Audio 系统研究第一季

先看看 Audio 里边有主要有三个:

AudioManager：这个主要是用来管理 Audio 系统的

AudioTrack：这个主要是用来播放声音的

AudioRecord：这个主要是用来录音的

其中 AudioManager 的理解需要考虑整个系统上声音的策略问题，例如来电话铃声，短信铃声等，主要是策略上的问题。

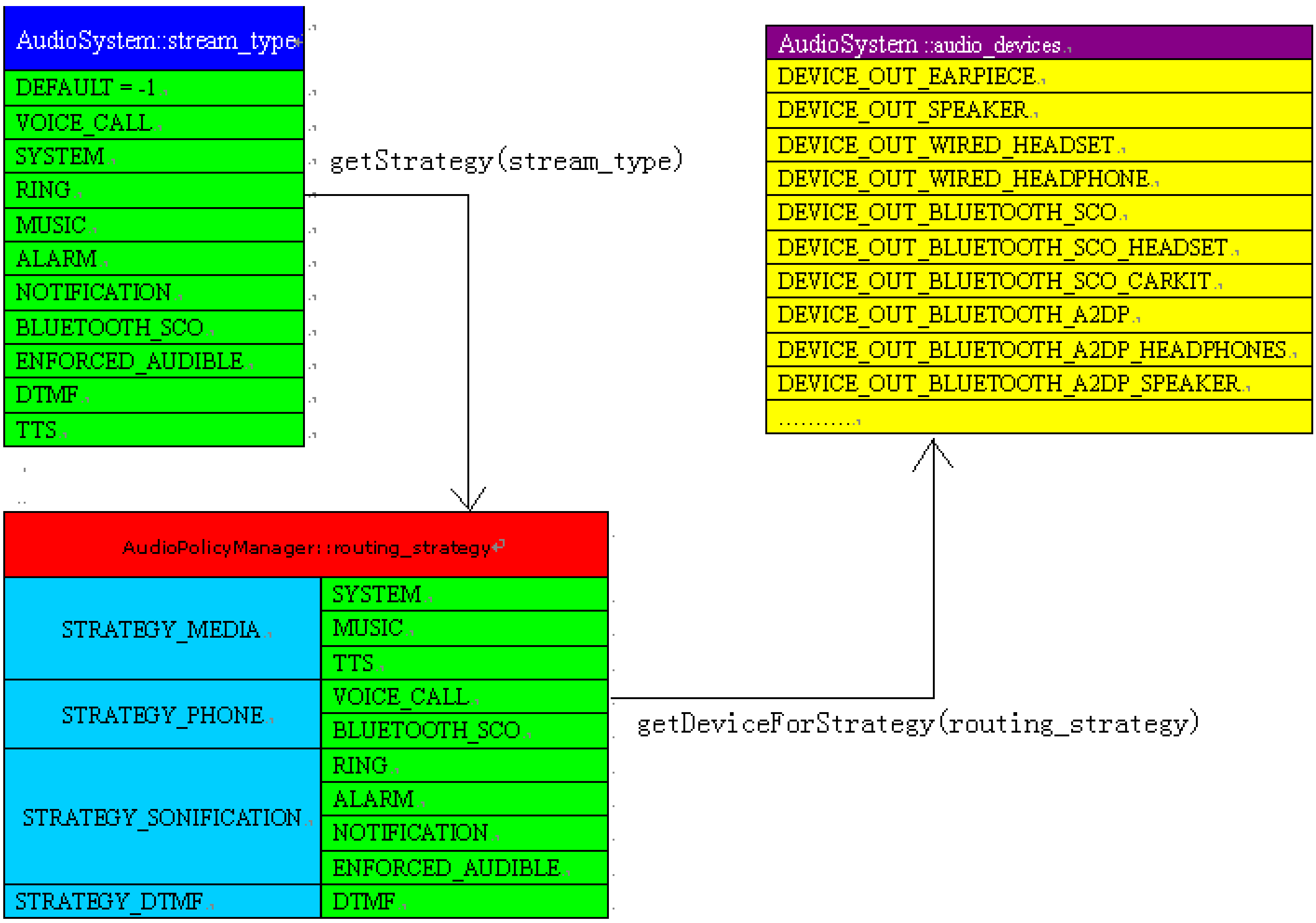
AudioPolicyManager

AudioPolicyService 的很大一部分管理工作都是在 AudioPolicyManager 中完成的。包括音频策略（strategy）管理，音量管理，输入输出设备管理。

