跨操作系统的音频驱动模块设计与实现

摘　要

在现代计算机中，存在多种操作系统。为了使一个操作系统能够作为 Guest OS 在虚拟化框架上运行，需要开发相应的驱动程序来支持虚拟设备，也就是 virtio 设备。然而，实际上针对特定操作系统编写的 virtio 驱动程序很难移植到其他操作系统上。因此，为了减轻开发人员的负担，实现一个通用的 virtio 驱动程序框架是非常必要的，同时还需要为这个通用框架增加新的 virtio 设备支持。

基于已经存在的通用驱动程序框架，本文设计与实现了基于 VirtIO 协议的跨操作系统的音频驱动模块。

在设计方面，使用 VirtIO v1.2 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，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：驱动程序；声卡驱动；Rust；virtio；操作系统；qemu

Cross-Platform Audio Driver Module Design and Implementation

Abstract

In modern computing, there are various operating systems. To enable an operating system to run as a Guest OS on a virtualization framework, it is necessary to develop corresponding drivers to support virtual devices, known as virtio devices. However, it is often challenging to port a virtio driver written for a specific operating system to other operating systems. Therefore, to alleviate the burden on developers, it is crucial to implement a universal virtio driver framework and add support for new virtio devices to this universal framework.

Based on an existing universal driver framework, this article designs and implements a cross-operating system audio driver module based on the VirtIO protocol.

In terms of design, we use the protocol interface provided by the VirtIO v1.2 documentation and leverage the existing cross-operating system driver framework, virtio-drivers, to add support for the virtio-sound device.

In terms of implementation, we fully utilize the memory safety and high abstraction features of the Rust programming language, making the driver code simple and readable while providing robust extensibility. Even if there are future functional extensions for VirtIO sound devices, this implementation can be easily updated. In this implementation, no modules or structures associated with the standard library are used, making it independent of the operating system. Therefore, one only needs to include this implementation as a dependency in the corresponding operating system's Cargo.toml file and use the interfaces provided by the driver to achieve control of the VirtIO sound device.

Key Words: Driver；Sound Card Driver；Rust；virtio；Operating System；qemu

目　录

第 1 章 绪论

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[1]，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[2]。驱动程序的主要功能如下：

- **硬件控制**：驱动程序直接与硬件设备通信，执行设备的初始化、配置和控制等操作。
- **操作系统接口**：为操作系统提供一个标准化的接口，使操作系统能够使用不同的硬件设备而不需关心具体实现细节。
- **中断处理**：响应硬件设备的中断信号，处理相应的事件并通知操作系统。
- **数据传输**：管理数据在操作系统和硬件设备之间的传输，包括读写操作和缓冲区管理。

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（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。

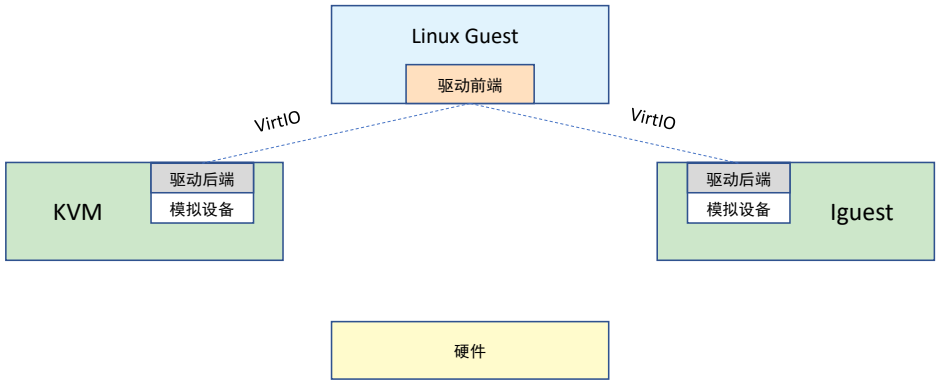


图 1-1 VirtIO 协议下的系统架构[3]

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[4]，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”驱动程序的请求，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[5]。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。本文基于 virtio drivers 框架，开发了跨操作系统的 virtio sound 驱动程序，为广大开源社区贡献了自己的绵薄之力。

