摘要

Linux 设备驱动一般运行在内核空间，近些年来，为了提高驱动开发效率，一些研究人员提出了在用户空间下开发驱动的概念，相对于内核驱动，用户空间下的驱动有着以下优点：1，调试的方便； 2，用户空驱动出错不会引起系统崩溃；3，用户空间驱动可以使用 C 语言库，简化驱动的开发；4，用户空间驱动对内核依赖性较少，有更好的移植性，并且可以发布封闭源代码的驱动。

应用程序可以通过系统调用和内核驱动进行交互，但对于用户空间驱动，需要实现相应的框架来支持应用程序与驱动之间的交互。本文针对视频设备的特性，设计并实现了用户空间的 V4L2 交互框架，通过该框架，应用程序可以方便的以 V4L2 标准与用户空间驱动进行交互，并能减少用户空间驱动的开发工作量。

视频设备有较复杂的工作机制和数据格式，在用户空间下开发，能够有效的提高驱动开发效率和降低调试的难度。本文从 USB 视频设备的共性出发，利用 Libusb 库设计并实现了 USB 视频设备的用户空间驱动，并使用 C++语言以面向对象的思想来组织驱动的开发，并实现了视频驱动中可重用的传输缓冲区模块，同时 C++相对与 C 语言更强的语言安全性也提高了驱动的稳定性。

关键词： Linux 驱动,视频设备,用户空间,V4L2 交互框架，传输缓冲区

Abstract

Linux device drivers usually run in kernel space. Recent years, in order to improve the driver development efficiency, some researchers propose the user space driver development concept. Compare to kernel space driver, user-space driver has the following advantages: 1, convenience of debugging; 2, the user space driver error will not cause system crashes; 3, the user space drivers can use the C language library, simplify driver development; 4, user-space drivers are less dependent on the kernel, so have better portability, and can release closed source drivers.

The application uses the system calls to communicate with kernel driver, while in user space driver, there needs an appropriate framework to facilitate the interaction between application and driver. In this paper, considering the characteristics of video equipment, it designs and implements the user space V4L2 interaction framework. Using this framework, the application can communicate with user-space drivers under V4L2 standard easily, and also reduces the workload of the user space driver development.

Video device has complex working mechanisms and data formats, develop its driver in user space and improve the develop efficiency and reduce debugging difficulty. This article uses Libusb library to design and implement a user space USB video device driver, and uses the C + +, with object-oriented idea to the organize driver development, and realizes a reused transmission buffer module. C++' s more safety in language can increase the driver stability.

Keywords: Linux driver, video device, user space, VL42 interaction framework, transfer buffer

图目录

表目录

致谢

转眼间，两年半的研究生生涯快要结束了，在这里要感谢所有帮助我的老师和同学。

首先，我要感谢我的导师——陈奇老师，陈老师严谨的治学态度和平易近人的为人处世给我留下了深刻印象，也将使我终身受益。陈老师在科研上对我不断的指导和鞭策是我不断成长的动力，同时陈老师在生活和学业上给了我各方面的关怀，使我能够顺利的完成毕业设计。

其次，我要感谢何江峰师兄，无论遇到科研还是生活上的问题，他都能乐于帮助并，并耐心解决。

接着我要感谢王志鹏、宋国兵同学正是与他们一起合作，才顺利的完成了实验室与华为预研部和合作的项目。同时也要感谢夏杰、程春惠、车延辙、钱丰等同学，正是这些同学组成了实验室一个团结的集体。

最后我要感谢我的父母亲及其他关爱的人，我的成长和进步里不开你们的支持。

徐家

2010 年 1 月于求是园

第1章 绪论

1.1 课题研究背景

设备驱动是硬件与应用程序之间的桥梁，也是Linux系统中不可缺少的一部分。从Linux 1.0到2.6版本的发展过程中，Linux内核对硬件设备的支持越来越全面，同时随着硬件设备种类、型号的不断增加，Linux内核中的驱动程序的代码总量也在不断增加，已占到内核代码的70%左右[1]。研究表明设备驱动问题是影响操作系统稳定性的主要因素之一[23]，而Linux内核中驱动代码中的错误是其他内核代码的7倍左右[24]。随着计算机硬件性能的不断提高，设备驱动效率不再成为制约硬件设备性能表现的瓶颈，而驱动的稳定性的越来越受到重视。同时，硬件设备的更新换代和新设备出现的速度也越来越快，对于硬件驱动的开发时间周期也要求越来越高[21]。传统的Linux设备驱动一般由C语言开发，而C语言缺乏类型安全性是影响Linux设备驱动稳定性的重要因素之一。相对于应用程序，Linux 设备驱动在开发方法和工具上所取得进展也相当有限，缺乏相关的开发SDK和开发工具，驱动开发往往需要开发人员精通内核底层的数据结构，不仅有着较高的门槛，同时也不利于缩短驱动开发周期。

1.2 国内外研究现状分析

针对Linux设备驱动的稳定性和开发效率问题，国内外的一些学者进行了相关的研究，研究领域主要集中在以下几个方向：1，在Linux内核中引入JAVA等类型安全语言进行驱动开发；2，通过高层的建模语言对驱动进行开发；3 用户空间驱动开发的相关技术研究。

Shan Chen, Lingling Zhou[23]提出了在Linux内核中使用java语言进行驱动开发的方法。Shan Chen, Lingling Zhou在实现内核空间Java虚拟机基础上，给出了基于内核Java虚拟机的驱动开发方法，在驱动中引入Java虚拟机能够从编译、运行时提高驱动的稳定性，当然在驱动执行效率上可能会有所降低。

Fabrice Mérillon, Laurent Réveillère[21]提出了使用接口定义语言 Devil进行驱动开发的方法。Devil语言从I/O端口、寄存器和设备变量三个层次来对驱动进行抽象，通过Devil语言能够清晰明确的表示驱动涉及到的端口、寄存器

和设备变量，以及三者之间的相互联系 。Devil 程序编译后生成的相应 C 语言宏，驱动程序需要操作硬件时直接调用这些C语言宏，这样既能够避免直接编写对硬件操作的代码而带来的错误，并能够使开发人员在一个更高的语言层次开发驱动，提高驱动开发的效率。

Christopher L. Conway, Stephen A. Edwards[25]提出了通过 Domain-Specific Language-NDL 语言开发驱动的方法，NDL 借鉴了 Devil 语言的语法结构。NDL 驱动的描述由寄存器集合、访问寄存器的协议和一系列设备函数构成。NDL 语言通过对驱动中常用的写寄存器、I/O 数据拷贝命令进行了抽象和封装，使得需要多行 C 语言代码实现的操作在 NDL 中可以简洁的被表示，同时 NDL 将设备相关的系统调用或执行代码封装为库函数，方便开发人员的使用。

目前大多数设备驱动开发方法的研究都在 Linux 内核空间中展开，但 Linux 内核空间编程相对与用户空间编程来说限制较多，使用一些的新的方法或非传统的手段往往需要对内核进行重新改造，工程量较大。用户空间驱动作为一个新兴的概念，则有着如下优点：1，驱动调试的方便，用户用 GDB 等常用工具即可进行驱动的调试；2，驱动开发的便利，在用户空间开发驱动时，可以使用 C 语言库，并使用第三方的开发库，提高驱动的开发效率；3，用户空间驱动挂起了，可以被简单杀死并重启动，并不会影响整个系统的运行[3]；4，有学者提出在内核使用 C++面向对象的思想来组织驱动，提高驱动的重用性[4]，在用户空间，由于不受内核对 C++语言的限制，可以用更方便的通过面向对象的思想来实现驱动程序，提高同类型驱动代码的重用性；5，在驱动内部可以实现一些复杂的功能，如音视频数据的压缩等。因此在用户空间中，开发驱动程序不仅有着较大灵活性，并且能够提高驱动程序的健壮性和开发效率。

1.3 用户空间驱动分析

Linux 地址空间可以分为内核空间和用户空间，应用程序主要运行在用户空间，而系统服务、驱动程序等一般运行在内核空间，运行在内核空间的程序优先级较高，并能使用特权级别的操作，有较高的执行效率，但内核空间程序的问题可能会导致系统崩溃。内核空间中只支持 C 语言编程，无法直接使用较高层的语言如 C++、Java 等进行开发，在内核空间中无法使用 C 语言库或其他开发库，一般只能使用内核数据结构和系统函数，同时也不支持浮点数操作，内核空间程序

的这些限制直接了制约了驱动程序的开发速度，同时不利于提高驱动的稳定性。

User Space I/O System，简称UIO，是作为Linux内核中对用户空间驱动开发提供支持的一个框架在Linux2.6.23版本中被加正式入内核。UIO并不是一个广义上的驱动框架，在Linux内核中被支持的很好的驱动子系统，如串口和USB等子系统，并不在UIO支持的范围之列。UIO支持的设备需要满足如下特性：1，设备内存可以被映射，并且通过写设备I/O内存可以直接操控硬件；2，设备通常能够产生中断信号；3，设备并不在标准的Linux设备子系统支持之列[5]。UIO可以将驱动大部分的工作都放在用户空间，但在内核空间还是需要编写小部分代码，这部分代码主要是工作是将驱动连接到总线，并注册中断处理程序，使得用户空间驱动能接收到硬件中断。用户空间的驱动进行注册后，会生成/dev/uio0的设备文件，应用程序能够以的open、ioctl等常规指令对该设备文件进行操作。据估计，在内核空间实现68个标准的ioctl指令的驱动程序需要5000行左右的代码，但在用户空间实现同样功能的驱动程序，只需要3000行用户空间代码和156行内核空间代码。但UIO作为一个刚刚发展起来的框架，功能还比较有限，目前只支持字符型设备驱动，而不支持块设备、网络设备等驱动，对其他Linux设备子系统支持也不是很好。

Libusb是支持用户空间下访问USB设备的函数库。Linux下最初只能在内核空间中访问USB设备，但Linux内核开发人员看到了在用户空间下访问USB设备的必要性，因此在Linux中引入了usbdevfs文件系统。usbdevfs和proc一样，并不是一个实际的文件系统，而是在内存中动态生成的虚拟文件系统，通过usbdevfs及其提供的函数接口，在用户空间能够直接访问USB设备，也使得在用户空间开发USB设备驱动成为可能。为了避免与原有的devfs文件系统产生混淆，usbdevfs在Linux2.6内核中被更名为usbfs。usbfs加载在/proc/bus/usb目录下，该目录下的每一个设备文件对应了一个实际USB设备，usbfs定义了相关的函数和数据结构，通过引用相关头文件，便可以在用户空间程序中访问usb设备。Libusb实际上是通过usbfs来访问USB设备[6]，不过对相关usbfs函数接口进行了封装，是一个更高层的函数库。Libusb有良好的跨平台特性，可以运行在Linux、FreeBSD、NetBSD、MacOs x、Windows等平台，因此基于Libusb开发的USB视频设备驱动、USB存储设备驱动或其他USB外设驱动也有良好的移植性。

VGALIB是Linux下一个底层的图形库，功能类似与X Windows，应用程序通过VGALIB提供的API，可以直接操作VGA的端口。VGALIB为进程分配了和视频内存区域同样大小的内存，并支持进程能够直接访问视频内存来控制屏幕显示[7]。

Linux用户空间驱动作为一个起步不是很久的事物，也存在着一些不足，虽然一些现有的技术能够使得在用户空间直接访问硬件设备成为可能，但相对于内核空间驱动，主要还存在以下不足之处：1，用户空间驱动还未形成类似内核驱动一样形成完善的框架，包括应用程序如何调用空间驱动，应用程序和驱动之间的消息和数据交互的机制。2，用户空间驱动与应用程序的藕合性问题，现有的交互方式下，用户空间驱动与应用程序往往会高度藕合，不符合软件设计中高内聚低藕合的要求[9]；3，用户空间驱动的效率问题，相对于内核驱动，用户空间驱动在效率上可能会有所降低。4，一些内核驱动中的标准协议，如V4L2等，需要在用户空间驱动中实现。

1.4 本文的工作与创新

本文分析了用户空间与内核空间中驱动开发在实现机制和具体细节上的差异，针对用户空间驱动现有的不足，设计并实现了V4L2交互框架，该交互框架有良好的重用性和扩展性，方便应用程序和视频设备用户空间驱动之间的交互，同时能够对用户空间下视频设备驱动开发提供有效的支持，降低驱动开发的工作量。并利用Libusb库提出并实现了USB视频设备用户空间驱动开发方法，该方法能够有效提高USB视频设备驱动开发的效率。并在用户空间驱动开发中使用C++语言面向对象的思想来组织驱动的开发与设计，实现了可重用的传输缓冲区模块，并使驱动的代码有着良好的维护性和扩展性，并提高了驱动程序的健壮性。

本文对以下几方面内容进行了研究：

- 分析对比了Linux内核驱动和用户空间驱动在程序特性、框架支持、调试方法、可移植性方面的差别，并指出了用户空间驱动需要改进的地方。
- 对于用户空间驱动与内核空间驱动在调用接口上的不同，提出并实现了V4L2交互框架，该框架有效的屏蔽和解决了调用接口的问题，并提供了应用程序与用户空间视频设备驱动之间的控制传递，数据传输的相应机制，并实现了框架的标准调用接口，通过该框架，可以减少用户空间驱

动开发的复杂度。

- 分析了USB视频设备的相关协议与标准，讨论了对USB视频设备进行控制的基本原理和方法。
- 分析了用户空间下对USB设备进行读写访问的Libusb库，并在此基础上提出了USB视频设备驱动在用户空间的实现的方法，针对用户空间驱动与内核空间驱动的不同，分析并给出了在用户空间实现USB视频设备驱动的实现方法，并重点讨论了驱动传输缓冲区的实现机制与同步策略。
- 对用户空间驱动和内核驱动进行了相应的测试，分析对比了两者在驱动性能，传输效率、CPU占用率等方面的差别。

1.5 本文内容组织结构

第2章对Linux用户空间驱动和用户空间驱动在程序特性、框架支持、调试方法和移植性方面做了对比，分析了两种形式的驱动各自的优缺点，并指出了用户空间驱动需要改进的方面。

第3章对Linux下的视频标准V4L2做了分析和阐述，并针对用户空间视频驱动，提出了针对支持用户空间驱动的V4L2交互框架，并阐述了该框架的工作机制和实现原理与方法，以及提供的相应的接口。

第4章分析了Linux下访问USB设备的开源库Libusb的原理和使用方法，以及使用Libusb进行驱动开发的优点，并提出了在用户空间下通过Libusb库开发USB视频设备驱动的方案，并给出了驱动的详细设计。

第5章 分析了内核驱动和用户空间驱动的实现在具体技术细节上区别，结合C++面向的对象思想，给出了USB视频设备用户空间驱动的实现，并重点分析了驱动传输缓冲区的并发访问以及加锁机制，并实现了一个可重用的驱动传输缓冲区模块。

