文本

描述已自动生成

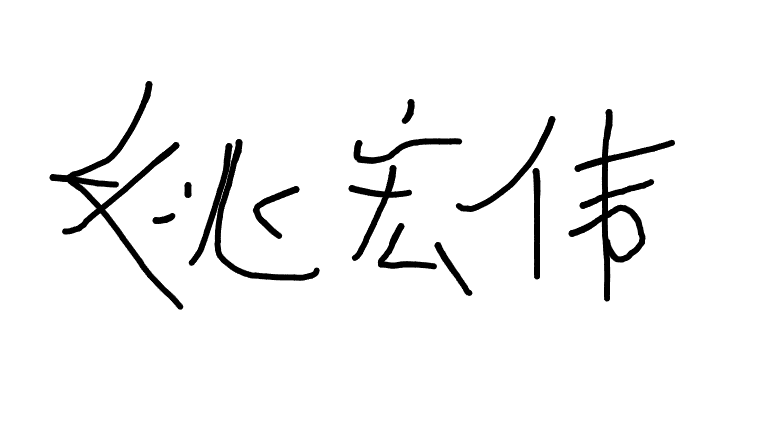
**本 科 毕 业 论 文（设 计）**

|  |  |
| --- | --- |
| 题 目 | 基于Rust的跨平台USB驱动 |
|  | 子系统的设计与实现 |
| 学生姓名 | 姚宏伟 |
| 学 号 | 2311729 |
| 学 院 | 信息技术学院 |
| 专业班级 | 计科B21-10 |
| 指导教师 | 丁菊 |
| 交稿日期 | 2025年5月12日 |

教务处制

**上海建桥学院毕业论文（设计）学术诚信声明**

本人郑重声明：所呈交的毕业论文（设计），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本毕业论文（设计）不含任何其他个人或集体已经发表或撰写过的作品或成果。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。本人完全意识到本声明的法律结果由本人承担。



作者签名： 日期： 2025年 5 月 12日

**上海建桥学院毕业论文（设计）版权使用授权书**

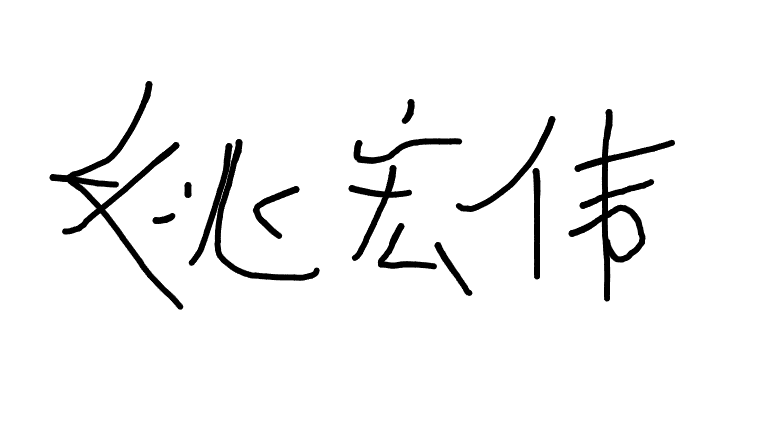
本毕业论文（设计）作者同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海建桥学院可以将本毕业论文（设计）的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本毕业论文（设计）。

**保 密** □，在 年解密后适用本授权书。

本论文属于

**不保密** ☑。

（请在以上方框内打“**√**”，如作者未做出选择的情况下，按不保密处理。）

b60368acd548ae665211a13bcc360f7

作者签名： 指导教师签名：

日期：2025年 5 月12 日 日期： 2025 年 5 月12 日

基于Rust的跨平台USB驱动子系统的设计与实现

摘 要

随着计算机技术与硬件的不断进步，USB（通用串行总线）已经成为连接计算机与各类电子设备的通用通信接口。USB驱动程序在这些设备间的通信过程中扮演着至关重要的角色。然而，随着国内操作系统开发的兴起，众多国产操作系统应运而生。由于USB接口的复杂性，甚至可以明确的说，它是目前最复杂的总线之一，其驱动程序的开发颇具挑战性。此外，传统的USB驱动程序往往针对特定操作系统进行定制开发，这使得将现有的USB驱动程序迁移到其他操作系统框架下变得异常困难。

鉴于此，本文提出了一种跨架构、跨操作系统的USB主机端驱动框架。该框架兼容当前主流的USB主机控制器版本，并为上层驱动提供了一个统一的抽象接口。它支持在单线程或多线程环境下异步运行，并采用了一种基于中断、主动让出、轮询进行任务切换的异步编程模型。

此框架采用Rust编程语言实现，旨在充分利用Rust语言的安全性和可移植性优势。

**关键词：通用串行总线（USB）；驱动程序；操作系统；Rust编程语言**

A Cross-Platform USB Host Driver Subsystem Based On Rust

Abstract

With the continuous advancement of computer technology and hardware, USB (Universal Serial Bus) has become a universal communication interface for connecting computers with various electronic devices. USB drivers play a crucial role in the communication process between these devices. However, with the rise of domestic operating system development, numerous locally produced operating systems have emerged. Due to the complexity of USB interfaces—which can even be unequivocally stated as one of the most complex buses currently—the development of USB drivers is quite challenging. In addition, traditional USB drivers are often custom-developed for specific operating systems, making it extremely difficult to migrate existing USB drivers to other operating system frameworks.

In view of this, this paper proposes a cross-architecture and cross-operating system USB host driver framework. This framework is compatible with current mainstream USB host controller versions and provides a unified abstract interface for upper-layer drivers. It supports asynchronous operation in single-threaded or multi-threaded environments and adopts an asynchronous programming model based on interrupts, active yielding, and polling for task switching.

The framework is implemented using the Rust programming language, aiming to fully leverage Rust's advantages of safety and portability.

**Key Words：Universal Serial Bus (USB); Driver System; Operating System; Rust Programming Language**

目 录

[1 绪论 1](#_Toc9817)

[1.1 课题背景 1](#_Toc16161)

[1.2 相关领域国内外发展现状与趋势 2](#_Toc22050)

[1.2.1 国外发展现状 2](#_Toc11072)

[1.2.2 国内相关技术的发展趋势 2](#_Toc25327)

[1.3 论文研究内容及主要工作 3](#_Toc18043)

[1.3.1 论文研究内容 3](#_Toc11520)

[1.3.2 论文主要工作 5](#_Toc29477)

[2 跨平台USB主机端驱动子系统的整体设计方案及原理概述 7](#_Toc27659)

[2.1 系统整体方案设计 7](#_Toc13830)

[2.2 下文所涉及概念与名词解释 8](#_Toc13819)

[2.3 系统各模块具体的技术研究 11](#_Toc26780)

[2.3.1 主机驱动层的技术细节 11](#_Toc5289)

[2.3.2 USB驱动层的技术细节 13](#_Toc29284)

[2.4 本章小结 16](#_Toc27264)

[3 硬件与仿真环境 17](#_Toc26996)

[3.1 开发所用平台环境 17](#_Toc9517)

[3.1.1 对于ARM架构的测试与验证 17](#_Toc15604)

[3.1.2 对于RISC-V架构的测试与验证 19](#_Toc30399)

[3.2 测试所用硬件设备 19](#_Toc24620)

[3.3 本章小结 21](#_Toc20715)

[4 软件程序设计与实现 22](#_Toc25416)

[4.1 程序设计语言及开发环境 22](#_Toc31569)

[4.1.1 编程语言：Rust程序设计语言 22](#_Toc3846)

[4.1.2 工具：Makefile 22](#_Toc9452)

[4.1.3 集成开发环境：VSCode 22](#_Toc730)

[4.2 模块化设计与项目结构 23](#_Toc4638)

[4.3 USB驱动子系统框架的实现 24](#_Toc4444)

[4.3.1 USB驱动子系统模块 24](#_Toc21487)

[4.3.2 主机控制器接口 25](#_Toc13562)

[4.3.3 USB系统驱动层模块 26](#_Toc6782)

[4.3.4 核心思想：结构化并发调度 27](#_Toc1847)

[4.4 USB描述符解析系统的实现 28](#_Toc943)

[4.4.1 USB描述符的数据流 28](#_Toc14091)

[4.4.2 USB描述符解析系统的设计 30](#_Toc4254)

[4.5 本章小结 31](#_Toc7966)

[5 系统测试 32](#_Toc32612)

[5.1 USB描述符解析器的软件测试 32](#_Toc29734)

[5.2 USB驱动子系统软硬件协同测试 33](#_Toc29228)

[5.2.1 USB驱动子系统的初始化流程 34](#_Toc29580)

[5.2.2 基于驱动无关设备中各描述符所对应USB功能的驱动指派 35](#_Toc3008)

[5.2.3 HID驱动模块的测试 37](#_Toc536)

[5.2.4 USB-CDC-CH34X驱动模块的测试 38](#_Toc26214)

[5.3 实际应用：使用USB驱动控制小车运动 39](#_Toc5867)

[5.4 在进一步测试过程中所暴露出的问题与解决方案 40](#_Toc4801)

[5.5 本章小结 42](#_Toc17915)

[结 论 43](#_Toc2891)

[参 考 文 献 44](#_Toc25809)

[致 谢 45](#_Toc18580)

# 绪论

## 课题背景

USB接口协议复杂，其驱动开发难度较高，而且传统的USB驱动程序通常针对特定的操作系统进行开发，这导致了众多非Linux发行版的国产操作系统总是缺少USB驱动支持的问题。事实上，驱动生态匮乏是整个领域（包括嵌入式领域）的共性问题。不同平台的软硬件底层架构和驱动模型不同。此外，特定驱动通常仅针对单一平台开发，缺乏通用性，导致大量重复工作。开发人员精力分散也阻碍了统一规范的形成，进一步影响了驱动代码的安全性。驱动程序所隐含的漏洞已成为现代操作系统生态中最主要的威胁。由H. Peng和M. Payer在29th USENIX Security Symposium上提出的USBFuzz[1]测试框架揭示了Linux中现有USB驱动中可能存在的安全漏洞及其测试方法，凸显了对新开发驱动进行严格安全性测试的重要性。

另一方面，随着USB规范的更新迭代，USB总线的吞吐容量越来越高，其规范也越来越复杂，这也对相应的设备驱动编写人员的技术，乃至于操作系统管理IO的能力提出了更高的要求，J. Huang和Y. Wang的研究[2]专注于基于USB 3.0外设控制器的高速同步数据传输系统，这表明未来USB驱动将更加注重性能优化。这令使用传统C语言进行驱动开发的开发者们愈发难以应付高层次协议的高抽象程度的要求。Rust语言是一种内存安全，提供零成本抽象的高级编程语言，其独特的设 计理念与极高的性能优势使得其在面向计算机底层的开发的环境下大展身手。而G. Wong的研究[3]也指出，Rust在嵌入式开发上比起传统的C语言开发在多个方面上更具优势。

2019年，图灵奖获得者John Hennessy和David Patterson在他们合著的《计算机体系结构：量化研究方法》中，提出了“计算机体系结构即将迎来新黄金年代”的观点。而且也能看到，不同架构，设计理念的操作系统正如雨后春笋一般涌现出来。韩乃平和李蕾的文章[4]分析了国产操作系统的生态建设现状。文章也指出，构建跨平台兼容的驱动程序对于推动技术自主可控具有重要意义。

## 相关领域国内外发展现状与趋势

### 国外发展现状

在欧美等地区，相关科技公司与研究机构历史较长，主导着业界工业标准的制定。如USB-IF（USB Implementers Forum）和PCI-SIG（Peripheral Component Interconnect Special Interest Group）这类国际标准组织，其成员多为大型科技企业。这些组织制定的标准不仅保障了产品兼容性，也是驱动开发的依据。但现实的情况是，对于驱动程序生态而言，如Intel、NVIDIA、AMD这类大型跨国企业往往自行开发其设备的驱动以推广自家产品。虽然也积极的参与到开源社区中，分享其成果，但也反过来产生了强者恒强的羊群效应。对于各类全新的平台，之所以难以构建其驱动生态，正是因为业界往往只会给当前已经存在的，已有了大量基础工作的unix-like与windows系统进行驱动开发。而可预见的，随着时代的发展，此类技术债问题会越积越多，若是不早以想办法解决，最终会导致计算机体系结构的基础设施创新的难度越来越高。

