

# ALSA 实现过程分析

作者: 向仔州

如何创建 pcmC0D0C, pcmC0D0p, controlC0 设备节点.....	2
Pcm 声卡设备节点创建后如何调用.....	7
Alsa 应用层调用过程.....	14

## 如何创建 pcmC0D0C, pcmC0D0p, controlC0 设备节点

pcmC0D0c → 用于录音的 pcm 设备

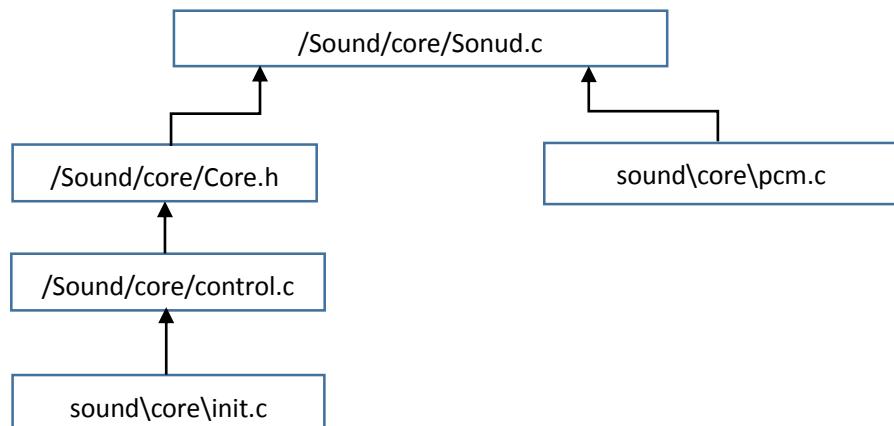
pcmC0D0p → 用于播放的 pcm 设备

controlC0 → 用于声卡的控制，例如通道选择，混音，麦克风的控制等

在 /dev/snd/ 下有三个 pcmC0D0C, pcmC0D0p, controlC0 节点供应

用层 open, write, read。

需要用到的.c 文件，.c 文件的调用过程



声卡节点创建过程也就涉及 4 个文件。Core.h, control.c, init.c, pcm.c

这四个文件实现了之后就会出现 **pcmC0D0C, pcmC0D0p, controlC0 这些设备节点**

**设备节点**

文件调用过程 controls 部分是 init.c 调用 control.c, control.c 又调用

core.h, core.h 又调用 sound.c

数据流获取部分是 pcm.c 调用 sound.c, 所以不管是 controls 接口还

是 pcm 接口最后都要调用 sound.c 里面的函数

下面我们来分析这 4 个文件的调用实现过程

在内核启动后会先执行 sound.c 里面的函数

```
register_chrdev(major, "alsa", &snd_fops); //注册一个 alsa 字符设备
struct file_operations snd_fops = //实现该设备的 file_operations 结构体
{
    .owner = THIS_MODULE,
    .open =      snd_open,
    .llseek =  noop_llseek,
};
```

这里没有发现什么 write 和 read 函数的实现，那么可能在 snd\_open 这个回调函数里面实现的。

```
static int snd_open(struct inode *inode, struct file *file)
{
从 mptr = snd_minors[minor];这个数组中取出新的 file_operations 结构体
}
```

然后要向这个 snd\_minors[minor] 数组里面写一个东西，来初始化这个数组

```
int snd_register_device_for_dev (int type, struct snd_card *card, int dev,const struct
file_operations *f_ops,...)
{
snd_minors[minor] = preg;//用 snd_register_device_for_dev 这个函数里面的 preg 来初始化这
个数组
}
```

以上 sound.c 的工作就做完了。

### Control 调用函数过程

```
snd_card_create
snd_ctl_create
snd_device_new      → SNDDEV_DEV_CONTROL 这个决定了创建什么类型的设备
snd_register_device
snd_register_device_for_dev
```

### PCM 调用函数过程

```
snd_pcm_new
_snd_pcm_new
snd_device_new      → SNDDEV_DEV_PCM 这个决定了创建什么类型的设备
snd_pcm_dev_register
snd_register_device_for_dev
```

## 下面我们来看 controls 函数实现过程

在 sound\core\init.c 里面 snd\_card\_create 函数中 调用 snd\_ctl\_create 函数来创建声卡  
在一个没有任何声卡的 linux 平台上，要设计声卡，先创建一个 struct snd\_card \*card;这个过程在 init.c 里面先初始化

```
int snd_card_create(int idx, const char *xid, struct module *module, int extra_size, struct snd_card **card_ret)
{
    err = snd_ctl_create(card);调用下面的 snd_ctl_create
}
```

在 sound\core\里面有有个 control.c

```
static int snd_ctl_dev_register(struct snd_device *device)
{
    snd_register_device(SNDDEV_TYPE_CONTROL, card, -1, &snd_ctl_f_ops, card, name); //它将调用下面 snd_register_device
}

int snd_ctl_create(struct snd_card *card)
{
    static struct snd_device_ops ops = {
        .dev_free = snd_ctl_dev_free,
        .dev_register = snd_ctl_dev_register, //上面这个函数 snd_ctl_dev_register 被 snd_ctl_create 里面的结构体调用，前提是
                                            //执行了下面的 snd_device_new 后
        .dev_disconnect = snd_ctl_dev_disconnect,
    }

    return snd_device_new(card, SNDDEV_TYPE_CONTROL, card, &ops); //一个声卡里面可能有很多个逻辑设备，NDRV_TYPE_CONTROL 的意思就是
                                                                //创建一个控制类的逻辑设备，这个函数执行后将使 snd_ctl_dev_register 函数被调用
};
```

在 include\sound\core.h 里面

```
static inline int snd_register_device(int type, struct snd_card *card, int dev, const struct file_operations *f_ops, void *private_data, const char *name)
{
    将传进来的&snd_ctl_f_ops 结构体赋值给 snd_register_device_for_dev 里面的 f_ops
    return snd_register_device_for_dev(type, card, dev, f_ops, private_data, name, snd_card_get_device_link(card)); //调用下面的
    snd_register_device_for_dev
}
```

/Sound/core/Sound.c

```
int snd_register_device_for_dev(int type, struct snd_card *card, int dev, const struct file_operations *f_ops, ...)
{
    这里面的 f_ops 就是&snd_ctl_f_ops 结构体，然后将 snd_ctl_f_ops 用来填充 snd_minors[minor]数组
    snd_minors[minor] = preg; //用 snd_register_device_for_dev 这个函数里面的 preg 来初始化这个数组
}
```