第6章 针对USB摄像头，通过对Luvcview播放器的改造，对USB视频设备用户空间驱动进行了测试，并和内核空间驱动在性能进行了比较。

第7章 对本文的工作进行了总结，以及对需要改进的地方进行了展望。

第2章 用户空间驱动和内核驱动比较

2.1 程序特性

用户空间驱动和内核驱动在程序特性有比较大的差别，用户空间驱动属于用户空间编程，而内核驱动属于内核编程的范畴，两者在使用到的头文件、函数、数据结构等都有比较大的差别，程序的特权级别与优先级也不同，下面阐述了两者在程序基本特性上的差别。

2.1.1 内核空间驱动程序特性

Linux 的内核结构继承了 Unix 的单内核结构，整个内核作为一个大的单的进程运行。运行在内核的驱动程序可以分为两大类，一种是被直接编译进内核，随着内核的启动而被加载；另一类是作为模块而被动态加载，两种驱动虽然加载方式不同，但一旦运行起来，内核便无差别对待它们。作为模块加载的驱动由 module_init 函数作为加载入口，由 module_exit 函数作为卸载的出口。内核驱动虽然是由 C 语言开发，但在内核驱动中，无法使用 C 语言库，如 printf、malloc 等函数，而必须使用相应的系统调用或内核函数来替代，如 printk、kmalloc 等。从效率上来说，系统调用优于 C 语言库，因为用户空间 C 语言库函数执行时一般展开为若干个系统调用。内核驱动或是内核线程的栈空间一般只有 8K，该 8K 栈空间还会被线程引起的中断程序共享使用，因此内核程序能使用的栈空间是相当有限的，所以在内核驱动中需要避免使用大的结构体或递归操作，否则容易导致驱动程序出错。内核驱动在编译时会依赖于 Linux 内核的源代码，运行时则依赖于内核的其他模块，而 Linux 各个版本之间都会发生源代码的变化，内核驱动代码也需要进行修改才能适应不同版本的源代码，因此，内核驱动在可移植性上一般都不是很好。并且运行在内核空间的程序往往需要应付更多的并发操作，这些并发操作可能来源多个进程对驱动的访问或是中断的异步运行等[3]

2.1.2 用户空间驱动程序特性

用户空间驱动属于用户空间程序的范畴，基本特性与其他的用户空间进程类似。程序的优先级一般默认为 20，并且无法进行修改。而在内核中，线程不但

能够设置优先级值，甚至将自己设为实时。同时用户空间驱动无法使用一些内核中才有的特权操作，如关闭系统中断等，这也避免了用户空间驱动产生致命的错误。从理论上讲，用户空间驱动的在执行效率上一般没有内核驱动高，尽管如此，现在还是有一些用户空间驱动被广泛使用，如 Linux 下的 X 服务器，随着 CPU 运行速度和内存性能的不断提高，效率已经不是一个瓶颈，用户对驱动程序稳定性和开发者对驱动程序研发速度和质量的需求会成为开发驱动时考虑的一个重要因素[4]，当然开发用户空间驱动时，也需要借鉴内核中一些高效的程序架构。对于一些要求实时性高，频繁需要中断响应的驱动，应该还是放在内核中运行，但对于许多设备，在用户空间驱动并不会影响其实际性能，如 USB 视频设备、显示设备等。由于用户空间驱动不会直接依赖与内核代码和内核模块，相对于内核模块，有着较好的移植性，驱动开发人员可以直接发布可执行程序，而不必发布源代码。

2.2 框架支持

对于 Linux 内核驱动，Linux 的设备模型和系统调用可以减少驱动开发者的实际工作，当应用程序使用系统调用来向硬件发送控制命令时，Linux 内核提供的消息转发机制会自动将控制命令发送到对应的内核模块，并通过系统函数，完成应用程序和驱动程序的之间的数据拷贝，而设备模型可以自动为驱动处理一些事务，如热插拔、引用计数等。应用程序与内核驱动的调用关系如图 2.1 。

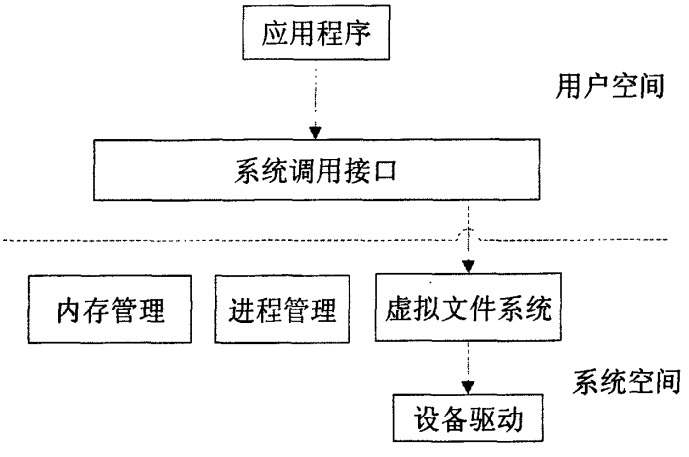


图 2.1 应用程序与内核驱动交互方式

用户空间驱动作为一个起步不久的事物，框架层面的支持比较少，开发人员可能要自行处理应用程序和驱动之间的消息命令的发送，和一些驱动与应用程序之间标准的实现，不过相应的框架一旦实现，就可以被同类设备驱动重用，简化驱动的开发。应用程序与用户空间驱动的关系如图 2.2 。

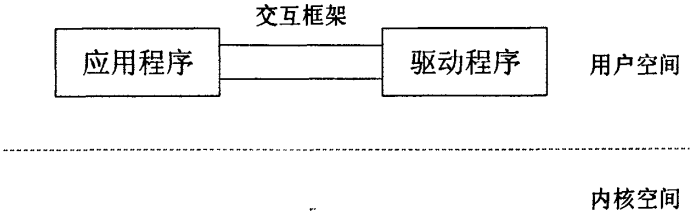


图 2.2 应用程序与用户空间驱动交互方式

2.2.1 Linux 设备模型

Linux 设备模型直接支持着内核驱动的开发，Linux 设备在内核中都有一个共同的祖先 kobject。 kobject 作为一个最原始和底层的内核对象，经过层层嵌套后被实际设备对象引用。Linux 中一个硬件不论其实际功能是什么，在 Linux 设备模型中，一般需要满足以下几个功能：引用计数、在 sys 目录中被表示、热插拔事件处理等 。 在编写驱动程序时，开发人员不需要去处理这些细节，正是 kobject 完成了这些事情。kobject 结构中有几个比较重要的数据结构，分别是：1, struct kref kerf 表示该 kobject 对象的引用计数，当该 kobject 被创建时，引用计数为 1，通过 get 和 put 操作可以增加或减少该 kobject 的引用计数，当引用计数为 0 时，该 kobject 被释放；2，struct kobject * parent，指向其父 kobject 的指针，父 kobject 表示拥有该 kobject 设备的上一层次设备，对于一个 usb 设备来说，其父 kobject 表示的设备可能是一个 hub, parent 指针的主要作用是在 sysfs 目录中层级结构的表示；3, struct kset * kset,包含该 kobject 对象的 kset 指针。[8]一个 kobject 对象由 kobject_init 函数创建，不过单纯创建一个 kobject 对象是没有实际意义的，一般需要设置其从属的 kset，并通过 kobject_add 函数来设置其父 kobject,这样通过层层关联,一个 kobject 才能和实际设备结合发挥其作用。

Kset 中包含了一组 kobject 对象的集合，kset 类似于一个容器类对象，其中包含相关联的 kobject 对象集合。Kset 中所有的 kobject 被组织成一个双向链表,被包含的 kobject 可能被内嵌于其他的对象中。Kset 的初始化等函数也类似于 kobject,可以遍历 kset 来枚举所有的关联设备，如所有的 PCI 设备等。

Subsystem 描述了系统中某一类子系统，如块设备子系统，一个设备子系统结构中包含了一个 kset 对象和 rw_semaphore 对象，每一个 kset 必须属于某个 subsystem，而 rw_semaphore 信号量则提供了对 subsystem 中 kset 对象的互斥访问。由于 subsystem 只是简单的对 kset 的封装，在 Linux2.6.21 版本之后，subsystem 已被 kset 所取代。

一个设备在驱动程序中可能以 usb_device、pci_dev 等具体设备的结构来表示，但在底层，一个设备通常以 struct device 结构来表示 。struct device 中比较重要的数据结构有：1，struct device *parent ，表示该设备依附的父设备；2，struct kobject kobj，代表内嵌于该设备的 kobject 对象；3，char bus_id 和 struct bus_type *bus 用来表示挂载该设备的总线；struct device_driver *driver 表示该设备的驱动。struct device 结构一般被内嵌更在高层的设备结构中,如 USB 设备结构 usb_device,视频设备结构 video_device 等。

2.3 驱动调试方法

在驱动调试方法上，内核驱动和用户空间驱动的差异比较大，用户空间驱动的调试可以用常见的 GDB 调试，也可以在其他 C/C++ IDE 中调试。

但内核驱动的调试复杂的多，一般有以下几种方法： printk 函数，printk 将内核信息打印到内核信息缓冲区，在用户空间可以通过 dmesg 命令来查看驱动打印信息，但该方法调试效率偏低，并且打印数据过多的话，后打印的信息会覆盖较早打印的信。

kdb，kdb 只能对内核驱动进行汇编语言级的调试，对于调试复杂的驱动来说调试过程不是很直观。

kcore，kcore 配合 gdb 可以将整个内核当作一个应用程序来调试，不过在这种方式下，无法设置断点调试或单步执行，调试功能比较有限。

kgdb，kgdb 可以对内核驱动进行源代码级的调试，kgdb 需要两台机器进行调试，一台作为目标机运行被调试的内核，另一台作为主机通过串口连接到目标

机进行调试[31]，对于可加载模块的调试，需要先捕获模块的 symbol 信息，但调试过程中往往无法捕获实际运行过程中由于一些异步事件引发的错误，kgdb 的调试方法是所有内核调试方法中功能最强大的，但使用起来也是最复杂的。

2.4 驱动可移植性

影响内核驱动移植性的主要因素为内核驱动对 Linux 源代码和内核版本的依赖性，和不同平台在基本数据结构上的差异，Linux2.4 到 Linux 2.6 在源代码有了很多的变化，甚至在内核模块的编译方式都有差别，Linux 2.6 的各个子版本之间的源代码也又差异，内核驱动在不同版本的 Linux 内核之间移植可能会遇到需要修改驱动源代码的情况。

用户空间驱动的开发不会直接依赖于内核的源代码，更多依赖的是 Linux 内核向用户空间提供的系统调用和函数调用，相对于源代码来说，这些系统调用和函数调用接口比较固定，即使在函数实现上发生改变，但在调用接口、参数类型上一般是不会变化的，因此用户空间驱动的移植相对比较方便。

2.5 本章小结

本章重点比较了 Linux 内核驱动和用户空间驱动在程序特性、框架支持、调试方法和移植性方面的差别，同时也分析了内核驱动和用户空间驱动各自的优缺点。

第3章 V4L2交互框架的设计与实现

3.1 Linux视频V4L2标准

Video for Linux 2，简称为V4L2，是Linux下的视频驱动的标准框架，在前身V4L的基础上发展起来。V4L2支持的设备几乎涵盖了Linux下所有的视频设备，包括摄像头、数字电视设备、模拟电视设备、视频输出设备等等。作为一个标准框架，V4L2有两层含义，首先对于应用程序，应用程序可以按照V4L2定义的操作命令，进行打开视频设备、获取视频设备信息、传输视频数据、设置视频属性等等操作；对于驱动，必须相应的实现V4L2定义的一系列回调函数，与应用程序的调用相对应，当然并不是V4L2定义的所有回调函数驱动都要实现，驱动可以根据设备的特性，实现V4L2定义操作的一个子集，但和视频数据传输相关的操作一般是必须实现的。

V4L2中应用程序的工作流程如下：首先通过open系统调用打开视频设备，接着通过ioctl系统调用发送V4L2命令来查询视频设备基本信息，并初始化视频设备参数。主要的查询设备信息及初始化设备的V4L2命令如表3.1[14]。

表3.1 查询以及初始化设备V4L2命令

V4L2命令	命令作用
VIDIOC_QUERYCAP	查询视频设备功能
VIDIOC_S_FMT	设置视频帧格式属性（高度、宽度、像素格式）
VIDIOC_S_PARA	设置视频传输参数(视频帧传输间隔)

当应用程序成功执行上述查询和初始化命令后，可以开始视频数据的传输了。V4L2中，视频数据的传输一般不通过read、write系统调用来进行，而是采取内存映射的流传输方式，流传输方式不要实际数据的拷贝，驱动和应用程序之间只需要交换被映射内存的指针。具体做法如下：应用程序首先通过ioctl系统调用发送VIDIOC_REQBUFS命令来申请传输缓冲区，申请到传输缓冲区位于驱动程序内，应用程序接着通过调用ioctl发送VIDIOC_QUERYBUF命令来查询申请到的传输缓冲区的具体信息，并通过mmap系统调用将传输缓冲区映射到应用程序的

地址空间，驱动和应用程序之间的数据传输，只需通过读写被映射的传输缓冲区。然后应用程序发送VIDIOC_STREAMON命令，通知驱动开始数据传输。应用程序获取一个视频帧，需通过发送VIDIOC_QBUF命令将一个v4l2_buffer放入传输缓冲区进行传输，然后通过发送VIDIOC_DQBUF命令获得完成传输的v4l2_buffer。应用程序通过不断的发送VIDIOC_QBUF和VIDIOC_DQBU命令，便能持续的从视频设备获得视频数据。最后，当应用程序退出时，通过发送VIDIOC_STREAMOFF命令来结束传输。视频数据传输相关的V4L2命令如表3.2。

表 3.2 数据传输V4L2命令

V4L2命令	命令作用
VIDIOC_STREAMON	开启视频数据的流传输
VIDIOC_STREAMOFF	关闭视频数据的流传输
VIDIOC_REQBUFS	申请传输缓冲区
VIDIOC_QUERYBUFS	查询传输缓冲区
VIDIOC_QBUF	出 将空的视频帧放入传输缓冲区
VIDIOC_DQBUF	将填充过的的视频帧从传输缓冲区取出

当视频数据传输开始后，应用程序还可以对视频的显示属性进行进一步的操作，如亮度、对比度、伽玛值等等，这部分操作通过发送VIDIOC_S_CTRL命令来完成。

应用程序中通过ioctl系统调用发送的V4L2命令最终会发送到驱动，驱动执行相应命令后将执行结果返回给应用程序。通过ioctl系统调用发送的命令，在驱动中的入口一般为驱动中的ioctl回调函数，ioctl 回调函数中负责将该命令附带的参数数据拷贝到驱动空间内并根据命令执行对应的代码。

