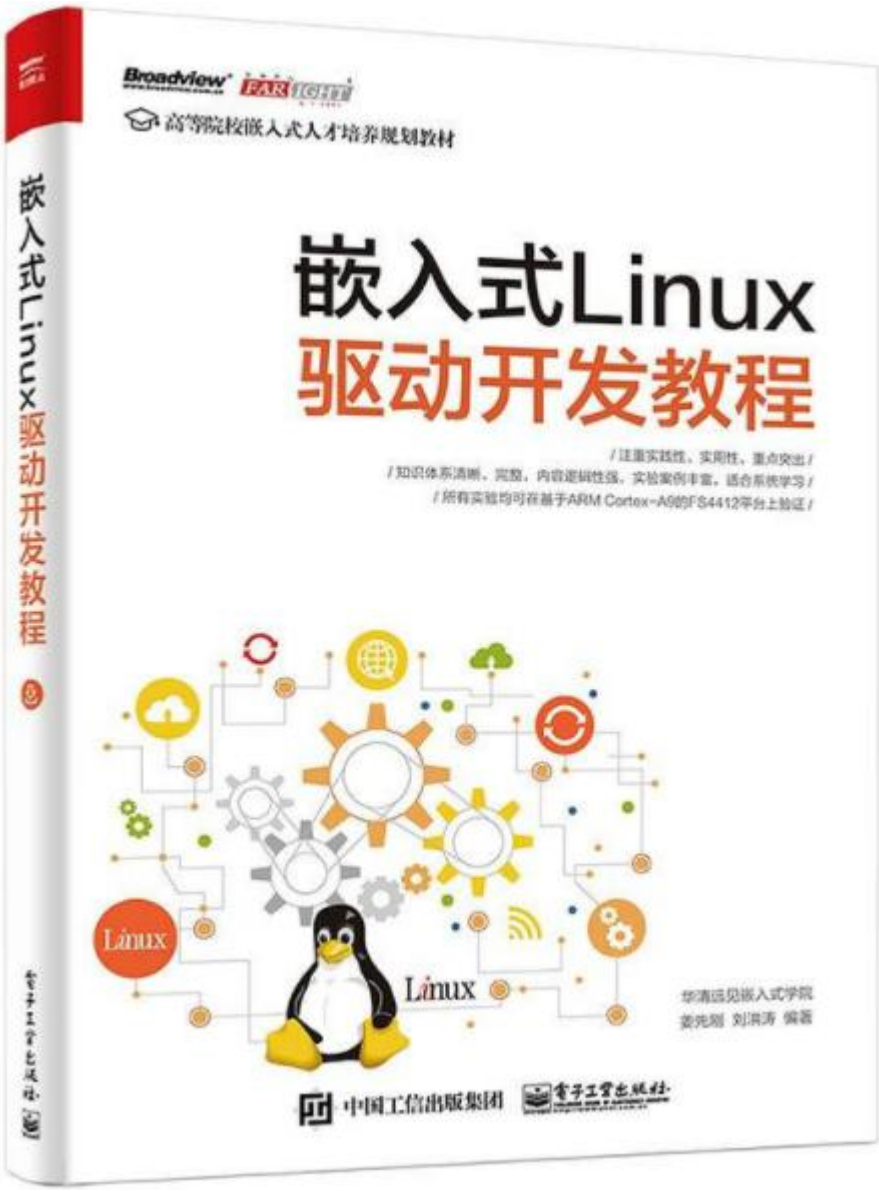

华清远见原创图书《嵌入式 Linux 驱动开发教程》

以下为 1-3 章电子版分享

更多图书简介及电子版下载：http://www.hqyj.com/FarsightBooks/

目录

摘要

本书结合大量实例，在基于 ARM Cortex-A9 四核处理器 Exynos4412 的硬件教学平台和 PC 机上，全面详细讲解了 Linux 设备驱动开发。主要内容包含开发环境的搭建，内核模块，字符设备驱动框架，高级 I/O，中断和时间管理，互斥和同步，内存和 DMA，Linux 设备模型，外设的驱动实例，总线类设备驱动，块设备驱动，网络设备驱动和内核调试技巧。对每一个知识点都有一个对应的典型实例，大多数实例既可以在上面说到的嵌入式平台上运行，也可以在 PC 机上运行。另外，本书也引入了新内核的一些新特性，比如高分辨率定时器，针对嵌入式平台的 dmaengine 和设备树。在需要重点关注的地方还加入了大量的内核源码分析，使读者能够快速并深刻理解 Linux 设备驱动的开发。

本书可作为大学院校电子、通信、计算机、自动化等专业的嵌入式 Linux 设备驱动开发课程的教材，也可供嵌入式 Linux 驱动开发人员参考。

前言

随着嵌入式及物联网技术的快速发展，ARM 处理器已经广泛地应用到了工业控制、智能仪表、汽车电子、医疗电子、军工电子、网络设备、消费类电子、智能终端等领域。而较新的 ARM Cortex-A9 架构的四核处理器更是由于其优越的性能被广泛应用到了中高端的电子产品市场。比如基于 ARM Cortex-A9 的三星 Exynos4412 处理器就被应用在了三星 GALAXY Note II 智能手机上。

另一方面，Linux 内核由于其高度的稳定性和可裁剪性等特点，被广泛地应用到了嵌入式系统，其中 Android 系统就是一个典型的例子。这样，ARM 处理器就和 Linux 操作系统紧密地联系在了一起。所以，基于 ARM 和 Linux 的嵌入式系统就得到了快速的发展。

嵌入式系统是一个定制的系统，所以硬件上千变万化，形形色色的硬件都必须要有对应的驱动才能使其正常工作，为这些硬件设备编写驱动就是不可避免的了。虽然有很多内核开发人员已经为很多常见的硬件开发了驱动，但是驱动的升级一般都跟不上新硬件的升级。笔者就多次遇到过内核的驱动和同一系列的升级版本芯片不匹配的情况，这时就要改写驱动程序。所以内核层次的底层开发几乎都要和驱动打交道。另外，了解驱动（或者说内核）的一些底层工作原理，也有助于我们写出更稳定、更高效的应用层代码。为了能够实现这一目标，并促进嵌入式技术的推广，华清远见研发中心自主研发了一套基于 Exynos4412 处理器的开发板 FS4412，并组织编写了本书。本书注重实践、实用，没有长篇大论来反复强调一些旁枝末节的东西，但是对于会影响理解的部分又非常详细地分析了内核源码，并给出了大量的图示。书中的各个实例虽然为了突出相关的知识重点而简化了某些问题的讨论，不能称得上工程上严格意义的好的驱动，但是对应的设备驱动开发所必须的各方面确实也是具备了的。例子按照工程上驱动开发的增量式方式来进行，即先有主体再逐渐完善，循序渐进。读者跟着例子能够迅速掌握对应驱动的开发精要，对整个驱动的实现也就有了一个清晰的思路。

本书共 14 章，循序渐进地讲解了嵌入式 Linux 设备驱动开发所涉及到的理论基础和大量 API 说明，并配有大量驱动实例。全书主要分成了五个部分，第一部分是 Linux 设备驱动开发的概述，包含第 1 章；第二部分是模块及字符设备驱动的理论，包含第 2 章到第 8 章；第三部分是字符设备驱动实例，包含第 9 章和第 10 章；第四部分是 Linux 块设备驱动和网络设备驱动，包含第 11 章和第 12 章；最后一部分是 Linux 内核的调试和开发环境的搭建，包含第 13 章和第 14 章。各章节的主要内容如下。

第 1 章概述了需要了解 Linux 驱动程序的人群，Linux 驱动开发的特点和本书剩下各章节的核心内容。

第 2 章对 Linux 内核的模块进行了介绍，现在的驱动几乎都以 Linux 内核模块的形式来实现，所以这是后续的基础。

第 3 章讲解 Linux 字符设备驱动的主体框架，并以一个假想的串口来实现驱动。这是 Linux 设备驱动入门的关键，所以分析了大量的内核源码。当然这个驱动是不完善的，需要在后面的各章节逐步添加功能。

第 4 章在上一章的基础上探讨了字符设备的高级 I/O 操作，包括 ioctl、阻塞、I/O 多路复

用、异步通知、mmap、定位等。另外，还特别介绍了 proc 相关的接口。

第 5 章讲解中断和时间管理，为便于理解，特别加入了中断进入的内核源码分析。时间管理则包含了延时和定时两部分，其中定时还讨论了新内核中的高分辨率定时器。

第 6 章讲解了互斥和同步，为了让读者明白互斥对驱动开发的重要性，特别从 ARM 汇编的层次来讨论了竞态。除了对传统的互斥（自旋锁、信号量等）进行讨论外，还特别说明了 RCU 机制和使用的范例。

第 7 章讲解了内核中内存的各种分配方式，特别还谈到了 per-CPU 变量的使用。最后，对 DMA 的讨论则专注于新内核引入的 dmaengine 子系统，并用一个实例来进行了具体的展现。

第 8 章讲解了 Linux 设备模型，这部分内容比较抽象。为了能帮助读者理解这部分内容，专门实现了设备、总线、驱动的三个最简单实例，从而能使读者完全掌握三者之间的关系。这一章的后半部分有大量实用技术的展现，包括电源管理、驱动的自动加载、设备节点的自动创建等。最后还讲解了较新的内核引入的 ARM 体系结构的设备树。

第 9 章在前面的理论基础上实现了大量外设的驱动。这些驱动并不都是通过字符设备框架来实现，目的就是想告诉读者，如果我们能够简化驱动的编写就尽量简化驱动的编写，多使用内核中已经实现的机制。

第 10 章讲解了总线类设备驱动的开发，对流行的 I2C 总线、SPI 总线、USB 总线和 PCI 总线都进行了讨论。这些总线都有个共同的特性，就是都有主机控制器和连接在总线上的设备，我们只讨论了在主机控制器驱动之上的设备驱动，不讨论主机控制器驱动及设备自身的固件或驱动，因为设备驱动是最常开发的驱动。

第 11 章讲解块设备驱动，为了便于读者对这部分知识的理解，特别介绍了磁盘的内部结构，然后用内存虚拟了一个磁盘，用两种方式来实现了该虚拟磁盘的块设备驱动。

第 12 章讲解了网络设备驱动，用一个虚拟的环回以太网卡的驱动来展现了网络设备驱动的主体框架，后面还分析了 DM9000 网卡驱动的主体框架部分，并和前面的虚拟网卡驱动进行了对比。

第 13 章介绍了内核的一些调试技巧。内核的调试相对来说比较麻烦一些，但只要能熟练使用这些调试技术，还是能较快找出问题所在。

第 14 章是嵌入式 Linux 设备驱动开发环境的搭建，包含了主机系统的准备和各个软件的安装。特别的是，用 vim 搭建了一个适合于驱动开发的类似于 IDE 的编辑环境，能够大大提高代码的编写效率。

本书由华清远见成都中心的姜先刚编写，北京中心的刘洪涛承担全书的统稿及审校工作。本书的内容是华清远见嵌入式培训中心所有老师心血的结晶，是多年教学成果的积累。他们认真阅读了书稿，提出了大量的建议，并纠正了书稿中的很多错误，在此特表示感谢。

由于作者水平有限，书中不妥之处在所难免，恳请读者批评指正。对于本书的批评和建议，可以发表到 www.farsight.com.cn 技术论坛。

编 者

2016 年 10 月

第 1 章 概述

本章目标

本章首先概要性地介绍了需要了解 Linux 设备驱动程序的人群，然后在不涉及过多具体知识点的情况下讨论了 Linux 驱动程序的部分显著特点，最后概要说明了本书各个章节的核心内容。

对于 Linux 这样成功的、优秀的开源项目，随着他应用的日益广泛，已受到越来越多软件开发者的追捧，很多人开始在学习他，研究他。但是当从官网上下载了源码并解压后，我们往往会迷失在浩瀚的代码汪洋中，这巨大的代码量，会令大多数人望而却步。那么，我们是不是就没有办法征服他了呢？记得很有经验的前辈曾提到过如下两个突破口：驱动和网络。确实，如果对 Linux 的各部分内核源码做一个统计的话，会发现这两部分代码所占的比例是最高的。本书是一本驱动开发的入门级教程，希望能帮助各位 Linux 内核初学者从中找到突破口，从而打开 Linux 内核的大门，进而能够掌握 Linux 驱动程序的开发。

总体来说，专门从事 Linux 驱动开发的工程师并不是特别多，但是这并不是说我们就完全没有和他打交道。笔者认为，除了专职的 Linux 驱动开发工程师而外，学习或了解 Linux 驱动开发对下面几类开发人员还是有益的。

（1）Linux 系统移植工程师。虽然现在 ARM 体系结构的 Linux 内核也引入了设备树，使得内核移植工程师几乎不需要再改写 BSP 文件，只是修改设备树和对内核进行选配就可以了，但是移植的过程并不总是会特别地顺利。比如笔者曾经在移植 LCD 驱动时就发现，官网源码中提供的设备树节点在驱动中不能被正确识别，查看驱动源码发现，要修改节点的编写形式。也就是说，要能写好一个设备树节点，除了参考内核的文档外，有时还要参考驱动源码，所以能看懂相应的驱动源码很重要。另外，所选配的驱动可能不能使硬件完美工作起来，还是以前面那个 LCD 驱动为例，笔者发现驱动中假设了 U-Boot 已经将 LCD 的时钟初始化好了，但事实情况是 U-Boot 并没有初始化这部分时钟。为了使驱动的通用性更强，笔者决定在驱动中添加相关的时钟初始化代码。这也说明了 Linux 系统移植工程师几乎必须要和驱动打交道。最后，Linux 驱动的更新不一定能跟得上硬件的更新，当使用了同一系列中较新的芯片时，往往也会涉及到驱动代码的修改。

