1 开始之前

1.1 目的前提

本文目的帮助理解音频系统中的 dapm 机制建立和触发过程，在这之中，涉及到音频路径的搭建和选择（相信这部分对于大家来说更有应用意义）。在 dapm 机制深入分析中，会展现 alsa 设计者的各种理念及实现技巧，很多细节上的考虑和处理值得我们去学习。

前提是必须掌握基本的音频系统知识，对 AUDIO CODEC 驱动有一定的了解，这方面有一份很好的文档《write an alsa driver》可以参考。另外最好熟练使用 alsa_amixer、alsa_aplay 这两个基本音频调试工具。

1.2 说明

dapm 除了动态音频电源管理外，还肩负了音频路径建立的重任。下文会先阐述音频路径，再到动态音频电源管理，因为 dapm 的触发条件之一是混音器设置。而在这之前，会相当长的篇幅用于介绍混音器设置软件 alsa_amixer 的使用方法、mixer controls 的建立和触发过程。

文中颜色说明：

浅灰色	：	表征引用，摘自其他文档
浅蓝色	：	表征命令行输入
紫罗兰	：	表征流程简要说明

1.3 版权声明

关于版权问题，欢迎转载，但请注明作者和来源，如下：

```
FILE    : AUDIO CODEC DAPM
AUTHOR  : Loon Zhong <sepnic@gmail.com>
BLOG    : http://blog.csdn.net/sepnic
```

如有纠错、疑问、建议，feel free to contact me，谢谢！

2 alsa_amixer

alsa_amixer 是 alsa utils 中的一个，是音频混音器配置工具，如控制某个开关（switch）和调节器（slider）。比如打开录音通道、调整音量大小等这些都可以通过 alsa_amixer 来控制。因此，alsa_amixer 是音频驱动开发中的常用工具。其用法是相当简单的，如下：

```
~ # alsa_amixer help
Usage: amixer <options> [command]

Available options:
  -h,--help        this help
  -c,--card N      select the card
  -D,--device N    select the device, default 'default'
  -d,--debug       debug mode
  -n,--nocheck     do not perform range checking
  -v,--version     print version of this program
  -q,--quiet       be quiet
  -i,--inactive    show also inactive controls
  -a,--abstract L  select abstraction level (none or basic)
  -s,--stdin       Read and execute commands from stdin sequentially

Available commands:
  scontrols        show all mixer simple controls
  scontents        show contents of all mixer simple controls (default command)
  sset sID P       set contents for one mixer simple control
  sget sID         get contents for one mixer simple control
  controls         show all controls for given card
  contents         show contents of all controls for given card
  cset cID P       set control contents for one control
  cget cID         get control contents for one control
```

留意命令字cset/sset、cget/sget，带c前缀的表示操作对象是mixer controls，带s前缀的表示操作对象是mixer simple controls。那么这两者有什么区别呢？这里不累述，详见：http://blog.csdn.net/sepnic/article/details/6324901

例如，我们知道一个 numid=3,iface=MIXER,name='Playback Volume'的 mixer control，该 control 用于调整回放音量，对这个 control 设置值 100，则有：

```
amixer cset numid=3,iface=MIXER,name='Playback Volume' 100
amixer cset numid=3 100
amixer cset name='Playback Volume' 100
```

以上三种方法都可以的，对于参数 cID，只提供 numid 或 name 都可以找到正确的底层的 kcontrol。

3 mixer control

kcontrol 对于许多开关（switch）和调节器（slider）应用广泛，它能被用户空间存取，从而读写 CODEC 相关寄存器。kcontrol 主要用于 mixer（故称为 mixer control），用 snd_kcontrol_new 结构体描述。
注:kcontrol 是 kernel control 的简称,即内核态的控制接口;代码中还会发现有 ucontrol，这是 user control 的简称，保存用户层传递进来的参数。

3.1 snd_kcontrol_new

```
struct snd_kcontrol_new {
    snd_ctl_elem_iface_t iface; /* interface identifier */
    unsigned int device;        /* device/client number */
    unsigned int subdevice;     /* subdevice (substream) number */
    unsigned char *name;        /* ASCII name of item */
    unsigned int index;     /* index of item */
    unsigned int access;        /* access rights */
    unsigned int count;     /* count of same elements */
    snd_kcontrol_info_t *info;
    snd_kcontrol_get_t *get;
    snd_kcontrol_put_t *put;
    union {
        snd_kcontrol_tlv_rw_t *c;
        const unsigned int *p;
    } tlv;
    unsigned long private_value;
};
```

iface
定义了 kcontrol 的类型，形式为 SNDRV_CTL_ELEM_IFACE_xxx，对于 mixer 是 SNDRV_CTL_ELEM_IFACE_MIXER，对于不属于 mixer 的全局控制，使用 CARD；如果关联到某类设备，则是 PCM、RAWMIDI、TIMER 或 SEQUENCER。在这里，我们主要关注 mixer。

access
访问控制权限。SNDRV_CTL_ELEM_ACCESS_READ 意味着只读，这时 put()函数不必实现；SNDRV_CTL_ELEM_ACCESS_WRITE 意味着只写，这时 get()函数不必实现。若 control 值频繁变化，则需定义 VOLATILE 标志。当 kcontrol 处于非激活状态时，应设置 INACTIVE 标志。

private_value
包含 1 个长整型值，可以通过它给 info()、get()和 put()这些回调函数传递参数。

name

名称标识，这个字段非常重要，因为 kcontrol 的作用由名称来区分，对于名称相同的 kcontrol，则使用 index 区分。下面会详细介绍上层应用如何根据 name 名称标识来找到底层相应的 kcontrol。

name 定义的标准是“SOURCE DIRECTION FUNCTION”即“源 方向 功能”，SOURCE 定义了 control 的源，如“Master”、“PCM”等；DIRECTION 则为“Playback”、“Capture”等，如果 DIRECTION 忽略，意味着 Playback 和 capture 双向；FUNCTION 则可以是“Switch”、“Volume”和“Route”等。详细参考内核文档 ControlNames.txt。

3.2 kcontrol 宏

alsa 中定义各种宏用于 kcontrol 的初始化，详见 soc.h，以下仅举一两例。

```
#define SOC_SINGLE(xname, reg, shift, max, invert) /
{   .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, /
    .info = snd_soc_info_volsw, .get = snd_soc_get_volsw,/
    .put = snd_soc_put_volsw, /
    .private_value =  SOC_SINGLE_VALUE(reg, shift, max, invert) }
```

SOC_SINGLE 对寄存器 reg 的位偏移 shift 设置 0-max 的数值。

```
#define SOC_DOUBLE_R_TLV(xname, reg_left, reg_right, xshift, xmax, xinvert, tlv_array) /
{   .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = (xname),/
    .access = SNDRV_CTL_ELEM_ACCESS_TLV_READ |/
         SNDRV_CTL_ELEM_ACCESS_READWRITE,/
    .tlv.p = (tlv_array), /
    .info = snd_soc_info_volsw_2r, /
    .get = snd_soc_get_volsw_2r, .put = snd_soc_put_volsw_2r, /
    .private_value = (unsigned long)&(struct soc_mixer_control) /
        {.reg = reg_left, .rreg = reg_right, .shift = xshift, /
        .max = xmax, .invert = xinvert} }
```

SOC_DOUBLE_R_TLV 对两个寄存器 reg_left 和 reg_right 进行同一操作，对位偏移 xshift 设置 0-xmax 的数值。CODEC 中左右声道寄存器配置一般是类似的，这个宏方便了这种情况。

iface、name、access 之前已介绍了。

invert 是反转的意思，0 表示不反转，1 表示反转，即 value = (invert == 0) ? value : max-value。比如说一个调整音量的 kcontrol，如果 invert=0，设置 max，则此时音量调到了最大；如果 invert=1 时，设置 max，则此时音量调到了最小。

tlv_array 是 kcontrol 的元数据（metadata），用于显示以 dB 为单位的信息，我们可用 DECLARE_TLV_DB_SCALE 来定义包含这种信息的变量，然后把 kcontrol 的 tlv.p 字段指向这些变量。如下：

```
static const DECLARE_TLV_DB_SCALE(adc_tlv, -7200, 75, 1);
```

该宏的第一个参数是变量名字；第二个参数最小值，以 0.01dB 为单位；第三个参数步进值，也是以 0.01dB 为单位；如果 kcontrol 的值设置为最小时会做出 mute 时，则需要将第四个参数设置为 1。

下面以 put 回调函数（snd_soc_put_volsw_2r）为例具体分析一下：

```
/**
 * snd_soc_put_volsw_2r - double mixer set callback
 * @kcontrol: mixer control
 * @ucontrol: control element information
 *
 * Callback to set the value of a double mixer control that spans 2 registers.
 *
 * Returns 0 for success.
 */
int snd_soc_put_volsw_2r(struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_value *ucontrol)
{
    //这里请看 soc_mixer_control、snd_kcontrol 结构体的定义，配合 control.c 稍作分析
    //即可理解，不会太复杂
    struct soc_mixer_control *mc =
        (struct soc_mixer_control *)kcontrol->private_value;
    struct snd_soc_codec *codec = snd_kcontrol_chip(kcontrol);
    //取得 reg、reg2、shift、max、invert，对应于 SOC_DOUBLE_R_TLV 中的参数
    unsigned int reg = mc->reg;
    unsigned int reg2 = mc->rreg;
    unsigned int shift = mc->shift;
    int max = mc->max;
    unsigned int mask = (1 << fls(max)) - 1;
    unsigned int invert = mc->invert;
    int err;
    unsigned int val, val2, val_mask;

    //ucontrol 保存应用层传递进来的参数，ucontrol->value.integer.value[]是要设置的值
    val_mask = mask << shift;
    val = (ucontrol->value.integer.value[0] & mask);
    val2 = (ucontrol->value.integer.value[1] & mask);

    //根据 invert 参数判断是否反转
    if (invert) {
        val = max - val;
        val2 = max - val2;
    }
    val = val << shift;
    val2 = val2 << shift;

    //调用 snd_soc_update_bits 更新寄存器的值
    err = snd_soc_update_bits(codec, reg, val_mask, val);
    if (err < 0)
        return err;

    err = snd_soc_update_bits(codec, reg2, val_mask, val2);
    return err;
}
```

