DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

ALSA声卡驱动中的DAPM详解之一：kcontrol

分类： Linux音频子系统 2013-10-18 15:19 976人阅读 评论(8) 收藏 举报

alsa dapm linux 动态电源管理 音频驱动

目录(?) [-]

个人资料

DroidPhone

访问： 382367次
积分： 3435分
排名： 第2116名

原创： 46篇 转载： 0篇
译文： 4篇 评论： 356条

文章搜索

文章分类

移动开发之Android (11)
Linux内核架构 (15)
Linux设备驱动 (16)
Linux电源管理 (3)
Linux音频子系统 (15)
Linux中断子系统 (5)
Linux时间管理系统 (8)
Linux输入子系统 (4)

文章存档

2013年11月 (4)
2013年10月 (3)
2013年07月 (3)
2012年12月 (4)
2012年10月 (4)

展开

阅读排行

Android Audio System 之 (38982)
Android Audio System 之 (25553)
Android Audio System 之 (25317)
Linux ALSA声卡驱动之一 (24001)
Linux ALSA声卡驱动之二 (18421)
Android SurfaceFlinger中

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

DAPM是Dynamic Audio Power Management的缩写，直译过来就是动态音频电源管理的意思，DAPM是为了使基于linux的移动设备上的音频子系统，在任何时候都工作在最小功耗状态下。DAPM对用户空间的应用程序来说是透明的，所有与电源相关的开关都在ASoc core中完成。用户空间的应用程序无需对代码做出修改，也无需重新编译，DAPM根据当前激活的音频流（playback/capture）和声卡中的mixer等的配置来决定那些音频控件的电源开关被打开或关闭。

DAPM控件是由普通的soc音频控件演变而来的，所以本章的内容我们先从普通的soc音频控件开始。

snd_kcontrol_new结构

在正式讨论DAPM之前，我们需要先搞清楚ASoc中的一个重要的概念：kcontrol，不熟悉的读者需要浏览一下我之前的文章：Linux ALSA声卡驱动之四：Control设备的创建。通常，一个kcontrol代表着一个mixer（混音器），或者是一个mux（多路开关），又或者是一个音量控制器等等。 从上述文章中我们知道，定义一个kcontrol主要就是定义一个snd_kcontrol_new结构，为了方便讨论，这里再次给出它的定义：

```cpp
struct snd_kcontrol_new {
        snd_ctl_elem_iface_t iface;     /* interface identifier */
        unsigned int device;            /* device/client number */
        unsigned int subdevice;         /* subdevice (substream) number */
        const unsigned char *name;      /* ASCII name of item */
        unsigned int index;             /* index of item */
        unsigned int access;            /* 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;
};
```

回到Linux ALSA声卡驱动之四：Control设备的创建中，我们知道，对于每个控件，我们需要定义一个和他对应的snd_kcontrol_new结构，这些snd_kcontrol_new结构会在声卡的初始化阶段，通过snd_soc_add_codec_controls函数注册到系统中，用户空间就可以通过amixer或alsamixer等工具查看和设定这些控件的状态。

snd_kcontrol_new结构中，几个主要的字段是get，put，private_value，get回调函数用于获取该控件当前的状态值，而put回调函数则用于设置控件的状态值，而private_value字段则根据不同的控件类型有不同的意义，比如对于普通的控件，private_value字段可以用来定义该控件所对应的寄存器的地址以及对应的控制位在寄存器中的位置信息。值得庆幸的是，ASoc系统已经为我们准备了大量的宏定义，用于定义常用的控件，这些宏定义位于include/sound/soc.h中。下面我们分别讨论一下如何用这些预设的宏定义来定义一些常用的控件。

简单型的控件

SOC_SINGLE SOC_SINGLE应该算是最简单的控件了，这种控件只有一个控制量，比如一个开关，或者是一个数值变量（比如Codec中某个频率，FIFO大小等等）。我们看看这个宏是如何定义的：

```cpp
#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) }
```

宏定义的参数分别是：xname（该控件的名字），reg（该控件对应的寄存器的地址），shift（控制位在寄存器中的位移），max（控件可设置的最大值），invert（设定值是否逻辑取反）。这里又使用了一个宏来定义private_value字段：SOC_SINGLE_VALUE，我们看看它的定义：

```cpp
#define SOC_DOUBLE_VALUE(xreg, shift_left, shift_right, xmax, xinvert) \
```

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

```
        ((unsigned long)&(struct soc_mixer_control) \
        {.reg = xreg, .rreg = xreg, .shift = shift_left, \
        .rshift = shift_right, .max = xmax, .platform_max = xmax, \
        .invert = xinvert})
#define SOC_SINGLE_VALUE(xreg, xshift, xmax, xinvert) \
        SOC_DOUBLE_VALUE(xreg, xshift, xshift, xmax, xinvert)
```

这里实际上是定义了一个soc_mixer_control结构，然后把该结构的地址赋值给了private_value字段，soc_mixer_control结构是这样的：

```cpp
/* mixer control */
struct soc_mixer_control {
        int min, max, platform_max;
        unsigned int reg, rreg, shift, rshift, invert;
};
```

看来soc_mixer_control是控件特征的真正描述者，它确定了该控件对应寄存器的地址，位移值，最大值和是否逻辑取反等特性，控件的put回调函数和get回调函数需要借助该结构来访问实际的寄存器。我们看看这get回调函数的定义：

```cpp
int snd_soc_get_volsw(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_value *ucontrol)
{
        struct soc_mixer_control *mc =
                (struct soc_mixer_control *)kcontrol->private_value;
        struct snd_soc_codec *codec = snd_kcontrol_chip(kcontrol);
        unsigned int reg = mc->reg;
        unsigned int reg2 = mc->rreg;
        unsigned int shift = mc->shift;
        unsigned int rshift = mc->rshift;
        int max = mc->max;
        unsigned int mask = (1 << fls(max)) - 1;
        unsigned int invert = mc->invert;

        ucontrol->value.integer.value[0] =
                (snd_soc_read(codec, reg) >> shift) & mask;
        if (invert)
                ucontrol->value.integer.value[0] =
                        max - ucontrol->value.integer.value[0];

        if (snd_soc_volsw_is_stereo(mc)) {
                if (reg == reg2)
                        ucontrol->value.integer.value[1] =
                                (snd_soc_read(codec, reg) >> rshift) & mask;
                else
                        ucontrol->value.integer.value[1] =
                                (snd_soc_read(codec, reg2) >> shift) & mask;
                if (invert)
                        ucontrol->value.integer.value[1] =
                                max - ucontrol->value.integer.value[1];
        }

        return 0;
}
```

上述代码一目了然，从private_value字段取出soc_mixer_control结构，利用该结构的信息，访问对应的寄存器，返回相应的值。

SOC_SINGLE_TLV　SOC_SINGLE_TLV是SOC_SINGLE的一种扩展，主要用于定义那些有增益控制的控件，例如音量控制器，EQ均衡器等等。

```cpp
#define SOC_SINGLE_TLV(xname, reg, shift, max, invert, 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, .get = snd_soc_get_volsw,\
        .put = snd_soc_put_volsw, \
        .private_value =  SOC_SINGLE_VALUE(reg, shift, max, invert) }
```

从他的定义可以看出，用于设定寄存器信息的private_value字段的定义和SOC_SINGLE是一样的，甚至put、get回调函数也是使用同一套，唯一不同的是增加了一个tlv_array参数，并把它赋值给了tlv.p字段。关于tlv，已经在Linux ALSA声卡驱动之四：Control设备的创建中进行了阐述。用户空间可以通过对声卡的control设备发起以下两种ioctl来访问tlv字段所指向的数组：

- SNDRV_CTL_IOCTL_TLV_READ
- SNDRV_CTL_IOCTL_TLV_WRITE
- SNDRV_CTL_IOCTL_TLV_COMMAND

通常，tlv_array用来描述寄存器的设定值与它所代表的实际意义之间的映射关系，最常用的就是用于音量控件时，设定值与对应的dB值之间的映射关系，请看以下例子：

```cpp
static const DECLARE_TLV_DB_SCALE(mixin_boost_tlv, 0, 900, 0);

static const struct snd_kcontrol_new wm1811_snd_controls[] = {
SOC_SINGLE_TLV("MIXINL IN1LP Boost Volume", WM8994_INPUT_MIXER_1, 7, 1, 0,
               mixin_boost_tlv),
SOC_SINGLE_TLV("MIXINL IN1RP Boost Volume", WM8994_INPUT_MIXER_1, 8, 1, 0,
               mixin_boost_tlv),
};
```

DECLARE_TLV_DB_SCALE用于定义一个dB值映射的tlv_array，上述的例子表明，该tlv的类型是SNDRV_CTL_TLVT_DB_SCALE，寄存器的最小值对应是0dB，寄存器每增加一个单位值，对应的dB数增加是9dB（0.01dB*900），而由接下来的两组SOC_SINGLE_TLV定义可以看出，我们定义了两个boost控件，寄存器的地址都是WM8994_INPUT_MIXER_1，控制位分别是第7bit和第8bit，最大值是1，所以，该控件只能设定两个数值0和1，对应的dB值就是0dB和9dB。

SOC_DOUBLE 与SOC_SINGLE相对应，区别是SOC_SINGLE只控制一个变量，而SOC_DOUBLE则可以同时在一个寄存器中控制两个相似的变量，最常用的就是用于一些立体声的控件，我们需要同时对左右声道进行控制，因为多了一个声道，参数也就相应地多了一个shift位移值，

```cpp
#define SOC_DOUBLE(xname, reg, shift_left, shift_right, 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_DOUBLE_VALUE(reg, shift_left, shift_right, \
                                          max, invert) }
```

SOC_DOUBLE_R 与SOC_DOUBLE类似，对于左右声道的控制寄存器不一样的情况，使用SOC_DOUBLE_R来定义，参数中需要指定两个寄存器地址。

SOC_DOUBLE_TLV 与SOC_SINGLE_TLV对应的立体声版本，通常用于立体声音量控件的定义。

SOC_DOUBLE_R_TLV 左右声道有独立寄存器控制的SOC_DOUBLE_TLV版本

Mixer控件

Mixer控件用于音频通道的路由控制，由多个输入和一个输出组成，多个输入可以自由地混合在一起，形成混合后的输出：

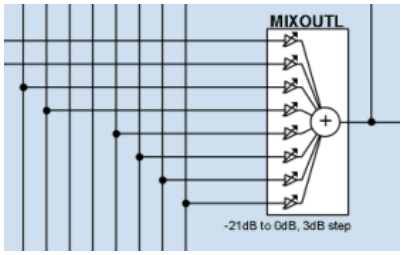


图1 Mixer混音器

对于Mixer控件，我们可以认为是多个简单控件的组合，通常，我们会为mixer的每个输入端都单独定义一个简单

控件来控制该路输入的开启和关闭，反应在代码上，就是定义一个soc_kcontrol_new数组：

```cpp
static const struct snd_kcontrol_new left_speaker_mixer[] = {
SOC_SINGLE("Input Switch", WM8993_SPEAKER_MIXER, 7, 1, 0),
SOC_SINGLE("IN1LP Switch", WM8993_SPEAKER_MIXER, 5, 1, 0),
SOC_SINGLE("Output Switch", WM8993_SPEAKER_MIXER, 3, 1, 0),
SOC_SINGLE("DAC Switch", WM8993_SPEAKER_MIXER, 6, 1, 0),
};
```

以上这个mixer使用寄存器WM8993_SPEAKER_MIXER的第3，5，6，7位来分别控制4个输入端的开启和关闭。

Mux控件

mux控件与mixer控件类似，也是多个输入端和一个输出端的组合控件，与mixer控件不同的是，mux控件的多个输入端同时只能有一个被选中。因此，mux控件所对应的寄存器，通常可以设定一段连续的数值，每个不同的数值对应不同的输入端被打开，与上述的mixer控件不同，ASoc用soc_enum结构来描述mux控件的寄存器信息：

```cpp
/* enumerated kcontrol */
struct soc_enum {
        unsigned short reg;
        unsigned short reg2;
        unsigned char shift_l;
        unsigned char shift_r;
        unsigned int max;
        unsigned int mask;
        const char * const *texts;
        const unsigned int *values;
};
```

两个寄存器地址和位移字段：reg，reg2，shift_l，shift_r，用于描述左右声道的控制寄存器信息。字符串数组指针用于描述每个输入端对应的名字，value字段则指向一个数组，该数组定义了寄存器可以选择的值，每个值对应一个输入端，如果value是一组连续的值，通常我们可以忽略values参数。下面我们先看看如何定义一个mux控件：

第一步，定义字符串和values数组，以下的例子因为values是连续的，所以不用定义：

```cpp
static const char *drc_path_text[] = {
        "ADC",
        "DAC"
};
```

第二步，利用ASoc提供的辅助宏定义soc_enum结构，用于描述寄存器：

```cpp
static const struct soc_enum drc_path =
        SOC_ENUM_SINGLE(WM8993_DRC_CONTROL_1, 14, 2, drc_path_text);
```

第三步，利痛ASoc提供的辅助宏，定义soc_kcontrol_new结构，该结构最后用于注册该mux控件：

```cpp
static const struct snd_kcontrol_new wm8993_snd_controls[] = {
SOC_DOUBLE_TLV(......),
......
SOC_ENUM("DRC Path", drc_path),
......
}
```

以上几步定义了一个叫DRC PATH的mux控件，该控件具有两个输入选择，分别是来自ADC和DAC，用寄存器WM8993_DRC_CONTROL_1控制。其中，soc_enum结构使用了辅助宏SOC_ENUM_SINGLE来定义，该宏的声明如下：

```cpp
#define SOC_ENUM_DOUBLE(xreg, xshift_l, xshift_r, xmax, xtexts) \
{       .reg = xreg, .shift_l = xshift_l, .shift_r = xshift_r, \
        .max = xmax, .texts = xtexts, \
```

```
        .mask = xmax ? roundup_pow_of_two(xmax) - 1 : 0}
#define SOC_ENUM_SINGLE(xreg, xshift, xmax, xtexts) \
        SOC_ENUM_DOUBLE(xreg, xshift, xshift, xmax, xtexts)
```

定义soc_kcontrol_new结构时使用了SOC_ENUM，列出它的定义如下：

```
[cpp]

#define SOC_ENUM(xname, xenum) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname,\
        .info = snd_soc_info_enum_double, \
        .get = snd_soc_get_enum_double, .put = snd_soc_put_enum_double, \
        .private_value = (unsigned long)&xenum }
```

思想如此统一，依然是使用private_value字段记录soc_enum结构，不过几个回调函数变了，我们看看get回调对应的snd_soc_get_enum_double函数：

```
[cpp]

int snd_soc_get_enum_double(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_value *ucontrol)
{
        struct snd_soc_codec *codec = snd_kcontrol_chip(kcontrol);
        struct soc_enum *e = (struct soc_enum *)kcontrol->private_value;
        unsigned int val;

        val = snd_soc_read(codec, e->reg);
        ucontrol->value.enumerated.item[0]
                = (val >> e->shift_l) & e->mask;
        if (e->shift_l != e->shift_r)
                ucontrol->value.enumerated.item[1] =
                        (val >> e->shift_r) & e->mask;

        return 0;
}
```

通过info回调函数则可以获取某个输入端所对应的名字，其实就是从soc_enum结构的texts数组中获得：

```
[cpp]

int snd_soc_info_enum_double(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_info *uinfo)
{
        struct soc_enum *e = (struct soc_enum *)kcontrol->private_value;

        uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
        uinfo->count = e->shift_l == e->shift_r ? 1 : 2;
        uinfo->value.enumerated.items = e->max;

        if (uinfo->value.enumerated.item > e->max - 1)
                uinfo->value.enumerated.item = e->max - 1;
        strcpy(uinfo->value.enumerated.name,
                e->texts[uinfo->value.enumerated.item]);
        return 0;
}
```

以下是另外几个常用于定义mux控件的宏：

SOC_VALUE_ENUM_SINGLE　用于定义带values字段的soc_enum结构。

SOC_VALUE_ENUM_DOUBLE　SOC_VALUE_ENUM_SINGLE的立体声版本。

SOC_VALUE_ENUM　用于定义带values字段snd_kcontrol_new结构，这个有点特别，我们还是看看它的定义：

```
[cpp]

#define SOC_VALUE_ENUM(xname, xenum) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname,\
        .info = snd_soc_info_enum_double, \
        .get = snd_soc_get_value_enum_double, \
        .put = snd_soc_put_value_enum_double, \
        .private_value = (unsigned long)&xenum }
```

从定义可以看出来，回调函数被换掉了，我们看看他的get回调：

```
[cpp]
```

```
int snd_soc_get_value_enum_double(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_value *ucontrol)
{
        struct snd_soc_codec *codec = snd_kcontrol_chip(kcontrol);
        struct soc_enum *e = (struct soc_enum *)kcontrol->private_value;
        unsigned int reg_val, val, mux;

        reg_val = snd_soc_read(codec, e->reg);
        val = (reg_val >> e->shift_l) & e->mask;
        for (mux = 0; mux < e->max; mux++) {
                if (val == e->values[mux])
                        break;
        }
        ucontrol->value.enumerated.item[0] = mux;
        if (e->shift_l != e->shift_r) {
                val = (reg_val >> e->shift_r) & e->mask;
                for (mux = 0; mux < e->max; mux++) {
                        if (val == e->values[mux])
                                break;
                }
                ucontrol->value.enumerated.item[1] = mux;
        }

        return 0;
}
```

与SOC_ENUM定义的mux不同，它没有直接返回寄存器的设定值，而是通过soc_enum结构中的values字段做了一次转换，与values数组中查找和寄存器相等的值，然后返回他在values数组中的索引值，所以，尽管寄存器的值可能是不连续的，但返回的值是连续的。

通常，我们还可以用以下几个辅助宏定义soc_enum结构，其实和上面所说的没什么区别，只是可以偷一下懒，省掉struct soc_enum xxxx=几个单词而已：

- **SOC_ENUM_SINGLE_DECL**
- **SOC_ENUM_DOUBLE_DECL**
- **SOC_VALUE_ENUM_SINGLE_DECL**
- **SOC_VALUE_ENUM_DOUBLE_DECL**

其它控件

其实，除了以上介绍的几种常用的控件，ASoc还为我们提供了另外一些控件定义辅助宏，详细的请读者参考include/sound/soc.h。这里列举几个：

需要自己定义get和put回调时，可以使用以下这些带EXT的版本：

- **SOC_SINGLE_EXT**
- **SOC_DOUBLE_EXT**
- **SOC_SINGLE_EXT_TLV**
- **SOC_DOUBLE_EXT_TLV**
- **SOC_DOUBLE_R_EXT_TLV**
- **SOC_ENUM_EXT**

更多 1

上一篇：Linux动态频率调节系统CPUFreq之三：governor

下一篇：ALSA声卡驱动中的DAPM详解之二：widget-具备路径和电源管理信息的kcontrol

查看评论

6楼 imliuda 2013-11-27 17:25发表

讲得很好！谢谢！

5楼 yronaldo 2013-11-26 19:51发表

"用户空间就可以通过amixer或alsamixer等工具查看和设定这些控件的状态。"
如果使用tinyalsa的话用什么工具查看控件状态呢？

Re: kevinyujm 2013-12-16 18:41发表

回复yronaldo：tinymix

4楼 jeffreyliu 2013-11-05 20:19发表

感谢对您和AZURE两位大神对DAPM的分享,音频动态电源管理相关的内容资料太少了

3楼 shaohua312 2013-10-31 20:59发表

感谢版主的分享！！！

2楼 silvervi 2013-10-22 16:11发表

最新的文章啊，最近也在学DAPM，期待楼主更新下一篇

1楼 ComputerSience123 2013-10-18 17:45发表

楼主什么时候写一下DAPM？觉得这一部分逻辑比较复杂，不是很好理解。

Re: DroidPhone 2013-10-18 20:24发表

回复ComputerSience123：嗯，会的，DAPM比较复杂，要分成多个博文才能说清楚，本篇只是开个头。

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382367次
积分： 3435分
排名： 第2116名

原创： 46篇 转载： 0篇
译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之二：widget-具备路径和电源管理信息的kcontrol

分类： Linux音频子系统 2013-10-23 20:31 644人阅读 评论(2) 收藏 举报

上一篇文章中，我们介绍了音频驱动中对基本控制单元的封装：kcontrol。利用kcontrol，我们可以完成对音频系统中的mixer，mux，音量控制，音效控制，以及各种开关量的控制，通过对各种kcontrol的控制，使得音频硬件能够按照我们预想的结果进行工作。同时我们可以看到，kcontrol还是有以下几点不足：

- 只能描述自身，无法描述各个kcontrol之间的连接关系；
- 没有相应的电源管理机制；
- 没有相应的时间处理机制来响应播放、停止、上电、下电等音频事件；
- 为了防止pop-pop声，需要用户程序关注各个kcontrol上电和下电的顺序；
- 当一个音频路径不再有效时，不能自动关闭该路径上的所有的kcontrol；