（2）内核应用开发工程师。这个听起来有一点别扭，其实就是内核开发工程师，只是他们主要调用 Linux 内核的 API 接口来完成某些特定的功能。比如开一个内核线程来完成某件事情，绕过文件系统相关的代码来直接访问磁盘，过滤一些网络数据等等。这类开发者直接在内核层工作，通常以优化性能，提高程序效率或进行底层的监控为目标。这些代码都很有可能会调用到驱动所提供的接口。

（3）系统编程工程师。主要调用系统调用接口来完成应用软件的编程，通常是一些平台软件，SDK 之类的，为更上层的应用开发者提供编程接口。了解底层的工作原理，有助于写出更稳定、更高效的应用程序。

当然，Linux 内核源码是由全世界众多优秀开发者共同开发出来的，他的稳定性足以证明他设计的合理性，但凡从事软件开发的人员应该都能够从中得到一些启发或有所借鉴。

那么 Linux 驱动开发和一般的应用开发有哪些区别，或者这种开发具有哪些最鲜明的特点呢？笔者认为目前可以先谈到以下几个方面。

（1）Linux 驱动开发是内核级别的开发，驱动程序的任何问题都可能会引起整个系统的崩溃。因为应用程序由操作系统来管理，应用程序的崩溃通常不会影响到其他的程序或整个系统，他造成的破坏是比较小的。比如应用程序对非法指针进行解引用，通常只会引发一个段错误，然后程序本身崩溃而已。但是驱动如果对一个非法指针进行解引用的话，通常就会导致整个内核的崩溃。

（2）驱动程序通常都要进行中断处理，在中断上下文中（或类似的原子上下文中）的编

程有比较严格的限制（这在相关的章节会做更详细的描述），处理不好也会导致内核崩溃。而在应用程序中则没有这些方面的内容。

（3）驱动程序有更多的并发环境需要考虑，比如上面说到的中断，还有就是在多处理器系统下的驱动编程。一个好的驱动，不应该假设自己的运行环境，或者说都应该假设运行在各种并发环境下。

（4）驱动程序是被动接受上层调用的代码，是为上层提供服务的一套代码，所以我们会在驱动中看到很多注册和注销的函数。

（5）一类驱动程序都有一个特定的实现模板，在这里姑且称之为驱动的框架。另外，所有的驱动都有一种类似的实现模式，就是构造核心的数据结构，然后注册到内核。我们学习驱动开发，主要是在学习这些核心的数据结构和与之相关的一套 API。编写驱动则是按照特定类型驱动的框架来构造这些核心数据结构，然后再注册到内核。驱动中灵活的部分是这些框架规定的接口函数的实现。

（6）驱动程序虽然是用 C 语言来开发的，但是很多地方都体现了面向对象的编程思想，这在 Linux 设备模型中体现得尤为突出。

（7）应该尽量利用内核中已有的实现，而不是自己从头来构建，本书中有几个驱动的例子都是这样的。还有就是，内核源码中有同一类设备驱动很好的实现范例，在不熟悉一种驱动的框架前，我们可以参考内核源码，从而快速掌握该类驱动的开发。

（8）Linux 内核是基于 GNU C 来进行开发的，他对标准的 C 语言做了一些扩展，但是本书的示例代码都避免使用这些扩展的特性，尽量和标准 C 一致。还有就是，Linux 内核代码有一套编码风格，大家请参考 https://www.kernel.org/doc/Documentation/CodingStyle，在此就不细说了。

上面几点只是一个概要性的描述，有些涉及到了具体的知识点就没有细说，或者根本没有说，这在本书的具体章节还会进行更进一步的归纳和总结。接下来对本书的各个章节和核心内容做一个简要的说明。

Linux 的内核源码编译后将会生成一个总的镜像，将该镜像加载到内存中并运行之，就会启动内核。驱动属于内核代码的一部分，当每次对驱动做任何修改都会重新编译整个内核，还要重新加载运行内核，这个时间消耗是比较大的。所以如果驱动能独立于内核镜像之外，并能动态加载和卸载的话，那么驱动开发的时间消耗将会大大降低，本书第 2 章讨论的就是这个独立于内核之外，并能动态加载和卸载的模块。

有一类设备的数据访问是按字节流的方式来进行的，也就是访问的单位可以小到字节，比如说键盘、鼠标等。针对这一类设备的 Linux 驱动叫字符设备驱动，是开发中最有可能遇到的一类驱动，也是内核中最多的一类驱动。我们前面说过，学习一类驱动就是要学习他的核心数据结构和一组 API，然后由此而组成的框架。在第 3 章我们将会首次接触到这个概念，我们也会注册一些接口函数，从而为上层提供服务，或者说供上层调用。掌握了这些概念后，应用同样的方式，就可以快速地掌握其他驱动的框架。

在类 Unix 系统中，设备也当成文件来对待，或者说将设备抽象成了文件。这样就可以统一应用层代码对普通文件和设备文件的访问接口。对于文件的操作，有很多种 I/O 模型，比如大家最熟悉的阻塞、非阻塞，或者 Windows 系统经常提到的同步和异步。既然设备被抽象成了文件，而设备又是最终被驱动程序所管理的，那么自然设备的驱动就应该提供这些 I/O

模型的支持，本书第 4 章将重点阐述这方面的内容。

前面说过，Linux 驱动编程相对于应用程序编程多了中断的处理。因为驱动是管理硬件的，而为了提高硬件的访问效率，通常不是由 CPU 来轮询硬件的状态，而是在硬件准备好后主动通知 CPU，这种硬件上的异步通知就是中断。第 5 章将会讨论在驱动中如何编写中断服务例程，以及中断服务例程中的限制和一些对应的策略。另外，传统的硬件定时器也是以中断方式工作的，所以在这一章还讨论了延时和定时方面的内容。

前面也提到过，在内核中的并发方式多于应用层中遇到的并发。为了避免并发的执行路径对共享数据访问带来的相互覆盖的问题，我们需要认真处理好这个问题。不管在应用层还是在内核层，数据的紊乱都可能会导致灾难性的结果，但是因为内核层的这种情况更多，关系更为复杂，所以要更谨慎对待之。当然，内核提供的解决方案也要更多一些，不过这也带来了另外一个问题，就是如何从这些方案中选择一种最合适的方案。本书第 6 章将会详细讨论这方面的内容。

关于内存的使用，内核层提供了更多的选择。Linux 向来以高效、稳定著称，内存是计算机系统中的宝贵资源，在内存的管理上下再多的功夫都不为过。当然，Linux 在这方面考虑得很周全，所以他才能在小到手表大小的嵌入式系统上和大到集群服务器这样的系统上都能运行自如。在本书第 7 章我们将会从驱动编程实用性的角度来学习相关的知识点。另外，驱动会利用 DMA 操作来减轻 CPU 的负担，所以第 7 章也会讨论在嵌入式系统上的 DMA 编程接口。

自从 Linus 抱怨了 ARM 体系结构相关内核代码的混乱后，ARM 社区就开始积极处理这个问题，最后引入了设备树。在第 8 章中，我们首先会细数前面那种驱动开发模式的弊端，然后逐渐按照历史的顺序来还原这个 Linux 设备模型的产生和升级过程。在第 8 章也会详细阐明驱动开发中的设备和驱动分离的思想，这种思想使得驱动能够动态获取设备的信息，而不是将设备信息硬编码在驱动中，从而提高了驱动的灵活性。这也是减轻 Linux 系统移植工作量的关键所在。

学习驱动的目的最终是为了能为各种各样的设备写出驱动代码。本书第 9 章将会针对一个嵌入式目标板上常见的外设，从原理图和芯片手册出发，配合驱动的框架，来逐一实现这些设备的驱动。前面也说过，我们要善于利用内核中已有的设施，以最快最简单的方式来实现设备的驱动。所以在第 9 章，部分外设在自己用代码实现了驱动后，还会分析内核中已有的驱动，并利用这些驱动来达到访问、控制设备的目的。还有一类连接在总线上的设备，在第 10 章将会对之进行讨论。第 10 章首先分析了这些总线类设备驱动的框架，然后再用实例来做展示。

Linux 除了前面谈到的字符设备驱动这一大类，还有块设备驱动和网络设备驱动这两大类。相对于字符设备驱动而言，这两类设备的驱动要少一些，但是会更复杂一些。这包含两个方面，第一是框架要复杂一些，涉及到的内核组件要多一些，第二是硬件本身要复杂一些。为了突出框架的主体，避免过多涉及到硬件的细节，选择用虚拟硬件的方式来实现这两类设备的驱动。

出于完整性考虑，本书的第 13 章讨论了内核的几种调试方法，第 14 章详细描述了 Linux 驱动程序开发环境的搭建，如果对 Linux 驱动开发环境不熟悉的读者，请首先阅读第 14 章的内容。

第 2 章　内核模块

本章目标

绝大多数的驱动都是以内核模块的形式来实现的。本章主要围绕什么是内核模块，以及如何编写、编译、加载并测试模块程序来进行展开。另外，本章还将会讨论模块的一些其他重要特性。

Linux 是宏内核（或单内核）的操作系统典型代表，他和微内核（典型的代表是 Windows 操作系统）的最大区别在于所有的内核功能都被整体编译在一起，形成一个单独的内核镜像文件。其显著的优点就是效率非常高，内核中各功能模块的交互是通过直接的函数调用来进行的。而微内核则只实现内核中相当关键和核心的一部分，其他功能模块被单独编译，功能模块之间的交互需要通过微内核提供的某种通信机制来建立。对于像 Linux 这类的宏内核而言，其缺点也是不言而喻的，如果要增加、删除、修改内核的某个功能的话，不得不重新编译整个内核，然后重新启动整个系统。这对驱动开发者来说基本上是不可接受的，因为驱动程序的特殊性，在驱动开发的初期，需要经常修改驱动的代码，即便是经验丰富的驱动开发者也是如此。

为了弥补这一缺点，Linux 引入了内核模块（后面在不至于引起混淆的情况下简称模块）。简单地说，内核模块就是被单独编译的一段内核代码，他可以在需要的时候动态地加载到内核，从而动态地增加内核的功能。在不需要的时候，可以动态地卸载，从而减少内核的功能，并节约一部分内存（这要求内核配置了模块可卸载的选项才行）。而不论是加载还是卸载，都不需要重新启动整个系统。这种特性使得他非常适合于驱动程序的开发（注意，内核模块不一定都是驱动程序，驱动程序也不一定都是模块的形式）。驱动开发者可以随时修改驱动的代码，然后仅仅是编译驱动代码本身（而非整个内核），并将新编译的驱动加载到内核进行测试。只要新加入的驱动不会使得内核崩溃，那么可以不用重新启动系统。

内核模块的这一特点也有助于减小内核镜像文件的体积，自然也就减少了内核所占用的内存空间（因为整个内核镜像将会被加载到内存中运行）。不必把所有的驱动都编译进内核，而是以模块的形式单独编译驱动程序，这是基于不是所有的驱动都会同时工作的原理。因为不是所有的硬件都要同时接入系统，比如一个 USB 无线网卡。

讨论完内核模块的这些特性后，接下来我们正式开始编写模块程序。

2.1 第一个内核模块程序

```
/* vser.c */
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 int init_module(void)
 6 {
 7         printk("module init\n");
 8         return 0;
 9 }
10
11 void cleanup_module(void)
12 {
```

```
13          printk("cleanup module\n");
14 }
```

内核模块程序和一般的应用程序一样，也需要包含相应的头文件，只不过这里包含的头文件都是内核源码的头文件。例如第一行代码包含的<linux/init.h>就是内核源码树中的include/linux/init.h 头文件。<linux/init.h>头文件包含了这里的 init_module 和 cleanup_module 的两个函数原型声明；<linux/kernel.h>头文件包含了这里的 printk 函数的原型声明；<linux/module.h>头文件在这里暂时没用，不过在后面添加的代码中将会用到该文件中的一些声明。一个模块程序几乎都要直接或间接包含上面三个头文件。

程序第 5 行到第 9 行，是模块的初始化函数，在模块被动态加载到内核时被调用。该函数的返回值为 int 类型，返回 0 表示模块的初始化函数执行成功，否则通常返回一个负值表示失败。函数不接受参数，函数体中调用 printk 向控制台输出“module init”，然后返回 0，表示模块的初始化函数执行成功。这里的 printk 函数类似于应用程序中的 printf，只是 printk 函数支持额外的打印级别，在内核源码树中的“include/linux/printk.h”头文件有关于该函数的详细描述，在本书的 13 章中也有该函数的详细说明，这里就不做过多的讨论。这里添加的 printk 打印主要是用于调试目的，用于演示模块在加载时调用了 init_module 这个函数，不是必须的。不仅如此，init_module 函数也不是必须的。内核在加载模块时，如果没有发现该函数，则不会调用。模块初始化函数的作用正如其名字一样，将会对某些对象进行初始化，比如进行内存的分配，驱动的注册等等。随着之后学习的深入，我们会更进一步看到该函数中通常应该实现的代码。

程序第 11 行到第 14 行，是模块的清除函数，在模块从内核中卸载时被调用。该函数没有返回类型，也不接受任何参数。调用 printk 打印“cleanup module”同样出于调试的目的。该函数同样不是必须的，在模块卸载时如果没有发现该函数，则不调用。但是，如果提供了模块初始化函数，那么就应该提供一个对应的模块清除函数（除非不允许内核模块卸载）。顾名思义，该函数主要完成清除性的操作，是初始化函数的逆操作，通常完成内存释放，驱动注销等。

