第 8 章　Linux 下音频设备编程

本章着重阐述了 Linux 下对音频设备的编程方法。读完本章，读者将了解以下内容：

- 音频信号的数字化和相关概念；
- 音频总线接口 IIS 的控制原理和控制程序；
- Linux 下音频设备编程的特点和操作方法；
- MPlayer 媒体播放器在嵌入式 Linux 上的移植实例。

8.1　音频信号基础

音频信号是一种连续变化的模拟信号，但计算机只能处理和记录二进制的数字信号，而由自然音源得到的音频信号必须经过一定的变换，成为数字音频信号之后，才能送到计算机中做进一步的处理。

8.1.1　数字音频信号

数字音频信号是相对模拟音频信号来说的。声音的本质是波，人能听到的声音的频率范围是 0.02~20kHz。数字音频信号是对模拟信号的一种量化，量化过程如图 8.1 所示。模拟音频信号数字化的典型方法是对时间坐标按相等的时间间隔做采样，对振幅做量化，单位时间内的采样次数称为采样频率。这样，一段声波被数字化后就可以变成一串数值，每个数值对应相应抽样点的振幅值，按顺序将这些数字排列起来就是数字音频信号了。这就是模拟-数字转化（ADC）过程。数字-模拟转化（DAC）过程则相反，将连续的数字按采样时的频率和顺序转换成对应的电压。通俗一点讲，音频 ADC/DAC 就是录音/放音。放音是数字音频信号转换成模拟音频信号，以驱动耳机、功放等模拟设备，而录音则是要将麦克风等产生的模拟音频信号转换成数字音频信号，并最终转换成计算机可以处理的通用音频文件格式。

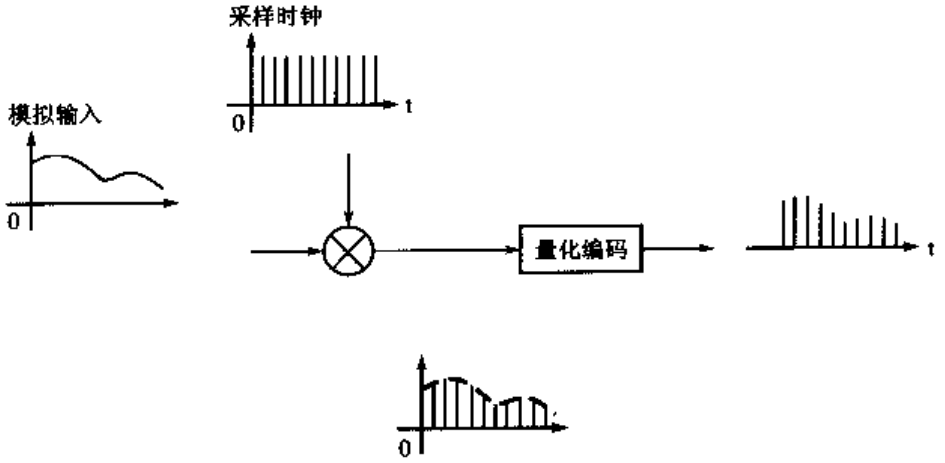

图 8.1　模拟音频信号数字化

数字音频涉及的概念非常多，对于在 Linux 下进行音频编程的程序员来说，最重要的是理解声音数字化的两个关键步骤：采样和量化。采样就是每隔一定时间读一次声音信号的幅度，而量化则是将采样得到的声音信号幅度转换为数字值。从本质上讲，采样是时间上的数字化，而量化则是幅度上的数字化。下面对几个在进行音频编程时经常需要用到的采样频率、量化位数和声道数等指标再进行一下解释。

采样频率是指将模拟声音波形进行数字化时，每秒钟抽取声波幅度样本的次数。采样频率的选择应该遵循奈奎斯特（Nyquist）采样理论：如果对某一模拟信号进行采样，则采样后可还原的最高信号频率只有采样频率的一半，或者说只要采样频率高于输入信号最高频率的两倍，就能从采样信号序列重构原始信号。正常人听觉的频率范围大约在 20Hz~20kHz 之间，根据奈奎斯特采样理论，为了保证声音不失真，采样频率应该在 40kHz 左右。常用的音频采样频率有 8kHz、11.025kHz、22.05kHz、16kHz、37.8kHz、44.1kHz、48kHz 等，如果采用更高的采样频率，还可以达到 DVD 的音质。

量化是对模拟音频信号的幅度进行数字化，量化位数决定了模拟信号数字化以后的动态范围，常用的有 8 位、12 位和 16 位。量化位越高，信号的动态范围越大，数字化后的音频信号就越接近原始信号，但所需要的存储空间也越大。

声道数是反映音频数字化质量的另一个重要因素，它有单声道、双声道和多声道之分。双声道又称为立体声，在硬件中有两条线路，音质和音色都要优于单声道，但数字化后占据的存储空间的大小要比单声道多一倍。多声道能提供更好的听觉感受，不过占用的存储空间也更大。

8.1.2　音频文件格式

1. MP3

MP3 的全称应为 MPEG1 Layer-3 音频文件。MPEG（Moving Picture Experts Group）在汉语中译为活动图像专家组，特指活动影音压缩标准，MPEG 音频文件是 MPEG1 标准中的声音部分，也叫 MPEG 音频层，它根据压缩质量和编码复杂程度划分为三层，即 Layer-1、Layer-2、Layer-3，且分别对应 MP1、MP2、MP3 这三种声音文件，并根据不同的用途，使用不同层次的编码。MPEG 音频编码的层次越高，编码器越复杂，压缩率也越高，MP1 和 MP2 的压缩率分别为 4:1 和 6:1~8:1，而 MP3 的压缩率则高达 10:1~12:1，也就是说，一分钟 CD 音质的音乐，未经压缩需要 10MB 的存储空间，而经过 MP3 压缩编码后只有 1MB 左右。不过 MP3 对音频信号采用的是有损压缩方式，为了降低声音失真度，MP3 采取了“感官编码技术”，即编码时先对音频文件进行频谱分析，然后用过滤器滤掉噪音电平，接着通过量化的方式将剩下的每一位打散排列，最后形成具有较高压缩比的 MP3 文件，并使压缩后的文件在回放时能够达到比较接近原音源的声音效果。

2. WMA

WMA 就是 Windows Media Audio 编码后的文件格式，由微软开发。WMA 针对的不是单机市场，而是网络。它的竞争对手就是网络媒体市场中著名的 Real Networks。微软声称，在只有在 64kbps 的码率情况下，WMA 可以达到接近 CD 的音质。与以往的编码不同，WMA 支持防复制功能，它支持通过 Windows Media Rights Manager 加入保护，可以限制播放时间和播放次数甚至于播放的机器等。由于 WMA 支持流技术，即一边读一边播放，因此 WMA

可以很轻松的实现在线广播。WMA 有着优秀的技术特征，在微软的大力推广下，这种格式被越来越多的人所接受。

3. WAV

这是一种古老的音频文件格式，由微软开发。WAV 文件格式符合 RIFF（Resource Interchange File Format，资源互换文件格式）规范。所有的 WAV 都有一个文件头，这个文件头保存了音频流的编码参数。WAV 对音频流的编码没有硬性规定，除了 PCM 之外，还有几乎所有支持 ACM 规范的编码都可以为 WAV 的音频流进行编码。在 Windows 平台下，基于 PCM 编码的 WAV 是被支持得最好的音频格式，所有音频软件都能完美支持。由于本身可以达到较高的音质的要求，WAV 也是音乐编辑创作的首选格式，适合保存音乐素材。因此，基于 PCM 编码的 WAV 被作为一种中介的格式，常常使用在其他编码的相互转换之中，例如，MP3 转换成 WMA。