3.3 kcontrol 触发过程

snd_soc_put_volsw_2r()作为一个 callback 函数，用户层要设置某些功能时，如调整回放音量：

```
amixer cset numid=3,iface=MIXER,name='Playback Volume' 100
```

到内核层时，会遍历一个节点类型为 struct snd_kcontrol *的链表，找到 numid 与 3 相匹配的 kctl（这个过程见 snd_ctl_find_id 函数），然后调用 kctl.put()函数将 100 写到 Playback Volume 寄存器中。当然如果上层没有提供 numid，则可根据 name 找到与'Playback Volume'相匹配的 kctl。大致流程如下：

```
alsa_amixer-用户态
  |->snd_ctl_ioctl-系统调用
       |->snd_ctl_elem_write_user-内核钩子函数
            |->snd_ctl_elem_wirte-
                 |->snd_ctl_find_id-遍历 kcontrol 链表找到 name 字段匹配的 kctl
                 |->kctl->put()-调用 kctl 的成员函数 put()
                      |->snd_soc_put_volsw_2r
```

kcontrol 分析就到这里，建立过程请自行参考 control.c，实际上并不复杂。主要要了解 kcontrol 宏的写法及其触发过程，对之后理解 dapm kcontrol 有较大的参考作用。

4 DAPM 之一：概述

dapm 是音频系统颇难理解的一块，然而又非常重要，建立音频路径和最小化便携设备的音频能耗都依赖它了。令人诧异的是，几乎找不到这方面的资料，即使 alsa 官网也只是一个简要介绍，甚至没有接口说明，更谈不上可媲美《write an alsa driver》的文档了。因此我花了点时间把这块内容消化下来，形成这份文档，希望能对后来者有所帮助。

dapm的简单描述：动态音频电源管理（DAPM）用来使得任何时候便携Linux设备都最小化音频子系统的功耗，而且它独立于其它内核电源管理，容易与其他电源管理系统模块共存。dapm的切换根据设备内的音频流活动（捕获/回放）和混音器设置来决定的。

理解以上内容非常重要，dapm的目的是任何时候都最小化音频系统的功耗，触发依据是音频流事件或混音器设置。在篇章“dapm机制深入分析”中，基本就围绕这两点来分析的。

如下是内核文档 dapm.txt，精简介绍了 dapm 概念、设计目的、如何声明一个 dapm widget、dapm widgets 的类型和域、如何建立一个音频路径、endpoint 的概念、dapm 事件回调函数等。这对我们有很好的指导意义，实际上我的分析只是对它细节上的补全。

```
Dynamic Audio Power Management for Portable Devices
===================================================


1. Description
==============


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.


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 spans the whole machine. It covers power control within the entire
audio subsystem, this includes internal codec power blocks and machine
level power systems.


There are 4 power domains within DAPM


   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.

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

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

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

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 Supply     - Power or clock supply widget used by other widgets.
 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.

Most widgets have a name, register, shift and invert. Some widgets have extra parameters for stream name and kcontrols.

2.1 Stream Domain Widgets

Stream Widgets relate to the stream power domain and only consist of ADCs (analog to digital converters) and DACs (digital to analog converters).

Stream widgets have the following format:-

```
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.

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

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:-

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

Any widget kcontrols can be set using the controls and num_controls members.

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.

2.3 Platform/Machine domain Widgets

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.

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

A machine widget can have an optional call back.

e.g. Jack connector widget for an external Mic that enables Mic Bias when the Mic is inserted:-

```
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

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.

2.5 Virtual Widgets

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.

```
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.

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().

3. Codec Widget Interconnections

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.

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.

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.

```
    /* 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.

Interconnections are created with a call to:-

```
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.

3.1 Machine Widget Interconnections

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 */
```

5 DAPM 之二： dapm kcontrol 与 audio paths

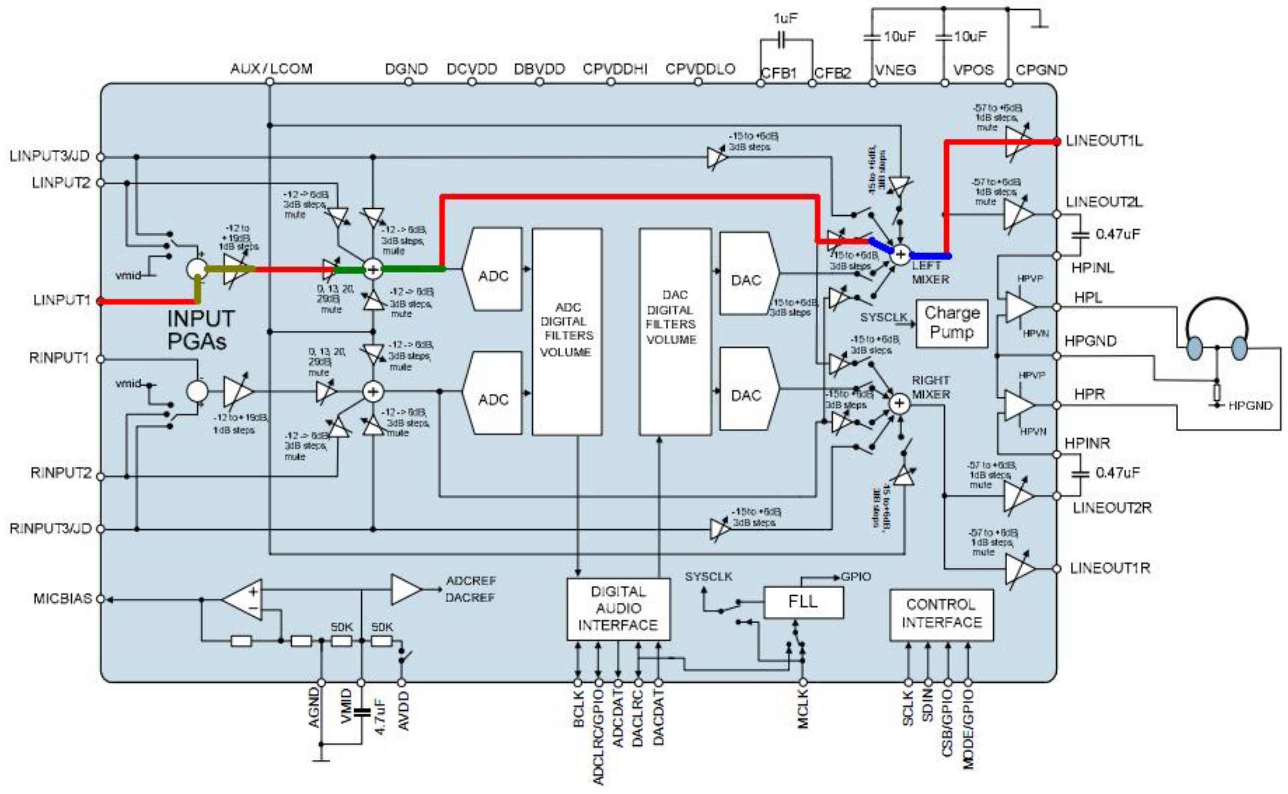
在用 alsa_amixer controls 时，除了我们之前提到的 snd_soc_add_controls 添加的 kcontrols 外，还有一些多出来的 controls。其实多出来的那些都是 dapm kcontrol，主要用于切换音频路径。

5.1 AUDIO PATHS OVERVIEW

以标准内核 2.6.32 的 wm8900 codec 为例。先看 AUDIO PATHS OVERVIEW，红色线路是“LINPUT1->LEFT INPUT PGA->LEFT INPUT MIXER->LEFT OUTPUT MIXER->LINEOUT1L”，表示从 LINPUT 输入的信号通过这条路径送到 LINEOUT 输出，即是录音信号直接放到 SPK 播出。在这条路径上，有三个带+号的圆圈，那就是多路混合器 mixer，用于切换输入源或将多个输入源混合输出。

土黄部分为 LEFT INPUT PGA　　：可选择 LINPUT1、LINPUT2 或 LINPUT3
绿色部分是 LEFT INPUT MIXER ：可选择 INPUTPGA、LINPUT2、LINPUT3 或 AUX/LCOM
蓝色部分是 LEFT OUTPUT MIXER ：可选择 INPUTMIXER、LINPUT3、AUX/LCOM 或 LEFT DAC

配置声音通路时，主要是对 mixer 做切换输入源操作。如要实现 Playback，则需要打通“DAC->OUTPUT MIXER->LINEOUT”通路。



5.2 配置音频通道

这里先绕开代码，先用 alsa_amixer 切换声音路径（以上图的红色路径为例），有个直观印象。

