linux音频alsa-uda134x驱动分析之一
http://www.usr.cc/thread-1740-1-1.html
前言
目前，linux系统常用的音频驱动有两种形式:alsa oss
alsa:现在是linux下音频驱动的主要形式，与简单的oss兼容。
oss：过去的形式
而我们板子上的uda1341用的就是alsa驱动。
alsa概述：
因为我们用的是板上系统,用的也是alsa 的一个soc子系统。所以我们直接讲解alsa soc子系统。

ALSA SoC Layer
ALSA板上系统层
==========

The overall project goal of the ALSA System on Chip (ASoC) layer is to provide better ALSA support for embedded system-on-chip processors (e.g. pxa2xx, au1x00, iMX, etc) and portable audio codecs. Prior to the ASoC subsystem there was some support in the kernel for SoC audio, however it had some limitations:-
ALSA板上系统(ASoC)层的总体项目目标，是为对SOC嵌入式处理器和便携音频解码器提供更好的ALSA支持。在ASoC子系统之前，已有对内核的SoC音频支持，但是那些支持存在一些局限：
Codec drivers were often tightly coupled to the underlying SoC
    CPU. This is not ideal and leads to code duplication - for example,
    Linux had different wm8731 drivers for 4 different SoC platforms.
解码器常常与底层嵌入式处理器一对一紧密结合。这是非理想化的，因为这将导致代码的重复－例如，对四个不同的嵌入式平台，Linux要有不同的wm8731驱动。(理想的状态是我们可以只有一个wm8731的驱动代码，就可以对应于四个不同的处理器，但由上面说的，解码器－这里的wm8731与底层嵌入式处理器结合过于紧密，无法实现wm8731驱动代码的复用)

  * There was no standard method to signal user initiated audio events (e.g.
    Headphone/Mic insertion, Headphone/Mic detection after an insertion
    event). These are quite common events on portable devices and often require
    machine specific code to re-route audio, enable amps, etc., after such an
    event.
没有一个标准的方法可以产生用户初始化音频事件的信号（即，耳机／麦克插入，响应插入事件的耳机／麦克探测)。这些在便携设备上都是十分常见的事件并且在这些事件之后经常需要机器相关的代码来对音频重设路径，开启放大器等。

  * Drivers tended to power up the entire codec when playing (or
    recording) audio. This is fine for a PC, but tends to waste a lot of
    power on portable devices. There was also no support for saving

power via changing codec oversampling rates, bias currents, etc.
放音（录音）时，驱动常常会打开整个解码器。对个人电脑来说这没什么问题，但是在便携设备上往往会导致电能的浪费。另外，也没有通过改变解码器采样率、偏置电流等方式来省电的支持。

ASoC Design
ASoC 设计
==========

The ASoC layer is designed to address these issues and provide the following features :-
ASoC层被设计用来解决这些问题并提供如下特性：

* Codec independence. Allows reuse of codec drivers on other platforms and machines.
解码器独立。允许在其它平台或机器上重用解码器驱动。

* Easy I2S/PCM audio interface setup between codec and SoC. Each SoC interface and codec registers it's audio interface capabilities with the core and are subsequently matched and configured when the application hardware parameters are known.
解码器与SoC的I2S/PCM音频接口设置很容易。每个SoC接口与解码器都向ALSA核心注册它的音频接口能力，而且应用硬件参数已知时顺序匹配并配置。

* Dynamic Audio Power Management (DAPM). DAPM automatically sets the codec to
its minimum power state at all times. This includes powering up/down internal power blocks depending on the internal codec audio routing and any active streams.
动态音频电源管理（DAPM）。DAPM自动无论何时，总是把解码器自动设置为它的最小电源状态。这包括依据内部解码音频线路和活跃的流来开启和关闭内部电源模块

* Pop and click reduction. Pops and clicks can be reduced by powering the codec up/down in the correct sequence (including using digital mute). ASoC signals the codec when to change power states.
咔嗒声减少。咔嗒声可以通过使用正确的解码器电源开启和关闭顺序而减少（包括使用数字消音）。ASoC在改变电源状态时向解码器发出信号。

* Machine specific controls: Allow machines to add controls to the sound card (e.g. volume control for speaker amplifier).
机器相关的控制：允许机器增加对声卡的控制。（如扬声器放大器的音量控制）。

To achieve all this, ASoC basically splits an embedded audio system into 3

components :-
要实现这些，ASoC基本上将嵌入式音频系统分为3个部分：

* Codec driver: The codec driver is platform independent and contains audio controls, audio interface capabilities, codec DAPM definition and codec IO functions.
解码器驱动：解码器驱动是平台无关的，包含音频控制、音频接口能力、解码器动态音频电源管理和解码器IO函数。

* Platform driver: The platform driver contains the audio DMA engine and audio interface drivers (e.g. I2S, AC97, PCM) for that platform.
平台驱动：平台驱动包含相应平台的音频DAM引擎和音频接口驱动(如I2S,AC97,PCM)

* Machine driver: The machine driver handles any machine specific controls and
audio events (e.g. turning on an amp at start of playback).
机器驱动：机器驱动处理所有机器相关的控制和音频事件（如回放开始时打开放大器）。

Documentation
文档
==========

The documentation is spilt into the following sections:-
本文档分成如下部分：

overview.txt: This file.
overview.txt:概述，本文件。
codec.txt: Codec driver internals.
codec.txt:解码器驱动内部实现

DAI.txt: Description of Digital Audio Interface standards and how to configure a DAI within your codec and CPU DAI drivers.
DAI.txt:对数字音频接口（DAI）标准和如何配置你的解码器和CPU的数字音频接品驱动中的数字音频接口的描述。

dapm.txt: Dynamic Audio Power Management
dapm.txt:动态音频电源管理

platform.txt: Platform audio DMA and DAI.
platform.txt:平台音频DMA和DAI。

machine.txt: Machine driver internals.
machine.txt:机器驱动内容介绍。

pop_clicks.txt: How to minimise audio artifacts.
pop_clicks.txt:如何最小化音步噪声。

clocking.txt: ASoC clocking for best power performance.
clocking.txt:最佳电源表现下的ASoC时钟

linux音频alsa-uda134x驱动分析之二（时钟）
http://www.usr.cc/thread-1741-1-1.html
Audio Clocking
音频时钟
==========

This text describes the audio clocking terms in ASoC and digital audio in general. Note: Audio clocking can be complex!
本文本总体描述ASoC和数字音频中的音频时钟条款。

Master Clock
主时钟

Every audio subsystem is driven by a master clock (sometimes referred to as MCLK or SYSCLK). This audio master clock can be derived from a number of sources (e.g. crystal, PLL, CPU clock) and is responsible for producing the correct audio playback and capture sample rates.
每个数字音频子系统都是由主时钟来驱动的（有时称为MCLK或SYSCLK）。音频主时钟可以派生于多种源（如晶振，锁相环，处理器时钟）。负责产生正确的音频播放和捕获采样率。

Some master clocks (e.g. PLLs and CPU based clocks) are configurable in that their speed can be altered by software (depending on the system use and to save power). Other master clocks are fixed at a set frequency (i.e. crystals).
有些主时钟是可配置的（如基于锁相环或处理器的时钟），它们可以通过软件改变速度（依赖于系统应用和省电的考虑）。另一些主时钟则是固定于一个特定的频率值（如晶振）。

DAI Clocks
数字音频时钟

The Digital Audio Interface is usually driven by a Bit Clock (often referred

to
as BCLK). This clock is used to drive the digital audio data across the link between the codec and CPU.
数字音频接口往往是由一个位时钟来驱动的（通常记为BCLK）。这个时钟用于驱动数字音频数据在解码器与处理器间的传输。

The DAI also has a frame clock to signal the start of each audio frame. This clock is sometimes referred to as LRC (left right clock) or FRAME. This clock runs at exactly the sample rate (LRC = Rate).
数字音频接口还有一个帧时钟，用来指示一帧音频的开始。该时钟有时记为LRC（left right clock）或FRAME。该时钟严格工作于采样率上。

Bit Clock can be generated as follows:-
位时钟可以有如下产生方式：

BCLK = MCLK / x

or

BCLK = LRC * x

or

BCLK = LRC * Channels * Word Size

This relationship depends on the codec or SoC CPU in particular. In general it is best to configure BCLK to the lowest possible speed (depending on your rate, number of channels and word size) to save on power.
这个关系依赖于解码器，特别是板上处理器。大体上讲，最好将位时钟尽可能低速（取决于你的采样率，通道数和字长）以省电。

It is also desirable to use the codec (if possible) to drive (or master) the audio clocks as it usually gives more accurate sample rates than the CPU.
可能的话，最好使用解码器来驱动（或控制）音频时钟，因为通常它给出的采样率比处理器更精确。

linux音频alsa-uda134x驱动分析之三（解码器）
http://www.usr.cc/thread-1742-1-1.html
ASoC Codec Driver
ASoC解码器驱动
==========

