

# 北京理工大学

## 本科生毕业设计（论文）

跨操作系统的音频驱动模块设计与实现

Cross-Platform Audio Driver Module Design and  
Implementation

|       |            |
|-------|------------|
| 学 院：  | 计算机学院      |
| 专 业：  | 计算机科学与技术   |
| 班 级：  | 07112005   |
| 学生姓名： | 穆新宇        |
| 学 号：  | 1120202695 |
| 指导教师： | 陆慧梅        |

2024 年 5 月 16 日

## 原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名：

日期：

年

月

日

## 关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名：

日期：

年

月

日

指导老师签名：

日期：

年

月

日

# 跨操作系统的音频驱动模块设计与实现

## 摘 要

本文提出了一种基于 VirtIO 协议的跨操作系统的音频驱动模块的设计与实现方案。

在设计方面，使用 VirtIO v1.2<sup>[1]</sup>文档提供的协议接口，利用已有的跨操作系统驱动程序框架 virtio-drivers<sup>[2]</sup>，向其中添加对 virtio-sound 设备的支持。

在实现方面，充分利用 Rust 语言的内存安全和高度抽象等特性，使得编写的驱动程序简单易读，同时具有强大的扩展性，即便未来有针对于 VirtIO sound 设备的功能扩展，此实现也能轻易完成同步更新。在此实现中，没有用到任何与标准库相关联的模块与结构体，即与操作系统无关。因此只需要在对应操作系统源码的 Cargo.toml 中引入此实现的依赖，使用驱动程序提供的接口，就能完成对 VirtIO sound 设备的驱动与控制。

**关键词：**北京理工大学；本科生；毕业设计（论文）；驱动程序；声卡驱动；Rust；virtio；操作系统；qemu

## **Cross-Platform Audio Driver Module Design and Implementation**

### **Abstract**

This article presents a design and implementation solution for a cross-operating system audio driver module based on the VirtIO protocol.

In terms of design, the protocol interface provided by VirtIO v1.2<sup>[1]</sup> documentation is utilized, and support for the virtio-sound device is added to the existing cross-operating system driver framework, virtio-drivers<sup>[2]</sup>.

In terms of implementation, the driver program is written in Rust, making full use of the language's features such as memory safety and high-level abstraction. This ensures that the driver program is simple, readable, and highly extensible. Even if there are future feature extensions specific to the VirtIO sound device, this implementation can easily accommodate synchronous updates. This implementation does not rely on any modules or structures associated with the standard library, making it independent of the operating system. Therefore, by simply including the dependency of this implementation in the Cargo.toml of the corresponding operating system's source code and using the interfaces provided by the driver program, the driver and control of the VirtIO sound device can be achieved.

**Key Words:** BIT; Undergraduate; Graduation Project (Thesis); Driver; Sound Card Driver; Rust; virtio; Operating System; qemu

## 目 录

|  |    |
|--|----|
| 摘 要 .....                                      | I  |
| Abstract .....                                 | II |
| 第 1 章 引言 .....                                 | 1  |
| 1.1 背景 .....                                   | 2  |
| 1.2 毕业设计目标 .....                               | 3  |
| 第 2 章 virtio .....                             | 4  |
| 2.1 virtio 概述 .....                            | 4  |
| 2.2 virtio 虚拟队列 (Virtqueue) .....              | 5  |
| 2.2.1 描述符表 (Descriptor Table) .....            | 5  |
| 2.2.2 可用环 (Available Ring) .....               | 6  |
| 2.2.3 已用环 (Used Ring) .....                    | 6  |
| 2.3 通用 virtio 驱动程序 .....                       | 7  |
| 2.3.1 设备初始化 .....                              | 7  |
| 2.3.2 设备操作 .....                               | 8  |
| 2.3.3 设备清理 .....                               | 8  |
| 第 3 章 数字音频基础 .....                             | 9  |
| 3.1 PCM .....                                  | 9  |
| 3.2 数字音频基本概念 .....                             | 10 |
| 3.2.1 采样率 (Sample Rate) .....                  | 10 |
| 3.2.2 采样位数 (Bit Depth) .....                   | 10 |
| 3.2.3 声道 (Channel) .....                       | 10 |
| 3.3 ALSA .....                                 | 11 |
| 3.3.1 PCM 帧 (PCM frame) .....                  | 12 |
| 3.3.2 period 和 buffer .....                    | 12 |
| 第 4 章 VirtIO Sound Device 概要 .....             | 14 |
| 4.1 虚拟队列 (Virtqueues) .....                    | 14 |
| 4.2 设备配置空间 (Device Configuration Layout) ..... | 14 |
| 4.3 PCM 生命周期 .....                             | 15 |
| 4.4 设备操作 .....                                 | 16 |
| 4.4.1 配置信息查询 .....                             | 17 |
| 4.4.2 插孔重新映射 (Jack Remap) .....                | 18 |

|   |    |
|---|----|
| 4.4.3 设置 PCM 流的参数 .....                   | 18 |
| 4.4.4 PCM 帧的传输 .....                      | 19 |
| 第 5 章 实现 .....                            | 20 |
| 5.1 整体架构 .....                            | 21 |
| 5.2 初始化 .....                             | 21 |
| 5.2.1 构建结构体 .....                         | 21 |
| 5.2.2 读取配置信息 .....                        | 24 |
| 5.3 获取输出/输入流 .....                        | 26 |
| 5.4 流的基本信息查询 .....                        | 26 |
| 5.5 插孔重新映射 (Jack Remap) .....             | 27 |
| 5.6 设置流的参数 .....                          | 28 |
| 5.7 PCM 帧的传输 .....                        | 29 |
| 第 6 章 测试 .....                            | 31 |
| 6.1 裸机 .....                              | 31 |
| 6.2 Alien .....                           | 32 |
| 6.2.1 Alien 简介 .....                      | 32 |
| 6.2.2 在 Alien 中测试 virtio sound 驱动程序 ..... | 32 |
| 第 7 章 未来可能的改进 .....                       | 35 |
| 7.1 支持 VirtIO v1.3 .....                  | 35 |
| 7.2 异步改进 .....                            | 35 |
| 结 论 .....                                 | 37 |
| 参考文献 .....                                | 38 |
| 附 录 .....                                 | 39 |
| 附录 A 英文缩略词表 .....                         | 39 |
| 致 谢 .....                                 | 40 |

## 第1章 引言

设备驱动程序是在操作系统和设备之间提供一种接口抽象的特殊软件<sup>[3]</sup>，它通常运行于操作系统的内核态，以便发出内核态的 IO 指令，来控制设备读写。实际上，甚至可以说设备驱动程序出现在操作系统之前，那时它们是以可以调用的子程序库的形式存在，例如远古计算机 ENIAC 需要与磁盘和磁带等外设进行复杂的交互，负责给 ENIAC 提供这些外设接口的程序，就可以说是对应设备的驱动程序。<sup>[4]</sup>

传统的驱动程序指的是针对某一物理上真实存在的设备编写的系统软件，但是自从虚拟化的概念出现以后，“驱动程序”这种“驱动某一外设工作的软件”的外沿扩展到了虚拟设备，即通过某些虚拟化技术模拟出来的设备，这些设备在物理上并不存在，例如使用 VMware Workstation 安装 Linux 虚拟机时，虚拟机所要面对的底层“硬件”就是通过 VMware VMI 虚拟化技术模拟出来的“假设备”。

在云计算时代，面对这些虚拟设备并不是一件稀罕的事，Linux 内核代码中就有专门为这些虚拟设备编写的驱动程序，这时 Linux 系统是以 Guest OS 的形式被用户使用的，如图1-1所示。在图1-1中，驱动程序被分为了两个部分——驱动前端

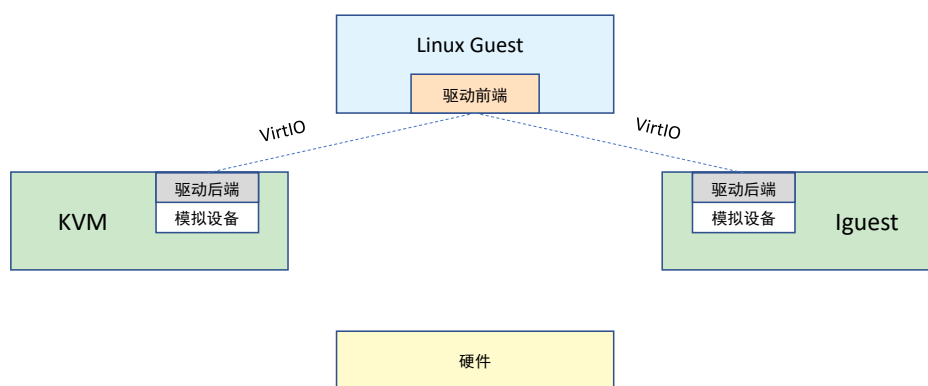


图 1-1 VirtIO 协议下的系统架构<sup>[5]</sup>

（Front-end Drivers）和驱动后端（Back-end Drivers），它们之间使用 VirtIO 协议进行

通信。

驱动后端通常由 KVM、QEMU 等虚拟化解决方案实现，它为虚拟机（图1-1中的 Linux Guest）提供设备接口，将虚拟机对设备的请求提供到虚拟化框架，并完成对实际物理设备的操作。

