Audio Server Design Document

V0.1

2023/7/27

思澈科技（上海）有限公司

www.sifli.com

Contents

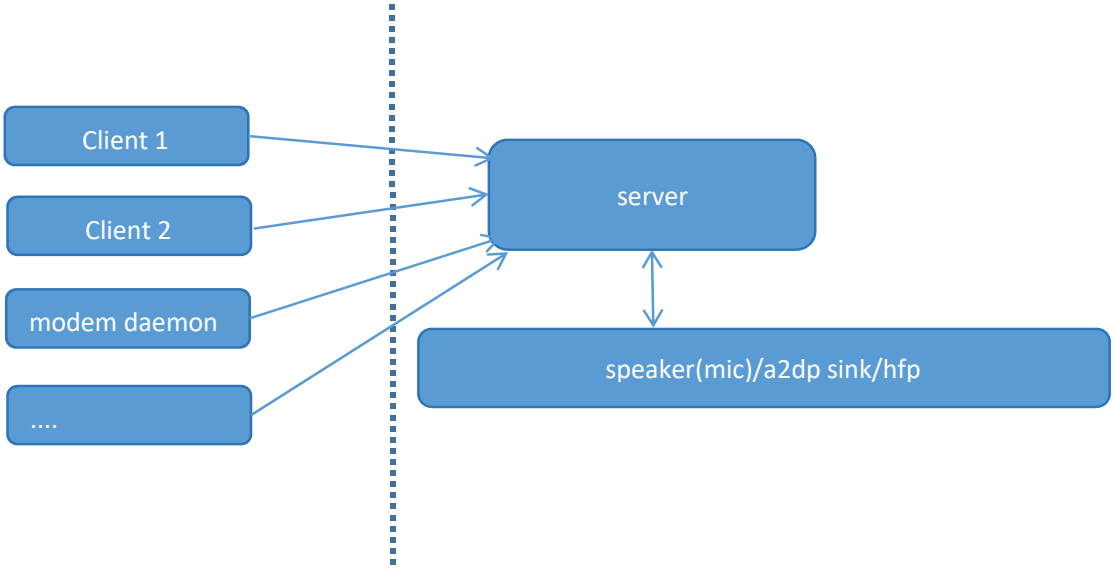
1 Purpose

This document describes the APIs and message processing provided by Audio. If you want to manage audio devices independently without using the audio server, you can refer to the underlying drivers called by the audio server and call them directly for management.

