Linux ALSA

目录

Linux ALSA 声卡驱动之一：ALSA 架构简介

1. 概述

ALSA 是 Advanced Linux Sound Architecture 的缩写，目前已经成为了 linux 的主流音频体系结构，想了解更多的关于 ALSA 的这一开源项目的信息和知识，请查看以下网址：http://www.alsa-project.org/。

在内核设备驱动层，ALSA 提供了 alsa-driver，同时在应用层，ALSA 为我们提供了 alsa-lib，应用程序只要调用 alsa-lib 提供的 API，即可以完成对底层音频硬件的控制。

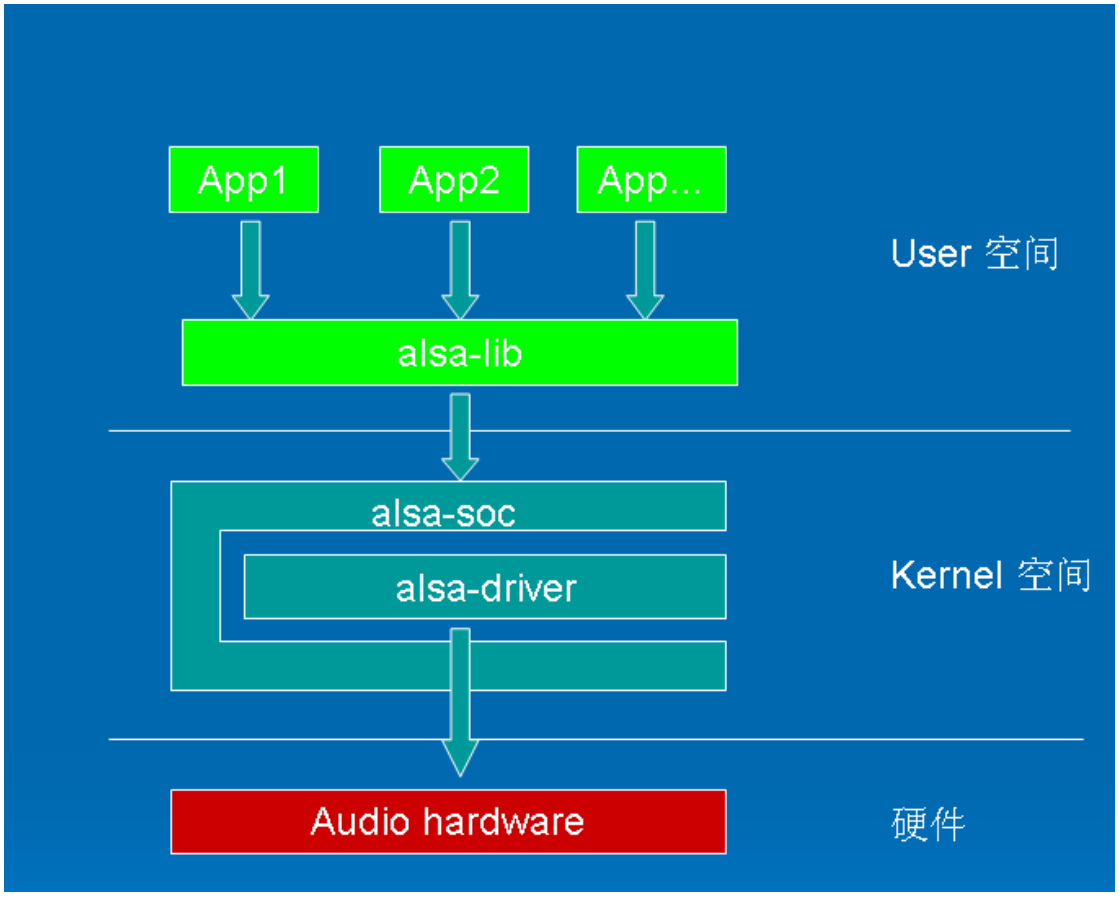


图 1.1 alsa 的软件体系结构

由图 1.1 可以看出，用户空间的 alsa-lib 对应用程序提供统一的 API 接口，这样可以隐藏了驱动层的实现细节，简化了应用程序的实现难度。内核空间中，alsa-soc 其实是对 alsa-driver 的进一步封装，他针对嵌入式设备提供了一些列增强的功能。本系列博文仅对嵌入式系统中的 alsa-driver 和 alsa-soc 进行讨论。

2. ALSA 设备文件结构

我们从 alsa 在 linux 中的设备文件结构开始我们的 alsa 之旅. 看看我的电脑中的 alsa 驱动的设备文件结构:

$ cd /dev/snd
$ ls -l

crw-rw----+ 1 root audio 116, 8 2011-02-23 21:38 controlC0
crw-rw----+ 1 root audio 116, 4 2011-02-23 21:38 midiC0D0
crw-rw----+ 1 root audio 116, 7 2011-02-23 21:39 pcmC0D0c
crw-rw----+ 1 root audio 116, 6 2011-02-23 21:56 pcmC0D0p
crw-rw----+ 1 root audio 116, 5 2011-02-23 21:38 pcmC0D1p
crw-rw----+ 1 root audio 116, 3 2011-02-23 21:38 seq
crw-rw----+ 1 root audio 116, 2 2011-02-23 21:38 timer
$

我们可以看到以下设备文件:

- controlC0 --> 用于声卡的控制，例如通道选择，混音，麦克风的控制等
- midiC0D0 --> 用于播放 midi 音频
- pcmC0D0c --〉 用于录音的 pcm 设备
- pcmC0D0p --〉 用于播放的 pcm 设备
- seq --〉 音序器
- timer --〉 定时器

其中，C0D0 代表的是声卡 0 中的设备 0，pcmC0D0c 最后一个 c 代表 capture，pcmC0D0p 最后一个 p 代表 playback，这些都是 alsa-driver 中的命名规则。从上面的列表可以看出，我的声卡下挂了 6 个设备，根据声卡的实际能力，驱动实际上可以挂上更多种类的设备，在 include/sound/core.h 中，定义了以下设备类型：

[c-sharp] view plaincopy

```
1.  #define SNDRV_DEV_TOPLEVEL  ((__force snd_device_type_t) 0)
2.  #define SNDRV_DEV_CONTROL   ((__force snd_device_type_t) 1)
3.  #define SNDRV_DEV_LOWLEVEL_PRE  ((__force snd_device_type_t) 2)
4.  #define SNDRV_DEV_LOWLEVEL_NORMAL ((__force snd_device_type_t) 0x1000)
```

```
#define SNDRV_DEV_PCM       ((__force snd_device_type_t) 0x1001)
#define SNDRV_DEV_RAWMIDI   ((__force snd_device_type_t) 0x1002)
#define SNDRV_DEV_TIMER     ((__force snd_device_type_t) 0x1003)
#define SNDRV_DEV_SEQUENCER ((__force snd_device_type_t) 0x1004)
#define SNDRV_DEV_HWDEP     ((__force snd_device_type_t) 0x1005)
#define SNDRV_DEV_INFO      ((__force snd_device_type_t) 0x1006)
#define SNDRV_DEV_BUS       ((__force snd_device_type_t) 0x1007)
#define SNDRV_DEV_CODEC     ((__force snd_device_type_t) 0x1008)
#define SNDRV_DEV_JACK          ((__force snd_device_type_t) 0x1009)
#define SNDRV_DEV_LOWLEVEL  ((__force snd_device_type_t) 0x2000)
```

通常，我们更关心的是 pcm 和 control 这两种设备。

3. 驱动的代码文件结构

在 Linux2.6 代码树中，Alsa 的代码文件结构如下：

sound
/core
/oss
/seq
/ioctl32
/include
/drivers
/i2c
/synth
/emux
/pci
/(cards)
/isa
/(cards)
/arm
/ppc
/sparc
/usb
/pcmcia /(cards)

/oss

/soc

/codecs

- core 该目录包含了 ALSA 驱动的中间层，它是整个 ALSA 驱动的核心部分
- core/oss 包含模拟旧的 OSS 架构的 PCM 和 Mixer 模块
- core/seq 有关音序器相关的代码
- include ALSA 驱动的公共头文件目录，该目录的头文件需要导出给用户空间的应用程序使用，通常，驱动模块私有的头文件不应放置在这里
- drivers 放置一些与 CPU、BUS 架构无关的公用代码
- i2c ALSA 自己的 I2C 控制代码
- pci pci 声卡的顶层目录，子目录包含各种 pci 声卡的代码
- isa isa 声卡的顶层目录，子目录包含各种 isa 声卡的代码
- soc 针对 system-on-chip 体系的中间层代码
- soc/codecs 针对 soc 体系的各种 codec 的代码，与平台无关

Linux ALSA 声卡驱动之二：声卡的创建

1. struct snd_card

1.1. snd_card 是什么

snd_card 可以说是整个 ALSA 音频驱动最顶层的一个结构，整个声卡的软件逻辑结构开始于该结构，几乎所有与声音相关的逻辑设备都是在 snd_card 的管理之下，声卡驱动的第一个动作通常就是创建一个 snd_card 结构体。正因为如此，本节中，我们也从 struct cnd_card 开始吧。

1.2. snd_card 的定义

snd_card 的定义位于改头文件中：include/sound/core.h

[c-sharp] view plaincopy

```
/* main structure for soundcard */

struct snd_card {
    int number;         /* number of soundcard (index to
                                    snd_cards) */

    char id[16];            /* id string of this card */
    char driver[16];        /* driver name */
    char shortname[32];     /* short name of this soundcard */
    char longname[80];      /* name of this soundcard */
    char mixername[80];     /* mixer name */
    char components[128];       /* card components delimited with
                                    space */
    struct module *module;      /* top-level module */

    void *private_data;     /* private data for soundcard */
    void (*private_free) (struct snd_card *card); /* callback for freein
g of
                                    private data */
    struct list_head devices;   /* devices */

    unsigned int last_numid;    /* last used numeric ID */
    struct rw_semaphore controls_rwsem; /* controls list lock */
    rwlock_t ctl_files_rwlock;  /* ctl_files list lock */
    int controls_count;     /* count of all controls */
    int user_ctl_count;     /* count of all user controls */
    struct list_head controls;  /* all controls for this card */
    struct list_head ctl_files; /* active control files */

    struct snd_info_entry *proc_root;   /* root for soundcard specific
files */
    struct snd_info_entry *proc_id; /* the card id */
    struct proc_dir_entry *proc_root_link;  /* number link to real id *
/

    struct list_head files_list;    /* all files associated to this ca
rd */
    struct snd_shutdown_f_ops *s_f_ops; /* file operations in the shutd
own
                                    state */
    spinlock_t files_lock;      /* lock the files for this card */
    int shutdown;           /* this card is going down */
    int free_on_last_close;     /* free in context of file_release */
```

```
39.     wait_queue_head_t shutdown_sleep;
40.     struct device *dev;     /* device assigned to this card */
41. #ifndef CONFIG_SYSFS_DEPRECATED
42.     struct device *card_dev;    /* cardX object for sysfs */
43. #endif
44.
45. #ifdef CONFIG_PM
46.     unsigned int power_state;   /* power state */
47.     struct mutex power_lock;    /* power lock */
48.     wait_queue_head_t power_sleep;
49. #endif
50.
51. #if defined(CONFIG_SND_MIXER_OSS) || defined(CONFIG_SND_MIXER_OSS_MODULE)
52.     struct snd_mixer_oss *mixer_oss;
53.     int mixer_oss_change_count;
54. #endif
55. };
```

- struct list_head devices　　记录该声卡下所有逻辑设备的链表
- struct list_head controls　　记录该声卡下所有的控制单元的链表
- void *private_data　　声卡的私有数据，可以在创建声卡时通过参数指定数据的大小

2. 声卡的建立流程

2.1.1. 第一步，创建 snd_card 的一个实例

[c-sharp] view plaincopy

```
1.  struct snd_card *card;
2.  int err;
3.  ....
4.  err = snd_card_create(index, id, THIS_MODULE, 0, &card);
```

- index　　一个整数值，该声卡的编号
- id　　字符串，声卡的标识符

- 第四个参数　该参数决定在创建 snd_card 实例时，需要同时额外分配的私有数据的大小，该数据的指针最终会赋值给 snd_card 的 private_data 数据成员
- card　　返回所创建的 snd_card 实例的指针

2.1.2. 第二步，创建声卡的芯片专用数据

声卡的专用数据主要用于存放该声卡的一些资源信息，例如中断资源、io 资源、dma 资源等。可以有两种创建方法：

1

- 通过上一步中 snd_card_create()中的第四个参数，让 snd_card_create 自己创建

[c-sharp] view plaincopy

```
// struct mychip 用于保存专用数据
err = snd_card_create(index, id, THIS_MODULE,
                sizeof(struct mychip), &card);
// 从 private_data 中取出
struct mychip *chip = card->private_data;
```

2

- 自己创建：

[c-sharp] view plaincopy

```
struct mychip {
    struct snd_card *card;
    ....
};
struct snd_card *card;
struct mychip *chip;

chip = kzalloc(sizeof(*chip), GFP_KERNEL);
......
err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
// 专用数据记录 snd_card 实例
chip->card = card;
.....
```

然后，把芯片的专有数据注册为声卡的一个低阶设备：

[c-sharp] view plaincopy

```c
static int snd_mychip_dev_free(struct snd_device *device)
{
    return snd_mychip_free(device->device_data);
}

static struct snd_device_ops ops = {
    .dev_free = snd_mychip_dev_free,
};
....
snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
```

注册为低阶设备主要是为了当声卡被注销时，芯片专用数据所占用的内存可以被自动地释放。

2.1.3. 第三步，设置 Driver 的 ID 和名字

[c-sharp] view plaincopy

```c
strcpy(card->driver, "My Chip");
strcpy(card->shortname, "My Own Chip 123");
sprintf(card->longname, "%s at 0x%lx irq %i",
            card->shortname, chip->ioport, chip->irq);
```

snd_card 的 driver 字段保存着芯片的 ID 字符串，user 空间的 alsa-lib 会使用到该字符串，所以必须要保证该 ID 的唯一性。shortname 字段更多地用于打印信息，longname 字段则会出现在/proc/asound/cards 中。

2.1.4. 第四步，创建声卡的功能部件（逻辑设备），例如 PCM，Mixer，MIDI 等

这时候可以创建声卡的各种功能部件了，还记得开头的 snd_card 结构体的 devices 字段吗？每一种部件的创建最终会调用 snd_device_new()来生成一个 snd_device 实例，并把该实例链接到 snd_card 的 devices 链表中。

通常，alsa-driver 的已经提供了一些常用的部件的创建函数，而不必直接调用 snd_device_new()，比如：

PCM ---- snd_pcm_new()

RAWMIDI -- snd_rawmidi_new()

CONTROL -- snd_ctl_create()

TIMER -- snd_timer_new()

INFO -- snd_card_proc_new()

JACK -- snd_jack_new()

2.1.5. 第五步，注册声卡

[c-sharp] view plaincopy

```c
err = snd_card_register(card);
if (err < 0) {
    snd_card_free(card);
    return err;
}
```

2.2. 一个实际的例子

我把/sound/arm/pxa2xx-ac97.c 的部分代码贴上来：

[cpp] view plaincopy

```cpp
static int __devinit pxa2xx_ac97_probe(struct platform_device *dev)
{
    struct snd_card *card;
    struct snd_ac97_bus *ac97_bus;
    struct snd_ac97_template ac97_template;
    int ret;
    pxa2xx_audio_ops_t *pdata = dev->dev.platform_data;
```

```
8.
9.      if (dev->id >= 0) {
10.         dev_err(&dev->dev, "PXA2xx has only one AC97 port./n");
11.         ret = -ENXIO;
12.         goto err_dev;
13.     }
14. ////(1)////
15.     ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
16.                           THIS_MODULE, 0, &card);
17.     if (ret < 0)
18.         goto err;
19.
20.     card->dev = &dev->dev;
21. ////(3)////
22.     strncpy(card->driver, dev->dev.driver->name, sizeof(card->driver));
23.
24. ////(4)////
25.     ret = pxa2xx_pcm_new(card, &pxa2xx_ac97_pcm_client, &pxa2xx_ac97_pcm);
26.     if (ret)
27.         goto err;
28. ////(2)////
29.     ret = pxa2xx_ac97_hw_probe(dev);
30.     if (ret)
31.         goto err;
32.
33. ////(4)////
34.     ret = snd_ac97_bus(card, 0, &pxa2xx_ac97_ops, NULL, &ac97_bus);
35.     if (ret)
36.         goto err_remove;
37.     memset(&ac97_template, 0, sizeof(ac97_template));
38.     ret = snd_ac97_mixer(ac97_bus, &ac97_template, &pxa2xx_ac97_ac97);
39.     if (ret)
40.         goto err_remove;
41. ////(3)////
42.     snprintf(card->shortname, sizeof(card->shortname),
43.          "%s", snd_ac97_get_short_name(pxa2xx_ac97_ac97));
44.     snprintf(card->longname, sizeof(card->longname),
45.          "%s (%s)", dev->dev.driver->name, card->mixername);
46.
47.     if (pdata && pdata->codec_pdata[0])
48.         snd_ac97_dev_add_pdata(ac97_bus->codec[0], pdata->codec_pdata[0]);
49.     snd_card_set_dev(card, &dev->dev);
50. ////(5)////
51.     ret = snd_card_register(card);
```

```
52.     if (ret == 0) {
53.         platform_set_drvdata(dev, card);
54.         return 0;
55.     }
56.
57. err_remove:
58.     pxa2xx_ac97_hw_remove(dev);
59. err:
60.     if (card)
61.         snd_card_free(card);
62. err_dev:
63.     return ret;
64. }
65.
66. static int __devexit pxa2xx_ac97_remove(struct platform_device *dev)
67. {
68.     struct snd_card *card = platform_get_drvdata(dev);
69.
70.     if (card) {
71.         snd_card_free(card);
72.         platform_set_drvdata(dev, NULL);
73.         pxa2xx_ac97_hw_remove(dev);
74.     }
75.
76.     return 0;
77. }
78.
79. static struct platform_driver pxa2xx_ac97_driver = {
80.     .probe      = pxa2xx_ac97_probe,
81.     .remove     = __devexit_p(pxa2xx_ac97_remove),
82.     .driver     = {
83.         .name   = "pxa2xx-ac97",
84.         .owner  = THIS_MODULE,
85. #ifdef CONFIG_PM
86.         .pm = &pxa2xx_ac97_pm_ops,
87. #endif
88.     },
89. };
90.
91. static int __init pxa2xx_ac97_init(void)
92. {
93.     return platform_driver_register(&pxa2xx_ac97_driver);
94. }
95.
```

```
96. static void __exit pxa2xx_ac97_exit(void)
97. {
98.     platform_driver_unregister(&pxa2xx_ac97_driver);
99. }
100.
101. module_init(pxa2xx_ac97_init);
102. module_exit(pxa2xx_ac97_exit);
103.
104. MODULE_AUTHOR("Nicolas Pitre");
105. MODULE_DESCRIPTION("AC97 driver for the Intel PXA2xx chip");
```

驱动程序通常由 probe 回调函数开始，对一下 2.1 中的步骤，是否有相似之处？

经过以上的创建步骤之后，声卡的逻辑结构如下图所示：

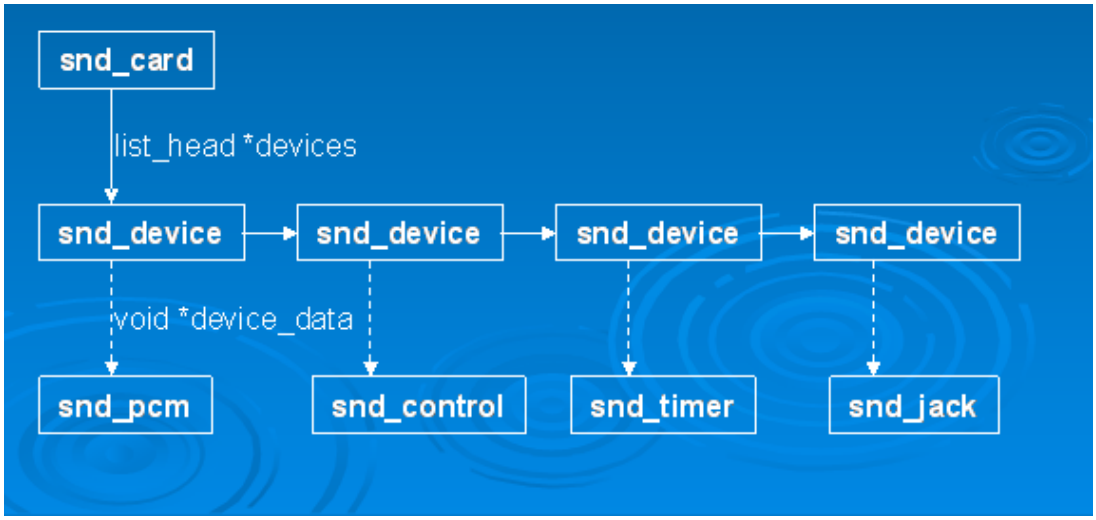


图 2.2.1 声卡的软件逻辑结构

下面的章节里我们分别讨论一下 snd_card_create()和 snd_card_register()这两个函数。

3. snd_card_create()

snd_card_create()在/sound/core/init.c 中定义。

[cpp] view plaincopy

```cpp
/**
 *  snd_card_create - create and initialize a soundcard structure
 *  @idx: card index (address) [0 ... (SNDRV_CARDS-1)]
 *  @xid: card identification (ASCII string)
 *  @module: top level module for locking
 *  @extra_size: allocate this extra size after the main soundcard structure
 *  @card_ret: the pointer to store the created card instance
 *
 *  Creates and initializes a soundcard structure.
 *
 *  The function allocates snd_card instance via kzalloc with the given
 *  space for the driver to use freely.  The allocated struct is stored
 *  in the given card_ret pointer.
 *
 *  Returns zero if successful or a negative error code.
 */
int snd_card_create(int idx, const char *xid,
            struct module *module, int extra_size,
            struct snd_card **card_ret)
```

