# 音频设备驱动

音频设备广泛应用在嵌入式产品中，目前PDA、手机都有音频和视频播放功能。本小节将介绍音频设备接口分类以及Linux内核音频设备驱动程序实现。

## 1概述

计算机内部只能存储和处理数字信号，因此录音时模拟的话音信号需要经过采样、量化（脉冲编码调制PCM），转换成数字信号（AD转换）后才能供计算机存储和处理。放音时需要将计算机存储的数字音频数据经数模转换（DA转换）成模拟信号用于驱动放音设备（耳机、喇叭等），如下图所示。



根据采样定理，为了在DA转换时恢复原信号，采样频率（1/T）需大于信号最大频率的2倍。人耳可以听到的声音频率范围是20Hz~20KHz，因此通常将采样频率设为48KHz（大于2×20KHz）。

音频设备的主要工作是在录音时对话筒输入的模拟语音信号进行采样和量化，将量化后的数字信号提交给CPU处理；在放音时，将CPU发送的数字信号进行数模转换，变成模拟信号用于驱动放音设备。

### 1.1音频设备接口

处理器与音频设备接口如下图所示，处理器通过PCM、IIS、AC97等接口与外部音频编解码器相连。编解码器主要负责录音时对话筒信号进行AD转换，放音时对输出音频信号进行DA转换用于驱动放音设备。

处理器可以对原始的音频编码数据进行压缩保存至文件内，放音时对压缩的数据进行解压，通过接口输出到外部编解码器。



Linux内核支持的音频设备接口包括PCM、IIS和AC97。这三类接口分别用于不同的场合，PCM接口多应用在移动电话中，IIS接口多应用在MP3随身听、CD等产品中，AC97接口多用在PDA中。

●**PCM接口**：PCM（脉冲编码调制）接口针对不同的数字音频子系统，用于数字转换的接口，它是最简单的音频接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在FS信号的上升沿，数据传输从MSB位开始，按顺序进行单个数据位的传输，每个时钟周期传输1个数据字，FS频率等于采样频率。

PCM接口的特点是容易实现，原则上能够支持任何数据方案和任何采样频率，但需要每个音频通道获得一个独立的数据队列。

●**IIS接口**：IIS（Inter-IC Sound）接口是20世纪80年代首先被飞利浦用于消费音频，并在一个称为LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号合成单一的数据队列。当LRCLK信号为高时，左声道数据被传输，LRCLK信号为低时，右声道数据被传输。

IIS接口的特点：与PCM相比，IIS接口更适用于立体声系统。对于多通道系统，也可以并行执行几个数据队列。

●**AC97接口**：AC97（Audio Code 1997）是以Intel为首的5个PC厂商共同提出的规格标准。与PCM和IIS不同，AC97除了是一种数据格式外，还具有控制能力，用于音频编码的内部架构规范。AC97采用AC-Link接口与外部的编解码器相连，AC-Link接口包括位时钟BITCLK、同步信号SYNC、以及数据输入输出的SDATAIN和SDATAOUT接口（串行）。

AC97的特点：与控制接口分离的IIS方案相比，AC97明显减少了整体的管脚数。它是一种数据格式，还具有控制功能。

本节将讲解Linux内核音频设备驱动框架，并介绍AC97接口声卡驱动程序的实现。

### 1.2驱动框架

Linux内核具有两类音频设备驱动框架：OSS（Open Sound System）和ALSA（Advanced Linux Sound Architecture）。前者包含/dev/dsp和/dev/mixer字符设备接口，在用户空间编程中，使用文件操作（老式接口）。后者以card（声卡）和组件（pcm、mixer等）为主线，在用户空间的编程中不使用文件接口而使用alsalib（函数库）。ALSA框架同时支持OSS接口（需选择相应的配置选项），本书主要介绍ALSA驱动架框的实现，ALSA项目网址：<http://www.alsa-project.org/>。ALSA驱动框架主要代码位于/sound/core/目录下。

ALSA结构如下图所示：



用户程序通过库函数访问音频设备，库函数通过系统调用进入内核，而后调用ALSA提供的内核API访问音频设备（通过音频设备文件访问）。

ALSA驱动架框如下图所示：



ALSA驱动框架中声卡由snd\_card结构体表示，每个声卡下有多个设备（接口、组件），例如：设备控制接口、数据流接口等，声卡下设备由snd\_device结构体表示。声卡下最重要的接口就是实现数据流的PCM设备，它负责音频数据流的输入和输出。

内核在初始化时创建了声卡设备通用的字符设备驱动框架，声卡主设备号为SND\_MAJOR。每个声卡下的设备（接口）对应一个设备文件，在驱动框架中由snd\_minor结构体表示，结构体中包含设备文件操作结构实例，用户通过此实例操作声卡接口。

具体声卡驱动程序需要创建并注册snd\_card实例，以及创建并注册声卡下设备的snd\_device实例。

### 1.3音频设备类

内核在初始化阶段将调用init\_soundcore()函数为音频设备创建设备类，函数定义如下：

static int \_\_init init\_soundcore(void) /\*/sound/sound\_core.c\*/

{

int rc;

rc = init\_oss\_soundcore(); /\*没有选择SOUND\_OSS\_CORE配置项为空操作\*/

...

**sound\_class** = class\_create(THIS\_MODULE, "sound"); /\*创建音频设备类\*/

...

sound\_class->devnode = **sound\_devnode**; /\*获取设备节点名称（设备文件名称）函数\*/

return 0;

}

subsys\_initcall(init\_soundcore);

在注册sound\_class设备类中设备device实例创建设备节点（设备文件）时，将调用sound\_devnode()函数获取设备文件名称，函数定义如下：

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

{

if (MAJOR(dev->devt) == SOUND\_MAJOR) /\*适用于OSS框架\*/

return NULL;

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

}

由函数可知设备文件将在/dev/snd/目录下创建，请读者记住这点。

## 2声卡

在ALSA框架中声卡由snd\_card结构体实例表示，本小节先介绍snd\_card结构体的定义，然后介绍创建和注册结构体实例函数的实现。

### 2.1数据结构

snd\_card结构体表示系统中的声卡，结构体定义如下（/include/sound/core.h）：

struct snd\_card {

int number; /\*声卡编号\*/

char id[16]; /\*声卡ID，字符数组\*/

char **driver[16]**; /\*驱动名称\*/

char shortname[32]; /\*声卡短名\*/

char longname[80]; /\*声卡名称\*/

char mixername[80]; /\*混音器名称\*/

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

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

void \*private\_data; /\*声卡私有数据，不同类型声卡指向不同数据结构\*/

void (\*private\_free) (struct snd\_card \*card); /\*释放私有数据结构的回调函数\*/

**struct list\_head devices**; /\*声卡下设备（接口、组件）双链表头\*/

struct device **ctl\_dev**; /\*表示声卡控制接口的device实例，用于创建控制设备文件\*/

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; /\*所有控制接口数量\*/

int user\_ctl\_count; /\*用户控制接口数量\*/

**struct list\_head controls**; /\*控制设备（接口）双链表\*/

struct list\_head ctl\_files; /\*活跃的控制设备文件链表\*/

struct mutex user\_ctl\_lock;

struct snd\_info\_entry \*proc\_root; /\*声卡特殊文件proc根目录\*/

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;** /\*声卡下设备文件链表，链接snd\_monitor\_file实例\*/

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

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

int shutdown; /\*声卡正在关闭\*/

struct completion \*release\_completion;

struct device \*dev; /\*指向父设备device实例\*/

struct device **card\_dev**;

/\*表示声卡的device实例，添加到通用驱动模型，是其下音频设备的父设备\*/

const struct attribute\_group \*dev\_groups[4]; /\*设备属性，赋予card\_dev成员\*/

bool registered; /\* card\_dev是否已注册\*/

#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; /\*支持OSS框架\*/

int mixer\_oss\_change\_count;

#endif

};

内核在/sound/core/init.c文件内定义了全局的指针数组用于管理注册的snd\_card实例：

struct snd\_card \***snd\_cards**[SNDRV\_CARDS];

数组项数SNDRV\_CARDS定义在/include/sound/core.h头文件，取值为8（SND\_MAX\_CARDS配置选项未设置时）。

### 2.2创建声卡

创建并初始化声卡的函数为**snd\_card\_new()**，函数定义如下（/sound/core/init.c）：

int snd\_card\_new(struct device \*parent, int idx, const char \*xid,

struct module \*module, int extra\_size,struct snd\_card \*\***card\_re**t)

/\*parent：父设备，idx：关联snd\_cards[]数组项索引值，xid：识别字符串，extra\_size：额外字节数\*/

{

struct snd\_card \*card;

int err;

...

\*card\_ret = NULL;

if (extra\_size < 0)

extra\_size = 0;

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

/\*为实例分配空间，额外字节数为私有数据\*/

...

if (extra\_size > 0)

card->**private\_data** = (char \*)card + sizeof(struct snd\_card); /\*额外字节即私有数据结构\*/

if (xid)

strlcpy(card->id, xid, sizeof(card->id)); /\*xid指向字符串复制到snd\_card->id[]成员\*/

err = 0;

mutex\_lock(&snd\_card\_mutex);

if (idx < 0) /\* first check the matching module-name slot \*/

idx = get\_slot\_from\_bitmask(idx, module\_slot\_match, module);

if (idx < 0) /\* if not matched, assign an empty slot \*/

idx = get\_slot\_from\_bitmask(idx, check\_empty\_slot, module);

if (idx < 0)

err = -ENODEV;

else if (idx < snd\_ecards\_limit) {

if (test\_bit(idx, snd\_cards\_lock))

err = -EBUSY; /\* invalid \*/

} else if (idx >= SNDRV\_CARDS)

err = -ENODEV;

if (err < 0) {

...

}

set\_bit(idx, snd\_cards\_lock); /\* lock it \*/

if (idx >= snd\_ecards\_limit)

snd\_ecards\_limit = idx + 1; /\* increase the limit \*/

mutex\_unlock(&snd\_card\_mutex);

**card->dev = parent**; /\*父设备\*/

card->number = idx;

card->module = module;

INIT\_LIST\_HEAD(&card->devices);

init\_rwsem(&card->controls\_rwsem);

rwlock\_init(&card->ctl\_files\_rwlock);

mutex\_init(&card->user\_ctl\_lock);

INIT\_LIST\_HEAD(&card->controls);

INIT\_LIST\_HEAD(&card->ctl\_files);

spin\_lock\_init(&card->files\_lock);

INIT\_LIST\_HEAD(&card->files\_list);

#ifdef CONFIG\_PM

mutex\_init(&card->power\_lock);

init\_waitqueue\_head(&card->power\_sleep);

#endif

**device\_initialize(&card->card\_dev)**; /\*初始化表示声卡的device实例\*/

card->card\_dev.parent = parent; /\*父设备\*/

card->card\_dev.class = **sound\_class**; /\*设备类，在初始化函数init\_soundcore()中创建\*/

card->card\_dev.release = release\_card\_device;

card->card\_dev.groups = card->dev\_groups; /\*设备属性\*/

card->dev\_groups[0] = &card\_dev\_attr\_group;

**err = kobject\_set\_name(&card->card\_dev.kobj, "card%d", idx)**; /\*设置名称\*/

...

err = **snd\_ctl\_create(card)**; /\*创建控制设备，见9.5.5小节，/sound/core/control.c\*/

...

err = **snd\_info\_card\_create(card)**; /\*在procfs中创建文件，/sound/core/info.c\*/

...

**\*card\_ret** = card; /\*指向新创建实例\*/

return 0;

...

}

snd\_card\_new()函数主要是为snd\_card实例及其私有数据结构分配内存空间，初始化实例，并为声卡创建控制设备（接口）等。

### 2.3注册声卡

声卡（接口控制器）设备驱动程序在创建声卡snd\_card实例后，还需要调用后面介绍的创建PCM设备的接口函数为声卡注册音频设备（组件、接口），最后调用**snd\_card\_register()**函数向ALSA驱动框架注册声卡snd\_card实例。

注册声卡snd\_card实例的函数snd\_card\_register()定义如下（/sound/core/init.c）：

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

{

int err;

...

if (!card->registered) { /\*如果声卡没有注册过\*/

err = **device\_add(&card->card\_dev)**; /\*添加表示声卡的device实例，不创建设备文件\*/

if (err < 0)

return err;

card->registered = true;

}

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

/\*注册声卡下所有设备（card->devices），见下文，/sound/core/device.c\*/

return err;

mutex\_lock(&snd\_card\_mutex);

if (snd\_cards[card->number]) { /\*如果声卡已经关联到snd\_cards[]指针数组\*/

mutex\_unlock(&snd\_card\_mutex);

return snd\_info\_card\_register(card); /\* register pending info \*/

}

if (\*card->id) { /\*设置名称\*/

/\* make a unique id name from the given string \*/

char tmpid[sizeof(card->id)];

memcpy(tmpid, card->id, sizeof(card->id));

snd\_card\_set\_id\_no\_lock(card, tmpid, tmpid);

} else {

/\* create an id from either shortname or longname \*/

const char \*src;

src = \*card->shortname ? card->shortname : card->longname;

snd\_card\_set\_id\_no\_lock(card, src,retrieve\_id\_from\_card\_name(src));

}

**snd\_cards[card->number] = card**; /\*关联指针数组\*/

mutex\_unlock(&snd\_card\_mutex);

**init\_info\_for\_card(card)**; /\*初始化procfs中信息，/sound/core/init.c\*/

#if IS\_ENABLED(CONFIG\_SND\_MIXER\_OSS)

if (snd\_mixer\_oss\_notify\_callback)

snd\_mixer\_oss\_notify\_callback(card, SND\_MIXER\_OSS\_NOTIFY\_REGISTER);

#endif

return 0;

}

snd\_card\_register()函数的主要工作是向通用驱动模型中添加表示声卡的device实例，注册声卡下的音频设备（接口），以及将实例关联到snd\_cards[]指针数组项等。

声卡下的音频设备由snd\_device结构体表示，snd\_device\_register\_all(card)函数用于注册声卡下设备，详见下一小节。

## 3通用音频设备

一个声卡可能包含多个音频设备。音频设备是声卡下的逻辑设备，可理解成声卡的功能组件（或接口）。音频设备包括PCM设备、控制设备、原始MIDI设备等类型。所有音频设备在驱动框架中由snd\_device结构体表示，通过结构体中的私有数据结构指针区分不同类型的设备。

本小节介绍通用snd\_device结构体的定义、创建和注册，后面小节将介绍具体音频设备的私有数据结构，以及具体音频设备的创建和注册。

### 3.1数据结构

声卡下设备（组件或接口）由snd\_device结构体表示，定义如下（/include/sound/core.h）：

struct snd\_device {

struct list\_head list; /\*双链表成员，将实例添加到snd\_card实例设备双链表中\*/

struct snd\_card \*card; /\*指向snd\_card实例\*/

enum snd\_device\_state state; /\*设备状态\*/

enum snd\_device\_type type; /\*设备类型\*/

void \***device\_data**; /\*指向设备私有数据结构，用于区分不同类型的设备\*/

**struct snd\_device\_ops \*ops**; /\*音频设备操作结构\*/

};

snd\_device结构体成员简介如下：

●**list**：双链表成员，将实例插入到snd\_card实例设备双链表中。

●**type**：音频设备类型，取值定义如下：

enum snd\_device\_type {

SNDRV\_DEV\_LOWLEVEL,

SNDRV\_DEV\_**CONTROL**, /\*控制设备\*/

SNDRV\_DEV\_INFO, /\*声卡信息接口\*/

SNDRV\_DEV\_BUS,

SNDRV\_DEV\_**CODEC**, /\*编解码器\*/

SNDRV\_DEV\_**PCM**, /\*PCM设备\*/

SNDRV\_DEV\_COMPRESS,

SNDRV\_DEV\_RAWMIDI, /\*原始MIDI设备\*/

SNDRV\_DEV\_TIMER, /\*定时器\*/

SNDRV\_DEV\_SEQUENCER, /\*声音合成设备\*/

SNDRV\_DEV\_HWDEP,

SNDRV\_DEV\_JACK,

};

**●device\_data**：指向具体设备私有数据结构，用于区分不同类型的音频设备。

●**state**：设备状态，取值定义如下：

enum snd\_device\_state {

SNDRV\_DEV\_BUILD,

SNDRV\_DEV\_REGISTERED, /\*已注册\*/

SNDRV\_DEV\_DISCONNECTED, /\*已断开连接\*/

};

**●ops：**指向音频设备操作结构snd\_device\_ops，结构体定义如下：

struct snd\_device\_ops {

int (\*dev\_free)(struct snd\_device \*dev); /\*释放音频设备\*/

int (\***dev\_register**)(struct snd\_device \*dev); /\*注册音频设备，创建音频设备私有数据结构等\*/

int (\*dev\_disconnect)(struct snd\_device \*dev); /\*与声卡断开连接操作函数\*/

};

音频设备与声卡数据结构组织关系如下图所示：



声卡驱动程序在创建声卡snd\_card实例后，需要为声卡创建音频设备，最后才能注册声卡，在注册声卡的操作中将会注册声卡下音频设备。下文中将介绍创建、注册音频设备的接口函数。

### 3.2音频设备管理

在创建音频设备前需要实现音频设备操作结构snd\_device\_ops实例，在注册音频设备的操作中将调用其中实例的dev\_register()函数，函数内主要完成音频设备私有数据结构实例的创建和初始化。

#### ■创建设备

为声卡创建音频设备的函数**snd\_device\_new()**定义如下（/sound/core/device.c）：

int snd\_device\_new(struct snd\_card \*card, enum snd\_device\_type type, \

void \***device\_data**, struct snd\_device\_ops \*ops)

/\*card：snd\_card实例，type：设备类型，device\_data：设备数据结构，op：音频设备操作结构\*/

{

struct snd\_device \*dev;

struct list\_head \*p;

if (snd\_BUG\_ON(!card || !device\_data || !ops))

return -ENXIO;

dev = **kzalloc**(sizeof(\*dev), GFP\_KERNEL); /\*分配snd\_device实例\*/

if (!dev)

return -ENOMEM;

INIT\_LIST\_HEAD(&dev->list); /\*初始化实例\*/

dev->card = card;

dev->type = type;

dev->state = **SNDRV\_DEV\_BUILD**; /\*初始状态\*/

dev->device\_data = **device\_data**; /\*私有数据结构\*/

dev->ops = **ops**; /\*设备操作结构\*/

/\*将实例按类型递增顺序插入声卡设备链表\*/

list\_for\_each\_prev(p, &**card->devices**) {

struct snd\_device \*pdev = list\_entry(p, struct snd\_device, list);

if ((unsigned int)pdev->type <= (unsigned int)type)

break;

}

**list\_add(&dev->list, p)**; /\*将snd\_device实例插入card->devices双链表头部\*/

return 0;

}

#### ■注册设备

**snd\_device\_register()**函数用于注册音频设备，通常在注册声卡实例的snd\_card\_register()函数中被调用，函数定义如下（/sound/core/device.c）：

int snd\_device\_register(struct snd\_card \*card, void \*device\_data)

{

struct snd\_device \*dev;

if (snd\_BUG\_ON(!card || !device\_data))

return -ENXIO;

dev = look\_for\_dev(card, device\_data); /\*通过dev->device\_data查找关联的snd\_device实例\*/

if (dev)

return **\_\_snd\_device\_register(dev)**; /\*注册查找到的snd\_device实例\*/

snd\_BUG();

return -ENXIO;

}

注册函数通过device\_data指向实例在card->devices双链表中查找snd\_device实例，查找到则注册实例。注册操作由\_\_snd\_device\_register(dev)函数完成，函数定义如下：

static int \_\_snd\_device\_register(struct snd\_device \*dev)

{

if (dev->state == SNDRV\_DEV\_BUILD) { /\*如果设备为初始状态\*/

if (dev->ops->dev\_register) {

int err = **dev->ops->dev\_register(dev)**; /\*调用音频设备操作结构中的注册函数\*/

if (err < 0)

return err;

}

dev->state = SNDRV\_DEV\_REGISTERED; /\*设备已经注册\*/

}

return 0;

}

在前面介绍的注册声卡的snd\_card\_register(card)函数中将调用**snd\_device\_register\_all(card)**函数，注册card->devices双链表下的所有设备，注册操作与snd\_device\_register()函数相同。音频设备驱动程序在注册声卡前需要为声卡创建音频设备，然后才能调用snd\_card\_register(card)函数注册声卡。

### 3.3设备文件

ALSA框架提供了库函数，用户可通过库函数对声卡进行操作。库函数内部是通过设备文件对音频设备进行操作的。声卡下所有音频设备主设备号为CONFIG\_SND\_MAJOR（116，/include/sound/core.h），从设备号数量为SNDRV\_OS\_MINORS（256），定义在/include/sound/minors.h头文件。在表示音频设备的字符设备驱动框架中，音频设备由snd\_minor结构体表示，结构体中包括音频设备文件操作结构实例指针，文件层通过此实例操作音频设备文件。

#### ■注册从设备

在音频设备的字符设备驱动框架中，每个音频设备（从设备）由snd\_minor结构体表示，结构体定义如下（/include/sound/core.h）：

struct snd\_minor {

int type; /\*设备类型，SNDRV\_DEVICE\_TYPE\_XXX，/include/sound/minors.h \*/

int card; /\*声卡编号\*/

int device; /\*设备编号\*/

**const struct file\_operations \*f\_ops;** /\*设备文件操作结构\*/

void \*private\_data; /\*打开设备文件时，指向snd\_device实例私有数据结构\*/

**struct device \*dev**; /\*指向表示音频设备的device实例\*/

struct snd\_card \*card\_ptr; /\*声卡实例指针\*/

};

内核通过snd\_minor结构体指针数组snd\_minors[SNDRV\_OS\_MINORS]管理注册的snd\_minor实例。

内核对每种类型的音频设备需要实现对应的file\_operations实例，在音频设备操作结构体中的注册函数snd\_device**->**ops->dev\_register(dev)内需要创建并注册表示音频设备的snd\_minor实例。

ALSA框架提供了创建并注册音频设备snd\_minor实例的接口函数**snd\_register\_device()**，函数定义如下（/sound/core/sound.c）：

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

const struct file\_operations \***f\_ops**,void \***private\_data**, struct device \***device**)

/\*type：设备类型，dev：设备编号，**f\_ops**：音频设备文件file\_operations实例，

\***private\_data**：设备私有数据结构，device：表示音频设备的device实例指针。\*/

{

int minor;

int err = 0;

struct snd\_minor \*preg;

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

return -EINVAL;

**preg = kmalloc(sizeof \*preg, GFP\_KERNEL)**; /\*分配snd\_minor实例\*/

if (preg == NULL)

return -ENOMEM;

preg->type = type;

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

preg->device = dev;

**preg->f\_ops = f\_ops**; /\*文件操作结构实例\*/

**preg->private\_data = private\_data**; /\*私有数据\*/

**preg->card\_ptr = card**; /\*指向声卡实例\*/

mutex\_lock(&sound\_mutex);

minor = **snd\_find\_free\_minor(type, card, dev)**; /\*生成从设备号，/sound/core/sound.c\*/

if (minor < 0) {

...

}

preg->dev = device; /\*device实例\*/

device->devt = **MKDEV(major, minor)**; /\*合成设备号，major为音频设备主设备号（116）\*/

err = **device\_add(device)**; /\*添加device实例，在/dev/snd/下创建设备文件等\*/

if (err < 0)

goto error;

**snd\_minors[minor] = preg**; /\*关联指针数组\*/

error:

mutex\_unlock(&sound\_mutex);

if (err < 0)

kfree(preg);

return err;

}

这里需要注意的是snd\_minors[minor]指针数组按照音频设备的从设备号管理snd\_minor实例。所有音频设备的主设备号是相同的，从设备号根据所在声卡编号、音频设备类型等由**snd\_find\_free\_minor()**函数生成，源代码请读者自行阅读。

创建并注册音频设备snd\_minor实例的执行结果如下图所示：



#### ■字符设备驱动

内核在/sound/core/sound.c文件的初始化函数中注册了表示所有音频设备的cdev实例，函数定义如下：

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

{

snd\_major = major; /\*声卡设备主设备号\*/

snd\_ecards\_limit = cards\_limit;

if (**register\_chrdev(major, "alsa", &snd\_fops)**) { /\*注册cdev实例\*/

...

}

if (snd\_info\_init() < 0) { /\*在procfs中创建目录项，/sound/core/info.c\*/

...

}

...

return 0;

}

subsys\_initcall(alsa\_sound\_init);

初始化函数中注册了适用音频设备的cdev实例，实例file\_operations结构体指针成员指向**snd\_fops**实例，实例定义如下：

static const struct file\_operations snd\_fops =

{

.owner = THIS\_MODULE,

.open = **snd\_open**, /\*打开设备文件\*/

.llseek = noop\_llseek,

};

打开设备文件snd\_open()函数通过从设备号查找snd\_minor结构体数组snd\_minors[minor]，找到特定于具体设备的file\_operations实例并赋予file实例，如上图中所示。

音频设备文件打开操作snd\_open()函数定义如下（/sound/core/sound.c）：

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

{

unsigned int minor = **iminor(inode)**; /\*从设备号\*/

struct snd\_minor \*mptr = NULL;

const struct file\_operations \*new\_fops;

int err = 0;

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

return -ENODEV;

mutex\_lock(&sound\_mutex);

**mptr = snd\_minors[minor]**; /\*指针数组项\*/

if (mptr == NULL) {

...

}

new\_fops = **fops\_get(mptr->f\_ops)**; /\*snd\_minor->f\_ops\*/

mutex\_unlock(&sound\_mutex);

if (!new\_fops)

return -ENODEV;

**replace\_fops(file, new\_fops);**

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

err = **file->f\_op->open(inode, file)**; /\*调用snd\_minor->f\_ops->open()函数\*/

return err;

}

snd\_open()函数以从设备号为索引值查找snd\_minors[]数组，查找对应的snd\_minor实例，并其将f\_ops成员赋予file->f\_op成员，最后调用file->f\_op->open(inode, file)函数执行具体音频设备定义的打开操作函数。

## 4 PCM设备

PCM全称为脉冲编码调制，在音频处理里面就是对模拟音频信号进行脉冲采样，再将采样值进行量化编码以便于CPU处理和存储，采样率表示1秒时间内的采样次数。录音时，声卡对模拟音频信号按照一定的采样率进行采样，每个采样点的音频数据为一帧（frame），再对模拟的采样值（如电压值等）用二进制数进行编码，编码的二进制bit位数称为精度，例如前面概述中所示的采样精度为4bit。一帧音频数据的二进制位数（bit位数）为精度×声道，一秒内音频数据的bit位数为精度×声道×采样率。一帧一帧的音频数据按照时间顺序串在一起组成了音频流，PCM数据实际就是原始的音频流数据。放音时，CPU将量化编码按采样周期进行输出，然后经由DA（模数）转换，变成模拟音频信号以驱动放音设备。

ALSA框架中，PCM设备用来管理音频流，包括输入流（采集）和输出流（播放），音频流由snd\_pcm结构体描述（snd\_device的私有数据）。一个声卡可以包含多个PCM设备，每个PCM设备包含采集或播放流，每个流又可包含多个子流。

### 4.1数据结构

前面介绍的snd\_device是表示所有音频设备的通用数据结构，插入到声卡snd\_card实例设备链表中，snd\_pcm结构体作为PCM音频流设备snd\_device实例的私有数据结构，snd\_device实例**device\_data**成员指向snd\_pcm结构体实例。

#### ■snd\_pcm

**snd\_pcm**结构体定义如下（/include/sound/pcm.h）：

struct snd\_pcm {

struct snd\_card \*card; /\*指向所属声卡实例\*/

struct list\_head **list**; /\*将实例插入全局双链表snd\_pcm\_devices，用于管理snd\_pcm实例\*/

int **device**; /\*设备编号\*/

unsigned int info\_flags;

unsigned short dev\_class;

unsigned short dev\_subclass;

char id[64]; /\*标识字符串\*/

char name[80];

**struct snd\_pcm\_str streams[2]**; /\*播放流和录音流\*/

struct mutex open\_mutex;

wait\_queue\_head\_t **open\_wait**; /\*等待队列头部\*/

void \*private\_data; /\*私有数据\*/

void (\*private\_free) (struct snd\_pcm \*pcm); /\*释放私有数据函数\*/

bool **internal**; /\* pcm只用于内部使用\*/

bool nonatomic; /\* whole PCM operations are in non-atomic context \*/

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

struct snd\_pcm\_oss oss;

#endif

};

snd\_pcm结构体中包含两个PCM流，由**snd\_pcm\_str**结构体数组表示，snd\_pcm\_str结构体中又可包含多个子流，子流由snd\_pcm\_substream结构体表示，各数据结构组织关系如下图所示：



为声卡snd\_card实例创建PCM设备的接口函数为**snd\_pcm\_new()**，函数内将创建snd\_pcm实例，并创建表示音频设备的通用snd\_device实例，详见下文。

#### ■snd\_pcm\_str

PCM设备数据流由snd\_pcm\_str结构体表示，结构体定义如下（/include/sound/pcm.h）：

struct snd\_pcm\_str {

int **stream**; /\*数据流方向，/include/uapi/sound/asound.h\*/

struct snd\_pcm \*pcm; /\*指向snd\_pcm实例\*/

/\*子流相关成员\*/

unsigned int substream\_count;

unsigned int substream\_opened;

**struct snd\_pcm\_substream \*substream**; /\*指向子流单链表\*/

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

struct snd\_pcm\_oss\_stream oss;

#endif

#ifdef CONFIG\_SND\_VERBOSE\_PROCFS

...

#endif

struct snd\_kcontrol \*chmap\_kctl; /\* channel-mapping controls \*/

**struct device dev**; /\*表示数据流的device实例，用于创建设备文件\*/

};

snd\_pcm\_str结构体主要成员简介如下：

●**stream：**数据流方向，相关宏定义在/include/uapi/sound/asound.h头文件内：

enum {

SNDRV\_PCM\_STREAM\_PLAYBACK = 0, /\*播音流\*/

SNDRV\_PCM\_STREAM\_CAPTURE, /\*录音流\*/

SNDRV\_PCM\_STREAM\_LAST = SNDRV\_PCM\_STREAM\_CAPTURE,

};

PCM设备中包含两个数据流，分别是输出流和输入流。

●**substream：**指向子流snd\_pcm\_substream实例，多个子流组成单链表，详见下文。

●**dev：**device实例，添加到通用驱动模型中，并创建设备文件，每个数据流对应一个设备文件。

PCM数据流下可包含多个子流，各子流组成单链表，子流由snd\_pcm\_substream结构体表示，定义简列如下：

struct snd\_pcm\_substream {

struct snd\_pcm \*pcm;

struct snd\_pcm\_str \*pstr;

void \*private\_data; /\*复制pcm->private\_data成员\*/

int number;

char name[32]; /\*数据流名称\*/

int stream; /\*数据流方向\*/

struct pm\_qos\_request latency\_pm\_qos\_req; /\* pm\_qos request \*/

size\_t buffer\_bytes\_max; /\* limit ring buffer size \*/

struct snd\_dma\_buffer dma\_buffer;

size\_t dma\_max;

**const struct snd\_pcm\_ops \*ops**; /\*硬件操作结构\*/

**struct snd\_pcm\_runtime \*runtime**; /\*运行时数据结构\*/

/\*定时器部分\*/

struct snd\_timer \*timer; /\*定时器\*/

unsigned timer\_running: 1; /\* time is running \*/

**struct snd\_pcm\_substream \*next**; /\*下一个子流\*/

struct list\_head link\_list; /\*双链表成员，链接的子流\*/

struct snd\_pcm\_group self\_group; /\* fake group for non linked substream \*/

struct snd\_pcm\_group \*group; /\* pointer to current group \*/

/\*对应文件信息\*/

**void \*file**;

int ref\_count;

atomic\_t mmap\_count;

unsigned int f\_flags;

void (\*pcm\_release)(struct snd\_pcm\_substream \*);

struct pid \*pid;

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

/\* -- OSS things -- \*/

struct snd\_pcm\_oss\_substream oss;

#endif

#ifdef CONFIG\_SND\_VERBOSE\_PROCFS

...

#endif

/\* misc flags \*/

unsigned int hw\_opened: 1;

};

snd\_pcm\_substream结构体中比较重要的成员是指向snd\_pcm\_ops结构体的ops成员，以及指向结构体snd\_pcm\_runtime的runtime成员。snd\_pcm\_ops结构体包含硬件操作的函数指针，是特定于设备的数据结构，snd\_pcm\_runtime结构体包含数据流运行时的信息，详见下文。

#### ■**snd\_pcm\_ops**

**snd\_pcm\_ops**结构体包含特定于设备的数据子流操作函数指针，结构体定义如下：

struct snd\_pcm\_ops {

int (\*open)(struct snd\_pcm\_substream \*substream);

int (\*close)(struct snd\_pcm\_substream \*substream);

int (\*ioctl)(struct snd\_pcm\_substream \* substream,unsigned int cmd, void \*arg);

int (\*hw\_params)(struct snd\_pcm\_substream \*substream,struct snd\_pcm\_hw\_params \*params);

int (\*hw\_free)(struct snd\_pcm\_substream \*substream);

int (\*prepare)(struct snd\_pcm\_substream \*substream);

int (\*trigger)(struct snd\_pcm\_substream \*substream, int cmd);

snd\_pcm\_uframes\_t (\*pointer)(struct snd\_pcm\_substream \*substream);

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

struct timespec \*system\_ts, struct timespec \*audio\_ts,

struct snd\_pcm\_audio\_tstamp\_config \*audio\_tstamp\_config,

struct snd\_pcm\_audio\_tstamp\_report \*audio\_tstamp\_report);

int (\***copy**)(struct snd\_pcm\_substream \*substream, int channel,

snd\_pcm\_uframes\_t pos,void \_\_user \*buf, snd\_pcm\_uframes\_t count); /\*读与数据流\*/

int (\*silence)(struct snd\_pcm\_substream \*substream, int channel,

snd\_pcm\_uframes\_t pos, snd\_pcm\_uframes\_t count);

struct page \*(\*page)(struct snd\_pcm\_substream \*substream,unsigned long offset);

int (\*mmap)(struct snd\_pcm\_substream \*substream, struct vm\_area\_struct \*vma);

int (\*ack)(struct snd\_pcm\_substream \*substream);

};

声卡设备驱动程序需要实现snd\_pcm\_ops结构体实例，并在创建snd\_pcm实例后（同时创建snd\_device实例），调用**snd\_pcm\_set\_ops()**函数（/sound/core/pcm\_lib.c）设置数据流（子流）的snd\_pcm\_ops指针成员ops。snd\_pcm\_set\_ops()函数定义如下（/sound/core/pcm\_lib.c）：

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

/\*direction：数据流方向，ops：snd\_pcm\_ops实例指针\*/

{

struct snd\_pcm\_str \*stream = &pcm->streams[direction];

struct snd\_pcm\_substream \*substream;

for (substream = stream->substream; substream != NULL; substream = substream->next)

**substream->ops = ops**; /\*遍历子流链表赋予snd\_pcm\_ops实例\*/

}

#### ■snd\_pcm\_runtime

子流运行时snd\_pcm\_runtime结构体表示子流运行时的信息，结构体定义如下（/include/sound/pcm.h），

struct snd\_pcm\_runtime {

/\*状态\*/

struct snd\_pcm\_substream \*trigger\_master;

struct timespec trigger\_tstamp; /\*触发时间戳\*/

bool trigger\_tstamp\_latched; /\* trigger timestamp latched in low-level driver/hardware \*/

int overrange;

snd\_pcm\_uframes\_t avail\_max;

snd\_pcm\_uframes\_t hw\_ptr\_base; /\*缓存区复位时位置\*/

snd\_pcm\_uframes\_t hw\_ptr\_interrupt; /\*缓存区中断时位置\*/

unsigned long hw\_ptr\_jiffies; /\* Time when hw\_ptr is updated \*/

unsigned long hw\_ptr\_buffer\_jiffies; /\* buffer time in jiffies \*/

snd\_pcm\_sframes\_t delay; /\* extra delay; typically FIFO size \*/

u64 hw\_ptr\_wrap; /\* offset for hw\_ptr due to boundary wrap-around \*/

/\*硬件参数\*/

snd\_pcm\_access\_t access; /\*访问模式\*/

snd\_pcm\_format\_t format; /\* SNDRV\_PCM\_FORMAT\_\* \*/

snd\_pcm\_subformat\_t subformat; /\*子格式\*/

unsigned int rate; /\*速率，单位Hz \*/

unsigned int channels; /\*通道数\*/

snd\_pcm\_uframes\_t period\_size; /\*周期大小\*/

unsigned int periods; /\*周期数\*/

snd\_pcm\_uframes\_t buffer\_size; /\*缓存区大小\*/

snd\_pcm\_uframes\_t min\_align; /\*对齐的最小块大小\*/

size\_t byte\_align; /\*字节对齐\*/

unsigned int frame\_bits; /\*每帧比特数\*/

unsigned int sample\_bits; /\*采样的比特数\*/

unsigned int info;

unsigned int rate\_num;

unsigned int rate\_den;

unsigned int no\_period\_wakeup: 1;

/\*软件参数\*/

int tstamp\_mode; /\*更新内存时间戳\*/

unsigned int period\_step;

snd\_pcm\_uframes\_t start\_threshold;

snd\_pcm\_uframes\_t stop\_threshold;

snd\_pcm\_uframes\_t silence\_threshold; /\* Silence filling happens when

noise is nearest than this \*/

snd\_pcm\_uframes\_t silence\_size; /\* Silence filling size \*/

snd\_pcm\_uframes\_t boundary; /\* pointers wrap point \*/

snd\_pcm\_uframes\_t silence\_start; /\* starting pointer to silence area \*/

snd\_pcm\_uframes\_t silence\_filled; /\* size filled with silence \*/

union snd\_pcm\_sync\_id sync; /\* hardware synchronization ID \*/

/\*内存映射\*/

struct snd\_pcm\_mmap\_status \*status;

struct snd\_pcm\_mmap\_control \*control;

/\*锁和调度\*/

snd\_pcm\_uframes\_t twake; /\* do transfer (!poll) wakeup if non-zero \*/

wait\_queue\_head\_t sleep; /\* poll sleep \*/

wait\_queue\_head\_t tsleep; /\* transfer sleep \*/

struct fasync\_struct \*fasync;

/\*私有数据\*/

void \*private\_data;

void (\*private\_free)(struct snd\_pcm\_runtime \*runtime);

/\*硬件描述\*/

struct snd\_pcm\_hardware hw; /\*/include/sound/pcm.h\*/

struct snd\_pcm\_hw\_constraints hw\_constraints; /\*/include/sound/pcm.h\*/

/\*中断回调函数\*/

void (\*transfer\_ack\_begin)(struct snd\_pcm\_substream \*substream);

void (\*transfer\_ack\_end)(struct snd\_pcm\_substream \*substream);

/\*定时器\*/

unsigned int timer\_resolution; /\* timer resolution \*/

int tstamp\_type; /\* timestamp type \*/

/\* DMA \*/

unsigned char \***dma\_area**; /\* DMA area \*/

dma\_addr\_t dma\_addr; /\* physical bus address (not accessible from main CPU) \*/

size\_t dma\_bytes; /\* size of DMA area \*/

struct snd\_dma\_buffer \*dma\_buffer\_p; /\* allocated buffer \*/

/\* -- audio timestamp config -- \*/

struct snd\_pcm\_audio\_tstamp\_config audio\_tstamp\_config;

struct snd\_pcm\_audio\_tstamp\_report audio\_tstamp\_report;

struct timespec driver\_tstamp;

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

/\* -- OSS things -- \*/

struct snd\_pcm\_oss\_runtime oss;

#endif

#ifdef CONFIG\_SND\_PCM\_XRUN\_DEBUG

struct snd\_pcm\_hwptr\_log \*hwptr\_log;

#endif

};

snd\_pcm\_runtime结构体实例在打开PCM设备文件时创建并初始化。

### 4.2创建PCM设备

声卡驱动程序需要为声卡snd\_card实例创建PCM设备snd\_pcm实例，接口函数为**snd\_pcm\_new()。**函数内将创建snd\_pcm实例，并为其创建子流，最后创建对应的snd\_device实例，snd\_pcm实例为其私有数据结构。

#### ■接口函数

为声卡创建PCM设备的接口函数**snd\_pcm\_new()**定义在/sound/core/pcm.c文件内，代码如下：

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

int playback\_count, int capture\_count, struct snd\_pcm \*\*rpcm)

/\*card：声卡，id：id字符串，device：设备索引值，playback\_count：playback（播放）子流数量，

\*capture\_count：capture（录音）子流数量，rpcm：保存snd\_pcm实例指针\*/

{

return **\_snd\_pcm\_new**(card, id, device, playback\_count, capture\_count,**false**, rpcm);

}

\_snd\_pcm\_new()函数定义如下：

static int \_snd\_pcm\_new(struct snd\_card \*card, const char \*id, int device,

int playback\_count, int capture\_count, bool internal,struct snd\_pcm \*\*rpcm)

{

struct snd\_pcm \*pcm;

int err;

static struct **snd\_device\_ops ops** = { /\*音频设备操作结构\*/

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

.dev\_register =**snd\_pcm\_dev\_register**, /\*注册设备时调用\*/

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

};

...

pcm **= kzalloc**(sizeof(\*pcm), GFP\_KERNEL); /\*分配snd\_pcm实例\*/

...

pcm->card = card; /\*初始化实例\*/

pcm->device = device;

pcm->internal = internal; /\*false\*/

mutex\_init(&pcm->open\_mutex);

init\_waitqueue\_head(&pcm->open\_wait);

INIT\_LIST\_HEAD(&pcm->list);

if (id)

strlcpy(pcm->id, id, sizeof(pcm->id));

/\*创建数据流\*/

if ((err = snd\_pcm\_new\_stream(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, playback\_count)) < 0) {

...

} /\*创建播放流\*/

if ((err = snd\_pcm\_new\_stream(pcm, SNDRV\_PCM\_STREAM\_CAPTURE, capture\_count)) < 0) {

...

} /\*创建录音流\*/

/\*创建表示PCM设备的snd\_device实例，device\_data指向snd\_pcm实例，ops指向ops实例\*/

if ((err = **snd\_device\_new**(card, SNDRV\_DEV\_PCM, **pcm**, **&ops**)) < 0) {

...

}

if (rpcm)

\*rpcm = pcm;

return 0;

}

\_snd\_pcm\_new()函数内首先创建snd\_pcm实例，然后为其创建播放和录音数据流，最后为PCM设备创建snd\_device实例。snd\_device实例的device\_data成员指向snd\_pcm实例，ops指向函数内定义的实例snd\_device\_ops，注意此实例中的**snd\_pcm\_dev\_register()**函数，将在注册设备时被调用。

#### ■创建数据流

snd\_pcm\_new\_stream()函数在接口函数**snd\_pcm\_new()**中被调用，用于为PCM设备创建PCM数据流实例，函数定义如下（/sound/core/pcm.c）：

int snd\_pcm\_new\_stream(struct snd\_pcm \*pcm, int stream, int substream\_count)

/\*stream：数据流方向，substream\_count：子流数量\*/

{

int idx, err;

struct snd\_pcm\_str \*pstr = **&pcm->streams[stream]**; /\*数据流实例\*/

struct snd\_pcm\_substream \*substream, \*prev;

#if IS\_ENABLED(CONFIG\_SND\_PCM\_OSS)

mutex\_init(&pstr->oss.setup\_mutex);

#endif

pstr->stream = stream;

pstr->pcm = pcm;

pstr->substream\_count = substream\_count;

if (!substream\_count)

return 0;

**snd\_device\_initialize(&pstr->dev, pcm->card)**;

/\*初始化表示设备的device实例，设置**sound\_class**设备类等，/sound/core/init.c\*/

pstr->dev.groups = pcm\_dev\_attr\_groups;

dev\_set\_name(&pstr->dev, **"pcmC%iD%i%c"**, pcm->card->number, pcm->device,

stream == SNDRV\_PCM\_STREAM\_PLAYBACK ? 'p' : 'c');

/\*设备名称（设备文件），声卡编号，设备编号，再加p或c字符\*/

if (!pcm->internal) {

err = snd\_pcm\_stream\_proc\_init(pstr);

...

}

prev = NULL;

for (idx = 0, prev = NULL; idx < substream\_count; idx++) {

substream = **kzalloc(sizeof(\*substream), GFP\_KERNEL)**; /\*分配子流数据结构实例\*/

if (!substream)

return -ENOMEM;

substream->pcm = pcm;

substream->pstr = pstr;

substream->number = idx;

substream->stream = stream;

sprintf(substream->name, "subdevice #%i", idx);

substream->buffer\_bytes\_max = UINT\_MAX;

if (prev == NULL)

pstr->substream = substream;

else

prev->next = substream; /\*子流实例组成单链表\*/

if (!pcm->internal) {

err = snd\_pcm\_substream\_proc\_init(substream);

if (err < 0) {

...

}

}

substream->group = &substream->self\_group;

spin\_lock\_init(&substream->self\_group.lock);

mutex\_init(&substream->self\_group.mutex);

INIT\_LIST\_HEAD(&substream->self\_group.substreams);

**list\_add\_tail(&substream->link\_list, &substream->self\_group.substreams);**

atomic\_set(&substream->mmap\_count, 0);

prev = substream;

}

return 0;

}

snd\_pcm\_new\_stream()函数初始化pcm->streams[stream]数组项成员，初始化表示此设备的device实例，注意名称的设置，这将用于设备文件名称，最后为其创建子流实例（链表）。

### 4.3注册PCM设备

在随后注册设备snd\_device实例（注册声卡时注册设备）时，在\_\_snd\_device\_register(dev)注册函数中将调用snd\_device->ops->dev\_register()函数，执行特定音频设备的注册操作。

对于PCM设备，即调用**snd\_pcm\_dev\_register()**函数，在此函数中将注册snd\_pcm实例，并为其下两个数据流注册snd\_minor实例以及创建设备文件等。

**snd\_pcm\_dev\_register()**函数定义如下（/sound/core/pcm.c）：

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

{

int cidx, err;

struct snd\_pcm\_substream \*substream;

struct snd\_pcm\_notify \*notify;

**struct snd\_pcm \*pcm**;

...

**pcm = device->device\_data**; /\*snd\_pcm实例\*/

...

mutex\_lock(&register\_mutex);

err = **snd\_pcm\_add(pcm)**; /\*将snd\_pcm实例添加到全局双链表snd\_pcm\_devices\*/

...

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

int devtype = -1;

if (pcm->streams[cidx].substream == NULL)

continue;

switch (cidx) {

case SNDRV\_PCM\_STREAM\_PLAYBACK: /\*数据流方向\*/

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

break;

case SNDRV\_PCM\_STREAM\_CAPTURE:

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

break;

}

/\*为数据流snd\_pcm\_str注册从设备，创建设备文件/dev/snd/**pcmCxDxp**\*/

err = **snd\_register\_device**(devtype, pcm->card, pcm->device,

&snd\_pcm\_f\_ops[cidx], pcm, &**pcm->streams[cidx].dev**); /\*/sound/core/sound.c\*/

...

for (substream = pcm->streams[cidx].substream; substream; substream = substream->next)

snd\_pcm\_timer\_init(substream); /\*初始化定时器\*/

}

list\_for\_each\_entry(notify, &snd\_pcm\_notify\_list, list)

notify->n\_register(pcm);

unlock:

mutex\_unlock(&register\_mutex);

return err;

}

snd\_pcm\_dev\_register()函数内将snd\_pcm实例插入全局双链表snd\_pcm\_devices，snd\_register\_device()函数（详见9.5.3小节）将为数据流创建并注册表示从设备的snd\_minor实例，设置文件操作结构实例，并添加表示从设备的device实例，创建设备文件，设备文件名称为/dev/snd/**pcmCxDxp**或**pcmCxDxc**，C字符后是声卡编号，D字符后是设备编号，p表示播放流，c表示录音流。

snd\_pcm\_dev\_register()函数执行结果如下图所示：



### 4.4设备文件操作

在前面介绍的注册snd\_pcm实例的函数中，创建的PCM设备数据流对应的snd\_minor实例关联的文件操作结构实例为snd\_pcm\_f\_ops[cidx]，这也是设备文件的文件操作结构实例。snd\_pcm\_f\_ops[cidx]实例定义如下（/sound/core/pcm\_native.c）：

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

{ /\*播音流文件操作结构实例\*/

.owner = THIS\_MODULE,

.write = **snd\_pcm\_write**, /\*播音操作\*/

.write\_iter = **snd\_pcm\_writev**,

.open = **snd\_pcm\_playback\_open**, /\*打开操作\*/

.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,

},

{ /\*录音流文件操作结构实例\*/

.owner = THIS\_MODULE,

.read = **snd\_pcm\_read**, /\*录音操作\*/

.read\_iter = **snd\_pcm\_readv**,

.open = **snd\_pcm\_capture\_open**, /\*打开操作\*/

.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,

}

};

下面简要介绍一下其中的打开函数和读/写操作函数。

#### ■打开函数

打开函数snd\_pcm\_playback\_open()/snd\_pcm\_capture\_open()将为file实例创建snd\_pcm\_file结构体实例，用于建立file与子流snd\_pcm\_substream实例之间的关联，为子流创建并初始化运行时snd\_pcm\_runtime结构体实例，file实例中文件操作结构fops指针成员指向snd\_pcm\_f\_ops[cidx]数组项，函数调用关系及数据结构关系，如下图所示。





#### ■写函数

写函数适用于播放流对应的设备文件，用于放音操作，函数定义如下（/sound/core/pcm\_native.c）：

static ssize\_t snd\_pcm\_write(struct file \*file, const char \_\_user \*buf,size\_t count, loff\_t \* offset)

{

struct snd\_pcm\_file \*pcm\_file;

struct snd\_pcm\_substream \*substream;

struct snd\_pcm\_runtime \*runtime;

snd\_pcm\_sframes\_t result;

pcm\_file = **file->private\_data**;

**substream = pcm\_file->substream**; /\*子流\*/

...

runtime = substream->runtime; /\*运行时结构实例\*/

...

count = bytes\_to\_frames(runtime, count);

result = **snd\_pcm\_lib\_write(substream, buf, count)**;

/\*调用substream->ops->copy()函数等，/sound/core/pcm\_lib.c\*/

if (result > 0)

result = frames\_to\_bytes(runtime, result);

return result;

}

写操作函数将用户空间数据复制到运行时数据结构实例中的缓存内，随后由DMA机制将数据传送给编解码器。

#### ■读函数

读函数适用于录音流对应的设备文件，用于录音操作，函数定义如下（/sound/core/pcm\_native.c）：

static ssize\_t snd\_pcm\_read(struct file \*file, char \_\_user \*buf, size\_t count,loff\_t \* offset)

{

struct snd\_pcm\_file \*pcm\_file;

struct snd\_pcm\_substream \*substream;

struct snd\_pcm\_runtime \*runtime;

snd\_pcm\_sframes\_t result;

pcm\_file = **file->private\_data**;

substream = **pcm\_file->substream**;

...

runtime = substream->runtime;

...

count = bytes\_to\_frames(runtime, count);

result = **snd\_pcm\_lib\_read(substream, buf, count)**;

/\*调用substream->ops->copy()函数等，/sound/core/pcm\_lib.c\*/

if (result > 0)

result = frames\_to\_bytes(runtime, result);

return result;

}

读函数将运行时结构实例缓存区中数据复制到用户空间。

## 5控制设备

在创建声卡snd\_card实例的snd\_card\_new()函数中将调用snd\_ctl\_create(card)函数为声卡创建控制设备snd\_device实例，在注册设备时将为控制设备创建设备文件及注册对应的snd\_minor实例。通过控制接口可对控制设备进行控制，用户进程可通过设备文件/dev/snd/controlCx对声卡进行控制。

### 5.1概述

在创建声卡snd\_card实例时将为声卡创建控制设备，声卡驱动程可向控制设备添加控制接口。

#### ■创建控制设备

**snd\_ctl\_create()**函数用于为声卡创建控制设备，函数定义如下（/sound/core/control.c）：

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**, /\*注册snd\_minor实例\*/

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

};

int err;

...

**snd\_device\_initialize(&card->ctl\_dev, card)**; /\*初始化snd\_card->ctl\_dev实例，设置设备类等\*/

**dev\_set\_name(&card->ctl\_dev, "controlC%d", card->number)**; /\*设备文件名称controlCx\*/

err = **snd\_device\_new**(card, **SNDRV\_DEV\_CONTROL**, **card**, &ops); /\*创建通用设备\*/

...

return err;

}

snd\_ctl\_create()函数初始化card->ctl\_dev成员（表示控制设备的device实例），创建表示控制设备的通用snd\_device实例。注意snd\_device\_ops实例中的dev\_register()函数snd\_ctl\_dev\_register()，它将在注册设备时调用。

#### ■控制设备框架

在创建声卡的snd\_card\_new()函数中将为声卡创建控制设备snd\_device实例，在注册声卡时调用控制设备操作结构的注册函数，为控制设备创建并注册snd\_minor实例，以及相应的设备文件。

声卡控制接口由snd\_kcontrol结构体表示，结构体中包含控制接口的类型、名称、获取/设置控制参数的操作函数指针等成员，声卡驱动程序需要向声卡添加控制接口。



声卡控制设备文件名称为/dev/snd/controlCx（x表示声卡编号），设备文件操作结构为snd\_ctl\_f\_ops实例。在打开控制设备文件操作中将根据从设备号在snd\_minors[]数组中查找对应的snd\_minor实例，并将其关联的file\_operations实例赋予file实例，调用执行其中的open()函数。open()函数中将为file实例创建snd\_ctl\_file实例用于关联file实例与声卡驱动数据结构实例。用户通过对控制设备文件执行ioctl()系统调用实现对控制参数的操作（读或写）。

### 5.2控制接口

控制设备通过控制接口对设备进行控制，控制接口（功能）由snd\_kcontrol结构体表示，实例由snd\_card实例管理。内核提供了创建和添加控制接口的接口函数，**snd\_ctl\_new1()**函数用于动态创建控制接口（控制接口也可以由驱动程序静态创建），控制接口需要通过**snd\_ctl\_add()**函数添加到声卡。

#### ■数据结构

控制接口由snd\_kcontrol结构体表示，定义如下（/include/sound/control.h）：

struct snd\_kcontrol {

struct list\_head **list**; /\*双链表成员，将实例添加到snd\_card实例中双链表\*/

**struct snd\_ctl\_elem\_id id;**  /\*控制接口标识，snd\_ctl\_elem\_id结构体成员\*/

unsigned int count; /\* count of same elements \*/

**snd\_kcontrol\_info\_t \*info**; /\*获取接口信息\*/

**snd\_kcontrol\_get\_t \*get**; /\*获取控制参数值\*/

**snd\_kcontrol\_put\_t \*put**; /\*设置控制参数值\*/

union {

snd\_kcontrol\_tlv\_rw\_t \*c;

const unsigned int \*p;

} tlv;

unsigned long private\_value;

void \***private\_data**; /\*私有数据结构\*/

void (\*private\_free)(struct snd\_kcontrol \*kcontrol); /\*释放私有数据结构\*/

struct snd\_kcontrol\_volatile vd[0]; /\* volatile data \*/

};

snd\_kcontrol结构体主要成员简介如下：

●**list**：双链表成员，将实例链接到snd\_card->controls双链表中。

●**id：**snd\_ctl\_elem\_id结构体，表示控制接口标识。声卡每一个控制功能由一个控制接口表示，例如：音量控制等。内核在/include/uapi/sound/asound.h头文件定义了snd\_ctl\_elem\_id结构体及控制接口标识等。

struct snd\_ctl\_elem\_id {

unsigned int numid; /\* numeric identifier, zero = invalid \*/

**snd\_ctl\_elem\_iface\_t iface**; /\*接口标识（控制功能），/include/uapi/sound/asound.h\*/

unsigned int device; /\* device/client number \*/

unsigned int subdevice; /\* subdevice (substream) number \*/

unsigned char name[SNDRV\_CTL\_ELEM\_ID\_NAME\_MAXLEN]; /\* ASCII name of item \*/

unsigned int index; /\* index of item \*/

};

iface成员为snd\_ctl\_elem\_iface\_t数据类型，实际是一个整型数，表示控制接口的类型或者说功能，取值定义如下（/include/uapi/sound/asound.h）：

#define SNDRV\_CTL\_ELEM\_IFACE\_CARD ((\_\_force snd\_ctl\_elem\_iface\_t) 0) /\* global control \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_HWDEP ((\_\_force snd\_ctl\_elem\_iface\_t) 1)

/\* hardware dependent device \*/

#define **SNDRV\_CTL\_ELEM\_IFACE\_MIXER** ((\_\_force snd\_ctl\_elem\_iface\_t) 2)

/\* virtual mixer device \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_PCM ((\_\_force snd\_ctl\_elem\_iface\_t) 3) /\* PCM device \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_RAWMIDI ((\_\_force snd\_ctl\_elem\_iface\_t) 4)

/\* RawMidi device \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_TIMER ((\_\_force snd\_ctl\_elem\_iface\_t) 5) /\* timer device \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_SEQUENCER ((\_\_force snd\_ctl\_elem\_iface\_t) 6)