/***/
声明：本博内容均由http://blog.csdn.net/droidphone原创，转载请注明出处，谢谢！
/***/

为此，DAPM框架正是为了要解决以上这些问题而诞生的，DAPM目前已经是ASoc中的重要组成部分，让我们先从DAPM的数据结构开始，了解它的设计思想和工作原理。

DAPM的基本单元：widget

文章的开头，我们说明了一下目前kcontrol的一些不足，而DAPM框架为了解决这些问题，引入了widget这一概念，所谓widget，其实可以理解为是kcontrol的进一步升级和封装，她同样是指音频系统中的某个部件，比如mixer，mux，输入输出引脚，电源供应器等等，甚至，我们可以定义虚拟的widget，例如playback stream widget。widget把kcontrol和动态电源管理进行了有机的结合，同时还具备音频路径的连结功能，一个widget可以与它相邻的widget有某种动态的连结关系。在DAPM框架中，widget用结构体snd_soc_dapm_widget来描述：

```
[cpp]

struct snd_soc_dapm_widget {
        enum snd_soc_dapm_type id;
        const char *name;               /* widget name */

        ......
        /* dapm control */
        int reg;                                /* negative reg = no direct dapm */
        unsigned char shift;                    /* bits to shift */
        unsigned int value;                             /* widget current value */
        unsigned int mask;                      /* non-shifted mask */
        ......

        int (*power_check)(struct snd_soc_dapm_widget *w);

        int (*event)(struct snd_soc_dapm_widget*, struct snd_kcontrol *, int);

```

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

```
17.         /* kcontrols that relate to this widget */
18.         int num_kcontrols;
19.         const struct snd_kcontrol_new *kcontrol_news;
20.         struct snd_kcontrol **kcontrols;
21.
22.         /* widget input and outputs */
23.         struct list_head sources;
24.         struct list_head sinks;
25.         ......
26. };
```

snd_soc_dapm_widget结构比较大，为了简洁一些，这里我没有列出该结构体的完整字段，不过不用担心，下面我会说明每个字段的意义：

id　该widget的类型值，比如snd_soc_dapm_output，snd_soc_dapm_mixer等等。

***name**　该widget的名字

***sname**　代表该widget所在stream的名字，比如对于snd_soc_dapm_dai_in类型的widget，会使用该字段。

***codec *platform**　指向该widget所属的codec和platform。

list　所有注册到系统中的widget都会通过该list，链接到代表声卡的snd_soc_card结构的widgets链表头字段中。

***dapm**　snd_soc_dapm_context结构指针，ASoc把系统划分为多个dapm域，每个widget属于某个dapm域，同一个域代表着同样的偏置电压供电策略，比如，同一个codec中的widget通常位于同一个dapm域，而平台上的widget可能又会位于另外一个platform域中。

***priv**　有些widget可能需要一些专有的数据，可以使用该字段来保存，像snd_soc_dapm_dai_in类型的widget，会使用该字段来记住与之相关联的snd_soc_dai结构指针。

***regulator**　对于snd_soc_dapm_regulator_supply类型的widget，该字段指向与之相关的regulator结构指针。

***params**　目前对于snd_soc_dapm_dai_link类型的widget，指向该dai的配置信息的snd_soc_pcm_stream结构。

reg shift mask　这3个字段用来控制该widget的电源状态，分别对应控制信息所在的寄存器地址，位移值和屏蔽值。

value on_val off_val　电源状态的当前只，开启时和关闭时所对应的值。

power invert　用于指示该widget当前是否处于上电状态，invert则用于表明power字段是否需要逻辑反转。

active connected　分别表示该widget是否处于激活状态和连接状态，当和相邻的widget有连接关系时，connected位会被置1，否则置0。

new　我们定义好的widget（snd_soc_dapm_widget结构），在注册到声卡中时需要进行实例化，该字段用来表示该widget是否已经被实例化。

ext　表示该widget当前是否有外部连接，比如连接mic，耳机，喇叭等等。

force　该位被设置后，将会不管widget当前的状态，强制更新至新的电源状态。

ignore_suspend new_power power_checked　这些电源管理相关的字段。

subseq　该widget目前在上电或下电队列中的排序编号，为了防止在上下电的过程中出现pop-pop声，DAPM会给每个widget分配合理的上下电顺序。

***power_check**　用于检查该widget是否应该上电或下电的回调函数指针。
event_flags　该字段是一个位或字段，每个位代表该widget会关注某个DAPM事件通知。只有被关注的通知事件会被发送到widget的事件处理回调函数中。

***event**　DAPM事件处理回调函数指针。

num_kcontrols *kcontrol_news **kcontrols　这3个字段用来描述与该widget所包含的kcontrol控件，例如一个mixer控件或者是一个mux控件。

sources sinks　两个链表字段，两个widget如果有连接关系，会通过一个snd_soc_dapm_path结构进行连

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

接，sources链表用于链接所有的输入path，sinks链表用于链接所有的输出path。

power_list 每次更新整个dapm的电源状态时，会根据一定的算法扫描所有的widget，然后把需要变更电源状态的widget利用该字段链接到一个上电或下电的链表中，扫描完毕后，dapm系统会遍历这两个链表执行相应的上电或下电操作。

dirty 链表字段，widget的状态变更后，dapm系统会利用该字段，把该widget加入到一个dirty链表中，稍后会对dirty链表进行扫描，以执行整个路径的更新。

inputs 该widget的所有有效路径中，连接到输入端的路径数量。

outputs 该widget的所有有效路径中，连接到输出端的路径数量。

***clk** 对于snd_soc_dapm_clock_supply类型的widget，指向相关联的clk结构指针。

以上我们对snd_soc_dapm_widget结构的各个字段所代表的意义一一做出了说明，这里只是让大家现有个概念，至于每个字段的详细作用，我们会在以后相关的章节中提及。

widget的种类

在DAPM框架中，把各种不同的widget划分为不同的种类，snd_soc_dapm_widget结构中的id字段用来表示该widget的种类，可选的种类都定义在一个枚举中：

```cpp
/* dapm widget types */
enum snd_soc_dapm_type {......}
```

下面我们逐个解释一下这些widget的种类：

- **snd_soc_dapm_input** 该widget对应一个输入引脚。
- **snd_soc_dapm_output** 该widget对应一个输出引脚。
- **snd_soc_dapm_mux** 该widget对应一个mux控件。
- **snd_soc_dapm_virt_mux** 该widget对应一个虚拟的mux控件。
- **snd_soc_dapm_value_mux** 该widget对应一个value类型的mux控件。
- **snd_soc_dapm_mixer** 该widget对应一个mixer控件。
- **snd_soc_dapm_mixer_named_ctl** 该widget对应一个mixer控件，但是对应的kcontrol的名字不会加入widget的名字作为前缀。
- **snd_soc_dapm_pga** 该widget对应一个pga控件（可编程增益控件）。
- **snd_soc_dapm_out_drv** 该widget对应一个输出驱动控件
- **snd_soc_dapm_adc** 该widget对应一个ADC
- **snd_soc_dapm_dac** 该widget对应一个DAC
- **snd_soc_dapm_micbias** 该widget对应一个麦克风偏置电压控件
- **snd_soc_dapm_mic** 该widget对应一个麦克风。
- **snd_soc_dapm_hp** 该widget对应一个耳机。
- **snd_soc_dapm_spk** 该widget对应一个扬声器。
- **snd_soc_dapm_line** 该widget对应一个线路输入。
- **snd_soc_dapm_switch** 该widget对应一个模拟开关。
- **snd_soc_dapm_vmid** 该widget对应一个codec的vmid偏置电压。
- **snd_soc_dapm_pre** machine级别的专用widget，会先于其它widget执行检查操作。
- **snd_soc_dapm_post** machine级别的专用widget，会后于其它widget执行检查操作。
- **snd_soc_dapm_supply** 对应一个电源或是时钟源。
- **snd_soc_dapm_regulator_supply** 对应一个外部regulator稳压器。
- **snd_soc_dapm_clock_supply** 对应一个外部时钟源。
- **snd_soc_dapm_aif_in** 对应一个数字音频输入接口，比如I2S接口的输入端。
- **snd_soc_dapm_aif_out** 对应一个数字音频输出接口，比如I2S接口的输出端。
- **snd_soc_dapm_siggen** 对应一个信号发生器。
- **snd_soc_dapm_dai_in** 对应一个platform或codec域的输入DAI结构。
- **snd_soc_dapm_dai_out** 对应一个platform或codec域的输出DAI结构。
- **snd_soc_dapm_dai_link** 用于链接一对输入/输出DAI结构。

widget之间的连接器：path

之前已经提到，一个widget是有输入和输出的，而且widget之间是可以动态地进行连接的，那它们是用什么来连接两个widget的呢？DAPM为我们提出了path这一概念，path相当于电路中的一根跳线，它把一个widget的输出端和另一个widget的输入端连接在一起，path用snd_soc_dapm_path结构来描述：

```cpp
struct snd_soc_dapm_path {
        const char *name;

        /* source (input) and sink (output) widgets */
        struct snd_soc_dapm_widget *source;
        struct snd_soc_dapm_widget *sink;
        struct snd_kcontrol *kcontrol;

        /* status */
        u32 connect:1;  /* source and sink widgets are connected */
        u32 walked:1;   /* path has been walked */
        u32 walking:1;  /* path is in the process of being walked */
        u32 weak:1;     /* path ignored for power management */

        int (*connected)(struct snd_soc_dapm_widget *source,
                         struct snd_soc_dapm_widget *sink);

        struct list_head list_source;
        struct list_head list_sink;
        struct list_head list;
};
```

当widget之间发生连接关系时，snd_soc_dapm_path作为连接者，它的source字段会指向该连接的起始端widget，而它的sink字段会指向该连接的到达端widget，还记得前面snd_soc_dapm_widget结构中的两个链表头字段：sources和sinks么？widget的输入端和输出端可能连接着多个path，所有输入端的snd_soc_dapm_path结构通过list_sink字段挂在widget的souces链表中，同样，所有输出端的snd_soc_dapm_path结构通过list_source字段挂在widget的sinks链表中。这里可能大家会被搞得晕呼呼的，一会source，一会sink，不要紧，只要记住，连接的路径是这样的：起始端widget的输出-->path的输入-->path的输出-->到达端widget输入。

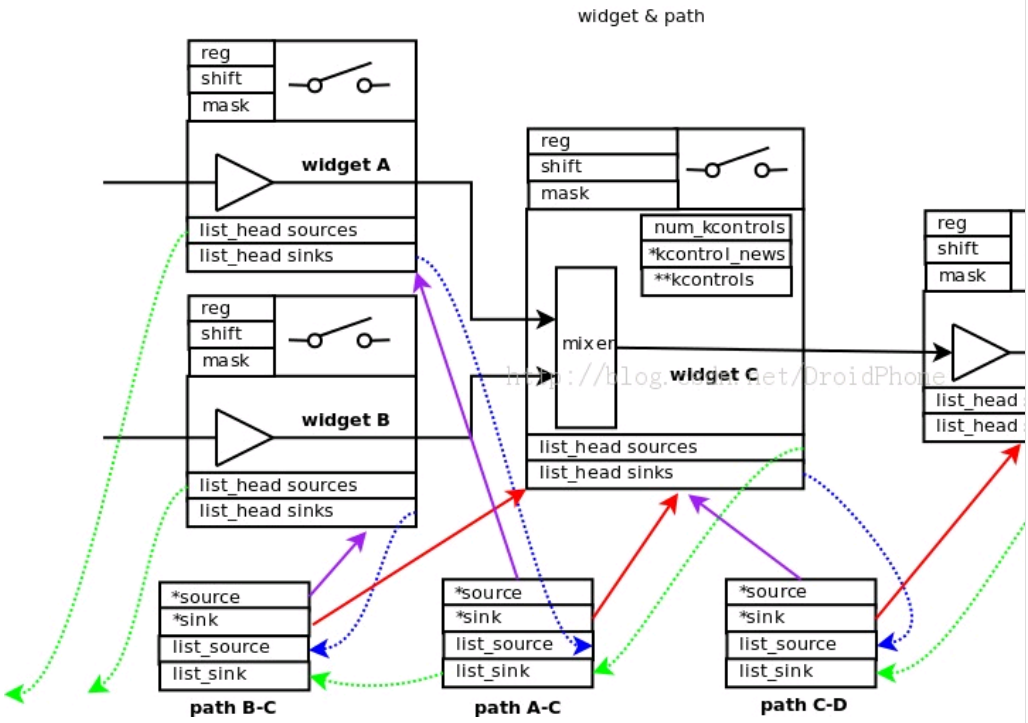


图1　widget通过path进行连接

另外，snd_soc_dapm_path结构的list字段用于把所有的path注册到声卡中，其实就是挂在snd_soc_card结构的paths链表头字段中。如果你要自己定义方法来检查path的当前连接状态，你可以提供自己的connected回调函数指针。

connect，walked，walking，weak是几个辅助字段，用于帮助所有path的遍历。

widget的连接关系：route

通过上一节的内容，我们知道，一个路径的连接至少包含以下几个元素：起始端widget，跳线path，到达端widget，在DAPM中，用snd_soc_dapm_route结构来描述这样一个连接关系：

```cpp
struct snd_soc_dapm_route {
        const char *sink;
        const char *control;
        const char *source;
        int (*connected)(struct snd_soc_dapm_widget *source,
                         struct snd_soc_dapm_widget *sink);
};
```

sink指向到达端widget的名字字符串，source指向起始端widget的名字字符串，control指向负责控制该连接所对应的kcontrol名字字符串，connected回调则定义了上一节所提到的自定义连接检查回调函数。该结构的意义很明显就是：source通过一个kcontrol，和sink连接在一起，现在是否处于连接状态，请调用connected回调函数检查。

这里直接使用名字字符串来描述连接关系，所有定义好的route，最后都要注册到dapm系统中，dapm会根据这些名字找出相应的widget，并动态地生成所需要的snd_soc_dapm_path结构，正确地处理各个链表和指针的关系，实现两个widget之间的连接，具体的连接代码分析，我们留到以后的章节中讨论。

上一篇：ALSA声卡驱动中的DAPM详解之一：kcontrol
下一篇：ALSA声卡驱动中的DAPM详解之三：如何定义各种widget

查看评论

2楼 jeffreyliu 2013-11-05 20:40发表

博主如遇到LINE IN输入的时候(用作模拟输入通道非MIC) 在切换到该通道的时候发现该条路径上的寄存器值不能更新过来只有当打开录音流的时候才会更新寄存器的值,组件注册如下：
/* Input path */
SND_SOC_DAPM_ADC("ADC Left", "Capture", CS42L52_PWRCTL1, 1, 1),
SND_SOC_DAPM_ADC("ADC Right", "Capture", CS42L52_PWRCTL1, 2, 1),
如果去掉Capture填充NULL参数CS42L52_PWRCTL1寄存器的值会更新但是该PATH的下游PATH又不能导通 请问这是什么情况？

1楼 Color_Fly 2013-11-01 09:37发表

抢个沙发，感谢博主分享！！！

Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382367次

积分： 3435分

排名： 第2116名

原创： 46篇 转载： 0篇

译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之三：如何定义各种widget

分类： Linux音频子系统 2013-10-24 21:01 700人阅读 评论(9) 收藏 举报

linux dapm widget audio driver

目录(?) [-]

上一节中，介绍了DAPM框架中几个重要的数据结构：snd_soc_dapm_widget，snd_soc_dapm_path，snd_soc_dapm_route。其中snd_soc_dapm_path无需我们自己定义，它会在注册snd_soc_dapm_route时动态地生成，但是对于系统中的widget和route，我们是需要自己进行定义的，另外，widget所包含的kcontrol与普通的kcontrol有所不同，它们的定义方法与标准的kcontrol也有所不同。本节的内容我将会介绍如何使用DAPM系统提供的一些辅助宏定义来定义各种类型的widget和它所用到的kcontrol。

定义widget

和普通的kcontrol一样，DAPM框架为我们提供了大量的辅助宏用来定义各种各样的widget控件，这些宏定义根据widget的类型，按照它们的电源所在的域，被分为了几个域，他们分别是：

- **codec域** 比如VREF和VMID等提供参考电压的widget，这些widget通常在codec的probe/remove回调中进行控制，当然，在工作中如果没有音频流时，也可以适当地进行控制它们的开启与关闭。
- **platform域** 位于该域上的widget通常是针对平台或板子的一些需要物理连接的输入/输出接口，例如耳机、扬声器、麦克风，因为这些接口在每块板子上都可能不一样，所以通常它们是在machine驱动中进行定义和控制，并且也可以由用户空间的应用程序通过某种方式来控制它们的打开和关闭。
- **音频路径域** 一般是指codec内部的mixer、mux等控制音频路径的widget，这些widget可以根据用户空间的设定连接关系，自动设定他们的电源状态。
- **音频数据流域** 是指那些需要处理音频数据流的widget，例如ADC、DAC等等。

codec域widget的定义

目前，DAPM框架只提供了定义一个codec域widget的辅助宏：

```
[cpp]

#define SND_SOC_DAPM_VMID(wname) \
{       .id = snd_soc_dapm_vmid, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0}
```

platform域widget的定义

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

DAPM框架为我们提供了多种platform域widget的辅助定义宏：

```cpp
#define SND_SOC_DAPM_SIGGEN(wname) \
{       .id = snd_soc_dapm_siggen, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM }
#define SND_SOC_DAPM_INPUT(wname) \
{       .id = snd_soc_dapm_input, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM }
#define SND_SOC_DAPM_OUTPUT(wname) \
{       .id = snd_soc_dapm_output, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM }
#define SND_SOC_DAPM_MIC(wname, wevent) \
{       .id = snd_soc_dapm_mic, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM, .event = wevent, \
        .event_flags = SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD}
#define SND_SOC_DAPM_HP(wname, wevent) \
{       .id = snd_soc_dapm_hp, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM, .event = wevent, \
        .event_flags = SND_SOC_DAPM_POST_PMU | SND_SOC_DAPM_PRE_PMD}
#define SND_SOC_DAPM_SPK(wname, wevent) \
{       .id = snd_soc_dapm_spk, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM, .event = wevent, \
        .event_flags = SND_SOC_DAPM_POST_PMU | SND_SOC_DAPM_PRE_PMD}
#define SND_SOC_DAPM_LINE(wname, wevent) \
{       .id = snd_soc_dapm_line, .name = wname, .kcontrol_news = NULL, \
        .num_kcontrols = 0, .reg = SND_SOC_NOPM, .event = wevent, \
        .event_flags = SND_SOC_DAPM_POST_PMU | SND_SOC_DAPM_PRE_PMD}
```

以上这些widget分别对应信号发生器，输入引脚，输出引脚，麦克风，耳机，扬声器，线路输入接口。其中的reg字段被设置为SND_SOC_NOPM（-1），表明这些widget是没有寄存器控制位来控制widget的电源状态的。麦克风，耳机，扬声器，线路输入接口这几种widget，还可以定义一个dapm事件回调函数wevent，从event_flags字段的设置可以看出，他们只会响应SND_SOC_DAPM_POST_PMU（上电后）和SND_SOC_DAPM_PMD（下电前）事件，这几个widget通常会在machine驱动中定义，而SND_SOC_DAPM_INPUT和SND_SOC_DAPM_OUTPUT则用来定义codec芯片的输出输入脚，通常在codec驱动中定义，最后，在machine驱动中增加相应的route，把麦克风和耳机等widget与相应的codec输入输出引脚的widget连接起来。

音频路径（path）域widget的定义

这种widget通常是对普通kcontrols控件的再封装，增加音频路径和电源管理功能，所以这种widget会包含一个或多个kcontrol，普通kcontrol的定义方法我们在ALSA声卡驱动中的DAPM详解之一：kcontrol中已经介绍过，不过这些被包含的kcontrol不能使用这种方法定义，它们需要使用dapm框架提供的定义宏来定义，详细的讨论我们后面有介绍。这里先列出这些widget的定义宏：

```cpp
#define SND_SOC_DAPM_PGA(wname, wreg, wshift, winvert,\
         wcontrols, wncontrols) \
{       .id = snd_soc_dapm_pga, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = wncontrols}
#define SND_SOC_DAPM_OUT_DRV(wname, wreg, wshift, winvert,\
         wcontrols, wncontrols) \
{       .id = snd_soc_dapm_out_drv, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = wncontrols}
#define SND_SOC_DAPM_MIXER(wname, wreg, wshift, winvert, \
         wcontrols, wncontrols)\
{       .id = snd_soc_dapm_mixer, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = wncontrols}
#define SND_SOC_DAPM_MIXER_NAMED_CTL(wname, wreg, wshift, winvert, \
         wcontrols, wncontrols)\
{       .id = snd_soc_dapm_mixer_named_ctl, .name = wname, .reg = wreg, \
        .shift = wshift, .invert = winvert, .kcontrol_news = wcontrols, \
        .num_kcontrols = wncontrols}
#define SND_SOC_DAPM_MICBIAS(wname, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_micbias, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = NULL, .num_kcontrols = 0}
#define SND_SOC_DAPM_SWITCH(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_switch, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = 1}
#define SND_SOC_DAPM_MUX(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_mux, .name = wname, .reg = wreg, .shift = wshift, \
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = 1}
#define SND_SOC_DAPM_VIRT_MUX(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_virt_mux, .name = wname, .reg = wreg, .shift = wshift, \
```

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

```
        .invert = winvert, .kcontrol_news = wcontrols, .num_kcontrols = 1}
#define SND_SOC_DAPM_VALUE_MUX(wname, wreg, wshift, winvert, wcontrols) \
{       .id = snd_soc_dapm_value_mux, .name = wname, .reg = wreg, \
        .shift = wshift, .invert = winvert, .kcontrol_news = wcontrols, \
        .num_kcontrols = 1}
```

可以看出，这些widget的reg和shift字段是需要赋值的，说明这些widget是有相应的电源控制寄存器的，DAPM框架在扫描和更新音频路径时，会利用这些寄存器来控制widget的电源状态，使得它们的供电状态是按需分配的，需要的时候（在有效的音频路径上）上电，不需要的时候（不再有效的音频路径上）下电。这些widget需要完成和之前介绍的mixer、mux等控件同样的功能，实际上，这是通过它们包含的kcontrol控件来完成的，这些kcontrol我们需要在定义widget前先定义好，然后通过wcontrols和num_kcontrols参数传递给这些辅助定义宏。
如果需要自定义这些widget的dapm事件处理回调函数，也可以使用下面这些带“_E”后缀的版本：