The codec driver is generic and hardware independent code that configures the codec to provide audio capture and playback. It should contain no code that is

specific to the target platform or machine. All platform and machine specific code should be added to the platform and machine drivers respectively.
解码器驱动是通用、硬件无关的代码，它配置解码器以支持音频捕获与回放。它不应包含任何与目标平台或机器相关的代码。平台或机器相关代码应该分别加 入到平台和机器驱动中去。

Each codec driver *must* provide the following features:-
各解码器驱动必须提供如下特性：

1) Codec DAI and PCM configuration
2) Codec control IO - using I2C, 3 Wire(SPI) or both APIs
3) Mixers and audio controls
4) Codec audio operations
1）解码器数字音频接口和PCM配置。
2）解码器控制IO－使用I2C，3总线（SPI）或两个都有。
3）混音器和音频控制。
4）解码器音频操作。

Optionally, codec drivers can also provide:-
解码器驱动可以选择性提供：

5) DAPM description.
6) DAPM event handler.
7) DAC Digital mute control.
5）动态音频电源管理描述。
6）动态音频电源管理事件控制。
7）数模转换数字消音控制。

Its probably best to use this guide in conjunction with the existing codec driver code in sound/soc/codecs/
大家也许最好联同已存在于sound/soc/codecs/中的驱动代码一起来使用这个指导书。

ASoC Codec driver breakdown
ASoC 解码器崩溃
==========================

1 - Codec DAI and PCM configuration
1－解码器数字音频接口和PCM配置

Each codec driver must have a struct snd_soc_codec_dai to define its DAI and

PCM capabilities and operations. This struct is exported so that it can be registered with the core by your machine driver.
各解码器驱动必须有一个snd_soc_codec_dai数据结构，它用来定义DAI和PCM提供的功能和操作。这个数据结构要导出，好让你的机器驱动程序将它注册到ALSA核心中去。

e.g.
例如：

```
struct snd_soc_codec_dai wm8731_dai = {
    .name = "WM8731",
    /* playback capabilities */
    .playback = {
        .stream_name = "Playback",
        .channels_min = 1,
        .channels_max = 2,
        .rates = WM8731_RATES,
        .formats = WM8731_FORMATS,},
    /* capture capabilities */
    .capture = {
        .stream_name = "Capture",
        .channels_min = 1,
        .channels_max = 2,
        .rates = WM8731_RATES,
        .formats = WM8731_FORMATS,},
    /* pcm operations - see section 4 below */
    .ops = {
        .prepare = wm8731_pcm_prepare,
        .hw_params = wm8731_hw_params,
        .shutdown = wm8731_shutdown,
    },
    /* DAI operations - see DAI.txt */
    .dai_ops = {
        .digital_mute = wm8731_mute,
        .set_sysclk = wm8731_set_dai_sysclk,
        .set_fmt = wm8731_set_dai_fmt,
    }
};
EXPORT_SYMBOL_GPL(wm8731_dai);
```

2 - Codec control IO
2—解码器控制IO

The codec can usually be controlled via an I2C or SPI style interface (AC97 combines control with data in the DAI). The codec drivers provide

functions to read and write the codec registers along with supplying a register cache:-
解码器通常可以通过I2C或SPI类型的接器进行控制（AC97 的数字音频接口中把数据和控制结合在了一起）。解码器驱动提供读写解码器寄存器和供应寄存器缓存的功能。

```
/* IO control data and register cache */
void *control_data; /* codec control (i2c/3wire) data */
void *reg_cache;
```

Codec read/write should do any data formatting and call the hardware read write below to perform the IO. These functions are called by the core and ALSA when performing DAPM or changing the mixer:-
解码器读写要可以作用于任何格式，调用底层硬件的读写功能操作IO。ALSA核或ALSA在动态音频电源管理或改变混音器时会调用这些函数。

```
unsigned int (*read)(struct snd_soc_codec *, unsigned int);
int (*write)(struct snd_soc_codec *, unsigned int, unsigned int);
```

Codec hardware IO functions - usually points to either the I2C, SPI or AC97 read/write:-
解码器硬件IO函数－通常指向I2C，SPI或AC97的读写：

```
hw_write_t hw_write;
hw_read_t hw_read;
```

3 - Mixers and audio controls
3－混音器和音频控制

All the codec mixers and audio controls can be defined using the convenience macros defined in soc.h.
所有解码器混音器和音频控制都可以通过使用soc.h中定义的宏而带来方便。

```
#define SOC_SINGLE(xname, reg, shift, mask, invert)
```

Defines a single control as follows:-
这个宏可以定义一个单次操作：

```
xname = Control name e.g. "Playback Volume"
reg = codec register
shift = control bit(s) offset in register
mask = control bit size(s) e.g. mask of 7 = 3 bits
invert = the control is inverted
```

Other macros include:-
其它的宏还有：

```
    #define SOC_DOUBLE(xname, reg, shift_left, shift_right, mask, invert)
```

A stereo control
下面是一个立体声控制：

```
    #define SOC_DOUBLE_R(xname, reg_left, reg_right, shift, mask, invert)
```

A stereo control spanning 2 registers
扩用两个寄存器的立体声控制如下：

```
    #define SOC_ENUM_SINGLE(xreg, xshift, xmask, xtexts)
```

Defines an single enumerated control as follows:-
定义一个枚举控制如下：

```
   xreg = register
   xshift = control bit(s) offset in register
   xmask = control bit(s) size
   xtexts = pointer to array of strings that describe each setting

   #define SOC_ENUM_DOUBLE(xreg, xshift_l, xshift_r, xmask, xtexts)
```

Defines a stereo enumerated control
上面的宏定义一个立体声枚举控制。

4 - Codec Audio Operations
4－解码器音频操作

The codec driver also supports the following ALSA operations:-
解码器驱动还支持下面的ALSA操作：

```
/* SoC audio ops */
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 *);
};
```

Please refer to the ALSA driver PCM documentation for details.
详情请参考ALSA驱动PCM文档：
http://www.alsa-project.org/~iwai/writing-an-alsa-driver/c436.htm

5 - DAPM description.
5－动态音频电源管理

The Dynamic Audio Power Management description describes the codec power components and their relationships and registers to the ASoC core.
Please read dapm.txt for details of building the description.
动态音频电源管理描述的是解码器电源组件和它们的关系，和向ASoC核注册的方法。详情请阅dapm.txt。

Please also see the examples in other codec drivers.
也请参照其它解码器的例子。

6 - DAPM event handler
6－动态音频电源管理事件句柄

This function is a callback that handles codec domain PM calls and system domain PM calls (e.g. suspend and resume). It is used to put the codec to sleep when not in use.
该功能是一个回调函数，用来处理解码器域的电源管理调用和系统域的电源管理调用（如挂起和恢复）。它用来在不用时使解码器进入休眠状态。

Power states:-
电源状态：

```
    SNDRV_CTL_POWER_D0: /* full On */
    /* vref/mid, clk and osc on, active */

    SNDRV_CTL_POWER_D1: /* partial On */
    SNDRV_CTL_POWER_D2: /* partial On */

    SNDRV_CTL_POWER_D3hot: /* Off, with power */
    /* everything off except vref/vmid, inactive */

    SNDRV_CTL_POWER_D3cold: /* Everything Off, without power */
```

7 - Codec DAC digital mute control

7－解码器数模转换消音控制

Most codecs have a digital mute before the DACs that can be used to minimise any system noise. The mute stops any digital data from entering the DAC.
多数解码器都有一个数字消音装置放在数模转换器前面。它可以用来最小化系统躁声。消音器不让任何数字数据进入DAC。

A callback can be created that is called by the core for each codec DAI when the mute is applied or freed.
消音器使用或释放时会创造一个回调。ASoC核会为每个解码器数字音频接口调用这个回调函数。

i.e.

```
static int wm8974_mute(struct snd_soc_codec *codec,
    struct snd_soc_codec_dai *dai, int mute)
{
    u16 mute_reg = wm8974_read_reg_cache(codec, WM8974_DAC) & 0xffbf;
    if(mute)
        wm8974_write(codec, WM8974_DAC, mute_reg | 0x40);
    else
        wm8974_write(codec, WM8974_DAC, mute_reg);
    return 0;
}
```

linux音频alsa-uda134x驱动分析之四（数字音频接口）
http://www.usr.cc/thread-1743-1-1.html

ASoC currently supports the three main Digital Audio Interfaces (DAI) found on SoC controllers and portable audio CODECs today, namely AC97, I2S and PCM.
ASoC现在支持如今的SoC控制器和便携音频解码器上的三个主要数字音频接口，即AC97, I2S，PCM。

AC97
AC97
====

AC97 is a five wire interface commonly found on many PC sound cards. It is now also popular in many portable devices. This DAI has a reset line and time multiplexes its data on its SDATA_OUT (playback) and SDATA_IN (capture) lines. The bit clock (BCLK) is always driven by the CODEC (usually 12.288MHz) and the

frame (FRAME) (usually 48kHz) is always driven by the controller. Each AC97 frame is 21uS long and is divided into 13 time slots.
AC97是一种个人电脑声卡上常见的五线接口。现在在很多便携设备中也很流行。这个数字音频接口有一个复位线，分时在SDATA__OUT（回放）和SDATA__IN（捕获）线上传送数据。位时钟常由解码器驱动（通常是12.288MHz).帧时钟（通常48kHz)总是由控制器驱动。每个AC97帧21uS,并分为13个时间槽。