/\* sequencer client \*/

#define SNDRV\_CTL\_ELEM\_IFACE\_LAST SNDRV\_CTL\_ELEM\_IFACE\_SEQUENCER

●**info：**函数指针，用于获取接口信息，函数原型如下：

typedef int (snd\_kcontrol\_info\_t) (struct snd\_kcontrol \* kcontrol, struct snd\_ctl\_elem\_info \* uinfo);

●**get：**获取控制参数值，函数原型如下：

typedef int (snd\_kcontrol\_get\_t) (struct snd\_kcontrol \* kcontrol, struct snd\_ctl\_elem\_value \* ucontrol);

**●put：**设置控制参数值，函数原型如下：

typedef int (snd\_kcontrol\_put\_t) (struct snd\_kcontrol \* kcontrol, struct snd\_ctl\_elem\_value \* ucontrol);

以上函数参数中，snd\_ctl\_elem\_info结构体表示控制接口信息，snd\_ctl\_elem\_value结构体表示控制参数值，这两个结构体都定义在/include/uapi/sound/asound.h头文件，请读者自行阅读。

#### ■创建控制接口

在介绍创建控制接口函数前，先介绍一下snd\_kcontrol\_new结构体的定义，它是用于创建接口实例的模板，结构体定义如下（/include/sound/control.h）：

struct snd\_kcontrol\_new {

**snd\_ctl\_elem\_iface\_t iface**; /\*控制接口标识，前面介绍过\*/

unsigned int device; /\* device/client number \*/

unsigned int subdevice; /\* subdevice (substream) number \*/

const unsigned char \***name;** /\* ASCII name of item \*/

unsigned int index; /\* index of item \*/

unsigned int access; /\*访问权限，/include/sound/control.h\*/

unsigned int **count**; /\* count of same elements \*/

**snd\_kcontrol\_info\_t \*info**;

**snd\_kcontrol\_get\_t \*get**;

**snd\_kcontrol\_put\_t \*put**;

union {

snd\_kcontrol\_tlv\_rw\_t \*c;

const unsigned int \*p;

} tlv;

unsigned long private\_value;

};

snd\_kcontrol\_new结构体与snd\_kcontrol结构体类似。

动态创建snd\_kcontrol实例的接口函数**snd\_ctl\_new1()**定义如下（/sound/core/control.c）：

struct snd\_kcontrol \*snd\_ctl\_new1(const struct snd\_kcontrol\_new \*ncontrol,void \*private\_data)

/\*ncontrol：snd\_kcontrol\_new实例指针，由驱动程序传递，private\_data：私有数据结构\*/

{

struct snd\_kcontrol **\*kctl**;

unsigned int count;

unsigned int access;

int err;

...

count = ncontrol->count;

if (count == 0)

count = 1;

access = ncontrol->access; /\*访问权限\*/

if (access == 0)

access = SNDRV\_CTL\_ELEM\_ACCESS\_READWRITE;

access &= (SNDRV\_CTL\_ELEM\_ACCESS\_READWRITE |

SNDRV\_CTL\_ELEM\_ACCESS\_VOLATILE |

SNDRV\_CTL\_ELEM\_ACCESS\_INACTIVE |

SNDRV\_CTL\_ELEM\_ACCESS\_TLV\_READWRITE |

SNDRV\_CTL\_ELEM\_ACCESS\_TLV\_COMMAND |

SNDRV\_CTL\_ELEM\_ACCESS\_TLV\_CALLBACK);

err = **snd\_ctl\_new(&kctl, count, access, NULL);** /\*创建snd\_kcontrol实例，/sound/core/control.c\*/

if (err < 0)

return NULL;

/\* The 'numid' member is decided when calling snd\_ctl\_add(). \*/

kctl->id.iface = ncontrol->iface; /\*初始化snd\_kcontrol实例\*/

kctl->id.device = ncontrol->device;

kctl->id.subdevice = ncontrol->subdevice;

if (ncontrol->name) {

strlcpy(kctl->id.name, ncontrol->name, sizeof(kctl->id.name));

if (strcmp(ncontrol->name, kctl->id.name) != 0)

...

}

kctl->id.index = ncontrol->index;

kctl->info = ncontrol->info;

kctl->get = ncontrol->get;

kctl->put = ncontrol->put;

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

kctl->private\_value = ncontrol->private\_value;

kctl->private\_data = private\_data;

return kctl;

}

snd\_ctl\_new1()函数内调用snd\_ctl\_new()函数创建snd\_kcontrol实例，并依snd\_kcontrol\_new实例对创建实例进行初始化。

#### ■添加控制接口

控制接口snd\_kcontrol实例可动态创建也可静态定义，然后需要调用**snd\_ctl\_add()**函数将控制接口添加到声卡snd\_card实例，添加函数定义如下（/sound/core/control.c）：

int snd\_ctl\_add(struct snd\_card \*card, struct snd\_kcontrol \*kcontrol)

{

struct snd\_ctl\_elem\_id id;

unsigned int idx;

unsigned int count;

int err = -EINVAL;

...

id = kcontrol->id;

if (id.index > UINT\_MAX - kcontrol->count)

goto error;

down\_write(&card->controls\_rwsem);

if (snd\_ctl\_find\_id(card, &id)) { /\*查找控制接口是否已经存在\*/

...

}

if (snd\_ctl\_find\_hole(card, kcontrol->count) < 0) {

...

}

**list\_add\_tail(&kcontrol->list, &card->controls)**;

/\*将snd\_kcontrol实例添加到card->controls双链表末尾\*/

card->controls\_count += kcontrol->count;

kcontrol->id.numid = card->last\_numid + 1;

card->last\_numid += kcontrol->count;

id = kcontrol->id;

count = kcontrol->count;

up\_write(&card->controls\_rwsem);

for (idx = 0; idx < count; idx++, id.index++, id.numid++)

snd\_ctl\_notify(card, SNDRV\_CTL\_EVENT\_MASK\_ADD, &id); /\*向用户空间发送通知\*/

return 0;

...

}

snd\_ctl\_add()函数主要工作是将snd\_kcontrol实例添加到snd\_card->controls双链表末尾，并初始化相关成员。

另外，snd\_ctl\_register\_ioctl(snd\_kctl\_ioctl\_func\_t fcn)函数用于注册设备特定的控制接口。

### 5.3设备文件操作

在前面介绍的创建控制设备的snd\_ctl\_create()函数中，snd\_device实例关联的snd\_device\_ops实例的dev\_register()函数，即**snd\_ctl\_dev\_register()**函数用于注册表示控制设备的snd\_minor实例，函数定义如下：

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

{

struct **snd\_card** \*card = device->device\_data;

return **snd\_register\_device**(SNDRV\_DEVICE\_TYPE\_CONTROL, card, -1,

&**snd\_ctl\_f\_ops**, **card**, **&card->ctl\_dev**);

}

snd\_card->ctl\_dev成员（device）实例表示控制设备的device实例并添加到通用驱动模型中，内核依此创建设备文件，设备文件操作结构实例为**snd\_ctl\_f\_ops**。

控制设备文件操作结构实例为**snd\_ctl\_f\_ops，**定义如下（/sound/core/control.c）：

static const struct file\_operations snd\_ctl\_f\_ops =

{

.owner = THIS\_MODULE,

.read = **snd\_ctl\_read**, /\*读取snd\_ctl\_event链表中实例中数据\*/

.open = **snd\_ctl\_open**, /\*打开函数\*/

.release = snd\_ctl\_release,

.llseek = no\_llseek,

.poll = snd\_ctl\_poll,

.unlocked\_ioctl = **snd\_ctl\_ioctl**, /\*设备控制\*/

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

.fasync = snd\_ctl\_fasync,

};

下面简要介绍其中的打开函数和设备控制函数的实现。

#### ■打开函数

在介绍打开函数前，先介绍一下snd\_ctl\_file结构体的定义（/include/sound/control.h）：

struct snd\_ctl\_file {

struct list\_head list; /\*双链表成员\*/

struct snd\_card \*card; /\*声卡\*/

struct pid \*pid; /\*打开控制设备文件的进程pid\*/

int preferred\_subdevice[SND\_CTL\_SUBDEV\_ITEMS];

wait\_queue\_head\_t change\_sleep; /\*等待队列\*/

spinlock\_t read\_lock;

struct fasync\_struct \*fasync;

int subscribed; /\* read interface is activated \*/

struct list\_head **events**; /\*读操作时的等待事件链表\*/

};

snd\_ctl\_file结构体用于建立控制设备文件file与音频设备驱动之间的关联，用户通过控制设备文件/dev/snd/controlCx（x表示系统中声卡编号）对声卡进行控制，打开设备文件时为其创建一个snd\_ctl\_file实例。snd\_ctl\_file结构体主要成员简介如下：

●**list**：双链表成员，将实例链接到声卡控制设备文件双链表snd\_card-> ctl\_files。

●**events**：读操作时的等待事件链表，链表成员为snd\_kctl\_event结构体实例。

控制设备文件操作打开函数snd\_ctl\_open()定义如下（/sound/core/control.c）：

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

{

unsigned long flags;

struct snd\_card \*card;

struct **snd\_ctl\_file** \*ctl;

int i, err;

err = nonseekable\_open(inode, file);

if (err < 0)

return err;

card = snd\_lookup\_minor\_data(iminor(inode), SNDRV\_DEVICE\_TYPE\_CONTROL);

/\*获取snd\_card实例，snd\_minor实例的私有数据\*/

...

err = **snd\_card\_file\_add(card, file)**; /\*将设备文件添加到声卡设备文件链表，/sound/core/init.c\*/

...

if (!try\_module\_get(card->module)) {

...

}

ctl = **kzalloc(sizeof(\*ctl), GFP\_KERNEL)**; /\*分配**snd\_ctl\_file**实例\*/

...

INIT\_LIST\_HEAD(&ctl->events); /\*初始化等待事件链表\*/

init\_waitqueue\_head(&ctl->change\_sleep);

spin\_lock\_init(&ctl->read\_lock);

ctl->card = card;

for (i = 0; i < SND\_CTL\_SUBDEV\_ITEMS; i++)

ctl->preferred\_subdevice[i] = -1;

ctl->pid = get\_pid(task\_pid(current)); /\*当前进程pid\*/

**file->private\_data = ctl**; /\*snd\_ctl\_file实例作为file的私有数据\*/

write\_lock\_irqsave(&card->ctl\_files\_rwlock, flags);

**list\_add\_tail(&ctl->list, &card->ctl\_files)**; /\*添加到控制设备文件双链表末尾\*/

write\_unlock\_irqrestore(&card->ctl\_files\_rwlock, flags);

snd\_card\_unref(card);

return 0;

...

}

#### ■设备控制

用户进程在打开声卡控制设备文件后，通过ioctl()系统调用对声卡控制参数进行操作。声卡控制命令及相关的数据结构定义在/include/uapi/sound/asound.h头文件，例如：

#define SNDRV\_CTL\_IOCTL\_PVERSION \_IOR('U', 0x00, int)

#define SNDRV\_CTL\_IOCTL\_CARD\_INFO \_IOR('U', 0x01, struct snd\_ctl\_card\_info)

#define SNDRV\_CTL\_IOCTL\_ELEM\_LIST \_IOWR('U', 0x10, struct snd\_ctl\_elem\_list)

#define SNDRV\_CTL\_IOCTL\_ELEM\_INFO \_IOWR('U', 0x11, struct snd\_ctl\_elem\_info)

#define SNDRV\_CTL\_IOCTL\_ELEM\_READ \_IOWR('U', 0x12, struct snd\_ctl\_elem\_value)

#define SNDRV\_CTL\_IOCTL\_ELEM\_WRITE \_IOWR('U', 0x13, struct snd\_ctl\_elem\_value)

...

ioctl()系统调用最终调用snd\_ctl\_f\_ops实例中的unlocked\_ioctl()函数，即**snd\_ctl\_ioctl()**函数，对声卡控制参数进行操作，函数代码简列如下（/sound/core/control.c）：

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

{

struct snd\_ctl\_file \*ctl;

struct snd\_card \*card;

**struct snd\_kctl\_ioctl \*p**;

void \_\_user \*argp = (void \_\_user \*)arg;

int \_\_user \*ip = argp;

int err;

**ctl = file->private\_data**;

card = ctl->card;

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

return -ENXIO;

switch (cmd) {

case SNDRV\_CTL\_IOCTL\_PVERSION:

return put\_user(SNDRV\_CTL\_VERSION, ip) ? -EFAULT : 0;

case SNDRV\_CTL\_IOCTL\_CARD\_INFO:

return snd\_ctl\_card\_info(card, ctl, cmd, argp);

case SNDRV\_CTL\_IOCTL\_ELEM\_LIST:

return snd\_ctl\_elem\_list(card, argp);

case SNDRV\_CTL\_IOCTL\_ELEM\_INFO:

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

case **SNDRV\_CTL\_IOCTL\_ELEM\_READ**:

return **snd\_ctl\_elem\_read\_user(card, argp)**; /\*读控制参数，snd\_kcontrol->get()\*/

case **SNDRV\_CTL\_IOCTL\_ELEM\_WRITE**:

return **snd\_ctl\_elem\_write\_user(ctl, argp)**; /\*写控制参数，snd\_kcontrol->put()\*/

case SNDRV\_CTL\_IOCTL\_ELEM\_LOCK:

return snd\_ctl\_elem\_lock(ctl, argp);

case SNDRV\_CTL\_IOCTL\_ELEM\_UNLOCK:

return snd\_ctl\_elem\_unlock(ctl, argp);

case SNDRV\_CTL\_IOCTL\_ELEM\_ADD:

return snd\_ctl\_elem\_add\_user(ctl, argp, 0);

case SNDRV\_CTL\_IOCTL\_ELEM\_REPLACE:

return snd\_ctl\_elem\_add\_user(ctl, argp, 1);

case SNDRV\_CTL\_IOCTL\_ELEM\_REMOVE:

return snd\_ctl\_elem\_remove(ctl, argp);

case SNDRV\_CTL\_IOCTL\_SUBSCRIBE\_EVENTS:

return snd\_ctl\_subscribe\_events(ctl, ip);

case SNDRV\_CTL\_IOCTL\_TLV\_READ:

return snd\_ctl\_tlv\_ioctl(ctl, argp, SNDRV\_CTL\_TLV\_OP\_READ);

case SNDRV\_CTL\_IOCTL\_TLV\_WRITE:

return snd\_ctl\_tlv\_ioctl(ctl, argp, SNDRV\_CTL\_TLV\_OP\_WRITE);

case SNDRV\_CTL\_IOCTL\_TLV\_COMMAND:

return snd\_ctl\_tlv\_ioctl(ctl, argp, SNDRV\_CTL\_TLV\_OP\_CMD);

case SNDRV\_CTL\_IOCTL\_POWER:

return -ENOPROTOOPT;

case SNDRV\_CTL\_IOCTL\_POWER\_STATE:

#ifdef CONFIG\_PM

return put\_user(card->power\_state, ip) ? -EFAULT : 0;

#else

return put\_user(SNDRV\_CTL\_POWER\_D0, ip) ? -EFAULT : 0;

#endif

}

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

list\_for\_each\_entry(p, &snd\_control\_ioctls, list) {

/\*扫描音频设备（PCM设备等）注册的特殊控制接口\*/

err = p->fioctl(card, ctl, cmd, arg);

...

}

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

dev\_dbg(card->dev, "unknown ioctl = 0x%x\n", cmd);

return -ENOTTY;

}

## 6 SOC音频驱动

SOC嵌入式处理器中，CPU通过数字音频接口DAI（PCM、IIS、AC97接口控制器）与外部编解码器相连，编解码器与播放音设备相连，如下图所示。



ALSA SOC音频驱动框架可为SOC嵌入式处理器和便携音频编码器提供更好的ALSA支持，它在ALSA的基础上实现。SOC音频驱动框架分为CPU侧驱动、平台驱动、音频编解码器驱动和机器驱动，各部分驱动相互独立，代码可重用性更好。例如，同一个编解码器驱动又可以适用于所有平台。另外，它提供了标准的音频事件的处理接口、电源控制接口。

SOC音频驱动框架相关代码位于/sound/soc/目录下，主要公共层代码在/sound/soc/soc-core.c文件内实现。

### 6.1驱动框架

SOC音频驱动框架主要分成四部分，分别是CPU驱动、编解码器codec驱动、平台platform驱动和机器machine板级驱动，如下图所示。



如上图中所示，各部分驱动（除机器驱动外）的核心数据结构是snd\_soc\_component结构体，所有实例（称为组件）由全局双链表component\_list管理。组件有对应的组件驱动，由snd\_soc\_component\_driver结构体表示。

CPU驱动中，snd\_soc\_component实例管理了表示CPU侧DAI的snd\_soc\_dai结构体实例，以及对应的DAI驱动snd\_soc\_dai\_driver实例，主要实现CPU侧DAI的控制和设置。CPU侧对DAI的操作主要是对控制器寄存器的操作。

CPU侧驱动的主要工作是实现组件驱动snd\_soc\_component\_driver实例和DAI驱动snd\_soc\_dai\_driver实例（一个实例对应一个DAI实例），并依此创建并注册CPU侧的组件snd\_soc\_component实例。

编解码器侧驱动中，编解码器由snd\_soc\_codec结构体表示，内嵌组件snd\_soc\_component结构体成员，编解码器驱动由snd\_soc\_codec\_driver结构体表示。组件成员中包含编解码器侧DAI的snd\_soc\_dai结构体实例，以及对应的DAI驱动snd\_soc\_dai\_driver实例，主要实现编解码器侧DAI的控制和设置。编解码器侧DAI的操作需要按编解码器的通信协议对其进行操作，例如通过写入命令。

编解码器侧驱动需要实现snd\_soc\_codec\_driver实例，以及DAI驱动snd\_soc\_dai\_driver实例，并依此创建并注册编解码器snd\_soc\_codec实例（同时注册snd\_soc\_component实例）。

平台驱动中，平台由snd\_soc\_platform结构体表示（内嵌snd\_soc\_component结构体），平台驱动由snd\_soc\_platform\_driver结构体表示（包含snd\_pcm\_ops结构体指针成员）。平台驱动最重要的工作是实现snd\_pcm\_ops结构体中的部分函数，用于实现CPU侧DAI接口控制器与内存之间的数据传输，例如：SOC通过DMA实现DAI接口控制器与内存之间的数据传输。

平台侧驱动需要实现驱动snd\_soc\_platform\_driver实例，并依此创建和注册snd\_soc\_platform实例（同时注册snd\_soc\_component实例）。所有注册的编解码器和平台实例都由各自的全局双链表管理。

机器驱动的主要作用是建立CPU驱动、编解码器驱动、平台驱动三者之间的关联。机器驱动中需要定义snd\_soc\_card实例，以及描述CPU侧DAI和关联编解码器等信息的snd\_soc\_dai\_link实例（数组），并注册snd\_soc\_card实例。

SOC音频驱动框架公共层定义了名称为“soc-audio”的platform\_driver实例，并在内核启动阶段注册。机器驱动可以定义名称“soc-audio”的platform\_device实例和snd\_soc\_card实例（含snd\_soc\_dai\_link实例（数组）），并将snd\_soc\_card实例指针赋予platform\_device.dev成员（device实例）的驱动数据结构指针driver\_data成员，最后向内核核注册platform\_device实例。

