Audio Server 设计文档

V0.1

2023 年 7 月 27 日

思澈科技（上海）有限公司

www.sifli.com

版权 ©2023

思澈科技

目录

1 目的 *3*

2 Audio 框架 *3*

2.1 软件层次 **3**

2.2 client 端 API **5**

2.2.1 audio_open 5

2.2.2 audio_close 6

2.2.3 audio_write 6

2.2.4 audio_flush 6

2.3 音量 API **6**

2.3.1 设置 public 音量 7

2.3.3 设置喇叭静音 7

2.3.4 设置 mic 静音 7

2.4 设置音频输出设备 **7**

2.4.1 注册音频设备函数 8

2.4.2 选择音频设备函数 8

2.5 特殊 API **9**

3. 消息数据流程 *10*

3.1 mic 数据采集数据流程 10

3.2 本地音乐播放 speaker 数据播放流程 11

3.3 A2DP sink 音乐播放 speaker 数据播放流程 12

3.4 本地音乐到蓝牙耳机数据流程 13

3.5 BT voice 播放到 speaker 数据播放流程 14

3.6 录音数据播放流程 15

3.7 modem 到 speaker/mic 数据流程 16

3.8 modem 到蓝牙耳机流程 16

4. EQ 调试 *18*

更新历史 *19*

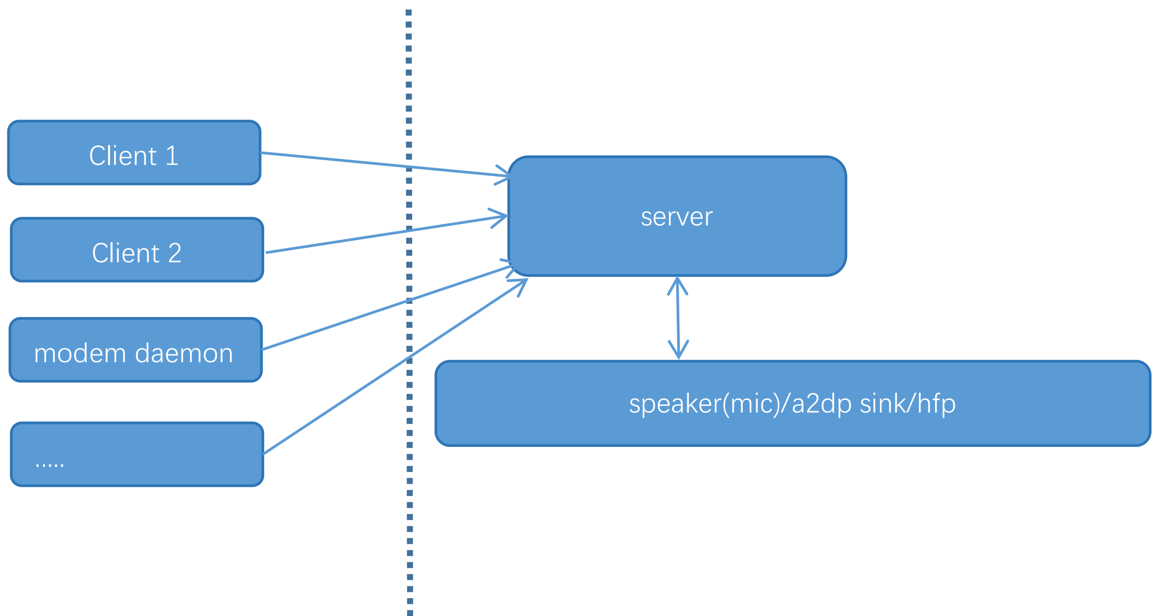
1 目的

Audio 对外提供的 API 和消息处理，如果抛弃 audio server，自己管理音频设备，这参考 audio server 调的底层驱动自己直接调用管理。

