Codec、Platform、Machine驱动的组成部分及其注册过程，这三者都是物理设备相关的，对音频物理链路有了一定的认知。

接着分析音频驱动的中间层，由于这些并不是真正的物理设备，故我们称之为逻辑设备。

PCM逻辑设备，我们又习惯称之为PCM中间层或pcm native，

起着承上启下的作用：

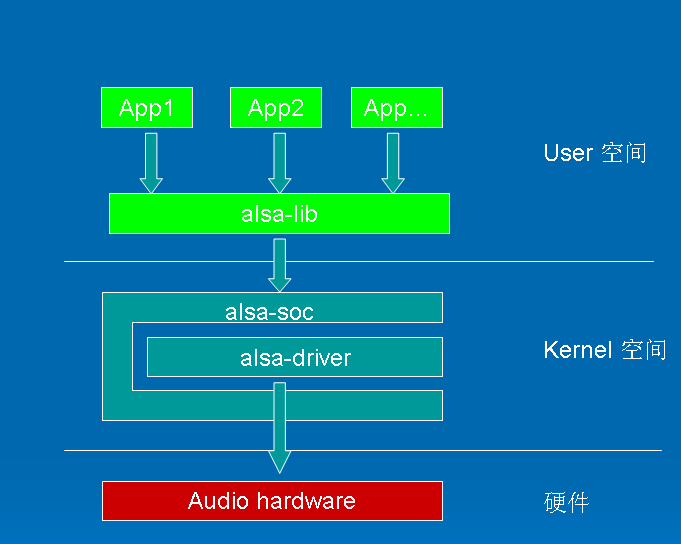
往上是与用户态接口的交互，实现音频数据在用户态和内核态之间的拷贝；

往下是触发codec、platform、machine的操作函数，实现音频数据在dma\_buffer<->

Platform和Codec的DAI通过snd\_soc\_dai\_link结构，在Machine驱动中进行绑定连接

cpu\_dai <-> codec之间的传输。

**ALSA架构：**

****

ALSA提供了alsa-driver，同时在应用层，ALSA为我们提供了alsa-lib，应用程序只要调用alsa-lib提供的API，即可以完成对底层音频硬件的控制

**/de/snd下的设备文件：**

* **controlC0 -->                 用于声卡的控制，例如通道选择，混音，麦克风的控制等**
* **midiC0D0  -->                用于播放midi音频**
* **pcmC0D0c --〉               用于录音的pcm设备**
* **pcmC0D0p --〉               用于播放的pcm设备**
* **seq  --〉                        音序器**
* **timer --〉                       定时器**

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

**声卡结构概述**

回顾下ASoC是如何注册声卡的

一： Machine驱动初始化时，name="soc-audio"的platform\_device与platform\_driver匹配成功，触发soc\_probe()调用；

二：继而调用snd\_soc\_register\_card()，该函数做的事情很多：

           1.  为每个音频物理链路找到对应的codec、codec\_dai、cpu\_dai、platform设备实例， 完成dai\_link的绑定；

           2.  调用snd\_card\_create()创建声卡；

           3.  依次回调cpu\_dai、codec、platform的probe()函数，完成物理设备的初始化；

三： 随后调用soc\_new\_pcm()创建pcm逻辑设备：

           1.  设置pcm native中要使用的pcm操作函数，这些函数用于操作音频物理设备， 包括machine、codec\_dai、cpu\_dai、platform；

           2.  调用snd\_pcm\_new()创建pcm设备，回放子流实例和录制子流实例都在这里创 建；

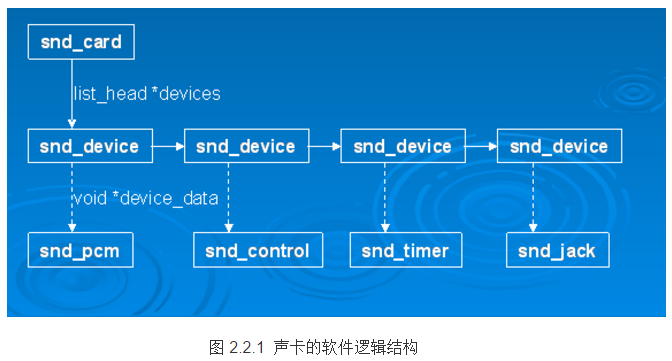
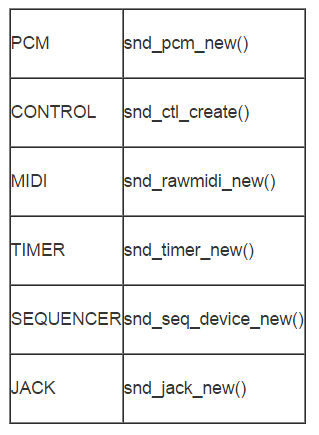
           3.  回调platform驱动的pcm\_new()，完成音频dma设备初始化和dma buffer内存 分配；