4. Ogg Vorbis

OGG 是一个庞大的多媒体开发计划的项目名称，涉及视频音频等方面的编码开发。整个 OGG 项目计划的目的就是向任何人提供完全免费的多媒体编码方案，OGG 的信念就是开源和免费。Vorbis 是 OGG 项目中音频编码的正式命名，目前 Vorbis 已经开发成功，并且开发出了编码器。Ogg Vorbis 是高质量的音频编码方案，官方数据显示：Ogg Vorbis 可以在相对较低的数据速率下实现比 MP3 更好的音质，而且它可以支持多声道。多声道音乐的兴起，给音乐欣赏带来了革命性的变化，尤其在欣赏交响时，会带来更多临场感。这场革命性的变化是 MP3 无法适应的，因为 MP3 只能编码 2 个声道。与 MP3 一样，Ogg Vorbis 是一种灵活开放的音频编码，能够在编码方案已经固定下来后继续对音质进行明显的调节和新算法的改良。因此，它的声音质量将会越来越好。与 MP3 相似，Ogg Vorbis 更像一个音频编码框架，可以不断导入新技术，逐步完善。

5. RA

RA 就是 RealAudio 格式，这是因特网上接触得非常多的一种格式，大部分音乐网站的在线试听都是采用了 RealAudio。这种格式完全针对网络上的媒体市场，支持非常丰富的功能。这种格式最大的特点是可以根据听众的带宽来控制码率，在保证流畅的前提下尽可能提高音质。RA 可以支持多种音频编码，其中包括 ATRAC3。和 WMA 一样，RA 不但支持边读边放，也同样支持使用特殊协议来隐匿文件的真实网络地址，从而实现只在线播放而不提供下载的欣赏方式。这对唱片公司和唱片销售公司很重要，在各方的大力推广下，RA 和 WMA 是目前因特网上，用于在线试听最多的音频媒体格式。

6. APE

APE 是 Monkey's Audio 提供的一种无损压缩格式。由于 Monkey's Audio 提供了 Winamp 的插件支持，因此这就意味着压缩后的文件不再是单纯的压缩格式，而是与 MP3 一样可以播放的音频文件格式。这种格式的压缩比远低于其他格式，但由于能够做到真正无损，因此获得了不少发烧用户的青睐。现在有不少无损压缩方案，APE 是其中有着突出性能的格式，它具有令人满意的压缩比，以及飞快的压缩速度，成为不少朋友私下交流发烧音乐的惟一选择。

7. AAC

AAC（高级音频编码技术，Advanced Audio Coding）是杜比实验室为音乐社区提供的技术，声称最大能容纳 48 通道的音轨，采样率达 96kHz。AAC 在 320kbps 的数据速率下能为 5.1 声道音乐节目提供相当于 ITU-R 广播的品质。AAC 是遵循 MPEG-2 的规格所开发的技术，与 MP3 比起来，它的音质比较好，也能够节省大约 30%的存储空间与带宽。

8. ATRAC 3

ATRAC3（Adaptive Transform Acoustic Coding3）由日本索尼公司开发，是 MD 所采用的 ATRAC 的升级版，其压缩率（约为 ATRAC 的 2 倍）和音质均与 MP3 相当。压缩原理包括同时掩蔽、时效掩蔽和等响度曲线等，与 MP3 大致相同。ATRAC3 的版权保护功能采用的是 OpenMG。目前，对应 ATRAC3 的便携式播放机主要是索尼公司自己的产品。不过，该公司已于 2000 年 2 月与富士通、日立、NEC、Rohm、三洋和 TI 等半导体制造商签署了制造并销售 ATRAC3 用 LSI 的专利许可协议。

8.1.3　WAVE 文件格式剖析

WAVE 文件作为多媒体中使用的声波文件格式之一，是以 RIFF 格式为标准的。RIFF 可以看成是一种树形结构，其基本构成单位为 chunk，犹如树形结构中的节点，每个 chunk 由辨别码、数据大小，以及数据所组成。

WAVE 文件的“RIFF”格式辨别码为“WAVE”，整个文件由两个 chunk 所组成，辨别码分别是“fmt”和“data”。在“fmt”chunk 下包含了一个 PCM 波形格式的数据结构，在此之后是包含原始声音信息的采样数据，这些数据是可以直接送到 IIS 总线的数字音频信号。WAVE 文件各部分内容及格式如表 8.1 所示。

表 8.1　WAVE 文件格式说明表

偏移地址	字节数	数据类型	内　容
00H	4	char	“RIFF”标志
04H	4	long int	文件长度
08H	4	char	“WAVE”标志
0CH	4	char	“fmt”标志
10H	4	过渡字节、类型内容不定	
14H	2	int	格式类别，取值“10H”为 PCM 形式的声音数据
16H	2	int	通道数，单声道为 1，双声道为 2
18H	2	int	采样率，每秒样本数，表示每个通道的播放速度
1CH	4	long in t	波形音频数据传送速率，其值为通道数×每秒数据位数×每样本的数据位数/8。播放软件利用此值可以估计缓冲区的大小
20H	2	int	数据块的调整数（按字节算），其值为通道数×每样本的数据位值/8。播放软件需要一次处理多个该值大小的字节数据，以便将其值用于缓冲区的调整
22H	2		每样本的数据位数，表示每个声道中各个样本的数据位数。如果有多个声道，对每个声道而言，样本大小都一样
24H	4	char	数据标记符“data”
28H	4	long in	语音数据的长度

常见的声音文件主要有两种，分别对应于单声道和双声道。对于单声道声音文件，采样速率是 11.025kHz，采样数据为 8 位的短整数（short int）；而对于双声道立体声声音文件，采样速率为 44.1kHz，每次采样数据为一个 16 位的整数（int），高 8 位和低 8 位分别代表左右两个声道。

WAVE 文件数据块包含以脉冲编码调制（PCM）格式表示的样本。WAVE 文件是由样本组织而成的。在 WAVE 文件中，声道 0 代表左声道，声道 1 代表右声道。在多声道 WAVE 文件中，样本是交替出现的。例如，对于 8 位双声道的立体声，存储数据格式依次为：0 声道（左）、1 声道（右）、0 声道（左）、1 声道（右）。对于 16 位立体声，存储数据依次为：0 声道（左）低字节、0 声道（左）高字节、1 声道（右）低字节、1 声道（右）高字节。

8.2　基于 IIS 接口的音频系统

本节首先介绍 IIS 音频总线接口，并讨论 S3C2410X 内置 IIS 总线控制器的使用方法；然后给出音频编解码器与 IIS 总线的接口电路设计；在此基础上，给出放音和录音的控制程序代码。

8.2.1　IIS 接口控制原理

IIS 总线（Inter-IC Sound bus）由 Philips 公司提出，是一种面向多媒体计算机的串行数字音频总线协议。目前很多音频芯片和 MCU 都提供了对 IIS 的支持。该总线专门用于音频设备之间的数据传输，为立体声音频序列提供一个至标准编解码器的连接。

S3C2410X 内置了一个 IIS 总线控制器，该控制器实现到一个外部 8/16 位立体声音频编解码器接口。支持 IIS 总线数据格式和 MSB-justified 数据格式，能够和其他厂商提供的多媒体编解码芯片配合使用。S3C2410X 中有两条串行数据线，一条是输入信号数据线，一条是输出信号数据线，以同时发送和接收数据。该 IIS 接口能够读取 IIS 总线上的数据，同时也为 FIFO 数据提供 DMA 的传输模式，这样能够同时传送和接收数据。IIS 接口有 3 种工作方式：

- 正常传输模式，正常模式下使用 IISCON 寄存器对 FIFO 进行控制。如果传输 FIFO 缓存为空，IISCON 的第 7 位被设置为“0”，表示不能继续传输数据，需要 CPU 对缓存进行处理。如果传输 FIFO 缓存非空，IISCON 的第 7 位被设置成“1”，表示可以继续传输数据。同样，数据接收时，如果 FIFO 满，标识位是“0”，此时，需要 CPU 对 FIFO 进行处理，如果 FIFO 没有满，那么标志位是“1”，这个时候可以继续接收数据。
- DMA 模式，通过设置 IISFCON 寄存器可以使 IIS 接口工作于这种模式下。在这种模式中，FIFO 寄存器组的控制权掌握在 DMA 控制器上，当 FIFO 满了，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，这种模式下，IIS 数据可以同时接收和发送音频数据。