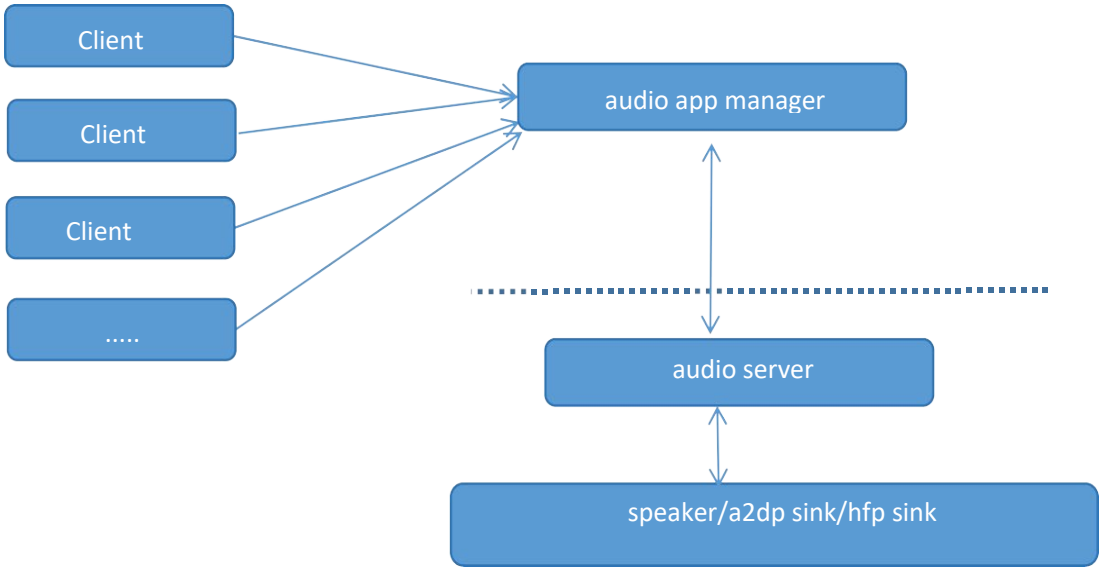
2 Audio Framework

2.1 Software Architecture

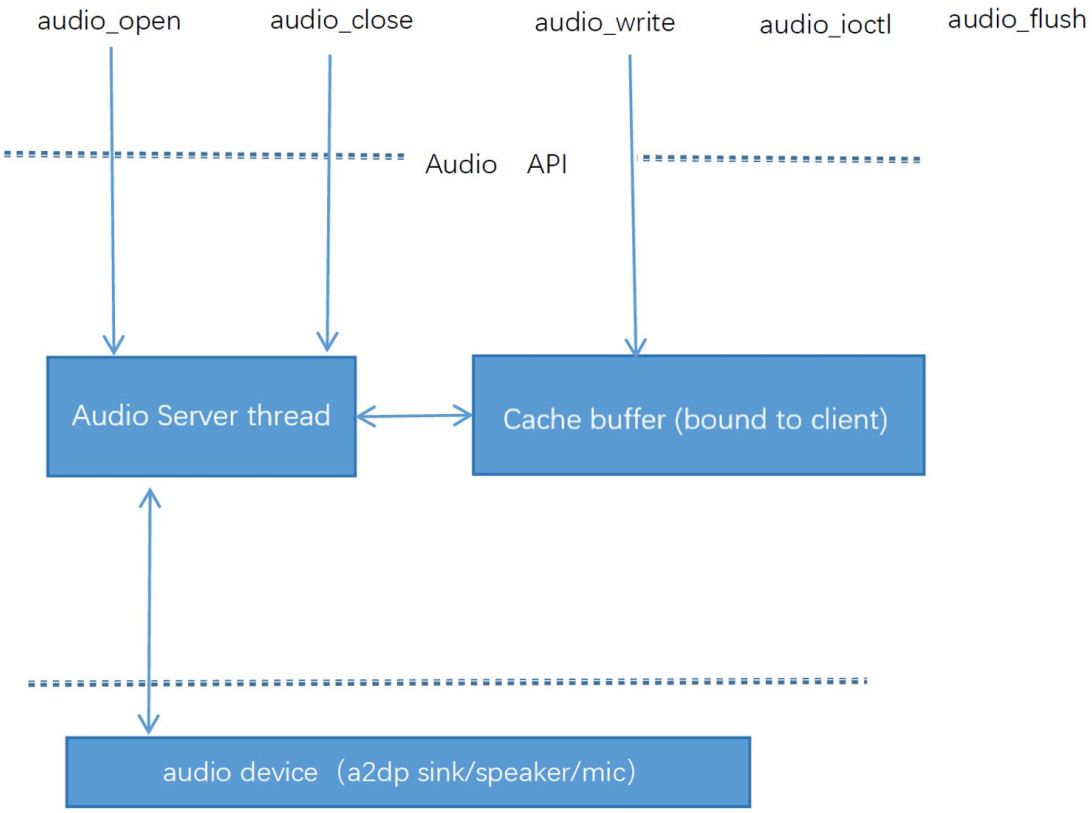
There is only one audio hardware resource. A client/server model is used to manage hardware resources. When multiple APPs access audio, each APP acts as a client. First, it opens and initiates a usage request. The server arbitrates whether to allow usage. High-priority requests interrupt low-priority ones, placing low-priority requests in a suspend queue. When a high-priority request closes, the highest priority request is restored from the suspend queue. Due to memory constraints in actual design, the solution side sometimes actively closes the current request and opens a new one. In the diagram below, the left side of the dotted line represents clients (sound sources), which are active, while the right side represents destinations that need to be selected. Both sides may handle bidirectional audio data.



As the number of audio input and output sources increases, different customers may have different priority requirements for apps. Priority arbitration may need to be separated from the server. The server would only be responsible for hardware resource usage, while inter-app priority would be separated from the server to upper-layer processing. Currently, this separation has not been implemented. The mic is selected based on the read/write flag when opening speaker(mic). It does not support simultaneous independent opening of speaker and mic.