看完了内核模块的代码后，接下来则是与之配对的 Makefile 文件。简单的做法可以把刚才的代码添加到内核源码树中，然后修改对应的 Makefile 文件即可，但这会修改内核的源码。另一种常用的做法则是将刚才的 c 文件放在内核源码树外的一个单独的目录，然后在该目录下编写一个对应的 Makefile，该 Makefile 的内容如下。

```
# Makefile
 1 ifeq ($(KERNELRELEASE),)
 2
 3 ifeq ($(ARCH),arm)
 4 KERNELDIR ?= /home/farsight/fs4412/linux-3.14.25-fs4412
 5 ROOTFS ?= /nfs/rootfs
 6 else
 7 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 8 endif
 9 PWD := $(shell pwd)
```

```
     10
     11 modules:
     12         $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
     13 modules_install:
     14         $(MAKE) -C $(KERNELDIR) M=$(PWD) INSTALL_MOD_PATH=$(ROOTFS) modu
les_install
     15 clean:
     16         rm -rf *.o *.ko .*.cmd *.mod.* modules.order Module.symvers .tmp
_versions
     17 else
     18
     19 obj-m := vser.o
     20
     21 endif
```

代码被外层的 ifeq…else…endif 语句分为了两部分，第一部分是在 KERNELRELEASE 变量值为空的情况下执行的代码（代码第 1 行至第 16 行），第二部分则与之相反（代码第 17 行至第 21 行）。KERNELRELEASE 是内核源码树中顶层 Makefile 文件中定义的一个变量，并对其赋值为 Linux 内核源码的版本，该变量会用 export 导出，从而可以在子 Makefile 中使用该变量。

在模块目录下执行 make 操作，将会导致 make 工具对当前目录下的 Makefile 的解释执行。即会解释执行上面的 Makefile 文件，第一次解释执行该 Makefile 时，代码第 1 行的 KERNELRELEASE 变量没有被定义，也没有被赋值，所以 ifeq 条件成立，则解释执行第一部分的内容。第一部分内容中包含了内核源码目录的变量 KERNELDIR 的定义，并且根据是编译 ARM 平台下的驱动还是 PC 机上运行的驱动对该变量进行了不同的赋值，这样我们可以在命令行中对 ARCH 进行赋值来选择编译哪个平台下运行的驱动，其中“/home/farsight/fs4412/linux-3.14.25-fs4412”是上一章中谈到的 FS4412 目标板的内核源码目录。如果是编译 ARM 平台下的驱动，则还对根文件系统目录的变量 ROOTFS 进行了定义和赋值。接下来是对当前的模块所在的目录的变量 PWD 定义（代码第 9 行）。Makefile 文件中的第一个目标 modules 为默认目标（代码第 11 行），执行 make 而不跟参数，则会默认生成该目标。而生成该目标就是要执行第 12 行的命令，在代码第 12 行中，$(MAKE)相当于 make，主要用于平台的兼容。代码第 12 行的含义是进入到内核源码目录（由“-C $(KERNELDIR) ”指定）编译在内核源码树之外的一个目录（由“M=$(PWD)”指定）中的模块（由最后的“modules”指定）。

当编译过程折返回来（退出内核源码目录，再次进入模块目录，由“M=$(PWD)”指定）编译模块时，上述的 Makefile 第二次被解释执行，不过这一次的情况和上一次不同，主要体现在 KERNELRELEASE 变量已经被赋值，并且被导出，导致 ifeq 条件不成立，那么将解释执行 Makefile 的第二部分，即外层 else 和 endif 之间的部分。其中 obj-m 表示将后面跟的目标编译成一个模块。相关的说明请参见内核文档中的 Kbuild 部分。

modules_install 目标表示把编译之后的模块安装到指定目录，安装的目录为

$(INSTALL_MOD_PATH)/lib/modules/$(KERNELRELEASE)，在没对 INSTALL_MOD_PATH 赋值的情况下，模块将会被安装到/lib/modules/$(KERNELRELEASE)目录下。关于 modules 和 modules_install 的更详细解释，请在内核源码树下执行 make help 命令，查看相应的帮助信息。

clean 目标则用于清除 make 生成的中间文件。

上面两个文件的完整内容请参见“光盘资料/程序源码/module /ex1”。

模块的源文件和 Makefile 文件编写完成后，执行下面的命令即可完成对运行在 PC 机上的模块进行编译和安装（注意，根据运行平台的不同，安装的目录会不同。如果权限不够，请用 root 用户）。

```
# make
# make modules_install
```

如果要编译运行在 ARM 目标机上的驱动，则使用下面的命令。

```
# make ARCH=arm
# make ARCH=arm modules_install
```

编译完成后，在当前目录下会生成一个.ko 的文件。安装成功后，如果是 PC 机平台，则将会把生成的.ko 文件拷贝到/lib/modules/3.13.0-32-generic/extra 目录下（3.13.0-32-generic 是内核源码的版本，视内核源码版本的不同而不同）；如果是目标板平台，则会把生成的.ko 文件拷贝到/nfs/rootfs/lib/modules/3.14.25/extra 目录下（3.14.25 是内核源码的版本，视内核源码版本的不同而不同）。

2.2 内核模块的相关工具

模块相关工具及使用说明如下。

（1）模块加载

insmod：加载指定目录下的一个.ko 到内核。比如加载刚编译好的模块，可以用下面命令中的一个。

```
# insmod vser.ko
# insmod /lib/modules/3.13.0-32-generic/extra/vser.ko
```

模块加载成功后，使用 dmesg 命令，将会看到控制台有如下输出。

```
[   83.884417] vser: module license 'unspecified' taints kernel.
[   83.884423] Disabling lock debugging due to kernel taint
[   83.885685] module init
```

其中“module init”是加载模块时，调用了模块的初始化函数，是模块的初始化函数中的打印输出。

modprobe：自动加载模块到内核，相对于 insmod 来讲更智能，推荐使用该命令。但前提条件是模块要执行安装操作，在运行该命令前最好运行一次 depmod 命令来更新模块的依

赖信息（后面会更详细地说明）。用法如下(注意，使用 modprobe 不指定路径及后缀)。

```
# depmod
# modprobe vser
```

（2）模块信息

modinfo：查看模块的信息，在安装了模块并运行 depmod 命令后，可以不指定路径和后缀，也可以指定查看某一特定.ko 文件的模块信息，示例如下。

```
# modinfo vser
filename:       /lib/modules/3.13.0-32-generic/extra/vser.ko
srcversion:     533BB7E5866E52F63B9ACCB
depends:
vermagic:       3.13.0-32-generic SMP mod_unload modversions 686
```

（2）模块卸载

rmmod：如果内核配置为允许卸载模块，那么 rmmod 将指定的模块从内核中卸载。示例如下。其中“cleanup module”是卸载模块时调用了模块清除函数，在模块清除函数中的打印输出。

```
# rmod vser
# dmesg
......
[  823.366584] cleanup module
```

上面的命令要在目标板上运行的话，请先按照 14.5 章节的内容在目标板上运行 Linux 系统，然后按照前面说的编译和安装方法在 Ubuntu 开发主机上进行编译和安装。最后在串口终端上执行上面的命令进行测试。

2.3 内核模块更一般的形式

在前面的模块加载实验中，我们看到内核有如下打印信息的输出。

```
[   83.884417] vser: module license 'unspecified' taints kernel.
[   83.884423] Disabling lock debugging due to kernel taint
```

其大概意思是因为加载了 vser 这个模块而导致内核被污染，并且因此而禁止了锁的调试功能。那这是什么原因造成的呢?众所周知，Linux 是一个开源的项目，为了使得 Linux 在发展的过程中不至于成为一个闭源的项目，这就要求任何使用 Linux 内核源码的个人或组织在免费获得源码并可针对源码做任意的修改和再发布的同时，也必须将修改后的源码发布。这就是所谓的 GPL 许可证协议。在此并不讨论该许可证协议的详细内容，而是讨论在代码中如何来反应我们接受该许可证协议。在代码中我们需要添加如下的代码来表示该代码接受相应的许可证协议。

```
MODULE_LICENSE("GPL");
```

MODULE_LICENSE 是一个宏，里面的参数是一个字符串，代表的是相应的许可证协议。可以是："GPL"、"GPL v2"、"GPL and additional rights"、"Dual BSD/GPL"、"Dual MIT/GPL"、"Dual MPL/GPL"等，详细内容请参见 include/linux/module.h 头文件。这个宏将会生成一些模块信息，放在 ELF 文件中的一个特殊的段中，模块在加载时会将该信息拷贝到内存中，并检查该信息。可能读者会认为不加这行代码，即不接受那些许可证协议只是导致内核报一些警告或关闭某些调试功能而已，对于可以不开源的这个结果，这个代价似乎是可以接受的。但是正如本章的后面我们会看到的一样，没有这行代码的话，内核中的某些功能函数是不能够调用的，而我们在开发驱动时几乎不可避免地要去使用内核中的一些基础设施，即调用一些内核的 API 函数。

除了 MODULE_LICENSE 而外，还有很多类似的描述模块信息的宏，比如 MODULE_AUTHOR 用户描述模块的作者信息，通常包含作者的姓名和邮箱地址；MODULE_DESCRIPTION 用于模块的详细信息说明，通常是该模块的功能说明；MODULE_ALIAS 提供了给用户空间使用的一个更合适的别名。

模块的初始化函数和清除函数的名字是固定的，如同 C 的应用程序一样，入口函数基本上都叫 main。这对于追求个性化和更想表达函数真实意图的我们来说显得比较呆板了一些。幸亏内核借助于 GNU 的函数别名机制，使得我们可以更灵活地指定模块的初始化函数和清除函数的别名。

```
module_init(vser_init);
module_exit(vser_exit);
```

上面的 module_init 和 module_exit 是两个宏，分别用于指定了 init_module 的函数别名是 ver_init，以及 cleanup_module 的别名是 vser_exit。那么我们的模块初始化函数和清除函数就可以用别名来定义了。

函数名可以任意指定又带来了一个新的问题，那就是可能会和内核中已有的函数重名，因为模块的代码最终也是属于内核代码的一部分。C 没有类似于 C++的命名空间的概念，为了避免因为重名而带来的重复定义的问题，函数可以加 static 关键字修饰。经过 static 修饰后的函数的链接属性为内部，从而解决了该问题。这就是几乎所有的驱动程序的函数前都要加 static 关键字修饰的原因。

Linux 是节约内存的操作系统的典范，任何可能节约下来的内存都不会放过。上面的模块代码看上去已经足够简单了，但仔细思考，还是会发现一些可以优化的地方。模块的初始化函数有且仅会被调用一次，在调用完成后，该函数不应该被再次调用。所以该函数所占用的内存应该被释放掉，在函数名前加__init 可以达到此目的。__init 是把标记的函数放到 ELF 文件的特定代码段，在模块加载这些段时将会单独分配内存，这些函数调用成功后，模块的加载程序会释放这部分内存空间。__exit 用于修饰清除函数，和__init 的作用类似，但用于模块的卸载，如果模块不允许卸载，那么这段代码完全就不用加载。

加入上述的内容后，一个模块程序的代码形式更像下面的样子（完整的代码请参见“光盘资料/程序源码/module /ex2”）。

```
/* vser.c */
  1 #include <linux/init.h>
```

```
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 static int __init vser_init(void)
 6 {
 7         printk("vser_init\n");
 8         return 0;
 9 }
10
11 static void __exit vser_exit(void)
12 {
13         printk("vser_exit\n");
14 }
15
16 module_init(vser_init);
17 module_exit(vser_exit);
18
19 MODULE_LICENSE("GPL");
20 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
21 MODULE_DESCRIPTION("A simple module");
22 MODULE_ALIAS("virtual-serial");
```

按照前面的方法编译、安装、加载后，使用 modinfo 命令可以查看到如下的信息。用 dmesg 查看打印信息也不会看到之前的内核被污染之类的信息。

```
filename:       /lib/modules/3.13.0-32-generic/extra/vser.ko
alias:          virtual-serial
description:    A simple module
author:         Kevin Jiang <jiangxg@farsight.com.cn>
license:        GPL
srcversion:     693EDC6D54EC62B7A38982A
depends:
vermagic:       3.13.0-32-generic SMP mod_unload modversions 686
```

对于模块的加载也可以使用别名，命令如下。

```
# modprobe virtual-serial
```

2.4 多个源文件编译生成一个内核模块

对于一个比较复杂的驱动程序，将所有的代码写在一个源文件中通常是不太现实的。我

们通常会把程序的功能进行拆分，由不同的源文件来实现对应的功能，应用程序是这样的，驱动也是如此。下面的这个简单例子演示了如何用多个源文件来生成一个内核模块（完整的代码请参见“光盘资料/程序源码/module /ex3”）。

```
/* foo.c */
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 extern void bar(void);
 6
 7 static int __init vser_init(void)
 8 {
 9         printk("vser_init\n");
10         bar();
11         return 0;
12 }
13
14 static void __exit vser_exit(void)
15 {
16         printk("vser_exit\n");
17 }
18
19 module_init(vser_init);
20 module_exit(vser_exit);
21
22 MODULE_LICENSE("GPL");
23 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
24 MODULE_DESCRIPTION("A simple module");
25 MODULE_ALIAS("virtual-serial");
```

```
/* bar.c */
 1 #include <linux/kernel.h>
 2
 3 void bar(void)
 4 {
 5         printk("bar\n");
 6 }
```

```
/* Makefile */
 1 ifeq ($(KERNELRELEASE),)
```

```
 2
 3 ifeq ($(ARCH),arm)
 4 KERNELDIR ?= /home/farsight/fs4412/linux-3.14.25-fs4412
 5 ROOTFS ?= /nfs/rootfs
 6 else
 7 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 8 endif
 9 PWD := $(shell pwd)
10
11 modules:
12         $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
13 modules_install:
14         $(MAKE) -C $(KERNELDIR) M=$(PWD) INSTALL_MOD_PATH=$(ROOTFS) modu
les_install
15 clean:
16         rm -rf *.o *.ko .*.cmd *.mod.* modules.order Module.symvers .tmp
_versions
17 else
18
19 obj-m := vser.o
20 vser-objs = foo.o bar.o
21
22 endif
```

上面的代码相对于以前的变化是：新增了 bar.c，在里面定义了一个 bar 函数；vser.c 更名为 foo.c，调用了 bar 函数，并添加相应的函数声明。最重要的修改时在 Makefile 中，第 20 行“vser-objs = foo.o bar.o”表示 vser 模块是由 foo.o 和 bar.o 两个目标文件来共同生成的。编译、安装和测试的方法和前面的相同。