四：最后调用snd\_card\_register()注册声卡。

1. 创建声卡的sysfs设备；
2. 调用snd\_device\_register\_all()注册所有挂在该声卡下的逻辑设备；
3. 建立proc信息文件和sysfs属性文件

声卡驱动上挂着多个逻辑子设备，有pcm（音频数据流）、control（混音器控制）、midi（迷笛）、timer（定时器）、sequencer（音序器）等。

每个逻辑设备创建时，都会调用snd\_device\_new()生成一个snd\_device实例，并把该实例挂到声卡snd\_card的devices链表上



**ALSA声卡架构执行流程：**

Advanced Linux Sound Architecture Driver Version 1.0.21. ----->sound.c

machine:..device\_add...

soc-core:..snd\_soc\_init... ----->soc-core.c

Uda134X:..uda134x\_init... ----->uad134x.c

snd\_soc\_register\_dai...注册了声卡的数字音频接口

No device for DAI UDA134X

INIT\_LIST\_HEAD(&dai->list);//初始化链表头dai->list

snd\_soc\_instantiate\_cards...

list\_for\_each\_entry(card（链表所在结构体）, &card\_list（链表头）, list（遍历的成员）)

snd\_soc\_instantiate\_card(card);//遍历card\_list链表，对于每个list所在的card结构体执行snd\_soc\_instantiate\_card操作。

list\_for\_each\_entry(platform, &platform\_list, list)

if (card->platform == platform) {

found = 1;

break;

}//遍历platform\_list链表，在snd\_soc\_register\_platform 中注册该链表，如果在list中找到与card->platform匹配的则标记为1

list\_for\_each\_entry(dai, &dai\_list, list)

if (card->dai\_link[i].cpu\_dai == dai) {

found = 1;

break;

}//同理，遍历dai\_list,在snd\_soc\_register\_dai中注册

//以上两个遍历链表，在失败后返回，在没有platform等初始化的情况下会遍历失败返回，

之后在其他模块的初始化中，也遍历以上链表知道找到匹配向下执行。

s3c24xx\_pcm:..s3c24xx\_soc\_platform\_init...

snd\_soc\_register\_platform...

INIT\_LIST\_HEAD(&platform->list);//初始化链表头platform->list

list\_add(&platform->list, &platform\_list);//将platform结构体下的list连接添加到platform\_list中

snd\_soc\_instantiate\_cards...//遍历card\_list

s3c24xx\_i2s:..s3c24xx\_i2s\_init...

snd\_soc\_register\_dai...

INIT\_LIST\_HEAD(&dai->list);//初始化链表头dai->list

No device for DAI s3c24xx-i2s

snd\_soc\_instantiate\_cards...

s3c24xx\_uda134x:..s3c24xx\_uda134x\_init...

s3c24xx\_uda134x\_probe Machine驱动探测函数（s3c24xx\_uda134x.c）

s3c24xx\_uda134x\_snd\_device = platform\_device\_alloc("soc-audio", -1)

// 注册了一个名为soc-audio的Platform设备

platform\_set\_drvdata(s3c24xx\_uda134x\_snd\_device,&s3c24xx\_uda134x\_snd\_devd

ata);//将devdata设到platform\_device结构的dev.drvdata字段中，

==========================================================

引出结构体s3c24xx\_uda134x\_snd\_devdata（包含三个重要结构体）

static struct snd\_soc\_device s3c24xx\_uda134x\_snd\_devdata = {

.card = &snd\_soc\_s3c24xx\_uda134x, //卡设备结构体

.platform = &s3c24xx\_soc\_platform,

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

.codec\_dev = &soc\_codec\_dev\_uda134x, //codec结构体

.codec\_data = &s3c24xx\_uda134x,//平台结构体

};

===========================================================

S3C24XX\_UDA134X SoC Audio driver

s3c24xx\_uda134x:..s3c24xx\_uda134x\_setup\_pin...

s3c24xx\_uda134x:..s3c24xx\_uda134x\_setup\_pin...

s3c24xx\_uda134x:..s3c24xx\_uda134x\_setup\_pin...

platform\_device\_add(s3c24xx\_uda134x\_snd\_device)

machine:..platform\_device\_add...

machine:..device\_add...

...really\_probe

..............//最终调用了soc-core下的probe函数。

//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Machine驱动部分s3c24xx\_uda134x.c在probe添加设备后，触发soc-core.c //下的soc\_probe

🡺

soc\_probe

snd\_soc\_register\_card

snd\_soc\_instantiate\_cards

list\_for\_each\_entry(card, &card\_list, list)

snd\_soc\_instantiate\_card(card);

list\_for\_each\_entry(platform, &platform\_list, list)

card->platform == platform // s3c24xx-pcm.c

//pcm驱动进入音频核心层便是通过此函数以"插件"的 //形式插入音频子系统

//platform指向的是struct snd\_soc\_platform //s3c24xx\_soc\_platform:

ct snd\_soc\_platform s3c24xx\_soc\_platform = {

.name = "s3c24xx-audio",

.pcm\_ops = &s3c24xx\_pcm\_ops,

.pcm\_new = s3c24xx\_pcm\_new,

.pcm\_free = s3c24xx\_pcm\_free\_dma\_buffers,

};

