嵌入式Linux

驱动程序设计

从入门到精通

冯国进　编著

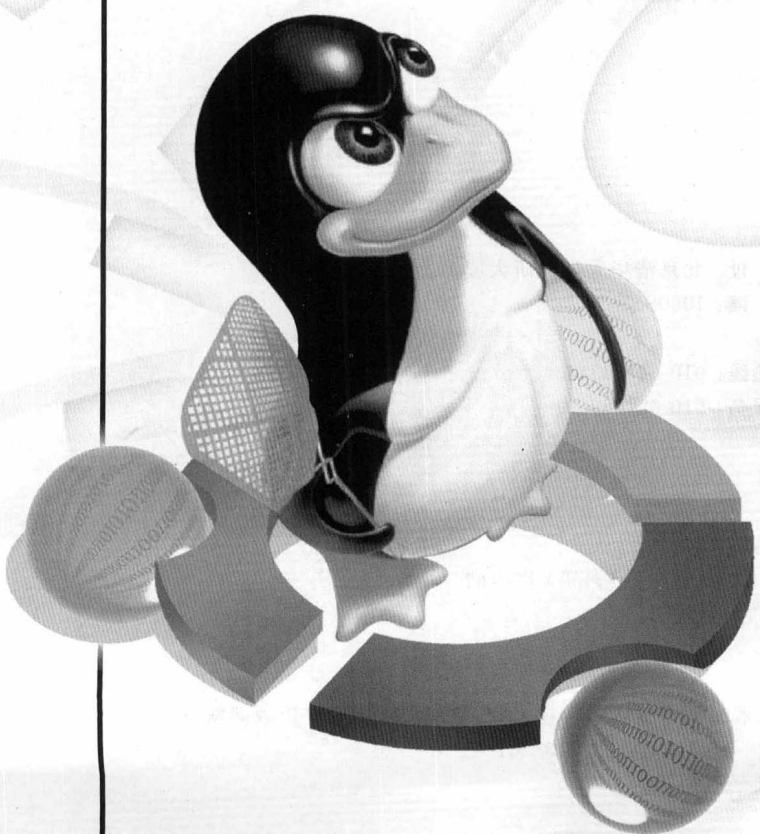
附光盘

- 各章驱动范例代码、内核代
- 本书所有芯片资料、标准文
- 驱动开发交叉编译环境

清华大学出版社

嵌入式Linux
驱动程序设计
从入门到精通

冯国进　编著



附光盘

- 各章驱动范例代码、内核代码
- 本书所有芯片资料、标准文档
- 驱动开发交叉编译环境

清华大学出版社
北京

内 容 简 介

本书基于 Linux 2.6 内核讲述了 Linux 嵌入式驱动程序开发的知识，全书内容涵盖了 Linux 2.6 下的三类驱动设备，包括 Linux 下字符设备、块设备、网络设备的开发技术。具体内容包括 Linux 驱动开发入门基础知识，Linux 操作系统下驱动开发核心技术，并对 ARM 系统的各类接口的原理、驱动开发与应用层开发进行逐一分析，其中包括 GPIO、CAN、I^2C、LCD、USB、触摸屏、网络、块设备、红外、SD 卡等接口。

本书主要面向嵌入式 Linux 系统的内核、驱动和应用程序的开发人员以及 ARM 嵌入式系统的接口设计人员，可以作为各类嵌入式系统培训机构和高校操作系统课程的实验教材和辅导书籍。

图书在版编目（CIP）数据

嵌入式 Linux 驱动程序设计从入门到精通 / 冯国进编著. —北京：清华大学出版社，2008.3
ISBN 978-7-302-16942-0

Ⅰ. 嵌…　Ⅱ. 冯…　Ⅲ. Linux 操作系统－程序设计　Ⅳ. TP316.89

中国版本图书馆 CIP 数据核字（2008）第 012807 号

责任编辑：夏兆彦
责任校对：张　剑
责任印制：王秀菊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　**邮　　编**：100084
c－service@tup.tsinghua.edu.cn
社 总 机：010-62770175　**邮购热线**：010-62786544
投稿咨询：010-62772015　**客户服务**：010-62776969
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：203×260　**印　张**：20.5　**字　数**：531 千字
版　　次：2008 年 3 月第 1 版　　**印　次**：2008 年 3 月第1 次印刷
印　　数：1～4000
定　　价：39.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　　产品编号：027613－01