1.2 毕业设计目标和实际意义

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

本项目的成功实现具有以下实际意义：

- **性能和效率**：virtio 音频驱动程序通过标准化的接口减少了虚拟化开销，提高了音频设备在虚拟机中的性能。传统的虚拟化音频设备可能需要复杂的模拟和翻译过程，而 virtio 通过直接、优化的数据传输路径显著提升了效率。
- **可移植性**：virtio 标准旨在跨不同的虚拟化平台和操作系统提供一致的接口。跨平台 virtio 音频驱动程序的实现意味着同一套驱动程序可以在多种虚拟化环境中工作，如 KVM、QEMU、Xen 等，从而简化了驱动程序的开发和维护。
- **简化开发和维护**：采用 virtio 标准的音频驱动程序具有统一的接口规范，开发者无需针对不同的虚拟化平台编写不同的驱动程序。这不仅减少了开发工作量，还简化了后续的维护和升级工作，使驱动程序更快地适应新版本的虚拟化平台和操作系统。

- **灵活性和扩展性**：virtio 音频驱动程序可以方便地扩展和定制，以满足不同应用场景的需求。通过标准化的接口，开发者可以轻松添加新特性或优化现有功能，而无需担心平台特定的限制。

1.3 文章结构

本文的总体结构如下：

第一章作为绪论，主要简述了驱动程序的发展历史——从最初的程序库到现在的针对虚拟设备的驱动；还介绍了研究背景以及实现跨操作系统虚拟设备驱动的现实意义。

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。

第三章以 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。

第四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，主要包括如何使用 Rust 语言基于 virtio-drivers 框架进行代码编写，与第三章之间存在一定的对应关系。

为了验证实现的正确性，第五章给出了基于 qemu 模拟器分别在裸机和 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，因此在本章也对本实现的缺点和改进方案做了一定的剖析。

第 2 章 相关技术简介

2.1 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，每种虚拟化解决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[6]。virtio 旨在提供一种高效的、标准化的方式来管理虚拟设备，使得虚拟机中的设备能够更高效地与宿主系统交互，现已成为许多虚拟化平台的标准。virtio 具有以下特点：

- **高效性**：Virtio 通过减少虚拟化开销来提高性能。例如，它可以减少因设备模拟带来的开销。
- **标准化**：提供一套标准的接口，使得不同的虚拟化平台可以更容易地实现设备兼容性。
- **灵活性**：支持多种类型的虚拟设备，包括网络、块设备、控制台等。

virtio 支持多种类型的虚拟设备，一些常见的类型如下：

- **Virtio-Net**：用于虚拟化网络接口。Virtio-Net 提供高效的网络传输，减少网络延迟和开销。
- **Virtio-Block**：用于虚拟化块设备（例如磁盘）。它提供高效的磁盘 I/O 操作。
- **Virtio-Console**：用于虚拟化控制台设备，允许虚拟机与宿主系统之间进行控制台输入输出。
- **Virtio-Input**：用于虚拟化输入设备，如键盘和鼠标。

virtio 是虚拟化技术中的一个重要组件，通过标准化和高效的设备接口，显著提升了虚拟机的性能和兼容性。无论是在服务器虚拟化、桌面虚拟化还是云计算中，virtio 都发挥着重要作用。

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）。

2.1.1 virtio 概述

virtio 可以简单地划分为三层，如图2-1所示。

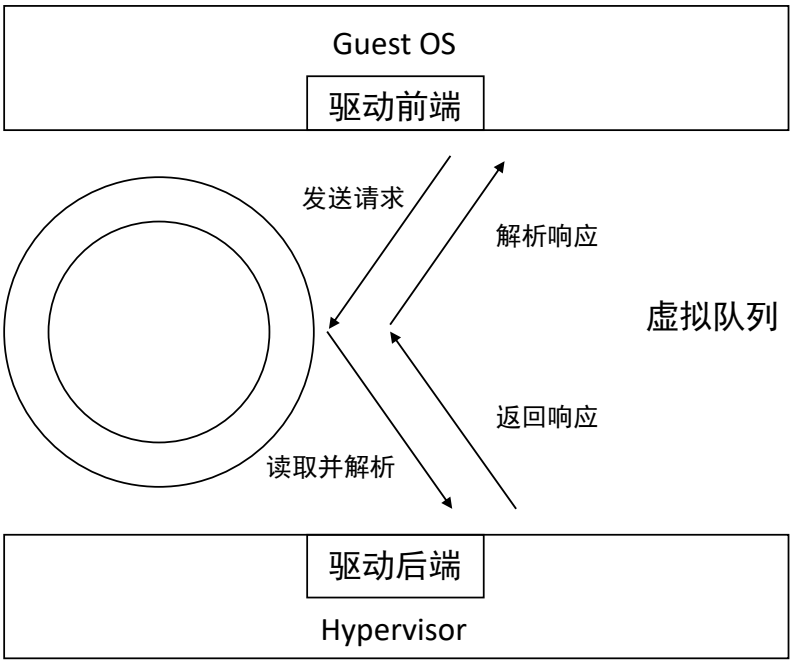


图 2-1 virtio 的三个层次

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。

2.1.2 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。

2.1.2.1 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。

2.1.2.2 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。[6]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。