V4L2中的另一个特点是其数据传输的方式，v4l2_buffer代表一个视频帧，驱动中可以使用 v4l2_buffer 队列来管理传输缓冲区，应用程序读数据是从v4l2_buffer队列中取出一个视频帧，使用完后将视频帧再次入队，进行重新填充，如此循环来获取数据。

3.2 V4L2交互框架的必要性

在内核驱动中，应用程序和驱动程序的交互可以通过基本的read、write，更复杂的交互可以通过ioctl命令。无论是read、write还是ioctl命令，都是Linux的标准系统调用，也就是说，驱动程序和应用程序不需要关心read、write和ioctl是如何从用户空间经过复杂调用和转换，最终将指令发送到对应的内核模块。应用程序只需要使用这些函数，而驱动程序只需要负责对接收到的read、write、和ioctl等命令进行处理， Linux内核负责了应用程序和驱动程序之间的消息和数据转发。

系统调用的实质是内核空间向用户空间提供的一种接口，用户进程可以通过read、write等系统调用来访问设备文件[17]，执行系统调用时，会在用户态和内核态之间发生切换。在编写用户空间驱动程序时，驱动和应用程序都工作在用户空间，无法直接使用系统调用来实现消息和数据的转发，因此需要有一种新的框架来支持用户空间驱动和应用程序之间的消息数据传递，并且需要特定的驱动标准，对于用户空间的视频设备驱动，本文提出的V4L2交互框架正是起到了相应作用。

3.3 V4L2交互框架整体架构

V4L2交互框架有以下特点：首先，该交互框架有着良好的通用性，支持所有符合V4L2标准的视频设备驱动，同时，该框架还有着良好的扩展性，进行扩展后，对可以支持其他类型设备的用户空间驱动；其次，该交互框架实现了相应的open、close、mmap、ioctl等原有系统调用的基本功能，并提供了类似的调用接口，使的原有的应用程序仍可以只需修改极少源代码的情况下实现对用户空间驱动的访问和使用，用户空间驱动程序则仍可以保留内核驱动对open、close、ioctl命令传统的处理方式；最后，该框架充分借鉴和利用了Linux内核中的消息传递机制，有较高的执行效率，框架结构如图3.1。

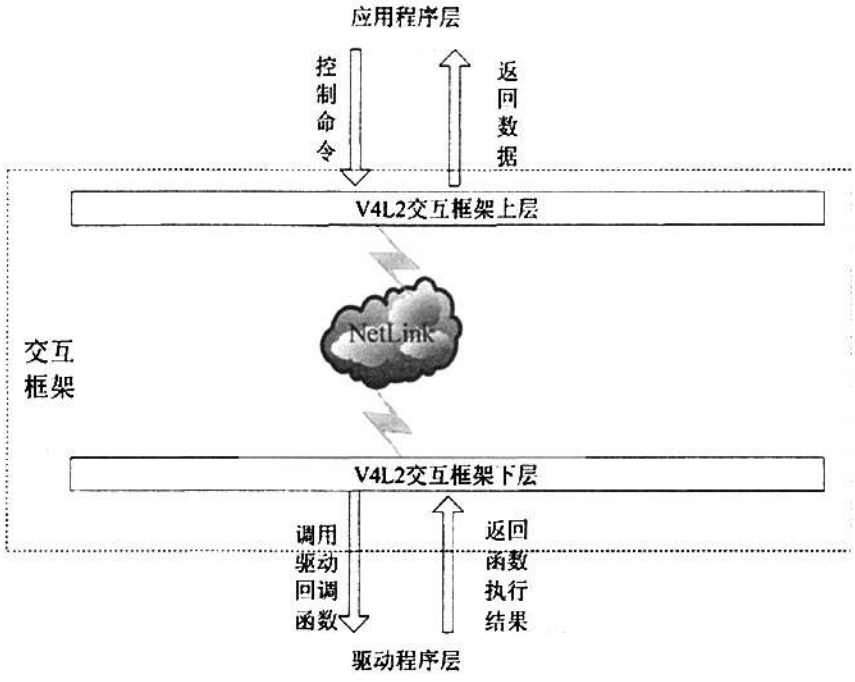


图 3.1 V4L2 交互框架结构图

如图 3.1 所示，V4L2 交互框架可以分为两层，上层主要负责接受和解析应用程序的控制命令，并将其转换成 V4L2 交互框架的消息格式发送到交互框架下层。下层根据接收到的框架消息，进行解析，并调用驱动程序相应的回调函数进行处理。交互框架并不是一个独立运行的进程，框架上层运行在应用程序的地址空间内，而框架下层运行在驱动程序的地址空间内，因此有着较低的开销。V4L2 交互框架上层和下层之间通过 netlink 发送消息命令。netlink 是 Linux 中用户空间和内核空间通信的基本方法[18]，Linux 内核中的消息传递和转发机制基本都是基于 netlink 实现的，如 ioctl 系统调用的消息发送等，不过 Netlink 同样可以在用户进程间通信间使用，netlink 把进程 PID 作为通信地址，更加方便了进程间的消息传递。

交互框架最大程度上屏蔽了用户空间驱动和内核空间驱动调用接口上的差异，并有良好的通用性和扩展性。用户空间的驱动程序不需要关心与应用程序的消息数据的与控制命令的传递问题，应用程序也可以通过与原系统调用类似的接口来对用户空间驱动程序进行调用。

3.4 V4L2交互框架实现

内核驱动中，Linux 内核自动完成了应用程序和驱动程序之间的消息命令传递、接口调用的标准化，因此在开发内核驱动时，只需要按照一定的步骤调用系统函数进行设备和驱动的注册，当应用程序使用某个系统调用访问设备时，Linux内核中相应的机制进行应用程序和驱动之间的消息传递，内核空间和用户空间之间的数据拷贝。对于视频设备用户空间驱动，交互框架需要实现以下的功能：1，设备文件的生成，在内核驱动中，使用相应的设备注册函数后，便会生成对应的设备文件，如/dev/video0，应用程序通过 open 该设备文件，便可以和驱动进行交互，在用户空间驱动中，也需要实现设备文件的生成；2，控制命令的传递，当应用程序发送 ioctl、open、close、等命令时，需要完成驱动与应用程序之间的消息数据的传递与拷贝；3 驱动与应用程序之间内存的映射，对于视频设备驱动一般通过共享内存来传输数据，因此交互框架需要完成驱动和应用程序之间的内存映射。

3.4.1 基于 netlink 的消息传递机制

netlink在Linux中被广泛用于用户空间和内核之间的消息传递，Linux2.4版本后的内核中，几乎全部的中断过程与用户态进程的通信都是使用netlink实现的[20]。相对于其他IPC通信手段，netlink是一种全双工异步的通信方式[26]，更适合于复杂的消息数据传递和处理，并且netlink支持组播的消息传递方式，更是其他通信手段不可比拟的。netlink在本质上是一种基于socket的通信方式，但相对于传统的socket，netlink通信地址不是IP和端口号，而是以进程PID作为通信地址，这为进程间的消息传递提供了很大的便利。当用户空间进程和内核通信时，内核被看作一个大的进程， PID为0。当用户空间进程互相通信时，PID就为各自的进程ID，通过getpid()函数可以获得进程自身的PID。

netlink的初始化函数如下：socket(PF_NETLINK, socket_type, netlink_family)，sokcet_type可以为SOCK_RAW和SOCK_DGRAM, netlink_family表示了通信协议，对于用户空间驱动和应用程序之间的消息传递，通信的双方必须有相同的协议，netlink规定了一系列的标准协议，有NETLINK_ROUTE、NETLINK_ROUTE6、NETLINK_ARPD、NETLINK_FIREWALL等等，不同的协议对应着不

同用途，如NETLINK_ROUTE被用来修改、获取设备信息，而NETLINK_ARPD用于维护arp表信息。为了不和netlink所规定的标准协议发生冲突，在交互框架中，新定义了一个协议NETLINK_USRDIVER用于框架中的通信，netlink_family实质上是一个整型值，具体做法如下

```
#define NETLINK_USRDIVER 22
```

V4L2 交互框架中有两个 netlink 对象，分别位于框架上层和下层，两个 netlink 对象负责驱动和应用程序之间的框架消息发送。netlink 中的 struct nlmsghdr 表示了一个消息头，结构如下：

```
struct nlmsghdr {
 u32 nlmsg_len;
 u16 nlmsg_type;
 u16 nlmsg_flags;
 u32 nlmsg_seq;
 u32 nlmsg_pid;
}
```

nlmsg_len 表示了该消息头的所附带的数据区的长度(包括消息头的长度)，nlmsg_pid 表示了消息发送者的进程 PID，NLMSG_DATA(nlh)宏则返回了消息头的实际数据区域。因此，发送一个消息，至少需要设置 nlmsg_len、nlmsg_pid、、NLMSG_DATA(nlh)三个字段的值。除了 struct nlmsghdr ，发送消息还需要设置 struct iovec 和 struct msghdr 两个结构， struct iovec 结构支持将多个消息放入该结构，通过一次调用送多个消息， struct msghdr 表示了发送的消息结构整体，包括了消息内容和发送目的地地址，设置完后通过 sendmsg 函数将消息发送。

netlink相对与传统的socket，不存在客户端和服务器端的概念，通信的双方可以主动发起通信，也可以被动的接收信息，因此在框架运行时，既可以由应用程序通过上层的netlink对象主动向驱动发送消息命令，当驱动中发生特定事件时，驱动也可以通过下层的netlink对象主动通知应用程序。

3.4.2 设备文件生成

在内核驱动中，可以通过 register_device 等函数在/dev 目录下生成特定的

设备文件，应用程序通过该设备文件来调用驱动。在V4L2交互框架中，使用了命名管道来实现设备文件的生成。

管道在Linux中有匿名管道和命名管道两种，匿名管道只能在同一进程内使用，而命名管道可以在进程间使用。Unix系统中还有流式管道，流式管道支持全双工的通信，但Linux中的管道只支持半双工的通信。向管道写入数据会阻塞直到管道另一端有进程读取该数据，Linux管道也支持非阻塞的读写，但非阻塞的方式不能保证数据能够被正确读写。

当用户空间驱动被加载时，交互框架会为该驱动在/tmp目录下生成一个命名管道并向该管道写入驱动进程的PID，从应用程序的角度来看，该管道即为设备文件。当应用程序调用框架接口打开该管道文件时，交互框架读取该命名管道并返回驱动的进程PID。当应用程序调用u_ioctl(fd，cmd，args)函数向驱动发送控制命令时，文件操作符fd即为驱动进程PID。交互框架就将该ioctl函数及其参数进行解析并转换成netlink消息，并根据参数fd中的来确定消息的接收方的PID，从而来发送消息命令。

使用命名管道来生成设备文件的另一个好处是对于多个应用程序访问的控制，视频设备与其他设备不同，一般都有访问的唯一性，即同一时间只能有一个应用程序能够使用视频设备。当有一个应用程序成功打开该设备文件后，其他应用程序无法从该设备文件对应的管道中读到进程PID，即无法访问该驱动，当应用程序释放对该驱动的使用后，框架将驱动的PID重新写入该管道，这样该驱动可以重新被其他应用程序所使用。

3.4.3 关键数据结构

交互框架上层为应用程序提供调用接口，下层主要负责驱动注册、设备文件的生成、netlink守护线程的运行、进程间的数据拷贝等。其中主要的数据结构如下：

```
struct u_device
{
  struct u_file_operations * file_operations;
  enum u_dev_type type;
  unsigned int pid;
  key_t shmkey;
};
```

struct u_device 表示了一个用户空间驱动，file_operations 表示了该驱动的操作函数集合，type 表示了设备类型 pid 表示了该驱动的进程 PID，shmkey 表示驱动创建的共享内存的标志符。struct u_file_operations 的结构如下：

```
struct u_file_operations
{
int (*open)(struct u_file*);
int (*release)(struct u_file*);
int (*ioctl)(struct u_file*,unsigned int,void *);
int (*mmap)(void * addr,int len,int port,int flags,int fd,int offet);
 }
```

该结构中的函数指针指向驱动中定义的 open、relese、ioctl、mmap 等函数，当交互框架下层收到框架的消息后，对消息进行解析后，就通过 u_file_operations 的函数指针来调用驱动的函数。在驱动程序中，向交互框架注册驱动的步骤如下：首先初始化 struct u_file_operations f_ops 对象，并对其中的 open、release、ioctld 等函数指针进行赋值，然后初始化 struct u_device device 对象，并对其中 file_operations、type、pid、shmkey 分别赋值，最后通过 register_device(&device)向交互框架注册该驱动。注册完驱动后，交互框架获得了该驱动的所有的基本信息，能够根据实际的框架消息来调用该驱动。

3.4.4 消息封装

对于视频驱动，应用程序使用的最多是的 ioctl 系统调用，ioctl 原型如下：

```
int ioctl(int fd,int cmd,void *args)
```

交互框架中封装了 u_ioctl 函数来实现了 ioctl 的功能，两者仅在函数名上

有区别，参数个数和类型都一致。对于 u_ioctl 调用，需要封装在框架消息体内的参数有两个，cmd 和 args，fd 在交互框架内是被作为 netlink 的发送地址，因此不需要封装在消息内。cmd 参数表明了该调用的 V4L2 命令类型，如 VIDIOC_S_FMT、VIDIOC__DQUEUE 等，该参数为一整型变量，因此可以方便的以四个字节将其封装在消息体内。args 参数表示了该命令附带的参数的指针。在内核驱动中，该指针参数作为 long 型变量直接传递给驱动，内核驱动可以调用 copy_from_user 函数来把该指针的数据从用户空间拷贝到内核空间。但在用户空间驱动，如果传送实际的指针，由于地址空间的不同，是无法进行进程间的指针数据拷贝的。在用户空间驱动，需要传参数数据，而不是参数指针。

在交互框架中以实际数据来传输该参数，需要把数据从参数指针中拷贝出来，args 参数可能的类型有很多，如 struct v4l2_capabilit、struct v4l2_format、struct v4l2_buffer 等等， 但在 u_ioctl 函数内，参数指针类型都退化成了空指针，显然无法从空指针中获得数据区域的实际大小，并且不同类型的 args 参数结构和大小都不同，如何从空指针中拷贝到合适大小的数据，首先可以参考下 Linux 内核代码中对 ioctl 参数拷贝的处理。

内核驱动中的 ioctl 回调函数实际获得了一个指向参数的 long 型的指针，内核中一般调用 video_usercopy 来拷贝该指针的实际数据，而 video_usercopy 中又会调用 copy_from_user 函数来拷贝数据，调用形式如下：

```
copy_from_user (parg, (void _user *) arg, _IOC_SIZE (cmd))
```

parg 为拷贝的目的地，而 arg 则是参数指针，cmd 为命令类型，_IOC_SIZE(cmd) 计算出了该命令对应参数的长度。_IOC_SIZE 宏的定义如下：

```
#define _IOC_SIZE (nr) (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
```

通过查看 linux/include/asm-i386/ioctl.h 文件，可以得到_IOC_SIZESHIFT 宏为 16，而_IOC_SIZEMASK 宏为 255。因此从命令的类型可以推断出参数的具体长度，举例来说，VIDIOC_S_FMT 命令对应的参数类型为 struct v4l2_foramt，_IOC_SIZE(VIDIOC_S_FMT)的值为 204，即为 sizeof(struct v4l2_foramt)的值，因此在交互框架中，也使用了该方法来计算空指针数据的长度。

消息体的封装可以采取以下办法：0 到 3 字节为命令类型，4 到 5 字节为参数的长度，6 字节到消息末尾为参数数据。如果采取这种办法来传输 u_ioctl 调用，以 VIDIOC_S_FMT 命令来说，首先框架上层需要把 struct v4l2_foramt 对象

从应用程序拷贝到消息体中，然后将消息体发送到框架下层，驱动程序从框架下层读取消息体中的数据，最后将执行结构返回给应用程序，一共进行了4次共916字节的数据拷贝。在内核中,对应的ioctl参数数据传递只需要copy_from_user和copy_to_user两次数据拷贝操作，对比之下，上述的方法的开销相对过大。

通过利用共享内存的特性，应用和驱动之间的参数数据传输可以通过两次拷贝完成,实现方法如下:在框架上层申请一小块共享内存用于参数数据传递，当应用程序调用u_ioctl函数时，将args指针内的数据拷贝到该共享内存中，拷贝字节数根据_IOC_SIZE (cmd)来确定，然后消息体以以下格式封装：0到3字节为命令类型，4到7字节为参数共享内存的标志符，8到11字节为参数相对于共享内存起始位置的偏移量，考虑到在视频驱动中，应用程序对驱动的访问往往是同步的，一般后一个操作要等前一个操作完成才能进行，所以在共享内存中参数一般只有一个,参数相对于共享内存的偏移量一般为0。框架下层接收到消息体后，通过共享内存标志符，将偏移量转化成绝对指针地址，驱动直接对共享内存中的参数进行读写，执行完毕后，框架上层把共享内存中的参数数据拷贝到应用程序，因此驱动和应用之间的消息传递只需两次数据拷贝。

3.4.5 接口实现

V4L2交互框架上层向应用程序提供了u_open、u_close、u_ioctl、u_mmap四个调用接口，u_open和u_close主要为关闭和打开设备。使用最频繁的是u_ioctl,负责所有的V4L2命令发送,而u_mmap则是将驱动中的传输缓冲区内存映射到应用程序中。函数原型如表3.3。

表 3.3 V4L2交互框架上层接口

函数原型	说明
int u_open(const char *filename)	打开设备
int u_close(int fd)	关闭设备
int u_ioctl(int fd, int cmd, void *args)	发送V4L2命令
void * u_mmap(void *start, int length, int port, int flags, int pid, int offset);	将驱动的传输内存映射到应用程序

open函数是去读取驱动生成的命名管道文件，从而获得驱动的进程PID，随后的ioctl将该驱动进程的PID看做是设备的操作符，ioctl函数中将相应的参数数据拷贝到参数共享内存中，然后根据PID,将封装的消息发送到驱动进程。而mmap函数首先或去获得驱动传输缓冲区内存的标志符，然后对该标志符对应的共享内存进行映射。应用程序对该框架接口的调用代码如下：

```
v4l2_buffer buf;
v4l2_requestbuffers rq;
int pid=u_open("/tmp/video0",0);//打开设备
rq.count=4;
u_ioctl(pid,VIDIOC_REQBUFS,&rq);//申请传输缓冲
 for(int i=0;i<4;i++)
{
   buf.index=i;
u_ioctl(pid,VIDIOC_QUERYBUF,&buf);//查询传输视频帧
mem[i]=(unsigned char*)u_mmap(NULL,buf.length,0,0,pid,buf.m.offset);
 }
u_ioctl(pid,VIDIOC_STREAMON,NULL); //开始传输
while(1)
{
int ret= u_ioctl(pid,VIDIOC_DQBUF,(void*)&buf);//获取视频帧
ret=u_ioctl(pid,VIDIOC_QBUF,(void*)&buf);//空视频帧入队
}
```

框架下层向驱动程序提供了void register_device(struct u_device * device)接口，该接口用来向框架注册驱动，调用代码如下：

```
struct u_device device;//
struct u_file_operations f_ops;//
f_ops.open=v4l_open;//
f_ops.release=v4l_release;
f_ops.ioctl=v4l_ioctl;
device.type=TYPE_VIDEO;
device.pid=getpid();
device.file_operations=&f_ops;
```

```
device.shmkey=shmkey;
register_device(&device);//
```

device代表驱动结构，f_ops表示该驱动的操作符集合，shmkey表示该驱动传输缓冲区的共享内存标志符，在这里，交互框架负责了共享内存映射的处理，因此驱动实际上不需要实现内存映射的处理

3.5 本章小结

本章首先介绍了Linux 下的视频标准V4L2，分析了V4L2的工作原理，针对现在用户空间驱动缺少的框架支持的情况，提出了基于用户空间视频驱动的V4L2交互框架，分析了该框架的工作原理以及实现技术，重点研究了该框架的运行效率，并给出了该框架的调用接口。

第4章 USB视频设备用户空间驱动设计

随着多媒体技术的发展，电视卡、摄像头等视频设备也层出不穷，而USB由于其即插即用，传输速度快的优点，被许多视频设备采用为标准传输接口。针对不同类型的USB视频设备，往往需要开发新的驱动，但不同类型的USB视频设备驱动在实现原理的上有很大的共性，如原始数据的读取、缓冲区的设计、数据传输的方式等，可以说在基本的驱动结构上是类似的，不同的地方可能在于某些硬件有一些厂家自定的控制命令，需要在驱动中实现，还有的硬件需要在运行时需要写入 Firmware，一些特定格式视频数据需要进行编解码，的有些设备可能还需要实现 I^2C 驱动等。、

本文结合V4L2交互框架，从USB视频设备驱动的共性出发，利用Libusb库，设计并实现了USB视频设备在用户空间下高效稳定的驱动方案，对用户空间下实现某些特定的USB视频设备的驱动做了有力支持。

4.1 USB视频设备协议

Universal Serial Bus简称USB，是1994年底由Microsoft、IBM等公司提出的外部总线标准，现在主流的USB设备一般为USB1.1和USB2.0版本，USB1.1好的最大传输速率为12Mbps,而USB2.0则达到了480Mbps。USB有着接口统一、支持扩展、热插拔、独立供电的优点，被计算机外设广泛采用[30]。USB规定了USB硬件设备的传输标准协议，对于具体类型的设备，USB可以分为USB通信类设备、USB视频类设备、USB人机交互类设备等等，每一类设备都有相关的标准和协议。这些协议首先是符合USB标准协议的，但在原有框架的基础上对USB协议进行了扩充，便于驱动程序的管理和开发。USB视频设备协议包含了摄像头、数码摄像机、数字电视调谐器、支持视频流的照相机等USB接口硬件的相关标准[10]。

USB设备在组织上包含四个逻辑层次即：设备、配置、接口、端点。设备包含了该USB设备的基本属性，如设备协议号，产品编号等；配置决定了该USB表现出何种属性：如是视频设备还是存储设备等，一个设备中可以包含若干个配置；端点是USB通信的最基本形式：一个端点对应一个数据传输通道，端点可分为中断端点、控制端点、批量传输端点、等时传输端点等；接口是相关端点的集合，一个接口完成特定的功能，如发送控制

视频参数、或是传输视频数据等，而一个配置可以包含多个接口[11]。具体如图4.1所示：

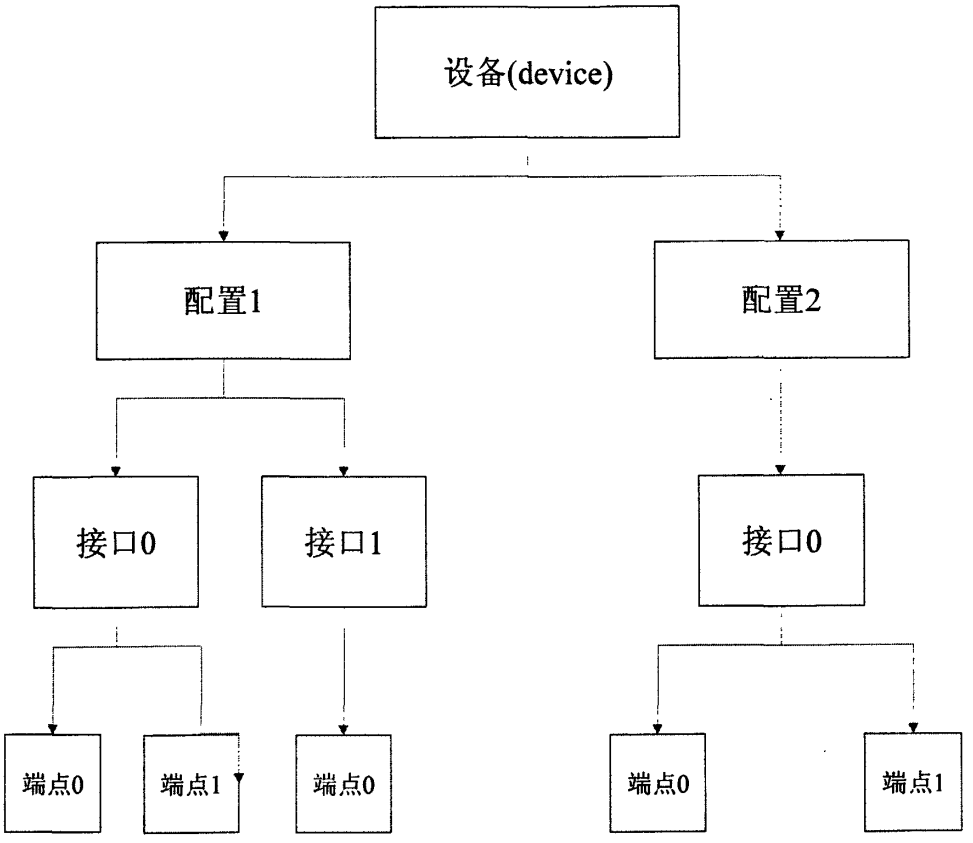


图 4.1 USB设备的配置、接口与端点关系图

一个接口中，往往会包含几个设置，这几个设置一般共享该接口的硬件资源，如端点等。但每个设置会对这些硬件资源采取不同的参数设置，如会选择不同的端点传输速率等。一个接口中只有一个设置能被选择为激活状态，其他的设置则是处于备用状态，驱动程序需要根据应用程序输入的参数，来决定具体选择哪个配置。

根据《 Universal Serial Bus Device Class Definition for Video Devices》的定义，一个标准的USB视频设备的拓扑结构如图4.2所示。

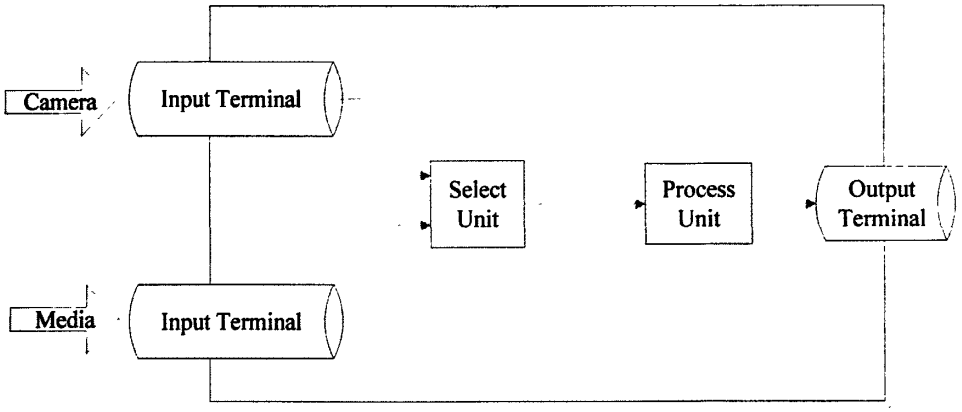


图 4.2 USB视频设备拓扑结构

Input Terminal 表示该USB设备的输入，输入一般可以分为两种，一种是相机(包括摄像头)的输入，另一种是多媒体数据的输入，如数字电视信号、流媒体数据等。作为相机输入的 Input Terminal 一般被称为 Camera Terminal, Camera Terminal 支持自动对焦、自动曝光等等特性。Select Unit 表示输入选择单元，可以在若干个输入中选择一路，作为进行处理的数据。Process Unit 代表处理单元，Process Unit 一般选择 Select Unit 的输入作为其输入，在 Process Unit 中，进行视频数据的进一步处理，如亮度、对比度、锐度、白平衡等等。Output Terminal 表示视频数据流的输出，一般 Output Terminal 只有一个。另外,有些USB设备中还存在 Extension Unit, Extension Unit 表示扩展处理单元，该单元对视频数据做额外的处理。实际的设备中，一些单元并不是必须的，如 Select Unit,如果只有一个输入单元,则不需要 Select Unit。另一个值得注意的是，这些 Unit 或 Terminal 所规定的具体特性的实现并不是强制的，实际设备实现的功能可能只是标准规定的一个子集。

在 Linux 下输入 lsusb -v 命令，如果此时有一个 USB 视频设备被连接到计算机,终端下除了显示标准的 USB 配置、接口、设置、端点等描述信息，还会显示视频类设备特定的描述信息，主要包括 Interface Association、VideoControl Interface 、VideoStream Interface 。VideoControl Interface 表示对该视频设备进行控制的接口，VideoStream Interface 表示对数据传输的接口。图 4.3 表示了一个 USB 视频设备典型的接口描述符。