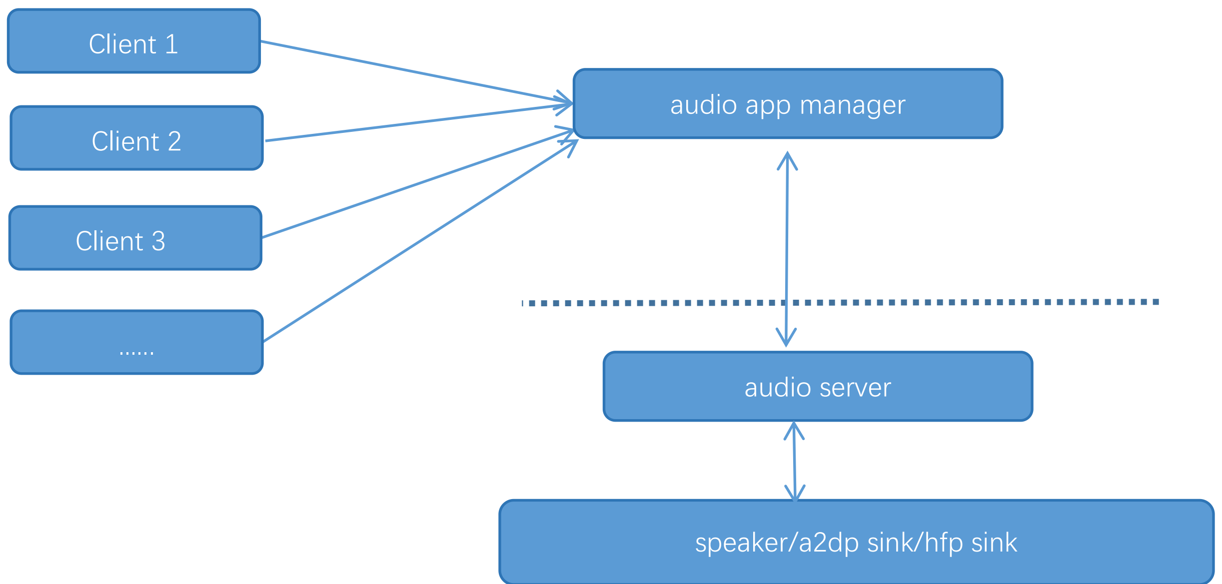
2 Audio 框架

2.1 软件层次

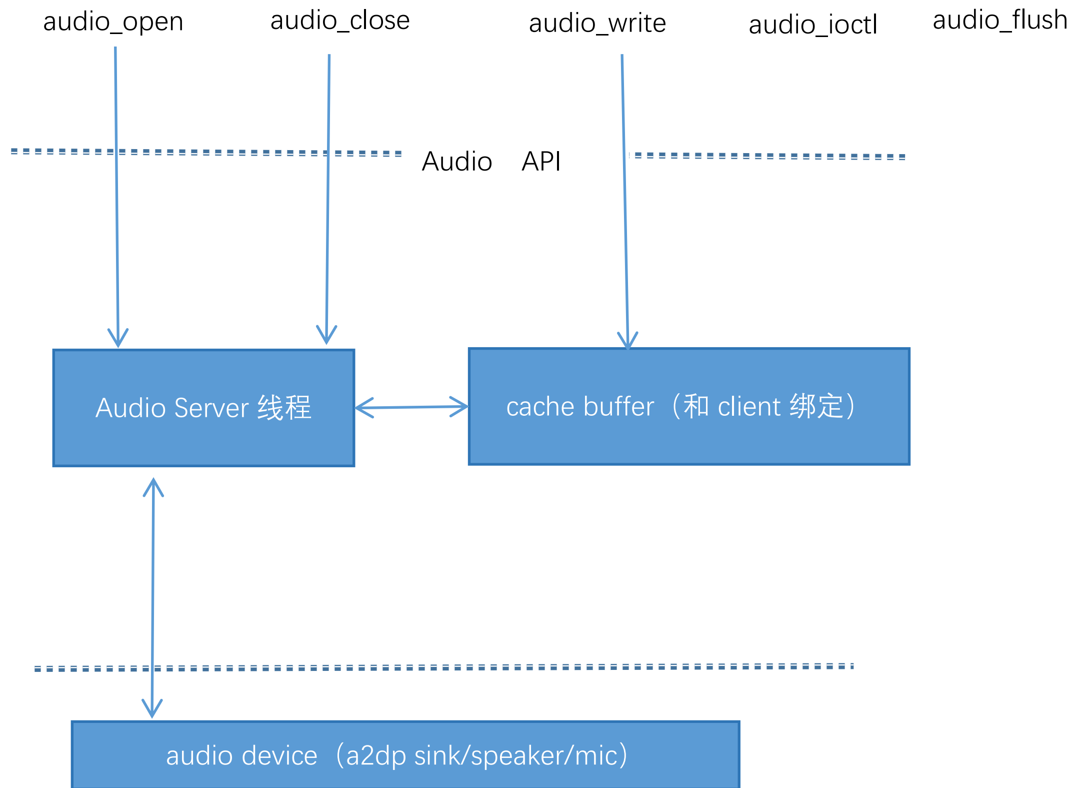
audio 硬件资源只有一个，采用 client/server 模式管理硬件资源，多个 APP 访问 audio， 每个 APP 就是 client，先 open 发起使用申请，在 server 里仲裁是否允许使用，高优先级打断低优先级的，把低优先级的放入 suspend 队列，高优先级 close 后，从 suspend 队列里找一个最高优先级的恢复。实际设计内存不足问题，solution 那边有时也会主动关闭当前，再打开新的。下面虚线左边为 client，声音的源头，是主动的，右边为目的，是需要选择的，两边可能是双向语音数据。



随着 audio 输入和输出源头的增加，不同的客户对 app 的优先级需求可能不同，优先级的仲裁可能要从 server 分离出来，server 只负责硬件资源的使用，app 间的优先级从 server 分离到上层处理，变成这样的， 目前没有分离。mic 是根据打开 speaker(mic)是根据打开时的读写标志选择一个或二个，不支持 speaker 和 mic 分别同时独立打开。


Client 1
Client 2
Client 3
......
audio app manager
audio server
speaker/a2dp sink/hfp sink

