不小心

参考文章：http://blog.csdn.net/droidphone/

http://blog.chinaunix.net/uid/22917448.html

分析只列出部分重要代码，具体请参考 linux3.0 内核代码。

Alsa 架构整体来说十分复杂，但对于驱动移植来说我们仅仅只需要关心 ASOC 就足够了。

在学习 asoc 之前我们先了解一些专业术语：

ASoC currently supports the three main Digital Audio Interfaces (DAI) found on SoC controllers and portable audio CODECs today, namely AC97, I2S and PCM.

ASoC 现在支持如今的 SoC 控制器和便携音频解码器上的三个主要数字音频接口，即 AC97，I2S，PCM（与 pcm 音频格式注意区分，前者是一种音频接口，后者是一种输入声卡的音频格式）。

AC97

AC97

====

AC97 is a five wire interface commonly found on many PC sound cards. It is now also popular in many portable devices. This DAI has a reset line and time multiplexes its data on its SDATA_OUT (playback) and SDATA_IN (capture) lines. The bit clock (BCLK) is always driven by the CODEC (usually 12.288MHz) and the frame (FRAME) (usually 48kHz) is always driven by the controller. Each AC97 frame is 21uS long and is divided into 13 time slots.

AC97 是一种个人电脑声卡上常见的五线接口。现在在很多便携设备中也很流行。这个数字音频接口有一个复位线，分时在 SDATA__OUT（回放）和 SDATA__IN（捕获）线上传送数据。位时钟常由解码器驱动（通常是 12.288MHz).帧时钟（通常 48kHz)总是由控制器驱动。每个 AC97 帧 21uS, 并分为 13 个时间槽。

I2S

I2S

===

I2S is a common 4 wire DAI used in HiFi, STB and portable devices. The Tx and Rx lines are used for audio transmission, whilst the bit clock (BCLK) and left/right clock (LRC) synchronise the link. I2S is flexible in that either the controller or CODEC can drive (master) the BCLK and LRC clock lines. Bit clock usually varies depending on the sample rate and the master system clock (SYSCLK). LRCLK is the same as the sample rate. A few devices support separate ADC and DAC LRCLKs, this allows for simultaneous capture and playback at different sample rates.

I2S 是一个 4 线数字音频接口，常用于 HiFi，STB 便携设备。Tx 和 Rx 信号线用于音频传输。而位时钟和左右时钟（LRC）用于同步链接。I2S 具有灵活性，因为控制器和解码器都可以控制位时钟和左右时钟。位时钟因采样率和主系统时钟而有不同。LRCLK 与采样率相同。少数设备支持独立的 ADC 和 DAC 的 LRCLK。这使在不同采样率情况下同步捕获和回放成为可能。

I2S has several different operating modes:-

I2S 有几个不同的操作模式：

o I2S - MSB is transmitted on the falling edge of the first BCLK after LRC transition.

I2S 模式－MSB 在 LRC 后的第一个位时钟的下降沿传送。

o Left Justified - MSB is transmitted on transition of LRC.

左对齐模式：MSB 在 LRC 传送时传送。

o Right Justified - MSB is transmitted sample size BCLKs before LRC
transition.
右对齐模式：MSB 在（此句不懂）

PCM
PCM
===

PCM is another 4 wire interface, very similar to I2S, which can support a more flexible protocol. It has bit clock (BCLK) and sync (SYNC) lines that are used to synchronise the link whilst the Tx and Rx lines are used to transmit and receive the audio data. Bit clock usually varies depending on sample rate whilst sync runs at the sample rate. PCM also supports Time Division Multiplexing (TDM) in that several devices can use the bus simultaneously (this is sometimes referred to as network mode).
PCM 也是一种 4 线制接口。与 I2S 非常相像，但支持一个更灵活的协议。它有位时钟（BCLK）和同步时钟 （SYNC）用来在 Tx 和 Rx 在传送和接收音频数据是同步连接。位时钟通常因采样率的不同而不同，然而同步时钟（SYNC）与采样频率相同。PCM 同样支 持时分复用，可以几个设备同时使用总线（这有时被称为 net work 模式）。

