_CHAR1_UUID) },
                GATT_PROP_READ | GATT_PROP_WRITE,
                SP_CHAR1_VALUE_LEN,
                NULL,
            },
    // 定义特征的描述，只读属性
    [SP_IDX_CHAR1_USER_DESCRIPTION]     =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CHAR_USER_DESC_UUID) },
                GATT_PROP_READ,
                SP_CHAR1_DESC_LEN,
                (uint8_t *)sp_char1_desc,
            },
    // 定义一个特征
    [SP_IDX_CHAR2_DECLARATION]          =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CHARACTER_UUID) },
                GATT_PROP_READ,
                0,
                NULL,
            },
    // 定义特征的值，属性为通知类型，数据长度为 SP_CHAR2_VALUE_LEN
    [SP_IDX_CHAR2_VALUE]                =   {
                { UUID_SIZE_16, (SP_CHAR2_UUID) },
                GATT_PROP_NOTI,
                SP_CHAR2_VALUE_LEN,
                NULL,
            },
    // 定义特征的配置属性，用于开启和关闭通知
    [SP_IDX_CHAR2_CFG]                  =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CLIENT_CHAR_CFG_UUID) },
                GATT_PROP_READ | GATT_PROP_WRITE,
                SP_CHAR2_CCC_LEN,
                NULL,
            },
    // 定义特征的描述，只读属性
    [SP_IDX_CHAR2_USER_DESCRIPTION]     =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CHAR_USER_DESC_UUID) },
```

```
                GATT_PROP_READ,
                SP_CHAR2_DESC_LEN,
                (uint8_t *)sp_char2_desc,
            },
    // 定义一个特征
    [SP_IDX_CHAR3_DECLARATION]          =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CHARACTER_UUID) },
                GATT_PROP_READ,
                0,
                NULL,
            },
    // 定义特征的值，属性为可读可写，但需要鉴权后才可访问，数据长度为 SP_CHAR3_VALUE_LEN
    [SP_IDX_CHAR3_VALUE]                =   {
                { UUID_SIZE_16, (SP_CHAR3_UUID) },
                GATT_PROP_AUTHEN_WRITE | GATT_PROP_AUTHEN_READ,
                SP_CHAR5_VALUE_LEN,
                NULL,
            },
    // 定义特征的描述，只读属性
    [SP_IDX_CHAR3_USER_DESCRIPTION]     =   {
                { UUID_SIZE_2, UUID16_ARR(GATT_CHAR_USER_DESC_UUID) },
                GATT_PROP_READ,
                SP_CHAR3_DESC_LEN,
                (uint8_t *)sp_char3_desc,
            },
};
```

GATT 消息回调函数用于接收协议栈库向上返回的读、写、通知、完成操作等事件，下面针对这些事件给出一个范例：

```
static uint16_t sp_gatt_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        // 收到了对方的读请求
        case GATTC_MSG_READ_REQ:
            switch (p_msg->att_idx)
            {
                // 将数据直接填写到 p_msg->param.msg.p_msg_data 指向的地址中，该存储空间协议栈库
                // 在调用该回调时已经按照定义的 SP_CHAR_VALUE_LEN 分配好
                case SP_IDX_CHAR1_VALUE:
```

```
12.                     memcpy(p_msg->param.msg.p_msg_data, sp_char1_value, SP_CHAR1_VALUE_LEN)
    ;
13.                     p_msg->param.msg.msg_len = SP_CHAR1_VALUE_LEN;
14.                     break;
15.                 case SP_IDX_CHAR2_CFG:
16.                     p_msg->param.msg.msg_len = 2;
17.                     memcpy(p_msg->param.msg.p_msg_data, sp_char4_ccc, 2);
18.                     break;
19.                 case SP_IDX_CHAR3_VALUE:
20.                     memcpy(p_msg->param.msg.p_msg_data, sp_char5_value, SP_CHAR5_VALUE_LEN)
    ;
21.                     p_msg->param.msg.msg_len = SP_CHAR3_VALUE_LEN;
22.                     break;
23.                 default:
24.                     break;
25.             }
26.             break;
27.         // 收到了对方的写请求
28.         case GATTC_MSG_WRITE_REQ:
29.             switch (p_msg->att_idx)
30.             {
31.                 // 收到的数据存储在 p_msg->param.msg.p_msg_data 指向的空间中，
32.                 // 长度为 p_msg->param.msg.msg_len
33.                 case SP_IDX_CHAR1_VALUE:
34.                     break;
35.                 case SP_IDX_CHAR2_CFG:
36.                     break;
37.                 case SP_IDX_CHAR3_VALUE:
38.                     break;
39.                 default:
40.                     break;
41.             }
42.             break;
43.         // 建立好连接
44.         case GATTC_MSG_LINK_CREATE:
45.             break;
46.         // 连接断开
47.         case GATTC_MSG_LINK_LOST:
48.             break;
49.         default:
50.             break;
51.     }
52.     return p_msg->param.msg.msg_len;
```

```
53. }
```

4.2.3.2 发出通知消息

以 SP_CHAR2_UUID 为例介绍如何发出通知消息，通知消息需要在客户端开启通知时才允许进行发送操作：

```
1.  gatt_ntf_t ntf_att;
2.
3.  ntf_att.att_idx = SP_IDX_CHAR2_VALUE;
4.  // 链路索引
5.  ntf_att.conidx = conn_idx;
6.  // 服务 ID，也就是添加服务时的返回值
7.  ntf_att.svc_id = svc_id;
8.  // 通知信息长度
9.  ntf_att.data_len = 4;
10. uint8_t tmp[] = "12345";
11. // 通知信息存放指针
12. ntf_att.p_data = tmp;
13. gatt_notification(ntf_att);
```

4.2.4 客户端

4.2.4.1 客户端的注册

客户端一般存在于主机中，主机连接好从机后需要搜索从机中的特征值后，才可以基于搜索结果进行后续的操作。在 SDK 中提供了一种搜索用户所关注的特征值的方法，用户程序在初始化时把关注的特征值注册到协议栈库中，用户程序在连接成功后调用函数 gatt_discovery_all_peer_svc 发起搜索服务的指令，协议栈库会自动执行后续操作，把用户关注的特征值对应的属性句柄通过回调函数反馈上来。初始化时采用下面的方法进行注册：

```
1.  // 用户程序关注的特征值列表
2.  client.p_att_tb = client_att_tb;
3.  client.att_nb = 2;
4.  // 添加回调函数
5.  client.gatt_msg_handler = simple_central_msg_handler;
6.  client_id = gatt_add_client(&client);
```

用户程序关注的特征值指的是客户端会进行读写、接收通知的特征值，这个列表举例如下：

```
1.  const gatt_uuid_t client_att_tb[] =
2.  {
3.      [0]  =
4.      { UUID_SIZE_16, (SP_CHAR1_UUID)},
```

```
    [1]  =
    { UUID_SIZE_16, (SP_CHAR2_UUID)},
    [2]  =
    { UUID_SIZE_16, (SP_CHAR3_UUID)},
};
```

Gatt_msg_handler 用于接收协议栈库上传的各种事件，举例如下：

```
static uint16_t simple_central_msg_handler(gatt_msg_t *p_msg)
{
    switch(p_msg->msg_evt)
    {
        // 接收到了对端的通知消息
        case GATTC_MSG_NTF_REQ:
            {
                // 消息中的 att_idx 是用户程序注册的特征值列表索引，1 对应的是 SP_CHAR2_UUID
                if(p_msg->att_idx == 1)
                {
                    show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);
                }
            }
            break;
        // 在发起读操作后获得对方的回复
        case GATTC_MSG_READ_IND:
            {
                // 对方对 SP_CHAR1_UUID 特征值读操作的返回消息
                if(p_msg->att_idx == 0)
                {
                    show_reg(p_msg->param.msg.p_msg_data,p_msg->param.msg.msg_len,1);
                }
            }
            break;
        // 协议栈库完成 GATT 操作后的回调消息
        case GATTC_MSG_CMP_EVT:
            {
                // 完成了特征值扫描过程
                if(p_msg->param.op.operation == GATT_OP_PEER_SVC_REGISTERED)
                {
                    // p_msg->param.op.arg 中包含的就是特征值列表包含的特征值对应的属性句柄
                    // 如果没有搜索到特征值，对应的位置则为 0（协议中规定句柄从 1 开始）。
                    uint16_t att_handles[3];
                    memcpy(att_handles,p_msg->param.op.arg,6);
                    show_reg((uint8_t *)att_handles,6,1);
```

```
                }
            }
            break;

        default:
            break;
    }

    return 0;
}
```

4.2.4.2 使能服务端通知

在搜索完成服务端的特征后，客户端程序根据需要采用如下函数开启或关闭服务端的通知功能：

```
// 开启对端的通知功能，注意 att_idx 参数填写的是特征值列表索引
gatt_client_enable_ntf_t ntf_enable;
ntf_enable.conidx = p_msg->conn_idx;
ntf_enable.client_id = client_id;
ntf_enable.att_idx = 1;
gatt_client_enable_ntf(ntf_enable);
```

4.2.4.3 发起读请求

```
// 发起读操作，注意 att_idx 参数填写的是特征值列表索引
gatt_client_read_t read;

// 链接索引
read.conidx = conn_idx;
// 客户端 ID，也就是注册客户端时的返回值
read.client_id = client_id;
// 特征值列表的索引，0 表示上面的 SP_CHAR1_UUID
read.att_idx = 0;
gatt_client_read(read);
```

4.2.4.4 发起写请求

写请求一般有两种，一种需要对方返回 response，一种不需要。协议栈库中提供了两个函数用来实现这两种写操作：

```
gatt_client_write_t write;
```

```
2.
3.  // 链接索引
4.  write.conidx = conn_idx;
5.  // 客户端索引
6.  write.client_id = client_id;
7.  // 表示对 SP_CHAR1_UUID 发起写操作
8.  write.att_idx = 0;
9.  // 发送数据的指针
10. write.p_data = "\x1\x2\x3\x4\x5\x6\x7";     //Buffer to be written
11. // 发送数据的长度
12. write.data_len = 7;             //buffer len
13.
14. // 采用 ATT_WRITE_CMD 发送，不需要对方的 response
15. gatt_client_write_cmd(write);       //start write opertaion
16. // 采用 ATT_WRITE_REQ 发送，需要对方的 response
17. gatt_client_write_req(write);       //start write opertaion
```

5. 操作系统抽象层（OSAL）

FR5080 支持用户创建自己的任务，创建软件定时器和动态分配释放内存。注意在 FR5080 中，任务不区分优先级并且只允许创建一个用户任务，用户任务主要用来接收和处理来自中断中推送出的消息。下面章节将介绍任务创建，消息推送，任务处理，定时器应用及动态内存分配的详细用法。具体 API 请参考 components\modules\os 目录。

5.1 用户任务系统

5.1.1 OS Task Create

uint16_t os_task_create(os_task_func_t task_func)

创建一个任务（仅支持创建一个），消息按抛送的顺序进行处理。

参数：

task_func	-	任务的执行函数。

返回：

uint16_t	-	创建任务的 id 号， 0xff，任务创建失败。 其他值，任务创建成功，返回值是任务的 id 号。

5.1.2 OS Task Delete

void os_task_delete(uint16_t task_id)

删除一个已经创建的任务。

参数：

task_id	-	创建任务时返回的任务 id 号。

返回：

None

5.1.3 OS Message Post

5.1.3.1 消息类型

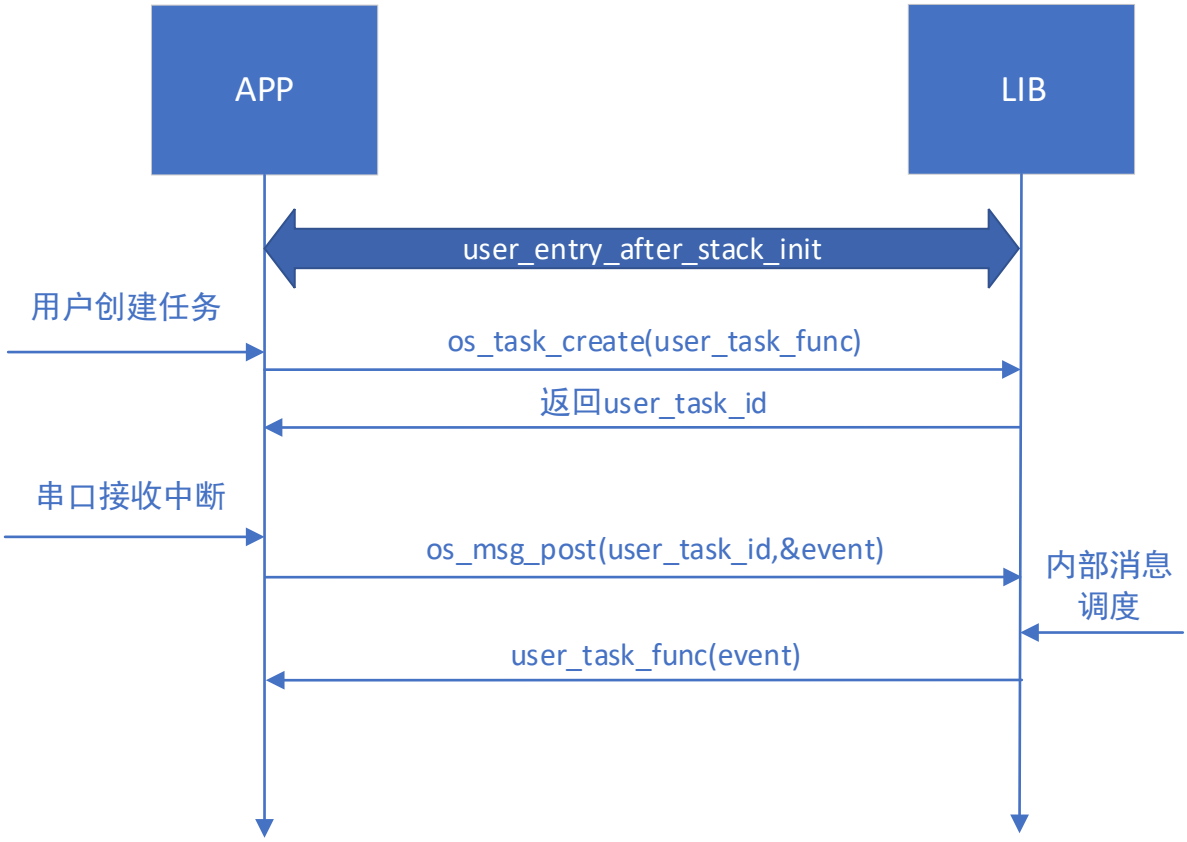
```
typedef struct _os_event_
{
    uint16_t event_id;      //事件序号，用户自定义
    uint16_t src_task_id;   //源端任务 ID，保留不用
    void *param;            //参数指针
    uint16_t param_len;     //参数长度
} os_event_t;
```

5.1.3.2 OS Msg Post

void os_msg_post(uint16_t dst_task_id,os_event_t *evt)		
向某个已经创建的目标任务抛一个消息事件。对于用户而言，dst_task_id 即为用户创建的任务 ID。		
参数：		
dst_task_id	-	目标任务的任务 id 号
evt	-	指向要抛送的消息事件的指针。
返回：		
None		

5.1.4 串口消息处理任务示例

参考\examples\none_evm\btdm_demo，该示例中创建了一个用户任务，用来接收 uart 中断接收到消息，并解析。注意用户任务创建需要在 user_entry_after_stack_init 入口处。



具体代码实现如下：
用户创建任务，下面代码定义了用户消息处理函数及用户任务初始化，在 user_entry_after_stack_init 中调用 user_task_init 即可。