2.2 client 端 API



2.2.1 audio_open

```
audio_client_t audio_open( audio_type_t audio_type,
                          audio_rwflag_t rwflag,
                          audio_parameter_t *paramter,
                          audio_server_callback_func callback,
                          void *callback_userdata);
```

audio_type --- 声音类型，内部有根据类型排优先级

paramter --- 设置声音通道采样率，缓存大小等

callback --- 被挂起，恢复，录音数据到达， 缓存半空/全空等消息回调

返回 client 端的句柄

在这个数组里判断优先级，现在不支持混音，数字大的优先级高，最大 255，优先级高的可以打断优先级低的。

```
336: static const audio_mix_policy_t mix_policy[AUDIO_TYPE_NUMBER] =
337: {
338:     [AUDIO_TYPE_BT_VOICE]       = {94, BT_VOICE_MIX_WITH},
339:     [AUDIO_TYPE_BT_MUSIC]       = {11, BT_MUSIC_MIX_WITH},
340:     [AUDIO_TYPE_ALARM]          = {88, ALARM_MIX_WITH},
341:     [AUDIO_TYPE_NOTIFY]         = {80, NOTIFY_MIX_WITH},
342:     [AUDIO_TYPE_LOCAL_MUSIC]    = {10, LOCAL_MUSIC_MIX_WITH},
343:     [AUDIO_TYPE_LOCAL_RING]     = {92, LOCAL_RING_MIX_WITH},
344:     [AUDIO_TYPE_LOCAL_RECORD]   = {93, 0},
345:     [AUDIO_TYPE_MODEM_VOICE]    = {94, 0},
346: };
```

2.2.2 audio_close

int audio_close(audio_client_t handle);

关闭句柄

2.2.3 audio_write

```
int audio_write( audio_client_t handle,
                 uint8_t *data,
                 uint32_t data_len)
```

数据写入句柄对应的缓存

2.2.4 audio_flush

现在是 ring buffer 传递数据，非 queue buffer 模式，每一次的 write 无法对应一次该数据播放完成的回调，只有整个缓存半空或空的回调，最后的数据需要 flush 完成。考虑到性能问题，缓存空了并不能认为上层的播放完了，需要上层主动 flush 等待底下把缓存都播放完成。

2.3 音量 API

音量按 public 和 private 两种，public 音量只有一个，每种音乐类型有自己的 private 音量。

如果某种音乐类型没有设置过它的 private 音量，那播放这种类型音乐就使用 public 音量，否则使用它自己的 private 音量，提供的音量 API 设置都在内存里，重启会丢失。

2.3.1 设置 public 音量

int audio_server_set_public_volume(uint8_t volume);

volume ---- [0, 15]

2.3.2 设置 private 音量

int audio_server_set_private_volume(audio_type_t audio_type, uint8_t volume);

2.3.3 设置喇叭静音

不区分音乐类型

int audio_server_set_public_speaker_mute(uint8_t is_mute);

2.3.4 设置 mic 静音

int audio_server_set_public_mic_mute(uint8_t is_mute);

2.4 设置音频输出设备

本地音频可能走 speaker，也可能走 a2dp sink，或 tws 耳机， 暂时称为音频的设备选择。音频设备目前的设计切换在 server 里进行的，关闭旧的，打开新的. 这部分自动切换 server 内暂未实现，需要上层先关闭旧的 audio_open 的句柄，再选择设备打开。设备选择有如下 API.

设备和音量一样，按音乐类型，支持 public 和 private， 某种类型没设置过设备，就使用 public 的，设置了就使用 private 的。

audio 分为输入和输出，之前的输入要么本地音乐，要么 BT HFP/a2dp source，都是他们直

接直接调用 audio 接口的，BT 中断来了，server 里调他们的接口取数据。有些语音是双向的，可以分为远端和近端，这里的设备只针对近端，两边语音可能是双向的。modem 算在远端里， 并不通过设备接口设置，可以在音乐类型里有个 modem 类型，通过 audio_open(audio_type,)接口里的音乐类型变量区分。
从手表端看，目前规划的远端和近端定义：
远端： modem、本地音乐文件播放器、手机蓝牙语音、手机蓝牙音乐
近端： 喇叭/mic、蓝牙耳机电话语音、 蓝牙耳机音乐

设备是选择近端. 远端的实现不在 server 里，远端设备类型由 app 或一个模块用 audio_open()选择。

2.4.1 注册音频设备函数

```
struct audio_device
{
    int (*open)(void *user_data, audio_device_input_callback callback);
    int (*close)(void *user_data);
    uint32_t (*output)(void *user_data, const struct rt_ringbuffer *rb);
    void *user_data; //not use now
    int (*ioctl)(void *user_data, int cmd, void *val);
};
```

open里的这个callback每个类型都要在server实现这个回调函数，处理音频设备发来的data coming， buffer empty等消息

```
int audio_server_register_audio_device(audio_device_e device_type,
                                       struct audio_device *p_audio_device);
```