声卡调用过程就分析完毕了，最终还是调用的 sound.c 里面的函数，其实就执行 `snd_card_create` 这个函数就可以创建声卡了

## 下面是音频数据流接口 PCM 函数实现过程，也就是 I2S 总线过来的音频流

创建 PCM 设备

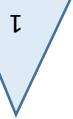
在 sound\core\pcm.c 文件 `snd_pcm_new` 函数中

```
int snd_pcm_new(struct snd_card *card, const char *id, int device,...)
{
    return _snd_pcm_new(card, id, device, playback_count, capture_count, false, rpcm);
}
```

谁来调用 `snd_pcm_new` 这个函数呢？

答：那就是某个声卡的驱动程序来调用 `snd_pcm_new`，有 PCI 总线的声卡 ISA 总线的声卡，但

是我没有看到 I2S 总线的声卡

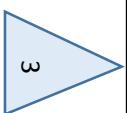


创建一个新的 PCM 设备

在 sound\core\pcm.c 文件

```
static int _snd_pcm_new(struct snd_card *card, const char *id, int device,...) {
    static struct snd_device_ops ops = {
        .dev_free = snd_pcm_dev_free,
        .dev_register = snd_pcm_dev_register, ←
        .dev_disconnect = snd_pcm_dev_disconnect,
    }
    snd_device_new(card, SNDDRV_DEV_PCM, pcm, &ops); // 创建一个声卡的逻辑设
    snd_device_new(card, SNDDRV_DEV_PCM, pcm, &ops); // 创建一个声卡的逻辑设，这个逻辑设备将会导致 snd_pcm_dev_register 被调用
}
```

2



在 sound\core\pcm.c 将调用

`snd_pcm_f_ops[cidx]`这个结构体在 PCM.c 里面定义的

```
snd_pcm_dev_register(struct snd_device *device)
{
    for (cidx = 0; cidx < 2; cidx++){
        err = snd_register_device_for_dev(devtype, &snd_pcm_f_ops[cidx], ...); // 调用 sound.c 里面的 snd_register_device_for_dev 函数
    }
}
```

为什么用 for 循环，因为声卡有录音和播放两种功能，所以要创建两个 PCM

`snd_pcm_f_ops[cidx]`这个数组 0 项表示播放，1 项表示录音

```
}
```

然后调用 `snd_register_device_for_dev` 将 `snd_pcm_f_ops[cidx]` 填充进 `snd_minors[minor]` 中某一项，和控制中的某一项接口一样的。

PCM 接口其实也就只用到了两个文件 PCM.c 和 sound.c, 和前面的 control 接口一样, 最后都要调用到 sound.c 里面的 `snd_register_device_for_dev` 函数

不管是 PCM 的 `snd_pcm_f_ops[cidx]`, 还是 control 的 `&snd_ctl_f_ops`, 都将传进 sound.c 里面的 `snd_register_device_for_dev` 函数中

```
snd_register_device_for_dev(int type, struct snd_card *card, int dev, const struct file_operations *f_ops, ....)
{
    struct snd_minor *preg;
    preg = kmalloc(sizeof *preg, GFP_KERNEL);
    preg->type = type;
    preg->card = card ? card->number : -1;
    preg->device = dev;
    preg->f_ops = f_ops; // 然后将传进来的 snd_pcm_f_ops[cidx] 或 &snd_ctl_f_ops 赋值给 preg
    preg->private_data = private_data;
    preg->card_ptr = card;
    snd_minors[minor] = preg; // 然后将 preg 赋值给 snd_minors[minor]
```

```
preg->dev = device_create(sound_class, device, MKDEV(major, minor), private_data, "%s",
name); // 为了让 /dev/snd 下面有声卡的设备节点, 我们要创建 sound_class, 和
device_create 但是 sound_class 这个类不是在 sound.c 里面创建的, 而是在 sound_core.c 里
面创建的然后用 sound.c 里面的 register_chrdev(major, "alsa", &snd_fops); 注册一个 alsa 字
符设备这些前面 sound.c 没有讲, 现在把他讲完。
```

/dev/snd 设备节点名字是什么

-----在 control 部分是-----

这个名字就是前面 control.c 里面的 `snd_ctl_dev_register` 函数里面实现的

```
sprintf(name, "controlC%i", cardnum);
```

然后用 `snd_register_device(SNDDRV_DEVICE_TYPE_CONTROL, card, -1, &snd_ctl_f_ops, card, name)` 注册进设备

-----在 PCM 部分是-----

这个名字是前面 pcm.c 里面的 `snd_pcm_dev_register` 函数里面实现的

case SNDDRV\_PCM\_STREAM\_PLAYBACK: 如果是播放

```
sprintf(str, "pcmC%id%ip", pcm->card->number, pcm->device);
```

case SNDDRV\_PCM\_STREAM\_CAPTURE: 如果是录音

```
sprintf(str, "pcmC%id%ic", pcm->card->number, pcm->device);
```

到此为止声卡就已经创建完毕了, 你将会在 /dev/snd 设备节点下看到

## pcmC0D0C, pcmC0D0p, controlC0 这些设备

以上只是让 linux 有这些设备节点，具体调用方法，看下面

以前注册声卡必须将上面的函数进行手工调用

1.snd\_card\_create

2.snd\_pcm\_new 进行一系列初始化

3.snd\_card\_register

这样才能创建声卡。

但是现在我们有 ASOC 框架，只需要 machine 文件进行注册，就可以创建声卡驱动，我以 S3C24XX 为例，在 linux3.4 内核上实现

S3C24XX\_uda134x.c

soc-core.c

在 machine 文件里面

```
static struct platform_driver s3c24xx_uda134x_driver = {
    .probe    = s3c24xx_uda134x_probe,
    .remove   = s3c24xx_uda134x_remove,
    .driver = {
        .name = "s3c24xx_uda134x",
        .owner = THIS_MODULE,
    },
};
```

```
static struct snd_soc_card snd_soc_s3c24xx_uda134x = {
    .dai_link = &s3c24xx_uda134x_dai_link,
```