FOREWORD

序言

随着现代智能设备的不断升级换代，基于 ARM 等嵌入式系统等的 32 位机智能系统在现代生活中的地位也越来越重要。我也经常上书店，期望寻觅一些比较贴近某一实际系统开发，更有参考价值与使用意义的书籍，可是发现除了部分针对 ARM9 系统开发板的简单介绍外，少有比较系统全面地介绍驱动开发的书籍，这一现实情况给科研工作者的实际工作带来不少困难。我周围就有不少科研人员从事该项技术研究，由于不太了解更深层次的技巧与方法，有时遇到问题不得不设法与芯片供应商的技术人员进行沟通，但问题往往还是不能得到及时圆满的解决。

冯国进和我提到正在编写一本嵌入式 linux 驱动开发方面的书，同时也向我介绍了此书的主要内容和组织结构，并嘱我为此书作序，我欣然同意。纵观本书，其内容涵盖了 linux2.6 下的三类驱动设备，包括 linux 下字符设备、块设备、网络设备的开发技术。本书全面地分析了嵌入式 linux 下驱动开发的核心技术，并深入探讨了 ARM 嵌入式系统各类接口的原理、驱动开发与应用层开发技术，其内容不禁深深吸引了我，相信该书将会对从事嵌入式系统研究的科研人员有极大的帮助。

冯国进在攻读硕士学位期间就对软件开发、图像处理算法和硬件设计显示出了浓厚的兴趣和独有的天赋，同时在科研学术上又非常执着与严谨，在此期间他所做的工作在我们学校科研工作中发挥了重要作用，至今我仍常向我现在的学生提起他，并希望他们能从他身上学到点什么。冯国进硕士毕业后一直致力于嵌入式系统的开发研究，应该说本书是其多年研究工作的经验总结和结晶，各个功能模块基本都是经他亲自调试验证过了的。

学习、研究过程的一个重要环节是总结，本书就是一个嵌入式系统开发研究人员的总结。我也希望更多的科研人员在学习此书、从事该类技术研究的过程中能多一些像这样的总结。

借此机会，向出版此书的清华大学出版社表示感谢，同时，也向承担此书编辑、整理、审订等浩繁工作的所有人员表示敬意。

顾国华
南京理工大学电子工程与光电技术学院　教授
2007-11-3

FOREWORD

前言

我学编程是从 VB 开始的，看的第一本编程书有《红楼梦》那么厚，现在想来仍然可怕。从那时开始我与软件开发便结下了不解之缘。后来我逐渐转向 VC++编程。三年前有幸接触到 Linux 操作系统开发，才真正领略到软件艺术的迷人风采，并真正喜欢上软件开发工作。兴趣是最好的老师。在工作中，我常常需要自己开发 Linux 下的驱动程序，但是国内市场上关于驱动开发的书籍寥若晨星，网络上的资料又不全面，不成系统。于是我萌生了写一本驱动开发方面的书籍的念头，希望能够稍微揭开一点点 Linux 驱动开发的神秘面纱，一方面是让更多的人少走弯路，为中国的软件业做一点微不足道的贡献；更重要的是抛砖引玉，希望以后有更多的同类书籍问世。

内核版本：

为什么选择 Linux，开源无疑是最大的诱惑。很多程序员喜欢 Linux，因为 Linux 让他们掌握了自己的命运。Linux 最初是在网络应用方面比较占优势，后来逐渐在嵌入式系统中占有一席之地，成为能与 Wince、Vxworks 抗衡的高端嵌入式操作系统之一，受众多大公司的欢迎，具有广阔的发展潜力。本书基于 Linux 2.6 内核。相对于 Linux 2.4 内核，Linux 2.6 内核做了相当大地改进，修改了驱动模型，增加了对更多设备的支持。

硬件平台：

随着人们生活水平的提高，人们对智能产品的需求越来越高。很多产品用 8 位机已经很难满足要求，我们已经迎来了 32 位应用的时代。目前在 32 位机市场上，基于 ARM 的嵌入式市场目前的出货量每年超过 15 亿，ARM 系统在业界拥有最多的第三方工具和解决方案供应商。ARM 公司自 1990 年正式成立以来，在 32 位 RISC（Reduced Instruction Set Computer）CPU 开发领域不断取得突破，其结构已经从 V3 发展到 V6。目前非常流行的 ARM 芯核有 ARM7TDMI，ARM720T，ARM9TDMI，ARM920T，ARM940T，ARM926EJ-S，ARM1020E 和 XScale 等。本书基于三星公司的 ARM920T 处理器 S3C2410X。

组织结构：

本书涵盖了 Linux 2.6 下的三类驱动设备，包括 Linux 下字符设备、块设备、网络设备的开发技术。第 1 章为 Linux 驱动开发入门基础知识。第 2 章讲述 Linux 操作系统下驱动开发核心技术。第 3～11 章对 ARM 系统的各类接口的原理、驱动开发与应用层开发进行逐一分析，其中包括 GPIO、CAN、I^2C、LCD、USB、触摸屏、网络、块设备、红外、SD 卡等接口。