Common PCM operating modes:-
常用的 PCM 操作模式：

o Mode A - MSB is transmitted on falling edge of first BCLK after FRAME/SYNC.
模式 A－MSB 在 FRAME/SYNC 后第一个 BCLK 的下降沿传送。

o Mode B - MSB is transmitted on rising edge of FRAME/SYNC.
模式 B－MSB 在 FRAME/SYNC 的上升沿传送。

Codec(解码器)
各解码器驱动必须提供如下特性：
1) Codec DAI and PCM configuration
2) Codec control IO - using I2C, 3 Wire(SPI) or both APIs
3) Mixers and audio controls
4) Codec audio operations
1）解码器数字音频接口和 PCM 配置。
2）解码器控制 IO－使用 I2C，3 总线（SPI）或两个都有。
3）混音器和音频控制。
4）解码器音频操作。

Optionally, codec drivers can also provide:-
解码器驱动可以选择性提供：

5) DAPM description.
6) DAPM event handler.
7) DAC Digital mute control.
5）动态音频电源管理描述。
6）动态音频电源管理事件控制。
7）数模转换数字消音控制。

SoC DAI Drivers
板级 DAI 驱动
==========

Each SoC DAI driver must provide the following features:-

每个 SoC DAI 驱动都必须提供如下性能：

1) Digital audio interface (DAI) description

1)数字音频接口描述

2) Digital audio interface configuration

2)数字音频接口配置

3) PCM's description

3）PCM 描述

4) SYSCLK configuration

4）系统时钟配置

5) Suspend and resume (optional)

5）挂起和恢复（可选的）

以上由君子翻译，本人实在没办法比他描述的更好了，所以把重要的部分提取出来直接 copy。在这里对君子表示由衷感谢，赋上君子注。

君子注:

您现在所阅读的，是君子阅读 Linux 音频 SoC 驱动时，写下的文档译文。

君子写些译文，一方面是作为自己的笔记，帮助记忆，另一方面也希望能对他人有所帮助。

如果您能于君子的译文中有所收获，则吾心甚慰

现在我们开始分析 ASOC：

ASoC 被分为 Machine、Platform 和 Codec 三大部分。其中的 Machine 驱动负责 Platform 和 Codec 之间的耦合和设备或板子特定的代码。