```
~ # alsa_amixer controls
numid=1,iface=MIXER,name='Mic Bias Level'
......省略......
numid=68,iface=MIXER,name='Left Input Mixer AUX Switch'
numid=69,iface=MIXER,name='Left Input Mixer Input PGA Switch'
numid=66,iface=MIXER,name='Left Input Mixer LINPUT2 Switch'
numid=67,iface=MIXER,name='Left Input Mixer LINPUT3 Switch'
numid=73,iface=MIXER,name='Left Input PGA LINPUT1 Switch'
numid=74,iface=MIXER,name='Left Input PGA LINPUT2 Switch'
numid=75,iface=MIXER,name='Left Input PGA LINPUT3 Switch'
numid=3,iface=MIXER,name='Left Input PGA Switch'
numid=2,iface=MIXER,name='Left Input PGA Volume'
numid=4,iface=MIXER,name='Left Input PGA ZC Switch'
numid=57,iface=MIXER,name='Left Output Mixer AUX Bypass Switch'
numid=60,iface=MIXER,name='Left Output Mixer DACL Switch'
numid=56,iface=MIXER,name='Left Output Mixer LINPUT3 Bypass Switch'
numid=58,iface=MIXER,name='Left Output Mixer Left Input Mixer Switch'
numid=59,iface=MIXER,name='Left Output Mixer Right Input Mixer Switch'
......省略......
~ #
```

以上打印出所有kcontrols，以之前对应的颜色区分了 INPUT PGA、INPUT MIXER、OUTPUT MIXER 这三个 mixer 的路径选择。要打开红色通路，则进行如下操作：

```
alsa_amixer cset numid=3,iface=MIXER,name='Left Input PGA Switch' 1
alsa_amixer cset numid=73,iface=MIXER,name='Left Input PGA LINPUT1 Switch' 1
alsa_amixer cset numid=69,iface=MIXER,name='Left Input Mixer Input PGA Switch' 1
alsa_amixer cset numid=58,iface=MIXER,name='Left Output Mixer Left Input Mixer Switch' 1
alsa_amixer cset numid=43,iface=MIXER,name='LINEOUT1 Switch' 1
```

令 Left Input PGA 选择 LINPUT1 作为输入，Left Input Mixer 选择 Input PGA 作为输入，Left Output Mixer 选择 Left Input Mixer 作为输入，这样整条路径就通了。

注：这里仅以最基本的alsa-util来配置回路，在实际应用中，可自己实现alsa mixer或编写asound.conf虚拟出不同的devices。前者较典型见Android2.2 的ALSAControl.cpp，后者在之后的章节会提一下。我偏向于asound.conf实现，灵活性较高，不同的codec改动asound.conf就行了。

5.3 AUDIO PATHS实现

知道声音通路如何配置后，回到驱动代码实现。音频路径功能与普通的 kcontrol 差不多，但写法稍微复杂一点，以路径'**Left Output Mixer Left Input Mixer Switch**'为例说明，具体寄存器配置请参考 wm8900 数据手册。

5.3.1 定义controls

```
static const struct snd_kcontrol_new wm8900_loutmix_controls[] = {
SOC_DAPM_SINGLE("LINPUT3 Bypass Switch", WM8900_REG_LOUTMIXCTL1, 7, 1, 0),
SOC_DAPM_SINGLE("AUX Bypass Switch", WM8900_REG_AUXOUT_CTL, 7, 1, 0),
SOC_DAPM_SINGLE("Left Input Mixer Switch", WM8900_REG_BYPASS1, 7, 1, 0),
SOC_DAPM_SINGLE("Right Input Mixer Switch", WM8900_REG_BYPASS2, 3, 1, 0),
SOC_DAPM_SINGLE("DACL Switch", WM8900_REG_LOUTMIXCTL1, 8, 1, 0),
};
```

5.3.2 定义 dapm widget

```
static const struct snd_soc_dapm_widget wm8900_dapm_widgets[] = {
/* Externally visible pins */
......省略......
/* Input */
......省略......
SND_SOC_DAPM_MIXER("Left Input Mixer", WM8900_REG_POWER2, 5, 0,
           wm8900_linmix_controls,
           ARRAY_SIZE(wm8900_linmix_controls)),
......省略......
/* Output */
......省略......
SND_SOC_DAPM_MIXER("Left Output Mixer", WM8900_REG_POWER3, 3, 0,
           wm8900_loutmix_controls,
           ARRAY_SIZE(wm8900_loutmix_controls)),
......省略......
};
```

注：dapm widget 是分类型的，不同的类型的 widget 用不同的 dapm 宏定义，如 Mixer 类型用 SND_SOC_DAPM_MIXER，PGA 类型用 SND_SOC_DAPM_PGA 等等。关于 widget 类型解释，回头翻 dapm.txt：

```
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
```

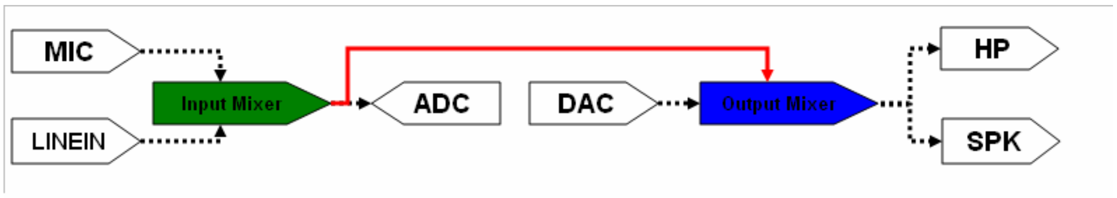
```
o Switch     - An analog switch
o Input      - A codec input pin
o Output     - A codec output pin
o Headphone  - Headphone (and optional Jack)
o Mic        - Mic (and optional Jack)
o Line       - Line Input/Output (and optional Jack)
o Speaker    - Speaker
o Supply     - Power or clock supply widget used by other widgets.
o Pre        - Special PRE widget (exec before all others)
o Post       - Special POST widget (exec after all others)
```

Mixer ： 允许多个输入源混合成一个输出
Mux　 ： 多路选择器，多个输入，但只能选择一路作为输出
PGA　 ： 单路输入，单路输出，带增益（gain）调整的部件
以上三个部件在本章节中非常重要。

5.3.3 搭建音频路由表

5.3.3.1 path 的概念

以上都是介绍部件（widget），这里说一下路径（也称路由，path）的概念：



如图，绿色为 **Input Mixer** 部件，蓝色为 **Output Mixer** 部件，红色线路将 Input Mixer 和 Output Mixer 串联起来，这就是一条路径。这条路径的源是 Input Mixer，目的是 Output Mixer，红色路线是选择行为 control（选择打通或关闭，通常通过目的部件的寄存器来设置）。

dapm.txt 也有一段是描述 path 的，摘录如下：

```
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.
     /* 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.
```

5.3.3.2 snd_soc_dapm_route

在 alsa soc driver 中，路径是用 snd_soc_dapm_route 结构体定义的。

```
/*
 * DAPM audio route definition.
 *
 * Defines an audio route originating at source via control and finishing
 * at sink.
 */
struct snd_soc_dapm_route {
    const char *sink;
    const char *control;
    const char *source;
};
```

注意该结构体的注释，sink 是目的部件，source 是源部件，control 是目的部件定义的 kcontrol；通过 control 可以选择 source 作为 sink 的输入源。

5.3.3.3 路径定义

之前的两步，我们已定义了 Left Input Mixer 和 Left Output Mixer 这两个 widget，并且 Left Output Mixer 的 kcontrols 也定义好了，因此我们可搭建上图中的红色路径：

```
static const struct snd_soc_dapm_route audio_map[] = {
/* Inputs */
......省略......
/* Outputs */
......省略......
{"Left Output Mixer", "Left Input Mixer Switch", "Left Input Mixer"},
......省略......
};
```

5.3.3.4 dapm kcontrol name

在“配置音频通道”章节中，用 alsa_amixer controls 显示所有 controls 时，其中有：

```
numid=58,iface=MIXER,name='Left Output Mixer Left Input Mixer Switch'
```

这个就是一个 dapm kcontrol，用于选择上节中的那条红色线路。用命令：

```
alsa_amixer cset name='Left Output Mixer Left Input Mixer Switch' 1
```

就可以打通 Left Output Mixer 和 Left Input Mixer 之间的通道。

我想大家都已经留意这个 dapm control 的名称标识了吧。

name='Left Output Mixer Left Input Mixer Switch'

不错，对于这个 dapm kcontrol 来说，dapm_kcontrol_name = sink_name + control_name，忽略了 source name。这个过程会在下个章节详细分析。

5.3.4 将 controls、widgets、route 串联起来

```
static int wm8900_add_widgets(struct snd_soc_codec *codec)
{
    snd_soc_dapm_new_controls(codec, wm8900_dapm_widgets,
                  ARRAY_SIZE(wm8900_dapm_widgets));
    snd_soc_dapm_add_routes(codec, audio_map, ARRAY_SIZE(audio_map));
    snd_soc_dapm_new_widgets(codec);
    return 0;
}
```

小结：要建立一条音频路径，首先定义目的部件的 controls、目的部件的 dapm widget、源部件的 dapm widget，再通过 snd_soc_dapm_route 定义通道，最后注册到 alsa soc core 中，形成相应的 dapm kcontrol，这样上层就可以控制 dapm control 来切换音频路径了。可见搭建音频路径并不难，这里有几个地方是非常重要的：path 的概念（包括目的部件-sink、源部件-source、选择控制-control）、dapm kcontrol 的概念、dapm kcontrol name。下章节分析 dapm kcontrol 的建立过程，知其然又知其所以然嘛。

5.4 dapm kcontrol 建立过程

5.4.1 接口函数

在“5.3.4 将 controls、widgets、route 串联起来”小节中，我们看到有三个接口函数，dapm kcontrol 的建立可从这三个函数下手分析。

5.4.1.1 snd_soc_dapm_new_controls

为每个 widget 进行内存分配工作，初始化 snd_soc_dapm_widget 中的链表，并把 widget->list 挂到 codec->dapm_widgets 链表上(方便 dapm 机制管理,在之后章节会分析,这里留意 codec->dapm_widgets 会记录每一个 widget 就行了)。比较简单，按下不表。

5.4.1.2 snd_soc_dapm_add_routes

该函数的注释值得一看：