2.2 Client 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);

Parameters:

- audio_type - Sound type, internally prioritized based on type
- rwflag - Read/write flag
- parameter - Sets sound channel sample rate, buffer size, etc.
- callback - Callback for suspend, resume, recording data arrival, buffer half-empty/empty messages
- callback_userdata - User data for callback

Returns Client handle

Priority is determined in this array. Currently, mixing is not supported. Higher numbers have higher priority (maximum 255). Higher priority can interrupt lower priority.

```
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);

Closes the handle.

2.2.3 audio_write

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

Writes data to the buffer corresponding to the handle.

2.2.4 audio_flush

Currently using ring buffer for data transfer, not queue buffer mode. Each write cannot correspond to a callback for completion of that data playback - only callbacks for buffer half-empty or empty. The final data needs flush to complete. Considering performance issues, an empty buffer doesn't mean upper layer playback is complete. The upper layer needs to actively flush and wait for the lower layer to finish playing all buffered content.

2.3 Volume API

Volume is divided into public and private types. There is only one public volume, and each music type has its own private volume.
If a music type hasn't set its private volume, it uses the public volume when playing. Otherwise, it uses its own private volume. The provided volume API settings are stored in memory and will be lost on restart.

2.3.1 Set public volume

int audio_server_set_public_volume(uint8_t volume);

volume - Range [0, 15]

2.3.2 Set private volume

int audio_server_set_private_volume(audio_type_t audio_type, uint8_t volume);

2.3.3 Set speaker mute

Does not distinguish music types

int audio_server_set_public_speaker_mute(uint8_t is_mute);

2.3.4 Set mic mute

int audio_server_set_public_mic_mute(uint8_t is_mute);

2.4 Set Audio Output Device

Local audio may go through speaker, a2dp sink, or TWS earphones - this is referred to as audio device selection.

Audio device switching is currently designed to occur within the server - closing the old one and opening the new one. This automatic switching has not been implemented in the server yet. The upper layer needs to first close the old audio_open handle, then select and open the device. Device selection has the following APIs. Like volume, devices support public and private settings by music type. If a type hasn't set a device, it uses the public one; if set, it uses the private one.

Audio is divided into input and output. Previous inputs were either local music or BT HFP/a2dp source, all directly calling audio interfaces. When BT interrupts arrive, the server calls their interfaces to retrieve data. Some voice communications are bidirectional and can be divided into remote and local ends. The device settings here only apply to the local end, though voice may be bidirectional on both sides. Modem is counted as remote and is not set through device interfaces. It can have a modem type in music types, distinguished through the music type variable in audio_open(audio_type).

From the watch's perspective, the currently planned remote and local definitions are:

- Remote: modem, local music file player, phone Bluetooth voice, phone Bluetooth music
- Local: speaker/mic, Bluetooth headset phone voice, Bluetooth headset music

Device selection is for the local end. Remote implementation is not in the server - remote device types are selected by apps or modules using audio_open().

2.4.1 Register audio device function

```
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);

};
```

The callback in open must be implemented in the server for each type to handle messages from audio devices like data coming, buffer empty, etc.

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

Among them, open requires an audio_device_input_callback callback function to be passed when calling, which is used to accept the device
End message events and input data events. The speaker is registered by default.
After this function is called, the server will close the old audio device and call the open() callback function of the new audio device
Open() is a callback function passed to the local implementation of this type of audio device, which is used to receive messages, including data arrival and sound
Messages such as buffer half empty of the frequency device are usually received by the audio device only when the buffer half empty message is received
Send data in the output() callback.

2.4.2 Select audio device function

```
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_device(audio_device_e device_type);
```

If this function has never been called after startup, the default is AUDIO_DEVICE_SPEAKER.

2.5 Special API

