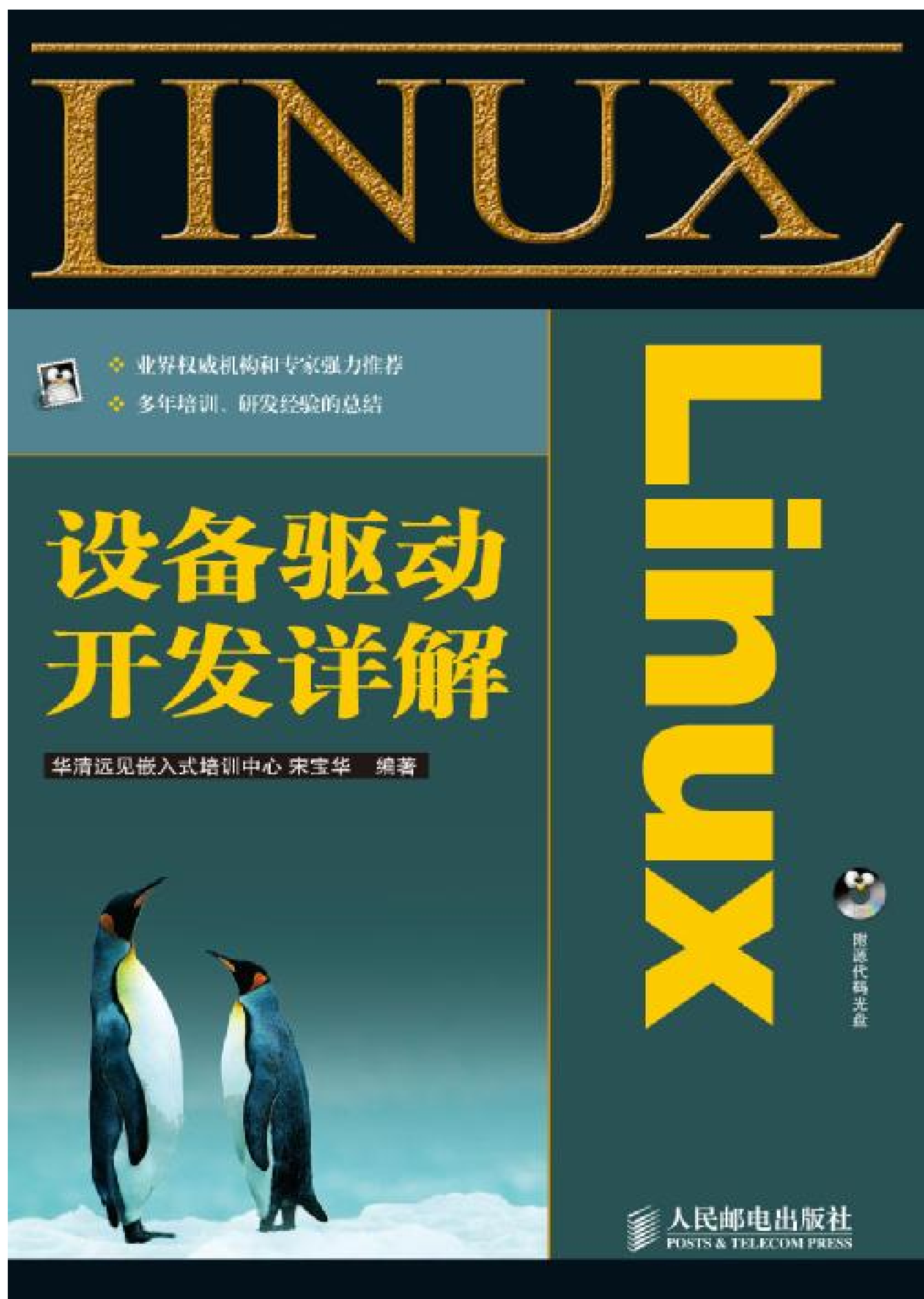

LINUX
业界权威机构和专家强力推荐
多年培训、研发经验的总结
设备驱动开发详解
华清远见嵌入式培训中心 宋宝华 编著
Linux
附源代码光盘
人民邮电出版社
POSTS & TELECOM PRESS

第 17 章　Linux 音频设备驱动

在 Linux 系统中，先后出现了音频设备的两种框架：OSS 和 ALSA，本节将在介绍数字音频设备及音频设备硬件接口的基础上讲解 OSS 和 ALSA 驱动的结构。

17.1～17.2 节讲解了音频设备及 PCM、IIS 和 AC97 硬件接口。

17.3 节讲解了 Linux OSS 音频设备驱动的组成、mixer 接口、dsp 接口及用户空间编程方法。

17.4 节讲解了 Linux ALSA 音频设备驱动的组成、card 和组件管理、PCM 设备、control 接口、AC97 API 及用户空间编程方法。

17.5 节以 S3C2410 通过 IIS 接口外接 UDA1341 编解码器的实例讲解了 OSS 驱动。

17.6 节以 PXA255 通过 AC97 接口外接 AC97 编解码器的实例讲解了 ALSA 驱动。

华清远见

17.1 数字音频设备

目前，手机、PDA、MP3 等许多嵌入式设备中包含了数字音频设备，一个典型的数字音频系统的电路组成如图 17.1 所示。图 17.1 中的嵌入式微控制器/DSP 中集成了 PCM、IIS 或 AC97 音频接口，通过这些接口连接外部的音频编解码器即可实现声音的 AD 和 DA 转换，图中的功放完成模拟信号的放大功能。

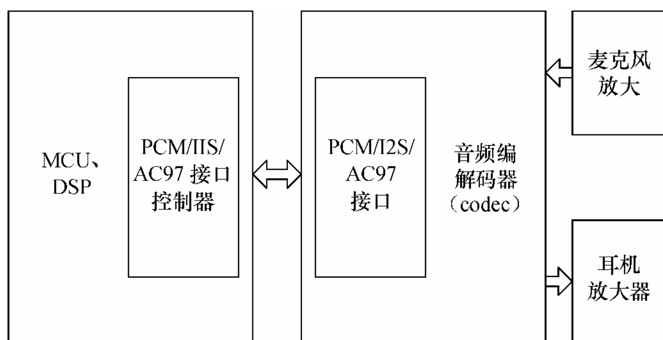


图 17.1　典型的数字音频系统电路

音频编解码器是数字音频系统的核心，衡量它的主要指标如下。

1．采样频率

采样的过程就是将通常的模拟音频信号的电信号转换成二进制码 0 和 1 的过程，这些 0 和 1 便构成了数字音频文件。图 17.2 中的正弦曲线代表原始音频曲线，方格代表采样后得到的结果，二者越吻合说明采样结果越好。

华清远见

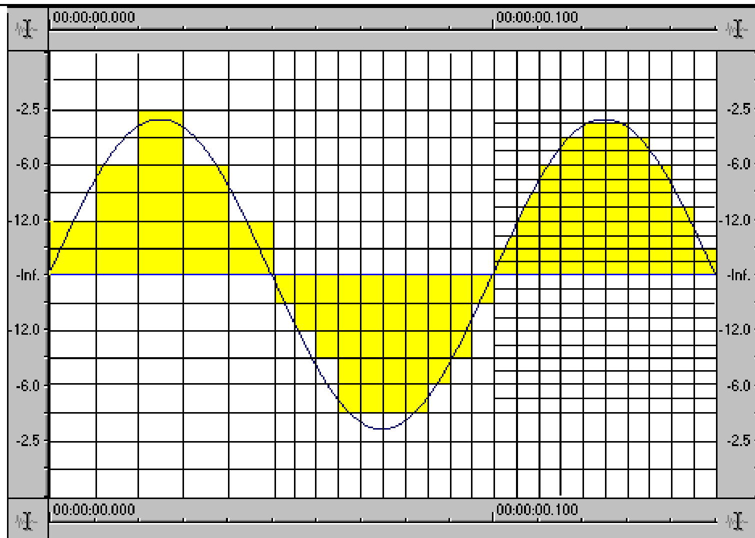


图 17.2　数字音频采样

采样频率是每秒钟的采样次数，我们常说的 44.1kHz 采样频率就是每秒钟采样 44100 次。理论上采样频率越高，转换精度越高，目前主流的采样频率是 48kHz。

2．量化精度

量化精度是指对采样数据分析的精度，比如 24bit 量化精度就是指将标准电平信号按照 2 的 24 次方进行分析，也就是说将图 17.2 中的纵坐标等分为 2^{24} 等分。量化精度越高，声音就越逼真。

17.2 音频设备硬件接口

17.2.1　PCM 接口

针对不同的数字音频子系统，出现了几种微处理器或 DSP 与音频器件间用于数字转换的接口。

最简单的音频接口是 PCM（脉冲编码调制）接口，该接口由时钟脉冲（BCLK）、帧同步信号（FS）及接收数据（DR）和发送数据（DX）组成。在 FS 信号的上升沿，数据传输从 MSB（Most Significant Bit）开始，FS 频率等于采样率。FS 信号之后开始数据字的传输，单个的数据位按顺序进行传输，一个时钟周期传输一个数据字。发送 MSB 时，信号的等级首先降到最低，以避免在不同终端的接口使用不同的数据方案时造成 MSB 的丢失。

PCM 接口很容易实现，原则上能够支持任何数据方案和任何采样率，但需要每个音频通道获得一个独立的数据队列。

17.2.2　IIS 接口

IIS 接口（Inter-IC Sound）在 20 世纪 80 年代首先被 PHILIPS 用于消费音频产品，并在

一个称为 LRCLK（Left/Right CLOCK）的信号机制中经过多路转换，将两路音频信号变成单一的数据队列。当 LRCLK 为高时，左声道数据被传输；LRCLK 为低时，右声道数据被传输。与 PCM 相比，IIS 更适合于立体声系统。对于多通道系统，在同样的 BCLK 和 LRCLK 条件下，并行执行几个数据队列也是可能的。

17.2.3 AC97 接口

AC'97（Audio Codec 1997）是以 Intel 为首的 5 个 PC 厂商 Intel、Creative Labs、NS、Analog Device 与 Yamaha 共同提出的规格标准。与 PCM 和 IIS 不同，AC'97 不只是一种数据格式，用于音频编码的内部架构规格，它还具有控制功能。AC'97 采用 AC-Link 与外部的编解码器相连，AC-Link 接口包括位时钟（BITCLK）、同步信号校正（SYNC）和从编码到处理器及从处理器中解码（SDATDIN 与 SDATAOUT）的数据队列。AC'97 数据帧以 SYNC 脉冲开始，包括 12 个 20 位时间段（时间段为标准中定义的不同的目的服务）及 16 位“tag”段，共计 256 个数据序列。例如，时间段“1”和“2”用于访问编码的控制寄存器，而时间段“3”和“4”分别负载左、右两个音频通道。“tag”段表示其他段中哪一个包含有效数据。把帧分成时间段使传输控制信号和音频数据仅通过 4 根线到达 9 个音频通道或转换成其他数据流成为可能。与具有分离控制接口的 IIS 方案相比，AC97 明显减少了整体管脚数。一般来说，AC'97 编解码器采用 TQFP48 封装，如图 17.3 所示。

PCM、IIS 和 AC97 各有其优点和应用范围，例如在 CD、MD、MP3 随身听多采用 IIS 接口，移动电话会采用 PCM 接口，具有音频功能的 PDA 则多使用和 PC 一样的 AC'97 编码格式。

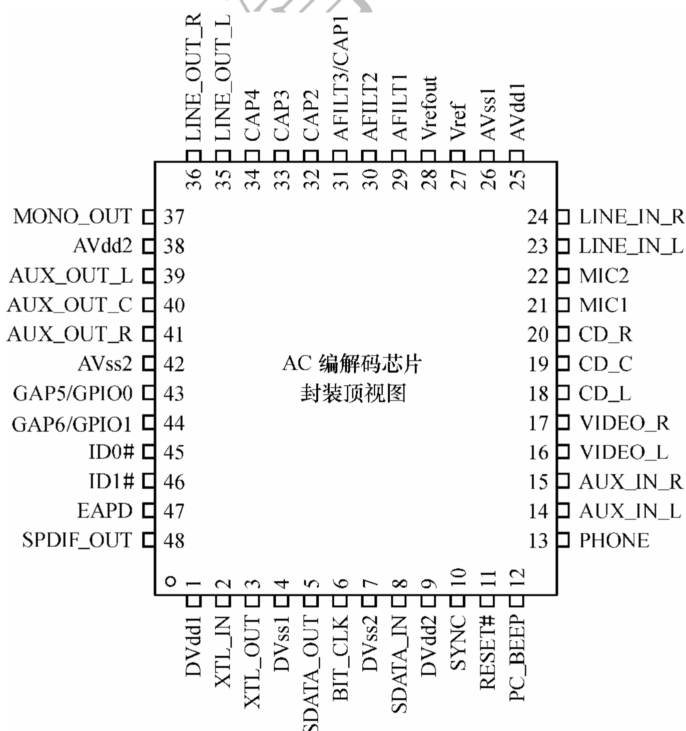


图 17.3　AC97 Codec 芯片

17.3 Linux OSS 音频设备驱动

17.3.1　OSS 驱动的组成

OSS 标准中有两个最基本的音频设备：mixer（混音器）和 dsp（数字信号处理器）。

在声卡的硬件电路中，mixer 是一个很重要的组成部分，它的作用是将多个信号组合或者叠加在一起，对于不同的声卡来说，其混音器的作用可能各不相同。OSS 驱动中，/dev/mixer 设备文件是应用程序对 mixer 进行操作的软件接口。

混音器电路通常由两部分组成：输入混音器（input mixer）和输出混音器（output mixer）。输入混音器负责从多个不同的信号源接收模拟信号，这些信号源有时也被称为混音通道或者混音设备。模拟信号通过增益控制器和由软件控制的音量调节器，在不同的混音通道中进行级别（level）调制，然后被送到输入混音器中进行声音的合成。混音器上的电子开关可以控制哪些通道中有信号与混音器相连，有些声卡只允许连接一个混音通道作为录音的音源，而有些声卡则允许对混音通道做任意的连接。经过输入混音器处理后的信号仍然为模拟信号，它们将被送到 A/D 转换器进行数字化处理。

输出混音器的工作原理与输入混音器类似，同样也有多个信号源与混音器相连，并且事先都经过了增益调节。当输出混音器对所有的模拟信号进行了混合之后，通常还会有一个总控增益调节器来控制输出声音的大小，此外还有一些音调控制器来调节输出声音的音调。经过输出混音器处理后的信号也是模拟信号，它们最终会被送给喇叭或者其他的模拟输出设备。

对混音器的编程包括如何设置增益控制器的级别，以及怎样在不同的音源间进行切换，这些操作通常来讲是不连续的，而且不会像录音或者播放那样需要占用大量的计算机资源。由于混音器的操作不符合典型的读/写操作模式，因此除了 open()和 close()这两个系统调用之外，大部分的操作都是通过 ioctl()系统调用来完成的。与/dev/dsp 不同，/dev/mixer 允许多个应用程序同时访问，并且混音器的设置值会一直保持到对应的设备文件被关闭为止。

DSP 也称为编解码器，实现录音（录音）和放音（播放），其对应的设备文件是/dev/dsp 或/dev/sound/dsp。OSS 声卡驱动程序提供的/dev/dsp 是用于数字采样和数字录音的设备文件，向该设备写数据即意味着激活声卡上的 D/A 转换器进行播放，而向该设备读数据则意味着激活声卡上的 A/D 转换器进行录音。

从 DSP 设备读取数据时，从声卡输入的模拟信号经过 A/D 转换器变成数字采样后的样本，保存在声卡驱动程序的内核缓冲区中，当应用程序通过 read()系统调用从声卡读取数据时，保存在内核缓冲区中的数字采样结果将被复制到应用程序所指定的用户缓冲区中。

需要指出的是，声卡采样频率是由内核中的驱动程序所决定的，而不取决于应用程序从声卡读取数据的速度。如果应用程序读取数据的速度过慢，以致低于声卡的采

样频率，那么多余的数据将会被丢弃（即 overflow）；如果读取数据的速度过快，以致高于声卡的采样频率，那么声卡驱动程序将会阻塞那些请求数据的应用程序，直到新的数据到来为止。

向 DSP 设备写入数据时，数字信号会经过 D/A 转换器变成模拟信号，然后产生声音。应用程序写入数据的速度应该至少等于声卡的采样频率，过慢会产生声音暂停或者停顿的现象（即 underflow）。如果用户写入过快的话，它会被内核中的声卡驱动程序阻塞，直到硬件有能力处理新的数据为止。

与其他设备有所不同，声卡通常不需要支持非阻塞（non-blocking）的 I/O 操作。即便内核 OSS 驱动提供了非阻塞的 I/O 支持，用户空间也不宜采用。

无论是从声卡读取数据，或是向声卡写入数据，事实上都具有特定的格式（format），如无符号 8 位、单声道、8kHz 采样率，如果默认值无法达到要求，可以通过 ioctl()系统调用来改变它们。通常说来，在应用程序中打开设备文件/dev/dsp 之后，接下去就应该为其设置恰当的格式，然后才能从声卡读取或者写入数据。

17.3.2　mixer 接口

```
int register_sound_mixer(struct file_operations *fops, int dev);
```

上述函数用于注册一个混音器，第一个参数 fops 即是文件操作接口，第二个参数 dev 是设备编号，如果填入-1，则系统自动分配一个设备编号。mixer 是一个典型的字符设备，因此编码的主要工作是实现 file_operations 中的 open()、ioctl()等函数。

mixer 接口 file_operations 中的最重要函数是 ioctl()，它实现混音器的不同 I/O 控制命令，代码清单 17.1 所示为一个 ioctl()的范例。

代码清单 17.1　mixer()接口的 ioctl()函数范例

```
1   static int mixdev_ioctl(struct inode *inode, struct file *file, unsigned int cmd, 
unsigned long arg)
2   {
3    ...
4    switch (cmd)
5    {
6      case SOUND_MIXER_READ_MIC:
7        ...
8      case SOUND_MIXER_WRITE_MIC:
9        ...
10     case SOUND_MIXER_WRITE_RECSRC:
11       ...
12     case SOUND_MIXER_WRITE_MUTE:
13       ...
14   }
15   //其他命令
```

```
16   return mixer_ioctl(codec, cmd, arg);
17 }
```

17.3.3　dsp 接口

```
int register_sound_dsp(struct file_operations *fops, int dev);
```

上述函数与 register_sound_mixer()类似，它用于注册一个 dsp 设备，第一个参数 fops 即是文件操作接口，第二个参数 dev 是设备编号，如果填入-1，则系统自动分配一个设备编号。dsp 也是一个典型的字符设备，因此编码的主要工作是实现 file_operations 中的 read()、write()、ioctl()等函数。

dsp 接口 file_operations 中的 read()和 write()函数非常重要，read()函数从音频控制器中获取录音数据到缓冲区并复制到用户空间，write()函数从用户空间复制音频数据到内核空间缓冲区并最终发送到音频控制器。

dsp 接口 file_operations 中的 ioctl()函数处理对采样率、量化精度、DMA 缓冲区块大小等参数设置 I/O 控制命令的处理。

在数据从缓冲区复制到音频控制器的过程中，通常会使用 DMA，DMA 对声卡而言非常重要。例如，在放音时，驱动设置完 DMA 控制器的源数据地址（内存中的 DMA 缓冲区）、目的地址（音频控制器 FIFO）和 DMA 的数据长度，DMA 控制器会自动发送缓冲区的数据填充 FIFO，直到发送完相应的数据长度后才中断一次。

在 OSS 驱动中，建立存放音频数据的环形缓冲区（ring buffer）通常是值得推荐的方法。此外，在 OSS 驱动中，一般会将一个较大的 DMA 缓冲区分成若干个大小相同的块（这些块也被称为“段”，即 fragment），驱动程序使用 DMA 每次在声音缓冲区和声卡之间搬移一个 fragment。在用户空间，可以使用 ioctl()系统调用来调整块的大小和个数。

除了 read()、write()和 ioctl()外，dsp 接口的 poll()函数通常也需要被实现，以向用户反馈目前能否读写 DMA 缓冲区。

在 OSS 驱动初始化过程中，会调用 register_sound_dsp()和 register_sound_mixer()注册 dsp 和 mixer 设备；在模块卸载的时候，会调用代码清单 17.2。

代码清单 17.2　OSS 驱动初始化注册 dsp 和 mixer 设备

```
1 static int xxx_init(void)
2 {
3   struct xxx_state *s = &xxx_state;
4   ...
5   //注册 dsp 设备
6   if ((audio_dev_dsp = register_sound_dsp(&xxx_audio_fops, - 1)) <
0)
7        goto err_dev1;
8   //设备 mixer 设备
9   if ((audio_dev_mixer = register_sound_mixer(&xxx_mixer_fops, - 1))
< 0)
10       goto err_dev2;
11  ...
12 }
13
14 void __exit xxx_exit(void)
```


华清远见

```
15 {
16   //注销 dsp 和 mixer 设备接口
17   unregister_sound_dsp(audio_dev_dsp);
18   unregister_sound_mixer(audio_dev_mixer);
19   ...
20 }
```

根据 17.3.2 和 17.3.3 小节的分析，可以画出一个 Linux OSS 驱动结构的简图，如图 17.4 所示。

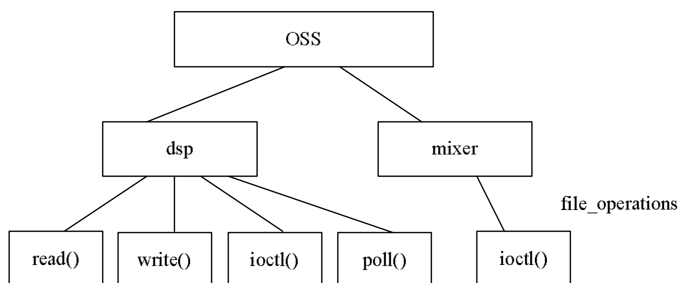


图 17.4　Linux OSS 驱动结构

17.3.4　OSS 用户空间编程

1．dsp 编程

对 OSS 驱动声卡的编程使用 Linux 文件接口函数，如图 17.5 所示，dsp 接口的操作一般包括如下几个步骤。

（1）打开设备文件/dev/dsp。

采用何种模式对声卡进行操作也必须在打开设备时指定，对于不支持全双工的声卡来说，应该使用只读或者只写的方式打开，只有那些支持全双工的声卡，才能以读写的方式打开，这还依赖于驱动程序的具体实现。Linux 允许应用程序多次打开或者关闭与声卡对应的设备文件，从而能够很方便地在放音状态和录音状态之间进行切换。

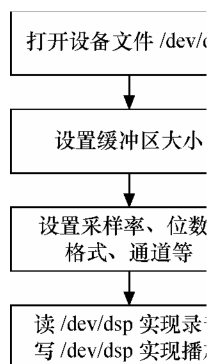


图 17.5　OSS dsp 接口用户空间操作流程

（2）如果有需要，设置缓冲区大小。

运行在 Linux 内核中的声卡驱动程序专门维护了一个缓冲区，其大小会影响到播放和录音时的效果，使用 ioctl()系统调用可以对它的尺寸进行恰当设置。调节驱动程序中缓冲区大小的操作不是必须的，如果没有特殊的要求，一般采用默认的缓冲区大小也就可以了。如果想设置缓冲区的大小，则通常应紧跟在设备文件打开之后，这是因为对声卡的其他操作有可能会导致驱动程序无法再修改其缓冲区的大小。

（3）设置声道（channel）数量。

根据硬件设备和驱动程序的具体情况，可以设置为单声道或者立体声。

（4）设置采样格式和采样频率

采样格式包括 AFMT_U8（无符号 8 位）、AFMT_S8（有符号 8 位）、AFMT_U16_LE

（小端模式，无符号 16 位）、AFMT_U16_BE（大端模式，无符号 16 位）、AFMT_MPEG、AFMT_AC3 等。使用 SNDCTL_DSP_SETFMT IO 控制命令可以设置采样格式。

对于大多数声卡来说，其支持的采样频率范围一般为 5kHz～44.1kHz 或者 48kHz，但并不意味着该范围内的所有连续频率都会被硬件支持，在 Linux 系统下进行音频编程时最常用到的几种采样频率是 11025Hz、16000Hz、22050Hz、32000Hz 和 44100Hz。使用 SNDCTL_DSP_SPEED IO 控制命令可以设置采样频率。

（5）读写/dev/dsp 实现播放或录音。

代码清单 17.3 的程序实现了利用/dev/dsp 接口进行声音录制和播放的过程，它的功能是先录制几秒钟音频数据，将其存放在内存缓冲区中，然后再进行播放。

代码清单 17.3　OSS dsp 接口应用编程范例

```
1  #include <unistd.h>
2  #include <fcntl.h>
3  #include <sys/types.h>
4  #include <sys/ioctl.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <linux/soundcard.h>
8  #define LENGTH 3          /* 存储秒数 */
9  #define RATE 8000         /* 采样频率 */
10 #define SIZE 8            /* 量化位数 */
11 #define CHANNELS 1        /* 声道数目 */
12 /* 用于保存数字音频数据的内存缓冲区 */
13 unsigned char buf[LENGTH *RATE * SIZE * CHANNELS / 8];
14 int main()
15 {
16   int fd; /* 声音设备的文件描述符 */
17   int arg; /* 用于 ioctl 调用的参数 */
18   int status; /* 系统调用的返回值 */
19   /* 打开声音设备 */
20   fd = open("/dev/dsp", O_RDWR);
21   if(fd < 0)
22   {
23      perror("open of /dev/dsp failed");
24      exit(1);
25   }
26   /* 设置采样时的量化位数 */
27   arg = SIZE;
28   status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
29   if (status ==  - 1)
30       perror("SOUND_PCM_WRITE_BITS ioctl failed");
31   if (arg != SIZE)
32     perror("unable to set sample size");
33   /* 设置采样时的通道数目 */
34   arg = CHANNELS;
```

```
35   status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
36   if(status ==  - 1)
37     perror("SOUND_PCM_WRITE_CHANNELS ioctl failed");
38   if(arg != CHANNELS)
39     perror("unable to set number of channels");
40   /* 设置采样率 */
41   arg = RATE;
42   status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
43   if(status ==  - 1)
44     perror("SOUND_PCM_WRITE_WRITE ioctl failed");
45   /* 循环，直到按下[ControltC]*/
46   while (1)
47   {
48     printf("Say something:\n");
49     status = read(fd, buf, sizeof(buf)); /* 录音 */
50     if(status != sizeof(buf))
51       perror("read wrong number of bytes");
52     printf("You said:\n");
53     status = write(fd, buf, sizeof(buf)); /* 放音 */
54     if(status != sizeof(buf))
55       perror("wrote wrong number of bytes");
56     /* 在继续录音前等待放音结束 */
57     status = ioctl(fd, SOUND_PCM_SYNC, 0);
58     if(status ==  - 1)
59       perror("SOUND_PCM_SYNC ioctl failed");
60   }
61 }
```

2．mixer 编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件/dev/mixer 进行编程。对混音器的操作一般都通过 ioctl()系统调用来完成，所有控制命令都以 SOUND_MIXER 或者 MIXER 开头，表 17.1 列出了常用的混音器控制命令。

表 17.1　混音器常用命令

命　令	作　用
SOUND_MIXER_VOLUME	主音量调节
SOUND_MIXER_BASS	低音控制
SOUND_MIXER_TREBLE	高音控制
SOUND_MIXER_SYNTH	FM 合成器
SOUND_MIXER_PCM	主 D/A 转换器
SOUND_MIXER_SPEAKER	PC 喇叭

SOUND_MIXER_LINE	音频线输入
SOUND_MIXER_MIC	麦克风输入
SOUND_MIXER_CD	CD 输入
SOUND_MIXER_IMIX	放音音量
SOUND_MIXER_ALTPCM	从 D/A 转换器
SOUND_MIXER_RECLEV	录音音量

续表

命　　令	作　　用
SOUND_MIXER_IGAIN	输入增益
SOUND_MIXER_OGAIN	输出增益
SOUND_MIXER_LINE1	声卡的第 1 输入
SOUND_MIXER_LINE2	声卡的第 2 输入
SOUND_MIXER_LINE3	声卡的第 3 输入

对声卡的输入增益和输出增益进行调节是混音器的一个主要作用，目前大部分声卡采用的是 8 位或者 16 位的增益控制器，声卡驱动程序会将它们变换成百分比的形式，也就是说无论是输入增益还是输出增益，其取值范围都是从 0～100。

（1）SOUND_MIXER_READ 宏。

在进行混音器编程时，可以使用 SOUND_MIXER_READ 宏来读取混音通道的增益大小，例如，如下代码可以获得麦克风的输入增益：

```
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
```

对于只有一个混音通道的单声道设备来说，返回的增益大小保存在低位字节中。而对于支持多个混音通道的双声道设备来说，返回的增益大小实际上包括两个部分，分别代表左、右两个声道的值，其中低位字节保存左声道的音量，而高位字节则保存右声道的音量。下面的代码可以从返回值中依次提取左右声道的增益大小：

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
```

（2）SOUND_MIXER_WRITE 宏。

如果想设置混音通道的增益大小，则可以通过 SOUND_MIXER_WRITE 宏来实现，例如下面的语句可以用来设置麦克风的输入增益：

```
vol = (right << 8) + left;
ioctl(fd, SOUND_MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

（3）查询 MIXER 信息。

声卡驱动程序提供了多个 ioctl()系统调用来获得混音器的信息，它们通常返回一个整型的位掩码，其中每一位分别代表一个特定的混音通道，如果相应的位为 1，则说明与之对应的混音通道是可用的。

通过 SOUND_MIXER_READ_DEVMASK 返回的位掩码查询出能够被声卡支持的每一个混音通道，而通过 SOUND_MIXER_READ_RECMAS 返回的位掩码则可以查询出能够被当作录音源的每一个通道。例如，如下代码可用来检查 CD 输入是否是一

个有效的混音通道：

```
ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
if (devmask & SOUND_MIXER_CD)
   printf("The CD input is supported");
```

如下代码可用来检查 CD 输入是否是一个有效的录音源：

```
ioctl(fd, SOUND_MIXER_READ_RECMASK, &recmask);
if (recmask & SOUND_MIXER_CD)
   printf("The CD input can be a recording source");
```

大多数声卡提供了多个录音源，通过 SOUND_MIXER_READ_RECSRC 可以查询出当前正在使用的录音源，同一时刻可使用两个或两个以上的录音源，具体由声卡硬件本身决定。相应地，使用 SOUND_MIXER_WRITE_RECSRC 可以设置声卡当前使用的录音源，如下代码可以将 CD 输入作为声卡的录音源使用。

```
devmask = SOUND_MIXER_CD;

ioctl(fd, SOUND_MIXER_WRITE_RECSRC, &devmask);
```

此外，所有的混音通道都有单声道和双声道的区别，如果需要知道哪些混音通道提供了对立体声的支持，可以通过 SOUND_MIXER_READ_STEREODEVS 来获得。

代码清单 17.4 的程序实现了利用/dev/mixer 接口对混音器进行编程的过程，该程序可对各种混音通道的增益进行调节。

代码清单 17.4　OSS mixer 接口应用编程范例

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/ioctl.h>
5  #include <fcntl.h>
6  #include <linux/soundcard.h>
7  /* 用来存储所有可用混音设备的名称 */
8  const char *sound_device_names[] = SOUND_DEVICE_NAMES;
9  int fd; /* 混音设备所对应的文件描述符 */
10 int devmask, stereodevs; /* 混音器信息对应的 bit 掩码 */
11 char *name;
12 /* 显示命令的使用方法及所有可用的混音设备 */
13 void usage()
14 {
15   int i;
16        fprintf(stderr,  "usage:  %s  <device>  <left-gain%%>
<right-gain%%>\n"
17     "%s <device> <gain%%>\n\n""Where <device> is one of:\n", name,
name);
18   for(i = 0; i < SOUND_MIXER_NRDEVICES; i++)
19      if ((1 << i) &devmask)
20    /* 只显示有效的混音设备 */
21       fprintf(stderr, "%s ", sound_device_names[i]);
22   fprintf(stderr, "\n");
23   exit(1);
24 }
25
```

```
26  int main(int argc, char *argv[])
27  {
28    int left, right, level; /* 增益设置 */
29    int status; /* 系统调用的返回值 */
30    int device; /* 选用的混音设备 */
31    char *dev; /* 混音设备的名称 */
32    int i;
33    name = argv[0];
34    /* 以只读方式打开混音设备 */
35    fd = open("/dev/mixer", O_RDONLY);
36    if (fd ==  - 1)
37    {
38      perror("unable to open /dev/mixer");
39      exit(1);
40    }
41
42    /* 获得所需要的信息 */
43    status = ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
44    if(status ==  - 1)
45       perror("SOUND_MIXER_READ_DEVMASK ioctl failed");
46    status = ioctl(fd, SOUND_MIXER_READ_STEREODEVS, &stereodevs);
47    if (status ==  - 1)
48         perror("SOUND_MIXER_READ_STEREODEVS ioctl failed");
49    /* 检查用户输入 */
50    if (argc != 3 && argc != 4)
51        usage();
52    /* 保存用户输入的混音器名称 */
53    dev = argv[1];
54    /* 确定即将用到的混音设备 */
55    for (i = 0; i < SOUND_MIXER_NRDEVICES; i++)
56                    if   (((1   <<   i)   &devmask)   &&   !strcmp(dev,
sound_device_names[i]))
57            break;
58    if (i == SOUND_MIXER_NRDEVICES)
59    {
60        /* 没有找到匹配项 */
61      fprintf(stderr, "%s is not a valid mixer device\n", dev);
62      usage();
63    }
64    /* 查找到有效的混音设备 */
65    device = i;
66    /* 获取增益值 */
67    if (argc == 4)
68    {
69      /* 左、右声道均给定 */
70      left = atoi(argv[2]);
71      right = atoi(argv[3]);
72    }
73    else
74    {
75      /* 左、右声道设为相等 */
76      left = atoi(argv[2]);
77      right = atoi(argv[2]);
78    }
79
```

```
    80    /* 对非立体声设备给出警告信息 */
    81    if ((left != right) && !((1 << i) &stereodevs))
    82    {
    83      fprintf(stderr, "warning: %s is not a stereo device\n", dev);
    84    }
    85
    86    /* 将两个声道的值合到同一变量中 */
    87    level = (right << 8) + left;
    88
    89    /* 设置增益 */
    90    status = ioctl(fd, MIXER_WRITE(device), &level);
    91    if (status ==  - 1)
    92      {
    93        perror("MIXER_WRITE ioctl failed");
    94        exit(1);
    95    }
    96    /* 获得从驱动返回的左右声道的增益 */
    97       left = level &0xff;
    98       right = (level &0xff00) >> 8;
    99       /* 显示实际设置的增益 */
    100       fprintf(stderr, "%s gain set to %d%% / %d%%\n", dev, left,
right);
    101      /* 关闭混音设备 */
    102      close(fd);
    103      return 0;
    104 }
```

编译上述程序为可执行文件 mixer，执行 ./mixer <device> <left-gain%> <right-gain%>或./mixer <device> <gain%>可设置增益，device 可以是 vol、pcm、speaker、line、mic、cd、igain、line1、phin、video。

17.4 Linux ALSA 音频设备驱动

17.4.1 ALSA 的组成

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，而 ALSA（Advanced Linux Sound Architecture）恰好弥补了这一空白，它符合 GPL，是在 Linux 下进行音频编程时另一种可供选择的声卡驱动体系结构。ALSA 除了像 OSS 那样提供了一组内核驱动程序模块之外，还专门为简化应用程序的编写提供了相应的函数库，与 OSS 提供的基于 ioctl 的原始编程接口相比，ALSA

函数库使用起来要更加方便一些。ALSA 的主要特点如下。

- 支持多种声卡设备。
- 模块化的内核驱动程序。
- 支持 SMP 和多线程。
- 提供应用开发函数库（alsa-lib）以简化应用程序开发。
- 支持 OSS API，兼容 OSS 应用程序。

ALSA 具有更加友好的编程接口，并且完全兼容于 OSS，对应用程序员来讲无疑是一个更佳的选择。ALSA 系统包括驱动包 alsa-driver、开发包 alsa-libs、开发包插件 alsa-libplugins、设置管理工具包 alsa-utils、其他声音相关处理小程序包 alsa-tools、特殊音频固件支持包 alsa- firmware、OSS 接口兼容模拟层工具 alsa-oss 共 7 个子项目，其中只有驱动包是必需的。

alsa-driver 指内核驱动程序，包括硬件相关的代码和一些公共代码，非常庞大，代码总量达数十万行；alsa-libs 指用户空间的函数库，提供给应用程序使用，应用程序应包含头文件 asoundlib.h，并使用共享库 libasound.so；alsa-utils 包含一些基于 ALSA 的用于控制声卡的应用程序，如 alsaconf（侦测系统中声卡并写一个适合的 ALSA 配置文件）、alsactl（控制 ALSA 声卡驱动的高级设置）、alsamixer（基于 ncurses 的混音器程序）、amidi（用于读写 ALSA RawMIDI）、amixer（ALSA 声卡混音器的命令行控制）、aplay（基于命令行的声音文件播放）、arecord（基于命令行的声音文件录制）等。

目前 ALSA 内核提供给用户空间的接口有：

- 信息接口（Information Interface，/proc/asound）；
- 控制接口（Control Interface，/dev/snd/controlCX）；
- 混音器接口（Mixer Interface，/dev/snd/mixerCXDX）；
- PCM 接口（PCM Interface，/dev/snd/pcmCXDX）；
- Raw 迷笛接口（Raw MIDI Interface，/dev/snd/midiCXDX）；
- 音序器接口（Sequencer Interface，/dev/snd/seq）；
- 定时器接口（Timer Interface，/dev/snd/timer）。

和 OSS 类似，上述接口也以文件的方式被提供，不同的是这些接口被提供给 alsa-lib 使用，而不是直接给应用程序使用的。应用程序最好使用 alsa-lib，或者更高级的接口，比如 jack 提供的接口。

图 17.6 所示为 ALSA 声卡驱动与用户空间体系结构的简图，从中可以看出 ALSA 内核驱动与用户空间库及 OSS 之间的关系。

华清远见

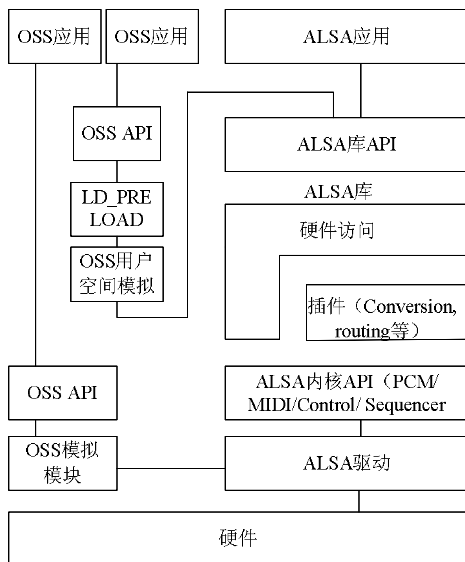


图 17.6　ALSA 体系结构

17.4.2　card 和组件管理

对于每个声卡而言，必须创建一个 card 实例。card 是声卡的"总部"，它管理这个声卡上的所有设备（组件），如 PCM、mixers、MIDI、synthesizer 等。因此，card 和组件是 ALSA 声卡驱动中的主要组成元素。

1．创建 card

```
struct snd_card *snd_card_new(int idx, const char *xid,
                               struct module *module, int extra_size);
```

idx 是 card 索引号，xid 是标识字符串，module 一般为 THIS_MODULE，extra_size 是要分配的额外数据的大小，分配的 extra_size 大小的内存将作为 card->private_data。

2．创建组件

```
int snd_device_new(struct snd_card *card, snd_device_type_t type,
                   void *device_data, struct snd_device_ops *ops);
```

当 card 被创建后，设备（组件）能够被创建并关联于该 card。第 1 个参数是 snd_card_new()创建的 card 指针，第 2 个参数 type 指的是 device-level 即设备类型，形式为 SNDRV_DEV_XXX，包括 SNDRV_DEV_CODEC、SNDRV_DEV_CONTROL、SNDRV_DEV_PCM、SNDRV_DEV_ RAWMIDI 等，用户自定义设备的 device-level 是 SNDRV_DEV_LOWLEVEL，ops 参数是 1 个函数集（定义为 snd_device_ops 结构体）的指针，device_data 是设备数据指针，注意函数 snd_device_ new()本身不会分配设备数据的内存，因此应事先分配。

3．组件释放

每个 ALSA 预定义的组件在构造时需调用 snd_device_new()，而每个组件的析构方法则在函数集中被包含。对于 PCM、AC97 此类预定义组件，我们不需关心它们的析构，而对于自定义的组件，则需要填充 snd_device_ops 中的析构函数指针 dev_free，这样，当 snd_card_free()被调用时，组件将自动被释放。

4．芯片特定的数据（Chip-Specific Data）

芯片特定的数据一般以 struct xxxchip 结构体形式组织，这个结构体中包含芯片相关的 I/O 端口地址、资源指针、中断号等，其意义等同于字符设备驱动中的 file->private_data。

定义芯片特定的数据主要有两种方法，一种方法是将 sizeof(struct xxxchip)传入 snd_card_new()作为 extra_size 参数，它将自动成为 snd_card 的 private_data 成员，如代码清单 17.5 所示；另一种方法是在 snd_card_new()传入给 extra_size 参数 0，再分配 sizeof(struct xxxchip)的内存，将分配内存的地址传入 snd_device_new()的 device_data 的参数，如代码清单 17.6 所示。

代码清单 17.5　创建芯片特定的数据方法 1

```
1 struct xxxchip //芯片特定的数据结构体
2 {
3   ...
4 };
5 card = snd_card_new(index, id, THIS_MODULE, sizeof(struct
6 xxxchip)); //创建声卡并申请 xxx_chi 内存作为 card-> private_data
7 struct xxxchip *chip = card->private_data;
```

代码清单 17.6　创建芯片特定的数据方法 2

```
1  struct snd_card *card;
2  struct xxxchip *chip;
3  //使用 0 作为第 4 个参数，并动态分配 xxx_chip 的内存
4  card = snd_card_new(index[dev], id[dev], THIS_MODULE, 0);
5  ...
6  chip = kzalloc(sizeof(*chip), GFP_KERNEL);
7  //在 xxxchip 结构体中，应该包括声卡指针
8  struct xxxchip
9  {
10   struct snd_card *card;
11   ...
12 };
13 //并将其 card 成员赋值为 snd_card_new()创建的 card 指针
14 chip->card = card;
15 static struct snd_device_ops ops =
16 {
17   . dev_free = snd_xxx_chip_dev_free,  //组件析构
18 };
19 ...
20 //创建自定义组件
21 snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
22 //在析构函数中释放 xxxchip 内存
23 static int snd_xxx_chip_dev_free(struct snd_device *device)
```

```
24 {
25   return snd_xxx_chip_free(device->device_data); //释放
26 }
```

5．注册/释放声卡

当 snd_card 被准备好以后，可使用 snd_card_register()函数注册这个声卡，如下所示：

```
int snd_card_register(struct snd_card *card)
```

对应的 snd_card_free()完成相反的功能，如下所示：

```
int snd_card_free(struct snd_card *card);
```

17.4.3　PCM 设备

每个声卡最多可以有四个 PCM 实例，一个 PCM 实例对应一个设备文件。PCM 实例由 PCM 播放和录音流组成，而每个 PCM 流又由一个或多个 PCM 子流组成。有的声卡支持多重播放功能，例如，emu10k1 包含一个有 32 个立体声子流的 PCM 播放设备。

1．PCM 实例构造

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

第 1 个参数是 card 指针，第 2 个是标识字符串，第 3 个是 PCM 设备索引（0 表示第 1 个 PCM 设备），第 4 和第 5 个分别为播放和录音设备的子流数。当存在多个子流时，需要恰当地处理 open()、close()和其他函数。在每个回调函数中，可以通过 snd_pcm_substream 的 number 成员得知目前操作的究竟是哪个子流，如下所示：

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

一种习惯的做法是在驱动中定义一个 PCM“构造函数”，负责 PCM 实例的创建，如代码清单 17.7 所示。

代码清单 17.7　PCM 设备的“构造函数”

```
1 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
2 {
3   struct snd_pcm *pcm;
4   int err;
5   //创建 PCM 实例
6   if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1,  &pcm)) <
0)
7     return err;
8   pcm->private_data = chip; //置 pcm->private_data 为芯片特定数据
9   strcpy(pcm->name, "xxx Chip");
10  chip->pcm = pcm;
11  ...
```

```
    12  return 0;
    13 }
```

2．设置 PCM 操作

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

第 1 个参数是 snd_pcm 的指针，第 2 个参数是 SNDRV_PCM_STREAM_PLAYBACK 或 SNDRV_ PCM_STREAM_CAPTURE，而第 3 个参数是 PCM 操作结构体 snd_pcm_ops，这个结构体的定义如代码清单 17.8 所示。

代码清单 17.8　snd_pcm_ops 结构体

```
    1 struct snd_pcm_ops
    2 {
    3 int (*open)(struct snd_pcm_substream *substream);//打开
    4 int (*close)(struct snd_pcm_substream *substream);//关闭
    5 int (*ioctl)(struct snd_pcm_substream * substream,
    6              unsigned int cmd, void *arg);//I/O 控制
    7 int (*hw_params)(struct snd_pcm_substream *substream,
    8           struct snd_pcm_hw_params *params);//硬件参数
    9 int (*hw_free)(struct snd_pcm_substream *substream); //资源释放
    10  int (*prepare)(struct snd_pcm_substream *substream);//准备
    11  //在 PCM 被开始、停止或暂停时调用
    12  int (*trigger)(struct snd_pcm_substream *substream, int cmd);
    13  snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);//
当前缓冲区的硬件位置
    14  //缓冲区复制
    15  int (*copy)(struct snd_pcm_substream *substream, int channel,
    16             snd_pcm_uframes_t pos,
    17             void __user *buf, snd_pcm_uframes_t count);
    18  int (*silence)(struct snd_pcm_substream *substream, int channel,
    19                   snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
    20  struct page *(*page)(struct snd_pcm_substream *substream,
    21                  unsigned long offset);
    22     int  (*mmap)(struct  snd_pcm_substream  *substream,  struct
vm_area_struct *vma);
    23  int (*ack)(struct snd_pcm_substream *substream);
    24 };
```

snd_pcm_ops 中的所有操作都需事先通过 snd_pcm_substream_chip()获得 xxxchip 指针，例如：

```
    int xxx()
    {
```

```
    struct xxxchip *chip = snd_pcm_substream_chip(substream);
   ...
 }
```

当一个 PCM 子流被打开时，snd_pcm_ops 中的 open()函数将被调用，在这个函数中，至少需要初始化 runtime->hw 字段，代码清单 17.9 所示为 open()函数的范例。

代码清单 17.9　snd_pcm_ops 结构体中的 open()函数

```
1  static int snd_xxx_open(struct snd_pcm_substream *substream)
2  {
3    //从子流获得 xxxchip 指针
4    struct xxxchip *chip = snd_pcm_substream_chip(substream);
5    //获得 PCM 运行时信息指针
6    struct snd_pcm_runtime *runtime = substream->runtime;
7    ...
8    //初始化 runtime->hw
9    runtime->hw = snd_xxxchip_playback_hw;
10   return 0;
11 }
```

上述代码中的 snd_xxxchip_playback_hw 是预先定义的硬件描述。在 open()函数中，可以分配一段私有数据。如果硬件配置需要更多的限制，也需设置硬件限制。

当 PCM 子流被关闭时，close()函数将被调用。如果 open()函数中分配了私有数据，则在 close()函数中应该释放 substream 的私有数据，代码清单 17.10 所示为 close()函数的范例。

代码清单 17.10　snd_pcm_ops 结构体中的 close()函数

```
1 static int snd_xxx_close(struct snd_pcm_substream *substream)
2 {
3   //释放子流私有数据
4   kfree(substream->runtime->private_data);
5   //...
6 }
```

驱动中通常可以给 snd_pcm_ops 的 ioctl()成员函数传递通用的 snd_pcm_lib_ioctl()函数。

snd_pcm_ops 的 hw_params()成员函数将在应用程序设置硬件参数（PCM 子流的周期大小、缓冲区大小和格式等）的时候被调用，它的形式如下：

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

在这个函数中，将完成大量硬件设置，甚至包括缓冲区分配，这时可调用如下辅

助函数：

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

仅当 DMA 缓冲区已被预先分配的情况下，上述调用才可成立。

与 hw_params()对应的函数是 hw_free()，它释放由 hw_params()分配的资源，例如，通过如下调用释放 snd_pcm_lib_malloc_pages()缓冲区：

```
snd_pcm_lib_free_pages(substream);
```

当 PCM 被“准备”时，prepare()函数将被调用，在其中可以设置采样率、格式等。prepare()函数与 hw_params()函数的不同在于对 prepare()的调用发生在 snd_pcm_prepare()每次被调用的时候。prepare()的形式如下：

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

trigger()成员函数在 PCM 被开始、停止或暂停时调用，函数的形式如下：

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

cmd 参数定义了具体的行为，在 trigger()成员函数中至少要处理 SNDRV_PCM_TRIGGER_ START 和 SNDRV_PCM_TRIGGER_STOP 命令，如果 PCM 支持暂停，还应处理 SNDRV_PCM_ TRIGGER_PAUSE_PUSH 和 SNDRV_PCM_TRIGGER_PAUSE_RELEASE 命令。如果设备支持挂起/恢复，当能量管理状态发生变化时将处理 SNDRV_PCM_TRIGGER_SUSPEND 和 SNDRV_PCM_ TRIGGER_RESUME 这两个命令。注意 trigger()函数是原子的，中途不能睡眠。代码清单 17.11 所示为 trigger()函数的范例。

代码清单 17.11　snd_pcm_ops 结构体中的 trigger()函数

```
1  static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd)
2  {
3    switch (cmd)
4    {
5        case SNDRV_PCM_TRIGGER_START:
6        // 开启 PCM 引擎
7        break;
8      case SNDRV_PCM_TRIGGER_STOP:
9        // 停止 PCM 引擎
10       break;
11     ...//其他命令
12     default:
13       return  - EINVAL;
14   }
15 }
```

pointer()函数用于 PCM 中间层查询目前缓冲区的硬件位置，该函数以帧的形式返回 0～buffer_size – 1 的位置（ALSA 0.5.x 中为字节形式），此函数也是原子的。

copy()和 silence()函数一般可以省略，但是，当硬件缓冲区不处于常规内存中时需

要。例如，一些设备有自己的不能被映射的硬件缓冲区，这种情况下，我们不得不将数据从内存缓冲区复制到硬件缓冲区。当内存缓冲区在物理和虚拟地址上都不连续时，这两个函数也必须被实现。

3．分配缓冲区

分配缓冲区的最简单方法是调用如下函数：

```
int snd_pcm_lib_preallocate_pages_for_all(struct snd_pcm *pcm,
                              int type, void *data, size_t size, size_t max);
```

type 参数是缓冲区的类型，包含 SNDRV_DMA_TYPE_UNKNOWN（未知）、SNDRV_DMA_ TYPE_CONTINUOUS（连续的非 DMA 内存）、SNDRV_DMA_TYPE_DEV（连续的通用设备），SNDRV_DMA_TYPE_DEV_SG（通用设备 SG-buffer）和 SNDRV_DMA_TYPE_SBUS（连续的 SBUS）。如下代码将分配 64KB 的缓冲区：

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

4．设置标志

在构造 PCM 实例、设置操作集并分配缓冲区之后，如果有需要，应设置 PCM 的信息标志，例如，如果 PCM 设备只支持半双工，则这样定义标志：

```
pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;
```

5．PCM 实例析构

PCM 实例的“析构函数”并非是必须的，因为 PCM 实例会被 PCM 中间层代码自动释放，如果驱动中分配了一些特别的内存空间，则必须定义“析构函数”，代码清单 17.12 所示为 PCM“析构函数”与对应的“构造函数”，“析构函数”会释放“构造函数”中创建的 xxx_private_pcm_data。

代码清单 17.12　PCM 设备“析构函数”

```
1  static void xxxchip_pcm_free(struct snd_pcm *pcm)
2  {
3    /* 从 pcm 实例得到 chip */
4    struct xxxchip *chip = snd_pcm_chip(pcm);
5    /* 释放自定义用途的内存 */
6    kfree(chip->xxx_private_pcm_data);
7    ...
8  }
9
10 static int __devinit snd_xxxchip_new_pcm(struct xxxchip
```

```
*chip)
   11 {
   12   struct snd_pcm *pcm;
   13     ...
   14     /* 分配自定义用途的内存 */
   15     chip->xxx_private_pcm_data = kmalloc(...);
   16     pcm->private_data = chip;
   17     /* 设置“析构函数” */
   18     pcm->private_free = xxxchip_pcm_free;
   19     ...
   20 }
```

上述代码第 4 行的 snd_pcm_chip()从 PCM 实例指针获得 xxxchip 指针，实际上它就是返回第 16 行给 PCM 实例赋予的 xxxchip 指针。

6．PCM 信息运行时结构体

当 PCM 子流被打开后，PCM 运行时实例（定义为结构体 snd_pcm_runtime，如代码清单 17.13 所示）将被分配给这个子流，这个指针通过 substream->runtime 获得。运行时指针包含各种各样的信息：hw_params 及 sw_params 配置的拷贝、缓冲区指针、mmap 记录、自旋锁等，几乎 PCM 的所有控制信息均能从中取得。

代码清单 17.13　snd_pcm_runtime 结构体

```
1  struct snd_pcm_runtime
2  {
3    /* 状态 */
4    struct snd_pcm_substream *trigger_master;
5    snd_timestamp_t trigger_tstamp; /* 触发时间戳 */
6    int overrange;
7    snd_pcm_uframes_t avail_max;
8    snd_pcm_uframes_t hw_ptr_base; /* 缓冲区复位时的位置 */
9    snd_pcm_uframes_t hw_ptr_interrupt; /* 中断时的位置*/
10   /* 硬件参数 */
11   snd_pcm_access_t access; /* 存取模式 */
12   snd_pcm_format_t format; /* SNDRV_PCM_FORMAT_* */
13   snd_pcm_subformat_t subformat; /* 子格式 */
14   unsigned int rate; /* rate in Hz */
15   unsigned int channels; /* 通道 */
16   snd_pcm_uframes_t period_size; /* 周期大小 */
17   unsigned int periods; /* 周期数 */
18   snd_pcm_uframes_t buffer_size; /* 缓冲区大小 */
19   unsigned int tick_time; /* tick time */
20   snd_pcm_uframes_t min_align; /* 格式对应的最小对齐*/
21   size_t byte_align;
22   unsigned int frame_bits;
23   unsigned int sample_bits;
24   unsigned int info;
25   unsigned int rate_num;
26   unsigned int rate_den;
27   /* 软件参数 */
28   struct timespec tstamp_mode; /* mmap 时间戳被更新*/
```

```
    29  unsigned int period_step;
    30  unsigned int sleep_min; /* 睡眠的最小节拍 */
    31  snd_pcm_uframes_t xfer_align;
    32  snd_pcm_uframes_t start_threshold;
    33  snd_pcm_uframes_t stop_threshold;
    34  snd_pcm_uframes_t silence_threshold; /* Silence 填充阈值 */
    35  snd_pcm_uframes_t silence_size; /* Silence 填充大小 */
    36  snd_pcm_uframes_t boundary;
    37  snd_pcm_uframes_t silenced_start;
    38  snd_pcm_uframes_t silenced_size;
    39  snd_pcm_sync_id_t sync; /* 硬件同步 ID */
    40  /* mmap  */
    41  volatile struct snd_pcm_mmap_status *status;
    42  volatile struct snd_pcm_mmap_control *control;
    43  atomic_t mmap_count;
    44  /* 锁/调度 */
    45  spinlock_t lock;
    46  wait_queue_head_t sleep;
    47  struct timer_list tick_timer;
    48  struct fasync_struct *fasync;
    49  /* 私有段 */
    50  void *private_data;
    51  void(*private_free)(struct snd_pcm_runtime *runtime);
    52  /* 硬件描述 */
    53  struct snd_pcm_hardware hw;
    54  struct snd_pcm_hw_constraints hw_constraints;
    55  /* 中断回调函数 */
    56  void(*transfer_ack_begin)(struct snd_pcm_substream*substream);
    57  void(*transfer_ack_end)(struct snd_pcm_substream *substream);
    58  /* 定时器 */
    59  unsigned int timer_resolution; /* timer resolution */
    60  /* DMA */
    61  unsigned char *dma_area; /* DMA 区域*/
    62  dma_addr_t dma_addr; /* 总线物理地址*/
    64  size_t dma_bytes; /* DMA 区域大小 */
    65  struct snd_dma_buffer *dma_buffer_p; /* 被分配的缓冲区 */
    66                      #if       defined(CONFIG_SND_PCM_OSS)        ||
defined(CONFIG_SND_PCM_OSS_MODULE)
    67    /* OSS 信息 */
    68    struct snd_pcm_oss_runtime oss;
    69  #endif
    70 };
```

snd_pcm_runtime 中的大多数记录对被声卡驱动操作集中的函数是只读的，仅仅 PCM 中间层可从更新或修改这些信息，但是硬件描述、中断回调函数、DMA 缓冲区信息和私有数据是例外的。

下面解释 snd_pcm_runtime 结构体中的几个重要成员。

（1）硬件描述。

硬件描述（snd_pcm_hardware 结构体）包含了基本硬件配置的定义，需要在 open()函数中赋值。runtime 实例保存的是硬件描述的拷贝而非指针，这意味着在 open()函数中可以修改被拷贝的描述（runtime->hw），例如：

```
struct snd_pcm_runtime *runtime = substream->runtime;
...
runtime->hw = snd_xxchip_playback_hw; /* “大众”硬件描述 */
/* 特定的硬件描述 */
if (chip->model == VERY_OLD_ONE)
runtime->hw.channels_max = 1;
```

snd_pcm_hardware 结构体的定义如代码清单 17.14 所示。

代码清单 17.14　snd_pcm_hardware 结构体

```
1  struct snd_pcm_hardware
2  {
3  unsigned int info;          /* SNDRV_PCM_INFO_* /
4  u64 formats;                /* SNDRV_PCM_FMTBIT_* */
5  unsigned int rates;         /* SNDRV_PCM_RATE_* */
6  unsigned int rate_min;         /* 最小采样率 */
7  unsigned int rate_max;         /* 最大采样率 */
8  unsigned int channels_min;     /* 最小的通道数 */
9  unsigned int channels_max;     /* 最大的通道数 */
10  size_t buffer_bytes_max;      /* 最大缓冲区大小 */
11  size_t period_bytes_min;      /* 最小周期大小 */
12  size_t period_bytes_max;      /* 最大奏曲大小 */
13  unsigned int periods_min;     /* 最小周期数 */
14  unsigned int periods_max;     /* 最大周期数 */
15  size_t fifo_size;             /* FIFO 字节数 */
16 };
```

snd_pcm_hardware 结构体中的 info 字段标识 PCM 设备的类型和能力，形式为 SNDRV_PCM_ INFO_XXX。info 字段至少需要定义是否支持 mmap，当支持时，应设置 SNDRV_PCM_INFO_MMAP 标志；当硬件支持 interleaved 或 non-interleaved 格式时，应设置 SNDRV_PCM_INFO _INTERLEAVED 或 SNDRV_PCM_INFO_NONINTERLEAVED 标志；如果都支持，则两者都可设置。

MMAP_VALID 和 BLOCK_TRANSFER 标志针对 OSS mmap，只有 mmap 被真正支持时，才可设置 MMAP_VALID；SNDRV_PCM_INFO_PAUSE 意味着设备可支持暂停操作，而 SNDRV_ PCM_INFO_RESUME 意味着设备可支持挂起/恢复操作；当 PCM 子流能被同步，如同步播放和录音流的 start/stop，可设置 SNDRV_PCM_INFO_SYNC_START 标志。

formats 包含 PCM 设备支持的格式，形式为 SNDRV_PCM_FMTBIT_XXX，如果设备支持多种模式，应将各种模式标志进行“或”操作。

rates 包含了 PCM 设备支持的采样率，形式如 SNDRV_PCM_RATE_XXX，如果支持连续的采样率，则传递 CONTINUOUS。

rate_min 和 rate_max 分别定义了最大和最小的采样率，注意：要与 rates 字段相符。

channel_min 和 channel_max 定义了最大和最小的通道数量。

buffer_bytes_max 定义最大的缓冲区大小，注意：没有 buffer_bytes_min 字段，这

是因为它可以通过最小的周期大小和最小的周期数量计算出来。

period 信息与 OSS 中的 fragment 对应，定义了 PCM 中断产生的周期。更小的周期大小意味着更多的中断，在录音时，周期大小定义了输入延迟，在播放时，整个缓冲区大小对应着输出延迟。

PCM 可被应用程序通过 alsa-lib 发送 hw_params 来配置，配置信息将保存在运行时实例中。对缓冲区和周期大小的配置以帧形式存储，而 frames_to_bytes()和 bytes_to_frames()可完成帧和字节的转换，如：

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

（2）DMA 缓冲区信息。

包含 dma_area（逻辑地址）、dma_addr（物理地址）、dma_bytes（缓冲区大小）和 dma_private（被 ALSA DMA 分配器使用）。可以由 snd_pcm_lib_malloc_pages()实现，ALSA 中间层会设置 DMA 缓冲区信息的相关字段，这种情况下，驱动中不能再写这些信息，只能读取。也就是说，如果使用标准的缓冲区分配函数 snd_pcm_lib_malloc_pages()分配缓冲区，则我们不需要自己维护 DMA 缓冲区信息。如果缓冲区由自己分配，则需要在 hw_params()函数中管理缓冲区信息，至少需管理 dma_bytes 和 dma_addr，如果支持 mmap，则必须管理 dma_area，对 dma_private 的管理视情况而定。

（3）运行状态。

通过 runtime->status 可以获得运行状态，它是 snd_pcm_mmap_status 结构体的指针，例如，通过 runtime->status->hw_ptr 可以获得目前的 DMA 硬件指针。此外，通过 runtime->control 可以获得 DMA 应用指针，它指向 snd_pcm_mmap_control 结构体指针。

（4）私有数据。

驱动中可以为子流分配一段内存并赋值给 runtime->private_data，注意不要与 pcm->private_ data 混淆，后者一般指向 xxxchip，而前者是在 PCM 设备的 open()函数中分配的动态数据，例如：

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
  struct xxx_pcm_data *data;
  ....
  data = kmalloc(sizeof(*data), GFP_KERNEL);
  substream->runtime->private_data = data; //赋值 runtime->private_data
  ....
}
```

（5）中断回调函数：

transfer_ack_begin()和 transfer_ack_end()函数分别在 snd_pcm_period_elapsed()的开始和结束时被调用。

根据以上分析，代码清单 17.15 给出了一个完整的 PCM 设备接口模板。

代码清单 17.15　PCM 设备接口模板

```
1   #include <sound/pcm.h>
2   ....
3   /* 播放设备硬件定义 */
4   static struct snd_pcm_hardware snd_xxxchip_playback_hw =
5   {
6     .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
7       SNDRV_PCM_INFO_BLOCK_TRANSFER | SNDRV_PCM_INFO_MMAP_VALID),
8     .formats = SNDRV_PCM_FMTBIT_S16_LE,
9     .rates = SNDRV_PCM_RATE_8000_48000,
10    .rate_min = 8000,
11    .rate_max = 48000,
12    .channels_min = 2,
13    .channels_max = 2,
14    .buffer_bytes_max = 32768,
15    .period_bytes_min = 4096,
16    .period_bytes_max = 32768,
17    .periods_min = 1,
18    .periods_max = 1024,
19  };
20
21  /* 录音设备硬件定义 */
22  static struct snd_pcm_hardware snd_xxxchip_capture_hw =
23  {
24    .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
25                                  SNDRV_PCM_INFO_BLOCK_TRANSFER    |
SNDRV_PCM_INFO_MMAP_VALID),
26    .formats = SNDRV_PCM_FMTBIT_S16_LE,
27    .rates = SNDRV_PCM_RATE_8000_48000,
28    .rate_min = 8000,
29    .rate_max = 48000,
30    .channels_min = 2,
31    .channels_max = 2,
32    .buffer_bytes_max = 32768,
33    .period_bytes_min = 4096,
34    .period_bytes_max = 32768,
35    .periods_min = 1,
36    .periods_max = 1024,
37  };
38
39  /* 播放：打开函数 */
40            static      int       snd_xxxchip_playback_open(struct
snd_pcm_substream*substream)
41  {
42    struct xxxchip *chip = snd_pcm_substream_chip(substream);
43    struct snd_pcm_runtime *runtime = substream->runtime;
```

```
    44   runtime->hw = snd_xxxchip_playback_hw;
    45   ... // 硬件初始化代码
    46   return 0;
    47  }
    48
    49  /* 播放：关闭函数 */
    50          static      int      snd_xxxchip_playback_close(struct
snd_pcm_substream*substream)
    51  {
    52   struct xxxchip *chip = snd_pcm_substream_chip(substream);
    53   // 硬件相关的代码
    54   return 0;
    55  }
    56
    57  /* 录音：打开函数 */
    58          static      int      snd_xxxchip_capture_open(struct
snd_pcm_substream*substream)
    59  {
    60   struct xxxchip *chip = snd_pcm_substream_chip(substream);
    61   struct snd_pcm_runtime *runtime = substream->runtime;
    62   runtime->hw = snd_xxxchip_capture_hw;
    63   ... // 硬件初始化代码
    64   return 0;
    65  }
    66
    67  /* 录音：关闭函数 */
    68          static      int      snd_xxxchip_capture_close(struct
snd_pcm_substream*substream)
    69  {
    70   struct xxxchip *chip = snd_pcm_substream_chip(substream);
    71   ... // 硬件相关的代码
    72   return 0;
    73  }
    74  /* hw_params 函数 */
    75          static      int      snd_xxxchip_pcm_hw_params(struct
snd_pcm_substream*substream, struct
    76   snd_pcm_hw_params *hw_params)
    77  {
```

```
78                    return     snd_pcm_lib_malloc_pages(substream,
params_buffer_bytes(hw_params));
79  }
80  /* hw_free 函数 */
81             static      int       snd_xxxchip_pcm_hw_free(struct
snd_pcm_substream*substream)
82  {
83   return snd_pcm_lib_free_pages(substream);
84  }
85  /* prepare 函数 */
86             static      int       snd_xxxchip_pcm_prepare(struct
snd_pcm_substream*substream)
87  {
88   struct xxxchip *chip = snd_pcm_substream_chip(substream);
89   struct snd_pcm_runtime *runtime = substream->runtime;
90   /* 根据目前的配置信息设置硬件
91    * 例如:
92    */
93   xxxchip_set_sample_format(chip, runtime->format);
94   xxxchip_set_sample_rate(chip, runtime->rate);
95   xxxchip_set_channels(chip, runtime->channels);
96                xxxchip_set_dma_setup(chip,    runtime->dma_addr,
chip->buffer_size, chip
97    ->period_size);
98   return 0;
99  }
100 /* trigger 函数 */
101         static        int        snd_xxxchip_pcm_trigger(struct
snd_pcm_substream*substream, int cmd)
102 {
103   switch (cmd)
104   {
105     case SNDRV_PCM_TRIGGER_START:
106       // do something to start the PCM engine
107       break;
108     case SNDRV_PCM_TRIGGER_STOP:
109       // do something to stop the PCM engine
110       break;
111     default:
112       return  - EINVAL;
```

```
113   }
114 }
115
116 /* pointer 函数 */
117   static   snd_pcm_uframes_t   snd_xxxchip_pcm_pointer(struct
snd_pcm_substream
118   *substream)
119 {
120   struct xxxchip *chip = snd_pcm_substream_chip(substream);
121   unsigned int current_ptr;
122   /*获得当前的硬件指针*/
123   current_ptr = xxxchip_get_hw_pointer(chip);
124   return current_ptr;
125 }
126 /* 放音设备操作集 */
127 static struct snd_pcm_ops snd_xxxchip_playback_ops =
128 {
129   .open = snd_xxxchip_playback_open,
130   .close = snd_xxxchip_playback_close,
131   .ioctl = snd_pcm_lib_ioctl,
132   .hw_params = snd_xxxchip_pcm_hw_params,
133   .hw_free = snd_xxxchip_pcm_hw_free,
134   .prepare = snd_xxxchip_pcm_prepare,
135   .trigger = snd_xxxchip_pcm_trigger,
136   .pointer = snd_xxxchip_pcm_pointer,
137 };
138 /* 录音设备操作集 */
139 static struct snd_pcm_ops snd_xxxchip_capture_ops =
140 {
141   .open = snd_xxxchip_capture_open,
142   .close = snd_xxxchip_capture_close,
143   .ioctl = snd_pcm_lib_ioctl,
144   .hw_params = snd_xxxchip_pcm_hw_params,
145   .hw_free = snd_xxxchip_pcm_hw_free,
146   .prepare = snd_xxxchip_pcm_prepare,
147   .trigger = snd_xxxchip_pcm_trigger,
148   .pointer = snd_xxxchip_pcm_pointer,
149 };
150
151 /* 创建一个 PCM 设备 */
152 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
```

```
    153 {
    154   struct snd_pcm *pcm;
    155   int err;
    156   if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1, &pcm)) <
0)
    157       return err;
    158   pcm->private_data = chip;
    159   strcpy(pcm->name, "xxx Chip");
    160   chip->pcm = pcm;
    161   /* 设置操作集 */
    162           snd_pcm_set_ops(pcm,      SNDRV_PCM_STREAM_PLAYBACK,
&snd_xxxchip_playback_ops);
    163           snd_pcm_set_ops(pcm,      SNDRV_PCM_STREAM_CAPTURE,
&snd_xxxchip_capture_ops);
    164    /* 分配缓冲区 */
    165                   snd_pcm_lib_preallocate_pages_for_all(pcm,
SNDRV_DMA_TYPE_DEV,
    166      snd_dma_pci_data(chip -  > pci), 64 *1024, 64 *1024);
    167   return 0;
    168 }
```

17.4.4 控制接口

1．control

控制接口对于许多开关（switch）和调节器（slider）而言应用相当广泛，它能从用户空间被存取。control 的最主要用途是 mixer，所有的 mixer 元素基于 control 内核 API 实现，在 ALSA 中，control 用 snd_kcontrol 结构体描述。

ALSA 有一个定义很好的 AC97 控制模块，对于仅支持 AC97 的芯片而言，不必实现本节的内容。

创建一个新的 control 至少需要实现 snd_kcontrol_new 中的 info()、get()和 put()这 3 个成员函数，snd_kcontrol_new 结构体的定义如代码清单 17.16 所示。

代码清单 17.16　snd_kcontrol_new 结构体

```
1  struct snd_kcontrol_new
2  {
3    snd_ctl_elem_iface_t iface; /*接口 ID，SNDRV_CTL_ELEM_IFACE_XXX */
4    unsigned int device; /* 设备号 */
5    unsigned int subdevice; /* 子流（子设备）号 */
6    unsigned char *name; /* 名称(ASCII 格式) */
7    unsigned int index; /* 索引 */
8    unsigned int access; /* 访问权限 */
9    unsigned int count; /* 享用元素的数量 */
10   snd_kcontrol_info_t *info;
11   snd_kcontrol_get_t *get;
```

```
12    snd_kcontrol_put_t *put;
13    unsigned long private_value;
14 };
```

iface 字段定义了 control 的类型，形式为 SNDRV_CTL_ELEM_IFACE_XXX，通常是 MIXER，对于不属于 mixer 的全局控制，使用 CARD。如果关联于某类设备，则使用 HWDEP、PCM、RAWMIDI、TIMER 或 SEQUENCER。

name 是名称标识字符串，control 的名称非常重要，因为 control 的作用由名称来区分。对于名称相同的 control，则使用 index 区分。name 定义的标准是“SOURCE DIRECTION FUNCTION”即“源方向功能”，SOURCE 定义了 control 的源，如“Master”、“PCM”、“CD”和“Line”，方向则为“Playback”、“Capture”、“Bypass Playback”或“Bypass Capture”，如果方向省略，意味着 playback 和 capture 双向，第 3 个参数可以是“Switch”、“Volume”和“Route”等。

“SOURCE DIRECTION FUNCTION”格式的名称例子如 Master Capture Switch、PCM Playback Volume。

下面几种 control 的命名不采用“SOURCE DIRECTION FUNCTION”格式，属于例外。

（1）全局控制。

“Capture Source”、“Capture Switch”和“Capture Volume”用于全局录音源、输入开关和录音音量控制；“Playback Switch”、“Playback Volume”用于全局输出开关和音量控制。

（2）音调控制。

音调控制名称的形式为“Tone Control – XXX”，例如“Tone Control – Switch”、“Tone Control – Bas”和“Tone Control – Center”。

（3）3D 控制。

3D 控制名称的形式为“3D Control – XXX”，例如“3D Control – Switch”、“3D Control – Center”和“3D Control – Space”。

（4）麦克风增益（Mic boost）。

麦克风增益被设置为“Mic Boost”或“Mic Boost (6dB)”。

snd_kcontrol_new 结构体的 access 字段是访问控制权限，形式如 SNDRV_CTL_ELEM_ ACCESS_XXX。SNDRV_CTL_ELEM_ACCESS_READ 意味着只读，这时 put()函数不必实现；SNDRV_CTL_ELEM_ACCESS_WRITE 意味着只写，这时 get()函数不必实现。若 control 值频繁变化，则需定义 VOLATILE 标志。当 control 处于非激活状态时，应设置 INACTIVE 标志。

private_value 字段包含一个长整型值，可以通过它给 info()、get()和 put()函数传递参数。

2．info()函数

snd_kcontrol_new 结构体中的 info()函数用于获得该 control 的详细信息，该函数必须填充传递给它的第二个参数 snd_ctl_elem_info 结构体，info()函数的形式如下：

```
static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo);
```

snd_ctl_elem_info 结构体的定义如代码清单 17.17 所示。

代码清单 17.17 snd_ctl_elem_info 结构体

```
1  struct snd_ctl_elem_info
2  {
3     struct snd_ctl_elem_id id; /* W: 元素 ID */
4     snd_ctl_elem_type_t type; /* R: 值类型 - SNDRV_CTL_ELEM_TYPE_* */
5          unsigned int access; /* R: 值访问权限(位掩码) - SNDRV_CTL_ELEM_ACCESS_* */
6     unsigned int count; /* 值的计数 */
7     pid_t owner; /* 该 control 的拥有者 PID */
8     union
9     {
10      struct
11      {
12         long min; /* R: 最小值 */
13         long max; /* R: 最大值 */
14         long step; /* R: 值步进 (0 可变的) */
15      } integer;
16      struct
17      {
18         long long min; /* R: 最小值 */
19         long long max; /* R: 最大值 */
20         long long step; /* R: 值步进 (0 可变的) */
21      } integer64;
22      struct
23      {
24         unsigned int items; /* R: 项目数 */
25         unsigned int item; /* W: 项目号 */
26         char name[64]; /* R: 值名称 */
27      } enumerated; /* 枚举 */
28      unsigned char reserved[128];
29    }
30    value;
31    union
32    {
33      unsigned short d[4];
34      unsigned short *d_ptr;
35    } dimen;
36    unsigned char reserved[64-4 * sizeof(unsigned short)];
37 };
```

snd_ctl_elem_info 结构体的 type 字段定义了 control 的类型，包括 BOOLEAN、INTEGER、ENUMERATED、BYTES、IEC958 和 INTEGER64。count 字段定义了这个 control 中包含的元素的数量，例如一个立体声音量 control 的 count = 2。value 是一个联合体，其所存储的值的具体类型依赖于 type。代码清单 17.18 所示为一个 info()函数填充 snd_ctl_elem_info 结构体的范例。

代码清单 17.18 snd_ctl_elem_info 结构体中的 info()函数范例

```
1 static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_info *uinfo)
3 {
4     uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;//类型为 BOOLEAN
5     uinfo->count = 1;//数量为 1
6     uinfo->value.integer.min = 0;//最小值为 0
7     uinfo->value.integer.max = 1;//最大值为 1
8     return 0;
9 }
```

枚举类型和其他类型略有不同，对枚举类型，应为目前项目索引设置名称字符串，如代码清单 17.19 所示。

代码清单 17.19 填充 snd_ctl_elem_info 结构体中的权举类型值

```
1  static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2      snd_ctl_elem_info *uinfo)
3  {
4      //值名称字符串
5      static char *texts[4] =
6     {
7       "First", "Second", "Third", "Fourth"
8     };
9      uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;//枚举类型
10     uinfo->count = 1;//数量为 1
11     uinfo->value.enumerated.items = 4;//项目数量为 1
12     //超过 3 的项目号改为 3
13     if (uinfo->value.enumerated.item > 3)
14       uinfo->value.enumerated.item = 3;
15     //为目前项目索引复制名称字符串
16                               strcpy(uinfo->value.enumerated.name,
texts[uinfo->value.enumerated.item]);
17     return 0;
18 }
```

3. get()函数

get()函数用于得到 control 的目前值并返回用户空间，代码清单 17.20 所示为 get()函数的范例。

代码清单 17.20 snd_ctl_elem_info 结构体中的 get()函数范例

```
1 static int snd_xxxctl_get(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_value *ucontrol)
3 {
4     //从 snd_kcontrol 获得 xxxchip 指针
5     struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
6     //从 xxxchip 获得值并写入 snd_ctl_elem_value
7     ucontrol->value.integer.value[0] = get_some_value(chip);
8     return 0;
9 }
```

get()函数的第二个参数的类型为 snd_ctl_elem_value，其定义如代码清单 10.21 所示。snd_ctl_ elem_ value 结构体的内部也包含一个由 integer、integer64、enumerated 等组成的值联合体，它的具体类型依赖于 control 的类型和 info()函数。

代码清单 17.21 snd_ctl_elem_value 结构体

```
1  struct snd_ctl_elem_value
2  {
3     struct snd_ctl_elem_id id; /* W: 元素 ID */
4     unsigned int indirect: 1; /* W: 使用间接指针(xxx_ptr 成员) */
5     //值联合体
6     union
7     {
8        union
9       {
10         long value[128];
11         long *value_ptr;
12      } integer;
13      union
14      {
15         long long value[64];
16         long long *value_ptr;
17      } integer64;
18      union
19      {
20         unsigned int item[128];
21         unsigned int *item_ptr;
22      } enumerated;
23      union
24      {
25         unsigned char data[512];
26         unsigned char *data_ptr;
27       } bytes;
28       struct snd_aes_iec958 iec958;
29    }
30    value; /* 只读 */
31    struct timespec tstamp;
32    unsigned char reserved[128-sizeof(struct timespec)];
33 };
```

4．put()函数

put()用于从用户空间写入值，如果值被改变，该函数返回 1，否则返回 0；如果发生错误，该函数返回一个错误码。代码清单 17.22 所示为一个 put()函数的范例。

代码清单 17.22　snd_ctl_elem_info 结构体中的 put()函数范例

```
    1 static int snd_xxxctl_put(struct snd_kcontrol *kcontrol, struct
    2     snd_ctl_elem_value *ucontrol)
    3 {
    4     //从 snd_kcontrol 获得 xxxchip 指针
    5     struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
    6     int changed = 0;//默认返回值为 0
    7     //值被改变
    8     if (chip->current_value != ucontrol->value.integer.value[0])
    9     {
    10                                              change_current_value(chip,
ucontrol->value.integer.value[0]);
    11        changed = 1;//返回值为 1
    12    }
    13    return changed;
    14  }
```

对于 get()和 put()函数而言，如果 control 有多于一个元素，即 count >1，则每个元素都需要被返回或写入。

5．构造 control

当所有事情准备好后，我们需要创建一个 control，调用 snd_ctl_add()和 snd_ctl_new1()这两个函数来完成，这两个函数的原型为：

```
    int snd_ctl_add(struct snd_card *card, struct snd_kcontrol *kcontrol);


    struct  snd_kcontrol  *snd_ctl_new1(const  struct  snd_kcontrol_new
*ncontrol,
                              void *private_data);
```

snd_ctl_new1()函数用于创建一个 snd_kcontrol 并返回其指针，snd_ctl_add()函数用于将创建的 snd_kcontrol 添加到对应的 card 中。

6．变更通知

如果驱动中需要在中断服务程序中改变或更新一个 control，可以调用 snd_ctl_notify()函数，此函数原型为：

```
    void snd_ctl_notify(struct snd_card *card, unsigned int mask, struct
snd_ctl_elem_id *id);
```

该函数的第二个参数为事件掩码（event-mask），第三个参数为该通知的 control 元素 id 指针。

例如，如下语句定义的事件掩码 SNDRV_CTL_EVENT_MASK_VALUE 意味着

control 值的改变被通知：

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

17.4.5 AC97 API 接口

ALSA AC97 编解码层被很好地定义，利用它，驱动工程师只需编写少量底层的控制函数。

1. AC97 实例构造

为了创建一个 AC97 实例，首先需要调用 snd_ac97_bus()函数构建 AC97 总线及其操作，这个函数的原型为：

```
int snd_ac97_bus(struct snd_card *card, int num, struct snd_ac97_bus_ops *ops,
                void *private_data, struct snd_ac97_bus **rbus);
```

该函数的第 3 个参数 ops 是一个 snd_ac97_bus_ops 结构体，其定义如代码清单 17.23 所示。

代码清单 17.23 snd_ac97_bus_ops 结构体

```
1 struct snd_ac97_bus_ops
2 {
3    void(*reset)(struct snd_ac97 *ac97); //复位函数
4    //写入函数
5    void(*write)(struct snd_ac97 *ac97, unsigned short reg, unsigned short val);
6    //读取函数
7    unsigned short(*read)(struct snd_ac97 *ac97, unsigned short reg);
8    void(*wait)(struct snd_ac97 *ac97);
9    void(*init)(struct snd_ac97 *ac97);
10 };
```

接下来，调用 snd_ac97_mixer()函数注册混音器，这个函数的原型为：

```
int snd_ac97_mixer(struct snd_ac97_bus *bus, struct snd_ac97_template *template, struct snd_ac97 **rac97);
```

代码清单 17.24 所示为 AC97 实例的创建过程。

代码清单 17.24 AC97 实例的创建过程范例

```
1 struct snd_ac97_bus *bus;
2 //AC97 总线操作
3 static struct snd_ac97_bus_ops ops =
4 {
5   .write = snd_mychip_ac97_write,
6   .read = snd_mychip_ac97_read,
7   };
8   //AC97 总线与操作创建
9   snd_ac97_bus(card, 0, &ops, NULL, &bus);
```

```
10 //AC97 模板
11 struct snd_ac97_template ac97;
12 int err;
13 memset(&ac97, 0, sizeof(ac97));
14 ac97.private_data = chip;//私有数据
15 //注册混音器
16 snd_ac97_mixer(bus, &ac97, &chip->ac97);
```

上述代码第一行的 snd_ac97_bus 结构体指针 bus 的指针被传入第 9 行的 snd_ac97_bus()函数并被赋值，chip->ac97 的指针被传入第 16 行的 snd_ac97_mixer()并被赋值，chip->ac97 将成员新创建 AC97 实例的指针。

如果一个声卡上包含多个编解码器，这种情况下，需要多次调用 snd_ac97_mixer()并对 snd_ac97 的 num 成员（编解码器序号）赋予相应的序号。驱动中可以为不同的编解码器编写不同的 snd_ac97_bus_ops 成员函数中，或者只是在相同的一套成员函数中通过 ac97.num 获得序号后再区分进行具体的操作。

2．snd_ac97_bus_ops 成员函数

snd_ac97_bus_ops 结构体中的 read()和 write()成员函数完成底层的硬件访问，reset()函数用于复位编解码器，wait()函数用于编解码器标准初始化过程中的特定等待，如果芯片要求额外的等待时间，则应实现这个函数，init()用于完成编解码器附加的初始化。代码清单 17.25 所示为 read()和 write()函数的范例。

代码清单 17.25　snd_ac97_bus_ops 结构体中的 read()和 write()函数范例

```
 1 static unsigned short snd_xxxchip_ac97_read(struct snd_ac97 *ac97, unsigned
 2     short reg)
 3 {
 4     struct xxxchip *chip = ac97->private_data;
 5     ...
 6     return the_register_value; //返回寄存器值
 7 }
 8
 9 static void snd_xxxchip_ac97_write(struct snd_ac97 *ac97, unsigned short reg,
10     unsigned short val)
11 {
12     struct xxxchip *chip = ac97->private_data;
13     ...
14     // 将被给的寄存器值写入 codec
15 }
```

3．修改寄存器

如果需要在驱动中访问编解码器，可使用如下函数：

```
    void snd_ac97_write(struct snd_ac97 *ac97, unsigned short reg, unsigned
short value);

    int snd_ac97_update(struct snd_ac97 *ac97, unsigned short reg, unsigned
short value);

    int snd_ac97_update_bits(struct snd_ac97 *ac97, unsigned short reg,
unsigned short mask, unsigned short value);

    unsigned short snd_ac97_read(struct snd_ac97 *ac97, unsigned short
reg);
```

snd_ac97_update()与 void snd_ac97_write()的区别在于前者在值已经设置的情况下不会再设置，而后者则会再写一次。snd_ac97_update_bits()用于更新寄存器的某些位，由 mask 决定。

除此之外，还有一个函数可用于设置采样率：

```
    int snd_ac97_set_rate(struct snd_ac97 *ac97, int reg, unsigned int
rate);
```

这个函数的第二个参数 reg 可以是 AC97_PCM_MIC_ADC_RATE、AC97_PCM_FRONT_DAC_ RATE、AC97_PCM_LR_ADC_RATE 和 AC97_SPDIF，对于 AC97 _SPDIF 而言，寄存器并非真地被改变了，只是相应的 IEC958 状态位将被更新。

4．时钟调整

在一些芯片上，编解码器的时钟频率不是 48000Hz，而是使用 PCI 时钟以节省一个晶振，在这种情况下，我们应该改变 bus->clock 为相应的值，例如 intel8x0 和 es1968 包含时钟的自动测量函数。

5．proc 文件

ALSA AC97 接口会创建如/proc/asound/card0/codec97#0/ac97#0-0 和 ac97#0-0+regs 这样的 proc 文件，通过这些文件可以查看编解码器目前的状态和寄存器。

如果一个芯片上有多个 codecs，可多次调用 snd_ac97_mixer()。

17.4.6　ALSA 用户空间编程

ALSA 驱动的声卡在用户空间不宜直接使用文件接口，而应使用 alsa-lib，代码清单 17.26 所示为基于 ALSA 音频驱动的最简单的播放应用程序。

代码清单 17.26　ALSA 用户空间播放程序

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <alsa/asoundlib.h>
4
```

```
    5  main(int argc, char *argv[])
    6  {
    7    int i;
    8    int err;
    9    short buf[128];
    10   snd_pcm_t *playback_handle;      //PCM 设备句柄
    11   snd_pcm_hw_params_t *hw_params; //硬件信息和 PCM 流配置
    12   //打开 PCM，最后一个参数为 0 意味着标准配置
    13        if   ((err   =   snd_pcm_open(&playback_handle,   argv[1],
SND_PCM_STREAM_PLAYBACK, 0)
    14     ) < 0)
    15   {
    16     fprintf(stderr, "cannot open audio device %s (%s)\n", argv[1],
snd_strerror
    17       (err));
    18     exit(1);
    19   }
    20   //分配 snd_pcm_hw_params_t 结构体
    21   if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0)
    22   {
    23      fprintf(stderr, "cannot allocate hardware parameter structure
(%s)\n",
    24      snd_strerror(err));
    25      exit(1);
    26   }
    27   //初始化 hw_params
    28   if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) < 0)
    29   {
    30          fprintf(stderr,  "cannot  initialize  hardware  parameter
structure (%s)\n",
    31          snd_strerror(err));
    32      exit(1);
    33   }
    34   //初始化访问权限
    35      if   ((err   =   snd_pcm_hw_params_set_access(playback_handle,
hw_params,
    36      SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
    37   {
    38            fprintf(stderr,   "cannot   set   access   type   (%s)\n",
snd_strerror(err));
    39      exit(1);
    40   }
    41   //初始化采样格式
    42      if   ((err   =   snd_pcm_hw_params_set_format(playback_handle,
```

```
hw_params,
    43      SND_PCM_FORMAT_S16_LE)) < 0)
    44   {
    45            fprintf(stderr,  "cannot  set  sample  format  (%s)\n",
snd_strerror(err));
    46      exit(1);
    47   }
    48   //设置采样率，如果硬件不支持我们设置的采样率，将使用最接近的
    49    if  ((err  =  snd_pcm_hw_params_set_rate_near(playback_handle,
hw_params, 44100,
    50    0)) < 0)
    51   {
    52           fprintf(stderr,  "cannot  set  sample  rate  (%s)\n",
snd_strerror(err));
    53    exit(1);
    54   }
    55    //设置通道数量
    56     if  ((err  =  snd_pcm_hw_params_set_channels(playback_handle,
hw_params, 2)) < 0)
    57   {
    58           fprintf(stderr,  "cannot  set  channel  count  (%s)\n",
snd_strerror(err));
    59     exit(1);
    60   }
    61   //设置 hw_params
    62   if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0)
    63   {
    64               fprintf(stderr,  "cannot  set  parameters  (%s)\n",
snd_strerror(err));
    65      exit(1);
    66   }
    67   //释放分配的 snd_pcm_hw_params_t 结构体
    68   snd_pcm_hw_params_free(hw_params);
    69   //完成硬件参数设置，使设备准备好
    70   if ((err = snd_pcm_prepare(playback_handle)) < 0)
    71   {
    72    fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
    73        snd_strerror(err));
    74    exit(1);
    75   }
```

```
76
77   for (i = 0; i < 10; ++i)
78   {
79     //写音频数据到 PCM 设备
80     if ((err = snd_pcm_writei(playback_handle, buf, 128)) != 128)
81     {
82        fprintf(stderr, "write to audio interface failed (%s)\n",
snd_strerror
83          (err));
84       exit(1);
85     }
86   }
87   //关闭 PCM 设备句柄
88   snd_pcm_close(playback_handle);
89   exit(0);
90 }
```

由上述代码可以看出，ALSA 用户空间编程的流程与 17.3.4 小节给出的 OSS 驱动用户空间编程的流程基本是一致的，都经过了“打开－设置参数－读写音频数据”的过程，不同在于 OSS 打开的是设备文件，设置参数使用的是 ioctl()系统调用，读写音频数据使用的是 read()、write()文件 API，而 ALSA 则全部使用 alsa-lib 中的 API。

把上述代码第 80 行的 snd_pcm_writei()函数替换为 snd_pcm_readi()，变成了一个最简单的录音程序。

代码清单 17.27 的程序打开一个音频接口，配置它为立体声、16 位、44.1kHz 采样和基于 interleave 的读写。它阻塞等待直接接口准备好接收放音数据，这时候将数据复制到缓冲区。这种设计方法使得程序很容易移植到类似 JACK、LADSPA、Coreaudio、VST 等 callback 机制驱动的系统。

代码清单 17.27　ALSA 用户空间播放程序（基于“中断”）

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <errno.h>
4   #include <poll.h>
5   #include <alsa/asoundlib.h>
6
7   snd_pcm_t *playback_handle;
8   short buf[4096];
9
10  int playback_callback(snd_pcm_sframes_t nframes)
11  {
```

```
    12   int err;
    13   printf("playback callback called with %u frames\n", nframes);
    14   /* 填充缓冲区 */
    15   if ((err = snd_pcm_writei(playback_handle, buf, nframes)) < 0)
    16   {
    17      fprintf(stderr, "write failed (%s)\n", snd_strerror(err));
    18   }
    19
    20   return err;
    21 }
    22
    23 main(int argc, char *argv[])
    24 {
    25
    26   snd_pcm_hw_params_t *hw_params;
    27   snd_pcm_sw_params_t *sw_params;
    28   snd_pcm_sframes_t frames_to_deliver;
    29   int nfds;
    30   int err;
    31   struct pollfd *pfds;
    32
    33          if   ((err   =   snd_pcm_open(&playback_handle,   argv[1],
SND_PCM_STREAM_PLAYBACK, 0)
    34        ) < 0)
    35   {
    36     fprintf(stderr, "cannot open audio device %s (%s)\n", argv[1],
snd_strerror
    37        (err));
    38     exit(1);
    39   }
    40
    41   if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0)
    42   {
    43       fprintf(stderr, "cannot allocate hardware parameter structure
(%s)\n",
    44           snd_strerror(err));
    45     exit(1);
    46   }
    47
    48   if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) <
0)
```

```
    49    {
    50            fprintf(stderr, "cannot initialize hardware parameter
structure (%s)\n",
    51            snd_strerror(err));
    52        exit(1);
    53    }
    54
    55        if ((err = snd_pcm_hw_params_set_access(playback_handle,
hw_params,
    56        SND_PCM_ACCESS_RW_INTERLEAVED)) < 0)
    57    {
    58            fprintf(stderr, "cannot set access type (%s)\n",
snd_strerror(err));
    59        exit(1);
    60    }
    61
    62        if ((err = snd_pcm_hw_params_set_format(playback_handle,
hw_params,
    63        SND_PCM_FORMAT_S16_LE)) < 0)
    64    {
    65            fprintf(stderr, "cannot set sample format (%s)\n",
snd_strerror(err));
    66        exit(1);
    67    }
    68
    69      if ((err = snd_pcm_hw_params_set_rate_near(playback_handle,
hw_params, 44100,
    70        0)) < 0)
    71    {
    72            fprintf(stderr, "cannot set sample rate (%s)\n",
snd_strerror(err));
    73        exit(1);
    74    }
    75
    76      if ((err = snd_pcm_hw_params_set_channels(playback_handle,
hw_params, 2)) < 0)
    77    {
    78            fprintf(stderr, "cannot set channel count (%s)\n",
```

```
snd_strerror(err));
    79      exit(1);
    80    }
    81
    82    if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0)
    83    {
    84              fprintf(stderr,  "cannot  set  parameters  (%s)\n",
snd_strerror(err));
    85      exit(1);
    86    }
    87
    88    snd_pcm_hw_params_free(hw_params);
    89
    90    /* 告诉 ALSA 当 4096 个以上帧可以传递时唤醒我们 */
    91    if ((err = snd_pcm_sw_params_malloc(&sw_params)) < 0)
    92    {
    93      fprintf(stderr, "cannot allocate software parameters structure
(%s)\n",
    94          snd_strerror(err));
    95      exit(1);
    96    }
    97    if ((err = snd_pcm_sw_params_current(playback_handle, sw_params))
< 0)
    98    {
    99            fprintf(stderr,  "cannot  initialize  software  parameters
structure (%s)\n",
    100         snd_strerror(err));
    101       exit(1);
    102    }
    103    /* 设置 4096 帧传递一次数据 */
    104     if  ((err  =  snd_pcm_sw_params_set_avail_min(playback_handle,
sw_params, 4096))
    105       < 0)
    106    {
    107      fprintf(stderr, "cannot set minimum available count (%s)\n",
snd_strerror
    108         (err));
    109     exit(1);
    110   }
    111   /* 一旦有数据就开始播放 */
```

```
    112                                   if            ((err            =
snd_pcm_sw_params_set_start_threshold(playback_handle, sw_params,
    113        0U)) < 0)
    114   {
    115             fprintf(stderr,  "cannot  set  start  mode  (%s)\n",
snd_strerror(err));
    116     exit(1);
    117   }
    118   if ((err = snd_pcm_sw_params(playback_handle, sw_params)) < 0)
    119   {
    120         fprintf(stderr, "cannot set software parameters (%s)\n",
snd_strerror(err));
    121       exit(1);
    122   }
    123
    124   /* 每 4096 帧接口将中断内核，ALSA 将很快唤醒本程序 */
    125
    126   if ((err = snd_pcm_prepare(playback_handle)) < 0)
    127   {
    128        fprintf(stderr, "cannot prepare audio interface for use
(%s)\n",
    129           snd_strerror(err));
    130       exit(1);
    131   }
    132
    133   while (1)
    134   {
    135
    136      /* 等待，直到接口准备好传递数据，或者 1s 超时发生 */
    137      if ((err = snd_pcm_wait(playback_handle, 1000)) < 0)
    138   {
    139        fprintf(stderr, "poll failed (%s)\n", strerror(errno));
    140        break;
    141     }
    142
    143     /* 查出有多少空间可放置 playback 数据 */
    144     if ((frames_to_deliver = snd_pcm_avail_update(playback_handle))
< 0)
```

```
145     {
146         if (frames_to_deliver ==  - EPIPE)
147         {
148             fprintf(stderr, "an xrun occured\n");
149             break;
150         }
151         else
152         {
153            fprintf(stderr, "unknown ALSA avail update return value
(%d)\n",
154                frames_to_deliver);
155            break;
156        }
157     }
158
159         frames_to_deliver = frames_to_deliver > 4096 ? 4096 :
frames_to_deliver;
160
161     /* 传递数据 */
162     if (playback_callback(frames_to_deliver) != frames_to_deliver)
163     {
164       fprintf(stderr, "playback callback failed\n");
165       break;
166     }
167   }
168
169   snd_pcm_close(playback_handle);
170   exit(0);
171 }
```

17.5 S3C2410+UDA1341 OSS 驱动实例

17.5.1 S3C2410 与 UDA1341 接口硬件描述

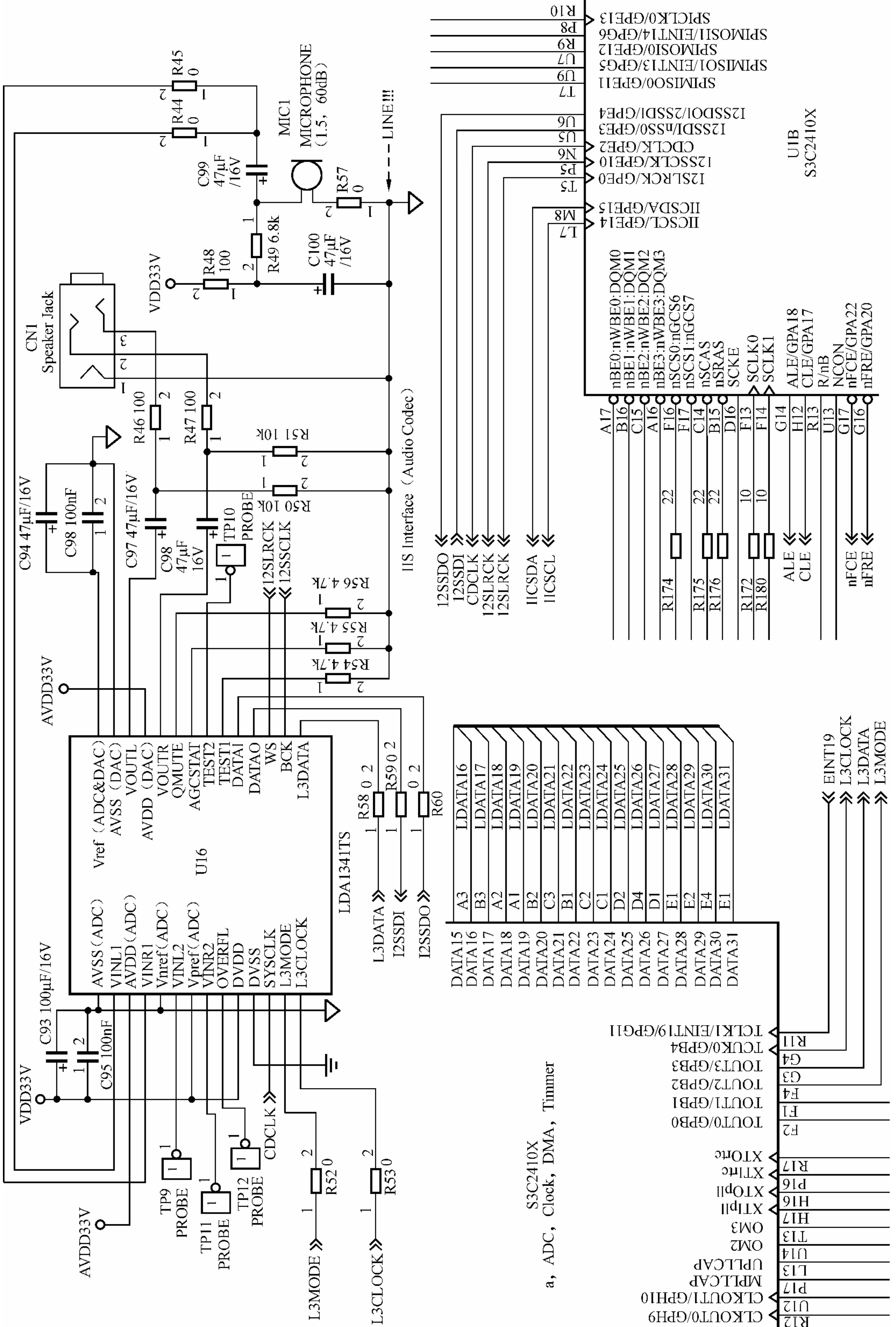
如图 17.7 所示，S3C2410 处理器内置了 IIS 总线接口，S3C2410 的 IIS 总线时钟信号 SCK 与 Philips 公司的 UDA1341 的 BCK 连接，字段选择连接于 WS 引脚。UDA1341 提供两个音频通道，分别用于输入和输出，对应的引脚连接：IIS 总线的音频输出 IISSDO 对应于 UDA1341 的音频输入，IIS 总线的音频输入 IISSDI 对应于 UDA1341 的音频输出。

UDA1341 的 L3 接口相当于一个混音器控制接口，可以用来控制输入/输出音频信号的音量大小、低音等。L3 接口的引脚 L3MODE、L3DATA、L3CLOCK 分别连接到 S3C2410

的 3 个 GPIO 来控制。

Philips 公司的 UDA1341 支持 IIS 总线数据格式，采用位元流转换技术进行信号处理，完成声音信号的 A/D 转换，具有可编程增益放大器和数字自动增益控制器，其低功耗、低电压的特点使其非常适合用于 MD/CD、笔记本电脑等便携式设备。UDA1341 对外提供两组音频信号输入接口，每组包括左右两个声道。

华清远见



如图 17.8 所示，两组音频输入在 UDA1341 内部的处理存在很大差别：第一组音频信号输入后经过一个 0 dB/6 dB 开关后采样送入数字混音器；第二组音频信号输

入后先经过可编程增益放大器（PGA），然后再进行采样，采样后的数据要再经过数字自动增益控制器（AGC）送入数字混音器。

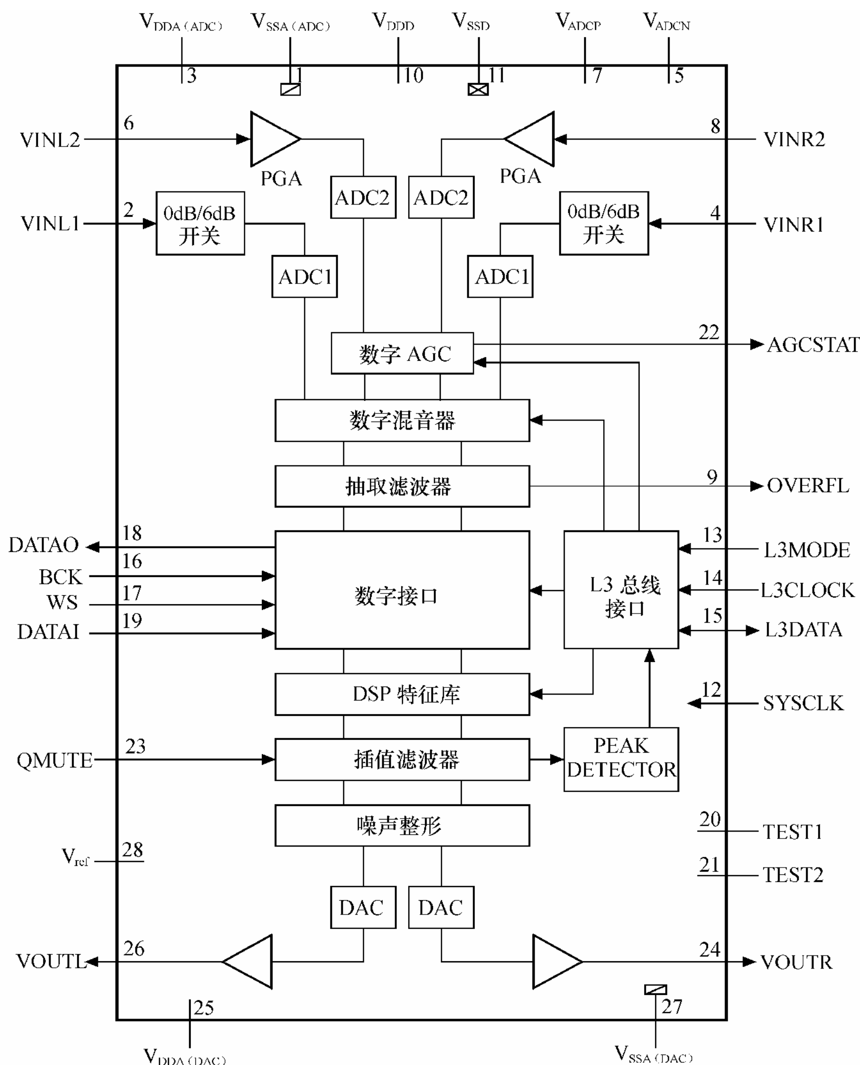


图 17.8　UDA1341 内部结构

设计硬件电路时选用第二组输入音频信号，这样可以通过软件的方法实现对系统输入音量大小的调节。显然选用第二组可以通过 L3 总线接口控制 AGC 来实现。另外，选择通道 2 还可以通过 PGA 对从 MIC 输入的信号进行片内放大。

S3C2410 与 UDA1341 之间的 IIS 接口有以下 3 种工作方式。

- 正常传输模式：该模式下使用 IISCON 寄存器对 FIFO 进行控制，CPU 通过轮询方式访问 FIFO 寄存器，以完成对 FIFO 缓存传输或接收的处理。
- DMA 模式：通过设置 IISFCON 寄存器使 IIS 接口工作于这种模式。在该模式下，FIFO 寄存器组的控制权掌握在 DMA 控制器上，当 FIFO 满时，由

DMA 控制器对 FIFO 中的数据进行处理。DMA 模式的选择由 IISCON 寄存器的第 4 位和第 5 位控制。

- 传输/接收模式：在该模式下，IIS 数据线将通过双通道 DMA 同时接收和发送音频数据。在 OSS 驱动中，将使用此模式。

17.5.2 注册 dsp 和 mixer 接口

在 UDA1341 OSS 驱动的模块加载函数中，将完成如下工作，如代码清单 17.28 所示。

- 初始化 IIS 接口硬件，设置 L3 总线对应的 GPIO。
- 申请用于音频数据传输的 DMA 通道。
- 初始化 UDA1341 到恰当的工作模式。
- 注册 dsp 和 mixer 接口。

代码清单 17.28 UDA1341 OSS 驱动的模块加载函数

```
1  //音频(dsp)文件操作
2  static struct file_operations smdk2410_audio_fops =
3  {
4      llseek: smdk2410_audio_llseek,
5      write: smdk2410_audio_write,
6      read:  smdk2410_audio_read,
7      poll: smdk2410_audio_poll,
8      ioctl: smdk2410_audio_ioctl,
9      open: smdk2410_audio_open,
10     release: smdk2410_audio_release
11 };
12 //混音器文件操作
13 static struct file_operations smdk2410_mixer_fops =
14 {
15     ioctl: smdk2410_mixer_ioctl,
16     open: smdk2410_mixer_open,
17     release: smdk2410_mixer_release
18 };
19
20 int __init s3c2410_uda1341_init(void)
21 {
22     unsigned long flags;
23
24     local_irq_save(flags);
25
26     /* 设置 IIS 接口引脚 GPIO */
27
28    set_gpio_ctrl(GPIO_L3CLOCK); // GPB 4: L3CLOCK, 输出
```

```
29     set_gpio_ctrl(GPIO_L3DATA);  // GPB 3: L3DATA, 输出
30     set_gpio_ctrl(GPIO_L3MODE);  // GPB 2: L3MODE, 输出
31
32
33      set_gpio_ctrl(GPIO_E3 | GPIO_PULLUP_EN | GPIO_MODE_IISSDI);
//GPE 3: IISSDI
34      set_gpio_ctrl(GPIO_E0 | GPIO_PULLUP_EN | GPIO_MODE_IISSDI);
//GPE 0: IISLRCK
35      set_gpio_ctrl(GPIO_E1 | GPIO_PULLUP_EN | GPIO_MODE_IISSCLK);
//GPE 1:IISSCLK
36     set_gpio_ctrl(GPIO_E2 | GPIO_PULLUP_EN | GPIO_MODE_CDCLK); //GPE
2: CDCLK
37      set_gpio_ctrl(GPIO_E4 | GPIO_PULLUP_EN | GPIO_MODE_IISSDO);
//GPE 4: IISSDO
38
39     local_irq_restore(flags);
40
41     init_uda1341();
42
43     /* 输出流采样 DMA 通道 2 */
44     output_stream.dma_ch = DMA_CH2;
45
46     if (audio_init_dma(&output_stream, "UDA1341 out"))
47     {
48         audio_clear_dma(&output_stream);
49         printk(KERN_WARNING AUDIO_NAME_VERBOSE ": unable to get DMA
channels\n");
50         return  - EBUSY;
51     }
52     /* 输入流采样 DMA 通道 1 */
53     input_stream.dma_ch = DMA_CH1;
54
55     if (audio_init_dma(&input_stream, "UDA1341 in"))
56     {
57        audio_clear_dma(&input_stream);
58        printk(KERN_WARNING AUDIO_NAME_VERBOSE ": unable to get DMA
channels\n");
59        return  - EBUSY;
```

```
60      }
61
62          /* 注册 dsp 和 mixer 设备接口 */
63          audio_dev_dsp = register_sound_dsp(&smdk2410_audio_fops,  -
1);
64          audio_dev_mixer = register_sound_mixer(&smdk2410_mixer_fops,
- 1);
65
66          printk(AUDIO_NAME_VERBOSE " initialized\n");
67
68          return 0;
69  }
```

UDA1341 OSS 驱动的模块卸载函数中，将完成与模块加载函数相反的工作，如代码清单 17.29 所示。

代码清单 17.29　UDA1341 OSS 驱动的模块卸载函数

```
1  void __exit s3c2410_uda1341_exit(void)
2  {
3    //注销 dsp 和 mixer 设备接口
4   unregister_sound_dsp(audio_dev_dsp);
5   unregister_sound_mixer(audio_dev_mixer);
6
7   //注销 DMA 通道
8   audio_clear_dma(&output_stream);
9   audio_clear_dma(&input_stream); /* input */
10  printk(AUDIO_NAME_VERBOSE " unloaded\n");
11 }
```

17.5.3　mixer 接口的 I/O 控制函数

UDA1341 OSS 驱动的 ioctl()函数处理多个 mixer 命令，如 SOUND_MIXER_INFO、SOUND_ MIXER_READ_STEREODEVS、SOUND_MIXER_WRITE_VOLUME 等，用于获得或设置音量和增益等信息，如代码清单 17.30 所示。

代码清单 17.30　UDA1341 OSS 驱动的 ioctl()函数

```
1  static int smdk2410_mixer_ioctl(struct inode *inode, struct file
*file,
2     unsigned int cmd, unsigned long arg)
3  {
4     int ret;
5     long val = 0;
6
7     switch (cmd)
8     {
9        case SOUND_MIXER_INFO:   //获得 mixer 信息
10          {
11             mixer_info info;
```

```
    12            strncpy(info.id, "UDA1341", sizeof(info.id));
    13                          strncpy(info.name,  "Philips  UDA1341",
sizeof(info.name));
    14            info.modify_counter = audio_mix_modcnt;
    15            return copy_to_user((void*)arg, &info, sizeof(info));
    16          }
    17
    18        case SOUND_OLD_MIXER_INFO:
    19          {
    20            _old_mixer_info info;
    21            strncpy(info.id, "UDA1341", sizeof(info.id));
    22                          strncpy(info.name,  "Philips  UDA1341",
sizeof(info.name));
    23            return copy_to_user((void*)arg, &info, sizeof(info));
    24          }
    25
    26         case SOUND_MIXER_READ_STEREODEVS://获取设备对立体声的支持
    27            return put_user(0, (long*)arg);
    28
    29        case SOUND_MIXER_READ_CAPS:  //获取声卡能力
    30            val = SOUND_CAP_EXCL_INPUT;
    31            return put_user(val, (long*)arg);
    32
    33        case SOUND_MIXER_WRITE_VOLUME:   //设置音量
    34            ret = get_user(val, (long*)arg);
    35            if (ret)
    36                return ret;
    37            uda1341_volume = 63-(((val &0xff) + 1) *63) / 100;
    38            uda1341_l3_address(UDA1341_REG_DATA0);
    39            uda1341_l3_data(uda1341_volume);
    40            break;
    41
    42        case SOUND_MIXER_READ_VOLUME:   //获取音量
    43            val = ((63-uda1341_volume) *100) / 63;
    44            val |= val << 8;
    45            return put_user(val, (long*)arg);
    46
    47        case SOUND_MIXER_READ_IGAIN:   //获得增益
    48            val = ((31-mixer_igain) *100) / 31;
    49            return put_user(val, (int*)arg);
    50
    51        case SOUND_MIXER_WRITE_IGAIN:  //设置增益
    52            ret = get_user(val, (int*)arg);
    53            if (ret)
    54                return ret;
    55            mixer_igain = 31-(val *31 / 100);
    56            /* 使用 mixer 增益通道 1 */
    57            uda1341_l3_address(UDA1341_REG_DATA0);
    58            uda1341_l3_data(EXTADDR(EXT0));
    59            uda1341_l3_data(EXTDATA(EXT0_CH1_GAIN(mixer_igain)));
    60            break;
```

```
61
62      default:
63        DPRINTK("mixer ioctl %u unknown\n", cmd);
64        return  - ENOSYS;
65    }
66
67    audio_mix_modcnt++;
68    return 0;
69 }
```

17.5.4 dsp 接口音频数据传输

OSS 声卡驱动中，dsp 接口的读写函数是核心，直接对应着录音和播放的流程。

OSS 的读函数存在一个与普通字符设备驱动读函数不同的地方。一般来说，对于普通字符设备驱动，如果用户要求读 count 个字节，而实际上只有 count1 字节可获得（count1< count）时，它会将这 count1 字节复制给用户后即返回 count1；而 dsp 接口的读函数会分次复制，如果第一次不能满足，它会等待第二次，直到"count1 + count2 + … = count"为止再返回 count。这种设计是合理的，因为 OSS 驱动应该负责音频数据的流量控制。代码清单 17.31 所示为 UDA1341 OSS 驱动的读函数实现。

代码清单 17.31　UDA1341 OSS 驱动的读函数

```
  1 static ssize_t smdk2410_audio_read(struct file *file, char *buffer, size_t
  2     count, loff_t *ppos)
  3 {
  4     const char *buffer0 = buffer;
  5     audio_stream_t *s = &input_stream; //得到数据区的指针
  6     int chunksize, ret = 0;
  7
  8     DPRINTK("audio_read: count=%d\n", count);
  9
 10     if (ppos !=  &file->f_pos)
 11        return  - ESPIPE;
 12
 13     if (!s->buffers)
 14     {
 15         int i;
 16
 17         if (audio_setup_buf(s))
 18             return  - ENOMEM;
 19         //依次从缓存区读取数据
 20         for (i = 0; i < s->nbfrags; i++)
 21         {
 22              audio_buf_t *b = s->buf;
 23      down(&b->sem);
 24        s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr, s->fragsize,
 25          DMA_BUF_RD);
 26      NEXT_BUF(s, buf);
```

```
27      }
28    }
29
30    //满足用户的所有读需求
31    while (count > 0)
32    {
33      audio_buf_t *b = s->buf;
34
35      if (file->f_flags &O_NONBLOCK) //非阻塞
36      {
37          ret =  - EAGAIN;
38          if (down_trylock(&b->sem))
39              break;
40      }
41      else
42      {
43          ret =  - ERESTARTSYS;
44          if (down_interruptible(&b->sem))
45              break;
46      }
47
48      chunksize = b->size;
49      //从缓存区读取数据
50      if (chunksize > count)
51          chunksize = count;
52      DPRINTK("read %d from %d\n", chunksize, s->buf_idx);
53      if (copy_to_user(buffer, b->start + s->fragsize - b->size,  //
调用复制函数
54      chunksize))
55      {
56          up(&b->sem);
57          return  - EFAULT;
58      }
59      b->size -= chunksize;
60
61      buffer += chunksize;
62      count -= chunksize;  //已经给用户复制了一部分，count 减少
63      if (b->size > 0)
64      {
65          up(&b->sem);
66          break;
67      }
```

```
    68      //将缓存区释放
    69        s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr,
s->fragsize,
    70          DMA_BUF_RD);
    71
    72      NEXT_BUF(s, buf);
    73    }
    74
    75    if ((buffer - buffer0))
    76        ret = buffer - buffer0;
    77
    78    return ret;
    79  }
```

OSS 驱动 dsp 接口的写函数与读函数类似，一般来说，它也应该满足用户的所有写需求后再返回，如代码清单 17.32 所示。

代码清单 17.32　UDA1341 OSS 驱动的写函数

```
    1  static ssize_t smdk2410_audio_write(struct file *file, const char
*buffer,
    2      size_t count, loff_t *ppos)
    3  {
    4      const char *buffer0 = buffer;
    5      audio_stream_t *s = &output_stream;
    6      int chunksize, ret = 0;
    7
    8      DPRINTK("audio_write : start count=%d\n", count);
    9
    10     switch (file->f_flags &O_ACCMODE)
    11     {
    12       case O_WRONLY: //只写
    13       case O_RDWR: //读写
    14         break;
    15       default:  //只读不合法
    16         return  - EPERM;
    17     }
    18     //设置 DMA 缓冲区
    19     if (!s->buffers && audio_setup_buf(s))
    20        return  - ENOMEM;
    21
    22     count &= ~0x03;
    23
    24     while (count > 0) //直到满足用户的所有写需求
    25     {
    26       audio_buf_t *b = s->buf;
```

```
27          //非阻塞访问
28          if (file->f_flags &O_NONBLOCK)
29           {
30               ret =  - EAGAIN;
31               if (down_trylock(&b->sem))
32                   break;
33          }
34          else
35          {
36               ret =  - ERESTARTSYS;
37               if (down_interruptible(&b->sem))
38                   break;
39         }
40         //从用户空间复制音频数据
41         if (audio_channels == 2)
42         {
43              chunksize = s->fragsize - b->size;
44              if (chunksize > count)
45                  chunksize = count;
46              DPRINTK("write %d to %d\n", chunksize, s->buf_idx);
47              if (copy_from_user(b->start + b->size, buffer, chunksize))
48           {
49               up(&b->sem);
50               return  - EFAULT;
51            }
52           b->size += chunksize;
53        }
54        else
55        {
56            chunksize = (s->fragsize - b->size) >> 1;
57
58            if (chunksize > count)
59                chunksize = count;
60            DPRINTK("write %d to %d\n", chunksize *2, s->buf_idx);
61            if (copy_from_user_mono_stereo(b->start + b->size, buffer,
chunksize))
62            {
63                up(&b->sem);
64                return  - EFAULT;
```

```
    65          }
    66
    67          b->size += chunksize * 2;
    68        }
    69
    70        buffer += chunksize;
    71        count -= chunksize;  //已经从用户复制了一部分，count 减少
    72        if (b->size < s->fragsize)
    73        {
    74            up(&b->sem);
    75            break;
    76        }
    77       //发起 DMA 操作
    78         s3c2410_dma_queue_buffer(s->dma_ch, (void*)b, b->dma_addr,
b->size,
    79            DMA_BUF_WR);
    80       b->size = 0;
    81       NEXT_BUF(s, buf);
    82     }
    83
    84     if ((buffer - buffer0))
    85        ret = buffer - buffer0;
    86
    87      DPRINTK("audio_write : end count=%d\n\n", ret);
    88
    89      return ret;
    90  }
```

17.6 SA1100+ UDA1341 ALSA 驱动实例

17.6.1 card 注册与注销

同样是 UDA1341 芯片，如果以 ALSA 体系结构来实现它的驱动，会和 OSS 大不一样。在模块初始化和卸载的时候，需要注册和注销 card，另外在模块加载的时候，也会注册 mixer 和 PCM 组件，如代码清单 17.33 所示。

代码清单 17.33 UDA1341 ALSA 驱动的模块初始化与卸载

```
    1  static int __init sa11xx_uda1341_probe(struct platform_device
*devptr)
    2  {
    3  int err;
    4  struct snd_card *card;
    5  struct sa11xx_uda1341 *chip;
    6
    7  /* 新建 card */
    8     card  =  snd_card_new(-1,  id,  THIS_MODULE,  sizeof(struct
```

```
sa11xx_uda1341));
    9  if (card == NULL)
    10     return -ENOMEM;
    11
    12  chip = card->private_data;
    13  spin_lock_init(&chip->s[0].dma_lock);
    14  spin_lock_init(&chip->s[1].dma_lock);
    15
    16  card->private_free = snd_sa11xx_uda1341_free;//card 私有数据释放
    17  chip->card = card;
    18  chip->samplerate = AUDIO_RATE_DEFAULT;
    19
    20  // 注册 control(mixer)接口
    21  if ((err = snd_chip_uda1341_mixer_new(card, &chip->uda1341)))
    22    goto nodev;
    23
    24  // 注册 PCM 接口
    25  if ((err = snd_card_sa11xx_uda1341_pcm(chip, 0)) < 0)
    26    goto nodev;
    27
    28  strcpy(card->driver, "UDA1341");
    29  strcpy(card->shortname, "H3600 UDA1341TS");
    30   sprintf(card->longname,  "Compaq  iPAQ  H3600  with  Philips
UDA1341TS");
    31
    32  snd_card_set_dev(card, &devptr->dev);
    33   //注册 card
    34  if ((err = snd_card_register(card)) == 0) {
    35    printk( KERN_INFO "iPAQ audio support initialized\n" );
    36    platform_set_drvdata(devptr, card);
    37    return 0;
    38  }
    39
    40  nodev:
    41  snd_card_free(card);
    42  return err;
    43 }
    44
    45 static int __devexit sa11xx_uda1341_remove(struct platform_device
*devptr)
    46 {
    47  //释放 card
    48  snd_card_free(platform_get_drvdata(devptr));
    49  platform_set_drvdata(devptr, NULL);
    50  return 0;
    51 }
```

17.6.2 PCM 设备的实现

PCM 组件直接对应着 ALSA 驱动的录音和播放，由 17.4.2 小节可知，驱动从需要定义对应相应的 snd_pcm_hardware 结构体进行 PCM 设备硬件描述，如代码清单 17.34 所示。

代码清单 17.34 UDA1341 ALSA 驱动的 PCM 接口的 snd_pcm_hardware 结构体

```
1 static struct snd_pcm_hardware snd_sa11xx_uda1341_capture =
2 {
3 .info                = (SNDRV_PCM_INFO_INTERLEAVED |
4                      SNDRV_PCM_INFO_BLOCK_TRANSFER |
5                      SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_MMAP_VALID
|
6                      SNDRV_PCM_INFO_PAUSE | SNDRV_PCM_INFO_RESUME),
7 .formats       = SNDRV_PCM_FMTBIT_S16_LE,
8 .rates            = (SNDRV_PCM_RATE_8000 | SNDRV_PCM_RATE_16000 |\
9                   SNDRV_PCM_RATE_22050 | SNDRV_PCM_RATE_32000 |\
10                  SNDRV_PCM_RATE_44100 | SNDRV_PCM_RATE_48000 |\
11                  SNDRV_PCM_RATE_KNOT),
12  .rate_min        = 8000,
13  .rate_max            = 48000,
14  .channels_min            = 2,
15  .channels_max            = 2,
16  .buffer_bytes_max        = 64*1024,
17  .period_bytes_min        = 64,
18  .period_bytes_max        = DMA_BUF_SIZE,
19  .periods_min             = 2,
20  .periods_max             = 255,
21  .fifo_size               = 0,
22 };
23
24 static struct snd_pcm_hardware snd_sa11xx_uda1341_playback =
25 {
26  .info                = (SNDRV_PCM_INFO_INTERLEAVED |
27                     SNDRV_PCM_INFO_BLOCK_TRANSFER |
28                     SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_MMAP_VALID
|
29                     SNDRV_PCM_INFO_PAUSE | SNDRV_PCM_INFO_RESUME),
30  .formats       =SNDRV_PCM_FMTBIT_S16_LE,
31  .rates             = (SNDRV_PCM_RATE_8000 | SNDRV_PCM_RATE_16000 |\
32                    SNDRV_PCM_RATE_22050 | SNDRV_PCM_RATE_32000 |\
33                    SNDRV_PCM_RATE_44100 | SNDRV_PCM_RATE_48000 |\
```

```
34                      SNDRV_PCM_RATE_KNOT),
35  .rate_min                   = 8000,
36  .rate_max               = 48000,
37  .channels_min               = 2,
38  .channels_max           = 2,
39  .buffer_bytes_max           = 64*1024,
40  .period_bytes_min           = 64,
41  .period_bytes_max           = DMA_BUF_SIZE,
42  .periods_min                = 2,
43  .periods_max                = 255,
44  .fifo_size                  = 0,
45 };
```

PCM 接口的主要函数被封装在 snd_pcm_ops 结构体内，UDA1341 ALSA 驱动对 snd_pcm_ops 结构体的定义如代码清单 17.35 所示。

代码清单 17.35　UDA1341 ALSA 驱动的 PCM 接口的 snd_pcm_ops 结构体

```
1  static struct snd_pcm_ops snd_card_sa11xx_uda1341_playback_ops =
2  {
3  .open                   = snd_card_sa11xx_uda1341_open,
4  .close                  = snd_card_sa11xx_uda1341_close,
5  .ioctl                  = snd_pcm_lib_ioctl,
6  .hw_params              = snd_sa11xx_uda1341_hw_params,
7  .hw_free                = snd_sa11xx_uda1341_hw_free,
8  .prepare                = snd_sa11xx_uda1341_prepare,
9  .trigger            = snd_sa11xx_uda1341_trigger,
10 .pointer            = snd_sa11xx_uda1341_pointer,
11 };
12
13 static struct snd_pcm_ops snd_card_sa11xx_uda1341_capture_ops =
14 {
15 .open                   = snd_card_sa11xx_uda1341_open,
16 .close                  = snd_card_sa11xx_uda1341_close,
17 .ioctl                  = snd_pcm_lib_ioctl,
18 .hw_params              = snd_sa11xx_uda1341_hw_params,
19 .hw_free                = snd_sa11xx_uda1341_hw_free,
20 .prepare                = snd_sa11xx_uda1341_prepare,
21 .trigger                = snd_sa11xx_uda1341_trigger,
22 .pointer                = snd_sa11xx_uda1341_pointer,
```

```
23 };
```

代码清单 17.33 第 25 行调用的 snd_card_sa11xx_uda1341_pcm()即是 PCM 组件的“构造函数”，其实现如代码清单 17.36 所示。

代码清单 17.36　UDA1341 ALSA 驱动的 PCM 组件构造函数

```
1    static   int   __init   snd_card_sa11xx_uda1341_pcm(struct sa11xx_uda1341 *sa11xx_ uda1341, int device)
2  {
3  struct snd_pcm *pcm;
4  int err;
5  /* 新建 PCM 设备，playback 和 capture 都为 1 个 */
6   if ((err = snd_pcm_new(sa11xx_uda1341->card, "UDA1341 PCM", device, 1, 1, &pcm)) < 0)
7      return err;
8
9   /* 建立初始缓冲区并设置 dma_type 为 isa */
10  snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
11                        snd_dma_isa_data(),
12                        64*1024, 64*1024);
13 /* 设置 PCM 的操作 */
14            snd_pcm_set_ops(pcm,          SNDRV_PCM_STREAM_PLAYBACK, snd_card_sa11xx_uda1341_playback_ops);
15            snd_pcm_set_ops(pcm,          SNDRV_PCM_STREAM_CAPTURE, &snd_card_sa11xx_uda1341_capture_ops);
16 pcm->private_data = sa11xx_uda1341;
17 pcm->info_flags = 0;
18 strcpy(pcm->name, "UDA1341 PCM");
19
20 sa11xx_uda1341_audio_init(sa11xx_uda1341);
21
22 /* 设置 DMA 控制器 */
23     audio_dma_request(&sa11xx_uda1341->s[SNDRV_PCM_STREAM_PLAYBACK], audio_dma_callback);
24      audio_dma_request(&sa11xx_uda1341->s[SNDRV_PCM_STREAM_CAPTURE], audio_dma_callback);
25
26 sa11xx_uda1341->pcm = pcm;
27
28 return 0;
29 }
```