```
* Connects 2 dapm widgets together via a named audio path. The sink is
* the widget receiving the audio signal, whilst the source is the sender
* of the audio signal.
```

以 Mixer 类型的 widget 为例简要流程：

```
snd_soc_dapm_add_routes
  -->snd_soc_dapm_add_route [添加一个 Mixer 类型的 route]
       -->分配 snd_soc_dapm_path 内存，初始化这个 path，
       -->令 path->sink=sink，path->source=source，记录着 path 的目的部件和源部件
       -->dapm_connect_mixer
            -->检查操作行为 control 是否存在：从 sources 中找到 name 匹配的 control
            -->path->name = control.name;
            -->将 path->list_source 记录到 sink 的链表中，path->list_sink 记录到 source 的链表中
            -->dapm_set_path_status
                 -->判定指定 source 是否被选择了，是则置 path 为连接状态，否则置 path 为 NC 状态
```

以上核心是 path 的建立及其链表的插入操作，如果在目的部件 sink->kcontrols 找不到 name 相匹配的 control，则这个 route 无效，不创建这个 path；接着会设置 path 的连接状态，依据是是否选择了 path->source 作为 path->sink 的输入源。【有点拗口，是不？其实结合上个章节的“Input Mixer->Output Mixer”这条路径来说，就是检查 Output Mixer 是否选择 Input Mixer 作为输入源而已。】

简要流程中的粗体部分对于 dapm 机制非常关键，会有专门的章节分析 path 的链表，这里不要太关心它。

注：其实 dapm_set_path_status 挺困惑的，path 的 connect 理应包含两个操作：1 是指定

输入源 source 的选择，2 是 sink 本身的使能（即 power up）。但这里 connect 状态仅仅是检查 source 是否选择而已，不检查指定 sink 是否使能，看起来不太合理。请保留这个疑问到 dapm 机制分析。

5.4.1.3 snd_soc_dapm_new_widgets

Path 的大部分初始化都在 snd_soc_dapm_add_routes 中做了，包括 path 的目的部件、源部件、几个链表。但是还有两个最关键的放在 snd_soc_dapm_new_widgets 处理，这两个东西分别是 path 的 long_name 和 path 的 kcontrol，其中 dapm kcontrol name 就是 path 的 long_name。

简要流程：

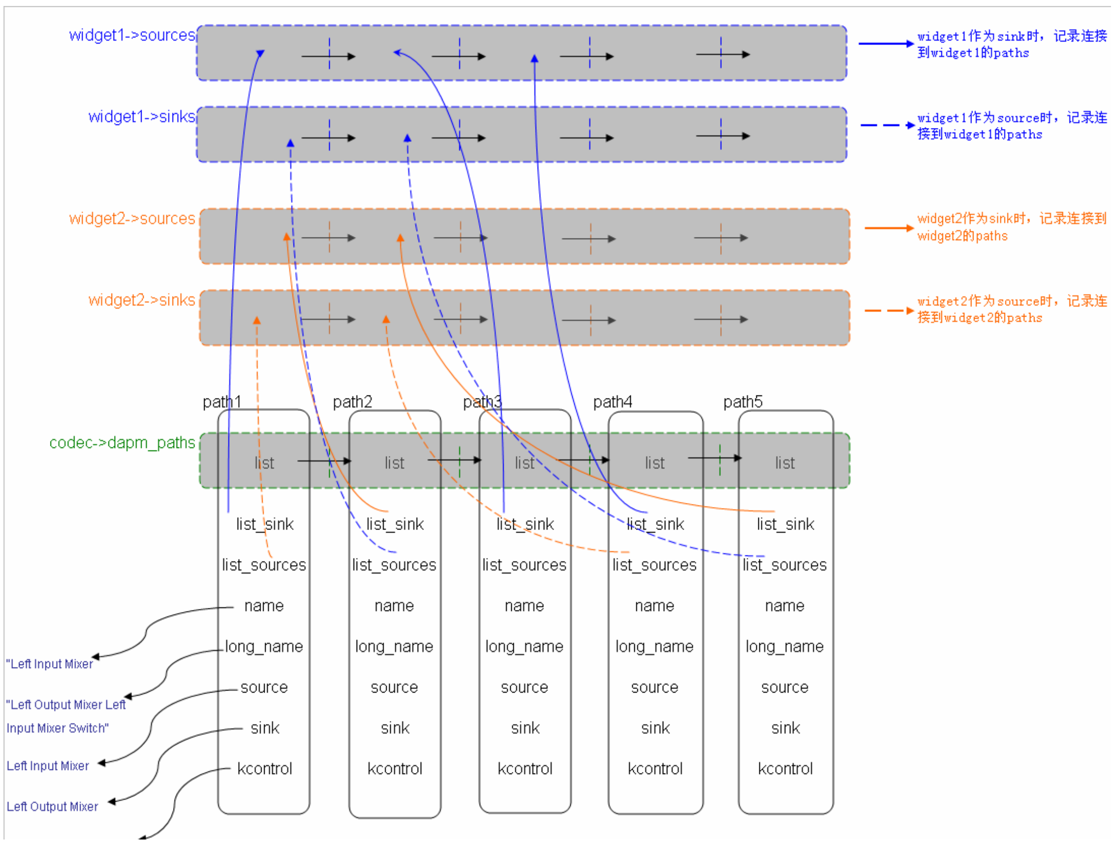
```
snd_soc_dapm_new_widgets
  -->遍历 dapm_widgets 链表，找到为 switch、mixer 类型的 widget
       -->w->power_check = dapm_generic_check_power;
       -->dapm_new_mixer
            -->/* add dapm control with long name.
                * for dapm_mixer this is the concatenation of the
                * mixer and kcontrol name.
                * for dapm_mixer_named_ctl this is simply the
                * kcontrol name.
                */
               snd_soc_dapm_mixer           : path->long_name = sink->name + control->name
               snd_soc_dapm_mixer_named_ctl : path->long_name = control->name
            -->snd_soc_cnew 为 path 分配一个 kcontrol，置这个 kcontrol 的 name 为 path->long_name
            -->snd_ctl_add 注册这个 dapm kcontrol 到 soc core 中
            -->dapm_power_widgets
```

```
path->kcontrol = snd_soc_cnew(&w->kcontrols[i], w, path->long_name);
```
可见 path->kcontrol 对应 sink->kcontrol[i]，但名称标识为 path->long_name。因此上层可以通过 path->long_name 找到对应的 sink->kcontrol[i]。

注：根据 widget 的类型，path 的 long_name（即 dapm kcontrol name）也会有所不同。简要流程中粗体部分的 dapm_power_widget 暂时不用关心。

5.4.2 path 剖析

以上过程都是围绕 path 来展开的，搞懂 path 的成员数据，就等于明白了对应的 dapm kcontrol 的建立过程了。同时，path 对于 dapm 机制也是非常非常重要的，有一个关键的工作是围绕 path 的链表展开的，相信你们以后看 dapm 机制时还会回过头复习这部分内容。因此，对 path 的分析是重中之重。


widget1->sources
widget1->sinks
widget2->sources
widget2->sinks
widget1作为sink时，记录连接到widget1的paths
widget1作为source时，记录连接到widget1的paths
widget2作为sink时，记录连接到widget2的paths
widget2作为source时，记录连接到widget2的paths
path1
path2
path3
path4
path5
codec->dapm_paths
list
list_sink
list_sources
name
long_name
source
sink
kcontrol
"Left Input Mixer
"Left Output Mixer Left Input Mixer Switch"
Left Input Mixer
Left Output Mixer

在“5.4.1.2 snd_dapm_add_route”小节中提到“将 path->list_source 记录到 sink 的链表中，path->list_sink 记录到 source 的链表中”，其代码如下：

```
list_add(&path->list, &codec->dapm_paths);
list_add(&path->list_sink, &dest->sources);
list_add(&path->list_source, &src->sinks);
```

其中 dest 是 path 的目的部件 sink，src 是 path 的源部件 source。

为什么要这样做？因为这很灵活。

codec->dapm_paths 记录着每一个 path，path 记录着 sink 和 source，而每一个 widgets 都记录着连接到它本身的 paths（widget 本身作为 path 的 sink 或 source）。

因此，找到一个 path ->就可以找到 path 对应的 sink 和 soure ->接着可以找到连接到 sink/source 上的其他 paths ->依次类推。

5.5 dapm kcontrol 触发过程

dapm kcontrol 的触发很大程度上可以参考“3.3 kcontrol 触发过程”，但更为复杂。当上层触发 dapm kcontrol 时，会做两个重要动作：1 是切换音频通路，这与普通的 kcontrol 做法基本一致；2 是使能 dapm widget(power up/down)，这就是分歧之处。其实这里已经解决了“5.4.1.2 snd_soc_dapm_add routes”中的有关 dapm_set_path_statu 的疑问，因为上层触发 dapm kcontrol 的时候，首先对 widget 进行 power up/down 操作，这样没必要再去检查 widget 是否使能了。

普通的 kcontrol 宏定义：