驱动前端一般由 Guest 操作系统实现，例如 Linux 内核中就包含了针对 VirtIO 协议实现的驱动前端代码<sup>[6]</sup>，这里的驱动前端类似传统的驱动程序，它负责给操作系统提供设备接口。

当 Guest 操作系统要访问外设时，会调用前端驱动提供的接口，前端驱动获取操作系统传来的参数后，基于 VirtIO 协议与驱动后端进行通信，驱动后端结合虚拟化框架将抽象的设备请求转为具体的、针对宿主操作系统的“真正的”驱动程序的需求，再由这个“真驱动程序”来完成具体的设备操作。

### 1.1 背景

正如前文所述，驱动前端一般是由 Guest 操作系统实现，这就出现了一个问题：Linux 内核中实现的 VirtIO 驱动前端无法在其它操作系统中使用。不仅如此，由于 Linux 内核使用 C 语言编写，它的 VirtIO 驱动大量依赖于 Linux 内核的其他模块，简单地对这些 C 代码进行移植是一件十分困难的工作。因此对于一个新开发的操作系统，无法简单地将其作为 Guest OS 运行在各种虚拟化框架上——因为需要重写各种设备的驱动前端——尽管这是一种“重复造轮子”的工作。

对于某一个版本的 VirtIO 协议，驱动前端需要实现的操作是固定的，只与 VirtIO 协议有关，而与操作系统无关，如果每出现一个新的操作系统，就要实现一次已经实现过的驱动前端，对于商业公司和开发人员来说，无疑是一种巨大的资源浪费。

为了减轻开发人员的负担，亟需一种跨操作系统的驱动前端实现，这个实现应该具有“开箱即用、即插即用”的功能，让新操作系统可以快速运行在虚拟化框架上，完成预期功能的验证。

所幸的是，Rust 社区已经实现了跨操作系统的驱动前端框架——virtio-drivers<sup>[2]</sup>。目前实现了 console、network、gpu、input、socket、block 虚拟设备的驱动，并且有些驱动已经在 rCore（一个小的教学操作系统）中得到验证。

本文第二章首先介绍了 virtio 协议，包括虚拟队列等关键概念和 virtio 驱动程序在实现时的公用部分；然后在第三章针对数字音频中的 PCM (Pulse Code Modulation,



脉冲调制编码) 做了一个粗浅的探究, 本章也简要介绍了 ALSA, 主要论述其与项目实现之间的关系; 第四章以 virtio v1.2 为蓝本, 针对 virtio sound 设备驱动应当实现的功能进行详细分析; 第五章从驱动程序实现的不同方面对其实现进行了详细的论述, 与第四章之间存在一定的对应关系; 为了验证实现的正确性, 第六章给出了分别在裸机和 Alien 操作系统上测试的过程, 并对结果进行了总结; 当然, 本实现目前还有一些可以改进的地方, 在第七章对本实现的缺点和改进方案做了一定的剖析。

### 1.2 毕业设计目标

本毕业设计的预期目标如下:

- 实现 virtio sound 虚拟设备的跨操作系统驱动前端
- 在裸机上进行正确性验证
- 在 Alien 上进行正确性验证

## 第 2 章 virtio

虚拟化技术的历史远比我们想象的要久远，早在 2008 年，Linux 内核就已经支持了至少 8 中虚拟化系统，例如 KVM，VMware 的 VMI，用户态 Linux 等等。虚拟化系统的多样性导致一系列问题出现，尤其是，每种虚拟化解方案都拥有各自的独特的驱动实现，这对于考虑兼容性的系统无疑是一件十分令人头疼的事。

IBM Ozlabs 的 Rusty Russell 在探索虚拟机中设备驱动程序性能和效率的改进问题时提出了 virtio——一系列高效的 Linux 驱动程序，它们通过一个 shim 层来适配到各种虚拟化平台。<sup>[7]</sup>

从 Linux 内核 2.6.25 版本开始，virtio 驱动就被包括在内核中，并且可以通过内核模块加载进行使用。随着时间的推移，virtio 驱动在后续的内核版本中得到了改进和增强，以提供更好的性能和功能。也渐渐地成为了事实上的虚拟 IO 设备标准（a De-Facto Standard For Virtual I/O Devices）。

### 2.1 virtio 概述

virtio 可以简单地划分为三层，如图2-1所示。

最上层是前端驱动，运行在 Guest OS 上，它按照 virtio 标准规定的格式（主要是结构体字段）对请求封装，通过传输协议将请求发送到后端，同时前端驱动也应具备读取后端发来的响应的功能。

中间层是传输协议，使用虚拟队列（Virtqueue）来完成，对于不同的设备，可能有多个虚拟队列，每个虚拟队列有不同的职责，例如control\_queue、event\_queue等。

虚拟队列使用 vring 来实现，vring 本质上是一个环形缓冲区，是 Guest OS 与 Hypervisor 之间的一块共享内存，前端负责放入数据，后端负责读取数据。

最下层是后端驱动，由 Hypervisor 实现，从虚拟队列中接收来自前端的信息，按照 virtio 标准进行解析，完成对物理设备的操作，再将结果通过虚拟队列返回给前端。

后端驱动主要由 Hypervisor 实现，超出了这篇文章所要讨论的范围，这里我们主要把目光聚集在前端驱动的实现上。在认识前端驱动之前，首先需要对传输协议——也就是虚拟队列，有一个简单的了解。

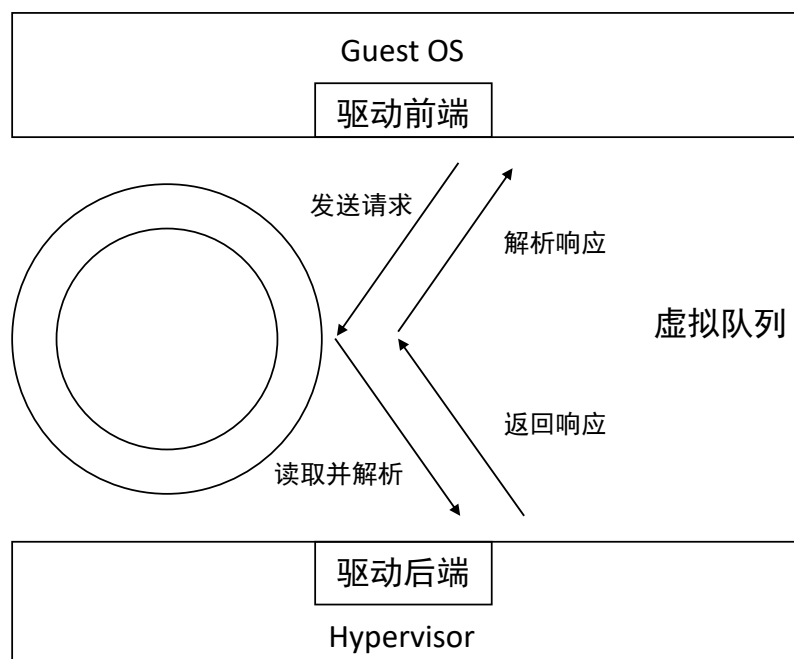


图 2-1 virtio 的三个层次

## 2.2 virtio 虚拟队列 (Virtqueue)

虚拟队列是对 virtio 中间传输协议一层的抽象，主要提供 `add_buf`、`get_buf`、`disable_cb`、`enable_cb` 这五种接口，这种抽象表示通常使用 `vring` 来实现，`vring` 包括描述符表、可用环、已用环这三个部分。

### 2.2.1 描述符表 (Descriptor Table)

描述符表是描述符为组成元素的数组，描述符由描述了一个内存 buffer 的 “address/length” 对组成。描述符的 C 语言描述类似图 2-2：

```

1 struct vring_desc {
2     __u64 addr;
3     __u32 len;
4     __u16 flags;
5     __u16 next;
6 };

```

图 2-2 描述符的 C 语言描述

每个描述符包含 Guest OS 的

- buffer 的物理地址
- buffer 的长度
- next 指针，用于指示下一个描述符地址的
- 两个 flags，一个 flag 表示 next 的值是否有效，另一个 flag 表示这个 buffer 的读写属性（只写或只读）

描述符的读写属性允许描述符表中的描述符指向具有不同读写属性的 buffer，便于实现某些虚拟设备（例如块设备，每个磁盘块的读写属性可能不同）。多个描述符链接在一起组成描述符链，一个描述符链可以表示一个完整的 IO 操作。

### 2.2.2 可用环 (Available Ring)

可用环由自由运行的索引 (free-running index)，中断抑制标志，和指向描述符表的索引数组组成。<sup>[7]</sup>它的 C 语言描述如图2-3所示。

---

```
1 struct vring_avail {  
2     __u16 flags;  
3     __u16 idx;  
4     __u16 ring[NUM];  
5 };
```

---

图 2-3 可用环的 C 语言描述

比较重要的是ring[]字段，它记录了被 virtio 设备驱动程序更新的描述符索引的集合，以供 virtio 设备读取来确定需要完成的 IO 操作，并作出相应响应。

