# 目录

[目录 1](#_Toc482955220)

[专业术语 2](#_Toc482955221)

[音频总线 2](#_Toc482955222)

[PCM(脉冲编码调制) 2](#_Toc482955223)

[IIS 3](#_Toc482955224)

[AC97 4](#_Toc482955225)

[声道,5.1声道和7.1声道有何区别? 4](#_Toc482955226)

[一、声道的定义 4](#_Toc482955227)

[二、声卡所支持的声道 4](#_Toc482955228)

[（１） 单声道Mono 4](#_Toc482955229)

[（２） 立体声Stereo 4](#_Toc482955230)

[（３） 准立体声Prospective stereo 4](#_Toc482955231)

[（４）四声环绕 5](#_Toc482955232)

[（5） 5.1声道 5](#_Toc482955233)

[（6） 7.1声道 5](#_Toc482955234)

[数据传输图 6](#_Toc482955235)

[PCM MONO Data 6](#_Toc482955236)

[PCM Stereo Data 7](#_Toc482955237)

[PCM TDM Data： 1个8组数据 7](#_Toc482955238)

[I2S Data 8](#_Toc482955239)

[I2S TDM Data：2个8组数据 9](#_Toc482955240)

[ALSA和ASOC 10](#_Toc482955241)

[ASOC的硬件结构 11](#_Toc482955242)

[ASOC的软件结构 11](#_Toc482955243)

[ASoC架构中的Machine 12](#_Toc482955244)

[注册Platform Driver 12](#_Toc482955245)

[注册Platform Device 13](#_Toc482955246)

[声卡的创建 16](#_Toc482955247)

[PCM设备的创建 18](#_Toc482955248)

[声卡作为字符设备的注册 25](#_Toc482955249)

[PCM设备的打开 26](#_Toc482955250)

[Control设备的创建 26](#_Toc482955251)

[Codec的创建 28](#_Toc482955252)

[Platform的创建 35](#_Toc482955253)

[DAI设备的创建 38](#_Toc482955254)

[DMA内存管理 39](#_Toc482955255)

[Sample: snd-soc-dummy 45](#_Toc482955256)

[ALSA设备文件结构 48](#_Toc482955257)

[寄存器配置 48](#_Toc482955258)

[录音 48](#_Toc482955259)

[I2S\_SPDIF 48](#_Toc482955260)

[ALC: 49](#_Toc482955261)

[alc5632\_snd\_controls: 49](#_Toc482955262)

[alc5632\_dapm\_widgets 50](#_Toc482955263)

# 专业术语

ASLA - Advanced Sound Linux Architecture

ASoC--ALSA System on Chip。

OSS - 以前的Linux音频体系结构，被ASLA取代并兼容

Codec - Coder/Decoder

I2S/PCM/AC97 - Codec与CPU间音频的通信协议/接口/总线

DAI - Digital Audio Interface 其实就是I2S/PCM/AC97

DAC - Digit to Analog Conversion

ADC - Analog to Digit Conversion

DSP - Digital Signal Processor

Mixer - 混音器，将来自不同通道的几种音频模拟信号混合成一种模拟信号

Mute - 消音，屏蔽信号通道

PCM - Pulse Code Modulation 一种从音频模拟信号转换成数字信号的技术，区别于PCM音频通信协议

采样频率 - ADC的频率，每秒采样的次数，典型值如44.1KHZ

量化精度 - 比如24bit，就是将音频模拟信号按照2的24次方进行等分

SSI - Serial Sound Interface

DAPM - Dynamic Audio Power Management（动态音频电源管理）

mic输入：麦克风输入

line-in输入：线路输入，通过其他音频设备采集音频信号

mono-line输入：单线输入，与其他音频设备的连接是一根线

speaker输出：扬声器输出

headphone输出：头戴式耳机输出

line-out输出：线路输出，向其他设备输出音频信号

# 音频总线

IIS的采样频率一般为44.1KHZ和48KHZ。一般传立体声。数据格式为PCM格式。IIS为一个周期传2点。

PCM采样频率一般为8K,16K，一般传单声道的声音，也可以传立体声。数据格式为PCM格式。

它们有四组信号: 位时钟信号，同步信号，数据输入，数据输出。

左（右）声道的一个点一般为16位，两个声道加起来为32位。

## PCM(脉冲编码调制)

(1) 时钟脉冲 BCLK ；BCLK的频率=2×采样频率×采样位数

(2) 帧同步信号FS；该信号为低时该帧数据有效

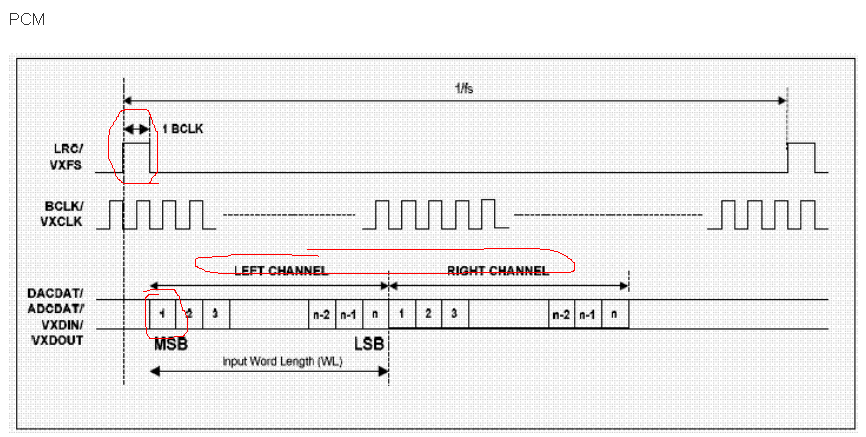
(3)接收数据DR

(4)发送数据DX

在FS信号的上升沿，数据传输从MSB（Most Significant Bit）字开始，FS频率等于采样率。

FS信号之后开始数据字的传输，单个的数据位按顺序进行传输，1个时钟周期传输1个数据字。

先传左声道数据，再传右声道数据



## IIS

IIS接口（Inter-IC Sound）在一个称为LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号变成单一的数据队列：

(1) LRCLK为高时，左声道数据被传输

(2) LRCLK为低时，右声道数据被传输

与PCM相比，IIS更适合于立体声系统。

对于多通道系统，在同样的BCLK和LRCLK条件下，并行执行几个数据队列也是可能的。

(1)分时复用数据线；两路音频信号变成单一的数据队列。

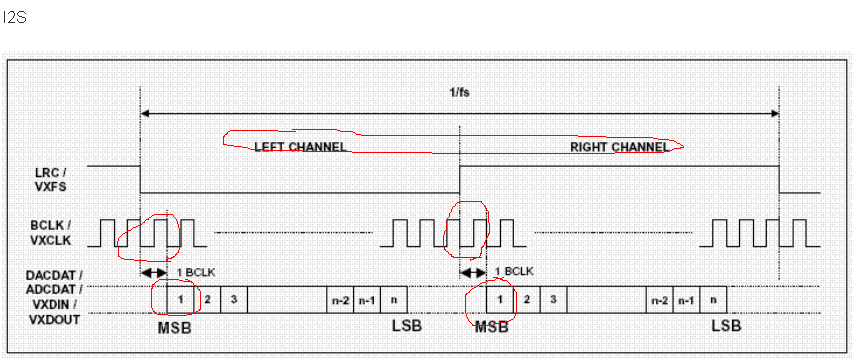
(2)字段选择线；高时传左声道数据，低传右声道数据。 采样频率：44.1K，48K.

(3)时钟信号线,SCLK；SCLK的频率=2×采样频率×采样位数

根据SDATA数据相对于LRCK和SCLK的位置不同，分为左对齐（较少使用）、I2S格式（即飞利浦规定的格式）和右对齐（也叫日本格式、普通格式）

第一个BCLK有效信号之后开始传数据

FS的信号切换标志着左右声道数据的切换



## AC97

AC97 标准把音频设备中的数字部分和模拟部分分开, 并规定数字信号处理由CPU 来负责,或者采用专门的DSP 芯片; 而模拟部分, 即A/D、D/A 转换与Mix 混音操作, 则由CODEC 芯片(Coder2Decoder: 编码/解码器) 完成。

音频处理的流程如下:

①CODEC 采样音频数据, 把模拟信号转换成数字信号通过AC2link 和AC 控制单元传送给CPU 或DSP 处理。

②CPU 对数字信号处理后交由AC 控制单元通过AC2link 传输给CODEC 进行混音处理,再转换成模拟信号输出。

AC 控制单元是AC97 CODEC 的控制器。

AC2link 是AC97 标准定义的全双工的串行接口, 负责传输音频数据、寄存器控制命令和状态信息。

AC2link 由4 根串行总线和一个nRESET 信号线组成。

串行总线分别是: 位时钟信号线(B IT\_CL K ) , 帧同步信号线(SYNC ) , 数据输出(SDA TA \_OU T ) 和数据输入(SDA TA \_IN ) 信号线。它按照AC97 rev211 标准规定的AC2link数字串口接口协议[ 1 ]进行数据的传输。AC2link在B IT \_CL K 的上升沿发送数据, 在下降沿接收数据, 实现全双工的数据传输。

# 声道,5.1声道和7.1声道有何区别?

## 一、声道的定义

　　声道(Sound Channel) 是指声音在录制或播放时在不同空间位置采集或回放的相互独立的音频信号，所以声道数也就是声音录制时的音源数量或回放时相应的扬声器数量。

## 二、声卡所支持的声道

　　声卡所支持的声道数是衡量声卡档次的重要指标之一，从单声道到最新的环绕立体声，下面一一详细介绍：

### （１） 单声道Mono

　　单声道是比较原始的声音复制形式，早期的声卡采用的比较普遍。当通过两个扬声器回放单声道信息的时候，我们可以明显感觉到声音是从两个音箱中间传递到我们耳朵里的。这种缺乏位置感的录制方式用现在的眼光看自然是很落后的，但在声卡刚刚起步时，已经是非常先进的技术了。

### （２） 立体声Stereo

单声道缺乏对声音的位置定位，而立体声技术则彻底改变了这一状况。声音在录制过程中被分配到两个独立的声道，从而达到了很好的声音定位效果。这种技术在音乐欣赏中显得尤为有用，听众可以清晰地分辨出各种乐器来自的方向，从而使音乐更富想象力，更加接近于临场感受。立体声技术广泛运用于自Sound Blaster Pro以后的大量声卡，成为了影响深远的一个音频标准。时至今日，立体声依然是许多产品遵循的技术标准。

### （３） 准立体声Prospective stereo

　　准立体声声卡的基本概念就是：在录制声音的时候采用单声道，而放音有时是立体声，有时是单声道。采用这种技术的声卡也曾在市面上流行过一段时间，但现在已经销声匿迹了。

### （４）四声环绕

　　人们的欲望是无止境的，立体声虽然满足了人们对左右声道位置感体验的要求，但是随着技术的进一步发展，大家逐渐发现双声道已经越来越不能满足我们的需求。由于PCI声卡的出现带来了许多新的技术，其中发展最为神速的当数三维音效。三维音效的主旨是为人们带来一个虚拟的声音环境，通过特殊的HRTF技术营造一个趋于真实的声场，从而获得更好的游戏听觉效果和声场定位。而要达到好的效果，仅仅依靠两个音箱是远远不够的，所以立体声技术在三维音效面前就显得捉襟见肘了，但四声道环绕音频技术则很好的解决了这一问题。

　　四声道环绕规定了4个发音点：前左、前右，后左、后右，听众则被包围在这中间。同时还建议增加一个低音音箱，以加强对低频信号的回放处理(这也就是如今4.1声道音箱系统广泛流行的原因)。就整体效果而言，四声道系统可以为听众带来来自多个不同方向的声音环绕，可以获得身临各种不同环境的听觉感受，给用户以全新的体验。如今四声道技术已经广泛融入于各类中高档声卡的设计中，成为未来发展的主流趋势。

### （5） 5.1声道

　　5.1声道已广泛运用于各类传统影院和家庭影院中，一些比较知名的声音录制压缩格式，譬如杜比AC-3（Dolby Digital）、DTS等都是以5.1声音系统为技术蓝本的，其中“.1”声道，则是一个专门设计的超低音声道，这一声道可以产生频响范围20～120Hz的超低音。其实5.1声音系统来源于4.1环绕，不同之处在于它增加了一个中置单元。这个中置单元负责传送低于80Hz的声音信号，在欣赏影片时有利于加强人声，把对话集中在整个声场的中部，以增加整体效果。



### （6） 7.1声道

　　7.1声道系统在5.1的基础上又增加了中左和中右两个发音点。简单来说就是在听者的周围建立起一套前后声场相对平衡的声场,增加了后中声场声道。

　　同时它也不同于普通6.1声道声场,因为7.1声道有双路后中置,而这双路后中置的最大作用就是为了防止听者因为没有坐在皇帝位而在听觉上产生声场的偏差。因为人的耳朵分左右两个，这时如果你的后面只有一个中置喇叭，声场就会有所偏差，这个偏差会造成有时你觉得声音是比较靠近左边，因为你左耳先收到声音，有时又会觉得声音在右边，而且声场不会有立体感，几乎是很平面的声音，听起来不对劲。

　　道理是：当你的耳朵正面不是正对着发音点时，你需要两只喇叭来修正相位差，这是为什么听音乐要至少用两只喇叭（立体声）。 所以，用两个后环绕喇叭所能营造的音场与相位是家庭影院领域里更高级的配置。但由于7.1成本比较高，没有广泛普及。

# 数据传输图

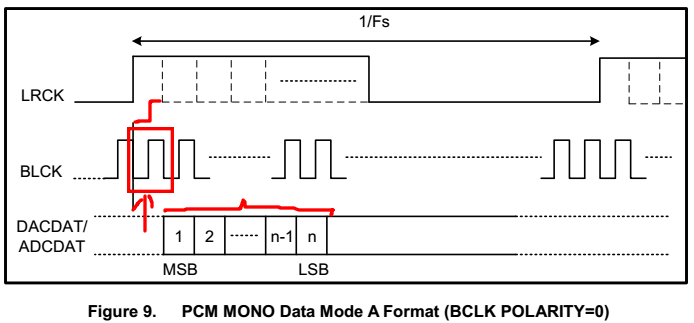
PCM数据：不同

## PCM MONO Data

PCM MONO Data: 1个LRCK周期传输1组数据。

A Format：有效数据从第二个BLCK周期开始。

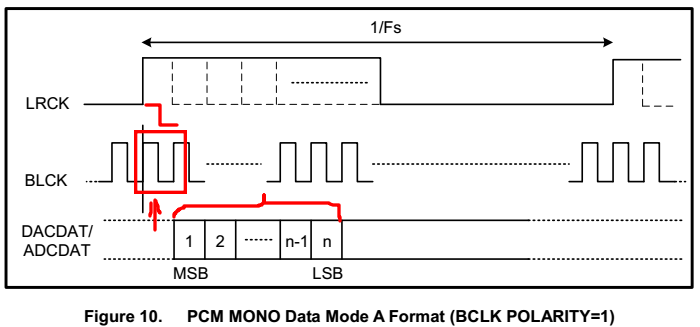
BCLK POLARITY: 0表示BLCK从低电平开始。



PCM MONO Data: 1个LRCK周期传输1组数据。

A Format：有效数据从第二个BLCK周期开始。

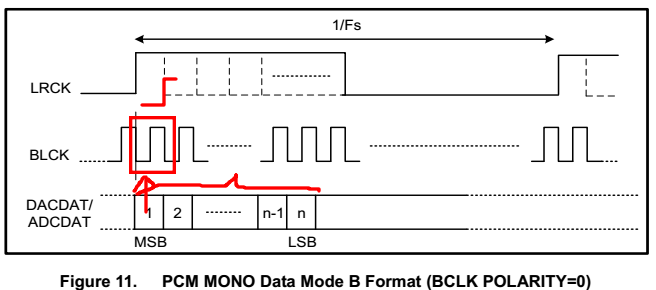
BCLK POLARITY: 1表示BLCK从高电平开始。



PCM MONO Data: 1个LRCK周期传输1组数据。

B Format：有效数据从第一个BLCK周期开始。

BCLK POLARITY: 0表示BLCK从低电平开始。

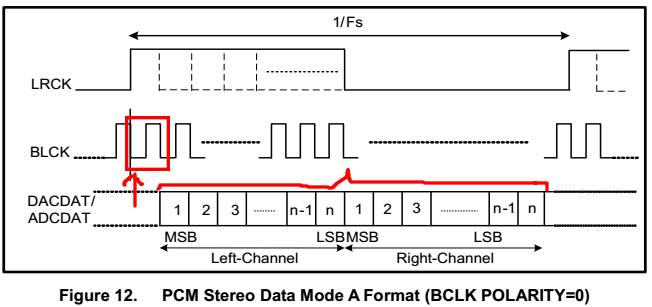


## PCM Stereo Data

PCM Stereo Data: 1个LRCK周期传输2组数据。为左右声道，且左右声道数据相连。

A Format：有效数据从第二个BLCK周期开始。

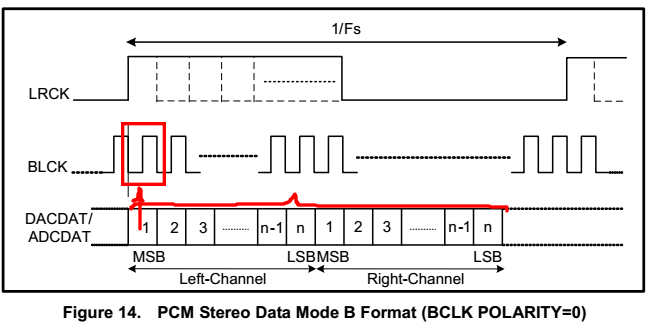
BCLK POLARITY: 0表示BLCK从低电平开始。



PCM Stereo Data: 1个LRCK周期传输2组数据。为左右声道，且左右声道数据相连。

B Format：有效数据从第一个BLCK周期开始。

BCLK POLARITY: 0表示BLCK从低电平开始。

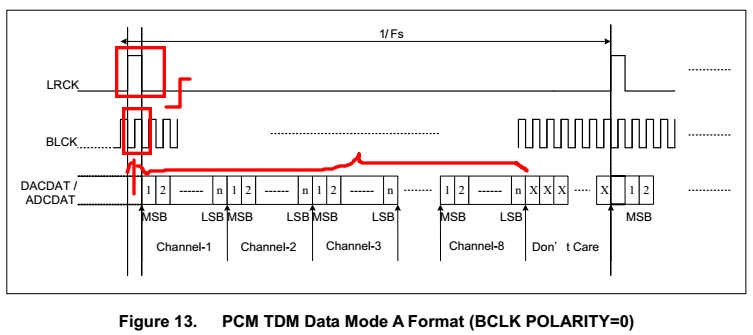


## PCM TDM Data： 1个8组数据

PCM TDM Data: 1个LRCK周期传输1个8组数据。为7.1声道，且7.1声道数据相连。

A Format：有效数据从第二个BLCK周期开始。

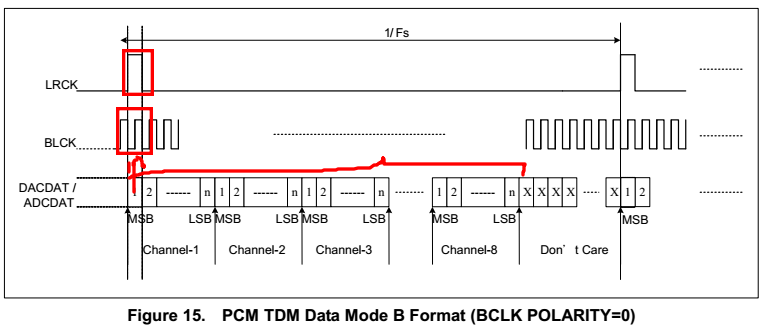
BCLK POLARITY: 0表示BLCK从低电平开始。



PCM TDM Data: 1个LRCK周期传输1个8组数据。为7.1声道，且7.1声道数据相连。

B Format：有效数据从第一个BLCK周期开始。

BCLK POLARITY: 0表示BLCK从低电平开始。

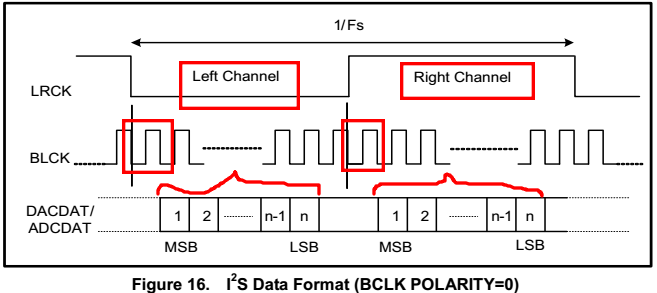


## I2S Data

I2S Data: 1个LRCK周期传输2组数据。为左右声道，且左右声道数据不相连。

Nomal Format：有效数据从第二个BLCK周期开始。

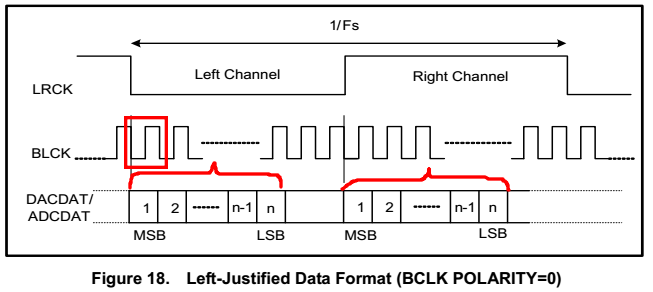
BCLK POLARITY: 0表示BLCK从低电平开始。



I2S Data: 1个LRCK周期传输2组数据。为左右声道，且左右声道数据不相连。

Left-Justified Format：有效数据从第一个BLCK周期开始。

BCLK POLARITY: 0表示BLCK从低电平开始。

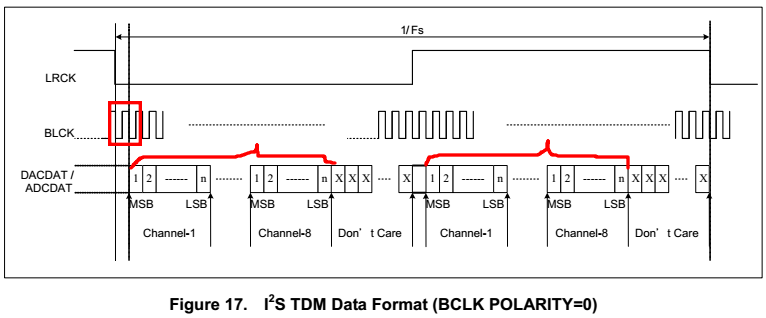


## I2S TDM Data：2个8组数据

I2S TDM Data: 1个LRCK周期传输2个8组数据。为7.1声道，且7.1声道数据不相连。

Nomal Format：有效数据从第二个BLCK周期开始。

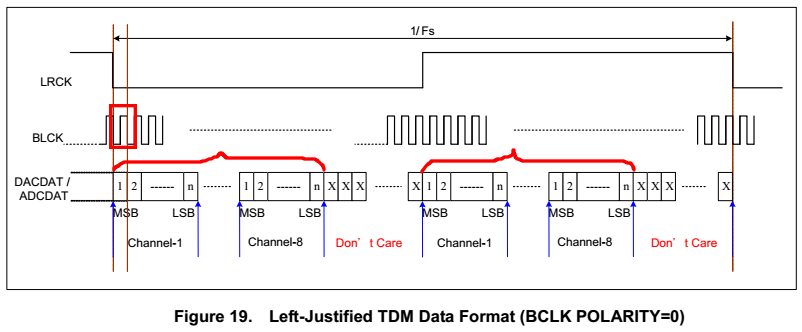
BCLK POLARITY: 0表示BLCK从低电平开始。



I2S TDM Data: 1个LRCK周期传输2个8组数据。为7.1声道，且7.1声道数据不相连。

Left-JustifiedFormat：有效数据从第一个BLCK周期开始。

BCLK POLARITY: 0表示BLCK从低电平开始。



# 声卡命令

ls /dev/snd 查看音频设备

查看声卡： cat /proc/asound/cards

配置文件: /etc/asond.conf

## 双声卡

### cat /proc/asound/cards

0 [atm9009link ]: atm9009\_link - atm9009\_link

atm9009\_link

1 [KNC220F ]: USB-Audio - KNC\_220F

Kingcome KNC\_220F at usb-xhci-hcd.0.auto-1, high speed

### ls /dev/snd/

root@gs700c:/data/test # ls -al /dev/snd/

crw-rw---- system audio 116, 0 2015-01-01 08:00 controlC0

crw-rw---- system audio 116, 32 2015-01-01 08:19 controlC1 //第二个声卡控制设备

crw-rw---- system audio 116, 24 2015-01-01 08:00 pcmC0D0c

crw-rw---- system audio 116, 16 2015-01-01 08:00 pcmC0D0p

crw-rw---- system audio 116, 17 2015-01-01 08:00 pcmC0D1p

crw-rw---- system audio 116, 18 2015-01-01 08:00 pcmC0D2p

crw-rw---- system audio 116, 19 2015-01-01 08:00 pcmC0D3p

crw-rw---- system audio 116, 56 2015-01-01 08:19 pcmC1D0c //第二个声卡capture设备

crw-rw---- system audio 116, 33 2015-01-01 08:00 timer

### # ./tinypcminfo

Usage: ./tinypcminfo -D card -d device

root@gs700c:/data/test # ./tinypcminfo -D 0 -d 0

Info for card 0, device 0: //第一个声卡信息

PCM out:

Access: 0x000009

Format[0]: 0x000004

Format[1]: 00000000

Format Name: S16\_LE

Subformat: 0x000001

Rate: min=8000Hz max=192000Hz

Channels: min=1 max=8

Sample bits: min=16 max=16

Period size: min=16 max=16384

Period count: min=2 max=16

PCM in:

Access: 0x000009

Format[0]: 0x000004

Format[1]: 00000000

Format Name: S16\_LE

Subformat: 0x000001

Rate: min=8000Hz max=96000Hz

Channels: min=1 max=4

Sample bits: min=16 max=16

Period size: min=32 max=16384

Period count: min=2 max=256

root@gs700c:/data/test # ./tinypcminfo -D 1 -d 0

Info for card 1, device 0: //第二个声卡信息

PCM out:

cannot open device '/dev/snd/pcmC1D0p'

Device does not exist.

PCM in:

Access: 0x000009

Format[0]: 0x000004

Format[1]: 0x000001

Format Name: S16\_LE, S24\_3LE

Subformat: 0x000001

Rate: min=22050Hz max=96000Hz

Channels: min=1 max=2

Sample bits: min=16 max=24

Period size: min=23 max=262144

Period count: min=2 max=1024

# ALSA和ASOC

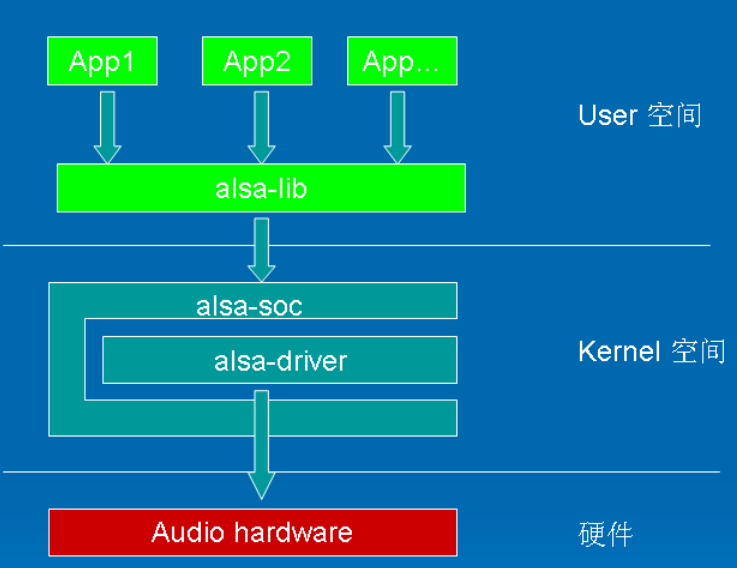
**ASoC的一切都从Machine驱动开始，包括声卡的注册，绑定Platform和Codec驱动等等**

ALSA在kernel层提供了alsa-driver，在user层提供了alsa-lib。

ASoC是建立在标准ALSA驱动层上，为了更好地支持嵌入式处理器和移动设备中的音频Codec的一套软件体系。在ASoc出现之前，内核对于SoC中的音频已经有部分的支持，不过会有一些局限性：

* Codec驱动与SoC CPU的底层耦合过于紧密，这种不理想会导致代码的重复。
* 音频事件没有标准的方法来通知用户，例如耳机、麦克风的插拔和检测，这些事件在移动设备中是非常普通的，而且通常都需要特定于机器的代码进行重新对音频路劲进行配置。
* 当进行播放或录音时，驱动会让整个codec处于上电状态，这对于PC没问题，但对于移动设备来说，这意味着浪费大量的电量。同时也不支持通过改变过取样频率和偏置电流来达到省电的目的。

ASoC正是为了解决上述种种问题而提出的，目前已经被整合至内核的代码树中：sound/soc。 ASoC不能单独存在，他只是建立在标准ALSA驱动上的一个它必须和标准的ALSA驱动框架相结合才能工作。



## ASOC的硬件结构

嵌入式设备的音频系统可以被划分为板载硬件（Machine）、Soc（Platform）、Codec三大部分：

Machine：是指某一款机器，可以是某款设备，某款开发板，又或者是某款智能手机，由此可以看出Machine几乎是不可重用的，每个Machine上的硬件实现可能都不一样，CPU不一样，Codec不一样，音频的输入、输出设备也不一样，Machine为CPU、Codec、输入输出设备提供了一个载体。

Platform：一般是指某一个SoC平台，比如pxaxxx,s3cxxxx,omapxxx等等，与音频相关的通常包含该SoC中的时钟、DMA、I2S、PCM等等，只要指定了SoC，那么我们可以认为它会有一个对应的Platform，它只与SoC相关，与Machine无关，这样我们就可以把Platform抽象出来，使得同一款SoC不用做任何的改动，就可以用在不同的Machine中。实际上，把Platform认为是某个SoC更好理解。

Codec：字面上的意思就是编解码器，Codec里面包含了I2S接口、D/A、A/D、Mixer、PA（功放），通常包含多种输入（Mic、Line-in、I2S、PCM）和多个输出（耳机、喇叭、听筒，Line-out），Codec和Platform一样，是可重用的部件，同一个Codec可以被不同的Machine使用。嵌入式Codec通常通过I2C对内部的寄存器进行控制。

## ASOC的软件结构

在软件层面，ASoC也把嵌入式设备的音频系统同样分为3大部分，Machine，Platform和Codec。

**Codec驱动：**ASoC中的一个重要设计原则就是要求Codec驱动是平台无关的，它包含了一些音频的控件（Controls），音频接口，DAMP（动态音频电源管理）的定义和某些Codec IO功能。为了保证硬件无关性，任何特定于平台和机器的代码都要移到Platform和Machine驱动中。所有的Codec驱动都要提供以下特性：

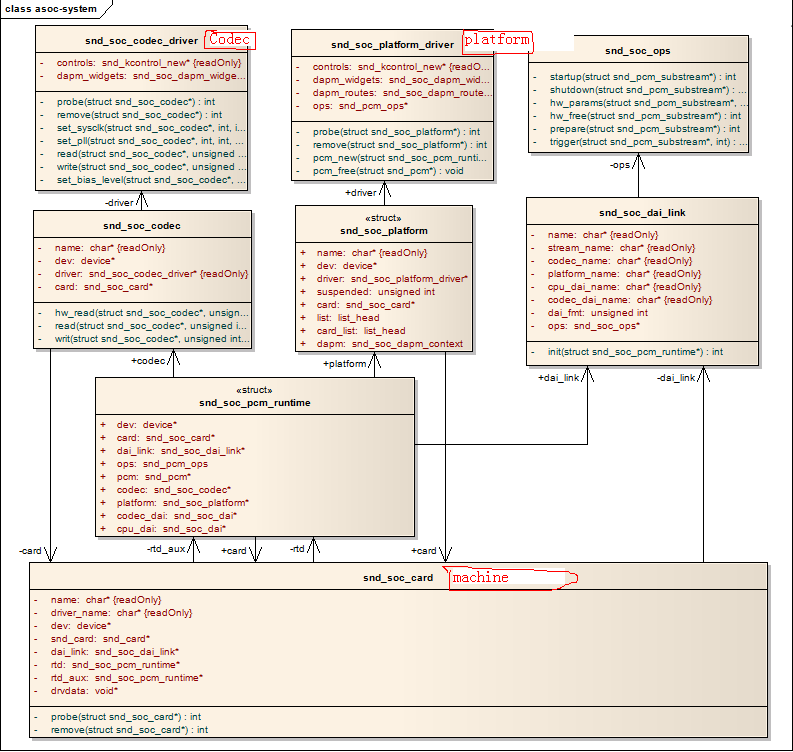
1. Codec DAI 和 PCM的配置信息；
2. Codec的IO控制方式（I2C，SPI等）；
3. Mixer和其他的音频控件；
4. Codec的ALSA音频操作接口；

必要时，也可以提供以下功能：

1. DAPM描述信息；
2. DAPM事件处理程序；
3. DAC数字静音控制

**Platform驱动：**它包含了该SoC平台的音频DMA和音频接口的配置和控制（I2S，PCM，AC97等等）；它也不能包含任何与板子或机器相关的代码。

**Machine驱动：** Machine驱动负责处理机器特有的一些控件和音频事件（例如，当播放音频时，需要先行打开一个放大器）；单独的Platform和Codec驱动是不能工作的，它必须由Machine驱动把它们结合在一起才能完成整个设备的音频处理工作。



## ASoC架构中的Machine

### 注册Platform Driver

#### struct platform\_driver owl\_atc260x\_driver

sound/soc/atc260x/link-owl.c

static struct platform\_driver **owl\_atc260x\_driver** = {

.driver = {

**.name = OWL\_SOC\_CARD,**

.owner = THIS\_MODULE,

.pm = &snd\_soc\_pm\_ops,

.of\_match\_table = owl\_atc260x\_soc\_card\_of\_match,

},

.probe = **owl\_atc260x\_probe**,

.remove = owl\_atc260x\_remove,

};

**module\_platform\_driver(owl\_atc260x\_driver);**

#### owl\_atc260x\_probe()

**直接注册snd\_soc\_card数据结构**

static int **owl\_atc260x\_probe**(struct platform\_device \*pdev)

{

struct **snd\_soc\_card** \*card = &**snd\_soc\_atm9009\_atc260x\_link**; //参见platform device

struct owl\_atc260x \***machine**;

pdev->dev.init\_name = **OWL\_SOC\_CARD; //"owl-soc-card"**

machine = **devm\_kzalloc**(&pdev->dev, sizeof(struct owl\_atc260x),GFP\_KERNEL);

platform\_set\_drvdata(pdev, card);

snd\_soc\_card\_set\_drvdata(card, machine);

card->dev = &pdev->dev;

ret = **snd\_soc\_register\_card(card)**; //参见platform device

}

### 注册Platform Device

通过snd\_soc\_card结构，又引出了Machine驱动的另外两个个数据结构：

snd\_soc\_dai\_link（实例：smdk\_dai[] ）

snd\_soc\_ops（实例：smdk\_ops ）

其中，snd\_soc\_dai\_link中，指定了Platform、Codec、codec\_dai、cpu\_dai的名字，稍后Machine驱动将会利用这些名字去匹配已经在系统中注册的platform，codec，dai，这些注册的部件都是在另外相应的Platform驱动和Codec驱动的代码文件中定义的，这样看来，Machine驱动的设备初始化代码无非就是选择合适Platform和Codec以及dai，用他们填充以上几个数据结构，然后注册Platform设备即可。当然还要实现连接Platform和Codec的dai\_link对应的ops实现，本例就是smdk\_ops，它只实现了hw\_params函数：smdk\_hw\_params。