```
#define SOC_SINGLE(xname, reg, shift, max, invert) /
{   .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, /
    .info = snd_soc_info_volsw, .get = snd_soc_get_volsw,/
    .put = snd_soc_put_volsw, /
    .private_value =  SOC_SINGLE_VALUE(reg, shift, max, invert) }
```

dapm kcontrol 宏定义：

```
/* dapm kcontrol types */
#define SOC_DAPM_SINGLE(xname, reg, shift, max, invert) /
{   .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, /
    .info = snd_soc_info_volsw, /
    .get = snd_soc_dapm_get_volsw, .put = snd_soc_dapm_put_volsw, /
    .private_value =  SOC_SINGLE_VALUE(reg, shift, max, invert) }
```

以 put 回调函数为例说明：

```
snd_soc_dapm_put_volsw
  -->dapm_mixer_update_power
       -->snd_kcontrol_chip
            -->找到 dapm kcontrol 所在的 widget
       -->snd_soc_test_bits
            -->Tests a register with a new value and checks if the new
                 value is different from the old value.
       -->dapm_power_widgets
            -->power up/down 部件，更底层可追溯到 dapm_seq_run_coalesced
  -->如果有 event 回调函数，则执行 event
  -->snd_soc_update_bits
       -->控制 SOC_DAPM_SINGLE 定义的 dapm kcontrol，方法与普通的 kcontrol 一样
```

粗体部分属于 dapm 机制范围，这里有个概念就行了。
这是底层方法流程，往上应该没必要说了，与普通的 kcontrol 触发过程是一样的。

6 DAPM 之三：audio paths 与 asound.conf

其实 asound.conf 真跟 dapm 没多大关系，之所以把它也纳入 dapm 系列之一，是为了考虑到知识的连贯性。在“5.2 配置音频通道”小节中提到：通过配置好 asound.conf，上层则可打开 asound.conf 中定义的虚拟设备，而自动选择相应的音频通道。这是 asound.conf 很重要的一个作用，从这方面来说，并不是跟 dapm 完全没关系。

6.1 认识 asound.conf

做 alsa 的基本都能体会到 alsa-lib 的复杂与强大，而 alsa-lib 的强大正是从 asound.conf 与.asoundrc 等配置文件体现出来。alsa 驱动开发只是一个方面，而真正想随心所欲的配置音频设备，asound.conf 与.asoundrc 的掌握是必不可少的。所幸，这方面的资料还是比较丰富，所需了解的知识点基本都能从官网上找到文档甚至 example。

http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html
http://alsa.opensrc.org/.asoundrc

6.2 认识 plugin hooks

首先我们先看看 plugin hooks 的描述及其语法：
http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html#pcm_plugins_hooks

This plugin is used to call some 'hook' function when this plugin is opened, modified or closed. Typically, it is used to change control values for a certain state specially for the PCM (see the example below).

```
# Hook arguments definition
hook_args.NAME {
        ...                     # Arbitrary arguments
}

# PCM hook type
pcm_hook_type.NAME {
        [lib STR]               # Library file (default libasound.so)
        [install STR]           # Install function (default _snd_pcm_hook_NAME_install)
}

# PCM hook definition
pcm_hook.NAME {
        type STR                # PCM Hook type (see pcm_hook_type)
        [args STR]              # Arguments for install function (see hook_args)
        # or
        [args { }]              # Arguments for install function
}

# PCM hook plugin
pcm.NAME {
        type hooks              # PCM with hooks
        slave STR               # Slave name
        # or
        slave {                 # Slave definition
                pcm STR         # Slave PCM name
                # or
                pcm { }         # Slave PCM definition
        }
        hooks {
                ID STR          # Hook name (see pcm_hook)
                # or
                ID { }          # Hook definition (see pcm_hook)
        }
}
```

Example:

```
        hooks.0 {
                type ctl_elems
                hook_args [
                        {
                                name "Wave Surround Playback Volume"
                                preserve true
                                lock true
                                optional true
                                value [ 0 0 ]
                        }
                        {
                                name "EMU10K1 PCM Send Volume"
                                index { @func private_pcm_subdevice }
                                lock true
                                value [ 0 0 0 0 0 0 255 0 0 0 0 255 ]
                        }
                ]
        }
```

Here, the controls "Wave Surround Playback Volume" and "EMU10K1 PCM Send Volume" are set to the given values when this pcm is accessed. Since these controls take multi-dimensional values, the value field is written as an array. When preserve is true, the old values are saved and restored when the pcm is closed. The lock means that the control is locked during this pcm is opened, and cannot be changed by others. When optional is set, no error is returned but ignored even if the specified control doesn't exist.

我们可以定义一个名为 NAME 的 hook plugin，在这个 plugin 中，我们可以操作之前提到的 dapm kcontrol，达到音频通道切换的目的。另外注意：

1、 When preserve is true, the old values are saved and restored when the pcm is closed.
    当 preserve 设置为 true 时，则该 pcm 关闭时，kcontrol 会恢复到之前的值。
2、 The lock means that the control is locked during this pcm is opened, and cannot be changed by others.
    当 lock 设置为 true 时，则在该 pcm 打开期间，kcontrol 的值不会被其他的 pcm 改变。
3、 When optional is set, no error is returned but ignored even if the specified control doesn't exist.
    当 optional 设置为 true 时，则指定的 kcontrol 不存在时不会返回错误。

6.3 配置 audio paths

以“5.1 AUDIO PATHS OVERVIEW”小节中的红色线路为例，在 Android 平台上写一个 linein 录音直送到 SPK 输出的 hooks plugin：

```
pcm.AndroidPlayback_Speaker_normal {
    type hooks
    slave.pcm {
        type hw
        card 0
        device 0
    }
    hooks.0 {
        type ctl_elems
        hook_args [
            {
                name 'Left Input PGA Switch'
                value true
            }
            {
                name 'Left Input PGA LINPUT1 Switch'
                preserve true
                lock true
                value true
            }
            {
                name 'Left Input Mixer Input PGA Switch'
                preserve true
                lock true
                value true
            }
            {
                name 'Left Output Mixer Left Input Mixer Switch'
                preserve true
                lock true
                value true
            }
            {
                name 'LINEOUT1 Switch'
                value true
            }
        ]
    }
}
```

把这个 asound.conf 放到/etc 目录下，再启动 Android 做这个测试，应该可以听到 Linein 输入的录音信号直接在 SPK 上输出。

Android 三种声音模式：normal、ringtone 和 incall，这些模式的音频通道切换可以通过配置 asound.conf 来实现。当然也可用 ALSACcontrol 类来操作，问题是不太方便。

7 DAPM 之四：dapm 机制深入分析（上）

OK，终于进入本文主角 dapm 机制篇章了。在分析之前，我们先复习一下 dapm 的设计意图：动态音频电源管理（DAPM）用来使得任何时候便携Linux设备都最小化音频子系统的功耗，而且它独立于其它内核电源管理，容易与其他电源管理系统模块共存。dapm的切换根据设备内的音频流活动（捕获/回放）和混音器设置来决定的。

有几个地方值得我们注意的：

1、 独立于其他内核模块电源管理。这比较容易理解，整个模块就集中在 dapm.c 文件内，与 Linux 驱动模型没有交集。
2、 最小化音频子系统的能耗。怎样才能做到最小化？自然是只打开必要的部件（widget）。例如在一个时刻内，只播放回放子流（playback substream），则只需打开“DAC->PGA->outmixer->spk/hp”通道，其他部件则关闭。
3、 消除部件通电/断电产生的 pops。这个在 dapm 描述中并没有提到，但有点音频基础知识的都知道，音频部件通电/断电瞬间会产生或大或小的 pop 音。那么 dapm 机制是如何处理的？
4、 dapm 切换（我习惯称为触发）依据。一是音频流活动，二是混音器设置。

准备将第 2、3 点即 dapm 机制处理细节放在本篇分析，第 4 点 dapm 触发以及大致处理流程放到下篇来说。

7.1 最小化音频子系统的能耗

7.1.1 endpoint 的概念

```
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.