### 2.2.3 已用环 (Used Ring)

已用环和可用环类似，但在使用描述符链时由 host 操作系统写入。它的 C 语言描述如图2-4所示。

与可用环相反，已用环的主要功能是记录 virtio 设备完成 IO 操作后更新的描述符索引集合，这些描述符等待设备驱动程序处理。

```
1 struct vring_used_elem {
2     __u32 id;
3     __u32 len;
4 };
5 struct vring_used {
6     __u16 flags;
7     __u16 idx;
8     struct vring_used_elem ring[];
9 };
```

---

图 2-4 已用环的 C 语言描述

## 2.3 通用 virtio 驱动程序

virtio 协议标准单独定义了 virtio 驱动程序的通用部分，包括设备初始化，设备操作和设备清理（Device Cleanup），这里简单做个介绍。

### 2.3.1 设备初始化

virtio 标准规定，设备驱动程序必须按照下面的步骤来完成设备初始化<sup>[1]</sup>：

1. 重置设备。
2. 设置状态域为ACKNOWLEDGE，表示 Guest OS 已经发现了这个虚拟设备。
3. 设置状态域为DRIVER，表示 Guest OS 知道如何驱动该设备。
4. 读取设备特征位（features bit），并将操作系统和驱动程序理解的特征位的子集写入设备。在此过程中，驱动程序可以读取（但不得写入）设备特定的配置字段，以检查是否可以支持该设备。
5. 设置状态域为FEATURES\_OK，在这之后驱动程序不再接受新的 feature。
6. 重新读取设备状态域，以确保状态域是FEATURES\_OK；如果不是，就表明设备不支持驱动程序设置的特征位，设备此时不可用。
7. 设备初始化的通用部分结束，接下来进行设备有关的初始化工作，包括设备虚拟队列的发现，每条总线的配置（可选），读取和可能写入设备的 virtio 配置空间，以及填充 virtqueues。

8. 设置状态域为DRIVER\_OK，此时设备是“live”的。

如果上述任何步骤出现无法恢复的错误，驱动程序应将状态域设置为FAILED，以指示已放弃该设备（如果需要，可以在稍后重置设备以重新启动）。

### 2.3.2 设备操作

在操作设备时，设备配置空间中的每个字段都可以由驱动程序或设备更改。每当设备触发此类配置更改时，驱动程序都会收到通知。这使得驱动程序能够缓存设备配置，除非收到通知，否则避免进行昂贵的配置空间读取。

### 2.3.3 设备清理

一旦驱动程序设置了DRIVER\_OK状态域，设备的所有已配置 virtqueue 都被视为活动状态。设备重置后，设备的任何 virtqueue 都不会处于活动状态。

驱动程序不得更改已暴露缓冲区的 virtqueue 条目，即已提供给设备且设备尚未使用的实时的 virtqueue 的缓冲区。因此，驱动程序在删除已暴露的缓冲区之前，必须通过设备重置确保 virtqueue 不处于活动状态。

### 第3章 数字音频基础

音频信号分为两种，即模拟（Analog）信号和数字（Digital）信号。模拟信号是连续变化的，例如将声音的物理波形变化转变为磁带上的磁场强度变化，这样就实现了声音的存储；数字信号是离散的，在计算机中不可能有无限多个连续变化的值，因此需要对模拟的过程简化，将连续变化的模拟信号转变为有限个表示声音信息的值，方便在计算机中存储和处理。

那么如何将模拟信号转变为数字信号呢？最常见的方法就是脉冲编码调制（Pulse Code Modulation, PCM）。

#### 3.1 PCM

PCM 的完整过程分为三步，分别是：采样、量化、编码。图3-1描述了 PCM 的这三个过程。

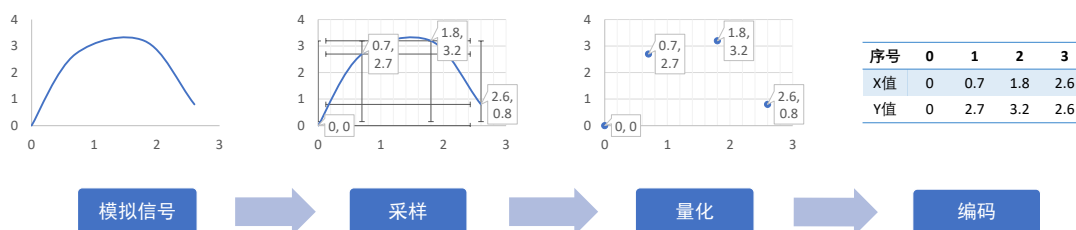


图 3-1 脉冲编码调制（PCM）的过程

简单来讲，采样就是按照一定的时间间隔获取模拟信号的值；量化就是对得到的样本值进行一定处理；编码就是按照一定的方式将量化后的样本值转换为便于计算机存储的 01 序列。

## 3.2 数字音频基本概念

### 3.2.1 采样率 (Sample Rate)

视频实际上是将看到的影响变成一系列静止的图片，这些静止的图片连在一起就变成了我们看到的运动的画面，每秒采样的图片次数叫做帧率。

视频可以如此，那么在声音领域，有没有一种方法，按照一定的间隔采样，用一系列静止的“点”来描绘连续运动的声音呢？答案是可以的。

每秒采样的次数越多，在时空间隔上这些点就越密集，就越与原来的声音接近，这样通过这一系列的点就可以连成声波曲线。

- 这些点被称为“采样点”。
- 每秒钟采样的次数称为采样率，单位是赫兹 (Hz)。<sup>[8]</sup>

根据奈奎斯特采样定理（式3.2.1），要想由数字信号还原模拟信号，采样频率至少是原始模拟信号中最高频率的两倍。人类能听到的最高声音频率是 20000Hz，频率超过 20000Hz 的声音被称为超声波，人耳听不到，因此要想保留人耳能听到的声音细节，采样频率至少是 40000Hz，常见的 CD 的采样频率是 44100Hz，基本刚好达到这个水准。

$$f_s \geq 2f_m \quad (3-1)$$

### 3.2.2 采样位数 (Bit Depth)

采样位数就是采样值的二进制编码的位数。例如如果使用 3bit 来记录声音的强度信息，就有  $2^3 = 8$  中可能的采样结果，用 8bit 的话，就有  $2^8 = 256$  种可能的采样结果。显然，采样位数越多，记录采样点声音信息的能力越强，记录得就越精细，保真度就越高。

### 3.2.3 声道 (Channel)

声道指的是在采样时不同空间位置采集的不同的音频信号数量，简单来说就是录制时的麦克风的数量。常见的声道数为单声道和立体声，前者录制和播放是只有



一个麦克风和扬声器，后者是很常见的音频声道数，在日常生活中使用耳机听音乐就是采用立体声（但是前提是音频本身是双声道的）。

此外，还有多声道音频，如 5.1 声道、7.1 声道等。这些多声道格式通过增加额外的声道，如中央声道、环绕声道和低频效果声道，可以提供更加沉浸式的音频体验，适用于家庭影院系统、游戏和音频制作等领域。

### 3.3 ALSA

ALSA (Advanced Linux Sound Architecture, 高级 Linux 音频体系结构) 是 Linux 操作系统中提供音频和 MIDI 功能的软件架构。它是一个模块化的系统，负责处理音频设备的驱动程序和接口，使应用程序能够与音频硬件进行通信并播放、录制和处理音频数据。

ALSA 的主要目标是提供高效、灵活和可靠的音频功能。它支持各种类型的音频接口，从普通的消费者声卡到专业的多通道音频接口。ALSA 还具有多线程和对称多处理器（SMP）支持，可以在多个应用程序同时访问音频设备而不会出现冲突。

除了底层的驱动程序和接口，ALSA 还提供了用户空间库（alsa-lib），简化了应用程序开发过程，并提供了更高级别的功能，如音频数据的采集、回放、混合和音频效果处理等。

此外，ALSA 还提供对旧版 Open Sound System (OSS) API 的支持，以确保与使用 OSS 的旧程序的二进制兼容性。<sup>[9]</sup>

在对驱动程序的正确性进行测试时，需要在 qemu 中指定宿主机的音频后端<sup>[10]</sup>，这里我们使用 ALSA，如图3-2所示。

---

```
# examples/riscv/Makefile
qemu-system-$(arch) \
  $(QEMU_ARGS) \
    -machine virt \
  # ...
  -device virtio-sound-device,audiodev=audio0 \
  -audiodev alsa,id=audio0 # 音频后端指定为 ALSA
```

---

图 3-2 在 qemu 中指定宿主机的音频后端

### 3.3.1 PCM 帧 (PCM frame)

一个 PCM 帧 (PCM frame) 的大小等于一个要播放的样本 (sample) 的大小, 主要和通道数和采样深度有关, 例如

- 一帧立体声 48kHz 16 位 PCM 流占据  $2 \times 16\text{bit} = 4 \text{ Bytes}$
- 一帧 5.1 声道 48KHz 16 位 PCM 流占据  $6 \times 16\text{bit} = 12 \text{ Bytes}$

### 3.3.2 period 和 buffer

一个 period 是每个硬件中断之间的帧数。buffer 是一个环形缓冲区, 它的大小一般为 period 的 2 倍。但是有些硬件也会使用非整数倍 period 的大小。<sup>[9]</sup>