其中的 open，在调用时需要传入一个 audio_device_input_callback 回调函数，用于接受设备端的消息事件和输入数据事件。 speaker 默认注册了。
调了这个函数后，server 里会关闭旧的音频设备，调新的音频设备的 open()回调函数，并在 open()里传入本地实现的该类型音频设备的回调函数，用于接受消息，包括数据到达，和音频设备的 buffer half emtpy 等消息。一般是收到 buffer half emtpy 消息才会往音频设备的 output()回调里发送数据。

2.4.2 选择音频设备函数

```
typedef enum
{
```

```
    AUDIO_DEVICE_SPEAKER      = 0,
    AUDIO_DEVICE_A2DP_SINK    = 1,
    AUDIO_DEVICE_TO_HFP    = 2,

    AUDIO_DEVICE_TO_NUM
} audio_device_e;
```

int audio_server_select_public_audio_devicer(audio_devie_e device_type);
开机此函数从来没调的话，默认是到 AUDIO_DEVICE_SPEAKER

2.5 特殊 API

void bt_rx_event_to_audio_server()

BT 语音下行数据到了，BT 会调用此函数，可以在此函数里给 BT 下行数据处理线程发消息，收到消息再去读 BT 语音数据并处理。

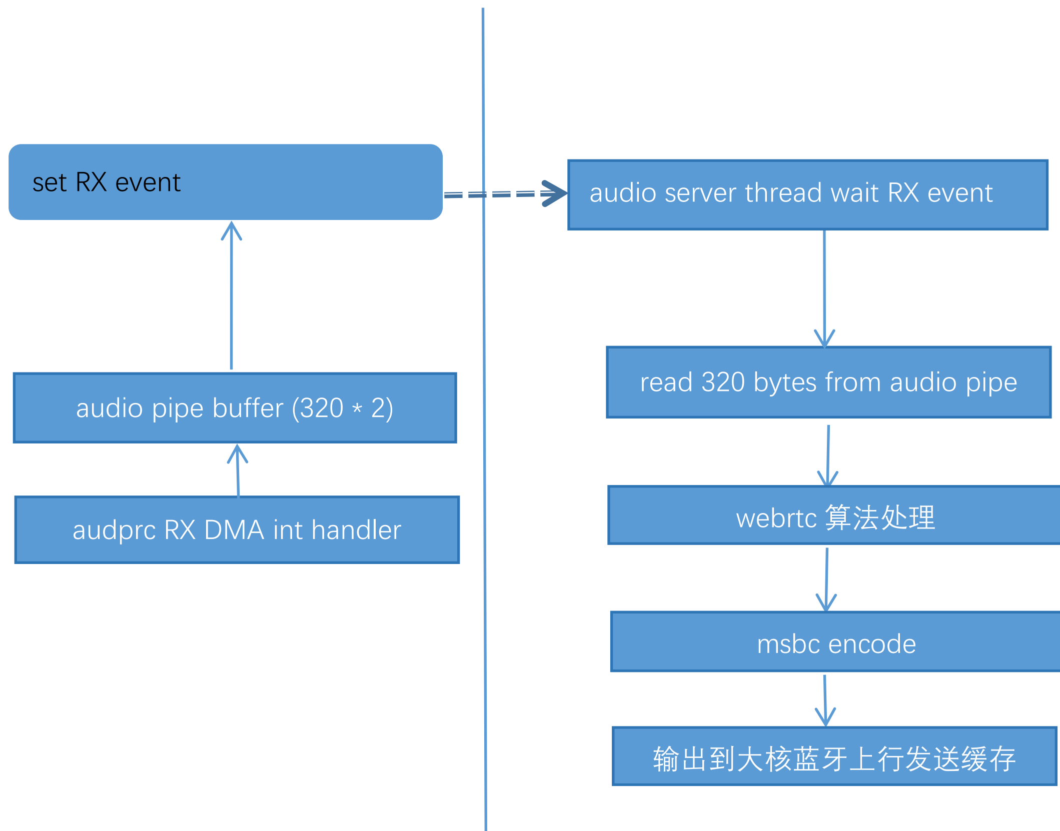
```
void audio_server_register_listener(audio_server_listener_func func,
                                     uint32_t what_to_listen, uint32_t reserved);
```

早期 solution 要求的函数, 监听一些 play/stop 事件, 想看看 audio 状态, 这些目前 solution 需要一个音频管理的模块，这样它能知道 audio 状态， 和 UI 能匹配，不用靠回调，不够集中管理 UI 状态。

