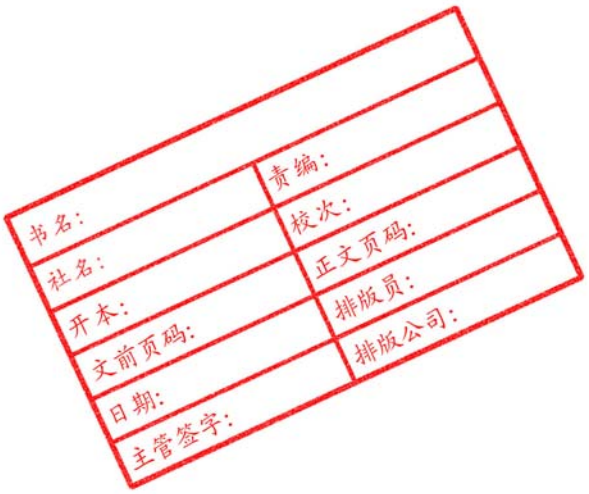
电子工程应用精讲系列

嵌入式Linux驱动程序和系统开发实例精讲

罗苑棠　编著



電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内 容 简 介

本书是《嵌入式 Linux 应用系统开发实例精讲》的改版。全书通过大量实例精讲的形式，详细介绍了嵌入式 Linux 驱动程序与系统开发的方法与流程。全书分 3 篇共 19 章，第 1 篇为基础知识篇，介绍了 Linux 的移植、开发环境平台、Linux 程序设计基础及常用开发工具，引导读者技术入门。第 2 篇为 Linux 驱动程序开发与实例篇，结合 6 个实际案例阐述了网卡驱动、声卡驱动、显卡驱动、USB 驱动、闪存 Flash 驱动的开发原理技术和应用。第 3 篇为 Linux 系统开发实例篇，安排了 8 个实际应用系统实例，涵盖工业设备、视频处理、指纹识别、网络传输通信、摄像监控、移动校园系统等嵌入式热门领域，实战和商业价值高，利于读者举一反三，快速掌握 Linux 系统设计的流程，提高实际设计能力。

本书配有光盘 1 张，包含了全书所有实例的硬件原理图和程序源代码，方便读者学习和使用。本书适合计算机、自动化、电子及通信等相关专业的大学生，以及从事 Linux 开发的科研人员使用。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

嵌入式 Linux 驱动程序和系统开发实例精讲 / 罗苑棠编著. —北京：电子工业出版社，2009.1
（电子工程应用精讲系列）
ISBN 978-7-121-07936-8

Ⅰ. 嵌… Ⅱ. 罗… 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 188979 号

责任编辑：王鹤扬
印　　刷：北京市通州大中印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：30.75　字数：746 千字
印　　次：2009 年 1 月第 1 次印刷
印　　数：4000 册　定价：59.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

丛书说明

工程技术的电子化、集成化和系统化促进了电子工程技术的发展，同时也促进了电子工程技术在社会各行业中的广泛应用，从近年的人才招聘市场来看，电子工程师的人才需求更是一路走高。

电子工程师如此紧俏，除需求不断走高，人才供不应求外，另一重要原因则是电子工程师的门槛相对而言比较高，这个高门槛则来自于工程师的“经验”和“实践”!

因此，为了满足读者学习和工作需要，解决各种工作中的专业问题，我们紧紧围绕“经验”和“实践”，精心策划组织了此套丛书。

第1篇 Linux 基础知识 1
第1章 嵌入式基础入门 2
第2章 Linux 系统开发环境平台 33
第3章 嵌入式

1. 丛书范围

现代电子科学技术的一个特点，工程师应当了解、掌握两门以上的相关学科，知识既精深又广博是某领域专家的重要标志。本丛书内容涉及软件开发、研发电子及嵌入式项目开发等，包括单片机、USB 接口、ARM、CPLD/FPGA、DSP 和移动通信系统等。

2. 读者对象

本套书面向各领域的初、中级用户，具体为高校计算机、电子信息、通信工程、自动化控制专业在校大学生，以及从事电子开发和应用行业的科研人员。

3. 内容组织形式

本套书紧紧围绕“经验”和“实践”，首先介绍一些相关的基础知识，然后根据不同

的模块或应用领域，分篇安排应用程序实例的精讲。基础知识用来为一些初级读者打下一定的知识功底；基础好一点的读者则可以跳过这一部分，直接进入实例的学习。

4．实例特色

在应用实例的安排上，着重突出“应用”和“实用”两个基本原则，安排具有代表性、技术领先性，以及应用广泛的典型实例，让读者学习借鉴。这些实例是从作者多年程序开发项目中挑选出来的，也是经验的归纳与总结。

在应用实例的讲解上，既介绍了设计原理、基本步骤和流程，也穿插了一些经验、技巧与注意事项。特别在程序设计思路上，在决定项目开发的质量和成功与否的细节上，尽可能地用简洁的语言来清晰阐述大众易于理解的概念和思想；同时，程序代码部分做了很详细的中文注释，有利于读者举一反三，快速应用和提高。

5．光盘内容

本套书的光盘中包含了丰富的实例原图文件和程序源代码，读者稍加修改便可应用于自己的工作中或者完成自己的课题（毕业设计），物超所值。读者使用之前，最好先将光盘内容全部复制到电脑硬盘中，以便于以后可以直接调用，而不需要反复使用光盘，提高操作速度和学习效率。

6．学习指南

对于有一定基础的读者，建议直接从实例部分入手，边看边上机练习，这样印象会比较深，效果更好。基础差一点的读者请先详细学习书中基础部分的理论知识，然后再进行应用实例的学习。在学习中，尽量做到反复理解和演练，以达到融会贯通、举一反三的功效；特别希望尽量和自己的工作设计联系起来，以达到“即学即会，学以致用”的最大化境界。

本套丛书主要偏重于实用性，具有很强的工程实践指导性。期望读者在学习中顺利、如意！

前　言

Linux 是一套免费使用和自由传播的类 UNIX 操作系统，这个系统是由世界各地成千上万的程序员设计和实现的。它以高效性和灵活性著称，并且能够在 PC 上实现全部的 UNIX 特性，具有多任务、多用户的能力。Linux 现在受到了广大计算机爱好者的喜爱，原因主要有两个：一是 Linux 属于自由软件，用户不用支付任何费用就可以获得它及其源代码，并且可以根据自己的需要进行必要的修改；另一个原因是它具有 UNIX 的全部功能。

随着 Linux 在我国政府、金融、电信、消费电子等行业的广泛应用，企业对 Linux 人才的需求也开始持续升温。目前 IT 业内许多著名大企业都有急剧扩招 Linux 人才的倾向。巨大的人才需求将使更多的人参与到 Linux 学习的行列中来。

Linux 应用领域比较多，比较常用的有服务器配置与应用、驱动设备开发、嵌入式系统开发等。目前市场上虽存在一些 Linux 驱动程序与嵌入式系统设计图书，但大多以介绍基础理论为主，缺乏商业应用案例的实践指导。本书就是为了弥补这种不足而精心组织编写的。

本书内容

全书以理论为辅、实践为主，重点以典型实例的形式，详细介绍嵌入式 Linux 驱动程序与系统开发的思路、方法与实际应用案例。全书分 3 篇共 19 章，具体内容如下：

第 1~4 章为基础知识篇，主要讲述了嵌入式基础入门、Linux 环境开发平台、C 程序设计基础、Linux 常用开发工具。通过本部分学习，初级读者可以具备一定的 Linux 程序设计功底；基础好一点的读者则可以跳过这一部分。

第 5~10 章为 Linux 驱动程序开发与实例篇，结合 6 个实际案例阐述了网卡驱动、声卡驱动、显卡驱动、USB 驱动、闪存 Flash 驱动的开发原理技术和应用。

第 11~19 篇为 Linux 系统开发实例篇，安排了 8 个实际应用系统实例，涵盖工业设备、视频处理、指纹识别、网络传输通信、摄像监控、移动校园系统等领域，这些实例具有代表性、技术领先性、应用广泛性及热门性的特点，全部调试通过并进入商品化，是作者多

年开发经验的归纳与总结。

本书特色

与同类型书相比，本书主要具备以下一些特色。

（1）整体讲解思路：首先简要讲述了 Linux 嵌入式系统开发的环境平台、程序基础和常用开发工具，然后是驱动程序与系统开发典型实例的介绍，并穿插了一些经验、技巧与注意事项，符合读者循序渐进的学习过程。

（2）包括 GUI、QT 图形工具、驱动程序开发及系统实例的介绍，使本书在内容上更加完美、全面。

（3）本书 6 个驱动程序实例和 8 个应用系统实例，全部典型实用，涉及 Linux 开发的诸多热门与核心技术，工程实战价值高。

（4）本书不但提供了程序设计的详细思路与流程，而且对实例的程序代码做了详细注释，利于读者理解和巩固知识点，学会举一反三。

（5）光盘中包含了丰富的实例硬件电路图文件和程序源代码，读者稍加修改，便可应用于自己的工作中或者完成自己的课题设计，物超所值。

- **光盘的内容说明**　该光盘为实例素材文件，按照章节序号来组织，每章包括电路图、程序代码两部分内容。其中，“电路图”文件夹中的内容为各章的电路图，多用 Protel 软件制作。
- **光盘的使用说明**　光盘中的程序需要采用 C 语言的编译软件打开阅读，也可以使用“UltraEdit”等软件打开阅读或者编辑。
- **系统要求**　该光盘运行只需一般的 PC 就可以。系统配置推荐为 256MB 以上内存，1280×1024 分辨率，32MB 以上显存。

本书读者对象为计算机、电子信息及相关专业的在校大学生，还有从事 Linux 嵌入式开发的初、中级设计人员。

本书主要由罗苑棠编写。另外参加编写的人员还有唐清善、邱宝良、周克足、刘 斌、李亚捷、李永怀、李宁宇、刘伟捷、黄小欢、严剑忠、黄小宽、李彦超、付军鹏、张广安、贾素龙、王艳波、金平、徐春林、谢正义、郑贞平、张小红等。他们在资料收集、整理和技术支持方面做了大量的工作，在此一并向他们表示感谢！

由于时间仓促，再加之作者的水平有限，书中难免存在一些不足之处，欢迎广大读者批评和指正，联系方式 jsj@phei.com.cn。

编　者

2008 年 11 月

目　录

第 1 篇　Linux 基础知识

第 1 章　嵌入式基础入门 …… 2
1.1　嵌入式操作系统简介 …… 2
1.1.1　嵌入式系统的基本概念 …… 2
1.1.2　嵌入式系统的内核介绍 …… 3
1.1.3　嵌入式系统的应用领域 …… 4
1.2　Linux 操作系统概述 …… 5
1.2.1　嵌入式 Linux 发展现状 …… 5
1.2.2　Linux 相关的常用术语 …… 6
1.3　Linux 操作系统的移植 …… 8
1.3.1　BootLoader 技术详解 …… 8
1.3.2　Linux 内核基本结构 …… 17
1.3.3　移植 Linux 操作系统 …… 28
1.4　本章总结 …… 32

第 2 章　Linux 系统开发环境平台 …… 33
2.1　进程/线程管理 …… 33
2.1.1　进程/线程的概念 …… 33
2.1.2　进程基本操作 …… 37
2.1.3　进程通信与同步 …… 49
2.1.4　线程基本操作 …… 57
2.1.5　简单的多线程编程 …… 59
2.2　文件系统结构和类型 …… 62
2.2.1　FAT 文件系统 …… 62
2.2.2　RAMFS 内核文件系统 …… 66
2.2.3　JFFS 与 YAFFS 文件系统 …… 68
2.2.4　EXT2/EXT3 文件系统 …… 71
2.2.5　/proc 文件系统 …… 74
2.2.6　Linux 文件操作函数 …… 75
2.3　存储管理 …… 79
2.3.1　MTD 内存管理 …… 79
2.3.2　Linux 内存管理 …… 83
2.4　设备管理 …… 84
2.4.1　概述 …… 84
2.4.2　字符设备与块设备 …… 84
2.4.3　主设备号和次设备号 …… 87
2.5　本章总结 …… 88

第 3 章　嵌入式 Linux 程序设计基础 …… 89
3.1　建立嵌入式 Linux 交叉编译环境 …… 89
3.1.1　编译环境概述 …… 89
3.1.2　建立交叉编译环境流程 …… 92
3.2　工程管理器 make …… 97
3.2.1　make 概述 …… 97
3.2.2　Makfile 文件书写规则 …… 101
3.3　Linux C/C++程序设计 …… 104
3.3.1　C/C++程序结构 …… 104
3.3.2　C/C++数据类型 …… 107
3.3.3　表达式/语句、函数 …… 108
3.3.4　C/C++设计注意事项 …… 111
3.4　Linux 汇编程序设计 …… 117
3.4.1　Linux 汇编语法格式 …… 118
3.4.2　汇编程序实例 …… 119
3.5　Linux Shell 语言编程 …… 120

3.5.1 Shell 环境变量及配置文件 …… 121
3.5.2 Shell 编程实例 …… 123
3.6 Linux Perl 语言编程 …… 124
3.6.1 Perl 基本程序 …… 124
3.6.2 Perl 变量 …… 125
3.6.3 文件句柄和文件操作 …… 128
3.6.4 循环结构 …… 129
3.6.5 条件结构 …… 130
3.7 本章总结 …… 131
第 4 章 Linux 常用开发工具 …… 132
4.1 GCC 编译器 …… 132
4.1.1 GCC 版本信息 …… 132
4.1.2 GCC 目录结构 …… 132
4.1.3 GCC 执行过程 …… 133
4.1.4 GCC 的基本用法和选项 …… 134
4.1.5 g++ …… 134
4.2 gdb 调试器 …… 135
4.2.1 基本用法和选项 …… 135
4.2.2 gdb 常用命令 …… 135
4.3 Linux 汇编工具 …… 136
4.3.1 汇编器 …… 136
4.3.2 链接器 …… 136
4.3.3 调试器 …… 137
4.3.4 系统调用 …… 137
4.3.5 命令行参数 …… 137
4.3.6 GCC 内联汇编 …… 138
4.4 Linux 调试工具 …… 139
4.4.1 JTAG 调试工具 …… 139
4.4.2 kgdb 内核调试环境 …… 144
4.5 Linux 图形开发工具 …… 149
4.5.1 GUI 图形界面开发 …… 149
4.5.2 GTK 图形开发工具 …… 157
4.5.3 QT 图形开发工具 …… 161
4.6 本章总结 …… 167

第 2 篇 Linux 驱动程序开发与实例

第 5 章 Linux 设备驱动基础 …… 170
5.1 驱动程序基本概念 …… 170
5.1.1 驱动程序与应用程序的区别 … 170
5.1.2 内核版本与编译器的版本依赖 …… 171
5.2 设备驱动模块概述 …… 171
5.2.1 模块的基本概念 …… 171
5.2.2 模块的初始化和退出 …… 172
5.2.3 Linux 内核模块加载 …… 174
5.3 Linux 设备驱动结构分析 …… 176
5.3.1 内核和用户接口 …… 176
5.3.2 inode 节点 …… 177
5.3.3 File 结构 …… 178
5.4 常用接口函数介绍 …… 181
5.5 驱动程序的调试 …… 187
5.6 本章总结 …… 189
第 6 章 网卡驱动程序开发 …… 190
6.1 网卡概述 …… 190
6.2 RTL8193 网卡驱动 …… 190
6.2.1 网卡驱动的初始化 …… 191
6.2.2 网卡数据收发 …… 197
6.3 典型实例——Ralink 无线网卡驱动开发 …… 198
6.3.1 Ralink 无线网卡 …… 198
6.3.2 802.11 无线通信协议的选用 … 199
6.3.3 设备驱动关键数据结构 …… 200
6.3.4 rt2500 无线网卡驱动分析 …… 202
6.3.5 rt2500 程序源代码 …… 207
6.4 本章总结 …… 215
第 7 章 显卡驱动程序开发 …… 216
7.1 显卡驱动概述 …… 216
7.1.1 Linux framebuffer …… 216
7.1.2 帧缓冲设备数据结构 …… 220

7.2 典型实例——显卡 Framebuffer 驱动实现 225
7.2.1 Framebuffer 驱动框架程序 225
7.2.2 NVDIA 显卡设备驱动文件 231
7.3 本章总结 233
第 8 章 声卡驱动程序开发 234
8.1 声卡驱动概述 234
8.2 OSS 声卡驱动 234
8.3 ALSA 声卡驱动 235
8.4 典型实例——AC97 声卡驱动实现 237
8.4.1 AC97 驱动分析 237
8.4.2 Realtek 声卡驱动配置 241
8.5 本章总结 243
第 9 章 USB 驱动程序开发 244
9.1 USB 设备驱动概述 244
9.2 USB 驱动设备示例 245
9.2.1 Linux 驱动程序概述 245
9.2.2 驱动程序分析 246
9.3 典型实例——单片机的主从通信实例 253
9.3.1 主从通信介绍 253
9.3.2 USB 设备驱动程序 254
9.3.3 主机程序源代码 260
9.4 本章总结 261
第 10 章 闪存 Flash 驱动程序开发 262
10.1 Flash 闪存基础 262
10.2 Flash MTD 技术 264
10.3 典型实例 1——NAND Flash 驱动实例 265
10.3.1 NAND Flash 驱动设备 265
10.3.2 NAND Flash 驱动源代码 266
10.4 典型实例 2——NOR Flash 驱动实例 270
10.4.1 芯片驱动与 MTD 原始设备 270
10.4.2 NOR Flash 驱动分析 270
10.4.3 NOR Flash 驱动源代码 274
10.5 本章总结 276

第 3 篇 Linux 系统开发实例

第 11 章 嵌入式系统开发的模式与流程 278
11.1 嵌入式系统的结构 278
11.1.1 嵌入式系统的硬件架构 278
11.1.2 嵌入式系统的软件结构 278
11.2 嵌入式开发的模式及流程 279
11.2.1 嵌入式系统开发模式 279
11.2.2 嵌入式系统开发流程 280
11.3 本章总结 282
第 12 章 工业温度监控设备开发实例 283
12.1 应用环境与硬件设计概要 283
12.1.1 嵌入式 Linux 在工业控制领域的应用 283
12.1.2 工控串行通信协议标准 286
12.2 相关开发技术——异步串行通信接口 288
12.2.1 异步串行通信标准 288
12.2.2 设置串口控制信号 290
12.2.3 读入串口控制信号 291
12.2.4 文件 Open()系统调用 292
12.3 实例——基于 DS1820 的实时温度监控系统 292
12.3.1 系统基本结构 293
12.3.2 系统工作流程 296
12.3.3 系统模块源代码实现 298
12.4 本章总结 306
第 13 章 实时视频采集系统开发实例 307
13.1 应用环境与硬件设计概要 307

13.2 相关开发技术 …… 308
13.2.1 视频图像压缩技术 …… 308
13.2.2 视频采集驱动 …… 310
13.2.3 视频驱动加载运行 …… 313
13.3 实例——基于 MV86S02 实时视频采集系统设计 …… 313
13.3.1 系统基本结构 …… 313
13.3.2 系统工作流程 …… 316
13.3.3 系统模块源代码实现 …… 319
13.3.4 视频数据比较及分析 …… 335
13.4 本章总结 …… 336
第 14 章 指纹识别门禁系统开发实例 …… 337
14.1 应用环境与硬件设计概要 …… 338
14.2 相关开发技术 …… 340
14.2.1 指纹识别原理 …… 340
14.2.2 设备驱动编写框架 …… 344
14.2.3 指纹芯片驱动 …… 346
14.3 实例——基于 ARM Linux 的指纹识别门禁系统 …… 347
14.3.1 系统基本结构 …… 347
14.3.2 系统工作流程 …… 349
14.3.3 系统模块源代码实现 …… 350
14.4 本章总结 …… 360
第 15 章 基于 RTL8019 的以太网应用系统开发实例 …… 361
15.1 以太网应用技术概述 …… 361
15.2 相关开发技术 …… 362
15.2.1 基于 RTL8019 的以太网帧传输原理 …… 362
15.2.2 RTL8019 的初始化 …… 363
15.2.3 RTL8019 驱动程序的框架 …… 364
15.2.4 数据结构和函数 …… 365
15.2.5 RTL8109 驱动程序的加载 …… 368
15.3 实例——基于 RTL8019 的以太网应用系统设计 …… 368
15.3.1 系统基本结构 …… 368
15.3.2 系统工作流程 …… 371
15.3.3 系统模块源代码实现 …… 372
15.3.4 系统调试 …… 380
15.4 本章总结 …… 381
第 16 章 无线网络数据传输系统开发实例 …… 382
16.1 无线网络传输系统简介 …… 382
16.2 相关开发技术 …… 383
16.2.1 无线网络接入技术 …… 383
16.2.2 基于 PCMCIA 的无线网卡接口 …… 385
16.2.3 PCMCIA 驱动程序 …… 386
16.3 实例——基于 PCMCIA 的无线网络嵌入式前端系统设计 …… 387
16.3.1 系统基本结构 …… 387
16.3.2 系统工作流程 …… 389
16.3.3 系统模块源代码实现 …… 391
16.3.4 系统调试 …… 398
16.4 本章总结 …… 398
第 17 章 基于 PDIUSBD12 的数据传输系统实例 …… 399
17.1 USB 应用环境与硬件设计概要 …… 400
17.2 相关开发技术——USB 系统与总线驱动 …… 401
17.2.1 USB 系统组成 …… 401
17.2.2 USB Host 总线驱动 …… 402
17.2.3 USB Device 总线驱动 …… 403
17.3 实例——基于 PDIUSBD12 的数据传输设计 …… 406
17.3.1 系统基本结构 …… 406
17.3.2 系统工作流程 …… 412
17.3.3 系统模块源代码实现 …… 412
17.4 本章总结 …… 424

第 18 章　家庭安全监控系统设计实例 …… 425
18.1　应用环境与硬件设计概要 …… 425
18.1.1　系统功能和组成 …… 425
18.1.2　系统模块功能描述 …… 426
18.2　系统硬件结构 …… 430
18.2.1　Linux 客户端系统硬件结构 …… 430
18.2.2　传感器系统硬件结构 …… 433
18.3　系统软件结构 …… 435
18.3.1　Linux 客户端系统软件结构 …… 435
18.3.2　传感器系统软件结构 …… 438
18.4　Linux 客户端系统设计实现 …… 440
18.4.1　系统数据结构设计 …… 440
18.4.2　通信模块设计说明 …… 441
18.4.3　显示模块设计说明 …… 442
18.4.4　用户管理模块设计说明 …… 443
18.4.5　系统设置模块设计说明 …… 445
18.4.6　客户端主要代码与注释 …… 445
18.5　系统主要模块设计实现 …… 447
18.5.1　红外监控模块设计说明 …… 447
18.5.2　报警模块（warnning） …… 448
18.5.3　触发监控模块 …… 449
18.5.4　管理模块 …… 450
18.5.5　主要代码与注释 …… 453
18.6　本章总结 …… 459
第 19 章　移动校园系统设计实例 …… 460
19.1　应用环境与硬件设计概要 …… 460
19.1.1　系统功能和组成 …… 460
19.1.2　系统模块功能和软件图 …… 460
19.2　系统硬件结构 …… 462
19.3　系统软件结构 …… 463
19.3.1　软件整体结构 …… 463
19.3.2　软件模块结构 …… 464
19.3.3　接口设计 …… 467
19.3.4　运行过程设计 …… 468
19.3.5　系统数据结构设计 …… 469
19.3.6　搭建开发环境 …… 470
19.4　系统模块程序代码 …… 472
19.4.1　主函数 …… 472
19.4.2　Syllabus 课表模块 …… 472
19.4.3　BBS 论坛模块 …… 474
19.4.4　Map 地图模块 …… 476
19.4.5　Message 系统消息模块 …… 478
19.5　本章总结 …… 478

第 1 篇

Linux 基础知识

- 第 1 章　嵌入式基础入门
- 第 2 章　Linux 系统开发环境平台
- 第 3 章　嵌入式 Linux 程序设计基础
- 第 4 章　Linux 常用开发工具

第 1 章

嵌入式基础入门

随着微电子技术的飞速发展及后 PC 时代的到来，嵌入式芯片被广泛运用到消费、汽车、电子、微控制、无线通信、数码产品、网络设备、安全系统等领域。越来越多的公司、研究单位、大专院校，以及个人开始进行嵌入式系统的研究与应用，嵌入式系统设计将是未来相当长一段时间内电子领域研究的热点。下面首先对嵌入式操作系统进行概述。

1.1 嵌入式操作系统简介

随着嵌入式操作系统及嵌入式处理器技术的发展，嵌入式操作系统已经被广泛应用到大量以嵌入式处理器为硬件基础的系统中，常见的嵌入式操作系统有：Linux、Windows CE、Symbian、Palm 和 μC/OS-II 等。

这些操作系统都各有自己强劲的优势，Linux 以其开源的经济优势被广泛应用到很多嵌入式系统中，得到了中小型企业的青睐；Windows CE 有着全球最大的操作系统厂商 Microsoft 强大的技术后盾，得到了越来越多的市场份额；Symbian 操作系统是全球最大的手机研发制造商 NOKIA 的手机操作系统，被广泛应用于高端智能手机上。在将来相当长的一段时间内，将存在几个操作系统并存发展、齐头并进的情况，但是，经过一段时间的角逐，常用的嵌入式设备所采用的操作系统将会集中到其中的 2~3 种。

1.1.1 嵌入式系统的基本概念

业界有多种不同的关于嵌入式系统（Embedded System）的定义，被大多数人所接受的是根据嵌入式系统的特点下的定义：它是“以应用为中心、以计算机技术为基础、软件硬件可裁剪，对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。”该定义强调软硬件可裁剪、专用计算机系统的特点，这也是嵌入式系统与通用计算机平台最为显著的差别。

由于嵌入式的应用太广泛，因此，这里仅给出未来发展空间最为看好的嵌入式系统特点，即嵌入式系统是一类在硬件上采用专用（相对于通用的 X86 来说）的高性能处理器（通常为 32 位），在软件上以一个多任务的操作系统为基础的专用系统。一方面，它与通用的计算机平台有本质的区别（软硬件可裁剪）；另一方面，又与以前的单片机有着本质的区别，因为单片机几乎无法使用移植操作系统，而 32 位嵌入式处理器设备能够很便捷地移植操作系统。

实时嵌入式系统也称为实时系统，它反映了嵌入式系统对时间响应要求较高的特点，

即如果逻辑和时序出现偏差将会引起严重后果。常见的实时系统有两种类型，即软实时系统和硬实时系统，它们各自任务要求如下。

- 软实时系统。系统的宗旨是使各个任务运行得越快越好，但并不要求限定某一任务必须在多长时间内完成。
- 硬实时系统。各任务不仅要执行无误，而且要做到准时，例如：火星车。

在实际应用中，大多数实时系统是以上二者的结合。常见的实时操作系统分为以下 3 类。

- 具有强实时特点的操作系统。系统响应时间在毫秒或者微秒级（如数控机床）；
- 一般实时特点的操作系统。系统响应时间在毫秒到几秒的数量级上（如电子点菜机）；
- 弱实时特点的操作系统。系统响应时间约数十秒以至更长时间（如 MP3 系统）。

下面列出部分实时操作系统所具有的特点。

（1）高效的任务管理。实时操作系统支持多任务、优先级管理和任务调度，其中任务调度是基于优先级的抢占式调度，并采用时间片轮转调度的算法。

（2）快速灵活的任务间通信。实时操作系统的通信机制采用消息队列和管道等技术，有效地保障快速灵活的任务间通信。

（3）高度的可裁剪性。实时操作系统的系统功能可针对需求对软件进行裁剪、调整。

（4）便捷地实现动态链接与部件增量加载。

（5）快速有效地实现中断和异常事件处理。

（6）动态内存管理。

1.1.2　嵌入式系统的内核介绍

（1）内核（Kernel）：多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通信。内核提供的基本服务是任务切换。使用实时内核可以大大简化应用系统的设计的原因在于，实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷，代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，这一块消耗起内存来是相当厉害的。内核本身对 CPU 的占用时间的比例一般是 2%~5%。

单片机一般不能运行实时内核，因为单片机的 RAM 很有限。实时内核通过提供必不可少的系统服务，如信号量管理、邮箱、消息队列、延时等，使得 CPU 的利用更为有效。一旦用户用实时内核做过系统设计，将绝不再想返回到前后台系统。

（2）调度（Scheduler）：这是内核的主要职责之一，决定该轮到哪个任务运行了。多数实时内核是基于优先级调度的。每个任务根据其重要程度的不同被赋予不同的优先级。基于优先级的调度是指 CPU 总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权，就要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

（3）可剥夺型内核：当系统响应时间很重要时，要使用可剥夺型内核。因此，μC/OS-Ⅱ及绝大多数商业上销售的实时内核都是可剥夺型内核。最高优先级的任务一旦就绪，总能得到 CPU 的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了

CPU 的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级高的那个任务开始运行，如图 1-1 所示。

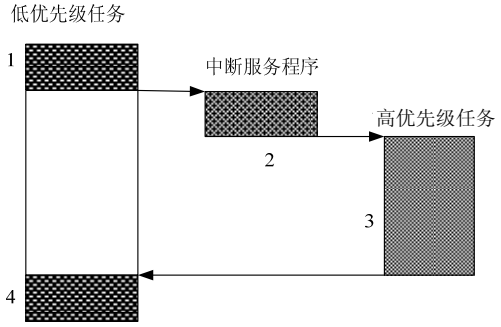


图 1-1　可剥夺型内核任务管理示意图

使用可剥夺型内核，最高优先级的任务什么时候可以执行，可以得到 CPU 的控制权是可知的。使用可剥夺型内核使得任务级响应时间得以最优化。

使用可剥夺型内核时，应用程序不应直接使用不可重入型函数。调用不可重入型函数时，要满足互斥条件，这一点可以用互斥型信号量来实现。当调用不可重入型函数时，低优先级的任务 CPU 的使用权被高优先级任务剥夺，不可重入型函数中的数据将有可能被破坏。由此可见，可剥夺型内核总是让就绪态的高优先级的任务先运行，中断服务程序可以抢占 CPU，到中断服务完成时，内核让此时优先级最高的任务运行（不一定是那个被中断了的任务）。这样，任务级系统响应时间得到了最优化，并且是可知的。μC/OS-Ⅱ属于可剥夺型内核。

（4）死锁（Deadly Embrace）：死锁也称做抱死，指两个任务无限期地互相等待对方控制着的资源。设任务 T1 正独享资源 R1，任务 T2 在独享资源 R2，而此时 T1 又要独享 R2，T2 也要独享 R1，于是哪个任务都没法继续执行了，发生了死锁。最简单的防止发生死锁的方法是让每个任务都具备以下特征。

- 先得到全部需要的资源再做下一步的工作；
- 用同样的顺序去申请多个资源；
- 释放资源时使用相反的顺序。

内核大多允许用户在申请信号量时定义等待超时，以此化解死锁。当等待时间超过了某一确定值，信号量还是无效状态，就会返回某种形式的出现超时错误的代码，这个出错代码告知该任务，不是得到了资源使用权，而是系统错误。死锁一般发生在大型多任务系统中，在嵌入式系统中不易出现。

1.1.3　嵌入式系统的应用领域

嵌入式系统的应用很广泛，可以这样说，除了通用的计算机系统应用外，其他所有的智能电子设备都属于嵌入式系统。以下简要列出嵌入式系统在数字家电、个人数据处理、通用技术等领域的应用。

嵌入式系统在数字家电领域的应用。当前家庭电子设备中包含了越来越多的嵌入式处理器产品，以下电子设备都属于嵌入式系统：IP 电话、PDA、无线音频系统、在线游戏、

无线接入设备、微波炉、电冰箱、洗衣机、电视、收音机、CD 播放器、个人电脑、遥控开关。一般来说，每个家庭拥有至少多于 20 种以上的电子设备。据有关调查表明，预计 2010 年后每个城市家庭都将基本实现电子化，那时每个人所使用的全部设备中包含的 MCU 数量将会超过 100 个！

基于嵌入式的多用途 PDA（Personal Digital Assistant）解决方案。随着各行各业对信息化要求的日益提高，行业用户对 PDA 的需求量越来越大，同时对 PDA 的功能要求也越来越高，而且不同的行业用户所需的功能要求也不同，将呈现多元化和个性化趋势。采用嵌入式系统的智能化多功能 PDA 终端平台可以提供个性化定制业务，可定制与所在行业相对应的专用功能，并可适应不同环境下的功能要求。基于嵌入式的多用途 PDA 可以广泛运用于军警用设备、信息查询、服务行业、石油、地质、电力、水利、GPRS 应用（上网、通话、短信等）、GPS 定位、指纹识别技术、GIS（地理信息系统）应用、IC 卡应用、蓝牙技术、CCD 摄像处理、条形码识别、红外、USB 传输、多媒体、MP3 等领域。

嵌入式系统在通信领域的应用。根据市场调查，基于 ARM 处理器的嵌入式系统在 GSM/UMTS 市场（GSM850、900、1800、1900、GPRS、EDGE、UMTS）中将超过 85%的市场占有率，并主要为 OEM（Original Equipment Manufacturer）客户，在 CDMA 系统（IS95A/B、CDMA2000 1X、EV-DO、BREW 等）中将超过 99%的市场占有率，在 Bluetooth 系统中将超过 75%的市场占有率。

1.2　Linux 操作系统概述

1.2.1　嵌入式 Linux 发展现状

UNIX 操作系统于 1969 年由 Ken Thompson 在 AT&T 贝尔实验室的一台 DEC PDP-7 计算机上实现。后来 Ken Thompson 和 Dennis Ritchie 使用 C 语言对整个系统进行了再加工和编写，使得 UNIX 能够很容易地移植到其他硬件的计算机上。由于此时 AT&T 还没有把 UNIX 作为它的正式商品，因此研究人员只是在实验室内部使用并完善它。也正是由于 UNIX 被作为研究项目，其他科研机构和大学的计算机研究人员也希望能得到这个系统，以便进行自己的研究。AT&T 以分发许可证的方法，对 UNIX 仅仅收取很少的费用，UNIX 的源代码就被散发到各个大学，使得科研人员能够根据需要改进系统，或者将其移植到其他的硬件环境中去；另一方面，也培养了大量懂得 UNIX、使用 UNIX 和编程的学生，这使得 UNIX 的普及更为广泛。

到了 20 世纪 70 年代末，在 UNIX 发展到了版本 6 之后，AT&T 认识到了 UNIX 的价值，成立了 UNIX 系统实验室（UNIX System Lab，USL）来继续发展 UNIX。此时，AT&T 一方面继续发展内部使用的 UNIX 版本 7，一方面由 USL 开发对外正式发行的 UNIX 版本，同时 AT&T 也宣布对 UNIX 产品拥有所有权。几乎同时，加州大学伯克利分校计算机系统研究小组（CSRG）使用 UNIX 对操作系统进行研究，他们对 UNIX 的改进相当多，增加了很多当时非常先进的特性，包括更好的内存管理，快速且健壮的文件系统等，大部分原有的源代码都被重新写过，很多其他 UNIX 使用者，包括其他大学和商业机构，都希望能得到 CSRG 改进的 UNIX 系统。也正因为如此，CSRG 中的研究人员把他们的 UNIX 组成一个完整的 UNIX 系统——BSD UNIX（Berkeley Software Distribution）向外发行。

而 AT&T 的 UNIX 系统实验室同时也在不断改进它们的商用 UNIX 版本，直到它们吸收了 BSD UNIX 中已有的各种先进特性，并结合其本身的特点，推出了 UNIX System V 版本。从此以后，BSD UNIX 和 UNIX System V 形成了当今 UNIX 的两大主流，目前的 UNIX 版本大部分都是这两个版本的衍生产品：IBM 的 AIX4.0、HP/UX11、SCO 的 UNIXWare 等属于 System V，而 Minix、freeBSD、NetBSD、OpenBSD 等属于 BSD UNIX。

Linux 从一开始，就决定自由扩散 Linux，包括源代码也发布在网上，随即就引起爱好者的注意，他们通过互联网也加入了 Linux 的内核开发工作，一大批高水平程序员的加入，使得 Linux 得到迅猛发展。1993 年年底，Linux 1.0 终于诞生。Linux 1.0 已经是一个功能完备的操作系统了，其内核写得紧凑高效，可以充分发挥硬件的性能，在 4MB 内存的 80386 机器上也表现得非常好。

Linux 加入 GNU 并遵循公共版权许可证（GPL）。由于不排斥商家对自由软件的进一步开发，不排斥在 Linux 上开发商业软件，故而使 Linux 又开始了一次飞跃，出现了很多的 Linux 发行版，如 Slackware、Redhat、TurboLinux、OpenLinux 等 10 多种，而且还在增加，还有一些公司在 Linux 上开发商业软件或把其他 UNIX 平台的软件移植到 Linux 上来。如今很多 IT 界的大腕，如 IBM、Intel、Oracle、Infomix、Sysbase、Netscape、Novell 等都宣布支持 Linux。商家的加盟弥补了纯自由软件的不足和发展障碍，Linux 得以迅速普及。

Linux 由 UNIX 操作系统的发展而来，它的内核由 Linus Torvalds 及网络上组织松散的黑客队伍一起从零开始编写而成。Linux 的目标是保持和 POSIX 的兼容。Linux 操作系统具有以下特点。

- Linux 具备现代一切功能完整的 UNIX 系统所具备的全部特征，其中包括真正的多任务、虚拟内存、共享库、需求装载、共享的写时复制程序执行、优秀的内存管理，以及 TCP/IP 网络支持等。
- Linux 的发行遵守 GNU 的通用公共许可证（GPL）。
- 在原代码级上兼容绝大部分的 UNIX 标准（如 IEEE POSIX、System V、BSD），它遵从 POSIX 规范。读者可以在 http://www.linuxresources.com/what.html 和 http://www.linux.org 得到更多的信息。

1.2.2 Linux 相关的常用术语

1．POSIX 及其重要地位

POSIX 表示可移植操作系统接口（Portable Operating System Interface，POSIX）。由电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）开发，主要为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX，许多其他的操作系统例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准，尤其是 IEEE STD.1003.1-1990（1995 年修订）或 POSIX.1。POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC9945- 1:1990 标准。现在 POSIX 已经发展成为一个非常庞大的标准族，某些部分正处在开发过程中。

2．GNU 和 Linux 的关系

GNU 是 GNU Is Not UNIX 的递归缩写，是自由软件基金会的一个项目，该项目的目

标是开发一个自由的 UNIX 版本，这一 UNIX 版本称为 HURD。尽管 HURD 尚未完成，但 GNU 项目已经开发了许多高质量的编程工具，包括 emacs 编辑器、著名的 GNU C 和 C++编译器（gcc 和 g++），这些编译器可以在任何计算机系统上运行。所有的 GNU 软件和派生工作均适用 GNU 通用公共许可证，即 GPL。GPL 允许软件作者拥有软件版权，但同时授予其他任何人以合法复制、发行和修改软件的权利。

Linux 的开发使用了许多 GNU 工具。Linux 系统上用于实现 POSIX.2 标准的工具几乎都是 GNU 项目开发的，如 Linux 内核、GNU 工具，以及其他一些由软件组成的人们常说的 Linux——C 语言编译器和其他开发工具及函数库、X Window 窗口系统、各种应用软件（包括字处理软件、图像处理软件等）、其他各种 Internet 软件（包括 FTP 服务器、WWW 服务器）、关系数据库管理系统等。

3．GPL（General Public License）公共许可协议

GPL 的文本保存在 Linux 系统的不同目录下的命名为 COPYING 的文件里。例如，键入 cd/usr/doc/ghostscript*然后再键入 more COPYING 可查看 GPL 的内容。GPL 与软件是否免费无关，它的主要目标是保证软件对所有的用户来说是自由的。GPL 通过如下途径实现这一目标。

- 要求软件以源代码的形式发布，并规定任何用户能够以源代码的形式将软件复制或发布给别的用户。
- 提醒每个用户，对于该软件不提供任何形式的担保。
- 如果用户的软件使用了受 GPL 保护的任何软件的一部分，那么该软件就成为 GPL 软件，也就是说必须随应用程序一起发布源代码。
- GPL 并不排斥对自由软件进行商业性质的包装和发行，也不限制在自由软件的基础上打包发行其他非自由软件。
- 遵照 GPL 的软件并不是可以任意传播的，这些软件通常都有正式的版权，GPL 在发布软件或者复制软件时声明限制条件。但是，从用户的角度考虑，这些根本不能算是限制条件，相反用户只会从中受益，因为用户可以确保获得源代码。

尽管 Linux 内核也属于 GPL 范畴，但 GPL 并不适用于通过系统调用而使用内核服务的应用程序，通常把这种应用程序看做是内核的正常使用。假如准备以二进制的形式发布应用程序（像大多数商业软件那样），则必须确保自己的程序未使用 GPL 保护的任何软件。如果软件通过库函数调用而使用了别的软件，则不必受到这一限制。大多数函数库受另一种 GNU 公共许可证即 LGPL 的保护，将在下面介绍。

4．LGPL（Libraray General Public License）程序库公共许可证

GNU LGPL 的内容全部包括在命名为 COPYING.LIB 的文件中。如果安装了内核的源程序，在任意一个源程序的目录下都可以找到 COPYING.LIB 文件的一个拷贝。

即使在不公开自己源程序的情况下，LGPL 也允许在自己的应用程序中使用程序库。但是，LGPL 还规定用户必须能够获得在应用程序中使用的程序库的源代码，并且允许用户对这些程序库进行修改。

由于大多数 Linux 程序库，包括 C 程序库（libc.a）都属于 LGPL 范畴，因此，如果在 Linux 环境下使用 GCC 编译器建立自己的应用程序，程序所链接的多数程序库是受 LGPL 保护的。如果想以二进制的形式发布自己的应用程序，则必须注意遵循 LGPL 的有关规定。

遵循 LGPL 的一种方法是随应用程序一起发布目标代码，并可以发布将这些目标程序与受 LGPL 保护的、更新的 Linux 程序库链接起来的 makefile 文件。

遵循 LGPL 的比较好的一种方法是使用动态链接。使用动态链接时，即使程序在运行中调用函数库中的函数时，应用程序本身和函数库也是不同的实体。通过动态链接，用户可以直接使用更新后的函数库，而不用对应用程序进行重新链接。

在 GPL 的保护范围以外，也有 GNU dbm 和 GNU bison 的相应的替代程序。例如，对于数据库类的程序库，可以使用 Berkeley 数据库 db 来替代 gdbm，对于分析器生成器，可以使用 yacc 来替代 bison。

1.3 Linux 操作系统的移植

Linux 操作系统是一个完全开源的操作系统，用户可以自己下载、阅读、修改并重新编译内核，从而使开发人员能够完全自己定制相关的操作系统功能以适合自己的需要。本节将就以下内容作详细介绍。

BootLoader 程序：BootLoader 是一个用来初始化嵌入式硬件最小系统，进而引导操作系统的底层程序，其主要代码由汇编语言和 C 程序编写。在 X86 上常见的 BootLoader 有 GRUB 和 LILO，在嵌入式设备中 U-boot 和 VIVI 用得比较多。

Linux 源代码分开，读者可以在相关网站上下载这些源代码。随着 Linux 的发展，目前 2.6 版内核的 Linux 源代码已经超过 30MB。1.3.2 小节将详细介绍 Linux 源代码目录结构，从而为读者快速阅读 Linux 内核程序提供参考。

1.3.3 小节将详细介绍如何重新编译适合嵌入式 ARM 处理的 Linux 内核程序的过程，主要包括如何剪裁 Linux 内核源程序。

读者通过对本节的学习，将对 Linux 内核源代码有一个比较清楚的认识，能够独立裁剪 Linux 内核，并移植 Linux 内核到 ARM 处理器中运行。

1.3.1 BootLoader 技术详解

在专用的嵌入式板子上运行 GNU/Linux 系统已经变得越来越流行。一个嵌入式系统从软件角度大致可以分为四个层次（注：有些系统中并没有严格划分），如图 1-2 所示。

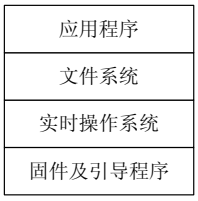


图 1-2　嵌入式系统软件层次

（1）引导加载程序。包括固化在固件（firmware）中的 boot 代码（可选，并不是所有系统都有）和 BootLoader 程序两个部分。

（2）实时操作系统。特定嵌入式板子的定制内核以及内核的启动参数。

（3）文件系统。包括根文件系统和建立于 Flash 内存设备之上的文件系统。通常有 RAMFS、ROMFS、JAFFS、YAFFS 等文件系统。

（4）上层用户应用程序。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有 QT 和 MiniGUI。

引导加载程序是系统加电后运行的第一段软件代码。例如 X86 体系结构中，系统的引导加载程序由 BIOS（其本质就是一段固件程序）和位于硬盘 MBR 中的 OS BootLoader（比如 LILO 和 GRUB 等）一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的 BootLoader 读到系统的 RAM 中，然后将控制权交给 OS BootLoader。BootLoader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，即开始启动操作系统。而在嵌入式系统中，通常并没有像 BIOS 那样的固件程序（注：有的嵌入式处理器也会内嵌一段短小的启动程序），因此整个系统的加载启动任务就完全由 BootLoader 来完成。

1．BootLoader 概述

BootLoader 是在嵌入式实时操作系统内核启动之前运行的一段小程序。其主要完成初始化硬件设备，建立内存空间的映射图，设置系统的软硬件环境，为最终调用操作系统内核准备好正确的环境。

通常 BootLoader 是严重依赖于硬件而实现的，因此，建立一个通用的 BootLoader 几乎是不可能的。每种不同的 CPU 体系结构都有不同的 BootLoader，部分 BootLoader 也支持多种体系结构的 CPU，比如 U-Boot 可以支持 ARM 体系结构和 MIPS 体系结构。

除了依赖于 CPU 的体系结构外，BootLoader 同样依赖于具体的嵌入式板级设备的配置。因此，即使基于同一种 CPU 而构建的嵌入式硬件环境，要想让 BootLoader 程序互换使用，通常也都需要修改 BootLoader 源程序。

系统加电或复位后，所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备（比如 ROM、EEPROM 或 Flash 等）被映射到这个预先安排的地址上。因此在系统加电后，CPU 将首先执行 BootLoader 程序。图 1-3 为一个固态存储设备的典型空间分配结构图。

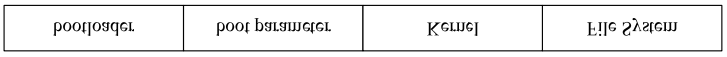


图 1-3　固态存储设备的典型空间分配结构

大多数 BootLoader 都包含两种不同的操作模式：“启动加载”模式和“下载”模式。

启动加载（Bootloading）模式：这种模式也称为“自主”模式。BootLoader 从目标机上的某个固态存储设备上将操作系统加载到 RAM 中运行，整个过程并没有用户的介入。这种模式是 BootLoader 的正常工作模式，在嵌入式产品发布时，BootLoader 必须工作在这种模式下。

下载（Downloading）模式：在这种模式下，目标机上的 BootLoader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件，比如下载内核映像和根文件系统映像等。从主机下载的文件通常首先被 BootLoader 保存到目标机的 RAM 中，然后再被 BootLoader 写到目标机上的 Flash 类固态存储设备中。BootLoader 的这种模式通常在第一次安装内核与根文件系统时被使用；此外，以后的系统更新也会使用 BootLoader 的这种工作模式。工作于这种模式下的 BootLoader 通常都会向它的终端用户提供一个简单的命令行接口。

一般的 BootLoader 通常同时支持这两种工作模式，而且允许用户在这两种工作模式之

间进行切换。比如，VIVI 在启动时处于正常的启动加载模式，但是它会延时几秒等待终端用户按下任意键而将 VIVI 切换到下载模式。如果在等待时间内没有用户按键，则继续启动 Linux 内核。

BootLoader 的启动过程有单阶段（Single 阶段）和多阶段（Multi-阶段）两种，通常多阶段的 BootLoader 能提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 BootLoader 大多都是 2 阶段的启动过程，即启动过程可以分为阶段 1 和阶段 2 两部分。

2．BootLoader 运行流程

从操作系统的角度看，BootLoader 的主要作用就是正确加载操作系统内核。大多数 BootLoader 都分为阶段 1 和阶段 2 两大部分。依赖于 CPU 体系结构的代码通常都放在阶段 1 中，而且通常用汇编语言来实现，以达到短小精悍的目的。阶段 2 部分通常用 C 语言来实现，这样可以实现复杂的功能，而且代码会具有更好的可读性和可移植性。

BootLoader 的阶段 1 通常包括以下步骤（以执行的先后顺序）。

- 硬件设备初始化。
- 为加载 BootLoader 的阶段 2 准备 RAM 空间。
- 将 BootLoader 的阶段 2 复制到 RAM 空间中。
- 设置堆栈空间。
- 跳转到阶段 2 的 C 入口点。

BootLoader 的阶段 2 通常包括以下步骤（以执行的先后顺序）。

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射情况。
- 将 kernel 映像和根文件系统映像从 Flash 上读到 RAM 空间中。
- 为内核设置启动参数。
- 调用内核。

（1）阶段 1 启动流程

- 基本的硬件初始化：这是 BootLoader 一开始就执行的操作，其目的是为阶段 2 的执行及 kernel 的执行准备好一些基本的硬件环境。它通常包括以下步骤。
  - 屏蔽所有的外部中断。为中断提供服务通常是 OS 设备驱动程序的责任，因此在 BootLoader 的执行全过程中可以不必响应任何外部中断。中断屏蔽可以通过设置 CPU 的中断屏蔽寄存器或状态寄存器（比如 ARM 的 CPSR 寄存器）来完成。
  - 设置 CPU 的速度和时钟频率。
  - RAM 空间初始化。包括正确地设置系统的内存控制器的功能寄存器及各内存库控制寄存器等。
  - 初始化某一简单的典型外部接口，表明系统运行正常，通常通过初始化 UART 向串口打印 BootLoader 的信息。另外，可以用 GPIO 来驱动 LED，其目的是表明系统的状态是 OK 还是 Error。
  - 关闭 CPU 内部指令/数据高速缓存（cache）。
- 为加载阶段 2 准备 RAM 空间：为了获得更快的执行速度，通常把阶段 2 加载到 RAM 空间中来执行，因此必须为加载 BootLoader 的阶段 2 准备好一段可用的 RAM 空间范围。由于阶段 2 通常是 C 语言执行代码，因此在考虑空间大小时，除了阶段 2

可执行映像文件的大小外，还必须把堆栈空间也考虑进来。

此外，空间大小最好是 memorypage 大小的倍数。一般而言，1MB 的 RAM 空间已经足够了。具体的地址范围可以任意安排，比如 blob 就将它的阶段 2 可执行映像安排到从系统 RAM 起始地址 0xc0200000 开始的 1MB 空间内执行。但是，将阶段 2 安排到整个 RAM 空间的最顶 1MB（即 RamEnd-1MB）是一种值得推荐的方法。

- 将阶段 2 代码复制到 RAM 中。复制时要确定两点：（1）阶段 2 的可执行映像在固态存储设备的存放起始地址和终止地址；（2）RAM 空间的起始地址。
- 设置堆栈指针 SP：堆栈指针的设置是为了执行 C 语言代码作好准备。通常可以把 SP 的值设置为阶段 2_end-4。经过上述这些执行步骤后，系统的物理内存布局应该如图 1-4 所示。

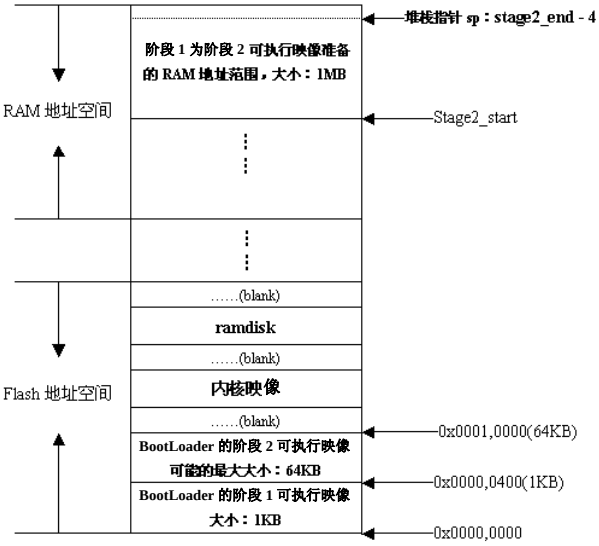


图 1-4　物理内存布局

- 跳转到阶段 2 的 C 入口点：在上述一切都就绪后，就可以跳转到 BootLoader 的阶段 2 去执行了。

（2）阶段 2 启动流程

阶段 2 的代码通常用 C 语言来实现，以便于实现更复杂的功能，并取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是，在编译和链接 BootLoader 这样的程序时，不能使用 glibc 库中的任何支持函数。这就带来一个问题，那就是从哪里跳转进 main()函数呢？直接把 main()函数的起始地址作为整个阶段 2 执行映像的入口点或许是最直接的想法。但是这样做有两个缺点。

- 无法通过 main()函数传递函数参数；
- 无法处理 main()函数返回的情况。

一种更为巧妙的方法是利用汇编语言写一段启动代码作为阶段 2 可执行映像的执行入口点。然后用 CPU 跳转指令跳入 main()函数中去执行。

初始化本阶段要使用到的硬件设备通常包括：

- 初始化至少一个串口，以便和终端用户进行 I/O 输出信息；

- 初始化计时器等。

- 检测系统的内存映射（memorymap）

所谓内存映射就是指在整个 4GB 物理地址空间中有哪些地址范围被分配用来寻址系统的 RAM 单元。

可以用如下数据结构来描述 RAM 地址空间中的一段连续（continuous）的地址范围。

```
typedefstructmemory_area_struct{
u32start;           /* the base address of the memory region */
u32 size;           /* the byte number of the memory region */
int used;
} memory_area_t;
```

这段 RAM 地址空间中的连续地址范围可以处于下面两种状态之一。

- used=1，说明这段连续的地址范围已被实现，即真正地被映射到 RAM 单元上。
- used=0，说明这段连续的地址范围并未被系统所实现，而是处于未使用状态。

基于上述 memory_area_t 数据结构，整个 CPU 预留的 RAM 地址空间可以用一个 memory_area_t 类型的数组来表示，如下所示。

```
memory_area_t memory_map[NUM_MEM_AREAS] =
{
[0 ... (NUM_MEM_AREAS - 1)] =
{
.start = 0,
.size = 0,
.used = 0
},
};
```

下面给出一个可用来检测整个 RAM 地址空间内存映射情况的简单而有效的算法。

```
/*数组初始化*/
for(i=0;i<NUM_MEM_AREAS;i++)
memory_map[i].used=0;
/* first write a 0 to all memory locations */
for(addr=MEM_START;addr<MEM_END;addr+=PAGE_SIZE)
*(u32*)addr=0;
for(i=0,addr=MEM_START;addr<MEM_END;addr+=PAGE_SIZE){
/*
*检测从基地址 MEM_START+i*PAGE_SIZE 开始,大小为
*PAGE_SIZE 的地址空间是否是有效的 RAM 地址空间。
*/
```

调用 3.1.2 节中的算法 test_mempage()。

```
if(currentmemory page isnot a valid ram page)
{
/*noRAM here */
if(memory_map[i].used )
    i++;
continue;
}
```

```
/*当前页已经是一个被映射到 RAM 的有效地址范围* 但是还要看看当前页是否只是 4GB 地址*//*空间中某个地址页的别名？*/
```

```
if(* (u32 *)addr != 0) { /* alias? */
/* 这个内存页是 4GB 地址空间中某个地址页的别名 */
if ( memory_map[i].used )
i++;
continue;
}

/*
* 当前页已经是一个被映射到 RAM 的有效地址范围
* 而且它也不是 4GB 地址空间中某个地址页的别名。
*/
if (memory_map[i].used == 0) {
memory_map[i].start = addr;
memory_map[i].size = PAGE_SIZE;
memory_map[i].used = 1;
} else {
memory_map[i].size += PAGE_SIZE;
}
}   /* end of for (…) */
```

在用上述算法检测完系统的内存映射情况后，Boot Loader 也可以将内存映射的详细信息打印到串口。

- 加载内核映像和根文件系统映像

规划内存占用的布局包括两个方面：（1）内核映像所占用的内存范围；（2）根文件系统所占用的内存范围。在规划内存占用的布局时，主要考虑基地址和映像的大小两方面。对于内核映像，一般将其复制到从（MEM_START＋0x8000）这个基地址开始的大约 1MB 的内存范围内（嵌入式 Linux 的内核一般都不超过 1MB）。为什么要把从 MEM_START 到 MEM_START＋0x8000 这段 32KB 大小的内存空出来呢？这是因为 Linux 内核要在这段内存中放置一些全局数据结构，如启动参数和内核页表等信息。而对于根文件系统映像，则一般将其复制到 MEM_START+0x0010,0000 开始的地方。如果用 Ramdisk 作为根文件系统映像，则其解压后的大小一般是 1MB。

由于像 ARM 这样的嵌入式 CPU 通常都是在统一的内存地址空间中寻址 Flash 等固态存储设备的，因此从 Flash 上读取数据与从 RAM 单元中读取数据并没有什么不同。用一个简单的循环就可以完成从 Flash 设备上复制映像的工作。

```
while(count) {
*dest++ = *src++;       /* they are all aligned with word boundary */
count -= 4;             /* byte number */
};
```

- 设置内核的启动参数

应该说在将内核映像和根文件系统映像复制到 RAM 空间后，就可以准备启动 Linux 内核了。但是在调用内核之前，应该做一步准备工作，即设置 Linux 内核的启动参数。Linux 2.4.x 以后的内核都期望以标记列表（tagged list）的形式来传递启动参数。启动参数标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。每个标记由标识被传递参数的 tag_header 结构及随后的参数值数据结构组成。数据结构 tag 和 tag_header 定义在 Linux 内核源代码的 include/asm/setup.h 头文件中。

```
/* The list ends with an ATAG_NONE node. */
```

```
#define ATAG_NONE 0x00000000
struct tag_header {
u32 size;                    /* 注意，这里 size 是以字数为单位的 */
u32 tag;
};
……
struct tag {
struct tag_header hdr;
union {
    struct tag_core core;
    struct tag_mem32 mem;
    struct tag_videotext videotext;
    struct tag_ramdisk ramdisk;
    struct tag_initrd initrd;
    struct tag_serialnr serialnr;
    struct tag_revision revision;
    struct tag_videolfb videolfb;
    struct tag_cmdline cmdline;
            /*  Acorn specific */
    struct tag_acorn acorn;
        /*  DC21285 specific */
struct tag_memclk memclk;
} u;
};
```

在嵌入式Linux系统中，通常需要由Boot Loader设置的常见启动参数有ATAG_CORE、ATAG_MEM、ATAG_CMDLINE、ATAG_RAMDISK、ATAG_INITRD 等。

比如，设置 ATAG_CORE 的代码如下。

```
params = (struct tag *)BOOT_PARAMS;
params->hdr.tag = ATAG_CORE;
params->hdr.size = tag_size(tag_core);
params->u.core.flags = 0;
params->u.core.pagesize = 0;
params->u.core.rootdev = 0;
params = tag_next(params);
```

其中，BOOT_PARAMS 表示内核启动参数在内存中的起始基地址，指针 params 是一个 struct tag 类型的指针。宏 tag_next()将以指向当前标记的指针为参数，计算紧邻当前标记的下一个标记的起始地址。注意，内核的根文件系统所在的设备 ID 就是在这里设置的。下面是设置内存映射情况的示例代码。

```
for(i = 0; i < NUM_MEM_AREAS; i++) {
if(memory_map[i].used) {
params->hdr.tag = ATAG_MEM;
params->hdr.size = tag_size(tag_mem32);
params->u.mem.start = memory_map[i].start;
params->u.mem.size = memory_map[i].size;
params = tag_next(params);
}
}
```

可以看出，在 memory_map [ ] 数组中，每一个有效的内存段都对应一个 ATAG_MEM 参数标记。Linux 内核在启动时可以以命令行参数的形式来接收信息，利用这一点可以向内核提供那些内核不能自己检测的硬件参数信息，或者重载（override）内核自己检测到的

信息。比如，用这样一个命令行参数字符串“console=ttyS0,115200n8”来通知内核以 ttyS0 作为控制台，且串口采用“115200bps、无奇偶校验、8 位数据位”这样的设置。下面是一段设置调用内核命令行参数字符串的示例代码。

```
char *p;
/* eat leading white space */
for(p = commandline; *p == ' '; p++)
;
/* skip non-existent command lines so the kernel will still
* use its default command line.
*/
if(*p == '\0')
return;
params->hdr.tag = ATAG_CMDLINE;
params->hdr.size = (sizeof(struct tag_header) + strlen(p) + 1 + 4) >> 2;
strcpy(params->u.cmdline.cmdline, p);
params = tag_next(params);
```

下面是设置 ATAG_INITRD 的示例代码，它告诉内核在 RAM 中的什么地方可以找到 initrd 映像（压缩格式）及它的大小。

```
params->hdr.tag = ATAG_INITRD2;
params->hdr.size = tag_size(tag_initrd);
params->u.initrd.start = RAMDISK_RAM_BASE;
params->u.initrd.size = INITRD_LEN;
params = tag_next(params);
```

下面是设置 ATAG_RAMDISK 的示例代码，它告诉内核解压后的 Ramdisk 有多大（单位是 KB）。

```
params->hdr.tag = ATAG_RAMDISK;
params->hdr.size = tag_size(tag_ramdisk);
params->u.ramdisk.start = 0;
params->u.ramdisk.size = RAMDISK_SIZE; /* 请注意，单位是 KB */
params->u.ramdisk.flags = 1; /* automatically load ramdisk */
params = tag_next(params);
```

最后，设置 ATAG_NONE 标记，结束整个启动参数列表。

```
static void setup_end_tag(void)
{
params->hdr.tag = ATAG_NONE;
params->hdr.size = 0;
}
```

- 调用内核

Boot Loader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处，即直接跳转到 MEM_START＋0x8000 地址处。

3．常见 BootLoader 程序

（1）VIVI

VIVI 是专用于 ARM9 处理器的 BootLoader 程序，其主要目录结构如下。

```
drwxr-xr-x      5 root      root        4096 2005-08-12  arch
-rw-r--r--      1 root      root       18008 2004-08-05  COPYING
```

```
drwxr-xr-x      2 root      root        4096 2005-08-12  CVS
drwxr-xr-x      4 root      root        4096 2005-08-12  Documentation
drwxr-xr-x      5 root      root        4096 2005-08-12  drivers
drwxr-xr-x      6 root      root        4096 2005-08-12  include
drwxr-xr-x      3 root      root        4096 2005-08-12  init
drwxr-xr-x      4 root      root        4096 2005-08-12  lib
-rw-r--r--      1 root      root        5680 2004-08-05  Makefile
-rw-r--r--      1 root      root        5547 2004-08-05  Makefile.newSDK
-rw-r--r--      1 root      root        4348 2004-08-05  Rules.make
drwxr-xr-x      4 root      root        4096 2005-08-12  scripts
```

其中各目录内容如下：

arch 文件夹主要包括 vivi 所支持的硬件目标开发板，如 S3C2410；

drivers 文件夹主要包括引导内核所需要的外部设备驱动程序，其下有 MTD 文件夹，为 MTD 设备的驱动程序。

init 文件夹为 main.c 文件所在位置，这是程序的主函数入口。

lib 文件夹是一些公共平台的接口代码。

include 文件夹为所有头文件位置。

（2）U-Boot

U-Boot 是在 PPC-Boot 的基础上发展起来的一个开源的嵌入式 BootLoader 程序，其理念是成为嵌入式设备标准的 BootLoader 程序，其主要目录结构如下。

```
[root@yangzongde u-boot-1.1.1]# ls -l
总用量 1276
drwxr-xr-x  139 yangzongde yangzongde  4096 2004-08-31    board
-rw-r--r--  1 yangzongde yangzongde    82401 2002-08-15   CHANGELOG
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    command
-rw-r--r--  1 yangzongde yangzongde    5096 2002-08-15    config.mk
-rw-r--r--  1 yangzongde yangzongde    15127 2002-08-15   COPYING
drwxr-xr-x  25 yangzongde yangzongde   4096 2002-08-15    cpu
-rw-r--r--  1 yangzongde yangzongde    8122 2002-08-15    CREDITS
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    disk
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    doc
drwxr-xr-x  3 yangzongde yangzongde    4096 2005-06-28    drivers
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    dtt
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    examples
drwxr-xr-x  7 yangzongde yangzongde    4096 2002-08-15    fs
-rw-r--r--  1 yangzongde yangzongde    910 2002-08-15     i386_config.mk
drwxr-xr-x  16 yangzongde yangzongde   4096 2005-06-28    include
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    lib_arm
drwxr-xr-x  2 yangzongde yangzongde    4096 2005-06-28    lib_generic
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_i386
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_m68k
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_microblaze
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_mips
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_nios
drwxr-xr-x  2 yangzongde yangzongde    4096 2002-08-15    lib_ppc
-rw-r--r--  1 yangzongde yangzongde    934 2002-08-15     m68k_config.mk
-rw-r--r--  1 yangzongde yangzongde    7321 2002-08-15    MAINTAINERS
-rwxr-xr-x  1 yangzongde yangzongde    6098 2002-08-15    MAKEALL
-rw-r--r--  1 yangzongde yangzongde    39112 2005-06-27   Makefile
-rw-r--r--  1 yangzongde yangzongde    905 2002-08-15     mips_config.mk
-rwxr-xr-x  1 yangzongde yangzongde    1130 2002-08-15    mkconfig
```

```
drwxr-xr-x  2 yangzongde yangzongde     4096 2005-06-28     net
-rw-r--r--  1 yangzongde yangzongde     939 2002-08-15      nios_config.mk
drwxr-xr-x  3 yangzongde yangzongde     4096 2005-06-28     post
-rw-r--r--  1 yangzongde yangzongde     936 2002-08-15      ppc_config.mk
-rw-r--r--  1 yangzongde yangzongde     112404 2002-08-15   README
drwxr-xr-x  2 yangzongde yangzongde     4096 2005-06-28     rtc
-rw-r--r--  1 root    root              13934 2005-06-28    System.map
drwxr-xr-x  9 yangzongde yangzongde     4096 2005-06-28     tools
```

其中：

board 文件夹主要存放的是 U-Boot 所支持的目标板的子目录，也就是相应的硬件处理器分类，另外还需要修改 Flash.c 文件。

cpu 文件主要存入的是 u-boot 所支持的 CPU 类型，此文件夹下主要是一段初始可执行环境，包括中断处理等基本设置。其 start.S 文件是可执行代码的第一阶段代码。

command 文件夹存入的是一些公共命令的实现，也就是说，用户进入到 u-boot 后可以输入运行的命令全部在此文件夹下。

drivers 文件夹主要存入的是一些外部设备接口的驱动程序；

fs 文件夹为文件系统相关的源代码文件。

1.3.2　Linux 内核基本结构

Linux 内核是一个 Linux 操作系统的核心。它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。Linux 的一个重要的特点就是其源代码的公开性，所有的内核源程序都可以在/usr/src/linux 目录下找到，大部分应用软件也都是遵循 GPL 而设计的，读者都可以获取相应的源程序代码。

任何一个软件工程师都可以将自己认为优秀的代码加入到其中，由此引发的一个明显的好处就是 Linux 修补漏洞的快速及对最新软件技术的利用。而 Linux 的内核则是这些特点的最直接的代表。拥有了内核的源程序可以让读者了解系统是如何工作的，通过通读源代码，读者可以了解系统的工作原理，另外可以针对自己的需要定制适合自己的系统，这是经常提到的重新编译内核工作。

Linux 作为一个自由软件，在广大爱好者的支持下，内核版本不断更新。新的内核修订了旧内核的缺陷，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统量身定制一个更高效、更稳定的内核，就需要重新编译内核。通常更新的内核会支持更多的硬件，具备更好的进程管理能力，运行速度更快、更稳定，并且一般会修复老版本中发现的许多漏洞等，经常性选择升级更新的系统内核是 Linux 使用者的必要操作内容。为了正确合理地设置内核编译配置选项，从而只编译系统需要的功能的代码，一般主要有如下四方面的考虑。

- 自己定制编译的内核运行更快（具有更少的代码）。
- 系统将拥有更多的内存（内核部分将不会被交换到虚拟内存中）。
- 不需要的功能编译进入内核可能会增加被系统攻击者利用的漏洞。
- 将某种功能编译为模块方式会比编译到内核内的方式速度要慢一些。

要增加对某部分功能的支持，比如网络之类，可以把相应部分编译到内核中（build-in），也可以把该部分编译成模块（module），实现动态调用。如果编译到内核中，在内核启动时就可以自动支持相应部分的功能，这样的优点是方便、速度快，机器一启动就可以使用

这部分功能；缺点是会使内核变得庞大起来，不管是否需要这部分功能，它都会存在。如果编译成模块，就会生成对应的.o 文件，在使用的时候可以动态加载，优点是不会使内核过分庞大，缺点是需要每次自己来调用这些模块。

1．Linux 内核结构

Linux 核心源程序通常都安装在/usr/src/linux 目录下，都是一个稳定地发行的核心，而任何奇数核心源程序的文件按树形结构进行组织，在源程序树的最上层。目录/usr/src/linux 下有这样一些目录和文件。

（1）COPYING：GPL 版权声明。对具有 GPL 版权的源代码改动而形成的程序，或使用 GPL 工具产生的程序，具有使用 GPL 发表的义务，如公开源代码。

（2）CREDITS：光荣榜。对 Linux 做出过很大贡献的一些人的信息。

（3）MAINTAINERS：维护人员列表，对当前版本的内核各部分都由谁负责。

（4）Makefile：用来组织内核的各模块，记录了模块间的相互联系和依托关系，编译时使用；仔细阅读各子目录下的 Makefile 文件对弄清各个文件之间的联系和依托关系很有帮助。

（5）ReadMe：核心及其编译配置方法的简单介绍。

（6）Rules.make：各种 Makefilemake 所使用的一些共同规则。

（7）REPORTING-BUGS：有关报告 Bug 的一些内容。

（8）Arch/：arch 子目录包括了所有和体系结构相关的核心代码。它的每一个子目录都代表一种支持的体系结构，例如 i386 就是关于 Intel CPU 及与之相兼容体系结构的子目录。PC 一般都基于此目录，在 2.6 版本以后，增加了 ARM 目录。

（9）Include/：include 子目录包括编译核心所需要的大部分头文件。与平台无关的头文件在 include/linux 子目录下，与 Intel CPU 相关的头文件在 include/asm-i386 子目录下，而 include/scsi 目录则是有关 SCSI 设备的头文件目录。

（10）Init/：这个目录包含核心的初始化代码（注：不是系统的引导代码），包含两个文件 main.c 和 Version.c，是研究核心如何工作的好的起点之一。

（11）Mm/：这个目录包括所有独立于 CPU 体系结构的内存管理代码，如页式存储管理内存的分配和释放等；而和体系结构相关的内存管理代码则位于 arch/*/mm/目录下，例如 arch/i386/mm/Fault.c。

（12）Kernel/：主要的核心代码，此目录下的文件实现了大多数 Linux 系统的内核函数，其中最重要的文件当属 sched.c；同样，和体系结构相关的代码在 arch/*/kernel 中。

（13）Drivers/：放置系统所有的设备驱动程序；每种驱动程序又各占用一个子目录，如/block 下为块设备驱动程序，比如 ide（ide.c）。如果希望查看所有可能包含文件系统的设备是如何初始化的，可以查看 drivers/block/genhd.c 中的 device_setup()。它不仅初始化硬盘，也初始化网络，因为安装 NFS 文件系统的时候需要网络。

（14）Documentation/：文档目录，没有内核代码，只是一套有用的文档，。

（15）Fs/：所有的文件系统代码和各种类型的文件操作代码，它的每一个子目录支持一个文件系统，例如 fat 和 ext。

（16）ipc/：这个目录包含核心的进程间通信的代码。

（17）Lib/：放置核心的库代码。

（18）Net/：核心与网络相关的代码。

（19）Modules/：模块文件目录，是个空目录，用于存放编译时产生的模块目标文件。

（20）Scripts/：描述文件、脚本，用于对核心的配置。

另外，在每个子目录下一般都有一个 Makefile 和一个 Readme 文件。

2．Linux 内核配置

首先分析 Linux 内核中的配置系统结构，然后解释内核中的 Makefile 和配置文件的格式及配置语句的含义，最后通过一个简单的例子 TEST Driver 具体说明如何将自行开发的代码加入到 Linux 内核中。

（1）配置系统的基本结构

Linux 内核的配置系统由三部分组成，分别如下所示。

- Makefile：分布在 Linux 内核源代码中的 Makefile（一般在每个文件夹下都有一个 Makefile 文件），定义 Linux 内核的编译规则；
- 配置文件（config.in）：给用户提供配置选择的功能；
- 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面、基于 Ncurses 图形界面及基于 Xwindows 图形界面的用户配置界面，各自对应于 Make config、Make menuconfig 和 make xconfig）。

这些配置工具都是使用脚本语言的，如 Tcl/TK、Perl 编写的（也包含一些用 C 编写的代码）。

（2）内核 Makefile 相关文件

在内核中，Makefile 的作用是根据配置的情况构造出需要编译的源文件列表，然后分别编译，并把目标代码链接到一起，最终形成 Linux 内核二进制文件。由于 Linux 内核源代码是按照树形结构组织的，所以 Makefile 也被分布在目录树中。Linux 内核中的 Makefile 及与 Makefile 直接相关的文件如下所示。

- Makefile：顶层 Makefile，是整个内核配置、编译的总体控制文件。
- .config：内核配置文件，包含由用户选择的配置选项，用来存放内核配置后的结果（如 make config）。
- arch/*/Makefile：位于各种 CPU 体系目录下的 Makefile，如 arch/arm/Makefile，是针对特定平台的 Makefile。
- 各个子目录下的 Makefile：比如 drivers/Makefile，负责所在子目录下源代码的管理。
- Rules.make：规则文件，被所有的 Makefile 使用。

用户通过 make config 配置后，产生了.config。顶层 Makefile 读入.config 中的配置选择。顶层 Makefile 有两个主要的任务：产生 vmlinux 文件和内核模块（module）。为了达到此目的，顶层 Makefile 递归进入到内核的各个子目录中，分别调用位于这些子目录中的 Makefile。至于到底进入哪些子目录，取决于内核的配置。在顶层 Makefile 中 include arch/$(ARCH)/Makefile，包含了特定 CPU 体系结构下的 Makefile，这个 Makefile 中包含了平台相关的信息。

位于各个子目录下的 Makefile 同样也根据.config 给出的配置信息，构造出当前配置下需要的源文件列表，并在文件的最后有 include $(TOPDIR)/Rules.make。

Rules.make 定义了所有 Makefile 共用的编译规则。比如，如果需要将本目录下所有的 C 程序编译成汇编代码，需要在 Makefile 中有以下的编译规则。

```
%.s: %.c
$(CC) $(CFLAGS) -S $< -o $@
```

有很多子目录下都有同样的要求，就需要在各自的 Makefile 中包含此编译规则，这会比较麻烦。而 Linux 内核中则把此类的编译规则统一放置到 Rules.make 中，并在各自的 Makefile 中包含进了 Rules.make（include Rules.make），这样就避免了在多个 Makefile 中重复同样的规则。对于上面的例子，在 Rules.make 中对应的规则为：

```
%.s: %.c
$(CC) $(CFLAGS) $(EXTRA_CFLAGS) $(CFLAGS_$(*F)) $(CFLAGS_$@) -S $< -o
$@
```

（3）内核中 Makefile 变量

顶层 Makefile 定义并向环境中输出了许多变量，为各个子目录下的 Makefile 传递一些信息。有些变量比如 SUBDIRS，不仅在顶层 Makefile 中定义并且赋初值，而且在 arch/*/Makefile 还作了扩充。常用的变量有以下几类。

- 版本信息：版本信息有 VERSION、PATCHLEVEL、SUBLEVEL、EXTRAVERSION、KERNELRELEASE。版本信息定义了当前内核的版本，比如 VERSION=2，PATCHLEVEL=4，SUBLEVEL=18，EXATAVERSION=-rmk7，它们共同构成内核的发行版本 KERNELRELEASE：2.4.18-rmk7。
- CPU 体系结构：在顶层 Makefile 的开头，用 ARCH 定义目标 CPU 的体系结构，比如 ARCH: =arm 等。许多子目录的 Makefile 中，要根据 ARCH 的定义选择编译源文件的列表。
- 路径信息：TOPDIR 定义了 Linux 内核源代码所在的根目录。例如，各个子目录下的 Makefile 通过 $(TOPDIR)/Rules.make 就可以找到 Rules.make 的位置。SUBDIRS 定义了一个目录列表，在编译内核或模块时，顶层 Makefile 就是根据 SUBDIRS 来决定进入哪些子目录。SUBDIRS 的值取决于内核的配置，在顶层 Makefile 中 SUBDIRS 赋值为 kernel drivers mm fs net ipc lib；根据内核的配置情况，在 arch/*/Makefile 中扩充了 SUBDIRS 的值，参见下面的例子。
- 内核组成信息：有 HEAD、CORE_FILES、NETWORKS、DRIVERS、LIBS。Linux 内核文件 vmlinux 是由以下规则产生的。

```
vmlinux: $(CONFIGURATION) init/main.o init/version.o linuxsubdirs
$(LD) $(LINKFLAGS) $(HEAD) init/main.o init/version.o \
--start-group \
$(CORE_FILES) \
$(DRIVERS) \
$(NETWORKS) \
$(LIBS) \
--end-group \
-o vmlinux
```

可以看出，vmlinux 是由 HEAD、main.o、version.o、CORE_FILES、DRIVERS、NETWORKS 和 LIBS 组成的。这些变量（如 HEAD）都是用来定义连接生成 vmlinux 的目

标文件和库文件列表。其中，HEAD 在 arch/*/Makefile 中定义，用来确定被最先链接进 vmlinux 的文件列表。比如，对于 ARM 系列的 CPU，HEAD 定义为：

```
HEAD := arch/arm/kernel/head-$(PROCESSOR).o \
arch/arm/kernel/init_task.o
```

表明 head-$(PROCESSOR).o 和 init_task.o 需要最先被链接到 vmlinux 中。PROCESSOR 为 armv 或 armo，取决于目标 CPU。CORE_FILES、NETWORK、DRIVERS 和 LIBS 在顶层 Makefile 中定义，并且由 arch/*/Makefile 根据需要进行扩充。CORE_FILES 对应着内核的核心文件，有 kernel/kernel.o、mm/mm.o、fs/fs.o、ipc/ipc.o。可以看出，这些是组成内核最为重要的文件。同时，arch/arm/Makefile 对 CORE_FILES 进行了扩充。

```
# arch/arm/Makefile
# If we have a machine-specific directory, then include it in the build.
MACHDIR := arch/arm/mach-$(MACHINE)
ifeq ($(MACHDIR),$(wildcard $(MACHDIR)))
SUBDIRS += $(MACHDIR)
CORE_FILES := $(MACHDIR)/$(MACHINE).o $(CORE_FILES)
endif
HEAD := arch/arm/kernel/head-$(PROCESSOR).o \
arch/arm/kernel/init_task.o
SUBDIRS += arch/arm/kernel arch/arm/mm arch/arm/lib arch/arm/nwfpe
CORE_FILES := arch/arm/kernel/kernel.o arch/arm/mm/mm.o $(CORE_FILES)
LIBS := arch/arm/lib/lib.a $(LIBS)
```

- 编译信息：CPP、CC、AS、LD、AR、CFLAGS、LINKFLAGS

在 Rules.make 中定义的是编译的通用规则，具体到特定的场合，需要明确给出编译环境，编译环境就是在以上的变量中定义的。针对交叉编译的要求，定义了 CROSS_COMPILE。例如：

```
CROSS_COMPILE = arm-linux-
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)ld
......
```

CROSS_COMPILE 定义了交叉编译器前缀 arm-linux-，表明所有的交叉编译工具都是以 arm-linux-开头的，所以在各个交叉编译器工具之前，都加入了$(CROSS_COMPILE)，以组成一个完整的交叉编译工具文件名，比如 arm-linux-gcc。

CFLAGS 定义了传递给 C 编译器的参数。

LINKFLAGS 是链接生成 vmlinux 时，由链接器使用的参数。LINKFLAGS 在 arm/*/Makefile 中定义，比如：

```
# arch/arm/Makefile
LINKFLAGS :=-p -X -T arch/arm/vmlinux.lds
```

- 配置变量 CONFIG_*

.config 文件中有许多配置变量等式，用来说明用户配置的结果。例如 CONFIG_MODULES=y 表明用户选择了 Linux 内核的模块功能。.config 被顶层 Makefile 包含后，就形成许多配置变量，每个配置变量具有确定的值。

➢ y 表示本编译选项对应的内核代码被静态编译进 Linux 内核；

➢ m 表示本编译选项对应的内核代码被编译成模块；

➢ n 表示不选择此编译选项；如果根本就没有选择，那么配置变量的值为空。

（4）Rules.make 变量

前面讲过，Rules.make 是编译规则文件，所有的 Makefile 中都会包括 Rules.make。Rules.make 文件定义了许多变量，最重要的是那些编译、链接列表变量。

- O_OBJS，L_OBJS，OX_OBJS，LX_OBJS：本目录下需要编译进 Linux 内核 vmlinux 的目标文件列表，其中 OX_OBJS 和 LX_OBJS 中的“X”表明目标文件使用了 EXPORT_SYMBOL 输出符号。
- M_OBJS，MX_OBJS：本目录下需要被编译成可装载模块的目标文件列表。同样，MX_OBJS 中的“X”表明目标文件使用了 EXPORT_SYMBOL 输出符号。
- O_TARGET，L_TARGET：每个子目录下都有一个 O_TARGET 或 L_TARGET，Rules.make 首先从源代码编译生成 O_OBJS 和 OX_OBJS 中所有的目标文件，然后使用 $(LD) -r 把它们链接成一个 O_TARGET 或 L_TARGET。O_TARGET 以.o 结尾，而 L_TARGET 以.a 结尾。

（5）子目录 Makefile

子目录 Makefile 用来控制本级目录以下源代码的编译规则。下面通过一个例子来讲解子目录 Makefile 的组成。

```
#
# Makefile for the linux kernel.
#
# All of the (potential) objects that export symbols.
# This list comes from 'grep -l EXPORT_SYMBOL *.[hc]'.
export-objs := tc.o
# Object file lists.
obj-y :=
obj-m :=
obj-n :=
obj- :=
obj-$(CONFIG_TC) += tc.o
obj-$(CONFIG_ZS) += zs.o
obj-$(CONFIG_VT) += lk201.o lk201-map.o lk201-remap.o
# Files that are both resident and modular: remove from modular.

obj-m := $(filter-out $(obj-y), $(obj-m))
# Translate to Rules.make lists.
L_TARGET := tc.a

L_OBJS := $(sort $(filter-out $(export-objs), $(obj-y)))
LX_OBJS := $(sort $(filter $(export-objs), $(obj-y)))
M_OBJS := $(sort $(filter-out $(export-objs), $(obj-m)))
MX_OBJS := $(sort $(filter $(export-objs), $(obj-m)))

include $(TOPDIR)/Rules.make
```

此文件中包含以下内容。

- 注释：对 Makefile 的说明和解释，由#开始。
- 编译目标定义：类似于 obj-$(CONFIG_TC) += tc.o 的语句是用来定义编译的目标，

是子目录 Makefile 中最重要的部分。编译目标定义那些在本子目录下需要编译到 Linux 内核中的目标文件列表。为了只当用户选择了此功能后才编译，所有的目标定义都融合了对配置变量的判断。

前面说过，每个配置变量取值范围是 y、n、m 和空，obj-$(CONFIG_TC)分别对应着 obj-y、obj-n、obj-m、obj-。如果 CONFIG_TC 配置为 y，那么 tc.o 就进入了 obj-y 列表。obj-y 为包含到 Linux 内核 vmlinux 中的目标文件列表；obj-m 为编译成模块的目标文件列表；obj-n 和 obj-中的文件列表被忽略。配置系统就根据这些列表的属性进行编译和链接。

export-objs 中的目标文件都使用 EXPORT_SYMBOL()定义了公共的符号，以便可装载模块使用。在 tc.c 文件的最后部分，有“EXPORT_SYMBOL(search_tc_card);”，表明 tc.o 有符号输出。

这里需要指出的是，对于编译目标的定义存在着两种格式，分别是老式定义和新式定义。老式定义就是前面 Rules.make 使用的那些变量，新式定义就是 obj-y、obj-m、obj-n 和 obj-。Linux 内核推荐使用新式定义，不过由于 Rules.make 不理解新式定义，需要在 Makefile 中的适配段将其转换成老式定义。

- 适配段：适配段的作用是将新式定义转换成老式定义。在上面的例子中，适配段就是将 obj-y 和 obj-m 转换成 Rules.make 能够理解的 L_TARGET、L_OBJS、LX_OBJS、M_OBJS、MX_OBJS。

L_OBJS:=$(sort $(filter-out $(export-objs), $(obj-y))) 定义了 L_OBJS 的生成方式，在 obj-y 的列表中过滤掉 export-objs（tc.o），然后排序并去除重复的文件名。这里使用到 GNU Make 的一些特殊功能，具体的含义可参考 Make 的文档（info make）。

- include $(TOPDIR)/Rules.make：包含上层 Rules.make 文件。

（6）config.in 配置功能概述

除了 Makefile 的编写，另外一个重要的工作就是把新功能加入到 Linux 的配置选项中，提供此项功能的说明，让用户有机会选择此项功能。所有的这些都需要在 config.in 文件中用配置语言来编写配置脚本，在 Linux 内核中，配置命令有多种方式。

```
Make config, make oldconfig scripts/Configure
Make menuconfig scripts/Menuconfig
Make xconfig scripts/tkparse
```

以字符界面配置（make config）为例，顶层 Makefile 调用 scripts/Configure，按照 arch/arm/config.in 来进行配置。命令执行完后产生文件.config，其中保存着配置信息。下一次再做 make config 将产生新的.config 文件，原.config 被改名为.config.old。

（7）配置语言

- 顶层菜单：mainmenu_name /prompt/，其中/prompt/ 是用单引号'或双引号"包围的字符串，单引号'与双引号"的区别是'…'中可使用 $ 引用变量的值。mainmenu_name 设置最高层菜单的名字，它只在 make xconfig 时才会显示。
- 询问语句

```
bool /prompt/ /symbol/
hex /prompt/ /symbol/ /word/
int /prompt/ /symbol/ /word/
string /prompt/ /symbol/ /word/
tristate /prompt/ /symbol/
```

询问语句首先显示一串提示符/prompt/，等待用户输入，并把输入的结果赋给/symbol/所代表的配置变量。不同的询问语句的区别在于它们接受的输入数据类型不同，比如 bool 接受布尔类型（y 或 n），hex 接受 16 进制数据。有些询问语句还有第三个参数/word/，用来给出默认值。

- 定义语句

```
define_bool /symbol/ /word/
define_hex /symbol/ /word/
define_int /symbol/ /word/
define_string /symbol/ /word/
define_tristate /symbol/ /word/
```

不同于询问语句等待用户输入，定义语句显式地给配置变量/symbol/赋值/word/。

- 依赖语句

```
dep_bool /prompt/ /symbol/ /dep/ ...
dep_mbool /prompt/ /symbol/ /dep/ ...
dep_hex /prompt/ /symbol/ /word/ /dep/ ...
dep_int /prompt/ /symbol/ /word/ /dep/ ...
dep_string /prompt/ /symbol/ /word/ /dep/ ...
dep_tristate /prompt/ /symbol/ /dep/ ...
```

与询问语句类似，依赖语句也是定义新的配置变量。不同的是，配置变量/symbol/的取值范围将依赖于配置变量列表/dep/ …。这就意味着被定义的配置变量所对应功能的取舍取决于依赖列表所对应功能的选择。以 dep_bool 为例，如果/dep/ …列表的所有配置变量都取值 y，则显示/prompt/，用户可输入任意的值给配置变量/symbol/，但是只要有一个配置变量的取值为 n，则/symbol/被强制成 n。不同依赖语句的区别在于它们由依赖条件所产生的取值范围不同。

- 选择语句

choice /prompt/ /word/ /word/：choice 语句首先给出一串选择列表，供用户选择其中一种。比如 Linux for ARM 支持多种基于 ARM core 的 CPU，Linux 使用 choice 语句提供一个 CPU 列表，供用户选择。

```
choice 'ARM system type' \
"Anakin CONFIG_ARCH_ANAKIN \
Archimedes/A5000 CONFIG_ARCH_ARCA5K \
Cirrus-CL-PS7500FE CONFIG_ARCH_CLPS7500 \
……
SA1100-based CONFIG_ARCH_SA1100 \
Shark CONFIG_ARCH_SHARK" RiscPC
```

Choice 首先显示/prompt/，然后将/word/分解成前后两个部分，前部分为对应选择的提示符，后部分是对应选择的配置变量。用户选择的配置变量为 y，其余的都为 n。

- if 语句

```
if [ /expr/ ] ; then
/statement/
...
fi

if [ /expr/ ] ; then
```

```
/statement/
...
else
/statement/
...
fi
```

if 语句对配置变量（或配置变量的组合）进行判断，并做出不同的处理。判断条件/expr/可以是单个配置变量或字符串，也可以是带操作符的表达式。操作符有=、!=、-o、-a 等。

- 菜单块（menu block）语句

```
mainmenu_option next_comment
comment '……'
…
endmenu
```

以上代码用来引入新的菜单。在向内核增加新的功能后，需要相应地增加新的菜单，并在新菜单下给出此项功能的配置选项。Comment 后带的注释就是新菜单的名称。所有归属于此菜单的配置选项语句都写在 comment 和 endmenu 之间。

- Source 语句：source /word/

/word/是文件名，source 的作用是调入新的文件。

（8）默认配置

Linux 内核支持非常多的硬件平台，对于具体的硬件平台而言，有些配置是必需的，有些配置不是必需的。另外，新增加功能的正常运行往往也需要一定的先决条件，针对新功能，必须做相应的配置。因此，特定硬件平台能够正常运行对应着一个最小的基本配置，这就是默认配置。

Linux 内核中针对每个 ARCH 都会有一个默认配置。在向内核代码增加了新的功能后，如果新功能对于这个 ARCH 是必需的，就要修改此 ARCH 的默认配置。修改方法如下（在 Linux 内核根目录下）。

备份.config 文件

```
cp arch/arm/deconfig .config
```

修改.config

```
cp .config arch/arm/deconfig
```

恢复.config

如果新增的功能适用于许多的 ARCH，只要针对具体的 ARCH，重复上面的步骤就可以。

（9）帮助文件

在配置 Linux 内核时，遇到不懂含义的配置选项，可以查看它的帮助，从中可得到选择的建议。所有配置选项的帮助信息都在 Documentation/Configure.help 中，它的格式为：

```
<description>
<variable name>
<help file>
```

<description>给出本配置选项的名称，<variable name>对应配置变量，<help file>对应配置帮助信息。在帮助信息中，首先简单描述此功能，其次说明选择了此功能后会有什么

效果，不选择又有什么效果，最后，不要忘了写上“如果不清楚，选择 N（或者）Y”，给不知所措的用户以提示。

3．内核配置实例

对于一个开发者来说，将自己开发的内核代码加入到 Linux 内核中，需要 3 个步骤。

- 首先，确定把自己开发代码放入到内核的位置；
- 其次，把自己开发的功能增加到 Linux 内核的配置选项中，使用户能够选择此功能；
- 最后，构建子目录 Makefile，根据用户的选择，将相应的代码编译到最终生成的 Linux 内核中去。

以下通过一个简单的例子 test driver，结合前面学到的知识，来说明如何向 Linux 内核中增加新的功能。

（1）源代码目录结构

添加自己的源代码，将 test driver 文件夹放置在 drivers/test/目录下，其目录结构如下。

```
$cd drivers/test
$tree
.
|-- Config.in
|-- Makefile
|-- cpu
| |-- Makefile
| `-- cpu.c
|-- test.c
|-- test_client.c
|-- test_ioctl.c
|-- test_proc.c
|-- test_queue.c
`-- test
|-- Makefile
`-- test.c
```

（2）配置文件说明

- drivers/test/Config.in 文件

```
#
# TEST driver configuration
#
mainmenu_option next_comment
comment 'TEST Driver'

bool 'TEST support' CONFIG_TEST
if [ "$CONFIG_TEST" = "y" ]; then
tristate 'TEST user-space interface' CONFIG_TEST_USER
bool 'TEST CPU ' CONFIG_TEST_CPU
fi

endmenu
```

test driver 对于内核来说是新的功能，所以首先创建一个菜单 TEST Driver；然后，显示“TEST support”，等待用户选择；接下来判断用户是否选择了 TEST Driver，如果是（CONFIG_TEST=y），则进一步显示子功能——用户接口与 CPU 功能支持；由于用户接口

功能可以被编译成内核模块，所以这里的询问语句使用了 tristate（因为 tristate 的取值范围包括 y、n 和 m，m 就是对应的模块）。

- arch/arm/config.in

在文件的最后加入 source drivers/test/Config.in，将 TEST Driver 子功能的配置纳入到 Linux 内核的配置中。

（3）Makefile 文件说明

- drivers/test/Makefile 文件

```
# drivers/test/Makefile
#
# Makefile for the TEST.
#
SUB_DIRS :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS) cpu
L_TARGET := test.a
export-objs := test.o test_client.o

obj-$(CONFIG_TEST) += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER) += test_ioctl.o
obj-$(CONFIG_PROC_FS) += test_proc.o

subdir-$(CONFIG_TEST_CPU) += cpu

include $(TOPDIR)/Rules.make

clean:
for dir in $(ALL_SUB_DIRS); do make -C $$dir clean; done
rm -f *.[oa] .*.flags
```

drivers/test 目录下最终生成的目标文件是 test.a。在 test.c 和 test-client.c 中使用了 EXPORT_SYMBOL 输出符号，所以 test.o 和 test-client.o 位于 export-objs 列表中。然后，根据用户的选择（具体来说，就是配置变量的取值），构建各自对应的 obj-*列表。由于 TEST Driver 中包含一个子目录 cpu，当 CONFIG_TEST_CPU=y（即用户选择了此功能）时，需要将 cpu 目录加入到 subdir-y 列表中。

- drivers/test/cpu/Makefile 文件说明

```
# drivers/test/test/Makefile
#
# Makefile for the TEST CPU
#

SUB_DIRS :=
MOD_SUB_DIRS := $(SUB_DIRS)
ALL_SUB_DIRS := $(SUB_DIRS)
L_TARGET := test_cpu.a
obj-$(CONFIG_test_CPU) += cpu.o
include $(TOPDIR)/Rules.make
clean:
rm -f *.[oa] .*.flags
```

- drivers/Makefile 文件修改

```
……
subdir-$(CONFIG_TEST) += test
……
include $(TOPDIR)/Rules.make
```

在 drivers/Makefile 中加入 subdir-$(CONFIG_TEST)+= test，使得在用户选择 TEST Driver 功能后，内核编译时能够进入 test 目录。

- 顶层 Makefile 文件修改

```
……
DRIVERS-$(CONFIG_PLD) += drivers/pld/pld.o
DRIVERS-$(CONFIG_TEST) += drivers/test/test.a
DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a
DRIVERS := $(DRIVERS-y)
……
```

在顶层 Makefile 中加入 DRIVERS-$(CONFIG_TEST) += drivers/test/test.a 和 DRIVERS-$(CONFIG_TEST_CPU) += drivers/test/cpu/test_cpu.a。如果用户选择了 TEST Driver，那么 CONFIG_TEST 和 CONFIG_TEST_CPU 都是 y，test.a 和 test_cpu.a 就都位于 DRIVERS-y 列表中，然后又被放置在 DRIVERS 列表中。在前面曾经提到过，Linux 内核文件 vmlinux 的组成中包括 DRIVERS，所以 test.a 和 test_cpu.a 最终可被链接到 vmlinux 中。

关于驱动程序源代码这里不再介绍。

1.3.3 移植 Linux 操作系统

本小节将介绍如何编译可以运行于 ARM 处理器的 Linux 操作系统可执行文件。由于前面已经详细介绍构建嵌入式 Linux 交叉编译环境，这里仅介绍如何重新编译 Linux 内核源代码程序。

1．配置前准备工作

Linux 内核版本发布的官方网站是 http://www.kernel.org。编译内核需要 root 权限，以下以 Linux-2.4.0 版本内核为例介绍编译过程。

（1）准备工作，解压源代码

把需要升级的内核复制到/usr/src/下，命令为

```
#cp linux-2.4.0test8.tar.gz /usr/src
```

首先来查看一下当前/usr/src 的内容，有一个 Linux 的符号链接，它指向一个类似于 Linux-X.X.X（对应于现在使用的内核版本号）的目录。首先删除这个链接。

```
#cd /usr/src
#rm -f linux
```

接着解压下载的源程序文件。如果所下载的是.tar.gz（.tgz）文件，请使用下面的命令。

```
#tar -xzvf linux-2.4.0test8.tar.gz
```

如果所下载的是.bz2 文件，例如 linux-2.4.0test8.tar.bz2，请使用下面的命令。

```
#bzip2 -d linux-2.4.0test8.tar.bz2
```

```
#tar -xvf linux.2.4.0.test8.tar
```

现在再来看一下/usr/src 下的内容，会发现有一个名为 Linux 的目录，里面就是需要升级到的版本的内核的源程序。之所以使用前面删除的那个链接就是防止在升级内核的时候会不慎把原来版本内核的源程序覆盖掉。这里需要同样处理。

```
#mv linux linux-2.4.0test8
#ln -s linux-2.4.0test8 linux
```

另外，如果还下载了 patch 文件，比如 patch-2.4.0test8，就可以进行 patch 操作（下面假设 patch-2.4.0test8 已经位于/usr/src 目录下了，否则需要先把该文件复制到/usr/src 下）。

```
#patch -p0 < patch-2.4.0test8
```

（2）准备工作

运行的第一个命令是：

```
#cd /usr/src/linux;make mrproper
```

该命令确保源代码目录下没有不正确的.o 文件及文件的互相依赖。

确保/usr/include/目录下的 asm、Linux 和 scsi 等链接是指向要升级的内核源代码的。它们分别链向源代码目录下该计算机体系结构（对于 PC 来说，使用的体系结构是 i386）所需要的真正的 include 子目录。如 asm 指向/usr/src/linux/include/asm-i386 等。若没有这些链接，就需要手工创建，按照下面的步骤进行。

```
# cd /usr/include/
# rm -r asm linux scsi
# ln -s /usr/src/linux/include/asm-i386 asm
# ln -s /usr/src/linux/include/linux linux
# ln -s /usr/src/linux/include/scsi scsi
```

这是配置中非常重要的一部分。删除/usr/include 下的 asm、Linux 和 scsi 链接后，再创建新的链接指向新内核源代码目录下的同名的目录。这些头文件目录包含着保证内核在系统上正确编译所需要的重要的头文件。现在应该明白为什么上面又在/usr/src 下“多余”地创建名为 linux 的链接。

（3）配置内核

接下来的内核配置过程比较烦琐，但是配置的适当与否与日后 Linux 的运行直接相关，所以有必要了解一下一些主要的且经常用到的选项的设置。配置内核可以根据需要与爱好使用下面命令中的一个。

- #make config（基于文本的最传统的配置界面，不推荐使用）。
- #make menuconfig（基于文本选单的配置界面，字符终端下推荐使用）。
- #make xconfig（基于图形窗口模式的配置界面，Xwindow 下推荐使用）。
- #make oldconfig（如果只想在原来内核配置的基础上修改一些小地方，会省去不少麻烦）。

这 4 个命令中，make xconfig 的界面最为友好，如果可以使用 Xwindow，那么就推荐使用这个命令。如果不能使用 Xwindow，那么就使用 make menuconfig。界面虽然比 make xconfig 差一些，总比 make config 要好得多。

选择相应的配置时，有三种选择，它们分别代表的含义如下。

- Y——将该功能编译进内核；
- N——不将该功能编译进内核；
- M——将该功能编译成可以在需要时动态插入到内核中的模块。

如果使用的是 make xconfig，使用鼠标就可以选择对应的选项。如果使用的是 make menuconfig，则需要使用空格键进行选取。会发现在每一个选项前都有个括号，但有的是方括号有的是尖括号，还有一种圆括号。用空格键选择时可以发现，方括号里要么是空，要么是"*"，而尖括号里可以是空。"*"和"M"表示前者对应的项要么不要，要么编译到内核里；后者则多一种选择，可以编译成模块。而圆括号的内容是要在所提供的几个选项中选择一项。

在编译内核的过程中，最繁杂的事情就是配置工作，很多初学者不清楚到底该如何选取这些选项。实际上在配置时大部分选项可以使用其默认值，只有小部分需要根据用户不同的需要选择。选择的原则是将与内核其他部分关系较远且不经常使用的部分功能代码编译成为可加载模块，有利于减小内核的长度，减小内核消耗的内存，简化该功能相应的环境改变时对内核的影响；不需要的功能就不要选；与内核关系紧密而且经常使用的部分功能代码直接编译到内核中。以下就常用的选项分别加以介绍。

2．Linux 内核配置选项说明

（1）Code maturity level options：代码成熟等级。此处只有一项 prompt for development and/or incomplete code/drivers，如果要试验现在仍处于实验阶段的功能，比如 khttpd、IPv6 等，就必须把该项选择为 Y 了，否则可以把它选择为 N。

（2）Loadable module support：对模块的支持。这里面有三项，如下所示。

- Enable loadable module support：除非准备把所有需要的内容都编译到内核里面，否则该项应该是必选的。
- Set version information on all module symbols：可以不选它。
- Kernel module loader：让内核在启动时有自己装入必需模块的能力，建议选上。

（3）Processor type and features：CPU 类型。内容很多，有关的几个如下所示。

- Processor family：根据自己的情况选择 CPU 类型。
- High Memory Support：大容量内存的支持。可以支持到 4GB、64GB，一般可以不选。
- Math emulation：协处理器仿真。协处理器是在 386 时代的宠儿，现在早已不用了。
- MTTR support：MTTR 支持。可不选。
- Symmetric multi-processing support：对称多处理支持。除非有多个 CPU，否则不用选。

（4）General setup：这里是对最普通的一些属性进行设置。这部分内容非常多，一般使用默认设置就可以了。下面介绍经常使用的一些选项。

- Networking support：网络支持。必选，没有网卡也建议选上。PCI support：PCI 支持。如果使用了 PCI 的卡，当然必选。
- PCI access mode：PCI 存取模式。可供选择的有 BIOS、Direct 和 Any，选择 Any。
- Support for hot-pluggabel devices：热插拔设备支持。支持得不是太好，可不选。
- PCMCIA/CardBus support：PCMCIA/CardBus 支持。有 PCMCIA 就必选。
- System V IPC：System V 进程间通信。

- BSD Process Accounting：BSD 的进程处理。
- Sysctl support：这三项是有关进程处理/IPC 调用的，主要是 System V 和 BSD 两种风格。如果不是使用 BSD，就可以按照默认设置。
- Power Management support：电源管理支持。Advanced Power Management BIOS support：高级电源管理 BIOD 支持。

（5）Memory Technology Device（MTD）：MTD 设备支持。如果是 Flash 设备，则需要选择。

（6）Parallel port support：并口支持。

（7）Plug and Play configuration：即插即用支持。

（8）Block devices：块设备支持。

- Normal PC floppy disk support：普通 PC 软盘支持。
- XT hard disk support：XT 硬件支持。
- Compaq SMART2 support：Compaq SMART 阵列控制器支持
- Mulex DAC960/DAC1100 PCI RAID Controller support：RAID 镜像使用。
- Loopback device support：回环设备支持。
- Network block device support：网络块设备支持。如果想访问网上邻居，就选择。
- Logical volume manager（LVM）support：逻辑卷管理支持。
- Multiple devices driver support：多设备驱动支持。
- RAM disk support：RAM 盘支持。

（9）Networking options：网络选项。这里配置的是网络协议。主要包括 TCP/IP、ATM、IPX、DECnet、Appletalk 等，支持的协议很多，IPv6 也支持。

（10）Telephony Support：电话支持。Linux 下可以支持电话卡，这样就可以在 IP 上使用普通的电话提供语音服务了。

（11）ATA/IDE/MFM/RLL support：这个是有关各种接口的硬盘/光驱/磁带/软盘支持的。

（12）SCSI support：SCSI 设备的支持。

（13）IEEE 1394（FireWire）support：IEEE 1394 支持。

（14）I2O device support：在智能 Input/Output（I2O）体系接口中使用。

（15）Network device support：网络设备支持。根据选择的协议相应设备有：ARCnet 设备、Ethernet（10 or 100 Mb/s）、Ethernet（1000Mb/s）、Wireless LAN（non-hamradio）、Token Ring device、WAN interfaces、PCMCIA network device support 几大类。

（16）Amateur Radio support：配置业余无线广播支持。

（17）IrDA（infrared）support：红外支持。

（18）ISDN subsystem：支持 ISDN 上网。

（19）Old CD-ROM drivers（not SCSI、not IDE）：老式光盘。

（20）Character devices：字符设备。包括的设备如下所示。

I2C support：I2C 是 Philips 极力推动的微控制应用中使用的低速串行总线协议。如果要选择下面的 Video For Linux，该项必选。

Mice：鼠标。现在可以支持总线、串口、PS/2、C&T 82C710 mouse port、PC110 digitizer pad。

Joysticks：手柄。

Video For Linux：支持有关的音频/视频卡。

（21）File systems：文件系统。包括的内容如下所示。

Quota support：Quota 可以限制每个用户可以使用的硬盘空间的上限，在多用户共同使用一台主机的情况中十分有效。

DOS FAT fs support：DOS FAT 文件格式的支持，可以支持 FAT16、FAT32。

ISO 9660 CD-ROM file system support：光盘使用的就是 ISO 9660 的文件格式。

NTFS file system support：NTFS 是 NT 使用的文件格式。

/proc file system support：/proc 文件系统是 Linux 提供给用户和系统进行交互的通道，建议选上，否则有些功能没法正确执行。

另外还有 Network File Systems（网络文件系统）、Partition Types（分区类型）、Native Language Support（本地语言支持）。值得一提的是 Network File Systems 中的两种——NFS 和 SMB 分别是 Linux 和 Windows 相互以网络邻居的形式访问对方所使用的文件系统，根据需要加以选择。

（22）Console drivers：控制台驱动。

（23）Sound：声卡驱动。

（24）USB supprot：USB 支持。很多 USB 设备比如鼠标、调制解调器、打印机、扫描仪等，在 Linux 中都可以得到支持，根据需要自行选择。

（25）Kernel hacking：配置这个选项，即使在系统崩溃时，也可以进行一定的工作。

3．编译内核

在繁杂的配置工作完成以后，就可以耐心等候了。与编译有关的命令有如下几个。

```
#make dep
#make clean
#make zImage
#make bzImage
#make modules
#make modules_install
#depmod -a
```

第一个命令 make dep 实际上读取配置过程生成的配置文件，来创建对应于配置的依赖关系树，从而决定哪些需要编译而哪些不需要；

第二个命令 make clean 完成删除前面步骤留下的文件，以避免出现一些错误；

第三个命令 make zImage 和第四个命令 make bzImage 实现完全编译内核，二者生成的内核都是使用 gzip 压缩的，只要使用一个就可以，它们的区别在于使用 make bzImage 可以生成大一点的内核，比如在编译 2.4.0 版本的内核时如果使用 make zImage 命令，那么就会出现 system too big 的错误提示。建议大家使用 make bzImage 命令。

后面三个命令只有在进行配置的过程中，回答 Enable loadable module support (CONFIG_MODULES)时选择“Yes”才是必要的，make modules 和 make modules_install 分别生成相应的模块和把模块复制到需要的目录中。

1.4 本章总结

本章简单介绍了嵌入式的基础入门知识，主要包括嵌入式系统的基本概念、内核结构及操作系统的移植。通过本章的学习，读者对嵌入式的类型和应用有一个大致的了解。

第 2 章

Linux 系统开发环境平台

嵌入式 Linux 应用程序开发与 X86 计算机上的 Linux 应用程序开发最主要的区别在于应用平台不一样，相应的编译环境不一样，但由于语法结构、编程理念完全一样，只需要选用正确的编译环境，应用于 X86 平台上的上层 Linux 应用程序几乎可以完全不做任何修改就可以移植到嵌入式设备上。因此，本章将详细讲述 Linux 系统开发的环境平台，主要包括进程/线程管理、文件系统结构和类型、存储管理、设备管理等，这是 Linux 环境下最基本、最重要的知识点。

2.1 进程/线程管理

进程/线程是 Linux 对任务管理的基本理论基础，本节主要阐述 Linux 下进程和线程的基本概念、基本操作及进程间的通信等内容，同时对每一类操作都给出相应的实例。

2.1.1 进程/线程的概念

1. 进程的概念

Linux 是一个多用户多任务的操作系统，系统的所有任务在内核的调度下在 CPU 中执行，Linux 在很多时候将任务和进程的概念合在一起，进程是一个动态地使用系统资源、处于活动状态的应用程序。Linux 进程管理由进程控制块 PCB、进程调度、中断管理、任务队列等组成，它是 Linux 文件系统、存储管理、设备管理和驱动程序的基础。

一个程序可以启动多个进程，它的每个运行副本都有自己的进程空间。

每一个进程都有自己特有的属性，所有这些信息都存储在进程控制块的 PCB 中，主要包括进程 PID、进程所占有的内存区域、文件描述符和进程环境等信息，它用 task_struct 的数据结构表示。此数据结构在 Linux 的源代码文件夹 include/linux/sch.h 中定义。

```
struct task_struct {
       /*
        * offsets of these are hardcoded elsewhere - touch with care
        */
       volatile long state;
       unsigned long flags;    /* per process flags, defined below */
       int sigpending;
       mm_segment_t addr_limit;       /* thread address space:
                                         0-0xBFFFFFFF for user-thead
                                         0-0xFFFFFFFF for kernel-thread
                                  */
```

```
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;

        int lock_depth;       /* Lock depth */

        /*
         * offset 32 begins here on 32-bit platforms.
         */
        unsigned int cpu;
        int prio, static_prio;
        struct list_head run_list;
        prio_array_t *array;

        unsigned long sleep_avg;
        unsigned long last_run;

        unsigned long policy;
        unsigned long cpus_allowed;
        unsigned int time_slice, first_time_slice;

        atomic_t usage;

        struct list_head tasks;
        struct list_head ptrace_children;
        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;

/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
        int pdeath_signal;
        unsigned long personality;
        int did_exec:1;
        unsigned task_dumpable:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;
        /* boolean value for session group leader */
        int leader;

        struct task_struct *real_parent;
        struct task_struct *parent;
        struct list_head children;
        struct list_head sibling;
        struct task_struct *group_leader;

        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];

        wait_queue_head_t wait_chldexit;      /* for wait4() */
        struct completion *vfork_done;        /* for vfork() */
        int *set_child_tid;
        int *clear_child_tid;
```

```
        unsigned long rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        struct tms times;
        struct tms group_times;
        unsigned long start_time;
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];

        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t  groups[NGROUPS];
        kernel_cap_t  cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;
/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
/* file system info */
        int link_count, total_link_count;
        struct tty_struct *tty; /* NULL if no tty */
        unsigned int locks; /* How many file locks are being held */
/* ipc stuff */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;

        struct thread_struct thread;
        struct fs_struct *fs;
        struct files_struct *files;
        struct namespace *namespace;
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;

        /* TUX state */
        void *tux_info;
        void (*tux_exit)(void);


/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
        spinlock_t alloc_lock;
```

```
        spinlock_t switch_lock;

        void *journal_info;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
};
        void (*tux_exit)(void);


/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
        spinlock_t alloc_lock;
        spinlock_t switch_lock;

        void *journal_info;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
};
```