#### struct snd\_soc\_card snd\_soc\_atm9009\_atc260x\_link

sound/soc/atc260x/link-owl.c

static struct snd\_soc\_card **snd\_soc\_atm9009\_atc260x\_link** = {

**.name = "atm9009\_link",**

.owner = THIS\_MODULE,

.dai\_link = **atm9009\_atc260x\_link\_dai**,

.num\_links = ARRAY\_SIZE(atm9009\_atc260x\_link\_dai),

.controls = owl\_atc260x\_controls,

.num\_controls = ARRAY\_SIZE(owl\_atc260x\_controls),

.dapm\_widgets = owl\_atc260x\_dapm\_widgets,

.num\_dapm\_widgets = ARRAY\_SIZE(owl\_atc260x\_dapm\_widgets),

.dapm\_routes = owl\_atc260x\_dapm\_routes,

.num\_dapm\_routes = ARRAY\_SIZE(owl\_atc260x\_dapm\_routes),

};

#### struct snd\_soc\_dai\_link atm9009\_atc260x\_link\_dai[]

static struct snd\_soc\_dai\_link **atm9009\_atc260x\_link\_dai**[] = {

{ .name = "ATM9009 ATC2603C",

**.stream\_name = "ATC2603C PCM",**

.cpu\_dai\_name = CPU\_DAI\_0\_NAME,//"owl-audio-i2s",

.codec\_dai\_name = "atc260x-dai",

.init = atm9009\_link\_snd\_init,

.platform\_name = OWL\_PCM\_PLATFORM\_NAME,//"atm9009-pcm-audio",

**.codec\_name = I2S\_AUDIO\_NAME,//"atc260x-audio",**

.ops = &atm9009\_link\_ops,},

{

.name = "ATM9009 HDMI AUDIO",

**.stream\_name = "HDMI PCM",**

.cpu\_dai\_name = CPU\_DAI\_1\_NAME,//"owl-audio-i2s",

.codec\_dai\_name = "atm9009-hdmi-dai",

.init = atm9009\_link\_snd\_init,

.platform\_name = OWL\_PCM\_PLATFORM\_NAME,//"atm9009-pcm-audio",

.**codec\_name = HDMI\_AUDIO\_NAME,//"atm9009-hdmi-audio",**

.ops = &atm9009\_link\_ops,},

{

.name = "ATM9009 SPDIF AUDIO",

**.stream\_name = "SPDIF PCM",**

.cpu\_dai\_name = CPU\_DAI\_2\_NAME,

.codec\_dai\_name = "atm9009-spdif-dai",

.init = atm9009\_link\_snd\_init,

.platform\_name = OWL\_PCM\_PLATFORM\_NAME,

**.codec\_name = SPDIF\_AUDIO\_NAME, //** **owl-spdif-audio**

.ops = &atm9009\_link\_ops,},

//this tongshi output must be the last

{

.name = "ATM9009-I2S\_HDMI\_SPDIF-AUDIO",

**.stream\_name = "I2S\_HDMI\_SPDIF PCM",**

.cpu\_dai\_name = CPU\_DAI\_3\_NAME,

.codec\_dai\_name = "atm9009-i2s\_hdmi\_spdif-dai",

.init = atm9009\_link\_snd\_init,

.platform\_name = OWL\_PCM\_PLATFORM\_NAME,

**.codec\_name = I2S\_HDMI\_SPDIF\_AUDIO\_NAME, //** **owl-i2s\_hdmi\_spdif-audio**

.ops = &atm9009\_link\_ops,},

};

#### snd\_soc\_register\_card()

sound/soc/soc-core.c

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

{

dev\_set\_drvdata(card->dev, card);

//创建(card->num\_links + card->num\_aux\_devs)个**snd\_soc\_pcm\_runtime**结构体

card->rtd = **devm\_kzalloc**(card->dev,sizeof(struct snd\_soc\_pcm\_runtime) \*

(card->num\_links + card->num\_aux\_devs),GFP\_KERNEL);

card->num\_rtd = 0;

card->rtd\_aux = &card->rtd[card->num\_links];

for (i = 0; i < card->num\_links; i++)

card->rtd[i].dai\_link = &card->dai\_link[i];

card->instantiated = 0; //snd\_soc\_card是否实例化

ret = **snd\_soc\_instantiate\_card**(card); //见声卡的创建

}

### 声卡的创建

#### snd\_soc\_instantiate\_card()

该函数首先利用card->instantiated来判断该卡是否已经实例化，如果已经实例化则直接返回，否则遍历每一对dai\_link，进行codec、platform、dai的绑定工作。

ASoC定义了三个全局的链表头变量：codec\_list、dai\_list、platform\_list，系统中所有的Codec、DAI、Platform都在注册时连接到这三个全局链表上。

soc\_bind\_dai\_link函数逐个扫描这三个链表，根据card->dai\_link[]中的名称进行匹配，匹配后把相应的codec，dai和platform实例赋值到card->rtd[]中（snd\_soc\_pcm\_runtime）。经过这个过程后，snd\_soc\_pcm\_runtime：（card->rtd）中保存了本Machine中使用的Codec，DAI和Platform驱动的信息。

完成snd\_card和snd\_pcm的创建后，接着对dapm和dai支持的格式做出一些初始化合设置工作后，调用了 card->late\_probe(card)进行一些最后的初始化合设置工作，最后则是调用标准alsa驱动的声卡注册函数对声卡进行注册

sound/soc/soc-core.c

static int **snd\_soc\_instantiate\_card**(struct snd\_soc\_card \*card)

{

//逐个扫描dai\_list、codec\_list、platform\_list链表，根据card->dai\_link[]中的名称进行匹配，

//匹配后把相应的codec，dai和platform实例赋值到card->rtd[]中

//card->num\_rtd++;

for (i = 0; i < card->num\_links; i++) ret = **soc\_bind\_dai\_link**(card, i);

//初始化Codec的寄存器缓存

list\_for\_each\_entry(codec, &codec\_list, list) snd\_soc\_init\_codec\_cache(codec, compress\_type);

#define SNDRV\_DEFAULT\_IDX1 (-1)

#define SNDRV\_DEFAULT\_STR1 NULL

//创建声卡: snd\_card

**snd\_card\_create**(SNDRV\_DEFAULT\_IDX1, SNDRV\_DEFAULT\_STR1, card->owner, 0, &card->snd\_card);

card->snd\_card->dev = card->dev;

snprintf(card->snd\_card->shortname, sizeof(), "%s", card->name);

// longname字段则会出现在/proc/asound/cards中

snprintf(card->snd\_card->longname, sizeof(), "%s", card->long\_name?card->long\_name: card->name);

snprintf(card->snd\_card->driver, sizeof(), "%s", card->driver\_name ?card->driver\_name: card->name);

//依次调用各个子结构的probe函数：

if (card->probe) ret = card->probe(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\_components(card, i, order);

//创建声卡下的PCM, 最多支持4个

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);

for (i = 0; i < card->num\_aux\_devs; i++) ret = soc\_probe\_aux\_dev(card, i);

//注册声卡，在这个阶段会遍历声卡下的所有逻辑设备，并且调用各设备的注册回调函数，

//对于pcm，就是第二步提到的snd\_pcm\_dev\_register函数，

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

//设备文件节点:/dev/snd/pcmCxxDxxp和/dev/snd/pcmCxxDxxc

ret = **snd\_card\_register**(card->snd\_card);

card->instantiated = 1;

}

#### snd\_card\_create()

sound/core/init.c

int **snd\_card\_create**(**int idx**, const char \*xid, struct module \*module, int extra\_size, struct snd\_card \*\*card\_ret)

{

struct snd\_card \*card;

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

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

**card->number = idx;**  //寻找可用的声卡序号，并修正idx的值，代码被省略

card->module = module;

//建立逻辑设备：Control

err = **snd\_ctl\_create**(card);

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

err = **snd\_info\_card\_create**(card);

if (extra\_size > 0) card->private\_data = (char \*)card + sizeof(struct snd\_card);

\*card\_ret = card;

}

#### snd\_card\_register()

**snd\_card\_register()需要等所有声卡下的逻辑设备都创建完之后执行，逻辑设备有PCM, CONTROL等**

struct snd\_card \*snd\_cards[SNDRV\_CARDS]; //全局变量

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

{

//创建sysfs下的设备：

if (!card->card\_dev)

card->card\_dev = device\_create(sound\_class, card->dev, MKDEV(0, 0), card, "card%i", card->number);

//注册所有挂在该声卡下的逻辑设备:

//通过snd\_card的devices链表，遍历所有的snd\_device，

//并且调用snd\_device的ops->dev\_register()来实现各自设备的注册

snd\_device\_register\_all(card);

snd\_cards[card->number] = card;

init\_info\_for\_card(card);

if (card->card\_dev) {

err = device\_create\_file(card->card\_dev, &card\_id\_attrs);

err = device\_create\_file(card->card\_dev, &card\_number\_attrs);

}

}

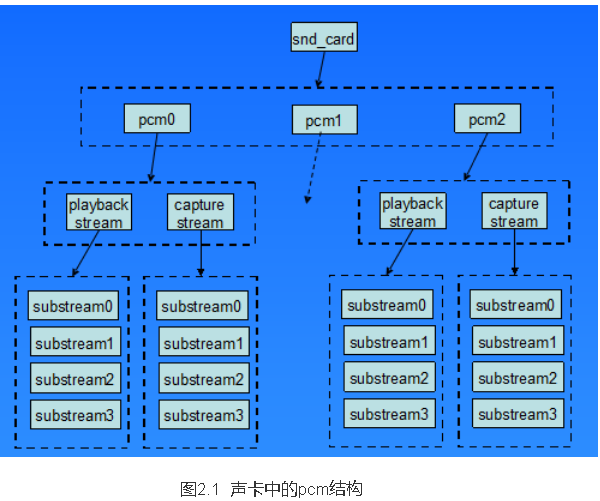
### PCM设备的创建

**snd\_device\_new(card, SNDRV\_DEV\_PCM, pcm, &ops);**

每个声卡最多可以包含4个pcm的实例，每个pcm实例对应一个pcm设备文件。

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

pcm实例数量的这种限制源于Linux设备号所占用的位大小，如果以后使用64位的设备号，我们将可以创建更多的pcm实例。



* **snd\_card\_create** pcm是声卡下的一个设备（部件），所以第一步是要创建一个声卡
* **snd\_pcm\_new** 调用该api创建一个pcm，才该api中会做以下事情

1. 如果有，建立playback stream，相应的substream也同时建立
2. 如果有，建立capture stream，相应的substream也同时建立
3. 调用snd\_device\_new()把该pcm挂到声卡中，参数ops中的dev\_register字段指向了函数snd\_pcm\_dev\_register，这个回调函数会在声卡的注册阶段被调用。

* **snd\_pcm\_set\_ops** 设置操作该pcm的控制/操作接口函数，参数中的snd\_pcm\_ops结构中的函数通常就是我们驱动要实现的函数
* **snd\_card\_register** 注册声卡，在这个阶段会遍历声卡下的所有逻辑设备，并且调用各设备的注册回调函数，对于pcm，就是第二步提到的snd\_pcm\_dev\_register函数，该回调函数建立了和用户空间应用程序（alsa-lib）通信所用的设备文件节点:/dev/snd/pcmCxxDxxp和/dev/snd/pcmCxxDxxc

#### soc\_probe\_link\_dais()

该函数出了挨个调用了codec，dai和platform驱动的probe函数外，在最后还调用了soc\_new\_pcm()函数用于创建标准alsa驱动的pcm逻辑设备。

static int **soc\_probe\_link\_dais**(struct snd\_soc\_card \*card, int num, int order)

