linux2.6 内核 ALSA 框架分析和 imx6-wm8960 移植

machine 部分：

1. 定义结构体 static struct　snd_soc_card　snd_soc_s3c24xx_uda134x
2. 定义平台的结构体 static struct platform_device *s3c24xx_uda134x_snd_device;
3. platform_set_drvdata(s3c24xx_uda134x_snd_device,&snd_soc_s3c24xx_uda134x);设置结构体私有数据
4. 注册 mahcine 里面的设 platform_device_add(s3c24xx_uda134x_snd_device);

I2S 部分：

1. 定义结构体 static struct　snd_soc_dai_driver　s3c24xx_i2s_dai
2. 定义平台 static const struct snd_soc_component_driver s3c24xx_i2s_component
3. snd_soc_register_component(&pdev->dev, &s3c24xx_i2s_component,&s3c24xx_i2s_dai, 1);
   //这个 snd_soc_register_component 里面包含了 snd_soc_register_dai

DMA 部分

1. 定义结构 static struct　snd_soc_platform_driver　samsung_asoc_platform
2. 注册 snd_soc_register_platform(dev, &samsung_asoc_platform);

Codec 部分：

1. 定义结构 static struct　snd_soc_codec_driver　soc_codec_dev_uda134x
2. 定义 I2S 结构 static struct　snd_soc_dai_driver　uda134x_dai
3. snd_soc_register_codec(&pdev->dev,&soc_codec_dev_uda134x, &uda134x_dai, 1);//注册

以上就是 S3C2440 平台和音频芯片的框架功能实现。

Machine 就是 snd_soc_card 这个结构

CPU 的 I2S 就是 snd_soc_dai_driver 这个结构体
DMA 就是 snd_soc_platform_driver 这个结构
Codec 芯片是 snd_soc_codec_driver 这个结构
codec 的 I2S 总线是 snd_soc_dai_driver 这个结构

以上结构的实现

Machine 部分的实现

```
static struct snd_soc_card snd_soc_s3c24xx_uda134x = {
	.name = "S3C24XX_UDA134X",
	.owner = THIS_MODULE,
	.dai_link = &s3c24xx_uda134x_dai_link,//主要就是这个
	.num_links = 1,
};
static struct snd_soc_dai_link s3c24xx_uda134x_dai_link = {
	.name = "UDA134X",//名字无所谓
	.stream_name = "UDA134X",//名字无所谓
	.codec_name = "uda134x-codec",//音频芯片驱动 uda134x.c 里面
	//platform_driver 的名字

	.codec_dai_name = "uda134x-hifi",//音频芯片驱动 uda134x.c 里面的 I2S 接口
	//snd_soc_dai_driver，dai 里面的名字

	.cpu_dai_name = "s3c24xx-iis",//s3c24xx-iis.c 里面 platform_driver 的名字
	.ops = &s3c24xx_uda134x_ops,// struct snd_soc_ops s3c24xx_uda134x_ops 单
独实现
	.platform_name  = "DMA",//这里应该是 DMA.C 里面 DMA 的名字
};

static struct snd_soc_ops s3c24xx_uda134x_ops = {
	.startup = s3c24xx_uda134x_startup,
	.shutdown = s3c24xx_uda134x_shutdown,
	.hw_params = s3c24xx_uda134x_hw_params,
};
```

I2S 部分实现：

```
static struct snd_soc_dai_driver s3c24xx_i2s_dai = {
	.probe = s3c24xx_i2s_probe,
	.suspend = s3c24xx_i2s_suspend,
	.resume = s3c24xx_i2s_resume,
	.playback = {
		.channels_min = 2,
		.channels_max = 2,
		.rates = S3C24XX_I2S_RATES,
		.formats = SNDRV_PCM_FMTBIT_S8 | SNDRV_PCM_FMTBIT_S16_LE,},
	.capture = {
		.channels_min = 2,
		.channels_max = 2,
		.rates = S3C24XX_I2S_RATES,
```

```
            .formats = SNDRV_PCM_FMTBIT_S8 | SNDRV_PCM_FMTBIT_S16_LE,},
        .ops = &s3c24xx_i2s_dai_ops,
};

static const struct snd_soc_dai_ops s3c24xx_i2s_dai_ops = {
        .trigger   = s3c24xx_i2s_trigger,
        .hw_params = s3c24xx_i2s_hw_params,
        .set_fmt = s3c24xx_i2s_set_fmt,
        .set_clkdiv    = s3c24xx_i2s_set_clkdiv,
        .set_sysclk    = s3c24xx_i2s_set_sysclk,
};
```