2.5 内核模块参数

通过前面的了解，我们知道模块的初始化函数在模块被加载时调用。但是该函数不接受参数，如果我们想在模块加载时对模块的行为进行控制的话，就不是那么方便了。比如编写了一个串口驱动，想要在串口驱动加载时，波特率由命令行参数设定，就像运行一个普通的应用程序，通过命令行参数来传递信息那样。为此模块提供了另外一种形式来支持这种行为，这就叫做模块参数。

模块参数允许用户在加载模块时通过命令行指定参数值，在模块的加载过程中，加载程序会得到命令行参数，并转换成相应类型的值，然后赋值给对应的变量，这个过程发生在调用模块初始化函数之前。内核支持的参数类型有：bool、invbool（反转值 bool 类型）、charp

（字符串指针）、short、int、long、ushort、uint、ulong。这些类型又可以复合成对应的数组类型。为了说明模块参数的用法，下面分别使用了整型、整型数组和字符串类型为例来进行说明（完整的代码请参见“光盘资料/程序源码/module /ex4”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 static int baudrate = 9600;
 6 static int port[4] = {0, 1, 2, 3};
 7 static char *name = "vser";
 8
 9 module_param(baudrate, int, S_IRUGO);
10 module_param_array(port, int, NULL, S_IRUGO);
11 module_param(name, charp, S_IRUGO);
12
13 static int __init vser_init(void)
14 {
15         int i;
16
17         printk("vser_init\n");
18         printk("baudrate: %d\n", baudrate);
19         printk("port: ");
20         for (i = 0; i < ARRAY_SIZE(port); i++)
21                 printk("%d ", port[i]);
22         printk("\n");
23         printk("name: %s\n", name);
24
25         return 0;
26 }
27
28 static void __exit vser_exit(void)
29 {
30         printk("vser_exit\n");
31 }
32
33 module_init(vser_init);
34 module_exit(vser_exit);
35
36 MODULE_LICENSE("GPL");
```

```
37 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
38 MODULE_DESCRIPTION("A simple module");
39 MODULE_ALIAS("virtual-serial");
```

代码第 5 行到第 7 行分别定义了一个整型变量，一个整型数组和一个字符串指针。代码第 9 行到第 11 行将这三种类型的变量声明为模块参数。分别用到了 module_param 和 module_param_array 两个宏，两者的参数说明如下。

```
module_param(name, type, perm)
module_param_array(name, type, nump, perm)
```

name：变量的名字。

type：变量或数组元素的类型。

nump：数组元素个数的指针，可选。

perm：在 sysfs 文件系统中对应文件的权限属性。

权限的取值请参见< linux/stat.h >头文件，含义和普通文件的权限是一样的。但是如果 perm 为 0 的话，在 sysfs 文件系统中将不会出现对应的文件。

按照前面说的方法编译、安装模块后，在加载模块时，如果不指定模块参数的值，那么使用的命令和内核的打印信息如下。

```
# modprobe vser
# dmesg
[54925.319528] vser_init
[54925.319535] baudrate: 9600
[54925.319536] port: 0 1 2 3
[54925.319539] name: vser
```

可见打印的值都是代码中的默认值。如果需要指定模块参数的值，可以使用下面的命令。

```
# modprobe vser baudrate=115200 port=1,2,3,4 name="virtual-serial"
# dmesg
[55234.889650] vser_init
[55234.889655] baudrate: 115200
[55234.889656] port: 1 2 3 4
[55234.889659] name: virtual-serial
```

参看 sysfs 文件系统下的内容，可以发现和模块参数对应的文件及相应的权限。

```
$ ls -l /sys/module/vser/parameters/
total 0
-r--r--r-- 1 root root 4096 Jul  5 14:25 baudrate
-r--r--r-- 1 root root 4096 Jul  5 14:25 name
-r--r--r-- 1 root root 4096 Jul  5 14:25 port
$ cat /sys/module/vser/parameters/baudrate
115200
```

```
$ cat /sys/module/vser/parameters/port
1,2,3,4
$ cat /sys/module/vser/parameters/name
virtual-serial
```

虽然通过代码中增加模块参数的写权限使得用户可以通过 sysfs 文件系统来修改模块参数的值，但这并不推荐使用。因为通过这种方式对模块参数的修改对模块本身而言是一无所知的。

2.6 内核模块依赖

在介绍模块依赖之前，首先让我们看看导出符号。在之前的模块代码中，都是用到了 printk 这个函数，很显然，这个函数不是我们来实现的，他是内核代码的一部分。我们的模块之所以能够编译通过，那是因为对于模块的编译仅仅是编译，并没有链接。编译出来的.ko 文件是一个普通的 ELF 目标文件，使用 file 命令和 nm 命令，可以得到相关的细节信息。

```
$ file vser.ko
vser.ko: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), BuildID
[sha1]=0x09ca747e6f75c65b19a5da9102113b98d7ce2a47, not stripped
$ nm vser.ko
......
00000004 d port
         U printk
00000000 t vser_exit
00000000 t vser_init
```

使用 nm 查看模块目标文件的符号信息时，可以看到 vser_exit 和 vser_init 的符号类型是 t，表示的是函数，而 printk 的符号类型是 U，表示他是一个未决符号。这表示在编译阶段不知道这个符号的地址，因为他被定义在其他文件中，没有放在模块代码中一起编译。那么 printk 函数的地址怎么来解决呢，让我们来看看 printk 的实现代码（位于 kernel/printk/printk.c）。

```
1674 asmlinkage int printk(const char *fmt, ...)
1675 {
1676         va_list args;
......
1692 }
1693 EXPORT_SYMBOL(printk);
```

代码中通过一个叫 EXPORT_SYMBOL 的宏，将 printk 导出，其目的是为动态加载的模块提供 printk 的地址信息。大致的工作原理是：利用 EXPORT_SYMBOL 宏生成一个特定的结构并放在 ELF 文件的一个特定的段中，在内核的启动过程中，会将符号的确切地址填充到那个结构的特定成员中。模块加载时，加载程序将会去处理未决符号，用符号的名字在那个

特殊的段中搜索，如果找到，则将获得的地址填充在被加载的模块的相应的段中，这样符号的地址就可以确定。使用这种方式处理未决符号，其实是相当于把链接的过程推后，进行了动态链接。这和普通的应用程序使用共享库函数的道理是类似的。可以发现，内核将会有大量的符号导出，为模块提供了丰富的基础设施。

通常情况下，一个模块只使用内核导出的符号，自己不导出符号。但是一个模块需要提供全局变量或函数给另外的模块使用，那么就需要将这些符号导出。这在一个驱动程序代码调用另外一个驱动程序代码时比较常见。这样模块和模块之间就形成了依赖关系，使用导出符号的模块将会依赖于导出符号的模块，接下来的代码说明了这一点（完整的代码请参见“光盘资料/程序源码/module /ex5”）。

```
/* vser.c */
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 extern int expval;
 6 extern void expfun(void);
 7
 8 static int __init vser_init(void)
 9 {
10         printk("vser_init\n");
11         printk("expval: %d\n", expval);
12         expfun();
13
14         return 0;
15 }
16
17 static void __exit vser_exit(void)
18 {
19         printk("vser_exit\n");
20 }
21
22 module_init(vser_init);
23 module_exit(vser_exit);
24
25 MODULE_LICENSE("GPL");
26 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
27 MODULE_DESCRIPTION("A simple module");
28 MODULE_ALIAS("virtual-serial");
```

```
/* dep.c */
```

```
 1 #include <linux/kernel.h>
 2 #include <linux/module.h>
 3
 4 static int expval = 5;
 5 EXPORT_SYMBOL(expval);
 6
 7 static void expfun(void)
 8 {
 9         printk("expfun");
10 }
11
12 EXPORT_SYMBOL_GPL(expfun);
13
14 MODULE_LICENSE("GPL");
15 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
```

```
# Makefile
 1 ifeq ($(KERNELRELEASE),)
 2
 3 ifeq ($(ARCH),arm)
 4 KERNELDIR ?= /home/farsight/fs4412/linux-3.14.25-fs4412
 5 ROOTFS ?= /nfs/rootfs
 6 else
 7 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 8 endif
 9 PWD := $(shell pwd)
10
11 modules:
12         $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
13 modules_install:
14         $(MAKE) -C $(KERNELDIR) M=$(PWD) INSTALL_MOD_PATH=$(ROOTFS) modu
les_install
15 clean:
16         rm -rf *.o *.ko .*.cmd *.mod.* modules.order Module.symvers .tmp
_versions
17 else
18
19 obj-m := vser.o
20 obj-m += dep.o
21
```

```
22 endif
```

上面的代码中，dep.c 里定义了一个全局变量 expval，定义了一个函数 expfun，并分别使用 EXPORT_SYMBOL 和 EXPORT_SYMBOL_GPL 导出。在 vser.c 首先用 extern 声明了这个变量和函数，并打印了该变量的值和调用了该函数。在 Makefile 中添加了第 20 行的代码，增加了对 dep 模块的编译。编译、安装后，使用下面的命令加载并查看内核的打印信息。

```
$ modprobe vser
$ dmesg
[58278.204677] vser_init
[58278.204683] expval: 5
[58278.204684] expfun
```

从上面的输出可以看到我们期望的信息。但是，这里有几点需要特别说明。

（1）如果使用 insmod 命令加载模块，则必须先加载 dep 模块，再加载 vser 模块，因为 vser 模块使用到了 dep 模块导出的符号，如果在 dep 模块没有加载的情况下加载 ver 模块，那么将会在加载的过程中因为处理未决符号而失败。从这里可以看出，modprobe 优于 insmod 的地方在于 modporbe 可以自动加载被依赖的模块。而这一切又要归功于 depmod 命令，depmod 命令将会生成模块的依赖信息，保存在/lib/modules/3.13.0-32-generic/modules.dep 文件中，其中 3.13.0-32-generic 是内核源码的版本，视版本的不同而不同。查看该文件可以发现 vser 模块所依赖的模块。

```
$ cat /lib/modules/3.13.0-32-generic/modules.dep
……


extra/vser.ko: extra/dep.ko
extra/dep.ko:
```

（2）两个模块存在依赖关系，如果两模块分别编译的话，将会出现类似于下面的警告信息，并且即便是加载顺序正确，加载也不会成功。

```
WARNING: "expfun" [/home/farsight/fs4412/driver/module/ex5/vser.ko] undefi
ned!
WARNING: "expval" [/home/farsight/fs4412/driver/module/ex5/vser.ko] undefi
ned!


$ sudo insmod dep.ko
$ sudo insmod vser.ko
insmod: error inserting 'vser.ko': -1 Invalid parameters
```

这是因为在编译 vser 模块时在内核的符号表中找不到 expval 和 expfun 的项，而 vser 模块又完全不知道 dep 模块的存在。解决的方法是两个模块放在一起编译，或者将 dep 模块放在内核源码中，先在内核源码下编译完所有的模块，再编译 vser 模块。

（3）卸载模块时要先卸载 vser，再卸载 dep，否则因为 dep 被 vser 模块使用，而不能卸

载。内核将会创建模块依赖关系的链表，只有当依赖于这个模块的链表为空时，模块才能被卸载。

2.7 关于内核模块的进一步讨论

Linux 的内核是由全世界的志愿者来开发的，对于这样一个组织，内核开发者会毫不顾虑的删除不适合的接口或者对接口进行修改，只要认为这是必要的。所以，往往在前一个版本这个接口函数是一种形式存在，而到了下一个版本，函数的接口就发生了变化。这对内核模块的开发具有重要的影响，这就是所谓的内核模块版本控制。在一个版本上编译出来的内核模块.ko 文件中详细记录了内核源码版本信息、体系结构信息、函数接口信息（通过 CRC 校验来实现）等，在开启了版本控制选项的内核中加载一个模块时，内核将会核对这些信息，如果不一致，则会拒绝加载。下面就是把一个在 3.13 内核版本上编译的内核模块放在 3.5 内核版本的系统上加载的相关输出信息。

```
$ modinfo vser.ko
filename:       vser.ko
alias:          virtual-serial
description:    A simple module
author:         Kevin Jiang <jiangxg@farsight.com.cn>
license:        GPL
srcversion:     BA8BD66A92BF5D4C7FA3110
depends:
vermagic:       3.13.0-32-generic SMP mod_unload modversions 686
$ uname -r
3.5.0-23-generic
# insmod vser.ko
insmod: error inserting 'vser.ko': -1 Invalid module format
# dmesg
......
[  599.260504] vser: disagrees about version of symbol module_layout
```

最后再总结一下内核模块和普通应用程序之间的差异。

（1）内核模块是操作系统内核的一部分，运行在内核空间；而应用程序运行在用户空间。

（2）内核模块中的函数是被动被调用的，比如初始化函数和清除函数分别是在内核模块被加载和被卸载的时候调用，模块通常注册一些服务性质的函数供其它功能单元在之后调用。而应用程序则是顺序执行，然后通常进入一个循环反复调用某些函数。

（3）内核模块处于 C 函数库之下，自然就不能调用 C 库函数（内核源码中会实现类似的函数）；应用程序则可以随意调用 C 库函数。

（4）内核模块要做一些清除性的工作，比如在一个操作失败后或者在内核的清除函数中；而应用程序有些通常不需要做，比如在程序退出前关闭所有已打开的文件。

（5）内核模块如果产生了非法访问（比如对野指针的访问），将很有可能会导致整个系统的崩溃；而应用程序通常只影响自己。

（6）内核模块中的并发更多，比如中断、多处理器。而应用程序一般只考虑多进程或多线程。

（7）整个内核空间的调用链上只有 4K 或 8K 大小的栈，相对于应用程序来说非常的小。所以如果需要大的内存空间，通常应该动态分配。

（8）虽然 printk 和 printf 的行为非常相似，但是通常 printk 不支持浮点数，例如要打印一个浮点变量，在编译时通常会出现如下警告，并且模块也不会加载成功。