这两个概念在设置 PCM 流的参数时需要用到, 如图3-3所示。

---

```
371 // src/device/sound.rs
372 /// Set selected stream parameters for the specified stream ID.
373 pub fn pcm_set_params(
374     &mut self,
375     stream_id: u32,
376     buffer_bytes: u32,
377     period_bytes: u32,
378     // ...
379 ) -> Result {
380     // ...
381 }
```

---

---

```
210 // examples/riscv/src/main.rs
211 sound
212     .pcm_set_params(
213         output_stream_id,
214         441 * 2,
215         441,
216         // ...
217     )
```

---

图 3-3 设置 PCM 流的 period bytes 和 buffer bytes

在图3-3所示的代码中, examples/riscv/src/main.rs的第 211 行使用了驱动程

序提供的`pcm_set_params`方法,设置`period_bytes`为 441 ,`buffer_bytes`为  $441 \times 2 = 882$ , 表示设备每读取 441 个字节发生一次中断, 在 `virtio` 标准中表现为设备向驱动发送`hdr`字段为`VIRTIO_SND_EVT_PCM_PERIOD_ELAPSED`的 notification。<sup>[1]</sup>

## 第 4 章 VirtIO Sound Device 概要

VIRTIO v1.2<sup>[1]</sup>于 2022 年 7 月 1 日发布，相较于 1.1 版本，新增了很多虚拟设备的定义，其中就包括 virtio sound 设备，VIRTIO 标准对 sound device 的虚拟队列、配置空间、设备初始化和设备操作进行了详细的论述。本章基于 VIRTIO v1.2 定义的 sound device，对实现其驱动程序之前应当了解的信息进行简要的总结。

### 4.1 虚拟队列（Virtqueues）

sound device 包含 4 个虚拟队列，分别是control\_queue(0)，event\_queue(1)，tx\_queue(2)以及rx\_queue(3)。每个队列后面的数字代表它们的队列编号。

它们的功能如下：

- control\_queue用于从驱动程序向设备发送控制消息。
- event\_queue用于驱动程序接受来自设备的通知。
- tx\_queue用于向输出流发送 PCM 帧。
- rx\_queue用于从输入流接收 PCM 帧。

### 4.2 设备配置空间（Device Configuration Layout）

virtio sound 的设备配置空间包含三个字段，如图4-1所示。

其中，

- **jacks** 表示所有可用插孔的总数。

---

```
1055 // src/device/sound.rs
1056 #[derive(Debug, Clone, Copy)]
1057 struct VirtIOSoundConfig {
1058     jacks: ReadOnly<u32>,
1059     streams: ReadOnly<u32>,
1060     chmaps: ReadOnly<u32>,
1061 }
```

---

图 4-1 virtio sound 的设备配置空间

- **streams** 表示所有可用 PCM 流的总数。
- **chmaps** 表示所有可用通道映射的总数。<sup>[1]</sup>

### 4.3 PCM 生命周期

一个 PCM 流有以下生命周期阶段，将一个 PCM 流设置为对应的生命周期可以通过驱动程序的同名方法实现。

1. **SET PARAMETERS**: 驱动程序和设备协商流的参数（采样率，采样位数等）。
2. **PREPARE**: 设备为流做准备（资源分配等）。
3. 仅输出：驱动程序在预缓冲时传输数据。
4. **START**: 设备开始播放/接收一个流（取消静音等）。
5. 驱动程序向流传输数据（输出流），从流中接收数据（输入流）。
6. **STOP**: 设备停止一个流（静音等）。
7. **RELEASE**: 设备释放一个流（释放资源等）。

上面标注的大写粗体英文斜体表明了 PCM 流可能处于的状态，它们之间可能的状态转换如图4-2所示。

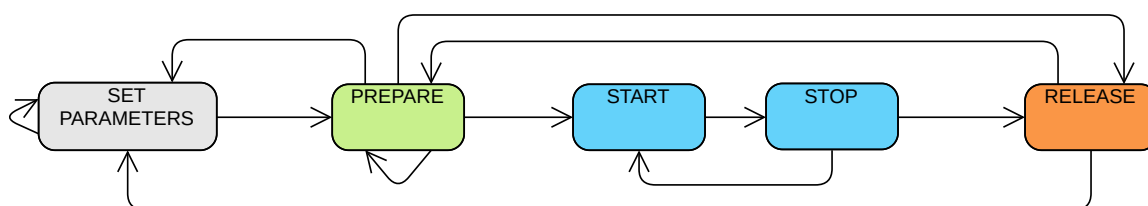


图 4-2 PCM 生命周期转换图

在实现时，用枚举表示 PCM 流的可能状态，进行状态转换时，利用 Rust 宏来简化生命周期的转换<sup>[1]</sup>，如图4-3所示。

---

```

726 // src/device/sound.rs
727 macro_rules! set_new_state {
728     ($new_state_fn:ident, $new_state:expr,
729     ↪ $($valid_source_states:tt)*) => {
729         fn $new_state_fn(&mut self) -> PCMStateResult<()> {
730             if !matches!(self, $($valid_source_states)*) {
731                 return Err(StreamError::InvalidStateTransition(*self,
732                 ↪ $new_state));
732             }
733             *self = $new_state;
734             Ok(())
735         }
736     };
737 }

```

---

图 4-3 生命周期转换的实现，使用宏来减少重复代码

## 4.4 设备操作

virtio 标准规定，对于 virtio sound 设备，所有的控制信息都应当通过控制队列传输；所有的通知都应当通过事件队列传输。

无论是控制信息还是通知，都需要一个通用头部（common header），它的定义如图4-4所示。

---

```

1145 // src/device/sound.rs
1146 /// A common header
1147 #[repr(C)]
1148 #[derive(AsBytes, FromBytes, FromZeroes, PartialEq, Eq)]
1149 struct VirtIOSndHdr {
1150     command_code: u32,
1151 }

```

---

图 4-4 VirtIOSndHdr 的定义，用于表明控制消息的类型

在VirtIOSndHdr中，command\_code表示操作的类型，包括配置信息查询、PCM流的控制、通知类型和请求的返回值等。

#### 4.4.1 配置信息查询

一种特殊的控制消息是对设备配置空间中的字段的查询,整个请求的构成如图4-5所示。

---

```
1196 // src/device/sound.rs
1197 #[repr(C)]
1198 #[derive(AsBytes, FromBytes, FromZeroes)]
1199 struct VirtIOSndQueryInfo {
1200     hdr: VirtIOSndHdr,
1201     start_id: u32,
1202     count: u32,
1203     size: u32,
1204 }
```

---

图 4-5 配置信息查询的请求结构体实现

在图4-5中, `hdr` 字段表明了要查询的配置字段类型, 可选值有 `VirtioSndRJackInfo`, `VirtioSndRPcmInfo` 和 `VirtioSndRChmapInfo`, 分别表示对插孔信息, PCM 流信息, 通道映射信息的查询。 `start_id` 表示查询的起始位置, `count` 表示要查询的个数, `size` 表示设备返回的结构体的字节数。

例如, 要查询所有的 PCM 流的信息, 应当按照图4-6设置各个字段的值。

---

```
1 VirtIOSndQueryInfo {
2     hdr: CommandCode::VirtioSndRPcmInfo.into(),
3     start_id: 0, // 从第 0 个 PCM 流开始
4     count: self.streams, // 查询所有的 PCM 流, self.streams 是从配置空
      ↪ 间读取的 PCM 流的个数
5     size: mem::size_of::<VirtIOSndPcmInfo>() as u32 // PCM 流信息的结
      ↪ 构体大小
6 }
```

---

图 4-6 查询所有 PCM 流的信息

#### 4.4.2 插孔重新映射 (Jack Remap)

对于设备的某一插孔，如果该插孔支持重新映射（即它的features包含 VIRTIO\_SND\_JACK\_F\_REMAP），则可以使用VIRTIO\_SND\_R\_JACK\_REMAP作为hdr来请求对其进行重新映射，请求结构体如图4-7所示。

---

```
1278 // src/device/sound.rs
1279 #[repr(C)]
1280 #[derive(AsBytes, FromBytes, FromZeroes)]
1281 struct VirtIOSndJackRemap {
1282     hdr: VirtIOSndJackHdr,
1283     association: u32,
1284     sequence: u32,
1285 }
```

---

图 4-7 插孔重新映射请求结构体

其中，association指定了所选的关联号码，sequence指定了所选的序列号码。

#### 4.4.3 设置 PCM 流的参数

可以为给定的流设置参数，例如采样率、采样深度等，请求结构体如图4-8所示。

---

```
1278 #[repr(C)]
1279 #[derive(AsBytes, FromBytes, FromZeroes)]
1280 struct VirtIOSndPcmSetParams {
1281     hdr: VirtIOSndPcmHdr, /* .code = VIRTIO_SND_R_PCM_SET_PARAMS */
1282     buffer_bytes: u32,
1283     period_bytes: u32,
1284     features: u32, /* 1 << VIRTIO_SND_PCM_F_XXX */
1285     channels: u8,
1286     format: u8,
1287     rate: u8,
1288
1289     _padding: u8,
1290 }
```