```
1.  uint16_t user_task_id;
2.
```

```
///用户消息处理函数
static int user_task_func(os_event_t *param)
{
    switch(param->event_id)
    {
        ///事件类型为串口 AT 命令
        case USER_EVT_AT_COMMAND:
            ///消息处理
            app_at_cmd_recv_handler(param->param, param->param_len);
            break;
    }

    ///返回 EVT_CONSUMED，LIB 对该消息进行释放
    return EVT_CONSUMED;
}

void user_task_init(void)
{
    ///创建用户任务
    user_task_id = os_task_create(user_task_func);
}
```

在串口中断处理函数中将消息抛送出来。

```
os_event_t at_cmd_event;
at_cmd_event.event_id = USER_EVT_AT_COMMAND;///event id
at_cmd_event.param = at_recv_buffer;        ///串口接收 buffer
at_cmd_event.param_len = at_recv_index;     ///串口接收 buffer 长度
os_msg_post(user_task_id, &at_cmd_event);   ///发送消息
```

之后 user_task_func 将会收到该消息，并处理。

5.2 软件定时器

FR5080 内部有软件定时器，可以最多支持 40 个定时器，注意该定时器易被其他事件打断，导致定时时间有偏差，用于对时间精准度不高的应用。

5.2.1 OS Timer Initialization

void os_timer_init(os_timer_t *ptimer, os_timer_func_t pfunction, void *parg)
初始化一个软件定时器。使用软件定时器之前，必须调用该函数进行初始化。 **参数：**

ptimer	-	指向软件定时器结构体的指针。
pfunction	-	定时器的执行函数。
parg	-	定时器执行函数的输入参数指针。

返回：

None

5.2.2 OS Timer Start

void os_timer_start(os_timer_t *ptimer,uint32_t ms, bool repeat_flag)

启动一个软件定时器。

参数：

ptimer	-	指向软件定时器结构体的指针。
ms	-	定时时间，单位:ms。取值范围，10 ~ 0x3FFFFF
repeat_flag	-	定时器是否重复。

返回：

None

5.2.3 OS Timer Stop

void os_timer_stop(os_timer_t *ptimer)

停止一个软件定时器。

参数：

ptimer	-	指向软件定时器结构体的指针。

返回：

None

5.2.4 软件定时器使用示例

在 user_entry_after_stack_init 之后，可以启用 timer

```
1.  os_timer_init(&test_timer, test_timer_func, NULL);  ///timer 初始化
2.  os_timer_start(&test_timer, 2200, true);            ///2.2s 间隔，重复运行
```

test_timer 和 test_timer_func 定义如下

```
1.  os_timer_t test_timer;          ///用户 timer
2.  ///timer 回调函数
3.  void test_timer_func(void *arg)
4.  {
5.      uart_putc_noint('T');
6.  }
```

5.3 内存分配

5.3.1 OS Malloc

void *os_malloc(uint32_t size)

向系统 heap 申请分配一段内存。

参数：

size	-	要申请的内存的大小。

返回：

void *	-	指向分配内存地址的指针。

5.3.2 OS Free

void os_free(void *ptr)

释放已分配的一段内存。

参数：

ptr	-	指向已分配内存地址的指针

返回：

None	-	

5.3.3 OS Get Free Heap Size

uint16_t os_get_free_heap_size(void)

获取系统 heap 剩余的空间大小。

参数：

None

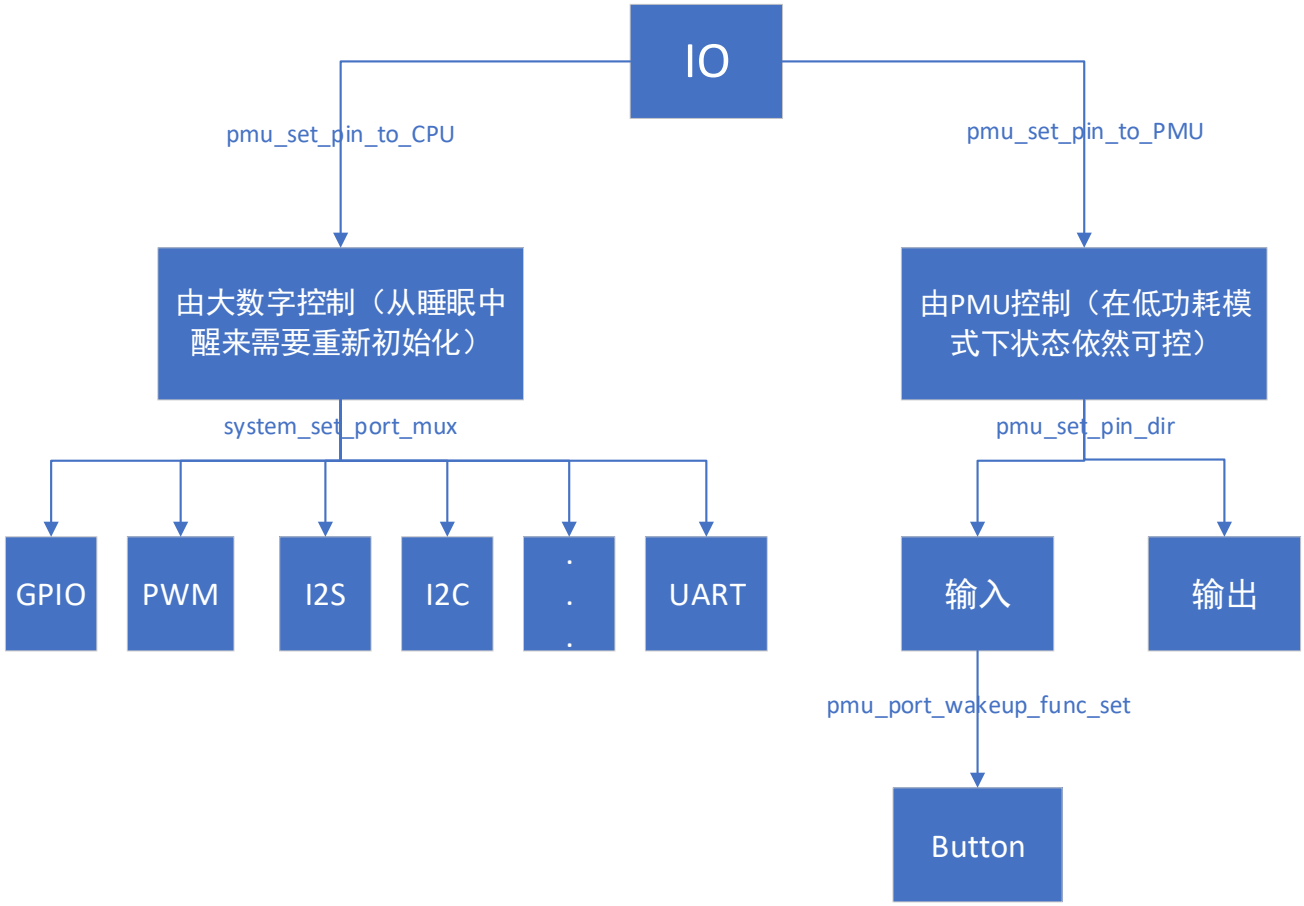
返回：

uint16_t	系统剩余的 heap 空间。

6. MCU 外设驱动

6.1 IO MUX

FR5080 系列芯片共有 4 组 IO，不同型号的芯片引出的 IO 数量有所不同。每个 IO 可配置为上拉模式，上拉电阻约为 50K 欧姆。IO 的工作状态和模式可选择由大数字（进入低功耗后断电）或者 PMU（进入低功耗模式后继续工作）控制。



6.1.1 普通 IO 接口

以下 API 位于 components\driver\include\driver_syscntl.h 中。

6.1.1.1 IO 功能设置

void system_set_port_mux(enum system_port_t port, enum system_port_bit_t bit, uint8_t func)	
设置 IO 由大数字控制时的功能，单次设置一个 IO	
参数：	
port	IO 所属的端口组
bit	IO 的 channel 编号

func	所要设置的功能

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_0, PORTA0_FUNC_UART0_RXD);
```

6.1.1.2 IO 上拉设置

void system_set_port_pull_up (uint32_t port, uint8_t pull)

设置 IO 由大数字控制时的上拉功能，可以一次设置多个 IO

参数：

port	IO 所属的端口组
pull	选择是否开启上拉

返回：

None

示例：

```
system_set_port_pull_up ((GPIO_PA0 | GPIO_PA1), true);//PA0 PA1 设置为上拉
```

6.1.1.3 IO 下拉设置

void system_set_port_pull_down (uint32_t port, uint8_t pull)

设置 IO 由大数字控制时的上拉功能，可以一次设置多个 IO

参数：

port	IO 所属的端口组
pull	选择是否开启下拉

返回：

None

示例：

```
system_set_port_pull_down( (GPIO_PC6|GPIO_PC7), true); //PC6 PC7 设置为下拉
```

6.1.2 支持低功耗模式的 IO 接口

以下 API 位于 driver_pmu.h 中，都是在低功耗模式下的 IO 操作。

6.1.2.1 IO 使能低功耗模式

void pmu_set_pin_to_PMU(enum system_port_t port, uint8_t bits)

将某个 pin 脚配置给 pmu 控制。调用 pmu_gpio 函数前，需要首先调用该函数将对应管脚给 pmu 控制。

参数：

port	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

返回：

None

示例：

```
//Select PA0,PA1 to be controlled by PMU
pmu_set_pin_to_PMU(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1);
```

6.1.2.2 IO 关闭低功耗模式

void pmu_set_pin_to_CPU(enum system_port_t port, uint8_t bits)

将某个 pin 脚配置给 CPU 控制。所有管脚默认是被 CPU 控制的。如果希望配置给 PMU 控制的管脚，被 CPU 的外设来控制，需要首先调用该函数将对应管脚给 CPU 控制。

参数：

port	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

返回：

None

示例：

```
//Select PA0,PA1 to be controlled by CPU
pmu_set_pin_to_CPU(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1);
```

6.1.2.3 选择输出的控制方式

void pmu_set_gpio_output_select (enum system_port_t port, uint8_t bits, enum port_output_sel sel)

某个 pin 脚输出来源选择。Led 或者 gpio

参数：

port	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。
sle	选择控制方式：SEL_LED_CTL: LED 控制, SEL_PMU_REG: PMU 寄存器控制。参见 enum port_output_sel 定义。

返回：

None

示例：

```
//Select PA4,PA5 to be controlled by PMU reg
pmu_set_gpio_output_select(GPIO_PORT_A,BIT(4)|BIT(5), SEL_PMU_REG);
```

6.1.2.4 IO 低功耗模式输入输出设置

void pmu_set_pin_dir(enum system_port_t port, uint8_t bits, uint8_t dir)

配置某个 pin 脚 pmu 控制时的输入输出选择。

参数：

port　　选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits　　选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

dir　　选择 pin 脚对应的输入/输出。只能填以下二值：GPIO_DIR_OUT，表示该 pin 为输出。GPIO_DIR_IN，表示该 pin 为输入。

返回：

None

示例：

```
// configure PA0~PA1 as output
pmu_set_pin_dir(GPIO_PORT_A,BIT(0)|BIT(1), GPIO_DIR_OUT);
```

6.1.2.5 IO 低功耗模式上拉设置

void pmu_set_pin_pull_up (enum system_port_t port, uint8_t bits, bool flag)

配置某个 pin 脚 pmu 控制时是否内部上拉。

参数：

port　　选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits　　选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

flag　　选择 pin 脚是否内部上拉。True，表示该 pin 内部上拉。False，表示该 pin 没有内部上拉。

返回：

None

示例：

```
pmu_set_pin_pull(GPIO_PORT_A, GPIO_PA0 | GPIO_PA1, true);//配置  PA0 PA1 为上拉
```

6.1.2.6 IO 低功耗模式下拉设置

void pmu_set_pin_pull_down (enum system_port_t port, uint8_t bits, bool flag)

配置某个 pin 脚 pmu 控制时是否内部下拉。

参数：

port　　选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。

bits　　选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。

flag　　选择 pin 脚是否内部上拉。True，表示该 pin 内部下拉。False，表示该 pin 没有内部下拉。

返回：

None

示例：

```
pmu_set_pin_pull_down (GPIO_PORT_A, GPIO_PA0 | GPIO_PA1, true); //配置  PA0 PA1 为下拉
```

6.1.2.7 IO 使能低功耗唤醒

void pmu_port_wakeup_func_set(uint32_t gpios)

设置 PMU 中对 IO 的状态监控功能，该函数内部完成了选择 PMU 控制、IO MUX 选择、设置为输入模式。可以一次设置多个 IO。当被监测的 IO 高低电平发生变化时就可产生 pmu gpio monitor 中断，如果在睡眠状态下发生变化则先产生唤醒信号，同时产生中断。

