密级状态：绝密（　　）　秘密（　　）　内部（　　）　公开（ √ ）

RK 音频相关 debug

（技术部）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V1.1
	作　　者：	蔡焕钦
	完成日期：	2016-6-12
	审　　核：	
	完成日期：	

福州瑞芯微电子有限公司

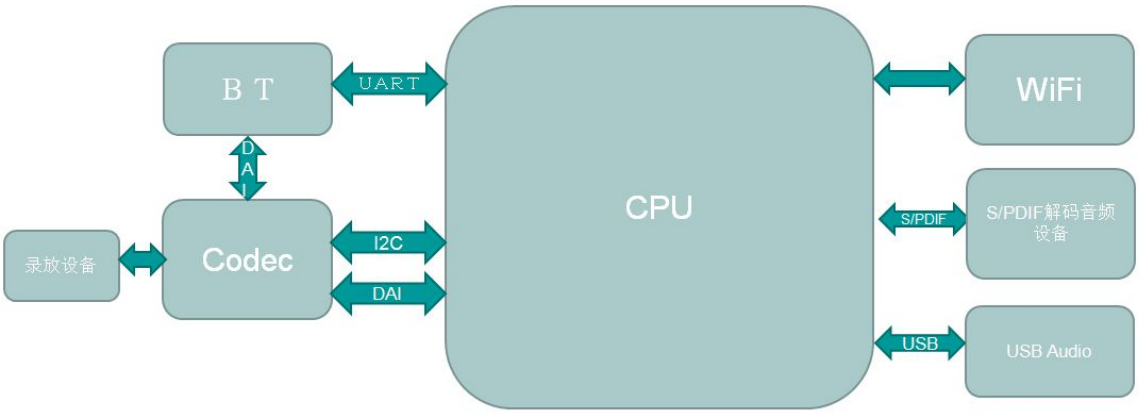
Fuzhou　Rockchips　Semiconductor　Co . , Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	备注

目 录

音频系统基本硬件电路



音频编解码器 Codec 负责处理音频信息，包括 ADC,DAC,Mixer,DSP,输入输出以及音量控制等所有与音频相关的功能。

Codec 与处理器之间通过 I2C 总线和数字音频接口 DAI 进行通信。

I2C 总线 - 实现对 Codec 寄存器数据的读写。

DAI –（digital audio interface）实现音频数据在 CPU 和 Codec 间的通信，包括 I2S、PCM 和 AC97 等。

蓝牙立体声音乐播放走的是蓝牙跟 CPU 直接的 UART 接口。

在只有一组 I2S 的主控上面，蓝牙通话 SCO 暂时是通过 Codec 中继，即通过 codec 来路由选择是否走通话，走的是 I2S 接口 。如果主控有两组 I2S，那么可以将 BT PCM 直接接到主控的 PCM 接口上。

S/DIF（Sony/Philips Digital Interface Format），通过光纤或同轴电缆传输音频，保证音频质量。

Codec 内部通路举例 RK616

学习完 ppt，清晰概念：

1、了解声卡是怎么打开，如何播放和录音，并能指定哪个声卡播放。

2、知道怎么注册新的声卡到平台上。

3、简单问题排查。

ALSA 设备文件结构

一. ALSA 设备文件结构

我们从 alsa 在 linux 中的设备文件结构开始我们的 alsa 之旅. 看看我的电脑中的 alsa 驱动的设备文件结构:

$ cd /dev/snd

$ ls -l

crw-rw----+ 1 root audio 116, 8 2011-02-23 21:38 controlC0

crw-rw----+ 1 root audio 116, 7 2011-02-23 21:39 pcmC0D0c

crw-rw----+ 1 root audio 116, 6 2011-02-23 21:56 pcmC0D0p

crw-rw----+ 1 root audio 116, 5 2011-02-23 21:38 pcmC0D1p

$

我们可以看到以下设备文件:

controlC0 --> 用于声卡的控制，例如通道选择，混音，麦克风的控制等

pcmC0D0c --〉 用于录音的 pcm 设备

pcmC0D0p --〉 用于播放的 pcm 设备

其中， C0D0 代表的是声卡 0 中的设备 0， pcmC0D0c 最后一个 c 代表 capture， pcmC0D0p 最后一个 p 代表 playback，这些都是 alsa-driver 中的命名规则。

Pcm.c (sound\core) 跟下去 snd_register_device_for_dev 函数

就可以发现调用的 device_create 、device_add 创建设备节点。

```c-sharp
static int snd_pcm_dev_register(struct snd_device *device)
{
    int cidx, err;
    char str[16];
    struct snd_pcm *pcm;
    struct device *dev;

    pcm = device->device_data;
         ......
    for (cidx = 0; cidx < 2; cidx++) {
                    ......
        switch (cidx) {
        case SNDRV_PCM_STREAM_PLAYBACK:
            sprintf(str, "pcmC%iD%ip", pcm->card->number, pcm->device);
            devtype = SNDRV_DEVICE_TYPE_PCM_PLAYBACK;
            break;
        case SNDRV_PCM_STREAM_CAPTURE:
            sprintf(str, "pcmC%iD%ic", pcm->card->number, pcm->device);
            devtype = SNDRV_DEVICE_TYPE_PCM_CAPTURE;
            break;
        }
        /* device pointer to use, pcm->dev takes precedence if
         * it is assigned, otherwise fall back to card's device
         * if possible */
        dev = pcm->dev;
        if (!dev)
            dev = snd_card_get_device_link(pcm->card);
        /* register pcm */
        err = snd_register_device_for_dev(devtype, pcm->card,
                                pcm->device,
                                &snd_pcm_f_ops[cidx],
                                pcm, str, dev);
                    ......
    }
```

以上代码我们可以看出，对于一个 pcm 设备，可以生成两个设备文件，一个用于 playback，一个用于 capture，代

码中也确定了他们的命名规则：

playback -- pcmCxDxp，通常系统中只有一个声卡和一个 pcm，它就是 pcmC0D0p

capture -- pcmCxDxc，通常系统中只有一个声卡和一个 pcm，它就是 pcmC0D0c

二. Android 下的 tinyalsa 如何去播放音乐与录音：

1、播放音乐流程：

pcm_open → pcm_write → pcm_close

所以只要打开设备，往设备写数据，使用完成后关闭设备，就和普通的操作过程是一模一样的。

现在我们解析一下音频参数：

out->pcm[PCM_CARD] = pcm_open(PCM_CARD, PCM_DEVICE,

PCM_OUT | PCM_MONOTONIC, &out->config);

注：

以 pcmC0D0p 为例

PCM_CARD 就是 C0 中的 C(card)，表示声卡 0.

PCM_DEVICE 就是 D0 中的 D(device)，表示 设备 0，一个声卡有可能有多个设备，所以需要指定。

PCM_OUT 表示 播放。

PCM_MONOTONIC 指时钟模式，表示使用的是相对时间,他的时间是通过 jiffies 值来计算的。如果去掉则使用系统实时时钟计时，这个参数不要修改。

out->config 就是配置此声卡的采样率、通道等，下面结构体就能够了解声卡参数：

```
struct pcm_config pcm_config = {
    .channels = 2,
    .rate = 44100,
    .period_size = 1024,
    .period_count = 4,
    .format = PCM_FORMAT_S16_LE,
};
```

看到这里应该明白，设置声卡的采样率、通道数、格式等就是在打开声卡时设置的。

所以打开 pcmC0D0p，pcm_open 可以写为：

```
out->pcm[0] = pcm_open(0, 0,
                       PCM_OUT | PCM_MONOTONIC, &out->config);
```

为什么呢？简单解析 pcm_open 看看。

```
struct pcm *pcm_open(unsigned int card, unsigned int device,
                     unsigned int flags, struct pcm_config *config)
{
    struct pcm *pcm;
    struct snd_pcm_info info;
    struct snd_pcm_hw_params params;
    struct snd_pcm_sw_params sparams;
    char fn[256];
    int rc;
    //ALOGD("====%s===%d======", __func__, __LINE__);
    pcm = calloc(1, sizeof(struct pcm));
    if (!pcm || !config)
        return &bad_pcm; /* TODO: could support default config here */

    pcm->config = *config;

    snprintf(fn, sizeof(fn), "/dev/snd/pcmC%uD%u%c", card, device,
             flags & PCM_IN ? 'c' : 'p');

    pcm->flags = flags;
    pcm->fd = open(fn, O_RDWR);
    if (pcm->fd < 0) {
        oops(pcm, errno, "cannot open device '%s'", fn);
        return pcm;
    }

    if (ioctl(pcm->fd, SNDRV_PCM_IOCTL_INFO, &info)) {
        oops(pcm, errno, "cannot get info");
```

