编写一个 ALSA 驱动

Takashi Iwai

编写一个 ALSA 驱动

(by Takashi Iwai)

0.3.6 版本

翻译：creator sz111@126.com

翻译这篇文章主要是为了学习 ALSA 驱动，因为感觉 ALSA 是 Linux 音频发展方向，所以下决心仔细看看，但是中文资料太少，就想翻译一份奉献给广大初学并且英文不好的朋友。不过自己的英文也非常不好，我也在努力学习中。

翻译的不好，有些地方也不准确，希望大家多提宝贵意见，共同维护这篇文档。

这篇文档主要描述如何写一个 ALSA（Linux 高级声音体系 ）驱动。

目录

析构器
头文件
3.管理声卡和部件
声卡实例
部件
chip 相关数据
1.通过 snd_card_new()分配
2.分配其他设备
注册和释放
4.PCI 资源管理
代码示例
一些必须做的事情
资源分配
设备结构体的注册
PCI 入口
5.PCM 接口
概述
代码示例
构造器
析构器
PCM 相关的运行时数据
硬件描述
PCM 配置
DMA 缓冲区信息
运行状态
私有数据
中断的回调函数
操作函数（回调函数）
open
close
ioctl
hw_params
hw_free
prepare
trigger
pointer
copy,silence
ack
page
中断向量
周期中断
高频率的时钟中断
调用 snd_pcm_period_elapsed()
原子
约束
6.控制接口

前言

这篇文章主要介绍如何写一个 ALSA（Advanced Linux Sound Architecture）(http://www.alsa-project.org)驱动.这篇文档主要针对 PCI 声卡。假如是其他类似的设备，API 函数可能会是不同的。尽管如此，至少 ALSA 内核的 API 是一样的。因此，这篇文章对于写其他类型的设备驱动同样有帮助。

本文的读者需要有足够的 C 语言的知识和基本的 linux 内核编程知识。本文不会去解释一些 Linux 内核编码的基本话题，也不会去详细介绍底层驱动的实现。它仅仅描述如何写一个基于 ALSA 的 PCI 声卡驱动。

假如你对 0.5.x 版本以前的 ALSA 驱动非常熟悉的话，你可以检查驱动文件，例如 es1938.c 或 maestro3.c，那些是在 0.5.x 版本基础上而来的，你可以对比一下他们的差别。

这篇文档仍然是一个草稿，非常欢迎大家的反馈和指正。

第一章 文件目录结构

概述

有两种方式可以得到ALSA驱动程序。

一个是通过ALSA的ftp网站上下载，另外一个是2.6（或以后）Linux源码里面。为了保持两者的同步，ALSA驱动程序被分为两个源码树：alsa-kernel和alsa -drvier。前者完全包含在Linux2.6（或以后）内核树里面，这个源码树仅仅是为了保持2.6（或以后）内核的兼容。后者，ALSA驱动，包含了很多更细分的文件，为了在2.2和2.4内核上编译，配置，同时为了适应最新的内核API函数，和一些在开发中或尚在测试的附加功能。当他们那些功能完成并且工作稳定的时候最终会被移入到alsa 内核树中。

ALSA驱动的文件目录描述如下。 alsa-kernel和alsa-drvier几乎含有相同的文件架构，除了“core”目录和alsa-drvier目录树中被称为“acore”。

示例1-1.ALSA文件目录结构

```
sound
    /core
        /oss
        /seq
            /oss
            /instr
    /ioctl32
    /include
    /drivers
        /mpu401
        /opl3
    /i2c
        /l3
    /synth
        /emux
    /pci
        /(cards)
    /isa
        /(cards)
    /arm
    /ppc
    /sparc
    /usb
    /pcmcia/(cards)
    /oss
```

core目录

这个目录包含了中间层，ALSA的核心驱动。那些本地ALSA模块保持在这个目录里。一些子目录包含那些与内核配置相关的不同的模块。

core/oss

关于PCM和mixer的OSS模拟的模块保存在这个目录里面。Raw midi OSS模拟也被包含在ALSA rawmidi 代码中，因为它非常小。音序器代码被保存在core/seq/oss目录里面（如下）。

core/ioctl32

这个目录包含 32bit-ioctl 到 64bit 架构（如 x86-64，ppc64，sparc64）的转换。对于 32bit 和 alpha 的架构，他们是不被编译的。

core/seq

这和它的子目录主要是关于 ALSA 的音序器。它包含了音序器的 core 和一些主要的音序器模块如：snd-seq-midi,snd-seq-virmidi 等等。它们仅仅在内核配置中当 CONFIG_SND_SEQUENCER 被设定的时候才会被编译。

core/seq/oss

包含了 OSS 音序器的模拟的代码。

core/seq/instr

包含了一些音序器工具层的一些模块。

include 目录

这里面放的是 ALSA 驱动程序开放给用户空间，或者被其他不同目录引用的共同头文件。一般来说，私有头文件不应该被放到此文件目录，但是你仍然会发现一些这样的文件，那是历史遗留问题了。

Drivers 目录

这个目录包含了在不同架构的系统中的不同驱动共享的文件部分。它们是硬件无关的。例如：一个空的 pcm 驱动和一系列 MIDI 驱动会放在这个文件夹中。在子目录里面，会放一些不同的组件的代码，他们是根据不同的 bus 和 cpu 架构实现的。

drivers/mpu401

MPU401 和 MPU401-UART 模块被放到这里。

drviers/opl3 和 opl4

OPL3 和 OPL4 FM-synth 相关放到这里。

i2c 目录

这里面包含了 ALSA 的 i2c 组件。

虽然 Linux 有个 i2c 的标准协议层，ALSA 还是拥有它关于一些 card 的专用 i2c 代码，因为一些声卡仅仅需要一些简单的操作，而标准的 i2c 的 API 函数对此显得太过复杂了。

i2c/l3

这是 ARM L3 i2c 驱动的子目录

synth 目录

它包含了 synth（合成器）的中间层模块

到目前为止，仅仅在 synth/emux 目录下面有 Emu8000/Tmu10k1 synth 驱动。

pci 目录

它和它的一些子目录文件负责 PCI 声卡和一些 PCI BUS 的上层 card 模块。

在 pci 目录下面保存着一些简单的驱动文件，而一些比较复杂的，同时包含多个程序文件的驱动会被放置在 pci 目录下面一个单独的子目录里面（如：emu10k1,ice1712)。

isa 目录

它和它的一些子目录文件是处理 ISA 声卡的上层 card 模块。

arm，ppc，和 sparc 目录

这里放置一些和芯片架构相关的一些上层的 card 模块。

usb 目录

这里包含一些 USB-AUDIO 驱动。在最新版本里面，已经把 USB MIDI 驱动也集成进 USB-AUDIO 驱动了。

pcmcia 目录

PCMCIA 卡，特别是 PCCard 驱动会放到这里。CardBus 驱动将会放到 pci 目录里面，因为 API 函数和标准 PCI 卡上统一的。

oss 目录

OSS/Lite 源文件将被放置在 linux2.6（或以后）版本的源码树中。（在 ALSA 驱动中，它是空的:）

第二章 PCI 驱动的基本流程

概要

PCI 声卡的最简单的流程如下：

- 定义一个 PCI Id 表（参考 PCI Entries 章节）
- 建立 probe()回调函数
- 建立 remove()回调函数
- 建立包含上面三个函数入口的 pci_driver 表
- 建立 init()函数，调用 pci_register_driver()注册上面的 pci_drvier 表
- 建立 exit()函数，调用 pci_unregister_driver()函数

代码示例

代码如下所示。一些部分目前还没有实现，将会在后续章节逐步实现。如：在 snd_mychip_probe()函数的注释部分的号码。

Example2-1.PCI 驱动示例的基本流程

```
#include <sound/driver.h>
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>
/*模块参数（参考“Module Parameters”）*/
static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
static int enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;

/*定义和 chip 相关的记录*/
struct mychip{
        struct snd_card *card;
        //剩余的其他实现将放到这里
        //”PCI Resource Management”
};
/*chip-specific destrcutor
*(see “PCI Resource Management”
*/
static int snd_mychip_free(struct mychip *chip)
{
     ...//后续会实现
}

/*component-destructor
 *(see “Management of Cards and Components”
 */
static int snd_mychip_dev_free(struct snd_device *device)
{
    return snd_mychip_free(device->device_data);
}
```

```
/*chip-specific constructor
 *(see “Management of Cards and Components”)
*/
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
    struct mychip *chip;
    int err;
    static struct snd_device_ops ops = {
           .dev_freee    = snd_mychip_dev_free,
    };
    *rchip = NULL;
    //check PCI avilability here
    //(see "PCI Resource Management")
    /*allocate a chip-specific data with zero filled*/
    chip = kzalloc(sizeof(*chip), GFP_KERNEL);
    if (chip == NULL)
        return -ENOMEM;

    chip->card = card;

    //rest of initialization here;will be implemented
    //later,see "PCI Resource Management"
    ....

    if ((err = snd_device_new(card, SNDRV_DEV_LOWLEVEL,
                              chip, &ops)) < 0) {
        snd_mychip_free(chip);
        return err;
    }

    snd_card_set_dev(card, &pci->dev);
    *rchip = chip;
    return 0;
}

/*constructor --see "Constructor"sub - section*/
static int __devinit snd_mychip_probe(struct pci_dev *pci,
                              const struct pci_device_id *pci_id)
{
     static int dev;
     struct snd_card *card;
     struct mychip *chip;
     int err;
     /*(1)*/
```

```
        if (dev >= SNDRV_CARDS)
            return -ENODEV;
        if (!enable[dev]) {
            dev++;
            return -ENOENT;
        }
        /*(2)*/
        card = snd_card_new(index[dev], id[dev], THIS_MODULE, 0);
        if (card == NULL)
         return -ENOMEM;

      /*(3)*/
      if ((err = snd_mychip_create(card, pci, &chip)) < 0){
          snd_card_free(card);
          return err;
      }
      /*(4)*/
      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);

      /*(5)*/
      ....//implemented later

      /*(6)*/
      if ((err = snd_card_register(card)) < 0) {
          snd_card_free(card);
          return err;
      }

      /*(7)*/
      pci_set_drvdata(pci, card);
      dev++;
      return 0;
    }

    /*destructor --seee"Destructor" sub-section*/
    static void __devexit snd_mychip_remove(struct pci_dev *pci)
    {
         snd_card_free(pci_get_drvdata(pci);
         pci_set_drvdata(pci,NULL);
    }
```