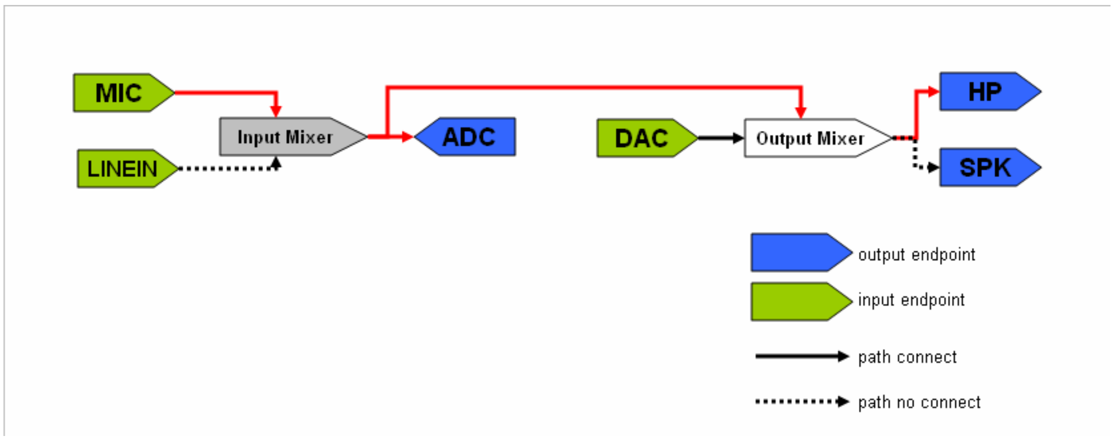
This also applies to machine widgets. e.g. if a headphone is connected to a jack then the jack can be marked active. If the headphone is removed, then the headphone jack can be marked inactive.

endpoint 是完整通道（complete path，稍后解释）的端点，分为 input endpoint 和 output endpoint。
input endpoint 指声音（模拟的或数字的）的输入点，如 MIC、LINEIN、DAC 这些部件属于 input endpoint。
output endpoint 指声音（模拟的或数字的）的输出点，如 HP、SPK、ADC 这些部件属于 output endpoint。

7.1.2 complete path 的概念

上面提到“怎样才能做到最小化？自然是只打开必要的部件”，实际上如何操作的呢？很简单，遍历所有的 widgets，判断 widget 是否处在一个完整通道（complete path）中，是则 power up，否则 power down。

complete path 意思是在 input endpoint 和 output endpoint 之间的所有 paths 是连接的。因此，complete path 是由数条相互连接的 paths 构成的，始于 input endpoint，终于 output endpoint。



上图有三个 complete path，分别是“MIC->Input Mixer->ADC”、“MIC->Input Mixer->Output Mixer->HP”、“DAC->Output Mixer->HP”。

7.1.3 path 的连接状态

连接状态是根据 path->connect 来判断的，改变 mixer/mux 部件的 path->connect 有两个地

方：

1、 在 dapm kcontrols 建立时，会检查 mixer/mux 部件的连接状态，具体见 dapm_set_path_status：判定是否选择 path 的 source 作为 path 的 sink 的输入源，是则置 p->connect = 1，否则置 p->connect = 0。
2、 上层通过 alsa_amixer 或其他来操作 dapm kcontrols 时，会更新路径的连接状态 path->connect，简略流程如下：

```
amixer-应用层
  |->snd_ctl_ioctl-系统调用
       |->snd_ctl_elem_write_user-内核钩子函数
            |->snd_ctl_elem_wirte-
                 |->snd_ctl_find_id-遍历 kcontrol 链表找到 name 字段匹配的 kctl
                 |->kctl->put()-调用 kctl 的成员函数 put()
                      |->snd_soc_dapm_put_volsw
                           |->dapm_mixer_update_power
                                |->更新 path->connect
```

注：pga、adc、dac、input、output 等部件初始化 path->connect=1，hp、spk、mic 等部件初始化 path->connect=0，具体见 snd_soc_dapm_add_route 函数。

注意：在 dapm_mixer_update_power 中，先更新 path->connect 状态，然后才试图使能 widget 的。为啥特意说明这一点，因为接下来马上就要检查 path 的连接状态了。

7.1.4 is_connected_output_ep

is_connected_output_ep 挺有趣的，一路往前走，直至达到 output endpoint，检查 widget 是否真能到达 output endpoint。同理 is_connected_input_ep 往回走，检查是否能到达 input endpoint。目的是检查路径的完整性，保证 widget 是贯穿 input endpoint 和 output endpoint 的。

complete path 图中浅绿色为 input endpoint，浅蓝色为 output endpoint。如红色线路对于 Input Mixer，它如何找到 output endpoint 呢？其实这是一个递归过程：

1、 首先会找到 Input Mixer->sinks，发现有两个分别是 ADC 和 Output Mixer；
2、 检查“Input Mixer->ADC”是否连接，接着检查到 ADC 是属于 snd_soc_dapm_adc 类型的 widget，返回 1，因为这个 widget 属于 output endpoint 类型并且是连接上的；
3、 检查“Input Mixer->Output Mixer”是否连接的，如果不是则跳出；如果是则继续往下走，由于这个 widget 不属于 output endpoint，继续找 Output Mixer->sinks，有两个分别是 HP 和 SPK；
4、 再检查到 HP 连接到 Output Mixer，而且 HP 也属于 output endpoint，因此“Input Mixer->Output Mixer->HP”这个通道也是贯穿 output endpoint 的。

以上是 is_connected_output_ep 的过程，至于 is_connected_input_ep，则遍历的方向是相反的（即 widget->sources），往前遍历到 MIC（input endpoint）。当然已走过的 path 会置 walked 标识，不会重复检查。

只有 is_connected_output_ep 和 is_connected_input_ep 同时不为 0（即往后能到 input endpoint，往前能到 output endpoint），才认为 widget 位于一条完整合法的音频通道中。

完整代码如下，写得很巧妙：

```
/*
 * Recursively check for a completed path to an active or physically connected
 * output widget. Returns number of complete paths.
 */
static int is_connected_output_ep(struct snd_soc_dapm_widget *widget)
{
    struct snd_soc_dapm_path *path;
    int con = 0;

    if (widget->id == snd_soc_dapm_supply)
        return 0;

    //如果 widget 类型是 adc 或 aif_out，则到达 endpoint，找到一条完整通道。
    switch (widget->id) {
    case snd_soc_dapm_adc:
    case snd_soc_dapm_aif_out:
        if (widget->active)
            return 1;
    default:
        break;
    }

    //如果 widget 类型是 output、hp、spk 等，则到达 endpoint，找到一条完整通道。
    if (widget->connected) {
        /* connected pin ? */
        if (widget->id == snd_soc_dapm_output && !widget->ext)
            return 1;

        /* connected jack or spk ? */
        if (widget->id == snd_soc_dapm_hp || widget->id == snd_soc_dapm_spk ||
            (widget->id == snd_soc_dapm_line && !list_empty(&widget->sources)))
            return 1;
    }

    //这里如果觉得很难理解，请回到“5.4.2 path 剖析”看下 path 的链表操作。
    //这里递归遍历 widget->sink，直至到达一个 endpoint 或者遍历完毕，过程中检查 path 连接状态。
    //is_connected_output_ep 是将 widget 作为 source 往后递归找到 output endpoint;
    //is_connected_input_ep 是将 widget 作为 sink 往前递归找到 input endpoint。
    list_for_each_entry(path, &widget->sinks, list_source) {
        if (path->walked)
            continue; //这条路径已走过，找下一个 path
        if (path->sink && path->connect) {
            path->walked = 1;
            con += is_connected_output_ep(path->sink);
        }
    }

    return con;
}
```

7.1.5 dapm_generic_check_power

```
/* Generic check to see if a widget should be powered.
 */
static int dapm_generic_check_power(struct snd_soc_dapm_widget *w)
{
    int in, out;

    in = is_connected_input_ep(w);
    dapm_clear_walk(w->codec);
    out = is_connected_output_ep(w);
    dapm_clear_walk(w->codec);
    return out != 0 && in != 0;
}
```

检查一个 widget 是否需要打开，调用 is_connected_input_ep 检查该 widget 是否连接到 input endpoint，is_connected_output_ep 检查该 widget 是否连接到 output endpoint。一个 widget 只有同时连接到 input endpoint 和 output endpoint，才认为这个 widget 需要打开。

这其实是一系列的函数，分别是
dapm_generic_check_power、
dapm_adc_check_power、
dapm_dac_check_power、
dapm_supply_check_power，
视 widget 的类型使用。如：
mixer/mux 的 widget   ：w->power_check = dapm_generic_check_power;
adc/aif_out 的 widget ：w->power_check = dapm_adc_check_power
具体见 snd_soc_dapm_new_widgets。

小结：这小章节阐述了如何保证最小化音频系统能耗，细节处理很巧妙，同时有些地方较难理解，需要掌握 endpoint、complete path、path 连接状态、path 链表作用等知识点。

7.2 depop

7.2.1 pops clicks

如下摘录内核文档 pops_clicks.txt，解释 pops 和 clicks 是如何产生的，如何尽量消除（一般认为不能完全消除 pops）。请留意其中的通电/断电次序及 Zipper Noise 的内容，dapm 机制就是根据这两点来处理的。

```
Audio Pops and Clicks
=====================


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).

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.
```

7.2.2 dapm_set_pga

这个函数会在 dapm 触发时调用，主要是为了减少通电/断电 PGA 产生的 pops。如下详细分析了这个函数：

```c
/* ramps the volume up or down to minimise pops before or after a
 * DAPM power event */
static int dapm_set_pga(struct snd_soc_dapm_widget *widget, int power)
{
    //取得该 widget 的 kcontrol，这个 kcontrol 应该是用于调节 PGA 的音量的，请留意这点
    //对于 PGA widget，一般是单路输入，单路输出，带 gain 调整
    const struct snd_kcontrol_new *k = widget->kcontrols;

    //检查该 widget 是否为 muted，从下面看来，当 widget power up 时，置 muted=0，反之置 1
    if (widget->muted && !power)
        return 0;
    if (!widget->muted && power)
        return 0;

    if (widget->num_kcontrols && k) {
        //以下操作取得 kcontrol 的 reg、shift 和 mask 等，用于调整该 PGA widget 的增益 gain
        struct soc_mixer_control *mc =
            (struct soc_mixer_control *)k->private_value;
        unsigned int reg = mc->reg;
        unsigned int shift = mc->shift;
        int max = mc->max;
        unsigned int mask = (1 << fls(max)) - 1;
        unsigned int invert = mc->invert;

        if (power) {
            int i;
            /* power up has happended, increase volume to last level */
            //power up widget，则将保存值恢复到 widget。注意：这是一个逐步的过程，
            //从 0 至保存值 saved_value 逐步恢复，这是为了减少 pops
            if (invert) {
                for (i = max; i > widget->saved_value; i--)
                    snd_soc_update_bits(widget->codec, reg, mask, i);
            } else {
                for (i = 0; i < widget->saved_value; i++)
                    snd_soc_update_bits(widget->codec, reg, mask, i);
            }
            widget->muted = 0;
        } else {
            /* power down is about to occur, decrease volume to mute */
            //power down widget，保存 widget 当前的 gain 值，并逐步减少该 widget 的音量，直至 0
            int val = snd_soc_read(widget->codec, reg);
            int i = widget->saved_value = (val >> shift) & mask;
            if (invert) {
                for (; i < mask; i++)
                    snd_soc_update_bits(widget->codec, reg, mask, i);
            } else {
                for (; i > 0; i--)
                    snd_soc_update_bits(widget->codec, reg, mask, i);
            }
            widget->muted = 1;
        }
    }
    return 0;
}
```