{

if (!cpu\_dai->probed && cpu\_dai->driver->probe\_order == order) {

if (cpu\_dai->driver->probe) ret = cpu\_dai->driver->probe(cpu\_dai);

cpu\_dai->probed = 1;

list\_add(&cpu\_dai->card\_list, &card->dai\_dev\_list); }

if (!codec\_dai->probed && codec\_dai->driver->probe\_order == order) {

if (codec\_dai->driver->probe) ret = codec\_dai->driver->probe(codec\_dai);

codec\_dai->probed = 1;

list\_add(&codec\_dai->card\_list, &card->dai\_dev\_list);}

ret = soc\_post\_component\_init(card, codec, num, 0);

if (cpu\_dai->driver->compress\_dai) {

ret = soc\_new\_compress(rtd, num);

} else {

if (!dai\_link->params) {

ret = **soc\_new\_pcm**(rtd, num); //创建标准alsa驱动的pcm逻辑设备

} else {

**/\* link the DAI widgets \*/**

play\_w = codec\_dai->playback\_widget;

capture\_w = cpu\_dai->capture\_widget;

if (play\_w && capture\_w)

ret = snd\_soc\_dapm\_new\_pcm(card, dai\_link->params,capture\_w, play\_w);

play\_w = cpu\_dai->playback\_widget;

capture\_w = codec\_dai->capture\_widget;

if (play\_w && capture\_w)

ret = snd\_soc\_dapm\_new\_pcm(card, dai\_link->params,capture\_w, play\_w);

}

}

**/\* add platform data for AC97 devices \*/**

if (rtd->codec\_dai->driver->ac97\_control)

snd\_ac97\_dev\_add\_pdata(codec->ac97, rtd->cpu\_dai->ac97\_pdata);

}

#### soc\_new\_pcm()

sound/soc/soc-pcm.c

该函数首先初始化snd\_soc\_runtime中的snd\_pcm\_ops字段，也就是rtd->ops中的部分成员，例如open，close，hw\_params等，

紧接着调用标准alsa驱动中的创建pcm的函数snd\_pcm\_new()创建声卡的pcm实例，pcm的private\_data字段设置为该runtime变量rtd，

然后用platform驱动中的snd\_pcm\_ops替换部分pcm中的snd\_pcm\_ops字段，

最后，调用platform驱动的pcm\_new回调，该回调实现该platform下的dma内存申请和dma初始化等相关工作。到这里，声卡和他的pcm实例创建完成。

int **soc\_new\_pcm**(struct snd\_soc\_pcm\_runtime \*rtd, int num)

{

int playback = 0, capture = 0;

if (codec\_dai->driver->playback.channels\_min && cpu\_dai->driver->playback.channels\_min) playback = 1;

if (codec\_dai->driver->capture.channels\_min && cpu\_dai->driver->capture.channels\_min) capture = 1;

snprintf(new\_name, sizeof(), "%s %s-%d", rtd->dai\_link->stream\_name, codec\_dai->name, num);

//创建PCM及PCM下的playback和capture

ret = **snd\_pcm\_new(**rtd->card->snd\_card, new\_name, num, **playback, capture**, &pcm);

**/\* DAPM dai link stream work \*/**

INIT\_DELAYED\_WORK(&rtd->delayed\_work, close\_delayed\_work);

rtd->pcm = pcm;

pcm->private\_data = rtd;

**/\* ASoC PCM operations \*/**

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

**rtd->ops.hw\_params = soc\_pcm\_hw\_params;**

rtd->ops.prepare = soc\_pcm\_prepare;

rtd->ops.trigger = soc\_pcm\_trigger;

rtd->ops.hw\_free = soc\_pcm\_hw\_free;

rtd->ops.close = soc\_pcm\_close;

rtd->ops.pointer = soc\_pcm\_pointer;

**rtd->ops.ioctl = soc\_pcm\_ioctl;**

rtd->ops.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;

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); //实现该platform下的dma内存申请和dma初始化等

pcm->private\_free = platform->driver->pcm\_free;

}

#### snd\_pcm\_new() :创建PCM

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

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

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

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

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;

static struct snd\_device\_ops ops = {

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

**.dev\_register = snd\_pcm\_dev\_register,**

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

};

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

pcm->card = card;

pcm->device = device;

pcm->internal = internal;

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

snd\_pcm\_new\_stream(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK, playback\_count);

snd\_pcm\_new\_stream(pcm, SNDRV\_PCM\_STREAM\_CAPTURE, capture\_count);

snd\_device\_new(card, SNDRV\_DEV\_PCM, pcm, &ops);

}

#### snd\_pcm\_dev\_register()

static LIST\_HEAD(snd\_pcm\_devices); //全局变量

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

{

//将pcm添加到snd\_pcm\_devices链表中;

struct snd\_pcm \*pcm = device->device\_data;

err = snd\_pcm\_add(pcm);

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

switch (cidx) {

case SNDRV\_PCM\_STREAM\_PLAYBACK:

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

devtype = SNDRV\_DEVICE\_TYPE\_PCM\_**PLAYBACK**;

break;

case SNDRV\_PCM\_STREAM\_CAPTURE:

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

devtype = SNDRV\_DEVICE\_TYPE\_PCM\_**CAPTUR**E;

break;

}

/\* register pcm \*/

snd\_register\_device\_for\_dev(devtype, pcm->card,pcm->device,&**snd\_pcm\_f\_ops[cidx]**,pcm, str, dev);

snd\_add\_device\_sysfs\_file(devtype, pcm->card, pcm->device,&pcm\_attrs);

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

snd\_pcm\_timer\_init(substream);

}

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

notify->n\_register(pcm);

}

#### snd\_register\_device\_for\_dev()

每个snd\_minor结构体保存了声卡下某个逻辑设备的上下文信息，他在逻辑设备建立阶段被填充，在逻辑设备被使用时就可以从该结构体中得到相应的信息。

#define SNDRV\_OS\_MINORS 256

static struct snd\_minor \*snd\_minors[SNDRV\_OS\_MINORS];

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

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

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

{

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

preg->type = type;

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

preg->device = dev;

preg->f\_ops = f\_ops;

preg->private\_data = private\_data;

preg->card\_ptr = card;

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

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

snd\_minors[minor] = preg;

//调用device\_create创建设备节点

preg->dev = device\_create(sound\_class, device, MKDEV(major, minor), private\_data, "%s", name);

}

#### struct file\_operations snd\_pcm\_f\_ops [2]

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

{

.owner = THIS\_MODULE,

.write = snd\_pcm\_write,

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

.open = snd\_pcm\_playback\_open,

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_playback\_poll,

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

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

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

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

},

{

.owner = THIS\_MODULE,

.read = snd\_pcm\_read,

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

.open = snd\_pcm\_capture\_open,

.release = snd\_pcm\_release,

.llseek = no\_llseek,

.poll = snd\_pcm\_capture\_poll,

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

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

.mmap = snd\_pcm\_mmap,

.fasync = snd\_pcm\_fasync,

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

}

};

### 声卡作为字符设备的注册

#### alsa\_sound\_init()

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

**subsys\_initcall**(alsa\_sound\_init);

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

{

snd\_major = major;

snd\_ecards\_limit = cards\_limit;

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

snd\_info\_init();

snd\_info\_minor\_register();

}

#### struct file\_operations snd\_fops

static const struct file\_operations snd\_fops =

{

.owner = THIS\_MODULE,

.open = snd\_open,

.llseek = noop\_llseek,

};

### PCM设备的打开

#### snd\_open()

最终跳到**snd\_pcm\_f\_ops[]中的open：** snd\_pcm\_playback\_open， snd\_pcm\_capture\_open

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

{

unsigned int minor = iminor(inode);

struct snd\_minor \*mptr = snd\_minors[minor];

old\_fops = file->f\_op;

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

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

fops\_put(old\_fops);

}

### Control设备的创建

**snd\_device\_new(card, SNDRV\_DEV\_CONTROL, card, &ops);**

Control设备和PCM设备一样，都属于声卡下的逻辑设备。用户空间的应用程序通过alsa-lib访问该Control设备，读取或控制control的控制状态，从而达到控制音频Codec进行各种Mixer等控制操作。

snd\_card\_create()通过调用snd\_ctl\_create()函数创建control设备节点。所以我们无需显式地创建control设备，只要建立声卡，control设备被自动地创建。

和pcm设备一样，control设备的名字遵循一定的规则：controlCxx，这里的xx代表声卡的编号。

#### snd\_ctl\_create()

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,

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

};

return snd\_device\_new(card, SNDRV\_DEV\_CONTROL, card, &ops);

}

#### snd\_ctl\_dev\_register()

snd\_ctl\_dev\_register()函数会在snd\_card\_register()中，即声卡的注册阶段被调用。注册完成后，control设备的相关信息被保存在snd\_minors[]数组中，用control设备的此设备号作索引，即可在snd\_minors[]数组中找出相关的信息。

snd\_register\_device()会调用snd\_register\_device\_for\_dev()函数，参见”PCM设备的创建”

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

{

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

cardnum = card->number;

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

snd\_register\_device(SNDRV\_DEVICE\_TYPE\_CONTROL, card, -1, &snd\_ctl\_f\_ops, card, name);

}

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

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

{

.owner = THIS\_MODULE,

.read = snd\_ctl\_read,

**.open = snd\_ctl\_open,**

.release = snd\_ctl\_release,

.llseek = no\_llseek,

.poll = snd\_ctl\_poll,

.unlocked\_ioctl = snd\_ctl\_ioctl,

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

.fasync = snd\_ctl\_fasync,

};

### Codec的创建

定义snd\_soc\_codec\_driver和snd\_soc\_dai\_driver的实例，并将这两个实例添加到Codec\_list和dai\_list链表中。

#### static struct platform\_driver atc260x\_platform\_driver

static struct platform\_driver **atc260x\_platform\_driver** = {

.probe = atc260x\_platform\_probe,

.remove = atc260x\_platform\_remove,

.driver = {

.name = I2S\_AUDIO\_NAME,//"atc2603c-audio",

.owner = THIS\_MODULE,

.of\_match\_table = atc260x\_audio\_of\_match,

},

.shutdown = atc260x\_platform\_shutdown,

};

**module\_platform\_driver**(atc260x\_platform\_driver);

#### atc260x\_platform\_probe()

Codec驱动的第一个步骤就是定义snd\_soc\_codec\_driver和snd\_soc\_dai\_driver的实例，然后调用snd\_soc\_register\_codec函数对Codec进行注册。

static int **atc260x\_platform\_probe**(struct platform\_device \*pdev)

{

struct atc260x\_dev \*atc260x = dev\_get\_drvdata(pdev->dev.parent);

atc260x\_dev\_ic\_type = atc260x\_get\_ic\_type(atc260x);

switch (atc260x\_dev\_ic\_type) {

case ATC260X\_ICTYPE\_2603A:

break;

case ATC260X\_ICTYPE\_2603C:

//根据不同Codec修正soc\_codec\_atc260x的入口参数

atc260x\_codec\_plugin = &atc2603c\_codec\_plugin;

set\_soc\_codec\_driver\_attr(&soc\_codec\_atc260x, atc260x\_codec\_plugin);

break;

case ATC260X\_ICTYPE\_2609A:

break;

default:

}

platform\_set\_drvdata(pdev, atc260x);

pdev->dev.init\_name = I2S\_AUDIO\_NAME; // "atc260x-audio"

snd\_soc\_register\_codec(&pdev->dev, &soc\_codec\_atc260x, codec\_atc260x\_dai, ARRAY\_SIZE());

}

#### struct snd\_soc\_codec\_driver soc\_codec\_atc260x

snd\_soc\_instantiate\_cards()中的soc\_bind\_dai\_link函数逐个扫描codec\_list、dai\_list、platform\_list这三个链表，根据card->dai\_link[]中的名称进行匹配，匹配后把相应的codec，dai和platform实例赋值到card->rtd[]中，最后调用各实例的driver->probe()

static struct **snd\_soc\_codec\_driver** soc\_codec\_atc260x = {

.probe = atc260x\_probe,

.remove = atc260x\_remove,

.suspend = atc260x\_suspend,

.resume = atc260x\_resume,

.set\_bias\_level = atc260x\_set\_bias\_level,

.idle\_bias\_off = true,

};

void **set\_soc\_codec\_driver\_attr**(struct snd\_soc\_codec\_driver \*soc\_codec\_atc260x,

struct atc260x\_audio\_codec\_plugin \*atc260x\_codec\_plugin)

