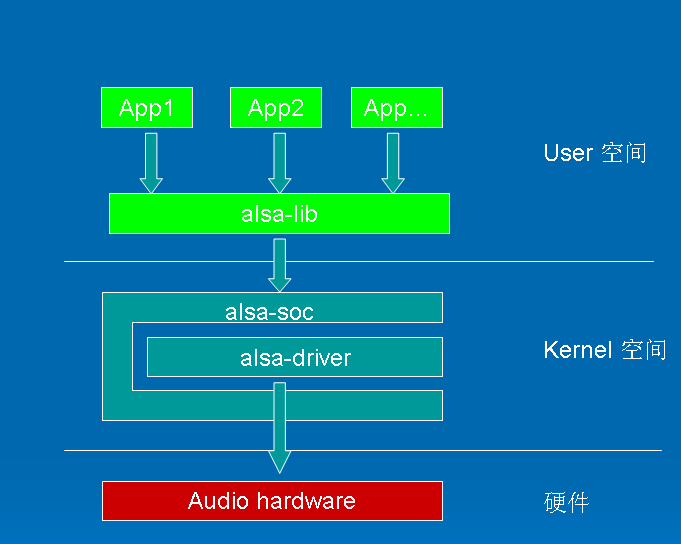
## [ALSA架构简介](http://blog.csdn.net/droidphone/article/details/6271122)

<http://blog.csdn.net/DroidPhone/article/category/1118446>

ALSA是Advanced Linux Sound Architecture 的缩写，目前已经成为了linux的主流音频体系结构，想了解更多的关于ALSA的这一开源项目的信息和知识，请查看以下网址：http://www.alsa-project.org/。

在内核设备驱动层，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。



由图1.1可以看出，用户空间的alsa-lib对应用程序提供统一的API接口，这样可以隐藏了驱动层的实现细节，简化了应用程序的实现难度。内核空间中，alsa-soc其实是对alsa-driver的进一步封装，他针对嵌入式设备提供了一些列增强的功能。本系列博文仅对嵌入式系统中的alsa-driver和alsa-soc进行讨论。

### ALSA设备文件结构

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

$ cd /dev/snd

$ ls -l

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

crw-rw----+ 1 root audio 116, 4 2011-02-23 21:38 midiC0D0

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

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

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

crw-rw----+ 1 root audio 116, 3 2011-02-23 21:38 seq

crw-rw----+ 1 root audio 116, 2 2011-02-23 21:38 timer

我们可以看到以下设备文件:

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

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

其中，C0D0代表的是声卡0中的设备0，pcmC0D0c最后一个c代表capture，pcmC0D0p最后一个p代表playback，这些都是alsa-driver中的命名规则。从上面的列表可以看出，我的声卡下挂了6个设备，**根据声卡的实际能力，驱动实际上可以挂上更多种类的设备**，在include/sound/core.h中，定义了以下设备类型：

#define SNDRV\_DEV\_TOPLEVEL ((\_\_force snd\_device\_type\_t) 0)

#define SNDRV\_DEV\_CONTROL ((\_\_force snd\_device\_type\_t) 1)

#define SNDRV\_DEV\_LOWLEVEL\_PRE ((\_\_force snd\_device\_type\_t) 2)

#define SNDRV\_DEV\_LOWLEVEL\_NORMAL ((\_\_force snd\_device\_type\_t) 0x1000)

#define SNDRV\_DEV\_PCM ((\_\_force snd\_device\_type\_t) 0x1001)

#define SNDRV\_DEV\_RAWMIDI ((\_\_force snd\_device\_type\_t) 0x1002)

#define SNDRV\_DEV\_TIMER ((\_\_force snd\_device\_type\_t) 0x1003)

#define SNDRV\_DEV\_SEQUENCER ((\_\_force snd\_device\_type\_t) 0x1004)

#define SNDRV\_DEV\_HWDEP ((\_\_force snd\_device\_type\_t) 0x1005)

#define SNDRV\_DEV\_INFO ((\_\_force snd\_device\_type\_t) 0x1006)

#define SNDRV\_DEV\_BUS ((\_\_force snd\_device\_type\_t) 0x1007)

#define SNDRV\_DEV\_CODEC ((\_\_force snd\_device\_type\_t) 0x1008)

#define SNDRV\_DEV\_JACK ((\_\_force snd\_device\_type\_t) 0x1009)

#define SNDRV\_DEV\_LOWLEVEL ((\_\_force snd\_device\_type\_t) 0x2000)