```
void bt_rx_event_to_audio_server()
```

When BT voice downlink data arrives, BT calls this function. In this function, you can send a message to the BT downlink data processing thread, which then reads and processes BT voice data upon receiving the message.

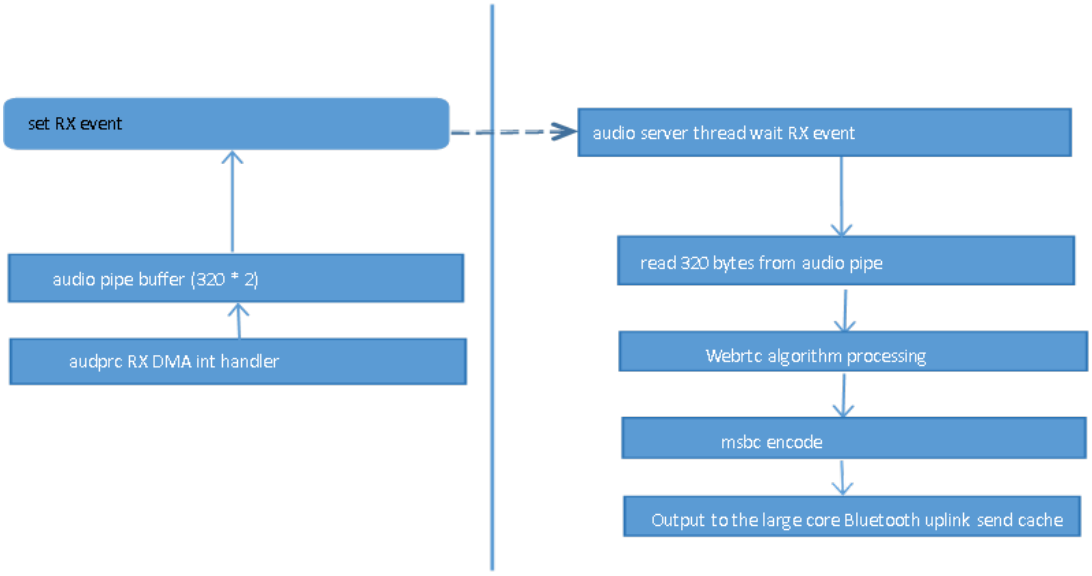
```
void audio_server_register_listener( audio_server_listener_func func,
                                     uint32_t what_to_listen,
                                     uint32_t reserved );
```

A function requested by the early solution to monitor some play/stop events to check audio status. Currently, the solution needs an audio management module so it can know audio status and match with UI without relying on callbacks, providing more centralized UI state management.