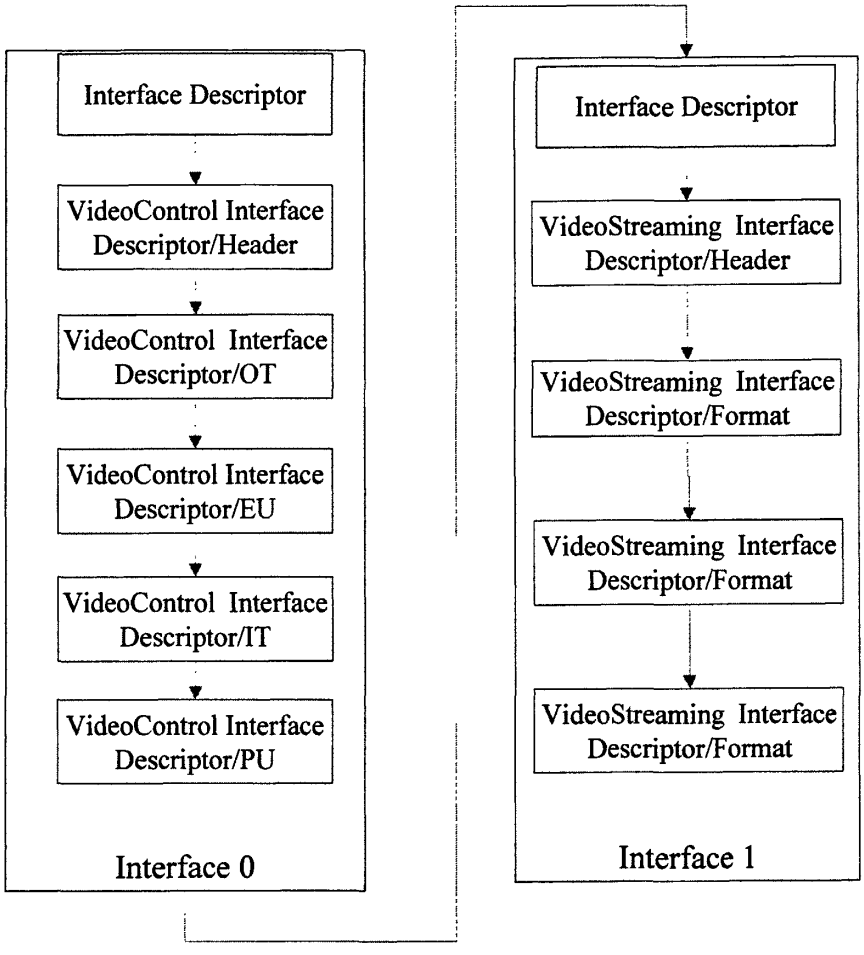


图 4.3 典型USB视频设备的接口描述符

每个接口以一个标准的接口描述符开始， Interface 0中的标准接口描述符描述了该接口的类型为VideoControl，紧跟其后的是VideoControl的接口描述符，该描述符的 subtype 为 Header，表示该描述符为VideoControl 的起始描述符。接着是 VideoControl/OT 描述符，描述了Output Terminal 的特性；接下来的EU、IT、PU描述符分别描述了 Extension Unit、Input Unit、Process Unit 的特性。Interface 1 中标准接口描述符表明了该接口的类型为VideoStreaming，紧跟其后的是VideoStreaming接口描述符，该描述符的 subtype 为 Header，表示该描述符为该VideoStreaming接口的起始描述符。之后的各个Format描述符中描述了该

Interface支持的数据传输格式，传输格式主要包括视频的宽高度、最大最小传输速率、视频帧间隔等。

每个接口中除了包括标准接口描述符以及VideoControl或VideoStreaming描述符,还包含了若干端点描述符。对于VideoControl接口，端点的类型主要为控制端点和中断端点，这两种端点负责小数据量的控制命令传输,控制端点作为0号端点一般是默认存在的。VideoStreaming接口也包括若干个端点，端点的类型主要为批量传输和等时传输端点。批量传输端点和等时传输端点都用来大数据量的传输，在传输速度方面，批量传输端点的端点优于等时传输端点，但在实际应用中，由于等时传输的实时性更强[12]，因此对于视频类设备，等时传输比批量传输更常用。表4.1是批量传输端点和等时传输端点的特性比较。

表 4.1 批量传输端点和等时传输端点比较

	优点	缺点
批量传输端点	(1) 充分利用传输带宽，较快的传输速度 (2) 能够保证数据传输的无误性	(1)无法预留带宽，与其他设备竞争时可以严重降其传输速度 (2)不能保证数据传输实时性
等时传输端点	(1) 良好的数据传输实时性 (2) 能够预留传输带宽,稳定的传输速率。	(1) 带宽利用不充分 (2) 无纠错能力

由此可见，对于实时性较高的视频数据传输，如数字电视或视频会议等应用，等时传输是一种更好的选择。对于如大容量存储等外设，批量传输则更符合要求。

4.2 Libusb 库

Usb file system 提供了用户空间下对 usb 设备进行通信的函数接口和数据结构，在开发用户空间驱动时，可以直接利用这些函数接口来实现对USB 设备的控制和传输。但这些接口比较复杂和低层，不利于提高开发效率，使用过程中也容易出错。在用户空间，则可以通过 Libusb 库来进行驱动的开发。Libusb 对 Usb file system 提供的函数接口和数据结构进行了封装，不仅很大程度减少一些重复性的工作，如 USB 设备的探测等，并且能够有效减少程序中由于程序中函数和数据结构使用不当造成的错误。Libusb 的和 Linux 内核中 USB-Core 在功能和原理上无本质的区别，不过 libusb 更加层次化和结构化，是一个相对高层的 API，能够提高驱动开发的效率，代表了驱动开发的一种趋势[13]。

4.2.1 Libusb 同步与异步访问机制比较

Libusb 对 USB 设备的访问提供了两种机制，即同步访问和异步访问，同步访问对 USB 设备的读写会一直阻塞直到执行完毕；异步访问对 USB 设备进行非阻塞的读写，执行完异步的读写函数后，程序不需要等待读写函数的返回而可以继续执行，异步读写的结果会在用户定义的回调函数中进行处理。同步访问比起异步访问相对简单，因为读写函数都是以阻塞的方式进行，不存在多个读写函数同时对 USB 设备进行访问的情况，所有的读写函数都是顺序进行的；而异步访问则教复杂，可能有多个函数同时对 USB 设备进行读写，函数的返回顺序和调用的顺序不一定相关，更复杂的情况可能在于一个异步读写函数出错后，会影响到其他的读写函数,这时程序必须对出错的异步读写函数进行适当的处理，这样才能保证不影响其他函数对 USB 设备的访问。不过异步访问往往比同步访问所提供的功能更强大，首先其提供了非阻塞的读写，这样提高了程序的执行效率。其次，对于大数据量传输的读写,如获取视频数据等，同步读写往往会有较长的等待时间，这种情况下，异步读写的优点更明显了。最后，异步读写某种程度上是一种多线程的机制，这样可以提高 USB 数据读写的吞吐量。

对 USB 设备的同步访问主要用在发送控制命令等操作，如 Linux 的 USB-Core 中的 usb_control_msg 函数，该函数负责向 USB 设备发送控制命令，便是一个同

步的读写函数。之所以向USB发送控制命令等操作时用同步的访问机制，主要原因如下：首先,控制命令实际传输的数据量小，函数响应和返回的时间较快，一般不会造成较长的阻塞；其次，对USB设备的控制操作通常有顺序依赖关系，即后一条控制命令的执行可能依赖于前一条控制命令执行的结果，在这种情况下，异步的读写显然不能很好的满足这种要求。

对USB设备的异步访问主要是用在数据传输上，USB2.0的理论传输速度可以达到 480Mbps,但实际传输速度可能会受各种因素影响，因此数据传输时，特别数据传输量较大时，同步访问往往会有明显的阻塞现象，这种情况下,一般需要异步的传输方式，Linux USB-Core 的urb传输便是一种异步传输方式 。

4.2.2 Libusb等时传输方式

Libusb提供的异步传输方法和Linux USB-Core的URB传输十分相似，两者在原理上是一样的，用到的数据结构上也十分相似，只是某些字段和封装上有些区别，最主要的区别在于Linux USB-Core的URB传输支持DMA特性，但在Libusb中并不支持，在用户空间驱动如何使用DMA特性也是一个待研究和解决的问题。Libusb 提供的异步传输包含了控制、中断、批量传输、等时传输的四个类型的异步传输方式，用的比较多的还是中断、批量传输、等时传输这三个类型的异步传输。

以传输实时数据常用的等时传输方式来说，在libusb中发起一个等时传输需要以下几个步骤：首先为一个传输申请适当数量的等时传输包，过少的等时传输包不能充分利用带宽，而过多的等时传输包会造成资源浪费，因为可能在传输过程中一些多余的等时传输包中都没有实质的数据存在；接着要为等时传输包申请内存，用于存放传输获得的数据；然后要为该等时传输设置相应的参数，主要为传输端点、完成回调函数、超时时间等。USB-Core 中的等时传输一般还需要设置 URB 的轮询间隔，实际上该间隔一般被设置为传输端点自身的轮询间隔，Libusb中则自动设置了该轮询间隔。最后，通过libusb_submit_transfer提交该传输申请，数据传输便开始进行，传输完成后，自动调用完成回调函数进行数据处理的工作。

4.3 用户空间下视频设备驱动设计

与内核驱动相比，用户空间驱动的设计与实现有着一些较明显的区别，主要为：1，内核驱动需要module_init和module_exit作为驱动的入口和出口函数，用户驱动则是一个独立的进程，由相关的代码来实现驱动的初始化和清理工作；2，内核驱动可以直接使用内核函数和数据结构等，而在用户空间，这些函数和数据结构是无法直接使用的，一些能够通过对源代码进行修改后在用户空间使用，如list_head数据结构，而有些在用户空间是没有权限使用的，如关中断函数。但很多情况下，在用户空间可以找到更方便的函数或数据结构进行替代；3，在应用和驱动之间的消息的传递机制上，内核驱动由内核提供消息数据的转发和传输，而在用户空间，需要依赖与V4L2交互框架

4.3.1 用户空间USB视频驱动整体设计

USB视频驱动在整体上可以分为三部分：1，与USB设备进行交互的模块，主要为发送控制命令和接收USB数据，USB数据包括两大类，USB设备信息和USB视频数据；2，与应用程序进行交互的模块，主要为接收应用程序的命令并进行解析，然后去调用相应模块中的函数。3,传输缓冲区的模块,该模块负责对传输缓冲区的管理和对原始视频数据的重组等操作。三个模块的具体关系如图4.4所示。

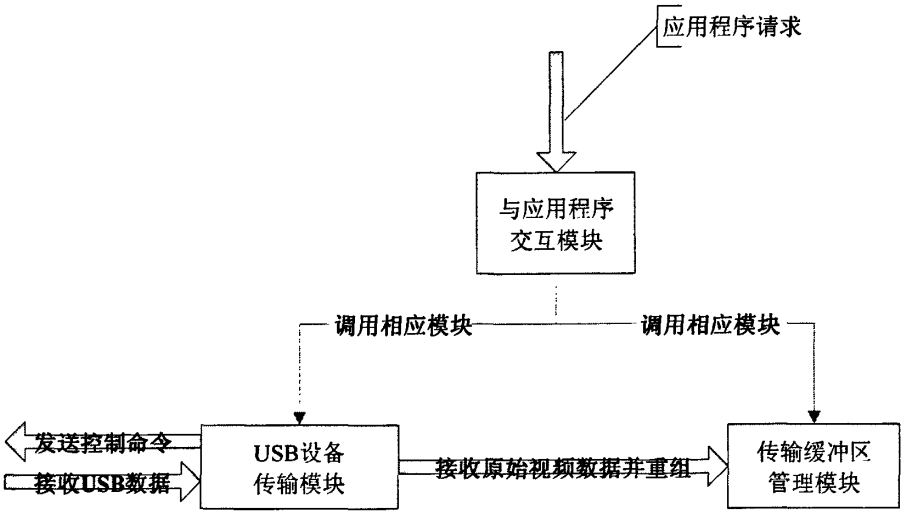


图 4.4 驱动整体模块示意图

4.4 USB设备传输模块设计

USB设备传输模块负责向USB设备发送控制命令和视频数据等时传输，控制命令功能主要包括：

- 选择输入

  如果视频设备有多路输入，可以选择其中一路作为输出

- 设置视频宽高与像素格式

  视频设备提供多种宽高比的视频输出，驱动程序需要根据应用程序指定的宽高度，从视频设备可选的宽高度组合中选择最匹配的。

- 设置传输带宽

  一般USB设备有多个传输设置，每个设置有不同的带宽，来针对不同视频格式的传输，驱动需要根据视频的宽高度和像素格式来选择合适的带宽。

- 设置视频的帧率

  设置视频的Frame Per Second,即FPS。

- 设置视频的亮度、对比度、伽玛值等显示属性

  视频的显示属性一般在硬件的Process Unit被处理，驱动处理显示属性的能力依赖于硬件的Process Unit处理能力，如果Process Unit不支持相关的实现属性操作，驱动也就无法执行对应的V4L2命令。

4.4.1 URB等时传输

USB设备传输模块另一个主要功能是视频数据的等时传输，Linux内核中USB数据传输的基本单位为URB(usb request block)[29]，Libusb库中则以libusb_transfer来表示一个URB，本章中为了更清晰的表述，暂时用URB来代替libusb_transfer。一个等时传输的URB可以包含多个等时传输包，传输过程中，每个等时传输包都对应着一个独立的子传输过程。传输完成后等时传输包中存储了原始的视频数据，原始的视频数据需要经过解析和重组后才能组成一个完整的视频帧。一个完整的视频帧的数据量往往需要多次URB传输才能完成，大

概的估算方法如下，假如一个完整视频帧的大小为 153600 字节,一个 URB 内有 32 个等时传输包，每个等时传输包容量为 800 字节，那理论上传输完一个视频帧至少需要 5 次 URB 传输，考虑到传输过程携带的视频头信息和非满载的情况，实际上需要远不止 5 次的 URB 传输。

URB 是异步传输,因此每一个 URB 有一个完成回调函数来处理 URB 传输完成后的工作。完成回调函数中对 URB 内的数据做进一步处理，包括解析视频头信息、将原始分散的视频数据重组形成完整的视频帧，最后重新提交 URB 进行再次传输。如果驱动只有一个 URB 在进行视频传输的话，就可能出现这种情况，当该 URB 完成传输后，正在完成回调函数中进行数据处理时，该时间段中内 URB 传输处于空闲状态，浪费了可用的传输带宽，降低了驱动的性能。采取两个或多个 URB 同时进行传输可以解决该问题,完成回调函数都对传输缓冲区存在访问竞争，同一时间只有一个 URB 回调函数内访问传输缓冲区，这就可能出现多个 URB 回调函数同时在等待对传输缓冲区的访问，而没有 URB 进行数据传输的状况，而采取多个 URB 能减少这种情况发生的概率，因此实际传输中采取了 4 个 URB 同时轮询数据的方式。

4.5 传输缓冲区模块设计

传输缓冲区是驱动中保存视频数据的内存区域，传输缓冲区的作用是解决驱动程序采集视频数据和应用程序提取视频数据的不同步关系。如果驱动采集到一个完整的视频帧，然后应用程序马上将该视频帧取走，那传输缓冲区没有存在的必要性了。但实际情况并非如此，驱动不知道采集到视频数据何时会被取走，因此需要一块内存空间来保存这些未被取走的数据，否则应用程序未来得及提取的视频数据就有可能丢失。

4.5.1 视频帧队列

在 V4L2 中，以 v4l2_buffer 来代表一个视频帧，因此驱动中的传输缓冲区中可以被映射为 v4l2_buffer 队列，驱动程序则以 v4l2_buffer 队列来管理该缓冲区。应用程序可以通过 VIDIOC_REQBUFS 命令来指定驱动程序 v4l2_buffer 队列的大小，同时也间接的指定了传输缓冲区的大小。传输缓冲区中的视频帧一般有三种状态，分别为等待状态、完成状态、应用程序使用状态,这三种状态之间