建议：当遇到 PGA 部件时，最好按照规范来写这个 dapm widget，该 widget 应包含 gain 调整的 kcontrol，这样系统会避免通电/断电时可能带来的 pops。

7.2.3 dapm_seq_insert

这个函数的作用是：将 widgets 按照一定的顺序插入到链表 list 中。在 power up/down 时，为了减少可能由此产生的 pops，必须按照一定的顺序来操作这些 widgets，这个顺序请参考

pops_clicks.txt 提及的。

```
/* Insert a widget in order into a DAPM power sequence. */
static void dapm_seq_insert(struct snd_soc_dapm_widget *new_widget,
                struct list_head *list,
                int sort[])
{
    struct snd_soc_dapm_widget *w;

    list_for_each_entry(w, list, power_list)
        if (dapm_seq_compare(new_widget, w, sort) < 0) {
            list_add_tail(&new_widget->power_list, &w->power_list);
            return;
        }

    list_add_tail(&new_widget->power_list, list);
}
```

这个函数还是蛮容易理解的，不多解释了。

power down 时，顺序如下：

```
static int dapm_down_seq[] = {
    [snd_soc_dapm_pre] = 0,
    [snd_soc_dapm_adc] = 1,
    [snd_soc_dapm_hp] = 2,
    [snd_soc_dapm_spk] = 2,
    [snd_soc_dapm_pga] = 4,
    [snd_soc_dapm_mixer_named_ctl] = 5,
    [snd_soc_dapm_mixer] = 5,
    [snd_soc_dapm_dac] = 6,
    [snd_soc_dapm_mic] = 7,
    [snd_soc_dapm_micbias] = 8,
    [snd_soc_dapm_mux] = 9,
    [snd_soc_dapm_value_mux] = 9,
    [snd_soc_dapm_aif_in] = 10,
    [snd_soc_dapm_aif_out] = 10,
    [snd_soc_dapm_supply] = 11,
    [snd_soc_dapm_post] = 12,
};
```

Power up 时，顺序如下：

```
static int dapm_down_seq[] = {
    [snd_soc_dapm_pre] = 0,
    [snd_soc_dapm_adc] = 1,
    [snd_soc_dapm_hp] = 2,
    [snd_soc_dapm_spk] = 2,
    [snd_soc_dapm_pga] = 4,
    [snd_soc_dapm_mixer_named_ctl] = 5,
    [snd_soc_dapm_mixer] = 5,
    [snd_soc_dapm_dac] = 6,
    [snd_soc_dapm_mic] = 7,
    [snd_soc_dapm_micbias] = 8,
    [snd_soc_dapm_mux] = 9,
    [snd_soc_dapm_value_mux] = 9,
    [snd_soc_dapm_aif_in] = 10,
    [snd_soc_dapm_aif_out] = 10,
    [snd_soc_dapm_supply] = 11,
    [snd_soc_dapm_post] = 12,
};
```

7.2.4 dapm_generic_apply_power

核心函数，power up/down 指定的 widget，dapm 机制绕来绕去最终就是为了控制 widget 的开关。但是原理是非常简单的，复杂的是以上提到的部分，如检查 complete path、depop 等前期处理细节。

注：这里把它称为核心函数，实际上并没有怎么用到它，等到下一章分析就明白了。但是它的处理是非常典型的。

```
/* Standard power change method, used to apply power changes to most
 * widgets.
 */
static int dapm_generic_apply_power(struct snd_soc_dapm_widget *w)
{
    int ret;

    /* call any power change event handlers */
    if (w->event)
        pr_debug("power %s event for %s flags %x\n",
             w->power ? "on" : "off",
             w->name, w->event_flags);

    //在开关 widget 前，先执行标识为 SND_SOC_DAPM_PRE_PMU/SND_SOC_DAPM_PRE_PMD 的 event
    //回调函数分支。event flag 定义见 dapm.h。

    /* power up pre event */
    if (w->power && w->event &&
        (w->event_flags & SND_SOC_DAPM_PRE_PMU)) {
        ret = w->event(w, NULL, SND_SOC_DAPM_PRE_PMU);
        if (ret < 0)
            return ret;
    }

    /* power down pre event */
    if (!w->power && w->event &&
        (w->event_flags & SND_SOC_DAPM_PRE_PMD)) {
        ret = w->event(w, NULL, SND_SOC_DAPM_PRE_PMD);
        if (ret < 0)
            return ret;
    }

    //关闭 widget 前，如果该 widget 属于 PGA 部件，调用 dapm_set_pga 控制 pga 部件音量逐步减少，
    //减少部件断电时产生的 POP 音。
    /* Lower PGA volume to reduce pops */
    if (w->id == snd_soc_dapm_pga && !w->power)
        dapm_set_pga(w, w->power);

    //操作 widget 开启或关闭。
    dapm_update_bits(w);

    //开启 widget 后，如果该 widget 属于 PGA 部件，调用 dapm_set_pga 控制 pga 部件音量逐步增大，
    //减少部件断电时产生的 POP 音。
    /* Raise PGA volume to reduce pops */
    if (w->id == snd_soc_dapm_pga && w->power)
        dapm_set_pga(w, w->power);

    //在开关 widget 后，执行标识为 SND_SOC_DAPM_POST_PMU/SND_SOC_DAPM_POST_PMD 的 event
    //回调函数分支。event flag 定义见 dapm.h。

    /* power up post event */
    if (w->power && w->event &&
        (w->event_flags & SND_SOC_DAPM_POST_PMU)) {
        ret = w->event(w,
                   NULL, SND_SOC_DAPM_POST_PMU);
        if (ret < 0)
            return ret;
    }

    /* power down post event */
    if (!w->power && w->event &&
        (w->event_flags & SND_SOC_DAPM_POST_PMD)) {
        ret = w->event(w, NULL, SND_SOC_DAPM_POST_PMD);
        if (ret < 0)
            return ret;
    }

    return 0;
}
```

8 DAPM 之五：dapm 机制深入分析（下）

dapm 触发时的入口函数是 dapm_power_widgets，稍后详细分析这个函数，这里仅说其作用：检查每个 dapm widget，如果该 widget 处在 complete paths 中，则 power up 这个 widget，否则 power down。

8.1 dapm 触发

1、 dapm widgets 建立时，详见 snd_soc_dapm_new_widgets。

2、 上层通过 alsa_amixer 等工具改变 codec 音频路径时，此时与此相关的 widgets 状态要重置，详见 dapm_mixer_update_power 和 dapm_mux_update_power。

```
amixer-应用层
  |->snd_ctl_ioctl-系统调用
       |->snd_ctl_elem_write_user-内核钩子函数
            |->snd_ctl_elem_wirte-
                 |->snd_ctl_find_id-遍历 kcontrol 链表找到 name 字段匹配的 kctl
                 |->kctl->put()-调用 kctl 的成员函数 put()
                      |->snd_soc_dapm_put_volsw
                           |->dapm_mixer_update_power
                                |->更新 path->connect 状态
                                |->dapm_power_widgets 触发 dapm，重置相关的 widgets
```

3、 发生 stream 事件时，会触发 snd_soc_dapm_stream_even。什么叫 stream 事件？准备或关闭一个 pcm stream 通道（snd_pcm_prepare/snd_pcm_close）这些都属于 stream 事件。另外 suspend 或 resume 时，也会触发 snd_soc_dapm_stream_event 处理。

```
snd_pcm_prepare
  |->soc_pcm_prepare
       |->处理 platform、codec-dai、cpu-dai 的 prepare 回调函数
       |->snd_soc_dapm_stream_event
            |->遍历 codec 每个 dapm widget，如果该 widget 的 stream name 与传递进来的 stream 是否
               相匹配，如果匹配则置 widget->active 为真
            |->dapm_power_widgets 触发 dapm，重置相关的 widgets
```

8.2 dapm_power_widgets

1、 初始化两个链表 up_list 和 down_list，如字面意思，up_list 指向要 power up 的 widgets，down_list

指向要 power down 的 widgets;

2、 遍历所有 widgets,检查是否需要对其进行 power 操作;要 power up 的则插入到 up_list,要 power down 的则插入到 down_list;

3、 先 power down down_list 上的 widgets, 再 power up up_list 上的 widgets;