就会导致 s3c24xx\_uda134x\_probe 函数被调用

```
static int s3c24xx_uda134x_probe(struct platform_device *pdev)
{
    s3c24xx_uda134x_snd_device =
    platform_device_alloc("soc-audio", -1);

    platform_set_drvdata(s3c24xx_uda134x_snd_device,
    &snd_soc_s3c24xx_uda134x); ←
    platform_device_add_data(s3c24xx_uda134x_snd_device,
    &s3c24xx_uda134x, sizeof(s3c24xx_uda134x));
    platform_device_add(s3c24xx_uda134x_snd_device); 添加
    这个设备后会导致 soc-core.c 里面的函数被调用
}
```

```
static struct platform_driver soc_driver = {
```

```
    .driver = {
        .name     = "soc-audio",
        .owner    = THIS_MODULE,
        .pm       = &snd_soc_pm_ops,
    },
    .probe    = soc_probe,
    .remove   = soc_remove,
};
```

```
static int soc_probe(struct platform_device *pdev)
{
    struct snd_soc_card *card = platform_get_drvdata
    (pdev) 这个 card 就是 machine 里面的
    return snd_soc_register_card(card); //这个函数将
    导致 snd_card_create,snd_card_register 被调用
    所以 ASOC 框架就自动帮你完成了声卡的创建
    然后你就可以看到 /dev/snd 下面的设备节点了
```

```
static struct snd_soc_dai_link s3c24xx_uda134x_dai_link =
{.name = "UDA134X",
.stream_name = "UDA134X",
.codec_name = "uda134x-codec", //用哪一个 codec 芯片
也就是 uda1341 芯片
.codec_dai_name = "uda134x-hifi", //codec 芯片里面的哪
一个接口 也就是 uda134x 芯片里面的 I2S 总线
.cpu_dai_name = "s3c24xx-iis", //s3c2440 的 I2S 总线
.ops = &s3c24xx_uda134x_ops,
.platform_name = "s3c24xx-DMA", //这个应该是 DMA
```

## Codec 芯片和 CPU 的 I2S 总线创建

Uda134x.c

```
static struct snd_soc_codec_driver  
soc_codec_dev_uda134x = {  
.probe = uda134x_soc_probe,  
.remove = uda134x_soc_remove,  
.suspend = uda134x_soc_suspend,  
.resume = uda134x_soc_resume,  
.....  
};  
static struct snd_soc_dai_driver uda134x_dai  
= {  
.playback = {  
.....  
}  
.capture = {  
.....  
}  
.ops = &uda134x_dai_ops,  
};
```

```
static struct platform_driver  
uda134x_codec_driver = {  
.driver = {  
.name = "uda134x-codec",  
},  
.probe = uda134x_codec_probe,  
.remove = uda134x_codec_remove,  
};  
static int uda134x_codec_probe(struct  
platform_device *pdev)  
{  
    return  
    snd_soc_register_codec(&pdev->dev,&soc_codec_dev_uda134x, &uda134x_dai, 1);  
}
```

s3c24xx-iis.c

```
static struct snd_soc_dai_driver s3c24xx_i2s_dai = {  
.probe = s3c24xx_i2s_probe,  
.suspend = s3c24xx_i2s_suspend,  
.resume = s3c24xx_i2s_resume,  
.playback = {  
.....  
},  
.capture = {  
.....  
}  
.ops = &s3c24xx_i2s_dai_ops,  
};  
static const struct snd_soc_dai_ops s3c24xx_i2s_dai_ops  
= {  
.trigger = s3c24xx_i2s_trigger,  
.hw_params = s3c24xx_i2s_hw_params,  
.set_fmt = s3c24xx_i2s_set_fmt, //设置格式  
.set_clkdiv = s3c24xx_i2s_set_clkdiv, //设置分频  
.set_sysclk = s3c24xx_i2s_set_sysclk, //设置系统 I2S 时钟  
};
```

```
static struct platform_driver s3c24xx_iis_driver  
= {  
.probe = s3c24xx_iis_dev_probe,  
.remove = s3c24xx_iis_dev_remove,  
.driver = {  
.name = "s3c24xx-iis",  
},  
static int s3c24xx_iis_dev_probe(struct  
platform_device *pdev){  
    snd_soc_register_component(&pdev->dev,  
&s3c24xx_i2s_component,&s3c24xx_i2s_dai,  
1); //注册 dai, 和 linux3.4 内核不同  
    snd_soc_register_dai(dev, dai_drv); //这个函数被  
    snd_soc_register_component 封装了一层, 结  
    果是一样的  
}
```

```

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)
{
    struct snd_soc_codec *codec;
    codec->driver = codec_drv;//将传入进来的
    codec_drv 放入 codec->driver 也就相当于把
    这个结构体&soc_codec_dev_uda134x 放入
    codec->driver

    list_add(&codec->list, &codec_list);//将 codec
    结构体放入了所谓的 codec_list

    ret = __snd_soc_register_component(dev,
    &codec->component,
    &codec_drv->component_driver, dai_drv,
    num_dai, false); //这个函数把
    ret = snd_soc_register_dais(dev, dai_drv,
    num_dai);这个封装了一遍，注意这个是 dais

}

snd_soc_register_dais(dev, dai_drv, num_dai){
    list_add(&dai->list, &dai_list);
    //把&uda134x_dai 这个结构体放入了
    dai_list, 这里和 CPU 里面 I2S 的 dai_list 是
    一样的
}

```

```

int snd_soc_register_component(struct device
*dev, const struct snd_soc_component_driver
*cmpnt_drv, struct snd_soc_dai_driver
*dai_drv, int num_dai)
{
    Return __snd_soc_register_component(dev,
    cmpnt, cmpnt_drv,dai_drv, num_dai, true);
}

```

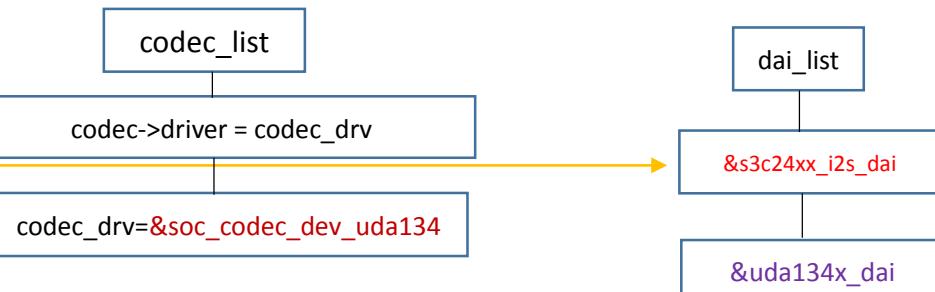
```

__snd_soc_register_component(struct device
*dev, struct snd_soc_component *cmpnt,
const struct snd_soc_component_driver
*cmpnt_drv,struct snd_soc_dai_driver
*dai_drv, int num_dai, bool allow_single_dai)
{
    ret = snd_soc_register_dai(dev, dai_drv);
}

static int snd_soc_register_dai(struct device
*dev,struct snd_soc_dai_driver *dai_drv)
{
    list_add(&dai->list, &dai_list); // 执行
    list_add(&dai->list, &dai_list); 将
    &s3c24xx_i2s_dai 这个结构体放入 dai_list 里
    面
}