的转换如图4.5：

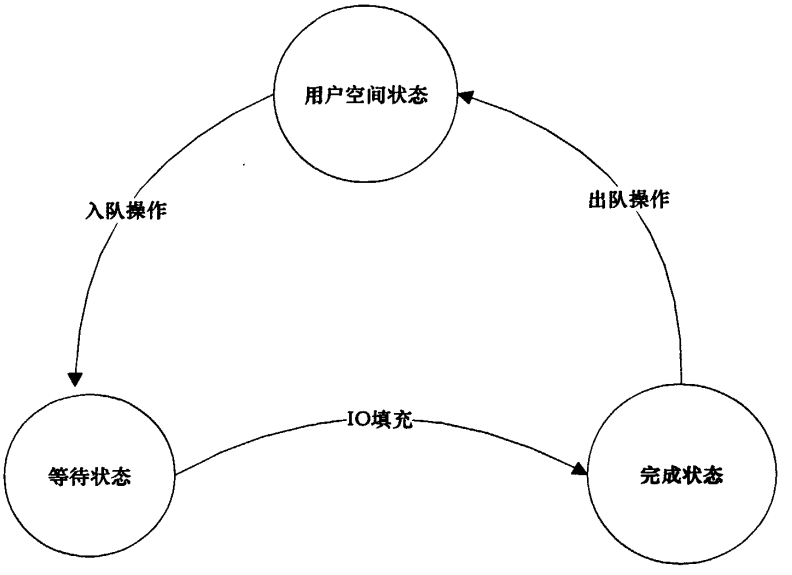


图 4.5 视频帧状态转换图

一开始，缓冲区内所有的视频帧处于等待状态，直到有URB传输完成，按照FIFO的顺序，从等待队列中取出一个视频帧进行填充，经过若干URB传输，视频帧填充完毕。接着该视频帧进入完成状态，并被放入完成队列，若此时应用向驱动发送提取视频帧的请求，则通过出队操作，将视频帧从完成队列中取出返回给应用程序，此时视频帧处于应用程序处理状态，应用程序使用完该视频帧之后，通过入队操作，将其放入缓冲队列呢，此时该视频帧回到了原来的等待状态，重新开始等待数据填充。

从应用程序的角度来看，视频数据的传输是不断调用V4L2的VIDIOC_DQBUF和VIDIOC_QBUF命令来完成的，驱动程序中也要不断的处理相应的出队列和入队列操作，如果以一个数组的形式来管理这些视频帧，每次出队列和入队列时都会要查询一遍该数组，显然不是一个很好的实现方法。因此可以用两个FIFO队列来管理这些视频帧，即等待队列和完成队列。在URB完成回调函数中，取出等待队列头部的视频帧进行填充，填充完毕后，将该视频帧添加到完成队列的尾部，并从等待队列中删除该视频帧。当应用程序提取视频帧时，从完成队列的头部取出视频帧返回给应用程序，并从等待队列中删除该视频帧。待应用程序使用完该视频帧后，再将其添加到等待队列的尾部。等待队列和完成队列中存放的是视频帧的指针，因此队列操作的开销也是较小的，队列的先进先出特性也保证了视频

帧的时间顺序不会被破坏。具体例子如图4.6。

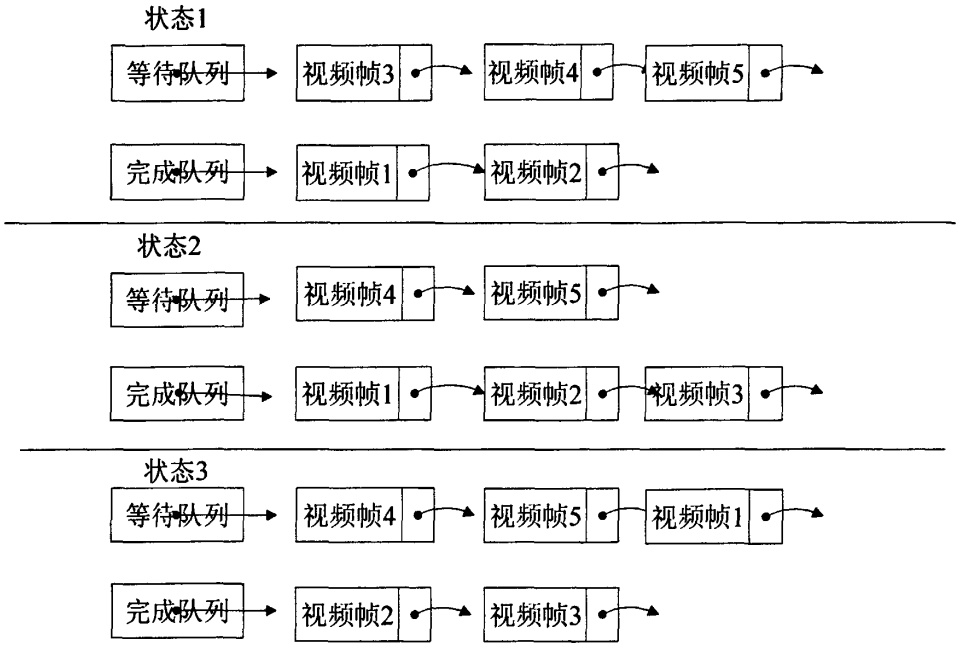


图 4.6 视频队列操作示意图

状态1中，等待队列中的视频帧3被正填充，状态2中，视频帧3已完成填充，被添加完成队列的末尾，而视频帧1正要被应用程序取出，状态3中，视频帧1以被应用程序使用完毕，被添加到等待队列的末尾。

4.5.2 访问同步机制

无论是对等待队列还是完成队列的读写都存在着访问竞争的问题，对于等待队列，有可能在多个URB的完成回调函数中对该队列进行竞争访问，试图获取等待队列中的视频帧进行数据填充。而对于完成队列，访问竞争存在与驱动程序对该队列的操作和应用程序对该队列的读取。对于资源的互斥访问，有两种基本加锁的方法，互斥锁和自旋锁，两者都可以实现互斥访问，但互斥锁和自旋锁最大的区别在于：互斥锁如果不能获得该锁，会进入睡眠状态，操作系统会进行进程调度，其他进程被调入执行，而自旋锁则不会引起睡眠状态，自旋锁类似于一个忙等待，会不断轮询，直到其获得该锁[15]。当锁不能获得时，互斥锁的开销的进程上下文切换时间，而自旋转锁的开销一般是临界区的代码执行时间，因此在两

种锁的选择方法上，如果临界区的代码执行时间小于进程上下文切换时间可以选择自旋锁，反之则选择互斥锁。

对于等待队列的并发访问，主要集中在不同URB的完成回调函数中，临界区代码执行时间较短，并且在 URB 的完成回调函数需要对等待队列进行快速的访问，是一种典型的快进快出的并发访问，过多上下文切换会直接影响到URB提交的频率，从而影响到视频数据的传输效率，因此需要对等待队列采用自旋锁。

完成队列的访问是一个比较典型的生产消费者模型，即驱动程序将填充完的视频帧放入该队列，而应用程序通过调用从该队列取得视频帧，对完成队列的访问会涉及到驱动和应用程序之间的数据交互，临界区代码执行的时间比较长，应用程序对响应时间的要求也不会很高，因而对完成队列可以采用互斥锁。

4.5.3 数据传输方式

在对于视频设备数据传输，如果使用传统的read、write系统调用会造成内核空间和用户空间大量的数据拷贝，形成了额外的开销。因此在内核驱动中，驱动和应用程序之间的数据传输一般是通过内存映射来完成的，经过内存映射后，应用程序可以直接读写驱动的内存，减少了数据的拷贝的开销，提高了传输效率[16]。应用程序一般通过mmap系统调用将驱动中的传输缓冲区内存映射到自己的地址空间中里，然后直接读取被映射的传输缓冲区内存来获得视频数据。

mmap系统调用实质上会去调用驱动程序的mmap方法，因此驱动程序需要实现相应的mmap方法。在用户空间中，驱动和应用程序之间的数据传输可以通过共享内存来实现，而共享内存的映射直接在VL42交互框架中完成，驱动实际上不要实现mmap方法。在本文的第五章详细描述了实现细节。

4.6 本章小结

本章首先分析了USB视频设备子协议，并阐述了USB视频设备的驱动开发的基本原理，并在此基础上分析了用户空间下USB视频设备驱动的设计思路。

第5章 USB视频设备用户空间驱动的实现

USB视频驱动在用户空间实现和在内核空间实现在原理上有很多地方类似，但在具体实现细节上，两者还是有很大差别。主要差别体现在：1，内核空间使用USB-Core来访问USB设备[19]，而在用户空间使用通过usbfs来访问USB设备；2，在用户空间驱动中，需要实现应用程序和驱动程序之间的消息数据交互机制，本文中的V4L2交互框架正是起到了该作用； 3，内核驱动和应用程序通过内存映射来加快数据传输，而用户空间驱动和应用程序可以通过共享内存来达到该目的；4，很多具体的实现技术不同，如队列的管理，加锁的实现等。

传统内核驱动中使用C语言作为开发语言，而C语言类型安全性的缺乏也是影响驱动程序稳定性的一个重要因素[22]，因此这里提出了使用C++语言进行驱动开发的方法。C++语言虽然不是一种类型安全语言，但相对于C语言，有较好的类型转换检查机制，同时相对与其他类型安全语言如Java，又有较高的执行效率，因此采用C++作为开发语言是驱动效率和稳定之间良好的平衡点。

Linux下基本的C++编译器有gcc和g++，gcc一般被用来编译C程序， 但gcc也可以编译C++文件，不过源文件必须以.cpp为后缀名。C++更好的编译检查机制能够减少驱动程序潜在的错误，代码中使用一些未声明的系统调用和函数，如open、close、malloc等，实际使用过程中，在Linux下作为C语言程序可以通过编译，但作为C++语言程序是无法通过编译的，更强的编译检查机制也能够帮助减少程序可能存在的隐患。

5.1 USB设备传输模块的实现

USB设备传输模块负责向USB设备发送控制命令和视频数据的等时传输。本文中利用了Libusb库开发了USB设备传输模块，Libusb库的功能和Linux内核中的USB-Core功能相似，不过Libusb经过了封装和抽象，能够提高开发效率和程序稳定性。

5.1.1 Libusb的核心数据结构

Libusb中有三个比较核心的数据结构，对USB设备的访问，控制命令的发送和数据的接收都和这三个数据结构有关，分别如下：

- libusb_device

  libusb_device代表了一个USB硬件设备，访问该USB硬件的设备属性，如获得USB硬件的总线地址、端点地址、最大数据包长度等都是通过该数据结构来获得。

- libusb_device_handle

  libusb_device_handle代表了一个被打开的USB设备的操作句柄，对USB设备的控制命令的发送和数据的收发都要经过该操作句柄。通过对libusb_device进行打开操作便可以获得libusb_device_handle，同样可以通过libusb_device_handle获得被打开的libusb_device。

- libusb_transfer

  libusb_ transfer代表了一个了USB传输，类似于USB-Core中的URB，不过libusb_ transfer除了可以表示批量传输和等时传输，还可以表示控制和中断传输。工作流程也和URB类似，首先进行初始化，接着申请传输缓存区，并绑定到指定的传输端点，最后在完成相关参数设置后提交该传输。

5.1.2 USB设备的打开

USB设备一般有Vendor ID和Product ID来确定其唯一性，Vendor ID和Product ID都是32位的整数，Libusb中可以通过libusb_open_with_vid_pid函数以指定Vendor ID和Product ID的方式打开一个USB设备，打开成功后返回该设备的操作句柄libusb_device_handle。一个USB设备可以有多个configuration，对于需要的某一种功能，如视频设备功能，则只需要选择对应的configuration，函数libusb_set_configuration提供了相应的功能，对于只有一个configuration的设备，会默认工作在configuration 1。

5.1.3 USB控制命令的发送

控制命令分两类，一类是向VideoControl Interface发送的命令，该类命令和设置视频的显示属性有关，为视频控制命令，另一类是向VideoStreaming Interface发送的命令，该类命令和设置视频传输属性有关，为视频传输命令。USB协议中的标准命令格式如表5.1

表 5.1 USB命令格式

偏移量	域名	大小	描述
0 byte	bmRequestType	1 byte	命令的基本属性
1 byte	bRequest	1 byte	具体命令
2 byte	wVálue	2 byte	命令值
4 byte	wIndex	2 byte	命令的目的地
6 byte	wLength	2 byte	参数的长度

表5.1中给出命令的基本格式和说明，对于某些域名，根据实际设备，其类型和取值可能比较复杂。bmRequestType的比特位7表示了该命令的数据方向，0表示主机到设备，即是设置命令，1表示设备到主机，应该是一个获取命令；比特位5-6表示了该命令是否和具体设备协议相关，如果是00则表示该命令是标准USB命令，和具体设备类型无无关，如果是01 表示该命令和具体设备协议有关。 wIndex的低字节表示了该命令的目标接口。wLength表示命令的附加参数的长度。

一个视频传输命令的例子如下： bmRequestType字段值为0x21, D7:0表示该命令方向 从主机到设备，D6..5: 01表示该命令和具体设备类相关的,D4..0: 1该命令的接收者是接口。bRequest的值为0x01,表示该命令为SET_CUR,即为设置命令。wValue值为0x0200,表示该命令是commit命令,wIndex为0x0001,表示该命令的接收者为1号接口,即为VideoStreaming Interface。wLength为0x1A,表示该命令后跟的参数长度为26字节。

视频传输参数主要包括视频帧的格式索引、帧之间的传输间隔、单帧的最大大小、关键帧帧率、压缩质量等等。驱动根据应用程序指定的视频格式来设置传输参数，但参数和参数之间有互相依赖关系，硬件设备也可能无法满足驱动设置的传输参数，因此，在设置传输参数过程中，需要经过以下几个步骤:首先驱动将需要设置的传输参数以probe的方式向硬件发送，probe的方式是一种试探性的方式，因此，硬件可以对驱动发过来的传输参数进行协商，如果硬件对某些参数不能满足，则硬件将其修改为其认为的合理的值，驱动然后获得经过协商过的传输参数，如果认为可以接受，则通过commit的方式向硬件发送传输参数， 确认并提交协商过的传输参数。

Libusb 中提供了两种方式来发送控制命令，即同步和异步的方式， 通过 libusb_control_transfer 函数可以以同步的方式来发送命令，函数原型如下：

```
int libusb_control_transfer (libusb_device_handle *handle,
                             uint8_t          bmRequestType,
                             uint8_t          bRequest,
                             uint8_t          wValue,
                             uint8_t          wIndex,
                             unsigned char *   data,
                             uint16_t          length,
                             unsigned int     timeout
                            )
```

handle 为该设备的操作句柄，bmRequestType、bRequest、wValue、wIndex 即为表 5.1 中的 USB 控制命令的域名对应的值，data 指针为命令附带参数的地址， length 为附带参数的长度。该函数会阻塞直到命令发送并执行完毕，如果成功则会返回实际的发送的字节数。