读者对象：

本书是一本专门介绍嵌入式 Linux 驱动开发的书籍。本书主要面向嵌入式 Linux 系统的内核与驱动和应用程序的开发人员以及 ARM 嵌入式系统的接口设计人员，也可以作为各类嵌入式系统培训机构的培训实验教材和高校操作系统课程的辅导书籍。

光盘内容：

本书所附的光盘中包含本书各章代码、一些芯片资料和驱动开发文档。所有代码均在 YL2410 开发板上实验通过。

特别致谢：

特别感谢深圳优龙科技提供的 YL2410 开发板，他们的专业精神为嵌入式开发人员奉献了许多优秀的开发板。由于 Linux 下的驱动开发相当复杂，加之本人水平有限，错误在所难免，敬请各位读者谅解并指正，以便再版时修正。

冯国进

2007 年 10 月

CONTENTS

目录

第 1 章　Linux 驱动程序基础 …… 1
1.1　驱动程序的概念 …… 1
1.2　Linux 驱动程序模型 …… 1
1.3　最基本的调试手段 …… 5
1.4　导出符号的方法 …… 5
1.5　动态加载驱动程序 …… 6
1.6　在内核中加入新驱动 …… 6
1.7　应用程序操作接口 …… 7
1.8　第一个驱动 …… 10

第 2 章　Linux 驱动开发核心技术 …… 17
2.1　同步机制 …… 17
2.1.1　自旋锁 …… 17
2.1.2　信号量 …… 18
2.1.3　原子操作 …… 18
2.1.4　读写锁（rwlock） …… 19
2.1.5　seqlock 机制 …… 21
2.1.6　RCU …… 22
2.2　完成事件 …… 24
2.3　阻塞与非阻塞 …… 25
2.4　时间 …… 27
2.4.1　Linux 下延迟 …… 27
2.4.2　内核定时器 …… 27
2.5　内存分配与映射 …… 28
2.5.1　内存分配与释放 …… 28
2.5.2　用户态和内核态内存交互 …… 29
2.5.3　内存池 …… 30
2.5.4　物理地址到虚拟地址的映射 …… 31
2.5.5　内核空间到用户空间的映射 …… 31
2.6　中断处理 …… 32
2.6.1　硬件中断 …… 32
2.6.2　软中断机制 …… 35
2.7　/proc 系统 …… 36
2.8　工作队列 …… 38
2.9　异步 I/O …… 39
2.10　DMA …… 42

2.11 platform 概念……43
2.12 简单驱动例程……45
2.12.1 信号量同步……45
2.12.2 阻塞式读写……46
2.12.3 定时器……48
2.12.4 内存映射……49
2.12.5 /proc 访问……53
2.12.6 工作队列……55

第 3 章 GPIO 驱动……57
3.1 ARM 体系结构概述……57
3.1.1 RISC 结构……57
3.1.2 处理器模式……58
3.1.3 寄存器组织……58
3.1.4 异常处理……60
3.2 S3C2410X 处理器……61
3.3 S3C2410X I/O 端口……63
3.4 最简单的设备驱动——LED 灯驱动……64
3.5 S3C2410X GPIO 键盘驱动……66

第 4 章 串行总线驱动……73
4.1 串行总线综述……73
4.1.1 I²C 总线……73
4.1.2 SMBus 总线……75
4.1.3 SPI 总线……76
4.1.4 CAN 总线……76
4.2 CAN 接口芯片 MCP2510……79
4.2.1 数据发送……79
4.2.2 数据接收……81
4.2.3 中断……83
4.2.4 波特率设置……84
4.2.5 工作模式……85
4.3 MCP2510 驱动开发……86
4.4 Linux 的 I²C 驱动架构……96
4.5 Linux I²C 驱动开发……100

第 5 章 USB 驱动程序……102
5.1 USB 总线……102
5.1.1 USB 总线概述……102
5.1.2 USB 系统组成……102
5.1.3 USB 传输模式……104
5.1.4 主机规范……105
5.1.5 USB 设备描述符……105
5.1.6 HID 类规范……110
5.2 Linux 下的 USB 驱动框架……111
5.3 USB 请求块 urb……114
5.4 USB 骨架程序……118
5.5 USB 文件系统……126
5.6 USB 摄像头驱动……127
5.6.1 USB 摄像头原理……127
5.6.2 Video4Linux 规范……128
5.6.3 OV511 驱动分析与编译……132
5.6.4 spca5xx 编译与使用……139
5.7 USB Gadget……140
5.7.1 USB 设备控制器驱动……142
5.7.2 Gadget 驱动……146

