AudioPolicy :

1. The first step is to parse the xml file related to the audio policy.

virtio {

global\_configuration {

attached\_output\_devices AUDIO\_DEVICE\_OUT\_SPEAKER

default\_output\_device AUDIO\_DEVICE\_OUT\_SPEAKER

attached\_input\_devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC

audio\_hal\_version 2.0

}

outputs {

virtio {

sampling\_rates 48000

channel\_masks AUDIO\_CHANNEL\_OUT\_STEREO

formats AUDIO\_FORMAT\_PCM\_16\_BIT

devices AUDIO\_DEVICE\_OUT\_SPEAKER

flags AUDIO\_OUTPUT\_FLAG\_DIRECT

}

}

inputs {

virtio {

sampling\_rates 48000

channel\_masks AUDIO\_CHANNEL\_IN\_MONO

formats AUDIO\_FORMAT\_PCM\_16\_BIT

devices AUDIO\_DEVICE\_IN\_BUILTIN\_MIC

}

}

}

|  |
| --- |
|  |
|  | <!-- Virtio Audio HAL Audio Policy Configuration file --> |
|  | <module name="virtio" halVersion="2.0"> |
|  | <mixPorts> |
|  | <mixPort name="virtio output" role="source"> |
|  | <profile name="" format="AUDIO\_FORMAT\_PCM\_16\_BIT" |
|  | samplingRates="44100" |
|  | channelMasks="AUDIO\_CHANNEL\_OUT\_STEREO"/> |
|  | </mixPort> |
|  | <mixPort name="virtio input" role="sink"> |
|  | <profile name="" format="AUDIO\_FORMAT\_PCM\_16\_BIT" |
|  | samplingRates="44100,48000" |
|  | channelMasks="AUDIO\_CHANNEL\_IN\_MONO,AUDIO\_CHANNEL\_IN\_STEREO"/> |
|  | </mixPort> |
|  | </mixPorts> |
|  | <devicePorts> |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | <devicePort tagName="Virtio Headphones" type="AUDIO\_DEVICE\_OUT\_VIRTIO\_HEADPHONES" role="sink"> |
|  | <profile name="" format="AUDIO\_FORMAT\_PCM\_16\_BIT" |
|  | samplingRates="44100" |
|  | channelMasks="AUDIO\_CHANNEL\_OUT\_STEREO"/> |
|  | </devicePort> |
|  | <devicePort tagName="virtio Speaker" type="AUDIO\_DEVICE\_OUT\_ VIRTIO\_SPEAKER" role="sink"> |
|  | <profile name="" format="AUDIO\_FORMAT\_PCM\_16\_BIT" |
|  | samplingRates="44100" |
|  | channelMasks="AUDIO\_CHANNEL\_OUT\_STEREO"/> |
|  | </devicePort> |
|  | <devicePort tagName="virtio In" type="AUDIO\_DEVICE\_IN\_VIRTIO\_MIC" role="source"> |
|  | <profile name="" format="AUDIO\_FORMAT\_PCM\_16\_BIT" |
|  | samplingRates="44100,48000" |
|  | channelMasks="AUDIO\_CHANNEL\_IN\_MONO,AUDIO\_CHANNEL\_IN\_STEREO"/> |
|  | </devicePort> |
|  | </devicePorts> |
|  | <routes> |
|  | <route type="mix" sink="VIRTIO Out" |
|  | sources="virtio output"/> |
|  | <route type="mix" sink="VIRTIO Headphones" |
|  | sources=" virtio output"/> |
|  | <route type="mix" sink="Virtio Speaker" |
|  | sources=" virtio output"/> |
|  | <route type="mix" sink=" virtio input" |
|  | sources="VIRTIO In"/> |
|  | </routes> |
|  | </module> |