```
    WARNING: "__extendsfdf2" [/home/farsight/fs4412/driver/module/ex5/vser.ko]
 undefined!
    WARNING: "__truncdfsf2" [/home/farsight/fs4412/driver/module/ex5/vser.ko]
undefined!
    WARNING: "__divdf3" [/home/farsight/fs4412/driver/module/ex5/vser.ko] unde
fined!
    WARNING: "__floatsidf" [/home/farsight/fs4412/driver/module/ex5/vser.ko] u
ndefined!
```

2.8 习题

1、默认情况下，模块初始化函数的名字是（ ），模块清除函数的名字是（ ）。

[A] init_module

[B] cleanup_module

[C] mod_init

[D] mod_exit

2、加载模块可以用哪个命令（ ）。

[A] insmod

[B] rmmod

[C] depmod

[D] modprobe

3、查看模块信息用哪个命令（ ）。

[A] insmod

[B] rmmod

[C] modinfo

[D] modprobe

4、内核模块参数的类型不包括（ ）。

[A] 布尔
[B] 字符串指针
[C] 数组
[D] 结构

5、内核模块导出符号用哪个宏（ ）。
[A] MODULE_EXPORT
[B] MODULE_PARAM
[C] EXPORT_SYMBOL
[D] MODULE_LICENSE

6、内核模块能否调用 C 库的函数接口（ ）。
[A] 能
[B] 不能

7、内核模块代码中，我们能否定义任意大小的局部变量（ ）。
[A] 能
[B] 不能

第 3 章　字符设备驱动

本章目标

字符设备驱动是 Linux 系统中最多的一类设备驱动，同时也是相对较为简单的一类驱动。本章首先概要介绍几类设备驱动的特点，然后较详细地介绍了在编写字符设备驱动前所需要的一些必备基础知识，最后以逐步实现一个较完整的字符设备驱动的方式，来对字符设备驱动的编程进行一步步的展示。

- 字符设备驱动基础
- 字符设备驱动框架
- 虚拟串口设备
- 虚拟串口设备驱动
- 一个驱动支持多个设备

Linux 系统中根据驱动程序实现的模型框架将设备的驱动分为了三大类，这三大类驱动的特点分别如下。

（1）字符设备驱动：设备对数据的处理是按照字节流的形式进行的，可以支持随机访问，也可以不支持随机访问，因为数据流量通常不是很大，所以一般没有页高速缓存。典型的设备如串口、键盘、帧缓存设备等。以串口为例，串口对收发的数据长度没有具体要求，可以是任意多个字节；串口也不支持 lseek 操作，即不能定位到一个具体的位置进行读写，因为串口按顺序发送或接收数据；串口的数据通常保存在一个较小的 FIFO 中，并且不会重复利用 FIFO 中的数据。再者，帧缓存设备（就是我们通常说的显卡），也是一个字符设备，但他可以进行随机访问，这样我们就能修改某个具体位置的帧缓存数据，从而改变屏幕上的某些确定像素点的颜色。

（2）块设备驱动：设备对数据的处理是按照若干个块进行的，一个块有其固定的大小，比如 4096 字节，那么每次读写的数据就至少是 4096 个字节。这类设备都支持随机访问，并且为了提高效率，可以将之前用到的数据缓存起来，以便下次使用。典型的设备如硬盘、光盘、SD 卡等。以硬盘为例，一个硬盘的最小访问单位是一个扇区，一个扇区通常是 512 字节，那么块的大小就至少是 512 字节。我们可以访问硬盘中的任何一个扇区，也就是说硬盘支持随机访问。因为硬盘的访问速度非常慢，如果每次都去硬盘上获取数据，那么效率会非常低，所以一般将之前从硬盘上得到的数据放在一个叫做页高速缓存的内存中，如果程序要访问的数据是之前访问过的，那么程序会直接从页高速缓存中获得数据，从而来提高效率。

（3）网络设备驱动：顾名思义，他就是专门针对网络设备的一类驱动，其主要目的是进行网络数据的收发。

以上驱动程序的分类是按照驱动的模型框架进行分类的，现实生活中有的设备很难被严格界定为字符设备还是块设备。甚至有的设备同时具有两类驱动，如 MTD（存储技术设备，如闪存）。一个设备的驱动选择上述三类中的哪一类，还要看具体的使用场合和最终的用途。

3.1 字符设备驱动基础

在正式学习字符设备驱动的编写之前，我们首先来看看相关的基础知识。在类 UNIX 系统中，有一个众所周知的说法，即“一切皆文件”，当然网络设备是一个例外。这就意味着设备最终也会体现为一个文件，应用程序要对设备进行访问，最终就会转化为对文件的访问，这样做的好处是统一了对上层的接口。设备文件通常位于/dev 目录下，使用下面的命令可以看到很多设备文件及其相关的信息。

```
$ ls -l /dev
total 0
......
brw-rw----  1 root disk      8,   0 Jul  4 10:07 sda
brw-rw----  1 root disk      8,   1 Jul  4 10:07 sda1
brw-rw----  1 root disk      8,   2 Jul  4 10:07 sda2
brw-rw----  1 root disk      8,   5 Jul  4 10:07 sda5
```

```
......
crw--w----  1 root tty      4,   0 Jul  4 10:07 tty0
crw-rw----  1 root tty      4,   1 Jul  4 10:07 tty1
......
```

上面列出的信息中，前面的字母“b”表示的是块设备，“c”表示的是字符设备。比如上面的 sda、sda1、sda2、sda5 就是块设备，实际上这些设备是笔者的 Ubuntu 主机上的一个硬盘和这个硬盘上的三个分区，其中 sda 表示的是整个硬盘，而 sda1、sda2、sda5 分别是三个分区。tty0、tty1 就是终端设备，shell 程序使用这些设备来同用户进行交互。从上面的打印信息来看，设备文件和普通文件有很多相似之处，都有相应的权限，所属的用户和组，修改时间和名字。但是设备文件会比普通文件多出两个数字，这两个数字分别叫主设备号和次设备号。这两个号是设备在内核中的身份或标志，是内核用于区分不同设备的唯一信息。通常内核用主设备号区别一类设备，次设备号用于区分同一类设备的不同的个体或不同的分区。而路径名则是用户层用于区别设备的信息。

现在的 Linux 系统中，设备文件通常是自动创建的。即便如此，我们还是可以通过 mknod 命令来手动创建一个设备文件，如下所示。

```
# mknod /dev/vser0 c 256 0
# ls -li /dev/vser0
126695 crw-r--r-- 1 root root 256, 0 Jul 13 10:03 /dev/vser0
```

那么 mknod 命令具体做了什么呢?mknod 是 make node 的缩写，顾名思义就是创建了一个节点（所以设备文件有时又叫做设备节点）。在 Linux 系统中，一个节点代表一个文件，创建一个文件的最主要的根本工作就是要分配一个新的节点（注意，这是存在于磁盘上的节点，之后我们还会看到位于内存中的节点 inode），包含节点号的分配（节点号在一个文件系统中是唯一的，可以依此来区别不同的文件。如上面 ls 命令的-i 选项就列出了/dev/vser0 设备的节点号为 126695），然后初始化好这个新节点（包含文件模式、访问时间、用户 ID、组 ID、等元数据信息，如果是设备文件则还要初始化好设备号），接下来将这个初始化好的节点写入磁盘。另一方面，需要在文件所在目录下添加一个目录项，目录项中包含了前面分配的节点号和文件的名字，然后写入磁盘。存在于磁盘上的这个节点用一个结构封装，下面用 Linux 系统中最常见的 ext2 文件系统为例进行说明。

```
/* fs/ext2/ext2.h */

294 /*
295  * Structure of an inode on the disk
296  */
297 struct ext2_inode {
298         __le16  i_mode;         /* File mode */
299         __le16  i_uid;          /* Low 16 bits of Owner Uid */
300         __le32  i_size;         /* Size in bytes */
301         __le32  i_atime;        /* Access time */
```

```
302         __le32  i_ctime;       /* Creation time */
303         __le32  i_mtime;       /* Modification time */
……
320         __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
……
349 };
```

ext2_inode 是一个最终会写在磁盘上的一个 inode，可以很清楚地看到，刚才所述的元数据信息包含在该结构中。另外，对于 i_block 成员，如果是普通文件的话，这个数组存放的是真正的文件数据所在的块号（可以看成是对文件数据块的索引，所以 EXT2 文件按照索引方式存储文件，其性能远远优于 FAT 格式），而如果是设备文件，那么这个数组的被用来存放设备的主次设备号，可以从下面的代码得出结论。

```
/* fs/ext2/inode.c */


1435 static int __ext2_write_inode(struct inode *inode, int do_sync)
1436 {
……
1443        struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
……
1456        raw_inode->i_mode = cpu_to_le16(inode->i_mode);
……
1513        if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode)) {
1514                if (old_valid_dev(inode->i_rdev)) {
1515                        raw_inode->i_block[0] =
1516                                cpu_to_le32(old_encode_dev(inode->i_rde
v));
1517                        raw_inode->i_block[1] = 0;
1518                } else {
1519                        raw_inode->i_block[0] = 0;
1520                        raw_inode->i_block[1] =
1521                                cpu_to_le32(new_encode_dev(inode->i_rde
v));
1522                        raw_inode->i_block[2] = 0;
1523                }
……
}
```

代码 1443 行获得了一个要写入磁盘的 ext2_inode 结构，并初始化了部分成员，代码第 1513 行到 1523 行，判断了设备的类型，如果是字符设备或块设备，那么将设备号写入 iblock 的前 2 个或前 3 个元素，其中 ionde 的 i_rdev 成员就是设备号。而这里的 inode 是存在于内存

中的节点，是涉及文件操作的一个非常关键的数据结构，关于该结构我们之后还要讨论，这里只需要知道写入磁盘中的 ext2_inode 结构内的成员基本上都是靠存在于内存中的 inode 中对应的成员初始化的即可，其中就包含了这里讲的设备号。之前我们说过，设备号分主、次设备号两个，而这里的设备号只有一个。原因是主、次设备号的位宽有限制，可以将两个号合并在一起，之后我们会看到相应的代码。从代码 1456 行我们还可以看到，文件的类型也被保存在了 ext2_inode 结构中，并且写在了磁盘上。

刚才还谈到了需要在文件所在目录下添加目录项，那么这又是怎么完成的呢?在 Linux 系统中，目录本身也是一个文件，其中保存的数据是若干个目录项，目录项的主要内容就是刚才分配的节点号和文件或子目录的名字。ext2 文件系统中，写入磁盘的目录项数据结构如下。

```
    /* fs/ext2/ext2.h */

    574 /*
    575  * Structure of a directory entry
    576  */
    577
    578 struct ext2_dir_entry {
    579         __le32  inode;                  /* Inode number */
    580         __le16  rec_len;                /* Directory entry length */
    581         __le16  name_len;               /* Name length */
    582         char    name[];                 /* File name, up to EXT2_NAME_LEN *
/
    583 };
```

上面的 inode 成员就是节点号，name 成员就是文件或子目录的名字。具体的代码实现可以参考“fs/ext2/namei.c”的 ext2_mknod 函数，在此不再赘述。可以通过图 3.1 来说明 mknod 命令在 ext2 文件系统的上所完成的工作。

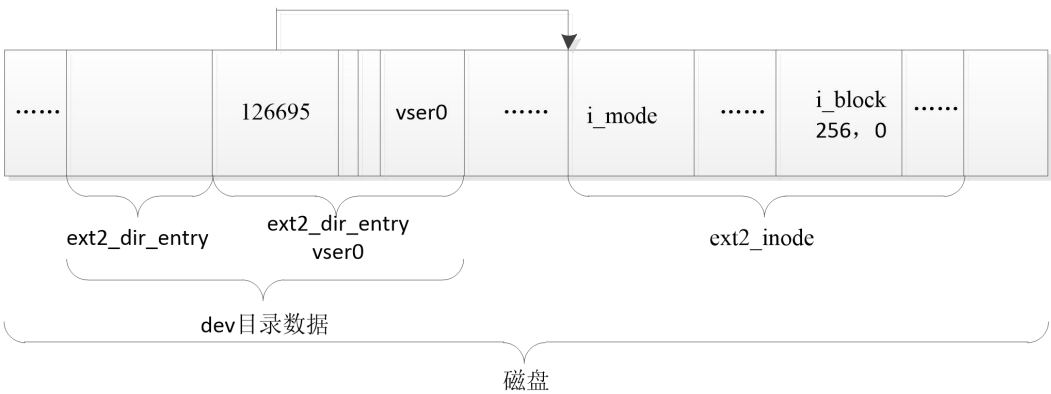

图 3.1 创建设备文件示意图

上面的整个过程，一言蔽之就是 mknod 命令将文件名、文件类型和主、次设备号等信息保存在了磁盘上。

接下来我们来讨论如何打开一个文件，这是理解上层应用程序和底层的驱动程序如何建立联系的关键，也是理解字符设备驱动编写方式的关键。整个过程非常繁琐，涉及到的数据结构和相关的内核知识非常的多。为了便于大家理解，下面将该过程进行大量简化，并以图 3.2 和调用流程来进行说明。

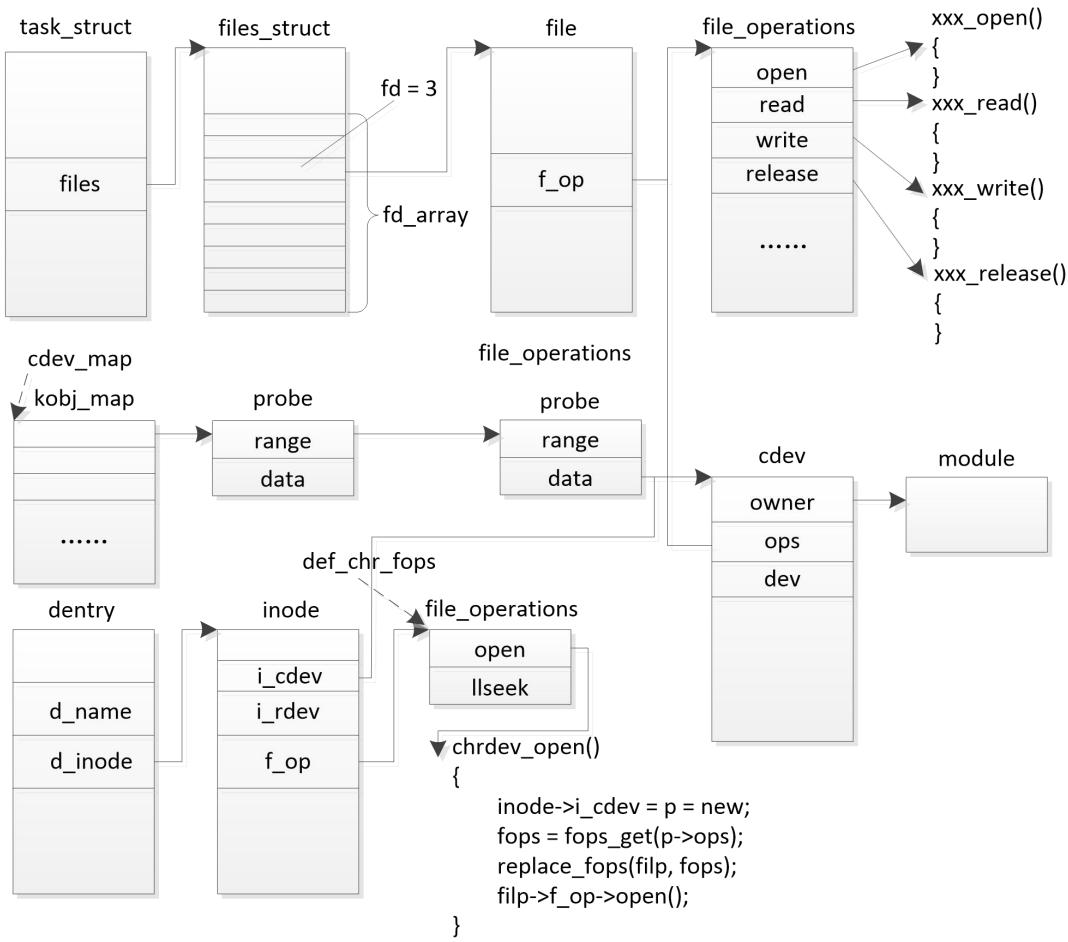

图 3.2 文件操作的相关数据结构及关系