第 6 章 Linux Framebuffer 驱动……150
6.1 LCD 原理……150
6.2 Linux 下 LCD 驱动架构……151
6.3 S3C2410X LCD 控制器……157
6.4 S3C2410X LCD 驱动开发……163
6.5 基于 Framebuffer 的界面系统开发……168

第 7 章 输入子系统驱动……174
7.1 Linux 输入设备驱动……174
7.2 键盘输入设备驱动……179
7.3 在 MiniGUI 中加入键盘驱动……184
7.4 LED 输入设备驱动……188
7.5 USB 鼠标输入设备驱动……190

第 8 章 触摸屏驱动……196
8.1 触摸屏原理……196
8.2 S3C2410X 触摸屏控制器……197
8.3 S3C2410X 触摸屏驱动设计……200
8.4 校准原理及编程思路……204
8.4.1 线性校准原理……205
8.4.2 三点校准原理……205

8.5 利用 tslib 库校准 ……207
8.6 在 MiniGUI 中加入触摸屏驱动 ……211

第 9 章 块设备驱动 ……213
9.1 Linux 块设备驱动 ……213
9.2 简单块设备驱动 ……217
9.3 Linux 文件系统 ……221
9.4 MTD 驱动分析 ……223
9.5 cramfs 文件系统 ……224
9.6 NAND 和 NOR Flash ……225
9.7 在系统中添加 JFFS2 分区 ……226

第 10 章 SD 卡驱动 ……229
10.1 SD 卡概述 ……229
10.2 SD 卡的通信 ……231
10.3 SD 卡寄存器 ……233
10.4 Linux 对 SD 卡的支持 ……235
10.4.1 重要数据结构 ……236
10.4.2 MMC/SD 卡块设备驱动 ……238
10.4.3 SD 卡扫描 ……243
10.5 如何开发一个 SD 驱动 ……244

第 11 章 网络设备驱动 ……249
11.1 网络驱动基础 ……249
11.2 sk_buff ……253
11.3 Linux 网络设备驱动架构 ……256
11.4 一个虚拟网络设备驱动 ……259
11.5 DM9000 网卡芯片 ……262
11.6 DM9000 网卡驱动程序分析 ……265

第 12 章 红外设备驱动 ……276
12.1 红外通信协议规范 ……276
12.2 S3C2410X 红外接口 ……277
12.3 S3C2410X 红外设备驱动 ……279
12.4 Linux 对红外网络通信的支持 ……282
12.5 红外 SOCKET 通信 ……285

第 13 章 音频设备驱动 ……291
13.1 Linux 音频体系 ……291
13.2 UDA1341TS 音频原理 ……292
13.3 S3C2410X 的音频接口 ……294
13.4 UDA1341TS 驱动开发 ……302
13.5 音频应用层编程 ……308
13.5.1 OSS 音频编程接口 ……308
13.5.2 ALSA 音频编程接口 ……310

附录：深圳优龙科技 YL2410 开发板简介 ……313

主要参考文献 ……316

第1章 Linux驱动程序基础

Linux是操作系统领域的奇迹。Linux操作系统的迅猛发展，与其具有的良好特性是分不开的。Linux是一种性能优良、源码公开、多用户、多任务操作系统，目前主要运用在大型服务器领域、网络处理应用和嵌入式系统。为了加强在嵌入式系统领域的优势，Linux 2.6已经在内核中加入了提高中断性能和调度响应时间的改进，包括采用可抢占内核、效率更高的调度算法和同步特性。另外，Linux 2.6内核加入了包括S3C2410X在内的多种微控制器的支持，并开始支持多种流行的无MMU单元的微控制器，如Dragonball、ColdFire、Hitachi H8/300。掌握嵌入式Linux驱动开发逐渐成为一种趋势。

1.1 驱动程序的概念

驱动程序实际上就是硬件与应用程序之间的中间层。驱动程序工作在内核空间，应用程序一般运行于用户态。在内核态下，CPU可执行任何指令，在用户态下CPU只能执行非特权指令。当CPU处于内核态，可以随意进入用户态；而当CPU处于用户态，只能通过特殊的方式进入内核态，比如Linux操作系统中的系统调用。系统调用是操作系统内核和应用程序之间的接口，设备驱动程序是操作系统内核和机器硬件之间的接口。