首先，根据 extra_size 参数的大小分配内存，该内存区可以作为芯片的专有数据使用（见前面的介绍）：

[c-sharp] view plaincopy

```c-sharp
card = kzalloc(sizeof(*card) + extra_size, GFP_KERNEL);
if (!card)
    return -ENOMEM;
```

拷贝声卡的 ID 字符串：

[c-sharp] view plaincopy

```
1.  if (xid)
2.      strlcpy(card->id, xid, sizeof(card->id));
```

如果传入的声卡编号为-1，自动分配一个索引编号：

[c-sharp] view plaincopy

```
1.  if (idx < 0) {
2.      for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
3.          /* idx == -1 == 0xffff means: take any free slot */
4.          if (~snd_cards_lock & idx & 1<<idx2) {
5.              if (module_slot_match(module, idx2)) {
6.                  idx = idx2;
7.                  break;
8.              }
9.          }
10. }
11. if (idx < 0) {
12.     for (idx2 = 0; idx2 < SNDRV_CARDS; idx2++)
13.         /* idx == -1 == 0xffff means: take any free slot */
14.         if (~snd_cards_lock & idx & 1<<idx2) {
15.             if (!slots[idx2] || !*slots[idx2]) {
16.                 idx = idx2;
17.                 break;
18.             }
19.         }
20. }
```

初始化 snd_card 结构中必要的字段：

[c-sharp] view plaincopy

```
1.      card->number = idx;
2.      card->module = module;
3.      INIT_LIST_HEAD(&card->devices);
4.      init_rwsem(&card->controls_rwsem);
5.      rwlock_init(&card->ctl_files_rwlock);
6.      INIT_LIST_HEAD(&card->controls);
7.      INIT_LIST_HEAD(&card->ctl_files);
8.      spin_lock_init(&card->files_lock);
9.      INIT_LIST_HEAD(&card->files_list);
10.     init_waitqueue_head(&card->shutdown_sleep);
11. #ifdef CONFIG_PM
12.     mutex_init(&card->power_lock);
13.     init_waitqueue_head(&card->power_sleep);
```

```
14. #endif
```

建立逻辑设备：Control

[c-sharp] view plaincopy

```
1.  /* the control interface cannot be accessed from the user space u
    ntil */
2.  /* snd_cards_bitmask and snd_cards are set with snd_card_register
     */
3.  err = snd_ctl_create(card);
```

建立 proc 文件中的 info 节点：通常就是/proc/asound/card0

[c-sharp] view plaincopy

```
1.  err = snd_info_card_create(card);
```

把第一步分配的内存指针放入 private_data 字段中：

[c-sharp] view plaincopy

```
1.  if (extra_size > 0)
2.      card->private_data = (char *)card + sizeof(struct snd_card);
```

4. snd_card_register()

snd_card_create()在/sound/core/init.c 中定义。

[c-sharp] view plaincopy

```
1.  /**
2.   *  snd_card_register - register the soundcard
3.   *  @card: soundcard structure
4.   *
5.   *  This function registers all the devices assigned to the sound
    card.
6.   *  Until calling this, the ALSA control interface is blocked fro
    m the
7.   *  external accesses.  Thus, you should call this function at th
    e end
8.   *  of the initialization of the card.
9.   *
```

```
 *  Returns zero otherwise a negative error code if the registrai
n failed.
 */
int snd_card_register(struct snd_card *card)
```

首先，创建 sysfs 下的设备：

[c-sharp] view plaincopy

```
if (!card->card_dev) {
    card->card_dev = device_create(sound_class, card->dev,
                       MKDEV(0, 0), card,
                       "card%i", card->number);
    if (IS_ERR(card->card_dev))
        card->card_dev = NULL;
}
```

其中，sound_class 是在/sound/sound_core.c 中创建的：

[c-sharp] view plaincopy

```
static char *sound_devnode(struct device *dev, mode_t *mode)
{
    if (MAJOR(dev->devt) == SOUND_MAJOR)
        return NULL;
    return kasprintf(GFP_KERNEL, "snd/%s", dev_name(dev));
}
static int __init init_soundcore(void)
{
    int rc;

    rc = init_oss_soundcore();
    if (rc)
        return rc;

    sound_class = class_create(THIS_MODULE, "sound");
    if (IS_ERR(sound_class)) {
        cleanup_oss_soundcore();
        return PTR_ERR(sound_class);
    }

    sound_class->devnode = sound_devnode;

    return 0;
}
```

由此可见，声卡的 class 将会出现在文件系统的/sys/class/sound/下面，并且，sound_devnode()也决定了相应的设备节点也将会出现在/dev/snd/下面。

接下来的步骤，通过 snd_device_register_all()注册所有挂在该声卡下的逻辑设备，snd_device_register_all()实际上是通过 snd_card 的 devices 链表，遍历所有的 snd_device，并且调用 snd_device 的 ops->dev_register()来实现各自设备的注册的。

[c-sharp] view plaincopy

```
1.  if ((err = snd_device_register_all(card)) < 0)
2.      return err;
```

最后就是建立一些相应的 proc 和 sysfs 下的文件或属性节点，代码就不贴了。

至此，整个声卡完成了建立过程。

Linux ALSA 声卡驱动之三：PCM 设备的创建

1. PCM 是什么

PCM 是英文 Pulse-code modulation 的缩写，中文译名是脉冲编码调制。我们知道在现实生活中，人耳听到的声音是模拟信号，PCM 就是要把声音从模拟转换成数字信号的一种技术，他的原理简单地说就是利用一个固定的频率对模拟信号进行采样，采样后的信号在波形上看就像一串连续的幅值不一的脉冲，把这些脉冲的幅值按一定的精度进行量化，这些量化后的数值被连续地输出、传输、处理或记录到存储介质中，所有这些组成了数字音频的产生过程。

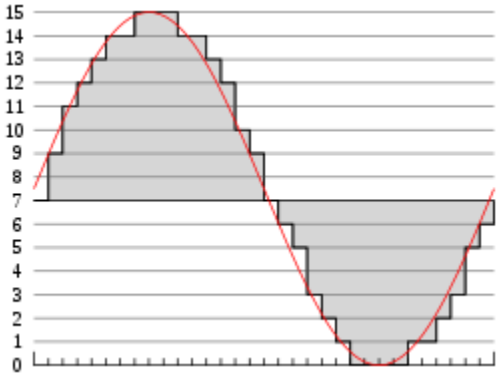


图 1.1　模拟音频的采样、量化

PCM 信号的两个重要指标是采样频率和量化精度，目前，CD 音频的采样频率通常为 44100Hz，量化精度是 16bit。通常，播放音乐时，应用程序从存储介质中读取音频数据（MP3、WMA、AAC.....），经过解码后，最终送到音频驱动程序中的就是 PCM 数据，反过来，在录音时，音频驱动不停地把采样所得的 PCM 数据送回给应用程序，由应用程序完成压缩、存储等任务。所以，音频驱动的两大核心任务就是：

- playback　如何把用户空间的应用程序发过来的 PCM 数据，转化为人耳可以辨别的模拟音频
- capture　把 mic 拾取到得模拟信号，经过采样、量化，转换为 PCM 信号送回给用户空间的应用程序

2. alsa-driver 中的 PCM 中间层

ALSA 已经为我们实现了功能强劲的 PCM 中间层，自己的驱动中只要实现一些底层的需要访问硬件的函数即可。

要访问 PCM 的中间层代码，你首先要包含头文件<sound/pcm.h>，另外，如果需要访问一些与 hw_param 相关的函数，可能也要包含<sound/pcm_params.h>。

每个声卡最多可以包含 4 个 pcm 的实例，每个 pcm 实例对应一个 pcm 设备文件。pcm 实例数量的这种限制源于 linux 设备号所占用的位大小，如果以后使用 64 位的设备号，我们将可以创建更多的 pcm 实例。不过大多数情况下，在嵌入式设备中，一个 pcm 实例已经足够了。

一个 pcm 实例由一个 playback stream 和一个 capture stream 组成，这两个 stream 又分别有一个或多个 substreams 组成。

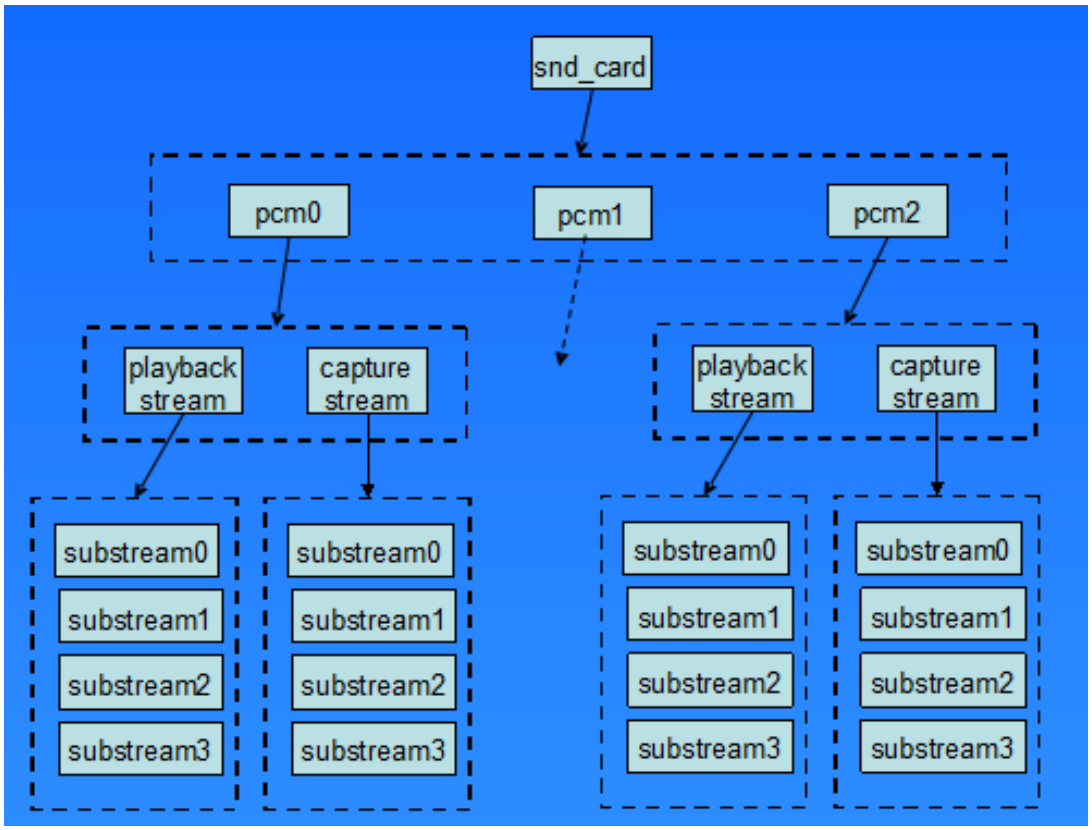


图 2.1　声卡中的 pcm 结构

在嵌入式系统中，通常不会像图 2.1 中这么复杂，大多数情况下是一个声卡，一个 pcm 实例，pcm 下面有一个 playback 和 capture stream，playback 和 capture 下面各自有一个 substream。

下面一张图列出了 pcm 中间层几个重要的结构，他可以让我们从 uml 的角度看一看这列结构的关系，理清他们之间的关系，对我们理解 pcm 中间层的实现方式。

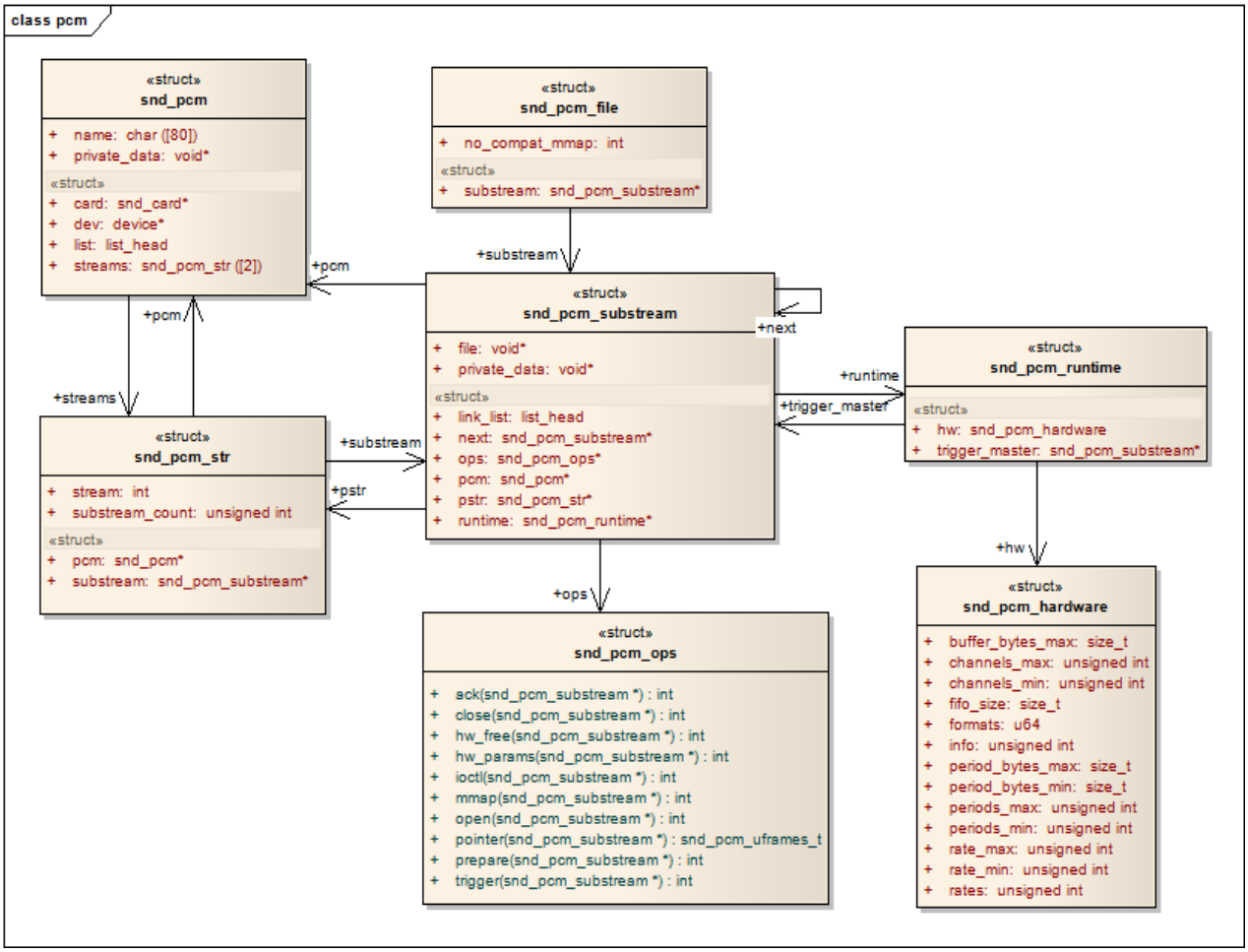


图 2.2　pcm 中间层的几个重要的结构体的关系图

- snd_pcm 是挂在 snd_card 下面的一个 snd_device
- snd_pcm 中的字段：streams[2]，该数组中的两个元素指向两个 snd_pcm_str 结构，分别代表 playback stream 和 capture stream
- snd_pcm_str 中的 substream 字段，指向 snd_pcm_substream 结构
- snd_pcm_substream 是 pcm 中间层的核心，绝大部分任务都是在 substream 中处理，尤其是他的 ops（snd_pcm_ops）字段，许多 user 空间的应用程序通过 alsa-lib 对驱动程序的请求都是由该结构中的函数处理。它的 runtime 字段则指向 snd_pcm_runtime 结构，snd_pcm_runtime 记录这 substream 的一些重要的软件和硬件运行环境和参数。

3. 新建一个 pcm

alsa-driver 的中间层已经为我们提供了新建 pcm 的 api：

```
int snd_pcm_new(struct snd_card *card, const char *id, int device, int
playback_count, int capture_count,
                struct snd_pcm ** rpcm);
```

参数 device 表示目前创建的是该声卡下的第几个 pcm，第一个 pcm 设备从 0 开始。

参数 playback_count 表示该 pcm 将会有几个 playback substream。

参数 capture_count 表示该 pcm 将会有几个 capture substream。

另一个用于设置 pcm 操作函数接口的 api：

```
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction, struct snd_pcm_ops
*ops);
```

新建一个 pcm 可以用下面一张新建 pcm 的调用的序列图进行描述：

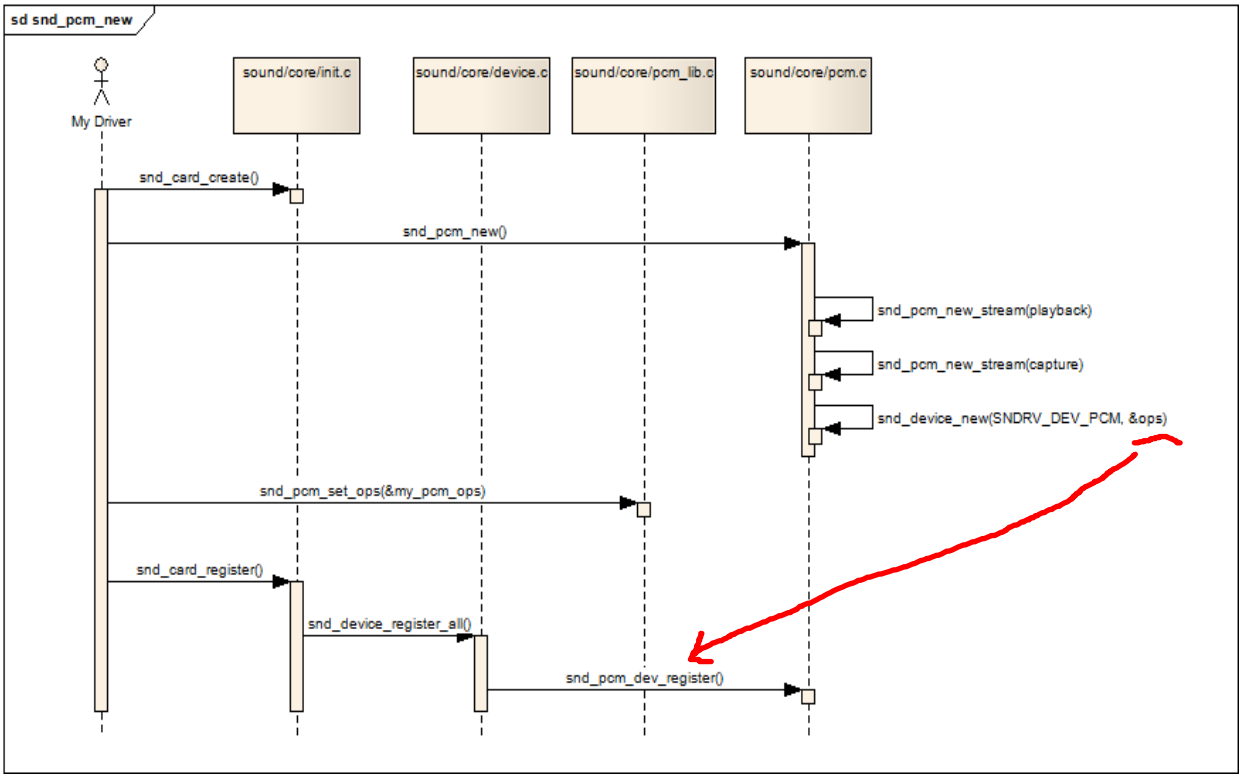


图 3.1 新建 pcm 的序列图

- **snd_card_create** pcm 是声卡下的一个设备（部件），所以第一步是要创建一个声卡
- **snd_pcm_new** 调用该 api 创建一个 pcm，在该 api 中会做以下事情
  - 如果有，建立 playback stream，相应的 substream 也同时建立
  - 如果有，建立 capture stream，相应的 substream 也同时建立
  - 调用 snd_device_new()把该 pcm 挂到声卡中，参数 ops 中的 dev_register 字段指向了函数 snd_pcm_dev_register，这个回调函数会在声卡的注册阶段被调用。
- **snd_pcm_set_ops** 设置操作该 pcm 的控制/操作接口函数，参数中的 snd_pcm_ops 结构中的函数通常就是我们驱动要实现的函数
- **snd_card_register** 注册声卡，在这个阶段会遍历声卡下的所有逻辑设备，并且调用各设备的注册回调函数，对于 pcm，就是第二步提到的 snd_pcm_dev_register 函数，该回调函数建立了和用户空间应用程序（alsa-lib）通信所用的设备文件节点:/dev/snd/pcmCxxDxxp 和/dev/snd/pcmCxxDxxc

