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

/Sound/core/Sonud.c

sound\core\pcm.c

sound\core\init.c

/Sound/core/Core.h

/Sound/core/control.c

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**

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 🡪 SNDRV\_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 🡪 SNDRV\_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\_card\_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

}

1

在sound\core\里面有个control.c

static int snd\_ctl\_dev\_register(struct snd\_device \*device)

{

snd\_register\_device(SNDRV\_DEVICE\_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)

{

tatic 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, SNDRV\_DEV\_CONTROL, card, &ops);//一个声卡里面可能有很多个逻辑设备，NDRV\_DEV\_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

};

4

3

2

在include\ound\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/Sonud.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

1

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, SNDRV\_DEV\_PCM, pcm, &ops)；创建一个声卡的逻辑设snd\_device\_new(card, SNDRV\_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

}

}  
}

3

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(SNDRV\_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 SNDRV\_PCM\_STREAM\_PLAYBACK: 如果是播放

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

case SNDRV\_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

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

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 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 s3c24xx-iis.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 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

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);

}

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

codec\_list

dai\_list

codec->driver = codec\_drv

&s3c24xx\_i2s\_dai

codec\_drv=&soc\_codec\_dev\_uda134x

&uda134x\_dai

**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

platform->list

&samsung\_asoc\_platform

现在我们已经把链表创建好了，分为platform->list，codec\_list和dai\_list

dai\_list

codec\_list

codec->driver = codec\_drv

codec\_drv=&soc\_codec\_dev\_uda134x

platform->list

&s3c24xx\_i2s\_dai

&samsung\_asoc\_platform

&uda134x\_dai

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

在machine文件里面有platform\_set\_drvdata(s3c24xx\_uda134x\_snd\_device,&snd\_soc\_s3c24xx\_uda134x);

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 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

}

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

}

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

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(SNDRV\_DEFAULT\_IDX1, SNDRV\_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);

S3c24xx\_uda134x.c文件

static struct snd\_soc\_card snd\_soc\_s3c24xx\_uda134x = {

.name ="S3C24XX\_UDA134X",

.dai\_link=&s3c24xx\_uda134x\_dai\_link,

.num\_links = 1,

};

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,

};

&s3c24xx\_i2s\_dai

dai\_list

&uda134x\_dai

codec\_list

codec\_drv=&soc\_codec\_dev\_uda134x

codec->driver = codec\_drv

platform->list

&samsung\_asoc\_platform

**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

static long snd\_ctl\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

应用层ioctl(fd，SNDRV\_CTL\_IOCTL\_PVERSION)

case SNDRV\_CTL\_IOCTL\_PVERSION:

return put\_user(SNDRV\_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，SNDRV\_CTL\_IOCTL\_CARD\_INFO)

case SNDRV\_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,SNDRV\_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 SNDRV\_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\_control\_ioctls, list)

{

//从这个&snd\_control\_ioctls链表里面取出某一个结构体给p

err = p->fioctl(card, ctl, cmd, arg);//调用p指向结构体的ioctl函数

if (err != -ENOIOCTLCMD) {

up\_read(&snd\_ioctl\_rwsem);

return err; }

}

&snd\_control\_ioctls这个结构体里面有PCM注册的ioctl，所以就会自动从control.c文件跳到pcm.c里面去寻找SNDRV\_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。

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,

};

pcm.c文件

static int snd\_pcm\_control\_ioctl(struct snd\_card \*card, struct snd\_ctl\_file \*control,unsigned int cmd, unsigned long arg)

{

case SNDRV\_CTL\_IOCTL\_PCM\_PREFER\_SUBDEVICE:

get\_user(val, (int \_\_user \*)arg) //貌似也没做什么事情

control->prefer\_pcm\_subdevice = val;//只是把用户空间传入进来的值记录在这里。

}

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

snd\_pcm\_playback\_open(struct inode \*inode, struct file \*file)

{

err = snd\_pcm\_open(file, pcm, SNDRV\_PCM\_STREAM\_PLAYBACK);

}

在pcm\_native.c里面定义了snd\_pcm\_f\_ops[cidx]

const struct file\_operations snd\_pcm\_f\_ops[2] = {

{

.owner = THIS\_MODULE,

.write = snd\_pcm\_write,

.aio\_write = snd\_pcm\_aio\_write,

.open = snd\_pcm\_playback\_open, //对应的是应用层open，为什么是第一项数组呢，因为我们打开的是/dev/snd/pcmC0D0p

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_playback\_poll,

.unlocked\_ioctl = snd\_pcm\_playback\_ioctl,

.compat\_ioctl = snd\_pcm\_ioctl\_compat,

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

.get\_unmapped\_area = snd\_pcm\_get\_unmapped\_area,

},

{

.owner = THIS\_MODULE,

.read = snd\_pcm\_read,

.aio\_read = snd\_pcm\_aio\_read,

.open = snd\_pcm\_capture\_open,

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_capture\_poll,

.unlocked\_ioctl = snd\_pcm\_capture\_ioctl,

.compat\_ioctl = snd\_pcm\_ioctl\_compat,

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

.get\_unmapped\_area = snd\_pcm\_get\_unmapped\_area,

}

};

snd\_pcm\_f\_ops[2] 这个数组第0项对应播放，第1项对应录音

pcm\_native.c文件

static int snd\_pcm\_open(struct file \*file, struct snd\_pcm \*pcm, int stream)

{

err = snd\_pcm\_open\_file(file, pcm, stream);

}

pcm\_native.c文件

snd\_pcm\_open\_file(struct file \*file,struct snd\_pcm \*pcm,int stream)