- SND_SOC_DAPM_PGA_E
- SND_SOC_DAPM_OUT_DRV_E
- SND_SOC_DAPM_MIXER_E
- SND_SOC_DAPM_MIXER_NAMED_CTL_E
- SND_SOC_DAPM_SWITCH_E
- SND_SOC_DAPM_MUX_E
- SND_SOC_DAPM_VIRT_MUX_E

音频数据流（stream）域widget的定义

这些widget主要包含音频输入/输出接口，ADC/DAC等等：

```cpp
#define SND_SOC_DAPM_AIF_IN(wname, stname, wslot, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_aif_in, .name = wname, .sname = stname, \
        .reg = wreg, .shift = wshift, .invert = winvert }
#define SND_SOC_DAPM_AIF_IN_E(wname, stname, wslot, wreg, wshift, winvert, \
                              wevent, wflags)                           \
{       .id = snd_soc_dapm_aif_in, .name = wname, .sname = stname, \
        .reg = wreg, .shift = wshift, .invert = winvert, \
        .event = wevent, .event_flags = wflags }
#define SND_SOC_DAPM_AIF_OUT(wname, stname, wslot, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_aif_out, .name = wname, .sname = stname, \
        .reg = wreg, .shift = wshift, .invert = winvert }
#define SND_SOC_DAPM_AIF_OUT_E(wname, stname, wslot, wreg, wshift, winvert, \
                               wevent, wflags)                          \
{       .id = snd_soc_dapm_aif_out, .name = wname, .sname = stname, \
        .reg = wreg, .shift = wshift, .invert = winvert, \
        .event = wevent, .event_flags = wflags }
#define SND_SOC_DAPM_DAC(wname, stname, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_dac, .name = wname, .sname = stname, .reg = wreg, \
        .shift = wshift, .invert = winvert}
#define SND_SOC_DAPM_DAC_E(wname, stname, wreg, wshift, winvert, \
                           wevent, wflags)                              \
{       .id = snd_soc_dapm_dac, .name = wname, .sname = stname, .reg = wreg, \
        .shift = wshift, .invert = winvert, \
        .event = wevent, .event_flags = wflags}
#define SND_SOC_DAPM_ADC(wname, stname, wreg, wshift, winvert) \
{       .id = snd_soc_dapm_adc, .name = wname, .sname = stname, .reg = wreg, \
        .shift = wshift, .invert = winvert}
#define SND_SOC_DAPM_ADC_E(wname, stname, wreg, wshift, winvert, \
                           wevent, wflags)                              \
{       .id = snd_soc_dapm_adc, .name = wname, .sname = stname, .reg = wreg, \
        .shift = wshift, .invert = winvert, \
        .event = wevent, .event_flags = wflags}
#define SND_SOC_DAPM_CLOCK_SUPPLY(wname) \
{       .id = snd_soc_dapm_clock_supply, .name = wname, \
        .reg = SND_SOC_NOPM, .event = dapm_clock_event, \
        .event_flags = SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD }
```

除了上面这些widget，还有另外三种widget没有提供显示的定义方法，它们的种类id分别是：

- snd_soc_dapm_dai_in
-
- snd_soc_dapm_dai_out
- snd_soc_dapm_dai_link

还记得我们在Linux ALSA声卡驱动之七：ASoC架构中的Codec中的snd_soc_dai结构吗？每个codec有多个dai，

而cpu（通常就是指某个soc cpu芯片）也会有多个dai，dai注册时，dapm系统会为每个dai创建一个snd_soc_dapm_dai_in或snd_soc_dapm_dai_out类型的widget，通常，这两种widget会和codec中具有相同的stream name的widget进行连接。另外一种情况，当系统中具有多个音频处理器（比如多个codec）时，他们之间可能会通过某两个dai进行连接，当machine驱动确认有这种配置时（通过判断dai_links结构中的param字段），会为他们建立一个dai link把他们绑定在一起，因为有连接关系，两个音频处理器之间的widget的电源状态就可以互相传递。

除了还有几个通用的widget，他们的定义方法如下：

```cpp
#define SND_SOC_DAPM_REG(wid, wname, wreg, wshift, wmask, won_val, woff_val) \
{       .id = wid, .name = wname, .kcontrol_news = NULL, .num_kcontrols = 0, \
        .reg = -((wreg) + 1), .shift = wshift, .mask = wmask, \
        .on_val = won_val, .off_val = woff_val, .event = dapm_reg_event, \
        .event_flags = SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD}
#define SND_SOC_DAPM_SUPPLY(wname, wreg, wshift, winvert, wevent, wflags) \
{       .id = snd_soc_dapm_supply, .name = wname, .reg = wreg,  \
        .shift = wshift, .invert = winvert, .event = wevent, \
        .event_flags = wflags}
#define SND_SOC_DAPM_REGULATOR_SUPPLY(wname, wdelay, wflags)        \
{       .id = snd_soc_dapm_regulator_supply, .name = wname, \
        .reg = SND_SOC_NOPM, .shift = wdelay, .event = dapm_regulator_event, \
        .event_flags = SND_SOC_DAPM_PRE_PMU | SND_SOC_DAPM_POST_PMD, \
        .invert = wflags}
```

定义dapm kcontrol

上一节提到，对于音频路径上的mixer或mux类型的widget，它们包含了若干个kcontrol，这些被包含的kcontrol实际上就是我们之前讨论的mixer和mux等，dapm利用这些kcontrol完成音频路径的控制。不过，对于widget来说，它的任务还不止这些，dapm还要动态地管理这些音频路径的连结关系，以便可以根据这些连接关系来控制这些widget的电源状态，如果按照普通的方法定义这些kcontrol，是无法达到这个目的的，因此，dapm为我们提供了另外一套定义宏，由它们完成这些被widget包含的kcontrol的定义。

```cpp
#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) }
#define SOC_DAPM_SINGLE_TLV(xname, reg, shift, max, invert, tlv_array) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_volsw, \
        .access = SNDRV_CTL_ELEM_ACCESS_TLV_READ | SNDRV_CTL_ELEM_ACCESS_READWRITE,\
        .tlv.p = (tlv_array), \
        .get = snd_soc_dapm_get_volsw, .put = snd_soc_dapm_put_volsw, \
        .private_value =  SOC_SINGLE_VALUE(reg, shift, max, invert) }
#define SOC_DAPM_ENUM(xname, xenum) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_enum_double, \
        .get = snd_soc_dapm_get_enum_double, \
        .put = snd_soc_dapm_put_enum_double, \
        .private_value = (unsigned long)&xenum }
#define SOC_DAPM_ENUM_VIRT(xname, xenum)                    \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_enum_double, \
        .get = snd_soc_dapm_get_enum_virt, \
        .put = snd_soc_dapm_put_enum_virt, \
        .private_value = (unsigned long)&xenum }
#define SOC_DAPM_ENUM_EXT(xname, xenum, xget, xput) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_enum_double, \
        .get = xget, \
        .put = xput, \
        .private_value = (unsigned long)&xenum }
#define SOC_DAPM_VALUE_ENUM(xname, xenum) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname, \
        .info = snd_soc_info_enum_double, \
        .get = snd_soc_dapm_get_value_enum_double, \
        .put = snd_soc_dapm_put_value_enum_double, \
        .private_value = (unsigned long)&xenum }
#define SOC_DAPM_PIN_SWITCH(xname) \
{       .iface = SNDRV_CTL_ELEM_IFACE_MIXER, .name = xname " Switch", \
```

```
        .info = snd_soc_dapm_info_pin_switch, \
        .get = snd_soc_dapm_get_pin_switch, \
        .put = snd_soc_dapm_put_pin_switch, \
        .private_value = (unsigned long)xname }
```

可以看出，SOC_DAPM_SINGLE对应与普通控件的SOC_SINGLE，SOC_DAPM_SINGLE_TLV对应SOC_SINGLE_TLV.....，相比普通的kcontrol控件，dapm的kcontrol控件只是把info，get，put回调函数换掉了。dapm kcontrol的put回调函数不仅仅会更新控件本身的状态，他还会把这种变化传递到相邻的dapm kcontrol，相邻的dapm kcontrol又会传递这个变化到他自己相邻的dapm kcontrol，知道音频路径的末端，通过这种机制，只要改变其中一个widget的连接状态，与之相关的所有widget都会被扫描并测试一下自身是否还在有效的音频路径中，从而可以动态地改变自身的电源状态，这就是dapm的精髓所在。这里我只提一下这种概念，后续的章节会有较为详细的代码分析过程。

建立widget和route

上面介绍了一大堆的辅助宏，那么，对于一个实际的系统，我们怎么定义我们需要的widget？怎样定义widget的连接关系？下面我们还是以Wolfson公司的codec芯片WM8993为例子来了解这个过程。

第一步，利用辅助宏定义widget所需要的dapm kcontrol：

```cpp
static const struct snd_kcontrol_new left_speaker_mixer[] = {
SOC_DAPM_SINGLE("Input Switch", WM8993_SPEAKER_MIXER, 7, 1, 0),
SOC_DAPM_SINGLE("IN1LP Switch", WM8993_SPEAKER_MIXER, 5, 1, 0),
SOC_DAPM_SINGLE("Output Switch", WM8993_SPEAKER_MIXER, 3, 1, 0),
SOC_DAPM_SINGLE("DAC Switch", WM8993_SPEAKER_MIXER, 6, 1, 0),
};

static const struct snd_kcontrol_new right_speaker_mixer[] = {
SOC_DAPM_SINGLE("Input Switch", WM8993_SPEAKER_MIXER, 6, 1, 0),
SOC_DAPM_SINGLE("IN1RP Switch", WM8993_SPEAKER_MIXER, 4, 1, 0),
SOC_DAPM_SINGLE("Output Switch", WM8993_SPEAKER_MIXER, 2, 1, 0),
SOC_DAPM_SINGLE("DAC Switch", WM8993_SPEAKER_MIXER, 0, 1, 0),
};

static const char *aif_text[] = {
        "Left", "Right"
};

static const struct soc_enum aifinl_enum =
        SOC_ENUM_SINGLE(WM8993_AUDIO_INTERFACE_2, 15, 2, aif_text);


static const struct snd_kcontrol_new aifinl_mux =
        SOC_DAPM_ENUM("AIFINL Mux", aifinl_enum);


static const struct soc_enum aifinr_enum =
        SOC_ENUM_SINGLE(WM8993_AUDIO_INTERFACE_2, 14, 2, aif_text);


static const struct snd_kcontrol_new aifinr_mux =
        SOC_DAPM_ENUM("AIFINR Mux", aifinr_enum);
```

以上，我们定义了wm8993中左右声道的speaker mixer控件：left_speaker_mixer和right_speaker_mixer，同时还为左右声道各定义了一个叫做AIFINL Mux和AIFINR Mux的输入选择mux控件。

第二步，定义真正的widget，包含第一步定义好的dapm控件：

```cpp
static const struct snd_soc_dapm_widget wm8993_dapm_widgets[] = {
    ......
    SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),
    SND_SOC_DAPM_AIF_IN("AIFINR", "Playback", 1, SND_SOC_NOPM, 0, 0),
    ......
    SND_SOC_DAPM_MUX("DACL Mux", SND_SOC_NOPM, 0, 0, &aifinl_mux),
    SND_SOC_DAPM_MUX("DACR Mux", SND_SOC_NOPM, 0, 0, &aifinr_mux),

```

```
    SND_SOC_DAPM_MIXER("SPKL", WM8993_POWER_MANAGEMENT_3, 8, 0,
                       left_speaker_mixer, ARRAY_SIZE(left_speaker_mixer)),
    SND_SOC_DAPM_MIXER("SPKR", WM8993_POWER_MANAGEMENT_3, 9, 0,
                       right_speaker_mixer, ARRAY_SIZE(right_speaker_mixer)),
    ......
};
```

这一步，为左右声道各自定义了一个mux widget：DACL Mux和DACR Mux，实际的多路选择由dapm kcontrol：aifinl_mux和aifinr_mux，来完成，因为传入了SND_SOC_NOPM参数，这两个widget不具备电源属性，但是mux的切换会影响到与之相连的其它具备电源属性的电源状态。我们还为左右声道的扬声器各自定义了一个mixer widget：SPKL和SPKR，具体的mixer控制由上一步定义的left_speaker_mixer和right_speaker_mixer来完成，两个widget具备电源属性，所以，当这两个widget在一条有效的音频路径上时，dapm框架可以通过寄存器WM8993_POWER_MANAGEMENT_3的第8位和第9位控制它的电源状态。

第三步，定义这些widget的连接路径：

```
[cpp]

static const struct snd_soc_dapm_route routes[] = {
        ......

        { "DACL Mux", "Left", "AIFINL" },
        { "DACL Mux", "Right", "AIFINR" },
        { "DACR Mux", "Left", "AIFINL" },
        { "DACR Mux", "Right", "AIFINR" },

        ......

        { "SPKL", "DAC Switch", "DACL" },
        { "SPKL", NULL, "CLK_SYS" },

        { "SPKR", "DAC Switch", "DACR" },
        { "SPKR", NULL, "CLK_SYS" },
};
```

通过第一步的定义，我们知道DACL Mux和DACR Mux有两个输入引脚，分别是

- Left
- Right

而SPKL和SPKR有四个输入选择引脚，分别是：

- Input Switch
- IN1LP Switch/IN1RP Switch
- Output Switch
- DAC Switch

所以，很显然，上面的路径定义的意思就是：

- AIFINL连接到DACL Mux的Left输入脚
- AIFINR连接到DACL Mux的Right输入脚
- AIFINL连接到DACR Mux的Left输入脚
- AIFINR连接到DACR Mux的Right输入脚
- DACL连接到SPKL的DAC Switch输入脚
- DACR连接到SPKR的DAC Switch输入脚

第四步，在codec驱动的probe回调中注册这些widget和路径：

```
[cpp]

static int wm8993_probe(struct snd_soc_codec *codec)
{
        ......
        snd_soc_dapm_new_controls(dapm, wm8993_dapm_widgets,
                                  ARRAY_SIZE(wm8993_dapm_widgets));
        ......

        snd_soc_dapm_add_routes(dapm, routes, ARRAY_SIZE(routes));
        ......
```

```
10. }
```

在machine驱动中，我们可以用同样的方式定义和注册板子特有的widget和路径信息。

更多 0

上一篇：ALSA声卡驱动中的DAPM详解之二：widget-具备路径和电源管理信息的kcontrol

下一篇：ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route

查看评论

4楼 elliepfsang 6天前 15:41发表

大侠，你好！
这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次，也写到了dapm kcontrol与普通control的区别，但有个疑问是在定义普通control和dapm control时，如何区分哪些用普通control来定义，哪些用dapm control来定义，您能举个例子帮我解释一下吗？

Re: DroidPhone 5天前 19:31发表

回复u012389631：和电源管理和音频路径相关的control需要定义为dapm control（比如电源或时钟开关，混音器，多路器等），否则定义为普通的control（例如：音量，音效，某些特殊的特性等）。

3楼 elliepfsang 2013-12-16 18:56发表

大侠
---
struct snd_soc_dapm_widget wm8993_dapm_widgets[] = {
.....
SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_IN("AIFINR", "Playback", 1, SND_SOC_NOPM, 0, 0),
---
中的"AIFINL", "Playback"等字符串的设置是参考哪些而设的啊，关于这点满疑惑的

Re: DroidPhone 2013-12-16 21:04发表

回复u012389631：这里的playback，是指stream name，请参看：ALSA声卡驱动中的DAPM详解之七：dapm事件机制（dapm event）（http://blog.csdn.net/droidphone/article/details/14548631）
：“连接dai widget和stream widget”一节的内容

2楼 hustljh 2013-12-11 20:16发表

kcontrol控件的get和put会调用snd_soc_codec_driver的read、write函数，wm8994.c中好像没有实现这两个函数，请教下博主，这是什么情况?是在其他地方实现了还是不需要？

Re: DroidPhone 2013-12-11 20:37发表

回复hustljh：wm8994_codec_probe-->snd_soc_codec_set_cache_io-->codec->write = hw_write;

1楼 Color_Fly 2013-11-01 10:41发表

版主你好，这里有个问题请教一下：
static const struct soc_enum aifinl_enum =
SOC_ENUM_SINGLE(WM8993_AUDIO_INTERFACE_2, 15, 2, aif_text);

对于这句：我查看了WM8993_AUDIO_INTERFACE_2寄存器的15位有两个选择（0： Left DAC outputs left interface data，1： Left DAC outputs right interface data），这里为什么在宏中的xmax项却填写了2，我的理解是最大值是1，不知道这个地方怎么解释，请版主给科普下，thankyou！！！

Re: DroidPhone 2013-11-01 17:09发表

回复u012498713：我也注意到这点了，应该是内核的代码上的一个bug。

Re: Color_Fly 2013-11-01 19:47发表

回复DroidPhone：我查看了好几个，都是这种情况，不明白！！！！

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

专区推荐内容

HTML5应用性能调优工具WAP...
Android应用开发
NDK 安卓应用移植方法
Struts2 高危漏洞修复方案
HTML5 经典小游戏之坦克
新用户编译器基本用法

更多招聘职位 **我公司职位也要出现在这里**

【博彦科技（上海）有限公司】oracle数据库开发
【长沙中科院文化创意与科技产业研究院】Java研发工程师
【哈票网络科技（北京）有限公司】Java研发组组长
【万维嘉信（山东）电子商务有限公司】高级运维经理
【广州友魄信息科技有限公司】Asp.net（C#）开发工程师

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382370次
积分： 3435分
排名： 第2116名

原创： 46篇 转载： 0篇
译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route

分类： Linux音频子系统 2013-11-01 22:41 453人阅读 评论(0) 收藏 举报

linux widget audio driver dapm alsa

前几篇文章我们从dapm的数据结构入手，了解了代表音频控件的widget，代表连接路径的route以及用于连接两个widget的path。之前都是一些概念的讲解以及对数据结构中各个字段的说明，从本章开始，我们要从代码入手，分析dapm的详细工作原理：

- 如何注册widget
- 如何连接两个widget
- 一个widget的状态裱画如何传递到整个音频路径中

/***/
声明：本博内容均由http://blog.csdn.net/droidphone原创，转载请注明出处，谢谢！
/***/

dapm context

在讨论widget的注册之前，我们先了解另一个概念：dapm context，直译过来的意思是dapm上下文，这个好像不好理解，其实我们可以这么理解：dapm把整个音频系统，按照功能和偏置电压级别，划分为若干个电源域，每个域包含各自的widget，每个域中的所有widget通常都处于同一个偏置电压级别上，而一个电源域就是一个dapm context，通常会有以下几种dapm context：

- 属于codec中的widget位于一个dapm context中
- 属于platform的widget位于一个dapm context中
- 属于整个声卡的widget位于一个dapm context中

对于音频系统的硬件来说，通常要提供合适的偏置电压才能正常地工作，有了dapm context这种组织方式，我们可以方便地对同一组widget进行统一的偏置电压管理，ASoc用snd_soc_dapm_context结构来表示一个dapm context：

```cpp
struct snd_soc_dapm_context {
        enum snd_soc_bias_level bias_level;
        enum snd_soc_bias_level suspend_bias_level;
        struct delayed_work delayed_work;
        unsigned int idle_bias_off:1; /* Use BIAS_OFF instead of STANDBY */

        struct snd_soc_dapm_update *update;
```

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

```
08.
09.         void (*seq_notifier)(struct snd_soc_dapm_context *,
10.                              enum snd_soc_dapm_type, int);
11.
12.         struct device *dev; /* from parent - for debug */
13.         struct snd_soc_codec *codec; /* parent codec */
14.         struct snd_soc_platform *platform; /* parent platform */
15.         struct snd_soc_card *card; /* parent card */
16.
17.         /* used during DAPM updates */
18.         enum snd_soc_bias_level target_bias_level;
19.         struct list_head list;
20.
21.         int (*stream_event)(struct snd_soc_dapm_context *dapm, int event);
22.
23. #ifdef CONFIG_DEBUG_FS
24.         struct dentry *debugfs_dapm;
25. #endif
26. };
```

snd_soc_bias_level的取值范围是以下几种：

- SND_SOC_BIAS_OFF
- SND_SOC_BIAS_STANDBY
- SND_SOC_BIAS_PREPARE
- SND_SOC_BIAS_ON

snd_soc_dapm_context被内嵌到代表codec、platform、card、dai的结构体中：

```
[cpp]

01. struct snd_soc_codec {
02.         ......
03.         /* dapm */
04.         struct snd_soc_dapm_context dapm;
05.         ......
06. };
07.
08. struct snd_soc_platform {
09.         ......
10.         /* dapm */
11.         struct snd_soc_dapm_context dapm;
12.         ......
13. };
14.
15. struct snd_soc_card {
16.         ......
17.         /* dapm */
18.         struct snd_soc_dapm_context dapm;
19.         ......
20. };
21. :
22. struct snd_soc_dai {
23.         ......
24.         /* dapm */
25.         struct snd_soc_dapm_widget *playback_widget;
26.         struct snd_soc_dapm_widget *capture_widget;
27.         struct snd_soc_dapm_context dapm;
28.         ......
29. };
```

代表widget结构snd_soc_dapm_widget中，有一个snd_soc_dapm_context结构指针，指向所属的codec、platform、card、或dai的dapm结构。同时，所有的dapm结构，通过它的list字段，链接到代表声卡的snd_soc_card结构的dapm_list链表头字段。