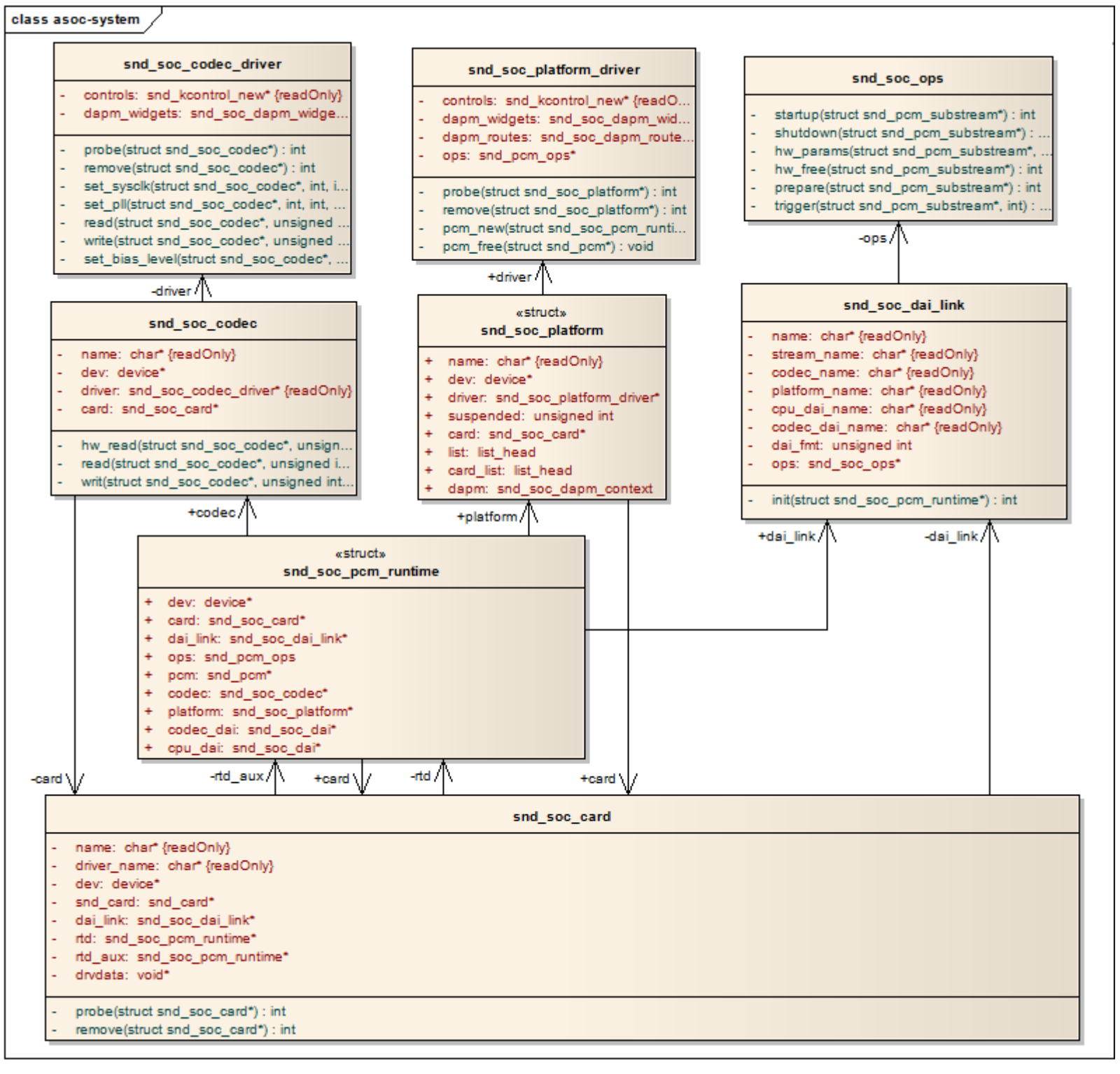
看起来挺复杂，其实需要我们做的事情并不多，大部分内核已经完成。下面我们分析哪些是我们需要自己做的：

codec 驱动：负责音频解码。这部分代码完全无平台无关，设备原厂提供，我们只需要把它加进内核编译就好了。

platform 驱动：与处理器芯片相关，这部分代码在该芯片商用之前方案产商提供的 demo 板已完全确定了，也就是说我们只需要使用就可以了。

machine 驱动：好了，到了最关键的地方了，machine 驱动是耦合 platform 和 codec 驱动，同时与上层交互的代码。由于上层是标准的 alsa 架构，所以下层接口肯定要做了统一，所以我很负责的告诉你，这部分由 machine 本身的 platform 驱动和 platform 设备组成(请跟 asoc 的 platform 驱动区别)，platform 驱动内核帮我们完成了，所以你无须过多的关心你的驱动怎么跟上层 alsa 怎么衔接的问题，我们只需要注册一个 machine 的 platform 设备以及完成 platform 和 codec 耦合就 ok

asoc 的关系图如下：（以下适应于 linux3.0。linux2.6 会有所不同）



上图把 asoc 架构显示的淋漓尽致，如果你分析了 asoc 你就会发现上图描述的结构以及函数真的一个都跑不了。

Machie：

Machine platform device: (~/sound/soc/samsung/smdk_wm8994.c)

```
smdk_snd_device = platform_device_alloc("soc-audio", -1);
    if (!smdk_snd_device)
        return -ENOMEM;

    platform_set_drvdata(smdk_snd_device, &smdk);

    ret = platform_device_add(smdk_snd_device);
```

```
static struct snd_soc_dai_link smdk_dai[] = {
    { /* Primary DAI i/f */
        .name = "WM8994 AIF1",
        .stream_name = "Pri_Dai",
        .cpu_dai_name = "samsung-i2s.0",
        .codec_dai_name = "wm8994-aif1",
        .platform_name = "samsung-audio",
        .codec_name = "wm8994-codec",
        .init = smdk_wm8994_init_paiftx,
         .ops = &smdk_ops,
     }, { /* Sec_Fifo Playback i/f */
         .name = "Sec_FIFO TX",
         .stream_name = "Sec_Dai",
```

```
14.         .cpu_dai_name = "samsung-i2s.4",
15.         .codec_dai_name = "wm8994-aif1",
16.         .platform_name = "samsung-audio",
17.         .codec_name = "wm8994-codec",
18.         .ops = &smdk_ops,
19.     },
20. };
21.
22. static struct snd_soc_card smdk = {
23.     .name = "SMDK-I2S",
24.     .owner = THIS_MODULE,
25.     .dai_link = smdk_dai,
26.     .num_links = ARRAY_SIZE(smdk_dai),
27. };
```