5.1.4 USB 视频数据等时传输

等时传输有着容错性好，带宽稳定、实时性较强的特点，USB 视频设备中数据传输一般也采用等时传输，libusb 中的 libusb_ transfer 可以用来实现等时传输，具体过程如下：首先通过 libusb_alloc_transfer 函数生成 libusb_ transfer 结构，同时为其申请指定数量的等时传输包。接着通过 libusb_fill_iso_transfer 为 libusb_ transfer 申请传输内存，并将该内存分配给 libusb_ transfer 中各个等时传输包，并将 USB 传输端点绑定到 libusb_ transfer。最后通过 libusb_submit_transfer 提交 libusb_ transfer。由于等时传输是一种异步传输，因此提交完该传输后，无法直接获得传输后的数据，需要在传输的完成回调函数中进行进一步的数据处理，处理完之后，再重新提交该传输，进行下一次的等时传输，具体流程如图 5.1。

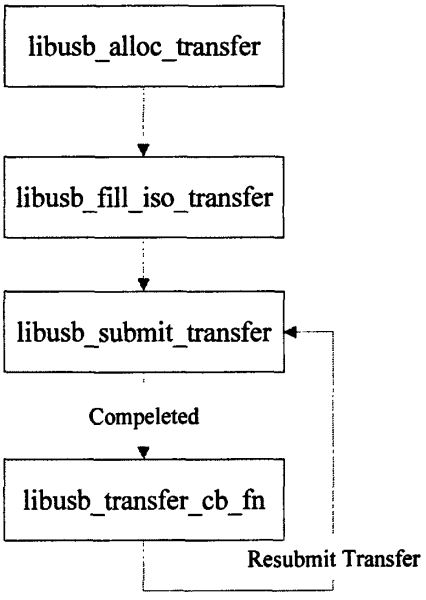


图 5.1 等时传输函数调用流程

传输完成后，获得的视频数据便位于 libusb_ transfer 的传输内存中，实际传输过程中，一个等时传输包为一个独立的单位，因此实际获得的视频数据分布在各个等时传输包中。等时传输包中的数据有两部分构成，头信息和视频数据。对于比较常见无压缩的视频数据传输方式，头信息有 12 个字节构成，第一字节表示了头信息的长度，第二个字节 BFH 构成如表 5.2:

表 5.2 BFH 字段构成

EOF	ERR	STI	RES	SCR	PTS	EOF	FID

FID 标志位在每帧的开始都会切换，并保持不变直到新的一帧开始，EOF 被置 1 则表示达到了该帧的末尾，在传输完成回调函数中进行视频帧的重组会参考两个标志位。libusb 中 iso_packet_desc 结构的 actual_length 字段表示了一个等时传输包中实际数据的长度，因此包中视频数据的实际长度为 actual_lenght-12。视频帧的重组过程如图 5.2:

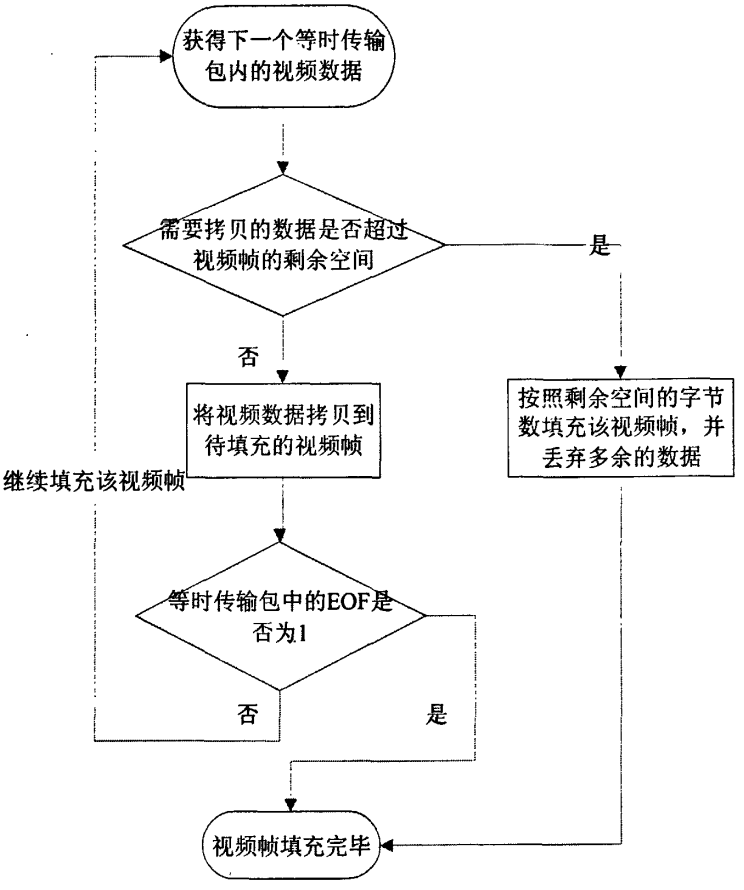


图 5.2 视频帧填充过程

首先从传输缓冲区中取出一个待填充的视频帧，并将等时传输包内的视频数据拷贝到该视频帧，如果视频帧中的剩余空间不足，则按照剩余的空间进行拷贝，并将该视频帧放到完成队列中。否则，拷贝完数据后检查等时传输包中的EOF位是否被置1，如果是的话表示该帧已经结束，同样将其放入完成队列，否则继续从下一个等时传输包拷贝数据。

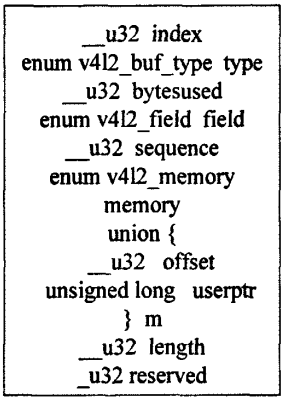
5.2 传输缓冲区的实现

传输缓冲区的有若干个视频帧构成，视频帧可以分为两类，一类为空白的视频帧，另一类为完成填充的视频帧。传输缓冲区模块接收USB设备传输模块的视频数据，将其填充的空白的视频帧中，并接受应用程序交互模块的命令，取出完成的视频帧返回给应用程序，或将一个空白的视频帧放入。

5.2.1 基于v4l2_buffer 的缓冲队列

传输缓冲区在逻辑上由等待队列和完成队列构成，两者都是先进先出的队列，在Linux内核中，对队列的管理可以通过list_head数据结构和相关的函数实现，在用户空间内，则可以用一种更简洁和熟悉的结构来管理队列，即C++标准模版库中的list结构。

传输缓冲区实质上是动态申请的一块内存区域，对这块内存区域进行管理，并将其与具体的视频帧进行关联、可以利用V4L2提供的v4l2_buffer的数据结构， v4l2_buffer结构中主要域如下：



一个视频帧可以对应一个v4l2_buffer结构,index 表示该视频帧的索引号，bytesused表示该视频帧中已被使用了多少字节的数据，m.offset表示该视频帧的内存位置偏移量，length表示视频帧所占内存的大小。

因此在申请传输缓冲区内存时，整个内存区域的大小应该是单个视频帧的大大小乘以视频帧的个数,缓冲区内的视频帧个数由应用程序通过VIDIOC_REQBUFS命令来指定。申请完内存区域后，对视频帧数组进行初始化，index设置为数组在中的序号，length为视频帧的内存区域大小，m.offset=index*length，通过上述操作，申请的传输缓冲区便被映射到各个视频帧中。不过v4l2_buffer结构中的并没有表示视频帧状态的域，为了记录视频帧的状态，可以直接利用v4l2_buffer中的保留字段reserved来存放视频帧的状态。

传输缓冲区中的等待队列和完成队列的定义分别为 list<v4l2_buffe*r> wait_list 和 list<v4l2_buffer*> done_list，一开始，所有的视频帧都是未填充的，因此都被加到wait_list中，经过视频数据填充和应用程序的读取，视频

帧不断在两个队列之间移动。而对队列的操作，在STL模板库的支持下也是方便和高效的。

5.2.2 基于自旋锁与互斥锁的同步访问机制

无论是等待队列还是完成队列，都会出现并发访问的情况，当然两者的情形不同，等待队列的并发访问出现在不同等时传输的完成函数中对等待队列的竞争访问，而对于完成队列，是一个比较典型的生产消费者模型，即驱动程序将填充完的视频帧放入该队列，而应用程序通过调用从该队列取得视频帧。4.5.2节中分析了两种队列的加锁机制，采取了对等待队列加自旋锁而完成队列加互斥锁的策略。

自旋转锁是一种忙等待，Linux内核中大量使用了自旋锁，内核中的自旋锁往往配合关中断和开中断使用[27]，在获得自旋锁并进入临界区时关闭中断，在释放自旋锁时打开中断，这样能够保证临界区的代码不可抢占性，从而防止死锁的发生，同时也降低了临界区执行时间。在用户空间可以有几种方式实现忙等待的加锁机制，如Peterson算法、汇编指令等，不过前者只能解决两个进程之间的互斥访问，后者比较底层并需要硬件的支持，而Linux下的POSIX线程标准库pthread则提供了在用户空间使用自旋锁的快捷方法。pthread的pthread_spinlock_t提供了自旋锁的功能，而pthread_spin_init()、pthread_spin_lock()、pthread_spin_unlock()、pthread_spin_destroy()四个函数分别提供了自旋锁的初始化、上锁、解锁、和销毁的功能[28]。在进入临界区的执行代码如下：

```
……
pthread_spin_lock(&wait_list_lock);
if (!wait_list.empty())
    video_buf=wait_list.front();
 else
    video_buf=NULL;
 pthread_spin_unlock(&wait_list_lock);
……
```

临界区的执行代码只有4行，执行速度比较快，为了精简临界区代码，可以

将else video_buf=NULL;移到临界区前。

对于完成的队列的访问， 还有一种比较特殊的情况，即应用程序采取阻塞读的方式，当驱动收到应用程序进行阻塞读的命令，就对完成队列加上互斥锁，然后试图从完成队列中取出一个视频帧，如果不能获得，则一直等待，直到完成队列中有可用的视频帧或等到超时。这时，如果有一个在等待队列里的视频帧填充完毕，将要被添加到完成队列，试图对完成队列进行访问时，发现队列已被锁住，这种情形下，获得互斥锁的线程无法读取数据，但试图添加数据的线程却无法获得互斥锁，死锁便发生了。对于这种情况，一种简单的方法是读取数据的线程主动放弃该互斥锁，然后重新加锁来尝试读取，但这种方式不仅开销大，也不能完全保证写数据的线程在与读数据的线程竞争中能够获得该互斥锁。信号量则对这种情况提供了简便的解决方法，信号量和互斥锁类似，如果竞争不上，就会进入睡眠，发生进程调度，Linux 中<semaphore.h>中包含了对信号量 sem_t 的定义，sem_wait 函数对信号量减 1，当信号量为 0 时阻塞，sem_post 则对信号量加 1，sem_init 函数可以初始化信号量的值。对于阻塞读，执行流程如图 5.3。

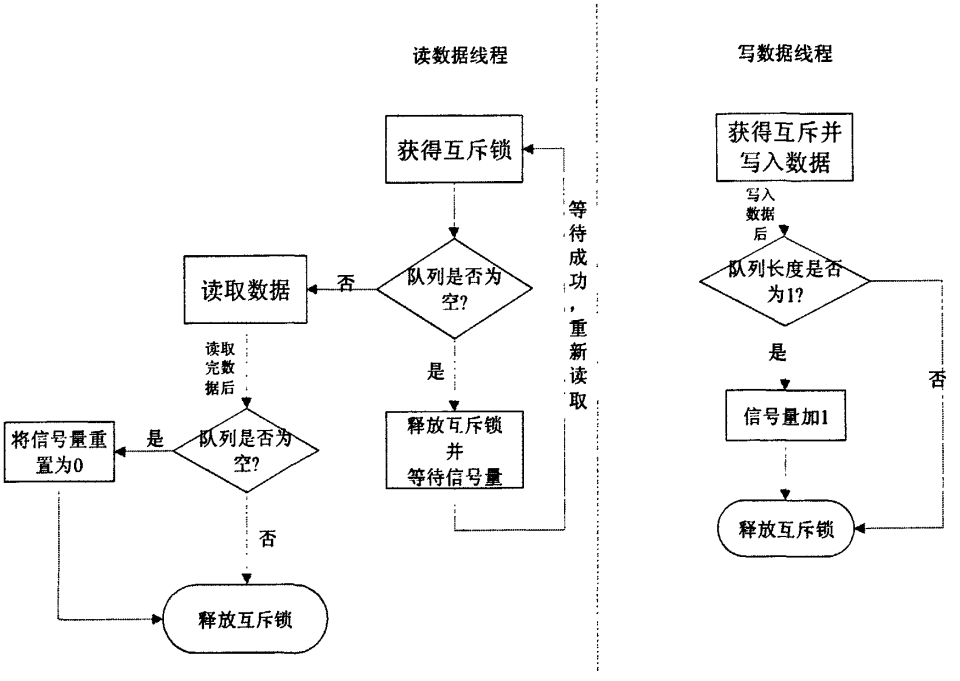


图 5.3 数据读写同步方法

5.2.3 共享内存的传输方式

在内核驱动中，当驱动与应用程序之间需要传输大量数据时，往往不会采取read、write的方式，而是采取内存映射的方式，应用程序通过mmap系统调用将驱动中的内存数据区映射到用户空间，直接读写驱动的内存区域。在用户空间中，可以采取一种更为方便的数据传输方式，即共享内存。

当应用程序调用VIDIOC_REQBUFS命令宏申请传输缓冲区时，驱动程序先通过ftok函数生成一个共享内存的标志符，再通过shmget函数申请共享内存，并将该共享内存作为传输缓冲区，分配给视频队列。当应用程序向驱动请求视频数据时，执行流程如图5.4：