### 国内相关技术的发展趋势

近年来，中国在信息技术领域，特别是移动互联网、大数据及人工智能方面，发展迅速。得益于政府对科技创新的支持力度加大以及市场环境的改善，本土企业在追赶国际先进水平的同时也逐渐形成了自己的特色。

华为、中兴通讯等企业通过自主研发掌握了核心技术，包括特定场景的定制化驱动开发。但也依然存在与国外类似的问题，即对于非主流平台的生态支持程度不高。事实上，由于国内IT行业起步较晚，并未形成类似于国外那般繁荣的开源社区与文化，使得计算机体系结构的生态构建的问题更加严峻，呈现出了“头重脚轻”的问题。一些专注于嵌入式系统研发的小型创业团队正努力填补国内市场空白，他们往往聚焦于某一垂直领域深入挖掘，力求提供差异化的产品和服务，但小型企业更容易面临人手不足，从而导致社区支持力度不足的问题。除了大型企业之外，各家研究机构也加大了对于整个计算机体系结构方向的投入。而计算机体系结构是一个整体，软件与硬件之间有着联动的关系。相信随着国内硬件供应链的逐步完善，大量的硬件驱动开发需求也会随之涌现，与此同时产生的效应是大量的开源社区成果的诞生。

## 论文研究内容及主要工作

### 论文研究内容

本研究旨在设计并实现一套高效、可靠、安全的USB主机端驱动子系统，其核心组成部分与要求如下：

1. 主机控制器驱动层及其抽象层

对于当前的USB协议栈，USB主机端的工作模式与USB设备端的工作模式存在着显著的差异。具体的，在USB设备端的视角下，USB接口就仅仅只是一个IO接口而已，所需要在意的也仅仅只是数据的收发格式。而USB主机端的复杂度则明显高了很多。USB主机端往往通过特定的控制器来统一管理整个USB总线的资源。随着时代的变迁，USB主机控制器也演化出了诸多版本。因此，出于历史包袱与现实的需要，USB主机端的驱动开发通常不仅限于针对特定USB设备类型的驱动编写与管理，而是首先需要构建一个高效的USB主机控制器驱动，在此控制器的基础上去再去写具体的主机端协议驱动。更糟糕的是，由于历史原因，不同版本的USB主机控制器规范并不保持向下兼容性。而实践中厂商所生产的设备平台，其外设组合也千变万化。鉴于此，为了确保系统的兼容性和通用性，有必要将不同版本的USB主机控制器驱动抽象化为统一的接口层，为上层驱动提供统一接口，屏蔽不同控制器规范的差异，实现与外设交互。

1. USB功能驱动层

USB协议是分层设计的，其如互联网架构一般，从物理层，到链路层，再到传输层与协议层，各层的信息基本单位不同。不同层之间有一套特定的规范，规定了信息的分解与组合的方式。如用户所看见的U盘（USB块协议），USB鼠标（USB HID协议），这类协议实际上是处于协议层的应用协议。更甚者，单个物理设备可以包含着多个应用协议。对于同一个应用协议，甚至还有着不同版本之间的差异。但总而言之，在USB规范的视角下，应用协议的内部版本由协议驱动自身管理，而特定的应用协议会被抽象为一个USB Function（USB功能）。通常，一个特定的USB功能可由一个三元组唯一标识，该三元组包括Class（类别）、SubClass（子类别）和Protocol（协议）。一个设备可以存在多个USB功能，在传输层，有一套用于管理单个设备上多个USB功能的机制。鉴于存在众多不同的USB协议，并且可预见到的，未来这些协议的数量必将持续增长，因此，构建一套高效且可扩展、可灵活管理的驱动模块管理系统显得尤为必要。

该系统应能实现驱动模块的热拔插。当有着新连接的USB设备时，其应加载相应的驱动程序，实现即插即用的功能。同时，系统还应能妥善处理设备的移除，确保在设备断开连接时，能够安全的终止通信，避免数据丢失或系统稳定性。其与下层的主机驱动层应当规定一套高效的通信机制，使得应用层驱动与主机控制器协议之间不至于拥有过高的耦合度。

1. 可单线程运行的异步工作

作为IO总线，USB面临的主要问题是不同层级设备间数据传输延迟差异大。对于USB而言，通常会在同一时刻并发的对多个USB功能进行读写操作。这一现象要求系统必须采用非阻塞的工作模式，以避免任务饥饿现象的发生，进而影响用户的体验与系统的稳定。但在资源受限的嵌入式环境中，通常采用定时轮询结合中断的方式处理IO任务。进一步的，在实时操作系统（RTOS）场景中，其对实时性要求更为严苛，传统的延迟-轮询工作模式也并不适用，中断机制也会受到严格的限制。Rust语言提供了一套独特的异步编程范式，称为无栈协程。其允许IO任务在即将发生阻塞时主动让出控制权（Yield），从而实现不同于以往线程模型中，由线程控制块（PCB）为基础的栈上下文切换来实现的异步工作模式，而是基于Future机制的控制流切换，故此被称为无栈协程。这一理念使得即便在单线程环境下，甚至在缺乏中断支持的情况下，也能实现异步程序的运行。这种编程范式不仅提升了程序的运行效率，还增强了系统的响应能力，确保系统在高并发和高实时性要求的场景下能够稳定且高效的运作。

1. USB设备报文解析模块

正如前文所说，USB协议栈在传输层具有一套特定的设备控制机制，其定义了一组称为USB描述符（USB Descriptor）的控制信息格式。USB设备的协议栈的详细信息通过这些描述符得以表达，这些描述符通常以树状结构组织，具体结构可见第2.2章。目前，市场上尚未出现通用的描述符解析库。因此，本驱动子系统还须包含一套通用的解析系统以供应用层使用。同时，鉴于不同的应用层往往会添加特定的描述符结构，描述符解析系统也应同USB功能驱动层一样，为不同的应用层协议提供描述符解析模块的热拔插机制。

1. 事件总线及其订阅机制

动态调试系统组件是工业领域的重要研究课题。研究者们通常期望在组件正常运行期间，对其性能进行深入的评估。如eBPF，XDP等技术，为了实现这一目标，这类技术通常会在代码中插入一种特殊的注入点。当程序运行到该注入点时，会触发挂载于该注入点上的外部代码的调用，从而探查进程的上下文变量，甚至改变原始运行逻辑。这种注入的代码逻辑通常被称为“钩子（Hook）”。通过钩子技术，开发者能够在不修改原有代码的基础上，对系统的运行行为进行干预和控制，实现更灵活的功能扩展和性能优化。

对于典型的unix类的系统，一类常见的需求就是将外部IO设备挂载至文件上。这要求系统能够监控对应块设备驱动程序的工作状态，并根据需要发起请求。因此，USB系统必须向外部提供相应的操作接口。在直觉上，往往会倾向于直接向外暴露出一个驱动设备的索引表。问题在于，对于驱动来说，各个驱动及各个驱动的各个驱动设备实例往往是互不相关且状态大小均可变的。因此，直接向外暴露出索引表，但索引表内的指针类型却实际上不同，从而导致了大量的问题。并且，这样的设计还不是进程安全的。这便产生了更进一步探索的价值。

为了满足这一类需求，本系统采用事件驱动下的外部闭包注入的方式来实现外部逻辑注入的目的。事件驱动模型是一种编程范式，它允许系统在特定事件发生时，如设备连接、数据传输请求等，自动触发相应的处理函数或方法。在本系统中，存在着一条逻辑上的事件总线。各驱动设备实例及系统本身的各部分模块都会向该事件总线发布带有上下文引用的事件以供订阅了该事件的外部逻辑进行操作。通过这种方式，USB系统可以更加高效和灵活的响应外部设备的变化，确保数据的正确传输和设备的稳定运行。通过事件系统将平台的耦合逻辑编写事件处理器，亦可以保证系统本身与平台代码上的解耦与系统本身的可扩展性。

### 论文主要工作

本文围绕跨平台USB主机端驱动子系统的设计与实现展开研究，主要工作涵盖理论分析、架构设计、模块实现以及实验验证等方面，主要的内容如下：

第一章，绪论。本章系统阐述了USB驱动开发的背景与挑战，分析了国内外技术发展现状，明确了跨平台兼容性、安全性及异步执行效率等核心问题。通过文献调研，提出了基于Rust语言构建跨平台驱动框架的可行性，并规划了论文的研究方向与目标。

第二章，跨平台USB主机端驱动子系统的整体设计方案及原理概述。本章提出了分层架构设计，包含主机驱动层、USB驱动层和系统抽象层。重点设计了驱动无关的设备抽象层与统一的主机控制器接口（Trait），实现了对不同版本USB主机控制器的兼容。此外，结合Rust的异步编程模型，提出了一种基于Future的无栈协程调度机制，支持单线程环境下的异步任务主动让出与轮询切换，降低了多线程资源竞争风险。

第三章，硬件与仿真环境。本章构建了跨平台测试验证体系，在ARM架构（飞腾派硬件平台与QEMU仿真）和RISC-V架构（QEMU仿真）上完成系统部署。设计了多种测试场景，包括USB设备热拔插、高并发数据传输以及异常状态恢复，并通过USB协议分析仪对数据传输的完整性与实时性进行验证，确保系统在不同硬件环境下的稳定运行。

第四章，软件程序设计与实现。本章基于Rust语言实现了驱动框架的核心模块，包括：

1. 主机驱动层：开发了XHCI控制器驱动模块，支持事件环、控制环与传输环的动态管理；
2. USB驱动层：设计了可热拔插的驱动模块管理系统，通过统一的行为规范实现驱动实例的动态加载与卸载；
3. 描述符解析系统：自主开发了支持动态解析的USB描述符树处理模块，兼容厂商自定义描述符与复杂拓扑结构；
4. 事件总线机制：构建了基于订阅-发布模式的事件系统，实现驱动层与外部组件的高效交互。

第五章，系统测试。本章对USB系统的各模块进行了详细的单元测试，并逐步剖析了代码的主要逻辑结构，其中包括：

1. USB描述符解析器的单元测试：对所编写的USB描述符解析器模块在描述符树的多个层次上进行了测试，以验证其解析结果的正确性。
2. USB驱动子系统的软硬件协同测试：基于第三章所引入的跨平台测试验证体系，在实际环境与虚拟环境下展开了对USB驱动子系统的功能测试，验证了主机驱动层在实际硬件与虚拟硬件上的兼容性。并编写了实验性的HID设备驱动以验证热拔插驱动模块的正确性及对应接口设计的合理性。
3. USB驱动子系统在进一步测试过程中所暴露出的问题与解决方案：对在测试过程中所发现的问题进行了全面分析，找出了问题发生的原因，并通过所提出的解决方案修复了问题。
4. 对于HID协议即USB-CDC协议的驱动实例功能验证，及间接的对于USB驱动层的功能验证。
5. 一个落地应用的实例，即使用USB-HID协议去控制一台小车运动。

# 跨平台USB主机端驱动子系统的整体设计方案及原理概述

## 系统整体方案设计

图2.1所示系统架构包含三个主要层次：主机驱动层、USB驱动层及系统抽象层。主机驱动层与USB驱动层之间设有驱动无关的设备抽象层，存储特定USB设备的基本信息，包括共享描述符、请求缓冲区及控制传输状态锁。这些信息供USB驱动层各实例通用，并通过系统抽象层的事件总线与外部交互。主机驱动层与USB驱动层间异步运行，确保系统高效响应。在USB驱动层内部，每个具体的驱动实例都持有一份对驱动无关设备的引用，并将发起的请求组织成USB请求块（URB）的形式。主机驱动层负责不断的从各个设备的请求缓冲区中获取这些USB请求块，并将它们批量的执行。采用USB请求块(URB)传递请求，而非直接方法调用，可避免多层调用中的频繁加锁，减少访问竞争，确保任务响应公平性。