设备驱动程序的本质是实现逻辑设备到物理设备的转换，启动相应的I/O设备，发出I/O命令，完成相应的I/O操作，它是内核与外围设备数据交流的核心代码。设备驱动程序为应用程序屏蔽了硬件的细节。在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作。

编写设备驱动必须掌握操作系统的工作原理，并对硬件设备的运行机制有清楚的认识。开发设备驱动程序必须特别注意代码的安全性。因为驱动程序的错误往往导致整个操作系统崩溃。另外，同一个设备驱动可能被不同的进程调用，所以开发设备驱动程序必须考虑并发问题的处理。

1.2 Linux驱动程序模型

在Linux操作系统中，设备驱动程序对各种不同设备提供了一致的访问接口，把设备映射为一个特殊的设备文件，用户程序可以像对其他文件一样对此设备文件进行操作。Linux支持三类硬件设备：字符设备、块设备及网络设备。

字符设备接口支持面向字符的I/O操作，它不经过系统的快速缓存，所以它们负责管理自己

的缓冲区结构。字符设备接口只支持顺序存取的功能，一般不能进行任意长度的 I/O 请求，而是限制 I/O 请求的长度必须是设备要求的基本块长的倍数。典型的字符设备包括鼠标、键盘、串行口等。

块设备接口主要是针对磁盘等慢速设备设计的，以免耗费过多的 CPU 等待时间。它仅支持面向块的 I/O 操作，所有 I/O 操作都通过在内核地址空间中的 I/O 缓冲区进行，它可以支持几乎任意长度和任意位置上的 I/O 请求，即提供随机存取的功能。块设备主要包括硬盘软盘设备、CD-ROM 等。

LINUX 操作系统中的网络设备是一类特殊的设备。Linux 的网络子系统主要是基于 BSD unix 的 socket 机制。在网络子系统和驱动程序之间定义有专门的数据结构（sk_buff）进行数据传递。Linux 操作系统支持对发送数据和接收数据的缓存，提供流量控制机制，提供对多协议的支持。网络接口不存在于 Linux 的文件系统中，而是在核心中用一个 device 数据结构表示。对每一个字符设备或块设备的访问是通过文件系统中相应的特殊设备文件来进行的。网络设备在做数据包发送和接收时，直接通过接口访问，不需要进行文件的操作；而对字符设备和块设备的访问都需通过文件操作界面。

Linux 系统为每个设备分配了一个主设备号与次设备号，主设备号唯一标识了设备类型，次设备号标识具体设备的实例。由同一个设备驱动控制的所有设备具有相同的主设备号。从设备号用来区分具有相同主设备号且由相同设备驱动控制的不同设备。系统中每种设备都用一种特殊的设备相关文件来表示，例如系统中第一个 IDE 硬盘表示成/dev/hda。例如串口设备/dev/tty0 与/dev/tty1，它们的主设备号为 4，次设备号分别为 0 和 1。例如主 IDE 硬盘的每个分区的从设备号都不相同。如/dev/hda2 表示主 IDE 硬盘的主设备号为 3，而从设备号为 2。块设备和字符设备的设备相关文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但当 Linux 寻找和初始化网络设备时才建立这种文件。在驱动程序中，可以使用下列宏获得驱动的设备号：

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

如果想把设备号转换成 dev_t 类型，使用：

```
MKDEV(int major,int minor);
```

Linux 2.6 内核的一个重要特色是提供了统一的内核设备模型。随着技术的不断进步，系统的拓扑结构越来越复杂，对智能电源管理、热插拔以及 Plug and Play 的支持要求也越来越高，Linux 2.4 内核已经难以满足这些需求。为适应这种形势的需要，Linux 2.6 内核开发了全新的设备模型，它采用 sysfs 文件系统，该文件系统是一个类似于 proc 文件系统的特殊文件系统，用于将系统中的设备组织成层次结构，并向用户模式程序提供详细的内核数据结构信息。

Linux 2.6 引入新的设备管理机制 kobject，通过这个数据结构使所有设备在底层都具有统一的接口，kobject 提供基本的对象管理，是构成 Linux 2.6 设备模型的核心结构，它与 sysfs 文件系统紧密关联，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。kobject 通常通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合。

```
struct kobject {
  char          * k_name;
  char          name[KOBJ_NAME_LEN];
  atomic_t      refcount;
  struct list_head  entry;
  struct kobject    * parent;
  struct kset       * kset;
  struct kobj_type  * ktype;
  struct dentry     * dentry;
};
struct kset {
  struct subsystem  * subsys;
  struct kobj_type  * ktype;
  struct list_head  list;
  struct kobject    kobj;
  struct kset_hotplug_ops   * hotplug_ops;
};
```