```
sys_open(fs/open.c)
    |-do_sys_open(fs/open.c)
        |-getname(fs/namei.c)
        |-get_unused_fd_flags(fs/file.c)
        |-do_filp_open(fs/namei.c)
        |   |-path_openat(fs/namei.c)
        |       |-get_empty_filp(fs/file_table.c)
        |       |-link_path_walk(fs/namei.c)
        |       |-do_last(fs/namei.c)
```

```
|           |-lookup_fast(fs/namei.c)
|           |-lookup_open(fs/namei.c)
|               |-lookup_dcache(fs/namei.c)
|               |-lookup_real(fs/namei.c)
|                   |-ext2_lookup(fs/ext2/namei.c)
|                       |-ext2_iget(fs/ext2/inode.c)
|                           |-init_special_inode(fs/inode.c)
|                               |-inode->i_fop = &def_chr_fops;
|-fd_install(fs/namei.c)
```

一个进程在内核中用一个 task_struct 结构对象来表示，其中的 files 成员指向了一个 files_struct 结构变量，该结构中有一个 fd_array 的指针数组（用于维护打开文件的信息），数组的每一个元素是指向 file 结构的一个指针。open 系统调用函数在内核中对应的函数是 sys_open，sys_open 调用了 do_sys_open，在 do_sys_open 中首先调用了 getname 函数将文件名从用户空间拷贝到了内核空间。接着调用 get_unused_fd_flags 来获取一个未使用的文件描述符，要获得该描述符，其实就是搜索 files_struct 中的 fd_array 数组，查看哪一个元素没有被使用，然后返回其下标即可。接下来调用 do_filp_open 函数来构造一个 file 结构，并初始化里面的成员。其中最重要的是将他的 f_op 成员指向和设备对应的驱动程序的操作方法集合的结构 file_operations，这个结构中的绝大多数成员都是函数指针，通过 file_operations 中的 open 函数指针可以调用驱动中实现的特定于设备的打开函数，从而完成打开的操作。do_filp_open 函数执行成功后，最后调用 fd_install 函数，该函数将刚才得到的文件描述符作为访问 fd_array 数组的下标，让下标对应的元素指向新构造的 file 结构。最后系统调用返回到应用层，将刚才的数组下标作为打开文件的文件描述符返回。

do_filp_open 函数的内容很多，是这个过程中最复杂的一部分，下面进行一个非常简化的介绍。do_filp_open 函数调用 path_openat 来进行实际的打开操作，path_openat 调用 get_empty_filp 来快速得到一个 file 结构，接下来 path_openat 调用 link_path_walk 来处理文件路径中除最后一个分量的前面部分。举个例子来说，如果要打开/dev/vser0 这个文件，那么 link_path_walk 需要处理/dev 这部分，包含根目录和 dev 目录。path_openat 接下来调用 do_last 来处理最后一个分量，而 do_last 首先调用 lookup_fast 在 RCU 模式下来尝试快速查找，如果第一次这么做，会失败，所以又继续调用 lookup_open，而 lookup_open 首先调用 lookup_dcache 在目录项高速缓存中进行查找，第一次也会失败，所以转而调用 lookup_real，lookup_real 则开始在磁盘上真正开始查找最后一个分量所对应的节点，如果是 ext2 文件系统，则会调用 ext2_lookup，得到 inode 的编号后，ext2_lookup 又会调用 ext2_iget 来从磁盘上获取之前使用 mknod 保存的节点信息。对字符设备驱动来说，这里最重要的就是将文件类型和设备号取出并填充到了内存中的 inode 结构的相关成员中。另外，通过判断文件的类型，还将 inode 中的 f_op 指针指向了 def_chr_fops，这个结构中 open 函数指针指向了 chrdev_open，那么自然 chrdev_open 紧接着会被调用。chrdev_open 完成的主要工作是：首先根据设备号找到添加在内核中代表字符设备的 cdev（cdev 是放在 cdev_map 散列表中的，驱动加载时会构造相应的 cdev 并添加到这个散列表中，并且在构造这个 cdev 时还实现了一个操作方法集合，由 cdev 的 ops 成员指向他），找到对应的 cdev 对象后，用 cdev 关联的操作方法集合替代之前构造的

file 结构中的操作方法集合，然后调用 cdev 所关联的操作方法集合中的打开函数，完成设备真正的打开操作，这也标志着 do_filp_open 函数基本也结束。

为了下一次能够快速打开文件，内核在第一次打开一个文件或目录时都会创建一个 dentry 的目录项，他保存了文件名和所对应的 inode 信息，所有的这些 dentry 使用散列的方式存储在目录项高速缓存中，内核在打开文件时会首先在这个高速缓存中找相应的 dentry，如果找到，则可以立即获取文件所对应的 inode，否则就会像前面那样在磁盘上获取。

对于字符设备驱动来说，设备号、cdev 和操作方法集合至关重要，内核在找到路径名所对应的 inode 后，要和驱动建立连接，首要要做的就是根据 inode 中的设备号找到 cdev，然后再根据 cdev 找到关联的操作方法集合，从而调用驱动所提供的操作方法来完成对设备的具体操作。可以说，字符设备驱动的框架就是围绕着设备号、cdev 和操作方法集合来实现的。

虽然设备的打开操作很繁琐，但是其他的系统调用过程就要简单得多。因为打开操作返回了一个文件描述符，其他系统调用都会以这个文件描述符作为参数传递给内核，内核得到这个文件描述符后可以直接索引 fd_array，找到对应的 file 结构，然后调用相应的方法。

3.2 字符设备驱动框架

通过上一节的分析我们知道，要实现一个字符设备驱动，其中最重要的事就是要构造一个 cdev 结构对象，并让 cdev 同设备号和设备的操作方法集合相关联，然后将该 cdev 结构对象添加到内核的 cdev_map 散列表中。下面我们逐步来实现这一过程，首先就是在驱动中注册设备号，代码如下（完整的代码请参见“光盘资料/程序源码/chrdev/ex1”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 #include <linux/fs.h>
 6
 7 #define VSER_MAJOR      256
 8 #define VSER_MINOR      0
 9 #define VSER_DEV_CNT    1
10 #define VSER_DEV_NAME   "vser"
11
12 static int __init vser_init(void)
13 {
14         int ret;
15         dev_t dev;
16
17         dev = MKDEV(VSER_MAJOR, VSER_MINOR);
18         ret = register_chrdev_region(dev, VSER_DEV_CNT, VSER_DEV_NAME);
```

```
19        if (ret)
20                goto reg_err;
21        return 0;
22
23 reg_err:
24        return ret;
25 }
26
27 static void __exit vser_exit(void)
28 {
29
30        dev_t dev;
31
32        dev = MKDEV(VSER_MAJOR, VSER_MINOR);
33        unregister_chrdev_region(dev, VSER_DEV_CNT);
34 }
35
36 module_init(vser_init);
37 module_exit(vser_exit);
38
39 MODULE_LICENSE("GPL");
40 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
41 MODULE_DESCRIPTION("A simple character device driver");
42 MODULE_ALIAS("virtual-serial");
```

在模块的初始化函数中，首先在代码第 17 行使用 MKDEV 宏将主设备号和次设备号合并成一个设备号。在 3.14.25 版本的内核源码中，相关的宏定义如下。

```
 6 #define MINORBITS      20
 7 #define MINORMASK      ((1U << MINORBITS) - 1)
 8
 9 #define MAJOR(dev)     ((unsigned int) ((dev) >> MINORBITS))
10 #define MINOR(dev)     ((unsigned int) ((dev) & MINORMASK))
11 #define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

不难发现，该宏的作用是将主设备号左移 20 位和次设备号相或。在当前的内核版本中，dev_t 是一个无符号的 32 为整数，那么很自然的，主设备号占 12 位，次设备号占 20 位。另外还有两个宏为 MAJOR 和 MINOR，他们分别是从设备号中取出主设备号和次设备号的两个宏。尽管我们知道设备号是怎样构成的，但是我们在代码中不应该自己来构造设备号，而是应该调用相应的宏，因为不能保证以后的内核会改变这一规则。

构造好设备号之后，代码第 18 行调用 register_chrdev_region 将构造的设备号注册到内核

中，表明该设备号已经被占用，如果有其他驱动随后要注册该设备号，将会失败。其函数原型如下。

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
```

该函数一次性可以注册多个连续的号，由 count 形参指定个数，由 from 指定起始的设备号，name 用于标记主设备号的名称。该函数成功返回 0，不成功返回负数，通常是在设备号已经被其他的驱动抢先注册的情况。如果注册出错，则使用 goto 语句跳转到错误处理代码处执行，否则初始化函数返回 0。使用 goto 函数进行集中错误处理在驱动中非常常见，也非常实用，虽然这和一般的 C 语言编程规则相悖。

在模块卸载时，已注册的号应该从内核中注销，否则再次加载该驱动时，注册设备号操作会失败。代码第 33 行调用了 unregister_chrdev_region 函数，该函数只有两个形参，和 register_chrdev_region 函数的前两个形参的意义一样。

上面的代码可以再一次印证在模块章节中所说的内容，即模块初始化函数中负责注册、分配内存等操作，而在模块清除函数中负责相反的操作，即注销、释放内存等操作。以上的代码可以编译并进行测试，在 Ubuntu 主机上测试的步骤如下（在 ARM 目标板上的测试和前面模块在 ARM 目标板上测试的过程类似）。

```
# make
# make modules_install
# depmod
# modprobe vser
# cat /proc/devices
Character devices:
……
256 vser
……
# rmmod vser
# cat /proc/devices | grep vser
```

使用 cat 命令查看/proc/devices 会发现 vser 一项，并且主设备号为 256，说明设备号注册成功。使用 register_chrdev_region 注册设备号的方式称为静态注册设备号的方式，但是该方式有一个明显的缺点，那就是如果两个驱动都使用了同样的设备号，那么后加载的驱动将会失败，因为设备号冲突了。为了解决这个问题，可以使用动态分配设备号的函数，其原型如下。