通过 snd_soc_card 结构，又引出了 Machine 驱动的另外两个个数据结构：

- snd_soc_dai_link（实例：smdk_dai[] ）
- snd_soc_ops（实例：smdk_ops ）

snd_soc_dai_link 看名字就知道，很明显它是起耦合链接作用的。它指定了 Platform、Codec、codec_dai、cpu_dai 的名字，稍后 Machine 驱动将会利用这些名字去匹配已经在系统中注册的 platform，codec，dai。
snd_soc_ops 连接 Platform 和 Codec 的 dai_link 对应的 ops 操作函数，本例就是 smdk_ops，它只实现了 hw_params 函数：smdk_hw_params。
到此为止，最主要的部分 machine 的平台设备注册我们完成了。

下面我们关注 machine 平台驱动部分 (这部分内核不需要我们实现，但我们需要知道它是怎么工作的）
ASoC 的 platform_driver 在以下文件中定义：sound/soc/soc-core.c。

还是先从模块的入口看起：

[cpp] view plaincopy

```
1. static int __init snd_soc_init(void)
2. {
3.     ......
4.     return platform_driver_register(&soc_driver);
5. }
```

soc_driver 的定义如下：

[cpp] view plaincopy

```
1. /* ASoC platform driver */
2. static struct platform_driver soc_driver = {
3.     .driver     = {
4.         .name       = "soc-audio", //确保你注册 machine 平台设备和它保持一致
5.         .owner      = THIS_MODULE,
6.         .pm     = &soc_pm_ops,
7.     },
8.     .probe      = soc_probe,
9.     .remove     = soc_remove,
10. };
```

初始化入口 soc_probe()

soc_probe 函数本身很简单，它先从 platform_device 参数中取出 snd_soc_card，然后调用 snd_soc_register_card，通过 snd_soc_register_card，为 snd_soc_pcm_runtime 数组申请内存，每一个 dai_link 对应 snd_soc_pcm_runtime 数组的一个单元，然后把 snd_soc_card 中的 dai_link 配置复制到相应的 snd_soc_pcm_runtime 中，最后，大部分的工作都在 snd_soc_instantiate_card 中实现，下面就看看 snd_soc_instantiate_card 做了些什么：

该函数首先利用 card->instantiated 来判断该卡是否已经实例化，如果已经实例化则直接返回，否则遍历每一对 dai_link，进行 codec、platform、dai 的绑定工作，下只是代码的部分选节，详细的代码请直接参考完整的代码树。

```
static int soc_probe(struct platform_device *pdev)
{
    struct snd_soc_card *card = platform_get_drvdata(pdev);//别忘记了 machine 的 platform_set_drvdata   //取出 snd_soc_card
```

```
        .............
        ret = snd_soc_register_card(card);//注册
            .............
        }
```

下面我们看 snd_soc_register_card()函数:

```
int snd_soc_register_card(struct snd_soc_card *card)
{
。。。。。。。。
        card->rtd = kzalloc(sizeof(struct snd_soc_pcm_runtime) *
                        (card->num_links + card->num_aux_devs),
                        GFP_KERNEL);//为 snd_soc_pcm_runtime 数组申请内存，每一个 dai_link 对应一个 snd_soc_pcm_runtime 数组单元
。。。。。。。。
        for (i = 0; i < card->num_links; i++)
                card->rtd[i].dai_link = &card->dai_link[i];//把 snd_soc_card 中的 dai_link 复制到相应的 snd_soc_pcm_runtime
。。。。。。。。
        snd_soc_instantiate_cards();//将调用 snd_soc_instantiate_card()//最为重要
}
```