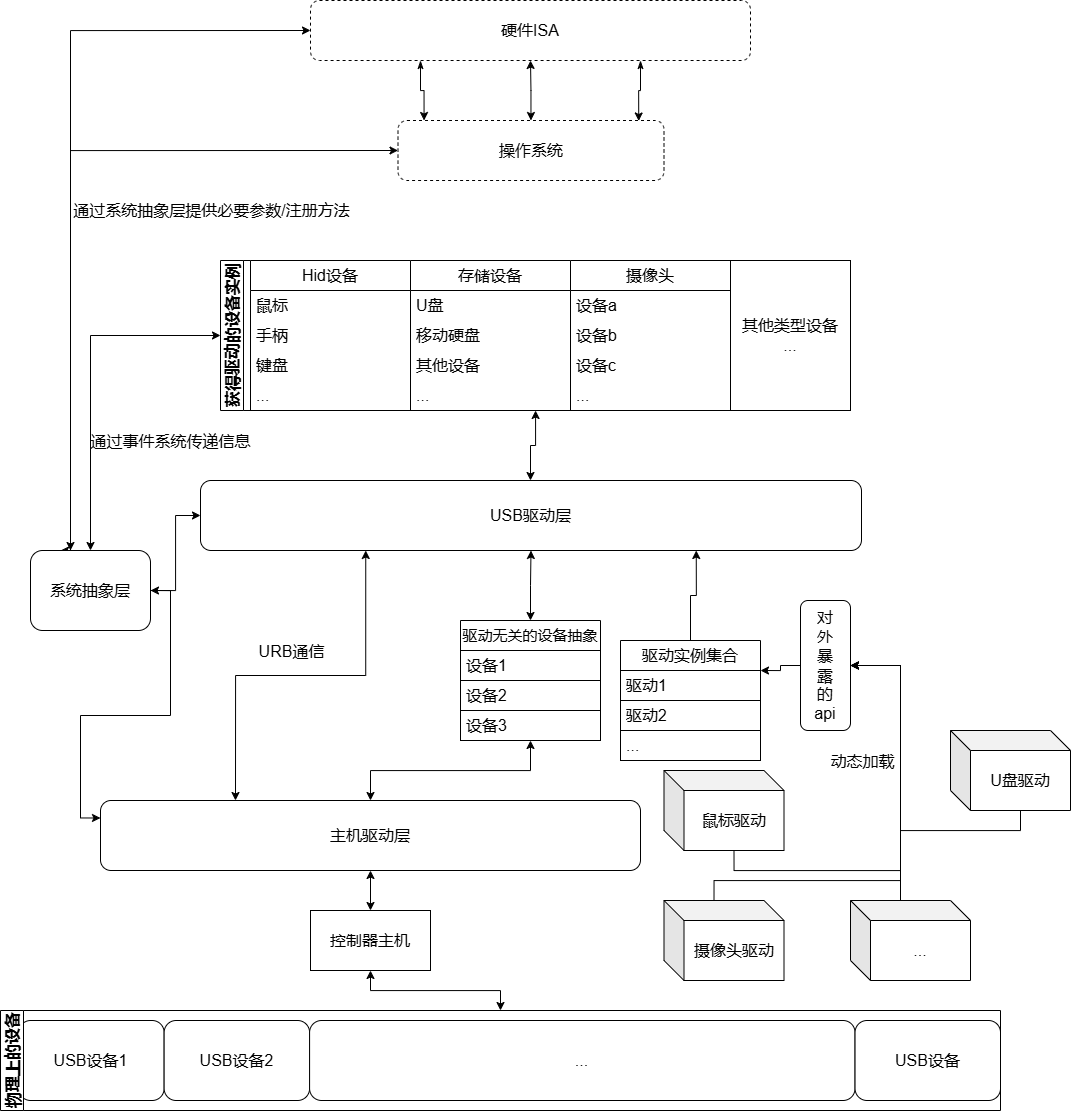


图 2.1 系统总体设计方案

## 下文所涉及概念与名词解释

1. USB描述符，即一组用于描述USB硬件设备信息的报文段，其拓扑结构如图2.2所示，其中主要部分包括：
   1. USB设备描述符，此描述符内主要包括该设备的制造厂商，设备型号，USB协议物理层相关参数，及设备的不同配置模式数量。
   2. USB配置描述符，是USB设备描述符的下级节点，不同的配置标志着设备的不同工作模式，一个配置下面会一组不同的USB功能，即一个设备可能同时有多个功能。
   3. USB接口描述符，是USB配置描述符的下级节点，每个接口描述符都是一个独项工作的USB功能(USB Function)，包含着描述功能本身的三元组。每个USB功能都可能存在不同的配置，因此其还额外的包含了表示当前处于何种配置模式的二元组 (Interface Number, Alternative Interface Numver)。 其中，前者为每个配置下全局唯一的接口编号，后者为每个接口的不同工作模式的编号。
   4. USB端点描述符，是USB接口描述符的下级节点，每个USB功能都需要占用USB设备上的USB端点，及物理信道。该描述符包含所用端点的编号，类型，数据包大小等物理信息。
2. USB端点，即设备上的物理端点，每个端点都是独立的通信信道。端点有四种类型，分别对应一种特定的数据传输模式，适应于不同的任务。

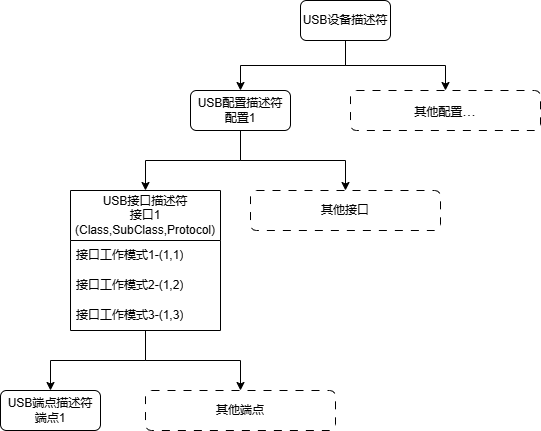


图 2.2 USB描述符拓扑结构

1. 在Rust异步编程中，Future指封装为状态机的异步计算单元。Future作为状态机，有三种状态：
   1. Pending，该状态表明Future内的异步代码块正在执行阻塞IO操作。
   2. Waiting，该状态表面Future内的异步代码块已经完成了先前的阻塞IO操作，正在等待被再次调度以继续执行。
   3. Ready，该状态表面Future的任务已经彻底完成，此时Future指向了其内部异步代码块的最终返回值，等待被调用者回收。

Future引用一片内存区域，Future内的异步代码块正是对该片内存区域进行操 作的逻辑，而Future本身并非存放数据的结构体。

1. Cargo.toml配置文件，Rust拥有一套完善的项目管理机制，单个完整的Rust项目被称为Crate，大多数Rust开发者们使用 Cargo来管理他们的Rust项目。Cargo可以处理很多任务，比如构建代码、下载依赖库，以及编译这些库。Cargo.toml是Cargo所用的，描述一个Crate的各类属性的文件，其内包含该Crate所引用的库，该Crate的可选功能（feature），及作者等信息。
2. Rust异步编程，其思想类似于传统的多线程/多进程编程，其主要目的是解决IO所带来的高额延迟开销，在当前代码片段因IO而进入阻塞时，主动将资源让出，转而去执行其他与当前IO资源不直接相关的可执行代码段，大大提升了编程灵活性。Rust的异步编程采用的是无栈协程的思想，其将异步代码段组织为可嵌套的Future，Future并不会主动执行自己，而是要由外部去执行。Future本身封装为状态机，如图2.3所示，当Future由于IO而进入阻塞时，便将自身的状态转为Pending，外部的调用者便会将该Future暂存起来，转而去执行其他Future，当时机来临时，回来执行可继续执行的Future。

调用者可以是一个封装好的运行时——类似于操作系统中的任务调度器，或者 也可以通过良好的设计，使得外部调用者以阻塞的方式执行最上层的Future。

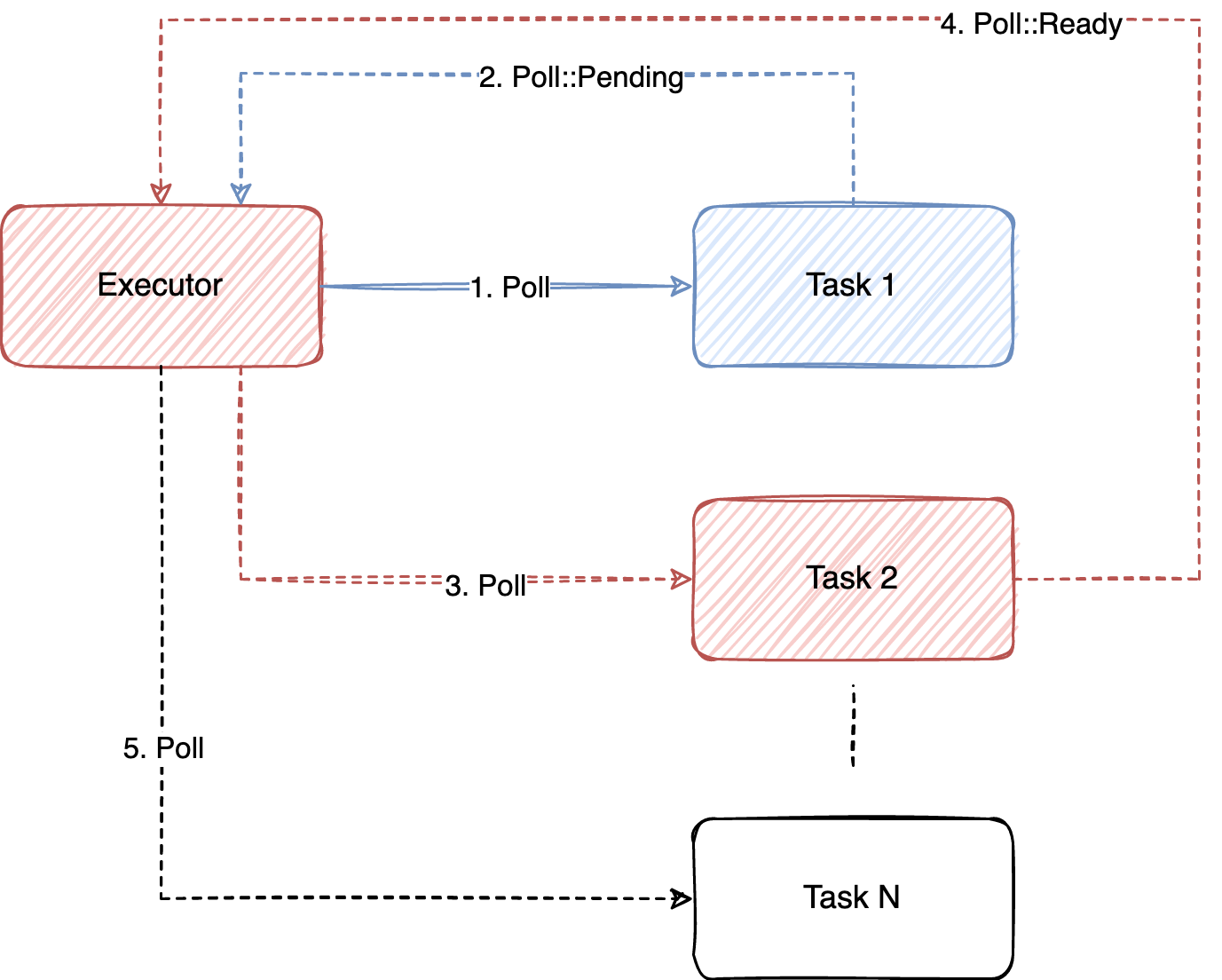


图 2.3 Rust的异步执行模型

## 系统各模块具体的技术研究

### 主机驱动层的技术细节

1. XHCI主机控制器

XHCI（eXtensible Host Controller Interface）是目前主流的USB HOST控制器标准，其作为第三代USB主机控制器被设计出来，并向下兼容USB1.0/USB2.0规范。其设计使得上层USB驱动的部分与XHCI控制器本身的驱动解耦，提高了灵活性。同时XHCI从逻辑上是事件驱动的，对高IO，高并发的情况更为友好。杨志荣等人[5]介绍了FPGA的xHCI协议软核IP设计及其应用，这为理解USB主机控制器接口提供了基础。C. Wang探讨了通过多线程编程分析硬件和软件并行性的局限性[6]，这对优化USB数据传输机制具有指导作用。张帅林在其博士论文中详细描述了基于嵌入式处理器的高性能中间件软件的设计与实现[7]，这对于提升USB驱动的整体性能至关重要。刘巍与黄智勇研究了DDS中间件的跨平台实现与优化技术[8]，为构建高效稳定的驱动组件提供了参考。王硕与胡飞的工作则集中在基于数据发布订阅服务的嵌入式通信中间件设计上[9]，有助于探索事件驱动模型在USB驱动中的应用。

XHCI控制器设计的核心理念就是事件驱动与状态机。其工作是围绕三种环形缓冲区与本身的状态展开的，即事件环，控制环与传输环。在这三种环中，传输的信息被封装为一个个的TRB(Transfer Request Block)，其工作逻辑可简要的概括为：

* 1. 控制环内容纳改变主机控制器本身状态的TRB请求。
  2. 传输环容纳主机端所发起的USB传输请求，每个设备的每个端点都有一个传输环。
  3. 当向环内填充好了请求之后，置位门铃寄存器（Doorbell Register）以通知主机控制器开始处理这一轮请求。
  4. 当某个请求完成时，主机向事件环填充一个标志着请求处理结果的TRB，如果配置了中断寄存器且请求时要求以中断的形式进行通知，则通过中断唤醒对应的事件处理程序，否则，由程序负责轮询并处理事件环中的新事件。

1. 驱动无关的设备实例

如前文所述，驱动无关的设备实例位于USB驱动层与主机驱动层之间。对于USB驱动层而言，USB主机所做的工作是透明的。所有请求均被导入设备实例的缓冲区。在本系统架构中，每个设备实例均维护一个环形缓冲区。设备实例在此扮演生产者（Producer）的角色，而主机驱动层则承担消费者（Consumer）的角色。主机驱动层定期的从各个设备实例中获取设备请求，并集中处理。设备实例也保存USB设备的描述符信息，各个USB驱动正是基于这些描述符信息开发的，并且每个驱动仅引用描述符中与自己功能相关的接口部分，这些接口部分彼此之间独立，互不干扰。这种设计确保了即使有若干特定的驱动模块发生崩溃，也不会对整个USB子系统的稳定运行造成影响。之所以仅保存一份描述符树，其原因在于描述符树往往会很大，复制会产生很多额外的开销，并且描述符树本身不会被变动，在逻辑上属于常量。

1. 需要不断发起请求的特殊事务

在USB技术的应用范畴内，众多上层功能协议常常表现出一些相同的需求。以HID协议为例，无论是何种HID设备，其总是频繁的重复发起相同的报文请求，并总是会收到大量的空回复。HID协议使用的是中断传输（Interrupt Transfer）这一传输层所规定的传输类型，这是一种一次只传输一份数据的传输类型，其特点在于，中断传输并不要求立即返回响应，而是当外设发生了对应的事件后，再返回报文。这种类似于中断的工作模式正是中断传输这一名称的由来。然而，在实际应用中，外设在没有事件，或事件结束后的第一次请求时，通常会选择直接返回一条空报文。而HID驱动在接收到报文后会再度发起请求，这就导致了大量无意义的带宽被浪费。用户通常对HID设备的响应速度有一定的期望和要求，因此这类损耗是难以接受且不必要的。类似于这一类的情况还有很多，其特征都如同HID一样，主要的开销来自于反复的重复请求。也就是说，需要让主机驱动层承担起一部分责任，由主机驱动层来不断的发起请求，并且负责过滤掉那些无意义的消息。实际上，主机驱动层会维护一个请求完成后的收尾工作集合，每个请求块都携带有一个唯一的编号，与集合中的收尾工作一一对应。这样的设计模式带来的好处在于，其统一了各类请求的处理流程。对于那些需要不断发起请求的特殊事务，主机驱动层同样会按照这种方式进行管理。

1. 主机驱动层的异步运作模式

从宏观视角审视主机驱动层，其功能可抽象化为图2.4所描绘的模型。在此模型中，主机驱动层负责异步的执行请求的发送以及收尾工作的处理。收尾工作的处理策略因USB子系统驱动框架所处的工作环境差异而异。当请求完成时，若已配置中断机制，则中断处理函数将被激活以执行处理。若未配置中断，则系统将通过轮询机制持续检查请求的完成状态。根据实时性要求的不同，系统将采取定时轮询或主动让出轮询的策略进行资源的合理分配，以确保系统的高效运作。

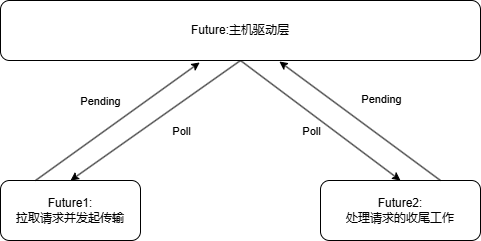


图 2.4 主机驱动层的工作模型

### USB驱动层的技术细节

1. 可热拔插的USB驱动模块

在嵌入式系统领域，鉴于资源的有限性和特定应用需求，通常不具备完整的文件系统。因此，在此类环境中，传统的基于文件系统的动态链接库加载机制并不适宜。实际上，为了满足嵌入式环境的特殊要求，驱动子系统的设计必须保证其始终在内存中常驻，并由操作系统内核直接进行管理。由此，依赖文件系统来加载和管理驱动模块的需求得以免除。核心在于，必须构建一套机制，使得系统能够在运行时动态的添加或移除USB驱动模块。换言之，应当明确一套USB驱动模块必须遵循的行为规范，确保所有遵循该规范的驱动模块在内存中的实例均具有统一的内存布局。这样，操作系统内核便能够无缝的管理这些驱动模块，而无需关注它们的加载时机及方式。

1. USB驱动模块的工作模式

USB驱动模块的工作过程可以被划分为两个核心组成部分：首先是USB驱动模块实例的创建，其次是USB功能实例的构建。USB驱动模块实例本身并不直接执行任何实际的驱动代码，它的主要职责和功能在于对USB描述符进行筛选和分析，以确定目标设备是否与该驱动模块相兼容。一旦通过了兼容性检查，确认目标设备与驱动模块匹配，USB驱动模块实例就会承担起一个重要的任务，那就是负责创建和初始化相应的USB功能实例。这个实例化过程是USB驱动模块能够正常工作和与设备通信的关键步骤，确保了驱动模块能够根据设备的特定需求来调整其行为，从而实现高效的数据传输和设备管理。

1. USB功能实例的工作模式

USB功能实例由两个核心组成部分构成：USB功能实例结构体以及对应的封装为Future的运行逻辑。USB功能实例结构体的布局和设计会根据不同的USB驱动而有所差异。在USB驱动层，它会负责拉取所有的USB功能实例结构体中定义的逻辑Future，并对这些Future进行执行。这一过程在图2.5中得到了清晰的展示，图中详细描绘了USB功能实例结构体与Future运行逻辑之间的关系以及USB驱动层如何处理这些结构体。

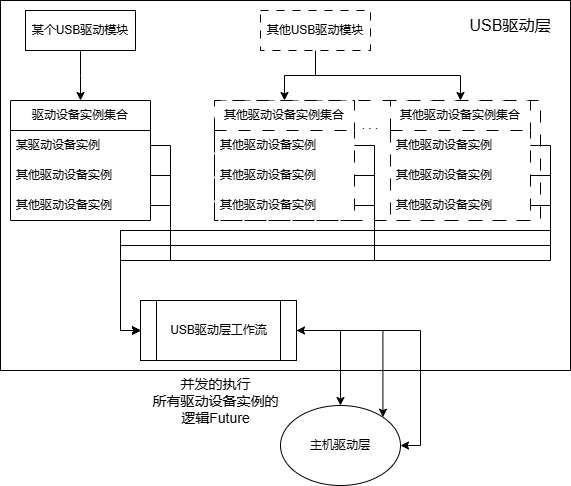


图 2.5 USB驱动层工作流

1. 事件系统

如图2.6所示，事件系统架构的设计允许USB驱动子系统通过事件总线与外部环境进行交互。事件本身携带有上下文信息，这些信息对于事件处理函数的执行至关重要。以USB移动存储设备的创建为例，当一个新的USB Block Device驱动实例被创建时，系统会触发一个名为USB\_NEW\_BLOCK\_DEVICE\_EVENT的事件。该事件中包含了丰富的上下文信息，包括对新创建的驱动实例的引用。操作系统在接收到此事件后，能够利用上下文信息中的驱动设备实例引用，执行挂载操作，将该驱动设备实例挂载至文件系统，从而实现对USB移动存储设备的识别与管理。

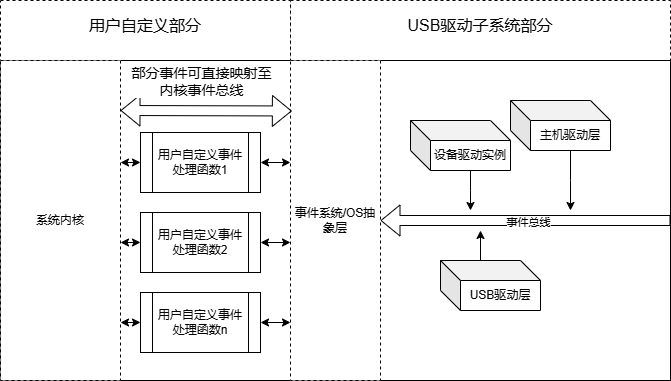


图 2.6 事件系统架构

## 本章小结

本章提出了分层架构的跨平台USB驱动子系统设计方案，包含主机驱动层、USB驱动层与系统抽象层。通过定义驱动无关的设备抽象层与统一的主机控制器接口，实现了对不同版本USB主机控制器的兼容支持。系统采用Rust语言的异步编程模型，结合Future无栈协程调度机制，支持单线程环境下的任务主动让出与轮询切换，有效降低多线程资源竞争风险。核心模块设计中，主机驱动层通过事件环、控制环与传输环协同管理，实现高效请求处理；USB驱动层则通过热拔插模块化设计，确保驱动实例的动态加载与安全隔离。此外，事件总线机制的设计强化了系统与外部组件的交互能力，为即插即用功能提供了技术保障。

# 硬件与仿真环境

## 开发所用平台环境

本研究旨在确保系统的跨平台及跨架构兼容性，因此必须在多种硬件架构和操作系统上进行广泛的测试验证。目前，该系统已在基于ARM架构和RISC-V架构的平台上完成了测试。

### 对于ARM架构的测试与验证

对于ARM架构，系统在飞腾派硬件平台（参数见表3.1）及QEMU仿真环境下测试。飞腾派由中电港萤火工场与飞腾公司联合开发，采用其自研高能效异构多核处理器技术。其具体技术规格如下表3.1所示

表 3.1 飞腾派参数列表

| 参数名 | 参数描述 |
| --- | --- |
| CPU | 飞腾四核处理器，兼容ARMv8指令集，2×FTC664@1.8GHz+2×FTC310@1.5GHz |
| 内存 | 64位DDR4，分2G版本和4G版本 |
| 硬盘 | 支持microSD 和EMMC启动 |
| 网络 | 2x千兆以太网（RJ45） |
| USB | 1×USB 3.0 host，3xUSB 2.0 host（键鼠） |
| PCIe | 1×Mini-PCle，支持AI、5G/4G、SATA转换等模组 |
| 蓝牙 | 蓝牙BT 4.2/BLE 4.2 |
| WiFi | 2.4G + 5G 双频WIFI |
| 4G/5G | 可通过miniPCIE扩展实现 |
| AI加速 | 可通过miniPCIE扩展实现 |
| 显示 | 1x HDMI，支持分辨率1920\*1080 |
| 视频解码 | 支持VPU硬解，格式：H.265\264 |
| 音频 | 3.5mm耳机口输出 |
| UART | 1x调试串口+2xMIO (可配置多功能IO口为UART模式) |
| 12C | 2+2xMIO (可配置多功能IO口为I2C模式) |
| 12S | 1路 |
| SPI | 2路 |
| CAN | 1路CANFD |
| GPIO | 最多29个 |
| SIM卡 | 支持1路SIM 卡 |
| SD卡 | 支持1路SD卡 |
| LED灯 | 电源指示灯和状态指示灯 |
| 供电要求 | 12V 3A直流电源 |
| 产品尺寸 | 119mmx93.1mm |
| 重量 | 72±2g |
| 工作 (运行) 温度 | 0~50°C |

QEMU仿真所使用的环境为QEMU-system-aarch64，其余设置如表3.2所示

表 3.2 ARM架构的QEMU仿真参数

| 参数 | 参数值 |
| --- | --- |
| -m | 2G |
| -smp | 1 |
| -cpu | Cortex-a72 |
| -machine | virt,highmem=off |
| -device | QEMU-xhci,id=xhci |
| -device | usb-mouse,bus=xhci.0 |

### 对于RISC-V架构的测试与验证

对于RISC-V架构，使用QEMU-system-riscv64进行仿真测试，参数配置见表3.3。

表 3.3 RISC-V架构的QEMU仿真参数

|  |  |
| --- | --- |
| 参数 | 参数值 |
| -m | 2G |
| -bios | default |
| -machine | virt,highmem=off |
| -device | QEMU-xhci,id=xhci |
| -device | usb-mouse,bus=xhci.0 |

## 测试所用硬件设备

对于USB系统的测试，所使用的是USB总线分析仪进行协议栈测试，如图3.1，图3.2所示，此外，还使用实物设备进行功能测试。

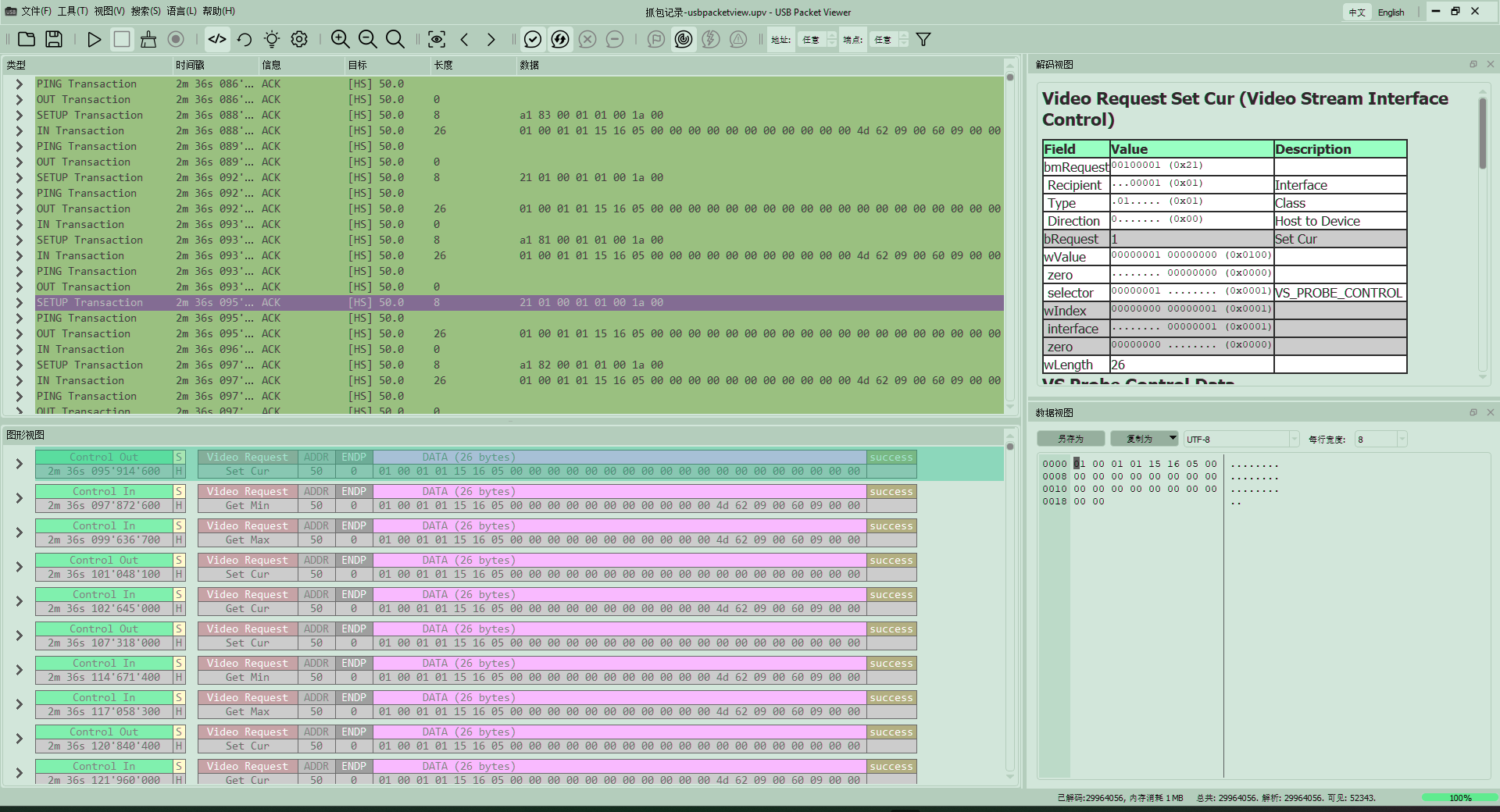


图 3.1 USB协议分析仪软件界面



图 3.2 USB总线分析仪实物

## 本章小结

本章构建了跨架构的硬件与仿真测试体系，以验证系统的兼容性与稳定性。测试环境覆盖ARM架构（飞腾派硬件平台与QEMU仿真）和RISC-V架构（QEMU仿真）以验证系统的可移植性。

# 软件程序设计与实现

## 程序设计语言及开发环境

### 编程语言：Rust程序设计语言

系统使用Rust语言开发。Rust是一种由Mozilla基金会开发的现代系统编程语言。其核心设计理念‘安全、并发、高效’，旨在解决C/C++等语言中的内存安全与并发挑战。Rust 通过所有权系统（Ownership System）、借用检查器（Borrow Checker）和生命周期（Lifetime）等独特机制，在编译时确保内存安全，从而避免了空指针、悬垂指针和数据竞争等问题。这些机制使得 Rust 在编写高并发、高可靠性和高性能的软件时表现出色。此外，Rust 的生态系统非常活跃，拥有丰富的库和工具（如 Cargo 包管理器），由于其编译器基于LLVM进行开发，这使得Rust能够轻松的编译到不同的硬件平台上。Rust 被广泛应用于系统编程、嵌入式开发、高性能计算等领域。

### 工具：Makefile

Makefile是一种用于自动化构建软件项目的工具，广泛应用于 C/C++、Rust 等编程语言的项目中。它是基于make工具的脚本文件，通过定义一系列规则和命令，帮助开发者自动化编译、构建、测试和部署等流程。在Rust项目中，虽Cargo是官方推荐的构建工具，但Makefile仍然在一些复杂场景中发挥重要作用。当需要自定义构建逻辑、管理多个构建目标或与其他工具链集成时，其能提供更高的灵活性和控制力。

### 集成开发环境：VSCode

Visual Studio Code（简称VSCode）是一款由微软开发的轻量级、开源且高度可定制的代码编辑器，广泛应用于多种编程语言和开发场景。它以其卓越的性能、丰富的插件生态和跨平台支持而著称，能够满足从个人开发者到大型团队的多样化需求。

* VSCode插件：Rust Analyzer

Rust Analyzer是一个专为Rust编程语言设计的先进语言服务器协议 (LSP) 实现，它通过提供智能感知、代码补全、错误检测、重构建议和文档查找等功能，显著提升了开发者的编码效率和体验；基于Rust的强大类型系统和所有权模型，Rust Analyzer能够深入理解代码结构，从而实现精准的上下文感知和高效的代码分析；此外，它还支持与多种主流IDE和编辑器（如VS Code、IntelliJ IDEA和 Neovim）无缝集成，并通过持续优化性能和扩展插件生态系统，已成为Rust开发者不可或缺的工具之一。

## 模块化设计与项目结构

基于功能与复用性考量，USB子系统划分为两个Crate：负责描述符解析的usb-descriptor-decoder和核心架构axusb-host。axusb-host的Cargo.toml配置文件如图4.1所示。

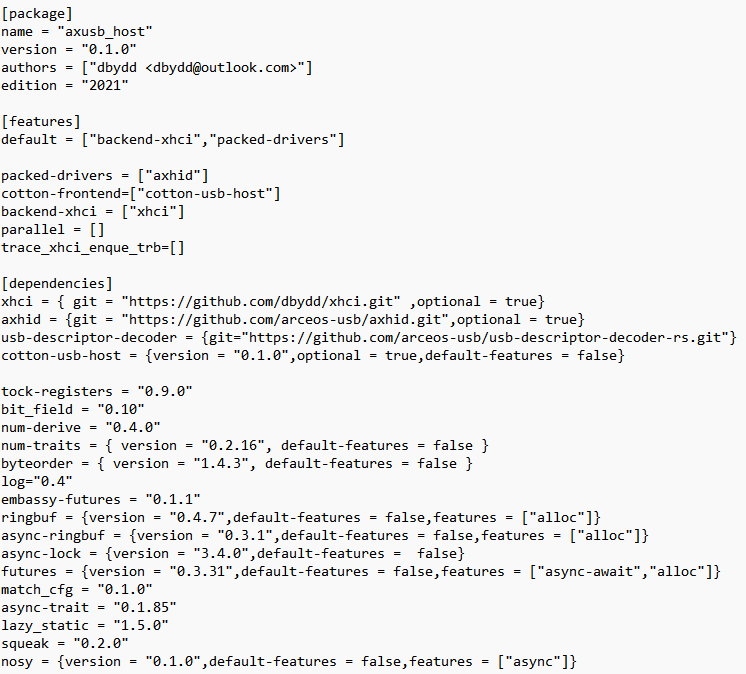


图 4.1 axusb-host Crate的Cargo.toml配置文件

注1） xhci Crate为修改版本，添加了辅助性的函数，并修复了若干潜在的BUG。

注2） axhid Crate为额外编写的为USB-HID设备提供报文解析的工具库。

usb-descriptor-decoder的Cargo.toml配置文件如图4.2所示。

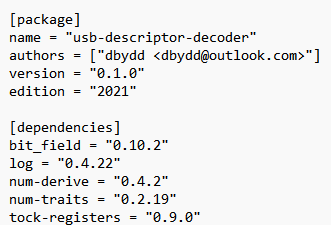


图 4.2 usb-descriptor-decoder Crate的Cargo.toml配置文件

## USB驱动子系统框架的实现

作为驱动子系统的主体框架，axusb-host的模块组织如图4.3所示：

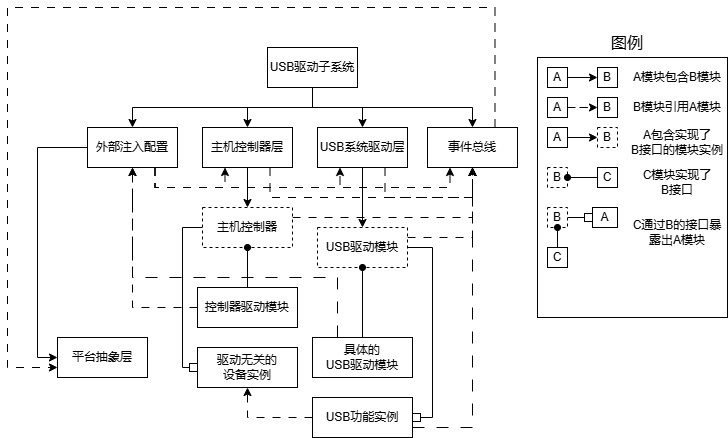


图 4.3 USB驱动子系统的模块组织

### USB驱动子系统模块

作为子系统的最外层封装，USBSystem结构体通过事件总线管理主机控制器层与USB驱动层的交互。其定义如下：

pub struct USBSystem<'a, O, const RING\_BUFFER\_SIZE: usize>

where

O: PlatformAbstractions + 'static,

'a: 'static,

{

config: Arc<USBSystemConfig<O, RING\_BUFFER\_SIZE>>,

controller: Box<dyn Controller<'a, O, RING\_BUFFER\_SIZE>>,

usb\_layer: USBLayer<'a, O, RING\_BUFFER\_SIZE>,

event\_bus: Arc<EventBus<'a, O, RING\_BUFFER\_SIZE>>,

}

其中Arc为线程安全的智能指针，USBSystemConfig为包含了常量级系统超参数的结构体，Box为指向堆上内存的指针，Box<dyn Controller>表明这个Box指针指向了一个Controller Trait的VTable结构体。任何数据结构，只要实现了Controller Trait，就能创建对应的VTable结构体类型，从而被外部调用。’a是生命周期参数，这里’a要求至少派生自’static周期，意味着USB子系统应当常驻内存。