参数：

gpios	IO 对应的编号

返回：

None

示例：

```
pmu_port_wakeup_func_set(GPIO_PA0|GPIO_PA1);
```

6.1.2.8 IO 低功耗模式中断入口

__attribute__((weak)) void pmu_gpio_isr_ram(void)

pmu_gpio 中断 weak 函数。用于需要重定义来获取中断的入口

参数：

None

返回：

None

示例：

```
void pmu_gpio_isr_ram(void)
{
    uint32_t pmu_int_pin_setting = ool_read32(PMU_REG_PORTA_TRIG_MASK);
    uint32_t gpio_value = ool_read32(PMU_REG_GPIOA_V);

    ool_write32(PMU_REG_PORTA_LAST, gpio_value);
    uint32_t tmp = gpio_value & pmu_int_pin_setting;
    uint32_t pressed_key = tmp^pmu_int_pin_setting;
    co_printf("K:0x%08x\r\n", (pressed_key);
}

void user_entry_after_ble_init(void)
{
    pmu_port_wakeup_func_set(GPIO_PD5|GPIO_PD4|GPIO_PD3);
}
```

6.2 GPIO

位于 components\driver\include\driver_gpio.h。

6.2.1 普通 GPIO 接口

6.2.1.1 GPIO 输出

void gpio_portX_write(uint8_t value)	
设置一组 IO 由大数字控制时的输出值，x 为 a、b、c、d	
参数：	
value	IO 的输出值
返回：	
None	
示例：	
gpio_porta_write(0xFF);	

6.2.1.2 GPIO 获取当前值

uint8_t gpio_portX_read(void)	
获取一组 IO 由大数字控制时的当前值，x 为 a、b、c、d	
参数：	
None	
返回：	
uint8_t	IO 的当前值

6.2.1.3 GPIO 设置整个 port 输入输出

void gpio_portX_set_dir(uint8_t dir)	
设置一组 IO 由大数字控制时的输入输出，x 为 a、b、c、d	
参数：	
dir	输入输出，每一位对应一个 IO，0：输出；1：输入
返回：	
None	

6.2.1.4 GPIO 获取整个 port 输入输出配置

uint8_t gpio_portX_get_dir(void)
获取一组 IO 由大数字控制时的输入输出设置，x 为 a、b、c、d
参数：
None

返回：

uint8_t	当前的输入输出配置

6.2.1.5 GPIO 设置单个 IO 输入输出

void gpio_set_dir(enum system_port_t port, enum system_port_bit_t bit, uint8_t dir)

设置 IO 由大数字控制时的输入输出，一次设置一个 IO

参数：

port	IO 所属的端口组
bit	IO 的 channel 编号
dir	输入或者输出

返回：

None

示例：

```
gpio_set_dir(GPIO_PORT_A, GPIO_BIT_0, GPIO_DIR_OUT);
```

6.2.1.6 GPIO 设置单个 IO 输出状态

void gpio_set_pin_value (enum system_port_t port, enum system_port_bit_t bit, uint8_t value)

设置 IO 由大数字控制时的输入输出，一次设置一个 IO

参数：

port	IO 所属的端口组
bit	IO 的 channel 编号
value	0：输出低电平，1：输出高电平

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_C,GPIO_BIT_4,PORTC4_FUNC_C4);
system_set_port_mux(GPIO_PORT_C,GPIO_BIT_5,PORTC5_FUNC_C5);
gpio_set_dir(GPIO_PORT_C, GPIO_BIT_4, GPIO_DIR_OUT);
gpio_set_dir(GPIO_PORT_C, GPIO_BIT_5, GPIO_DIR_OUT);

gpio_set_pin_value(GPIO_PORT_C,GPIO_BIT_4,1);//PC4 输出高电平
gpio_set_pin_value(GPIO_PORT_C,GPIO_BIT_5,0);//PC5 输出低电平
```

6.2.2 低功耗模式 GPIO 接口

6.2.2.1 GPIO 低功耗模式输出值

void pmu_set_gpio_value(enum system_port_t port, uint8_t bits, uint8_t value)

当某个 pin 脚被配置为 pmu gpio 控制，并且是输出模式时，设置该 pin 脚的值。

参数：

port	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bits	选择 pin 脚对应的 pin 号码，bit7~bit0 分别代表每个 port 口的 pin7~pin0。每个 bit 位表示该 pin 被选中。
Value	设置 pin 脚输出值。只能填以下二值：1，该 pin 输出为高。0，该 pin 输出为低。

返回：

None

示例：

```
void pmu_gpio_test(void)
{
    pmu_set_pin_to_PMU(GPIO_PORT_A,BIT(0)|BIT(1) );
    pmu_set_pin_dir(GPIO_PORT_A,BIT(0)|BIT(1), GPIO_DIR_OUT);
    pmu_set_pin_pull_up (GPIO_PORT_A, BIT(0)|BIT(1), true);
  pmu_set_gpio_output_select(GPIO_PORT_A,BIT(0)|BIT(1), SEL_PMU_REG);

  pmu_set_gpio_value(GPIO_PORT_A, BIT(0)|BIT(1), 1);
  co_delay_100us(10);
  pmu_set_gpio_value(GPIO_PORT_A, BIT(0)|BIT(1), 0);
}
```

6.2.2.2 GPIO 低功耗模式输入值

uint8_t pmu_get_gpio_value(enum system_port_t port, uint8_t bit)

当某个 pin 脚被配置为 pmu gpio 控制，并且是输入模式时，获取该 pin 脚的值。

参数：

port	选择 pin 脚对应的 port 口，一共有 4 个 port 口，PA，PB，PC，PD。参见 enum system_port_t 定义。
bit	选择 pin 脚对应的 pin 号码，参见 enum system_port_bit_t 定义

返回：

None

示例：

```
void pmu_gpio_test(void)
{
    pmu_set_pin_to_PMU(GPIO_PORT_A,BIT(2)|BIT(3) );
    pmu_set_pin_dir(GPIO_PORT_A,BIT(2)|BIT(3), GPIO_DIR_IN);
    co_printf("PA2:%d,PA3:%d\r\n",pmu_get_gpio_value(GPIO_PORT_A,GPIO_BIT_2)
                        ,pmu_get_gpio_value(GPIO_PORT_A,GPIO_BIT_3) );
}
```

6.3 UART

位于 components\driver\include\driver_uart.h。

6.3.1 UART 初始化

void uart_init(uint8_t bandrate)

初始化 UART 模块，函数内会清空 fifo，使能接收非空中断

参数：

bandrate　　配置的波特率，例如 BAUD_RATE_115200

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_B, GPIO_BIT_6, PORTB6_FUNC_CM3_UART_RXD);
system_set_port_mux(GPIO_PORT_B, GPIO_BIT_7, PORTB7_FUNC_CM3_UART_TXD);
uart_init(BAUD_RATE_115200);
NVIC_EnableIRQ(UART_IRQn);
```

6.3.2 UART 中断入口

__attribute__((section("ram_code"))) void uart_isr_ram(void)

参数：

None

返回：

None

示例：

```
__attribute__((section("ram_code"))) void uart_isr_ram(void)
{
    uint8_t int_id;
    uint8_t c;
    volatile struct uart_reg_t *uart_reg = (volatile struct uart_reg_t *)UART_BASE;
    int_id = uart_reg->u3.iir.int_id;
    if(int_id == 0x04 || int_id == 0x0c )    /* Receiver data available or Character time-out indication */
    {
        c = uart_reg->u1.data;
        uart_putc_noint_no_wait(c);
    }
    else if(int_id == 0x06)
    {
        uart_reg->u3.iir.int_id = int_id;
    }
}
```

6.3.3 从串口读取数据

void uart_read(uint8_t *buf, uint32_t size)	
采用阻塞方式从串口接收	
参数：	
buf	存放接收数据的指针
size	需要接收的数据长度
返回：	
None	
示例：	

6.3.4 从串口发送数据

void uart_write(const uint8_t *bufptr, uint32_t size)	
采用阻塞方式发送数据到串口	
参数：	
buf	待发送数据的指针
size	待发送的数据长度
返回：	
None	
示例：	

6.3.5 UART 发送一个字节且等待完成

void uart_putc_noint(uint8_t c)	
发送一个字符，且等待发送 fifo 为空	
参数：	
c	待发送的字符
返回：	
None	

6.3.6 UART 发送一个字节且立即返回

void uart_putc_noint_no_wait(uint8_t c)	
将待发送的字符写入到发送 fifo	
参数：	
c	待发送的字符
返回：	
None	

6.3.7 UART 发送多个字节且等待完成

void uart_put_data_noint(const uint8_t *d, int size)

发送特定长度的字符，且等待发送 fifo 为空

参数：

d	待发送数据的保存地址指针
size	待发送数据的长度

返回：

None

6.3.8 UART 读取特定个数字节

void uart_get_data_noint(uint8_t *buf, int size)

获取特定长度的字符

参数：

buf	待接收数据的保存地址指针
size	待接收数据的长度

返回：

None

6.3.9 UART 读取特定个数字节，诺 FIFO 为空则先返回

int uart_get_data_nodelay_noint(uint8_t *buf, int size)

获取特定长度的字符，当接收 fifo 为空时就会返回

参数：

buf	待接收数据的保存地址指针
size	待接收数据的长度

返回：

int	实际接收到的数据长度

6.4 SPI

位于 components\driver\include\driver_ssp.h。

6.4.1 SPI 初始化

void ssp_init(uint8_t bit_width, uint8_t frame_type, uint8_t ms, uint32_t bit_rate, uint8_t prescale, void (*ssp_cs_ctrl)(uint8_t))

初始化 SPI 模块

参数：

bit_width	总线上的数据位宽，取值为 1~8

frame_type	-	SPI 总线类型，可取值 SSP_FRAME_MOTO、SSP_FRAME_SS、SSP_FRAME_NATTIONAL_M
ms	-	主从模式选择，可取值 SSP_MASTER_MODE、SSP_SLAVE_MODE
bit_rate	-	需要配置的总线速率
prescale	-	基于系统时钟的模块分频比
ssp_cs_ctrl	-	自定义的 CS 控制函数

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_4, PORTA4_FUNC_SSP_SCLK);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_5, PORTA5_FUNC_SSP_CSN);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_6, PORTA6_FUNC_SSP_MOSI);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_7, PORTA7_FUNC_SSP_MISO);

ssp_init(8,SSP_FRAME_MOTO, SSP_MASTER_MODE,150000,2,NULL);
```

6.4.2 SPI 发送

void ssp_send_data(uint8_t *buffer, uint32_t length)

工作在主模式时，发送一定长度数据

参数：

buffer		待接收数据的保存地址指针
length	-	待接收数据的长度

返回：

None

示例：

```
ssp_send_data(dummy,32);
```

6.4.3 SPI 接收

void ssp_recv_data(uint8_t *buffer, uint32_t length)

工作在主模式时，接收一定长度数据

参数：

buffer		待接收数据的保存地址指针
length	-	待接收数据的长度

返回：

None

示例：

```
rec_num = ssp_get_data_from_fifo(buff, 16);
```

6.5 I2C

位于 components\driver\include\driver_iic.h。

6.5.1 I2C 初始化

void iic_init(enum iic_channel_t channel, uint8_t addr_type, uint16_t speed, uint16_t slave_addr)

初始化 IIC 模块，默认配置为为 7 位地址模式

参数：

channel	初始化对象，可选 IIC_CHANNEL_0、IIC_CHANNEL_1
addr_type	从机地址格式 1: 7-its, 0: 10-bits
speed	配置总线时钟速率为 speed*1000
slave_addr	当本机工作在从机模式时的从机地址

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_A,GPIO_BIT_0,PORTA0_FUNC_I2C0_SCL);
system_set_port_mux(GPIO_PORT_A,GPIO_BIT_1,PORTA1_FUNC_I2C0_SDA);
iic_init(IIC_CHANNEL_0,IIC_ADDR_7_BITS,100,0);
```

6.5.2 I2C 发送一个字节

uint8_t iic_write_byte(enum iic_channel_t channel, uint8_t addr_type, uint8_t slave_addr, uint8_t reg_addr, uint8_t data)

将一个字节数据发送给从机的特定地址

参数：

channel	操作对象
addr_type	从机地址格式 1: 7-its, 0: 10-bits
slave_addr	从机地址
reg_addr	操作的从机寄存器地址
data	待写入的数据

返回：

uint8_t	true：写入成功；false：写入失败

6.5.3 I2C 发送多个字节

uint8_t iic_write_bytes(enum iic_channel_t channel, uint8_t addr_type,uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer, uint16_t length)

将多个字节数据发送给从机的特定地址

参数：

channel	操作对象
addr_type	从机地址格式 1: 7-its, 0: 10-bits
slave_addr	从机地址
reg_addr	操作的从机寄存器起始地址
buffer	待写入的数据
length	待写入的数据长度

返回：

uint8_t	true：写入成功；false：写入失败

6.5.4 I2C 读取一个字节

uint8_t iic_read_byte(enum iic_channel_t channel, uint8_t addr_type,uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer)

从从机的特定地址读取一个字节数据

参数：

channel	操作对象
addr_type	从机地址格式 1: 7-its, 0: 10-bits
slave_addr	从机地址
reg_addr	操作的从机寄存器地址
buffer	读取数据的保存地址

返回：

uint8_t	true：读取成功；false：读取失败

6.5.5 I2C 读取多个字节

uint8_t iic_read_bytes(enum iic_channel_t channel, uint8_t addr_type,uint8_t slave_addr, uint8_t reg_addr, uint8_t *buffer, uint16_t length)

从从机的特定地址读取多个字节数据

参数：

channel	操作对象
addr_type	从机地址格式 1: 7-its, 0: 10-bits
slave_addr	从机地址
reg_addr	操作的从机寄存器起始地址
buffer	读取数据的保存地址
length	待读取的数据长度

返回：

uint8_t	true：读取成功；false：读取失败

6.6 Timer

位于 components\driver\include\driver_timer.h。

6.6.1 Timer 初始化

uint8_t timer_init(uint32_t timer_addr, uint32_t count_us, uint8_t run_mode)

初始化 timer 模块

参数：

timer_addr	初始化对象，可选 TIMER0、TIMER1
count_us	定时器周期
run_mode	单次模式还是周期模式

返回：

False：初始化失败

True：初始化成功

示例：

```
if(timer_init(TIMER0,300000,TIMER_PERIODIC)){
    printf("TIMER0 init ok\r\n");
}else{
    printf("TIMER0 init fail\r\n");
}
timer_run(TIMER0);
NVIC_EnableIRQ(TIMER0_IRQn);
```