{

soc\_codec\_atc260x->reg\_cache\_size = atc260x\_codec\_plugin->reg\_cache\_size;

soc\_codec\_atc260x->reg\_word\_size = sizeof(u16);

soc\_codec\_atc260x->reg\_cache\_default = atc260x\_codec\_plugin->reg\_cache\_default;

soc\_codec\_atc260x->volatile\_register = atc260x\_codec\_plugin->volatile\_register;

soc\_codec\_atc260x->readable\_register = atc260x\_codec\_plugin->readable\_register;

soc\_codec\_atc260x->reg\_cache\_step = 1;

soc\_codec\_atc260x->controls = atc260x\_codec\_plugin->controls;

soc\_codec\_atc260x->num\_controls = atc260x\_codec\_plugin->num\_controls;

soc\_codec\_atc260x->dapm\_widgets = atc260x\_codec\_plugin->dapm\_widgets;

soc\_codec\_atc260x->num\_dapm\_widgets = atc260x\_codec\_plugin->num\_dapm\_widgets;

soc\_codec\_atc260x->dapm\_routes = atc260x\_codec\_plugin->dapm\_routes;

soc\_codec\_atc260x->num\_dapm\_routes = atc260x\_codec\_plugin->num\_dapm\_routes;

}

#### struct atc260x\_audio\_codec\_plugin atc2603c\_codec\_plugin

struct **atc260x\_audio\_codec\_plugin** atc2603c\_codec\_plugin = {

.set\_dai\_digital\_mute = atc2603c\_audio\_set\_dai\_digital\_mute,

.hw\_params = atc2603c\_audio\_hw\_params,

.hw\_free = atc2603c\_audio\_hw\_free,

.prepare = atc2603c\_audio\_prepare,

.set\_dai\_fmt = atc2603c\_audio\_set\_dai\_fmt,

.set\_dai\_sysclk = atc2603c\_audio\_set\_dai\_sysclk,

.shutdown = atc2603c\_audio\_shutdown,

.probe = atc2603c\_probe,

.remove = atc2603c\_remove,

.suspend = atc2603c\_suspend,

.resume = atc2603c\_resume,

.reg\_cache\_size = ARRAY\_SIZE(atc2603c\_reg),//get\_atc2603c\_reg\_cache\_size,

.reg\_cache\_default = &atc2603c\_reg[0],//get\_atc2603c\_reg\_cache\_addr,

.volatile\_register = atc2603c\_volatile\_register,

.readable\_register =atc2603c\_readable\_register,

.controls = &atc2603c\_snd\_controls[0],

.num\_controls = ARRAY\_SIZE(atc2603c\_snd\_controls),

.dapm\_widgets = atc2603c\_dapm\_widgets,

.num\_dapm\_widgets = ARRAY\_SIZE(atc2603c\_dapm\_widgets),

.dapm\_routes = atc2603c\_dapm\_routes,

.num\_dapm\_routes = ARRAY\_SIZE(atc2603c\_dapm\_routes),

.platform\_shutdown = atc2603c\_platform\_shutdown,

.dump\_pmu\_audio\_regs = dump\_2603c\_regs,

.mic0\_set\_mode = atc2603c\_mic0\_set\_mode,

};

#### struct snd\_soc\_dai\_driver codec\_atc260x\_dai

snd\_soc\_dai\_driver 该结构需要自己根据不同的soc芯片进行定义，关键字段介绍如下：

* probe、remove 回调函数，分别在声卡加载和卸载时被调用；
* suspend、resume 电源管理回调函数；
* ops 指向snd\_soc\_dai\_ops结构，用于配置和控制该dai；
* playback snd\_soc\_pcm\_stream结构，用于指出该dai支持的声道数，码率，数据格式等能力；
* capture snd\_soc\_pcm\_stream结构，用于指出该dai支持的声道数，码率，数据格式等能力；

struct **snd\_soc\_dai\_driver** codec\_atc260x\_dai[] = {

{

.name = "atc260x-dai",

.id = ATC260X\_AIF,

.playback = {

.stream\_name = "AIF Playback",

.channels\_min = 1,

.channels\_max = 8,

.rates = ATC260x\_RATES,

.formats = ATC260x\_FORMATS,

},

.capture = {

.stream\_name = "AIF Capture",

.channels\_min = 1,

.channels\_max = 4,

.rates = ATC260x\_RATES,

.formats = ATC260x\_FORMATS,

},

.ops = &atc260x\_aif\_dai\_ops,

},

}

#### struct snd\_soc\_dai\_ops atc260x\_aif\_dai\_ops

工作时钟配置函数 通常由machine驱动调用：

* + set\_sysclk 设置dai的主时钟；
  + set\_pll 设置PLL参数；
  + set\_clkdiv 设置分频系数；
  + dai的格式配置函数 通常由machine驱动调用：
  + set\_fmt 设置dai的格式；
  + set\_tdm\_slot 如果dai支持时分复用，用于设置时分复用的slot；
  + set\_channel\_map 声道的时分复用映射设置；
  + set\_tristate 设置dai引脚的状态，当与其他dai并联使用同一引脚时需要使用该回调；

标准的snd\_soc\_ops回调 通常由soc-core在进行PCM操作时调用：

* + startup
  + shutdown
  + hw\_params
  + hw\_free
  + prepare
  + trigger

抗pop，pop声 由soc-core调用：

* digital\_mute

struct **snd\_soc\_dai\_ops** atc260x\_aif\_dai\_ops = {

.shutdown = atc260x\_audio\_shutdown,

.hw\_params = atc260x\_audio\_hw\_params,

.hw\_free = atc260x\_audio\_hw\_free,

.prepare = atc260x\_audio\_prepare,

.set\_fmt = atc260x\_audio\_set\_dai\_fmt,

.set\_sysclk = atc260x\_audio\_set\_dai\_sysclk, //设置dai的主时钟

.digital\_mute = atc260x\_audio\_set\_dai\_digital\_mute,

};

#### snd\_soc\_register\_codec()： codec\_list链表

确定codec的名字，这个名字很重要，Machine驱动定义的snd\_soc\_dai\_link中会指定每个link的codec和dai的名字，进行匹配绑定时就是通过和这里的名字比较，从而找到该Codec的！

static LIST\_HEAD(codec\_list);

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)

{

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

codec->name = fmt\_single\_name(dev, &codec->id); //确定codec的名字

codec->write = codec\_drv->write;

codec->read = codec\_drv->read;

codec->volatile\_register = codec\_drv->volatile\_register;

codec->readable\_register = codec\_drv->readable\_register;

codec->writable\_register = codec\_drv->writable\_register;

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

codec->dapm.bias\_level = SND\_SOC\_BIAS\_OFF;

codec->dapm.dev = dev;

codec->dapm.codec = codec;

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

codec->dapm.stream\_event = codec\_drv->stream\_event;

codec->dev = dev;

codec->driver = codec\_drv;

codec->num\_dai = num\_dai;

//把codec实例链接到全局链表codec\_list中

list\_add(&codec->list, &codec\_list);

//对本Codec的dai进行注册

ret = snd\_soc\_register\_dais(dev, dai\_drv, num\_dai);

}

#### snd\_soc\_register\_dais()： dai\_list链表

static LIST\_HEAD(dai\_list);

static int **snd\_soc\_register\_dais**(struct device \*dev, struct snd\_soc\_dai\_driver \*dai\_drv, size\_t count)

{

struct snd\_soc\_dai \*dai;

for (i = 0; i < count; i++) {

dai = kzalloc(sizeof(struct snd\_soc\_dai), GFP\_KERNEL);

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

dai->dev = dev;

dai->driver = &dai\_drv[i];

dai->id = i;

dai->dapm.dev = dev;

list\_for\_each\_entry(codec, &codec\_list, list) {

if (codec->dev == dev) {

dai->codec = codec;

break;

}

}

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

}

}

#### atc260x\_probe()

snd\_soc\_instantiate\_cards()中的soc\_bind\_dai\_link函数逐个扫描codec\_list、dai\_list、platform\_list这三个链表，根据card->dai\_link[]中的名称进行匹配，匹配后把相应的codec，dai和platform实例赋值到card->rtd[]中，最后调用各实例的driver->probe()

static int **atc260x\_probe**(struct snd\_soc\_codec \*codec)

{

atc260x\_codec\_plugin->probe(codec); //指向atc2603c\_probe()

atc260x\_codec\_plugin->atc260x\_codec = codec;

}

static int **atc2603c\_probe**(struct snd\_soc\_codec \*codec)

{

atc2603c\_get\_cfg(); //获取DTS中的配置，并保存到全局变量audio\_hw\_cfg中

codec->read = atc2603c\_read;

codec->write = atc2603c\_write;

reenable\_audio\_block(codec);//set hw\_init\_flag in the func

atc2603c\_pa\_up\_all(codec);

}

#### DTS: atc260x-audio

atc260x-audio{

compatible = "actions,atc2603c-audio";

status = "okay";

clocks = <&clock CLK\_AUDIO\_PLL>, <&clock CLK\_HDMI\_AUDIO>;

clock-names = "audio\_pll", "hdmia";

//earphone\_detect\_gpios = <&gpiob 11 0>; /\*0: high, 1: low hr280ac GPIOB11 0\*/

//speaker\_en\_gpios = <&gpioa 19 1>; /\*hr280ac GPIOA(19) 0: high, 1: low\*/

earphone\_output\_mode = <1>; /\*0: direct mode, 1: undirect mode\*/

mic\_num = <1>; /\*just suport 1, 2\*/

mic0\_gain = < 0x7 0x7 >; /\*<Lchannel,Rchannel> [0x0~0xF]\*/

speaker\_gain = < 0xaa 0xaa >; /\*<Lchannel,Rchannel>[0x00~0xbebe]\*/

earphone\_gain = < 0xbe 0xbe >; /\*<Lchannel,Rchannel>[0x00~0xbebe]\*/

speaker\_volume = < 0x28 0x28 >; /\*volume [0x00~0x28]\*/

earphone\_volume = < 0x28 0x28 >; /\*volume [0x00~0x28]\*/

earphone\_detect\_mode = <0>; /\*0: polling, 1: interrupt\*/

};

### Platform的创建

定义**snd\_soc\_platform\_driver**实例，并将这实例添加到platform\_list链表中。

#### struct platform\_driver atm9009\_pcm\_driver

static struct **platform\_driver** atm9009\_pcm\_driver = {

.driver = {

.name = OWL\_PCM\_PLATFORM\_NAME,//"atm9009-pcm-audio",

.owner = THIS\_MODULE,

.of\_match\_table = owl\_pcm\_platform\_of\_match,

},

.probe = atm9009\_pcm\_probe,

.remove = atm9009\_pcm\_remove,

};

**module\_platform\_driver**(atm9009\_pcm\_driver);

#### atm9009\_pcm\_probe()

pcm\_priv->output\_mode: I2S, HDMI, PDIF, I2S\_HDMI\_SPDIF

static int **atm9009\_pcm\_probe**(struct platform\_device \*pdev)

{

struct atm9009\_pcm\_priv \*pcm\_priv = kzalloc(sizeof(), GFP\_KERNEL);

pcm\_priv->output\_mode = O\_MODE\_I2S;

platform\_set\_drvdata(pdev, pcm\_priv);

pdev->dev.init\_name = OWL\_PCM\_PLATFORM\_NAME;//"atm9009-pcm-audio";

snd\_soc\_register\_platform(&pdev->dev, &atm9009\_soc\_platform);

}

#### struct snd\_soc\_platform\_driver atm9009\_soc\_platform

static struct **snd\_soc\_platform\_driver** atm9009\_soc\_platform = {

.ops = &atm9009\_pcm\_ops,

.pcm\_new = atm9009\_pcm\_new, // 参见DMA内存管理

.pcm\_free = atm9009\_pcm\_free\_dma\_buffers,

.controls = atm9009\_pcm\_controls,

.num\_controls = ARRAY\_SIZE(atm9009\_pcm\_controls),

};