```

根据上面的一系列操作就会产生几个链表； dai\_list 会链接 &s3c24xx\_i2s\_dai 和 &uda134x\_dai, codec\_list 会链接 codec->driver = codec\_drv



## PCM 码流也就是 DMA 创建

```
static struct snd_pcm_ops dma_ops = {
    .open      = dma_open,
    .close     = dma_close,
    .ioctl     = snd_pcm_lib_ioctl,
    .hw_params = dma_hw_params,
    .hw_free   = dma_hw_free,
    .prepare   = dma_prepare,
    .trigger   = dma_trigger,
    .pointer   = dma_pointer,
    .mmap      = dma_mmap,
};

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

```
int samsung_asoc_dma_platform_register(struct device *dev){
    return snd_soc_register_platform(dev, &samsung_asoc_platform);
}

int snd_soc_register_platform(struct device *dev, const struct snd_soc_platform_driver *platform_drv)
{
    ret = snd_soc_add_platform(dev, platform, platform_drv);
}

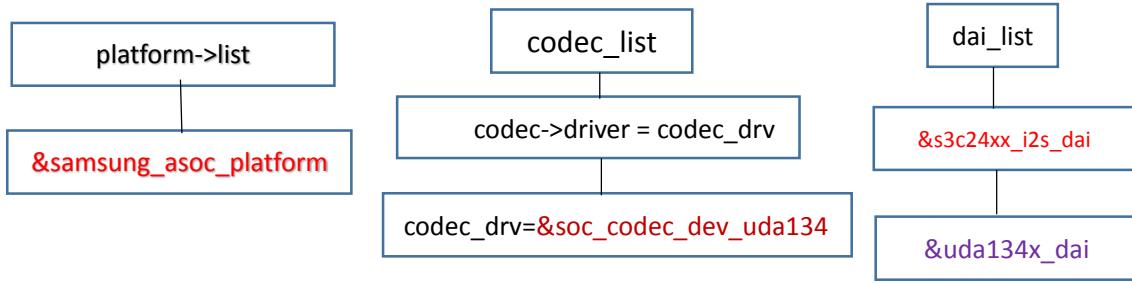
int snd_soc_add_platform(struct device *dev, struct snd_soc_platform *platform,
const struct snd_soc_platform_driver *platform_drv)
{
    list_add(&platform->list, &platform_list); // 将 samsung_asoc_platform 这个结构体
    放入 platform->list 链表里面
}
```

这样函数执行完成之后，会创建一个 platform\_list 链表来存放

&samsung\_asoc\_platform 结构体



现在我们已经把链表创建好了，分为 `platform->list`, `codec_list` 和 `dai_list`



然后这三个 `list` 如何关联上的呢？，就是通过 `machine` 文件来关联的  
上面已经将了如何用 ASOC 来创建声卡，这里还要讲一边，因为这里不仅是创建声卡，还要关联其他的 I2S, DMA。Codec 设备

```
static struct snd_soc_dai_link s3c24xx_uda134x_dai_link = {
    .name = "UDA134X",
    .stream_name = "UDA134X",
    .codec_name = "uda134x-codec",
    .codec_dai_name = "uda134x-hifi",
    .cpu_dai_name = "s3c24xx-iis",
    .ops = &s3c24xx_uda134x_ops,
    .platform_name = "s3c24xx-iis",
};
```

其实关联其它设备靠的就是这个 `dai_link`

进入 `machine` 文件 `S3C24XX-UDA134X.c`

```
s3c24xx_uda134x_snd_device = platform_device_alloc("soc-audio", -1);
platform_set_drvdata(s3c24xx_uda134x_snd_device, &snd_soc_s3c24xx_uda134x);
&snd_soc_s3c24xx_uda134x 里面有个 s3c24xx_uda134x_dai_link
platform_device_add(s3c24xx_uda134x_snd_device);
```

执行完该 `platform_device_add` 之后程序就会进入 `soc-core.c` 文件

因为 `soc-core.c` 文件里面有同名的驱动名称"soc-audio"

```
static struct platform_driver soc_driver = {
    .driver      = {
        .name      = "soc-audio",
        .owner      = THIS_MODULE,
        .pm        = &snd_soc_pm_ops,
    },
    .probe      = soc_probe,
    .remove     = soc_remove,
};
```

然后我们进入 soc-core.c 文件的 probe 函数去看 dai 关联过程

```
static int soc_probe(struct platform_device *pdev){  
    struct snd_soc_card *card = platform_get_drvdata(pdev);  
    return snd_soc_register_card(card); //这个 card 就是  
    machine 文件里面的私有数据  
    &snd_soc_s3c24xx_uda134x  
}
```

在 machine 文件里面有  
platform\_set\_drvdata(s3c  
24xx\_uda134x\_snd\_devic  
e,&snd\_soc\_s3c24xx\_uda  
134);

```
int snd_soc_register_card(struct snd_soc_card *card)  
{  
    card->rtd = devm_kzalloc(card->dev,...);  
    for (i = 0; i < card->num_links; i++)  
        card->rtd[i].dai_link = &card->dai_link[i]; //这个 dai_link  
    ret = snd_soc_instantiate_card(card); //实例化声卡  
}
```

在 soc-core.c 文件里面有  
platform\_get\_drvdata(pdev);

证明了这两个文件是相辅相成的

```
static int snd_soc_instantiate_card(struct snd_soc_card  
*card)  
/* bind DAIs */ 绑定 dai  
    for (i = 0; i < card->num_links; i++) {  
        ret = soc_bind_dai_link(card, i);  
    }  
    ret = snd_soc_init_codec_cache(codec); //初始化 codec 芯片  
    里面的默认值  
    ret = snd_card_create(SNDDRV_DEFAULT_IDX1,  
    SNDDRV_DEFAULT_STR1, card->owner, 0, &card->snd_card);  
    ret = snd_card_register(card->snd_card);  
但是奇怪的是为什么没有 snd_pcm_new 呢?  
其实是在 soc_probe_link_dais 函数里面的 ret = soc_new_pcm(rtd,  
num);
```

创建声卡三步骤  
Snd\_card\_create  
Snd\_pcm\_new  
Snd\_card\_register  
在 ASOC 里面一次帮你完成