6.6.2 Timer 启动

void timer_run(uint32_t timer_addr)

开启一个已经初始化好的 timer

参数：

timer_addr	操作对象

返回：

None

6.6.3 Timer 停止

void timer_stop(uint32_t timer_addr)

停止一个已经开启的 timer

参数：

timer_addr	操作对象

返回：

None

6.6.4 Timer 获取 load 值

uint32_t timer_get_load_value(uint32_t timer_addr)

获取当前 timer 的预设计时值

参数：

timer_addr	-	操作对象

返回：

uint32_t	-	timer 的预设值

6.6.5 Timer 获取当前计数值

uint32_t timer_get_current_value(uint32_t timer_addr)

获取当前 timer 的计数值

参数：

timer_addr	-	操作对象

返回：

uint32_t	-	当天计数值

6.6.6 Timer0 中断入口

__attribute__((section("ram_code"))) void timer0_isr_ram (void)

参数：

None

返回：

None

示例：

```
__attribute__((section("ram_code"))) void timer0_isr_ram(void)
{
    timer_clear_interrupt(TIMER0);

    printf("0\r\n");
}
```

6.6.7 Timer 清中断

void timer_clear_interrupt(uint32_t timer_addr)

清除中断标志位

参数：

timer_addr	操作对象

返回：

None

6.7 I2S

位于 components\driver\include\driver_timer.h。

6.7.1 I2S 初始化

void i2s_init_(uint8_t type, uint32_t sample_rate, uint8_t mode);

初始化 i2s 模块

参数：

type	I2s 格式：输入、输出， 参考 enum i2s_dir_t
sample_rate	i2s 采样率
mode	I2s 工作模式，参考 enum i2s_mode_t

返回：

None

示例：

```
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_4, PORTA4_FUNC_I2S_SCLK);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_5, PORTA5_FUNC_I2S_FRM);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_6, PORTA6_FUNC_I2S_MOSI);
system_set_port_mux(GPIO_PORT_A, GPIO_BIT_7, PORTA7_FUNC_I2S_MISO);

i2s_init_(I2S_DIR_TX,8000,1);
i2s_init_(I2S_DIR_RX,8000,1);
NVIC_EnableIRQ(I2S_IRQn);
i2s_start_();
```

6.7.2 I2S 启动

void i2s_start_(void)

启动 i2s 模块

参数：

None

返回：

None

6.7.3 I2S 停止

void i2s_stop_(void)

停止 i2s 模块

参数：

None

返回：

None

6.7.4 I2S 中断入口

void i2s_stop_(void)

停止 i2s 模块

参数：

None

返回：

None

示例：

```
__attribute__((section("ram_code"))) void i2s_isr_ram1(void)
{
    volatile uint32_t data;
    if(i2s_reg->status.rx_half_full) {
       for(uint32_t i=0; i<(I2S_FIFO_DEPTH/2); i++) {
            i2s_data = i2s_reg->data;
        }
        uart_putc_noint_no_wait(i2s_data);
    }
    if(i2s_reg->status.tx_half_empty) {
        for(uint32_t i=0; i<(I2S_FIFO_DEPTH/2); i++) {
            i2s_reg->data = 0;
        }
    }
}
```

7. OTA

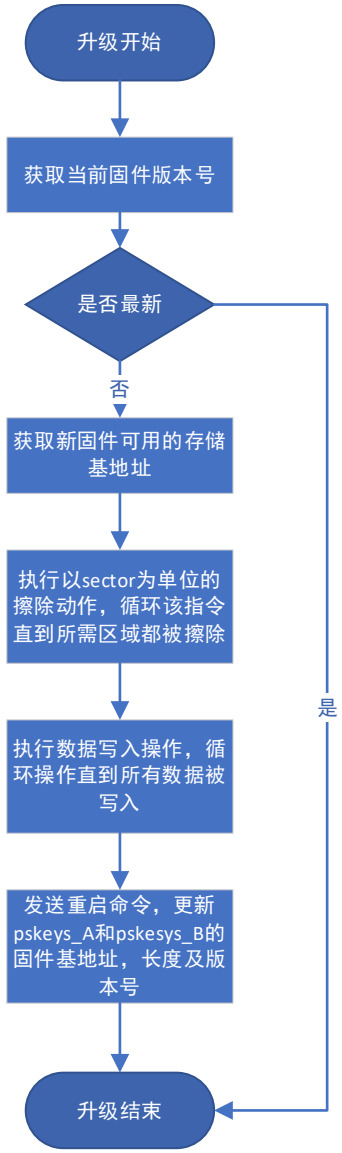
在 FR5080 的 SDK 中集成了一套完整的 OTA profile，用户可以基于此开发手机应用程序等 OTA 主机。FR5080 采用了双备份的方式进行固件的存储，参考 1.2.2 节的空间划分。

7.1 OTA profile

该 profile 的定义如下：
Service UUID：0xFE00
Write attribute UUID: 0xFF01
Notify attribute UUID: 0xFF02
Write attribute 用于 FR5080 接收来自 OTA 主机的指令，Notify attribute 用于 FR5080 回复 OTA 主机。在执行写操作时建议采用 write command 方式，以提高写入速度。
该代码位于 components\btdm\profiles\ble_ota 中。

7.2 OTA 流程

建议采用如下图所示的升级流程：

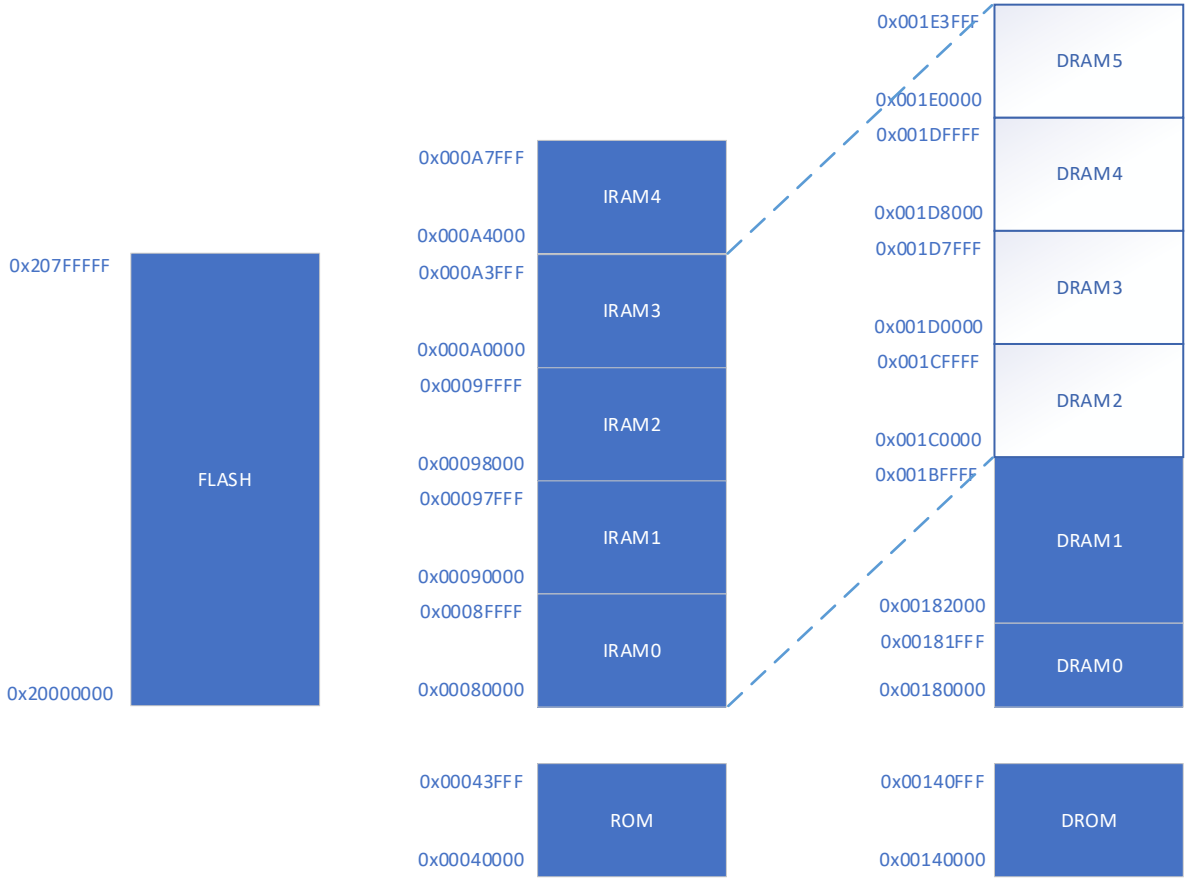

升级开始
获取当前固件版本号
是否最新
否
是
获取新固件可用的存储基地址
执行以sector为单位的擦除动作，循环该指令直到所需区域都被擦除
执行数据写入操作，循环操作直到所有数据被写入
发送重启命令，更新pskeys_A和pskesys_B的固件基地址，长度及版本号
升级结束

8. DSP

8.1 DSP 资源

8.1.1 DSP 存储结构

DSP 端的 memory 结构如下图所示：



DSP 端内置了 bootloader 用于用户代码的加载，这部分代码存储在 ROM 和 DROM 中，且占用了 DRAM0。IRAM 和 DRAM 为用户程序的运行空间，其中 IRAM0~IRAM3 可以被映射为 DRAM2~DRAM5，该部分映射受 MCU 控制，用户可以根据实际代码和数据大小进行调整，下面代码示例将 IRAM0 映射给 DRAM2，实现 IRAM 为 96KB，DRAM 为 312K。若外挂了 FLASH，用户程序可以存放在外部 FLASH 中。

system_set_dsp_mem_config(SYSTEM_DSP_MEM_CTRL_I96_D312)

8.1.2 DSP 系统时钟

DSP 的时钟由 MCU 端寄存器控制，在启动 DSP 前需要提前配置到合适的值，DSP 的主时钟源最高可以到 312MHz，经过分频最高系统时钟可以工作在 156MHz。用户可以通过函数：

void system_set_dsp_clk(enum system_dsp_clk_src_sel_t src, uint8_t div)

来进行 DSP 系统时钟的配置。

8.1.3 系统外设

8.1.3.1 UART

8.1.3.1.1 UART 初始化

void uart_init(uint8_t bandrate, void (*callback)(uint8_t))
初始化 UART 模块 参数: Bandrate: 波特率参数设置，例如 BAUD_RATE_115200 Callback : 接收完数据的回调函数 返回: None

8.1.3.1.2 UART 等待发送 FIFO 为空

void uart_finish_transfers(void);
等待发送 fifo 为空 参数: None 返回: None

8.1.3.1.3 UART 发送一个字节且等待完成

void uart_putc_noint(uint8_t c)
发送一个字符，且等到发送 fifo 为空 参数: c：发送的字符 返回: None

8.1.3.1.4 UART 发送字符串且等待完成

void uart_puts_noint(const char *s);

发送字符串，且等到发送 fifo 为空

参数:

s：发送字符串

返回:

None

8.1.3.1.5 UART 发送多个字节且等待完成

void uart_put_data_noint(const uint8_t *d, int size);
发送自定义字符长度，且等到发送 fifo 为空 参数: d：待发送数据的地址指针 size: 发送数据长度 返回: None

8.1.3.1.6 UART 读取特定长度的字符

void uart_get_data_noint(uint8_t *buf, int size);
读取特定长度的字符数据 参数: buf：待接收数据的地址指针 size: 读取数据长度 返回: None

8.1.3.1.7 UART 读取特定长度的字符，fifo 为空则返回

int uart_get_data_nodelay_noint(uint8_t *buf, int size);

读取特定长度的字符数据

参数:

buf：待接收数据的地址指针

size: 读取数据长度

返回:

int: 接收到数据的长度

8.1.3.2 TIMER

8.1.3.2.1 设置中断信息

_xtos_handler _xtos_set_interrupt_handler(int32_t n, _xtos_handler f)
设置中断源和中断回调处理函数 参数: n：中断源编号 f : 中断回调处理函数 返回: _xtos_handler: 返回函数指针

8.1.3.2.2 设置 timer 中断比较值

void xthal_set_ccompare(int, unsigned)
设置 timer 定时时间 参数: 参数一：中断源 参数二: 设置要定时时间长度 返回: None

8.1.3.2.3 使能 timer 中断源

Inline void _xtos_set_interrupt_handler(unsigned int intnum)

使能中断源

参数:

intnum:使能中断源

返回:

None

8.1.3.2.4 Timer 使用示例

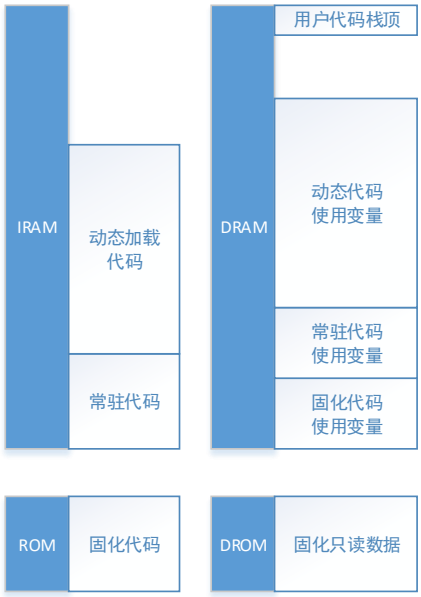
```
1.  Void set_timer_count(uint32_t count)
2.  {
3.      uint32_t current_ccount;
4.      //获取当前系统运行时钟数值
5.      current_ccount = xthal_get_ccount();
6.      //设置中断源和中断处理回调函数
7.      _xtos_set_interrupt_handler(XCHAL_TIMER1_INTERRUPT, timer0_isr);
8.      //设置 timer 定时值(当前时间+定时时间)
9.      xthal_set_ccompare(XCHAL_TIMER1_INTERRUPT,  current_ccount + count);
10.     //使能中断源
11.     _xtos_interrupt_enable(XCHAL_TIMER1_INTERRUPT);
12. }
13. ///注:count 计算方式,例如要定时 5ms, 5000/(1/156)= 780000
```

8.2 DSP 运行方式

FR5080 DSP 有两种运行方式：RAM 方式及 XIP 方式。RAM 方式是指 DSP 代码存放在芯片叠封的 flash 中，运行时，全部搬运到 DSP 的 IRAM 和 DRAM 中运行。XIP 方式是指 DSP 代码存放在外挂的 flash 中，运行时，直接从外部 flash 中直接运行，但启动代码需要放到芯片叠封的 flash 上。