1、音频策略管理

我想首先要搞清楚 stream_type，device，strategy 三者之间的关系：

- AudioSystem::stream_type 音频流的类型，一共有 10 种类型
- AudioSystem::audio_devices 音频输入输出设备，每一个 bit 代表一种设备，见前面的说明
- AudioPolicyManager::routing_strategy 音频路由策略，可以有 4 种策略



getStrategy(stream_type)根据 stream type，返回对应的 routing strategy 值，

getDeviceForStrategy()则是根据 routing strategy，返回可用的 device。Android 把 10 种 stream type 归纳为 4 种路由策略，然后根据路由策略决定具体的输出设备。

释义：

DTMF: dual-tone multifrequency 双音多频，由高频群和低频群组成，高低频群各包含 4 个频率。一个高频信号和一个低频信号叠加组成一个组合信号，代表一个数字。DTMF 信号有 16 个编码。利用 DTMF 信令可选择呼叫相应的对讲机

TTS：Text To Speech 的缩写，即“从文本到语音”、语音合成技术(Text To Speech) ……

2、声音管理

```
 /** @hide Maximum volume index values for audio streams */

  private int[] MAX_STREAM_VOLUME = new int[] {

      5,  // STREAM_VOICE_CALL

      7,  // STREAM_SYSTEM

      7,  // STREAM_RING

      15, // STREAM_MUSIC

      7,  // STREAM_ALARM

      7,  // STREAM_NOTIFICATION

      15, // STREAM_BLUETOOTH_SCO

      15, // STREAM_FM

      15, // STREAM_DTMF

      15,  // STREAM_TTS

      7,  // STREAM_SYSTEM_ENFORCED

  };
```

由此可见，电话铃声可以有 7 个级别的音量，而音乐则可以有 15 个音量级别，java 的代码通过 jni，最后调用 AudioPolicyManager 的 initStreamVolume()，把这个数组的内容传入 AudioPolicyManager 中，这样 AudioPolicyManager 也就记 住了每一个音频流的

音量级别。应用程序可以调用 setStreamVolumeIndex 设置各个音频流的音量级 别，setStreamVolumeIndex 会把这个整数的音量级别转化为适合人耳的对数级别，然后通过 AudioPolicyService 的 AudioCommandThread，最终会将设置应用到 AudioFlinger 的相应的 Track 中。

3、输入输出设备管理

音频系统为音频设备定义了一个枚举：AudioSystem::audio_devices，例 如：DEVICE_OUT_SPEAKER，DEVICE_OUT_WIRED_HEADPHONE，DEVICE_OUT_BLUETOOTH_A2DP，DEVICE_IN_BUILTIN_MIC，DEVICE_IN_VOICE_CALL 等等，每一个枚举值其实对应一个 32bit 整数的某一个位，所以这些值是可以进行位或操作的，例如我希望同时打开扬声器和耳机，那么可以这样：

1. newDevice = DEVICE_OUT_SPEAKER | DEVICE_OUT_WIRED_HEADPHONE；
2. setOutputDevice(mHardwareOutput, newDevice);

newDevice = DEVICE_OUT_SPEAKER | DEVICE_OUT_WIRED_HEADPHONE； setOutputDevice(mHardwareOutput, newDevice);

AudioPolicyManager 中有两个成员变量：mAvailableOutputDevices 和 mAvailableInputDevices，他们记录了当前可用的输入和输出设备，当系统检测到耳机或者蓝牙已连接好时，会调用 AudioPolicyManager 的成员函数:

➔ status_t AudioPolicyManager::setDeviceConnectionState(AudioSystem::audio_devices device,
➔ AudioSystem::device_connection_state state,
➔ const char *device_address)

status_t AudioPolicyManager::setDeviceConnectionState(AudioSystem::audio_devices device, AudioSystem::device_connection_state state, const char *device_address)

该函数根据传入的 device 值和 state (DEVICE_STATE_AVAILABLE/DEVICE_STATE_UNAVAILABLE) 设置 mAvailableOutputDevices 或者 mAvailableInputDevices，然后选择相应的输入或者输出设备。

其他一些相关的函数：

- setForceUse()设置某种场合强制使用某一设备，例如
- setForceUse(FOR_MEDIA, FORCE_SPEAKER)会在播放音乐时打开扬声器
- startOutput()/stopOutput()
- startInput()/stopInput()

前端时间增加的一个小功能：ring from speaker and headset together when incoming call if headset is present .