3. 消息数据流程

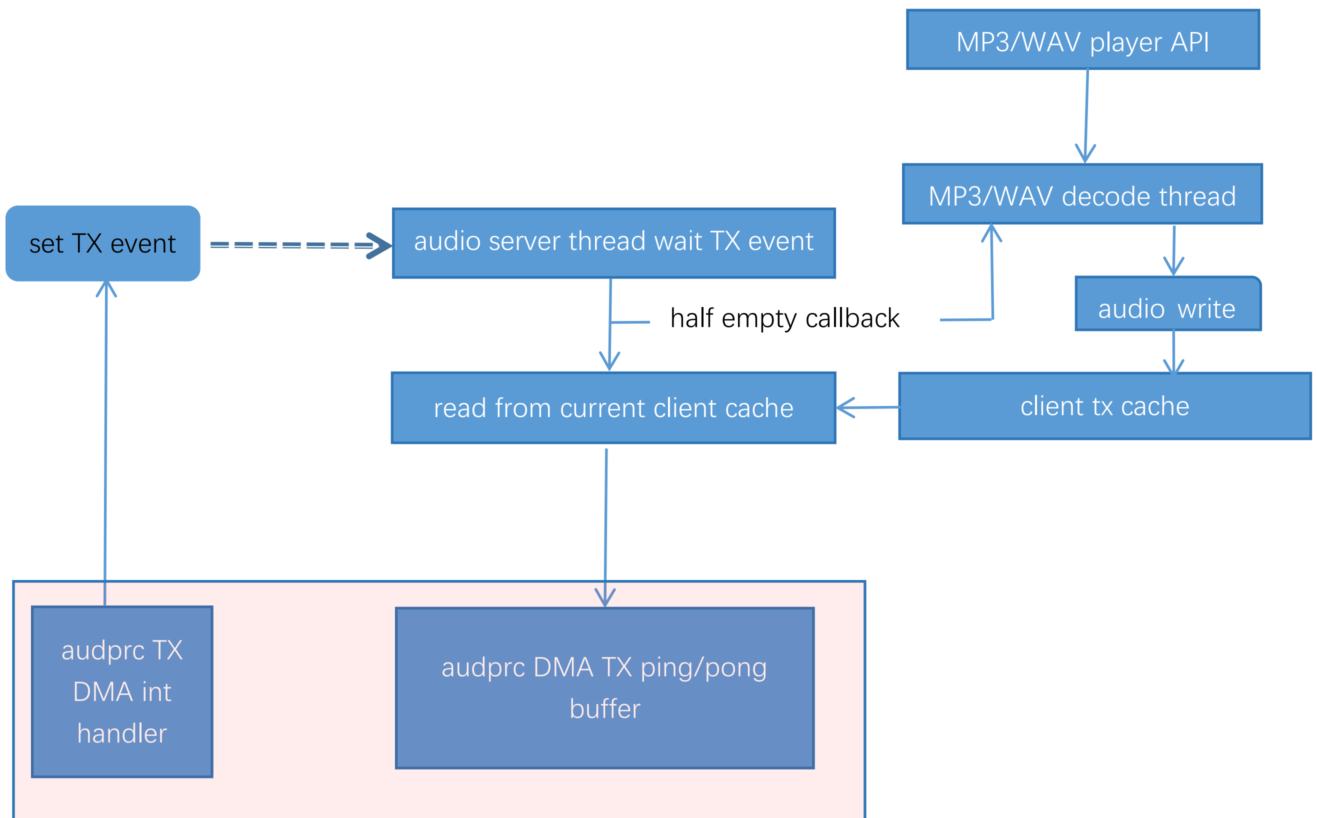
3.1 mic 数据采集数据流程

音频设备为 speaker 才有效



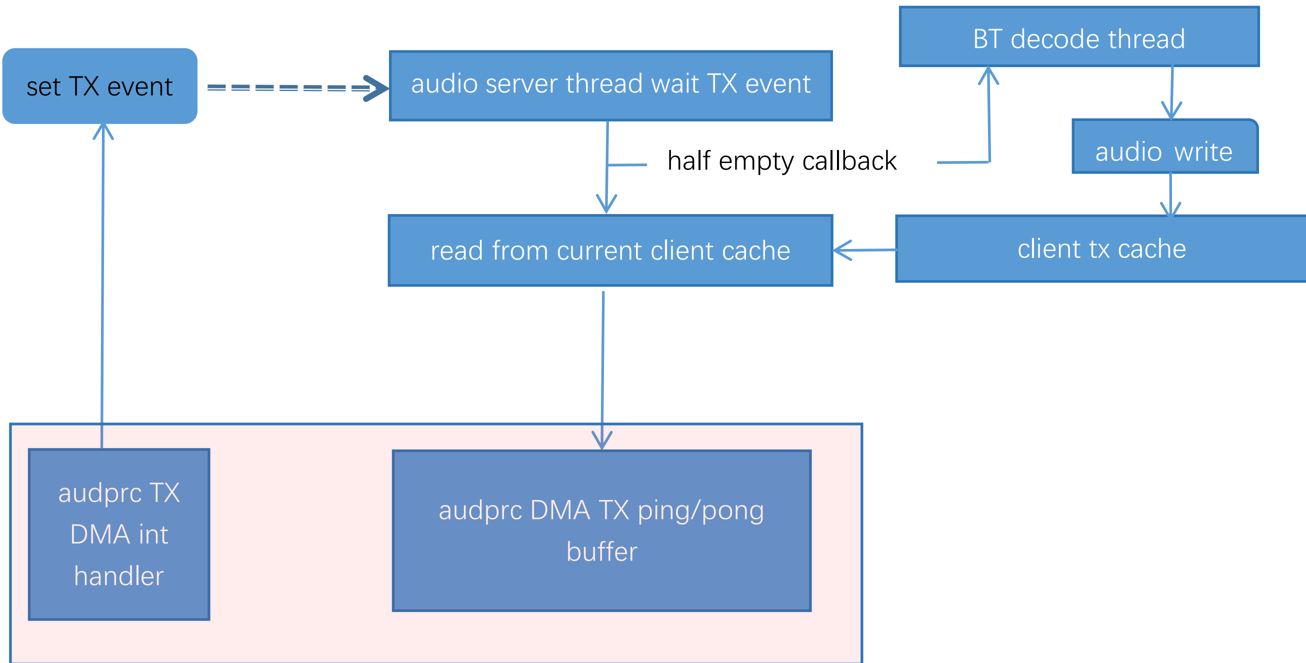
3.2 本地音乐播放 speaker 数据播放流程

音频设备为 speaker 才有效



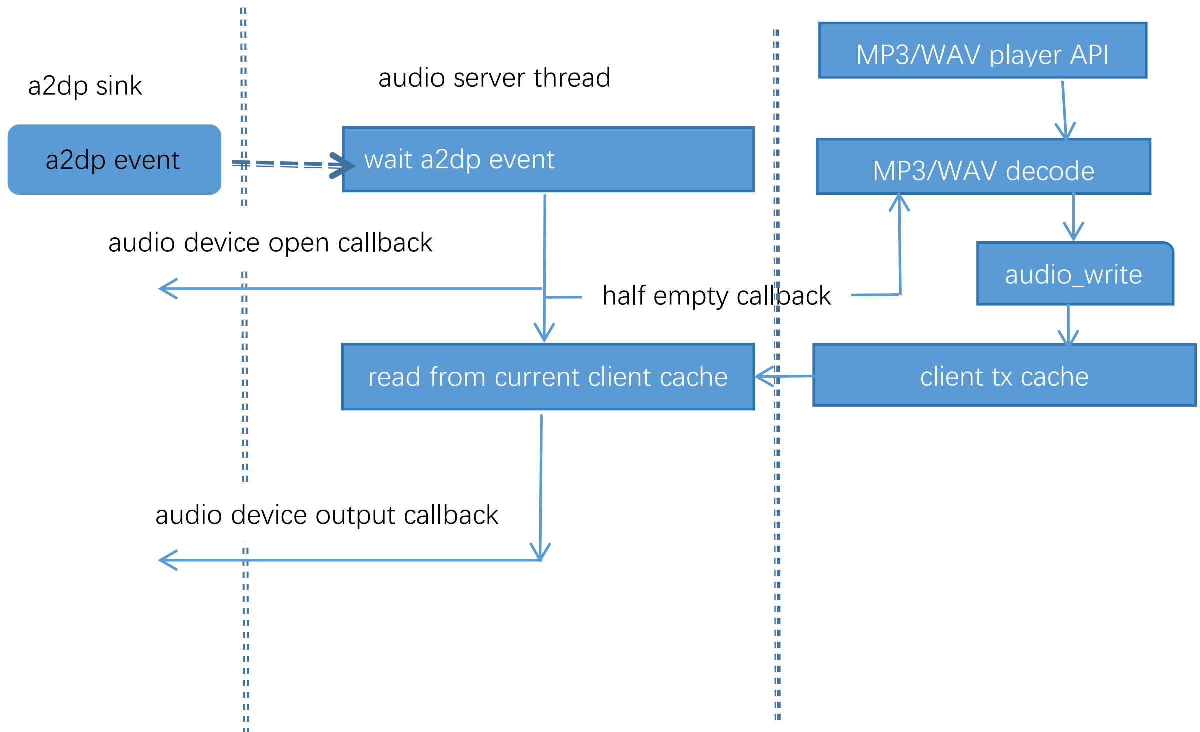
3.3 A2DP sink 音乐播放 speaker 数据播放流程

手表作为 a2dp sink， 音频设备为 speaker



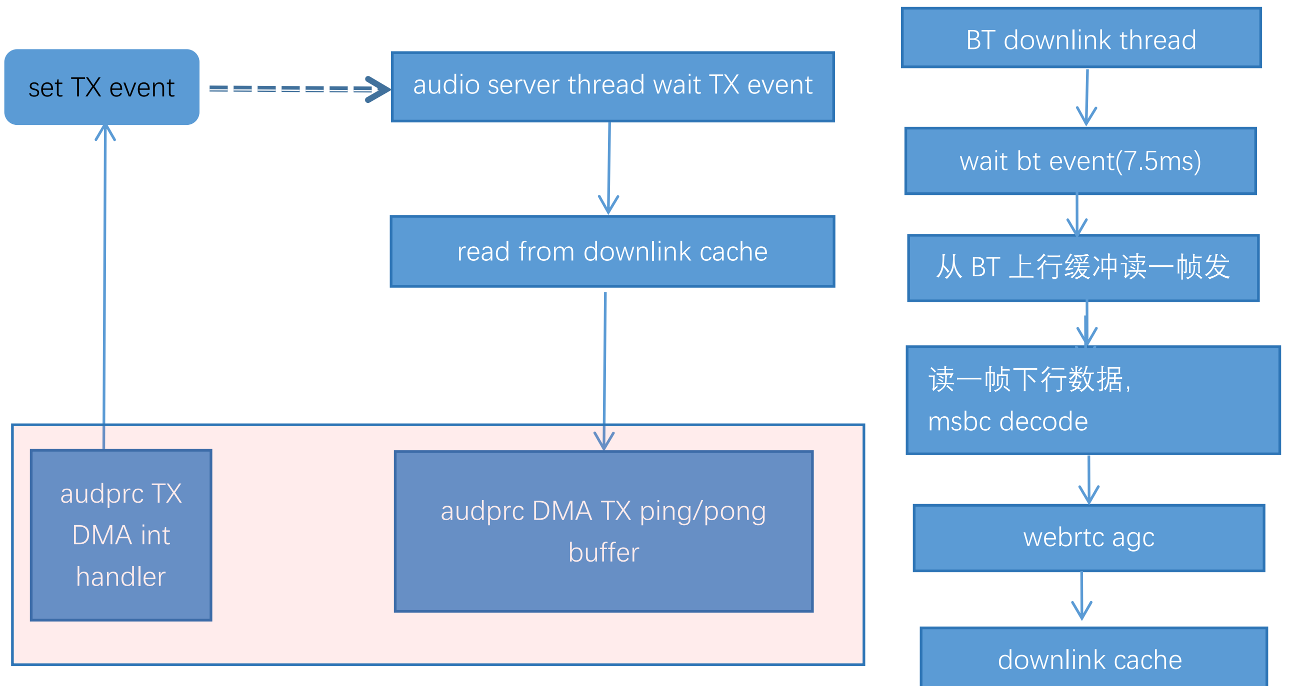
3.4 本地音乐到蓝牙耳机数据流程

手表作为 a2dp source，音频设备为 A2DP sink

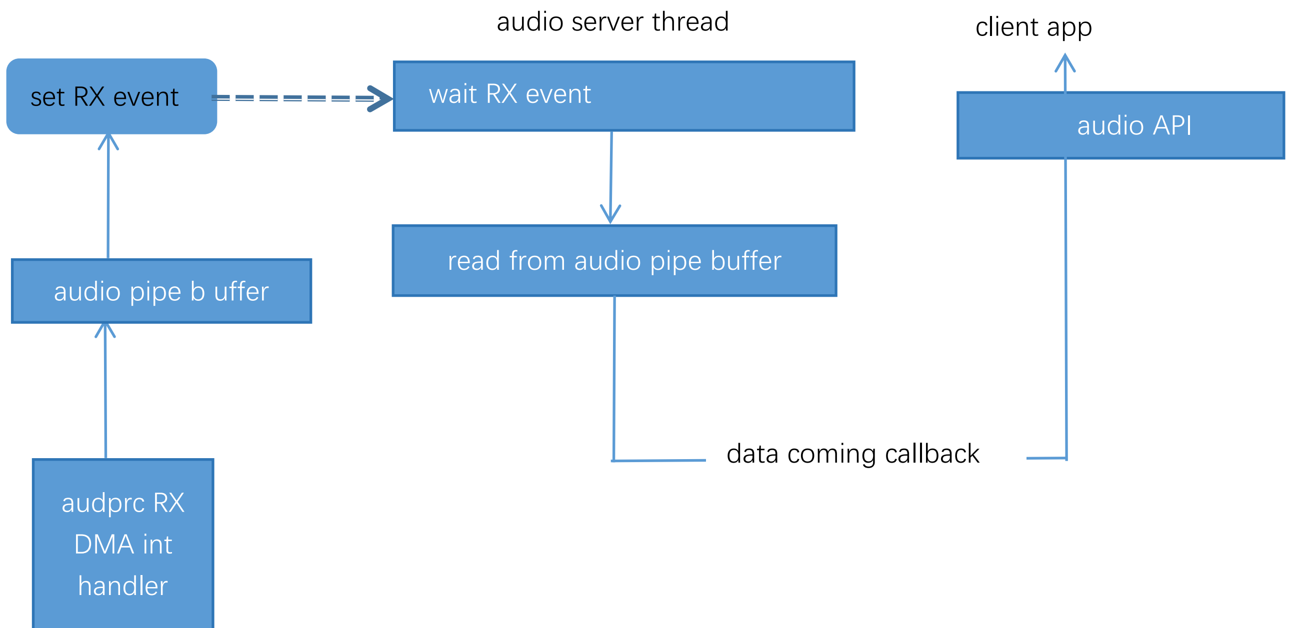


3.5 BT voice 播放到 speaker 数据播放流程