创建和注册widget

我们已经知道，一个widget用snd_soc_dapm_widget结构体来描述，通常，我们会根据音频硬件的组成，分别在声卡的codec驱动、platform驱动和machine驱动中定义一组widget，这些widget用数组进行组织，我们一般会使用dapm框架提供的大量的辅助宏来定义这些widget数组，辅助宏的说明请参考前一偏文章：ALSA声卡驱动中的DAPM详解之三：如何定义各种widget。

codec驱动中注册　我们知道，我们会通过ASoc提供的api函数snd_soc_register_codec来注册一个codec驱

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

动，该函数的第二个参数是一个snd_soc_codec_driver结构指针，这个snd_soc_codec_driver结构需要我们在codec驱动中显式地进行定义，其中有几个与dapm框架有关的字段：

```
[cpp]

struct snd_soc_codec_driver {
        ......
        /* Default control and setup, added after probe() is run */
        const struct snd_kcontrol_new *controls;
        int num_controls;
        const struct snd_soc_dapm_widget *dapm_widgets;
        int num_dapm_widgets;
        const struct snd_soc_dapm_route *dapm_routes;
        int num_dapm_routes;
        ......
}
```

我们只要把我们定义好的snd_soc_dapm_widget结构数组的地址和widget的数量赋值到dapm_widgets和num_dapm_widgets字段即可，这样，经过snd_soc_register_codec注册codec后，在machine驱动匹配上该codec时，系统会判断这两个字段是否被赋值，如果有，它会调佣dapm框架提供的api来创建和注册widget，注意这里我说还要创建这个词，你可能比较奇怪，既然代表widget的snd_soc_dapm_widget结构数组已经在codec驱动中定义好了，为什么还要在创建？事实上，我们在codec驱动中定义的widget数组只是作为一个模板，dapm框架会根据该模板重新申请内存并初始化各个widget。我们看看实际的例子可能是这样的：

```
[cpp]

static const struct snd_soc_dapm_widget wm8993_dapm_widgets[] = {
        ......
        SND_SOC_DAPM_SUPPLY("VMID", SND_SOC_NOPM, 0, 0, NULL, 0),
        SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),
        SND_SOC_DAPM_AIF_IN("AIFINR", "Playback", 1, SND_SOC_NOPM, 0, 0),
        ......
};

static struct snd_soc_codec_driver soc_codec_dev_wm8993 = {
        .probe =        codec_xxx_probe,
        ......
        .dapm_widgets =       &wm8993_dapm_widgets[0],
        .num_dapm_widgets =      ARRAY_SIZE(wm8993_dapm_widgets),
        ......
};

static int codec_wm8993_i2c_probe(struct i2c_client *i2c,
                              const struct i2c_device_id *id)
{
        ......
        ret = snd_soc_register_codec(&i2c->dev,
                        &soc_codec_dev_wm8993, &wm8993_dai, 1);
        ......
}
```

上面这种注册方法有个缺点，有时候我们为了代码的清晰，可能会根据功能把不同的widget定义成多个数组，但是snd_soc_codec_driver中只有一个dapm_widgets字段，无法设定多个widget数组，这时候，我们需要主动在codec的probe回调中调用dapm框架提供的api来创建这些widget：

```
[cpp]

static int wm8993_probe(struct snd_soc_codec *codec)
{
        ......
        snd_soc_dapm_new_controls(dapm, wm8993_dapm_widgets,
                                  ARRAY_SIZE(wm8993_dapm_widgets));
        ......
}
```

实际上，对于第一种方法，snd_soc_register_codec内部其实也是调用snd_soc_dapm_new_controls来完成的。后面会有关于这个函数的详细分析。

platform驱动中注册　和codec驱动一样，我们会通过ASoc提供的api函数snd_soc_register_platform来注册一个platform驱动，该函数的第二个参数是一个snd_soc_platform_driver结构指针，snd_soc_platform_driver结构中同样也包含了与dapm相关的字段：

```cpp
struct snd_soc_platform_driver {
        ......
        /* Default control and setup, added after probe() is run */
        const struct snd_kcontrol_new *controls;
        int num_controls;
        const struct snd_soc_dapm_widget *dapm_widgets;
        int num_dapm_widgets;
        const struct snd_soc_dapm_route *dapm_routes;
        int num_dapm_routes;
        ......
}
```

要注册platform级别的widget，和codec驱动一样，只要把定义好的widget数组赋值给dapm_widgets和num_dapm_widgets字段即可，snd_soc_register_platform函数注册paltform后，当machine驱动匹配上该platform时，系统会自动完成创建和注册的工作。同理，我们也可以在platform驱动的probe回调函数中主动使用snd_soc_dapm_new_controls来完成widget的创建工作。具体的代码和codec驱动是类似的，这里就不贴了。

machine驱动中注册　有些widget可能不是位于codec中，例如一个独立的耳机放大器，或者是喇叭功放等，这种widget通常需要在machine驱动中注册，通常他们的dapm context也从属于声卡（snd_soc_card）域。做法依然和codec驱动类似，通过代表声卡的snd_soc_card结构中的几个dapm字段完成：

```cpp
struct snd_soc_card {
        ......
        /*
         * Card-specific routes and widgets.
         */
        const struct snd_soc_dapm_widget *dapm_widgets;
        int num_dapm_widgets;
        const struct snd_soc_dapm_route *dapm_routes;
        int num_dapm_routes;
        bool fully_routed;
        ......
}
```

只要把定义好的widget数组和数量赋值给dapm_widgets指针和num_dapm_widgets即可，注册声卡使用的api：snd_soc_register_card()，也会通过snd_soc_dapm_new_controls来完成widget的创建工作。

注册音频路径

系统中注册的各种widget需要互相连接在一起才能协调工作，连接关系通过snd_soc_dapm_route结构来定义，关于如何用snd_soc_dapm_route结构来定义路径信息，请参考：ALSA声卡驱动中的DAPM详解之三：如何定义各种widget中的"建立widget和route"一节的内容。通常，所有的路径信息会用一个snd_soc_dapm_route结构数组来定义。和widget一样，路径信息也分别存在与codec驱动，machine驱动和platform驱动中，我们一样有两种方式来注册音频路径信息：

- 通过snd_soc_codec_driver/snd_soc_platform_driver/snd_soc_card结构中的dapm_routes和num_dapm_routes字段；
- 在codec、platform的的probe回调中主动注册音频路径，machine驱动中则通过snd_soc_dai_link结构的init回调函数来注册音频路径；

两种方法最终都是通过调用snd_soc_dapm_add_routes函数来完成音频路径的注册工作的。以下的代码片段是omap的pandora板子的machine驱动，使用第二种方法注册路径信息：

```cpp
static const struct snd_soc_dapm_widget omap3pandora_in_dapm_widgets[] = {
        SND_SOC_DAPM_MIC("Mic (internal)", NULL),
        SND_SOC_DAPM_MIC("Mic (external)", NULL),
        SND_SOC_DAPM_LINE("Line In", NULL),
};

static const struct snd_soc_dapm_route omap3pandora_out_map[] = {
        {"PCM DAC", NULL, "APLL Enable"},
        {"Headphone Amplifier", NULL, "PCM DAC"},
        {"Line Out", NULL, "PCM DAC"},
        {"Headphone Jack", NULL, "Headphone Amplifier"},
```

```
};

static const struct snd_soc_dapm_route omap3pandora_in_map[] = {
        {"AUXL", NULL, "Line In"},
        {"AUXR", NULL, "Line In"},

        {"MAINMIC", NULL, "Mic (internal)"},
        {"Mic (internal)", NULL, "Mic Bias 1"},

        {"SUBMIC", NULL, "Mic (external)"},
        {"Mic (external)", NULL, "Mic Bias 2"},
};
static int omap3pandora_out_init(struct snd_soc_pcm_runtime *rtd)
{
        struct snd_soc_codec *codec = rtd->codec;
        struct snd_soc_dapm_context *dapm = &codec->dapm;
        int ret;

        /* All TWL4030 output pins are floating */
        snd_soc_dapm_nc_pin(dapm, "EARPIECE");
        ......
        //注册kcontrol控件
        ret = snd_soc_dapm_new_controls(dapm, omap3pandora_out_dapm_widgets,
                                ARRAY_SIZE(omap3pandora_out_dapm_widgets));
        if (ret < 0)
                return ret;
        //注册machine的音频路径
        return snd_soc_dapm_add_routes(dapm, omap3pandora_out_map,
                ARRAY_SIZE(omap3pandora_out_map));
}

static int omap3pandora_in_init(struct snd_soc_pcm_runtime *rtd)
{
        struct snd_soc_codec *codec = rtd->codec;
        struct snd_soc_dapm_context *dapm = &codec->dapm;
        int ret;

        /* Not comnnected */
        snd_soc_dapm_nc_pin(dapm, "HSMIC");
        ......
        //注册kcontrol控件
        ret = snd_soc_dapm_new_controls(dapm, omap3pandora_in_dapm_widgets,
                                ARRAY_SIZE(omap3pandora_in_dapm_widgets));
        if (ret < 0)
                return ret;
        //注册machine音频路径
        return snd_soc_dapm_add_routes(dapm, omap3pandora_in_map,
                ARRAY_SIZE(omap3pandora_in_map));
}

/* Digital audio interface glue - connects codec <--> CPU */
static struct snd_soc_dai_link omap3pandora_dai[] = {
        {
                .name = "PCM1773",
                ......
                .init = omap3pandora_out_init,
        }, {
                .name = "TWL4030",
                .stream_name = "Line/Mic In",
                ......
                .init = omap3pandora_in_init,
        }
};
```

dai widget

上面几节的内容介绍了codec、platform以及machine级别的widget和route的注册方法，在dapm框架中，还有另外一种widget，它代表了一个dai（数字音频接口），关于dai的描述，请参考：Linux ALSA声卡驱动之七：ASoC架构中的Codec。dai按所在的位置，又分为cpu dai和codec dai，在硬件上，通常一个cpu dai会连接一个codec dai，而在machine驱动中，我们要在snd_soc_card结构中指定一个叫做snd_soc_dai_link的结构，该结构定义了声卡使用哪一个cpu dai和codec dai进行连接。在Asoc中，一个dai用snd_soc_dai结构来表述，其中有几个字段和dapm框架有关：

```cpp
struct snd_soc_dai {
        ......
        struct snd_soc_dapm_widget *playback_widget;
        struct snd_soc_dapm_widget *capture_widget;
        struct snd_soc_dapm_context dapm;
        ......
}
```

dai由codec驱动和平台代码中的iis或pcm接口驱动注册，machine驱动负责找到snd_soc_dai_link中指定的一对cpu/codec dai，并把它们进行绑定。不管是cpu dai还是codec dai，通常会同时传输播放和录音的音频流的能力，所以我们可以看到，snd_soc_dai中有两个widget指针，分别代表播放流和录音流。这两个dai widget是何时创建的呢？下面我们逐一进行分析。

codec dai widget

首先，codec驱动在注册codec时，会传入该codec所支持的dai个数和记录dai信息的snd_soc_dai_driver结构指针：

```cpp
static struct snd_soc_dai_driver wm8993_dai = {
        .name = "wm8993-hifi",
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8993_RATES,
                .formats = WM8993_FORMATS,
                .sig_bits = 24,
        },
        .capture = {
                 .stream_name = "Capture",
                 .channels_min = 1,
                 .channels_max = 2,
                 .rates = WM8993_RATES,
                 .formats = WM8993_FORMATS,
                 .sig_bits = 24,
         },
        .ops = &wm8993_ops,
        .symmetric_rates = 1,
};

static int wm8993_i2c_probe(struct i2c_client *i2c,
                            const struct i2c_device_id *id)
{
        ......
        ret = snd_soc_register_codec(&i2c->dev,
                        &soc_codec_dev_wm8993, &wm8993_dai, 1);
        ......
}
```

这回使得ASoc把codec的dai注册到系统中，并把这些dai都挂在全局链表变量dai_list中，然后，在codec被machine驱动匹配后，soc_probe_codec函数会被调用，他会通过全局链表变量dai_list查找所有属于该codec的dai，调用snd_soc_dapm_new_dai_widgets函数来生成该dai的播放流widget和录音流widget：

```cpp
static int soc_probe_codec(struct snd_soc_card *card,
                           struct snd_soc_codec *codec)
{
        ......
        /* Create DAPM widgets for each DAI stream */
        list_for_each_entry(dai, &dai_list, list) {
                if (dai->dev != codec->dev)
                        continue;

                snd_soc_dapm_new_dai_widgets(&codec->dapm, dai);
        }
        ......
}
```

我们看看snd_soc_dapm_new_dai_widgets的代码：

```cpp
int snd_soc_dapm_new_dai_widgets(struct snd_soc_dapm_context *dapm,
                                 struct snd_soc_dai *dai)
{
        struct snd_soc_dapm_widget template;
        struct snd_soc_dapm_widget *w;

        WARN_ON(dapm->dev != dai->dev);

        memset(&template, 0, sizeof(template));
        template.reg = SND_SOC_NOPM;
        // 创建播放 dai widget
        if (dai->driver->playback.stream_name) {
                template.id = snd_soc_dapm_dai_in;
                template.name = dai->driver->playback.stream_name;
                template.sname = dai->driver->playback.stream_name;

                w = snd_soc_dapm_new_control(dapm, &template);

                w->priv = dai;
                dai->playback_widget = w;
        }
        // 创建录音 dai widget
        if (dai->driver->capture.stream_name) {
                template.id = snd_soc_dapm_dai_out;
                template.name = dai->driver->capture.stream_name;
                template.sname = dai->driver->capture.stream_name;

                w = snd_soc_dapm_new_control(dapm, &template);

                w->priv = dai;
                dai->capture_widget = w;
        }

        return 0;
}
```

分别为Playback和Capture创建了一个widget，widget的priv字段指向了该dai，这样通过widget就可以找到相应的dai，并且widget的名字就是snd_soc_dai_driver结构的stream_name。

cpu dai widget

这里顺便说一个小意外，昨天晚上手贱，执行了一下git pull，版本升级到了3.12 rc7，结果发现ASoc的代码有所变化，于是稍稍纠结了一下，用新的代码继续还是恢复之前的3.10 rc5?经过查看了一些变化后，发现还是新的版本改进得更合理，现在决定，后面的内容都是基于3.12 rc7了。如果大家发现后面贴的代码和之前贴的有差异的地方，自己比较一下这两个版本的代码吧！

回到cpu dai，以前的内核版本由驱动通过snd_soc_register_dais注册，新的版本中，这个函数变为了soc-core的内部函数，驱动改为使用snd_soc_register_component注册，snd_soc_register_component函数再通过调用snd_soc_register_dai/snd_soc_register_dais来完成实际的注册工作。和codec dai widget一样，cpu dai widget也发生在machine驱动匹配上相应的platform驱动之后，soc_probe_platform会被调用，在soc_probe_platform函数中，通过比较dai->dev和platform->dev，挑选出属于该platform的dai，然后通过snd_soc_dapm_new_dai_widgets为cpu dai创建相应的widget：

```cpp
static int soc_probe_platform(struct snd_soc_card *card,
                              struct snd_soc_platform *platform)
{
        int ret = 0;
        const struct snd_soc_platform_driver *driver = platform->driver;
        struct snd_soc_dai *dai;

        ......

        if (driver->dapm_widgets)
                snd_soc_dapm_new_controls(&platform->dapm,
                        driver->dapm_widgets, driver->num_dapm_widgets);

        /* Create DAPM widgets for each DAI stream */
        list_for_each_entry(dai, &dai_list, list) {
                if (dai->dev != platform->dev)
                        continue;

                snd_soc_dapm_new_dai_widgets(&platform->dapm, dai);
```

```
        }

        platform->dapm.idle_bias_off = 1;

        ......

        if (driver->controls)
                snd_soc_add_platform_controls(platform, driver->controls,
                                    driver->num_controls);
        if (driver->dapm_routes)
                snd_soc_dapm_add_routes(&platform->dapm, driver->dapm_routes,
                                        driver->num_dapm_routes);
        ......

        return 0;
}
```

从上面的代码我们也可以看出，在上面的"创建和注册widget"一节提到的第一种方法，即通过给snd_soc_platform_driver结构的dapm_widgets和num_dapm_widgets字段赋值，ASoc会自动为我们创建所需的widget，真正执行创建工作就在上面所列的soc_probe_platform函数中完成的，普通的kcontrol和音频路径也是一样的原理。反推回来，codec的widget也是一样的，在soc_probe_codec中会做同样的事情，只是我上面贴出来soc_probe_codec的代码里没有贴出来，有兴趣的读者自己查看一下它的代码即可。
花了这么多篇幅来讲解dai widget，好像现在看来它还没有什么用处。嗯，不要着急，实际上dai widget是一条完整dapm音频路径的重要元素，没有她，我们无法完成dapm的动态电源管理工作，因为它是音频流和其他widget的纽带，细节我们要留到下一篇文章中来阐述了。

端点widget

一条完整的dapm音频路径，必然有起点和终点，我们把位于这些起点和终点的widget称之为端点widget。以下这些类型的widget可以成为端点widget：

codec的输入输出引脚：

- snd_soc_dapm_output
- snd_soc_dapm_input

外接的音频设备：

- snd_soc_dapm_hp
- snd_soc_dapm_spk
- snd_soc_dapm_line

音频流（stream domain）：

- snd_soc_dapm_adc
- snd_soc_dapm_dac
- snd_soc_dapm_aif_out
- snd_soc_dapm_aif_in
- snd_soc_dapm_dai_out
- snd_soc_dapm_dai_in

电源、时钟和其它：

- snd_soc_dapm_supply
- snd_soc_dapm_regulator_supply
- snd_soc_dapm_clock_supply
- snd_soc_dapm_kcontrol

当声卡上的其中一个widget的状态发生改变时，从这个widget开始，dapm框架会向前和向后遍历路径上的所有widget，判断每个widget的状态是否需要跟着变更，到达这些端点widget就会认为它是一条完整音频路径的开始和结束，从而结束一次扫描动作。至于代码的分析，先让我歇一会.....，我会在后面的文章中讨论。

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

专区推荐内容

关于HTML怎样用图片做背景
Android 游戏教程:让人物...
HTML5应用性能调优工具WAP...
Android应用开发
NDK 安卓应用移植方法
Struts2 高危漏洞修复方案

更多招聘职位 **我公司职位也要出现在这里**

【博彦科技（上海）有限公司】oracle数据库开发
【长沙中科院文化创意与科技产业研究院】Java研发工程师
【哈票网络科技（北京）有限公司】Java研发组组长
【万维嘉信（山东）电子商务有限公司】高级运维经理
【广州友魄信息科技有限公司】Asp.net（C#）开发工程师

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382367次

积分： 3435分

排名： 第2116名

原创： 46篇 转载： 0篇

译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之五：建立widget之间的连接关系

分类： Linux音频子系统 2013-11-04 21:25 450人阅读 评论(3) 收藏 举报

linux audio driver widget alsa dapm

目录(?) [-]

前面我们主要着重于codec、platform、machine驱动程序中如何使用和建立dapm所需要的widget，route，这些是音频驱动开发人员必须要了解的内容，经过前几章的介绍，我们应该知道如何在alsa音频驱动的3大部分（codec、platform、machine）中，按照所使用的音频硬件结构，定义出相应的widget，kcontrol，以及必要的音频路径，而在本章中，我们将会深入dapm的核心部分，看看各个widget之间是如何建立连接关系，形成一条完整的音频路径。

前面我们已经简单地介绍过，驱动程序需要使用以下api函数创建widget：

- snd_soc_dapm_new_controls

实际上，这个函数只是创建widget的第一步，它为每个widget分配内存，初始化必要的字段，然后把这些widget挂在代表声卡的snd_soc_card的widgets链表字段中。要使widget之间具备连接能力，我们还需要第二个函数：

- snd_soc_dapm_new_widgets

这个函数会根据widget的信息，创建widget所需要的dapm kcontrol，这些dapm kcontol的状态变化，代表着音频路径的变化，从而影响着各个widget的电源状态。看到函数的名称可能会迷惑一下，实际上，snd_soc_dapm_new_controls的作用更多地是创建widget，而snd_soc_dapm_new_widget的作用则更多地是创建widget所包含的kcontrol，所以在我看来，这两个函数名称应该换过来叫更好！下面我们分别介绍一下这两个函数是如何工作的。

创建widget：snd_soc_dapm_new_controls

snd_soc_dapm_new_controls函数完成widget的创建工作，并把这些创建好的widget注册在声卡的widgets链表中，我们看看他的定义：

```cpp
int snd_soc_dapm_new_controls(struct snd_soc_dapm_context *dapm,
        const struct snd_soc_dapm_widget *widget,
        int num)
{
        ......
        for (i = 0; i < num; i++) {
                w = snd_soc_dapm_new_control(dapm, widget);
```

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

```
08.                 if (!w) {
09.                         dev_err(dapm->dev,
10.                                 "ASoC: Failed to create DAPM control %s\n",
11.                                 widget->name);
12.                         ret = -ENOMEM;
13.                         break;
14.                 }
15.                 widget++;
16.         }
17.         ......
18.         return ret;
19. }
```

该函数只是简单的一个循环，为传入的widget模板数组依次调用snd_soc_dapm_new_control函数，实际的工作由snd_soc_dapm_new_control完成，继续进入该函数，看看它做了那些工作。