The AC97 specification can be found at :-
AC97说明书可以在下面的网址找到：
http://www.intel.com/design/chipsets/audio/ac97_r23.pdf

I2S
I2S
===

I2S is a common 4 wire DAI used in HiFi, STB and portable devices. The Tx and Rx lines are used for audio transmission, whilst the bit clock (BCLK) and left/right clock (LRC) synchronise the link. I2S is flexible in that either the controller or CODEC can drive (master) the BCLK and LRC clock lines. Bit clock usually varies depending on the sample rate and the master system clock (SYSCLK). LRCLK is the same as the sample rate. A few devices support separate ADC and DAC LRCLKs, this allows for simultaneous capture and playback at different sample rates.
I2S是一个4线数字音频接口，常用于HiFi，STB便携设备。Tx 和Rx信号线用于音频传输。而位时钟和左右时钟（LRC）用于同步链接。I2S具有灵活性，因为控制器和解码器都可以控制位时钟和左右时钟。位时钟因采样率和主系统时钟而有不同。LRCLK与采样率相同。少数设备支持独立的ADC和DAC的LRCLK。这使在不同采样率情况下同步捕获和回放成为可能。

I2S has several different operating modes:-
I2S有几个不同的操作模式：

o I2S - MSB is transmitted on the falling edge of the first BCLK after LRC transition.
I2S模式－MSB在LRC后的第一个位时钟的下降沿传送。

o Left Justified - MSB is transmitted on transition of LRC.
左对齐模式：MSB在LRC传送时传送。

o Right Justified - MSB is transmitted sample size BCLKs before LRC transition.

右对齐模式：MSB在（此句不懂）

PCM
PCM
===

PCM is another 4 wire interface, very similar to I2S, which can support a more flexible protocol. It has bit clock (BCLK) and sync (SYNC) lines that are used to synchronise the link whilst the Tx and Rx lines are used to transmit and receive the audio data. Bit clock usually varies depending on sample rate whilst sync runs at the sample rate. PCM also supports Time Division Multiplexing (TDM) in that several devices can use the bus simultaneously (this is sometimes referred to as network mode).
PCM也是一种4线制接口。与I2S非常相像，但支持一个更灵活的协议。它有位时钟（BCLK）和同步时钟（SYNC）用来在Tx和Rx在传送和接收音频数据是同步连接。位时钟通常因采样率的不同而不同，然而同步时钟（SYNC）与采样频率相同。PCM同样支持时分复用，可以几个设备同时使用总线（这有时被称为net work 模式）。

Common PCM operating modes:-
常用的PCM操作模式：

o Mode A - MSB is transmitted on falling edge of first BCLK after FRAME/SYNC.
模式A－MSB在FRAME/SYNC后第一个BCLK的下降沿传送。

o Mode B - MSB is transmitted on rising edge of FRAME/SYNC.
模式B－MSB在FRAME/SYNC的上升沿传送。

linux音频alsa-uda134x驱动分析之五（动态音频电源管理）
http://www.usr.cc/thread-1744-1-1.html

Dynamic Audio Power Management for Portable Devices
便携设备的动态音频电源管理
==========

1. Description
1、概述
==========

Dynamic Audio Power Management (DAPM) is designed to allow portable Linux devices to use the minimum amount of power within the audio subsystem at all times. It is independent of other kernel PM and as such, can easily co-exist with the other PM systems.
动态音频电源管理(DAPM)设计用来使便携Linux设备可以在任何时最小化音频子系统的电能。它独立于其它内核电源管理，因此容易与其它电源管理系统共存。

DAPM is also completely transparent to all user space applications as all power switching is done within the ASoC core. No code changes or recompiling are required for user space applications. DAPM makes power switching decisions based upon any audio stream (capture/playback) activity and audio mixer settings within the device.
DAPM对用户程序也是完全透明的，因为所有的电源切换都是在ASoC核内部进行的。用户程序无须修改代码或重新编译。DAPM的切换是依据设备内部的各音频流（捕获／回放）活动和音频混音机设置决定的。

DAPM spans the whole machine. It covers power control within the entire audio subsystem, this includes internal codec power blocks and machine level power systems.
DAPM扩展到整个机器。它包括整个音频子系统的电源控制。这包括内部解码器电源模块和机器级电源系统。

There are 4 power domains within DAPM
DAPM有4个电源域：

1. Codec domain - VREF, VMID (core codec and audio power)
   Usually controlled at codec probe/remove and suspend/resume, although can be set at stream time if power is not needed for sidetone, etc.

1、解码器域－VREF，VMID（核解码器和音频电源）
通常在解码器检测／移除和挂起／恢复时控制，虽然在不需电源或侧音时也可以在流时间设置。

2. Platform/Machine domain - physically connected inputs and outputs
   Is platform/machine and user action specific, is configured by the machine driver and responds to asynchronous events e.g when HP are inserted

2、平台／机器域－有物理连接的输入输出
是平台／机器相关且用户动作相关的。由机器驱动配置HP插入时响应同步事件。

3. Path domain - audio susbsystem signal paths
   Automatically set when mixer and mux settings are changed by the user. e.g. alsamixer, amixer.

3、路径域－音频子系统信号路径
混音机和复用器设置被用户改变时自动设置。如alsomixer,amixer.

4. Stream domain - DACs and ADCs.
   Enabled and disabled when stream playback/capture is started and stopped respectively. e.g. aplay, arecord.

4、数据流域－DAC和ADC
流插放／捕获开始和停止时分别使能和禁用，如aplay,arecord.

All DAPM power switching decisions are made automatically by consulting an audio routing map of the whole machine. This map is specific to each machine and consists of the interconnections between every audio component (including internal codec components). All audio components that effect power are called widgets hereafter.
所有动态音频电源管理(DAPM)电源开关自动取决于整个音频芯片的通路表。
这张通路表是具体到每台机器和每个音频之间的相互联系组成部分（包括内部解码器组件）。

2. DAPM Widgets
==========

Audio DAPM widgets fall into a number of types:-

o Mixer - Mixes several analog signals into a single analog signal.
o Mux - An analog switch that outputs only one of many inputs.
o PGA - A programmable gain amplifier or attenuation widget.
o ADC - Analog to Digital Converter
o DAC - Digital to Analog Converter
o Switch - An analog switch
o Input - A codec input pin
o Output - A codec output pin
o Headphone - Headphone (and optional Jack)
o Mic - Mic (and optional Jack)
o Line - Line Input/Output (and optional Jack)
o Speaker - Speaker
o Pre - Special PRE widget (exec before all others)
o Post - Special POST widget (exec after all others)

(Widgets are defined in include/sound/soc-dapm.h)

Widgets are usually added in the codec driver and the machine driver. There are convenience macros defined in soc-dapm.h that can be used to quickly build a list of widgets of the codecs and machines DAPM widgets.
Widget是通常添加在解码器驱动和机器的驱动程序。
有在soc-dapm.h方便的宏定义，可用于快速建立一个编解码器的动态音频电源管理部件和机器部件清单。

Most widgets have a name, register, shift and invert. Some widgets have extra parameters for stream name and kcontrols.
大部分部件有一个名称，寄存器，偏移量和取反值。有些部件有extra parameters for

stream name 和kcontrols额外的参数.

2.1 Stream Domain Widgets
2.1 数据流域工具

Stream Widgets relate to the stream power domain and only consist of ADCs (analog to digital converters) and DACs (digital to analog converters).
数据流域工具涉及到流电源域，只有确定ADC（模拟到数字转换器）和DAC（数字到模拟转换器）组成。

Stream widgets have the following format:-
流部件的格式如下： -

SND_SOC_DAPM_DAC(name, stream name, reg, shift, invert),

NOTE: the stream name must match the corresponding stream name in your codec snd_soc_codec_dai.
the stream name必须符合在你的解码器snd_soc_codec_dai相应流的名称。

e.g. stream widgets for HiFi playback and capture

```
SND_SOC_DAPM_DAC("HiFi DAC", "HiFi Playback", REG, 3, 1),
SND_SOC_DAPM_ADC("HiFi ADC", "HiFi Capture", REG, 2, 1),
```

2.2 Path Domain Widgets
2.2 路径域小工具

Path domain widgets have a ability to control or affect the audio signal or audio paths within the audio subsystem. They have the following form:-
路径域部件有能力控制或影响的音频信号或音频子系统内的音频通道。他们有以下形式： -

SND_SOC_DAPM_PGA(name, reg, shift, invert, controls, num_controls)