音频设备为 speaker， BT 并没有使用 audio API，直接用消息和 ring buffer 交互的



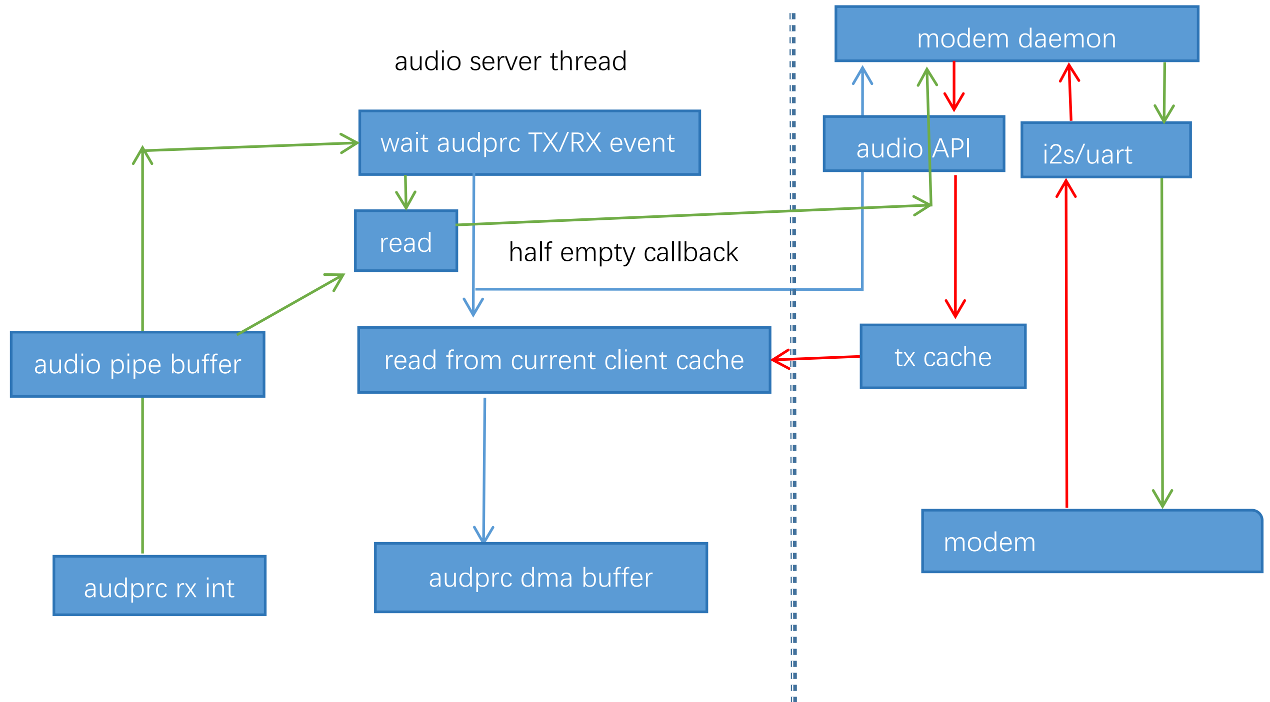
3.6 录音数据播放流程



3.7 modem 到 speaker/mic 数据流程

音频设备选择为 speaker。

绿色为 mic 数据走向，先到 pipe buffer，然后发事件通知 server，server 从 pipe buffer 读取后，用 audio_open()时记录的 callback，像 client 端（modem daemon）发送 data coming 消息和数据，然后 client 在往 modem 发送。 红色为下行到喇叭，和播放音乐原理一样。



3.8 modem 到蓝牙耳机流程

音频设备选择为 bt hfp, modem 部分和上一节一样，区别是不走 speaker 了，而是替换为注册的 bt hfp 设备。 下行数据和 a2dp sink 那个音频设备类似， 区别是注册这个音频设备是在音频设备的.open()里有个回调函数要收数据，和 a2dp sink 一样，只不过这个这个回调要处理 data coming 消息，收到蓝牙耳机发来的语音后，再发送给 modem client

4. EQ 调试

EQ 调试有专门的工具和文档，工具生成的代码在 drv_audprc.c 文件里。
电话和音乐的 EQ 数据是分开的。
g_adc_volume 为 mic 的增益，单位是 1db
g_tel_max_vol 是电话的最大音量，单位是 0.5db
g_music_max_vol 是音乐的最大音量，单位是 0.5db
g_tel_vol_level[]和 g_tel_music_level[]分别是电话和音乐的 16 个音量等级对应的音量，是 1db 为单位，和最大音量不同。
如果比上面对应的最大音量大，则使用最大音量，最小为-54，比-54 小就是静音了，
这些可以用 eq 工具调试。

如果要关闭 EQ，可以直接软件修改 audio_serverl.c 里，把调 bf0_audprc_eq_enable_offline(1) 的地方改为 bf0_audprc_eq_enable_offline(0);

```
773:
774:        /*set eq before device open*/
775:        if (client->audio_type == AUDIO_TYPE_BT_VOICE)
776:            bf0_audprc_eq_enable_offline(1);
777:        else if (client->audio_type == AUDIO_TYPE_BT_MUSIC)
778:            bf0_audprc_eq_enable_offline(1);
779:        else
780:            bf0_audprc_eq_enable_offline(1);
781:
```

更新历史

日期	版本	发布说明
2023 年 7 月 27 日	V0.1	Draft 版本