8.2.1 RAM 方式

RAM 方式下，DSP 用户代码分为常驻代码和动态加载代码，常驻代码负责加载动态代码，动态加载代码属于按需加载（不同格式的语音编解码等）。这样整个系统的代码存放结构如下图所示：



针对 DSP 端代码我们提供了两个示例工程，一个是 fr5080_basic 用于构建常驻代码，一个是 fr5080_mp3_decoder 用于动态加载代码示例。

8.2.1.1 常驻代码功能简介

常驻代码工程 fr5080_basic 实现了基本的动态代码加载工程，对应使用的连接脚本为 fr5080_lsp_basic。代码首先初始化好 IPC；然后发送 IPC_MSG_DSP_READY 给 MCU，等待 MCU 后续指令。在该工程中提供了一组函数给动态加载代码区域使用：

- app_ipc_rx_set_user_handler：用于动态加载代码注册接收 MCU 发送过来的 IPC 消息的回调函数；
- app_register_default_task_handler：常驻代码提供了一个简单的任务队列机制，该函数用于注册任务处理用户消息处理函数；
- task_msg_alloc：创建一个 task，并分配这个 task 需要的参数空间；
- task_msg_insert：将消息推送到队列中；
- pvPortMalloc：分配一段内存空间，这段可分配空间的大小由 heap_4.c 中的 configTOTAL_HEAP_SIZE 变量决定，单位为字节，用户可以根据自己的需要进行调整；
- vPortFree：释放动态分配的空间；
- ipc_alloc_channel：分配 IPC 通道；
- ipc_free_channel：释放 IPC 通道；
- ipc_insert_msg：将 IPC 消息发送给对方；
- ipc_get_buffer_offset：获取 IPC 通道对应的共享内存地址。

8.2.1.2 常驻代码编译

默认的 IRAM 和 DRAM 分别配置为 160KB 和 256KB，如果这个分配不做调整，那么常驻代码的编译链接脚本不需要进行调整。当 IRAM 和 DRAM 的空间分配修改之后，那么这个链接脚本也要作相应的修改，以将 IRAM 和 DRAM 分别调整为 96KB 和 320KB 为例：

- 由于 IRAM 的起始地址有变化，IRAM 相关的地址要做修改：
  iram0_0_seg : org = 0x00080000, len = 0x400

iram0_1_seg : org = 0x00080400, len = 0x17C
iram0_2_seg : org = 0x0008057C, len = 0x24
iram0_3_seg : org = 0x000805A0, len = 0x40
iram0_4_seg : org = 0x000805E0, len = 0x3C
iram0_5_seg : org = 0x0008061C, len = 0x20
iram0_6_seg : org = 0x0008063C, len = 0x20
iram0_7_seg : org = 0x0008065C, len = 0x19A4
修改为：
iram0_0_seg : org = 0x00090000, len = 0x400
iram0_1_seg : org = 0x00090400, len = 0x17C
iram0_2_seg : org = 0x0009057C, len = 0x24
iram0_3_seg : org = 0x000905A0, len = 0x40
iram0_4_seg : org = 0x000905E0, len = 0x3C
iram0_5_seg : org = 0x0009061C, len = 0x20
iram0_6_seg : org = 0x0009063C, len = 0x20
iram0_7_seg : org = 0x0009065C, len = 0xf9A4

- 由于 IRAM 的起始地址有变化，向量地址要做修改：
PROVIDE(_memmap_vecbase_reset = 0x80400);
PROVIDE(_memmap_reset_vector = 0x80000);
修改为：
PROVIDE(_memmap_vecbase_reset = 0x90400);
PROVIDE(_memmap_reset_vector = 0x90000);
- 由于 DRAM 的结束地址有变化，栈顶需要做修改：
PROVIDE(__stack = 0x1c0000);
_heap_sentry = 0x1c0000;
修改为：
PROVIDE(__stack = 0x1d0000);
_heap_sentry = 0x1d0000;

8.2.1.3 动态代码流程介绍

用户可以根据需求进行动态加载代码的加载，比如打电话时动态加载回音消除程序、播放本地音频时切换不同的音频解码等。本节以 fr5080_mp3_decoder 为例，介绍在示例代码中如何通过调用常驻代码提供的接口实现从 MCU 接收原始 MP3 数据，然后进行解码，解码之后再编码成 SBC 数据传回给 MCU。

1. 首先定义程序的入口函数，并将该函数地址存放在所占代码空间的最头部，加载完动态加载代码后，MCU 需通知常驻代码到该位置回去入口函数地址，并调用进行初始化：

```
1.  // 指定该变量存放在 entry_point_section 段，在链接脚本中将该段存放在代码空间头部
2.  __attribute__((section("entry_point_section"))) const uint32_t entry_point[] = {
3.      (uint32_t)app_entry,
4.  };
5.
6.  void app_entry(void)
```

```
{
    uint32_t *ptr;
    uint8_t channel;
    // 初始化代码的 bss 段，该部分必不可少
    for(ptr = &_bss_start; ptr < & _bss_end;) {
        *ptr++ = 0;
    }
    printf("enter app entry: BUILD DATE: %s, TIME: %s\r\n", __DATE__, __TIME__);

    //mp3 解码器初始化
    mp3_decoder_init();

    // 向常驻代码中注册接收用户 ipc 消息的入口函数
    app_ipc_rx_set_user_handler(ipc_rx_user_handler);

   // 注册用户消息处理函数
    app_register_default_task_handler(user_task_handler);
}

```

2. 实现 ipc 消息的处理函数：

```
static void ipc_rx_user_handler(struct ipc_msg_t *msg, uint8_t chn)
{
    uint8_t channel;

    switch(msg->format) {
        //收到 MCU 发过来的 mp3 原始数据
        case IPC_MSG_RAW_FRAME:
            mp3_decoder_recv_frame(ipc_get_buffer_offset(IPC_DIR_MCU2DSP, chn), msg->length);
            break;
        //MCU 发过来的 sbc 格式参数
        case IPC_MSG_SET_SBC_CODEC_PARAM:
            memcpy(&sbc_info,ipc_get_buffer_offset(IPC_DIR_MCU2DSP, chn),sizeof(struct sbc_info_t));
            break;
        case IPC_MSG_WITHOUT_PAYLOAD:
            switch(msg->length) {
                case IPC_SUB_MSG_DECODER_START:
                    mp3_decoder_init();
                    break;
                //MCU 请求 sbc 数据
```

```
                case IPC_SUB_MSG_NEED_MORE_SBC:
                    sbc_encoder_recv_frame_req();
                    break;
                case IPC_SUB_MSG_REINIT_DECODER:
                    break;
            }
            break;
        default:
            break;
    }
}
```

3. 用户消息处理函数：

```
static void user_task_handler(struct task_msg_t *msg)
{
    switch(msg->id) {
        case MP3_DECODER_DO_DECODE:
            mp3_decoder_do_decoder_handler();
            break;
        default:
            break;
    }
}
```

4. 收到 mp3 数据后进行缓存并推送解码消息给任务队列：
   具体过程参见函数 void mp3_decoder_recv_frame(pWORD8 buffer, UWORD32 length)。
5. 在注册的用户消息处理函数中对用户消息进行处理：
   调用函数 void mp3_decoder_do_decoder_handler(void)对 mp3 原始数据进行解码
6. 解码完成后，调用函数 void sbc_encoder_recv_frame(pWORD8 buffer, UWORD32 length)对 mp3 解码后数据进行编码并缓存
7. 收到 MCU 请求 sbc 时，调用函数 void sbc_encoder_recv_frame_req(void)将缓存的 sbc 数据发送给 MCU

8.2.1.4 动态代码编译

针对动态加载代码的编译有以下几点需要注意：

1. 在工程的共目录下放置最新编译出来的常驻代码编译结果 fr5080_basic，从路径 fr5080_basic\bin\frbt_hifi3\Release 中获取
2. 代码和数据的起始地址和所占据的长度要合理配置：

```
MEMORY
{
  iram_seg :    org = 0x00082000, len = 0x10000
```

```
4.    dram_seg :    org = 0x00188000, len = 0x10000
5.  }
```

org 为起始地址，比如代码段起始地址设置为 0x00082000，也就是常驻代码的代码段空间结尾，起始地址加上长度也要保证不能超过实际 IRAM 和 DRAM 的上限，DRAM 还需要预留足够的空间给栈。当常驻代码所占用的代码和数据空间发生变化时，这里要做相应的修改。

3. 指定入口函数的地址存放位置：

```
1.  /*  Default entry point:  */
2.  ENTRY(app_entry)
3.
4.  SECTIONS
5.  {
6.    .entry_table : ALIGN(4)
7.    {
8.      _entry_table_start = ABSOLUTE(.);
9.      KEEP(*(entry_point_section))
10.       } >iram_seg :iram_phdr
11.
12.       …
```

8.2.1.5 生成 bin 文件

用户可以通过 PC 串口烧录工具将 DSP 的软件烧录到 flash 中，烧录的对象为 bin 文件，该文件可以通过工程根目录下的 generate_bin.py（基于 Python2.7）脚本生成，生成结果为 fr5080_basic.bin。

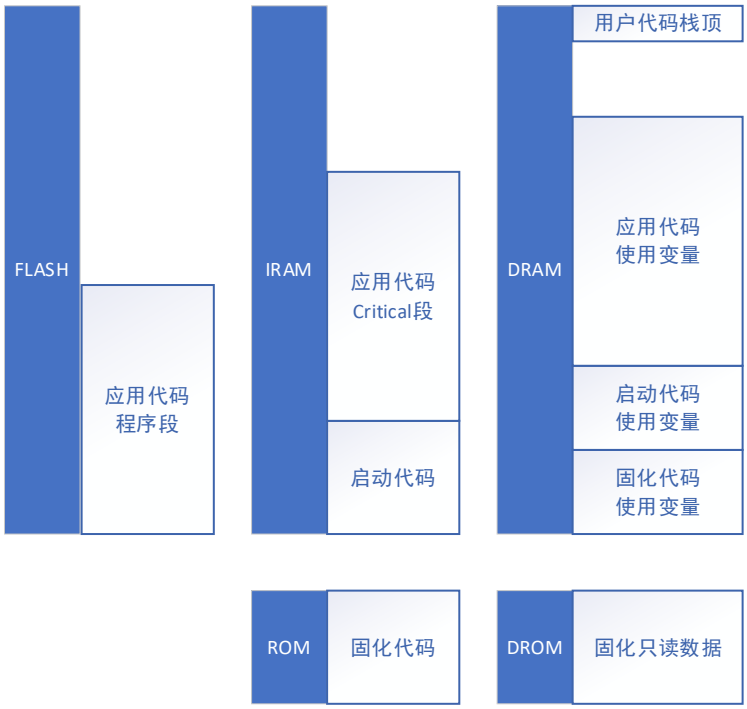
通过工程根目录下的 generate_bin.py（基于 Python2.7）脚本可以生成烧录用的 bin 文件。常驻代码生成 fr5080_basic.bin，动态代码生成为 fr5080_mp3_decoder.bin 和 fr5080_call.bin。同时我们提供 generate_total.py 脚本用于将常驻代码和动态加载代码组合成一个完整的 bin 文件用于烧录。组合时将需要组合的 bin 文件和脚本放在同一目录下，把要加入的 bin 文件按照顺序加入到脚本（generate_dsp_bin.py）的一个列表中：

source_table = ("fr5080_basic.bin", “fr5080_call.bin”, “”,"fr5080_mp3_decoder.bin",””)

执行后就可以生成用于烧录的 fr5080_dsp.bin。

8.2.2 XIP 方式

XIP 方式下，用户代码分为启动代码和应用代码部分。启动代码负责 dsp 串口烧录及 qspi 初始化；应用代码是用户程序，完成音频编解码，通话降噪，UI 显示等功能。XIP 方式的代码存放结构如下图所示：



针对 XIP 方式的 DSP 端代码我们提供了两个示例工程，一个是 fr5080_basic_xip 用于构建启动代码，一个是 fr5080_user_code_xip 用于应用代码示例。

8.2.2.1 启动代码功能简介

启动代码工程 fr5080_basic_xip 实现了 DSP 端串口，qspi，ipc 的初始化，提供了和上位机握手的交互程序，对应使用的连接脚本为 fr5080_lsp_basic_xip。代码初始化好后，发送 IPC_MSG_DSP_READY 给 MCU，等待 MCU 发送 IPC_MSG_EXEC_USER_CODE 消息，然后跳转到 FLASH 空间运行。

```
1.  // 这个工程的代码如无特殊需求，无需修改
2.  int main(void)
3.  {
4.      *(volatile uint32_t *)(0x50020004) = 0x02;
5.      *(volatile uint32_t *)(0x50020000) = 0xff;
6.
7.      // 初始化串口用于程序烧录
8.      uart_init(BAUD_RATE_115200, uart_recv_callback);
9.
10.     // 初始化 qspi，用于支持 flash 烧录和 XIP
11.     qspi_flash_init(1, 1);
12.     // 与上位机握手，执行烧录，这个参数决定了等待时间，后面可以进行调整，等待时间控制在 5ms 即可
13.     app_boot_host_comm(100);
14.
15.     qspi_cfg_set_baudrate(QSPI_BAUDRATE_DIV_4);
16.
17.     xthal_set_icacheattr(0x22222244);
```

```
    _xtos_set_interrupt_handler(XCHAL_UART_INTERRUPT, uart_isr);

    printf("\r\nDSP basic function start running.\r\n");

    // 初始化 IPC
    app_ipc_init();

    // 收到来自 M3 的 IPC_MSG_EXEC_USER_CODE 消息后，这个变量会进行修改，变成非 NULL 之后就进行调用运行到用户代码，不会再返回
    while(user_func_entry == NULL);
    user_func_entry();

    return 0;
}
```

8.2.2.2 启动代码编译

类似与 RAM 方式下常驻代码编译，连接脚本使用 fr5080_lsp_basic_xip。差别主要在于 XIP 模式下，iram 空间需求小，一般会转换部分 iram 空间到 dram 上，调用函数 system_set_dsp_mem_config 将 iram 地址设置为 32K，dram 设置为 384K。因此 Iram 起始地址从 0x80000 变成 0xa0000。