我们之前已经说过，驱动中定义的snd_soc_dapm_widget数组，只是作为一个模板，所以，snd_soc_dapm_new_control所做的第一件事，就是为该widget重新分配内存，并把模板的内容拷贝过来：

```
[cpp]

01. static struct snd_soc_dapm_widget *
02. snd_soc_dapm_new_control(struct snd_soc_dapm_context *dapm,
03.                          const struct snd_soc_dapm_widget *widget)
04. {
05.         struct snd_soc_dapm_widget *w;
06.         int ret;
07.
08.         if ((w = dapm_cnew_widget(widget)) == NULL)
09.                 return NULL;
```

由dapm_cnew_widget完成内存申请和拷贝模板的动作。接下来，根据widget的类型做不同的处理：

```
[cpp]

01.         switch (w->id) {
02.         case snd_soc_dapm_regulator_supply:
03.                 w->regulator = devm_regulator_get(dapm->dev, w->name);
04.                 ......
05.
06.                 if (w->on_val & SND_SOC_DAPM_REGULATOR_BYPASS) {
07.                         ret = regulator_allow_bypass(w->regulator, true);
08.                         ......
09.                 }
10.                 break;
11.         case snd_soc_dapm_clock_supply:
12. #ifdef CONFIG_CLKDEV_LOOKUP
13.                 w->clk = devm_clk_get(dapm->dev, w->name);
14.                 ......
15. #else
16.                 return NULL;
17. #endif
18.                 break;
19.         default:
20.                 break;
21.         }
```

对于snd_soc_dapm_regulator_supply类型的widget，根据widget的名称获取对应的regulator结构，对于snd_soc_dapm_clock_supply类型的widget，根据widget的名称，获取对应的clock结构。接下来，根据需要，在widget的名称前加入必要的前缀：

```
[cpp]

01. if (dapm->codec && dapm->codec->name_prefix)
02.         w->name = kasprintf(GFP_KERNEL, "%s %s",
03.                 dapm->codec->name_prefix, widget->name);
04. else
05.         w->name = kasprintf(GFP_KERNEL, "%s", widget->name);
```

然后，为不同类型的widget设置合适的power_check电源状态回调函数，widget类型和对应的power_check回调函数设置如下表所示：

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

widget的power_check回调函数

widget类型	**power_check回调函数**
mixer类： snd_soc_dapm_switch snd_soc_dapm_mixer snd_soc_dapm_mixer_named_ctl	dapm_generic_check_power
mux类： snd_soc_dapm_mux snd_soc_dapm_mux snd_soc_dapm_mux	dapm_generic_check_power
snd_soc_dapm_dai_out	dapm_adc_check_power
snd_soc_dapm_dai_in	dapm_dac_check_power
端点类： snd_soc_dapm_adc snd_soc_dapm_aif_out snd_soc_dapm_dac snd_soc_dapm_aif_in snd_soc_dapm_pga snd_soc_dapm_out_drv snd_soc_dapm_input snd_soc_dapm_output snd_soc_dapm_micbias snd_soc_dapm_spk snd_soc_dapm_hp snd_soc_dapm_mic snd_soc_dapm_line snd_soc_dapm_dai_link	dapm_generic_check_power
电源/时钟/影子widget： snd_soc_dapm_supply snd_soc_dapm_regulator_supply snd_soc_dapm_clock_supply snd_soc_dapm_kcontrol	dapm_supply_check_power
其它类型	dapm_always_on_check_power

当音频路径发生变化时，power_check回调会被调用，用于检查该widget的电源状态是否需要更新。

power_check设置完成后，需要设置widget所属的codec、platform和dapm context，几个用于音频路径的链表也需要初始化，然后，把该widget加入到声卡的widgets链表中：

```
[cpp]

01.  w->dapm = dapm;
02.  w->codec = dapm->codec;
03.  w->platform = dapm->platform;
04.  INIT_LIST_HEAD(&w->sources);
05.  INIT_LIST_HEAD(&w->sinks);
06.  INIT_LIST_HEAD(&w->list);
07.  INIT_LIST_HEAD(&w->dirty);
08.  list_add(&w->list, &dapm->card->widgets);
```

几个链表的作用如下：

- sources　用于链接所有连接到该widget输入端的snd_soc_path结构
- sinks　用于链接所有连接到该widget输出端的snd_soc_path结构
- list　用于链接到声卡的widgets链表
- dirty　用于链接到声卡的dapm_dirty链表

最后，把widget设置为connect状态：

```
[cpp]

01.  /* machine layer set ups unconnected pins and insertions */
```

```
02.     w->connected = 1;
03.     return w;
```

connected字段代表着引脚的连接状态，目前，只有以下这些widget使用connected字段：

- snd_soc_dapm_output
- snd_soc_dapm_input
- snd_soc_dapm_hp
- snd_soc_dapm_spk
- snd_soc_dapm_line
- snd_soc_dapm_vmid
- snd_soc_dapm_mic
- snd_soc_dapm_siggen

驱动程序可以使用以下这些api来设置引脚的连接状态：

- snd_soc_dapm_enable_pin
- snd_soc_dapm_force_enable_pin
- snd_soc_dapm_disable_pin
- snd_soc_dapm_nc_pin

到此，widget已经被正确地创建并初始化，而且被挂在声卡的widgets链表中，以后我们就可以通过声卡的widgets链表来遍历所有的widget，再次强调一下snd_soc_dapm_new_controls函数所完成的主要功能：

- 为widget分配内存，并拷贝参数中传入的在驱动中定义好的模板
- 设置power_check回调函数
- 把widget挂在声卡的widgets链表中

为widget建立dapm kcontrol

定义一个widget，我们需要指定两个很重要的内容：一个是用于控制widget的电源状态的reg/shift等寄存器信息，另一个是用于控制音频路径切换的dapm kcontrol信息，这些dapm kcontrol有它们自己的reg/shift寄存器信息用于切换widget的路径连接方式。前一节的内容中，我们只是创建了widget的实例，并把它们注册到声卡的widgts链表中，但是到目前为止，包含在widget中的dapm kcontrol并没有建立起来，dapm框架在声卡的初始化阶段，等所有的widget（包括machine、platform、codec）都创建好之后，通过snd_soc_dapm_new_widgets函数，创建widget内包含的dapm kcontrol，并初始化widget的初始电源状态和音频路径的初始连接状态。我们看看声卡的初始化函数，都有那些初始化与dapm有关：

```
[cpp]

01. static int snd_soc_instantiate_card(struct snd_soc_card *card)
02. {
03.         ......
04.         /* card bind complete so register a sound card */
05.         ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
06.                         card->owner, 0, &card->snd_card);
07.         ......
08.
09.         card->dapm.bias_level = SND_SOC_BIAS_OFF;
10.         card->dapm.dev = card->dev;
11.         card->dapm.card = card;
12.         list_add(&card->dapm.list, &card->dapm_list);
13.
14. #ifdef CONFIG_DEBUG_FS
15.         snd_soc_dapm_debugfs_init(&card->dapm, card->debugfs_card_root);
16. #endif
17.         ......
18.         if (card->dapm_widgets)    /* 创建machine级别的widget  */
19.                 snd_soc_dapm_new_controls(&card->dapm, card->dapm_widgets,
20.                                           card->num_dapm_widgets);
21.         ......
22.         snd_soc_dapm_link_dai_widgets(card);  /*  连接dai widget  */
23.
24.         if (card->controls)    /*  建立machine级别的普通kcontrol控件  */
25.                 snd_soc_add_card_controls(card, card->controls, card->num_controls);
26.
27.         if (card->dapm_routes)    /*  注册machine级别的路径连接信息  */
28.                 snd_soc_dapm_add_routes(&card->dapm, card->dapm_routes,
29.                                         card->num_dapm_routes);
```

```
30.         ......
31.
32.         if (card->fully_routed)    /*  如果该标志被置位，自动把codec中没有路径连接信息的引脚设置为无用widget  */
33.                 list_for_each_entry(codec, &card->codec_dev_list, card_list)
34.                         snd_soc_dapm_auto_nc_codec_pins(codec);
35.
36.         snd_soc_dapm_new_widgets(card);    /*初始化widget包含的dapm kcontrol、电源状态和连接状态*/
37.
38.         ret = snd_card_register(card->snd_card);
39.         ......
40.         card->instantiated = 1;
41.         snd_soc_dapm_sync(&card->dapm);
42.         ......
43.         return 0;
44. }
```

正如我添加的注释中所示，在完成machine级别的widget和route处理之后，调用的snd_soc_dapm_new_widgets函数，来为所有已经注册的widget初始化他们所包含的dapm kcontrol，并初始化widget的电源状态和路径连接状态。下面我们看看snd_soc_dapm_new_widgets函数的工作过程。

snd_soc_dapm_new_widgets函数

该函数通过声卡的widgets链表，遍历所有已经注册了的widget，其中的new字段用于判断该widget是否已经执行过snd_soc_dapm_new_widgets函数，如果num_kcontrols字段有数值，表明该widget包含有若干个dapm kcontrol，那么就需要为这些kcontrol分配一个指针数组，并把数组的首地址赋值给widget的kcontrols字段，该数组存放着指向这些kcontrol的指针，当然现在这些都是空指针，因为实际的kcontrol现在还没有被创建：

```
[cpp]

01. int snd_soc_dapm_new_widgets(struct snd_soc_card *card)
02. {
03.         ......
04.         list_for_each_entry(w, &card->widgets, list)
05.         {
06.                 if (w->new)
07.                         continue;
08.
09.                 if (w->num_kcontrols) {
10.                         w->kcontrols = kzalloc(w->num_kcontrols *
11.                                                 sizeof(struct snd_kcontrol *),
12.                                                 GFP_KERNEL);
13.                         ......
14.                 }
```

接着，对几种能影响音频路径的widget，创建并初始化它们所包含的dapm kcontrol：

```
[cpp]

01. switch(w->id) {
02. case snd_soc_dapm_switch:
03. case snd_soc_dapm_mixer:
04. case snd_soc_dapm_mixer_named_ctl:
05.         dapm_new_mixer(w);
06.         break;
07. case snd_soc_dapm_mux:
08. case snd_soc_dapm_virt_mux:
09. case snd_soc_dapm_value_mux:
10.         dapm_new_mux(w);
11.         break;
12. case snd_soc_dapm_pga:
13. case snd_soc_dapm_out_drv:
14.         dapm_new_pga(w);
15.         break;
16. default:
17.         break;
18. }
```

需要用到的创建函数分别是：

- dapm_new_mixer()　对于mixer类型，用该函数创建dapm kcontrol；
- dapm_new_mux()　对于mux类型，用该函数创建dapm kcontrol；
- dapm_new_pga()　对于pga类型，用该函数创建dapm kcontrol；

然后，根据widget寄存器的当前值，初始化widget的电源状态，并设置到power字段中：

```cpp
/* Read the initial power state from the device */
if (w->reg >= 0) {
        val = soc_widget_read(w, w->reg) >> w->shift;
        val &= w->mask;
        if (val == w->on_val)
                w->power = 1;
}
```

接着，设置new字段，表明该widget已经初始化完成，我们还要吧该widget加入到声卡的dapm_dirty链表中，表明该widget的状态发生了变化，稍后在合适的时刻，dapm框架会扫描dapm_dirty链表，统一处理所有已经变化的widget。为什么要统一处理？因为dapm要控制各种widget的上下电顺序，同时也是为了减少寄存器的读写次数（多个widget可能使用同一个寄存器）：

```cpp
w->new = 1;

dapm_mark_dirty(w, "new widget");
dapm_debugfs_add_widget(w);
```

最后，通过dapm_power_widgets函数，统一处理所有位于dapm_dirty链表上的widget的状态改变：

```cpp
dapm_power_widgets(card, SND_SOC_DAPM_STREAM_NOP);
......
return 0;
```

dapm mixer kcontrol

上一节中，我们提到，对于mixer类型的dapm kcontrol，我们会使用dapm_new_mixer来完成具体的创建工作，先看代码后分析：

```cpp
static int dapm_new_mixer(struct snd_soc_dapm_widget *w)
{
        int i, ret;
        struct snd_soc_dapm_path *path;

        /* add kcontrol */
<span style="font-family:Arial,Helvetica,sans-serif">（1）
</span>        for (i = 0; i < w->num_kcontrols; i++) {
                /* match name */
（2）                list_for_each_entry(path, &w->sources, list_sink) {
                    /* mixer/mux paths name must match control name */
（3）                    if (path->name != (char *)w->kcontrol_news[i].name)
                        continue;

（4）                    if (w->kcontrols[i]) {
                        dapm_kcontrol_add_path(w->kcontrols[i], path);
                        continue;
                    }

（5）                    ret = dapm_create_or_share_mixmux_kcontrol(w, i);
                    if (ret < 0)
                        return ret;

（6）                    dapm_kcontrol_add_path(w->kcontrols[i], path);
                }
        }

        return 0;
}
```

（1） 因为一个mixer是由多个kcontrol组成的，每个kcontrol控制着mixer的一个输入端的开启和关闭，所以，该函数会根据kcontrol的数量做循环，逐个建立对应的kcontrol。

（2）（3） 之前多次提到，widget之间使用snd_soc_path进行连接，widget的sources链表保存着所有和输入端连接的snd_soc_path结构，所以我们可以用kcontrol模板中指定的名字来匹配对应的snd_soc_path结构。

（4） 因为一个输入脚可能会连接多个输入源，所以可能在上一个输入源的path关联时已经创建了这个kcontrol，所以这里判断kcontrols指针数组中对应索引中的指针值，如果已经赋值，说明kcontrol已经在之前创建好了，所以

我们只要简单地把连接该输入端的path加入到kcontrol的path_list链表中，并且增加一个虚拟的影子widget，该影子widget连接和输入端对应的源widget，因为使用了kcontrol本身的reg/shift等寄存器信息，所以实际上控制的是该kcontrol的开和关，这个影子widget只有在kcontrol的autodisable字段被设置的情况下才会被创建，该特性使得source的关闭时，与之连接的mixer的输入端也可以自动关闭，这个特性通过dapm_kcontrol_add_path来实现这一点：

```cpp
static void dapm_kcontrol_add_path(const struct snd_kcontrol *kcontrol,
        struct snd_soc_dapm_path *path)
{
        struct dapm_kcontrol_data *data = snd_kcontrol_chip(kcontrol);
        /*  把kcontrol连接的path加入到paths链表中  */
        /*  paths链表所在的dapm_kcontrol_data结构会保存在kcontrol的private_data字段中  */
        list_add_tail(&path->list_kcontrol, &data->paths);

        if (data->widget) {
                snd_soc_dapm_add_path(data->widget->dapm, data->widget,
                    path->source, NULL, NULL);
        }
}
```

（5） 如果kcontrol之前没有被创建，则通过dapm_create_or_share_mixmux_kcontrol创建这个输入端的kcontrol，同理，kcontrol对应的影子widget也会通过dapm_kcontrol_add_path判断是否需要创建。

dapm mux kcontrol

因为一个widget最多只会包含一个mux类型的damp kcontrol，所以他的创建方法稍有不同，dapm框架使用dapm_new_mux函数来创建mux类型的dapm kcontrol：

```cpp
static int dapm_new_mux(struct snd_soc_dapm_widget *w)
{
        struct snd_soc_dapm_context *dapm = w->dapm;
        struct snd_soc_dapm_path *path;
        int ret;

(1)     if (w->num_kcontrols != 1) {
                dev_err(dapm->dev,
                        "ASoC: mux %s has incorrect number of controls\n",
                        w->name);
                return -EINVAL;
        }

        if (list_empty(&w->sources)) {
                dev_err(dapm->dev, "ASoC: mux %s has no paths\n", w->name);
                return -EINVAL;
        }

(2)     ret = dapm_create_or_share_mixmux_kcontrol(w, 0);
        if (ret < 0)
                return ret;
(3)       list_for_each_entry(path, &w->sources, list_sink)
                dapm_kcontrol_add_path(w->kcontrols[0], path);
        return 0;
}
```

（1） 对于mux类型的widget，因为只会有一个kcontrol，所以在这里做一下判断。

（2） 同样地，和mixer类型一样，也使用dapm_create_or_share_mixmux_kcontrol来创建这个kcontrol。

（3） 对每个输入端所连接的path都加入dapm_kcontrol_data结构的paths链表中，并且创建一个影子widget，用于支持autodisable特性。

dapm pga kcontrol

目前对于pga类型的widget，kcontrol的创建函数是个空函数，所以我们不用太关注它：

```cpp
static int dapm_new_pga(struct snd_soc_dapm_widget *w)
{
        if (w->num_kcontrols)
                dev_err(w->dapm->dev,
```

```
                        "ASoC: PGA controls not supported: '%s'\n", w->name);

        return 0;
}
```

dapm_create_or_share_mixmux_kcontrol函数

上面所说的mixer类型和mux类型的widget，在创建他们所包含的dapm kcontrol时，最后其实都是使用了dapm_create_or_share_mixmux_kcontrol函数来完成创建工作的，所以在这里我们有必要分析一下这个函数的工作原理。这个函数中有很大一部分代码实在处理kcontrol的名字是否要加入codec的前缀，我们会忽略这部分的代码，感兴趣的读者可以自己查看内核的代码，路径在：sound/soc/soc-dapm.c中，简化后的代码如下：

```
[cpp]

static int dapm_create_or_share_mixmux_kcontrol(struct snd_soc_dapm_widget *w,
        int kci)
{
          ......
(1)       shared = dapm_is_shared_kcontrol(dapm, w, &w->kcontrol_news[kci],
                                           &kcontrol);

(2)       if (!kcontrol) {
(3)            kcontrol = snd_soc_cnew(&w->kcontrol_news[kci], NULL, name,prefix）;
               ......
               kcontrol->private_free = dapm_kcontrol_free;
(4)            ret = dapm_kcontrol_data_alloc(w, kcontrol);
                ......
(5)            ret = snd_ctl_add(card, kcontrol);
                ......
        }
(6)     ret = dapm_kcontrol_add_widget(kcontrol, w);
        ......
(7)     w->kcontrols[kci] = kcontrol;
        return 0;
}
```

（1） 为了节省内存，通过kcontrol名字的匹配查找，如果这个kcontrol已经在其他widget中已经创建好了，那我们不再创建，dapm_is_shared_kcontrol的参数kcontrol会返回已经创建好的kcontrol的指针。
（2） 如果kcontrol指针被赋值，说明在（1）中查找到了其他widget中同名的kcontrol，我们不用再次创建，只要共享该kcontrol即可。
（3） 标准的kcontrol创建函数，请参看：Linux ALSA声卡驱动之四：Control设备的创建中的“创建control“一节的内容。
（4） 如果widget支持autodisable特性，创建与该kcontrol所对应的影子widget，该影子widget的类型是：snd_soc_dapm_kcontrol。
（5） 标准的kcontrol创建函数，请参看：Linux ALSA声卡驱动之四：Control设备的创建中的“创建control“一节的内容。
（6） 把所有共享该kcontrol的影子widget（snd_soc_dapm_kcontrol），加入到kcontrol的private_data字段所指向的dapm_kcontrol_data结构中。
（7） 把创建好的kcontrol指针赋值到widget的kcontrols数组中。
需要注意的是，如果kcontol支持autodisable特性，一旦kcontrol由于source的关闭而被自动关闭，则用户空间只能操作该kcontrol的cache值，只有该kcontrol再次打开时，该cache值才会被真正地更新到寄存器中。
现在。我们总结一下，创建一个widget所包含的kcontrol所做的工作：

- 循环每一个输入端，为每个输入端依次执行下面的一系列操作
- 为每个输入端创建一个kcontrol，能共享的则直接使用创建好的kcontrol
- kcontrol的private_data字段保存着这些共享widget的信息
- 如果支持autodisable特性，每个输入端还要额外地创建一个虚拟的snd_soc_dapm_kcontrol类型的影子widget，该影子widget也记录在private_data字段中
- 创建好的kcontrol会依次存放在widget的kcontrols数组中，供路径的控制和匹配之用。

为widget建立连接关系

如果widget之间没有连接关系，dapm就无法实现动态的电源管理工作，正是widget之间有了连结关系，这些连接关系形成了一条所谓的完成的音频路径，dapm可以顺着这条路径，统一控制路径上所有widget的电源状态，前面我们已经知道，widget之间是使用snd_soc_path结构进行连接的，驱动要做的是定义一个snd_soc_route结构数组，该数组的每个条目描述了目的widget的和源widget的名称，以及控制这个连接的kcontrol的名称，最终，驱动

程序使用api函数snd_soc_dapm_add_routes来注册这些连接信息，接下来我们就是要分析该函数的具体实现方式：

```cpp
int snd_soc_dapm_add_routes(struct snd_soc_dapm_context *dapm,
                            const struct snd_soc_dapm_route *route, int num)
{
        int i, r, ret = 0;

        mutex_lock_nested(&dapm->card->dapm_mutex, SND_SOC_DAPM_CLASS_INIT);
        for (i = 0; i < num; i++) {
                r = snd_soc_dapm_add_route(dapm, route);
                ......
                route++;
        }
        mutex_unlock(&dapm->card->dapm_mutex);

        return ret;
}
```

该函数只是一个循环，依次对参数传入的数组调用snd_soc_dapm_add_route，主要的工作由snd_soc_dapm_add_route完成。我们进入snd_soc_dapm_add_route函数看看：

```cpp
static int snd_soc_dapm_add_route(struct snd_soc_dapm_context *dapm,
                                  const struct snd_soc_dapm_route *route)
{
        struct snd_soc_dapm_widget *wsource = NULL, *wsink = NULL, *w;
        struct snd_soc_dapm_widget *wtsource = NULL, *wtsink = NULL;
        const char *sink;
        const char *source;
        ......
        list_for_each_entry(w, &dapm->card->widgets, list) {
                if (!wsink && !(strcmp(w->name, sink))) {
                        wtsink = w;
                        if (w->dapm == dapm)
                                wsink = w;
                        continue;
                }
                if (!wsource && !(strcmp(w->name, source))) {
                        wtsource = w;
                        if (w->dapm == dapm)
                                wsource = w;
                }
        }
```