---

图 4-8 设置流的参数的请求结构体



其中，`buffer_bytes`和`period_bytes`的含义在3.3.2中已经解释过；中间的几个字段表明声道数、采样位数采样率等信息，也很好理解；最后的`_padding`没有实际意义，仅仅是为了结构体对其而设置，在具体实现中，具有类似功能的`_padding`有很多处。

### 4.4.4 PCM 帧的传输

PCM 流的 IO 消息由头部、缓冲区、状态部组成（注意，这里只是逻辑上的组成，实际上在实现中，这三个部分位于不同的内存单元，是分开存储的），用 C 语言描述如图4-9所示。

---

```
1  /* IO 消息头部 */
2  struct virtio_snd_pcm_xfer {
3      le32 stream_id
4  };
5  /* IO 消息状态部 */
6  struct virtio_snd_pcm_status {
7      le32 status,
8      le32 latency_bytes
9  }
```

---

图 4-9 PCM 流的 IO 消息组成<sup>[1]</sup>

状态部中的`status`字段表示传输状态，如果其值是`VIRTIO_SND_S_OK`表明传输操作成功，是`VIRTIO_SND_S_IO_ERR`表明传输操作失败；`latency_bytes`字段表示当前设备的传输延迟。

## 第5章 实现

本章主要介绍 virtio sound 驱动程序使用 Rust 语言的实现，所用到的依赖如图5-1所示。

---

```
16 # Cargo.toml
17 [dependencies]
18 log = "0.4"
19 bitflags = "2.3.0"
20 zerocopy = { version = "0.7.5", features = ["derive"] }
21
22 num_enum = { version = "0.7.2", default-features = false}
```

---

图 5-1 项目使用的库

使用 Rust 实现 virtio sound 驱动程序具有以下优势：

- **内存安全性**：Rust 在编译时通过所有权、借用和生命周期规则来保障内存安全。使用zerocopy crate可以在零成本的情况下进行内存布局和数据传输,而alloc库则提供了堆内存分配功能。这使得在驱动程序中可以避免许多常见的内存错误，如空指针引用、缓冲区溢出等。
- **高效的数据处理**：zerocopy crate 允许在驱动程序中进行零拷贝数据处理，避免了不必要的数据复制和内存移动,提高驱动程序的性能和效率。
- **强大的枚举和位标志支持**：num\_enum和bitflags crate 提供了强大的枚举和位标志操作支持。枚举可以帮助驱动程序开发者更好地组织和表示各种状态和选项，而位标志可以简化对位字段的操作。
- **no\_std 环境支持**：Rust 的no\_std环境允许在没有标准库的情况下进行开发，这对于驱动程序开发非常有用。通过使用alloc库，可以进行堆内存分配，并使用其他核心功能，如core模块中的数据结构和算法。

## 5.1 整体架构

virtio sound 驱动程序主要由虚拟队列、配置空间字段、配置信息、缓冲区以及 PCM 状态表组成，如图5-2所示。

---

```
1 pub struct VirtIOSound<H: Hal, T: Transport> {
2     transport: T,
3     // 4 个虚拟队列
4     control_queue: VirtQueue<H, { QUEUE_SIZE as usize }>,
5     // ...
6     // 协商之后的特征
7     negotiated_features: SoundFeatures,
8     // 配置空间读取的字段
9     jacks: u32,
10    streams: u32,
11    chmaps: u32,
12    // 每个 PCM 流的信息，这里省略了 jack 和 chmap 的信息
13    pcm_infos: Option<Vec<VirtIOSndPcmInfo>>,
14    // ...
15    // PCM 流设置的参数
16    pcm_parameters: Vec<PcmParameters>,
17    // 用于控制消息的缓冲区
18    queue_buf_send: Box<[u8]>,
19    queue_buf_recv: Box<[u8]>,
20    // 设置是否完成初始化（读取配置信息）
21    set_up: bool,
22    // ...
23 }
```

---

图 5-2 驱动程序的整体架构

## 5.2 初始化

根据 virtio 标准，virtio sound 驱动程序的初始化分为两个阶段，分别是**构建结构体阶段**和**读取配置信息阶段**。

### 5.2.1 构建结构体

这一阶段主要包括虚拟队列的初始化、读取配置空间字段、分配缓冲区以及初始化所有 PCM 流的状态。

### 5.2.1.1 虚拟队列的初始化

virtio sound 驱动程序一共有四个虚拟队列，它们的队列编号为 0~3，使用编译期常量来记录队列编号和队列大小，如图5-3所示。

---

```
768 // src/device/sound.rs
769 const QUEUE_SIZE: u16 = 32;
770 const CONTROL_QUEUE_IDX: u16 = 0;
771 const EVENT_QUEUE_IDX: u16 = 1;
772 const TX_QUEUE_IDX: u16 = 2;
773 const RX_QUEUE_IDX: u16 = 3;
```

---

图 5-3 队列编号和队列大小的定义

在初始化各个虚拟队列时，将对应的虚拟队列号传入，通过检查驱动程序和设备协商之后的negotiated\_features比特位，来确定队列是否支持对应的features，如图5-4所示（这里以control\_queue为例）。

VirtQueue::new方法接受 4 个参数，第一个参数是实现了Transport trait 的对象；第二个参数是创建的虚拟队列的队列编号；第三个参数是个布尔值，表明是否使用非直接的描述符(indirect descriptors)，如果协商之后的特征包含VIRTIO\_F\_INDIRECT\_DESC，该参数应当设置为 true；最后一个参数是event\_idx，也是一个布尔值，表明是否使用 used\_event 和 avail\_event 来启用通知抑制（notification suppression），同样地，如果协商之后的特征包含VIRTIO\_F\_EVENT\_IDX，其值应当设置为 true。

---

```
64 // src/device/sound.rs
65 let control_queue = VirtQueue::new(
66     &mut transport,
67     CONTROL_QUEUE_IDX,
68     negoti-
69     ated_features.contains(SoundFeatures::VIRTIO_F_INDIRECT_DESC),
70     negotiated_features.contains(SoundFeatures::VIRTIO_F_EVENT_IDX),
71 )?;
```

---

图 5-4 初始化control\_queue

### 5.2.1.2 读取配置空间字段

virtio sound 的配置空间组成在4.2已经介绍过，这里使用 virtio-drivers 框架提供的volread!宏从设备中读取，如图5-5所示。

---

```
92 // src/device/sound.rs
93 let config_ptr = transport.config_space::<VirtIOSoundConfig>()?;
94 let (jacks, streams, chmaps) = unsafe {
95     (
96         volread!(config_ptr, jacks),
97         volread!(config_ptr, streams),
98         volread!(config_ptr, chmaps),
99     )
100 };
```

---

图 5-5 读取配置空间

每个volread!宏接受两个参数，第一个参数是设备配置空间的地址；第二个参数是要读取的字段名。在图5-5中，每个volread!都返回读取的值，三个值组合在一起以元组的形式返回。这里利用了 Rust 的模式匹配来存储返回的值。

### 5.2.1.3 分配缓冲区

使用fromzero的FromZeroes::new\_box\_slice\_zeroed方法来初始化各个缓冲区，每个缓冲区的大小设置为 virtio-drivers 框架设置的页大小PAGE\_SIZE，如图5-6所示。

---

```
105 // src/device/sound.rs
106 // 用于控制消息的缓冲区
107 let queue_buf_send = FromZeroes::new_box_slice_zeroed(PAGE_SIZE);
108 let queue_buf_recv = FromZeroes::new_box_slice_zeroed(PAGE_SIZE);
109 // 用于接受 PCM 传输相应的缓冲区
110 let output_rsp = FromZeroes::new_box_slice_zeroed(PAGE_SIZE);
111 // 用于从设备接收通知的缓冲区
112 let event_buf = FromZeroes::new_box_slice_zeroed(PAGE_SIZE);
```

---

图 5-6 初始化缓冲区

#### 5.2.1.4 初始化 PCM 流的状态

读取配置空间发生在初始化 PCM 流的状态之前，因此现在已经知道了 virtio sound 支持的流的数量（在图5-5中所示的代码中获取），对于每个 PCM 流，初始化其状态为SetParams，如图5-7所示。

---

```
119 // src/device/sound.rs
120 let mut pcm_states = vec![];
121 for _ in 0..streams {
122     pcm_states.push(PCMState::default());
123 }
```

---

图 5-7 初始化 PCM 流的状态为SetParams

#### 5.2.2 读取配置信息

上述的构建结构体阶段实际上在实现VirtIOSound::new方法，注意到在构建结构体时，pcm\_infos、jack\_infos、chmap\_infos这三个字段都初始化为None，只有调用set\_up方法后，配置信息才会被写入结构体，在这之后上面的存储配置信息的三个字段才不为None。

查询配置信息已经在4.4.1一节中介绍过，这里以查询 PCM 流信息为例，简要介绍读取配置信息的实现。

在图5-8中，第7行的request\_hdr指定了读取配置信息的类型——VirtioSndRPcmInfo；第8行向设备发送VirtIOSndQueryInfo请求，请求的参数来自于函数的参数和刚刚构建的request\_hdr；设备的返回值存放在queue\_buf\_recv字段中，返回值由一个头部和stream\_count个 PCM 流的配置信息组成，在第8行我们用一个hdr变量读取头部，来判断请求是否成功（这里省略了判断的代码）；紧接着在第12~22行，我们创建了一个pcm\_infos变量，它是一个Vec<VirtIOSndPcmInfo>，用于记录每个PCM流的信息；在for循环中，我们利用fromzero提供的FromBytes、FromZero宏，从self.queue\_buf\_recv对应位置构造VirtIOSndPcmInfo结构体，放入pcm\_infos。