DMA 部分实现：

```
        static struct snd_soc_platform_driver samsung_asoc_platform = {
        .ops       = &dma_ops,
        .pcm_new    = dma_new,
        .pcm_free    = dma_free_dma_buffers,
};
```

Uda1341 部分实现：

```
        static const struct snd_soc_dai_ops uda134x_dai_ops = {
        .startup  = uda134x_startup,
        .shutdown    = uda134x_shutdown,
        .hw_params = uda134x_hw_params,
        .digital_mute      = uda134x_mute,
        .set_sysclk    = uda134x_set_dai_sysclk,
        .set_fmt = uda134x_set_dai_fmt,
};
static struct snd_soc_dai_driver uda134x_dai = {
        .name = "uda134x-hifi",
        /* playback capabilities */
        .playback = {
                .stream_name = "Playback",
                .channels_min = 1,
                .channels_max = 2,
                .rates = UDA134X_RATES,
                .formats = UDA134X_FORMATS,
        },
        /* capture capabilities */
        .capture = {
                .stream_name = "Capture",
                .channels_min = 1,
                .channels_max = 2,
```

```
            .rates = UDA134X_RATES,
            .formats = UDA134X_FORMATS,
        },
        /* pcm operations */
        .ops = &uda134x_dai_ops,
};


static struct snd_soc_codec_driver soc_codec_dev_uda134x = {
        .probe =            uda134x_soc_probe,
        .remove =           uda134x_soc_remove,
        .suspend =          uda134x_soc_suspend,
        .resume =           uda134x_soc_resume,
        .reg_cache_size = sizeof(uda134x_reg),
        .reg_word_size = sizeof(u8),
        .reg_cache_default = uda134x_reg,
        .reg_cache_step = 1,
        .read = uda134x_read_reg_cache,
        .write = uda134x_write,
        .set_bias_level = uda134x_set_bias_level,
        .dapm_widgets = uda134x_dapm_widgets,
        .num_dapm_widgets = ARRAY_SIZE(uda134x_dapm_widgets),
        .dapm_routes = uda134x_dapm_routes,
        .num_dapm_routes = ARRAY_SIZE(uda134x_dapm_routes),
};
```

以上就是四个部分的结构体实现过程，结构体里面的函数和参数都要另外实现这里就不多讲了。

在 linux 内核里面配置声卡驱动 make menuconfig
I2s 驱动在：

```
    -->Device Drivers
        -->Sound card support
            --> Advanced Linux Sound Architecture
                -->ALSA for SoC audio support
        在这个下面就要支持 2440 的 I2S 总线，如果是 IMX6 平台就是 Synopsys
    I2S Device Driver
```

Machine 驱动在：

```
    -->Device Drivers
        -->Sound card support
            --> Advanced Linux Sound Architecture
                -->ALSA for SoC audio support
```

Machine 文件 2440 在这个文件下面，如果是 imx6-wm8962 就在 SoC Audio for Freescale CPUs --->这个里面.

Machine 常用的四个 snd_soc 函数 API

snd_soc_dai_set_fmt()

snd_soc_dai_set_pll()

snd_soc_dai_set_sysclk()

snd_soc_dai_set_clkdiv()

snd_soc_dai_set_fmt 设置 I2S 的格式

```
if (data->is_codec_master)
        fmt = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF |
            SND_SOC_DAIFMT_CBM_CFM;
else
        fmt = SND_SOC_DAIFMT_I2S | SND_SOC_DAIFMT_NB_NF |
            SND_SOC_DAIFMT_CBS_CFS;
```