Any widget kcontrols can be set using the controls and num_controls members.
任何部件kcontrols可以设置使用的controls和num_controls成员。

e.g. Mixer widget (the kcontrols are declared first)

```
/* Output Mixer */
static const snd_kcontrol_new_t wm8731_output_mixer_controls[] = {
```

```
SOC_DAPM_SINGLE("Line Bypass Switch", WM8731_APANA, 3, 1, 0),
SOC_DAPM_SINGLE("Mic Sidetone Switch", WM8731_APANA, 5, 1, 0),
SOC_DAPM_SINGLE("HiFi Playback Switch", WM8731_APANA, 4, 1, 0),
};

SND_SOC_DAPM_MIXER("Output Mixer", WM8731_PWR, 4, 1,
wm8731_output_mixer_controls,
    ARRAY_SIZE(wm8731_output_mixer_controls)),
```

If you dont want the mixer elements prefixed with the name of the mixer widget, you can use SND_SOC_DAPM_MIXER_NAMED_CTL instead. the parameters are the same as for SND_SOC_DAPM_MIXER.
如果你不想要的混频器，混频器的部件名称为前缀的元素，可以使用SND_SOC_DAPM_MIXER_NAMED_CTL代替。
该参数是作为SND_SOC_DAPM_MIXER相同。

2.3 Platform/Machine domain Widgets
2.3 平台域工具

Machine widgets are different from codec widgets in that they don't have a codec register bit associated with them. A machine widget is assigned to each machine audio component (non codec) that can be independently powered. e.g.
平台域工具不同于其他codec域，它们没有一个编解码器寄存器位与他们有联系。
一台机器部件被分配到每台机器音频组件（非编解码器），可以独立供电。例如：

- o Speaker Amp
- o Microphone Bias
- o Jack connectors

A machine widget can have an optional call back.
一台机器部件可以有一个可选的回调。

e.g. Jack connector widget for an external Mic that enables Mic Bias when the Mic is inserted:-
例如：对一个外部麦克风，使麦克风偏置当麦克风插入插孔部件： -

```
static int spitz_mic_bias(struct snd_soc_dapm_widget* w, int event)
{
    gpio_set_value(SPITZ_GPIO_MIC_BIAS, SND_SOC_DAPM_EVENT_ON(event));
    return 0;
}

SND_SOC_DAPM_MIC("Mic Jack", spitz_mic_bias),
```

2.4 Codec Domain
2.4 Codec域

The codec power domain has no widgets and is handled by the codecs DAPM event handler. This handler is called when the codec powerstate is changed wrt to any stream event or by kernel PM events.
编解码器的电源域没有部件，是由编解码器动态音频电源管理事件处理程序处理。
此处理程序被调用时，编解码器电源状态改变流相对于任何事件或内核PM事件。

2.5 Virtual Widgets
2.5 虚拟工具

Sometimes widgets exist in the codec or machine audio map that don't have any corresponding soft power control. In this case it is necessary to create a virtual widget - a widget with no control bits e.g.
有些工具在codec或平台音频表中无法被软件控制。
在这种情况下，有必要建立一个虚拟的小工具 - 例如，没有一个部件控制位。

SND_SOC_DAPM_MIXER("AC97 Mixer", SND_SOC_DAPM_NOPM, 0, 0, NULL, 0),

This can be used to merge to signal paths together in software.
这可以用于信号的软件合并到一起的路径。

After all the widgets have been defined, they can then be added to the DAPM subsystem individually with a call to snd_soc_dapm_new_control().
当所有的部件已经确定，他们就可以被添加到动态音频电源管理子系统单独以snd_soc_dapm_new_control（）调用。

3. Codec Widget Interconnections
3. Codec工具互联
==========

Widgets are connected to each other within the codec and machine by audio paths (called interconnections). Each interconnection must be defined in order to create a map of all audio paths between widgets.
Widget是相互连接的编解码器和内（称为互连）音频通道机。
每个互连必须界定，以便建立一个部件之间的所有音频路径图。

This is easiest with a diagram of the codec (and schematic of the machine audio system), as it requires joining widgets together via their audio signal paths.

这是一个编解码器（和机器音响系统示意图）图最简单的，因为它需要通过加入他们的音频信号通道部件在一起。

e.g., from the WM8731 output mixer (wm8731.c)

The WM8731 output mixer has 3 inputs (sources)

1. Line Bypass Input
2. DAC (HiFi playback)
3. Mic Sidetone Input

Each input in this example has a kcontrol associated with it (defined in example above) and is connected to the output mixer via it's kcontrol name. We can now connect the destination widget (wrt audio signal) with it's source widgets.
这个例子中的每个输入都有与之相关联的kcontrol（在上面的例子定义），
并连接到调音台的输出通过它的kcontrol名称。
我们现在可以连接目标部件（相对于音频信号）与它的源部件。

```
    /* output mixer */
    {"Output Mixer", "Line Bypass Switch", "Line Input"},
    {"Output Mixer", "HiFi Playback Switch", "DAC"},
    {"Output Mixer", "Mic Sidetone Switch", "Mic Bias"},
```

So we have :-

```
    Destination Widget  <=== Path Name <=== Source Widget
```

Or:-

```
    Sink, Path, Source
```

Or :-

```
    "Output Mixer" is connected to the "DAC" via the "HiFi Playback Switch".
```

When there is no path name connecting widgets (e.g. a direct connection) we pass NULL for the path name.
当没有连接部件的路径名（例如，直接连接），我们传递的路径名称为NULL。

Interconnections are created with a call to:-
互连创建一个调用：

```
snd_soc_dapm_connect_input(codec, sink, path, source);
```

Finally, snd_soc_dapm_new_widgets(codec) must be called after all widgets and interconnections have been registered with the core. This causes the core to scan the codec and machine so that the internal DAPM state matches the physical state of the machine.
最后，snd_soc_dapm_new_widgets（codec）必须被调用后，所有的部件和互连已登记的核心。
这能使得核心的codec和平台以便于使内部的动态音频电源管理(DAPM)状态匹配的机器的物理状态.

3.1 Machine Widget Interconnections
3.1 平台互联工具

Machine widget interconnections are created in the same way as codec ones and directly connect the codec pins to machine level widgets.

e.g. connects the speaker out codec pins to the internal speaker.

```
    /* ext speaker connected to codec pins LOUT2, ROUT2  */
    {"Ext Spk", NULL , "ROUT2"},
    {"Ext Spk", NULL , "LOUT2"},
```

This allows the DAPM to power on and off pins that are connected (and in use) and pins that are NC respectively.

4 Endpoint Widgets
==========

An endpoint is a start or end point (widget) of an audio signal within the machine and includes the codec. e.g.

o Headphone Jack
o Internal Speaker
o Internal Mic
o Mic Jack
o Codec Pins

When a codec pin is NC it can be marked as not used with a call to

snd_soc_dapm_set_endpoint(codec, "Widget Name", 0);

The last argument is 0 for inactive and 1 for active. This way the pin and its input widget will never be powered up and consume power.

This also applies to machine widgets. e.g. if a headphone is connected to a

jack then the jack can be marked active. If the headphone is removed, then the headphone jack can be marked inactive.

5 DAPM Widget Events

Some widgets can register their interest with the DAPM core in PM events. e.g. A Speaker with an amplifier registers a widget so the amplifier can be powered only when the spk is in use.

```
/* turn speaker amplifier on/off depending on use */
static int corgi_amp_event(struct snd_soc_dapm_widget *w, int event)
{
    gpio_set_value(CORGI_GPIO_APM_ON, SND_SOC_DAPM_EVENT_ON(event));
    return 0;
}

/* corgi machine dapm widgets */
static const struct snd_soc_dapm_widget wm8731_dapm_widgets =
    SND_SOC_DAPM_SPK("Ext Spk", corgi_amp_event);
```

Please see soc-dapm.h for all other widgets that support events.