为每个音频物理链路找到对应的codec、codec\_dai、cpu\_dai、platform设备实例，完成dai\_link的绑定；

list\_for\_each\_entry(dai, &dai\_list, list)

if (card->dai\_link[i].cpu\_dai == dai) // s3c24xx-i2s.c里 面初始化

//函数snd\_soc\_register\_dai()是音频子系统核心层的 //函数.DAI(Digital Audio Interface)便是通过此函数以" //插件"的形式插入音频子系统的核心层

dai指向的是struct snd\_soc\_dai s3c24xx\_i2s\_dai

===============================

struct snd\_soc\_dai s3c24xx\_i2s\_dai = {

.name = "s3c24xx-i2s",

.id = 0,

.probe = s3c24xx\_i2s\_probe,

.suspend = s3c24xx\_i2s\_suspend,

.resume = s3c24xx\_i2s\_resume,

.playback = {

.channels\_min = 2,

.channels\_max = 2,

.rates = S3C24XX\_I2S\_RATES,

.formats = SNDRV\_PCM\_FMTBIT\_S8 | SNDRV\_PCM\_FMTBIT\_S16\_LE,},

.capture = {

.channels\_min = 2,

.channels\_max = 2,

.rates = S3C24XX\_I2S\_RATES,

.formats = SNDRV\_PCM\_FMTBIT\_S8 | SNDRV\_PCM\_FMTBIT\_S16\_LE,},

.ops = &s3c24xx\_i2s\_dai\_ops,

};

=======================================

//依次回调cpu\_dai、codec、platform的probe()函数，完 成物理设备的初始化

card->probe //未赋值

cpu\_dai->probe //平台相关的函数static int //s3c24xx\_i2s\_probe，平台相关的IIS总线 //配置

codec\_dev->probe //编解码器驱动

// sound/soc/codecs/uda134x.c,指向uda134x\_soc\_probe,

----------------------------------------------------------------------------------------------------------------------

uda134x\_soc\_probe

snd\_soc\_new\_pcms

//音频子系统的核心,实现音频控制的设备节点/dev/snd/controlCn,播放音频的设备节点/dev/snd/pcmCiDjp,录制音频的设备节点/dev/snd/pcmCiDjc都是在此函数生成,并在此函数在直接绑定了对应上层用户空间的系统调用//

向音频子系统注册一个声卡和PCM设备,主要完成AD/DA转换

snd\_card\_create //创建声卡

snd\_ctl\_create**//创建control逻辑设备，如音量控制**

//创建了/dev/snd/controlCn(n = 0,1,2,...)设备节点.并且在此设备节点上绑定了 //操作集snd\_ctl\_f\_ops,来最直接对应用户空间的系统调用,储如read、write、 //ioctl等

=====================

static struct snd\_device\_ops ops = {

.dev\_free = snd\_ctl\_dev\_free,

.dev\_register = snd\_ctl\_dev\_register,

.dev\_disconnect = snd\_ctl\_dev\_disconnect,

};

=========================

snd\_device\_new

list\_add(&dev->list,card)

//设备节点的生成及操作集的绑定都是在回调函数

//snd\_ctl\_dev\_register()里面完成

soc\_new\_pcm ) ------

//创建pcm逻辑设备，回放子流实例和录制子流实例都在这 里创建

//生成了音频里面的两个设备节点(播音，录音)

/dev/snd/pcmCiDjp(i = 0,1,2,...; j = 0,1,2,...) //播音 /dev/snd/pcmCiDjc(i = 0,1,2,...; j = 0,1,2,...)//录音

//生成播音和放音这两个设备节点:/dev/snd/pcmCiDjp和/dev/snd/pcmCiDic.并绑定 //了最直接面向用户空间的系统调用的操作集snd\_pcm\_f\_ops.

        //和函数snd\_ctl\_create()相仿,内部都调用了函数snd\_device\_new()

snd\_pcm\_new **//创建pcm逻辑设备**

===============================

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\_pcm\_new\_stream(SNDRV\_PCM\_STREAM\_PLAYBACK)

//回放子流实例创建

snd\_pcm\_new\_stream(SNDRV\_PCM\_STREAM\_CAPTURE);