4. 设备文件节点的建立（dev/snd/pcmCxxDxxp、pcmCxxDxxc）

4.1 struct snd_minor

每个 snd_minor 结构体保存了声卡下某个逻辑设备的上下文信息，他在逻辑设备建立阶段被填充，在逻辑设备被使用时就可以从该结构体中得到相应的信息。pcm 设备也不例外，也需要使用该结构体。该结构体在 include/sound/core.h 中定义。

[c-sharp] view plaincopy

```
struct snd_minor {
    int type;           /* SNDRV_DEVICE_TYPE_XXX */
    int card;           /* card number */
    int device;         /* device number */
    const struct file_operations *f_ops;    /* file operations */
    void *private_data;     /* private data for f_ops->open */
    struct device *dev;     /* device for sysfs */
};
```

在 sound/sound.c 中定义了一个 snd_minor 指针的全局数组：

[c-sharp] view plaincopy

```
static struct snd_minor *snd_minors[256];
```

前面说过，在声卡的注册阶段（snd_card_register），会调用 pcm 的回调函数 snd_pcm_dev_register()，这个函数里会调用函数 snd_register_device_for_dev()：

[c-sharp] view plaincopy

```
static int snd_pcm_dev_register(struct snd_device *device)
{
    ......

    /* register pcm */
    err = snd_register_device_for_dev(devtype, pcm->card,
                          pcm->device,
                      &snd_pcm_f_ops[cidx],
                      pcm, str, dev);
    ......
```

```
11. }
```

我们再进入 snd_register_device_for_dev()：

[c-sharp] view plaincopy

```
1.  int snd_register_device_for_dev(int type, struct snd_card *card, int dev,
2.                  const struct file_operations *f_ops,
3.                  void *private_data,
4.                  const char *name, struct device *device)
5.  {
6.      int minor;
7.      struct snd_minor *preg;
8.
9.      if (snd_BUG_ON(!name))
10.         return -EINVAL;
11.     preg = kmalloc(sizeof *preg, GFP_KERNEL);
12.     if (preg == NULL)
13.         return -ENOMEM;
14.     preg->type = type;
15.     preg->card = card ? card->number : -1;
16.     preg->device = dev;
17.     preg->f_ops = f_ops;
18.     preg->private_data = private_data;
19.     mutex_lock(&sound_mutex);
20. #ifdef CONFIG_SND_DYNAMIC_MINORS
21.     minor = snd_find_free_minor();
22. #else
23.     minor = snd_kernel_minor(type, card, dev);
24.     if (minor >= 0 && snd_minors[minor])
25.         minor = -EBUSY;
26. #endif
27.     if (minor < 0) {
28.         mutex_unlock(&sound_mutex);
29.         kfree(preg);
30.         return minor;
31.     }
32.     snd_minors[minor] = preg;
33.     preg->dev = device_create(sound_class, device, MKDEV(major, minor),
34.                   private_data, "%s", name);
35.     if (IS_ERR(preg->dev)) {
36.         snd_minors[minor] = NULL;
37.         mutex_unlock(&sound_mutex);
38.         minor = PTR_ERR(preg->dev);
```

```
39.         kfree(preg);
40.         return minor;
41.     }
42.
43.     mutex_unlock(&sound_mutex);
44.     return 0;
45. }
```

- 首先，分配并初始化一个 snd_minor 结构中的各字段
  - type：SNDRV_DEVICE_TYPE_PCM_PLAYBACK/SNDRV_DEVICE_TYPE_PCM_CAPTURE
  - card: card 的编号
  - device：pcm 实例的编号，大多数情况为 0
  - f_ops：snd_pcm_f_ops
  - private_data：指向该 pcm 的实例
- 根据 type，card 和 pcm 的编号，确定数组的索引值 minor，minor 也作为 pcm 设备的此设备号
- 把该 snd_minor 结构的地址放入全局数组 snd_minors[minor]中
- 最后，调用 device_create 创建设备节点

4.2 设备文件的建立

在 4.1 节的最后，设备文件已经建立，不过 4.1 节的重点在于 snd_minors 数组的赋值过程，在本节中，我们把重点放在设备文件中。

回到 pcm 的回调函数 snd_pcm_dev_register()中：

[c-sharp] view plaincopy

```
1.  static int snd_pcm_dev_register(struct snd_device *device)
2.  {
```

```
3.      int cidx, err;
4.      char str[16];
5.      struct snd_pcm *pcm;
6.      struct device *dev;
7.
8.      pcm = device->device_data;
9.           ......
10.     for (cidx = 0; cidx < 2; cidx++) {
11.                  ......
12.         switch (cidx) {
13.         case SNDRV_PCM_STREAM_PLAYBACK:
14.             sprintf(str, "pcmC%iD%ip", pcm->card->number, pcm->device);
15.             devtype = SNDRV_DEVICE_TYPE_PCM_PLAYBACK;
16.             break;
17.         case SNDRV_PCM_STREAM_CAPTURE:
18.             sprintf(str, "pcmC%iD%ic", pcm->card->number, pcm->device);
19.             devtype = SNDRV_DEVICE_TYPE_PCM_CAPTURE;
20.             break;
21.         }
22.         /* device pointer to use, pcm->dev takes precedence if
23.          * it is assigned, otherwise fall back to card's device
24.          * if possible */
25.         dev = pcm->dev;
26.         if (!dev)
27.             dev = snd_card_get_device_link(pcm->card);
28.         /* register pcm */
29.         err = snd_register_device_for_dev(devtype, pcm->card,
30.                                 pcm->device,
31.                                 &snd_pcm_f_ops[cidx],
32.                                 pcm, str, dev);
33.                  ......
34.     }
35.          ......
36. }
```

以上代码我们可以看出，对于一个 pcm 设备，可以生成两个设备文件，一个用于 playback，一个用于 capture，代码中也确定了他们的命名规则：

- playback -- pcmCxDxp，通常系统中只有一各声卡和一个 pcm，它就是 pcmC0D0p
- capture -- pcmCxDxc，通常系统中只有一各声卡和一个 pcm，它就是 pcmC0D0c

snd_pcm_f_ops

snd_pcm_f_ops 是一个标准的文件系统 file_operations 结构数组，它的定义在 sound/core/pcm_native.c 中：

[c-sharp] view plaincopy

```
const struct file_operations snd_pcm_f_ops[2] = {
    {
        .owner =        THIS_MODULE,
        .write =        snd_pcm_write,
        .aio_write =        snd_pcm_aio_write,
        .open =         snd_pcm_playback_open,
        .release =      snd_pcm_release,
        .llseek =       no_llseek,
        .poll =         snd_pcm_playback_poll,
        .unlocked_ioctl =   snd_pcm_playback_ioctl,
        .compat_ioctl =     snd_pcm_ioctl_compat,
        .mmap =         snd_pcm_mmap,
        .fasync =       snd_pcm_fasync,
        .get_unmapped_area =    snd_pcm_get_unmapped_area,
    },
    {
        .owner =        THIS_MODULE,
        .read =         snd_pcm_read,
        .aio_read =     snd_pcm_aio_read,
        .open =         snd_pcm_capture_open,
        .release =      snd_pcm_release,
        .llseek =       no_llseek,
        .poll =         snd_pcm_capture_poll,
        .unlocked_ioctl =   snd_pcm_capture_ioctl,
        .compat_ioctl =     snd_pcm_ioctl_compat,
        .mmap =         snd_pcm_mmap,
        .fasync =       snd_pcm_fasync,
        .get_unmapped_area =    snd_pcm_get_unmapped_area,
    }
};
```

snd_pcm_f_ops 作为 snd_register_device_for_dev 的参数被传入，并被记录在 snd_minors[minor]中的字段 f_ops 中。最后，在 snd_register_device_for_dev 中创建设备节点：

[c-sharp] view plaincopy

```
1.  snd_minors[minor] = preg;
2.  preg->dev = device_create(sound_class, device, MKDEV(major, minor),
3.                  private_data, "%s", name);
```

4.3 层层深入，从应用程序到驱动层 pcm

4.3.1 字符设备注册

在 sound/core/sound.c 中有 alsa_sound_init()函数，定义如下：

[c-sharp] view plaincopy

```
1.  static int __init alsa_sound_init(void)
2.  {
3.      snd_major = major;
4.      snd_ecards_limit = cards_limit;
5.      if (register_chrdev(major, "alsa", &snd_fops)) {
6.          snd_printk(KERN_ERR "unable to register native major device number %
    d/n", major);
7.          return -EIO;
8.      }
9.      if (snd_info_init() < 0) {
10.         unregister_chrdev(major, "alsa");
11.         return -ENOMEM;
12.     }
13.     snd_info_minor_register();
14.     return 0;
15. }
```

register_chrdev 中的参数 major 与之前创建 pcm 设备是 device_create 时的 major 是同一个，这样的结果是，当应用程序 open 设备文件/dev/snd/pcmCxDxp 时，会进入 snd_fops 的 open 回调函数，我们将在下一节中讲述 open 的过程。

4.3.2 打开 pcm 设备

从上一节中我们得知，open 一个 pcm 设备时，将会调用 snd_fops 的 open 回调函数，我们先看看 snd_fops 的定义：

[c-sharp] view plaincopy

```c
static const struct file_operations snd_fops =
{
    .owner =    THIS_MODULE,
    .open =     snd_open
};
```

跟入 snd_open 函数，它首先从 inode 中取出此设备号，然后以次设备号为索引，从 snd_minors 全局数组中取出当初注册 pcm 设备时填充的 snd_minor 结构（参看 4.1 节的内容），然后从 snd_minor 结构中取出 pcm 设备的 f_ops，并且把 file->f_op 替换为 pcm 设备的 f_ops，紧接着直接调用 pcm 设备的 f_ops->open()，然后返回。因为 file->f_op 已经被替换，以后，应用程序的所有 read/write/ioctl 调用都会进入 pcm 设备自己的回调函数中，也就是 4.2 节中提到的 snd_pcm_f_ops 结构中定义的回调。

[c-sharp] view plaincopy

```c
static int snd_open(struct inode *inode, struct file *file)
{
    unsigned int minor = iminor(inode);
    struct snd_minor *mptr = NULL;
    const struct file_operations *old_fops;
    int err = 0;

    if (minor >= ARRAY_SIZE(snd_minors))
        return -ENODEV;
    mutex_lock(&sound_mutex);
    mptr = snd_minors[minor];
    if (mptr == NULL) {
        mptr = autoload_device(minor);
        if (!mptr) {
            mutex_unlock(&sound_mutex);
            return -ENODEV;
        }
    }
    old_fops = file->f_op;
    file->f_op = fops_get(mptr->f_ops);
    if (file->f_op == NULL) {
        file->f_op = old_fops;
        err = -ENODEV;
    }
    mutex_unlock(&sound_mutex);
    if (err < 0)
```

```
27.         return err;
28. 
29.     if (file->f_op->open) {
30.         err = file->f_op->open(inode, file);
31.         if (err) {
32.             fops_put(file->f_op);
33.             file->f_op = fops_get(old_fops);
34.         }
35.     }
36.     fops_put(old_fops);
37.     return err;
38. }
```

下面的序列图展示了应用程序如何最终调用到 snd_pcm_f_ops 结构中的回调函数：

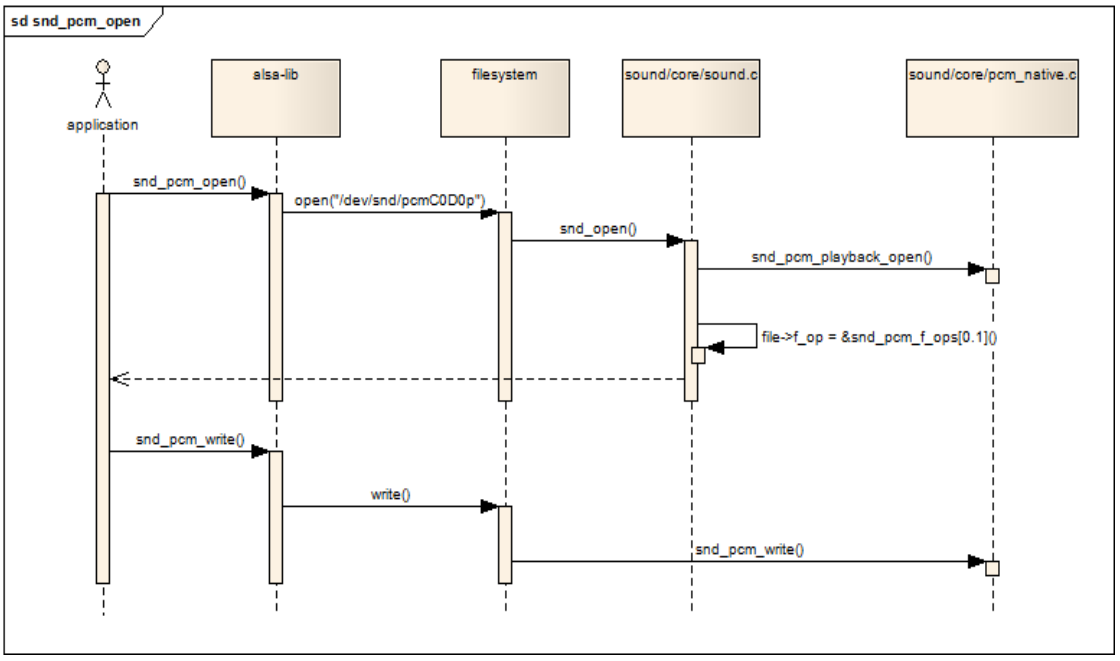


图 4.3.2.1　应用程序操作 pcm 设备

Linux ALSA 声卡驱动之四：Control 设备的创建

1.Control 接口

Control 接口主要让用户空间的应用程序（alsa-lib）可以访问和控制音频 codec 芯片中的多路开关，滑动控件等。对于 Mixer（混音）来说，Control 接口显得尤为重要，从 ALSA 0.9.x 版本开始，所有的 mixer 工作都是通过 control 接口的 API 来实现的。

ALSA 已经为 AC97 定义了完整的控制接口模型，如果你的 Codec 芯片只支持 AC97 接口，你可以不用关心本节的内容。

<sound/control.h>定义了所有的 Control API。如果你要为你的 codec 实现自己的 controls，请在代码中包含该头文件。

1.1Controls 的定义

要自定义一个 Control，我们首先要定义 3 各回调函数：info，get 和 put。然后，定义一个 snd_kcontrol_new 结构：

[c-sharp] view plaincopy

```c
static struct snd_kcontrol_new my_control __devinitdata = {
    .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
    .name = "PCM Playback Switch",
    .index = 0,
    .access = SNDRV_CTL_ELEM_ACCESS_READWRITE,
    .private_value = 0xffff,
    .info = my_control_info,
    .get = my_control_get,
    .put = my_control_put
};
```

iface 字段指出了 control 的类型，alsa 定义了几种类型（SNDDRV_CTL_ELEM_IFACE_XXX），常用的类型是 MIXER，当然也可以定义属于全局的 CARD 类型，也可以定义属于某类设备的类型，例如 HWDEP，PCMRAWMIDI，TIMER 等，这时需要在 device 和 subdevice 字段中指出卡的设备逻辑编号。

name 字段是该 control 的名字，从 ALSA 0.9.x 开始，control 的名字是变得比较重要，因为 control 的作用是按名字来归类的。ALSA 已经预定义了一些 control 的名字，我们再 Control Name 一节详细讨论。

index 字段用于保存该 control 的在该卡中的编号。如果声卡中有不止一个 codec，每个 codec 中有相同名字的 control，这时我们可以通过 index 来区分这些 controls。当 index 为 0 时，则可以忽略这种区分策略。

access 字段包含了该 control 的访问类型。每一个 bit 代表一种访问类型，这些访问类型可以多个“或”运算组合在一起。

private_value 字段包含了一个任意的长整数类型值。该值可以通过 info，get，put 这几个回调函数访问。你可以自己决定如何使用该字段，例如可以把它拆分成多个位域，又或者是一个指针，指向某一个数据结构。

tlv 字段为该 control 提供元数据。

1.2Control 的名字

control 的名字需要遵循一些标准，通常可以分成 3 部分来定义 control 的名字：源--方向--功能。

- 源，可以理解为该 control 的输入端，alsa 已经预定义了一些常用的源，例如：Master，PCM，CD，Line 等等。
- 方向，代表该 control 的数据流向，例如：Playback，Capture，Bypass，Bypass Capture 等等，也可以不定义方向，这时表示该 Control 是双向的（playback 和 capture）。
- 功能，根据 control 的功能，可以是以下字符串：Switch，Volume，Route 等等。

也有一些命名上的特例：

- **全局的 capture 和 playback**　"Capture Source"，"Capture Volume"，"Capture Switch"，它们用于全局的 capture source，switch 和 volume。同理，"Playback Volume"，"Playback Switch"，它们用于全局的输出 switch 和 volume。
- **Tone-controles**　音调控制的开关和音量命名为：Tone Control - XXX，例如，"Tone Control - Switch"，"Tone Control - Bass"，"Tone Control - Center"。
- **3D controls**　3D 控件的命名规则：，"3D Control - Switch"，"3D Control - Center"，"3D Control - Space"。
- **Mic boost**　麦克风音量加强控件命名为："Mic Boost"或"Mic Boost(6dB)"。

访问标志（ACCESS Flags）

Access 字段是一个 bitmask，它保存了改 control 的访问类型。默认的访问类型是：SNDDRV_CTL_ELEM_ACCESS_READWRITE，表明该 control 支持读和写操作。如果 access 字段没有定义（.access==0），此时也认为是 READWRITE 类型。

如果是一个只读 control，access 应该设置为：SNDDRV_CTL_ELEM_ACCESS_READ，这时，我们不必定义 put 回调函数。类似地，如果是只写 control，access 应该设置为：SNDDRV_CTL_ELEM_ACCESS_WRITE，这时，我们不必定义 get 回调函数。

如果 control 的值会频繁地改变（例如：电平表），我们可以使用 VOLATILE 类型，这意味着该 control 会在没有通知的情况下改变，应用程序应该定时地查询该 control 的值。

2.回调函数

2.1 info 回调函数

info 回调函数用于获取 control 的详细信息。它的主要工作就是填充通过参数传入的 snd_ctl_elem_info 对象，以下例子是一个具有单个元素的 boolean 型 control 的 info 回调：

[c-sharp] view plaincopy

```
1.  static int snd_myctl_mono_info(struct snd_kcontrol *kcontrol,
2.      struct snd_ctl_elem_info *uinfo)
3.  {
4.      uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
5.      uinfo->count = 1;
6.      uinfo->value.integer.min = 0;
7.      uinfo->value.integer.max = 1;
8.      return 0;
9.  }
```

type 字段指出该 control 的值类型，值类型可以是 BOOLEAN, INTEGER, ENUMERATED, BYTES,IEC958 和 INTEGER64 之一。count 字段指出了改 control 中包含有多少个元素单元，比如，立体声的音量 control 左右两个声道的音量值，它的 count 字段等于 2。value 字段是一个联合体（union），value 的内容和 control 的类型有关。其中，boolean 和 integer 类型是相同的。

ENUMERATED 类型有些特殊。它的 value 需要设定一个字符串和字符串的索引，请看以下例子：

[c-sharp] view plaincopy

```c
static int snd_myctl_enum_info(struct snd_kcontrol *kcontrol,
struct snd_ctl_elem_info *uinfo)
{
    static char *texts[4] = {
        "First", "Second", "Third", "Fourth"
    };
    uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
    uinfo->count = 1;
    uinfo->value.enumerated.items = 4;
    if (uinfo->value.enumerated.item > 3)
        uinfo->value.enumerated.item = 3;
    strcpy(uinfo->value.enumerated.name,
        texts[uinfo->value.enumerated.item]);
    return 0;
}
```

alsa 已经为我们实现了一些通用的 info 回调函数，例如：snd_ctl_boolean_mono_info()，snd_ctl_boolean_stereo_info()等等。

2.2 get 回调函数

该回调函数用于读取 control 的当前值，并返回给用户空间的应用程序。

[c-sharp] view plaincopy

```c
static int snd_myctl_get(struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_value *ucontrol)
{
    struct mychip *chip = snd_kcontrol_chip(kcontrol);
    ucontrol->value.integer.value[0] = get_some_value(chip);
    return 0;
}
```

value 字段的赋值依赖于 control 的类型（如同 info 回调）。很多声卡的驱动利用它存储硬件寄存器的地址、bit-shift 和 bit-mask，这时，private_value 字段可以按以下例子进行设置：

.private_value = reg | (shift << 16) | (mask << 24);

然后，get 回调函数可以这样实现：

```
static int snd_sbmixer_get_single(struct snd_kcontrol *kcontrol,
    struct snd_ctl_elem_value *ucontrol)

{
    int reg = kcontrol->private_value & 0xff;
    int shift = (kcontrol->private_value >> 16) & 0xff;
    int mask = (kcontrol->private_value >> 24) & 0xff;
    ....

    //根据以上的值读取相应寄存器的值并填入 value 中
}
```

如果 control 的 count 字段大于 1，表示 control 有多个元素单元，get 回调函数也应该为 value 填充多个数值。

2.3 put 回调函数

---

put 回调函数用于把应用程序的控制值设置到 control 中。

[c-sharp] view plaincopy

```
1.  static int snd_myctl_put(struct snd_kcontrol *kcontrol,
2.      struct snd_ctl_elem_value *ucontrol)
3.  {
4.      struct mychip *chip = snd_kcontrol_chip(kcontrol);
5.      int changed = 0;
6.      if (chip->current_value !=
7.          ucontrol->value.integer.value[0]) {
8.          change_current_value(chip,
9.          ucontrol->value.integer.value[0]);
10.         changed = 1;
11.     }
12.     return changed;
13. }
```

如上述例子所示，当 control 的值被改变时，put 回调必须要返回 1，如果值没有被改变，则返回 0。如果发生了错误，则返回一个负数的错误号。

和 get 回调一样，当 control 的 count 大于 1 时，put 回调也要处理多个 control 中的元素值。

3.创建 Controls

当把以上讨论的内容都准备好了以后，我们就可以创建我们自己的 control 了。alsa-driver 为我们提供了两个用于创建 control 的 API：

- snd_ctl_new1()
- snd_ctl_add()

我们可以用以下最简单的方式创建 control：

[c-sharp] view plaincopy

```
1.  err = snd_ctl_add(card, snd_ctl_new1(&my_control, chip));
2.  if (err < 0)
3.      return err;
```

在这里，my_control 是一个之前定义好的 snd_kcontrol_new 对象，chip 对象将会被赋值在 kcontrol->private_data 字段，该字段可以在回调函数中访问。

snd_ctl_new1()会分配一个新的 snd_kcontrol 实例，并把 my_control 中相应的值复制到该实例中，所以，在定义 my_control 时，通常我们可以加上__devinitdata 前缀。snd_ctl_add 则把该 control 绑定到声卡对象 card 当中。

3.1 元数据（Metadata）

很多 mixer control 需要提供以 dB 为单位的信息，我们可以使用 DECLARE_TLV_xxx 宏来定义一些包含这种信息的变量，然后把 control 的 tlv.p 字段指向这些变量，最后，在 access 字段中加上 SNDRV_CTL_ELEM_ACCESS_TLV_READ 标志，就像这样：

static DECLARE_TLV_DB_SCALE(db_scale_my_control, -4050, 150, 0);

```
static struct snd_kcontrol_new my_control __devinitdata = {
    ...
    .access = SNDRV_CTL_ELEM_ACCESS_READWRITE |
          SNDRV_CTL_ELEM_ACCESS_TLV_READ,
    ...
    .tlv.p = db_scale_my_control,
};
```

DECLARE_TLV_DB_SCALE 宏定义的 mixer control，它所代表的值按一个固定的 dB 值的步长变化。该宏的第一个参数是要定义变量的名字，第二个参数是最小值，以 0.01dB 为单位。第三个参数是变化的步长，也是以 0.01dB 为单位。如果该 control 处于最小值时会做出 mute 时，需要把第四个参数设为 1。

DECLARE_TLV_DB_LINEAR 宏定义的 mixer control，它的输出随值的变化而线性变化。 该宏的第一个参数是要定义变量的名字，第二个参数是最小值，以 0.01dB 为单位。第二个参数是最大值，以 0.01dB 为单位。如果该 control 处于最小值时会做出 mute 时，需要把第二个参数设为 TLV_DB_GAIN_MUTE。

这两个宏实际上就是定义一个整形数组，所谓 tlv，就是 Type-Lenght-Value 的意思，数组的第 0 各元素代表数据的类型，第 1 个元素代表数据的长度，第三个元素和之后的元素保存该变量的数据。

3.2.Control 设备的建立

Control 设备和 PCM 设备一样，都属于声卡下的逻辑设备。用户空间的应用程序通过 alsa-lib 访问该 Control 设备，读取或控制 control 的控制状态，从而达到控制音频 Codec 进行各种 Mixer 等控制操作。

Control 设备的创建过程大体上和 PCM 设备的创建过程相同。详细的创建过程可以参考本博的另一篇文章：Linux 音频驱动之三：PCM 设备的创建。下面我们只讨论有区别的地方。

我们需要在我们的驱动程序初始化时主动调用 snd_pcm_new()函数创建 pcm 设备，而 control 设备则在 snd_card_create()内被创建，snd_card_create()通过调用 snd_ctl_create()函数创建 control 设备节点。所以我们无需显式地创建 control 设备，只要建立声卡，control 设备被自动地创建。

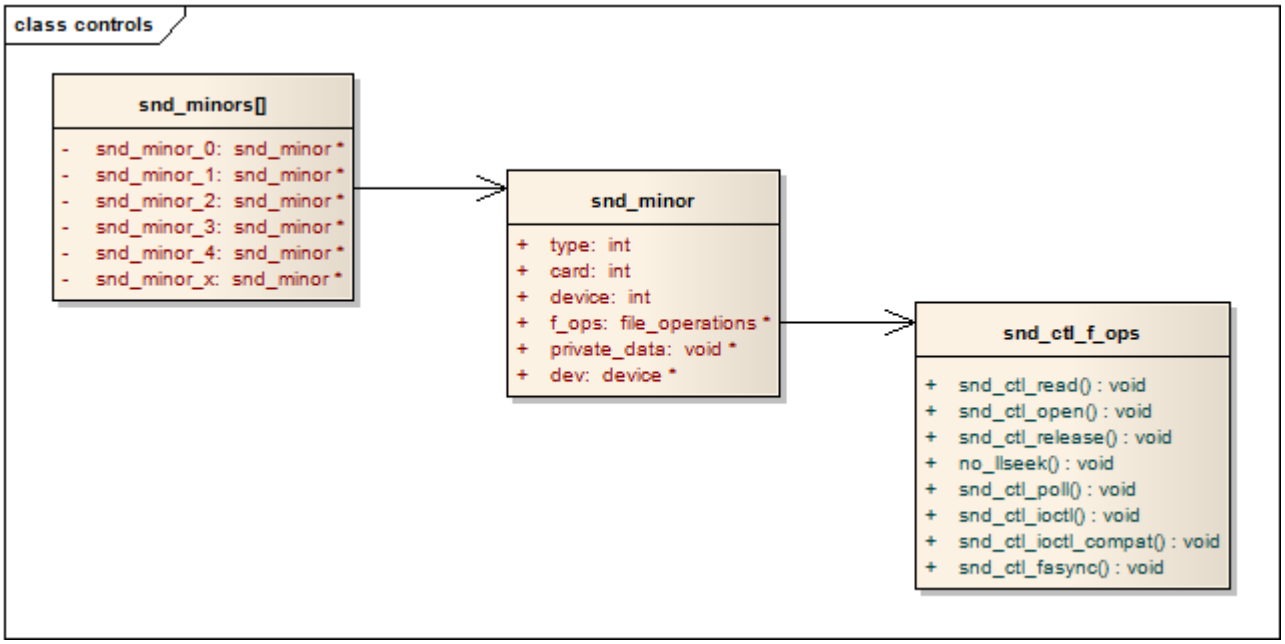
和 pcm 设备一样，control 设备的名字遵循一定的规则：controlCxx，这里的 xx 代表声卡的编号。我们也可以通过代码正是这一点，下面的是 snd_ctl_dev_register()函数的代码：

[c-sharp] view plaincopy

```c
/*
 * registration of the control device
 */
static int snd_ctl_dev_register(struct snd_device *device)
{
    struct snd_card *card = device->device_data;
    int err, cardnum;
    char name[16];

    if (snd_BUG_ON(!card))
        return -ENXIO;
    cardnum = card->number;
    if (snd_BUG_ON(cardnum < 0 || cardnum >= SNDRV_CARDS))
        return -ENXIO;
        /* control 设备的名字 */
    sprintf(name, "controlC%i", cardnum);
    if ((err = snd_register_device(SNDRV_DEVICE_TYPE_CONTROL, card, -1,
                        &snd_ctl_f_ops, card, name)) < 0)
        return err;
    return 0;
}
```

snd_ctl_dev_register()函数会在 snd_card_register()中，即声卡的注册阶段被调用。注册完成后，control 设备的相关信息被保存在 snd_minors[]数组中，用 control 设备的此设备号作索引，即可在 snd_minors[]数组中找出相关的信息。注册完成后的数据结构关系可以用下图进行表述：



control 设备的操作函数入口

用户程序需要打开 control 设备时，驱动程序通过 snd_minors[]全局数组和此设备号，可以获得 snd_ctl_f_ops 结构中的各个回调函数，然后通过这些回调函数访问 control 中的信息和数据（最终会调用 control 的几个回调函数 get，put，info）。详细的代码我就不贴了，大家可以读一下代码：/sound/core/control.c。

Linux ALSA 声卡驱动之五：移动设备中的 ALSA（ASoC）

1. ASoC 的由来

ASoC--ALSA System on Chip ，是建立在标准 ALSA 驱动层上，为了更好地支持嵌入式处理器和移动设备中的音频 Codec 的一套软件体系。在 ASoc 出现之前，内核对于 SoC 中的音频已经有部分的支持，不过会有一些局限性：

- Codec 驱动与 SoC CPU 的底层耦合过于紧密，这种不理想会导致代码的重复，例如，仅是 wm8731 的驱动，当时 Linux 中有分别针对 4 个平台的驱动代码。

- 音频事件没有标准的方法来通知用户，例如耳机、麦克风的插拔和检测，这些事件在移动设备中是非常普通的，而且通常都需要特定于机器的代码进行重新对音频路劲进行配置。
- 当进行播放或录音时，驱动会让整个 codec 处于上电状态，这对于 PC 没问题，但对于移动设备来说，这意味着浪费大量的电量。同时也不支持通过改变过取样频率和偏置电流来达到省电的目的。

ASoC 正是为了解决上述种种问题而提出的，目前已经被整合至内核的代码树中：sound/soc。ASoC 不能单独存在，他只是建立在标准 ALSA 驱动上的一个它必须和标准的 ALSA 驱动框架相结合才能工作。

2. 硬件架构

通常，就像软件领域里的抽象和重用一样，嵌入式设备的音频系统可以被划分为板载硬件（Machine）、Soc（Platform）、Codec 三大部分，如下图所示：

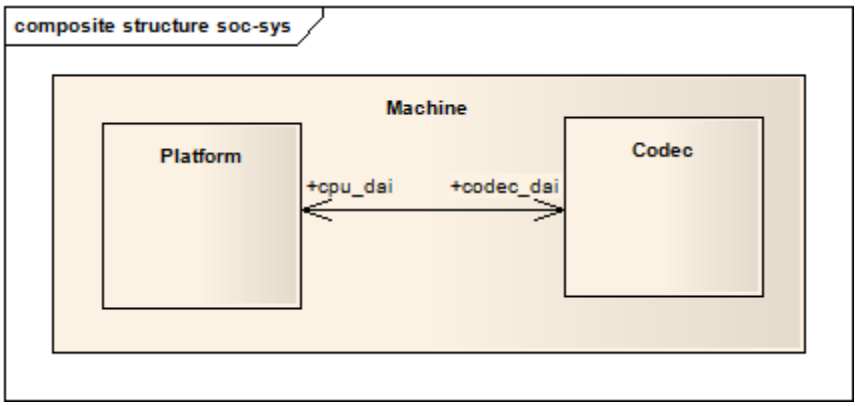


图 2.1　音频系统结构

- **Machine**　是指某一款机器，可以是某款设备，某款开发板，又或者是某款智能手机，由此可以看出 Machine 几乎是不可重用的，每个 Machine 上的硬件实现可能都不一样，CPU 不一样，Codec 不一样，音频的输入、输出设备也不一样，Machine 为 CPU、Codec、输入输出设备提供了一个载体。
- **Platform**　一般是指某一个 SoC 平台，比如 pxaxxx,s3cxxxx,omapxxx 等等，与音频相关的通常包含该 SoC 中的时钟、DMA、I2S、PCM 等等，只要指定了 SoC，

那么我们可以认为它会有一个对应的 Platform，它只与 SoC 相关，与 Machine 无关，这样我们就可以把 Platform 抽象出来，使得同一款 SoC 不用做任何的改动，就可以用在不同的 Machine 中。实际上，把 Platform 认为是某个 SoC 更好理解。

- **Codec** 字面上的意思就是编解码器，Codec 里面包含了 I2S 接口、D/A、A/D、Mixer、PA（功放），通常包含多种输入（Mic、Line-in、I2S、PCM）和多个输出（耳机、喇叭、听筒，Line-out），Codec 和 Platform 一样，是可重用的部件，同一个 Codec 可以被不同的 Machine 使用。嵌入式 Codec 通常通过 I2C 对内部的寄存器进行控制。

3. 软件架构

在软件层面，ASoC 也把嵌入式设备的音频系统同样分为 3 大部分，Machine，Platform 和 Codec。

- **Codec 驱动** ASoC 中的一个重要设计原则就是要求 Codec 驱动是平台无关的，它包含了一些音频的控件（Controls），音频接口，DAMP（动态音频电源管理）的定义和某些 Codec IO 功能。为了保证硬件无关性，任何特定于平台和机器的代码都要移到 Platform 和 Machine 驱动中。所有的 Codec 驱动都要提供以下特性：
    - Codec DAI 和 PCM 的配置信息；
    - Codec 的 IO 控制方式（I2C，SPI 等）；
    - Mixer 和其他的音频控件；
    - Codec 的 ALSA 音频操作接口；

必要时，也可以提供以下功能：

    - DAPM 描述信息；
    - DAPM 事件处理程序；
    - DAC 数字静音控制

- **Platform 驱动** 它包含了该 SoC 平台的音频 DMA 和音频接口的配置和控制(I2S，PCM，AC97 等等）；它也不能包含任何与板子或机器相关的代码。
- **Machine 驱动** Machine 驱动负责处理机器特有的一些控件和音频事件（例如，当播放音频时，需要先行打开一个放大器）；单独的 Platform 和 Codec 驱动是不能工作的，它必须由 Machine 驱动把它们结合在一起才能完成整个设备的音频处理工作。

4. 数据结构

整个 ASoC 是由一些列数据结构组成，要搞清楚 ASoC 的工作机理，必须要理解这一系列数据结构之间的关系和作用，下面的关系图展示了 ASoC 中重要的数据结构之间的关联方式：

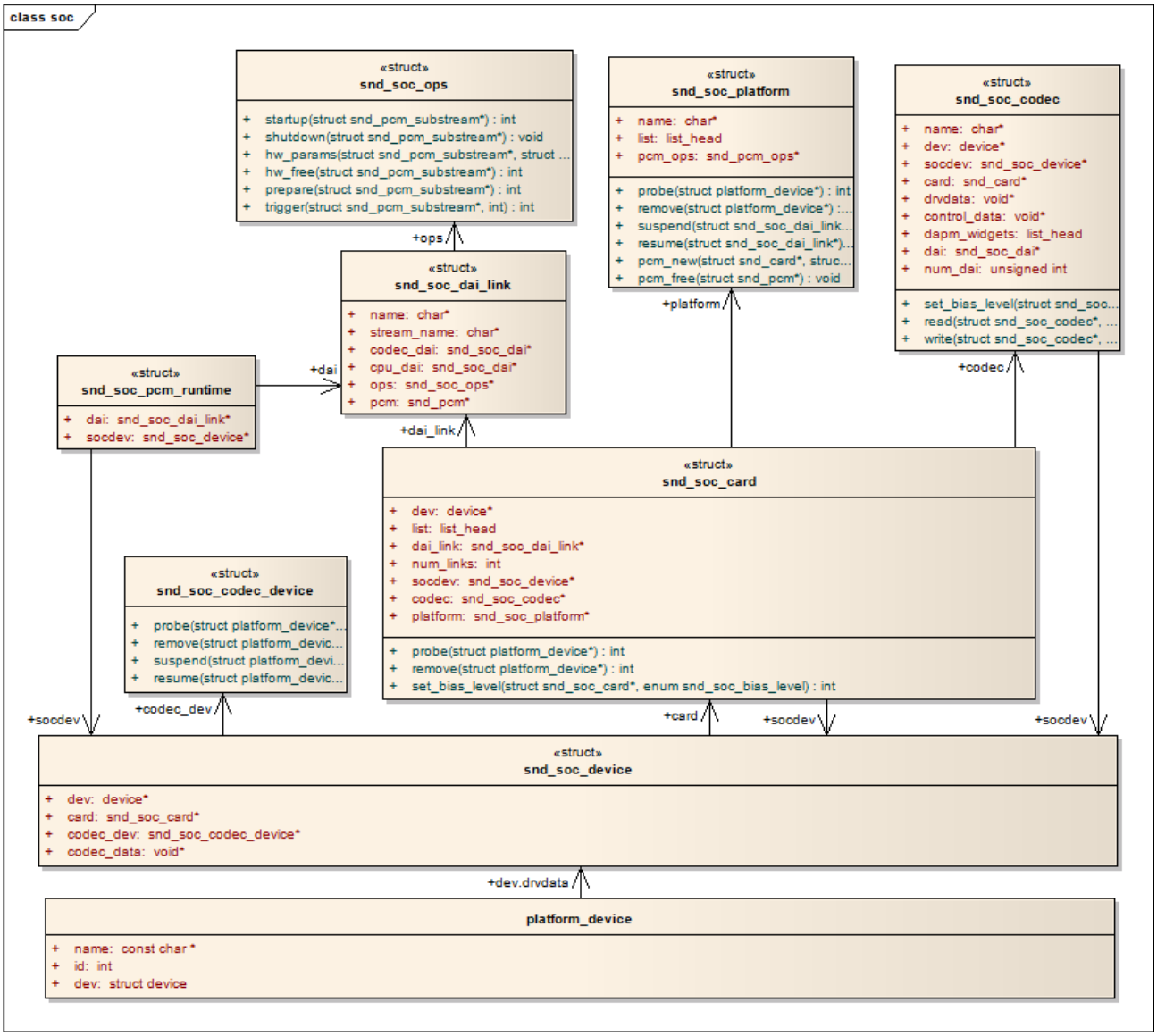


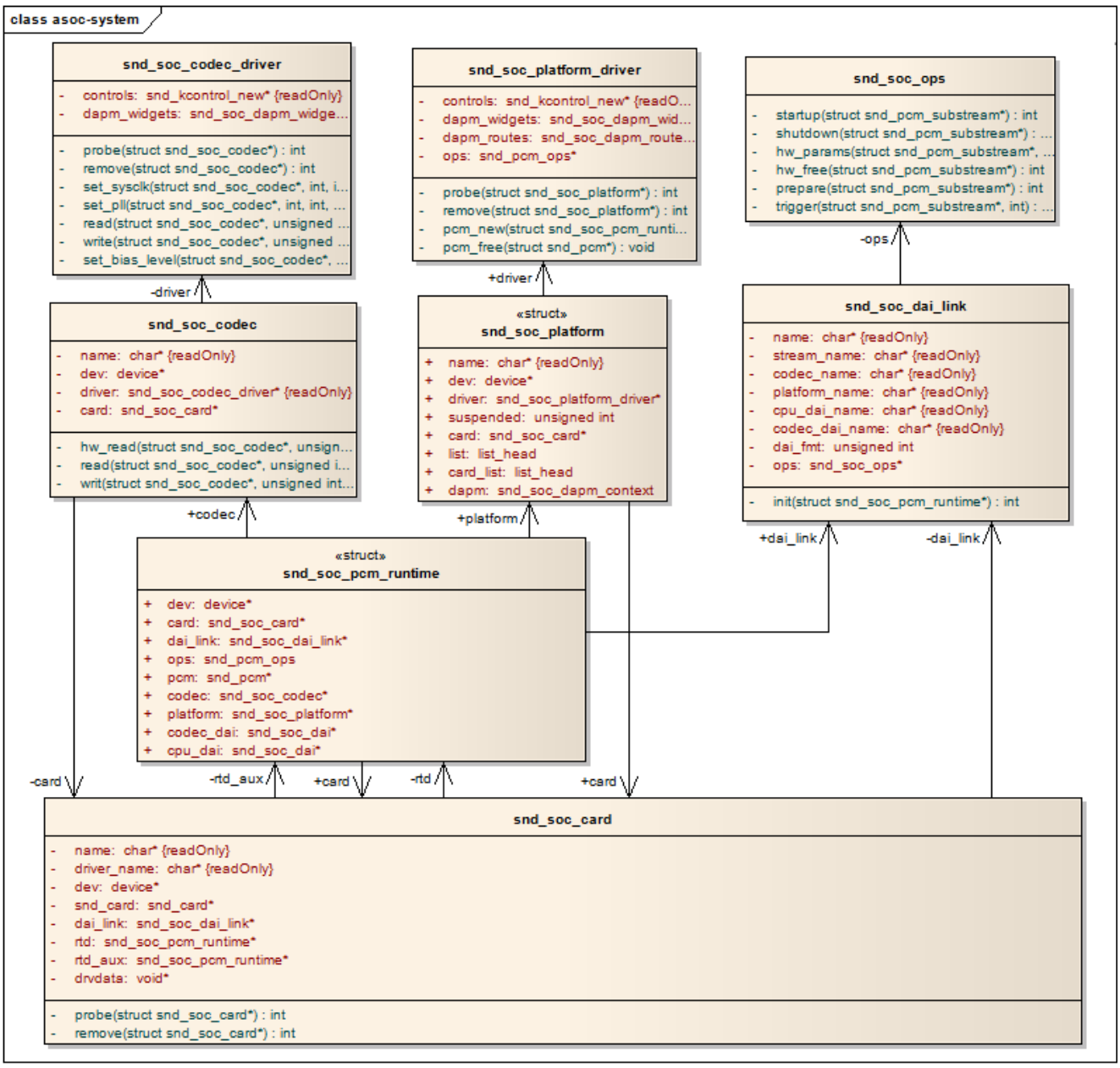
图 4.1　Kernel-2.6.35-ASoC 中各个结构的静态关系

ASoC 把声卡实现为一个 Platform Device，然后利用 Platform_device 结构中的 dev 字段：dev.drvdata，它实际上指向一个 snd_soc_device 结构。可以认为 snd_soc_device 是整个 ASoC 数据结构的根本，由他开始，引出一系列的数据结构用于表述音频的各种特性和功能。snd_soc_device 结构引出了 snd_soc_card 和 soc_codec_device 两个结构，然后 snd_soc_card 又引出了 snd_soc_platform、snd_soc_dai_link 和 snd_soc_codec 结构。如上所述，ASoC 被划分为 Machine、Platform 和 Codec 三大部分，如果从这些数据结构看

来，snd_codec_device 和 snd_soc_card 代表着 Machine 驱动，snd_soc_platform 则代表着 Platform 驱动，snd_soc_codec 和 soc_codec_device 则代表了 Codec 驱动，而 snd_soc_dai_link 则负责连接 Platform 和 Codec。

5. 3.0 版内核对 ASoC 的改进

本来写这篇文章的时候参考的内核版本是 2.6.35，不过有 CSDN 的朋友提出在内核版本 3.0 版本中，ASoC 做了较大的变化。故特意下载了 3.0 的代码，发现确实有所变化，下面先贴出数据结构的静态关系图：



图

5.1　Kernel 3.0 中的 ASoC 数据结构

由上图我们可以看出，3.0 中的数据结构更为合理和清晰，取消了 snd_soc_device 结构，直接用 snd_soc_card 取代了它，并且强化了 snd_soc_pcm_runtime 的作用，同时还增加了另外两个数据结构 snd_soc_codec_driver 和 snd_soc_platform_driver，用于明确代表 Codec 驱动和 Platform 驱动。

后续的章节中将会逐一介绍 Machine 和 Platform 以及 Codec 驱动的工作细节和关联。

Linux ALSA 声卡驱动之六 :ASoC 架构中的 Machine

前面一节的内容我们提到，ASoC 被分为 Machine、Platform 和 Codec 三大部分，其中的 Machine 驱动负责 Platform 和 Codec 之间的耦合以及部分和设备或板子特定的代码，再次引用上一节的内容：Machine 驱动负责处理机器特有的一些控件和音频事件（例如，当播放音频时，需要先行打开一个放大器）；单独的 Platform 和 Codec 驱动是不能工作的，它必须由 Machine 驱动把它们结合在一起才能完成整个设备的音频处理工作。

ASoC 的一切都从 Machine 驱动开始，包括声卡的注册，绑定 Platform 和 Codec 驱动等等，下面就让我们从 Machine 驱动开始讨论吧。

1. 注册 Platform Device

ASoC 把声卡注册为 Platform Device，我们以装配有 WM8994 的一款 Samsung 的开发板 SMDK 为例子做说明，WM8994 是一颗 Wolfson 生产的多功能 Codec 芯片。

代码的位于：/sound/soc/samsung/smdk_wm8994.c，我们关注模块的初始化函数：

[cpp] view plaincopy

```cpp
static int __init smdk_audio_init(void)
{
    int ret;

    smdk_snd_device = platform_device_alloc("soc-audio", -1);
```

```
6.      if (!smdk_snd_device)
7.          return -ENOMEM;
8.
9.      platform_set_drvdata(smdk_snd_device, &smdk);
10.
11.     ret = platform_device_add(smdk_snd_device);
12.     if (ret)
13.         platform_device_put(smdk_snd_device);
14.
15.     return ret;
16. }
```

由此可见，模块初始化时，注册了一个名为 soc-audio 的 Platform 设备，同时把 smdk 设到 platform_device 结构的 dev.drvdata 字段中，这里引出了第一个数据结构 snd_soc_card 的实例 smdk，他的定义如下：

[cpp] view plaincopy

```
1.  static struct snd_soc_dai_link smdk_dai[] = {
2.      { /* Primary DAI i/f */
3.          .name = "WM8994 AIF1",
4.          .stream_name = "Pri_Dai",
5.          .cpu_dai_name = "samsung-i2s.0",
6.          .codec_dai_name = "wm8994-aif1",
7.          .platform_name = "samsung-audio",
8.          .codec_name = "wm8994-codec",
9.          .init = smdk_wm8994_init_paiftx,
10.         .ops = &smdk_ops,
11.     }, { /* Sec_Fifo Playback i/f */
12.         .name = "Sec_FIFO TX",
13.         .stream_name = "Sec_Dai",
14.         .cpu_dai_name = "samsung-i2s.4",
15.         .codec_dai_name = "wm8994-aif1",
16.         .platform_name = "samsung-audio",
17.         .codec_name = "wm8994-codec",
18.         .ops = &smdk_ops,
19.     },
20. };
21.
22. static struct snd_soc_card smdk = {
23.     .name = "SMDK-I2S",
24.     .owner = THIS_MODULE,
25.     .dai_link = smdk_dai,
```

```
26.     .num_links = ARRAY_SIZE(smdk_dai),
27. };
```

通过 snd_soc_card 结构，又引出了 Machine 驱动的另外两个关键数据结构：

- snd_soc_dai_link（实例：smdk_dai[] ）
- snd_soc_ops（实例：smdk_ops ）

其中，snd_soc_dai_link 中，指定了 Platform、Codec、codec_dai、cpu_dai 的名字，稍后 Machine 驱动将会利用这些名字去匹配已经在系统中注册的 platform，codec，dai，这些注册的部件都是在另外相应的 Platform 驱动和 Codec 驱动的代码文件中定义的，这样看来，Machine 驱动的设备初始化代码无非就是选择合适 Platform 和 Codec 以及 dai，用他们填充以上几个数据结构，然后注册 Platform 设备即可。当然还要实现连接 Platform 和 Codec 的 dai_link 对应的 ops 实现，本例就是 smdk_ops，它只实现了 hw_params 函数：smdk_hw_params。

2. 注册 Platform Driver

按照 Linux 的设备模型，有 platform_device，就一定会有 platform_driver。ASoC 的 platform_driver 在以下文件中定义：sound/soc/soc-core.c。

还是先从模块的入口看起：

[cpp] view plaincopy

```
1.  static int __init snd_soc_init(void)
2.  {
3.      ......
4.      return platform_driver_register(&soc_driver);
5.  }
```

soc_driver 的定义如下：

[cpp] view plaincopy

```
1.  /* ASoC platform driver */
2.  static struct platform_driver soc_driver = {
3.      .driver     = {
4.          .name       = "soc-audio",
```

```
5.          .owner      = THIS_MODULE,
6.          .pm     = &soc_pm_ops,
7.      },
8.      .probe      = soc_probe,
9.      .remove     = soc_remove,
10. };
```

我们看到 platform_driver 的 name 字段为 soc-audio，正好与 platform_device 中的名字相同，按照 Linux 的设备模型，platform 总线会匹配这两个名字相同的 device 和 driver，同时会触发 soc_probe 的调用，它正是整个 ASoC 驱动初始化的入口。

3. 初始化入口 soc_probe()

soc_probe 函数本身很简单，它先从 platform_device 参数中取出 snd_soc_card，然后调用 snd_soc_register_card，通过 snd_soc_register_card，为 snd_soc_pcm_runtime 数组申请内存，每一个 dai_link 对应 snd_soc_pcm_runtime 数组的一个单元，然后把 snd_soc_card 中的 dai_link 配置复制到相应的 snd_soc_pcm_runtime 中，最后，大部分的工作都在 snd_soc_instantiate_card 中实现，下面就看看 snd_soc_instantiate_card 做了些什么：

该函数首先利用 card->instantiated 来判断该卡是否已经实例化，如果已经实例化则直接返回，否则遍历每一对 dai_link，进行 codec、platform、dai 的绑定工作，下面是代码的部分选节，详细的代码请直接参考完整的代码树。

[cpp] view plaincopy

```
1.  /* bind DAIs */
2.  for (i = 0; i < card->num_links; i++)
3.      soc_bind_dai_link(card, i);
```

ASoC 定义了三个全局的链表头变量：codec_list、dai_list、platform_list，系统中所有的 Codec、DAI、Platform 都在注册时连接到这三个全局链表上。soc_bind_dai_link 函数逐个扫描这三个链表，根据 card->dai_link[]中的名称进行匹配，匹配后把相应的 codec，dai 和 platform 实例赋值到 card->rtd[]中（snd_soc_pcm_runtime）。经过这个过程后，snd_soc_pcm_runtime:（card->rtd）中保存了本 Machine 中使用的 Codec，DAI 和 Platform 驱动的信息。

snd_soc_instantiate_card 接着初始化 Codec 的寄存器缓存，然后调用标准的 alsa 函数创建声卡实例：

[cpp] view plaincopy

```
1.  /* card bind complete so register a sound card */
2.  ret = snd_card_create(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
3.          card->owner, 0, &card->snd_card);
4.  card->snd_card->dev = card->dev;
5.
6.  card->dapm.bias_level = SND_SOC_BIAS_OFF;
7.  card->dapm.dev = card->dev;
8.  card->dapm.card = card;
9.  list_add(&card->dapm.list, &card->dapm_list);
```

然后，依次调用各个子结构的 probe 函数：

[cpp] view plaincopy

```
1.  /* initialise the sound card only once */
2.  if (card->probe) {
3.      ret = card->probe(card);
4.      if (ret < 0)
5.          goto card_probe_error;
6.  }
7.
8.  /* early DAI link probe */
9.  for (order = SND_SOC_COMP_ORDER_FIRST; order <= SND_SOC_COMP_ORDER_LAST;
10.         order++) {
11.     for (i = 0; i < card->num_links; i++) {
12.         ret = soc_probe_dai_link(card, i, order);
13.         if (ret < 0) {
14.             pr_err("asoc: failed to instantiate card %s: %d\n",
15.                 card->name, ret);
16.             goto probe_dai_err;
17.         }
18.     }
19. }
20.
21. for (i = 0; i < card->num_aux_devs; i++) {
22.     ret = soc_probe_aux_dev(card, i);
23.     if (ret < 0) {
24.         pr_err("asoc: failed to add auxiliary devices %s: %d\n",
25.                card->name, ret);
26.         goto probe_aux_dev_err;
27.     }
28. }
```

在上面的 soc_probe_dai_link()函数中做了比较多的事情，把他展开继续讨论：

[cpp] view plaincopy

```cpp
static int soc_probe_dai_link(struct snd_soc_card *card, int num, int order)

{
        ......
    /* set default power off timeout */
    rtd->pmdown_time = pmdown_time;

    /* probe the cpu_dai */
    if (!cpu_dai->probed &&
            cpu_dai->driver->probe_order == order) {

        if (cpu_dai->driver->probe) {
            ret = cpu_dai->driver->probe(cpu_dai);
        }
        cpu_dai->probed = 1;
        /* mark cpu_dai as probed and add to card dai list */
        list_add(&cpu_dai->card_list, &card->dai_dev_list);
    }

    /* probe the CODEC */
    if (!codec->probed &&
            codec->driver->probe_order == order) {
        ret = soc_probe_codec(card, codec);
    }

    /* probe the platform */
    if (!platform->probed &&
            platform->driver->probe_order == order) {
        ret = soc_probe_platform(card, platform);
    }

    /* probe the CODEC DAI */
    if (!codec_dai->probed && codec_dai->driver->probe_order == order) {
        if (codec_dai->driver->probe) {
            ret = codec_dai->driver->probe(codec_dai);
        }

        /* mark codec_dai as probed and add to card dai list */
        codec_dai->probed = 1;
```

```
39.         list_add(&codec_dai->card_list, &card->dai_dev_list);
40.     }
41.
42.     /* complete DAI probe during last probe */
43.     if (order != SND_SOC_COMP_ORDER_LAST)
44.         return 0;
45.
46.     ret = soc_post_component_init(card, codec, num, 0);
47.     if (ret)
48.         return ret;
49.         ......
50.     /* create the pcm */
51.     ret = soc_new_pcm(rtd, num);
52.         ........
53.     return 0;
54. }
```

该函数出了挨个调用了 codec，dai 和 platform 驱动的 probe 函数外，在最后还调用了 soc_new_pcm()函数用于创建标准 alsa 驱动的 pcm 逻辑设备。现在把该函数的部分代码也贴出来：

[cpp] view plaincopy

```
1.  /* create a new pcm */
2.  int soc_new_pcm(struct snd_soc_pcm_runtime *rtd, int num)
3.  {
4.      ......
5.      struct snd_pcm_ops *soc_pcm_ops = &rtd->ops;
6.
7.      soc_pcm_ops->open    = soc_pcm_open;
8.      soc_pcm_ops->close   = soc_pcm_close;
9.      soc_pcm_ops->hw_params   = soc_pcm_hw_params;
10.     soc_pcm_ops->hw_free = soc_pcm_hw_free;
11.     soc_pcm_ops->prepare = soc_pcm_prepare;
12.     soc_pcm_ops->trigger = soc_pcm_trigger;
13.     soc_pcm_ops->pointer = soc_pcm_pointer;
14.
15.     ret = snd_pcm_new(rtd->card->snd_card, new_name,
16.             num, playback, capture, &pcm);
17.
18.     /* DAPM dai link stream work */
```

```
19.     INIT_DELAYED_WORK(&rtd->delayed_work, close_delayed_work);
20.
21.     rtd->pcm = pcm;
22.     pcm->private_data = rtd;
23.     if (platform->driver->ops) {
24.         soc_pcm_ops->mmap = platform->driver->ops->mmap;
25.         soc_pcm_ops->pointer = platform->driver->ops->pointer;
26.         soc_pcm_ops->ioctl = platform->driver->ops->ioctl;
27.         soc_pcm_ops->copy = platform->driver->ops->copy;
28.         soc_pcm_ops->silence = platform->driver->ops->silence;
29.         soc_pcm_ops->ack = platform->driver->ops->ack;
30.         soc_pcm_ops->page = platform->driver->ops->page;
31.     }
32.
33.     if (playback)
34.         snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, soc_pcm_ops);
35.
36.     if (capture)
37.         snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, soc_pcm_ops);
38.
39.     if (platform->driver->pcm_new) {
40.         ret = platform->driver->pcm_new(rtd);
41.         if (ret < 0) {
42.             pr_err("asoc: platform pcm constructor failed\n");
43.             return ret;
44.         }
45.     }
46.
47.     pcm->private_free = platform->driver->pcm_free;
48.     return ret;
49. }
```

该函数首先初始化 snd_soc_runtime 中的 snd_pcm_ops 字段，也就是 rtd->ops 中的部分成员，例如 open，close，hw_params 等，紧接着调用标准 alsa 驱动中的创建 pcm 的函数 snd_pcm_new()创建声卡的 pcm 实例，pcm 的 private_data 字段设置为该 runtime 变量 rtd，然后用 platform 驱动中的 snd_pcm_ops 替换部分 pcm 中的 snd_pcm_ops 字段，最后，调用 platform 驱动的 pcm_new 回调，该回调实现该 platform 下的 dma 内存申请和 dma 初始化等相关工作。到这里，声卡和他的 pcm 实例创建完成。

回到 snd_soc_instantiate_card 函数，完成 snd_card 和 snd_pcm 的创建后，接着对 dapm 和 dai 支持的格式做出一些初始化合设置工作后，调用了 card->late_probe(card)进行一些最后的初始化合设置工作，最后则是调用标准 alsa 驱动的声卡注册函数对声卡进行注册：

[cpp] view plaincopy

```cpp
if (card->late_probe) {
    ret = card->late_probe(card);
    if (ret < 0) {
        dev_err(card->dev, "%s late_probe() failed: %d\n",
            card->name, ret);
        goto probe_aux_dev_err;
    }
}

snd_soc_dapm_new_widgets(&card->dapm);

if (card->fully_routed)
    list_for_each_entry(codec, &card->codec_dev_list, card_list)
        snd_soc_dapm_auto_nc_codec_pins(codec);

ret = snd_card_register(card->snd_card);
if (ret < 0) {
    printk(KERN_ERR "asoc: failed to register soundcard for %s\n", card->name);
    goto probe_aux_dev_err;
}
```

至此，整个 Machine 驱动的初始化已经完成，通过各个子结构的 probe 调用，实际上，也完成了部分 Platfrom 驱动和 Codec 驱动的初始化工作，整个过程可以用一下的序列图表示：