8.2.2.3 应用代码流程介绍

类似与动态代码流程，应用代码主要完成屏显示，各种算法实现等功能，MCU 通过 ipc 消息控制 DSP 进行相应的操作。下述代码为引用代码的入口函数：

```
void app_entry(void)
{
    uint8_t channel;
    //初始化 memory
    init_memory();

    printf("enter app entry: BUILD DATE: %s, TIME: %s\r\n", __DATE__, __TIME__);
    printf("user main v1.0.4\r\n");
    //初始化 flash，根据 flash 型号及特点，配置不同参数，
    flash_reinit();

    // 注册 ipc 消息处理函数
    app_ipc_rx_set_user_handler(ipc_rx_user_handler);
    // 任务初始化
    task_init();
    // 定时器初始化
    os_timer_engine_init();
```

```
    //串口初始化
    uart_init(BAUD_RATE_115200, uart_receive_char);
    _xtos_interrupt_enable(XCHAL_UART_INTERRUPT);

    //通知 MCU，DSP 已准备好
    ipc_msg_send(IPC_MSG_WITHOUT_PAYLOAD, IPC_SUM_MSG_DSP_USER_CODE_READY, NULL);

    //用户自定义任务处理函数
    task_schedule(user_msg_handler, sizeof(user_msg_handler)/sizeof(user_msg_handler[0]));

}
```

8.2.2.4 应用代码编译

针对应用代码的编译有以下几点需要注意：

- 在工程的共目录下放置最新编译出来的启动代码编译结果 fr5080_basic_xip，从路径 fr5080_basic_xip\bin\frbt_hifi3\Release 中获取
- 代码和数据的起始地址和所占据的长度要合理配置：

```
MEMORY
{
  flash_seg :                    org = 0x20000000, len = 0x100000
  dram_seg :                     org = 0x00183000, len = 0x5c000
  iram_seg :                     org = 0x000a3500, len = 0x4b00
}
```

org 为起始地址，比如代码段起始地址设置为 0x20000000，iram 的起始地址为 0x000a3500，实际根据启动代码的 iram 起始地址加 iram 长度获取。起始地址加上长度也要保证不能超过实际 IRAM 和 DRAM 的上限，DRAM 还需要预留足够的空间给栈。当启动代码所占用的代码和数据空间发生变化时，这里要做相应的修改。

8.2.2.5 生成 bin 文件

XIP 方式下，最终会生成两个 bin 文件，一个启动代码生成的 fr5080_basic_xip.bin 和一个应用代码生成的 fr5080_user_code_xip.bin。fr5080_basic_xip.bin 需要和 MCU 端的代码一起烧录到芯片叠封的 flash 中，而 fr5080_user_code_xip.bin 需要烧录到外挂 flash 中。

8.3 IPC 数据通信

8.3.1 硬件资源

MCU 与 DSP 的通信采用了 IPC 和共享内存机制，共有 8 组 IPC 控制路（MCU 和 DSP 各 4 路）和 2K（DSP2MCU 和 MCU2DSP 各 1K）的共享内存。每路 IPC 控制器有独立的控制寄存器和收发中断标志位，寄存器的定义如下：

Tag (1 bit)	Tog (1 bit)	Msg_type (4 bits)	Msg_length (10 bits)

Tag：对于消息发送方写 1 在对方产生中断，通知对方有新消息产生；接收方向该位写 0 为清除动作，通知对方该消息已经被处理结束；

Tog：表示当前传输的内容中存在 Payload 的情况，Payload 存放在哪个 buffer 中；

Msg_type：指定本次传输的消息类型。

Msg_length：如果发送的指令存在 Payload，这个字段用于存放 Payload 的长度。

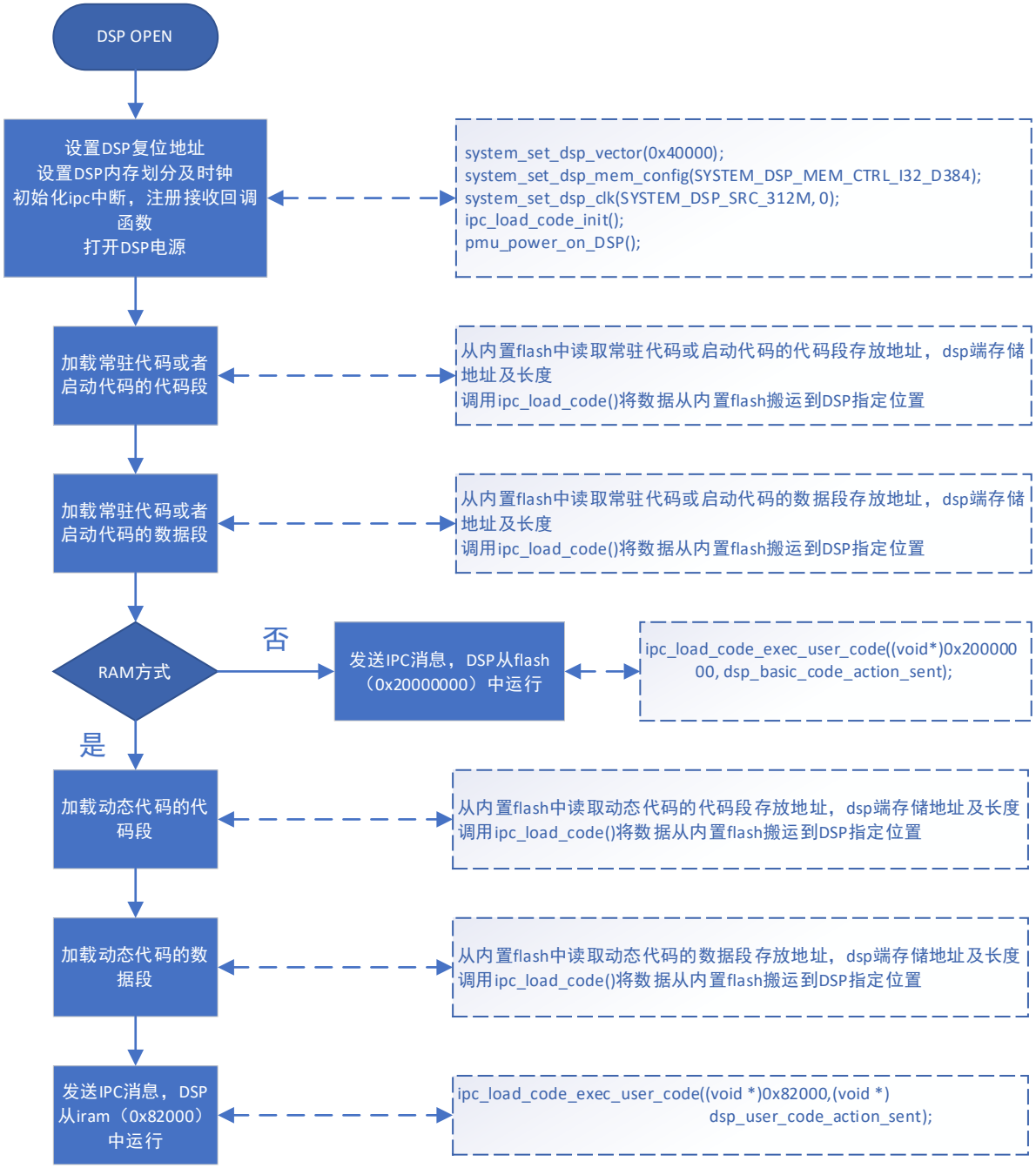
8.3.2 SDK 中的实现

8.3.2.1 初始化

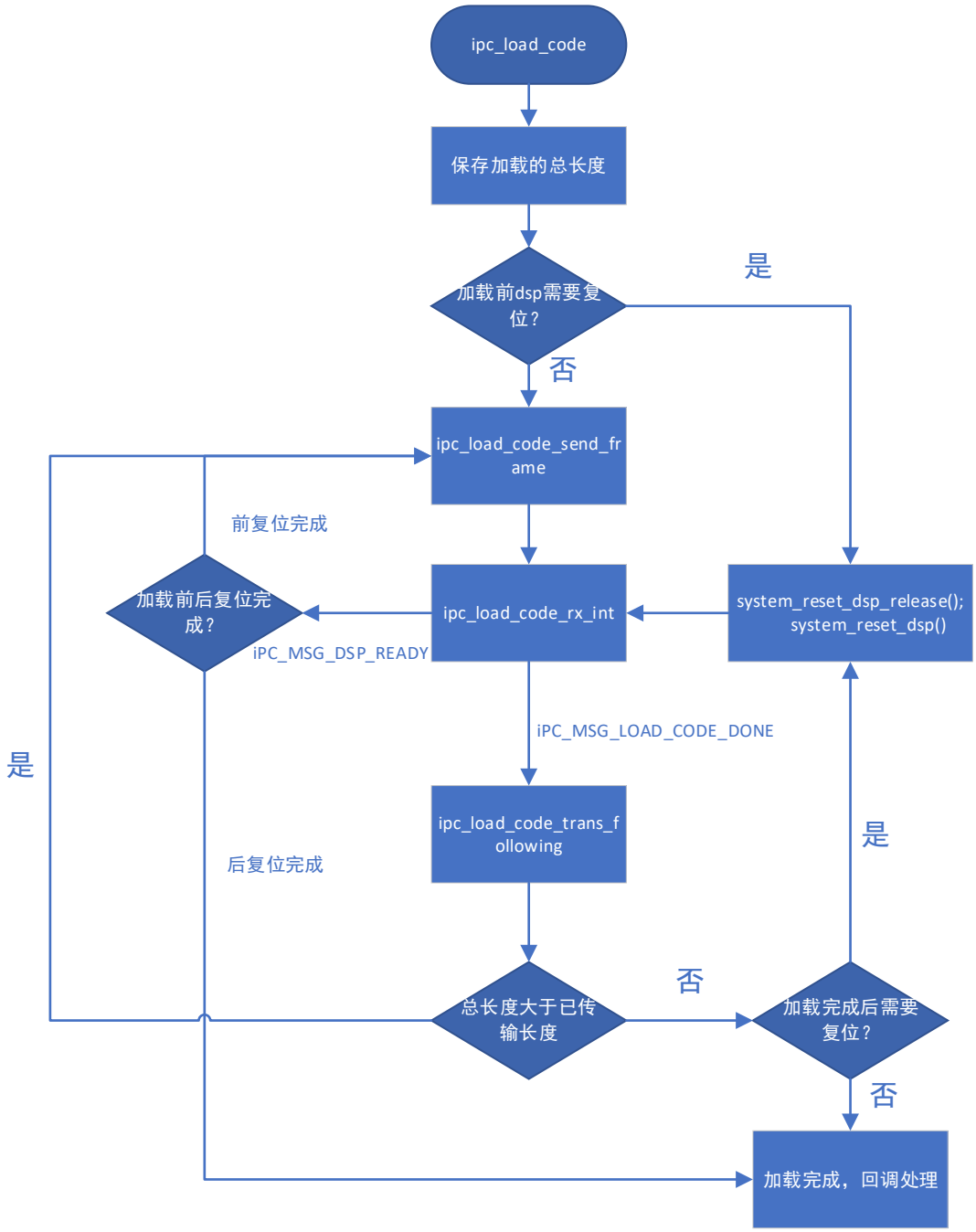
调用函数 void ipc_load_code_init(void) 进行 ipc 的初始化，使能 ipc 收发中断，注册回调函数，处理 DSP 代码加载过程及 DSP 发送过来的消息。

8.3.2.2 DSP 代码加载

不管是 RAM 方式还是 XIP 方式，都需要将代码从 flash 中搬到 iram 和 dram 中执行，SDK 中对于该部分的加载处理流程见下图。



MCU 加载代码到 DSP 的过程，主要是依靠 ipc_load_code()函数，详细流程如下图所示



8.3.2.3 消息发送

关于 IPC 的消息发送，SDK 有两个函数入口提供给用户使用，分别为 ipc_msg_send 和 ipc_msg_with_payload_send。相关参数定义及使用说明如下：

```
1.  /*********************************************************************
2.   * @fn      ipc_msg_send
3.   *
4.   * @brief   send message without payload.
5.   *
```

```
 * @param   msg         - message type.
 *          sub_msg     - this field will be set in ipc_msg_length segment.
 *          callback    - callback function for tx done
 *
 * @return  None.
 */
void ipc_msg_send(enum ipc_msg_type_t msg, uint16_t sub_msg, ipc_tx_callback callback);

/*********************************************************************
 * @fn      ipc_msg_with_payload_send
 *
 * @brief   send message with payload. total length of header and payload should
 *          not be larger than 1023.
 *
 * @param   msg         - message type.
 *          header      - message header, these data will be copy into the beginning of
 *                        exchange memory.
 *          header_length - header length
 *          payload     - message payload, these data will be copy following header
 *          payload_length - payload length
 *          callback    - callback function for tx done
 *
 * @return  None.
 */
void ipc_msg_with_payload_send(enum ipc_msg_type_t msg,
                               void *header,
                               uint16_t header_length,
                               uint8_t *payload,
                               uint16_t payload_length,
                               ipc_tx_callback callback);
```

IPC 消息类型定义如下，用户可以在该基础上自行添加

```
enum ipc_user_sub_msg_type_t {
    IPC_SUB_MSG_NEED_MORE_SBC,  // MCU notice DSP to send more sbc frame
    IPC_SUB_MSG_DECODER_START,  // MCU notice DSP to initiate MP3, AAC, etc. decoder
    IPC_SUB_MSG_REINIT_DECODER, // MCU notice DSP to reinitiate decoder engine
    IPC_SUB_MSG_NREC_START,         // MCU notice DSP to start nrec algorithm, not used,start immeadiately after user data load done
    IPC_SUB_MSG_NREC_STOP,          // MCU notice DSP to stop nrec algorithm
    IPC_SUB_MSG_FLASH_COPY_ACK,
    IPC_SUM_MSG_DSP_USER_CODE_READY, //DSP notice MCU user code ready
    IPC_SUB_MSG_DECODER_STOP,   // MCU notice DSP to stop adecoder
```

```
    IPC_SUB_MSG_DECODER_PREP_NEXT,  // MCU notice DSP to prepare for next song
    IPC_SUB_MSG_DECODER_PREP_READY, // DSP notice MCU ready to play next song
};

enum ipc_user_msg_type_t {
    IPC_MSG_LOAD_CODE = 0,
    IPC_MSG_LOAD_CODE_DONE = 1,
    IPC_MSG_EXEC_USER_CODE = 2,
    IPC_MSG_DSP_READY = 10,
    IPC_MSG_RAW_FRAME = 3,          // MCU send new raw frame to DSP
    IPC_MSG_DECODED_PCM_FRAME = 4,  // DSP send decoded pcm data to MCU
    IPC_MSG_ENCODED_SBC_FRAME = 5,  // DSP send encoded sbc frame to MCU
    IPC_MSG_WITHOUT_PAYLOAD = 6,    // some command without payload, use length segment in ipc-msg to indicate sub message
    IPC_MSG_RAW_BUFFER_SPACE = 7,   // used by DSP to tell MCU how much buffer space left to save raw data, return_value = actual_length / 256
    IPC_MSG_FLASH_OPERATION = 8,
    IPC_MSG_SET_SBC_CODEC_PARAM = 9, // MCU send codec sbc parameters to DSP(bitpool,sample rate...)
};
```

Ipc 消息的发送不用管数据是否已经发送出去，只需要调用上述的两个函数入口，函数内部会自行判断是否有空闲的 ipc 信道，若没有，则添加到链表里，等上一次的传输完成后，则从链表中取出缓存的消息，继续发送。

8.3.2.4 消息接收

消息接收的入口函数为 void ipc_user_rx_int(struct ipc_msg_t *msg, uint8_t chn)。用户在该函数里实现对 DSP 消息的处理。

```
__attribute__((section("ram_code"))) void ipc_user_rx_int(struct ipc_msg_t *msg, uint8_t chn)
{
    os_event_t event;

    switch(msg->format) {
        //audio source 应用中 DSP mp3 原始文件解码编码生成的 sbc 帧
        case IPC_MSG_ENCODED_SBC_FRAME:
            {
                uint32_t param[2];

                param[0] = (uint32_t)ipc_get_buffer_offset(IPC_DIR_DSP2MCU, chn);
                param[1] = msg->length;
```

```
13.
14.                 audio_source_statemachine(USER_EVT_NEW_SBC_FRAME, (void *)¶m[0]);
15.             }
16.             break;
17.         //audio source 应用中 DSP 请求 mp3 原始数据长度
18.         case IPC_MSG_RAW_BUFFER_SPACE:
19.             //uart_putc_noint_no_wait('~');
20.             //printf("raw space = %d\r\n",msg->length * 256);
21.             {
22.                 event.event_id = USER_EVT_REQ_RAW_DATA;
23.                 event.param = NULL;
24.                 event.param_len = msg->length * 256;
25.                 os_msg_post(user_task_id, &event);
26.             }
27.             break;
28.         //针对 without_payload 消息，一般消息的长度变量会被复用成子消息
29.         case IPC_MSG_WITHOUT_PAYLOAD:
30.             //DSP 用户代码已成功运行起来
31.             if(msg->length == IPC_SUM_MSG_DSP_USER_CODE_READY) {
32.                 event.event_id = USER_EVT_DSP_OPENED;
33.                 event.param = NULL;
34.                 event.param_len = 0;
35.                 os_msg_post(user_task_id, &event);
36.             }
37.             //DSP 已准备播放下一首歌
38.             else if(msg->length == IPC_SUB_MSG_DECODER_PREP_READY) {
39.                 event.event_id = USER_EVT_DECODER_PREPARE_NEXT_DONE;
40.                 event.param = NULL;
41.                 event.param_len = 0;
42.                 os_msg_post(user_task_id, &event);
43.             }
44.             break;
45.         default:
46.             break;
47.     }
48. }
```

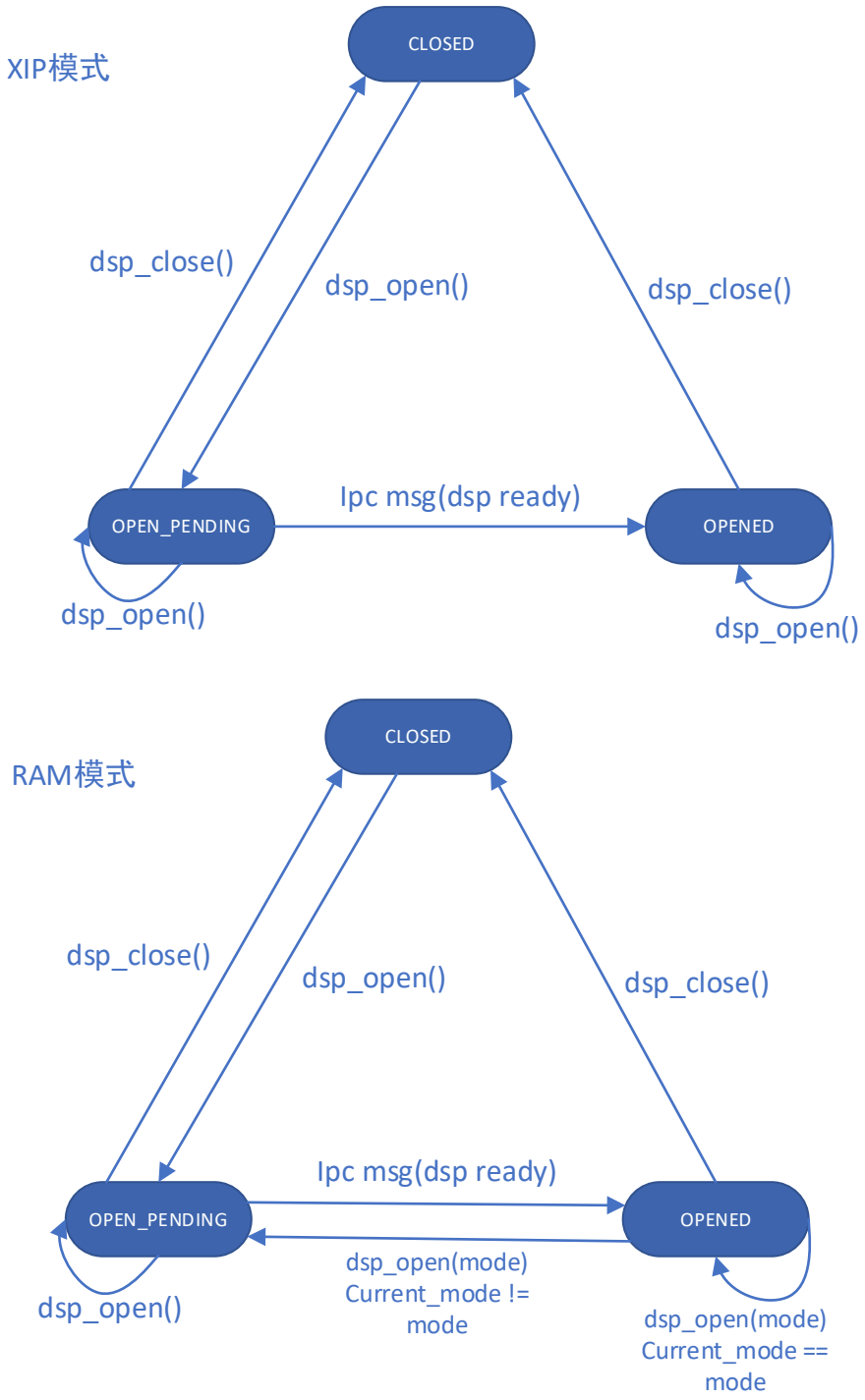
8.3.2.5 状态机

考虑到多个应用需要打开或者关闭 DSP，本章节以 audio source 应用为例介绍 MCU 端状态机设计。主要涉及到如下几个状态定义。

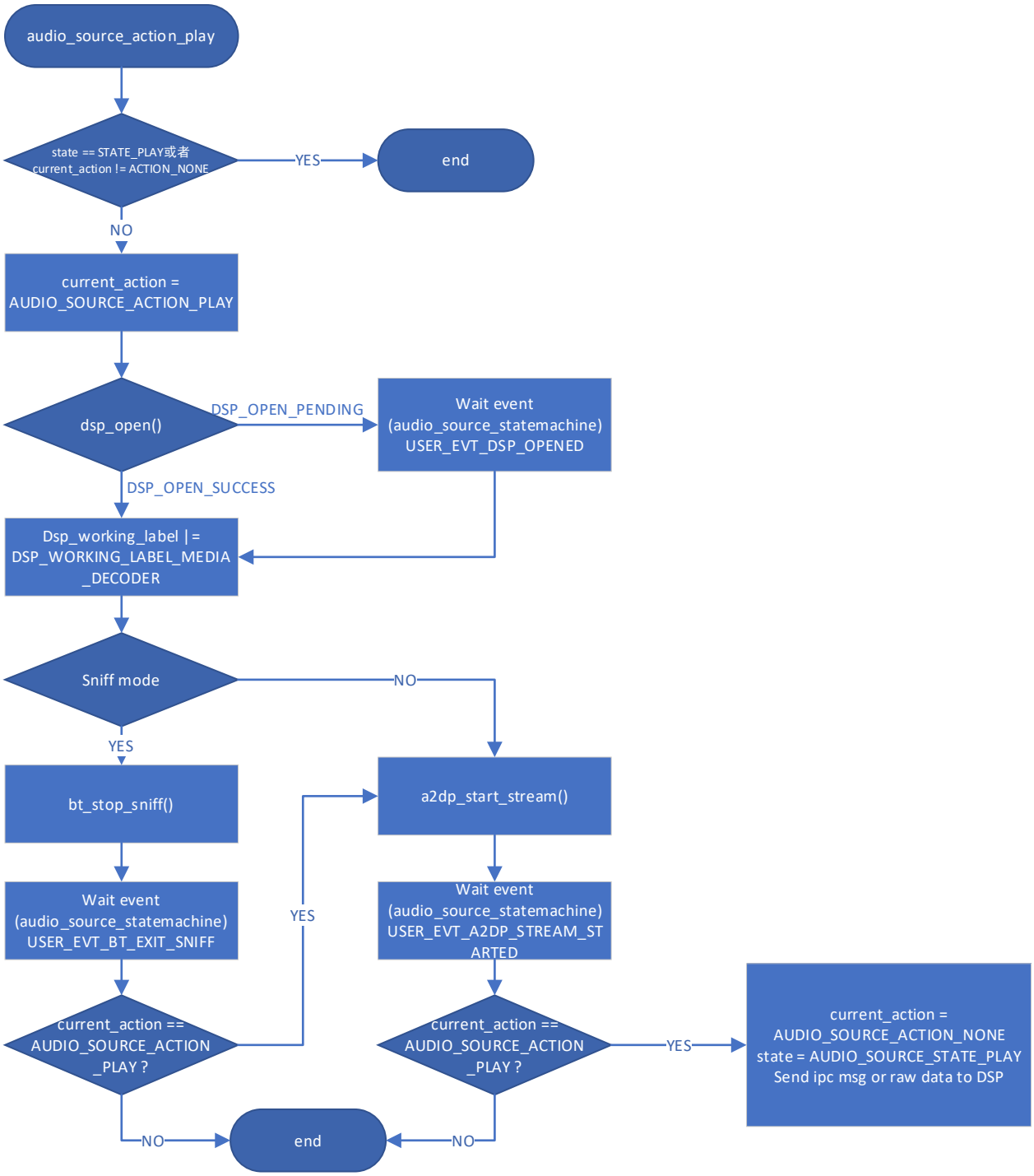
```
1.  //DSP 工作标志
```

```
/*表示 DSP 正在运行哪些任务，可以多个任务同时运行，当该标志不为 0 时，不能关闭 DSP*/
#define DSP_WORKING_LABEL_GUI                   0x01
#define DSP_WORKING_LABEL_VOICE_ALGO            0x02
#define DSP_WORKING_LABEL_AUDIO_SOURCE          0x04
#define DSP_WORKING_LABEL_NATIVE_PLAYBACK       0x08

//DSP 加载类型
enum dsp_load_type_t {
    /* 当工作在 XIP 模式，仅需要加载常驻代码，当工作在 RAM 方式，需要加载常驻代
    码和用户代码*/
    DSP_LOAD_TYPE_BASIC,

    /* 当工作在 RAM 方式，不同的用户代码需要按情况加载 */
    DSP_LOAD_TYPE_VOICE_ALGO,       //音频算法，例如 AEC,NR
    DSP_LOAD_TYPE_A2DP_DECODER,     // a2dp 解码，例如 AAC，LC3
    DSP_LOAD_TYPE_AUDIO_SOURCE,     // audio source 模式，MP3 解码和 SBC 编码
    DSP_LOAD_TYPE_NATIVE_PLAYBACK,  //MP3 解码并本地播放
};

//DSP 状态
enum dsp_state_t {
    DSP_STATE_CLOSED,       //关闭状态
    DSP_STATE_OPENING,       //正在打开状态
    DSP_STATE_OPENED,       //已打开状态
};

//Audio source 状态
enum audio_source_state_t {
    AUDIO_SOURCE_STATE_IDLE,    //空闲状态
    AUDIO_SOURCE_STATE_PLAY,    //播放状态
    AUDIO_SOURCE_STATE_PAUSE,    //暂停状态
};

//Audio source 命令
enum audio_source_action_t {
    AUDIO_SOURCE_ACTION_NONE,    //空闲
    AUDIO_SOURCE_ACTION_PLAY,    //播放
    AUDIO_SOURCE_ACTION_PAUSE,    //暂停
    AUDIO_SOURCE_ACTION_NEXT,    //下一曲
    AUDIO_SOURCE_ACTION_PREV,    //上一曲
    AUDIO_SOURCE_ACTION_FAST_FORWARD,    //快进
    AUDIO_SOURCE_ACTION_FAST_BACKWARD,   //快退
};
```