构造函数

PCI 驱动的真正构造函数是回调函数 probe。probe 函数和其他一些被 probe 调用组件构

造函数，那些组件构造函数被放入了__devinit 前缀。你不能放__init 前缀，因为 PCI 设备是热拔插设备。

在 probe 函数中，下面几个是比较常用的。

1）检查并增加设备索引（device index）。

```
static int dev;
...
if (dev >= SNDDRV_CARDS)
     return -ENODEV;
if (!enable[dev]){
     dev++;
     return -ENOENT;
}
```

enable[dev]是一个 module 选项。

每当 probe 被调用的时候，都要检查 device 是否可用。假如不可用，简单的增加设备索引并返回。dev 也会被相应增加一。（step 7）。

2）创建一个 card 实例

```
struct snd_card *card;
....
card = snd_card_new(index[dev], id[dev], THIS_MODULE, 0);
```

这个函数详细展开在“管理 card 和组件”一节。

3）建立一个组件

在这个部分，需要分配 PCI 资源。

```
struct mychip *chip;
....
if ((err = snd_mychip_create(card, pci, &chip)) < 0){
    snd_card_free(card);
    return err;
}
```

这个函数详细展开在“PCI 资源管理”一节。

4）设定驱动 ID 和名字。

```
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);
```

驱动的一些结构变量保存 chip 的 ID 字串。它会在 alsa-lib 配置的时候使用，所以要保证他的唯一和简单。甚至一些相同的驱动可以拥有不同的 ID,可以区别每种 chip 类型的各种功能。

shortname 域是一个更详细的名字。longname 域将会在/proc/asound/cards 中显示。

5）创建其他类似于 mixex，MIDI 的组件。

你已经定义了一些基本组件如 PCM，mixer（e.g.AC97),MIDI(e.g.MPU-401)，还有

其他的接口。同时，假如你想要一些 proc 文件，也可以在这里定义。

6）注册 card 实例

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

这个函数详细展开在“管理 card 和组件”一节。

7）设定 PCI 驱动数据，然后返回零

```
pci_set_drvdata(pci,card);
dev++;
return 0;
```

如上，card 记录将会被保存。在 remove 回调函数和 power-management 回调函数中会被用到。

析构函数

remove 回调函数会释放 card 实例。ALSA 中间层也会自动释放所有附在这个 card 上的组件。

典型应用如下：

```
static void __devexit snd_mychip_remove(struct pci_dev *pci)
{
   snd_card_free(pci_get_drvdata(pci));
   pci_set_drvdata(pci,NULL);
}
```

上面代码是假定 card 指针式放到 PCI driver data 里面的。

头文件

对于以上的代码，至少需要下面几个头文件。

```
#include <sound/driver.h>
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>
```

最后一个是仅仅当源文件中定义了 module 选项的时候才需要。加入代码被分成多个文件，那些不需要 module 选项的文件就不需要。

除此之外，加入你需要中断处理，你需要<linux/interrupt.h>,需要 i/o 接口就要加入<asm/io.h>,如果需要 mdelay(),udelay()函数就需要加入<linux/delay.h>等等。

类似于 PCM 或控制 API 的 ALSA 接口会放到类似于<sound/xxx.h>的头文件当中。它们必须被包含并且要放到<sound/core.h>的后面。

第三章　管理 card 和组件

card 实例

对于每个声卡，都要分配一个 card 记录。

一个 card 记录相当于一个声卡的总部。它管理着声卡中的所有设备（组件），例如 PCM,mixers，MIDI,音序器等等。同时，card 记录还保持着卡的 ID 和 name 字符串，管理 proc 文件，控制电源管理状态和热拔插。Card 的组件列表用来在合适的时候释放资源。

如上，为了创建一个 card 实例，需要调用 snd_card_new().

```
Struct snd_card *card;
card = snd_card_new(index, id, module, extra_size);
```

这个函数需要 4 个参数，card-index 号，id 字符串，module 指针（通常是 THIS_MODULE),extra-data 空间的大小。最后一个参数是用来分配 chip-specific 数据 card->private_data 的。注意这个数据是通过 snd_card_new()进行分配。

组件

card 被创建之后，你可以把组件（设备）附加到 card 实例上面。在 ALSA 驱动 程序中，一个组件用一个 snd_devie 结构体表示。可以是 PCM，控制接口或 raw MIDI 接口等等。每个组件都有个入口函数。

通过 snd_device_new()函数来创建组件。

```
snd_device_new(card, SNDRV_DEV_XXX, chip, &ops);
```

它需要 card 指针，device_level（SNDRV_DEV_XXX),数据指针和回调函数 (&ops)。device-level 定义了组件类型和注册和卸载的顺序。对于大部分的组件来说，device_level 已经定义好了。对于用户自定义的组件，你可以用 SNDRV_DEV_LOWLEVEL.

这个函数自己并不分配数据空间。数据必须提前分配，同时把分配好的数据指针传递给这个函数作为参数。这个指针可以作为设备实例的标识符（上述代码的 chip）。

每个 ALSA 预定义组件，如 ac97 或 pcm 都是通过在各自的构造函数中调用 snd_device_new().在一些回调函数中定义了每个组件的析构函数。因此，你不需要关心这种组件的析构函数的调用。

假如你要创建一个自己的组件，你需要在 dev_free 中的回调函数 ops 中设定析构函数。以便它可以通过 snd_card_free()自动被释放。下面的例子就会显示 chip-specific 数据的实现。

Chip-Specific Data

chip-specific 信息，如：i/o 口，资源，或者中断号就会保持在 chip-specific 记录里面。

```
Struct mychip{
    ....
};
```

通常来说，有两种方式来分配 chip 记录。

1.通过 snd_card_new()分配。

如上面所述，你可以通过传递 extra-data-length 到 snd_card_new()的第四个参数。

```
card = snd_card_new(index[dev], id[dev], THIS_MODULE, sizeof(struct mychip));
```

无论 struct mychip 是否是 chip 记录类型。

分配之后，可以通过如下方式引用：

```
struct mychip *chip = char->private_data;
```

通过这种方法，你不必分配两次。这个记录将会和 card 实例一起被释放。

2.分配一个 extra device。

当你通过 snd_card_new()(第四个参数要设定为 NULL)分配之后，通过调用 kzalloc().

```
Struct snd_card *card;
struct mychip *chip;
card = snd_card_new(index[dev], id[dev], THIS_MODULE, NULL);
....
chip = kzalloc(sizeof(*chip), GFP_KERNEL);
```

chip 记录应该至少拥有一个 snd_card 指针的成员变量。

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

然后，设定 chip 的 card 为 snd_card_new 返回的 card 指针。

```
    chip->card = card;
```

下一步，初始化各个成员变量，使用一个含有特殊的 ops 的 low-level 设备注册 chip 记录。

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

snd_mychip_dev_free()是作为设备的析构函数会调用真正的析构函数调用。

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

snd_mychip_free 是真正的设备析构函数。

注册和释放

当所有的组件都被分配之后，就可以通过 snd_card_register()来注册 card 实例了。对于设备文件的操作也会使能。那是因为，在调用 snd_card_register()调用之前，组件是不能被外界调用的。假如注册失败，必须要调用 snd_card_free()来释放 card。

对于释放 card，你可以简单的通过 snd_card_free().如前所述，所有的组件都可以通过它来自动释放。

要特别注意，这些析构函数（包括 snd_mychip_dev_free 和 snd_mychip_free)不能被定义加入__devexit 前缀，因为它们也有可能被构造函数调用（当构造失败的时候调用）。