看到图中就可以知道，snprintf 得到的 fn 就是 pcmC0D0p，图中后面还有若干 ioctl，就是设置声卡参数，这里就不贴出来了，请自行查看源码。

ret = pcm_write(out->pcm[i], (void *)buffer, bytes);

out->pcm[i] 就是上面申请的 fd，

buffer 就是将写入的数据，

bytes 就是写入数据的字节数。

看到这里，就类似于往文件里面写数据，写多少个字节。

pcm_close(in->pcm);

in->pcm = NULL;

和关闭文件操作是一样的。

2、录音流程：

pcm_open → pcm_read → pcm_close

pcm_open pcm_close 前面已经说了，这里就简单解析 pcm_read

in->read_status = pcm_read(in->pcm, (void*)in->buffer,bytes);

in->pcm 就是上面申请的 fd，

(void*)in->buffer 就是将读出的数据。

bytes 就是读出数据的字节数。

Alsa 的上层应用程序

如果不希望使用 android 的 api 接口进行录放的话可以查阅相关代码 external/tinyalsa 。

Tinyalsa 是对 alsa-lib 进行了裁剪，alsa 官方提供的比较强大。BOX 4.4 和 4.2 的 SDK 是用 alsa lib。

后面的 5.1 SDK 统一使用 tinyalsa。

tinycap.c 是录音程序;

tinyplay.c 是放音程序;

pcm.c 是对 linux alsa 驱动的一些封装;

mixer.c 是相关的 ctl 设置主要是 codec 寄存器相关。

另外 5.1 SDK 的 audio Hal 层也是引用该目录的代码。

```
lxt@rksz-server101:~/rk3288-4.4.2/external/tinyalsa$ tree
├── Android.mk
├── include
│   └── tinyalsa
│       └── asoundlib.h
├── mixer.c
├── MODULE_LICENSE_BSD
├── NOTICE
├── pcm.c
├── README
├── tinycap.c
├── tinymix.c
├── tinypcminfo.c
└── tinyplay.c

2 directories, 11 files
```

注： extern/tinyalsa 下的 tinyplay.c 可以编译出播放音乐的 tinyplay，录音是 tinycap，他们就是简单的播放录音例子。

例如

tinyplay /sdcard/44_1k.wav 可以播放 44.1k 的 wav 文件，默认播放声卡 0 的 0 设备，44.1k 的采样率如果需要其它采样率，或者可以设置 tinyplay 的参数，同时播放的文件也要使用相应的采样率。

tinycap 同理。

1.编译 tinyalsa 配套工具

$ mmm external/tinyalsa/

编译完后会产生 tinyplay/tinymix/tinycap 等等工具。

tinymix: 查看配置混音器

tinyplay: 播放音频

tinycap: 录音

2.查看当前系统的声卡

[python] view plain copy

```
root@android:/ # cat /proc/asound/cards
 0 [RKRK616        ]: RK_RK616 - RK_RK616
                      RK_RK616
 1 [ROCKCHIPSPDIF  ]: ROCKCHIP-SPDIF - ROCKCHIP-SPDIF
                      ROCKCHIP-SPDIF
root@android:/ #
```

3.tinymix 查看混响器

tinymix 使用方法 a.不加任何参数-显示当前配置情况　b.tinymix [ctrl id] [var]不加[var]可以查看该[ctrl id]可选选项。

[python] view plain copy

```
root@android:/ # tinymix
Number of controls: 7
ctl type    num name                                value
0   ENUM    1   Playback Path                       OFF
1   ENUM    1   Capture MIC Path                    MIC OFF
2   ENUM    1   Voice Call Path                     OFF
3   ENUM    1   Voip Path                           OFF
4   INT 2   Speaker Playback Volume                 0 0
```

```
5   INT 2   Headphone Playback Volume               0 0
6   ENUM    1   Modem Input Enable                      ON
root@android:/ #
```

对应解释：

英文	中文	备注
Playback Path	音频输出通道	
Capture MIC Path	音频输入通道	
Voice Call Pah	通话音频通道	设备没有通话模块，暂无法测试
Voip Pah	IP 电话音频通道	场景 Gtalk;值有:SPK/HP_NO_MIC/BT
Speaker Playback Volume	扬声器音量	和上层音量值无关
Headphone Playback Volume	耳机音量	同上
Modem Input Enable	暂不知何用	经测试不能控制音频输入输出

Playback Path 有：

英文	中文	备注
OFF	关闭	
RCV	—	
SPK	扬声器	常用
HP	耳机带麦	
HP_NO_MIC	耳机无麦	常用
BT	蓝牙	
SPK_HP	—	

RING_SPK	—
RING_HP	—
RING_HP_NO_MIC	—
RING_SPK_HP	—

例：将输出切换到扬声器

root@android:/ # tinymix 0 SPK

关于 tinymix 小结：

通过观察发现，Android 系统的声音音量的调节并没有直接使用 tinyalsa，而基于上层软件实现，因为无论上层音量怎么改变，这里看到的都是 24（以我采用的设备为例）。通道的切换是真正使用了 tinyalsa，当通过不同通道播放音乐的时候可以实时观察到通道的切换。

4.使用 tinyplay 播放 wav 音乐

这个只是一个最基本的播放器，所以不支持播放 MP3 等等压缩过格式的音乐。没有学会使用前，网上都说很麻烦，但是现在看来一点也不麻烦，直接播放了 44.1kHz/44.8kHz 的 wav 音乐。

[python] view plain copy

```
root@android:/ # tinyplay /sdcard/0_16.wav
Playing sample: 2 ch, 44100 hz, 16 bit
root@android:/ #
```

注：播放之前得首先使用 tinymix 把通道设置好，上文中已经给出了设置到扬声器中的例子;由于播放时使用的最大音量进行播放的，所以注意防止被吓到。这里将测试音频文件上传。

5.tinycap 使用

root@android:/ # tinycap /sdcard/test.wav

可以进行录音。

rk 平台默认 codec 为 card0，hdmi 与 card0 共用 i2s。外部 spdif 为 card1.usb 设备为 card3.

打开哪一个声卡都可以用上面的方式打开并播放。

简单说明几个平台的共性：

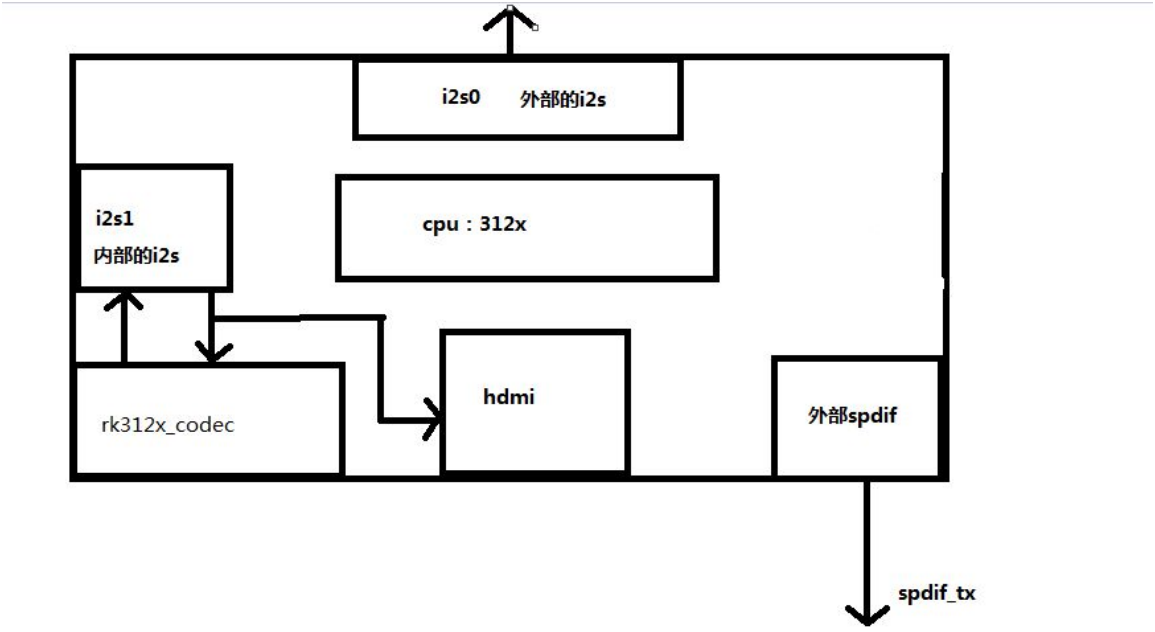
1、312x 内部 codec(即 312x_codec.c)与 hdmi 共用 i2s1.

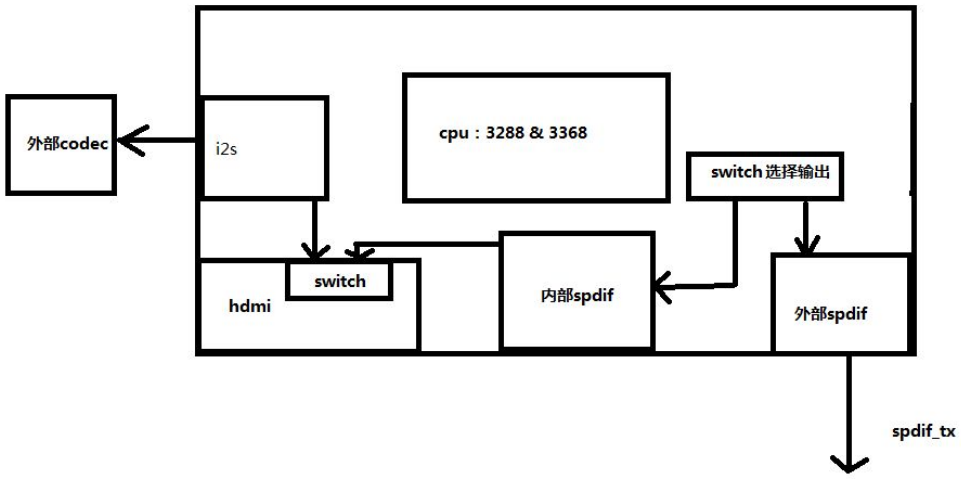
2、3288 和 3368 均是 外部 codec 与 hdmi 共用 i2s0.注意，hdmi 有两个输入选通，内部 spdif 和 i2s，

所以 hdmi 可以设置为 spdif 输入。搜索 dts 和 dtsi 文件 rockchip,hdmi_audio_source = <0>;

设置为 0 表示 i2s 输入，1 表示 spdif 输入。

各个芯片结构会另外配个图来展示。





了解完声卡的打开和播放，现在说明下驱动怎么去配置声卡。

Sound card 驱动配置相关代码

ALSA 驱动目录结构

```
lxt@rksz-server101:~/rk312x-sdk-4.4.4/kernel/sound$ tree -L 1
├── ac97_bus.c
├── aoa
├── arm
├── atmel
├── built-in.mod.c
├── built-in.o
├── core
├── drivers
├── firewire
├── i2c
├── isa
├── Kconfig
├── last.c
├── last.o
├── Makefile
├── mips
├── oss
├── parisc
├── pci
├── pcmcia
├── ppc
├── sh
├── soc
├── sound_core.c
├── soundcore.mod.c
├── sound_core.o
```

```
├──── soundcore.o
├──── sound_firmware.c
├──── sparc
├──── spi
├──── synth
└──── usb

20 directories, 13 files
```

嵌入式设备的音频系统可以被划分为板载硬件（Machine）、Soc（Platform）、Codec 三大部分同时音频驱动有三部分：

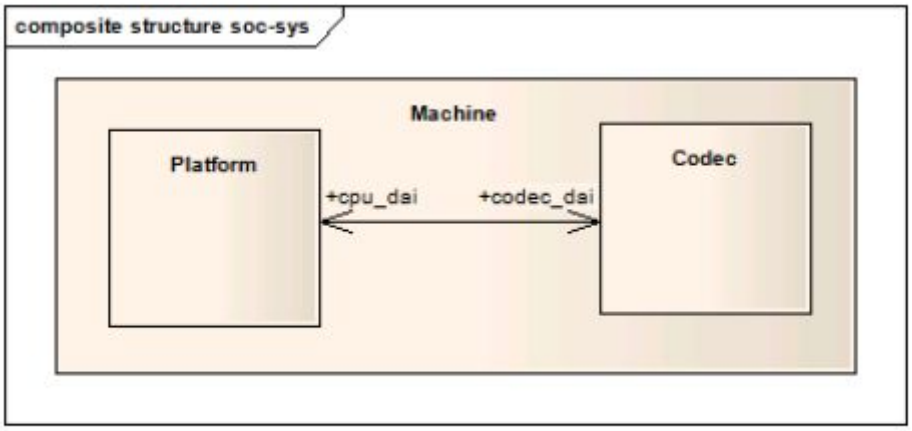


图2.1 音频系统结构

Machine driver

sound/soc/rockchip/rk_es8323.c

其中的 Machine 驱动负责 Platform 和 Codec 之间的耦合以及部分和设备或板子特定的代码采样率时钟配置

Platform driver

sound/soc/ rockchip /rk30_i2s.c

I2S 控制器驱动 采样率时钟 DMA 等配置

Codec driver

sound/soc/codecs/es8323.c

codec 的寄存器通路的配置

想要深入了解 alsa 驱动最好的办法是阅读 kernel 代码

这个 blog 写的不错 http://blog.csdn.net/droidphone/article/details/6271122

因此需要注册一个 codec，上面对应的三部分在 dts 都要配置。

Dts 的 machine 部分：

```
rockchip-es8316 {
        compatible = "rockchip-es8316";
        dais {
                dai0 {
                        audio-codec = <&es8316>;
                        i2s-controller = <&i2s0>;
                        format = "i2s";
                };
        };
        };
```

注： compatible = "rockchip-es8316"; 属性，这个即用来是 macth 代码 rk_es8323.c 的。

audio-codec = <&es8316>; 指定使用的是哪个 codec。

i2s-controller = <&i2s0>; 指定使用的是哪个 i2s

format = "i2s"; 指的是通信格式，i2s 总线上可以使用 i2s、pcm a、pcm b、左对齐等格式。

Dts 的 codec 部分：

```
&i2c1 {
    status = "okay";
    es8316: es8316@10 {
        compatible = "es8316";
        reg = <0x10>;
        spk-con-gpio = <&gpio0 GPIO_D3 GPIO_ACTIVE_HIGH>;
        hp-det-gpio = <&gpio0 GPIO_C7 GPIO_ACTIVE_LOW>;
        status = "okay";
    };
};
```

这部分就和普通注册一个 i2c 设备是一致的，就不赘述了。

Dts 的 platform 的 i2s 部分：

```
    i2s0: i2s0@ff898000 {
        compatible = "rockchip-i2s";
        reg = <0x0 0xff898000 0x0 0x1000>;
        i2s-id = <0>;
        clocks = <&clk_i2s>, <&i2s_out>, <&clk_gates12 7>;
```

```
        clock-names = "i2s_clk", "i2s_mclk", "i2s_hclk";
        interrupts = <GIC_SPI 53 IRQ_TYPE_LEVEL_HIGH>;
        dmas = <&pdma0 0>, <&pdma0 1>;
        #dma-cells = <2>;
        dma-names = "tx", "rx";
        pinctrl-names = "default", "sleep";
        pinctrl-0 = <&i2s_mclk &i2s_sclk &i2s_lrckrx &i2s_lrcktx &i2s_sdi &i2s_sdo0 &i2s_sdo1
&i2s_sdo2 &i2s_sdo3>;
        pinctrl-1 = <&i2s_gpio>;
    };
```

注：

compatible = "rockchip-i2s"; 属性，这个即用来是 macth 代码 rk30_i2s.c 的。

dmas = <&pdma0 0>, <&pdma0 1>;

#dma-cells = <2>;

dma-names = "tx", "rx";

这里指定了使用的 dma，dma 通道是 pdma0 0 和 pdma0 1 分别作为播音和录音的 dma 通道。

Dts 的 platform 的 dma 部分：

```
        pdma0: pdma@ff600000 {
            compatible = "arm,pl330", "arm,primecell";
            reg = <0x0 0xff600000 0x0 0x4000>;
            clocks = <&clk_gates12 11>;
            clock-names = "apb_pclk";
            interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>,
                         <GIC_SPI 1 IRQ_TYPE_LEVEL_HIGH>;
            #dma-cells = <1>;

        };
```

这下齐了。可以看到，dma 和 i2s 部分平台这边已经写好。dts 和代码可以 match 上，所以客户添加一个新的 codec 时只需要改 machine 部分和 codec 部分即可，其中 machine 部分引用平台的 i2s。就可以成功注册一个声卡，应该注意的细节后面的常见问题会细说。

```
shell@rk3288_box:/d/clk # cat clk_summary | grep i2s
  i2s_clkin                      0        0        0
        clk_i2s_pll              1        1        594000000
```

i2s_frac	1	1	11289600
clk_i2s	2	2	11289600
clk_i2s_out	1	1	11289600
g_hclk_i2s	1	1	148500000

对应：

clock	enable_cnt	prepare_cnt	rate

延伸部分：

另外附个注册声卡流程与播放录音.txt 来说明

alsa HAL 层

android 5.1 BOX MID 的 SDK 之后统一使用这个目录下面的代码

\hardware\rockchip\audio\tinyalsa_hal

4.4 MID HAL 代码路径

hardware\rk29\audio

4.4 BOX HAL 代码路径

hardware\alsa_sound

android audio route 通路介绍

android 定义了很多的音频 devices

如 AudioSystem .java 中：

```
211     // output devices, be sure to update AudioManager.java also
212     public static final int DEVICE_OUT_EARPIECE = 0x1;
213     public static final int DEVICE_OUT_SPEAKER = 0x2;
214     public static final int DEVICE_OUT_WIRED_HEADSET = 0x4;
215     public static final int DEVICE_OUT_WIRED_HEADPHONE = 0x8;
216     public static final int DEVICE_OUT_BLUETOOTH_SCO = 0x10;
217     public static final int DEVICE_OUT_BLUETOOTH_SCO_HEADSET = 0x20;
218     public static final int DEVICE_OUT_BLUETOOTH_SCO_CARKIT = 0x40;
219     public static final int DEVICE_OUT_BLUETOOTH_A2DP = 0x80;
220     public static final int DEVICE_OUT_BLUETOOTH_A2DP_HEADPHONES = 0x100;
221     public static final int DEVICE_OUT_BLUETOOTH_A2DP_SPEAKER = 0x200;
222     public static final int DEVICE_OUT_AUX_DIGITAL = 0x400;
223     public static final int DEVICE_OUT_ANLG_DOCK_HEADSET = 0x800;
224     public static final int DEVICE_OUT_DGTL_DOCK_HEADSET = 0x1000;
225     public static final int DEVICE_OUT_USB_ACCESSORY = 0x2000;
226     public static final int DEVICE_OUT_USB_DEVICE = 0x4000;
227     public static final int DEVICE_OUT_REMOTE_SUBMIX = 0x8000;
```

HAL 层通过一定转换跟 android 上层 对应

在 audio/alsa_route.c 中通过

route_set_controls(route) 中通过 get_route_config(int route) 将以上的各个 route 值转换为 codec_config 中的 xxx.h 配置文件

```
const struct config_route *get_route_config(unsigned route)
{
    ALOGV("get_route_config() route %d", route);

    if (!route_table) {
        ALOGE("get_route_config() route_table is NULL!");
        return NULL;
    }
    switch (route) {
    case SPEAKER_NORMAL_ROUTE:
        return &(route_table->speaker_normal);
    case SPEAKER_INCALL_ROUTE:
        return &(route_table->speaker_incall);
    case SPEAKER_RINGTONE_ROUTE:
        return &(route_table->speaker_ringtone);
    case SPEAKER_VOIP_ROUTE:
        return &(route_table->speaker_voip);
    case EARPIECE_NORMAL_ROUTE:
        return &(route_table->earpiece_normal);
    case EARPIECE_INCALL_ROUTE:
        return &(route_table->earpiece_incall);
    case EARPIECE_RINGTONE_ROUTE:
        return &(route_table->earpiece_ringtone);
    case EARPIECE_VOIP_ROUTE:
        return &(route_table->earpiece_voip);
    case HEADPHONE_NORMAL_ROUTE:
        return &(route_table->headphone_normal);
    case HEADPHONE_INCALL_ROUTE:
        return &(route_table->headphone_incall);
case HEADPHONE_RINGTONE_ROUTE:
龎龎龎龎龎龎龎龎龎龎龎龎?龎龎龎?
}
}
```

HAL 层中定义的 route table，在通过 route_set_controls 将这些 table 配置，最终设置到 codec 的寄存器

```
struct config_route_table
{
    const struct config_route speaker_normal;
    const struct config_route speaker_incall;
    const struct config_route speaker_ringtone;
    const struct config_route speaker_voip;
```

```
    const struct config_route earpiece_normal;
    const struct config_route earpiece_incall;
    const struct config_route earpiece_ringtone;
    const struct config_route earpiece_voip;

    const struct config_route headphone_normal;
    const struct config_route headphone_incall;
    const struct config_route headphone_ringtone;
    const struct config_route speaker_headphone_normal;
    const struct config_route speaker_headphone_ringtone;
    const struct config_route headphone_voip;

    const struct config_route headset_normal;
    const struct config_route headset_incall;
    const struct config_route headset_ringtone;
    const struct config_route headset_voip;

    const struct config_route bluetooth_normal;
    const struct config_route bluetooth_incall;
    const struct config_route bluetooth_voip;

    const struct config_route main_mic_capture;
    const struct config_route hands_free_mic_capture;
    const struct config_route bluetooth_sco_mic_capture;

    const struct config_route playback_off;
    const struct config_route capture_off;
    const struct config_route incall_off;
    const struct config_route voip_off;

    const struct config_route hdmi_normal;

    const struct config_route usb_normal;
    const struct config_route usb_capture;

    const struct config_route spdif_normal;
};
```

HAL 层会根据 sound card name 选择使用哪个 table，如果没有匹配默认使用 default_config.h

```
struct alsa_sound_card_config sound_card_config_list[] = {
    {
        .sound_card_name = "RKRK616",
        .route_table = &rk616_config_table,
    },
    {
        .sound_card_name = "RK29RT3224",
        .route_table = &rt3224_config_table,
    },
    {
        .sound_card_name = "RK29RT3261",
        .route_table = &rt3261_config_table,
```

```
    },
    {
        .sound_card_name = "RK29WM8960",
        .route_table = &wm8960_config_table,
    },
    {
        .sound_card_name = "RKRT3224",
        .route_table = &rt3224_config_table,
    },
```

RK codec 芯片使用 default_config.h

如下这种 tinyalsa 形式的代码：

参数：

```
static const char *rk616_playback_path_mode[] = {
        "OFF", "RCV", "SPK", "HP", "HP_NO_MIC", "BT", "SPK_HP", //0-6
        "RING_SPK", "RING_HP", "RING_HP_NO_MIC", "RING_SPK_HP"};//7-10
```

Playback Path 对应的值有 11 个，分别对应：
关闭、听筒（NORMAL 模式）、喇叭（NORMAL 模式）、带 MIC 耳机（NORMAL 模式）、不带 MIC 耳机（NORMAL 模式）、蓝牙（NORMAL 模式）、喇叭与耳机（NORMAL 模式）、喇叭（RINGTONE 模式）、带 MIC 耳机（RINGTONE 模式）、不带 MIC 耳机（RINGTONE 模式）、喇叭与耳机（RINGTONE 模式）

当 HAL 层调用设置 Playback Path 对应的 control 是，ALSA 会调用注册时对应的 put 函数，即 rk616_playback_path_put 函数。传入的值为 int 类型，与 controls 注册时的字符串顺序一一对应。如 HAL 层调用‘OFF’，那么调用 put 函数传入的值就为 0，而‘SPK’对应的就是 2。

对于 playback 来说：
（1）、SPK_PATH 与 RING_SPK 相同都为喇叭播放音乐通路
（2）、HP_PATH、HP_NO_MIC、RING_HP、RING_HP_NO_MIC 相同，都为耳机播放音乐通路
（3）、SPK_HP、RING_SPK_HP 相同，都为耳机和喇叭同时播放音乐通路