snd_soc_dai_set_fmt(cpu_dai, fmt);调用 snd_soc_dai_set_fmt 函数将格式注册进系统

snd_soc_dai_set_sysclk 设置编解码芯片 ADC 和 DAC 系统时钟

```
if (!(params_rate(params) % 11025))

snd_soc_dai_set_sysclk(codec_dai, WM8994_SYSCLK_MCLK1, 11289600,SND_SOC_CLOCK_IN);

else

snd_soc_dai_set_sysclk(codec_dai, WM8994_SYSCLK_MCLK1, 12288000,SND_SOC_CLOCK_IN);
```

snd_soc_dai_set_pll()设置 Codec 的 PLL 频率：11289600

```
snd_soc_dai_set_pll(codec_dai,1,0,priv->sysclk/2,pll_out)
```

```
/* 设置 Codec 的系统时钟：11289600*/
snd_soc_dai_set_clkdiv( codec_dai, WM8960_SYSCLKDIV, sysclk_div );

/* 设置采样频率（LRCK）的时钟，lrckclk= sysclk/1*256=44100 */
rsnd_soc_dai_set_clkdiv( codec_dai, WM8960_DACDIV, dacdiv);

/* 设置 D 类放大器时钟频率:dclk= sysclk/16 */
snd_soc_dai_set_clkdiv( codec_dai, WM8960_DCLKDIV,dclk);

/* 设置 BCLK 类放大器时钟频率:bitclk=sysclk/4 = 2822400 */
 snd_soc_dai_set_clkdiv( codec_dai, WM8960_BCLKDIV, bclk);
```

讲述 IMX6 平台移植 WM8960 音频芯片方法

首先 imx6 移植 WM8960 只需要修改 machine 文件和设备树文件就可以了。Machine 文件是 imx-wm8960.c 但是这个文件有好几个版本，我在 linux3.14.28 的内核上用了老版本内核里面的 imx-wm8960.c，就会出现有几个函数编译不过所以我在 linux3.14.28 内核上用的是 linux3.14.52 或者 linux4.0 内核版本里面的 imx-wm8960.c 文件。我将文件改成了 imx6-wm8960.c 所以拷贝到 linux3.14.28 内核后将 imx6-wm8960.c 改为 imx-wm8960.c，然后再 Makefile 和 Kconfig 里面加上配置参数。然后编译会出现一个错误 sound/soc/fsl/imx-wm8960.c:399:39: error: 'struct fsl_sai' has no member named 'is_stream_opened'。
这个错误的修改方式是因为 linux3.14.28 的 fsl_sai.h 没有定义 bool is_stream_opened[2]; 这个整形。这个整形在 linux3.14.52 或者 linux4.0 版本中的 fsl_sai.h 里面定义了的，所以将高版本的这个整形变量拷贝到 linux3.14.28 里面的 fsl_sai.h 就可以了。

设备树移植是按照 IMX6ul 里面内核的方法来移植，如果 imx6solo 要将 wm8962 换成 wm8960 的话，建议下载一个 linux3.14.52 或者 linux4.0 内核，里面有 imx6ul 的内核，也有 imx6，imx7 的内核，这样就很方便。

但是我已经在文件夹里面存放了 imx6 在 linux3.14.28 要移植 wm8960 的文件，所以也可以不用下载新版本的内核。

以上 wm8960 移植方法有许多地方有许多问题
所以请看下面新的移植方法

首先 imx6 移植 WM8960 只需要修改 machine 文件和设备树文件就可以了。
这个套路和上面相似
但是 machine 文件要下载 linux-3.10.17-imx 文件里面的 imx-wm8960.c
所以最好去 github 里面下载，地址为：https://github.com/xuhuashan/imx6q/tree/47a7ef6a73667bd855af3ae4b14f370800983de2/linux-3.10.17-imx/arch/arm/boot

然后 dts 设备树不要 linux-3.10.17-imx 目录里面的标准 IM6qdl-sabresd.c 设备树来，其他 linux 版本也是一样，所以要按照 linux-3.10.17-imx 目录里面的 **imx6q-tqe9.dts 天嵌的版本是最好的。**

这样 machine 和 dts 都已经移植好了，但是在编译的时候会报错。
1：报 machine 文件里面的函数错误 codec_dev = of_find_i2c_device_by_node(codec_np);

```
if (!codec_dev || !codec_dev->driver) {
        dev_err(&pdev->dev, "failed to find codec platform device\n");
        ret = -EINVAL;
         goto fail;
}
```