1. The second step is to load the corresponding audio hal according to the hal name in the xml,

init.rc-> audioserver.rc-> main\_audioserver.cpp

path : /[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[media](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/)/[audioserver](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/audioserver/)/ audioserver.rc

main\_audioserver.cpp

Start the audio flinger and audio policy instance

[AudioFlinger](http://androidxref.com/9.0.0_r3/s?defs=AudioFlinger&project=frameworks)::[instantiate](http://androidxref.com/9.0.0_r3/s?defs=instantiate&project=frameworks)();

[AudioPolicyService](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyService&project=frameworks)::[instantiate](http://androidxref.com/9.0.0_r3/s?defs=instantiate&project=frameworks)();

[**AudioPolicyService**](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyService&project=frameworks)**:**

File Path:/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audiopolicy](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/)/[service](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/)/[AudioPolicyService.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/AudioPolicyService.cpp)

**void** [**AudioPolicyService**](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/AudioPolicyService.cpp#AudioPolicyService)::**[onFirstRef](http://androidxref.com/9.0.0_r3/s?refs=onFirstRef&project=frameworks)**()-> Create audio policy manager object.

[mAudioPolicyManager](http://androidxref.com/9.0.0_r3/s?defs=mAudioPolicyManager&project=frameworks) = [createAudioPolicyManager](http://androidxref.com/9.0.0_r3/s?defs=createAudioPolicyManager&project=frameworks)([mAudioPolicyClient](http://androidxref.com/9.0.0_r3/s?defs=mAudioPolicyClient&project=frameworks));

File path /[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audiopolicy](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/)/[managerdefault](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/managerdefault/)/[AudioPolicyManager.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp)

**[AudioPolicyManager](http://androidxref.com/9.0.0_r3/s?refs=AudioPolicyManager&project=frameworks)**::**[AudioPolicyManager](http://androidxref.com/9.0.0_r3/s?refs=AudioPolicyManager&project=frameworks)**([AudioPolicyClientInterface](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyClientInterface&project=frameworks) \***[clientInterface](http://androidxref.com/9.0.0_r3/s?refs=clientInterface&project=frameworks)**, **bool** /\*forTesting\*/)🡪 Constructor

1. [AudioPolicyManager](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyManager&project=frameworks)::**[loadConfig](http://androidxref.com/9.0.0_r3/s?refs=loadConfig&project=frameworks)**()

It will load vendor and system config files .

1. [AudioPolicyManager](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyManager&project=frameworks)::**[initialize](http://androidxref.com/9.0.0_r3/s?refs=initialize&project=frameworks)**()
   * Once policy config has been parsed, retrieve an instance of the engine and initialize it.
   * Retrieve the Policy Manager Interface
   * mAvailableOutputDevices and mAvailableInputDevices now contain all attached devices, open all output streams needed to access attached devices

**Audio device opening:**

File path :

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audioflinger](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audioflinger/)/[AudioFlinger.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audioflinger/AudioFlinger.cpp)

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[media](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/)/[libaudiohal](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/libaudiohal/)/[4.0](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/libaudiohal/4.0/)/[DevicesFactoryHalLocal.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/media/libaudiohal/4.0/DevicesFactoryHalLocal.cpp)

 /[hardware](http://androidxref.com/9.0.0_r3/xref/hardware/)/[libhardware](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/)/[include](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/)/[hardware](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/)/[audio.h](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h)

[AudioPolicyService](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyService&project=frameworks)::[AudioPolicyClient](http://androidxref.com/9.0.0_r3/s?defs=AudioPolicyClient&project=frameworks)::**[loadHwModule](http://androidxref.com/9.0.0_r3/s?refs=loadHwModule&project=frameworks)**(**const** **char** \***[name](http://androidxref.com/9.0.0_r3/s?refs=name&project=frameworks)**)🡪

[af](http://androidxref.com/9.0.0_r3/s?defs=af&project=frameworks)->**[loadHwModule](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/AudioPolicyClientImpl.cpp" \l "loadHwModule)**([**name**](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/AudioPolicyClientImpl.cpp#name));🡪 [**AudioFlinger**](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audioflinger/AudioFlinger.cpp#AudioFlinger)::**[loadHwModule\_l](http://androidxref.com/9.0.0_r3/s?refs=loadHwModule_l&project=frameworks)**(**const** **char** \*[**name**](http://androidxref.com/9.0.0_r3/s?refs=name&project=frameworks))🡪

**int** **[rc](http://androidxref.com/9.0.0_r3/s?refs=rc&project=frameworks)** = [mDevicesFactoryHal](http://androidxref.com/9.0.0_r3/s?defs=mDevicesFactoryHal&project=frameworks)->[openDevice](http://androidxref.com/9.0.0_r3/s?defs=openDevice&project=frameworks)([name](http://androidxref.com/9.0.0_r3/s?defs=name&project=frameworks), &[dev](http://androidxref.com/9.0.0_r3/s?defs=dev&project=frameworks));🡪

[**audio\_hw\_device\_open**](http://androidxref.com/9.0.0_r3/s?refs=audio_hw_device_open&project=hardware)(**const** **struct** [hw\_module\_t](http://androidxref.com/9.0.0_r3/s?defs=hw_module_t&project=hardware)\* **[module](http://androidxref.com/9.0.0_r3/s?refs=module&project=hardware)**,

[804](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h" \l "804) **struct** [**audio\_hw\_device**](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h#audio_hw_device)\*\* [device](http://androidxref.com/9.0.0_r3/s?defs=device&project=hardware))🡪

[**module**](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h#module)->[methods](http://androidxref.com/9.0.0_r3/s?defs=methods&project=hardware)->[open](http://androidxref.com/9.0.0_r3/s?defs=open&project=hardware)([**module**](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h#module), [**AUDIO\_HARDWARE\_INTERFACE**](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h#AUDIO_HARDWARE_INTERFACE),

[807](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware/include/hardware/audio.h" \l "807) [TO\_HW\_DEVICE\_T\_OPEN](http://androidxref.com/9.0.0_r3/s?defs=TO_HW_DEVICE_T_OPEN&project=hardware)([device](http://androidxref.com/9.0.0_r3/s?defs=device&project=hardware)));

-🡪 Open hardware module

   //This is the interface provided by the HAL framework. This method is in Hardware.c. Get the corresponding module through this AUDIO\_HARDWARE\_MODULE\_ID.

   rc = hw\_get\_module\_by\_class(AUDIO\_HARDWARE\_MODULE\_ID, if\_name,&mod);

  //Open the corresponding device through the corresponding module.

   rc = audio\_hw\_device\_open(mod, dev);

**Code Changes related to audio Policy :**

1. /[hardware](http://androidxref.com/9.0.0_r3/xref/hardware/)/[libhardware\_legacy](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/)/[audio](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/audio/)/[audio\_hw\_hal.cpp](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/audio/audio_hw_hal.cpp)

**In above file**

**static** [uint32\_t](http://androidxref.com/9.0.0_r3/s?defs=uint32_t&project=hardware) **[audio\_device\_conv\_table](http://androidxref.com/9.0.0_r3/s?refs=audio_device_conv_table&project=hardware)**[][[**HAL\_API\_REV\_NUM**](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/audio/audio_hw_hal.cpp#HAL_API_REV_NUM)] =

{ [AudioSystem](http://androidxref.com/9.0.0_r3/s?defs=AudioSystem&project=hardware)::[DEVICE\_OUT\_SPEAKER](http://androidxref.com/9.0.0_r3/s?defs=DEVICE_OUT_SPEAKER&project=hardware), [AUDIO\_DEVICE\_OUT\_SPEAKER](http://androidxref.com/9.0.0_r3/s?defs=AUDIO_DEVICE_OUT_SPEAKER&project=hardware) },

1. /[system](http://androidxref.com/9.0.0_r3/xref/system/)/[media](http://androidxref.com/9.0.0_r3/xref/system/media/)/[audio](http://androidxref.com/9.0.0_r3/xref/system/media/audio/)/[include](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/)/[system](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/)/[audio-base.h](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h)