上面的代码我再次省略了关于名称前缀的处理部分。我们可以看到，用widget的名字来比较，遍历声卡的widgets链表，找出源widget和目的widget的指针，这段代码虽然正确，但我总感觉少了一个判断退出循环的条件，如果链表的开头就找到了两个widget，还是要遍历整个链表才结束循环，好浪费时间。
下面，如果在本dapm context中没有找到，则使用别的dapm context中找到的widget：

```cpp
if (!wsink)
        wsink = wtsink;
if (!wsource)
        wsource = wtsource;
```

最后，使用来增加一条连接信息：

```cpp
        ret = snd_soc_dapm_add_path(dapm, wsource, wsink, route->control,
                route->connected);
        ......

        return 0;
}
```

snd_soc_dapm_add_path函数是整个调用链条中的关键，我们来分析一下：

```cpp
static int snd_soc_dapm_add_path(struct snd_soc_dapm_context *dapm,
```

```
        struct snd_soc_dapm_widget *wsource, struct snd_soc_dapm_widget *wsink,
        const char *control,
        int (*connected)(struct snd_soc_dapm_widget *source,
                         struct snd_soc_dapm_widget *sink))
{
        struct snd_soc_dapm_path *path;
        int ret;

        path = kzalloc(sizeof(struct snd_soc_dapm_path), GFP_KERNEL);
        if (!path)
                return -ENOMEM;

        path->source = wsource;
        path->sink = wsink;
        path->connected = connected;
        INIT_LIST_HEAD(&path->list);
        INIT_LIST_HEAD(&path->list_kcontrol);
        INIT_LIST_HEAD(&path->list_source);
        INIT_LIST_HEAD(&path->list_sink);
```

函数的一开始，首先为这个连接分配了一个snd_soc_path结构，path的source和sink字段分别指向源widget和目的widget，connected字段保存connected回调函数，初始化几个snd_soc_path结构中的几个链表。

```
[cpp]

/* check for external widgets */
        if (wsink->id == snd_soc_dapm_input) {
                if (wsource->id == snd_soc_dapm_micbias ||
                        wsource->id == snd_soc_dapm_mic ||
                        wsource->id == snd_soc_dapm_line ||
                        wsource->id == snd_soc_dapm_output)
                        wsink->ext = 1;
        }
        if (wsource->id == snd_soc_dapm_output) {
                if (wsink->id == snd_soc_dapm_spk ||
                        wsink->id == snd_soc_dapm_hp ||
                        wsink->id == snd_soc_dapm_line ||
                        wsink->id == snd_soc_dapm_input)
                        wsource->ext = 1;
        }
```

这段代码用于判断是否有外部连接关系，如果有，置位widget的ext字段。判断方法从代码中可以方便地看出：

- 目的widget是一个输入脚，如果源widget是mic、line、micbias或output，则认为目的widget具有外部连接关系。
- 源widget是一个输出脚，如果目的widget是spk、hp、line或input，则认为源widget具有外部连接关系。

```
[cpp]

dapm_mark_dirty(wsource, "Route added");
dapm_mark_dirty(wsink, "Route added");

/* connect static paths */
if (control == NULL) {
        list_add(&path->list, &dapm->card->paths);
        list_add(&path->list_sink, &wsink->sources);
        list_add(&path->list_source, &wsource->sinks);
        path->connect = 1;
        return 0;
}
```

因为增加了连结关系，所以把源widget和目的widget加入到dapm_dirty链表中。如果没有kcontrol来控制该连接关系，则这是一个静态连接，直接用path把它们连接在一起。在接着往下看：

```
[cpp]

/* connect dynamic paths */
switch (wsink->id) {
case snd_soc_dapm_adc:
case snd_soc_dapm_dac:
case snd_soc_dapm_pga:
case snd_soc_dapm_out_drv:
case snd_soc_dapm_input:
case snd_soc_dapm_output:
case snd_soc_dapm_siggen:
case snd_soc_dapm_micbias:
case snd_soc_dapm_vmid:
```

```
case snd_soc_dapm_pre:
case snd_soc_dapm_post:
case snd_soc_dapm_supply:
case snd_soc_dapm_regulator_supply:
case snd_soc_dapm_clock_supply:
case snd_soc_dapm_aif_in:
case snd_soc_dapm_aif_out:
case snd_soc_dapm_dai_in:
case snd_soc_dapm_dai_out:
case snd_soc_dapm_dai_link:
case snd_soc_dapm_kcontrol:
        list_add(&path->list, &dapm->card->paths);
        list_add(&path->list_sink, &wsink->sources);
        list_add(&path->list_source, &wsource->sinks);
        path->connect = 1;
        return 0;
```

按照目的widget来判断，如果属于以上这些类型，直接把它们连接在一起即可，这段感觉有点多余，因为通常以上这些类型的widget本来也没有kcontrol，直接用上一段代码就可以了，也许是dapm的作者们想着以后可能会有所扩展吧。

```
[cpp]

case snd_soc_dapm_mux:
case snd_soc_dapm_virt_mux:
case snd_soc_dapm_value_mux:
        ret = dapm_connect_mux(dapm, wsource, wsink, path, control,
                &wsink->kcontrol_news[0]);
        if (ret != 0)
                goto err;
        break;
case snd_soc_dapm_switch:
case snd_soc_dapm_mixer:
case snd_soc_dapm_mixer_named_ctl:
        ret = dapm_connect_mixer(dapm, wsource, wsink, path, control);
        if (ret != 0)
                goto err;
        break;
```

目的widget如果是mixer和mux类型，分别用dapm_connect_mixer和dapm_connect_mux函数完成连接工作，这两个函数我们后面再讲。

```
[cpp]

        case snd_soc_dapm_hp:
        case snd_soc_dapm_mic:
        case snd_soc_dapm_line:
        case snd_soc_dapm_spk:
                list_add(&path->list, &dapm->card->paths);
                list_add(&path->list_sink, &wsink->sources);
                list_add(&path->list_source, &wsource->sinks);
                path->connect = 0;
                return 0;
        }

        return 0;
err:
        kfree(path);
        return ret;
}
```

hp、mic、line和spk这几种widget属于外部器件，也只是简单地连接在一起，不过connect字段默认为是未连接状态。

现在，我们回过头来看看目的widget是mixer和mux这两种类型时的连接方式：

dapm_connect_mixer 用该函数连接一个目的widget为mixer类型的所有输入端：

```
[cpp]

static int dapm_connect_mixer(struct snd_soc_dapm_context *dapm,
        struct snd_soc_dapm_widget *src, struct snd_soc_dapm_widget *dest,
        struct snd_soc_dapm_path *path, const char *control_name)
{
        int i;

        /* search for mixer kcontrol */
        for (i = 0; i < dest->num_kcontrols; i++) {
```

```
                if (!strcmp(control_name, dest->kcontrol_news[i].name)) {
                        list_add(&path->list, &dapm->card->paths);
                        list_add(&path->list_sink, &dest->sources);
                        list_add(&path->list_source, &src->sinks);
                        path->name = dest->kcontrol_news[i].name;
                        dapm_set_path_status(dest, path, i);
                        return 0;
                }
        }
        return -ENODEV;
}
```

用需要用来连接的kcontrol的名字，和目的widget中的kcontrol模板数组中的名字相比较，找出该kcontrol在widget中的编号，path的名字设置为该kcontrol的名字，然后用dapm_set_path_status函数来初始化该输入端的连接状态。连接两个widget的链表操作和其他widget是一样的。

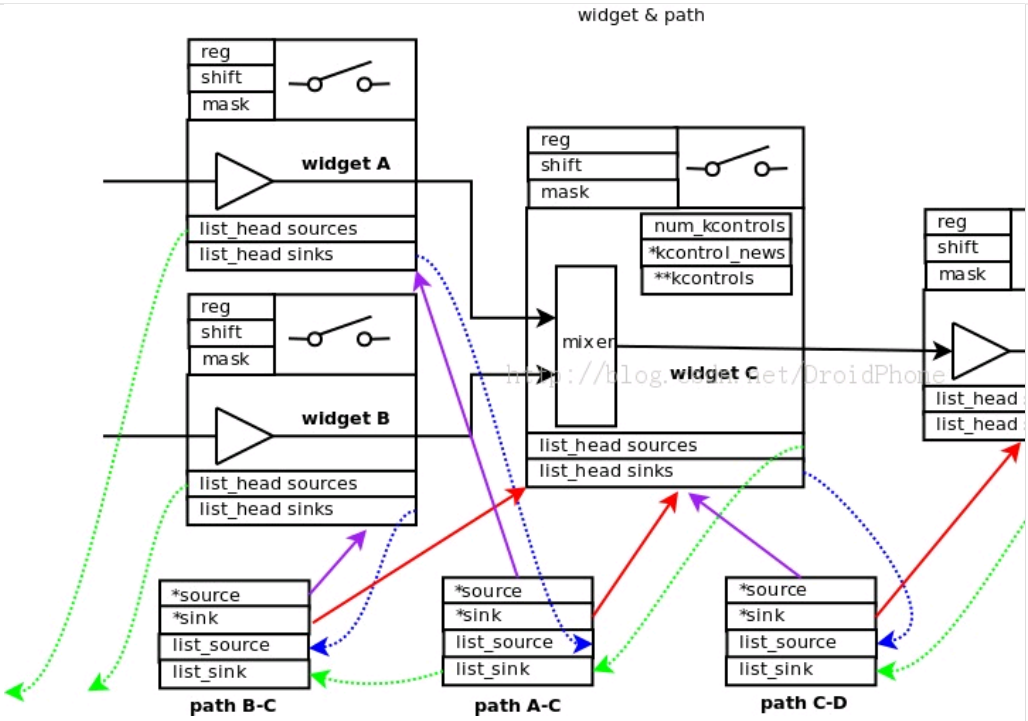
dapm_connect_mux 用该函数连接一个目的widget是mux类型的所有输入端：

```
[cpp]

static int dapm_connect_mux(struct snd_soc_dapm_context *dapm,
        struct snd_soc_dapm_widget *src, struct snd_soc_dapm_widget *dest,
        struct snd_soc_dapm_path *path, const char *control_name,
        const struct snd_kcontrol_new *kcontrol)
{
        struct soc_enum *e = (struct soc_enum *)kcontrol->private_value;
        int i;

        for (i = 0; i < e->max; i++) {
                if (!(strcmp(control_name, e->texts[i]))) {
                        list_add(&path->list, &dapm->card->paths);
                        list_add(&path->list_sink, &dest->sources);
                        list_add(&path->list_source, &src->sinks);
                        path->name = (char*)e->texts[i];
                        dapm_set_path_status(dest, path, 0);
                        return 0;
                }
        }

        return -ENODEV;
}
```

和mixer类型一样用名字进行匹配，只不过mux类型的kcontrol只需一个，所以要通过private_value字段所指向的soc_enum结构找出匹配的输入脚编号，最后也是通过dapm_set_path_status函数来初始化该输入端的连接状态，因为只有一个kcontrol，所以第三个参数是0。连接两个widget的链表操作和其他widget也是一样的。

dapm_set_path_status 该函数根据传入widget中的kcontrol编号，读取实际寄存器的值，根据寄存器的值来初始化这个path是否处于连接状态，详细的代码这里就不贴了。

当widget之间通过path进行连接之后，他们之间的关系就如下图所示：



到这里为止，我们为声卡创建并初始化好了所需的widget，各个widget也通过path连接在了一起，接下来，dapm等待用户的指令，一旦某个dapm kcontrol被用户空间改变，利用这些连接关系，dapm会重新创建音频路径，脱离音频路径的widget会被下电，加入音频路径的widget会被上电，所有的上下电动作都会自动完成，用户空间的应用程序无需关注这些变化，它只管按需要改变某个dapm kcontrol即可。

更多 0

上一篇：ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route

下一篇：ALSA声卡驱动中的DAPM详解之六：精髓所在，牵一发而动全身

查看评论

2楼 hustljh 2013-11-06 21:01发表

《AUDIO+CODEC+DAPM》也有一个widget和path关系的图，我之前的理解是path B-C也是由widget C的source指出的，现在仔细看来，那篇文章跟博主表达的应该是同一个意思，是我理解错了。

1楼 hustljh 2013-11-05 23:03发表

widget&path图中指向path B-C的绿色箭头起始端是不是画错了？

Re: DroidPhone 2013-11-06 09:22发表

回复hustljh：没有画错。您如何理解它们之间的关系？

专区推荐内容

如何提高游戏后台数据查找效率
从零开始编写网络游戏
#HTML(HTML5)：多媒體...
[C/C++]类别互相引用
超极本 和平板电脑 Window...
以用户为中心实现绝佳的用户体验

更多招聘职位 **我公司职位也要出现在这里**

【哈票网络科技（北京）有限公司】Java研发组组长
【万维嘉信（山东）电子商务有限公司】高级运维经理
【广州友魄信息科技有限公司】Asp.net（C#）开发工程师
【北京奥鹏远程教育中心有限公司】专职/兼职IT培训讲师（Web前端开发，PHP，3G开发方向）
【langoon】java架构师

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382370次
积分： 3435分
排名： 第2116名

原创： 46篇 转载： 0篇
译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之六：精髓所在，牵一发而动全身

分类： Linux音频子系统 2013-11-04 23:20 538人阅读 评论(7) 收藏 举报

linux audio driver widget dapm alsa

设计dapm的主要目的之一，就是希望声卡上的各种部件的电源按需分配，需要的就上电，不需要的就下电，使得整个音频系统总是处于最小的耗电状态，最主要的就是，这一切对用户空间的应用程序是透明的，也就是说，用户空间的应用程序无需关心那个部件何时需要电源，它只要按需要设定好音频路径，播放音频数据，暂停或停止，dapm框架会根据音频路径，完美地对各种部件的电源进行控制，而且精确地按某种顺序进行，防止上下电过程中产生不必要的pop-pop声。这就是本章我们需要讨论的内容。

统计widget连接至端点widget的路径个数

ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route这篇文章中的最后一节，我们曾经提出了端点widget这一概念，端点widget位于音频路径的起始端或者末端，所以通常它们就是指codec的输入输出引脚所对应的widget，或者是外部器件对应的widget，这些widget的类型有以下这些：

端点widget的种类

分类	widget类型
codec的输入输出引脚	snd_soc_dapm_output snd_soc_dapm_input
外接的音频设备	snd_soc_dapm_hp snd_soc_dapm_spk snd_soc_dapm_line snd_soc_dapm_mic
音频流（stream domain）	snd_soc_dapm_adc snd_soc_dapm_dac snd_soc_dapm_aif_out snd_soc_dapm_aif_in snd_soc_dapm_dai_out snd_soc_dapm_dai_in

(18248)
Linux ALSA声卡驱动之三
(17112)
Android中的sp和wp指针
(13786)
Linux ALSA声卡驱动之七
(13061)
Android SurfaceFlinger中
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

电源、时钟	snd_soc_dapm_supply snd_soc_dapm_regulator_supply snd_soc_dapm_clock_supply
影子widget	snd_soc_dapm_kcontrol

dapm要给一个widget上电的其中一个前提条件是：这个widget位于一条完整的音频路径上，而一条完整的音频路径的两头，必须是输入/输出引脚，或者是一个外部音频设备，又或者是一个处于激活状态的音频流widget，也就是上表中的前三项，上表中的后两项，它们可以位于路径的末端，但不是构成完成音频路径的必要条件，我们只用它来判断扫描一条路径的结束条件。dapm提供了两个内部函数，用来统计一个widget连接到输出引脚、输入引脚、激活的音频流widget的有效路径个数：

- is_connected_output_ep　返回连接至输出引脚或激活状态的输出音频流的路径数量
- is_connected_input_ep　返回连接至输入引脚或激活状态的输入音频流的路径数量

下面我贴出is_connected_output_ep函数和必要的注释：

```
[html]

static int is_connected_output_ep(struct snd_soc_dapm_widget *widget,
        struct snd_soc_dapm_widget_list **list)
{
        struct snd_soc_dapm_path *path;
        int con = 0;
        /*  多个路径可能使用了同一个widget，如果在遍历另一个路径时，*/
        /*  已经统计过该widget，直接返回output字段即可。              */
        if (widget->outputs >= 0)
                return widget->outputs;

        /*  以下这几种widget是端点widget，但不是输出，所以直接返回0，结束该路径的扫描   */
        switch (widget->id) {
        case snd_soc_dapm_supply:
        case snd_soc_dapm_regulator_supply:
        case snd_soc_dapm_clock_supply:
        case snd_soc_dapm_kcontrol:
                return 0;
        default:
                break;
        }
        /*  对于音频流widget，如果处于激活状态，如果没有休眠，返回1，否则，返回0  */
        /*  而且对于激活的音频流widget是端点widget，所以也会结束该路径的扫描   */
        /*  如果没有处于激活状态，按普通的widget继续往下执行   */
        switch (widget->id) {
        case snd_soc_dapm_adc:
        case snd_soc_dapm_aif_out:
        case snd_soc_dapm_dai_out:
                if (widget->active) {
                        widget->outputs = snd_soc_dapm_suspend_check(widget);
                        return widget->outputs;
                }
        default:
                break;
        }

        if (widget->connected) {
                /* 处于连接状态的输出引脚，也根据休眠状态返回1或0 */
                if (widget->id == snd_soc_dapm_output && !widget->ext) {
                        widget->outputs = snd_soc_dapm_suspend_check(widget);
                        return widget->outputs;
                }

                /* 处于连接状态的输出设备，也根据休眠状态返回1或0 */
                if (widget->id == snd_soc_dapm_hp ||
                    widget->id == snd_soc_dapm_spk ||
                    (widget->id == snd_soc_dapm_line &&
                     !list_empty(&widget->sources))) {
                        widget->outputs = snd_soc_dapm_suspend_check(widget);
                        return widget->outputs;
                }
        }
        /*  不是端点widget，循环查询它的输出端   */
        list_for_each_entry(path, &widget->sinks, list_source) {
                DAPM_UPDATE_STAT(widget, neighbour_checks);

                if (path->weak)
                        continue;
```

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

```
58.
59.                 if (path->walking)   /* 比较奇怪，防止无限循环的路径？ */
60.                         return 1;
61.
62.                 if (path->walked)
63.                         continue;
64.
65.                 if (path->sink && path->connect) {
66.                         path->walked = 1;
67.                         path->walking = 1;
68.                         ......
69.                         /*  递归调用，统计每一个输出端  */
70.                         con += is_connected_output_ep(path->sink, list);
71.
72.                         path->walking = 0;
73.                 }
74.         }
75.
76.         widget->outputs = con;
77.
78.         return con;
79. }
```

该函数使用了递归算法，直到遇到端点widget为止才停止扫描，把统计到的输出路径个数保存在output字段中并返回。is_connected_intput_ep函数的原理差不多，有兴趣的苏浙可以自己查看内核的原码。

dapm_dirty链表

在代表声卡的snd_soc_card结构中，有一个链表字段：dapm_dirty，所有状态发生了改变的widget，dapm不会立刻处理它的电源状态，而是需要先挂在该链表下面，等待后续的进一步处理：或者是上电，或者是下电。dapm为我们提供了一个api函数来完成这个动作：

```
[html]

01. void dapm_mark_dirty(struct snd_soc_dapm_widget *w, const char *reason)
02. {
03.         if (!dapm_dirty_widget(w)) {
04.                 dev_vdbg(w->dapm->dev, "Marking %s dirty due to %s\n",
05.                         w->name, reason);
06.                 list_add_tail(&w->dirty, &w->dapm->card->dapm_dirty);
07.         }
08. }
```

power_check回调函数

在文章 ALSA声卡驱动中的DAPM详解之五：建立widget之间的连接关系中，我们知道，在创建widget的时候，widget的power_check回调函数会根据widget的类型，设置不同的回调函数。当widget的状态改变后，dapm会遍历dapm_dirty链表，并通过power_check回调函数，决定该widget是否需要上电。大多数的widget的power_check回调被设置为：dapm_generic_check_power：

```
[html]

01. static int dapm_generic_check_power(struct snd_soc_dapm_widget *w)
02. {
03.         int in, out;
04.
05.         DAPM_UPDATE_STAT(w, power_checks);
06.
07.         in = is_connected_input_ep(w, NULL);
08.         dapm_clear_walk_input(w->dapm, &w->sources);
09.         out = is_connected_output_ep(w, NULL);
10.         dapm_clear_walk_output(w->dapm, &w->sinks);
11.         return out != 0 && in != 0;
12. }
```

很简单，分别用is_connected_output_ep和is_connected_input_ep得到该widget是否有同时连接到一个输入端和一个输出端，如果是，返回1来表示该widget需要上电。

对于snd_soc_dapm_dai_out和snd_soc_dapm_dai_in类型，power_check回调是dapm_adc_check_power和dapm_dac_check_power，这里以dapm_dac_check_power为例：

```
[html]
```

```
static int dapm_dac_check_power(struct snd_soc_dapm_widget *w)
{
        int out;

        DAPM_UPDATE_STAT(w, power_checks);

        if (w->active) {
                out = is_connected_output_ep(w, NULL);
                dapm_clear_walk_output(w->dapm, &w->sinks);
                return out != 0;
        } else {
                return dapm_generic_check_power(w);
        }
}
```