```
if (rtd->dai_link->dynamic) {.....} else {  
    rtd->ops.open      = soc_pcm_open;  
    rtd->ops.hw_params = soc_pcm_hw_params;  
    rtd->ops.prepare   = soc_pcm_prepare;  
    rtd->ops.trigger   = soc_pcm_trigger;  
    rtd->ops.hw_free   = soc_pcm_hw_free;  
    rtd->ops.close     = soc_pcm_close;  
    rtd->ops.pointer   = soc_pcm_pointer;  
    rtd->ops.ioctl     = soc_pcm_ioctl;
```

```
snd_soc_dai_link s3c24xx_uda134x_dai_link = {  
    .name = "UDA134X",  
    .stream_name = "UDA134X",  
    .codec_name = "uda134x-codec",  
    .codec_dai_name = "uda134x-hifi",  
    .cpu_dai_name = "s3c24xx-iis",  
    .ops = &s3c24xx_uda134x_ops,  
    .platform_name = "s3c24xx-iis",  
};
```

```

static int soc_bind_dai_link(struct snd_soc_card *card, int num)
{
    list_for_each_entry(cpu_dai, &dai_list, list) { //查找 CPU_dai
        .....
        if (dai_link->cpu_dai_name &&strcmp(cpu_dai->name,
                                                dai_link->cpu_dai_name)) //根据 machine 文件 cpu_dai 的名字在
        dai_list 链表里面找到 &s3c24xx_i2s_dai 这个结构体
        continue;
        rtd->cpu_dai = cpu_dai; 找出来之后将 &s3c24xx_i2s_dai 结构体
        赋值给 rtd 里面的 cpu_dai
    }
    list_for_each_entry(codec, &codec_list, list) { //查找 codec
        if (dai_link->codec_of_node) {
            if (codec->dev->of_node != dai_link->codec_of_node)
            continue;} else {
            if (strcmp(codec->name, dai_link->codec_name)) //根据 machine 文件 codec_name 的名字在 codec_list 里面找到
            &soc_codec_dev_uda134 这个结构体
            continue;
        }
        rtd->codec = codec; //codec 里面的
        codec->driver=&soc_codec_dev_uda134x
    }

    list_for_each_entry(platform, &platform_list, list) { //查 platform
    也就是 dma
        if (dai_link->platform_of_node) {
            if(platform->dev->of_node !=dai_link->platform_of_node)
            continue;
        } else {
            if (strcmp(platform->name, platform_name)) continue;}
            根据 machine 文件 platform_name 名字在 platform->list 里面找
            到 &samsung_asoc_platform 结构体
            rtd->platform = platform; //将 samsung_asoc_platform 放入
            rtd->platform }
    }

    查找 codec_dai 和上面一样也是根据 strcmp 来查找的
}

```

```

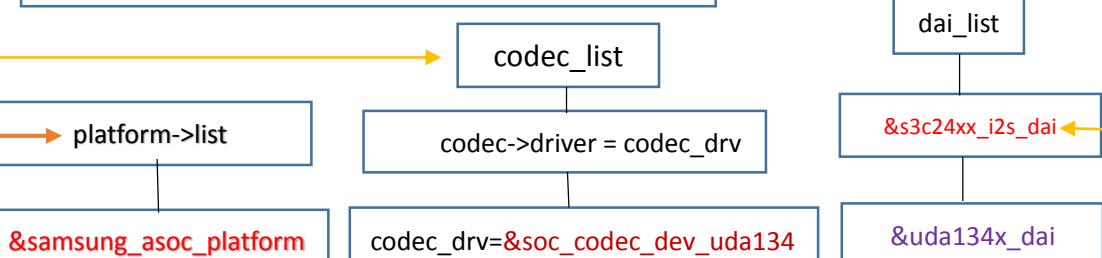
static struct snd_soc_codec_driver
soc_codec_dev_uda134x = {
    .probe = uda134x_soc_probe,
    .remove = uda134x_soc_remove,
    .suspend = uda134x_soc_suspend,
    .resume = uda134x_soc_resume,
    .....
};

```

```

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

```



# alsa 应用层调用过程

open /dev/snd/controlC0

应用程序打开 controlC0 这个设备节点时，一定会去找驱动层与 controlC0 对应的 file\_operations 结构体

该结构体在 control.c 文件里面，进入 snd\_ctl\_ioctl 函数

Control.c 文件

```
static const struct file_operations snd_ctl_f_ops = {  
    .owner = THIS_MODULE,  
    .read =      snd_ctl_read,  
    .open =      snd_ctl_open, //对应 应用层  
    的 open 函数  
    .release = snd_ctl_release,  
    .llseek = no_llseek,  
    .poll =      snd_ctl_poll,  
    .unlocked_ioctl = snd_ctl_ioctl, //对应 应  
    用层的 ioctl 函数  
    .compat_ioctl = snd_ctl_ioctl_compat,  
    .fasync = snd_ctl_fasync,  
};
```

```
static long snd_ctl_ioctl(struct file *file, unsigned  
int cmd, unsigned long arg)
```

```
{  
    应用层 ioctl(fd, SNDDRV_CTL_IOCTL_PVERSION)  
    case SNDDRV_CTL_IOCTL_PVERSION:  
        return put_user(SNDDRV_CTL_VERSION, ip) ? -  
   EFAULT : 0; //返回给应用层声卡版本号不涉及硬  
   件操作  
    应用层 ioctl(fd, SNDDRV_CTL_IOCTL_CARD_INFO)  
    case SNDDRV_CTL_IOCTL_CARD_INFO:  
        return snd_ctl_card_info(card, ctl, cmd, argp); //没  
   什么用返回一些信息给用户空间
```

应用层