**enum** {

[290](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "290) ***[AUDIO\_DEVICE\_NONE](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_NONE&project=system)*** = 0x0u,

[291](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "291) ***[AUDIO\_DEVICE\_BIT\_IN](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_BIT_IN&project=system)*** = 0x80000000u,

[292](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "292) ***[AUDIO\_DEVICE\_BIT\_DEFAULT](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_BIT_DEFAULT&project=system)*** = 0x40000000u,

[293](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "293)

[294](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "294) ***[AUDIO\_DEVICE\_OUT\_EARPIECE](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_EARPIECE&project=system)*** = 0x1u,

[295](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "295) ***[AUDIO\_DEVICE\_OUT\_SPEAKER](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_SPEAKER&project=system)*** = 0x2u,

[296](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "296) ***[AUDIO\_DEVICE\_OUT\_WIRED\_HEADSET](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_WIRED_HEADSET&project=system)*** = 0x4u,

[297](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "297) ***[AUDIO\_DEVICE\_OUT\_WIRED\_HEADPHONE](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_WIRED_HEADPHONE&project=system)*** = 0x8u,

[298](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "298) ***[AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_BLUETOOTH_SCO&project=system)*** = 0x10u,

[299](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "299) ***[AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_HEADSET](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_BLUETOOTH_SCO_HEADSET&project=system)*** = 0x20u,

[300](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base.h" \l "300) ***[AUDIO\_DEVICE\_OUT\_BLUETOOTH\_SCO\_CARKIT](http://androidxref.com/9.0.0_r3/s?refs=AUDIO_DEVICE_OUT_BLUETOOTH_SCO_CARKIT&project=system)*** = 0x40u,

Same as above update the eum for virtio devices.

1. /[hardware](http://androidxref.com/9.0.0_r3/xref/hardware/)/[libhardware\_legacy](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/)/[include](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/include/)/[hardware\_legacy](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/include/hardware_legacy/)/[AudioSystemLegacy.h](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/include/hardware_legacy/AudioSystemLegacy.h)
2. /[system](http://androidxref.com/9.0.0_r3/xref/system/)/[media](http://androidxref.com/9.0.0_r3/xref/system/media/)/[audio](http://androidxref.com/9.0.0_r3/xref/system/media/audio/)/[include](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/)/[system](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/)/[audio-base-utils.h](http://androidxref.com/9.0.0_r3/xref/system/media/audio/include/system/audio-base-utils.h)

**enum** ***[audio\_devices](http://androidxref.com/9.0.0_r3/s?refs=audio_devices&project=hardware)*** {

[231](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/include/hardware_legacy/AudioSystemLegacy.h" \l "231) // output devices

[232](http://androidxref.com/9.0.0_r3/xref/hardware/libhardware_legacy/include/hardware_legacy/AudioSystemLegacy.h" \l "232) ***[DEVICE\_OUT\_EARPIECE](http://androidxref.com/9.0.0_r3/s?refs=DEVICE_OUT_EARPIECE&project=hardware)*** = 0x1,

Same as above update the eum audio\_device for virtio devices.

PlayBack flow :

Audio Track 🡪 2 modes

1. MODE\_STATIC🡪the playback data to AudioTrack at one time,
2. MODE\_STREAM🡪 The user process needs to continue to call write() to write data to the FIFO

AudioTrack Native API four data transmission modes:

1. TRANSFER\_CALLBACK: In the AudioTrackThread thread, use the audioCallback callback function to actively request data from the application process. ToneGenerator uses this mode.
2. TRANSFER\_OBTAIN: In process needs to call obtainBuffer()/releaseBuffer() to fill in the data.
3. TRANSFER\_SYNC: The application process needs to continuously call write() to write data to the FIFO. When writing data, it may encounter blocking (waiting for AudioFlinger::PlaybackThread to consume the previous data), which is basically applicable to all audio scenarios; corresponds to the MODE\_STREAM mode of AudioTrack Java API
4. TRANSFER\_SHARED: The application process pays the playback data to AudioTrack at a time, which is suitable for scenarios with small data volume and high latency requirements; corresponding to the MODE\_STATIC mode of AudioTrack Java API