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,con
st char *name);
```

其中 count 和 name 形参同 register_chrdev_region 函数中相应的形参一致。baseminor 是动态分配的设备号的起始次设备号，而 dev 则是分配得到的第一个设备号。该函数成功返回 0，失败返回负数。这样就避免了各个驱动使用相同的设备号而带来的冲突，但是也会存在另外一个问题，那就是不能事先知道主次设备号是多少，在使用 mknod 命令创建设备节点时，

必须先要查看/proc/devices 文件才能确定主设备号（次设备号在代码中确定），也就是要求 mknod 命令要后于驱动加载执行，不过这个问题在新的 Linux 设备模型中已经得到了比较好的解决，那就是设备节点会自动的创建和销毁，这在后面的章节会详细描述。

成功地注册了设备号，接下来应该构造并添加 cdev 结构对象了，其代码如下（完整的代码请参见“光盘资料/程序源码/chrdev/ex2”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 #include <linux/fs.h>
 6 #include <linux/cdev.h>
 7
 8 #define VSER_MAJOR     256
 9 #define VSER_MINOR     0
10 #define VSER_DEV_CNT    1
11 #define VSER_DEV_NAME   "vser"
12
13 static struct cdev vsdev;
14
15 static struct file_operations vser_ops = {
16         .owner = THIS_MODULE,
17 };
18
19 static int __init vser_init(void)
20 {
21         int ret;
22         dev_t dev;
23
24         dev = MKDEV(VSER_MAJOR, VSER_MINOR);
25         ret = register_chrdev_region(dev, VSER_DEV_CNT, VSER_DEV_NAME);
26         if (ret)
27                 goto reg_err;
28
29         cdev_init(&vsdev, &vser_ops);
30         vsdev.owner = THIS_MODULE;
31
32         ret = cdev_add(&vsdev, dev, VSER_DEV_CNT);
33         if (ret)
34                 goto add_err;
```

```
35
36          return 0;
37
38  add_err:
39          unregister_chrdev_region(dev, VSER_DEV_CNT);
40  reg_err:
41          return ret;
42  }
43
44  static void __exit vser_exit(void)
45  {
46
47          dev_t dev;
48
49          dev = MKDEV(VSER_MAJOR, VSER_MINOR);
50
51          cdev_del(&vsdev);
52          unregister_chrdev_region(dev, VSER_DEV_CNT);
53  }
54
55  module_init(vser_init);
56  module_exit(vser_exit);
57
58  MODULE_LICENSE("GPL");
59  MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
60  MODULE_DESCRIPTION("A simple character device driver");
61  MODULE_ALIAS("virtual-serial");
```

相比于之前的代码，第 13 行定义了一个 struct cdev 类型的全局变量 vsdev，在第 15 行到第 17 行定义了一个 struct file_operations 类型的全局变量 vser_ops。我们知道，这两个数据结构是实现字符设备驱动的关键。其中 vsdev 代表了一个具体的字符设备，而 vser_ops 是操作该设备的一些方法。代码第 29 行调用了 cdev_init 函数初始化了 vsdev 里面的部分成员，另外一个最重要的操作就是将 vsdev 里面的 ops 指针指向了 vser_ops，这样在通过设备号找到 vsdev 对象后，就能找到相关的操作方法集合，并调用其中的方法。cdev_init 函数的原型如下，第一个参数是要初始化的 cdev 地址，第二个参数是设备操作方法集合的结构地址。

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

代码第 16 行和代码第 30 行都将一个 owner 成员赋值为 THIS_MUDULE，owner 是一个指向 struct module 类型变量的指针，THIS_MUDULE 是包含驱动的模块中的 struct module 类型对象的地址，类似于 c++中的 this 指针。这样就能通过 vsdev 或 vser_fops 找到对应的模块，

在对前面两个对象进行访问时都要调用类似于 try_module_get 的函数增加模块的引用计数，其目的是这两个对象在使用的过程中，模块是不能被卸载的，因为模块被卸载的前提条件是引用计数为 0。

cdev 对象初始化好以后，就应该添加到内核中的 cdev_map 散列表当中，调用的函数是 cdev_add，其函数原型如下。

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

根据前面的几个函数原型，不难得出该函数的各个形参的意义。cdev_add 函数的主要工作是将主设备号通过对 255 取余，将余数作为 cdev_map 数组的下标索引，然后构造一个 probe 对象，并让 data 指向要添加的 cdev 结构地址，然后加入到链表当中（参见图 3.2）。该函数的最后一个参数 count 指定了被添加的 cdev 可以管理多少个设备。这里需要特别注意的是参数 p 只指向一个 cdev 对象，但该对象可以同时管理多个设备，由 count 的值来决定具体有多少个设备，那么 cdev 和设备就并不是一一对应的关系。这样一来，对于一个驱动支持多个设备的情况，我们可以采用两种方法来实现，第一种方法就是为每一个设备分配一个 cdev 对象，每次调用 cdev_add 添加一个 cdev 对象，直到多个 cdev 对象全部被添加到内核中；第二种方法就是只构造一个 cdev 对象，但在调用 cdev_add 时，指定添加的 cdev 可以管理多个设备。这两种方法在稍后的例子中我们都会看到。以上是稍微简化的讨论，实际的实现要比上面的讨论还要复杂一些，如果要详细了解，请参考 cdev_add 的内核源码。

在初始化函数中添加了 cdev 对象，那么在清除函数中自然就应该删除该 cdev 对象，代码第 51 行演示了这一操作，实现的函数是 cdev_del，其函数原型如下。很自然地会想到该函数的作用就是根据 cdev 找到散列表中的 probe，并进行删除。

```
void cdev_del(struct cdev *p);
```

在上述的示例代码中，cdev 对象是静态定义的，我们也可以进行动态分配，对应的函数是 cdev_alloc，其函数原型如下。

```
struct cdev *cdev_alloc(void);
```

该函数成功返回动态分配的 cdev 对象地址，失败返回 NULL。

上面的代码基本上实现了一个字符设备驱动程序的框架，即使目前还没有任何的实际意义，但是还是能够对之进行操作了，其相关的命令如下。

```
# mknod /dev/vser0 c 256 0
# cat /dev/vser0
cat: /dev/vser0: No such device or address
# make
# make modules_install
# depmod
# modprobe vser
# cat /dev/vser0
cat: /dev/vser0: Invalid argument
```

从上面的操作可以看到，在未加载驱动之前，使用 cat 命令读取/dev/vser0 设备，错误信

息是设备找不到，这是因为找不到和设备号对应的 cdev 对象。在驱动加载后，cat 命令的错误信息变成了参数无效，说明驱动工作了，只是现在还未实现具体的设备操作的方法。

3.3 虚拟串口设备

在进一步实现字符设备驱动之前，我们先来讨论一下本书中用到的一个虚拟串口设备。这个设备是驱动代码虚拟出来的，不能实现真正的串口数据收发，但是他能够接收来自用户想要发送的数据，并且将该数据原封不动地环回给串口的收端，从而用户也能从该串口接收数据。也就是说，该虚拟串口设备是一个功能弱化之后的只具备内环回作用的串口，示意图如图 3.3 所示。

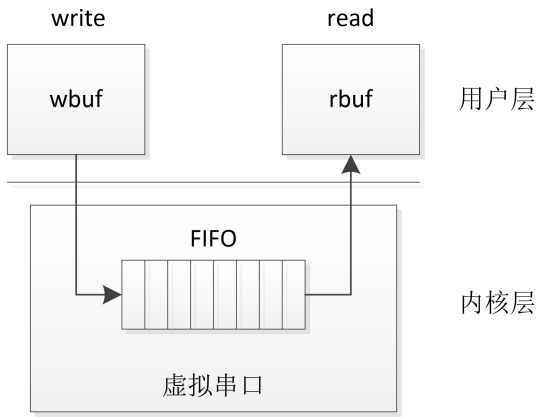

图 3.3 虚拟串口设备

这一功能的实现，主要是在驱动中实现一个 FIFO，驱动接收来自用户层传来的数据，然后将之放入 FIFO，当应用层要获取数据时，驱动将 FIFO 中的数据读出，然后拷贝给应用层。一个更贴近于实际的形式应该是在驱动中有两个 FIFO，一个用于发送，一个用于接收，但是这并不是实现这个简单的虚拟串口设备驱动的关键，所以为了简单起见，这里只用了一个 FIFO。

内核中已经有了一个关于 FIFO 的数据结构 struct kfifo，相关的操作宏或函数的声明或定义都在在“include/linux/kfifo.h”头文件中，下面将最常用的宏罗列如下。

```
DEFINE_KFIFO(fifo, type, size)
kfifo_from_user(fifo, from, len, copied)
kfifo_to_user(fifo, to, len, copied)
```

DEFINE_KFIFO 用于定义并初始化一个 FIFO，这个变量的名字由 fifo 参数决定，type 是 FIFO 中成员的类型，size 则指定这个 FIFO 有多少个元素，但是 size 必须是 2 的幂。kfifo_from_user 是将用户空间的数据（from）放入 FIFO 中，元素个数由 len 来指定，实际放入的元素个数由 copied 返回。kfifo_to_user 则是将 FIFO 中的数据取出，拷贝到用户空间（to）。len 和 copied 的含义同 kfifo_from_user 中对应的参数。

3.4 虚拟串口设备驱动

字符设备驱动除了前面搭建好代码的框架外，接下来最重要的就是要实现特定于设备的操作方法，这是驱动的核心和关键所在，是每个驱动区别于其他驱动的最本质所在，也是整个驱动代码中最灵活的代码所在。在前面了解了虚拟串口设备的工作方式后，接下来就可以针对该设备编写一个驱动程序，其完整的代码如下（完整的代码请参见“光盘资料/程序源码/chrdev/ex3”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 #include <linux/fs.h>
 6 #include <linux/cdev.h>
 7 #include <linux/kfifo.h>
 8
 9 #define VSER_MAJOR     256
10 #define VSER_MINOR     0
11 #define VSER_DEV_CNT    1
12 #define VSER_DEV_NAME   "vser"
13
14 static struct cdev vsdev;
15 DEFINE_KFIFO(vsfifo, char, 32);
16
17 static int vser_open(struct inode *inode, struct file *filp)
18 {
19         return 0;
20 }
21
22 static int vser_release(struct inode *inode, struct file *filp)
23 {
24         return 0;
25 }
26
27 static ssize_t vser_read(struct file *filp, char __user *buf, size_t co
unt, loff_t *pos)
28 {
29         unsigned int copied = 0;
30
31         kfifo_to_user(&vsfifo, buf, count, &copied);
```

```
32
33          return copied;
34  }
35
36  static ssize_t vser_write(struct file *filp, const char __user *buf, si
ze_t count, loff_t *pos)
37  {
38          unsigned int copied = 0;
39
40          kfifo_from_user(&vsfifo, buf, count, &copied);
41
42          return copied;
43  }
44
45  static struct file_operations vser_ops = {
46          .owner = THIS_MODULE,
47          .open = vser_open,
48          .release = vser_release,
49          .read = vser_read,
50          .write = vser_write,
51  };
52
53  static int __init vser_init(void)
54  {
55          int ret;
56          dev_t dev;
57
58          dev = MKDEV(VSER_MAJOR, VSER_MINOR);
59          ret = register_chrdev_region(dev, VSER_DEV_CNT, VSER_DEV_NAME);
60          if (ret)
61                  goto reg_err;
62
63          cdev_init(&vsdev, &vser_ops);
64          vsdev.owner = THIS_MODULE;
65
66          ret = cdev_add(&vsdev, dev, VSER_DEV_CNT);
67          if (ret)
68                  goto add_err;
69
```

```
70          return 0;
71
72  add_err:
73          unregister_chrdev_region(dev, VSER_DEV_CNT);
74  reg_err:
75          return ret;
76  }
77
78  static void __exit vser_exit(void)
79  {
80
81          dev_t dev;
82
83          dev = MKDEV(VSER_MAJOR, VSER_MINOR);
84
85          cdev_del(&vsdev);
86          unregister_chrdev_region(dev, VSER_DEV_CNT);
87  }
88
89  module_init(vser_init);
90  module_exit(vser_exit);
91
92  MODULE_LICENSE("GPL");
93  MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
94  MODULE_DESCRIPTION("A simple character device driver");
95  MODULE_ALIAS("virtual-serial");
```

相比于上一个示例代码，新的驱动在代码第 15 行定义并初始化了一个名叫 vsfifo 的 struct kfifo 对象，每个元素的数据类型为 char，总共有 32 个元素的空间。代码第 17 行到第 25 行实现了设备的打开和关闭函数，分别对应于 file_operations 内的 open 和 release 方法。因为是虚拟设备，所以在这里并没有需要特别处理的操作，仅仅返回 0 表示成功。这两个函数都有两个相同的形参，其中第一个形参是要打开或关闭文件的 inode，第二个形参则是打开对应文件后由内核构造并初始化好的 file 结构，在前面的章节中我们已经较深入分析了这两个对象的作用。在这里之所以叫 release 而不叫 close 是因为一个文件可以被打开多次，那么 vser_open 函数相应地会被调用多次，但是关闭文件只有直到最后一个 close 操作才会导致 vser_release 函数被调用，所以用 release 更贴切。

代码第 27 行到 34 行是 read 系统调用的驱动实现，按照上一节的描述，这里主要是要把 FIFO 中的数据返回给用户层，使用了 kfifo_to_user 这个宏。read 系统调用要求用户返回实际读取的字节数，而 copied 变量的值正好符合这一要求。代码第 36 行到第 43 行是对应的 write 系统调用的驱动实现，同 read 一样，只是数据流向相反而已。

读和写函数引入了 3 个新的形参，分别是 buf，count 和 pos，根据上面的代码，已经不难发现他们的含义，buf 代表的是用户空间的内存起始地址，count 表示用户想要读写多少个字节的数据，而 pos 是文件的位置指针，在虚拟串口这样一个不支持随机访问的设备中，该参数无用。__user 是提醒驱动代码编写者，这个内存空间属于用户空间。

代码第 47 行到代码第 50 行是将 file_operations 里面的函数指针分别指向上面定义的函数，这样在应用层发生相应的系统调用后，在驱动里面的函数就会被相应的调用。上面这个示例实现了一个功能非常简单，但是基本可用的一个虚拟串口驱动程序。按照下面的步骤可以进行验证。

```
# mknod /dev/vser0 c 256 0
# make
# make modules_install
# depmod
# modprobe vser
# echo "vser deriver test" > /dev/vser0
# cat /dev/vser0
vser deriver test
```

通过实验结果可以看到，对/dev/vser0 写入什么数据，就可以从这个设备读到什么数据，和一个具备内环回功能的串口是一致的。

最后，为了方便读者对照查阅，特将 file_operations 结构类型的定义代码粘贴如下。从中我们可以看到，还有很多接口函数还没有实现，在后面的章节中，我们会陆续再实现一些接口。显然，一个驱动对下面的接口的实现越多，那么他对用户提供的功能就越多，但这也不是说我们必须要实现下面的所有函数接口。比如对于串口来讲，不支持随机访问，那么 llseek 函数接口自然就不用实现。