下面我们分析 snd_soc_instantiate_card()函数：

```
static void snd_soc_instantiate_card(struct snd_soc_card *card)
{
。。。。。。。。
        if (card->instantiated) {    //判断该卡是否已经实例化，如果是就返回
                mutex_unlock(&card->mutex);
                return;
        }
        /* bind DAIs */
        for (i = 0; i < card->num_links; i++) //否则遍例每一对 dai_link，进行 codec,flatrom,dai 的绑定工作
                soc_bind_dai_link(card, i);
/*
**********************************************************************************************************************************************************
ASoC 定义了三个全局的链表头变量：codec_list、dai_list、platform_list，系统中所有的 Codec、DAI、Platform 都在注册时连接到这三个全局链表上。soc_bind_dai_link 函数逐个扫描这三个链表，根据 card->dai_link[]中的名称进行匹配，匹配后把相应的 codec，dai 和 platform 实例赋值到 card->rtd[]中（snd_soc_pcm_runtime）。经过这个过程后，snd_soc_pcm_runtime：（card->rtd）中保存了本 Machine 中使用的 Codec，DAI 和 Platform 驱动的信息。
**********************************************************************************************************************************************************/

        /* bind completed ? */
        if (card->num_rtd != card->num_links) {
                mutex_unlock(&card->mutex);
                return;
        }
。。。。。。。。。
        /* card bind complete so register a sound card */
        ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
                        card->owner, 0, &card->snd_card);
。。。。。。。。。
        card->snd_card->dev = card->dev;
        card->dapm.bias_level = SND_SOC_BIAS_OFF;
        card->dapm.dev = card->dev;
        card->dapm.card = card;
        list_add(&card->dapm.list, &card->dapm_list);//初始化 codec 缓存，创建声卡实例


。。。。。。。。。。。。。。
//下面是最重要的 probe 匹配工作
        if (card->probe) {
                ret = card->probe(card);
                if (ret < 0)
                        goto card_probe_error;
        }
 。。。。。。。。。
                ret = soc_probe_dai_link(card, i, order);//主要的耦合链接工作在此函数完成
。。。。。。。。。。
                ret = soc_probe_aux_dev(card, i);
。。。。。。。。。。

        if (card->late_probe) {//最后的声卡初始化工作，
                ret = card->late_probe(card);
```

```
        }
。。。。。。。。。。
        ret = snd_card_register(card->snd_card);//然后调用标准的 alsa 驱动的声卡函数进行声卡注册
。。。。。。。。。。
        }
```

到这里声卡已经注册好了，声卡可以正常工作了，我们再分析最后一个函数，也就是它们是怎么匹配的 soc_probe_dai_link():

```
static int soc_probe_dai_link(struct snd_soc_card *card, int num, int order)
{
。。。。。。。。。。
        /* probe the cpu_dai */
        if (!cpu_dai->probed &&
                        cpu_dai->driver->probe_order == order) {
                if (!try_module_get(cpu_dai->dev->driver->owner))
                        return -ENODEV;


                if (cpu_dai->driver->probe) {
                        ret = cpu_dai->driver->probe(cpu_dai);
                        if (ret < 0) {
                                printk(KERN_ERR "asoc: failed to probe CPU DAI %s\n",
                                                cpu_dai->name);
                                module_put(cpu_dai->dev->driver->owner);
                                return ret;
                        }
                }
                cpu_dai->probed = 1;
                /* mark cpu_dai as probed and add to card dai list */
                list_add(&cpu_dai->card_list, &card->dai_dev_list);
        }


        /* probe the CODEC */
        if (!codec->probed &&
                        codec->driver->probe_order == order) {
                ret = soc_probe_codec(card, codec);
                if (ret < 0)
                        return ret;
        }


        /* probe the platform */
        if (!platform->probed &&
                        platform->driver->probe_order == order) {
                ret = soc_probe_platform(card, platform);
                if (ret < 0)
                        return ret;
        }


        /* probe the CODEC DAI */
        if (!codec_dai->probed && codec_dai->driver->probe_order == order) {
                if (codec_dai->driver->probe) {
                        ret = codec_dai->driver->probe(codec_dai);
                        if (ret < 0) {
                                printk(KERN_ERR "asoc: failed to probe CODEC DAI %s\n",
                                                codec_dai->name);
                                return ret;
                        }
                }


                /* mark codec_dai as probed and add to card dai list */
                codec_dai->probed = 1;
                list_add(&codec_dai->card_list, &card->dai_dev_list);
        }
```

```
        /* complete DAI probe during last probe */
        if (order != SND_SOC_COMP_ORDER_LAST)
                return 0;


。。。。。。。。。。
        /* create the pcm */
        ret = soc_new_pcm(rtd, num);//如果上面都匹配成功将创建标准的 alsa 的 pcm 逻辑设备
。。。。。。。。。。
        }
```