修改点：

在 AP side 修改的几个文件

snd.h,board-msm7x27.c，AudioPolicyManagerBase.cpp ,AudioHardware.cpp

在 snd.h 文件里如下枚举里加一个新的逻辑设备 ID： SND_DEVICE_ST_HDST_SPKR

typedef enum {

SND_DEVICE_DEFAULT = 0,

SND_DEVICE_HANDSET = SND_DEVICE_DEFAULT+0,

SND_DEVICE_HFK = SND_DEVICE_DEFAULT+1,

SND_DEVICE_HEADSET = SND_DEVICE_DEFAULT+2, /* Mono headset */

SND_DEVICE_STEREO_HEADSET = SND_DEVICE_DEFAULT+3, /* Stereo headset */ SND_DEVICE_AHFK = SND_DEVICE_DEFAULT+4,

SND_DEVICE_SDAC = SND_DEVICE_DEFAULT+5,

SND_DEVICE_SPEAKER_PHONE = SND_DEVICE_DEFAULT+6,

SND_DEVICE_TTY_HFK = SND_DEVICE_DEFAULT+7,

SND_DEVICE_TTY_HEADSET = SND_DEVICE_DEFAULT+8,

SND_DEVICE_TTY_VCO = SND_DEVICE_DEFAULT+9,

SND_DEVICE_TTY_HCO = SND_DEVICE_DEFAULT+10,

SND_DEVICE_BT_INTERCOM = SND_DEVICE_DEFAULT+11,

SND_DEVICE_BT_HEADSET = SND_DEVICE_DEFAULT+12,

SND_DEVICE_BT_AG_LOCAL_AUDIO = SND_DEVICE_DEFAULT+13,

SND_DEVICE_USB = SND_DEVICE_DEFAULT+14,

SND_DEVICE_STEREO_USB = SND_DEVICE_DEFAULT+15,

SND_DEVICE_IN_S_SADC_OUT_HANDSET = SND_DEVICE_DEFAULT+16, /* Input Mono SADD, Output Handset */

SND_DEVICE_IN_S_SADC_OUT_HEADSET = SND_DEVICE_DEFAULT+17, /* Input Stereo SADD, Output Headset */

SND_DEVICE_EXT_S_SADC_OUT_HANDSET =

```
SND_DEVICE_DEFAULT+18, /* Input Stereo  SADD, Output Handset */

  SND_DEVICE_EXT_S_SADC_OUT_HEADSET     =
SND_DEVICE_DEFAULT+19, /* Input Stereo SADD, Output Headset  */

  SND_DEVICE_BT_A2DP_HEADSET         = SND_DEVICE_DEFAULT+20, /* A
BT device supporting A2DP */

  SND_DEVICE_BT_A2DP_SCO_HEADSET      =
SND_DEVICE_DEFAULT+21, /* A BT headset supporting A2DP and SCO */

  /* Input Internal Codec Stereo SADC, Output External AUXPCM  */

  SND_DEVICE_TX_INT_SADC_RX_EXT_AUXPCM     =
SND_DEVICE_DEFAULT+22,

  SND_DEVICE_RX_EXT_SDAC_TX_INTERNAL    =
SND_DEVICE_DEFAULT+23,

  SND_DEVICE_BT_CONFERENCE              = SND_DEVICE_DEFAULT+24,

  SND_DEVICE_IN_S_SADC_OUT_SPEAKER_PHONE =
SND_DEVICE_DEFAULT+25,

#if defined(PWV_AUDIO_HEADSET_SPEAKER)

  SND_DEVICE_ST_HDST_SPKR     =   SND_DEVICE_DEFAULT+26, //yanglin
add for audio

  SND_DEVICE_MAX                 = SND_DEVICE_DEFAULT+27,

  SND_DEVICE_CURRENT             = SND_DEVICE_DEFAULT+28,

#else

  SND_DEVICE_MAX                 = SND_DEVICE_DEFAULT+26,

  SND_DEVICE_CURRENT             = SND_DEVICE_DEFAULT+27,

#endif

  /* DO NOT USE: Force this enum to be a 32bit type */

  SND_DEVICE_32BIT_DUMMY                =
SNDDEV_DUMMY_DATA_UINT32_MAX
```

} snd_device_type;