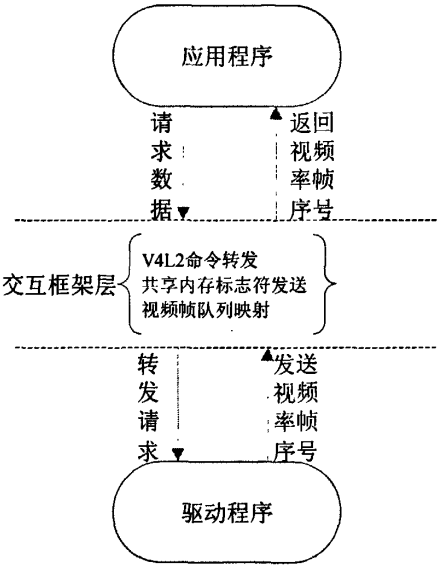


图 5.4 视频数据传输流程

由于共享内存的存在,应用程序和驱动之间的视频传输只需要传输对应视频帧的序列号。具体实现机制如下，在驱动程序申请完共享的传输缓冲区并分配给相应的驱动内的视频队列后，应用程序通过VIDIOC_QUERYBUF命令宏，获得驱动中视频队列每个视频帧的序号、在共享内存中的起始位置、数据区的长度等，并在应用程序中生成一个v4l2_buffer队列来保存驱动中视频队列信息。如此一来，当应用程序请求视频数据时，首先通过交互框架转发请求命令，驱动收到相应命令后，从完成队列中取出一个视频帧，并将该视频帧的序号返回，当交互框架将该帧序号的转发给应用程序后，应用程序由于已经映射了驱动中的视频队

列，因此根据帧的序号即可获得该帧在共享内存中的起始位置，从而进行数据的读取。

5.2.4 可重用传输缓冲区的结构设计

在用户空间开发驱动的一个好处是可以用C++语言以面向对象的思想来实现和封装一个结构，这样既可以提高代码的重用性和降低代码维护的复杂度，和增加代码结构的可读性。传输缓冲区作为一个整体结构，被封装成一个videoqueue对象，UML结构如图5.5。

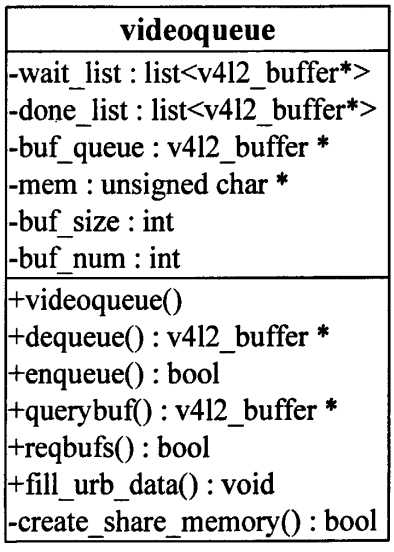


图 5.5 传输缓冲区对象UML图

该对象中的 wait_list 和 done_list 分别表示了等待队列和完成队列，buf_queue 代表了整个视频队列数组，mem 表示了该缓冲区的共享内存，buf_size 和 buf_num 分别表示了视频帧数据区的大小和总的视频帧个数。

dequeue、enqueue 分别表示了缓冲区的出队和入队方法，出队从 done_list 中取出一个完整的视频帧，而入队则将一个视频帧放入 wait_list 等待填充。reqbufs 函数调用 create_share_memory 为缓冲区申请共享内存和初始化视频队列，fill_urb_data 则在等时传输完成回调函数中被调用，负责将等时传输中的视频数据填充到合适的视频帧。

videoqueue 对象代表了一个缓冲区对象，其中的操作和数据结构等都是硬件

无关的，并提供了良好的并发访问处理机制，因此有着较好的重用性。

5.3 应用程序交互模块的实现

应用程序交互模块的主要作用是接收应用程序的请求，并将其转发给相应的模块处理，最后向应用程序返回数据。应用程序交互模块是直接依赖与交换框架的，没有交换框架的支持，应用程序交互模块无法获得应用程序的请求，同时也无法发送数据。对应USB视频设备驱动来说，应用程序交互模块主要接受的命令为应用程序通过ioctl发送的V4L2命令宏，基本的V4L2命令宏如表4.1表示。有了交互框架的支持，应用程序交互模块的实现也比较简单，只需要实现一个ioctl 回调函数即可，在 ioctl 函数中对命令进行解析并发送到对应模处理。ioctl回调函数原型如下：

```
int ioctl(file *file, int cmd, void * args)
```

file 指针暂时不会用到，cmd 表示命令类型，args 表示命令参数。命令分两大类，与 USB 硬件控制相关的 V42L 命令有 VIDIOC_S_FMT、VIDIOC_G_FMT、VIDIOC_S_PARAM 、VIDIOC_S_INPUT 等等，和传输缓冲区相关的有 VIDIOC_REQBUFS 、VIDIOC_QUERYBUF、VIDIOC_QBUF、VIDIOC_DQBUF 等等。根据命令类型不同，ioctl 分别调用USB传输模块或传输缓冲区模块。Args 参数类型和具体命令相关，如果是 VIDIOC_QBUF、VIDIOC_DQBUF 命令，则 args 类型为 v4l2_buffer，若是 VIDIOC_REQBUFS，args 则为 v4l2_requestbuffers， V4L2 的标准中规定了命令和参数类型的对应关系。

实际使用过程中，需要将驱动中的ioctl回调函数向交换框架进行注册，这样ioctl回调函数才能接受到应用程序的命令请求。

5.4 本章小结

本章首先分析了USB视频设备驱动在内核空间和用户空间中实现的不同，对户空空间对USB设备进行访问的Libusb库进行了介绍及分析。分析了通过Libusb来控制USB设备，传输视频数据的方法。重点分析了驱动传输缓冲区的实现，结合V4L2框架从可扩展和可重用的角度实现了传输缓冲区的代码，使得其可以被其他视频设备驱动所重用。

第6章 USB视频用户空间驱动的测试

本章中对用户空间USB视频设备驱动和交互框架进行了测试，并与传统的内核驱动进行了分析对比。

测试软件为Luvcview，Luvcview为Linux下一款主要针对web camera的开源播放软件，并支持V4L2标准。USB视频设备为Chicony摄像头。Luvcview使用标准系统调用来打开视频设备和获取视频数据，主要为open、ioctl、close等。V4L2交互框架提供了类似的接口来让应用程序访问用户空间驱动并获得数据，主要为u_open、u_ioctl、u_close三个调用接口，和原有的系统调用除了在函数名上有区别，在参数类型和使用方法上是完全一样的，因此只要对Luvcview源代码稍做修改便能使其从用户空间驱动中获取数据并正常播放视频。

对Luvcview中对视频驱动进行访问的源代码主要集中在v4l2uvc.c文件中，修改过程如下：

- 在v4l2uvc.c文件中添加包含交互框架的头文件“syscall.h”
- 将v4l2uvc.c文件中打开视频设备的系统调用open改为u_open
- 将v4l2uvc.c文件中所有的ioctl系统调用该为u_ioctl
- 在Luvcview的makefile文件中的OBJECTS一行加入syscall.o,即交互框架的源代码目标文件

在交互框架支持下，对源代码的修改也是相待方便的，基本上是简单的字符串替换，而且新的接口保留了原有系统调用的作用的含义。修改完后，重新make，输入 .\luvcview -d /tmp/video0 -f yuv 启动luvcview播放器，视频设备文件为/tmp/video0由V4L2交互框架自动生成。Luvcview能够正确运行，并流畅的播放视频输入。

同时，表6.1对比了Luvcview采用默认的播放参数时，分别使用内核驱动和用户间驱动，获得100个视频帧所需要的时间。

表 6.1 获取视频帧时间对比

	视频帧数	平均时间	平均 FPS
内核空间驱动	100	6. 94s	14
用户空间驱动	100	6. 95s	14

从表6.1中可以看出，针对同一硬件，内核空间驱动和用户空间驱动在视频数据采集和传输的能力基本持平。

而在CPU使用率上，经过修改过的Luvcview访问用户空间驱动时比未经修改的Luvcview访问内核空间驱动高出1个百分点，高出的CPU使用率主要为V4L2交互框架的运行开销，而在内存使用率上，两者基本持平。

表 6.2 Luvcview资源使用对比

	原始的Luvcview	修改后的Luvcview
CPU平均使用率	2%	3%
内存平均使用率	0.2%	0.2%

用户空间驱动运行时CPU占用率为2%，由于内核驱动的只是作为一个模块运行，因此无法直接查看其运行时CPU占率，无法直接对两者进行对比，不过2%的占用率对于一个工作机制相对复杂的驱动来说，在一个合理的范围内。

第7章 总结与展望

本文对Linux用户空间驱动开发做了研究，针对USB视频设备用户空间驱动的开发进行了分析与实现，相对与传统的内核驱动开发，在用户空间中，使用Libusb库进行驱动开发能够有效的降低代码量，在内核驱动中对USB的探测函数，至少需要四五十行的代码，但通过Libusb，实现同样的功能只需10行左右代码。

7.1 总结

本文首先分析了Linux下设备驱动开发方法的存在的不足和发展趋势，并分析了对比了几种驱动开发方法，重点分析了用户空间下进行驱动开发的优势，并指出了用户空间驱动存在的不足，针对这些不足，结合视频设备，提出了V4L2交互框架，该框架不仅可以实现应用程序和用户空间驱动之间基于V4L2标准的命令与数据交互，并且最大程度上保留了原有的应用程序与内核驱动之间的交互方式，使的应用程序可以方便的调用用户空间驱动。在此基础上，针对视频设备驱动数据传输的方式，该框架自动实现了驱动和应用程序之间的内存映射。通过该框架的支持，用户空间驱动可以很大程度减少开发的工作量。

接着，针对USB视频设备，从USB视频设备的共性出发，提出了用户空间下驱动开发的设计方案与实现，并重点分析对比了用户空间和内核空间中实现技术的不同，重点分析了驱动传输缓冲区的高效的组织与管理策略和缓冲区的访问同步机制，并使用C++面向对象的思想，将传输缓冲区实现代码进行抽象与封装，使的其可以方便的被同类驱动重用。

最后，针对web camera，通过对开源播放器Luvcview的改造，对用户空间下的USB视频设备驱动进行了测试，并分析了与内核驱动在性能和效率的差别。

7.2 展望

首先，第三章提出的V4L2交互框架，主要是支持用户空间下的视频设备驱动，对于其他类型设备驱动的支持，该交互框架还需要进一步的扩展。

其次，本文提出的USB视频设备用户空间驱动，和内核驱动相比，主要不足

在以下两方面：对DMA操作的支持，和驱动热插拔动态加载功能，这也是用户空间驱动一个需要解决的一个问题。针对USB视频设备用户空间驱动，本文提出了可重用的传输缓冲区结构，但对于USB视频设备整个驱动的开发，可以封装成一个二次开发库，方便其他有特定需求的视频设备驱动的开发。

最后，由于视频设备的独占性，本文中未对多应用程序同时访问驱动的同步问题进行研究，但对于其他设备，需要在驱动实现中考虑多进程并发访问的处理。

参考文献

[1] 颜跃进，秦莹,孔金珠，戴华东,邵立松.操作系统设备驱动可靠性研究综述[J].计算机工程与科学,2009,31(5)：121-124

[2] Yoann Padioleau, Julia L. Lawall, Gilles Muller. Understanding Collateral Evolution in Linux Device Drivers.ACM, 2006

[3]Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.Linux Device Drivers, Third Edition. United States of America: O' Reilly Media, 2005

[4]袁丽慧，彭磊.可重用Linux设备驱动程序框架[J].计算机工程,2008,34(5)：89-91

[5] Linux-2.6.26\Documentation\DocBook

[6]Greg Kroah-Hartman.Driving Me Nuts snooping the USB Data Stream[J]. August 2004 Linux Journal，Volume 2004 Issue 124

[7] Young Jun Cho, Yun Chan Cho and Jae Wook Jeon.D-Bus based user device driver framework design for Linux mobile software platform[C]. Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on,5-8 July 2009 :426 - 431

[8] 丁晓波，桑楠，张宁. Linux 2. 6内核的内核对象机制分析[J]. 计算机应用,2005,25(1):76-77

[9]宋宝华.Linux设备驱动开发详解.北京：人民邮电出版社，2008.2

[10]Universal Serial Bus Device Class Definition for Video Devices

[11]刘宏伟,郑立云.USB2.0 数据传输类型分析[J].计算机工程与设计,2005,26(7):1823-1826

[12]施剑，何成林，杜利民. 基于USB2. 0的麦克风阵列语音数据采集系统设计[J]. 计算机工程,2006,32(24):216-218

[13] LibUSB. http://libusb.sourceforge.net/

[14] Michael H Schimek, Bill Dirks, HansVerkuli.Video for Linux Two API Specification [M].http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf. 2008

[15]李晋，葛敬国.Linux下互斥机制及其分析[J]. 计算机应用研究，2005，

22(8):72-74, 77

[16] 王乐，张晓彤，李磊，樊勇。Linux下的DDR DIMM总线接口设备检测方法[J]. 计算机工程,2007,33(18):256-258

[17] 李革梅,刘福岩. 基于嵌入式Linux系统的设备驱动实现研究[J]. 计算机应用,2008,28:297-298

[18] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov.Linux Netlink as an IP Services Protocol.RFC3549, 2003, 7

[19] Greg Kroah-Hartman.Driving Me Nuts The USB Serial Driver Layer[J]. Linux Journal, 2003, 2003(106):9

[20] 董昱，马鑫. 基于netlink 机制内核空间与用户空间通信的分析[J].测控技术,2007,26(9):57-58, 60

[21] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, Gilles Muller. Devil: an IDL for hardware programming[C]. October 2000 OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4

[22] Christopher L. Conway, Stephen A. Edwards. NDL: A Domain-Specific Language for Device Drivers[C]. July 2004 LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems

[23] Shan Chen, Lingling Zhou, Rendong Ying, Yi Ge. Safe device driver model based on kernel-mode JVM[C]. November 2007 VTDC '07: Proceedings of the 3rd international workshop on Virtualization technology in distributed computing

[24] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler. An empirical study of operating systems errors[C]. December 2001 SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles

[25] Christopher L. Conway, Stephen A. Edwards. NDL: a domain-specific language for device drivers[C]. July 2004 LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for

embedded systems

[26] 章晓明 杜春燕 陆建德. IPsec VPN 中 Netlink 消息通信机制的研究和设计[J]. 计算机工程与应用,2006,42(34):139-141

[27] Robert Love. Kernel korner: Kernel locking techniques[J]. August 2002 Linux Journal, Volume 2002 Issue 100

[28] Hans-J. Boehm . Reordering constraints for pthread-style locks[C]. March 2007 PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming

[29] 张永生，谭成翔，汪海航. 基于 Linux 的主机间 USB 通信的实现[J]. 计算机应用研究，2007，24 (8)：306-308

[30] 杨伟，刘强，顾新. Linux 下 USB 设备驱动研究与开发. 计算机工程[J]. 2006，32(19):283-284

[31] 孙劲飞,戎蒙恬,刘文江. KGDB 在基于 ARM Linux 的嵌入式系统中的应用[J]. 计算机应用与软件，2008,25(6):231-232

作者简历

徐家,男，1985 年 7 月出生与江苏无锡，2003 年 9 月 2007 年 6 月就读与浙江大学计算机科学与技术学院，获学士学位。2007 年 9 月至今就读于浙江大学计算机学院攻读硕士学位。