### 主机控制器接口

如图4.3所示主机控制器层并不直接持有特定的主机控制器驱动，而是通过定义了一组通用的行为，而达到可以使用任何实现了该行为的主机控制器驱动的目的。这组行为的定义如下：

pub trait Controller<'a, O, const RING\_BUFFER\_SIZE: usize>: Send + Sync

where

O: PlatformAbstractions,

{

fn new(

config: Arc<USBSystemConfig<O, RING\_BUFFER\_SIZE>>,

event\_bus: Arc<EventBus<'a, O, RING\_BUFFER\_SIZE>>,

) -> Self

where

Self: Sized;

fn init(&self);

fn device\_accesses(&self) ->

&Vec<Arc<USBDevice<O, RING\_BUFFER\_SIZE>>>;

fn workaround(&'a self) -> BoxFuture<'a, ()>;

}

所谓VTable，就是一个由函数指针组成的结构体，其中各个函数指针的名字与定义其结构的Trait中的方法名字相一致。这种设计模式正是面向对象编程范式的底层实现方式。此处给出了Controller Trait的定义，其中new为创建dyn Controller的构造方法，init为控制器初始化所要做的工作，device\_accesses为预留的接口，其目的向外部扩展层暴露出当前主机控制器所管理的物理设备，workaround返回的是一个异步Future，该Future包含了主机控制器层初始化完成后的封闭状态机，会在上层调用中以结构化并发的方式组织任务调度。

### USB系统驱动层模块

USB系统驱动层（USBLayer）管理多个驱动模块及其实例，其结构体定义如下：

pub struct USBLayer<'a, O, const RING\_BUFFER\_SIZE: usize>

where

O: PlatformAbstractions,

{

config: Arc<USBSystemConfig<O, RING\_BUFFER\_SIZE>>,

eventbus: Arc<EventBus<'a, O, RING\_BUFFER\_SIZE>>,

pub driver\_modules: BTreeMap<

String,

Box<dyn USBSystemDriverModule<'a, O, RING\_BUFFER\_SIZE>>

>,

pub functional\_interfaces: RwLock<

BTreeMap<String,

Vec<Arc<RwLock<

dyn USBSystemDriverModuleInstanceFunctionalInterface< 'a,O>>>

>

>,

>,

pub dynamic\_join\_array: Arc<DynamicJoinArray>,

}

可以看到，其中driver\_modules为一个Map类，其关键字为模块的名称，类型为String，对应的值为实现了USBSystemDriverModule Trait的实例，而functional\_interface为由驱动模块所创建的驱动设备实例，同样的由Map结构组织，键为创建该组驱动设备的模块名，值为一个 Vec 类型的可变长数组，其内包含了对应模块所创建的所有实例。驱动设备创建时，会同步的创建一个引用了该驱动设备的，包含了其内部驱动逻辑的Future，并将该Future提交至dynamic\_join\_array，即一个可变长的Future任务池。

### 核心思想：结构化并发调度

结构化并发调度，在Rust编程实践中，是一种将异步代码块组织为状态机的编程思想。在Rust中，异步代码块被封装为Future，Future并不会主动的运行自身，而是需要由外部的Future进行推动运行。因此，将Future逐层嵌套，就可以创建出一个分层运行的状态机。这部分通常是由join!宏实现的，但是对于运行时Future数量可变的任务池来说，编译期固定了参数个数的join!宏并不适用，因此自行编写了DynamicJoinArray任务池，其核心逻辑如下：  
pub struct DynamicJoinArray {

job\_queue: RwLock<BTreeMap<usize,BoxFuture<()>>>,

cycle\_counter: RwLock<usize>,

}

impl<'a> Future for DynamicJoinArrayFuture<'a> {

type Output = ();

fn poll<'b>(

self: core::pin::Pin<&'b mut Self>,

cx: &mut core::task::Context<'\_>,

) -> core::task::Poll<Self::Output> {

let this = unsafe { self.get\_unchecked\_mut() };

match this.ref\_to\_array.job\_queue.try\_write() {

Some(mut queue\_ref) => {

let to\_drop = queue\_ref

.iter\_mut()

.filter\_map(

|(id, job)| match unsafe { Pin::new\_unchecked(job).poll(cx) } {

core::task::Poll::Ready(\_) => Some(id.clone()),

core::task::Poll::Pending => None,

},

)

.collect::<Vec<\_>>();

if !to\_drop.is\_empty() {

to\_drop.iter().for\_each(|to\_drop\_e| {

queue\_ref.remove(to\_drop\_e);

});

}

}

None => {}

}

Poll::Pending

}

}

其中，poll方法即为该任务池所对应的Future在每次被从poll时所进行的任务，其逻辑可描述为：对于内部的每个Future，将其上下文暂时保存到本地栈上，然后执行该Future的逻辑。若该Future返回值为Poll::Pending，则将其放回调度队列等待下次的poll，否则将其从调度队列中删除。在对当前调度队列中的所有任务完成一次poll后返回Poll::Pending。

对于这样的代码组织方式，不难看出对于最外层的Future，依然以非异步编程的方式去定期的调用最外层的poll方法。虽然在最外侧看来，整体依然处于阻塞运行的状态，但是程序内部却实现了无异步调度器的上下文任务切换。

## USB描述符解析系统的实现

### USB描述符的数据流

USB描述符在逻辑上组织为具有特定拓扑的树状结构（描述符树）。传输时，描述符按特定顺序以字节流传递。字节流起始部分为描述符树前序遍历序列，后接厂商自定义描述符。如图4.4所示，每个USB描述符起始处包含两个定长字段：长度和类型。

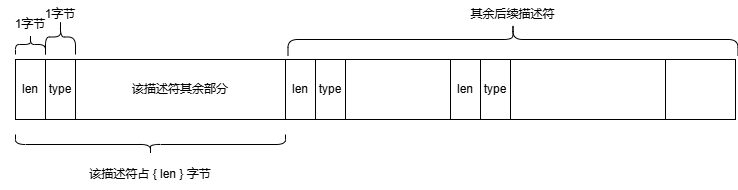


图 4.4 USB描述符字节流

特别需要指出的是，在USB接口描述符的后面，除了包含端点信息之外，通常还会附加一些特定于该USB接口描述符所对应的USB协议栈的特殊信息。在一些更为复杂的情形中，接口描述符可能会进行嵌套，从而形成一个更为庞大的USB功能实例。一个典型的例子就是UVC协议栈，即USB Video Class（USB视频类设备）。其扩展后的描述符树结构如图4.5所示。

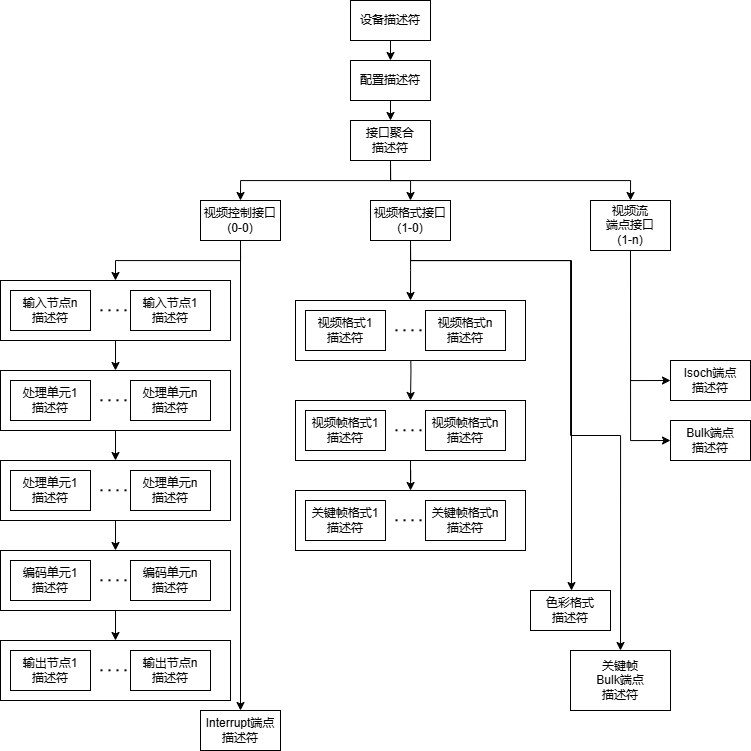


图 4.5 UVC协议栈的描述符拓扑

其中，接口聚合描述符负责将下层接口聚合为单个USB Function类。

### USB描述符解析系统的设计

鉴于USB描述符树在接口层面上的复杂性，以及不同厂商和协议栈之间私有描述符的类型编号可能存在冲突，这就意味着只有在成功解析出接口描述符之后，方能明确接下来将会遇到何种格式的描述符子树。因此，为了处理这些复杂性并确保系统的灵活性和可扩展性，有必要提出一套合理，高效且可扩展的解析流程。USB描述符解析系统的解析流程如图4.6所示：

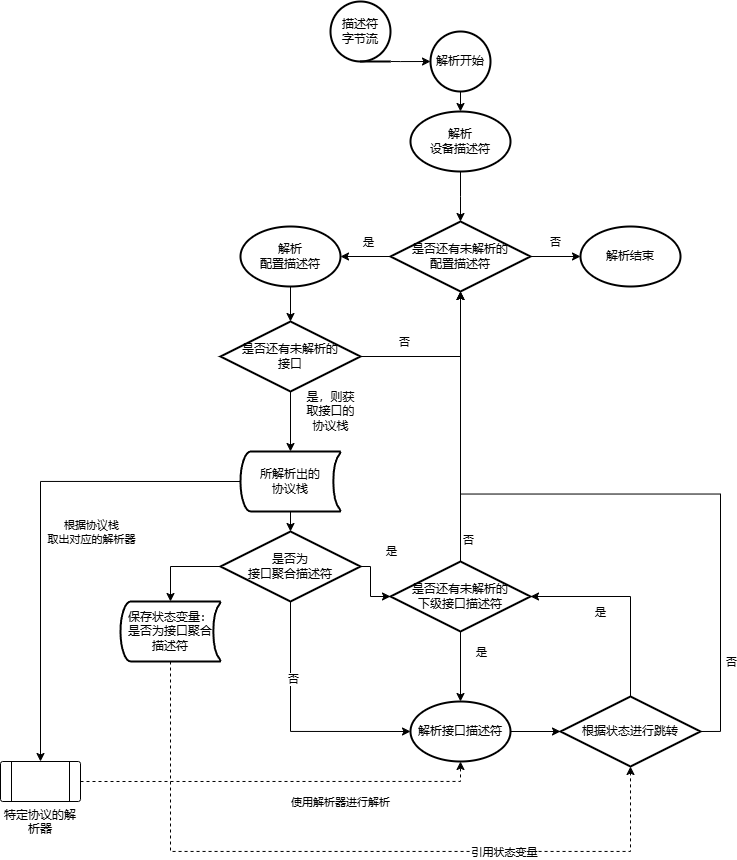


图 4.6 描述符解析系统的流程

## 本章小结

本章基于Rust语言实现驱动子系统核心模块，涵盖主机驱动层、USB驱动层、描述符解析系统与事件总线机制。通过模块化设计，系统被拆分为axusb-host与usb-descriptor-decoder两个程序库，前者负责框架主体，后者负责解析USB描述符树。主机驱动层通过抽象接口来实现不同版本规范主机的管理；USB驱动层采用热拔插模块化设计，依托Trait实现驱动实例的动态加载。

# 系统测试

## USB描述符解析器的软件测试

使用Rust内置测试框架对USB描述符解析器进行单元测试，部分代码如下：

#[cfg(test)]

mod test {

#[test]

fn test\_parse\_config() {

let input = [

9u8, 2, 34, 0, 1, 1, 6, 160, 50, 9, 4, 0, 0, 1, 3, 1, 2, 0, 9, 33, 1, 0, 0, 1, 34, 52,