因此处理参数上有些就可以直接一起归类。
下面简单说明 Playback Path 的 put 函数里需要做的事情：

```
static int rk616_playback_path_put(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_value *ucontrol)
{
    switch (ucontrol->value.integer.value[0]) {
    case OFF:
       关闭播放通路包括喇叭和耳机
        break;
    case RCV:
       原先 tinyalsa 关闭 incall 时会调用，HAL 层修改后，现在不会调用。
        break;
    case SPK_PATH:
    case RING_SPK:
       开启喇叭播放音乐通路
        break;
    case HP_PATH:
    case HP_NO_MIC:
```

```
    case RING_HP:
    case RING_HP_NO_MIC:
       开启耳机播放音乐通路
        break;
    case BT:
       蓝牙播放音乐通路不会调用 Playback Path，这边不做处理
        break;
    case SPK_HP:
    case RING_SPK_HP:
       开启耳机、喇叭同时播放音乐通路
        break;
    default:
        return -EINVAL;
    }
    return 0;
}
```

如何 debug 音频问题

第一步首先看 声卡有没有注册

通过看 kernel log 确认 alsa sound card 有没有注册

```
<6>[    2.713102] Enter::rk29_rt3261_init----218
<6>[    2.721654] asoc: rt5639-aif1 <-> rk29_i2s.1 mapping ok
<6>[    2.728488] asoc: rt5639-aif2 <-> rk29_i2s.1 mapping ok
<6>[    2.729289] ALSA device list:
<6>[    2.729318]   #0: RK29_RT5639
```

声卡没有注册参考《Audio 部分常见问题处理方法》声卡注册问题 debug。

第二步 确认 route 是否正常

声卡注册好之后确认 alsa route 是否正确

通过命令 logcat -s alsa_route

```
root@HCTT1:/ # logcat -s alsa_route
logcat -s alsa_route
--------- beginning of /dev/log/system
--------- beginning of /dev/log/main
D/alsa_route(   90): route_info->sound_card 0, route_info->devices 0
E/alsa_route(   90): route_pcm_open() DEVICES_0
D/alsa_route(   90): route_set_controls() set route 0
E/alsa_route(   90): route_pcm_close()route 24
D/alsa_route(   90): route_set_controls() set route 24
```

上面命令跑了两个路由 route 0 route 24 各个 route 表示的意义

在 hardware\rk29\audio \alsa_audio.h 中定义，表示打开了一次 speaker 后面又关闭了一次 speaker

```
typedef enum _AudioRoute {
```

```
    SPEAKER_NORMAL_ROUTE = 0,
    SPEAKER_INCALL_ROUTE, // 1
    SPEAKER_RINGTONE_ROUTE,
    SPEAKER_VOIP_ROUTE,

    EARPIECE_NORMAL_ROUTE, // 4
    EARPIECE_INCALL_ROUTE,
    EARPIECE_RINGTONE_ROUTE,
    EARPIECE_VOIP_ROUTE,

    HEADPHONE_NORMAL_ROUTE, // 8
    HEADPHONE_INCALL_ROUTE,
    HEADPHONE_RINGTONE_ROUTE,
    SPEAKER_HEADPHONE_NORMAL_ROUTE,
    SPEAKER_HEADPHONE_RINGTONE_ROUTE,
    HEADPHONE_VOIP_ROUTE,

    HEADSET_NORMAL_ROUTE, // 14
    HEADSET_INCALL_ROUTE,
    HEADSET_RINGTONE_ROUTE,
    HEADSET_VOIP_ROUTE,

    BLUETOOTH_NORMAL_ROUTE, // 18
    BLUETOOTH_INCALL_ROUTE,
    BLUETOOTH_VOIP_ROUTE,

    MAIN_MIC_CAPTURE_ROUTE, // 21
    HANDS_FREE_MIC_CAPTURE_ROUTE,
    BLUETOOTH_SOC_MIC_CAPTURE_ROUTE,

    PLAYBACK_OFF_ROUTE, // 24
    CAPTURE_OFF_ROUTE,
    INCALL_OFF_ROUTE,
    VOIP_OFF_ROUTE,

    HDMI_NORMAL_ROUTE, // 28

    USB_NORMAL_ROUTE, // 29
    USB_CAPTURE_ROUTE,

    MAX_ROUTE, // 31
} AudioRoute;
```

一般 route 的错误

1）、 耳机喇叭切换 有误 这个时候需要确认耳机检测是否正常

通过 cat sys/class/switch/h2w/state 查看耳机插入状态：

```
root@HCTT1:/sys/class/switch/h2w # cat state
```

```
cat state
0
```

state <= 0 表示无耳机插入

state = 1 表示带 Mic 耳机插入

state = 2 表示不带 Mic 耳机插入

如果耳机状态不对则需要确认耳机检测是否正常，首先要确认耳机检测的 GPIO 拔插时候电平是否有高点变化。如果没有插需要进行硬件排查。如果有则需要参考《RK android 带 MIC 耳机检测以及 hook-kernel.pdf》文档中确认 拔插耳机是否有中断。

如果没有则 需要确认 GPIO 配置是否正确，

board.c (3.10 之前的 kernel)参考《Audio 部分常见问题处理方法》处理。

3.10 之后需要 确认 dts 中的 rockchip_headset 配置是否正确

```
666 &adc {
 667         status = "okay";
 668
 669         rockchip_headset {
 670                 compatible = "rockchip_headset";
 671                 headset_gpio = <&gpio7 GPIO_A7 GPIO_ACTIVE_LOW>;
 672                 pinctrl-names = "default";
 673                 pinctrl-0 = <&gpio7_a7>;
 674                 io-channels = <&adc 2>;
 675         /*
 676                 hook_gpio = ;
 677                 hook_down_type = ; //interrupt hook key down status
 678                 */
 679         };
 680
```

三段 四段耳机的区分 有两种方案 一种是通过 adc 另外一种是通过 GPIO 区分是 3 段耳机还是 4 段耳机插入，详情参考《RK android 带 MIC 耳机检测以及 hook-kernel.pdf》、《android 耳机监听路由切换-framework-hal.pdf》

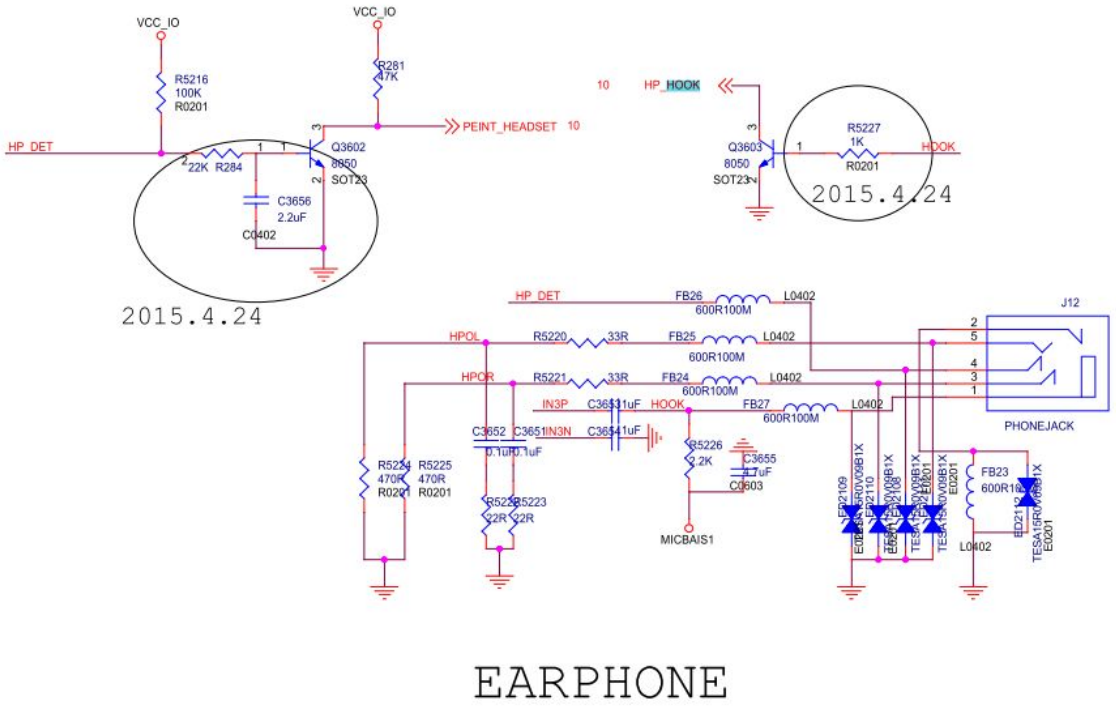
通过 adc 来区分 dts 中配置

```
io-channels = <&adc 2>;
```

通过 GPIO 来区分

```
676                 hook_gpio = ;
677                 hook_down_type = ; //interrupt hook key down status
```