通常，我们更关心的是pcm和control这两种设备。

### 驱动的代码文件结构

在Linux2.6代码树中，Alsa的代码文件结构如下：

sound

/core

/oss

/seq

/ioctl32

/include

/drivers

/i2c

/synth

/emux

/pci

/(cards)

/isa

/(cards)

/arm

/ppc

/sparc

/usb

/pcmcia /(cards)

/oss

/soc

/codecs

core 该目录包含了ALSA驱动的中间层，它是整个ALSA驱动的核心部分

core/oss 包含模拟旧的OSS架构的PCM和Mixer模块

core/seq 有关音序器相关的代码

include ALSA驱动的公共头文件目录，该目录的头文件需要导出给用户空间的应用程序使用，通常，驱动模块私有的头文件不应放置在这里

drivers 放置一些与CPU、BUS架构无关的公用代码

i2c ALSA自己的I2C控制代码

pci pci声卡的顶层目录，子目录包含各种pci声卡的代码

isa isa声卡的顶层目录，子目录包含各种isa声卡的代码

soc 针对system-on-chip体系的中间层代码

soc/codecs 针对soc体系的各种codec的代码，与平台无关

## 声卡的创建

### struct snd\_card

#### snd\_card是什么

snd\_card可以说是整个ALSA音频驱动最顶层的一个结构，整个声卡的软件逻辑结构开始于该结构，几乎所有与声音相关的逻辑设备都是在snd\_card的管理之下，声卡驱动的第一个动作通常就是创建一个snd\_card结构体。正因为如此，本节中，我们也从 struct cnd\_card开始吧。

#### snd\_card的定义

/\* main structure for soundcard \*/

struct snd\_card {

int number; /\* number of soundcard (index to

snd\_cards) \*/

char id[16]; /\* id string of this card \*/

char driver[16]; /\* driver name \*/

char shortname[32]; /\* short name of this soundcard \*/

char longname[80]; /\* name of this soundcard \*/

char mixername[80]; /\* mixer name \*/

char components[128]; /\* card components delimited with

space \*/

struct module \*module; /\* top-level module \*/

void \*private\_data; /\* private data for soundcard \*/

void (\*private\_free) (struct snd\_card \*card); /\* callback for freeing of

private data \*/

struct list\_head devices; /\* devices \*/

unsigned int last\_numid; /\* last used numeric ID \*/

struct rw\_semaphore controls\_rwsem; /\* controls list lock \*/

rwlock\_t ctl\_files\_rwlock; /\* ctl\_files list lock \*/

int controls\_count; /\* count of all controls \*/

int user\_ctl\_count; /\* count of all user controls \*/

struct list\_head controls; /\* all controls for this card \*/

struct list\_head ctl\_files; /\* active control files \*/

struct snd\_info\_entry \*proc\_root; /\* root for soundcard specific files \*/

struct snd\_info\_entry \*proc\_id; /\* the card id \*/

struct proc\_dir\_entry \*proc\_root\_link; /\* number link to real id \*/

struct list\_head files\_list; /\* all files associated to this card \*/

struct snd\_shutdown\_f\_ops \*s\_f\_ops; /\* file operations in the shutdown

state \*/

spinlock\_t files\_lock; /\* lock the files for this card \*/

int shutdown; /\* this card is going down \*/

int free\_on\_last\_close; /\* free in context of file\_release \*/

wait\_queue\_head\_t shutdown\_sleep;

struct device \*dev; /\* device assigned to this card \*/

#ifndef CONFIG\_SYSFS\_DEPRECATED

struct device \*card\_dev; /\* cardX object for sysfs \*/

#endif

#ifdef CONFIG\_PM

unsigned int power\_state; /\* power state \*/

struct mutex power\_lock; /\* power lock \*/

wait\_queue\_head\_t power\_sleep;

#endif

#if defined(CONFIG\_SND\_MIXER\_OSS) || defined(CONFIG\_SND\_MIXER\_OSS\_MODULE)

struct snd\_mixer\_oss \*mixer\_oss;

int mixer\_oss\_change\_count;

#endif

};

struct list\_head devices 记录该声卡下所有逻辑设备的链表

struct list\_head controls 记录该声卡下所有的控制单元的链表

void \*private\_data 声卡的私有数据，可以在创建声卡时通过参数指定数据的大小

### 声卡的建立流程

#### 创建snd\_card的一个实例

struct snd\_card \*card;

int err;

....

err = snd\_card\_create(index, id, THIS\_MODULE, 0, &card);

#### 创建声卡的芯片专用数据

声卡的专用数据主要用于存放该声卡的一些资源信息，例如中断资源、io资源、dma资源等。可以有两种创建方法：

**通过上一步中snd\_card\_create()中的第四个参数，让snd\_card\_create自己创建**

// struct mychip 用于保存专用数据

err = snd\_card\_create(index, id, THIS\_MODULE,

sizeof(struct mychip), &card);

// 从private\_data中取出

struct mychip \*chip = card->private\_data;

**自己创建：**

struct mychip {

struct snd\_card \*card;

....