处于激活状态时，只判断是否有连接到有效的输出路径即可，没有激活时，则需要同时判断是否有连接到输入路径和输出路径。

widget的上电和下电顺序

在扫描dapm_dirty链表时，dapm使用两个链表来分别保存需要上电和需要下电的widget：

- up_list　　保存需要上电的widget
- down_list　保存需要下电的widget

dapm内部使用dapm_seq_insert函数把一个widget加入到上述两个链表中的其中一个：

```
[cpp]

static void dapm_seq_insert(struct snd_soc_dapm_widget *new_widget,
                            struct list_head *list,
                            bool power_up)
{
        struct snd_soc_dapm_widget *w;

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

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

上述函数会按照一定的顺序把widget加入到链表中，从而保证正确的上下电顺序：

上电顺序	下电顺序
static int dapm_up_seq[] = { [snd_soc_dapm_pre] = 0, [snd_soc_dapm_supply] = 1, [snd_soc_dapm_regulator_supply] = 1, [snd_soc_dapm_clock_supply] = 1, [snd_soc_dapm_micbias] = 2, [snd_soc_dapm_dai_link] = 2, [snd_soc_dapm_dai_in] = 3, [snd_soc_dapm_dai_out] = 3, [snd_soc_dapm_aif_in] = 3, [snd_soc_dapm_aif_out] = 3, [snd_soc_dapm_mic] = 4, [snd_soc_dapm_mux] = 5, [snd_soc_dapm_virt_mux] = 5, [snd_soc_dapm_value_mux] = 5, [snd_soc_dapm_dac] = 6, [snd_soc_dapm_switch] = 7, [snd_soc_dapm_mixer] = 7, [snd_soc_dapm_mixer_named_ctl] = 7, [snd_soc_dapm_pga] = 8,	static int dapm_down_seq[] = { [snd_soc_dapm_pre] = 0, [snd_soc_dapm_kcontrol] = 1, [snd_soc_dapm_adc] = 2, [snd_soc_dapm_hp] = 3, [snd_soc_dapm_spk] = 3, [snd_soc_dapm_line] = 3, [snd_soc_dapm_out_drv] = 3, [snd_soc_dapm_pga] = 4, [snd_soc_dapm_switch] = 5, [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_virt_mux] = 9, [snd_soc_dapm_value_mux] = 9, [snd_soc_dapm_aif_in] = 10, [snd_soc_dapm_aif_out] = 10,

[snd_soc_dapm_adc] = 9, [snd_soc_dapm_out_drv] = 10, [snd_soc_dapm_hp] = 10, [snd_soc_dapm_spk] = 10, [snd_soc_dapm_line] = 10, [snd_soc_dapm_kcontrol] = 11, [snd_soc_dapm_post] = 12, };	[snd_soc_dapm_dai_in] = 10, [snd_soc_dapm_dai_out] = 10, [snd_soc_dapm_dai_link] = 11, [snd_soc_dapm_clock_supply] = 12, [snd_soc_dapm_regulator_supply] = 12, [snd_soc_dapm_supply] = 12, [snd_soc_dapm_post] = 13, };

widget的上下电过程

dapm_power_widgets

当一个widget的状态改变后，该widget会被加入dapm_dirty链表，然后通过dapm_power_widgets函数来改变整个音频路径上的电源状态，下图展现了这个函数的调用过程：

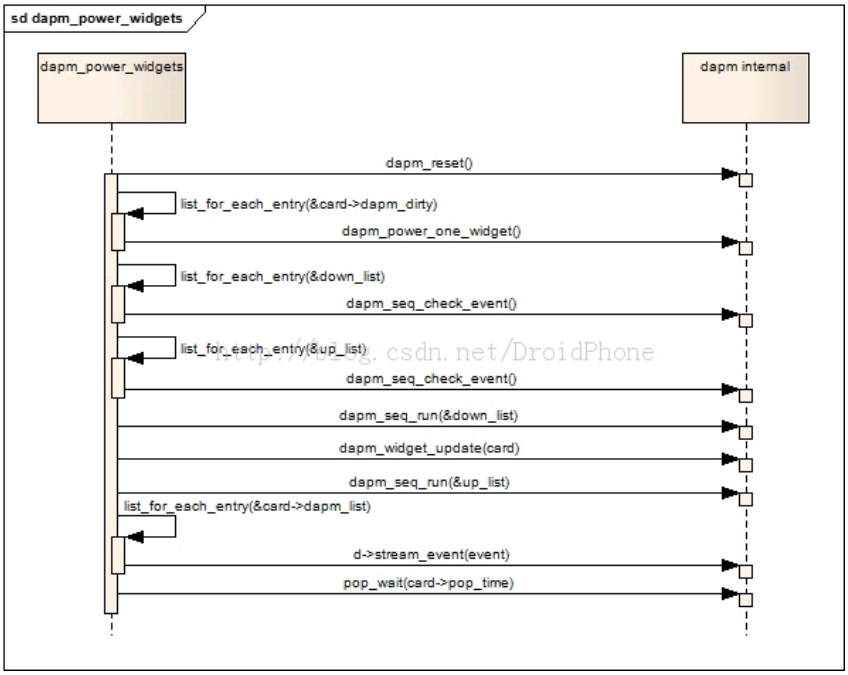


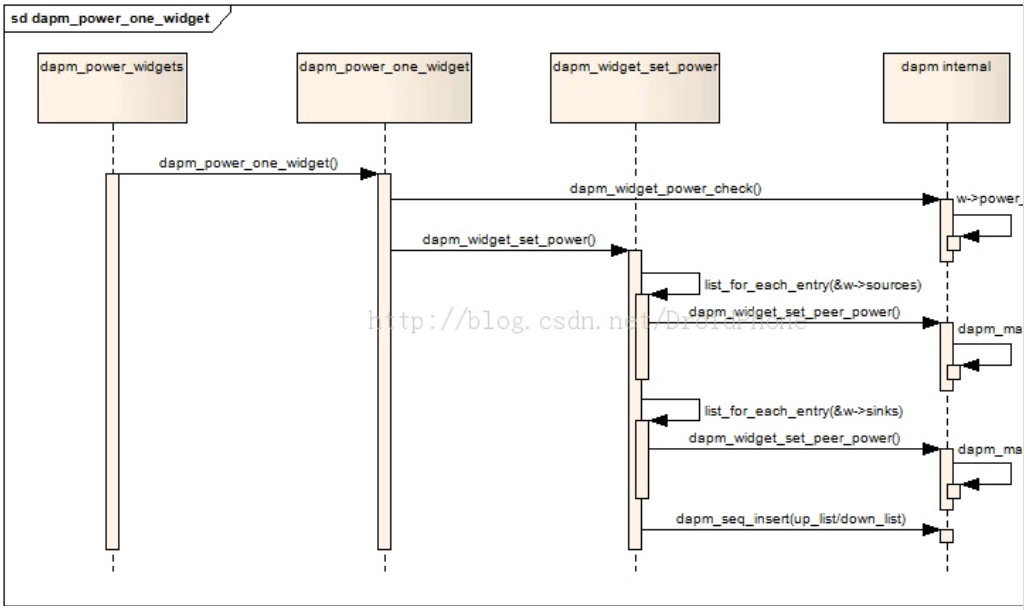
图1　widget的上电过程

- 可见，该函数通过遍历dapm_dirty链表，对每个链表中的widget调用dapm_power_one_widget，dapm_power_one_widget函数除了处理自身的状态改变外，还把自身的变化传递到和它相连的邻居widget中，结果就是，所有需要上电的widget会被放在up_list链表中，而所有需要下电的widget会被放在down_list链表中，这个函数我们稍后再讨论。
- 遍历down_list链表，向其中的widget发出SND_SOC_DAPM_WILL_PMD事件，感兴趣该事件的widget的event回调会被调用。
-
- 遍历up_list链表，向其中的widget发出SND_SOC_DAPM_WILL_PMU事件，感兴趣该事件的widget的event回调会被调用。
- 通过dapm_seq_run函数，处理down_list中的widget，使它们按定义好的顺序依次下电。
- 通过dapm_widget_update函数，切换触发该次状态变化的widget的kcontrol中的寄存器值，对应的结果就是：改变音频路径。
-
- 通过dapm_seq_run函数，处理up_list中的widget，使它们按定义好的顺序依次上电。
- 对每个dapm context发出状态改变回调。
- 适当的延时，防止pop-pop声。

dapm_power_one_widget

dapm_power_widgets的第一步，就是遍历dapm_dirty链表，对每个链表中的widget调用dapm_power_one_widget，把需要上电和需要下电的widget分别加入到up_list和down_list链表中，同时，他还会

把受到影响的邻居widget再次加入到dapm_dirty链表的末尾，通过这个动作，声卡中所以受到影响的widget都会被“感染”，依次被加到dapm_dirty链表，然后依次被执行dapm_power_one_widget函数。下图展示了dapm_power_one_widget函数的调用序列：



图二　dapm_power_one_widget函数调用过程

- 通过dapm_widget_power_check，调用widget的power_check回调函数，获得该widget新的电源状态。
- 调用dapm_widget_set_power，“感染”与之相连的邻居widget。
  - 遍历source widget，通过dapm_widget_set_peer_power函数，把处于连接状态的source widget加入dapm_dirty链表中。
  - 遍历sink widget，通过dapm_widget_set_peer_power函数，把处于连接状态的sink widget加入dapm_dirty链表中。
- 根据第一步得到的新的电源状态，把widget加入到up_list或down_list链表中。

可见，通过该函数，一个widget的状态改变，邻居widget会受到“感染”而被加入到dapm_dirty链表的末尾，所以扫描到链表的末尾时，邻居widget也会执行同样的操作，从而“感染”邻居的邻居，直到没有新的widget被加入dapm_dirty链表为止，这时，所有受到影响的widget都被加入到up_list或down_li链表中，等待后续的上下电操作。这就是文章的标题所说的那样：**牵一发而动全身**。

dapm_seq_run

参看图一的上电过程，当所有需要上电或下电的widget都被加入到dapm_dirty链表后，接着会通过dapm_seq_run处理down_list链表上的widget，把该链表上的widget按顺序下电，然后通过dapm_widget_update更新widget中的kcontrol（这个kcontrol通常就是触发本次状态改变的触发源），接着又通过apm_seq_run处理up_list链表上的widget，把该链表上的widget按顺序上电。最终的上电或下电操作需要通过codec的寄存器来实现，因为定义widget时，如果这是一个带电源控制的widget，我们必须提供reg/shift等字段的设置值，如果该widget无需寄存器控制电源状态，则reg字段必须赋值为：

- SND_SOC_NOPM　　(该宏定义的实际值是-1)

具体实现上，dapm框架使用了一点技巧：如果位于同一个上下电顺序的几个widget使用了同一个寄存器地址（一个寄存器可能使用不同的位来控制不同的widget的电源状态），dapm_seq_run通过dapm_seq_run_coalesced函数合并这几个widget的变更，然后只需要把合并后的值一次写入寄存器即可。

dapm kcontrol的put回调

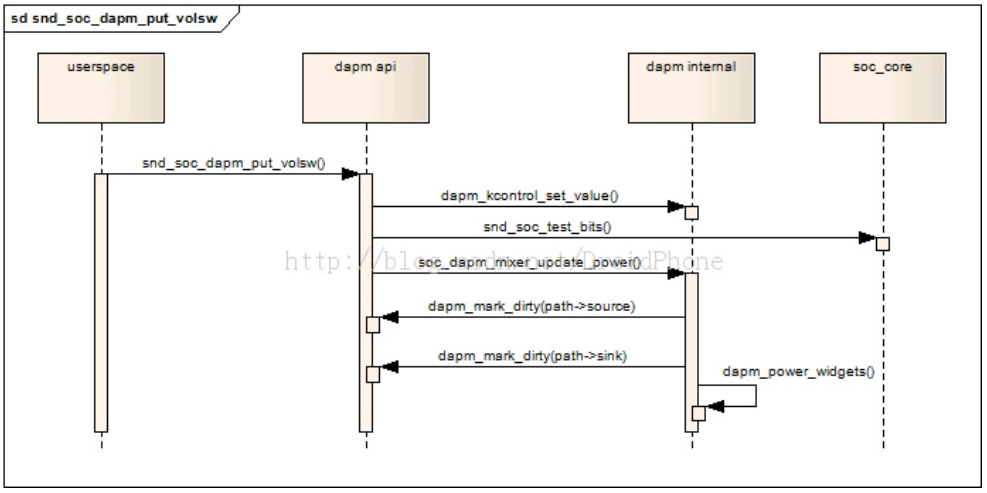
上面我们已经讨论了如何判断一个widget是否需要上电，以及widget的上电过程，一个widget的状态改变如何传递到整个音频路径上的所有widget。这些过程总是需要一个起始点：是谁触动了dapm，使得它需要执行上述的扫描和上电过程？事实上，以下几种情况可以触发dapm发起一次扫描操作：

- 声卡初始化阶段，snd_soc_dapm_new_widgets函数创建widget包含的kcontrol后，会触发一次扫描操作。
- 用户空间的应用程序修改了widget中包含的dapm kcontrol的配置值时，会触发一次扫描操作。
- pcm的打开或关闭，会通过音频流widget触发一次扫描操作。
- 驱动程序在改变了某个widget并把它加入到dapm_dirty链表后，主动调用snd_soc_dapm_sync函数触发扫描操作。

这里我们主要讨论一下第二种，用户空间对kcontrol的修改，最终都会调用到kcontrol的put回调函数。对于常用的dapm kcontrol，系统已经为我们定义好了它们的put回调函数：

- snd_soc_dapm_put_volsw　　mixer类型的dapm kcontrol使用的put回调
- snd_soc_dapm_put_enum_double　　mux类型的dapm kcontrol使用的put回调
- snd_soc_dapm_put_enum_virt　　虚拟mux类型的dapm kcontrol使用的put回调
- snd_soc_dapm_put_value_enum_double　　控制值不连续的mux类型的dapm kcontrol使用的put回调
- snd_soc_dapm_put_pin_switch　　引脚类dapm kcontrol使用的put回调

我们以mixer类型的dapm kcontrol的put回调讲解一下触发的过程：



图三　mixer dapm kcontrol的put回调

```cpp
int snd_soc_dapm_put_volsw(struct snd_kcontrol *kcontrol,
        struct snd_ctl_elem_value *ucontrol)
{
        struct snd_soc_codec *codec = snd_soc_dapm_kcontrol_codec(kcontrol);
        struct snd_soc_card *card = codec->card;
        struct soc_mixer_control *mc =
                (struct soc_mixer_control *)kcontrol->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;
        unsigned int val;
        int connect, change;
        struct snd_soc_dapm_update update;
        ......
        /* 从参数中取出要设置的新的设置值 */
        val = (ucontrol->value.integer.value[0] & mask);
        connect = !!val;

        if (invert)
                val = max - val;

        /* 把新的设置值缓存到kcontrol的影子widget中 */
        dapm_kcontrol_set_value(kcontrol, val);

        mask = mask << shift;
        val = val << shift;
        /* 和实际寄存器中的值进行对比，不一样时才会触发寄存器的写入 */
        /* 寄存器通常都会通过regmap机制进行缓存，所以这个测试不会发生实际的寄存器读取操作 */
        /* 这里只是触发，真正的寄存器写入操作要在扫描完dapm_dirty链表后的执行 */
        change = snd_soc_test_bits(codec, reg, mask, val);
        if (change) {
                update.kcontrol = kcontrol;
                update.reg = reg;
                update.mask = mask;
                update.val = val;

                card->update = &update;
                /* 触发dapm的上下电扫描过程 */
                soc_dapm_mixer_update_power(card, kcontrol, connect);

```

```
43.                 card->update = NULL;
44.         }
45.         ......
46.         return change;
47. }
```

其中的dapm_kcontrol_set_value函数用于把设置值缓存到kcontrol对应的影子widget，影子widget是为了实现autodisable特性而创建的一个虚拟widget，影子widget的输出连接到kcontrol的source widget，影子widget的寄存器被设置为和kcontrol一样的寄存器地址，这样当source widget被关闭时，会触发影子widget被关闭，其作用就是kcontrol也被自动关闭从而在物理上断开与source widget的连接，但是此时逻辑连接依然有效，dapm依然认为它们是连接在一起的。

触发dapm进行电源状态扫描关键的函数是soc_dapm_mixer_update_power：

```
[cpp]

01. static int soc_dapm_mixer_update_power(struct snd_soc_card *card,
02.                                       struct snd_kcontrol *kcontrol, int connect)
03. {
04.         struct snd_soc_dapm_path *path;
05.         int found = 0;
06.
07.         /* 更新所有和该kcontrol对应输入端相连的path的connect字段 */
08.         dapm_kcontrol_for_each_path(path, kcontrol) {
09.                 found = 1;
10.                 path->connect = connect;
11.                 /*把自己和相连的source widget加入到dirty链表中*/
12.                 dapm_mark_dirty(path->source, "mixer connection");
13.                 dapm_mark_dirty(path->sink, "mixer update");
14.         }
15.         /* 发起dapm_dirty链表扫描和上下电过程 */
16.         if (found)
17.                 dapm_power_widgets(card, SND_SOC_DAPM_STREAM_NOP);
18.
19.         return found;
20. }
```

最终，还是通过dapm_power_widgets函数，触发整个音频路径的扫描过程，这个函数执行后，因为kcontrol的状态改变，被断开连接的音频路径上的所有widget被按顺序下电，而重新连上的音频路径上的所有widget被顺序地上电，所以，尽管我们只改变了mixer kcontrol中的一个输入端的连接状态，所有相关的widget的电源状态都会被重新设定，这一切，都是自动完成的，对用户空间的应用程序完全透明，实现了dapm的原本设计目标。

更多 0

上一篇：ALSA声卡驱动中的DAPM详解之五：建立widget之间的连接关系

下一篇：ALSA声卡驱动中的DAPM详解之七：dapm事件机制（dapm event）

查看评论

5楼 wsc_168 昨天 17:57发表

楼主,您好：
现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描到了wm8962-audio, I2S接口也连接成功了。但系统启动的最后阶段就出现了以下的错误信息：
POST_PMU: HPOUTL PGA event failed: -5
POST_PMU: HPOUTR PGA event failed: -5
wm8962 0-0034: DC servo timed out
Failed to set MUTE: -5
Failed to set MUTE: -5
您觉得可能是那个地方的问题，导致上面的错误。麻烦您指点指点，谢谢了。

4楼 yelite 2013-12-16 16:10发表

楼主，准备出书吧，肯定有出版社找你。
感谢分享，真是下了功夫。

3楼 fshh520 2013-11-13 22:58发表

确实好文，博主可以考虑出一本书，不用太厚，就把你博客上的东西整理一下了就好了，另外请教一下博主，我看你画的图很好看，是用什么软件画的啊？

Re: DroidPhone 2013-11-14 13:25发表

回复fshh520：嗯，好脑袋不如硬笔头，写一遍之后，一来可以分享，二来以后忘了也可以翻出来看看，出书就免了，图是用EA画的。

2楼 hustljh 2013-11-07 19:50发表

图文并茂，深入浅出，好文！！！请教下博主，Android系统进入休眠前后，一般会对widget做操作吗?会做哪些操作？

Re: DroidPhone 2013-11-08 00:23发表

回复hustljh：我也是边读代码边写文章，还没有涉及休眠状态，后续的文章如果有机会，也许会分析一下。

1楼 jeffreyliu 2013-11-05 21:13发表

期待中。。。。。

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

专区推荐内容

多线程编程3 - NSOpera...
数据中心：提高温度需要什么条件？
介绍一种服务器缓存结构-多级 H...
一次编写，随处运行 英特尔HTM...
开源 - 开放式虚拟化解决方案
绳命不息，代码不止

更多招聘职位 **我公司职位也要出现在这里**

【广州友魄信息科技有限公司】Asp.net（C#）开发工程师
【北京奥鹏远程教育中心有限公司】专职/兼职IT培训讲师（Web前端开发，PHP，3G开发方向）
【langoon】java架构师
【北京富邦天成信息技术有限公司】java高级工程师
【南京科莱斯企业管理咨询有限公司】Flex开发工程师

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

个人资料

DroidPhone

访问： 382370次
积分： 3435分
排名： 第2116名

原创： 46篇 转载： 0篇
译文： 4篇 评论： 356条

ALSA声卡驱动中的DAPM详解之七：dapm事件机制（dapm event）

分类： Linux音频子系统 2013-11-09 01:05 516人阅读 评论(5) 收藏 举报

linux audio driver widget alsa dapm

前面的六篇文章，我们已经讨论了dapm关于动态电源管理的有关知识，包括widget的创建和初始化，widget之间的连接以及widget的上下电顺序等等。本章我们准备讨论dapm框架中的另一个机制：事件机制。通过dapm事件机制，widget可以对它所关心的dapm事件做出反应，这种机制对于扩充widget的能力非常有用，例如，对于那些位于codec之外的widget，好像喇叭功放、外部的前置放大器等等，由于不是使用codec内部的寄存器进行电源控制，我们就必须利用dapm的事件机制，获得相应的上下电事件，从而可以定制widget自身的电源控制功能。

dapm event的种类

dapm目前为我们定义了9种dapm event，他们分别是：

事件类型	说明
SND_SOC_DAPM_PRE_PMU	widget要上电前发出的事件
SND_SOC_DAPM_POST_PMU	widget要上电后发出的事件
SND_SOC_DAPM_PRE_PMD	widget要下电前发出的事件
SND_SOC_DAPM_POST_PMD	widget要下电后发出的事件
SND_SOC_DAPM_PRE_REG	音频路径设置之前发出的事件
SND_SOC_DAPM_POST_REG	音频路径设置之后发出的事件
SND_SOC_DAPM_WILL_PMU	在处理up_list链表之前发出的事件
SND_SOC_DAPM_WILL_PMD	在处理down_list链表之前发出的事件
SND_SOC_DAPM_PRE_POST_PMD	SND_SOC_DAPM_PRE_PMD和SND_SOC_DAPM_POST_PMD的合并

前8种每种占据一个位，所以，我们可以在一个整数中表达多个我们需要关心的dapm事件，只要把它们按位或进行合并即可。

widget的event回调函数

ALSA声卡驱动中的DAPM详解之二：widget-具备路径和电源管理信息的kcontrol中，我们已经介绍过代表widget的snd_soc_widget结构，在这个结构体中，有一个event字段用于保存该widget的事件回调函数，同时，event_flags字段用于保存该widget需要关心的dapm事件种类，只有event_flags字段中相应的事件位被设置了

Linux ALSA声卡驱动之三 (18248)
Android中的sp和wp指针 (17112)
Linux ALSA声卡驱动之七 (13786)
Android SurfaceFlinger中 (13061)
(12550)

评论排行

Android Audio System 之 (49)
Linux ALSA声卡驱动之八 (30)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (18)
Linux ALSA声卡驱动之三 (16)
Android Audio System 之 (16)
Linux中断（interrupt）子 (15)
Android中的sp和wp指针 (13)
Linux中断（interrupt）子 (12)
Android SurfaceFlinger中 (11)

推荐文章

* SharePoint 2010/2013 使用Javascript来判断权限的三种方法
* 坚持前进的方向：总结2013，规划2014
* 创业者那些鲜为人知的事情
* ListView具有多种item布局——实现微信对话列
* 实现自己的类加载时，重写方法loadClass与findClass的区别
* GDAL影像投影转换

最新评论

Linux输入子系统：多点触控协议
gocy123: 很有用，多谢分享
ALSA声卡驱动中的DAPM详解之
wsc_168: 楼主,您好： 现在正在移植wm8962的驱动，遇到了一些问题，请教一下您。串口信息显示已经扫描...
Linux ALSA声卡驱动之五：移动
slcsss: @DroidPhone:感谢您的回复，我是新手，想问下这个配置的具体位置在哪里？
Linux ALSA声卡驱动之五：移动
DroidPhone: @u013222557:这种情况通常是你的codec中的音频路径把Mic至HP的路径被打开了，请仔细...
Linux ALSA声卡驱动之五：移动
slcsss: 您好： 我是一名在读研究生，最近在ALSA架构下搞嵌入式音频程序开发，遇到了一个棘手的问...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:和电源管理和音频路径相关的control需要定义为dapm control（...
ALSA声卡驱动中的DAPM详解之
elliepfsang: 大侠，你好！ 这两天把您的文章1-7 看了一遍，关于control这个概念在您的文章中有提到过多次...
ALSA声卡驱动中的DAPM详解之
elliepfsang: @DroidPhone:因为这个是wm8962的machine上的现有代码，但如果我要porting...

的事件才会发到event回调函数中进行处理。

我们知道，dapm为我们提供了常用widget的定义辅助宏，使用以下这几种辅助宏定义widget时，默认需要我们提供dapm event回调函数

- SND_SOC_DAPM_MIC
- SND_SOC_DAPM_HP
- SND_SOC_DAPM_SPK
- SND_SOC_DAPM_LINE

这些widget都是位于codec外部的器件，它们无法使用通用的寄存器操作来控制widget的电源状态，所以需要我们提供event回调函数。以下的例子来自dapm的内核文档，外部的喇叭功放通过CORGI_GPIO_APM_ON这个gpio来控制它的电源状态：