{struct snd\_pcm\_substream \*substream;//定义一个substream结构体

err = snd\_pcm\_open\_substream(pcm, stream, file, &substream);//然后打开substream结构体

pcm\_file->substream = substream;//然后将substream放在pcm\_file里面

file->private\_data = pcm\_file; //最后将pcm\_file放入你打开的这个文件的私有数据里面

}//整个意思就是打开pcmC0D0p这个设备节点时，会出现一个对应的file结构体，这个file结构体的私有数据等pcm\_file，这个pcm\_file里面的substream结构体等于我们所打开的substream结构体

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

}

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

pcm.h文件

struct snd\_pcm\_ops \*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,SNDRV\_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，SNDRV\_PCM\_IOCTL\_INFO)

Ioctl(fd，SNDRV\_PCM\_IOCTL\_PVERSION)

Ioctl(fd，SNDRV\_PCM\_IOCTL\_TTSTAMP)

Ioctl(fd，SNDRV\_PCM\_IOCTL\_SYNC\_PTR)

Ioctl(fd，SNDRV\_PCM\_IOCTL\_HW\_REFINE)

Ioctl(fd，SNDRV\_PCM\_IOCTL\_HW\_PARAMS)

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

在pcm\_native.c文件里

snd\_pcm\_playback\_ioctl(struct file \*file, unsigned int cmd,unsigned long arg)

{

return snd\_pcm\_playback\_ioctl1(file, pcm\_file->substream, cmd,(void\_\_user \*)arg);

}

在pcm\_native.c里面定义了snd\_pcm\_f\_ops[cidx]

const struct file\_operations snd\_pcm\_f\_ops[2] = {

{

.owner = THIS\_MODULE,

.write = snd\_pcm\_write,

.aio\_write = snd\_pcm\_aio\_write,

.open = snd\_pcm\_playback\_open,

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_playback\_poll,

.unlocked\_ioctl = snd\_pcm\_playback\_ioctl, //对应应用层pcmC0D0p设备节点的ioctl

.compat\_ioctl = snd\_pcm\_ioctl\_compat,

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

.get\_unmapped\_area = snd\_pcm\_get\_unmapped\_area,

},

{

.owner = THIS\_MODULE,

.read = snd\_pcm\_read,

.aio\_read = snd\_pcm\_aio\_read,

.open = snd\_pcm\_capture\_open,

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_capture\_poll,

.unlocked\_ioctl = snd\_pcm\_capture\_ioctl,

.compat\_ioctl = snd\_pcm\_ioctl\_compat,

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

.get\_unmapped\_area = snd\_pcm\_get\_unmapped\_area,

}

};

snd\_pcm\_f\_ops[2] 这个数组第0项对应播放，第1项对应录音

在pcm\_native.c文件里

snd\_pcm\_playback\_ioctl1(file \*file，struct snd\_pcm\_substream \*substream,unsigned int cmd, void \_\_user \*arg)

{

snd\_pcm\_common\_ioctl1（...）； }

在pcm\_native.c文件里

snd\_pcm\_common\_ioctl1(struct file \*file, struct snd\_pcm\_substream \*substream, unsigned int cmd, void \_\_user \*arg){

case SNDRV\_PCM\_IOCTL\_INFO:

return snd\_pcm\_info\_user(substream, arg);

case SNDRV\_PCM\_IOCTL\_PVERSION:

put\_user(SNDRV\_PCM\_VERSION, (int \_\_user \*)arg) ? -EFAULT : 0; 把ALSA版本号返回给用户空间

case SNDRV\_PCM\_IOCTL\_TTSTAMP:

return snd\_pcm\_tstamp(substream, arg); 没什么实际的操作

case SNDRV\_PCM\_IOCTL\_SYNC\_PTR:

return snd\_pcm\_sync\_ptr(substream, arg); 同步指针

case SNDRV\_PCM\_IOCTL\_HW\_REFINE:

return snd\_pcm\_hw\_refine\_user(substream, arg); 规范硬件参数

case SNDRV\_PCM\_IOCTL\_HW\_PARAMS:

snd\_pcm\_hw\_params\_user(substream, arg);

设置硬件参数

}

在pcm\_native.c文件里

int snd\_pcm\_info(snd\_pcm\_substream \*substream, struct snd\_pcm\_info \*info)

{

substream->ops->ioctl(substream, SNDRV\_PCM\_IOCTL1\_INFO, info);

在dma.c里面snd\_pcm\_ops的ioctl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\_info\_use(...)

{

err = snd\_pcm\_info(substream, info);

}

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,params);

}

pcm.h文件

Substream->ops是这个结构体struct snd\_pcm\_ops \*ops;

在前面soc-core.c文件里面有没有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

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

}

}

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/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，SNDRV\_CTL\_IOCTL\_CARD\_INFO)

Ioctl(fd，SNDRV\_CTL\_IOCTL\_PVERSION)

Ioctl(fd，SNDRV\_CTL\_IOCTL\_ELEM\_INFO)//把这个元素信息复制回用户空间

Ioctl(fd，SNDRV\_CTL\_IOCTL\_ELEM\_READ)//读这个元素

Ioctl(fd，SNDRV\_CTL\_IOCTL\_ELEM\_WRITE)//读出来之后写这个元素

然后关闭设备节点

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);

}

在sound/core/control.c文件里面

static long snd\_ctl\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

case SNDRV\_CTL\_IOCTL\_ELEM\_INFO://把当前值复制回用户空间

return snd\_ctl\_elem\_info\_user(ctl, argp);

case SNDRV\_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(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

}