};

struct snd\_card \*card;

struct mychip \*chip;

chip = kzalloc(sizeof(\*chip), GFP\_KERNEL);

......

err = snd\_card\_create(index[dev], id[dev], THIS\_MODULE, 0, &card);

// 专用数据记录snd\_card实例

chip->card = card;

.....

然后，把芯片的专有数据注册为声卡的一个低阶设备：

static int snd\_mychip\_dev\_free(struct snd\_device \*device)

{

return snd\_mychip\_free(device->device\_data);

}

static struct snd\_device\_ops ops = {

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

};

....

snd\_device\_new(card, SNDRV\_DEV\_LOWLEVEL, chip, &ops);

注册为低阶设备主要是为了当声卡被注销时，芯片专用数据所占用的内存可以被自动地释放。

#### 设置Driver的ID和名字

strcpy(card->driver, "My Chip");

strcpy(card->shortname, "My Own Chip 123");

sprintf(card->longname, "%s at 0x%lx irq %i",

card->shortname, chip->ioport, chip->irq);

snd\_card的driver字段保存着芯片的ID字符串，user空间的alsa-lib会使用到该字符串，所以必须要保证该ID的唯一性。shortname字段更多地用于打印信息，longname字段则会出现在/proc/asound/cards中。

#### 创建声卡的功能部件（逻辑设备），例如PCM，Mixer，MIDI等

这时候可以创建声卡的各种功能部件了，还记得开头的snd\_card结构体的devices字段吗？每一种部件的创建最终会调用snd\_device\_new()来生成一个snd\_device实例，并把该实例链接到snd\_card的devices链表中。

通常，alsa-driver的已经提供了一些常用的部件的创建函数，而不必直接调用snd\_device\_new()，比如：

PCM ---- snd\_pcm\_new()

RAWMIDI -- snd\_rawmidi\_new()

CONTROL -- snd\_ctl\_create()

TIMER -- snd\_timer\_new()

INFO -- snd\_card\_proc\_new()

JACK -- snd\_jack\_new()

#### 注册声卡

err = snd\_card\_register(card);

if (err < 0) {

snd\_card\_free(card);

return err;

}

#### 一个实际的例子

/sound/arm/pxa2xx-ac97.c的部分代码贴上来：

static int \_\_devinit pxa2xx\_ac97\_probe(struct platform\_device \*dev)

{

struct snd\_card \*card;

struct snd\_ac97\_bus \*ac97\_bus;

struct snd\_ac97\_template ac97\_template;

int ret;

pxa2xx\_audio\_ops\_t \*pdata = dev->dev.platform\_data;

if (dev->id >= 0) {

dev\_err(&dev->dev, "PXA2xx has only one AC97 port./n");

ret = -ENXIO;

goto err\_dev;

}

////(1)////

ret = snd\_card\_create(SNDRV\_DEFAULT\_IDX1, SNDRV\_DEFAULT\_STR1,

THIS\_MODULE, 0, &card);

if (ret < 0)

goto err;

card->dev = &dev->dev;

////(3)////

strncpy(card->driver, dev->dev.driver->name, sizeof(card->driver));

////(4)////

ret = pxa2xx\_pcm\_new(card, &pxa2xx\_ac97\_pcm\_client, &pxa2xx\_ac97\_pcm);

if (ret)

goto err;

////(2)////

ret = pxa2xx\_ac97\_hw\_probe(dev);

if (ret)

goto err;

////(4)////

ret = snd\_ac97\_bus(card, 0, &pxa2xx\_ac97\_ops, NULL, &ac97\_bus);

if (ret)

goto err\_remove;

memset(&ac97\_template, 0, sizeof(ac97\_template));

ret = snd\_ac97\_mixer(ac97\_bus, &ac97\_template, &pxa2xx\_ac97\_ac97);

if (ret)

goto err\_remove;

////(3)////

snprintf(card->shortname, sizeof(card->shortname),

"%s", snd\_ac97\_get\_short\_name(pxa2xx\_ac97\_ac97));

snprintf(card->longname, sizeof(card->longname),

"%s (%s)", dev->dev.driver->name, card->mixername);

if (pdata && pdata->codec\_pdata[0])

snd\_ac97\_dev\_add\_pdata(ac97\_bus->codec[0], pdata->codec\_pdata[0]);

snd\_card\_set\_dev(card, &dev->dev);