```
1525 struct file_operations {
1526         struct module *owner;
1527         loff_t (*llseek) (struct file *, loff_t, int);
1528         ssize_t (*read) (struct file *, char __user *, size_t, loff_t
*);
1529         ssize_t (*write) (struct file *, const char __user *, size_t, lo
ff_t *);
1530         ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsi
gned long, loff_t);
1531         ssize_t (*aio_write) (struct kiocb *, const struct iovec *, uns
igned long, loff_t);
1532         int (*iterate) (struct file *, struct dir_context *);
1533         unsigned int (*poll) (struct file *, struct poll_table_struct
*);
1534         long (*unlocked_ioctl) (struct file *, unsigned int, unsigned l
```

```
ong);
    1535        long (*compat_ioctl) (struct file *, unsigned int, unsigned lon
g);
    1536        int (*mmap) (struct file *, struct vm_area_struct *);
    1537        int (*open) (struct inode *, struct file *);
    1538        int (*flush) (struct file *, fl_owner_t id);
    1539        int (*release) (struct inode *, struct file *);
    1540        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    1541        int (*aio_fsync) (struct kiocb *, int datasync);
    1542        int (*fasync) (int, struct file *, int);
    1543        int (*lock) (struct file *, int, struct file_lock *);
    1544        ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
    1545        unsigned long (*get_unmapped_area)(struct file *, unsigned long,
 unsigned long, unsigned long, unsigned long);
    1546        int (*check_flags)(int);
    1547        int (*flock) (struct file *, int, struct file_lock *);
    1548        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
 loff_t *, size_t, unsigned int);
    1549        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_ino
de_info *, size_t, unsigned int);
    1550        int (*setlease)(struct file *, long, struct file_lock **);
    1551        long (*fallocate)(struct file *file, int mode, loff_t offset,
    1552                          loff_t len);
    1553        int (*show_fdinfo)(struct seq_file *m, struct file *f);
    1554 };
```

3.5 一个驱动支持多个设备

如果一类设备有多个个体（比如系统上有 2 个串口），那么很自然的是我们应该写一个驱动来支持这几个设备，而不是每一个设备都写一个驱动。对于多个设备所引入的变化是什么呢?首先我们应向内核注册多个设备号，其次就是要在添加 cdev 对象时指明该 cdev 对象管理了多个设备，或者添加多个 cdev 对象，每个 cdev 对象管理一个设备。接下来最麻烦的部分在于读写操作，因为设备是多个，那么设备对应的资源也应该是多个（比如虚拟串口驱动中的 FIFO），那么在读写操作时，怎么来区分究竟应该对那个设备进行操作呢（对于虚拟串口驱动而言，就是要确定对哪个 FIFO 进行操作），观察读和写函数，没有发现能够区别设备的形参。不过再观察 open 接口，我们会发现有一个 inode 形参，通过前面的内容我们知道，inode 里面包含了所对应设备的设备号以及所对应的 cdev 对象的地址，那么我们可以在 open

接口函数中取出这些信息，并存放在 file 结构对象的某个成员中，而在读写的接口函数中，获取该 file 结构的成员，从而可以区分出对哪个设备进行操作。

下面首先展示用一个 cdev 实现对多个设备的支持（完整的代码请参见“光盘资料/程序源码/chrdev/ex4”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 #include <linux/fs.h>
 6 #include <linux/cdev.h>
 7 #include <linux/kfifo.h>
 8
 9 #define VSER_MAJOR      256
10 #define VSER_MINOR      0
11 #define VSER_DEV_CNT    2
12 #define VSER_DEV_NAME   "vser"
13
14 static struct cdev vsdev;
15 static DEFINE_KFIFO(vsfifo0, char, 32);
16 static DEFINE_KFIFO(vsfifo1, char, 32);
17
18 static int vser_open(struct inode *inode, struct file *filp)
19 {
20         switch (MINOR(inode->i_rdev)) {
21         default:
22         case 0:
23                 filp->private_data = &vsfifo0;
24                 break;
25         case 1:
26                 filp->private_data = &vsfifo1;
27                 break;
28         }
29         return 0;
30 }
31
32 static int vser_release(struct inode *inode, struct file *filp)
33 {
34         return 0;
35 }
```

```
36
37 static ssize_t vser_read(struct file *filp, char __user *buf, size_t co
unt, loff_t *pos)
38 {
39         unsigned int copied = 0;
40         struct kfifo *vsfifo = filp->private_data;
41
42         kfifo_to_user(vsfifo, buf, count, &copied);
43
44         return copied;
45 }
46
47 static ssize_t vser_write(struct file *filp, const char __user *buf, si
ze_t count, loff_t *pos)
48 {
49         unsigned int copied = 0;
50         struct kfifo *vsfifo = filp->private_data;
51
52         kfifo_from_user(vsfifo, buf, count, &copied);
53
54         return copied;
55 }
56
57 static struct file_operations vser_ops = {
58         .owner = THIS_MODULE,
59         .open = vser_open,
60         .release = vser_release,
61         .read = vser_read,
62         .write = vser_write,
63 };
64
65 static int __init vser_init(void)
66 {
67         int ret;
68         dev_t dev;
69
70         dev = MKDEV(VSER_MAJOR, VSER_MINOR);
71         ret = register_chrdev_region(dev, VSER_DEV_CNT, VSER_DEV_NAME);
72         if (ret)
```

```
 73                goto reg_err;
 74
 75        cdev_init(&vsdev, &vser_ops);
 76        vsdev.owner = THIS_MODULE;
 77
 78        ret = cdev_add(&vsdev, dev, VSER_DEV_CNT);
 79        if (ret)
 80                goto add_err;
 81
 82        return 0;
 83
 84 add_err:
 85        unregister_chrdev_region(dev, VSER_DEV_CNT);
 86 reg_err:
 87        return ret;
 88 }
 89
 90 static void __exit vser_exit(void)
 91 {
 92
 93        dev_t dev;
 94
 95        dev = MKDEV(VSER_MAJOR, VSER_MINOR);
 96
 97        cdev_del(&vsdev);
 98        unregister_chrdev_region(dev, VSER_DEV_CNT);
 99 }
100
101 module_init(vser_init);
102 module_exit(vser_exit);
103
104 MODULE_LICENSE("GPL");
105 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
106 MODULE_DESCRIPTION("A simple character device driver");
107 MODULE_ALIAS("virtual-serial");
```

上面的代码针对前一示例做的修改是：将 VSER_DEV_CNT 宏定义为 2，表示支持两个设备；用 DEFINE_KFIFO 定义了两个 FIFO，分别是 vsfifo0 和 vsfifo1（很显然，这里动态分配 FIFO 要优于静态定义，但是这会涉及到后面章节中的内核内存分配的相关知识，故此使用静态的方法）；在 open 接口函数中根据次设备号的值来确定保存哪个 FIFO 结构的地址到

file 结构中的 private_data 成员中，file 结构中的 private_data 是一个 void *类型的指针，内核保证不会使用该指针，所以正如其名一样，是驱动私有的；在读写接口函数中则是从 file 结构中取出了 private_data 的值，即 FIFO 结构的地址，然后再进行进一步的操作。

接下来演示如何将每一个 cdev 对象对应到一个设备来实现一个驱动对多个设备的支持。完整的示例代码如下（完整的代码请参见“光盘资料/程序源码/chrdev/ex5”）。

```
 1 #include <linux/init.h>
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4
 5 #include <linux/fs.h>
 6 #include <linux/cdev.h>
 7 #include <linux/kfifo.h>
 8
 9 #define VSER_MAJOR     256
10 #define VSER_MINOR     0
11 #define VSER_DEV_CNT    2
12 #define VSER_DEV_NAME   "vser"
13
14 static DEFINE_KFIFO(vsfifo0, char, 32);
15 static DEFINE_KFIFO(vsfifo1, char, 32);
16
17 struct vser_dev {
18         struct kfifo *fifo;
19         struct cdev cdev;
20 };
21
22 static struct vser_dev vsdev[2];
23
24 static int vser_open(struct inode *inode, struct file *filp)
25 {
26         filp->private_data = container_of(inode->i_cdev, struct vser_dev,
cdev);
27         return 0;
28 }
29
30 static int vser_release(struct inode *inode, struct file *filp)
31 {
32         return 0;
33 }
```

```
34
35 static ssize_t vser_read(struct file *filp, char __user *buf, size_t co
unt, loff_t *pos)
36 {
37         unsigned int copied = 0;
38         struct vser_dev *dev = filp->private_data;
39
40         kfifo_to_user(dev->fifo, buf, count, &copied);
41
42         return copied;
43 }
44
45 static ssize_t vser_write(struct file *filp, const char __user *buf, si
ze_t count, loff_t *pos)
46 {
47         unsigned int copied = 0;
48         struct vser_dev *dev = filp->private_data;
49
50         kfifo_from_user(dev->fifo, buf, count, &copied);
51
52         return copied;
53 }
54
55 static struct file_operations vser_ops = {
56         .owner = THIS_MODULE,
57         .open = vser_open,
58         .release = vser_release,
59         .read = vser_read,
60         .write = vser_write,
61 };
62
63 static int __init vser_init(void)
64 {
65         int i;
66         int ret;
67         dev_t dev;
68
69         dev = MKDEV(VSER_MAJOR, VSER_MINOR);
70         ret = register_chrdev_region(dev, VSER_DEV_CNT, VSER_DEV_NAME);
```

```
     71        if (ret)
     72                goto reg_err;
     73
     74        for (i = 0; i < VSER_DEV_CNT; i++) {
     75                cdev_init(&vsdev[i].cdev, &vser_ops);
     76                vsdev[i].cdev.owner = THIS_MODULE;
     77                vsdev[i].fifo = i == 0 ? (struct kfifo *) &vsfifo0 : (stru
ct kfifo*)&vsfifo1;
     78
     79                ret = cdev_add(&vsdev[i].cdev, dev + i, 1);
     80                if (ret)
     81                        goto add_err;
     82        }
     83
     84        return 0;
     85
     86 add_err:
     87        for (--i; i > 0; --i)
     88                cdev_del(&vsdev[i].cdev);
     89        unregister_chrdev_region(dev, VSER_DEV_CNT);
     90 reg_err:
     91        return ret;
     92 }
     93
     94 static void __exit vser_exit(void)
     95 {
     96        int i;
     97        dev_t dev;
     98
     99        dev = MKDEV(VSER_MAJOR, VSER_MINOR);
    100
    101        for (i = 0; i < VSER_DEV_CNT; i++)
    102                cdev_del(&vsdev[i].cdev);
    103        unregister_chrdev_region(dev, VSER_DEV_CNT);
    104 }
    105
    106 module_init(vser_init);
    107 module_exit(vser_exit);
    108
```

```
109 MODULE_LICENSE("GPL");
110 MODULE_AUTHOR("Kevin Jiang <jiangxg@farsight.com.cn>");
111 MODULE_DESCRIPTION("A simple character device driver");
112 MODULE_ALIAS("virtual-serial");
```

代码第 17 行至第 20 行新定义了一个结构类型 vser_dev，这个代表一种具体的设备类，通常和设备相关的内容都应该和 cdev 一起定义在一个结构中。如果用面向对象的思想来理解这种做法将会变得很容易。cdev 是所有字符设备的一个抽象，他是一个基类，而一个具体类型的设备应该是由该基类派生出来的一个子类，子类包含了特定设备所特有的属性，比如 vser_dev 中的 fifo，这样子类就更能刻画好一类具体的设备。代码第 22 行创建了两个 vser_dev 类型的对象，和 c++不同的是，创建这两个对象仅仅是为其分配了内存，并没有调用构造函数来初始化这两个对象，但在代码的第 74 行到第 77 行完成了这个操作。查看内核源码，会发现这种面向对象的思想处处可见，只能说是因为语言的特性，并没有把这种形式体现得很明显而已。代码的第 74 行到第 82 行通过 2 次循环完成了两个 cdev 对象的初始化和添加工作，并且初始化了 fifo 成员的指向。这里需要说明的是，用 DEFINE_KFIFO 定义的 FIFO，每定义一个 FIFO 就会新定义一种数据类型，所以严格的说 vsfifo0 和 vsfifo1 是两种不同类型的对象，但好在这里能和 struct kfifo 类型兼容。

代码第 26 行用到了一个 container_of 宏，这是在 Linux 内核中设计得非常巧妙的一个宏，在整个 Linux 内核源码中几乎随处可见。他的作用就是根据结构成员的地址来反向得到结构的起始地址。代码中，inode->i_cdev 给出了 struct vser_dev 结构类型中 cdev 成员的地址（参见图 3.2），通过 container_of 宏就得到了包含该 cdev 的结构地址。

上面两种方式都可以实现一个驱动对多个同类型设备的支持。使用下面的命令可以测试这两个驱动程序。

```
# mknod /dev/vser0 c 256 0
# mknod /dev/vser1 c 256 1
# make
# make modules_install
# depmod
# modprobe vser
# echo "xxxxx" > /dev/vser0
# echo "yyyyy" > /dev/vser1
# cat /dev/vser0
xxxxx
# cat /dev/vser1
yyyyy
```

3.6 习题

1、字符设备和块设备的区别不包括（ ）。
[A] 字符设备按字节流进行访问，块设备按块大小进行访问
[B] 字符设备只能处理可打印字符，块设备可以处理二进制数据
[C] 多数字符设备不能随机访问，而块设备一定能随机访问
[D] 字符设备通常没有页高速缓存，而块设备有

2、在 3.14.25 版本的内核中，主设备号占（ ）位，次设备号占（ ）位。
[A] 8
[B] 16
[C] 12
[D] 20

3、用于分配主次设备号的函数是（ ）。
[A] register_chrdev_region
[B] MKDEV
[C] alloc_chrdev_region
[D] MAJOR

4、字符设备驱动中 struct file_operations 结构中的函数指针成员不包含（ ）。
[A] open
[B] close
[C] read
[D] show_fdinfo