//录制子流实例创建

snd\_device\_new

snd\_pcm\_set\_ops(pcm,SNDRV\_PCM\_STREAM\_PLAYBACK,

&soc\_pcm\_ops);

snd\_pcm\_set\_ops(pcm, SNDRV\_PCM\_STREAM\_CAPTURE,

&soc\_pcm\_ops);

platform->pcm\_new(codec->card, codec\_dai, pcm)

s3c24xx\_pcm\_new ------------> s3c24xx-pcm.c

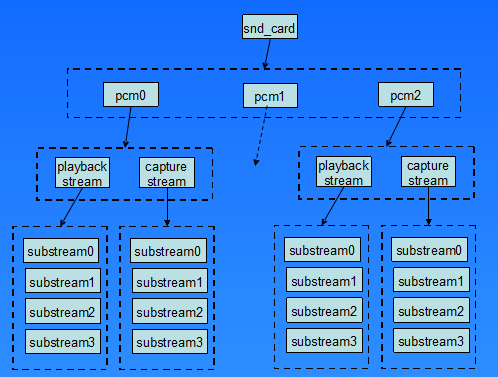
s3c24xx\_pcm\_preallocate\_dma\_buffer

//  回调platform驱动的pcm\_new()，完成音频dma设备初始化和dma buffer 内存分配

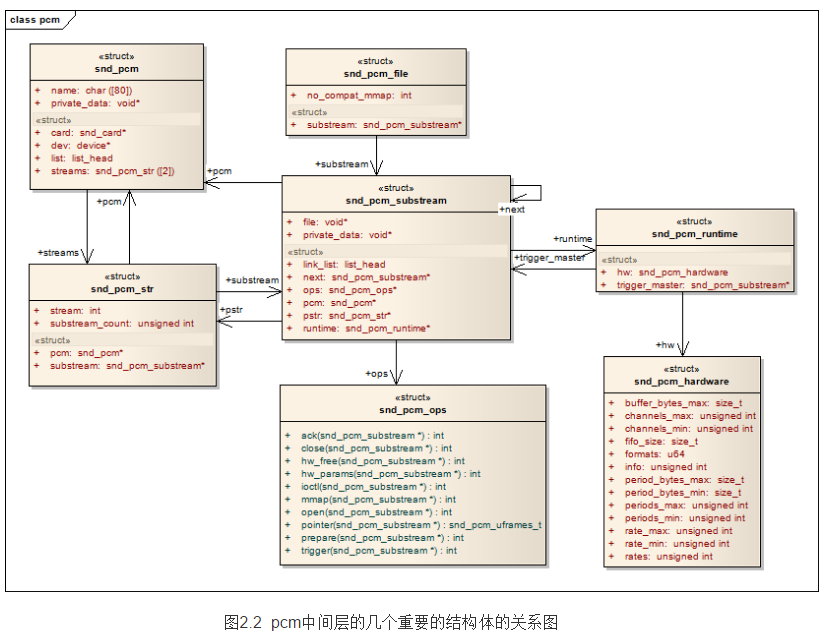
start================================================================

**PCM中间层（逻辑设备）**

一个pcm实例由一个playback stream和一个capture stream组成，这两个stream又分别有一个或多个substreams组成。

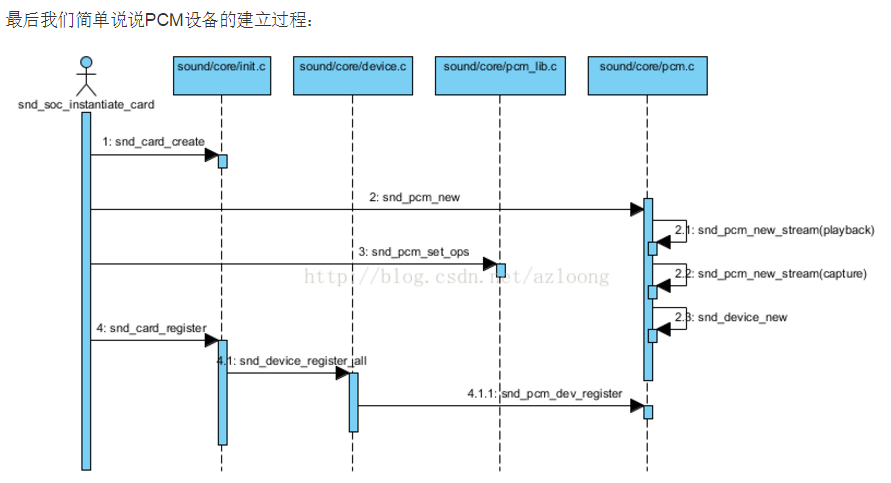


通常情况下一个声卡，一个pcm实例，pcm下面有一个playback和capture stream，playback和capture下面各自有一个substream



结构体说明：

* snd\_pcm是挂在snd\_card下面的一个snd\_device
* snd\_pcm中的字段：streams[2]，该数组中的两个元素指向两个snd\_pcm\_str结构，分别代表playback stream和capture stream
* snd\_pcm\_str中的substream字段，指向snd\_pcm\_substream结构
* snd\_pcm\_substream是pcm中间层的核心，绝大部分任务都是在substream中处理，尤其是他的ops（snd\_pcm\_ops）字段，许多user空间的应用程序通过alsa-lib对驱动程序的请求都是由该结构中的函数处理。它的runtime字段则指向snd\_pcm\_runtime结构，snd\_pcm\_runtime记录这substream的一些重要的软件和硬件运行环境和参数

PCM逻辑设备的建立流程，其他逻辑设备如timer等类似

end================================================================

continue🡺

uda134x\_soc\_probe

snd\_soc\_new\_pcms

snd\_soc\_add\_controls(codec, uda1340\_snd\_controls,

ARRAY\_SIZE(uda1340\_snd\_controls));

snd\_ctl\_add(card, snd\_soc\_cnew(control, codec, NULL));

snd\_ctl\_new1

snd\_ctl\_new……

snd\_soc\_init\_card

snd\_card\_register

snd\_device\_register\_all

dev->ops->dev\_register(dev)//该函数分别会调用ctrl和pcm的回调函数：

snd\_ctl\_dev\_register

snd\_pcm\_dev\_register

snd\_pcm\_dev\_register

case SNDRV\_PCM\_STREAM\_PLAYBACK:播放文件节点名

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

…………………最会会在/dev/snd下生成pcmC0D0p

break;

case SNDRV\_PCM\_STREAM\_CAPTURE:采集文件节点名

sprintf(str,"pcmC%iD%ic",pcm->card->numberpcm->device);

……………………最后会在/dev/snd下生成pcmC0D0c

break;

}

snd\_register\_device\_for\_dev(devtype, pcm->card,

pcm->device,

&snd\_pcm\_f\_ops[cidx],

pcm, str, dev);

//分配并初始化一个snd\_minor实例；

        //保存该snd\_minor实例到snd\_minors数组中；

      //调用device\_create()生成设备文件节点。

该回调函数建立了和用户空间应用程序（alsa-lib）通信所用的设备文件节点:/dev/snd/pcmCxxDxxp和/dev/snd/pcmCxxDxxc

* Playback(播放)    如何把用户空间的应用程序发过来的PCM数据，转化为人耳可以辨别的模拟音频//将数字信息模拟化发到硬件设备。
* Capture（采集）     把mic拾取到得模拟信号，经过采样、量化，转换为PCM信号送回给用户空间的应用程序//将硬件采集的模拟信息数字化发给上层应用。

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

----playback

{

.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,

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

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

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

},

{

------capture

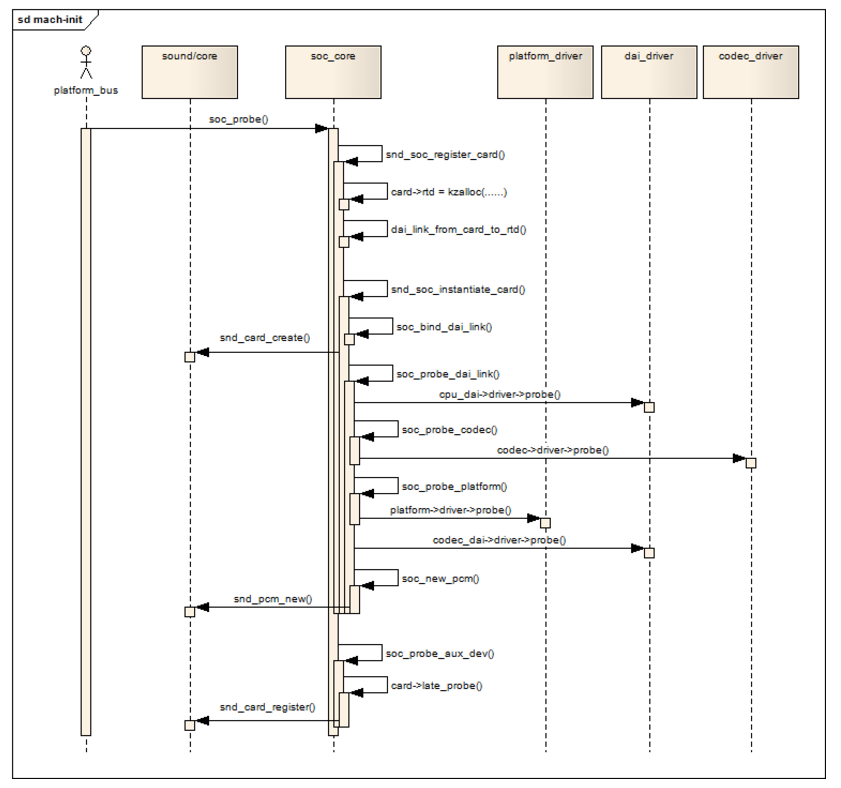
.open = snd\_pcm\_capture\_open,

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

……………………….}与上类似。

};以上为pcm的操作函数。

platform->probe //平台驱动



**Machine驱动soc\_probe初始化流程**

uda134x\_soc\_probe

snd\_soc\_new\_pcms

snd\_card\_create

soc\_new\_pcm

snd\_pcm\_new

snd\_pcm\_new\_stream

snd\_device\_new

snd\_pcm\_set\_ops (pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, &soc\_pcm\_ops);