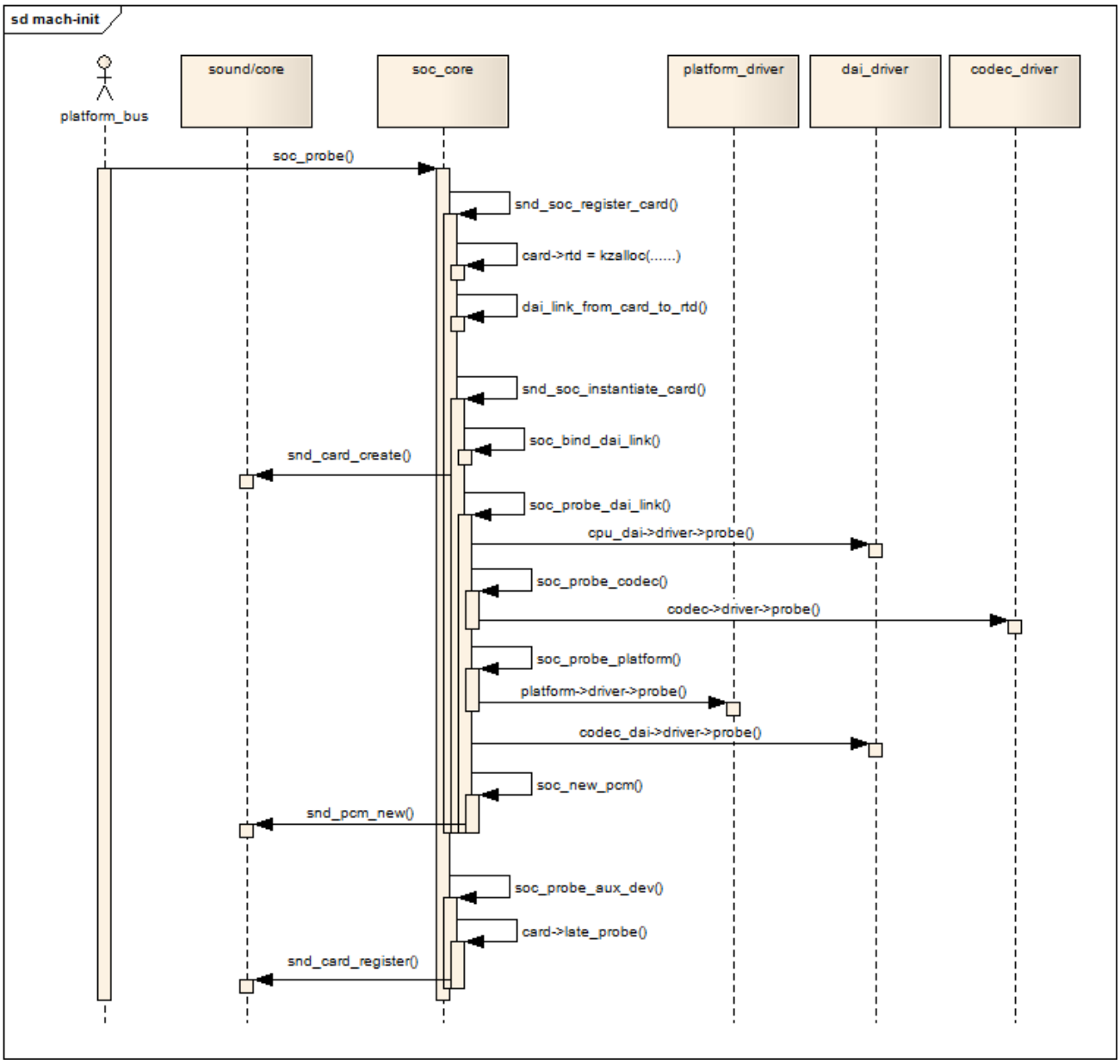


图 3.1　基于 3.0 内核　soc_probe 序列图

下面的序列图是本文章第一个版本，基于内核 2.6.35，大家也可以参考一下两个版本的差异：

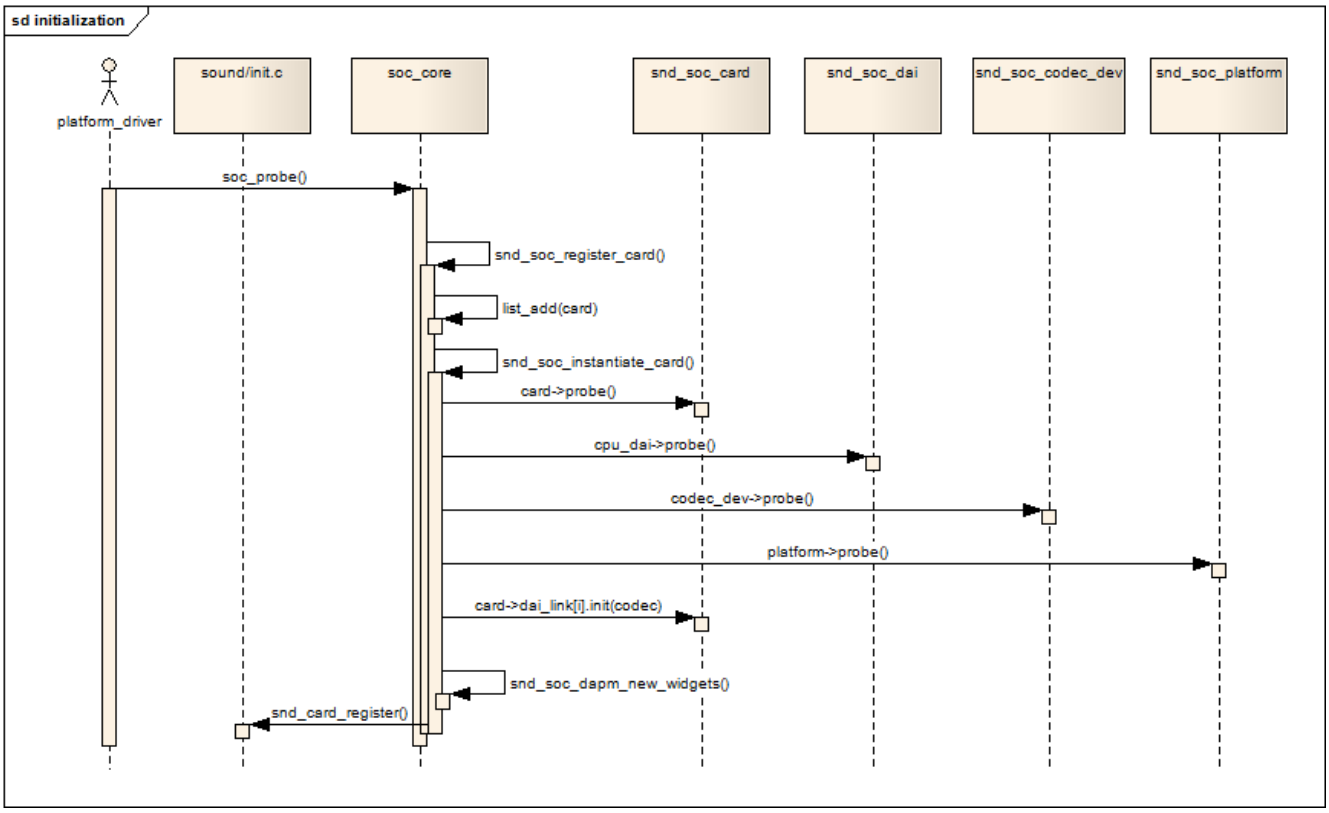


图 3.2　基于

2.6.35　soc_probe 序列图

Linux ALSA 声卡驱动之七：ASoC 架构中的 Codec

1. Codec 简介

在移动设备中，Codec 的作用可以归结为 4 种，分别是：

- 对 PCM 等信号进行 D/A 转换，把数字的音频信号转换为模拟信号
- 对 Mic、Linein 或者其他输入源的模拟信号进行 A/D 转换，把模拟的声音信号转变 CPU 能够处理的数字信号
- 对音频通路进行控制，比如播放音乐，收听调频收音机，又或者接听电话时，音频信号在 codec 内的流通路线是不一样的
- 对音频信号做出相应的处理，例如音量控制，功率放大，EQ 控制等等

ASoC 对 Codec 的这些功能都定义好了一些列相应的接口，以方便地对 Codec 进行控制。ASoC 对 Codec 驱动的一个基本要求是：驱动程序的代码必须要做到平台无关性，以方便同一个 Codec 的代码不经修改即可用在不同的平台上。以下的讨论基于 wolfson 的 Codec 芯片 WM8994，kernel 的版本 3.3.x。

2. ASoC 中对 Codec 的数据抽象

描述 Codec 的最主要的几个数据结构分别是：snd_soc_codec，snd_soc_codec_driver，snd_soc_dai，snd_soc_dai_driver，其中的 snd_soc_dai 和 snd_soc_dai_driver 在 ASoC 的 Platform 驱动中也会使用到，Platform 和 Codec 的 DAI 通过 snd_soc_dai_link 结构，在 Machine 驱动中进行绑定连接。下面我们先看看这几个结构的定义，这里我只贴出我要关注的字段，详细的定义请参照：/include/sound/soc.h。

snd_soc_codec：

[html] view plaincopy

```
1.  /* SoC Audio Codec device */
2.  struct snd_soc_codec {
3.      const char *name;  /* Codec 的名字*/
4.      struct device *dev; /* 指向 Codec 设备的指针 */
5.      const struct snd_soc_codec_driver *driver; /* 指向该 codec 的驱动的指
    针 */
6.      struct snd_soc_card *card;     /* 指向 Machine 驱动的 card 实例 */
7.      int num_dai; /* 该 Codec 数字接口的个数，目前越来越多的 Codec 带有多个 I2S 或者是
    PCM 接口 */
8.      int (*volatile_register)(...);  /* 用于判定某一寄存器是否是 volatile */
9.      int (*readable_register)(...);  /* 用于判定某一寄存器是否可读 */
10.     int (*writable_register)(...);  /* 用于判定某一寄存器是否可写 */
11.
12.     /* runtime */
13.     ......
14.     /* codec IO */
15.     void *control_data; /* 该指针指向的结构用于对 codec 的控制，通常和 read，write
    字段联合使用 */
16.     enum snd_soc_control_type control_type;/* 可以是 SND_SOC_SPI，SND_SOC_I2C，
    SND_SOC_REGMAP 中的一种 */
17.     unsigned int (*read)(struct snd_soc_codec *, unsigned int);  /* 读取
    Codec 寄存器的函数 */
18.     int (*write)(struct snd_soc_codec *, unsigned int, unsigned int);  /* 写
    入 Codec 寄存器的函数 */
19.     /* dapm */
20.     struct snd_soc_dapm_context dapm;  /* 用于 DAPM 控件 */
21. };
```

snd_soc_codec_driver:

[html] view plaincopy

```
1.  /* codec driver */
2.  struct snd_soc_codec_driver {
3.      /* driver ops */
4.      int (*probe)(struct snd_soc_codec *);  /* codec 驱动的 probe 函数，由
    snd_soc_instantiate_card 回调 */
5.      int (*remove)(struct snd_soc_codec *);
6.      int (*suspend)(struct snd_soc_codec *);  /* 电源管理 */
7.      int (*resume)(struct snd_soc_codec *);  /* 电源管理 */
8.
9.      /* Default control and setup, added after probe() is run */
10.     const struct snd_kcontrol_new *controls;  /* 音频控件指针 */
11.     const struct snd_soc_dapm_widget *dapm_widgets;  /* dapm 部件指针 */
12.     const struct snd_soc_dapm_route *dapm_routes;  /* dapm 路由指针 */
13.
14.     /* codec wide operations */
15.     int (*set_sysclk)(...);  /* 时钟配置函数 */
16.     int (*set_pll)(...);  /* 锁相环配置函数 */
17.
18.     /* codec IO */
19.     unsigned int (*read)(...);  /* 读取 codec 寄存器函数 */
20.     int (*write)(...);  /* 写入 codec 寄存器函数 */
21.     int (*volatile_register)(...);  /* 用于判定某一寄存器是否是 volatile */
22.     int (*readable_register)(...);  /* 用于判定某一寄存器是否可读 */
23.     int (*writable_register)(...);  /* 用于判定某一寄存器是否可写 */
24.
25.     /* codec bias level */
26.     int (*set_bias_level)(...);  /* 偏置电压配置函数 */
27.
28. };
```

snd_soc_dai:

[html] view plaincopy

```
1.  /*
2.   * Digital Audio Interface runtime data.
3.   *
4.   * Holds runtime data for a DAI.
5.   */
6.  struct snd_soc_dai {
7.      const char *name;  /* dai 的名字 */
```

```
8.      struct device *dev;  /* 设备指针 */
9.
10.     /* driver ops */
11.     struct snd_soc_dai_driver *driver;  /* 指向 dai 驱动结构的指针 */
12.
13.     /* DAI runtime info */
14.     unsigned int capture_active:1;      /* stream is in use */
15.     unsigned int playback_active:1;     /* stream is in use */
16.
17.     /* DAI DMA data */
18.     void *playback_dma_data;  /* 用于管理 playback dma */
19.     void *capture_dma_data;  /* 用于管理 capture dma */
20.
21.     /* parent platform/codec */
22.     union {
23.         struct snd_soc_platform *platform;  /* 如果是 cpu dai，指向所绑定的平
    台 */
24.         struct snd_soc_codec *codec;  /* 如果是 codec dai 指向所绑定的
    codec */
25.     };
26.     struct snd_soc_card *card;  /* 指向 Machine 驱动中的 crad 实例 */
27. };
```

snd_soc_dai_driver:

[html] view plaincopy

```
1.  /*
2.   * Digital Audio Interface Driver.
3.   *
4.   * Describes the Digital Audio Interface in terms of its ALSA, DAI and AC97

5.   * operations and capabilities. Codec and platform drivers will register thi
    s
6.   * structure for every DAI they have.
7.   *
8.   * This structure covers the clocking, formating and ALSA operations for eac
    h
9.   * interface.
10.  */
11. struct snd_soc_dai_driver {
12.     /* DAI description */
13.     const char *name;  /* dai 驱动名字 */
14.
15.     /* DAI driver callbacks */
```

```
    int (*probe)(struct snd_soc_dai *dai);  /* dai 驱动的 probe 函数，由 snd_soc_instantiate_card 回调 */
    int (*remove)(struct snd_soc_dai *dai);
    int (*suspend)(struct snd_soc_dai *dai);  /* 电源管理 */
    int (*resume)(struct snd_soc_dai *dai);

    /* ops */
    const struct snd_soc_dai_ops *ops;  /* 指向本 dai 的 snd_soc_dai_ops 结构 */

    /* DAI capabilities */
    struct snd_soc_pcm_stream capture;  /* 描述 capture 的能力 */
    struct snd_soc_pcm_stream playback;  /* 描述 playback 的能力 */
};
```

snd_soc_dai_ops 用于实现该 dai 的控制盒参数配置：

[html] view plaincopy

```
struct snd_soc_dai_ops {
    /*
     * DAI clocking configuration, all optional.
     * Called by soc_card drivers, normally in their hw_params.
     */
    int (*set_sysclk)(...);
    int (*set_pll)(...);
    int (*set_clkdiv)(...);
    /*
     * DAI format configuration
     * Called by soc_card drivers, normally in their hw_params.
     */
    int (*set_fmt)(...);
    int (*set_tdm_slot)(...);
    int (*set_channel_map)(...);
    int (*set_tristate)(...);
    /*
     * DAI digital mute - optional.
     * Called by soc-core to minimise any pops.
     */
    int (*digital_mute)(...);
    /*
     * ALSA PCM audio operations - all optional.
     * Called by soc-core during audio PCM operations.
     */
    int (*startup)(...);
```

```
    void (*shutdown)(...);
    int (*hw_params)(...);
    int (*hw_free)(...);
    int (*prepare)(...);
    int (*trigger)(...);
    /*
     * For hardware based FIFO caused delay reporting.
     * Optional.
     */
    snd_pcm_sframes_t (*delay)(...);
};
```

3. Codec 的注册

因为 Codec 驱动的代码要做到平台无关性，要使得 Machine 驱动能够使用该 Codec，Codec 驱动的首要任务就是确定 snd_soc_codec 和 snd_soc_dai 的实例，并把它们注册到系统中，注册后的 codec 和 dai 才能为 Machine 驱动所用。以 WM8994 为例，对应的代码位置：/sound/soc/codecs/wm8994.c，模块的入口函数注册了一个 platform driver：

[html] view plaincopy

```
static struct platform_driver wm8994_codec_driver = {
    .driver = {
            .name = "wm8994-codec",
            .owner = THIS_MODULE,
            },
    .probe = wm8994_probe,
    .remove = __devexit_p(wm8994_remove),
};

module_platform_driver(wm8994_codec_driver);
```

有 platform driver，必定会有相应的 platform device，这个 platform device 的来源后面再说，显然，platform driver 注册后，probe 回调将会被调用，这里是 wm8994_probe 函数：

[html] view plaincopy

```
static int __devinit wm8994_probe(struct platform_device *pdev)
{
    return snd_soc_register_codec(&pdev->dev, &soc_codec_dev_wm8994,
            wm8994_dai, ARRAY_SIZE(wm8994_dai));
}
```

其中，soc_codec_dev_wm8994 和 wm8994_dai 的定义如下（代码中定义了 3 个 dai，这里只列出第一个）：

[html] view plaincopy

```
1.  static struct snd_soc_codec_driver soc_codec_dev_wm8994 = {
2.      .probe =    wm8994_codec_probe,
3.      .remove =   wm8994_codec_remove,
4.      .suspend =  wm8994_suspend,
5.      .resume =   wm8994_resume,
6.      .set_bias_level = wm8994_set_bias_level,
7.      .reg_cache_size = WM8994_MAX_REGISTER,
8.      .volatile_register = wm8994_soc_volatile,
9.  };
```

[html] view plaincopy

```
1.  static struct snd_soc_dai_driver wm8994_dai[] = {
2.      {
3.          .name = "wm8994-aif1",
4.          .id = 1,
5.          .playback = {
6.              .stream_name = "AIF1 Playback",
7.              .channels_min = 1,
8.              .channels_max = 2,
9.              .rates = WM8994_RATES,
10.             .formats = WM8994_FORMATS,
11.         },
12.         .capture = {
13.             .stream_name = "AIF1 Capture",
14.             .channels_min = 1,
15.             .channels_max = 2,
16.             .rates = WM8994_RATES,
17.             .formats = WM8994_FORMATS,
18.          },
19.         .ops = &wm8994_aif1_dai_ops,
20.     },
21.     ......
22. }
```

可见，Codec 驱动的第一个步骤就是定义 snd_soc_codec_driver 和 snd_soc_dai_driver 的实例，然后调用 snd_soc_register_codec 函数对 Codec 进行注册。进入 snd_soc_register_codec 函数看看：

首先，它申请了一个 snd_soc_codec 结构的实例：

[html] view plaincopy

```
1.  codec = kzalloc(sizeof(struct snd_soc_codec), GFP_KERNEL);
```

确定 codec 的名字，这个名字很重要，Machine 驱动定义的 snd_soc_dai_link 中会指定每个 link 的 codec 和 dai 的名字，进行匹配绑定时就是通过和这里的名字比较，从而找到该 Codec 的！

[html] view plaincopy

```
/* create CODEC component name */
    codec->name = fmt_single_name(dev, &codec->id);
```

然后初始化它的各个字段，多数字段的值来自上面定义的 snd_soc_codec_driver 的实例 soc_codec_dev_wm8994：

[html] view plaincopy

```
codec->write = codec_drv->write;
codec->read = codec_drv->read;
codec->volatile_register = codec_drv->volatile_register;
codec->readable_register = codec_drv->readable_register;
codec->writable_register = codec_drv->writable_register;
codec->dapm.bias_level = SND_SOC_BIAS_OFF;
codec->dapm.dev = dev;
codec->dapm.codec = codec;
codec->dapm.seq_notifier = codec_drv->seq_notifier;
codec->dapm.stream_event = codec_drv->stream_event;
codec->dev = dev;
codec->driver = codec_drv;
codec->num_dai = num_dai;
```

在做了一些寄存器缓存的初始化和配置工作后，通过 snd_soc_register_dais 函数对本 Codec 的 dai 进行注册：

[html] view plaincopy

```
/* register any DAIs */
if (num_dai) {
    ret = snd_soc_register_dais(dev, dai_drv, num_dai);
    if (ret < 0)
        goto fail;
}
```

最后，它把 codec 实例链接到全局链表 codec_list 中，并且调用 snd_soc_instantiate_cards 函数触发 Machine 驱动进行一次匹配绑定操作：

[html] view plaincopy

```
list_add(&codec->list, &codec_list);
snd_soc_instantiate_cards();
```

上面的 snd_soc_register_dais 函数其实也是和 snd_soc_register_codec 类似，显示为每个 snd_soc_dai 实例分配内存，确定 dai 的名字，用 snd_soc_dai_driver 实例的字段对它进行必要初始化，最后把该 dai 链接到全局链表 dai_list 中，和 Codec 一样，最后也会调用 snd_soc_instantiate_cards 函数触发一次匹配绑定的操作。

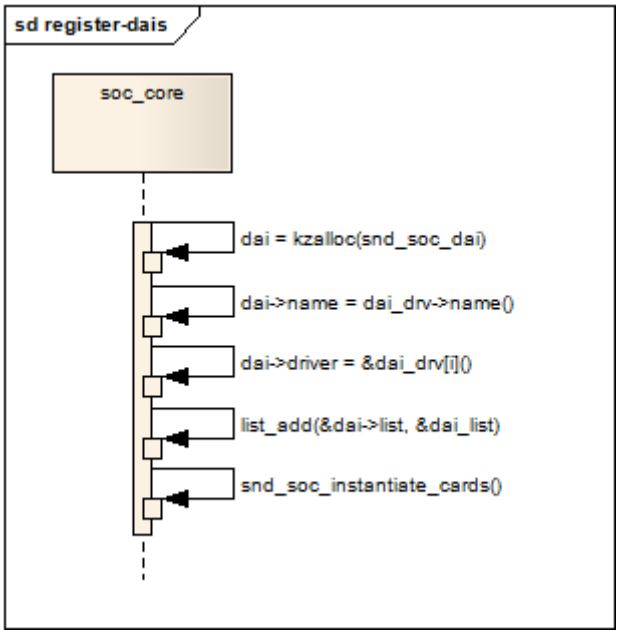


图 3.1 dai 的注册

关于 snd_soc_instantiate_cards 函数，请参阅另一篇博文：Linux 音频驱动之六：ASoC 架构中的 Machine。

4. mfd 设备

前面已经提到，codec 驱动把自己注册为一个 platform driver，那对应的 platform device 在哪里定义？答案是在以下代码文件中：/drivers/mfd/wm8994-core.c。