2．任务状态及转换

Linux 任务（进程）分为以下几种状态，分别是运行状态、等待状态（可以被中断）、等待状态（不可以被中断）、停止状态、睡眠状态和僵死状态。其定义如下：

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_ZOMBIE             8
#define TASK_DEAD               16
```

进程的状态转换关系如图 2-1 所示。

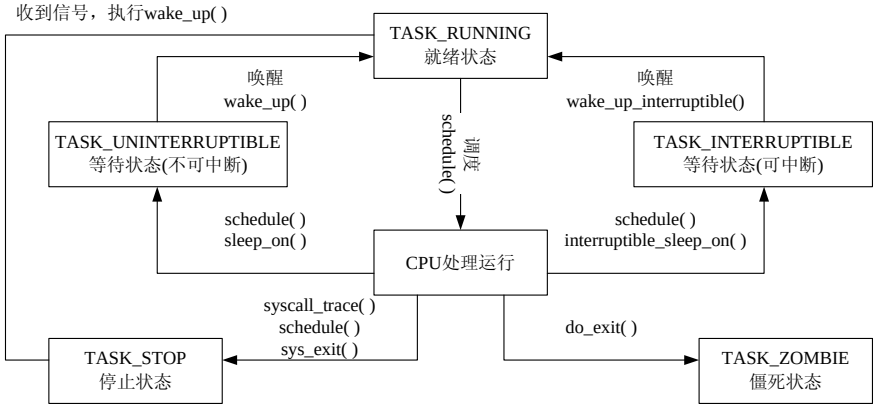


图 2-1　进程状态转换图

3．线程的概念

线程是 Linux 任务管理中另一个重要概念，一个进程中可以包含多个线程，一个线程可以与当前进程中的其他线程进行数据交换，共享数据，但拥有自己的栈空间。线程和进

程各有自己的优缺点，线程开销小，但不利于资源的保护，进程相反。线程可以分为内核线程、轻量级线程和用户级线程 3 种。在 Linux 应用程序开发中，在很多情况下采用多线程，多线程作为一种多任务、并发的工作方式，有以下的优点。

（1）提高应用程序响应速度。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种情况。

（2）使多 CPU 系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。

（3）改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序有利于理解和修改。

另外，LIBC 中的 pthread 库提供了大量的 API 函数，为用户编写应用程序提供支持。

2.1.2　进程基本操作

由于进程是任务管理的基本形式，因此，程序员需要有效地对进程进行管理。常见的进程管理方式包括获取进程信息、设置进程属性、创建进程、执行进程、退出进程及跟踪进程 6 个主要操作。

1．获取进程信息函数

获取进程信息主要通过读取进程控制块 PCB 中的信息，主要包括：

- getpid()函数获取进程 PID；
- getppid()函数获得父进程 PID；
- getpgid()函数获得组识别码；
- getpgrp()函数获得当前进程组识别码；
- getpriority()函数获得进程执行的优先级。

（1）getpid()

功能：getpid 用来获得目前进程的进程标识。

定义函数：pid_t getpid(void)。

返回值：返回当前进程的进程识别号。

头文件：#include<unistd.h>

（2）getppid()

功能：getppid 用来获得目前进程的父进程标识。

定义函数：pid_t getppid(void)。

返回值：返回当前进程的父进程识别号。

头文件：#include<unistd.h>

（3）getpgid()

功能：getpgid 用来获得参数 pid 指令进程所属于的组识别码，如果参数为 0，则会返回当前进程的组识别码。

定义函数：pid_t getpgid(pid_t pid)。

返回值：执行成功则返回正确的组识别码，如果有错误则返回-1，错误原因存在于 errno 中。

头文件：#include<unistd.h>

（4）getpgrp()

功能：getpgrp 用来获得目前进程所属于的组识别码。此函数相当于调用 getpgid(0)。

定义函数：pid_t getpgrp(void)。

返回值：执行成功则返回正确的组识别码。

头文件：#include<unistd.h>

（5）getpriority 获得进程执行的优先级

功能：getpriority 用来获得进程、进程组和用户的进程执行优先权。

定义函数：int getpriority(int which,int who)。其中参数如表 2-1 所示。

表 2-1　不同 which 下 who 代表的意义

which	who 代表的意义
PRIO_PROCESS	who 为进程的识别码
PRIO_PGRP	who 为进程的组识别码
PRIO_USER	who 为用户识别码

返回值：执行成功则返回当前进程的优先级（–20~20），值越小优先级越高。如果错误返回–1，错误存在于 errno 中。

头文件：#include<sys/time.h>或<sys/resource.h>

```
 [root@yangzongde ch03_01]# cat get_process_information.c
#include<sys/resource.h>                    //getpriority 头文件
#include<unistd.h>
main()
       {
       printf("the process pid is %d\n",getpid());
                                            //获取进程 PID
       printf("the process parent pid is %d\n",getppid());
                                            //获取父进程 PID
       printf("the current process gid is %d\n",getpgid(getpid()));
                                            //获取当前进程的进程组 PID
       printf("the process gid is %d\n",getpgrp());
                                          //获取当前进程的进程组 PID
       printf("the  process  priority  is  %d\n",getpriority(PRIO_PROCESS,
getpid()));                                 //获取当前进程的优先级
       }
    [root@yangzongde  ch03_01]#  gcc  -o  get_process_information  get_process_
information.c
    [root@yangzongde ch03_01]# ls
    get_process_information  get_process_information.c  get_process_information.o
    [root@yangzongde ch03_01]# ./get_process_information
    the process pid is 13537
    the process parent pid is 13438
    the current process gid is 13537
    the process gid is 13537
    the process priority is 0
```

2．设置进程属性

设置进程属性操作主要用来修改进程 PCB 中的进程属性，主要包括：

- nice()函数改变进程优先级；
- setpgid()函数设置进程组识别码；
- setpgrp()函数设置进程组识别码；
- setpriority()函数设置程序进程执行优先级。

（1）nice()

功能：nice 用来改变进程的进程执行优先级，其参数越大则优先级顺序越低，只有超级用户才能使用负的优先级。

定义函数：int nice(int inc)。

返回值：如果执行成功返回 0，否则返回–1，失败原因存在于 errno 中。

头文件：#include<unistd.h>

（2）setpgid()

功能：setpgid()将参数 pid 指定进程所属的组识别码设置为参数 pgid 指定的组识别码，如果参数 pid 为 0，用来设置目前进程的组识别码，如果参数 pgid 为 0，则会以目前进程的进程识别码来取代。

定义函数：int setpgid(pid_t pid,pid_pgid)。

返回值：如果执行成功返回组识别码，否则返回-1，失败原因存在于 errno 中。

头文件：#include<unistd.h>

（3）setpgrp()

功能：setpgrp 用来将目前进程所属于的组识别码设置成为目前进程的进程识别码，此函数相当于调用 setpgid(0,0)。

定义函数：int setpgrp(void)。

返回值：如果执行成功返回组识别码，否则返回-1，失败原因存在于 errno 中。

头文件：#include<unistd.h>

（4）setpriority()

功能：setpriority 用来设置进程、进程组和用户的进程优先级，参数 which 有 3 种值，应用参数 who 有 3 种不同意义。其中参数如表 2-2 所示。

表 2-2　不同 which 下 who 代表意义

which	who 代表的意义
PRIO_PROCESS	who 为进程的识别码
PRIO_PGRP	who 为进程的组识别码
PRIO_USER	who 为用户识别码

定义函数：int setpriority(int which,int who,int prio)。

返回值：如果执行成功返回 0，否则返回-1，失败原因存在于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_02]# cat set_process_information.c
#include<sys/resource.h>
#include<unistd.h>
main()
       {
       printf("the process priority is %d\n",getpriority(PRIO_PROCESS,
getpid()));
```

```
          nice(10);
          printf("after nice(10),the process priority is %d\n",getpriority
(PRIO_PROCESS,getpid()));
          printf("the progress gid is %d\n",getpgid(getpid()));
          printf("the process current priority is %d\n",getpriority(PRIO_
PROCESS,getpid()));
          setpriority(PRIO_PROCESS,getpid(),-10);
          printf("the modify process priority is %d\n",getpriority(PRIO_
PROCESS,getpid()));
          }
    [root@yangzongde ch03_02]# gcc -o set_process_information set_process_
information.c
    [root@yangzongde ch03_02]# ./set_process_information
    the process priority is 0
    after nice(10),the process priority is 10
    the progress gid is 13759
    the process current priority is 10
    the modify process priority is -10
```

3．进程创建

在 Linux 环境下创建进程主要用来调用 fork()函数以建立一个新进程。Linux 下所有的进程都是由进程 init 创建的。fork()函数有以下功能。

功能：fork 函数用来产生一个新进程，其子进程会复制父进程的数据和堆栈空间，并继承父进程的用户代码、组代码、环境变量、已经打开的文件代码、工作目录和资源限制。另外，Linux 采用写时复制技术（Copy-on-write），只有当进程试图修改欲复制的空间时才会真正地复制数据，由于这些继承的信息是复制而来的，并非指相同的内存空间，因此子进程对这些变量的修改和父进程并不会同步，此外，子进程不会继承父进程的文件锁定和未处理的信号。

定义函数：pid_t fork(void)。

返回值：如果执行成功将在父进程中返回新建子进程的 PID，而在新建立的子进程中返回 0，如果失败返回–1，失败代码存放在 errno 中。

头文件：#include<unistd.h>

```
    [root@yangzongde ch03_03]# cat fork_example.c
    #include <sys/types.h>
    #include <unistd.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <errno.h>
    extern int errno;
    int main()
    {
      char buf[100];
      pid_t cld_pid;
      int fd;
      int status;
    if ((fd=open("temp",O_CREAT|O_TRUNC | O_RDWR,S_IRWXU)) == -1)
      {
       printf("open error %d\n",errno);
       exit(1);
```

```
 }
strcpy(buf,"This is parent process write");
if ((cld_pid=fork()) == 0)
 {
     strcpy(buf,"This is child process write");
     printf("This is child process");
     printf("My PID(child) is %d\n",getpid());
     printf("My parent PID is %d\n",getppid());
     write(fd,buf,strlen(buf));
     close(fd);
     exit(0);
 }
 else
     {
   printf("This is parent process");
   printf("My PID(parent) is %d\n",getpid());
   printf("My child PID is %d\n",cld_pid);
   write(fd,buf,strlen(buf));
   close(fd);
     }
 wait(&status);
}
[root@yangzongde ch03_03]# gcc -o fork_example fork_example.c
[root@yangzongde ch03_03]# ./fork_example
This is child processMy PID(child) is 13795
My parent PID is 13794
This is parent processMy PID(parent) is 13794
My child PID is 13795
[root@yangzongde ch03_03]#
```

4．进程执行

用 fork 函数创建子进程后，子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程完全由新程序替代，而新程序则从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后的进程 ID 并未改变。exec 只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

有 6 种不同的 exec 函数可供使用，具体是 execl()、execle()、execlp()、execv()、execve()、execvp()，它们常常被统称为 exec 函数。这些 exec 函数都是 UNIX 进程控制原语。用 fork 可以创建新进程，用 exec 可以执行新的程序。exit 函数和两个 wait 函数处理终止和等待终止。这些是需要的基本进程控制原语。在后面各节中将使用这些原语构造另外一些如 popen 和 system 之类的函数。

```
#include<unistd.h>
int execl(const char *pathname,const char *arg0,.../*(char*)0*/);
int execv(const char *pathname,char *const argv[]);
int execle(const char *pathname,const char *arg0, ...
/*(char *)0,char *const envp []*/);
int execve(const char *pathname,char *const argv[], char *const envp[]);
int execlp(const char *filename,const char *arg0,.../*(char *)0*/);
int execvp(const char *filename,char *const argv[]);
```

（1）execl()

功能：execl()用来执行参数 path 字符串所代表的文件路径（绝对路径），接下来的参

数代表执行文件时传递的 agrv，最后一个参数必须以空指针结束。

定义函数：int execl(const char *path,const char *arg,...)。

返回值：如果执行成功，不返回，否则失败返回–1，失败代码存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execute_example.c
#include<unistd.h>
main()
{
      execl("/bin/ls","ls","-l","/home",(char *)0);
}
[root@yangzongde ch03_04]# gcc -o execute_example execute_example.c
[root@yangzongde ch03_04]# ./execute_example
总用量 9448
drwxr-xr-x  10 root    root       4096  6ÔÂ 15 16:58 book
drwxr-xr-x   7 2840    562        4096 2002-06-15  Disk1
drwxrwxr-x   3 2840    562        4096 2002-05-14  Disk2
drwxrwxr-x   3 2840    562        4096 2002-05-14  Disk3
drwxr-xr-x   2 root    root       4096  3ÔÂ 27 13:45 document
drwx------  28 ftp1    users      4096  5ÔÂ 13 19:19 ftp1
drwxr-xr-x   4 ftp1    users      4096  3ÔÂ 30 09:28 gui
drwx------   2 root    root      16384  3ÔÂ  3 18:46 lost+found
drwx------  13 security_test2 security_test2    4096  3ÔÂ 27 13:46
oracle
drwxr-xr-x   4 root    root       4096  3ÔÂ 12 10:50 Program
drwx------   3 security_test1 security_test1    4096  3ÔÂ 27 13:54
security _test1
drwx------   2 security_test2 security_test2    4096  3ÔÂ 26 15:32 security
_test2
drwxr-xr-x   2 root    root       4096  3ÔÂ 27 10:03 software
drwxr-xr-x   4 root    root       4096  4ÔÂ  4 15:29 test
drwxr-xr-x   5 root    root       4096  5ÔÂ 13 19:17 work
-rw-------   1 root    root    9577556  4ÔÂ  8 17:47 wxX11-2.6.3.tar.gz
drwx------  20 yangzongde yangzongde    4096  4ÔÂ  3 09:58 yangzongde
```

（2）execle()

功能：execle()用来执行参数 path 字符串所代表的文件路径（绝对路径），接下来的参数代表执行文件时传递的 agrv，最后一个参数必须指向一个新的环境变量数组，此新的环境变量数组即成为新执行程序的环境变量。

定义函数：int execle(const char *path,const char *arg,...,char* const envp[])。

返回值：如果执行成功，不返回，否则失败返回–1，失败原因存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execle.c
#include<unistd.h>
main(int argc,char *argv[],char *env[])
{
      execle("/bin/ls","ls","-l","/home",(char *)0,env);
}
[root@yangzongde ch03_04]# gcc -o execle execle.c
[root@yangzongde ch03_04]# ./execle
×ÜÓÃÁ¿ 9448
drwxr-xr-x  10 root    root       4096  6ÔÂ 15 16:58 book
```

```
drwxr-xr-x    7 2840    562         4096 2002-06-15  Disk1
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk2
```

（3）execlp()

功能：execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后执行该文件，接下来的参数代表执行文件时传递的 agrv[0]，最后一个参数必须用空指针 NULL。

定义函数：int execlp(const char *path,const char *arg,...)。

返回值：如果执行成功，不返回，否则失败返回–1，失败代码存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execlp.c
#include<unistd.h>
main()
{
       execlp("ls","ls","-l","/home",(char *)0);
}

[root@yangzongde ch03_04]# gcc -o execlp execlp.c
[root@yangzongde ch03_04]# ./execlp
总用量 9448
drwxr-xr-x   10 root    root        4096  6ÔÂ 15 16:58 book
drwxr-xr-x    7 2840    562         4096 2002-06-15  Disk1
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk2
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk3
drwxr-xr-x    2 root    root        4096  3ÔÂ 27 13:45 document
```

（4）execv()

功能：execv()用来执行参数 path 字符串所代表的文件路径，第二个参数利用数组指针来传递给执行文件。

定义函数：int execv(const char *path, char const *arg[])。

返回值：如果执行成功，不返回，否则失败返回–1，失败代码存于 errno 中。

库头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execv.c
#include<unistd.h>
main()
{
       char *argv[]={"ls","-l","/home",(char *)0};
       execv("/bin/ls",argv);
}
[root@yangzongde ch03_04]# gcc -o execv execv.c
[root@yangzongde ch03_04]# ./execv
总用量 9448
drwxr-xr-x   10 root    root        4096  6ÔÂ 15 16:58 book
drwxr-xr-x    7 2840    562         4096 2002-06-15  Disk1
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk2
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk3
drwxr-xr-x    2 root    root        4096  3ÔÂ 27 13:45 document
```

（5）execve()

功能：execve()用来执行参数 filename 字符串所代表的文件路径，第二个参数利用数

组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。

定义函数：int execve(const char *filename, char *const arg[],char *const envp[])。

返回值：如果执行成功，不返回，否则失败返回–1，失败代码存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execve.c
#include<unistd.h>
main()
{
       char *argv[]={"ls","-l","/home",(char *)0};
       char *envp[]={"PATH=/bin",0};
       execve("/bin/ls",argv,envp);
}

[root@yangzongde ch03_04]# gcc -o execve execve.c
[root@yangzongde ch03_04]# ./execve
total 9448
drwxr-xr-x    7 2840    562         4096 Jun 15  2002 Disk1
drwxrwxr-x    3 2840    562         4096 May 14  2002 Disk2
drwxrwxr-x    3 2840    562         4096 May 14  2002 Disk3
drwxr-xr-x    4 root    root        4096 Mar 12 10:50 Program
```

（6）execvp()

功能：execvp()从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行此文件，第二个参数 argv 传递给要执行的文件。

定义函数：int execvp(const char *filename, char *const arg[])。

返回值：如果执行成功，不返回，否则失败返回–1，失败原因存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_04]# cat execvp.c
#include<unistd.h>
main()
{
       char *argv[]={"ls","-l","/home",0};
       execvp("ls",argv);
}
[root@yangzongde ch03_04]# gcc -o execvp execvp.c
[root@yangzongde ch03_04]# ./execvp
×ÜÓÃÁ¿ 9448
drwxr-xr-x   10 root    root        4096  6ÔÂ 15 16:58 book
drwxr-xr-x    7 2840    562         4096 2002-06-15  Disk1
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk2
drwxrwxr-x    3 2840    562         4096 2002-05-14  Disk3
drwxr-xr-x    2 root    root        4096  3ÔÂ 27 13:45 document
```

5．退出进程

（1）wait()

功能：wait()函数会暂停目前进程的执行，直到有信号来到或子进程结束。如果调用 wait()时子进程已经结束，则 wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一起返回，如果不需要结束状态值，则参数 status 可以设置为 NULL。

定义函数：pid_t wait(int *status)。

返回值：如果执行成功则返回子进程识别码 PID，如果有错误发生则返回–1，失败原因存于 errno。

头文件：#include<sys/types.h>，#include<sys/wait.h>

（2）waitpid()

功能：waitpid()函数会暂停目前进程的执行，直到有信号来到或子进程结束，如果调用 wait()时子进程已经结束，则 waitpid()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一块返回，如果不需要结束状态值，则参数 status 可以设置为 NULL。参数 pid 为将等待的子进程 PID，其数值定义如下。

pid<–1：等待进程组识别码为 pid 绝对值的任何子进程；

pid=–1：等待任何子进程，相当于 wait()；

pid=0：等待进程组识别码与目前进程相同的任何子进程；

pid>0：等待任何子进程识别码为 pid 的子进程。

参数 options 可以为 0 或者以下组合。

WHOHANG：如果没有任何已经结束的子进程则马上返回，不予等待；

WUNTRACED：如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会。

子进程的结束状态写于 status 中，以下是常见的几个宏。

WIFEXITED(status)：如果子进程正常结束则为非值；

WEXITSTATUS(status)：取得子进程由 exit()返回的结束代码，一般会先用 WIFEXITED 判断是否正常结束然后才能作用；

WIFSIGNALED(status)：如果子进程是因为信号而结束则此宏值为真；

WTERMSIG(status)：取得子进程因信号而中止的信号代码，一般会先用 WIFSIGNALED 判断后才用此宏。

WIFSTOPPED(status)：如果子进程处于暂停执行情况则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况。

WSTOPSIG(status)：取得引发进程暂停的信号代码，一般会先用 WIFSTOPPED 判断后才使用此宏。

定义函数：pid_t waitpid(pid_t pid,int *status,int options)。

返回值：如果执行成功则返回子进程识别码 PID，如果有错误发生则返回–1，失败原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/wait.h>

```
[root@yangzongde ch03_05]# cat wait_example.c
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
main()
{
       pid_t pid;
       int status,i;
       if(fork()==0)
       {
              printf("this is the child process pid=%d\n",getpid());
```

```
                exit(5);
        }else
        {
        sleep(1);
        printf("this is the parent process wait for child...\n");
        pid=wait(&status);
        i=WEXITSTATUS(status);
        printf("the child's pid is %d,exit status is %d\n",pid,i);
        }
}
[root@yangzongde ch03_05]# gcc –o wait_example wait_example.c
[root@yangzongde ch03_05]# ./wait_example
this is the child process pid=14914
this is the parent process wait for child...
the child's pid is 14914,exit status is 5
[root@yangzongde ch03_05]#
```

（3）exit()

功能：exit()用来正常结束目前进程的执行，并把参数 status 返回给父进程，而进程所有的缓冲区数据会自动写回并关闭文件。

定义函数：void exit(status)。

返回值：如果执行成功，不返回，否则失败返回–1，失败原因存于 errno 中。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_06]# cat exit_example.c
#include<stdlib.h>
main()
        {
        printf("output begin\n");
        printf("content in buffer\n");
        exit(0);
        }
[root@yangzongde ch03_06]# gcc -o exit_example exit_example.c
[root@yangzongde ch03_06]# ./exit_example
output begin
content in buffer
```

（4）_exit()

功能：_exit()用来正常结束目前进程的执行，并把参数 status 返回给父进程，并关闭文件，此函数调用后不会返回，而是会传递 SIGCHLD 信号给父进程，父进程可以由 wait()函数取得子进程结束状态。

定义函数：void _exit(status)。

返回值：无。_exit()不会处理标准 I/O 缓冲区，如果要更新需要调用 exit()。

头文件：#include<unistd.h>

```
[root@yangzongde ch03_06]# cat _exit_example.c
#include<stdlib.h>
main()
        {
        printf("output\n");
        printf("content in buffer");
        _exit(0);
        }
```

```
[root@yangzongde ch03_06]# gcc -o _exit_example _exit_example.c
[root@yangzongde ch03_06]# ./_exit_example
output
[root@yangzongde ch03_06]#
```

（5）on_exit()

功能：on_exit()用来设置一个程序正常结束前调用的函数，当程序通过调用 exit()或者从 main 中返回时，参数 function 所指定的函数会被先调用，然后才真正由 exit()结束程序，参数 arg 指针会传给 function 函数。

定义函数：int on_exit(void(*function)(int void *),void *arg)。

返回值：如果执行成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<stdlib.h>

```
[root@yangzongde ch03_06]# cat on_exit_example.c
#include<stdlib.h>
void test_exit(int status,void *arg)
{
       printf("before exit()!\n");
       printf("exit %d\n",status);
       printf("arg=%s\n",(char *)arg);

}
main()
{
       char *str="test";
       on_exit(test_exit,(void *)str);
       exit(4321);
}
[root@yangzongde ch03_06]# gcc -o on_exit_example on_exit_example.c
[root@yangzongde ch03_06]# ./on_exit_example
before exit()!
exit 4321
arg=test
[root@yangzongde ch03_06]#
```

6．跟踪进程

跟踪进程的实现函数为 ptrace()。

功能：ptrace()提供数种服务让父进程来对子进程进行追踪，被追踪子进程处于挂起状态时，父进程可以存取子进程的内存空间，参数 request 用来要求系统提供的服务有以下几种。

PTRACE_TRACEME：此进程将由父进程追踪；

PTRACE_PEEKTEXT：从 pid 子进程的内存地址 addr 中读取一个 word；

PTRACE_PEEKDATA：同 PTRACE_PEEKTEXT；

PTRACE_PEEKUSR：从 pid 子进程的 USER 区域内地址 addr 中读取一个 word；

PTRACE_POKETEXT：将 data 写入 pid 子进程的内存地址 addr 中；

PTRACE_POKEDATA：同 PTRACE_POKETEXT；

PTRACE_POKEUSR：将 data 写入 pid 子进程的 user 区域内地址 addr 中；

PTRACE_SYSCALL：继续 pid 子进程的执行；

PTRACE_CONT：同 PTRACE_SYSCALL；

PTRACE_SINGLESTEP：设置 pid 子进程的单步追踪旗标；
PTRACE_ATTACH：附带 pid 子进程；
PTRACE_DETACH：分离 pid 子进程。
定义函数：int ptrace(int request,int pid ,int addr,int data)。
返回值：如果执行成功则返回 0，执行失败则返回–1，失败原因存于 errno 中。
头文件：#include<sys/ptrace.h>

```
[root@yangzongde ch03_07]# cat ptrace_example.c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<signal.h>
#include<sys/ptrace.h>
#include<asm/ptrace.h>

#define attach(pid)    ptrace(PRTACE_ATTACH,pid,(char *)1,0)
#define detach(pid)    ptrace(PTRACE_DETACH,pid,(char *)1,0)

#define trace_sys(pid)  ptrace(PTRACE_SYSCALL,pid,(char *)1,0)
#define trace_me(pid)   ptrace(PTRACE_TRACEME,0,(char *)1,0)

long get_regs(int pid,int reg)
{
       long val;
       val=ptrace(PTRACE_PEEKUSER,pid,(char *)(4*reg),0);
       if(val==-1)
       perror("ptrace(PTRACE_PEEKUSER,......)");
       return val;
}
void print_regs(pid)
{
       struct pt_regs Regs;
       Regs.orig_eax=get_regs(pid,ORIG_EAX);
       Regs.eax=get_regs(pid,EAX);
       Regs.eip=get_regs(pid,EIP);
       if(Regs.orig_eax!=0x4)
              return;
       printf("ORIG_EAX=0x%x,EAX=0x%x,",Regs.orig_eax,Regs.eax);
       printf("EIP=0x%x\n",Regs.eip);
}
int trace_syscall(pid)
{
       int value;
       value=trace_sys(pid);
       if(value<0) perror("ptrace");
       return value;
}
main (int argc,char *argv[])
{
       int pid;
       int p,status;
       if(argc<2)
       {
              printf("usage:%s program\n",argv[0]);
```

```
            return;
        }
        if(!(pid=fork()))
        {
            if(trace_me()<0) perror("ptrace");
            execl(argv[1],"traceing",0);
        }else
        {
            printf("ptrace %s(pid=%d)...\n",argv[1],pid);
            sleep(1);
            do
            {
                trace_syscall(pid);
                p=wait(&status);
                if(WIFEXITED(status))
                    {
                    printf("child exit()\n");
                    exit(1);
                    }
                if(WIFSIGNALED(status))
                    {
                    printf("child exit(),beacuse a signal\n");
                    exit(1);
                    }
                print_regs(pid);
            }while(1);
        }
}
[root@yangzongde ch03_07]# gcc -o ptrace_example ptrace_example.c
[root@yangzongde ch03_07]# ./ptrace_example /bin/echo
ptrace /bin/echo(pid=5418)...
ORIG_EAX=0x4,EAX=0xffffffda,EIP=0xffffe002

ORIG_EAX=0x4,EAX=0x1,EIP=0xffffe002
child exit()
[root@yangzongde ch03_07]#
```

2.1.3　进程通信与同步

在一个大型的 Linux 应用系统中，各进程间通信显得十分重要。Linux 下的进程通信手段绝大多数是从 UNIX 平台的进程继承而来的，AT&T 的贝尔实验室和 BSD 在进程通信方面各有自己的侧重，Linux 将两者继承下来。Linux 下进程间通信的几个主要手段如下。

管道：管道有无名管道和有名管道两种，管道只能实现具有亲缘关系的进程间的通信，有名管道克服管道没有名字的限制；

信号：信号为通知进程某一事件发生，从而触发进程执行；

消息队列：消息队列是消息的连接表，有足够权限的进程向消息队列中添加信息，有读权限的进程可以从消息队列中读取消息；

共享内存：共享内存机制使多个进程可以访问同一块内存空间，从而快速地实现进程间的通信；

信号量：信号量主要用于同一进程中各线程之间的信息交互和同步。

1．管道

管道是 Linux 最先使用的进程通信机制之一，管道只能实现具有亲缘关系的进程间的通信，而有名管道（有名称的管道）克服了这一缺点，管道是单向的，数据只能从一端写入，另一端读取。常见的 Linux 下管理管道的函数如下。

（1）mkfifo()

功能：mkfifo()会依参数 pathname 建立特殊的 FIFO 文件，该文件必须存在，而参数 mode 为该文件的权限（mode%~umask），因此 umask 值也会影响到 FIFO 文件的权限，mkfifo()建立的 FIFO 文件其他进程都可以用读写一般文件的方式存取。当使用 open()函数打开 FIFO 文件时，O_NONBLOCK 会有影响。

- 当使用 O_NONBLOCK 时，打开 FIFO 文件来读取的操作会立刻返回，但是如果没有其他进程打开 FIFO 文件来读取，则写入的操作会返回 ENXIO 错误代码。
- 没有使用 O_NONBLOCK 旗标时，打开 FIFO 来读取的操作会等到其他进程打开 FIFO 文件来写入才正常返回。同样地，打开 FIFO 文件来写入的操作会等到其他进程打开 FIFO 文件来读取后才正常返回。

定义函数：int mkfifo(const char *pathname,mode_t mode)。

返回值：如果执行成功返回 0，否则失败返回–1，失败原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/stat.h>

```
[root@yangzongde ch03_08]# cat mkfifo_example.c
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>

#define FIFO "/tmp/fifo"
main()
{
       char buffer[80];
       int fd;
       unlink(FIFO);          //delFIFO file
       mkfifo(FIFO,0744);
       if(fork()>0)
       {
              char s[]="Hello!\n";
              fd=open(FIFO,O_WRONLY);
              write(fd,s,sizeof(s));
              close(fd);
       }else
       {
              fd=open(FIFO,O_RDONLY);
              read(fd,buffer,80);
              printf("%s",buffer);
              close(fd);
       }
}
[root@yangzongde ch03_08]# gcc -o mkfifo_example mkfifo_example.c
[root@yangzongde ch03_08]# ./mkfifo_example
[root@yangzongde ch03_08]# Hello!
ls
```

```
mkfifo_example  mkfifo_example.c
[root@yangzongde ch03_08]#
```

（2）pclose()

功能：pclose()用来关闭由 popen()所建立的管道及文件指针，参数 stream 为先前由 popen()所返回的文件指针。

定义函数：int pclose(FILE *stream)。

返回值：返回子进程的结束状态，如果有错误则返回–1，错误原因存于 errno 中。

头文件：#include<unistd.h>

（3）pipe()

功能：pipe()函数建立管道，并将文件描述词由参数 filedes 数组返回，filedes[0]为管道里的读取端，filedes[1]为管道的写入端。

定义函数：int pipe(int filedes[2])。

返回值：成功返回 0，如果有错误则返回–1，错误原因存于 errno 中。

头文件：#include<unistd.h>

父进程借管道将字符串传递给子进程。

```
[root@yangzongde ch03_08]# cat pipe_example.c
#include<unistd.h>
main()
{
       int filedes[2];
       char buffer[100];
       pipe(filedes);
       if(fork()>0)
       {//father
              char s[]="test";
              write(filedes[1],s,sizeof(s));
       }else
       {//son
              read(filedes[0],buffer,100);
              printf("the text is %s\n",buffer);
       }
}
[root@yangzongde ch03_08]# gcc -o pipe_example pipe_example.c
[root@yangzongde ch03_08]# ./pipe_example
the text is test
```

（4）popen()

功能：popen()调用 fork()产生子进程，然后从子进程中调用/bin/sh –c 来执行参数 command 的指令，参数 type 可以使用 r 代表读，w 代表写，依靠 type 的值，popen 会建立管道连接到子进程的标准输出设备或标准输入设备或者写入到子进程的标准输入设备中。此外，所有使用文件指针（FILE *）操作的函数也都可以使用，除 fclose()外。

定义函数：FILE *popen(const char *comm.and,const char *type)。

返回值：成功返回 0，如果有错误则返回–1，错误原因存于 errno 中。

头文件：#include<stdio.h>

2．信号

信号是模拟中断的一种软件处理机制，一个进程收到一个信号量就相当于收到了一个

处理器中断请求，信号是 Linux 进程通信中的异步通信机制，信号来源有以下两种。

- 硬件来源；
- 软件来源，用发送信号函数实现。这些函数特点如下所示。

（1）kill 传送信号给指定进程

功能：kill()可以用来传递参数 sig 指定的信号给参数 pid 所指定的进程，参数 pid 有以下几种情况。

pid>0：将信号传递给进程识别码为 pid 的进程；

pid=0：将信号传递给目前进程相同的进程组的所有进程；

pid=–1：将信号像广播般传递给系统内所有的进程；

pid<0：将信号传递给进程组识别码为 pid 绝对值的所有进程。

定义函数：int kill(pid_t pid,int sig)。

返回值：执行成功则返回 0，如果有错误则返回–1。

头文件：#include<sys/types.h>，#include<signal.h>

（2）raise()传递信号给目前的进程。

功能：raise()用来将参数 sig 指定的信号传递给目前的进程，相当于 kill(getpid(),sig)。

定义函数：int raise(int sig)。

返回值：执行成功则返回 0，否则返回非 0 值。

库头文件：#include<signal.h>

（3）alarm()

功能：此函数用来设置信号 SIGALRM 在经过参数 seconds 指定的时间后传送给目前进程，如果参数 seconds 为 0，则之前设置的闹钟会被取消，并将剩下的时间返回。

定义函数：unsigned int alarm(unsigned int seconds)。

返回值：返回之前闹钟的剩余时间，如果之前未设定则返回 0。

头文件：#include<unsitd.h>

（4）signal()

功能：此函数依参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数 handler 指定的函数执行。如果参数 handler 不是函数指针，则必须是以下两个常数之一。

SIG_IGN：忽略参数 signum 指定的信号；

SIG_DFL：将参数 signum 指定的信号重设为核心预设的信号处理方式。

定义函数：void(*signal(int signum,void(handler)(int)))(int)。

返回值：返回先前的信号处理函数指针，如果有错误则返回 SIG__ERR(–1)。

库头文件：#include<signal.h>

（5）singaction()

功能：singaction 依参数 signum 指定的信号编号来设置该信号的处理函数，参数 signum 可以指定 SIGKILL 和 SIGSTOP 以外的所有信号，如果参数 act 不是 NULL 指针，则用来设置新的信号处理方式。结构 sigaction 定义如下：

```
struct sigaction
{
    void (*sa_handler)(int);
    sigset_t sa_mask;
```

```
    int sa_flags;
    void (*sa_restorer) (void);
}
```

其中：

handler：此参数和 signal()的参数 handler 相同，代表新的信号处理函数；

sa_mask：用来设置在处理该信号时暂时将 sa_mask 指定的信号搁置；

sa_restorer：没有使用；

sa_flags：用来设置信号处理的其他相关操作。

定义函数：int sigaction(int signum,const struct sigaction *act,struct sigaction *oldact)。

返回值：执行成功返回 0，如果有错误则返回–1。

库头文件：#include<signal.h>

（6）sigprocmask()

功能：此函数用来改变目前的信号遮罩（mask），其操作依参数 how 来决定。

SIG_BLOCK：新的信号遮罩由目前的信号遮罩和参数 set 指定的信号遮罩作联集；

SIG_UNBLOCK：将目前的信号遮罩删除掉参数 set 指定的信号遮罩；

SIG_SETMASK：将目前的信号遮罩设置成参数 set 指定的信号遮罩。

定义函数：int sigprocmask(int how,const sigset_t *set,sigset_t *oldest)。

返回值：执行成功返回 0，如果有错误返回–1。

头文件：#include<signal.h>

（7）sigpending()

功能：将被搁置的信号集合由参数 set 指针返回。

定义函数：int sigpending(sigset_t *set)。

返回值：执行成功返回 0，如果有错误返回–1。

库头文件：#include<signal.h>

（8）siguspend()

功能：将目前的信号遮罩暂时换成参数 mask 所指定的信号遮罩，并将此进程暂停，直到有信号到来才恢复原来的信号遮罩，继续程序执行。

定义函数：int sigsuspend(const sigset_t *mask)。

返回值：返回–1。

头文件：#include<signal.h>

3．消息队列

消息队列是一个消息链接，程序员可以把消息看做一个记录，它具有特定的格式及特定的优先级。对消息队列有写权限的进程可以向消息队列按照一定的规则添加新的消息，而对消息队列有读权限的进程则可以从消息队列中读取信息。目前主要的消息队列有两种类型：POSIX 消息队列和系统 V 消息队列。

每一个消息队列都有一个队列头，队列头中包含了消息队列的绝大多数信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等。

进程可用 megsnd( )系统调用来发送一个消息，并将它链入消息队列的尾部。该系统调用的语法格式如下：

```
int msgsnd(msgid，msgp，msgsz，msgflg)
int msgid;
struct msgbuf * msgp;
int msgsz，msgflg;
```

其中，msgid 是由 msgget 返回的消息队列描述符；msgp 指向包含这条消息的结构，关于消息队列处理主要函数定义如下。

（1）msgget()

功能：msgget()用来取得参数 key 所关联的信息队列识别代号。如果参数 key 为 IPC__PRIVATE 则会建立新的消息队列，如果 KEY 不是 IPC_PRIVATE，也不是已经建立的 IPC key，系统会检查参数 msgFlash 是否有 IPC_CREAT 位来决定建立 IPC key 为 key 的信息队列。如果参数 msgflag 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立信息队列，则表示信息队列已经存在。此外，参数 msgflag 也用来决定信息队列的存取权限，其值相当于 open()的参数 mode 用法。

定义函数：int msgget(key_t key,int msgflg)；

返回值：若成功则返回信息队列识别代号，否则返回–1。

头文件：#include<sys/tpye.h>，#include<sys/ipc.h>，#include<sys/msg.h>

（2）msgrcv()

功能：msgrcv()用来从参数 msqid 指定的信息队列读取信息出来，然后存放到参数 msgp 所指定的数据结构中，msgbuf 的数据结果定义如下：

```
struct msgbuf
{
    long mtype;//信息种类
    char mtext[1];//信息数据
}
```

参数 msgsz 为信息数据的长度，即 mtext 参数的长度；

参数 msgtyp 是用来指定所要读取的信息种类。msgtyp=0 返回队列内第一项信息；msgtyp>0 返回队列内第一项 msgtyp 与 mtype 相同的信息；msgtyp<0 返回队列内第一项 mtype 小于或等于 msgtyp 绝对值的信息。

参数 msgflg 可以设置成 IPC_NOWAIT，意思是如果队列内没有信息可读，则不要等待，立即返回 ENOMSG。如果 msgflg 设置成 MSG_NOERROR，则信息大小超过参数 msgsz 时会被截断。

定义函数：int msgrcv(int msqid,struct msgbuf *msgp,int msgsz,long msgtyp,int msgflg)；

返回值：若成功则返回实际读取到的信息数据长度，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/tpye.h>，#include<sys/ipc.h>，#include<sys/msg.h>

（3）msgsnd()

功能：msgsnd()用来将参数 msgp 指定的信息送到参数 msqid 的信息队列中，参数 msgp 为 msgbuf 结构。其定义如下：

```
struct msgbuf
{
    long mtype;         //信息种类
    char mtext[1];      //信息数据
```

}

参数 msgsz 为信息数据的长度，即 mtext 参数的长度；

参数 msgflg 可以设置成 IPC_NOWAIT，意思是如果队列已满或者有其他情况无法马上送入信息，则立刻返回 EAGIN。

定义函数：int msgsnd(int msqid,stuct msgbuf *msgp,int msgsz,int msgflg)；

返回值：若成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/tpye.h>，#include<sys/ipc.h>，#include<sys/msg.h>

（4）msgctl()

功能：msgctl()提供了几种方式来控制信息队列的运作，参数 msgid 为欲处理的信息队列识别代码，参数 cmd 为欲执行的操作。

定义函数：int msgctl(int semid, int cmd,struct msqid_ds *buf)；

返回值：若成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/tpye.h>，#include<sys/ipc.h>，#include<sys/msg.h>

4．共享内存

共享内存是最便捷、速率最快的进程通信方式，共享内存方式将同一块物理内存分别映射到 A、B 两个进程逻辑空间，由于多个进程共享同一块内存空间，因此需要其他同步机制协同工作，如互斥锁和信号量。

（1）shmat()

功能：shmat()用来将参数 shmid 所指的共享内存和目前进程连接，参数 shmid 为欲连接的共享内存识别代码，而参数 shmaddr 有以下几种情况。

0：核心自动选择一个地址；

不为 0，但参数 shmflg 也没有指定 SHM_RND 旗标：则以参数 shmaddr 为链接地址；

不为 0，但参数 shmflg 设置了 SHM_RND 旗标：则参数 shmaddr 会自动调整为 SHMLBA 的整数位。

定义函数：void *shmat(int shmid,const void *shmaddr,int shmflg);

返回值：若“是”，返回已经连接好的地址，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/shm.h>

（2）shmctl()

功能：shmctl()提供了几种方式来控制共享内存的操作，参数 shmid 为欲处理的共享内存识别码，参数 cmd 为欲控制的操作。其中 shmid_ds 的结构定义如下：

```
struct shmid_ds
{
struct ipc_perm shm_perm;
int shm_segsz;                  //共享内存的大小
time_t shm_atime;               //最后一次 attach 此共享内存的时间
time_t shm_dtime;               //最后一次 detach 此共享内存的时间
time_t shm_ctime;               //最后一次更改此共享内存结构的时间
unsigned short shm_cpid;        //建立此共享内存的进程识别码
unsigned short shm_lpid;        //最后一个操作共享内存的进程识别码
short shm_nattch;
unsigned long *shm_pages;
struct shm_desc *attaches;
}
```

定义函数：int shmctl(int shmid,int cmd,struct shmid_ds *buf)；

返回值：若成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/shm.h>

（3）shmdt detach 共享内存

功能：shmdt()用来将先前用 shmat()连接好的共享内存脱离目前进程，参数 shmaddr 为 shmat 返回的共享内存地址。

定义函数：int shmdt(const void *shmaddr)；

返回值：若成功则返回 0，否则返回–1，错误原因存在于 errno 中。

头文件：#include<sys/types.h>，#include<sys/shm.h>

（4）shmget 配置共享内存

功能：shmget()用来取得参数 key 所关联的共享内存识别代号。

定义函数：int shmget(key_t key,int size,int shmflg)；

返回值：若成功，返回共享内存识别码，否则返回–1，错误原因存于 errno 中。

库头文件：#include<sys/types.h>，#include<sys/shm.h>

5．信号量

（1）semget()

功能：semget()用来取得参数 key 所关联的信号识别码。如果参数 key 为 IPC_PRIVATE，则会建立新的信号队列，参数 nsems 为信号集合的数目。如果 key 不为 IPC_PRIVATE，也不是已经建立的信号队列 IPC key，那么系统会视参数 semflg 是否有 IPC_CREAT 位来决定 IPC key 为参数 key 的信号队列。

如果参数 semflg 包含了 IPC_CREAT 和 IPC_EXCL 位，而无法依参数 key 来建立信号队列，则表示队列已经存在。

定义函数：int semget(key_t key,int nsems,int semflg);

返回值：如果成功则返回信号队列识别码，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/sem.h>

（2）semctl()

功能：semctl()提供了几种方式来控制信号队列的操作。参数 semid 为欲处理的信号队列识别码，参数 cmd 为欲控制的操作。union semun 定义如下：

```
union semun
{
int val;                          //SETVAL 用的 semval 值
struct semid_ds *buf;             //指向 IP__STAT 或 IPC_set 用的 semid_ds 结构
unsigned short int *array;        //GETALL 或 SETAL 用的数组
sturct seminfo *_buf;
}
```

定义函数：int semctl(int semid,int semnum,int cmd,union semun arg)；

返回值：若成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/sem.h>，#include<sys/ipc.h>

（3）semop()

功能：semop()函数中参数 semid 为欲处理的信号队列识别代码，参数 sops 指向结构

sembuf，其结构定义如下：

```
struct sembuf
{
short int sem_num;          //欲处理的信号编码，0 代表第一个
short int sem_op;
short int sem_flg;
}
```

定义函数：int semop(int semid,stuct sembuf *sops,unsigned nsops);

返回值：若成功则返回 0，否则返回–1，错误原因存于 errno 中。

头文件：#include<sys/types.h>，#include<sys/sem.h>，#include<sys/ipc.h>

2.1.4　线程基本操作

1．创建线程

线程创建函数 pthread_create 用来创建一个新的线程。其函数原型如下所示：

```
int pthread_create (pthread_t * thread_id, __const pthread_attr_t * __attr,
void *(*__start_routine) (void *),void *__restrict __arg)
```

线程创建函数第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里由于函数 thread 不需要参数，所以最后一个参数设为空指针。第二个参数也设为空指针，这样将生成默认属性的线程。当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败，常见的错误返回代码为 EAGAIN 和 EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

2．等待指定的线程结束

pthread_join 函数用来等待一个线程的结束。函数原型为：

```
int pthread_join (pthread_t __th, void **__thread_return)
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，处于等待状态的线程资源被收回。

3．获得父线程 ID

```
pthread_t pthread_self (void)
```

4．测试两个线程号是否相同

```
int pthread_equal (pthread_t __thread1, pthread_t __thread2)
```

5．线程退出

pthread_exit 函数

一个线程的结束有两种途径，一种是函数结束了，调用它的线程也就结束了；另一种方式是通过函数 pthread_exit 来结束。它的函数原型为：

```
void pthread_exit (void *__retval)
```

该函数的参数是函数的返回代码，只要 pthread_join 中的第二个参数 thread_return 不是 NULL，这个值将被传递给 thread_return。需要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用 pthread_join 的线程则返回错误代码 ESRCH。

6．线程通信相关函数

使用互斥锁可实现线程间数据的共享和通信，但互斥锁的一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其他的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线程间的同步。

（1）pthread_cond_init 函数

条件变量的结构为 pthread_cond_t，函数 pthread_cond_init()用来初始化一个条件变量。它的原型为：

```
int pthread_cond_init (pthread_cond_t * cond, __const pthread_condattr_t
* cond_attr)
```

其中，cond 是一个指向结构 pthread_cond_t 的指针，cond_attr 是一个指向结构 pthread_condattr_t 的指针。结构 pthread_condattr_t 是条件变量的属性结构，与互斥锁一样可以用它来设置条件变量是进程内可用还是进程间可用，默认值是 PTHREAD_PROCESS_PRIVATE，即此条件变量被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为 pthread_cond_destroy(pthread_cond_t cond)。

（2）pthread_cond_wait 函数

该函数线程阻塞在一个条件变量上。它的函数原型为：

```
extern int pthread_cond_wait (pthread_cond_t *__restrict__cond,
pthread_ mutex_t *__restrict __mutex)
```

线程解开 mutex 指向的锁并被条件变量 cond 阻塞。线程可以被函数 pthread_cond_signal 和函数 pthread_cond_broadcast 唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出，例如一个变量是否为 0 等，这一点从后面的例子中可以看到。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，等待被下一次唤醒。这个过程一般用 while 语句实现。

（3）pthread_cond_timedwait 函数

另一个用来阻塞线程的函数是 pthread_cond_timedwait()，它的原型为：

```
extern int pthread_cond_timedwait __P ((pthread_cond_t
*__cond,pthread_ mutex_t *__mutex, __const struct timespec *__abstime))
```

它比函数 pthread_cond_wait()多了一个时间参数，经历 abstime 段时间后，即使条件变量不满足，阻塞也被解除。

（4）pthread_cond_signal 函数

它的函数原型为：

```
extern int pthread_cond_signal (pthread_cond_t *__cond)
```

它用来释放被阻塞在条件变量 cond 上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤醒是由线程的调度策略所决定的。需要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件满足信号又可能在测试条件和调用 pthread_cond_wait 函数之间被发出，从而造成无限制的等待。

（5）互斥量初始化

它的函数原型为：

```
pthread_mutex_init (pthread_mutex_t *,__const pthread_mutexattr_t *)
```

（6）销毁互斥量

它的函数原型为：

```
int pthread_mutex_destroy (pthread_mutex_t *__mutex)
```

（7）再试一次获得对互斥量的锁定（非阻塞）

它的函数原型为：

```
int pthread_mutex_trylock (pthread_mutex_t *__mutex)
```

（8）锁定互斥量（阻塞）

它的函数原型为：

```
int pthread_mutex_lock (pthread_mutex_t *__mutex)
```

（9）解锁互斥量

它的函数原型为：

```
int pthread_mutex_unlock (pthread_mutex_t *__mutex)
```

2.1.5　简单的多线程编程

本程序为著名的生产者—消费者问题模型的实现，主程序中分别启动生产者线程和消费者线程。生产者线程不断顺序地将 0 到 1000 的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区读取数据。流程图如图 2-2 所示。

```
[root@yangzongde ch03_11_pthread]# cat pthread.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pthread.h"

#define BUFFER_SIZE 16

/* Circular buffer of integers. */
struct prodcons {
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  pthread_cond_t notempty;     /* signaled when buffer is not empty */
  pthread_cond_t notfull;      /* signaled when buffer is not full */
```

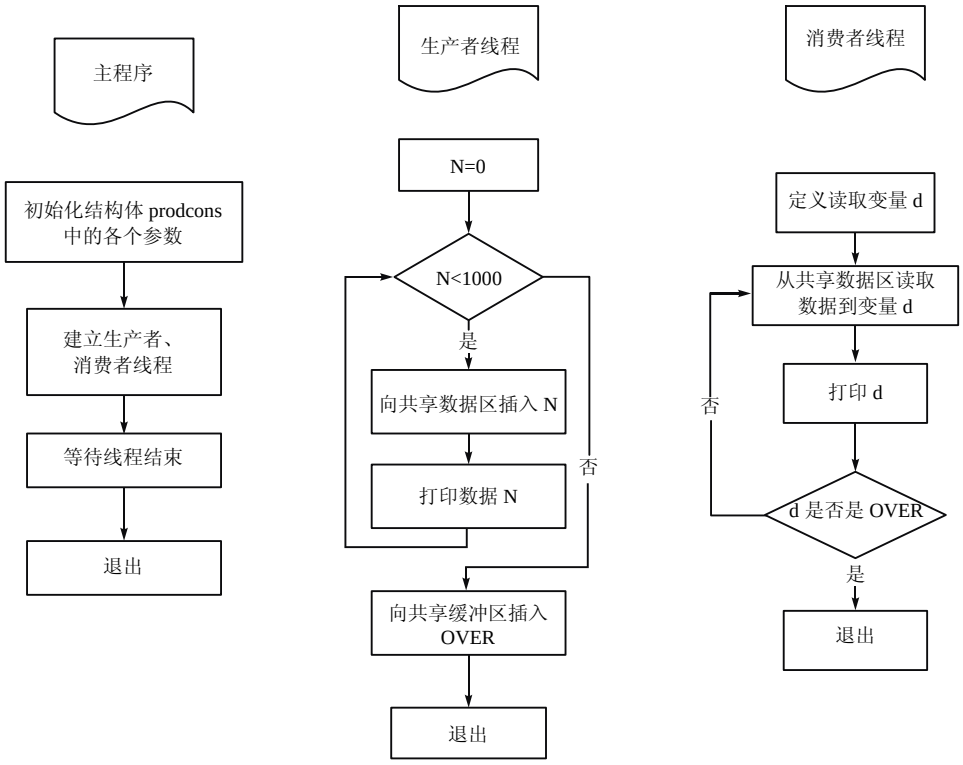


图 2-2　数据处理流程图

```
};

/*-------------------------------------------------------------*/
/* Initialize a buffer */
void init(struct prodcons * b)
{
  pthread_mutex_init(&b->lock, NULL);
  pthread_cond_init(&b->notempty, NULL);
  pthread_cond_init(&b->notfull, NULL);
  b->readpos = 0;
  b->writepos = 0;
}
/*-------------------------------------------------------------*/
/* Store an integer in the buffer */
void put(struct prodcons * b, int data)
{
       pthread_mutex_lock(&b->lock);

       while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
              printf("wait for not full\n");
       pthread_cond_wait(&b->notfull, &b->lock);
       }

       b->buffer[b->writepos] = data;
       b->writepos++;
       if (b->writepos >= BUFFER_SIZE) b->writepos = 0;

       pthread_cond_signal(&b->notempty);
```

```
        pthread_mutex_unlock(&b->lock);
        sleep(1);
}
/*-------------------------------------------------------*/
int get(struct prodcons * b)
{
        int data;
        pthread_mutex_lock(&b->lock);

        /* Wait until buffer is not empty */
        while (b->writepos == b->readpos) {
        printf("wait for not empty\n");
                pthread_cond_wait(&b->notempty, &b->lock);
        }


        data = b->buffer[b->readpos];
        b->readpos++;
        if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
        /* Signal that the buffer is now not full */
        pthread_cond_signal(&b->notfull);

        pthread_mutex_unlock(&b->lock);
        return data;
        sleep(1);
}
/*-------------------------------------------------------*/
#define OVER (-1)
struct prodcons buffer;
/*-------------------------------------------------------*/
void * producer(void * data)
{
        int n;
        for (n = 0; n < 1000; n++) {
        printf(" put-->%d\n", n);
        put(&buffer, n);
        }
  put(&buffer, OVER);
  printf("producer stopped!\n");
  return NULL;
}
/*-------------------------------------------------------*/
void * consumer(void * data)
{
  int d;
  while (1) {
    d = get(&buffer);
    if (d == OVER ) break;
    printf("            %d-->get\n", d);
  }
  printf("consumer stopped!\n");
  return NULL;
}
/*-------------------------------------------------------*/
int main(void)
{
```

```
        pthread_t th_a, th_b;
        void * retval;

        init(&buffer);
        pthread_create(&th_a, NULL, producer, 0);
        pthread_create(&th_b, NULL, consumer, 0);

        pthread_join(th_a, &retval);
        pthread_join(th_b, &retval);

        return 0;
}
```

编译程序

```
[root@yangzongde ch03_11_pthread]# gcc -o pthread pthread.c -lpthread
```

运行程序

```
[root@yangzongde ch03_11_pthread]# ./pthread
 put-->0
           0-->get
wait for not empty
 put-->1
           1-->get
wait for not empty
 put-->2
           2-->get
wait for not empty
 put-->3
           3-->get
wait for not empty
[root@yangzongde ch03_11_pthread]#
```

2.2 文件系统结构和类型

本节重点介绍几种常见的文件系统类型，包括 FAT 文件、RAMFS、JFFS/JFFS2、YAFFS、EXT2/3 和/proc。

FAT 文件系统是早期 Windows 平台下使用的，此文件系统简单、易实现，目前在部分嵌入式设备中仍然可以采用。RAMFS 是一个内存文件系统，工作在 Linux 的虚拟文件系统 VFS 层。JFFS/JFFS2 是一种用于 Nor Flash 的文件系统；YAFFS 是一种用于 NAND Flash 的可读可写文件系统；EXT2/3 是标准 Linux 环境下使用的文件系统；/proc 是一个内核使用的文件系统，是一个虚拟的文件系统，它通过文件系统接口实现对内核的访问，输出系统的运行状态。

2.2.1 FAT 文件系统

每个 FAT（File Allocation Table）文件系统由 4 部分组成，这些基本区域按如下顺序排列分别为：

- 保留区（Reserved Region）。
- FAT 区（FAT Region）。
- 根目录区（Root Directory Region，FAT32 卷没有此域）。
- 文件和目录数据区（File and Directory Data Region）。

1．保留区

BPB（BIOS Parameter Block）是 FAT 文件系统中第一个重要的数据结构，它位于该 FAT 卷的第一个扇区，同时也属于 FAT 文件系统基本区域的保留区。这个扇区又叫做“启动扇区”、“保留扇区”、“0 扇区”。这是 FAT 文件系统中第一个让人感到迷惑的地方，对于 MS-DOS 1.x 的版本，启动扇区中并没有 BPB。FAT 文件系统的最早期版本只有两种不同的格式：用于单面或双面的 360KB 的 5 寸软盘。这两种格式是通过 FAT 的第一个字节（FAT[0]的低 8 位）来区分的。

在 MS-DOS 2.x 以后，启动扇区里增加了 BPB 用于区分磁盘介质，同时不再支持老的磁盘介质区分方式（用 FAT 的第一个字节来区分），所有的 FAT 文件系统卷必须在启动扇区中包含 BPB。这又是一个迷惑人的地方，BPB 究竟是什么样的？在 MS-DOS 2.x 的定义中，每个 FAT 卷的扇区数不能多于 65536（每个扇区 512 字节的话最多 32MB），这一限定是由于定义“总扇区数”的变量本身是一个 16-bit 的数据类型。这一限制在 MS-DOS 3.x 中有所改进，它使用一个 32-bit 的变量来存储“总扇区数”。

在 Windows 95 操作系统，确切地说应该是在 OSR2(OEM Service Release 2)出现的时候，BPB 的内容有了新的变化，在这一版本中引入了新的 FAT 类型——FAT32。在 FAT16 中，由于 FAT 表的大小限制了有效的簇数（Cluster），同时也就限制了磁盘空间的大小，如果每个扇区为 512 字节的话，那么 FAT16 格式只能支持到 2GB。FAT32 的引入改变了这一状况，不再需要增加分区来管理大于 2GB 的硬盘。

FAT32 的 BPB 内容与 FAT12/FAT16 的内容在 BPB_ToSet32 区域以前完全一致，而从偏移量 36 开始，它们的内容有所区别，具体内容要看 FAT 类型为 FAT12/FAT16 还是 FAT32（后面的内容会提到如何区分 FAT 格式），这点保证了在启动扇区中包含一个完整的 FAT12/FAT16 或 FAT32 的 BPB 内容，这么做是为了达到最好的兼容性，同时也为了保证所有的 FAT 文件系统驱动程序能正确地识别和驱动不同的 FAT 格式，并让它们良好地工作，因为它们包含了现有的全部内容。表 2-3 列出保留区的相关信息。

表 2-3　保留区各字节信息

名　　称	Offset	大小	描　　述
BS_jmpBoot	0	3	跳转指令。指向启动代码，允许以下两种形式： jmpBoot[0] = 0xEB，jmpBoot[1] = 0x??, jmpBoot[2] = 0x90 和 jmpBoot[0] = 0xE9, jmpBoot[1] = 0x??, jmpBoot[2] = 0x??，0x??表示该字节可以为任意 8-bit 值，这是 Intel x86 架构 3 字节的无条件转移指令，跳转到操作系统的启动代码，这些启动代码往往紧接 BPB 后面 0 扇区里的剩余字节，当然也可能位于其他扇区。以上的两种形式任取。jmpBoot[0] =0xEB 是较常用的一种格式
BS_OEMName	3	8	建议值为“MSWIN4.1”，此域经常引起人们的误解，其实这只是一个字符串而已，Microsoft 的操作系统似乎并不关心此域。但其他厂商的 FAT 驱动程序可能会检测此项，这就是为什么建议将此域设为 “MSWIN4.1”的原因，这样可以尽量避免兼容性的问题。可以更改它的内容，但这有可能造成某些 FAT 驱动程序无法识别该磁盘。很多情况下该域用于显示格式化该 FAT 卷的操作系统的名称
BPB_BytsPerSec	11	2	每扇区字节数，取值只能是以下的几种情况：512、1024、2048 或 4096，设置为 512 将取得最好的兼容性，目前有很多的 FAT 代码都是硬性地规定每扇区字节数为 512，而不是实际检测该域的值，Microsoft 的操作系统能够很好地支持 1024、2048 和 4096 各种数值

续表

名　称	Offset	大小	描　述
BPB_SecPerClus	13	1	每簇扇区数，其值必须是 2 的整数次方（该整数必须>=0），如 1、2、4、8、16、32、64 或 128，同时还必须保证每簇的字节数不超过 32KB，即保证（BPB_BytsPerSec * BPB_SecPerClus <= 32K(1024*32) 该值大于 32KB 是绝对不允许的，虽然有些版本的操作系统支持每簇字节数最大到 64KB，但很多应用程序的安装程序都无法在这样的 FAT 文件系统上正常运行
BPB_RsvdSecCnt	14	2	保留区中保留扇区的数目，保留扇区从 FAT 卷的第一个扇区开始，此域不能为 0，对于 FAT12 和 FAT16 必须为 1，FAT32 的典型取值为 32，目前很多 FAT 程序都是硬性规定 FAT12/FAT16 的保留扇区数为 1，而不对此域进行实际的检测，Microsoft 的操作系统支持任何非零的值
BPB_NumFATs	16	1	此卷中 FAT 表的份数。任何 FAT 格式此域都建议为 2。虽然此域取值为其他>=1 的数值也是合法的，但是很多 FAT 程序和部分操作系统对于此项不为 2 的时候将无法正常工作。当不为 2 时，Microsoft 的操作系统仍能良好地工作
BPB_RootEntCnt	17	2	对于 FAT12 和 FAT16 此域包含根目录中的目录项数（每个项长度为 32 字节），对于 FAT32，此项必须为 0。对于 FAT12 和 FAT16，此数值乘以 32 必须为 BPB_BytsPerSec 的偶数倍，为了达到最好的兼容性，FAT12/FAT16 应该取值为 512
BPB_TotSec16	19	2	早期版本中 16-bit 的总扇区数，这里的总扇区数包括 FAT 卷上 4 个基本区的全部扇区，此域可以为 0，若此域为 0，那么 BPB_TotSec32 必须非 0，对于 FAT32，此域必须为 0。对于 FAT12/FAT16，此域填写总扇区数，如果该数值小于 0x10000 的话，BPB_TotSec32 必须为 0
BPB_Media	21	1	对于“固定”（不可移动）存储介质而言，0xF8 是标准值，对于可移动存储介质，经常使用的数值是 0xF0，此域合法的取值可以取 0xF0，0xF8，0xF9，0xFA，0xFB，0xFC，0xFD，0xFE 和 0xFF。另外要提醒的一点是，无论此域写入什么数值，同时也必须在 FAT[0]的低字节写入相同的值，这是因为早期的 MSDOS 1.x 使用该字节来判定是何种存储介质
BPB_FATSz16	22	2	FAT12/FAT16 一个 FAT 表所占的扇区数，对于 FAT32 此域必须为 0，在 BPB_FATSz32 中有指定其 FAT 表的大小
BPB_SecPerTrk	24	2	每磁道扇区数，用于 BIOS 中断 0x13，此域只对于有“特殊形状”（由磁头和柱面分割为若干磁道）的存储介质有效，同时必须可以调用 BIOS 的 0x13 中断得到此数值
BPB_NumHeads	26	2	磁头数，用于 BIOS 的 0x13 中断，类似于上面的 BPB_SecPerTrk，只有对特殊的介质才有效，此域包含一个至少为 1 的数值，比如 1.4MB 的软盘此域为 2
BPB_FATSz32	36	4	一个 FAT 表所占的扇区数，此域为 FAT32 特有，同时 BPB_FATSz16 必须为 0

2．FAT 数据结构（FAT Data Structure）

接下来一个重要的数据结构就是 FAT 表（File Allocation Table），它是一 一对应于数据区簇号的列表。

由于文件系统分配磁盘空间按簇来分配的。因此，文件占用磁盘空间时，基本单位不是字节而是簇，即使某个文件只有一个字节，操作系统也会给它分配一个最小单元——一个簇。为了可以将磁盘空间有序地分配给相应的文件，而读取文件的时候又可以从相应的地址读出文件，将数据区空间分成 BPB_BytsPerSec*BPB_SecPerClus 字节长的簇来管理，

FAT 表项的大小与 FAT 类型有关，FAT12 的表项为 12-bit，FAT16 为 16-bit，而 FAT32 则为 32-bit。对于大文件，需要分配多个簇。同一个文件的数据并不一定完整地存放在磁盘中一个连续的区域内，而往往会分成若干段，像链表一样存放。这种存储方式称为文件的链式存储。为了实现文件的链式存储，文件系统必须准确地记录哪些簇已经被文件占用，还必须为每个已经占用的簇指明存储后继内容的下一个簇的簇号，对文件的最后一簇，则要指明本簇无后继簇。

以上这些都是由 FAT 表来保存的，FAT 表的对应表项中记录着它所代表的簇的有关信息：诸如是否空，是否是坏簇，是否已经是某个文件的尾簇等。

FAT 的项数与硬盘上的总簇数相关（因为每一个项要代表一个簇，簇越多当然需要的 FAT 表项越多），每一项占用的字节数也与总簇数有关（因为其中需要存放簇号，簇号越大当然每项占用的字节数就大）。这里介绍一下 FAT 目录，其实它和普通的文件并没有什么不一样的地方，只是多了一个表示它是目录的属性（attrib），另外就是目录所链接的内容是一个 32 字节的目录项（32-byte FAT directory entries 后面有具体讨论）。除此之外，目录和文件没什么区别。FAT 表是根据簇数和文件对应的。第一个存放数据的簇是簇 2。

簇 2 的第一个扇区（磁盘的数据区）根据 BPB 来计算，首先计算根目录所占的扇区数：

RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec – 1)) / BPB_BytsPerSec;

因为 FAT32 的 BPB_RootEntCnt 为 0，所以对于 FAT32 卷 RootDirSectors 的值也一定是 0。上式中的 32 是每个目录项所占的字节数。计算结果四舍五入。

数据区的起始地址，簇 2 的第一个扇区由下面公式计算：

```
If(BPB_FATSz16 != 0)
    FATSz = BPB_FATSz16;
Else
    FATSz = BPB_FATSz32;
    FirstDataSector = BPB_RsvdSecCnt + (BPB_NumFATs * FATSz) + RootDirSectors;
```

注意：扇区号指的是针对卷中包含 BPB 的第一个扇区的偏移量（包含 BPB 的第一个扇区是扇区 0），并不是必须直接和磁盘的扇区相对应。因为卷的扇区 0 并不一定就是磁盘的扇区 0。给一个合法的簇号 N，该簇的第一个扇区号（针对 FAT 卷扇区 0 的偏移量）由下式计算：

```
FirstSectorofCluster = ((N – 2) * BPB_SecPerClust) + FirstDataSector;
```

注意：因为 BPB_SecPerClus 总是 2 的整数次方（1,2,4,8,…）这意味着 BPB_SecPerClus 的乘除法运算可以通过移位（SHIFT）来进行。在当前 Intel x86 架构二进制的机器上乘法（MULT）和除法（DIV）的机器指令非常繁杂和庞大，而使用移位来运算则会相对快许多。

3．FAT 目录结构（FAT Directory Structure）

FAT 目录其实就是一个由 32 字节的线性表构成的“文件”。根目录（root directory）是一个特殊的目录，它存在于每一个 FAT 卷中。对于 FAT12/16，根目录存储在磁盘中固定的地方，它紧跟在最后一个 FAT 表后。根目录的扇区数也是固定的，可以根据 BPB_RootEntCnt 计算得出（参见前文计算公式），对于 FAT12/16，根目录的扇区号是相对该 FAT 卷第一个扇区（0 扇区）的偏移量。

FirstRootDirSecNum = BPB_RsvdSecCnt + (BPB_NumFATs * BPB_FATSz16);

FAT32 的根目录由簇链组成，其扇区数不确定，这点与普通的文件相同，根目录的第一个扇区号存储在 BPB_RootClus 中，根目录不同于其他的目录，没有日期和时间戳，也没有目录名（“/”并不是其目录名），同时根目录里没有“.”和“..”这两个目录项，根目录另一个特殊的地方在于，根目录中有一个设置了 ATTR_VOLUME_ID 位（如表 2-4 所示）的文件，这个文件在整个 FAT 卷中是唯一的。表 2-4 列出了 32 字节目录项结构。

表 2-4　FAT 的 32 字节目录项结构

名　称	Offset	大小	描　述
DIR_Name	0	11	短文件名
DIR_Attr	11	1	文件属性： ATTR_READ_ONLY 0x01 ATTR_HIDDEN 0x02 ATTR_SYSTEM 0x04 ATTR_VOLUME_ID 0x08 ATTR_DIRECTORY 0x10 ATTR_ARCHIVE 0x20 ATTR_LONG_NAME ATTR_READ_ONLY | ATTR_HIDDEN | ATTR_SYSTEM | ATTR_VOLUME_ID 前两个属性位为保留位，在文件创建时应把这两位设为 0，在以后的使用中不能读写和更改
DIR_NTRes	12	1	保留给 Window NT 使用，在文件创建时设置该位为 0，在以后的使用中不能读写和更改
DIR_CrtTimeTeenth	13	1	文件创建时间的毫秒级时间戳，由于 DIR_CrtTime 的精度为 2 秒，所以此域的有效值在 0~199 之间
DIR_CrtTime	14	2	文件创建时间
DIR_CrtDate	16	2	文件创建日期
DIR_LastAccDate	18	2	最后访问日期，请注意并没有最后访问时间域，而只有日期，这日期是指文件被读或写的日期，如果是写，该日期还应该被写到 DIR_WrtDate 中
DIR_FstClusHI	20	2	该目录项簇号的高位字（FAT12/16 此位为 0）
DIR_WrtTime	22	2	最后写的时间，文件创建被认作写
DIR_WrtDate	24	2	最后写的日期，文件创建被认作写
DIR_FstClusLO	26	2	该目录项簇号的低位字
DIR_FileSize	28	2	文件大小，由 32 字节双字组成

2.2.2　RAMFS 内核文件系统

RAMFS 是一个利用 VFS 自身结构的内存文件系统，RAMFS 没有自己的文件存储结构，它的文件存储于 page cache 中，目录结构由 dentry 链表本身描述，文件由 VFS 的 inode 结构描述。在 Linux2.4.20 内核中，fs/ramfs/inode.c 定义这一文件系统和文件系统的基本操作。

（1）page 操作：对页的操作。

```
static struct address_space_operations ramfs_aops = {
       readpage:      ramfs_readpage,
       writepage:     fail_writepage,
       prepare_write:  ramfs_prepare_write,
       commit_write:   ramfs_commit_write
};
```

（2）文件操作。

```
static struct file_operations ramfs_file_operations = {
       read:          generic_file_read,
       write:         generic_file_write,
       mmap:          generic_file_mmap,
       fsync:         ramfs_sync_file,
};
```

（3）inode 操作。

```
static struct inode_operations ramfs_dir_inode_operations = {
       create:        ramfs_create,
       lookup:        ramfs_lookup,
       link:          ramfs_link,
       unlink:        ramfs_unlink,
       symlink:       ramfs_symlink,
       mkdir:         ramfs_mkdir,
       rmdir:         ramfs_rmdir,
       mknod:         ramfs_mknod,
       rename:        ramfs_rename,
};
```

（4）超级操作。

```
static struct super_operations ramfs_ops = {
       statfs:        ramfs_statfs,
       put_inode:     force_delete,
};
```

ramfs 文件系统模板在 Linux 下以模块方式实现，以下是实现代码。

```
static int __init init_ramfs_fs(void)
{
       return register_filesystem(&ramfs_fs_type);
}

static void __exit exit_ramfs_fs(void)
{
       unregister_filesystem(&ramfs_fs_type);
}

module_init(init_ramfs_fs)
module_exit(exit_ramfs_fs)

int __init init_rootfs(void)
{
       return register_filesystem(&rootfs_fs_type);
}
```

2.2.3 JFFS 与 YAFFS 文件系统

JFFS 文件系统是瑞典 Axis Communications AB 为嵌入式系统开发的日志文件系统。JFFS1 应用在 Linux 2.2 以上版本中，JFFS2 在 Linux 2.4 内核和 Ecos 中。在 Linux 的实现中，JFFS 必须建立在 MTD 驱动程序的上层。

1. JFFS 文件系统

由于 JFFS 是针对以闪存为存储介质的嵌入式系统，所以充分考虑了闪存的物理局限性，使用了尽可能高效的日志系统，使文件系统更加可靠。与前面介绍的 TrueFFS 及其他中间层驱动相比，JFFS 是专门针对闪存的文件系统，这个文件系统除了有日志功能外，还包含了前面在 TrueFFS 章节中介绍的负载平衡、垃圾收集等功能。另外一个重要特点是这个文件系统是源代码公开的，方便了学习和使用。

日志文件系统的主要设计思想是跟踪文件系统的变化而不是文件系统的内容。在日志文件系统中，存储系统上面有一系列节点记录了对文件的操作。日志节点上面记录的信息包括以下内容。

- 与日志节点关联的文件的标示符。
- 日志节点的序列号（version）。
- 当前节点的 uid、gid 等信息。
- 其他关于文件内容分布的信息。

JFFS 是一种纯日志文件系统。Linux 中所谓文件系统是一系列存放在存储介质上的节点。在 JFFS1 中，只有一种日志节点 struct jffs_raw_inode 用于在闪存芯片中存放数据，节点的定义如下：

```
struct jffs_raw_inode
{
__u32 magic; /* 魔数 */
__u32 ino; /* I 节点编号 */
__u32 pino; /* 该节点的父节点编号 */
__u32 version; /* Version 号 */
__u32 mode; /* 文件类型 */
__u16 uid; /* 属组 */
__u16 gid; /* 文件属组 */
__u32 atime; /* 最后访问时间 */
__u32 mtime; /*最后修改时间 */
__u32 ctime; /* 创建时间 */
__u32 offset; /* 数据偏移 */
__u32 dsize; /* 数据长度 */
__u32 rsize; /* 要删除的数据长度 */
__u8 nsize; /* 名称长度 */
__u8 nlink; /* 文件链接数 */
__u8 spare : 6; /* 保留位 */
__u8 rename : 1; /* 是否需要更名? */
__u8 deleted : 1; /* 是否被删除 */
__u8 accurate; /* 是否是可用的数据 */
__u32 dchksum; /* 数据校验码 */
__u16 nchksum; /* 名称校验码 */
__u16 chksum; /* 节点信息校验码 */
};
```

2．JFFS2 文件系统

JFFS2 是 Redhat 公司基于 JFFS 开发的闪存文件系统，它主要是针对 Redhat 公司的嵌入式产品 eCos 开发的嵌入式文件系统，当然 JFFS2 也可以使用在 Linux、ucLinux 中。JFFS2 克服了 JFFS 中的如下缺点：

- 使用了基于哈希表的日志节点结构，大大加快了对节点的操作速度。
- 支持数据压缩。
- 提供了“写平衡”支持。
- 支持多种节点类型（数据 I 节点，目录 I 节点等）；而 JFFS 只支持一种节点。
- 提高了对闪存的利用率，降低了内存的消耗。

JFFS2 中定义了多种节点，但是每种节点都包含下面的信息。

```
struct jffs2_unknown_node
{
__u16 magic; /*作为 nodetype 的补充*/
__u16 nodetype; /*节点类型*/
__u32 totlen; /* 节点总长度 */
__u32 hdr_crc;/*CRC 校验码*/
};
```

3．YAFFS 文件系统

YAFFS（Yet Another Flash File System）类似于 JFFS/JFFS2，是专门为 NAND 闪存设计的嵌入式文件系统，适用于大容量的存储设备。它是日志结构的文件系统，提供了损耗平衡和掉电保护，可以有效地避免意外掉电对文件系统一致性和完整性的影响。YAFFS 文件系统是按层次结构设计的，分为文件系统管理层接口、YAFFS 内部实现层和 NAND 接口层，这样就简化了其与系统的接口设计，可以方便地集成到系统中去。与 JFFS 相比，它减少了一些功能，因此速度更快，占用内存更少。

YAFFS 充分考虑了 NAND 闪存的特点，根据 NAND 闪存以页面为单位存取的特点，将文件组织成固定大小的数据段。利用 NAND 闪存提供的每个页面 16 字节的备用空间来存放 ECC(Error Correction Code)和文件系统的组织信息，不仅能够实现错误检测和坏块处理，也能够提高文件系统的加载速度。YAFFS 采用一种多策略混合的垃圾回收算法，结合了贪心策略的高效性和随机选择的平均性，达到了兼顾损耗平均和系统开销的目的。

（1）YAFFS 文件组织结构：YAFFS 将文件组织成固定大小（512 字节）的数据段。每个文件都有一个页面专门存放文件头，文件头保存了文件的模式、所有者 id、组 id、长度、文件名等信息。为了提高文件数据块的查找速度，文件的数据段被组织成树形结构。YAFFS 在文件进行改写时总是先写入新的数据块，然后将旧的数据块从文件中删除。YAFFS 使用存放在页面备用空间中的 ECC 进行错误检测，出现错误后会进行一定次数的重试，多次重试失败后，该页面就被停止使用。

（2）YAFFS 物理数据组织：YAFFS 充分利用了 NAND 闪存提供的每个页面 16 字节的备用空间，参考了 SmartMedia 的设定，备用空间中 6 个字节被用做页面数据的 ECC，两个字节分别用做块状态字和数据状态字，其余的 8 字节（64 位）用来存放文件系统的组织信息，即元数据。由于文件系统的基本组织信息保存在页面的备份空间中，因此，在文件系统加载时只需要扫描各个页面的备份空间，即可建立起整个文件系统的结构，而不需

要像 JFFS 那样扫描整个介质，从而大大加快了文件系统的加载速度。

（3）YAFFS 擦除块和页面分配：YAFFS 中用数据结构来描述每个擦除块的状态。该数据结构记录了块状态，并用一个 32 位的位图表示块内各个页面的使用情况。在 YAFFS 中，有且仅有一个块处于“当前分配”状态。新页面从当前进行分配的块中顺序进行分配，若当前块已满，则顺序寻找下一个空闲块。

（4）YAFFS 垃圾收集机制：YAFFS 使用一种多策略混合的算法来进行垃圾回收，将贪心策略和随机选择策略按一定比例混合使用，当满足特定的小概率条件时，垃圾回收器会试图随机选择一个可回收的页面。通过使用多策略混合的方法，YAFFS 能够有效地改善贪心策略造成的不平均；通过不同的混合比例，则可以控制损耗平均和系统开销之间的平衡。考虑到 NAND 的擦除很快（与 NOR 相比可忽略不计），YAFFS 将垃圾收集的检查放在写入新页面时进行，而不是采用 JFFS 那样的后台线程方式，从而简化了设计。

在 Linux 2.6 内核中加入了这一文件系统，在 fs/yaffs/yaffs.c 中详细定义此文件系统相关数据结构和操作。

文件操作。

```
static struct file_operations yaffs_file_operations = {
#ifdef CONFIG_YAFFS_USE_GENERIC_RW
    read:       generic_file_read,
    write:      generic_file_write,
#else
    read:       yaffs_file_read,
    write:      yaffs_file_write,
#endif
    mmap:       generic_file_mmap,
    flush:      yaffs_file_flush,
    fsync:      yaffs_sync_object,
};
```

inode 操作。

```
static struct inode_operations yaffs_file_inode_operations = {
    setattr:    yaffs_setattr,
};
```

dir_inode 操作。

```
static struct inode_operations yaffs_dir_inode_operations = {
    create:     yaffs_create,
    lookup:     yaffs_lookup,
    link:       yaffs_link,
    unlink:     yaffs_unlink,
    symlink:    yaffs_symlink,
    mkdir:      yaffs_mkdir,
    rmdir:      yaffs_unlink,
    mknod:      yaffs_mknod,
    rename:     yaffs_rename,
    setattr:    yaffs_setattr,
};
```

dir 操作。

```
static struct file_operations yaffs_dir_operations = {
    read:       generic_read_dir,
```

```
    readdir:    yaffs_readdir,
    fsync:      yaffs_sync_object,
};
```

yaff_super 操作。

```
static struct super_operations yaffs_super_ops = {
    statfs:         yaffs_statfs,
    read_inode:     yaffs_read_inode,
    put_inode:      yaffs_put_inode,
    put_super:      yaffs_put_super,
//  remount_fs:
    delete_inode:   yaffs_delete_inode,
    clear_inode:    yaffs_clear_inode,
};
```

2.2.4　EXT2/EXT3 文件系统

EXT2（The Second Extended File System）是由 Remy Card 发明的，它是 Linux 的一个可扩展的、强大的文件系统。至少在 Linux 社区中，EXT2 是最成功的文件系统，是所有当前的 Linux 发布版的基础。

与大多数文件系统一样，EXT2 文件系统建立在这样的前提下：文件的数据存放在数据块中，这些数据块的长度都相同。虽然不同的 EXT2 文件系统的块长度可以不同，但是对于一个特定的 EXT2 文件系统，在它创建时，其块长度就确定了（使用 mke2fs）。每一个文件的长度都按块取整。如果块大小是 1024 字节，一个 1025 字节的文件会占用两个 1024 字节的块。这意味着平均每一个文件要浪费半个块。在通常的计算中，用户会用内存和磁盘的使用来交换对 CPU 的使用（空间交换时间），在这种情况下，Linux 如大多数操作系统一样，会为了较少的 CPU 负载，而使用相对低效的磁盘利用率。不是文件系统中所有的块都用来存储数据，因为必须用一些块放置描述文件系统结构的信息。

EXT2 用一个 inode 数据结构描述系统中的每一个文件，从而定义了文件系统的拓扑结构。一个 inode 描述了一个文件中的数据占用了哪些块及文件的访问权限、文件的修改时间和文件的类型等。EXT2 文件系统中的每一个文件都用一个 inode 描述，而每一个 inode 都用一个独一无二的数字标识。文件系统的所有 inode 都放在 inode 表中。EXT2 的目录是简单的特殊文件（它们也使用 inode 描述），该文件的内容是一组指针，每一个指针指向一个 inode，该 inode 描述了目录中的一个文件。

如图 2-3 所示是一个 EXT2 文件系统的布局，该文件系统占用了一个块结构设备上的一系列的块。从文件系统所关心的角度来看，块设备都可以被当做一系列能够读写的块。文件系统自己不需要关心一个块应该放在物理介质的哪个位置，因为这是设备驱动程序的工作。当一个文件系统需要从包括它的块设备上读取信息或数据的时候，它只是请求支撑它的设备驱动程序来读取整数数目的块。

EXT2 文件系统把它占用的逻辑分区划分成块组（Block Group）。每一个组除了当做信息和数据的块来存放真实的文件和目录之外，还复制对于文件系统一致性至关重要的信息。当发生灾难、文件系统需要恢复的时候，这些复制的信息是必要的。下面对于每一个块组的内容进行了详细的描述。

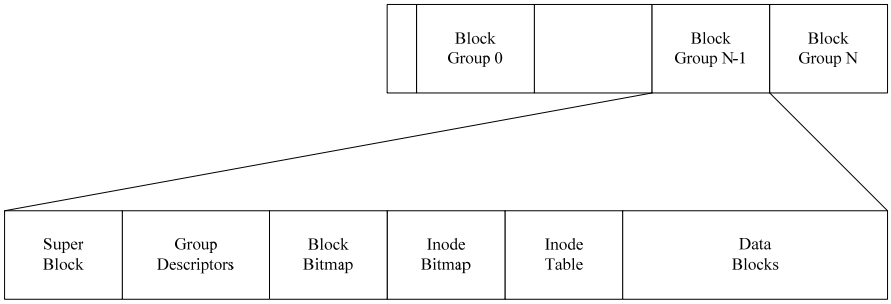


图 2-3　EXT2 文件系统的布局

1．The EXT2 Inode

在 EXT2 文件系统中，I 节点是建设的基石，文件系统中的每一个文件和目录都用一个且只用一个 inode 描述。每一个块组的 EXT2inode 都集中放在 inode 表中，另有一个位图，让系统跟踪 inode 表中分配和未分配的 I 节点信息。如图 2-4 所示为一个 EXT2 inode 的格式，它包括下面一些域。

（1）Mode：包括两组信息，这个 inode 描述的是什么和使用它的用户应具有的权限。对于 EXT2，一个 inode 可以描述一个文件、目录、符号链接、块设备、字符设备或 FIFO。

（2）Owner Information：这个文件或目录的属主的用户和组标识符。这允许文件系统正确地进行文件访问权限控制（分类）。

（3）Size：文件的大小（字节）。

（4）Timestamps：这个 inode 创建的时间和它上次被修改的时间。

（5）Datablocks：指向这个 inode 描述的数据所占用的一组块的指针。前面的 12 个是指向这个 inode 描述的数据的物理块的指针，最后的 3 个指针包括更多级的间接的数据块。例如，两级的间接块指针指向一个数据块，该数据块中包含指向物理块的指针。这意味着访问小于或等于 12 个数据块的文件要比访问大于 12 个数据块的文件快。

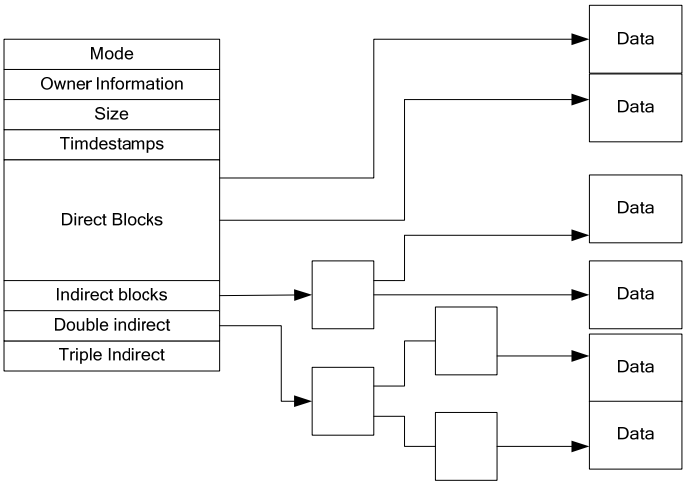


图 2-4　Inode 节点

2．EXT2 目录

EXT2 文件系统中，目录是特殊文件，用来创建和存放到文件系统中的文件的访问路径。如图 2-5 所示为内存中一个目录条目的布局。一个目录文件是一个目录条目的列表，每一个目录条目包括以下信息。

（1）inode：目录的 inode。这是一个索引，是该 inode 在块组的 inode 表中的索引。在图 2-5 中，文件名为“file”的文件的目录条目所引用的 inode 为“i1”。

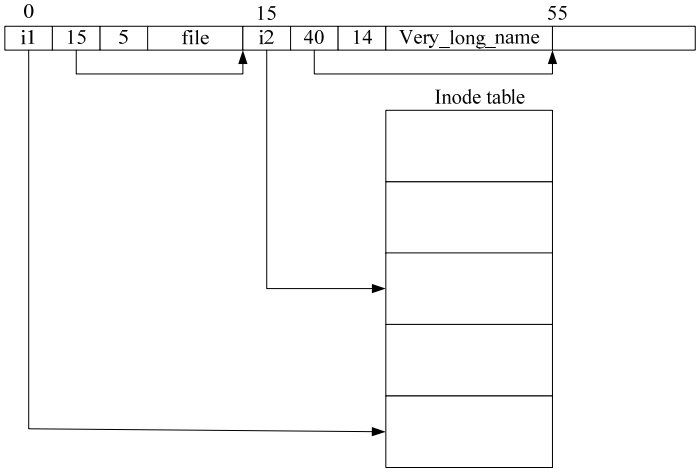


图 2-5　EXT2 目录

（2）Name length：这个目录条目的长度（以字节计）。

（3）Name：这个目录条目的名字。每一个目录中的前两个条目总是标准的“.”和“..”，分别表示“本目录”和“父目录”。

3．在 EXT2 文件系统中查找文件

Linux 的文件名和 UNIX 文件名的格式一样。一个文件名的例子是/home/rusling/.cshrc，其中/home 和/rusling 是路径，文件名是.cshrc。像其他 UNIX 系统一样，Linux 并不关心文件名本身的格式，它是由可打印字符组成的任意长度的字符串。为了在 EXT2 文件系统中找到代表这个文件的 inode，系统必须一个目录一个目录地解析文件名，直到得到这个文件本身。

需要的第一个 inode 是这个文件系统的根的 inode。可以通过文件系统的超级块找到它的编号。为了读取一个 EXT2 inode，必须在适当的块组中的 inode 表中查找。例如，如果根的 inode 编号是 42，那么需要块组 0 中的 inode 表中的第 42 个 inode。Root inode 是一个 EXT2 目录，换句话说 root inode 的模式（mode 域）描述它是一个目录，它的数据块包括 EXT2 目录条目。

Home 是这些目录之一，这个目录条描述/home 目录的 inode 编号。必须读取这个目录（首先读取它的 inode，然后读取从这个 inode 描述的数据块读取目录条目），从中查找 rusling 条目，从而得到描述/home/rusling 目录的 inode 编号。最后，读取描述/home/rusling 目录的 inode 指向的目录条目，找到.cshrc 文件的 inode 编号，这样就得到了包含文件信息的数据块。

2.2.5 /proc 文件系统

/proc 文件系统是一个伪文件系统，它只存在内存中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过/proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息如进程是动态改变的，所以用户或应用程序读取/proc 文件时，/proc 文件系统是动态从系统内核读出所需信息并提交的。它的目录结构如表 2-5 所示。

表 2-5 /proc 文件系统的目录结构

目录名称	目录内容
apm	高级电源管理信息
cmdline	内核命令行
Cpuinfo	关于 CPU 信息
Devices	可以用到的设备（块设备/字符设备）
Dma	使用的 DMA 通道
Filesystems	支持的文件系统
Interrupts	中断的使用
Ioports I/O	端口的使用
Kcore	内核核心印象
Kmsg	内核消息
Ksyms	内核符号表
Loadavg	负载均衡
Locks	内核锁
Meminfo	内存信息
Misc	杂项
Modules	加载模块列表
Mounts	加载的文件系统
Partitions	系统识别的分区表
Rtc	实时时钟
Slabinfo Slab	池信息
Stat	全面统计状态表
Swaps	对换空间的利用情况
Version	内核版本
Uptime	系统正常运行时间

并不是所有这些目录在系统中都有，这取决于内核配置和装载的模块。另外，在/proc 下还有 3 个很重要的目录：net，scsi 和 sys。sys 目录是可写的，可以通过它来访问或修改内核的参数（见后续介绍），而 net 和 scsi 则依赖于内核配置。例如，如果系统不支持 scsi，则 scsi 目录不存在。

除了以上介绍的这些，还有的是一些以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的 PID 号为目录名，它们是读取进程信息的接口。而 self 目录则是读取进程本身的信息接口，是一个 link。proc 文件系统的名字就是由之而起。进程目录的结构如表 2-6 所示。

表 2-6　进程目录的结构

目录名称	目录内容
Cmdline	命令行参数
Environ	环境变量值
Fd	一个包含所有文件描述符的目录
Mem	进程的内存被利用情况
Stat	进程状态
Status	进程当前状态，以可读的方式显示出来
Cwd	当前工作目录的链接
Exe	指向该进程的执行命令文件
Maps	内存映像
Statm	进程内存状态信息
Root	链接此进程的 root 目录

用户如果要查看系统信息，可以用 cat 命令。例如：

```
[root@yangzongde fs]# cat /proc/filesystems
nodev   rootfs
nodev   bdev
nodev   proc
nodev   sockfs
nodev   tmpfs
nodev   shm
nodev   pipefs
        ext2
nodev   ramfs
        iso9660
nodev   devpts
        ext3
nodev   usbdevfs
nodev   usbfs
        ntfs
nodev   autofs
[root@yangzongde fs]#
```

用户还可以修改内核参数。在/proc 文件系统中有一个目录/proc/sys。它不仅提供了内核信息，而且可以通过它修改内核参数来优化系统。但是必须很小心，因为如果修改不当可能会造成系统崩溃。

要改变内核的参数，只要用 vi 编辑或 echo 参数重定向到文件中即可。下面有一个例子。

```
# cat /proc/sys/fs/file-max
4096
# echo 8192 > /proc/sys/fs/file-max
# cat /proc/sys/fs/file-max
8192
```

2.2.6　Linux 文件操作函数

1．open/fopen 打开文件

（1）open 函数

函数定义：int open(const char *pathname,int flags)

功能说明：参数 pathname 为欲打开的文件路径字符串，flags 所使用的旗标如表 2-7 所示。

表 2-7　open 函数的 flags

flags	说　明
O_RDONLY	只读方式打开文件
O_WRONLY	只写方式打开文件
O_RDWR	可读可写方式打开文件
O_CREAT	若欲打开的文件不存在则自动建立该文件
O_EXCL	如果 O_CREAT 也设置，此指令会检查文件是否存在。文件若不存在则建立此文件，否则将导致打开文件错误，此外，若 O_CREAT 与 O_EXCL 同时设置，且以打开的文件为符号，则打开文件失败
O_NOCTTY	如果欲打开的文件为终端设备时，则不会将该终端机当成进程控制终端机
O_TRUNC	若文件存在并且以可写的方式打开时，此标识令文件长度为 0，而原来存在于该文件的资料也会消失
O_APPEND	当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入到文件后面
O_NONBLOCK	以不可阻断方式打开，也就是无论有无数据读取或等待，都会立即返回进程之中
O_NDELAY	同 O_NONBLOCK
O_SYNC	以同步方式打开文件
O_NOFOLLOW	如果参数 pathname 所指的文件为一符号连接，则会令打开文件失败
O_DIRECTORY	如果参数 pathname 所指的文件并非为一目录，则会令打开文件失败
S_IRWXU00700	权限，代表读文件所有者具有可读可写可执行权限
S_IRUSR	同 S_IREAD，00400 权限，代表该文件所有者具有可读取的权限
S_IWUSR	同 S_IWRITE，00200 权限，代表该文件所有者具有可写入的权限
S_IXUSR	同 S_IEXEC，00100 权限，代表该文件所有者具有可执行的权限
S_IRWXG	00070 权限，代表该文件用户组具有可读可写可执行权限
S_IRGRP	00040 权限，代表该文件用户组具有可读权限
S_IWGRP	00020 权限，代表该文件用户组具有可写权限
S_IXGRP	00010 权限，代表该文件用户组具有可执行权限
S_IRWXO	00007 权限，代表其他用户具有可读可写可执行权限
S_IROTH	00004 权限，代表其他用户具有可读权限
S_IWOTH	00002 权限，代表其他用户具有可写权限
S_IXOTH	00001 权限，代表其他用户具有可执行权限

库头文件：include<sys/tpyes.h>，include<sys/stat.h>，include<fcntl.h>

返回值：若所有欲核查的权限都通过则返回 0，表示成功。只要有一个权限被禁止，则返回–1。

（2）fopen 函数

函数定义：FILE *fopen(const char *path,const char *mode)

功能说明：参数 path 为欲打开文件的路径及文件名，参数 mode 字符串为打开形态。分别为：

r：打开只读文件，该文件必须存在。

r+：打开可读写的文件，此文件必须存在。

w：打开只写文件，该文件存在则长度清零，即内容消息，若不存在就创建。

w+：打开可读写文件，该文件存在则长度清零，即内容消息，若不存在就创建。

a：以附加方式打开只写文件，若文件不存在，就建立文件，如果文件存在，写入的数据会被加到文件后面。

a+：以附加方式打开可读写文件，若文件不存在，就建立文件，如果文件存在，写入的数据会被加到文件后面。

库头文件：include<stdio.h>

返回值：文件打开后，指向该流的文件指针就会被返回，若文件打开失败则返回 NULL，错误代码写入 errno 中。

2．建立文件

creat 函数

函数定义：int creat(const char *pathname,mode_tmode)

功能说明：参数 pathname 指向欲建立的文件路径，其相当于使用以下方式调用 open 函数。

```
open(const char *pathnaen,(O_CREAT|O_WRONLY|O_TRUNC),mode_t mode)
```

头文件：include<sys/tpyes.h>，include<sys/stat.h>，include<fcntl.h>

返回值：create()函数返回新的文件描述词，若有错误返回–1，并把错误代码设置为 errno。相关的返回代码描述如下：

EEXIST：参数 pathname 所指的文件已经存在；

EACCESS：参数 pathname 所指定的文件不符合所要求测试的权限；

EROFS：欲打开写入权限的文件存在于只读文件系统中；

EFAULT：参数 pathname 指针超出可存取内存空间；

EINVAL：参数 mode 不正确；

ENAMETOOLONG：参数 pathname 太长；

ENOTDIR：参数 pathname 为一目录；

ENOMEM：核心内存不足；

ELOOP：参数 pathname 有过多符号连接问题；

EMFILE：已达到进程可同时打开的文件数上限；

ENFILE：已达到系统可以同时打开的文件数上限。

3．关闭文件

（1）close()函数

函数定义：int close(int fd)

功能说明：当使用文件后需要使用此函数来关闭打开的文件，从而让数据写回磁盘，释放系统资源，参数 fd 为打开的文件描述词。

头文件：#include<unistd.h>

返回值：正常执行返回 0，否则返回–1。

（2）fclose()函数

函数定义：int fclose(FILE *stream)

功能说明：用来关闭由 fopen()打开的文件，让缓冲区数据回写入磁盘中，并释放系统资源。

头文件：#include<stdio.h>

返回值：如果成功返回 0，有错误发生则返回 EOF 并把错误代码存放到 errno 中。

4．读取数据

（1）read 函数

函数定义：ssize_t read(int fd,void *buf,size_t count)

功能说明：read()把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中，若参数 count 为 0，则 read()不会有作用并返回 0。返回值为实际读取到的字数。如果返回 0，表示已经达到文件尾部或者无数据可读，另外文件读写位置会随读取到的字节移动。

头文件：#include<unistd.h>

（2）fread 函数

函数定义：size_t fread(void *pth,size_t size,size_t nmemb,FILE *stream)

功能说明：从文件流读取数据，参数 stream 为已经打开的文件指针，参数 ptr 指向欲存放读取的数据空间，读取的字符以参数 size*nmemb 来决定，fread()会返回实际读取到的 nmemb 数目，如果此值比参数 nmemb 小，则代表可能读到了文件的尾部，这时必须用 feof 或者 ferror 来决定发生什么情况。

头文件：#include<stdio.h>

返回值：返回实际读取到的 nmemb 数目。

5．写入文件

（1）write 函数

函数定义：ssize_t write(int fd,const void *buf,size_t count)

功能说明：将参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。

库头文件：#include<unistd.h>

返回值：如果顺序则返回实际写入的数据字节数，当有错误发生时则返回–1，错误代码存入 errno 中。

（2）fwrite 函数

函数定义：size_t fwrite(const void *ptr,size_t size,size_t nmemb,FILE *stream)

功能说明：将数据写入文件流中，参数 stream 为已经打开的文件指针，参数 ptr 指向欲写入的数据地址，总共写入的实际字符以参数 size *nmemb 来决定。

头文件：#include<stdio.h>

返回值：返回实际写入的 nmemb 数目。

6．移动文件的读写位置

（1）lseek()

函数定义：off_t lseek(int fildes,off_t offset,int whence)

功能说明：当打开文件时通常都有一个读写位置，通常是指向文件头部，若是以附加的方式打开文件，则在文件尾部，lseek 用来控制文件的读写位置，参数 fildes 为已经打开的文件，offset 为根据参数 whence 来移动读写位置的位移数。whence 为下列其中一种。

SEEK_SET：参数 offset 即为新的读写位置；

SEEK_CUR：以目前的读写位置往后增加 offset 个位移量；

SEEK_END：将读写位置指向文件后再增加 offset 个位置量。

头文件：#include<unistd.h>

返回值：当执行成功则返回目前读写位置，也就是距离文件头部的字节数，若错误则返回-1。

（2）fseek()

函数定义：int fseek(FILE *stream,long offset,int whence)

功能说明：移动文件的读写位置，参数 stream 为已经打开的文件指针，参数 offset 为根据参数 whence 来移动读写位置的位移数。

头文件：#include<stdio.h>

返回值：如果成功返回 1，否则返回 0。

关于文件的其他操作，如 dup/dup2 复制文件、fcntl 文件描述词操作、flock 锁定文件或者解锁文件、fsync 将缓冲区数据写回磁盘等本书将不再介绍，请读者参阅相关书籍。

2.3　存储管理

2.3.1　MTD 内存管理

MTD（Memory Technology Device，内存技术设备）是用于访问 memory 设备（ROM、Flash）的 Linux 的子系统。MTD 的主要目的是为了使新的 memory 设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。MTD 的所有源代码在/drivers/mtd 子目录下，如图 2-6 所示。MTD 设备分为四层（从设备节点直到底层硬件驱动），这四层从上到下依次是：设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层。

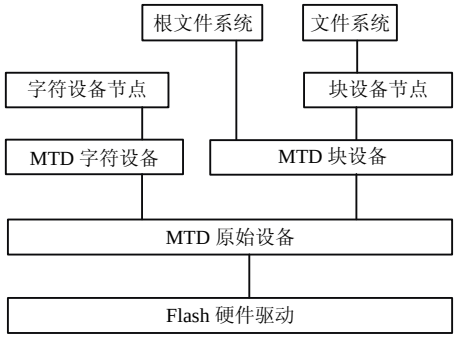


图 2-6　MTD 设备层次

（1）Flash 硬件驱动层。硬件驱动层负责在初始化时驱动 Flash 硬件，Linux MTD 设备的 NOR Flash 芯片驱动遵循 CFI 接口标准，其驱动程序位于 drivers/mtd/chips 子目录下。NAND 型 Flash 的驱动程序则位于/drivers/mtd/nand 子目录下。

（2）MTD 原始设备。原始设备层由两部分组成，一部分是 MTD 原始设备的通用代码，另一部分是各个特定的 Flash 的数据，例如分区。用于描述 MTD 原始设备的数据结构是 mtd_info，这其中定义了大量的关于 MTD 的数据和操作函数。mtd_table(mtdcore.c)则是所有 MTD 原始设备的列表，mtd_part(mtd_part.c)是用于表示 MTD 原始设备分区的结构，其

中包含了 mtd_info，因为每一个分区都是被看成一个 MTD 原始设备加在 mtd_table 中的，mtd_part.mtd_info 中的大部分数据都从该分区的主分区 mtd_part->master 中获得。

在 drivers/mtd/maps/子目录下存放的是特定的 Flash 的数据，每一个文件都描述了一块板子上的 Flash。其中调用 add_mtd_device()、del_mtd_device()建立/删除 mtd_info 结构并将其加入/删除 mtd_table（或者调用 add_mtd_partition()、del_mtd_partition()(mtdpart.c)建立/删除 mtd_part 结构并将 mtd_part.mtd_info 加入/删除 mtd_table 中）。

（3）MTD 设备层：基于 MTD 原始设备，Linux 系统可以定义出 MTD 的块设备（主设备号 31）和字符设备（设备号 90）。MTD 字符设备的定义在 mtdchar.c 中实现，通过注册一系列 file operation 函数（lseek、open、close、read、write）。MTD 块设备则是定义了一个描述 MTD 块设备的结构 mtdblk_dev，并声明了一个名为 mtdblks 的指针数组，这数组中的每一个 mtdblk_dev 和 mtd_table 中的每一个 mtd_info 一一对应。

（4）设备节点：通过 mknod 在/dev 子目录下建立 MTD 字符设备节点（主设备号为 90）和 MTD 块设备节点（主设备号为 31），通过访问此设备节点即可访问 MTD 字符设备和块设备。

（5）根文件系统：在 Bootloader 中将 JFFS（或 JFFS2）的文件系统映像 jffs.image（或 jffs2.img）烧到 Flash 的某一个分区中，在/arch/arm/mach-your/arch.c 文件的 your_fixup 函数中将该分区作为根文件系统挂载。

（6）文件系统：内核启动后，通过 mount 命令可以将 Flash 中的其余分区作为文件系统挂载到 mountpoint 上。

一个 MTD 原始设备可以通过 mtd_part 分割成数个 MTD 原始设备并注册进 mtd_table，mtd_table 中的每个 MTD 原始设备都可以被注册成一个 MTD 设备，其中字符设备的主设备号为 90，次设备号为 0、2、4、6…（奇数次设备号为只读设备），块设备的主设备号为 31，次设备号为 0、1、2、3…。图 2-7 为设备层和原始设备层的函数调用关系。

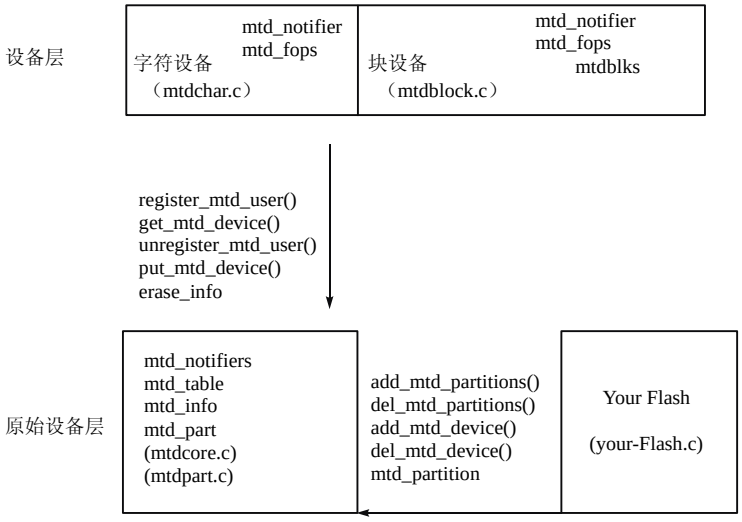


图 2-7　设备层和原始设备层的函数调用关系

1．NOR 型 Flash 芯片驱动与 MTD 原始设备

所有的 NOR 型 Flash 的驱动（探测 probe）程序都放在 drivers/mtd/chips 下，一个 MTD

原始设备可以由一块或者数块相同的 Flash 芯片组成，如图 2-8 所示。假设由 4 块 devicetype 为 x8 的 Flash，每块大小为 8M，interleave 为 2，起始地址为 0x01000000，地址相连，则构成一个 MTD 原始设备（0x01000000-0x03000000），其中两块 interleave 成一个 chip，其地址从 0x01000000 到 0x02000000，另两块 interleave 成一个 chip，其地址从 0x02000000 到 0x03000000。

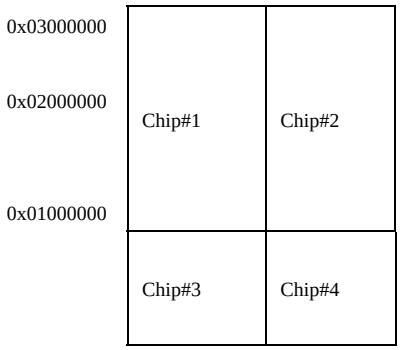


图 2-8　Flash 地址信息

请注意，所有组成一个 MTD 原始设备的 Flash 芯片必须是同类型的（无论是 interleave 还是地址相连），在描述 MTD 原始设备的数据结构中也只是采用了同一个结构来描述组成它的 Flash 芯片。

2．NAND 和 NOR 的比较

NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，提供更高的性能，并且像磁盘一样可以通过接口轻松升级。但是经过了十多年之后，仍然有相当多的硬件工程师分不清 NOR 和 NAND 闪存。“Flash 存储器”经常可以与“NOR 存储器”互换使用。许多业内人士也搞不清楚 NAND 闪存技术相对于 NOR 技术的优越之处，因为大多数情况下闪存只是用来存储少量的代码，这时 NOR 闪存更适合一些。而 NAND 则是高数据存储密度的理想解决方案。NOR 的特点是芯片内执行（eXecute In Place , XIP），这样应用程序可以直接在 Flash 闪存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1~4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 Flash 的管理需要特殊的系统接口。

3．性能比较

Flash 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。由于任何 Flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。

由于擦除 NOR 器件时是以 64~128KB 的块进行的，执行一个写入/擦除操作的时间为 5s，与此相反，擦除 NAND 器件是以 8~32KB 的块进行的，执行相同的操作最多只需要 4ms。

执行擦除时块尺寸的不同进一步拉大了 NOR 和 NADN 之间的性能差距，统计表明，对于给定的一套写入操作（尤其是更新小文件时），更多的擦除操作必须在基于 NOR 的单元中进行。这样，当选择存储解决方案时，设计师必须权衡以下的各项因素。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小，相应的擦除电路更少。

（1）接口差别：NOR Flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。NAND 器件使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。NAND 读和写操作采用 512 字节的块，这一点有点像硬盘管理此类操作，很自然地，基于 NAND 的存储器就可以取代硬盘或其他块设备。

（2）容量和成本：NAND Flash 的单元尺寸几乎是 NOR 器件的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。NOR Flash 占据了容量为 1~16MB 闪存市场的大部分，而 NAND Flash 只是用在 8~128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

（3）可靠性和耐用性：采用 Flash 介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说，Flash 是非常合适的存储方案。可以从寿命（耐用性）、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性。

（4）寿命（耐用性）：在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。NAND 存储器除了具有 10 比 1 的块擦除周期优势，典型的 NAND 块尺寸要比 NOR 器件小 8 倍，每个 NAND 存储器块在给定的时间内的删除次数要少一些。

（5）位交换：所有 Flash 器件都受位交换现象的困扰。在某些情况下（很少见，NAND 发生的次数要比 NOR 多），一个比特位会发生反转或被报告反转了。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。当然，如果这个位真的改变了，就必须采用错误探测/错误更正（EDC/ECC）算法。位反转的问题更多见于 NAND 闪存，NAND 的供应商建议使用 NAND 闪存的时候，同时使用 EDC/ECC 算法。这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用 EDC/ECC 系统以确保可靠性。

（6）坏块处理：NAND 器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

（7）易于使用：可以非常直接地使用基于 NOR 的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。由于需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异。在使用 NAND 器件时，必须先写入驱动程序，才能继续执行其他操作。向 NAND 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意

味着在 NAND 器件上自始至终都必须进行虚拟映射。

（8）软件支持：当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序（MTD），NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。使用 NOR 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 NOR 器件的更高级软件，这其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。驱动还用于对 DiskOnChip 产品进行仿真和 NAND 闪存的管理，包括纠错、坏块处理和损耗平衡。

2.3.2　Linux 内存管理

（1）alloca 配置内存空间。

函数定义：void *alloca(size_t,size)

功能说明：alloca()用来配置 size 个字节的内存空间，alloca()是从堆栈空间（Stack）中配置内存，因此在函数返回时会自动释放空间。

库头文件：＃include<stdlib.h>

返回值：若配置成功则返回指针，失败则返回 NULL。

（2）calloc 配置内存空间。

函数定义：void *calloc(size_t nmemb,size_t size)

功能说明：calloc()用来配置 nmemb 个相邻单位，每一个单位大小为 size，并返回指向第一个元素的指针。

库头文件：＃include<stdlib.h>

返回值：若配置成功则返回指针，失败则返回 NULL。

（3）malloc 配置内存空间。

函数定义：void malloc(size_t,size)

功能说明：malloc()用来配置内存空间大小，其大小由指定的 size 决定，如 void *p=malloc(1024);

库头文件：＃include<stdlib.h>

返回值：若配置成功则返回指针，失败则返回 NULL。

（4）realloc 更改已经配置的内存空间。

函数定义：void *realloc(void *ptr,size_t size)

功能说明：realloc 用来更改已经配置的内存空间，参数 ptr 为指向先前由 malloc、calloc 和 realloc 所返回的内存指针，而参数 size 为新配置的内存大小。

库头文件：＃include<stdlib.h>

返回值：若配置成功则返回指针，失败则返回 NULL。

（5）getpagesize 取得内存分页大小。

函数定义：size_t getpagesize(void)

功能说明：　返回内存一页的大小，单位为字节，此为系统的分页大小。

库头文件：#include<unistd.h>

返回值：内存分页的大小，在 intel X86 上为 4096 字节。

（6）free 释放原先配置的内存。

函数定义：void free(void *ptr)

功能说明：参数 ptr 为指向先前由 malloc 等函数所返回的内存指针，调用此函数即收回这一部分内存空间。

库头文件：＃include<stdlib.h>

2.4 设备管理

2.4.1 概述

Linux 设备驱动程序属于 Linux 内核的一部分，并在 Linux 内核中扮演着十分重要的角色。它们像一个个“黑盒子”使某个特定的硬件响应一个定义良好的内部编程接口，同时完全隐蔽了设备的工作细节。用户通过一组标准化的调用来完成相关操作，这些标准化的调用是和具体设备驱动无关的，而驱动程序的任务就是把这些调用映射到具体设备对于实际硬件的特定操作上。

我们可以将设备驱动作为内核的一部分，直接编译到内核中，即静态链接，也可以单独作为一个模块（module）编译，在需要它的时候再动态地把它插入到内核中。在不需要时也可把它从内核中删除，即动态链接。显然动态链接比静态链接有更多的好处，但在嵌入式开发领域往往要求进行静态链接，尤其是像 S3C44B0 这种不带 MMU 的芯片。但在 S3C2410 等带 MMU 的 ARM 芯片中依然可以使用动态链接。

目前 Linux 支持的设备驱动可分为 3 种：字符设备（Character Device）、块设备（Block Deivce）和网络接口设备（Network Interface）。当然它们之间也并不是要严格加以区分。

字符设备：所有能够像字节流一样访问的设备，比如，文件等在 Linux 中都通过字符设备驱动程序来实现。在 Linux 中它们也被映射为文件系统的一个节点，常在/dev 目录下。字符设备驱动程序一般要包含 open、close、read、write 等几个系统调用。

块设备：Linux 的块设备通常是指诸如磁盘、内存、Flash 等可以容纳文件系统的存储设备。与字符设备类似，块设备也是通过文件系统来进行访问，它们之间的区别仅仅在于内核内部管理数据的方式不同。它也允许像字符设备一样地访问，可以一次传递任意多的字节。Linux 中的块设备包含整数个块，每个块包含 2 的几次幂的字节。

网络接口设备：网络接口设备是 Linux 中比较复杂的一种设备，通常它们指的是硬件设备，但有时也可是一个软件设备（如回环接口 loopback）。它们由内核中网络子系统驱动，负责发送和接收数据包，而且它并不需要了解每一项事务是如何映射到实际传送的数据包的。由于它们的数据传送往往并不是面向流的（少数如 telnet、FTP 等是面向流的），所以不容易把它们映射到一个文件系统的节点上。在 Linux 中采用给网络接口设备分配一个唯一名字的方法来访问该设备。

2.4.2 字符设备与块设备

驱动程序在 Linux 内核中往往是以模块形式出现的。与应用程序的执行过程不同，模块通常只是预先向内核注册自己，当内核需要时响应请求。模块中包含两个重要的函数：init_module 和 cleanup_module。前者是模块的入口，它为模块调用做好准备工作，而后者

则是在模块即将卸载时被调用，做一些清理工作。

驱动程序模块通过以下函数来完成向内核注册的。

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

其中 unsigned int major 为主设备号，const char *name 为设备名，至于结构指针 struct file_operations *fops 在驱动程序中十分重要，还要做详细介绍。

在 Linux 中字符设备是通过文件系统中的设备名来进行访问的。这些名称通常放在/dev 目录下，通过命令 ls－l /dev 可以看到该目录下的设备文件，其中第一个字母是“C”的为字符设备，而第一个字母是“b”的为块设备文件。其中每个设备文件都具有一个主设备号（major）和一个次设备号（minor）。当驱动程序调用 open 系统调用时，内核就是利用主设备号把该驱动与具体设备对应起来的。而内核并不关心次设备号，它是给主设备号已经确定的驱动程序使用的，一个驱动程序往往可以控制多个设备，如一个硬盘的多个分区，这时该硬盘拥有一个主设备号，而每个分区拥有自己的次设备号。2.0 以前版本的内核支持 128 个主设备号，在 2.4 版内核中已经增加到 256 个，但显然是不够的。计算机技术在迅猛发展，各种外设也是层出不穷，这样主设备号就越来越显得是一种稀缺资源了。这也就给实际开发造成麻烦，为了解决这一问题，在 2.4 版本以后的内核中引入了 devfs 设备文件系统，它提供一套自动管理设备文件的手段，使得设备文件的管理容易了许多。在编写好一个驱动程序模块后，按传统的主次设备号的方法来进行设备管理，则应手工为该模块建立一个设备节点。命令如下所示。

```
mknod /dev/ts c 254 0
```