IIS 总线控制器结构如图 8.2 所示，各功能说明如下：

- 两个 5 比特预除器 IPSR，IPSA_A 用于产生 IIS 总线接口的主时钟，IPSA_B 用做外部 CODEC 时钟产生器。
- 16 字节 FIFO，在发送数据时数据被写进 TxFIFO，在接收数据时数据从 RxFIFO 中读取。
- 主 IISCLK 产生器 SCLKG，在主模式下，有主时钟产生串行位时钟。

- 通道产生器和状态机 CHNC，IISCLK 和 IISLRCK 有通道状态机产生并控制。
- 16 比特移位寄存器（SFTR），在发送数据时，并行数据经由 SFTR 变成串行数据输出；在数据接收时，串行数据由 SFTR 转变成并行数据。

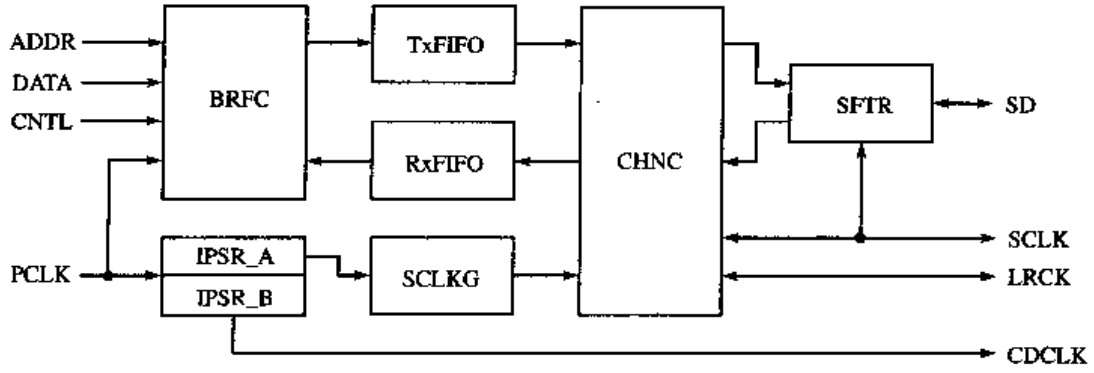

图 8.2　IIS 总线控制器结构图

IIS 相关的寄存器包括 IIS 控制寄存器 IISCON、IIS 模式寄存器 IISMOD 和 IIS 分频寄存器 IISPSR。限于篇幅，只简单描述各个寄存器的意义。

内存控制器为访问外部存储空间提供存储器控制信号，共有 13 个寄存器，如表 8.2 所示。S3C2410X 微处理器的内存控制器在整个系统工作中起至关重要作用，只有清楚地了解内存控制器在系统中的作用及工作原理，才能进行程序设计和系统开发。

表 8.2　IIS 相关寄存器

寄存器	地　址	读/写	说　明	复位后的值
IISCON	0X55000000	读/写	IIS 控制寄存器	0X100
IISMOD	0X55000004		IIS 模式存器	0X000
IISPSR	0X55000008		IIS 分频寄存器	0X000
IISFCON	0X4800000C		FIFO 控制寄存器	0X0000
IISFIFO（FENTRY）	0X48000010		FIFO 寄存器	0X0000

（1）IISCON 控制寄存器。

[8] 左右声道标记，0=左声道，1=右声道；

[7] 发送 FIFO 就绪标记，取 0 时表示没有就绪，取 1 时表示 FIFO 就绪；

[6] 接收 FIFO 就绪标记，取 0 时表示没有就绪，取 1 时表示 FIFO 就绪；

[5] 发送 DMA 请求使能，取 0 时请求禁止，取 1 时请求使能；

[4] 接收 DMA 请求使能，取 0 时请求禁止，取 1 时请求使能；

[3]发送通道空闲命令，在空闲状态（暂停传输）时，IISLRCK 是不激活的，0 表示 IISLRCK 产生，1 表示不产生；

[2]接收通道空闲命令，在空闲状态（暂停接收）时，IISLRCK 是不激活的，0 表示 IISLRCK 产生，1 表示不产生；

[1] IIS 预分频器使能，取 0 时预分频器禁止，取 1 时预分频器使能；

[0] IIS 接口使能，取 0 时 IIS 禁止，取 1 时 IIS 使能。

（2）IISMOD 模式寄存器。

[8] 主从模式选择，取 0 时为主模式，取 1 时为从模式；

[7:6] 发送/接收模式选择，00=无，01=接收模式，10=发送模式，11=发送/接收模式；

[5] 左右通道优先级，取 0 时右通道高左通道低，取 1 时右通道低左通道高；

[4] 串行接口格式，取 0 时 IIS 兼容格式，取 1 时 MSB 可调格式；

[3] 每通道串行数据位，取 0 时 8 位，取 1 时 16 位；

[2] 主时钟频率选择，取 0 时主时钟是 256fs（采样频率），取 1 时为 384fs；

[1:0] 串行位时钟频率选择，00=位时钟是 16fs，01=位时钟是 32fs，10=位时钟是 48fs，11=未定义。

（3）IIS 分频寄存器。

[9:5] A 预分频值，预分频器 A 的除因子，IIS 总线接口主时钟=MCLK/A 预分频因子；

[4:0] B 预分频值，预分频器 B 的除因子，外部 CODEC 时钟=MCLK/B 预分频因子。

（4）IISFCON 寄存器。

[15] 发送 FIFO 访问模式选择，取 0 时工作于普通模式，取 1 时工作在 DMA 模式；

[14] 接收 FIFO 访问模式选择，取 0 时工作于普通模式，取 1 时工作在 DMA 模式；

[13] 控制发送 FIFO 使能，取 1 时使能，取 0 时禁止；

[12] 控制接收 FIFO 使能，取 1 时使能，取 0 时禁止；

[11:6] 发送端 FIFO 数据计数，计数值 0~32；

[5:0] 接收端 FIFO 数据计数，计数值 0~32。

（5）FIFO 寄存器。

IIS 总线接口在发送/接收模式有两个 64 字节的 FIFO，每个 FIFO 由宽 16 位、深度 32 的表组成，并且每个 FIFO 单元可以分别操作高字节或低字节。通过 FIFO 入口 FENTRY 访问发送和接收 FIFO，入口地址为 0X55000010。

8.2.2 音频接口电路设计

UDA1341 是 Philips 公司的一款经济型音频 CODEC，用于实现模拟音频信号的采集和数字音频信号的模拟输出，并通过 IIS 数字音频接口，实现音频信号的数字化处理。

如图 8.3 所示，S3C2410X 的 IIS 总线时钟信号 SCK 与 UDA1341TS 的 BCK 连接，字段选择连接在 WS 引脚上。UDA1341TS 提供了两个音频通道，分别用于输入和输出，对应的引脚连接为：IIS 总线的音频输出 I2SSDO 对应于 UDA1341 的音频输入；IIS 总线的音频输入 I2SSDI 对应于 UDA1341 的音频输出。UDA1341TS 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 GPB2、GPOB3、GPB4 三个通用数据输出引脚上，实现混音控制。