如下所示是使用 GPIO 来区分 3,4 段耳机的插入，耳机上按键按下获取松开 gpio 电平有高低变化系统可以通过这个变化 向系统上报按键 默认 media。



EARPHONE

headset->sdev.name = "h2w";

headset->sdev.print_name = h2w_print_name;

ret = switch_dev_register(&headset->sdev); // 注册 uevent，即上面的 h2w 节点。

switch_set_state(&headset_info->sdev, 0); // 上报值。

rk_headset.c (drivers\headset_observe) // 针对 hook 检测耳机按键是接到主控 gpio 上的。

rk_headset_irq_hook_adc.c (drivers\headset_observe) // 接到 adc 上。

2）、多声卡切换

查看当前系统有哪些声卡：

shell@rk3288_box:/ # ll /proc/asound/

lrwxrwxrwx root root 2013-01-21 10:56 RKRK1000 -> card0

lrwxrwxrwx root root 2013-01-21 10:56 RKSPDIFCARD -> card1

dr-xr-xr-x root root 2013-01-21 10:56 card0

dr-xr-xr-x root root 2013-01-21 10:56 card1

-r--r--r-- root root 0 2013-01-21 10:56 cards

-r--r--r-- root root 0 2013-01-21 10:56 devices
-r--r--r-- root root 0 2013-01-21 10:56 hwdep
-r--r--r-- root root 0 2013-01-21 10:56 pcm
-r--r--r-- root root 0 2013-01-21 10:56 timers
-r--r--r-- root root 0 2013-01-21 10:56 version

看声卡是否正确打开：

播放音乐的情况：

shell@rk322x_box:/ # cat /proc/asound/card0/pcm0p/sub0/hw_params
closed

录音的情况

shell@rk322x_box:/ # cat /proc/asound/card0/pcm0c/sub0/hw_params
closed

目前 closed 状态表示此声卡没打开，如果想查看 card1，则将上面的 card0 改为 card1 即可，即适用于 3.10 以上的所有内核支持的所有声卡类型。

当播放或录音都差不多：下面例子是播放时的例子：

shell@rk322x_box:/ # cat /proc/asound/card0/pcm0p/sub0/hw_params
cat /proc/asound/card1/pcm0p/sub0/hw_params
access: RW_INTERLEAVED
format: S16_LE
subformat: STD
channels: 2
rate: 44100 (44100/1)
period_size: 2048
buffer_size: 8192

可以看到我们 pcm_open 时的参数。

系统默认支持 codec sound card card0

Hdmi(spdif) 为 card 1 插入 HDMI 时候系统会自动切换到 card1 可以通过 route 来确认 route =28

插 HDMI 不需要 进行声卡切换，

可以更改 framework

```
diff --git a/services/java/com/android/server/WiredAccessoryManager.java
b/services/java/com/android/server/WiredAccessoryManager.java
index c8d3510..2fb231e 100644
--- a/services/java/com/android/server/WiredAccessoryManager.java
```

```
+++ b/services/java/com/android/server/WiredAccessoryManager.java
@@ -374,7 +374,7 @@ final class WiredAccessoryManager implements
WiredAccessoryCallbacks {
             //
             // If the kernel does not have an "hdmi_audio" switch, just fall back
on the older
             // "hdmi" switch instead.
-            uei = new UEventInfo(NAME_HDMI_AUDIO, BIT_HDMI_AUDIO, 0);
+/*           uei = new UEventInfo(NAME_HDMI_AUDIO, BIT_HDMI_AUDIO, 0);
             if (uei.checkSwitchExists()) {
                 retVal.add(uei);
             } else {
@@ -385,7 +385,7 @@ final class WiredAccessoryManager implements
WiredAccessoryCallbacks {
                     Slog.w(TAG, "This kernel does not have HDMI audio support");
                 }
             }
-
+*/
             return retVal;
         }
```

或者 将 kernel 里面的 HDMI 读取到的 state 不进行上报

```
diff --git a/drivers/video/rockchip/hdmi/rk_hdmi_task.c
b/drivers/video/rockchip/hdmi/rk_hdmi_task.c
index 6cdb9eb..2ca862f 100755
--- a/drivers/video/rockchip/hdmi/rk_hdmi_task.c
+++ b/drivers/video/rockchip/hdmi/rk_hdmi_task.c
@@ -258,7 +258,7 @@ void hdmi_work(struct work_struct *work)
                                         hdmi_dbg(hdmi->dev,"base_audio_support
=%d,sink_hdmi = %d\n",hdmi->edid.base_audio_suppo
                                         #ifdef CONFIG_SWITCH
                                         if(hdmi->edid.base_audio_support == 1 &&
hdmi->edid.sink_hdmi == 1)
-                                                switch_set_state(&(hdmi-
>switch_hdmi), 1);
+                                                switch_set_state(&(hdmi-
>switch_hdmi), 0);
                                         #endif
                                         #ifdef CONFIG_RK_HDMI_CTL_CODEC
                                         #ifdef CONFIG_MACH_RK_FAC
```

Usb audio 为 card2 4.4 SDK 默认支持 usb audio route =29 、30, 5.1 之后的 SDK kernel 固定 usb audio 为 card3，使用 google 原生的 usb audio hal。

```
    //usb audio
    .usb_normal = {
        .sound_card = 2,
        .devices = DEVICES_0,
        .controls_count = 0,
    },
```

```
    .usb_capture = {
        .sound_card = 2,
        .devices = DEVICES_0,
        .controls_count = 0,
    },
```

3）、两个声卡同时输出

需要同时向两个声卡同时 write audio data 即可， 4.4 的实例代码如下:

```
diff --git a/AudioHardware.cpp b/AudioHardware.cpp
index bc2e553..60df29e 100755
--- a/AudioHardware.cpp
+++ b/AudioHardware.cpp
@@ -84,6 +84,7 @@ AudioHardware::AudioHardware() :
     mInit(false),
     mMicMute(false),
     mPcm(NULL),
+    mPcm1(NULL),
     mPcmOpenCnt(0),
     mMixerOpenCnt(0),
     mInCallAudioMode(false),
@@ -117,6 +118,10 @@ AudioHardware::~AudioHardware()
         TRACE_DRIVER_OUT
     }

+    if(mPcm1)
+    {
+   route_pcm1_close();
+    }
     TRACE_DRIVER_IN(DRV_MIXER_CLOSE)
     route_uninit();
     TRACE_DRIVER_OUT
@@ -685,6 +690,7 @@ struct pcm *AudioHardware::openPcmOut_1()
             mPcmOpenCnt--;
             mPcm = NULL;
         }
+   mPcm1 = route_pcm1_open(1, flags);
     }
     return mPcm;
 }
@@ -703,6 +709,8 @@ void AudioHardware::closePcmOut_1()
         route_pcm_close(PLAYBACK_OFF_ROUTE);
         TRACE_DRIVER_OUT
         mPcm = NULL;
+   route_pcm1_close();
+   mPcm1= NULL;
```

```
     }
 }

@@ -987,7 +995,9 @@ ssize_t AudioHardware::AudioStreamOutALSA::write(const void*
buffer, size_t byte
         }

         TRACE_DRIVER_IN(DRV_PCM_WRITE)
+        // ret = pcm_write(mHardware->getPcm(),(void*) p, bytes);
         ret = pcm_write(mHardware->getPcm(),(void*) p, bytes);
+        ret = pcm_write(mHardware->getPcm1(),(void*) p, bytes);
         TRACE_DRIVER_OUT

         if (ret == 0) {
diff --git a/AudioHardware.h b/AudioHardware.h
index 4d216df..14adf53 100755
--- a/AudioHardware.h
+++ b/AudioHardware.h
@@ -137,6 +137,7 @@ public:
            void closePcmOut_l();

            struct pcm *getPcm() { return mPcm; };
+           struct pcm *getPcm1() { return mPcm1; };

             android::sp <AudioStreamOutALSA>  output() { return mOutput; }

@@ -151,6 +152,7 @@ private:
      android::SortedVector <  android::sp<AudioStreamInALSA> >   mInputs;
      android::Mutex           mLock;
     struct pcm*     mPcm;
+    struct pcm*     mPcm1;
     uint32_t        mPcmOpenCnt;
     uint32_t        mMixerOpenCnt;
     bool            mInCallAudioMode;
diff --git a/alsa_audio.h b/alsa_audio.h
index 101fa2a..25583b1 100755
--- a/alsa_audio.h
+++ b/alsa_audio.h
@@ -163,5 +163,7 @@ int route_set_input_source(const char *source);
 int route_set_voice_volume(const char *ctlName, float volume);
 int route_set_controls(unsigned route);
 struct pcm *route_pcm_open(unsigned route, unsigned int flags);
+struct pcm *route_pcm1_open(int card, unsigned int flags);
 int route_pcm_close(unsigned route);
+int route_pcm1_close(void);
 #endif
diff --git a/alsa_route.c b/alsa_route.c
index 8e890b6..4006c2f 100755
--- a/alsa_route.c
+++ b/alsa_route.c
@@ -40,6 +40,7 @@
```

```
 const struct config_route_table *route_table;

 struct pcm* mPcm[PCM_MAX + 1];
+struct pcm* mPcm1;
 struct mixer* mMixerPlayback;
 struct mixer* mMixerCapture;

@@ -510,6 +511,34 @@ struct pcm *route_pcm_open(unsigned route, unsigned int flags)
     return is_playback ? mPcm[PCM_DEVICE0_PLAYBACK] : mPcm[PCM_DEVICE0_CAPTURE];
 }

+struct pcm *route_pcm1_open(int card, unsigned flags)
+{
+
+    ALOGD("lxt route_pcm1_open()");
+    flags &= ~PCM_CARD_MASK;
+    switch(card) {
+    case 1:
+        flags |= PCM_CARD1;
+        break;
+    case 2:
+        flags |= PCM_CARD2;
+        break;
+    default:
+        flags |= PCM_CARD1;
+        break;
+    }
+    flags &= ~PCM_DEVICE_MASK;
+    flags |= PCM_DEVICE0;
+    mPcm1 = pcm_open(flags);
+    return mPcm1;
+}
+
+int route_pcm1_close(void)
+{
+   ALOGD("lxt route_pcm1_close()");
+   pcm_close(mPcm1);
+   return 0;
+}
 int route_pcm_close(unsigned route)
 {
     unsigned i;
```