对于一个运行热拔插的设备，你可以用 snd_card_free_when_closed。 它将推迟析构直到所有的设备都关闭。

代码示例

本节我们会完成一个chip-specific的构造函数，析构函数和PCI entries。先来看代码。

1.Example4-1.PCI资源管理示例

```
struct mychip{
     struct snd_card *card;
     struct pci_dev *pci;
     unsigned long port;
     int irq;
};
static int snd_mychip_free(struct mychip *chip)
{
    /*disable hardware here if any*/
    ....//这篇文档没有实现

   /*release the irq*/
   if (chip->irq >= 0)
       free_irq(chip->irq,chip);
   /*释放io和memory*/
   pci_release_regions(chip->pci);
   /*disable the PCI entry*/
   pci_disable_device(chip->pci);
   /*release the data*/
   kfree(chip);
   return 0;
}

/*chip-specific constructor*/
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
     struct mychip *chip;
     int err;
     static struct snd_device_ops ops ={
         .dev_free = snd_mychip_dev_free,
     };
     *rchip = NULL;
     /*initialize the PCI entry*/
     if ((err = pci_enable_device(pci)) < 0)
         return err;
     /*check PCI availability (28bit DMA)*/
     if (pci_set_dma_mask(pci, DMA_28BIT_MASK) < 0 ||
         pci_set_consistent_dma_mask(pci,DMA_28BIT_MASK) < 0 ){
         printk(KERN_ERR “Error to set 28bit mask DMA\n”);
```

```
        pci_disable_device(pci);
        return -ENXIO;
    }

    chip = kzalloc(sizeof(*chip), GFP_KERNEL);
    if (chip == NULL){
        pci_disable_device(pci);
        return -ENOMEM;
    }
    /*initialize the stuff*/
    chip->card = card;
    chip->pci = pci;
    chip->irq = -1;
    /*(1)PCI 资源分配*/
    if ((err = pci_request_regions(pci, “My Chip”)) < 0){
        kfree(chip);
        pci_disable_device(pci);
        return err;
    }
    chip->port = pci_resource_start(pci,0);
    if (request_irq(pci->irq, snd_mychip_interrupt,
                    IRQF_SHARED, “My Chip”,chip){
         printk(KERN_ERR “Cannot grab irq %d\n”,pci->irq);
         snd_mychip_free(chip);
         return -EBUSY;
    }
    chip->irq = pci->irq;
    /*(2)chip hardware的初始化*/
    ....//本文未实现

    if ((err = snd_device_new(card, SNDRV_DEV_LOWLEVEL,
                              chip, &ops)) < 0){
        snd_mychip_free(chip);
        return err;
    }
    snd_card_set_dev(card,&pci->dev);
    *rchip = chip;
    return 0;
}

/*PCI Ids*/
static struct pci_device_id snd_mychip_ids[] = {
    {PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
     PCI_ANY_ID, PCI_ANY_ID,0,0,0,},
    ....
    {0,}
```

```
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);

/*pci_driver定义*/
static struct pci_drvier driver={
    .name = “My Own Chip”,
    .id_table = snd_mychip_ids,
    .probe = snd_mychip_probe,
    .remove = __devexit_p(snd_mychip_remove),
};
/*module的初始化*/
static int __init alsa_card_mychip_init(void)
{
    return pci_register_driver(&driver);
}
/*clean up the module*/
static void __exit alsa_card_mychip_exit(void)
{
     pci_unregister_drvier(&drvier);
 }
 module_init(alsa_card_mychip_init);
 module_exit(alsa_card_mychip_eixt);

 EXPORT_NO_SYMBOLS /*为了和老版本的内核兼容*/
```

一些必须做的事情

一般在probe()函数中分配PCI资源，通常采用一个xxx_create()函数来完成上述功能。

在PCI设备驱动中，在分配资源之前先要调用pci_enable_device().同样，你也要设定合适的PCI DMA mask限制i/o接口的范围。在一些情况下，你也需要设定pci_set_master().

资源分配

利用标准的内核函数来分配I/O和中断。不像ALSA ver0.5.x那样没有什么帮助。同时那些资源必须在析构函数中被释放（如下）。在ALSA 0.9.x，你不需要像0.5.x那样还要为PCI分配DMA。

现在假定PCI设备拥有8直接的I/O口和中断，mychip的结构体可以如下所示：

```
struct mychip{
    struct snd_card *card;
    unsigned long port;
    int irq;
}
```

对于一个I/O端口（或者也有内存区域），你需要得到可以进行标准的资源管理的资源的指针。对于中断，你必须保存它的中断号。但是你必须实际分配之前先把初始化中断号为-1，因为中断号为0也是被允许的。端口地址和它的资源指针可以通过kzalloc()来分配初始化为null，所以你不用非要重新设定他们。

I/O端口的分配可以采用如下方式：

```
if ((err = pci_request_region(pci, “My Chip”)) < 0){
    kfree(chip);
    pci_disable_device(pci);
    return err;
}
chip->port = pci_resource_start(pci, 0);
```

它将会保留给定PCI设备的8字节的I/O端口区域。返回值chip->res_port在request_region函数中通过kmalloc分配。这个分配的指针必须通过kfree()函数进行释放，但是这个部分有些问题，具体将会在下面详细分析。

分配一个中断源如下所示：

```
if (request_irq(pci->irq, snd_mychip_interrupt,
                IRQF_DISABLED | IRQF_SHARED, “My Chip”,chip)){
    printk(KERN_ERR “cannot grab irq %d\n”, pci->irq);
    snd_mychip_free(chip);
    return -EBUSY;
}
chip->irq = pci->irq;
```

snd_mychip_interrupt()就是下面定义的中断处理函数。注意request_irq()成功的时候，返回值要保持在chip->irq里面。

在PCI bus上，中断是共享的。因此，申请中断的函数request_irq要加入IRQF_SHARED标志位。

request_irq的最后一个参数是被传递给中断处理函数的数据指针。通常来说，chip-specific记录会用到它，你也可以按你喜欢的方式用它。

我不想在这时候详细解释中断向量函数，但是至少这里出现的会解释一下。中断向量如下所示：

```
static irqreturn_t snd_mychip_interrup(int irq, void *dev_id)
{
    struct mychip *chip = dev_id;
    return IRQ_HANDLED;
}
```

现在，让我们为上述的资源写一个相应的析构函数。析构函数是非常简单的：关闭硬件（假如它被激活）同时释放它的资源。到目前为止，我们没有硬件，所以关闭硬件的部分就不写了。

为了释放资源，“check-and-release”（检查然后释放）的方式是比较安全的。对于中断，采用如下的方式：

```
if (chip->irq >= 0)
     free_irq(chip->irq,chip);
```

因为irq号是从0开始的，所以你必须初始化chip->irq为一个负数（例如-1），所以你可以按上面的方式检查irq的有效性。

通过pci_request_region()或pci_request_regions()申请I/O端口和内存空间,相应的，通过pci_release_region()或pci_release_regions来释放。

```
pci_release_regions(chip->pci);
```

通常可以通过request_region()或request_mem_region()来申请，也可以通过release_region()来释放。假定你把request_region返回的resource pointer保存在chip->res_port,释放的程序如下所示：

```
release_and_free_resource(chip->res_port);
```

在所有都结束的时候不要忘记了调用pci_disable_device().最后释放chip-specific记录。

```
kfree(chip);
```

再提醒一下，不能在析构函数前面放__devexit。

我们在上面没有实现关闭硬件的功能部分。假如你需要做这些，请注意析构函数可能会在chip的初始化完成之前被调用。最好设定一个标志位确定是否硬件已经初始化，来决定是否略过这部分。

如果chip-data放置在在含有SNDRV_DEV_LOWLEVEL标志的snd_device_new()函数申请的设备中，它的析构函数将会最后被调用。那是因为，它要确认像PCM和一些控制组件已经被释放。你不必显式调用停止PCM的函数，只要在low-level中停止这些硬件。

mamory-mapped的区域的管理和i/o端口管理一样。需要如下3个结构变量：

```
struct mychip{
    ....
    unsigned long iobase_phys;
    void __iomem *iobase_virt;
};
```

通过如下方式来申请：

```
if ((err = pci_request_regions(pci, “My Chip”)) < 0){
    kfree(chip);
    return err;
}
chip->iobase_phys = pci_resource_start(pci, 0);
chip->iobase_virt = ioremap_nocache(chip->iobase_phys,
                                    pci_resource_len(pci, 0));
```

对应的析构函数如下：

```
static int snd_mychip_free(struct mychip *chip)
{
      ....
      if (chip->iobase_virt)
           iounmap(chip->iobase_virt);
      ....
      pci_release_regions(chip->pci);
      ....
}
```

注册设备结构体

在一些地方，典型的是在调用snd_device_new()，假如你想通过udev或者ALSA提供的一些老版本内核的兼容的宏来控制设备，你需要注册chip的设备结构体。很简单如下所示：

```
snd_card_set_dev(card,&pci->dev);
```

所以它保存了 card 的 PCI 设备指针。它将会在后续设备注册的时候被 ALSA 内核功能调用。

假如是非 PCI 设备，就需要传递一个合适的设备结构指针。（假如是不可以热拔插的 ISA 设备，你就不需要这么做了。）

PCI Entries

到目前为止已经做得非常好了，下面让我们完成 PCI 的剩余的一些工作。首先，我们需要一个这个芯片组的 pci_device_id 表。它是一个含有 PCI 制造商和设备 ID 的表，还有一些 mask。

例如：

```
static struct pci_device_id snd_mychip_ids[] = {
    {PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
     PCI_ANY_ID, PCI_ANY_ID,0,0,0,},
    ....
    {0,}
};
MODULE_DEVICE_TABLE(pci,snd_mychip_ids);
```

pci_device_id 结构体的第一个和第二个结构变量是制造商和设备的 ID。假如你没有关于设备的特别选择，你可以采用上述的。pci_device_id 结构体的最后一个结构变量是一个私有变量。你可以放一些可以区别其他的值，如：可以区分每个设备 ID 的不同操作类型。这些例子出现在 intel 的驱动里面。最后一个 table 元素是代表结束符。必须把所有成员设定为 0.

然后，我们来准备 pci_driver 记录：

```
static struct pci_drvier driver = {
    .name = “My Own Chip”,
    .id_table = snd_mychip_ids,
    .probe = snd_mychip_probe,
    .remove = __devexit_p(snd_mychip_remove),
};
```

probe 和 remove 函数在以前的章节已经介绍过。remove 应该用一个__devexit_p()宏来定义。所以，它不是为那些固定的或不支持热拔插的设备定义的。name 结构变量是标识设备的名字。注意你不能用“/”字符。

最后，module 入口如下：

```
static int __init alsa_card_mychip_init(void)
{
    return pci_register_driver(&driver);
}
static void __exit alsa_card_mychip_exit(void)
{
    pci_unregister_driver(&driver);
}
module_init(alsa_card_mychip_init);
module_exit(alsa_card_mychip_exit);
```

注意 module 入口函数被标识为__init 和__exit 前缀，而不是__devinit 或者 __devexit.

哦，忘记了一件事，假如你没有 export 标号，如果是 2.2 或 2.4 内核你必须显式声明 r 如下：（当然，2.6 内核已经不需要了。）

EXPORT_NO_SYMBOLS;

就这些了。

第五章　PCM接口

概述

PCM中间层是ALSA中作用非常大的。它是唯一需要在每个驱动中都需要实现的low-level的硬件接口。

为了访问PCM层，你需要包含<sound/pcm.h>。除此之外，如果你要操作hw_param相关的函数，还需要包含<sound/pcm_param.h>。

每个card设备可以最多拥有4个pcm实例。一个pcm实例对应予一个pcm设备文件。组件的号码限制主要是和Linux的可用的设备号多少有关。假如允许64bit的设备号，我们可以拥有更多的pcm实例。

一个pcm实例包含pcm 放音和录音流，而每个pcm流由一个或多个pcm子流组成。一些声卡支持多重播放的功能。例如：emu10k1就包含一个32个立体声子流的PCM放音设备。事实上，每次被打开的时候，一个可用的子流会自动的被选中和打开。同时，当一个子流已经存在，并且已经被打开，当再次被打开的时候，会被阻塞，或者根据打开文件的模式不同返回一个EAGAIND的错误信息。你也不需要知道你的驱动细节部分，PCM中间层会处理那些工作。

代码示例

下面的代码没有包含硬件接口程序，主要显示一个如何构建一个PCM接口的骨架。

Example5-1.PCM示例代码

```
#include <sound/pcm.h>
....
/*硬件定义*/
static struct snd_pcm_hardware snd_mychip_playback_hw = {
    .info = (SNDRV_PCM_INFO_MMAP |
             SNDRV_PCM_INFO_INTERLEAVED |
             SNDRV_PCM_INFO_BLOCK_TRANSFER |
             SNDRV_PCM_INFO_MMAP_VALID),
    .formats = SNDRV_PCM_FORMAT_S16_LE,
    .rates = SNDRV_PCM_RATE_8000_48000,
    .rate_min = 8000,
    .rate_max = 48000,
    .channels_min = 2,
    .channels_max = 2,
    .buffer_bytes_max = 32768,
    .period_bytes_min = 4096,
    .period_bytes_max = 32768,
    .periods_min = 1,
    .periods_max = 1024,
};
/*硬件定义*/
static struct snd_pcm_hardware snd_mychip_capture_hw = {
    .info = (SNDRV_PCM_INFO_MMAP |
             SNDRV_PCM_INFO_INTERLEAVED |
             SNDRV_PCM_INFO_BLOCK_TRANSFER |
             SNDRV_PCM_INFO_MMAP_VALID),
    .formats = SNDRV_PCM_FORMAT_S16_LE,
```

```
    .rates = SNDRV_PCM_RATE_8000_48000,
    .rate_min = 8000,
    .rate_max = 48000,
    .channels_min = 2,
    .channels_max = 2,
    .buffer_bytes_max = 32768,
    .period_bytes_min = 4096,
    .period_bytes_max = 32768,
    .periods_min = 1,
    .periods_max = 1024,
};

/*播放 open 函数*/
static int snd_mychip_playback_open(struct snd_pcm_substream *substream)
{
    struct mychip *chip = snd_pcm_substream_chip(substream);
    struct snd_pcm_runtime *runtime = substream->runtime;

    runtime->hw = snd_mychip_playback_hw;
    /*其他硬件初始化的部分在这里完成*/
    return 0;
}
/*播放 close 函数*/
static int snd_mychip_playback_close(struct snd_pcm_substream *substream)
{
     struct mychip *chip = snd_pcm_substream_chip(substream);
     //硬件相关代码写在这
     return 0;
 }
 /*录音 open 函数*/
 static int snd_mychip_capture_open(struct snd_pcm_substream *substream)
 {
     struct mychip *chip = snd_pcm_substream_chip(substream);
     struct snd_pcm_runtime *runtime = substream->runtime;

     runtime->hw = snd_mychip_capture_hw;
     /*其他硬件初始化的部分在这里完成*/
     return 0;
 }
 /*录音 close 函数*/
 static int snd_mychip_capture_close(struct snd_pcm_substream *substream)
 {
     struct mychip *chip = snd_pcm_substream_chip(substream);
     //硬件相关代码写在这
     return 0;
 }
```

```
/*hw_params 函数*/
static int snd_mychip_pcm_hw_params(struct snd_pcm_substream *substream,
                                    struct snd_pcm_hw_params *hw_params)
{
    return snd_pcm_lib_malloc_pages(substream,
                                    params_buffer_bytes(hw_params));
}
/*hw_free 函数*/
static int snd_mychip_pcm_hw_free(struct snd_pcm_substream *substream)
{
    return snd_pcm_lib_free_pages(substream);
}
/*prepare 函数*/
static int snd_mychip_pcm_prepare(struct snd_pcm_substream *substream)
{
    struct mychip *chip = snd_pcm_substream_chip(substream);
    struct snd_pcm_runtime *runtime = substream->runtime;
    /*在此做设定一些硬件配置
     *例如....
     */
    mychip_set_sample_format(chip, runtime->format);
    mychip_set_sample_rate(chip, runtime->rate);
    mychip_set_channels(chip, runtime->channels);
    mychip_set_dma_setup(chip, runtime->dma_addr,
                         chip->buffer_size,
                         chip->period_size);
    return 0;
}
/*trigger 函数*/
static int snd_mychip_pcm_trigger(struct snd_pcm_substream *substream,
                                  int cmd)
{
    switch(cmd){
    case SNDRV_PCM_TRIGGER_START:
        //启动一个 PCM 引擎
        break;
    case SNDRV_PCM_TRIGGER_STOP:
        //停止一个 PCM 引擎
        break;
    default:
        return -EINVAL;
    }
}
/*pointer 函数*/
static snd_pcm_uframes_t
```

```
snd_mychip_pcm_pointer(struct snd_pcm_substream *substream)
{
    struct mychip *chip = snd_pcm_substream_chip(substream)
    unsigned int current_ptr;
    /*得到当前缓冲区的硬件位置*/
    current_ptr = mychip_get_hw_pointer(chip);
    return current_ptr;
}
/*操作函数*/
static struct snd_pcm_ops snd_mychip_playback_ops = {
    .open = snd_mychip_playback_open,
    .close = snd_mychip_playback_close,
    .ioctl = snd_pcm_lib_ioctl,
    .hw_params = snd_mychip_pcm_hw_params,
    .hw_free = snd_mychip_pcm_hw_free,
    .prepare = snd_mychip_pcm_prepare,
    .trigger = snd_mychip_pcm_trigger,
    .pointer = snd_mychip_pcm_pointer,
};
/*操作函数*/
static struct snd_pcm_ops snd_mychip_capture_ops = {
    .open = snd_mychip_capture_open,
    .close = snd_mychip_capture_close,
    .ioctl = snd_pcm_lib_ioctl,
    .hw_params = snd_mychip_pcm_hw_params,
    .hw_free = snd_mychip_pcm_hw_free,
    .prepare = snd_mychip_pcm_prepare,
    .trigger = snd_mychip_pcm_trigger,
    .pointer = snd_mychip_pcm_pointer,
};

/*关于录音的定义在这里省略....*/

/*创建一个pcm设备*/
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
    struct snd_pcm *pcm;
    int err;
    if ((err = snd_pcm_new(chip->card, “My Chip”, 0 , 1, 1,&pcm) < 0)
        return err;
    pcm->private_data = chip;
    strcpy(pcm->name,”My Chip”);
    chip->pcm = pcm;
    /*设定操作函数*/
    snd_pcm_set_ops(pcm,SNDRV_PCM_STREAM_PLAYBACK,
                    &snd_mychip_playback_ops);
```

```
        snd_pcm_set_ops(pcm,SNDRV_PCM_STREAM_CAPTURE,
                        &snd_mychip_capture_ops);
        /*预分配缓冲区
         *注意：可能会失败
         */
        snd_pcm_lib_preallocate_pages_for_all(pcm,SNDRV_DMA_TYPE_DEV,
                                              snd_dma_pci_data(chip->pci),
                                              64*1024,64*1024);
        return 0;
    }
```