好了，到此为止我们最主要的部分 machine 部分分析完成了。
接着是 codec 驱动部分：

Codec 简介

在移动设备中，Codec 的作用可以归结为 4 种，分别是：

- 对 PCM 等信号进行 D/A 转换，把数字的音频信号转换为模拟信号
- 对 Mic、Linein 或者其他输入源的模拟信号进行 A/D 转换，把模拟的声音信号转变 CPU 能够处理的数字信号
- 对音频通路进行控制，比如播放音乐，收听调频收音机，又或者接听电话时，音频信号在 codec 内的流通路线是不一样的
- 对音频信号做出相应的处理，例如音量控制，功率放大，EQ 控制等等

ASoC 对 Codec 的这些功能都定义好了一些列相应的接口，以方便地对 Codec 进行控制。ASoC 对 Codec 驱动的一个基本要求是：驱动程序的代码必须要做到平台无关性，以方便同一个 Codec 的代码不经修改即可用在不同的平台上。以下的讨论基于 wolfson 的 Codec 芯片 WM8994，kernel 的版本 3.3.x。
描述 Codec 的最主要的几个数据结构分别是：snd_soc_codec，snd_soc_codec_driver，snd_soc_dai，snd_soc_dai_driver，其中的 snd_soc_dai 和 snd_soc_dai_driver 在 ASoC 的 Platform 驱动中也会使用到，Platform 和 Codec 的 DAI 通过 snd_soc_dai_link 结构，在 Machine 驱动中进行绑定连接。

Codec 的注册

因为 Codec 驱动的代码要做到平台无关性，要使得 Machine 驱动能够使用该 Codec，Codec 驱动的首要任务就是确定 snd_soc_codec 和 snd_soc_dai 的实例，并把它们注册到系统中，注册后的 codec 和 dai 才能为 Machine 驱动所用。以 WM8994 为例，对应的代码位置：/sound/soc/codecs/wm8994.c，模块的入口函数注册了一个 platform driver：

[html] view plaincopy

```
1. static struct platform_driver wm8994_codec_driver = {
2.      .driver = {
3.              .name = "wm8994-codec",  //注意 machine device 里面和这里保持一致
4.              .owner = THIS_MODULE,
5.              },
6.      .probe = wm8994_probe,
7.      .remove = __devexit_p(wm8994_remove),
8. };
9.
10.module_platform_driver(wm8994_codec_driver);
```

有 platform driver，必定会有相应的 platform device，platform device 其实在我们之前讲过的 machine device 注册时已经引入了。

[html] view plaincopy

```
1. static int __devinit wm8994_probe(struct platform_device *pdev)
2. {
3.     return snd_soc_register_codec(&pdev->dev, &soc_codec_dev_wm8994,
4.              wm8994_dai, ARRAY_SIZE(wm8994_dai));
5. }
```

其中，soc_codec_dev_wm8994 和 wm8994_dai 的定义如下（代码中定义了 3 个 dai，这里只列出第一个）：

[html] view plaincopy

```
1. static struct snd_soc_codec_driver soc_codec_dev_wm8994 = {
2.      .probe =    wm8994_codec_probe,
3.      .remove =   wm8994_codec_remove,
4.      .suspend =  wm8994_suspend,
5.      .resume =   wm8994_resume,
6.      .set_bias_level = wm8994_set_bias_level,
7.      .reg_cache_size = WM8994_MAX_REGISTER,
```

```
8.      .volatile_register = wm8994_soc_volatile,
9. };
```

