摘要

随着信息化技术的发展和数字化产品的普及,以计算机技术、集成电路技术和软件技术为核心的嵌入式系统得到了广泛的应用。然而,随着嵌入式设备的广泛应用和深入发展,功能日益强大的嵌入式系统所要求的外接设备越来越丰富,于是就出现了外接设备的识别和驱动问题。由于嵌入式备的种类和特点决定了嵌入式产品在开发时必须设计与自己设备相配套的驱动程序,所以怎样在特定的设备中开发自己的驱动程序成为本文研究的重心。

Linux 操作系统以其开放源代码、易于开发、功能强大稳定、可裁剪和成本低等特点,迅速成为当今嵌入式系统的主流开发平台。

嵌入式平台下 linux 系统的外围设备驱动分为字符型设备、块设备驱动和网络设备驱动,本文主要研究了字符设备和块设备的驱动开发的一般技巧和方法,在此基础之上以 USB 驱动为例，简要介绍了 USB 驱动的开发流程。最后通过 mini2440 开发板实现了 USB 接口与 GPIB 接口之间的通信。

关键词: 嵌入式 Linux，S3C2440，驱动程序，USB　GPIB

ABSTRACT

With the development of the information technology and Popularization of digital Products,the embedded system has been widely used with the computer technology, chip technology and software technology as the core. However, with the spreading and development of embedded equipment, it gets more powerful And requires various external equipments.There occur the problems such as Identity and driving of the external equipments. It is necessary to have the Matched driving Program in developing the embedded Product for the types and Features of the embedded system,so how to explore the matched driving Program for special equipment is the core of the research paper.

Linux system becomes rapidly the main exploration for embedded equipment due to the features of being easily explored, Powerful functions,stability,being cuttable and low cost.

The driver of external equipment equipped with Linux system can be divided into character equipment driver, bulk equipment driver and network equipment driver. it mainly introduces the general tactics and methods of developing the driver of character equipment and bulk equipment.It studies the development

Keywords: Linux, S3C2440, Drivers USB GPIB

目 录

第一章 引言

1.1 选题的背景与意义

随着大规模数字集成电路和通讯技术的高速发展,嵌入式系统的应用已经成为IT 行业的一个焦点,在自动控制、工业生产、军事技术和家用消费电子等各个领域都有着广泛的应用。

在当前的电子产业中,桌面 PC 的发展已经进入了平稳时期,而嵌入式技术高速兴起已经使人们提出了“后 PC 时代”的概念,计算机无处不在,无时不有,它渗透到工作和生活的各个方面。这个计算机不再是传统意义上的桌面 PC,而是诸如智能手机、个人数字助理、机顶盒、路由器等等。

开发国人自主知识产权的嵌入式处理器、嵌入式操作系统和嵌入式高端产品对于我们国家的民族 IT 业来讲,有着十分重要的战略意义。

1.2 Linux 的内核应用和研究现状

1991 年,Linus Torvalds 为当时使用 Intel 80386 微处理器的计算机开发了一种全新的操作系统内核,linux 由此诞生。现在,linux 内核由全世界内核开发社区维护和开发,并把最新的经过测试的版本发布在 http://www.kernel.org 上。从 1991 年的 linux0.0.2 版本到现在的 2.6.16 版本,linux 支持的处理器种类在不断增加,内核本身也不断完善。

从八十年代末开始,陆续出现了一些嵌入式操作系统,如 Vxwbrks、Palmos、WindowsCE、Nucleus 等。在这众多的嵌入式操作系统中,linux 以其具备的稳定、高效、易定制、硬件支持广泛、源代码开放等特点,在嵌入式系统领域近两年内迅速崛起,被国际上许多大型的跨国企业用作嵌入式产品的系统平台。现在国外基于嵌入式 linux 系统的产品有:韩国三星公司的 linux PDA 与可以联网的 linux 相机,美国 Transmeta 公司的 linux 手机,NetGem 公司的机顶盒,IBM 的 linux 腕表等,另外嵌入式 linux 还用于手持设备、交换机、防火墙等。国内的嵌入式 linux 厂商也在迅速发展壮大中,主要的嵌入式 linux 厂商有中软、红旗、蓝点、网虎科技等。产品主

要有中软股份开发的中软嵌入式 linux 操作系统,具有微秒级的强实时功能,已经在数据处理领域得到了很好的应用,红旗的嵌入式 linux 已经在机顶盒、彩票机等方面获得了成功。

随着 linux 内核的不断发展和完善,对 linux 的研究和应用主要有以下几个方面。

1：内核主版本的不断更新

这部分研究工作由全世界内核开发社区与 Linus 承担,主要改进现有版本的不足,为内核的几个主要模块,如进程管理、文件系统、存储管理等实现新的算法或增加新的特性。

2：其它特定应用的内核版本

由于 linux 主版本内核是基于 PC 平台的,所以在把 linux 内核应用于各种嵌入式系统时需要按照具体的要求进行改进,特别是在实时应用场合,在这个方向发展了一些实时的 linux 内核版本,如:rt-linux、kurt-linux 等,而为了在一些没有 MMU 硬件支持的芯片上运行 linux 又发展了如 uclinux 的内核版本。

3：内核的移植与驱动开发

Linux 要应用于嵌入式系统,首要的任务就是系统移植和驱动开发。在这个方面,主要有各大芯片厂商和网络上许多优秀的开源社区提供对 linux 移植的支持,比较有名的有 armlinux project,powerPC projeet 和 mips projeet 等。Linux 的移植可以分为两个方面,其一是和具体芯片架构相关的代码,如内存管理、中断管理部分,这部分也叫做 CSP,其二是系统上硬件驱动的移植,也称为 BSP。

4：上层应用

Linux 内核在嵌入式领域的广泛应用离不开大量的上层应用支持,在这个方面国内比较出名的有 MiniGui,国外有 QT 等大型的图形窗口系统以及其它应用库。

1.3 本课题研究的内容

在深入了解 USB 体系结构和 Linux 内核的基础之上,使用了 Samsung 公司内部集成的 USB 主机控制器和 USB 设备控制器 S3C2440 芯片作为处理器,这为研究和设计 USB 驱动程序提供了方便。本课题是在广州友善之臂计算机科技有限公司的 S3C2440 处理器 Mini2440 硬件开发平台上作了嵌入式 Linux-2.6.32.2 的系统构建,完成了 USB 接口的 USB Mass Storage 设备的识别。

1.4 论文结构

第一部分简述了嵌入式的概念及发展趋势以及 S3C2440 的内部结构。

第二部分详细的论述了 Linux 下，驱动开发的一般原理和方法。

第三部分是以 USB 驱动为例，简述了 USB 驱动的开发流程。

第四部分是基于嵌入式 Linux 下的接口之间的通信，本文以 USB 接口和 GPIB 接口为例。

第二章 嵌入式概述及 S3C2440 的内部结构

2.1 嵌入式系统介绍

2.1.1 嵌入式系统定义

嵌入式系统的定义[1]:以计算机技术应用为基础,软硬件均可方便裁减,适用于应对功能、成本、布局、体积、功耗等有严格要求的专用计算机应用系统。微型机以小型、价廉、高速数值计算等特点迅速走向市场,它所具备的智能化在工业控制领域发挥了重要作用,常被组装成各种形状“嵌入”到某个对象体系统,进行某类智能化的控制,这样一来,原来的计算机便失去了原来的形态与通用的功能,为了区别于通用的计算机系统,将这类为了某个专用目的而嵌入到对象体系中的计算机系统,称为嵌入式计算机系统,简称嵌入式系统。含有嵌入式系统的设备称为嵌入式设备,在生活中随处可见:比如电子表、MP3 播放器、摇控器等,涵盖生产、工业控制、通信、网络、消费电子、汽车电子、军工等各个领域。从广义的角度来说,除了超级计算机等具有较强大的计算能力及系统资源(比如内存、存储器等)的电子系统之外,凡具备计算能力的设备都可以称为嵌入式设备。

2.1.2 嵌入式系统的特点

嵌入式设备常应用于特定的场合,与通用的个人 PC 相比,特点如下[2]:

1、嵌入式系统是面向具体的特定的应用对象。

嵌入式系统是为特定的用户群设计的系统,他把许多集成芯片集成到自己内部,具有能耗低、体积小、集成度高等特点,把许多本来由 CPU 板卡完成的任务集成到自己内部,这有利于嵌入式系统的小型化和移动,方便更紧密的藕合网络。

2、嵌入式系统是根据各行业实际需要把计算机技术、半导体技术和电子技术结合在一起的产物。因此他必然是技术和资金密集,与时俱进的知识集成系统。
嵌入式系统要求硬件和软件设计高效,根据需要裁减,力争提高硅片单位面积的利用率,实现更多的功能,这样才能方便的选择处理器。

3、嵌入式系统是和实际的应用有机结合的产物,他的升级换代是由产品的升级换代决定的,因此生命周期较长。

4、嵌入式系统软件一般储存在存储芯片或单片机中,并不放在磁盘等载体中,目的是为了提高执行速度和系统的可靠性。

5、嵌入式系统本身不能进行自我开发,只能在一定的工具和环境下才能进行开发,不能被用户修改。

2.1.3 嵌入式系统的发展

随着信息时代、数字时代的到来,使得嵌入式产品得到了广泛应用,他们有了美好的发展前景,同时也对嵌入式提出了新的挑战,从中可知嵌入式系统发展的几大趋势[3]:

1、嵌入式广泛应用于通信和消费类产品的开发,在通信领域,数字技术正在全面取代模拟技术。

2、联网成为必然趋势

要求系统标配以适应嵌入式分布处理结果和应用上的需求,硬件方面,不仅有各大公司的微处理器芯片,还有用于学习和研发所配置的网络接口,为了满足联网,嵌入式设备必须具备通讯接口和相应的 tcp/ip 协议软件的支持,新的嵌入式设备具有 iEEE1394,usb,can,bluetooth 等通讯接口,以满足现场协调工作的需要,同时也需要提供相应的通讯组网协议软件和物理层驱动。为了编程的需要可能还需要用到浏览器,如 HTMLHEWML 等。

3、小型化低成本低功耗

为了满足人们大众化和便捷的需要,就要求设备必须向小的方向发展;为了满足用户长时间工作的需要,就要求系统向低功耗和轻便化方向发展,而嵌入式的可裁减正好可以满足上述要求。

4、提供精巧的多媒体人机界面

嵌入式设备之所有被接受就是由于他们与使用者之间的亲和力,自然的人机交互界面。例如手写文字输入、语言拨号、彩信、电子邮件等,现在一些 PDA 已经实现汉字写入,短信语言发布,但离语言同声翻译还有很大的距离。

2.2 Linux 系统概述

2.2.1 Linux 简介

1991 年,linux 操作系统核心最早是由芬兰的赫尔辛基大学的 linux torvald 在芬兰大学上学时发布的,1994 年 linux 加入了 GNU,由于其许可条款的约定吸引众多顶

尖软件工程师对其修改和完善,这样 Linux 得以在全球广泛使用,在服务器领域及个人桌面领域得到越来越多的应用[4]

Linux 是 GNT 公共许可权限下免费获得的,是一款符合 POSI 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 以其高效和灵活的模块化的设计结构而著称,使得它能够在任何 PC 机上实现全部的 Unix 特性。它拥有自己的公司,能向用户提供一套完整的服务,这使得它特别适合在公共网络中使用,Linux 当前有很多发行版本,较流行的有：redhatlinux,ubuntu,debian 等。

2.2.2 Linux 的优势

集成电路技术的飞速发展为嵌入式系统的开发从单片机开发时代到系统开发时代提供了条件,这就要求嵌入式操作系统的出现。从国内外看,嵌入式操作系统在世界范围内主要有 windows CE,VxWorks,psos,QNX,Palmox,Lynxos 等,Linux 是一个成熟、稳定、可靠的操作系统,嵌入式操作系统在开发方面具有其它系统无可比拟的优势。经过这些年的发展,已成为主要的嵌入式开发平台之一。Linux 作为嵌入式操作系统的优势在于[5]:

1、跨平台的硬件支持

用 C 语言编写,采用了可移植的 unix 标准应用程序接口,可以支持多个系统平台和嵌入式在内的各种硬件设备。

2、丰富的软件支持

与其他系统不同,Linux 系统安装后用户常用的一些办公软件、图形处理工具、多媒体播放器和网络工具都无需安装,包含多种程序语言与开发工具,如 gcc,perl 等。

3、多用户任务

Linux 系统是一个真正的多用户多任务的操作系统,每个用户可以同时使用系统资源,互不影响,多个用户可以在同一时间以网络联机的方式使用计算机系统,由于 linux 系统调度每一个进程访问处理器的权限是相同的,所以他可以同时执行多个相互独立程序。

4、可靠的安全性

Linux 系统是一个具有先天病毒免疫的系统,很少受到病毒的攻击。

对一个开方式系统来说,在方便用户的同时,很可能存在安全隐患,不过利用 Linux 自带的防火墙、入侵检测和安全认证等工具及时修补系统的漏洞,就能大大提高 Linux 系统的安全性,让黑客们无机可乘。

5、良好的稳定性

Linux 内核的源代码设计是以标准规范的 32 位机来做的,可确保系统的稳定性,由于 Linux 的稳定,才使得一些安装 linux 的主机像俪 Unix 机一样常年不关而不曾宕机。

6、完善的网络功能

Linux 内置了很丰富的免费网络服务软件,数据库和网页的开发工具,如 Sendlmail,VSFtp,MYSQL,php 和 jsp 等。近年来越来越多的企业看到了 linux 的这些强大的功能,利用 Linux 担任全方位的网络服务器。

2.3 S3C2440 的内部结构

2.3.1 ARM 的种类及特点

1、ARM7 微处理器

ARM7 内核是冯·诺依曼体系结构,数据和指令使用的是同一条总线。内核的一条 3 级流水线用来执行 ARMv4 指令集。ARM7 系列微处理器是一种 32 位 RSC 处理器,功耗低,最适用于对价位和功耗要求比较高的消费类的应用。ARM7 微处理器系列的具体特点如下[6]:

(l)具有嵌入式 ICE-RT 逻辑,调试;

(2)功耗低,适合低功耗应用,比如便携式产品;

(3)兼容 16 位的 Thumb 指令集,代码密度高;

(4)支持 Windows CE、Linux 等操作系统;

(5)主频可达到 130MIPs,能适应于各种高速的运算处理;

(6)ARM7 系列微处理器的应用于工业控制、网络设备和调制解调器等多种嵌入 式应用。ARM7 系列微处理器有以下几种:ARM7TDMI、ARM72OT、ARM7EJ。其中,ARM7TMDI 使用的是最广泛的 32 位嵌入式 RSC 处理器,属低端 ARM 处理器核。

2、ARM9 处理器

ARM9 系列微处理器具有低功耗和高性能的特点。具有以下特点[7]:

(1)5 级整数流水线指令执行,效率更高;

(2)支持 16 位 Thumb 指令集和 32 位 ARM 指令集;

(3)可以支持犯位 AMBA 总线接口;

(4)具有全性能的 MMU,支持 Linux 等多种主流嵌入式操作系统;

(5)具有数据 Cache 和指令 Cache,能够更快的指令和数据。

ARM9 主要应用于无线设备、高端打印机、机顶盒、仪器仪表和数字产品等。ARM9 系列有 ARM920T、ARM922T 和 ARM940T 三种处理器类型,来满足不同的应用需求。

3、ARM10 处理器

ARM10E 系列微处理器功耗低,性能高,具有新的体系结构,与 ARM9 相比较,在同样的时钟频率下,性能高出了 50%。ARM10E 系列微处理器的主要特点如下[7]:

(l)支持 DSP 指令集,应用于高速数字信号处理;

(2)6 级整数流水线,指令被高效的执行;

(3)支持 32 位 ARM 指令集和 16 位 Thumb 指令集;

(4)支持 32 位 AMBA 总线接口;

(5)支持 VFP10 浮点处理;

(6)全性能的 MMU,支持 Linux、等多种嵌入式操作系统;

(7)支持数据 Cache 和指令 Cache,指令和数据处理能力强;

(8)主频最高可达 400MIPS;

(9)内嵌并行读/写操作部件。

ARM10E 系列微处理器主要应用于下一代成像设备、无线设备、工业控制、通信数字消费品等领域。该系列包含 ARM1020E、ARM1022E 和 ARM1026EJ-S 三种微处理器类型，来满足不同的应用场合的需求。

2.3.2 S3C2440 微处理器的主要结构

S3C2440 的频率是 400MHz,其中 ARM92OT 的两个内部协处理器分别是起到调试控制作用的 CP14 和起存储系统控制及测试控制作用的 CP15,ARM920T 核的组成部分是高速缓存、存储管理单元(MMU)存等(其中,高速缓存的组成部分是独立的 16KB 地址和 16KB 数据高速 Cache,MMU 用于管理虚拟内存)。

2.3.3 S3C2440 片内资源

S3C2440 有丰富的片内资源，主要包括[8]:

(1)1 个 LCD 控制器(支持 STN 和 TFT 液晶显不屏);

(2)SDRAM 控制器;

(3)3 个 UART 通道;

(4)4 个 DMA 通道;

(5)4 个具有 PWN 功能的计时器和 1 个内部时钟;

(6)触摸屏接口；

(7)IC 总线接口；

(8)3 个 USB 接口；

(9)SD 卡接口和 MMC 接口；

(10)看门狗计数器；

(11)117 个通用 I/O,24 个外部中断源；

(12)8 通道 10 位 AD 控制器。

2.3.4 S3C2440 的体系结构

在通用嵌入式应用中,S3C2440 处理器可以提供多种片上集成系统的解决方案，主要包括[9]：

(1)16/32 位 RSC 结构和 ARM920T 指令集；

(2)加强的 MMU,支持 WinCE 和 Linux；

(3)采用 ARM920TCPU 内核,支持 ARM 调试；

(4)支持高速和异步总线模式,支持外部等待信号延长总线周期。

2.3.5 S3C2440 的片内资源详解

S3C2440 有丰富的内部资源,S3C2440 所集成的接口可运用于各种场合,例如医疗设备、移动设备以及工业控制设备等。这些接口使用特别方便,与硬件的连接方式简单,一般通过软件控制寄存器的操作方式来实现。

1、Cache 高速缓存

Cache 高速缓存是 64 项的全相连模式,具有每行 8 字长度的 I-Cache(16KB)和 D-cache(16KB),每行都带有一个有效位和两个 dirty 位。主存储器通过伪随机数或转循环替换算法位采用写穿式(write-through)或写回式(write-back)cache 操作来更新的。写缓冲器可以存储 16 个字数据和 4 个地址。

2、时钟和电源管理

S3C2440 的时钟有以下几种模式[10]:片上 UPLL 和 MPLL；操作 USB 主机设备的时钟采用 UPLL 产生的；MCU 所需要的时钟是通过 MPLL 产生的最大 400MHZ@1.3V 来操作的;每个功能模块可以有选择性地通过软件提供时钟；电源模式:正常、慢速、空闲和掉电模式;慢速模式:不加 PLL 的低时钟频率模式；空闲模式:只停止 CPU 的时钟；掉电模式:切断了所有外设和内核的电源；在掉电模式中处理器是用 EINT[15:0]或 RTC 报警中断来唤醒的。

时钟电源控制块由三部分组成[10]:USB 控制、电源控制和时钟控制。S3C2440 中的时钟控制逻辑能生成所以所需时钟信号。HCLK 应用于高级高精度系统的总线外围设备。FCLK 应用于 CPU 的时钟,53C2440 有两个锁相环,一个用于 HCLK、PCLK 和 FCLK,另一个则用于 USB 块(48MHz)。时钟控制逻辑能产生没有锁相环(PLL)的
慢时钟及链接或不链接提供给外围设备块的软件时钟,这样就可以降低功耗。

关于电源控制逻辑,S3C2440 的多种电源配置方案保证了实现既定任务的最佳功能。在 S3C2440 中,电源控制块能被激活成四种模式:慢模式、掉电模式、通用模式和空闲模式[11]。

3、中断控制器

中断控制器的特点如下[12]:

S3C2440 有 60 个中断源:1 个看门狗定时器,5 个定时器,9 个 UARI,24 个外部中断 EINT,4 个 DMA,2 个 RTC,2 个 ADC,l 个 IIC，2 个 SPI,1 个 SDI,2 个 USB,1 个 LCD 和 1 个电池故障,1 个 NAND 和 2 个 CAMERA,1 个 AC97 音频。外部中断源可以电平/边沿触发或可编程的边沿/电平触发。S3C2440 中断控制器提供快速中断服务为紧急中断请求。

上面这些中断源中,URAT、AC97 及外部中断是和中断控制器做“或”运算得到的。当从内部设备或者外部中断请求管脚上收到多个中断请求时,中断控制器会在仲裁判决以后,向 ARM920T 内核发出 FIQ 或者 IRQ 的请求。如果 ARM92OT 的程序状态寄存器(PSR)的 F 位值为 1,那么 CPU 就不会接收来自中断控制器的快速中断请求(FIQ)。同样,如果 PSR 得 I 位值为 1,CPU 就不会接收来自中断控制器的中断请求(IRQ)。所以,必须将寄存器 PSR 的 F 位和 I 位清零以
后,同时设置相应中断掩码(INTMSK)寄存器为。以后,中断控制器才一可以接收相应的中断。

4、脉冲带宽调制定时器（PWM）

脉冲带宽调制定时器的特点[13]:

S3C2440 有 5 个 16 位定时器。定时器 0、1、2、3 具有脉冲宽度可调(PWM)功能。定时器 4 是没有输出管脚的内部定时器。定时器 0 有一个死去发生器,可以和一个大电流装置一起使用。

定时器 0 和 1 共用一个 8 位的分频器,而定时器 2、3、4 共用一个 8 位的分频器。每个定时器有一个时钟分频器,能产生 5 个不同的分频信号(1/2,1/4,1/8,1/16 和 TCLK)。每个定时器块接收它自己的来自时钟分频器的时钟信号,而这个时钟分频

信号来自相应的 8 位分频器。8 位分频器是可编程的,根据寄存器 TCFGO 和 TCFG1 的值,可将 PCLK 时钟信号进行分频。

当定时器使能以后,定时器计时器缓冲寄存器(TCNTBn)里有一个初始值存放在递减计数器中。定时器比较缓冲寄存器(TCMPBn)里有一个初始值存放在比较寄存器中,来和递减计数器的值比较。当频率和占空比发生改变的时候,这样由 TCNTBn 和 TCMPBn 组成的双缓冲器的特点使计时器能够产生一个稳定的输出。

每个定时器有它自己的 16 位递减计数器,由定时器时钟驱动。当递减计数器达到 O 时就会产生一个定时器中断请求,以便通知 CPU 定时器操作已经结束。当定时器计数器达到 O 时,相应的 TCNBn 中的值就会自动地存放到递减计数器中,用来继续下面的操作。然而,如果定时器停止时,例如,在定时器运行模式下,清空定时器寄存器的 TCONn 使能位,TCNTBn 中的值就不会再次存放到计数器中。

TCMPBn 的值是用来脉冲宽度调节的。当递减计数器的值和比较寄存器中的值匹配时,时钟控制逻辑就会改变输出电平。所以,比较寄存器决定了 PWM 输出的开(或者关)的时间。S3C2440 的 PWM 部分具有以下几个特点[14]:5 个 10 位计时器;2 个 8 位的分频器和 2 个 4 位的分频器;可编程占空比以控制输出波形;自动重载模式和单脉冲模式;死区发生器。