0, 7, 5, 129, 3, 4, 0, 7, 0, 0,

];

let descriptor\_decoder = DescriptorDecoder::new();

let (result, \_) = descriptor\_decoder.parse\_config(&input[..]).unwrap();

let formatted = format!("{:?}", result);

assert\_eq!(formatted, "TopologyConfigDesc { desc: Configuration { length: 9, ty: 2, total\_length: 34, num\_interfaces: 1, config\_val: 1, config\_string: 6, attributes: 160, max\_power: 50 }, functions: [Interface([USBInterface { interface: Interface { len: 9, descriptor\_type: 4, interface\_number: 0, alternate\_setting: 0, num\_endpoints: 1, interface\_class: 3, interface\_subclass: 1, interface\_protocol: 2, interface: 0 }, endpoints: [Endpoint { len: 7, descriptor\_type: 5, endpoint\_address: 129, attributes: 3, max\_packet\_size: 4, interval: 7, ssc: None }], flag: \"hid\", extra: [Hid { len: 9, descriptor\_type: 33, hid\_bcd: 1, country\_code: 0, num\_descriptions: 1, report\_descriptor\_type: 34, report\_descriptor\_len: 52 }] }])] }")

}

}

其中，input是手动构造出的对应描述符字节流，其经过解析器解析后解析结果由assert\_eq!与右侧预期的解析结果进行比对，若assert\_eq!宏判定通过，则该方法不会panic，从而测试成功。否则panic，则测试失败。

测试结果如下图5.1所示。

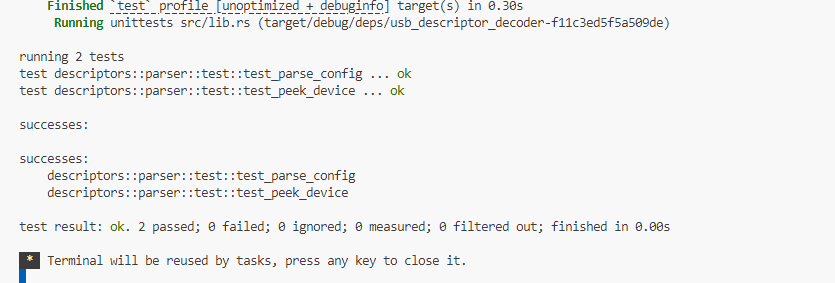


图 5.1 USB描述符解析器单元测试结果

测试结果均为ok（如test\_parse\_config通过）。解析速度在毫秒级以下（显示用时0.00s），性能高效。

## USB驱动子系统软硬件协同测试

USB驱动子系统的各模块完成对应的单元测试后，即可进行软硬件协同测试。软硬件协同测试所使用的硬件平台为飞腾派，其各参数如前文3.1.1章节中所示。测试环境如下：



图 5.2 软硬件协同测试

飞腾派使用U-Boot作为开机引导，通过串口与主机进行交互及下载操作系统内核文件，测试所用的操作系统为ArceOS,ArceOS是一个架构为Unikernel的操作系统，在其上进行驱动开发，可以在不更改所编写的代码的前提下进行多种环境下的测试（如裸机，开启虚拟内存，开启分页等环境状态下的测试）。

### USB驱动子系统的初始化流程

USB驱动子系统的初始化流程如图5.3所示:

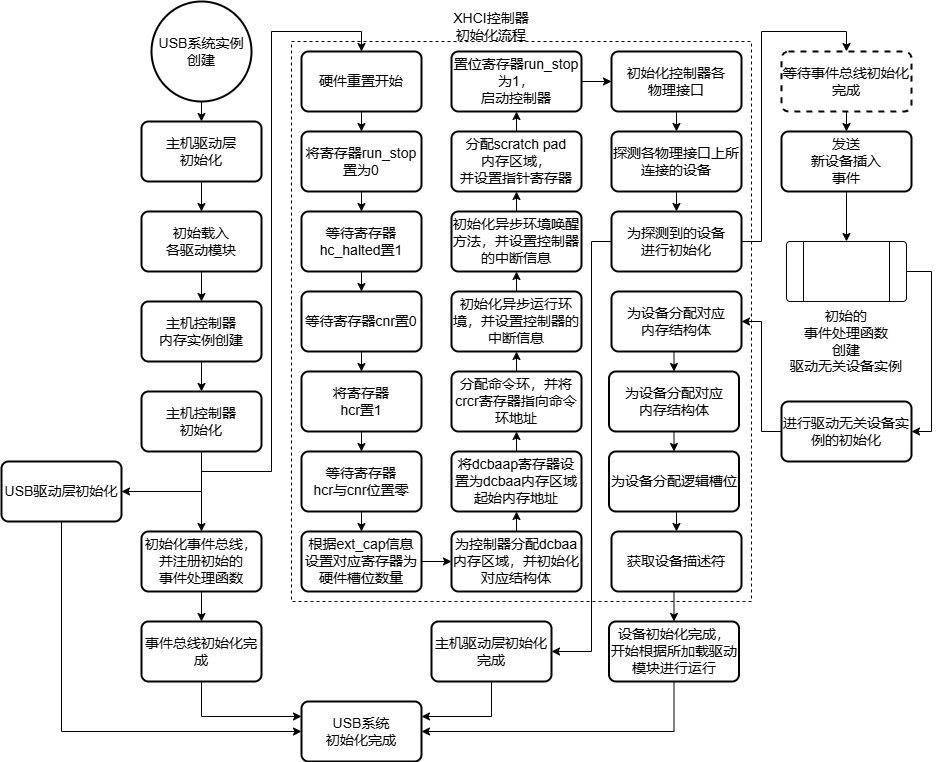


图 5.3 USB驱动子系统的初始化流程

其中，由于采用异步的编程范式，在主机控制器初始化后会同步的进行USB驱动层，事件总线及主机驱动层的初始化。由于飞腾派所使用的主机控制器为XHCI控制器，因此主机驱动层实际上是在运行XHCI控制器的初始化及工作流程。由于硬件工作的特殊性，不妨从是否成功获取到所插入设备的描述符来判断整个系统是否正常工作，进而判断该系统是否成功初始化，图5.4所展示的为串口中的部分输出结果。



图 5.4 所获取到的鼠标设备描述符树

### 基于驱动无关设备中各描述符所对应USB功能的驱动指派

在驱动无关设备被创建后，其经过初始化后将会获得所对应物理设备的描述符树。基于描述符树，驱动无关设备会向事件总线发送 NEW\_DEVICE\_PLUGIN事件以表示有新设备等待被分配驱动。当USB驱动层收到该事件后，将会取出事件上下文中驱动无关设备的引用，并根据其中不同的接口描述符所对应的USB功能进行驱动模块的指派，这部分代码如下：

pub fn new\_device\_initialized(&self, device: Arc<USBDevice<O, RING\_BUFFER\_SIZE>>) {

self.driver\_modules

.values()

.filter\_map(|module| {

module.should\_active(device.clone(), &self.config)

.map(|a| (a, module.name()))

},)

.for\_each(|(function, name)| {

//safety: feature holded ref would drop while module drop or device drop

let future = unsafe {

(\*(function.as\_ref()//取引用

as \*const RwLock<dyn

USBSystemDriverModuleInstanceFunctionalInterface<'a, O>,

>//引用转换为不可变指针

as \*mut RwLock<dyn

USBSystemDriverModuleInstanceFunctionalInterface<'a, O>,

>))//强制转换为可变指针

.get\_mut()//解引用，获取可变引用

.run()

};

let idx = embassy\_futures::block\_on(self.dynamic\_join\_array.add(future));

trace!("Settled driver instance future!");

embassy\_futures::block\_on(self.functional\_interfaces.write())

.entry(name)

.or\_insert(Vec::new())

.push((function, idx));

trace!("placed instance into array!");

});

info!("initialized new device!");

}

此处因为使用了裸指针来额外的获取可变引用，使用了unsafe代码块来跳出Rust本身的编译期生命周期检查，但这并不意味着这里的代码是不安全的，实际上，unsafe关键字的作用是告知编译器，此处所产生的引用由开发人员手动管理并释放，可以看到在//safety注释中，显式的声明了此处的安全性：feature中所包含的对于functional\_interfaces实例的引用，将会随着该实例或该模块的释放而一起手动释放，因此此处并未违反生命周期包含关系的原则。该部分代码的正确性随驱动模块的正确性一同测试。若驱动模块正常工作，则说明驱动实例所产生的future被正确的存入了调度队列中。

### HID驱动模块的测试

为测试设备（USB-HID鼠标）编写实验性HID驱动，验证USB子系统工作状态。HID驱动逻辑如图5.5所示。

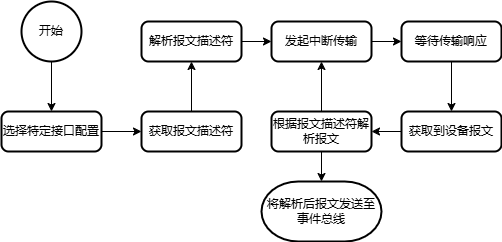
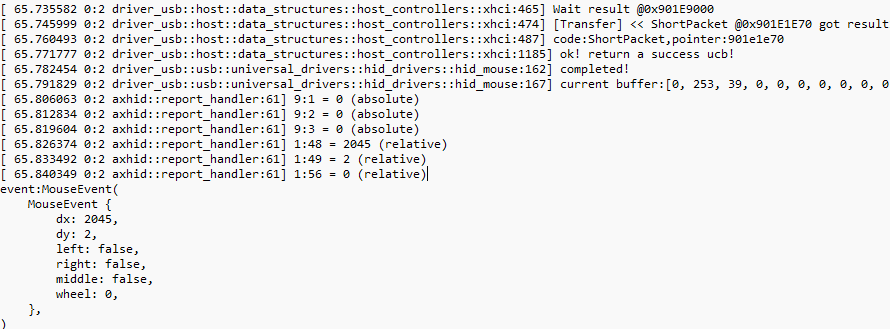


图 5.5 HID驱动的工作流程

对于HID驱动模块，测试方法为通过从事件总线读出设备报文来判断对应驱动是否正常工作，从事件总线读出的报文如下图5.6所示：

图 5.6 HID驱动的设备报文

可以看到，HID驱动成功获取到了设备的回报报文，并正确的解析了报文。图中，该鼠标事件描述了一次鼠标的平面滑动。该鼠标的x,y轴分别被施加了对应的移动速度。

### USB-CDC-CH34X驱动模块的测试

USB-CDC（Communications Device Class）协议用于通信设备，如串口、网卡、Modem等。然而，厂商私有功能导致不同CDC设备甚至同类型不同厂家的设备常需专用驱动。本节介绍为CH34X系列USB转串口模块开发的驱动。驱动模块逻辑如图5.7所示

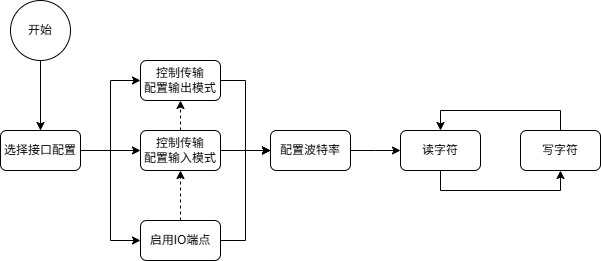


图 5.7 USB-CDC-CH34X驱动逻辑



图 5.8 CH341芯片实物图

对于CH34X驱动的测试，使用如图5.8所示的CH341芯片在飞腾派下进行串口收发实地测试。对于此类串口芯片，只需要将两侧TXD与RXD引脚对换连接，并正常接地即可，测试结果如下：