其中/dev/ts 表示设备名是 ts，“C”说明它是字符设备，“254”是主设备号，“0”是次设备号。一旦通过 mknod 创建了设备文件，它就一直保留下来，除非手工删除它。这里要注意的是，在 Linux 内核中有许多设备号已经静态地赋予一些常用设备，剩下可用的设备号已经不多。如果我们的设备也随意找一个空闲的设备号，并进行静态编译的话，当其他的开发者也采用类似手段分配设备号，那很快就会造成冲突。如何解决这个问题呢？比较好的方法就是采用动态分配的方法。在用 register_chrdev 注册模块时，给 major 赋值为 0，则系统就采用动态方式分配设备号。它会在所有未被使用的设备号中选定一个，作为函数返回值返回。一旦分配了设备号，就可以在/proc/devices 中看到相关内容。/proc 在前面关于操作系统移植的实验中已经提到，它是一个伪文件系统，它实际并不占用任何硬盘空间，而是在内核运行时在内存中动态生成的。它可以显示当前运行系统的许多相关信息。显然这一点对动态分配主设备号是非常有意义的。因为，正如前面提到的，采用主次设备号的方式管理设备文件，要在/dev 目录下为设备创建一个设备名，可设备号却是动态产生的，每次都不一样，这样就不得不每次都重新运行一次 mknod 命令。这个过程通常通过编写自动执行脚本来完成，而其中的主设备号就可以通过/proc/devices 中获得。当设备模块被卸载时，往往也会通过一个卸载脚本来显示删除/dev 中相关设备名，这是一个比较好的习惯，因为内核包找不到相关设备文件，总比内核找到一个错误的驱动去执行要好得多。

前面已经提到了 file_operations 这个结构。内核就是通过这个结构来访问驱动程序的。在内核中对于一个打开的文件包括一个设备文件，都用 file 结构来标志，而常给出一个 file_operations 类型的结构指针，一般命名为 fops 来指向该 file 结构。可以说 file 与

file_operations 这两个结构就是驱动设备管理中最重要的两个结构。在 file_operations 结构中每个字段都必须指向驱动程序中实现特定的操作函数。对于不支持的操作，对应字段就被置为 NULL。这样随着内核不断增加新功能，file_operations 结构也就变得越来越庞大。现在的内核开发人员采用一种叫“标记化”的方法来为该结构进行初始化。即对驱动中用到的函数记录到相应字段中，没有用到的就不管。这样代码就精简了许多。

结构体 file_operations 在头文件 linux/fs.h 中定义的，在内核 2.4 版内核中可以看到 file_operations 结构常是如下的一种定义：

```
 struct file_operations {
struct module *owner;//标示模块拥有者
loff_t (*llseek) (struct file *, loff_t, int);
//loff_t 是一个 64 位的长偏移数，llseek 方法标示当前文件的操作位置
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
//ssize_t（signed size）表示当前平台的固有整数类型。Read 是读函数
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
//写函数
int (*readdir) (struct file *, void *, filldir_t);
//readdir 方法用于读目录，其只对文件系统有效
unsigned int (*poll) (struct file *, struct poll_table_struct *);
//该方法用于查询设备是否可读，可写或处于某种状态。当设备不可读写时它们可以被阻塞直至设备变为可读或可写。如果驱动程序中没有定义该方法则它驱动的设备就会被认为是可读写的
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
//ioctl 是一个系统调用，它提供了一种执行设备特定命令的方法
int (*mmap) (struct file *, struct vm_area_struct *);
//该方法请求把设备内存映射到进程地址空间
int (*open) (struct inode *, struct file *);
//即打开设备文件，它往往是设备文件执行的第一个操作
int (*flush) (struct file *);
//进程在关闭设备描述符副本之前会调用该方法，它会执行设备上尚未完成的操作
int (*release) (struct inode *, struct file *);
//当 file 结构被释放时就会调用该方法
int (*fsync) (struct file *, struct dentry *, int datasync);
//该方法用来刷新待处理的数据
int (*fasync) (int, struct file *, int);
//即异步通知它是比较高级功能这里不作介绍
int (*lock) (struct file *, int, struct file_lock *);
//该方法用来实现文件锁定
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
//应用程序有时需要进行涉及多个内存区域的单次读写操作，利用该方法以及下面的 writev 可以完成这类操作
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

前面已经提到，目前采用的是“标记化”方法来为该结构赋值。下面要给出的代码中可以看到如下一段。

```
 static struct file_operations s3c44b0_fops = {
owner: THIS_MODULE,
open: s3c44b0_ts_open,
read: s3c44b0_ts_read,
release: s3c44b0_ts_release,
poll: s3c44b0_ts_poll,
};
```

它只对需要的函数赋值，对不需要的没有进行操作。这样使得代码结构更为清晰。

下面要讲到的是另一个重要的结构——file，它也定义在头文件 linux/fs.h 中。它代表一个打开的文件，由内核在调用 open 时创建。并传递给在该文件上进行操作的所有函数，直到最后的 close 函数被调用。在文件的所有实例都关闭时，内核释放这个数据结构。下面对其中一些重要字段做一些解释。

mode_t f_mode：该字段表示文件模式，它通过 FMODE_READ 和 FMODE_WRITE 位来标示文件是否可读，可写。

loff_t f_pos：该字段标示文件当前读写位置。

unsigned int_f_flags：这是文件标志，如 O_RDONLY、O_NONBLOCK、O_SYNC 等，驱动程序为了支持非阻塞型操作需要检查这个标志。

struct file_operations *f_op：这就是对前面介绍的 file_operations 结构的操作。内核在执行 open 操作时对这个指针赋值，以后需要处理这些操作时就读取这个指针。

void *private_data：这是个应用非常灵活的字段，驱动可以把它应用于任何目的，可以将它指向已经分配的数据，但一定要在内核销毁 file 结构前在 release 方法中释放该内存。

struct dentry *f_dentry：它对应一个目录项结构，是一种优化的设计。

2.4.3　主设备号和次设备号

传统方式中的设备管理中，除了设备类型外，内核还需要一对称做主次设备号的参数，才能唯一标识一个设备。主设备号相同的设备使用相同的驱动程序，次设备号用于区分具体设备的实例。

例如，PC 中的 IDE 设备，一般主设备号使用 3，Windows 下进行的分区，一般将主分区的次设备号为 1，扩展分区的次设备号为 2、3、4，逻辑分区使用 5、6 等。

设备操作宏 MAJOR()和 MINOR()可分别用于获取主次设备号，宏 MKDEV()用于将主设备号和次设备号合并为设备号，这些宏定义在 include/linux/kdev_t.h 中。

```
[root@yangzongde root]# cat /usr/src/linux-2.4.20-8/include/linux/kdev_
t.h
......
#define MINORBITS      8
#define MINORMASK      ((1U << MINORBITS) - 1)

typedef unsigned short kdev_t;

#define MAJOR(dev)     ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)     ((unsigned int) ((dev) & MINORMASK))
#define HASHDEV(dev)   ((unsigned int) (dev))
#define NODEV          0
#define MKDEV(ma,mi)   (((ma) << MINORBITS) | (mi))
#define B_FREE         0xffff          /* yuk */
......
```

对于 Linux 中对设备号的分配原则可以参考 Documentation/devices.txt。

对于查看/dev 目录下的设备的主次设备号可以使用如下命令：

```
[root@yangzongde root]# ls -l /dev/ |more
```

```
总用量 228
crw-------   1 root     root      10,  10 2003-01-30  adbmouse
crw-r--r--   1 root     root      10, 175 2003-01-30  agpgart
crw-------   1 root     root      10,   4 2003-01-30  amigamouse
crw-------   1 root     root      10,   7 2003-01-30  amigamouse1
crw-------   1 root     root      10, 134 2003-01-30  apm_bios
drwxr-xr-x   2 root     root         4096  3ÔÂ 22 05:30 ataraid
crw-------   1 root     root      10,   5 2003-01-30  atarimouse
crw-------   1 root     root      10,   3 2003-01-30  atibm
crw-------   1 root     root      10,   3 2003-01-30  atimouse
crw-------   1 root     root      14,   4 2003-01-30  audio
crw-------   1 root     root      14,  20 2003-01-30  audio1
crw-------   1 root     root      14,   7 2003-01-30  audioctl
brw-rw----   1 root     disk      29,   0 2003-01-30  aztcd
crw-------   1 root     root      10, 128 2003-01-30  beep
brw-rw----   1 root     disk      41,   0 2003-01-30  bpcd
crw-------   1 root     root      68,   0 2003-01-30  capi20
crw-------   1 root     root      68,   1 2003-01-30  capi20.00
crw-------   1 root     root      68,   2 2003-01-30  capi20.01
crw-------   1 root     root      68,   3 2003-01-30  capi20.02
crw-------   1 root     root      68,   4 2003-01-30  capi20.03
crw-------   1 root     root      68,   5 2003-01-30  capi20.04
......
```

设备类型、主次设备号是内核与设备驱动程序通信时所使用的，但是由于对于开发应用程序的用户来说比较难于理解和记忆，所以 Linux 使用了设备文件的概念来统一对设备的访问接口，在引入设备文件系统（devfs）之前 Linux 将设备文件放在/dev 目录下，设备的命名一般为设备文件名＋数字或字母表示的子类，例如/dev/hda1、/dev/hda2 等。

在 Linux 2.4 及以后内核中引入了设备文件系统（devfs），所有的设备文件作为一个可以挂装的文件系统，这样就可以被文件系统进行统一管理，从而设备文件就可以挂装到任何需要的地方。命名规则也发生了变化，一般将主设备建立一个目录，再将具体的子设备文件建立在此目录下。

2.5 本章总结

用户要进行程序开发，熟悉系统开发环境是非常重要的。本章重点介绍了 Linux 系统开发的环境平台，主要包括进程/线程管理、文件系统结构和类型、存储管理、设备管理等知识。通过本章的学习，读者对 Linux 系统开发的环境平台有较深刻的了解，有助于在以后的程序设计和系统开发中更加得心应手。

第 3 章

嵌入式 Linux 程序设计基础

Linux 操作系统虽然在嵌入式设备上得到了广泛运用，但是，Linux 操作系统最初并不是为嵌入式设备所设计的，因此，要在相应的嵌入式设备中运行 Linux 操作系统，需要建立适合相关应用环境的交叉编译环境。本章将介绍嵌入式 Linux 开发环境及程序设计基础。

3.1 建立嵌入式 Linux 交叉编译环境

3.1.1 编译环境概述

因为应用的硬件环境不一样，在台式机上用 X86 的 GCC 编译的可执行文件是不能在嵌入式 ARM 处理器上运行的，因为 X86 处理器和 ARM 处理器是不同的硬件平台，即硬件体系结构存在差异。但是，由于目前嵌入式设备本身资源的特点，应用于 ARM 处理器上的程序又不得不借助 X86 的台式机进行编辑和编译工作。因此，在 X86 的台式机上编译能够在 ARM 处理器上运行的应用程序就不能用于 X86 处理器的 GCC，而是需要一组特殊的编译环境，这就是本章要讨论的交叉编译环境。

GCC 工具链是目前最受欢迎，应用最为广泛的交叉编译环境，支持绝大多数嵌入式处理器，包括 ARM、MIPS、SuperH、PowerPC 以及 X86。目前，网络已经有很多成功建立交叉编译环境的方式。以下详细介绍这一组工具特点。

1．Binutils 二进制工具包

Binutils 二进制工具包是一组二进制处理工具的集合，主要工具有以下两种。

- ld：链接器。
- as：GNU 汇编器。

除此之外，还包括以下工具。

- addr2line：把程序地址转换为文件名和行号，在命令行中给出一个地址和可执行文件，其就会使用这个可执行文件的调试信息给出的地址上的文件及行号。
- ar：建立、修改以及提取归档文件。
- c++filt：连接器使用它来过滤 C++符号。
- gprof：显示程序调用段的信息。
- nlmconv：转换目标文件为 NLM。
- nm：列出目标文件中的符号。
- objcopy：复制目标文件。

- objdump：显示一个或多个目标文件信息。
- ranlib：产生归档文件索引，并将其保存到这个归档文件中。
- readelf：显示 elf 格式可执行文件信息。
- size：列出目标文件每一段的大小及总体大小。
- strings：打印某个文件的可打印字符串。
- strip：丢弃目标文件中的全部或者部分符号。
- windres：一个 Windows 资源的编译器。

2．GCC 交叉编译器

GCC 工具是编译程序的最为重要的工具，其包括以下几个主要工具。

- Cpp：C 预处理器。
- g++：C++编译器。
- gcc：C 编译器。
- gccbug：创建 bug 报告的 Shell 脚本。
- gcov：分析在程序中哪里做优化效果最好。
- libgcc*：gcc 的运行库。
- libstdc++：标准 C++库，包含许多常用的函数。
- libsupc++：提供支持 C++语言的库函数。

如果是专门用于 ARM 处理器，这些工具名称前面一般会加上 armv4l-unknown-linux-，即分别为 armv4l-unknown-linux-addr2line、armv4l-unknown-linux-g++、armv4l-unknown-linux-objcopy、armv4l-unknown-linux-size、armv4l-unknown-linux-ar、armv4l-unknown-linux-gasp、armv4l-unknown-linux-objdump、armv4l-unknown-linux-strings、armv4l-unknown-linux-as、armv4l-unknown-linux-gcc、armv4l-unknown-linux-protoize、armv4l-unknown-linux-strip、armv4l-unknown-linux-c++、armv4l-unknown-linux-gdb、armv4l-unknown-linux-ranlib、armv4l-unknown-linux-unprotoize、armv4l-unknown-linux-c++filt、armv4l-unknown-linux-ld、armv4l-unknown-linux-readelf、armv4l-unknown-linux-cpp、armv4l-unknown-linux-nm、armv4l-unknown-linux-run。

3．Glibc 库说明

Glibc 库提供了系统调用和基本函数的 C 库，所有动态链接的程序都要用到它。相关版本可下载的地址如下。

- Glibc (2.3.2): ftp://ftp.gnu.org/gnu/glibc/
- Glibc-linuxthreads (2.3.2): ftp://ftp.gnu.org/gnu/glibc/
- GlibcSscanfPatch:http://www.linuxfromscratch.org/patches/lfs/cvs/glibc-2.3.2-sscanf-1.patch

Glibc 中主要有下列程序。

catchsegv：当程序发生 segmentation fault 的时候，用来建立一个堆栈跟踪。

gencat：建立消息列表。

getconf：针对文件系统的指定变量显示其系统设置值。

getent：从系统管理数据库获取一个条目。

glibcbug：建立 glibc 的 bug 报告并且 E-mail 到 bug 报告的邮件地址。

iconv：转化字符集。

iconvconfig：建立快速读取的 iconv 模块所使用的设置文件。

ldconfig：设置动态链接库的实时绑定。

ldd：列出每个程序或者命令需要的共享库。

lddlibc4：辅助 ldd 操作目标文件。

locale：是一个 Perl 程序，可以告诉编译器打开或关闭内建的 locale 支持。

localedef：编译 locale 标准。

nscd：提供对常用名称设备调用的缓存的守护进程。

nscd_nischeck：检查在进行 NIS+侦查时是否需要安全模式。

pcprofiledump：打印 PC profiling 产生的信息。

pt_chown：是一个辅助程序，帮助 grantpt 设置子虚拟终端的属主、用户组和读写权限。

rpcgen：产生实现 RPC 协议的 C 代码。

rpcinfo：对 RPC 服务器产生一个 RPC 呼叫。

sln：用来创建符号链接，由于它本身是静态链接的，在动态链接不起作用时，sln 仍然可以建立符号链接。

sprof：读取并显示共享目标的特征描述数据。

tzselect：对用户提出关于当前位置的问题，并输出时区信息到标准输出。

xtrace：通过打印当前执行的函数跟踪程序执行情况。

zdump：显示时区。

zic：时区编译器。

以下为相关的库文件。

ld.so：帮助动态链接库的执行。

libBrokenLocale：帮助程序处理破损 locale，如 Mozilla。

libSegFault：处理 segmentation fault 信号，试图捕捉 segfaults。

libanl：异步名称查询库。

libbsd-compat：为了在 Linux 下执行一些 BSD 程序，libbsd-compat 提供了必要的可移植性。

libc：是主要的 C 库——常用函数的集成。

libcrypt：加密编码库。

libdl：动态链接接口。

libg g++：g++的运行时。

libieee IEEE：浮点运算库。

libm：数学函数库。

libmcheck：包括了启动时需要的代码。

libmemusage：帮助 memusage 搜集程序运行时内存占用的信息。

libnsl：网络服务库。

libnss*：名称服务切换库，包含了解释主机名、用户名、组名、别名、服务、协议等的函数。

libpcprofile：帮助内核跟踪在函数、源代码行和命令中 CPU 使用时间。

libpthread：POSIX 线程库。

libresolv：创建、发送及解释到因特网域名服务器的数据包。
librpcsvc：提供 RPC 的其他服务。
libr：提供了大部分的 POSIX.1b 实时扩展的接口。
libthread_db：对建立多线程程序的调试很有用。
libutil：包含了在很多不同的 UNIX 程序中使用的“标准”函数。

3.1.2 建立交叉编译环境流程

在裁剪和定制 Linux 运用于嵌入式系统之前，由于嵌入式开发系统存储大小有限，通常需要在 PC 上建立一个用于目标板的交叉编译环境。这是一个由编译器、连接器和解释器组成的综合开发环境。交叉编译工具主要由 binutils、gcc 和 glibc 几个部分组成。有时出于对减小 libc 库大小的考虑，也可以用别的 c 库来代替 glibc，例如 uClibc、dietlibc 和 newlib。建立一个交叉编译工具链是一个比较复杂的过程，如果用户不想自己经历复杂的编译过程，网上有一些编译好的可用的交叉编译工具链可以直接下载。

下面将以建立针对 ARM 的交叉编译开发环境为例来说明整个过程，其他的体系结构与这个过程相类似，只要做一些对应的改动。这里的开发环境是宿主机 i386-redha，目标板为 ARM 试验箱。这个过程如下：

- 下载源文件、补丁和建立编译的目录；
- 建立内核头文件；
- 建立二进制工具（binutils）；
- 建立初始编译器（bootstrap gcc）；
- 建立 c 库（glibc）；
- 建立全套编译器（full gcc）。

1. 下载源文件、补丁和建立编译的目录

选择软件版本号时，先看看 glibc 源代码中的 INSTALL 文件。那里列举了该版本的 glibc 编译时所需的 binutils 和 gcc 的版本号。各个软件的版本及可下载地址为：

```
binutils-2.14.tar.gz：
ftp://ftp.gnu.org/gnu/binutils/binutils-2.14.tar.gz
gcc-core-2.95.3.tar.gz：
ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3/gcc-core-2.95.3.tar.gz
gcc-g++2.95.3.tar.gz：
ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3/gcc-g++-2.95.3.tar.gz
glibc-2.2.4.tar.gz：
ftp://ftp.gnu.org/gnu/glibc/glibc-2.2.4.tar.gz
glibc-linuxthreads-2.2.4.tar.gz ：
ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.2.4.tar.gz
linux-2.4.21.tar.gz：
ftp://ftp.kernle.org/pub/linux/kernel/v2.4/linux-2.4.21.tar.gz
patch-2.4.21-rmk1.gz：
ftp://ftp.arm.linux.org.uk/pub/linux/arm/kernel/v2.4/patch-2.4.21-rmk1.gz
```

接着建立工作目录：首先建立以下几个用来工作的目录。

在读者的用户目录（编者用的是用户 yangzongde，用户目录为 /home/yangzongde）下，先建立一个项目目录 embedded。

```
$pwd
/home/yangzongde
$mkdir embedded
```

再在这个项目目录 embedded 下建立 3 个目录 build-tools、kernel 和 tools。

- build-tools：用来存放下载的 binutils、gcc 和 glibc 源代码和用来编译这些源代码的目录。
- kernel：用来存放内核源代码和内核补丁。
- tools：用来存放编译好的交叉编译工具和库文件。

```
$cd embedded
$mkdir  build-tools kernel tools
```

执行完后目录结构如下：

```
$ls embedded
build-tools kernel tools
```

接下来就需要输出和环境变量以方便编译。其内容如下：

```
$export PRJROOT=/home/yangzongde/embedded
$export TARGET=arm-linux
$export PREFIX=$PRJROOT/tools
$export TARGET_PREFIX=$PREFIX/$TARGET
$export PATH=$PREFIX/bin:$PATH
```

如果读者不习惯用环境变量，可以直接用绝对或相对路径。另外，可以通过 glibc 下的 config.sub 脚本知道 TARGET 变量是否被支持，例如：

```
$./config.sub  arm-linux
arm-unknown-linux-gnu
```

接下来建立编译目录：为了将源码和编译时生成的文件分开，一般的编译工作不在源代码目录中，要另建一个目录来专门用于编译。用以下的命令来建立编译下载的 binutils、gcc 和 glibc 的源代码的目录。

```
$cd $PRJROOT/build-tools
$mkdir build-binutils build-boot-gcc build-gcc build-glibc gcc-patch
```

- build-binutils：编译 binutils 的目录。
- build-boot-gcc：编译 gcc 启动部分的目录。
- build-glibc：编译 glibc 的目录。
- build-gcc：编译 gcc 全部的目录。
- gcc-patch：放 gcc 的补丁的目录。gcc-2.95.3 的补丁有 gcc-2.95.3-2.patch、gcc-2.95.3-no-fixinc.patch 和 gcc-2.95.3-returntype-fix.patch，可以从 http://www.linuxfromscratch.org/下载到这些补丁。

再将下载的 binutils-2.10.1、gcc-2.95.3、glibc-2.2.3 和 glibc-linuxthreads-2.2.3 源代码放入 build-tools 目录中，查看 build-tools 目录，有以下内容。

```
$ls
binutils-2.10.1.tar.bz2    build-gcc           gcc-patch
build-binutls         build-glibc         glibc-2.2.3.tar.gz
build-boot-gcc        gcc-2.95.3.tar.gz glibc-linuxthreads-2.2.3.tar.gz
```

2．建立内核头文件

把从 http.// www.kernel.org 下载的内核源代码放入$PRJROOT /kernel 目录，进入 kernel 目录。

```
$cd $PRJROOT /kernel
```

解开内核源代码

```
$tar -xzvf linux-2.4.21.tar.gz
```

或

```
$tar -xjvf linux-2.4.21.tar.bz2
```

小于 2.4.19 的内核版本解开会生成一个 Linux 目录，若没带版本号，就将其改名。

```
$mv linux linux-2.4.x
```

给 Linux 内核打上补丁

```
$cd linux-2.4.21
$patch -p1 < ../patch-2.4.21-rmk2
```

编译内核生成头文件

```
$make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

也可以用 config 和 xconfig 来代替 menuconfig，但这样用可能会没有设置某些配置文件选项和没有生成下面编译所需的头文件。推荐大家用 make menuconfig，这也是内核开发人员用得最多的配置方法。配置完退出并保存，检查一下内核目录中的 include/linux/version.h 和 include/linux/autoconf.h 文件是不是已经生成，这是编译 glibc 是要用到的 version.h 和 autoconf.h 文件，如果它们存在，也说明生成了正确的头文件。另外，还要建立几个正确的链接。

```
$cd include
$ln -s asm-arm asm
$cd asm
$ln -s arch-epxa arch
$ln -s proc-armv proc
```

接下来为交叉编译环境建立内核头文件的链接。

```
$mkdir -p $TARGET_PREFIX/include
$ln -s $PRJROOT/kernel/linux-2.4.21/include/linux  $TARGET_PREFIX/include
/linux
$in  -s  $PRJROOT/kernel/linux-2.4.21/include/asm-arm    $TARGET_PREFIX/
include/asm
```

也可以把 Linux 内核头文件复制过来用。

```
$mkdir -p $TARGET_PREFIX/include
$cp -r $PRJROOT/kernel/linux-2.4.21/include/linux  $TARGET_PREFIX/include
$cp -r $PRJROOT/kernel/linux-2.4.21/include/asm-arm  $TARGET_PREFIX/include
```

3．建立二进制工具（binutils）

binutils 是一些二进制工具的集合，其中包含了常用到的 as 和 ld。首先，解压下载的 binutils 源文件。

```
$cd $PRJROOT/build-tools
$tar -xvjf binutils-2.10.1.tar.bz2
```

然后进入 build-binutils 目录配置和编译 binutils。

```
$cd build-binutils
$../binutils-2.10.1/configure --target=$TARGET --prefix=$PREFIX
```

其中：

--target 选项是指出生成的是 arm-linux 的工具；

--prefix 是指出可执行文件安装的位置。

完成后，生成 Makefile 文件。有了 Makefile 后，编译并安装 binutils，命令很简单。

```
$make
$make install
```

下面是$PREFIX/bin 下的生成的文件。

```
$ls $PREFIX/bin
arm-linux-addr2line     arm-linux-gasp      arm-linux-objdump
arm-linux-strings
arm-linux-ar            arm-linux-ld        arm-linux-ranlib
arm-linux-strip
arm-linux-as            arm-linux-nm        arm-linux-readelf
arm-linux-c++filt       arm-linux-objcopy   arm-linux-size
```

4．建立初始编译器（bootstrap gcc）

首先进入 build-tools 目录，将下载 gcc 源代码解压。

```
$cd $PRJROOT/build-tools
$tar -xvzf  gcc-2.95.3.tar.gz
```

然后进入 gcc-2.95.3 目录给 gcc 打上补丁。

```
$cd gcc-2.95.3
$patch -p1< ../gcc-patch/gcc-2.95.3.-2.patch
$patch -p1< ../gcc-patch/gcc-2.95.3.-no-fixinc.patch
$patch -p1< ../gcc-patch/gcc-2.95.3-returntype-fix.patch
echo timestamp > gcc/cstamp-h.in
```

在编译并安装 gcc 前，先要改一个文件$PRJROOT/gcc/config/arm/t-linux，把

```
TARGET_LIBGCC2-CFLAGS = -fomit-frame-pointer -fPIC
```

这一行改为：

```
TARGET_LIBGCC2-CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc -D__gthr_
posix_h
```

如果没定义-Dinhibit，编译时将会报如下的错误。

```
../../gcc-2.95.3/gcc/libgcc2.c:41: stdlib.h: No such file or directory
../../gcc-2.95.3/gcc/libgcc2.c:42: unistd.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
make[2]: *** [stmp-multilib-sub] Error 2
make[1]: *** [stmp-multilib] Error 1
make: *** [all-gcc] Error 2
```

如果没有定义-D__gthr_posix_h，编译时会报如下的错误。

```
In file included from gthr-default.h:1,
from ../../gcc-2.95.3/gcc/gthr.h:98,
from ../../gcc-2.95.3/gcc/libgcc2.c:3034:
../../gcc-2.95.3/gcc/gthr-posix.h:37: pthread.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
make[2]: *** [stmp-multilib-sub] Error 2
make[1]: *** [stmp-multilib] Error 1
make: *** [all-gcc] Error 2
```

还有一种与-Dinhibit 同等效果的方法，那就是在配置 configure 时多加一个参数-with-newlib，这个选项不会迫使我们必须使用 newlib。编译 bootstrap-gcc 后，仍然可以选择任何 c 库。接着就是配置 boostrap gcc，因为后面要用 bootstrap gcc 来编译 glibc 库。

```
$cd ..; cd build-boot-gcc
$../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX >--without-headers
--enable-languages=c --disable-threads
```

这条命令中的--target、--prefix 和配置 binutils 的含义是相同的，--without-headers 就是指不需要头文件，因为是交叉编译工具，不需要本机上的头文件。-enable-languages=c 是指 boot-gcc 只支持 c 语言。--disable-threads 是去掉 thread 功能，这个功能需要 glibc 的支持。

接着编译并安装 boot-gcc。

```
$make all-gcc
$make install-gcc
```

下面来看看$PREFIX/bin 里面多了哪些东西。

```
$ls $PREFIX/bin
```

从而会发现多了 arm-linux-gcc、arm-linux-unprotoize、cpp 和 gcov 几个文件。

- Gcc：gnu 的 C 语言编译器。
- Unprotoize：将 ANSI C 的源代码转化为 K&R C 的形式，去掉函数原型中的参数类型。
- Cpp：gnu 的 C 的预编译器。
- Gcov：gcc 的辅助测试工具，可以用它来分析和优化程序。

5．建立 c 库（glibc）

首先解压 glibc-2.2.3.tar.gz 和 glibc-linuxthreads-2.2.3.tar.gz 源代码。

```
$cd $PRJROOT/build-tools
$tar -xvzf glibc-2.2.3.tar.gz
$tar -xzvf glibc-linuxthreads-2.2.3.tar.gz --directory=glibc-2.2.3
```

然后进入 build-glibc 目录配置 glibc。

```
$cd build-glibc
$CC=arm-linux-gcc ../glibc-2.2.3/configure --host=$TARGET --prefix="/usr"
--enable-add-ons --with-headers=$TARGET_PREFIX/include
```

其中：

- CC=arm-linux-gcc 是把 CC 变量设成刚编译完的 boostrap gcc，用它来编译 glibc。
- --enable-add-ons 告诉 glibc 用 linuxthreads 包，在上面已经将它放入了 glibc 源代码目录中，这个选项等价于 Linuxthreads，即 -enable-add-ons=linuxthreads。

- --with-headers 告诉 glibc 当前 Linux 内核头文件的目录位置。配置完后就可以编译和安装 glibc。

```
$make
$make install_root=$TARGET_PREFIX prefix="" install
```

然后还要修改 libc.so 文件，将

```
GROUP ( /lib/libc.so.6 /lib/libc_nonshared.a)
```

改为

```
GROUP ( libc.so.6 libc_nonshared.a)
```

这样，连接程序 ld 就会在 libc.so 所在的目录查找它需要的库，因为/lib 目录可能已经装了一个相同名字的库，一个为编译可以在宿主机上运行的程序的库，而不是用于交叉编译的。

6．建立全套编译器（full gcc）

在建立 boot-gcc 的时候，只支持了 C 语言。到这里，这就要建立全套编译器，来支持 C 和 C++。

```
$cd $PRJROOT/build-tools/build-gcc
$../gcc-2.95.3/configure  --target=$TARGET  --prefix=$PREFIX  --enable-languages=c,c++
```

其中--enable-languages=c,c++用来告诉 full gcc 支持 c 和 c++语言。

然后编译和安装 full gcc。

```
$make all
$make install
```

此时再来看看$PREFIX/bin 里面多了哪些内容。

```
$ls $PREFIX/bin
```

会发现多了 arm-linux-g++、arm-linux-protoize 和 arm-linux-c++几个文件。

其中各参数的含义如下。

- G++：gnu 的 c++编译器。
- Protoize：与 Unprotoize 相反，将 K&R C 的源代码转化为 ANSI C 的形式，函数原型中加入参数类型。
- C++：gnu 的 c++编译器。

完成以上操作后，基于 Linux 的 ARM 交叉编译环境建立即完成了。

3.2　工程管理器 make

3.2.1　make 概述

无论是在 Linux 还是在 UNIX 环境中，make 都是一个非常重要的编译命令。无论是自己进行项目开发还是安装应用软件，都需要使用 make 或 make install 工具。利用 make 工具，可以将大型的开发项目分解成多个更易于管理的模块，对于一个包括几百个源文件的

应用程序，使用 make 和 makefile 工具就可以清晰地理顺各个源文件之间的关系，而且如此多的源文件，如果每次都要键入 gcc 命令进行编译的话，这对程序员来说是无法忍受的。而 make 工具可自动完成编译工作，并且可以只对程序员在上次编译后修改过的部分进行编译。因此，有效地利用 make 和 makefile 工具可以大大提高项目开发的效率。

1．make 是如何工作的

make 工具最基本的功能就是通过 makefile 文件来描述源程序之间的相互依赖关系，并自动维护编译工作。当然，makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，要求定义源文件之间的依赖关系。makefile 文件是许多编译器（包括 Windows 下的编译器）维护编译信息的常用方法，只是在集成开发环境中，用户可以通过友好的界面修改 makefile 文件而已。

例如，在当前目标下有一个文件名为“makefile”的文件，其内容如下（此文件的语法结构在接下来的内容中介绍）。

```
    #It is a example for describing makefile
    edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
         cc -o edit main.o kbd.o command.o display.o  insert.o search.o files.o
utils.o
    main.o : main.c defs.h
        cc -c main.c
    kbd.o : kbd.c defs.h command.h
         cc -c kbd.c
    command.o : command.c defs.h command.h
         cc -c command.c
    display.o : display.c defs.h buffer.h
         cc -c display.c
    insert.o : insert.c defs.h buffer.h
         cc -c insert.c
    search.o : search.c defs.h buffer.h
         cc -c search.c
    files.o : files.c defs.h buffer.h command.h
        cc -c files.c
    utils.o : utils.c defs.h
        cc -c utils.c
    clean :
       rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

这个描述文档就是一个简单的 makefile 文件。在这个例子中：

第一个字符为#的行为注释行。

第一个非注释行指定 edit 由目标文件 main.o kbd.o command.o display.o insert.o search.o files.o utils.o 链接生成，这只是说明一个依赖关系。

第三行描述了如何从 edit 所依赖的文件建立可执行文件，即执行命令 cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o”，即调用 gcc 编译以上.o 文件生成 edit 可执行文件。

接下来的各行分别指定各自的目标文件，以及它们所依赖的.c 和.h 文件。而紧跟的依赖关系行则指定了如何从目标所依赖的文件中建立目标。

在默认的方式下，在当前目录提示符下输入“make”命令。系统将自动完成以下操作。

（1）make 会在当前目录下寻找名字为“Makefile”或“makefile”的文件。

（2）如果找到，它会查找文件中的第一个目标文件（target），在上面的例子中，系统将查找到“edit”这个目标，并把这个文件作为最终的目标文件。

（3）如果 edit 文件不存在，或是 edit 所依赖的后面的.o 文件的文件修改时间要比 edit 这个文件新，那么，系统就会执行后面所定义的命令来生成 edit 这个文件。

（4）如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到，则根据这一个规则生成.o 文件。

（5）如果此文件中列出的*.C 文件和*.H 文件都存在，于是 make 会首先生成.o 文件，然后再用.o 文件生成可执行文件 edit。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在查找的过程中，如果出现错误，比如，最后被依赖的文件找不到，那么 make 就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性。

通常，makefile 中还定义 clean 目标，可用来清除编译过程中的中间文件，例如上例中的内容：

```
clean :
    rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

另外更为简单的方法为：

```
clean:
    rm -f *.o
```

在上述 makefile 文件中，像 clean 这个没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，可以在命令中要求执行，即使用命令“make clean”，以此来清除所有的目标文件，以便重编译。

在 Linux 系统中，习惯使用 Makefile 作为 makefile 文件。如果要使用其他文件作为 makefile，则可利用以下 make 命令选项来指定 makefile 文件。

```
$ make -f filename
```

2．Makefile 文件

Makefile 文件中主要包含了 5 项内容：显式规则、隐晦规则、变量定义、文件指示和注释。

（1）显式规则。显式规则说明如何生成一个或多个目标文件。这是由 Makefile 的书写者明确指出要生成的文件、文件的依赖文件、生成的命令。

（2）隐晦规则。由于 make 有自动推导的功能，所以隐晦的规则可以让程序员比较粗糙、简略地书写 Makefile，这是 make 所支持的。

（3）变量的定义。在 Makefile 中需要定义一系列的变量，变量一般都是字符串，这点类似于 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

（4）文件指示。包括 3 部分，一个是在一个 Makefile 文件中引用另一个 Makefile 文件，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；另外还可以定义一个多行的命令。

（5）注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，类似于 C/C++中的“//”。如果需要在 Makefile 中使用“#”字符，可以用反斜框进行

转义，如：“\#”。

从语法结构中来看，Makefile 文件作为一种描述文档一般需要包含以下内容。

- 宏定义。
- 源文件之间的相互依赖关系。
- 可执行的命令。

Makefile 中允许使用简单的宏指代源文件及其相关编译信息，在 Linux 中也称宏为变量。在引用宏时只需在变量前加$符号，但值得注意的是，如果变量名的长度超过一个字符，在引用时就必须加圆括号()。

下面都是有效的宏引用：

```
$(CFLAGS)
$2
$Z
$(Z)
```

最后两个引用是完全一致的。需要注意的是一些预定义宏，在 Linux 系统中，$*、$@、$?和$<4 个特殊宏的值在执行命令的过程中会发生相应的变化，而在 GNU make 中定义了更多的预定义变量。例如：

```
# Define a macro for the object files
OBJECTS= filea.o fileb.o filec.o
# Define a macro for the library file
LIBES= -LS
# use macros rewrite makefile
prog: $(OBJECTS)
cc $(OBJECTS) $(LIBES) -o prog
    ……
```

此时如果执行不带参数的 make 命令，将连接 3 个目标文件和库文件 LS；但是如果在 make 命令后带有新的宏定义：

```
make "LIBES= -LL -LS"
```

则命令行后面的宏定义将覆盖 makefile 文件中的宏定义。若 LL 也是库文件，此时 make 命令将连接 3 个目标文件及两个库文件 LS 和 LL。

3．Make 命令

Make 命令可带有 4 种参数：标志、宏定义、描述文件名和目标文件名。其标准形式为：

```
Make [flags] [macro definitions] [targets]
```

Linux 系统下标志位 flags 选项及其含义如下。

-f file：指定 file 文件为描述文件，如果 file 参数为“-”符，那么描述文件指向标准输入。如果没有“-f”参数，则系统将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文件。

-i：忽略命令执行返回的出错信息。

-s：沉默模式，在执行之前不输出相应的命令行信息。

-r：禁止使用 build-in 规则。

-n：非执行模式，输出所有执行命令，但并不执行。

-t：更新目标文件。

-q：make 操作将根据目标文件是否已经更新返回“0”或非“0”的状态信息。

-p：输出所有宏定义和目标文件描述。

-d：Debug 模式，输出有关文件和检测时间的详细信息。

-c di：在读取 makefile 之前改变到指定的目录 dir。

-I dir：包含其他 makefile 文件时，利用该选项指定搜索目录。

-h：help 文档，显示所有的 make 选项。

-w：在处理 makefile 之前和之后，都显示工作目录。

通过命令行参数中的 target，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中第一个目标文件。

3.2.2　Makfile 文件书写规则

在 Makefile 中规则的顺序是很重要的，因为 Makefile 中只应该有一个最终目标，其他的目标都是被这个目标所引入的，所以一定要让 make 知道最终目标是什么。一般来说，定义在 Makefile 中的目标可能会有很多，则第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make 所完成的也就是这个目标。

1．规则举例

```
foo.o : foo.c defs.h       # foo 模块
       cc -c -g foo.c
```

在此例中，foo.o 是目标，foo.c 和 defs.h 是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（此行一定要以 Tab 键开头）。这个规则有以下两个主要内容。

（1）文件的依赖关系，foo.o 依赖于 foo.c 和 defs.h 的文件，如果 foo.c 和 defs.h 的文件日期比 foo.o 文件日期要新，或是 foo.o 不存在，那么依赖关系发生。

（2）第二行的 cc 命令说明了如何生成 foo.o 这个文件（当然 foo.c 文件 include 了 defs.h 文件）。

2．规则的语法

```
targets : prerequisites
  command
  ...
```

或是这样：

```
targets : prerequisites ; command
     command
     ...
```

targets 是文件名。一般来说，目标基本上是一个文件，但也有可能是多个文件，以空格分开，可以使用通配符。

command 是命令行，如果不与“target:prerequisites”在一行，那么，必须以 Tab 键开

头，如果和 prerequisites 在一行，那么可以用分号作为分隔。

prerequisites 也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件比目标文件新（时间），那么，目标就被认为是“过时的”，被认为是需要重新生成的。如果命令太长，可以使用反斜框（‘\’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何生成目标文件。

3．在规则中使用通配符

如果想定义一系列比较类似的文件，很自然地就想起使用通配符。make 支持 3 种通配符：“*”，“?”和“[...]”。这与 B-Shell 是相同的。另外，波浪号（“~”）字符在文件名中也有比较特殊的用途。如果是“~/test”，这就表示当前用户的$HOME 目录下的 test 目录。而“~yzd/test”则表示用户 yzd 的宿主目录下的 test 目录。

通配符代替了一系列内容，如“*.c”表示所有以后缀为 c 的文件。需要注意的是，如果文件名中有通配符，如：“*”，则可以用转义字符“\”，如“*”来表示真实的“*”字符。

通配符使用在规则中：

```
print: *.c
    lpr -p $?
    touch print
```

目标 print 依赖于所有的[.c]文件。

通配符同样用在变量中：

```
objects = *.o
```

在这里，objects 的值就是“*.o”。Makefile 中的变量其实就是 C/C++中的宏。如果需要让通配符在变量中展开，也就是让 objects 的值是所有[.o]的文件名的集合，那么，可以这样定义：

```
objects := $(wildcard *.o)
```

4．文件搜寻

在一些大的工程中，有大量的源文件，由于通常的做法是将这许多的源文件分类存放在不同的目录中。所以，当 make 需要去找寻文件的依赖关系时，可以在文件前加上路径，但最好的方法是把路径告诉 make，让 make 自动查找。

Makefile 文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make 只会在当前的目录中去查找依赖文件和目标文件。如果定义了这个变量，那么，make 在当前目录找不到相关文件的情况下，会自动到所指定的目录中查找文件。

```
VPATH = src:../headers
```

这句代码指定两个目录，“src”和“../headers”，make 会按照这个顺序进行搜索。目录由“冒号”分隔。

另一个设置文件搜索路径的方法是使用 make 的“vpath”关键字（注意，是小写的），这不是变量，而是一个 make 的关键字，这与上面提到的那个 VPATH 变量很类似，但是它更为灵活，它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能，它的使用方法有 3 种：

（1）vpath <pattern> <directories>：为符合模式<pattern>的文件指定搜索目录<directories>。

（2）vpath <pattern>：清除符合模式<pattern>的文件的搜索目录。

（3）vpath：清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零个或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了<pattern>的文件集的搜索的目录。例如：

```
vpath %.h ../headers
```

表示要求 make 在“../headers”目录下搜索所有以“.h”结尾的文件（如果某文件在当前目录没有找到的话）。

程序员可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的<pattern>，或是被重复了的<pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。如：

```
vpath %.c foo
vpath %.c blish
vpath %.c bar
```

其表示“.c”结尾的文件，先在“foo”目录，然后是“blish”，最后是“bar”目录。

5．伪目标

在前面内容中提到过一个“clean”的目标，这是一个“伪目标”。

```
clean:
	rm *.o temp
```

此段语法并不是要生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以 make 无法生成它的依赖关系和决定它是否要执行。因此，只有通过指明这个“目标”才能让其生效（“伪目标”的取名不能和文件名重名）。为了避免和文件重名的这种情况，可以使用一个特殊的标记“.PHONY”来指明一个目标是“伪目标”，向 make 说明，不管是否有这个文件，这个目标就是“伪目标”。即

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”命令有效。即整个这段代码为：

```
 .PHONY: clean
clean:
	rm *.o temp
```

一般情况下，伪目标没有依赖的文件。但是，也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。例如，Makefile 文件需要生成若干个可执行文件，但用户又只想简单地敲一个 make 命令，并且，所有的目标文件都写在一个 Makefile 文件中，那么此时可以使用“伪目标”这个特性。

```
all : prog1 prog2 prog3
```

```
.PHONY : all
prog1 : prog1.o utils.o
	cc -o prog1 prog1.o utils.o
prog2 : prog2.o
	cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
	cc -o prog3 prog3.o sort.o utils.o
```

Makefile 中的第一个目标会被作为其默认目标。这里声明了一个“all”的伪目标，其依赖于其他 3 个目标。由于伪目标总是被执行的，所以其依赖的那 3 个目标就总是不如“all”这个目标新。所以，其他 3 个目标的规则总是会被执行。“.PHONY : all”声明了“all”这个目标为“伪目标”。

6．多目标

Makefile 规则中的目标可以不止一个，即支持多目标，有可能多个目标同时依赖于一个文件，并且其生成的命令大体类似。因此可以将其合并起来。

例如：

```
bigoutput littleoutput : text.g
	generate text.g -$(subst output,,$@) > $@
```

上述规则等价于：

```
bigoutput : text.g
	generate text.g -big > bigoutput
littleoutput : text.g
	generate text.g -little > littleoutput
```

其中，-$(subst output,,$@)中的“$”表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数。

关于 Makefile 文件的规则有很多，本书仅简要介绍基础内容，如果读者需要进一步深入研究，请参阅相关 Makefile 书籍。

3.3 Linux C/C++程序设计

C/C++适合编写基于嵌入式 Linux 的应用程序，Linux 除操作系统接口外的函数库非常丰富，对 16 位或更长的数据类型的算法的重复编码任务能自动生成代码。它还能对硬件进行特殊性的直观处理，一个串行 Flash 闪存设备的读或写能用 C 语言表达为一个简单的赋值语句，尽管存储操作需要一些编码。由于 C/C++的资料较多，读者也比较了解，这里仅对 Linux 下的 C/C++设计进行简单的概括。

3.3.1 C/C++程序结构

每个 C/C++程序通常分为两个文件。一个文件用于保存程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现（implementation），称为定义（definition）文件。

C/C++程序的头文件以“.h”为后缀，C 程序的定义文件以“.c”为后缀，C++程序的定义文件通常以“.cpp”为后缀（也有一些系统以“.cc”或“.cxx”为后缀）。其中定义文件有 3 部分内容。

（1）定义文件开头处的版权和版本声明。

（2）对一些头文件的引用。

（3）程序的实现体（包括数据和代码）。

1．头文件

头文件由 3 部分内容组成。

（1）头文件开头处的版权和版本声明；头文件和定义文件的开头有时还有版权和版本的声明，如版权信息、当前版本号、作者/修改者、完成日期等，但一般不需要。

（2）预处理块（如文件包含命令 include）；

（3）函数和类结构声明等。

在 C++语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。建议将成员函数的定义与声明分开，不论该函数体有多么小。

早期的编程语言如 Basic、Fortran 没有头文件的概念，C/C++语言的头文件的作用主要有以下两点。

（1）通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口是怎样实现的。编译器会从库中提取相应的代码。

（2）能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

2．目录

如果一个软件的头文件数目比较多（如超过 10 个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如，可将头文件保存于 include 目录，将定义文件保存于 source 目录（可以是多级目录）。如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

3．程序

C/C++程序由标识符、关键字、运算符、分隔符、常量、注释符组成，它们又由字母、数字和空白符（空格符、制表符、换行符）构成。

（1）变量名、函数名、标号等统称为标识符。

标识符最好采用英文单词或其组合，便于记忆和阅读。标识符的长度应当符合“min-length && max-information”原则，而且命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如，Windows 应用程序的标识符通常采用“大小写”混排的方式，如 AddChild。而 UNIX、Linux 应用程序的标识符通常采用“小写加下画线”的方式，如 add_child。别把这两类风格混在一起用。

（2）C/C++的关键字分为以下几类。

- 类型说明符：用于定义、说明变量、函数或其他数据结构的类型，如 int、double。

- 语句定义符：用于表示一个语句的功能。如 if-else 就是条件语句的语句定义符。
- 预处理命令字：用于表示一个预处理命令。如前面各例中用到的 include。

（3）C/C++中含有相当丰富的运算符，运算符与变量、函数一起组成表达式，表示各种运算功能。运算符由一个或多个字符组成，其中单目运算符优先级较高，赋值运算符优先级低；算术运算符优先级较高，关系和逻辑运算符优先级较低。

表 3-1 给出了 C/C++的运算符。

表 3-1　C 语言的运算

运算符	含　义
() [] -> . ! ++ -- (cast)	括号、成员、逻辑非、自加、自减、强制转换
++ -- * & ~ ! + - sizeof	单目运算符
* / %　+ -	算术运算符
<< >>	位运算符
< <= > >=　== !=	关系运算符号
= += -= *= /= %= <<= >>= &= |= ^=	赋值运算符
,	顺序运算符
&、^、|	位与、异或、或
*、&	指针运算符，取内容、取地址
sizeof	求字节数运算符
&&、||、!	逻辑与、或、非
?:	条件运算符

（4）分隔符有逗号和空格两种。逗号主要用在类型说明和函数参数表中，分隔各个变量。空格多用于语句各单词之间做间隔符。在关键字、标识符之间必须要有一个以上的空格符做间隔，否则将会出现语法错误，例如，把“int a;”写成“inta;”将出错。

（5）常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等。表 3-2 给出了 C/C++的转义符和格式字符。

表 3-2　C/C++的转义符和格式字符

转义符	含　义	格式字符	含　义
\n	回车换行	%d	以十进制形式输出带符号整数（正数不输出符号）
\t	横向跳到下一制表位置	%o	以八进制形式输出无符号整数（不输出前缀 O）
\v	竖向跳格	%x	以十六进制形式输出无符号整数（不输出前缀 OX）
\b	退格	%u	以十进制形式输出无符号整数
\r	回车	%f/ %e	以小数/ 指数形式输出单、双精度实数
\f	走纸换页	%g	以%f%e 中较短的输出宽度输出单、双精度实数
\n	回车换行	%c/ %s	输出单个字符/ 字符串

（6）注释符是以“/*”开头并以“*/”结尾或者以“//”（C89 以后的编译器支持，C++或 C#一贯都支持）开始的串。在“/*”和“*/”之间或“//”以后的即为注释。程序编译时，不对注释做任何处理。

3.3.2　C/C++数据类型

1．基本数据类型

C 语言的数据类型可分为基本类型、构造类型、指针类型、空类型。其中，基本数据类型是自我说明的；构造类型包括数组类型/结构类型/联合类型，构造类型其成员还是基本类型或构造类型；指针类型用来表示某个量在内存储器中的地址；空类型其类型说明符为 void，调用后并不需要向调用者返回函数值。限于篇幅，这里仅介绍基本类型。

（1）基本类型的分类及特点，如表 3-3 所示。

表 3-3　基本类型的分类及特点

类　　型	说 明 符	字　　节	数值范围
字符型	char	1	C 字符集
基本整型	int	2	-32768~32767
短整型	short	int	2
长整型	long	int	4
无符号型	unsigned	2	0~65535
无符号长整型	unsigned	long	4
单精度实型	float	4	3/4E-38~3/4E+38
双精度实型	double	8	1/7E-308~1/7E+308

基本数据类型量，按其取值是否可改变又分为常量和变量两种。在程序中，常量是可以不经说明而直接引用的，而变量则必须先说明后使用。

C 语言用#define 来定义常量（称为宏常量）。C++语言除了#define 外还可以用 const 来定义常量（称为 const 常量）。其中，auto 为自动变量；register 为寄存器变量；extern 为外部变量；static 为静态变量。

（2）数据类型转换有以下两种。

- 自动转换。在不同类型数据的混合运算中，由系统自动实现转换，由少字节类型向多字节类型转换。不同类型的量相互赋值时也由系统自动进行转换，把赋值号右边的类型转换为左边的类型。
- 强制转换。由强制转换运算符完成转换。

2．数组

如果一个变量名后面跟着一个有数字的中括号，这个声明就是数组声明。字符串也是一种数组。它们以 ASCII 的 NULL 作为数组的结束。如：

```
float mymatrix [3] [2] = {2.0 , 10.0, 20.0, 123.0, 1.0, 1.0}
char lexicon [10000] [300] ;        /* 共 10 000 个最大长度为 300 的字符数组。*/
int a[3][4];
```

上面最后一个例子创建了一个数组，但也可以把它看成是一个多维数组。注意数组的下标从 0 开始。这个数组的结构如下：

```
a[0][0] a[0][1] a[0][2] a[0][3]
a[1][0] a[1][1] a[1][2] a[1][3]
a[2][0] a[2][1] a[2][2] a[2][3]
```

3．指针

如果一个变量声明时在前面使用*号，表明这个变量是一个指针。定义指针的目的是为了通过指针去访问内存单元。例如：

```
int *pi;                          /* 指向整型数据的指针 */
int *api[3];                      /* 指向整型数据的一个三维数组指针 */
char **argv;                      /* 指向一个字符指针的指针 */
```

储存在指针中的地址所指向的数值在程序中可以由*读取，如*pi 是一个整型数据，引用了一个指针。

另一个运算符&，是取地址运算符，它将返回一个变量、数组或函数的存储地址。如下面的例子：

```
int i, *pi;                      /* int and pointer to int */
pi = &i;
```

i 和*p 在程序中可以相互交替使用，直到 pi 被改变成指向另一个变量的指针。

3.3.3 表达式/语句、函数

1．表达式与语句

C/C++表达式是由运算符连接常量、变量、函数所组成的式子，每个表达式都有一个值和类型。表达式求值按运算符的优先级和结合性所规定的顺序进行。表达式和语句都属于 C/C++的短语结构语法。

（1）复合语句。

如 a = b = c = 0 这样的表达式称为复合表达式。C 语言中的复合语句的格式为：

```
{语句；语句；……}
```

复合语句可以使得几个语句变成一个语句。允许复合表达式存在的理由是：书写简洁，可以提高编译效率。但要防止滥用复合表达式。

（2）条件语句。

C/C++主要有 3 种条件语句形式。两种是 if 语句，另一种是 switch 语句。

在 if 条件表达式中，任何非零的值表示条件为真；如果条件不满足，程序将跳过 if 后面的语句，直接执行 if 后面的语句。但是如果 if 后面有 else，则当条件不成立时，程序跳到 else 处执行。两种 if 语句包括：

```
            /****1****/
if（条件表达式）
    语句；
            /****2****/
if（条件表达式）
    语句；
else
    语句；
```

switch 是多分支选择语句，而 if 语句只有两个分支可供选择。虽然可以用嵌套的 if 语句来实现多分支选择，但那样的程序冗长难读。

switch 通常用于对几种有明确值的条件进行控制，它要求的条件值通常是整数或字符。与 switch 搭配的条件转移是 case。使用 case 后面的标值，控制程序将跳到满足条件的 case 处一直往下执行，直到语句结束或遇到 break。通常可以使用 default 把其他例外的情况包含进去。如果 switch 语句中的条件不成立，控制程序将跳到 default 处执行。switch 是可以嵌套的。

```
switch (<表达式>)
{
    case <值 1> :
      <语句>
    case <值 2> :
      <语句>
    default :
      <语句>
 }
```

（3）循环语句。

C/C++有三种形式的循环语句：

```
                        /****1****/
do
     <语句>
  while (<表达式>);
 /****2****/
while (<表达式>)
     <语句>;
/****3****/
 for (<表达式 1> ; <表达式 2> ; <表达式 3>)
     <语句>;
```

在 while 和 do 中，语句将执行到表达式的值为零时结束。在 do…while 语句中，循环体将至少被执行一次。这 3 种循环结构可以互相转化。

```
for (e1; e2; e3)
    s;
```

相当于：

```
  e1;
  while (e2)
{
     s;
     e3;
  }
```

当循环条件一直为真时，将产生死循环。

C/C++循环语句中，for 语句使用频率最高，while 语句其次，do 语句很少用。提高循环体效率的基本办法是降低循环体的复杂性。

在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨越循环层的次数。

（4）跳转语句。

跳转语句主要有 continue，break 和 return。

- continue 语句用在循环语句中，作用是结束当前一轮循环，马上开始下一轮循环。
- break 语句用在循环语句或 switch 中，作用是结束当前循环，跳到循环体外继续执行。但是使用 break 只能跳出一层循环。在要跳出多重循环时，可以使用 goto 使得程序更为简洁。
- 当一个函数执行结束后要返回一个值时，使用 return。return 可以跟一个表达式或变量。如果 return 后面没有值，将执行不返回值。

2．函数

C/C++函数由返回值、函数名、参数列表（或 void 表示没有返回值）和函数体组成，函数体的语法和其他的复合语句部分是一样的。C/C++各类函数不仅数量多，而且有的还需要硬件知识才会使用，因此要想全部掌握则需要一个较长的学习过程。这里仅做粗略的讲述。

函数接口的两个要素是参数和返回值。在 C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。C++语言中多了引用传递（pass by reference）。由于引用传递的性质像指针传递，而使用方式却像值传递，初学者常常迷惑不解，容易引起混乱。

在 C/C++中，程序从 main 开始执行。main 函数通过调用和控制其他函数进行工作。程序员可以自己写函数，或从库中调用函数。如语句 return 0；可使得 main 返回一个值给调用程序的外壳，表明程序已经成功运行。

（1）输入输出函数。

C 语言中没有提供专门的输入输出语句，所有的输入输出都是由调用标准库函数中的输入输出函数来实现的，其函数原型在头文件“stdio.h”中。

scanf 和 getchar 函数是输入函数，接收来自键盘的输入数据。scanf 是格式输入函数，可按指定的格式输入任意类型的数据；getchar 函数是字符输入函数，只能接收单个字符。

printf 和 putchar 函数是输出函数，向显示器屏幕输出数据。printf 是格式输出函数，可按指定的格式显示任意类型的数据；putchar 是字符显示函数，只能显示单个字符。

（2）字符串函数。

C/C++提供了丰富的字符串处理函数，大致可分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索几类。使用这些函数可大大减轻编程的负担。用于输入输出的字符串函数，在使用前应包含头文件“studio.h”，使用其他字符串函数则应包含头文件“string.h”。

（3）内存管理函数。

在实际的编程中，所需的内存空间往往取决于实际输入的数据，而无法预先确定。对于这种问题，用数组的办法很难解决。为了解决上述问题，C/C++提供了一些内存管理函数，这些内存管理函数可以按需要动态地分配内存空间，也可把不再使用的空间回收待用，为有效地利用内存资源提供了手段。

- 分配内存空间函数 malloc（size）表示在内存的动态存储区中分配一块长度为“size”字节的连续区域，函数的返回值为该区域的首地址。
- 分配内存空间函数 calloc（n,size）也用于分配内存空间。calloc 函数与 malloc 函数的区别仅在于一次可以分配 n 块区域。

- 释放内存空间函数 free（void*ptr）；表示释放 ptr 所指向的一块内存空间。

（4）用户自定义函数。

C/C++函数大部分是用户编写的函数，这种函数不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

编写自定义函数时需要注意：

- 避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。如果函数没有参数，则用 void 填充。
- 不要省略返回值的类型。如果函数没有返回值，那么应声明为 void 类型。
- 函数的功能要单一，不要设计多用途的函数。函数体的规模要小，尽量控制在 50 行代码之内。
- 尽量避免函数带有“记忆”功能，相同的输入应当产生相同的输出。带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必须使用。
- 不仅要检查输入参数的有效性，还要检查通过其他途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

3.3.4　C/C++设计注意事项

1．C/C++内存分配方式

一个由 C/C++编译的程序占用的内存分为以下几个部分。

（1）栈区（stack）：由编译器（Compiler）自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

（2）堆区（heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

（3）全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

（4）文字常量区：常量字符串就是放在这里。程序结束后由系统释放。

（5）程序代码区：存放函数体的二进制代码。

（1）内存分配方式

内存分配方式有下面三种。

- 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量、static 变量。
- 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- 从堆上分配，亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由用户决定，使用非常灵活。

（2）内存问题总结

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。首先需要区分栈与堆的不同。

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序。另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的 delete 语句才能正确地释放本内存空间。另外，由于找到的堆节点的大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表中。

堆和栈的区别可以用如下的比喻来看出，使用栈就像去饭馆里吃饭，去商场买衣服，好处是快捷，不用考虑太多，但是自由度小，有大小限制。使用堆就像是自己动手，丰衣足食，比较麻烦，但是比较符合自己，而且自由度大。

常见的内存错误及其对策如下。

- 内存分配未成功，却使用了它。刚编程时容易犯这种错误，因为没有意识到内存分配会不成功。常用解决办法是在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数，那么在函数的入口处用 assert(p!=NULL)进行检查。如果是用 malloc 或 new 来申请内存，应该用 if(p==NULL) 或 if(p!=NULL)进行防错处理。
- 内存分配虽然成功，但是尚未初始化就引用它。犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的默认初值全为零，导致引用初值错误（例如数组）。
- 内存的默认初值究竟是什么并没有统一的标准，尽管有些时候为零值。无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。
- 内存分配成功并且已经初始化，但操作越过了内存的边界。例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。
- 忘记了释放内存，造成内存泄漏。含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，用户看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

如果释放了内存却继续使用它，将出现下面三种情况。

➢ 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

➢ 函数的 return 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。

➢ 使用 free 或 delete 释放了内存后，没有将指针设置为 NULL，导致产生“野指针”。

2．C/C++编程常见问题

下面介绍 C/C++编程中常见的一些问题及其处理办法。

（1）内存分配

用 malloc 或 farmalloc 动态分配内存时，如

```
char *buffer;
buffer=(char *)malloc(300);
```

因为并不是在所有的情况下，都会分配成功，所以应加 if(buffer==NULL) {.....}

char 的取值范围为 0~127。

unsigned char 的取值范围为 0~255。

在 256 色的屏幕模式下，颜色索引值为 0~255。

（2）指针问题

- 指针与指针数组

```
int (*p)[4]; //定义一个指向包含 4 个整数元素的指针
int *p[4]; //定义一个指针数组，该指针数组包含 4 个指向整形变量的指针。
```

char **argv 等价于 char *argv[]。

Linux 程序中经常出现 int main(int argc, char * argv[])，其中 argc 是外部命令参数的个数，argv[]存放的是各参数的内容。

例如下面一段程序。

```
include <iostream>
#include <cstring>
using namespace std;
int main(int argc, char* argv[])
{
     char *str1 = argv[1];
     char *str2 = argv[2];
     int result;
     if ((result = strcmp(str1,str2)) == 0) cout << "equal" << endl;
     else
     {
           if (result > 0) cout << str1 << ">" << str2 << endl;
           else cout << str1 << "<" << str2 <<endl;
     }
     return 0;
}
```

经过编译后，在命令行输入./test abc def ，输出结果为 abc<edf。

abc 就是 argv[1], def 为 argv; 而 argv[0]是文件名 test.cpp。由于有两个参数，故 argc=2。

- 函数指针与指针函数

可以这么理解：函数指针是指针，是指向函数的指针；指针函数是函数，是返回一个指针的函数。

```
int (*p)(); //函数指针，也就是函数的入口地址。
int *p(); //指针函数，也就是函数返回的值是一个指针。
```

例如：

（1）int (*p)(char);

这里 p 被声明为一个函数指针，这个函数带一个 char 类型的参数，并且有一个 int 类型的返回值。

（2）常见内存拷贝

```
unsigned char *memcpy(unsigned char *dest,const unsigned char *src, int lenth);
```

这是一个函数返回值为 unsigned char 类型的指针。

3．Linux C 链接库开发

在 Linux 下有静态库（static library，.a 文件）和共享库（shared library，.so 文件）之分，就像 Win32 系统中，存在静态库（.lib）和动态库（.dll）一样。静态库和共享库都是一个 obj 文件的集合，但静态链接后，执行程序中存在自己所需 obj 的一份拷贝；而动态链接后，执行程序仅仅是包含对共享库的一个引用。共享库相当于一个由多个 obj 文件组合而成的 obj 文件，在链接后其所有代码被加载，不管需要的还是不需要的。

静态库与动态库具体区别如下：

- 静态链接后的程序比动态链接的所用存储空间大，因为执行程序中包含了库中代码拷贝；而动态链接的程序比静态链接的所用的运行空间大，因为它将不需要的代码也加载到运行空间。
- 动态链接库虽然可以类似插件给编译好了的二进制程序实现一些功能及升级，但程序发布的时候，需要随身携带那些.so 文件，而静态库是一劳永逸，编译后不需要带一堆库文件，而且不管放置到哪里都可正常运行。

（1）静态库

- 字符串转换

建立一个简单的静态库，实现一字符串大小写相互转换，需要建立下面三个文件。

➢ 头文件 convert.h

它声明相关函数原形，内容如下：

```
#ifndef CONVERT_H
#define CONVERT_H
extern char*   toLowerString(char*  sSource);
extern char*   toUpperString(char*  sSource);
#endif
```

➢ toLower.c

函数把字符串转换成小写字符串，内容如下：

```
 char*   toLowerString(char* src)
 {
     assert(src   !=   NULL);
char*   des   =  src;
 while   (*des)
 {
    if (isupper(*des))
 {
 *des  =   tolower(*des);
 }
 des++;
 }
    return   src;
 }
```

➢ toUpper.c

函数把字符串转换成大写的写字符串，内容如下：

```
char*   toUpperString(char*   src)
  {
    assert(src   !=   NULL);
char*   des   =   src;
  while   (*des)
  {
     if   (islower(*des))
  {
  *des   =   toupper(*des);
  }
  des++;
  }
  return   src;
  }
```

- 生成静态库

利用 GCC 生成对应目标文件。

```
gcc –c toLower.c toUpper.c
```

gcc 首先会对文件进行编译，生成 toLower.o 和 toUpper.o 两个目标文件（相当于 Windows 下的 obj 文件）。

然后用 ar 创建一个名字为 libconvert.a 的库文件，并把 toLower.o 和 toUpper.o 的内容插入到对应的库文件中。相关命令如下。

```
ar –rcs libconvert.a toLower.o 和 toUpper.o
```

命令执行成功以后，对应的静态库 libconvert.a 已经成功生成。

- 测试

```
  #include   <stdio.h>
  #include   <ctype.h>
  #include   <assert.h>
  #include  <convert.h>
  int main()
  {
    char a[]   = "abcd";
    char b[]  = "ABCD";
printf("%s\n", toUpperString(a,   0));
printf("%s\n", toLowerString(b,   1));
return   0;
  }
```

静态库的调用比较简单，跟标准库函数调用一样，如果把头文件和库文件复制到 gcc 默认的头文件目录/usr/include 和/usr/lib 里面，编译程序就跟标准库函数调用一模一样了。

```
gcc -o test main.c -I 头文件的路径 -L 库文件的路进 -lconvert
gcc main.c -o test –static –L. -lconvert
```

（2）动态链接库

Linux 动态链接库为隐式调用和显式调用两种调用方法，下面分别介绍。

- 隐式调用

隐式调用的使用方法和静态库的调用差不多，在这种调用方式中，需要维护动态链接库的配置文件/etc/ld.so.conf 来让动态链接库为系统所使用，通常将动态链接库所在目录名追加到动态链接库配置文件中。否则在执行相关的可执行文件的时候就会出现载入动态链接库失败的现象。在编译所引用的动态库时，可以在 gcc 采用 –l 或-L 选项或直接引用所需的动态链接库方式进行编译。在 Linux 里面，可以采用 ldd 命令来检查程序依赖共享库。

按照正常编程，include 方式只是在编译程序的时候加上 –Lxxx –lxxx 的方式来利用动态链接库，这种方式利用了动态链接库升级方便的特点。在这种方式下，用 ldd 程序名能够查看到程序所使用的动态链接库。

下面生成动态链接库：

```
gcc  -fPIC -shared -o libconvert.so toLower.o toUpper.o
```

-fPIC 使输出的对象模块是按照可重定位地址方式生成的。

-shared 指定把对应的源文件生成对应的动态链接库文件 libconvert.so 文件。

对应的链接库已经生成，下面看一下如何使用对应的链接库。

```
gcc –g main.c -o test –L. -lconvert
```

为了执行程序，告诉动态链接器 ld.so 如何找到这个库。

```
LD_LIBRARY_PATH=$(PWD)
```

- 显式调用

显式调用是通过调用 dlfcn 系列函数所制作的程序，如 dlopen 函数。在这种方式下，程序可自由指定所要加载的动态链接库，适用于加载 Plug-in 式的动态链接库。在这种方式下，用 ldd 程序名查看不到程序所使用的动态链接库。

```
#include<stdio.h>
#include<dlfcn.h>
int main(int argc, char* argv[])
{
   char a[]   = "abcd";
   char b[]  = "ABCD";
   //define function pointor
   int (*pUpper)(char* pStr);     //声明对应函数的函数指针
   int (*pLower)(char* pStr, );
   void *pdlHandle;
   char *perr;
   pdlHandle = dlopen("./libconvert.so", RTLD_LAZY);  //加载链接库
                                                          /libconvert.so

   if(!pdlHandle)
   {
      printf("Failed load library\n");
   }
   perr = dlerror();
   if(perr != NULL)
   {
```

```
        printf("%s\n", perr);
        return 0;
    }
    //get function from lib
    pUpper = dlsym(pdlHandle, "toUpperString"); //获取函数的地址
    perr = dlerror();
    if(perr != NULL)
    {
        printf("%s\n", perr);
        return 0;
    }
    pLower = dlsym(pdlHandle, "toLowerString");
    perr = dlerror();
    if(perr != NULL)
    {
        printf("%s\n", perr);
        return 0;
    }
  printf("%s\n", pUpper(a);
  printf("%s\n", pLower(b);
  dlclose(pdlHandle);
  return  0;
 }
```

```
gcc -o test -ldl main.c
```

用 gcc 编译对应的源文件生成可执行文件，-ldl 选项表示生成的对象模块需要使用共享库。执行对应的文件同样可以得到正确的结果。

相关函数的说明如下：

- dlopen()

第一个参数，指定共享库的名称，将会在下面位置查找指定的共享库。

- dlsym()

调用 dlsym 时，利用 dlopen()返回的共享库的 phandle 以及函数名称作为参数，返回要加载函数的入口地址。

- dlerror()

该函数用于检查调用共享库的相关函数出现的错误。

3.4　Linux 汇编程序设计

汇编语言的优点是速度快，可以直接对硬件进行操作，虽然 Linux 是一个用 C 语言开发的操作系统，以下介绍 Linux 汇编语言的语法格式和开发工具。

作为最基本的编程语言之一，汇编语言虽然应用的范围不算很广，但重要性却毋庸置疑，因为它能够完成许多其他语言所无法完成的功能。在 Linux 操作系统中，虽然绝大部分代码是用 C 语言编写的，但仍然不可避免地在某些关键地方使用了汇编代码。

在大多数情况下，Linux 程序员不需要使用汇编语言，因为即便是硬件驱动这样的底层程序，在 Linux 操作系统中也可以完全用 C 语言来实现，加之 GCC 目前已经能够对最终生成的代码进行很好的优化。但是，在移植 Linux 到某一特定的嵌入式硬件环境下时，则需要汇编程序。汇编语言直接同计算机的底层进行交互，它具有以下一些优点。

- 能够直接访问与硬件相关的存储器或 I/O 端口；
- 能够不受编译器的限制，对生成的二进制代码进行完全的控制；
- 能够对关键代码进行更准确的控制，避免因线程共同访问或者硬件设备共享引起的死锁；
- 能够根据特定的应用对代码做最佳的优化，提高运行速度；
- 能够最大限度地发挥硬件的功能。

当然，由于汇编语言作为一种层次非常低的语言，仅仅高于直接手工编写二进制的机器指令码，因此不可避免地存在一些缺点。

- 编写的代码非常难懂，不好维护；
- 很容易产生 bug，难以调试；
- 只能针对特定的体系结构和处理器进行优化；
- 开发效率很低，时间长且单调。

Linux 下用汇编语言编写的代码具有两种不同的形式。

第一种是完全的汇编代码，即整个程序全部用汇编语言编写。尽管是完全的汇编代码，Linux 平台下的汇编工具也吸收了 C 语言的长处，使得程序员可以使用#include、#ifdef 等预处理指令，并能够通过宏定义来简化代码。

第二种是内嵌的汇编代码，即可以嵌入到 C 语言程序中的汇编代码片段。虽然 ANSI 的 C 标准中没有关于内嵌汇编代码的相应规定，但各种实际使用的 C 编译器都做了这方面的扩充。

3.4.1 Linux 汇编语法格式

在 DOS/Windows 下的汇编语言基本上都是 Intel 风格的。但在 UNIX 和 Linux 系统中，更多的是采用 AT&T 格式，两者在语法和格式上有着很大的不同。

在 AT&T 汇编格式中，寄存器名要加上“%”作为前缀；而在 Intel 汇编格式中，寄存器名不需要加前缀。例如：

```
AT&T 格式 ：pushl %eax
Intel 格式：push eax
```

在 AT&T 汇编格式中，用'$'前缀表示一个立即操作数；而在 Intel 汇编格式中，立即数的表示不用带任何前缀。例如：

```
AT&T 格式：pushl $1
Intel 格式：push 1
```

AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；而在 AT&T 汇编格式中，目标操作数在源操作数的右边。例如：

```
AT&T 格式：addl $1, %eax
Intel 格式：add eax, 1
```

在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀'b'、'w'、'l'分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32 比特）；而在 Intel 汇编格式中，操作数的字长是用'byte ptr'和'word ptr'等前缀来表示的。例如：

```
AT&T 格式：movb val, %al
Intel 格式：mov al, byte ptr val
```

在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上'*'作为前缀，而在 Intel 格式中则不需要。远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为'ljump'和'lcall'，而在 Intel 汇编格式中则为'jmp far'和'call far'，即：

```
AT&T 格式：   ljump $section, $offset ，       lcall $section, $offset
Intel 格式：  jmp far section:offset ，       call far section:offset
```

与之相应的远程返回指令则为：

```
AT&T 格式：lret $stack_adjust
Intel 格式：ret far stack_adjust
```

在 AT&T 汇编格式中，内存操作数的寻址方式为：

```
section:disp(base, index, scale)
```

而在 Intel 汇编格式中，内存操作数的寻址方式为：

```
section:[base + index*scale + disp]
```

由于 Linux 工作在保护模式下，用的是 32 位线性地址，所以在计算地址时不用考虑段基址和偏移量，而是采用如下的地址计算方法：

```
disp + base + index * scale
```

下面是一些内存操作数的例子。

AT&T 格式：

```
movl -4(%ebp), %eax
movl array(, %eax, 4), %eax
movw array(%ebx, %eax, 4), %cx
movb $4, %fs:(%eax)
```

Intel 格式：

```
mov eax, [ebp - 4]
 mov eax, [eax*4 + array]
mov cx, [ebx + 4*eax + array]
 mov fs:eax, 4
```

3.4.2　汇编程序实例

以下介绍如何在 Linux 系统下编写第一个汇编程序。Linux 是一个运行在保护模式下的 32 位操作系统，采用 flat memory 模式，目前最常用到的是 ELF 格式的二进制代码。一个 ELF 格式的可执行程序通常划分为如下几个部分：.text、.data 和.bss。

- text 是只读的代码区。
- data 是可读可写的数据区。
- bss 则是可读可写且没有初始化的数据区。

代码区和数据区在 ELF 中统称为 section，根据实际需要可以使用其他标准的 section，也可以添加自定义的 section，一个 ELF 可执行程序至少应该有一个.text 部分。下面是一个 AT&T 格式的汇编语言程序。

```
# hello.s (AT&T 格式)
.data                                          # 数据段声明
       msg : .string "Hello, world!just test!\n"    # 要输出的字符串
       len = . - msg                           # 字串长度

.text                                          # 代码段声明
.global _start                                 # 指定入口函数

_start:                                        # 在屏幕上显示一个字符串
       movl $len, %edx                         # 参数三：字符串长度
       movl $msg, %ecx                         # 参数二：要显示的字符串
       movl $1, %ebx                           # 参数一：文件描述符(stdout)
       movl $4, %eax                           # 系统调用号(sys_write)
       int  $0x80                              # 调用内核功能

                                               # 退出程序
       movl $0,%ebx                            # 参数一：退出代码
       movl $1,%eax                            # 系统调用号(sys_exit)
       int  $0x80                              # 调用内核功能
```

这是一个最简单的 Linux 汇编程序，其调用 Linux 内核提供的 sys_write 来显示一个字符串，然后再调用 sys_exit 退出程序。在 Linux 内核源文件 include/asm-i386/unistd.h 中，可以找到所有系统调用的定义。此段码如果采用 Intel 格式则为：

```
; hello.asm (Intel 格式)
section .data                                  ; 数据段声明
       msg db "Hello, world!", 0xA             ; 要输出的字符串
       len equ $ - msg                         ; 字串长度

section .text                                  ; 代码段声明
global _start                                  ; 指定入口函数

_start:                                        ; 在屏幕上显示一个字符串
       mov edx, len                            ; 参数三：字符串长度
       mov ecx, msg                            ; 参数二：要显示的字符串
       mov ebx, 1                              ; 参数一：文件描述符(stdout)
       mov eax, 4                              ; 系统调用号(sys_write)
       int 0x80                                ; 调用内核功能
                                               ; 退出程序
       mov ebx, 0                              ; 参数一：退出代码
       mov eax, 1                              ; 系统调用号(sys_exit)
       int 0x80                                ; 调用内核功能
```

3.5 Linux Shell 语言编程

在使用 Linux 操作系统时，每个用户登录系统后，系统总会出现不同的命令提示符，如#、$或者~等，然后用户输入的任何命令（正确的命令），系统都可以根据命令的要求执行，直到用户注销，在这期间，用户的所有命令都会经过解释才能被执行，而完成这一功能的机制就是 Shell，如图 3-1 所示是 Shell 在 Linux 操作系统的位置。Shell 具有以下主要特点。

（1）对已有命令进行适当组合，构成新的命令，而组合方式很简单。

（2）它们提供了文件名扩展字符，使得用单一的字符串可以匹配多个文件名，省去键入一长串文件名的麻烦。

（3）可以直接使用 Shell 的内置命令，而不需要创建新的进程。

（4）Shell 允许灵活地使用数据流，提供了通配符、输入/输出重定向、管道线等机制，方便了模式匹配、I/O 处理和数据传输。

（5）结构化的程序模块，提供了顺序流程控制、条件控制、循环控制等。

（6）Shell 提供了在后台执行命令的能力。

（7）Shell 提供了可配置的环境，允许用户创建和修改命令、命令提示符和其他的系统行为。

（8）Shell 提供了一个高级的命令语言，允许用户能创建从简单到复杂的程序。

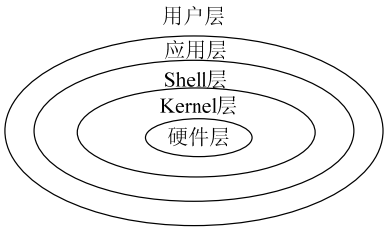


图 3-1　Shell 在 Linux 操作系统中的位置

在 Linux 和 UNIX 系统里可以使用多种不同的 Shell。最常用的几种是 Bourne Shell(sh)，C Shell(csh) 和 Korn Shell(ksh)。这 3 种 Shell 都有它们的优点和缺点。

（1）Bourne SHELL 的作者是 Steven Bourne，它是 UNIX 最初使用的 Shell 并且在每种 UNIX 上都可以使用。Bourne Shell 在 Shell 编程方面相当优秀，但在处理与用户的交互方面不如其他几种 Shell。

（2）C Shell 由 Bill Joy 所写，它更多地考虑了用户界面的友好性，它支持如命令补齐（command-line completion）等一些 Bourne Shell 所不支持的特性。普遍认为 C Shell 的编程接口不如 Bourne Shell，但 C Shell 被很多 C 程序员使用是因为 C Shell 的语法和 C 语言的语法很相似，这也是 C Shell 名称的由来。

（3）Korn Shell(ksh)由 Dave Korn 所写。它集合了 C Shell 和 Bourne Shell 的优点，并且和 Bourne Shell 完全兼容。

除了这些 Shell 以外，许多其他的 Shell 程序吸收了这些原来的 Shell 程序的优点而成为新的 Shell。在 Linux 上常见的有 tcsh（csh 的扩展），Bourne Again Shell（bash、sh 的扩展）和 Public Domain Korn Shell（pdksh、ksh 的扩展）。bash 是大多数 Linux 系统的默认 Shell。Bourne Again Shell（bash），正如它的名字所暗示的，是 Bourne Shell 的扩展。bash 与 Bourne Shell 完全向后兼容，并且在 Bourne Shell 的基础上增加和增强了很多特性。bash 也包含了很多 C 和 Korn Shell 里的优点。bash 有很灵活和强大的编程接口，同时又有很友好的用户界面。

3.5.1　Shell 环境变量及配置文件

因为 Linux 支持多种 Shell，故读者可以根据个人习惯选择不同的 Shell，要查看目录

所使用的 Shell 或者系统默认 Shell，只需要运行“echo”命令来查询 Shell 环境变量即可，用法如下：

```
[root@yangzongde root]# echo $SHELL
/bin/bash                    //目前使用的是 Bash
[root@yangzongde root]# echo ${SHELL}
/bin/bash
```

如果要改变当前使用的 Shell，只需要运行该 Shell 程序名即可进行切换。

```
[root@yangzongde root]# sh
sh-2.05b# bash
[root@yangzongde root]# ash
# bsh
# tcsh
[root@yangzongde ~]#
```

Shell 环境变量（Environment Variables）是 Shell 用来保存系统信息的变量，这些变量可供 Shell 中运行的程序使用。不同的 Shell 有不同的环境变量以及环境变量值，运行“set”命令可以显示当前环境变量名及变量值。

```
[root@yangzongde ~]# set
COLORS  /etc/DIR_COLORS
_

addsuffix
argv    ()
autologout      60
cwd     /root
dirstack        /root
dspmbyte        euc
echo_style      both
edit
file    /root/.i18n
gid     0
group   root
history 100
home    /root
killring        30
owd
path    (/usr/local/sbin /usr/sbin /sbin /usr/local/sbin /usr/local/bin
/sbin /bin /usr/sbin /usr/bin /usr/X11R6/bin /root/bin /usr/local/sbin /usr/
local/bin /root/bin)
prompt  [%n@%m %c]#
prompt2 %R?
prompt3 CORRECT>%R (y|n|e|a)?
shell   /bin/tcsh
shlvl   4
sourced 1
status  0
tcsh    6.12.00
term    vt100
tty     pts/0
uid     0
user    root
version tcsh 6.12.00 (Astron) 2002-07-23 (i386-intel-linux) options 8b,nls,
```

```
dl,al,kan,rh,color,dspm,filec
```

在 Linux 操作系统中，有以下几个主要的与 Shell 有关的配置文件。

（1）/etc/profile 文件：这是系统最重要的 Shell 配置文件，也是用户登录系统最先检查的文件，系统的环境变量多定义在此文件中。主要包括“PATH、USER、LGNAME、MAIL、HOSTNAME、HISTSIZE 以及 INPUTRC”。

（2）~/.bash_profile 文件：每个用户的 BASH 环境配置文件，存在于用户的主目录中，当系统运行/etc/profile 后，将读取此文件的内容，此文件定义了“USERNAME\BASH ENV 和 PATH”等环境变量，此处的 PATH 包括了用户自己定义的路径，以及用户的“bin”路径。

（3）~/.bashrc 文件：前两个文件仅在系统登录时读取，此文件将在每次运行 bash 时读取，此文件主要定义的是一些终端设置，以及 Shell 提示符等功能，而不定义环境变量等内容。

（4）~/.bash_login 文件：如果~/.bash_profile 文件不存在，系统会读取这个文件的内容。

（5）~/.profile 文件：如果~/.bash_profile 和~/.bash_login 文件都不存在时将读取此文件内容。

（6）~/.bash_history 文件：记录了用户使用的历史命令。

3.5.2　Shell 编程实例

Shell 程序是一个包含 UNIX 命令的普通文件，这个文件的许可权限至少应该为可读和可执行。在 Shell 提示符下键入文件名就可执行 Shell 程序。Shell 程序可以通过环境变量、命令行参数、用户的输入 3 种方式接收数据。

Shell 是一个命令解释器，它会解释并执行命令提示符下输入的命令。如果用户想要多次执行一组命令，Shell 提供了一种功能将这组命令存放在一个文件中，然后可以像 Linux 系统提供的其他程序一样执行这个文件，这个命令文件就叫做 Shell 程序或者 Shell 脚本程序。

当运行这个文件，它会如同在命令行输入这些命令一样执行这些命令，为了让 Shell 能读取并且执行 Shell 程序，Shell 脚本的文件权限必须被设置为可读和可执行。程序员可以写出非常复杂的 Shell 脚本，因为 Shell 脚本支持变量、命令行参数、交互式输入、tests（判断））、branches（分支）和 loops（循环）等复杂的结构。以下是一个简单的 Shell 程序 myshell 运行过程（此程序源代码见光盘 ch01 文件夹）。

```
[root@yangzongde ~]# cat myshell
#!/bin/bsh                                  #这句不是注释，标识要使用的 SHELL
#this is a example of shell                 #注释
echo $SHELL                                 #显示当前 SHELL 类型
echo "Hello World"                          #显示 Hello World
ls –l                                       #列出当前上当文件详情
[root@yangzongde ~]# chmod u+x myshell      #修改文件权限为可执行
[root@yangzongde ~]# ls -l myshell          #查看文件权限
-rwxr--r--    1 root    root         76  4 月  1 17:14 myshell
[root@yangzongde ~]# ./myshell              #运行此 SHELL 文件
/bin/bash                                   #echo $SHELL 运行结果
Hello World                                 #echo "Hello World"运行结果
```

```
总用量 36                                       # ls –l 运行结果
-rw-r--r--    1 root    root        1276  3 月 22 05:44 anaconda-ks.cfg
-rw-r--r--    1 root    root       20771  3 月 22 05:44 install.log
-rw-r--r--    1 root    root        3956  3 月 22 05:44 install.log.syslog
-rwxr--r--    1 root    root          76  4 月 1 17:14 myshell
```

一个 Shell 脚本程序编写的步骤如下。

（1）用编辑器（如 VI）编辑包含所有操作的.sh 文件；

（2）修改文件的权限为可读可执行；

（3）运行当前 Shell 程序。

Shell 编程能够使系统管理员方便地执行系统相关操作，Shell 编程是 Linux 环境下一个非常重要的编程内容。关于详细的 Shell 编程请读者参阅梁普选等编著的《Linux 编程命令详解》（电子工业出版社出版）一书。

3.6 Linux Perl 语言编程

Perl（Practical Extractionand Report Language）是一种解释性的语言，专门为搜索纯文本文件而做了优化。它也可以十分方便地完成很多系统管理任务。它集成了 C、sed、awk 和 sh 语言的优点，可以运行于 Linux、UNIX、MVS、VMS、MS-DOS、Macintosh、OS/2、Amiga 以及其他的一些操作系统上。特别是近年来，随着 Internet 的普及，Perl 也越来越多地用于 WorldWideWeb 上 CGI 等的编程，逐渐成为系统、数据库和用户之间的桥梁。

有两种程序员喜欢用 Perl，系统程序员可以用 Perl 结合系统命令一起处理数据和过程，并且可以使用 Perl 的格式匹配函数进行系统信息的搜寻和总结；还有一些开发 UNIX Web 服务器 CGI 程序的程序员发现 Perl 比 C 语言易学易用，而且更容易处理数据库和数据搜索。

3.6.1 Perl 基本程序

Perl 的创建人 Larry Wall 在 1994 年 10 月发表了 Perl 的第 5 版本，目前发展到 V5.8。Perl 5 增加了面向对象的能力，提供了更多的数据结构、系统和数据库之间的新的标准接口，以及其他的一些功能。

例如，希望更换大量文件中的一些相同内容，可以使用下面的一条命令。

```
perl -e 's/gopher/World Wide Web/gi' -p -i.bak *.html
```

以下是一个基本的 perl 程序。

```
[root@yangzongde perl]# cat perl        显示程序内容
#!/usr/bin/perl
#
#Progarm to do the Example
#
print'hello World!';    #printf a message
[root@yangzongde perl]# chmod u+x perl 加上执行权限
[root@yangzongde perl]# ls
perl
[root@yangzongde perl]# ./perl          运行程序
hello World![root@yangzongde perl]#     运行结果
```

每个 perl 程序都以# !/usr/bin/perl 开始，这样系统的外壳知道应该使用 perl 运行该程序。

Perl 表达式必须以分号结尾，就如同 C 语言一样。此语句为显示语句，只是简单地显示出 hello World 字符串。

3.6.2　Perl 变量

Perl 中有 3 种变量：标量，数组（列表）和相关数组。

（1）标量。

Perl 中最基本的变量类型是标量。标量既可以是数字，也可以是字符串，而且两者是可以互换的。具体是数字还是字符串，可以由上下文决定。标量变量的语法为$variable_name。例如：

```
$priority = 9;
```

把 9 赋予标量变量$priority，也可以将字符串赋予该变量。

```
$priority = 'high';
```

注意在 Perl 中，由于变量名的大小写是敏感的，所以$a 和$A 是不同的变量。

Perl 中有 3 种类型的引用。

双引号（""）括起来的字符串中的任何标量和特殊意义的字符都将被 Perl 解释。如果不想让 Perl 解释字符串中的任何标量和特殊意义的字符，应该将字符串用单括号括起来。这时，Perl 不解释其中的任何字符，除了\\和\'。最后，可以用（'）将命令括起来，这样，其中的命令可以正常运行，并能得到命令的返回值。请看下面的例子。

```
[root@yangzongde perl]# cat perl2          程序内容
#!/usr/bin/perl                                                    #1
$folks="100";                                                      #2
print "\$folks = $folks \n";                                       #3
print '\$folks = $folks \n';                                       #4
print "\n\n BEEP! \a \LSOME BLANK \ELINES HERE \n\n";              #5
$date = `date +%D`;                                                #6
print "Today is [$date] \n";                                       #7
chop $date;                                                        #8
print "Date after chopping off carriage return: [".$date."]\n";#9
[root@yangzongde perl]# chmod u+x perl2    程序执行权限
[root@yangzongde perl]# ls
perl  perl2
[root@yangzongde perl]# ./perl2            运行程序
$folks = 100
\$folks = $folks \n

 BEEP!  some blank LINES HERE

Today is [05/07/06
]
Date after chopping off carriage return: [05/07/06]
[root@yangzongde perl]#
```

在此程序中，第 3 行显示$folks 的值。$之前必须使用换码符\，以便 Perl 显示字符串$folks 而不是$folks 的值 100。

第 4 行使用的是单引号，结果 Perl 不解释其中的任何内容，只是原封不动地将字符串

显示出来。

第 6 行使用的是（'），则 date +%D 命令的执行结果存储在标量$date 中。

（2）数组。

数组也叫做列表，是由一系列的标量组成的。数组变量以@开头。请看以下的赋值语句。

```
@food = ("apples", "pears", "eels");
@music = ("whistle", "flute");
```

数组的下标从 0 开始，可以使用方括号引用数组的下标。

```
$food[2]
```

返回 eels。注意@已经变成了$，因为 eels 是一个标量。

在 Perl 中，数组有多种赋值方法，例如：

```
@moremusic = ("organ", @music, "harp");
@moremusic = ("organ", "whistle", "flute", "harp");
```

还有一种方法可以将新的元素增加到数组中。

```
push(@food, "eggs");
```

把 eggs 增加到数组@food 的末端。要往数组中增加多个元素，可以使用下面的语句。

```
push(@food, "eggs", "lard");
push(@food, ("eggs", "lard"));
push(@food, @morefood);
```

push 返回数组的新长度。

pop 用来将数组的最后一个元素删除，并返回该元素。例如：

```
@food = ("apples", "pears", "eels");
$grub = pop(@food);#此时$grub = "eels"
```

请看下面的例子。

```
[root@yangzongde perl]# cat perl3
1 #!/usr/bin/perl
2 #
3 # An example to show how arrays work in Perl
4 #
5 @amounts = (10, 24, 39);
6 @parts = ('computer', 'rat', "kbd");
7
8 $a = 1; $b = 2; $c = '3';
9 @count = ($a, $b, $c);
10
11 @empty = ();
12
13 @spare = @parts;
14
15 print '@amounts = ';
16 print "@amounts \n";
17
18 print '@parts = ';
```

```
19 print "@parts \n";
20
21 print '@count = ';
22 print "@count \n";
23
24 print '@empty = ';
25 print "@empty \n";
26
27 print '@spare = ';
28 print "@spare \n";
29
30
31 #
32 # Accessing individual items in an array
33 #
34 print '$amounts[0] = ';
35 print "$amounts[0] \n";
36 print '$amounts[1] = ';
37 print "$amounts[1] \n";
38 print '$amounts[2] = ';
39 print "$amounts[2] \n";
40 print '$amounts[3] = ';
41  print "$amounts[3] \n";
42
43 print "Items in \@amounts = $#amounts \n";
44 $size = @amounts; print "Size of Amount = $size\n";
45 print "Item 0 in \@amounts = $amounts[$[]\n";
[root@yangzongde perl]# chmod u+x perl3
[root@yangzongde perl]# ls
perl  perl2  perl3
[root@yangzongde perl]# ./perl3
@amounts = 10 24 39
@parts = computer rat kbd
@count = 1 2 3
@empty =
@spare = computer rat kbd
$amounts[0] = 10
$amounts[1] = 24
$amounts[2] = 39
$amounts[3] =
Items in @amounts = 2
Size of Amount = 3
Item 0 in @amounts = 10
[root@yangzongde perl]#
```

在第 5 行，3 个整数值赋给了数组@amounts。第 6 行，3 个字符串赋给了数组@parts。第 8 行，字符串和数字分别赋给了 3 个变量，然后将 3 个变量赋给了数组@count。第 11 行创建了一个空数组。第 13 行将数组@spare 赋给了数组@parts。第 15 到第 28 行输出了显示的前 5 行。第 34 到第 41 行分别存取数组@amounts 的每个元素。注意$amount[3]不存在，所以显示一个空项。第 43 行中使用$#array 方式显示一个数组的最后一个下标，所以数组@amounts 的大小是（$#amounts+1）。第 44 行中将一个数组赋给了一个标量，则将数组的大小赋给了标量。第 45 行使用了一个 Perl 中的特殊变量$ [，用来表示一个数组的起始位置（默认为 0）。

（3）数组。

一般的数组允许通过数字下标存取其中的元素。例如，数组 food 的第一个元素是$food[0]，第二个元素是$food[1]，以此类推。但 Perl 允许创建相关数组，这样可以通过字符串存取数组。其实，一个相关数组中每个下标索引对应两个条目，第一个条目叫做关键字，第二个条目叫做数值。这样就可以通过关键字来读取数值。相关数组名以百分号（%）开头，通过花括号（{}）引用条目。例如：

```
%ages = ("Michael Caine", 39,
"Dirty Den", 34,
"Angie", 27,
"Willy", "21 in dog years",
"The Queen Mother", 108);
```

可以通过下面的方法读取数组的值。

```
$ages{"Michael Caine"};# Returns 39
$ages{"Dirty Den"};# Returns 34
$ages{"Angie"};# Returns 27
$ages{"Willy"};# Returns "21 in dog years"
$ages{"The Queen Mother"};# Returns 108
```

3.6.3 文件句柄和文件操作

可以通过下面的程序了解一下文件句柄的基本用法。此程序的执行结果和 UNIX 系统的 cat 命令一样。

```
#!/usr/local/bin/perl
#
# Program to open the password file, read it in,
# print it, and close it again.
$file = '/etc/passwd';                  # Name the file
open(INFO, $file);                      # Open the file
@lines = <INFO>;                        # Read it into an array
close(INFO);                            # Close the file
print @lines;                           # Print the array
```

open 函数打开一个文件，其中第一个参数是文件句柄（filehandle）。文件句柄用来标识一个文件。第二个参数指向该文件的文件名。close 函数关闭该文件。

如果 open 函数以写入和追加的方式打开文件，只需分别在文件名之前加上>和> >。

```
open(INFO, $file);# Open for input
open(INFO, ">$file");# Open for output
open(INFO, ">>$file");# Open for appending
open(INFO, "<$file");# Also open for input
```

另外，如果希望输出内容到一个已经打开的文件中，可以使用带有额外参数的 print 语句。例如：

```
print INFO "This line goes to the file.\n";
```

最后，可以使用如下的语句打开标准输入（通常为键盘）和标准输出（通常为显示器）。

```
open(INFO, '-');# Open standard input
open(INFO, '>-');# Open standard output
```

一个 Perl 程序在它一启动时就已经建立 3 个标准文件：STDIN（标准输入设备）、STDOUT（标准输出设备）和 STDERR（标准错误信息输出设备）。

如果想要从一个已经打开的文件句柄中读取信息，可以使用< > 运算符。

使用 read 和 write 函数可以读写一个二进制的文件，其用法如下所示。

```
read(HANDLE，$buffer，$length[，$offset]);
```

此命令可以把文件句柄是 HANDLE 的文件从文件开始位移$ offset 处，共$length 字节，读到$buffer 中。其中$ offset 是可选项，如果省略$offset，则 read( )从当前位置的前$length 个字节读取到当前位置。可以使用下面的命令查看是否到了文件末尾。

```
eof(HANDLE);
```

如果返回一个非零值，则说明已经到达文件的末尾。

打开文件时可能出错，所以可以使用 die( )显示出错信息。下面打开一个叫做“test.data”的文件。

```
open(TESTFILE，"test.data") || die "\n $0 Cannot open $! \n";
```

3.6.4　循环结构

（1）foreach 循环。

在 Perl 中，可以使用 foreach 循环来读取数组或其他类似列表结构中的每一行。请看下面的例子。

```
foreach $morsel (@food)# Visit each item in turn
# and call it $morsel
{
print "$morsel\n";# Print the item
print "Yum yum\n";# That was nice
}
```

每次要执行的命令用花括号括出。第一次执行时$morsel 被赋予数组@food 的第一个元素的值，第二次执行时$morsel 被赋予数组@food 的第二个元素的值，以此类推直到数组的最后一个元素。

（2）判断运算。

在 Perl 中任何非零的数字和非空的字符串都被认为是真。零、全为零的字符串和空字符串都为假。

下面是一些判断运算符。

```
$a == $b 如果$a 和$ b 相等，则返回真
$a != $b 如果$ a 和$ b 不相等，则返回真
$a eq $b 如果字符串$ a 和字符串$ b 相同，则返回真
$a ne $b 如果字符串$ a 和字符串$ b 不相同，则返回真
```

可以使用逻辑运算符：

```
($a && $b) $ a 与$ b
($a || $b)$a 或$ b
! ( $ a ) 非$ a
```

（3）for 循环。

Perl 中的 for 结构和 C 语言中的 for 结构基本一样。

```
for (initialise; test; inc)
{
first_action;
second_action;
etc
}
```

下面是一个 for 循环的例子，用来显示从 0 到 9 的数字。

```
for ($i = 0; $i < 10; ++$i)# Start with $i = 1
# Do it while $i < 10
# Increment $i before repeating
{
print "$i\n";
}
```

（4）while 和 until 循环。

下面是一个 while 和 until 循环的例子。它从键盘读取输入直到得到正确的口令为止。

```
#!/usr/local/bin/perl
print "Password? ";# Ask for input
$a = <STDIN>;# Get input
chop $a;# Remove the newline at end
while ($a ne "fred")# While input is wrong...
{
print "sorry. Again? ";# Ask again
$a = <STDIN>;# Get input again
chop $a;# Chop off newline again
}
```

当输入和口令不相符时，执行 while 循环。也可以在执行体的末尾处使用 while 和 until，这时需要用 do 语句。

```
#!/usr/local/bin/perl
do
{
"Password? ";                       # Ask for input
$a = <STDIN>;                       # Get input
chop $a;                            # Chop off newline
}
while ($a ne "fred")                # Redo while wrong input
```

3.6.5 条件结构

Perl 也允许使用 if/then/else 表达式。请看下面的例子。

```
if ($a)
    {
    print "The string is not empty\n";
    }
else
    {
    print "The string is empty\n";
    }
```

注意在 Perl 中，空字符被认为是假。If 结构中也可以使用嵌套结构。

```
if (!$a)# The ! is the not operator
    {
        print "The string is empty\n";
    }
elsif (length($a) == 1)# If above fails，try this
    {
        print "The string has one character\n";
    }
elsif (length($a) == 2)# If that fails，try this
    {
        print "The string has two characters\n";
    }
Else            # Now，everything has failed
    {
        print "The string has lots of characters\n";
    }
```

3.7　本章总结

本章详细介绍了 Linux 程序设计的基础知识，包括如何建立嵌入式 Linux 交叉编译环境、了解和使用工程管理器 make，以及 LinuxC/C++程序、汇编程序、Shell 编程、Perl 编程的基础，几乎包括了所有的 Linux 程序设计内容。通过本章的学习，读者将了解 Linux 程序设计的编译和管理环境，熟悉各类程序设计语言特点，为后面的实例学习打下坚实的基础。

第 4 章

Linux 常用开发工具

工欲善其事，必先利其器。本章将重点介绍一些 Linux 常用开发工具，包括：—GCC 编译器、gdb 调试器、Linux 汇编工具和–调试工具。

4.1 GCC 编译器

GCC 是 GNU 最优秀的自由软件之一，其主要提供 C/C++程序的编译、调试工作。在嵌入式 Linux 开发过程中，一般都采用 GCC 软件进行软件开发。本节将重点介绍如何使用 GCC 进行软件开发。

4.1.1 GCC 版本信息

GCC 是一种 C++语言的编译器，它可以免费获得，其功能十分强大。你可以在 http://ftp.gnu.org/gnu/gcc/的网站上找到正式的 Linux GCC 发布系统。

在“#”提示符号下键入 gcc -v，屏幕上就会显示出你目前正在使用的 GCC 的版本。同时也可确定你现在所用的是 ELF 格式或是 a . out 格式。

```
[root@yangzongde root]# gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-
checking --with-system-zlib --enable-__cxa_atexit --host=i386-redhat-linux
Thread model: posix
gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)
```

在以上的信息中显示了以下几个主要信息：。

（1）–i386–这是指明你现在正在用的 gcc 是为 386 微处理器写的。这 3 种微处理器的芯片所编译而成的程序代码，彼此间是可以兼容使用的。

（2）redhat 可以说没有什么用处，当前 linuxLinux 版本为 redhat。

（3）linux 其实这是指 linuxelf 或是 linuxaout。这一项会令人使人引起不必要的困惑，究竟是指哪一种会根据你所用的版本而异定。linux Linux 若版本序号是 2.7.0（或者更新）就是指 ELF 格式，否则就是指 a.out 格式。

（4）3.2.2 是版本的序号。

4.1.2 GCC 目录结构

安装后，与 GCC 相关以下的几个重要目录分别是：

（1）/usr/lib/gcc-lib/i386-redhat-linux/（与子目录）：大部分的编译器就是在~~这个地方~~这里。~~在~~这里有可执行的程序做编译的工作；另外，还有一些特定版本的程序库与头文件等也会保存在这里。

（2）/usr/bin/gcc：编译器的驱动程序，也就是~~你~~实际在命令行上执行的程序。这个目录可供各种版本的 gcc 使用，只要用不同的编译器目录~~（如上所述）~~来安装就可以了。如果想强迫执行某个版本，就键入 gcc -V version。例如：

```
    [root@yangzongde root]# gcc -v
    Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/specs
    Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-
checking --with-system-zlib --enable-__cxa_atexit --host=i386-redhat-linux
    Thread model: posix
    gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)
    [root@yangzongde root]# gcc -V 2.6.3 -v
    Using built-in specs.
    Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-
checking --with-system-zlib --enable-__cxa_atexit --host=i386-redhat-linux
    Thread model: posix
    gcc driver version 3.2.2 20030222 (Red Hat Linux 3.2.2-5) executing gcc version
2.6.3
```

如果~~你~~安装了几种目标对象，例如 a.out 与 elf，或者某种交叉编译器，那些属于非主流目标对象的程序库~~——~~binutils（as、ld 等~~等~~），工具与头文件等都可以在这里找到。即使~~你~~只安装了一种 gcc，还是可以在这里找到这些原本就是替它们准备的东西。如果不在这个目录中，那么应该是在/usr/ (bin|lib|include)中。

4.1.3　GCC 执行过程

gcc 编译器能将 C、C++语言源程序、汇编程序和目标程序编译、连接成可执行文件，如果没有给出可执行文件的名字，gcc 将生成一个名为 a.out 的文件。在 Linux 系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。而 gcc 则通过后缀来区别输入文件的类别，下面来介绍 gcc 所遵循的部分约定规则。

（1）.c 为后缀的文件：C 语言源代码文件。

（2）.a 为后缀的文件：由目标文件构成的档案库文件。

（3）.C，.cc 或.cxx 为后缀的文件：是 C++源代码文件。

（4）.h 为后缀的文件：程序所包含的头文件。

（5）.i 为后缀的文件：已经预处理过的 C 源代码文件。

（6）.ii 为后缀的文件：已经预处理过的 C++源代码文件。

（7）.m 为后缀的文件：Objective-C 源代码文件。

（8）.o 为后缀的文件：编译后的目标文件。

（9）.s 为后缀的文件：汇编语言源代码文件。

（10）.S 为后缀的文件：经过预编译的汇编语言源代码文件。

虽然 gcc 是 C 语言的编译器，但使用 gcc 由 C 语言源代码文件生成可执行文件的过程不仅仅是编译的过程，而是要经历下面四个相互关联的步骤。

（1）预处理（也称预编译，Preprocessing）：命令 gcc 首先调用 cpp 进行预处理，在预处理过程中，对源代码文件中的文件包含（include）、预编译语句（如宏定义 define 等）进行分析。

（2）编译（Compilation）：接着调用 cc1 进行编译，这个阶段根据输入文件生成以.o 为后缀的目标文件。

（3）汇编（Assembly）：汇编过程是针对汇编语言的步骤，调用 as 进行工作，一般来讲，.S 为后缀的汇编语言源代码文件和.s 为后缀的汇编语言文件经过预编译和汇编之后都生成以.o 为后缀的目标文件。

（4）连接（Linking）：当所有的目标文件都生成之后，gcc 就调用 ld 来完成最后的关键性工作，这个阶段就是连接。在连接阶段，所有的目标文件被安排在可执行程序中恰当的位置，同时，该程序所调用到的库函数也从各自所在的档案库中连到合适的地方。

4.1.4 GCC 的基本用法和选项

在使用 gcc 编译器时，必须给出一系列必要的调用参数和文件名称。gcc 编译器的调用参数大约有 100 多个，其中多数参数可能根本就用不到，这里只介绍其中最基本、最常用的参数。

gcc 最基本的用法是：

```
gcc [options] [filenames].
```

其中 options 就是编译器所需要的参数；filenames 给出相关的文件名称。常用参数如下所示。

（1）-c：只编译，不连接成为可执行文件，编译器只是由输入的.c 为后缀的源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。

（2）-o：output_filename，确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

（3）-g：产生符号调试工具（GNU 的 gdb）所必要的符号资讯，要想对源代码进行调试，就必须加入这个选项。

（4）-O：对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是编译、连接的速度就相应地要慢一些。

（5）-O2：比-O 更好地优化编译、连接，当然整个编译、连接过程会更慢。

（6）-Idirname：将 dirname 所指出的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。

（7）-E：生成.i 文件，让 gcc 在预处理后停止编译，从而生成.i 文件，此文件中包含所有预处理信息。

详细参数说明请读者参阅附件内容 GCC 中文手册。

4.1.5 g++

gcc 中包含专用于 C++程序编译的程序 g++，该编译器能够读取并编译任何 C++程序，在编译时需要使用如下命令来编译链接 C++程序。

```
G++ -c *.cpp
```

常见的 C++源程序扩展名有.C（大写）、.cc、.cxx。

4.2　gdb 调试器

GNU 的调试器称为 gdb，该程序是一个交互式工具，工作在字符模式。在 X Window 系统中，有一个 gdb 的前端图形工具，称为 xxgdb。或许很多程序员习惯了图形界面的程序开发，如 VC、VB 等集成开发环境，但是在 UNIX/Linux 环境下开发软件，gdb 比传统 C 语言的继承开发环境具有更为强大的功能。

gdb 作为功能强大的调试程序，可完成如下的调试任务。

（1）设置断点；

（2）监视程序变量的值；

（3）程序的单步执行；

（4）修改变量的值。

在使用 gdb 调试程序之前，编译源文件时必须使用 -g 选项（即 gcc –c –g *.c）加上调试信息。另外，如果使用 makefile 文件，还可以在 makefile 中如下定义 CFLAGS 变量（关于 makefile 在本书第 2.2 节已有所介绍）。

```
CFLAGS = -g
```

4.2.1　基本用法和选项

运行 gdb 调试程序时通常使用如下命令。

```
gdb progname
```

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有如下几种。

（1）aliases：命令别名；

（2）breakpoints：断点定义；

（3）data：数据查看；

（4）files：指定并查看文件；

（5）internals：维护命令；

（6）running：程序执行；

（7）stack：调用栈查看；

（8）statu：状态查看；

（9）tracepoints：跟踪程序执行。

另外，如果键入 help 后跟命令的分类名，可获得该类命令的详细清单。

4.2.2　gdb 常用命令

以下列出了常用的 gdb 命令及解释。

（1）break NUM ：在指定的行上设置断点。

（2）bt：显示所有的调用栈帧。该命令可用来显示函数的调用顺序。

（3）clear：删除设置在特定源文件、特定行上的断点。其用法为：clear FILENAME:NUM。

（4）continue：继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行时。

（5）display EXPR：每次程序停止后显示表达式的值。表达式由程序定义的变量组成。

（6）file FILE：装载指定的可执行文件进行调试。

（7）help NAME：显示指定命令的帮助信息。

（8）info break：显示当前断点清单，包括到达断点处的次数等。

（9）info files：显示被调试文件的详细信息。

（10）info func：显示所有的函数名称。

（11）info local：显示当前函数中的局部变量信息。

（12）info prog：显示被调试程序的执行状态。

（13）info var：显示所有的全局和静态变量名称。

（14）kill：终止正在被调试的程序。

（15）list：显示源代码段。

（16）make：在不退出 gdb 的情况下运行 make 工具。

（17）next：在不单步执行进入其他函数的情况下，向前执行一行源代码。

（18）print EXPR：显示表达式 EXPR 的值。

（19）Shell shell 命令：不退出 gdb 运行 Shell 命令。

4.3 Linux 汇编工具

Linux 平台下的汇编工具种类很多，最基本的工具包括汇编器、链接器和调试器。

4.3.1 汇编器

汇编器（assembler）的作用是将用汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法，可以用来汇编用 AT&T 格式编写的程序。例如上述源程序即可用以下命令编译。

```
[root@localhots yangzongde]# as -o hello.o hello.s
```

Linux 平台上另一个经常用到的汇编器是 NASM，它提供了很好的宏指令功能，并能够支持相当多的目标代码格式，包括 bin、a.out、coff、elf、rdf 等。NASM 采用的是人工编写的语法分析器，因而执行速度要比 GAS 快很多，更重要的是它使用的是 Intel 汇编语法，可以用来编译用 Intel 语法格式编写的汇编程序。

```
[root@localhots yangzongde]# nasm -f elf hello.asm
```

4.3.2 链接器

由汇编器产生的目标代码必须经过链接器的处理才能生成可执行代码。链接器通常用来将多个目标代码链接成一个可执行代码，这样可以先将整个程序分成几个模块来单独开发，然后再将它们组合（链接）成一个应用程序。Linux 使用 ld 作为标准的链接程序，它同样也包含在 binutils 软件包中。汇编程序在成功通过 GAS 或 NASM 的编译并生成目标代码后，就可以使用 ld 将其链接成可执行程序了。

```
[xiaowp@gary code]$ ld -s -o hello hello.o
```

4.3.3　调试器

有人说程序不是编出来而是调出来的，足见调试在软件开发中的重要作用，在用汇编语言编写程序时尤其如此。Linux 下调试汇编代码既可以用 GDB、DDD 这类通用的调试器，也可以使用专门用来调试汇编代码的 ALD(Assembly Language Debugger)。

从调试的角度来看，使用 GAS 的好处是可以在生成的目标代码中包含符号表（symbol table），这样就可以使用 GDB 和 DDD 进行源码级的调试了。要在生成的可执行程序中包含符号表，可以采用下面的方式进行编译和链接。

```
[root@localhots yangzongde]#as --gstabs -o hello.o hello.s
[root@localhots yangzongde]#ld -o hello hello.o
```

执行 as 命令时带上参数--gstabs 可以告诉汇编器在生成的目标代码中加上符号表，同时需要注意的是，在用 ld 命令进行链接时不要加上-s 参数，否则目标代码中的符号表在链接时将被删去。

在 GDB 中调试汇编代码和调试 C 语言代码是一样的，可以通过设置断点来中断程序的运行，查看变量和寄存器的当前值，并可以对代码进行单步跟踪。另外，短小精悍的 ALD 可能更符合实际的需要。

4.3.4　系统调用

即便是最简单的汇编程序，也需要使用输入、输出以及退出等操作，而要进行这些操作都需要调用操作系统所提供的服务，即所谓的系统调用。在 Linux 平台下有两种方式使用系统调用：利用封装后的 C 库（libc）或者通过汇编直接调用。通过汇编语言直接调用系统调用，是最高效地使用 Linux 内核服务的方法，因为最终生成的程序不需要与任何库进行链接，而是直接和内核通信。

与 DOS 一样，Linux 的系统调用也是通过中断（int 0x80）来实现的。在执行 int 0x80 指令时，寄存器 eax 中存放的是系统调用的功能号，而传给系统调用的参数则必须按顺序放到寄存器 ebx、ecx、edx、esi、edi 中，当系统调用完成之后，返回值可以在寄存器 eax 中获得。

所有的系统调用功能号都可以在文件/usr/include/bits/syscall.h 中找到，为了便于使用，它们是用 SYS_<name>这样的宏来定义的，如 SYS_write、SYS_exit 等。例如，经常用到的 write 函数是如下定义的。

```
ssize_t write(int fd, const void *buf, size_t count);
```

该函数的功能最终是通过 SYS_write 这一系统调用来实现的。根据上面的约定，参数 bf、buf 和 count 分别存在寄存器 ebx、ecx 和 edx 中，而系统调用号 SYS_write 则放在寄存器 eax 中，当 int 0x80 指令执行完毕后，返回值可以从寄存器 eax 中获得。

4.3.5　命令行参数

在 Linux 操作系统中，当一个可执行程序通过命令行启动时，其所需的参数将被保存到栈中。首先是 argc，然后是指向各个命令行参数的指针数组 argv，最后是指向环境变量

的指针数据 envp。在编写汇编语言程序时，很多时候需要对这些参数进行处理，下面的代码示范了如何在汇编代码中进行命令行参数的处理。

```
#args.s
.text
.globl _start
_start:
popl    %ecx                # argc
vnext:
popl    %ecx                # argv
test    %ecx, %ecx          # 空指针表明结束
jz  exit
movl    %ecx, %ebx
xorl    %edx, %edx
strlen:
movb    (%ebx), %al
inc %edx
inc %ebx
test    %al, %al
jnz strlen
movb    $10, -1(%ebx)
    movl    $4, %eax        # 系统调用号(sys_write)
movl    $1, %ebx            # 文件描述符(stdout)
int $0x80
jmp vnext

exit:
movl    $1,%eax             # 系统调用号(sys_exit)
xorl    %ebx, %ebx          # 退出代码
int $0x80
ret
```

4.3.6 GCC 内联汇编

用汇编编写的程序虽然运行速度快，但开发速度非常慢，效率也很低。如果只是想对关键代码段进行优化，更好的办法是将汇编指令嵌入到 C 语言程序中，从而充分利用高级语言和汇编语言各自的特点。

GCC 提供了很好的内联汇编支持，最基本的格式是：

```
__asm__("asm statements");
```

例如：

```
__asm__("nop");
```

如果需要同时执行多条汇编语句，则应该用“\\n\\t”将各个语句分隔开，例如：

```
__asm__( "pushl %%eax \\n\\t" "movl $0, %%eax \\n\\t" "popl %eax");
```