这个错误是因为 codec_dev->driver 是 linux3.10.17 版本里面支持的，linux3.14.28 版本必须要这样 codec_dev->dev.driver 也就是在 driver 前面加 dev，因为该结构又被封装了。

2：报错头文件问题，头文件 i2c 是 include<linux/of_i2c.h>,但是 linux3.14.28 在 i2c 头文件里面已经封装了 of 函数了所有不需要 of_i2c.h，直接 i2c.h 就行。
3：aplay 音乐的时候 I2S 总线只有 MCLK 有时钟，其他 IO 没有时钟，比如 BCLK, DACLRCLK, 没有时钟输出。还有 ADDAT,DACDAT 没有数据输出。这个问题是因为没有打开线性 pcm 开关命令
打开 amixer numid=41,iface=MIXER,name='Right Output Mixer PCM Playback Switch'
打开 amixer numid=44,iface=MIXER,name='Left Output Mixer PCM Playback Switch'
这样你就可以听到耳机输出正常的声音了。

在 aplay 播放的时候，驱动程序会进入 machine 文件里面执行 hw_params 回调函数，同时也要进入 codec 文件里面执行 set_pll 回调函数, set_fmt 回调函数，hw_params 回调函数。

在 arecord 录音的时候，驱动程序会进入 machine 文件里面执行 hw_params 回调函数，同时也要进入 codec 文件里面执行 set_pll 回调函数，set_fmt 回调函数，hw_params 回调函数。

所以主要修改这两个回调函数里面的寄存器设置

所以在 wm8960.c 里面

```
static const struct snd_soc_dai_ops wm8960_dai_ops = {
    .hw_params = wm8960_hw_params,        aplay 和 arecord 会执行该函数
    .digital_mute = wm8960_mute,
    .set_fmt = wm8960_set_dai_fmt,        aplay 和 arecord 会执行该函数
    .set_clkdiv = wm8960_set_dai_clkdiv,
    .set_pll = wm8960_set_dai_pll,        aplay 和 arecord 会执行该函数
};
```

在 imx-wm8960.c 里面

```
static struct snd_soc_ops imx_hifi_ops = {
    .startup = imx_hifi_startup,        执行 aplay 和 arecord 会第一个执行该函数
    .shutdown = imx_hifi_shutdown, ctrl+C 或者自动音乐播放完会执行该函数
    .hw_params = imx_hifi_hw_params,    aplay 和 arecord 会执行该函数
    .hw_free = imx_hifi_hw_free,
};
```

但是我加入了 TQ 的 imx-wm8960.c 可以播放声音为什么无法录音呢,这个问题，我自己没有找到，但是找方案商解决了这个问题，将目录下方案商修改好的 imx-wm8960.c 文件用来替换内核里面现有的 imx-wm8960.c 文件，路径 fsl-linux/sound/soc/fsl
然后将方案商改好的 wm8960.c 文件替换内核现有的 wm8960.c 文件，路径 fsl-linux/sound/soc/codecs
然后再修改设备树文件

```
/fsl-linux/arch/arm/boot/dts#
```
这是设备树在内核的路径

```
sound {
        compatible = "fsl,imx-audio-wm8960";
        model = "wm8960-audio";
        cpu-dai = <&ssi2>;
        audio-codec = <&codec>;
/*      asrc-controller = <&asrc>;*/
        codec-master;
/*      gpr = <&gpr>;*/
        /*
         * hp-det = <hp-det-pin hp-det-polarity>;
         * hp-det-pin: JD1 JD2  or JD3
         * hp-det-polarity = 0: hp detect high for headphone
         * hp-det-polarity = 1: hp detect high for speaker
         */
        hp-det = <1 0>;
        audio-routing =

                "Headphone Jack", "HP_L",
                "Headphone Jack", "HP_R",
                "Ext Spk", "SPK_LP",
                "Ext Spk", "SPK_LN",
                "Ext Spk", "SPK_RP",
                "Ext Spk", "SPK_RN",

                "LINPUT2", "AMIC",
                "RINPUT2", "AMIC";

        mux-int-port = <2>;
        mux-ext-port = <3>;
        hp-det-gpios = <&gpio7 8 1>;
        mic-det-gpios = <&gpio1 9 1>;

};
```