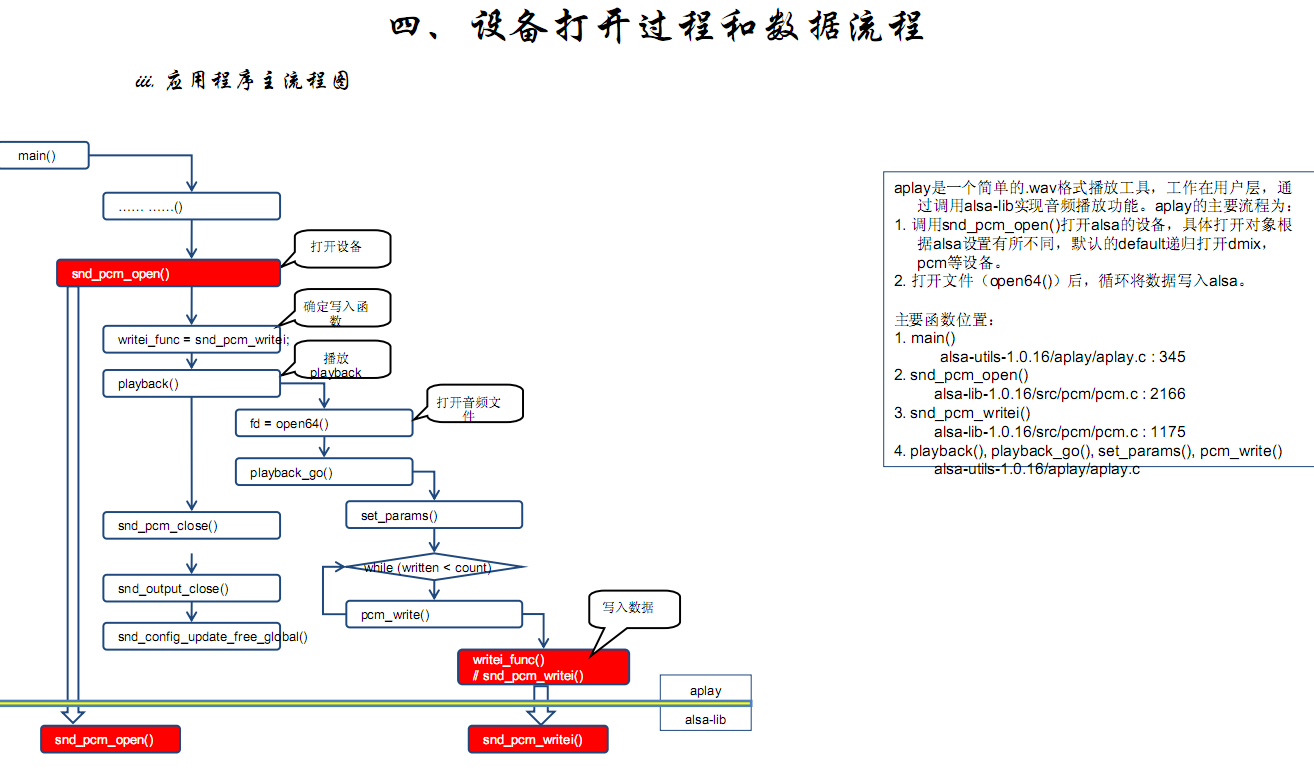
snd\_pcm\_set\_ops

// soc\_pcm\_ops.注意,这个操作集将是上层到IIS硬件层中间层的第二"逻辑层 //"(第一层是上述的snd\_pcm\_f\_ops).并且,在函数snd\_pcm\_set\_ops()暂存在 //动态分配的pcm的streams域

struct snd\_pcm\_substream \*substream

substream->ops = ops

二、static int snd\_pcm\_playback\_open(struct inode \*inode, struct file \*file)



一：应用aplay调用open函数后：

上层应用aplay播放音乐调用，首先通过alsa-lib下的snd\_pcm\_open函数调用alsa-driver下的snd\_pcm\_open函数，打开位于/dev/snd/pcmC0D0p设备，之后打开音频文件，将数据循环写入，通过调用alsa-driver下的snd\_pcm\_writei函数。

实际调用（应该）：播放：alsa-lib下的snd\_pcm\_open，调用alsa-driver下的snd\_pcm\_playback\_open---》snd\_pcm\_open。

sound.c将pcm的ops赋值到sound下，以后调用直接调用pcm的ops函数。

**Sound/core/sound.c字符设备驱动**

alsa\_sound\_init

register\_chrdev(major, "alsa", &snd\_fops)

………………..

static const struct file\_operations snd\_fops =

{

.owner = THIS\_MODULE,

.open = snd\_open,

.llseek = noop\_llseek,

};

// register\_chrdev中的参数major与之前创建pcm设备是device\_create时的major是同一个<即116>，这样的结果是，当应用程序open设备文件/dev/snd/pcmCxDxp时，会进入snd\_fops的open回调函数，

snd\_open

unsigned int minor = iminor(inode);//取出次设备号

mptr = snd\_minors[minor]; 以次设备号为索引取出当初注册pcm设备时填充的snd\_minor结构

file->f\_op = fops\_get(mptr->f\_ops);// 取出pcm设备的f\_ops，并且把file->f\_op替换为pcm设备的f\_ops

file->f\_op->open//调用pcm设备的f\_ops->open()