8.2.3 音频接口程序设计

1. 放音

放音程序代码如下：

```
#include "2410addr.h"
#include "2410lib.h"
#include "def.h"
```

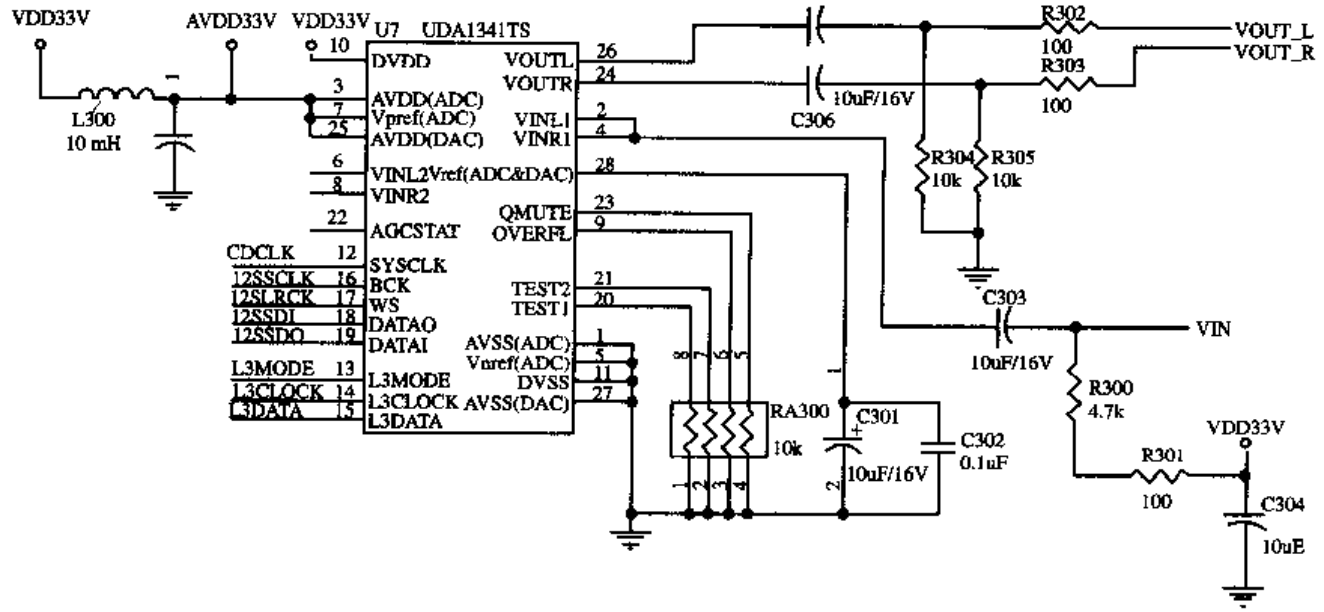

图 8.3　IIS 总线控制器结构图

```
#include "2410iis.h"

void ChangeDMA2(void);
void IIS_PortSetting(void);
void _WrL3Addr(U8 data);
void _WrL3Data(U8 data,int halt);
void __irq DMA2_Done(void);
void __irq DMA2_Rec_Done(void);
void __irq RxInt(void);
void __irq Muting(void);

#define L3C (1<<4)              //GPB4 = L3CLOCK
#define L3D (1<<3)              //GPB3 = L3DATA
#define L3M (1<<2)              //GPB2 = L3MODE

#define PLAY    0
#define RECORD  1
//#define REC_LEN 0x50000         //327,680 Bytes
#define REC_LEN 0x100000        //1,048,576 Bytes

#define DataCount   0x10000     //IIS Master/Slave Data Rx/Tx Count
#define DataDisplay 0x100       //IIS Master Data Display Count

#define PollMode    0           //1: Polling Mode
#define DMA2Mode    1           //1: DMA2 Mode

#define MICGain_Amp_Sel     0 //0: Input channel 2 Amp. 1: MIC Amp.
```

```
unsigned char  *Buf,*_temp;
unsigned short *rec_buf;

volatile unsigned int size = 0;
volatile unsigned int   fs = 0;
volatile char    which_Buf = 1;
volatile char     Rec_Done = 0;
volatile char         mute = 1;

void PlayTest_Iis(void)
{
   unsigned int save_B, save_E, save_PB, save_PE;
   Uart_TxEmpty(0);
```

由于 IIS 时钟从系统分频得到，为了得到合适的 IIS 时钟，必须对系统时钟进行适当的降频处理，下面的代码将系统 PCLK 降到 33MHz，而且降频后必须对串口重新进行初始化。

```
ChangeClockDivider(1,1);        //1:2:4
ChangeMPllValue(0x96,0x5,0x1);  //FCLK=135.428571MHz (PCLK=33.857142MHz)
Uart_Init(33857142,115200);
Uart_Printf("[ IIS test (Play) using UDA1341 CODEC ]\n");
```

然后将用到的端口保存起来，并进行端口初始化。

```
save_B  = rGPBCON;
save_E  = rGPECON;
save_PB = rGPBUP;
save_PE = rGPEUP;
IIS_PortSetting();
```

IIS 采用 DMA 方式进行录音和播放，因此需要进行 DMA 中断的注册。

```
pISR_DMA2  = (unsigned)DMA2_Done;
```

然后获取语音数据及其大小、采样频率。

```
rINTSUBMSK = ~(BIT_SUB_RXD0);
rINTMSK    = ~(BIT_EINT0 | BIT_UART0 | BIT_DMA2);
//Non-cacheable area = 0x31000000 ~ 0x33feffff
Buf   = (unsigned char *)0x31000000;
_temp = Buf;
Uart_Printf("Download the PCM(no ADPCM) file by DNW(With header)!!\n");
size = *(Buf) | *(Buf + 1)<<8 | *(Buf + 2)<<16 | *(Buf + 3)<<24;
Uart_Printf("\nNow, Downloading...",size);
rINTSUBMSK |= BIT_SUB_RXD0;
size = *(Buf + 0x2c) | *(Buf + 0x2d)<<8 | *(Buf + 0x2e)<<16 | *(Buf + 0x2f)<<24;
size = (size>>1)<<1;
fs   = *(Buf + 0x1c) | *(Buf + 0x1d)<<8 | *(Buf + 0x1e)<<16 | *(Buf + 0x1f)<<24;
```

接着初始化 UDA1341，设置为放音模式：

```
Init1341(PLAY);
```

接着进行 DMA 初始化：

```
rDISRC2  = (int)(Buf + 0x30); //0x31000030~(Remove header)
rDISRCC2 = (0<<1) + (0<<0);     //源地址位于系统总线 AHB，地址递增
rDIDST2  = ((U32)IISFIFO);                  //IISFIFO
rDIDSTC2 = (1<<1) + (1<<0); //目的地址位于外设总线 APB，地址固定
rDCON2   = (1<<31)+(0<<30)+(1<<29)+(0<<28)+(0<<27)+(0<<24)+
(1<<23)+(0<<22)+(1<<20)+(size/4);
  //1010 0000 1001 xxxx xxxx xxxx xxxx xxxx
  //Handshake[31],Sync PCLK[30],CURR_TC Interrupt Request[29],
//Single Tx[28], Single service[27],
  //I2SSDO[26:24], DMA source selected[23],Auto-reload[22],
//Half-word[21:20], size/2[19:0]
rDMASKTRIG2 = (0<<2) + (1<<1) + (0<<0); //No-stop[2], DMA2 channel On[1],
No-sw trigger[0]
```

IIS 初始化：

```
if(fs==44100)              //11.2896MHz(256fs)
   rIISPSR = (2<<5) + 2;   //Prescaler A,B=2 <- FCLK 135.4752MHz(1:2:4)
else                      //fs=22050, 5.6448MHz(256fs)
   rIISPSR = (5<<5) + 5;   //Prescaler A,B=5 <- FCLK 135.4752MHz(1:2:4)
rIISCON = (1<<5) + (1<<2) + (1<<1);//Tx DMA enable[5], Rx idle[2],
Prescaler enable[1]
//Master mode[8],Tx mode[7:6],Low for Left Channel[5],IIS format[4],
16bit ch.[3],CDCLK 256fs[2],IISCLK 32fs[1:0]
rIISMOD = (0<<8) + (2<<6) + (0<<5) + (0<<4) + (1<<3) + (0<<2) + (1<<0);
rIISFCON = (1<<15) + (1<<13);       //Tx DMA,Tx FIFO --> start piling....
```