通常嵌入到 C 代码中的汇编语句很难做到与其他部分没有任何关系，因此更多时候需要用到完整的内联汇编格式。

```
__asm__("asm statements" : outputs : inputs : registers-modified);
```

插入到 C 代码中的汇编语句是以“:”分隔的 4 个部分，其中:

（1）第一部分就是汇编代码本身，通常称为指令部，其格式和在汇编语言中使用的格式基本相同。指令部分是必需的，而其他部分则可以根据实际情况而省略。

在将汇编语句嵌入到 C 代码中时，操作数如何与 C 代码中的变量相结合是最值得关注的问题。GCC 采用如下方法来解决这个问题：程序员提供具体的指令，而对寄存器的使用则只需给出模式和约束条件就可以了，具体如何将寄存器与变量结合起来完全由 GCC 和 GAS 来负责。在 GCC 内联汇编语句的指令部中，加上前缀“%”的数字（如%0，%1）表示的就是需要使用寄存器的操作数。指令部中使用了几个这样的操作数，就表明有几个变量需要与寄存器相结合，这样 GCC 和 GAS 在编译和汇编时会根据后面给定的约束条件进行恰当的处理。另外，由于需要使用寄存器的操作数使用“%”作为前缀，因此在涉及具体的寄存器时，寄存器名前面应该加上两个“%”，以免产生混淆。

（2）紧跟在指令部后面的是输出部，是规定输出变量如何与需要使用寄存器的操作数进行结合的条件，每个条件称为一个“约束”，必要时可以包含多个约束，相互之间用逗号分隔开就可以了。每个输出约束都以“=”号开始，然后紧跟一个对操作数类型进行说明的字~~后~~，最后是如何与变量相结合的约束。凡是与输出部中说明的操作数相结合的寄存器或操作数本身，在执行完嵌入的汇编代码后均不保留执行之前的内容，这是 GCC 在调度寄存器时所使用的依据。

（3）输出部后面是输入部，输入约束的格式和输出约束相似，但不带“=”号。如果一个输入约束要求使用寄存器，则 GCC 在预处理时就会为之分配一个寄存器，并插入必要的指令将操作数装入该寄存器。与输入部中说明的操作数结合的寄存器或操作数本身，在执行完嵌入的汇编代码后也不保留执行之前的内容。

（4）有时在进行某些操作时，除了要用到进行数据输入和输出的寄存器外，还要使用多个寄存器来保存中间计算结果，这样就难免会破坏原有寄存器的内容。在 GCC 内联汇编格式中的最后一个部分中，可以对将产生副作用的寄存器进行说明，以便 GCC 能够采用相应的措施。

以下是一个内联汇编的简单例子。

```
/* inline.c */
int main()
{
    int a = 10, b = 0;
    __asm__ __volatile__("movl %1, %%eax;\n\r"
                         "movl %%eax, %0;"
                         :"=r"(b)      /* 输出 */
                         :"r"(a)       /* 输入 */
                         :"%eax");     /* 不受影响的寄存器 */
    printf("Result: %d, %d\n", a, b);
}
```

4.4　Linux 调试工具

本节将介绍 JTAG 调试工具以及 ~~KGDB~~kgdb 内核调试环境的内容。

4.4.1　JTAG 调试工具

这部分主要介绍 ARM JTAG 调试的基本原理。基本的内容包括了 TAP（TEST ACCESS

PORT）和 BOUNDARY-SCAN ARCHITECTURE 的介绍。

JTAG 是 JOINT TEST ACTION GROUP 的简称。IEEE 1149.1 标准就是由 JTAG 这个组织最初提出的，最终由 IEEE 批准并且标准化的。所以，这个 IEEE 1149.1 标准一般也俗称 JTAG 调试标准。

1．边界扫描

在 JTAG 调试当中，边界扫描（Boundary-Scan）是一个很重要的概念。边界扫描技术的基本思想是在靠近芯片的输入输出管脚上增加一个移位寄存器单元。因为这些移位寄存器单元都分布在芯片的边界上（周围），所以被称为边界扫描寄存器（Boundary-Scan Register Cell）。当芯片处于调试状态时，这些边界扫描寄存器可以将芯片和外围的输入输出隔离开来。通过这些边界扫描寄存器单元，可以实现对芯片输入输出信号的观察和控制。对于芯片的输入管脚，可以通过与之相连的边界扫描寄存器单元把信号（数据）加载到该管脚中去；对于芯片的输出管脚，也可以通过与之相连的边界扫描寄存器“捕获”（Capture）该管脚上的输出信号。在正常的运行状态下，这些边界扫描寄存器对芯片来说是透明的，所以正常的运行不会受到任何影响。这样，边界扫描寄存器提供了一个便捷的方式用以观测和控制所需要调试的芯片。另外，芯片输入输出管脚上的边界扫描（移位）寄存器单元可以相互连接起来，在芯片的周围形成一个边界扫描链（Boundary-Scan Chain）。一般的芯片都会提供几条独立的边界扫描链，用来实现完整的测试功能。边界扫描链可以串行地输入和输出，通过相应的时钟信号和控制信号，就可以方便地观察和控制处在调试状态下的芯片。

2．TAP 测试访问接口

利用边界扫描链可以实现对芯片的输入输出进行观察和控制。那么，如何来管理和使用这些边界扫描链？对边界扫描链的控制主要是通过 TAP（Test Access Port）Controller 来完成的。

在 IEEE 1149.1 标准中，寄存器被分为数据寄存器（DR－Data Register）和指令寄存器（IR－Instruction Register）两大类。边界扫描链属于数据寄存器中很重要的一种。边界扫描链用来实现对芯片的输入输出进行观察和控制。而指令寄存器用来实现对数据寄存器的控制，例如，在芯片提供的所有边界扫描链中，选择一条指定的边界扫描链作为当前的目标扫描链，并作为访问对象。下面从 TAP（Test Access Port）开始。

TAP 是一个通用的端口，通过 TAP 可以访问芯片提供的所有数据寄存器（DR）和指令寄存器（IR）。对整个 TAP 的控制是通过 TAP Controller 来完成的。TAP 总共包括 5 个信号接口 TCK、TMS、TDI、TDO 和 TRST。其中 4 个是输入信号接口，另外 1 个是输出信号接口。一般见到的开发板上都有一个 JTAG 接口，该 JTAG 接口的主要信号接口就是这 5 个。下面分别介绍这 5 个接口信号及其作用。

- Test Clock Input（TCK）：TCK 为 TAP 的操作提供了一个独立的、基本的时钟信号，TAP 的所有操作都是通过这个时钟信号来驱动的。TCK 在 IEEE 1149.1 标准中是强制要求的。
- Test Mode Selection Input（TMS）：TMS 信号用来控制 TAP 状态机的转换。通过 TMS 信号，可以控制 TAP 在不同的状态间相互转换。TMS 信号在 TCK 的上升沿有效。TMS 在 IEEE 1149.1 标准中是强制要求的。
- Test Data Input（TDI）：TDI 是数据输入的接口。所有要输入到特定寄存器的数据都

是通过 TDI 接口一位一位串行输入的（由 TCK 驱动）。TDI 在 IEEE 1149.1 标准中是强制要求的。

- Test Data Output（TDO）：TDO 是数据输出的接口。所有要从特定的寄存器中输出的数据都是通过 TDO 接口一位一位串行输出的（由 TCK 驱动）。TDO 在 IEEE 1149.1 标准中是强制要求的。
- Test Reset Input（TRST）：TRST 可以用来对 TAP Controller 进行复位（初始化）。不过这个信号接口在 IEEE 1149.1 标准中是可选的，并不是强制要求的。因为通过 TMS 也可以对 TAP Controller 进行复位（初始化）。

通过 TAP 接口对数据寄存器（DR）进行访问的一般过程是：

（1）通过指令寄存器（IR）选定一个需要访问的数据寄存器；

（2）把选定的数据寄存器连接到 TDI 和 TDO 之间；

（3）由 TCK 驱动，通过 TDI 把需要的数据输入到选定的数据寄存器当中；同时把选定的数据寄存器中的数据通过 TDO 读出来。

如图 4-1 所示为 TAP 的状态机，其总共有 16 个状态。

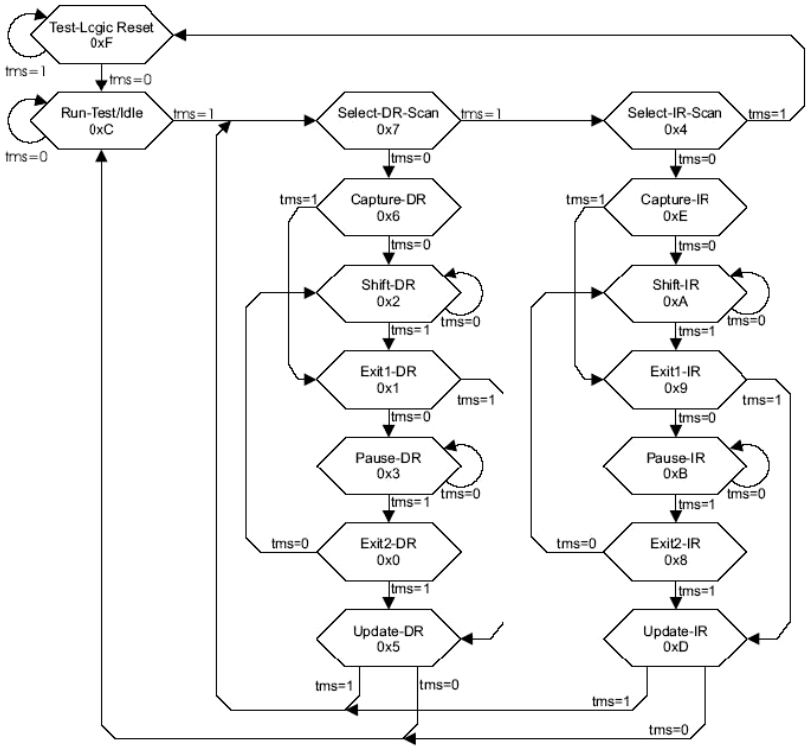


图 4-1　TAP 状态机

在图中，每个六边形表示一个状态，六边形中标有该状态的名称和标识代码。图中的箭头表示了 TAP Controller 内部所有可能的状态转换流程。状态的转换是由 TMS 控制的，所以在每个箭头上有标有 tms = 0 或者 tms = 1。在 TCK 的驱动下，从当前状态到下一个状态的转换是由 TMS 信号决定。假设 TAP Controller 的当前状态为 Select-DR-Scan，在 TCK 的驱动下，如果 TMS ＝ 0，TAP Controller 进入 Capture-DR 状态；如果 TMS = 1，TAP

Controller 进入 Select-IR-Scan 状态。

这个状态机看似很复杂，其实很直接、很简单。从图中可以看出，除了 Test-Logic Reset 和 Run-Test/Idle 状态外，其他的状态有些类似。例如 Select-DR-Scan 和 Select-IR-Scan 对应，Capture-DR 和 Capture-IR 对应，Shift-DR 和 Shift-IR 对应等。在这些对应的状态中，DR 表示 Data Register，IR 表示 Instruction Register。标识有 DR 的这些状态是用来访问数据寄存器的，而标识有 IR 的这些状态是用来访问指令寄存器的。

下面介绍每个状态具体含义及功能。

Test-Logic Reset：系统上电后，TAP Controller 自动进入该状态。在该状态下，测试部分的逻辑电路全部被禁用，以保证芯片核心逻辑电路的正常工作。通过 TRST 信号也可以对测试逻辑电路进行复位，使得 TAP Controller 进入 Test-Logic Reset 状态。前面介绍过 TRST 是可选的一个信号接口，这是因为在 TMS 上连续加 5 个 TCK 脉冲宽度的“1”信号也可以对测试逻辑电路进行复位，使得 TAP Controller 进入 Test-Logic Reset 状态。所以，在不提供 TRST 信号的情况下，也不会产生影响。在该状态下，如果 TMS 一直保持为“1”，TAP Controller 将保持在 Test-Logic Reset 状态下；如果 TMS 由“1”变为“0”（在 TCK 的上升沿触发），将使 TAP Controller 进入 Run-Test/Idle 状态。

Run-Test/Idle：这个是 TAP Controller 在不同操作间的一个中间状态。这个状态下的动作取决于当前指令寄存器中的指令。有些指令会在该状态下执行一定的操作，而有些指令在该状态下不需要执行任何操作。在该状态下，如果 TMS 一直保持为“0”，TAP Controller 将一直保持在 Run-Test/Idle 状态下；如果 TMS 由“0”变为“1”（在 TCK 的上升沿触发），将使 TAP Controller 进入 Select-DR-Scan 状态。

Select-DR-Scan：这是一个临时的中间状态。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 进入 Capture-DR 状态，后续的系列动作都将以数据寄存器作为操作对象；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入 Select-IR-Scan 状态。

Capture-DR：当 TAP Controller 在这个状态中，在 TCK 的上升沿，芯片输出管脚上的信号将被“捕获”到与之对应的数据寄存器的各个单元中去。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 进入 Shift-DR 状态；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入 Exit1-DR 状态。

Shift-DR：在这个状态中由 TCK 驱动，每一个时钟周期，被连接在 TDI 和 TDO 之间的数据寄存器将从 TDI 接收一位数据，同时通过 TDO 输出一位数据。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 保持在 Shift-DR 状态；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入到 Exit1-DR 状态。假设当前的数据寄存器的长度为 4，如果 TMS 保持为 0，那在 4 个 TCK 时钟周期后，该数据寄存器中原来的 4 位数据（一般是在 Capture-DR 状态中捕获的数据）将从 TDO 输出来；同时该数据寄存器中的每个寄存器单元中将分别获得从 TDI 输入的 4 位新数据。

Update-DR：在 Update-DR 状态下，由 TCK 上升沿驱动，数据寄存器当中的数据将被加载到相应的芯片管脚上去，用以驱动芯片。在该状态下，如果 TMS 为“0”，TAP Controller 将回到 Run-Test/Idle 状态；如果 TMS 为“1”，TAP Controller 将进入 Select-DR-Scan 状态。

Select-IR-Scan：这是一个临时的中间状态。如果 TMS 为“0”（在 TCK 的上升沿

触发），TAP Controller 进入 Capture-IR 状态，后续的系列动作都将以指令寄存器作为操作对象；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入 Test-Logic Reset 状态。

Capture-IR：当 TAP Controller 在这个状态中，在 TCK 的上升沿，一个特定的逻辑序列将被装载到指令寄存器中去。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 进入 Shift-IR 状态；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入 Exit1-IR 状态。

Shift-IR ：在这个状态中由 TCK 驱动，每一个时钟周期，被连接在 TDI 和 TDO 之间的指令寄存器将从 TDI 接收一位数据，同时通过 TDO 输出一位数据。如果 TMS 为“0”（在 TCK 的上升沿触发），TAP Controller 保持在 Shift-IR 状态；如果 TMS 为“1”（在 TCK 的上升沿触发），TAP Controller 进入到 Exit1-IR 状态。假设指令寄存器的长度为 4，如果 TMS 保持为 0，那在 4 个 TCK 时钟周期后，指令寄存器中原来的 4 位长的特定逻辑序列（在 Capture-IR 状态中捕获的特定逻辑序列）将从 TDO 输出来，该特定的逻辑序列可以用来判断操作是否正确；同时指令寄存器将获得从 TDI 输入的一个 4 位长的新指令。

Update-IR：在这个状态中，在 Shift-IR 状态下输入的新指令将被用来更新指令寄存器。

下面介绍指令寄存器和数据寄存器访问的一般过程，以便给读者一个直观的印象。

（1）系统上电，TAP Controller 进入 Test-Logic Reset 状态，然后依次进入 Run-Test/Idle、Select-DR-Scan、Select-IR-Scan、Capture-IR、Shift-IR、Exit1-IR、Update-IR 状态，最后回到 Run-Test/Idle 状态。在 Capture-IR 状态中，一个特定的逻辑序列被加载到指令寄存器当中；然后进入到 Shift-IR 状态。在 Shift-IR 状态下，通过 TCK 的驱动，可以将一条特定的指令送到指令寄存器当中去。每条指令都将确定一条相关的数据寄存器。然后依次进入 Shift-IR、Exit1-IR、Update-IR 状态。在 Update-IR 状态，刚才输入到指令寄存器中的指令将用来更新指令寄存器。最后，进入到 Run-Test/Idle 状态，指令生效，完成对指令寄存器的访问。

（2）当前可以访问的数据寄存器由指令寄存器中的当前指令决定。要访问由刚才的指令选定的数据寄存器，需要以 Run-Test/Idle 为起点，依次进入 Select-DR-Scan、Capture-DR、Shift-DR、Exit1-DR、Update-DR 状态，最后回到 Run-Test/Idle 状态。在这个过程当中，被当前指令选定的数据寄存器会被连接在 TDI 和 TDO 之间。通过 TDI 和 TDO，就可以将新的数据加载到数据寄存器当中去，同时，也可以捕获数据寄存器中的数据。具体过程如下：在 Capture-DR 状态中，由 TCK 的驱动，芯片管脚上的输出信号会被“捕获”到相应的边界扫描寄存器单元中去。这样当前的数据寄存器中就记录了芯片相应管脚上的输出信号。接下来从 Capture-DR 状态进入到 Shift-DR 状态中去。在 Shift-DR 状态中，由 TCK 驱动，在每一个时钟周期内，一位新的数据可以通过 TDI 串行输入到数据寄存器当中去，同时，数据寄存器可以通过 TDO 串行输出一位先前捕获的数据。在经过与数据寄存器长度相同的时钟周期后，就可以完成新信号的输入和捕获数据的输出。接下来通过 Exit1-DR 状态进入到 Update-DR 状态。在 Update-DR 状态中，数据寄存器中的新数据被加载到与数据寄存器的每个寄存器单元相连的芯片管脚上去。最后，回到 Run-Test/Idle 状态，完成对数据寄存器的访问。

3．指令寄存器、公共指令以及数据寄存器介绍

在 IEEE 1149.1 标准当中，规定了一些指令寄存器、公共指令和相关的一些数据寄存器。对于特定的芯片而言，芯片厂商都一般都会在 IEEE 1149.1 标准的基础上，扩充一些私有的指令和数据寄存器，以帮助在开发过程中进行方便的测试和调试。在这一部分将简单介绍 IEEE 1149.1 规定的一些常用的指令及其相关的寄存器。

（1）指令寄存器：指令寄存器允许特定的指令被装载到指令寄存器当中，用来选择需要执行的测试，或者选择需要访问的测试数据寄存器。每个支持 JTAG 调试的芯片必须包含一个指令寄存器。

（2）BYPASS 指令和 Bypass 寄存器：Bypass 寄存器是一个一位的移位寄存器，通过 BYPASS 指令，可以将 Bypass 寄存器连接到 TDI 和 TDO 之间。在不需要进行任何测试时，将 Bypass 寄存器连接在 TDI 和 TDO 之间，在 TDI 和 TDO 之间提供一条长度最短的串行路径。这样允许测试数据可以快速地通过当前的芯片送到开发板上别的芯片上去。

（3）IDCODE 指令和 Device Identification 寄存器：Device Identification 寄存器中可以包括生产厂商的信息、部件号码和器件的版本信息等。使用 IDCODE 指令，就可以通过 TAP 来确定器件的这些相关信息。例如，ARM MULTI-ICE 可以自动识别当前调试的是什么片子，其实就是通过 IDCODE 指令访问 Device Identification 寄存器来获取的。

（4）INTEST 指令和 Boundary-Scan 寄存器：Boundary-Scan 寄存器就是前面例子中说到的边界扫描链。通过边界扫描链，可以进行部件间的连通性测试。当然更重要的是可以对测试器件的输入输出进行观测和控制，以达到测试器件的内部逻辑的目的。INTEST 指令是在 IEEE 1149.1 标准里面定义的一条很重要的指令。结合边界扫描链，该指令允许对开发板上器件的系统逻辑进行内部测试。在 ARM JTAG 调试中，这是一条频繁使用的测试指令。

4.4.2 ~~KGDB~~kgdb 内核调试环境

Linux 内核调试器（Linux kernel debugger，kdb）是 Linux 内核的补丁，它提供了一种在系统运行时对内核内存和数据结构进行检查的办法。Kprobes 提供了一个强行进入任何内核例程，并从中断处理器无干扰地收集信息的接口。使用 Kprobes 可以轻松地收集处理器寄存器和全局数据结构等调试信息，而无须对 Linux 内核频繁编译和启动。这些调试技术的一个共同的特点在于，它们都不能提供源代码级的有效的内核调试手段，有些只能称为错误跟踪技术，因此这些方法都只能提供有限的调试能力。下面将介绍三种实用的源代码级的内核调试方法。

图 4-2、图 4-3 分别是 Linux 不同内核对调试技术的支持（在配置 Linux 内核程序时可以查看）。其中，图 4-2 为 2.4 内核对调试技术的支持，图 4-3 为 2.6 内核对调试技术的支持。

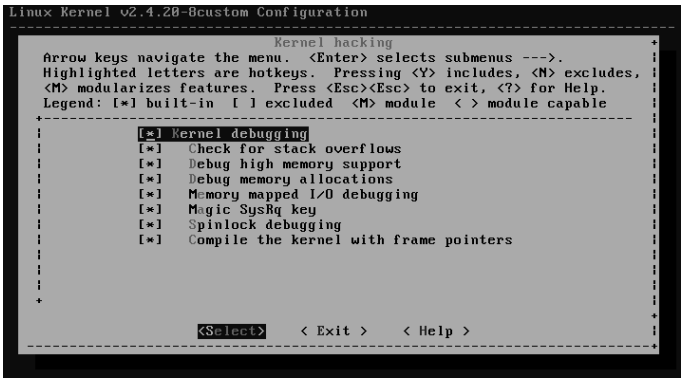


图 4-2　Linux 2.4 内核对调试技术的支持

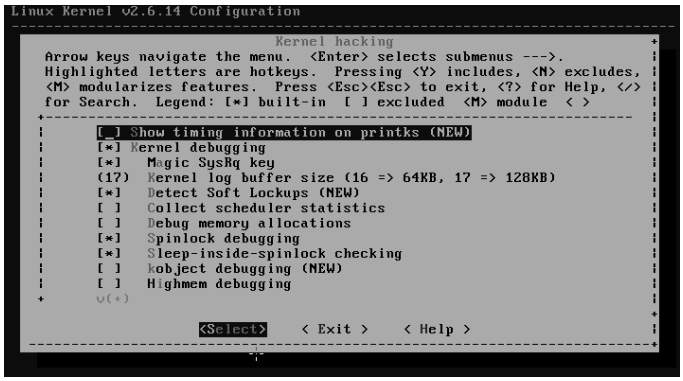


图 4-3　Linux 2.6 内核对调试技术的支持

1．使用 ~~KGDB~~kgdb 构建 Linux 内核调试环境

~~kgdb~~kgdb 提供了一种使用 gdb 调试 Linux 内核的机制。使用 ~~KGDB~~kgdb 可以像调试普通的应用程序那样，在内核中进行设置断点、检查变量值、单步跟踪程序运行等操作。使用 ~~KGDB~~kgdb 调试时需要两台机器，一台作为宿主机（Development Machine），另一台作为目标板（Target Machine），两台机器之间通过串口或者以太网口相连。串口连接线是一根 RS-232 接口的电缆，在其内部两端的第 2 脚（TXD）与第 3 脚（RXD）交叉相连，第 7 脚（接地脚）直接相连。调试过程中，被调试的内核运行在目标板上，gdb 调试器运行在宿主机上。目前，~~kgdb~~kgdb 发布支持 i386、x86_64、32-bit PPC、SPARC 等几种体系结构的调试器。

安装 ~~kgdb~~kgdb 调试环境需要为 Linux 内核应用 ~~kgdb~~kgdb 补丁，补丁实现的 gdb 远程调试所需要的功能包括命令处理、陷阱处理及串口通信 3 个主要的部分。~~kgdb~~kgdb 补丁的主要作用是在 Linux 内核中添加了一个调试 stub。调试 stub 是 Linux 内核中的一小段代码，提供了运行 gdb 的宿主机和所调试内核之间的一个媒介。gdb 和调试 stub 之间通过 gdb 串行协议进行通信。gdb 串行协议是一种基于消息的 ASCII 码协议，包含了各种调试命令。当设置断点时，~~kgdb~~kgdb 负责在设置断点的指令前增加一条 trap 指令，当执行到断点时控制权就转移到调试 stub 中去。此时，调试 stub 的任务就是使用远程串行通信协议将当前环境传送给 gdb，然后从 gdb 处接受命令。gdb 命令告诉 stub 下一步该做什么，当 stub 收到

继续执行的命令时，将恢复程序的运行环境，把对 CPU 的控制权重新交还给内核。

如图 4-4 所示连接好串口线后，使用以下命令来测试两台机器之间串口连接情况，stty 命令可以对串口参数进行设置。

在宿主机上执行：

```
# stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

在目标板上执行：

```
# stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

在 developement 机上执行：

```
# echo hello > /dev/ttyS0
```

在 target 机上执行：

```
# cat /dev/ttyS0
```

如果串口连接没问题的话在将在 target 机的屏幕上显示"hello"。

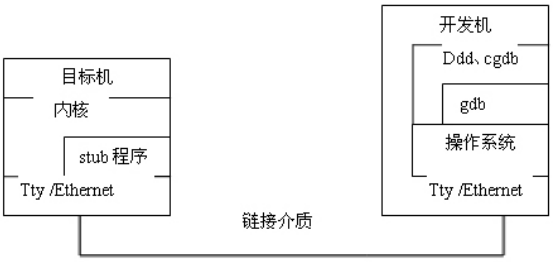


图 4-4　目标板－宿主机调试模式

2．安装与配置

以下介绍应用 ~~kgdb~~kgdb 补丁到 Linux 内核中，从而配置内核选项并编译。以下是如何重新配置编译内核。

（1）内核的配置与编译

```
# tar -jxvf linux-2.6.7.tar.bz2
#tar -jxvf linux-2.6.7-~~kgdb~~kgdb-2.2.tar.tar
#cd inux-2.6.7
```

应用补丁的命令如下所示。

```
#patch -p1 <../linux-2.6.7-~~kgdb~~kgdb-2.2/core-lite.patch
```

如果内核正确，那么应用补丁时应该不会出现任何问题（不会产生*.rej 文件）。为 Linux 内核添加了补丁之后，需要进行内核的配置。内核的配置可以按照习惯选择配置 Linux 内核的任意一种方式。

```
#make menuconfig
```

在内核配置菜单的 Kernel hacking 选项中选择 ~~kgdb~~kgdb 调试项，例如：

```
  [*] ~~KGDB~~kgdb: kernel debugging with remote gdb
      Method for ~~KGDB~~kgdb communication (~~KGDB~~kgdb: On generic serial port
```

```
(8250))  --->
      [*] KGDBkgdb: Thread analysis
      [*] KGDBkgdb: Console messages through gdb
    #make
```

内核编译完成后，将内核文件重新上载到目标板上。

（2）kgdbkgdb 的启动

在将编译出的内核上传到目标板之后，需要配置系统引导程序，加入内核的启动选项。在 kgdbkgdb 2.0 版本之后内核的引导参数已经与以前的版本有所不同。使用 grub 引导程序时，直接将 kgdbkgdb 参数作为内核 vmlinuz 的引导参数。下面给出引导器的配置示例。

```
    title 2.6.7 kgdbkgdb
    root (hd0,0)
    kernel  /boot/vmlinuz-2.6.7-kgdbkgdb  ro  root=/dev/hda1  kgdbkgdbwait
kgdbkgdb8250= 1,115200
```

在使用 lilo 作为引导程序时，需要把 kgdbkgdb 参数放在由 append 修饰的语句中。下面给出使用 lilo 作为引导器时的配置示例。

```
    image=/boot/vmlinuz-2.6.7-kgdbkgdb
    label=kgdbkgdb
       read-only
       root=/dev/hda3
    append="gdb gdbttyS=1 gdbbaud=115200"
```

保存好以上配置后重新启动计算机，选择启动带调试信息的内核，内核将在短暂的运行后在创建 init 内核线程之前停下来，打印出以下信息，并等待宿主机的连接。

```
    Waiting for connection from remote gdb...
```

在宿主机上执行：

```
    gdb
    file vmlinux
    set remotebaud 115200
    target remote /dev/ttyS0
```

其中 vmlinux 是指向源代码目录下编译出来的 Linux 内核文件的链接，它是没有经过压缩的内核文件，gdb 程序从该文件中得到各种符号地址信息。

这样，就与目标板上的 kgdbkgdb 调试接口建立了联系。一旦建立联系之后，对 Linux 内核的调试工作与对普通的应用程序的调试就没有什么区别了。任何时候都可以通过键入“Ctrl+C”组合键打断目标板的执行，进行具体的调试工作。

在 kgdbkgdb 2.0 之前的版本中，编译内核后在 arch/i386/kernel 目录下还会生成可执行文件 gdbstart。将该文件复制到 target 机器的/boot 目录下，此时无须更改内核的启动配置文件，直接使用命令。

```
    #gdbstart -s 115200 -t /dev/ttyS0
```

可以在 KGDBkgdb 内核引导启动完成后建立宿主机与目标板之间的调试联系。

3．通过网络接口进行调试

kgdbkgdb 也支持使用以太网接口作为调试器的连接端口。在对 Linux 内核应用补丁包时，需应用 eth.patch 补丁文件。配置内核时在 Kernel hacking 中选择 kgdbkgdb 调试项，配

置 kgdbkgdb 调试端口为以太网接口，例如。

```
 [*]KGDBkgdb: kernel debugging with remote gdb
Method for KGDBkgdb communication (KGDBkgdb: On ethernet)  --->
( ) KGDBkgdb: On generic serial port (8250)
(X) KGDBkgdb: On ethernet
```

另外使用 eth0 网口作为调试端口时，grub.list 的配置如下。

```
title 2.6.7 kgdbkgdb
root (hd0,0)
kernel  /boot/vmlinuz-2.6.7-kgdbkgdb  ro  root=/dev/hda1  kgdbkgdbwait
kgdbkgdboe=@192.168.
5.13/,@192.168. 6.13/
```

其他的过程与使用串口作为连接端口时的设置过程相同。

4．模块的调试方法

内核可加载模块的调试具有其特殊性。由于内核模块中各段的地址是在模块加载进内核的时候才最终确定的，所以宿主机的 gdb 无法得到各种符号地址信息。所以，使用 kgdbkgdb 调试模块所需要解决的一个问题是，需要通过某种方法获得可加载模块的最终加载地址信息，并把这些信息加入到 gdb 环境中。

（1）在 Linux 2.4 内核中的内核模块调试方法

在 Linux 2.4.x 内核中，可以使用 insmod -m 命令输出模块的加载信息，例如：

```
# insmod -m hello.ko >modaddr
```

查看模块加载信息文件 modaddr 如下。

```
.this           00000060  c88d8000  2**2
.text           00000035  c88d8060  2**2
.rodata         00000069  c88d80a0  2**5
……
.data           00000000  c88d833c  2**2
.bss            00000000  c88d833c  2**2
……
```

在这些信息中，我们关心的只有 4 个段的地址：text、.rodata、.data、.bss。在宿主机上将以上地址信息加入到 gdb 中，就可以进行模块功能的测试了。

```
(gdb) Add-symbol-file hello.o 0xc88d8060 -s .data 0xc88d80a0 -s
.rodata 0xc88d80a0 -s .bss 0x c88d833c
```

这种方法也存在一定的不足，它不能调试模块初始化的代码，因为此时模块初始化代码已经执行过了。而如果不执行模块的加载又无法获得模块插入地址，更不可能在模块初始化之前设置断点了。对于这种调试要求可以采用以下替代方法。

（2）在目标板上用上述方法得到模块加载的地址信息，然后再用 rmmod 卸载模块。在 development 机上将得到的模块地址信息导入到 gdb 环境中，在内核代码调用初始化代码之前设置断点。这样，在 target 机上再次插入模块时，代码将在执行模块初始化之前停下来，这样就可以使用 gdb 命令调试模块初始化代码了。

（3）另外一种调试模块初始化函数的方法是：当插入内核模块时，内核模块机制将调用函数 sys_init_module（kernel/modle.c）执行对内核模块的初始化，该函数将调用所插入

模块的初始化函数。程序代码片断如下：

```
…… ……
    if (mod->init != NULL)
        ret = mod->init();
…… ……
```

5．使用 printk 进行调试

Printk（）是调试内核代码时最常用的一种技术。在内核代码中的特定位置加入 printk（） 调试调用，可以直接把所关心的信息打印到屏幕上，从而可以观察程序的执行路径和所关心的变量、指针等信息。

Printk 具有可以随时调用、在中断中调用、在进程上下文中调用、在持有锁时调用以及在多处理器上同时使用等优点，但是，printk 在终端启动前无法调用。

Printk 有以下几种记录等级。

（1）0 KERN_EMERG：紧急情况。

（2）1KERN_ALERT：需要立即被注意到的。

（3）2KERN_CRIT：临界情况。

（4）3KERN_ERR：错误。

（5）4KERN_WARNING：警告。

（6）5KERN_NOTICE：普通的，可能需要注意。

（7）6 KERN_INFO：非正式的。

（8）7KERN_DEBUG：一般的调试信息。

例如：

```
printk( KERN_WARNING"This is a warning\n");
printk( KERN_DEBUG"This is a warning!\n" );
printk( "No LogLevel is specified!\n");
```

另外还有一些其他调试手段，例如用户空间的守护进程 klogd，用来从记录缓冲区获取内核消息，其特点为：

（1）只有日记级别小于 console_loglevel，消息才能显示出来，console_loglevel 的值可以通过 sys_syslogd 系统调用进行修改；

（2）载入 klogd 时，可以使用-c 标志改变终端的记录等级；

（3）运行 klogd 后，消息将追加到/var/log/messages；

（4）没有运行 klogd，消息不会传递到用户空间，此时可以查看/proc/kmsg 文件。

另外，syslogd 进程也可以用来进行调试，其特点为：

（1）保存 klogd 进程获取的内核消息到系统日志文件中；

（2）默认的文件是/var/log/messages；

（3）可通过/etc/syslog.conf 文件重新配置；

（4）如果没有运行 klogd 进程，数据将保留在循环缓冲区中，直到某个进程读取和缓冲区溢出为止。

Linux 的调试技术很多，以上仅给出几点示例。读者可以根据需要查看 Linux 调试文件说明。

4.5 Linux 图形开发工具

常用的 Linux 图形开发工具有 GUI、GTK 以及 QT 图形开发工具，本节逐一进行介绍。

4.5.1 GUI 图形界面开发

下面将介绍 GUI 图形界面设计技术。首先介绍 GUI 基础知识。

1．GUI 基础知识

（1）图形用户界面系统的结构模型

一个图形用户界面（GUI）系统通常由三个基本模型组成。它们是显示模型、窗口模型和用户模型。用户模型包含了显示和交互的主要特征，因此图形用户界面有时也仅指用户模型。最底层是计算机硬件平台。硬件平台之上是计算机的操作系统。大多数图形用户界面系统都只能在特定的操作系统上运行，只有少数的产品例外。操作系统之上是图形用户界面的显示模型。它决定了图形在屏幕上的基本显示方式。不同的图形用户界面系统所采用的显示模型各不相同。例如大多数在 UNIX 之上运行的图形用户界面系统都采用 X 窗口作显示模型；Microsoft Windows 则采用 Microsoft 公司自己设计的图形设备接口（GDI）作显示模型。

图形用户界面系统的层次结构包括用户模型、窗口模型、显示模型、操作系统、硬件平台。

显示模型之上是图形用户界面系统的窗口模型。窗口模型确定窗口如何在屏幕上显示、如何改变大小、如何移动以及窗口的层次关系等。它通常包括两部分：一是编程工具；二是对如何移动、输出和读取屏幕显示信息的说明。因为 X 窗口不但规定了如何显示基本图形对象也规定了如何显示窗口，所以它不但可以充当图形用户界面的显示模型，而且可以充当它的窗口模型。

窗口模型之上是用户模型，它也包括两部分：一是构造用户界面的工具；二是对于如何在屏幕上组织各种图形对象以及这些对象之间如何交互的说明。比如，每个图形用户界面模型都会说明它支持什么样的菜单和什么样的显示方式。图形用户界面系统的应用程序接口由其显示模型、窗口模型和用户模型的应用程序接口共同组成。例如 OSF/Motif 的应用程序接口就是由它的显示模型和窗口模型的应用程序接口 Xlib 和用户模型的应用程序接口 Xt Intrinsics 及 MotifToolkit 共同组成的。

（2）Linux 基本图形系统（函数库）

Linux 基本图形系统包括 X Window、SVGALib、FrameBuffer 等。这些系统（或者函数库）一般作为其他高级图形或者图形应用程序的基本函数库。

- X Window

X Window 是由美国麻省理工学院（MIT）推出的窗口系统，简称 X。它旨在建立不依赖于特定硬件系统的图形和文字显示窗口系统的标准。1987 年 9 月，MIT 推出了 X 系统的 11 版，称为 X11，它的出现标志着计算机工作站的一个新时代的到来。现在几乎所有的工作站都采用了 X 窗口的标准，这些工作站上的应用软件采用了基于 X Window 的软件平台。同时，计算机的 X 系统也日益增多。X 窗口系统之所以能受到人们的广泛青睐，与其优越的特点是分不开的。首先，它不依赖于硬件系统的特点，在任意一种计算机上用

X 系统开发的图形软件可以不需任何修改或只需极少改动就能移植到几十种其他类型的计算机上。其次，X 是一种基于网络的窗口系统，采用 X 的应用软件可以在由不同机器组成的网络上运行，能方便地在远程计算机上运行软件，而将结果显示到本机上。

基于 X 的应用软件是通过调用 X 的一系列 C 语言函数实现其各种功能的。这些函数称为 Xlib（X 库），提供了建立窗口、画图、处理用户操作事件等基本功能。Xlib 是一种底层库，用它来编写图形和交互界面程序虽非常灵活，但却比较复杂甚至烦琐。为此又发展出了一些比 Xlib 更高层的库函数，称做工具包（Toolkits），它们将一些常用的界面图形（如窗口、菜单、按键等，通常称做工具包中的组件(widgets)）按面向对象编程的方式组织到一起供应用软件使用，而工具包的 Intrinsics（内在、本质）允许在它们之上建立新的组件。

Xlib、X Intrinsic 以及工具包之间的关系是 Xlib 控制 X 协议以及网络问题，对开发者而言只是一些非常原始的接口函数，只提供基本的 C 语言 API。

在 Xlib 上是 X Intrinsic，X Intrinsic 为高层的工具包提供面向对象的框架，如果用户需要自己开发一套工具包，可以从这里开始。在其上就是工具包库。工具包提供完整的用户界面开发包，里面有“菜单”、“按钮”等基本的窗口对象，这些常被称做组件。开发者编写的程序可以基于上面三种的任何一种进行开发。其于 Xlib 库（工作量很大）、基于 XIntrinsic（仍然很困难）、工具包/组件（种类较多，开发起来也相对容易得多）。基本的结构如图 4-5 所示。

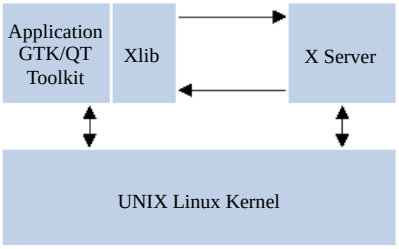


图 4-5　X 系统基本结构图

X 下的程序设计并不困难，但如果只是基于 Xlib，则类似单片机使用汇编语言编程，工作量比较大。如果界面要求不复杂，注重效率，可以使用这种方式。如果需要开发工具有完整 Windows 风格的程序，最好还是选用其他方法。基于 X 工具包进行开发以前是 X 程序开发的主流，不过 X 工具包提供的面向对象功能并不强，而且调用函数多，概念多，不容易上手。QT、GTK、MiniGUI 是开源软件中发展起来的优秀图形库，具有非常明显的面向对象的特点，特别是 QT，采用 C++类作为接口 API，具有界面美观（可以很像 Windows）、开发时间短（可以使用 VC 一类的图形工具画窗口）、运行效率高（直接基于 Xlib）的特点，已经成为目前进行 X Window 程序设计的首选。

- 客户—服务器模型

X Window 是一种协议，应用程序可以通过它在支持图显示和能接受输入的计算机上产生输出。X Window 系统建立在一种客户—服务器模型之上，这里应用程序是客户，它通过 X 协议与服务器联系，服务器完成直接向显示设备产生输出和接受输入的工作。另外，X 系统是一种基于网络的窗口系统。一个基于 X 系统的应用程序既可以在本机上运行，也可以在另外一台机器上运行，通过网络（TCP/IP 或 DECnet 网络）将输出与输入的工作交

到任一指定的显示终端上。这里控制显示的程序称做 Server 或 X Server（X 服务器），它是直接与计算机硬件平台打交道的程序。X 服务器起着在本地机或远程机上运行的用户应用程序和执行输入输出的本地机的资源之间的桥梁作用。

X Window 系统中经常被混淆的一个关键概念是服务器与客户之间的区别。在计算机网络中，服务器指的是向其他机器传输文件的机器，然而 X Window 系统中的服务器起的是完全不同的作用。X Window 系统中，服务器是从用户处接收输入并向用户显示输出的硬件（或）软件。例如，用户面前的键盘、鼠标和显示器就是服务器的一部分，即它们图形化地向用户提供信息“服务”。客户是指连接到服务器的应用程序。

在 X Window 系统中，服务器与客户可以存在于同一个工作站或者计算机上，使用进程间通信（IPC）机制，如 UNIX 管道与套接字，在它们之间传递信息。本地客户是指运行在用户面前的机器上的应用程序。远程客户是指运行在通过网络连接到用户的服务器上的应用程序。但不管是本地客户还是远程客户，对于 X Window 系统的用户来讲，外观与感受都是完全一样的。X Window 客户与服务器拓朴图如图 4-6 所示。

X Window 作为一个图形环境是成功的，它上面运行着包括 CAD 建模工具和办公套件在内的大量应用程序。但是，由于 X Window 在体系接口上的原因，限制了其对游戏、多媒体的支持能力。

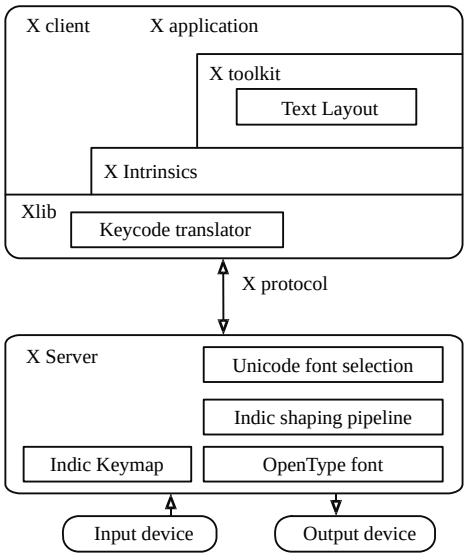


图 4-6　X Window 客户与服务器拓朴图

Tiny-X 是 X Server 在嵌入式系统的小巧实现，它由 Xfree86 Core Team 的 Keith Packard 开发。它的目标是运行于小内存系统环境，典型的运行于 X86 CPU 上的 Tiny-X Server 接近（小于）1MB。

- FrameBuffer

FrameBuffer 是出现在 2.2.xx 内核当中的一种驱动程序接口。这种接口将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反映在屏幕上。该驱动程序的设备文件一般是/dev/fb0、/dev/fb1 等。

在应用程序中，一般通过将 FrameBuffer 设备映射到进程地址空间的方式使用，比如下面的程序就打开/dev/fb0 设备，并通过 mmap 系统调用进行地址映射，随后用 memset 将屏幕清空（这里假设显示模式是 1024x766-8 位色模式，线性内存模式），代码如下。

```
int fb;
unsigned char* fb_mem;
fb = open ("/dev/fb0", O_RDWR);
fb_mem = mmap (NULL, 1024*768,
PROT_READ|PROT_WRITE,MAP_SHARED,fb,0);
memset (fb_mem, 0, 1024*768);
```

FrameBuffer 设备还提供了若干 ioctl 命令，通过这些命令，可以获得显示设备的一些固定信息（比如显示内存大小）、与显示模式相关的可变信息（比如分辨率、像素结构、每扫描线的字节宽度）以及伪彩色模式下的调色板信息等。

FrameBuffer 实际上只是一个提供显示内存和显示芯片寄存器从物理内存映射到进程地址空间中的设备。所以，对于应用程序而言，如果希望在 FrameBuffer 之上进行图形编程，还需要完成其他许多工作。FrameBuffer 就像一张画布，使用什么样的画笔，如何画画，还需要自己动手完成。

- LibGGI

GGI 即 General Graphics Interface，是新一代的图形支持库。

GGI 的主要功能特性如下：

- 可在 FrameBuffer、SVGALib、X 等设备上运行，在这些设备上是二进制兼容的；
- 在所有平台上提供了一致的输入设备接口，比如鼠标和键盘；
- 与 LinuxThreads 线程库兼容，接口线程安全；
- 提供异步绘制模式，可提高屏幕刷新速度；
- 提供良好的颜色处理接口；
- 接口简单易用；
- 采用共享库机制，实现底层支持库的动态装载；
- GGI 的主要不足在于安装和配置较为复杂。

2．Ncurses 库

（1）Ncurses 简介

假设在使用 terminfo 的情况下，让所有的应用程序访问 terminfo 数据库控制输出（比如发送控制字符等），不久这些调用代码将会使整个程序变得难以控制和管理。这些问题的出现导致了 CURSES 的诞生。CURSES 的命名是来自一个叫做“cursor optimization”（光标最优化）的双关语。CURSES 库通过对终端原始控制代码（转义序列）的封装，向用户提供了一个灵活高效的 API（应用程序接口）。它提供了一套控制光标、建立窗口、改变前景背景颜色以及处理鼠标操作的函数，使用户在字符终端下编写应用程序时绕过了那些麻烦的底层机制。

Ncurses 是从 System V Release 4.0（SVr4）中 CURSES 克隆过来的。这是一个可自由配置的库，完全兼容旧版本的 CURSES。简而言之，它是一个可以使应用程序直接控制终端屏幕显示的库。当后面提到 CURSES 库时，同时也是 Ncurses 库。

（2）Ncurses 的作用

Ncurses 不仅仅只是封装了底层的终端功能，还提供了一个相当稳固的工作框架（Framework）用以产生漂亮的界面。它包含了一些创建窗口的函数，而它的姊妹库 Menu、Panel 和 Form 则是对 CURSES 基础库的扩展，这些库一般都随同 CURSES 一起发行。可以建立一个同时包含多窗口（multiple windows）、菜单（menus）、面板（panels）和表单的应用程序。窗口可以被独立管理，例如让它卷动（scrollability）或者隐藏。菜单可以让用户建立命令选项，从而方便执行命令。而窗体允许用户建立一些简单的数据输入和显示的窗口。面板是 Ncurses 窗口管理功能的扩展，可以用它覆盖或堆积窗口。

（3）Ncurses 安装

在安装操作系统时（Linux）一般都附带 Ncurses。如果操作系统没有安装 Ncurses 库，可以通过以下的途径下载得到。

下载并编译安装文件包：用户可以通过 ftp://ftp.gnu.org/pub/gnu/ncurses/ncurses.tar.gz 免费下载 Ncurses，也可以通过 GNU 的 FTP 目录 http://www.gnu.org/order/ftp.html 找到提供免费下载 Ncurses 文件包的站点。

包裹文件中的 README 和 INSTALL 文件是安装 Ncurses 库的最主要资料。通常是这样安装 Ncurses 的。

```
tar zxvf ncurses.tar.gz  #解压缩并且释放文件包
```

Ncurses 程序包提供字符终端处理库，包括面板和菜单。

预计编译时间：0.6 SBU

所需磁盘空间：18.6 MB

安装依赖于：Bash, Binutils, Coreutils, Diffutils, Gawk, GCC, Glibc, Grep, Make, Sed

- 安装 Ncurses

首先准备编译 Ncurses。

./configure --prefix=/usr --with-shared --without-debug 编译软件包，make 这个软件包没有附带测试程序。然后安装软件包，make install 赋予 Ncurses 库文件可执行权限。

chmod -v 755 /usr/lib/*.5.4 修正一个不应该有可执行权限的库文件。

chmod -v 644 /usr/lib/libncurses++.a 把库文件移到更合理的/lib 目录里。

mv -v /usr/lib/libncurses.so.5* /lib 由于库文件移动了，所以有的符号链接就指向了不存在的文件。需要重新创建这些符号链接。

```
ln -sfv ../../lib/libncurses.so.5 /usr/lib/libncurses.so
ln -sfv libncurses.so /usr/lib/libcurses.so
```

- Ncurses 主要组成

安装的程序如下：

captoinfo (链接到 tic), clear, infocmp, infotocap (链接到 tic), reset (链接到 tset), tack, tic, toe, tput, tset

安装的库如下：

libcurses.[a,so] (链接到 libncurses.[a,so]), libform.[a,so], libmenu.[a,so], libncurses++.a, libncurses.[a,so], libpanel.[a,so]

简要描述如下。

captoinfo：将 termcap 描述转化成 terminfo 描述。

clear：如果可能，就进行清屏操作。

infocmp：比较或显示 terminfo 描述。

infotocap：将 terminfo 描述转化成 termcat 描述。

reset：重新初始化终端到默认值。

tack：terminfo 动作检测器。主要用来测试 terminfo 数据库中某一条目的正确性。

Tic：Tic 是 terminfo 项说明的编译器。这个程序通过 Ncurses 库将源代码格式的 terminfo 文件转换成编译后格式（二进制）的文件。Terminfo 文件包含终端能力的信息。

Toe：列出所有可用的终端类型，分别列出名称和描述。

Tput：利用 terminfo 数据库使与终端相关的能力和信息值对 shell 可用，初始化和重新设置终端，或返回所要求终端为类型的长名。

tset：可以用来初始化终端。

libcurses：链接到 libncurses。

libncurses：用来在显示器上显示文本的库。一个实例就是在内核的 make menuconfig 进程中。

libform：在 Ncurses 中使用表格。

libmenu：在 Ncurses 中使用菜单。

libpanel：在 Ncurses 中使用面板。

3．ncurse 程序示例分析

下面来了解和分析两个典型的 ncurse 程序示例。

（1）Hello World 程序分析

用户如果要调用 Ncurses 库中的函数，必须在代码中加载 ncurses.h 文件，就是在 C 或 C++程序中添加"#include <ncurses.h>"这一行，然后在连接程序中标记出 Ncurses（注：Ncurses 库已经包含了"stdio.h"）。

```
#include <ncurses.h>
编译和连接命令：gcc <程序文件> -lncurses
例 1Hello World 程序
#include <ncurses.h>
int main()
{
 initscr();                      /* 初始化，进入 NCURSES 模式     */
 printw("Hello World! ");        /* 在虚拟屏幕上打印 Hello World */
 refresh();                      /* 将虚拟屏幕上的内容写到显示器上，并刷新 */
 getch();                        /* 等待用户输入  */
 endwin();                       /* 退出 NCURSES 模式  */
 return 0;
}
```

上面这个程序在显示器屏幕上打印“Hello World！”后等待用户按任意键退出。这个小程序展示了如何初始化并进入 curses 模式、处理屏幕操作和退出 curses 模式。下面逐行分析这个小程序。

- initscr()函数

initscr()函数将终端屏幕初始化为 curses 模式。它清除屏幕上所有的字符，使屏幕变为空白，等待下一步处理，所以在调用其他 Ncurses 函数前，都要先调用 initscr()函数初始化屏幕。这个函数初始化 curses 系统，并且为当前屏幕一个被叫做“stdscr”的虚拟窗口以及其他相关的数据结构分配内存。在以前的计算机上曾经出现过一个非常极端的例子：因为系统中的可用内存太小，以至于 initscr()函数无法分配足够的内存给相关的数据结构，导致 curses 系统初始化失败。初学者学习时要特别注意。

- refresh()函数

第二行的 printw 函数用于把字符串“Hello World！”输出到虚拟的屏幕上。这个函数用法上和 printf()函数很像。但是区别在于 printw 函数把字符串输出到被称做“stdscr”的虚拟坐标（0,0）上。从显示的结果来看，坐标（0,0）在屏幕的左上角。

在使用 printw 函数打印“Hello World！”时，实际上这个数据被打印到一个叫做“stdscr”的虚拟窗口上，没有被直接输出到屏幕上。printw()函数的作用是不断将一些显示标记和相关的数据结构写在虚拟显示器上，并将这些数据写入 stdscr 的缓冲区内。所以为了显示这些缓冲区中的数据必须使用 refresh（）函数告诉 curses 系统将缓冲区的内容输出到屏幕上。

这种机制可以使程序员不断在虚拟屏幕上写入数据，而调用 refresh（）函数时让一切看起来似乎是一次完成的。因为 refresh（）函数只核查窗口和数据中变动的部分，这种富有弹性的设计提供了一个高效的反馈机制。但是这样有时会增加初学者的困难，因为对于初学者来说，忘记在输出后调用 refresh()函数是很烦人的错误。

- endwin()函数

在结束 curses 显示模式才可以返回到普通字符行模式，否则在程序结束后终端可能会运转得不正常。endwin()函数释放了 curses 子系统和相关数据结构的内存，使终端回到普通字符模式。这个函数必须是在完成所有的 curses 操作以后才可以调用（注意：如果在 endwin()函数后再调用 curses 的函数，那些语句不会执行。如果程序不能正常显示，请务必查看 initscr()函数和 endwin()函数是不是在不该被调用的地方调用了）。

（2）Ncurses 初始化程序分析

在程序中调用 initscr()函数，会让屏幕初始化并进入 CURSES 模式，还有一些其他的函数可以根据用户自己的方案初始化 CURSES。不同的初始化函数可以让屏幕进入不同的显示模式，比如终端模式（terminal mode）、彩色模式（color mode）、鼠标模式（mouse mode）等。当然还可以将这些模式混合起来。下面介绍一些经常使用的初始化函数。

- raw()和 cbreak()函数

通常情况下，用户输入的字符将被终端程序送入终端的缓冲区。但当用户输入换行符时，终端程序将会中断，同时输出当前的缓冲区内容并启用新行的输出缓冲。但是大多数程序需要当用户输入单个字符时这些字符能够立即显示在屏幕上。这两个函数就是用来禁用行缓冲（line buffering），所初始化的模式同样可以用来给程序传送控制字符，比如挂起（Ctrl+Z）、中断或退出（Ctrl+C）。区别在于在 raw()函数模式下，这些字符将传送给程序去处理而不作为终端程序处理的信号；在 cbreak()模式下，这些控制字符将被认为是终端驱动程序中的控制字符，因而将这些字符传送给终端程序。推荐使用 raw()，因为那样可以进行更多的控制操作。

- echo()和 noecho()函数

这两个函数控制用户输入的键盘回显，就是在运行程序时是否将输入的字符出现在屏

幕上，比如程序在运行时需要使用控制字符，但是不想让控制字符出现在屏幕上，就可以使用这两个函数。也就是说，当用户调用 getch()函数向程序输入数据时，不想让输入的字符出现在屏幕上。noecho()函数就可以不让控制字符（比如 Ctrl+C）出现在屏幕上。大多数的交互式程序要进入控制模式时，一般都使用 echo()、noecho()函数初始化、关闭键盘回显。这样给程序员更大的灵活性。

- keypad()函数

这个函数允许使用功能键 F1、F2、方向键等。几乎所有的交互式程序都使用这个函数，令用户使用方向键控制整个用户界面。使用 keypad（stdscr,TURE）就可以在“标准显示设备（stdscr）”上使用这些功能。

- halfdelay()函数

这个函数虽然不经常使用，但有时却非常有用。halfdelay()函数会启用半延时模式（half-delay mode），和 cbreak()函数一样，当程序需要当用户输入这些字符时，它们能够立即显示在屏幕上，但是它要停滞一段限定时间（以 0.1 秒为单位）等待输入，如果没有有效的输入，返回 ERR。给 halfdelay()传递一个整型参数（以 0.1 秒为单位），它就会按照参数中的时间等待用户输入。一般来说，这个函数在需要等待输入的程序中可以被用到，如果用户没有及时做出响应，程序就可以去处理其他的事情了。最常见到的应用实例是在输入密码时的超时响应。

- 其他的初始化函数

上面提到的这些函数可以定制 CURSES 在初始化后的行为。这些函数不能被广泛使用在程序的各个部分，所以这些函数的调用要处在整个 CURSES 会话的开始部分。

下面通过一个程序来说明这些函数的用法。

初始化函数用法的示例：

```
#include <ncurses.h>
int main()
{
int ch;
initscr();  /* 开始 curses 模式 */
raw();      /* 禁用行缓冲 */
keypad(stdscr, TRUE); /* 开启功能键响应模式 */
noecho();  /* 当执行 getch()函数的时候关闭键盘回显 */
printw("Type any character to see it in bold\n");
ch = getch();
/* 如果没有调用 raw()函数，必须按下 Enter 键才可以执行下面的程序 */
if(ch == KEY_F(1)) /* 如果没有调用 keypad（）初始化，将不会执行这条语句 */
printw("Type any character to see it in bold\n");
ch = getch();
/* 如果没有调用 raw()函数，必须按下 Enter 键才可以执行下面的程序 */
if(ch == KEY_F(1)) /* 如果没有调用 keypad（）初始化，将不会执行这条语句 */
printw("F1 Key pressed");
/* 如果没有使用 noecho() 函数，一些难看的控制字符将会被打印到屏幕上 */
else
{ printw("The pressed key is ");
attron(A_BOLD);
printw("%c", ch);
attroff(A_BOLD);
}
```

```
refresh(); /* 将缓冲区的内容打印到显示器上 */
getch(); /* 等待用户输入 */
endwin(); /* 结束 curses 模式 */
return 0;
}
```

上面这个程序很简单，不需要太多的说明，但其中还是有一些前面没有提到的函数。

getch()函数用来取得用户输入的信息，它和通常的 getchar()函数相似，但是 getch()可以在禁用行缓冲时避免在输入完成后还要按 Enter 键。attron()和 attroff()函数作为切换开关用来开启和关闭字符的修饰效果。在这个例子中它们使显示的字符字体加粗。

4.5.2 GTK 图形开发工具

下面将详细介绍 GTK 设计技术。首先介绍 GTK 的相关基础知识。

1．GTK 基础知识

GTK（GIMP Toolkit）是一个图形用户编程的接口工具。它注册完全免费，所以用来开发自由软件或商业软件不需要太多花费。现在很多 Linux 集成系统都已经将 GTK 打包进去了，包括 RedHat Linux、Ubantu 等发行版本，还有中文化的红旗、Turbo Linux 等。它也越来越被普遍地应用于 UNIX 系统编程。有一个组件叫 Glib，它包含了一些标准应用的新扩展，用来提高 GTK 的兼容性。

用于 Linux 系统的某些函数不适合标准的 UNIX 系统，例如 g_strerror()函数等。某些函数也扩展了 GNUC 的一般功能，例如 g_malloc 函数就有自己加强的调试功能。GTK 可以与多种语言绑定，包括 C++、Guile、Perl、Python、Ton、Ada95、Objective C、Free Pascal、Eiffel。用标准 C 开发的程序，编译软件可用 GNU 并附带上 GTK 选项即可。如果使用标准 C 语言以外的其他语言来开发 X Window 图形用户程序，则需要先参考一下有关绑定软件的内容。如果用 C++语言来调用 GTK 进行开发，可以用已经和 C++绑定的软件（叫 GTK 软件），来提供一个比 GTK 更好的 C++编译环境。

目前已经开发出 GTK 的增强版 GTK+。GTK+是将 GTK、GDK、GLIB 集成在一起的开发包，可以工作在许多类似于 UNIX 的系统上，没有 GTK 的平台限制。

GTK+是一种图形用户界面（GUI）工具包。它是一个库（或者实际上是若干个密切相关的库的集合），它支持创建基于 GUI 的应用程序。可以把 GTK+想像成一个工具包，从这个工具包中可以找到用来创建 GUI 的许多已经准备好的构造块。

最初，GTK+是作为另一个著名的开放源码项目——GNU Image Manipulation Program（GIMP）——的副产品而创建的。在开发早期的 GIMP 版本时，Peter Mattis 和 Spencer Kimball 创建了 GTK（它代表 GIMP Toolkit），作为 Motif 工具包的替代。

使用 GTK+这样的库比起编写自己的 GUI 代码来有多个优势。例如，它可以显著节约开发时间，让开发人员把精力集中在项目真正重要和真正独特的地方，而不必重复公共的功能。对于用户来说，这意味着他们使用的应用程序之间具有更好的一致性，工具包能在哪使用，应用程序就能跟到哪里，就像使用 LEGO 一样，所有的人都使用同一兼容尺寸这一事实意味着设计可以在使用库的人之间共享，不论他们在哪里使用。

GTK+采用了软件开发中的最新技术开发，只要发现缺陷，开发人员就会尽力在下一版本中修补缺陷。这意味着用户不会陷在过时的工作中，而跟不上时代的发展。持续的维

护和开发也意味着用户拥有影响工具包的未来发展方向的能力。另外，在出现新的发行版时，会引入基于用户反馈的新特性和新功能，而旧的问题则将得到修补。

GTL+特点叙述如下。

（1）国际化、本地化和可访问性

在创建要让所有人使用的软件时，请记住三个关键字：国际化、本地化和可访问性（通常分别缩写为 i18n、l10n 和 a11y）。国际化是将程序准备为被母语不是开发应用程序所采用的语言的人使用的过程，所以应用程序不依赖于对任何特定语言的任何假设。i18n 远远不只是对程序使用的文本进行翻译。它还意味着要考虑所使用的不同脚本和字母表、不同的编写方向、显示许多语言所需要的特殊处理以及为用户提供输入文本的适当方法。不是每种语言都可以简单地把每个字母映射到键盘上的不同键，同时还必须实现更好的复杂性，例如确保在错误消息中使用正确的复数。

本地化与 i18n 密切相关，因为为国际用户准备应用程序不仅仅是改变语言，程序还必须能够理解并尊重日期、货币显示、数字标注、文本排序所使用的不同习惯，以及许多可能不太注意的细节之处——例如有些符号的使用，在世界的不同地方可能会被认为是不恰当的或无礼的。与 i18n 一样，正确的 l10n 要求在代码中添加很多东西，而这些是事后很难轻松加入的。GTK+提供了针对 i18n 和 l10n 的恰当工具，会让代码（和二进制）可以在许多语言和地域上不加修改地运行。同时，GTK+对于开发人员也易于使用。它允许开发人员用简单的方式表达出自己想要的东西，不会给开发人员带来负担。

（2）设计良好、灵活和可扩展

编写 GTK+的方式允许在不改变基本设计的情况下，让维护人员添加新功能，让用户使用新功能。工具包也是可扩展的，用户可以向其中添加自己的块，并用使用内置块一样的方式使用它们，例如可以编写自己的控制元素。

同时，GTK+是可定制的，这样就可以适应自己的需求。GTK+有一个系统，可以在所有应用程序之间复制设置，包括主题的选择。主题是一组一同发布的定制设置，使用主题可以模拟另一个操作系统的观感。

GTK+是可移植的。一方面，这意味着用户可以在许多平台和系统上运行它。另一方面，开发人员可以把软件提供给众多用户，却只要编写一次程序，还可以使用许多不同的编程和开发平台、工具和编程语言。

所有这些优势组合在一起，让 GTK+成为软件开发的坚实基础。有了它，用户就能够把注意力集中在解决实际问题上，而不必创建新的应用程序。

2．GTK 程序示例解析

下面挑选出 2 个 GTK 程序示例，进行细致分析。

（1）简单的 GTK 应用程序

首先给出一个简单的应用程序。

```
*例子 base.c */
#include <gtk/gtk.h>
int main( int argc,char *argv[ ] )
{
GtkWidget *window;
gtk_init (&argc, &argv);        /* 初始化显示环境 */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL); /* 创建一个新的窗口*/
```

```
gtk_widget_show (window);        /*显示窗口*/
gtk_main ();                     /*进入睡眠状态，等待事件激活*/
return(0);
}
```

编译方式：

gcc base.c -o base `pkg-config --cflags --libs gtk+-2.0`

从上面的程序可以看出，GTK 是一个事件驱动工具包，当它运行到 gtk_main()函数时会自动睡眠，直到有事件发生，控制权转让给相应的函数调用，在该函数中可以用标准 C 语言写出相应的事务逻辑。这与 Windows 上的程序处理是一样的。

对窗口对象上发生的事件（如按下鼠标，激活键盘等），GTK 也有相应的消息信号产生。这时就需要程序员创建一个信号处理器来捕获该信号，并告诉 GTK 程序事件发生后调用哪个回调函数。信号处理器的创建函数定义如下：

```
gint gtk_signal_connect( GtkObject *object, gchar *name,GtkSignalFunc
callback_func, gpointer func_data );
```

返回值是一个区分同一对象中的事件与不同回调函数的关联标签。这样可以做到一个对象的一个信号有任意多个回调函数，并且每一个都会按照声明的顺序执行。函数调用的第一个参数是产生信号的 widget 组件（即按钮等窗口构件），而 name 则是希望捕获的信号或事件的名称，callback_func 则是事件发生后所调用的回调函数名称，第四个参数 func_data 则是传递给回调函数的参数。回调函数要定义在主程序的前面，它们的一般格式都如下所示。

```
void callback_func( GtkWidget *widget, gpointer func_data );
```

调用下面这个方法，将允许将回调函数与事件的关联断开。

```
void gtk_signal_disconnect( GtkObject *object, gint id );
```

该函数的第二个参数就是上述 gtk_signal_connect（）函数的返回值，即关联标签。第一个参数指向去除关联的对象名称。这样可以做到断开事件与回调函数的关联，使得事件发生后，不会调用相关的回调函数。

（2）Hello World 程序

下面给出较完整的 GTK 应用程序。

```
#include <gtk/gtk.h>
/* 这是一个回调函数。data 参数在本示例中被忽略
 * 后面有更多的回调函数示例*/
void hello( GtkWidget *widget, gpointer   data )
{
   g_print ("Hello World\n");
}
gint delete_event( GtkWidget *widget, GdkEvent  *event, gpointer   data )
{
   /* 如果"delete_event" 信号处理函数返回 FALSE，GTK 会发出 "destroy" 信号
   * 返回 TRUE，不希望关闭窗口
   * 当想弹出“你确定要退出吗?”对话框时它很有用*/
    g_print ("delete event occurred\n");
    /* 改 TRUE 为 FALSE 程序会关闭*/
```

```
    return TRUE;
}
/* 另一个回调函数 */
void destroy( GtkWidget *widget,
              gpointer   data )
{
    gtk_main_quit ();
}
int main( int   argc,
          char *argv[] )
{
    /* GtkWidget 是构件的存储类型 */
    GtkWidget *window;
    GtkWidget *button;

    /*这个函数在所有的 GTK 程序都要调用。参数由命令行中解析出来并且送到该程序中*/
    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 当窗口收到 "delete_event" 信号（这个信号由窗口管理器发出，通常是“关闭”
    * 选项或是标题栏上的关闭按钮发出的），让它调用在前面定义的 delete_event() 函数
     * 传给回调函数的 data 参数值是 NULL，它会被回调函数忽略*/
    g_signal_connect (G_OBJECT (window), "delete_event",
              G_CALLBACK (delete_event), NULL);

    /* 在这里连接 "destroy" 事件到一个信号处理函数
     * 对这个窗口调用 gtk_widget_destroy() 函数或在 "delete_event" 回调函数中返
回 FALSE 值
     * 都会触发这个事件*/
    g_signal_connect (G_OBJECT (window), "destroy",
              G_CALLBACK (destroy), NULL);

    /* 设置窗口边框的宽度*/
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个标签为 "Hello World" 的新按钮*/
    button = gtk_button_new_with_label ("Hello World");

    /* 当按钮收到 "clicked" 信号时会调用 hello() 函数，并将 NULL 传给
     * 它作为参数。hello() 函数在前面已经定义*/
    g_signal_connect (G_OBJECT (button), "clicked",
              G_CALLBACK (hello), NULL);

    /* 当单击按钮时，会通过调用 gtk_widget_destroy(window) 来关闭窗口。
     * "destroy" 信号会从这里或从窗口管理器发出*/
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                  G_CALLBACK (gtk_widget_destroy),
                          window);

    /* 把按钮放入窗口 (一个 gtk 容器) 中*/
    gtk_container_add (GTK_CONTAINER (window), button);
       /* 最后一步是显示新创建的按钮和窗口 */
    gtk_widget_show (button);
```

```
    gtk_widget_show (window);
        /* 所有的 GTK 程序必须有一个 gtk_main() 函数。程序运行停在这里
         * 等待事件（如键盘事件或鼠标事件）的发生*/
    gtk_main ();
    return 0;
}
```

4.5.3 QT 图形开发工具

QT 是一个跨平台的 C++图形用户界面库，由挪威 TrollTech 公司出品。本小节详细介绍 QT 程序设计开发技术。首先介绍 QT 的相关基础知识。

1．QT 基础知识

Trolltech 公司在 1994 年成立，于 1995 年推出 QT 的第一个商业版本，然后 QT 以很快的速度发展。QT 和 X Window 上的 Motif、Openwin、GTK 等图形界面库同类型，但是 QT 最大的特点是具有跨平台特性，QT 支持 Microsoft Windows 95/98，Microsoft Windows NT，MAC，Linux，Solaris，SunOS，HP-UX，Digital UNIX（OSF/1、Tru64），Irix，FreeBSD，BSD/OS，SCO，AIX，OS390，QNX 等。

但真正使得 QT 在自由软件界的众多组件（如 Lesstif、Gtk、EZWGL、Xforms、fltk 等）中脱颖而出的还是基于 QT 的重量级软件 KDE。如果用户使用 C++，对库的稳定性、健壮性要求比较高，并且希望跨平台开发，那么使用 QT 是较好的选择。

QT 现在的版本是 4.3，有商用 License 和 FreeLicense。虽然 QT 的 Free Edition 采用了 GPL 宣言，但是如果用户开发 Windows 上的 QT 软件或者是 UNIX 上的商业软件，还是需要向 Trolltech 公司支付版权费用的。

（1）QT 安装

网络上有很多版本 QT 安装过程的说明文档。下面给出 Linux 平台下和 Windows 平台下详细的安装过程。

- Linux 平台下安装
  - 下载并解压

```
tar zxvf qt-x11-opensource-src-4.2.3.tar.gz
mv qt-x11-opensource-src-4.2.3 /usr/local/qt
```

  - 编译和安装 QT lib

```
./configure
gmake
gmake install   # the destination is /usr/local/Trolltech/QT-4.2.3
```

  - 更新全局变量

```
vi ~/.bash_profile
QTDIR=/usr/local/Trolltech/QT-4.2.3
PATH=$QTDIR/bin:$PATH
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
LIBRARY_PATH=$LD_LIBRARY_PATH
MANPATH=$QTDIR/man:$MANPATH
CPLUS_INCLUDE_PATH=$QTDIR/include:$CPLUS_INCLUDE_PATH
export PATH QTDIR LD_LIBRARY_PATH LIBRARY_PATH MANPATH CPLUS_INCLUDE_PATH
```

  - 创建符号链接

```
rm -f /usr/bin/qmake
ln -s /usr/local/Trolltech/QT-4.2.3/bin/qmake/usr/bin/qmake
```

➢ 编译 QT 程序

```
qmake -project   #gen *.pro
qmake   #gen Makefile
make
```

- Windows 下 GUN 工具和 QT 的安装
  - ➢ execute MinGW-3.2.0-rc-3.exe
  - ➢ execute qt-win-opensource-4.2.3-mingw.exe
  - ➢ 设置 PATH
  - ➢ command line

```
qt/configure -platform win32-g++
mingw32-make
```

➢ 编译 qt 程序

```
qmake -project   #gen *.pro
qmake   #gen Makefile
make
```

关于 MinGW，读者可从下面的地址下载：

http://www.qtcn.org/download/devcpp-4.9.9.2_setup.exe

ftp://503.mygis.org:2200/QT 相关/devcpp-4.9.9.2_setup.exe

当前 QT 最新版本 QT 4.3.0 OpenSource 版下载。

- QT 4.3.0 Windows OpenSource 版下载

http://www.qtcn.org/download/qt-win-opensource-src-4.3.0.zip

http://www.qtcn.org/download/qt-win-opensource-4.3.0-mingw.exe

ftp://ftp.trolltech.com/qt/source/qt-win-opensource-src-4.3.0.zip

ftp://ftp.trolltech.com/qt/source/qt-win-opensource-4.3.0-mingw.exe

- QT 4.3.0 X11 OpenSource 版下载

http://www.qtcn.org/download/qt-x11-opensource-src-4.3.0.tar.gz

ftp://ftp.trolltech.com/qt/source/qt-x11-opensource-src-4.3.0.tar.gz

（2）QT 程序示例

下面介绍一个显示“Hello QT!”的程序的完整源代码，效果如图 4-7 所示。读者可以参考安装目录下的 QT tutorial。



图 4-7　Hello QT!

```
#include <qapplication.h>
#include <qlabel.h>
/*主程序*/
int main( int argc, char **argv )
{
     QApplication app( argc, argv );
      QLabel *hello = new QLabel( "<fontcolor=blue>Hello <i>QT!</i>"
"</font>", 0 );
     app.setMainWidget( hello );
     hello->show();
     return app.exec();
```

```
}
```

QT 具有一系列丰富的部件可以满足很多应用。如果有特殊需要，可以很容易灵活派生出子类，所谓部件就是一个视觉元素，它们合在一起构成用户界面。按钮、菜单、卷轴、消息窗口和应用程序窗口都是部件的例子。QT 的部件并不严格区分控件和容器，所有的部件既可以用做控件，也可以用做容器。通过继承现有的 QT 部件可以很容易地定制自己的部件。当然在极少数情况下，为了特殊的应用，也可能要重头开始创建自己的部件。图 4-8 为 QT 类层次的一个片段。

QT 最常用的基本部件是 QWidget，它的子类或者由这些子类创建自定制类的实例。一个部件可以包含很多子部件。这些子部件显示在父部件的区域内。一个没有父部件的部件（称为顶层部件，如一个窗口）一般会在桌面环境的任务栏上占据一个位置。任何部件都可以使用顶层部件，任何部件都可以成为其他部件的子部件。布局管理器会自动安排子部件在父部件区域中的位置，当然用户也可以手动安排。当父部件无效、隐藏或者被删除时，这些动作也会影响相应的子部件。

标签、消息框、工具提示等可以使用不只一种的颜色、字体和语言，例如，通过使用 HTML 的子集 QT 的文本部件就可以显示多语种的文本。

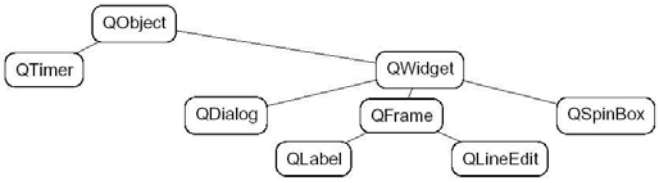


图 4-8　QT 类层次的一个片段

2．QT 消息机制

QT 的内部对象通信使用信号和槽机制。信号和槽机制很容易理解和使用，并且被 QT Designer 支持。

（1）信号和槽

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
#include <qvbox.h>
int main( int argc, char **argv )
{
    QApplication a( argc, argv );
    QVBox box;
    box.resize( 200, 120 );
    QPushButton quit( "Quit", &box );
quit.setFont( QFont( "Times", 18, QFont::Bold ) );
    QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
    //把信号和槽连接起来
    a.setMainWidget( &box );
    box.show();
    return a.exec();
}
```

上面程序实现功能是单击 Quit 按钮程序退出，如图 4-9 所示。

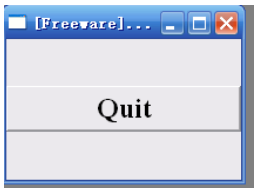


图 4-9　Quit 退出按钮

GUI 程序需要用户相应的动作。例如，当用户单击一个菜单项或者工具栏按钮时，程序就会执行相应的代码。一般情况下，希望任何类型的对象之间可以互相通信。编程人员需要把相应的事件和代码联系起来。以前的工具包使用了一种回调机制，这种机制不是类型安全的，它不够强壮并且不是面向对象的。Trolltech 提出了“信号和槽”的解决方案。它是一种强大的内部对象通信机制，信号和槽非常灵活，完全面向对象并且是使用 C++来实现的。图 4-10 为信号和槽连接的抽象图。

当一个事件发生时，QT 部件会发送一个信号。例如，当一个按钮被按下时，它就可能发送“clicked”信号。编程人员可以创建一个函数（槽）并调用 connect()函数把这个槽和信号联系起来。QT 的信号和槽机制并不要求一个类知道另一个类的信息，因此可以开发出高度可重用的类。信号和槽是类型安全的，当类型不匹配时，它会给出警告。

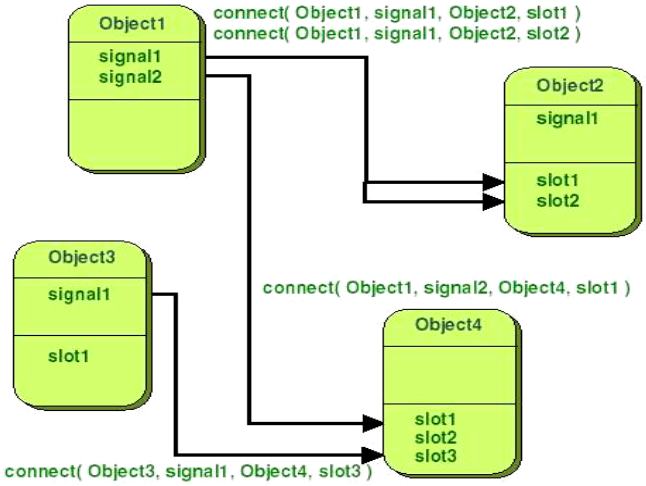


图 4-10　信号和槽连接的抽象图

例如，假设把退出按钮的 clicked()信号和程序的 quit()槽联系在一起，那么当用户单击退出按钮将终止程序。代码可能如下。

```
connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

- 信号

当对象的内部状态发生改变，信号就被发射。只有定义了一个信号的类和它的子类，才能发射这个信号。

例如，一个列表框同时发射 highlighted()和 activated()这两个信号，绝大多数对象也许只对 activated()这个信号感兴趣，但是有时想知道列表框中的哪个条目在当前是高亮的。如果两个不同的类对同一个信号感兴趣，可以把这个信号和这两个对象连接起来。当一个

信号被发射，它所连接的槽会被立即执行，就像一个普通函数调用一样。信号/槽机制完全不依赖于任何一种图形用户界面的事件回路。当所有的槽都返回后 emit 也将返回。如果几个槽被连接到一个信号，当信号被发射时，这些槽就会被按任意顺序一个接一个地执行。

- 槽

当一个和槽连接的信号被发射时，这个槽被调用。槽也是普通的 C++函数，并且可以像它们一样被调用，唯一的特点就是它们可以被信号连接。槽的参数不能含有默认值，并且和信号一样，为了槽的参数而使用自己特定的类型是很不明智的。因为槽就是普通成员函数，但它们也和普通成员函数一样有访问权限。一个槽的访问权限决定了谁可以和它相连。

一个 public slots：区包含了任何信号都可以相连的槽。这对于组件编程来说非常有用，生成了许多对象，它们互相并不知道，把它们的信号和槽连接起来，这样信息就可以正确地传递，并且就像一个铁路模型，把它打开然后让它跑起来。

一个 protected slots：区包含了之后这个类和它的子类的信号才能连接的槽。这就是说这些槽只是类的实现的一部分，而不是它和外界的接口。

一个 private slots：区包含了之后这个类本身的信号可以连接的槽。这就是说它和这个类是非常紧密的，甚至它的子类都没有获得连接权利这样的信任。

也可以把槽定义为虚的，这在实践中也是非常有用的。

信号和槽的机制非常有效，但是它不像“真正的”回调那样快。信号和槽稍微有些慢，这是因为它们所提供的灵活性不同，尽管在实际应用中这些不同可以被忽略。通常发射一个和槽相连的信号，大约只比直接调用那些非虚函数调用的接收器慢十倍。这是定位连接对象所需的开销，可以安全地重复所有的连接（例如在发射期间检查并发接收器是否被破坏），并且可以按一般的方式安排任何参数。

信号和槽之间的联系可以在程序运行期间动态地添加和删除。

信号和槽的实现扩展了 C++语法，并且充分利用了 C++面向对象的性质。信号和槽是类型安全的，可以重载或者重新实现，而且它们在类中可以定义为公有的、保护的或者私有的。

（2）QT 窗口设计

QWidget 提供了几个处理窗口几何结构的函数，有不包含窗口的框架和其他一些包括窗口的框架。

包括窗口的框架有 x()、y()、frameGeometry()、pos()和 move()。

不包括窗口的框架有 geometry()、width()、height()、rect()和 size()。

这种区别仅仅对于被装饰的顶层窗口部件有效。对于所有的子窗口部件，框架的几何结构和这个窗口部件的客户几何结构相同。图 4-11 显示了所使用的绝大多数函数。

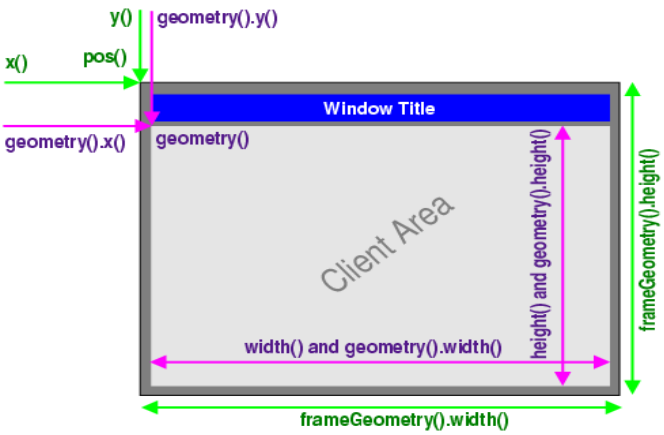


图 4-11　处理窗口几何结构的函数

一个工作区在 show()之后调用 setGeometry()。这样做缺点如下：窗口部件会在 1ms 中出现在一个错误的位置（结果是闪烁）和通常在 1s 之后窗口管理器会正确地得到它。一个安全的方式是存储 pos()和 size()，并且在 show()之前调用 resize()和 move()来恢复几何结构，如下面这个例子所示。

```
MyWidget* widget = new MyWidget
...
QPoint p = widget->pos();   // 存储位置
QSize s = widget->size();   // 存储大小
...
widget = new MyWidget;
widget->resize( s );        // 恢复大小
widget->move( p );          // 恢复位置
widget->show();             // 显示窗口部件
```

这种方法可以在 Microsoft-Windows 和绝大多数现存的 X11 窗口管理器上工作。参考程序（见 QT 官方教程）如下。

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
class MyWidget : public QWidget
{
public:
    MyWidget( QWidget *parent=0, const char *name=0 );
};
MyWidget::MyWidget( QWidget *parent, const char *name )
       : QWidget( parent, name )
{
    setMinimumSize( 200, 120 );
    setMaximumSize( 200, 120 );
    QPushButton *quit = new QPushButton( "Quit", this, "quit" );
    quit->setGeometry( 62, 40, 75, 30 );
    quit->setFont( QFont( "Times", 18, QFont::Bold ) );
    connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}
int main( int argc, char **argv )
{
```

```
    QApplication a( argc, argv );
    MyWidget w;
    w.setGeometry( 100, 100, 200, 120 );
    a.setMainWidget( &w );
    w.show();
    return a.exec();
}
```

程序执行结果如图 4-12 所示。

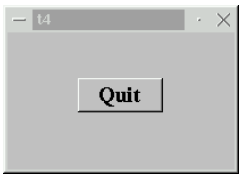


图 4-12　程序执行结果

4.6　本章总结

本章重点介绍了一些常用 Linux 开发工具的内容，包括 GCC 编译器、gdb 调试器、Linux 调试工具以及 GUI\GTK\QT 图形开发工具。学习 Linux 开发工具是进行 Linux 程序开发的一个重要先决条件，希望读者多加熟悉和使用。

第 2 篇

Linux 驱动程序开发与实例

- 第 5 章　Linux 设备驱动基础
- 第 6 章　网卡驱动程序开发
- 第 7 章　显卡驱动程序开发
- 第 8 章　声卡驱动程序开发
- 第 9 章　USB 驱动程序开发
- 第 10 章　闪存 Flash 驱动程序开发

第 5 章

Linux 设备驱动基础

在介绍具体的设备驱动开发技术之前，首先介绍一些关于 Linux 设备驱动的基础知识。

Linux 中的驱动设计是 Linux 开发中十分重要的部分，它要求开发者不仅熟悉 Linux 的内核机制、驱动程序与用户级应用程序的接口关系、系统对设备的并发操作等，而且还要求开发人员非常熟悉所开发硬件的工作原理。

驱动程序是应用程序与硬件之间的一个中间软件层，驱动程序为应用程序展现硬件的所有功能，但对于硬件使用的权限和限制由应用程序层控制。当然，如果驱动程序的设计是跟所开发的项目相关的，这时就可能在驱动层加入一些与应用相关的设计考虑，采用这种方式主要基于以下考虑。

- 驱动层的效率比应用层高；
- 为了项目的需要可能只需要强化或优化硬件的某个功能，而弱化或关闭其他一些功能。

到底需要展现硬件的哪些功能由开发者根据需要而定。另外，驱动程序有时会被多个进程同时使用，这时要考虑如何处理并发性的问题，这就需要调用一些内核的函数并使用互斥量和锁等机制。

驱动程序主要需要考虑以下 3 个方面。

- 提供尽量多的选项给用户；
- 提高驱动程序的速度和效率；
- 尽量使驱动程序简单，使之易于维护。

5.1 驱动程序基本概念

Linux 的驱动开发调试有两种方法：一种是直接编译到内核，再运行新的内核来测试；另一种是编译为模块的形式，单独加载运行调试。第一种方法效率较低，但在某些场合是唯一的方法。模块方式调试效率很高，可以使用 insmod 工具将编译的模块直接插入内核，如果出现故障，可以使用 rmmod 从内核中卸载模块，从而不需要重新启动内核，使驱动调试效率大大提高。

5.1.1 驱动程序与应用程序的区别

应用程序一般有一个 main 函数，从头到尾执行一个任务；驱动程序却不同，它没有 main 函数，而是通过使用宏 module_init（初始化函数名）将初始化函数加入到内核全局初

始化函数列表中，然后在内核初始化时执行驱动的初始化函数，从而完成驱动的初始化和注册，之后驱动便停止，一直等待被应用软件调用。驱动程序中有一个宏 moudule_exit（退出处理函数名），注册退出处理函数用于在驱动退出时被调用。

另外，应用程序可以与 GLIBC 库链接，因此可以包含标准的头文件，比如<stdio.h>、<stdlib.h>等库文件。然而，在驱动程序中是不能使用标准 C 库的，因此不能调用所有的 C 库函数，比如，输出打印函数只能使用内核的 printk()函数，包含的头文件只能是内核的头文件，比如<linux/module.h>。

5.1.2　内核版本与编译器的版本依赖

当模块与内核链接时，insmod 会检查模块与当前内核版本是否匹配，每个模块都定义了版本符号__module_kernel_version，这个符号位于模块文件的 ELF 头的.modinfo 段中。只要在模块中包含了<linux/module.h>库文件，编译器就会自动定义这个符号。

每个内核版本都需要特定版本的编译器的支持，高版本的编译器并不适合低版本的内核。Linux 2.4 版本的 insmod 命令装载模块时，首先会从/lib/modules 目录和内核相关的子目录中查找模块文件，如果需要从当前目录装载，则使用 insmod module.o。

5.2　设备驱动模块概述

本节将简单叙述驱动模块的基本概念、初始化以及加载方法。

5.2.1　模块的基本概念

可装载模块 LKM（Loadabel Kernel Module）：在 Linux 系统启动以后，可以通过一些命令（如 insmod、rmmod、modprobe）来加载一些目标文件到内核中，使之成为系统的一部分，这些被加载的目标文件就被称为可加载模块 LKM。模块从 1995 年（Linux 1.2）开始引入，至 2000 年左右开始变得成熟。

1．LKM 模块功能

LKM 主要用于以下情况。

- 提供设备驱动，这是可装载模块的主要用途，当引入新的设备驱动时只需要加载相应的驱动模块，用于硬件设备的管理。
- 文件系统模块，Linux 支持很多文件系统，不同的文件系统模块管理不同的文件系统。
- 系统调用，用于替换原有系统调用或增加新的系统调用，较少使用。
- 可执行程序解释器，通过引入可执行程序解释器，Linux 系统能执行非 ELF 类型的可执行程序。

2．模块文件

模块也是一种 obj 文件。从内核 2.6 开始，内核把模块文件的扩展名从.o 改为.ko，以便与普通的目标文件相区别，而且在.ko 文件中增加了.modinfo section，故可以通过 modinfo mod.ko 来查看某个模块的基本信息，类似如下。

```
filename：hello.ko
```

```
license：GPL v2
depends：
vermagic：2.6.15.5 686 gcc-4.1
```

模块的编写方式与普通的编程相似，但是它引用的外部符号必须是内核所提供的，即通过查看 cat /proc/kallsyms 所看到的符号，这些符号包括系统调用函数，如 sys_fork、syscall_exit 等。

在当前的 2.6 版本中，支持的许可认证有如下五个。

- GPL
- GPL v2
- GPL and additional rights
- Dual BSD/GPL
- Dual MPL/GPL

从 Linux 2.4 以后，Linux 系统引入了模块认证机制。系统会在提供服务的函数中加入前缀或后缀来判断，引出符号时需要使用EXPORT_SYMBOL或EXPORT_SYMBOL_GPL；如果模块是私有的，却使用了 GPL 所提供的服务，那么模块将不能被加载到系统中，系统会提示 unresolved symbol。

在 2.6 内核以前，系统使用 genksyms 命令来产生.ver 文档，这些文档用于采用宏定义的方式来重命名内核符号，目的是使模块编译时的内核版本与其加载时的内核版本相同，这样可以保证系统的模块与 base-kernel 的一致。从 2.6 内核起，因为在目标文件中引入了 modinfo section，所以使得在加载模块时能够使用 modinfo section 所提供的信息进行内核版本的校验，达到一致性的目标，所以在 fc5 上已经没有 genksyms 命令了。

5.2.2 模块的初始化和退出

从 kernel-2.3.13 开始，内核引入了新的模块机制，这样用户写 module 时就没有必要把初始化函数和清除函数写做 init_module（返回类型必须是 int）和 cleanup_module（返回类型必须是 void），而可以使用两个宏 module_init(init_func)和 module_exit(exit_func)来设置初始化函数和退出函数。

用户仍然可以使用原来在 kernel-2.3.13 以前的函数定义方式。通过查看实际的 include/Linux /init.h 源代码可以发现，对于动态加载的模块，其 module_init 和 module_exit 定义为：

```
#define module_init(initfn) \
static inline initcall_t __inittest(void) \
{ return initfn; } \
int init_module(void) __attribute__((alias(#initfn)));
#define module_exit(exitfn) \
static inline exitcall_t __exittest(void) \
{ return exitfn; } \
void cleanup_module(void) __attribute__((alias(#exitfn)));
```

系统把 init/cleanup_module 函数声明为函数 initfn 或 exitfn 的别名函数，即调用 initfn 或 exitfn 时，相当于调用 init_module 或 cleanup_moudle 函数。如果没有初始化函数，那么可以不定义 init_module 函数或不使用 moudle_inti 来声明初始化函数。如果希望 module 是

unloadabel 的，即该模块被装载之后就不能被 unload，那么还可以不定义 cleanup_module 函数或不使用 module_exit 来声明清除函数。此时，当 cat /proc/modules 时，会看到该模块的信息中有[permanent]的输出，这就表明该模块是不可卸载的，即永久存在于系统，除非系统重启，内核重新运行。

下面介绍初始化和退出示例（__init 和__exit）。

```
/*
 *  hello.c
 */
#include <linux /module.h>  /* 模块必需的 */
#include <linux /kernel.h>  /* 内核信息需要 */
#include <linux /init.h>    /* 宏定义需要 */
static int hello3_data __initdata = 3;
static int __init hello_3_init(void)
{
            printk(KERN_INFO "Hello, world %d\n", hello3_data);
            return 0;
}
static void __exit hello_3_exit(void)
{
            printk(KERN_INFO "Goodbye, world 3\n");
}
module_init(hello_3_init);
module_exit(hello_3_exit);
```

在模块加载和卸载时分别调用前缀有__init 和__exit 的函数。

通过 include/Linux /init.h，能够看到如下的定义。

```
#define __init __attribute__ ((__section__ (".init.text")))
#define __initdata __attribute__ ((__section__ (".init.data")))
```

如果某个函数带有__init 或变量带有__initdata，那么它们在链接阶段会被放置在.init.text 或.init.data 区域。内核启动时会调用 init 函数来执行初始化函数，之后它又会调用 free_initmem 函数，这个函数会释放 init section，释放的地址空间为链接时从__init_begin 到__init_end 之间的内核空间。

__exit 和__exitdata 的作用与__init 差不多。

moudle_init 和 module_exit 对于被静态编译进内核的 module，它们都使用了__init 或__initdata 声明，其 module_init 定义为：

```
typedef int (*initcall_t)(void);
#define __define_initcall(level,fn) \
static initcall_t __initcall_##fn __attribute_used__ \
__attribute__((__section__(".initcall" level ".init"))) = fn
#define device_initcall(fn) __define_initcall("6",fn)
#define __initcall(fn) device_initcall(fn)
#define module_init(x) __initcall(x);
```

__init 函数在区段.initcall.init 中保存了一份函数指针，在初始化时内核会通过这些函数指针调用这些__init 函数指针，initcall.init 区段又分成.initcall1.init 、initcall2.init、.initcall3.init、.initcall4.init、.initcall5.init、.initcall6.init、initcall7.init 7 个子区段。参见<linux/init.h>。

```
#define core_initcall(fn)           __define_initcall("1",fn)
#define postcore_initcall(fn)       __define_initcall("2",fn)
#define arch_initcall(fn)           __define_initcall("3",fn)
#define subsys_initcall(fn)         __define_initcall("4",fn)
#define fs_initcall(fn)             __define_initcall("5",fn)
#define device_initcall(fn)         __define_initcall("6",fn)
#define late_initcall(fn)           __define_initcall("7",fn)
```

在内核中，不同的 init 函数被放在不同的子区段中，因此也就决定了它们的调用顺序。这样也就解决了一些 init 函数之间必须保证一定的调用顺序的问题。

在系统启动初始化过程中对这些初始化函数的调用都是通过这些函数指针间接调用的。而在函数 free_initmem 中进行内存释放时，不仅会释放.init 区中的函数代码或数据，而且还会释放这些函数指针区域。module_exit 的作用与之类似。对于 loadable 的 module（即系统启动后可以通过 insmod 或 modprobe 来装载的模块），虽然使用了__init，它也会被放置在.init 区，但是没有相应的内存释放函数来释放这个区的空间，所以它不会被释放，但是编程时还是应该加入这个属性。

给模块传递参数有时需要设置模块中一些变化的初值，此时需要使用宏 MODULE_PARM 说明参数的名字和类型。该宏支持的类型有 b（bool 类型）、h（短整形）、i（整数）、l（长整数）、s（字符串类型）。同时还可以使用数组，如果类型前面包括一个整数，那么表示该数组的最大长度，如果是通过“-”字符分开的两个整数，那么表示数组的最小和最大长度。如以下的变量：

```
int myint = 3; MODULE_PARM(myint, "i");
char *mystr; MODULE_PARM(mystr, "s");
```

对于数组的示例如下：

```
int myshortArray[4];
```

如果为 MODULE_PARM(myintArray, "4i")，表示数组的最大长度为 4，如果为 MODULE_PARM(myintArray, "2-4i")，表示最小长度为 2，最大长度为 4。启动系统时可以给内核传递参数。当使用 insmod 来加载模块时，可以给模块传递参数，模块文件中有 modinfo section，这个 section 包括模块使用宏 MODULE_PARM 所定义的模块参数，insmod 命令使用这些信息给模块参数赋值。

5.2.3 Linux 内核模块加载

在 Linux 操作系统中，内核模块加载方式一般分为动态加载和静态加载。

1．编译时静态加载

在执行 make menuconfig 命令进行内核配置时，在窗口中可以选择是否编译入内核，还是放入/lib/modules/下相应内核版本目录中，还是不选。Linux 设备一般分为字符设备、块设备和网络设备，每种设备在内核源代码目录树 drivers/下都有对应的目录，其加载方法可以参照如下。

（1）将 hello.c 源程序放入到内核源码 drivers/char/下；

（2）修改 drivers/char/Kconfig 文件，具体修改如下：

```
config CONFIG_HELLO
```

```
tristate "hello test"
help
        help description
```

按照打开文件中的格式添加即可；在文件的适当位置（这个位置随便都可以，但这个位置决定其在 make menuconfig 窗口中所在位置）加入上述代码。

使用 tristate 来定义一个宏,表示此驱动可以直接编译至内核（用*选择）,也可以编译至/lib/modules/下（用 M 选择）,或者不编译（不选）。

（3）修改 drivers/char/Makefile 文件，在适当位置加入下面一行代码。

```
obj-$(CONFIG_HELLO)  +=  hello.o
```

或者在 obj-y 一行中加入 hello.o,如 obj-y += hello.o，后面不变。经过以上的设置就可以在执行 make menuconfig 命令后的窗口中的“character devices--->”中进行选择配置了。选择后重新编译就可以了。当然也可以直接修改.configure 文件。

2．动态加载

动态加载是在系统启动后将驱动模块加载到内核中。

在 2.4 内核中，加载驱动命令为 insmod，删除模块为 rmmod。

在 2.6 以上内核中，除了 insmod 与 rmmod 外，加载命令还有 modprobe。

对于 Linux 2.4 和 Linux 2.6，虽然它们都使用 insmod 来加载模块，但是它们的处理方式是不同的。对于 Linux 2.4，insmod 扮演 linker 的作用，它先对模块文件进行符号重定位，然后再把经过重定位的模块传递给内核。对于 Linux 2.6，insmod 进行版本认证和参数检查后，直接把模块传递给内核，不进行符号重定位。

（1）modprobe 工作原理

在 Linux 2.2 之前，内核使用守护进程 kerneld 根据需要自动加载模块，如当系统打开一个 DOS 文件时，如果当前系统没有加载 msdoc.ko 模块，那么系统自动加载该模块，然后再进行文件操作。从 Linux 2.2 开始，内核使用 modprobe 命令进行自动加载，该命令最终调用 insmod 进行加载。系统使用/proc/sys/kernel/modprobe 来记录该命令所存在的位置。

当使用该命令加载模块时，不需要指定模块的绝对路径。当该命令接收到参数时，它会在系统文件/etc/modules.conf 和/etc/modprobe.d/目录下（这些文件定义模块的别名，如 alias eth0 net0 表示模块 eth0 的别名是模块 net0）寻找是否存在这个模块名。

如果找到别名，那么该命令会通过/lib/modules/<kernle-version>/modules.dep（该文件使用绝对路径来指明模块的位置和其依赖模块的位置，所以可以调用 insmod 来进行加载）来查看该模块依赖于哪些模块，然后使用 insmod 命令来依次加载这些模块。如果没有找到别名，那么直接在/lib/modules/<kernle-version>目录下以传入的模块参数名来寻找模块，如果找到，那么使用 insmod 命令进行加载，否则打印出“FATAL：Module XX not found.”的错误信息。

使用 modprobe 加载或卸载模块可以根据模块之间的依赖关系来自动地加载或卸载所依赖的模块。但是，编译并安装了自己的模块后（如果需要安装 target，那么需要在模块的 Makefile 中加入 make -C /lib/module/$(shell uname –r)/build M=$(PWD) modules_install），仍然不能使用该命令来加载模块。这是因为当安装了自己的模块后（该模块会被安装到/lib/module/<kernel-version>/extra 目录下），并没有更新 modules.dep 文件。此时，仅仅需

要在命令下运行 depmod 命令（不需要在任何特殊的目录），modules.dep 文件就会得到更新，此时模块会被加入到该文件的顶端。例如/lib/modules/2.6.15-1.2054_FC5/extra/jixu.ko：，在冒号的后面是空的，表示它不依赖于其他模块；如果有内容，表示它依赖于其他模块，这样 modprobe 就可以根据这个依赖关系来装载模块了，如/lib/modules/2.6.16/extra/hello-test.ko：/lib/modules/2.6.16/extra/hello.ko 表示如果要加载 hello-test.ko 模块，那么必须先加载 hello.ko 模块。

（2）THIS_MODULE 宏

在 Linux /module.h 中定义有一个宏 THIS_MODULE，该宏实际定义为__this_module 指针。最终该模块在被 insmod 时，会为之分配一个 module 的结构，而该指针就指向该分配的结构体。

5.3 Linux 设备驱动结构分析

本节对 Linux 设备驱动结构做详细分析。

5.3.1 内核和用户接口

Linux 系统应用程序通过系统调用和内核通信，用户程序通过系统调用来调用内核提供的服务，其内核的行为对于应用程序是不可知的。实际上用户程序并不直接和内核通信，而是通过 libc（标准 C 库）和内核通信的。

由于这种内核体系，Linux 的进程以系统调用接口为界，分为用户态的执行和内核态的执行（内核进程除外）。系统框图如图 5-1 所示。

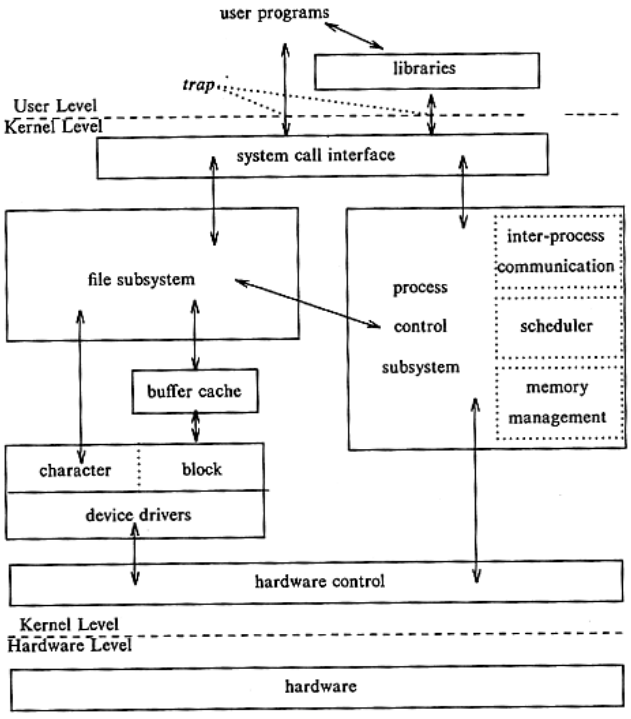


图 5-1　系统框图

系统调用发生在用户进程通过调用特定函数以请求内核提供服务时，用户进程被暂时挂起，内核检验用户请求，尝试执行，并把结果反馈给用户进程，接着用户进程继续运行。系统调用负责保护对内核所管理的资源的访问。系统调用分为几个大类，主要有处理 I/O 请求（open、close、read、write、poll 等）、进程（fork、execve、kill 等）、时间（time、settimeofday 等）以及内存（mmap、brk 等）的系统调用。几乎所有的系统调用都可以归入到这几类。系统调用必须返回 int 值，并且也只能返回 int 值。

5.3.2　inode 节点

inode 是索引节点。每个存储设备（存储设备是硬盘、软盘、U 盘等）或存储设备的分区被格式化为文件系统后，应该有两部分，一部分是 inode，另一部分是 Block。Block 用来存储数据，而 inode 用来存储这些数据的信息，这些信息包括文件大小、属主、归属的用户组、读写权限等。inode 为每个文件进行信息索引，所以就有了 inode 的数值。操作系统根据指令，能通过 inode 值最快找到相对应的文件。比如一本书，存储设备或分区就相当于这本书，Block 相当于书中的每一页，inode 就相当于这本书前面的目录，一本书有很多内容，如果想查找某部分的内容，可以先查目录，通过目录能最快地找到想要看的内容。

当用 ls 查看某个目录或文件时，如果加上-i 参数，就可以看到 inode 节点了，比如前面所说的例子。

```
[root@localhost ]# ls -li test.sh
3055708 -rwxr-xr-x   1 root   root   14627 Dec 12 12:36 test.exe
```

test.sh 的 inode 值是 3055708；查看一个文件或目录的 inode，要通过 ls 命令的-i 参数。下面通过图 5-2 来解释节点功能。

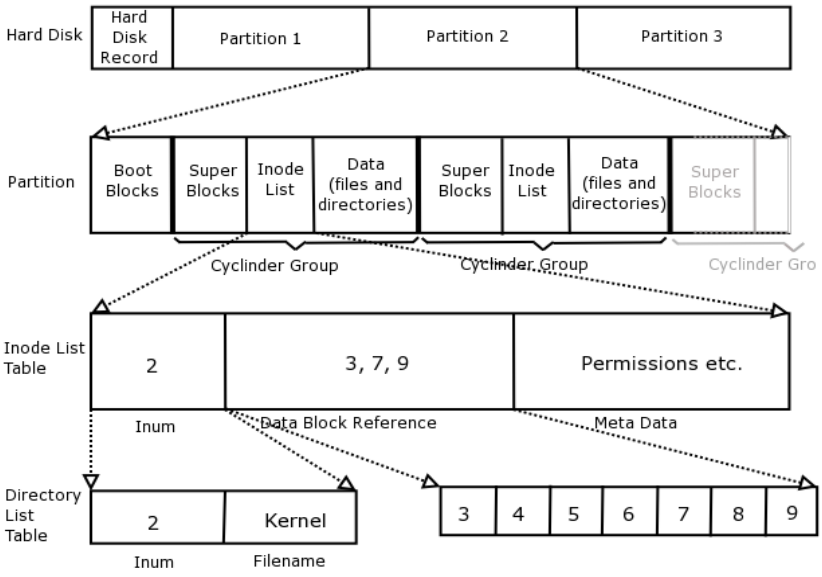


图 5-2　节点例图

节点指向直接的数据块和数据块指向的数据块即 indirect inode、double indirect 等，类似于指针、指向指针的指针等。

这里以 foo.Txt 文件为例，文件 foo.Txt 指向文件节点，通过直接块和间接块指向具体的数据块，如图 5-3 所示。

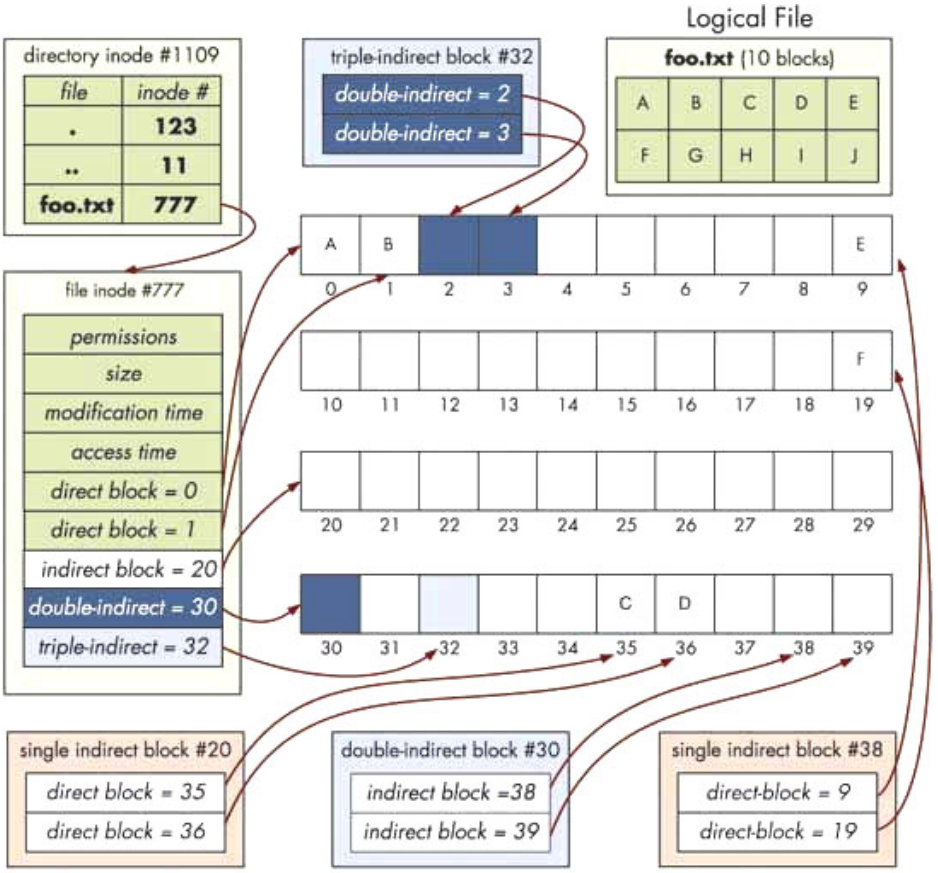


图 5-3　文件 foo.Txt 指向文件节点

当文件具有相同的 inode 时，是具有相同内容的文件，即一个 inode 可以对应多个文件名，但不能跨分区，因为 inode 相对于分区才有意义，这样可以保护误删除。

Linux/UNIX 文件系统中有所谓的连接（link），可以将其视为文件的别名。连接又可分为硬连接（hard link）与软连接（symbolic link）两种。硬连接的意思是一个文件可以有多个名称，硬连接是存在于同一个文件系统中。软件接是符号连接或快捷方式，其内容是一个文件的物理路径，因此可以跨分区使用，软连接的方式则是产生一个特殊的文档，文档的内容是指向另一个文档的位置，因而软连接可以跨越不同的文件系统。无论是硬连接还是软连接都不会将原本的文件复制一份，所以只会占用非常少量的磁盘空间。

5.3.3　File 结构

File 结构代表了设备的一个实例句柄。一个设备同时被多个进程打开时，具有多个 file 句柄，而 inode 只有一个。设备 1，2，3…分别代表多个不同的设备，使用一个驱动程序。用户进程 1 和进程 2 同时打开了设备 1，进程 1 和进程 2 的 inode 节点是一样的，都代表了设备 1。进程 1 和进程 2 的 struct file 结构是不同的，该结构与进程相关，file 的个数表

示同时共享的设备数。

File 结构的内容如下：