构造函数

通过 snd_pcm_new()来分配一个 pcm 实例。更好的方式是为 pcm 创建一个构造函数。

```
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
    struct snd_pcm *pcm;
    int err;
    if ((err = snd_pcm_new(chip->card, “My Chip”, 0 , 1, 1,&pcm) < 0)
        return err;
    pcm->private_data = chip;
    strcpy(pcm->name,” My Chip”);
    chip->pcm = pcm;
    ....
    return 0;
}
```

snd_pcm_new()需要 4 个参数。第一个是 card 指针，第二个是标识符字符串，第三个是 PCM 设备索引。假如你创建了超过一个 pcm 实例，通过设备索引参数来区分 pcm 设备。例如：索引为 1 代表是第二个 PCM 设备。

第四个和第五个参数是表示播放和录音的子流的数目。上面的示例都是为 1。当没有播放或录音可以用的时候，可以设定对应的参数为 0。

假如声卡支持多个播放和录音子流，你可以分配更多的号码，但是它们必须可以在 open()和 close()函数中被正确处理。你可以通过 snd_pcm_substream 的成员变量 number 来知道那用到的是那个子流。如：

```
struct snd_pcm_substream *substream;
int index = substream->number;
```

pcm 创建之后，必须设定 pcm 流的操作函数。

```
snd_pcm_set_ops(pcm,SNDRV_PCM_STREAM_PLAYBACK,
                &snd_mychip_playback_ops);

snd_pcm_set_ops(pcm,SNDRV_PCM_STREAM_CAPTURE,
                &snd_mychip_capture_ops);
```

操作函数类似如下：

```
/*操作函数*/
```

```
static struct snd_pcm_ops snd_mychip_playback_ops = {
    .open = snd_mychip_playback_open,
    .close = snd_mychip_playback_close,
    .ioctl = snd_pcm_lib_ioctl,
    .hw_params = snd_mychip_pcm_hw_params,
    .hw_free = snd_mychip_pcm_hw_free,
    .prepare = snd_mychip_pcm_prepare,
    .trigger = snd_mychip_pcm_trigger,
    .pointer = snd_mychip_pcm_pointer,
};
```

每一个回调函数都会在“操作函数”一节中详细介绍。

设定完操作函数之后，大部分情况要预分配缓冲区。一般情况采用如下方式分配：

```
snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                      snd_dma_pci_data(chip->pci),
                                      64*1024,64*1024);
```

默认它会分配64KB的缓冲。关于缓冲区的管理会在“缓冲区和内存管理”一章详细描述。

除此之外，你还可以为这个pcm设定一些附加的信息保存在pcm->info_flags变量中。一些可用的变量定义例如SNDRV_PCM_INFO_XXX都在<sound/asound.h>中，这些主要是为了硬件定义的（稍后会详细描述）。假如你的声卡仅仅支持半双工，你要指定如下：

```
pcm->info_flags = SNDDRV_PCM_INFO_HALF_DUPLEX;
```

到了析构部分？

一个pcm实例不是总是要求析构的。既然pcm中间层会自动的把pcm设备是否，你就不用特别的在调用析构函数了。

析构函数在一种情况下是必须的，就是你在内部创建了一个特殊的的记录，需要自己去释放他们。这种情况下，你可以把pcm->private_free指向你的析构函数。

Example5-2.含有一个析构函数的PCM实例

```
static void mychip_pcm_free(struct snd_pcm *pcm)
{
    struct mychip *chip = snd_pcm_chip(pcm);
    /*释放你自己的数据*/
    kfree(chip->my_private_pcm_data);
    //做其他事情
}
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
    struct snd_pcm *pcm;
    /*分配你自己的数据*/
    chip->myprivate_pcm_data = kmalloc(..);
    /*设定析构函数*/
    pcm->private_data = chip;
    pcm->private_free = mychip_pcm_free;
    ....
}
```

PCM信息运行时指针

当打开一个一个PCM子流的时候，PCM运行时实例就会分配给这个子流。这个指针可以通过substream->runtime获得。运行时指针拥有多种信息：hw_params和sw_params的配置的拷贝，缓冲区指针，mmap记录，自旋锁等等。几乎你想控制PCM的所有信息都可以在这里得到。

```
Struct _snd_pcm_runtime {
    /*状态*/
    struct snd_pcm_substream *trigger_master;
    snd_timestamp_t trigger_tstamp;/*触发时间戳*/、
    int overrange;
    snd_pcm_uframes_t avail_max;
    snd_pcm_uframes_t hw_ptr_base /*缓冲区复位时的位置*/
    snd_pcm_uframes_t hw_ptr_interrupt;/*中断时的位置*/
    /*硬件参数*/
    snd_pcm_access_t access; /*存取模式*/
    snd_pcm_format_t format; /*SNDRV_PCM_FORMAT_* */
    snd_pcm_subformat_t subformat; /*子格式*/
    unsigned int rate; /*rate in HZ*/
    unsigned int channels; /*通道*/
    snd_pcm_uframe_t period_size; /*周期大小*/
    unsigned int periods /*周期数*/
    snd_pcm_uframes_t buffer_size; /*缓冲区大小*/
    unsigned int tick_time; /*tick time滴答时间*/
    snd_pcm_uframes_t min_align; /*格式对应的最小对齐*/
    size_t byte_align;
    unsigned int frame_bits;
    unsigned int sample_bits;
    unsigned int info;
    unsigned int rate_num;
    unsigned int rate_den;
    /*软件参数*/
    struct timespec tstamp_mode; /*mmap时间戳被更新*/
    unsigned int sleep_min; /*睡眠的最小节拍数*/
    snd_pcm_uframes_t xfer_align; /*xfer的大小需要是成倍数的*/
    snd_pcm_uframes_t start_threshold;
    snd_pcm_uframes_t stop_threshold;
    snd_pcm_uframes_t silence_threshold;/*silence填充阀值*/
    snd_pcm_uframes_t silence_size; /*silence填充大小*/
    snd_pcm_uframes_t boundary;
    snd_pcm_uframes_t silenced_start;
    snd_pcm_uframes_t silenced_size;

    snd_pcm_sync_id_t sync; /*硬件同步ID*/
    /*mmap*/
    volatile struct snd_pcm_mmap_status *status;
    volatile struct snd_pcm_mmap_control *control;
```

```
    atomic_t mmap_count;
    /*锁/调度*/
    spinlock_t lock;
    wait_queue_head_t sleep;
    struct timer_list tick_timer;
    struct fasync_struct *fasync;
    /*私有段*/
    void *private_data;
    void (*private_free)(struct snd_pcm_runtime *runtime);

    /*硬件描述*/
    struct snd_pcm_hardware hw;
    struct snd_pcm_hw_constraints hw_constraints;
    /*中断的回调函数*/
    void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
    void (*transfer_ack_end)(struct snd_pcm_substream *substream);
    /*定时器*/
    unsigned int timer_resolution; /*timer resolution*/
    /*DMA*/
    unsigned char *dma_area;
    dma_addr_t dma_addr; /*总线物理地址*/
    size_t dma_bytes; /*DMA 区域大小*/
    struct snd_dma_buffer *dma_buffer_p; /*分配的缓冲区*/
    #if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
    struct snd_pcm_oss_runtime oss;
    #endif
};
```

snd_pcm_runtime 对于大部分的驱动程序操作集的函数来说是只读的。仅仅 PCM 中间层可以改变/更新这些信息。但是硬件描述，中断响应，DMA 缓冲区信息和私有数据是例外的。此外，假如你采用标准的内存分配函数 snd_pcm_lib_malloc_pages()，就不再需要自己设定 DMA 缓冲区信息了。

下面几章，会对上面记录的现实进行解释。

硬件描述

硬件描述（struct snd_pcm_hardware)包含了基本硬件配置的定义。如前面所述，你需要在 open 的时候对它们进行定义。注意 runtime 实例拥有这个描述符的拷贝而不是已经存在的描述符的指针。换句话说，在 open 函数中，你可以根据需要修改描述符的拷贝。例如，假如在一些声卡上最大的通道数是 1，你仍然可以使用相同的硬件描述符，同时在后面你可以改变最大通道数。

```
Struct snd_pcm_runtime *runtime = substream->runtime;
....
runtime->hw = snd_mychip_playback_hw; /*通用定义*/
if (chip->model == VERY_OLD_ONE)
    runtime->hw.channels_max = 1;
```

典型的硬件描述如下：

```
static struct snd_pcm_hardware snd_mychip_playback_hw = {
    .info = (SNDRV_PCM_INFO_MMAP |
             SNDRV_PCM_INFO_INTERLEAVED |
             SNDRV_PCM_INFO_BLOCK_TRANSFER |
             SNDRV_PCM_INFO_MMAP_VALID),
    .formats = SNDRV_PCM_FORMAT_S16_LE,
    .rates = SNDRV_PCM_RATE_8000_48000,
    .rate_min = 8000,
    .rate_max = 48000,
    .channels_min = 2,
    .channels_max = 2,
    .buffer_bytes_max = 32768,
    .period_bytes_min = 4096,
    .period_bytes_max = 32768,
    .periods_min = 1,
    .periods_max = 1024,
};
```

info 字段包含 pcm 的类型和能力。位标志在<sound/asound.h>中定义，如：SNDRV_PCM_INFO_XXX。这里，你必须定义 mmap 是否支持和支持哪种 interleaved 格式。当支持 mmap 的时候，应当设定 SNDRV_PCM_INFO_MMAP。当硬件支持 interleaved 或 no-interleaved 格式的时候，要设定 SNDRV_PCM_INFO_INTERLEAVED 或 SNDRV_PCM_INFO_NONINTERLEAVED 标志位。假如两者都支持，你也可以都设定。