启动 IIS。IIS 启动后，将采用 DMA 方式播放语音数据，播放完毕后将引发中断，并重新播放语音数据。可通过按任意键，决定播放是否结束。

```
//IIS Tx Start
   Uart_Printf("\nPress any key to exit!!!\n");
   rIISCON |= 0x1;           //IIS Interface start
   while(!Uart_GetKey())
   {
   if((rDSTAT2 & 0xfffff) < (size/6))
   ChangeDMA2();
   }
```

其中，ChangeDMA2()函数根据标志位 which_Buf 决定是否重新播放，标志位 which_Buf 在中断服务函数 DMA2_Done()中设置。

语音播放结束后，通知 IIS，并恢复寄存器。

```
Delay(10);                  //For end of H/W Tx
rIISCON     = 0x0;          //IIS Interface stop
rDMASKTRIG2 = (1<<2);       //DMA2 stop
rIISFCON    = 0x0;          //For FIFO flush
size = 0;
rGPBCON = save_B;
rGPECON = save_E;
rGPBUP  = save_PB;
rGPEUP  = save_PE;
```

最后关闭中断，并恢复系统时钟：

```
rINTMSK = (BIT_DMA2 | BIT_EINT0);
ChangeMPllValue(0xa1,0x3,0x1);    // FCLK=202.8MHz
Uart_Init(0,115200);
mute = 1;
}
```

2. 录音

录音程序在初始化等动作上与放音类似，代码如下：

```
void Record_Iis(void)
{
    unsigned int save_B, save_E, save_PB, save_PE;
    Uart_TxEmpty(0);
    ChangeClockDivider(1,1);        //1:2:4
    ChangeMPllValue(0x96,0x5,0x1);  //FCLK=135428571Hz, PCLK=3.385714MHz
    Uart_Init(33857142,115200);
    Uart_Printf("[ Record test using UDA1341 ]\n");
    save_B  = rGPBCON;
    save_E  = rGPECON;
    save_PB = rGPBUP;
    save_PE = rGPEUP;
    IIS_PortSetting();
```

录音数据保存在 rec_buf 中：

```
rec_buf = (unsigned short *)0x31000000;
pISR_DMA2  = (unsigned)DMA2_Rec_Done;
pISR_EINT0 = (unsigned)Muting;
rINTMSK = ~(BIT_DMA2);
Init1341(RECORD);
//--- DMA2 Initialize
rDISRCC2 = (1<<1) + (1<<0);                          //APB, Fix
rDISRC2  = ((U32)IISFIFO);                           //IISFIFO
rDIDSTC2 = (0<<1) + (0<<0);                          //AHB, Increment
rDIDST2  = (int)rec_buf;                             //0x31000000 ~
rDCON2   = (1<<31)+(0<<30)+(1<<29)+(0<<28)+(0<<27)+(1<<24)
```

```
+(1<<23)+(1<<22)+(1<<20)+REC_LEN;
    //Handshake, sync PCLK, TC int, single tx, single service,
//I2SSDI, I2S Rx request,
     //Off-reload, half-word, 0x50000 half word.
rDMASKTRIG2 = (0<<2) + (1<<1) + 0;    //No-stop, DMA2 channel on,
 No-sw trigger
//IIS Initialize
//Master,Rx,L-ch=low,IIS,16bit ch,CDCLK=256fs,IISCLK=32fs
rIISMOD = (0<<8) + (1<<6) + (0<<5) + (0<<4) + (1<<3) + (0<<2) + (1<<0);
rIISPSR = (2<<5) + 2; //Prescaler_A/B=2 <- FCLK
135.4752MHz(1:2:4),11.2896MHz(256fs),44.1KHz
rIISCON = (0<<5) + (1<<4) + (1<<3) + (0<<2) + (1<<1);
//Tx DMA disable,Rx DMA enable,Tx idle,Rx not idle,prescaler enable,stop
rIISFCON = (1<<14) + (1<<12);     //Rx DMA,Rx FIFO --> start piling....
```

开始录音：

```
//Rx start
rIISCON |= 0x1;
```

录音完毕将引发 DMA2 中断，如下代码等待录音结束：

```
while(!Rec_Done);
rINTMSK  = BIT_DMA2;
Rec_Done = 0;
//IIS Stop
Delay(10);                          //For end of H/W Rx
rIISCON     = 0x0;                  //IIS stop
rDMASKTRIG2 = (1<<2);                //DMA2 stop
rIISFCON    = 0x0;                  //For FIFO flush
```

录音完毕，然后播放声音：

```
Uart_Printf("End of Record!!!\n");
Uart_Printf("Press any key to play recorded data\n");
Uart_Printf("If you want to mute or no mute push the 'EIN0' key
repeatedly\n");
Uart_Getch();
size = REC_LEN * 2;
Uart_Printf("Size = %d\n",size);
Init1341(PLAY);
pISR_DMA2 = (unsigned)DMA2_Done;
rINTMSK   = ~(BIT_DMA2 | BIT_EINT0);

//DMA2 Initialize
rDISRCC2 = (0<<1) + (0<<0);                    //AHB, Increment
rDISRC2  = (int)rec_buf;                       //0x31000000
rDIDSTC2 = (1<<1) + (1<<0);                    //APB, Fixed
```

```
rDIDST2  = ((U32)IISFIFO);                          //IISFIFO
rDCON2   = (1<<31)+(0<<30)+(1<<29)+(0<<28)+(0<<27)+(0<<24)
+(1<<23)+(0<<22)+(1<<20)+(size/2);
//Handshake, sync PCLK, TC int, single tx, single service,
I2SSDO, I2S request,
//Auto-reload, half-word, size/2
rDMASKTRIG2 = (0<<2)+(1<<1)+0;    //No-stop, DMA2 channel on,
 No-sw trigger
//IIS Initialize
//Master,Tx,L-ch=low,iis,16bit ch.,CDCLK=256fs,IISCLK=32fs
rIISMOD = (0<<8) + (2<<6) + (0<<5) + (0<<4) + (1<<3) + (0<<2) + (1<<0);
//  rIISPSR = (4<<5) + 4;           //Prescaler_A/B=4 for 11.2896MHz
rIISCON = (1<<5)+(0<<4)+(0<<3)+(1<<2)+(1<<1);
//Tx DMA enable,Tx DMA disable,Tx not idle,Rx idle,prescaler enable,stop
rIISFCON = (1<<15) + (1<<13);      //Tx DMA,Tx FIFO --> start piling....
Uart_Printf("Press any key to exit!!!\n");
rIISCON |= 0x1;                  //IIS Tx Start
while(!Uart_GetKey());
//IIS Tx Stop
Delay(10);                 //For end of H/W Tx
rIISCON     = 0x0;         //IIS stop
rDMASKTRIG2 = (1<<2);      //DMA2 stop
rIISFCON    = 0x0;         //For FIFO flush
size = 0;

rGPBCON = save_B;
rGPECON = save_E;
rGPBUP  = save_PB;
rGPEUP  = save_PE;

rINTMSK = (BIT_DMA2 | BIT_EINT0);
ChangeMPllValue(0xa1,0x3,0x1);              // FCLK=202.8MHz
Uart_Init(0,115200);
mute = 1;
}
```

8.3　音频设备程序的实现

在 Linux 下，音频设备程序的实现与文件系统的操作密切相关。Linux 将各种设备以文件的形式给出统一的接口，这样的设计使得对设备的编程与对文件的操作基本相同，对 Linux 内核的系统调用也基本一致，从而简化了设备编程。