在这些内核对象机制的基础上，Linux 的设备模型（/include/linux/device.h）包括设备结构 devices、驱动结构 drivers、总线结构 buses、设备类结构 classes 几个关键组件。Linux 中的任一设备在设备模型中都由一个 device 对象描述，其对应的数据结构 struct device 定义为：

```
struct device {
  struct list_head node;              //在同类设备列表中的节点
  struct list_head bus_list;          //在总线列表中节点
  struct list_head driver_list;
  struct list_head children;
  struct device     * parent;    //父设备
  struct kobject kobj;
  char  bus_id[BUS_ID_SIZE];     //父总线上的 ID
  struct bus_type   * bus;       //设备挂接的总线类型
  struct device_driver *driver; //分配本设备的驱动
  void      *driver_data;        //私有数据
  void      *platform_data;      //平台特定数据（包括与设备相关的 ACPI，BIOS 数据）
  struct dev_pm_info power;
  u32 power_state;               //电源状态，在 ACPI 中包括 D0～D3，D0 表示全功能，
                                 //D3 表示停止
  unsigned char *saved_state;    //设备状态
  u32       detach_state;        //设备删除时进入的状态
  u64       *dma_mask;           //DMA 掩码
  u64       coherent_dma_mask;   //类似 DMA 掩码，用于 alloc_coherent mappings
  struct list_head  dma_pools;   //DMA 池
  void  (*release)(struct device * dev);
};
//注册与注销函数
int device_register(struct device * dev);
void device_unregister(struct device * dev);
```

系统中的每个驱动程序由一个 device_driver 对象描述，对应的数据结构定义为：

```
struct device_driver {
    char * name;                          //设备驱动程序的名称
    struct bus_type * bus;                //该驱动所管理的设备挂接的总线类型
    struct semaphore    unload_sem;
    struct kobject      kobj;             //内嵌 kobject 对象
    struct list_head    devices;          //该驱动所管理的设备链表头
    int (*probe)(struct device * dev);    //指向设备探测函数，用于探测设备是否可以被
                                          //该驱动程序管理
    int (*remove)    (struct device * dev);              //用于删除设备的函数
    void(*shutdown) (struct device * dev);              //停止设备的函数
    int (*suspend)   (struct device * dev, u32 state, u32 level);//挂起设备的函数
    int (*resume)    (struct device * dev, u32 level);  //恢复设备的函数
};
//注册与注销函数
int driver_register(struct device_driver * drv);
void driver_unregister(struct device_driver * drv);
```

系统中总线由 struct bus_type 描述，定义为：

```
struct bus_type
{
    char * name;                                //总线类型的名称
    struct subsystem subsys;                    //与该总线相关的 subsystem
    struct kset drivers;                        //所有与该总线相关的驱动程序集合
    struct kset devices;                        //所有挂接在该总线上的设备集合
    struct bus_attribute * bus_attrs;           //总线属性
    struct device_attribute * dev_attrs;        //设备属性
    struct driver_attribute * drv_attrs;        //驱动程序属性
    int (*match)(struct device * dev, struct device_driver * drv);
    int (*hotplug) (struct device *dev, char **envp, int num_envp, char *buffer,
    int buffer_size);
    int (*suspend)(struct device * dev, u32 state);
    int (*resume)(struct device * dev);
};
```

Linux 2.6 源代码中设备驱动大多数放在/drivers 目录。音频驱动则是一个例外，被放到源代码根目录下的/sound 目录中。表 1.1 为/drivers 目录下重要的子目录的介绍。

表 1.1　内核驱动目录

目录	主要内容	目录	主要内容
/drivers/char	字符型设备驱动	/drivers/media	视频采集、广播、数字电视设备
/drivers/block	块设备驱动	/drivers/base	一切驱动基本函数
/drivers/net	网络设备驱动	/drivers/usb	USB 设备驱动
/drivers/video	显示相关驱动、控制台设备、启动 LOGO	/drivers/mtd	MTD 设备驱动，包括 FLASH 驱动
/drivers/mmc	MMC/SD 卡驱动	/drivers/serial	串口设备驱动