```
struct file {
struct file              *f_next，  **f_pprev;
struct dentry             *f_dentry;
struct file_operations     *f_op;
mode_t                   f_mode;
loff_t                   f_pos;
unsigned int              f_count，  f_flags;
unsigned long             f_reada，  f_ramax，  f_raend，  f_ralen，  f_rawin;
struct fown_struct         f_owner;
unsigned long             f_version;
void                     *private_data;
};
```

f_next 会指到下一个 file 结构，而 f_pprev 则会指到上一个 file 结构地址的地址，f_dentry 会记录其 inode 的 dentry 地址，f_mode 为文件调用的种类，f_pos 则是目前文件的 offset，每次读写都从 offset 记录的位置开始读写，f_count 是此 file 结构的 reference cout，f_flags 则是开启此文件的模式，f_reada、f_ramax、f_raend、f_ralen、f_rawin 则是控制 read ahead 的参数，f_owner 记录了要接收 SIGIO 和 SIGURG 的行程 ID 或行程群组 ID，private_data 则是 tty driver 所使用的字段。最后来介绍 f_op 字段。这个字段记录了一组函数是专门用来操作文件的。

1．常见的文件操作函数

（1）llseek（file, offset, where）；

写程序会调用 lseek()系统调用，设定从文件哪个位置开始读写，这个函数可以不用提供，因为系统已经有一个写好的，但是系统提供的 llseek()没有办法将 where 设为 SEEK_END，因为系统不知道文件长度是多少。如果不提供 llseek()，那系统会直接使用它已经有的 llseek()。llseek()必须要将 file->offset 的值更新。

（2）read（file, buf, buflen, poffset）；

当读取一个文件时，最终会调用 read()函数来读取文件内容。这些参数 VFS 会准备好，至于 poffset 则是 offset 的指标，要告诉 read()从哪里开始读，读完之后必须更新 poffset 的内容。在这里 buf 是一个地址，而且是一个位于 user space 的地址。

（3）write（file, buf, buflen, poffset）；

write()的动作跟 read()是相反的，参数也都一样，buf 依然是位于 user space 的地址。

（4）readdir（file, dirent, filldir）；

这是用来读取目录的下一个 direntry 的，其中 file 是 file 结构地址，dirent 则是一个 readdir_callback 结构，这个结构里包含了使用者调用 readdir()系统调用时所传过去的 dirent 结构地址，这个函数在 VFS 中已经提供，该函数其实是增加了 kernel 在读取 dirent 方面的弹性。当文件系统的 readdir()被调用时，在它把下一个 dirent 取出来之后，应该要调用 filldir()，把所需的资料写到 user space 的 dirent 结构里，也许还会多做些处理。

（5）poll（file, poll_table）；

之前的 Kernel 版本本来是在 file_operations 结构里有 select()函数而不是 poll()函数的。这并不代表 Linux 不提供 select()系统调用，相反地，Linux 仍然提供 select()系统调用，只

不过 select()系统调用 implement 的方式是使用 poll()函数。

（6）ioctl（inode, file, cmd, arg）；

ioctl()这个函数其实有很大的用途，尤其它可以作为 user space 的程序对 Kernel 的一个通信管道。那 ioctl()什么时候被调用呢？平常写程序时偶尔会用到 ioctl()系统调用直接控制文件或 device，ioctl()系统调用最后就是把命令交给文件的 f_op->ioctl()来执行。f_op->ioctl()要做的事很简单，只要根据 cmd 的值做出适当的行为，并传回值即可。但是，ioctl()系统调用其实是分几个步骤的。第一，系统有几个内定的 command 自己可以处理，在这种情形下，它不会调用 f_op->ioctl()来处理。

（7）mmap（file, vmarea）；

这个函数用来将文件的部分内容映像到内存中，file 是指要被映像的文件，而 vmarea 则是用来描述映像到内存的位置。

（8）open（inode, file）；

当调用 open()系统调用打开文件时，open()会把所有的事都做好，最后则会调用 f_op->open()看文件系统是否要做些什么事，一般来讲，VFS 已经把事做好了，所以很多系统事实上根本不提供这个函数。

（9）flush（file）；

这个函数是在调用 close()系统调用关闭文件时所调用的。只要调用 close()系统调用，那 close()就会调用 flush()，不管那时 f_count 的值是否为 0。

（10）release（inode, file）；

这个函数也是在 close()系统调用中使用的，当然不仅在 close()中使用，在别的地方也使用。基本上这个函数的定位跟 open()很像，不过当对一个文件调用 close()时，只有当 f_count 的值归 0 时，VFS 才会调用这个函数做处理。至于 f_count 的值则是不用在 open()和 release()中控制，VFS 已经在 fget()和 fput()中增减 f_count 了。

（11）fsync（file, dentry）；

fsync()这个函数主要是由 buffer cache 所使用，它与 File 文件的资料一起写到 disk 上。事实上，Linux 里有两个系统调用 fsync()和 fdatasync()，都是调用 f_op->fsync()。它们几乎一模一样，差别在于 fsync()调用 f_op->fsync()之前会使用 semaphore 将 f_op->fsync()设成 critical section，而 fdatasync()则是直接调用 f_op->fsync()而不设 semaphore。

（12）fasync（fd, file, on）；

当调用 fcntl()系统调用，并使用 F_SETFL 命令来设定文件的参数时，VFS 就会调用 fasync()这个函数，而当读写文件的动作完成时，会收到 SIGIO 信息。

（13）check_media_change（dev）；

这个函数只对可以使用可移动的磁盘块设备有效，如 CDROM、floopy disk 等。从字面上大概可以知道，该函数用来检查 disk 是否换过。

（14）revalidate（dev）；

当用户执行 mount 要挂上一个文件系统时，mount 会先调用 check_disk_change()，如果文件系统所属的设备提供这个函数，那 check_disk_change()会先调用 f_op->check_media_change()来检查其中的磁盘是否换过，如果是，则调用 invalidate_inodes()和

invalidate_buffers()将与原本磁盘有关的 buffer 或 inode 都设为无效，如果文件系统所属的设备提供 revalidate()，就调用 revalidate()将此设备的资料记录好。

（15）lock（file, cmd, file_lock）；

在 Linux 中，可通过调用 fcntl()对一个文件使用 lock。如果调用 fcntl()时，cmd 的参数设为 F_GETLK、F_SETLK 或 F_SETLKW，那系统会间接调用 f_op->lock。

2．Linux 驱动开发中容易混淆的概念

（1）主、次设备号

在 Linux 系统中，设备的描述是通过主、次设备号描述的。不同的设备主次设备号的组合必须唯一，否则视为同一个设备。具有相同主设备号不同次设备号的设备之间共用同一个驱动程序。

（2）设备驱动程序

一个设备驱动程序可以驱动多个相同或不同的设备，通常对具有相同属性或单独设备编写相应的驱动程序。

（3）inode 节点

inode 驱动程序的许多函数体的入口参数带有 struct inode 结构。一个 inode 代表一个具体的设备。

5.4　常用接口函数介绍

1．open 函数

open 函数提供给驱动程序初始化设备的能力，从而为以后的设备操作做好准备，此外 open 操作一般还会递增使用计数，用以防止文件关闭前模块被卸载出内核。在大多数驱动程序中 open 函数应完成如下工作。

- 递增使用计数；
- 检查特定设备错误；
- 如果设备是首次打开，则对其进行初始化；
- 识别次设备号，如有必要修改 f_op 指针；
- 分配并填写 filp->private_data 中的数据。

```
static int demo_open(struct inode *inode, struct file *file)
{
    sprintf(drv_buf,"device open sucess!\n");
    printk("device open sucess!\n");
    return 0;
}
```

2．release 函数

与 open 函数相反，release 函数应完成如下功能。

- 释放由 open 分配的 filp->private_data 中的所有内容；
- 在最后一次关闭操作时关闭设备；
- 使用 MOD_DEC_USE_COUNT，计数减 1。

```
static int  demo_release(struct inode *inode, struct file *filp)
    {
```

```
    MOD_DEC_USE_COUNT;
    printk("device release\n");
    return 0;
    }
```

3．read 和 write 函数

read 函数完成将数据从内核复制到应用程序空间，write 函数相反，将数据从应用程序空间复制到内核。对于这两个函数，参数 filp 是文件指针，count 是请求传输数据的长度，buffer 是用户空间的数据缓冲区，ppos 是文件中进行操作的偏移量，类型为 64 位数。由于用户空间和内核空间的内存映射方式完全不同，所以不能使用像 memcpy 之类的函数，必须使用如下函数。

```
unsigned long copy_to_user (void *to,const void *from,unsigned long count);
unsigned long copy_from_user (void *to,const void *from,unsigned long count);
```

read 函数的返回值有以下特点：

- 返回值等于传递给 read 系统调用的 count 参数，表明请求的数据传输成功；
- 返回值大于 0，但小于传递给 read 系统调用的 count 参数，表明部分数据传输成功，根据设备的不同，导致这个问题的原因也不同，一般采取再次读取的方法；
- 返回值=0，表示到达文件的末尾；
- 返回值为负数，表示出现错误，并且指明是何种错误；
- 在阻塞型 io 中，read 调用会出现阻塞。

write 函数的返回值有以下特点：

- 返回值等于传递给 write 系统调用的 count 参数，表明请求的数据传输成功；
- 返回值大于 0，但小于传递给 write 系统调用的 count 参数，表明部分数据传输成功，根据设备的不同，导致这个问题的原因也不同，一般采取再次读取的方法；
- 返回值=0，表示没有写入任何数据。标准库在调用 write 时，出现这种情况会重复调用 write；
- 返回值为负数，表示出现错误，并且指明是何种错误。错误号的定义参见<linux/errno.h>；
- 在阻塞型 io 中，write 调用会出现阻塞。

```
    static ssize_t  demo_read(struct file *filp, char *buffer, size_t count,
loff_t *ppos)
    {
        if(count > MAX_BUF_LEN)
            count=MAX_BUF_LEN;
        copy_to_user(buffer, drv_buf,count);
        printk("user read data from driver\n");
        return count;
    }
    static ssize_t  demo_write(struct file *filp,const char *buffer, size_t
count)
    {
        if(count > MAX_BUF_LEN)count = MAX_BUF_LEN;
        copy_from_user(drv_buf , buffer, count);
        WRI_LENGTH = count;
        printk("user write data to driver\n");
        do_write();
```

```
    return count;
}
```

4．ioctl 函数

ioctl 函数主要用于对设备进行读写之外的其他控制，比如配置设备、进入或退出某种操作模式，这些操作一般都无法通过 read/write 文件操作来完成。

用户空间的 ioctl 函数的原型为：

```
int ioctl(int fd,int cmd )
```

其中的 fd 代表可变数目的参数表，实际中是一个可选参数，一般定义为：

```
int ioctl(int fd,int cmd,*argp)
```

驱动程序中定义的 ioctl 方法原型为：

```
int (*ioctl)(stuct inode *inode,struct file *file,unsigned int cmd,unsigned long arg)
```

inode 和 filp 两个指针对应应用程序传递的文件描述符 fd，cmd 不被修改地传递给驱动程序，可选的参数 arg 无论用户应用程序使用的是指针还是其他类型值，都以 unsigned long 的形式传递给驱动。由于为了防止向不该控制的设备发出正确的命令，Linux 驱动的 ioctl 方法中的 cmd 参数推荐使用唯一编号，ioctl 方法的命令编号方法根据如下规则定义，编号分为 4 个字段。

（1）type（类型）：也称为幻数，8 位宽。

（2）number（号码）：顺序数，8 位宽。

（3）direction（方向）：如果该命令有数据传输，则要定义传输方向，2 位宽，可使用的数值为_IOC_NONE、_IOC_READ 和_IOC_WRITE。

（4）size（大小）：数据大小，宽度与体系结构有关，在 ARM 上为 14 位。这些定义在<linux/ioctl.h>中可以找到。其中还定义了一些用于构造命令的宏：

```
[root@yangzongde root]# cat /usr/src/linux-2.4.20-8/include/linux/ioctl.h
#ifndef _LINUX_IOCTL_H
#define _LINUX_IOCTL_H
#include <asm/ioctl.h>
#endif /* _LINUX_IOCTL_H */

[root@yangzongde root]# cat /usr/src/linux-2.4.20-8/include/asm/ioctl.h
......
#define _IOC_NRBITS    8
#define _IOC_TYPEBITS   8
#define _IOC_SIZEBITS   14
#define _IOC_DIRBITS   2
#define _IOC_NRMASK    ((1 << _IOC_NRBITS)-1)
#define _IOC_TYPEMASK   ((1 << _IOC_TYPEBITS)-1)
#define _IOC_SIZEMASK   ((1 << _IOC_SIZEBITS)-1)
#define _IOC_DIRMASK   ((1 << _IOC_DIRBITS)-1)
#define _IOC_NRSHIFT   0
#define _IOC_TYPESHIFT  (_IOC_NRSHIFT+_IOC_NRBITS)
#define _IOC_SIZESHIFT  (_IOC_TYPESHIFT+_IOC_TYPEBITS)
#define _IOC_DIRSHIFT   (_IOC_SIZESHIFT+_IOC_SIZEBITS)
/*
 * Direction bits.
```

```
 */
#define _IOC_NONE      0U
#define _IOC_WRITE     1U
#define _IOC_READ      2U
#define _IOC(dir,type,nr,size) \
       (((dir)  << _IOC_DIRSHIFT) | \
        ((type) << _IOC_TYPESHIFT) | \
        ((nr)   << _IOC_NRSHIFT) | \
        ((size) << _IOC_SIZESHIFT))
/* used to create numbers */
#define _IO(type,nr)          _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)    _IOC(_IOC_READ,(type),(nr),sizeof(size))
#define _IOW(type,nr,size)    _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
#define _IOWR(type,nr,size)   _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),
sizeof(size))
/* used to decode ioctl numbers.. */
#define _IOC_DIR(nr)          (((nr) >> _IOC_DIRSHIFT) & _IOC_DIRMASK)
#define _IOC_TYPE(nr)         (((nr) >> _IOC_TYPESHIFT) & _IOC_TYPEMASK)
#define _IOC_NR(nr)           (((nr) >> _IOC_NRSHIFT) & _IOC_NRMASK)
#define _IOC_SIZE(nr)         (((nr) >> _IOC_SIZESHIFT) & _IOC_SIZEMASK)
/* ...and for the drivers/sound files... */
#define IOC_IN        (_IOC_WRITE << _IOC_DIRSHIFT)
#define IOC_OUT       (_IOC_READ << _IOC_DIRSHIFT)
#define IOC_INOUT     ((_IOC_WRITE|_IOC_READ) << _IOC_DIRSHIFT)
#define IOCSIZE_MASK   (_IOC_SIZEMASK << _IOC_SIZESHIFT)
#define IOCSIZE_SHIFT  (_IOC_SIZESHIFT)
#endif /* _ASMI386_IOCTL_H */
```

下面介绍 ioctl 方法的返回值。

ioctl 通常实现一个基于 switch 语句的各个命令的处理，当用户程序传递了不合适的命名参数时，POSIX 标准规定应返回–ENOTTY，返回–EINVAL 是以前常见的方法。不能使用与 Linux 预定义命令相同的号码，因为这些号码会被内核 sys_ioctl 函数识别，并且不再将命令传递给驱动的 ioctl。由于 Linux 针对所有文件的预定义命令的幻数为“T”，所以不应使用 TYPE 为“T”的幻数。

```
static int demo_ioctl(struct inode *inode, struct file *file,
             unsigned int cmd, unsigned long arg)
{
    printk("ioctl runing\n");
    switch(cmd){
        case 1:printk("runing command 1 \n");break;
        case 2:printk("runing command 2 \n");break;
        default:
            printk("error cmd number\n");break;
    }
    return 0;
}
```

static struct file_operations demo_fops = {…} 完成了将驱动函数映射为标准接口的功能，其定义如下：

```
static struct file_operations demo_fops = {
    owner:  THIS_MODULE,
    write:  demo_write,
    read:   demo_read,
```

```
    ioctl:  demo_ioctl,
    open:   demo_open,
    release:    demo_release,
};
```

5．初始化函数 init 及退出函数 exit

```
static int __init demo_init(void)
{
#ifdef CONFIG_DEVFS_FS
    devfs_demo_dir = devfs_mk_dir(NULL, "demo", NULL);
    devfs_demoraw = devfs_register(devfs_demo_dir, "0", DEVFS_FL_DEFAULT,
            demo_MAJOR, demo_MINOR, S_IFCHR | S_IRUSR | S_IWUSR,
            &demo_fops, NULL);
#else
    int  result;
   SET_MODULE_OWNER(&demo_fops);
   result = register_chrdev(demo_MAJOR, "demo", &demo_fops);
   if (result < 0) return result;
//   if (demo_MAJOR == 0) demo_MAJOR = result; /* dynamic */
#endif
    printk(DEVICE_NAME " initialized\n");
    return 0;
}
static void __exit  demo_exit(void)
{
   unregister_chrdev(demo_MAJOR, "demo");
   //kfree(demo_devices);
    printk(DEVICE_NAME " unloaded\n");
}
module_init(demo_init);
module_exit(demo_exit);
```

devfs_register 函数的其原型为：

```
devfs_register(devfs_handle_t dir,const char *name,unsigned int flags,
unsigned int major,unsigned int minor,umode_t mode,void *ops,void *info)
```

其中的参数说明如下所示。

Dir：新创建的设备文件的父目录，一般设为 null，表示父目录为/dev。

Name：设备名称，如想包含子目录，可以直接在名字中包含“/”。

Flags：Devfs 标志的位掩码。

Major：主设备号，如果在 flags 参数中指定为 DEVFS_FL_AUTO_DEVNUM，则主次设备号就无用了。

Minor：次设备号。

Mode：设备的访问模式。

Ops：设备的文件操作数据结构指针。

Info：filp->private_data 的默认值。

6．阻塞型 IO

read 调用有时会出现当前没有数据可读，但是马上就会有数据到达的情况，此时就会使用睡眠并等待数据的方法，这就是阻塞型 IO，write 也是同样的道理。在阻塞型 IO 中涉

及如何使进程睡眠、如何唤醒、如何在阻塞的情况查看是否有数据等问题。

当进程等待一个事件时，应该进入睡眠，等待被事件唤醒，这主要是由等待队列这种机制来处理多个进程的睡眠与唤醒、这里要使用到如下几个函数和结构，这些结构和函数的定义在<linux/wait.h>文件中。

```
struct __wait_queue_head {
      wq_lock_t lock;
      struct list_head task_list;
#if WAITQUEUE_DEBUG
      long __magic;
      long __creator;
#endif
};
typedef struct __wait_queue_head wait_queue_head_t;
```

初始化函数

```
struct inline void init_waitqueue_head(wait_queue_head_t *q)
```

如果声明了等待队列，并完成初始化，进程就可以睡眠。

大多数情况下应使用“可中断”的函数，也就是带 interruptible 的函数。还要注意睡眠进程被唤醒并不一定代表有数据，由于也有可能被其他信号唤醒，所以醒来后需要测试 condition。

7．并发访问与数据保护

（1）使用循环缓冲区并且避免使用共享变量方法是类似于“生产者消费者问题”的处理方法，生产者向缓冲区写入数据，消费者从缓冲区读取数据。

（2）使用自旋锁实现互斥访问

自旋锁的操作函数定义在<linux/spinlock.h>文件中。其中包含了许多宏定义，主要的函数如表 5-1 所示。

表 5-1　自旋锁操作函数

函　　数	功　　能
spin_lock_init(lock))	初始化锁
spin_lock(lock)	获取给定的自旋锁
spin_is_locked(lock)	查询自旋锁的状态
spin_unlock_wait(lock))	释放自旋锁
spin_unlock(lock)	释放自旋锁
spin_lock_irqsave(lock, flags)	保存中断状态获取自旋锁
spin_lock_irq(lock)	不保存中断状态获取自旋锁
spin_lock_bh(lock)	获取给定的自旋锁并阻止底半部的执行

Linux 中还提供了称为读者/写者自旋锁，这种锁的类型为 rwlock_t，可以通过<linux/spinlock.h>文件查看更详细的内容。

8．中断处理

中断是现在所有微处理器的重要功能，Linux 驱动程序中对于中断的处理方法一般使

用以下几个函数。下面是请求安装某个中断号的处理程序。

```
    extern int request _irq(unsigned int irq,void (*handleer)(int ,void *,struct
pt_regs *),unsigned long flag,const char *dev_name,void *dev_id)
```

request_irq 函数中的参数说明如下。

void (*handler)(int, void *, struct pt_regs *)：要安装的处理函数指针。

unsigned long flag：与中断管理相关的位掩码。

const char * dev_name：用于在/proc/interrupts 中显示的中断的拥有者。

void *dev_id：用于标识产生中断的设备号。

释放中断：

```
extern void free_irq(unsigned int,void *)
```

中断处理程序与普通 C 代码没有太大不同，不同的是中断处理程序在中断期间运行，它有如下限制。

- 不能向用户空间发送或接受数据。
- 不能执行有睡眠操作的函数。
- 不能调用调度函数。

5.5　驱动程序的调试

1．使用 printk 函数

驱动程序的调试最简单的方法是使用 printk 函数，printk 函数中可以附加不同的日志级别或消息优先级，如下所示。

```
printk(KERN_DEBUG"Here is ：%s: %i\n",_FILE,_LINE_)
```

上述例子中宏 KERN_DEBUG 和后面的""之间没有逗号，因为宏实际是字符串，在编译时会由编译器将它与后面的文本拼接在一起。在头文件<linux/kernel.h>中定义了 8 种可用的日志级别字符串。

- KERN_EMERG
- KERN_ALERT
- KERN_CRIT
- KERN_ERR
- KERN_WARNING
- KERN_NOTICE
- KERN_INFO
- KERN_DEBUG

当优先级小于 Console_loglevel 这个整数时，消息才能被显示到控制台，如果系统运行了 klogd 和 syslogd，则内核将把消息输出到/var/log/messages 中。

2．使用/proc 文件系统

/proc 文件系统是由程序创建的文件系统，内核利用它向外输出信息。/proc 目录下的

每一个文件都被绑定到一个内核函数，这个函数在此文件被读取时，动态地生成文件的内容。典型的例子就是 ps、top 命令，就是通过读取/proc 下的文件来获取需要的信息。大多数情况下 proc 目录下的文件是只读的。使用/proc 的模块必须包含<linux/proc_fs.h>头文件。接口函数 read_proc 可用于输出信息，其定义如下：

```
int(*read_proc)(char *page,char **start,off_t offset,int count,int *eof,
void *data)
```

参数的含义如下。

page：将要写入数据的缓冲区指针。

start：数据将要写入的页面位置。

offset：页面中的偏移量。

count：写入的字节数。

eof：指向一个整型数，当没有更多数据时，必须设置这个参数。

data：驱动程序特定的数据指针，可用于内部使用。

函数的返回值表示实际放入页面缓冲区的数据字节数。建立函数与/proc 目录下的文件之间的关联使用 create_proc_read_entry()函数，其定义如下：

```
stuct proc_dir_entry *creat_proc_entry(const char *name,mode_t mode,stuct
proc_dir_entry *parent);
```

其中参数的含义介绍如下。

name：文件名称。

mode：文件权限。

parent：文件的父目录的指针，为 null 时代表父目录为/proc。

3．使用 ioctl 方法

ioctl 系统调用会调用驱动的 ioctl 方法，可以通过设置不同的命名号来编写一些测试函数，使 ioctl 系统调用在用户级调用这些函数进行调试。

4．使用 strace 命令进行调试

strace 是一个功能强大的工具，它可以显示用户空间的程序发出的全部系统调用，不仅可以显示调用，还可以显示调用的参数和用符号方式表示的返回值。

strace 有如下几个有用的参数。

-t：显示调用发生的时间。

-T：显示调用花费的时间。

-e：限定被跟踪的系统调用的类型。

-o：将输出重定向到一个文件。

由于 strace 是从内核接收信息，所以它可以跟踪没有使用调试方式编译的程序，还可以跟踪一个正在运行的进程，可以使用它生成跟踪报告，交给应用程序开发人员，对于内核开发人员同样有用，可以通过每次对驱动调用的输入输出数据的检查，发现驱动的工作是否正常。

5.6　本章总结

本章简要介绍了 Linux 设备驱动的基础知识，包括基本概念、设备驱动结构、常用接口函数以及驱动程序的调试。读者学习时要多温习和巩固常用接口函数的使用，对以后学习驱动程序与系统开发很有帮助。

第 6 章

网卡驱动程序开发

本章将详细介绍网卡驱动程序的开发，首先介绍网卡基础知识。

6.1 网卡概述

网络接口卡 NIC（Network Interface Card）又称网络适配器（NIA-Network Interface Adapter），简称网卡，用于实现联网计算机和网络电缆之间的物理连接，为计算机之间相互通信提供一条物理通道，并通过这条通道进行高速数据传输。在局域网中，每一台联网计算机都需要安装一块或多块网卡，通过介质连接器将计算机接入网络电缆系统。网卡完成物理层和数据链路层的大部分功能。大多具有 10/100/1000 Mbps 自行配置功能，自动与以太网、快速以太网和千兆位以太网兼容。

6.2 RTL8193 网卡驱动

系统启动时，PCI 总线驱动会扫描总线上所有的 PCI 设备，为每一个设备创建一个 struct pci_dev 结构，相当于设备的配置空间的信息结构，所以系统在启动后，没有加载驱动前，列出所有 PCI 设备的信息。当系统检测到某个 PCI 设备时，将为该设备分配中断号、存储空间基址等，这些信息都被填入 PCI 设备的配置空间中，即填入 struct pci_dev *pdev 中，驱动程序只要访问这个数据结构，就可以读出 PCI 设备的信息。

PCI 是一个总线标准，PCI 总线上的设备就是 PCI 设备，这些设备有很多类型，当然也包括网卡设备，每一个 PCI 设备在内核中抽象为一个数据结构 pci_dev，它描述了一个 PCI 设备所有的特性，但是有几个地方和驱动程序的关系特别大，必须予以说明。

PCI 设备都遵守 PCI 标准，这个部分所有的 PCI 设备都是一样的，每个 PCI 设备都有一段寄存器存储着配置空间，这一部分格式是一样的，比如第一个寄存器总是生产商号码，如 Realtek 就是 10ec，而 Intel 则是另一个数字。用户可以通过配置空间来辨别其生产商、设备号，不论采用什么平台，x86 也好、ppc 也好，它们都是同一标准格式。但光有这些 PCI 配置空间的统一格式还是不够的，网卡设备是 PCI 设备必须遵守的规则，在设备里集成了 PCI 配置空间，但是一个网卡就必须同时集成能控制网卡工作的寄存器。而寄存器的访问就成了一个问题。

在 Linux 里是把这些寄存器映射到主存虚拟空间上的，换句话说，CPU 访存指令就可以访问到这些处于外设中的控制寄存器。PCI 设备主要包括两类空间：一类是配置空间，它是操作系统或 BIOS 控制外设的统一格式的空间，CPU 指令不能访问，访问这个空间要

借助 BIOS 功能，事实上 Linux 的访问配置空间的函数是通过 CPU 指令驱使 BIOS 来完成读写访问的；而另一类是普通的控制寄存器空间，这一部分映射完后 CPU 可以访问来控制设备工作。

介绍设备驱动之前先预览网络收发过程，如图 6-1 所示。

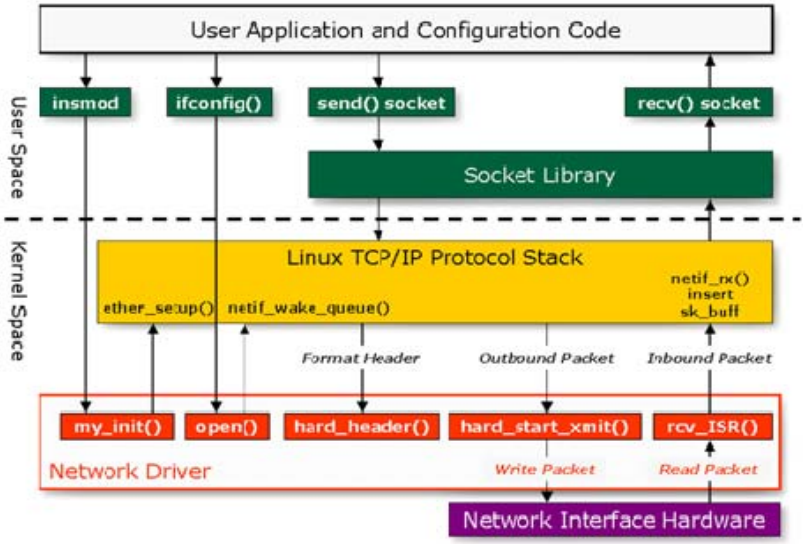


图 6-1　网络收发过程

网络卡收发数据在网络最底层，主要由 hard_start_xmit()等函数实现。

6.2.1　网卡驱动的初始化

驱动程序的初始化函数 rtl8139_init_module 调用 pci_register_driver(&rtl8139_pci_driver)，其中参数 rtl8139_pci_driver 结构如下。

```
static struct pci_driver rtl8139_pci_driver = {
     .name       = DRV_NAME,
     .id_table   = rtl8139_pci_tbl,
     .probe      = rtl8139_init_one,
     .remove     = __devexit_p(rtl8139_remove_one),
#ifdef CONFIG_PM
     .suspend    = rtl8139_suspend,
     .resume     = rtl8139_resume,
#endif /* CONFIG_PM */
};
/*最重要的参数*/
static struct pci_device_id rtl8139_pci_tbl[] = {
     {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
     {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
     {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
     {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
#ifdef CONFIG_SH_SECUREEDGE5410
     /* Bogus 8139 silicon reports 8129 without external PROM : -( */
     {0x10ec, 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
#endif
#ifdef CONFIG_8139TOO_8129
```

```
        {0x10ec, 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8129 },
    #endif

        /* some crazy cards report invalid vendor ids like
         * 0x0001 here.  The other ids are valid and constant,
         * so we simply don't match on the main vendor id.
         */
        {PCI_ANY_ID, 0x8139, 0x10ec, 0x8139, 0, 0, RTL8139 },
        {PCI_ANY_ID, 0x8139, 0x1186, 0x1300, 0, 0, RTL8139 },
        {PCI_ANY_ID, 0x8139, 0x13d1, 0xab06, 0, 0, RTL8139 },
        {0,}
    };
```

rtl8139_pci_tbl 中由 VerdonID、DeviceID 等组成，表示该设备驱动程序可以控制的设备，每个 PCI 设备在配置空间中固化了自己的基本信息。内核启动时 PCI 总线驱动程序会扫描 PCI 总线上的设备，把这些信息收集起来，并把每个设备的信息保存在数据结构中。pci_register_driver 进行必要的初始化后，调用参数指定的 rtl8139_init_one。根据 id_table 的信息，将 device、device 对应的 pci_dev 和 device 对应的 driver 三者联系起来。它先注册一个 pci_driver，找出真正的 pci_device，然后在 init 程序中把 pci_device 与 net_device 关联起来。详细过程如下：

- 初始化函数：rtl8139_init_one，rtl8139_init_board。
- 打开函数：rtl8139_open，初始化 DMA 空间，调用 rtl8139_hw_start 开启设备，调用 netif_start_queue 通知上层可以发数据下来了。
- 总的中断函数：rtl8139_interrupt。
- 接收新数据中断函数：rtl8139_rx_interrupt。
- 发送完毕中断函数：rtl8139_tx_interrupt。
- 发送函数：rtl8139_start_xmit。

首先来看看设备的初始化。当正确编译完程序后，就需要把生成的目标文件加载到内核中，模块也有第一个执行的函数，即 module_init(rtl8139_init_module)，在程序中，rtl8139_init_module（）在 insmod 之后首先执行，它的代码如下：

```
static int __init rtl8139_init_module (void)
{
#ifdef MODULE
    printk (KERN_INFO RTL8139_DRIVER_NAME "\n");
#endif
    return pci_register_driver(&rtl8139_pci_driver);
}
```

它直接调用了 pci_module_init（），这个函数代码在 Linux/drivers/net/8130o100too.c 中，把 rtl8139_pci_driver 的地址作为参数传给了它。rtl8139_pci_driver 定义如下：

```
static struct pci_driver rtl8139_pci_driver = {
name：MODNAME,
id_table：rtl8139_pci_tbl,
probe：rtl8139_init_one,
remove：rtl8139_remove_one,
};
```

pci_module_init（）在驱动代码里没有定义，读者可能想到了，它是 Linux 内核提供

给模块的一个标准接口，里面调用了 pci_register_driver（），这个函数代码在 Linux/drivers/pci/pci.c 中。pci_register_driver 进行以下三项过程。

（1）把带过来的参数 rtl8139_pci_driver 在内核中进行了注册，内核中有一个 PCI 设备的大的链表，这里负责把这个 PCI 驱动挂到里面去。

（2）查看总线上所有 PCI 设备（网卡设备属于 PCI 设备的一种）的配置空间。如果发现标识信息与 rtl8139_pci_driver 中的 id_table 相同，即 rtl8139_pci_tbl，它的定义如下：

```
static struct pci_device_id rtl8139_pci_tbl[] __devinitdata = {
{0x10ec, 0x8129, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 1},
{PCI_ANY_ID, 0x8139, 0x10ec, 0x8139, 0, 0,0 },
{0,}
};
```

这个驱动就是用来驱动这个设备的，于是调用 rtl8139_pci_driver 中的 probe 函数（即 rtl8139_init_one），这个函数是在驱动程序中定义的，它用来初始化整个设备并进行一些准备工作。需要注意 pci_device_id 是内核定义的用来辨别不同 PCI 设备的一个结构，例如在这里 0x10ec 代表的是 Realtek 公司，扫描 PCI 设备配置空间如果发现有 Realtek 公司制造的设备时，两者就对上了。当然对上了公司号后还得看其他的设备号等信息，都对上了才说明这个驱动是可以为这个设备服务的。

（3）把这个 rtl8139_pci_driver 结构挂在这个设备的数据结构（pci_dev）上，表示这个设备从此就有了自己的驱动了，而驱动也找到了它服务的对象。

看上面 pci_register_driver 的第二步，如果找到相关设备和 pci_device_id 结构数组对上了，说明找到服务对象了，则调用 rtl8139_init_one。主要实现过程如下：

（1）建立 net_device 结构，让它在内核中代表这个网络设备。读者可能会问，pci_dev 也是代表着这个设备，两者有什么区别呢？正如上面讨论的，网卡设备既要遵循 PCI 规范，也要担负起其作为网卡设备的职责，于是就分了两块，pci_dev 用来负责网卡的 PCI 规范，而这里要说的 net_device 则是负责网卡的网络设备。

```
dev = init_etherdev (NULL, sizeof (*tp));
if (dev == NULL) {
printk ("unable to alloc new ethernet\n");
return -ENOMEM;
}
tp = dev->priv;
```

init_etherdev 函数在 Linux/drivers/net/net_init.c 中，在这个函数中分配了 net_device 的内存并进行了初步的初始化。需要注意的是，net_device 中的一个成员 priv，它代表着不同网卡的私有数据，比如 Intel 的网卡和 Realtek 的网卡在内核中都是以 net_device 来代表。但是它们是有区别的，比如 Intel 和 Realtek 实现同一功能的方法不一样，这些都是靠着 priv 来体现。在分配内存时，net_device 中除了 priv 以外的成员都是固定的，而 priv 的大小是任意的，所以分配时要把 priv 的大小传过去。

开启这个设备（其实是开启了设备的寄存器映射到内存的功能），代码如下。

```
rc = pci_enable_device (pdev);
if (rc)
goto err_out;
```

pci_enable_device 也是一个内核开发出来的接口，代码在 drivers/pci/pci.c 中，这个函

数主要就是把 PCI 配置空间的 Command 域的 0 位和 1 位置成了 1，从而达到了开启设备的目的，因为 rtl8139 的官方数据表中，说明了这两位的作用就是开启内存映射和 I/O 映射，如果不开的话，那以上讨论的把控制寄存器空间映射到内存空间的这一功能就被屏蔽了，这是非常不利的；除此之外，pci_enable_device 还做了些中断开启工作。

（2）获得各项资源

```
mmio_start = pci_resource_start (pdev, 1);
mmio_end = pci_resource_end (pdev, 1);
mmio_flags = pci_resource_flags (pdev, 1);
mmio_len = pci_resource_len (pdev, 1);
```

在硬件加电初始化时，BIOS 固件统一检查了所有的 PCI 设备，并统一为它们分配了一个和其他互不冲突的地址，让它们的驱动程序可以向这些地址映射它们的寄存器，这些地址被 BIOS 写进了各个设备的配置空间。因为这个活动是一个 PCI 的标准的活动，所以自然写到各个设备的配置空间里，而不是它们风格各异的控制寄存器空间里。当然只有 BIOS 可以访问配置空间。当操作系统初始化时，BIOS 为每个 PCI 设备分配了 pci_dev 结构，并且把 BIOS 获得的并写到了配置空间中的地址读出来写到了 pci_dev 中的 resource 字段中。这样以后再读这些地址就不需要再访问配置空间了，直接从 pic-dev 中读出就可以了。这里的四个函数就是直接从 pci_dev 读出了相关数据，代码在 include/Linux /pci.h 中。定义如下：

```
#define pci_resource_start(dev,bar) ((dev)->resource[(bar)].start)
#define pci_resource_end(dev,bar) ((dev)->resource[(bar)].end)
```

每个 PCI 设备有 0~5 共 6 个地址空间，通常只使用前两个，这里把参数 1 传给了 bar 即使用内存映射的地址空间。把得到的地址进行映射如下：

```
ioaddr = ioremap (mmio_start, mmio_len);
if (ioaddr == NULL) {
printk ("cannot remap MMIO, aborting\n");
rc = -EIO;
goto err_out_free_res;
}
```

ioremap 是内核提供的用来映射外设寄存器到主存的函数，要映射的地址已经从 pci_dev 中读了出来（上一步），这样就成功映射了而不会和其他地址有冲突。映射完后有什么效果呢？比如某个网卡有 100 个寄存器，它们都是连在一块的，位置是固定的，加入每个寄存器占 4 个字节，那么一共 400 个字节的空间被映射到内存成功后，ioaddr 就是这段地址的开头（注意 ioaddr 是虚拟地址，而 mmio_start 是物理地址，它们都是 BIOS 得到的，而保护模式下 CPU 不认物理地址，只认虚拟地址），ioaddr+0 就是第一个寄存器的地址，ioaddr+4 就是第二个寄存器地址（每个寄存器占 4 个字节），依此类推，就能够在内存中访问到所有的寄存器进而操控它们了。

重启网卡设备是初始化网卡设备的一个重要部分，它的原理就是向寄存器中写入命令（注意这里写寄存器，而不是配置空间，因为跟 PCI 没有什么关系），代码如下：

```
writeb ((readb(ioaddr+ChipCmd) & ChipCmdClear) | CmdReset,ioaddr+ChipCmd);
```

第二个参数 ioaddr+ChipCmd 中，ChipCmd 是一个位移，使地址刚好对应的就是

ChipCmd 那个寄存器，读者可以查阅官方数据表得到这个位移量，在程序中定义的这个值为 ChipCmd = 0x37；与数据表是吻合的。把这个命令寄存器中相应位（RESET）置 1 就可以完成操作。

（3）获得 MAC 地址，并把它存储到 net_device 中。

```
for(i = 0; i < 6; i++) { /* Hardware Address */
dev->dev_addr[i] = readb(ioaddr+i);
dev->broadcast[i] = 0xff;
}
```

可以看到读的地址是 ioaddr+0 到 ioaddr+5，读者查看官方数据表会发现寄存器地址空间的开头 6 个字节正好存的是这个网卡设备的 MAC 地址，MAC 地址是网络中标识网卡的物理地址，这个地址在今后的收发数据包时会用得上。

向 net_device 中登记一些主要的函数，代码如下。

```
dev->open = rtl8139_open;
dev->hard_start_xmit = rtl8139_start_xmit;
dev->stop = rtl8139_close;
```

由于 dev（net_device）代表着设备，把这些函数注册完后，rtl8139_open 就是用于打开这个设备，rtl8139_start_xmit 就是当应用程序要通过这个设备往外面发数据时被调用，这个函数其实是在网络协议层中调用的，这就涉及 Linux 网络协议栈的内容，这部分内容不在讨论范围内，这里只是负责实现它。rtl8139_close 用来关掉这个设备。

至此把 rtl8139_init_one 函数介绍完了。初始化设备后，通过 ifconfig eth0 up 命令来把设备激活。这个命令直接对刚刚注册的 rtl8139_open 进行了调用。这个函数激活了设备，主要进行下面三项过程。

（1）注册这个设备的中断处理函数。

当网卡发送数据完成或者接收到数据时，是用中断的形式告知的，比如有数据从网线传来，中断也通知了，那么必须要有一个处理这个中断的函数来完成数据的接收。中断号的分配和内存地址映射一样，中断号也是 BIOS 在初始化阶段分配并写入设备配置空间的，然后 Linux 在建立 pci_dev 时从配置空间读出这个中断号，再写入 pci_dev 的 irq 成员中。所以注册中断程序需要中断号，直接从 pci_dev 里取就可以了。

```
    retval = request_irq (dev->irq, rtl8139_interrupt, SA_SHIRQ, dev->name,
dev);
    if (retval) {
    return retval;
    }
```

这里注册的中断处理函数是 rtl8139_interrupt，也就是说当网卡发生中断（如数据到达）时，中断控制器 8259A 把中断号发给 CPU，CPU 根据这个中断号找到处理程序，这里是 rtl8139_interrupt，然后执行。rtl8139_interrupt 也是在程序中定义好的，这是驱动程序的一个基本功能。request_irq 的代码在 arch/i386/kernel/irq.c 中。

（2）分配发送和接收的缓存空间

根据官方文档，发送一个数据包的过程为：先从应用程序中把数据包复制到一段连续的内存中（这段内存就是这里要分配的缓存），然后把这段内存的地址写进网卡的数据发送地址寄存器（TSAD）中，这个寄存器的偏移量是 TxAddr0 = 0x20。再把这个数据包

的长度写进另一个寄存器（TSD）中，它的偏移量是 TxStatus0 = 0x10。接着就把这段内存的数据发送到网卡内部的发送缓冲区（FIFO），最后由这个发送缓冲区把数据发送到网线上。

```
tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TX_BUF_TOT_LEN,
&tp->tx_bufs_dma);
tp->rx_ring = pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN,
&tp->rx_ring_dma);
```

上面 tp 是 net_device 的 priv 的指针，tx_bufs 是发送缓冲内存的首地址，rx_ring 是接收缓存内存的首地址，它们都是虚拟地址，而最后一个参数 tx_bufs_dma 和 rx_ring_dma 均是这一段内存的物理地址。为什么同一个事物既用虚拟地址来表示还要用物理地址来表示呢？这是因为 CPU 执行程序用到这个地址时，用虚拟地址；而网卡设备向这些内存中存取数据时用的是物理地址。pci_alloc_consistent 的代码在 Linux/arch/i386/kernel/pci-dma.c 中。

（3）发送和接收缓冲区初始化和网卡开始工作的操作

RTL8139 有 4 个发送描述符（包括 4 个发送缓冲区的基地址寄存器（TSAD0-TSAD3）和 4 个发送状态寄存器（TSD0-TSD3）。因此，分配的缓冲区要分成四等份并把这 4 个空间的地址都写到相关寄存器里去，下面这段代码完成了这个操作。

```
for (i = 0; i < NUM_TX_DESC; i++)
((struct rtl8139_private*)dev->priv)->tx_buf[i] =
&((struct rtl8139_private*)dev->priv)->tx_bufs[i * TX_BUF_SIZE];
```

上面这段代码负责把发送缓冲区虚拟空间进行了分割。

```
for (i = 0; i < NUM_TX_DESC; i++)
{
writel(tp->tx_bufs_dma+(tp->tx_buf[i]tp->tx_bufs),ioaddr+TxAddr0+(i*4));
readl(ioaddr+TxAddr0+(i * 4));
}
```

上面这段代码负责把发送缓冲区物理空间进行了分割，并把它写到了相关寄存器中，这样在网卡开始工作后就能够迅速定位及找到这些内存并存取它们的数据。

```
writel(tp->rx_ring_dma,ioaddr+RxBuf);
```

上面这行代码是把接收缓冲区的物理地址写到了相关寄存器中，这样网卡接收到数据后就能准确地把数据从网卡中搬运到这些内存空间中，等待 CPU 来领走它们。

```
writeb((readb(ioaddr+ChipCmd) & ChipCmdClear) |
CmdRxEnb | CmdTxEnb,ioaddr+ChipCmd);
```

重新设定设备后，要激活设备的发送和接收的功能，上面这行代码就是向相关寄存器中写入相应值，激活了设备的这些功能。

```
writel ((TX_DMA_BURST << TxDMAShift),ioaddr+TxConfig);
```

上面这行代码是向网卡的 TxConfig（位移是 0x44）寄存器中写入 TX_DMA_BURST << TxDMAShift 这个值，翻译过来就是 6<<8，就是把第 8 到第 10 这三位置成 110，官方文档 6 就是 110 代表着一次 DMA 的数据量为 1024 字节。另外在这个阶段设置了接收数据的模式、开启中断等。

6.2.2　网卡数据收发

下面进入数据收发阶段。当一个网络应用程序要向网络发送数据时，它要利用 Linux 的网络协议栈来解决一系列问题，找到网卡设备的代表 net_device，由这个结构找到并控制这个网卡设备来完成数据包的发送，具体是调用 net_device 的 hard_start_xmit 成员函数，这是一个函数指针，在驱动程序里它指向的是 rtl8139_start_xmit，通过它来完成发送工作。下面就来剖析这个函数。它一共进行如下四项过程。

（1）检查这个要发送的数据包的长度，如果它达不到以太网帧的长度，必须采取措施进行填充。

```
if( skb->len < ETH_ZLEN ){//if data_len < 60
if( (skb->data + ETH_ZLEN) <= skb->end ){
memset( skb->data + skb->len, 0x20, (ETH_ZLEN - skb->len) );
skb->len = (skb->len >= ETH_ZLEN) ? skb->len : ETH_ZLEN;}
else{
printk("%s: (skb->data+ETH_ZLEN) > skb->end\n",__FUNCTION__);
}
}
```

skb->data 和 skb->end 就决定了这个包的内容，如果这个包本身总共的长度(skb->end-skb->data)都达不到要求，就出错返回，否则就采取措施填充。

（2）把包的数据复制到已经建立好的发送缓存中。

```
memcpy (tp->tx_buf[entry], skb->data, skb->len);
```

其中 skb->data 就是数据包数据的地址，而 tp->tx_buf[entry]就是发送缓存地址，这样就完成了复制。

（3）光有了地址和数据还不行，还要让网卡知道这个包的长度，才能保证数据不多不少精确地从缓存中截取出来搬运到网卡中去，这是靠写发送状态寄存器（TSD）来完成的。

```
writel(tp->tx_flag | (skb->len >= ETH_ZLEN ? skb->len : ETH_ZLEN),\
ioaddr+TxStatus0+(entry * 4));
```

把这个包的长度和一些控制信息一起写进了状态寄存器，使网卡的工作有了依据。

（4）判断发送缓存是否已经满了，如果满了再发就覆盖数据了，就要停发。

```
if ((tp->cur_tx - NUM_TX_DESC) == tp->dirty_tx)
netif_stop_queue (dev);
```

接收过程是：当有数据从网线上过来时，网卡产生一个中断，调用的中断服务程序是 rtl8139_interrupt，它主要进行下面三项过程。

- 从网卡的中断状态寄存器中读出状态值进行分析

```
status = readw(ioaddr+IntrStatus);
if ((status &(PCIErr | PCSTimeout | RxUnderrun | RxOverflow |
RxFIFOOver | TxErr | TxOK | RxErr | RxOK)) == 0)
goto out;
```

上面代码说明如果上面这 9 种情况均没有，表示没什么需要处理的，退出。

- 进行接收处理

```
if (status & (RxOK | RxUnderrun | RxOverflow | RxFIFOOver))/* Rx interrupt */
rtl8139_rx_interrupt (dev, tp, ioaddr);
```

如果是以上 4 种情况，属于接收信号，调用 rtl8139_rx_interrupt 进行接收处理。

- 进行发送善后处理

```
 if (status & (TxOK | TxErr)) {
spin_lock (&tp->lock);
rtl8139_tx_interrupt (dev, tp, ioaddr);
spin_unlock (&tp->lock);
}
```

如果是传输完成的信号，就调用 rtl8139_tx_interrupt 发送善后处理。下面先来看看接收中断处理函数 rtl8139_rx_interrupt，在这个函数中主要进行下面四项过程。

- 这个函数是一个大循环，循环条件是只要接收缓存不为空就还可以继续读取数据，循环不会停止，读空了之后就跳出。

```
int ring_offset = cur_rx % RX_BUF_LEN;
rx_status = le32_to_cpu (*(u32 *) (rx_ring + ring_offset));
rx_size = rx_status >> 16;
```

上面三行代码是计算出要接收的包的长度。

- 根据这个长度来分配包的数据结构。

```
skb = dev_alloc_skb (pkt_size + 2);
```

- 如果分配成功就把数据从接收缓存中复制到这个包中。

```
eth_copy_and_sum (skb, &rx_ring[ring_offset + 4], pkt_size, 0);
```

这个函数在 include/Linux /etherdevice.h 中，实质还是调用了 memcpy（）。

```
static inline void eth_copy_and_sum(struct sk_buff*dest, unsigned char *src,
int len, int base)
{
memcpy(dest->data, src, len)；
}
```

现在已经熟知，&rx_ring[ring_offset + 4]就是接收缓存，也是源地址，而 skb->data 就是包的数据地址，也是目的地址。

- 把这个包送到 Linux 协议栈去进行下一步处理。

```
skb->protocol = eth_type_trans (skb, dev);
netif_rx (skb);
```

在 netif_rx()函数执行完后，这个包的数据就脱离了网卡驱动范畴，而进入了 Linux 网络协议栈中。把这些数据包的以太网帧头、IP 头、TCP 头都脱下来，最后把数据送给了应用程序，但协议栈不在本书讨论范围内。netif_rx 函数在 net/core/dev.c 中，rtl8139_remove_one 则基本是 rtl8139_init_one 的逆过程。

6.3 典型实例——Ralink 无线网卡驱动开发

6.3.1 Ralink 无线网卡

Ralink 无线网卡是由台湾 Ralink 公司生产的一款基于 802.11g 协议的无线网卡，采用了 Ralink 公司生产的 rt2560 无线网络芯片。同时，为了更好地用于手持式设备并减小体积，

使用了 minipci 的接口方式（minipci 是 PCI 规格的小型化版本）。由于该设备具有功耗低、成本低廉的特点，所以被用于嵌入式无线数据传送。Ralink 无线网卡主要由扩频通信机、网络接口、天线三大部分组成。其中扩频通信机（也就是 rt2560 芯片）由扩频/解扩（SS）模块、中频调制/解调（IF）模块及微波收发模块（RF）组成。网络接口控制卡（NIC）与数据链路层中的媒体访问控制层相对应，完成从上层接收数据并装帧发送、从下层接收数据比特流、帧同步等工作。扩频通信机使得网络数据信息实现无线电信号的接收与发射，完成从上层接收数据流并经过基带加扰、扩频、调制、上混频、功率等处理后，把数据流经天线发送出去，从天线接收信号并经低噪声放大、下混频、调制、解扩、数据解扰等处理后把信号恢复成比特流送至网络接口层等工作。当上层协议有数据要发送时，NIC 负责接收上层协议发送的数据，按照一定格式封装成帧，然后根据多址接入协议 CSMA/CA 把数据帧发送到信道去；当接收数据时，NIC 根据接受帧中的目的地址判别是否是发往本机的数据，如果是则接收该帧信息，并进行 CRC 检验，拆去帧头，把数据提交给上层协议。

6.3.2　802.11 无线通信协议的选用

802.11 协议是由 IEEE 在 1997 年发布的，这也是在无线局域网领域内的第一个国际上被认可的协议。802.11 协议主要工作在 ISO 协议的最低两层上，并在物理层上进行了一些改动，加入了高速数字传输的特性和连接的稳定性。其主要内容包括如下几个部分。

1．802.11 工作方式

802.11 定义了两种类型的设备，一种是无线站，通常是通过一台 PC 加上一块无线网络接口卡构成的，另一个称为无线接入点（Access Point, AP），它的作用是提供无线和有线网络之间的桥接。

2．802.11 物理层

在 802.11 最初定义的三个物理层包括了两个扩散频谱技术和一个红外传播规范，无线传输的频道定义在 2.4GHz 的 ISM 波段内，可以使用 FHSS（frequency hopping spread spectrum）和 DSSS（direct sequence spread spectrum）这两种技术。

3．802.11 数字链路层

802.11 的 MAC 和 802.3 协议的 MAC 非常相似，都是在一个共享媒体之上支持多个用户共享资源，由发送者在发送数据前先检测网络的可用性。在 802.11 无线局域网协议中，由于存在“Near/Far”现象，这是由于要检测冲突，设备必须能够一边接受数据信号一边传送数据信号，而这在无线系统中是无法办到的，所以采用了新的协议 CSMA/CA（Carrier Sense Multiple Access withCollision Avoidance）。

4．联合结构、蜂窝结构和漫游

802.11 的 MAC 子层负责解决客户端工作站和访问接入点之间的连接。当一个 802.11 客户端进入一个或者多个接入点的覆盖范围时，它会根据信号的强弱以及包错误率来自动选择一个接入点来进行连接，一旦被一个接入点接受，客户端就会将发送接受信号的频道切换为接入点的频段。在拥塞的情况下，将重新协商实现“负载平衡”的功能，它能够使整个无线网络的利用率达到最高。802.11 的 DSSS 中一共存在着相互覆盖的 14 个频道，利用这些频道作为多蜂窝覆盖是合适的。

6.3.3 设备驱动关键数据结构

下面介绍在驱动程序中用到的几个关键数据结构。

首先，网络设备的数据结构是最重要的，定义在 include/Linux /netdevice.h 里。它的注释已经足够详尽。这里只列出它的几个关键数据成员，详细情况参看 netdevice.h。

```
    struct device {
    char *name;
    …….
    /*指向驱动程序的初始化方法*/
    int (*init)(struct device *dev);
    /*一些硬件可以在一块板上支持多个接口，可能用到 if_port*/
    /*trans_start 记录最后一次成功发送的时间。可以用来确定硬件是否工作正常*/
    unsigned long trans_start;          /* 最近 Tx 的 Time (in jiffies)*/
    unsigned long last_rx;              /*最近 Rx 的 Time*/
    /*flags 里面有很多内容，定义在 include/Linux /if.h 里*/
    unsigned short flags;               /* 接口标志*/
    unsigned short family;              /*地址 family ID (AF_INET) */
    unsigned short metric;              /* 路由 metric (未使用) */
    unsigned short mtu;                 /*接口 MTU 值*/
        /* type 标明物理硬件的类型。主要说明硬件是否需要 arp。定义在 include/Linux /if
*_arp.h 里*/
    unsigned short type;                /*接口硬件类型*/
    /*上层协议层根据 hard_header_len 在发送数据缓冲区前面预留硬件帧头空间*/
    unsigned short hard_header_len;     /* 硬件 hdr 长度*/
    void *priv;                         /* priv 指向驱动程序自己定义的一些参数*/
    unsigned char broadcast[MAX_ADDR_LEN]; /* 硬件广播地址*/
    unsigned char pad;                  /*让 dev_addr 对齐到 8bytes*/
    unsigned char dev_addr[MAX_ADDR_LEN];  /* 硬件地址*/
    unsigned char addr_len;             /*硬件地址长度*/
    …….
    /*指向接口 buffers 的指针*/
    struct sk_buff_head buffs[DEV_NUMBUFFS];
    /*指向接口服务程序指针*/
    int (*open)(struct device *dev);
    int (*stop)(struct device *dev);
    int (*hard_start_xmit) (struct sk_buff *skb,
    struct device *dev);
    int (*hard_header) (struct sk_buff *skb,
    ……
    };
        /*这一数据结构是整个 Linux 中网络设备的核心结构，它既包括和具体网络设备相关的配置
参数信息，又包含了指向相关处理方法的函数指针*/
    static struct pci_driver rt2500_driver =
    {
    name： "rt2500", /* 设备名*/
    id_table： rt2500_pci_tbl, /* 设备 ID 表*/
    probe： RT2500_init_one, /* 初始化函数*/
    remove： __devexit(RT2500_remove_one),  /* 设备卸载函数*/
    };
```

上面是 PCI 设备驱动的具体框架，主要用于设备注册时的函数传递，通常是在设备驱

动程序安装时调用。

下面的是设备接口私有数据结构，由于结构成员太多，只列出了比较关键的成员。

```
typedef struct _RTMP_ADAPTER
{
    ULONG CSRBaseAddress; /* PCI MMIO 基地址*/
    UCHAR PermanentAddress[ETH_LENGTH_OF_ADDRESS]; /* MAC 地址*/
    UCHAR CurrentAddress[ETH_LENGTH_OF_ADDRESS]; /* 用户修改后的 MAC 地址*/
    UCHAR EEPROMAddressNum; /*eeprom 地址 93c46=6 93c66=8*/
    USHORT EEPROMDefaultValue[NUM_EEPROM_BBP_PARMS];
    struct ring_desc TxRing[TX_RING_SIZE]; /* Tx 环数组，用于传送数据排队*/
    struct ring_desc AtimRing[ATIM_RING_SIZE];
    struct ring_desc PrioRing[PRIO_RING_SIZE];
    struct ring_desc RxRing[RX_RING_SIZE]; /* Rx 环数组，用于接收数据排队*/
    struct ring_desc BeaconRing; /* Beacon 环，只有一个*/
    MGMT_STRUC MgmtRing[MGMT_RING_SIZE];
    ULONG CurRxIndex; /* 下一个 RxD 读指针*/
    ULONG CurDecryptIndex;
    ULONG CurTxIndex; /*下一个 TxD 写指针*/
    ……..
    /*标志*/
    ULONG Flags; /*表示当前设备状态*/
    /*发送队列列表*/
    QUEUE_HEADER TxSwQueue0;
    QUEUE_HEADER TxSwQueue1;
    QUEUE_HEADER TxSwQueue2;
    QUEUE_HEADER TxSwQueue3;
    USHORT Sequence; /* 当前队列号*/
    ……
    PRIVATE_STRUC PrivateInfo; /* 私有数据结构*/
    ……
    struct pci_dev *pPci_Dev; /*指向 PCI 设备数据结构的指针*/
    struct net_device *net_dev; /*指向网络设备数据结构的指针*/
    …….
} RTMP_ADAPTER, *PRTMP_ADAPTER;
```

这个数据结构是 Ralink 无线网卡的核心，对网卡的各种操作都要通过这个设备进行。Ralink 无线网卡驱动程序在设备驱动对象数据结构中定义了一个叫做 ring_desc 的数据结构，主要用于数据接收和发送时数据传递。具体如下：

```
struct ring_desc
{
/* Descriptor 大小 & dma 地址*/
u32 size;
void *va_addr;
dma_addr_t pa_addr;
/*用于实际传输的 Dma buffer 大小和地址*/
u32 data_size;
void *va_data_addr;
dma_addr_t pa_data_addr;
UCHAR FrameType; /* Ring buffer 中的帧类型*/
};
```

其中，va_addr 是描述符的虚拟地址，pa_addr 是描述符的物理地址，va_data_addr 是实际要发送的或者接收到的数据的虚拟地址，pa_data_addr 当然就是物理地址。在 Ralink 无线网卡驱动程序中，数据的发送和接收都是通过这个数据结构来完成的，同时为了提高效率，在程序中以数据结构数组的形式分配多个这种结构，以环行缓冲区的方式来使用。以数据发送为例，当从上层收到一个 skb 结构以后，在发送函数中对它进行处理填充上一些无线 802.11 的协议头，包装成 ring_desc 数据结构的形式存放在 TX 或者 RX 环行队列中，而实际的发送则由中断处理程序在相关函数中通过对寄存器的操作来实现。

6.3.4 rt2500 无线网卡驱动分析

1．初始化

根据 Linux 2.4.21 内核的要求，先通过内核调用 module_init(rt2500_init_module)来注册一个初始化函数 rt2500_init_module( )，然后再在函数 rt2500_init_module( )中通过 pci_module_init(&rt2500_driver)来负责注册模块所提供的任何设施，实际的工作主要是在 RT2500_init_one（）中完成的。

在 RT2500_init_one（）中，驱动程序主要完成了以下工作。

- 初始化 miniPCI 设备；
- 系统资源分配；
- 注册驱动程序。

（1）初始化 miniPCI 设备，在该部分驱动程序主要完成了如下工作。

唤醒和激活 miniPCI 设备：通过 pci_enable_device (pPci_Dev)函数实现，pPci_Dev 为指向数据结构 pci_dev 的指针。

读出配置头中的信息（包括 MAC 地址，配置地址空间等）提供给驱动程序使用：主要是由 RTMP_IO_READ32()函数对 CSR3、CSR4 进行读操作。CSR3、CSR4 是 STA MAC 地址寄存器，地址为：

```
#define CSR3 0x000C //低 4 位寄存器
#define CSR4 0x0010 //高 2 位寄存器
```

设置该设备为 pci 主设备：由 pci_set_master(pPci_Dev)函数完成。系统资源分配在这一阶段，分配了 net_device 数据结构，并且根据从配置头读出的信息和已定义的具体操作函数对这一数据结构进行了填充，其他更重要的内存分配、硬件设置等动作，将其放在设备打开这一阶段来实现，当然也可以直接在这一阶段完成，要根据实际需要来决定。

（2）注册驱动程序

在 Linux 中，向内核注册网络驱动程序一般通过内核函数 register_netdev( )来完成，其具体方式如下：

```
register_netdev(struct net_device *net_dev);
```

在对设备进行注册之前，已经对数据结构 net_dev 进行了填充，特别要重视的是在数据结构 net_dev 中函数指针所指向的 RT2500_open、RTMPSendPackets、RT2500_close、RT2500_ioctl 等关键的操作函数。这样就建立了上层和驱动程序的联系，来自于上层的数据就可以直接调用 RTMPSendPackets 向外发送数据，而来自外部的数据被驱动程序接收到后，则会调用 netif_rx()直接送给上层去处理。

2．设备打开模块驱动程序

一般在 Linux 下的设备驱动程序中，对设备的打开是通过调用 file_operations 结构中的 open 方法来实现的，在注册网络设备驱动程序中 net_dev->open = RT2500_open;具体由 RT2500_open（）函数来实现对设备文件的打开操作，提供给驱动程序以初始化设备的能力，从而为下一步的操作做准备。对设备文件的打开操作始终是启用设备后执行的第一操作。

在 RT2500_open（）函数中，驱动程序主要完成如下工作。

（1）进行 DMA BUFFER 分配。

（2）注册中断处理程序。

在无线网卡设备驱动程序中，需要使用中断让设备在发生某种事件时通知处理器，一般来说，主要是有数据到达或者有管理信息到达。在 Linux 下，Linux 处理中断的方式与它在用户空间处理信号基本一样。驱动程序需要为它自己设备的中断注册一个处理程序。在无线网卡程序中，驱动程序通过如下的代码来实现注册中断处理程序。

```
request_irq(pAd->pPci_Dev->irq, &RTMPIsr, SA_SHIRQ, net_dev->name,net_dev);
```

其中 pAd->pPci_Dev->irq 是申请的中断号，在 Linux 的引导阶段，PCI 设备被赋予了一个唯一的中断号，被保存在配置寄存器 61(PCI_INTERRUPT_PIN)中，该寄存器为一个字节宽，通过读取它就可以获得 PCI 设备的中断号。net_dev 这个指针用于指向相关数据结构，驱动程序使用它指向驱动程序自己的私有数据区。RTMPIsr 是中断发生时将被调用的函数，这是中断处理的重点，在中断发生后，系统就会调用该函数来实现中断处理。SA_SHIRQ 则是一个控制标志，表示所注册的中断使用共享的方式，可以和其他设备共用一个中断号。

（3）读取配置信息，分配并填写相关的数据结构。

Ralink 网卡有两个地方可以保存相关配置信息，一个是 PCI 的配置空间，另一个则是它的 EEPROM，可以通过两种方式来读取；

- 对 PCI 配置空间的数据读写，使用如下的方式。

```
#define RTMP_IO_READ32(_A, _R, _pV) (*_pV = readl(_A->CSRBa
seAddress+_R))
#define RTMP_IO_WRITE32(_A, _R, _V) (writel(_V, _A->CSRBase
Address+_R))
```

通过调用 RTMP_IO_READ32（）函数使用系统函数 readl()来读取数据，A->CSRBase Address 是寄存器基地址，_R 为相对于基地址的偏移地址，_V 是具体要写入的数据（4 字节），_pV 是用于保存读取数据的地址指针。

- 对 EEPROM 的访问，先要通过对 PCI 配置空间相关寄存器进行设置，然后使用 RTMP_IO_READ32（）函数来访问相关地址。由于该网卡没有使用 EEPROM，所以就不详细介绍了。在设备启动前，有一个关键的数据结构 RTMP_ADAPTER 需要完成填写，这是一个很复杂的数据结构。其中最主要的参数如下。
- TxRate 默认传输速度，硬件手册要求为 11M。
- RfType 射频类型，这里使用的是 RFIC_2522。
- BeaconPeriod Beacon 帧发送周期，设置为 100ms.
- Channel 默认频道，取值范围在 1~14，设定为 1。

（4）初始化硬件设置。

这里的硬件指的是 ASIC，具体来说就是控制无线网卡实际操作的各种寄存器，它既有 TXCSR、RXCSR 等传输接收控制寄存器，又有 CSR 等控制/状态寄存器。当然这一配置是严格按照 Ralink 芯片手册的要求。对这些寄存器的操作同样是使用函数 RTMP_IO_READ32(_A, _R, _pV)/ RTMP_IO_WRITE32(_A, _R, _V)。

现在以 CSR7 为例，具体讲述如何对硬件寄存器进行操作。CSR7 是一个 4 字节 32 位的寄存器，它的每一位都代表产生的某特定中断，每当有中断产生时，这一寄存器中的相应位就会自动置 1，通过写入 1 来清 0。

```
#define CSR7 0x001C //中断源寄存器，地址 0x001c
```

寄存器值和中断源对应关系如下：

数据类型在寄存器中位数功能说明

```
ULONG Rsvd：12; //12 个保留位
ULONG Timecsr3Expired：1;
// TIMECSR3 硬件定时器超时在初始化硬件时，通过 RTMP_IO_WRITE32(pAdapter, CSR7, 0xffffffff)来实现，pAdapter 是指向数据结构 RTMP_ADAPTER 的指针，0xffffffff 表示向所有位写入 1，也就是把所有位清 0
```

（5）设置并启动 AP 服务。

无线网卡一般支持 3 种模式，一是 AP 模式（即访问点模式，可以支持其他处于 STA 模式无线网卡和它的连接）；二是 STA 模式（这种模式网卡只有和处于 AP 模式的无线网卡连接）；三是 AD-HOC 模式，也称为对等网模式，在这种模式下，所有这种模式的网卡可以互相连接。

由于本驱动是一个 AP 模式的网卡，所以需要进行 AP 相关设置，其主要内容：一是为 AP 设置 BSSID 号，就是该 AP 的硬件地址，和上面类似，也是由函数 RTMP_IO_WRITE32()来完成，对应的寄存器为 CSR5、CSR6，CSR5 保存低 4 个字节的硬件地址，CSR6 保存高 2 个字节的硬件地址。二是设置网卡使用的频道、速度；对 Ralink 无线网卡而言，它把 2.4G 的频段分为 14 个频道，一般默认设置为 1。速度是指数据发送时使用的发送速度。三是设置一个 beacon 帧定时器，通过不断发送 beacon 帧来搜索周围是否有处于 STA 模式的网卡，达到连接的目的。

3．数据发送模块驱动程序

数据发送是一个网络设备最基本也是最重要的功能，一个应用程序数据要通过网络设备发送出去需要经过分段、打包、组帧的复杂过程。

（1）数据封装

在驱动程序层次上的发送数据是通过低层对硬件的读写来完成的。当上层已封装好的数据到来时，它会自动调用由 hard_start_xmit 函数指针对应的发送函数来进行处理，由它完成数据包的发送。在函数 RT2500_probe 中，把 net_device 结构的 hard_start_xmit 指针初始化为 RTMPSendPackets 函数，所以也就是使用 RTMPSendPackets 来实现这一过程。当系统调用驱动程序的 xmit 时，发送的已封装的数据都会放在一个 sk_buff 结构中，然后传递给 RTMPSendPackets 函数，这个函数把数据传给硬件发出去。如果发送成功，TMPSendPackets 函数释放 sk_buff，返回 0（发

送成功）。如果设备暂时无法处理，比如硬件忙，则返回 1。如果 RTMPSendPackets 发送不成功，则不会释放 sk_buff。

不同于一般的网络驱动程序，AP 模式的无线网卡驱动程序会在启动以后就建立一个 STA MAC 表存放检测到的无线网卡 MAC，对于目的 MAC 地址不在 MAC 表中的数据包，它会直接丢弃，而且由于使用 CSMA/CA 的工作方式，它不能像普通网卡一样使用全双工方式，在发送数据的同时就不能接收数据。由于在无线的情况下，线路质量不能得到保证，使用了一种动态检测网络质量的方式来随时动态调整发送的速度，这就造成发送过程比一般的有线网络发送更加复杂。在这里节电队列、组播队列、发送队列都是在设备打开时，初始化设备软件结构时生成的数据结构，用来存放发送的 skb 数据，而实际的发送动作都是在中断处理程序中完成的，其具体操作将后面阐述。由于传送下来送到各种队列中的 sk_buff 中的数据是由 802.3 协议封装的 MAC 硬件帧头，所以在函数 RTMPSendPackets 中还需要经历一个去掉 MAC 帧头和重新填充 802.11 的 MAC 硬件帧头的过程，然后数据才可以直接提交硬件发送，而实际发送是在中断中实现。

为了方便对这一转换过程的理解，首先介绍一下 802.3 的帧结构。

- 前导（preamble）字段和帧起始字段（SFD）。前导字段 7 字节长（7 个 10101010），其对于快速以太网是多余的，但出于兼容考虑被保留下来。SFD 字段是单字节，表示为 10101011，现已不再使用。
- 目标 MAC 地址（destination address，DA）：6 字节。
- 源 MAC 地址（source address，SA）：6 字节。
- 长度（length）/类型字段，值小于十进制 1546（十六进制 0600）则代表长度。在 LLC 子层中用做提供协议标识，反之标识类型。在以太网处理过程结束后，类型用来指定接收数据的高层协议，2 字节
- 数据和填充（data）：46（64-18）到 1500（1518-18）字节（以太网的最大传输单元 MTU），如果用户的数据小于最小帧长度，就会在用户数据后面进行数据填充。
- 帧校验序列（frame check sequence，FCS）：4 字节，存储 CRC 校验和值。

（2）802.11 的帧结构具体转换过程

- 将发送数据块也就是一个 802.3 的帧，从 sk_buff 结构中复制出来；要注意的是这里只复制从数据块第 15 个字节开始的数据，同时在新的数据块前面要留出 30 个字节作为 802.11 协议帧头。
- 从 sk_buff 指向的数据块之前 14 个字节的帧头中取出目标 MAC 地址、源 MAC 地址和长度/类型字段。
- 按照具体发送帧类型填充新数据块的 802.11 帧头的帧控制域、持续时间域/关联识别码、目标 MAC 地址、源 MAC 地址等信息。
- 计算 FCS 校验码，并将其加到新数据块的尾部。
- 将已经转化好的 802.11 协议的新数据块放入 TxRing 中未使用的 ring_desc 数据结构。
- 将 ring_desc 数据结构放入发送缓冲环 TxRing。

概括来说，把上面的数据发送流程和帧头转换结合起来就构成了一个数据发送模块的全部工作。

（3）中断处理模块驱动程序的开发

在无线网卡驱动程序中，中断处理程序是最核心的部分之一，无论是有数据到达还是要发送数据，其实际操作都是通过硬件触发中断的形式来通知系统对数据进行处理。可以说最底层对硬件的直接操作大部分是在中断中完成。而中断处理模块则主要由两个部分构成。

- 安装中断处理程序：这一步骤已经在打开设备模块中实现。
- 调用中断处理函数：通常这一步是在发出中断后，由系统自动执行。

在无线网卡驱动程序中，对中断信号的屏蔽与启动主要通过硬件来实现，只需将 PCI 设备处理中断的寄存器按硬件要求改变其值，具体方法如下。

```
RTMP_IO_WRITE32(pAd, CSR8, 0x000FFFFF);
#define CSR8 0x0020
```

CSR8 中断掩码寄存器，地址是 0x0020 向 CSR8 中所有中断位置写 1，就完成中断屏蔽的功能。驱动程序在打开设备时就已经安装了中断处理程序，当硬件设备触发中断时，中断处理程序首先从 CSR7（中断源寄存器）中读取其中的值，然后程序判断中断类型，根据中断类型调用相关函数进行处理。

在中断触发以后，系统就会调用已经注册好的相应中断处理程序，这里就是 VOID RTMPIsr(IN INTirq, IN VOID *dev_instance,IN structpt_regs *rgs)这一函数，dev_instance 是该网络设备的数据结构指针，irq 是中断号，rgs 则是在进入 ISR（中断处理过程）之前保存的进程上下文，通过对中断寄存器状态位的读取，可以知道发生中断的原因，然后转入相应的处理。虽然可以产生中断的原因很多，但中断中最常见和最重要的还是数据发送中断和数据接收中断，所以这里仅仅详细说明一下这两个部分。

- 对数据发送中断而言，其处理过程是取出放置在 TxRing 中最前面的一个 ring_desc 数据结构，将发送数据块物理地址放入 TXCSR3，数据大小和相应配置信息放入寄存器 TXCSR1、TXCSR2，置 TXCSR0 的第 0 位为 1，以启动发送。
- 对数据接收中断而言，和数据发送中断正好相反，数据接收中断是当数据到达设备时，产生一个数据接收中断信号，再调用中断处理程序。

程序工作如下：

- 从 RxRing 缓冲行中取出未使用的 ring_desc 数据结构；
- 根据 RXCSR1 寄存器内容填充 ring_desc 的描述头；
- 检查 ring_desc 的描述头中的内容，如果发现错误，直接丢掉数据，退出中断处理。
- 将数据由寄存器中复制出来，复制到 ring_desc 中的数据区；
- 从数据中提取出 802.11 的 MAC 头；
- 由于接收到的数据可能是管理帧、控制帧或数据帧，所以需要检测 MAC 头，判断数据类型；
- 如果是管理帧、控制帧则送到管理控制子函数处理，如果是数据帧，则将 ring_desc 放回 RxRing 缓冲环；
- 退出中断处理程序，等待下一次的调用。

考虑到系统工作效率，放入 RxRing 缓冲行的已接收数据的处理，并没有在中断中进行，而是在中断外。先通过一个帧头转换，变成一个 802.3 的数据帧，然后填充到 sk_buff 数据结构中去，最后通过函数 netif_rx()把数据传送给上层进行进一步的处理。

（4）设备关闭模块驱动程序

设备关闭模块调用设备文件结构中的 stop 方法来实现，对无线网卡来说，就是由RT2500_close()函数实现操作，其作用与打开设备正好相反。

主要完成的工作包括：

- 停止 mlme 状态机（mlme 是一个主要用于维护各网卡连接状态的状态机）。删除 mlme 的描述符、队列、锁和相关定时器，使 mlme 任务不能再工作。
- 停止 AP 功能，释放相关资源。这里主要指和 AP 相关的资源，包括 MAC 地址表及 ASIC 的 LED 等，操作就是将其状态全部清空。
- 删除所有定时器。程序停止监视、RFTuning 等所有的系统定时器的工作。
- 释放中断。在 Linux 下，设备驱动程序对中断的释放是通过内核函数 free_irq( )（free_irq(net_dev->irq, net_dev);）来实现的。第一个参数是申请释放的中断号，对应于在打开设备时申请的中断号。为了保证中断释放的成功，在驱动程序中释放中断之前，终止系统设备产生的中断信号，主要通过“RTMP_IO_WRITE32(pAd, CSR8, 0x000FFFFF);”修改 CSR8（中断掩码寄存器）来实现。
- 重设 ASIC。主要是让 ASIC 恢复到在设备启动之前的初始状态。主要是对 TXCSR0、RXCSR0、CSR0、CSR1、CSR3、CSR4 的清 0 操作。
- 释放环行缓冲。为了发送和接收数据，在打开设备时分配了大量的环行缓冲区，这里要将其全部释放，其中主要就是 RxRing 和 TxRing。它们的大小会严重影响设备的性能，所以其具体的值需要反复测试以达到最佳。

（5）设备卸载模块驱动程序

在采用模块方式开发 Linux 下设备驱动程序时，当不打算使用该设备时。可以通过 rmmod 命令将设备模块从内核中卸载，以减少内核的大小及其他的资源开支。卸载设备模块与初始化设备模块是相对应的，rmmod 命令调用系统调用 delete_modeule，这个系统调用随后检查模块的使用计算，如果为 0 则调用模块本身的 cleanup_module( )函数，否则返回出错信息。cleanup_modeule( )函数对模块注册的每一项进行注销，其所做的工作与初始化设备模块相反。在无线网卡设备驱动程序中，其主要完成的工作包括：

- 注销网络设备。作为网络设备在卸载模块前，先要向系统申请注销这一设备，通常使用 unregister_netdev(net_dev)函数实现，net_dev 就是设备数据结构。
- 取消 CSR 基地址的映射。通过 iounmap 函数（iounmap((char *)(net_dev->base_addr));）取消对 I/O 空间地址的映射。
- 释放设备驱动程序占用的内存。在 Linux 下，针对不同申请内存的方式采用不同的释放内存方式。初始化设备模块时为 PCI 设备分配物理内存时，是通过 kmalloc 来获得内存的，释放内存则必须通过 kfree（）函数来完成。

6.3.5　rt2500 程序源代码

```
#include <Linux /kernel.h>
#include <Linux /module.h>
#include <Linux /init.h>
#include <Linux /pci.h>
#include <Linux /delay.h>
#include <asm/io.h>
#include "rt2x00.h"
```

```
    #include "rt2500pci.h"
    #ifdef DRV_NAME
    #undef DRV_NAME
    #define DRV_NAME            "rt2500pci"
    #endif /* DRV_NAME */
    /* 中断历程 rt2x00_interrupt_txdone 处理所有的发送数据包*/
    static void rt2x00_interrupt_txdone(struct _data_ring *ring)
    {
         struct _txd        *txd = NULL;
         u8         tx_result = 0x00;
         u8         retry_count = 0x00;
         do{
             txd = DESC_ADDR_DONE(ring);
             if(rt2x00_get_field32(txd->word0, TXD_W0_OWNER_NIC)
             || !rt2x00_get_field32(txd->word0, TXD_W0_VALID))
                  break;
             if(ring->ring_type == RING_TX){
                  tx_result = rt2x00_get_field32(txd->word0, TXD_W0_RESULT);
                  retry_count = rt2x00_get_field32(txd->word0, TXD_W0_RETRY_
COUNT);
                  rt2x00_update_stats(ring->device, STATS_TX_RESULT, tx_result);
                  rt2x00_update_stats(ring->device, STATS_TX_RETRY_COUNT, retry
_count);
             }
             rt2x00_set_field32(&txd->word0, TXD_W0_VALID, 0);
             rt2x00_ring_index_done_inc(ring);
         }while(!rt2x00_ring_empty(ring));
    }
    /* rt2x00_interrupt_rxdone 处理所有接收数据包*/
    static void rt2x00_interrupt_rxdone(struct _data_ring *ring)
    {
         struct _rt2x00_pci *rt2x00pci = rt2x00_priv(ring->device);
         struct _rxd        *rxd = NULL;
         void           *data = NULL;
         u16        size = 0x0000;
         u16        rssi = 0x0000;
         while(1){
             rxd = DESC_ADDR(ring);
             data = DATA_ADDR(ring);
             if(rt2x00_get_field32(rxd->word0, RXD_W0_OWNER_NIC))
                  break;
             size = rt2x00_get_field32(rxd->word0, RXD_W0_DATABYTE_COUNT);
             rssi = rt2x00_get_field32(rxd->word2, RXD_W2_RSSI);
             if(rt2x00_get_field32(rxd->word0, RXD_W0_CRC))
                  rt2x00_update_stats(ring->device, STATS_RX_CRC, 1);
             else if(rt2x00_get_field32(rxd->word0, RXD_W0_PHYSICAL_ERROR))
                  rt2x00_update_stats(ring->device, STATS_RX_PHYSICAL, 1);
             else if(rt2x00pci->sensitivity > rssi)
                  rt2x00_update_stats(ring->device, STATS_RX_QUALITY, 1);
             else
                  rt2x00_ring_rx_packet(ring->device, size, data, rssi);
             rt2x00_set_field32(&rxd->word0, RXD_W0_OWNER_NIC, 1);
             rt2x00_ring_index_inc(&rt2x00pci->rx);
         }
```

```
    }
    /*
     * 处理中断需要的一些步骤
     * 1. 获取中断源，保存到局部变量
     * 2. 写寄存器值清除中断
     * 3. 处理中断
     */
    static irqreturn_t rt2x00_interrupt(int irq, void *dev_instance, struct
pt_regs *regs)
    {
         struct _rt2x00_device   *device = (struct _rt2x00_device*)dev_instance;
         struct _rt2x00_pci *rt2x00pci = rt2x00_priv(device);
         u32        reg = 0x00000000;
         u8         ring_type = 0x0000;
         rt2x00_register_read(rt2x00pci, CSR7, &reg);
         rt2x00_register_write(rt2x00pci, CSR7, reg);
         if(!reg)
              return IRQ_NONE;
         if(rt2x00_get_field32(reg, CSR7_TBCN_EXPIRE)){/*Beacon 时间无效中断*/
              rt2x00_tx(device, RING_BEACON);
         }
         if(rt2x00_get_field32(reg, CSR7_RXDONE)){/* Rx ring 完成中断 t */
              rt2x00_interrupt_rxdone(&rt2x00pci->rx);
         }
         if(rt2x00_get_field32(reg, CSR7_TXDONE_ATIMRING)){/* Atim ring 传输完
成中断*/
              rt2x00_interrupt_txdone(&rt2x00pci->atim);
              ring_type |= RING_ATIM;
         }
         if(rt2x00_get_field32(reg, CSR7_TXDONE_PRIORING)){/* Priority ring
完成中断*/
              rt2x00_interrupt_txdone(&rt2x00pci->prio);
              ring_type |= RING_PRIO;
         }
         if(rt2x00_get_field32(reg, CSR7_TXDONE_TXRING)){   /* Tx ring 传输完成
中断 */
              rt2x00_interrupt_txdone(&rt2x00pci->tx);
              ring_type |= RING_TX;
         }
         if(ring_type)
              rt2x00_tx(device, ring_type);
        return IRQ_HANDLED;
    }
    * rt2x00_init_eeprom，初始化 EPROM */
    static  void  rt2x00_init_eeprom(struct  _rt2x00_pci  *rt2x00pci,  struct
_rt2x00_config *config)
    {
         u32    reg = 0x00000000;
         u16    eeprom = 0x0000;
         u16    val_a = 0x0000;
         u16    val_b = 0x0000;
         /*
          * 1 – 检测 EEPROM 位宽
```

```
         */
        rt2x00_register_read(rt2x00pci, CSR21, &reg);
        rt2x00pci->eeprom_width=rt2x00_get_field32(reg,CSR21_TYPE_93C46)?
EEPROM_WIDTH_93c46 ： EEPROM_WIDTH_93c66;
        /*
         * 2 – 获取 rf 标志
         */
        eeprom = rt2x00_eeprom_read_word(rt2x00pci, EEPROM_ANTENNA);
        set_chip(&rt2x00pci->chip,RT2560,rt2x00_get_field16(eeprom,EEPROM_
ANTENNA_RF_TYPE));
        /*
         * 3 – 获取默认天线配置
         */
        val_a = rt2x00_get_field16(eeprom, EEPROM_ANTENNA_TX_DEFAULT);
        val_b = rt2x00_get_field16(eeprom, EEPROM_ANTENNA_RX_DEFAULT);
        config->user.antenna_flags |= val_a;
        config->user.antenna_flags |= val_b << 8;
        if((config->user.antenna_flags & ANTENNA_TX) == 0)
             config->user.antenna_flags |= ANTENNA_TX_DIV;
        if((config->user.antenna_flags & ANTENNA_RX) == 0)
             config->user.antenna_flags |= ANTENNA_RX_DIV;
        /*
         * 4 – 默认的 geography 配置
         */
        eeprom = rt2x00_eeprom_read_word(rt2x00pci, EEPROM_GEOGRAPHY);
        config->user.geography = rt2x00_get_field16(reg, EEPROM_GEOGRAPHY_GEO);
        /*
         * 5 – 从 EEPROM 读取 BBP 数据，保存到私有结构体
         */
        memset(&rt2x00pci->eeprom, 0x00, sizeof(rt2x00pci->eeprom));
        for(eeprom = 0; eeprom < EEPROM_BBP_SIZE; eeprom++)
             rt2x00pci->eeprom[eeprom] = rt2x00_eeprom_read_word(rt2x00pci,
EEPROM_BBP_START + eeprom);
    }
    /*设备探测函数，在设备注册时被调用*/
    static  int  rt2x00_dev_probe(struct  _rt2x00_device  *device,  struct
_rt2x00_config *config, void *priv)
    {
        struct pci_dev*pci_dev = (struct pci_dev*)priv;
        struct _rt2x00_pci *rt2x00pci = rt2x00_priv(device);
        memset(rt2x00pci, 0x00, sizeof(*rt2x00pci));
        if(unlikely(!pci_dev)){
             ERROR("invalid priv pointer.\n");
             return -ENODEV;
        }
        rt2x00pci->pci_dev = pci_dev;
        rt2x00pci->rx.data_addr = NULL;
        rt2x00pci->tx.data_addr = NULL;
        rt2x00pci->atim.data_addr = NULL;
        rt2x00pci->prio.data_addr = NULL;
        rt2x00pci->beacon.data_addr = NULL;
        rt2x00pci->csr_addr = ioremap(pci_resource_start(pci_dev, 0), pci_
resource_len(pci_dev, 0));
        if(!rt2x00pci->csr_addr){
```

```
            ERROR("ioremap failed.\n");
            return -ENOMEM;
        }
        rt2x00_init_eeprom(rt2x00pci, config);
        rt2x00_dev_read_mac(rt2x00pci, device->net_dev);
        set_bit(DEVICE_CAP_802_11B, &device->flags);
        set_bit(DEVICE_CAP_802_11G, &device->flags);
        if(rt2x00_rf(&rt2x00pci->chip, RF5222))
            set_bit(DEVICE_CAP_802_11A, &device->flags);
        return 0;
    }
    static int rt2x00_dev_remove(struct _rt2x00_device *device)
    {
        struct _rt2x00_pci *rt2x00pci = rt2x00_priv(device);
        if(rt2x00pci->csr_addr){
            iounmap(rt2x00pci->csr_addr);
            rt2x00pci->csr_addr = NULL;
        }
        return 0;
    }
    /****************部分程序略************************/
    /*发送数据包*/
    static int rt2x00_dev_xmit_packet(struct _rt2x00_device *device, struct
sk_buff *skb, u8 ring_type, u16 rate, u16 xmit_flags)
    {
        struct _rt2x00_pci *rt2x00pci = rt2x00_priv(device);
        struct _data_ring*ring = NULL;
        struct _txd*txd = NULL;
        void*data = NULL;
        u32reg = 0x00000000;
        rt2x00_register_read(rt2x00pci, TXCSR0, &reg);
        if(ring_type == RING_TX){
            ring = &rt2x00pci->tx;
            rt2x00_set_field32(&reg, TXCSR0_KICK_TX, 1);
        }else if(ring_type == RING_PRIO){
            ring = &rt2x00pci->prio;
            rt2x00_set_field32(&reg, TXCSR0_KICK_PRIO, 1);
        }else if(ring_type == RING_ATIM){
            ring = &rt2x00pci->atim;
            rt2x00_set_field32(&reg, TXCSR0_KICK_ATIM, 1);
        }else if(ring_type == RING_BEACON){
            ring = &rt2x00pci->beacon;
        }
        if(skb){
            txd = DESC_ADDR(ring);
            data = DATA_ADDR(ring);
            if(rt2x00_get_field32(txd->word0, TXD_W0_OWNER_NIC)
            || rt2x00_get_field32(txd->word0, TXD_W0_VALID))
                  return -ENOMEM;
            memcpy(data, skb->data, skb->len);
            rt2x00_write_tx_desc(rt2x00pci, txd, skb->len, rate, skb->priority,
xmit_flags);
            rt2x00_ring_index_inc(ring);
        }
```

```
        if(xmit_flags & XMIT_START)
             rt2x00_register_write(rt2x00pci, TXCSR0, reg);
        return 0;
    }
    /*
     * PCI device 核心模块处理函数句柄，由指针指向这些函数
     */
    static struct _rt2x00_dev_handler rt2x00_pci_handler = {
        .dev_module            = THIS_MODULE,
        .dev_probe         = rt2x00_dev_probe,
        .dev_remove            = rt2x00_dev_remove,
        .dev_radio_on      = rt2x00_dev_radio_on,
        .dev_radio_off     = rt2x00_dev_radio_off,
        .dev_update_config = rt2x00_dev_update_config,
        .dev_update_stats  = rt2x00_dev_update_stats,
        .dev_test_tx       = rt2x00_dev_test_tx,
        .dev_xmit_packet   = rt2x00_dev_xmit_packet,
    };
    /*
     * PCI 驱动句柄
     */
    static int
    rt2x00_pci_probe(struct pci_dev *pci_dev, const struct pci_device_id *id)
    {
        struct net_device  *net_dev = NULL;
        intstatus = 0x00000000;
        if(id->driver_data != RT2560){
             ERROR("detected device not supported.\n");
             status = -ENODEV;
             goto exit;
        }
        if(pci_enable_device(pci_dev)){
             ERROR("enable device failed.\n");
             status = -EIO;
             goto exit;
        }
        pci_set_master(pci_dev);
        if(pci_set_mwi(pci_dev))
             NOTICE("MWI not available\n");
        if(pci_set_dma_mask(pci_dev, DMA_64BIT_MASK)
        && pci_set_dma_mask(pci_dev, DMA_32BIT_MASK)){
             ERROR("PCI DMA not supported\n");
             status = -EIO;
             goto exit_disable_device;
        }
        if(pci_request_regions(pci_dev, pci_name(pci_dev))){
             ERROR("PCI request regions failed.\n");
             status = -EBUSY;
             goto exit_disable_device;
        }
        net_dev = rt2x00_core_probe(&rt2x00_pci_handler, pci_dev, sizeof(struct
_rt2x00_pci), &pci_dev->dev);
        if(!net_dev){
             ERROR("net_device allocation failed.\n");
```

```
            status = -ENOMEM;
            goto exit_release_regions;
      }
      net_dev->irq = pci_dev->irq;
      pci_set_drvdata(pci_dev, net_dev);
      return 0;
exit_release_regions:
      pci_release_regions(pci_dev);
exit_disable_device:
      if(status != -EBUSY)
            pci_disable_device(pci_dev);
exit:
      return status;
}
/*从 PCI 卸载设备
static void rt2x00_pci_remove(struct pci_dev *pci_dev)
{
      struct net_device  *net_dev = pci_get_drvdata(pci_dev);
      rt2x00_core_remove(net_dev);
      pci_set_drvdata(pci_dev, NULL);
      pci_release_regions(pci_dev);
      pci_disable_device(pci_dev);
}
#ifdef CONFIG_PM
static int rt2x00_pci_suspend(struct pci_dev *pci_dev, pm_message_t state)
{
      struct net_device*net_dev = pci_get_drvdata(pci_dev);
      struct _rt2x00_device*device = rt2x00_device(net_dev);
      struct _rt2x00_pci*rt2x00pci = rt2x00_priv(device);
      u32reg = 0x00000000;
      if(!test_and_clear_bit(DEVICE_AWAKE, &device->flags)){
            NOTICE("Device already asleep.\n");
            return 0;
      }
      if(rt2x00_suspend(device))
            return -EBUSY;
      NOTICE("Going to sleep.\n");
      netif_device_detach(net_dev);
      reg = 0x00000000;
      rt2x00_set_field32(&reg, PWRCSR1_SET_STATE, 1);
      rt2x00_set_field32(&reg, PWRCSR1_BBP_DESIRE_STATE, 1);
      rt2x00_set_field32(&reg, PWRCSR1_RF_DESIRE_STATE, 1);
      rt2x00_set_field32(&reg, PWRCSR1_PUT_TO_SLEEP, 1);
      rt2x00_register_write(rt2x00pci, PWRCSR1, reg);
      pci_save_state(pci_dev);
      pci_disable_device(pci_dev);
      pci_set_power_state(pci_dev, pci_choose_state(pci_dev, state));
      return 0;
}
static int rt2x00_pci_resume(struct pci_dev *pci_dev)
{
      struct net_device*net_dev = pci_get_drvdata(pci_dev);
      struct _rt2x00_device*device = rt2x00_device(net_dev);
      struct _rt2x00_pci*rt2x00pci = rt2x00_priv(device);
```

```
        u32reg = 0x00000000;
        if(test_and_set_bit(DEVICE_AWAKE, &device->flags)){
              NOTICE("Device already awake.\n");
              return 0;
        }
        NOTICE("Waking up.\n");
        pci_set_power_state(pci_dev, PCI_D0);
        if(pci_enable_device(pci_dev)){
              ERROR("enable device failed.\n");
              return -EIO;
        }
        pci_restore_state(pci_dev);
        rt2x00_set_field32(&reg, PWRCSR1_SET_STATE, 1);
        rt2x00_set_field32(&reg, PWRCSR1_BBP_DESIRE_STATE, 3);
        rt2x00_set_field32(&reg, PWRCSR1_RF_DESIRE_STATE, 3);
        rt2x00_set_field32(&reg, PWRCSR1_PUT_TO_SLEEP, 0);
        rt2x00_register_write(rt2x00pci, PWRCSR1, reg);
        netif_device_attach(net_dev);
        return rt2x00_resume(device);
    }
    #endif /* CONFIG_PM */
    /*
     * RT2x00 PCI 模块信息
     */
    static char version[] = DRV_NAME " - " DRV_VERSION " (" DRV_RELDATE ") by
" DRV_PROJECT;
    static struct pci_device_id rt2x00_device_pci_tbl[] = {
        { PCI_DEVICE(0x1814, 0x0201), .driver_data = RT2560}, /*     Ralink
802.11g */
        {0,}
    };
    MODULE_AUTHOR(DRV_PROJECT);
    MODULE_VERSION(DRV_VERSION);
    MODULE_DESCRIPTION("Ralink RT2500 PCI & PCMCIA Wireless LAN driver.");
    MODULE_SUPPORTED_DEVICE("Ralink RT2560 PCI & PCMCIA chipset based cards");
    MODULE_DEVICE_TABLE(pci, rt2x00_device_pci_tbl);
    MODULE_LICENSE("GPL");
    #ifdef CONFIG_RT2X00_DEBUG
    module_param_named(debug, rt2x00_debug_level, bool, S_IWUSR | S_IRUGO);
    MODULE_PARM_DESC(debug, "Set this parameter to 1 to enable debug output.");
    #endif /* CONFIG_RT2X00_DEBUG */
    static struct pci_driver rt2x00_pci_driver = {
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 15)
        .owner= THIS_MODULE,
    #endif /* LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 15) */
        .name= DRV_NAME,
        .id_table= rt2x00_device_pci_tbl,
        .probe= rt2x00_pci_probe,
        .remove= __devexit_p(rt2x00_pci_remove),
    #ifdef CONFIG_PM
        .suspend   = rt2x00_pci_suspend,
        .resume= rt2x00_pci_resume,
    #endif /* CONFIG_PM */
    };
```

```
/*网卡 pci 初始化
static int __init rt2x00_pci_init(void)
{
     printk(KERN_INFO "Loading module： %s\n", version);
     return pci_register_driver(&rt2x00_pci_driver);
}
/*卸载设备驱动*/
static void __exit rt2x00_pci_exit(void)
{
     printk(KERN_INFO "Unloading module： %s\n", version);
     pci_unregister_driver(&rt2x00_pci_driver);
}
module_init(rt2x00_pci_init);
module_exit(rt2x00_pci_exit);
```

6.4　本章总结

本章首先介绍了网卡驱动的基础知识，然后以最常见的 8193 网卡和无线 usb 网卡 rt2500 为例，介绍了网卡设备驱动程序的设计过程。读者通过学习，可以了解 Ralink 无线网卡驱动实现的原理与具体过程。

第 7 章

显卡驱动程序开发

本章介绍 Linux 显卡驱动开发的原理与实现过程，首先介绍显卡驱动的一些基础知识。

7.1 显卡驱动概述

Linux 系统兼容大多数显卡。在桌面系统中显卡的配置正确与否，主要影响 X Window 的使用。在桌面 Linux 中，X Window 的主要配置文件是/etc/X11/XF86Config，X Window 正常工作的关键是使用的 X Server 与显卡相一致。设置 X Window 时，调用 Xconfigurator、XF86Setup 或 x86config 程序，用户可以利用这些程序方便地设置 X Window，而不需要手工修改 XF86Config 文件。

Xconfigurator 是很好的设置程序。使用 Xconfigurator 时，系统可以自动检测出显卡的类型，并且正确设置，如果没有检测到，用户可以在显卡列表中选择自己的显卡，设置程序就会正确地设置 X Server。除了设置 X Server 以外，设置显示器的分辨率对于 X 能否正常工作也是至关重要的。设置何种分辨率取决于显示器的类型。在设置程序中，如果显示器类型在列表中出现，那么选定它就可以了，如果没有出现，那么可以选择 custom 项（自定义模式），在随后的显示器列表中选择一款合适的显示器类型就可以了。以 Xconfigurator 设置程序为例，在它的列表中就可以选择分辨率和刷新率，用户可以参照显示器的技术指标来选择正确的项目。如果用户不知道显示器的性能指标，不妨从最低的性能开始试验，直到确定合适的显示器类型。

在设置了显示器类型后，就可以运行 start 程序启动 X Window。具体实现涉及以下 4 个方面。

- 一个支持 VESA framebuffer 的内核；
- 建立 framebuffer 设备；
- 配置的启动选项，使内核启动时能切换到指定的显示模式；
- XFree86 的 framebuffer Server（XF86_FBDev）以及在 XF86Config 中为其配置一个 Screen。

在用户开发常见的嵌入式系统中，Linux 显卡驱动以 framebuffer 为主，本章以 framebuffer 驱动为主要介绍对象。framebuffer 只是一个提供显示内存和显示芯片寄存器从物理内存映射到进程地址空间中的设备。所以对于应用程序而言，读者如果希望在 framebuffer 之上进行图形编程，还需要自己动手完成其他许多工作。

7.1.1 Linux framebuffer

framebuffer 设备在内核中作为显卡驱动模型，许多函数和数据结构都是特定的，为编

程提供了方便。

Source\Source\Documentation\fb 中有说明文件，对 framebuffer 设备驱动有详细说明，比较重要的文件是 framebuffer.txt 和 internals.txt，其他文件都是针对具体显卡芯片说明。

- framebuffer.txt：framebuffer 设备介绍。
- internals.txt：framebuffer 设备内部快速浏览。
- modedb.txt：关于视频模式的资料。
- aty128fb.txt：关于 ATI Rage128 显卡的 framebuffer 设备。
- clgenfb.txt：关于 Cirrus Logic 的显卡。
- matroxfb.txt：关于 Matrox 的显卡。
- pvr2fb.txt：关于 PowerVR 2 的显卡。
- tgafb.txt：关于 TGA（DECChip 21030）显卡。
- vesafb.txt：关于 VESA 显卡。

帧缓冲设备提供了显卡的抽象描述，同时代表了显卡上的显存，应用程序通过定义好的接口可以访问显卡，而不需要知道底层的任何操作。

该设备使用特殊的设备节点，通常位于/dev 目录，如/dev/fb*。

1．读写/dev/fb*

从用户的角度看，帧缓冲设备和其他位于/dev 下面的设备类似，是一个字符设备，主设备号是 29，次设备号定义帧缓冲的个数。通常使用如下方式（前面的数字代表次设备号）。

```
  0 = /dev/fb0    First frame buffer
  1 = /dev/fb1    Second frame buffer
...
  31 = /dev/fb31  32nd frame buffer
```

考虑到向下兼容，可以创建符号链接。

```
/dev/fb0current -> fb0
/dev/fb1current -> fb1
```

帧缓冲设备也是一种普通的内存设备，可以读写其内容。例如，对屏幕抓屏如下代码所示。

```
cp /dev/fb0 myfile
```

也可以同时有多个显示设备，例如主板上除了内置的显卡外还有另一个独立的显卡。对应的帧缓冲设备（/dev/fb0 and /dev/fb1 etc.）可以独立工作。应用程序（如 X Server）一般使用/dev/fb0 作为默认的显示帧缓冲区。可以自定把某个设备作为默认的帧缓冲设备，设置$FRAMEBUFFER 环境变量即可。

在 sh/bash 中：

```
export FRAMEBUFFER=/dev/fb1
```

在 csh 中：

```
setenv FRAMEBUFFER /dev/fb1
```

设定后，X server 将使用第二个帧缓冲区设备。

2．/dev/fb*原理

一个帧缓冲设备和内存设备/dev/mem 类似，并且有许多共性。可以使用 read、write、seek 以及 mmap()。不过仅仅是帧缓冲的内存，不是所有的内存区，是显卡专用的那部分

内存。/dev/fb*允许对 ioctl 进行各种操作，通过 ioctl 可以读取或设定设备参数。颜色映射表也是通过 ioctl 设定。查看<Linux/fb.h>就知道有多少 ioctl 应用以及相关数据结构。

用户可以获取设备一些不变的信息，如设备名，屏幕的组织（平面、像素等）对应内存区的长度和起始地址，也可以获取能够发生变化的信息，例如位深、颜色格式、时序等。如果改变这些值，驱动程序将对值进行优化，以满足设备特性（返回 EINVAL，如果设备不支持），也可以获取或设定部分颜色表。所有这些特性让应用程序十分容易地使用设备。X Server 可以使用/dev/fb*而不需要知道硬件的寄存器是如何组织的。

XF68_FBDev 是一个用于位映射（单色）的 X Server，唯一要做的就是在应用程序相应的位置设定是否显示。在新内核中，帧缓冲设备可以工作于模块中，允许动态加载。这类驱动必须调用 register_framebuffer()在系统中注册，使用模块更方便。

3．帧缓冲分辨率设定

帧缓冲的分辨率可以用工具 fbset 设定，可以改变视频设备的显示模式，主要就是改变当前视频模式，如在启动过程中，在/etc/rc.* 或 /etc/init.d/* 文件中调用，可以把视频模式从单色显示变成真彩。fbset 使用存储在配置文件中的视频模式数据表，可以在文件中增加自己需要的显示模式。

4．X Server

X Server (XF68_FBDev)是对帧缓冲设备的最主要应用。从 XFree86 3.2 后，X Server 就是 XFree86 的一部分，有 2 个工作模式。

（1）在/etc/XF86Config 文件中，如果'Display'段关于 'fbdev'的配置如下：

```
Modes "default"
```

X Server 将使用前面讨论的从环境变量$FRAMEBUFFER 获取当前帧缓冲设备。

其次，用户也可以设定颜色位深，使用 Depth 关键字，使用 Virtual 设定虚拟分辨率，这也是默认设置。

（2）也可以通过设定/etc/XF86Config 改变分辨率。唯一的不足就是必须设定刷新频率。可以使用 fbset –x 通过 fbset 或 xvidtune 切换显示模式。

5．视频模式频率

CRT 显示器是用 3 个电子枪轰击磷粉完成颜色的显示的。电子枪从左到右水平扫描，并从上至下垂直扫描。通过改变枪的电压，所显示的颜色可以不同。当电子枪完成一行扫描重新回到下一行的开始，被称做“水平折回”。当一屏幕全部扫描完毕，电子枪将回到最左上角，被称为“垂直折回”。在折回的途中电子枪是关闭的。电子枪打点的移动速度取决于点时钟。如果点时钟是 28.37516 MHz，打一个点需要 35242ps。

$$1/(28.37516E6\ Hz) = 35.242E{-}9\ s$$

如果屏幕分辨率是 640×480，那么一行的时间是：

$$640*35.242E{-}9\ s = 22.555E{-}6\ s$$

然而水平折回也是需要时间的，通常 272 个打点时间，因此一行总共需要：

$$(640+272)*35.242E{-}9\ s = 32.141E{-}6\ s$$

所以认为水平扫描的频率是 31KHz。

$$1/(32.141E{-}6\ s) = 31.113E3\ Hz$$

一屏幕含有 480 行，加上垂直折回时间 49，一屏所需的时间是：

$$(480+49)*32.141E{-}6\ s = 17.002E{-}3\ s$$

所以认为垂直扫描的频率是 59Hz。

$$1/(17.002E{-}3\ s) = 58.815\ Hz$$

这也意味着屏幕数据每秒钟刷新 59 次。为了得到稳定的图像显示效果，VESA 垂直扫描频率不低于 72Hz。但是也因人而异，有些人 50Hz 感觉不到任何问题，有些至少在 80Hz 以上才可以。

由于显示器不知道什么时候新行开始扫描，显卡为每一行扫描提供水平同步信号。类似地，也为每一帧显示提供垂直同步信号。图像在屏幕上点的位置取决于这些同步信号的发生时刻。

图 7-1 给出了所有时序的概要。水平折回的时间就是左边空白＋右边空白＋水平同步长度。垂直折回的时间就是上空白＋下空白＋垂直同步长。

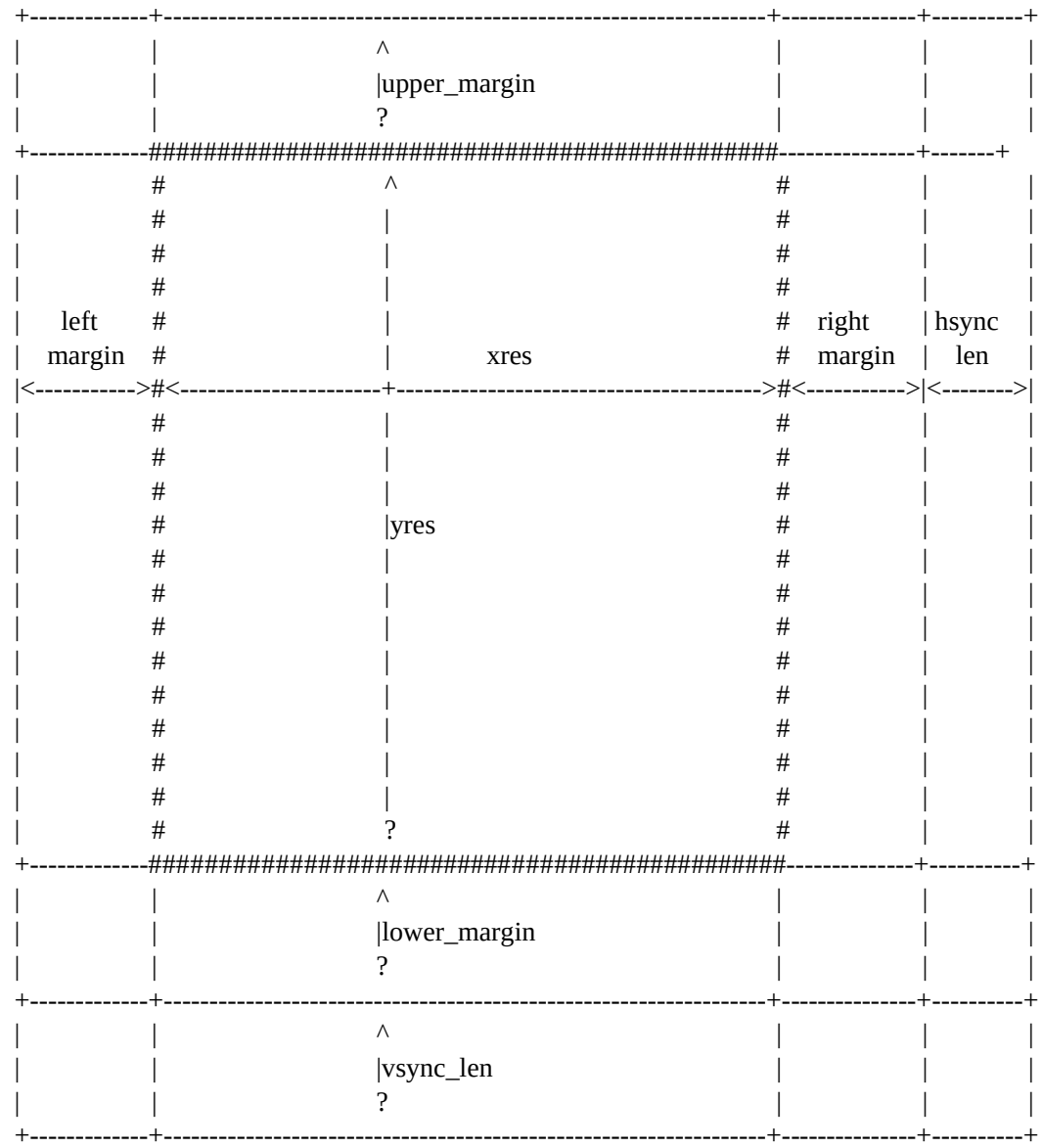


图 7-1　时序的概要

6．把 XFree86 时序变成 framebuffer device 时序

典型的显示模式为：

```
"800x600"     50     800  856  976 1040    600  637  643  666
< name >     DCF      HR  SH1  SH2  HFL     VR  SV1  SV2  VFL
```

而帧缓冲设备使用下面的参数：

```
- pixclock： 点时钟 in ps (pico seconds)
- left_margin： time from sync to picture
- right_margin： time from picture to sync
- upper_margin： time from sync to picture
- lower_margin： time from picture to sync
- hsync_len： length of horizontal sync
- vsync_len： length of vertical sync
```

（1）Pixelclock：

```
xfree： in MHz
fb： in picoseconds (ps)
pixclock = 1000000 / DCF
```

（2）horizontal timings：

```
left_margin = HFL - SH2
right_margin = SH1 - HR
hsync_len = SH2 - SH1
```

（3）vertical timings：

```
upper_margin = VFL - SV2
lower_margin = SV1 - VR
vsync_len = SV2 - SV1
```

更好的 VESA 的例子可以在 XFree86 的源码中找到，“xc/programs/Xserver/hw/xfree86/doc/modeDB.txt”。

要获取更多关于帧缓冲设备以及应用的参考，请访问 http：//Linux-fbdev. sourceforge.net/或者查阅下面的文档。

- The manual pages for fbset：fbset(8), fb.modes(5)
- The manual pages for XFree86：XF68_FBDev(1), XF86Config(4/5)
- The mighty kernel sources：
- Linux /drivers/video/
- Linux /include/Linux /fb.h
- Linux /include/video/

7.1.2 帧缓冲设备数据结构

在开始写驱动之前，需要详细阅读如下文件：

```
\Documentation\fb 目录
 vesafb.txt，matroxfb.txt，sa1100fb.txt
\drivers\video 目录
          fbmem.c，fbgen.c，fbmon.c，fbcmap.c
```

skeletonfb.c vesafb.c，sa1100fb.c，sa1100fb.h
include\Linux 目录 fb.h

最值得关注的是 skeletonfb.c，本小节会给出一个 fb device 驱动的框架下数据结构在帧缓冲设备的使用，定义在<Linux /fb.h>。

1．Kernel 外部（用户空间）

- struct fb_fix_screeninfo

帧缓冲设备中设备无关的常值数据信息。可以通过 Ioctl 的 FBIOGET_FSCREENINFO 获取。

- struct fb_var_screeninfo

帧缓冲设备中设备无关的变量数据信息和特定的显示模式。可以通过 iotcl 的 FBIOGET_VSCREENINFO 获取，并通过 ioctl 的 FBIOPUT_VSCREENINFO 设定。还有 FBIOPAN_DISPLAY 可以用。

- struct fb_cmap

设备无关的颜色表信息。可以通过 ioctl 的 FBIOGETCMAP 和 FBIOPUTCMAP 读取或设定。

2．Kernel 内部

- struct fb_info

常规信息，API 以及帧缓冲设备的底层信息（主板地址等）。

- struct 'par'

唯一指定该设备的显示模式的设备相关信息。

- struct display

帧缓冲设备和控制台驱动之间的接口。

3．常用的帧缓冲 API

Monochrome (FB_VISUAL_MONO01 and FB_VISUAL_MONO10)

每个像素是黑或白。

Pseudo color (FB_VISUAL_PSEUDOCOLOR and FB_VISUAL_STATIC_PSEUDOCOLOR)

索引颜色显示。

True color (FB_VISUAL_TRUECOLOR)

真彩显示，分成红绿蓝三基色。

Direct color (FB_VISUAL_DIRECTCOLOR)

每个像素颜色也由红绿蓝组成，不过每个颜色值都是索引，需要查表。

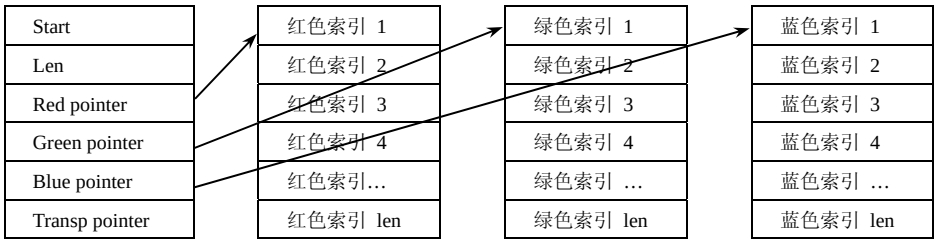
Grayscale displays

灰度显示，红绿蓝的值都一样。

下面是/Linux /fb.h 的部分代码与注释。

```
#ifndef _LINUX_FB_H
#define _LINUX_FB_H
#nclude <Linux /tty.h>
#include <asm/types.h>
/* Definitions of frame buffers                    */
 #define FB_MAJOR       29  /*主设备号*/
#define FB_MAX         32  /* sufficient for now */
/* ioctls
   0x46 is 'F'                           */
#define FBIOGET_VSCREENINFO  0x4600
#define FBIOPUT_VSCREENINFO  0x4601
#define FBIOGET_FSCREENINFO  0x4602
#define FBIOGETCMAP          0x4604
#define FBIOPUTCMAP          0x4605
#define FBIOPAN_DISPLAY      0x4606
#define FBIOGET_CON2FBMAP    0x460F
#define FBIOPUT_CON2FBMAP    0x4610
#define FBIOBLANK      0x4611
#define FBIOGET_VBLANK       _IOR('F', 0x12, struct fb_vblank)
#define FBIO_ALLOC           0x4613
#define FBIO_FREE            0x4614
#define FBIOGET_GLYPH        0x4615
#define FBIOGET_HWCINFO      0x4616
#define FBIOPUT_MODEINFO     0x4617
#define FBIOGET_DISPINFO     0x4618
#define FB_TYPE_PACKED_PIXELS      0
#define FB_TYPE_PLANES             1
#define FB_TYPE_INTERLEAVED_PLANES 2
#define FB_TYPE_TEXT               3
#define FB_TYPE_VGA_PLANES         4
#define FB_AUX_TEXT_MDA            0
#define FB_AUX_TEXT_CGA            1
#define FB_AUX_TEXT_S3_MMIO        2
#define FB_AUX_TEXT_MGA_STEP16     3
#define FB_AUX_TEXT_MGA_STEP8      4
#define FB_AUX_VGA_PLANES_VGA4         0   /* 16 color planes (EGA/VGA) */
#define FB_AUX_VGA_PLANES_CFB4         1   /* CFB4 in planes (VGA) */
#define FB_AUX_VGA_PLANES_CFB8         2   /* CFB8 in planes (VGA) */
#define FB_VISUAL_MONO01               0   /* Monochr. 1=Black 0=White */
#define FB_VISUAL_MONO10               1   /* Monochr. 1=White 0=Black */
#define FB_VISUAL_TRUECOLOR            2   /* 真彩色*/
#define FB_VISUAL_PSEUDOCOLOR          3   /* Pseudo color (like atari) */
#define FB_VISUAL_DIRECTCOLOR          4   /* Direct color */
#define FB_VISUAL_STATIC_PSEUDOCOLOR   5   /* Pseudo color readonly */
 ..................................................
 #define FB_ACCEL_3DLABS_PERMEDIA3 37
/*不可修改的屏幕信息，用户空间可见*/
struct fb_fix_screeninfo {
   char id[16];
   unsigned long smem_start;
   __u32 smem_len;                 /*显存的大小 */
   __u32 type;
   __u32 type_aux;
   __u32 visual;
   __u16 xpanstep;
```