5、实时时钟

实时时钟(RTC)单元在系统电源关闭以后,能够依靠后备电池运行。RTC 能够使用 STRB/LDRB 的 ARM 操作,以二进制编码的十进制(就是 BCD 码)向 CPU 发送 8 位的数据。这些数据具有全面的时钟特性:秒、分、时、日期、星期、月和年。其工作频率为 32.768kHZ,具有报警中断和节拍中断功能。RTC 单元的特点包括以下几个[16]:BCD 编码:秒、分、时、日期、星期、月和年;闰年发生器;报警功能:报警中断,或者断电模式下的唤醒功能;千年虫问题已经解决;独立的电源管脚(RTCVDD);

6、通用 I/O 端口

S3C2440 有 130 个多功能输入/输出端口,具体如下[15]:

(l)端口 A(GPA):25 个输出端口;

(2)端口 B(GPB):11 个输入 z 输出端口;

(3)端口 C(GPC):16 个输入 z 输出端口;

(4)端口 D(GPD):16 个输入 z 输出端口;

(5)端口 E(GPE):16 个输入/输出端口;

(6)端口 F(GPF):8 个输入/输出端口;

(7)端口 G(GPO):16 个输入/输出端口；

(8)端口 H(GPH):9 个输入 z 输出端口；

(9)端口 J(GPJ):13 个输入/输出端口。

每个端口可以很方便地通过软件进行配置,以满足不同的配置和设计需要。用户在启动主函数时,必须对每一个端口定义一个功能。如果一个端口没有用于多功能端口,就必须被配置为输入/输出端口。

7、UART 控制器

S3C2440 的串口控制器提供了三个独立的输入输出端口,每一个端口都能够在基于中断模式或DMA 模式下工作。也就是说,UARI 控制器能够产生中断或者 DMA 请求,便于在 CPU 和 UARI 端口之间传送数据。在利用系统时钟的情况下,UART 控制器支持的最高波特率达到 115200。如果一个外部设备提供给 UART 控制器一个外部时钟,那么 UART 控制器就能够以更高的传送速度运行。每个 UART 通道包含两个 64 字节的 FIFO 用来发送和接收数据。

S3C2440 的 UART 控制器包含有一个波特率发生器、发送器、接收器和控制单元。波特率发生器能够从 PCLK,FCLK 或者 UEXTCLK(外部输入时钟)得到时钟。发送和接收器包括 64 字节的 FIFO 和数据移位器。数据发送过程是:先将数据写到 FIFO 中,再复制到发送移位器中准备发送,最后,发送数据管脚(TxDn)将数据移位出去。相应地,数据接收过程是:先将数据从接收管脚(RxDn)上面得到,然后将从移位器中得到的数据复制到 FIFO 中。

S3C2440 的串口控制器具有以下的特点:RxD0、TxD0、RxD1、TxD1、RxD2 和 TxD2 基于 DMA 或者中断模式运行；UART 通道 0 和 1 有 nRTS0、nCTS0、nRTS1 和 nCTS1；支持外部时钟作为 UART 的运行时钟(UEXTCLK)；可编程的波特率；每个通道都具有内部 64 字节的发送 FIFO 和 64 字节的接收 FIFO。

8、看门狗定时器

如果受到外部噪声或者系统错误的干扰,S3C2440 的看门狗定时器就会恢复控制操作。它能作为一个 16 位步长的定时器请求中断服务。其在定时器溢出时发生 128 个 PCLK 时长的中断请求或系统复位信号。时长是 128 个 PCLK。看门狗计时器有以下几个特点:正常步长的计时器模式,带有中断请求；当计数器的值为 O 时(时间用完),激活内部 128 个 PCLK 时长的复位信号。

看门狗计时器只用 PCLK 作为源时钟。PCLK 的周期分频以后用来产生相应的看门狗计数器时钟,得到的结果继续频分。

分频值和频分因子由看门狗控制寄存器(WTCON)中的值来决定。频分因子可

以选择为 16、32、64 或者 128。

9、USB 设备控制器

S3C2440 有两个 USB 主设备接口和 1 个 USB 从设备接口。两个主设备接口具有这样的特点:兼容 USBver1.1；兼容 OHCIverl.0 规格；支持低速和全速 USB 设备。

一个 USB 从设备的特点是[18]:支持控制、中断和块传输；全速 USB 设备控制器兼容 USBverl.1 规格；全速模式下能够达到 12Mbps；支持 DMA 接口,用于发送和接收快数据(EP1、EP2、EP3 和 EP4)；5 个带有 FIFO 的 Endpoint。其中一个是双向控制 endpoint,带有 16 字节 FIFO 的(EP0)；另外四个是双向的快 endpoint,带有 128 个字节的 FIFO(EP1、EP2、EP3 和 EP4)；支持休眠和远程唤醒功能。集成的 USB 发送接收器；接收和发送的缓冲是独立的 128 字节的 FIFO 存储器,以达到最大的吞吐量；

10、工作电压

S3C2440 的内核工作在 30OMHz 时,核心电压为 l.2V；工作在 400MHz 时,核心电压是 1.3V。内存支持 1.8V/2.5V/3.0V/3.3V,输入/输出为 3.3V。

第三章 Linux 下的设备驱动开发

Linux 设备驱动开发并不遵循特定的流程,本文介绍的开发驱动流程并不针对某个特定的具体硬件,而是通用的驱动开发过程和规则,是理解和开发具体硬件驱动的基础。

3.1 Linux 设备的分类和驱动概述

Linux 设备被分为三类[16]:块设备、字符设备和网络设备。本文主要研究字符设备和块设备的驱动。

Linux 设备一般分为三类[16]:块设备、字符设备和网络设备。字符设备是一般不需要通过缓冲区而直接将字符数据进行读写操作的设备。块设备则一般要通过缓冲区把以数据块为单位的输入输出数据传送于设备与内存之间。网络设备一般是指将数据使用通信网络传输的设备,例如网络适配器(网卡)等。

3.1.1 设备驱动程序的作用

设备驱动程序完成系统对设备的控制和操作,并且将硬件的细节屏蔽了,这样应用程序就可以对只作为一个设备文件的硬件设备进行操作。

设备驱动程序包括中断处理程序和设备服务子程序。设备服务子程序包含各种操作设备的代码,中断处理子程序是对设备中断进行处理。

设备驱动程序的主要操作流程是:首先初始化设备,对设备的运行进行启动或停止操作,然后将设备上的数据传送到内存,再从内存传送到设备,最后进行检测设备状态的操作。

设备驱动程序的特点如下[17]:与设备相关的；由内核统一管理其代码；是在具有特权级的内核态下运行；是输入输出子系统的一部分；其服务于某个进程,该进程的运行过程包含了驱动程序的执行过程；如果执行程序需要设备的某种状态还没有达到,当前进程就会被其阻塞,纳入该设备的等待队列里。

3.1.2 访问设备的实现步骤

(1)识别设备

使用设备类型、主设备号和次设备号对设备进行识别。其中设备类型是指字符设各或者块设备;主设备号是与驱动程序相对应的,即设备使用的驱动程序不同,设备赋予的主设备号就不同;次设备号是用来区分使用同一驱动程序的各个具体设备。

（2）对相应设备文件进行操作

设备文件一般放在/dev 目录下,虚拟文件系统 VFS 被 Linux 系统用来作为统一操作接口处理文件和设备。与普通的目录和文件一样,每个设备也使用一个 VFSinode 来描述,其中包含着该种设备的主次设备号。

对设备进行操作也就是驱动程序的设备服务子程序通过对文件进行操作的 file_operations 结构调用出来。向设备输出数据、启动设备、停止设备运行的操作是通过使用 file_operations 中的 write()、open()、close()来分别完成的。

3.1.3 Linux 设备控制方式

Linux 的 I/O 控制方式有三种[28]:查询等待方式、中断方式和 DMA(直接内存存取)方式。

(l)查询等待方式

查询等待方式又称轮询方式(Polling mode)。对于不支持中断方式的机器只能采用这种方式来控制 I/O 过程,所以 Linux 中也配置了查询等待方式。例如,并行接口的驱动程序中默认的控制方式就是查询等待方式。

(2)中断方式

在硬件支持中断的情况下,驱动程序可以使用中断方式控制 I/O 过程。对 I/O 过程控制使用的中断是硬件中断,当某个设备需要服务时就向 CPU 发出一个中断脉冲信号,CPU 接收到信号后根据中断请求号 IRQ 启动中断服务例程。在中断方式中,Linux 设备管理的一个重要任务就是在 CPU 接收到中断请求后,能够执行该设备驱动程序的中断服务例程。

(3)DMA 方式

DMA 是一种无须 CPU 的参与让外设与系统 RAM 之间进行双向数据传输的硬件控制。使用 DMA 可以使系统 CPU 从实际的 I/O 数据传输过程中摆脱出来,从而大大提高系统的吞吐率。由于 DMA 是一种硬件机制,因此它通常与硬件体系结构相关,尤其依赖于外设的总线技术。

DMA 由 DMA 控制器(DMA Controller,简称 DMAC)来完成整个数据传输过程。在此期间,CPU 可以并发地执行其他任务,当 DMA 结束后,DMAC 通过中断通知 CPU 数据传输已经结束,然后由 CPU 执行相应的 ISR 进行后处理。

3.2 字符设备驱动的开发

一组完成不同任务的函数集合是由 linux 下的设备驱动程序来组织的,通过些函数可以使应用程序操作同文件操作一样,对只作为一个设备文件的硬件设备进行

操作。字符设备的驱动开发在下面会进行具体介绍:

每个设备驱动的第一步是初始化模块函数一般用如下函数对设备进行初始化

```
static int_init chrdev_init(void)
{
if(register_chrdev(MAJOR_NUM,“chardev,&chrdev_fops))
{
/*注册失败*/
}
else
{
/*注册成功*/
}}
```

MAJOR_NUM 在 registe_chrdev 的参数中为主设备号,其中“chardev”为设备名,chrdev_fops 的类型是 file_operations,是包含基本函数入点口的结构体,在用户访问时,为其提供入口指针。

模块卸载是与模块初始化函数相对应的,需要调用 chardev_exit(void

```
Static void_exit chrdev_exit(void)
if(unregisterer_chrdev(MAJOR_NUM,“ehardev))
{
/*卸载失败*/
else
{
/*卸载成功*/
}}
```

虽然 file_oprations[29]结构体很大,但是只有其中一部分被大多数驱动程序使用,给字符设备提供的主要入口有:open(),release(),read(),write(),ioctl(),llseek(),poll()等。

open()函数

```
int(*open)(struct inode*,struct file*)
```

参数 inode 为设备文件的 inode 结构的指针,file 是指向这一设备的文件结构体指针。open()的主要任务是确定硬件就绪设备合法,次设备号可以用(inede 一>i-rdev)取得,控制设备的进程,根据执行情况返回状态码。

release()函数

当最后一个打开设备的用户进程执行 close()系统调用时,将调用 release()函数:

void(*release)(struct inode*,structfile*);

该函数主要是清除没结束的输入输出操作,释放资源。

read()函数

ssize_t(*read)(struct file*,char*,size_t,loff_t*);

当 read()函数指针被赋予 NULL 时将导致 read 系统调用报错,函数返回的非负值为读取的字节数。

write()

ssize_t(*write)(struct file*,char*,size_t,loff_t*)

函数返回的非负返回值为写入的字节数。

ioctl()函数

int(*ioctl)(struct inode*,struct file*,unsigned int,unsigned long*);

unsigned int 参数为序要执行的命令编码,unsigned long 为相应的命令提供参数,设备如果不提供 ioctl 入口,则对内核任何未预定的请求,ioctl 将返回错误。

llseek()函数

跳转到指定的位置,并将新位置作为返回值。

Loft_t(*llseek)(struct file*,loff_t int)

设备 chardev 的驱动程序的这些函数名分别被命名为 chardev_open, ehardev_release, chardev_read, chardev_write, chardev_ioctl,因此 chardev 的基本入口的结构变量 chardev_fops 赋值如下:

```
struet file_operations chardev_fops={
open:chardev_open
release:chardev_release
read:chardev_read
write:chardev_write
ioctl:chardev_ioctl
};
```

3.3 块设备驱动的开发

块设备是与字符设备并列的概念, 在 Linux 中这两类设备驱动的结构有很大区

别,与字符设备驱动相比,块设备驱动在 I/O 操作上有很大的差异,块设备驱动要用到请求队列、缓冲、I/O 调度等都概念.

3.3.1 块设备的 I/O 操作特点

字符设备与块设备 I/O 操作的区别[18]:

块设备只能以块为单位输入输出,而字符设备则以字节为单位。并且大多数设备都是字符设备,因为它们的操作不需要使用缓冲区且不以固定块大小进行。

块设备对于 I/O 请求可以选择以什么顺序来响应对应的缓冲区,字符设备无须缓冲且被直接读写。调整读写的顺序对于存储设备作用巨大,因为对连续的扇区读写比分离的扇区更快。

块设备可以随机访问,而字符设备只能被顺序读写。对于磁盘设备,有序的组织块设备的访问可以提高性能。

3.3.2 块设备驱动用到的函数和结构体

1.block_device_operations 结构体

在块设备驱动中,有 1 个类似于字符设备驱动中 file_perations 结构体的 block_device_operations[18]它是对块设备操作的集合,定义代码如下:

```
struct block_device_operations
{
int(*open)(struet inode*,sturet file*); /*打开*/
int(*release)(struet inode*,struet file*); /*释放*/
int(*ioctl)(struet inode*,stuet file*,unsigned,unsigned long); /*ioctl*/
long(*unlocked_ioetl)(struet file*,unsigned,unsigned 1ong);
long(*compat_ioetl)(struet file*,unsigned,unsigned long);
int(*dircet_aceess)(struct block_device*,sector_t,unsigned long*);
int(*media_changed)(struct,gendisk); /*介质被改变?*/
int(*revalidate_disk)(struct gendisk*); /*使介质有效*/
int(*getgeo)(struct block_device*,struct hd_geometry*); /*填充驱动器信息*/
struct module*owner;     /*模块拥有者*/
};
```

2. gendisk 结构体

在 Linux 内核中,使用 gendisk(通用磁盘)结构体来表示一个独立的磁盘设备(或分区),该结构体的定义代码如下:

struct gendisk[18]

```
{
 int major; /*主设备号*/
 int first_minor; /*第 1 个次设备号*/
 int minors; /*最大的次设备数,如果不能分区,则为 1*/
 char disk_name[32]: /*设备名称*/
 struct hd_struct**part; /*磁盘上的分区信息*/
 struct block_device_operations*fops; /*块设备操作结构体*/
 struct request_queue*queue:;/*请求队列*/
 void*private_data; /*私有数据*/
 sector_t capacity; /*扇区数,512 字节为 1 个扇区*/
 int flags;
 char devfs_name[64」;
 int number;
 struct device *driver_fsdev;
 struet kobject;kobj;

 struet timer_rand_state*random;
 int policy;

 atomic_t sync_io; /*RAID*/
 unsigned long stamp;
 int in--flight;
  # ifdefCONFIG_SMP;
 struct disk_stats*dkstats;
 # else
 Struct disk_stats dkstats;
 # endif
 };
```

Linux 内核提供了一组函数来操作 gendisk,主要包括:

(l)分配 gendisk

gendisk 结构体是一个动态分配的结构体,它的初始化需要通过特别的内核来操作,这个结构体不能被驱动自己分配,分配 gendisk 使用下列函数来实现:

struct gendisk*alloc_disk(int minors);

(2)增加 gendisk

分配了 gendisk 结构体后,这个磁盘还需要注册系统才能正常使用,而磁盘注册通过下面函数来实现:

void add_disk(struct gendisk*gd);

(3)释放 gendisk

如果一个磁盘不需要了,就可以使用 gendisk 函数来释放;

void del_gendisk(struct gendisk*gd);

(4)设置 gendisk 容量

扇区是块设备中的基本单元,是块设备寻址和操作的最小单元,但是许多块设备能够一次访问多个扇区。扇区大小是设备的物理属性,扇区大小一般是 2 的整数倍,最常见的大小是 512 字节,当然还有很多常见其他大小的扇区,比如,很多 CD-ROM 盘的扇区都是 2KB 大小。

void set_capacity(struct gendisk*disk, sector_t size);

3.request 结构体

request 结构体涉及内核能否高效处理 I/O 请求,必须对其每一个成员的含义和作用充分的理解。

(1)request 结构体的定义

在 Linux 块设备驱动中,使用 request 结构体来描述等待进行的 I/O 请求,该结构体的定义代码如下[19]:

```
struct request
{
struct list_head queuelist;          /*链表结构*/
unsigned long flags;                 /*REQ*/
sector_t sector;                     /*要传送的下一个扇区*/
unsigned long nr-sectors;            /*要送的扇区数目*/
/*当前要被完成的扇区数目*/
unsigned int hard_cur_sectors;
struct bio*bio;                  /*请求的 bio 结构体的链表*/
Struct bio*biotail;               /*请求的 bio 结构体的链表尾*/
void*elevator_private;
unsigned short ioprio;
```

```
int rq_status;
struct gendisk*rq_disk;
int errors;
unsigned long start_time;
/*请求在物理内存中占据的不连续段的数目,scatter/gather 列表的尺寸*/
unsigned short nr_phys_segments;
/*与 nr_phy_segments 相同,但考虑了系统 I/O MMU 的 remap*/
unsigned short nr_hw_segments;
int tag;
char*buffer;                    /*传送的缓冲,内核虚拟地址*/
 int ref_count;                 /*引用计数*/
...
};
```

(2)request 结构体的主要成员

request 结构体的主要成员包括:

```
Sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
```

上述三个成员标识还未完成的扇区,hard sector;是第一个尚未传输的扇区,hard nr_sectors 是待完成的扇区数,hard_cur_sectors 是当前 I/O 操作中待完成的扇区数。这些成员只用于内核块设备层,驱动不应当使用它们。驱动中会经常与下面三个成员打交道;

```
sector_t sector;
unsigned long nr_sectors;
unsigned int current_nr_sectors;
```

这三个成员在内核和驱动交互中发挥着重大作用。它们以 512 字节大小为一个扇区,如果硬件的扇区大小不是 512 字节,则需要进行相应的调整。

(3)块请求队列

一个块请求队列是一个块 I/O 请求的队列,其定义代码如下[20]:

```
struct request_queue
{
    ...
```

```
/*保护队列结构体的自旋锁*/
spinlock_t_queue_lock;
spinlock_t *queue_ock;
/*队列 kobject*/
  struct kobject kobj;
/*队列设置*/
unsigned long nr_requests;          /*最大请求数量*/
unsigned int nr_congestion_on;
unsigned int nr_congestion_off;
unsigned int nr_batching;
unsigned short max_sectors;        /*最大的扇区数*/
unsigned short max_hw_sectors;
unsigned short max_phys_segments;        /*最大的段数*/
unsigned short max_hw_segments;
unsigned short hardsect_size;          /*硬件扇区尺寸*/
unsigned int max_segments_size;      /*最大的段尺寸*/
unsigned long seg_boundary_mask;      /*段边界掩码*/
unsigned int dma_alignment;        /*DMA 传送的内存对齐限制*/
sturct blk_queue_tag*queue_tags;
atomic_t refcnt;                    /*引用计数*/
unsigned int in_flight;
unsigned int Sg_timeout;
unsigned int sg_reserve_size;
int node;
struct list_head drain_list;
struct request*flush_rq;
unsigned char ordered;
};
```

请求队列跟踪等待的块 I/O 请求,它存储用于描述这个设备能够支持的请求的类型信息,它们的大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求

等参数,其结果是:如果请求队列被配置正确时,它不会交给该设备一个不能处理的请求。

(4) I/O 调度器

请求队列还实现一个插入接口,这个接口允许使用多个 I/O 调度器,I/O 调度器的工作是以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器累计批量的 I/O 请求,并将它们排列为递增(或递减)的块索引顺序后提交给驱动。这样做的目的是当给定顺序排列的请求时磁头可以使得磁盘顺序地从一头到另一头工作,在一个方向移动直到所有“请求”已被执行。

另外,I/O 调度器还负责合并邻近的请求,当一个新 I/O 请求被提交给调度器后,它会在队列里寻找包含邻近扇区的请求。如果找到一个,并且结果的请求不是太大,调度器将会合并这两个请求。

4.request 操作函数

request 操作涉及到的一系列相关函数如下:

(1)初始化请求队列

函数原型如下:

request_queue_t*blk_init_queue(request_fn_proc*rfn,spinlock_t*lock);

该函数的第一个参数是请求处理函数的指针,第二个参数是控制访问队列的权限的自旋锁,这个函数会进行内存分配的动作,因为他可能会失败,因此一定要检查返回值。这个函数一般在块设备驱动的模块加载函数中调用。

(2)清楚请求队列

函数原型如下:

void blk_cleanup_queue(request_queue_t*q);

这个函数完成将请求队列返回给系统的任务,一般在块设备驱动模块卸载函数中调用。

(3)分配“请求队列”

函数原型如下:

request_queue_t*blk_alloc_queue(int gfp_mask);

对于 FLASH、RAM 盘等完全随机访问的非机械设备,并不需要进行复杂的 I/O 调度,这个时候,应使用上述函数分配一个“请求队列”,并使用如下函数来绑定“请求队列”和“制造请求”函数。

void blk_queue_make_request(request_queue_t*q,make_request_fn*mfn);

这种方式分配的“请求队列”实际上不包含任何请求,所以给其加上引号。

(4)提取请求

函数原型如下:

struct request elv_nexst_request(request_queue_t*queue);

该函数用于返回下一个要处理的请求,如果没有请求则返回 NULL。它不会清除请求,而是仍然将这个请求保留在队列上,但是标识它为活动的。这个标识将阻止 I/O 调度器合并其他的请求到已开始执行的请求。因为 elv_next_request()不从队列里清除请求,因此连接调用它两次后会返回同一个请求结构体。

(5)去除请求

函数原型如下:

void blkdev_dequeue_request(struct request*req);

函数从队列中去除一个请求。如果驱动中同时从同一个队列中操作了多个请求,它必须以这样的方式将它们从队列中去除。如果需要将一个已经出列的请求归还到队列中,可以调用如下函数。

void elv_requeue_request(request_queue_t*queue,struct request*req);

另外,块设备层还提供了一套函数,这些函数可被驱动用来控制一个请求队列的操作,主要包括:

启停请求队列;

参数设置;

通告内核。

启停请求队列函数原型如下:

void blk_stop_queue(request_queue_*queue);

void blk_start_queue(request_queue_t*queue);

如果块设备到达不能处理等候命令的状态,应调用 blk_stop_queue()来告知块设备

层。之后,请求函数将不被调用,除非再次调用 blk--start--queueo 将设备恢复到可处理请求的状态。参数设置函数用于设置描述块设备可处理的请求的参数。函数原型如下[21]:

void blk_queue_max_sectors(request_queue*queue,unsigned short max);

void blk _queue_max_phys_segments(request _queue_t*queue,unsigned short max);

void blk_queue_max_segments(request -queue_t*queue,unsigned short max);

void blk_queue_max_segment_size(request_queue_set*queue,unsigned int max);

blk_queue_max_sectors()描述任一请求可包含的最大扇区数,默认值为 255;

Blk_queue_max_phys_segments()和 blk_queue_max_hw_segments()都控制一个请求中可包含的最大物理段(系统内存中不相邻的区),blk_queue_max_segrnenis()考虑了系统 I/O 内存管理单元的重映射,这两个参数默认都是 128。blk_queue_max_segment_size 告知内核请求段得最大字节数,默认值为 65,536。

5. bio 结构体

通常一个 bio 对应一个 I/O 请求,I/O 调度算法可将连续的 bio 合并成一个请求。所以,一个请求可以包含多个 bi。bio 结构体的定义代码如下[21]:

```
struct bio
{
sector_t bi_sector;     /*要传输的第一个扇区*/
struct bio*bi_next;       /*下一个 bio*/
struct block_device*bi_bdev;
unsigned long bi_lags;      /*状态,命令等*/
unsigned long bi_fw;       /*低位表示 READ/WRITE,高位表示优先级*/
unsigned short bi_vcnt;       /*bio_vce 数量*/
unsigned short bi_idx;        /*当前 bvl_vce 索引*/
/*不相邻的物理段的数目*/
unsigned short bi_phys_segments;
/*物理合并和 DMAremap 合并后不相邻的物理段的数目*/
unsigned short bi_hw_segments;
unsigned int bi_size;       /*以字节为单位所需传输的数据大小*/
/*为了明了最大的 hw 尺寸,考虑这个 bio 中第一个和最后一个虚拟的可合并的段的尺寸*/
unsigned int_bi_hw_front_size;
unsigned int bi_hw_back_size;
unsigned int bi_max_vecs;       /*能持有的最大 bvi_vecs 数*/
struct bio_vec*bi_io_vec;       /*实际的 vec 列表*/
bio_end_io_t*bi_end_io;
atomic_t bi_cnt;
void*bi_rivate;
bio_destructor_t*bi_estruetor;         /*destruetor*/
```

};

下面就对其中的核心成员进行分析:

bi_sector:表示这个 bio 要传送的第一个(512 字节)扇区;

bi_size:指以字节为单位的传送数据的大小,驱动中可以使用 bio_sectors(bi。)宏获得以扇区为单位的大小;

bi_nags:一组描述 bi。的标志,如果这是标志一个写请求,最低有效为被置位,可以使用 bio_data_dir(bio)宏来获得读写方向;

bio_phys_segments,bio_hw_segments:分别表示包含在这个 bio 中要处理的不连续的物理内存段的数目,以及考虑 DMA 重映像后的不连续的内存段的数目。

bio 的核心是一个称为 bi_io_vec 的数组由 bio_vec 结构体组成,bio_vec 结构体的定义代码如下:

```
struct_ bio_vec
{
struct page*by_page;       /*页指针*/
unsigned int bv_len        ;/*传输的字节数*/
unsigned int bv_offset;      /*偏移位置*/
};
```

不应该直接访问 bio 的 bio_vec 成员,而应该使用 bio_for_each_segment()宏来进行这项工作,可以用该宏循环遍历整个 bio 中的每个段,其定义代码如下:

```
#define_bio_for_eaeh_segment(bvl,bio,i,start_idx)          \
for(bvl=bio_iovec_idx((bio),(start_idx)),i=(start_idx);     \
i<(bio)->bi_vcnt;           \
Bvl++,i++)
#define bio_for_each_segment(bvl,bio,i)        \
bio_for_eaeh_segment(bvl,bio,i,(bio)->bi_idx
```

内核还提供了一组函数(宏)用于操作 bio;

```
int bio_data_dir(struct bio*bio);
```

这个是用于获取数据是被 READ 还是 WRITE 的函数。

```
struct page*bio_page(struc tbio*bio);
```

这个是用于获得目前的页指针函数。

```
int bio_offset(struct bio*bio);
```

获得当前页内的偏移,通常块 I/O 操作是页对齐的。

int bio_cur_sectors(struct bio*bio);

获得传输的扇区数的函数。

char*bio_data(struct bio*bio);

用来获得数据缓冲区的内核虚拟地址的函数。

char*bvec_kmap_irq(struct bio_vec*bvec,unsigned long*flags);

这个函数返回一个内核虚拟地址,该地址可用于存取被给定的 bio_vec 入口指向的数据缓冲区。其也会屏蔽中断并返回一个原子 kmap,因此,在 bvec_kunmap_rq()被调用以前,驱动不应该休眠。

void bvec_kunmap_jrq(char*buffer,unsigned long*flags);

这个函数是 bvec_kunmap_irq()函数的“反函数”,它撤销 bvec_kunmap_irq()创建的映射。

char*bio_kmap_irq(struct bio*bio,unsigned long*flags);

这个函数是对 bvec_kunmap_irq()的包装,它返回给定的 bio 的当前 bio_vec 入口的映射。

char*bio_kmap_atomic(struct bio*bio, int i, enum km_type type);

这个函数通过 kmap_atomic()获得返回给定 bio 的第 i 个缓冲区的虚拟地址。

void_bio_kunmap_atomic(char*addr, enum km_type type);

这个函数返回由 bio_kmap_atomic()获得的内核虚拟地址。

另外,对 bio 的引用计数通过如下函数完成:

void bio_get(struct bio*bio);

void bio_put(struct bio*bio);

/*引用 bio*/

/*释放对 bio 的引用*/

3.3.3 注册与注销

块设备驱动中的第一个工作通常是向内核注册,完成这个任务的函数 registe_blkdev(),其原型为:

int registe_blkdev(unsigned int major,const char*name);

major 参数是块设备要使用的主设备号, name 为设备名,它会在 proc/devices 中被显示。如果 major 为 0,内核会自动分配一个新的主设备号,register_blkdev()函数的返回值就是这个主设备号。如果 register--blkdev()返回 1 个负值,表明发生了一个错误。与 registe_blkdev()对应的注销函数 unregiste_blkdev(),其原型为:

int unregiste_blkdev(unsigned int major const char*name);

这里,传递给 registe_blkdev()的参数必须与传递给 register_blkdev()的参数匹配,否则这个函数返回-EINVAL。

在 2.6 内核中,对 register_blkdev()的调用完全是可选的,registe_blkdev()的功能已随时间正在减少,这个调用只完成:在需要时分配一个动态主设备号;在/Proc/devices 中创建一个入口。将来,registe_blkdev()可能会被去掉。但是目前的大部分驱动仍然调用它。如下是一个块设备驱动注册的模块代码:

```
xxx_major=register_blkdev(xxx_major,“xxx”);
if(xxx_major<=0)          /*注册失败*/
{
printk(KEEN_WARNING “xxx: unable to get major number\n”);
Return-EBUSY
}
```

3.3.4 加载与卸载

在块设备驱动的模块加载函数中通常需要完成如下工作:

分配、初始化请求队列,绑定请求队列和请求函数;

分配、初始化 gendisk,给定 gendisk 的 major、fops、queue 等成员赋值,最后添加 gendisk;

注册块设备驱动。

以下代码示意了使用 blk_alloc_queue()分配请求队列,并使用 blk_queue_make_request()绑定请求队列和制造请求函数的块设备驱动模块加载函数模板。

```
static int_init_xxx_init(void)
{
/*分配 gendisk*/
xxx_disks=alloc_disk(l);
if (!xxx_disks)
{
gotoout ;
}
/*块设备驱动注册*/
if(register_blkdev(XXX_MAJOR,“xxx”))
{
```

```
err=-EIO ;
goto out;
}
/*“请求队列”分配*/
xxx_queue=blk_allo_queue(GFP_KERNEL);
if(!xxx_queue)
{
goto out_queue
}
blk_queue_make_request(xxx_queue,&xxx_make_request);
/*绑定制造请求函数*/
blk_queue_hardsect_size(xxx_queue,xxx_blocksize);     /*硬件扇区尺寸设置*/
/*gendisk 初始化*/
xxx_disks->major=XXX_MAJOR;
xxx_disks->first_minor=0;
xxx_disks->fops=&xxx_op;
xxx_disks->queue=xxx_queue;
sprintf(xxx_disks->disk_name,“xxx%d”, i);
set_capcity(xxx_disks,xxx_size);       /*xxx_size 以 512bytes 为单位*/
add_disk(xxx_disks);     /*添加 gendisk*/
Return();
out_queue:unregiste_blkdev(XXX_MAJOR,“xxx”);
out:put_disk(xxx_disks);
blk_cleanup_queue(xxx_queue);
return-ENOMEM;
}
```

以下代码示意了使用 blk_int_queue()初始化请求队列,并绑定请求队列与请求处理函数的块设备驱动模块加载函数模板。

```
static int_init xxx_init(void)
{
/*块设备驱动注册*/
if (register_blkdev(XXX_MAJOR,“xxx”))
```

```
{
err=-EIO;
goto out;
}
/*请求队列初始化*/
xxx_queue=blk_init_queue(xxx_request,xxx_lock);
if(!xxx_queue)
{
goto out_queue;
}
 blk_queue_hardsect_size(xxx_queue, xxx_blocksize);   /*硬件扇区尺寸设置*/
/*gendisk 初始化*/
xxx_disks->major=XXX_MAJOR;
xxx_disks->first_minor=0;
xxx_disks->fops=&xxx_op;
xxx_disks->queue=xxx_queue;
sprintf(xxx_disks->disk_name,“xxx%d”,i);
set_capacity(xxx_disks,xxx_size*2);
add_disk(xxx_disks);       /*添加 gendisk*/
return();
out_queue: unregister_blkdev(XXX_MAJOR,“xxx”);
out:put_disk(xxx_disks);
blk_cleanup_queue(xxx_queue);
return-ENOMEM;
```

在块设备驱动的模块写在函数中通常需要与模块加载函数相反的工作:清除请求队列;删除 gendisk 和对 gendisk 的引用;删除对块设备的引用,注销块设备驱动。

下面的代码给出 f 块设备驱动模块卸载函数的模板。

```
static void_exit xxx_exit(void)
{
   if (bdev)
   {
      invalidate_bdev(xxx_bdev,l);
```

```
    blkdev_put(xxx_bedv);
    }
    del_gendisk(xxx_disks);        /*删除 gendisk*/
    put_disk(xxx_disks);
    blk_cleanup_queue(xxx_queue[i]);     /*清除请求队列*/
    unregister_blkdev(XXX_MAJOR“xxx”,);
}
```

3.3.5 打开与释放

块设备驱动的叩 oen()和 re1ease()函数并非是必须的,一个简单的块设备驱动可以不提供 open()和 release()函数。

块设备驱动的叩 open()函数和其字符设备驱动中的对等体非常类似,都以相关的 inode 和 file 结构体指针作为参数。当一个节点引用一个块设备时,inode->i_dev->bd_disk 包含一个指向关联 gendisk 结构体的指针。因此,类似于字符设备驱动,也可以将 gendisk 的 private_data 赋给 file 的 private_ata,privat_data 同样最好是指向描述该设备的设备结构体 xxx_dev 的指针,代码如下:

```
static int xxx_open(struct inode*inode,struct file*filp)
{
 struct xxx_dev*dev= inode->i_bdev->bd_disk->privat_data
 filp->private_date=dev;        /*赋值 file 的 private--date*/
    ......
 return0
 }
```

释放函数

```
tatic int xxx_release(struct inode*inode,struct file*filp)
```

第四章 Linux 内核 S3C2440 USB 驱动程序开发

4.1 USB 协议

如今,每台计算机都有 USB 端口,可与键盘、鼠标、游戏手柄、扫描仪、数码相机、打印机、驱动器以及更多设备相连 USB 具有可靠、高速、省电、价格便宜、用途广泛等特点,并且又得到了主要操作系统的支持,USB 3.0 还新增了超高速总线,这些特性意味着 USB 将继续作为不断增多的周边设备接口的主要选择[22]。

按照 USB 规范的 USB 物理接口特性,USB 接口可以分为:

- USB 主机(USB Host)端。
- USB 集线器(USBHub)。
- USB 设备(USB Device)端。

USB 集线器其实就是一类特殊的 USB 设备。在一个完整的 USB 拓扑结构上,必须有且仅有一个 USB 主机,一个或多个 USB 集线器和 USB 设备。

4.1.1 USB 规范介绍

目前使用最广泛的外部总线:USB。

USB 采用单一方向的主从设备通信模式。总线上唯一的主机负责轮询设备并发起各种传送,因此实现相对简单,成本低廉。

从拓扑上讲,USB 类似于主机同设备之间点对点连接,设备连接聚集于 HUB 集线器上。

最新的 USB 规范是 USB3.0 版本,但使用最广泛的是 USB 2.0,它定义了三种传输速率。

Low speed-----1.5Mbit/s。

Full speed-------12 Mbit/s。

High speed------480 Mbit/s。

1.USB1.1 规范

USB 1.1 规范支持低速(1.5 Mbit/s)和全速(12 Mbit/s)两种不同速率的数据传输和四种不同类型的数据传输:

(1)控制传输(CONTROL TRANSFER)。

控制传输用来控制对 USB 设备不同部分的访问,通常用于配置设备,获取设备信息,发送命令到设备,或者获取设备的状态报告。总之就是用来传送控制信息的,

每个 USB 设备都会有一个名为”端点 0”的控制端点,内核中的 USBCore 使用它在设备插入时进行设备的配置。

（2）中断传输(INTERRUPT TRANSFER)。

中断传输用来以一个固定的速率传送少量的数据,USB 键盘和 USB 鼠标使用的就是这种方式,USB 触摸屏也是使用这种方式,传输的数据包含了坐标信息。

（3）批量传输(BULK TRANSFER)。

批量传输用来传输大量的数据,确保数据不丢失,但不保证在特定时间内完成。U 盘使用的就是批量传输,用它备份数据时需要确保数据不能丢,而且也不能指望它能在一个固定的比较快的时间内复制完。

（4）等时传输(ISOCHRONOUS TRANSFER)。

等时传输同样用来传输大量的数据,但并不保证数据是否发达,以稳定的速率发送和接收实时的信息,对传输延迟非常敏感,显然是用于音频和视频一类的设备。这类设备期望能够有一个比较稳定的数据流。

2. USB 2.0 规范

除集线器外,USB 2.0 设备都可支持低速、全速或者高速模式;且高速设备在连到 USB 1.1 总线上时,也可支持全速模式。USB 2.0 集线器必须支持所有三种 USB 2.0 速度。虽然这种能够在任何速度完成通信的能力增加集线器的复杂度,但节省了总线带宽并免除了对不同速度使用不同集线器的麻烦。

USB 2.0 可反向兼容 USB 1.1。换句话说,USB 2.0 设备可使用与 1.1 设备相同的连接器和线缆,且 USB 2.0 设备连接到支持 USB 1.1 或 USB 2.0 计算机上时仍可正常工作;一些只在高速模式?工作的设备则是例外,因此他们需要 USB 2.0 支持。

3.USB OTG 规范

2001 年 12 月,USB OTG(On-Tlie-Go)规范发布,它是作为对 USB 2.0 规范的补充而出现的,新增了主机谈判协议(HNP)和对话请求协议(SRP),其目的就是为了允许两个 USB 设备可以不用 PC 主机的情况下进行通信。

根据 USB OTG 规范,一个 USB 设备的接口即可作为 USB 主机同时也可作为 USB 设备两种功能,根据与其连接的其他设备 ID 属性,USBOTG 接口的角色会自动更换为适合 USB 总线需求的接口类型。系统一旦连接后,如果采用新的 HNP 协议,USBOTG 的角色还可以变换。

4.USB 3.0 规范

USB 3.0 定义了一个新的双总线架构,带有两个并行操作的总线实体。USB3.0 将一对导线用于 USB 2.0 通信,而另外一对导线则用来支持新的 5 Gbit/s 的超高速总

线通信。超高速传输提供了 一个速率超过 USB 2.0 高速模式 10 倍以上的通信方式。而且不同于 USB 2.0,超高速模式在每个传递方向上都有一对数据线,可同时在来去两个方向上传递数据。USB 3.0 还提高了现有设备拉动的总线数据量,并且制定了为进一步实现省电和更有效传输的协议。

USB 3.0 能够反向兼容 USB 2.0。USB 3.0 主机和集线器支持所有四种速率类型。USB 2.0 电缆也匹配 USB 3.0 插槽。USB 3.0 只是对 USB 2.0 的一个补充,而不是替代。低速、全速和高速设备依旧遵照 USB 2.0 协议,但却不能利用 USB3.0 的诸如更高总线流量限制和更大数据包的特点。

5. 无线 USB

想要设计无线接口设备的 发者有几种选择。Wireless USB Promoter Group 的无线通用串行总线规范定义了已注册的无线 USB(WUSB)接口,通信速率可达 480 Mbit/s。Cypress Semiconductor 的 WirelessUSB 使实现功能类似于低速 USB 的无线设备称为可能。另一个选择是使用适配器来实现在 USB 和一些诸如 Zigbee、蓝牙或 Win 等的无线接口间的转换。

4.1.2 USB 的拓扑结构

USB 是一种主从结构的系统。主机叫做 HOST,从机叫做 DEVICE(也叫做设备)。

通常所说的主机具有一个或者多个 USB 主机控制器(HOST CONTROLLER)和根集线器(ROOT HUB)。主机控制器主要负责数据处理,而根集线器则提供一个连接主机控制器与设备之间的通路和接口。另外,还有一类非常特殊的 USB 设备-USB 集线器(USB HUB),它可以对原有的 USB 接口在数量上进行扩展,可以获得更多的 USB 接口。

一般来说, PC 上有多个 USB 主机控制器和多个 USB 接口。通常,每个主机控制器下都有一个根集线器,根集线器具有一个或者几个 USB 接口。当多个不同的 USB 设备需要较大的数据带宽时,可以将它们分别接到不同的主机控制器的根集线器上,以避免带宽不足。

USB 数据传输只能发生在主机与设备之间,主机与主机、设备与设备之间不可直接相连和传输数据。在物理上,为了区别主机和设备,使用不同的插头和插座。由主机主动发起所有的数据传输,而设备只能被动地负责应答。例如,在读取数据时,USB 先发出读命令,设备接收到该命令后,返回数据。在 USB OTG 规范中,一个设备可以在从机与主机之间转换,这样就可以实现设备与设备之间的通信,大大增加了 USB 的使用范围。但此时仍然没有脱离这种主从关系,两个设备之间有一个

作为主机,另一个作为从机。USB OTG 规范增加了一种 MINI USB 接头,它比普通的 4 线 USB 多了一条 ID 识别线,表明它是主机还是设备。

根层为 USB 主机控制器和根集线器,下面接 USB 集线器,USB 集线器将一个 USB 接口扩展为多个 USB 接口,多个 USB 接口又可以通过集线器扩展出更多的接口。但 USB 规范中对集线器的层数是有限制的。USB1.1 中规定最多为 4 层,USB 2.0 中规定最多为 6 层。一个 USB 主机控制器理论上最多可接 127 个设备,这是因为规范中规定每个 USB 设备具有 7 bit 的地址(取值范围为 0~127,而地址 0 是保留给未初始化的设备使用的)。事实上,一般是不会连接 127 个设备的。所说的一个 USB 主机控制器可以连接多个 USB 设备,并不是简单的将多个设备并联或者串联,而是要由集线器负责接口扩展,才能连接更多的设备。在 PC 个人计算机上,也有一个或者多个(视主板上的 USB 主机控制器的个数而定)集线器,它就是上面提到的根集线器,直接连在 USB 主机控制器上。

一个完整的 USB 数据传输过程如下:首先由 USB 主机控制器发出命令和数据,通过根集线器,再通过下面的集线器(如果有的话)发给 USB 设备;设备对接收到的数据进行处理后,返回一些信息或者数据,它首先到达其上一层的集线器,上层的集线器再交给更上层的集线器,一直到 USB 主机控制器为止;最终,USB 主机控制器将数据交给计算机的 CPU 处理。在标准的 PC 电脑上,USB 主机控制器是挂接在 PCI 总线上的。

4.1.3 USB 枚举过程

在应用程序与设备通信之前,主机需要了解设备,并为其分配驱动程序。枚举(Enumeration)是完成上述任务的信息交换过程。这个过程包括给设备分配地址、从设备读取描述字、分配并加载驱动程序以及选择规定了设备功耗要求和接口的配置信息。接下来,设备就准备好传输数据了。

获悉新设备时,主机将向设备所属的集线器发送请求,使集线器建立一个处于主机和设备间的通信路径。然后主机会发布通向此设备、含有标准 USB 请求的控制传输,以尝试枚举此设备。所有的 USB 设备必须支持控制传输、标准请求和端点 0。对于成功的枚举,设备必须通过返回被请求信息的形式相应请求,并执行其他所请求的动作。

4.2 Linux 内核 USB 驱动总体结构

USB 总线采用树形拓扑结构,主机端和设备端的 USB 控制器分别称为 USB 主

机端控制器(USB HOST CONTROLLER)和 USB 设备端控制器(USB DEVICE CONTROLLER, 即UDC),每一条总线上只有唯一的主机控制器,负责协调主机和设备之间的通信,而设备不能主动向主机发送任何消息。在 Linux 操作系统中,USB 驱动可以从两个角度来观察,一个角度是主机端,另外一个角度是设备端。

作为 USB device。在硬件之上是 UDC 驱动,这通常由硬件厂家提供并集成在 Linux 内核中。UDC 驱动之上,是由 Linux 内核提供的一组统一的 Gadget API。基于这些统一的 Gadget API,就是可以开发 USB Gadget 驱动,例如:file-storage(该驱动将 Linux 系统作为 U 盘向 USB host 呈现)。

作为 USB host。在硬件一 USB 主机控制器之上,是 USB 主机控制器驱动,这通常由硬件厂家提供并集成在 Linux 内核中。

在 Linux USB 驱动中,处于最底层的硬件是 USB 主机控制器,其上层运行的是 USB 主机控制器驱动,主机控制器驱动之上层为 USB 核心,再上层为 USB 设备驱动(插入主机上的 U 盘、鼠标、USB 转串口等设备驱动)。所以,在主机端的层次结构中,要实现的 USB 驱动包含两类:USB 主机控制器驱动和 USB 设备驱动,前者控制插入其中的 USB 设备,后者控制 USB 设备如何与主机通信。Linux 内核 USB 核心主要负责管理和处理协议。主机控制器驱动和设备驱动之间的 USB 核心很重要,其功能包括定义一些数据结构、宏和功能函数,向上为设备驱动提供编程接口,向下为 USB 主机控制器驱动提供编程接口;通过全局变量维护整个系统的 USB 设备信息;完成设备热插拔控制、总线数据传输控制等。

Linux 内核中 USB 设备端驱动程序分为 3 个层次:UDC 驱动程序、Gadget API 和 USB Gadget 驱动程序。UDC 驱动程序直接访问硬件,控制 USB 设备和主机之间的底层通信,向上提供与硬件相关操作的回调函数。Gadget API 是 UDC 驱动程序提供回调函数的简单集合。USB Gadget 驱动具有控制 USB 设备功能的实现。使设备表现出“网络连接”、“打印机”或“USB Mass Storage(如:U 盘)”等特性,它使用 Gadget API 控制 UDC 实现上述功能。Gadget API 把下层的 UDC 驱动程序和上层的 USB Gadget 驱动程序隔离开,使得在 Linux 内核中编写 USB 设备端驱动程序时能够把功能的实现和底层通信分离。

4.3 USB 主机控制器的整体结构

4.3.1 HC 主机控制器

USB 主机控制器分为以下几种:

（1）UHCI (即 Universal Host Controller Interface,通用型主机控制器接 口):该标准是 Intel 提出来的,基于 Intel 的 PC 机很可能就会有这种控制器。

（2） OHCI(即 Open Host Controller Interface,开放型主机控制器接口):该标准是 Compaq 和 Microsoft 等公司提出来的。兼容 OHCI 的控制器硬件智能程度比 UHCI 高,所以 OHCI 的 HCD(Host Controller Drivers,主机控制器驱动程序)比基于 UHCI 的 HCD 更简单。

（3）EHCI(即 Enhanced Host Controller Interface,增强型主机控制器接口):该主机控制器支持高速的 USB 2.0 设备。为支持低速的 USB 设备,该控制器通常同时包含 UHCI 和 OHCI 控制器。

（4）USB OTG 控制器:这类控制器在嵌入式微控制器领域越来越受欢迎。由于采用了 OTG 控制器,每个通信终端就能充当 DRD(Dual-Role Device,双重角色设备)。用 HNP(Host Negotiation Protocol,主机沟通协议)初始化设备连接后,这样的设备可以根据功能需要在主机模式和设备模式之间任意切换。

除上面主流的 USB 主机控制器以外,Linux 还支持好几种主机控制器,如用于 ISP116X 芯片的 HCD。

主机控制器内嵌了一个叫根集线器的硬件。根集线器是逻辑集线器,多个 USB 端口共用它,这些端口可以和内部或外部的集线器相连,扩展更多的 USB 端口,这样级联下来,USB 结构形成树状[23]。

4.3.2 HCD 主机控制器驱动

USB 主机控制器驱动中,最重要的接口是 HCD 主机控制器驱动与 USB Core 之间的接口。在 Linux 内核中,用 usb_hcd 结构表示与 USB Core 的接口,它用来描述 USB 主机控制器的所有信息,包含 USB 主机控制器的基本信息、厂商、硬件资源、状态描述和用于操作主机控制器的 hc_driver 等[37]。结构 usb_hcd 代码描述如下:

```
static struct usb_hcd {
/*.........*/
/*硬件信息和状态*/
const struct hc_driver *driver;       /*  硬件特定的钩子函数  */
unsigned uses_new_polling : 1 ;       /*  是否允许轮询根  hub  状态  */
unsigned poll rh : 1 ;                /*  轮询根  hub  状态  */
unsigned poll_pending ; 1 ;           /*  状态改变了吗? *1
unsigned rh_register : 1;             /*  根  hub  是否注册  */
int irq;                              /*控制器的中断号*/
void_iomem *regs;                     /*  设备的  io  内存  */
u64 rsrc_start;                       /*分配的内存起始地址*/
```

```
u64 rsrc_len;          /*分配的内存长度*丨
/* .....*/
};
```

4.4 Linux USB 设备驱动程序

4.4.1 USB 设备驱动的框架

Linux 内核源码树中的代码 linux-2.6.32.2/drivers/usb/usb-skeleton.c 这个文件为我们提供了一个最基本、最简单的 USB 设备驱动示例。编写 USB 设备驱动可以直接修改该文件来驱动新的USB设备。下面以usb_skeleton.c文件为例分析usb-skel设备的驱动框架[24、25]。

1.私有数据结构

usb_skel 设备使用自定义的结构 usb_skel,该结构根据具体的设备量身定制,用于记录设备驱动用到的所有描述符,该结构定义如下代码所示:

```
struct usb_skel {
struct usb_device *udev;              /* USB 设备描述符 */
 struct usb_interface *interface;      /* USB 接 口 描述符 */
struct semaphore limit_sem;           /* 限制进程写的数量 */
unsigned char *bulk_in_buffer;        /* 接收数据的缓冲区 */
size_t bulk_in_size;          /*接收数据缓冲区大小*/
_u8 bulk_in_endpointAddr;      /*"IN 入端点的地址 */
—u8 bulk_out_endpointAddr;       /* OUT 出端点的地址 */
/*.....*/
};
```

对于 usb_skel 设备驱动来说,其 usb_skel 的结构在 URB (USB Request Block 结构的 context 指针里保留。借助于 URB, usb_skel 设备的所有文件操作函数实际上都可以访问到 usb_skel 结构。然而,limit_sem 成员是一个互斥信号量,若干 usb_skel 设备存在于 Linux 系统中的时候,是需要控制设备间的数据同步。

2.设备驱动的初始化与注销

usb-skel 设备驱动使用 module_Lnit()和 module—exit()宏对驱动程序模块的始化函数和退出函数名称进行记录。usb_skd_init()函数是 usb-skel 设备驱动的初始化函数,其代码定义如下所示:

```
static int_init usb_skel_init(void) {
```

```
    int result;
    result = usb_register(&skel_driver);       /* 注册 USB 设备驱动 */
    if (result)
        err ( “usb_register failed. Error number %d”, result);
    return result;
  };
```

usb_skel_init()函数调用内核提供的 usb_register()函数,目的是注 usb_driver 类型的结构变量 skel_driver,该变量代码定义如下所示:

```
static struct usb_driver skel_driver = {
.name = "skeleton",     /* USB 设备名称 */
.probe = skel_probe,       /* USB 设备探测函数 */
.disconnect = skel_disconnect,     /* USB 设备断 函数 */
.id_table = skel_table,     /* USB 设备 ID 映射表 */
};
```

skel_driver 结构变量中,定义了 usb-skd 设备的名称、探测函数、断开函数和 ID 映射表。其中 usb-skel 设备的列表数组为 skel_table[],它的 ID 映射表代码定义如下所示:

```
static struct usb_device_id_skel_table [] = {
{USB_DEVICE(USB_SKEL_VENDOR_ID,USB_SKEL_PRODUCT_ID) },
 {}     /* Terminating entry */
};
```

skel_table 中只有一项,定义了一个默认的 usb_skel 设备的 ID。

其中 USB_SKEL_VENDOR_ID 是 USB 设备的厂商 ID,USB SKEL PRODUCT ID 是 USB 设备的产品 ID。

usb_ske 丨设备驱动的退出函数操作比较简单,调用 usb__deregister()函数退出 usb-skel 设备驱动,函数代码定义如下所示:

```
static void _exit usb_skel_exit(void) {
usb_deregister (&skel_driver);        /* 退出 USB 设备驱动 */
};
```

3.探测 USB 设备

从 usb_driver 结构类型的变量 skel_driver 中,我们可以看出 skel_probe()函数是 usb_skel 设备的探测函数。探测 USB 设备主要是探测 USB 设备的信息与类型、注

册 USB 设备操作函数、分配 USB 设备所用的 URB 资源等。usb_class_driver 结构变量 skel_class 记录了 usb_skel 设备信息,代码定义如下:

```
static struct usb_class_driver skel_class = {
.name = "skel%d",     /*  设备名称  */
.fops = &skeI_fops,     /*  设备操作函数  */
.minor_base = USB_SKEL _ MINOR_BASE,
};
```

成员变量 name 使用通配符"%d"表示一个整型变量,在一个 usb_skd 设备接到 USB 总线以后,它会按照次设备编号自动为其配置设备名称。结构变量 fops 是设备操作函数,代码定义如下所示:

```
static struct file_operations skel_fops = {
.owner = THIS_MODULE,
.read = skel_read,     /*  设备读函数  */
.write = skel_write,     /*  设备写函数  */
.open =skel_open,     /*  设备打开函数  */
.release = skel_release,     /*  设备释放函数  *丨
};
```

usb_skel 设备中的 skel_fops 为设备操作函数。如果在 usb_skel 设备上发生相关事件,usbfs(usb 文件系统)就会调用它对应的函数来处理。

4.断开 USB 设备

在断开设备的时候会调用 skel_disconnect()函数,其代码定义如下所示:

```
static void skel_disconnect( struct usb_interface * interface) {
/*.........*/
/*  阻止  skel_open()与  skel_disconnect()的竞争  */
lock_kernel();
dev = usb_get_intfdata(interface);     /*  获得  USB  设备接  口数据  */
/*设置 USB 设备接口数据为 NULL*/
usb_set_intfdata(interface, NULL);
/*注销 USB 设备,释放此设备号*/
usb_deregister_dev(interface, &skel_class);
unlock_kemel();    /*  操作完毕解锁  */
kref_put( &dev->kref, skel_delete);     /*  减小引用计数  */
```

};

4.4.2 USB 设备驱动的实现与调试

1. USB 设备驱动的实现

设备驱动实现的功能包含为初始化设备、提供设备服务、负责内核与设备之间的数据互换、以及检测和处理工作中设备出现的错误等。实现这些功能,每个设备驱动都需要注册和注销,只有这样,内核才知道这个驱动存在或者驱动已卸载[24]以下是两个最基本的模块:

module_init(usb_skel_init);

module_exit(usc_skel_exit);

2. USB 设备驱动的调试

设备驱动的调试主要有两种方式,一种是打印调试,即通过调用 printk()函数打印相关信息来观察函数运行结果printk()函数是内核空间的一个输出闲数,等同于用户空间的 printf()函数。这种调试方式是驱动 发者使用最为常见的。另外一种是 kgdb 远程调试,自 linux-2.6.26 内核开始,内核调试器 kgdb 被内嵌进内核。使用 kgdb 需要两台计算机,一台运行带有 kgdb 内核作为目标机,第二台作为主机通过串口或网口的方式使用 gdb 对目标机进行调试。目标机上的内核在启动时会等待远程 gdb 的连接,远程主机上的gdb 负责读取内核符号表和源代码并建立连接。通过 kgdb,gdb 的所有功能都能使用,包括修改变量值、设置端点、单步执行等。但是在实际操作时,大部分还是使用 printk 方法来调试。调试完成后可以用三种方法来加载驱动模块:

执行 insmod 动态加载驱动模块。

在编译内核时,执行 modprobe 以模块方式来加载模块。

在编译内核时,执行 cat /proc/kmesg 把模块编译进内核。

第五章 基于嵌入式 Linux 的接口通信

接口总线为一群互相连接的设备提供一种有效的通信方式，它是自动测试系统发展的标志, 可以说接口总线技术发展的历史就是自动测试系统发展史,接口总线技术发展的水平标志着自动测试发展的水平。本文实现了包括USB 总线、GPIB 总线、网络等几种通用标准接口之间的相互通信。

5.1 Linux系统移植

Linux 操作系统是可以运行在不同类型计算机上的一种操作系统的“内核”，它提供命令行或者程序与计算机硬件之间接口的软件核心部分。嵌入式 Linux 系统从软件角度分为 4 个层次，分别为引导加载程序（Boot Loader）、内核、文件系统和用户程序。

Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，可以初始化硬件设备、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，以便为最终调用操作系统内核准备好正确的环境。

本文 Linux 内核采用 Linux2.6.16 版的 kernel。因为本方案需要访问 U 盘，所以内核中必须加上支持 U 盘的模块，包括 SCSI support、SCSI disksupport、USB mass storage support、VFAT（windows95）support、MSDOSpartition tables 等。USB 器件端口与 USB 主机通信，可以有三种方式：一些功能最完备结构也最复杂的设备，采用用户定制的内核模块实现在标准 USB 总线上运行复杂的高级协议，由 USB 主机上相应的用户驱动程序和应用程序来完成连接；另一些基于 Linux 系统的 USB 设备则利用 USB 总线来实现该设备与主机之间的简单点对点串行连接，主机上的应用程序实际上是利用了主操作系统所提供的 U S B 编程接口，但实现的是串行通信协议；最后一种是设备将主计算机作为网关，将 USB 设备连接到办公局域网或互联网上，从而 USB 设备构成了一个模拟以太网接口。本文采用最后一种方式，配置了内核中的 USB RNDIS gadget 模块，该模块利用 USB 接口作为物理媒介，模拟出一个虚构的以太网设备。

文件系统是用户模式进程与内核模式进程交互的纽带，制作具有特定的功能的文件系统是移植嵌入式系统不可缺少的部分。本方案利用 Busybox 软件工具包构造 EXT2 文件系统，此软件包集成了 Linux 常用命令，可根据需求裁减，极大的方便了嵌入式系统的开发。

5.2 应用程序的具体实现

数据在多个接口之间的转换主要是在用户态的应用程序中实现。此应用程序主要完成的功能是两个USB 主口（一个接U S B 仪器，一个接U 盘）、一个U S B 从口、GPIB 口和网口之间的数据交换。程序自动检测接口是否处于连接状态：如果是，便监听是否有数据需要传送，并将监听到的数据传给其他处于连接状态的接口。

各个子进程的功能如下：

子进程1 ：USB 从口的实现主要靠内核中的USB RNDIS gadget 模块的支持，当USB 从口连接至PC 时，在PC 上模拟以太网接口，用TCP/IP 网络协议传输数据；

子进程2 ：USB 主口1 专用于USB设备的U S B 端口；

子进程3：USB 主口2 专用于U 盘，将U 盘实际连接，当有数据传给U 盘时，便在Linux 操作系统上挂载U 盘，并以二进制方式将数据存储到U 盘的Linux.txt 文件中，15s 内没有数据传输就会卸载U 盘；

子进程4：网口是以TCP/IP网络协议传输数据。子进程1和子进程4 分别实现了网络服务器功能；

子进程5：在Linux 操作系统中为GPIB 口准备一个中断例程，当GPIB 口有动作时，会首先产生一个中断，应用程序立即跳转中断处理执行程序相关操作。

由于接口总线协议的解析都在Linux 驱动中完成，所以子进程间的数据传输直接代表了接口总线的通信。子进程创建了自己专用的管道FIFO，以完成进程间的数据交换。

每个子进程都有一个主线程，完成端口的连接状态检测，主线程的流程图如图3 所示。端口输出数据的处理主要由线程readfifo 完成， 它读取该进程的专用管道FIFO 的数据，如果从FIFO 中读到了数据，并且该端口连接标志位flag 为1（表示该端口处于连接状态），便将此数据填入该端口的输出缓冲区，否则将读到的数据舍去。端口输入数据的处理由另一个线程完成，如果该端口处于连接状态，主线程就会创建该子线程实时读取该端口的输入数据，并将数据通过专用管道传给其他进程。

在整个应用程序中，各个端口的数据转发利用了Linux 进程间的数据通信技术。每个端口都设置了一定大小的数据缓冲区，使数据能够连续发送而不受外围控制器速度的影响，接收数据可靠，并尽可能减少了错误接收和错误判断的可能性。

第六章 总结与展望

6.1 总结

随着人们对消费电子设备功能的便利化和多样化要求,为了适应这种要求,人们把嵌入式设备与 linux 系统结合在一起。为了适应这种越来越丰富的接口驱动要求而研究本课题,本文主要研究了嵌入式的发展，S3C2440 的内部结构及相关接口，设备驱动的分类，字符设备和块设备驱动的开发方法、步骤和技巧。最后结合 mini2440 开发板和 Linux 系统平台实现了 USB 接口和 GPIB 接口之间的通信。

6.2 展望

今后, 嵌入式设备的发展趋向网络化, 并且能够支持多用户。

本文只论述了字符设备和块设备的驱动, 没有进行网络设备驱动的研究, 还需在网络设各驱动方面做一些研究。

致谢

在电子科技大学的学习期间得到了自动化专业老师的细心照顾和帮助使我不仅理论基础上了很好的提高,而且具有了很好的动手和实践能力,在此对所有关心和帮助过我的人表示衷心感谢。

首先我要感谢我的导师袁渊老师，感谢他在我做毕业设计的这段时间给予我悉心的指导和耐心的帮助。

其次感谢还要所有在我论文写作过程中给予指导和帮助的同学和朋友。也感谢在百忙之中对我论文进行评审的各位教授。

参考文献

[1]曲振华.基于 Linux 平台的嵌入式便携播放器的研究与设计[D].太原理工大学,2009

[2]宁波.linux 嵌入式系统在工控领域的研究与实现[D].电子科技大学,2007

徐晨辉.嵌入式 Linux 内核裁剪及移植的研究与实现〔D].东华大学,2009

[3]戈志华.基于 ARM 的 Linux 平台上 USB 驱动的实现[D].南昌:南昌大学,2008.

[4]李继伟.基于 ARM 的嵌入式系统研究及 USB 驱动程序设计[D].两安:西安电子科技大学,2005.

[5] JanAxelson.USB 开发大全(第四版)[M].李鸿鹏等译.北京:人民邮电出社,2011.

[6][美]Lewin A.R.W Edward.嵌入式工程师必知必会[M].张乐锋等译.北京:人民邮电出版社,2011.

[7]杜春雷.ARM 体系结构与编程[M].北京:清华大学,2004.

[8]张瑜,王益涵.ARM 嵌入式程序设计[M].北京:北京航空航天大学出版社.2009.

[9] Samsung Electronics Co, Ltd. S3C2440A 32-BIT MICROCONTROLLER USER'SMANUAL Revision 1 [EB/OL], 2004.

[10]Jan Axelson. USB Embedded Hosts: The Developer's Guide [M]. LakeviewResearch,2011.

[11]刘荣.圈圈教你玩 USB [M].北京:北京航空航天大学出版社,2009.

[12]华清远见嵌入式培训中心任桥伟.Linux 内核修炼之道[M].北京:人民邮电山版社,2010.

[13]潘伟.基于 ARM 的无线通信平台开发及 USB 驱动程序设计[D].武汉:武汉理工大学,2007.

[14]耿恒山.基于 ARM9 的车载定位终端研究及 USB 驱动程序设计[D].天津:河北工业人学,2006.

[15]孙戈,卢建军,高理等.基于 S3C2440 的嵌入式 Linux 开发实例[M].西安:西安电子科技大学出版社,2010.

[16]周荷琴,吴秀清.微型计算机原理与接口技术[M].合肥:中国科学技术大学出版社,2009.

[17]华清远见嵌入式培训中心肖林浦,肖季东,任桥伟.Linux 那些事儿之我是 USB[M].北京:电子工业出版社,2010.

[18]华清远见嵌入式培训中心宋宝华.Linux 设备驱动 发详解(第 2 版)[M].北京:人民邮电出版社,2010.

[19]肖刚,李纪扣,畅卫东等.嵌入式 Linux 下 USB 驱动的实现[J].微计算机信息,2007,23(9).

[20]杨建华,黄宇东.基于嵌入式 Linux 的 USB 驱动设计[J].福建电脑,2009,21(4).

[21]张玉民,陈定方.Linux 下 USB 驱动程序的设计与实现[J].湖北工业大学学报,2007,22(3).

[22]弓雷.ARM 嵌入式 Linux 系统 发详解[M].北京:清华人学出版社,2010.

[23]李传伟,胡金春.嵌入式 Linux 下 USB Gadget 驱动框架研究[J].航天控制,2006,24(6).

[24]Peter Jay Salzman, Michael Burian, Ori Pomerantz. The Linux Kernel Module Programming Guide [M], CreateSpace, 2009.

[25]Sreekrishnan Venkateswaran[印].精通 Linux 设备驱动程序开发[M].宋宝华,何昭然,史海滨等译.北京:人民邮电出版社,2010.

外文翻译

Linux Driver Verification

(Position Paper)

Abstract. Linux driver verification is a large application area for software verification methods, in particular, for functional, safety, and security verification. Linux driver software is industrial production code -IT infrastructures rely on its stability, and thus, there are strong requirements for correctness and reliability. This implies that if a verification engineer has identified a bug in a driver, the engineer can expect quick response from the development community in terms of bug confirmation and correction. Linux driver software is complex, low-level systems code, and its characteristics make it necessary to bring to bear techniques from program analysis, SMT solvers, model checking, and other areas of software verification. These areas have recently made a significant progress in terms of precision and performance, and the complex task of verifying Linux driver software can be successful if the conceptual state-of-the-art becomes available in tool implementations.

Overview

The Linux kernel is currently one of the most important software systems in our society. Linux is used as kernel for several popular desktop operating systems(e.g., Ubuntu, Fedora, Debian, Gentoo), and thus, the seamless workflow of many users depends on this software. Perhaps even more importantly, the server operating systems that currently dominate the market are based on Linux. Almost all (90% in 2010) supercomputers run a Linux-based operating system. Increasingly many embedded devices such as smart phones run Linux as kernel (e.g., Android, Maemo, WebOS). This explains an increasing need for automatic verification of Linux components.

Microsoft had identified the device drivers as the most important source of failures in their operating systems. Consequently, the company has significantly increased the reliability of Windows by integrating the Static Driver Verifier (Sdv) into

the production cycle. The foundations were developed in the Slam research project [1]. The Sdv kit is now included by default in the Windows Driver Kit (Wdk).

For Linux, an industry-funded verification project of the size of Sdv does not exist. But the development community is increasingly looking for automatic techniques for verifying crucial properties, and the verification community is using Linux drivers as application domain for new analysis techniques. During the last years, three verification environments were build in order to define verification tasks from Linux drivers: the Linux Driver Verification project 1 [23], the Avinux project [27], and the DDVerify project 2 [32].

The Linux code base is a popular source for verification tasks .Linux drivers provide a unique combination of specific characteristics that attract researchers and practitioners to challenge their tools. The most important benefits of using the Linux code as source for verification tasks are the following:

• the software is important – many people are interested in verification results;

• every bug in a driver is potentially critical because the driver runs with kernel privileges and in the kernel's address space;

• the code volume is enormously large (10MLOC) and continuously increases;

• the verification tasks are difficult enough to be challenging, but not too complex to be hopeless; and

• most Linux drivers are licensed as open source and therefore easy to use in verification and research projects.

Although many new advancements in the area of software verification have been made, it requires a special effort to transfer them to practice and make them applicable to complex industrial code such as Linux device drivers. The recent competition on software verification (SV-COMP'12) 3 [3] showed that even modern state-of-the-art tool implementations have problems analyzing the problems in the category on device drivers.

Linux 驱动程序验证

摘要

Linux 驱动程序验证是一款大型的进行软件验证方法的程序,尤其在功能及安全验证方面。Linux 驱动程序软件是工业生产代码——基础设施依赖它的稳定性,因此,对正确性和可靠性有极大要求。这意味着如果一个验证工程师发现了一个错误的驱动程序,开发社区的工程师可以做出快速反应并对错误进行确认和修正。Linux 驱动程序软件是复杂的,低级系统代码和其特点为解决 SMT,模型检查和其他领域的软件验证提供了技术支撑。这些地区最近在精度和性能方面取得了重大进展,所以如果利用最先进的工具验证 Linux 驱动程序软件的复杂任务将会取得成功。

概述

Linux 内核是目前我们社会最重要的一个软件系统之一。Linux 是流行的桌面操作系统的核心。如(Ubuntu Fedora,Debian,Gentoo)。因此,许多用户的无缝工作流取决于这个软件。也许更重要的是,目前主导市场的服务器操作系统本身就基于 Linux。几乎所有(2010 年是 90%)超级计算机运行一个基于 linux 的操作系统。越来越多的嵌入式设备如智能手机运行 Linux 内核(如 Android,Maemo,WebOS)。这就解释了对于自动验证 Linux 组件的持续需求。

微软已经确定了设备驱动程序的最重要来源失败在他们的操作系统。因此,该公司已通过集成的静态驱动匹配器(关闭阀)到生产周期的方法显著增加了 Windows 的可靠性，并以大力开发研究项目为基础[1]。Sdvkit 现在默认包含在 Windows 驱动程序内。

Linux 的工业资助验证项目的大小 ofSdvdoes 不存在。但是开发团体正在大力寻找自动验证技术至关重要的属性,他们使用 Linux 驱动程序作为新的分析技术的应用领域。在最后几年,为了通过 Linux 驱动程序定义验证任务，开发团体构建了三大验证计算机运行环境:Linux 驱动程序验证项目,Avinux 项目[27],和 DDVerify 项目。Linux 代码库对于验证任务来说是有用的来源[17 日,22 日,24 日,25 日]。Linux 驱动程序提供了一个独一无二的具体特征的结合，这种特征吸引了研究者和实践者们来挑战他们的工具。使用 Linux 的代码作为验证任务的来源有以下重要益处:

•软件很重要——许多人对验证结果感兴趣;

•驱动程序中的每一个错误都是潜在的威胁,因为驱动程序以内核的特权运行在内核空间;

•体积是非常大的代码(10 MLOC),不断增加;

•尽管挑战验证任务有困难,但不要因为太复杂而绝望;

•大多数 Linux 驱动程序是作为开放源码许可,因此易于使用的验证和研究项目。

尽管在软件验证领域取得许多新的进展,它需要一种特殊的努力转移到实践中,使其适用于复杂的工业,如 Linux 设备驱动程序代码。最近竞争软件验证(SV-COMP 12)显示,即使是现代最先进的工具也有设备驱动程序方面的问题。