WM8994 本身具备多种功能，除了 codec 外，它还有作为 LDO 和 GPIO 使用，这几种功能共享一些 IO 和中断资源，linux 为这种设备提供了一套标准的实现方法：mfd 设备。其基本思想是为这些功能的公共部分实现一个父设备，以便共享某些系统资源和功能，然后每个子功能实现为它的子设备，这样既共享了资源和代码，又能实现合理的设备层次结构，主要利用到的 API 就是：mfd_add_devices()，mfd_remove_devices()，mfd_cell_enable()，mfd_cell_disable()，mfd_clone_cell()。

回到 wm8994-core.c 中，因为 WM8994 使用 I2C 进行内部寄存器的存取，它首先注册了一个 I2C 驱动：

```
1.  static struct i2c_driver wm8994_i2c_driver = {
2.      .driver = {
3.          .name = "wm8994",
4.          .owner = THIS_MODULE,
5.          .pm = &wm8994_pm_ops,
6.          .of_match_table = wm8994_of_match,
7.      },
8.      .probe = wm8994_i2c_probe,
9.      .remove = wm8994_i2c_remove,
10.     .id_table = wm8994_i2c_id,
11. };
12.
13. static int __init wm8994_i2c_init(void)
14. {
15.     int ret;
16.
17.     ret = i2c_add_driver(&wm8994_i2c_driver);
18.     if (ret != 0)
19.         pr_err("Failed to register wm8994 I2C driver: %d\n", ret);
20.
21.     return ret;
22. }
23. module_init(wm8994_i2c_init);
```

进入 wm8994_i2c_probe()函数，它先申请了一个 wm8994 结构的变量，该变量被作为这个 I2C 设备的 driver_data 使用，上面已经讲过，codec 作为它的子设备，将会取出并使用这个 driver_data。接下来，本函数利用 regmap_init_i2c()初始化并获得一个 regmap 结构，该结构主要用于后续基于 regmap 机制的寄存器 I/O，关于 regmap 我们留在后面再讲。最后，通过 wm8994_device_init()来添加 mfd 子设备：

[html] view plaincopy

```
1.  static int wm8994_i2c_probe(struct i2c_client *i2c,
2.                  const struct i2c_device_id *id)
3.  {
4.      struct wm8994 *wm8994;
5.      int ret;
6.      wm8994 = devm_kzalloc(&i2c->dev, sizeof(struct wm8994), GFP_KERNEL);
7.      i2c_set_clientdata(i2c, wm8994);
8.      wm8994->dev = &i2c->dev;
9.      wm8994->irq = i2c->irq;
10.     wm8994->type = id->driver_data;
11.     wm8994->regmap = regmap_init_i2c(i2c, &wm8994_base_regmap_config);
12.
```

```
13.     return wm8994_device_init(wm8994, i2c->irq);
14. }
```

继续进入 wm8994_device_init()函数，它首先为两个 LDO 添加 mfd 子设备：

[html] view plaincopy

```
1.  /* Add the on-chip regulators first for bootstrapping */
2.  ret = mfd_add_devices(wm8994->dev, -1,
3.                  wm8994_regulator_devs,
4.                  ARRAY_SIZE(wm8994_regulator_devs),
5.                  NULL, 0);
```

因为 WM1811，WM8994，WM8958 三个芯片功能类似，因此这三个芯片都使用了 WM8994 的代码，所以 wm8994_device_init()接下来根据不同的芯片型号做了一些初始化动作，这部分的代码就不贴了。接着，从 platform_data 中获得部分配置信息：

[html] view plaincopy

```
1.  if (pdata) {
2.      wm8994->irq_base = pdata->irq_base;
3.      wm8994->gpio_base = pdata->gpio_base;
4.
5.      /* GPIO configuration is only applied if it's non-zero */
6.      ......
7.  }
```

最后，初始化 irq，然后添加 codec 子设备和 gpio 子设备：

[html] view plaincopy

```
1.  wm8994_irq_init(wm8994);
2.
3.  ret = mfd_add_devices(wm8994->dev, -1,
4.                  wm8994_devs, ARRAY_SIZE(wm8994_devs),
5.                  NULL, 0);
```

经过以上这些处理后，作为父设备的 I2C 设备已经准备就绪，它的下面挂着 4 个子设备：ldo-0，ldo-1，codec，gpio。其中，codec 子设备的加入，它将会和前面所讲 codec 的 platform driver 匹配，触发 probe 回调完成下面所说的 codec 驱动的初始化工作。

5. Codec 初始化

Machine 驱动的初始化，codec 和 dai 的注册，都会调用 snd_soc_instantiate_cards()进行一次声卡和 codec，dai，platform 的匹配绑定过程，这里所说的绑定，正如 Machine 驱动一文中所描述，就是通过 3 个全局链表，按名字进行匹配，把匹配的 codec，dai 和 platform 实例赋值给声卡每对 dai 的 snd_soc_pcm_runtime 变量中。一旦绑定成功，将会使得 codec

和 dai 驱动的 probe 回调被调用，codec 的初始化工作就在该回调中完成。对于 WM8994，该回调就是 wm8994_codec_probe 函数：

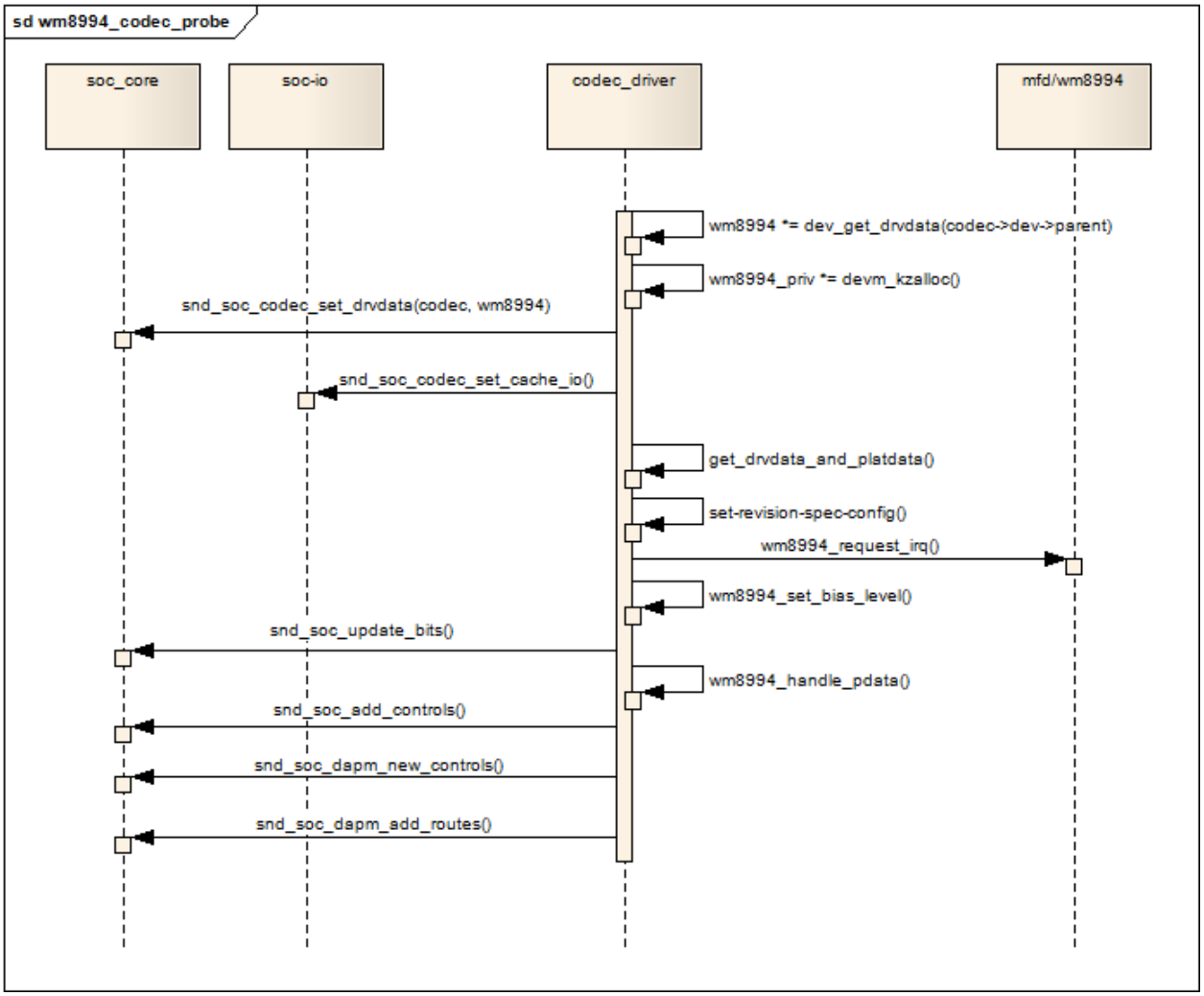


图 5.1　wm8994_codec_probe

- 取出父设备的 driver_data，其实就是上一节的 wm8994 结构变量，取出其中的 regmap 字段，复制到 codec 的 control_data 字段中；
- 申请一个 wm8994_priv 私有数据结构，并把它设为 codec 设备的 driver_data；
- 通过 snd_soc_codec_set_cache_io 初始化 regmap io，完成这一步后，就可以使用 API：snd_soc_read()，snd_soc_write()对 codec 的寄存器进行读写了；
- 把父设备的 driver_data（struct wm8994）和 platform_data 保存到私有结构 wm8994_priv 中；
- 因为要同时支持 3 个芯片型号，这里要根据芯片的型号做一些特定的初始化工作；
- 申请必要的几个中断；
- 设置合适的偏置电平；
- 通过 snd_soc_update_bits 修改某些寄存器；

- 根据父设备的 platform_data，完成特定于平台的初始化配置；
- 添加必要的 control，dapm 部件进而 dapm 路由信息；

至此，codec 驱动的初始化完成。

5. regmap-io

我们知道，要想对 codec 进行控制，通常都是通过读写它的内部寄存器完成的，读写的接口通常是 I2C 或者是 SPI 接口，不过每个 codec 芯片寄存器的比特位组成都有所不同，寄存器地址的比特位也有所不同。例如 WM8753 的寄存器地址是 7bits，数据是 9bits，WM8993 的寄存器地址是 8bits，数据也是 16bits，而 WM8994 的寄存器地址是 16bits，数据也是 16bits。在 kernel3.1 版本，内核引入了一套 regmap 机制和相关的 API，这样就可以用统一的操作来实现对这些多样的寄存器的控制。regmap 使用起来也相对简单：

- 为 codec 定义一个 regmap_config 结构实例，指定 codec 寄存器的地址和数据位等信息；
- 根据 codec 的控制总线类型，调用以下其中一个函数，得到一个指向 regmap 结构的指针：
  - struct regmap *regmap_init_i2c(struct i2c_client *i2c, const struct regmap_config *config);
  - struct regmap *regmap_init_spi(struct spi_device *dev, const struct regmap_config *config);
- 把获得的 regmap 结构指针赋值给 codec->control_data；
- 调用 soc-io 的 api：snd_soc_codec_set_cache_io 使得 soc-io 和 regmap 进行关联；

完成以上步骤后，codec 驱动就可以使用诸如 snd_soc_read、snd_soc_write、snd_soc_update_bits 等 API 对 codec 的寄存器进行读写了。

Linux ALSA 声卡驱动之八：ASoC 架构中的 Platform

1. Platform 驱动在 ASoC 中的作用

前面几章内容已经说过，ASoC 被分为 Machine，Platform 和 Codec 三大部件，Platform 驱动的主要作用是完成音频数据的管理，最终通过 CPU 的数字音频接口（DAI）把音频数据传送给 Codec 进行处理，最终由 Codec 输出驱动耳机或者是喇叭的音信信号。在具体实

现上，ASoC 有把 Platform 驱动分为两个部分：snd_soc_platform_driver 和 snd_soc_dai_driver。其中，platform_driver 负责管理音频数据，把音频数据通过 dma 或其他操作传送至 cpu dai 中，dai_driver 则主要完成 cpu 一侧的 dai 的参数配置，同时也会通过一定的途径把必要的 dma 等参数与 snd_soc_platform_driver 进行交互。

2. snd_soc_platform_driver 的注册

通常，ASoC 把 snd_soc_platform_driver 注册为一个系统的 platform_driver，不要被这两个相像的术语所迷惑，前者只是针对 ASoC 子系统的，后者是来自 Linux 的设备驱动模型。我们要做的就是：

- 定义一个 snd_soc_platform_driver 结构的实例；
- 在 platform_driver 的 probe 回调中利用 ASoC 的 API：snd_soc_register_platform() 注册上面定义的实例；
- 实现 snd_soc_platform_driver 中的各个回调函数；

以 kernel3.3 中的/sound/soc/samsung/dma.c 为例：

[cpp] view plaincopy

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

static int __devinit samsung_asoc_platform_probe(struct platform_device *pdev)
{
    return snd_soc_register_platform(&pdev->dev, &samsung_asoc_platform);
}

static int __devexit samsung_asoc_platform_remove(struct platform_device *pdev)
{
    snd_soc_unregister_platform(&pdev->dev);
    return 0;
}

static struct platform_driver asoc_dma_driver = {
    .driver = {
        .name = "samsung-audio",
        .owner = THIS_MODULE,
    },
```

```
23.
24.     .probe = samsung_asoc_platform_probe,
25.     .remove = __devexit_p(samsung_asoc_platform_remove),
26. };
27.
28. module_platform_driver(asoc_dma_driver);
```

snd_soc_register_platform() 该函数用于注册一个 snd_soc_platform，只有注册以后，它才可以被 Machine 驱动使用。它的代码已经清晰地表达了它的实现过程：

- 为 snd_soc_platform 实例申请内存；
- 从 platform_device 中获得它的名字，用于 Machine 驱动的匹配工作；
- 初始化 snd_soc_platform 的字段；
- 把 snd_soc_platform 实例连接到全局链表 platform_list 中；
- 调用 snd_soc_instantiate_cards，触发声卡的 machine、platform、codec、dai 等的匹配工作；

3. cpu 的 snd_soc_dai driver 驱动的注册

dai 驱动通常对应 cpu 的一个或几个 I2S/PCM 接口，与 snd_soc_platform 一样，dai 驱动也是实现为一个 platform driver，实现一个 dai 驱动大致可以分为以下几个步骤：

- 定义一个 snd_soc_dai_driver 结构的实例；
- 在对应的 platform_driver 中的 probe 回调中通过 API：snd_soc_register_dai 或者 snd_soc_register_dais，注册 snd_soc_dai 实例；
- 实现 snd_soc_dai_driver 结构中的 probe、suspend 等回调；
- 实现 snd_soc_dai_driver 结构中的 snd_soc_dai_ops 字段中的回调函数；

snd_soc_register_dai 这个函数在上一篇介绍 codec 驱动的博文中已有介绍，请参考：Linux ALSA 声卡驱动之七：ASoC 架构中的 Codec。

snd_soc_dai 该结构在 snd_soc_register_dai 函数中通过动态内存申请获得， 简要介绍一下几个重要字段：

- driver 指向关联的 snd_soc_dai_driver 结构，由注册时通过参数传入；
- playback_dma_data 用于保存该 dai 播放 stream 的 dma 信息，例如 dma 的目标地址，dma 传送单元大小和通道号等；
- capture_dma_data 同上，用于录音 stream；
- platform 指向关联的 snd_soc_platform 结构；

snd_soc_dai_driver 该结构需要自己根据不同的soc芯片进行定义，关键字段介绍如下：

- probe、remove 回调函数，分别在声卡加载和卸载时被调用；
- suspend、resume 电源管理回调函数；
- ops 指向 snd_soc_dai_ops 结构，用于配置和控制该 dai；
- playback snd_soc_pcm_stream 结构，用于指出该 dai 支持的声道数，码率，数据格式等能力；
- capture snd_soc_pcm_stream 结构，用于指出该 dai 支持的声道数，码率，数据格式等能力；

4. snd_soc_dai_driver 中的 ops 字段

ops 字段指向一个 snd_soc_dai_ops 结构，该结构实际上是一组回调函数的集合，dai 的配置和控制几乎都是通过这些回调函数来实现的，这些回调函数基本可以分为 3 大类，驱动程序可以根据实际情况实现其中的一部分：

工作时钟配置函数 通常由 machine 驱动调用：

- set_sysclk 设置 dai 的主时钟；
- set_pll 设置 PLL 参数；
- set_clkdiv 设置分频系数；
- dai 的格式配置函数 通常由 machine 驱动调用：
- set_fmt 设置 dai 的格式；
- set_tdm_slot 如果 dai 支持时分复用，用于设置时分复用的 slot；
- set_channel_map 声道的时分复用映射设置；
- set_tristate 设置 dai 引脚的状态，当与其他 dai 并联使用同一引脚时需要使用该回调；

标准的 snd_soc_ops 回调 通常由 soc-core 在进行 PCM 操作时调用：

- startup
- shutdown
- hw_params
- hw_free
- prepare
- trigger

抗 pop，pop 声　由 soc-core 调用：

- digital_mute

以下这些 api 通常被 machine 驱动使用，machine 驱动在他的 snd_pcm_ops 字段中的 hw_params 回调中使用这些 api：

- snd_soc_dai_set_fmt()　实际上会调用 snd_soc_dai_ops 或者 codec driver 中的 set_fmt 回调；
- snd_soc_dai_set_pll()　实际上会调用 snd_soc_dai_ops 或者 codec driver 中的 set_pll 回调；
- snd_soc_dai_set_sysclk()　实际上会调用 snd_soc_dai_ops 或者 codec driver 中的 set_sysclk 回调；
- snd_soc_dai_set_clkdiv()　实际上会调用 snd_soc_dai_ops 或者 codec driver 中的 set_clkdiv 回调；

snd_soc_dai_set_fmt(struct snd_soc_dai *dai, unsigned int fmt)的第二个参数 fmt 在这里特别说一下，ASoC 目前只是用了它的低 16 位，并且为它专门定义了一些宏来方便我们使用：

bit 0-3 用于设置接口的格式：

[cpp] view plaincopy

```
#define SND_SOC_DAIFMT_I2S      1 /* I2S mode */
#define SND_SOC_DAIFMT_RIGHT_J      2 /* Right Justified mode */
#define SND_SOC_DAIFMT_LEFT_J       3 /* Left Justified mode */
#define SND_SOC_DAIFMT_DSP_A        4 /* L data MSB after FRM LRC */
#define SND_SOC_DAIFMT_DSP_B        5 /* L data MSB during FRM LRC */
#define SND_SOC_DAIFMT_AC97     6 /* AC97 */
#define SND_SOC_DAIFMT_PDM      7 /* Pulse density modulation */
```

bit 4-7 用于设置接口时钟的开关特性：

[cpp] view plaincopy

```
#define SND_SOC_DAIFMT_CONT     (1 << 4) /* continuous clock */
#define SND_SOC_DAIFMT_GATED        (2 << 4) /* clock is gated */
```

bit 8-11 用于设置接口时钟的相位：

[cpp] view plaincopy

```
1.  #define SND_SOC_DAIFMT_NB_NF        (1 << 8) /* normal bit clock + frame */

2.  #define SND_SOC_DAIFMT_NB_IF        (2 << 8) /* normal BCLK + inv FRM */
3.  #define SND_SOC_DAIFMT_IB_NF        (3 << 8) /* invert BCLK + nor FRM */
4.  #define SND_SOC_DAIFMT_IB_IF        (4 << 8) /* invert BCLK + FRM */
```

bit 12-15 用于设置接口主从格式：

[cpp] view plaincopy

```
1.  #define SND_SOC_DAIFMT_CBM_CFM      (1 << 12) /* codec clk & FRM master */
2.  #define SND_SOC_DAIFMT_CBS_CFM      (2 << 12) /* codec clk slave & FRM maste
    r */
3.  #define SND_SOC_DAIFMT_CBM_CFS      (3 << 12) /* codec clk master & frame sl
    ave */
4.  #define SND_SOC_DAIFMT_CBS_CFS      (4 << 12) /* codec clk & FRM slave */
```

5. snd_soc_platform_driver 中的 ops 字段

该 ops 字段是一个 snd_pcm_ops 结构，实现该结构中的各个回调函数是 soc platform 驱动的主要工作，他们基本都涉及 dma 操作以及 dma buffer 的管理等工作。下面介绍几个重要的回调函数：

ops.open

当应用程序打开一个 pcm 设备时，该函数会被调用，通常，该函数会使用 snd_soc_set_runtime_hwparams()设置 substream 中的 snd_pcm_runtime 结构里面的 hw_params 相关字段，然后为 snd_pcm_runtime 的 private_data 字段申请一个私有结构，用于保存该平台的 dma 参数。

ops.hw_params

驱动的 hw_params 阶段，该函数会被调用。通常，该函数会通过 snd_soc_dai_get_dma_data 函数获得对应的 dai 的 dma 参数，获得的参数一般都会保存在 snd_pcm_runtime 结构的 private_data 字段。然后通过 snd_pcm_set_runtime_buffer 函数设置 snd_pcm_runtime 结构中的 dma buffer 的地址和大小等参数。要注意的是，该回调可能会被多次调用，具体实现时要小心处理多次申请资源的问题。

ops.prepare

正式开始数据传送之前会调用该函数，该函数通常会完成 dma 操作的必要准备工作。

ops.trigger

数据传送的开始，暂停，恢复和停止时，该函数会被调用。

ops.pointer

该函数返回传送数据的当前位置。

6. 音频数据的 dma 操作

soc-platform 驱动的最主要功能就是要完成音频数据的传送，大多数情况下，音频数据都是通过 dma 来完成的。

6.1. 申请 dma buffer

因为 dma 的特殊性，dma buffer 是一块特殊的内存，比如有的平台规定只有某段地址范围的内存才可以进行 dma 操作，而多数嵌入式平台还要求 dma 内存的物理地址是连续的，以方便 dma 控制器对内存的访问。在 ASoC 架构中，dma buffer 的信息保存在 snd_pcm_substream 结构的 snd_dma_buffer *buf 字段中，它的定义如下

[cpp] view plaincopy