AudioTrack Native API audio stream type:

1. AUDIO\_STREAM\_VOICE\_CALL
2. AUDIO\_STREAM\_SYSTEM
3. AUDIO\_STREAM\_RING
4. AUDIO\_STREAM\_MUSIC
5. AUDIO\_STREAM\_ALARM
6. AUDIO\_STREAM\_NOTIFICATION
7. AUDIO\_STREAM\_DTMF

**AudioTrack Native API output identification:**

AUDIO\_OUTPUT\_FLAG\_DIRECT:   
Indicates that the audio stream is directly output to the audio device, without software mixing, and is generally used for HDMI device sound output

AUDIO\_OUTPUT\_FLAG\_PRIMARY: Indicates that the audio stream needs to be output to the main output device, generally used for ringtones

AUDIO\_OUTPUT\_FLAG\_FAST: Indicates that the audio stream needs to be quickly output to the audio device, which is generally used in scenes with high delay requirements such as key-press sound and game background sound

AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER: Indicates that the audio stream output can accept a large delay, which is generally used in scenes that do not require high delays such as music and video playback.

AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD: Indicates that the audio stream has not been decoded by software and needs to be output to a hardware decoder for decoding.

For example, button sounds and game background sounds require high output delay. Then you need to set AUDIO\_OUTPUT\_FLAG\_FAST.

Frame : a frame represents a complete sound unit, and the so-called sound unit refers to a sample sample;

 frameSize = channelCount \* bytesPerSample.

transmission delay: transmission delay represents the transmission time of one cycle of audio data.

latency = periodSize / sampleRate.

audio resampling: Audio resampling refers to the process of converting data of one sampling rate into data of another sampling rate. On the Android native system, audio hardware devices generally work at a fixed sampling rate (such as 48 KHz), so all audio track data needs to be resampled to this fixed sampling rate, and then output.

MimFrameCount :

minFrameCount = (afFrameCount \* sampleRate / afSampleRate) \* minBufCount.

AudioFlinger:

Cpp files and its explanation

**AudioResampler.cpp:**  Resampling processing class, which can perform sample rate conversion and channel conversion; directly used by the recording thread AudioFlinger::RecordThread.

**AudioMixer.cpp** : **Audio** mixing processing class, including resampling, volume adjustment, channel conversion, etc. The resampling is multiplexed with AudioResampler; it is used directly by the playback thread AudioFlinger::MixerThread

**Effects.cpp** : Sound effect processing class

**Tracks.cpp** : Audio stream management class, which can control the state of the audio stream, such as start, stop, pause

**Threads.cpp** : the playback thread and recording thread class; the playback thread reads the playback data from the FIFO and mixes it, and then writes the data to the output stream device; the recording thread reads the recording data from the input stream device and resamples it, and then writes the data to FIFO

**AudioFlinger.cpp** : Service interface provided by AudioFlinger

## AudioFlinger service interface

It can be concluded that the main service requests that AudioFlinger responds to are:

* Get the configuration information of the hardware device
* Volume adjustment
* Silent operation
* Audio mode switch
* Audio parameter setting
* Input and output stream device management
* Audio stream management

## AudioFlinger playback recording thread:

## https://img-blog.csdn.net/20170301225332422?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvenl1YW55dW4=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast

AndioFlinger is responsible for the management of input and output stream devices and the processing and transmission of audio stream data.