8.3.1　音频编程接口

如何对各种音频设备进行操作是在 Linux 上进行音频编程的关键，通过内核提供的一组

系统调用，应用程序能够访问声卡驱动程序提供的各种音频设备接口，这是在 Linux 下进行音频编程最简单也是最直接的方法。

声卡不是 Linux 控制台的一部分，它是一个特殊的设备。声卡主要提供 3 个重要的特征：

- 数字取样输入/输出；
- 频率调制输出；
- MIDI 接口。

这 3 个特征都有它们自己的设备驱动程序接口，数字取样的接口是/dev/dsp，频率调制的接口/dev/sequencer，而 MIDI 接口是/dev/midi。混音设备（如音量、平衡或者贝斯）可以通过/dev/mixer 接口来控制。为了满足兼容性的需要，还提供了一个/dev/audio 设备，该设备可用于读 SUN_law 的声音数据，但它是映射到数字取样设备的。

程序员可以使用 ioctl()来操作这些设备，ioctl()请求是在 linux/soundcard.h 中定义的，它们以 SNDCTL_开头。从程序员的角度来说，对声卡的操作在很大程度上等同于对磁盘文件的操作：首先使用 open 系统调用建立起与硬件间的联系，此时返回的文件描述符将作为随后操作的标识；接着使用 read 系统调用从设备接收数据，或者使用 write 系统调用向设备写入数据，而其他所有不符合读/写这一基本模式的操作都可以由 ioctl 系统调用来完成；最后，使用 close 系统调用告诉 Linux 内核不会再对该设备做进一步的处理。

1. open 系统调用

系统调用 open 可以获得对声卡的访问权，同时还能为随后的系统调用做好准备，其函数原型如下所示：

```
int open(const char *pathname, int flags, int mode);
```

参数 pathname 是将要被打开的设备文件的名称，对于声卡来讲一般是/dev/dsp。参数 flags 用来指明应该以什么方式打开设备文件，它可以是 O_RDONLY、O_WRONLY 或者 O_RDWR，分别表示以只读、只写或者读写的方式打开设备文件；参数 mode 通常是可选的，它只有在指定的设备文件不存在时才会用到，指明新创建的文件应该具有怎样的权限。

如果 open 系统调用能够成功完成，它将返回一个正整数作为文件标志符，在随后的系统调用中需要用到该标志符。如果 open 系统调用失败，它将返回-1，同时还会设置全局变量 errno，指明是什么原因导致了错误的发生。

2. read 系统调用

系统调用 read 用来从声卡读取数据，其函数原型如下所示：

```
int read(int fd, char *buf, size_t count);
```

参数 fd 是设备文件的标志符，它是通过之前的 open 系统调用获得的；参数 buf 是指向缓冲区的字符指针，它用来保存从声卡获得的数据；参数 count 则用来限定从声卡获得的最大字节数。如果 read 系统调用成功完成，它将返回从声卡实际读取的字节数，通常情况会比 count 的值小一些；如果 read 系统调用失败，它将返回-1，同时还会设置全局变量 errno，来指明是什么原因导致了错误的发生。

3. write 系统调用

系统调用 write 用来向声卡写入数据，其函数原型如下所示：

```
size_t write(int fd, const char *buf, size_t count);
```

系统调用 write 和系统调用 read 在很大程度是类似的，差别只在于 write 是向声卡写入数据，而 read 则是从声卡读入数据。参数 fd 同样是设备文件的标志符，它也是通过之前的 open 系统调用获得的；参数 buf 是指向缓冲区的字符指针，它保存着即将向声卡写入的数据；参数 count 则用来限定向声卡写入的最大字节数。

如果 write 系统调用成功完成，它将返回向声卡实际写入的字节数；如果 write 系统调用失败，它将返回-1，同时还会设置全局变量 errno，来指明是什么原因导致了错误的发生。无论是 read 还是 write，一旦调用之后，Linux 内核就会阻塞当前应用程序，直到数据成功地从声卡读出或者写入为止。

4. ioctl 系统调用

系统调用 ioctl 可以对声卡进行控制，凡是对设备文件的操作不符合读/写基本模式的，都是通过 ioctl 来完成的，它可以影响设备的行为，或者返回设备的状态，其函数原型如下所示：

```
int ioctl(int fd, int request, ...);
```

参数 fd 是设备文件的标志符，它是在设备打开时获得的；如果设备比较复杂，那么对它的控制请求相应地也会有很多种，参数 request 的目的就是用来区分不同的控制请求；通常说来，在对设备进行控制时还需要有其他参数，这要根据不同的控制请求才能确定，并且可能是与硬件设备直接相关的。

5. close 系统调用

当应用程序使用完声卡之后，需要用 close 系统调用将其关闭，以便及时释放占用的硬件资源，其函数原型如下所示：

```
int close(int fd);
```

参数 fd 是设备文件的标志符，它是在设备打开时获得的。一旦应用程序调用了 close 系统调用，Linux 内核就会释放与之相关的各种资源，因此建议在不需要的时候尽量及时关闭已经打开的设备。

8.3.2 音频设备文件

对于 Linux 应用程序员来讲，音频编程接口实际上就是一组音频设备文件，通过它们可以从声卡读取数据，或者向声卡写入数据，并且能够对声卡进行控制，设置采样频率和声道数目等。经常用到的设备文件主要有以下几种。

- /dev/sndstat

设备文件/dev/sndstat 是声卡驱动程序提供的最简单的接口，通常它是一个只读文件，作用也仅仅只限于汇报声卡的当前状态。一般说来，/dev/sndstat 是提供给最终用户来检测声卡的，不宜用于程序当中，因为所有的信息都可以通过 ioctl 系统调用来获得。

- /dev/dsp

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，它对于 Linux 下的音频编程来讲非常重要。向该设备写数据即意味着激活声卡上的 D/A 转换器进行放音，而从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。目前，许多声卡都提供有多个数

字采样设备，它们在 Linux 下可以通过/dev/dsp 等设备文件进行访问。

• /dev/audio

/dev/audio 类似于/dev/dsp，它兼容于 Sun 工作站上的音频设备，使用的是 mu-law 编码方式。由于设备文件/dev/audio 主要出于对兼容性的考虑，所以在新开发的应用程序中最好不要尝试用它，而应该以/dev/dsp 进行替代。对于应用程序来说，同一时刻只能使用/dev/audio 或者/dev/dsp 其中之一，因为它们是相同硬件的不同软件接口。

• /dev/mixer

在声卡的硬件电路中，混音器（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。运行在 Linux 内核中的声卡驱动程序一般都会提供/dev/mixer 这一设备文件，它是应用程序对混音器进行操作的软件接口。

• /dev/sequencer

目前大多数声卡驱动程序还会提供/dev/sequencer 这一设备文件，用来对声卡内建的波表合成器进行操作，或者对 MIDI 总线上的乐器进行控制，通常只用于计算机音乐软件中。

8.3.3　音频设备编程设计

在 Linux 下进行音频编程时，重点在于如何正确地操作声卡驱动程序所提供的各种设备文件。下面介绍两个简单的基于通用框架的音频编程实例，以便大家使用。

1．DSP 编程

DSP 是数字信号处理器（Digital Signal Processor）的简称，它用来进行数字信号处理的特殊芯片，声卡使用它来实现模拟信号和数字信号的转换。声卡中的 DSP 设备实际上包含两个组成部分：在以只读方式打开时，能够使用 A/D 转换器进行声音的输入；而在以只写方式打开时，则能够使用 D/A 转换器进行声音的输出。严格说来，Linux 下的应用程序要么以只读方式打开/dev/dsp 输入声音，要么以只写方式打开/dev/dsp 输出声音，但事实上，某些声卡驱动程序仍允许以读写的方式打开/dev/dsp，以便同时进行声音的输入和输出。

在从 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；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。

在向 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，然后产生出声音。应用程序写入数据的速度同样应该与声卡的采样频率相匹配，过慢的话会产生声音暂停或者停顿的现象，而过快的话又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。