2.1.2.3 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。

2.1.3 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。

```
struct vring_used_elem {
    __u32 id;
    __u32 len;
};
struct vring_used {
    __u16 flags;
    __u16 idx;
    struct vring_used_elem ring[];
};
```

图 2-4 已用环的 C 语言描述

2.1.3.1 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[7]：

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）。

2.1.3.2 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。

2.1.3.3 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。

2.2 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）**。

2.2.1 PCM

PCM 的完整过程分为三步，分别是：**采样**、**量化**、**编码**。图2-5描述了 PCM 的这三个过程。

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。

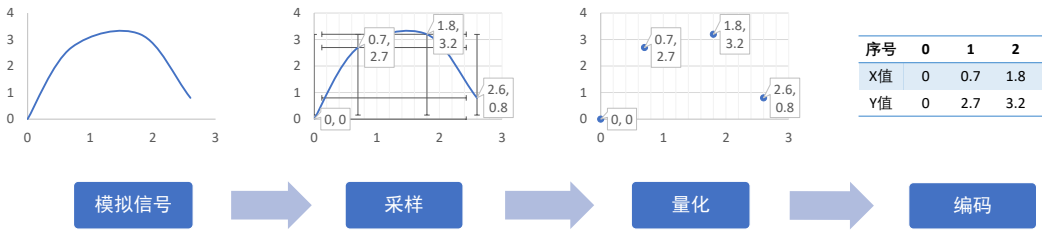


图 2-5 脉冲编码调制（PCM）的过程

2.2.2 数字音频基本概念

2.2.2.1 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）。[8]

根据奈奎斯特采样定理（式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。

$$f_s \geq 2f_m \tag{2-1}$$

2.2.2.2 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。

2.2.2.3 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。

2.2.3 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。[9]

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[10]，这里我们使用 ALSA，如图2-6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 2-6 在 qemu 中指定宿主机的音频后端

2.2.3.1 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 $2 \times 16bit = 4\ Bytes$
- 一帧 5.1 声道 48KHz 16 位 PCM 流占据 $6 \times 16bit = 12\ Bytes$

2.2.3.2 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。[9]

这两个概念在设置 PCM 流的参数时需要用到，如图2-7所示。

在图2-7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。[7]

```rust
// src/device/sound.rs
/// Set selected stream parameters for the specified stream ID.
pub fn pcm_set_params(
    &mut self,
    stream_id: u32,
    buffer_bytes: u32,
    period_bytes: u32,
    // ...
) -> Result {
    // ...
}
```

```rust
// examples/riscv/src/main.rs
sound
    .pcm_set_params(
        output_stream_id,
        441 * 2,
        441,
        // ...
    )
```

图 2-7 设置 PCM 流的 period bytes 和 buffer bytes

第 3 章 VirtIO Sound Device 概要

VIRTIO v1.2[7]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。

3.1 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。

3.2 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3-1所示。

其中，

- **jacks** 表示所有可用插孔的总数。

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

图 3-1 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。[7]

3.3 PCM 生命周期

一个 PCM 流有以下的生命周期阶段，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3-2所示。

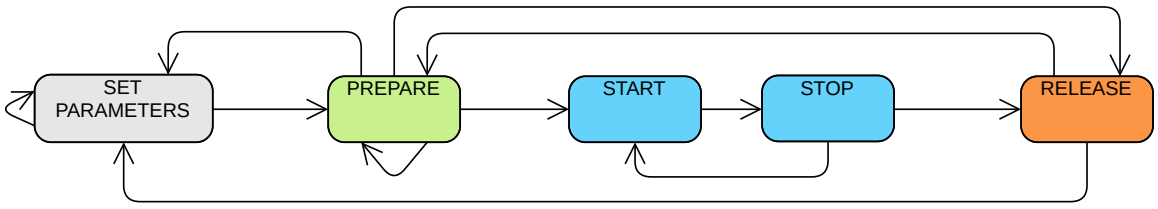


图 3-2 PCM 生命周期转换图

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[11]，如图3-3所示。

```rust
// src/device/sound.rs
macro_rules! set_new_state {
    ($new_state_fn:ident, $new_state:expr,
    ↪ $($valid_source_states:tt)*) => {
        fn $new_state_fn(&mut self) -> PCMStateResult<()> {
            if !matches!(self, $($valid_source_states)*) {
                return Err(StreamError::InvalidStateTransition(*self,
                ↪ $new_state));
            }
            *self = $new_state;
            Ok(())
        }
    };
}
```