```cpp
struct snd_dma_buffer {
    struct snd_dma_device dev;  /* device type */
    unsigned char *area;    /* virtual pointer */
    dma_addr_t addr;    /* physical address */
    size_t bytes;       /* buffer size in bytes */
    void *private_data; /* private for allocator; don't touch */
};
```

那么，在哪里完成了 snd_dam_buffer 结构的初始化赋值操作呢？答案就在 snd_soc_platform_driver 的 pcm_new 回调函数中，还是以/sound/soc/samsung/dma.c 为例：

[cpp] view plaincopy

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

static int __devinit samsung_asoc_platform_probe(struct platform_device *pdev)
{
    return snd_soc_register_platform(&pdev->dev, &samsung_asoc_platform);
}
```

pcm_new 字段指向了 dma_new 函数，dma_new 函数进一步为 playback 和 capture 分别调用 preallocate_dma_buffer 函数，我们看看 preallocate_dma_buffer 函数的实现：

[cpp] view plaincopy

```cpp
static int preallocate_dma_buffer(struct snd_pcm *pcm, int stream)
{
    struct snd_pcm_substream *substream = pcm->streams[stream].substream;
    struct snd_dma_buffer *buf = &substream->dma_buffer;
    size_t size = dma_hardware.buffer_bytes_max;

    pr_debug("Entered %s\n", __func__);

    buf->dev.type = SNDRV_DMA_TYPE_DEV;
    buf->dev.dev = pcm->card->dev;
    buf->private_data = NULL;
    buf->area = dma_alloc_writecombine(pcm->card->dev, size,
                       &buf->addr, GFP_KERNEL);
    if (!buf->area)
        return -ENOMEM;
    buf->bytes = size;
    return 0;
}
```

该函数先是获得事先定义好的 buffer 大小，然后通过 dma_alloc_weitecombine 函数分配 dma 内存，然后完成 substream->dma_buffer 的初始化赋值工作。上述的 pcm_new 回调会在声卡的建立阶段被调用，调用的详细的过程请参考 Linux ALSAs 声卡驱动之六：ASoC 架构中的 Machine 中的图 3.1。

在声卡的 hw_params 阶段，snd_soc_platform_driver 结构的 ops->hw_params 会被调用，在该回调中，通常会使用 api：snd_pcm_set_runtime_buffer()把 substream->dma_buffer 的数值拷贝到 substream->runtime 的相关字段中（.dma_area, .dma_addr, .dma_bytes），这样以后就可以通过 substream->runtime 获得这些地址和大小信息了。

dma buffer 获得后，即是获得了 dma 操作的源地址，那么目的地址在哪里？其实目的地址当然是在 dai 中，也就是前面介绍的 snd_soc_dai 结构的 playback_dma_data 和 capture_dma_data 字段中，而这两个字段的值也是在 hw_params 阶段，由 snd_soc_dai_driver 结构的 ops->hw_params 回调，利用 api：snd_soc_dai_set_dma_data 进行设置的。紧随其后，snd_soc_platform_driver 结构的 ops->hw_params 回调利用 api：snd_soc_dai_get_dma_data 获得这些 dai 的 dma 信息，其中就包括了 dma 的目的地址信息。这些 dma 信息通常还会被保存在 substream->runtime->private_data 中，以便在 substream 的整个生命周期中可以随时获得这些信息，从而完成对 dma 的配置和操作。

6.2 dma buffer 管理

播放时，应用程序把音频数据源源不断地写入 dma buffer 中，然后相应 platform 的 dma 操作则不停地从该 buffer 中取出数据，经 dai 送往 codec 中。录音时则正好相反，codec 源源不断地把 A/D 转换好的音频数据经过 dai 送入 dma buffer 中，而应用程序则不断地从该 buffer 中读走音频数据。

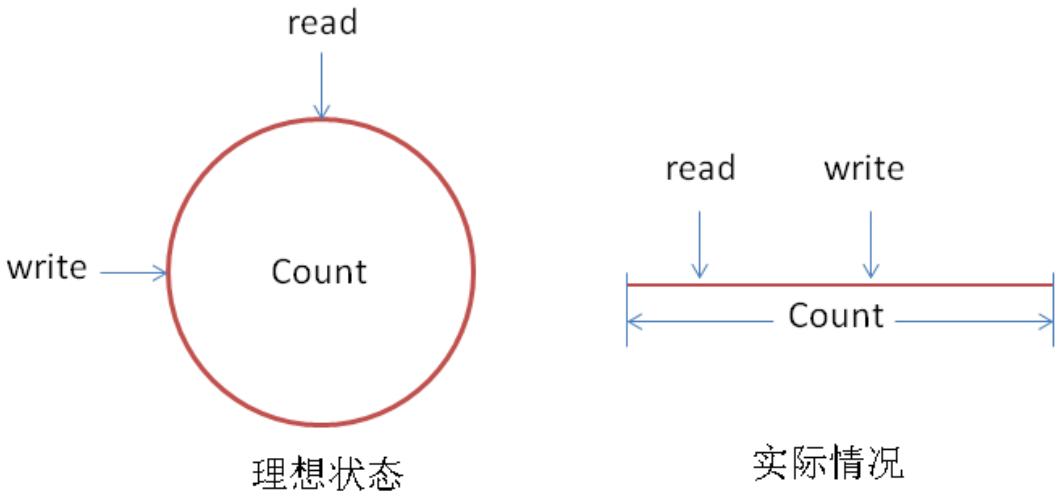


图 6.2.1 环形缓冲区

环形缓冲区正好适合用于这种情景的 buffer 管理，理想情况下，大小为 Count 的缓冲区具备一个读指针和写指针，我们期望他们都可以闭合地做环形移动，但是实际的情况确实：缓冲区通常都是一段连续的地址，他是有开始和结束两个边界，每次移动之前都必须进行一次

判断，当指针移动到末尾时就必须人为地让他回到起始位置。在实际应用中，我们通常都会把这个大小为 Count 的缓冲区虚拟成一个大小为 n*Count 的逻辑缓冲区，相当于理想状态下的圆形绕了 n 圈之后，然后把这段总的距离拉平为一段直线，每一圈对应直线中的一段，因为 n 比较大，所以大多数情况下不会出现读写指针的换位的情况（如果不对 buffer 进行扩展，指针到达末端后，回到起始端时，两个指针的前后相对位置会发生互换）。扩展后的逻辑缓冲区在计算剩余空间可条件判断时相对方便。alsa driver 也使用了该方法对 dma buffer 进行管理：

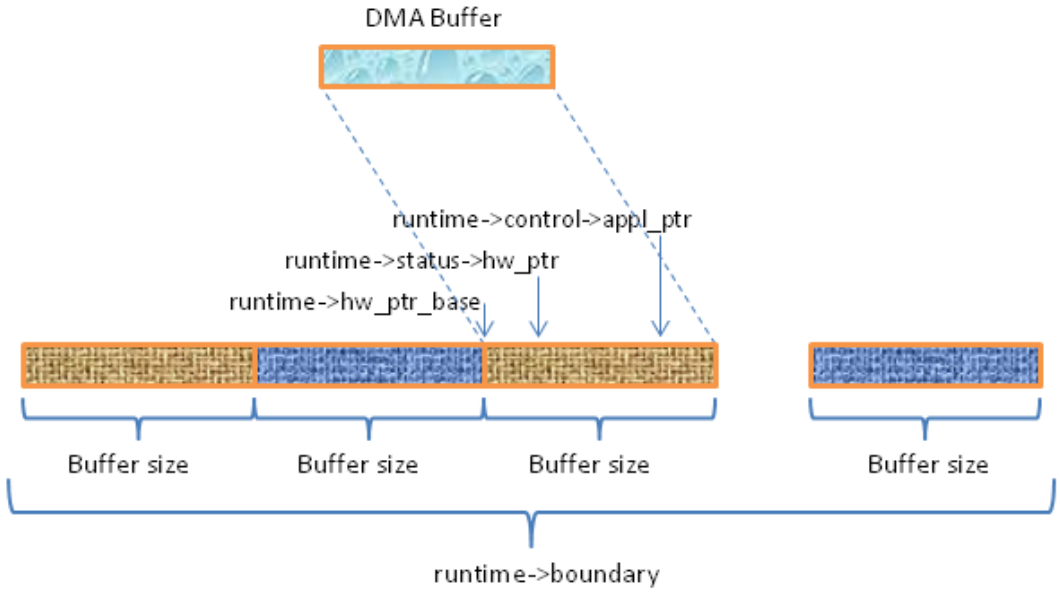


图 6.2.2　alsa driver 缓冲区管理

snd_pcm_runtime 结构中，使用了四个相关的字段来完成这个逻辑缓冲区的管理：

- snd_pcm_runtime.hw_ptr_base　环形缓冲区每一圈的基地址，当读写指针越过一圈后，它按 buffer size 进行移动；
- snd_pcm_runtime.status->hw_ptr　硬件逻辑位置，播放时相当于读指针，录音时相当于写指针；
- snd_pcm_runtime.control->appl_ptr　应用逻辑位置，播放时相当于写指针，录音时相当于读指针；
- snd_pcm_runtime.boundary　扩展后的逻辑缓冲区大小，通常是(2^n)*size；

通过这几个字段，我们可以很容易地获得缓冲区的有效数据，剩余空间等信息，也可以很容易地把当前逻辑位置映射回真实的 dma buffer 中。例如，获得播放缓冲区的空闲空间：

[csharp] view plaincopy

```cpp
static inline snd_pcm_uframes_t snd_pcm_playback_avail(struct snd_pcm_runtime *runtime)
{
    snd_pcm_sframes_t avail = runtime->status->hw_ptr + runtime->buffer_size - runtime->control->appl_ptr;
    if (avail < 0)
        avail += runtime->boundary;
    else if ((snd_pcm_uframes_t) avail >= runtime->boundary)
        avail -= runtime->boundary;
    return avail;
}
```

要想映射到真正的缓冲区位置，只要减去 runtime->hw_ptr_base 即可。下面的 api 用于更新这几个指针的当前位置：

[cpp] view plaincopy

```cpp
int snd_pcm_update_hw_ptr(struct snd_pcm_substream *substream)
```

所以要想通过 snd_pcm_playback_avail 等函数获得正确的信息前，应该先要调用这个 api 更新指针位置。

以播放(playback)为例，我现在知道至少有 3 个途径可以完成对 dma buffer 的写入：

- 应用程序调用 alsa-lib 的 snd_pcm_writei、snd_pcm_writen 函数；
- 应用程序使用 ioctl：SNDRV_PCM_IOCTL_WRITEI_FRAMES 或 SNDRV_PCM_IOCTL_WRITEN_FRAMES；
- 应用程序使用 alsa-lib 的 snd_pcm_mmap_begin/snd_pcm_mmap_commit;

以上几种方式最终把数据写入 dma buffer 中，然后修改 runtime->control->appl_ptr 的值。播放过程中，通常会配置成每一个 period size 生成一个 dma 中断，中断处理函数最重要的任务就是：

- 更新 dma 的硬件的当前位置，该数值通常保存在 runtime->private_data 中；
- 调用 snd_pcm_period_elapsed 函数，该函数会进一步调用 snd_pcm_update_hw_ptr0 函数更新上述所说的 4 个缓冲区管理字段，然后唤醒相应的等待进程；

[cpp] view plaincopy

```
<span style="font-family:Arial, Verdana, sans-serif;"><span style="white-space: normal;"></span></span><pre name="code" class="cpp">void snd_pcm_period_elapsed(struct snd_pcm_substream *substream)
{
    struct snd_pcm_runtime *runtime;
    unsigned long flags;

    if (PCM_RUNTIME_CHECK(substream))
        return;
    runtime = substream->runtime;

    if (runtime->transfer_ack_begin)
        runtime->transfer_ack_begin(substream);

    snd_pcm_stream_lock_irqsave(substream, flags);
    if (!snd_pcm_running(substream) ||
        snd_pcm_update_hw_ptr0(substream, 1) < 0)
        goto _end;

    if (substream->timer_running)
        snd_timer_interrupt(substream->timer, 1);
 _end:
    snd_pcm_stream_unlock_irqrestore(substream, flags);
    if (runtime->transfer_ack_end)
        runtime->transfer_ack_end(substream);
    kill_fasync(&runtime->fasync, SIGIO, POLL_IN);
}
</pre>如果设置了 transfer_ack_begin 和 transfer_ack_end 回调，snd_pcm_period_elapsed 还会调用这两个回调函数。<br>
<br>
<pre></pre>
<pre></pre>
<pre></pre>
```

7. 图说代码

最后，反正图也画了，好与不好都传上来供参考一下，以下这张图表达了 ASoC 中 Platform 驱动的几个重要数据结构之间的关系:

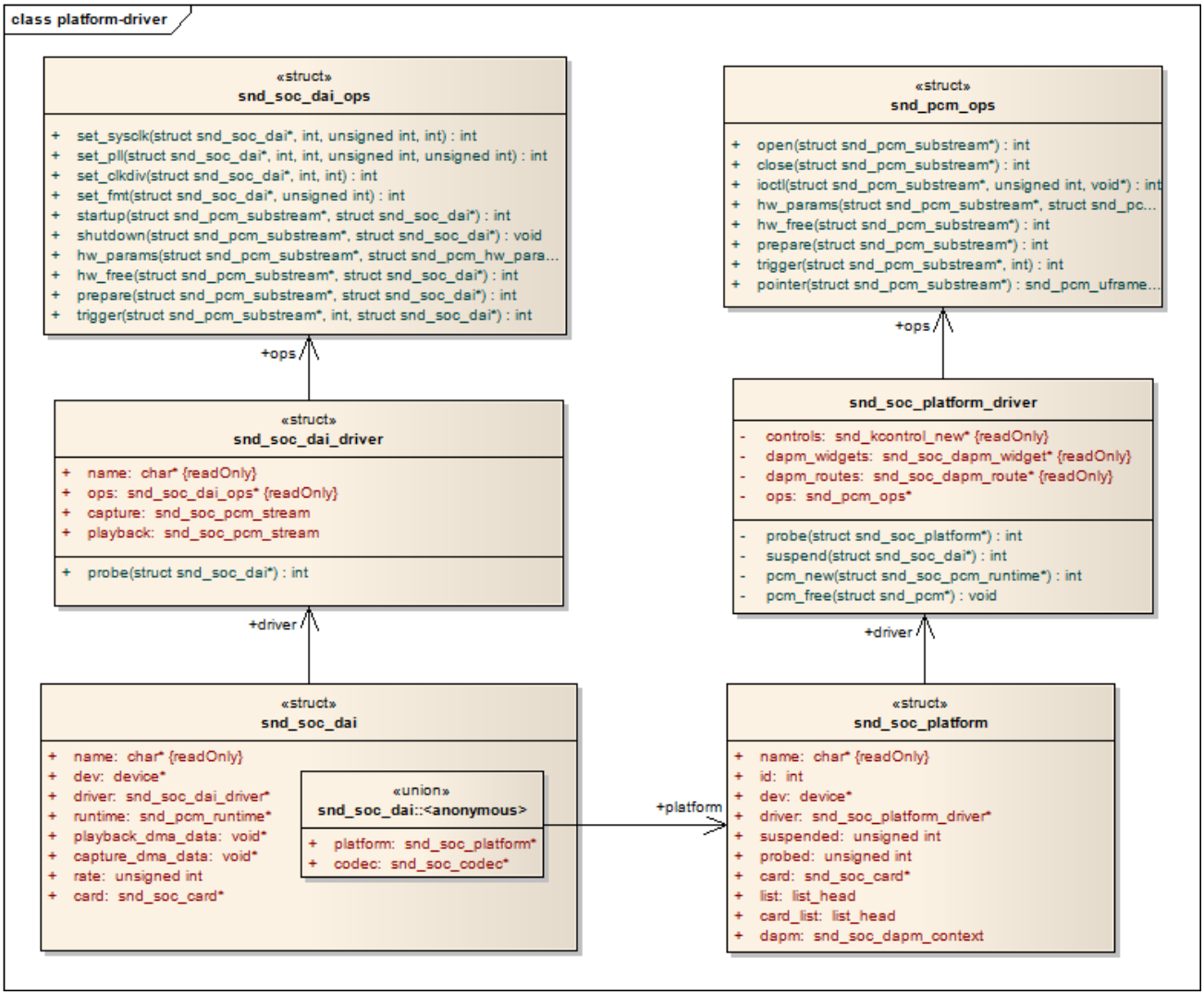


图 7.1　ASoC Platform 驱动

一堆的 private_data，很重要但也很容易搞混，下面的图不知对大家有没有帮助：

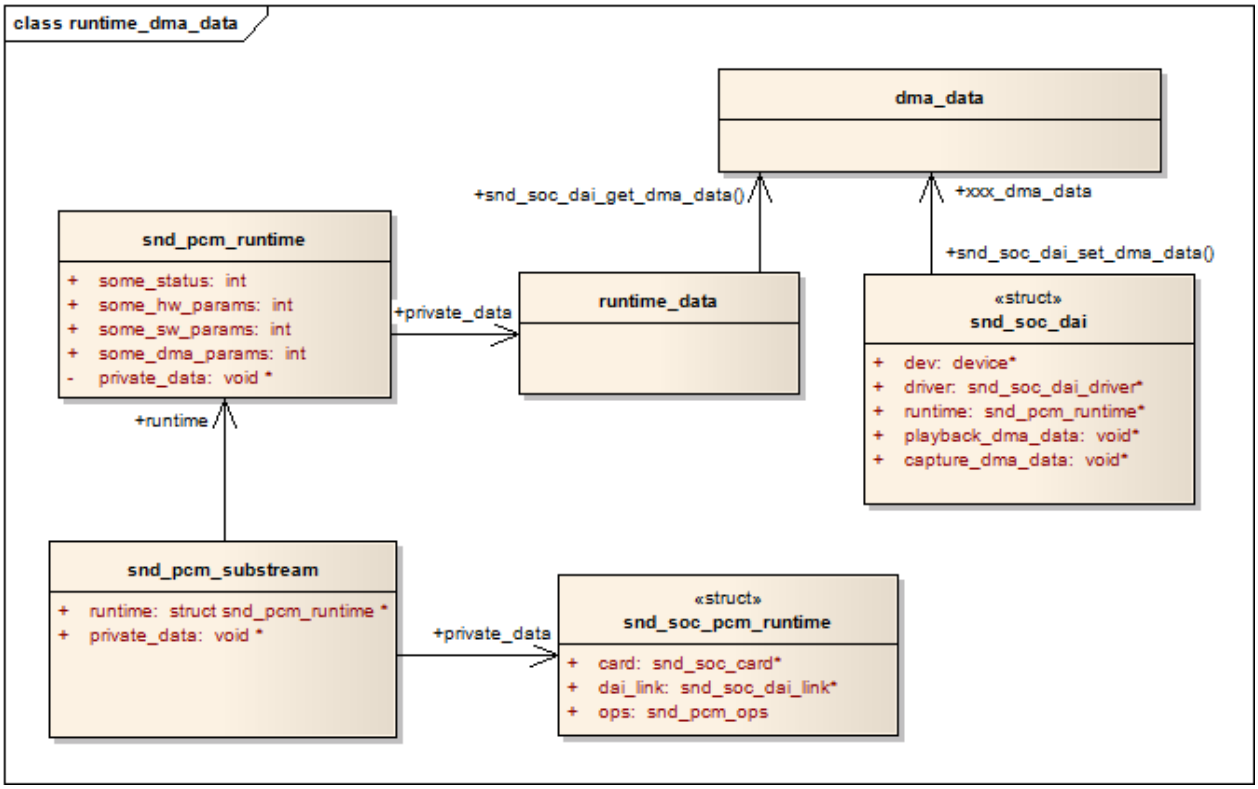


图 7.2　private_data