////(5)////

ret = snd\_card\_register(card);

if (ret == 0) {

platform\_set\_drvdata(dev, card);

return 0;

}

err\_remove:

pxa2xx\_ac97\_hw\_remove(dev);

err:

if (card)

snd\_card\_free(card);

err\_dev:

return ret;

}

static int \_\_devexit pxa2xx\_ac97\_remove(struct platform\_device \*dev)

{

struct snd\_card \*card = platform\_get\_drvdata(dev);

if (card) {

snd\_card\_free(card);

platform\_set\_drvdata(dev, NULL);

pxa2xx\_ac97\_hw\_remove(dev);

}

return 0;

}

static struct platform\_driver pxa2xx\_ac97\_driver = {

.probe = pxa2xx\_ac97\_probe,

.remove = \_\_devexit\_p(pxa2xx\_ac97\_remove),

.driver = {

.name = "pxa2xx-ac97",

.owner = THIS\_MODULE,

#ifdef CONFIG\_PM

.pm = &pxa2xx\_ac97\_pm\_ops,

#endif

},

};

static int \_\_init pxa2xx\_ac97\_init(void)

{

return platform\_driver\_register(&pxa2xx\_ac97\_driver);

}

static void \_\_exit pxa2xx\_ac97\_exit(void)

{

platform\_driver\_unregister(&pxa2xx\_ac97\_driver);

}

module\_init(pxa2xx\_ac97\_init);

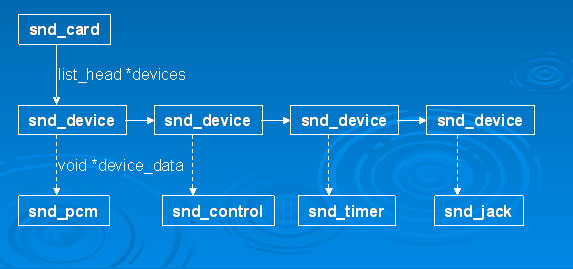
module\_exit(pxa2xx\_ac97\_exit);

MODULE\_AUTHOR("Nicolas Pitre");

MODULE\_DESCRIPTION("AC97 driver for the Intel PXA2xx chip");

驱动程序通常由probe回调函数开始，对一下2.1中的步骤，是否有相似之处？

经过以上的创建步骤之后，声卡的逻辑结构如下图所示：



下面的章节里我们分别讨论一下snd\_card\_create()和snd\_card\_register()这两个函数。

### snd\_card\_create()

snd\_card\_create()在/sound/core/init.c中定义。

首先，根据extra\_size参数的大小分配内存，该内存区可以作为芯片的专有数据使用（见前面的介绍）：

card = kzalloc(sizeof(\*card) + extra\_size, GFP\_KERNEL);

if (!card)

return -ENOMEM;

拷贝声卡的ID字符串：

if (xid)

strlcpy(card->id, xid, sizeof(card->id));

如果传入的声卡编号为-1，自动分配一个索引编号：

if (idx < 0) {

for (idx2 = 0; idx2 < SNDRV\_CARDS; idx2++)

/\* idx == -1 == 0xffff means: take any free slot \*/

if (~snd\_cards\_lock & idx & 1<<idx2) {

if (module\_slot\_match(module, idx2)) {

idx = idx2;

break;

}

}

}

if (idx < 0) {

for (idx2 = 0; idx2 < SNDRV\_CARDS; idx2++)

/\* idx == -1 == 0xffff means: take any free slot \*/

if (~snd\_cards\_lock & idx & 1<<idx2) {

if (!slots[idx2] || !\*slots[idx2]) {

idx = idx2;

break;

}

}

}

初始化snd\_card结构中必要的字段：

建立逻辑设备：Control

err = snd\_ctl\_create(card);

建立proc文件中的info节点：通常就是/proc/asound/card0

把第一步分配的内存指针放入private\_data字段中：

if (extra\_size > 0)

card->private\_data = (char \*)card + sizeof(struct snd\_card);

### snd\_card\_register

snd\_card\_create()在/sound/core/init.c中定义。

/\*\*

\* snd\_card\_register - register the soundcard

\* @card: soundcard structure

\*

\* This function registers all the devices assigned to the soundcard.

\* Until calling this, the ALSA control interface is blocked from the

\* external accesses. Thus, you should call this function at the end

\* of the initialization of the card.

\*

\* Returns zero otherwise a negative error code if the registrain failed.