图 3-3 生命周期转换的实现，使用宏来减少重复代码

3.4 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3-4所示。

```rust
// src/device/sound.rs
/// A common header
#[repr(C)]
#[derive(AsBytes, FromBytes, FromZeroes, PartialEq, Eq)]
struct VirtIOSndHdr {
    command_code: u32,
}
```

图 3-4 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。

3.4.1 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3-5所示。

```
// src/device/sound.rs
#[repr(C)]
#[derive(AsBytes, FromBytes, FromZeroes)]
struct VirtIOSndQueryInfo {
    hdr: VirtIOSndHdr,
    start_id: u32,
    count: u32,
    size: u32,
}
```

图 3-5 配置信息查询的请求结构体实现

在图3-5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3-6设置各个字段的值。

```
VirtIOSndQueryInfo {
    hdr: CommandCode::VirtioSndRPcmInfo.into(),
    start_id: 0, // 从第 0 个 PCM 流开始
    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
    size: mem::size_of::<VirtIOSndPcmInfo>() as u32 // PCM 流信息的结
    ↪ 构体大小
}
```

图 3-6 查询所有 PCM 流的信息

3.4.2 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_RE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3-7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 3-7 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。

3.4.3 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3-8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 3-8 设置流的参数的请求结构体

其中，buffer_bytes和period_bytes的含义在2.2.3.2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。

3.4.4 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3-9所示。

```
/* IO 消息头部 */
struct virtio_snd_pcm_xfer {
    le32 stream_id
};
/* IO 消息状态部 */
struct virtio_snd_pcm_status {
    le32 status,
    le32 latency_bytes
}
```

图 3-9 PCM 流的 IO 消息组成[7]

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 4 章 VirtIO Sound 驱动程序的实现

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4-1所示。

```
# Cargo.toml
[dependencies]
log = "0.4"
bitflags = "2.3.0"
zerocopy = { version = "0.7.5", features = ["derive"] }

num_enum = { version = "0.7.2", default-features = false}
```

图 4-1 项目使用的库

Rust 是一种系统编程语言，旨在提供内存安全性、并发性和高性能。它最早由 Mozilla 研究员 Graydon Hoare 于 2006 年构思，并在随后的几年中逐渐发展和成熟。

2006 年，Graydon Hoare 开始构思和开发 Rust 语言，最初是作为一个个人项目。2009 年，Mozilla 开始资助 Rust 项目，正式将其纳入 Mozilla 研究计划。2010 年，Rust 项目首次公开发布，吸引了早期开发者和社区的关注。

2011 年，Rust 发布了 0.1 版本，标志着语言和工具链的初步成形。2012 年，Rust 开始使用 LLVM（Low-Level Virtual Machine）作为其后台编译器基础。2013 年，Rust 语言设计团队引入了所有权（Ownership）系统，提供编译时的内存安全保障，避免了数据竞争和悬挂指针等问题。2014 年，Rust 发布了 1.0 alpha 版本，标志着语言的成熟和稳定版本的逐步接近。

2015 年，Rust 1.0 正式发布，这是第一个稳定版本，强调了语言的核心特性，如内存安全、并发性和无垃圾回收器（Garbage Collector）。2016 年，Rust 的包管理和构建工具 Cargo 变得更加成熟，极大地促进了开源生态系统的发展。2017 年，Rust 的异步编程模型开始形成，社区引入了 Future 和 async/await 语法，进一步增强了并发编程能力。2018 年，Rust 迎来了其第一个年度 Rust 2018 版（Rust Edition），这标志着语言的显著改进和演变，包括更好的 IDE 支持、模块系统改进和异步编程强化。2019 年，Rust 成为 Stack Overflow 开发者调查中最受欢迎的编程语言，显示了其在

开发者社区中的广泛接受度。

2020 年，Rust 继续强化其异步编程能力，推出了更多的库和工具支持，社区的活跃度不断提升。2021 年，Rust 基金会成立，这是一个独立的非营利组织，旨在支持和推动 Rust 语言的发展和社区建设。2022 年，Rust 2021 版（Rust Edition 2021）发布，进一步改进了语言的可用性和性能，增加了新的语法和编译器特性。2023 年，Rust 的应用领域不断扩展，除了系统编程外，还在 WebAssembly、嵌入式系统和区块链等领域取得了显著进展。

Rust 的关键特性与优势包括内存安全、高性能、并发性、包管理和构建工具，以及其活跃的社区。通过所有权系统，Rust 在编译时保证内存安全，避免了常见的内存错误，如空指针和数据竞争。Rust 没有运行时垃圾回收，性能接近 C 和 C++，适合系统编程和高性能应用。Rust 提供了强大的并发编程模型，使开发者能够编写高效、安全的并发代码。Cargo 是 Rust 的包管理和构建工具，极大地方便了依赖管理和构建过程。Rust 拥有一个活跃和友好的社区，提供了丰富的库和工具支持。

本项目使用 Rust 语言实现 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。

4.1 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4-2所示。

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

图 4-2 驱动程序的整体架构

4.2 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**。

4.2.1 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。

4.2.1.1 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$\tilde{3}$，使用编译期常量来记录队列编号和队列大小，如图4-3所示。

```
// src/device/sound.rs
const QUEUE_SIZE: u16 = 32;
const CONTROL_QUEUE_IDX: u16 = 0;
const EVENT_QUEUE_IDX: u16 = 1;
const TX_QUEUE_IDX: u16 = 2;
const RX_QUEUE_IDX: u16 = 3;
```

图 4-3 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4-4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。

```
// src/device/sound.rs
let control_queue = VirtQueue::new(
    &mut transport,
    CONTROL_QUEUE_IDX,
    ↪ negotiated_features.contains(SoundFeatures::VIRTIO_F_INDIRECT_DESC),
    negotiated_features.contains(SoundFeatures::VIRTIO_F_EVENT_IDX),
)?;
```

图 4-4 初始化control_queue

4.2.1.2 读取配置空间字段

virtio sound 的配置空间组成在3.2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4-5所示。

```
// src/device/sound.rs
let config_ptr = transport.config_space::<VirtIOSoundConfig>()?;
let (jacks, streams, chmaps) = unsafe {
    (
        volread!(config_ptr, jacks),
        volread!(config_ptr, streams),
        volread!(config_ptr, chmaps),
    )
};
```

图 4-5 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4-5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。

4.2.1.3 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4-6所示。

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

图 4-6 初始化缓冲区

4.2.1.4 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4-5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4-7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 4-7 初始化 PCM 流的状态为SetParams

4.2.2 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3.4.1一节中介绍过，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4-8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4-9所示。

```rust
fn pcm_info(
    &mut self,
    stream_start_id: u32,
    stream_count: u32,
) -> Result<Vec<VirtIOSndPcmInfo>> {
    // ...
    let request_hdr =
    ↪ VirtIOSndHdr::from(ItemInfomationRequestType::VirtioSndRPcmInfo);
    let hdr: VirtIOSndHdr = self.request(VirtIOSndQueryInfo {
        // ...
    })?;
    // ...
    let mut pcm_infos = vec![];
    for i in 0..stream_count as usize {
        const HDR_SIZE: usize = mem::size_of::<VirtIOSndHdr>();
        const PCM_INFO_SIZE: usize =
        ↪ mem::size_of::<VirtIOSndPcmInfo>();
        let start_byte_idx = HDR_SIZE + i * PCM_INFO_SIZE;
        let end_byte_idx = HDR_SIZE + (i + 1) * PCM_INFO_SIZE;
        let pcm_info = VirtIOSndPcmInfo::read_from(
            &self.queue_buf_recv[start_byte_idx..end_byte_idx]
        ).unwrap();
        pcm_infos.push(pcm_info);
    }
    Ok(pcm_infos)
}
```

图 4-8 读取 PCM 流的配置信息

```rust
    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 4-9 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4-8中第 7 行的`request_hdr`的类型，这里不再赘述。

4.3 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 4-10 获取所有输出流

在图4-10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。

4.4 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4-11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。

```rust
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 4-11 流的基本信息查询，以rates_supports为例

4.5 插孔重新映射（Jack Remap）

插孔重新映射的方法签名如下：

```rust
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的插孔才能进行从新映射；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4-12所示。