图 5.9 CH34X芯片的实地测试

在图5.9输出日志中可以发现，成功的接收到了串口的输出，内容为”hello,phytiumpi”。

在实际开发的过程中，由于CH34X系列芯片无官方开发文档，因此只能遵循Linux的代码进行对照调试。

## 实际应用：使用USB驱动控制小车运动

图5.10展示了应用实例：使用HID驱动控制小车的系统结构。

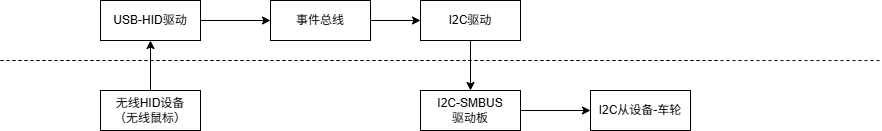


图 5.10 HID控制小车的结构图

图中，虚线上方是操作系统内核空间，虚线下方是外设硬件，USB子系统从外部的无线HID设备接受控制信号，并将控制信号发布到系统内核的事件总线上，I2C驱动接收到信号后通过I2C-SMBUS子协议，控制小车的四个车轮进行运动。小车运动前后如图5.11、图5.12所示：



图 5.11 小车运动前

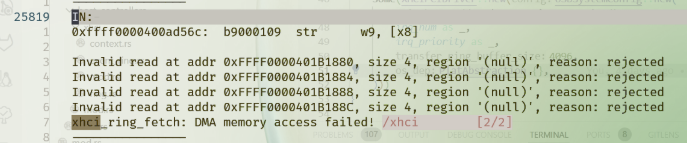


图 5.12 小车运动后

## 在进一步测试过程中所暴露出的问题与解决方案

为验证跨架构兼容性，使用QEMU（参数见3.1节）进行测试。测试发现USB驱动子系统存在虚拟内存适配问题：

现代系统普遍支持基于分页的虚拟内存。这需要内存管理单元(MMU)支持。DMA访问通常需要IOMMU支持虚拟地址寻址。对于现代的大多数计算机系统，甚至是嵌入式的系统而言，IOMMU都会存在。在当前背景下，这就使得向XHCI的特定几个指针寄存器写入内存地址时，可以直接写入虚拟地址。但是在QEMU仿真环境中，默认不使用此类MMU。因此在测试时，往往会发生驱动程序在特定步骤卡死的情况。通过对QEMU进行断点调试，并追踪到最底层的代码得知，DMA寻址在访问虚拟内存块时发生了失败的情况，如图5.13所示。

图 5.13 QEMU下DMA访问失败

这暴露出了目前的USB子系统尚未针对此类情况做适配。于是对此的修复方案为，在USB系统初始化时传入的平台配置中添加了针对是否使用了IOMMU相关功能的开关，并进一步的为平台抽象层添加了虚拟地址与物理地址之间相互转换的Trait方法，相关代码如下：

pub trait PlatformAbstractions: Clone + Send + Sync + Sized {

type VirtAddr:

From<Self::PhysAddr> + From<usize> + Into<usize> + Clone + Send + Sync;

type PhysAddr:

From<Self::VirtAddr> + From<usize> + Into<usize> + Clone + Send + Sync;

type DMA: Allocator + Send + Sync + Clone;

const PAGE\_SIZE: usize;

const RING\_BUFFER\_SIZE: usize;

const WORD: SystemWordWide;

fn dma\_alloc(&self) -> Self::DMA;

}

此处，为VirtAddr类型与PhysAddr类型添加了约束，要求彼此之间可以通过From Trait进行互相的转换。VirtAddr与PhysAddr是虚拟地址与物理地址的封装类型，其并未改变对应地址在内存上的布局（即内存地址以64位无符号整数表示），并为其附加了额外的编译期属性。用户可根据所用平台的实际情况，为虚拟与物理地址实现不同的转换方法。在主机驱动层中，所有对寄存器的内存读写都会使用PhysAddr这个物理地址包装类。修复后运行结果如图5.14所示，QEMU调试输出显示虚拟XHCI控制器正确处理了内存读写。

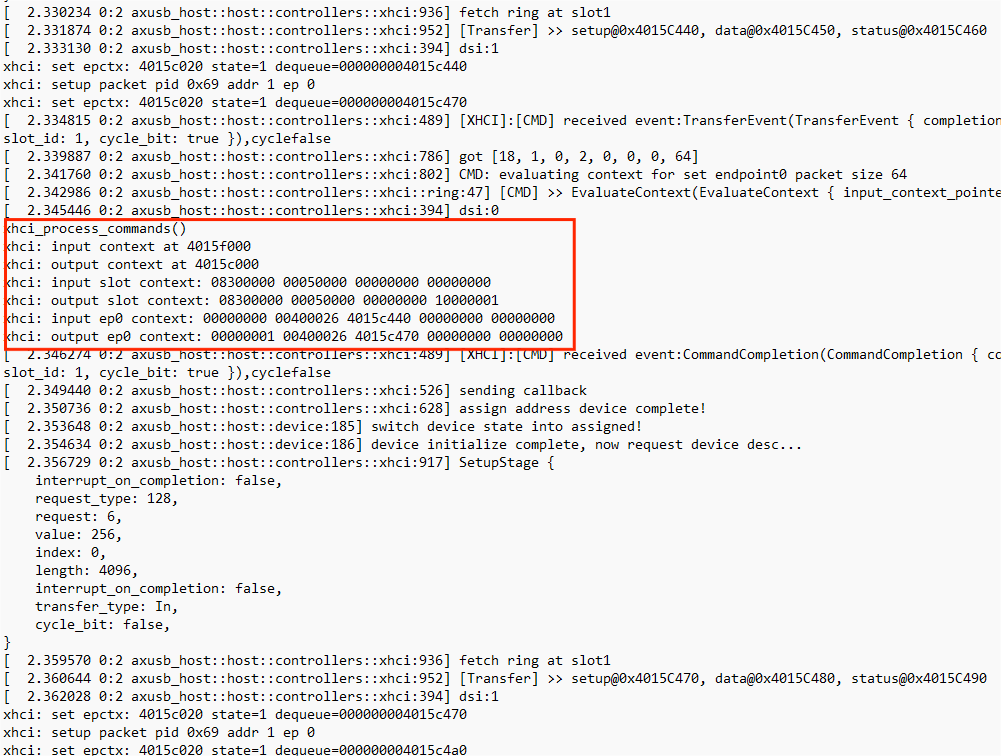


图 5.14 修复后的QEMU运行结果

## 本章小结

本章对整个USB主机端驱动子系统从硬件到软件，从内部的异步调度器到通用的USB描述符解析器，再到基于以上工作的应用层USB协议模块（USB-HID协议及USB-CDC协议）进行了全方位的测试，验证了系统的可用性，并且还展示了本系统在实际生产落地中的应用情况（使用USB-HID驱动控制小车运动）以及发现并排除了潜在的BUG。经过以上种种测试与调试，验证了所设计的系统符合“高可靠，易移植，模块化，异步化”的设计要求。

结 论

随着国产操作系统的快速发展与多样化硬件架构的广泛应用，USB驱动程序的跨平台兼容性已成为制约技术自主可控的关键问题。传统USB驱动开发高度依赖特定操作系统框架，导致迁移成本高昂且难以适应新兴硬件生态。本文设计并实现了基于Rust的跨平台USB主机驱动子系统，通过理论分析与实验验证，探索了通用、安全、高效的驱动开发新途径。

研究从USB主机控制器驱动层与功能驱动层解耦入手，构建分层抽象架构。通过引入驱动无关的设备实例与统一的主机控制器接口，实现了对不同版本USB主机控制器（如XHCI）的兼容支持。结合Rust异步模型，设计了基于Future的无栈协程调度机制，支持单线程下的任务主动让出与轮询切换。该机制降低了多线程资源竞争风险，并通过零成本抽象保障性能。此外，自主研发的USB描述符解析模块采用动态解析策略，有效应对了复杂拓扑结构与厂商自定义描述符的兼容性问题，为上层驱动的即插即用提供了可靠基础。

系统在ARM架构（飞腾派、QEMU仿真）与RISC-V架构（QEMU仿真）平台完成了功能与性能测试。测试结果表明，系统能够稳定识别并驱动多种USB设备（如HID设备、存储设备），并通过事件总线机制实现与外部组件的信息交互。通过协议分析仪验证了USB传输的完整性与低延迟特性，而异步任务调度模型在单线程环境下展现出与多线程相当的吞吐效率。

本文的主要贡献在于：其一，提出了一种面向跨平台场景的USB驱动框架设计范式，通过分层抽象与异步模型解决了操作系统异构性带来的兼容难题；其二，首次将Rust语言系统级安全特性深度融入USB驱动开发，利用所有权机制与借用检查器规避了内存泄漏与数据竞争风险；其三，设计的事件总线与热拔插驱动管理系统，为国产操作系统的外设生态建设提供了可扩展的技术支撑。

未来工作包括：一是扩展对USB4与Thunderbolt协议的支持，适应更高带宽设备的接入需求；二是探索Rust与现有C/C++驱动生态的互操作性，降低迁移成本；三是结合形式化验证工具，对关键模块（如异步调度器）进行数学证明，提升系统可靠性。本研究为构建自主可控的跨平台驱动生态提供了理论参考与实践基础，对推动国产操作系统技术演进具有重要意义。

参 考 文 献

[1] Peng H, Payer M. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation[C]//29th USENIX Security Symposium (USENIX Security 20). Berkeley, CA: USENIX Association, 2020: 2559-2575.

[2] J. Huang, Y. Wang, Research and Application of High-Speed and Adjustable Synchronous Data Transfer System Based on USB 3.0 Peripheral Controller[J]. Journal of Circuits, Systems and Computers, 2020, 30. DOI:10.1142/S0218126621501188.

[3] Wong G W .Can Rust Work for Safety-Critical Embedded Systems?[J].Electronic Design,2025,

[4] 韩乃平,李蕾.国产操作系统生态体系建设现状分析[J].信息安全研究,2020,6(10):887-891.

[5] 杨志荣,胡茂海,董理,李守业.FPGA的xHCI协议软核IP设计及应用[J].单片机与嵌入式系统应用,2022,22(12):7-11.

[6] C. Wang, Analyzing the limitations of parallelism in hardware and software through threaded programming[J]. Highlights in Science, Engineering and Technology, 2023, 41: 23-28. DOI:10.54097/hset.v41i.6738.

[7] 张帅林. 基于嵌入式处理器的高性能中间件软件设计与实现[D].北京理工大学,2021.DOI:10.26948/d.cnki.gbjlu.2021.001581.

[8] 刘巍,黄智勇.国产化实时通信中间件DDS的跨平台实现和优化技术[J].现代雷达,2021,41(07):39-43.DOI:10.16592/j.cnki.1004-7859.2021.07.007.

[9] 王硕,胡飞. 基于数据发布订阅服务的嵌入式通信中间件设计[C]//中国航空学会.第九届中国航空学会青年科技论坛论文集.中航出版传媒有限责任公司（China Aviation Publishing & Media CO.,2020:8.DOI:10.26914/c.cnkihy.2020.052267.

[10] 李荣.2022年物联网应用趋势[J].计算机与网络,2021,47(21):42-44.

[11] 张卓若,常瑞,杨申毅,等.面向Rust语言的形式化验证方法研究综述[J/OL].软件学报,2025,(08):3604-3636[2025-06-06].https://doi.org/10.13328/j.cnki.jos.007353.

[12] 胡霜,华保健,欧阳婉容,等.Rust语言安全研究综述[J].信息安全学报,2023,8(06):64-83.DOI:10.19363/J.cnki.cn10-1380/tn.2023.11.06.

[13] Hong J ,Shim S ,Park S , et al.Taming shared mutable states of operating systems in Rust[J].Science of Computer Programming,2024,238103152-.

致 谢

感谢丁菊老师，丁菊老师作为我的毕业设计指导老师，在我设计并完善该系统的过程中提供了很多的帮助。并为我的论文的撰写提供了非常多宝贵的建议与指导。

感想rCore-OS社区的诸多同志，USB驱动子系统最初编写的目的是作为ArceOS的USB驱动子系统模块而存在的，在我编写该模块的过程中，我逐渐意识到了该模块作为跨平台/跨操作系统的通用库存在的可行性与必要性，于是便有了如今的成果。在我设计并完善该库的过程中，为我提供了很多的帮助。

感谢我在项目中所使用的第三方库的诸多作者。他/她们为许多先进的理论与设计模式提供了良好且安全高效的封装，使得我可以以更安全高效的方式来进行该系统的编写。

感谢在我开发该系统的过程中提出了诸多宝贵建议，批评，指导及其他各种形式帮助的各位老师，同学！