当“soc-audio”名称的platform\_device实例匹配上同名的platform\_driver实例后，将调用其中的probe()函数，此函数内将注册snd\_soc\_card实例，创建并注册通用的声卡snd\_card实例，依据snd\_soc\_dai\_link实例描述的DAI接口信息，创建运行时snd\_soc\_pcm\_runtime实例，由此实例建立与CPU、编解码器、平台驱动之间的关联。

snd\_pcm文件操作将调用snd\_soc\_pcm\_runtime实例内嵌snd\_pcm\_ops实例中的函数，而此实例中的函数调用了CPU侧DAI驱动、编解码器侧DAI驱动关联snd\_soc\_dai\_ops实例中的函数，以及平台驱动关联的snd\_pcm\_ops实例中的函数。

### 6.2 CPU驱动

SOC音频驱动框架中组件由snd\_soc\_component结构体表示，组件驱动由snd\_soc\_component\_driver结构体表示。组件中包含组件内DAI驱动snd\_soc\_dai\_driver实例（数组）。

在注册组件时将动态创建组件snd\_soc\_component实例（插入全局双链表），依snd\_soc\_dai\_driver实例（数组）创建表示DAI的snd\_soc\_dai实例（双链表），如下图所示。



下面先简要介绍各数据结构的定义，然后介绍注册组件函数snd\_soc\_register\_component()的实现。

#### ■ DAI及驱动

DAI表示数字音频接口，在SOC音频驱动框架中，DAI分为CPU侧DAI和编解码器侧DAI。每个DAI由**snd\_soc\_dai**结构体表示，定义如下（/include/sound/soc-dai.h）：

struct snd\_soc\_dai {

const char \***name**; /\*名称，来自关联的snd\_soc\_dai\_driver实例\*/

int id; /\*编号\*/

struct device \*dev; /\*指向表示DAI的device实例\*/

**struct snd\_soc\_dai\_driver \*driver**; /\*指向DAI驱动实例\*/

/\* DAI 运行时信息\*/

unsigned int capture\_active:1; /\*录音流在使用\*/

unsigned int playback\_active:1; /\*播音流在使用\*/

unsigned int symmetric\_rates:1;

unsigned int symmetric\_channels:1;

unsigned int symmetric\_samplebits:1;

unsigned int active;

unsigned char probed:1;

struct snd\_soc\_dapm\_widget \*playback\_widget;

struct snd\_soc\_dapm\_widget \*capture\_widget;

/\* DAI DMA参数\*/

**void \*playback\_dma\_data**;

**void \*capture\_dma\_data**;

/\* Symmetry data - only valid if symmetry is being enforced \*/

unsigned int rate;

unsigned int channels;

unsigned int sample\_bits;

**struct snd\_soc\_codec \*codec**; /\*指向所属编解码器\*/

struct snd\_soc\_component \***component**; /\*指向所属组件\*/

/\* CODEC TDM slot masks and params (for fixup) \*/

unsigned int tx\_mask;

unsigned int rx\_mask;

**struct list\_head list**; /\*将实例添加到组件dai\_list双链表，见下文\*/

};

snd\_soc\_dai实例根据驱动snd\_soc\_dai\_driver实例动态创建，两个实例间一一对应。

DAI驱动由snd\_soc\_dai\_driver结构体表示，定义如下（/include/sound/soc-dai.h）：

struct snd\_soc\_dai\_driver {

/\* DAI描述\*/

const char \***name**; /\*名称，赋予关联snd\_soc\_dai实例\*/

unsigned int id; /\*编号，赋予关联snd\_soc\_dai实例\*/

unsigned int base;

/\* DAI驱动回调函数\*/

int (\*probe)(struct snd\_soc\_dai \*dai);

int (\*remove)(struct snd\_soc\_dai \*dai);

int (\*suspend)(struct snd\_soc\_dai \*dai);

int (\*resume)(struct snd\_soc\_dai \*dai);

/\* compress dai \*/

bool compress\_dai;

/\* DAI is also used for the control bus \*/

bool bus\_control;

**const struct snd\_soc\_dai\_ops \*ops**; /\*DAI操作结构指针\*/

/\* DAI capabilities \*/

**struct snd\_soc\_pcm\_stream capture**; /\*录音PCM流信息，/include/sound/soc.h\*/

**struct snd\_soc\_pcm\_stream playback**; /\*播音PCM流信息，/include/sound/soc.h\*/

unsigned int symmetric\_rates:1;

unsigned int symmetric\_channels:1;

unsigned int symmetric\_samplebits:1;

/\* probe ordering - for components with runtime dependencies \*/

int **probe\_order**;

int remove\_order;

};

DAI驱动snd\_soc\_dai\_driver结构体关联的**snd\_soc\_dai\_ops**结构体定义如下（/include/sound/soc-dai.h），主要包含对DAI的操作函数指针：

struct snd\_soc\_dai\_ops {

/\* DAI 时钟配置函数，可选。\*/

int (\*set\_sysclk)(struct snd\_soc\_dai \*dai,int clk\_id, unsigned int freq, int dir);

int (\*set\_pll)(struct snd\_soc\_dai \*dai, int pll\_id, int source,unsigned int freq\_in, unsigned int freq\_out);

int (\*set\_clkdiv)(struct snd\_soc\_dai \*dai, int div\_id, int div);

int (\*set\_bclk\_ratio)(struct snd\_soc\_dai \*dai, unsigned int ratio);

/\* DAI 格式化配置函数\*/

int (\*set\_fmt)(struct snd\_soc\_dai \*dai, unsigned int fmt);

int (\*xlate\_tdm\_slot\_mask)(unsigned int slots,unsigned int \*tx\_mask, unsigned int \*rx\_mask);

int (\*set\_tdm\_slot)(struct snd\_soc\_dai \*dai,

unsigned int tx\_mask, unsigned int rx\_mask,int slots, int slot\_width);

int (\*set\_channel\_map)(struct snd\_soc\_dai \*dai,

unsigned int tx\_num, unsigned int \*tx\_slot,unsigned int rx\_num, unsigned int \*rx\_slot);

int (\*set\_tristate)(struct snd\_soc\_dai \*dai, int tristate);

int (\*digital\_mute)(struct snd\_soc\_dai \*dai, int mute);

int (\*mute\_stream)(struct snd\_soc\_dai \*dai, int mute, int stream);

int (\*startup)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

void (\*shutdown)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

int (\*hw\_params)(struct snd\_pcm\_substream \*,struct snd\_pcm\_hw\_params \*, struct snd\_soc\_dai \*);

int (\*hw\_free)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

int (\*prepare)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

int (\*trigger)(struct snd\_pcm\_substream \*, int,struct snd\_soc\_dai \*);

int (\*bespoke\_trigger)(struct snd\_pcm\_substream \*, int,struct snd\_soc\_dai \*);

snd\_pcm\_sframes\_t (\*delay)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

};

#### ■组件及驱动

组件snd\_soc\_component结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_component {

const char \*name; /\*名称，来自驱动snd\_soc\_component\_driver实例\*/

int id;

const char \*name\_prefix;

**struct device \*dev;**  /\*指向表示组件的device实例\*/

**struct snd\_soc\_card \*card**; /\*指向SOC声卡实例\*/

unsigned int active;

unsigned int ignore\_pmdown\_time:1; /\* pmdown\_time is ignored at stop \*/

unsigned int registered\_as\_component:1;

unsigned int probed:1;

**struct list\_head list**; /\*将snd\_soc\_component实例添加到全局双链表component\_list\*/

**struct snd\_soc\_dai\_driver \*dai\_drv**; /\*指向DAI驱动实例（数组）\*/

int num\_dai; /\*DAI驱动数量\*/

**const struct snd\_soc\_component\_driver \*driver**; /\*指向组件驱动snd\_soc\_component实例\*/

**struct list\_head dai\_list**; /\*链接snd\_soc\_dai实例\*/

int (\***read**)(struct snd\_soc\_component \*, unsigned int, unsigned int \*);

int (\***write**)(struct snd\_soc\_component \*, unsigned int, unsigned int);

struct regmap \*regmap;

int val\_bytes;

struct mutex io\_mutex;

struct list\_head dobj\_list;

...

struct snd\_soc\_dapm\_context dapm;

struct snd\_soc\_dapm\_context \*dapm\_ptr;

**const struct snd\_kcontrol\_new \*controls;** /\*控制接口及数量\*/

**unsigned int num\_controls**;

const struct snd\_soc\_dapm\_widget \*dapm\_widgets;

unsigned int num\_dapm\_widgets;

const struct snd\_soc\_dapm\_route \*dapm\_routes;

unsigned int num\_dapm\_routes;

**struct snd\_soc\_codec \*codec**; /\*指向编解码器实例\*/

int (\***probe**)(struct snd\_soc\_component \*); /\*调用平台驱动或编解码器驱动中的probe()函数\*/

void (\*remove)(struct snd\_soc\_component \*);

...

};

组件驱动snd\_soc\_component\_driver结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_component\_driver {

const char \*name; /\*名称，赋予snd\_soc\_component实例\*/

**const struct snd\_kcontrol\_new \*controls;** /\*控制接口\*/

**unsigned int num\_controls**; /\*控制接口数量\*/

const struct snd\_soc\_dapm\_widget \*dapm\_widgets;

unsigned int num\_dapm\_widgets;

const struct snd\_soc\_dapm\_route \*dapm\_routes;

unsigned int num\_dapm\_routes;

int (\***probe**)(struct snd\_soc\_component \*); /\*赋予snd\_soc\_component\*/

void (\*remove)(struct snd\_soc\_component \*); /\*赋予snd\_soc\_component\*/

/\* DT \*/

int (\*of\_xlate\_dai\_name)(struct snd\_soc\_component \*component,

struct of\_phandle\_args \*args,const char \*\*dai\_name);

void (\*seq\_notifier)(struct snd\_soc\_component \*, enum snd\_soc\_dapm\_type,int subseq);

int (\*stream\_event)(struct snd\_soc\_component \*, int event);

/\* probe ordering - for components with runtime dependencies \*/

int **probe\_order**;

int remove\_order;

};

#### ■注册组件

CPU侧驱动程序需要实现组件驱动**s**nd\_soc\_component\_driver实例、各DAI驱动snd\_soc\_dai\_driver实例以及关联的snd\_soc\_dai\_ops实例。

函数**snd\_soc\_register\_component()**将动态创建并注册组件实例，函数定义如下（/sound/soc/soc-core.c）：

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**)

/\*dev：指向表示组件的device实例，cmpnt\_drv：指向组件驱动，

\*dai\_drv：指向DAI驱动实例（数组），num\_dai：DAI驱动数组项数\*/

{

struct snd\_soc\_component \*cmpnt; /\*组件\*/

int ret;

**cmpnt** = kzalloc(sizeof(\*cmpnt), GFP\_KERNEL); /\*创建snd\_soc\_component实例\*/

...

ret = **snd\_soc\_component\_initialize**(cmpnt, cmpnt\_drv, dev); /\*初始化snd\_soc\_component实例\*/

...

cmpnt->ignore\_pmdown\_time = true;

cmpnt->registered\_as\_component = true;

ret = **snd\_soc\_register\_dais(cmpnt, dai\_drv, num\_dai, true)**; /\*创建并注册snd\_soc\_dai实例\*/

...

**snd\_soc\_component\_add(cmpnt)**; /\*添加snd\_soc\_component实例至全局双链表component\_list\*/

return 0;

...

}

snd\_soc\_component\_initialize()函数用于初始化创建的snd\_soc\_component实例，源代码请读者自行阅读。snd\_soc\_component\_add(cmpnt)函数用于将创建的snd\_soc\_component实例添加到component\_list全局双链表。

snd\_soc\_register\_dais()函数用于根据参数传递的snd\_soc\_dai\_driver实例（数组），创建snd\_soc\_dai实例，并添加到snd\_soc\_component实例的**dai\_list**双链表中，函数定义如下（/sound/soc/soc-core.c）：

static int snd\_soc\_register\_dais(struct snd\_soc\_component \*component,

struct snd\_soc\_dai\_driver \***dai\_drv**, size\_t count,bool legacy\_dai\_naming)

/\*dai\_drv：指向snd\_soc\_dai\_driver实例（数组），count：数组项数（DAI驱动数量）\*/

{

struct device \*dev = component->dev;

struct snd\_soc\_dai \*dai;

unsigned int i;

int ret;

...

component->dai\_drv = dai\_drv; /\*指向snd\_soc\_dai\_driver实例（数组）\*/

component->num\_dai = count;

for (i = 0; i < count; i++) { /\*遍历snd\_soc\_dai\_driver实例（数组）\*/

**dai = kzalloc(sizeof(struct snd\_soc\_dai), GFP\_KERNEL)**; /\*分配snd\_soc\_dai实例\*/

...

if (count == 1 && legacy\_dai\_naming && /\*生成组件名称\*/

(dai\_drv[i].id == 0 || dai\_drv[i].name == NULL)) {

dai->name = fmt\_single\_name(dev, &dai->id);

} else {

dai->name = fmt\_multiple\_name(dev, &dai\_drv[i]);

if (dai\_drv[i].id)

**dai->id = dai\_drv[i].id**; /\*编号\*/

else

dai->id = i;

}

...

dai->component = component;

dai->dev = dev;

**dai->driver = &dai\_drv[i]**; /\*关联snd\_soc\_dai\_driver实例（与数组项一一对应）\*/

if (!dai->driver->ops)

dai->driver->ops = &null\_dai\_ops;

**list\_add(&dai->list, &component->dai\_list)**;

/\*将snd\_soc\_dai实例添加到snd\_soc\_component实例dai\_list双链表\*/

...

}

return 0;

...

}

### 6.3编解码器及驱动

编解码器由snd\_soc\_codec结构体表示，编解码器驱动由snd\_soc\_codec\_driver结构体表示。编解码器驱动程序需要实现snd\_soc\_codec\_driver实例，以及编解码器侧DAI驱动snd\_soc\_dai\_driver实例，然后依此调用**snd\_soc\_register\_code()**函数创建并注册编解码器实例，函数执行结果如下图所示。



#### ■数据结构

音频编解码器由snd\_soc\_codec结构体表示，结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_codec {

struct device \*dev; /\*指向表示编解码器的device实例\*/

**const struct snd\_soc\_codec\_driver \*driver**; /\*指向编解码器驱动\*/

struct list\_head **list**; /\*将实例添加到全局双链表codec\_list\*/

struct list\_head **card\_list**;

/\*运行时信息\*/

unsigned int cache\_bypass:1; /\* Suppress access to the cache \*/

unsigned int suspended:1; /\* Codec is in suspend PM state \*/

unsigned int cache\_init:1; /\* codec cache has been initialized \*/

/\* codec IO \*/

void \*control\_data; /\* codec control (i2c/3wire) data \*/

hw\_write\_t hw\_write;

void \***reg\_cache**; /\*指向数据缓存区\*/

**struct snd\_soc\_component component**; /\*内嵌snd\_soc\_component实例\*/

/\* Don't access this directly, use snd\_soc\_codec\_get\_dapm() \*/

struct snd\_soc\_dapm\_context dapm;

...

};

编解码器驱动由snd\_soc\_codec\_driver结构体表示，结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_codec\_driver {

/\* driver ops \*/

int (\***probe**)(struct snd\_soc\_codec \*);

int (\*remove)(struct snd\_soc\_codec \*);

int (\*suspend)(struct snd\_soc\_codec \*);

int (\*resume)(struct snd\_soc\_codec \*);

**struct snd\_soc\_component\_driver component\_driver**; /\*内嵌组件驱动\*/

/\* Default control and setup, added after probe() is run \*/

const struct snd\_kcontrol\_new \***controls**; /\*默认控制接口\*/

int **num\_controls**;

const struct snd\_soc\_dapm\_widget \*dapm\_widgets;

int num\_dapm\_widgets;

const struct snd\_soc\_dapm\_route \*dapm\_routes;

int num\_dapm\_routes;

/\* codec wide operations \*/

int (\*set\_sysclk)(struct snd\_soc\_codec \*codec,int clk\_id, int source, unsigned int freq, int dir);

int (\*set\_pll)(struct snd\_soc\_codec \*codec, int pll\_id, int source,

unsigned int freq\_in, unsigned int freq\_out);

/\* codec IO操作 \*/

struct regmap \*(\*get\_regmap)(struct device \*);

unsigned int (\***read**)(struct snd\_soc\_codec \*, unsigned int);

int (\***write**)(struct snd\_soc\_codec \*, unsigned int, unsigned int);

unsigned int reg\_cache\_size;

short reg\_cache\_step;

short reg\_word\_size;

const void \*reg\_cache\_default;

/\* codec bias level \*/

int (\*set\_bias\_level)(struct snd\_soc\_codec \*,enum snd\_soc\_bias\_level level);

bool idle\_bias\_off;

bool suspend\_bias\_off;

void (\*seq\_notifier)(struct snd\_soc\_dapm\_context \*,enum snd\_soc\_dapm\_type, int);

bool ignore\_pmdown\_time; /\* Doesn't benefit from pmdown delay \*/

};

#### ■注册编解码器

编解码器驱动程序需要实现snd\_soc\_codec\_driver实例，以及编解码器侧DAI驱动snd\_soc\_dai\_driver实例，调用函数**snd\_soc\_register\_codec()**创建并注册编解码器实例，函数定义如下（/sound/soc/soc-core.c）：

int snd\_soc\_register\_codec(struct device \*dev,const struct snd\_soc\_codec\_driver \***codec\_drv**,

struct snd\_soc\_dai\_driver \***dai\_drv,**int num\_dai)

/\*dev：指向表示编解码器的device实例，codec\_drv：指向编解码器驱动，

\*dai\_drv：指向DAI驱动（数组），num\_dai：DAI驱动数量\*/

{

struct snd\_soc\_codec \*codec;

struct snd\_soc\_dai \*dai;

int ret, i;

...

codec = **kzalloc(sizeof(struct snd\_soc\_codec), GFP\_KERNEL)**; /\*创建**snd\_soc\_codec**实例\*/

...

codec->component.dapm\_ptr = &codec->dapm;

**codec->component.codec = codec**; /\*编解码器\*/

ret = **snd\_soc\_component\_initialize**(&codec->component,&codec\_drv->component\_driver, dev);

/\*初始化snd\_soc\_codec内嵌snd\_soc\_component实例\*/

...

if (codec\_drv->controls) { /\*驱动控制接口赋予组件snd\_soc\_component实例\*/

**codec->component.controls = codec\_drv->controls**;

**codec->component.num\_controls = codec\_drv->num\_controls;**

}

if (codec\_drv->dapm\_widgets) {

codec->component.dapm\_widgets = codec\_drv->dapm\_widgets;

codec->component.num\_dapm\_widgets = codec\_drv->num\_dapm\_widgets;

}

if (codec\_drv->dapm\_routes) {

codec->component.dapm\_routes = codec\_drv->dapm\_routes;

codec->component.num\_dapm\_routes = codec\_drv->num\_dapm\_routes;

}

if (codec\_drv->probe) /\*设置内嵌snd\_soc\_component实例成员\*/

**codec->component.probe** = snd\_soc\_codec\_drv\_probe; /\*调用codec->driver->probe(codec)\*/

if (codec\_drv->remove)

codec->component.remove = snd\_soc\_codec\_drv\_remove;

if (codec\_drv->write)

codec->component.write = snd\_soc\_codec\_drv\_write;

/\*调用snd\_soc\_codec\_driver中write()函数\*/

if (codec\_drv->read)

codec->component.read = snd\_soc\_codec\_drv\_read; /\*调用snd\_soc\_codec\_driver中read()函数\*/

codec->component.ignore\_pmdown\_time = codec\_drv->ignore\_pmdown\_time;

codec->dapm.idle\_bias\_off = codec\_drv->idle\_bias\_off;

codec->dapm.suspend\_bias\_off = codec\_drv->suspend\_bias\_off;

if (codec\_drv->seq\_notifier)

codec->dapm.seq\_notifier = codec\_drv->seq\_notifier;

if (codec\_drv->set\_bias\_level)

codec->dapm.set\_bias\_level = snd\_soc\_codec\_set\_bias\_level;

codec->dev = dev;

**codec->driver = codec\_drv**; /\*指向编解码器驱动\*/

codec->component.val\_bytes = codec\_drv->reg\_word\_size;

...

if (codec\_drv->get\_regmap)

codec->component.regmap = **codec\_drv->get\_regmap(dev)**;

for (i = 0; i < num\_dai; i++) {

fixup\_codec\_formats(&dai\_drv[i].playback);

fixup\_codec\_formats(&dai\_drv[i].capture);

}

ret = **snd\_soc\_register\_dais**(&codec->component, **dai\_drv**, num\_dai, false);

/\*创建并注册snd\_soc\_dai实例\*/

...

list\_for\_each\_entry(dai, &codec->component.dai\_list, list)

**dai->codec = codec;** /\*指向编解码器\*/

mutex\_lock(&client\_mutex);

**snd\_soc\_component\_add\_unlocked**(&codec->component); /\*添加snd\_soc\_component实例\*/

**list\_add(&codec->list, &codec\_list)**; /\*snd\_soc\_codec实例添加到全局双链表codec\_list\*/

mutex\_unlock(&client\_mutex);

...

return 0;

...