```rust
// src/device/sound.rs
pub fn jack_remap(&mut self, jack_id: u32, association: u32, sequence:
↪  u32) -> Result {
    // ... 判断指定插孔是否支持重新映射
    let hdr: VirtIOSndHdr = self.request(VirtIOSndJackRemap {
        hdr: VirtIOSndJackHdr {
            hdr: CommandCode::VirtioSndRJackRemap.into(),
            jack_id,
        },
        association,
        sequence,
    })?;
    // ... 后续处理
}
```

图 4-12 插孔重新映射请求

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

图 4-13 设置流的参数并保存

4.6 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3.4.3已经介绍过。

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4-13所示。

4.7 PCM 帧的传输

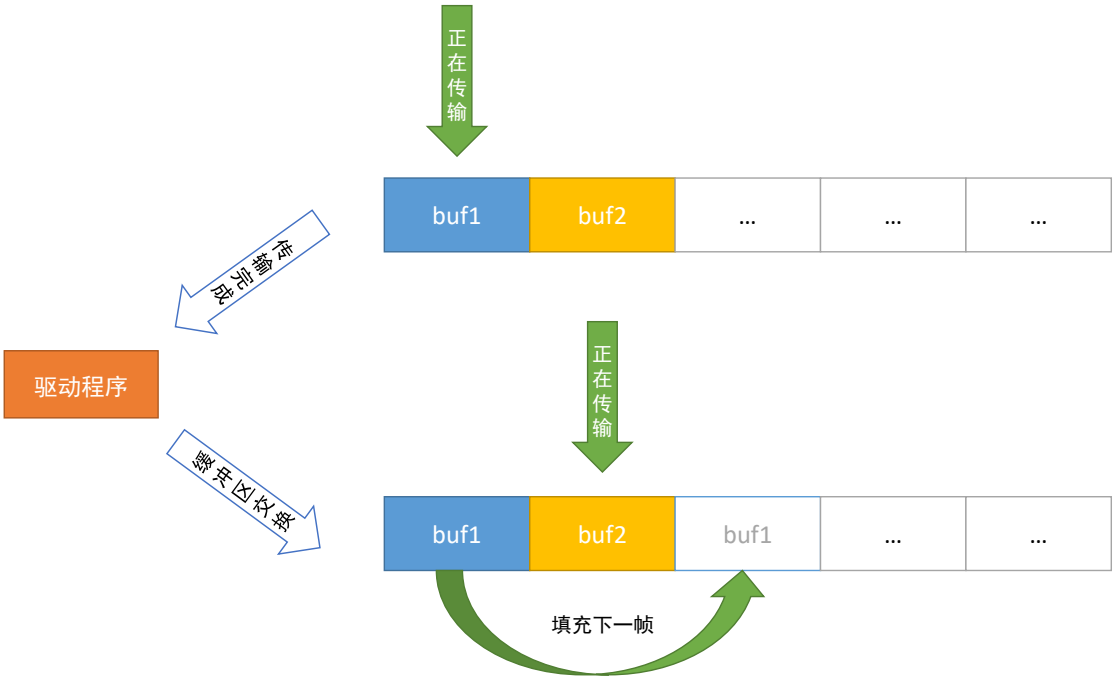


图 4-14 双缓冲区 PCM 帧的传输

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3.4.4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4-14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4-15所示。

```rust
pub fn pcm_xfer(&mut self, stream_id: u32, frames: &[u8]) -> Result {
    // ...stream_id 范围检查
    let mut buf1 = vec![0; U32_SIZE + buffer_size];
    let mut buf2 = vec![0; U32_SIZE + buffer_size];
    // 初始化 buf1
    buf1[..U32_SIZE].copy_from_slice(&stream_id.to_le_bytes());
    buf1[U32_SIZE..U32_SIZE + buffer_size]
        .copy_from_slice(&frames[..buffer_size]);
    // ... 初始化 buf2, 与 buf1 类似...
    // 获取初试入队后的 token
    let mut token1 = unsafe { self.tx_queue.add(&[&buf1], &mut [&mut
    ↪  outputs]).unwrap() };
    // ...
    for i in 2..xfer_times {
        // ...
        let start_byte = i * buffer_size;
        let end_byte = if i != xfer_times - 1 {
            (i + 1) * buffer_size
        } else {
            frames.len()
        };
        if turn1 {
            while let Err(_) = unsafe { self.tx_queue.pop_used(token1,
            ↪  &[&buf1], &mut [&mut outputs]) } {
                spin_loop();
            }
            turn1 = false;
            // 填充下一帧
            buf1[U32_SIZE..U32_SIZE + end_byte - start_byte]
                .copy_from_slice(&frames[start_byte..end_byte]);
            // 更新 token
            token1 = unsafe { self.tx_queue.add(&[&buf1], &mut [&mut
            ↪  outputs]).unwrap() }
        } else {
            // ... 等待 buf2 传输完成
        }
    }
    // ... 等待最后一帧传输完成
}
```

图 4-15 双缓冲区 PCM 帧传输的实现

第 5 章 测试与分析

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[12]。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。

目前的实现存在一些效率上的问题，这些将在“未来可能的改进”小结中进行探讨。

5.1 裸机

在裸机环境下，驱动后端由 qemu 提供，配置 ALSA 后端在2.2.3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 =
            ↪ include_bytes!("../Nocturne_44100Hz_u8_stereo.raw");
195         sound.pcm_xfer(output_stream_id, &music[..]);
196         sound.pcm_stop(output_stream_id).expect("pcm_stop error");
197         sound.pcm_release(output_stream_id).expect("pcm_release
            ↪ error");
198     }
199 }
```