最后，在set\_up函数中，我们调用pcm\_info，参数设置成配置空间中的 PCM 流的数量，这样就能获取全部流的信息并保存，如图5-9所示。

```
1 fn pcm_info(  
2     &mut self,  
3     stream_start_id: u32,  
4     stream_count: u32,  
5 ) -> Result<Vec<VirtIOSndPcmInfo>> {  
6     // ...  
7     let request_hdr = VirtIOSnd-  
8         ↪ Hdr::from(ItemInfomationRequestType::VirtioSndRPcmInfo);  
9     let hdr: VirtIOSndHdr = self.request(VirtIOSndQueryInfo {  
10         // ...  
11     })?;  
12     // ...  
13     let mut pcm_infos = vec![];  
14     for i in 0..stream_count as usize {  
15         const HDR_SIZE: usize = mem::size_of::<VirtIOSndHdr>();  
16         const PCM_INFO_SIZE: usize =  
17             ↪ mem::size_of::<VirtIOSndPcmInfo>();  
18         let start_byte_idx = HDR_SIZE + i * PCM_INFO_SIZE;  
19         let end_byte_idx = HDR_SIZE + (i + 1) * PCM_INFO_SIZE;  
20         let pcm_info = VirtIOSndPcmInfo::read_from(  
21             &self.queue_buf_recv[start_byte_idx..end_byte_idx]  
22         ).unwrap();  
23         pcm_infos.push(pcm_info);  
24     }  
25     Ok(pcm_infos)
```

---

图 5-8 读取 PCM 流的配置信息

```
fn set_up(&mut self) {  
    // ...  
    if let Ok(pcm_infos) = self.pcm_info(0, self.streams) {  
        // ... 打印每个 PCM 流的信息  
        self.pcm_infos = Some(pcm_infos);  
    } else {  
        self.pcm_infos = Some(vec![]);  
        warn!("[sound device] There are no available streams!");  
    }  
}
```

---

图 5-9 在set\_up函数中调用pcm\_info，获取流的信息

查询插孔信息和通道映射信息的实现和查询 PCM 流的信息的实现类似，只需替换图5-8中第 7 行的`request_hdr`的类型，这里不再赘述。

### 5.3 获取输出/输入流

`virtio sound` 可能提供了多个 PCM 流，其中有些是输入流，有些是输出流，用户需要获知哪些`stream_id`对应的流是输入流，哪些又是输出流。因此，我们提供了`output_streams`和`input_streams`方法来获取所有的输出流和输入流的集合。

---

```
600 // src/device/sound.rs
601 pub fn output_streams(&mut self) -> Vec<u32> {
602     // ...set_up
603     self.pcm_infos
604         .as_ref()
605         .unwrap()
606         .iter()
607         .enumerate()
608         .filter(|(_, info)| info.direction == VIRTIO_SND_D_OUTPUT)
609         .map(|(idx, _)| idx as u32)
610         .collect()
611 }
```

---

图 5-10 获取所有输出流

在图5-10所示的代码中，充分利用 Rust 提供的迭代器操作，快速获取了所有的输出流。

### 5.4 流的基本信息查询

驱动程序需要为用户提供查询流的基本信息的功能（例如流支持的采样率和采样深度等），因此这里实现了四个流的基本信息查询的方法，分别是`rates_supported`、`formats_supported`、`channel_range_supported`和`features_supported`，它们都只接受一个`stream_id`作为参数，获取对应流的采样率、采样深度、通道范围和支持的特征，它们的函数签名类似图5-11所给出的示例。

在实现时，只需查询`self.pcm_infos`中对应`stream_id`位置的`VirtIOSndPcmInfo`即可。



---

```

pub fn rates_supported(&mut self, stream_id: u32) -> Result<PcmRate> {
    // ... 省略了错误处理的代码
    Ok(self.pcm_infos.as_ref().unwrap()[stream_id as usize]
        .rate
        .into())
}

```

---

图 5-11 流的基本信息查询，以rates\_supports为例

## 5.5 插孔重新映射 (Jack Remap)

插孔重新映射的方法签名如下：

```

pub fn jack_remap(&mut self, jack_id: u32, association: u32, sequence:
    ↪ u32) -> Result

```

首先要根据jack\_id获取到对应插孔支持的 features, 只有支持VIRTIO\_SND\_JACK\_F\_REMAP的插孔才能进行重新映射; 如果选定的插孔支持, 只需向设备发送头部为VirtioSndRJackRemap的请求即可, 如图5-12所示。

---

```

325 // src/device/sound.rs
326 pub fn jack_remap(&mut self, jack_id: u32, association: u32, sequence:
    ↪ u32) -> Result {
327     // ... 判断指定插孔是否支持重新映射
328     let hdr: VirtIOSndHdr = self.request(VirtIOSndJackRemap {
329         hdr: VirtIOSndJackHdr {
330             hdr: CommandCode::VirtioSndRJackRemap.into(),
331             jack_id,
332         },
333         association,
334         sequence,
335     })?;
336     // ... 后续处理
337 }

```

---

图 5-12 插孔重新映射请求

```
372 // src/device/sound.rs
373 pub fn pcm_set_params(
374     // ...
375 ) -> Result {
376     // ... 构建 VirtIOSndPcmSetParams 请求并发送，结果储存在 rsp 中
377     if rsp == VirtIOSndHdr::from(RequestStatusCode::VirtioSndSOk) {
378         // 保存设置的参数
379         self.pcm_parameters[stream_id as usize] = PcmParameters {
380             ↪ setup: true, buffer_bytes, period_bytes, features,
381             ↪ channels, format, rate };
382         Ok(())
383     } else {
384         Err(Error::IoError)
385     }
386 }
```

图 5-13 设置流的参数并保存

## 5.6 设置流的参数

在向输出流传输 PCM 帧之前，需要为选定的输出流设置正确的参数（和 PCM 帧的参数一致），否则播放的音乐会出现噪声、频率降低、乱码等现象。

设置指定流的参数的方法签名如下：

```
pub fn pcm_set_params(&mut self, stream_id: u32, buffer_bytes: u32,
    ↪ period_bytes: u32, features: PcmFeatures, channels: u8, format:
    ↪ PcmFormats, rate: PcmRate) -> Result
```

每个参数对应VirtIOSndPcmSetParams结构体的同名字段，它们的含义在4.4.3已介绍过。

除了发送VirtIOSndPcmSetParams，还需要将设定的参数存储在驱动程序的pcm\_parameters字段，便于在传输 PCM 帧的时候查阅使用，如图5-13所示。

## 5.7 PCM 帧的传输

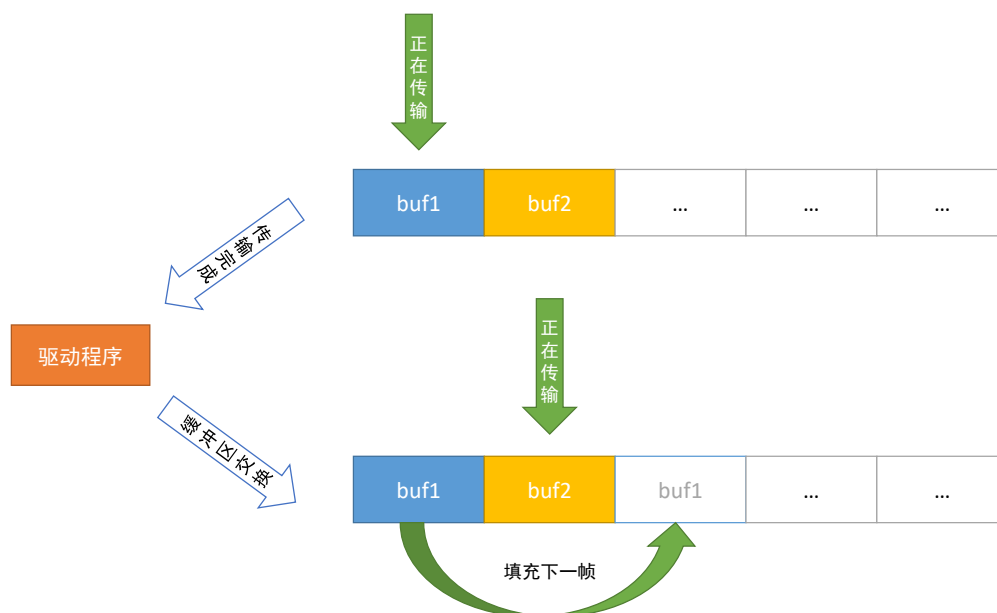


图 5-14 双缓冲区 PCM 帧的传输

设置好某个 PCM 流的参数之后，就可以进行 PCM 帧的传输了，这也是整个驱动程序最核心的部分。

在4.4.4中，我们介绍了 PCM 帧传输的三个重要组成：头部、缓冲区和响应。它们在逻辑上紧邻，在物理上彼此分开，因此用三个 buffer 分别存储。

另外，要想使播放的音乐不间断（即在tx\_queue中始终有设备未使用的缓冲区），我们需要设置双缓冲区，一个缓冲区保存正在传输的 PCM 帧，另一个缓冲区保存等待传输的 PCM 帧，两个缓冲区不断交换角色，就能保证tx\_queue始终不空，如图5-14所示。