\*/

int snd\_card\_register(struct snd\_card \*card)

首先，创建sysfs下的设备：

if (!card->card\_dev) {

card->card\_dev = device\_create(sound\_class, card->dev,

MKDEV(0, 0), card,

"card%i", card->number);

if (IS\_ERR(card->card\_dev))

card->card\_dev = NULL;

}

其中，sound\_class是在/sound/sound\_core.c中创建的：

static char \*sound\_devnode(struct device \*dev, mode\_t \*mode)

{

if (MAJOR(dev->devt) == SOUND\_MAJOR)

return NULL;

return kasprintf(GFP\_KERNEL, "snd/%s", dev\_name(dev));

}

static int \_\_init init\_soundcore(void)

{

int rc;

rc = init\_oss\_soundcore();

if (rc)

return rc;

sound\_class = class\_create(THIS\_MODULE, "sound");

if (IS\_ERR(sound\_class)) {

cleanup\_oss\_soundcore();

return PTR\_ERR(sound\_class);

}

sound\_class->devnode = sound\_devnode;

return 0;

}

由此可见，声卡的class将会出现在文件系统的/sys/class/sound/下面，并且，sound\_devnode()也决定了相应的设备节点也将会出现在/dev/snd/下面。

接下来的步骤，通过snd\_device\_register\_all()注册所有挂在该声卡下的逻辑设备，snd\_device\_register\_all()实际上是通过snd\_card的devices链表，遍历所有的snd\_device，并且调用snd\_device的ops->dev\_register()来实现各自设备的注册的。

if ((err = snd\_device\_register\_all(card)) < 0)

return err;

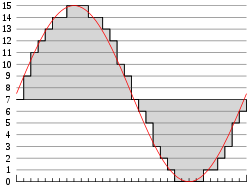
最后就是建立一些相应的proc和sysfs下的文件或属性节点，代码就不贴了。

至此，整个声卡完成了建立过程。

## PCM设备的创建

### PCM是什么

PCM是英文Pulse-code modulation的缩写，中文译名是脉冲编码调制。我们知道在现实生活中，人耳听到的声音是模拟信号，PCM就是要把声音从模拟转换成数字信号的一种技术，他的原理简单地说就是利用一个固定的频率对模拟信号进行采样，采样后的信号在波形上看就像一串连续的幅值不一的脉冲，把这些脉冲的幅值按一定的精度进行量化，这些量化后的数值被连续地输出、传输、处理或记录到存储介质中，所有这些组成了数字音频的产生过程。



PCM信号的两个重要指标是采样频率和量化精度，目前，CD音频的采样频率通常为44100Hz，量化精度是16bit。通常，播放音乐时，应用程序从存储介质中读取音频数据（MP3、WMA、AAC......），经过解码后，最终送到音频驱动程序中的就是PCM数据，反过来，在录音时，音频驱动不停地把采样所得的PCM数据送回给应用程序，由应用程序完成压缩、存储等任务。所以，音频驱动的两大核心任务就是：

playback 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。

capture 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。

### alsa-driver中的PCM中间层

ALSA已经为我们实现了功能强劲的PCM中间层，**自己的驱动中只要实现一些底层的需要访问硬件的函数即可。**

要访问PCM的中间层代码，你首先要包含头文件<sound/pcm.h>，另外，如果需要访问一些与 hw\_param相关的函数，可能也要包含<sound/pcm\_params.h>。

**每个声卡最多可以包含4个pcm的实例，每个pcm实例对应一个pcm设备文件。**pcm实例数量的这种限制源于linux设备号所占用的位大小，如果以后使用64位的设备号，我们将可以创建更多的pcm实例。不过大多数情况下，在嵌入式设备中，一个pcm实例已经足够了。