图 5-1 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5-1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5-2所示。

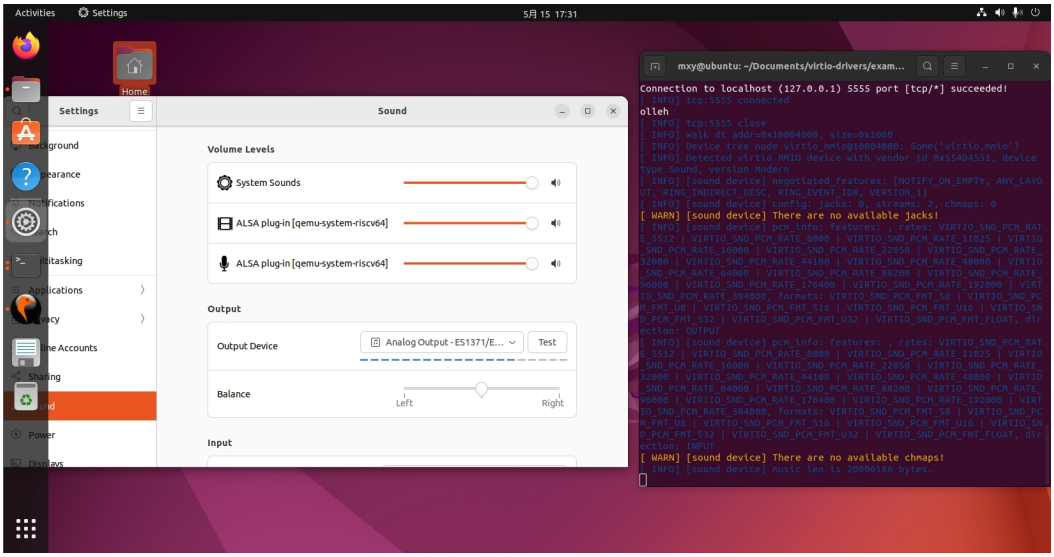


图 5-2 裸机下的测试结果，打印出了 qemu 的虚拟设备的信息，且播放了预设的音乐

5.2 Alien

5.2.1 Alien 简介

Alien[13]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。

5.2.2 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基本保持一致，如图5-3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

```rust
// subsystems/device_interface/src/lib.rs
pub trait SoundDevice: DeviceBase {
    fn jack_remap(&self, jack_id: u32, association: u32, sequence:
    ↪  u32) -> bool;
    fn pcm_set_params(&self, stream_id: u32, buffer_bytes: u32,
    ↪  period_bytes: u32, features: u32, channels: u8, format: u64,
    ↪  rate: u64,
    ) -> bool;
    fn pcm_prepare(&self, stream_id: u32) -> bool;
    fn pcm_release(&self, stream_id: u32) -> bool;
    fn pcm_start(&self, stream_id: u32) -> bool;
    fn pcm_stop(&self, stream_id: u32) -> bool;
    fn pcm_xfer(&self, stream_id: u32, frames: &[u8]) -> bool;
    // ...
}
```

图 5-3 定义音频设备的接口

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 设备（在 Alien 中，IO 设备以文件的形式存在），注册设备的过程如图5-4所示。

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

图 5-4 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5-5所示。

```rust
// subsystems/drivers/src/sound.rs
pub struct VirtIOSoundWrapper {
    sound: Mutex<VirtIOSound<HalImpl, MmioTransport>>,
}
// ...
impl SoundDevice for VirtIOSoundWrapper {
    // ... 实现 SoundDevice 应当具有的方法
}
```