```
ioctl(fd, SNDDRV_CTL_IOCTL_PCM_PREFER_SUBDEVICE)  
这个参数在 control.c 里面的 snd_ctl_ioctl 没有，但是程  
序会在 pcm.c 中找到该参数，怎么进入 pcm.c 的呢
```

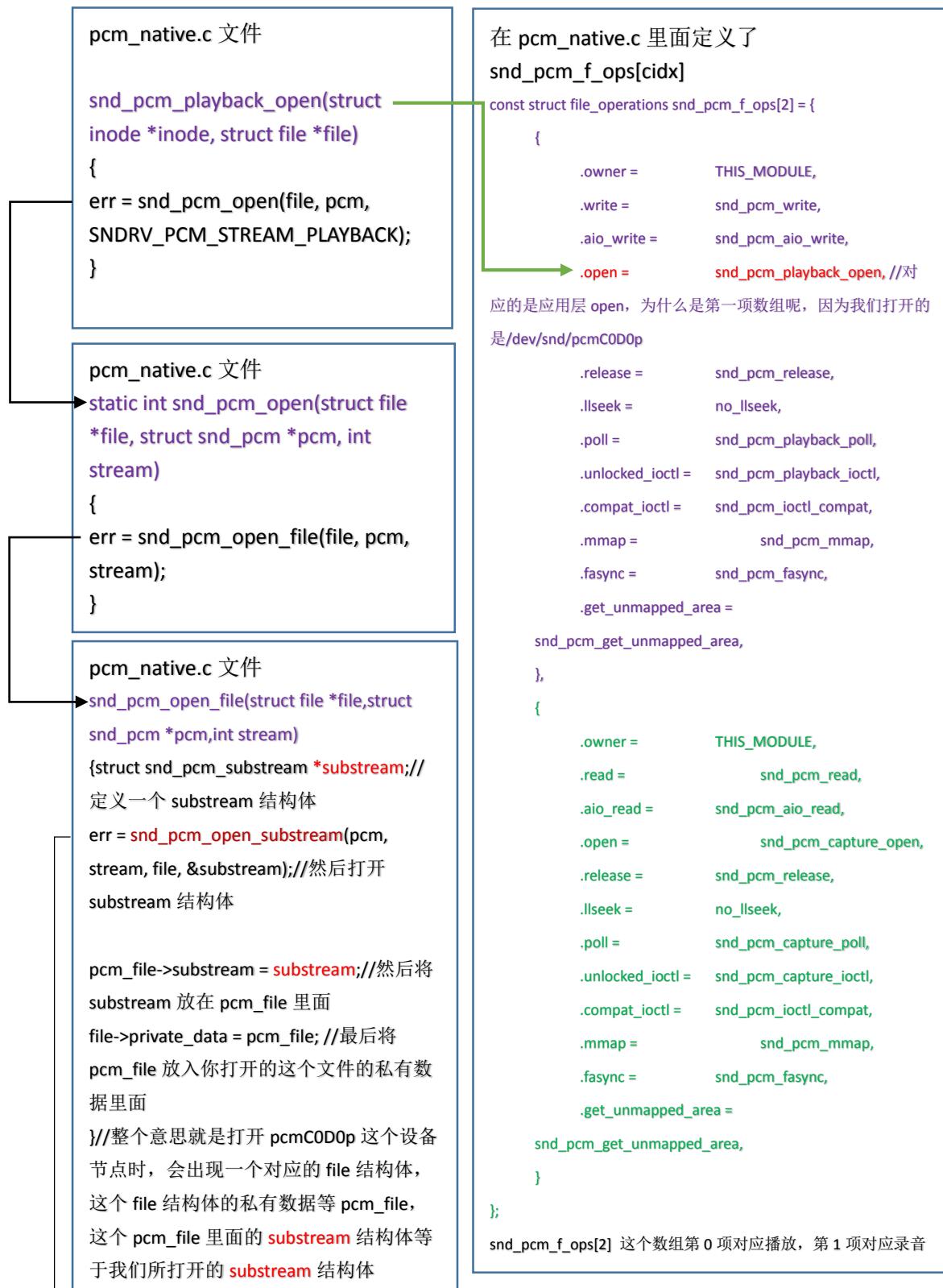
```
}  
ioctl(fd, SNDDRV_CTL_IOCTL_PCM_PREFER_SUBDEVICE);  
如果 ioctl 传进来的参数都不是 snd_ctl_ioctl 里  
面的参数的话就会执行  
list_for_each_entry(p, &snd_ctl_ioctls, list)  
{  
    //从这个&snd_ctl_ioctls 链表里面取出某一个  
    //结构体给 p  
    err = p->iocctl(card, ctl, cmd, arg); //调用 p 指向结  
    构体的 iocctl 函数  
    if (err != -ENOIOCTLCMD) {  
        up_read(&snd_ioctl_rwsem);  
        return err; }  
}
```

&snd\_ctl\_ioctls 这个结构体里面有 PCM 注册的  
iocctl，所以就会自动从 control.c 文件跳到 pcm.c 里面去  
寻找 SNDDRV\_CTL\_IOCTL\_PCM\_PREFER\_SUBDEVICE 这个  
参数所以最终会导致 pcm.c 里面 snd\_pcm\_control\_ioctl  
函数被调用。

从分析来开 controlC0 设备节点并没有做  
什么实质性的东西，硬件也没有操作，  
所以还是看下面的其他内容。

open /dev/snd/pcmC0D0p 播放节点

打开 pcmC0D0p 节点进入 pcm\_native.c



### pcm\_native.c 文件

```
int snd_pcm_open_substream(struct snd_pcm *pcm, int stream, struct file *file, struct  
snd_pcm_substream **rsubstream)  
{  
    err = snd_pcm_hw_constraints_init(substream); //硬件约束初始化  
    err = substream->ops->open(substream); //这个 substream->ops 结构体是 struct  
    snd_pcm_ops *ops;  
    这个 ops 和 soc_pcm_ops 的结构体是一样的，这里的 ops 就和操作硬件有关联了  
    所以 substream->ops = soc_new_pcm 函数里面的 soc_pcm_ops，这个 soc 打头的就是  
    操作硬件  
    substream->ops->open = soc_pcm_open  
}
```

### pcm.h 文件

```
struct snd_pcm_ops *ops;
```

在前面 soc-core.c 文件里面

snd\_soc\_instantiate\_card 函数里面有个  
soc\_probe\_link\_dais 函数，在  
soc\_probe\_link\_dais 函数里面有 soc\_new\_pcm  
函数，在 soc\_new\_pcm 里面也创建了 ops 结构体  
if (rtd->dai\_link->dynamic) {.....} else {  
 rtd->ops.open = soc\_pcm\_open;  
 rtd->ops.hw\_params = soc\_pcm\_hw\_params;  
 rtd->ops.prepare = soc\_pcm\_prepare;  
 rtd->ops.trigger = soc\_pcm\_trigger;  
 rtd->ops.hw\_free = soc\_pcm\_hw\_free;  
 rtd->ops.close = soc\_pcm\_close;  
 rtd->ops.pointer = soc\_pcm\_pointer;  
 rtd->ops.ioctl = soc\_pcm\_ioctl;  
}

rtd->ops 类型就是 snd\_pcm\_ops

substream->ops=snd\_pcm\_ops

所以 pcm\_native.c 文件里面 substream->ops 等于 soc-core.c 文件里面的 snd\_pcm\_ops

那么调用 substream->ops->open 就等于同时调用 soc\_pcm\_open

```
int soc_new_pcm(struct snd_soc_pcm_runtime  
*rtd, int num)  
{  
    snd_pcm_set_ops(pcm, SNDDRV_PCM_STREAM_  
PLAYBACK, &rtd->ops);  
}
```