这边的 ID 必须和 ARM9 端 ID 匹配起来，ARM9 那边也需要作同样的修改。

board-msm7x27.c 文件作以下修改

```
static struct snd_endpoint snd_endpoints_list[] = {
    SND(HANDSET, 0),
    SND(MONO_HEADSET, 2),
    SND(HEADSET, 3),
    SND(SPEAKER, 6),
    SND(TTY_HEADSET, 8),
    SND(TTY_VCO, 9),
    SND(TTY_HCO, 10),
    SND(BT, 12),
    SND(IN_S_SADC_OUT_HANDSET, 16),
    SND(IN_S_SADC_OUT_SPEAKER_PHONE, 25),
#if defined(PWV_AUDIO_HEADSET_SPEAKER)
    SND(HDST_SPKR, 26),
    SND(CURRENT, 28),
#else
    SND(CURRENT, 27),
#endif／／这边的 28 代表的是 current audio device ID,需要和 ARM9 匹配。这边设置错误会导致开机 assert!
};
```

AudioPolicyManagerBase.cpp 文件修改如下：

```
void AudioPolicyManagerBase::setPhoneState(int state)
{
    LOGV("setPhoneState() state %d", state);
    uint32_t newDevice = 0;
    if (state < 0 || state >= AudioSystem::NUM_MODES) {
        LOGW("setPhoneState() invalid state %d", state);
        return;
    }

    if (state == mPhoneState ) {
        LOGW("setPhoneState() setting same state %d", state);
        return;
    }

    // if leaving call state, handle special case of active streams
    // pertaining to sonification strategy see handleIncallSonification()
    if (isInCall()) {
        LOGV("setPhoneState() in call state management: new state is %d", 
state);
        for (int stream = 0; stream < AudioSystem::NUM_STREAM_TYPES; 
stream++) {
            handleIncallSonification(stream, false, true);
        }
```

```
    }

    // store previous phone state for management of sonification strategy below
    int oldState = mPhoneState;
    mPhoneState = state;
    // force routing command to audio hardware when starting call
    // even if no device change is needed
#if defined(PWV_AUDIO_HEADSET_SPEAKER)
    bool force = (mPhoneState == AudioSystem::MODE_IN_CALL)||
(mPhoneState == AudioSystem::MODE_RINGTONE);//yanglin modify for
audio
#else
    bool force = (mPhoneState == AudioSystem::MODE_IN_CALL);
#endif ／／修改的目的是为了在来电响铃时把变量 force 赋 true,后面会根据这个变量去
判断是否需要进行设备切换
    // are we entering or starting a call
    if (!isStateInCall(oldState) && isStateInCall(state)) {
        LOGV("  Entering call in setPhoneState()");
        // force routing command to audio hardware when starting a call
        // even if no device change is needed
        force = true;
    } else if (isStateInCall(oldState) && !isStateInCall(state)) {
        LOGV("  Exiting call in setPhoneState()");
        // force routing command to audio hardware when exiting a call
        // even if no device change is needed
```

```
        force = true;

    } else if (isStateInCall(state) && (state != oldState)) {

        LOGV("  Switching between telephony and VoIP in setPhoneState()");

        // force routing command to audio hardware when switching between
telephony and VoIP

        // even if no device change is needed

        force = true;

    }



    // check for device and output changes triggered by new phone state

    newDevice = getNewDevice(mHardwareOutput, false);

#ifdef WITH_A2DP

    checkOutputForAllStrategies();

    checkA2dpSuspend();

#endif

    updateDeviceForStrategy();



    AudioOutputDescriptor *hwOutputDesc =
mOutputs.valueFor(mHardwareOutput);



    // force routing command to audio hardware when ending call

    // even if no device change is needed

    if (isStateInCall(oldState) && newDevice == 0) {

        newDevice = hwOutputDesc->device();
```

```
        force = true;
    }

    // when changing from ring tone to in call mode, mute the ringing tone
    // immediately and delay the route change to avoid sending the ring tone
    // tail into the earpiece or headset.
    int delayMs = 0;
    if (isStateInCall(state) && oldState == AudioSystem::MODE_RINGTONE) {
        // delay the device change command by twice the output latency to have
some margin
        // and be sure that audio buffers not yet affected by the mute are out
when
        // we actually apply the route change
        delayMs = hwOutputDesc->mLatency*2;
        setStreamMute(AudioSystem::RING, true, mHardwareOutput);
    }

    // change routing is necessary
    setOutputDevice(mHardwareOutput, newDevice, force, delayMs);

    // if entering in call state, handle special case of active streams
    // pertaining to sonification strategy see handleIncallSonification()
    if (isStateInCall(state)) {
        LOGV("setPhoneState() in call state management: new state is %d",
state);
```

```
        // unmute the ringing tone after a sufficient delay if it was muted before
        // setting output device above
        if (oldState == AudioSystem::MODE_RINGTONE) {
            setStreamMute(AudioSystem::RING, false, mHardwareOutput,
MUTE_TIME_MS);
        }
        for (int stream = 0; stream < AudioSystem::NUM_STREAM_TYPES;
stream++) {
            handleIncallSonification(stream, true, true);
        }
    }
    // Flag that ringtone volume must be limited to music volume until we exit
MODE_RINGTONE
    if (state == AudioSystem::MODE_RINGTONE &&
        (hwOutputDesc->mRefCount[AudioSystem::MUSIC] ||
        (systemTime() - mMusicStopTime) <
seconds(SONIFICATION_HEADSET_MUSIC_DELAY))) {
        mLimitRingtoneVolume = true;
    } else {
        mLimitRingtoneVolume = false;
    }
}
```