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，默认为 8 位无符号数据、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。通常情况下，在应用程序中打开设备文件/dev/dsp 之后，接着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。

对声卡进行编程时，首先要做的是打开与之对应的硬件设备，这是借助于 open 系统调用来完成的，并且一般情况下使用的是/dev/dsp 文件。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，并且还要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，建议在进行音频编程时只要有可能就尽量使用只读或者只写的方式打开设备文件，因为这样不仅能够充分利用声卡的硬件资源，而且还有利于驱动程序的优化。下面的代码示范了如何以只写方式打开声卡进行放音操作：

```
int handle = open("/dev/dsp", O_WRONLY);
if (handle == -1) {
perror("open /dev/dsp");
return -1;
}
```

运行在 Linux 内核中的声卡驱动程序专门维护了一个缓冲区，其大小会影响到放音和录音时的效果，使用 ioctl 系统调用可以对它的尺寸进行恰当的设置。调节驱动程序中缓冲区大小的操作不是必需的，如果没有特殊的要求，一般采用默认的缓冲区大小就可以了。但需要注意的是，缓冲区大小的设置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。下面的代码示范了怎样设置声卡驱动程序中的内核缓冲区的大小：

```
int setting = 0xnnnnssss;
int result = ioctl(handle, SNDCTL_DSP_SETFRAGMENT, &setting);
if (result == -1) {
    perror("ioctl buffer size");
    return -1;
}
```

在设置缓冲区大小时，参数 setting 实际上由两部分组成，其低 16 位标明缓冲区的尺寸，相应的计算公式为 buffer_size = 2^ssss，即若参数 setting 低 16 位的值为 16，那么相应的缓冲区的大小会被设置为 65536 字节。参数 setting 的高 16 位则用来标明分片（fragment）的最大序号，它的取值范围从 2 到 0x7FFF，其中 0x7FFF 表示没有任何限制。

接下来要做的是设置声卡工作时的声道数目，根据硬件设备和驱动程序的具体情况，可以将其设置为 0（单声道，mono）或者 1（立体声，stereo）。下面的代码示范了应该怎样设置声道数目。

```
int channels = 0; // 0=mono 1=stereo
int result = ioctl(handle, SNDCTL_DSP_STEREO, &channels);
if ( result == -1 ) {
    perror("ioctl channel number");
    return -1;
}
if (channels != 0) {
```

```
    // 只支持立体声
}
```

采样格式和采样频率是在进行音频编程时需要考虑的另一个问题，声卡支持的所有采样格式可以在头文件 soundcard.h 中找到，而通过 ioctl 系统调用则可以很方便地更改当前所使用的采样格式。下面的代码示范了如何设置声卡的采样格式。

```
int format = AFMT_U8;
int result = ioctl(handle, SNDCTL_DSP_SETFMT, &format);
if ( result == -1 ) {
    perror("ioctl sample format");
    return -1;
}
```

声卡采样频率的设置也非常容易，只需在调用 ioctl 时将第二个参数的值设置为 SNDCTL_DSP_SPEED，同时在第三个参数中指定采样频率的数值就行了。对于大多数声卡来说，其支持的采样频率范围一般为 5kHz 到 44.1kHz 或者 48kHz，但并不意味着该范围内的所有频率都会被硬件支持，在 Linux 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。下面的代码示范了如何设置声卡的采样频率。

```
int rate = 22050;
int result = ioctl(handle, SNDCTL_DSP_SPEED, &rate);
if ( result == -1 ) {
    perror("ioctl sample format");
    return -1;
}
```

下面给出一个利用声卡上的 DSP 设备进行声音录制和回放的完整程序，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，然后再进行回放，其所有的功能都是通过读写/dev/dsp 设备文件来完成的。

```
/*
 * sound.c
 */
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/soundcard.h>
#define LENGTH 3    /* 存储秒数 */
#define RATE 8000   /* 采样频率 */
#define SIZE 8      /* 量化位数 */
#define CHANNELS 1  /* 声道数目 */

/* 用于保存数字音频数据的内存缓冲区 */
```

```
unsigned char buf[LENGTH*RATE*SIZE*CHANNELS/8];

int main()
{
  int fd;    /* 声音设备的文件描述符 */
  int arg;   /* 用于 ioctl 调用的参数 */
  int status;   /* 系统调用的返回值 */

  /* 打开声音设备 */
  fd = open("/dev/dsp", O_RDWR);
  if (fd < 0) {
    perror("open of /dev/dsp failed");
    exit(1);
  }

  /* 设置采样时的量化位数 */
  arg = SIZE;
  status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_BITS ioctl failed");
  if (arg != SIZE)
    perror("unable to set sample size");

  /* 设置采样时的声道数目 */
  arg = CHANNELS;
  status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_CHANNELS ioctl failed");
  if (arg != CHANNELS)
    perror("unable to set number of channels");

  /* 设置采样时的采样频率 */
  arg = RATE;
  status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
  if (status == -1)
    perror("SOUND_PCM_WRITE_WRITE ioctl failed");

  /* 循环，直到按下 Control-C */
  while (1) {
    printf("Say something:\n");
    status = read(fd, buf, sizeof(buf)); /* 录音 */
    if (status != sizeof(buf))
      perror("read wrong number of bytes");
    printf("You said:\n");
    status = write(fd, buf, sizeof(buf)); /* 回放 */
```

```
    if (status != sizeof(buf))
      perror("wrote wrong number of bytes");
    /* 在继续录音前等待回放结束 */
    status = ioctl(fd, SOUND_PCM_SYNC, 0);
    if (status == -1)
      perror("SOUND_PCM_SYNC ioctl failed");
  }
}
```

2. Mixer 编程

混音器电路通常由两个部分组成：输入混音器和输出混音器。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后，在不同的混音通道中分别进行调制，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不同通道中的信号与混音器相连，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增益，以及怎样在不同的音源间进行切换，这些操作通常是不连续的，而且不会像录音或者放音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 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。

为了简化应用程序的设计，Linux 上的声卡驱动程序大多都支持将混音器的 ioctl 操作直接应用到声音设备上，也就是说，如果已经打开了/dev/dsp，那么就不用再打开/dev/mixer 来对混音器进行操作，而是可以直接用打开/dev/dsp 时得到的文件标志符来设置混音器。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。对混音器的操作是通过 ioctl 系统调用来完成的，并且所有控制命令都由 SOUND_MIXER 或者 MIXER 开头，表 8.3 列出了常用的几个混音器控制命令。

表 8.3　混音器命令

名　称	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

续表

名　称	作　用
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	回放音量
SOUND_MIXER_ALTPCM	从 D/A 转换器
SOUND_MIXER_RECLEV	录音音量
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，但作为程序员来讲并不需要关心这些，因为声卡驱动程序会负责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 到 100。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，在获取麦克风的输入增益时，可以使用如下的代码：

```
int vol;
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
printf("Mic gain is at %d %%\n", 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;
printf("Left gain is %d %%, Right gain is %d %%\n", left, right);
```

类似地，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，此时遵循的原则与获取增益值时的原则基本相同，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);
```

在编写实用的音频程序时，混音器是在涉及兼容性时需要重点考虑的一个对象，这是因为不同的声卡所提供的混音器资源是有所区别的。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（bitmask），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。例如，通过 SOUND_

MIXER_READ_DEVMASK 返回的位掩码，可以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做录音源的每一个通道。下面的代码可以用来检查 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");
```

如果进一步还想知道其是不是一个有效的录音源，则可以使用如下语句：

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

目前，大多数声卡提供多个录音源，通过 SOUND_MIXER_READ_RECSRC 可以查询出当前正在使用的录音源，同一时刻能够使用几个录音源是由声卡硬件决定的。类似地，使用 SOUND_MIXER_WRITE_RECSRC 可以设置声卡当前使用的录音源，例如，下面的代码可以将 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。