}

### 6.4平台及驱动

SOC音频驱动框架中平台由snd\_soc\_platform结构体表示，平台驱动由snd\_soc\_platform\_driver结构体表示。平台驱动程序需要实现snd\_soc\_platform\_driver实例及关联的**snd\_pcm\_ops**实例（赋予PCM子流），调用接口函数**snd\_soc\_register\_platform()**创建并注册平台snd\_soc\_platform实例，函数执行结果如下图所示。



#### ■数据结构

平台snd\_soc\_platform结构体定义如下（/include/sound/soc.h）

struct snd\_soc\_platform {

struct device \*dev; /\*指向表示平台的device实例\*/

**const struct snd\_soc\_platform\_driver \*driver**; /\*指向平台驱动\*/

struct list\_head **list**; /\*将实例添加到全局双链表platform\_list\*/

struct snd\_soc\_component **component**; /\*内嵌snd\_soc\_component结构体实例\*/

};

平台驱动snd\_soc\_platform\_driver结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_platform\_driver {

int (\***probe**)(struct snd\_soc\_platform \*);

int (\*remove)(struct snd\_soc\_platform \*);

**struct snd\_soc\_component\_driver component\_driver**; /\*内嵌snd\_soc\_component\_driver实例\*/

int (\***pcm\_new**)(struct snd\_soc\_pcm\_runtime \*);

void (\*pcm\_free)(struct snd\_pcm \*);

snd\_pcm\_sframes\_t (\*delay)(struct snd\_pcm\_substream \*,struct snd\_soc\_dai \*);

**const struct snd\_pcm\_ops \*ops**; /\*PCM操作结构\*/

const struct snd\_compr\_ops \*compr\_ops;

int (\*bespoke\_trigger)(struct snd\_pcm\_substream \*, int);

};

平台驱动最重要的是要实例**snd\_pcm\_ops**结构体实例，此实例中的函数完成底层硬件的数据传输。

#### ■注册平台

注册平台的**snd\_soc\_register\_platform()**函数定义如下（/sound/soc/soc-core.c）：

int snd\_soc\_register\_platform(struct device \*dev,const struct snd\_soc\_platform\_driver \***platform\_drv**)

/\*dev：表示平台的device实例，platform\_drv：指向平台驱动实例\*/

{

struct snd\_soc\_platform \*platform;

int ret;

...

platform = kzalloc(sizeof(struct **snd\_soc\_platform**), GFP\_KERNEL); /\*分配snd\_soc\_platform\*/

...

ret = **snd\_soc\_add\_platform(dev, platform, platform\_drv)**; /\*添加snd\_soc\_platform实例\*/

...

return ret;

}

snd\_soc\_add\_platform()函数定义如下（/sound/soc/soc-core.c）：

int snd\_soc\_add\_platform(struct device \*dev, struct snd\_soc\_platform \***platform,**

const struct snd\_soc\_platform\_driver \***platform\_drv**)

{

int ret;

ret = **snd\_soc\_component\_initialize**(&platform->component,&platform\_drv->component\_driver, dev);

/\*初始化snd\_soc\_platform内嵌snd\_soc\_componentr实例\*/

..

platform->dev = dev;

platform->driver = **platform\_drv**; /\*指向snd\_soc\_platform\_driver实例\*/

if (platform\_drv->probe)

platform->component.probe = **snd\_soc\_platform\_drv\_probe**;

/\*调用platform->driver->probe(platform)\*/

if (platform\_drv->remove)

platform->component.remove = snd\_soc\_platform\_drv\_remove;

...

mutex\_lock(&client\_mutex);

**snd\_soc\_component\_add\_unlocked(&platform->component)**;

/\*将内嵌snd\_soc\_component实例添加到全局链表\*/

**list\_add(&platform->list, &platform\_list)**; /\*将snd\_soc\_platform实例添加到全局双链表\*/

mutex\_unlock(&client\_mutex);

...

return 0;

}

### 6.5机器驱动数据结构

在SOC音频驱动框架中机器machine驱动是CPU驱动、编解码器驱动和平台驱动的粘合剂，负责建立三者之间的关联，并关联通用声卡驱动层。下面先介绍一下相关的数据结构，然后再介绍机器machine驱动的实现。机器machine驱动相关数据结构组织关系如下图所示：



#### ■ snd\_soc\_card

snd\_soc\_card结构体表示SOC声卡，结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_card {

const char \*name;

const char \*long\_name;

const char \*driver\_name;

**struct device \*dev**; /\*指向表示SOC声卡的device实例，注册前不能为NULL\*/

**struct snd\_card \*snd\_card;** /\*指向通用声卡实例\*/

struct module \*owner;

struct mutex mutex;

struct mutex dapm\_mutex;

bool instantiated;

int (\*probe)(struct snd\_soc\_card \*card);

int (\*late\_probe)(struct snd\_soc\_card \*card);

int (\*remove)(struct snd\_soc\_card \*card);

int (\*suspend\_pre)(struct snd\_soc\_card \*card);

int (\*suspend\_post)(struct snd\_soc\_card \*card);

int (\*resume\_pre)(struct snd\_soc\_card \*card);

int (\*resume\_post)(struct snd\_soc\_card \*card);

/\*回调函数\*/

int (\*set\_bias\_level)(struct snd\_soc\_card \*,struct snd\_soc\_dapm\_context \*dapm,

enum snd\_soc\_bias\_level level);

int (\*set\_bias\_level\_post)(struct snd\_soc\_card \*,

struct snd\_soc\_dapm\_context \*dapm,enum snd\_soc\_bias\_level level);

long pmdown\_time;

/\* CPU至 Codec的DAI接口描述\*/

**struct snd\_soc\_dai\_link \*dai\_link**; /\*DAI描述，指向snd\_soc\_dai\_link实例（数组）\*/

int num\_links; /\*接口数量，snd\_soc\_dai\_link实例数量\*/

**struct snd\_soc\_pcm\_runtime \*rtd**; /\*运行时数据结构，与snd\_soc\_dai\_link实例对应\*/

int num\_rtd;

/\*可选的编解码器配置属性\*/

struct snd\_soc\_codec\_conf \*codec\_conf;

int num\_configs;

struct snd\_soc\_aux\_dev \*aux\_dev;

int num\_aux\_devs;

struct snd\_soc\_pcm\_runtime \*rtd\_aux;

int num\_aux\_rtd;

**const struct snd\_kcontrol\_new \*controls**; /\*音频控制接口及数量\*/

**int num\_controls**;

const struct snd\_soc\_dapm\_widget \*dapm\_widgets;

int num\_dapm\_widgets;

const struct snd\_soc\_dapm\_route \*dapm\_routes;

int num\_dapm\_routes;

const struct snd\_soc\_dapm\_widget \*of\_dapm\_widgets;

int num\_of\_dapm\_widgets;

const struct snd\_soc\_dapm\_route \*of\_dapm\_routes;

int num\_of\_dapm\_routes;

bool fully\_routed;

struct work\_struct deferred\_resume\_work;

/\*探测到的声卡下的设备\*/

struct list\_head codec\_dev\_list;

struct list\_head widgets;

struct list\_head paths;

struct list\_head dapm\_list;

struct list\_head dapm\_dirty;

/\* attached dynamic objects \*/

struct list\_head dobj\_list;

/\* Generic DAPM context for the card \*/

struct snd\_soc\_dapm\_context dapm;

struct snd\_soc\_dapm\_stats dapm\_stats;

struct snd\_soc\_dapm\_update \*update;

...

u32 pop\_time;

void \*drvdata;

};

snd\_soc\_card结构体主要成员简介如下：

●**snd\_card：**指向ALSA的snd\_card结构体实例，在注册snd\_soc\_card实例时创建。

●**dai\_link：**指向snd\_soc\_dai\_link结构体实例（数组），用于描述CPU侧音频接口，每个接口对应一个实例，结构体定义见下文。

●**rtd：**指向运行时snd\_soc\_pcm\_runtime结构体数组，每个snd\_soc\_dai\_link实例对应一个此实例，实例在注册snd\_soc\_card实例时创建，结构体定义见下文。

snd\_soc\_card实例需由板级相关代码定义，并定义dai\_link指向的snd\_soc\_dai\_link结构体实例（数组）。

#### ■ snd\_soc\_dai\_link

snd\_soc\_dai\_link结构体用于描述CPU侧的一个DAI音频接口信息，在板级（CPU）相关的代码中需要定义此结构体实例（或数组）。snd\_soc\_dai\_link结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_dai\_link {

/\*必须在machine驱动定义snd\_soc\_dai\_link实例时设置（初始化）\*/

const char \***name;** /\* Codec名称\*/

const char \***stream\_name**; /\*流名称\*/

const char \***cpu\_name**; /\*CPU名称\*/

struct device\_node \*cpu\_of\_node; /\*设备节点\*/

const char \***cpu\_dai\_name**; /\*CPU侧DAI名称\*/

/\*编解码侧信息，用于匹配编解码器\*/

const char \***codec\_name**; /\*编解码器名称\*/

struct device\_node \*codec\_of\_node;

const char \***codec\_dai\_name**; /\*编解码器侧DAI名称\*/

**struct snd\_soc\_dai\_link\_component \*codecs**; /\*指向snd\_soc\_dai\_link\_component实例（数组）\*/

unsigned int num\_codecs; /\*codecs指向数组项数，此处为1\*/

/\*平台侧信息，用于匹配平台\*/

const char \***platform\_name**; /\*平台名称\*/

struct device\_node \*platform\_of\_node;

int be\_id; /\* optional ID for machine driver BE identification \*/

**const struct snd\_soc\_pcm\_stream \*params**; /\*表示PCM流信息\*/

unsigned int num\_params;

unsigned int dai\_fmt; /\*初始化时格式\*/

enum snd\_soc\_dpcm\_trigger trigger[2]; /\*DPCM触发类型\*/

int (\*init)(struct snd\_soc\_pcm\_runtime \*rtd); /\*codec/machine特有的初始化\*/

/\* optional hw\_params re-writing for BE and FE sync \*/

int (\*be\_hw\_params\_fixup)(struct snd\_soc\_pcm\_runtime \*rtd,struct snd\_pcm\_hw\_params \*params);

/\*machine流操作\*/

**const struct snd\_soc\_ops \*ops**; /\*SOC声卡操作结构\*/

const struct snd\_soc\_compr\_ops \*compr\_ops;

/\*单向dai links标记\*/

bool playback\_only;

bool capture\_only;

bool nonatomic; /\*标记PCM是非原子操作\*/

/\* Keep DAI active over suspend \*/

unsigned int ignore\_suspend:1;

/\* Symmetry requirements \*/

unsigned int symmetric\_rates:1;

unsigned int symmetric\_channels:1;

unsigned int symmetric\_samplebits:1;

/\* Do not create a PCM for this DAI link (Backend link) \*/

**unsigned int no\_pcm:1**;

/\* This DAI link can route to other DAI links at runtime (Frontend)\*/

unsigned int **dynamic**:1;

/\* DPCM capture and Playback support \*/

unsigned int dpcm\_capture:1;

unsigned int dpcm\_playback:1;

/\* DPCM used FE & BE merged format \*/

unsigned int dpcm\_merged\_format:1;

/\* pmdown\_time is ignored at stop \*/

unsigned int ignore\_pmdown\_time:1;

};

snd\_soc\_dai\_link结构体主要成员简介如下：

●**codecs：**指向snd\_soc\_dai\_link\_component结构体，结构体定义如下：

struct snd\_soc\_dai\_link\_component { /\*各成员初始值来自snd\_soc\_dai\_link\*/

const char \*name; /\*编解码器名称，dai\_link->codec\_name\*/

struct device\_node \*of\_node; /\*编解码器设备节点，dai\_link->codec\_of\_node\*/

const char \*dai\_name; /\*编解码器侧DAI名称，dai\_link->codec\_dai\_name\*/

};

snd\_soc\_dai\_link\_component结构体中的成员用于查找匹配的平台、编解码器及DAI以及CPU侧DAI等。

**●params：**指向snd\_soc\_pcm\_stream结构体，表示PCM流信息，结构体定义如下：

struct snd\_soc\_pcm\_stream {

const char \*stream\_name;

u64 formats; /\* SNDRV\_PCM\_FMTBIT\_\* \*/

unsigned int rates; /\* SNDRV\_PCM\_RATE\_\* \*/

unsigned int rate\_min; /\* min rate \*/

unsigned int rate\_max; /\* max rate \*/

unsigned int channels\_min; /\* min channels \*/

unsigned int channels\_max; /\* max channels \*/

unsigned int sig\_bits; /\* number of bits of content \*/

};

**●ops：**指向snd\_soc\_ops结构体，表示操作结构，定义如下：

struct snd\_soc\_ops {

int (\*startup)(struct snd\_pcm\_substream \*);

void (\*shutdown)(struct snd\_pcm\_substream \*);

int (\*hw\_params)(struct snd\_pcm\_substream \*, struct snd\_pcm\_hw\_params \*);

int (\*hw\_free)(struct snd\_pcm\_substream \*);

int (\*prepare)(struct snd\_pcm\_substream \*);

int (\*trigger)(struct snd\_pcm\_substream \*, int);

};

snd\_soc\_ops结构体连同snd\_soc\_dai\_link结构体实例需要由板级相关代码定义。

#### ■ snd\_soc\_pcm\_runtime

snd\_soc\_pcm\_runtime结构体表示接口运行时信息，每个snd\_soc\_dai\_link实例对应一个此结构体实例，结构体定义如下（/include/sound/soc.h）：

struct snd\_soc\_pcm\_runtime {

struct device \*dev;

struct snd\_soc\_card \*card;

**struct snd\_soc\_dai\_link \*dai\_link**; /\*指向关联的snd\_soc\_dai\_link实例\*/

struct mutex pcm\_mutex;

enum snd\_soc\_pcm\_subclass pcm\_subclass;

**struct snd\_pcm\_ops ops**; /\*内嵌snd\_pcm\_ops结构体实例，赋予snd\_pcm实例\*/

unsigned int dev\_registered:1;

/\* Dynamic PCM BE runtime data \*/

struct snd\_soc\_dpcm\_runtime dpcm[2];

int fe\_compr;

long pmdown\_time;

unsigned char pop\_wait:1;

/\* runtime devices \*/

**struct snd\_pcm \*pcm**; /\*指向PCM设备\*/

struct snd\_compr \*compr;

**struct snd\_soc\_codec \*codec**; /\*指向编解码器\*/

**struct snd\_soc\_platform \*platform**; /\*指向平台\*/

**struct snd\_soc\_dai \*codec\_dai**; /\*指向编解码器侧DAI\*/

**struct snd\_soc\_dai \*cpu\_dai**; /\*指向CPU侧DAI\*/

struct snd\_soc\_component \*component; /\* Only valid for AUX dev rtds \*/

struct snd\_soc\_dai **\*\*codec\_dais**; /\*snd\_soc\_dai指针数组，指向编解码器侧DAI\*/

unsigned int num\_codecs;

struct delayed\_work delayed\_work; /\*延时工作\*/

...

};

snd\_soc\_pcm\_runtime结构体是SOC音频驱动框架的核心数据结构，它是连接SOC驱动各部分的枢纽。

### 6.6注册SOC声卡

机器（板级）相关代码需要定义snd\_soc\_card实例、snd\_soc\_dai\_link及关联的snd\_soc\_ops实例，定义表示SOC声卡设备的platform\_device实例，snd\_soc\_card实例指针赋予其dev成员的driver\_data指针成员（驱动数据结构），最后还需要注册platform\_device实例。

机器驱动程序中需要定义并注册对应的platfrom\_driver实例（通过名称匹配），platform\_device实例与platfrom\_driver实例匹配成功后，将调用其中的probe()函数（如下图所示），此函数在完成特定于设备的操作后，需要调用**snd\_soc\_register\_card()**函数，注册SOC声卡snd\_soc\_card实例，并绑定CPU侧DAI、编解码器侧DAI及平台等。



SOC音频驱动架框通用层代码中定义并注册了名称为“soc-audio”的platfrom\_driver实例，若要使用此驱动，需要将表示SOC声卡的的platform\_device实例名称设为“soc-audio”，并注册。

SOC音频驱动通用层定义的名称为“soc-audio”的platform\_driver实例如下（/sound/soc/soc-core.c）：

static struct platform\_driver **soc\_driver** = {

.driver = {

.name = **"soc-audio"**, /\*匹配platform\_device实例\*/

.pm = &snd\_soc\_pm\_ops, /\*功耗管理操作结构\*/

},

.probe = **soc\_probe,** /\*探测函数\*/

.remove = soc\_remove,

};

内核在启动阶段将调用snd\_soc\_init()函数注册soc\_driver实例，函数定义如下（/sound/soc/soc-core.c）：

static int \_\_init snd\_soc\_init(void)

{

snd\_soc\_debugfs\_init();

snd\_soc\_util\_init(); /\*注册"snd-soc-dummy"名称的platform设备和驱动，/sound/soc/soc-utils.c\*/

return **platform\_driver\_register(&soc\_driver)**; /\*注册soc\_driver实例，调用probe()函数\*/

}

module\_init(snd\_soc\_init);

平台注册的platform\_device实例与soc\_driver实例匹配后，将调用soc\_driver实例的probe()函数，即**soc\_probe()**，函数代码简列如下（/sound/soc/soc-core.c）：

static int soc\_probe(struct platform\_device \*pdev)

{

struct snd\_soc\_card \*card = platform\_get\_drvdata(pdev); /\*获取snd\_soc\_card实例\*/

... /\*错误处理\*/

card->dev = &pdev->dev; /\*device实例\*/

return **snd\_soc\_register\_card(card)**; /\*注册snd\_soc\_card实例\*/

}

下文将介绍注册SOC声卡snd\_soc\_register\_card(card)函数的实现。

#### ■注册函数

注册snd\_soc\_card实例的函数**snd\_soc\_register\_card()**定义如下（/sound/soc/soc-core.c）：

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

{

int i, j, ret;

if (!card->name || **!card->dev**) /\*card->dev不能为NULL\*/

return -EINVAL;

for (i = 0; i < card->num\_links; i++) { /\*遍历dai\_link指向的snd\_soc\_dai\_link数组项\*/

struct snd\_soc\_dai\_link \*link = &card->dai\_link[i]; /\*指向snd\_soc\_dai\_link数组项\*/

ret = **snd\_soc\_init\_multicodec(card, link)**;

/\*创建并初始化关联的snd\_soc\_dai\_link\_component实例，赋予编解码器信息\*/

...

for (j = 0; j < link->num\_codecs; j++) { /\*link->codecs[]指向snd\_soc\_dai\_link\_component实例\*/

if (!!link->codecs[j].name ==!!link->codecs[j].of\_node) {

...