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。

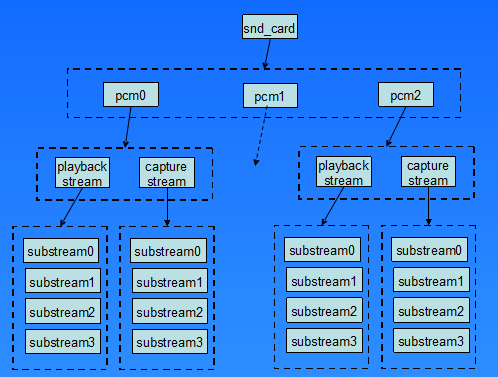
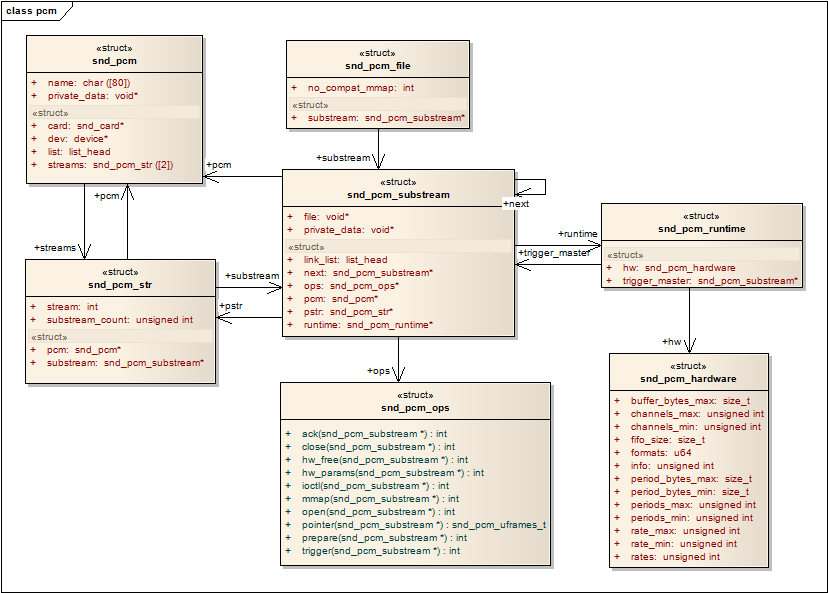


图2.1 声卡中的pcm结构

在嵌入式系统中，通常不会像图2.1中这么复杂，大多数情况下是一个声卡，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。

下面一张图列出了pcm中间层几个重要的结构，他可以让我们从uml的角度看一看这列结构的关系，理清他们之间的关系，对我们理解pcm中间层的实现方式。



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。**

### 新建一个pcm

alsa-driver的中间层已经为我们提供了新建pcm的api：

int snd\_pcm\_new(struct snd\_card \*card, const char \*id, int device, int playback\_count, int capture\_count,

struct snd\_pcm \*\* rpcm);

参数device 表示目前创建的是该声卡下的第几个pcm，第一个pcm设备从0开始。

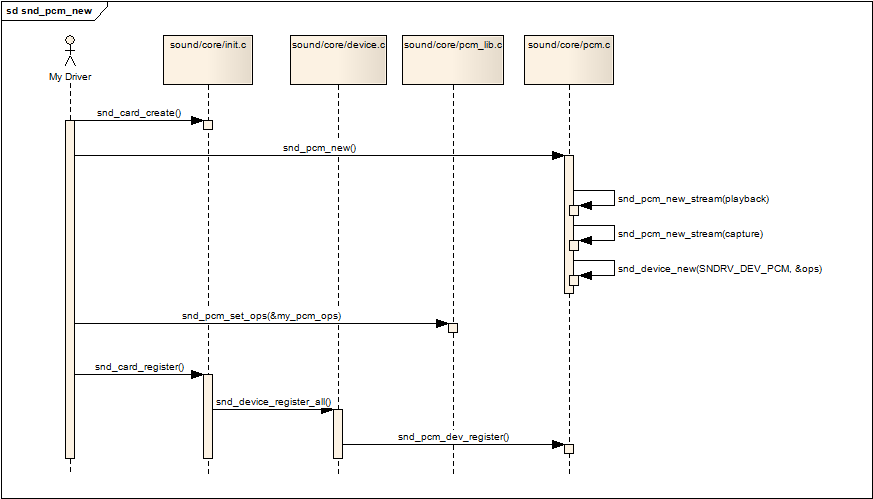
参数playback\_count 表示该pcm将会有几个playback substream。

参数capture\_count 表示该pcm将会有几个capture substream。

另一个用于设置pcm操作函数接口的api：

void snd\_pcm\_set\_ops(struct snd\_pcm \*pcm, int direction, struct snd\_pcm\_ops \*ops);

新建一个pcm可以用下面一张新建pcm的调用的序列图进行描述：



snd\_card\_create pcm是声卡下的一个设备（部件），所以第一步是要创建一个声卡

snd\_pcm\_new 调用该api创建一个pcm，才该api中会做以下事情

如果有，建立playback stream，相应的substream也同时建立

如果有，建立capture stream，相应的substream也同时建立

调用snd\_device\_new()把该pcm挂到声卡中，参数ops中的dev\_register字段指向了函数snd\_pcm\_dev\_register，这个回调函数会在声卡的注册阶段被调用。

snd\_pcm\_set\_ops 设置操作该pcm的控制/操作接口函数，参数中的snd\_pcm\_ops结构中的函数通常就是我们驱动要实现的函数

snd\_card\_register 注册声卡，在这个阶段会遍历声卡下的所有逻辑设备，并且调用各设备的注册回调函数，对于pcm，就是第二步提到的snd\_pcm\_dev\_register函数，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