在具体实现时，使用两个变量token1和token2来记录两个缓冲区进入tx\_queue后留下的标志，当buf1处于传输状态时，轮询检查其传输状态，一旦传输完成，就填充下一帧，将其入队后更新token1，然后轮询buf2的状态，直到所有 PCM 帧都传输完毕，如图5-15所示。

```
495 pub fn pcm_xfer(&mut self, stream_id: u32, frames: &[u8]) -> Result {
496     // ...stream_id 范围检查
497     let mut buf1 = vec![0; U32_SIZE + buffer_size];
498     let mut buf2 = vec![0; U32_SIZE + buffer_size];
499     // 初始化 buf1
500     buf1[..U32_SIZE].copy_from_slice(&stream_id.to_le_bytes());
501     buf1[U32_SIZE..U32_SIZE + buffer_size]
502         .copy_from_slice(&frames[..buffer_size]);
503     // ... 初始化 buf2, 与 buf1 类似...
504     // 获取初试入队后的 token
505     let mut token1 = unsafe { self.tx_queue.add(&[&buf1], &mut [&mut
        ↪ outputs])).unwrap() };
506     // ...
507     for i in 2..xfer_times {
508         // ...
509         let start_byte = i * buffer_size;
510         let end_byte = if i != xfer_times - 1 {
511             (i + 1) * buffer_size
512         } else {
513             frames.len()
514         };
515         if turn1 {
516             while let Err(_) = unsafe { self.tx_queue.pop_used(token1,
        ↪ &[&buf1], &mut [&mut outputs])) } {
517                 spin_loop();
518             }
519             turn1 = false;
520             // 填充下一帧
521             buf1[U32_SIZE..U32_SIZE + end_byte - start_byte]
522                 .copy_from_slice(&frames[start_byte..end_byte]);
523             // 更新 token
524             token1 = unsafe { self.tx_queue.add(&[&buf1], &mut [&mut
        ↪ outputs])).unwrap() }
525         } else {
526             // ... 等待 buf2 传输完成
527         }
528     }
529     // ... 等待最后一帧传输完成
530 }
```

图 5-15 双缓冲区 PCM 帧传输的实现

## 第 6 章 测试

软件测试是软件开发的重要组成部分，它通过发现问题和评估产品质量来间接改进产品质量<sup>[12]</sup>。驱动程式是一种系统软件，同样需要进行测试以验证其正确性。

对于 virtio sound 驱动程序，使用两种环境进行测试，分别在裸机环境下（无操作系统）和 Alien 下（一个使用 Rust 编写的模块化操作系统）验证其正确性。

### 6.1 裸机

在裸机环境下，驱动后端由 qemu 提供，配置 ALSA 后端在 3.3 已经介绍过，这里主要分析其测试程序如何实现。

---

```
184 // examples/riscv/src/main.rs
185 fn virtio_sound<T: Transport>(transport: T) {
186     let mut sound =
187         VirtIOSound::<HalImpl, T>::new(transport).expect("failed to
            ↳ create sound driver");
188     let output_streams = sound.output_streams();
189     if output_streams.len() > 0 {
190         // ... 获取输出流支持的采样率和采样深度等
191         sound.pcm_set_params(...);
192         sound.pcm_prepare(output_stream_id);
193         sound.pcm_start(output_stream_id);
194         let music =
195             ↳ include_bytes!("../Nocturne_44100Hz_u8_stereo.raw");
196         sound.pcm_xfer(output_stream_id, &music[..]);
197         sound.pcm_stop(output_stream_id).expect("pcm_stop error");
198         sound.pcm_release(output_stream_id).expect("pcm_release
            ↳ error");
199     }
200 }
```

---

图 6-1 裸机下的测试程序

首先要做的是查询 qemu 支持的 virtio sound 的情况，调用 output\_streams 方法获取所有的输出流，由于 qemu 只有一个输出流，因此第 0 个输出流就是我们要操作的输出流。然后按照设置参数、pcm\_prepare、pcm\_start、pcm\_xfer、pcm\_stop 和

pcm\_release 的顺序按部就班地控制输出流即可, 这里使用了预制的、headless 的 PCM 音频, 如图6-1所示。

在设置中查看扬声器情况, 发现正确播放出了预制的音乐, 如图6-2所示。

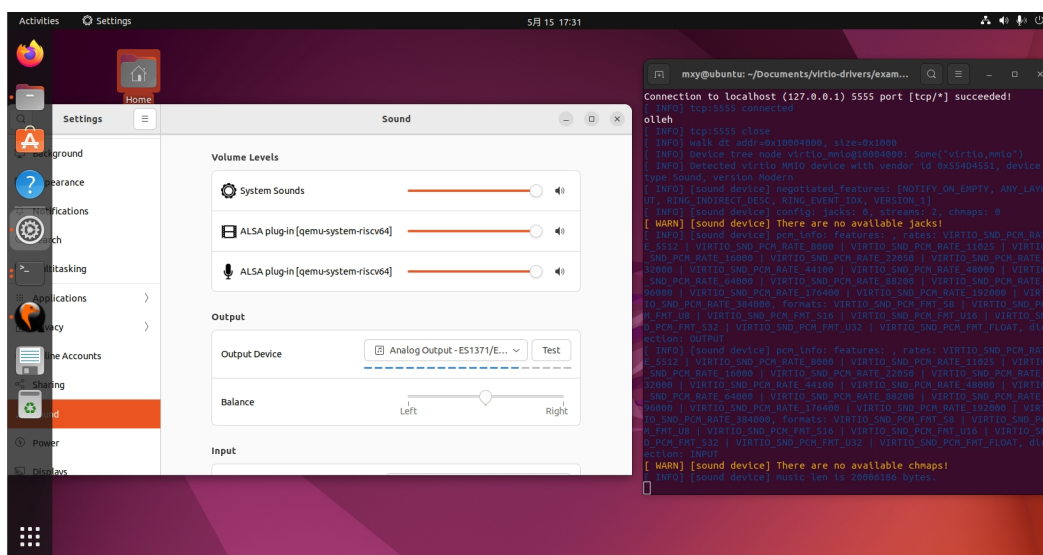


图 6-2 裸机下的测试结果, 打印出了 qemu 的虚拟设备的信息, 且播放了预设的音乐

## 6.2 Alien

### 6.2.1 Alien 简介

Alien<sup>[13]</sup>是一个用 Rust 实现的简单操作系统, 它的目的是探索通过使用模块构建完整的操作系统的可行性。Alien 操作系统由一系列独立的模块组成, 目前已经支持用户态程序和一些简单功能。

### 6.2.2 在 Alien 中测试 virtio sound 驱动程序

为了把我们的 virtio sound 驱动程序嵌入到 Alien 中, 首先需要在 subsystems/device\_interface 下创建音频设备的接口, 这里我们的接口和驱动程序提供的接口保持一致, 如图6-3所示。

与驱动程序提供的接口稍有不同的是, 为了简化错误处理过程, 将 Result 返回值替换为 bool, Ok 对应 true, Err 对应 false。

除了需要定义了音频设备的接口之外, 还需要向 Alien 的文件系统注册 Sound 设

---

```

54 // subsystems/device_interface/src/lib.rs
55 pub trait SoundDevice: DeviceBase {
56     fn jack_remap(&self, jack_id: u32, association: u32, sequence:
57         ↪ u32) -> bool;
58     fn pcm_set_params(&self, stream_id: u32, buffer_bytes: u32,
59         ↪ period_bytes: u32, features: u32, channels: u8, format: u64,
60         ↪ rate: u64,
61     ) -> bool;
62     fn pcm_prepare(&self, stream_id: u32) -> bool;
63     fn pcm_release(&self, stream_id: u32) -> bool;
64     fn pcm_start(&self, stream_id: u32) -> bool;
65     fn pcm_stop(&self, stream_id: u32) -> bool;
66     fn pcm_xfer(&self, stream_id: u32, frames: &[u8]) -> bool;
67     // ...
68 }

```

---

图 6-3 定义音频设备的接口

备（在 Alien 中，IO 设备以文件的形式存在），注册设备的过程如图6-4所示。

---

```

9 // subsystems/devices/src/sound.rs
10 pub static SOUND_DEVICE: Once<Arc<dyn SoundDevice>> = Once::new();
11 pub struct SOUNDDevice {
12     device_id: DeviceId,
13     device: Arc<dyn SoundDevice>
14 }
15 impl VfsFile for SOUNDDevice {
16     // IO 设备被当作文件，这里实现作为文件需要实现的方法
17 }
18 impl VfsInode for SOUNDDevice { // ... }