}

if (!link->codecs[j].dai\_name) { /\*必须设置dai\_name名称\*/

...

}

}

... /\*错误处理\*/

} /\*遍历dai\_link指向的snd\_soc\_dai\_link数组结束\*/

dev\_set\_drvdata(**card->dev**, card);

/\*device（表示SOC声卡）驱动数据结构指向snd\_soc\_card实例\*/

snd\_soc\_initialize\_card\_lists(card);

/\*初始化snd\_soc\_card实例中双链表成员，/include/sound/soc.h\*/

**card->rtd** = devm\_kzalloc(card->dev,sizeof(struct **snd\_soc\_pcm\_runtime**) \*

(card->num\_links + card->num\_aux\_devs),GFP\_KERNEL);

/\*为每个snd\_soc\_dai\_link实例（数组）创建对应的snd\_soc\_pcm\_runtime实例（数组）\*/

...

**card->num\_rtd = 0**; /\*已初始化的snd\_soc\_pcm\_runtime实例数量\*/

card->rtd\_aux = &card->rtd[card->num\_links];

for (i = 0; i < card->num\_links; i++) { /\*遍历snd\_soc\_pcm\_runtime实例（数组项）\*/

card->rtd[i].card = card;

card->rtd[i].dai\_link = &card->dai\_link[i];

**card->rtd[i].codec\_dais** = devm\_kzalloc(card->dev,

sizeof(struct **snd\_soc\_dai** \*) \*(card->rtd[i].dai\_link->num\_codecs),GFP\_KERNEL);

/\*创建snd\_soc\_dai指针数组，指向编解码器侧snd\_soc\_dai实例\*/

if (card->rtd[i].codec\_dais == NULL)

return -ENOMEM;

}

for (i = 0; i < card->num\_aux\_devs; i++)

card->rtd\_aux[i].card = card;

INIT\_LIST\_HEAD(&card->dapm\_dirty);

INIT\_LIST\_HEAD(&card->dobj\_list);

card->instantiated = 0;

mutex\_init(&card->mutex);

mutex\_init(&card->dapm\_mutex);

ret = **snd\_soc\_instantiate\_card(card)**; /\*初始化snd\_soc\_card实例，/sound/soc/soc-core.c\*/

...

for (i = 0; i < **card->num\_rtd**; i++) { /\*遍历snd\_soc\_pcm\_runtime实例（数组）\*/

struct snd\_soc\_pcm\_runtime \*rtd = &card->rtd[i];

struct snd\_soc\_dai \*cpu\_dai = rtd->cpu\_dai; /\*CPU侧DAI\*/

int j;

for (j = 0; j < rtd->num\_codecs; j++) {

struct snd\_soc\_dai \*codec\_dai = rtd->codec\_dais[j];

if (!codec\_dai->active)

pinctrl\_pm\_select\_sleep\_state(codec\_dai->dev);

}

if (!cpu\_dai->active)

pinctrl\_pm\_select\_sleep\_state(cpu\_dai->dev);

}

return ret;

}

snd\_soc\_register\_card()函数为各snd\_soc\_dai\_link数组项创建并初始化snd\_soc\_dai\_link\_component实例，以及对应的**snd\_soc\_pcm\_runtime**实例，然后调用**snd\_soc\_instantiate\_card()**函数初始化snd\_soc\_card实例，下面将介绍此函数的实现。

#### ■初始化SOC声卡

snd\_soc\_instantiate\_card(card)函数用于初始化SOC声卡snd\_soc\_card实例，主要工作是创建并注册通用声卡snd\_card实例，基于运行时snd\_soc\_pcm\_runtime实例建立CPU侧DAI、编解码器侧DAI及平台之间的关联。snd\_soc\_instantiate\_card(card)函数调用关系如下图所示：



snd\_soc\_instantiate\_card(card)函数定义如下（/sound/soc/soc-core.c）：

static int snd\_soc\_instantiate\_card(struct snd\_soc\_card \*card)

{

struct snd\_soc\_codec \*codec;

int ret, i, order;

mutex\_lock(&client\_mutex);

mutex\_lock\_nested(&card->mutex, SND\_SOC\_CARD\_CLASS\_INIT);

/\*绑定DAIs \*/

for (i = 0; i < card->num\_links; i++) { /\*遍历snd\_soc\_dai\_link数组\*/

ret = **soc\_bind\_dai\_link(card, i)**; /\*/sound/soc/soc-core.c\*/

/\*绑定CPU侧DAI，绑定编解码器及DAI，绑定平台，根据名称查找\*/

...

}

for (i = 0; i < card->num\_aux\_devs; i++) {

ret = **soc\_bind\_aux\_dev(card, i)**;

...

}

list\_for\_each\_entry(codec, &codec\_list, list) { /\*遍历注册的编解码器\*/

if (codec->cache\_init)

continue;

ret = **snd\_soc\_init\_codec\_cache(codec)**; /\*初始化编解码器数据缓存区\*/

...

}

ret = **snd\_card\_new**(card->dev, SNDRV\_DEFAULT\_IDX1, SNDRV\_DEFAULT\_STR1,

card->owner, 0, &card->snd\_card); /\*创建snd\_card实例\*/

...

soc\_init\_card\_debugfs(card);

card->dapm.bias\_level = SND\_SOC\_BIAS\_OFF;

card->dapm.dev = card->dev;

card->dapm.card = card;

list\_add(&card->dapm.list, &card->dapm\_list);

...

#ifdef CONFIG\_PM\_SLEEP

INIT\_WORK(&card->deferred\_resume\_work, soc\_resume\_deferred);

#endif

if (card->dapm\_widgets)

snd\_soc\_dapm\_new\_controls(&card->dapm, card->dapm\_widgets,card->num\_dapm\_widgets);

if (card->of\_dapm\_widgets)

snd\_soc\_dapm\_new\_controls(&card->dapm, card->of\_dapm\_widgets,card->num\_of\_dapm\_widgets);

if (card->probe) {

**ret = card->probe(card)**; /\*调用snd\_soc\_card->probe()函数\*/

...

}

for (order = SND\_SOC\_COMP\_ORDER\_FIRST; order <= SND\_SOC\_COMP\_ORDER\_LAST;

order++) {

for (i = 0; i < card->num\_links; i++) {

ret = **soc\_probe\_link\_components(card, i, order)**; /\*/sound/soc/soc-core.c\*/

/\*探测组件，调用各关联组件的probe()函数，组件控制接口赋予SOC声卡等\*/

...

}

}

/\* probe all DAI links on this card \*/

for (order = SND\_SOC\_COMP\_ORDER\_FIRST; order <= SND\_SOC\_COMP\_ORDER\_LAST;

order++) {

for (i = 0; i < card->num\_links; i++) {

ret = **soc\_probe\_link\_dais(card, i, order)**;

/\*探测所有DAI，调用DAI驱动probe()函数，创建**snd\_pcm**设备等，见下文\*/

...

}

}

for (i = 0; i < card->num\_aux\_devs; i++) {

ret = soc\_probe\_aux\_dev(card, i);

...

}

snd\_soc\_dapm\_link\_dai\_widgets(card);

snd\_soc\_dapm\_connect\_dai\_link\_widgets(card);

if (card->controls)

**snd\_soc\_add\_card\_controls(card, card->controls, card->num\_controls)**;

/\*向声卡添加控制接口，控制接口来组件，在探测组件时赋予SOC卡声\*/

if (card->dapm\_routes)

snd\_soc\_dapm\_add\_routes(&card->dapm, card->dapm\_routes,card->num\_dapm\_routes);

if (card->of\_dapm\_routes)

snd\_soc\_dapm\_add\_routes(&card->dapm, card->of\_dapm\_routes,card->num\_of\_dapm\_routes);

... /\*输出信息\*/

for (i = 0; i < ARRAY\_SIZE(card->snd\_card->driver); i++) {

switch (card->snd\_card->driver[i]) { /\*snd\_card驱动名称字符串\*/

case '\_':

case '-':

case '\0':

break;

default:

if (!isalnum(card->snd\_card->driver[i]))

card->snd\_card->driver[i] = '\_';

break;

}

}

if (card->late\_probe) {

ret = **card->late\_probe(card);**

...

}

snd\_soc\_dapm\_new\_widgets(card);

ret = **snd\_card\_register(card->snd\_card)**; /\*注册snd\_card实例\*/

...

**card->instantiated = 1**; /\*已经初始化\*/

snd\_soc\_dapm\_sync(&card->dapm);

mutex\_unlock(&card->mutex);

mutex\_unlock(&client\_mutex);

return 0;

...

}

在初始化SOC声卡的函数中，需要特别说明的是**soc\_probe\_link\_dais(card, i, order)**函数，此函数内将探测关联的各DAI（调用DAI驱动中的probe()函数），并调用**soc\_new\_pcm()**函数为声卡创建PCM音频设备，函数代码简列如下：

int soc\_new\_pcm(struct snd\_soc\_pcm\_runtime \*rtd, int num) /\*/sound/soc/soc-pcm.c\*/

/\*num：编号\*/

{

struct snd\_soc\_platform \*platform = rtd->platform; /\*平台结构\*/

struct snd\_soc\_dai \*codec\_dai;

struct snd\_soc\_dai \*cpu\_dai = rtd->cpu\_dai; /\*CPU侧DAI\*/

struct snd\_pcm \*pcm;

char new\_name[64];

int ret = 0, playback = 0, capture = 0;

int i;

if (rtd->dai\_link->dynamic || rtd->dai\_link->no\_pcm) {

playback = rtd->dai\_link->dpcm\_playback; /\*标记位\*/

capture = rtd->dai\_link->dpcm\_capture;

} else {

for (i = 0; i < rtd->num\_codecs; i++) {

codec\_dai = rtd->codec\_dais[i];

if (codec\_dai->driver->playback.channels\_min)

playback = 1;

if (codec\_dai->driver->capture.channels\_min)

capture = 1;

}

capture = capture && cpu\_dai->driver->capture.channels\_min; /\*子流数量\*/

playback = playback && cpu\_dai->driver->playback.channels\_min;

}

if (rtd->dai\_link->playback\_only) {

playback = 1;

capture = 0;

}

if (rtd->dai\_link->capture\_only) {

playback = 0;

capture = 1;

}

/\* create the PCM \*/

if (rtd->dai\_link->no\_pcm) { /\*\*/

...

ret = snd\_pcm\_new\_internal(rtd->card->snd\_card, new\_name, num,playback, capture, &pcm);

/\*/sound/core/pcm.c\*/

} else {

...

ret = **snd\_pcm\_new**(rtd->card->snd\_card, new\_name, num, playback,capture, &pcm);

/\*创建PCM设备\*/

}

...

/\* DAPM dai link stream work \*/

INIT\_DELAYED\_WORK(&rtd->delayed\_work, close\_delayed\_work);

pcm->nonatomic = rtd->dai\_link->nonatomic;

**rtd->pcm = pcm**;

**pcm->private\_data = rtd**; /\*私有数据结构，指向snd\_soc\_pcm\_runtime实例\*/

if (rtd->dai\_link->no\_pcm) {

if (playback)

pcm->streams[SNDRV\_PCM\_STREAM\_PLAYBACK].substream->private\_data = rtd;

if (capture)

pcm->streams[SNDRV\_PCM\_STREAM\_CAPTURE].substream->private\_data = rtd;

goto out;

}

/\* ASoC PCM operations \*/

if (rtd->dai\_link->dynamic) { /\*设置rtd内嵌**snd\_pcm\_ops**结构体实例\*/

rtd->ops.open = dpcm\_fe\_dai\_open;

rtd->ops.hw\_params = dpcm\_fe\_dai\_hw\_params;

rtd->ops.prepare = dpcm\_fe\_dai\_prepare;

rtd->ops.trigger = dpcm\_fe\_dai\_trigger;

rtd->ops.hw\_free = dpcm\_fe\_dai\_hw\_free;

rtd->ops.close = dpcm\_fe\_dai\_close;

rtd->ops.pointer = soc\_pcm\_pointer;

rtd->ops.ioctl = soc\_pcm\_ioctl;

} else { /\*以下函数调用CPU侧或编解码器侧DAI驱动关联**snd\_soc\_dai\_ops**实例中的函数\*/

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

/\*调用关联平台驱动关联snd\_pcm\_ops的open()函数、CPU侧DAI驱动，

\*编解码器侧DAI驱动的snd\_soc\_dai\_ops中的startup()函数等，/sound/soc/soc-pcm.c\*/

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;

}

if (**platform->driver->ops**) {

/\*用平台驱动snd\_pcm\_ops实例成员设置rtd内嵌**snd\_pcm\_ops**结构体实例\*/

rtd->ops.ack = platform->driver->ops->ack;

rtd->ops.copy = **platform->driver->ops->copy**;

rtd->ops.silence = platform->driver->ops->silence;

rtd->ops.page = platform->driver->ops->page;

rtd->ops.mmap = platform->driver->ops->mmap;

}

/\*设置snd\_pcm设备snd\_pcm\_ops结构体实例\*/

if (playback)

**snd\_pcm\_set\_ops**(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, &**rtd->ops**);

if (capture)

**snd\_pcm\_set\_ops**(pcm, SNDRV\_PCM\_STREAM\_CAPTURE, **&rtd->ops**);

if (platform->driver->pcm\_new) {

ret = platform->driver->pcm\_new(rtd);

...

}

pcm->private\_free = platform->driver->pcm\_free;

out:

...

return ret;

}

初始化SOC声卡最终结果请参阅本小节开头驱动框架中框图。

## 7 AC97驱动

AC97（Audio Code 1997）是以Intel为首的5个PC厂商共同提出的规格标准。AC97除了是一种数据格式外，还具有控制能力，用于音频编码的内部架构规范。

AC97采用AC-Link接口与外部的编解码器相连，AC-Link接口包括位时钟BITCLK、同步信号SYNC、复位信号RESET、以及数据输入的SDATAIN和数据输出的SDATAOUT接口（串行），如下图所示。



AC-Link接口时序如下图所示：



AC-Link接口按帧传输数据，SYNC为帧同步信号，频率为48kHz（同采样频率）。SYNC为高表示一个帧的开始，每一帧包含13个时间片，第0个时间片为标签片（16bit），第1至12个时间片为数据片（每片20bit），一帧包含256bit。SYNC信号的宽度与标签片时间宽度保持一致。

AC97控制单元和AC97编解码器通过SYNC和BITCLK来决定何时发送和接收音频数据。发送端在BITCLK的上升沿将数据放到总线上，接收端在BITCLK的下降沿采样数据。发送端负责填写标签片中的有效数据地址。

AC-Link输出帧各时间片数据格式如下表所示：

|  |  |
| --- | --- |
| 时间片编号 | 说明 |
| 0 | bit15=1,表示当前帧包含最少一个有效数据片。  bit14~bit3对应12个数据片，0表示数据片无效，1表示数据片有效。  bit0和bit1为编码器IO比特。 |
| 1 | bit19=0表示写AC97编解码器寄存器，1表示读AC97编解码器寄存器。  bit18~bit2为AC97编解码器寄存器地址。 |
| 2 | bit19~bit4为命令数据。 |
| 3 | PCM回放左声道数据。 |
| 4 | PCM回放右声道数据。 |

### 7.1 AC97驱动框架

SOC AC97驱动框架如下图所示：



通用AC97编解码器驱动位于/sound/soc/codecs/ac97.c文件内，文件内定义snd\_soc\_codec\_driver编解码器驱动实例soc\_codec\_dev\_ac97，以及DAI驱动snd\_soc\_dai\_driver结构体实例ac97\_dai，定义并注册了platform\_driver结构体实例ac97\_codec\_driver，在实例的probe()函数中将注册AC97编解码器。注意编解码器的probe()函数，后面将会介绍。

在板级（处理器）代码中需要实例CPU侧DAI驱动、平台驱动，并在相应platform\_driver实例的probe()函数中注册组件（CPU侧DAI）和平台。平台驱动关联的snd\_pcm\_ops实例中的函数内，需要实现对DMA的控制，实现DAI接口控制器与内存之间的数据传输。

在板级（处理器）代码中还需要实现snd\_soc\_card实例，定义并注册名称为“soc-audio”的platform\_device实例，在名称为“soc-audio”的platform\_driver实例的probe()函数中将注册snd\_soc\_card实例（见上一小节）。

在注册snd\_soc\_card实例的过程中将查找匹配的CPU侧DAI、平台以及编解码器，调用平台驱动探测函数、编解码器驱动的探测函数等。在AC97编解码器驱动的探测函数ac97\_soc\_probe()内将创建特于编解码器的数据结构实例。

通用AC97编解码器ac97\_soc\_probe()函数调用位于/sound/pci/ac97/ac97\_codec.c文件内的接口函数创建数据结构实例。用户也可自己创建并注册AC97编解码器实例，相应的创建数据结构实例的接口函数位于/sound/soc/soc-ac97.c文件内。

AC97私有数据结构包括AC97总线。。

ac97总线代码位于/sound/ac97\_bus.c文件内（需选择AC97\_BUS配置选项），SOC音频驱动框架中AC97编解码器位于/sound/soc/codecs/ac97.c文件内，/sound/soc/soc-ac97.c

线。

### 7.2通用AC97编解码器

通用AC97编解码器在/sound/soc/codecs/ac97.c文件内，文件内定义平台总线驱动platform\_driver实例，在实例的probe()函数中将注册通用AC97编解码器，如下图所示。



platform\_driver实例如下：

static struct platform\_driver ac97\_codec\_driver = {

.driver = {

.name = "**ac97-codec**", /\*名称，匹配platform\_device实例\*/

},

.probe = **ac97\_probe**, /\*探测函数\*/

.remove = ac97\_remove,

};

若要使用通用AC97编解码器，需要在板级文件内定义并注册名称为"ac97-codec" 的platform\_device实例。

platform\_driver实例探测函数ac97\_probe()定义如下：

static int ac97\_probe(struct platform\_device \*pdev)