4）、确认 route 都正确但是还是没有声音

可以通过正在录音和放音的寄存器打印出来 跟正常的机器比较看是否是因为 codec 寄存器的配置导致没有声音的，寄存器不一样则需要查阅 datasheet 或者联系 codec FAE 进行排查。

如果寄存器跟正常的一直说明 codec 寄存器配置正确，这个时候需要进行硬件排查，首先确认一

下 codec 各路电压、I2S 的 MCLK BCLK LRCLK 是否正常。

（1）寄存器打印：
Kernel3.0 中：cat sys/kernel/debug/asoc/RK_RK616/rk616-codec.0/codec_reg
Kernel3.10 中：cat sys/kernel/debug/asoc/RK_RT5616/rt5616.4-001b/codec_reg
（2）设置寄存器：
可以使用 echo ‘reg value’的方法写入寄存器。
Kernel3.0 中：echo ’01 bb’ > sys/kernel/debug/asoc/RK_RK616/rk616-codec.0/codec_reg
Kernel3.10 中：echo ’01 bb’ > sys/kernel/debug/asoc/RK_RT5616/rt5616.4-001b/codec_reg

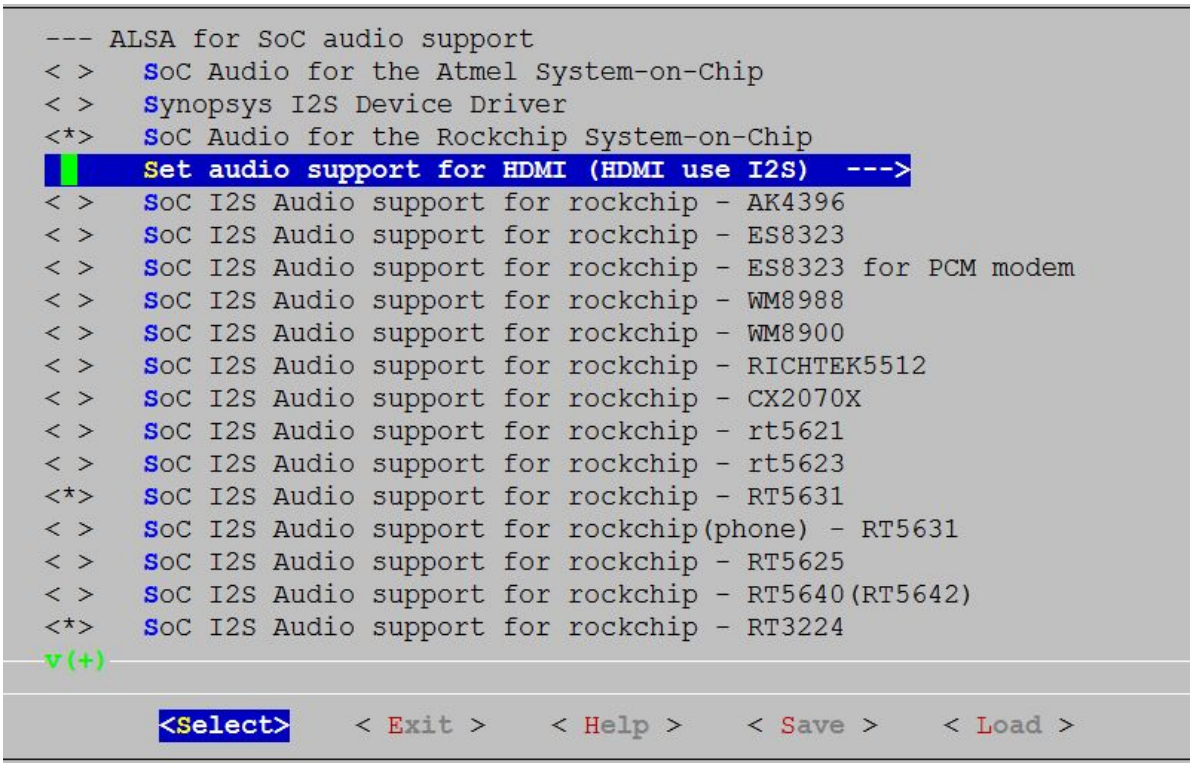
如何调试新的 sound card 驱动

第一步看 codec 芯片资料

确认 codec 内部是否还需要控制一般是 i2c 初始化寄存器或者是通路的配置，如果不需要的可以直接使用 kernel 里面的 HDMI I2S (SND_RK_SOC_HDMI_I2S [=y])这个声卡驱动即可。这个是一个最简单的声卡驱动，有了这个驱动可以是 I2S 控制器能够正常的工作。

Kernel 里面 sound card 驱动的 menuconfig 路径如下，可以看到当前 kernel 支持的 codec 列表
Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support

手上的 3288 4.4 kernel 如下：



Codec 需要通路的配置

完成一个 sound card 驱动，需要完成 3 个部分的驱动 Machine、Soc/Platform、Codec,其中 Soc/Platform 平台驱动有 soc 厂商做好，不需要再写，那么剩下的工作仅需要完成 Machine 和 codec dirver。

以 es8323 为例调试新的驱动

首先在 arch/arm/boot/dts/rk3288-tb_8846.dts 中增加 codec 的描述

```
161         rockchip-es8323{
162                 compatible = "rockchip-es8323";
163                 dais {
164                         dai0 {
165                                 audio-codec = <&es8323>;
166                                 i2s-controller = <&i2s>;
167                                 format = "i2s";
168                                 //continuous-clock;
169                                 //bitclock-inversion;
170                                 //frame-inversion;
171                                 //bitclock-master;
172                                 //frame-master;
173                         };
174                 };
175         };
```

es8323 是 i2c 设备，因此需要增加 i2c 的描述

```
553 &i2c2 {
554         status = "okay";
555         rt5631: rt5631@1a {
556                 compatible = "rt5631";
557                 reg = <0x1a>;
558         };
559         es8323: es8323@10 {
560                 compatible = "es8323";
561                 reg = <0x10>;
562         };
```