```
    __u16 ypanstep;
    __u16 ywrapstep;
    __u32 line_length;              /*每行的字节数  */
    unsigned long mmio_start;
    __u32 mmio_len;                 /* Memory Mapped I/O 长度  */
    __u32 accel;                    /* 可用的加速 */
    __u16 reserved[3];
};
 /*像素所占字节内各个颜色的位分配比如 RGB=888、565、555 等*/
struct fb_bitfield {
    __u32 offset;                   /* beginning of bitfield */
    __u32 length;                   /* length of bitfield      */
    __u32 msb_right;                /* != 0 :  Most significant bit is */
                                    /* right */
};
/*下面的宏不常用*/
#define FB_NONSTD_HAM           1   /* Hold-And-Modify (HAM)       */
#define FB_ACTIVATE_NOW         0   /* set values immediately (or vbl)*/
#define FB_ACTIVATE_NXTOPEN     1   /* activate on next open    */
#define FB_ACTIVATE_TEST        2   /* don't set, round up impossible */
#define FB_ACTIVATE_MASK        15  /* values          */
#define FB_ACTIVATE_VBL         16  /* activate values on next vbl  */
#define FB_CHANGE_CMAP_VBL      32  /* change colormap on vbl   */
#define FB_ACTIVATE_ALL         64  /* change all VCs on this fb    */
#define FB_ACCELF_TEXT          1   /* text mode acceleration */
#define FB_SYNC_HOR_HIGH_ACT    1   /* horizontal sync high active  */
#define FB_SYNC_VERT_HIGH_ACT   2   /* vertical sync high active    */
#define FB_SYNC_EXT             4   /* external sync       */
#define FB_SYNC_COMP_HIGH_ACT   8   /* composite sync high active   */
#define FB_SYNC_BROADCAST       16  /* broadcast video timings
struct fb_cmap
/*颜色映射表*/
struct fb_cmap {
    __u32 start;            /* First entry  */
    __u32 len;              /* Number of entries */
    __u16 *red;             /* 红色  */
    __u16 *green;           /*绿色*/
    __u16 *blue;            /*蓝色*/
    __u16 *transp;          /* 透明度，允许 NULL */
};
```



该结构在 fb.h 文件中定义，在 struct fb_ops 结构中有两个成员函数与其相关。

```
/*获取颜色表*/
int (*fb_get_cmap)(struct fb_cmap *cmap, int kspc, int con, struct fb_info *info);
/*设定颜色表*/
int (*fb_set_cmap)(struct fb_cmap *cmap, int kspc, int con, struct fb_info *info);
```

在 struct fb_info 结构中有变量：

```
struct fb_cmap cmap;              /* Current cmap */
```

在 fpgen 基础操作下提供：

```
    extern int fbgen_get_cmap(struct fb_cmap *cmap, int kspc, int con, struct
fb_info *info);
    extern int fbgen_set_cmap(struct fb_cmap *cmap, int kspc, int con, struct
fb_info *info);
    在文件/* drivers/video/fbcmap.c */中提供更多的 cmap 应用。
    extern int fb_alloc_cmap(struct fb_cmap *cmap, int len, int transp);
    extern void fb_copy_cmap(struct fb_cmap *from, struct fb_cmap *to, int
fsfromto);
    extern int fb_get_cmap(struct fb_cmap *cmap, int kspc,int (*getcolreg)(u_int,
u_int *, u_int *, u_int *,u_int *, struct fb_info *),
                           struct fb_info *fb_info);
    extern int fb_set_cmap(struct fb_cmap *cmap, int kspc,int (*setcolreg)(u_int,
u_int, u_int, u_int, u_int,struct fb_info *),
                          struct fb_info *fb_info);
    extern struct fb_cmap *fb_default_cmap(int len);
    extern void fb_invert_cmaps(void);
```

Fbcon **中的颜色查找表：**Fbcon-cfbx 表示该 console 使用的是 xbpp 颜色描述。颜色数为 2^x。

在此仅以 x=8,x=24 举例，使用颜色分别是 256 色和真彩 16M。

```
/driver/video/fbcon-cfb8.c
/driver/video/fbcon-cfb24.c
/include/video/fbcon-cfb8.h
/include/video/fbcon-cfb24.h
```

这 4 个文件实现具体的操作，而 fbcon 的底层操作请参考前面的 fbcon 的介绍。

实现 fbcon 的颜色映射只需完成下面的功能，以 fb8 为例。

```
    struct display_switch fbcon_cfb8;
    void fbcon_cfb8_setup(struct display *p);
    void fbcon_cfb8_bmove(struct display *p, int sy, int sx, int dy, int dx,
int height, int width);
    void fbcon_cfb8_clear(struct vc_data *conp, struct display *p, int sy, int
sx, int height, int width);
    void fbcon_cfb8_putc(struct vc_data *conp, struct display *p, int c, int
yy, int xx);
    void fbcon_cfb8_putcs(struct vc_data *conp, struct display *p, const unsigned
short *s, int count, int yy, int xx);
    void fbcon_cfb8_revc(struct display *p, int xx, int yy);
    void fbcon_cfb8_clear_margins(struct vc_data *conp, struct display *p,int
bottom_only);
```

fbcon_cfb8 是系统的实现关键，具体解释参考 fbcon 介绍。

fbcon_cfb8_setup 函数完成设定 display 结构中 next_line 和 next_palne 的值。

fbcon_cfb8_bmove 函数完成当前坐标的移动。

fbcon_cfb8_clear 函数通过调用 rectfill 函数清屏幕缓冲区。

fbcon_cfb8_putc 函数向屏幕输出单字符，字体宽度必须小于等于 16。

fbcon_cfb8_putcs 函数向屏幕输出字符串。
fbcon_cfb8_revc 函数从屏幕输入单个字符，并回显到 fb 上。
fbcon_cfb8_clear_margins 函数和 fbcon_cfb8_clear 类似，调用 rectfill 清除区域。
其中，fb_writel 函数和 fb_readl 函数实现输入输出的底层操作。

7.2　典型实例——显卡 Framebuffer 驱动实现

Framebuffer 驱动结构框架如图 7-2 所示。

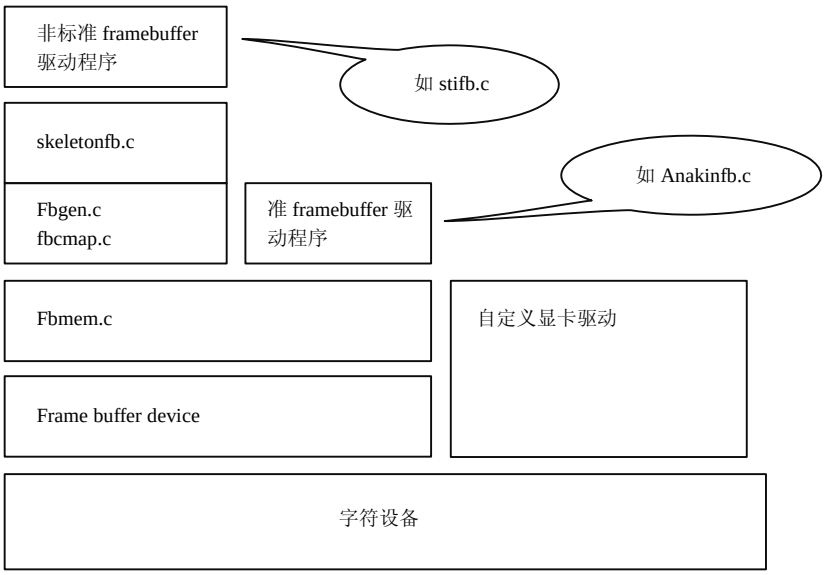


图 7-2　Framebuffer 驱动结构框架

读者在开发驱动程序时可以参考 skeletonfb.c。

7.2.1　Framebuffer 驱动框架程序

```
#include <Linux /module.h>
#include <Linux /kernel.h>
#include <Linux /errno.h>
#include <Linux /string.h>
#include <Linux /mm.h>
#include <Linux /slab.h>
#include <Linux /delay.h>
#include <Linux /fb.h>
#include <Linux /init.h>
/*这个结构定义了硬件的状态*/
struct xxx_par;
/*
 *这里定义默认的 structs fb_fix_screeninfo 和 fb_var_screeninfo
 */
static struct fb_fix_screeninfo xxxfb_fix __initdata = {
    .id ="FB's name",
    .type =FB_TYPE_PACKED_PIXELS,
```

```
        .visual =FB_VISUAL_PSEUDOCOLOR,
        .xpanstep =1,
        .ypanstep =1,
        .ywrapstep =1,
        .accel =FB_ACCEL_NONE,
    };
       /* 如果驱动支持多个 framebuffer，需要使用数组，或使用 framebuffer_alloc()动态
分配，使用 framebuffer_release()释放内存*/
     static struct fb_info info;
       /*
        * 硬件的状态
        */
    static struct xxx_par __initdata current_par;
    int xxxfb_init(void);
    int xxxfb_setup(char*);
    /*  xxxfb_open——可选的函数；当 framebuffer 第一次使用调用*/
    static int xxxfb_open(const struct fb_info *info, int user)
    {
       return 0;
    }
    /*  xxxfb_release——可选的函数 */
    static int xxxfb_release(const struct fb_info *info, int user)
    {
       return 0;
    }
    /**
     *     xxxfb_check_var——可选，验证 var 传入
     *     @var：frame buffer variable screen 结构
     *     @info：frame buffer structure 代表单一 frame buffer
     */
    static int xxxfb_check_var(struct fb_var_screeninfo *var, struct fb_info
*info)
    {
       /* ... */
       return 0;
    }
    static int xxxfb_set_par(struct fb_info *info)
    {
       struct xxx_par *par = info->par;
       /* ... */
       return 0;
    }
    /*xxxfb_setcolreg——可选函数，设置颜色寄存器 */
    static int xxxfb_setcolreg(unsigned regno, unsigned red, unsigned green,
                  unsigned blue, unsigned transp,
                  const struct fb_info *info)
    {
       if (regno >= 256)  /* no. of hw registers */
         return -EINVAL;
        if (info->var.grayscale) {
         /* grayscale = 0.30*R + 0.59*G + 0.11*B */
         red = green = blue = (red * 77 + green * 151 + blue * 28) >> 8;
       }
```

```
     /*颜色转换成硬件可接受的形式 */
#define CNVT_TOHW(val,width) ((((val)<<(width))+0x7FFF-(val))>>16)
    red = CNVT_TOHW(red, info->var.red.length);
    green = CNVT_TOHW(green, info->var.green.length);
    blue = CNVT_TOHW(blue, info->var.blue.length);
    transp = CNVT_TOHW(transp, info->var.transp.length);
#undef CNVT_TOHW
    if (info->fix.visual == FB_VISUAL_DIRECTCOLOR ||
     info->fix.visual == FB_VISUAL_TRUECOLOR)
        write_{red|green|blue|transp}_to_clut();
    if (info->fix.visual == FB_VISUAL_TRUECOLOR ||
     info->fix.visual == FB_VISUAL_DIRECTCOLOR) {
        u32 v;
        if (regno >= 16)
            return -EINVAL;
        v = (red << info->var.red.offset) |
            (green << info->var.green.offset) |
            (blue << info->var.blue.offset) |
            (transp << info->var.transp.offset);
        ((u32*)(info->pseudo_palette))[regno] = v;
    }
    /* ... */
    return 0;
}
/* xxxfb_pan_display——可选函数*/
static int xxxfb_pan_display(struct fb_var_screeninfo *var,
                const struct fb_info *info)
{
     /* ... */
    return 0;
}
/* ------------加速函数---------------------- */
/*如果有硬件加速，使用自己的函数，如果没有使用 generic unaccelerated 函数*/
/*xxxfb_fillrect——REQUIRED function 画一个矩形*/
void xxfb_fillrect(struct fb_info *p, const struct fb_fillrect *region)
{
/* 包括 X、Y、WIDTH、HEIGHT、COLOR 等*/
}
/* xxxfb_copyarea——REQUIRED function.内存复制函数，复制一个矩形区域*/
void xxxfb_copyarea(struct fb_info *p, const struct fb_copyarea *area)
{
/*
 *      @dx：@dy：目的区域位置
 *      @width：要复制的宽度
 *      @height：要复制的高度
 *      @sx：@sy：屏幕源区域位置
 */
}
/*xxxfb_imageblit——REQUIRED function.从系统内存复制一个图片到屏幕上*/
void xxxfb_imageblit(struct fb_info *p, const struct fb_image *image)
{
/*
 *      @dx：@dy：目的区域位置
```

```
 *      @width：要复制的宽度
 *      @height：要复制的高度
 *      @fg_color：单色位图的颜色数据
 *      @bg_color：写入 framebuffer 的前景色和背景色
 *      @depth：位深度
 *      @data：图像数据
 *      @cmap：颜色表
 */
}
/*xxxfb_cursor——可选的函数*/
int xxxfb_cursor(struct fb_info *info, struct fb_cursor *cursor)
{
}
/*可选的函数*/
void xxxfb_rotate(struct fb_info *info, int angle)
{
}
/*可选的函数*/
void xxxfb_poll(struct fb_info *info, poll_table *wait)
{
}
/*可选的函数*/
void xxxfb_sync(struct fb_info *info)
{
}
    /*
     *  系统初始化时调用
     */
/* static int __init xxfb_probe (struct device *device)——系统平台设备 */
static int __init xxxfb_probe(struct pci_dev *dev,
                 const_struct pci_device_id *ent)
{
    struct fb_info *info;
    struct xxx_par *par;
    struct device = &dev->dev; /* for pci drivers */
    int cmap_len, retval;
     /*
     * 动态分配 info 和 par
     */
    info = framebuffer_alloc(sizeof(struct xxx_par), device);
    if (!info) {
        /* goto error path */
    }
    par = info->par;
    info->screen_base = framebuffer_virtual_memory;
    info->fbops = &xxxfb_ops;
    info->fix = xxxfb_fix;
info->pseudo_palette = pseudo_palette;
/*设置标志指示能提供的加速(pan/wrap/copyarea/etc.) 部分选项*/
     /* FBINFO_HWACCEL_COPYAREA——hardware 移动*/
     /* FBINFO_HWACCEL_FILLRECT——hardware 填充*/
    info->flags = FBINFO_DEFAULT;
    info->pixmap.addr = kmalloc(PIXMAP_SIZE, GFP_KERNEL);
```

```
        if (!info->pixmap.addr) {
            /* goto error */
        }
    /*下面是一些可选内容*/
        info->pixmap.size = PIXMAP_SIZE;
        info->pixmap.flags = FB_PIXMAP_SYSTEM;
        info->pixmap.scan_align = 4;
        info->pixmap.buf_align = 4;
        info->pixmap.scan_align = 32
        /*
         * 这里应该给一个适当的视频模式*/
        if (!mode_option)
         mode_option = "640x480@60";
        retval = fb_find_mode(info->var, info, mode_option, NULL, 0, NULL, 8);
        if (!retval || retval == 4)
         return -EINVAL;
        /* 这里需要用户实现*/
        fb_alloc_cmap(info->cmap, cmap_len, 0);
         info->var = xxxfb_var;
        xxxfb_check_var(&info->var, info);
        /*
         * 在注册 framebuffer 前是否调用 fb_set_par()依赖特定的平台，x86 系统有一个 VGA
core；调用 set_par() 会破坏 VGA console,这里不需要调用*/
        /* xxxfb_set_par(info); */
        if (register_framebuffer(info) < 0)
         return -EINVAL;
        printk(KERN_INFO "fb%d： %s frame buffer device\n", info->node,
           info->fix.id);
        pci_set_drvdata(dev, info); /* or dev_set_drvdata(device, info) */
        return 0;
    }
    /* 静态 void __exit xxxfb_remove(struct device *device)，Cleanup 使用 */
    static void __exit xxxfb_remove(struct pci_dev *dev)
    {
        struct fb_info *info = pci_get_drv_data(dev);
        /* 或使用 dev_get_drv_data(device); */
        if (info) {
            unregister_framebuffer(info);
            fb_dealloc_cmap(&info.cmap);
            framebuffer_release(info);
        }
        return 0;
    }
    #if CONFIG_PCI
    /* 配置为 PCI 驱动 */
    static struct pci_driver xxxfb_driver = {
        .name ="xxxfb",
        .id_table = xxxfb_devices,
        .probe =xxxfb_probe,
        .remove =__devexit_p(xxxfb_remove),
        .suspend =xxxfb_suspend, /* optional */
        .resume =xxxfb_resume,  /* optional */
    };
```

```
static int __init xxxfb_init(void)
{
    /*
     *  内核引导时的选项(in 'video=xxxfb：<options>' format)
     */
#ifndef MODULE
    char *option = NULL;
    if (fb_get_options("xxxfb", &option))
        return -ENODEV;
    xxxfb_setup(option);
#endif
    return pci_register_driver(&xxxfb_driver);
}
static void __exit xxxfb_exit(void)
{
    pci_unregister_driver(&xxxfb_driver);
}
#else
#include <Linux /platform_device.h>
/*平台相关设备 */
static struct device_driver xxxfb_driver = {
    .name = "xxxfb",
    .bus  = &platform_bus_type,
    .probe = xxxfb_probe,
    .remove = xxxfb_remove,
    .suspend = xxxfb_suspend, /* 可选 */
    .resume = xxxfb_resume,   /*可选 */
};
static struct platform_device xxxfb_device = {
    .name = "xxxfb",
};
static int __init xxxfb_init(void)
{
    int ret;
    /*
     *  内核引导选项(in 'video=xxxfb：<options>' format)
     */
#ifndef MODULE
    char *option = NULL;
    if (fb_get_options("xxxfb", &option))
        return -ENODEV;
    xxxfb_setup(option);
#endif
    ret = driver_register(&xxxfb_driver);
    if (!ret) {
        ret = platform_device_register(&xxxfb_device);
        if (ret)
            driver_unregister(&xxxfb_driver);
    }
    return ret;
}
static void __exit xxxfb_exit(void)
{
    platform_device_unregister(&xxxfb_device);
```

```
    driver_unregister(&xxxfb_driver);
}
#endif
/*  fb_setup 当驱动有特别的选项时一般使用默认的 generic fb_setup() */
int __init xxxfb_setup(char *options)
{
   /* 解析用户特定选项('video=xxxfb: ') */
}
   /*
    *  Frame buffer 操作，指向特定的设备驱动函数
    */
static struct fb_ops xxxfb_ops = {
    .owner= THIS_MODULE,
    .fb_open= xxxfb_open,
    .fb_read= xxxfb_read,
    .fb_write= xxxfb_write,
    .fb_release= xxxfb_release,
    .fb_check_var= xxxfb_check_var,
    .fb_set_par= xxxfb_set_par,
    .fb_setcolreg= xxxfb_setcolreg,
    .fb_blank= xxxfb_blank,
    .fb_pan_display= xxxfb_pan_display,
    .fb_fillrect= xxxfb_fillrect,
    .fb_copyarea= xxxfb_copyarea,
    .fb_imageblit= xxxfb_imageblit,
    .fb_cursor= xxxfb_cursor,       /* 可选 */
    .fb_rotate= xxxfb_rotate,
    .fb_poll= xxxfb_poll,
    .fb_sync= xxxfb_sync,
    .fb_ioctl= xxxfb_ioctl,
    .fb_mmap= xxxfb_mmap,
};
module_init(xxxfb_init);
module_exit(xxxfb_cleanup);
MODULE_LICENSE("GPL");
```

7.2.2　NVDIA 显卡设备驱动文件

Nvdia 显卡特定驱动主要有以下几个文件：

- nvidia.c
- nv_hw.c nv_setup.c
- nv_accel.c
- nv_i2c.c
- nv_backlight.c

驱动程序概述如下代码所示。

```
/*Nv 显卡的设备驱动*/
static struct pci_driver nvidiafb_driver = {
    .name = "nvidiafb",
    .id_table = nvidiafb_pci_tbl,
    .probe= nvidiafb_probe,
    .suspend= nvidiafb_suspend,
```

```
    .resume= nvidiafb_resume,
    .remove= __exit_p(nvidiafb_remove),
};
/*模块初始化调用*/
static int __devinit nvidiafb_init(void)
{
#ifndef MODULE
    char *option = NULL;
    if (fb_get_options("nvidiafb", &option))
        return -ENODEV;
    nvidiafb_setup(option);
#endif
    return pci_register_driver(&nvidiafb_driver);
}
module_init(nvidiafb_init);
#ifdef MODULE
static void __exit nvidiafb_exit(void)
{
    pci_unregister_driver(&nvidiafb_driver);
}
module_exit(nvidiafb_exit);
static struct fb_ops nvidia_fb_ops = {
    .owner= THIS_MODULE,
    .fb_check_var= nvidiafb_check_var,
    .fb_set_par= nvidiafb_set_par,
    .fb_setcolreg= nvidiafb_setcolreg,
    .fb_pan_display= nvidiafb_pan_display,
    .fb_blank= nvidiafb_blank,
    .fb_fillrect= nvidiafb_fillrect,
    .fb_copyarea= nvidiafb_copyarea,
    .fb_imageblit= nvidiafb_imageblit,
    .fb_cursor= nvidiafb_cursor,
    .fb_sync= nvidiafb_sync,
};
```

设备操作函数结构，在 static int __devinit nvidiafb_probe(struct pci_dev *pd, const struct pci_device_id *ent)时被调用。

```
{  /*略*/
    info->fbops = &nvidia_fb_ops;
    info->fix = nvidiafb_fix;
       /*略*/
}
```

设备驱动编译如下代码所示。

```
# nVidia framebuffer 驱动 makefile
obj-$(CONFIG_FB_NVIDIA) += nvidiafb.o
nvidiafb-y: = nvidia.o nv_hw.o nv_setup.o \nv_accel.o
nvidiafb-$(CONFIG_FB_NVIDIA_I2C) += nv_i2c.o
nvidiafb-$(CONFIG_FB_NVIDIA_BACKLIGHT)  += nv_backlight.o
nvidiafb-$(CONFIG_PPC_OF) += nv_of.o
nvidiafb-objs: = $(nvidiafb-y)
```

7.3　本章总结

本章主要介绍了 Linux 下显卡驱动内容，以 Framebuffer 驱动为例介绍了显卡设备驱动的开发过程。读者在学习时，需要熟悉设备数据结构，巩固理解 Framebuffer 驱动框架程序的实现过程。

第 8 章

声卡驱动程序开发

本章介绍 Linux 声卡驱动开发的原理与实现过程，首先介绍声卡驱动的一些基础知识。

8.1 声卡驱动概述

（1）OpenSound 介绍

OpenSound 由 Hannu Savolainen 开发，他是最早开发 Linux 核心音效卡驱动程序的程序员之一，Hannu 后来继续开发了 OpenSound System。该程序是由 4Front Technologies 出售、支持多种 UNIX 系统的商业版音效驱动程序。RedHat 公司后来资助 Alan Cox（内核开发的第二号人物）来增强核心音效驱动程序，使它们完全模块化。

（2）ALSA 介绍

Jaroslav Kysela 及别人为 Gravis UltraSound 声卡写了可选的驱动程序。这个计划后来改名为 Advanced Linux Sound Architecture（先进 Linux 音效架构，简称 ALSA），产生了一个他们认为更加通用且可用来取代核心音效驱动程序的程序。ALSA 驱动程序支持许多常见的声卡，而且是全双工、全模块化，与现存核心中的音效架构兼容。

用户可以在 http：//www.alsa-project.org 网站了解更多 ALSA 的信息。它可以支持大多数流行的声卡，采用模块化架构，支持全双工、数字音频等声卡特性。当前 ALSA 最新版本为 1.0.14。

8.2 OSS 声卡驱动

OSS 的层次结构非常简单，应用程序通过 API 访问 OSS driver，OSS driver 控制声卡，如图 8-1 所示。

声卡中主要有 Mixer 和 CODEC（ADC/DAC）两个基本装置。其中，Mixer 用来控制输入音量的大小，对应的设备文件为 dev/mixer；CODEC 用来实现录音（模拟信号转变为数字信号）和播放声音（数字信号转变为模拟信号）的功能，对应的设备文件为/dev/dsp。

开发 OSS 应用程序的一般流程是：

（1）包含 OSS 头文件：#include。

（2）打开设备文件，返回文件描述符。

（3）使用 ioctl 设置设备的参数，控制设备的特性。

（4）对于录音，从设备读（read）。

（5）对于播放，向设备写（write）。

（6）关闭打开的设备。

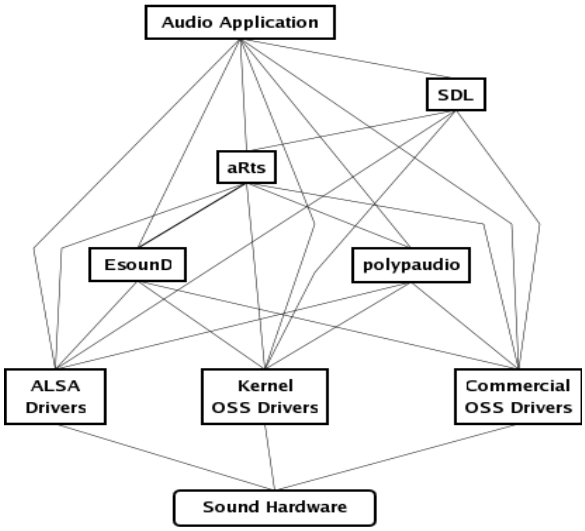


图 8-1　OSS 的层次结构

8.3　ALSA 声卡驱动

ALSA 声卡驱动以两种方式提供。ALSA ftp 网站上可以下载开发包，另外可以通过 2.6（或以上）Linux kernel tree。为了两者同步，ALSA 驱动目录树分成两个不同的目录树：alsa-kernel 和 alsa-driver。前者包含专门针对 Linux 2.6（或以上），这个目录仅用于 2.6（或以上）环境；后者 alsa-driver 包含一些配置文件，可用于 Linux 其他内核。

下面显示了 ALSA 声卡目录树。

1．ALSA 文件树结构

```
sound
       /core
              /oss
              /seq
                     /oss
                     /instr
       /ioctl32
       /include
       /drivers
              /mpu401
              /opl3
       /i2c
              /l3
       /synth
              /emux
       /pci
              /(cards)
       /isa
              /(cards)
       /arm
```

```
        /ppc
        /sparc
        /usb
        /pcmcia /(cards)
        /oss
```

2．ALSA 库 API

ALSA 库 API 是 ALSA drivers 的接口；开发者需要使用这些 API 的功能以实现 ALSA 支持的特定应用，ALSA 库文档对开发者有巨大的参考价值，为开发者提供了详细的开发教学和参考。

（1）ALSA 库 API 参考

当前设计的接口如下列表：

```
Information Interface (/proc/asound)
Control Interface (/dev/snd/controlCX)
Mixer Interface (/dev/snd/mixerCXDX)
PCM Interface (/dev/snd/pcmCXDX)
Raw MIDI Interface (/dev/snd/midiCXDX)
Sequencer Interface (/dev/snd/seq)
Timer Interface (/dev/snd/timer)
```

ALSA AC97 解码层封装得很好，用户不需要写额外的控制代码；只有底层的控制例程是需要的，AC97 codec API 定义如下。

```
<sound/ac97_codec.h>
AC97 Interface
  struct mychip {
          ...
          struct snd_ac97 *ac97;
          ...
  };
 static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                            unsigned short reg)
  {
          struct mychip *chip = ac97->private_data;
          ...
          /* 读寄存器值 */
          return the_register_value;
  }
  static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                                    unsigned short reg, unsigned short val)
  {
          struct mychip *chip = ac97->private_data;
          ...
          /* 写给定的寄存器值 */
  }
```

（2）特定 SOC 的驱动

```
static int snd_mychip_ac97(struct mychip *chip)
{
        struct snd_ac97_bus *bus;
        struct snd_ac97_template ac97;
        int err;
        static struct snd_ac97_bus_ops ops = {
```

```
                .write = snd_mychip_ac97_write,
                .read = snd_mychip_ac97_read
        };
        err = snd_ac97_bus(chip->card, 0, &ops, NULL, &bus);
        if (err < 0)
                return err;
        memset(&ac97, 0, sizeof(ac97));
        ac97.private_data = chip;
        return snd_ac97_mixer(bus, &ac97, &chip->ac97);
}
```

8.4　典型实例——AC97 声卡驱动实现

下面以 AC97 声卡驱动为例，介绍声卡驱动的详细配置与代码实现。

8.4.1　AC97 驱动分析

AC 97 是一声卡规范，全称为 Audio Codec 97，是 Intel 推出的一个廉价的集成声音解决方案，它去掉了声卡中成本最高的 DSP（数字信号处理器），通过特别编写的驱动程序让 CPU 来负责信号处理，使系统的成本降低，且效果可以接受，所以得到广泛的使用。

The AC97 规范可以在如下网站找到：http：//www.intel.com/design/chipsets/audio/ac97_r23.pdf

PC 的声卡驱动基本上使用 ALSA 驱动；而在嵌入式平台，针对特定的平台应用，有一些客户化的驱动程序。

Linux 内核主要有 at91、s3c24xx、pxa 的驱动，主要文件有 soc_core.c，Codec 目录下有 AC97.C 和 AC97.H。

其他文件在平台目录下，例如：

```
# S3c24XX Platform Support
snd-soc-s3c24xx-objs := s3c24xx-pcm.o
snd-soc-s3c24xx-i2s-objs := s3c24xx-i2s.o
snd-soc-s3c2443-ac97-objs := s3c2443-ac97.o
obj-$(CONFIG_SND_S3C24XX_SOC) += snd-soc-s3c24xx.o
obj-$(CONFIG_SND_S3C24XX_SOC_I2S) += snd-soc-s3c24xx-i2s.o
obj-$(CONFIG_SND_S3C2443_SOC_AC97) += snd-soc-s3c2443-ac97.o
# S3C24XX Machine Support
snd-soc-neo1973-wm8753-objs := neo1973_wm8753.o
snd-soc-smdk2443-wm9710-objs := smdk2443_wm9710.o
obj-$(CONFIG_SND_S3C24XX_SOC_NEO1973_WM8753) += snd-soc-neo1973-wm8753.o
obj-$(CONFIG_SND_S3C24XX_SOC_SMDK2443_WM9710) += snd-soc-smdk2443-wm9710.o
```

这些是平台相关的。

在 alsa-driver-1.0.14-4.06a\alsa-driver-1.0.14-4.06a\pci 下，有 pci 设备 id 文件 Ac97_id.h。

```
#define AC97_ID_AK4540      0x414b4d00
#define AC97_ID_AD1819      0x41445303
#define AC97_ID_STAC9704    0x83847604
……
/*Realtek 声卡 id*/
#define AC97_ID_ALC100      0x414c4300
#define AC97_ID_ALC650      0x414c4720
#define AC97_ID_ALC650D     0x414c4721
```

```
#define AC97_ID_ALC650E     0x414c4722
#define AC97_ID_ALC650F     0x414c4723
……
#define AC97_ID_CM9761_82   0x434d4982
```

下面对 ac97.c 进行分析。

```
/*
 * ac97.c  --  ALSA Soc AC97 codec support
 *
 * Copyright 2005 Wolfson Microelectronics PLC.
 * Author：  Liam Girdwood
 *        liam.girdwood@wolfsonmicro.com or Linux @wolfsonmicro.com
 *
 *  This program is free software; you can redistribute  it and/or modify it
 *  under  the terms of  the GNU General  Public License as published by the
 *  Free Software Foundation;  either version 2 of the  License, or (at your
 *  option) any later version.
 *
 *  Revision history
 *    17th Oct 2005   Initial version.
 *
 * Generic AC97 support.
 */
#include <Linux /init.h>
#include <Linux /kernel.h>
#include <Linux /device.h>
#include <sound/driver.h>
#include <sound/core.h>
#include <sound/pcm.h>
#include <sound/ac97_codec.h>
#include <sound/initval.h>
#include <sound/soc.h>
#define AC97_VERSION "0.6"
static int ac97_prepare(struct snd_pcm_substream *substream)
{
    struct snd_pcm_runtime *runtime = substream->runtime;
    struct snd_soc_pcm_runtime *rtd = substream->private_data;
    struct snd_soc_device *socdev = rtd->socdev;
    struct snd_soc_codec *codec = socdev->codec;
    int reg = (substream->stream == SNDRV_PCM_STREAM_PLAYBACK) ?
          AC97_PCM_FRONT_DAC_RATE ：  AC97_PCM_LR_ADC_RATE;
    return snd_ac97_set_rate(codec->ac97, reg, runtime->rate);
}
#define STD_AC97_RATES (SNDRV_PCM_RATE_8000 | SNDRV_PCM_RATE_11025 |\
        SNDRV_PCM_RATE_22050 | SNDRV_PCM_RATE_44100 | SNDRV_PCM_RATE_48000)
/*驱动的描述 dai-Description of Digital Audio * /
struct snd_soc_codec_dai ac97_dai = {
    .name = "AC97 HiFi",
    .type = SND_SOC_DAI_AC97,
    .playback = {
        .stream_name = "AC97 Playback",
        .channels_min = 1,
        .channels_max = 2,
        .rates = STD_AC97_RATES,
        .formats = SNDRV_PCM_FMTBIT_S16_LE,},
    .capture = {
```

```
            .stream_name = "AC97 Capture",
            .channels_min = 1,
            .channels_max = 2,
            .rates = STD_AC97_RATES,
            .formats = SNDRV_PCM_FMTBIT_S16_LE,},
        .ops = {
            .prepare = ac97_prepare,},
    };
    EXPORT_SYMBOL_GPL(ac97_dai);
    /*下面两个函数使用 soc_ac97_ops 指向特定平台的读写操作，如 S3C24XX 平台
        struct snd_ac97_bus_ops soc_ac97_ops = {
         .read  = s3c2443_ac97_read,
         .write = s3c2443_ac97_write,
         .warm_reset    = s3c2443_ac97_warm_reset,
         .reset = s3c2443_ac97_cold_reset,
        };
        */
    static unsigned int ac97_read(struct snd_soc_codec *codec,
        unsigned int reg)
    {
        return soc_ac97_ops.read(codec->ac97, reg);
    }
    static int ac97_write(struct snd_soc_codec *codec, unsigned int reg,
        unsigned int val)
    {
        soc_ac97_ops.write(codec->ac97, reg, val);
        return 0;
    }
```

系统初始化时执行__init 类型的函数，进行设备驱动的初始化。

```
    module_init(smdk2443_init);
    module_exit(smdk2443_exit);
    static int __init smdk2443_init(void)
    {
        int ret;
        smdk2443_snd_ac97_device = platform_device_alloc("soc-audio", -1);
        if (!smdk2443_snd_ac97_device)
            return -ENOMEM;
        platform_set_drvdata(smdk2443_snd_ac97_device,
                    &smdk2443_snd_ac97_devdata);
        smdk2443_snd_ac97_devdata.dev = &smdk2443_snd_ac97_device->dev;
        ret = platform_device_add(smdk2443_snd_ac97_device);
        if (ret)
            platform_device_put(smdk2443_snd_ac97_device);
        return ret;
    }
```

smdk2443 初始化时，设置驱动平台数据使用音频子系统的 soc 设备 sdk2443_snd_ac97_devdata，这里使用了 codec 设备 soc_codec_dev_ac97。

```
    static struct snd_soc_device smdk2443_snd_ac97_devdata = {
        .machine = &smdk2443,
        .platform = &s3c24xx_soc_platform,
        .codec_dev = &soc_codec_dev_ac97,
    };
```

soc_codec_dev_ac97 的 probe 函数如下所示。

```
struct snd_soc_codec_device soc_codec_dev_ac97= {
    .probe = ac97_soc_probe,
    .remove =ac97_soc_remove,
};
static int ac97_soc_probe(struct platform_device *pdev)
{
    struct snd_soc_device *socdev = platform_get_drvdata(pdev);
    struct snd_soc_codec *codec;
    struct snd_ac97_bus *ac97_bus;
    struct snd_ac97_template ac97_template;
    int ret = 0;
    printk(KERN_INFO "AC97 SoC Audio Codec %s\n", AC97_VERSION);
    socdev->codec = kzalloc(sizeof(struct snd_soc_codec), GFP_KERNEL);
    if (socdev->codec == NULL)
        return -ENOMEM;
    codec = socdev->codec;
    mutex_init(&codec->mutex);
    codec->name = "AC97";
    codec->owner = THIS_MODULE;
    codec->dai = &ac97_dai;
    codec->num_dai = 1;
    codec->write = ac97_write;
    codec->read = ac97_read;
    INIT_LIST_HEAD(&codec->dapm_widgets);
    INIT_LIST_HEAD(&codec->dapm_paths);
    /*注册 pcms */
    ret = snd_soc_new_pcms(socdev, SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1);
    if(ret < 0)
        goto err;
    /* 为 ac97 添加音频解码器为总线设备 */
    ret = snd_ac97_bus(codec->card, 0, &soc_ac97_ops, NULL, &ac97_bus);
    if(ret < 0)
        goto bus_err;
/* soc_ac97_ops 这个结构体在具体的设备里实例化，如 S3C2443 驱动
           struct snd_ac97_bus_ops soc_ac97_ops = {
           .read   = s3c2443_ac97_read,
           .write  = s3c2443_ac97_write,
           .warm_reset = s3c2443_ac97_warm_reset,
           .reset  = s3c2443_ac97_cold_reset,
        };
        */
    memset(&ac97_template, 0, sizeof(struct snd_ac97_template));
    ret = snd_ac97_mixer(ac97_bus, &ac97_template, &codec->ac97);
    if(ret < 0)
        goto bus_err;
/*注册 soc 声卡和 AC97 设备，成功返回 0*/
    ret = snd_soc_register_card(socdev);
    if (ret < 0)
        goto bus_err;
    return 0;
bus_err：
    snd_soc_free_pcms(socdev);
err：
    kfree(socdev->codec->reg_cache);
```

```
    kfree(socdev->codec);
    socdev->codec = NULL;
    return ret;
}

static int ac97_soc_remove(struct platform_device *pdev)
{
    struct snd_soc_device *socdev = platform_get_drvdata(pdev);
    struct snd_soc_codec *codec = socdev->codec;

    if(codec == NULL)
        return 0;
    snd_soc_free_pcms(socdev);
    kfree(socdev->codec->reg_cache);
    kfree(socdev->codec);
    return 0;
}
/*AC97 音频*/
Ac97 struct snd_soc_codec_device soc_codec_dev_ac97= {
    .probe =ac97_soc_probe,
    .remove =ac97_soc_remove,
};
EXPORT_SYMBOL_GPL(soc_codec_dev_ac97);
MODULE_DESCRIPTION("Soc Generic AC97 driver");
MODULE_AUTHOR("Liam Girdwood");
MODULE_LICENSE("GPL");
```

Ac97 驱动 makefile 如下所示。

```
ifndef SND_TOPDIR
SND_TOPDIR=../..
endif
include $(SND_TOPDIR)/toplevel.config
include $(SND_TOPDIR)/Makefile.conf
clean-files : = ac97_codec.c
export-objs  : = ac97_codec.o ac97_pcm.o ak4531_codec.o
include $(SND_TOPDIR)/alsa-kernel/pci/ac97/Makefile
include $(SND_TOPDIR)/Rules.make
ac97_codec.c:  ac97_codec.patch
$(SND_TOPDIR)/alsa-kernel/pci/ac97/ac97_ codec.c
```

在特定平台的配置如 pxa：

```
Linux -2.6.21gum/sound/soc/pxa/Kconfig
 config SND_PXA2XX_SOC_AC97
        tristate
        select AC97_BUS
        select SND_SOC_AC97_BUS
        select SND_PXA2XX_AC97
```

8.4.2　Realtek 声卡驱动配置

Realtek（瑞昱）在声卡市场使用很广泛，占大约一半市场。该声卡有个特色，可以自动探测输入设备阻抗来判断输入设备是耳机还是麦克风，这种技术叫做 Jack Sense，成为 AC97 的一大特点，Realek 是最早采用的厂商。

下面是针对 Linux 平台下的 ALC 声卡代码。

支持 Codec list：

```
====AC97 Codec=====
ALC100,100P
ALC200,200P
ALC650D
ALC650E
ALC650F
ALC650
ALC655
……
====HD Audio codec ====
ALC260
ALC262
ALC660
ALC861
```

自动安装：

```
./install
```

手工安装步骤如下：

（1）源代码解压缩（tar xfvj alsa-driver-1.0.xx.tar.bz2）。

（2）打开声音支持（soundcore module, default turn on）。

（3）编译代码。

- cd alsa-driver-1.0.xx
- /configure
- make
- make install
- /snddevices

（4）编辑/etc/modules.conf 或 conf.modules（参考相关的 modules.conf）。

```
 snd-xxxx is the card ID.
-- Azalia controller --ALC880 ALC882 ALC260 ALC262 ALC883 ALC885 ALC888
   --- Intel ICH6 ICH7 ---------
           snd-hda-intel
        --- ATI chipset -----
        snd-atiixp
   -- AC97 controller --ALC655 ALC650 ALC250 ALC255
      --- Intel ICH6 ICH7 , SiS 7012 and NVidia----------
          snd-intel8x0
      --- Via8233 Via686a  ------------------------------
          snd-via82xx
      --- ATI Chipset  ------------------------------
          snd-atiixp
```

之后复制下面的内容。到/etc/modules.conf 或/etc/modprobe.conf 文件的底部。

```
# ALSA portion
      alias char-major-116 snd
      alias snd-card-0 snd-xxxx
   # OSS/Free portion
      alias char-major-14 soundcore
      alias sound-slot-0 snd-card-0
```

```
# card #1
  alias sound-service-0-0 snd-mixer-oss
  alias sound-service-0-1 snd-seq-oss
  alias sound-service-0-3 snd-pcm-oss
  alias sound-service-0-8 snd-seq-oss
  alias sound-service-0-12 snd-pcm-oss
```

（5）重启机器。

（6）使用 alsamixer 取消静音（默认是静音的），需要编译和安装 ALSA library 和 utility（自动安装时这些都会安装）执行 alsamixer。

大部分内容读者可以参考 azx-021705.tar.bz2 中的 alsa-kernel/ Documenttation/ ALSA-Configuration.txt 文件；Kernel Version 必须 2.2.14 或更新的；mixer 默认静音。

在 Fedora Core 4.0 中，内核模块的添加或定义别名是在 /etc/modprobe.conf 文件。在其他版本可能是 modules.conf。如果系统中存在 modprobe.conf，就以这个文件为准。不同发行版本有不同的定义文件，比如 slackware 是定义在/etc/modules.conf 中，但也要在 /etc/rc.d/rc.modules 打开相关驱动模块。

modprobe.conf 或者 module.conf 是对系统已经加载的模块进行相应的配置，比如设置别名等。这些一般都是通过工具自动生成的，也可以通过查看硬件的文档和站点自己添加，如 855 主板，系统驱动用的是 snd-intel8x0，在/etc/modprobe.conf 的配置是如下的内容。下面这段内容是通过 alsaconf 配置工具自动生成的。

```
alias snd-card-0 snd-intel8x0
options snd-card-0 index=0
options snd-intel8x0 index=0
remove snd-intel8x0
{ /usr/sbin/alsactl store 0 >/dev/null 2>&1 || :  ; };
 /sbin/modprobe
 -r --ignore-remove snd-intel8x0
```

如果自己加载模块，就到这些目录中查看相应模块的信息，然后用 modprobe 加载，如 modinfo snd-intel8x0。

8.5　本章总结

本章首先介绍了声卡驱动的基础知识，包括 OSS 声卡驱动、ALSA 声卡驱动，然后以 AC97 声卡驱动为例，介绍了声卡设备驱动的实现过程与代码分析。通过本章的学习，读者将掌握声卡驱动的配置技术与实现流程。

第 9 章

USB 驱动程序开发

本章介绍 Linux USB 驱动开发的原理与实现过程，首先介绍 USB 驱动的一些基础知识。

9.1 USB 设备驱动概述

USB 是英文 Universal Serial Bus 的缩写，中文含义是“通用串行总线”。它是一种应用在 PC 领域的新型接口技术。早在 1995 年，就已经有 PC 带有 USB 接口了，但由于缺乏软件及硬件设备的支持，这些 PC 的 USB 接口都闲置未用。1998 年后，随着微软在 Windows 98 中内置了对 USB 接口的支持模块，加上 USB 设备的日渐增多，USB 接口才逐步走进了实用阶段。随着大量支持 USB 的个人电脑的普及，USB 逐步成为 PC 的标准接口已经是大势所趋。在主机（host）端，最新推出的 PC 几乎 100%支持 USB；而在外设（device）端，使用 USB 接口的设备也与日俱增，例如数码相机、扫描仪、游戏杆、图像设备、打印机、键盘、鼠标等。

1．USB 设备特点

USB 设备之所以会被大量应用，主要具有以下优点。

（1）可以热插拔。这就让用户在使用外接设备时，不需要重复“关机将并口或串口电缆接上再开机”这样的动作，而是直接在 PC 开机时，就可以将 USB 电缆插上使用。

（2）携带方便。USB 设备大多以“小、轻、薄”见长，对用户来说，同样 20GB 的硬盘，USB 硬盘比 IDE 硬盘要轻一半，想要随身携带大量数据时，当然 USB 硬盘会是首要之选了。

（3）标准统一。大家常见的是 IDE 接口的硬盘，串口的鼠标键盘，并口的打印机扫描仪，有了 USB 后，这些应用外设统统可以用同样的标准与 PC 连接，这就有了 USB 硬盘、USB 鼠标、USB 打印机等。

（4）可以连接多个设备。PC 上往往具有多个 USB 接口，可以同时连接几个设备，如果接上一个有 4 个端口的 USB Hub 时，就可以再连上 4 个 USB 设备，依此类推，最高可连接至 127 个设备。USB 2.0 是高速 USB，COMPAQ、Hewlett Packard、Intel、Lucent、Microsoft、NEC 和 PHILIPS 这 7 家厂商联合制定了 USB 2.0 接口标准。USB 2.0 将设备之间的数据传输速度增加到了 480Mbps，比 USB 1.1 标准快 40 倍左右。速度的提高对于用户的最大好处就是意味着用户可以使用到更高效的外部设备，而且具有多种速度的周边设备都可以被连接到 USB 2.0 线路上，而且无须担心数据传输时发生瓶颈效应。

2．USB 传输类型

- 中断传输

中断传输一般用于中断驱动设备的器件。由于不支持硬件中断，因而中断驱动的设备需要周期性查询，以确定设备是否有数据要传输。关于查询间隔全速设备为 1~255 ms 之间的任何值，低速设备为 10~255 ms 之间的任何值。

- 块传输

用于传输大块的，没有周期和传输速率限制的数据。

- 同步传输

同步传输要求维持一定的传输速率，必须保证数据发送方和接受方能够速率匹配。实时传输在帧中的最大封包容量为字节。

- 控制传输

控制传输用来把特定的请求传送给设备，常用于设备配置。

3．USB 的通信模型

整个结构分为三层。功能层的作用是实现设备的功能，客户软件负责把客户请求转化为一个或多个事务处理并产生对设备的访问。设备层的系统软件具有完成一般操作的功能，驱动程序提供了客户软件和主控制器之间的接口。总线接口层为主机和设备提供物理连接。客户软件不能直接访问设备硬件，而是通过系统软件和 USB 总线接口实现访问，如图 9-1 所示。

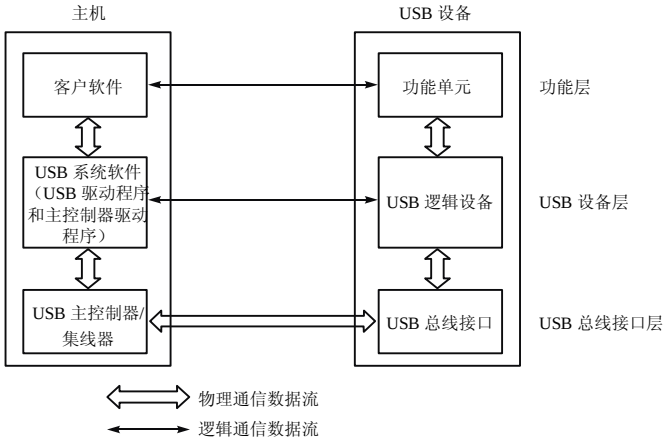


图 9-1　USB 通信模型结构

9.2　USB 驱动设备示例

首先对 Linux 驱动程序做简单的概述。

9.2.1　Linux 驱动程序概述

USB 驱动程序由主机驱动程序、USB 子系统、USB 设备驱动程序组成。在 Linux 操作系统中，存在一个连接 USB 设备驱动程序和主控制器驱动程序的子系统 USBCore，通

过定义一些数据结构、宏和功能函数来抽象所有的硬件设备。USBCore 提供了为硬件处理的所有下层接口。包含所有 USB 设备驱动和主机控制的通用程序，可称为 UpperAPI 和 LowerAPI。USB 子系统提供与设备驱动程序的接口，读取并解释 USB 设备描述符。配置描述符为 USB 设备分配唯一的地址，使用默认的配置来配置设备，支持基本的 USB 命令请求，连接设备与相应的驱动程序，转发设备驱动程序的数据包。

在做 USB 驱动开发时，对于简单的通信程序只需要参考相关驱动程序，改动很少即可；在做 Linux 下通信时，借助 USB_SKELETON.C，根据 USB 的设备 ID 改一个 VENDOR ID，然后加载模块，再建立节点，就可以对节点进行读写了。

9.2.2 驱动程序分析

下面对 usb_skeleton.c 进行分析。

```
#include <Linux /kernel.h>
#include <Linux /errno.h>
#include <Linux /init.h>
#include <Linux /slab.h>
#include <Linux /module.h>
#include <Linux /kref.h>
#include <asm/uaccess.h>
#include <Linux /usb.h>
#include <Linux /mutex.h>
```

/*设备 ID，根据连接到计算机上的 USB 设备进行修改。

```
#define USB_SKEL_VENDOR_ID  0xfff0
#define USB_SKEL_PRODUCT_ID 0xfff0
/*
```

id_table 用来告诉内核该模块支持的设备。USB 子系统通过设备的 production ID 和 vendor ID 的组合或者设备的 class、subclass 与 protocol 的组合来识别设备，并调用相关的驱动程序作处理

```
*/
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }
}
```

MODULE_DEVICE_TABLE 的第一个参数是设备的类型，如果是 USB 设备，那自然是 usb（如果是 PCI 设备，那将是 pci，这两个子系统用同一个宏来注册所支持的设备。这涉及 PCI 设备的驱动，在此不介绍）。后面一个参数是设备表，这个设备表的最后一个元素是空的，用于标识结束。代码定义了 USB_SKEL_VENDOR_ID 是 0xfff0，SB_SKEL_PRODUCT_ID 是 0xfff0，也就是说，当有一个设备接到集线器时，USB 子系统就会检查这个设备的 vendor ID 和 product ID，如果它们的值是 0xfff0 时，那么子系统就会调用这个 skeleton 模块作为设备的驱动。

```
MODULE_DEVICE_TABLE(usb, skel_table);
/*属于这个设备驱动程序的第一个设备号即 192，第二个设备的设备号 193，依次类推*/
#define USB_SKEL_MINOR_BASE 192
/* 私有定义*/
```

```
#define MAX_TRANSFER        (PAGE_SIZE - 512)
#define WRITES_IN_FLIGHT    8
```

先对 usb_skel 作简单分析，它拥有一个描述 usb 设备的结构体 udev，一个接口 interface，用于并发访问控制的 semaphore（信号量）limit_sem，用于接收数据的缓冲 bulk_in_buffer 及其尺寸 bulk_in_size，然后是批量输入输出端口地址 bulk_in_endpointAddr、bulk_out_endpointAddr，最后是一个内核使用的引用计数器。

```
/* 设备属性的结构体 */
struct usb_skel {
    struct usb_device      *dev;                /* 这个设备的 USB 设备 */
    struct usb_interface    *interface;         /* 这个设备的接口 */
    struct semaphore    limit_sem;
      unsigned char          *bulk_in_buffer;    /* 接收数据的缓冲 */
    size_tbulk_in_size;                         /* 接收缓冲的大小 */
    __u8bulk_in_endpointAddr;                   /* 批量输入端点的地址*/
    __u8bulk_out_endpointAddr;                  /*批量输出端点的地址*/
    struct kref     kref;
    struct mutex        io_mutex;               /* 同步 I/O */
};
#define to_skel_dev(d) container_of(d, struct usb_skel, kref)
static struct usb_driver skel_driver;
*/卸载设备驱动        */
static void skel_delete(struct kref *kref)
{
    struct usb_skel *dev = to_skel_dev(kref);
    usb_put_dev(dev->udev);
    kfree(dev->bulk_in_buffer);
    kfree(dev);
}
static int skel_open(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    struct usb_interface *interface;
    int subminor;
    int retval = 0;
    subminor = iminor(inode);
    interface = usb_find_interface(&skel_driver, subminor);
    if (!interface) {
        err ("%s - error, can't find device for minor %d",
            __FUNCTION__, subminor);
        retval = -ENODEV;
        goto exit;
    }
    dev = usb_get_intfdata(interface);
    if (!dev) {
        retval = -ENODEV;
        goto exit;
    }
    /* 阻止设备自动挂起 */
    retval = usb_autopm_get_interface(interface);
    if (retval)
        goto exit;
    /* 增加设备的使用计数 */
    kref_get(&dev->kref);
```

```
        /* file 的私有数据结构保存这个对象*/
        file->private_data = dev;
    exit:
        return retval;
    }
    static int skel_release(struct inode *inode, struct file *file)
    {
        struct usb_skel *dev;
        dev = (struct usb_skel *)file->private_data;
        if (dev == NULL)
            return -ENODEV;
        /*允许设备自动挂起*/
        mutex_lock(&dev->io_mutex);
        if (dev->interface)
            usb_autopm_put_interface(dev->interface);
        mutex_unlock(&dev->io_mutex);
        /*减小设备的使用计数*/
        kref_put(&dev->kref, skel_delete);
        return 0;
    }
```

用 usb_bulk_msg 函数在不需要创建 urbs 和操作 urb 函数的情况下发送数据给设备，或者从设备接收数据。调用 usb_bulk_msg 函数并传递一个存储空间，用来缓冲和放置驱动收到的数据，若没收到数据，就产生失败并返回一个错误信息。

```
    static ssize_t skel_read(struct file *file, char *buffer, size_t count, loff_t
*ppos)
    {
        struct usb_skel *dev;
        int retval;
        int bytes_read;
        dev = (struct usb_skel *)file->private_data;
        mutex_lock(&dev->io_mutex);
        if (!dev->interface) {      /* 调用 disconnect()*/
            retval = -ENODEV;
            goto exit;
        }
        /* 从设备以块的方式批量读数据*/
        retval = usb_bulk_msg(dev->udev,
                    usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                    dev->bulk_in_buffer,
                    min(dev->bulk_in_size, count),
                    &bytes_read, 10000);
        /* 如果读成功，复制数据到用户空间*/
        if (!retval) {
            if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
                retval = -EFAULT;
            else
                retval = bytes_read;
        }
    exit:
        mutex_unlock(&dev->io_mutex);
        return retval;
    }
    static void skel_write_bulk_callback(struct urb *urb)
    {
```

```
        struct usb_skel *dev;
        dev = (struct usb_skel *)urb->context;
        /* sync/async 未连接故障，不视为错误 */
        if (urb->status &&
            !(urb->status == -ENOENT ||
              urb->status == -ECONNRESET ||
              urb->status == -ESHUTDOWN)) {
             err("%s - nonzero write bulk status received： %d",
                __FUNCTION__, urb->status);
        }
        /* 释放分配的内存 */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
        up(&dev->limit_sem);
    }
    static ssize_t skel_write(struct file *file, const char *user_buffer, size_t
count, loff_t *ppos)
    {
        struct usb_skel *dev;
        int retval = 0;
        struct urb *urb = NULL;
        char *buf = NULL;
        size_t writesize = min(count, (size_t)MAX_TRANSFER);
        dev = (struct usb_skel *)file->private_data;
        /* 验证有数据要写入 */
        if (count == 0)
            goto exit;
        /*限制 URB 的数量，防止占用过多内存*/
        if (down_interruptible(&dev->limit_sem)) {
            retval = -ERESTARTSYS;
            goto exit;
        }
        mutex_lock(&dev->io_mutex);
        if (!dev->interface) {      /* 调用 disconnect()*/
            retval = -ENODEV;
            goto error;
        }
        /* 创建一个 urb 和 buffer,复制数据到 urb */
        urb = usb_alloc_urb(0, GFP_KERNEL);
        if (!urb) {
            retval = -ENOMEM;
            goto error;
        }
        buf    =    usb_buffer_alloc(dev->udev,    writesize,    GFP_KERNEL,
&urb->transfer_dma);
        if (!buf) {
            retval = -ENOMEM;
            goto error;
        }
        if (copy_from_user(buf, user_buffer, writesize)) {
            retval = -EFAULT;
            goto error;
        }
        /*初始化 urb*/
        usb_fill_bulk_urb(urb, dev->udev,
                  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
```

```
                buf, writesize, skel_write_bulk_callback, dev);
        urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
        /* 发送数据到批量传输端点 */
        retval = usb_submit_urb(urb, GFP_KERNEL);
        if (retval) {
            err("%s - failed submitting write urb, error %d", __FUNCTION__,
retval);
            goto error;
        }
        /* 释放对 urb 的使用，USB core 会完全释放它*/
        usb_free_urb(urb);
        mutex_unlock(&dev->io_mutex);
        return writesize;
    error:
        if (urb) {
            usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
            usb_free_urb(urb);
        }
        mutex_unlock(&dev->io_mutex);
        up(&dev->limit_sem);
    exit:
        return retval;
    }
```

Skel_fops 主要包括 Open 打开函数、Read 读函数、Write 写函数、ioltrl 设备控制函数以及用户各类设备的特殊控制。设备驱动程序的设计就是实现上述四个函数外加一个设备初始化的函数，这些函数在设备驱动程序中可以进行 skel_init()、skel_open()、skel_read()、skel_ioctrl()等调用。声明一个称为 file operation 的结构体将用户级的 open 等函数与设备 skel_open()等函数联系起来。

下面以对数据的读为例，进行更深入的说明。

read 系统调用→当前进程文件表对应的文件→fops→skel_read，从而实现对设备的读操作。

```
    static const struct file_operations skel_fops = {
        .owner =THIS_MODULE,
        .read =skel_read,
        .write =skel_write,
        .open =skel_open
        .release =  skel_release
    };
    /*
     * usb 类驱动信息，向 usb core 获取设备号,
     * 把设备注册到内核。
     */
    static struct usb_class_driver skel_class = {
        .name ="skel%d",
        .fops =&skel_fops,
        .minor_base =USB_SKEL_MINOR_BASE
    };
```

其中 probe 是 USB 子系统自动调用的一个函数，有 USB 设备接到硬件集线器时，USB 子系统会根据 production ID 和 vendor ID 的组合或者设备的 class、subclass 与 protocol 的组合来识别设备调用相应驱动程序的 probe（探测）函数，对于 skeleton 来说，就是 skel_probe。

系统会传递给探测函数一个 usb_interface *和一个 struct usb_device_id *作为参数。它们分别是该 USB 设备的接口描述（一般会是该设备的第 0 号接口，该接口的默认设置也是第 0 号设置）及它的设备 ID 描述（包括 Vendor ID、Production ID 等）。Probe 函数比较长，分段来分析这个函数。

```
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;
```

在初始化一些资源后，可以看到第一个关键的函数调用——interface_to_usbdev。它调用 usb_interface 得到该接口所在设备的设备描述结构。本来要得到 usb_device 只要用 interface_to_usbdev 就够了，但因为要增加对该 usb_device 的引用计数，所以应该增加 usb_get_dev 的操作，来增加引用计数，并在释放设备时用 usb_put_dev 来减少引用计数。需要解释的是，该引用计数值是对该 usb_device 的计数，并不是对本模块的计数，本模块的计数要由 kref 来维护。所以，probe 一开始就初始化 kref。事实上，kref_init 操作不单只初始化 kref，还将其置设成 1。所以在出错处理代码中有 kref_put，它把 kref 的计数减 1，如果 kref 计数已经为 0，那么 kref 会被释放。Kref_put 的第二个参数是一个函数指针，指向一个清理函数。

注意：*该指针不能为空，或者 kfree。该函数会在最后一个对 kref 的引用释放时被调用。*

```
    static int skel_probe(struct usb_interface *interface, const struct usb_
device_id *id)
    {
        struct usb_skel *dev;
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        size_t buffer_size;
        int i;
        int retval = -ENOMEM;
        /* 为设备分配内存并初始化*/
        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev) {
            err("Out of memory");
            goto error;
        }
        kref_init(&dev->kref);
        sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
        mutex_init(&dev->io_mutex);
```

当执行打开操作时，要增加 kref 的计数，可以用 kref_get 来完成。所有对 struct kref 的操作都有内核代码确保其原子性。得到了该 usb_device 后，要对自定义的 usb_skel 各个状态和资源做初始化。这部分工作的任务主要是向 usb_skel 注册该 USB 设备的端点。在一个 usb_host_interface 结构中有一个 usb_interface_descriptor（叫做 desc 的成员），它应该是用于描述该 interface 的一些属性，其中 bNubEndpoints 一个 8 位的数字，代表了该接口的端点数。Probe 然后遍历所有的端点，检查它们的类型跟方向，注册到 usb_skel。

```
        dev->udev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;
        /* 设置端点信息 */
        /* 使用第一个 bulk-in 和 bulk-out endpoints */
        iface_desc = interface->cur_altsetting;
```

```
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;
        if (!dev->bulk_in_endpointAddr &&
            usb_endpoint_is_bulk_in(endpoint)) {
            /* 发现一个批量传输输入端点 */
            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
            dev->bulk_in_size = buffer_size;
            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
            if (!dev->bulk_in_buffer) {
                err("Could not allocate bulk_in_buffer");
                goto error;
            }
        }

        if (!dev->bulk_out_endpointAddr &&
            usb_endpoint_is_bulk_out(endpoint)) {
            /*发现一个批量传输输出端点 */
            dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
        }
    }
    if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
        err("Could not find both bulk-in and bulk-out endpoints");
        goto error;
    }
    /* 为设备接口保存数据指针 */
    usb_set_intfdata(interface, dev);
    /* 注册设备 */
    retval = usb_register_dev(interface, &skel_class);
    if (retval) {
        /* 未能正常注册 */
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
    }
    /*通知用户设备结点可用 */
    info("USB Skeleton device now attached to USBSkel-%d", interface->minor);
    return 0;
error:
    if (dev)
        kref_put(&dev->kref, skel_delete);
    return retval;
}
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;
    lock_kernel();
    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);
    /*归还设备符号*/
    usb_deregister_dev(interface, &skel_class);
    /* 防止 I/O 操作*/
    mutex_lock(&dev->io_mutex);
    dev->interface = NULL;
    mutex_unlock(&dev->io_mutex);
```

```
    unlock_kernel();
    /* 减小引用计数 */
    kref_put(&dev->kref, skel_delete);
    info("USB Skeleton #%d now disconnected", minor);
}
static struct usb_driver skel_driver = {
    .name ="skeleton",
    .probe =skel_probe,
    .disconnect =skel_disconnect,
    .id_table = skel_table,
};
```

usb_driver 其实是一个系统定义的结构，里面包含了一个名字、一个文件操作结构体以及一个次设备号的基准值。事实上它定义真正完成对设备 I/O 操作的函数，所以它的核心内容应该是 skel_fops。因为 USB 设备可以有多个 interface，每个 interface 所定义的 I/O 操作可能不一样，所以系统注册的 usb_class_driver 要求注册到某一个 interface，因此 usb_register_dev 的第一个参数是 interface，而第二个参数就是某一个 usb_class_driver。通常情况下，Linux 系统用主设备号来识别某类设备的驱动程序，用次设备号管理识别具体的设备，驱动程序可以依照次设备号区分不同的设备，所以这里的次设备号其实是用来管理不同的 interface 的。

```
static int __init usb_skel_init(void)
{
    int result;
    /* 注册设备到 USB 子系统*/
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}
static void __exit usb_skel_exit(void)
{
    /* 注册带有 USB 次级系统的驱动器 */
    usb_deregister(&skel_driver);
}
```

这个 init 和 exit 函数的作用只是用来注册驱动程序，这个描述驱动程序的结构体是系统定义的标准结构 struct usb_driver。注册和注销的方法很简单，利用 usb_register（struct *usb_driver）和 usb_unregister（struct *usb_driver）即可实现。这个结构体需要做的，是要向系统提供几个函数入口和驱动的名字，代码如下。

```
module_init(usb_skel_init);
module_exit(usb_skel_exit);
MODULE_LICENSE("GPL");
```

9.3　典型实例——单片机的主从通信实例

下面介绍一个具体的 USB 主从通信实例。首先对主从通信做简要介绍。

9.3.1　主从通信介绍

这里介绍的主从通信程序是主机和 USB 设备之间的通信，主机一般是 PC 或工控机，

从机是 USB 单片机或其他 USB 设备。

目前市场上的 USB 接口芯片基本上可以分为以下几种。

（1）通用的 USB 控制器，内含 CPU（一般是 8 位或 16 位），可以通过编程开发多种 USB 设备。如 AN2131、C8051F320、CYPRESS USB controller。

（2）专用 USB 控制器，比如 USB 键盘控制器、USB 鼠标控制器，这种芯片只能用来开发某种特定的 USB 设备。

（3）USB 协议处理模块，一般是 SIE+Transceiver，不内置 CPU，使用时需要外接一个 CPU，比如国半的 USBN9602、Philips 的 D12。

下面给出 USB 单片机的驱动程序和应用程序，单片机选用通用型 C8051F320，USB 主机可以读写该单片机。

9.3.2 USB 设备驱动程序

```
/*
 * USB Skeleton driver - 2.2
  * Copyright (C) 2001-2004 Greg Kroah-Hartman (greg@kroah.com)
 *
 *   This program is free software; you can redistribute it and/or
 *   modify it under the terms of the GNU General Public License as
 *   published by the Free Software Foundation, version 2.
 *
 * This driver is based on the 2.6.3 version of drivers/usb/usb-skeleton.c
 * but has been rewritten to be easier to read and use.
 *
 */
#include <Linux /kernel.h>
#include <Linux /errno.h>
#include <Linux /init.h>
#include <Linux /slab.h>
#include <Linux /module.h>
#include <Linux /kref.h>
#include <asm/uaccess.h>
#include <Linux /usb.h>
#include <Linux /mutex.h>
#define USB_SKEL_VENDOR_ID  0xc410
 /*C8051F320厂商ID*/
#define USB_SKEL_PRODUCT_ID 0x0000
/*驱动程序的设备ID表 */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }
};
MODULE_DEVICE_TABLE(usb, skel_table);
/* 获取一个设备符号从192开始 */
#define USB_SKEL_MINOR_BASE 192
#define MAX_TRANSFER        (PAGE_SIZE - 512)
#define WRITES_IN_FLIGHT    8
struct usb_skel {
    struct usb_device*dev;                  /* 这个设备的USB设备 */
    struct usb_interface*interface;         /* 这个设备的接口 */
    struct semaphore    limit_sem;
    unsigned char*bulk_in_buffer;           /* 接收数据的缓冲 */
```

```
    size_tbulk_in_size;                     /* 接收缓冲的大小 */
    __u8bulk_in_endpointAddr;               /* 批量输入端点的地址*/
    __u8bulk_out_endpointAddr;              /*批量输出端点的地址*/
    struct krefkref;
    struct mutexio_mutex;                   /* 同步 I/O */
};
#define to_skel_dev(d) container_of(d, struct usb_skel, kref)
static struct usb_driver skel_driver;
static void skel_delete(struct kref *kref)
{
    struct usb_skel *dev = to_skel_dev(kref);
    usb_put_dev(dev->udev);
    kfree(dev->bulk_in_buffer);
    kfree(dev);
}
static int skel_open(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    struct usb_interface *interface;
    int subminor;
    int retval = 0;
    subminor = iminor(inode);
    interface = usb_find_interface(&skel_driver, subminor);
    if (!interface) {
        err ("%s - error, can't find device for minor %d",
            __FUNCTION__, subminor);
        retval = -ENODEV;
        goto exit;
    }
    dev = usb_get_intfdata(interface);
    if (!dev) {
        retval = -ENODEV;
        goto exit;
    }
    /* 防止设备自动挂起 */
    retval = usb_autopm_get_interface(interface);
    if (retval)
        goto exit;
    /* 增加设备的使用率 */
    kref_get(&dev->kref);
    file->private_data = dev;
exit:
    return retval;
}
static int skel_release(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    dev = (struct usb_skel *)file->private_data;
    if (dev == NULL)
        return -ENODEV;
    mutex_lock(&dev->io_mutex);
    if (dev->interface)
        usb_autopm_put_interface(dev->interface);
    mutex_unlock(&dev->io_mutex);
    /* 减小引用计数*/
    kref_put(&dev->kref, skel_delete);
```

```
        return 0;
    }
    static ssize_t skel_read(struct file *file, char *buffer, size_t count, loff_t
*ppos)
    {
        struct usb_skel *dev;
        int retval;
        int bytes_read;
        dev = (struct usb_skel *)file->private_data;
        mutex_lock(&dev->io_mutex);
        if (!dev->interface) {      /* 唤醒 disconnect() */
            retval = -ENODEV;
            goto exit;
        }
        /* 块设备读操作 */
        retval = usb_bulk_msg(dev->udev,
                    usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                    dev->bulk_in_buffer,
                    min(dev->bulk_in_size, count),
                    &bytes_read, 10000);
        /* 如果读成功复制数据到用户空间 */
        if (!retval) {
            if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
                retval = -EFAULT;
            else
                retval = bytes_read;
        }
    exit:
        mutex_unlock(&dev->io_mutex);
        return retval;
    }
    static void skel_write_bulk_callback(struct urb *urb)
    {
        struct usb_skel *dev;
        dev = (struct usb_skel *)urb->context;
        if (urb->status &&
           !(urb->status == -ENOENT ||
             urb->status == -ECONNRESET ||
             urb->status == -ESHUTDOWN)) {
            err("%s - nonzero write bulk status received: %d",
                __FUNCTION__, urb->status);
        }
        /*释放分配的内存 */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
        up(&dev->limit_sem);
    }
    static ssize_t skel_write(struct file *file, const char *user_buffer, size_t
count, loff_t *ppos)
    {
        struct usb_skel *dev;
        int retval = 0;
        struct urb *urb = NULL;
        char *buf = NULL;
        size_t writesize = min(count, (size_t)MAX_TRANSFER);
        dev = (struct usb_skel *)file->private_data;
```

```
        if (count == 0)
            goto exit;
            if (down_interruptible(&dev->limit_sem)) {
            retval = -ERESTARTSYS;
            goto exit;
        }
        mutex_lock(&dev->io_mutex);
        if (!dev->interface) {      /* 唤醒 disconnect()*/
            retval = -ENODEV;
            goto error;
        }
        urb = usb_alloc_urb(0, GFP_KERNEL);
        if (!urb) {
            retval = -ENOMEM;
            goto error;
        }
        buf  =  usb_buffer_alloc(dev->udev,  writesize,  GFP_KERNEL,  &urb->
transfer_dma);
        if (!buf) {
            retval = -ENOMEM;
            goto error;
        }
        if (copy_from_user(buf, user_buffer, writesize)) {
            retval = -EFAULT;
            goto error;
        }
        usb_fill_bulk_urb(urb,dev->udev,usb_sndbulkpipe(dev->udev,dev->bulk
_out_endpointAddr),
                  buf, writesize, skel_write_bulk_callback, dev);
        urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
        retval = usb_submit_urb(urb, GFP_KERNEL);
        if (retval) {
            err("%s - failed submitting write urb,error %d",__FUNCTION__,retval);
            goto error;
        }
        usb_free_urb(urb);
        mutex_unlock(&dev->io_mutex);
        return writesize;
    error:
        if (urb) {
            usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
            usb_free_urb(urb);
        }
        mutex_unlock(&dev->io_mutex);
        up(&dev->limit_sem);
    exit:
        return retval;
    }
    static const struct file_operations skel_fops = {
        .owner =THIS_MODULE,
        .read =skel_read,
        .write =skel_write,
        .open =skel_open,
        .release =skel_release,
    };
    static struct usb_class_driver skel_class = {
        .name ="skel%d",
```

```
        .fops =&skel_fops,
        .minor_base =USB_SKEL_MINOR_BASE,
    };
    static int skel_probe(struct usb_interface *interface, const struct
usb_device_id *id)
    {
        struct usb_skel *dev;
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        size_t buffer_size;
        int i;
        int retval = -ENOMEM;
        /* 为设备分配内存 */
        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev) {
            err("Out of memory");
            goto error;
        }
        kref_init(&dev->kref);
        sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
        mutex_init(&dev->io_mutex);
        dev->udev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;
        /* 建立端点信息 */
        iface_desc = interface->cur_altsetting;
        for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
            endpoint = &iface_desc->endpoint[i].desc;
            if (!dev->bulk_in_endpointAddr &&
                usb_endpoint_is_bulk_in(endpoint)) {
                出现许多端点
                buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
                dev->bulk_in_size = buffer_size;
                dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
                dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
                if (!dev->bulk_in_buffer) {
                    err("Could not allocate bulk_in_buffer");
                    goto error;
                }
            }
            if (!dev->bulk_out_endpointAddr &&
               usb_endpoint_is_bulk_out(endpoint)) {
                dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
            }
        }
        if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
            err("Could not find both bulk-in and bulk-out endpoints");
            goto error;
        }
        usb_set_intfdata(interface, dev);
        retval = usb_register_dev(interface, &skel_class);
        if (retval) {
                    err("Not able to get a minor for this device.");
            usb_set_intfdata(interface, NULL);
            goto error;
        }
            info("USB Skeleton device now attached to USBSkel-%d", interface->
minor);
```

```
    return 0;
error：
    if (dev)
        kref_put(&dev->kref, skel_delete);
    return retval;
}
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;
    lock_kernel();
    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);
    usb_deregister_dev(interface, &skel_class);
    mutex_lock(&dev->io_mutex);
    dev->interface = NULL;
    mutex_unlock(&dev->io_mutex);
    unlock_kernel();
    /*减小引用计数 */
    kref_put(&dev->kref, skel_delete);
    info("USB Skeleton #%d now disconnected", minor);
}
static struct usb_driver skel_driver = {
    .name ="skeleton",
    .probe =skel_probe,
    .disconnect =skel_disconnect,
    .id_table = skel_table,
};
static int __init usb_skel_init(void)
{
    int result;
    注册带有 USB 次级系统的驱动器
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}
static void __exit usb_skel_exit(void)
{
    /* deregister this driver with the USB subsystem */
    usb_deregister(&skel_driver);
}
module_init(usb_skel_init);
module_exit(usb_skel_exit);
MODULE_LICENSE("GPL");
Makefile 文件：
#Makefile for kernel module for kernel 2.6
ifneq ($(KERNELRELEASE),)
    obj-m：= cygnal.o
else
KDIR：= /lib/modules/$(shell uname -r)/build
PWD ：= $(shell pwd)
default：
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
Endif
```

加载设备驱动程序：

```
# insmod cygnal
```

使用 root 建立文件节点：

```
# mknod cygnal c 233 0
```

设备访问权限：

```
# chown 777 username cygnal
```

9.3.3 主机程序源代码

本例实现了 C8051f320 单片机和 USB 主机的通信，主机向单片机发送数据。发送的数据有初始化数据、大于 0x80 的数据以及小于 0x80 的数据。

下面是主机应用程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<error.h>
#include<unistd.h>
#include<fcntl.h>
#include <sys/stat.h>
int send_data(int fp, const char * cmd, int length);
int main(void){
int data_size = 16;
char data[8];
size_t cnt=0;
int fp0,fp1;
char c;
static char data_init[] = {0X80,0X80,0X80,0X80,0X80,0X800X80,0X80};
static char data_pos[] = {0X90,0X90,0X90,0X90,0X90,0X90,0X90,0X90};
static char data_neg[] = {0x70,0x70,0x70,0x70,0x70,0x70,0x70,0x70};
if((fp0=open("/dev/usb/USBSkel0",O_RDWR))<0){
perror("unable to open c8051 devices");
exit(1);
}
if((fp1=open("/dev/usb/USBSkel1",O_RDWR))<0){
perror("unable to open c8051 devices");
exit(1);
}
//初始化
send_data(fp0, data_init, sizeof(data_init));
send_data(fp1, data_init, sizeof(data_init));
//根据 C 的值发送数据
while(c=getchar()!='e'){
switch(c){
case 'p': send_data(fp0, data_pos, sizeof(data_pos));
send_data(fp1, data_neg, sizeof(data_neg));
break;
case 'n': send_data(fp0, data_neg, sizeof(data_neg));
send_data(fp1, data_pos, sizeof(data_pos));
break;
default:
break;
```

```
}
}
if ((cnt = read(fp0, data, data_size)) > 0) {
        printf("Read：  %u\n", cnt);
printf("%X\n",data[0]);
printf("%X\n",data[1]);
                }
close(fp0);
close(fp1);
return 0;
}
int send_data(int fp, const char * cmd, int length) {
    int result;
    int x;
    if((result = write(fp, cmd, length)) != length) {
        printf ("Write warning：  %d bytes requested, %d written\n");
    } else if (result < 0) {
        perror ("send_cmd failure");
        exit (1);
    }
printf("usb conctroling functioning");
    return (result);
```

9.4　本章总结

本章首先介绍了 USB 设备驱动开发的基础知识，然后针对一个 USB 设备进行了示例分析，并以主从通信为例，介绍了 Linux 下设备驱动程序的设计与实现。通过本章的学习，读者可以掌握 USB 驱动开发的原理与实现技术。

第 10 章

闪存 Flash 驱动程序开发

本章介绍 Linux 下闪存 Flash 驱动开发的原理与实现过程，首先介绍闪存 Flash 驱动的一些基础知识。

10.1 Flash 闪存基础

现在市场上两种主要的非易失闪存技术是 NOR 和 NAND。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，具有更高的性能，并且像磁盘一样可以通过接口轻松升级。“Flash 存储器”经常可以与“NOR 存储器”互换使用。大多数情况下闪存只是用来存储少量的代码，这时 NOR 闪存更适合一些。而 NAND 则是高数据存储密度的理想解决方案。NOR 的特点是芯片内执行（XIP, eXecute In Place），这样应用程序可以直接在 Flash 闪存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1~4MB 小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 Flash 的管理和需要特殊的系统接口。

下面对这两种技术特点做归纳比较。

1．性能比较

Flash 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何 Flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。由于擦除 NOR 器件时是以 64~128KB 的块进行的，执行一个写入/擦除操作的时间为 5s，与此相反，擦除 NAND 器件是以 8~32KB 的块进行的，执行相同的操作最多只需要 4ms。

执行擦除时块尺寸的不同进一步拉大了 NOR 和 NADN 之间的性能差距，统计表明，对于给定的一套写入操作（尤其是更新小文件时），更多的擦除操作必须在基于 NOR 的单元中进行。这样当选择存储解决方案时，必须权衡以下的各项因素。

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。

- NAND 的擦除单元更小，相应的擦除电路更少。

2．接口差别

NOR Flash 带有 SRAM 接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。NAND 器件使用复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。NAND 读和写操作采用 512 字节的块，这一点有点像硬盘管理此类操作，很自然地，基于 NAND 的存储器就可以取代硬盘或其他块设备。

3．容量和成本

NAND Flash 的单元尺寸几乎是 NOR 器件的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。NOR Flash 占据了容量为 1~16MB 闪存市场的大部分，而 NAND Flash 只是用在 8~128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

4．可靠性和耐用性

采用 Flash 介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说，Flash 是非常合适的存储方案。可以从寿命（耐用性）、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性。

- 寿命（耐用性）

在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。NAND 存储器除了具有 10 比 1 的块擦除周期优势，典型的 NAND 块尺寸要比 NOR 器件小 8 倍，每个 NAND 存储器块在给定的时间内的删除次数要少一些。

- 位交换

所有 Flash 器件都受位交换现象的困扰。在某些情况下（很少见，NAND 发生的次数要比 NOR 多），一个比特位会发生反转或被报告反转了。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。当然，如果这个位真的改变了，就必须采用错误探测/错误更正（EDC/ECC）算法。位反转的问题更多见于 NAND 闪存，NAND 的供应商建议使用 NAND 闪存时，同时使用 EDC/ECC 算法。这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用 EDC/ECC 系统以确保可靠性。

- 坏块处理

NAND 器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

5．易使用性

可以非常直接地使用基于 NOR 的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。由于需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂

家而异。在使用 NAND 器件时，必须先写入驱动程序，才能继续执行其他操作。向 NAND 器件写入信息需要相当的技巧，因为绝不能向坏块写入，这就意味着在 NAND 器件上自始至终都必须进行虚拟映射。

6．软件支持

当讨论软件支持时，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序（MTD），NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。使用 NOR 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 NOR 器件的更高级软件，这其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。

对于 16 位的器件，NOR 闪存大约需要 41 个 I/O 引脚；相对而言，NAND 器件仅需 24 个引脚。NAND 器件能够复用指令、地址和数据总线，从而节省了引脚数量。复用接口的一项好处就在于能够利用同样的硬件设计和电路板，支持较大的 NAND 器件。由于普通的 TSOP-1 封装已经沿用多年，该功能让客户能够把较高密度的 NAND 器件移植到相同的电路板上。NAND 器件的另外一个好处是其封装选项。NAND 提供一种厚膜的 2GB 裸片或能够支持最多四颗堆叠裸片，容许在相同的 TSOP-1 封装中堆叠一个 8GB 的器件。这就使得一种封装和接口能够在将来支持较高的密度。

NAND 闪存阵列分为一系列 128KB 的区块（block），这些区块是 NAND 器件中最小的可擦除实体。擦除一个区块就是把所有的位（bit）设置为“1”（而所有字节(byte)设置为 FFh）。有必要通过编程将已擦除的位从“1”变为“0”。最小的编程实体是字节（byte）。一些 NOR 闪存能同时执行读写操作。虽然 NAND 不能同时执行读写操作，它可以采用称为“映射（shadowing）”的方法，在系统级实现这一点。这种方法在个人电脑上已经沿用多年，即将 BIOS 从速率较低的 ROM 加载到速率较高的 RAM 上。

NAND 的效率较高，是因为 NAND 串中没有金属触点，（4F2：10F2）NOR 的每一个单元都需要独立的金属触点，NAND 闪存单元的大小比 NOR 要小。NAND 与硬盘驱动器类似，基于扇区（页），适合于存储连续的数据，如图片、音频或个人电脑数据。虽然通过把数据映射到 RAM 上，能在系统级实现随机存取，但是，这样做需要额外的 RAM 存储空间。此外，跟硬盘一样，NAND 器件存在坏的扇区，需要纠错码（ECC）来维持数据的完整性。

存储单元面积越小，裸片的面积也就越小。在这种情况下，NAND 就能够为当今的低成本消费市场提供存储容量更大的闪存产品。NAND 闪存用于几乎所有可擦除的存储卡。NAND 的复用接口为所有最新的器件和密度都提供了一种相似的引脚输出。这种引脚输出使得设计工程师无须改变电路板的硬件设计，就能从更小的密度移植到更大密度的设计上。

10.2 Flash MTD 技术

MTD（memory technology device，内存技术设备）用于访问存储设备（ROM、Flash）的 Linux 的子系统。MTD 的主要目的是为了使新的存储设备的驱动更加简单，为此它在硬

件和上层之间提供了一个抽象的接口。MTD 的所有源代码在/drivers/mtd 子目录下。通常将 CFI 接口的 MTD 设备分为四层（从设备节点直到底层硬件驱动），这四层从上到下依次是设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层。

（1）Flash 硬件驱动层。硬件驱动层负责在 init 时驱动 Flash 硬件，Linux MTD 设备的 NOR Flash 芯片驱动遵循 CFI 接口标准，其驱动程序位于 drivers/mtd/chips 子目录下。NAND 型 Flash 的驱动程序则位于/drivers/mtd/nand 子目录下。

（2）MTD 原始设备。原始设备层由两部分组成，一部分是 MTD 原始设备的通用代码，另一部分是各个特定的 Flash 的数据，例如分区。

用于描述 MTD 原始设备的数据结构是 mtd_info，这其中定义了大量关于 MTD 的数据和操作函数。mtd_table（mtdcore.c）则是所有 MTD 原始设备的列表，mtd_part（mtd_part.c）用于表示 MTD 原始设备分区的结构，其中包含了 mtd_info，因为每一个分区都被看成一个 MTD 原始设备加在 mtd_table 中，mtd_part.mtd_info 中的大部分数据都从该分区的主分区 mtd_part→master 中获得。

在 drivers/mtd/maps/子目录下存放的是特定的 Flash 的数据，每一个文件都描述了一块板上的 Flash。其中调用 add_mtd_device()、del_mtd_device()建立/删除 mtd_info 结构并将其加入到 mtd_table/从 mtd_table 中删除（或者调用 add_mtd_partition()、del_mtd_partition()（mtdpart.c）建立/删除 mtd_part 结构并将 mtd_part.mtd_info 加入到 mtd_table 中/从 mtd_table 中删除）。

（3）MTD 设备层。基于 MTD 原始设备，Linux 系统可以定义出 MTD 的块设备（主设备号 31）和字符设备（主设备号 90）。MTD 字符设备的定义在 mtdchar.c 中实现，通过注册一系列 file operation 函数（lseek、open、close、read、write）。MTD 块设备是定义了一个描述 MTD 块设备的结构 mtdblk_dev，并声明了一个名为 mtdblks 的指针数组，这数组中的每一个 mtdblk_dev 和 mtd_table 中的每一个 mtd_info 一一对应。

（4）设备节点。通过 mknod 在/dev 子目录下建立 MTD 字符设备节点（主设备号为 90）和 MTD 块设备节点（主设备号为 31），通过访问此设备节点即可访问 MTD 字符设备和块设备。

（5）根文件系统。在 Bootloader 中将 JFFS（或 JFFS2）的文件系统映像 jffs.image（或 jffs2.img）烧写 Flash 的某一个分区中，在/arch/arm/mach-your/arch.c 文件的 your_fixup 函数中将该分区作为根文件系统挂载。

（6）文件系统。内核启动后，通过 mount 命令可以将 Flash 中的其余分区作为文件系统挂载到 mountpoint 上。

10.3　典型实例 1——NAND Flash 驱动实例

10.3.1　NAND Flash 驱动设备

这里以 2GB NAND 器件为例介绍，它由 2048 个区块组成，每个区块有 64 个页，如图 10-1 所示。2GB NAND 闪存包含 2 048 个区块。

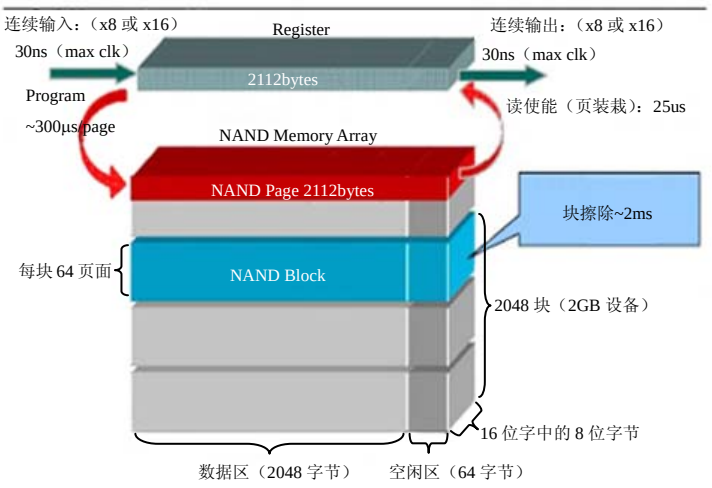


图 10-1　NAND 器件的闪存区块

一个页均包含一个 2 048 字节的数据区和 64 字节的空闲区，总共包含 2 112 字节。空闲区通常被用于 ECC、耗损均衡（wear leveling）和其他软件开销功能，尽管它在物理上与其他页并没有区别。NAND 器件具有 8 位或 16 位接口。通过 8 位或 16 位宽的双向数据总线，主数据被连接到 NAND 存储器。在 16 位模式，指令和地址仅仅利用低 8 位，而高 8 位在数据传输周期使用。

擦除区块所需时间约为 2ms。一旦数据被载入寄存器，对一个页的编程大约要 300μs。读一个页面需要大约 25μs，其中涉及存储阵列访问页，并将页载入 16 896 位寄存器中。

除了 I/O 总线，NAND 接口由 6 个主要控制信号构成。

（1）芯片启动（Chip Enable, CE#）：如果没有检测到 CE 信号，那么，NAND 器件就保持待机模式，不对任何控制信号作出响应。

（2）写使能（Write Enable, WE#）：WE#负责将数据、地址或指令写入 NAND 中。

（3）读使能（Read Enable, RE#）：RE#允许输出数据缓冲器。

（4）指令锁存使能（Command Latch Enable, CLE）：当 CLE 为高时，在 WE#信号的上升沿，指令被锁存到 NAND 指令寄存器中。

（5）地址锁存使能（Address Latch Enable, ALE）：当 ALE 为高时，在 WE#信号的上升沿，地址被锁存到 NAND 地址寄存器中。

（6）就绪/忙（Ready/Busy, R/B#）：如果 NAND 器件忙，R/B#信号将变低。该信号是漏极开路，需要采用上拉电阻。

数据每次进/出 NAND 寄存器都是通过 16 位或 8 位接口。当进行编程操作时，待编程的数据进入数据寄存器，处于在 WE#信号的上升沿。在寄存器内随机存取或移动数据，需要采用专用指令以便于随机存取。

10.3.2　NAND Flash 驱动源代码

```
/*
 * drivers/mtd/nand/ts7250.c
 *
 * Copyright (C) 2004 Technologic Systems (support@embeddedARM.com)
```

```
 *
 * Derived from drivers/mtd/nand/edb7312.c
 *   Copyright (C) 2004 Marius Grger (mag@sysgo.de)
 *
 * Derived from drivers/mtd/nand/autcpu12.c
 *   Copyright (c) 2001 Thomas Gleixner (gleixner@autronix.de)
 *
 * $Id：  ts7250.c,v 1.4 2004/12/30 22：02：07 joff Exp $
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * Overview：
 *   This is a device driver for the NAND FLASH device found on the
 *   TS-7250 board which utilizes a Samsung 32 Mbyte part.
 */
#include <Linux /slab.h>
#include <Linux /module.h>
#include <Linux /init.h>
#include <Linux /mtd/mtd.h>
#include <Linux /mtd/nand.h>
#include <Linux /mtd/partitions.h>
#include <asm/io.h>
#include <asm/arch/hardware.h>
#include <asm/sizes.h>
#include <asm/mach-types.h>
/*
 * MTD 结构体
 */
static struct mtd_info *ts7250_mtd = NULL;
#ifdef CONFIG_MTD_PARTITIONS
static const char *part_probes[] = { "cmdlinepart", NULL };
#define NUM_PARTITIONS 3
/*
 * 定义 Flash 设备的静态分区
 */
static struct mtd_partition partition_info32[] = {
    {
        .name= "TS-BOOTROM",
        .offset= 0x00000000,
        .size= 0x00004000,
    }, {
        .name= "Linux",
        .offset= 0x00004000,
        .size= 0x01d00000,
    }, {
        .name= "RedBoot",
        .offset= 0x01d04000,
        .size= 0x002fc000,
    },
};
/*
 * 定义 Flash 设备的静态分区
 */
static struct mtd_partition partition_info128[] = {
```

```
        {
            .name= "BOOTROM",
            .offset= 0x00000000,
            .size= 0x00004000
        }, {
            .name= "Linux",
            .offset= 0x00004000,
            .size= 0x07d00000,
        }, {
            .name= "uBoot",
            .offset= 0x07d04000,
            .size= 0x002fc000,
        },
    };
    #endif
    /* 硬件特定的读写控制如下：
     *  NAND_NCE：  bit 0 -> bit 2
     *  NAND_CLE：  bit 1 -> bit 1
     *  NAND_ALE：  bit 2 -> bit 0
     */
    static void ts7250_hwcontrol(struct mtd_info *mtd, int cmd, unsigned int
ctrl)
    {
        struct nand_chip *chip = mtd->priv;
        if (ctrl & NAND_CTRL_CHANGE) {
            unsigned long addr = TS72XX_NAND_CONTROL_VIRT_BASE;
            unsigned char bits;
            bits = (ctrl & NAND_NCE) << 2;
            bits |= ctrl & NAND_CLE;
            bits |= (ctrl & NAND_ALE) >> 2;
            __raw_writeb((__raw_readb(addr) & ~0x7) | bits, addr);
        }
        if (cmd != NAND_CMD_NONE)
            writeb(cmd, chip->IO_ADDR_W);
    }
    /*
     *  设备读 ready
     */
    static int ts7250_device_ready(struct mtd_info *mtd)
    {
        return __raw_readb(TS72XX_NAND_BUSY_VIRT_BASE) & 0x20;
    }
    /*主初始化例程如下：
     */
    static int __init ts7250_init(void)
    {
        struct nand_chip *this;
        const char *part_type = 0;
        int mtd_parts_nb = 0;
        struct mtd_partition *mtd_parts = 0;
        if (!machine_is_ts72xx() || board_is_ts7200())
            return -ENXIO;
        /* 为 MTD 设备结构和私有数据配置存储 */
        ts7250_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip),
GFP_KERNEL);
        if (!ts7250_mtd) {
```

```
            printk("Unable to allocate TS7250 NAND MTD device structure.\n");
            return -ENOMEM;
        }
        /* 为数据获取指针 */
        this = (struct nand_chip *)(&ts7250_mtd[1]);
        /* 初始化结构内存*/
        memset(ts7250_mtd, 0, sizeof(struct mtd_info));
        memset(this, 0, sizeof(struct nand_chip));
        /*把私有数据链接到 MTD structure */
        ts7250_mtd->priv = this;
        ts7250_mtd->owner = THIS_MODULE;
        /* 插入回叫信号 */
        this->IO_ADDR_R = (void *)TS72XX_NAND_DATA_VIRT_BASE;
        this->IO_ADDR_W = (void *)TS72XX_NAND_DATA_VIRT_BASE;
        this->cmd_ctrl = ts7250_hwcontrol;
        this->dev_ready = ts7250_device_ready;
        this->chip_delay = 15;
        this->ecc.mode = NAND_ECC_SOFT;
        printk("Searching for NAND FLASH...\n");
        /* 扫描以发现存在的设置 */
        if (nand_scan(ts7250_mtd, 1)) {
            kfree(ts7250_mtd);
            return -ENXIO;
        }
    #ifdef CONFIG_MTD_PARTITIONS
        ts7250_mtd->name = "ts7250-nand";
        mtd_parts_nb    =    parse_mtd_partitions(ts7250_mtd,    part_probes,
&mtd_parts, 0);
        if (mtd_parts_nb > 0)
            part_type = "command line";
        else
            mtd_parts_nb = 0;
    #endif
        if (mtd_parts_nb == 0) {
            mtd_parts = partition_info32;
            if (ts7250_mtd->size >= (128 * 0x100000))
                mtd_parts = partition_info128;
            mtd_parts_nb = NUM_PARTITIONS;
            part_type = "static";
        }
        /* 注册分区 */
        printk(KERN_NOTICE "Using %s partition definition\n", part_type);
        add_mtd_partitions(ts7250_mtd, mtd_parts, mtd_parts_nb);
        恰当地返回
        return 0;
    }
    module_init(ts7250_init);
    /*
     * 注销函数
     */
    static void __exit ts7250_cleanup(void)
    {
        未注册的设备
        del_mtd_device(ts7250_mtd);
        解除 MTD 设备装置
        kfree(ts7250_mtd);
```

```
}
module_exit(ts7250_cleanup);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jesse Off <joff@embeddedARM.com>");
MODULE_DESCRIPTION("MTD map driver for Technologic Systems TS-7250 board");
```

10.4 典型实例 2——NOR Flash 驱动实例

分析驱动原理与实现代码之前，首先介绍芯片驱动与 MTD 原始设备的知识。

10.4.1 芯片驱动与 MTD 原始设备

所有的 NOR 型 Flash 的驱动（探测 probe）程序都放在 drivers/mtd/chips 下，一个 MTD 原始设备可以由一块或者数块相同的 Flash 芯片组成。假设有 4 块设备类型为 x8 的 Flash，每块大小为 8M，交错通道为 2，起始地址为 0x01000000，地址相连，则构成一个 MTD 原始设备（0x01000000-0x03000000），其中两块交错通道成一个基片，其地址从 0x01000000 到 0x02000000；另两块交错通道成一个基片，其地址从 0x02000000 到 0x03000000。

所有组成一个 MTD 原始设备的 Flash 芯片必须是同类型的（无论是交错通道还是地址相连），在描述 MTD 原始设备的数据结构中也只是采用了同一个结构来描述组成它的 Flash 芯片，如图 10-2 所示。

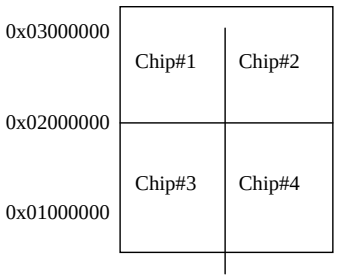


图 10-2　MTD 原始设备的数据结构描述

每个 MTD 原始设备都有一个 mtd_info 结构，其中的 priv 指针指向一个 map_info 结构，map_info 结构中的 fldrv_priv 指向一个 cfi_private 结构，cfi_private 结构的 cfiq 指针指向一个 cfi_ident 结构，chips 指针指向一个 flchip 结构的数组。其中 mtd_info、map_info 和 cfi_private 结构用于描述 MTD 原始设备；因为组成 MTD 原始设备的 NOR 型 Flash 相同，cfi_ident 结构用于描述 Flash 芯片的信息；而 flchip 结构用于描述每个 Flash 芯片的专有信息（比如起始地址等）。

10.4.2 NOR Flash 驱动分析

（1）drivers/mtd/maps 子目录

此目录下的每一个文件为一个具体的 MTD 原始设备的相关信息，包括该 MTD 原始设备的起始物理地址、大小、分区情况、读写函数、初始化和清除程序。每一个 MTD 驱动工程师的代码都应该写成一个文件放在这里。下面以 your-Flash.c 为例具体解释一下。

假设这个 MTD 原始设备是一个 NOR 型的 Flash，起始物理地址是 0x1000000，大小为 16M，总线宽度为 16 位；设备驱动程序需要的字段和函数说明如下。

WINDOW_ADDR：MTD 原始设备的起始地址（PA）。

定义为：#define WINDOW_ADDR 0x01000000

WINDOW_SIZE：MTD 原始设备的大小。

定义为：#define WINDOW_SIZE 0x02000000

大小：16M

BUSWIDTH：总线宽度。

定义为：#define BUSWIDTH 2

大小：16 位

mymtd：由 do_mtd_probe 搜索出的 MTD 设备主分区。

定义为：static struct mtd_info *mymtd;

（2）your_read8

格式：static __u8 your_read8(struct map_info *map,unsigned long ofs)

功能：读 Flash。

说明：NOR Flash 可以和内存一样读写，但是需要先进行一些设置，然后调用此函数进行读操作（写函数相同）。

注册调用：注册进 your_map

源代码如下：

```
{
      return __raw_readb(map->map_priv_1 + ofs);
}
your_read16
your_read32
your_copy_from
your_write8
your_write16
your_write32
your_copy_to
```

注册调用：注册进 your-map

（3）your_set_vpp

格式：static void your_set_vpp(struct map_info *map,int set)

功能：设置 Flash 的读写状态。

说明：向寄存器 SOME_REGISTER 写值，打开或去掉 Flash 的写保护（Write Protection）。

参数包括：

- map：Flash 芯片信息。
- set：为 1 设置为写状态，为 0 设置为读状态。

返回：无。

调用参数：

- __raw_writel()
- __raw_readl()

注册调用：

注册进 your_map

源代码如下：

```
{
       if(set)                    //取消写保护
              __raw_writel(__raw_readl(SOME_REGISTER)|0x00000001, SOME_REGISTER)
       else
              __raw_writel(__raw_readl(SOMEREGISTER)&0xfffffffe, SOME_REGISTER);
}
```

（4）your_partition

假设板 Flash 的分区结构如下，16MB 大小的 Flash 被分为 3 个区。

0—8KB：用于装载 Bootloader。

8KB—8MB：用于放置内核映像。

8MB—16MB：用于存放根文件系统（JFFS 或 JFFS2）的映像。

```
static struct mtd_partition your_partition[]={
   {
         name："Your Bootloader",
         offset：0,
         size：0x20000,
         mask_flags：MTD_WRITEABLE,  /* force read-only *///只读
   },
   {
        name："Your Kernel",
        offset：0x20000,
        size：0x7e0000,
        mask_flags：  MTD_WRITEABLE,  /* force read-only *///只读
   },
   {
         name："Your Rootfs",
         offset：0x800000,
         size：0x800000,
   }
};
```

（5）NUM_PARTITIONS

此参数功能表示分区的个数（3）。

```
#define NUM_PARTITIONS (sizeof(your_partition)/sizeof(your_partition[0]))
your_map
struct map_info your_map = {
      name："Your FLASH map",
      size：WINDOW_SIZE,
      buswidth：BUSWIDTH,
      read8：your_read8,
      read16：your_read16,
      read32：your_read32,
      copy_from：your_copy_from,
      write8：your_write8,
      write16：your_write16,
      write32：your_write32,
      copy_to：your_copy_to,
```

```
        set_vpp：your_set_vpp,
    };
```

（6）init_yourFlash

格式：static　int __init init_yourFlash (void)

注释：无。

功能：初始化 Flash。

说明：

- 调用 do_map_probe()搜索 MTD 设备并将其赋给 mymtd。
- 调用 add_mtd_partitions()将 your_partiton 的各个分区加入 mtd_table。

参数：无。

返回：

- 成功：返回 0。
- 失败：返回错误码。

调用：

- do_map_probe()搜索 MTD 设备。
- add_mtd_partitions()将分区加入 mtd_table。

被调用：

- -init
- module-init

源代码如下：

```
    {
        printk(KERN_NOTICE "Your FLASH mapping：size %x at %x\n", WINDOW_SIZE,
WINDOW_ADDR);
        your_map.map_priv_1 = (unsigned long)ioremap(WINDOW_ADDR, WINDOW_SIZE);
        if (!your_map.map_priv_1) {
            return -EIO;
        }
        mymtd = do_map_probe("cfi_probe", &your_map);
        if (mymtd) {
            mymtd->module = THIS_MODULE;
            add_mtd_partitions( mymtd, your_partition, NUM_PARTITIONS );
            //add_mtd_device(mymtd);
            return 0;
        }
        iounmap((void *)your_map.map_priv_1);
        return -ENXIO;
    }
```

（7）cleanup_yourFlash

格式：mod_exit_t cleanup_yourFlash (void)

注释：无。

功能：清除 Flash。

说明：调用 del_mtd_partitions 删除 mtd_table 中的 MTD 设备。

调用：

- del_mtd_partitions()

- map_destroy()

被调用：

- -exit
- module-exit

源代码如下：

```
{
      if (mymtd) {
            del_mtd_partitions(mymtd);
            map_destroy(mymtd);
      }
      if (your_map.map_priv_1) {
            iounmap((void *)your_map.map_priv_1);
            your_map.map_priv_1 = 0;
      }
}
```

10.4.3 NOR Flash 驱动源代码

```
*
 * Flash memory access on Hynix GMS30C7201/HMS30C7202 based
 * evaluation boards
 *
 * $Id： h720x-Flash.c,v 1.12 2005/11/07 11：14：27 gleixner Exp $
 *
 * (C) 2002 Jungjun Kim <jungjun.kim@hynix.com>
 *     2003 Thomas Gleixner <tglx@linutronix.de>
 */
#include <Linux /module.h>
#include <Linux /types.h>
#include <Linux /kernel.h>
#include <Linux /init.h>
#include <Linux /errno.h>
#include <Linux /slab.h>
#include <Linux /mtd/mtd.h>
#include <Linux /mtd/map.h>
#include <Linux /mtd/partitions.h>
#include <asm/hardware.h>
#include <asm/io.h>
static struct mtd_info *mymtd;
/*设备描述*/
static struct map_info h720x_map = {
    .name ="H720X",
    .bankwidth =4,
    .size =Flash_SIZE,
    .phys =Flash_PHYS,
};
/*静态设备分区*/
static struct mtd_partition h720x_partitions[] = {
       {
              .name = "ArMon",
              .size = 0x00080000,
              .offset = 0,
              .mask_flags = MTD_WRITEABLE
       },{
```

```
                .name = "Env",
                .size = 0x00040000,
                .offset = 0x00080000,
                .mask_flags = MTD_WRITEABLE
        },{
                .name = "Kernel",
                .size = 0x00180000,
                .offset = 0x000c0000,
                .mask_flags = MTD_WRITEABLE
        },{
                .name = "Ramdisk",
                .size = 0x00400000,
                .offset = 0x00240000,
                .mask_flags = MTD_WRITEABLE
        },{
                .name = "jffs2",
                .size = MTDPART_SIZ_FULL,
                .offset = MTDPART_OFS_APPEND
        }
};
#define NUM_PARTITIONS ARRAY_SIZE(h720x_partitions)
/*分区的数量*/
static intnr_mtd_parts;
static struct mtd_partition *mtd_parts;
static const char *probes[] = { "cmdlinepart", NULL };
/*
 * 初始化 Flash
 */
int __init h720x_mtd_init(void)
{
    char    *part_type = NULL;
    h720x_map.virt = ioremap(Flash_PHYS, FLASH_SIZE);
    if (!h720x_map.virt) {
        printk(KERN_ERR "H720x-MTD：ioremap failed\n");
        return -EIO;
    }
    simple_map_init(&h720x_map);
    /* Flash 驱动 Probe，调用 cfi 函数*/
    printk (KERN_INFO "H720x-MTD probing 32bit Flash\n");
    mymtd = do_map_probe("cfi_probe", &h720x_map);
    if (!mymtd) {
        printk (KERN_INFO "H720x-MTD probing 16bit Flash\n");
       // Probe for bankwidth 2
       h720x_map.bankwidth = 2;
       mymtd = do_map_probe("cfi_probe", &h720x_map);
    }
    if (mymtd) {
        mymtd->owner = THIS_MODULE;
/*如果配置分区，把这个 Flash 设备按照上面的定义分成几个 Linux 分区*/
#ifdef CONFIG_MTD_PARTITIONS
        nr_mtd_parts = parse_mtd_partitions(mymtd, probes, &mtd_parts, 0);
        if (nr_mtd_parts > 0)
            part_type = "command line";
#endif
        if (nr_mtd_parts <= 0) {
            mtd_parts = h720x_partitions;
```

```
            nr_mtd_parts = NUM_PARTITIONS;
            part_type = "builtin";
        }
        printk(KERN_INFO "Using %s partition table\n", part_type);
        add_mtd_partitions(mymtd, mtd_parts, nr_mtd_parts);
        return 0;
    }
    iounmap((void *)h720x_map.virt);
    return -ENXIO;
}
/*
 * 注销函数
 */
static void __exit h720x_mtd_cleanup(void)
{
    if (mymtd) {
        del_mtd_partitions(mymtd);
        map_destroy(mymtd);
    }
    /* 释放分区结构体占用的内存 */
    if (mtd_parts && (mtd_parts != h720x_partitions))
        kfree (mtd_parts);
    if (h720x_map.virt) {
        iounmap((void *)h720x_map.virt);
        h720x_map.virt = 0;
    }
}
module_init(h720x_mtd_init);
module_exit(h720x_mtd_cleanup);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Thomas Gleixner <tglx@linutronix.de>");
MODULE_DESCRIPTION("MTD map driver for Hynix evaluation boards");
```

10.5 本章总结

本章首先介绍了 Flash 闪存驱动开发的基础知识，然后分别介绍了 NAND Flash 驱动和 NOR Flash 驱动两种 Flash 驱动技术，并分别对其设备程序做了较详细的代码分析。通过本章的学习，读者可以掌握 Flash 闪存驱动开发的原理与实现过程。

第 3 篇

Linux 系统开发实例

- 第 11 章　嵌入式系统开发的模式与流程
- 第 12 章　工业温度监控设备开发实例
- 第 13 章　实时视频采集系统开发实例
- 第 14 章　指纹识别门禁系统开发实例
- 第 15 章　基于 RTL8019 的以太网应用系统开发实例
- 第 16 章　无线网络数据传输系统开发实例
- 第 17 章　基于 PDIUSBD12 的数据传输系统实例
- 第 18 章　家庭安全监控系统设计实例
- 第 19 章　移动校园系统设计实例

第 11 章

嵌入式系统开发的模式与流程

本章重点介绍嵌入式开发的模式与流程，为后面的实例学习打下坚实的基础。首先介绍嵌入式系统的软硬件结构。

11.1 嵌入式系统的结构

11.1.1 嵌入式系统的硬件架构

图 11-1 为嵌入式系统硬件模型结构，此系统主要由微处理器 MPU、外围电路以及外设组成。微处理器为 ARM 嵌入式处理芯片，如 ARM7TMDI 系列及 ARM9 系列微处理器，MPU 为整个嵌入式系统硬件的核心，决定了整个系统功能和应用领域。外围电路根据微处理器不同而略有不同，主要由电源管理模型、时钟模块、闪存 Flash、随机存储器 RAM 以及只读存储器 ROM 组成。这些设备是一个微处理器正常工作所必须的设备。外部设备将根据需要而各不相同，如通用通信接口 USB、RS-232、RJ-45 等，输入输出设备如键盘、LCD 等。外部设备将根据需要定制。

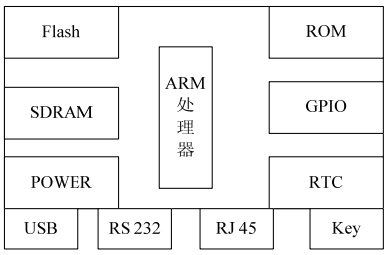


图 11-1 典型嵌入式系统硬件结构

嵌入式处理系统主要包括嵌入式微处理器、存储设备、模拟电路及电源电路、通信接口以及外设电路。

11.1.2 嵌入式系统的软件结构

嵌入式系统与传统的单片机在软件方面最大的不同就是可以移植操作系统，从而使软件设计层次化，传统的单片机在软件设计时将应用程序与系统、驱动等全部混在一起编译，系统的可扩展性、可维护性不高，上升到操作系统后，这一切变得简单可行。

嵌入式操作系统在软件上呈现明显的层次化，从与硬件相关的 BSP 到实时操作系统内

核 RTOS，到上层文件系统、GUI 界面，以及用户层的应用软件。各部分可以清晰地划分开来，如图 11-2 所示。当然在某些时候这种划分也不完全符合应用要求，需要程序设计人员根据特定的需要来设计自己的软件。

<table>
<tr><td colspan="3">应用程序层（Application）</td></tr>
<tr><td>FS 文件系统</td><td>图形界面 GUI</td><td>系统管理接口</td></tr>
<tr><td colspan="3">实时操作系统内核系统（RTOS）</td></tr>
<tr><td colspan="3">板级支持包（BSP）</td></tr>
<tr><td colspan="3">硬件层</td></tr>
</table>

图 11-2　嵌入式系统软件基本架构

板级支持包（Board Support Packet）主要用来完成底层硬件相关的信息，如驱动程序、加载实时操作系统等功能。

实时操作系统层主要就是常见的嵌入式操作系统，设计者根据自己特定的需要来设计移植自己的操作系统，即添加删除部分组件，添加相应的硬件驱动程序，为上层应用提供系统调用。

文件系统、图形界面以及系统管理接口主要应对需要，即如果需要文件系统及图形界面支持才需要设计，主要是为应用程序员开发应用程序提供更多更便捷丰富的 API 接口。

应用软件层即用户设计的针对特定应用的应用软件，在开发该应用软件时，可以用到底层提供的大量函数。

采用分层结构的软件设计使系统清晰明了，各个部分设计工作分工明确，从而避免整个系统过分庞大。

11.2　嵌入式开发的模式及流程

11.2.1　嵌入式系统开发模式

嵌入式系统开发分为软件开发部分和硬件开发部分。嵌入式系统在开发过程一般都采用如图 11-3 所示的“宿主机/目标板”开发模式，即利用宿主机（PC）上丰富的软硬件资源及良好的开发环境和调试工具来开发目标板上的软件，然后通过交叉编译环境生成目标代码和可执行文件，通过串口/USB/以太网等方式下载到目标板上，利用交叉调试器在监控程序运行，实时分析，最后将程序下载固化到目标机上，完成整个开发过程。

在软件设计上，图 11-4 为结合 ARM 硬件环境及 ADS 软件开发环境所设计的嵌入式系统开发流程图。整个开发过程基本包括以下几个步骤。

（1）源代码编写：编写源 C/C++及汇编程序；

（2）程序编译：通过专用编译器编译程序；

（3）软件仿真调试：在 SDK 中仿真软件运行情况；

（4）程序下载：通过 JTAG、USB、UART 方式下载到目标板上；

（5）软硬件测试、调试：通过 JTAG 等方式联合调试程序；

（6）下载固化：程序无误，下载到产品上生产。

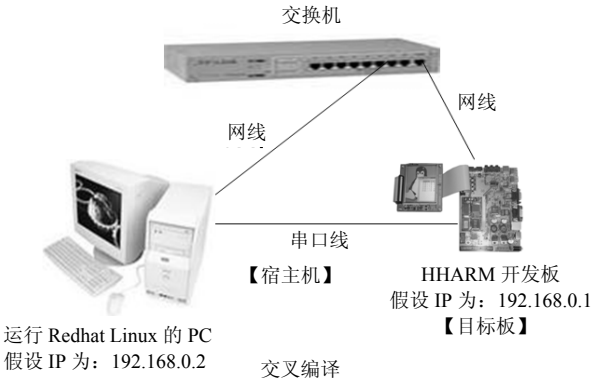


图 11-3 “宿主机/目标板”开发模式

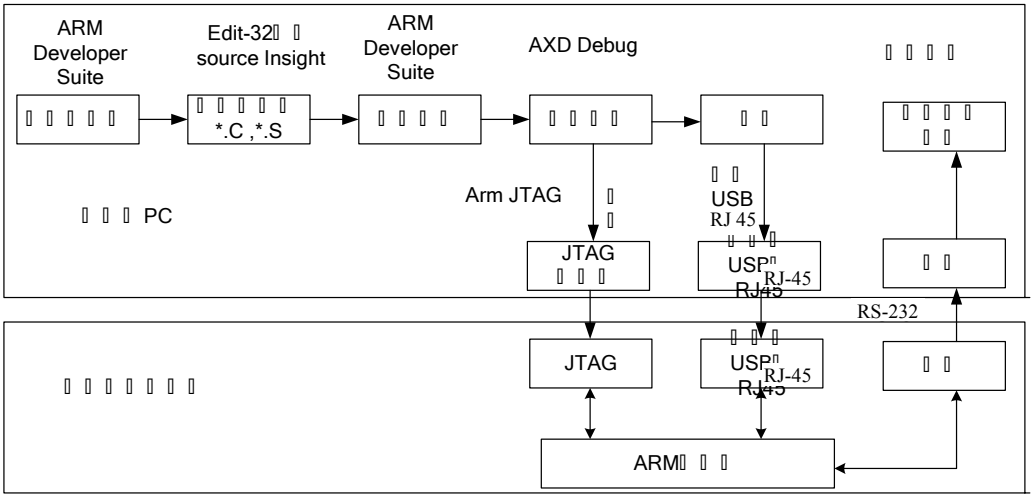


图 11-4 嵌入式系统软件开发流程

11.2.2 嵌入式系统开发流程

当前，嵌入式开发已经逐步规范化，在遵循一般工程开发流程的基础上，嵌入式开发有其自身的一些特点，图 11-5 为嵌入式系统开发的一般流程。主要包括系统需求分析（要求有严格规范的技术要求）、体系结构设计、软硬件及机械系统设计、系统集成、系统测试，最终得到产品。