如上面的例子，MMAP_VALID 和 BLOCK_TRANSFER 都是针对 OSS mmap 模式，通常情况它们都要设定。当然，MMAP_VALID 仅仅当 mmap 真正被支持的时候才会被设定。

其他一些标志位是 SNDRV_PCM_INFO_PAUSE 和 SNDRV_PCM_INFO_RESUME。SNDRV_PCM_INFO_PAUSE 标志位意思是 pcm 支持“暂停”操作，SNDRV_PCM_INFO_RESUME 表示是 pcm 支持“挂起/恢复”操作。假如 PAUSE 标志位被设定，trigger 函数就必须执行一个对应的（暂停 按下/释放）命令。就算没有 RESUME 标志位，也可以被定义挂起/恢复触发命令。

更详细的部分请参考“电源管理”一章。

当 PCM 子系统能被同步（如：播放流和录音流的开始/结束的同步）的时候，你可以设定 SNDRV_PCM_INFO_SYNC_START 标志位。在这种情况下，你必须在 trigger 函数中检查 PCM 子流链。下面的章节会想笑介绍这个部分。

formats 字段包含了支持格式的标志位(SNDRV_PCM_FMTBIT_XXX)。假如硬件支持超过一个的格式，需要对位标志位进行“或”运算。上面的例子就是支持 16bit 有符号的小端格式。

rates 字段包含了支持的采样率（SNDRV_PCM_RATE_XXX)。当声卡支持多种采样率的时候，应该附加一个 CONTINUOUS 标志。已经预先定义的典型的采样率，假如你的声卡支持不常用的采样率，你需要加入一个 KNOT 标志，同时手动的对硬件进行控制（稍后解释）。

rate_min 和 rate_max 定义了最小和最大的采样率。应该和采样率相对应。

channel_min 和 channel_max 定义了最大和最小的通道，以前可能你已看到。

buffer_bytes_max 定义了以字节为单位的最大的缓冲区大小。这里没有 buffer_bytes_min 字段，因为它可以通过最小的 period 大小和最小的 period 数量计算得出。同时，period_bytes_min 和定义的最小和最大的 period。periods_max 和 periods_min 定义了最大和最小的 periods。

period 信息和 OSS 中的 fragment 相对应。period 定义了 PCM 中断产生的周期。这个周

期非常依赖硬件。一般来说，一个更短的周期会提供更多的中断和更多的控制。如在录音中，周期大小定义了输入延迟，另外，整个缓存区大小也定义了播放的输出延迟。

字段 fifo_size。这个主要是和硬件的 FIFO 大小有关，但是目前驱动中或 alsa-lib 中都没有使用。所以你可以忽略这个字段。

PCM 配置

OK,让我们再次回到 PCM 运行时记录。最经常涉及的运行时实例中的记录就是 PCM 配置了。PCM 可以让应用程序通过 alsa-lib 发送 hw_params 来配置。有很多字段都是从 hw_params 和 sw_params 结构中拷贝过来的。例如：format 保持了应用程序选择的格式类型，这个字段包含了 enum 值 SNDRV_PCM_FORMAT_XXX。

其中要注意的一个就是，配置的 buffer 和 period 大小被放在运行时记录的“frame”中。在 ALSA 里，1frame=channel*samples-size。为了在帧和字节之间转换，你可以用下面的函数，frames_to_bytes()和 bytes_to_frames()。

```
period_bytes = frames_to_bytes(runtime,runtime->period_size);
```

同样，许多的软件参数(sw_params)也存放在 frames 字段里面。请检查这个字段的类型。snd_pcm_uframes_t 是作为表示 frames 的无符号整数，而 snd_pcm_sframes_t 是作为表示 frames 的有符号整数。

DMA 缓冲区信息

DMA 缓冲区通过下面 4 个字段定义，dma_area,dma_addr,dma_bytes,dma_private。其中 dma_area 是缓冲区的指针（逻辑地址）。可以通过 memcopy 来向这个指针来操作数据。dma_addr 是缓冲区的物理地址。这个字段仅仅当缓冲区是线性缓存的时候才要特别说明。dma_bytes 是缓冲区的 大小。dma_private 是被 ALSA 的 DMA 管理用到的。

如果采用 ALSA 的标准内存分配函数 snd_pcm_lib_mallock_pages()分配内存，那些字段会被 ALSA 的中间层设定，你不能自己改变他们，可以读取而不能写入。而如果你想自己分配内存，你就需要在 hw_params 回调里面自己管理它们。当内存被 mmap 之后，你至少要设定 dma_bytes 和 dma_area。但是如果你的驱动不支持 mmap，这些字段就不必一定设定.dma_addr 也不是强制的，你也可以根据灵活来用 dma_private。

运行状态

可以通过 runtime->status 来获得运行状态。它是一个指向 snd_pcm_mmap_status 记录的指针。例如，可以通过 runtime->status->hw_ptr 来得到当前 DMA 硬件指针。

可以通过 runtime->control 来查看 DMA 程序的指针，它是指向 snd_pcm_mmap_control 记录。但是，不推荐直接存取这些数据。

私有数据

可以为子流分配一个记录，让它保存在 runtime->private_data 里面。通常可以在 open 函数中做。不要和 pcm->private_data 混搅了，pcm->private_data 主要是在创建 PCM 的时候指向 chip 实例，而 runtime->private_data 是在 PCM open 的时候指向一个动态数据。

```
Struct int snd_xxx_open(struct snd_pcm_substream *substream)
{
    struct my_pcm_data *data;
    data = kmalloc(sizeof(*data),GFP_KERNEL);
    substream->runtime->private_data = data;
    ....
}
```

上述分配的对象要在close函数中释放。

中断函数

transfer_ack_begin()和transfer_ack_end()将会在snd_pcm_period_elapsed()的开始和结束。

操作函数

现在让我来详细介绍每个pcm的操作函数吧（ops）。通常每个回调函数成功的话返回0，出错的话返回一个带错误码的负值，如：-EINVAL。

每个函数至少要有一个snd_pcm_substream指针变量。主要是为了从给定的子流实例中得到chip记录，你可以采用下面的宏。

```
Int xxx(){
    struct mychip *chip = snd_pcm_substream_chip(substream);
    ....
}
```

open函数

```
static int snd_xxx_open(struct snd_pcm_substream *substream);
```

当打开一个pcm子流的时候调用。

在这里，你至少要初始化runtime->hw记录。典型应用如下：

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
     struct mychip *chip = snd_pcm_substream_chip(substream);
     struct snd_pcm_runtime *runtime = substream->runtime;

     runtime->hw = snd_mychip_playback_hw;
     return 0;
}
```

其中snd_mychip_playback_hw是预先定义的硬件描述。

close函数

```
static int snd_xxx_close(struct snd_pcm_substream *substream)
```

显然这是在pcm子流被关闭的时候调用。

所有在open的时候被分配的pcm子流的私有的实例都应该在这里被释放。

```
Static int snd_xxx_close(struct snd_pcm_substream *substream)
{
    ...
    kfree(substream->runtime->private_data);
    ...
 }
```

ioctl函数

这个函数主要是完成一些pcm ioctl的特殊功能。但是通常你可以采用通用的ioctl函数snd_pcm_lib_ioctl。

hw_params函数

```
static int snd_xxx_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_substream *hw_params);
```

这个函数和hw_free函数仅仅在ALSA0.9.X版本出现。

当pcm子流中已经定义了缓冲区大小，period大小，格式等的时候，应用程序可以通过这个函数来设定硬件参数。

很多的硬件设定都要在这里完成，包括分配内存。

设定的参数可以通过params_xxx()宏得到。对于分配内存，可以采用下面一个有用的函数，

```
snd_pcm_lib_malloc_pages(substream,params_buffer_bytes(hw_params));
```

snd_pcm_lib_malloc_pages()仅仅当DMA缓冲区已经被预分配之后才可以用。参考“缓存区类型”一章获得更详细的细节。

注意这个和prepare函数会在初始化当中多次被调用。例如，OSSemulation可能在每次通过ioctl改变的时候都要调用这些函数。

因而，千万不要对一个相同的内存分配多次，可能会导致内存的漏洞！而上面的几个函数是可以被多次调用的，如果它已经被分配了，它会先自动释放之前的内存。

另外一个需要注意的是，这些函数都不是原子的（可以被调到）。这个是非常重要的，因为trigger函数是原子的（不可被调度）。因此，mutex和其他一些和调度相关的功能函数在trigger里面都不需要了。具体参看“原子操作”一节。

hw_free函数

```
static int snd_xxx_hw_free(struct snd_pcm_substream *substream);
```

这个函数可以是否通过hw_params分配的资源。例如：通过如下函数释放那些通过snd_pcm_lib_malloc_pages()申请的缓存。