下面再给出一个简单的混音器控制程序，利用它可以对各种混音通道的增益进行调节，其所有的功能都是通过读写/dev/mixer 设备文件来完成的。

```
/*
 * mixer.c
 */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/soundcard.h>

/* 用来存储所有可用混音设备的名称 */
const char *sound_device_names[] = SOUND_DEVICE_NAMES;

int fd;                 /* 混音设备所对应的文件描述符 */
int devmask, stereodevs; /* 混音器信息对应的位图掩码 */
char *name;

/* 显示命令的使用方法及所有可用的混音设备 */
void usage()
{
  int i;
```

```
    fprintf(stderr, "usage: %s <device> <left-gain%%> <right-gain%%>\n"
        "       %s <device> <gain%%>\n\n"
        "Where <device> is one of:\n", name, name);
    for (i = 0 ; i < SOUND_MIXER_NRDEVICES ; i++)
      if ((1 << i) & devmask) /* 只显示有效的混音设备 */
        fprintf(stderr, "%s ", sound_device_names[i]);
    fprintf(stderr, "\n");
    exit(1);
}

int main(int argc, char *argv[])
{
    int left, right, level;  /* 增益设置 */
    int status;              /* 系统调用的返回值 */
    int device;              /* 选用的混音设备 */
    char *dev;               /* 混音设备的名称 */
    int i;

    name = argv[0];

    /* 以只读方式打开混音设备 */
    fd = open("/dev/mixer", O_RDONLY);
    if (fd == -1) {
      perror("unable to open /dev/mixer");
      exit(1);
    }
    /* 获得所需要的信息 */
    status = ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
    if (status == -1) perror("SOUND_MIXER_READ_DEVMASK ioctl failed");
    status = ioctl(fd, SOUND_MIXER_READ_STEREODEVS, &stereodevs);
    if (status == -1)  perror("SOUND_MIXER_READ_STEREODEVS ioctl failed");

    /* 检查用户输入 */
    if (argc != 3 && argc != 4)   usage();

    /* 保存用户输入的混音器名称 */
    dev = argv[1];

    /* 确定即将用到的混音设备 */
    for (i = 0 ; i < SOUND_MIXER_NRDEVICES ; i++)
     if(((1 << i)& devmask) && !strcmp(dev, sound_device_names[i]))   break;
    if (i == SOUND_MIXER_NRDEVICES) { /* 没有找到匹配项 */
      fprintf(stderr, "%s is not a valid mixer device\n", dev);
      usage();
    }
```

```
    /* 查找到有效的混音设备 */
    device = i;
    /* 获取增益值 */
    if (argc == 4) {
      /* 左、右声道均给定 */
      left  = atoi(argv[2]);
      right = atoi(argv[3]);
    } else {
      /* 左、右声道设为相等 */
      left  = atoi(argv[2]);
      right = atoi(argv[2]);
    }
      /* 对非立体声设备给出警告信息 */
    if ((left != right) && !((1 << i) & stereodevs)) {
      fprintf(stderr, "warning: %s is not a stereo device\n", dev);
    }
      /* 将两个声道的值合到同一变量中 */
    level = (right << 8) + left;
  /* 设置增益 */
    status = ioctl(fd, MIXER_WRITE(device), &level);
    if (status == -1) {
      perror("MIXER_WRITE ioctl failed");
      exit(1);
    }

    /* 获得从驱动返回的左右声道的增益 */
    left  = level & 0xff;
    right = (level & 0xff00) >> 8;

    /* 显示实际设置的增益 */
    fprintf(stderr, "%s gain set to %d%% / %d%%\n", dev, left, right);

    /* 关闭混音设备 */
    close(fd);
    return 0;
}
```

编译好上面的程序之后，先不带任何参数执行一遍，此时会列出声卡上所有可用的混音通道。之后可以很方便地设置各个混音通道的增益大小了。例如，下面的命令就能够将 CD 输入的左、右声道的增益分别设置为 80%和 90%。

```
$ ./mixer cd 80 90
cd gain set to 80% / 90%
```

8.4　综合训练之媒体播放器移植

MPlayer 是 Linux 下强大的媒体播放器，对媒体格式广泛支持，最新的版本可以支持 Divx、H.264、MPEG4 等最新的媒体格式，可以实时在线播放视频流，是目前嵌入式媒体播放器的首选。MPlayer 资源占用率极低、支持格式极广、输出设备支持极多，同时更为诱人的是它可以让 VCD 上损坏的 MPEG 文件播放更流畅。

MPlayer 支持相当多的媒体格式，无论是在音频播放方面还是在视频播放方面，可以说它支持的格式是相当全面的。它可以支持 MPEG、AVI、ASF 与 WMV、QuickTime 与 OGG/OGM、SDP、PVA、GIF 等视频格式，以及 MP3、WAV、OGG/OGM 文件（Vorbis）、WMA 与 ASF、MP4、CD 音频、XMMS 等音频格式。

MPlayer 支持广泛的输出设备，可以在 X11、Xv、DGA、OpenGL、SVGAlib、fbdev、AAlib、DirectFB 下工作，而且也能支持 GGI、SDL（由此可以使用它们支持的各种驱动模式）和一些低级的硬件相关的驱动模式（比如，Matrox、3Dfx 和 RADEON、Mach64、Permedia3）。同时，MPlayer 还支持通过硬件 MPEG 解码卡显示，例如，DVB 和 DXR3 与 Hollywood+。

1．安装和编译

MPlayer 的源代码可以从其主页 http://www.mplayerhq.hu 下载。打开 MPlayer 官方网站的主页，可以看到“download”链接，单击即进入下载页面。在该页面里可以看到“daily CVS snapshot source”的下载部分，这部分链接指向了每天提交到 CVS 服务器里最新的 MPlayer 源代码包，鼠标单击一下开始下载。最新的文件名是 MPlayer-current.tar.bz2。对下载的文件解压缩：

```
$tar zxvf MPlayer-current.tar.bz2
```

在解压缩得到的 MPlayer-0.93 目录下有一个脚本文件 mkall，这个文件是一个编译脚本，在该目录下直接执行：

```
$ ./ mkall
```

该脚本将配置并编译 mplayer，下面是该脚本所进行的配置和编译命令：

```
./configure --cc=usr/local/arm/2.95.3/bin/arm-linux-gcc
--target=arm-linux -with-extralibdir=/usr/local/arm/2.95.3/arm-linux/lib
-with-extraincdir=/usr/ local/arm/2.95.3/arm-linux/include/--disable-sdl
-enable-static -disable- dvdnav -disable-tv -disable-gui -disable-mpdvdkit
-enable-linux-devfs
make
```

编译成功后，将在 MPlayer-0.93 目录下生成 mplayer 文件，该文件为 mplayer 媒体播放程序。

2．下载运行

启动目标板后，以网络移动的方式下载应用程序。首先把 mplayer 复制到 ftp 共享目录，在 PC 端执行：

```
#cp mplayer /home/ftp
```

在 SBC-2410X 端进入 bin 目录，并登录 ftp 服务器：

```
#cd /bin
#ftp 192.168.0.1
```

然后下载 mplayer 和测试视频文件，并修改 mplayer 可执行权限：

```
>get mplayer
>get test.mpeg
>bye
#chmod a+x mplayer
```

最后在命令行下输入如下命令：

```
#mplayer -vo test.mpeg
```

练习题

1. 简述模拟语音信号转换成数字信号的过程。
2. 什么是采样？
3. 什么是量化？
4. 音频文件主要有哪几种格式？
5. 分析 WAVE 文件中的数据组织格式。
6. S3C2410X 支持哪几种工作方式，如何来实现。
7. 分析 8.2 节中的录音、放音程序。
8. 音频设备文件有哪几种？
9. 设计实现录音放音功能的 DSP 程序。
10. 设计实现音量调节的 Mixer 程序。