#### struct snd\_pcm\_ops atm9009\_pcm\_ops

1. ops.open

当应用程序打开一个pcm设备时，该函数会被调用，通常，该函数会使用snd\_soc\_set\_runtime\_hwparams()设置substream中的snd\_pcm\_runtime结构里面的hw\_params相关字段，然后为snd\_pcm\_runtime的private\_data字段申请一个私有结构，用于保存该平台的dma参数。

1. ops.hw\_params

驱动的hw\_params阶段，该函数会被调用。通常，该函数会通过snd\_soc\_dai\_get\_dma\_data函数获得对应的dai的dma参数，获得的参数一般都会保存在snd\_pcm\_runtime结构的private\_data字段。然后通过snd\_pcm\_set\_runtime\_buffer函数设置snd\_pcm\_runtime结构中的dma buffer的地址和大小等参数。要注意的是，该回调可能会被多次调用，具体实现时要小心处理多次申请资源的问题。

1. ops.prepare

正式开始数据传送之前会调用该函数，该函数通常会完成dma操作的必要准备工作。

1. ops.trigger

数据传送的开始，暂停，恢复和停止时，该函数会被调用。

1. ops.pointer

该函数返回传送数据的当前位置。

static **struct snd\_pcm\_ops atm9009\_pcm\_ops** = {

.open = atm9009\_pcm\_open,

.close = atm9009\_pcm\_close,

.ioctl = snd\_pcm\_lib\_ioctl,

.hw\_params = atm9009\_pcm\_hw\_params,

.hw\_free = atm9009\_pcm\_hw\_free,

.prepare = atm9009\_pcm\_prepare,

.trigger = snd\_dmaengine\_pcm\_trigger,

.pointer = snd\_dmaengine\_pcm\_pointer\_no\_residue,

.mmap = atm9009\_pcm\_mmap,

};

#### snd\_soc\_register\_platform()

int **snd\_soc\_register\_platform**(struct device \*dev,

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

{

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

ret = snd\_soc\_add\_platform(dev, platform, platform\_drv);

}

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

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

{

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

platform->dev = dev;

platform->driver = platform\_drv;

platform->dapm.dev = dev;

platform->dapm.platform = platform;

platform->dapm.stream\_event = platform\_drv->stream\_event;

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

}

### DAI设备的创建

#### struct platform\_driver atm9009\_dai\_driver

#### DTS: owl-audio-i2s@e0166000

owl-audio-i2s@e0166000 {

compatible = "actions,owl-audio-i2s";

reg = <0 0xe0166000 0 0x34>;

clocks = <&clock CLK\_I2SRX>, <&clock CLK\_I2STX>, <&clock CLK\_HDMI\_AUDIO>, <&clock CLK\_SPDIF>;

clock-names = "i2srx", "i2stx", "hdmia", "spdif";

dmas = <&dma DMA\_DRQ\_I2S\_TX>, <&dma DMA\_DRQ\_I2S\_RX>, <&dma DMA\_DRQ\_HDMI\_AUDIO>, <&dma DMA\_DRQ\_SPDIF>, <&dma DMA\_DRQ\_I2STX\_SPDIF\_HDMI>;

dma-names = "tx","rx","hdmia","spdif","i2stx\_spdif\_hdmi";

pinctrl-names = "default";

pinctrl-0 = <&i2s0\_state\_default>;

};

#### atm9009\_dai\_probe()

static int **atm9009\_dai\_probe**(struct platform\_device \*pdev)

{

char \* playback\_dma\_chan\_name[MAX\_CODEC\_DAI\_ID\_NUMS] = {

"tx", "hdmia", "spdif", "i2stx\_spdif\_hdmi",

};

char \* capture\_dma\_chan\_name = {"rx"};

dn = of\_find\_compatible\_node(NULL, NULL, OWL\_CPU\_DAI\_COMPATIBLE);

for(i=0; i<1; i++)

{

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, i);

dai\_res.base[i] = devm\_ioremap(&pdev->dev, res->start, resource\_size(res));

}

for(i=0;i<MAX\_CODEC\_DAI\_ID\_NUMS;i++)

dai\_res.playback\_chan[i] = dma\_request\_slave\_channel(&pdev->dev, playback\_dma\_chan\_name[i]);

dai\_res.rx\_chan = dma\_request\_slave\_channel(&pdev->dev, capture\_dma\_chan\_name);

pdev->dev.init\_name = CPU\_DAI\_NAME;//"owl-audio-i2s"

snd\_soc\_register\_component(&pdev->dev, &atm9009\_component, &atm9009\_dai, 1);

}

### DMA内存管理

播放时，应用程序把音频数据源源不断地写入dma buffer中，然后相应platform的dma操作则不停地从该buffer中取出数据，经dai送往codec中。

录音时则正好相反，codec源源不断地把A/D转换好的音频数据经过dai送入dma buffer中，而应用程序则不断地从该buffer中读走音频数据。

#### atm9009\_pcm\_new() //分配DMA内存

static int **atm9009\_pcm\_new**(struct snd\_soc\_pcm\_runtime \*rtd)

{

struct snd\_card \*card = rtd->card->snd\_card;

struct snd\_pcm \*pcm = rtd->pcm;

if (pcm->streams[SNDRV\_PCM\_STREAM\_PLAYBACK].substream)

ret = atm9009\_pcm\_preallocate\_dma\_buffer(pcm, SNDRV\_PCM\_STREAM\_PLAYBACK);

if (pcm->streams[SNDRV\_PCM\_STREAM\_CAPTURE].substream)

ret = atm9009\_pcm\_preallocate\_dma\_buffer(pcm, SNDRV\_PCM\_STREAM\_CAPTURE);

}

#### atm9009\_pcm\_preallocate\_dma\_buffer()

该函数先是获得事先定义好的buffer大小，然后通过**dma\_alloc\_coherent**函数分配dma内存，然后完成substream->dma\_buffer的初始化赋值工作。

static int **atm9009\_pcm\_preallocate\_dma\_buffer**(struct snd\_pcm \*pcm, int stream)

{

struct snd\_pcm\_substream \*substream = pcm->streams[stream].substream;

struct snd\_dma\_buffer \*buf = &substream->dma\_buffer;

if (stream == SNDRV\_PCM\_STREAM\_PLAYBACK)

size = atm9009\_playback\_hw\_info.buffer\_bytes\_max;

else

size = atm9009\_capture\_hw\_info.buffer\_bytes\_max;

buf->dev.type = SNDRV\_DMA\_TYPE\_DEV;

buf->dev.dev = pcm->card->dev;

buf->private\_data = NULL;

**buf->area = dma\_alloc\_coherent**(pcm->card->dev, size, &buf->addr, GFP\_KERNEL);

memset(buf->area, 0, size);

buf->bytes = size;

}

#### atm9009\_pcm\_open() //设定DMA通道

static int **atm9009\_pcm\_open**(struct snd\_pcm\_substream \*substream)

{

struct dma\_chan \*chan;

if (substream->stream == SNDRV\_PCM\_STREAM\_PLAYBACK) {

snd\_soc\_set\_runtime\_hwparams(substream, &atm9009\_playback\_hw\_info);

} else {

snd\_soc\_set\_runtime\_hwparams(substream, &atm9009\_capture\_hw\_info);

}

ret = snd\_dmaengine\_pcm\_open\_request\_chan(substream, NULL, NULL);

}

#### snd\_dmaengine\_pcm\_open\_request\_chan()

int **snd\_dmaengine\_pcm\_open\_request\_chan**(struct snd\_pcm\_substream \*substream,

dma\_filter\_fn filter\_fn, void \*filter\_data)

{

if(substream->stream == SNDRV\_PCM\_STREAM\_PLAYBACK)

snd\_dmaengine\_pcm\_open(substream, dai\_dma\_slave\_playback\_chan(codec\_dai\_id));

else

snd\_dmaengine\_pcm\_open(substream, dai\_dma\_slave\_rx\_chan()); //参见DAI设备的创建

}

#### atm9009\_pcm\_hw\_params() //配置DMA的dts或src信息

static int **atm9009\_pcm\_hw\_params**(struct snd\_pcm\_substream \*substream,

struct snd\_pcm\_hw\_params \*params)

{

struct dma\_slave\_config slave\_config;

ret = snd\_hwparams\_to\_dma\_slave\_config(substream, params, &slave\_config);

if (substream->stream == SNDRV\_PCM\_STREAM\_PLAYBACK) {

slave\_config.dst\_addr = I2S\_SPDIF\_BASE+I2STX\_DAT;

slave\_config.dst\_addr\_width = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES;

} else {

slave\_config.src\_addr = I2S\_SPDIF\_BASE+I2SRX\_DAT;

slave\_config.src\_addr\_width = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES;

}

struct dma\_chan \*chan = snd\_dmaengine\_pcm\_get\_chan(substream);

ret = dmaengine\_slave\_config(chan, &slave\_config); //配置DMA通道

//将substream->dma\_buffer的数值拷贝到substream->runtime的相关字段中

//这样以后就可以通过substream->runtime获得这些地址和大小信息了

snd\_pcm\_set\_runtime\_buffer(substream, &substream->dma\_buffer);

}

#### snd\_hwparams\_to\_dma\_slave\_config()

int **snd\_hwparams\_to\_dma\_slave\_config**(const struct snd\_pcm\_substream \*substream,

const struct snd\_pcm\_hw\_params \*params, struct dma\_slave\_config \*slave\_config)

{

switch (params\_format(params)) {

case SNDRV\_PCM\_FORMAT\_S8: buswidth = DMA\_SLAVE\_BUSWIDTH\_1\_BYTE; break;

case SNDRV\_PCM\_FORMAT\_S16\_LE: buswidth = DMA\_SLAVE\_BUSWIDTH\_2\_BYTES; break;

case SNDRV\_PCM\_FORMAT\_S18\_3LE: buswidth = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES; break;

case SNDRV\_PCM\_FORMAT\_S20\_3LE: buswidth = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES; break;

case SNDRV\_PCM\_FORMAT\_S24\_LE: buswidth = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES; break;

case SNDRV\_PCM\_FORMAT\_S32\_LE: buswidth = DMA\_SLAVE\_BUSWIDTH\_4\_BYTES; break;

}

if (substream->stream == SNDRV\_PCM\_STREAM\_PLAYBACK) {

slave\_config->direction = DMA\_MEM\_TO\_DEV;

slave\_config->dst\_addr\_width = buswidth;

} else {

slave\_config->direction = DMA\_DEV\_TO\_MEM;

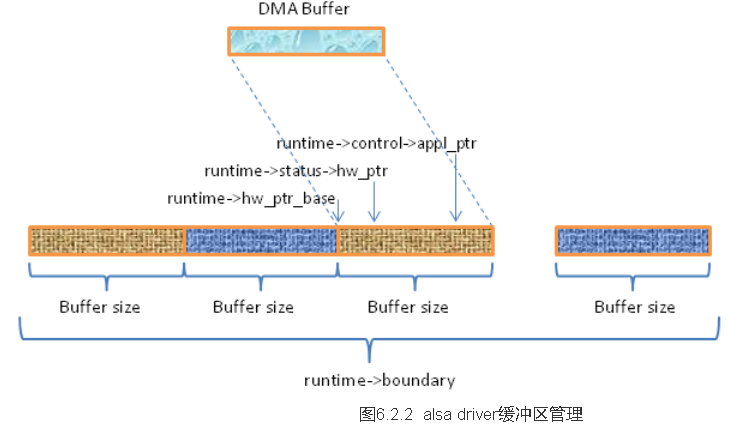
slave\_config->src\_addr\_width = buswidth;

}

slave\_config->device\_fc = false;

}

#### dma buffer管理



**snd\_pcm\_runtime结构中，使用了四个相关的字段来完成这个逻辑缓冲区的管理：**

1. snd\_pcm\_runtime.hw\_ptr\_base 环形缓冲区每一圈的基地址，当读写指针越过一圈后，它按buffer size进行移动；
2. snd\_pcm\_runtime.status->hw\_ptr 硬件逻辑位置，播放时相当于读指针，录音时相当于写指针；
3. snd\_pcm\_runtime.control->appl\_ptr 应用逻辑位置，播放时相当于写指针，录音时相当于读指针；
4. snd\_pcm\_runtime.boundary 扩展后的逻辑缓冲区大小，通常是(2^n)\*size；

**至少有3个途径可以完成对dma buffer的写入：**

1. 应用程序调用alsa-lib的snd\_pcm\_writei、snd\_pcm\_writen函数；
2. 应用程序使用ioctl：SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES或SNDRV\_PCM\_IOCTL\_WRITEN\_FRAMES；
3. 应用程序使用alsa-lib的snd\_pcm\_mmap\_begin/snd\_pcm\_mmap\_commit;

以上几种方式最终把数据写入dma buffer中，然后修改runtime->control->appl\_ptr的值。

**播放过程中，通常会配置成每一个period size生成一个dma中断，中断处理函数最重要的任务就是：**

1. 更新dma的硬件的当前位置，该数值通常保存在runtime->private\_data中；
2. 调用snd\_pcm\_period\_elapsed函数，该函数会进一步调用snd\_pcm\_update\_hw\_ptr0函数更新上述所说的4个缓冲区管理字段，然后唤醒相应的等待进程；

#### atm9009\_pcm\_mmap()

static int **atm9009\_pcm\_mmap**(struct snd\_pcm\_substream \*substream,

struct vm\_area\_struct \*vma)

{

struct **snd\_pcm\_runtime** \*runtime = substream->runtime;

dma\_mmap\_coherent(substream->pcm->card->dev, vma,

runtime->dma\_area, runtime->dma\_addr, runtime->dma\_bytes);

}

#### snd\_dmaengine\_pcm\_trigger()//设置DMA中断回调函数

int **snd\_dmaengine\_pcm\_trigger**(struct snd\_pcm\_substream \*substream, int cmd)

{

switch (cmd) {

case SNDRV\_PCM\_TRIGGER\_START:

**dmaengine\_pcm\_prepare\_and\_submit**(substream);

dma\_async\_issue\_pending(prtd->dma\_chan);

break;

case SNDRV\_PCM\_TRIGGER\_RESUME:

case SNDRV\_PCM\_TRIGGER\_PAUSE\_RELEASE:

dmaengine\_resume(prtd->dma\_chan);

break;

case SNDRV\_PCM\_TRIGGER\_SUSPEND:

case SNDRV\_PCM\_TRIGGER\_PAUSE\_PUSH:

dmaengine\_pause(prtd->dma\_chan);

break;

case SNDRV\_PCM\_TRIGGER\_STOP:

dmaengine\_terminate\_all(prtd->dma\_chan);

break;

}

}

#### dmaengine\_pcm\_prepare\_and\_submit()

static int **dmaengine\_pcm\_prepare\_and\_submit**(struct snd\_pcm\_substream \*substream)

{

struct dmaengine\_pcm\_runtime\_data \*prtd = substream\_to\_prtd(substream);

struct dma\_chan \*chan = prtd->dma\_chan;

struct dma\_async\_tx\_descriptor \*desc;

unsigned long flags = DMA\_CTRL\_ACK;

//DMA\_MEM\_TO\_DEV,DMA\_DEV\_TO\_MEM, DMA\_MEM\_TO\_MEM,

direction = snd\_pcm\_substream\_to\_dma\_direction(substream);

if (!substream->runtime->no\_period\_wakeup)

flags |= DMA\_PREP\_INTERRUPT;

prtd->pos = 0;

desc = dmaengine\_prep\_dma\_cyclic(chan, substream->runtime->dma\_addr,

snd\_pcm\_lib\_buffer\_bytes(substream),

snd\_pcm\_lib\_period\_bytes(substream), direction, flags);

desc->callback = dmaengine\_pcm\_dma\_complete; //设置DMA中断回调函数

desc->callback\_param = substream;

prtd->cookie = dmaengine\_submit(desc);

}

#### dmaengine\_pcm\_dma\_complete() //DMA中断处理函数

sound/soc/atc260x/dmaengine-pcm-owl.c

static void **dmaengine\_pcm\_dma\_complete**(void \*arg)

{

struct snd\_pcm\_substream \*substream = arg;

struct dmaengine\_pcm\_runtime\_data \*prtd = substream\_to\_prtd(substream);

spin\_lock\_irqsave(&lock, flags);

{

//修正prtd->pos的值

if (SNDRV\_PCM\_STREAM\_CAPTURE == substream->stream )

{ prtd->pos += snd\_pcm\_lib\_period\_bytes(substream);

if (prtd->pos >= snd\_pcm\_lib\_buffer\_bytes(substream)) prtd->pos = 0;

}

else if (SNDRV\_PCM\_STREAM\_PLAYBACK == substream->stream) {

int prev\_pos = prtd->pos;

struct dma\_tx\_state state;

dmaengine\_tx\_status(prtd->dma\_chan, prtd->cookie, &state);

int sum\_periods = (snd\_pcm\_lib\_buffer\_bytes(substream)/snd\_pcm\_lib\_period\_bytes(substream));

int remain\_frame\_cnt = state.residue/snd\_pcm\_lib\_period\_bytes(substream);

if (state.residue%snd\_pcm\_lib\_period\_bytes(substream))

remain\_frame\_cnt++;

remain\_frame\_cnt = remain\_frame\_cnt%sum\_periods;

prtd->pos = snd\_pcm\_lib\_buffer\_bytes(substream) -

(remain\_frame\_cnt\*snd\_pcm\_lib\_period\_bytes(substream));

if (prtd->pos >= snd\_pcm\_lib\_buffer\_bytes(substream)) prtd->pos = 0;

diff = (prtd->pos - prev\_pos);

if(diff>0) this\_interrupt\_num = diff/snd\_pcm\_lib\_period\_bytes(substream);

else

this\_interrupt\_num = (diff + snd\_pcm\_lib\_buffer\_bytes(substream)) /snd\_pcm\_lib\_period\_bytes(substream);

}

snd\_pcm\_period\_elapsed(substream);

}

spin\_unlock\_irqrestore(&lock, flags);

}

### Sample: snd-soc-dummy

sound/soc/soc-util.c

#### 注册device和driver: snd-soc-dummy

/sys/bus/platform/devices/snd-soc-dummy

/sys/bus/platform/drivers/snd-soc-dummy

/sys/bus/platform/drivers/snd-soc-dummy/snd-soc-dummy

/sys/devices/platform/snd-soc-dummy

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

static struct **platform\_device** \***soc\_dummy\_dev**;

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

static struct **platform\_driver soc\_dummy\_driver** = {

.driver = {

.name = "snd-soc-dummy",

.owner = THIS\_MODULE,

},

.probe = snd\_soc\_dummy\_probe,

.remove = snd\_soc\_dummy\_remove,

};

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

int \_\_init **snd\_soc\_util\_init**(void)

{

//注册platform\_device: "snd-soc-dummy"

soc\_dummy\_dev = platform\_device\_alloc("snd-soc-dummy", -1);

**platform\_device\_add**(soc\_dummy\_dev);

//注册platform\_driver: "snd-soc-dummy"

**platform\_driver\_register**(&soc\_dummy\_driver);

}

#### 注册codec, dai, platform设备

static int s**nd\_soc\_dummy\_probe**(struct platform\_device \*pdev)

{

//注册codec和dai设备

snd\_soc\_register\_codec(&pdev->dev, &dummy\_codec, &dummy\_dai, 1);

//注册platform设备

snd\_soc\_register\_platform(&pdev->dev, &dummy\_platform);

}

static struct **snd\_soc\_platform\_driver** dummy\_platform = {

.ops = &dummy\_dma\_ops,

};

static struct snd\_pcm\_ops dummy\_dma\_ops = {

.open = dummy\_dma\_open,

.ioctl = snd\_pcm\_lib\_ioctl,

};

static struct **snd\_soc\_codec\_driver** dummy\_codec;

static struct **snd\_soc\_dai\_driver** dummy\_dai = {

.name = "snd-soc-dummy-dai",

.playback = {

.stream\_name = "Playback",

.channels\_min = 1,

.channels\_max = 384,

.rates = SNDRV\_PCM\_RATE\_8000\_192000,

.formats = STUB\_FORMATS,

},

.capture = {

.stream\_name = "Capture",

.channels\_min = 1,

.channels\_max = 384,

.rates = SNDRV\_PCM\_RATE\_8000\_192000,

.formats = STUB\_FORMATS,

},

};

#define **STUB\_FORMATS** (SNDRV\_PCM\_FMTBIT\_S8 | SNDRV\_PCM\_FMTBIT\_U8 | \

SNDRV\_PCM\_FMTBIT\_S16\_LE | SNDRV\_PCM\_FMTBIT\_U16\_LE | \

SNDRV\_PCM\_FMTBIT\_S24\_LE | SNDRV\_PCM\_FMTBIT\_U24\_LE | \

SNDRV\_PCM\_FMTBIT\_S32\_LE | SNDRV\_PCM\_FMTBIT\_U32\_LE | \

SNDRV\_PCM\_FMTBIT\_IEC958\_SUBFRAME\_LE)

#### 分配DMA

static int **dummy\_dma\_open**(struct snd\_pcm\_substream \*substream)

{

snd\_soc\_set\_runtime\_hwparams(substream, &dummy\_dma\_hardware);

}

static const struct **snd\_pcm\_hardware** dummy\_dma\_hardware = {

.formats = 0xffffffff,

.channels\_min = 1,

.channels\_max = UINT\_MAX,

.info = SNDRV\_PCM\_INFO\_INTERLEAVED | SNDRV\_PCM\_INFO\_BLOCK\_TRANSFER,

.buffer\_bytes\_max = 128\*1024,

.period\_bytes\_min = PAGE\_SIZE,

.period\_bytes\_max = PAGE\_SIZE\*2,

.periods\_min = 2,

.periods\_max = 128,

};

# ALSA设备文件结构

$ cd /dev/snd

$ ls -l

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

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

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

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

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

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

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

我们可以看到以下设备文件:

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

midiC0D0 --> 用于播放midi音频

pcmC0D0c --〉 用于录音的pcm设备

pcmC0D0p --〉 用于播放的pcm设备

seq --〉 音序器

timer --〉 定时器

C0D0代表的是声卡0中的设备0，

pcmC0D0c最后一个c代表capture，

pcmC0D0p最后一个p代表playback，这些都是alsa-driver中的命名规则。

# 寄存器配置

I2S\_SPDIF: 0xe0166000

## 录音

### I2S\_SPDIF

root@gs700c:/ # busybox devmem 0xe0166000 32

0x00000C03

I2S TX: Enable, output word length=32bit, BCLK rising edge sample, TX mode=2.0-Channel mode,

I2S RX: Enable, RX mode=2.0-Channel mode, RX clock=frome I2S\_CLK1

I2S: pin mode=4-wire,

root@gs700c:/ # busybox devmem 0xe0166004 32

0x00053603

root@gs700c:/ # busybox devmem 0xe0166008 32

0x00000000

root@gs700c:/ # busybox devmem 0xe016600c 32

0xFFF75000

root@gs700c:/ # busybox devmem 0xe0166010 32

0x00000000

root@gs700c:/ # busybox devmem 0xe0166014 32

0x00000000

root@gs700c:/ # busybox devmem 0xe0166018 32

0x10550100

root@gs700c:/ # busybox devmem 0xe016601c 32

0x000000F2

root@gs700c:/ # busybox devmem 0xe0166020 32

0x00000000

root@gs700c:/ # busybox devmem 0xe0166024 32

0x00000000

root@gs700c:/ # busybox devmem 0xe0166028 32

0x00000000

root@gs700c:/ # busybox devmem 0xe016602c 32

0x00000001

root@gs700c:/ # busybox devmem 0xe0166030 32

0x00000000

# ALC:

## alc5632\_snd\_controls:

Auxout Playback Volume / Switch

Voice DAC Playback Volume / Switch

Phone Playback Volume

LineIn Playback Volume

Master Playback Volume / Switch

Mic1 Playback Volume

Mic2 Playback Volume

Rec Capture Volume

Mic 1 Boost Volume

Mic 2 Boost Volume

DMIC Boost Capture Volume

DMIC En Capture Switch

DMIC PreFilter Capture Switch

## alc5632\_dapm\_widgets