```
snd_pcm_lib_free_pages(substream)
```

这个函数要在close调用之前被调用。同时，它也可以被多次调用。它也会知道资源是否已经被分配。

Prepare函数

```
static int snd_xxx_prepare(struct snd_pcm_substream *substream);
```

当pcm“准备好了”的时候调用这个函数。可以在这里设定格式类型，采样率等等。和hw_params不同的是，每次调用snd_pcm_prepare()的时候都要调用prepare函数。

注意最近的版本prepare变成了非原子操作的了。这个函数中，你要做一些调度安全性策略。

在下面的函数中，你会涉及到runtime记录的值(substream->runtime)。例如：得到当前的采样率，格式或声道，可以分别存取runtime->rate，runtime->format,runtime->channels。分配的内存的地址放到runtime->dma_area中，内存和period大小分别保存在runtime->buffer_size和runtime->period_size中。

要注意在每次设定的时候都有可能多次调用这些函数。

trigger函数

```
static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd);
```

当pcm开始，停止或暂停的时候都会调用这个函数。

具体执行那个操作主要是根据第二个参数，在<sound/pcm.h>中声明了SNDRV_PCM_TRIGGER_XXX。最少START和STOP的命令必须定义的。

```
switch(cmd){
case SNDRV_PCM_TRIGGER_START:
     //启动PCM引擎
     break;
case SNDRV_PCM_TRIGGER_STOP:
     //停止PCM引擎
     break;
default:
     break;
}
```

当pcm支持暂停操作（在hareware表里面有这个），也必须处理PAUSE_PAUSE和PAUSE_RELEASE命令。前者是暂停命令，后者是重新播放命令。

假如pcm支持挂起/恢复操作，不管是全部或部分的挂起/恢复支持，都要处理SUSPEND和RESUME命令。这些命令主要是在电源状态改变的时候需要，通常它们和STOP，START命令放到一起。具体参看“电源管理”一章。

如前面提到的，这个操作上原子的。不要在调用这些函数的时候进入睡眠。而trigger函数要尽量小，甚至仅仅触发DMA。另外的工作如初始化hw_params和prepare应该在之前做好。

pointer函数

```
static snd_pcm_uframes_t snd_xxx_pointer(struct snd_pcm_substream *substream);
```

PCM中间层通过调用这个函数来获得缓冲区中的硬件位置。返回值需要以为frames为单位（在ALSA0.5.X是以字节为单位的），范围从0到buffer_size-1。

一般情况下，在中断程序中，调用snd_pcm_period_elapsed()的时候，在pcm中间层在在更新buffer的程序中调用它。然后，pcm中间层会更新指针位置和计算可用的空间，然后唤醒那些等待的线程。

这个函数也是原子的。

Copy和silence函数

这些函数不是必须的，同时在大部分的情况下是被忽略的。这些函数主要应用在硬件内存不在正常的内存空间的时候。一些声卡有一些没有被影射的自己的内存。在这种情况下，你必须把内存传到硬件的内存空间去。或者，缓冲区在物理内存和虚拟内存中都是不连续的时候就需要它们了。

假如定义了copy和silence，就可以做copy和set-silence的操作了。更详细的描述请参考“缓冲区和内存管理”一章。

Ack函数

这个函数也不是必须的。当在读写操作的时候更新appl_ptr的时候会调用它。一些类似于emu10k1-fx和cs46xx的驱动程序会为了内部缓存来跟踪当前的appl_ptr，这个函数仅仅对于这个情况才会被用到。

这个函数也是原子的。

page函数

这个函数也不是必须的。这个函数主要对那些不连续的缓存区 。mmap会调用这个函数得到内存页的地址。后续章节“缓冲区和内存管理”会有一些例子介绍。

中断处理

下面的pcm工作就是PCM中断处理了。声卡驱动中的PCM中断处理的作用主要是更新缓存的位置，然后在缓冲位置超过预先定义的period大小的时候通知PCM中间层。可以通过调用snd_pcm_period_elapsed()来通知。

声卡有如下几种产生中断。

period（周期）中断

这是一个很常见的类型：硬件会产生周期中断。每次中断都会调用snd_pcm_period_elapsed()。

snd_pcm_period_elapsed()的参数是substream的指针。因为，需要从chip实例中得到substream的指针。例如：在chip记录中定义一个substream字段来保持当前运行的substream指针，在open函数中要设定这个字段而在close函数中要复位这个字段。

假如在中断处理函数中获得了一个自旋锁，如果其他pcm也会调用这个锁，那你必须要在调用snd_pcm_period_elapsed()之前释放这个锁。

典型代码如下：

Example5-3.中断函数处理#1

```
struct irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
    struct mychip *chip = dev_id;
    spin_lock(&chip->lock);
    ....
    if (pcm_irq_invoked(chip)){
        spin_unlock(&chip->lock);
        snd_pcm_period_elapsed(chip->substream);
        spin_lock(&chip->lock);
        //如果需要的话，可以响应中断
    }
    ....
    spin_unlock(&chip->lock);
    return IRQ_HANDLED;
}
```

高频率时钟中断

当硬件不再产生一个period（周期）中断的时候，就需要一个固定周期的timer中断了（例如 es1968，ymfpci驱动）。这时候，在每次中断都要检查当前硬件位置，同时计算已经累积的采样的长度。当长度超过period长度时候，需要调用 snd_pcm_period_elapsed()同时复位计数值。

典型代码如下：

Example5-4.中断函数处理#2

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
    struct mychip *chip = dev_id;
    spin_lock(&chip->lock);
    ....
    if (pcm_irq_invoked(chip)){
        unsigned int last_ptr, size;
        /*得到当前的硬件指针（帧为单位）*/
```

```
            last_ptr = get_hw_ptr(chip);
            /*计算自从上次更新之后又处理的帧*/
            if (last_ptr < chip->last_ptr)
            {
                size = runtime->buffer_size + last_ptr - chip->last_ptr
            }else
            {
                size = last_ptr  -  chip->chip->last_ptr;
            }
            //保持上次更新的位置
            chip->last_ptr = last_ptr;
            /*累加 size 计数器*/
            chip->size += size;
            /*超过 period 的边界？ */
            if (chip->size >= runtime->period_size){
                /*重置 size 计数器*/
                chip->size %= runtime->period_size;
                spin_unlock(&chip->lock);
                snd_pcm_period_elapsed(substream);
                spin_lock(&chip->lock);
            }
            //需要的话，要相应中断
        }
        ....
        spin_unlock(&chip->lock);
        return IRQ_HANDLED;
    }
```

在调用 snd_pcm_period_elapsed()的时候

就算超过一个 period 的时间已经过去，你也不需要多次调用 snd_pcm_period_elapsed()，因为 pcm 层会自己检查当前的硬件指针和上次和更新的状态。

原子操作

在内核编程的时候，一个非常重要（又很难 dubug）的问题就是竞争条件。Linux 内核中，一般是通过自旋锁和信号量来解决的。通常来说，假如竞争发生在中断函数中，中断函数要具有原子性，你必须采用自旋锁来包含临界资源。假如不是发生在中断部分，同时比较耗时，可以采用信号量。

如我们看到的，pcm 的操作函数一些是原子的而一些不是。例如：hw_params 函数不是原子的，而 trigger 函数是原子的。这意味着，后者调用的时候，PCM 中间层已经拥有了锁。在这些函数中申请的自旋锁和信号量要做个计算。

在这些原子的函数中，不能那些可能调用任务切换和进入睡眠的函数。其中信号量和互斥体可能会进入睡眠，因此，在原子操作的函数中（如：trigger 函数）不能调用它们。如果在这种函数中调用 delay，可以用 udelay(),或 mdelay()。

约束

假如你的声卡支持不常用的采样率，或仅仅支持固定的采样率，就需要设定一个约束条

件。

例如：为了把采样率限制在一些支持的几种之中，就需要用到函数 snd_pcm_hw_constraint_list()。需要在 open 函数中调用它。

Example5-5.硬件约束示例

```
static unsigned int rates[] =
        {4000,10000,22050,44100};
static unsigned snd_pcm_hw_constraint_list constraints_rates = {
        .count = ARRAY_SIZE(rates),
        .list = rates,
        .mask = 0,
};
static int snd_mychip_pcm_open(struct snd_pcm_substream *substream)
{
    int err;
    ....
    err = snd_pcm_hw_constraint_list(substream->runtime,0,
                                     SNDRV_PCM_HW_PARAM_RATE,
                                     &constraints_rates);
    if (err < 0)
         return err;
    ....
}
```

有多种不同的约束。请参考 sound/pcm.h 中的完整的列表。甚至可以定义自己的约束条件。例如，假如 my_chip 可以管理一个单通道的子流，格式是 S16_LE,另外，它还支持 snd_pcm_hareware 中设定的格式（或其他 constraint_list）。可以设定一个：

Example5-6.为通道设定一个硬件规则

```
static int hw_rule_format_by_channels(struct snd_pcm_hw_params *params,
                                      struct snd_pcm_hw_rule *rule)
{
    struct snd_interval *c = hw_params_interval(params,
                               SNDRV_PCM_HW_PARAM_CHANNELS);
    struct snd_mask *f = hw_param_mask(params,SNDRV_PCM_HW_PARAM_FORMAT);
    struct snd_mask fmt;
    snd_mask_any(&fmt); /*初始化结构体*/
    if (c->min < 2){
        fmt.bits[0] &= SNDRV_PCM_FMTBIT_S16_LE;
        return snd_mask_refine(f, &fmt);
    }
    return 0;
}
```