1.3 最基本的调试手段

在内核编程中，不能使用用户态 C 库函数中的 printf()函数输出信息，而只能使用 printk()。printk 函数的用法：

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__,__LINE__);
```

宏__FILE__代表文件名，宏__LINE__代表代码在文件的行号。输出消息前的宏是优先级，内核一共有 8 个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。Linux 2.6 中 default_message_loglevel 是 kern_warning。如果优先级数字比 console_loglevel 变量小的话，消息就会打印到控制台上。console_loglevel 被初始化成 default_console_loglevel。注意只有在非界面环境才能看到打印输出，在图形界面下的终端里是看不到输出的。表 1.2 是 8 个日志级别。

表 1.2 错误日志级别

KERN_EMERG	紧急信息	KERN_WARNING	警告
KERN_ALERT	需要立即处理的情况	KERN_NOTICE	正常情况，但值得关注
KERN_CRIT	严重错误	**KERN_INFO**	提醒消息
KERN_ERR	硬件错误	KERN_DEBUG	调试信息

1.4 导出符号的方法

Linux2.4 内核下，默认情况时模块中的非静态全局变量及函数在模块加载后会输出到内核空间。Linux 2.6 内核下，默认情况时模块中的非静态全局变量及函数在模块加载后不会输出到内核空间，需要显式调用宏 EXPORT_SYMBOL 才能输出。所以在 Linux 2.6 内核的模块下，EXPORT_NO_SYMBOLS 宏的调用没有意义，是空操作。在同时支持 Linux 2.4 与 Linux 2.6 内核的设备驱动中，可以通过以下代码段来输出模块的内核符号。同时支持 Linux 2.4 与 Linux 2.6 的输出内核符号代码段：

```
EXPORT_NO_SYMBOLS;
EXPORT_SYMBOL(var);
EXPORT_SYMBOL(func);
```

需要注意的是如需在 Linux 2.4 内核下使用 EXPORT_SYMBOL，必须在 CFLAGS 中定义 EXPORT_SYMTAB，否则编译将会失败。从编写良好的代码风格角度出发，对于不需要输出到内核空间，而且模块中其他文件没有用到的全局变量及函数，最好显式声明为 static 类型；而需要输出的内核符号，在命名时通常要用模块名作为前缀。模块加载后，Linux 2.4 内核下可通过/proc/ksyms、Linux 2.6 内核下可通过/proc/kallsyms 查看模块输出的内核符号。

1.5 动态加载驱动程序

Linux 设备驱动属于内核的一部分，它可以采用两种方式被编译和加载：（1）直接编译进 Linux 内核，随同 Linux 启动时加载；（2）编译成一个可加载模块，使用 insmod 加载，使用 rmmod 删除。Linux 系统的可加载模块（Loadable Kernel Modules）是用于扩展 Linux 系统的功能的。使用内核模块的优点有:它们可以按照需要被动态地加载，而且不需要重新编译内核。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。Linux 2.6 中的模块必须包含以下基本接口：

```
module_init(your_init_func);
module_exit(your_exit_func);
```

加载一个模块（常常只限于 root 能够使用）的命令是：

```
insmod modulename.ko
```

卸载一个内核模块的命令是：

```
rmmod modulename
```

内核代码 include/linux/module.h 中定义的宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。在装载内核模块时，用户可以向模块传递一些参数：

```
insmod modname var=value
```

如果用户未指定参数，var 将使用模块内定义的默认值。

Linux 设备驱动属于内核的一部分，它直接被编译进内核，也可以作为可加载模块动态加载。编译进内核的驱动随系统启动而加载，而作为动态加载模块则在执行 insmod 时加载。

1.6 在内核中加入新驱动

如果希望将驱动编译进内核，需要修改内核代码。下面以字符型设备为例，说明如何在 Linux 2.6 中加一个新的设备驱动。如果驱动代码文件为 pxa_smbus.c，将 pxa_smbus.c 复制到/drivers/char 目录，更改该目录下 Kconfig，增加：

```
config SMBUS_PMIC
    tristate "SMBUS Driver for PMIC"
    depends on ARCH_PXA || ARCH_SA1100
    default y
```