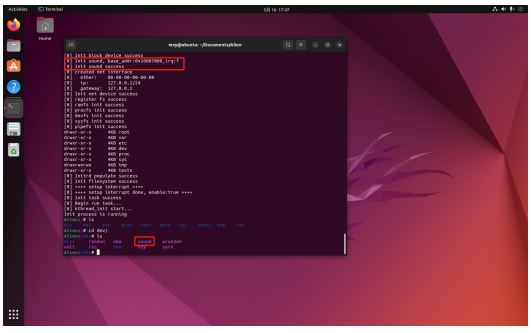
图 5-5 为驱动添加 wrapper，并为其实现 SoundDevice trait

在 Makefile 中以 alsa 为音频后端，添加对 virtio sound 的支持，如图5-6所示。

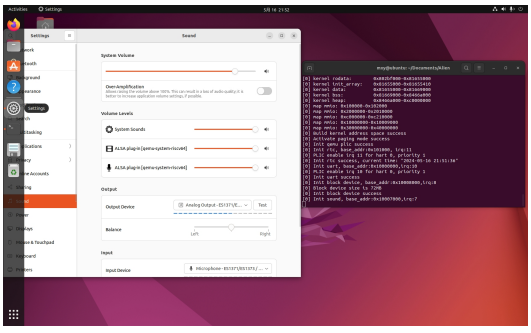
```
ifeq ($(SOUND),y)
QEMU_ARGS += -device virtio-sound-device,audiodev=audio0 \
             -audiodev alsa,id=audio0
endif
```

图 5-6 在 Makefile 中添加音频设备

启动时使用命令`make run SOUND=y`，观察到终端输出了 virtio sound 设备的有关信息，检查设置中的扬声器情况，发现检测到了音频的播放，如图5-7所示。



(a) 检测到 virtio sound 设备



(b) 检查扬声器，确定音乐的播放

图 5-7 在 Alien 内核态下播放预设的音乐

5.3 未来可能的改进

5.3.1 支持 VirtIO v1.3

VirtIO v1.3[14]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。

5.3.2 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[15]，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。

在实现过程中，借助 Rust 语言的所有权机制，完全消除了内存安全分风险；除此之外，程序针对用户的错误输入做了单独处理，鲁棒性较强；针对每个 package 的公开结构体或方法，都添加了文档说明，方便操作系统设计者使用此驱动程序。

本驱动程序的实现是一项完全创新的探索，尽管 Linux 内核早在 2021 年就实现了 virtio sound 驱动程序，但是目前开源社区还没有使用 Rust 语言开发跨操作系统的音频驱动程序的先例，因此本项目是一次大胆的尝试，从出发点和结果来看，应该算是很成功的。

虽然本实现版本达到了预期结果，但是仍有一些方面不尽如人意，尤其是性能表现方面。轮询设备使得 CPU 的占用率达到了 100%，但是 CPU 一直在做无用功，浪费大量的计算资源在等待外设完成 IO 上，导致整个系统的效率很低。解决的方案就是采用异步驱动或者是利用操作系统提供的中断服务——每传输完一个缓冲区，向操作系统发送中断请求，使其填满下一个缓冲区，然后操作系统回到原来中断的位置继续执行，这样 IO 设备和 CPU 之间就是异步关系，二者各司其职，大大提高系统的效率。因此异步改进也是本实现未来需要考虑的方向。

参考文献

[1] Kadav A, Swift M M. Understanding modern device drivers[J]. ACM SIGPLAN Notices, 2012, 47(4): 87-98.

[2] 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.

[3] Jones M. Virtio: An I/O virtualization framework for Linux[EB/OL]. 2010 [2024-02-07]. https://developer.ibm.com/articles/l-virtio.

[4] Torvalds L. The Linux Kernel Archives[EB/OL]. 2024 [2024-05-06]. https://github.com/torvalds/linux/tree/master/sound/virtio.

[5] Qwandor. virtio-drivers[EB/OL]. 2024 [2024-05-03]. https://github.com/rcore-os/virtio-drivers.

[6] Russell R. virtio: towards a de-facto standard for virtual I/O devices[J]. ACM SIGOPS Operating Systems Review, 2008, 42(5): 95-103.

[7] 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.

[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.

[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. 2024-05-16.

[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/. 2024-05-15.

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