* **ThreadBase** : Base class of PlaybackThread and RecordThread
* **RecordThread** : Recording thread class, derived from ThreadBase
* **PlaybackThread** : Playback thread base class, also derived from ThreadBase
* **MixerThread** : The mixing playback thread class, derived from PlaybackThread, is responsible for processing the audio streams identified as AUDIO\_OUTPUT\_FLAG\_PRIMARY, AUDIO\_OUTPUT\_FLAG\_FAST, AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER, MixerThread can mix the data of multiple audio tracks and then output
* **DirectOutputThread** : Direct input playback thread class, derived from PlaybackThread, responsible for processing the audio stream identified as AUDIO\_OUTPUT\_FLAG\_DIRECT. This kind of audio stream data does not require software mixing and can be directly output to the audio device
* **DuplicatingThread** : Duplicating playback thread class, derived from MixerThread, responsible for duplicating audio stream data to other output devices, use scenarios such as main sound card device, Bluetooth headset device, USB sound card device,**virtio sound card devices** and output at the same time
* **OffloadThread** : The hard-decoded playback thread class, derived from DirectOutputThread, is responsible for processing the audio stream identified as AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD. This audio stream has not been decoded by software (usually data in MP3, AAC, etc.) formats and needs to be output to a hardware decoder. Hardware decoder decodes into PCM data.

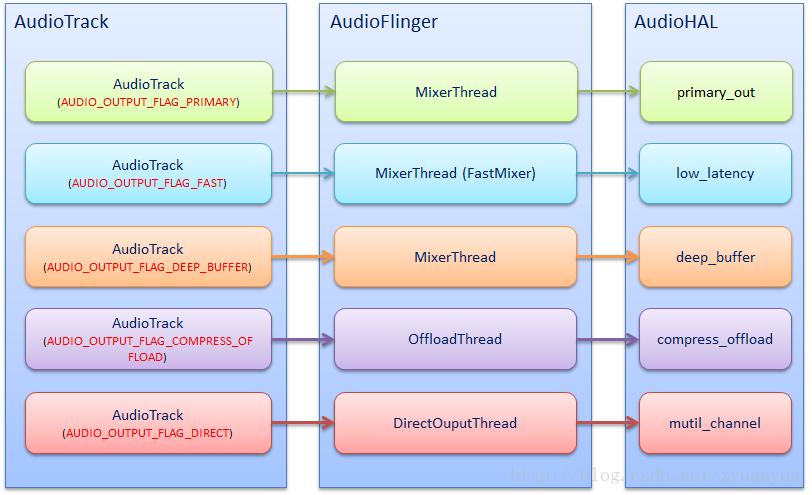
There is an extremely important function threadLoop() in PlaybackThread. When PlaybackThread is strongly referenced, threadLoop() will actually run into the loop body to process audio stream data related transactions.

1. The condition of the threadLoop() loop is that exitPending() returns false. If you want PlaybackThread to end the loop, you can call requestExit() to request exit;
2. processConfigEvents\_l(): Process configuration events; when a configuration change event occurs, you need to call sendConfigEvent\_l() to notify PlaybackThread, so that PlaybackThread can process configuration events in time; a common configuration event is to switch audio channels;
3. Check whether the standby conditions are met at this moment, for example, there is no Track in the ACTIVE state (mActiveTracks.size() = 0), then call threadLoop\_standby() to close the audio hardware device to save energy;
4. prepareTracks\_l(): Prepare audio stream and mixer. This function is very complicated. I will not analyze it in detail here, but just list the main points of the process:

* Traverse mActiveTracks, process the tracks on mActiveTracks one by one, and check whether the track is in the ACTIVE state;
* If the track setting is ACTIVE, then check whether the track data is ready;
* Configure the mixer parameters according to the volume value, format, number of channels, audio track sampling rate, and hardware device sampling rate of the audio stream;
* If the status of the Track is PAUSED or STOPPED, add the Track to the tracksToRemove vector;

1. threadLoop\_mix(): Read all the audio stream data in the ACTIVE state, and the mixer starts to process these data;
2. threadLoop\_write(): Write the data processed by the mixer to the output stream device;
3. threadLoop\_removeTracks(): Remove all tracks on tracksToRemove from mActiveTracks; this way these tracks will not be processed in the next loop.

