个人资料

DroidPhone

访问： 1077198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详解之四：在驱动程序中初始化并注册widget和route

标签： linux widget audio driver dapm alsa

2013-11-01 22:41 10222人阅读 评论(2) 收藏 举报

分类： Linux音频子系统（14）

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;

        void (*seq_notifier)(struct snd_soc_dapm_context *,
                             enum snd_soc_dapm_type, int);

        struct device *dev; /* from parent - for debug */
```

Linux时间子系统之六：高 (37505)
Android Audio System 之 (36411)
Linux ALSA声卡驱动之七 (36317)
Linux ALSA声卡驱动之四 (36146)
(31714)

评论排行

Android Audio System 之 (56)
Linux ALSA声卡驱动之三 (42)
Linux ALSA声卡驱动之八 (35)
Linux时间子系统之六：高 (25)
Linux中断（interrupt）子 (24)
Android SurfaceFlinger中 (21)
Linux ALSA声卡驱动之二 (19)
Android Audio System 之 (18)
Linux ALSA声卡驱动之七 (17)
Linux中断（interrupt）子 (17)

推荐文章

* 致JavaScript也将征服的物联网世界
* 从苏宁电器到卡巴斯基：难忘的三年硕士时光
* 作为一名基层管理者如何利用情商管理自己和团队（一）
* Android CircleImageView圆形ImageView
* 高质量代码的命名法则

最新评论

Linux输入子系统：输入设备编程
u012839187: 看着没有问题。你要弄清楚的是新的内核有许多函数宏的变动。
Linux时间子系统之一：clock sou
zhqh100: 1.2 read回调函数时钟源本身不会产生中断，要获得时钟源的当前计数，只能通过主动调用它的rea...
Linux中断（interrupt）子系统之
liunix61: 大神！必须顶@！！！
Linux时间子系统之三：时间的维
Kevin_Smart: 学习了
Linux时间子系统之六：高精度定
Kevin_Smart: 像楼主说的，原则上，hrtimer是利用一个硬件计数器来实现的，所以精度才可以做到ns级别。硬件的计...
Linux中断（interrupt）子系统之
12期-马金兴: 恩，虽然这么多字但是我要好好学习一下
已知二叉树的前序遍历和中序遍
重修月: 很喜欢博主的文章，刚刚用豆约翰博客备份专家备份了您的全部博文。
Linux ALSA声卡驱动之三：PCM
灿哥哥: 学习了
Android Audio System 之三： A
ss0429: 楼主的文章写的很精炼，多谢分享~
Linux中断（interrupt）子系统之
KrisFei: 针对这句话有两个问题想讨论下：1. disable_irq()放在中断上半部有会导致死锁。2. 如果...

```
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]
```

```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。

上一篇　ALSA声卡驱动中的DAPM详解之三：如何定义各种widget
下一篇　ALSA声卡驱动中的DAPM详解之五：建立widget之间的连接关系

我的同类文章

Linux音频子系统（14）

- ALS… 2013-11-09 阅读 8979
- ALS… 2013-11-04 阅读 9950
- ALS… 2013-11-04 阅读 12941
- ALS… 2013-10-24 阅读 12568
- ALS…
- ALS…

参考知识库

Git知识库
1154 关注 | 409 收录

算法与数据结构知识库
2034 关注 | 3500 收录

大型网站架构知识库
2045 关注 | 532 收录

猜你在找

uboot的硬件驱动部分-2.10.uboot源码分析6
数据结构基础系列(5)：数组与广义表
Android底层技术：Linux驱动框架与开发
从零写Bootloader及移植uboot、linux内核、文件系统和
Linux设备驱动开发入门
ALSA声卡驱动中的DAPM详解之五建立widget之间的连接
ALSA声卡驱动中的DAPM详解之五建立widget之间的连接
ALSA声卡驱动中的DAPM详解之五建立widget之间的连接
ALSA声卡驱动中的DAPM详解之三如何定义各种widget
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 jeffrey_yinke 2014-08-21 08:40发表

Thank you very much for your clear explanation. These are best series of documents describing ASoC and DAPM I have ever read. Hope to be able to talk to author directly in details. Sorry for not able to type Chinese characters on my PC.

1楼 zhangyongsi 2014-04-22 17:19发表

我在我kenerl里怎么也找不到snd_soc_dapm_new_dai_widgets去创建widget。早soc_prode_XXX中直接调用snd_soc_dapm_new_controls进行添加的。写的还不错，看了几遍。还有很多地方不是很懂，在学习中。你有邮箱吗？留个邮箱，不懂得交流交流。

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

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .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 HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap