电子科技大学

硕士学位论文

嵌入式Linux设备驱动程序的设计与研究

姓名：程科

申请学位级别：硕士

专业：软件工程

指导教师：李毅

20070423

摘 要

在数字信息技术和网络技术高速发展的后PC时代，嵌入式系统因其体积小、可靠性高、功能强、灵活方便等许多优点，已渗透到工业、农业、教育、国防、科研以及日常生活等各个领域，对各行各业的技术改造、产品更新换代、加速自动化进程、提高生产率等方面起到了极其重要的推动作用。同时，嵌入式 Linux 操作系统以其开放源代码、易于开发、功能强大、稳定、成本低等优势迅速跻身于主流嵌入式开发平台。基于嵌入式 Linux 操作系统的研究和应用具有巨大的学术和商业价值。在嵌入式 Linux 系统的开发中，嵌入式设备种类繁多的特点决定了不同的嵌入式产品在开发时都必须设计和开发自己的设备驱动程序，使得嵌入式 Linux 设备驱动程序的开发在整个嵌入式系统开发工作中占有举足轻重的地位。

本文首先详细介绍了嵌入式 Linux 系统的体系结构以及嵌入式 Linux 系统设计的基本步骤，并阐述了在基于 S3C2410 芯片的 FS2410 开发板上构建了一个可实用的嵌入式 Linux 系统的全过程；然后分析了 Linux 内核的组成和工作机制、内核与设备驱动之间的关系，Linux 设备驱动的结构，并总结了 Linux 设备驱动的设计和实现步骤；再概述了 USB 总线协议的拓扑结构、通信流模型、数据传输流程等问题，实现了基于 ZC301P 芯片的 USB 摄像头的嵌入式 Linux 驱动程序。最后概述了 PCI 总线协议的体系结构、总线信号、总线配置等问题，实现了流媒体数据缓存 PCI 卡的 Linux 设备驱动程序。

在硕士课题的研究工作中，作者的主要工作成果是：

1. 研究和分析了项目开发中 USB 摄像头和 USB 鼠标数据传输带宽冲突所造成的花屏、死机的问题和原因。从减小冲突的影响和降低传输带宽需求这两个方面出发，提出并实现了三种解决 USB 传输带宽冲突问题的方法。实验证明，综合使用这三种方法就能够有效地解决带宽冲突带来的花屏、死机问题。并且，据此发表核心文章一篇。

2. 总结了 Linux 设备驱动程序的设计和实现步骤，并实现了基于 ZC301P 芯片的 USB 摄像头和流媒体数据缓存 PCI 卡的 Linux 设备驱动程序。

关键词：嵌入式 Linux，设备驱动，USB，PCI，研究

Abstract

In Post-PC era when the digital information technology and network technology are developing at a high speed, the embedded system has many merits of small bulk, high solidity, powerful function, etc, so it has permeated into industry, agriculture, education, national defence, scientific research, and daily life, etc. The embedded system is the very important power to impulse the technology innovation of all kinds of industries, the evolution of the product, the acceleration of the automatization process, the advance of productivity, etc. At the same time, embedded Linux operating system has rapidly become the popular embedded development platform, because it has many merits of open source, easy development, powerful function, stabilization, low cost, etc. The research and development based on embedded Linux operating system have the much value of science and commerce. In the development of the embedded Linux system, we must design and develop our device driver in the development of many embedded product because of many kinds of the embedded device, so the development of the embedded Linux device driver is very important job in the development of the whole embedded system.

This thesis, firstly, introduces the embedded Linux system architecture and the basic steps of the design of embedded Linux system, and illuminates the entire process of the development of the real embedded Linux system based on FS2410 development board; secondly, analyzes the composition and work principle of Linux kernel, the relationship between the kernel and device driver and the architecture of Linux device driver, and summarizes the design and development steps of the Linux device driver; thirdly, introduces the USB bus topology architecture, USB communications model, USB data transmission flow, etc, implements the USB camera device driver based on ZC301p chip, and analyzes and solves the conflict of USB bandwidth; finally, introduces PCI bus architecture, PCI bus signals, PCI bus configuration problems, and implements the streaming media data cache PCI card device driver.

In the research work of the master thesis, the main contribution of the author are:

1. Investigates and analyzes the reason of the USB camera and USB mouse's data transfers bandwidth collision. The collision destroys the image data which is transferred

from the USB camera and causes the system breakdown. Gives and implements three methods about weakening the influence of the collision and reducing the request of the transfers bandwidth to deal with the problem. The result of the experimentation proves the integration of three methods can effectively solve the bandwidth collision. And author publishes a core paper based on this.

2. Summarizes the design and development steps of the Linux device driver, and implements the USB camera device driver based on ZC301p chip and the streaming media data cache PCI card device driver.

Keyword: Embedded Linux, device driver, USB, PCI, Research

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：程科 日期：2007年4月24日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文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、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：程科 导师签名：李毅

日期：2007年4月24日

第一章 绪论

1.1 研究背景

1.1.1 嵌入式系统

嵌入式系统（Embedded System）可以定义为：以应用为中心、以计算机技术为基础、软硬件可裁剪、适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统[1]。从上述定义可以看出嵌入式系统是针对特定应用的软硬件综合体，其一般具备以下几方面的特征：

- 嵌入式系统通常是面对用户、面向产品、面向特定应用的；
- 嵌入式系统是先进的计算机技术、半导体技术以及电子技术与各个行业的具体应用相结合的产物；
- 嵌入式系统必须根据应用需求对软硬件进行裁剪，满足应用系统的功能、可靠性、成本、体积等要求；
- 嵌入式系统开发需要专门的开发工具和环境；
- 为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在储存器芯片或单片机中；

1.1.2 Linux 操作系统

Linux 操作系统核心最早是由芬兰的 Linus Torvalds 于 1991 年 8 月在芬兰赫尔辛基大学上学时发布的（Linux 0.11 版），后来经过众多世界顶尖的软件工程师的不断修改和完善，Linux 得以在全球普及开来，在服务器领域及个人桌面领域得到越来越多的应用[2]。

Linux 是在 GNU 公共许可权限下免费获得的，是一款符合 POSIX 标准的多用户、多任务、支持多线程和多 CPU 的类 Unix 操作系统。Linux 以其高效性和灵活性著称。Linux 模块化的设计结构，使得它能够在价格昂贵的工作站上运行，也能够在廉价的 PC 机上实现全部的 Unix 特性。Linux 当前有很多发行版本，较流行的有：RedHat Linux、Debian Linux、RedFlag Linux 等[3]。

1.1.3 Linux 作为嵌入式操作系统的优势

在“后 PC”时代，由于集成电路技术的飞速发展，嵌入式系统的开发从单片机时代进入到了一个“系统”开发的阶段，嵌入式操作系统也逐渐走上了历史舞台。从国内和国外来看，嵌入式操作系统主要有 Windows CE、VxWorks、pSOS、Palm OS 等。Linux 是一个成熟、稳定的操作系统，由于其在嵌入式开发方面具有其它操作系统无可比拟的优势，经过这几年的发展，已迅速跻身主流嵌入式开发平台，并以每年 100%的用户递增数量显示了其强大的力量。Linux 作为嵌入式操作系统的优势在于[4]：

- Linux 是开放源代码的免费软件。只要遵守 GPL 的规定，就可以免费获得 Linux 内核和其它自由软件的源代码，采用 Linux 操作系统构建嵌入式系统，可以大大降低开发成本和周期。此外，由于拥有源代码，开发人员可以针对特定的应用来修改 Linux 内核和软件的源代码，更好地进行开发工作。
- Linux 具有完善的文档和广泛的技术支持。Linux 是互联网充分发展的产物，在网上能够找到许多关于 Linux 的文档以及强大的技术支持。
- Linux 内核功能强大，性能高效、稳定。Linux 的内核非常稳定，它的高效和稳定性已经在各个领域，尤其是在网络服务器领域得到了事实的验证。
- Linux 能够支持多种体系结构，是支持微处理器种类最多的操作系统。目前，Linux 已经被移植到数十种硬件平台上，几乎所有主流的硬件平台，如 X86、ARM、PPC、MIPS、ALPHA、SPARC 等，Linux 都支持。
- Linux 拥有强大的网络功能。随着嵌入式系统的发展，嵌入式系统与 Internet 结合得越来越紧密。与其它操作系统相比，Linux 在网络方面有较大的优势，基本上所有的网络协议和网络接口都可以在 Linux 上找到。
- Linux 大小和功能可定制。Linux 继承了 Unix 的优秀设计思想，内核与用户界面完全独立，各部分的可定制性很强，可以按照需求进行定制和配置，这对于硬件资源有限的嵌入式系统是一个理想的选择。

1.2 研究现状

如前所述，嵌入式 Linux 操作系统非常适合用于构建嵌入式系统，但由于

是一种通用操作系统，而不是一个真正的实时操作系统，内核不支持事件优先级和抢占实时特性(2003年底推出的Linux 2.6内核实现了一定程度上的可抢占性)，因此，也存在着如硬实时性、体积庞大等问题。

由于Linux具有免费开源的特性和嵌入式Linux广阔的市场前景，针对上述问题的研究具有巨大的学术和商业价值，国内外不少大学、研究机构和公司都纷纷加入到嵌入式Linux的研究开发当中，目前国际上对嵌入式Linux的研究开发主要集中在以下几个方面[5]：

- 实时性：在数据采集、控制、音/视频等设备中，对操作系统的实时性有比较高的要求。Linux并不是一个实时操作系统，因而必须提高实时性以满足这些设备的要求。
- 内核裁减：嵌入式设备资源有限，对软件的体积有比较苛刻的要求。由于Linux是单一模块结构，体积较大，不适合直接在嵌入式设备中应用。许多厂商致力于开发符合原Linux接口标准的小体积的Linux内核，并加强其可裁减性和可配置性。
- 集成开发环境：提供完整的集成开发环境是每一个嵌入式系统开发人员所期待的。一个完整的嵌入式系统的集成开发环境一般由编译器、连接器、内核调试/跟踪器和集成图形界面开发平台组成。目前嵌入式Linux还没有比较完善的集成开发环境，特别是基于图形界面的特定系统定制平台的研究上，与Windows操作系统相比还存在差距，因此，要使嵌入式Linux在嵌入式操作系统领域中的优势更加明显，整体集成开发环境还有待提高和完善。

1.3 课题研究的意义和内容

从国家战略目标与产业发展来看，嵌入式软件特别是嵌入式操作系统是实现传统制造业转型与提升的关键技术，它对整体提升我国制造业的竞争能力、大幅度地增强我国软件自主创新能力意义重大。但是我国当前嵌入式系统研究和开发的整体水平不高，与国际领先水平还有较大的差距，国内的嵌入式设备生产商大多数还是采用国外的商用嵌入式操作系统。而免费开源的Linux在嵌入式领域的发展为我国发展自己的嵌入式操作系统，扭转PC机软件市场的被动局面提供了难得的机遇，也为振兴国内软件行业找到了最佳的突破口。

设备驱动程序在Linux内核中扮演着特殊的角色，它是进入Linux内核世界

的大门。对嵌入式 Linux 设备驱动的研究，有助于深入理解嵌入式 Linux 内核代码。此外，硬件必须有配套的驱动程序才能正常工作，由于嵌入式设备的种类繁多特点决定了不同的嵌入式产品在开发时都必须设计自己的设备驱动程序，使得设备驱动程序的开发在整个嵌入式系统开发工作中占有举足轻重的地位。因此，对嵌入式 Linux 设备驱动的研究是一个很好的课题，具有重要的社会和商业价值。

当前嵌入式 Linux 设备驱动的研究工作主要集中在对特定设备的驱动程序实现和提升设备驱动性能方面，对嵌入式 Linux 设备驱动程序的设计和实现的基本步骤总结的比较少。本课题的研究内容和工作主要包括：

1. 总结了 Linux 设备驱动程序的设计和实现步骤，并实现了基于 ZC301P 芯片的 USB 摄像头和流媒体数据缓存 PCI 卡的 Linux 设备驱动程序。

2. 研究和分析了项目开发中 USB 摄像头和 USB 鼠标数据传输带宽冲突所造成的花屏、死机的问题和原因。从减小冲突的影响和降低传输带宽需求这两个方面出发，提出并实现了三种解决 USB 传输带宽冲突问题的方法。

1.4 论文的结构

全文共分五章，具体的内容组织如下：

第一章：绪论。概述了课题的背景以及研究状况，提出了课题的研究意义，介绍了本文的主要研究工作。

第二章：嵌入式 Linux 内核及设备驱动。介绍了嵌入式 Linux 系统设计的基本步骤，并基于 S3C2410 处理器的 FS2410 开发板上构建了嵌入式 Linux 系统，最后介绍了 Linux 操作系统内核的组成以及 Linux 设备驱动的结构。

第三章：嵌入式 Linux 下 USB 设备驱动的设计与研究。介绍了 USB 总线协议以及 USB 设备的 Linux 驱动框架，设计和编写了基于 ZC301P 芯片的 USB 摄像头的嵌入式 Linux 驱动，最后研究了项目开发中出现 USB 传输带宽冲突的问题，并提出了相应的解决办法。

第四章：嵌入式 Linux 下 PCI 设备驱动的设计与实现。介绍了 PCI 总线协议以及 PCI 设备的 Linux 驱动框架，设计和编写了流媒体数据缓存 PCI 卡的 Linux 设备驱动。

第五章：结束语。总结了本文的主要工作以及进一步的工作展望。

本文最后是参考文献和致谢。

第二章　嵌入式 Linux 内核以及设备驱动

2.1 嵌入式 Linux 系统的体系结构

嵌入式 Linux 系统有两层含义，狭义的嵌入式 Linux 系统指的是嵌入式 Linux 操作系统，广义的嵌入式 Linux 系统指的是基于嵌入式 Linux 操作系统构建的嵌入式系统。如图 2-1 所示，嵌入式系统主要分为两大部分：嵌入式硬件和嵌入式软件。嵌入式硬件部分主要由嵌入式处理器、储存器、I/O 端口和外围设备构成；嵌入式软件部分主要由嵌入式操作系统、设备驱动和嵌入式应用软件构成。

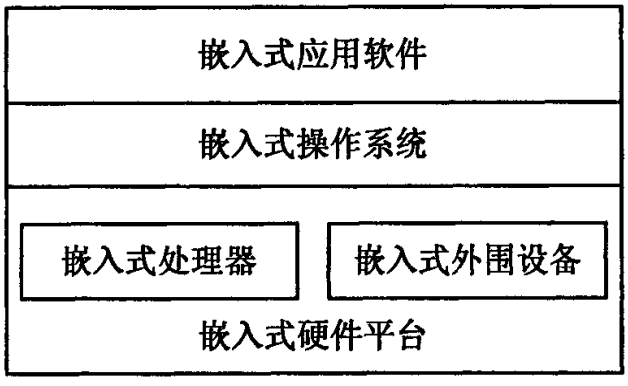

图 2-1 嵌入式系统体系结构

2.1.1 嵌入式处理器

嵌入式系统的核心是各种类型的嵌入式处理器，嵌入式处理器与通用处理器最大的不同点在于，嵌入式处理器大多工作在为特定用户群所专门设计的系统中，它将许多板卡上的接口电路集成到芯片内部，从而有利于嵌入式系统趋于小型化，同时还具有很高的效率和可靠性。嵌入式处理器可以分为以下几类：

- 嵌入式微处理器(Embedded Microprocessor Unit, EMU)。嵌入式微处理器的基础是通用计算机中的 CPU。在应用中，将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的母板功能，这样可以大幅度减小系统体积和功耗。嵌入式微处理器目前主要有 ARM、Power PC、MIPS、Am186/88、386EX、68000 等系列。

- 嵌入式微控制器(Microcontroller Unit, MCU)。嵌入式微控制器一般以某一种微处理器内核为核心，芯片内部集成 ROM/EPROM、RAM、I/O、串口、脉宽调制输出、A/D、D/A、Flash RAM 等各种必要功能和外设。嵌入式微控制器目前主要有 8051、P51XA、MCS-96/196/296、C166/167、MC68HC05/11/12/16 等系列。
- DSP 处理器(Digital Signal Processor, DSP)。DSP 处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法，编译效率较高，指令执行速度也较高，在数字滤波、FFT、频谱分析等方面得到了大量的应用。嵌入式 DSP 处理器比较有代表性的产品是 TI 的 TMS320 系列和 Motorola 的 DSP56000 系列。
- 嵌入式片上系统(System On Chip)。嵌入式片上系统指的是在单个芯片上集成一个完整的系统，所谓完整的系统一般包括中央处理器、存储器、以及外围电路等。通用的 SOC 系列包括 Infineon 的 TriCore，Motorola 的 M-Core，Echelon 和 Motorola 联合研制的 Neuron 芯片等。

本课题采用的是基于 Samsung 公司的 S3C2410X 处理器的 FS2410 开发板。这块处理器是一款基于 ARM920T 内核的 16/32 位 RISC 嵌入式微处理器，运行频率是 203MHz。处理器内部集成的资源主要有：一个 LCD 控制器（支持 STN 和 TFT）、SDRAM 控制器、3 个通道的 UART、4 个通道的 DMA、4 个具有 PWM 功能的计时器和 1 个内部时钟、2 个 USB 主机接口和 1 个 USB 设备接口、2 个 SPI 接口、I2S 总线接口、SD 接口和 MMC 卡接口等。

2.1.2 嵌入式外围硬件设备

嵌入式外围设备是指在一个嵌入硬件系统中，除了中心控制部件(EMU,MCU,DSP,SOC)以外的完成存储、通讯、保护、调试、显示等辅助功能的其它部件。根据外围设备的功能主要可分为以下三类：

- 存储设备类：静态易失型存储器(RAM,SRAM)、动态存储器(DRAM),非易失型存储器(ROM, EPROM, EEPROM, FLASH)。其中，FLASH 以可擦写次数多，存储速度快，容量大，价格便宜等优点在嵌入式领域中都有着广泛的应用。
- 通信设备类：目前存在的大多数通信设备在嵌入式领域中都有着广泛的应用，其中 RS232 接口（串行通信接口）、IrDA 接口（红外线接口）、I2C

总线接口（现场总线接口）、USB 接口（通用串行总线接口）和 Ethernet 接口（以太网接口）应用较广泛。

- 显示设备类：CRT、LCD 和触摸屏等外围显示设备。

本课题采用的外围设备主要有 16M 的 NAND Flash、256M 的 SDRAM 内存、LCD 显示器、USB 摄像头、USB 打印机和 USB 鼠标。

2.1.3 嵌入式操作系统

嵌入式操作系统是一种用途广泛的系统软件，负责嵌入式系统的全部软、硬件资源的分配、调度、控制和协调各部件的工作，嵌入式操作系统与一般操作系统相比，在系统实时高效性、硬件的相关依赖性、软件固化以及应用的专用性等方面具有较为突出的特点。一般情况下，嵌入式操作系统可以分为两类，一类是面向控制、通信等领域的实时操作系统，如 VxWorks、pSOS、QNX 等；另一类是面向个人数字助理(PDA)、移动电话、机顶盒等消费电子产品的非实时操作系统，如 Windows CE、嵌入式 Linux、Palm OS 等。

本课题采用的是 ARM Linux 操作系统，ARM Linux 是指成功的移植在 ARM 体系结构上的嵌入式 Linux 操作系统内核。

2.1.4 设备驱动

在 Linux 内核中，设备驱动程序是一个个独立的“黑盒子”，使某个特定硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。用户的操作通过一组标准化的调用执行，设备驱动负责将这些调用映射到作用于实际硬件设备特有的操作上。

2.1.5 嵌入式应用软件

嵌入式应用软件是针对特定应用领域，基于某一固定的硬件平台，用来达到用户预期目标的计算机软件。嵌入式应用软件和普通应用软件有一定的区别，它不仅要求其准确性、安全性和稳定性等方面能够满足实际应用的需要，而且还要尽可能地进行优化，以减少对系统资源的消耗，降低硬件成本。

2.2 嵌入式 Linux 系统的设计与实现

嵌入式系统的开发涉及两个方面：硬件部分与软件部分。硬件部分提供整个系统开发可见的或可触摸的“实体”，而软件部分则提供这个“实体”内部的功能逻辑，本文的研究工作主要集中在嵌入式 Linux 系统的软件开发上面。嵌入式 Linux 系统从软件的角度看通常可以分为四个层次[6]：

- 引导加载程序。包括固化在固件（firmware）中的启动代码（可选）和 Bootloader（引导加载程序）两大部分。
- 嵌入式 Linux 内核。包括特定于嵌入式开发板的定制内核以及控制内核引导系统的参数。
- 文件系统。包括根文件系统（RamDisk）和建立于 Flash 之上的文件系统。
- 用户应用程序。特定于用户的应用程序，有时还会包括一个嵌入式图形用户界面。

针对以上的四个软件层次，构建一个嵌入式 Linux 系统的基本步骤如下：

1. 搭建嵌入式 Linux 开发环境。
2. 根据开发板的硬件配置编写 BootLoader。
3. 裁减和编译嵌入式 Linux 内核。
4. 编写设备驱动程序和嵌入式 Linux 应用程序。
5. 构建嵌入式 Linux 根文件系统。

2.2.1 嵌入式 Linux 系统开发环境的搭建

嵌入式系统拥有的资源通常非常有限，直接在嵌入式系统的硬件平台上进行软件开发比较困难，甚至是不可能完成的任务。因此，目前嵌入式系统开发一般采用的办法是首先在 PC 机上裁减内核和编写程序，然后通过交叉编译生成目标平台上可以运行的二进制代码格式（若目标平台的体系结构与 PC 机相同则无需交叉编译），再将二进制代码下载到目标平台上的特定位置上运行，最后通过远程调试工具来调试程序。为了实现上述开发步骤，必须首先建立一个完备的嵌入式系统开发环境[7]。

基于嵌入式 Linux 系统的开发环境一般由宿主 PC 机、目标板、交叉编译工具链、远程调试工具和下载机制组成。本课题的工作是在装有 Linux 操作系统（Red Hat 9.0）的宿主 PC 机和基于 S3C2410X 的目标板上进行的。开发环境搭建步骤为：

1. 在宿主 PC 机上安装针对 ARM 的 GNU 交叉开发工具链。可到网上下载已经做好的工具链：cross-2.95.3.tar，然后作为 root 用户登陆，进入/usr/local 目录，新建 arm 目录，将交叉工具链解压到该目录下，最后将路径/usr/local/arm/2.95.3/bin 加入到环境变量 PATH 中。

2. 用 sjf2410 通过 JTAG 接口将内核引导程序烧写到目标板的 Flash 中，然后用串口线和 USB 总线连接宿主 PC 机和目标板，最后利用内核引导程序通过串口线和 USB 总线将嵌入式 Linux 内核和根文件系统下载到目标板的内存中运行调试。

3. 在宿主 PC 机上打开 NFS 服务，通过 NFS 方式来完成对嵌入式 Linux 的根文件系统的调试。NFS 服务的主要任务是把宿主 PC 机的一个本地目录通过网络输出，目标板可以远程挂接这个目录并且访问文件。这样可以避免在调试嵌入式 Linux 的根文件系统以及应用程序时反复烧写 Flash，方便和加速调试过程。

4. 建立嵌入式 Linux 的 GDB 远程调试环境，其分为两个部分在宿主 PC 机上安装的是 arm-linux-gdb，而在目标板上安装的是 gdbserver。这样，可以通过远程调试的方法，在宿主 PC 机上调试在目标板上运行的应用程序。

2.2.2 嵌入式 Linux 系统内核引导程序的实现

内核引导程序（Bootloader）是嵌入式系统上电后执行的第一个程序，其主要工作是将初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，并最终调用操作系统内核运行。Bootloader 一般有两种工作模式[6]：

- 启动加载模式：也称为自主（Autonomous）模式，即 Bootloader 从目标机上某个固态存储设备上将操作系统核心加载到 RAM 中运行。
- 下载模式：Bootloader 通过串口连接或网络连接等通信手段从主机上下载文件。

Bootloader 的实现依赖于嵌入式处理器的体系结构和嵌入式板级设备的配置。本课题实现的 Bootloader 分为 stage1 和 stage2 两个阶段，stage1 由汇编语言实现，stage2 由 C 语言实现。

- Bootloader 的 stage1

stage1 主要负责初始化相应的硬件和将 stage2 的代码拷贝到 RAM 空间执行，具体步骤如下：

1. 硬件初始化，其包括屏蔽所有的中断、设置 CPU 的时钟和速度、设置内存控制器等等。

2. 为加载 stage2 准备 RAM 空间。

3. 拷贝 stage2 的代码到 RAM 中。

4. 设置堆栈指针，跳转到 stage2 的 C 入口点

- Bootloader 的 stage2

stage2 主要负责检测系统的内存映射，将操作系统内核和根文件系统拷贝到内存中，调用内核运行，具体步骤如下：

1. 初始化本阶段要用到的硬件设备，根据本课题的研究需要，除了初始化了串口外，还初始化了 USB 接口以及计时器等。

2. 检测系统的内存映射，探查出 CPU 预留的全部 RAM 地址空间中有哪些被真正映射到物理 RAM 地址单元。

3. 将内核映像和根文件系统映像从 Flash 读到 RAM 中。

4. 设置内核的启动参数。

5. 调用内核，本课题的 Bootloader 通过直接跳转到内核的第一条指令处来调用 Linux 内核。

2.2.3 嵌入式 Linux 系统内核的裁减和移植

标准 Linux 内核功能十分强大，但对于针对特定应用并且资源受限的嵌入式系统来说过于庞大，因此，需要将标准 Linux 内核根据目标平台的情况进行裁减和移植，使其成为适合目标平台的嵌入式 Linux 内核[8]。

Linux 内核有很多版本，本课题采用的 Linux 内核版本是在嵌入式领域运用比较成熟的 2.4.18 版本，并应用了支持 S3C2410 的补丁，使得嵌入式 Linux 内核能够在 S3C2410 处理器上运行。得到了支持 S3C2410 处理器的 Linux 内核源码包后，就可以根据硬件电路对 Linux 内核进行裁减和配置，Linux 内核当前支持三种配置方式：

- 基于命令行的问答式，此配置方式通过 make config 命令启动，针对每一个内核配置选项会有一个提问，用户可以根据需求来给出答案，y 代表在内核中包含此项内核特性，m 代表此项内核特性作为模块编译（编译但不链入到内核镜像中），n 代表不对该特性提供支持。
- 基于字符的菜单式，此配置方式通过 make menuconfig 命令启动，用户可

以通过一个字符配置菜单对内核进行配置。

- 基于图形的菜单式，此配置方式通过 make menuconfig 命令启动，操作方式与第二种方式相似，界面更加友好，但需要 X windows 系统的支持。

本课题采用的是第二种内核配置方式，其操作界面如图 2-2 所示。根据硬件设备的具体情况进行的关键配置有：对系统类型（System Type）的配置时，选中 ARM920T-base 选项和 S3C2410-base 选项；对存储设备（Memory Technology Devices）配置时，选择支持 MTD 驱动以及 NAND FLASH 驱动；对文件系统（File System）配置时，选中对 CRAMFS 文件系统的支持。

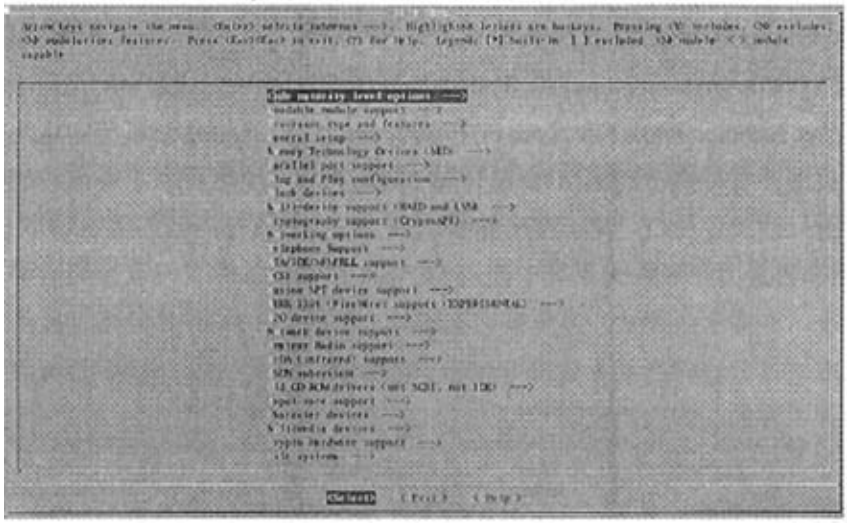

图 2-2 Linux 内核字符配置菜单

当根据系统需求配置好内核，退出配置菜单时，系统会询问是否需要保存修改后的内核配置，如图 2-3 所示。

图 2-3 Linux 内核字符配置菜单

若选择不保存，则进行的所有配置操作都无效，内核配置仍然为原来修改前的状态，若选择保存，系统会在当前目录下生成一个.config 文件，其后要进行的内核编译就是根据这个.config 文件来进行条件编译以生成相应的内核镜像文件[9]。

在配置工作完成后，就进入内核编译阶段。嵌入式 Linux 内核编译的步骤如下：

1. make clean，此命令会删除掉原来的编译结果以及一些旧的数据文件。

2. make dep，此命令会搜索内核编译中的依赖关系并修正内核编译配置。

3. make，此命令会执行实际的编译过程并在 arch/arm/boot 目录中生成内核镜像文件 zImage。

2.2.4 嵌入式 Linux 系统根文件系统的建立

在嵌入式 Linux 系统中，根文件系统是紧跟着嵌入式 Linux 内核存放，由内核加载，其主要用于组织系统中的文件和设备，为用户应用程序提供统一的接口。嵌入式 Linux 系统支持多种类型的根文件系统，常用的根文件系统类型有 CRAMFS（Compressed Rom File System）、 JFFS2(Journaling Flash File System)、YAFFS(Yet Another Flash File System)和基于 Ext2fs 的 RAMDISK 等。这几种根文件系统类型各有特点：CRAMFS 是针对 Linux 内核 2.4 之后版本所设计的一种高度压缩的只读文件系统，其压缩比能够达到 50%，并支持高效的随机读取；JFFS2 和 YAFFS 都是为在嵌入式系统中使用 FLASH 存储设备而专门设计的一种可读写文件系统，支持永久存储，但压缩比比较低；基于 Ext2fs 的 RAMDISK 是指将内存虚拟成硬盘分区并在其上建立 Ext2fs 文件系统，但掉电后 RAMDISK 的内容会随着内存内容的消失而消失[10][11]。

由于根文件系统不需要做太多的修改以及资源有限，故本课题采用的是 CRAMFS 文件系统，构建 CRAMFS 根文件系统的具体步骤如下：

1. 在网上下载制作 CRAMFS 根文件系统的工具 mkcramfs

2. 创建 CRAMFS 文件系统的根目录，如/root/rootfs

3. 在根目录下创建子目录以及文件，一个完整的 CRAMFS 文件系统通常包含以下几个重要的子目录和文件：

- /linuxrc 启动脚本文件：主要用于将/mnt/etc 目录下的配置文件拷贝到/mnt 目录，以及执行/sbin/init 程序，启动 shell 运行。
- /bin 目录：用于存放引导启动所需的命令或用户可能用到的命令。
- /sbin 目录：用于存放为系统管理员服务的命令
- /etc 目录：用于存放特定机器的配置文件。
- /lib 目录：用于存放文件系统上的程序执行所需的共享库
- /dev 目录：用于存放设备文件。
- /usr 目录：用于存放用户程序和配置文件。

4. 将自己开发的驱动和应用程序分别放在 lib/modules 目录和 usr/bin 目录下。

5. 在命令行输入命令 mkcramfs /root/rootfs rootfs.cramfs，利用 mkcramfs 工具生成 CRAMFS 文件系统。

2.3 Linux 操作系统内核

Linux 源代码的开放为修改和更新标准 Linux 内核代码，开发适合于目标平台的嵌入式 Linux 内核以及驱动程序提供了良好的机会，但这一切都必须建立在熟悉 Linux 内核结构和工作原理的基础之上。

2.3.1 Linux 内核的组成

Linux 是采用模块化程序设计方法开发的单内核结构的操作系统，如图 2-4 所示，可将 Linux 内核按功能划分为 5 个部分：进程管理、内存管理、文件系统、设备控制和网络[12]。

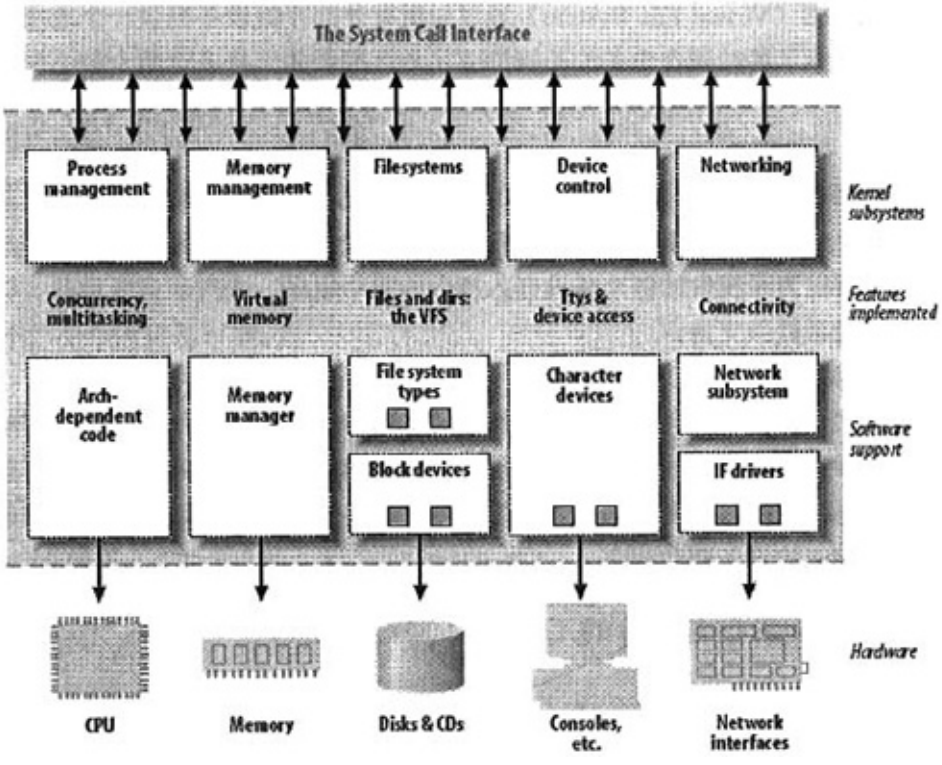

图 2-4 内核功能的划分

- 进程管理

进程管理模块负责创建和销毁进程，并且采用合适的调度策略对进程进行调度，控制进程对 CPU 的访问，使得各个进程能够公平合理的地访问 CPU，同时保证内核能够及时地执行硬件操作。除此之外，进程管理还支持进程间各种通信机制。

- 内存管理

内存管理模块用于确保所有进程能够安全地共享机器主内存区，同时内存管理模块还支持虚拟内存，将暂时不用的内存数据块交换到外部存储设备上去，当需要时再交换回来，这样使得 Linux 的进程可以使用比实际内存空间更多的内存容量。内存管理从逻辑上分为硬件无关部分和硬件相关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关的部分为内存管理硬件提供了虚拟接口。

- 文件系统

文件系统管理模块用于管理挂接在系统上的文件系统以及文件。Linux 虚拟文件系统 VFS（Virtual File System）通过向所有外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备的不同细节，从而提供并支持多达数十种不同的文件系统。

- 设备控制

几乎每一个系统操作最终都会映射到物理设备上。除了 CPU、内存以及其他有限的几个对象外，所有设备控制操作都由与被控制相关的代码即驱动程序来完成。内核通过为系统中的每个外设嵌入相应的驱动程序，来对硬盘驱动器、键盘和鼠标等设备提供支持。

- 网络接口

网络接口提供了对各种网络标准协议的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

2.3.2 Linux 内核各部分的工作机制

由于嵌入式 Linux 内核是通过对标准 Linux 进行裁减和修改后得到的，因此其支持的功能一般来说是标准 Linux 的子集，但作为一个操作系统，对于进程管

理、内存管理和文件系统管理的支持是必不可少的[13]。

- 进程管理

进程是一个执行中的程序实例，在 Linux 中每个进程都会经历一个从创建到执行，再到消亡的生命周期。在进程的生命周期中，Linux 内核是通过一个名为 task_struct 的数据结构来描述进程的。task_struct 也称为进程控制块 PCB（Process Control Block）或进程描述符 PD（Process Descriptor），其中保存着用于控制和管理进程的所有信息（进程调度信息、进程间通信信息、文件系统信息、虚拟内存信息等）。

Linux 内核可使用系统调用 fork()、clone()和 vfork()来创建一个进程，三个系统调用都会以不同的参数调用 do_fork()。do_fork()完成了进程创建中的大部分工作，其运行的大致流程为[14]：

1. 调用 alloc_task_struct()为新进程分配两个连续的物理页面，用作 task_struct 和系统空间堆栈，然后将父进程的 task_struct 复制到新分配的空间中。

2. 将新的子进程状态设置为 TASK_UNINTERRUPTIBLE，以保证它不会投入运行。

3. 调用 copy_flags()以更新 task_struct 的 flags 成员。

4. 调用 getpid()为新进程获取一个有效的 PID，再初始化 task_struct 相应的字段。

5. 根据参数标志，调用 copy_files()、copy_fs()、copy_sighand()和 copy_mm()来拷贝或共享打开的文件、文件系统信息、信号处理函数和进程地址空间等。

6. 将父进程的 task_struct 结构中的 counter 值（进程的运行时间配额）分成两半，父子进程各一半。

7. 再调 SET_LINKS()将子进程的 task_struct 结构链入内核的进程队列，然后通过 hash_pid()将其挂入可执行进程等待调度。

8. 新的子进程执行后一般会调用 exec()函数，执行一个新的程序。

进程被创建后就进入了执行阶段，进程的执行状态可以分为如图 2-5 所示的五种状态[15]：

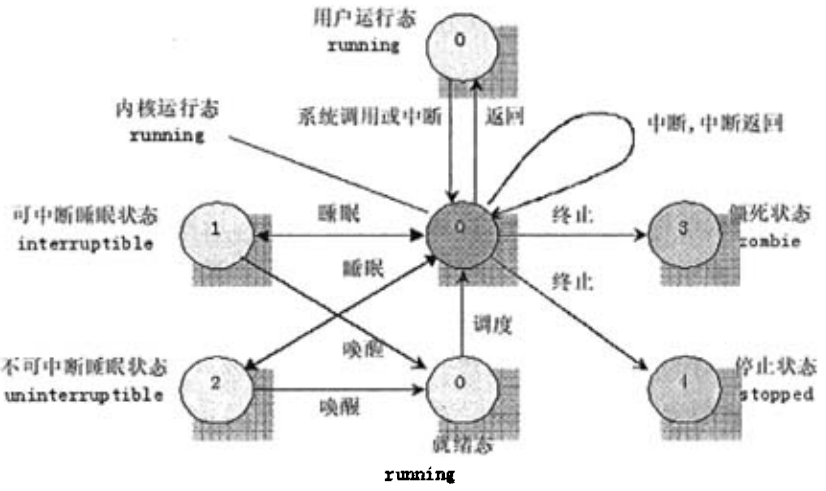

图 2-5 进程状态及转换关系

- 运行状态（TASK_RUNNING）。进程正在被 CPU 执行，或已经准备就绪在运行队列中等待执行。
- 可中断睡眠状态（TASK_INTERRUPTIBLE）。进程正在睡眠或被阻塞，当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程，转换到运行状态（就绪状态）。
- 不可中断睡眠状态（TASK_UNINTERRUPTIBLE）。与可中断睡眠状态类似，但是进程不会因为收到信号而被唤醒。这个状态通常在进程必须在等待时不受干扰或者等待事件很快就会发生时出现。
- 暂停状态（TASK_STOPPED）。进程暂时停止执行。这个状态通常发生在进程收到 SIGSTOP、SIGTTIN、SIGTTOU 的时候，当进程收到 SIGCONT 信号时会转换到运行就绪状态。
- 僵死状态（TASK_ZOMBIE）。当进程已停止运行，但其父进程还没有询问其状态。

进程之所以有五种执行状态，是因为 Linux 操作系统利用分时技术支持多个进程的同时运行。分时技术的原理是把 CPU 的运行时间划分成一个个规定长度的时间片，让每个进程在一个时间片内运行，当进程的时间片用完时系统就利用调度程序切换到另一个进程去运行。Linux 内核通过调度程序 schedule()来分时调度执行运行各个进程。在进程调度时，schedule()函数完成的主要工作是[14]：

1. 判断进程的 active_mm 是否为 0，检查进程是否在中断或低半部程序中，如果是则会出错。

2. 调 signal_pending()判断是否有信号发送给处于等待状态的当前进程，如果

有则将其唤醒，如果没有则将其从运行队列中删去。

3. 遍历可执行队列中的各个进程，为每个进程调 goodness()，计算出其当前具有的权值，最后找到队列中权值最高的那个进程，若该进程的权值不为 0 则将获得 CPU，若该进程的权值为 0 则要重新计算所有进程的时间配额。

4. 选出权值最高的进程后，schedule()最后会调用 switch_to()对进程进行切换。switch_to()主要通过堆栈的切换和程序执行的切换来完成进程的切换。

当一个进程完成自己的使命后，会通过调用 exit()或者从某个程序的主函数返回这两种方式进入消亡阶段，内核必须释放它所占的资源并通知其父进程，大部分的工作都是由 do_exit()来完成，其完成的工作主要有[14]：

1. 将 task_struct 中的标志成员设置为 PF_EXITING，调用 acct_process()来输出统计信息。

2. 调用__exit_files()、__exit_fs()、__exit_sighand()和__exit_mm()分别递减文件描述符、文件系统数据、信号处理函数和进程地址空间的引用计数。如果其中某些引用计数降为 0，则代表没有进程在使用相应的资源，可以彻底释放。

3. 调用 exit_notify()向父进程发送信号，通知父进程料理后事，并且将进程状态设置为 TASK_ZOMBIE。

4. 调用 schedule()切换到其他的进程，由于进程的状态为 TASK_ZOMBIE，进程将永远不会从 schedule()中返回。

父进程被唤醒后会遍历其子进程的状态，若发现有状态为 TASK_ZOMBIE 的子进程，则调用 release_task()释放子进程残存的资源。

- 内存管理

内存是计算机系统中最重要的资源之一，内存管理子系统是 Linux 内核的重要组成部分。Linux 的内存管理主要包括物理内存的管理和虚拟内存的管理。

为了有效的使用机器中的物理内存，Linux 内核将内存划分成几个功能区域，如图 2-6 所示。

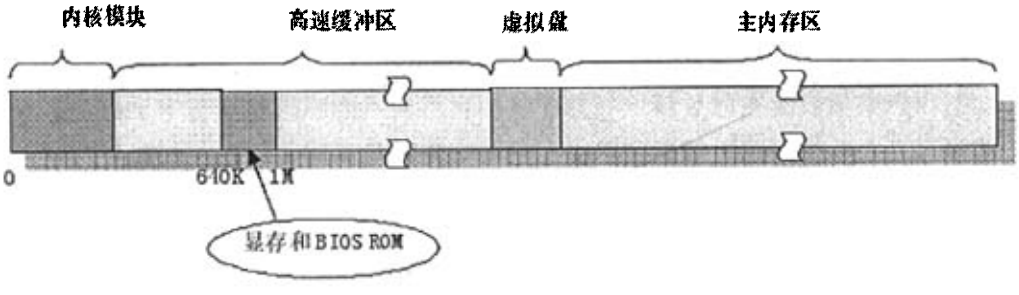

图 2-6 物理内存使用功能分布图

其中，Linux 内核程序占据在物理内存的开始部分，接下来是用于供硬盘等块设备使用的高速缓冲区部分。当有一个进程需要读取块设备中的数据时，系统会首先将数据读到高速缓冲区中；当有数据需要写到块设备上去时，系统也是先将数据放到高速缓冲区中，然后由块设备的驱动程序写到设备上。最后的部分是供所有进程可以随时申请使用的主内存区。所有程序在使用主内存区之前，都要首先向内核的内存管理子系统提出申请[15]。

Linux 内核支持多个进程同时运行，但内存是一种很有限的资源，它通常容纳不下系统中的全部活动进程。内存管理子系统利用了虚拟内存技术来解决这个问题，以满足活动进程所需的内存空间。虚拟内存通过使用磁盘作为 RAM 的扩展，而使系统看起来有多于实际内存的内存容量。Linux 内核利用地址映射机制、内存分配回收机制、缓存和刷新机制、请求页机制、交换机制和内存共享机制来实现虚拟内存。内存管理程序通过映射机制把用户程序的逻辑地址映射到物理地址。当用户程序运行时，如果发现程序中要用的虚地址没有对应的物理内存，就发出了请求页要求。如果有空闲的内存可供分配，就请求分配内存，并把正在使用的物理页记录在缓存中。如果没有足够的内存可供分配，那么就调用交换机制，腾出一部分内存。另外，在地址映射中要通过 TLB(翻译后援存储器)来寻找物理页；交换机制中也要用到交换缓存，并且把物理页内容交换到交换文件中，也要修改页表来映射文件地址。

- 文件系统

Linux 内核利用虚拟文件系统支持很多不同的逻辑文件系统和很多不同的硬件设备。虚拟文件系统对用户隐去了各种不同文件系统的实现细节，为用户程序提供了一个统一的、抽象的、虚拟的文件系统界面，此界面的主体为 file_operation、dentry_operations、inode_operations 和 super_operations 结构，它们是代表着文件系统逻辑文件操作、目录操作、物理文件操作和文件系统超级块操作的函数跳转表，用来指向具体文件系统实现的入口函数。文件管理子系统向用户提供系统调用 mount()和 umount()来实现文件系统的安装和拆卸。mount()负责将一个可访问的块设备安装到一个可访问的节点上，而 umount()则负责把已经安装的设备拆卸下来。

2.4. Linux 设备驱动程序

2.4.1 Linux 设备驱动概述

设备驱动程序是应用程序和实际设备之间的一个软件层，它向下负责和硬件设备的交互，向上通过一个通用的接口挂接到文件系统上，从而使用户或应用程序可以按操纵普通文件的方式进行访问控制硬件设备。作为 Linux 内核的重要组成部分，设备驱动程序主要完成以下的功能[17]：

- 对设备初始化和释放。
- 把数据从内核传送到硬件和从硬件读取数据。
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
- 检测错误和处理中断。

Linux 内核有三种类型的设备驱动程序：字符设备驱动程序、块设备驱动程序和网络设备驱动程序。字符设备是以字节为单位逐个进行 I/O 操作的设备，在对字符设备发出读写请求时，实际的硬件 I/O 紧接着就发生了，一般来说字符设备中的缓存是可有可无的，而且也不支持随机访问。块设备主要是针对磁盘等慢速设备设计的，其目的是避免耗费过多的 CPU 时间来等待操作的完成。它利用一块系统内存作为缓冲区，当用户进程对设备进行读写请求时，驱动程序先查看缓冲区中的内容，如果缓冲区中的数据能满足用户的要求就返回相应的数据，否则就调用相应的请求函数来进行实际的 I/O 操作。网络设备是一个能够和其他主机交换数据的设备，它通常是个物理设备，但也可能是个软件设备，如回环设备（loopback）。网络驱动程序负责驱动设备发送和接收数据包。除了设备类型以外，内核还使用了一个主设备号和一个次设备号来唯一标识设备，主设备号标识了设备对应的驱动程序，而次设备号仅由驱动程序解释，一般用于识别在若干可能的硬件设备中，I/O 请求所涉及到的那个设备[16]。

2.4.2 Linux 设备驱动与内核的关系

在 Linux 中将驱动程序嵌入内核有两种方式：一种是静态编译链接进内核，另一种是先编译成模块，再动态加载进内核。如果采用静态编译链接方式，需要把驱动程序的源代码放在内核源代码目录“Drivers/”下，并修改 Makefile 文件。在编译链接内核的时候，驱动程序会作为内核的一部分链接到内核镜像文件中。这种方式会增加内核的大小，还要改动内核的源文件，而且不利于调试，不如模

块方式方便灵活[18]。模块（module）在 Linux 中是一种已经编译好的目标文件，它可以被链接进内核从而生成可执行的机器代码。如果采用模块加载的方式，驱动程序首先会被编译成未链接的目标文件，具有 root 权限的用户在需要时可利用 insmod 命令将其动态的加载到内核中，而在不需要的时候可利用 rmmod 命令卸载该模块[19]。

驱动程序加载到 Linux 内核中后，会利用虚拟文件系统 VFS 提供的一个统一的接口（file_operations 数据结构）挂接到虚拟文件系统下，二者关系如图 2-7 所示：

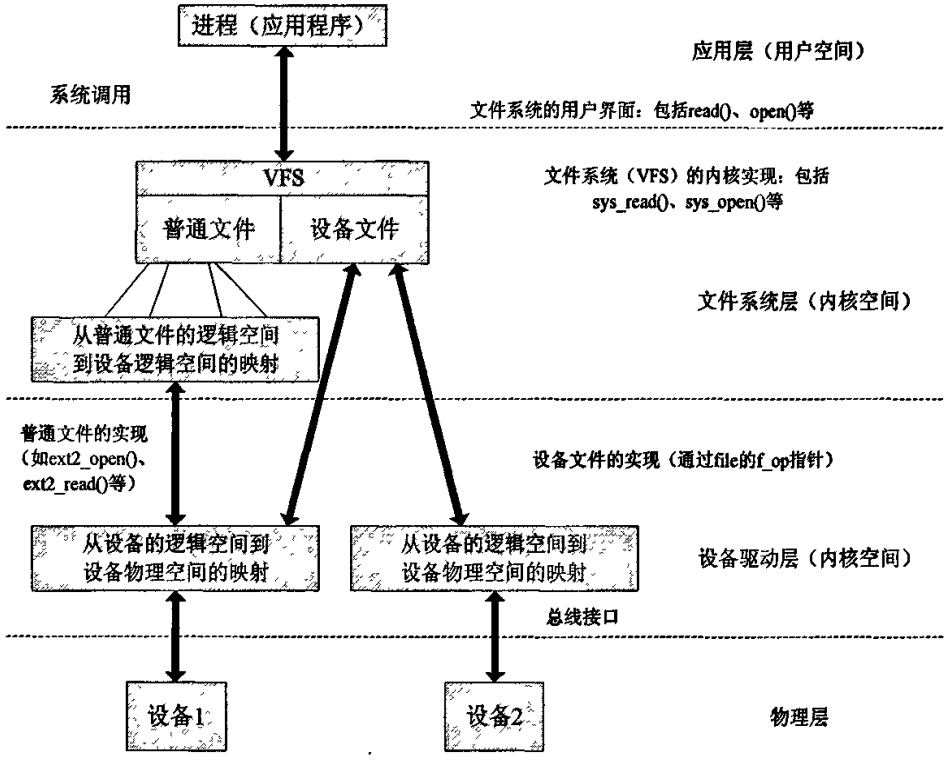

图 2-7 Linux 驱动程序与内核的关系

从上图可以看出，一个用户应用程序要对一个设备文件进行操作，它的流程大致如下[20]：

1. 用户程序通过 Linux 所提供的系统调用如 open()、read()等进入内核。这些函数到内核 2.4 版本为止总共有两百多个，提供了几乎所有应用程序里可能需要进入内核运行的操作。

2. 内核里对应这些系统调用的函数是 sys_open()、sys_read()等，函数的参数与系统调用函数的参数相同，它们由内核空间里的虚拟文件系统层实现。虚拟

文件系统 VFS 是一个从普通文件和设备文件抽象出来的一个文件系统层，完成了进入具体的设备文件的操作之前的准备工作。

3. Linux 内核将通过 file_operations 结构进入具体的设备文件的操作函数，这部分的函数就是由设备驱动程序所提供的。这些函数负责对设备硬件进行各种操作。

2.4.3 Linux 设备驱动的结构

Linux 的设备驱动程序按实现的功能大致可以分为如下几个部分：驱动程序的注册与注销、设备的打开与释放、设备的读写操作、设备的控制操作、设备的中断和轮询处理[21]。

- 驱动程序的注册与注销

向系统增加一个驱动程序意味着要赋予它一个主设备号，这可以通过在驱动程序的初始化过程中调用 register_chrdev()或者 register_blkdev()来完成。而在关闭字符设备或者块设备时，则需要通过调用 unregister_chrdev()或 unregister_blkdev()从内核中注销设备，同时释放占用的主设备号。

- 设备的打开与释放

打开设备是通过调用 file_operations 结构中的函数 open()来完成的，它是驱动程序用来为今后的操作完成初始化准备工作的。在大部分驱动程序中，open()通常需要完成下列工作：

1. 检查设备相关错误，如设备尚未准备好等。
2. 如果是第一次打开，则初始化硬件设备。
3. 识别次设备号，如果有必要则更新读写操作的当前位置指针 f_ops。
4. 分配和填写要放在 file->private_data 里的数据结构。
5. 使用计数增 1。

释放设备是通过调用 file_operations 结构中的函数 release()来完成的，这个设备方法有时也被称为 close()，它的作用正好与 open()相反，通常要完成下列工作：

1. 使用计数减 1。
2. 释放在 file->private_data 中分配的内存。
3. 如果使用计算为 0，则关闭设备。

- 设备的读写操作

字符设备的读写操作相对比较简单，直接使用函数 read()和 write()就可以了。

但如果是块设备的话，则需要调用函数 block_read()和 block_write()来进行数据读写，这两个函数将在设备请求表中增加读写请求，以便 Linux 内核可以对请求顺序进行优化。由于是对内存缓冲区而不是直接对设备进行操作的，因此能够很大程度上加快读写速度。如果内存缓冲区中没有所要读入的数据，或者需要执行写操作将数据写入设备，那么就要执行真正的数据传输，这是通过调用数据结构 blk_dev_struct 中的函数 request_fn()来完成的。

- 设备的控制操作

除了读写操作外，应用程序有时还需要对设备进行控制，这可以通过设备驱动程序中的函数 ioctl()来完成。ioctl()的用法与具体设备密切关联，因此需要根据设备的实际情况进行具体分析。

- 设备的中断和轮询处理

对于不支持中断的硬件设备，读写时需要轮流查询设备状态，以便决定是否继续进行数据传输。如果设备支持中断，则可以按中断方式进行操作。

2.4.3 Linux 设备驱动的设计和实现步骤

由于模块方式要比静态编译链接方式更加方便灵活，因此在 Linux 内核基础上二次开发的设备驱动程序一般是按照模块方式实现的。模块化驱动程序的设计和实现流程主要有编写模块化编程子程序、编写自动配置和初始化子程序、编写服务于 I/O 请求的子程序和编写中断服务子程序四个步骤。

- 模块化编程子程序

模块化编程子程序主要包括 init_module()函数和 cleanup_module()函数。init_module()函数在模块被加载到内核时调用，其主要负责向内核注册模块所提供的任何新功能。新功能可能是一个完整的驱动程序或者仅仅是一个新的软件抽象。对于每种新功能，init_module()函数会调用与其相对应的内核函数完成注册。字符设备驱动程序对应的内核函数的函数原型为：

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

其中，major 是为设备驱动程序向系统申请的主设备号，如果为 0 则系统为此驱动程序动态地分配一个主设备号。name 是设备名，fops 是指向与设备驱动对应的 file_operations 结构的指针。如果 register_chrdev()操作成功，驱动程序会注册到内核中，设备名也会出现在/proc/devices 文件里。在成功向系统注册了设备

驱动程序后，就可以使用 mknod 命令来将设备映射为一个设备文件，其它程序使用这个设备的时候，只要对此设备文件进行操作就行了。

cleanup_module()函数在模块被卸载的时候调用，其主要负责从内核中注销模块所提供的各种功能。cleanup_module ()函数会调用与其相对应的内核函数完成注销，字符驱动程序对应的内核函数的函数为 unregister_chrdev()。

- 自动配置和初始化子程序

自动配置和初始化子程序一般在设备接入系统时或者加载设备驱动时调用，其主要负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的驱动程序需要的软件状态进行初始化。自动配置和初始化子程序还负责为驱动程序申请包括内存、中断、时钟、I/O 端口等资源，这些资源也可以在 xxx_open()函数中申请。

- 服务于 I/O 请求的子程序

服务于 I/O 请求的子程序，又称为驱动程序的上半部分，调用这部分是由于系统调用的结果。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是进程的运行状态由用户态变成了核心态，具有进行此系统调用的用户程序的运行环境。驱动程序所提供的与设备的打开、释放、读写和控制操作相对应的入口点函数都属于服务于 I/O 请求的子程序，并且通过 file_operations 结构向系统进行说明。file_operations 结构的定义如下：

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
        //修改文件的当前读写位置
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        /从设备读取数据
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        //向设备发送数据
    int (*readdir) (struct file *, const char * size_t, loff_t *);
        //读目录，设备驱动中应设为 NULL
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
        //询问设备是否可读可写，如为 NULL，则表明设备总是可读写的
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        //调用设备控制相关命令
    int (*mmap) (struct file *, struct vm_area_struct *);
```

```
    //将设备内存映射到进程内存中
int (*open) (struct inode *, struct file *);
    //打开设备
int (*flush) (struct file *);
    //将缓冲中的数据写回设备
int (*release) (struct inode *, struct file *);
    //关闭设备
int (*fsync) (struct file *, struct dentry *);
    //刷新设备
int (*fasync) (int , struct file *, int);
    //用于异步触发，通知设备 FASYNC 标志的变化
int (*lock) (struct file *, int, struct file_lock *);
    //给文件上锁
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

从 file_operations 结构的定义中可以看出，该结构是一个函数指针表，这个表中的每一项都指向由驱动程序实现的处理相应请求的函数。在编写设备驱动程序时，根据具体设备提供的功能，分别编写相应的函数，函数的名称可任意起，但要符合函数名称的基本要求，然后按照其实现的功能填入 file_operations 结构相应的位置中。对于驱动程序来说，常用的方法指针主要有 open()、release()、read()、write()和 ioctl()等，它们对应着备的打开、释放、读写和控制等基本的设备操作。

- 中断服务子程序

中断服务子程序，又称为驱动程序的下半部分。在 Linux 系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由 Linux 系统接收硬件中断，再由系统调用中断服务子程序。中断可以产生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境相关的函数。

由于对中断的处理是属于系统核心的部分，因此如果设备与系统之间以中断方式进行数据交换的话，就必须把该设备的驱动程序作为系统核心的一部分。设备驱动程序在设备第一次打开、硬件被告知产生中断之前调用 request_irq()函数来申请中断，在最后一次关闭、硬件设备被告知不用再中断处理器后调用 free_irq()

函数来释放中断。它们的定义为：

```
int request_irq(unsigned int irq,
        void (*handler) (int irq, void dev_id, struct pt_regs *regs),
        unsigned long flags, const char *device, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

参数 irq 表示所要申请的硬件中断号，handler 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号，dev_id 为申请时告诉系统的设备标识，regs 为中断发生时寄存器内容。device 为设备名，将会出现在/proc/interrupts 文件里。flags 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是中断处理程序是快速中断处理程序（flags 里设置了 SA_INTERRUPT）还是慢速处理程序，快速处理程序运行时，所有中断都被屏蔽，而慢速处理程序运行时，处理正在处理的中断外，其它中断都没有被屏蔽。dev_id 用来区分共享同一中断的不同的中断处理程序（flags 里设置了 SA_SHIRQ），如果中断由某个处理程序独占，则 dev_id 可以为 NULL。

Linux 中断处理程序一般分为两个半部：上半部（tophalf）和下半部(bottom half)。上半部的功能是"登记中断"，当一个中断发生时，它首先会调用相应的硬件读写函数将设备的“中断挂起”位清除，让设备可以继续产生中断，然后再把中断例程的下半部挂到该设备的下半部执行队列中去。因此，上半部执行的速度就会很快，可以服务更多的中断请求，下半部则完成中断处理的绝大多数工作。上半部和下半部最大的不同是上半部是不可中断的，而下半部是可中断的，而且可以被新的中断打断。下半部相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。

第三章　嵌入式 Linux 下 USB 设备驱动的设计与研究

在嵌入式系统的应用中，数据的输入输出有多种方式，通过通用串行总线 USB 进行传输，效率是比较高的。USB 设备的工作必须由 USB 驱动程序来支持。本章介绍了 USB 总线协议以及 USB 设备的 Linux 驱动框架，并在基于 S3C2410 的 FS2410 开发板上实现了基于 ZC301P 芯片的 USB 摄像头的嵌入式 Linux 驱动程序，最后针对项目开发中出现 USB 传输带宽冲突的问题，提出了相应的解决办法。

3.1 USB 总线概述

USB 是一种轮询式电缆总线，它支持主控计算机与同时接入多个外设之间的数据交换。主计算机系统上的 USB 接口是主控制器，或 USB 主机端。主控制器的实现可以是硬件、固件或软件的组合。它初始化所有数据传输，因而所连接的外设可以通过一个由主机端安排、基于令牌的协议来共享 USB 带宽[22]。总线支持热插拔外设，用户可以在主机端和其它外设处于工作状态时插入、配置、使用和拔下外设。USB 总线协议的最新版本是 2.0。USB 2.0 支持三种总线传输速度：高速 480 Mbps、全速 12 Mbps 和低速 1.5 Mbps。

3.1.1 USB 总线拓扑结构

USB 的总线拓扑结构是由 USB 主机（USB Host）、USB 集线器（USB Hub）和 USB 设备（USB Device）三个元素组成，如图 3-1 所示，USB 的物理连接是有层次性的星型结构。

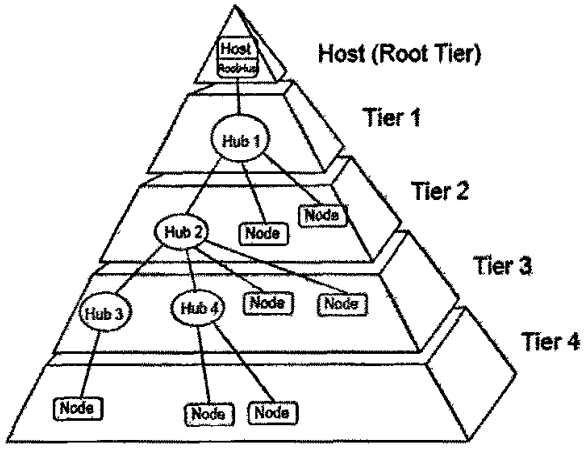

图 3-1 USB 总线的拓扑结构

USB 主机是整个星型结构的中心，它负责管理整个 USB 系统，包括 USB 设备的连接与删除、USB 主机与 USB 设备的通信、总线路的控制等等。USB 主机端有一个 Root Hub，可提供一个或者多个 USB 下行端口，每个端口可以连接一个 USB Hub 或者一个 USB 设备。USB Hub 是一类特殊的 USB 设备，是一组 USB 的连接点。图 3.1 中的每个 Node 就是一个 USB 设备，它连接到 USB Hub 上，负责响应由 USB 主机发起的数据传输。从 USB 主机到 USB Hub 或 USB 设备，或从 USB Hub 到 USB Hub 或 USB 设备，每条线段都是点点连接。

3.1.2 USB 总线的通信流模型

USB 通信通常指的是 USB 主机和 USB 设备之间的通信。USB 协议采用了分层的概念来描述了二者的通信机制。如图 3-2 所示：

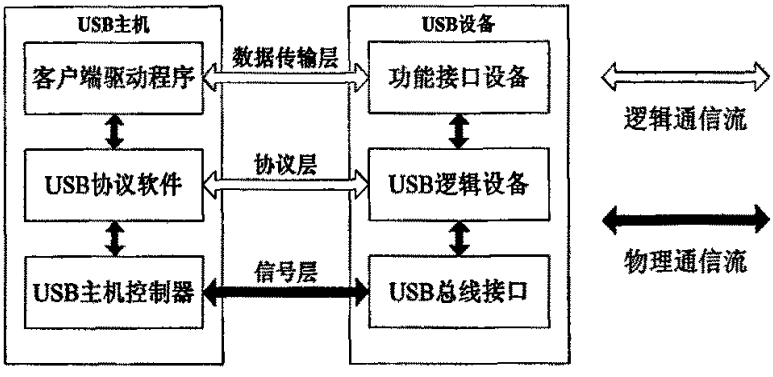

图 3-2 USB 通信的逻辑结构

从上图可以看出，USB 协议将 USB 通信在逻辑上划分为三层：信号层、协

议层和数据层，其中位于最底层的信号层是真正的物理对应关系，而其它两层则是逻辑对应关系。信号层为主机接口和设备接口的连接，传送的是二进制比特流，对应的信息流单元为数据包（Packet）；协议层逻辑上是USB协议栈和USB逻辑设备之间的对应，传送的是包字节流，对应的信息流单元为事务（Transaction）；数据层则是主机端驱动程序或者应用软件和设备端功能单元的对应，传送的是原始的数据信息，对应的信息流单元为数据传输（Transfer）。三层的信息流单元有着一个递进的组成结构关系：数据包是USB总线传输的基本单元，一组数据包构成了一个事务，而一个或者多个事务构成了一个数据传输。USB协议支持以下四种数据传输类型[23]：

- 控制数据传输

控制数据传输主要用于对设备进行配置管理。当USB设备初次安装时，USB系统软件使用控制数据传输对设备进行设置，数据传送是无损性的。

- 批量数据传输

批量数据传输主要用于在不确定的时间间隔进行大量数据传输。在硬件级上可使用错误检测可以保证可靠的数据传输，并在硬件级上引入了数据的多次传输。此外根据其它一些总线动作，被大量数据占用的带宽可以相应的进行改变。

- 中断数据传输

中断数据传输主要用于一些有确定的传输周期但数据量不大的设备数据传输。中断数据是少量的，且其数据延迟时间也是有限范围的。这种数据可由设备在任何时刻发送，并且以不慢于设备指定的速度在USB上传送。

- 同步数据传输

同步数据传输主要适用于对时间要求较高，数据量较大但对错误率要求不高的数据传输。同步数据的建立、传输和使用是连续且实时的，要求接收者与发送者保持相同的时间安排。

3.1.3 USB数据的传输流程

在USB总线上数据的传送是以帧(Frame)为单位进行的，一帧的传送时间通常为1ms（高速传输一帧为125μs）。一帧中能实现的最大数据传输量，即所能传输的最大字节数称为带宽[24]。USB采用共享带宽分配方案，USB允许同步和中断传输占用高达90%的带宽，剩下的10%的带宽用于控制传输，批量传输仅在带宽满足要求的情况下才会出现。如图3-3所示，在每一帧开始的时候，USB总线控

制器会首先传送同步传输数据包，然后再执行中断数据传输，在剩余 10%的时间里传送控制传输数据包和批量传输数据包，一直到帧中的剩余时间不足以完成一次数据包传输为止[23]。

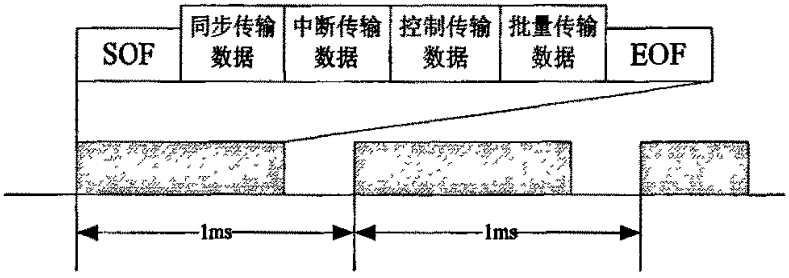

图 3-3 USB 总线上的数据传输结构

3.2 Linux 环境下的 USB 设备驱动程序

3.2.1 Linux 中 USB 设备驱动的核心数据结构

Linux 内核提供了几个与 USB 设备驱动程序开发直接相关的核心数据结构，它们是 USB 驱动程序的灵魂，在实现 USB 的体系结构、完成 USB 设备的即插即用、构建驱动程序的框架方面都起着非常重要的作用[25]。

- STRUCT USB_DRIVER

usb_driver 数据结构主要用于向 USBD 层注册 USB 设备驱动程序。在 Linux 内核中，每个 USB 设备驱动程序都对应着一个 usb_driver 数据结构。当总线上有设备接入的时候，USBD 中的 USB Hub 驱动程序通过该结构来查找相关的设备驱动程序，而对于设备断开操作，USBD 同样通过该结构来断开设备驱动程序与相关设备的连接，实现了 USB 设备的即插即用。usb_driver 结构的定义如下：

```
struct usb_driver{
    struct module *owner;                    //USB 驱动程序的模块所有者
    const char *name;                        //USB 驱动程序的名字
    struct file_operations *fops;            //USB 设备文件操作
    const struct usb_device_id * id_table;   //驱动所支持的设备 ID
    int (* probe) (struct usb_interface *intf, const struct usb_device_id *id);
                    //USB 驱动程序中的探测函数
    void (* disconnect) (struct usb_interface *intf);
```

```
                    //USB 驱动程序中的移除函数
    ... ... ... ... ...
};
```

- STRUCT USB_DEVICE

usb_device 数据结构是对系统中所有 USB 设备的抽象，USBD 通过该结构对连接到总线上的所有 USB 设备进行管理和资源分配。这个数据结构不仅包含了 USB 设备必要的硬件信息，而且通过它还可以了解到 USB 设备具体的配置信息。usb_device 结构的定义如下：

```
struct usb_device{
    int devnum;                          //USB 设备唯一的设备地址
    enum {
    USB_SPEED_UNKNOWN = 0,               //非法传输速率
    USB_SPEED_LOW,                       //低速 USB 设备
    USB_SPEED_FULL,                      //全速 USB 设备
    USB_SPEED_HIGH                       //高速 USB 设备
    } speed;                             //描述 USB 设备传输速度
    atomic_t refcnt;                     //访问计数
    struct usb_device *parent;           //指向该设备的上级设备
    struct usb_bus *bus;                 //指向 USB 设备所连接的总线
    struct usb_device_descriptor descriptor;        //USB 设备的设备描述字
    struct usb_config_descriptor *config;       //USB 设备的所有配置描述字
    struct usb_config_descriptor *actconfig; //处于使能状态的 USB 配置描述字
    ... ... ... ... ...
};
```

- STRUCT URB

统一的 USB 数据传输块（Universal Request Block，URB）包含一次 USB 数据传输所必需的所有信息，其主要构成有：USB 设备信息、数据传输管道、数据传输缓冲区、数据传输缓冲区长度、回调函数、不同类型数据传输必需参数、数据传输过程控制参数等。由于设备驱动程序与 USBD，以及 USBD 与 HCD 之间的数据传输都是通过数据结构 URB 来实现的，因而 URB 中包含了一个 USB 数据传输在三层驱动程序中的必要信息，其定义如下：

```
struct urb{
    struct usb_device *dev;                 //指向该 URB 所要访问的设备
    unsigned int pipe;                      //与 URB 相关联的数据传输管道
    unsigned int transfer_flags;            //URB 的数据传输标志
    void *transfer_buffer;                  //指向该 URB 的数据传输缓冲区
    int transfer_buffer_length;             //该 URB 的数据传输缓冲区的大小
    int bandwidth;                          //URB 所需的带宽
    int start_frame;                        //起始帧号
    int number_of_packets;                  //保存 URB 相关的数据包数据
    int interval;                           //URB 数据传输间隔
    int error_count;                        //URB 数据传输错误计数
    int timeout;                            //用于控制 URB 数据传输超时错误
    usb_complete_t complete;                //保存 URB 的回调函数
    ... ... ... ... ...
};
```

3.2.2 USB 主机驱动结构

一个 USB 系统包含了三类硬件设备：USB 主机（USB HOST）、USB 设备（USB DEVICE）和 USB 集线器（USB HUB）。由于各个 USB 设备按照树状网络结构连接到 USB 主机端，并且 USB 接口的工作方式属于主-从式，所以 USB 主机必须具有对所有连接在 USB 总线上的不同类型的 USB 设备进行管理的功能。如图 3-4 所示，Linux USB 主机驱动由三部分组成：USB 主机控制器驱动（HCD）、USB 驱动（USBD）和不同类型的 USB 设备类驱动[26][27]。

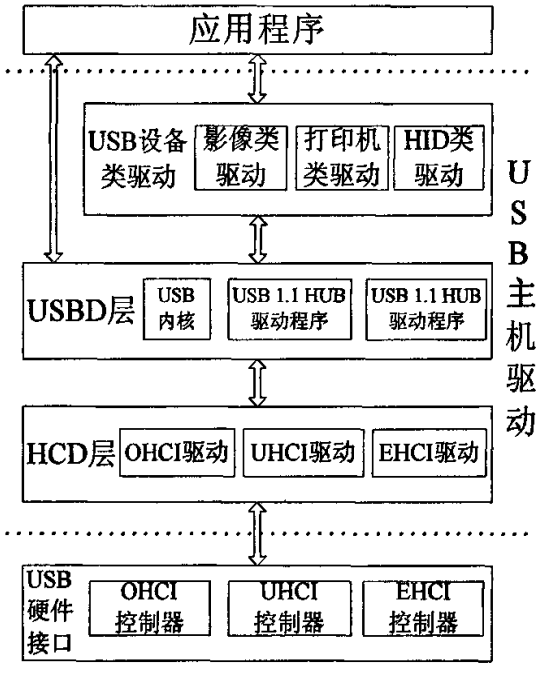

图 3-4 Linux USB 主机驱动程序结构

- USB 主机控制器驱动（HCD）

HCD 是 USB 主机驱动程序中直接与硬件交互的软件模块，其主要功能有：主机控制器初始化；为 USB 驱动（USBD）层提供相应的接口函数；提供根 HUB 的设备配置、控制功能等。

- USB 驱动（USBD）

USBD 是整个 USB 主机驱动的核心，负责控制全部 USB 协议的操作和中断处理控制。其主要功能有：USB 总线管理；USB 设备的配置、控制功能；USB HUB 驱动；完成 USB 的 4 种类型数据传输；为 USB 设备类驱动提供相关接口；为 USB 主机控制器驱动（HCD）提供相关接口等。

- USB 设备类驱动

USB 设备类驱动是最终与应用程序交互的软件模块，不同类型的设备通常对应于不同的设备类驱动。其主要功能有：配置和初始化特定的 USB 设备、为应用程序提供访问接口等。

- USB 数据传输

在本课题采用的 USB 设备中，USB 摄像头的数据传输类型为同步传输，而 USB 鼠标的数据传输类型属于中断传输，同步数据传输和中断数据传输又可以统称为周期性数据传输。

不管哪种传输类型，应用程序都是通过利用 USB 主机驱动提供的访问接口来进行数据的传输，而在 USB 主机驱动内部，数据传输都是通过统一的 USB 数据传输块 URB（Universal Request Block）数据结构完成的，其包含了一次 USB 数据传输所必需的所有信息。URB 最终会提交到 HCD 层，由 HCD 层将数据传输到指定的 USB 设备硬件上[28]。

3.2.3 USB 设备类驱动接口分析

从图 3.4 可以看出，USB 设备类驱动是位于应用程序层与 USBD 层之间的一层软件层，需要同时与应用程序和 USBD 进行交互。因此，USB 设备类驱动通过 usb_driver 结构提供了两类接口函数[33]：

- 与 USBD 层交互的接口

USB 设备类驱动提供给 USBD 调用的接口函数主要是 probe()函数和 disconnect()函数。probe()函数是 USB 设备类驱动实现的自动配置和初始化子程序，其主要负责检测和初始化设备，如果检测到匹配的硬件设备，还会初始化驱动程序的状态及申请资源。USBD 一般在设备接入系统时调用 USB 设备类驱动提供的 probe()函数来判断驱动程序与设备是否匹配。disconnect()函数则主要负责释放驱动程序申请过或者占用的资源。USBD 一般在设备从系统中断开时调用 disconnect()函数来完成一些善后工作。

- 与应用程序层交互的接口

USB 设备类驱动提供给应用程序调用的接口函数主要是服务于 I/O 请求的子程序，包括与设备的打开、释放、读写和控制操作相对应的 open()、close()、read()、write()和 ioctl()等入口点函数。这部分接口函数的函数指针一般由 usb_driver 结构中保存的 fops 指针指向的 file_operations 结构保存。当用户进程利用系统调用对设备文件进行操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个 file_operations 结构中的函数指针，最后将控制权交给该函数。

3.3 USB 摄像头的 Linux 设备类驱动的设计与实现

在前面几节中，讨论了 USB 主机驱动程序结构和核心数据结构。本节将按照 Linux 设备驱动的设计和实现步骤来介绍基于 ZC301P 芯片的 USB 摄像头驱动程序的具体实现。

3.3.1 USB 摄像头驱动的模块化编程子程序

在实现 USB 摄像头驱动的模块化编程子程序之前需要定义两个重要的静态全局变量 device_table[]和 spca5xx_driver，其中 device_table[]代表了当前驱动支持的 USB 设备，而 spca5xx_driver 则代表当前的 USB 驱动程序。

```
static struct usb_device_id device_table[] = {
{USB_DEVICE(USB_ZC301P_VENDOR_ID, USB_ZC301P_PRODUCT_ID)},
{ }
};
```

//USB_ZC301P_VENDOR_ID 和 USB_ZC301P_PRODUCT_ID 分别是 ZC301P 芯片的制造商 ID 和设备 ID。

```
static struct usb_driver spca5xx_driver = {
    module:     THIS_MODULE,
    name:       "spca5xx",
    probe:      spca5xx_probe,
    disconnect: spca5xx_disconnect,
    id_table:   device_table
};
```

//spca5xx_probe 和 spca5xx_disconnect 分别是探测函数和断开函数。

定义了静态全局变量 device_table[]和 spca5xx_driver 之后，接下来就是模块装载函数 usb_spca5xx_init ()和模块移除函数 usb_spca5xx_exit ()的实现。

```
static int __init usb_spca5xx_init (void)
{
    ... ... ... ... ...
    /* 注册硬件驱动程序 */
    if (usb_register (&spca5xx_driver) < 0)
        return -1;
    ... ... ... ... ....
     return 0;
}
static void __exit usb_spca5xx_exit (void)
{
    ... ... ... ... ...
    /* 注销硬件驱动程序 */
```

```
    usb_deregister (&spca5xx_driver);
    ... ... ... ... ...
}
```

usb_spca5xx_init()主要完成的工作是用&spca5xx_driver 作为参数调用 usb_register()函数向 USB 子系统注册 USB 设备驱动。而 usb_spca5xx_exit ()则用 &spca5xx_driver 作为参数调用 usb_register()函数向 USB 子系统卸载 USB 设备驱动。

3.3.2 USB 摄像头驱动的探测和移除子程序

当 USB 设备驱动向 USB 子系统注册后，如果有一个新的 USB 设备连接到系统中，USBD 会调用 USB 设备驱动的探测和配置初始化函数，本课题实现的 USB 设备驱动的探测和配置初始化函数原型为：

```
static void * spca5xx_probe (struct usb_device *dev, unsigned int ifnum,
        const struct usb_device_id *id)
```

其中参数 dev 指定了包含设备说明的内容，而参数 interface 指明了该设备所含的接口数目。spca5xx_probe()函数首先会检测新加入的 USB 设备硬件中的生产厂商和产品定义以及设备所属的类或子类定义是否与本驱动相符，若相符，再比较接口的数目与本驱动支持设备的接口的数目是否相符，如果都相符，将会确认新加入的 USB 设备会使用本驱动，最后进行一些初始化工作。spca5xx_probe()函数的具体实现如下：

```
static void * spca5xx_probe (struct usb_device *dev, unsigned int ifnum,
        const struct usb_device_id *id)
{
    struct usb_interface_descriptor *interface;        //接口描述结构
    struct usb_endpoint_descriptor *endpoint;          //端点描述结构
    struct usb_spca50x *spca50x;                       //特定设备实例
    interface = &dev->actconfig->interface[ifnum].altsetting[0];
    endpoint = &interface->endpoint[0]
    /* 检测设备是否与此驱动相符 */
    if ((dev->descriptor.idVendor != 0x0ac8) ||
        (udev->descriptor.idProduct != 0x301b)) {
```

```
        err (" Devices not found !! ");
        return NULL;
    }
    ... ... ... ... ...
    /* 为与设备相对应的实例分配存储空间 */
    if ((spca50x = kmalloc (sizeof (struct usb_spca50x),GFP_KERNEL)) ==
    NULL){
        err ("couldn't kmalloc spca50x struct");
        goto error;
    }
    memset (spca50x, 0, sizeof (struct usb_spca50x));
    spca50x->dev = dev;
    /* 分配一个新的 video_device 结构 */
    spca50x->vdev = video_device_alloc ();
    /* 将定义的 spca50x_template 赋给新的 video_device 结构 */
    memcpy (spca50x->vdev, &spca50x_template, sizeof (spca50x_template));
    /* 用定义的 video_device 结构向 Linux 视频子系统注册 */
    if (video_register_device (spca50x->vdev, VFL_TYPE_GRABBER, video_nr)
    < 0){
        err ("video_register_device failed");
        goto error;
    }
    ... ... ... ... ...
}
```

在目前的 Linux 核心中，视频部分的标准是 Video for Linux（简称 V4L）。这个标准定义了一套接口，内核、驱动、应用程序以这个接口标准进行交流。摄像头属于视频类设备，因此，USB 摄像头的驱动应当与内核提供的视频驱动挂钩。这部分工作一般由 USB 摄像头驱动的探测和配置初始化函数完成，从上述 spca5xx_probe()函数的实现中，可以看出 spca5xx_probe()函数会以一个已定义的 video_device 结构调用 video_register_device()函数来向 Linux 视频子系统注册，完成 USB 摄像头的驱动和内核提供的视频驱动挂钩。本驱动中的 video_device 结构定义如下：

```
static struct video_device spca50x_template = {
    name:           "SPCA5XX USB Camera",
    type:           VID_TYPE_CAPTURE,
    hardware:       VID_HARDWARE_SPCA5XX,
    open:           spca5xx_open,
    close:          spca5xx_close,
    read:           spca5xx_read,
    mmap:           spca5xx_mmap,
    ioctl:          spca5xx_ioctl
};
```

当本驱动支持的 USB 摄像头从系统中断开时，USBD 会调用 USB 设备驱动的移除函数 spca5xx_disconnect ()来清除那些已分配的私有数据，并且调用 video_unregister_device ()函数来从视频子系统中注销掉自己，spca5xx_disconnect () 函数的具体实现如下：

```
static void spca5xx_disconnect (struct usb_device *dev, void *ptr)
{
    ... ... ... ... ...
    /* 从视频子系统中注销 */
    if (spca50x->vdev)
        video_unregister_device (spca50x->vdev);
    spca50x->dev = NULL;
    /* 释放申请的内核空间 */
    if (spca50x && !spca50x->user){
        spca5xx_dealloc (spca50x);
        kfree (spca50x);
        spca50x = NULL;
    }
    ... ... ... ... ....
}
```

3.3.3 USB 摄像头驱动的服务于 I/O 请求的子程序

USB 摄像头驱动的服务于 I/O 请求的子程序主要包括设备的打开、关闭，读取视频数据和设备的控制等函数，如前所述，这些操作都是通过 video_device 结构提供给应用程序的。

- 设备的打开

在使用 USB 设备之前，必须先打开所要使用的 USB 设备。对于用户来说，只要在应用程序中调用 open()函数，由于驱动程序已经向 USB 核心子系统注册，应用程序就会自动进入驱动程序在 video_device 结构中的 spca5xx_open 函数，其具体实现如下：

```
static int spca5xx_open(struct video_device *vdev, int flags)
{
    ... ... ... ... ...
    /* 增加模块的使用计数 */
    MOD_INC_USE_COUNT;
    /* 初始化 USB 摄像头硬件设备 */
    err = spca50x_init_source (spca50x);
    /* 启动摄像头视频数据的同步传输 */
    err = spca50x_init_isoc (spca50x);
    ... ... ... ... ....
}
```

spca5xx_open()函数主要完成的工作是首先增加模块的使用计数，然后调用 spca50x_init_source() 函数打开硬件设备并对其进行初始化，再调用 spca50x_init_isoc()函数启动摄像头视频数据的同步传输。spca50x_init_isoc()函数通过调用 usb_alloc_urb()函数分配新的 URB 结构，然后填充新分配的 URB 结构，如将 URB 回调函数 spca50x_isoc_irq()函数赋给 URB 结构的 complete 字段，最后调用 usb_submit_urb()函数提交该 URB 完成视频数据传输的启动工作。

- 设备的关闭

在使用完 USB 设备后，必须关闭 USB 设备。对于用户来说，只要在应用程序中调用 close()函数，就会自动进入驱动程序在 video_device 结构中的 spca5xx_close()函数，其具体实现如下：

```
static void spca5xx_close( struct video_device *vdev)
{
    ... ... ... ... ...
    /* 停止摄像头视频数据的同步传输 */
    spca50x_stop_isoc (spca50x);
    /* 关闭 USB 摄像头硬件设备 */
    spcaCameraShutDown (spca50x);
    /* 减少模块的使用计数 */
    MOD_DEC_USE_COUNT;
    ... ... ... ... ....
}
```

spca5xx_close()函数与 spca5xx_open()函数正好相反，其先通过调用 spca50x_stop_isoc ()函数停止摄像头视频数据的同步传输，然后调用 spcaCameraShutDown ()关闭 USB 摄像头，最后减少模块的使用计数。

- 读取视频数据

如果需要采集摄像头捕获的视频数据，对于用户来说，只要在应用程序中调用 read()函数，就会自动进入驱动程序在 video_device 结构中的 spca5xx_read()函数，其具体实现如下：

```
static long spca5xx_read(struct video_device *dev, char * buf,
unsigned long count, int noblock)
{
    ... ... ... ... ...
    /* 等待一帧数据 */
    if((rc = wait_event_interruptible(spca50x->wq,
        spca50x->frame[0].grabstate == FRAME_DONE ||
        spca50x->frame[1].grabstate == FRAME_DONE ||
        spca50x->frame[2].grabstate == FRAME_DONE ||
        spca50x->frame[3].grabstate == FRAME_DONE ))){
            return rc;
    }
    /* 拷贝数据到用户空间 */
    if ((i = copy_to_user (buf, frame->data, count))){
```

```
            return -EFAULT;
        }
        ... ... ... ... ....
    }
```

spca5xx_read()函数首先会阻塞休眠在以 spca50x->wq 为队头的等待队列上。当有一帧数据准备好即 spca50x->frame[x].grabstate 变为 FRAME_DONE 时，spca5xx_read()函数会被唤醒。spca5xx_read()函数被唤醒后，会调用 copy_to_user()函数将捕获到的视频数据拷贝给用户进程。

3.4 USB 传输带宽冲突的研究

3.4.1 USB 摄像头与 USB 鼠标数据传输带宽冲突

在 USB 摄像头单独连接到系统、USB 鼠标单独连接到系统和 USB 摄像头与 USB 鼠标都连接到系统但不移动鼠标这三种情况下，系统和各个设备都能够正常工作。但如果 USB 摄像头和 USB 鼠标都连接到系统上，在摄像头采集视频数据的同时移动或者点击鼠标，则明显能够感觉到系统的性能下降，并且系统不停地输出“USB Schedule Overrun!!”这一出错信息，进而引起花屏、系统崩溃等现象的出现。

为了找到 USB 摄像头与 USB 鼠标数据传输冲突的原因，对 USB 摄像头与 USB 鼠标都连接到系统但不移动鼠标和 USB 摄像头与 USB 鼠标都连接到系统并移动鼠标这两种情况进行了实验。让应用程序取十帧视频数据，调用十次 read 系统调用，分别记录这两种情况下，摄像头驱动完成此十次 read 系统调用所需的总的 URB 个数和总时间，然后计算出 USB 摄像头驱动完成一次 read 系统调用平均所需的 URB 个数和时间。实验结果如表 3-1 所示：

表 3-1 带宽冲突实验结果 单位（毫秒）

移动鼠标	URB 数	时间	usb schedule overrun
否	45	450	不出现
是	152	1520	出现

从上表中可以看出，USB 摄像头和 USB 鼠标同时工作的情况下，完成一次 read 系统调用，取回同样大小的数据，需要的 URB 个数和占用的 CPU 时间是 USB 摄像头单独工作时的三倍。由于每个 URB 完成后都要拷贝数据和重发 URB，系

统在USB摄像头和USB鼠标同时工作的情况下的工作量也是USB摄像头单独工作时的三倍，大大地加重了系统的负担。

3.4.2 USB 摄像头与 USB 鼠标冲突的原因分析

USB Schedule 有两层含义：一层指的是 USB 设备与 USB 主机交互的一种工作方式，USB 提供了一种可共享的相互连接，通过 USB 总线与 USB 主机相连的各个 USB 设备都可以利用这种可共享的相互连接进行数据传输，为了支持同步数据传输和消除仲裁，对这种可共享的相互连接的占用应该提前预定[29]；另一层指的是帧时间分配表，主机控制器负责将 USB 时间划分为以 1 毫秒为单位的帧，一帧内要完成的传输工作由 USB Schedule 指定，其是由主机控制器维护的常驻内存表，由一系列的描述单个数据传输事务的 TD（Transfer Descriptor）队列组成[30]。

本课题采用的 S3C2410 芯片的 USB 主机控制器符合 OHCI 规范，OHCI 定义了一个名为主机控制器通信域（HOST Controller Communications Area）的通信通道。在 HCCA 中定义了 4 个链表：周期性数据链表、控制传输数据链表、批量传输数据链表和完成数据链表。主机控制器根据这 4 个链表来完成数据传输。正常情况下，USB 带宽预留安排如图 3-5 所示，

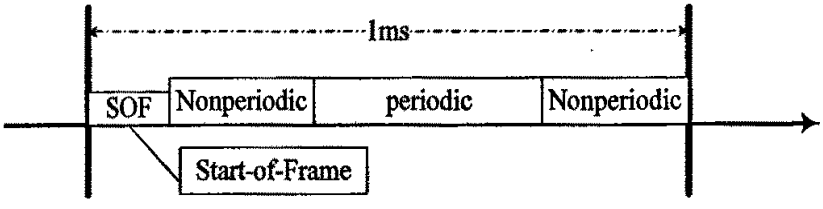

图 3-5　正常情况下一帧传输事务的安排

首先主机控制器会发送帧开始（Start-of-Frame）标识包，标识一帧的开始，若有需要主机控制器会先进行一些非周期性传输，然后进行必须处理的周期性传输，在周期性传输工作结束后，若还有剩余时间，主机控制器还会进行一些非周期性传输直到这一帧的结束[30]。如果在一帧内安排了过多了周期性工作，会引起 USB 带宽安排问题，如图 3-6 所示，

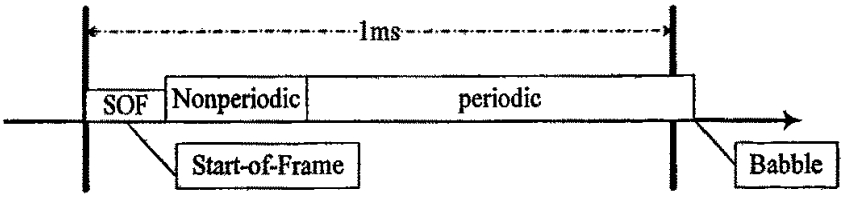

图 3-6　调度溢出时一帧传输事务的安排

主机控制器无法在一帧结束前完成预定的周期性传输工作，会发生超时干扰错误（Babble），一个调度溢出（Schedule Overrun）中断就会产生[31]。为了不破坏下一帧的开头，与设备相连的端口会自动关闭，并且数据接收方不会接收此次传输事务的数据[23]。

综上所述，当 USB 摄像头和 USB 鼠标同时工作时，要求的 USB 传输带宽超过了一帧所能提供给周期性传输工作的最大带宽，引发了超时干扰错误。由于带宽冲突频繁发生，摄像头驱动提交的 URB 无法带回期望的数据，要得到相同大小的数据，如表 3-1 所示，需要摄像头驱动提交更多的 URB，系统处理 URB 和拷贝数据的次数也随之增加，加重了系统的负担，再加上要处理超时干扰错误，造成系统性能下降，进而造成花屏和系统崩溃。

3.4.3 USB 摄像头与 USB 鼠标数据传输带宽冲突的解决办法

针对 USB 摄像头与 USB 鼠标传输冲突的原因，可以从减小冲突的影响和降低传输带宽要求这两个方面出发着手解决 USB 摄像头与 USB 鼠标的传输冲突问题。通过降低冲突发生的频率和增加 URB 的数据携带量可以缓解冲突带来的系统负担，避免花屏、死机现象，而降低带宽要求能够从根本上解决传输带宽冲突问题。

- 降低冲突发生的频率

USB 摄像头的同步数据 TD 和 USB 鼠标的中断数据 TD 都保存在 HCCA 的周期性数据链表中，在 OHCI 的定义中，此周期性数据链表实际上是由 32 个中断数据链表实现的。在每毫秒的 USB 数据传输中，主机控制器会依次循环发送其中一个中断数据链表上的所有 TD。如图 3-7 所示，这 32 个中断数据链表组成了一个平衡二叉树结构，通过将相关的 TD 链接到此二叉树的中间结点或者叶子结点上，可以实现 1 毫秒、2 毫秒、4 毫秒、8 毫秒、16 毫秒以及 32 毫秒 6 种不同时间间隔的周期性数据传输[32]。

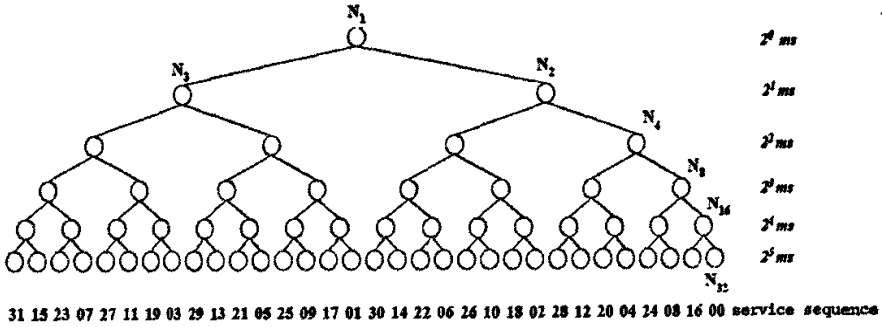

图 3-7 周期性数据链表结构

USB 摄像头和 USB 鼠标默认的数据传输间隔均为 1 毫秒，即在每个帧时内都有数据传输事务，造成了冲突频繁发生。通过调整中断传输和同步传输的时间间隔，错开 USB 摄像头与 USB 鼠标数据传输的时间点，能够降低 USB 摄像头与 USB 鼠标冲突发生的频率，减轻冲突带来的系统负担。

为了调整中断传输和同步传输的时间间隔，需要修改内核源码包中 driver/usb/usb-ohci.c 文件中的 ep_link()函数。此函数负责指定中断传输和同步传输的传输时间间隔，并将其赋给 ed->interval 变量，修改此变量即可重新指定中断传输和同步传输的传输时间间隔，重新指定传输时间间隔后还需要重新编译内核，生成新的内核镜像。调整了传输时间间隔后，再次进行了与 3.4.1 节相同的实验，由于避免摄像头缓存内的数据被新采集的图像数据覆盖，摄像头采集的数据应迅速传输到 USB 主机，因此，实验时只调整了中断传输的时间间隔。实验结果如表 3-2 所示：

表 3-2　带宽冲突实验结果　单位（毫秒）

中断间隔	URB 数	时间	URB 数 (移动鼠标)	时间 (移动鼠标)
1	45	450	152	1520
2	42	420	118	1180
4	42	420	97	970
8	43	430	82	820
16	45	450	71	710
32	41	410	62	620

从上表可以看出，随着中断传输间隔的不断加大，USB 摄像头和 USB 鼠标同时工作的情况下，完成一次 read 系统调用，取回同样大小的数据，需要的 URB

个数和时间也随之减小，说明摄像头与鼠标的冲突次数也在减少，冲突得到了有效的控制。

- 增加 URB 的数据携带量

当 USB 摄像头与 USB 鼠标发生传输冲突时，传输相同大小的数据所需的 URB 个数大幅增加，URB 传输完成后拷贝数据和重发 URB 的工作量也随之增加，加重了系统的负担。通过增加同步传输 URB 包含的同步数据包的个数能够让一个 URB 带回更多的数据，从而减少传输工作所需的 URB 个数，缓解冲突带来的系统负担。

同步数据包的个数由同步传输 URB 结构中的 number_of_packets 字段指定，URB 的初始化工作一般是设备类驱动完成的。为了增加同步传输 URB 包含的同步数据包的个数，需要修改 USB 摄像头驱动源码 spca5xx.h 文件中的 FRAMES_PER_DESC 宏。在摄像头驱动初始化同步传输 URB 的时候，根据此宏的取值来设置同步传输 URB 结构中的 number_of_packets 字段，其默认值为 10。调整了 FRAMES_PER_DESC 宏的值后，再次进行了与 3.4.1 节相同的实验，实验结果如表 3-3 所示：

表 3-3 带宽冲突实验结果 单位（毫秒）

数据包数	URB 数	时间	URB 数 (移动鼠标)	时间 (移动鼠标)
10	45	450	152	1520
16	33	520	89	1340
20	20	400	59	1180

从上表数据可以看出，随着同步数据包个数的增加，USB 摄像头和 USB 鼠标同时工作的情况下，完成一次 read 系统调用，取回同样大小的数据，需要的 URB 个数也随之减小，缓解了传输冲突带来的系统负担。

- 降低带宽要求

从降低冲突频率和增加 URB 的数据携带量这两个方面着手，可以缓解 USB 摄像头与 USB 鼠标传输带宽冲突带来的系统负担，花屏、死机现象消失了。但从实验结果来看，“USB Schedule Overrun!!” 这一报错信息仍然会出现，系统中的 USB 传输带宽冲突问题仍然存在。

由于摄像头驱动的带宽要求远远大于鼠标驱动的带宽要求，因此调整摄像头驱动要求的带宽，让其为鼠标驱动留出相应的带宽。为了降低摄像头驱动要求的带宽，需要修改摄像头驱动源码 spca5xx.c 文件中的 spca50x_set_packet_size（）

函数。此函数的主要作用是设置 spca50x 结构中的 packet_size 字段，摄像头驱动通过此字段来通知 HCD 为其预留 USB 带宽，其默认值为 1023 字节。调整了摄像头驱动的带宽要求后，再次进行了与 3.4.1 节相同的实验，实验结果如下表所示：

表 3-4　带宽冲突实验结果　单位（毫秒）

预留带宽	URB数	时间	URB 数(移动鼠标)	时间(移动鼠标)	usb schedule overrun
1023	45	450	152	1520	出现
959	45	450	47	470	不出现
785	47	470	49	490	不出现

从上表数据可以看出，当 packet_size 小于 960 时，完成一次 read 系统调用，取回同样大小的数据，需要的 URB 个数和摄像头单独工作时大致相等，并且“USB Schedule Overrun!!” 这一出错信息不再出现，USB 摄像头与 USB 鼠标数据传输带宽冲突不再发生。

为了综合三种解决办法的优势，分别从上述三中解决办法中抽取实验效果最好的一组，同时修改相应的字段，再次进行了与 3.4.1 节相同的实验，实验结果如表 3-5 所示：

表 3-5　带宽冲突实验结果　单位（毫秒）

中断间隔	数据包数	预留带宽	URB 数(移动鼠标)	时间(移动鼠标)	usb schedule overrun
32	20	959	23	460	不出现

从上表数据可以看出，综合三种解决办法后，完成一次 read 系统调用所需的 URB 个数仅为原来的一半，消耗的时间也与摄像头单独工作时大致相等，USB 摄像头与 USB 鼠标传输带宽冲突不再发生，其引起的花屏、死机等异常现象也不再出现。

第四章　嵌入式 Linux 下 PCI 设备驱动的设计与实现

在前面的章节中，讨论了 Linux 设备驱动的结构以及嵌入式 Linux 下 USB 摄像头驱动程序的设计与研究。本章则主要介绍 PCI 总线协议以及 PCI 设备的 Linux 驱动框架，并实现了流媒体数据缓存 PCI 卡的 Linux 设备驱动。

4.1 PCI 总线概述

PCI 是外围设备互连（Peripheral Component Interconnect）的简称，作为一种通用的总线接口标准，它在目前的计算机系统中得到了非常广泛的应用。PCI 提供了一组完整的总线接口规范，其目的是描述如何将计算机系统中的外围设备以一种结构化和可控化的方式连接在一起，同时它还刻画了外围设备在连接时的电气特性和行为规约，并且详细定义了计算机系统中的各个不同部件之间应该如何正确地进行交互[34]。

4.1.1 PCI 总线的体系结构

PCI 总线体系结构是一种层次式的（Hierarchical）体系结构。在这种层次式体系结构中， PCI 桥一类特殊的 PCI 设备，其将各个总线连接在一起，从而使整个系统看起来像一颗倒置的树型结构。典型的 PCI 总线体系如图 4-1 所示[35][36]：

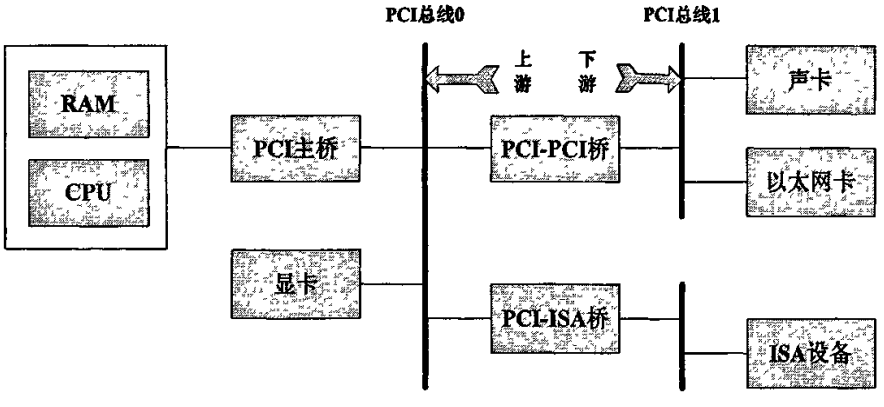

图 4-1 PCI 总线体系结构

从上图中可以看出，CPU 和 RAM 需要通过 PCI 主桥连接到 PCI 总线 0（也

称为主 PCI 总线）上，而具有 PCI 接口的显卡则可以直接连接到主 PCI 总线上。PCI-PCI 桥则负责将 PCI 总线 0 和 PCI 总线 1（也称为从 PCI 总线）连接在一起，通常 PCI 总线 1 称为 PCI-PCI 桥的下游，而 PCI 总线 0 则称为 PCI-PCI 桥的上游。图 4-1 中连接到从 PCI 总线上的是 SCSI 卡和以太网卡。为了兼容旧的 ISA 总线标准，PCI 总线还可以通过 PCI-ISA 桥来连接 ISA 总线，从而能够支持以前的 ISA 设备。

4.1.2 PCI 总线信号

PCI 总线设备分为主设备（master）和从设备（slave），主设备拥有总线控制权，是发起数据交换的一方，而从设备则响应主设备提出的数据交换请求。PCI 从设备至少需要 47 根信号线，PCI 主设备还需要 REQ#和 GNT#两个信号，共 49 根信号线。图 4-2 列出了 PCI 总线规范中的所有信号[37]。

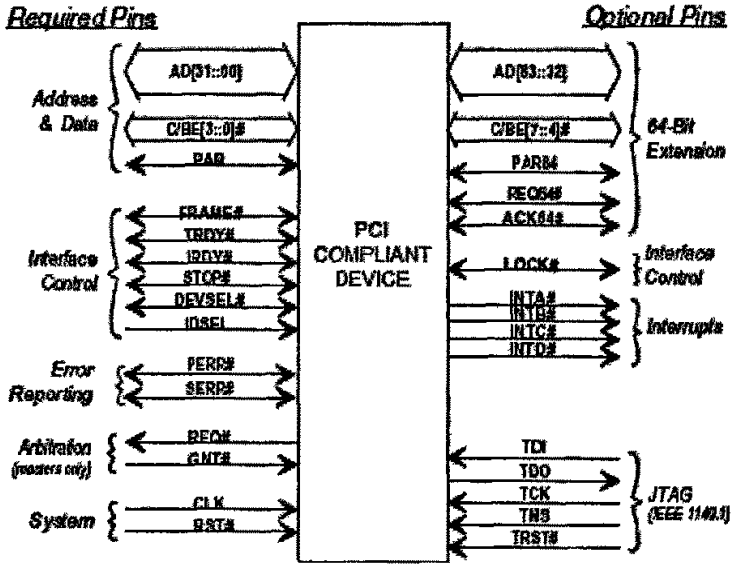

图 4-2　PCI 总线信号线

从上图可以看出，PCI 总线信号可以分为必选信号和可选信号两大类。表 4-1 给出了两大类信号以及功能。

表 4-1 PCI 总线信号及用途

	信号名	信 号 功 能 说 明
必备类信号	CLK	总线时钟线，提供同步时序基准。
	RST#	复位信号线，强制所有 PCI 寄存器和信号到初始态
	AD[31-0]	地址和数据复用线
	C/BE[3-0]	总线命令和字节有效复用线
	PAR	奇偶校验位线，对 AD[31-0]和 C/BE[3-0]#实施校验
	FRAME#	帧信号，当前主方驱动它有效以指示一个总线业务
	IRDY#	主方就绪信号，数据已到 AD 线上或已准备好数据
	TRDY#	目标方就绪信号，已准备好接受数据或数据已到 AD 线
	STOP#	停止信号，目标方要求主方中止当前总线业务
	DEVSEL#	设备选择信号
	IDSEL#	初始化设备选择或芯片选择
	REQ#	总线请求信号，主方送往中央仲裁器
	GNT#	总线授权信号，中央仲裁器送往主设备
	PERR#	奇偶错报告信号
	SERR#	系统错误报告信号
可选类信号	AD[63-32]	用于扩充到 6 4 位的地址、数据复用线
	C/BE[7-4]	总线命令和高 4 字节使能复用线
	REQ64#	用于请求 6 4 位传送
	ACK64#	目标方准许 6 4 位传送
	PAR64	对扩充的 AD 线和 C/BE 线提供偶校验
	INTA#	中断请求信号
	INTB#	中断请求信号（仅用于多功能设备）
	INTC#	中断请求信号（仅用于多功能设备）
	INTD#	中断请求信号（仅用于多功能设备）
	TCK	测试时钟
	TDI	测试输入
	TDD	测试输出
	TMS	测试模式选择
	TRST#	测试复位
	LOCK#	锁定信号，指示总线业务的不可分割性

4.1.3 PCI 总线配置

为了避免多个 PCI 设备竞争同一资源，PCI 总线提供了资源配置机制，能够让设备提出资源需要申请，系统收集所有的资源申请信息并综合分配设备所需要的资源，最后把资源分配信息写入到设备中去，以此作为设备访问的依据[38]。保存资源配置信息的寄存器称为配置寄存器，所有配置寄存器的整体就构成了配置空间。

每个 PCI 设备的配置空间至少有 256 个字节，其中前 64 个字节的配置空间头标区是标准化的，剩下的部分是与特定设备相关的。目前 PCI 2.3 规范定义了 3 种配置空间头标区格式，分别是 0、1 和 2 型。其中类型 0 为标准的 PCI 设备，类型 1 为 PCI-PCI 桥，类型 2 为 PCI-CardBus 桥，这里只对类型 0 的配置空间头标区加以说明，如图 4-3 所示[39]。

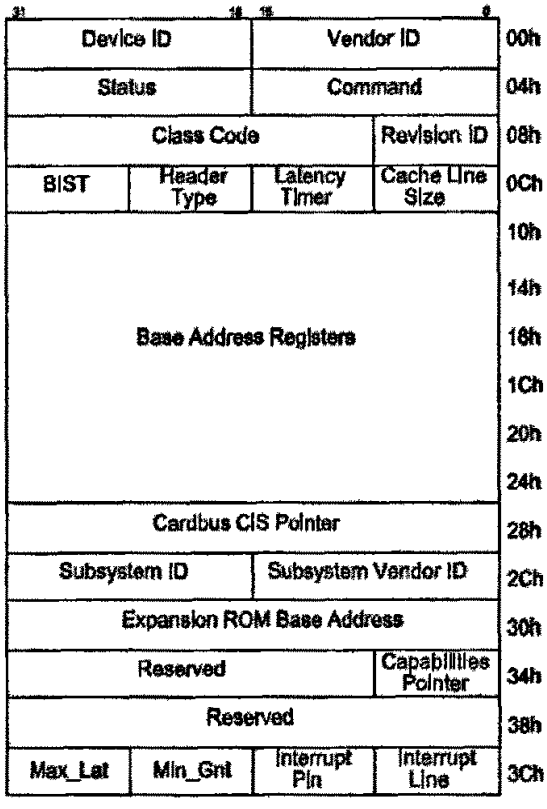

图 4-3 配置空间头标区

配置空间头标区包含了设备识别、设备控制、设备状态和基址寄存器四大类信息，每类信息都由相对应的配置寄存器保存，PCI 总线配置就是通过读写设备

的配置寄存器完成的。当系统加电时，配置软件会扫描 PCI 总线，确定存在哪些设备，针对设备的需求对资源进行分配，最后通过访问配置空间寄存器对设备进行配置。

4.2 Linux 环境下的 PCI 设备驱动程序

4.2.1 Linux 中 PCI 设备驱动的核心数据结构

Linux 内核提供了几个与 PCI 设备驱动程序开发直接相关的核心数据结构，它们在实现 PCI 的体系结构、构建 PCI 驱动程序的框架方面都起着非常重要的作用[40]。

- STRUCT PCI_DRIVER

pci_driver 数据结构是在 Linux 内核版本 2.4 之后为新型的 PCI 设备驱动程序所添加的，主要用于向 PCI 核心描述 PCI 设备驱动程序。为了正确地注册到 Linux 内核中，每个 PCI 设备驱动程序都必须构建 pci_driver 结构体，其定义如下：

```
struct pci_driver{
    const char *name;                          //PCI 驱动程序的名字
    const struct pci_device_id * id_table;     //驱动所支持的设备 ID
    int (* probe) (struct pci_dev *dev, const struct pci_device_id *id);
        //PCI 驱动程序中的探测函数
    void (* remove) (struct pci_dev *dev);
        //PCI 驱动程序中的移除函数
    ... ... ... ... ...
};
```

- STRUCT PCI_DEV

pci_dev 数据结构是对系统中所有 PCI 设备的抽象，PCI 核心通过该结构对连接到 PCI 总线上的所有 PCI 设备进行管理。这个数据结构详细描述了一个 PCI 设备几乎所有的硬件信息，包括厂商 ID、设备 ID、各种资源等等，其定义如下：

```
struct pci_dev{
    char  name[80];                //设备名
    struct pci_bus    *bus;        //指向 PCI 设备所在总线的 pci_bus 结构
    unsigned short    vendor;      //PCI 设备的厂商 ID
```

```
    unsigned short    device;                    //PCI 设备的设备 ID
    unsigned short    subsystem_vendor;          //PCI 设备的子系统厂商 ID
    unsigned short    subsystem_device;          //PCI 设备的子系统设备 ID
    unsigned int class;                          //PCI 设备的类别
    struct pci_driver *driver;          //设备所对应驱动的 pci_driver 结构
    struct resource resource[DEVICE_COUNT_RESOURCE];
        //设备可能用到的资源
    ... ... ... ... ...
};
```

4.2.2 PCI 设备类驱动接口分析

PCI 子系统与 USB 子系统相同，在扫描到新的 PCI 设备后，需要调用驱动程序实现的探测函数以查找与设备相匹配的 PCI 驱动程序。因此，除了与应用程序交互的接口外，PCI 驱动程序还需要提供与 PCI 核心交互的接口[41]。

- 与 PCI 核心交互的接口

PCI 驱动程序提供给 PCI 核心调用的接口函数主要是 probe()函数和 remove()函数。probe()函数是 PCI 驱动程序实现的探测和初始化函数，其主要负责检测设备，如果能够正常驱动该设备，还会进行一些初始化工作。当扫描到新的 PCI 设备后，PCI 核心会在内存中 pci_dev 类型的结构变量，然后将此变量作为参数传递给 PCI 驱动程序实现的 probe()函数来判断驱动程序与设备是否匹配。当 PCI 设备从系统中断开或者 PCI 驱动程序从内核中卸载时，PCI 核心会调用 remove()函数，其主要负责释放驱动程序申请过或者占用的资源。

- 与应用程序层交互的接口

PCI 设备驱动程序提供给应用程序调用的接口函数主要包括与设备的打开、释放、读写和控制操作相对应的 open()、close()、read()、write()和 ioctl()等入口点函数。由于根据设备不同的用途，PCI 设备可以分为字符设备和块设备两种类型，因此，这些入口点函数的函数指针会根据具体情况保存在 file_operations 结构或者 block_device_operations 结构中，再通过字符设备的注册函数或者块设备的注册函数提供给应用程序。本课题实现的流媒体数据缓存 PCI 卡属于字符设备。

4.3 流媒体数据缓存 PCI 卡的 Linux 设备驱动的设计与实现

在前面几节中，讨论了 Linux 中 PCI 设备驱动的核心数据结构和接口函数。本节将按照 Linux 设备驱动的设计和实现步骤来介绍流媒体数据缓存 PCI 卡的驱动程序的具体实现。

4.3.1 流媒体数据缓存 PCI 卡驱动的模块化编程子程序

在实现流媒体数据缓存 PCI 卡驱动的模块化编程子程序之前需要定义三个重要的静态全局变量 device_table[]、p2pcard_driver 和 p2pcard_fops，其中 device_table[]代表了当前驱动支持的 PCI 设备，p2pcard_driver 代表当前的 PCI 驱动程序，而 p2pcard_fops 则代表驱动提供给应用程序调用的设备文件操作入口点函数的集合。

```
static struct pci_device_id device_table[] = {
    {PCI_DEVICE(PCI_P2PCARD_VENDOR_ID,
PCI_P2PCARD_DEVICE_ID)},
    {0,}
};  //PCI_P2PCARD_VENDOR_ID 和 PCI_P2PCARD_DEVICE_ID 分别是流媒体数据缓存卡上 PCI 接口芯片的制造商 ID 和设备 ID。
static struct pci_driver p2pcard_driver = {
    name:       "p2pcard",
    id_table:   device_table,
    probe:      p2pcard_probe,
    remove:     p2pcard_remove,
};  //p2pcard_probe 和 p2pcard_remove 分别是探测函数和移除函数
static struct file_operations p2pcard_fops = {
    owner:      THIS_MODULE,
    open:       p2pcard_open,
    release:    p2pcard_release,
    read:       p2pcard_read,
    write:      p2pcard_write,
    ioctl:      p2pcard_ioctl
};  //设备文件操作入口点函数集合
```

定义了静态全局变量 device_table[]、p2pcard_driver 和 p2pcard_fops 之后，接下来就是模块装载函数 p2pcard_init_module() 和模块移除函数 p2pcard_cleanup_module ()的实现。

```
static int __init p2pcard_init_module(void)
{
    ... ... ... ... ...
    /* 注册硬件驱动程序 */
    rc = pci_register_driver(&p2pcard_driver);
    if(rc <= 0) {
        printk(KERN_ALERT "Can't register p2pcard as a pci device!\n");
        pci_unregister_driver(&p2pcard_driver);
        return -ENODEV;
    }
    /* 注册字符设备驱动*/
    rc = register_chrdev(MAJOR_NUM, "p2pcard", &p2pcard_fops);
    if(rc < 0) {
        printk(KERN_ALERT "Can't register p2pcard as a chr device!\n");
        return rc;
    }
    ... ... ... ... ...
}
static void p2pcard_cleanup_module(void)
{
    /* 注销硬件驱动程序 */
    unregister_chrdev(MAJOR_NUM, "p2pcard");
    /* 注销字符设备驱动*/
    pci_unregister_driver(&p2pcard_driver);
}
```

p2pcard_init_module()主要完成的工作是用&p2pcard_driver 作为参数调用 pci_register_module()函数向 PCI 子系统注册 PCI 设备驱动和用&p2pcard_fops 作为参数调用 register_chrdev()函数注册为字符设备驱动。而 p2pcard_cleanup_module() 则主要是用&p2pcard_fops 作为参数调用 unregister_chrdev()注销字符设备驱动以

及用&p2pcard_driver 作为参数调用 pci_unregister_driver()函数从 PCI 子系统中卸载 PCI 设备驱动。

4.3.2 流媒体数据缓存 PCI 卡驱动的探测和移除子程序

在扫描到新的 PCI 设备后，PCI 子系统需要调用设备驱动程序实现的探测函数以查找与设备相匹配的 PCI 驱动。本课题实现的 PCI 设备驱动的探测函数的具体实现如下：

```
static int p2pcard_probe(struct pci_dev *dev, const struct pci_device_id *idt)
{
    struct pci_p2pcard *p2pcard = NULL;          //特定设备实例
    /* 为与设备相对应的实例分配存储空间 */
    p2pcard = kmalloc(sizeof(struct pci_p2pcard), GFP_KERNEL);
    if(p2pcard == NULL){
        printk(KERN_ERR "Out of memory!\n");
        rc = -ENODEV;
        goto out;
    }
    /* 激活 PCI 设备 */
    if((rc = pci_enable_device(dev)) != 0){
        printk(KERN_ERR "Can't enable the pci device!\n");
        goto err_dev;
    }
    /* 初始化特定设备实例的私有化数据 */
    spin_lock_init(p2pcard->lock);
    pci_read_config_byte(dev, PCI_REVISION_ID, (u8 *)&(p2pcard->rev_id));
    printk(KERN_ALERT "the pci device revision is: %d\n", p2pcard->rev_id);
    p2pcard->pdev = dev;
    p2pcard->irq = dev->irq;
    p2pcard->user = 0;
    /* 读取 I/O 资源的配置信息*/
    p2pcard->mem_base = pci_resource_start(dev, 0);
```

```
        p2pcard->mem_size = pci_resource_len(dev, 0);
        p2pcard->io_base = pci_resource_start(dev, 1);
        /* 申请 I/O 区域 */
        if((rc = pci_request_regions(dev, "p2pcard")) != 0){
            printk(KERN_ALERT "Can't request the IO address region!\n");
            goto err_disable;
        }
        /*设置成总线主模式*/
        pci_set_master(dev);
        /* I/O 内存映射 */
        p2pcard->mem_start = (unsigned long) ioremap(p2pcard->mem_base,
                    p2pcard->mem_size);
        ... ... ... ... ...
    }
```

与 p2pcard_probe() 函数相对应的是 PCI 设备驱动的移除函数 p2pcard_remove()，其具体实现如下：

```
    static void p2pcard_remove(struct pci_dev *dev)
    {
        ... ... ... ... ...
        /* 释放掉映射的虚拟地址 */
        iounmap((void *)p2pcard->mem_start);
        /* 释放 I/O 区域 */
        pci_release_regions(dev);
        /* 关闭 PCI 设备 */
        pci_disable_device(dev);
        /* 释放为设备实例分配的内核空间 */
        if(!p2pcard->user) {
            memset(p2pcard, 0, sizeof(*p2pcard));
            kfree(p2pcard);
            p2pcard = NULL;
        }
    }
```

4.3.3 流媒体数据缓存 PCI 卡驱动的服务于 I/O 请求的子程序

流媒体数据缓存卡驱动的服务于 I/O 请求的子程序主要包括设备的打开、关闭，读写和控制等函数，如前所述，这些操作都是通过 file_operations 结构提供给应用程序的。

- 设备的打开

在使用 PCI 设备之前，必须先打开所要使用 PCI 设备。当用户在应用程序中调用 open()函数时，应用程序就会自动进入驱动程序中的 p2pcard_open()函数。p2pcard_open()函数主要负责增加模块的使用计数，并根据 p2pcard_probe()读到的中断号申请中断，注册中断处理程序，其具体实现如下：

```
static int p2pcard_open(struct inode *inode, struct file *file)
{
    ... ... ... ... ...
    /* 增加模块的使用计数 */
    MOD_INC_USE_COUNT;
    /* 申请中断，注册中断处理程序 */
    if((rc = request_irq(p2pcard->irq, p2pcard_interrupt, SA_SHIRQ,
            "p2pcard", p2pcard)) != 0) {
        goto err_irq;
    }
    ... ... ... ... ....
}
```

- 设备的关闭

在使用完 PCI 设备后，必须关闭 PCI 设备。当用户在应用程序中调用 close()函数时，应用程序就会自动进入驱动程序中的 p2pcard_release()函数。p2pcard_release()函数完成主要工作是释放中断和减少模块的使用计数，其具体实现如下：

```
static int p2pcard_release(struct inode *inode, struct file *file)
{
    ... ... ... ... ...
    /* 释放中断 */
    free_irq(p2pcard->irq, p2pcard);
```

```
    /* 减少模块的使用计数 */
    MOD_DEC_USE_COUNT;
    …… …… …… …… ……
}
```

● 设备的读写

当用户在应用程序中调用 read()函数和 write()函数对设备文件进行读写操作时，应用程序就会自动进入驱动程序中的 p2pcard_read()函数和 p2pcard_write()函数，其具体实现如下：

```
static ssize_t p2pcard_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    …… …… …… …… ……
    /* 等待读取数据 */
    if((rc = wait_event_interruptible(p2pcard->wq,
        p2pcard->state == READY ))){
        up(&spca50x->lock);
        return rc;
    }
    /* 将设备内存中的数据拷贝到内核空间 */
    if (count > p2pcard->mem_size)
        count = p2pcard->mem_size;
    memcpy_fromio(pbuf, p2pcard->mem_start, count);
    …… …… …… …… ……
    /* 将数据从内核空间拷贝到用户空间 */
    if ((i = copy_to_user (buf, pbuf, count))){
        return -EFAULT;
    }
    …… …… …… …… ……
}
```

p2pcard_read()函数首先会阻塞在以 p2pcard->wq 为队头的等待队列上。当流媒体数据缓存卡上的数据准备好即 p2pcard->state 变为 READY 时，p2pcard_read()函数会被唤醒。spca5xx_read()函数被唤醒后，会先将数据从设备 I/O 内存拷贝到内核空间，再从内核空间拷贝给用户进程。

```
static ssize_t p2pcard_write(struct file *file, const char *buf, size_t count, loff_t
*ppos)
{
    ... ... ... ... ...
    /* 将数据从用户空间拷贝到内核空间 */
    if ((i = copy_from_user (pbuf, buf, count))){
        return -EFAULT;
     }
    ... ... ... ... ...
    /* 将设备内存中的数据拷贝到内核空间 */
    if (count > p2pcard->mem_size)
        count = p2pcard->mem_size;
    memcpy_toio(p2pcard->mem_start, pbuf, count);
    ... ... ... ... ...
}
```

p2pcard_write()函数的主要工作是将数据从用户进程拷贝到内核空间，再将内核空间中的数据拷贝到设备 I/O 内存。

4.3.4 流媒体数据缓存 PCI 卡驱动的中断服务子程序

流媒体数据缓存卡驱动中的中断处理程序为 p2pcard_interrupt()函数。在本课题的实现中，p2pcard_interrupt()函数主要负责识别中断、响应中断和唤醒睡眠的进程，其具体实现如下：

```
static void p2pcard_interrupt(int irq, void * dev_id, struct pt_regs *regs)
{
    ... ... ... ... ...
    /* 识别中断 */
    status = inl(p2pcard->iobase + P2PCARD_INT_STA);
    if( ! (status & INT_MASK))
        return;
    /* 响应中断 */
    outl(status & INT_MASK, p2pcard->iobase + P2PCARD_INT_STA);
```

```
    /* 唤醒睡眠进程 */
    if (waitqueue_active (&p2pcard->wq))
        wake_up_interruptible (&p2pcard->wq);
    ... ... ... ... ...
}
```

第五章 总结

5.1 全文总结

本课题中将嵌入式 Linux 系统的构建以及 USB 摄像头和流媒体数据缓存 PCI 卡的驱动开发作为重点，围绕着嵌入式技术、USB 总线协议、PCI 总线协议和嵌入式 Linux 下的驱动程序开发的相关知识，进行了大量的学习和研究，最终完成开发并研究解决了在项目开发中所遇到的嵌入式 Linux 平台上 USB 传输带宽冲突问题。具体完成的工作内容如下：

1. 研究和探讨了嵌入式 Linux 系统的组成和设计步骤，给出了基于 S3C2410 芯片的 FS2410 开发板上实现嵌入式 Linux 系统的具体方案。

2. 分析了 Linux 内核以及与设备驱动程序的关系，根据 Linux 设备驱动的结构提出了 Linux 设备驱动程序的设计和实现流程。

3. 学习和研究了 USB 总线协议，设计和编写基于 ZC301P 芯片的 USB 摄像头的嵌入式 Linux 驱动，并研究解决嵌入式 Linux 平台上 USB 传输带宽冲突问题。

4. 学习和研究了 PCI 总线协议，设计和编写流媒体数据缓存 PCI 卡的 Linux 设备驱动。

5.2 进一步工作

虽然本文的工作取得了初步成功，完成了相关设备驱动程序的开发，但是由于 USB 摄像头和流媒体数据缓存 PCI 卡的实时性要求较高，数据传输量较大，驱动程序的性能还有待进一步提高。同时，随着 Linux 内核 2.6 版本在嵌入式系统开发中应用得越来越广泛，对支持 2.6 内核版本的嵌入式 Linux 设备驱动程序的需求也越来越大。下一步研究工作的重点，一是完善驱动开发方法，提高驱动程序的性能，二是将本文实现的 USB 摄像头和流媒体数据缓存 PCI 卡驱动程序移植到 Linux 2.6 内核上。

致谢

在本论文即将完成之际，谨此向我的导师李毅教授致以衷心的感谢和崇高的敬意！本论文的工作是在李老师的悉心指导下完成的。李老师以他敏锐的洞察力、渊博的知识、严谨的治学态度、精益求精的工作作风和对科学的献身精神给我留下了刻骨铭心的印象，这些使我受益匪浅，并将成为我终身献身科学和献身事业的动力。在攻读硕士的这三年里，导师不仅为我创造了优越的科研和学习环境，使我得以在计算机科学领域中自由翱翔，同时在思想上、人生态度和意志品质方面给予了谆谆教诲，这些教益必将激励着我在今后的人生道路上奋勇向前。

真诚感谢教研室的周小兵博士和师兄李益明硕士，他们不仅在学术上给我指引，而且在生活上予以帮助，从他们身上我学到很多知识。感谢项目组成员赵磊硕士、王春毅硕士、张博硕士、向文杰硕士、郑建勋硕士、王祥雒硕士和刘超硕士在项目开发中的互助合作，正是集体的努力才使得项目进展顺利。

由衷感谢我的室友同学刘丹硕士、寻伟硕士，他们开创性的研究拓展了我的学术视野，无数次的争论和探讨使我的研究工作有了长足的进展。

衷心的感谢我的父母和其他亲朋好友对我的关心、支持和理解，没有他们对我的关心、鼓励和支持，我无法完成现在的硕士学业。

最后，感谢曾经教育和帮助过我的所有老师。衷心地感谢为评阅本论文而付出宝贵时间和辛勤劳动的专家和教授们！

参考文献

[1] 李善平，刘文峰，王焕龙. Linux 与嵌入式系统[M]. 清华大学出版社,2003,2-4

[2] 钱磊. 中国 Linux 市场现状与发展趋势[J]. 中国政府采购, 2005/8

[3] Linux 操作系统介绍[EB/OL], http://www.tech163.net/xt/i/33307.html

[4] 邓冬梅，杨铁林. 嵌入式系统和 Linux[J]. 计算机与现代化, 2004/12

[5] 黄力，覃纪武. 嵌入式 Linux 的现状与前景研究[J]. 电脑知识与技术, 2005/15

[6] 孙天泽，袁文菊，张海峰.嵌入式设计及 Linux 驱动开发指南－基于 ARM9 处理器[M]. 电子工业出版社,2005,216～219

[7] 袁俊杰，曹作良. 基于 Linux 嵌入式系统开发平台的建立[J]. 天津理工大学学报, 2006/03

[8] San-Pil Moon; Joo-Won Kim; Kuk-Ho Bae; Jae-Cheon Lee; Dae-Wha Seo. Embedded Linux implementation on a commercial digital TV system[J]. Consumer Electronics, IEEE Transactions, Volume 49, Issue 4, Nov. 2003

[9] Haifeng He; John Trimble; Somu Perianayagam; Saumya Debray; Gregory Andrews. Code Compaction of an Operating System Kernel[J]. Code Generation and Optimization, 2007. CGO '07. International Symposium, March 2007

[10] Guo Peiyuan; Chen Wenhua; Bao Man; Chen Yan; He Xiaoyan. Design and Implementation of Flash File System in S3C44B0X Embedded System[J]. Innovative Computing, Information and Control, 2006. ICICIC '06. First International Conference, Volume 2, 30-01 Aug. 2006

[11] Chen Tianzhou; Hu Wei; Wang Xiangsheng. Smart file system: embedded file system based on NAND-flash[J]. Networking, Architecture, and Storages, 2006. NAS '06. International Workshop, 1-3 Aug. 2006.

[12] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux 设备驱动程序 第 3 版[M]. 魏永明，耿岳，钟书毅，译. 中国电力出版, 2005.11, 13~15, 331~332

[13] 陈莉君. Linux 操作系统内核分析[M]. 人民邮电出版社,2000, 11~12

[14] 毛德操，胡希明. Linux 内核源代码情景分析（上）[M]. 浙江大学出版社，2001，235~238

[15] 赵炯. Linux 内核完全注释[M]. 机械工业出版社，2004，25~28

[16] 钱晨，徐荣华，王钦若. 基于 Linux 操作系统的设备驱动程序开发[J]. 微计算机信息, 2004/09

[17] Yong Hoon Choi, Woo Il Kwon, Heung Nam Kim. Code generation for Linux device driver[J]. Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference, Volume 2, 20-24 April 2006

[18] Hongfeng Zhu, Tianhua Liu, Chuansheng Zhou, Guiran Chang. Research and Implementation

of Zero-Copy Technology Based on Device Driver in Linux[J]. Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums, Volume 2, 20-24 April 2006

[19] De Goyeneche J.-M., De Sousa E.A.F. Loadable kernel modules[J]. Software, IEEE, Volume 16, Issue 1, Jan.-Feb. 1999

[20] 董志国，李式巨. 嵌入式 Linux 设备驱动程序开发[J]. 计算机工程与设计，2006/20

[21] 梁金千，张跃，甄成. Linux 设备驱动程序架构的研究[J]. 计算机工程与应用，2002/08

[22] Kouyama T, Nano H, Kon, C, Shimizu, N. The design of a USB device controller IYOYOYO[J]. Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific, 21-24 Jan. 2003

[23] USB 2.0 Universal Serial Bus Specification [S]. http://www.usb.org.

[24] Proffitt J, Hammond W, Majewski S, Popov V, Raylman R R, Weisenberger A G, Wojcik R. A flexible high-rate USB2 data acquisition system for PET and SPECT imaging[J]. Nuclear Science Symposium Conference Record, 2005 IEEE, Volume 5, 23-29 Oct. 2005

[25] 赖娟，洪艳伟. 基于 Linux 的 USB 摄像头驱动程序设计[J]. 成都教育学院学报，2006/08

[26] 刘胜军，高济. 嵌入式 Linux 下 USB 主控制器驱动的研究[J]. 现代机械，2006/03

[27] 武甲东，陈新华，张志敏. Linux 下 USB 主机控制器驱动的设计实现[J]. 山东科技大学学报(自然科学版)，2005/02

[28] 杨伟，刘强，顾新. Linux 下 USB 设备驱动研究与开发[J]. 计算机工程，2006/19

[29] C Y Huang, L P Chang, T W Kuo. A Cyclic-Executive-Based QoS Guarantee over USB[J]. In IEEE 9th Real-time and Embedded Technology and Applications Symposium (RTAS' 03), 2003.

[30] Alberto Martinez. USB Latency Requirements and the Effect of Video Adapter PCI Retry Condition on Maintaining USB Streaming Pipelines [R]. Sacramento: California State University, 1997.

[31] 周立功，ISP1161A1：通用串行总线单片主机和设备控制器[EB/OL]. http://www.zlgmcu.com/philips/usb/ISP1161/ISP1161A1_02DataSheet.pdf, (2006-8-20).

[32] 陈青松，王健. 通用串行总线数据传输[J]. 计算机工程与设计，2006/11

[33] 郑智. 嵌入式 Linux 下 USB 驱动程序开发研究[J]. 武汉理工大学学报(信息与管理工程版)，2006/07

[34] Tom Shanley，Don Anderson(美)，刘晖等译. PCI 系统结构[M]. 电子工业出版社. 2000.7

[35] Knudsen P.V, Madsen J. Integrating communication protocol selection with partitioning in hardware/software codesign[J]. System Synthesis, 1998. Proceedings. 11th International Symposium, 2-4 Dec. 1998

[36] Linux下PCI设备驱动程序开发[EB/OL].http://www.ibm.com/developerworks/cn/linux/l-pci/

[37] PCI 2.3 PCI local Bus Specification [S]. http://www.pcisig.com

[38] Bishop W, Grant D, Loucks W. Configurable coprocessing with an ARC-PCI board[J]. Electrical and Computer Engineering, 2004. Canadian Conference, Volume 2, 2-5 May 2004

[39] Chame A. PCI bus in high speed I/O systems applications[J]. Aerospace Conference, 1998. Proceedings., IEEE, Volume 4, 21-28 March 1998

[40] 董春桥，李凯. Linux 系统 PCI 设备驱动程序开发[J]. 计算机测量与控制，2005/11

[41] 李娟，胡和平，郭耀辉，詹骞. PCI 设备卡的 Linux 驱动程序设计[J]. 计算机工程，2003/04

[42] 穆华俊. Linux 设备驱动程序设计实例[J]. 电子与电脑，2000/04

[43] 张宏伟. Linux 下 USB 设备驱动程序的编写[J]. 计算机应用研究，2001/09

[44] 巍骞，张焕强，方贵明. 基于 Linux 的 USB 驱动程序实现[J]. 计算机应用，2002/08

[45] 蔡震，蒋辉柏，周利华. Linux 系统下 USB 设备驱动程序的开发[J]. 计算机测量与控制，2002/08

[46] 郑伟，王钦若，吴乃优. Linux 内核空间设备驱动程序的开发[J]. 微计算机信息，2003/12

[47] 陆尔东，邓利平，符意德，杨旭光. Linux 下的设备驱动及其管理机制[J]. 计算机应用，2003/01

[48] 郑桦，刘清，邢航，徐智穹. 构造嵌入式 Linux 的文件系统[J]. 微计算机信息，2004/08

[49] 霍妍，孟凡荣. 基于 Linux 嵌入式系统的研究与实现[J]. 计算机系统应用，2004/08

[50] 陈俊楷，冯穗力，叶梧. Linux 下 PCI 设备驱动程序研究[J]. 计算机应用研究，2002/11

攻硕期间取得的研究成果

[1] 程科 嵌入式 Linux 平台上 USB 传输带宽冲突的研究. 计算机应用研究, 2007,11

嵌入式Linux设备驱动程序的设计与研究

作者： 程科
学位授予单位： 电子科技大学

本文链接：http://d.g.wanfangdata.com.cn/Thesis_Y1080927.aspx