（1）系统需求分析。确定设计任务和设计目标，并提炼出设计规格说明书，作为正式设计指导和验收的标准。系统的需求一般分功能性需求和非功能性需求两方面。功能性需求是系统的基本功能，如输入输出信号、操作方式等；非功能需求包括系统性能、成本、功耗、体积、重量等因素。

（2）体系结构设计。描述系统如何实现所述的功能和非功能需求，包括对硬件、软件和执行装置的功能划分，以及系统的软件、硬件选型等。一个好的体系结构是设计成功与否的关键。

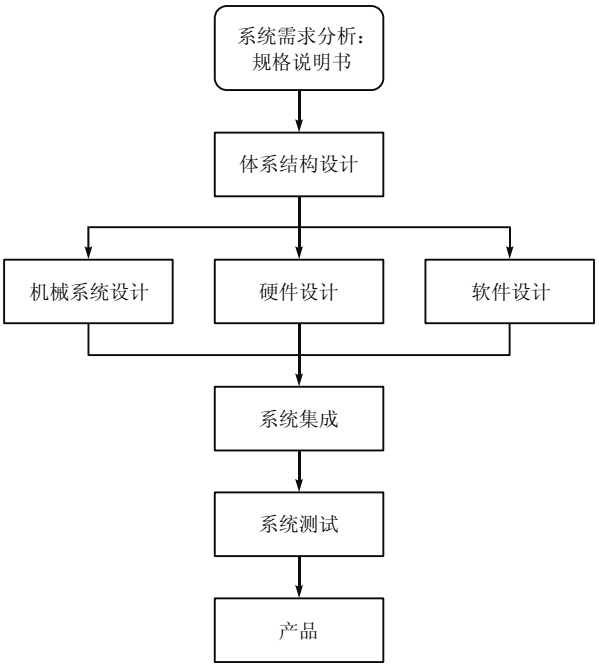


图 11-5　嵌入式开发流程

（3）硬件/软件协同设计。基于体系结构，对系统的软件、硬件进行详细设计。为了缩短产品开发周期，设计往往是并行的。嵌入式系统设计的工作大部分都集中在软件设计上，采用面向对象技术、软件组件技术、模块化设计是现代软件工程经常采用的方法。

（4）系统集成。把系统的软件、硬件和执行装置集成在一起，进行调试，发现并改进单元设计过程中的错误。

（5）系统测试。对设计好的系统进行测试，看其是否满足规格说明书中给定的功能要求。

嵌入式系统开发模式的最大特点是软件、硬件综合开发。这是因为嵌入式产品是软硬件的结合体，软件针对硬件开发、固化，不可修改。

如果在一个嵌入式系统中使用 Linux 技术开发，根据应用需求的不同有不同的配置开发方法，但是，一般情况下都需要经过如下过程。

（1）建立开发环境，操作系统一般使用 Redhat Linux，选择定制安装或全部安装，通过网络下载相应的 GCC 交叉编译器进行安装（比如 arm-linux-gcc、arm-uclibc-gcc），或者安装产品厂家提供的相关交叉编译器。

（2）配置开发主机，配置 MINICOM，一般的参数为波特率 115200 Baud/s，数据位 8 位，停止位为 1，9，无奇偶校验，软件硬件流控设为无。在 Windows 下的超级终端的配置也是这样。MINICOM 软件的作用是作为调试嵌入式开发板的信息输出的监视器和键盘输入的工具。配置网络主要是配置 NFS 网络文件系统，需要关闭防火墙，简化嵌入式网络调试环境设置过程。

（3）建立引导装载程序 BOOTLOADER，从网络上下载一些公开源代码的 BOOTLOADER，如 U-BOOT、BLOB、VIVI、LILO、ARM-BOOT、RED-BOOT 等，根据具体芯片进行移植修改。有些芯片没有内置引导装载程序，比如三星的 ARM7、ARM9 系

列芯片，这样就需要编写开发板上 Flash 的烧写程序，读者可以在网上下载相应的烧写程序，也有 Linux 下的公开源代码的 J-Flash 程序。如果不能烧写自己的开发板，就需要根据自己的具体电路进行源代码修改。这是让系统可以正常运行的第一步。如果用户购买了厂家的仿真器比较容易烧写 Flash，虽然无法了解其中的核心技术，但对于需要迅速开发自己的应用的人来说可以极大提高开发速度。

（4）下载已经移植好的 Linux 操作系统，如 MCLiunx、ARM-Linux、PPC-Linux 等，如果有专门针对所使用的 CPU 移植好的 Linux 操作系统那是再好不过，下载后再添加特定硬件的驱动程序，然后进行调试修改，对于带 MMU 的 CPU 可以使用模块方式调试驱动，而对于 MCLiunx 这样的系统只能编译内核进行调试。

（5）建立根文件系统，可以从 http://www.busybox.net 下载使用 BUSYBOX 软件进行功能裁剪，产生一个最基本的根文件系统，再根据自己的应用需要添加其他的程序。由于默认的启动脚本一般都不会符合应用的需要，所以就要修改根文件系统中的启动脚本，它的存放位置位于/etc 目录下，包括/etc/init.d/rc.S、/etc/profile、/etc/.profile 等，自动挂装文件系统的配置文件/etc/fstab，具体情况会随系统不同而不同。根文件系统在嵌入式系统中一般设为只读，需要使用 mkcramfs genromfs 等工具产生烧写映像文件。

（6）建立应用程序的 Flash 磁盘分区，一般使用 JFFS2 或 YAFFS 文件系统，这需要在内核中提供这些文件系统的驱动，有的系统使用一个线性 Flash（NOR 型）512KB~ 32MB，有的系统使用非线性 Flash（NAND 型）8MB~512MB，有的两个同时使用，需要根据应用规划 Flash 的分区方案。

（7）开发应用程序，可以放入根文件系统中，也可以放入 YAFFS、JFFS2 文件系统中，有的应用不使用根文件系统，直接将应用程序和内核设计在一起，这有点类似于μC/OS-II 的方式。

（8）烧写内核、根文件系统和应用程序，发布产品。

11.3 本章总结

本章简单介绍了嵌入式开发的模式与流程，内容包括嵌入式系统的硬件架构、嵌入式系统的软件架构、嵌入式系统的开发模式和嵌入式系统的开发流程。通过本章的学习，读者将熟悉嵌入式系统的结构、模式与开发流程，在以后的实际系统案例设计中做到有章可循，提高设计质量和效率。

第 12 章

工业温度监控设备开发实例

随着计算机技术、通信技术、消费电子技术为主的电子信息技术的高速发展和国际互联网（Internet）的广泛应用，人们的生活方式已经逐步发生了改变。信息技术对其他各产业的贡献越来越大，从而逐渐成为其他产业的支柱。

以单片机或微控制器为核心的电子系统在汽车电子、工业控制等领域得到广泛应用，MCU 与检测传感器、伺服机构和人机界面等构成能够实现一定功能的嵌入式系统。目前大多数嵌入式系统都是独立应用的，在一些工业和汽车应用领域为了实现多个 MCU（或子系统）之间信息交流，采用了 CAN、RS 485 等现场总线组网。这些网络的有效半径比较低，网络协议比较简单，并且它们都孤立于 Internet 之外。

随着 Internet 的飞速发展，网络应用日益广泛，对各种工业控制设备的网络功能要求逐渐提高。市场希望工业控制设备能够支持 TCP/IP 以及其他 Internet 协议，用户能够通过熟悉的浏览器查看设备状态、设置设备参数，或者将设备采集到的数据通过网络传送到 Windows 或 UNIX/Linux 服务器的数据库中，从而实现嵌入式系统利用 Internet 技术也可以实现低成本网络互联、信息沟通。图 12-1 是基于 TCP/IP 协议的工业控制系统模型。

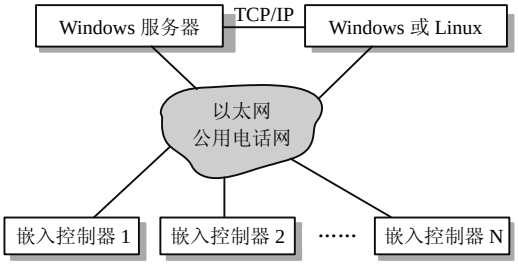


图 12-1　基于 TCP/IP 协议的工业控制系统模型

12.1　应用环境与硬件设计概要

12.1.1　嵌入式 Linux 在工业控制领域的应用

随着嵌入式技术及网络技术的发展，工业控制设备的网络功能要求也越来越高，系统要求工业控制设备能够支持 TCP/IP 以及其他 Internet 协议，用户能够方便地查看设备状态、设置设备参数、设备采集到的数据通过网络传送到 Windows 或 UNIX/Linux 服务器的数据库中。这要求工控系统必须具备以下两方面的功能。

- 在现场能够完成复杂的测控任务，通常一些任务都具有一定的实时性要求，这要求硬件系统有较强的处理能力和较强的实时性；
- 测控系统能够与某一类型的控制网络相连，以实现远程监控。但目前应用的大多数测控系统中，嵌入式系统的硬件采用的是 8/16 位单片机，软件多采用汇编语言编程，这些程序仅包含一些简单的循环处理控制流程。单片机与单片机或上位机之间的通信通常通过 RS-232、RS-485 来实现。这些网络存在通信速度慢、联网功能差、开发困难等问题。因此，目前的软硬件系统无法满足市场的需求。这就要求当前的嵌入式系统上能够运行操作系统，从而方便、可靠地实现多种通信。

目前，随着 32 位嵌入式 CPU 价格的下降和性能指标的提高，嵌入式系统的广泛应用成为可能，以嵌入式 Linux 为操作系统的软件系统也得到了前所未有的发展。基于嵌入式 Linux 的工控系统以嵌入式微处理器为核心，以嵌入式 Linux 为操作系统，应用程序可以方便地通过网络进行更新，并可通过键盘进行人机对话，数据可通过 LCD 现场显示，重要数据可用文件形式保存在 Flash 等闪存存储器中，数据和报警信息可通过串口向上位机传输，也可以通过以太网向工业以太网或 Internet 发布，用户还可通过网络实现远程监控和远程维护。更为关键的是，可充分利用 Internet 上已有的软件和协议（如 FTP、HTTP、Apache＋PHP＋MySQL 等应用程序）迅速搭建前台数据采集系统，以实现测控系统和后台管理系统的通信。这种方式的优点有：

（1）不需专用的通信线路即可用现成的 Internet 网络将数据传送到任何地方；

（2）不仅能够传递数据信号，也可以传递音频和图像信号；

（3）由于目前的 Internet 协议是公开的，因此，利用大到几十兆的 IE 浏览器，或小到只有 600KB 的 MOSAIC 浏览器都可以对网络数据进行读取。

图 12-2 是一个基于嵌入式 Linux 的工业控制系统一个节点的硬件系统的网络架构。在此系统中，采用 32 位嵌入式微处理器为硬件核心芯片，外部通过 RS-232 通信接口作自由上传和下载接口，通过显示接口、键盘接口 I/O 模块扩展接口实现人机通话，另外，整个系统与外界通过以太网接口实现高速实时数据互访。此硬件系统可以方便地移植嵌入式操作系统，扩展应用程序，同时，所有数据都可以便捷、可靠地传输到远端。

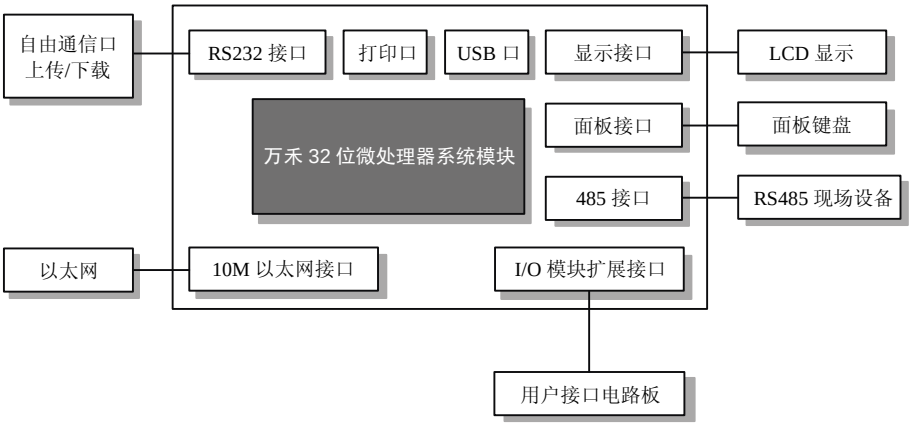


图 12-2　基于嵌入式 Linux 的工业控制系统

在软件系统设计方面，嵌入式 Linux 操作系统是整个嵌入式系统的软件核心，由于嵌

入式系统在内存容量和存储容量不足等问题，必须对 Linux 进行移植裁剪设计。在软件开发方面，主要涉及以下几个步骤的工作。

（1）内核的精简

由于标准 Linux 是面向 PC 的，它集成了许多 PC 所需要而嵌入式系统并不需要的功能。因此，对一些可独立加上或卸下的功能块，可在编译内核时仅保留嵌入式系统所需的功能模块，而删除不需要的功能块，这样重新编译过的内核就会显著减小，从而尽可能小地占用内存及存储器空间，使软硬件系统精简。

（2）虚拟内存机制的屏蔽

虚拟内存是导致现有 Linux 实时性不强的原因之一，在工业控制中，某些任务要满足一定的实时性要求，屏蔽内核的虚拟内存管理机制可以增强 Linux 的实时性。同时由于 Linux 系统对应用进程采用的是公平的时间分配调度算法，从而不能保证系统的实时性要求，因此要求对其进行更改。更改途径有两种：一是通过 POSIX，二是通过底层编程。

（3）设备驱动程序的编写

确定了内核的基本功能后，就需要为特定的设备编写驱动程序，可按照在 Linux 下编写驱动程序的规则进行编写。编写的设备驱动程序需要具有以下功能。

- 对设备进行初始化和释放；
- 完成数据从内核到硬件设备的传送和从硬件读取数据两项功能；
- 读取应用程序传递给设备文件的数据及回送应用程序请求的数据；
- 检测和处理设备出现的错误。

（4）开发基于闪存的文件系统 JFFS2

应用程序和重要数据通常以文件的形式被存放在闪存文件系统中。JFFS2 文件系统是日志结构化的，这意味着它基本上是一长列节点，每个节点包含着有关文件的部分信息。JFFS2 是专门为闪存芯片那样的嵌入式设备创建的，由于它的整个设计提供了更好的闪存管理，因而具有其他文件系统不可比拟的优点。具体特点如下：

- JFFS2 在扇区级别上执行闪存擦除/写/读操作要比 EXT2 文件系统好；
- JFFS2 提供了比 Ext2FS 更好的崩溃/掉电安全保护。当需要更改少量数据时，Ext2FS 文件系统会将整个扇区复制到内存（DRAM）中，并在内存中合并成新数据再写回整个扇区。而 JFFS2 则可以随时更改需要的（不是重写）整个扇区，同时还具有崩溃 / 掉电安全保护功能。

（5）上层应用程序的编写

根据系统应用需要，针对特定的系统应用环境，从而实现系统上层应用程序的编写。

实现上述几个步骤后，一个小型的 Linux 操作系统就构造完成了，整个 Linux 包括进程管理、内存管理和文件管理 3 个部分。它支持多任务并行，有完整的 TCP/IP 协议，同时 Linux 内建有对以太网控制器的支持，可以通过以太网接口连到以太网上，从而实现远程配置与监控。将裁剪好的内核移植到所用的目标板上时，首先应将内核编译成专用于该处理器的目标代码。由于不同硬件体系的移植启动代码会有所不同，因此，一些内核程序可能要改写。涉及编写 Linux 的引导代码和修改与体系结构相关部分的代码主要是启动引导、内存管理和中断处理部分。

整个嵌入式系统加电后，引导代码即可进行基本的硬件初始化，然后把内核映像装入内存并运行，最后，再将调试好的内核和应用程序烧录到闪存中。由于此时裁剪后的 Linux

已成功移植到目标平台上，因此，在启动可运行的开发系统时，就可以根据具体的应用来开发应用程序，如数据采集模块、数据处理模块、通信和数据发布模块等。

如今，把嵌入式 Linux 内核嵌入到 32 位 MCU 工业控制系统中，通过构造 TCP/IP 多种网络协议和基本网络通信协议，再利用嵌入式操作系统对底层硬件和网络协议的支持，以及对工控系统实时性要求的 Linux 内核和虚拟内存机制进行改造，即可保证测控任务完成的实时性和可靠性。这种方案在工业控制领域具有很好的应用前景，而且具有开发周期短、系统性能稳定可靠、适应性强等特点。

12.1.2 工控串行通信协议标准

目前，标准串行数据传输协议主要有 RS-232C、RS-422 以及 RS-485。它们最初都是由电子工业协会（EIA）制定并颁布的。RS-232C 作为一种工业标准发布于 1962 年，命名为 EIA-232-E，从而有效地保证不同厂家产品之间的兼容。

为改进 RS-232 通信距离短、速率低的缺点，RS-422 定义了一种平衡通信接口，将传输速率提高到 10Mbps，传输距离延长到 4000 英尺（速率低于 100kbps 时），并允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种一对多的单向平衡传输规范，被命名为 TIA/EIA-422-A 标准。

为扩展应用范围，EIA 又于 1983 年在 RS-422 基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，并增加了发送器的驱动能力和冲突保护特性，扩展了总线共模范围，后命名为 TIA/EIA-485-A 标准。由于 EIA 提出的建议标准都是以“RS”作为前缀，所以在信息工业领域，仍然习惯将上述标准以 RS 作前缀。其中“RS”表示推荐标准，232/422/485 为标识号。

1．RS-232C

目前，RS-232C 是 PC 与通信工业中应用最广泛的一种串行通信接口。RS-232C 采取不平衡传输方式，即所谓单端通信，并被定义成为一种在低速率串行通信中增加通信距离的单端标准。图 12-3 为常用 RS-232C 连线方式。

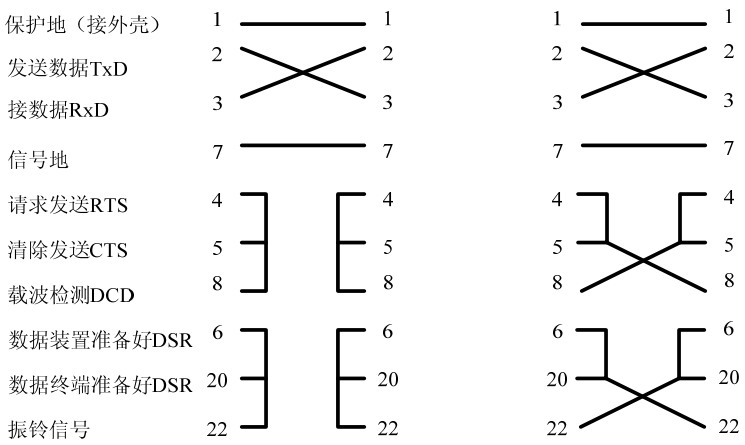


图 12-3　常用 RS-232C 连线方式

RS-232C 作为串行通信的行业标准，实现了 DCE（数据通信设备）与 DTE（数据终

端设备）之间的数据传输。表 12-1 中列出 RS-232C 标准接口各引脚的功能及特性。

表 12-1　RS-232C 接口引脚功能特性

DB9 引脚号	DB25 引脚号	信号名称	简称	方向	信号功能
—	1	保护地	—	—	接设备外壳，安全地线
3	2	发送数据	TXD	→DCE	DTE 发送串行数据
2	3	接受数据	RXD	DTE←	DTE 接收串行数据
7	4	请求发送	RTS	→DCE	DTE 请求切换到发送方式
8	5	清除发送	CTS	DTE←	DCE 已切换到准备接受
6	6	数传设备就绪	DSR	DTE←	DCE 准备就绪
5	7	信号地	—	—	信号地
1	8	载波检测	DCD	DTE←	DCE 已接受到远程信号
4	20	数据终端就绪	DTR	→DCE	DTE 准备就绪
9	22	振铃指示	RI	DTE←	通知 DTE，通信线路已妥

RS-232C 主要用于点对点（即只用一对收、发设备）的应用领域，其驱动器负载为 3~7kΩ，允许有 2500pF 的电容负载，相关的收、发端的数据信号是相对于信号地，典型的 RS-232 信号在正负电平之间摆动，在发送数据时，发送端驱动器输出正电平在+5~+15V，负电平在–5~–15V 电平。当无数据传输时，数据线为 TTL 电平，在传输数据时，从 TTL 电平跳转到 RS-232C 电平，通信结束后，再返回到 TTL 电平，RS-232C 接收器典型工作电平为+3~+12V 与–3~–12V。表 12-2 列出了 RS-232C 电气参数。

由于发送电平与接收电平的差仅为 2V 至 3V 左右，所以其共模抑制能力差，再加上双绞线上的分布电容，其传送距离最大约为 15 米，最高速率为 20kbps。数据传输速率可以在 50、75、100、150、300、600、1200、2400、4800、9600、19200bps 选择。当然，传输相应的通信距离随着通信速率的增大将受到一定的限制。

表 12-2　RS-232 电气参数

不带负载时驱动器输出电平	-25~+25V
负载电阻范围	3~7Ω
驱动器输出电阻	<300Ω
负载电容	<2500pF
逻辑“0”时驱动器输出电平	5~15V
逻辑“0”时负载接受电平	>+3V
逻辑“1”时驱动器输出电平	–5~–15V
逻辑“1”时负载接受电平	<–3V
输出短路电流	<500mA
驱动器转换速率	<30V/μs

2．RS-422 和 RS-485

RS-422 标准全称是“平衡电压数字接口电路的电气特性”，这个标准不仅改善了 RS-232C 的电气特性，而且也考虑了与 RS-232C 的兼容性。由于接收器采用高输入阻抗和发送驱动器，所以比 RS-232 更强的驱动能力。在 RS-422 通信中，允许在相同传输线上连接多个接收节点，最多可接 10 个节点，即一个主设备（Master），其余为从设备（Salve），

从设备之间不能通信，所以 RS-422 支持点对多的双向通信。RS-422 的最大传输距离为 4000 英尺（约 1219 米），最大传输速率为 10Mbps。其平衡双绞线的长度与传输速率成反比，在 100kbps 速率以下，才可能达到最大传输距离。只有在很短的距离下才能获得最高速率传输。一般 100 米长的双绞线上所能获得的最大传输速率仅为 1Mbps。

RS-485 是 RS-422A 的变型，RS-422 为全双工，可同时收发数据，RS-485 为半双工通信方式，所谓半双工通信是指在任意一个时刻，只能是一个发送一个接受，但是两者可以互换。RS-485 是一种多发送器的电路标准，其扩展了 RS-422 的性能，允许双线总线上一个发送器驱动 32 个负载设备，负载设备可以是被动发送器、接收器或者收发两用设备。

RS-422、RS-485 与 RS-232C 不一样，数据信号采用差分传输方式，也称做平衡传输，它使用一对双绞线，图 12-4 为平衡驱动差分接收电路。在通常情况下，发送驱动器之间的正电平在+2~+6V，是一个逻辑状态，负电平在–2~ –6V，是另一个逻辑状态。

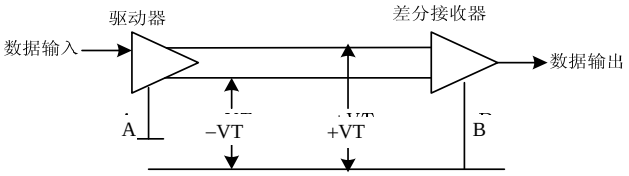


图 12-4　差分接收电路

RS-485 比 RS-232C 拥有更好的通信能力，更快的通信速率，更远的传输距离，表 12-3 为 RS-232C、RS-422 和 RS-485 三种通信标准的性能比较。

表 12-3　RS-232C、RS-422、RS-485 性能比较

规　　定	RS-232C	RS-422	RS-485
工作方式	单端	差分	差分
节点数	1 对 1	1 对 10	1 对 32
最大传输电缆长度（m）	15（24kbPs）	1200（100kbPs）	1200（100kPs）
最大传输速率	20kbPs	10MbPs	10MbPs
最大驱动输出电压	+/-25V	-0.25~+6V	-7~+12V
驱动器输出信号电平（空载）	+/-3V	+/-200mV	+/-1.5V
驱动器输出信号电平（负载）	+/-25V	+/-6V	+/-6V
驱动器负载阻抗	3~7kΩ	100Ω	54Ω
摆率（最大值）	30V/μs	N/A	N/A
接收器输入电压范围	+/-15V	-10~+10V	-7~+12V
接收器输入门限	+/-3V	+/-200mV	+/-200mV
接收器输入电阻（Ω）	3~7k	4k（最小）	≥12k
驱动器共模电压	—	-3~+3V	-1~+3V
接收器共模电压	—	-7~+7V	-7~+12V

12.2　相关开发技术——异步串行通信接口

12.2.1　异步串行通信标准

串行传输是二进制代码序列在一条信道上以位（码元）为单位、按时间顺序且按位传

输的通信方式。串行传输时，发送端按位发送，接收端按位接收，同时还要对所传输的位加以确认，所以收发双方要采取同步措施，否则接收端将不能正确区分出所传输的字符。串行通信按传输方式分为异步传输和同步传输。

- 异步传输方式是字符的异步传输技术；
- 同步传输方式即为位同步传输技术。

在异步传输模式下，传输数据以字符为单位，数据传输速率多在 1.2kbps 以下。当发送一个字符代码时，字符前面要加一个起始信号，其长度为一个码元，极性为“0”，即空号极性，字符后面要加一个终止符号，其长度为 1~2 个码元，极性为“1”，即传号极性。加上起始终止信号后，即可区分出所传输的字符，传送时，字符可以连续发送，也可以单独发送，不发字符时线路保持“1”状态，图 12-5 为起止式同步传输序列，每个字符由 8 位组成，加上起止位，信号共 11 位，两字符之间的间隔长度可以不固定。实现起来比较简单。

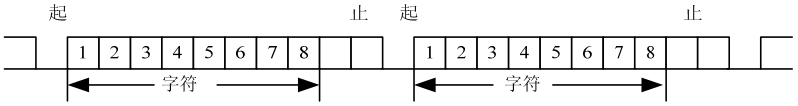


图 12-5　异步传输模式帧格式

异步串行通信协议规定字符数据的传输规范总结起来有以下几点。

（1）起始位

通信线上没有数据被传送时处于逻辑“1”状态，当发送设备要发送一个字符数据时，首先发送一个逻辑“0”信号，这个逻辑低电平就是起始位。起始位通过通信线传向接收机，接收设备检测到这个低电平后，就开始准备接收数据位信号。起始位所起的作用就是使设备同步，通信双方必须在传送数据位前一致同步。

（2）数据位

当接收设备收到起始位后，开始接收数据位。数据位的个数可以是 5~9 位，PC 机中经常采用 7~8 位数据传送。在字符传送过程中，数据位从最低有效位开始传送，依次在接收设备中被转换为并行数据。

（3）奇偶校验位

数据位发送完后，为了保证数据的可靠性传输，常传送奇偶校验位。奇偶校验用于有限差错检测。如果选择偶校验，则数据位和奇偶位的逻辑“1”的个数必须为偶数，相反，如果是奇校验，逻辑“1”的个数为奇数。

（4）停止位

在奇偶位或者数据位（当无奇偶校验时）之后发送停止位。停止位是一个字符数据的结束，可以是 1~2 位的低电平，接收设备收到停止位后，通信线路便恢复逻辑“1”状态，直到下一个字符数据的起始位到来。

（5）波特率设置

通信线路上传送的所有位信号都保持一致的信号持续时间，每一位的宽度都由数据的码元传送速率确定，而码元速率是单位时间内传送的码元的多少，即波特率。

在同步传输方式下，收发双方必须（要）建立准确的位定时信号，正确地区分每位数据，在该方式中，每个字符不增加任何附加位，而是连续发送，但在传输中，数据要分成

不同的组或者帧，一组含有多个字符代码或者多个独立码元，为使收发双方建立和保持同步，在每组的开始和结束需要加上规定的码元序列，作为标识序列。在发送数据之前必须先发送该标识序列，接收端通过检测出该标识序列来实现同步。由于同步传输方式适用于2.4kbps 以上的数据传输，不需要加起止信号，因而传输速率高，但实现较为复杂。

12.2.2 设置串口控制信号

在嵌入式 Linux 操作系统内核中提供了专用的驱动程序，在嵌入式开发串口驱动程序时，串口操作需要的头文件有：

```
#include <stdio.h>          /*标准输入输出定义*/
#include <stdlib.h>         /*标准函数库定义*/
#include <unistd.h>         /*linux 标准函数定义*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>          /*文件控制定义*/
#include <termios.h>        /*PPSIX 终端控制定义*/
#include <errno.h>          /*错误号定义*/
#include <pthread.h>        /*线程库定义*/
```

串口设备初始化设置包括波特率设置、校验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值。此结构体定义如下：

```
struct termio
    {
    unsigned short c_iflag;         /* 输入模式标志 */
    unsigned short c_oflag;         /* 输出模式标志 */
    unsigned short c_cflag;         /* 控制模式标志 */
    unsigned short c_lflag;         /* local mode flags */
    unsigned char c_line;           /* line discipline */
    unsigned char c_cc[NCC];        /* control characters */
    };
```

以下是修改当前串口波特率的代码。

```
struct termios Opt;
tcgetattr(fd, &Opt);
cfsetispeed(&Opt,B19200);   /*设置为 19200bps*/
cfsetospeed(&Opt,B19200);
tcsetattr(fd,TCANOW,&Opt);
```

以下是当前串口校验位和停止位的设置代码，其中设置串口为无校验 8 位代码如下。

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS8;
```

设置串口为奇校验(Odd) 7 位代码如下。

```
Option.c_cflag |= ~PARENB;
Option.c_cflag &= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

设置串口为偶校验(Even) 7 位代码如下。

```
Option.c_cflag &= ~PARENB;
Option.c_cflag |= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

设置串口为 Space 校验 7 位代码如下。

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= &~CSIZE;
Option.c_cflag |= CS8;
```

设置串口 1 位停止位代码如下。

```
options.c_cflag &= ~CSTOPB;
```

设置串口 2 位停止位代码如下。

```
options.c_cflag |= CSTOPB;
```

如果不是开发终端，只是串口传输数据，而不需要串口来处理，那么使用原始模式（Raw Mode）方式来通信，设置代码如下。

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); /*Input*/
options.c_oflag &= ~OPOST; /*Output*/
```

12.2.3　读入串口控制信号

Linux 操作系统将所有设备作为文件处理，在 Linux 下串口文件是位于/dev 下，串口 1 为/dev/ttyS0，串口 2 为/dev/ttyS1，打开串口是通过使用标准的文件打开函数操作，打开串口代码如下。

```
int fd;
/*以读写方式打开串口*/
fd = open( "/dev/ttyS0", O_RDWR);
if (-1 == fd)
    {
    perror(" 提示错误！");
    }
```

设置好串口属性并打开串口之后，读写串口很容易，同样把串口当做文件读写即可，其中发送数据操作代码如下。

```
char buffer[1024];
int Length＝1024;
int nByte;
nByte = write(fd, buffer ,Length)；
```

读取串口数据使用文件操作 read()函数读取，如果设置为原始模式（Raw Mode）传输数据，read 函数返回的字符数是实际串口收到的字符数。可以使用操作文件的函数来实现异步读取，如 fcntl，或者 select 等来操作。读取串口数据操作代码如下。

```
char buff[1024];
```

```
int Len＝1024;
int readByte = read(fd, buff, Len);
```

执行完成相关读写操作后需要执行关闭串口操作，其操作代码如下。

```
close(fd);
```

12.2.4 文件 Open()系统调用

Open()系统调用用于将一个文件名转换成一个文件描述符。当调用成功时，返回的文件描述符将是进程没有打开的最小数值的描述符。该调用创建一个新的打开文件，并不与任何其他进程共享。在执行 exec 函数时，该新的文件描述符将始终保持打开状态。文件的读写指针被设置在文件开始位置。参数 flag 是 O_RDONLY、O_WRONLY、O_RDWR 之一，分别代表文件只读打开、只写打开和读写打开方式，可以与其他一些标志一起使用。以下是 Open()函数源代码。

```
#define __LIBRARY__
#include <unistd.h>//Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数
//如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等
#include <stdarg.h>//标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个类型
// (va_list)和三个宏(va_start, va_arg 和 va_end)，用于 vsprintf、vprintf、
vfprintf 函数
//打开文件函数
//打开并有可能创建一个文件
//参数：filename - 文件名；flag - 文件打开标志
//返回：文件描述符，若出错则置出错码，并返回-1
int open(const char * filename, int flag, ...)
    {
    register int res;
    va_list arg;
    //利用 va_start()宏函数，取得 flag 后面参数的指针，然后调用系统中断 int 0x80 功
能 open
    //进行文件打开操作
    //%0-eax(返回的描述符或出错码)；%1-eax(系统中断调用功能号__NR_open)
    //%2-ebx(文件名 filename)；%3-ecx(打开文件标志 flag)
    //%4-edx(后随参数文件属性 mode)
    va_start(arg,flag);
    __asm__( "int $0x80"
    :"=a"(res)
    :""(__NR_open),"b"(filename),"c"(flag),
    "d"(va_arg(arg,int)));
    //系统中断调用返回值大于或等于 0，表示是一个文件描述符，则直接返回
    if (res>=0)
        return res;
        //否则说明返回值小于 0，则代表一个出错码。设置该出错码并返回-1
        errno = -res;
    return -1;
}
```

12.3 实例——基于 DS1820 的实时温度监控系统

Dallas 半导体公司的数字化温度传感器 DS1820 是世界上著名的“一线总线”接口的

温度传感器。这种“一线器件”具有体积更小、适用电压更宽、更经济的优势。一线总线独特而且经济的特点，使用户可轻松地组建传感器网络，为测量系统的构建引入全新概念。

DS1820 测量温度范围为-55~+125°C，在-10~+85°C 范围内，精度为±0.5°C。现场温度直接以“一线总线”的数字方式传输，大大提高了系统的抗干扰性。适合于恶劣环境的现场温度测量，例如环境控制、设备或过程控制、测温类消费电子产品等。用户可以选择更小的封装方式，更宽的电压适用范围，可以方便地实现分辨率设定，另外，用户设定的报警温度存储在 EEPROM 中，掉电后依然保存。

12.3.1　系统基本结构

1．DS1820 的内部结构

如图 12-6 所示，DS1820 内部结构主要由 4 部分组成：64 位光刻 ROM、温度传感器、非挥发的温度报警触发器 TH 和 TL、配置寄存器。其中，DQ 脚为数字信号输入/输出端；GND 脚为电源地；VDD 脚为外接供电电源输入端（在寄生电源接线方式时接地）。

光刻 ROM 的作用是使每一个 DS1820 代号各不相同，这样就可以实现一根总线上挂接多个 DS1820 的目的。光刻 ROM 中的 64 位序列号是出厂前被光刻好的，它可以看做是该 DS1820 的地址序列码。64 位光刻 ROM 的排列顺序是：开始 8 位（28H）是产品类型标号，接着的 48 位是该 DS1820 自身的序列号，最后 8 位是前面 56 位的循环冗余校验码（CRC=$X^8+X^5+X^4+1$）。

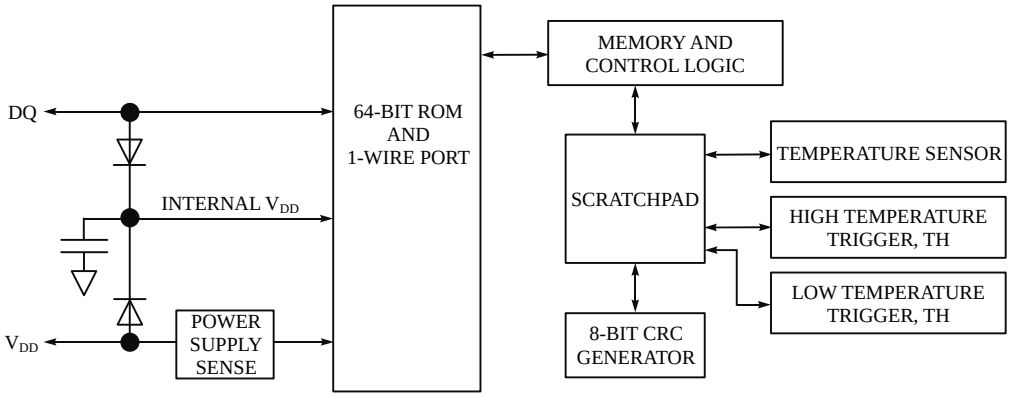


图 12-6　DS1820 内部结构

DS1820 中的温度传感器可以完成对温度的测量。以 12 位转化为例，用 16 位符号扩展的二进制补码读数形式提供数值，以 0.0625℃/LSB 形式表达，其中 S 为符号位。

如图 12-7 所示是 12 位转化后得到的 12 位数据，存储在 DS18B20 的两个 8 位的 RAM 中，二进制中的前面 5 位是符号位（S）。

如果测得的温度大于 0，这 5 位为 0，只要将测到的数值乘于 0.0625 即可得到实际温度；

如果温度小于 0，这 5 位为 1，测到的数值需要取反加 1 再乘于 0.0625 即可得到实际温度。

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
LS Byte	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
MS Byte	S	S	S	S	S	2^6	2^5	2^4

图 12-7　温度表示方法

不同温度值的数值表示如表 12-4 所示。

表 12-4　不同温度值的数值表示

温　　度	二进制数字输出	16 进制数字输出
+125℃	00000000 11111010	00FA
+25℃	00000000 00110010	0032h
+1/2 ℃	00000000 00000001	0001h
+0℃	00000000 00000000	0000h
–1/2 ℃	11111111 11111111	FFFFh
–25℃	11111111 11001110	FFCEh
–55℃	11111111 10010010	FF92h

2．DS1820 温度传感器的存储器

DS1820 温度传感器的内部存储器包括一个高速暂存 RAM 和一个非易失性的可电擦除的 E2RAM，E2RAM 用于存放高温度和低温度触发器 TH、TL 和结构寄存器，如图 12-8 所示，暂存存储器包含了 8 个连续字节。

- 前两个字节是测得的温度信息，第 1 个字节的内容是温度的低八位，第 2 个字节是温度的高八位；
- 第 3 个和第 4 个字节保留；
- 第 5 个字节是结构寄存器的易失性拷贝，3、4、5 这 3 个字节的内容在每一次上电复位时被刷新；
- 第 6、7、8 个字节用于内部计算；
- 第 9 个字节是冗余校验字节。

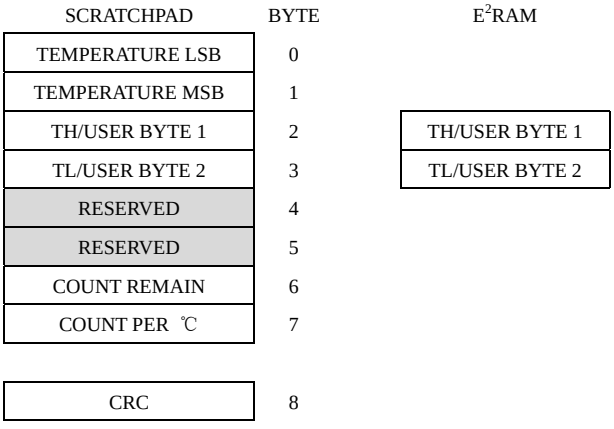


图 12-8　暂存存储器内容

3．一线协议

一线通信协议是一种通过一根信号线实现主机与一个或多个从机通信的协议，DS1820作为从机出现。以下从硬件配置、传输序列、一线通信信号 3 个方面阐述一线协议。

硬件配置：如图 12-9 所示为 DS1820 作为温度传感器时硬件配置图，一线协议只有一个信号线，也就不允许多个设备驱动，在适当的时候，所有连接在此上的设备必须具有一个三态输出，DS1820 的一线接口采用如图 12-9 所示的内部电路，因为一线通信线上有多个从机，故需要一个 5kΩ的上拉电阻。

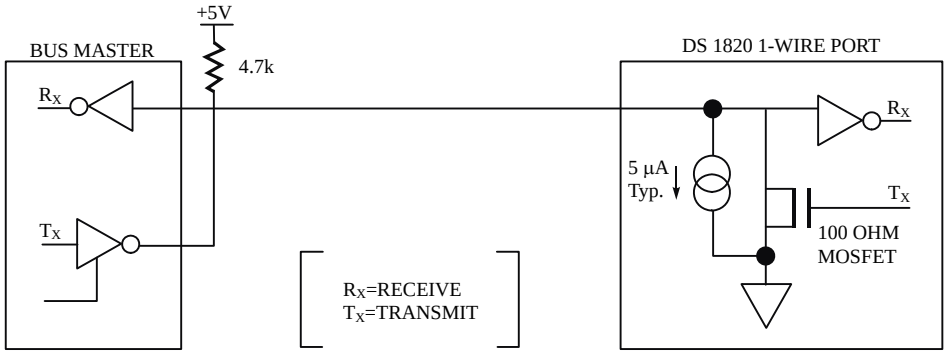


图 12-9　DS1820 硬件配置图

传输序列：通过一线协议访问 DS1820 的顺序如下。

（1）初始化：所有的从机设备都需要进行初始化，当主机发出一个复位信号，从设备发出准备好信号；

（2）ROM 功能配置命令：当初始化从设备后，主机即可发出 5 个 ROM 功能命令；

（3）存储器配置命令：完成 ROM 功能配置后，即对存储器进行配置。

（4）传输数据：完成以上操作后，即可通过一线协议传输数据。

4．系统硬件电路结构图

如图 12-10 所示是 DS1820 作为温度测量设备与嵌入式主机系统的硬件连接图，由图可知，DS1820 仅一根信号线与主机相连，嵌入式主机系统的输入输出信号通过上拉电阻和开路连接到此信号线上。

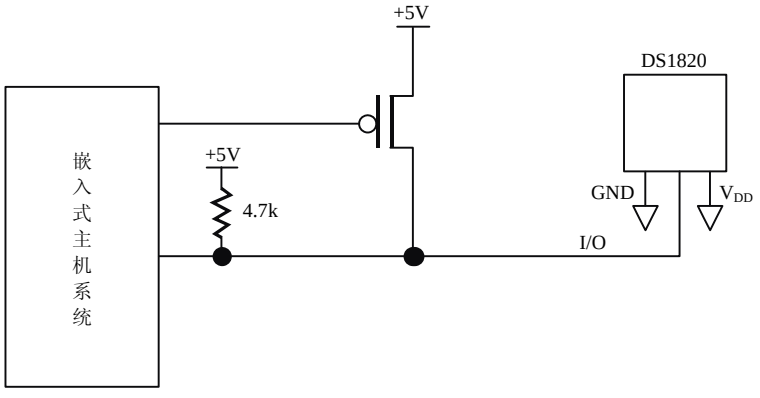


图 12-10　DS1820 与主机的连接图

DS1820 虽然具有测温系统简单、测温精度高、连接方便、占用口线少等优点，但在实际应用中也应注意以下几方面的问题。

（1）较小的硬件开销需要相对复杂的软件进行补偿，由于 DS1820 与微处理器间采用串行数据传送，因此，在对 DS1820 进行读写编程时，必须严格保证读写时序，否则将无法读取测温结果。在使用 PL/M、C 等高级语言进行系统程序设计时，对 DS1820 操作部分最好采用汇编语言实现。

（2）在 DS1820 的有关资料中均未提及单总线上所挂 DS1820 的数量问题，容易使人误认为可以挂任意多个 DS1820，在实际应用中并非如此。当单总线上所挂 DS1820 超过 8 个，就需要解决微处理器的总线驱动问题，这一点在进行多点测温系统设计时要加以注意。

（3）连接 DS1820 的总线电缆是有长度限制的。试验中，当采用普通信号电缆传输长度超过 50m 时，读取的测温数据将发生错误。当将总线电缆改为双绞线带屏蔽电缆时，正常通信距离可达 150m，当采用每米绞合次数更多的双绞线带屏蔽电缆时，正常通信距离进一步加长。这种情况主要是由总线分布电容使信号波形产生畸变造成的。由此可见，在用 DS1820 进行长距离测温系统设计时要充分考虑总线分布电容和阻抗匹配问题。

（4）在 DS1820 测温程序设计中，向 DS1820 发出温度转换命令后，程序总要等待 DS1820 的返回信号，一旦某个 DS1820 接触不好或断线，当程序读该 DS1820 时，将没有返回信号，程序进入死循环。这一点在进行 DS1820 硬件连接和软件设计时也要给予一定的重视。

（5）测温电缆线建议采用屏蔽 4 芯双绞线，其中一对线接地线与信号线，另一组接 VCC 和地线，屏蔽层在源端单点接地。

12.3.2 系统工作流程

1．系统调用

如图 12-11 所示为 DS1820 与嵌入式主机进行数据通信的工作流程图，主要包括以下几个主要步骤。

（1）初始化：主机对所有从机设备进行初始化，主机通过发送一个复位信号来初始化从机设备；

（2）ROM 功能配置 1：当初始化从设备后，主机读从机 ROM 的值；

（3）存储器配置 1：完成 ROM 功能配置后，即对存储器进行配置。

（4）传输数据：进行数据传输。

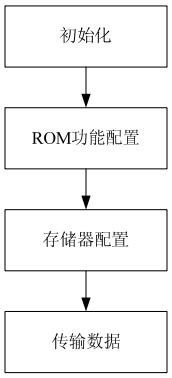


图 12-11 DS1820 与嵌入式主机通信流程

2．ROM 内容查找

ROM 处理程序主要重复读一个字节、读补充位、写读取的值 3 个简单的步骤。主机在每一个 ROM 位都执行以上 3 个步骤，当完成后，主机即知道设备 ROM 内容，其他设备的 ROM 内容按同样办法确认。如图 12-12 所示是通过一线协议查找 4 个从设备 ROM 内容的流程图。

其中 4 个从设备的 ROM 数据为：

ROM1 00110101...

ROM2 10101010...
ROM3 11110101...
ROM4 00010001...

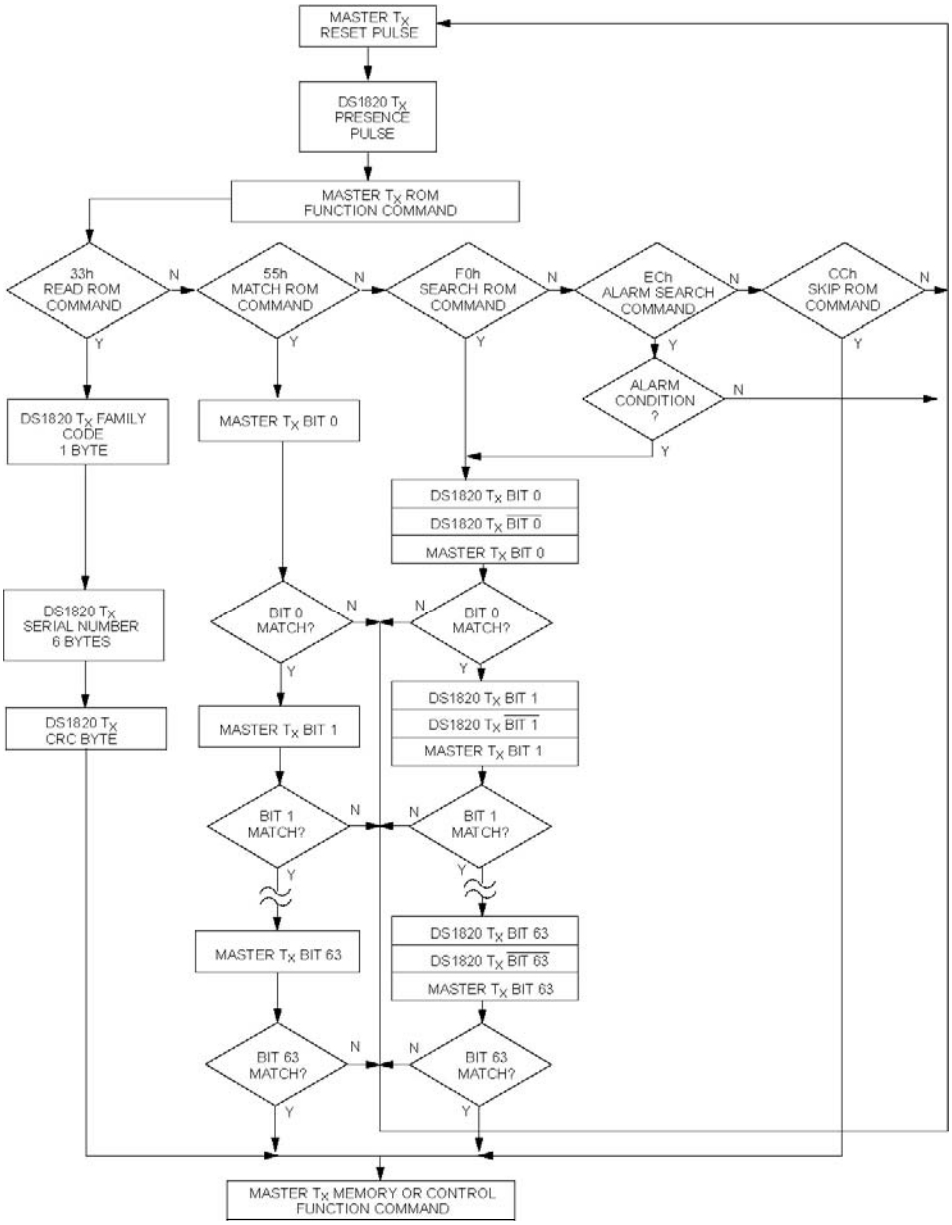


图 12-12　通过一线协议读从机 ROM 内容的流程图

执行步骤如下：

（1）主机通过一个复位信号初始初始化信号，从设备发出一个存在响应信号表示准备好；

（2）主机设备在一线总线上发出 ROM 查找命令；

（3）所有从设备都将响应自己 ROM 数据位的第一个字节在总线上，主机将读取这一

字节内容，其中，ROM1 和 ROM4 将响应 0，将线路拉低，ROM2 和 ROM3 将响应 1 在总线上，会将线路拉高，通过逻辑与可知结果为 0，故主机将读取数据 0。接着第二次读取数据，此时因为 ROM 数据命令已经被执行，故所有从设备将做第二次响应此字节内容（发出的是第一个位的反码），ROM1 和 ROM4 将响应 1，将线路拉低，ROM2 和 ROM3 将响应 0 在总线上，会将线路拉高，故结果为 0，此时主机可以确定当前有第一位为 0 和 1 两类设备。

- 如果结果为 00：总线上有第一位为 1 和 0 的两类设备；
- 如果结果为 01：所有的从设备第一位为 0；
- 如果结果为 10：所有的从设备第一位为 1；
- 如果结果为 11：没有发现设备；

（4）主机在总线上写 0 操作，此时 ROM2 和 ROM3 被反选，故只剩下 ROM1 和 ROM4；

（5）按第 3 步同样方法读取第 2 位，读取结果为 01，故可知存在第 2 位同时为 0 的从设备；

（6）主机在总线上写 0，同样选中 ROM1 和 ROM4；

（7）按第三步同样方法读取第 3 位，结果为 00，故可知存在第 3 位相冲突的从设备；

（8）总线写 1，反选 ROM1，从而选取 ROM4；

（9）从 ROM4 中读取剩下的位，从而读取了 ROM4 的值。

（10）完成 ROM4 读取后，总线重复以上步骤执行新的 ROM 探测序列，直到所有数据读取完毕为止。

12.3.3 系统模块源代码实现

1. 主机串口控制程序设计

在嵌入式 Linux 操作系统下，系统提供了专门的串口访问模块，用户只需要根据自身嵌入式硬件设备作适当裁剪即可。主要包括 Makefile 文件的编写、主机串口数据读取信号 Hostserial.c。

（1）Makefile 文件

```
KERNELDIR=/usr/src/linux-2.4.20-8
#CC = arm-linux-gcc
CC = gcc
CFLAGS = -I$(KERNELDIR)/include/ -Wall
host_serial:
    $(CC) $(CFLAGS) -o host_serial serial.c host_serial.c
clean:
    rm -f host_serial
```

（2）Hostserial.c

```
#include "serial.h"
#define TRUE  1
#define FALSE  0
int main()
    {
    int fd;
    char ch;
    char *dev ="/dev/ttyS0";//´®¿Ú1
```

```
    fd = init_serial(dev);
    if (fd>0)
        set_speed(fd,115200);
    else
        {
        printf("Can't Open Serial Port!\n");
        exit(0);
        }
    if (set_parity(fd,8,1,'n') == FALSE)
        {
        printf("Set Parity Error\n");
        exit(1);
        }
    printf("\nsuccess set\n");
    while(1)
        {
        read(fd,&ch,1);
        //ch=getchar();
        putchar(ch);
        //write(fd,&ch,1);
        }
    close(fd);
    return(0);
}
```

（3）erial.c

```
#include "serial.h"
int init_serial(char* dev)
    {
    int fd=open(dev,O_RDWR);
    if(fd<0)
        {
        printf("cannot open %s\n",dev);
        exit(-1);
        }
    else
        printf("%s open successfully.\n",dev);
    return fd;
    }
void set_speed(int fd,int speed)
    {
    int i,status;
    struct termios opt;
    tcgetattr(fd,&opt);
    for(i=0;i<sizeof(speed_arr)/sizeof(int);i++)
        {
        if(speed == name_arr[i])
            {
            tcflush(fd,TCIOFLUSH);
            cfsetispeed(&opt,speed_arr[i]);
            cfsetospeed(&opt,speed_arr[i]);
            status = tcsetattr(fd,TCSANOW,&opt);
            if(status != 0) perror("tcsetattr fd error!");
            break;
            }
        tcflush(fd,TCIOFLUSH);
```

```
        }
    }
int set_parity(int fd,int databits,int stopbits,int parity)    {
    struct termios opt;
    if(tcgetattr(fd,&opt) != 0)
        {
        perror("setup serial...");
        return 0;
        }
    opt.c_cflag &= ~CSIZE;
/*CSIZE  Character size mask.  Values are CS5, CS6, CS7, or CS8.*/
    switch(databits)
    {
    case 7:
        opt.c_cflag |= CS7;
        break;
    case 8:
        opt.c_cflag |= CS8;
        break;
    default:
        printf("unsupport data size.\n");
        return 0;
    }
    switch(parity)
        {
    case 'n':
    case 'N':
        opt.c_cflag &= ~PARENB;

        break;
    case 'o':
    case 'O':
        opt.c_cflag |= (PARENB|PARODD);
        opt.c_iflag |= INPCK;
        break;
    case 'e':
    case 'E':
        opt.c_cflag |= PARENB;
        opt.c_cflag &= ~PARODD;
        opt.c_iflag |= INPCK;
        break;
    case 's':
    case 'S':
        opt.c_cflag &= ~PARENB;
        opt.c_cflag &= ~CSTOPB;

        break;
    default:
        printf("unsupported parity.\n");
        return 0;
    }
        switch(stopbits)
        {
        case 1:
            opt.c_cflag &= ~CSTOPB;
            break;
        case 2:
```

```
                opt.c_cflag &= CSTOPB;
                break;
            default:
                printf("unsupported stopbits.\n");
                return 0;
            }
        if(parity !='n')    opt.c_iflag |= INPCK;
        tcflush(fd,TCIFLUSH);
        opt.c_cc[VTIME] = 150;
        opt.c_cc[VMIN] = 0;
            if(tcsetattr(fd,TCSANOW,&opt) != 0)
            {
            perror("\n");
            return 0;
            }
        return 1;
    }
```

2．从机串口控制程序设计

从机设备控制程序设计模块主要包括 Makefile 文件编写，从机串行数据传输控制程序 target_serial.c。

（1）Makefile

```
KERNELDIR=/home/sitsang/linux-2.4.20
#KERNELDIR = /usr
CC = arm-linux-gcc
CFLAGS = -I$(KERNELDIR)/include/ -Wall
serial:
    $(CC) $(CFLAGS)  -o target_serial serial.c target_serial.c
clean:
    rm -f target_serial
```

（2）target_serial.c

```
#include "serial.h"
#define TRUE   1
#define FALSE  0
int main()
    {
    int fd;
    char ch;
       char *dev ="/dev/tts/0";
    fd = init_serial(dev);
    if (fd>0)
        set_speed(fd,115200);
    else
        {
        printf("Can't Open Serial Port!\n");
        exit(0);
        }
    if (set_parity(fd,8,1,'N')== FALSE)
        {
        printf("Set Parity Error\n");
        exit(1);
        }
    printf("successfully set\n");
```

```
    while(1)
        {
        ch=getchar();
        write(fd, &ch, 1);
        //read(fd,&ch,1);
        //printf("%d:\t %d",i++,(int)ch);
        //putchar(ch);
        }
    close(fd);
    return(0);
}
```

3．DS1820 设备控制处理程序设计

DS1820 设备控制有一部分是用汇编语言编写的，主要包括主控制程序、DS1820 上电复位流程程序、读取温度子程序、读/写 DS1820 子程序、温度计算子程序、串口读取程序以及延时子程内容。

（1）主程序

```
MAIN:
LCALL INIT_1820     /*调用复位 DS1820 子程序*/
MAIN1:
LCALL GET_TEMPER    /*调用读温度子程序*/
LCALL FORMULA       /*通过公式计算,小数点后显示两位*/
LCALL BCD
LCALL DISPLAY       /*调用串口显示子程序*/
LCALL DELAY500      /*延时 0.5 秒*/
LCALL DELAY500      /*延时 0.5 秒*/
LCALL DELAY500      /*延时 0.5 秒*/
AJMP MAIN1
```

（2）DS1820 复位初始化程序。

```
INIT_1820:
SETB WDDATA
NOP
CLR WDDATA
/*主机发出延时 540 微秒的复位低脉冲*/
MOV R0,#36
LCALL DELAY
SETB WDDATA     /*然后拉高数据线*/
NOP
NOP
MOV R0,#36
TSR2:
JNB WDDATA,TSR3 /*等待 DS1820 回应*/
DJNZ R0,TSR2
LJMP TSR4       /*延时*/
TSR3:
SETB FLAG1      /*置标志位,表示 DS1820 存在*/
LJMP TSR5
TSR4:
CLR FLAG1       /*清标志位,表示 DS1820 不存在*/
LJMP TSR7
TSR5:
MOV R0,#06BH
```

```
TSR6:
DJNZ R0,TSR6    /*复位成功!时序要求延时一段时间*/
TSR7:
SETB WDDATA
RET
```

（3）读出转换后的温度值

```
GET_TEMPER:
SETB WDDATA          /*定时入口*/
LCALL INIT_1820      /*先复位 DS1820*/
JB FLAG1,TSS2
RET                  /*判断 DS1820 是否存在?若 DS1820 不存在则返回*/
TSS2:
MOV A,#0CCH          /*跳过 ROM 匹配*/
LCALL WRITE_1820
MOV A,#44H           /*发出温度转换命令*/
LCALL WRITE_1820
MOV R0,#50           /*等待 AD 转换结束,12 位的话 750 微秒*/
LCALL DELAY
LCALL INIT_1820      /*准备读温度前先复位*/
MOV A,#0CCH          /*跳过 ROM 匹配*/
LCALL WRITE_1820
MOV A,#0BEH          /*发出读温度命令*/
LCALL WRITE_1820
LCALL READ_18200     /*将读出的九个字节数据保存到 60H-68H*/
RET
```

（4）写 DS1820 的子程序

```
WRITE_1820:
MOV R2,#8            /*一共 8 位数据*/
CLR C
WR1:
CLR WDDATA
MOV R3,#6
DJNZ R3,$
RRC A
MOV WDDATA,C
MOV R3,#24
DJNZ R3,$
SETB WDDATA
NOP
DJNZ R2,WR1
SETB WDDATA
RET
```

（5）读 DS1820 的程序

```
/*从 DS1820 中读出九个字节的数据*/
READ_18200:
MOV R4,#9
MOV R1,#60H          /* 存入 60H 开始的 9 个单元*/
RE00:
MOV R2,#8
RE01:
CLR C
```

```
SETB WDDATA
NOP
NOP
CLR WDDATA
NOP
NOP
NOP
SETB WDDATA
MOV R3,#09
RE10:
DJNZ R3,RE10
MOV C,WDDATA
MOV R3,#23
RE20:
DJNZ R3,RE20
RRC A
DJNZ R2,RE01
MOV @R1,A
INC R1
DJNZ R4,RE00
RET
```

（6）温度计算子程序

```
FORMULA:/*按公式：T 实际=(T 整数-0.25)+( M 每度-M 剩余)/ M 每度*/
/*计算出实际温度，整数部分和小数部分分别存于 ZHENGSHU 单元和 DOT 单元*/
/*将 61H 中的低 4 位移入 60H 中的高 4 位，得到温度的整数部分，并存于 ZHENGSHU 单元*/
MOV 29H,61H
MOV A,60H
MOV C,48H
RRC A
MOV C,49H
RRC A
MOV C,4AH
RRC A
MOV C,4BH
RRC A
MOV ZHENGSHU,A
/*( M 每度-M 剩余)/ M 每度,小数值存于 A 中*/
MOV A,67h
SUBB A,66h
MOV B,#64H
MUL AB
MOV R4,B
MOV R5,A
MOV R7,67H
LCALL DIV457
MOV A,R3

/*再减去 0.25，实际应用中减去 25*/
SUBB A,#19H
MOV DOT,A            /*小数部分存于 DOT 中*/
MOV A,ZHENGSHU
SUBB A,#00H          /*整数部分减去来自小数部分的借位*/
MOV ZHENGSHU,A
MOV C,4BH
```

```
JNC ZHENG            /*是否为负数*/
CPL A
INC A
MOV DIS_1,#2DH       /*零度以下时,第一位显示"-"号*/
MOV ZHENGSHU,A
ZHENG:
MOV DIS_1,#2BH       /*零度以上时,第一位显示"+"号*/
RET
```

（7）串口显示数据子程序

```
DISPLAY:
CLR TI
MOV A,DIS_1
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示+/-*/
CLR TI
MOV A,DIS_2
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示整数第一位*/
CLR TI
MOV A,DIS_3
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示整数第二位*/
CLR TI
MOV A,#2EH
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示小数点*/
CLR TI
MOV A,DIS_4
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示小数第一位*/
CLR TI
MOV A,DIS_5
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示小数第一位*/
CLR TI
MOV A,#0DH;换行
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示*/
CLR TI
MOV A,#0AH           /*换行*/
MOV SBUF,A
JNB TI,$             /*发送给 PC,通过串口调试助手显示*/
RET
```

（8）延时子程序

```
/*为保证 DS1820 的严格 I/O 时序，需要做较精确的延时*/
/*在 DS1820 操作中，用到的延时有 15μs；90μs，270μs，540μs*/
/*因这些延时均为 15μs 的整数倍，因此可编写一个 DELAY15（n）函数*/
DELAY:/*;11.05962M 晶振*/
LOOP: MOV R1,#06H
LOOP1: DJNZ R1,LOOP1
DJNZ R0,LOOP
RET
```

```
/*500 毫秒延时子程序，占用 R4、R5*/
DELAY500:MOV R4,#248
DA222:MOV R5,#248
DJNZ R5,$
DJNZ R4,DA222
RET
END
```

12.4 本章总结

随着计算机应用的发展和深入，工业温度监控设备的开发日益普遍。本章详细介绍了 Linux 工业温度监控设备的开发实例。首先简单介绍了应用环境与硬件设计，然后介绍了相关开发技术——异步串行通信接口，为后续实例学习提供技术基础。最后讲解了一个基于 DS1820 的实时温度监控系统的开发，从系统基本结构、工作流程以及模块源代码实现 3 个方面进行介绍。读者学习时，建议在模块源代码的理解上多下工夫，因为掌握了模块编程的技术要领和细节，对改善程序设计方法和提高设计效率大有帮助。

第 13 章

实时视频采集系统开发实例

随着信息技术的飞速发展，信息采集不再停留在文字类型上，实时的、高品质的图像、视频信息是许多决策者和科技人员获得动感和感性认识的源泉。视频采集在这方面发挥了很大的作用，越来越受到人们的重视。

在众多的视频采集系统中，嵌入式的视频采集以其小巧、灵活、低成本、高性能的特点而独具优势。结合嵌入式 Linux 支持 TCP/IP 的特性，可以更好地利用发达的网络技术，通过建立 Client/Server（用户/服务器）工作模型来实现远程视频监控。嵌入式技术必将在信息采集应用领域中发挥越来越重要的作用。

如今，网络技术已经发展得非常成熟，通过网络实现远程监控是视频采集技术的一个发展趋势。使用嵌入式系统实现视频图像采集，然后通过网络传输图像数据更是其中的热点。如图 13-1 所示是基于 TCP/IP 协议的嵌入式视频采集系统框图。系统将设备采集到的数据通过网络传送到视频服务器或视频监控中心的数据库中，从而实现嵌入式系统利用 Internet 技术实现低成本网络互联、信息沟通。

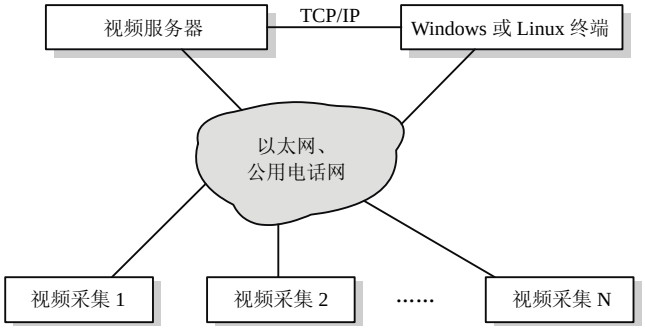


图 13-1　基于 TCP/IP 协议的嵌入式视频采集系统

在运行嵌入式 Linux 操作系统上，结合发达的网络技术，实现嵌入式视频采集的功能，非常方便、灵活。另外，为克服嵌入式系统资源紧张，处理器运算能力和资源不及 PC 的缺点，可以将图像数据进行压缩处理，如使用 H263+、MPEG2/4 等压缩算法，可以减少数据传送量，提高画面质量。

13.1　应用环境与硬件设计概要

视频采集的硬件系统如图 13-2 所示，系统主要由嵌入式处理板和控制板两块板组成，

加上用于网络连接的 Modem 或 RTL8019 网卡；视频处理中心是任意普通的视频终端计算机；中间的传输网络为公用电话网（PSTN：Public Switched Telephone Network）或 IP 网络。

系统以视频处理器 CPU 为核心，以视频采集芯片、可编程逻辑器件、存储器等为外围的嵌入式系统上，实现视频/图像的采集、压缩，并将压缩后的数据通过网络发送到接收端（视频处理中心），在接收端能够使用在 PC 上的程序将图像解压缩并显示出来。在系统中加入 TCP/IP 服务端应用程序，建立视频采集系统，使系统成为视频采集服务器，用户从远程的 PC 通过以太网和服务器建立连接，获取实时图像。在一般情况下，PC 和嵌入式系统只需建立 UDP 连接，系统把采集到的图像数据简单编号后，按顺序发送出去；在 PC 端按编号增加的顺序显示还原图像，并丢弃序号错误的数据，就可以得到比较理想的实时画面。系统还支持多个用户同时访问。

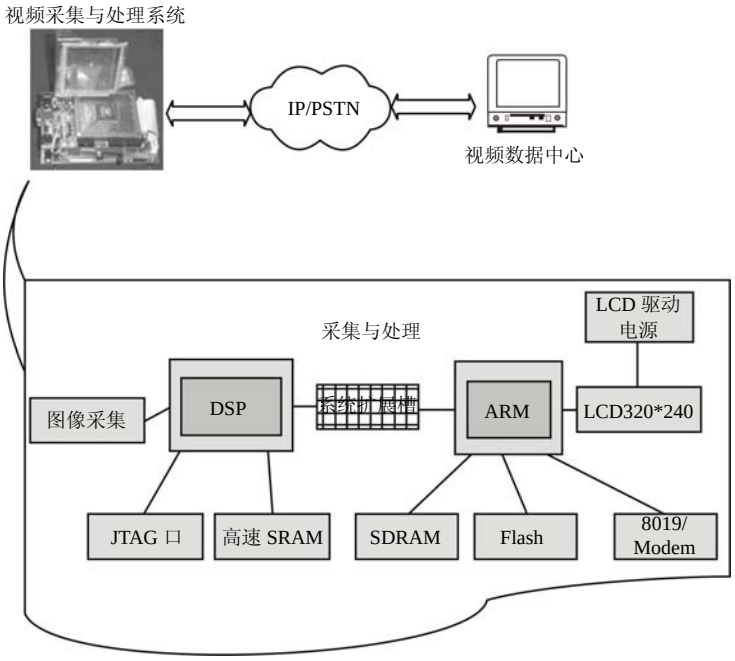


图 13-2　基于嵌入式 Linux 的工业控制系统

CMOS 传感器采集图像数据以 DMA 方式传送到由 DSP 和 ARM 为核心构成的嵌入式系统，每帧图像 320×240B（大小可设），可以获取 15 帧/s 图像信息。系统对接收到的图像数据进行处理，以服务器的方式通过以太网或 PSTN 传送给与它建立连接的远程 PC。PC 接收图像数据，可以显示或做进一步处理。

13.2　相关开发技术

13.2.1　视频图像压缩技术

数字化视频采集与处理系统是嵌入式系统的典型应用，涉及嵌入式系统技术和视频图像信息的编码技术。特别是在视频图像信息的编码技术方面，为了提高系统的视频图像质量与通用性，国际通信联盟（ITU）为视频图像信息的压缩编码制定了很多标准，新的编

码技术被迅速应用到图像与视频压缩上。

如表 13-1 所示是常用的用于图像与视频压缩的标准。其中 JPEG 是目前最常用的图像压缩格式之一，广泛应用于图像信息的传递与存储领域；H.263 和 H.264 是目前常用的视频压缩标准，可以在窄带信道上传递流畅的视频影像。应用 JPEG 格式可以进行图像压缩，如果想要进一步提高系统性能，可以考虑使用 H.263 来进行视频压缩。

表 13-1　常用的图像与视频压缩标准

标准名称	应用领域	压 缩 比
JPEG（T.81）	连续色调图像压缩	50~100:1
MPEG	VCD	30~50:1
MPEG2	数字电视、DVD	50~200:1
MPEG4	网络视频传输	100~300:1
H.261	视频压缩 ISDN	30~100:1
H.263	视频压缩 PSTN	100~300:1
H.264	视频压缩 Mobile Phone	>200:1

如图 13-3 所示是一个实时视频采集与播放系统，它可以实现双向视频和语音信息交换。



图 13-3　实时视频采集与播放系统

采集的数字视频信号的数据量很大，标准的 PAL 信号的速率为 216Mbps，如果不压缩基本上不可能传输，即使压缩到 2Mbps 时也只能在通信干线上传输，不能扩展到终端用户。因此需要提高压缩效率，一方面通过图像质量的降级实现，但主要还是提高编码效率。例如，用于较高速率视频编码的 MPEG1（VCD、1.5 Mbps）、MPEG2（DVD/广播级、4~10 Mbps），一部 100 min 的 DVD-5 的视频素材也要约 5 GB 的存储空间，目前的网络现状难以接受。

另一方面，视频数据的运算量也非常大，比如，仅图像运算中最基本的运算离散余弦转换（DCT），对于单通道 30 帧的 VGA 视频，它就需要每秒大约 6 千万次的乘加运算，这还仅是其中非常小的一部分。

由此可见，在视频处理中，巨大的数据量、运算量都需要进行低比特率视频处理（如图 13-4 所示）。专用媒体处理芯片的应用与通用的 DSP 相比，用于多媒体应用的专用芯片集成了许多专用模块，这些模块用硬件加速，可以处理大量多媒体方面的算法。

图 13-4 中的视频信息可分为形状信息（shape）、运动信息（motion）、纹理信息（texture）。对视频对象的编码就是对这 3 种信息进行编码。通过运动预测和运行补偿来去除连续帧之间的时间冗余；通过 DCT 可以有效压缩特征系数；量化一般包括量化、Z 字扫描和行程编

码，可以进一步提高压缩率；熵编码主要有 Huffman 编码和算术编码。

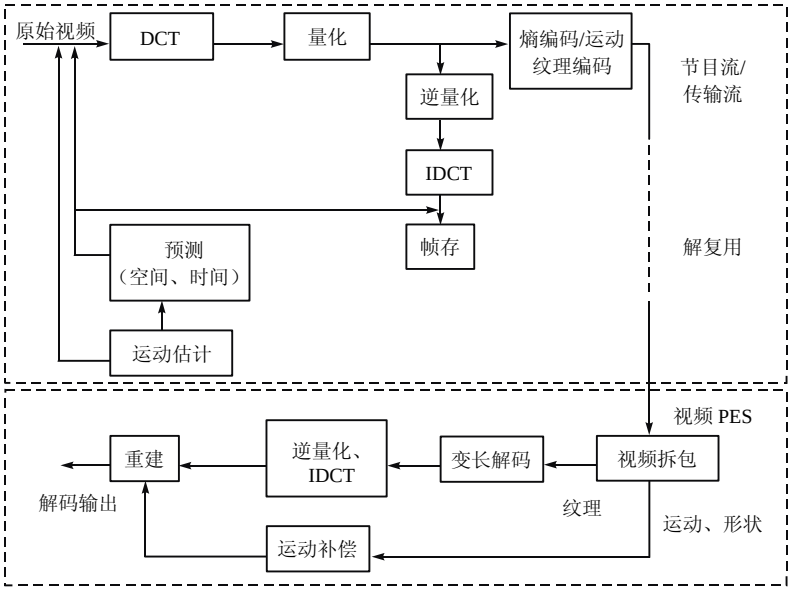


图 13-4　低比特率视频编解码框图

对于极低速率的视频编码，主要是 ITU-T H.263+/H.264 和 ISO 的 MPEG-4 标准。它们是针对视频会议、可视电话的甚低速率编码标准，基于内容的检索与编码，可对压缩数据内容直接访问。既可媲美 DVD 的视频质量，又具有 VCD 的压缩效率，从而在较小网络带宽的前提下获得更为满意的视频效果。例如，MPEG-4 可得到 25~30 帧/s（352×176）的无闪烁视频，而且能够同时将视频数据压缩到 300Kbps，使单路服务器在采用 100 Mbps 网卡的情况下可以支持 100 多个并发视频流。

通常采用专用芯片实现的视频会议系统和 IPTV 移动多媒体系统，可以应用于 IPTV-WebMedia、移动流媒体（Web Live/Mobile TV），为用户提供实时电视、数字视频广播（硬盘播出）、视频点播和视频通话等服务。当应用于网络视频会议时，速度能达到 2Mbps 以及 30 帧/s。

13.2.2　视频采集驱动

Linux 的驱动程序是视频采集系统的核心，这里采用模块方式加载，维护简易。绝大多数设备以文件的方式出现在系统中，供用户访问，抽象程度低，操作方便。

1．结构体、数据结构、函数定义模块

struct file_operations 结构是 Linux 中硬件数据与用户访问的交互枢纽，此结构体定义如下：

```
struct file_operations
{
  struct module *owner ;
  loff_t (*ll seek) ( struct file *, loff_t , int) ;
  ssize_t (*read) ( struct file *, char _user *, size_t , loff_t *) ;
  ssize_t (*write) ( struct file *, const char _user *, size_t , loff_t *) ;
```

```
    unsigned int (*poll) ( struct file *, struct poll_table_struct *) ;
    int (*ioctl) ( struct inode *, struct file *, unsigned int , unsigned long) ;
    int (*mmap) ( struct file *, struct vm_area_struct *) ;
    int (*open) ( struct inode *, struct file *) ;
    int (*flush) ( struct file *) ;
    int (*release) ( struct inode *, struct file *) ;
    int (*f sync) ( struct file *, struct dentry *, int datasync) ;
    ssize_t (*readv) ( struct file *, const struct iovec *, unsigned long ,
loff_t *) ;
    ssize_t (*writev) ( struct file *, const struct iovec *, unsigned long ,
loff_t *) ;
  } ;
```

当在应用程序中使用 open、read、write、close 等文件操作时，系统就调用这里面对应的成员函数。因此在这个驱动中只要实现 open、read、ioctl、release 等函数即可。另外，还需要自定义一个数据结构描述采集使用资源的信息。

open 函数的实现：

```
static int open( struct inode * inode , struct file * file)
{
  file-> private_data = &a ;    /* a 为一个描述全局数据*/
  MOD_INC_USE_COUN T ;          /* 打开计数 */
  return 0 ;
}
```

release 函数的实现：

```
static int release ( struct inode inode , struct file file)
{
                                /* 文件 close 时调用此函数 */
  MOD_DEC_USE_COUN T ;          /* 关闭计数 */
  return 0 ;
}
```

read 函数的实现：

```
static ssize_t read ( struct file * file , char * data , size_t count , loff_t
* ppos)
{
                /* 最主要的函数,负责从 DAM 内存中将数据复制到用户数据缓冲区*/
  char * pt r ;
  struct *a = ( struct a * ) file - > private_data ;
  interruptible_sleep_on ( &saa - > read_queue) ; /* 无数据则等待 */
  pt r = a- > dma_ addr ;
  _copy_to_user (data , pt r , RECORD_L EN GTH) ; /* 数据复制 */
  return RECORD_L EN GTH ;
}
```

ioctl 函数主要用来根据需要对一些硬件进行设置，视具体情况而定，在此不再列举，对功能的实现没有太大的影响。

数据结构定义如下：

```
struct a
{
  struct a_dev dev ;              /* 记录对应的数据结构*/
```

```
  unsigned long a_adr ;              /* PCI 板卡分配的资源起始地址 */
  unsigned char a_mem ;              /* 资源映射到内存空间的地址 */
  unsiagned char * dma_addr          /* DMA 的物理内存地址 */
  wait_queue_head_t read_queue ;     /* 对数据队列 */
  unsigned long board_type ;         /* 描述板卡的版本号,为兼容升级做准备 */
  unsigned long interrupt_enable_mask ; /* 中断掩码*/
  unsigned long dma_addr ;              /* DMA 数据传输的地址 */
} ;
```

2．驱动程序初始化模块

每个驱动都有一个唯一的指定入口 module_init 和一个唯一的指定出口 module_cleanup。驱动模块初始如下：

```
int module_init (void)
{
  struct a_dev * dev = NULL ;
  int result ;
  dev = a_find_ device (VENDOR_ ID , DEVICE_ ID, dev) ;   /* 查找设备*/
  if (dev)
  {
    a> a_adr = resource_start (dev ,0) ; /* 获取资源 */
    a> a_mem = ioremap ( a - > a_adr , resource_end (dev ,0),resource_start
(dev ,0) + 1) ;                 /* 进行内存空间映射*/
    request_irq (dev - > irq , a_irq ,A_SHIRQ | A_IN TERRUPT ,"pic",(void )
a) ;                            /* 申请中断号 */
    a > dma_addr = (char *) alloc_consistent (dev , DMA_BUF_LENGTH , NULL) ;
                    /* 申请 DMA 内存 */
    result = register_chrdev (0 , "pic" , &dvr_fop s) ; }
    if (result)
  {
     init_a () ;    /* 初始化采集 */
     init_i2c () ;  /* 初始化 I2C */
        ……
  }
    return 0 ;
}
```

3．中断函数

中断函数是驱动程序的核心部分，初始化时在成功获取中断号后，每次中断来临就调用该函数，主要是读取中断状态，判断该中断是何种中断，然后执行相应的处理。

```
static void a_irq (int irq , void * dev_id , struct pt_regs * regs)
{
  unsigned long t ransfer_lengt h ;
  struct a * a = ( struct * a ) dev_id ;
  volatile unsigned long isr = 0 ;
  isr = aread ( ISR) ;
  if (isr = = 0)
  {      /* 错误的中断 */
    return ;
  }
  if ( (isr & 0x40) )
  {     /* 从板卡到 DMA 空间的传输数据 */
    disable_GPIO3_int (a) ;
```

```
    enable_debi_int (a) ;
    transfer_lengt h = RECORD_L EN GTH ;
    awrite ( a2 > dma_addr , DEBI_AD) ;     /* 启动 DMA 传输 */
    awrite ( ( transfer_lengt h < < 17) | 0x10010 , DEBI_COMMAND) ;
    awrite (0x11dA0000 , DEBI_CONFIG) ; // 0x7c
    awrite (0x20002 , MC2) ; }
  else if ( (isr & 0x80000) = = 0x80000) {     /* DMA 完成通知数据可取 */
    wake_up_interruptible ( &a2 > read_queue) ; /* 若有数据,唤醒读等待队列 */
    disable_debi_int ( a) ;
    enable_GPIO3_int ( a) ;
  }
  awrite (isr , ISR) ;
  return ;
}
```

以上大部分函数都进行了简化处理，只保留了核心部分，但并不影响采集板卡的正常工作。

13.2.3　视频驱动加载运行

建立设备驱动程序模块后，通过以下方式运行系统。

将 CMOS 设置为 320×240 工作模式，在系统的 LCD 上显示获取的图像。以最快的速度获取图像数据，并在 LCD 上显示。系统可以获取 24 位色的原始图像数据约 15 帧/s，在 LCD 上可以清楚显示图像，无停顿感。

13.3　实例——基于 MV86S02 实时视频采集系统设计

MV86S02 是 Fujitsu（富士通）公司生产的一款 375×293（10 万）像素（CIF 格式）的 CMOS 图像传感器，它的片内集成了色彩信号处理器。这种 CMOS 图像传感器与色彩信号处理器集成到一个芯片内部的技术降低了功耗，而且减小了体积。MV86S02 非常适合应用于移动电话和掌上电脑。

13.3.1　系统基本结构

1．MB86S02 的功能框图

视频源信号来自于一个高度集成的 CMOS 数字图像传感器模块 MB86S02，它是富士通的产品。它不但集成了 CMOS 图像传感器阵列、自动增益信号放大器、模数转换器，还包括色彩信号处理和微型镜头，包含了图像采集的所有前端处理，可以直接输出数字信号。模块的系统框图如图 13-5 所示。

系统运行于嵌入式 Linux 操作系统，使系统的性能得以保证。使用具有 8 位并行数据接口的 CMOS 图像传感器，通过设备驱动程序与 Linux 操作系统结合在一起，实现高效率的图像采集。

MB86S02 是基于 CMOS 工艺，使用有源像素的传感器，与传统的 CCD 传感器相比有很多不同点。CMOS 技术的最大优点是每一个像素单元可以集成一个或多个晶体管，这样就具有了低功耗和小型化的优点，非常适用于手持设备，可以降低系统功耗、体积，提高电池效率；它的高度集成性大大简化了图像应用系统的设计。

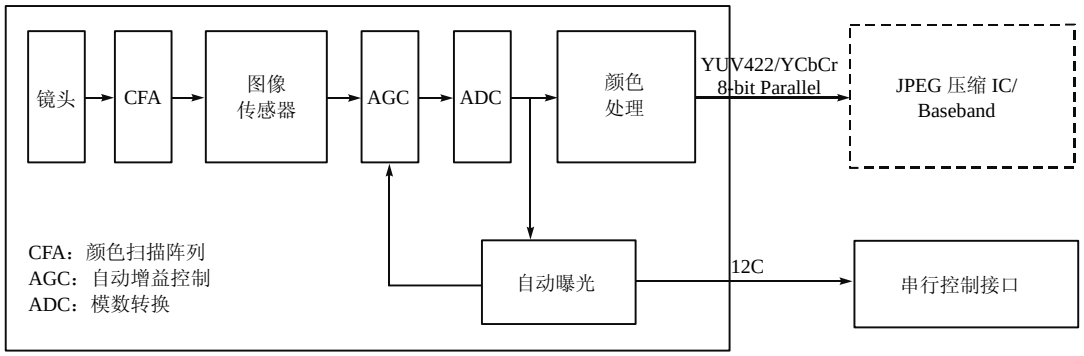


图 13-5　MB86S02 的功能框图

如图 13-6 所示是 MB86S02 的实物图。



图 13-6　MB86S02 的实物图

2．MB86S02 功能特性

MB86S02 能够直接输出 YCbCr422/YUV422 格式的数字图像的特性，大大方便了用户的使用。传统的图像采集系统都是使用模拟图像采集器，然后经过复杂的模数转换，不仅造价高，而且也大大增加了产品的体积。使用 MB86S02 可以直接得到图像的数字信号，简化了开发的工作，又提高了系统的性能。

MB86S02 的主要特性如下：

- 1/7 英寸图像传感器，有效像素为 352×288 共 11 万像素；
- 超低功耗 30mW@15fps；
- 输出 8 位 CMOS 电平并行数字信号，YCbCr422 或 YUV422 格式；
- 色彩信号处理包括自动增益、自动曝光、自动白平衡、Gamma 校正等；
- 寄存器设置通过标准 I^2C 串行接口；
- 支持 CIF（352×288）QCIF（176×144）格式；
- CCIR656 标准头输出；
- 抗闪烁功能；
- 低功耗模式；
- 掉电模式功耗 3μW。

MB86S02 不仅体积小，功耗低，而且接口也很简单，连接线的形状可以根据用户的需要来定制，使用起来十分方便，它有 21 个管脚定义如表 13-2 所示。

表 13-2　MB86S02 的引脚定义

引脚名称	I/O 方式	引脚编号	描　　述
OSCIN	I	1	时钟输入（9MHz）
XRESET	I	2	复位信号，低有效
PDWN	I	3	调电信号，高有效
D0~D7	O	4~11	数字视频输出
PCLK	O	12	同步时钟
DVss	—	13	数字地
DVdd	—	14	数字电源（2.8V）
SCL	I/O	15	I^2C 时钟信号
SDA	I/O	16	I^2C 数据信号
AVF	O	17	有效帧信号
AVH	O	18	行同步信号
VD	O	19	场同步信号
AVss	-	20	模拟地
AVdd	-	21	模拟电源

由于 MB86S02 采集的数字信号格式为 YCbCr 或 YUV，而 VGA 显示器需要的是 RGB 分量信号，所以如果想把 MB86S02 采集的图像直接显示在 VGA 显示器上需要进行 YCbCr 或 YUV 到 RGB 的颜色空间转换。

YCbCr 和 YUV 是基于亮度与色差的颜色空间，RGB 则是基于红绿蓝三基色的颜色空间。它们之间转换的理论公式见公式 13-1：

$$\begin{aligned} R &= Y + 1.371 \times V \\ G &= Y - 0.698 \times U - 0.336 \times V \\ B &= Y + 1.732 \times U \end{aligned} \qquad \text{（公式 13-1）}$$

当实现 8 位字长的 YUV 到 RGB 转换时，为了实现高速转换和宽度扩展简化了理论公式，简化后的公式见公式 13-2：

$$\begin{aligned} R' &= Y + V - 128 \\ B' &= Y + U - 128 \\ G' &= Y - 0.194 \times U - 0.5 \times V + 90 \end{aligned} \qquad \text{（公式 13-2）}$$

其中的 0.194 倍的 U 分量用查表的方法得到，0.5 倍 V 则直接用右移一位实现，经过这样的简化后，整个转换可以在两个时钟完成，经过测试转换效果比较理想。

3．系统硬件电路结构图

如图 13-7 所示是 MB86S02 的硬件接口模块图，左边端口是与 MB86S02 的接口；右边是内部信号。该模块的功能是把 MB86S02 输出的一帧图像信号整理为连续同步数据流，并在简化之后进行存储操作。

嵌入式处理器通过与 MB86S02 CMOS 图像模块的连接，读取图像数据后通过同步 SRAM 接口存储在外部高速 SRAM 中，然后还可以由 UART 模块或 RTL8019 模块把已经存储的图像数据发送到 PC，最后 PC 上的接收程序将显示接收的图像。通过调试板上的按

键可以选择采集图像、地址复位、串口发送、网口发送这几种功能。

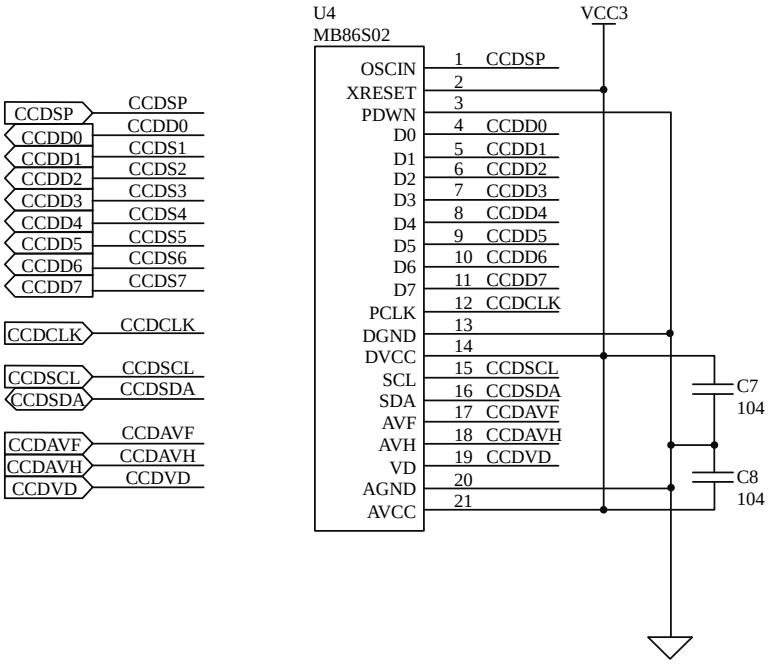


图 13-7　MB86S02 硬件连接图

13.3.2　系统工作流程

1．视频采集工作流程

视频采集程序包括类驱动和微驱动两个模块，视频采集程序的结构框架如图 13-8 所示。类驱动使用 GPIO（General Purpose IO Interface，通用 I/O 接口）模块，GPIO 模块的传输模式是基于流输入输出模块的同步 I/O 模式，更适合文件系统 I/O，如视频采集的应用。该模块的主要 API 函数的描述如表 13-3 所示。

表 13-3　GPIO 模块的主要 API 函数

函　　数	函数描述
GPIO_control	设备控制操作
GPIO_create	创建 GPIO 通道
GPIO_delete	取消 GPIO 通道
GPIO_submit	向微驱动发送数据包

在图 13-8 中，应用程序使用 GPIO_create 函数创建 GPIO 通道，并通过调用 GPIO_submit 函数直接与微驱动交换数据，完成视频数据的采集。

应用程序通过 GPIO 类驱动调用微驱动的标准 API 函数，这些标准 API 函数的描述如表 13-4 所示。这些规定的函数将放入微驱动的函数接口表中，以供应用程序通过 GPIO 类驱动调用。

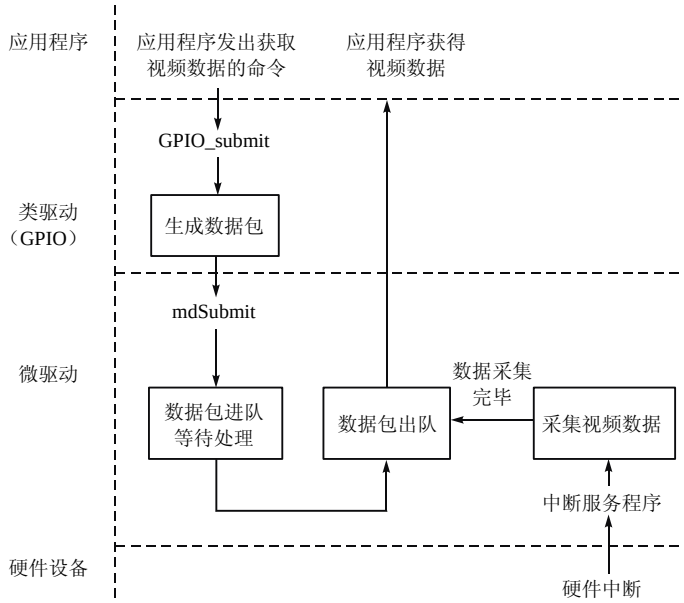


图 13-8　视频采集程序流程

表 13-4　GPIO 模块的主要 API 函数

函　　数	函数描述
mdBindDec	绑定通道
mdCreate/mdDelete	创建/删除通道
mdSubmitI/O	请求发送
ISRs	中断服务
mdControl	硬件设备控制

2．采集数据压缩流程

（1）JPEG 压缩

JPEG 压缩的基本系统是基于 DCT 和 VLC 的编码系统，首先将彩色图像转换到 YUV 颜色空间，YUV 每一个分量对应一张灰度图，这里使用 YUV 4:1:1 编码方式，即四个亮度 Y 对应一对色度 U、V（针对人眼对亮度比对色度更敏感的生理特性，而采取的减少数据量的方法），这样使得原始图像的数据就减少了一半。如图 13-9 所示是 JPEG 压缩的流程图。

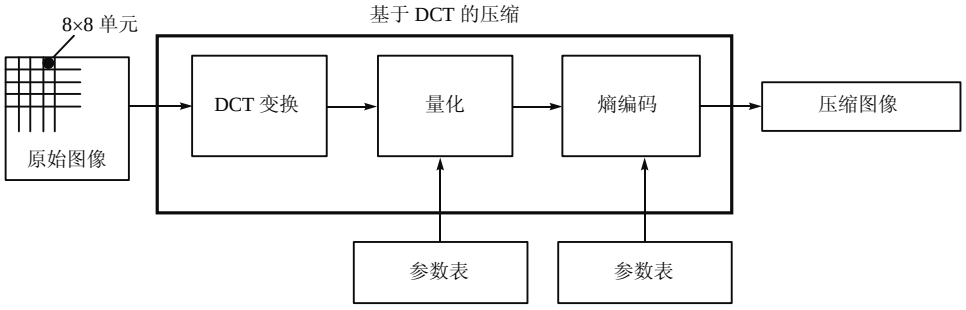


图 13-9　JPEG 压缩流程

如图 13-9 所示 JPEG 压缩的过程如下：

- 基于 DCT 的压缩以 8×8 个像素单元为最小处理单位，进行二维 8×8 的 DCT 变换；
- 然后对变换后 DCT 系数进行量化（分别除以亮度或色度量化系数矩阵）；
- 对量化后系数中的直流分量（第一个系数）进行差分编码；
- 其他交流分量（其余 63 个系数）经过“Z”形变换；
- 对这 63 个系数再进行 Huffman 编码，生成压缩码流。

如图 13-10 所示是 YUV 各分量处理顺序（图中每一个方块代表一个 8×8 的最小单元）。

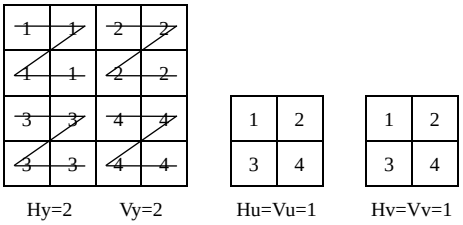


图 13-10　YUV 各分量处理顺序

在整个的压缩过程中使用了多个常量表：亮度量化系数矩阵、色度量化系数矩阵、Huffman 编码表等，这些数据连同 JPEG 压缩方式、图像分辨率等信息存储于 JPEG 文件头中。对于分辨率相同的图像如果使用相同的编码常量数据，则它们的 JPEG 文件头是相同的。

在实际应用中，嵌入式处理器从 MB86S02 获取图像信息后，执行 JPEG 压缩程序，压缩后的 JPEG 文件通过公共电话线路传到监控主机端。由于系统采用相同的图像分辨率和常量表，所以文件头都相同，为了减少传输数据量，不传送文件头，文件头在监控主机端由软件自动添加。

（2）JPEG 解压缩

JPEG 图像文件的解压缩是在 PC 上调用系统函数实现，因为 JPEG 的压缩与解压缩基本对称，所以也可以在本节的硬件平台上实现 JPEG 的解压缩。如图 13-11 所示是 JPEG 图像的解压缩流程。

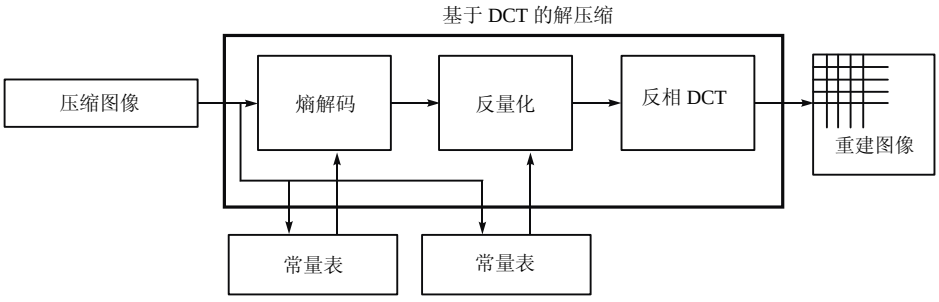


图 13-11　JPEG 解压缩流程

（3）压缩算法的简化设计

- 定点快速 DCT。

$N \times N$ 二维离散余弦变换的数学定义如下：

$$F(u,v)=\frac{2}{N}C(u)C(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\frac{(2x+1)u\pi}{2N}\cos\frac{(2y+1)v\pi}{2N} \quad （公式 13-3）$$

$u,v,x,y=0,1,2,...,N-1$

这里 x,y 为像素域的空间坐标，u,v 为变换域的坐标。

$$C(u),C(v)=\begin{cases}\frac{1}{\sqrt{2}},\text{对于}u,v\\1,\text{其他}\end{cases} \quad （公式 13-4）$$

JPEG 标准还规定了 DCT 的输入使用 9 位，输出 DCT 系数是 12 位，范围是[-2048，2047]。在 JPEG 图像压缩过程中，频繁使用 DCT 变换。对一幅 CIF 分辨率的彩色图像进行 JPEG 压缩需要的 DCT 变换次数是：352÷8×288÷8×2＝3 168 次。如果不使用快速算法，每一个 DCT 运算要进行 8×64×2＝1 024 次浮点乘法，整幅 CIF 图像需要 3 168×1 024＝3 244 032 次浮点乘法，运算量非常大。

对 DCT 运算的简化从两方面入手：首先按照与 FFT 类似的运算方法，考虑到运算中的对称性，减少乘法次数；然后使用定点整数运算代替浮点运算。改进以后一次 DCT 只需要 80 次定点乘法，减少到原来的 8%，大大提高了压缩编码效率。

- 查表量化

在 JPEG 的压缩流程中紧跟着 DCT 的一步操作就是量化，要使用刚计算的 8×8DCT 系数除以 8×8 的量化系数矩阵。由于 ARM720T 处理器核心不包括除法指令，所以这部分操作要通过调用软件除法函数完成。为了进一步提高系统处理速度，使用查表的方法来完成量化操作。在系统运行初期根据选定的量化系数矩阵生成 4096×64B 的量化表，以后的量化操作就变成了查表操作。经过实际测试，查表量化使压缩编码的性能大约提高了 20%。

13.3.3　系统模块源代码实现

1．视频采集数据打包、传输

微驱动接口将应用程序获取图像的命令打包生成数据包，并向微驱动发送。数据包的格式如下：

```
typedef struct Packet
{
    QUE_Elem link;      /* 数据包队列*/
    Ptr addr;           /* 数据地址*/
    Uns size;           /* 数据长度*/
    Arg misc;           /* 保留使用*/
    Arg arg;            /* 应用程序*/
    Uns cmd;            /* 命令字段*/
    Int status;         /* 命令完成状态*/
} Packet;
```

数据包中数据长度与数据地址两字段由应用程序提供，分别表示获取图像的大小及图像存储目的地址。微驱动依据数据包中的命令字段，调用 mdSubmit 函数将数据包放入数据包队列，等待中断服务函数的处理。视频采集中的硬件中断由视频端口内部 FIFO 的状态触发，中断服务程序根据数据包中的数据地址字段，将视频端口内部 FIFO 中的视频数

据读入 SDRAM 中的图像存储目的地址。依据数据包中的数据长度字段，在完成相应大小图像的采集后，中断服务程序还将完成以下功能：出列数据包；设置下一次传送或服务请求；设置数据包中的命令完成状态，并向应用程序返回。

视频端口内部 FIFO 与 SDRAM 之间的视频数据传输通常有以下两种方法：软件查询、中断。软件查询消耗 CPU 的资源太大，是不可取的，中断数据传输不仅可节省很多 CPU 时间，还可以在没有 CPU 参与的情况下完成不同存储空间之间的数据搬移。中断数据传输提供了 64 个独立的通道，通道的优先级可编程设置，在没有 CPU 参与的情况下实现片内存储器、片内外设以及外部存储空间之间的数据高速搬移。因此，为减轻 CPU 的负担，发挥强大的外部数据传输能力，可以使用多通道完成视频数据从 FIFO 到 SDRAM 的传输。

在应用程序通过 GPIO 类驱动调用微驱动之前，需使用配置工具注册微驱动，将其命名为 VP_CAPTURE，并启动 GPIO 模块。在应用程序中，GPIO_create 函数使用已注册的微驱动 VP_CAPTURE 创建 GPIO 通道，通过调用 GPIO_submit 函数完成应用程序对视频数据的采集操作。部分源代码如下。

（1）创建通道

```
GPIO_Handle cap;
int status;
cap = GPIO_create("VP_CAPTURE" ) ,
IO_INPUT, &status, (Ptr)&vp_CapParams, NULL);
```

（2）发送获取图像的数据包

```
GPIO_submit (cap, IO_READ, bufp, NULL, NULL);
```

其中，vp_CapParams 包含了视频采集的初始化参数，如图像大小、同步方式等；bufp 用于指出采集图像的存储地址。不同的视频应用程序在使用类驱动时，可以通过改变这两个变量复用视频设备。这样极大地提高了驱动程序的工作效率，对视频外设的控制也大大简化了。

使用类/微驱动模型开发的视频采集驱动程序，有效地解决了图像采集和图像实时处理之间的关系，在几乎不需要 CPU 的干涉下，完成了数字视频图像数据的高速传输；通过使用类驱动复用驱动程序，视频应用程序的开发效率获得了极大的提高。

2．视频图像数据的压缩

有关图像和视频处理的程序，包括采集的图像直接显示在 VGA 显示器上进行的 YCbCr 或 YUV 到 RGB 的颜色空间转换。

以下是编码头文件的程序：

```
#include "DataType.h"
void ffdct(LINT8 *mcubyte, LINT32 *array)
{
    LINT32 tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
    LINT32 tmp10, tmp11, tmp12, tmp13;
    LINT32 z1, z2, z3, z4, z5, z11, z13;
    LINT32* dataptr;
    LINT8* datain;
    int ctr;
    datain = mcubyte;
    dataptr = array;
    for(ctr=0;ctr<64;ctr++)
```

```
{
    dataptr[ctr]=datain[ctr]-128;
}
                                        /* 第一部分，对行进行计算 */
for (ctr = 7; ctr >= 0; ctr--)
{
    tmp0 = dataptr[0] + dataptr[7];
    tmp7 = dataptr[0] - dataptr[7];
    tmp1 = dataptr[1] + dataptr[6];
    tmp6 = dataptr[1] - dataptr[6];
    tmp2 = dataptr[2] + dataptr[5];
    tmp5 = dataptr[2] - dataptr[5];
    tmp3 = dataptr[3] + dataptr[4];
    tmp4 = dataptr[3] - dataptr[4];
                                        /* 对偶数项进行运算 */
    tmp10 = tmp0 + tmp3;                /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;
    dataptr[0] = tmp10 + tmp11;         /* phase 3 */
    dataptr[4] = tmp10 - tmp11;
    z1 = (tmp12 + tmp13) * (46341);     /* c4 / 0.707106781 */
    z1 = z1 >> 16;
    dataptr[2] = tmp13 + z1;            /* phase 5 */
    dataptr[6] = tmp13 - z1;
                                        /* 对奇数项进行计算 */
    tmp10 = tmp4 + tmp5;                /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;
    z5 = (tmp10 - tmp12) * (25080);     /* c6 /0.382683433 */
    z5 = z5 >> 16;
    z2 = tmp10*(35468);
    z2 = z2 >> 16;
    z2 = z2 + z5;

    z4 = tmp12 * (85627);
    z4 = z4 >> 16;
    z4 = z4 + z5;
    z3 = tmp11 * (46341);               /* c4 / 0.707106781*/
    z3 = z3 >> 16;
    z11 = tmp7 + z3;                    /* phase 5 */
    z13 = tmp7 - z3;
    dataptr[5] = z13 + z2;              /* phase 6 */
    dataptr[3] = z13 - z2;
    dataptr[1] = z11 + z4;
    dataptr[7] = z11 - z4;
    dataptr += 8;                       /* 将指针指向下一行 */
}
                                        /* 第二部分，对列进行计算 */
dataptr = array;
for (ctr = 7; ctr >= 0; ctr--)
{
    tmp0 = dataptr[0] + dataptr[56];
    tmp7 = dataptr[0] - dataptr[56];
    tmp1 = dataptr[8] + dataptr[48];
    tmp6 = dataptr[8] - dataptr[48];
    tmp2 = dataptr[16] + dataptr[40];
    tmp5 = dataptr[16] - dataptr[40];
```

```
        tmp3 = dataptr[24] + dataptr[32];
        tmp4 = dataptr[24] - dataptr[32];
                                                /* 对偶数项进行运算 */
        tmp10 = tmp0 + tmp3;                    /* phase 2 */
        tmp13 = tmp0 - tmp3;
        tmp11 = tmp1 + tmp2;
        tmp12 = tmp1 - tmp2;
        dataptr[0] = tmp10 + tmp11;             /* phase 3 */
        dataptr[32] = tmp10 - tmp11;
        z1 = (tmp12 + tmp13) * (46341);         /* c4 / 0.707106781 */
        z1 = z1 >> 16;
        dataptr[16] = tmp13 + z1;               /* phase 5 */
        dataptr[48] = tmp13 - z1;
                                                /* 对奇数项进行计算 */
        tmp10 = tmp4 + tmp5;                    /* phase 2 */
        tmp11 = tmp5 + tmp6;
        tmp12 = tmp6 + tmp7;
        z5 = (tmp10 - tmp12) * (25080);         /* c6 /0.382683433 */
        z5 = z5 >> 16;
        z2 = tmp10*(35468);
        z2 = z2 >> 16;
        z2 = z2 + z5;
        z4 = tmp12 * (85627);
        z4 = z4 >> 16;
        z4 = z4 + z5;
        z3 = tmp11 * (46341);                   /* c4 / 0.707106781*/
        z3 = z3 >> 16;
        z11 = tmp7 + z3;                        /* phase 5 */
        z13 = tmp7 - z3;
        dataptr[40] = z13 + z2;                 /* phase 6 */
        dataptr[24] = z13 - z2;
        dataptr[8] = z11 + z4;
        dataptr[56] = z11 - z4;
        ++dataptr;                              /* 将指针指向下一列 */
    }
    dataptr = array;
    for(ctr=0;ctr<64;ctr++) dataptr[ctr]=dataptr[ctr]>>3;
}
```

以下是视频图像压缩与编码程序。

```
#include "DataType.h"
#include "tables.h"
#include "dct.h"
#include "Flash.h"
#define DEBUG
#ifdef DEBUG
#include "serial.h"
#endif
#define CIFW 352
#define CIFH 288
LINT8 cha_y[CIFW][CIFH];
LINT8 cha_u[CIFW/2][CIFH/2];
LINT8 cha_v[CIFW/2][CIFH/2];
LINT8 ix=0;
LINT8 iy=0;
LINT32 ydc=0,udc=0,vdc=0;
LINT32 accoder[16][11],acbit[16][11];
```

```
LINT32 accoder2[16][11],acbit2[16][11];

//////////文件部分////////

#define MAXSIZE 262114 //256K
typedef struct{
    LINT8 buffer[MAXSIZE];
    LINT32 fp,lp,CurrentPosition,length;
    LINT8 CurrentData;
    LINT8 CurrentBits;
    }FILEBUF;
FILEBUF infile;
                                                    //输入数据，长度为 bits
void PutDatatoJpegFile(LINT32 data,LINT8 bits,FILEBUF *buf)
{
    LINT8 CurrentFreeBits=0;
    while (bits!=0)
    {
        if (buf->CurrentBits+bits<8)
        {
            buf->CurrentData=(buf->CurrentData<<bits)|data;
            buf->CurrentBits+=bits;
            bits=0;
        }
        else
        {
            CurrentFreeBits=8-buf->CurrentBits;
            bits=bits-CurrentFreeBits;
        buf->CurrentData=(buf->CurrentData<<CurrentFreeBits)|data>>bits;
            buf->buffer[buf->CurrentPosition]=buf->CurrentData;
            buf->CurrentPosition++;
            buf->length++;
                                                //如果 data=0xFF, 加 0
            if (buf->CurrentData==0xff)
            {
                buf->buffer[buf->CurrentPosition]=0;
                buf->CurrentPosition++;
                buf->length++;
            }
            buf->CurrentData=0;
            buf->CurrentBits=0;
            data=data&mask[bits];
        }
    }
}

//////////Huffman 编码 ////////

typedef struct
{
    LINT32 maxcode[16];
    LINT32 mincode[16];
    LINT8 ml;
    unsigned valptr[16];
}DHUFF;

typedef struct
{
```

```
    LINT8 huffval[256];
}XHUFF;

DHUFF dhuff,dhuffdc,dhuff2,dhuffdc2;
XHUFF xhuff,xhuffdc,xhuff2,xhuffdc2;
void len2huff(LINT8 *lengthtab,DHUFF *dhuff,XHUFF *xhuff,LINT8 *val)
{
    int i,l,k,base=0;
    dhuff->ml=16;
    for(i=0;i<dhuff->ml;i++)
    {
        dhuff->mincode[i]=0;
        dhuff->maxcode[i]=0;
        dhuff->valptr[i]=-1;
    }
    for(i=0;i<256;i++)
        xhuff->huffval[i]=0;
        i=0;
        base=0;
    for(l=0;l<dhuff->ml;l++)
    {
        if(lengthtab[l]!=0)
        {
            dhuff->mincode[l]=base;
            dhuff->valptr[l]=i;
        }
        for(k=0;k<lengthtab[l];k++)
        {
            xhuff->huffval[i]=val[i];
            i++;
            base++;
        }
        if(lengthtab[l]!=0)dhuff->maxcode[l]=base-1;
        base=base*2;
    }
}
void iztransfer(LINT32 *array)
{
    LINT32 temp[64],n,x,y;
    for(n=0;n<64;n++)
    {
        x=iztab[2*n];
        y=iztab[2*n+1];
        temp[n]=array[x*8+y];
    }
    for(n=0;n<64;n++)
    array[n]=temp[n];
}
                                                                    //量化
void qdata(LINT32 * in,LINT32 * out,LINT8 level)
{
    int i;
    for(i=0;i<64;i++)
    {
        if(in[i]>0)
            out[i]=in[i]/agraytab[level][i];
        else
            out[i]=in[i]/agraytab[level][i];
    }
```

```
    }
    void quvdata(LINT32 * in,LINT32 * out,LINT8 level)
    {
        int i;
        for(i=0;i<64;i++)
        {
            if(in[i]>0)
                out[i]=in[i]/acolortab[level][i];
            else
                out[i]=in[i]/acolortab[level][i];
        }
    }
                //从 SRAM 0xC0100000 读数据到 cha_y[CIFW][CIFH];
                //cha_u[CIFW/2][CIFH/2];
                //cha_v[CIFW/2][CIFH/2];

    #ifdef DEBUG
    void put_num8(unsigned char i)
    {
        put_char(((((i>>4) & 0x0f) + '0')> '9' )? ((i>>4) & 0x0f) +'0'+7 : ((i>>4)
& 0x0f) +'0' );
        put_char((((i & 0x0f) + '0')> '9' )? (i & 0x0f) +'0'+7 : (i & 0x0f) +'0' );
    }
    static void put_num32(int i)
    {
        put_num8((unsigned char)(( i>>24) & 0xff));
        put_num8( (unsigned char) (i>>16 & 0xff));
        put_num8((unsigned char)(( i>>8) & 0xff));
        put_num8( (unsigned char) (i & 0xff));
    }
    #endif
    void readfile(void)
    {
        int i,count=0;
        for (i=0;i<CIFW*CIFH*6/4;i+=6)
        {
                                                          //取决于图像的帧格式
            cha_y[(i*2/6%CIFW)][(i*2/6/CIFW)*2]=RAM_BYTE(i);
            cha_y[(i*2/6%CIFW)+1][(i*2/6/CIFW)*2]=RAM_BYTE(i+1);
            cha_y[(i*2/6%CIFW)][(i*2/6/CIFW)*2+1]=RAM_BYTE(i+2);
            cha_y[(i*2/6%CIFW)+1][(i*2/6/CIFW)*2+1]=RAM_BYTE(i+3);
            cha_u[i/6%(CIFW/2)][i/6/(CIFW/2)]=RAM_BYTE(i+4);
            cha_v[i/6%(CIFW/2)][i/6/(CIFW/2)]=RAM_BYTE(i+5);
            count++;
        }
    }
                                                          //写编码文件到串行口
    void writefile(FILEBUF *buf)
    {
        int i;
        for (i=0;i<=buf->length;i++)
        {
            put_num8(buf->buffer[i]);
        }
    }
    LINT8 GetLength(LINT32 in)
    {
        int val;
```