5.1 Event types

The following event types are supported by event widgets.

```
/* dapm event types */
#define SND_SOC_DAPM_PRE_PMU    0x1     /* before widget power up */
#define SND_SOC_DAPM_POST_PMU    0x2        /* after widget power up */
#define SND_SOC_DAPM_PRE_PMD    0x4     /* before widget power down */
#define SND_SOC_DAPM_POST_PMD    0x8        /* after widget power down */
#define SND_SOC_DAPM_PRE_REG    0x10    /* before audio path setup */
#define SND_SOC_DAPM_POST_REG    0x20    /* after audio path setup */
```

linux音频alsa-uda134x驱动分析之六（插口）
http://www.usr.cc/thread-1745-1-1.html

ASoC jack detection
ASoC插口探测

ALSA has a standard API for representing physical jacks to user space, the kernel side of which can be seen in include/sound/jack.h. ASoC provides a version of this API adding two additional features:
ALSA对标准的API用以向用户空间提供物理插口，内核层面上的API可见于include/sound/jack.h.ASoC提供的这种API的一个版本增加了两个特性：

- It allows more than one jack detection method to work together on one user visible jack. In embedded systems it is common for multiple to be present on a single jack but handled by separate bits of hardware.
－它可以允许在一个用户可见插口上有多于一个插口探测方法共同工作。在嵌入式系统中，不同硬件的复用表示为一个单独的插口是很常见的。

- Integration with DAPM, allowing DAPM endpoints to be updated automatically based on the detected jack status (eg, turning off the headphone outputs if no headphones are present).
－与DAPM集成，允许DAPM端点根据检测到的插口状态自动升级（如：关闭耳机输出则不显示耳机）。

This is done by splitting the jacks up into three things working together: the jack itself represented by a struct snd_soc_jack, sets of snd_soc_jack_pins representing DAPM endpoints to update and blocks of code providing jack reporting mechanisms.
我们通过把插口分为三个协同工作的部分而完成这一工作：插口本身表示为一个snd_soc_jack结构，几组snd_soc_jack_pins表示SAPM端点来更新而且有一大堆代码提供插口报告机制。

For example, a system may have a stereo headset jack with two reporting mechanisms, one for the headphone and one for the microphone. Some systems won't be able to use their speaker output while a headphone is connected and so will want to make sure to update both speaker and headphone when the headphone jack status changes.
例如，一个系统可以有一个立体声耳麦插口，代有两个报告机制。耳机一个，麦克一个。有些系统在耳机连着的时候不允许扬声器发声，所以要保证当耳机插口状态改变时扬声器和耳机都要更新。

The jack - struct snd_soc_jack
插口－snd_soc_jack结构体
==========

This represents a physical jack on the system and is what is visible to user space. The jack itself is completely passive, it is set up by the machine driver and updated by jack detection methods.

这表示系统的一个物理插口，而且对用户空间可见。插口本身是完全被动的，它由机器驱动启动，由插口探测方法更新。

Jacks are created by the machine driver calling snd_soc_jack_new().
机器驱动调用snd_soc_jack_new()时插口建立。

snd_soc_jack_pin
snd_soc_jack_pin结构体
================

These represent a DAPM pin to update depending on some of the status bits supported by the jack. Each snd_soc_jack has zero or more of these which are updated automatically. They are created by the machine driver and associated with the jack using snd_soc_jack_add_pins(). The status of the endpoint may configured to be the opposite of the jack status if required (eg, enabling a built in microphone if a microphone is not connected via a jack).
这代表DAPM引脚根据插口支持的一些状态位进行更新。每个snd_soc_jack有0个或更多的 snd_soc_jack_pin 可以自动更新。 它们由机器驱动创建并且通过snd_soc_jack_add_pins()函数与插口相联系。需要的话，端点状态位可以设置为与插口状态相反（如：当没有插入耳机时使能一个内嵌的耳机）。

Jack detection methods
插口探测方法
======================

Actual jack detection is done by code which is able to monitor some input to the system and update a jack by calling snd_soc_jack_report(), specifying a subset of bits to update. The jack detection code should be set up by the machine driver, taking configuration for the jack to update and the set of things to report when the jack is connected.
实际上的插口探测是由代码完成的。这些代码可以指示一些系统输入而且通过调过snd_soc_jack_report()来更新插口，使一些位的子集更新状态。插口探测代码应由机器驱动设定，配置插口更新和插口连接时报告一系列的事情。

Often this is done based on the status of a GPIO - a handler for this is provided by the snd_soc_jack_add_gpio() function. Other methods are also available, for example integrated into CODECs. One example of CODEC integrated jack detection can be see in the WM8350 driver.
通常，这些的完成要基于一个GPIO的状态－它的句柄是有snd_soc_jack_add_gpio()函数提供的。还有一些别的方法可用，如集成在解码器里的。解码器集成的插口探测的一个例子可以在wm8350驱动中见到。

Each jack may have multiple reporting mechanisms, though it will need at least one to be useful.
每个插口可以有多个报告机制，其中至少要有一个是有用的。

Machine drivers
机器驱动
==========

These are all hooked together by the machine driver depending on the system hardware. The machine driver will set up the snd_soc_jack and the list of pins to update then set up one or more jack detection mechanisms to update that jack based on their current status.
这一切都由机器驱动根据系统硬件挂在一起。机器驱动会设置snd_soc_jack和要更新的引脚列表，然后启动一个或多个插口探测机制更新，这些插口是依赖于它们当前的状态的。

linux音频alsa-uda134x驱动分析之七（机器驱动）
http://www.usr.cc/thread-1746-1-1.html

ASoC Machine Driver
ASoC机器驱动
==========

The ASoC machine (or board) driver is the code that glues together the platform and codec drivers.
ASoC机器或板级驱动是把平台和解码器驱动粘合在一起的代码。

The machine driver can contain codec and platform specific code. It registers the audio subsystem with the kernel as a platform device and is represented by the following struct:-
机器驱动可以包含解码器和平台相关代码。它把音频子系统注册为内核中的一个平台设备，并由下面的结构体表示：

```
/* SoC machine */
struct snd_soc_card {
    char *name;

    int (*probe)(struct platform_device *pdev);
    int (*remove)(struct platform_device *pdev);

    /* the pre and post PM functions are used to do any PM work before and
     * after the codec and DAIs do any PM work. */
```

```
    int (*suspend_pre)(struct platform_device *pdev, pm_message_t state);
    int (*suspend_post)(struct platform_device *pdev, pm_message_t state);
    int (*resume_pre)(struct platform_device *pdev);
    int (*resume_post)(struct platform_device *pdev);

    /* machine stream operations */
    struct snd_soc_ops *ops;

    /* CPU <--> Codec DAI links  */
    struct snd_soc_dai_link *dai_link;
    int num_links;
};
```

probe()/remove()
探测和移除函数
---
probe/remove are optional. Do any machine specific probe here.
probe/remove（探测和移除函数）是可选的。可以做一些机器相关的探测。

suspend()/resume()
挂起和恢复函数

The machine driver has pre and post versions of suspend and resume to take care of any machine audio tasks that have to be done before or after the codec, DAIs and DMA is suspended and resumed. Optional.
机器驱动有一前后两个版本的挂起和恢复函数来管理在解码前后要完成的机器音频任务。DAI和DMA都要挂起和恢复。这也是可选的。

Machine operations
机器操作

The machine specific audio operations can be set here. Again this is optional.
机器相关音频操作可以在这里设定。这也是可选的。

Machine DAI Configuration
机器数字音频接口配置

The machine DAI configuration glues all the codec and CPU DAIs together. It can also be used to set up the DAI system clock and for any machine related DAI initialisation e.g. the machine audio map can be connected to the codec audio map, unconnected codec pins can be set as such. Please see corgi.c, spitz.c

for examples.
机器的DAI配置把所有的解码器和CPU DAI粘合在一起。它也可以用来启动DAI系统时钟或机器相在DAI的初始化。如机器音频映像可以与解码器音频映像连在一起。非相连的解码器引脚也可以如此设置。例子请见corgi.c,spitz.c.

struct snd_soc_dai_link is used to set up each DAI in your machine. e.g.
结构体snd_soc_dai_link设置你机器的DAI。例如：

```
/* corgi digital audio interface glue - connects codec <--> CPU */
static struct snd_soc_dai_link corgi_dai = {
    .name = "WM8731",
    .stream_name = "WM8731",
    .cpu_dai = &pxa_i2s_dai,
    .codec_dai = &wm8731_dai,
    .init = corgi_wm8731_init,
    .ops = &corgi_ops,
};
```

struct snd_soc_card then sets up the machine with it's DAIs. e.g.
然后snd_soc_card用这个DAI来配置机器,例如：

```
/* corgi audio machine driver */
static struct snd_soc_card snd_soc_corgi = {
    .name = "Corgi",
    .dai_link = &corgi_dai,
    .num_links = 1,
};
```

Machine Audio Subsystem
机器音频子系统

The machine soc device glues the platform, machine and codec driver together.
Private data can also be set here. e.g.
机器soc 设备把平台，机器，解码器驱动结合在一起。
这里也可以设置私有数据。

```
/* corgi audio private data */
static struct wm8731_setup_data corgi_wm8731_setup = {
    .i2c_address = 0x1b,
};

/* corgi audio subsystem */
```

```
static struct snd_soc_device corgi_snd_devdata = {
    .machine = &snd_soc_corgi,
    .platform = &pxa2xx_soc_platform,
    .codec_dev = &soc_codec_dev_wm8731,
    .codec_data = &corgi_wm8731_setup,
};
```

Machine Power Map
机器电源映像

The machine driver can optionally extend the codec power map and to become an audio power map of the audio subsystem. This allows for automatic power up/down of speaker/HP amplifiers, etc. Codec pins can be connected to the machines jack sockets in the machine init function. See soc/pxa/spitz.c and dapm.txt for details.
机器驱动可以扩展解码器的电源映像，这样就变成一个音频子系统的音频电源映像。这允许扬声器／HP放大器等 源的自动通断。解码器引脚可以在机器初始化函数中连到机器插口中。详情请见soc/pxa/spitx.c和dapm.txt.