在 snd_pcm_ops 结构体的打开成员函数中，需要根据具体的子流赋值 snd_pcm_runtime 的 hw，如代码清单 17.37 所示。

代码清单 17.37　UDA1341 ALSA 驱动的 snd_pcm_ops 结构体的 open/close 成员函数

```
1  static int snd_card_sa11xx_uda1341_open(struct snd_pcm_substream *substream)
2  {
3  struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
```

```
    4 struct snd_pcm_runtime *runtime = substream->runtime;
    5 int stream_id = substream->pstr->stream;
    6 int err;
    7
    8 chip->s[stream_id].stream = substream;
    9
    10 if (stream_id == SNDRV_PCM_STREAM_PLAYBACK) //播放子流
    11       runtime->hw = snd_sa11xx_uda1341_playback;
    12 else    //录音子流
    13      runtime->hw = snd_sa11xx_uda1341_capture;
    14     if     ((err     =     snd_pcm_hw_constraint_integer(runtime,
SNDRV_PCM_HW_PARAM_PERIODS)) < 0)
    15       return err;
    16                              if((err=snd_pcm_hw_constraint_list(runtime,0,
SNDRV_PCM_HW_PARAM_RATE,&hw_constraints_rates)) < 0)
    17       return err;
    18
    19 return 0;
    20 }
    21
    22 static int snd_card_sa11xx_uda1341_close(struct snd_pcm_substream
*substream)
    23 {
    24  struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
    25
    26  chip->s[substream->pstr->stream].stream = NULL;
    27  return 0;
    28 }
```

在 snd_pcm_ops 结构体的 trigger()成员函数中，控制播放和录音的启/停、挂起/恢复，如代码清单 17.38 所示。

代码清单 17.38　UDA1341 ALSA 驱动的 snd_pcm_ops 结构体的 trigger 成员函数

```
    1    static int snd_sa11xx_uda1341_trigger(struct snd_pcm_substream
*substream, int
    2     cmd)
    3  {
    4     struct sa11xx_uda1341 *chip = snd_pcm_substream_chip(substream);
    5     int stream_id = substream->pstr->stream;
```

```
6     struct audio_stream *s = &chip->s[stream_id];
7     struct audio_stream *s1 = &chip->s[stream_id ^ 1];
8     int err = 0;
9
10    /* 注意本地中断已经被中间层代码禁止 */
11    spin_lock(&s->dma_lock);
12    switch (cmd)
13    {
14       case SNDRV_PCM_TRIGGER_START://开启 PCM
15         if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active) //
开启录音，不在播放
16         {
17            s1->tx_spin = 1;
18            audio_process_dma(s1);
19         }
20         else
21         {
22            s->tx_spin = 0;
23         }
24
25         /* 被请求的流启动 */
26         s->active = 1;
27         audio_process_dma(s);
28         break;
29       case SNDRV_PCM_TRIGGER_STOP:
30         /* 被请求的流关闭 */
31         audio_stop_dma(s);
32         if (stream_id == SNDRV_PCM_STREAM_PLAYBACK && s1->active) //
在录音时开启播放
33         {
34            s->tx_spin = 1;
35            audio_process_dma(s);  //启动 DMA
36         }
37         else
38         {
39            if (s1->tx_spin)
40            {
41               s1->tx_spin = 0;
42               audio_stop_dma(s1);  //停止 DMA
43            }
```

```
44          }
45
46          break;
47       case SNDRV_PCM_TRIGGER_SUSPEND: //挂起
48          s->active = 0;
49          #ifdef HH_VERSION
50            sa1100_dma_stop(s->dmach); //停止 DMA
51          #endif
52          s->old_offset = audio_get_dma_pos(s) + 1;
53          #ifdef HH_VERSION
54            sa1100_dma_flush_all(s->dmach);
55          #endif
56          s->periods = 0;
57          break;
58        case SNDRV_PCM_TRIGGER_RESUME:   //恢复
59          s->active = 1;
60          s->tx_spin = 0;
61          audio_process_dma(s);  //开启 DMA
62          if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active)
63          {
64            s1->tx_spin = 1;
65            audio_process_dma(s1);
66          }
67          break;
68        case SNDRV_PCM_TRIGGER_PAUSE_PUSH:   //暂停
69           #ifdef HH_VERSION
70             sa1100_dma_stop(s->dmach);  //停止 DMA
71           #endif
72           s->active = 0;
73           if (stream_id == SNDRV_PCM_STREAM_PLAYBACK)
74          {
75             if (s1->active)
76             {
77                s->tx_spin = 1;
78                s->old_offset = audio_get_dma_pos(s) + 1;
79                #ifdef HH_VERSION
80                  sa1100_dma_flush_all(s->dmach);
81                #endif
82                audio_process_dma(s); //开启 DMA
83             }
84          }
85          else
```

```
    86          {
    87              if (s1->tx_spin)
    88              {
    89                 s1->tx_spin = 0;
    90                 #ifdef HH_VERSION
    91                    sa1100_dma_flush_all(s1->dmach);
    92                 #endif
    93              }
    94          }
    95          break;
    96        case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:   //暂停释放
    97          s->active = 1;
    98          if (s->old_offset)
    99          {
    100             s->tx_spin = 0;
    101             audio_process_dma(s);
    102             break;
    103         }
    104         if (stream_id == SNDRV_PCM_STREAM_CAPTURE && !s1->active)
    105         {
    106             s1->tx_spin = 1;
    107             audio_process_dma(s1);
    108         }
    109         #ifdef HH_VERSION
    110            sa1100_dma_resume(s->dmach);
    111         #endif
    112         break;
    113       default:
    114          err =  - EINVAL;
    115          break;
    116     }
    117     spin_unlock(&s->dma_lock);
    118     return err;
    119 }
```

snd_pcm_ops 结构体中其他的 hw_params()、prepare()、pointer()等成员函数实现较为简单，这里不再赘述。

17.6.3 控制接口的实现

代码清单 17.33 第 21 行调用的 snd_chip_uda1341_mixer_new()可以认为是 UDA1341 ALSA 驱动 mixer 控制组件的“构造函数”，其中会创建的控制元素的定义如代码清单 17.39 所示，包括一些枚举和单值元素。

代码清单 17.39 UDA1341 ALSA 驱动控制接口的 snd_kcontrol_new 结构体

```
    1  #define UDA1341_SINGLE(xname, where, reg, shift, mask, invert) \
    2    {  .iface  =  SNDRV_CTL_ELEM_IFACE_MIXER,  .name  =  xname,  .info  =
snd_uda1341_info_single,\
    3      .get = snd_uda1341_get_single, .put = snd_uda1341_put_single, \
    4      .private_value = where | (reg << 5) | (shift << 9) | (mask << 12)
| (invert <<18)\
    5    }
    6
    7   #define UDA1341_ENUM(xname, where, reg, shift, mask, invert) \
    8     {  .iface  =  SNDRV_CTL_ELEM_IFACE_MIXER,  .name  =  xname,  .info  =
snd_uda1341_info_enum, \
```

```
    9      .get = snd_uda1341_get_enum, .put = snd_uda1341_put_enum, \
    10     .private_value = where | (reg << 5) | (shift << 9) | (mask << 12) |
(invert << 18) \
    11 }
    12
    13 static struct snd_kcontrol_new snd_uda1341_controls[] =
    14 {
    15   UDA1341_SINGLE("Master Playback Switch", CMD_MUTE, data0_2, 2, 1,
1),
    16   UDA1341_SINGLE("Master Playback Volume", CMD_VOLUME, data0_0, 0,
63, 1),
    17
    18   UDA1341_SINGLE("Bass Playback Volume", CMD_BASS, data0_1, 2, 15,
0),
    19   UDA1341_SINGLE("Treble Playback Volume", CMD_TREBBLE, data0_1, 0,
3, 0),
    20
    21   UDA1341_SINGLE("Input Gain Switch", CMD_IGAIN, stat1, 5, 1, 0),
    22   UDA1341_SINGLE("Output Gain Switch", CMD_OGAIN, stat1, 6, 1, 0),
    23
    24   UDA1341_SINGLE("Mixer Gain Channel 1 Volume", CMD_CH1, ext0, 0, 31,
1),
    25   UDA1341_SINGLE("Mixer Gain Channel 2 Volume", CMD_CH2, ext1, 0, 31,
1),
    26
    27   UDA1341_SINGLE("Mic Sensitivity Volume", CMD_MIC, ext2, 2, 7, 0),
    28
    29   UDA1341_SINGLE("AGC Output Level", CMD_AGC_LEVEL, ext6, 0, 3, 0),
    30   UDA1341_SINGLE("AGC Time Constant", CMD_AGC_TIME, ext6, 2, 7, 0),
    31   UDA1341_SINGLE("AGC Time Constant Switch", CMD_AGC, ext4, 4, 1, 0),
    32
    33   UDA1341_SINGLE("DAC Power", CMD_DAC, stat1, 0, 1, 0),
    34   UDA1341_SINGLE("ADC Power", CMD_ADC, stat1, 1, 1, 0),
    35
    36   UDA1341_ENUM("Peak detection", CMD_PEAK, data0_2, 5, 1, 0),
    37   UDA1341_ENUM("De-emphasis", CMD_DEEMP, data0_2, 3, 3, 0),
    38   UDA1341_ENUM("Mixer mode", CMD_MIXER, ext2, 0, 3, 0),
    39   UDA1341_ENUM("Filter mode", CMD_FILTER, data0_2, 0, 3, 0),
    40
    41   UDA1341_2REGS("Gain Input Amplifier Gain (channel 2)", CMD_IG, ext4,
ext5,0,0,3,31, 0),
    42 };
```

从上述代码中宏 UDA1341_SINGLE 和 UDA1341_ENUM 的定义可知，单值元素的 info()、get()、put() 成员函数分别为 snd_uda1341_info_single()、snd_uda1341_get_single()和 snd_uda1341_put _single()；枚举元素的 info()、get()、put()成员函数分别为 snd_uda1341_info_enum()、snd_ uda1341_get_enum()、和 snd_uda1341_put_enum()。作为例子，代码清单 17.40 给出了单值元素相关函数的实现。

代码清单 17.40　UDA1341 ALSA 驱动控制接口的单值元素 info/get/put 函数

```
    1  static int snd_uda1341_info_single(struct snd_kcontrol *kcontrol,
```

```
struct
    2     snd_ctl_elem_info *uinfo)
    3 {
    4     int mask = (kcontrol->private_value >> 12) &63;
    5
    6     uinfo->type = mask == 1 ? SNDRV_CTL_ELEM_TYPE_BOOLEAN :
    7       SNDRV_CTL_ELEM_TYPE_INTEGER;
    8     uinfo->count = 1;  //数量为1
    9     uinfo->value.integer.min = 0;  //最小值
    10    uinfo->value.integer.max = mask; //最大值
    11    return 0;
    12 }
    13
    14 static int snd_uda1341_get_single(struct snd_kcontrol *kcontrol,
struct
    15    snd_ctl_elem_value *ucontrol)
    16 {
    17    struct l3_client *clnt = snd_kcontrol_chip(kcontrol);
    18    struct uda1341 *uda = clnt->driver_data;
    19    int where = kcontrol->private_value &31;
    20    int mask = (kcontrol->private_value >> 12) &63;
    21    int invert = (kcontrol->private_value >> 18) &1;
    22
    23     ucontrol->value.integer.value[0] = uda->cfg[where]; //返回给
ucontrol
    24    if (invert) //如果反转
    25                ucontrol->value.integer.value[0]   =   mask   -
ucontrol->value.integer.value[0];
    26
    27     return 0;
    28 }
    29
    30 static int snd_uda1341_put_single(struct snd_kcontrol *kcontrol,
struct
    31    snd_ctl_elem_value *ucontrol)
    32 {
    33    struct l3_client *clnt = snd_kcontrol_chip(kcontrol);
    34    struct uda1341 *uda = clnt->driver_data;
    35    int where = kcontrol->private_value &31;
    36    int reg = (kcontrol->private_value >> 5) &15;
    37    int shift = (kcontrol->private_value >> 9) &7;
    38    int mask = (kcontrol->private_value >> 12) &63;
    39    int invert = (kcontrol->private_value >> 18) &1;
    40    unsigned short val;
    41
    42    val = (ucontrol->value.integer.value[0] &mask);//从 ucontrol 获
得值
    43    if (invert) //如果反转
    44    val = mask - val;
```

```
    45
    46    uda->cfg[where] = val;
    47     return snd_uda1341_update_bits(clnt, reg, mask, shift, val,
FLUSH);//更新位
    48 }
```

17.7 PXA255+AC97 ALSA 驱动实例

Intel 公司的 XScale PXA255 是一款基于 ARM 内核技术的嵌入式处理器。它提供了符合 AC97 rev2.0 标准的 AC97 控制单元（ACUNIT）和音频控制连接（AC-Link）。ACUNIT 就是 CODEC 控制器，它通过 AC-Link 连接和控制 AC97 CODEC 芯片，例如 Philips 公司出品的一款符合 AC97 标准的多功能 CODEC 芯片 UCB1400。它不仅是一枚 CODEC 芯片，还集成了触摸和能量管理两个功能模块，在嵌入式系统中应用广泛。

AC-Link 连接了 ACUNIT 和 UCB1400Codec，只要对 ACUNIT 寄存器操作就可以实现同 UCBI400 之间的数据传输。通过 ACUNIT 还可读写 CODEC 内部寄存器，实现对音频采样和混音处理的控制。当然这些读写操作也是经由 AC-Link 传输的。

Intel Xscale PXA255 提供了 16 个 DMA 通道，可以很方便地为外围设备提供数据传送。ACUNIT 也为 CODEC 的立体声输入/输出和话筒输入提供了独立的 16 bit 数据通道。每个通道都有一个专门的 FIFO。

由于它是一个标准的 AC97 设备，因此，其驱动的控制部分实现非常简单，按照 17.4.4 小节的要求，需要实现 AC97 codec 寄存器读写的硬件级函数 pxa2xx_ac97_read() 和 pxa2xx_ac97_ write()，并在模块初始化时注册相关的 AC97 组件，如代码清单 17.41 所示。

代码清单 17.41　PXA255 连接 AC97 codec ALSA 驱动控制组件

```
    1   static unsigned short pxa2xx_ac97_read(struct snd_ac97 *ac97,
unsigned short
    2      reg)
    3   {
    4      unsigned short val = - 1;
    5      volatile u32 *reg_addr;
    6
    7      down(&car_mutex);
    8
    9      /* 设置首/次 codec 空间 */
    10    reg_addr = (ac97->num &1) ? &SAC_REG_BASE: &PAC_REG_BASE;
```


华清远见

```
    11      reg_addr += (reg >> 1);
    12
    13      /* 通过ac97 link读 */
    14      GSR = GSR_CDONE | GSR_SDONE;
    15      gsr_bits = 0;
    16      val =  *reg_addr;
    17      if (reg == AC97_GPIO_STATUS)
    18         goto out;
    19      if (wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_SDONE, 1)
<= 0
    20         && !((GSR | gsr_bits) &GSR_SDONE))
    21      //等待
    22      {
    23        printk(KERN_ERR "%s: read error (ac97_reg=%d GSR=%#lx)\n",
    24          __FUNCTION__,reg, GSR | gsr_bits);
    25        val =  - 1;
    26        goto out;
    27     }
    28
    29     /* 置数据有效 */
    30     GSR = GSR_CDONE | GSR_SDONE;
    31     gsr_bits = 0;
    32     val =  *reg_addr;
    33     /*已经开启另一个周期... */
    34     wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_SDONE, 1);
    35
    36     out: up(&car_mutex);
    37     return val;
    38   }
    39
    40   static void pxa2xx_ac97_write(struct snd_ac97 *ac97, unsigned short
reg,
    41      unsigned short val)
    42   {
    43      volatile u32 *reg_addr;
    44
    45      down(&car_mutex);
    46
    47      /*设置首/次 codec 空间*/
    48      reg_addr = (ac97->num &1) ? &SAC_REG_BASE: &PAC_REG_BASE;
    49      reg_addr += (reg >> 1);
    50
    51      GSR = GSR_CDONE | GSR_SDONE;
```

```
52    gsr_bits = 0;
53    *reg_addr = val; //通过 AC97 link 写
54    if (wait_event_timeout(gsr_wq, (GSR | gsr_bits) &GSR_CDONE, 1)
<= 0
55       && !((GSR | gsr_bits) &GSR_CDONE))
56       printk(KERN_ERR "%s: write error (ac97_reg=%d GSR=%#lx)\n",
57          __FUNCTION__,reg, GSR | gsr_bits);
58
59    up(&car_mutex);
60  }
61
62  static int pxa2xx_ac97_probe(struct platform_device *dev)
63  {
64     struct snd_card *card;
65     struct snd_ac97_bus *ac97_bus;
66     struct snd_ac97_template ac97_template;
67     int ret;
68
69     ret =  - ENOMEM;
70     /* 新建 card */
71       card = snd_card_new(SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1,
THIS_MODULE, 0);
72     if (!card)
73        goto err;
74     card->dev = &dev->dev;
75                  strncpy(card->driver,   dev->dev.driver->name,
sizeof(card->driver));
76
77     /* 构造 PCM 组件 */
78            ret  =  pxa2xx_pcm_new(card,  &pxa2xx_ac97_pcm_client,
&pxa2xx_ac97_pcm);
79     if (ret)
80        goto err;
81
82    /* 申请中断 */
83    ret = request_irq(IRQ_AC97, pxa2xx_ac97_irq, 0, "AC97", NULL);
84    if (ret < 0)
85       goto err;
86
87    ...
88
89     /* 初始化 AC97 bus */
```

```
    90      ret = snd_ac97_bus(card, 0, &pxa2xx_ac97_ops, NULL, &ac97_bus);
    91      if (ret)
    92         goto err;
    93      memset(&ac97_template, 0, sizeof(ac97_template));
    94              ret   =   snd_ac97_mixer(ac97_bus,   &ac97_template,
&pxa2xx_ac97_ac97);
    95      if (ret)
    96         goto err;
    97     ...
    98
    99     /* 注册 card */
    100    ret = snd_card_register(card);
    101    if (ret == 0)
    102   {
    103        platform_set_drvdata(dev, card);
    104        return 0;
    105   }
    106
    107   err: if (card)
    108       snd_card_free(card);
    109   ...
    110
    111    returns ret;
    112 }
```

17.8 总结

音频设备接口包括 PCM、IIS 和 AC97 等，分别适用于不同的应用场合。针对音频设备，Linux 内核中包含了两类音频设备驱动框架，OSS 和 ALSA，前者包含 dsp 和 mixer 字符设备接口，在用户空间的编程中，完全使用文件操作；后者以 card 和组件（PCM、mixer 等）为主线，在用户空间的编程中不使用文件接口而使用 alsalib。

在音频设备驱动中，几乎必须使用 DMA，而 DMA 的缓冲区会被分割成一个一个的段，每次 DMA 操作进行其中的一段。OSS 驱动的阻塞读写具有流控能力，在用户空间不需要进行流量方面的定时工作，但是它需要及时地写（播放）和读（录音），以免出现缓冲区的 underflow 或 overflow。

推荐课程：嵌入式学院-嵌入式 Linux 长期就业班

- 招生简章：http://www.embedu.org/courses/index.htm
- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm

华清远见