```
void snd_pcm_set_ops(struct snd_pcm *pcm,  
int direction, const struct snd_pcm_ops *ops)  
{  
    substream->ops = ops;  
}
```

} 这个 rtd->ops 就是传递给 substream 的 ops

↓

```
static int soc_pcm_open(struct snd_pcm_substream *substream) //依次调用
cpu_dai, dma, codec_dai, machine 文件的 open 或 startup 函数
{
    /* startup the audio subsystem */
    if (cpu_dai->driver->ops && cpu_dai->driver->ops->startup) 如果 s3c24xx-i2s.c
文件里面的 snd_soc_dai_driver s3c24xx_i2s_dai 里面的.ops 结构体有 startup 回调
函数的话，就调用 s3c24xx-i2s.c 里面的 startup 函数，如果没有 startup 函数，那
么就不调用 startup 函数

    if (platform->driver->ops && platform->driver->ops->open) 在 dma.c 文件里面
查找 snd_pcm_ops 结构体 dma_ops 里面有有没有 open 函数，有 open 就调用该
open

    if (codec_dai->driver->ops && codec_dai->driver->ops->startup) 在 uda134x.c 文
件里面查找 snd_soc_dai_ops uda134x_dai_ops 结构里面有没有 startup 函数，有
就调用

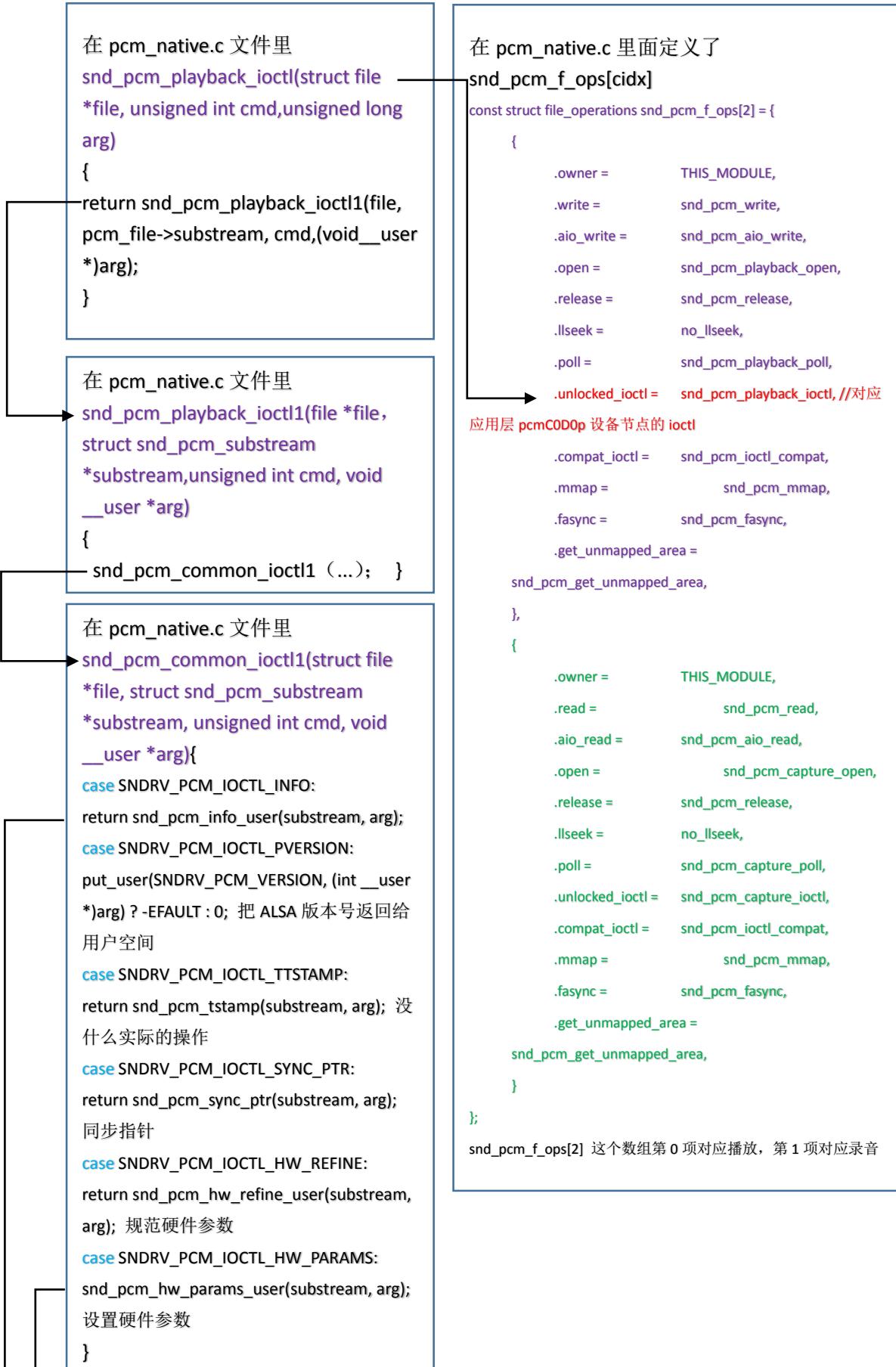
    if (rtd->dai_link->ops && rtd->dai_link->ops->startup) 在 s3c24xx-uda134x.c 文
件里面查找 snd_soc_ops s3c24xx_uda134x_ops 结构里面有没有 startup 函数，有
就调用

}
```

Open 之后就是 ioctl /dev/snd/pcmC0D0p 节点

以下 ioctl 的入口是 snd\_pcm\_f\_ops[2]这个结构体里面的 snd\_pcm\_playback\_ioctl,