From Audio HAL, we usually see the following 4 output stream devices, corresponding to different playback scenarios:

* **primary\_out** : primary output stream device, used for ringtone sound output, corresponding to the audio stream identified as AUDIO\_OUTPUT\_FLAG\_PRIMARY and a MixerThread playback thread instance
* **low\_latency** : low-latency output stream device, used for sound output with high latency requirements such as key-press sound and game background sound, corresponding to the audio stream identified as AUDIO\_OUTPUT\_FLAG\_FAST and a MixerThread playback thread instance
* **deep\_buffer** : Music audio track output stream device, used for audio output with low delay requirements such as music, corresponding to the audio stream identified as AUDIO\_OUTPUT\_FLAG\_DEEP\_BUFFER and a MixerThread playback thread instance
* **compress\_offload** : hard-decoded output stream device, used for data output that requires hardware decoding, corresponding to the audio stream identified as AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD and an OffloadThread playback thread instance
* 

When the system starts, the three output stream devices, primary\_out, low\_latency, and deep\_buffer, are already opened, and the corresponding MixerThread is created. At this time, DirectOutputThread and OffloadThread will not be created until the audio stream identified as AUDIO\_OUTPUT\_FLAG\_DIRECT/AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD needs to be output.

[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audiopolicy](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/)/[managerdefault](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/managerdefault/)/[AudioPolicyManager.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/managerdefault/AudioPolicyManager.cpp)

AudioPolicyManager::AudioPolicyManager(AudioPolicyClientInterface \*clientInterface)

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audiopolicy](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/)/[service](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/)/[AudioPolicyClientImpl.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audiopolicy/service/AudioPolicyClientImpl.cpp)

/[frameworks](http://androidxref.com/9.0.0_r3/xref/frameworks/)/[av](http://androidxref.com/9.0.0_r3/xref/frameworks/av/)/[services](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/)/[audioflinger](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audioflinger/)/[AudioFlinger.cpp](http://androidxref.com/9.0.0_r3/xref/frameworks/av/services/audioflinger/AudioFlinger.cpp)

Which mpClientInterface->openOutput()will eventually call to AudioFlinger::openOutput(): Open the output stream devices and create PlaybackThread objects:

**Bind the available output stream mpClientInterface->openOutput:**  
step 1. Specify the first opened device as the main output device

Call flow:

@\frameworks\av\services\audiopolicy\managerdefault\AudioPolicyManager.cpp

status\_t status = mpClientInterface->openOutput(outProfile->getModuleHandle(),

&output, &config, &outputDesc->mDevice, address, &outputDesc->mLatency, outputDesc->mFlags);

------------------------------------------------------------------------------------

@\frameworks\av\services\audiopolicy\service\AudioPolicyClientImpl.cpp

status\_t AudioPolicyService::AudioPolicyClient::openOutput(audio\_module\_handle\_t module,audio\_io\_handle\_t \*output,

audio\_config\_t \*config,audio\_devices\_t \*devices, const String8& address, uint32\_t \*latencyMs,audio\_output\_flags\_t flags)

{

sp<IAudioFlinger> af = AudioSystem::get\_audio\_flinger();

return af->openOutput(module, output, config, devices, address, latencyMs, flags);

}

@ \frameworks\av\media\libaudioclient\IAudioFlinger.cpp

virtual status\_t openOutput(audio\_module\_handle\_t module, audio\_io\_handle\_t \*output, audio\_config\_t \*config,

audio\_devices\_t \*devices, const String8& address, uint32\_t \*latencyMs, audio\_output\_flags\_t flags)

||

status\_t BnAudioFlinger::onTransact(

uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)

||

status\_t status = openOutput(module, &output, &config,

&devices, address, &latencyMs, flags);

------------------------------------------------------------------------------------