然后 make，将编译好的内核放入开发板，然后启动。
numid=41,iface=MIXER,name='Right Output Mixer PCM Playback Switch'
打开这个右边耳机的开关，aplay 的音乐才能从右耳机播放出来
numid=44,iface=MIXER,name='Left Output Mixer PCM Playback Switch'
打开这个左边耳机的开关，aplay 的音乐才能从左耳机播放出来
numid=9,iface=MIXER,name='Headphone Playback Volume'设置耳机输出音量大小
然后就可以用 aplay 播放音乐了。
下面我们来看看录音

arecord [-Dplughw:0,0] -r 44100 -f S16_LE -c 2 -d 5 record.wav

录音成功，能分别播放左右声道的声音

但是感觉录音的时候出现了很多背景噪音
numid=25,iface=MIXER,name='ALC Max Gain'
将 ALC Max Gain 降到 4 档，录音底噪就要下降很多

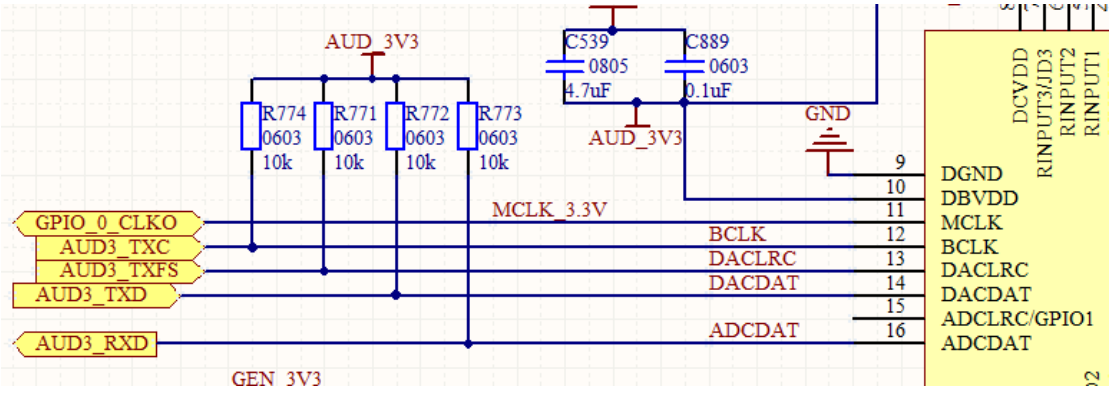
然后我们测试下直通，就是麦克风录入声音，然后从耳机直接输出出来。
numid=46,iface=MIXER,name='Left Output Mixer Boost Bypass Switch'
左声道麦克风和耳机直通
numid=43,iface=MIXER,name='Right Output Mixer Boost Bypass Switch'
右声道麦克风和耳机直通

arecord [-Dplughw:0,0] -r 44100 -f S16_LE -c 2 -d 5 record.wav

直通功能光是打开开关还不行，还要执行 arecod，然后直通才能启动，arecod 执行多久直通就只能执行这么久
numid=35,iface=MIXER,name='Left Output Mixer Boost Bypass Volume'
左声道麦克风直通录音声音大小
numid=37,iface=MIXER,name='Right Output Mixer Boost Bypass Volume'
右声道麦克风直通录音声音大小

WM8960 硬件设计注意事项

WM8960 芯片 DVDD 引脚如果接的 3.3V
那么 I2S 总线和 MCLK 必须接 3.3V，有一个没有接 3.3V 接的是 1.8V 就会出现 aplay 一会能播放，一会播放不了。最好使用上拉电阻接 3.3V
IMX6 公版的 WM8962 是将 CPU 的 MCLK3.3V，用芯片降到了 1.8V，导致我以为 I2S 接 3.3V，然后 MCLK 接的 1.8V。所以也出现了 aplay 不能正常播放。
一定要注意 DVDD 如果接的 3.3V，那么 I2S 总线和 MCLK 必须接 3.3V