Machine Controls
机器控制

Machine specific audio mixer controls can be added in the DAI init function.
机器相关的音频混音控制可以加入到DAI初始化函数中。

linux音频alsa-uda134x驱动分析之八（平台驱动）
http://www.usr.cc/forum.php?mod=viewthread&tid=1747&fromuid=5566

ASoC Platform Driver
ASoC平台驱动
===

An ASoC platform driver can be divided into audio DMA and SoC DAI configuration and control. The platform drivers only target the SoC CPU and must have no board specific code.
一个ASoC平台驱动可以分为音频DAM和SoC DAI配置和控制。平台驱动只锁定平台处理器为目标，必须不包含任何板级相关代码。

Audio DMA
音频DMA
=========

The platform DMA driver optionally supports the following ALSA operations:-
平台DMA驱动可以选择性地支持下面的ALSA操作：

```
/* SoC audio ops */
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);
};
```

The platform driver exports its DMA functionality via struct snd_soc_platform:-
平台驱动通过snd_doc_platform结构导出其DMA功能：

```
struct snd_soc_platform {
    char *name;

    int (*probe)(struct platform_device *pdev);
    int (*remove)(struct platform_device *pdev);
    int (*suspend)(struct platform_device *pdev, struct snd_soc_cpu_dai
*cpu_dai);
    int (*resume)(struct platform_device *pdev, struct snd_soc_cpu_dai
*cpu_dai);

    /* pcm creation and destruction */
    int (*pcm_new)(struct snd_card *, struct snd_soc_codec_dai *, struct
snd_pcm *);
    void (*pcm_free)(struct snd_pcm *);

    /* platform stream ops */
    struct snd_pcm_ops *pcm_ops;
};
```

Please refer to the ALSA driver documentation for details of audio DMA.
请参考ALSA驱动文档以更加详尽地了解音频DMA。
http://www.alsa-project.org/~iwai/writing-an-alsa-driver/c436.htm

An example DMA driver is soc/pxa/pxa2xx-pcm.c

DMA驱动的一个例子：soc/pxa/pxa2xx-pcm.c

SoC DAI Drivers
板级DAI驱动
==========

Each SoC DAI driver must provide the following features:-
每个SoC DAI驱动都必须提供如下性能：

1) Digital audio interface (DAI) description
1)数字音频接口描述
2) Digital audio interface configuration
2)数字音频接口配置
3) PCM's description
3）PCM描述
4) SYSCLK configuration
4）系统时钟配置
5) Suspend and resume (optional)
5）挂起和恢复（可选的）

Please see codec.txt for a description of items 1 - 4.
对1－4项的描述请见codec.txt

linux音频alsa-uda134x驱动分析之九（POP声）
http://www.usr.cc/thread-1748-1-1.html

Audio Pops and Clicks
音频pop声
==========

Pops and clicks are unwanted audio artifacts caused by the powering up and down of components within the audio subsystem. This is noticeable on PCs when an audio module is either loaded or unloaded (at module load time the sound card is
powered up and causes a popping noise on the speakers).
咔咔声是因为音频子系统中的元件的电源开启和关闭产生的意处杂音。
在PC中，当一个音频设备被加载和御载时是很明显的（加载时声卡上电，扬声器产生躁声）。

Pops and clicks can be more frequent on portable systems with DAPM. This is because the components within the subsystem are being dynamically powered depending on the audio usage and this can subsequently cause a small pop or click every time a component power state is changed.

Minimising Playback Pops and Clicks
==========

Playback pops in portable audio subsystems cannot be completely eliminated currently, however future audio codec hardware will have better pop and click suppression. Pops can be reduced within playback by powering the audio components in a specific order. This order is different for startup and shutdown and follows some basic rules:-

Startup Order :- DAC --> Mixers --> Output PGA --> Digital Unmute

Shutdown Order :- Digital Mute --> Output PGA --> Mixers --> DAC

This assumes that the codec PCM output path from the DAC is via a mixer and then a PGA (programmable gain amplifier) before being output to the speakers.

Minimising Capture Pops and Clicks
==========

Capture artifacts are somewhat easier to get rid as we can delay activating the ADC until all the pops have occurred. This follows similar power rules to playback in that components are powered in a sequence depending upon stream startup or shutdown.

Startup Order - Input PGA --> Mixers --> ADC

Shutdown Order - ADC --> Mixers --> Input PGA

Zipper Noise
==========
An unwanted zipper noise can occur within the audio playback or capture stream when a volume control is changed near its maximum gain value. The zipper noise is heard when the gain increase or decrease changes the mean audio signal amplitude too quickly. It can be minimised by enabling the zero cross setting for each volume control. The ZC forces the gain change to occur when the signal crosses the zero amplitude line.

读完上面的内容,让我们来看 2.6.30 内核中的原代码,这部分代码经过 make menuconfig 中进行简单的配制可以产生声音,但是声音不好,听网友说是有 bug.

但于我们分析没有太大的问题。通过这一系列文章的第一篇，也就是内核文档中的overview.txt，我们可以了解到，一个基本的 SOC 音频驱动应该包括三个部分：
1、解码器驱动
2、平台驱动
3、机器驱动
听起来有些迷糊，为什么要分为三个驱动，每个驱动起的是什么作用呢？
为什么要分为三个驱动呢？这是有意而为之的，这样是为了提高代码利用率，比如说uda1341 这个解码芯片，能用这块芯片进行解码的机型有很多，好像很多开发板都可以采用这块片子，而我们的内核以前就要为每一个机型写一分驱动，其中都包含 uda1341 部分的这段代码，这样很不好，很费事。因些代码作者决定把 uda1341 从中隔离出来，我们现在的原理是：
我们有多个解码器代码，多个平台代码。当我们要为一个机器写驱动时，只要把这个机器所有的平台和所用的解码器关联起来就可以了。
这里平台（platform)和机器(machine)的意思大家还要分清。比如说我们国内现在流行有多种开发板，这些开发板，我们叫做机器（machine).而所有这些开发板都是基于 arm9 的，都是 s3c24xx 系列的芯片做处理器，我们说这些机器都是 s3c24xx 平台的。

了解了上面的内容，我们可以来看代码了，对应三个驱动，我们有三个文件：
/linux-2.6.30/sound/soc/s3c24xx/s3c24xx-i2s.c 这个文件就是所谓的平台驱动(platform driver).从名字可看到，s3c24xx 平台用的是 I2S 音频总线。
/yellowater/linux-2.6.30/sound/soc/s3c24xx/s3c24xx_uda134x.c 这个文件是所谓的机器驱动(machine driver)从名字可以看出，是把平台 s3c34xx 和解码器 uda134x 相关在一起了。

/yellowater/linux-2.6.30/sound/soc/codecs/uda134x.c 这个文件显然就是解码器芯片驱动了。
下一步，我们将详细分析三个文件。请见下一篇文章。

linux音频alsa-uda134x驱动分析之十一（解码器驱动文件分析）
http://www.usr.cc/thread-1751-1-1.html

我里我们要对照原来的文档来看代码。
我们要打开一份代码：/linux-2.6.30/sound/soc/codecs/uda134x.c
还要打开第三篇文章的帖子

我们先不看具体实现，而是来看代码的最后面注册到系统中的结构体：
```
static int __init uda134x_init(void)
{
        return snd_soc_register_dai(&uda134x_dai);
}
module_init(uda134x_init);
```

要注册到系统中的结构体是：uda134x_dai, 注册时用的函数是：snd_soc_register_dai();

然后我们来看uda134x_dai 是一个什么样的结构体：

```
struct snd_soc_dai uda134x_dai = {
        .name = "UDA134X",
        /* playback capabilities */
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = UDA134X_RATES,
                .formats = UDA134X_FORMATS,
        },
        /* capture capabilities */
        .capture = {
                .stream_name = "Capture",
                .channels_min = 1,
                .channels_max = 2,
                .rates = UDA134X_RATES,
                .formats = UDA134X_FORMATS,
        },
        /* pcm operations */
        .ops = &uda134x_dai_ops,
};
EXPORT_SYMBOL(uda134x_dai);
```

这个结构体类型是：snd_soc_dai,
成员包括：
.name = "UDA134X",
.playback
.capture
.ops = &uda134x_dai_ops,

很显然，这与文档中提到的结构体并不一样，文档中提到的结构体类型是snd_soc_codec_dai，而提到的成员是一样，很自然，我们想到提文档涉后了，不是最新的内容。为证实我们的设想，我们去文档中提到的wm8731.c中去看一看，这个代码与uda1341的代码结构不同，因为它把总线也放进去了，但这不影响我们找到下面的字样：

```
struct snd_soc_dai wm8731_dai
snd_soc_register_dai(&wm8731_dai);
```

显然，这个代码，与文档中提到的也不一样，而与我们uda134x.c 中的代码是一致的。因为我们的设想是正确的。

下面我们再接着看这个结构体里的成员：
.name无需多讲，是一个字符串。
.playback
.capture也文档中常提到的两个功能，即播放和捕获，着我们也先不细看。

.ops=&uda134x_dai_ops,
我们来这个.ops项。
文档中描述的snd_soc_codec_dai与我们现在看到的snd_soc_dai还是有些不同之处的。
那就是文档中比现在我们看到的代码多了一项：

.....

```
  .ops = {
        .prepare = wm8731_pcm_prepare,
        .hw_params = wm8731_hw_params,
        .shutdown = wm8731_shutdown,
    },
    /* DAI operations - see DAI.txt */
    .dai_ops = {
        .digital_mute = wm8731_mute,
        .set_sysclk = wm8731_set_dai_sysclk,
        .set_fmt = wm8731_set_dai_fmt,
```

这与我们现在的代码是怎么对应的呢？我们来看我们现在的代码：

```
static struct snd_soc_dai_ops uda134x_dai_ops = {
        .startup        = uda134x_startup,
        .shutdown        = uda134x_shutdown,
        .hw_params        = uda134x_hw_params,
        .digital_mute        = uda134x_mute,
        .set_sysclk        = uda134x_set_dai_sysclk,
        .set_fmt        = uda134x_set_dai_fmt,
};
```

很显然，较新的代码把原来的.ops项和dai_ops合成了一项,并且改成了引用的放式传给snd_soc_dai结构体。
下面我们来看.ops中的各个成员，总共6个。
1、.startup= uda134x_startup

```
static int uda134x_startup(struct snd_pcm_substream *substream,
        struct snd_soc_dai *dai)
{

        struct snd_soc_pcm_runtime *rtd = substream->private_data;
        struct snd_soc_device *socdev = rtd->socdev;
        struct snd_soc_codec *codec = socdev->card->codec;
        struct uda134x_priv *uda134x = codec->private_data;
        struct snd_pcm_runtime *master_runtime;
        if (uda134x->master_substream) {
                master_runtime = uda134x->master_substream->runtime;
```

```
                pr_debug("%s constraining to %d bits at %d\n", __func__,
                         master_runtime->sample_bits,
                         master_runtime->rate);
                snd_pcm_hw_constraint_minmax(substream->runtime,
                                             SNDRV_PCM_HW_PARAM_RATE,
                                             master_runtime->rate,
                                             master_runtime->rate);
                snd_pcm_hw_constraint_minmax(substream->runtime,

SNDRV_PCM_HW_PARAM_SAMPLE_BITS,
                                             master_runtime->sample_bits,
                                             master_runtime->sample_bits);
                uda134x->slave_substream = substream;

        } else
                uda134x->master_substream = substream;
        return 0;
}
```

linux下没有source insight还真是不方便啊，哪天试下source nav以前在虚拟机里试后，跑不起来，卡死。不过还好有lxr.linux.no.
我们看到，这个函数传入两个参数：snd_pcm_substream *substream，snd_soc_dai *dai后面的参数很显然在调用时会传我们上面刚刚讲的uda134x_dai本身了。前一个参数呢？

```
struct snd_pcm_substream {
350         struct snd_pcm *pcm;
351         struct snd_pcm_str *pstr;
352         void *private_data;             /* copied from pcm->private_data */
353         int number;
354         char name[32];                  /* substream name */
355         int stream;                     /* stream (direction) */
356         char latency_id[20];            /* latency identifier */
357         size_t buffer_bytes_max;        /* limit ring buffer size */
358         struct snd_dma_buffer dma_buffer;
359         unsigned int dma_buf_id;
360         size_t dma_max;
361         /* -- hardware operations -- */
362         struct snd_pcm_ops *ops;
363         /* -- runtime information -- */
364         struct snd_pcm_runtime *runtime;
365         /* -- timer section -- */
366         struct snd_timer *timer;                /* timer */
367         unsigned timer_running: 1;      /* time is running */
```

```
368         /* -- next substream -- */
369         struct snd_pcm_substream *next;
370         /* -- linked substreams -- */
371         struct list_head link_list;     /* linked list member */
372         struct snd_pcm_group self_group;        /* fake group for non linked
substream (with substream lock inside) */
373         struct snd_pcm_group *group;            /* pointer to current group
*/
374         /* -- assigned files -- */
375         void *file;
376         int ref_count;
377         atomic_t mmap_count;
378         unsigned int f_flags;
379         void (*pcm_release)(struct snd_pcm_substream *);
380#if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
381         /* -- OSS things -- */
382         struct snd_pcm_oss_substream oss;
383#endif
384#ifdef CONFIG_SND_VERBOSE_PROCFS
385         struct snd_info_entry *proc_root;
386         struct snd_info_entry *proc_info_entry;
387         struct snd_info_entry *proc_hw_params_entry;
388         struct snd_info_entry *proc_sw_params_entry;
389         struct snd_info_entry *proc_status_entry;
390         struct snd_info_entry *proc_prealloc_entry;
391         struct snd_info_entry *proc_prealloc_max_entry;
392#endif
393         /* misc flags */
394         unsigned int hw_opened: 1;
395};
```

很长的一个结构体啊，(注：pcm=脉冲编码调制)，
到这里发现自己对pcm音频一无怕知，无法说下去，第十二部分开始探索pcm.

Codec：音频芯片的控制，比如静音、打开（关闭）ADC（DAC)、设置ADC（DAC）的增益、耳机模式的检测等操作。
I2S:数字音频接口,用于CPU和Codec之间的数字音频流raw data的传输。每当有playback或record操作时，snd_soc_dai_ops. prepare()会被调用，启动I2S总线。
PCM：我不知道为什么会取这个模块名，它其实是定义DMA操作的，用于将音频数据通过DMA传到I2S控制器的FIFO中。
音频数据流向：RAM--(dma)-->I2S Controller FIFO--(i2s)-->Codec-->Speaker/Headset

PCM模块初始化

调用snd_soc_register_platform()向ALSACore注册一个snd_soc_platform结构体。

```
struct snd_soc_platform loon_soc_platform = {
        .name           = " loon -pcm-audio",
        .pcm_ops        = & loon _pcm_ops,
        .pcm_new        = loon _pcm_new,
        .pcm_free       = loon _pcm_free_dma_buffers,
        .suspend        = loon _pcm_suspend,
        .resume         = loon _pcm_resume,
};
struct snd_soc_platform loon_soc_platform = {
        .name           = " loon -pcm-audio",
        .pcm_ops        = & loon _pcm_ops,
        .pcm_new        = loon _pcm_new,
        .pcm_free       = loon _pcm_free_dma_buffers,
        .suspend        = loon _pcm_suspend,
        .resume         = loon _pcm_resume,
};
```

成员pcm_new需要调用dma_alloc_writecombine()给DMA分配一块write-combining的内存空间，并把这块缓冲区的相关信息保存到substream->dma_buffer中，相当于构造函数。pcm_free则相反。

这些成员函数都还算简单，看看代码即可以理解其流程。

snd_pcm_ops

接着我们看一下snd_pcm_ops结构体，该结构体的操作集函数的实现是本模块的主体。

```
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 (*copy)(struct snd_pcm_substream *substream, int channel,
                    snd_pcm_uframes_t pos,
                    void __user *buf, snd_pcm_uframes_t count);
        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);
};
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 (*copy)(struct snd_pcm_substream *substream, int channel,
                    snd_pcm_uframes_t pos,
                    void __user *buf, snd_pcm_uframes_t count);
        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);
};
```

我们主要实现open、close、hw_params、hw_free、prepare和trigger接口。

open函数

open函数为PCM模块设定支持的传输模式、数据格式、通道数、period等参数，并为playback/capture stream分配相应的DMA通道。

其一般实现如下：

```
static int loon_pcm_open(struct snd_pcm_substream *substream)
{
        struct snd_soc_pcm_runtime *rtd = substream->private_data;
        struct snd_soc_dai *cpu_dai = rtd->dai->cpu_dai;
        struct snd_pcm_runtime *runtime = substream->runtime;
        struct audio_stream_a *s = runtime->private_data;
        int ret;
```

```
	if (!cpu_dai->active) {
		audio_dma_request(&s[0],  audio_dma_callback);  // 为 playback
stream分配DMA
		audio_dma_request(&s[1],  audio_dma_callback);  // 为 capture
stream分配DMA
	}

	//设定runtime硬件参数
	snd_soc_set_runtime_hwparams(substream, &loon_pcm_hardware);

	/* Ensure that buffer size is a multiple of period size */
	ret = snd_pcm_hw_constraint_integer(runtime,
				SNDRV_PCM_HW_PARAM_PERIODS);

	return ret;
}
static int loon_pcm_open(struct snd_pcm_substream *substream)
{
	struct snd_soc_pcm_runtime *rtd = substream->private_data;
	struct snd_soc_dai *cpu_dai = rtd->dai->cpu_dai;
	struct snd_pcm_runtime *runtime = substream->runtime;
	struct audio_stream_a *s = runtime->private_data;
	int ret;

	if (!cpu_dai->active) {
		audio_dma_request(&s[0],  audio_dma_callback);  // 为 playback
stream分配DMA
		audio_dma_request(&s[1],  audio_dma_callback);  // 为 capture
stream分配DMA
	}

	//设定runtime硬件参数
	snd_soc_set_runtime_hwparams(substream, &loon_pcm_hardware);

	/* Ensure that buffer size is a multiple of period size */
	ret = snd_pcm_hw_constraint_integer(runtime,
				SNDRV_PCM_HW_PARAM_PERIODS);

	return ret;
}
```