4、 设置 codec 的偏置（bias）电压。

```
/*
 * Scan each dapm widget for complete audio path.
 * A complete path is a route that has valid endpoints i.e.:-
 *
 *  o DAC to output pin.
 *  o Input Pin to ADC.
 *  o Input pin to Output pin (bypass, sidetone)
 *  o DAC to ADC (loopback).
 */
static int dapm_power_widgets(struct snd_soc_codec *codec, int event)
{
    struct snd_soc_device *socdev = codec->socdev;
    struct snd_soc_dapm_widget *w;

    //初始化两个链表 up_list 和 down_list, up_list 指向要 power up 的 widgets, down_list 指向要
    //power down 的 widgets
    LIST_HEAD(up_list);
    LIST_HEAD(down_list);
    int ret = 0;
    int power;
    int sys_power = 0;

    /* Check which widgets we need to power and store them in
     * lists indicating if they should be powered up or down.
     */
    //遍历所有的 dapm widgets, 检查是否需要对 widget 开关; '开'则把该 widget 插入到 up_list,
    //'关'则插入到 down_list
    list_for_each_entry(w, &codec->dapm_widgets, list) {
        switch (w->id) {
        case snd_soc_dapm_pre:
            //属 machine specific pre widget, 插入到 down_list 最前方
            dapm_seq_insert(w, &down_list, dapm_down_seq);
            break;
        case snd_soc_dapm_post:
            //属 machine specific post widget, 插入到 up_list 最后方
            dapm_seq_insert(w, &up_list, dapm_up_seq);
            break;

        default:
```

```
        //其他类型的 widgets，则调用自身的 power_check 函数进行检查需要开关。
        //关于 power_check，具体见<DAPM 之五：dapm 机制深入分析（上）>，非常重要的一个函数。

        if (!w->power_check)
            continue;

        /* If we're suspending then pull down all the
         * power. */
        switch (event) {
        case SND_SOC_DAPM_STREAM_SUSPEND:
            //上面注释很清楚了，如果是 suspend 事件，则 pull down 所有 widgets。
            power = 0;
            break;

        default:
            power = w->power_check(w);
            if (power)
                sys_power = 1;
            break;
        }

        //w->power 保存 widget 当前的 power 状态，如果当前状态和设置状态一致，那么显然不用
        //重复设置 widget
        if (w->power == power)
            continue;

        //将 widget 插入到 up_list 或 down_list 中
        if (power)
            dapm_seq_insert(w, &up_list, dapm_up_seq);
        else
            dapm_seq_insert(w, &down_list, dapm_down_seq);

        //更新 w->power 的状态
        w->power = power;
        break;
    }
}

/* If there are no DAPM widgets then try to figure out power from the
 * event type.
 */
if (list_empty(&codec->dapm_widgets)) {
    switch (event) {
    case SND_SOC_DAPM_STREAM_START:
```

```
		case SND_SOC_DAPM_STREAM_RESUME:
			sys_power = 1;
			break;
		case SND_SOC_DAPM_STREAM_SUSPEND:
			sys_power = 0;
			break;
		case SND_SOC_DAPM_STREAM_NOP:
			sys_power = codec->bias_level != SND_SOC_BIAS_STANDBY;
			break;
		default:
			break;
		}
	}

	/* If we're changing to all on or all off then prepare */
	if ((sys_power && codec->bias_level == SND_SOC_BIAS_STANDBY) ||
	    (!sys_power && codec->bias_level == SND_SOC_BIAS_ON)) {
		ret = snd_soc_dapm_set_bias_level(socdev,
						  SND_SOC_BIAS_PREPARE);
		if (ret != 0)
			pr_err("Failed to prepare bias: %d\n", ret);
	}

	//先 power down 链表 down_list 上的 widgets，接着 power up 链表 up_list 上的 widgets
	//按照这样的次序，目的是避免产生 pop 音
	//dapm_seq_run 核心函数，见其详细分析

	/* Power down widgets first; try to avoid amplifying pops. */
	dapm_seq_run(codec, &down_list, event, dapm_down_seq);

	/* Now power up. */
	dapm_seq_run(codec, &up_list, event, dapm_up_seq);

	/* If we just powered the last thing off drop to standby bias */
	if (codec->bias_level == SND_SOC_BIAS_PREPARE && !sys_power) {
		ret = snd_soc_dapm_set_bias_level(socdev,
						  SND_SOC_BIAS_STANDBY);
		if (ret != 0)
			pr_err("Failed to apply standby bias: %d\n", ret);
	}

	/* If we just powered up then move to active bias */
	if (codec->bias_level == SND_SOC_BIAS_PREPARE && sys_power) {
		ret = snd_soc_dapm_set_bias_level(socdev,
                        SND_SOC_BIAS_ON);
        if (ret != 0)
            pr_err("Failed to apply active bias: %d\n", ret);
    }

    pop_dbg(codec->pop_time, "DAPM sequencing finished, waiting %dms\n",
        codec->pop_time);

    return 0;
}
```

8.3 dapm_seq_run

```
/* Apply a DAPM power sequence.
 *
 * We walk over a pre-sorted list of widgets to apply power to.  In
 * order to minimise the number of writes to the device required
 * multiple widgets will be updated in a single write where possible.
 * Currently anything that requires more than a single write is not
 * handled.
 */
static void dapm_seq_run(struct snd_soc_codec *codec, struct list_head *list,
             int event, int sort[])
{
    struct snd_soc_dapm_widget *w, *n;
    //创建 pending 链表
    LIST_HEAD(pending);
    int cur_sort = -1;
    int cur_reg = SND_SOC_NOPM;
    int ret;

    //遍历 list（即 up_list 或 down_list），根据成员 power_list 找到挂到 list 上的每一个 widget
    list_for_each_entry_safe(w, n, list, power_list) {
        ret = 0;

        /* Do we need to apply any queued changes? */
        if (sort[w->id] != cur_sort || w->reg != cur_reg) {
            if (!list_empty(&pending))
                dapm_seq_run_coalesced(codec, &pending);

            INIT_LIST_HEAD(&pending);
            cur_sort = -1;
            cur_reg = SND_SOC_NOPM;
        }
```

```
switch (w->id) {
//为什么类型为 pre/post 的 widget 只执行 event 回调函数？看看它们的原型就明白了。
//#define SND_SOC_DAPM_PRE(wname, wevent)，显然这些 widget 只含有 stream name 和 event
//回调函数。
case snd_soc_dapm_pre:
    if (!w->event)
        list_for_each_entry_safe_continue(w, n, list,
                          power_list);

    if (event == SND_SOC_DAPM_STREAM_START)
        ret = w->event(w,
                  NULL, SND_SOC_DAPM_PRE_PMU);
    else if (event == SND_SOC_DAPM_STREAM_STOP)
        ret = w->event(w,
                  NULL, SND_SOC_DAPM_PRE_PMD);
    break;

case snd_soc_dapm_post:
    if (!w->event)
        list_for_each_entry_safe_continue(w, n, list,
                          power_list);

    if (event == SND_SOC_DAPM_STREAM_START)
        ret = w->event(w,
                  NULL, SND_SOC_DAPM_POST_PMU);
    else if (event == SND_SOC_DAPM_STREAM_STOP)
        ret = w->event(w,
                  NULL, SND_SOC_DAPM_POST_PMD);
    break;

case snd_soc_dapm_input:
case snd_soc_dapm_output:
case snd_soc_dapm_hp:
case snd_soc_dapm_mic:
case snd_soc_dapm_line:
case snd_soc_dapm_spk:
    /* No register support currently */
    //这里就比较奇怪了，input/output/hp/mic/line/spk 这些 widgets 也只有 stream name 和
    //event，但是仍然调用 dapm_generic_apply_power 试图控制 widget 的开关？根据这里的注
    //释，应该是预留的。
    ret = dapm_generic_apply_power(w);
    break;
    default:
        /* Queue it up for application */
        //遇到非以上类型的 widget，则插入到 pending 链表，进一步调用 dapm_seq_run_coalesced
        //处理。这里设计很巧妙！下面详细解析这点。
        cur_sort = sort[w->id];
        cur_reg = w->reg;
        list_move(&w->power_list, &pending);
        break;
        }

        if (ret < 0)
            pr_err("Failed to apply widget power: %d\n",
                   ret);
    }

    if (!list_empty(&pending))
        dapm_seq_run_coalesced(codec, &pending);
}
```

从 dapm_seq_run 的分析，我们可以看出，mixer 和 mux 类型的 widgets 处理是不同的。注意如下摘出来的两段代码：

```
/* Do we need to apply any queued changes? */
if (sort[w->id] != cur_sort || w->reg != cur_reg) {
    if (!list_empty(&pending))
        dapm_seq_run_coalesced(codec, &pending);

    INIT_LIST_HEAD(&pending);
    cur_sort = -1;
    cur_reg = SND_SOC_NOPM;
}

/* Queue it up for application */
cur_sort = sort[w->id];
cur_reg = w->reg;
list_move(&w->power_list, &pending);
```

结合这两段代码可理解：遇到操作对象是同一个 reg 的 widgets，则把他们放入 pending 链表中，随后调用 dapm_seq_run_coalesced 进行处理。这样做的意义何在？见 dapm_seq_run 注释：In order to minimise the number of writes to the device required multiple widgets will be updated in a single write where possible. 保证了同 reg 但不同 widgets 的一次性读写。这设计是相当巧妙高效的。

总结：dapm 机制分析到此结束了，这篇主要简单说了下其触发过程，同时分析两个主体函

数。其实原理是简单的，复杂的地方都在于细节处理，这些在上篇已详细分析了。主要有：path 的建立过程、根据 path->list_source 找到作为 sink 的 widget、根据 path->list_sink 找到作为 source 的 widget、endpoint 的概念、complete path 的概念、depop 通电/断电次序。