[html] view plaincopy

```
1. static struct snd_soc_dai_driver wm8994_dai[] = {
2.     {
3.         .name = "wm8994-aif1",
4.         .id = 1,
5.         .playback = {
6.             .stream_name = "AIF1 Playback",
7.             .channels_min = 1,
8.             .channels_max = 2,
9.             .rates = WM8994_RATES,
10.             .formats = WM8994_FORMATS,
11.         },
12.         .capture = {
13.             .stream_name = "AIF1 Capture",
14.             .channels_min = 1,
15.             .channels_max = 2,
16.             .rates = WM8994_RATES,
17.             .formats = WM8994_FORMATS,
18.         },
19.         .ops = &wm8994_aif1_dai_ops,
20.     },
21.     ......
22. }
```

可见，Codec 驱动的第一个步骤就是定义 snd_soc_codec_driver 和 snd_soc_dai_driver 的实例，然后调用 snd_soc_register_codec 函数对 Codec 进行注册。

snd_soc_register_codec（）函数是 machine driver 提供的，只要注册成功后 codec 提供的操作函数就能正常提供给 machine driver 使用了。

```
int snd_soc_register_codec(struct device *dev,
                const struct snd_soc_codec_driver *codec_drv,
                struct snd_soc_dai_driver *dai_drv,
                int num_dai)
{
    codec = kzalloc(sizeof(struct snd_soc_codec), GFP_KERNEL);
    。。。。。。。。。。。
    /* create CODEC component name */
    codec->name = fmt_single_name(dev, &codec->id);/*Machine 驱动定义的 snd_soc_dai_link 中会指定每个 link 的 codec 和 dai 的名字，进行匹配绑定时就是通过和这里的名字比较，从而找到该 Codec 的 */
//  然后初始化它的各个字段，多数字段的值来自上面定义的 snd_soc_codec_driver 的实例 soc_codec_dev_wm8994：
    codec->write = codec_drv->write;
    codec->read = codec_drv->read;
    codec->volatile_register = codec_drv->volatile_register;
    codec->readable_register = codec_drv->readable_register;
    codec->writable_register = codec_drv->writable_register;
    codec->dapm.bias_level = SND_SOC_BIAS_OFF;
    codec->dapm.dev = dev;
    codec->dapm.codec = codec;
    codec->dapm.seq_notifier = codec_drv->seq_notifier;
    codec->dev = dev;
    codec->driver = codec_drv;
    codec->num_dai = num_dai;
    mutex_init(&codec->mutex);


    /* allocate CODEC register cache */
    if (codec_drv->reg_cache_size && codec_drv->reg_word_size) {
        reg_size = codec_drv->reg_cache_size * codec_drv->reg_word_size;
        codec->reg_size = reg_size;
        /* it is necessary to make a copy of the default register cache
         * because in the case of using a compression type that requires
         * the default register cache to be marked as __devinitconst the
         * kernel might have freed the array by the time we initialize
         * the cache.
         */
```

```
        if (codec_drv->reg_cache_default) {
            codec->reg_def_copy = kmemdup(codec_drv->reg_cache_default,
                                reg_size, GFP_KERNEL);
            if (!codec->reg_def_copy) {
                ret = -ENOMEM;
                goto fail;
            }
        }
    }

。。。。。。。。。。。。
    /* register any DAIs */
    if (num_dai) {
        ret = snd_soc_register_dais(dev, dai_drv, num_dai);//通过 snd_soc_register_dais 函数对本 Codec 的 dai 进行注册
        if (ret < 0)
            goto fail;
    }


    mutex_lock(&client_mutex);
    list_add(&codec->list, &codec_list);/*最后，它把 codec 实例链接到全局链表 codec_list 中，并且调用 snd_soc_instantiate_cards 是函数触
发 Machine 驱动进行一次匹配绑定操作  */
        snd_soc_instantiate_cards();
        mutex_unlock(&client_mutex);
。。。。。。。。。。
}
```