3. Message Data Flow

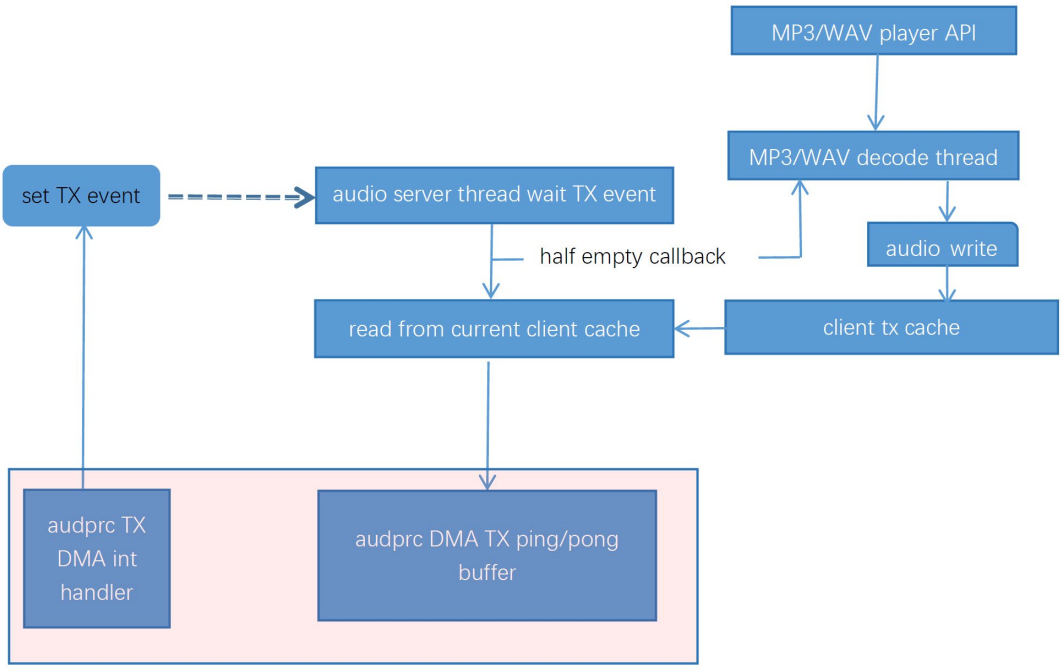
3.1 Mic data collection flow

Valid when audio device is speaker



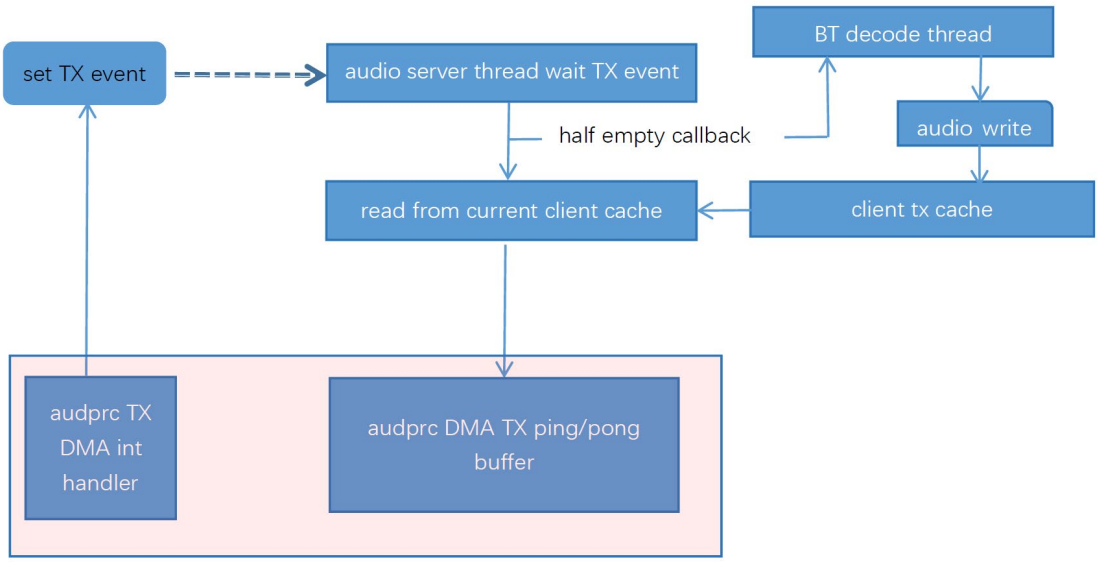
3.2 Local music playback to speaker data flow

Valid when audio device is speaker



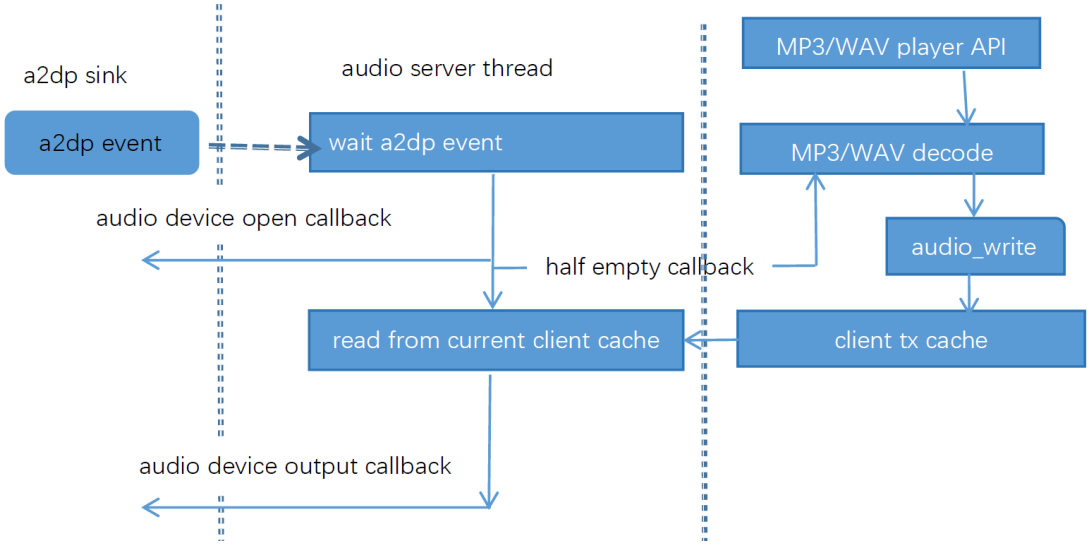
3.3 A2DP sink music playback to speaker data flow

Watch acts as a2dp sink, audio device is speaker



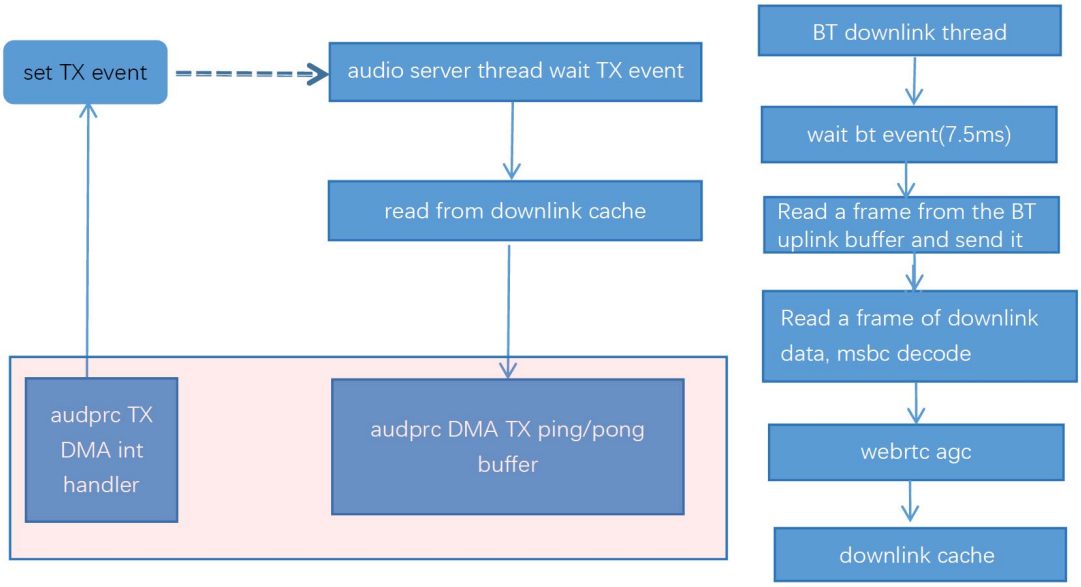
3.4 Local music to Bluetooth headset data flow

Watch acts as a2dp source, audio device is A2DP sink

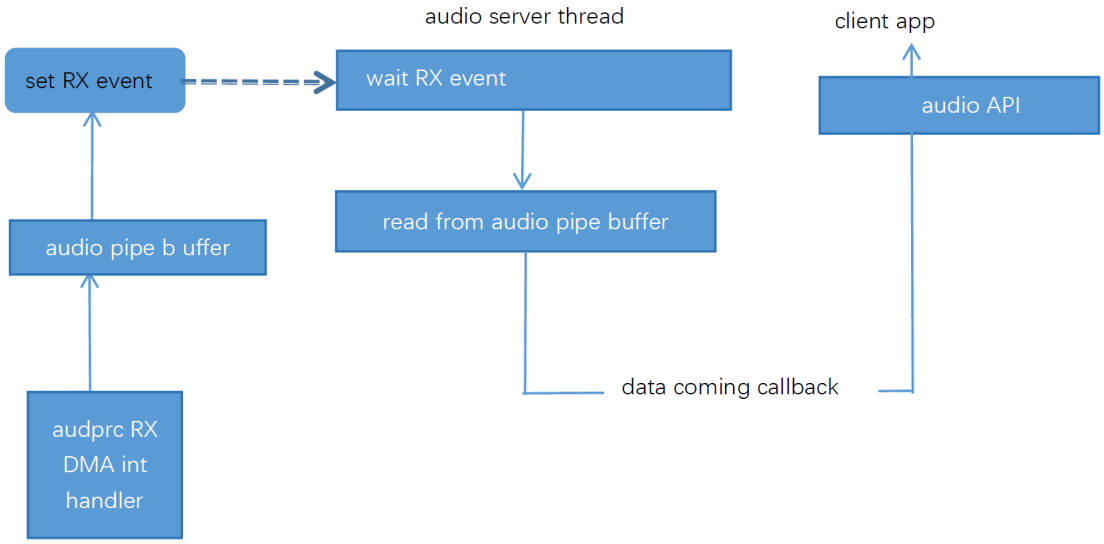


3.5 BT voice playback to speaker data flow

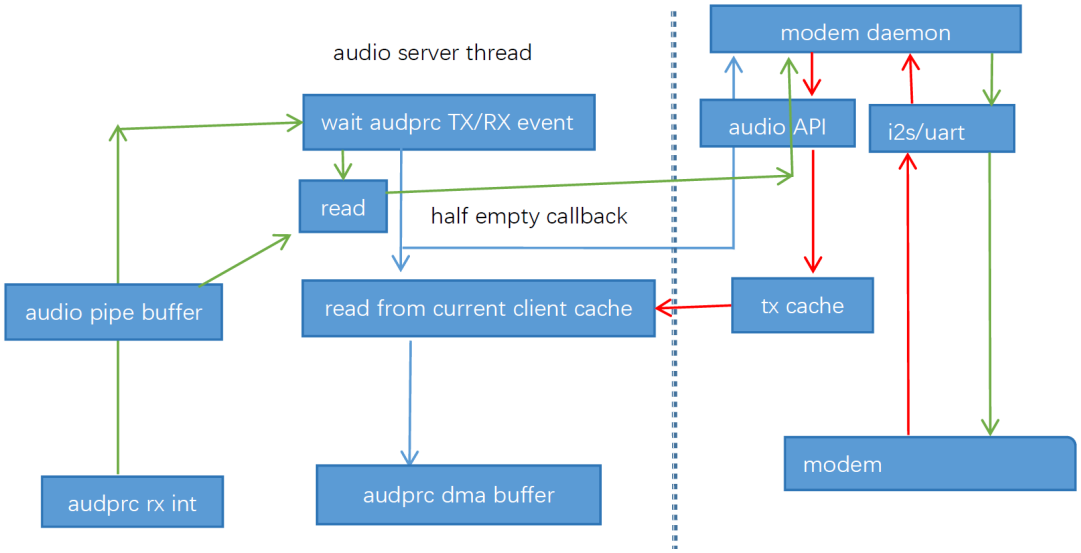
Audio device is speaker. BT doesn't use audio API, it interacts directly using messages and ring buffer