以后，应用程序的所有read/write/ioctl调用都会进入pcm设备自己的回调函数中，也就是之前提到的snd\_pcm\_f\_ops结构中定义的回调。

实际的open状况：

Alsa-lib：snd\_pcm\_open，打开设备文件pcmC0D0p，其主设备号与core/sound注册的设备号相同，

调用sound字符设备下file\_operations 下的open函数，该函数将pcm的ops赋值给sound的ops，之后调用open，进入snd\_pcm\_f\_ops下的snd\_pcm\_playback\_open函数，之后—》snd\_pcm\_open

snd\_pcm\_open\_file

snd\_pcm\_open\_substream

snd\_pcm\_attach\_substream

\*rsubstream = substream

substream->ops->open(substream)

/\*在snd\_pcm\_set\_ops 中定义了substream->ops=ops（&soc\_pcm\_ops） \*/

………………….

rtd->ops.open = soc\_pcm\_open;

cpu\_dai->driver->ops->startup

platform->driver->ops->open //如果platform主要为dma操作。。。 结构体：s3c24xx\_pcm\_ops

s3c24xx\_pcm\_open -----> s3c24xx\_pcm.c

snd\_soc\_set\_runtime\_hwparams

// msm-pcm-q6-v2:msm\_pcm\_open实际调用函数。

codec\_dai->driver->ops->startup//codec

uda134x\_startup -----> uda134x.c

uda134x\_write

struct uda134x\_platform\_data \*pd=codec->control\_data;

l3\_write (&pd->l3,addr, &data, 1);

setclk (在s3c24xx\_uda134x.c中定义)

setdat

setmode …….

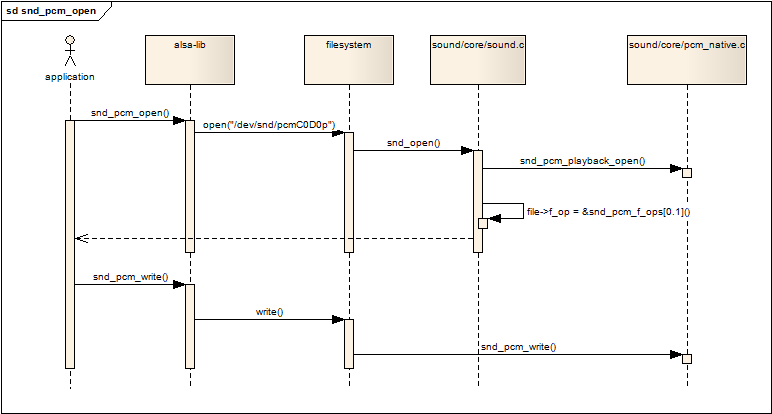
………………..

machine->ops->startup //machine

s3c24xx\_uda134x\_startup

至此open完毕。

//   pcm还是那个pcm,播音录音音频流的操作集(pcm->stream->ops)在函 //数snd\_pcm\_set\_ops()都被赋值为soc\_pcm\_ops.并且在open函数里面被/ //保存在file->private\_data



Open完成后：soc\_pcm\_ops下的

二：Aplay应用程序：set\_params();

Pcm逻辑中间层触发了下层物理设备codec、platform、machine的操作函数，实现音频数据在dma\_buffer<-> cpu\_dai <-> codec之间的传输。

soc\_pcm\_hw\_params

machine->ops->hw\_params

s3c24xx\_uda134x\_hw\_params

codec\_dai->ops->hw\_params

uda134x\_hw\_params

uda134x\_write

setclk (在s3c24xx\_uda134x.c中定义)

setdat

setmode …….

………………..

cpu\_dai->ops->hw\_params

s3c24xx\_i2s\_hw\_params

platform->pcm\_ops->hw\_params

s3c24xx\_pcm\_hw\_params

soc\_pcm\_prepare

machine->ops->prepare

platform->pcm\_ops->prepare

s3c24xx\_pcm\_prepare

codec\_dai->ops->prepare

cpu\_dai->ops->prepare

三：应用程序aplay 执行写数据播放音乐：

Playback()

playback\_go()

set\_params()

pcm\_write()

writei\_func= snd\_pcm\_writei

sound驱动部分：

…………………主要是write部分驱动

soc\_pcm\_trigger

具体的sound部分驱动实现：

WRITE:

以write为例,看用户空间的音频数据流是如何流窜到硬件IIS的.大体流程如下

file\_operations snd\_pcm\_f\_ops

.write = snd\_pcm\_write

pcm\_file = file->private\_data

substream = pcm\_file->substream

snd\_pcm\_lib\_write(substream, buf, count)

snd\_pcm\_lib\_write1

snd\_pcm\_update\_hw\_ptr

snd\_pcm\_update\_hw\_ptr\_pos

substream->ops->pointer

s3c24xx\_pcm\_pointer //多次触发该函数。目的不明

snd\_pcm\_start

snd\_pcm\_action (&snd\_pcm\_action\_start,

substream,SNDRV\_PCM\_STATE\_RUNNING);

================

static struct action\_ops snd\_pcm\_action\_start = {

.pre\_action = snd\_pcm\_pre\_start,

.do\_action = snd\_pcm\_do\_start,

.undo\_action = snd\_pcm\_undo\_start,

.post\_action = snd\_pcm\_post\_start

};

==========================

snd\_pcm\_action\_single

ops->do\_action //即static int snd\_pcm\_do\_start

🡪 snd\_pcm\_do\_start

substream->ops->trigger

//其中substream->ops->trigger=ops(soc\_pcm\_ops)

//即static int soc\_pcm\_trigger

🡪soc\_pcm\_trigger

codec\_dai->ops->trigger

platform->pcm\_ops->trigger

cpu\_dai->ops->trigger //调用了s3c24xx-i2s.c

🡪 s3c24xx\_pcm\_trigger

🡪 s3c24xx\_i2s\_trigger

写音频数据到音频接口IIS流程：

UserSpace ->

snd\_pcm\_f\_ops -> snd\_pcm\_action\_start -> soc\_pcm\_ops -> s3c24xx\_i2s\_trigger

主要是由上层写数据到s3c2440主控的硬件iiS接口处。

**大体的音频播放读写流程如上，以后待进一步分析。**

**声卡逻辑设备control创建和绑定**

声卡编解码器驱动：

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

Platform和Codec的DAI通过snd\_soc\_dai\_link结构，在Machine驱动中进行绑定连接

uda134x\_soc\_probe

snd\_soc\_new\_pcms

snd\_soc\_add\_controls (codec, uda1341\_snd\_controls,size);

// uda1341\_snd\_controls为代码中定义的control类别。在如下结构体中定义：

==================================================

static const struct snd\_kcontrol\_new uda1341\_snd\_controls[] = {。。。。。。}；

==================================================

//创建control设备

const struct snd\_kcontrol\_new \*control = &controls[i];

snd\_ctl\_add(card, snd\_soc\_cnew(control, codec, NULL));

snd\_soc\_cnew

kctl.info = ncontrol->info;

kctl.get = ncontrol->get;

kctl.put = ncontrol->put;

kctl.tlv.p = ncontrol->tlv.p;//

//其中ncontrol便为uda1341\_snd\_controls[]结构体数组中的项

//ncontrol即为uda1340\_snd\_controls

================================

定义相关的control项，

static const struct snd\_kcontrol\_new uda1341\_snd\_controls[] = {

SOC\_SINGLE("Master Playback Volume", UDA134X\_DATA000, 0, 0x3F, 1),

SOC\_SINGLE("Capture Volume", UDA134X\_EA010, 2, 0x07, 0),

SOC\_SINGLE("Analog1 Volume", UDA134X\_EA000, 0, 0x1F, 1),

……}

宏展开：

#define SOC\_SINGLE(xname, reg, shift, max, invert) \

{ .iface = SNDRV\_CTL\_ELEM\_IFACE\_MIXER, .name = xname, \

.info = snd\_soc\_info\_volsw, .get = snd\_soc\_get\_volsw,\

// info回调函数用于获取control的详细信息

主要工作就是填充通过参数传入的snd\_ctl\_elem\_info对象

.get回调用于读取control的当前值，并返回给用户空间的应用程序

.put = snd\_soc\_put\_volsw, \

//put回调函数用于把应用程序的控制值设置到control

.private\_value = SOC\_SINGLE\_VALUE(reg, shift, max, invert) }

**音频CONTROL实例：amixer**

amixer cset numid=1 45 //应用程序amixer设置音量大小执行流程：

==> /dev/snd/controlC 绑定 snd\_ctl\_f\_ops //从设备节点到底层寄存器

snd\_ctl\_f\_ops

snd\_ctl\_open...

snd\_lookup\_minor\_data...

snd\_card\_file\_add...

snd\_ctl\_ioctl...

snd\_ctl\_card\_info...

snd\_ctl\_elem\_info\_user...

kctl->info(kctl, info);

snd\_soc\_info\_volsw...

snd\_ctl\_ioctl...

snd\_ctl\_elem\_write\_user...

kctl->put(kctl, control);

++++++++++++++++++++++++++++++++++++++

snd\_soc\_add\_controls

snd\_ctl\_add

SOC\_SINGLE("Analog1 Volume", UDA134X\_EA000, 0, 0x1F, 1)

+++++++++++++++++++++++++++++++++++++++

snd\_soc\_put\_volsw...

snd\_soc\_update\_bits...

snd\_soc\_write

codec->write

codec->write = uda134x\_write;

//出自uda134x\_soc\_probe

uda134x\_write...

uda134x\_write\_reg\_cache...

setclk...

setdat...

setmode...

大体的执行流程:

UserSpace-->snd\_ctl\_f\_ops-->[kctl->put]-->uda1340\_snd\_controls->put

主要是上层应用通过音频芯片的操作函数，进行写数据。

**DAPM-------动态音频电源管理**

**待定**