```cpp
/* 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);
```

另外，我们也可以通过以下这些带"_E"后缀的辅助宏版本来定义需要dapm事件的widget：

- SND_SOC_DAPM_PGA_E
- SND_SOC_DAPM_OUT_DRV_E
- SND_SOC_DAPM_MIXER_E
- SND_SOC_DAPM_MIXER_NAMED_CTL_E
- SND_SOC_DAPM_SWITCH_E
- SND_SOC_DAPM_MUX_E
- SND_SOC_DAPM_VIRT_MUX_E

触发dapm event

我们已经定义好了带有event回调的widget，那么，在那里触发这些dapm event？答案是：在dapm_power_widgets函数的处理过程中，dapm_power_widgets函数我们已经在ALSA声卡驱动中的DAPM详解之六：精髓所在，牵一发而动全身中做了详细的分析，其中，在所有需要处理电源变化的widget被分别放入up_list和down_list链表后，会相应地发出各种dapm事件：

```cpp
static int dapm_power_widgets(struct snd_soc_card *card, int event)
{
        ......
        list_for_each_entry(w, &down_list, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_WILL_PMD);
        }

        list_for_each_entry(w, &up_list, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_WILL_PMU);
        }

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

        dapm_widget_update(card);

        /* Now power up. */
        dapm_seq_run(card, &up_list, event, true);
        ......
}
```

可见，在真正地进行上电和下电之前，dapm向down_list链表中的每个widget发出SND_SOC_DAPM_WILL_PMD事件，而向up_list链表中的每个widget发出SND_SOC_DAPM_WILL_PMU事件。在处理上下电的函数

Linux ALSA声卡驱动之六：ASoC
elliepfsang: @DroidPhone:关于snd_soc_dai_link结构中的代码注释中struct snd_...
ALSA声卡驱动中的DAPM详解之
DroidPhone: @u012389631:这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和co...

dapm_seq_run中，会调用dapm_seq_run_coalesced函数执行真正的寄存器操作，进行widget的电源控制，dapm_seq_run_coalesced也会发出另外几种dapm事件：

```cpp
static void dapm_seq_run_coalesced(struct snd_soc_card *card,
                                   struct list_head *pending)
{
        ......
        list_for_each_entry(w, pending, power_list) {
                ......
                /* Check for events */
                dapm_seq_check_event(card, w, SND_SOC_DAPM_PRE_PMU);
                dapm_seq_check_event(card, w, SND_SOC_DAPM_PRE_PMD);
        }

        if (reg >= 0) {
                ......
                pop_wait(card->pop_time);
                soc_widget_update_bits_locked(w, reg, mask, value);
        }

        list_for_each_entry(w, pending, power_list) {
                dapm_seq_check_event(card, w, SND_SOC_DAPM_POST_PMU);
                dapm_seq_check_event(card, w, SND_SOC_DAPM_POST_PMD);
        }
}
```

另外，负责更新音频路径的dapm_widget_update函数中也会发出dapm事件：

```cpp
static void dapm_widget_update(struct snd_soc_card *card)
{
        struct snd_soc_dapm_update *update = card->update;
        struct snd_soc_dapm_widget_list *wlist;
        struct snd_soc_dapm_widget *w = NULL;
        unsigned int wi;
        int ret;

        if (!update || !dapm_kcontrol_is_powered(update->kcontrol))
                return;

        wlist = dapm_kcontrol_get_wlist(update->kcontrol);

        for (wi = 0; wi < wlist->num_widgets; wi++) {
                w = wlist->widgets[wi];

                if (w->event && (w->event_flags & SND_SOC_DAPM_PRE_REG)) {
                        ret = w->event(w, update->kcontrol, SND_SOC_DAPM_PRE_REG);
                        ......
                }
        }

        ......
        /* 更新kcontrol的值，改变音频路径 */
        ret = soc_widget_update_bits_locked(w, update->reg, update->mask,
                                    update->val);
        ......

        for (wi = 0; wi < wlist->num_widgets; wi++) {
                w = wlist->widgets[wi];

                if (w->event && (w->event_flags & SND_SOC_DAPM_POST_REG)) {
                        ret = w->event(w, update->kcontrol, SND_SOC_DAPM_POST_REG);
                        ......
                }
        }
}
```

可见，改变路径的前后，分别发出了SND_SOC_DAPM_PRE_REG事件和SND_SOC_DAPM_POST_REG事件。

dai widget与stream widget

dai widget　在ALSA声卡驱动中的DAPM详解之四：在驱动程序中初始化并注册widget和route一文中，我们已经讨论过dai widget，dai widget又分为cpu dai widget和codec dai widget，它们在machine驱动分别匹配上相应的

codec和platform后，由soc_probe_platform和soc_probe_codec这两个函数通过调用dapm的api函数:

- snd_soc_dapm_new_dai_widgets

来创建的，通常会为playback和capture各自创建一个dai widget，他们的类型分别是：

- snd_soc_dapm_dai_in　对应playback dai
- snd_soc_dapm_dai_out　对应capture dai

另外，dai widget的名字是使用stream name来命名的，他通常来自snd_soc_dai_driver中的stream_name字段。dai widget的sname字段也使用同样的名字。

stream widget　stream widget通常是指那些要处理音频流数据的widget，它们包含以下这几种类型：

- snd_soc_dapm_aif_in　用SND_SOC_DAPM_AIF_IN辅助宏定义
- snd_soc_dapm_aif_out　用SND_SOC_DAPM_AIF_OUT辅助宏定义
- snd_soc_dapm_dac　用SND_SOC_DAPM_AIF_DAC辅助宏定义
- snd_soc_dapm_adc　用SND_SOC_DAPM_AIF_ADC辅助宏定义

对于这几种widget，我们除了要指定widget的名字外，还要指定他对应的stream的名字，保存在widget的sname字段中。

连接dai widget和stream widget

默认情况下，驱动不会通过snd_soc_route来主动定义dai widget和stream widget之间的连接关系，实际上，他们之间的连接关系是由ASoc负责的，在声卡的初始化函数中，使用snd_soc_dapm_link_dai_widgets函数来建立他们之间的连接关系：

```cpp
static int snd_soc_instantiate_card(struct snd_soc_card *card)
{
        ......
        /* card bind complete so register a sound card */
        ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
                        card->owner, 0, &card->snd_card);
        ......
        if (card->dapm_widgets)
                snd_soc_dapm_new_controls(&card->dapm, card->dapm_widgets,
                                          card->num_dapm_widgets);
        /*  建立dai widget和stream widget之间的连接关系  */
        snd_soc_dapm_link_dai_widgets(card);
        ......
        if (card->controls)
                snd_soc_add_card_controls(card, card->controls, card->num_controls);
        ......
        if (card->dapm_routes)
                snd_soc_dapm_add_routes(&card->dapm, card->dapm_routes,
                                        card->num_dapm_routes);
        ......
        if (card->fully_routed)
                list_for_each_entry(codec, &card->codec_dev_list, card_list)
                        snd_soc_dapm_auto_nc_codec_pins(codec);

        snd_soc_dapm_new_widgets(card);

        ret = snd_card_register(card->snd_card);
        ......
        return 0;
}
```

我们再来分析一下snd_soc_dapm_link_dai_widgets函数，看看它是如何连接这两种widget的，它先是遍历声卡中所有的widget，找出类型为snd_soc_dapm_dai_in和snd_soc_dapm_dai_out的widget，通过widget的priv字段，取出widget对应的snd_soc_dai结构指针：

```cpp
int snd_soc_dapm_link_dai_widgets(struct snd_soc_card *card)
{
        struct snd_soc_dapm_widget *dai_w, *w;
        struct snd_soc_dai *dai;

        /* For each DAI widget... */
        list_for_each_entry(dai_w, &card->widgets, list) {
                switch (dai_w->id) {
```

```
                case snd_soc_dapm_dai_in:
                case snd_soc_dapm_dai_out:
                        break;
                default:
                        continue;
                }

                dai = dai_w->priv;
```

接着，再次从头遍历声卡中所有的widget，找出能与dai widget相连接的stream widget，第一个前提条件是这两个widget必须位于同一个dapm context中：

```cpp
/* ...find all widgets with the same stream and link them */
list_for_each_entry(w, &card->widgets, list) {
        if (w->dapm != dai_w->dapm)
                continue;
```

dai widget不会与dai widget相连，所以跳过它们：

```cpp
switch (w->id) {
case snd_soc_dapm_dai_in:
case snd_soc_dapm_dai_out:
        continue;
default:
        break;
}
```

dai widget的名字没有出现在要连接的widget的stream name中，跳过这个widget：

```cpp
if (!w->sname || !strstr(w->sname, dai_w->name))
        continue;
```

如果widget的stream name包含了dai的stream name，则匹配成功，连接这两个widget：

```cpp
                if (dai->driver->playback.stream_name &&
                    strstr(w->sname,
                           dai->driver->playback.stream_name)) {
                        dev_dbg(dai->dev, "%s -> %s\n",
                                dai->playback_widget->name, w->name);

                        snd_soc_dapm_add_path(w->dapm,
                                dai->playback_widget, w, NULL, NULL);
                }

                if (dai->driver->capture.stream_name &&
                    strstr(w->sname,
                           dai->driver->capture.stream_name)) {
                        dev_dbg(dai->dev, "%s -> %s\n",
                                w->name, dai->capture_widget->name);

                        snd_soc_dapm_add_path(w->dapm, w,
                                dai->capture_widget, NULL, NULL);
                }
        }
}

return 0;
```

由此可见，dai widget和stream widget是通过stream name进行匹配的，所以，我们在定义codec的stream widget时，它们的stream name必须要包含dai的stream name，这样才能让ASoc自动把这两种widget连接在一起，只有把它们连接在一起，ASoc中的播放、录音和停止等事件，才能通过dai widget传递到codec中，使得codec中的widget能根据目前的播放状态，动态地开启或关闭音频路径上所有widget的电源。我们看看wm8993中的例子：

```cpp
SND_SOC_DAPM_AIF_OUT("AIFOUTL", "Capture", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_OUT("AIFOUTR", "Capture", 1, SND_SOC_NOPM, 0, 0),

```

```
SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),
SND_SOC_DAPM_AIF_IN("AIFINR", "Playback", 1, SND_SOC_NOPM, 0, 0),
```

分别定义了左右声道两个stream name为Capture和Playback的stream widget。对应的dai driver结构定义如下：

```
[cpp]

static struct snd_soc_dai_driver wm8993_dai = {
        .name = "wm8993-hifi",
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = WM8993_RATES,
                .formats = WM8993_FORMATS,
                .sig_bits = 24,
        },
        .capture = {
                 .stream_name = "Capture",
                 .channels_min = 1,
                 .channels_max = 2,
                 .rates = WM8993_RATES,
                 .formats = WM8993_FORMATS,
                 .sig_bits = 24,
         },
        .ops = &wm8993_ops,
        .symmetric_rates = 1,
};
```

可见，它们的stream name是一样的，声卡初始化阶段会把它们连接在一起。需要注意的是，如果我们定义了snd_soc_dapm_aif_in和snd_soc_dapm_aif_out类型的stream widget，并指定了他们的stream name，在定义DAC或ADC对应的widget时，它们的stream name最好不要也使用相同的名字，否则，dai widget即会连接上AIF，也会连接上DAC/ADC，造成音频路径的混乱：

```
[cpp]

SND_SOC_DAPM_ADC("ADCL", NULL, WM8993_POWER_MANAGEMENT_2, 1, 0),
SND_SOC_DAPM_ADC("ADCR", NULL, WM8993_POWER_MANAGEMENT_2, 0, 0),

SND_SOC_DAPM_DAC("DACL", NULL, WM8993_POWER_MANAGEMENT_3, 1, 0),
SND_SOC_DAPM_DAC("DACR", NULL, WM8993_POWER_MANAGEMENT_3, 0, 0),
```

stream event

把dai widget和stream widget连接在一起，就是为了能把ASoc中的pcm处理部分和dapm进行关联，pcm的处理过程中，会通过发出stream event来通知dapm系统，重新扫描并调整音频路径上各个widget的电源状态，目前dapm提供了以下几种stream event:

```
[cpp]

/* dapm stream operations */
#define SND_SOC_DAPM_STREAM_NOP                 0x0
#define SND_SOC_DAPM_STREAM_START               0x1
#define SND_SOC_DAPM_STREAM_STOP                0x2
#define SND_SOC_DAPM_STREAM_SUSPEND             0x4
#define SND_SOC_DAPM_STREAM_RESUME              0x8
#define SND_SOC_DAPM_STREAM_PAUSE_PUSH  0x10
#define SND_SOC_DAPM_STREAM_PAUSE_RELEASE       0x20
```

比如，在soc_pcm_prepare函数中，会发出SND_SOC_DAPM_STREAM_START事件：

```
[cpp]

snd_soc_dapm_stream_event(rtd, substream->stream,
                SND_SOC_DAPM_STREAM_START);
```

而在soc_pcm_close函数中，会发出SND_SOC_DAPM_STREAM_STOP事件：

```
[cpp]

snd_soc_dapm_stream_event(rtd,
                          SNDRV_PCM_STREAM_PLAYBACK,
                          SND_SOC_DAPM_STREAM_STOP);
```

snd_soc_dapm_stream_event函数最终会使用soc_dapm_stream_event函数来完成具体的工作：

```cpp
static void soc_dapm_stream_event(struct snd_soc_pcm_runtime *rtd, int stream,
        int event)
{

        struct snd_soc_dapm_widget *w_cpu, *w_codec;
        struct snd_soc_dai *cpu_dai = rtd->cpu_dai;
        struct snd_soc_dai *codec_dai = rtd->codec_dai;

        if (stream == SNDRV_PCM_STREAM_PLAYBACK) {
                w_cpu = cpu_dai->playback_widget;
                w_codec = codec_dai->playback_widget;
        } else {
                w_cpu = cpu_dai->capture_widget;
                w_codec = codec_dai->capture_widget;
        }
```

该函数首先从snd_soc_pcm_runtime结构中取出cpu dai widget和codec dai widget，接下来：

```cpp
if (w_cpu) {

        dapm_mark_dirty(w_cpu, "stream event");

        switch (event) {
        case SND_SOC_DAPM_STREAM_START:
                w_cpu->active = 1;
                break;
        case SND_SOC_DAPM_STREAM_STOP:
                w_cpu->active = 0;
                break;
        case SND_SOC_DAPM_STREAM_SUSPEND:
        case SND_SOC_DAPM_STREAM_RESUME:
        case SND_SOC_DAPM_STREAM_PAUSE_PUSH:
        case SND_SOC_DAPM_STREAM_PAUSE_RELEASE:
                break;
        }
}
```

把cpu dai widget加入到dapm_dirty链表中，根据stream event的类型，把cpu dai widget设定为激活状态或非激活状态，接下来，对codec dai widget做出同样的处理：

```cpp
if (w_codec) {

        dapm_mark_dirty(w_codec, "stream event");

        switch (event) {
        case SND_SOC_DAPM_STREAM_START:
                w_codec->active = 1;
                break;
        case SND_SOC_DAPM_STREAM_STOP:
                w_codec->active = 0;
                break;
        case SND_SOC_DAPM_STREAM_SUSPEND:
        case SND_SOC_DAPM_STREAM_RESUME:
        case SND_SOC_DAPM_STREAM_PAUSE_PUSH:
        case SND_SOC_DAPM_STREAM_PAUSE_RELEASE:
                break;
        }
}
```

最后，它调用了我们熟悉的dapm_power_widgets函数：

```cpp
dapm_power_widgets(rtd->card, event);
```

因为dai widget和codec上的stream widget是相连的，所以，dai widget的激活状态改变，会沿着音频路径传递到路径上的所有widget，等dapm_power_widgets返回后，如果发出的是SND_SOC_DAPM_STREAM_START事件，路径上的所有widget会处于上电状态，保证音频数据流的顺利播放，如果发出的是SND_SOC_DAPM_STREAM_STOP事件，路径上的所有widget会处于下电状态，保证最小的功耗水平。

查看评论

2楼 elliepfsang 2013-12-16 21:40发表

大侠
关于
SND_SOC_DAPM_AIF_IN("AIFINL", "Playback", 0, SND_SOC_NOPM, 0, 0),中的playback我是找到了
前面提问时 写错了，我是想问关于AIFINL之类的

小弟最近忙于porting wm8960的项目，正好在网上看到您的关于dapm的详解，虽然大致上看了一下，对于目前的我还是很有帮助的，关于在machine或者在codec中的widget定义有一个问题我没有弄明白

比如：
static const struct snd_soc_dapm_widget imx_dapm_widgets[] = {
SND_SOC_DAPM_HP("Headphone Jack", NULL),
SND_SOC_DAPM_SPK("Ext Spk", NULL),
SND_SOC_DAPM_MIC("AMIC", NULL),
SND_SOC_DAPM_MIC("DMIC", NULL),
};

其中的AMIC，DMIC，headphone jack等，这些字符串是根据什么来命名的，
我一开始以为是codec中会有对应的字符串，但是从codec代码中没有很明确的标明，而且codec中也有大量类似的字符串，然后看datasheet 还是没有发现对应关系，所以希望大侠能指导下。 谢谢了

Re: DroidPhone 2013-12-16 22:17发表

回复u012389631：这种名字根据实际的意义自己定义就好了，只要符合常识即可。不过通常还是会和codec中的描述相关。AMIC：模拟麦克风？DMIC：数字麦克风？

Re: elliepfsang 2013-12-16 22:24发表

回复DroidPhone：因为这个是wm8962的machine上的现有代码，但如果我要porting wm8960的machine代码，这个就无从下手了，目前我也只是参考8962的去做，感觉不明其原理，心里不踏实

1楼 Alivepea 2013-11-28 10:20发表

widget 的 event() 最重要的作用是可以接收上/下电按时序的消息，可用于防 pop/kick noise ..如果仅仅为了控制外部外部事件， 其 put() 方法就够了。

Re: DroidPhone 2013-11-28 12:37发表

回复Alivepea：嗯，widget对用户空间是透明的，所以widget本是没有put方法的，只有widget内前有kcontrol时，对应的kcontrol才有put方法，但是对于外部器件对应的widget，通常都没有对应的kcontrol。