文件 AudioHardware.cpp 作以下修改：

```
status_t AudioHardware::doRouting(AudioStreamInMSM72xx *input)
{
```

```
    /* currently this code doesn't work without the htc libacoustic */



    Mutex::Autolock lock(mLock);

    uint32_t outputDevices = mOutput->devices();

    status_t ret = NO_ERROR;

    int new_snd_device = -1;

    int new_post_proc_feature_mask = 0;



    //int (*msm72xx_enable_audpp)(int);

    //msm72xx_enable_audpp = (int (*)(int))::dlsym(acoustic,
"msm72xx_enable_audpp");



    if (input != NULL) {

        uint32_t inputDevice = input->devices();

        LOGI("do input routing device %x\n", inputDevice);

        // ignore routing device information when we start a recording in voice

        // call

        // Recording will happen through currently active tx device

        if(inputDevice == AudioSystem::DEVICE_IN_VOICE_CALL)

            return NO_ERROR;

        if (inputDevice != 0) {

            if (inputDevice &
AudioSystem::DEVICE_IN_BLUETOOTH_SCO_HEADSET) {

                LOGI("Routing audio to Bluetooth PCM\n");
```

```
                new_snd_device = SND_DEVICE_BT;

            } else if (inputDevice & AudioSystem::DEVICE_IN_WIRED_HEADSET)
{

                    LOGI("Routing audio to Wired Headset\n");

                    new_snd_device = SND_DEVICE_HEADSET;

            } else {

                if (outputDevices & AudioSystem::DEVICE_OUT_SPEAKER) {

                    LOGI("Routing audio to Speakerphone\n");

                    new_snd_device = SND_DEVICE_SPEAKER;

                    new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE |
RX_IIR_ENABLE | MBADRC_ENABLE);

                } else {

                    LOGI("Routing audio to Handset\n");

                    new_snd_device = SND_DEVICE_HANDSET;

                }

            }

        }

    }



    // if inputDevice == 0, restore output routing

    if (new_snd_device == -1) {

        if (outputDevices & (outputDevices - 1)) {

            if ((outputDevices & AudioSystem::DEVICE_OUT_SPEAKER) == 0) {

                LOGW("Hardware does not support requested route combination
(%#X),"
```

```
                " picking closest possible route...", outputDevices);

        }

    }



    if ((mTtyMode != TTY_OFF) && (mMode == 
AudioSystem::MODE_IN_CALL) &&

            (outputDevices & AudioSystem::DEVICE_OUT_WIRED_HEADSET)) 
{

        if (mTtyMode == TTY_FULL) {

            LOGI("Routing audio to TTY FULL Mode\n");

            new_snd_device = SND_DEVICE_TTY_HEADSET;

        } else if (mTtyMode == TTY_VCO) {

            LOGI("Routing audio to TTY VCO Mode\n");

            new_snd_device = SND_DEVICE_TTY_VCO;

        } else if (mTtyMode == TTY_HCO) {

            LOGI("Routing audio to TTY HCO Mode\n");

            new_snd_device = SND_DEVICE_TTY_HCO;

        }

    } else if (outputDevices &

               (AudioSystem::DEVICE_OUT_BLUETOOTH_SCO | 
AudioSystem::DEVICE_OUT_BLUETOOTH_SCO_HEADSET)) {

        LOGI("Routing audio to Bluetooth PCM\n");

        new_snd_device = SND_DEVICE_BT;

    } else if (outputDevices & 
AudioSystem::DEVICE_OUT_BLUETOOTH_SCO_CARKIT) {
```

```
            LOGI("Routing audio to Bluetooth PCM\n");

            new_snd_device = SND_DEVICE_CARKIT;

#ifdef COMBO_DEVICE_SUPPORTED

        } else if ((outputDevices &
AudioSystem::DEVICE_OUT_WIRED_HEADSET) &&