DSP 打开过程如下：

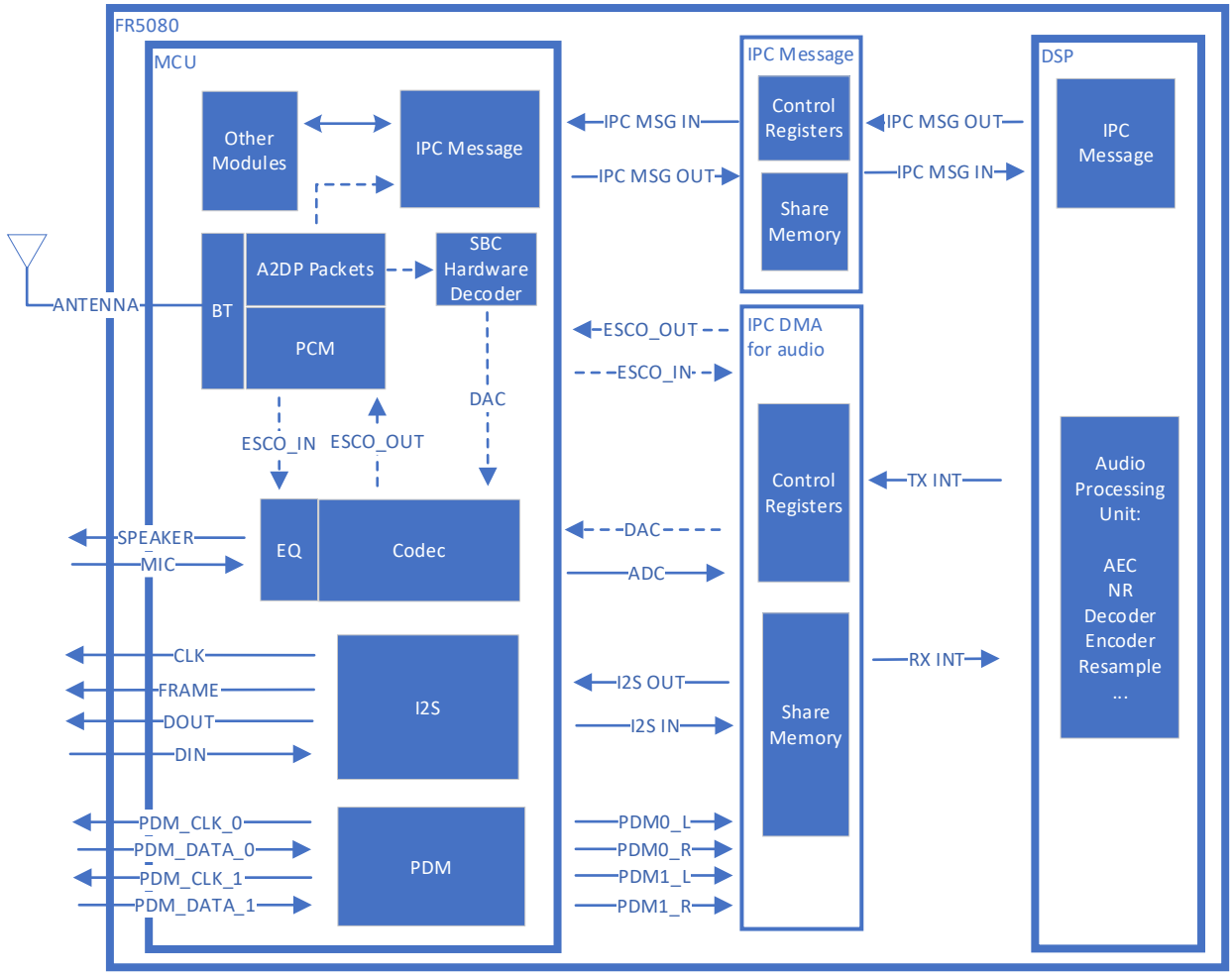


Audio source 播放基本流程如下图所示：



8.4 MCU 和 DSP 音频通路

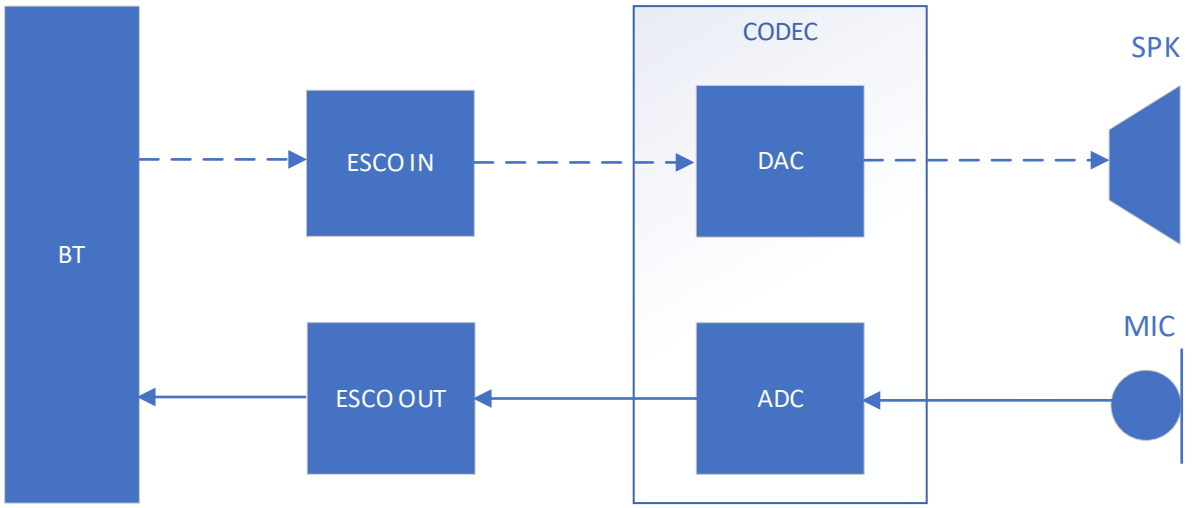
FR5080 中包含了 CODEC（ADC 和 DAC），IPC 共享内存，SBC 硬件解码器，EQ，I2S，PDM 等音频模块。下图描述了整个 FR5080 中音频交互通路。



ESCO_IN --- 表示远端 BT 传进来的语音数据（PCM 数据）
ESCO_OUT --- 表示通过 BT 传出去的语音数据（PCM 数据）
ADC --- 表示 MIC 进来的 ADC 数据
DAC --- 表示传给 SPEAKER 的 DAC 数据
TX INT --- 表示 DMA 搬运完成，产生的 TX 中断请求
RX INT --- 表示 DMA 接收完成，产生的 RX 中断请求
PDM0_L --- 表示 PDM0 左声道数据
PDM0_R --- 表示 PDM0 右声道数据
PDM1_L --- 表示 PDM1 左声道数据
PDM1_R --- 表示 PDM1 右声道数据
I2S IN --- 表示输入 I2S 数据
I2S OUT --- 表示输出 I2S 数据
下述章节详细描述了一些基本音频通路流程。

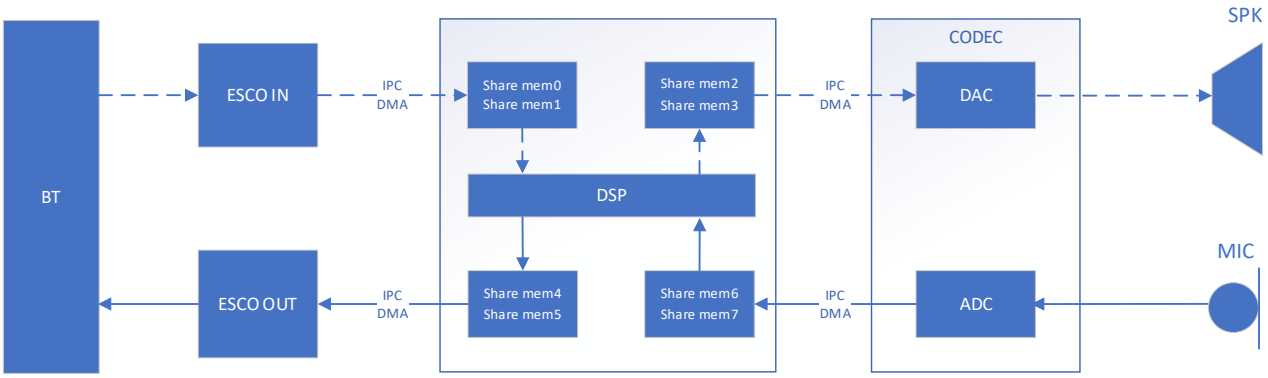
8.4.1 语音通话通路（模拟 mic，本地 speaker 播放，无 DSP 降噪算法）

模拟 mic 进来的数据，通过 ADC，自动填入到 ESCO OUT 缓存，通过 BT 发送出去，从 BT 接收的语音数据，会被填入到 ESCO IN 缓存，通过 CODEC 的 DAC 模块，转换成模拟信号交给本地 speaker 播放。（虚线代表 BT 接收数据通路，实线代表 BT 发送数据通路）



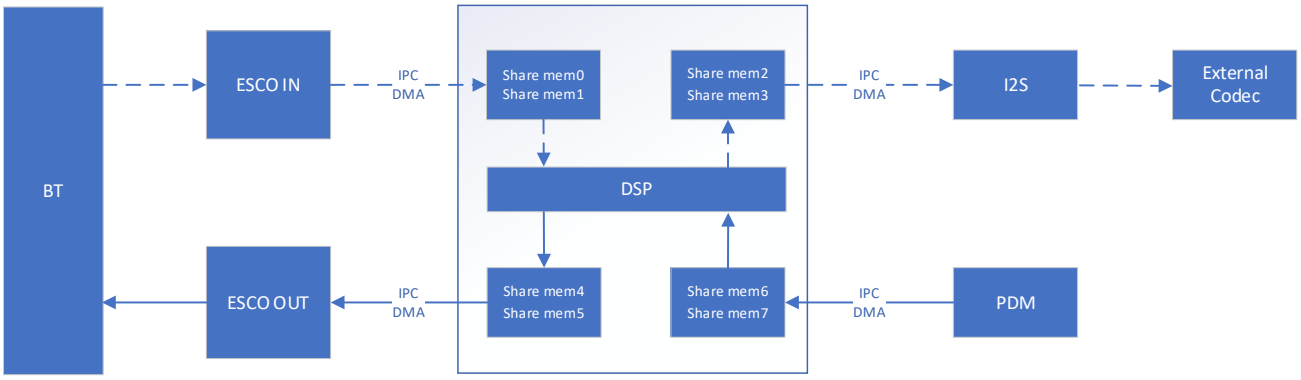
8.4.2 语音通话通路（模拟 mic，本地 speaker 播放，DSP 降噪算法）

BT 接收到的数据 ESCO_IN 和模拟 mic 进来的 ADC 数据经过 IPC DMA 搬运到 Share Memory，产出 IPC DMA 接收中断，DSP 获取数据处理后，再次填到 Share Memory，DMA 会将相应数据分别搬运到 Codec 的 DAC 模块和 ESCO OUT 缓存。注意 Share Memory 会被分成多个地址段，通过 IPC 控制寄存器设置，每个对应的地址段又会分为两个区域，保证每次写入和读取是在不同的区域。



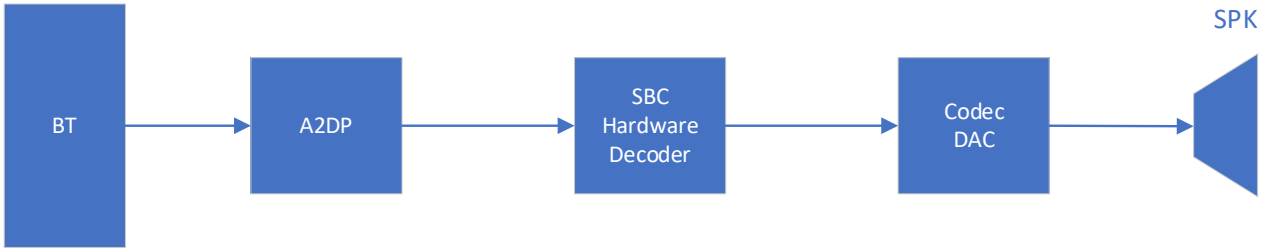
8.4.3 语音通话通路（数字 PDM 输入，I2S 输出，外部 codec 播放，DSP 降噪算法）

BT 接收到的数据 ESCO_IN 和数字麦 PDM 数据经过 IPC DMA 搬运到 Share Memory，产出 IPC DMA 接收中断，DSP 获取数据处理后，再次填到 Share Memory，DMA 会将相应数据分别搬运到 I2S FIFO 和 ESCO OUT 缓存。



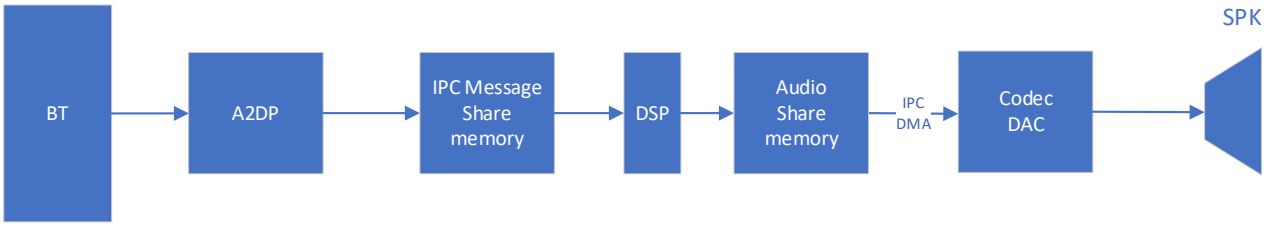
8.4.4 音乐播放通路（sbc，无 DSP 解码）

BT 接收到的 ACL 数据，在 A2DP 协议里提取 SBC 原始数据，通过 SBC 中断方式，将原始数据填入 SBC FIFO，SBC 硬件解码后，传输给 DAC 模块，转换成模拟信号交由本地 Speaker 播放



8.4.5 音乐播放通路（aac，DSP 解码）

BT 接收到的 ACL 数据，在 A2DP 协议里提取 AAC 原始数据，通过 IPC 消息发送给 DSP，DSP 解码后存入到 Audio share memory，之后 DMA 会将数据搬运到 DAC FIFO 中，经过数模转换后，交由 Speaker 播放。



联系方式

Feedback: Freqchip welcomes feedback on this product and this document. If you have comments or suggestions, please send an email to doc@freqchip.com.

Website: www.freqchip.com
Sales Email: sales@freqchip.com
Phone: +86-21-5027-0080

勘误记录

Reversion Number	Reversion Date	Description
V0.1	2020.9.2	draft
V1.0.0	2020.9.11	正式版本
V1.1.0	2021.7.12	增加固件烧录，OTA 及 DSP 部分
V1.2.1	2021.7.15	增加 DSP 音频通路部分说明