```

---

图 6-4 向 Alien 的文件系统注册 Sound 设备

最后一步就是接入我们实现的驱动程序了，在drivers文件夹下新增sound.rs，添加 SoundWrapper 作为驱动程序的包装器，为其实现SoundDevice trait 即可，如图6-5所示。

在 Makefile 中以 alsa 为音频后端，添加对 virtio sound 的支持，如图6-6所示。

启动时使用命令make run SOUND=y，观察到终端输出了 virtio sound 设备的有关

```

8 // subsystems/drivers/src/sound.rs
9 pub struct VirtIOSoundWrapper {
10     sound: Mutex<VirtIOSound<HalImpl, MmioTransport>>,
11 }
12 // ...
13 impl SoundDevice for VirtIOSoundWrapper {
14     // ... 实现 SoundDevice 应当具有的方法
15 }

```

图 6-5 为驱动添加 wrapper，并为其实现 SoundDevice trait

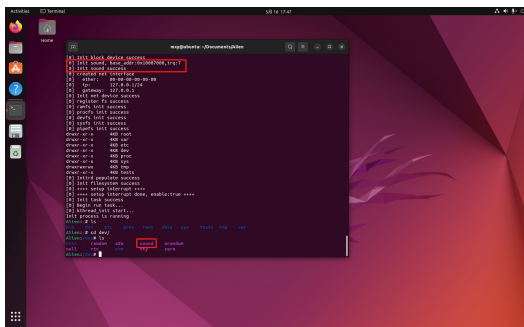
```

46 ifeq ($(SOUND),y)
47 QEMU_ARGS += -device virtio-sound-device,audiodev=audio0 \
48             -audiodev alsa,id=audio0
49 endif

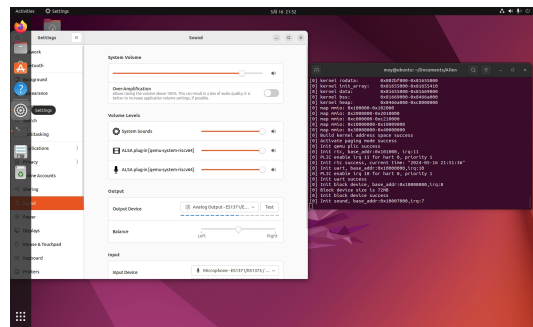
```

图 6-6 在 Makefile 中添加音频设备

信息，检查设置中的扬声器情况，发现检测到了音频的播放，的如图6-7所示。



(a) 检测到 virtio sound 设备



(b) 检查扬声器，确定音乐的播放

图 6-7 在 Alien 内核态下播放预设的音乐



## 第 7 章 未来可能的改进

### 7.1 支持 VirtIO v1.3

VirtIO v1.3<sup>[4]</sup>于 2023 年 10 月 6 日发布，对 virtio sound 设备进行了较大幅度的更新，尤其是在设备的 Feature Bits 新增了VIRTIO\_SND\_F\_CTLs字段来表明设备是否支持控制元素（control elements），针对这一 feature，在设备操作中也更新了对于控制元素的查询等操作。由于 qemu 对 virtio sound 的支持还比较原始（例如 qemu 不支持 jacks 和 chmaps），为了方便测试，在编写驱动时按照 VirtIO v1.2 版本进行实现，未来可能会添加对新标准新特性的支持。

### 7.2 异步改进

目前 virtio sound 的实现是基于轮询的，即 CPU 在无限循环中不断询问缓冲区是否被设备使用完毕，如果使用完毕，填充下一个缓冲区；如果还未使用完毕，继续等待。

基于轮询的 IO 控制有很多弊端，最明显的一点就是高速的 CPU 在等待慢速的 IO 设备传输完成，而不去执行其他程序的指令，二者串行执行，浪费了宝贵的计算资源；并且在执行无限循环时，有大部分时间 CPU 都在执行 JMP 指令，利用率常常到达 100%，看起来利用率高，但实际上在做“无用功”，效率极低。

为了提高 CPU 对实际计算资源的利用率，一般使用的 IO 控制方式分为下面几种：

- 中断驱动：允许 IO 设备主动打断 CPU 的执行并请求其为自身服务，程序的执行流一般是
  - A 进程发出 IO 请求，CPU 将设备需要的数据放入内存，并把缓冲区的内存首地址写入设备控制器的寄存器中；A 进程的状态标记为阻塞，CPU 在就绪队列中挑选进程 B 执行。
  - A 进程的 IO 请求完成，设备发出中断信号，如果该中断没有被屏蔽，CPU 停止 B 进程的执行，去中断向量表中寻找针对该中断需要执行的中断处理指令，同时确定是哪个进程发出的 IO 请求（这里是 A 进程），将发出

IO 请求的进程状态改为就绪，放入就绪队列。

- DMA (Direct Memory Access): 在中断控制中，设备缓冲区中的数据进入内存被 CPU 使用依然需要经过 CPU 寄存器，效率不是很高。DMA 方式则是在设备到内存之间的数据复制操作过程新增了 DMA 控制器，避免了 CPU 的干预，当数据拷贝完成，由 DMA 控制器向 CPU 发出信号，进一步提高了 CPU 的效率。
- I/O 通道是指专门负责输入/输出的处理机。I/O 通道方式是 DMA 方式的发展，它可以进一步减少 CPU 的干预，即把 CPU 对一个数据块读写的干预，减少为对一组数据块读写的干预。

为了提高 virtio sound 驱动程序的效率，实际上应当采用基于中断的 IO 传输方式。在实际实现过程中，利用 Rust 的异步编程机制，结合嵌入式的异步运行时 Embassy<sup>[15]</sup>，大大提高 CPU 的有效利用率，这也是本实现未来需要改进的地方。

## 结 论

本文以 virtio v1.2 为蓝本，使用 virtio-drivers 框架实现了 virtio sound 驱动程序，并在裸机和 Alien 操作系统中进行了测试，为开源操作系统的发展做了一定的贡献。

## 参考文献

- [1] OASIS-OPEN. Virtual I/O Device (VIRTIO) Version 1.2[EB/OL]. (2022-07-01) [2024-04-27]. <https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.pdf>.
- [2] Qwador. virtio-drivers[EB/OL]. 2024 [2024-05-03]. <https://github.com/rcore-os/virtio-drivers>.
- [3] Kadav A, Swift M M. Understanding modern device drivers[J]. ACM SIGPLAN Notices, 2012, 47(4): 87-98.
- [4] RCore-Tutorial-Book. rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档[EB/OL]. 2022 [2024-04-20]. <https://rcore-os.cn/rCore-Tutorial-Book-v3/chapter9/2device-driver-2.html#virtio>.
- [5] Jones M. Virtio: An I/O virtualization framework for Linux[EB/OL]. 2010 [2024-02-07]. <https://developer.ibm.com/articles/l-virtio>.
- [6] Torvalds L. The Linux Kernel Archives[EB/OL]. 2024 [2024-05-06]. <https://github.com/torvalds/linux/tree/master/sound/virtio>.
- [7] Russell R. virtio: towards a de-facto standard for virtual I/O devices[J]. ACM SIGOPS Operating Systems Review, 2008, 42(5): 95-103.
- [8] 安栋, 杨杰. 数字音频基础[M]. 上海市汾阳路 20 号: 上海音乐学院出版社, 2011.
- [9] ALSA-Developers. Advanced Linux Sound Architecture (ALSA) project homepage[EB/OL]. AlsaProject. (2000-10-20) [2024-05-03]. [https://www.alsa-project.org/wiki/Main\\_Page](https://www.alsa-project.org/wiki/Main_Page).
- [10] The-QEMU-Project-Developers. virtio sound[EB/OL]. 2024 [2024-04-30]. <https://www.qemu.org/docs/master/system/devices/virtio-snd.html>.
- [11] Bassey D, Larsen M V. Making VirtIO sing - implementing virtio-sound in rust-vmm project [EB/OL]. 2024 [2024-04-25]. <https://fosdem.org/2024/schedule/event/fosdem-2024-1910-making-virtio-sing-implementing-virtio-sound-in-rust-vmm-project>.
- [12] 肖汉, 郭运宏, 肖波. 软件测试[M]. 电子工业出版社, 2013: 6.
- [13] GitHub - Godones/Alien — github.com[Z]. <https://github.com/Godones/Alien>. [Accessed 16-05-2024].
- [14] OASIS-OPEN. Virtual I/O Device (VIRTIO) Version 1.3[EB/OL]. (2023-10-06) [2024-05-13]. <https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.pdf>.
- [15] Embassy. Embassy — embassy.dev[Z]. <https://embassy.dev/>. [Accessed 15-05-2024].

## 附 录

### 附录 A 英文缩略词表

|        |                                      |
|--------|--------------------------------------|
| QEMU   | Quick EMUlator                       |
| CPU    | Central processing unit              |
| KVM    | Kernel-based Virtual Machine         |
| VirtIO | Virtual Input/Output                 |
| PCM    | Pulse Code Modulation                |
| ALSA   | Advanced Linux Sound Architecture    |
| MIDI   | Musical Instrument Digital Interface |
| DMA    | Direct memory access                 |

## 致 谢

值此论文完成之际，首先向我的导师……

致谢正文样式与文章正文相同：宋体、小四；行距：22 磅；间距段前段后均为 0 行。阅后删除此段。