### 设备文件节点的建立（dev/snd/pcmCxxDxxp、pcmCxxDxxc）

#### struct snd\_minor

每个snd\_minor结构体保存了声卡下某个逻辑设备的上下文信息，他在逻辑设备建立阶段被填充，在逻辑设备被使用时就可以从该结构体中得到相应的信息。pcm设备也不例外，也需要使用该结构体。该结构体在include/sound/core.h中定义。

struct snd\_minor {

int type; /\* SNDRV\_DEVICE\_TYPE\_XXX \*/

int card; /\* card number \*/

int device; /\* device number \*/

const struct file\_operations \*f\_ops; /\* file operations \*/

void \*private\_data; /\* private data for f\_ops->open \*/

struct device \*dev; /\* device for sysfs \*/

};

在sound/sound.c中定义了一个snd\_minor指针的全局数组：

static struct snd\_minor \*snd\_minors[256];

前面说过，在声卡的注册阶段（snd\_card\_register），会调用pcm的回调函数snd\_pcm\_dev\_register()，这个函数里会调用函数snd\_register\_device\_for\_dev()：

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

{

......

/\* register pcm \*/

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

pcm->device,

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

pcm, str, dev);

......

}

我们再进入snd\_register\_device\_for\_dev()：

int snd\_register\_device\_for\_dev(int type, struct snd\_card \*card, int dev,

const struct file\_operations \*f\_ops,

void \*private\_data,

const char \*name, struct device \*device)

{

int minor;

struct snd\_minor \*preg;

if (snd\_BUG\_ON(!name))

return -EINVAL;

preg = kmalloc(sizeof \*preg, GFP\_KERNEL);

if (preg == NULL)

return -ENOMEM;

preg->type = type;

preg->card = card ? card->number : -1;

preg->device = dev;

preg->f\_ops = f\_ops;

preg->private\_data = private\_data;

mutex\_lock(&sound\_mutex);

#ifdef CONFIG\_SND\_DYNAMIC\_MINORS

minor = snd\_find\_free\_minor();

#else

minor = snd\_kernel\_minor(type, card, dev);

if (minor >= 0 && snd\_minors[minor])

minor = -EBUSY;

#endif

if (minor < 0) {

mutex\_unlock(&sound\_mutex);

kfree(preg);

return minor;

}

snd\_minors[minor] = preg;

preg->dev = device\_create(sound\_class, device, MKDEV(major, minor),

private\_data, "%s", name);

if (IS\_ERR(preg->dev)) {

snd\_minors[minor] = NULL;

mutex\_unlock(&sound\_mutex);

minor = PTR\_ERR(preg->dev);

kfree(preg);

return minor;

}

mutex\_unlock(&sound\_mutex);

return 0;

}

先，分配并初始化一个snd\_minor结构中的各字段

type：SNDRV\_DEVICE\_TYPE\_PCM\_PLAYBACK/SNDRV\_DEVICE\_TYPE\_PCM\_CAPTURE

card: card的编号

device：pcm实例的编号，大多数情况为0

f\_ops：snd\_pcm\_f\_ops

private\_data：指向该pcm的实例

根据type，card和pcm的编号，确定数组的索引值minor，minor也作为pcm设备的此设备号

把该snd\_minor结构的地址放入全局数组snd\_minors[minor]中

最后，调用device\_create创建设备节点

### 设备文件的建立

在4.1节的最后，设备文件已经建立，不过4.1节的重点在于snd\_minors数组的赋值过程，在本节中，我们把重点放在设备文件中。

回到pcm的回调函数snd\_pcm\_dev\_register()中：

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

{

int cidx, err;

char str[16];

struct snd\_pcm \*pcm;

struct device \*dev;

pcm = device->device\_data;

......

for (cidx = 0; cidx < 2; cidx++) {

......

switch (cidx) {

case SNDRV\_PCM\_STREAM\_PLAYBACK:

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

devtype = SNDRV\_DEVICE\_TYPE\_PCM\_PLAYBACK;

break;

case SNDRV\_PCM\_STREAM\_CAPTURE:

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

devtype = SNDRV\_DEVICE\_TYPE\_PCM\_CAPTURE;

break;

}

/\* device pointer to use, pcm->dev takes precedence if

\* it is assigned, otherwise fall back to card's device

\* if possible \*/

dev = pcm->dev;

if (!dev)

dev = snd\_card\_get\_device\_link(pcm->card);

/\* register pcm \*/

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

pcm->device,

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

pcm, str, dev);

......

}

......