好了，在这里我们的 codec 驱动也分析完了，其实这部分都是与平台无关代码，一般也不需要改动，这部分我们从设备原厂拿到代码后丢上去就可以了，只是我们在写 machine device 的时候要注意和这里的名字匹配。
接下来是 asoc 的 platform 驱动：

Platform 驱动的主要作用是完成音频数据的管理，最终通过 CPU 的数字音频接口（DAI）把音频数据传送给 Codec 进行处理，最终由 Codec 输出驱动耳机或者是喇叭的音信信号。在具体实现上，ASoC 有把 Platform 驱动分为两个部分：snd_soc_platform_driver 和 snd_soc_dai_driver。其中，platform_driver 负责管理音频数据，把音频数据通过 dma 或其他操作传送至 cpu dai 中，dai_driver 则主要完成 cpu 一侧的 dai 的参数配置，同时也会通过一定的途径把必要的 dma 等参数与 snd_soc_platform_driver 进行交互。

snd_soc_platform_driver 的注册

通常，ASoC 把 snd_soc_platform_driver 注册为一个系统的 platform_driver，不要被这两个想像的术语所迷惑，前者只是针对 ASoC 子系统的，后者是来自 Linux 的设备驱动模型。我们要做的就是：

- 定义一个 snd_soc_platform_driver 结构的实例；
- 在 platform_driver 的 probe 回调中利用 ASoC 的 API：snd_soc_register_platform()注册上面定义的实例；
- 实现 snd_soc_platform_driver 中的各个回调函数；

以 kernel3.3 中的/sound/soc/samsung/dma.c 为例：

[cpp] view plaincopy

```cpp
static struct snd_soc_platform_driver samsung_asoc_platform = {
    .ops        = &dma_ops,
    .pcm_new    = dma_new,
    .pcm_free   = dma_free_dma_buffers,
};

static int __devinit samsung_asoc_platform_probe(struct platform_device *pdev)
{
    return snd_soc_register_platform(&pdev->dev, &samsung_asoc_platform);
}

static int __devexit samsung_asoc_platform_remove(struct platform_device *pdev)
{
    snd_soc_unregister_platform(&pdev->dev);
    return 0;
}

static struct platform_driver asoc_dma_driver = {
    .driver = {
```

```
20.         .name = "samsung-audio",
21.         .owner = THIS_MODULE,
22.     },
23.
24.     .probe = samsung_asoc_platform_probe,
25.     .remove = __devexit_p(samsung_asoc_platform_remove),
26. };
27.
28. module_platform_driver(asoc_dma_driver);
```

snd_soc_register_platform() 该函数用于注册一个 snd_soc_platform，只有注册以后，它才可以被 Machine 驱动使用。它的代码已经清晰地表达了它的实现过程：

- 为 snd_soc_platform 实例申请内存；
- 从 platform_device 中获得它的名字，用于 Machine 驱动的匹配工作；
- 初始化 snd_soc_platform 的字段；
- 把 snd_soc_platform 实例连接到全局链表 platform_list 中；
- 调用 snd_soc_instantiate_cards，触发声卡的 machine、platform、codec、dai 等的匹配工作；

cpu 的 snd_soc_dai driver 驱动的注册

dai 驱动通常对应 cpu 的一个或几个 I2S/PCM 接口，与 snd_soc_platform 一样，dai 驱动也是实现为一个 platform driver，实现一个 dai 驱动大致可以分为以下几个步骤：

- 定义一个 snd_soc_dai_driver 结构的实例；
- 在对应的 platform_driver 中的 probe 回调中通过 API：snd_soc_register_dai 或者 snd_soc_register_dais，注册 snd_soc_dai 实例；
- 实现 snd_soc_dai_driver 结构中的 probe、suspend 等回调；
- 实现 snd_soc_dai_driver 结构中的 snd_soc_dai_ops 字段中的回调函数；

snd_soc_register_dai 这个函数在上一篇介绍 codec 驱动的博文中已有介绍

具体不再分析，这个驱动也不需要用户做任务修改，所以只要知道它的作用就已经够了。就像电话机一样，我们只要知道电话怎么打就够了，至于它怎么连接我们并不太需要关心。