因为 IMX6 芯片 clk 输出电压默认是 3.3V，所以 mclk 我没有接上拉电阻，记住 MCLK 一定要和 I2S 总线电压一样，我就是以为 I2S 总线接 3.3V 就可以了，MCLK 不用接 3.3V，导致音乐播放和录音不稳定，后来将 MCLK 改成 3.3V 就 OK 了，所以一定要谨记。
上面原理图是我正确的接法。

echo "1a 061" > /proc/wm8960/reg

这种 echo 是直接把 061 的值写在 1a 地址上，1a 是 wm8960 寄存器地址，061 是值

```
root@imx6qdlsolo:~# echo "1a 024" >/proc/wm8960/reg
root@imx6qdlsolo:~#
```

cat /proc/wm8960/reg 用于跟踪寄存器的状态，按 Ctrl+c 结束

我们看看刚才写入 1a 寄存器的值对不对

```
root@imx6qdlsolo:~# cat /proc/wm8960/reg
r[00] = 000001bf
r[01] = 000001bf
r[02] = 00000165
r[03] = 00000165
r[04] = 00000004
r[19] = 00000172
r[1a] = 00000024
```

一定要手速快赶紧退出否则就刷屏了

这样就很好调试 WM8960 寄存器了，这是怎么做到的？

```
static int imx_wm8960_late_probe(struct snd_soc_card *card)
{
        struct snd_soc_dai *codec_dai = card->rtd[0].codec_dai;
        struct imx_wm8960_data *data = snd_soc_card_get_drvdata(card);
        int ret;

        ret = snd_soc_dai_set_sysclk(codec_dai, WM8960_SYSCLK_MCLK,data->clk_frequency, SND_SOC_CLOCK_IN);
        if (ret < 0)
        printk("WM8960_SYSCLK_MCLK,data->clk_frequency failed !\n");
        wm8960_init(codec_dai);

#ifdef CONFIG_PROC_FS
        wm8960_proc_dir_entry = proc_mkdir("wm8960", NULL);
        if (wm8960_proc_dir_entry) {
                regs_file = proc_create("reg", 0664, wm8960_proc_dir_entry, &regs_proc_fops);
                if (!regs_file) {
                        printk("create reg file error\n");
                }

        }
#endif

        return 0;

}
```

这就是/proc/wm8960 产生的代码

在 IMX6-wm8960.c 初始化程序里加上这段代码，proc_mkdir 是创建/proc/wm8960 目录的

如果没有 proc_mkdir 这一段代码你向 wm8960 写寄存器就会出现问题。

```
root@imx6qdlsolo:~# echo "1a 024">/proc/wm8960/reg
-sh: /proc/wm8960/reg: No such file or directory
root@imx6qdlsolo:~# cat /proc/wm8960/reg
cat: /proc/wm8960/reg: No such file or directory
```

文件没有创建，所以 proc_create 函数是关键

```c
static int write_reg(struct file *file, const char __user *buf,
                             size_t count, loff_t *data)
{
        char regAH, regAL, regVH, regVM, regVL;
        regAH = buf[0];
        regAL = buf[1];

        regVH = buf[3];
        regVM = buf[4];
        regVL = buf[5];

        snd_soc_write(codec_tmp, conv2hex(regAH)*16+conv2hex(regAL), conv2hex(regVH)*16*16+conv2hex(regVM)*16+conv2hex(regVL));
        return count;
}

static const struct file_operations regs_proc_fops = {
 .read  = read_reg,
 .write = write_reg,
};
```

后面这个是实现
&regs_proc_fops 这个回调函
数结构体的字符设备驱动

```c
static struct proc_dir_entry *wm8960_proc_dir_entry=NULL, *regs_file;
static int read_reg(struct file *filep, char __user *buf,
                                size_t count, loff_t *ppos)
{
        char page[1000];
        int len=0;
        int index = 0;
        for (index=0; index<0x38; index++) {
                int ret=0;
                ret = snd_soc_read(codec_tmp, index);
                len += sprintf(page+len, "r[%02x] = %08x\n", index, ret);
        }

        copy_to_user(buf, page, len);
        return len;
}

int conv2hex(char c)
{
        if (c>='0' && c<='9')
                return c-'0';

        if (c>='a' && c<='f')
                return 10+(c-'a');

        if (c>='A' && c<='F')
                return 10+(c-'A');

        printk("invalid data\n");
        return 0;
}
```