}

以上代码我们可以看出，对于一个pcm设备，可以生成两个设备文件，一个用于playback，一个用于capture，代码中也确定了他们的命名规则：

playback -- pcmCxDxp，通常系统中只有一各声卡和一个pcm，它就是pcmC0D0p

capture -- pcmCxDxc，通常系统中只有一各声卡和一个pcm，它就是pcmC0D0c

#### snd\_pcm\_f\_ops

snd\_pcm\_f\_ops是一个标准的文件系统file\_operations结构数组，它的定义在sound/core/pcm\_native.c中：

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,

.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作为snd\_register\_device\_for\_dev的参数被传入，并被记录在snd\_minors[minor]中的字段f\_ops中。最后，在snd\_register\_device\_for\_dev中创建设备节点：

snd\_minors[minor] = preg;

preg->dev = device\_create(sound\_class, device, MKDEV(major, minor),

private\_data, "%s", name);

### 层层深入，从应用程序到驱动层pcm

#### 字符设备注册

在sound/core/sound.c中有alsa\_sound\_init()函数，定义如下：

static int \_\_init alsa\_sound\_init(void)

{

snd\_major = major;

snd\_ecards\_limit = cards\_limit;

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

snd\_printk(KERN\_ERR "unable to register native major device number %d/n", major);

return -EIO;

}

if (snd\_info\_init() < 0) {

unregister\_chrdev(major, "alsa");

return -ENOMEM;

}

snd\_info\_minor\_register();

return 0;

}

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，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，我们将在下一节中讲述open的过程。

#### 打开pcm设备

从上一节中我们得知，open一个pcm设备时，将会调用snd\_fops的open回调函数，我们先看看snd\_fops的定义：

static const struct file\_operations snd\_fops =

{

.owner = THIS\_MODULE,

.open = snd\_open

};

跟入snd\_open函数，它首先从inode中取出此设备号，然后以次设备号为索引，从snd\_minors全局数组中取出当初注册pcm设备时 填充的snd\_minor结构（参看4.1节的内容），然后从snd\_minor结构中取出pcm设备的f\_ops，并且把file->f\_op替 换为pcm设备的f\_ops，紧接着直接调用pcm设备的f\_ops->open()，然后返回。因为file->f\_op已经被替换，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4.2节中提到的snd\_pcm\_f\_ops结构中 定义的回调。

static int snd\_open(struct inode \*inode, struct file \*file)

{

unsigned int minor = iminor(inode);

struct snd\_minor \*mptr = NULL;

const struct file\_operations \*old\_fops;

int err = 0;

if (minor >= ARRAY\_SIZE(snd\_minors))

return -ENODEV;

mutex\_lock(&sound\_mutex);

mptr = snd\_minors[minor];

if (mptr == NULL) {

mptr = autoload\_device(minor);

if (!mptr) {

mutex\_unlock(&sound\_mutex);

return -ENODEV;

}

}

old\_fops = file->f\_op;

file->f\_op = fops\_get(mptr->f\_ops);

if (file->f\_op == NULL) {

file->f\_op = old\_fops;

err = -ENODEV;

}

mutex\_unlock(&sound\_mutex);

if (err < 0)

return err;

if (file->f\_op->open) {

err = file->f\_op->open(inode, file);

if (err) {

fops\_put(file->f\_op);

file->f\_op = fops\_get(old\_fops);

}

}

fops\_put(old\_fops);

return err;

}

下面的序列图展示了应用程序如何最终调用到snd\_pcm\_f\_ops结构中的回调函数：

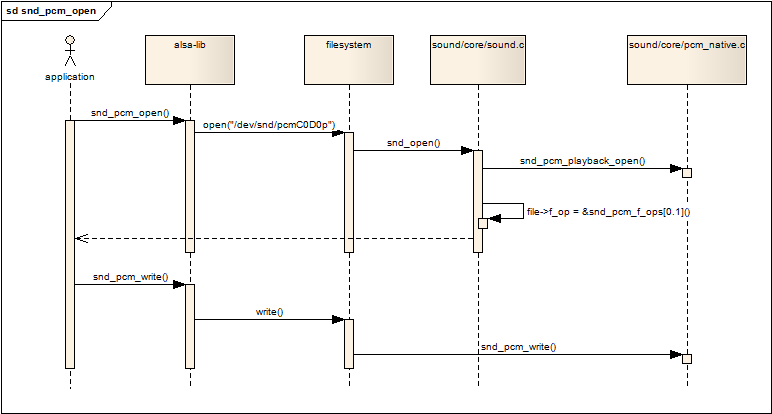


图4.3.2.1 应用程序操作pcm设备