其中硬件参数要根据芯片的数据手册来定义，如：

```
static const struct snd_pcm_hardware loon_pcm_hardware = {
	.info			= SNDRV_PCM_INFO_INTERLEAVED |
```

```
                                SNDRV_PCM_INFO_BLOCK_TRANSFER |
                                SNDRV_PCM_INFO_MMAP |
                                SNDRV_PCM_INFO_MMAP_VALID |
                                SNDRV_PCM_INFO_PAUSE |
                                SNDRV_PCM_INFO_RESUME,
        .formats          = SNDRV_PCM_FMTBIT_S16_LE |
                                SNDRV_PCM_FMTBIT_U16_LE |
                                SNDRV_PCM_FMTBIT_U8 |
                                SNDRV_PCM_FMTBIT_S8,
        .channels_min     = 2,
        .channels_max     = 2,
        .buffer_bytes_max = 128*1024,
        .period_bytes_min = PAGE_SIZE,
        .period_bytes_max = PAGE_SIZE*2,
        .periods_min      = 2,
        .periods_max      = 128,
        .fifo_size        = 32,
};
static const struct snd_pcm_hardware loon_pcm_hardware = {
        .info             = SNDRV_PCM_INFO_INTERLEAVED |
                                SNDRV_PCM_INFO_BLOCK_TRANSFER |
                                SNDRV_PCM_INFO_MMAP |
                                SNDRV_PCM_INFO_MMAP_VALID |
                                SNDRV_PCM_INFO_PAUSE |
                                SNDRV_PCM_INFO_RESUME,
        .formats          = SNDRV_PCM_FMTBIT_S16_LE |
                                SNDRV_PCM_FMTBIT_U16_LE |
                                SNDRV_PCM_FMTBIT_U8 |
                                SNDRV_PCM_FMTBIT_S8,
        .channels_min     = 2,
        .channels_max     = 2,
        .buffer_bytes_max = 128*1024,
        .period_bytes_min = PAGE_SIZE,
        .period_bytes_max = PAGE_SIZE*2,
        .periods_min      = 2,
        .periods_max      = 128,
        .fifo_size        = 32,
};
```

关于peroid的概念有这样的描述：The “period” is a term that corresponds to a fragment in the OSS world. The period defines the size at which a PCM interrupt is generated.
上层ALSA lib可以通过接口来获得这些参数的，如snd_pcm_hw_params_get_buffer_size_max()来取得buffer_bytes_max。

关于DMA的中断处理
另外留意audio_dma_request(&s[0], audio_dma_callback);中的audio_dma_callback，这是dma的中断函数，这里以callback的形式存在，其实到dma的底层还是这样的形式：static irqreturn_t dma_irq_handler(int irq, void *dev_id)，在DMA中断处理dma_irq_handler()中调用callback。这些跟具体硬件平台的DMA实现相关，如果没有类似的机制，那么还是要在pcm模块中实现这个中断。

```
/*
 *  This is called when dma IRQ occurs at the end of each transmited block
 */
static void audio_dma_callback(void *data)
{
	struct audio_stream_a *s = data;

	/*
	 * If we are getting a callback for an active stream then we inform
	 * the PCM middle layer we've finished a period
	 */
	if (s->active)
		snd_pcm_period_elapsed(s->stream);

	spin_lock(&s->dma_lock);
	if (s->periods > 0)
		s->periods--;

	audio_process_dma(s); //dma启动
	spin_unlock(&s->dma_lock);
}
/*
 *  This is called when dma IRQ occurs at the end of each transmited block
 */
static void audio_dma_callback(void *data)
{
	struct audio_stream_a *s = data;

	/*
	 * If we are getting a callback for an active stream then we inform
	 * the PCM middle layer we've finished a period
	 */
	if (s->active)
		snd_pcm_period_elapsed(s->stream);

	spin_lock(&s->dma_lock);
```

```
        if (s->periods > 0)
                s->periods--;

        audio_process_dma(s); //dma启动
        spin_unlock(&s->dma_lock);
}
```

hw_params函数

hw_params函数为substream（每打开一个playback或capture，ALSACore均产生相应的一个substream）设定DMA的源（目的）地址，以及DMA缓冲区的大小。

```
static int loon_pcm_hw_params(struct snd_pcm_substream *substream,
                              struct snd_pcm_hw_params *params)
{
        struct snd_pcm_runtime *runtime = substream->runtime;
        int err = 0;

        snd_pcm_set_runtime_buffer(substream, &substream->dma_buffer);
        runtime->dma_bytes = params_buffer_bytes(params);
        return err;
}
static int loon_pcm_hw_params(struct snd_pcm_substream *substream,
                              struct snd_pcm_hw_params *params)
{
        struct snd_pcm_runtime *runtime = substream->runtime;
        int err = 0;

        snd_pcm_set_runtime_buffer(substream, &substream->dma_buffer);
        runtime->dma_bytes = params_buffer_bytes(params);
        return err;
}
```

hw_free是hw_params的相反操作,调用snd_pcm_set_runtime_buffer(substream, NULL)即可。

注：代码中的dma_buffer 是DMA缓冲区，它通过4个字段定义：dma_area、dma_addr、dma_bytes和dma_private。其中dma_area是缓冲区逻辑地址，dma_addr是缓冲区的物理地址，dma_bytes是缓冲区的大小，dma_private是ALSA的DMA管理用到的。dma_buffer是在pcm_new()中初始化的；当然也可以把分配dma缓冲区的工作放到这部分来实现，但考虑到减少碎片，故还是在pcm_new中以最大size（即buffer_bytes_max）来分配。

prepare函数

当pcm“准备好了”调用该函数。在这里根据channels、buffer_bytes等来设定DMA传输

参数，跟具体硬件平台相关。
注：每次调用snd_pcm_prepare()的时候均会调用prepare函数。

trigger函数
当pcm开始、停止、暂停的时候都会调用trigger函数。

```
static int loon_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
{
	struct runtime_data *prtd = substream->runtime->private_data;
	int ret = 0;

	spin_lock(&prtd->lock);

	switch (cmd) {
	case SNDRV_PCM_TRIGGER_START:
	case SNDRV_PCM_TRIGGER_RESUME:
	case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
		prtd->state |= ST_RUNNING;
		dma_ctrl(prtd->params->channel, DMAOP_START); //DMA开启
		break;

	case SNDRV_PCM_TRIGGER_STOP:
	case SNDRV_PCM_TRIGGER_SUSPEND:
	case SNDRV_PCM_TRIGGER_PAUSE_PUSH:
		prtd->state &= ~ST_RUNNING;
		dma_ctrl(prtd->params->channel, DMAOP_STOP); //DMA停止
		break;

	default:
		ret = -EINVAL;
		break;
	}

	spin_unlock(&prtd->lock);

	return ret;
}
static int loon_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
{
	struct runtime_data *prtd = substream->runtime->private_data;
	int ret = 0;

	spin_lock(&prtd->lock);

	switch (cmd) {
```

```
        case SNDRV_PCM_TRIGGER_START:
        case SNDRV_PCM_TRIGGER_RESUME:
        case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
                prtd->state |= ST_RUNNING;
                dma_ctrl(prtd->params->channel, DMAOP_START); //DMA开启
                break;

        case SNDRV_PCM_TRIGGER_STOP:
        case SNDRV_PCM_TRIGGER_SUSPEND:
        case SNDRV_PCM_TRIGGER_PAUSE_PUSH:
                prtd->state &= ~ST_RUNNING;
                dma_ctrl(prtd->params->channel, DMAOP_STOP); //DMA停止
                break;

        default:
                ret = -EINVAL;
                break;
        }

        spin_unlock(&prtd->lock);

        return ret;
}
```

Trigger函数里面的操作应该是原子的，不要在调用这些操作时进入睡眠，trigger函数应尽量小，甚至仅仅是触发DMA。

pointer函数

static snd_pcm_uframes_t wmt_pcm_pointer(struct snd_pcm_substream *substream)

PCM中间层通过调用这个函数来获取缓冲区的位置。一般情况下，在中断函数中调用snd_pcm_period_elapsed()或在pcm中间层更新buffer的时候调用它。然后pcm中间层会更新指针位置和计算缓冲区可用空间，唤醒那些在等待的线程。这个函数也是原子的。

snd_pcm_runtime

我们会留意到ops各成员函数均需要取得一个snd_pcm_runtime结构体指针，这个指针可以通过substream->runtime来获得。snd_pcm_runtime是pcm运行时的信息。当打开一个pcm子流时，pcm运行时实例就会分配给这个子流。它拥有很多多种信息：hw_params和sw_params配置拷贝，缓冲区指针，mmap记录，自旋锁等。snd_pcm_runtime对于驱动程序操作集函数是只读的，仅pcm中间层可以改变或更新这些信息。