3.6 Recording data playback flow



3.7 Modem to speaker/mic data flow



3.8 Modem to Bluetooth headset flow

Audio device selected as bt hfp. Modem part is same as previous section, difference is it doesn't go through speaker but is replaced by registered bt hfp device. Downlink data is similar to a2dp sink audio device, difference is when registering this audio device, there's a callback function in the audio device's .open() to receive data, same as a2dp sink, except this callback needs to handle data coming messages. After receiving voice from Bluetooth headset, it sends to modem client.

4. EQ Tuning

EQ tuning has dedicated tools and documentation. Code generated by the tool is in the drv_audprc.c file.
Phone and music EQ data are separate.

- g_adc_volume - Mic gain, unit is 1dB
- g_tel_max_vol - Maximum phone volume, unit is 0.5dB
- g_music_max_vol - Maximum music volume, unit is 0.5dB
- g_tel_vol_level[] and g_tel_music_level[] - Volume levels for 16 volume grades for phone and music respectively, in 1dB units, different from maximum volume

If greater than the corresponding maximum volume above, the maximum volume is used. Minimum is -54; below -54 is muted. These can be tuned using the EQ tool.
To disable EQ, you can modify audio_server.c directly, changing
To disable EQ, you can modify audio_server.c directly,
changing bf0_audprc_eq_enable_offline(1) to 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);
```

Update History

Date	Version	Release Notes
July 27, 2023	V0.1	Draft version