Machine driver 的编写

```
#ifdef CONFIG_OF
static const struct of_device_id rockchip_es8323_of_match[] = {
    { .compatible = "rockchip-es8323", },
    {},
};
MODULE_DEVICE_TABLE(of, rockchip_es8323_of_match);
#endif /* CONFIG_OF */
```

```
static struct platform_driver rockchip_es8323_audio_driver = {
    .driver         = {
        .name   = "rockchip-es8323",
        .owner  = THIS_MODULE,
        .pm = &snd_soc_pm_ops,
        .of_match_table = of_match_ptr(rockchip_es8323_of_match),
    },
    .probe          = rockchip_es8323_audio_probe,
    .remove         = rockchip_es8323_audio_remove,
};

module_platform_driver(rockchip_es8323_audio_driver);
```

platform_driver 会通过 rockchip_es8323_of_match 进行查找到在 dts 中注册的资源节点将 sound card 进行注册

```
static int rockchip_es8323_audio_probe(struct platform_device *pdev)
{
    int ret;
    struct snd_soc_card *card = &rockchip_es8323_snd_card;

    card->dev = &pdev->dev;
   // 解析 dts 中的 rockchip-es8323
    ret = rockchip_of_get_sound_card_info(card);
    if (ret) {
        printk("%s() get sound card info failed:%d\n", __FUNCTION__, ret);
        return ret;
    }

    ret = snd_soc_register_card(card);
    if (ret)
        printk("%s() register card failed:%d\n", __FUNCTION__, ret);

    return ret;
}
```

```
static struct snd_soc_ops rk29_ops = {
      .hw_params = rk29_hw_params,
};

static struct snd_soc_dai_link rk29_dai = {
    .name = "ES8323",
    .stream_name = "ES8323 PCM",
    .codec_dai_name = "ES8323 HiFi",
    .init = rk29_es8323_init,
    .ops = &rk29_ops,
};

static struct snd_soc_card rockchip_es8323_snd_card = {
    .name = "RK_ES8323",
    .dai_link = &rk29_dai,
    .num_links = 1,
```

```
};
```

具体可以查阅相关代码，其他 codec 套用这个架构即可。

Codec 驱动 codec 驱动主要是要跟之前的 machine driver 匹配。

kernel\sound\soc\codecs\es8323.c

```
static struct i2c_driver es8323_i2c_driver = {
    .driver = {
    // 这个名字要跟 rk29_dai 里面的大小写要一致否则找不到设备注册不上。
        .name = "ES8323",
        .owner = THIS_MODULE,
    },
    .shutdown = es8323_i2c_shutdown,
    .probe = es8323_i2c_probe,
    .remove = es8323_i2c_remove,
    .id_table = es8323_i2c_id,
};

static int __init es8323_init(void)
{
    return i2c_add_driver(&es8323_i2c_driver);
}

static void __exit es8323_exit(void)
{
    i2c_del_driver(&es8323_i2c_driver);
}

module_init(es8323_init);
module_exit(es8323_exit);
```

es8323_i2c_probe 成功会调用到

ret = snd_soc_register_codec(&i2c->dev,&soc_codec_dev_es8323, &es8323_dai, 1);

进行 codec dai 的注册，

```
static struct snd_soc_dai_driver es8323_dai = {
    // name 字段要跟    .codec_dai_name = "ES8323 HiFi",一致否则也找不到无法注册，
    .name = "ES8323 HiFi",
    .playback = {
        .stream_name = "Playback",
        .channels_min = 1,
        .channels_max = 2,
        .rates = es8323_RATES,
        .formats = es8323_FORMATS,
    },
    .capture = {
        .stream_name = "Capture",
        .channels_min = 1,
        .channels_max = 2,
        .rates = es8323_RATES,
        .formats = es8323_FORMATS,
```

```
    },
    .ops = &es8323_ops,
    .symmetric_rates = 1,
};
```

Sound card 注册不上没有注册参考《Audio 部分常见问题处理方法》声卡注册问题 debug。

如果上面红色部分名称没对上，有可能打印如下信息：

[43.204222] rockchip-rt5631 rockchip-es8323.30: ASoC: CODEC (null) not registered

[43.204273] rockchip_rt5631_audio_probe() register card failed:-517

所以当打印此信息时，名称要先对比查看一下。

如果 codec 的通路比较简单则可以直接在 codec driver 里面进行 通路的配置。为了减小工作量可以不在 HAL 层里面增加 codec 的 config_list.h，当然增加 config list 也是可以的。

如： RK3368_ANDROID5.1-SDK_V1.00_20150415\kernel\sound\soc\codecs\es8316.c

.startup = es8316_pcm_startup, 录音或者放音打开时候 会调用

.shutdown = es8316_pcm_shutdown, 录音或者放音关闭时候会调用

具体可以查阅相关代码

```
static struct snd_soc_dai_ops es8316_ops = {
 .startup = es8316_pcm_startup,
 .hw_params = es8316_pcm_hw_params,
 .set_fmt = es8316_set_dai_fmt,
 .set_sysclk = es8316_set_dai_sysclk,
 .digital_mute = es8316_mute,
 .shutdown = es8316_pcm_shutdown,
};
```

如果 codec 的通路配置比较复杂 例如 alc3224 简单的 startup shutdown 不可能完成所有 case 的通路配置，则需要增加 codec 的 config_list.h，这个符合 alsa 的标准，需要联系 codec 原厂 fae 进行协助，例如 3224 等。

补充：

tinymix 和 amix 都差不多，下面以 tinymix 为例

root@android:/ # tinymix

Number of controls: 105

ctl	type	num	name	value
0	INT	2	SPKOUT Playback Volume	0 0
1	BOOL	2	SPKOUT Playback Switch	On On
2	INT	2	Earpiece Playback Volume	0 0
3	BOOL	2	Earpiece Playback Switch	On On
4	ENUM	1	SPK Amp Type	Class AB

5	ENUM	1	Left SPK Source	LPLN
6	ENUM	1	SPK Amp Ratio	1.00 Vdd
7	INT	2	AUXOUT Playback Volume	31 31
8	BOOL	2	AUXOUT Playback Switch	Off Off
9	INT	2	PCM Playback Volume	47 47
10	INT	1	Phone Playback Volume	23

...

蓝牙通话 3G 通话的方案 debug

参考《RK_Android 平台蓝牙通话功能说明 v1.3.pdf》

312x 平台 codec debug

参考《RK312X_CODEC_开发说明文档 V1.0.pdf》

POP 音问题

喇叭 pop 音可以增加 mute 电路将 pop 音隔断，待声音正常输出时候再将功放打开可将 pop 音消除。可以参考《RK312X_CODEC_开发说明文档 V1.0.pdf》 pop 音问题 或者《rk616-rk618 常见问题及功能修改.pdf》中的，

```
#define SPK_AMP_DELAY 150（用于喇叭 spk 功放使能后延时时间设置，有些功放需要延时后才能正常
输出，具体时间不同。）
#define HP_MOS_DELAY 5（耳机 mos 管拉高后延时时间设置，有些 mos 管拉高后需要延时后声音才能正
常输出。）
```

其他 codec 也一样增加 mute 电路 调整上电时间可以将 codec 产生的 pop 音消除可以举一反三。

关于 ALC 功能

不改喇叭功率但是要音量加大，codec 如果有 ALC 功能， 可以联系 codec FAE 开启 ALC 功

能，如果 codec 没有 ALC 功能，可以联系 FAE 窗口获取 ALC 补丁。

关于降噪算法

使用的是 speex 的开源算法库，录音默认开启降噪算法，可以将部分噪声过滤，但是同时也会把背景声音也会过滤一部分。同时如果输入的信号是固定的信号比如 1K 正弦波信号，因为算法是针对的是语音信号，对固定信号有衰减的作用，因此 1K 正弦波这则比实际增益小很多。如需要关闭 patch 如下：

关闭降噪 4.4patch

```
hardware/rk29/audio$ git diff ./
diff --git a/AudioHardware.h b/AudioHardware.h
index 4d216df..92bceb2 100755
--- a/AudioHardware.h
+++ b/AudioHardware.h
@@ -77,7 +77,7 @@ namespace android_audio_legacy {

 //1:Enable the denoise funtion ;0: disable the denoise function

-#define SPEEX_DENOISE_ENABLE 1
+#define SPEEX_DENOISE_ENABLE 0
```

关闭 5.1 录音降噪算法 patch

```
hardware/rockchip/audio/tinyalsa_hal$ git diff audio_hw.c
diff --git a/tinyalsa_hal/audio_hw.c b/tinyalsa_hal/audio_hw.c
index 691b8ca..56cfa65 100755
--- a/tinyalsa_hal/audio_hw.c
+++ b/tinyalsa_hal/audio_hw.c
@@ -101,7 +101,7 @@ FILE *in_debug;

 #define AUDIO_HAL_VERSION "ALSA Audio Version: V0.8.0"

-#define SPEEX_DENOISE_ENABLE
+/* #define SPEEX_DENOISE_ENABLE */
```