{

return **snd\_soc\_register\_codec**(&pdev->dev,&**soc\_codec\_dev\_ac97**, &**ac97\_dai**, 1);

/\*注册编解码器\*/

}

注册编解码器函数参数中，编解码器驱动为**soc\_codec\_dev\_ac97**，DAI驱动为**ac97\_dai**，定义如下：

static struct snd\_soc\_codec\_driver soc\_codec\_dev\_ac97 = {

.probe = **ac97\_soc\_probe**, /\*驱动探测函数，见下文\*/

.suspend = ac97\_soc\_suspend,

.resume = ac97\_soc\_resume,

.dapm\_widgets = ac97\_widgets,

.num\_dapm\_widgets = ARRAY\_SIZE(ac97\_widgets),

.dapm\_routes = ac97\_routes,

.num\_dapm\_routes = ARRAY\_SIZE(ac97\_routes),

};

DAI驱动定义如下：

static struct snd\_soc\_dai\_driver **ac97\_dai** = {

.name = "**ac97-hifi**", /\*DAI名称\*/

.playback = {

.stream\_name = "AC97 Playback",

.channels\_min = 1,

.channels\_max = 2,

.rates = SNDRV\_PCM\_RATE\_KNOT,

.formats = SND\_SOC\_STD\_AC97\_FMTS,},

.capture = {

.stream\_name = "AC97 Capture",

.channels\_min = 1,

.channels\_max = 2,

.rates = SNDRV\_PCM\_RATE\_KNOT,

.formats = SND\_SOC\_STD\_AC97\_FMTS,},

.ops = &**ac97\_dai\_ops**,

};

static const struct snd\_soc\_dai\_ops ac97\_dai\_ops = {

.prepare = **ac97\_prepare**, /\*设置速率\*/

};

### 7.3私有数据结构

AC97驱动私有数据结构如下图所示：



下面首先介绍一下AC97总线类型的定义和注册，然后介绍AC97驱动私有数据结构的定义，结构体都定义在/include/sound/ac97\_codec.h头文件。

#### ■总线类型

内核在/sound/ac97\_bus.c文件内定义了AC97总线类型，AC97总线并不只用于音频驱动，因此AC97总线的定义和注册并不依赖于音频驱动的配置选项。

AC97总线类型实例定义如下（/sound/ac97\_bus.c）：

struct bus\_type ac97\_bus\_type = {

.name = "ac97",

.match = **ac97\_bus\_match**, /\*直接返回1\*/

#ifdef CONFIG\_PM

.suspend = ac97\_bus\_suspend,

.resume = ac97\_bus\_resume,

#endif

};

内核在启动阶段将调用ac97\_bus\_init()函数注册总线实例，函数定义如下：

static int \_\_init ac97\_bus\_init(void)

{

return **bus\_register**(&ac97\_bus\_type);

}

**subsys\_initcall(ac97\_bus\_init);**

#### ■snd\_ac97

AC97驱动框架中挂载到AC97总线上的设备由snd\_ac97结构体表示，定义如下：

struct snd\_ac97 {

/\*底层驱动特性\*/

**const struct snd\_ac97\_build\_ops \*build\_ops**;

**void \*private\_data**;

void (\*private\_free) (struct snd\_ac97 \*ac97);

/\* --- \*/

**struct snd\_ac97\_bus \*bus**; /\*指向snd\_ac97\_bus\*/

struct pci\_dev \*pci; /\* assigned PCI device - used for quirks \*/

struct snd\_info\_entry \*proc;

struct snd\_info\_entry \*proc\_regs;

unsigned short subsystem\_vendor;

unsigned short subsystem\_device;

struct mutex reg\_mutex;

struct mutex page\_mutex; /\* mutex for AD18xx multi-codecs and paging (2.3) \*/

unsigned short num; /\* number of codec: 0 = primary, 1 = secondary \*/

unsigned short addr; /\* physical address of codec [0-3] \*/

unsigned int id; /\* identification of codec \*/

unsigned short caps; /\* capabilities (register 0) \*/

unsigned short ext\_id; /\* extended feature identification (register 28) \*/

unsigned short ext\_mid; /\* extended modem ID (register 3C) \*/

const struct snd\_ac97\_res\_table \*res\_table; /\* static resolution \*/

unsigned int scaps; /\* driver capabilities \*/

unsigned int flags; /\* specific code \*/

unsigned int rates[6]; /\* see AC97\_RATES\_\* defines \*/

unsigned int spdif\_status;

unsigned short regs[0x80]; /\* register cache \*/

DECLARE\_BITMAP(reg\_accessed, 0x80); /\* bit flags \*/

union { /\* vendor specific code \*/

...

} spec;

/\* jack-sharing info \*/

unsigned char indep\_surround;

unsigned char channel\_mode;

#ifdef CONFIG\_SND\_AC97\_POWER\_SAVE

unsigned int power\_up; /\* power states \*/

struct delayed\_work power\_work;

#endif

**struct device dev**; /\*挂载到AC97总线\*/

struct snd\_pcm\_chmap \*chmaps[2]; /\* channel-maps (optional) \*/

};

snd\_ac97结构体主要成员简介如下：

**●build\_ops**：snd\_ac97\_build\_ops结构体指针，结构体定义如下：

struct snd\_ac97\_build\_ops {

int (\*build\_3d) (struct snd\_ac97 \*ac97);

int (\*build\_specific) (struct snd\_ac97 \*ac97);

int (\*build\_spdif) (struct snd\_ac97 \*ac97);

int (\*build\_post\_spdif) (struct snd\_ac97 \*ac97);

#ifdef CONFIG\_PM

void (\*suspend) (struct snd\_ac97 \*ac97);

void (\*resume) (struct snd\_ac97 \*ac97);

#endif

void (\*update\_jacks) (struct snd\_ac97 \*ac97); /\* for jack-sharing \*/

};

**●bus：**指向snd\_ac97\_bus结构体，见下文。

●**dev：**device结构体实例，挂载到ac97\_bus\_type总线。

#### ■snd\_ac97\_bus

AC97总线用于音频接口时，snd\_ac97\_bus结构体用于表示音频接口属性，结构体定义如下：

struct snd\_ac97\_bus {

/\*底层硬件驱动特性\*/

**struct snd\_ac97\_bus\_ops \*ops**; /\*AC-Link接口操作结构\*/

void \*private\_data;

void (\*private\_free) (struct snd\_ac97\_bus \*bus);

/\* --- \*/

struct snd\_card \*card; /\*声卡\*/

unsigned short num; /\*总线编号\*/

unsigned short no\_vra: 1, /\* bridge doesn't support VRA \*/

dra: 1, /\* bridge supports double rate \*/

isdin: 1;/\* independent SDIN \*/

unsigned int clock; /\* AC'97 base clock (usually 48000Hz) \*/

spinlock\_t bus\_lock; /\* used mainly for slot allocation \*/

**unsigned short used\_slots[2][4]**; /\* actually used PCM slots \*/

unsigned short pcms\_count; /\* count of PCMs \*/

**struct ac97\_pcm \*pcms**;

**struct snd\_ac97 \*codec[4]**; /\*snd\_ac97结构体指针数组\*/

struct snd\_info\_entry \*proc;

};

snd\_ac97\_bus结构体主要成员简介如下：

●**ops**：指向的snd\_ac97\_bus\_ops结构体，结构体定义如下：

struct **snd\_ac97\_bus\_ops** {

void (\*reset) (struct snd\_ac97 \*ac97);

void (\*warm\_reset)(struct snd\_ac97 \*ac97);

void (\***write**) (struct snd\_ac97 \*ac97, unsigned short reg, unsigned short val); /\*写编解码器寄存器\*/

unsigned short (\***read)** (struct snd\_ac97 \*ac97, unsigned short reg); /\*读编解码器寄存器\*/

void (\*wait) (struct snd\_ac97 \*ac97);

void (\*init) (struct snd\_ac97 \*ac97);

};

snd\_ac97\_bus\_ops结构体主要包括AC97编解码器的操作函数，如读写编解码器寄存器等，结构体实例需由板级代码实现（特定于SOC）。

内核在/sound/soc/soc-ac97.c文件内定义了全局指针指向snd\_ac97\_bus\_ops实例：

struct snd\_ac97\_bus\_ops \***soc\_ac97\_ops**;

EXPORT\_SYMBOL\_GPL(soc\_ac97\_ops);

接口函数**snd\_soc\_set\_ac97\_ops**(struct snd\_ac97\_bus\_ops \*ops)用于将snd\_ac97\_bus\_ops实例赋予全局指针。

●**pcms：**指向ac97\_pcm结构体，结构体定义如下：

struct ac97\_pcm {

struct snd\_ac97\_bus \*bus;

unsigned int stream: 1, /\* stream type: 1 = capture \*/

exclusive: 1, /\* exclusive mode, don't override with other pcms \*/

copy\_flag: 1, /\* lowlevel driver must fill all entries \*/

spdif: 1; /\* spdif pcm \*/

unsigned short aslots; /\* active slots \*/

unsigned short cur\_dbl; /\* current double-rate state \*/

unsigned int rates; /\* available rates \*/

struct {

unsigned short slots; /\* driver input: requested AC97 slot numbers \*/

unsigned short rslots[4]; /\* allocated slots per codecs \*/

unsigned char rate\_table[4];

struct snd\_ac97 \*codec[4]; /\* allocated codecs \*/

} r[2]; /\* 0 = standard rates, 1 = double rates \*/

unsigned long private\_value; /\* used by the hardware driver \*/

};

### 7.4驱动探测函数

通用AC97编解码器驱动的探测函数为**ac97\_soc\_probe()**，此函数在注册snd\_soc\_card实例绑定编解码器时调用，函数内为声卡创建相应的音频设备snd\_device实例以及相应的私有数据结构实例，如下图所示。



ac97\_soc\_probe()函数定义如下（/sound/soc/codecs/ac97.c）：

static int ac97\_soc\_probe(struct snd\_soc\_codec \*codec)

{

struct snd\_ac97 \*ac97;

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

struct snd\_ac97\_template ac97\_template;

int ret;

ret = **snd\_ac97\_bus**(codec->component.card->snd\_card, 0, **soc\_ac97\_ops**,NULL, &ac97\_bus);

/\*创建snd\_ac97\_bus实例，声卡总线设备等，/sound/pci/ac97/ac97\_codec.c\*/

...

memset(&ac97\_template, 0, sizeof(struct snd\_ac97\_template));

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

/\*创建声卡mixer设备，/sound/pci/ac97/ac97\_codec.c\*/

...

**snd\_soc\_codec\_set\_drvdata(codec, ac97)**; /\*/include/sound/soc.h\*/

return 0;

}

#### ■创建总线设备

snd\_ac97\_bus()接口函数用于创建snd\_ac97\_bus实例，以及声卡的总线设备，函数定义如下：

int snd\_ac97\_bus(struct snd\_card \*card, int num, struct snd\_ac97\_bus\_ops \***ops**,

void \*private\_data, struct snd\_ac97\_bus \*\*rbus)

/\*ops：赋值全局指针**soc\_ac97\_ops**\*/

{

int err;

**struct snd\_ac97\_bus \*bus**;

static struct snd\_device\_ops dev\_ops = {

.dev\_free = snd\_ac97\_bus\_dev\_free,

};

...

bus = **kzalloc(sizeof(\*bus), GFP\_KERNEL)**; /\*分配snd\_ac97\_bus实例\*/

...

bus->card = card;

bus->num = num;

bus->ops = **ops**;

bus->private\_data = private\_data;

bus->clock = 48000;

spin\_lock\_init(&bus->bus\_lock);

snd\_ac97\_bus\_proc\_init(bus);

if ((err = **snd\_device\_new**(card, **SNDRV\_DEV\_BUS**, **bus**, &dev\_ops)) < 0) {

/\*创建声卡总线设备，私有数据指向snd\_ac97\_bus实例\*/

...

}

if (rbus)

\*rbus = bus;

return 0;

}

#### ■创建mixer设备

在编码器驱动探测函数ac97\_soc\_probe()函数中还将调用snd\_ac97\_mixer()函数创建mixer音频设备。在介绍函数代码前先看一下snd\_ac97\_template结构体的定义，它是用于创建snd\_ac97实例的一个中间数据结构。

struct snd\_ac97\_template {

void \*private\_data;

void (\*private\_free) (struct snd\_ac97 \*ac97);

struct pci\_dev \*pci; /\* assigned PCI device - used for quirks \*/

unsigned short num; /\* number of codec: 0 = primary, 1 = secondary \*/

unsigned short addr; /\* physical address of codec [0-3] \*/

unsigned int scaps; /\* driver capabilities \*/

const struct snd\_ac97\_res\_table \*res\_table; /\* static resolution \*/

};

snd\_ac97\_mixer()函数内主要是创建并设置snd\_ac97结构体实例，并创建声卡mixer音频设备，snd\_ac97实例作为snd\_device实例的私有数据构。snd\_ac97\_mixer()函数代码简列如下，省略代码主要是调用实例snd\_ac97\_bus关联snd\_ac97\_bus\_ops实例中的函数设置snd\_ac97实例。

int snd\_ac97\_mixer(struct snd\_ac97\_bus \***bus**, struct snd\_ac97\_template \*template,

struct snd\_ac97 \*\*rac97)

/\*bus：关联snd\_ac97\_bus实例指针\*/

{

int err;

struct snd\_ac97 \*ac97;

struct snd\_card \*card;

char name[64];

unsigned long end\_time;

unsigned int reg;

const struct ac97\_codec\_id \*pid;

static struct snd\_device\_ops ops = {

.dev\_free = snd\_ac97\_dev\_free,

.dev\_register = **snd\_ac97\_dev\_register**, /\*将snd\_ac97挂载到ac97\_bus\_type总线\*/

.dev\_disconnect = snd\_ac97\_dev\_disconnect,

};

if (rac97)

\*rac97 = NULL;

...

card = bus->card;

**ac97 = kzalloc(sizeof(\*ac97), GFP\_KERNEL)**; /\*创建snd\_ac97实例\*/

...

ac97->private\_data = template->private\_data;

ac97->private\_free = template->private\_free;

**ac97->bus = bus**; /\*指向snd\_ac97\_bus实例\*/

ac97->pci = template->pci;

ac97->num = template->num;

ac97->addr = template->addr;

ac97->scaps = template->scaps;

ac97->res\_table = template->res\_table;

bus->codec[ac97->num] = ac97;

mutex\_init(&ac97->reg\_mutex);

mutex\_init(&ac97->page\_mutex);

#ifdef CONFIG\_SND\_AC97\_POWER\_SAVE

INIT\_DELAYED\_WORK(&ac97->power\_work, do\_update\_power);

#endif

#ifdef CONFIG\_PCI

...

#endif

if (bus->ops->reset) {

bus->ops->reset(ac97);

goto \_\_access\_ok;

}

...

if (bus->ops->wait)

bus->ops->wait(ac97);

else {

...

}

...

pid = look\_for\_codec\_id(snd\_ac97\_codec\_ids, ac97->id);

if (pid)

ac97->flags |= pid->flags;

...

if (bus->ops->init)

bus->ops->init(ac97);

snd\_ac97\_get\_name(ac97, ac97->id, name, !ac97\_is\_audio(ac97));

snd\_ac97\_get\_name(NULL, ac97->id, name, !ac97\_is\_audio(ac97));

if (! ac97->build\_ops)

ac97->build\_ops = &null\_build\_ops;

if (ac97\_is\_audio(ac97)) {

char comp[16];

...

if ((err = **snd\_component\_add(card, comp)**) < 0) { /\*注册字符组件，/sound/core/init.c\*/

...

}

if (**snd\_ac97\_mixer\_build(ac97)** < 0) {

/\*添加控制接口，用于设备控制等，/sound/pci/ac97/ac97\_codec.c\*/

...

}

}

...

if (ac97\_is\_audio(ac97))

update\_power\_regs(ac97);

snd\_ac97\_proc\_init(ac97);

if ((err = **snd\_device\_new(card, SNDRV\_DEV\_CODEC, ac97, &ops)**) < 0) {

/\*创建mixer音频设备\*/

...

}

\*rac97 = ac97;

return 0;

}

### 7.5驱动示例

下面以龙芯1B开发板提供的内核中AC97驱动程序为例，说明音频驱动程序的实现。驱动代码位于目录/linux-3.0/sound/soc/loongson1/下，主要文件如下：

●**ls1x-ac97.c**：创建CPU侧DAI驱动snd\_soc\_dai\_driver实例，创建并注册platform\_driver实例，名称为"ls1x-ac97"，在probe()函数中注册CPU侧组件。

●**ls1x-pcm.c**：创建平台驱动snd\_soc\_platform\_driver实例，创建并注册名称为"loongson1-pcm-audio"的platform\_driver实例，在probe()函数中注册平台。

●**ls1x.c**：创建snd\_soc\_card实例，创建并注册名称"soc-audio"为的platform\_device实例，与名称为“soc-audio”的platform\_driver实例匹配，在probe()函数中注册SOC声卡。

在板级相关文件/arch/mips/loongson/ls1x/ls1b-board/platform.c内定义并注册了相关的platform\_device实例，如下：

#ifdef CONFIG\_SOUND\_LS1X\_AC97

static struct resource ls1x\_ac97\_resource[] = {

[0]={

.start = LS1X\_AC97\_BASE,

.end = LS1X\_AC97\_BASE + SZ\_16K - 1,

.flags = IORESOURCE\_MEM,

},

};

static struct platform\_device **ls1x\_audio\_device** = {

.name = **"ls1x-audio"**,

/\*匹配名称为"soc-audio"为的platform\_device实例，注册SOC声卡\*/

.id = -1,

.num\_resources = ARRAY\_SIZE(ls1x\_ac97\_resource),

.resource = ls1x\_ac97\_resource,

};

#endif

#ifdef CONFIG\_SND\_LS1X\_SOC\_AC97

static struct resource ls1x\_ac97\_resource[] = {

[0]={

.start = LS1X\_AC97\_BASE,

.end = LS1X\_AC97\_BASE + SZ\_16K - 1,

.flags = IORESOURCE\_MEM,

},

[1]={

.start = LS1X\_AC97\_IRQ,

.end = LS1X\_AC97\_IRQ,

.flags = IORESOURCE\_IRQ,

},

};

static struct platform\_device **ls1x\_ac97\_device** = {

.name = "**ls1x-ac97**",

/\*匹配名称为"ls1x-ac97"的platform\_driver实例，probe()函数注册CPU侧DAI组件\*/

.id = -1,

.num\_resources = ARRAY\_SIZE(ls1x\_ac97\_resource),

.resource = ls1x\_ac97\_resource,

};

static struct platform\_device ls1x\_stac\_dev = {

.name = "**ac97-codec**", /\*匹配编解码器驱动，注册AC97编解码器\*/

.id = -1,

};

#endif

#ifdef CONFIG\_SND\_LS1X\_SOC

static struct platform\_device ls1x\_pcm\_device = {

.name = **"loongson1-pcm-audio"**,

/\*匹配名称为"loongson1-pcm-audio"的platform\_driver实例，在probe()函数中注册平台\*/

.id = -1,

};

#endif

AC97驱动程序框架如下图所示：



各驱动定义及操作函数代码的请读者自行阅读