```
    help
```

在该目录下的 Makefile 中增添下行：

```
obj-$(CONFIG_SMBUS_PMIC)+=pxa_smbus.o
```

进入源代码目录，执行 make menuconfig 后，选择 character devices 项，进入图 1.1 所示的界面，在图中选项前如果为<*>，表示模块被编译进内核；如果为<M>，表示编译成可加载模块；如果是< >则表示不编译。如果选择<*>，用 make zImage 就可以了。如果选择<M>，则必须使用 make 命令，生成 ko 文件。

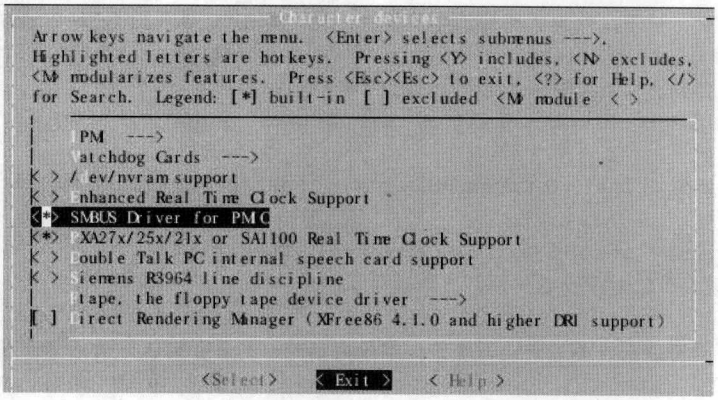


图 1.1　在内核中增加新驱动

1.7　应用程序操作接口

在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作，设备驱动需要提供应用程序操作接口，如 open 、close、read、write、seek、ioctl 等。驱动程序的原理如图 1.2 所示。

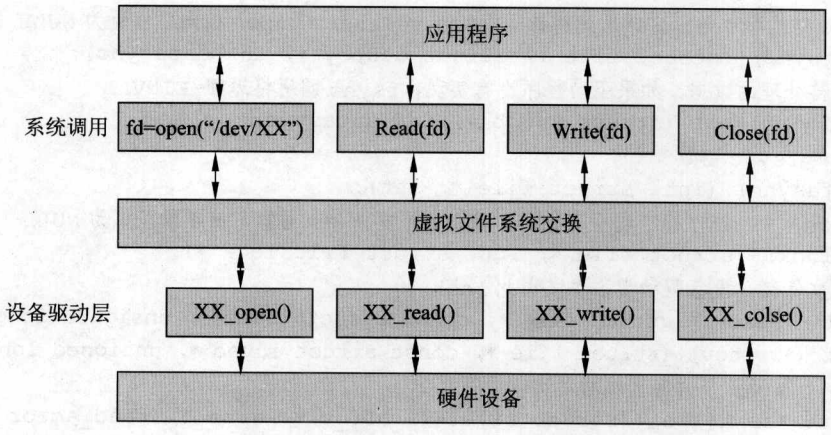


图 1.2　驱动程序的原理

内核定义了一个 struct file_operations 结构体，这个结构的每一个成员的名字都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如读写操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

```
struct file_operations {
  struct module *owner;
  //模块所有者指针，一般初始化为 THIS_MODULES
  loff_t (*llseek) (struct file *, loff_t, int);
  //用来修改文件当前的读写位置，返回新位置。loff_t 为一个"长偏移量"
  ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);
  //同步读取函数。读取成功返回读取的字节数。设置为 NULL，调用时返回-EINVAL
  ssize_t (*aio_read) (struct kiocb *, char _user *, size_t, loff_t);
  //异步读取操作，为 NULL 时全部通过 read 处理
  ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);
  //同步写入函数
  ssize_t (*aio_write) (struct kiocb *, const char _user *, size_t, loff_t);
  //异步写入操作
  int (*readdir) (struct file *, void *, filldir_t);
  // 仅用于读取目录，对于设备文件，该字段为 NULL
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  //判断目前是否可以对设备进行读写操作。字段为空时，设备会被认为既可读也可写
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  // 向设备发送 I/O 控制命令的函数。不设置入口点，返回-ENOTTY
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  // 不使用 BLK 的文件系统，将使用此种函数指针代替 ioctl
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  // 在 64 位系统上，32 位的 ioctl 调用，将使用此函数指针代替
  int (*mmap) (struct file *, struct vm_area_struct *);
  // 用于请求将设备内存映射到进程地址空间。如果无此方法，将访问-ENODEV
  int (*open) (struct inode *, struct file *);
  // 打开设备的函数，如果为空，设备的打开操作永远成功，但系统不会通知驱动程序
  int (*flush) (struct file *);
  // 进程在关闭设备文件描述符副本时调用，执行未完成的操作
  int (*release) (struct inode *, struct file *);
  // file 结构释放时，将调用此指针函数，若 release 与 open 相同可设置为 NULL
  int (*fsync) (struct file *, struct dentry *, int datasync);
  // 刷新待处理的数据，如果驱动程序没有实现，fsync 调用将返回-EINVAL
  int (*aio_fsync) (struct kiocb *, int datasync);
  // 异步的 fsync 函数
  int (*fasync) (int, struct file *, int);
  // 通知设备 FASYNC 标志发生变化，如果设备不支持异步通知，该字段可以为 NULL
  int (*lock) (struct file *, int, struct file_lock *);
  // 实现文件锁，设备驱动常不去实现此 lock
  ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
  //实现进行涉及多个内存区域的单次读或写操作
  ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
  // 实现 sendfile 调用的读取部分，将数据从一个文件描述符移到另一个
```