个人资料

DroidPhone

访问： 1077195次
积分： 8768
等级： BLOG 6
排名： 第1448名

原创： 51篇 转载： 0篇
译文： 4篇 评论： 537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 (20)
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

2014年07月 (1)
2014年04月 (4)
2013年11月 (4)
2013年10月 (3)
2013年07月 (3)

展开

阅读排行

Linux ALSA声卡驱动之一 (74187)
Android Audio System 之 (58758)
Linux ALSA声卡驱动之二 (46387)
Android Audio System 之 (42720)
Linux ALSA声卡驱动之三 (41553)
Linux ALSA声卡驱动之六

ALSA声卡驱动中的DAPM详解之一：kcontrol

标签： alsa dapm linux 动态电源管理 音频驱动

2013-10-18 15:19 17126人阅读 评论(12) 收藏 举报

分类： Linux音频子系统（14）

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。

/***/
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) \
        ((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：

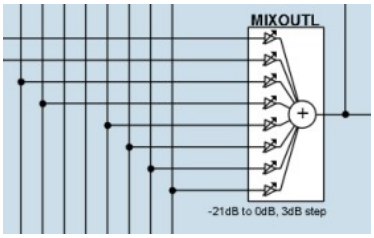

图1 Mixer混音器

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

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

我的同类文章

Linux音频子系统（14）

- ALSA声卡驱动中的DAPM详… 2013-11-09 阅读 8979
- ALSA声卡驱动中的DAPM详… 2013-11-04 阅读 9950
- ALSA声卡驱动中的DAPM详… 2013-11-04 阅读 12941
- ALSA声卡驱动中的DAPM详… 2013-11-01 阅读 10222
- ALSA声卡驱动中的DAPM详… 2013-10-24 阅读 12568
- ALSA声卡驱动中的DAPM详… 2013-10-23 阅读 10804
- Linux ALSA声卡驱动之八：… 2012-03-13 阅读 31315
- Linux ALSA声卡驱动之七：… 2012-02-23 阅读 36156
- Linux ALSA声卡驱动之六：… 2012-02-03 阅读 37515
- Linux ALSA声卡驱动之五：… 2012-01-17 阅读 26715

更多文章

猜你在找