\frameworks\av\services\audioflinger\AudioFlinger.cpp

status\_t AudioFlinger::openOutput(audio\_module\_handle\_t module, audio\_io\_handle\_t \*output, audio\_config\_t \*config,

audio\_devices\_t \*devices, const String8& address, uint32\_t \*latencyMs, audio\_output\_flags\_t flags)

sp<AudioFlinger::ThreadBase> AudioFlinger::openOutput\_l(audio\_module\_handle\_t module,audio\_io\_handle\_t \*output,

audio\_config\_t \*config, audio\_devices\_t devices, const String8& address, audio\_output\_flags\_t flags

step 2. Find the most suitable output device **findSuitableHwDev\_l()**

@ \frameworks\av\services\audioflinger\AudioFlinger.cpp

AudioHwDevice\* AudioFlinger::findSuitableHwDev\_l( audio\_module\_handle\_t module, audio\_devices\_t devices)

step 3. Open the output stream , Create an AudioStreamOut object and call its open method

\frameworks\av\services\audioflinger\AudioHwDevice.cpp

status\_t AudioHwDevice::openOutputStream( AudioStreamOut \*\*ppStreamOut, audio\_io\_handle\_t handle, audio\_devices\_t devices,

audio\_output\_flags\_t flags, struct audio\_config \*config, const char \*address)

Try to open the HAL first using the current format.

ALOGV("openOutputStream(), try sampleRate %d, Format %#x,channelMask %#x", config->sample\_rate, config->format, config->channel\_mask);

status\_t status = outputStream->open(handle, devices, config, address);

The prototype of the open method is as follows:

status\_t AudioStreamOut::open(audio\_io\_handle\_t handle, audio\_devices\_t devices, struct audio\_config \*config, const char \*address)

int status = hwDev()->openOutputStream( handle, devices, customFlags, config, address, &outStream);

In hwDev()->openOutputStream()🡪What is returned is an object of DeviceHalInterface type

@ /frameworks/av/media/libaudiohal/DeviceHalLocal.cpp

status\_t DeviceHalLocal::openOutputStream( audio\_io\_handle\_t handle,audio\_devices\_t devices,audio\_output\_flags\_t flags,

struct audio\_config \*config,const char \*address,sp<StreamOutHalInterface> \*outStream)

Finally calls hal

audio\_hw\_device\_t \*mDev;

@ /hardware/qcom/audio/hal/audio\_hw.c

adev->device.open\_output\_stream = adev\_open\_output\_stream;

adev->device.open\_input\_stream = adev\_open\_input\_stream;

step 4. Create the thread corresponding to the output stream and add it to mMmapThreads, the purpose of calling MmapPlaybackThread is to assign a different thread to different output streams , and finally call mPlaybackThreads.add(\*output, thread) to The current output stream threads are stored in mMmapThreads.

MmapThread(audioFlinger, id, hwDev, output->stream, outDevice, inDevice, systemReady)

---->

ThreadBase(audioFlinger, id, outDevice, inDevice, MMAP, systemReady),

step 5. If it is For other output stream types, create corresponding ones for offload, directoutput, MixerThread

**if** (flags & AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD) {

// AUDIO\_OUTPUT\_FLAG\_COMPRESS\_OFFLOAD 音频流，创建 OffloadThread 实例

**thread** = **new** OffloadThread(this, outputStream, \*output, devices, mSystemReady);

ALOGV("openOutput\_l() created offload output: ID %d thread %p", \*output, **thread**);

} **else** **if** ((flags & AUDIO\_OUTPUT\_FLAG\_DIRECT)

|| !isValidPcmSinkFormat(config->format)

|| !isValidPcmSinkChannelMask(config->channel\_mask)) {

// AUDIO\_OUTPUT\_FLAG\_DIRECT audio stream, create DirectOutputThread instance

**thread** = **new** DirectOutputThread(this, outputStream, \*output, devices, mSystemReady);

ALOGV("openOutput\_l() created direct output: ID %d thread %p", \*output, **thread**);

} **else** {

// Other identified audio streams, create an instance of MixerThread

**thread** = **new** MixerThread(this, outputStream, \*output, devices, mSystemReady);

ALOGV("openOutput\_l() created mixer output: ID %d thread %p", \*output, **thread**);

}