```
fd=Open (/dev/snd/pcmC0D0p)
ioctl(fd, SNDDRV_PCM_IOCTL_INFO)
ioctl(fd, SNDDRV_PCM_IOCTL_PVERSION)
ioctl(fd, SNDDRV_PCM_IOCTL_TTSTAMP)
ioctl(fd, SNDDRV_PCM_IOCTL_SYNC_PTR)
ioctl(fd, SNDDRV_PCM_IOCTL_HW_REFINE)
ioctl(fd, SNDDRV_PCM_IOCTL_HW_PARAMS)
ioctl(fd, SNDDRV_PCM_IOCTL_SYNC_PTR)//同步指针
ioctl(fd, SNDDRV_PCM_IOCTL_SW_PARAMS)//软件参数 不涉及硬件操作 暂时不分
析
ioctl(fd, SNDDRV_PCM_IOCTL_SYNC_PTR)
ioctl(fd, SNDDRV_PCM_IOCTL_PREPARE)//电源管理相关
ioctl(fd, SNDDRV_PCM_IOCTL_SYNC_PTR)
ioctl(fd, SNDDRV_PCM_IOCTL_SW_PARAMS) //软件参数不涉及硬件操作
```



在 pcm\_native.c 文件里  
→ snd\_pcm\_info\_use(...)  
{  
err = snd\_pcm\_info(substream, info);  
}

在 pcm\_native.c 文件里  
int snd\_pcm\_info(snd\_pcm\_substream  
\*substream, struct snd\_pcm\_info  
\*info)  
{  
substream->ops->iocctl(substream,  
SNDRV\_PCM\_IOCTL1\_INFO, info);  
在 dma.c 里面 snd\_pcm\_ops 的 iocctl  
函数中，直接执行了一个没什么用的 return ; }

在 pcm\_native.c 文件里  
→ snd\_pcm\_hw\_params\_user(struct  
snd\_pcm\_substream \*substream,  
struct snd\_pcm\_hw\_params \_\_user \*  
\_params)  
{  
snd\_pcm\_hw\_params(substream,para  
ms);  
}

pcm.h 文件  
→ Substream->ops 是这个结构体 struct  
snd\_pcm\_ops \*ops;

在 pcm\_native.c 文件里  
static int snd\_pcm\_hw\_params(struct  
snd\_pcm\_substream \*substream,  
struct snd\_pcm\_hw\_params \*params)  
{  
if (substream->ops->hw\_params != NULL)  
{  
substream->ops->hw\_params(substream,  
params);如果 ops 结构体里面有  
hw\_params  
函数的话，就去设置硬件参数  
}  
}

在前面 soc-core.c 文件里面有没  
hw\_params 回调函数呢  
rtd->ops.hw\_params =soc\_pcm\_hw\_params;  
答案是有的，所以依次调用 mahcine,  
cpu\_dai, codec\_dai, platform(dma)文件里面的  
hw\_params 函数，当然前提是你的这四个  
文件里面申请了 hw\_params 回调函数。  
在 hw\_params 函数里面就是设置硬件的代码  
了

```

ioctl(fd, SNDRV_PCM_IOCTL_SYNC_PTR)//同步指针
ioctl(fd, SNDRV_PCM_IOCTL_SW_PARAMS)//软件参数 不涉及硬件操作 暂时不分析
 ioctl(fd, SNDRV_PCM_IOCTL_SYNC_PTR)
ioctl(fd, SNDRV_PCM_IOCTL_PREPARE)//电源管理相关
 ioctl(fd, SNDRV_PCM_IOCTL_SYNC_PTR)
ioctl(fd, SNDRV_PCM_IOCTL_SW_PARAMS) //软件参数不涉及硬件操作

```

for 循环

```

{
    ioctl(fd, SNDRV_PCM_IOCTL_WRITEI_FRAMES) 这个才是真正的向 DMA 音频写数据
    ioctl(fd, SNDRV_PCM_IOCTL_SYNC_PTR)//同步指针
}

```

在 pcm\_native.c 文件里

```

static int snd_pcm_common_ioctl1 (struct file *file, struct snd_pcm_substream
*substream,unsigned int cmd, void __user
*arg)
{
    .....
    .....
    case SNDRV_PCM_IOCTL_PREPARE:
        return snd_pcm_prepare(substream, file);
    }

```

在 pcm\_native.c 文件里

```

static int snd_pcm_prepare(struct
snd_pcm_substream *substream,struct file
*file)
{
    if ((res = snd_power_wait(card,
SNDRV_CTL_POWER_D0)) >= 0) //上电 做些初始化操作
    }

```

在 pcm\_native.c 文件里

```

static int snd_pcm_playback_ioctl1(struct
file *file,struct snd_pcm_substream,.....)
{
    case SNDRV_PCM_IOCTL_WRITEI_FRAMES:
        if (copy_from_user(&xferi,_xferi,
sizeof(xferi))) 从用户空间读取音频数据
        result = snd_pcm_lib_write(substream,
_xferi.buf, xferi.frames);然后写数据到 dma
    }

```

在 sound/core/init.c 文件里面

```

int snd_power_wait(struct snd_card *card, unsigned int
power_state)
{ b 其实就是电源管理相关的，以后再分析 }

```

在 sound/core/Pcm.Lib.c 文件里面

```

snd_pcm_sframes_t snd_pcm_lib_write(struct
snd_pcm_substream *substream, .....)
{
    return snd_pcm_lib_write1(substream, (unsigned
long)buf, size, nonblock,snd_pcm_lib_write_transfer);
}
static snd_pcm_sframes_t snd_pcm_lib_write1(struct
snd_pcm_substream *substream,.....)
{
    err = snd_pcm_start(substream);//启动 DMA 传输
}

```

## amixer 分析

amixer cset numid=1 30 设置音量

先 open /dev/snd/controlC0

```
ioctl(fd, SNDDRV_CTL_IOCTL_CARD_INFO)
ioctl(fd, SNDDRV_CTL_IOCTL_PVERSION)
ioctl(fd, SNDDRV_CTL_IOCTL_ELEM_INFO)//把这个元素信息复制回用户空间
ioctl(fd, SNDDRV_CTL_IOCTL_ELEM_READ)//读这个元素
ioctl(fd, SNDDRV_CTL_IOCTL_ELEM_WRITE)//读出来之后写这个元素
```

然后关闭设备节点

在 sound/core/control.c 文件里面

```
static long snd_ctl_ioctl(struct file *file, unsigned int
cmd, unsigned long arg)
{
    case SNDDRV_CTL_IOCTL_ELEM_INFO://把当前值复
制回用户空间
        return snd_ctl_elem_info_user(ctl, argp);
    case SNDDRV_CTL_IOCTL_ELEM_WRITE:
        return snd_ctl_elem_write_user(ctl, argp); 这
个 write 应该就是写硬件了
}
```

```
static int snd_ctl_elem_write_user(struct snd_ctl_file
*file, struct snd_ctl_elem_value __user *_control)
{
    control = memdup_user(_control, sizeof(*control)); //从用户空间拷贝过来
    result = snd_ctl_elem_write(card, file, control);
}
```

```
static int snd_ctl_elem_write(struct snd_card *card,
struct snd_ctl_file *file, struct snd_ctl_elem_value
*control)
{
    result = kctl->put(kctl, control);这里的 ioctl 和驱动程
序里面某一个 kcontrol 回调函数是对应上的, info
get put
}
```