                   (outputDevices & AudioSystem::DEVICE_OUT_SPEAKER)) {

            LOGI("Routing audio to Wired Headset and Speaker\n");

            new_snd_device = SND_DEVICE_HEADSET_AND_SPEAKER;

            new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE |
RX_IIR_ENABLE | MBADRC_ENABLE);

        } else if (outputDevices &
AudioSystem::DEVICE_OUT_WIRED_HEADPHONE) {

            if (outputDevices & AudioSystem::DEVICE_OUT_SPEAKER) {

                LOGI("Routing audio to No microphone Wired Headset and Speaker
(%d,%x)\n", mMode, outputDevices);

                new_snd_device = SND_DEVICE_HEADSET_AND_SPEAKER;

                new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE |
RX_IIR_ENABLE | MBADRC_ENABLE);

            } else {

                LOGI("Routing audio to No microphone Wired Headset (%d,%x)\n",
mMode, outputDevices);

                new_snd_device = SND_DEVICE_NO_MIC_HEADSET;

            }

#endif

        } else if (outputDevices &
AudioSystem::DEVICE_OUT_WIRED_HEADSET) {

            LOGI("Routing audio to Wired Headset\n");
```

```
            new_snd_device = SND_DEVICE_HEADSET;
            new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE | RX_IIR_ENABLE | MBADRC_ENABLE);
        } else if (outputDevices & AudioSystem::DEVICE_OUT_SPEAKER) {
            LOGI("Routing audio to Speakerphone\n");
            new_snd_device = SND_DEVICE_SPEAKER;
            new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE | RX_IIR_ENABLE | MBADRC_ENABLE);
        } else {
            LOGI("Routing audio to Handset\n");
            new_snd_device = SND_DEVICE_HANDSET;
            new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE | RX_IIR_ENABLE | MBADRC_ENABLE);
        }
    }
#if defined(PWV_AUDIO_HEADSET_SPEAKER)
     //below yanglin add for audio
     if (mMode == AudioSystem::MODE_RINGTONE)
          {
            LOGE("Routing audio to Wired Headset and Speaker: %d\n",SND_DEVICE_HDST_SPKR);
            new_snd_device = SND_DEVICE_HDST_SPKR; //来电强制切换到我们自己添加的音频设备，通过 RPC 把 ID 发到 ARM9 端，ARM9 会调用用相应函数去设置寄存器，把 headset 和 speakerPA 等同时打开，
            new_post_proc_feature_mask = (ADRC_ENABLE | EQ_ENABLE | RX_IIR_ENABLE | MBADRC_ENABLE);
          }
```

```
    //above yanglin add for audio
#endif
    if (mDualMicEnabled && mMode == AudioSystem::MODE_IN_CALL) {
        if (new_snd_device == SND_DEVICE_HANDSET) {
            LOGI("Routing audio to handset with DualMike enabled\n");
            new_snd_device = SND_DEVICE_IN_S_SADC_OUT_HANDSET;
        } else if (new_snd_device == SND_DEVICE_SPEAKER) {
            LOGI("Routing audio to speakerphone with DualMike enabled\n");
            new_snd_device = SND_DEVICE_IN_S_SADC_OUT_SPEAKER_PHONE;
        }
    }

    if (new_snd_device != -1 && new_snd_device != mCurSndDevice) {
        ret = doAudioRouteOrMute(new_snd_device);

       //disable post proc first for previous session
       if(playback_in_progress)
           msm72xx_enable_postproc(false);

       //enable post proc for new device
       snd_device = new_snd_device;
       post_proc_feature_mask = new_post_proc_feature_mask;
```

```
        if(playback_in_progress)

            msm72xx_enable_postproc(true);



        mCurSndDevice = new_snd_device;

    }

    return ret;

}
```

以上 ARM11 端修改结束了，剩下的就是在 ARM9 添加一个逻辑设备，和设置相应设备寄存器值。在此不贴代码，需要我可以发给大家。