之后，需要把上述函数加入到你的规则当中去：

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_CHANNELS,
                    hw_rule_channels_by_format,0, SNDRV_PCM_HW_PARAM_FORMAT,
                    -1);
```

当应用程序设定声道数量的时候会调用上面的规则函数。但是应用程序可以在设定声道

数之前设定格式。所以也需要设定对应的规则。

Example5-7.为通道设定一个硬件规则

```
static int hw_rule_format_by_format(struct snd_pcm_hw_params *params,
                                    struct snd_pcm_hw_rule *rule)
{
    struct snd_interval *c = hw_param_interval(params,
                            SNDRV_PCM_HW_PARAM_CHANNELS);
    struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
    struct snd_interval ch;
    snd_interval_any(&ch);
    if (f->bits[0] == SNDRV_PCM_FORMAT_S16_LE){
        ch.min = ch.max = 1;
        ch.integer = 1;
        return snd_interval_refine(c, &ch);
    }
    return 0;
}
```

在open函数中：

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                  hw_rule_channels_by_format,0, SNDRV_PCM_HW_PARAM_CHANNELS,
                  -1);
```

这里我们不会更详细的描述，我仍然想说“直接看源码吧”。

第六章 控制接口

概要

控制接口非常广泛的应用在许多转换，变调等场合，可以从用户空间进行控制。混音器接口是一个最重要的接口。换句话说，在ALSA0.9.x版本，所有的混音器的工作都是通过控制接口API实现的（在0.5.x版本混音器内核API是独立出来的）。

ALSA有一个定义很好的AC97的控制模块。如果你的声卡仅仅支持AC97，你可以忽略这章。

控制接口定义

为了创建一个新的控制接口，需要定义三个函数：info,get和put。然后定义一个snd_kcontrol_new类型的记录，例如:

Example6-1.定义一个控制接口

```
static struct snd_kcontrol_new my_control __devinitdata = {
    .face = 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
};
```

大部分情况是通过snd_ctl_new1()来创建control，这种情况下，你可以像上面一样，在定义的前面加上__devinitdata前缀.

iface字段表示control的类型，如：SNDRV_CTL_ELEM_IFACE_XXX,通常情况都是MIXER。CARD表示一个全局控制，而不是混音器的一部分。假如control和声卡设备联系的非常紧密，如HWDEP,RAMDIDI,TIMER或SEQUENCER,需要特别通过device和subdevice字段标识出。

name字段是表示名称的标识符。在ALSA0.9.X,控制单元名字是非常重要的，因为的角色就是通过它的名字进行分类。有一些预定义的标准的control名字。详细描述请参考下节的“控制单元名字”。

index字段存放这个control的索引号。假如一个名字下面有多个不同的control，就要通过index（索引）来区分了。这是当一个声卡拥有多个解码的时候才会用到。加入索引设定为零，就可以忽略定义。

access字段包括了控制接口的存取控制。提供了一些位的组合，如:SNDRV_CTL_ELEM_XXX。详细描述请参考“接口标志位”一节。

private_value字段这个记录的一个专有的的长整型变量。当调用info,get和put函数的时候，可以通过这个字段传递一些参数值。如果参数都是些很小的数，可以把它们通过移位来组合，也可以存放一个指向一个记录的指针（因为它是长整型的）。

另外三个在回调函数一节介绍。

控制接口名字

有一些control名字的标准。一个control通常根据“源，方向，功能”三部分来命名。

首先，SOURCE定义了control的源，是一个字符串，如：“Master”，“PCM”,“CD”,“Line”。已经有很多预定义好的“源”了。

第二，“方向”则为“Playbck”，“Capture”，“Bypass Playback”，“Bypass

Capture”。或者，它如果省略，那就表示播放和录音双向。

第三个，“功能”。根据控制接口的功能不同有下面三个：“Switch”，“volume”，“route”。

一些控制接口名字的范例如下：“Master Capture Switch”，“PCM Playback Volume”.

也有一些不是采用“源，方向，功能”三部分来命名方式：

全局录音和播放

“Capture Source”，“Capture Switch”和“Capture Volume”用来做全局录音（输入）源，开关，和音量的控制。“Playback Switch”和“Playback Volume”用来做全局的输出开关和音量控制。

音调控制

音调开关和音量名称形式为“Tone Control-XXX”。如：“Tone Control-Switch”，“Tone Control - Bass”，“Tone Control - Center”。

3D 控制

3D 控制开关和音量命名形式为“3D Control - XXX”。如：“3D Control - Switch”，“3D Control - Switch”，“3D Control - Center”,“3D Control - Space”。

麦克风增益

麦克风增益命名如“Mic Boost”或“Mic Boost(6dB)”。

更精确的信息请参考文档（Documentation/sound/alsa/ControlNames.txt）

存取标志

存取标志是一个位标志，主要是区分给定的 control 的存取类型。缺省的存取类型是 SNDRV_CTL_ELEM_ACCESS_READWRITE,意思是允许对 control 进行读写控制。当这个标志位被忽略的时候（为 0），被认为是缺省读写（READWRITE）。

当这个 control 是只读的时候，需要传递 SNDRV_CTL_ELEM_ACCESS_READ。这时候，可以不定义 put 函数。类似的，如果 control 是只写的话（虽然这种可能性很低），要设定为 WRITE 标志，也可以不必定义 get 函数。

加入 control 的值是经常改变的，应该加上 VOLATILE 标志，这意味着 control 可以不用显式通知就可以改变。应用程序应经常查询 control。

当一个 control 是不活动的话，设定 INACTIVE 标志。还有 LOCK 和 OWNER 标志用来改变写权限。

回调函数

info 函数

info 函数可以得到对应 control 的详细信息。它必须存到一个给定的 snd_ctl_elem_info 对象中。例如,对于一个拥有一个元素的布尔型 control 的如下：

Example6-2.info 函数示例

```
static int snd_myctl_info(struct snd_kcontrol *kcontrol,
                          struct snd_ctl_elem_info *uinfo)
{
     uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
     uinfo->count = 1;
     uinfo->value.integer.min = 0;
     uinfo->value.integer.max = 1;
     return 0;
```

```
}
```

type 字段表示 control 的类型。有如下几种：布尔型，整形，枚举型，BYTES 型，IEC958,64 位整形。count 字段表示这个 control 里面的元素的数量。如：一个立体声音量拥有的 count 为 2。value 字段是一个 union（联合）类型。value 的存储依赖于类型。布尔型和整形是一样的。

枚举型和其他类型有一点不同，需要为当前给定的索引项设定字符串。

```
Static int snd_myctl_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_ENUMRATED;
    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;
}
```

get 函数

这个函数用来读取当前 control 的值并返回到用户空间。

例如：

Example6-3.get 函数示例

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

value 字段和 control 类型有关，这点和 info 类似。例如，子驱动和用这个字段来存储一些寄存器的偏移，位的屏蔽。private_value 设定如下：

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

可以通过以下函数重新得到：

```
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;
    ....
```

```
}
```

在 get 函数中，加入 control 拥有超过一个的元素。如 count 大于 1，就必须填充所有的元素。上面的例子中，因为我们假定 count 为 1，所以我们仅仅填充了一个元素（value.integer.value[0]）。

put 函数

这个函数主要是从用户空间写一个值

例如：

Example6-4.put 函数示例

```
static int snd_myctrl_put(struct snd_kcontrol *kcontrol,
                          struct snd_ctl_elem_value *ucontrol)
{
    struct mychip *chip = snd_kcontrol_chip(kcontrol);
    int changed = 0;
    if (chip->current_value != ucontrol->value.integer.value[0]){
        change_current_value(chip,ucontrol->value.integer.value[0]);
        changed = 1;
    }
    return changed;
}
```

如上，假如 value 被改变要返回 1，没有改变返回 0。假如有错误发生，通常返回一个带错误码的负数。

像 get 函数一样，如果 control 拥有超过一个的元素，在 put 函数中所有的元素都要比较一下。

回调函数不是原子的。

所有上面的 3 个回调函数都是非原子的。

构造器

当所有事情都准备好的时候，我们就可以创建一个新的 control 了。为了创建它，首先要调用两个函数，snd_ctl_new1()和 snd_ctl_add()。

一个简单的方式如下：

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

my_control 是前面定义好的 snd_kcontrol_new 类型的对象，chip 是一个指向 kcontrol->private_data 的指针，可以被回调函数调用。

snd_ctl_new1()分配了一个新的 snd_kcontrol 实例（这也是为何 my_control 可以带有 __devinitdata 前缀的原因了），snd_ctl_add 会把给定的 control 组件添加到 card 里面。

更改通知

假如你需要在中断程序中改变或更新一个 control，你需要调用 snd_ctl_notify()。

例如：

```
snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE,id_pointer);
```

这个函数需要 card 指针，event_mask，和 control id 指针作为参数。event-mask 表示 notification 的类型，如上述示例，是通知改变 control 的值。id 指针是指向一个 snd_ctl_elem_id 的结构体。在 es1938.c 或 es1968.c 中关于硬件的卷的中断部分有相关示例。