```
    if (in<0)
        val=-in;
    else
        val = in;
    if (val>511) return 10;
    else if (val>255) return 9;
    else if (val>127) return 8;
    else if (val>63) return 7;
    else if (val>31) return 6;
    else if (val>15) return 5;
    else if (val>7) return 4;
    else if (val>3) return 3;
    else if (val>1) return 2;
    else if (val>0) return 1;
    else return 0;
}
                                                    //从 22*18 矩阵压缩到 8*8,6(4Y+U+V)
void mcugen(LINT8 x,LINT8 y)
{
    LINT8 p_array[8][8];
    LINT32 dct_array[8][8];
    LINT32 q_array[8][8],diff;
    LINT8 length,flag=0;
    LINT8 AcCount=1,ZeroCount=0;
    unsigned int i,j;

///////////////Y1 亮度////////////////////////////////////

    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
        p_array[i][j]=cha_y[x*16+j][y*16+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        qdata(&dct_array[0][0],&q_array[0][0],5);
                                                                        //DC 编码
        diff=q_array[0][0]-ydc;
        ydc=q_array[0][0];
    if(diff<0)
    {
        diff=-diff;
        flag=1;
    }
    length=GetLength(diff);
    if(flag==1)
    {
        diff=two[length]-1-diff;
    }
    PutDatatoJpegFile(dccoder[length],dcbit[length],&infile);
    PutDatatoJpegFile(diff,length,&infile);
                                                                        //AC 编码
while(AcCount<64)
{
        if (q_array[AcCount/8][AcCount%8]==0)
        {
            AcCount++;
            ZeroCount++;
            if (AcCount==64)
            {
```

```
                    PutDatatoJpegFile(accoder[0][0],acbit[0][0],&infile);
                }
            }
            else
            {
                while (ZeroCount>=16)
                {
                    PutDatatoJpegFile(accoder[15][0],acbit[15][0],&infile);
                    ZeroCount-=16;
                }
                flag=0;
                diff=q_array[AcCount/8][AcCount%8];
                if (diff<0)
                {
                    diff=-diff;
                    flag=1;
                }
                length=GetLength(diff);
                if (flag==1)
                {
                    diff=two[length]-1-diff;
                }
        PutDatatoJpegFile(accoder[ZeroCount][length],acbit[ZeroCount][lengt
h],&infile);
            PutDatatoJpegFile(diff,length,&infile);
            AcCount++;
            ZeroCount=0;
        }
    }

    ////////////////////////////Y2/////////////////////////////////////////

    flag=0;
    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
        p_array[i][j]=cha_y[x*16+8+j][y*16+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        qdata(&dct_array[0][0],&q_array[0][0],5);
                                                          //DC 编码
        diff=q_array[0][0]-ydc;
        ydc=q_array[0][0];
    if(diff<0)
    {
        diff=-diff;
        flag=1;
    }
    length=GetLength(diff);
    if(flag==1)
    {
        diff=two[length]-1-diff;
    }
    PutDatatoJpegFile(dccoder[length],dcbit[length],&infile);
    PutDatatoJpegFile(diff,length,&infile);
                                                          //AC 编码
    AcCount=1;ZeroCount=0;
    while(AcCount<64)
    {
```

```
        if (q_array[AcCount/8][AcCount%8]==0)
        {
            AcCount++;
            ZeroCount++;
            if (AcCount==64)
            {
                PutDatatoJpegFile(accoder[0][0],acbit[0][0],&infile);
            }
        }
        else
        {
            while (ZeroCount>=16)
            {
                PutDatatoJpegFile(accoder[15][0],acbit[15][0],&infile);
                ZeroCount-=16;
            }
            flag=0;
            diff=q_array[AcCount/8][AcCount%8];
            if (diff<0)
            {
                diff=-diff;
                flag=1;
            }
            length=GetLength(diff);
            if (flag==1)
            {
                diff=two[length]-1-diff;
            }
        PutDatatoJpegFile(accoder[ZeroCount][length],acbit[ZeroCount][lengt
h],&infile);
        PutDatatoJpegFile(diff,length,&infile);
        AcCount++;
        ZeroCount=0;
        }
    }

    ////////////////////////////Y3/////////////////////////////////////

    flag=0;
    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
        p_array[i][j]=cha_y[x*16+j][y*16+8+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        qdata(&dct_array[0][0],&q_array[0][0],5);
                                                                    //DC 编码
        diff=q_array[0][0]-ydc;
        ydc=q_array[0][0];
        if(diff<0)
        {
            diff=-diff;
            flag=1;
        }
        length=GetLength(diff);
        if(flag==1)
        {
            diff=two[length]-1-diff;
        }
        PutDatatoJpegFile(dccoder[length],dcbit[length],&infile);
```

```
        PutDatatoJpegFile(diff,length,&infile);
                                                                    //AC 编码
        AcCount=1;ZeroCount=0;
    while(AcCount<64)
    {
            if (q_array[AcCount/8][AcCount%8]==0)
            {
                AcCount++;
                ZeroCount++;
                if (AcCount==64)
                {
                    PutDatatoJpegFile(accoder[0][0],acbit[0][0],&infile);
                }
        }
        else
        {
            while (ZeroCount>=16)
            {
                PutDatatoJpegFile(accoder[15][0],acbit[15][0],&infile);
                ZeroCount-=16;
            }
            flag=0;
            diff=q_array[AcCount/8][AcCount%8];
            if (diff<0)
            {
                diff=-diff;
                flag=1;
            }
            length=GetLength(diff);
            if (flag==1)
            {
                diff=two[length]-1-diff;
            }
        PutDatatoJpegFile(accoder[ZeroCount][length],acbit[ZeroCount][lengt
h],&infile);
            PutDatatoJpegFile(diff,length,&infile);
            AcCount++;
            ZeroCount=0;
        }
    }

    ////////////////////////////Y4/////////////////////////////////////

    flag=0;
    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
        p_array[i][j]=cha_y[x*16+8+j][y*16+8+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        qdata(&dct_array[0][0],&q_array[0][0],5);
                                                                    //DC 编码
        diff=q_array[0][0]-ydc;
        ydc=q_array[0][0];
        if(diff<0)
        {
            diff=-diff;
            flag=1;
        }
```

```
        length=GetLength(diff);
        if(flag==1)
        {
            diff=two[length]-1-diff;
        }
        PutDatatoJpegFile(dccoder[length],dcbit[length],&infile);
        PutDatatoJpegFile(diff,length,&infile);
                                                                    //AC 编码
        AcCount=1;ZeroCount=0;
        while(AcCount<64)
        {
            if (q_array[AcCount/8][AcCount%8]==0)
            {
                AcCount++;
                ZeroCount++;
                if (AcCount==64)
                {
                    PutDatatoJpegFile(accoder[0][0],acbit[0][0],&infile);
                }
            }
            else
            {
                while (ZeroCount>=16)
                {
                    PutDatatoJpegFile(accoder[15][0],acbit[15][0],&infile);
                    ZeroCount-=16;
                }
                flag=0;
                diff=q_array[AcCount/8][AcCount%8];
                if (diff<0)
                {
                    diff=-diff;
                    flag=1;
                }
                length=GetLength(diff);
                if (flag==1)
                {
                    diff=two[length]-1-diff;
                }
        PutDatatoJpegFile(accoder[ZeroCount][length],acbit[ZeroCount][lengt
h],&infile);
            PutDatatoJpegFile(diff,length,&infile);
            AcCount++;
            ZeroCount=0;
        }
    }

    ////////////////////////////U色度////////////////////////////////////

    flag=0;
    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
        p_array[i][j]=cha_u[x*8+j][y*8+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        quvdata(&dct_array[0][0],&q_array[0][0],5);
                                                                    //DC 编码
        diff=q_array[0][0]-udc;
```

```
        udc=q_array[0][0];
    if(diff<0)
    {
        diff=-diff;
        flag=1;
    }
    length=GetLength(diff);
    if(flag==1)
    {
        diff=two[length]-1-diff;
    }
    PutDatatoJpegFile(dccoder2[length],dcbit2[length],&infile);
    PutDatatoJpegFile(diff,length,&infile);
                                                                    //AC 编码
    AcCount=1;ZeroCount=0;
    while(AcCount<64)
    {
        if (q_array[AcCount/8][AcCount%8]==0)
        {
            AcCount++;
            ZeroCount++;
            if (AcCount==64)
            {
                PutDatatoJpegFile(accoder2[0][0],acbit2[0][0],&infile);
            }
        }
        else
        {
            while (ZeroCount>=16)
            {
                PutDatatoJpegFile(accoder2[15][0],acbit2[15][0],&infile);
                ZeroCount-=16;
            }
            flag=0;
            diff=q_array[AcCount/8][AcCount%8];
            if (diff<0)
            {
                diff=-diff;
                flag=1;
            }
            length=GetLength(diff);
            if (flag==1)
            {
                diff=two[length]-1-diff;
            }
        PutDatatoJpegFile(accoder2[ZeroCount][length],acbit2[ZeroCount][len
gth],&infile);
        PutDatatoJpegFile(diff,length,&infile);
        AcCount++;
        ZeroCount=0;
        }
    }

    ////////////////////////////V方向////////////////////////////////////

    flag=0;
    for(i=0;i<8;i++)
    for(j=0;j<8;j++)
```

```
        p_array[i][j]=cha_v[x*8+j][y*8+i];
        ffdct(&p_array[0][0],&dct_array[0][0]);
        iztransfer(&dct_array[0][0]);
        quvdata(&dct_array[0][0],&q_array[0][0],5);
                                                                    //DC 编码
        diff=q_array[0][0]-vdc;
        vdc=q_array[0][0];
    if(diff<0)
    {
        diff=-diff;
        flag=1;
    }
    length=GetLength(diff);
    if(flag==1)
    {
        diff=two[length]-1-diff;
    }
    PutDatatoJpegFile(dccoder2[length],dcbit2[length],&infile);
    PutDatatoJpegFile(diff,length,&infile);
                                                                    //AC 编码
    AcCount=1;ZeroCount=0;
    while(AcCount<64)
    {
        if (q_array[AcCount/8][AcCount%8]==0)
        {
            AcCount++;
            ZeroCount++;
            if (AcCount==64)
            {
                PutDatatoJpegFile(accoder2[0][0],acbit2[0][0],&infile);
            }
        }
        else
        {
            while (ZeroCount>=16)
            {
                PutDatatoJpegFile(accoder2[15][0],acbit2[15][0],&infile);
                ZeroCount-=16;
            }
            flag=0;
            diff=q_array[AcCount/8][AcCount%8];
            if (diff<0)
            {
                diff=-diff;
                flag=1;
            }
            length=GetLength(diff);
            if (flag==1)
            {
                diff=two[length]-1-diff;
            }
        PutDatatoJpegFile(accoder2[ZeroCount][length],acbit2[ZeroCount][len
gth],&infile);
            PutDatatoJpegFile(diff,length,&infile);
            AcCount++;
            ZeroCount=0;
            }
        }
```

```
}
void C_vMain(void)
{
    LINT32 i,j;
    LINT8 x;
    #ifdef DEBUG
        put_string("Load...\n");
        put_char('\n');
    #endif
    len2huff(dclength,&dhuffdc,&xhuffdc,(LINT8*)dcval);
    len2huff(aclength,&dhuff,&xhuff,(LINT8*)acval);
    len2huff(dclength2,&dhuffdc2,&xhuffdc2,(LINT8*)dcval2);
    len2huff(aclength2,&dhuff2,&xhuff2,(LINT8*)acval2);
    #ifdef DEBUG
        put_string("Start...\n");
    #endif
    for(i=0;i<16;i++)
    for(j=0;j<11;j++)
    {
        accoder[i][j]=0;
        acbit[i][j]=0;
    }
    for(i=0;i<16;i++)
    {
        for(j=0;j<dhuff.maxcode[i]-dhuff.mincode[i]+1;j++)
        {
            if(dhuff.valptr[i]!=-1)
            {
                x=acval[dhuff.valptr[i]+j];
                accoder[x/16][x%16]=dhuff.mincode[i]+j;
                acbit[x/16][x%16]=i+1;
            }
        }
    }
    #ifdef DEBUG
        put_string("Step1...\n");
    #endif
    for(i=0;i<16;i++)
    for(j=0;j<11;j++)
    {
        accoder2[i][j]=0;
        acbit2[i][j]=0;
    }
    for(i=0;i<16;i++)
    {
        for(j=0;j<dhuff2.maxcode[i]-dhuff2.mincode[i]+1;j++)
        {
            if(dhuff2.valptr[i]!=-1)
            {
                x=acval2[dhuff2.valptr[i]+j];
                accoder2[x/16][x%16]=dhuff2.mincode[i]+j;
                acbit2[x/16][x%16]=i+1;
            }
        }
    }
    MakeCt();
    readfile();
    for(j=0;j<18;j++)
```

```
    {
        for(i=0;i<22;i++)
        {
            mcugen(i,j);
        }
    }
    writefile(&infile);
    #ifdef DEBUG
        put_string("Done!\n");
        while(1);
    #endif
}
```

3．Linux Frame-Buffer 实时显示

目前 Linux 2.2x 以上版本内核支持 Frame-Buffer 的显示驱动技术。由于利用 Frame-Buffer 这种技术，能够很方便地实现视频显示，因此在编译嵌入式 Linux 内核时，应当将这项特性包括在内。从应用程序的角度看，Frame-Buffer 驱动主要使用了两种数据结构和几种有限的 ioctl 参数控制显示模式。视频数据的读写是通过 mmap ()的方法直接对显存操作。

```
ioctl ( screen _ fd , FBIOGET _ FSCREENINFO , &fix _screeninfo) ;
ioctl ( screen _ fd , FBIOGET _ VSCREENINFO , &var _screeninfo) ;
screen_pt r = (char 3 ) mmap (......) ;/ / 循环*
```

FBIOGET_FSCREENINFO 获得 Frame-Buffer 设备无关的固定信息，如显存大小、每行字符数、颜色模式等。而 FBIOGET_VSCREENINFO 获得设备无关的可变信息，如显示屏的长、宽、颜色深度等，需要根据具体应用设置。

用户可以将 Frame-buffer 看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作了，而写操作可以立即反映在屏幕上。这样把帧缓冲区 Frame-Buffer 看成一块内存，既可以向这块内存中写入数据，也可以从其中读出数据。

```
void display(uchar x,uchar y uchar const *m)
{
        uchar i;
        uchar temp;
        unsigned int z,cursorh,cursorl;
                                                //0x4000 为 Frame-buffer 首地址
        z=x*40+y+0x4000;
        cursorh=z/256;
        cursorl=z%256;
        wcom(Ox4f);
        wcom(Ox46);
        wdata(cursorl);
        wdata(cursorh);

        wcom(Ox42);
        for(i=O;i<8;i++)
        {
            temp=*(m+i);
            wdata(temp);
        }
}
```

以上程序用共享存储区域（虚拟的帧缓冲 Frame-Buffer）来模拟帧缓冲，并且模拟一个应用来显示帧缓冲 Frame-Buffer，显示的区域被周期性地改变和更新。通过指定显示设

备的宽度和颜色深度，虚拟出来的缓冲帧和物理的显示设备在每个像素上保持一致。

13.3.4　视频数据比较及分析

1．各压缩等级文件大小及图像效果对比

使用自己编写的 JPEG 压缩程序，对测试图像按不同压缩等级压缩，得到的压缩结果如表 13-5 所示。图 13-12 所示为各种压缩等级效果比较图。

表 13-5　JPEG 在不同压缩等级下的实际数据比较表（原始图像：297.1KB）

压缩等级	压缩文件大小	压 缩 比	图像质量评价
1	28.6KB	10:1	无差别
2	19.9KB	15:1	微小差别（可见噪点）
3	16.3KB	18:1	微小差别（噪点增加）
4	13.9KB	21:1	微小差别（噪点继续增加）
5	12.4KB	24:1	微小差别（边缘渐现振零现象）
6	10.9KB	27:1	较小差别（边缘噪声加大）
7	9.3KB	32:1	色度分量开始出现块状
8	7.4KB	40:1	图像边缘噪声加大，出现块状痕迹
9	5.0KB	59:1	图象质量一般，出现明显块状痕迹
10	2.6KB	114:1	图像大量色块堆积，可辨认大体轮廓



原图　　压缩等级 6

压缩等级 9　　压缩等级 10

图 13-12　JPEG 各压缩等级效果比较

2．JPEG 压缩等级分析

如图 13-13 所示是根据表 13-5 生成的 JPEG 各压缩等级比较图，图中圆饼的面积代表文件的大小，横轴对应压缩等级，纵轴对应压缩比。可见从等级 7 开始压缩比迅速提高，但是考虑到图像质量，这里可以使用等级 7 进行压缩。

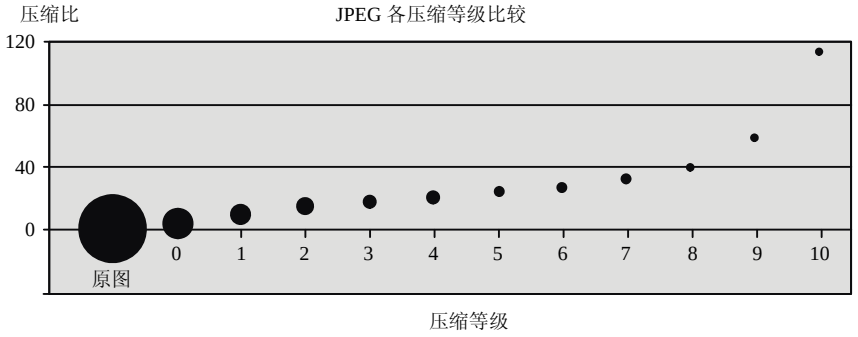


图 13-13　JPEG 各压缩等级比较图

JPEG 压缩的各种等级实际上就对应不同的量化系数矩阵，等级越高量化系数矩阵中元素的值越大，经过量化后的 DCT 系数的绝对值就越小，编码所需的比特也越少。例如，等级 0 的量化系数矩阵为就是全 1 的系数矩阵；等级 10 的量化系数矩阵为全'xFF。

13.4　本章总结

本章详细介绍了通过 MB86S02 CMOS 视频采集实现嵌入式视频采集的开发实例。利用嵌入式系统开发的特点和优点，可以提高视频采集系统的应用性能及范围。用户编写嵌入式 Linux 的设备驱动程序，可以更好地使用新硬件特性，提高系统访问硬件的效率，改善整个应用系统的性能。由于修改驱动程序非常方便，应用系统的设计也因此变得较灵活。读者学习时，要多用心体会开发思路和设计思想，以达到举一反三、融会贯通的目的。

第 14 章

指纹识别门禁系统开发实例

指纹识别技术是生物特征识别技术中的一种。顾名思义，生物特征识别技术是利用人体的生物特征来进行身份验证的一种解决方案。由于人的生物特征具有人体所固有的不可复制的唯一性，因此这一生物特征密钥无法复制、失窃或遗忘。利用生物特征识别技术，人们开发了指纹识别、语音识别、虹膜识别以及面部特征识别等多种系统，而且许多系统都已经发展成熟并得以应用，其中的指纹识别技术更是生物识别技术的热点。

如图 14-1 所示是基于嵌入式的指纹识别门禁系统，指纹识别门禁技术的发展得益于现代电子集成制造技术和快速可靠的算法研究。尽管指纹只是人体皮肤的小部分，但用于识别的数据量相当大，对这些数据进行比对也不是简单的相等与不相等的问题，而是需要使用进行大量运算的模糊匹配算法。另外，随着匹配算法可靠性的不断提高，指纹识别技术已经非常实用。

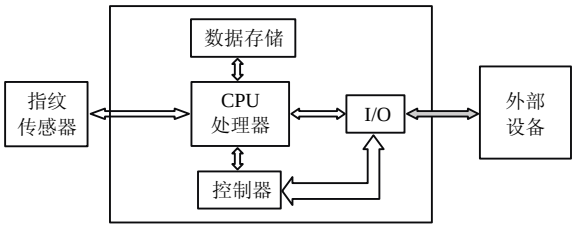


图 14-1　基于嵌入式的指纹识别门禁系统

按照一般人的看法，指纹识别技术是通过分析指纹的全局特征和指纹的局部特征，这些特征点如脊、谷和终点、分叉点或分歧点，从指纹中抽取详尽的特征值，以便可靠地通过指纹来确认一个人的身份。平均每个指纹都有几个独一无二可测量的特征点，每个特征点都有大约 7 个特征，10 个手指产生最少 4900 个独立可测量的特征；这足够来确认指纹识别是否是一个更加可靠的鉴别方式。

两种用来采集指纹图像的技术主要为光学技术和电容技术。光学技术需要一个光源从棱镜反射按在一个取像头的手指，光线照亮指纹从而采集到指纹。采用电容技术的半导体技术，其原理是由按压到采集头上的手指的脊和谷在手指表皮和芯片之间产生不同的电容，芯片通过测量空间中的不同的电容场得到完整的指纹。

电容技术的缺点是因为电容技术的芯片昂贵，芯片的大小和手指相当就已相当昂贵，故几个公司试图推出可提供比指纹更小的芯片只采集部分的指纹加以验证，使用这种采集方式，用户必须精确地放上手指以确保能正确地读取。而这样必然使读取头变得不易使用，

使用这种小芯片的另一个缺点是只使用部分的指纹必然没有采集全部指纹进行比对可靠。电容采集头的另一个缺点是容易受到干扰，如从 60Hz 的电缆线的干扰到用户接触时的干扰、指纹采集器内部的电干扰等。电容采集头的最后一个问题是可靠性，无论是静电干扰，汗液中的盐分或者其他的脏物，以及手指磨损都会使采集头受到影响。

实际上，到目前为止，通过改进原来的光学取像技术，新一代的光学指纹采集器以相对非常低的价格使电容方案相形见绌。但是有个比较严重的问题是采用光学采集头时指纹图像比较容易被伪造和滥用。从可靠性上来讲电容式采集头还是比较有优势的。

指纹识别技术的优点是：指纹是人体独一无二的特征，并且它们的复杂度足以提供用于鉴别的足够特征；如果想要增加可靠性，只需登记更多的指纹，鉴别更多的手指，最多可以多达十个，而每一个指纹都是独一无二的；扫描指纹的速度很快，使用非常方便；读取指纹时，用户必须将手指与指纹采集头相互接触，与指纹采集头直接接触是读取人体生物特征最可靠的方法。这也是指纹识别技术能够占领大部分市场的一个主要原因；指纹采集头可以更加小型化，并且价格会更加低廉。

指纹识别也存在有缺点：某些人或某些群体的指纹因为指纹特征很少，故而很难成像；每一次的使用指纹时都会在指纹采集头上留下用户的指纹印痕，而这些指纹痕迹存在被用来复制指纹的可能性。

经过综合比较可以看到，指纹识别技术是目前最方便、可靠、非侵害和价格便宜的生物识别技术解决方案。

14.1 应用环境与硬件设计概要

本指纹识别门禁系统采用电容式压感指纹传感器（FPS200 芯片）并且基于网络的指纹门禁考勤系统。

1．系统功能概述

（1）指纹门禁/考勤

指纹门禁：在终端上录入指纹，当指纹合法时，可以开门。

指纹考勤：在终端上录入指纹，做上班或者下班的考勤记录，用这个记录可以计算用户的工资、出勤情况。

（2）分布式和非分布式

分布式：终端将采集的指纹原始数据发送到服务器，由服务器进行指纹的识别算法，计算完毕以后，服务器决定是否向终端发送开门命令。为了与局域网兼容，采用 10M 以太网将终端和服务器进行连接。由于分布式系统没有复杂的算法，所以成本比较低，适用于门禁/考勤比较集中的地方，比如宾馆、办公楼等。

非分布式：非分布式指一个终端系统就可以进行指纹算法等复杂功能，基本上不需要服务器帮助处理。

（3）终端和服务器

终端有指纹采集、屏幕输出、键盘输入、声音报警、LED 灯指示（有绿色和红色各一只，绿色提示成功，红色提示错误）功能，终端通过 10M 以太网和服务器连接。终端机负责采集指纹原始数据并发送到服务器，不对指纹数据做处理。

服务器中的服务程序能够和终端机进行数据交换、指纹对比算法、查询数据库等复杂

的功能。指纹合法，发送开门命令；非法，则报警。

2．系统主要功能描述

（1）门禁功能

如果门禁功能被激活，当按下手指以后，服务器将判断是否要发送开门命令。

- 时段处理：在终端处输入 ID，然后用户按下手指。检查 ID 和手指是否符合，再看此用户是否允许在这个时间打开此门（全部符合才开门）。
- 多人开门：对于重要的场合，需要输入多个指纹才能开门。用户先输入组 ID，然后按手指，如果服务器发现要再输入一个手指，则提示输入下一个手指，直至输入全部手指以后发送一个开门命令。
- 指纹出错的报警：如果某次服务器发现输入的指纹不正确时，可以发送报警命令给终端。这个报警是否发送可以在服务器端选择，比如，白天有人在就不需要报警，报警的声音反而会带来噪声。
- 胁迫开门的报警：合法用户事先输入报警手指。当用户被胁迫开门时可以输入报警手指。终端处没有反应只在服务器端报警。报警的条件是 ID 和报警手指相符，在哪个终端录入不管。
- 在特殊情况下用户可以用输入密码来代替指纹的输入，过程与上述情况一样。
- 设置一个超级用户密码，这个密码在产品出厂时不告诉用户。如果用户确实需要超级用户密码开门，可以用程序的“生成超级用户密码序号”功能生成一个随机的序列号，然后将此序列号告诉厂家，厂家用特定程序生成此序列号对应的超级密码的前半部分，超级密码的后半部分是系统管理员密码，并且此密码指纹用于开门一次。这样只有管理员和厂家联合才能打开此门。

必须有个管理程序用于设置各个终端机上各个用户的开门时间。每个终端有一个特定的终端机号。只有在终端机号、开门时间、ID 和指纹都相符时才能开门。

（2）考勤功能

如果考勤功能被激活，则当按下手指以后，服务器首先判断 ID 和指纹是否相符，然后再看这个终端机是否在这个时间允许这个用户考勤，这一步的验证和门禁是一样的。如果是就做一次考勤记录，并且查询数据库，显示是否迟到、个人留言等信息。

3．系统硬件结构

这里的硬件只针对终端的硬件，硬件组成包括 3 部分：指纹传感器、核心处理板，以及外部控制板。其外部控制板上只有数个按键及 LED 用于指示部分工作状态。其硬件结构示意图如图 14-2 所示，由图 14-2 可以看出，终端的硬件结构主要包括以下几个部分。

- 基于嵌入式 Linux 的 ARM 处理器 CPU；
- DIP 封装的 SRAM 芯片；
- 32Kbit 串行 I^2C 总线 Flash 芯片做配置数据及相关 LCD 显示的字模数据存储器；
- 电容式触感指纹识别芯片及其外围电路；
- 128×64 点阵图形字符液晶显示器作为显示输出设备；
- GAL 器件及外围电路做寻址逻辑控制；
- RTL8019AS 网卡芯片及外围电路做终端与服务器信息交换；
- RS232 串口做程序调试接口；

- 数个 LED 及继电器做门锁控制输出。

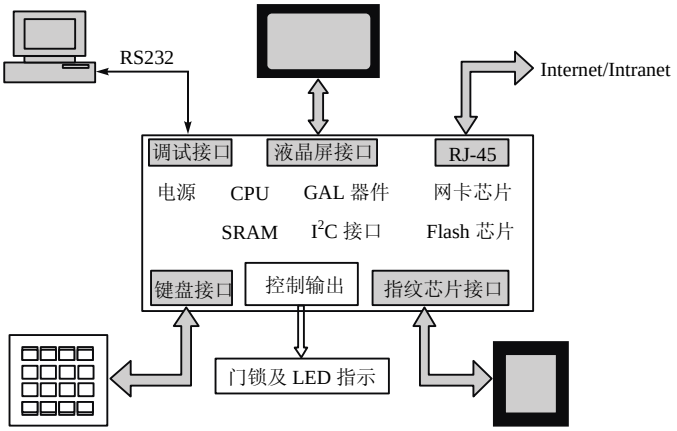


图 14-2　指纹识别门禁系统终端结构示意图

该系统终端部分工作于网络模式下，亦即联机方式，离开服务器和网络将无法工作。其工作流程为：

（1）终端系统上电后的各部件复位及初始化过程。首先是对各部件的配置初始化，然后由网卡发送与服务器联络数据包，总共发送 5 次，如果得到服务器正确的应答则转入联机工作状态，如果没有得到服务器的正确应答则重新复位及初始化，然后再发送联络数据包。

（2）终端正常工作状态亦即循环查询是否有按键或者是指纹按下，分别进行处理，处理完成后返回到等待状态。

（3）按键处理过程是等待状态的一个子状态。首先判断是进入系统设置菜单还是用户 ID（因为系统包括用户可以用 ID+密码开门的功能，是否能用密码开门可以在服务器端由系统管理员进行每个用户的属性设置），然后分别进行处理，在进入系统设置菜单时要进行密码验证。

14.2　相关开发技术

14.2.1　指纹识别原理

指纹是手指末端正面皮肤上凸凹不平产生的纹路。尽管指纹只是人体皮肤的一小部分，但是它蕴涵大量的信息。指纹特征可分为两类：总体特征和局部特征。总体特征指那些用人眼直接就可以观察到的特征，包括基本纹路图案、模式区、核心点、三角点、式样线和纹数等。基本纹路图案有环形、弓形、螺旋形。局部特征指指纹上的特征点，即指纹纹路上的终结点、分叉点和转折点。这些指纹特征点可用以下 4 种特性来描述。

（1）位置：特征点的位置通过（x,y）坐标来描述，可以是绝对的，也可以是相对于三角点的。

（2）方向：该特征点所在的局部脊线的方向。

（3）分类：特征点有终结点、分叉点、分歧点、孤立点、环点、短纹等类型。最典型的终结点和分叉点如图 14-3 所示。

（4）脊线：特征点对应的脊线（d_i，a_i）。特征点对应的脊线用在该脊线上的采样点来

表示。采样点用该点与对应特征点的距离 d_i，连接该点与对应特征点的直线，以及对应特征点方向的夹角 a_i 来表示。

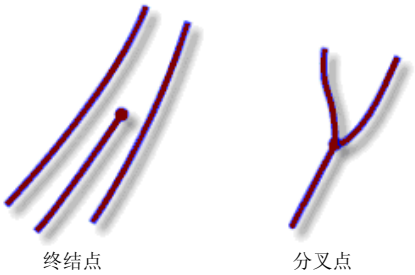


图 14-3　典型的指纹特征点

指纹识别技术通常使用指纹的总体特征如纹形、三角点等来进行分类，再用局部特征如位置和方向等来进行识别用户身份。通常首先从获取的指纹图像上找到“特征点”（minutiae），然后根据特征点的特性建立用户活体指纹的数字表示——指纹特征数据（一种单向的转换，可以从指纹图像转换成特征数据但不能从特征数据转换成为指纹图像）。由于两枚不同的指纹不会产生相同的特征数据，所以通过对所采集到的指纹图像的特征数据和存放在数据库中的指纹特征数据进行模式匹配，计算出它们的相似程度，最终得到两个指纹的匹配结果，根据匹配结果来鉴别用户身份。

总之，指纹识别技术首先通过读取指纹图像，然后用计算机识别软件提取指纹的特征数据，最后通过匹配识别算法得到识别结果。其基本原理框图如图 14-4 所示。

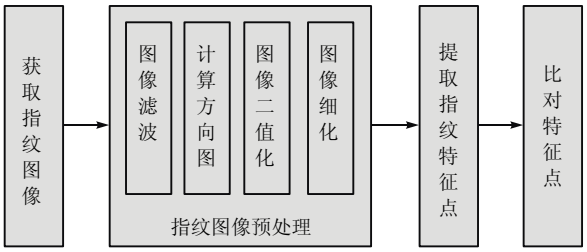


图 14-4　指纹识别原理框图

由指纹识别的基本原理可知，指纹识别技术主要经过以下 4 个步骤：指纹图像的获取、指纹图像的预处理、指纹特征的提取和指纹特征匹配。下面将对指纹识别技术的每个步骤做详细论述。

（1）指纹图像的获取

获取指纹图像的设备可分成三类：光学、硅晶体传感器和其他。光学取像设备应用的历史最久，它依据的是光的全反射原理。

应用晶体传感器是最近在市场上才出现的，这些含有微型晶体的平面通过多种技术来绘制指纹图像。

电容传感器就是其中的一种，它通过电子度量被设计来捕获指纹图像。电容设备能结合大约 100 000 导体金属阵列的传感器，其外面是绝缘的表面，当用户的手指放在上面时，皮肤组成了电容阵列的另一面。电容器的电容值由于金属间的距离而变化，这里指的是脊（近的）和谷（远的）之间的距离。除了以上两类，超声波扫描被认为是指纹取像技术中非常好

的一类，但价格太高、体积偏大。一般采集到的指纹图像都是存成 256 级灰度的图像。

（2）指纹图像预处理

为了得到比较准确的指纹特征点，指纹图像预处理一般要经过图像增强（滤波去掉噪声）、计算方向图、二值化和细化等过程。整个过程如图 14-5 所示。

- 图像增强

一般来说，刚获得的指纹图像都有很多噪声，比如，手指被弄脏，手指有疤痕、太干、太湿或撕破等，所以如何在获取指纹图像之后，有效地过滤图像噪声是指纹识别技术中的难题之一。

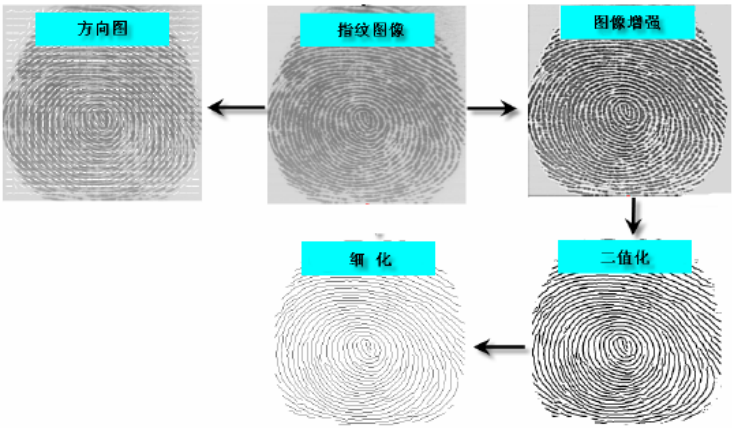


图 14-5　指纹预处理

通过图像增强可以过滤噪声，增强脊和谷的对比度。图像增强的方法有很多，但大多数是通过过滤图像与脊局部方向相匹配。图像首先被分成几个小区域（窗口），并在每个区域上计算出脊的局部方向来决定方向图，可以由空间域处理，或经过快速二维傅立叶变换后的频域处理来得到每个小窗口上的局部方向。然后设计合适的，相匹配的滤镜，使之适用于图像上所有的像素（空间场是其中的一个）。依据每个像素处脊的局部走向，滤镜应增强在同一方向脊的走向，并且在同一位置减弱任何不同于脊的方向。由于后者含有横跨脊的噪声，所以其垂直于脊的局部方向上的那些不正确的“桥”会被滤镜过滤掉。

- 计算方向图

方向图描述了指纹图像中每一像素点所在脊线或谷线在该点的切线方向，作为一种可直接从源灰度图像中得到的有用信息，它的计算一直是指纹识别技术中必不可少的一步。

方向图也可以看做是原始指纹源图像的一种变换表示方法，即用纹线上某点的方向来表示该纹线的方向。一般有两种方向图：一种是点方向图，表示原始指纹图像中每一像素点脊线的方向；另一种是块方向图，表示原始指纹图像中某点区域所有元素的平均方向。

计算方向图的基本思想是：在原始灰度指纹图像中计算每一点（或每一块）在各个方向上的某个统计量（如灰度差、梯度等），根据这些统计量在各个方向上的差异，确定该点（该块）的方向。

在实际处理中，往往采用块方向图，因为块方向图常常比点方向图有更强的抗噪性，而且块方向图可以减少计算量，有利于模块化处理。块方向图可以由点方向图得到，也可以用最小均方估计算法求得。

- 二值化

首先，根据指纹的脊线和谷线等宽的假设，再结合局部灰度分布的考察，可以得到具有自适应性的自动门限。自适应阈值的选取方法是先找到该点的法向方向，在理想情况下，法向上的平均值即可作为阈值。然而考虑到噪声的影响，故应该去掉最大最小值后的点的平均值再加上一修正值作为阈值。计算公式如下：

$$TT = R - (T - R)/2 \qquad \text{（公式 14-1）}$$

其中，R 为法向上去掉最大、最小点后的平均值；T 为最大、最小点的平均值；（T-R）/2 为修正值；TT 为阈值。阈值选定后，即可对该点进行二值化，逐点依次处理即可。

第二，在指纹图像中，考虑同一区域的像元应具有相近的连续变化的灰度，根据“灰度变化平稳”这一假设邻元灰度的变化来进一步确认像元素隶属前景和背景的程度，可以很好地排除不清晰指纹在自动门限附近的分割不一致性。

第三，为解决二值化在分割图像中视野太小的局限，并同时对模糊区域和孤立噪声进行处理，采用广义的拉普拉斯算子对图像进行滤波。

$\nabla_k^2 f(x,y) = f(x+k,y) + f(x-k,y) + f(x,y-k) - 4f(x,y)$，$k$ 为步长　（公式 14-2）

实验表明，该算法不但可使纹路突出，而且较好地保留了指纹的细节特征，并且在很大程度上减少了指纹的断缝和粘连等错误信息。

二值操作使一个灰度图像变成二值图像，图像在强度层次上从原始的 256 色降为 2 色。图像二值化后，随后的处理就会比较容易。

二值化的困难在于因为并不是所有的指纹图像都有相同的阈值，所以一般不能从单纯的强度入手，而且单一图像的对照物是变化的，比如，由于手在中心地带按得比较紧，因此一个叫“局部自适应的阈值”的方法被用来决定局部图像强度的阈值。

- 细化

在提取指纹特征点之前的最后一道工序是“细化”。细化是在不影响原图的拓扑连接关系下，将脊的宽度降为单个像素的宽度的处理过程。一个好的细化方法是保持原有脊的连续性，降低由于人为因素所造成的影响。人为因素主要有毛刺和短脊线，这些都造成提取出来的特征中有很多伪特征。

细化方法的优点是减少内存空间，它只需要存储图像中必需的结构信息。这样在对图像的处理中能简化数据结构。

根据细化的定义易知细化的关键是如何找到原图像的骨架，通常采用模板匹配方法，这种方法是根据某个像素的局部邻域的图像特征对其进行处理。当然也有外轮廓计算、神经网络等细化方法。

（3）提取指纹特征点

如表 14-1 所示，特征提取用一个 3×3 的模板来检测特征点的位置与类型，M 是被检测的指纹特征点，N_0，…，N_7 是特征点 M 按逆时针方向排列的邻近点。

表 14-1　特征点的位置与类型

N_3	N_2	N_1
N_4	M	N_0
N_5	N_6	N_7

如果 $C_N=\sum_{k=0}^{7}\left|N_{k+1}-N_k\right|=2$，其中 $N_8=N_0$，则 M 是终结点；如果 $C_N=\sum_{k=0}^{7}\left|N_{k+1}-N_k\right|=6$，其中 $N_8=N_0$，则 M 是分叉点，如图 14-3 所示。

由于图像噪声等因素的影响，从上述算法提取出来的特征点中有许多伪特征点有待删除，伪特征点的删除可以分为两个步骤。

- 如果脊图中的一段与局部脊方向完全正交，而且其长度小于定值 T，那么这段脊就会被消除。如果脊中的一个间断很短，没有其他脊穿过，缺的这段脊就应该补上。
- 如果一个小区域中的细节形成一簇，那么只留下最靠近中心点的一个，如果两个细节非常接近，而且中间没有脊，那么消除这两个细节。

进行特征提取后，对于每个特征应该保留以下的参数：特征点的 X 坐标和 Y 坐标、特征点的方向即与特征点相连的局部脊方向、特征点的类型即是终结点还是分叉点和与特征点相连的脊。与特征点相连的脊是通过沿着脊线方向以脊间距采样来表示。

（4）比对特征点。

在指纹录入时，即使是同一个手指，两次录入的指纹图像也不完全相同，会产生各种变形，比如平移、旋转等。要进行有效的匹配必须尽量减小各种变形，考虑到指纹的各种非线性变形通常是放射性的，可以在极坐标系中进行指纹匹配。另外由于非线性变形的存在，很难找到与指纹模板中特征点位置完全一致的特征点，因此匹配的算法应该是弹性的，即允许在某个范围内由于非线性变形引起的误差。

指纹特征匹配采用允许框来实现弹性，允许框是一个在特征点周围的框，如图 14-6 所示。

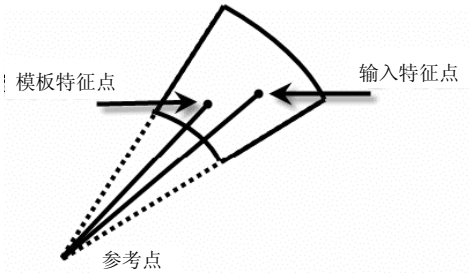


图 14-6　匹配允许框

14.2.2　设备驱动编写框架

在嵌入式 Linux 操作系统内核中提供了驱动程序的框架，在嵌入式开发指纹驱动程序时，首先根据 FPS200 芯片要实现的功能，编写 FPS200 的驱动。然后把 FPS200 硬件驱动程序嵌入 Linux 中。

设备驱动的框架如下：

```
#define MODULE
#include <Linux/module.h>
#include <asm/system.h>                                    // 汇编头文件
#include <asm/io.h>
#include <asm/irq.h>
#include <asm/uaccess.h>
```

```
#include <asm/bitops.h>
#include <asm/hardware.h>
#include <asm/hardware/clps.h>
int init_module( void )
{
                                                        // 使能
    clps_writel(0x01, PDDR);
    return 0;
}
void cleanup_module( void )
{
                                                        // 禁止
    clps_writel(~0x01,PDDR);
}
```

在 Linux 内核使用的设备文件，需要建立基本的/dev 目录下的结构，具体的方法如下。

```
#cd /arm/armroot
#vi mkdev                          //创建一个 shell 脚本，下面是文件的内容
mkdir dev
cd dev
mknod watchdog c 10 130            //看门狗
mknod zero c 1 5
mknod full c 1 7
mknod kmem c 1 2
mknod mem c 1 1                    //内存
mknod null c 1 3
mknod rtc c 10 135
mknod mtd0 c 90 0                  //Memory Technology Device,MTD 设备
mknod mtd1 c 90 1
mknod mtd2 c 90 2
mknod mtd3 c 90 3
mknod mtdblock0 b 31 0             //MTD 块设备
mknod mtdblock1 b 31 1
mknod mtdblock2 b 31 2
mknod mtdblock3 b 31 3
chmod 600 mtd*                     //修改权限
mknod cuam0 c 205 16
mknod cuam1 c 205 17
mknod ttyp0 c 3 0                  //终端设备
mknod ttyp1 c 3 1
mknod ttyp2 c 3 2
mknod ttyp3 c 3 3
mknod ttyp4 c 3 4
mknod ttyp5 c 3 5
mknod ttyp6 c 3 6
mknod ttyp7 c 3 7
mknod ttyp8 c 3 8
mknod ttyp9 c 3 9
mknod ttyAM0 c 204 16
mknod ttyAM1 c 204 17
ln -s ttyAM0 ttyS0                 //链接
ln -s ttyAM1 ttyS1
mknod tty c 5 0
mknod console c 5 1                //控制台
mknod ptyp0 c 2 0
```

```
mknod ptyp1 c 2 1
mknod ptyp2 c 2 2
mknod ptyp3 c 2 3
mknod ptyp4 c 2 4
mknod ptyp5 c 2 5
mknod ptyp6 c 2 6
mknod ptyp7 c 2 7
mknod ptyp8 c 2 8
mknod ptyp9 c 2 9
mknod ram0 b 1 0                                          //ram 设备
mknod ram1 b 1 1
mknod ram2 b 1 2
mknod ram3 b 1 3
chmod 660 ram*
```

将上面的所有内容保存之后修改文件属性为可执行。

```
#chmod +x mkdev
```

然后执行该文件：

```
#./mkdev
```

这样就在/arm/armroot/下面创建了一个 dev 目录，这个目录下面保存了 ARM Linux 内核所需要的基本的设备文件名称。

接下来创建 Linux 运行所需要的目录：

```
#mkdir tmp usr var proc
#mkdir var/log var/run
```

最后一步就是创建/etc 目录下的文件：

```
#cd /arm/src
#wget http://lmmasson.online.fr/dl/gz/etc.tar.gz
#tar vzxf etc.tar.gz -C /arm/armroot
#chmod u+x inittab rc
```

根据需要改 inittab 和 rc 这两个文件的内容。

14.2.3 指纹芯片驱动

FPS200 芯片的功能采集，它的工作方式是用户把手指放到采集板上之后，采集板产生一个硬件中断通知 ARM，此时用户程序可以通过读取中断标准位的方式得到该响应，然后用户程序通过 ioctl 发出控制指令读取指纹数据，如图 14-7 所示。

为了能够使用 FPS200 驱动，还需要在/dev 目录下创建一个设备文件，创建方法如下。

```
#cd /arm/armroot/dev
#mknod fps200 c 240 0
```

上面的命令表示，在 dev 目录下面创建了一个名字为 FPS200 的字符设备，该设备的主设备号是 240，次设备号是 0。

在编写驱动的时候，考虑到 FPS200 采集数据的时候需要调整参数，所以在设计时将调整参数的接口也提供给用户的程序，为了比较出哪一组参数所采集的图片最好，所以采用自动调整参数的方法，把每一个参数下的指纹图片自动保存，最后筛选出一个效果最好的参数作为最终参数。

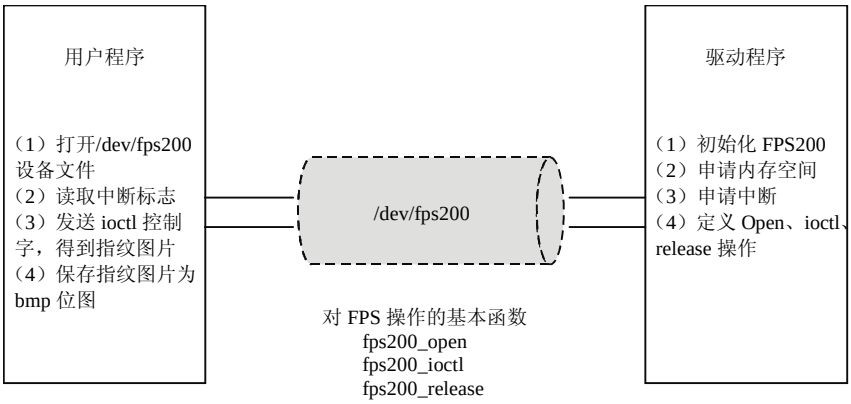


图 14-7　FPS200 驱动框图

FPS200 硬件驱动程序正确结果应该是：

（1）运行程序以后串口输出 ID：2022（对于 veridicom 公司的芯片），其他公司 20××。

（2）输出 reg ok 表示指纹芯片寄存器测试无误。

（3）从串口发送一个字符，程序将以 Hex 形式从串口输出一次采集的指纹数据，再从串口发送一个字符采集继续进行。将所有接收到的数据存在 I2C.txt 文件中，并放到 CtoHex.exe 所在的目录，运行 CtoHex.exe 程序，产生 finger1.dat 文件，用 PFProcess.exe 程序查看 finger1.dat 所存储的指纹图像。

14.3　实例——基于 ARM Linux 的指纹识别门禁系统

指纹采集芯片采用电容式传感器芯片 FPS200。FPS200 电容式传感器在 1.28cm×1.50cm 见方的表面集成了 256×300 个电容器。它提供与 8 位微处理器相连的接口，并且内置 8 位高速 A/D 转换器，可直接输出 8 位灰度图像。传感器采用标准 CMOS 技术，获取的图像大小为 256×300，分辨率为 500DPI（点每英寸）。

14.3.1　系统基本结构

1．FPS200 的内部结构

FPS200 传感器的每一列都有两个采样—保持电路，一个用来存储放电前电容两端的电压，另一个用来存储放电后电容两端的电压。两个采样—保持电路的差值可以度量电容的变化。先指定行高阶地址寄存器（RAH）和行低阶地址寄存器（RAL）中的数据以指定待读取的行，再指定列地址寄存器（CAL）从而启动行捕获，等待一段时间（行捕获时间）后，连续读取控制寄存器（CTRLA）、获得某一点的指纹采样值，读完会自动触发下一次 A/D 转换，读完一行后再写入 RAH、RAL 以读取下一行，直至最后一个像素。

在参数设置方面，其中 PGC 是放大倍数，通过它不能消除汗渍（模糊）；DCR 和 DTR 的组合大小和背景色以及双指纹现象。

- DCR：越小 DTR 要越大，DTR 可变范围越大（可变范围指图像不太黑，同时没有双指纹）。
- PGC：越大 DTR 可调范围变小，PGC 太小时整个图像将变成灰色，很难区分指纹和背景。

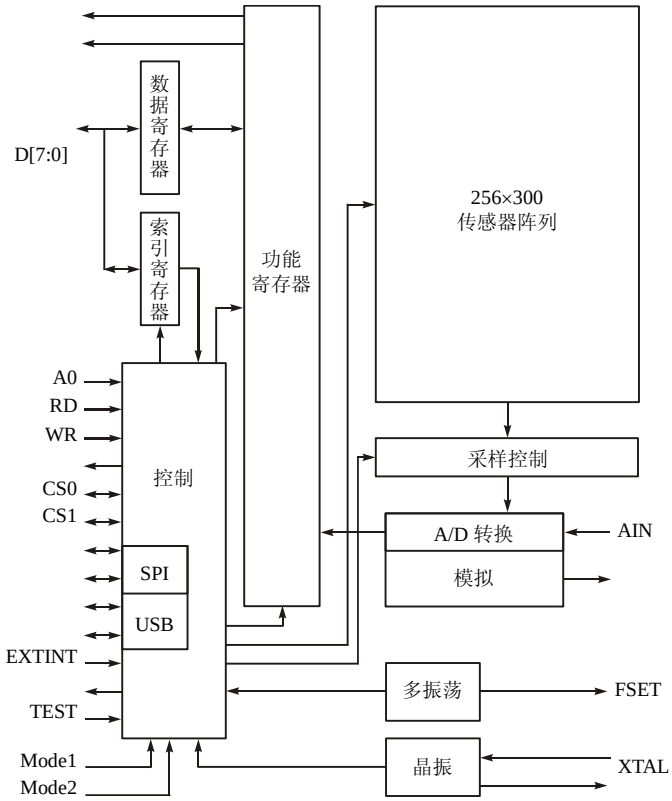


图 14-8　FPS200 内部控制逻辑

- DCR：越大将有效抑制汗渍（模糊）。但是 DCR 达到最大时背景为灰色，虽然指纹没有模糊。用减小 PGC 来看，此时整个图像为灰色。而 PGC 很大时调整范围小，图像不是很好。

2．系统硬件电路结构图

设计指纹采集模块的时候由于考虑到指纹芯片价格昂贵，万一设计存在错误将会造成很大的浪费，同时也考虑到以后在制作产品的时候指纹采集模块安装方便，所以将指纹模块单独设计成一块电路板连接在扩展板上面。

FPS200 接口电路如图 14-9 所示。

FPS200 和 ARM CPU 接线（布线要求）建议如下：

（1）为了减少干扰，ARM 输出到 FPS、ARM 的 RD/WR 输出到 FPS，这些数据线段中间不能再接其他线，即其他器件的数据线和 RD 线不能与 FPS 共用。

（2）这些线周围 0.5cm 左右不能有敷铜或者其他导线，这样来减少数据线和地之间地电容。

（3）其中的数据线应并排走，长度相等。这些线应尽量短，导线较粗。

（4）驱动输入引脚和地之间接一个 30pF 的电容，尽量贴近管脚。

（5）FSET 引脚和指纹自动探测有关。FSET 引脚的干扰将触发指纹采集。所以“FSET 引脚”和“接在 FSET 引脚上的电阻”之间的引线要尽量短，并且引线和周围引线之间有较大距离。

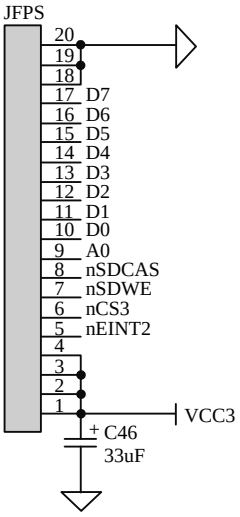


图 14-9　FPS200 接口电路

14.3.2　系统工作流程

指纹处理过程，由一个光电检测信号来确认是否有手指按下，如果有手指按下则此时直接读指纹芯片的缓冲区，将读到的指纹图像数据在 SRAM 中进行打包，然后发送到服务器，由服务器将接收到的指纹图像进行处理和辨识，然后向终端返回认证结果，由终端进行相应的显示和控制。系统的简单工作流程如图 14-10 所示。

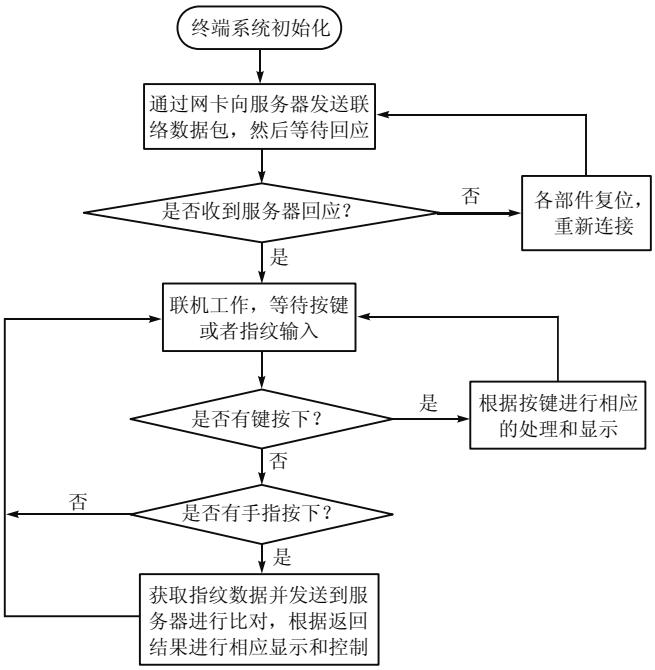


图 14-10　终端简单工作流程

14.3.3 系统模块源代码实现

1. 主机串口控制程序设计

在嵌入式 Linux 操作系统下，系统提供了专门的串口访问模块，用户只需要根据自身嵌入式硬件设备作适当裁剪即可。主要包括 Makefile 文件的编写、主机串口数据读取信号 Hostserial.c。

```
KERNELDIR=/usr/src/linux-2.4.20-8
#CC = arm-linux-gcc
CC = gcc
CFLAGS = -I$(KERNELDIR)/include/ -Wall
host_serial:
    $(CC) $(CFLAGS) -o host_serial serial.c host_serial.c
clean:
    rm -f host_serial
```

2. 指纹采集与处理程序

以下是指纹芯片采集程序，电路板上需要 RAM 和 GAL 器件。

```
#include "GloblDef.h"
#include "fps200.h"
#define START_ROW          0               //采集图像的起始行
#define END_ROW            299             //采集图像的末行
#define ONCE_READ_ROW_NUM  50              //一次从串口输出多少行的指纹数据
extern void FPS200Initial();
extern void FPSRow(BYTE xdata *buf,WORD RowI);
extern void FPSImg(BYTE xdata *buf);
extern void FPSSub(BYTE data *buf,WORD RowStart,WORD RowEnd,BYTE ColStart,
BYTE ColEnd);
extern WORD FPSReadID();
extern BYTE FPSMode();
extern BYTE FPSRegTest();
extern BYTE FPS200FingerPressed();
BYTE xdata FpsBuf[ONCE_READ_ROW_NUM * 256];

// 采集的指纹数据暂存的地方，一次采集 ONEC_READ_ROW_NUM 行，每行 256B
void SerialInitial()
{
    ......
}
main()
{
    WORD j,k,StartRow;
    BYTE value;
    SerialInitial();
                                           // 输出芯片 ID
    printf("\nID:%x",FPSReadID());
                                           //寄存器测试
    if(FPSRegTest() == TRUE)
        printf("\n reg ok");
    else
        printf("\n reg fail");
            //芯片初始化。必须在寄存器测试以后，因为寄存器测试会改变寄存器的设置
```

```
    FPS200Initial();
            // 开始采集指纹数据
            // 等待用户从串口发送一个字符以后才开始采集和输出
    scanf("%c",&value);
            // 采集开始
    for(j = START_ROW,StartRow = START_ROW; j <= END_ROW;)
    {
        FPSRow(FpsBuf + COL_NUM * (j - StartRow),j);
        j++;
        if(j - StartRow == ONCE_READ_ROW_NUM)
        {
            for(k = 0;k<ONCE_READ_ROW_NUM*256;k++)
            {
                printf("%c",FpsBuf[k]);
            }
            StartRow = j;
            scanf("%c",&value);// 等待用户从串口发送一个字符以后，开始下一次采集
        }
    }
    while(1);
}
```

FPS200 指纹处理程序如下：

```
// fps200.h 头文件
#ifndef _FPS200_H_
#define _FPS200_H_
#define ROW_NUM 300
#define COL_NUM 256
#define FPS200_IOCRESET _IO(FPS200_IOC_MAGIC)
#define FPS_RAH 0x00
#define FPS_RAL 0x01
#define FPS_CAL 0x02
#define FPS_REH 0x03
#define FPS_REL 0x04
#define FPS_CEL 0x05
#define FPS_DTR 0x06
#define FPS_DTR_TIME 0x70
#define FPS_DCR 0x07
#define FPS_DCR_CURRENT 0x6
#define FPS_CTRLA 0x08
#define FPS_CTRL_ASM_ARCH_HA_GETSUB 0x04
#define FPS_CTRLA_GETIMG 0x02
#define FPS_CTRLA_GETROW 0x01
#define FPS_CTRLA_AINSEL 0x08
#define FPS_CTRLB 0x09
#define FPS_CTRLB_MODE 0xC0
#define FPS_CTRLB_RDY 0x20
#define FPS_CTRLB_AFDEN 0x08
#define FPS_CTRLB_AUTOINCEN 0x04
#define FPS_CTRLB_XTALSEL 0x02
#define FPS_CTRLB_ENABLE 0x01
#define FPS_CTRLC 0x0A
#define FPS_SRA_ASM_ARCH_H 0x0B
#define FPS_SRA_GETSUB 0x04
#define FPS_SRA_GETIMG 0x02
#define FPS_SRA_GETROW 0x01
```

```
#define FPS_PGC 0x0C
#define FPS_PGC_VALUE 0x4//0xb
#define FPS_ICR 0x0D
#define FPS_ICR_IP1_RISE 0x80
#define FPS_ICR_IP0_RISE 0x40
#define FPS_ICR_IT1_LEVEL 0x20
#define FPS_ICR_IT0_LEVEL 0x10
#define FPS_ICR_IM1 0x08
#define FPS_ICR_IM0 0x04
#define FPS_ICR_IE1 0x02
#define FPS_ICR_IE0 0x01
#define FPS_ISR 0x0E
#define FPS_ISR_CLRINT 0x01
#define FPS_THR 0x0F
#define FPS_THR_THV 0x40
#define FPS_THR_THC 0x09
#define FPS_CIDH 0x10
#define FPS_CIDL 0x11
#define FPS_TST 0x12
#include <Linux/ioctl.h>
#undef PDEBUG
#ifdef fps200_DEBUG
# ifdef __KERNEL__
                                                    // 调试的内核空间
# define PDEBUG(fmt, args...) printk( KERN_DEBUG"fps200: " fmt, ## args)
# else
                                                    // 用户空间
# define PDEBUG(fmt, args...) fprintf(stderr, fmt, ##args)
# endif
#else
# define PDEBUG(fmt, args...)                       // 不调试
#endif
#undef PDEBUGG
#define PDEBUGG(fmt, args...)                       // 不调试
                                                    // 设备结构类型
typedef struct FPS200_Dev {
    unsigned char flag;
    void *data;
} FPS200_Dev;
                                                      // 用于 ioctl
#define FPS200_IOC_MAGIC 'k'

/*
* S 代表 "Set" through a ptr
* G 代表  "Get": reply by setting through a pointer
* C 代表  "Check"

*/
#define FPS200_IOCSDTR _IOC(_IOC_WRITE, FPS200_IOC_MAGIC, 1, 1)
#define FPS200_IOCSDCR _IOC(_IOC_WRITE, FPS200_IOC_MAGIC, 2, 1)
#define FPS200_IOCSPGC _IOC(_IOC_WRITE, FPS200_IOC_MAGIC, 3, 1)
#define FPS200_IOCGDTR _IOC(_IOC_READ, FPS200_IOC_MAGIC, 4, 1)
#define FPS200_IOCGDCR _IOC(_IOC_READ, FPS200_IOC_MAGIC, 5, 1)
#define FPS200_IOCGPGC _IOC(_IOC_READ, FPS200_IOC_MAGIC, 6, 1)
#define FPS200_IOCFCAP _IOC(_IOC_READ, FPS200_IOC_MAGIC,7, 4)
#define FPS200_IOCGDATA _IOC(_IOC_READ, FPS200_IOC_MAGIC, 8, 4)
```

```
#define FPS200_IOCEINT _IOC(_IOC_NONE, FPS200_IOC_MAGIC, 9, 0)
#define FPS200_IOCDINT _IOC(_IOC_NONE, FPS200_IOC_MAGIC, 10, 0)
#define FPS200_IOCCINT _IOC(_IOC_READ, FPS200_IOC_MAGIC, 11, 1)
#define FPS200_IOCCRDY _IOC(_IOC_READ, FPS200_IOC_MAGIC, 12, 1)
#define FPS200_IOCCLR _IOC(_IOC_NONE, FPS200_IOC_MAGIC, 13, 0)
#define FPS200_IOC_MAXNR 13
int fps200_open(struct inode *inode, struct file *filp);
int fps200_release(struct inode *inode, struct file *filp);
int fps200_ioctl (struct inode *inode, struct file *filp,
                  unsigned int cmd, unsigned long arg);
void fps_get_image();
#endif /* _FPS200_H_ */

//
fps200.c 文件
        //

#ifndef __KERNEL__
# define __KERNEL__
#endif
#ifndef MODULE
# define MODULE
#endif
#include <Linux/config.h>
#include <Linux/module.h>
#include <Linux/kernel.h>              // printk()
#include <Linux/slab.h>                // kmalloc()
#include <Linux/fs.h>
#include <Linux/errno.h>               // 错误代码
#include <Linux/types.h>               // size_t
#include <Linux/init.h>
#include <Linux/delay.h>               // udelay()
#include <asm/io.h>                    // ioremap(), iounmap()
#include <Linux/ioport.h>
#include <asm/irq.h>
#include <Linux/string.h>
#include <asm/uaccess.h>
#include <asm/arch/irqs.h>
#include "fps200.h"                    // 自定义
#define FPS200_VR 0xfd000000
#define FPS_INDEX (*(volatile unsigned char *)FPS200_VR)
#define FPS_DATA (*(volatile unsigned char *)(FPS200_VR+1))
#define FPS200_MAJOR 240
#define FPS200_NR_DEVS 0
#define FPS200_IRQ IRQ_EINT2           // irq = 6
#define FPS200_DATASIZE 76800
int fps200_major = FPS200_MAJOR;
int fps200_nr_devs = FPS200_NR_DEVS;
                                       // fps200 器件数目
MODULE_PARM(fps200_major,"i");
MODULE_PARM(fps200_nr_devs,"i");
MODULE_AUTHOR("Nankai Unversity 5-304");
MODULE_LICENSE("GPL");
struct file_operations fps200_fops =
{
    open: fps200_open,
```

```
    ioctl: fps200_ioctl,
    release: fps200_release
};
struct file_operations *fps200_fop_array[]=
{
    &fps200_fops,                                          // 类型 0
                                                           // 逐渐增加
};
#define FPS200_MAX_TYPE 0
FPS200_Dev *fps200_device;
void fps200_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    disable_irq(irq);
    fps_get_image();
    fps200_device->flag = 1;
}
void fps_get_image(void)
{
    int i = 0;
    int j = 0;
    FPS_INDEX = FPS_CTRLA;
    FPS_DATA = FPS_CTRLA_GETIMG;
    for(i=0; i<300; i++) {
        FPS_INDEX = FPS_CTRLB;
        while(!(FPS_CTRLB_RDY&FPS_DATA)){udelay(1);};
        for(j=0; j<256; j++) {
            FPS_INDEX = FPS_CTRLB;
            while(!(FPS_CTRLB_RDY&FPS_DATA)){udelay(1);};
            FPS_INDEX = FPS_CTRLA;
            *((unsigned char *)(fps200_device->data+i*256+j))=FPS_DATA;
        }
    }
}
int fps200_open(struct inode *inode, struct file *filp)
{
    MOD_INC_USE_COUNT;
    return(0);
}
int fps200_release(struct inode *inode, struct file *filp)
{
    MOD_DEC_USE_COUNT;
    return(0);
}
int fps200_ioctl(struct inode *inode, struct file *filp,
unsigned int cmd, unsigned long arg)
{
    int err = 0;
    int ret = 0;
    unsigned char tmp;
    if(_IOC_TYPE(cmd) != FPS200_IOC_MAGIC)
    return -ENOTTY;
    if(_IOC_NR(cmd) > FPS200_IOC_MAXNR)
    return -ENOTTY;
    if (_IOC_DIR(cmd) & _IOC_READ)
    err = verify_area(VERIFY_WRITE, (void *)arg,
    _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
```

```
err = verify_area(VERIFY_READ, (void *)arg,
_IOC_SIZE(cmd));
if (err)
return err;
switch(cmd)                                          //分支处理
{
case FPS200_IOCSDTR:
    ret = __get_user(tmp, (unsigned char *)arg);
    if(tmp > 0x7f)
    tmp = 0x7f;
    FPS_INDEX = FPS_DTR;
    FPS_DATA = tmp;
    break;
case FPS200_IOCSDCR:
    ret = __get_user(tmp, (unsigned char *)arg);
    if(tmp > 0x1f)
    tmp = 0x1f;
    FPS_INDEX = FPS_DCR;
    FPS_DATA = tmp;
    break;
case FPS200_IOCSPGC:
    ret = __get_user(tmp, (unsigned char *)arg);
    if(tmp > 0x0f)
    tmp = 0x0f;
    FPS_INDEX = FPS_PGC;
    FPS_DATA = tmp;
    break;
case FPS200_IOCGDTR:
    FPS_INDEX = FPS_DTR;
    tmp = FPS_DATA;
    ret = __put_user(tmp, (unsigned char *)arg);
    break;
case FPS200_IOCGDCR:
    FPS_INDEX = FPS_DCR;
    tmp = FPS_DATA;
    ret = __put_user(tmp, (unsigned char *)arg);
    break;
case FPS200_IOCGPGC:
    FPS_INDEX = FPS_PGC;
    tmp = FPS_DATA;
    ret = __put_user(tmp, (unsigned char *)arg);
    break;
case FPS200_IOCEINT:
    enable_irq(FPS200_IRQ);
    break;
case FPS200_IOCDINT:
    disable_irq(FPS200_IRQ);
    break;
case FPS200_IOCFCAP:
    fps_get_image();
case FPS200_IOCGDATA:
    copy_to_user((void *)arg, fps200_device->data,
    FPS200_DATASIZE);
    ret = 0;
    fps200_device->flag = 0;
    break;
case FPS200_IOCCLR:
```

```
            memset(fps200_device->data, 0, FPS200_DATASIZE);
            fps200_device->flag = 0;
            break;
        case FPS200_IOCCINT:
            if(((clps_readw(INTSR1))&0x40) == 0)
            {
            udelay(100);
            if(((clps_readw(INTSR1))&0x40) == 0)
            ret = __put_user(0x01, (unsigned char *)arg);
            else
            ret = __put_user(0x0, (unsigned char *)arg);
        }
            else
            ret = __put_user(0x0, (unsigned char *)arg);
            break;
        case FPS200_IOCCRDY:
            ret = __put_user(fps200_device->flag, (unsigned char*)arg);
            break;
        default:
        return -ENOTTY;
        }
        return ret;
    }
    static int __init fps200_init_module(void)
    {
        int result;
        char tmp;
        if((result = check_region (FPS200_VR,2)))
        {
        printk ("<1> can't get I/O port address \n");
        return (result);
        }
        if (!request_region (FPS200_VR,2,"fps200"))
        return -EBUSY;
        SET_MODULE_OWNER(&fps200_fops);
        result = register_chrdev(fps200_major, "fps200",&fps200_fops);
        if(result < 0)
        {
        printk("<1>fps200: can't get major %d\n",fps200_major);
        return result;
        }
        if(fps200_major == 0)
        fps200_major = result;          // 动态
                                        // 读芯片 id，如果不等于 0x20xx，报错

        FPS_INDEX = FPS_CIDH;
        tmp = FPS_DATA;
        if(tmp != 0x20)
        {
        printk("<1>wrong chip ID, insmod fail.\n");
        return -EIO;
        }
                                                        // 内部 12MHz 振荡

        FPS_INDEX = FPS_CTRLB;
        FPS_DATA = (FPS_CTRLB_AFDEN|FPS_CTRLB_AUTOINCEN|FPS_CTRLB_ENABLE);
```

```
                                                    // 等待 30μs
udelay(35);                                         //使时延大于 30μs
                                                    // 中断
FPS_INDEX = FPS_ICR;
FPS_DATA = (FPS_ICR_IE0|FPS_ICR_IT0_LEVEL);
FPS_INDEX = FPS_THR;
FPS_DATA = ( FPS_THR_THV | FPS_THR_THC );
                                                    // DTR, DCR, PGC
FPS_INDEX = FPS_DTR;
FPS_DATA = 0x23;
FPS_INDEX = FPS_DCR;
FPS_DATA = 0x1;
FPS_INDEX = FPS_PGC;
FPS_DATA = 0;
                                                    // 其他的初始化
FPS_INDEX = FPS_RAL;                                // raw 地址
FPS_DATA = 0;
FPS_INDEX = FPS_RAH;
FPS_DATA = 0;
FPS_INDEX = FPS_REL;
FPS_DATA = 0;
FPS_INDEX = FPS_REH;
FPS_DATA = 0;
FPS_INDEX = FPS_CAL;                                // column 地址
FPS_DATA = 0;
FPS_INDEX = FPS_CEL;
FPS_DATA = 0;
FPS_INDEX = FPS_CTRLC;
FPS_DATA = 0;
FPS_INDEX = FPS_CTRLA;
FPS_DATA = 0;// clear FPS_CTRLA_AINSEL
                                                    // 设置 irq
if(result)
{
printk("<1>can't get assigned irq.\n");
return -EIO;
}
fps200_device = kmalloc(sizeof(FPS200_Dev),
GFP_KERNEL);
if(!fps200_device)
{
FPS_INDEX = FPS_CTRLB;
FPS_DATA = 0;
return -ENOMEM;
}
memset(fps200_device, 0, sizeof(FPS200_Dev));
fps200_device->data = kmalloc(FPS200_DATASIZE,
GFP_KERNEL);
if(!fps200_device)
{
FPS_INDEX = FPS_CTRLB;
FPS_DATA = 0;
kfree(fps200_device);
return -ENOMEM;
}
```

```
        memset(fps200_device->data, 0, FPS200_DATASIZE);
                                                        // 设置 irq
        result = request_irq(FPS200_IRQ, fps200_interrupt,
        SA_INTERRUPT, "fps200", NULL);
        return(0);
    }
    static void __exit fps200_cleanup_module(void)
    {
        kfree(fps200_device->data);
        kfree(fps200_device);
        FPS_INDEX = FPS_CTRLB;
        FPS_DATA = 0;
        release_region (FPS200_VR,2);
        free_irq(FPS200_IRQ, NULL);
        unregister_chrdev(fps200_major, "fps200");
    }
    module_init(fps200_init_module);
    module_exit(fps200_cleanup_module);
```

3．服务器端的程序

服务器端的程序主要完成指纹比对和对终端、用户的管理以及考勤等功能，由以下 5 部分程序组成。

- QDServer：服务器监听程序。它的功能是与终端通信、记录事件到数据库、指纹比对、接收用户留言。程序模块主要分为网络部分、数据库部分和界面部分。其中网络部分能够同时和多个终端保持连接，记录终端的事件，查询数据库。用户通过界面了解当前的终端的状态和接收用户控制。
- QDMenage：服务器管理程序。功能是管理数据库。
- QDMessage：用户留言功能。
- QDInstallDB：安装或者卸载服务器程序时注册数据库。
- ConfigGen：产生配置文件。

（1）指纹比对算法链接库的调用

在指纹比对程序源代码中，对链接库的调用是基于对其中指纹比对算法函数的调用，由于知识产权的关系，下列程序源代码中将文件名用*号来代替。

```
    //FingerPrint.cpp:头文件包含及相关变量定义；
    CFingerPrint::CFingerPrint()    指纹比对处理函数名
    {
    #ifndef MY_LOAD

    //加载链接库
    fpfltr5 = LoadLibrary("******.dll");
    minulib = LoadLibrary("******.dll");
    vmatcher = LoadLibrary("******.dll");
    //加载函数
    func_fpProcess = (typefpProcess)GetProcAddress(fpfltr5,"fpProcess");
    func_sizeofTemplate=(typesizeofTemplate)GetProcAddress(minulib,
"sizeofTemplate");
    func_matchprints = (typematchprints)GetProcAddress(vmatcher,"matchprints");
    func_multiEnroll = (typemultiEnroll)GetProcAddress(vmatcher,"multiEnroll");
```

在这里需要将调用的语句中的文件名和函数名换为与指纹模块内的算法相对应的SDK 软件包内的相应链接库文件名和函数名。然后将程序中的函数名及传递参数都做相应的修改。实际上在改进后，用不到这么多的函数，因为提供的 SDK 软件包内的算法链接库文件中的函数及其传递参数都比较少，这是由于接收到终端传递的指纹数据已经由终端的 DSP 指纹模块提取了指纹图像的指纹特征值，所以在比对时就要简单得多。

（2）QDServer 网络部分

在应用层主要是通过网络控制命令对终端进行控制，服务器记录当前的状态，根据状态和事件的组合来判断，采取动作。应用层协议由 CacceptScock 负责处理。Caccepect 通过前缀为 Conn 的函数调用 CQDView 类的指纹识别和数据库模块。以下是具体的命令类型。

```
// 所有数据包的类型，也就是数据包命令字段类型
#define C_IDLE          0   // 空闲数据包，表示终端没有断线
#define C_FINGER_DATA    1   // 终端发送指纹数据给服务器
#define C_SEND_KEY       2   // 终端发送一个功能键给服务器
#define C_SEND_SETTING   3   // 双发发送的设置包
#define C_TERMINAL_OPENED 4   // 终端发给服务器，表示终端被打开
#define C_CLEAR_ALARM    5   // 服务器清除终端报警状态，终端清除以后发送此包给予确认
#define C_MESSAGE        6   // 服务器发送信息到终端
#define C_OPEN_DOOR      7   // 服务器要求终端开门
#define C_SEND_USER_INPUT 8   // 终端发送用户输入到服务器
```

（3）提示信息的显示和用户留言功能

```
CString tip;        // 在终端的第四行显示的提示语
CString m_strUserMessage; // 给用户的留言信息
intm_iUserMessage;  // 下一屏需要显示的用户留言信息的字符串 m_strUserMessage 中的开始位置
```

- 提示信息

当 SendMessage 时，如果是 SINITIAL 则看终端提示语是否为空，如果不为空则将提示语发送到终端。在 CQDView 的 ConnUpdateSetting 中初始化连接的提示语。

在监听程序中，增加刷新功能。在刷新过程中，刷新终端数据库，读取终端提示语，然后发送到终端。

- 用户留言提示信息

监听程序：用户可以开门，如果用户有留言信息，则将留言写入这个 AcceptSocket 的 m_strUserMessage 中，并且设置信息字符串的位置 m_iUserMessage 为 0，在 View 类中其他的处理相同。当返回到 AcceptSocket 中时，发现是 S_OPENDOOR 处理开门，然后需要看一下 m_UserMessage 是否为空。如果不为空则将状态从 S_OPENDOOR 转化到 S_READ_MESSAGE。

在 SendMessageToClient 中如果是 S_READ_MESSAGE，则将 m_UserMessage 读入，发送到客户端，增加 m_iUserMessage。

如果按下一页命令则发送下一页。如果按下“退出”，则转化到 S_READ_MESSAGE_DEL“是否删除这条信息？”，将 m_strUserMessgae 清空，然后返回 S_INITIAL。

以上部分是网络部分工作的部分情况简介，在这一部分中，只需要在终端将指纹数据包结构中的指纹图像数据部分的来源换为从串口读进的指纹特征值数据就可以了，同时在

定义中把包的大小根据指纹特征值的数据量大小重新进行定义。语句为：

```
#define SMALL_PAGE_SIZE 128     // 大于普通包的大小
#define LARGE_PAGE_SIZE (256*7)  // 大于网络中最大包的大小
```

将以上的语句中的页大小设置为 128 就可以了，也就是说（256*7）可以修改为 128，因为指纹特征值的数据包最大不超过 128B。

由以上分析可以看出，其工作过程是串口读到数据包根据命令字段判断数据包的类型是否为指纹特征值，然后由 TCP 部分进行封包处理发送给服务器，服务器再将数据解包，状态和标志在这里不用做修改，因此服务器监听程序会认为从该包中获取的数据是指纹数据，则会将获取的数据用指纹算法进行比对，而这时指纹算法链接库文件已经修改为与指纹模块内置算法一致的链接库文件，因此不会发生错误。产生比对结果后，服务器监听程序应将结果再封包返回给终端。在网络部分的所有状态都不用做任何修改。

14.4 本章总结

本章详细讲述了基于 ARM Linux 的指纹识别门禁应用系统。指纹识别算法需要大量的运算，从 PC 上运行的结果来看，由于放到 ARM 硬件平台上运行速度较慢，因此可以在系统硬件上增加一块 DSP 数字处理芯片作为专门的指纹处理模块。如果指纹识别门禁功能再配以必要的软件，便可用于如因特网、计算机、安全等很多领域。随着人们对加密技术要求的不断提高，基于嵌入式 Linux 实现的指纹识别系统和产品将会有很好的市场前景。

第 15 章

基于 RTL8019 的以太网应用系统开发实例

在电子设备日趋网络化的背景下，作为目前广泛使用的以太网及 TCP/ IP 已经成为事实上最常用的网络标准之一，它的高速、可靠、分层以及可扩充性使其在各个领域的应用越来越灵活，很多情况下运用以太网和 TCP/ IP，能够简化结构和降低成本。

15.1 以太网应用技术概述

以太网接口模块是构造一个通用的基于网络的嵌入式 Linux 系统的基础，该接口模块的主要任务就是完成与外界的信息交互，以达到网络监控的目的。基于嵌入式 Linux 的系统若没有以太网接口，其应用价值就会大打折扣，因此，就整个系统而言，以太网接口电路应是必不可少的，但同时也是相对较复杂的。在实际应用中，它运行稳定，能够十分方便地实现嵌入式系统的网络互联。在系统采用高性能的以太网控制器，系统通信和调试快速可靠，具有很高的实时性。

如图 15-1 所示是用 RTL8019 以太网接口在嵌入式处理器上实现 TCP/IP 协议来实现 Internet 接入功能的一种简单方案。

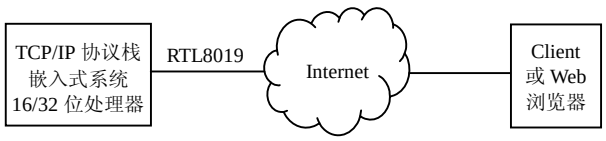


图 15-1 基于 RTL8019 的嵌入式应用环境

如图 15-2 所示的 ITU.T 802-3 规则中，TCP/IP 模型共分为 4 层：应用层、传输层（TCP 层）、网间网层（IP 层）、网络接口层（链路 MAC 层和物理 PHY 层）。

基于嵌入式 Linux 的 TCP/IP 协议栈由于运行硬件的限制，要求软件简洁、占用资源少。嵌入式 Linux 精简了 TCP/IP 协议栈的 IP options/UDP/Sliding TCP window 等功能，占用资源少，并能够实现出入式系统上网的要求。TCP/IP 协议组的四个基本协议实现为 ARP/IP/ICM P 和 TCP。链路层协议以 RTL8019 驱动程序实现；应用层协议用 HTTP 实现 IP 之上的应用程序。

从硬件的角度看，以太网接口电路主要由媒质接入控制 MAC 控制器和物理层接口（Physical Layer，PHY）两大部分构成，目前常见的以太网接口芯片如 RTL8019/8029/8039、CS8900、DM9008 以及 DWL650 无线网卡等，其内部结构也主要包含这两部分。

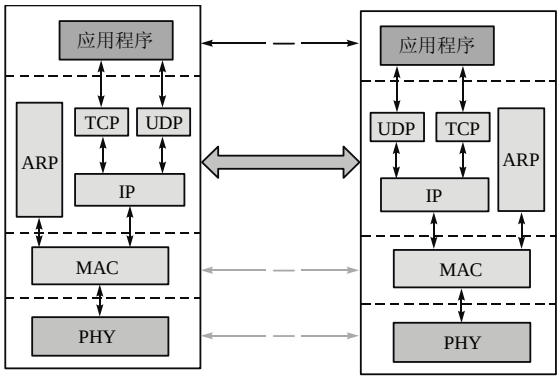


图 15-2　基于 802.3 的网络分层模型

15.2　相关开发技术

15.2.1　基于 RTL8019 的以太网帧传输原理

使用 RTL8019 作为以太网的物理层接口，它的基本工作原理是：在收到由主机发来的数据报后（从目的地址域到数据域），侦听网络线路。如果线路忙，它就等到线路空闲为止，否则，立即发送该数据帧。发送过程中，首先添加以太网帧头（如图 15-3 所示，包括先导字段和帧开始标志），然后生成 CRC 校验码，最后将此数据帧发送到以太网上。

在以太网的数据帧中，以太网包头为 14 字节，IP 包头为 20 字节，TCP 包头为 20 字节，有些以太网的最大帧长为 1514 字节。具体内容如图 15-3 所示，各个域中的字段的功能说明如表 15-1 所示。

表 15-1　以太网数据包各字段的说明

字 段 名	说　　明
前导码	共 7 字节，每一字节都是 10101010，从左向右传
帧起始标志	为 10101011，从左向右传
目的地址	目标地址既可以是单播地址，也可以是多播或广播地址
源地址	为了地址有效，本字段第一字节的 0 位必须为 0（即源地址不能为广播或多播地址）
长度/类型	大于 1500 的字段数据当做类型域，小于等于 1500 字段数据代表数据域的逻辑链路控制
有效数据	有效数据
填充	如果有效数据比 46 字节短，MAC 将传送全 0 的填充数据
帧校验和	本字段包含 16 位的错误检测码

在接收过程，它将从以太网收到的数据帧在经过解码、去帧头和地址检验等步骤后缓存在片内。在 CRC 校验通过后，它会根据初始化配置情况，通知 RTL8019 收到了数据帧，最后，用某种传输模式（I/O 模式、Memory 模式、DMA 模式）传到 ARM 的存储区中。

大多数嵌入式系统内嵌一个以太网控制器，支持媒体独立接口（Media Independent Interface，MII）和带缓冲 DMA 接口（Buffered DMA Interface，BDI）。可在半双工或全双工模式下提供 10M/100Mbps 的以太网接入。在半双工模式下，控制器支持 CSMA/CD 协议，在全双工模式下支持 IEEE 802.3 MAC 控制层协议。

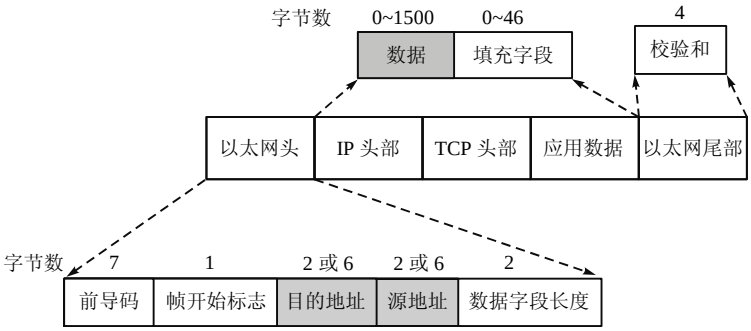


图 15-3　802.3 帧格式

由以上可以看出，系统实际上已包含了以太网 MAC 控制，但由于并未提供物理层接口，因此，需外接一片物理层芯片以提供以太网的接入通道。而常用的单口 10M/100Mbps 高速以太网物理层接口器件均提供 MII 接口和传统 7 线制网络接口，可方便地与 CPU 接口。以太网物理层接口器件主要功能一般包括物理编码子层、物理媒体附件、双绞线物理媒体子层、10BASE-TX 编码/解码器和双绞线媒体访问单元等。

15.2.2　RTL8019 的初始化

RTL8019 内部有 RAM，地址范围从 0x0000~0x7FFF，其中 0x4000~0x7FFF 是用于接收和发送缓冲区。

（1）读写 RAM

RTL8019 内部 RAM 是双口 RAM，因为它要支持两个独立的操作，一个是用户 CPU 读取 RAM 中的内容，对这个操作 RTL8019 提供一个读写口也就是寄存器中的 Remote DMA Port，另一个是 RTL8019 内部控制电路把从网络接收的数据写入 RAM 中，这时 RAM 称为 Loacal DMA。RTL8019 通过 Loacal DMA 写入 RAM 是不用用户干涉的，它通过 RemoateDMAPort 读写 RAM。

读 RAM 为 RTLReadRam 函数。这个函数表示从地址 address 开始读取 size 个字节的内容到 buff 指针指向的内存中。设置 CR 寄存器为：

```
WriteReg(CR,(0x00 | CR_REMOTE_WRITE | CR_START_COMMAND));
```

然后从 RemoteDMAPort 读取 size 次，就得到所需的数据。

写 RAM 使用 RTLWriteRam 函数，操作基本上和读 RAM 差不多，只要将设置 CR 寄存器语句改成：

```
WriteReg(CR,(0x00 | CR_REMOTE_READ | CR_START_COMMAND));
```

最后一步的读 size 次改成写 size 次就可以了。

（2）发送接收缓冲区

0x4000~0x7FFF 的接收和发送缓冲区，可以分为发送缓冲区和接收缓冲区。缓冲区是按页管理的，256 字节为一页，这样接收发送缓冲页面是从 0x40 到 0x7F。由于发送缓冲区的起始页在 TPSR 寄存器中设置，接收缓冲区的起始页在 PSTART 寄存器中设置，PSTART 实际上也表明了发送缓冲区的结束页，接收缓冲区的结束页是 PSTOP。所以发送缓冲区的页从 TPSR 到 PSTART–1，接收缓冲区的页从 PSTART 到 PSTOP–1。

接下来的问题是接收和发送缓冲各占多少才合适。这里设置如下：

```
#define RECEIVE_START_PAGE      0x4C
#define RECEIVE_STOP_PAGE       0x60
                /* 有些资料上为 80 但是实验发现 80 时存在发送和接收缓存冲突的问题 */
#define SEND_START_PAGE0        0x40
#define SEND_START_PAGE1        0x47
```

让发送缓冲区可以容纳下两个最大以太网帧（最大的以太网帧大小是 1514 字节），第一个帧放在 SEND_START_PAGE0 起始页，第二个帧放在 SEND_START_PAGE1 起始页，剩下的缓冲区都作为接收缓冲区。

（3）RTL8019 以太网口初始化

初始化第一步是复位以太网口。以太网口的复位分为硬件复位和软件复位。硬件复位通过给 RTL8019 的 RST 引脚一个脉冲来复位以太网口。软件复位通过写 ResetPort 达到复位，也就是给 18~1F 之间的任意一个寄存器写入任意一个数，就使得以太网口复位。第二步是设置一些寄存器的初始值。寄存器保存本机以太网的物理地址，只有与寄存器保存的物理地址相同的以太帧才被接收（如果 RCR 中 PRO=0）。

以太网口第一次复位必须是硬件复位，也就是说使用以太网口前一定要先硬件复位。另外硬件复位以后要经过大约 10ms 的等待才能对以太网口操作，特别是发送和接收操作。

15.2.3 RTL8019 驱动程序的框架

在 Linux 中，整个网络接口驱动程序的框架也可分为 4 层，从上到下分别为协议接口层、网络设备接口层、提供实际功能的设备驱动功能层，以及网络设备和网络媒介层。

这个框架在 Linux 内核网络模块中已经搭建好了，在设计网络驱动程序时，要做的主要工作就是根据上层网络设备接口层定义的 net_device 结构和底层具体的硬件特性，完成设备驱动的功能。在网络驱动程序部分主要有两个数据结构，一个是 sk_buff，TCP/IP 中不同协议层间，以及和网络驱动程序之间数据包的传递都是通过这个结构体来完成的，这个结构体主要包括传输层、网络层、连接层需要的变量，决定数据区位置和大小的指针，以及发送接收数据包所用到的具体设备信息等。它的详细定义可参阅内核源代码<include/linux/skbuff.h>。另一个就是 net_device 结构，它的定义在<include/linux/netdevice.h>中。这个结构是网络驱动程序的核心，它定义了很多供系统访问和协议层调用的设备标准的方法，包括供设备初始化和向系统注册用的 init 函数，打开和关闭网络设备的 open 和 stop 函数，处理数据包发送的函数 hard_start_xmit，中断处理函数，以及接口状态统计函数等。

以下是网卡驱动程序设计的框架和步骤。

```
//初始化函数
static int RTL8019_init(struct net_device *dev)
{
调用 ether_setup(dev)函数设置通用的以太网接口;
    填充 net_device 数据结构的属性字段;
    调用 kmalloc 申请需要的内存空间;
    手动设置 MAC 地址;
}

//设备打开与关闭函数
static int RTL8019_open(struct net_device *dev)
```

```
{
    关闭中断;
    注册中断号和 I/O 地址;
    初始化设备的寄存器;
    使能中断;
}
//设备关闭函数与打开函数的动作相反

//数据包发送函数
static int RTL8019_sendpacket(struct sk_buff *skb, struct net_device *dev)
{
    将标志位 tbusy 打开;
    将数据包写入 RTL8019 的发送缓冲区，启动
    DMA 发送功能;
    释放缓冲区;
}

//数据包接收函数
static int RTL8019_rx( int irq, void *dev_id, struct pt_regs *regs)
{
    申请 skb 缓存区存储新的数据包;
    从硬件中读取新到达的数据;
    调用函数 netif_rx()，将新的数据包向网络协议
    的上一层传送;
}
```

最后，将驱动程序编译进内核，加载模块，如果一切正常，使用 ifconfig、route add 命令设置 IP 地址和子网掩码，RTL8019 就能正常工作了。

15.2.4　数据结构和函数

Linux 中已经定义了为编写网络驱动程序必须使用的数据结构和函数。由于编写驱动程序的大部分工作就是填写这些数据结构并同时定义这些函数的入口点。因此必须分析清楚这些数据结构与函数。对网络设备的驱动程序来说最重要的数据结构有 struct device 和 struct sk_buff。

1．struct device

Linux 系统中的每一个网络界面都有相应的 device 结构与之对应。当驱动模块加载进系统时，驱动程序进行探测设备、资源请求等工作。这与字符设备和块设备驱动所做的工作基本是一样的，不同的地方在于网络设备驱动不像字符设备和块设备那样请求主设备号，而是在一个全局网络设备表里为每一个新探测到的网络界面插入一项 struct device 数据结构。

struct device 结构中较重要的域描述如下。

（1）char *name

设备名。如果名字的第 1 个字符是空字符或空格，注册程序自动为设备分配 eth *n* 名字。

（2）unsigned long rmem_end

```
unsigned long rmem_start
unsigned long mem_end
unsigned long mem_start
```

设备共享内存的起始和结束地址。当设备为发送和接收数据分别指定了不同的内存地址时，则 mem 域为发送内存而 rmem 域为接收内存。

（3）unsigned long base_addr

基本 I/O 地址，在设备探测的时候指定。ifconfig 命令可以对其进行修改。

（4）unsigned char irq

设备中断号。ifconfig 命令可以显示和修改该值，其值一般在系统启动或加载模块的时候指定。

（5）unsigned char start

```
unsigned char interrupt
```

二进制标志。start 一般在打开设备时设置，在关闭设备时清除。当其不为 0 时表示网络界面已经准备好。Interrupt 通知上层代码设备中断已到达并正在被服务。

（6）unsigned long tbusy

这个域用来表示“传输忙”。当设备驱动无法再接收新的发送数据包时（输出缓冲区已满）设置该位。

（7）unsigned char dev_addr[MAX_ADDR_LEN]

机器硬件地址。对于以太网来说，其长度为 6 个字节。

（8）unsigned long pa_addr

```
unsigned long pa_brdaddr
unsigned long pa-mask
```

分别是网络界面的地址、广播地址，以及子网掩码。如果 dev_family 是 AF_INET，那么这些都是 IP 地址，可用 ifconfig 命令设置。

2．struct sk_buff

当 kernel 准备发送数据包时，它调用 hard-start-transmit 函数将数据放入输出缓冲队列。每一待发送数据包都存放在 sk_buff 结构中。很多 sk_buff 结构的链接构成输入/输出缓冲区。缓冲区和缓冲队列的概念并不一样，缓冲区实际就是一个 struct skb_buff 的链表，而缓冲队列是指 RTL8019 上的硬件缓冲区。

struct skb_buff 在<linux/skbuff.h>中定义，其较重要的域描述如下。

（1）struct device *dev（与该缓冲区相关联的设备）

（2）_u32 saddr

```
u32 daddr
u32 raddr
```

IP 协议下的源地址、目的地址和路由地址。在数据发送之前必须设置好。

（3）unsigned char* head

```
unsigned char* data
unsigned char* tail
unsigned char* end
```

这 4 个指针用于定位包中的数据。head 指向已分配空间的起始位置；data 是有效数据的起始位置；tail 是有效数据的结束位置；end 指向 tail 能达到的最大地址。所以有效的缓冲区空间大小是 skb→end-skb→head。而目前所使用的数据空间大小是 skb→tail-skb→data。

与字符设备和块设备的 open 和 close 函数一样，网络设备也定义了很多作用于其上的函数（device method）。编写这些函数的入口点是主要工作。

（4）int(*open)(struct device* dev)

```
int(*stop)(struct device* dev)
```

设备驱动在系统启动或在模块装载的时候探测到网络界面并利用初始化程序进行 IO 地址分配、中断线分配等工作，然后必须给网络界面分配 IP 地址，这项工作由 ifconfig 命令完成，一般在系统初始化时就执行了这条命令。ifconfig 执行两项任务：首先利用 ioctl(SIOCSIFAD-DR)给界面分配地址簇、地址和子网掩码。这一部分工作由 kernel 完成，与设备驱动无关。然后利用 ioctl（SIOC-SIFFLAGS）设置 dev→flag 位激活网络界面，此外，ioctl（SIOCSIFFLAGS）函数也调用 open 函数进行资源（中断号，IO 地址）请求工作。同样，在 ifconfig 命令关闭网络界面时，ioctl（SIOCSIFFLAGS）函数清除 dev→flags 中的 IFF_ UP 位并调用 stop 函数。

（5）int(*hard_start_ xmit)(struct sk_buff* skb，structdevice* dev)

在每次数据发送时都必须调用该函数将数据放入硬件输出队列，待发送的数据包含于 struct skb_ buff 中。其主要工作就是根据不同的网络设备硬件把 struct skb_ buff 中的数据包送入到 RTL8019 上的输出缓冲队列。

（6）int(*rebuild_header)(void* buf, struct device* dev，unsigned long raddr，struct sk_ buff* skb)

```
int(*hard_header)(struct sk_buff* skb，struct device* dev，unsigned short type，void* daddr, void* saddr，unsigned len)
```

数据帧生成的时候调用 hard_ header 以生成帧头。对于以太网来说，hard_ header 并不能生成目标硬件地址。这一工作需要由 rebuild_ header 利用 ARP 来完成。

（7）int(*set_config)(struct device* dev，struct ifmap*map)

用于改变网络的配置，配置信息存放在 ifmap 结构中。

（8）int(*do_ioctl)(struct device* dev，struct ifreq* ifr，int cmd)

用于执行一些应用于网络的 ioctl 命令。

Kernel 同时还定义了很多操作 struct sk_ buff 的函数，主要有：

```
struct sk_buff* alloc_skb(unsigned int len，int priority);
struct sk_buff* dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff* skb，int rw);
void dev_kfree_skb(struct sk_buff* skb，int rw)
```

分配和释放 struct sk_ buff 缓冲区。alloc_skb 和 dev_ alloc_ skb 的区别在于 dev_ alloc_ skb 分配内存时默认设置了 GFP_ ATOMIC 位并在 skb_ head 和 skb_ data 之间留出了 16 字节的空间用于填充硬件帧头。驱动程序应该使用 dev_kfree_skb 正确处理缓冲区锁的问题。

unsigned char *skb_put(struct sk_buff* skb，int len)将 skb_ tail 增加 len 长度返回原来的 skb_tail。其定义就是为在 sk_ buff 的尾部添加数据做准备。

unsigned char* skb_push(struct sk_buff* skb，int len)将 skb_ head 减小 len 长度返回原来的 skb_head。

int skb_tailroom(struct sk_buff* skb)和 int skb_headroom(struct sk_buff* skb)，分别返回 skb 的尾部和头部所剩余的可存放数据的空间。

15.2.5 RTL8109 驱动程序的加载

除了 3Com、Intel 等驰名厂商的产品能在 Linux 识别后自动创建/ etc/conf . modules 系统可加载模组配置文件（作用类似 DOS 中的 CONFIG. SYS）外，一般的网卡芯片则需 Linux 用 vi 创建或修改/etc/conf. modules 文件。

而基于 RTL8109 芯片的网络系统，由于 Realtek 公司已投资 Linux 系统开发，所以 Linux 系统中含有 RTL8109 芯片的通用驱动程序（其可加载模组文件为/lib/modules/./ net/ret18109. o，源程序文件为/ usr/ src/ linux/ drivers/ net/ rtl8109.c），用户只需在/ etc/ conf . modules 文件中添加如下行：alias eth0 rt18109。也可以在 arm/kernel/linux/drivers/net 下修改这个文件的第 500 行为如下形式：

```
for (offset = 1; offset < (buf[0] & 0xff)/2; offset++) {
```

之所以这样修改是因为这个循环的循环次数取决于以太网接口模块所匹配的存储器的容量，该驱动程序中没有进行存储器的容量探测，而是直接采用以太网接口默认的匹配容量，但是硬件板所采用的存储器的容量是默认容量的一半，所以这里将循环次数减少一半。

15.3 实例——基于 RTL8019 的以太网应用系统设计

RTL8019 为台湾芯片生产商 Realtek 公司第三代快速以太网连接而设计 RTL8019 的 10M/100M 兼容以太网接口芯片，它支持多种嵌入式处理器芯片，内置 FIFO 缓存器用于发送和接收数据。

15.3.1 系统基本结构

1．RTL8019 的内部结构

如图 15-4 所示，RTL8019 采用 100 引脚 PQFP 封装，具有很好的性价比。它支持 PnP 自动探测，符合 Ethernet II 与 IEEE 802.3（10Base5、10Base2、10BaseT）标准，内嵌 16 KB SRAM，有全双工通信接口，可以通过交换机在双绞线上同时发送和接收数据，使带宽从 10MHz 增加到 20MHz，是进行以太网通信的理想器件。

RTL8019 的主要引脚功能如下。

- AEN（34）：地址使能引脚，决定电路被分得的地址空间；
- INT0- INT7（97~100，1~4）：中断请求引脚；
- IOCHRDY（35）：读/写命令准备引脚；
- IOCS16B（96）：8 位/16 位数据选择引脚，高电平选择 16 位数据总线，低电平选择 8 位数据总线；
- IORB，IOWB（29，30）：I/O 端口读命令、写命令；
- SMEMRB，SMEMWB（31，32）：寄存器读命令、写命令；
- RSTDRV（33）：复位信号；
- SD0-SD15（36~43，87，88，90~96）：数据线；
- SA0-SD19（5，7~13，15，16，18~27）：地址线；
- X1（50）：20 MHz 晶体振荡器或外部晶体振荡器输入引脚；
- LEDBNC，LED0，LED1，LED2（60~63）：网卡状态指示；

- TPOUT+、TPOUT-、TPIN-、TPIN+（45，46，58，59）：数据发送和接收引脚。

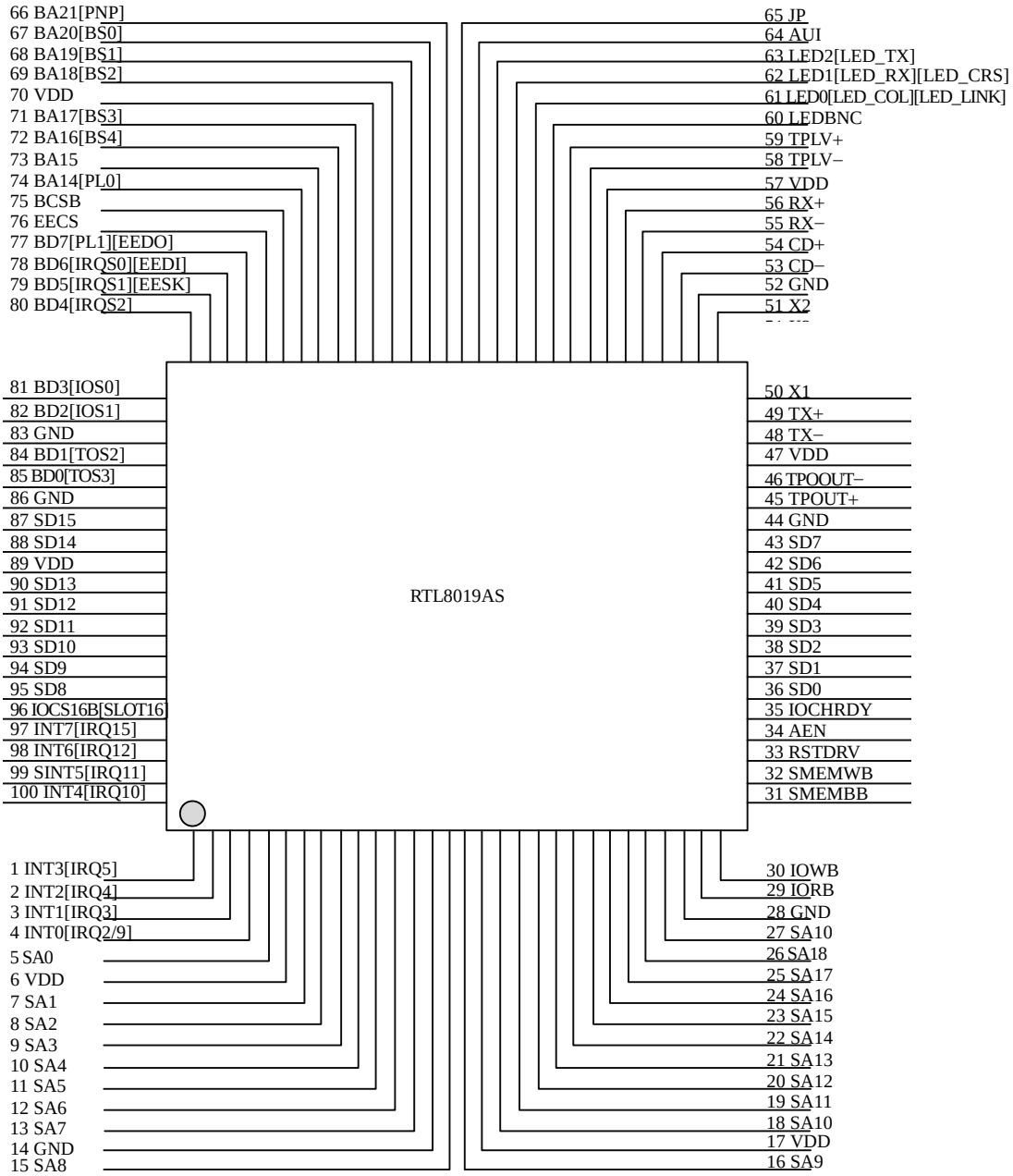


图 15-4　RTL8019 管脚图

在 RTL8019 的 IOS 引脚与基地址对应关系，如表 15-2 所示。

表 15-2　RTL8019 的 IOS 引脚与基地址对应关系表

IOS3	IOS2	IOS1	IOS0	I/O　BASE
0	0	0	0	300H
0	0	0	1	320H

续表

IOS3	IOS2	IOS1	IOS0	I/O BASE
0	0	1	0	340H
0	0	1	1	360H
1	0	0	0	380H
1	0	0	1	3A0H
1	0	1	0	3C0H
1	0	1	1	3E0H
0	1	0	0	200H
0	1	0	1	220H
0	1	1	0	240H
0	1	1	1	260H
1	1	0	0	280H
1	1	0	1	2A0H
1	1	1	0	2C0H
1	1	1	1	2E0H

选择其中一个 I/O 基地址，基地址在某些场合可能有作用，当与 CPU 处理器接口时，经过多次验证选择其中任何一个都不会对网卡的工作产生明显的影响，只是对应的接法不同，编程时的地址不同而已。需要注意的一点是，引脚定义中 IOS3 对应的另一种定义名称是 BD0，IOS2 对应 BD1，IOS1 对应 BD2，IOS0 对应 BD3，如果不仔细的话特别容易引起错误，引起网卡不能正常工作。

举例说明，在表 15-2 中可以看出每个基地址的寻址单元是 32 个（20H），比如图 15-4 中选择 I/O 基地址为 360H，则此时对应的 IOS3-IOS0 的逻辑电平为 0011，0 代表接地，1 代表接+5VDC（即 VDD）。基地址 360H 写为二进制就是 001101100000B，其寻址空间是 360H~37FH，37FH 写为二进制是 001101111111B，与 360H 的二进制相比，可以发现前面 7 位相同而只有后 5 位不同，但是网卡芯片的地址线从 SA19~SA0 有 20 个，综合一下，把前面的二进制补全成 20 位，可以写为 000000000011011xxxxxB，这里 x 表示可以为 0 或者 1。由此可以看到，SA19~SA0 20 根地址线引脚依次的接法是 SA19~SA10、SA7 都是接到数字地 DGND，SA9、SA8、SA6、SA5 都接到+5VDC，其余的 SA4~SA0 接到地址总线的 A4~A0 上。这样地址总线和 IOS 的接法就完成了。

2．寄存器配置

RTL8019 有 4 页寄存器组，每页寄存器组有 16 个寄存器，进行网络通信时需要对这些寄存器进行设置。

例如，CR 是控制命令寄存器，地址是 00H，这个寄存器用来选择寄存器页，控制远程 DMA 操作。其中 STP（0 bit）是停止命令位，PS0、PS1（6 bit、7 bit）是页寄存器选择位，在实际配置寄存器时，首先要指定要配置的寄存器属于哪一页（就是进行 PS1 和 PS0 的设置），然后对该页中的寄存器写入配置信息。

另外，还要进行网络通信必须对网络控制器的各个寄存器初始化，初始化比较烦琐，但非常重要，它决定网络通信的一些重要参数。初始化时需要对上述的各个寄存器进行详

细配置，这部分将在 RTL8019 初始化章节中详细介绍。

3．系统硬件电路结构图

用 RTL8019 芯片设计的以太网控制器相关电路，可以通过 RJ-45 连上以太网，其网络通信部分的框图如图 15-5 所示。采用跳线工作方式即网卡的 I/O 和中断由跳线决定，JP 引脚接高电平时选择 16 位数据总线。系统通过 4 条地址线 A0~A3 选择 RTL8019 的寄存器地址和存储器地址，控制并实现数据的读取。GREEN/RED LED 网卡状态指示引脚连接发光二极管，便于直观判断网卡状态。另外通过逻辑编程器对 RTL8019 的片选信号进行控制。

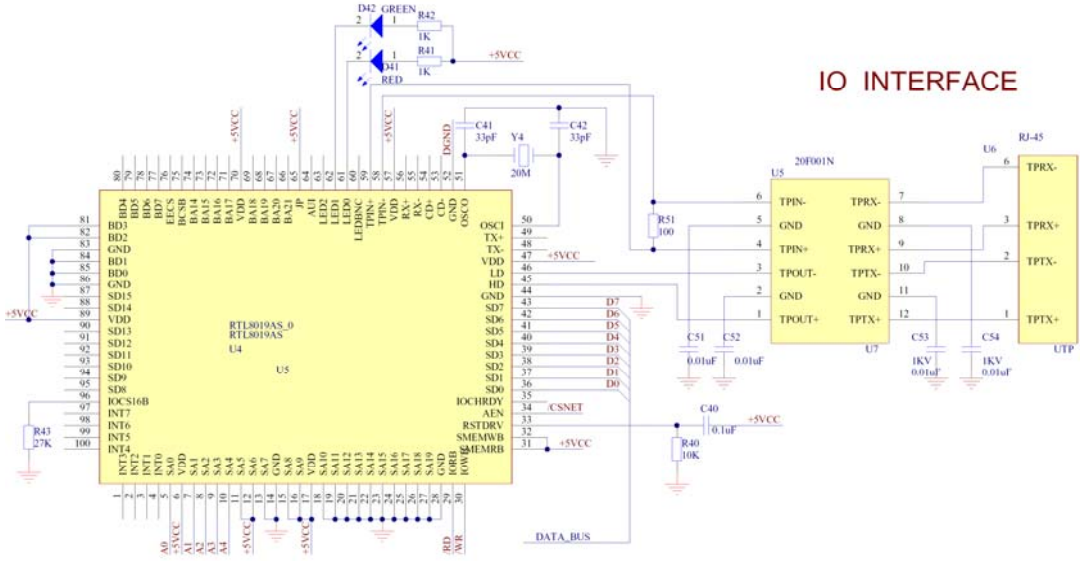


图 15-5　网卡芯片 RTL8019 接口电路原理图

15.3.2　系统工作流程

1．数据包发送

系统内部首先注册 RTL8019 网络设备，从而利用 Linux 为网络设备所提供的数据传输功能接口，实现专用网卡的数据传输。同时将网卡注册成块设备，通过块设备的注册，可利用 Linux 对块设备所提供的创建设备特殊文件的功能，从而方便用户访问专用网卡。

数据的发送过程如图 15-6 所示，数据发送时，首先数据由块设备的数据接口从用户态复制入核心态的系统缓冲区中，再由 RTL8019 的发送接口将数据发送入网络硬件设备。Linux 在进程调度或从系统调用返回时，调度程序判断是否被激活。然后判断写队列是否有请求块。有则将数据送给对方进程，在确认信号到来后，将请求块移去。若数据发送后响应的为重传信号，则根据约定进一步处理。重复处理下一个请求块直至请求队列为空。若队列空且有进程睡眠则唤醒睡眠进程。

2．数据包接收

数据的接收过程如图 15-7 所示，在接收时，通过 RTL8019 的设备接口接收数据到系统缓冲区中，再通过块设备接口将数据发送入用户缓冲区中。当有数据收到时，激活中断

服务子程序，中断服务子程序将数据读入数据队列，若有读睡眠进程则唤醒该进程，同时将到来的数据进行校验，数据正确则发回信号，否则发回重传信号。

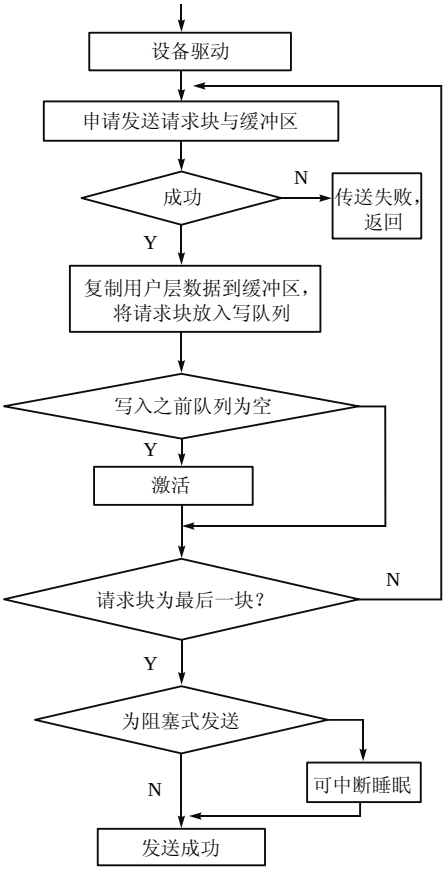


图 15-6　网络数据的发送

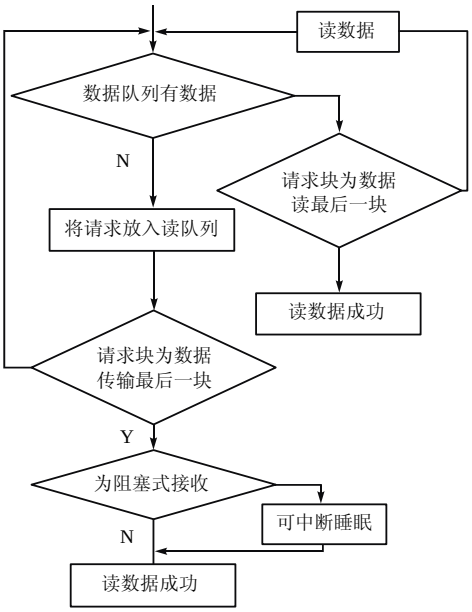


图 15-7　网络数据的接收

这部分功能具体的实现方法将在后面的章节中具体介绍。

15.3.3　系统模块源代码实现

1．初始化 RTL8019

初始化部分完成 RTL8019 在使用之前的初始化工作：设置相关工作模式的寄存器，分配和初始化接收及发送缓冲区，初始化网卡接收地址。

RTL8019 的初始化代码如下：

```
reg00 = 0x21 ;              // CR = 0x21 ;  // STOP| NO-DMA
reg0e = 0xc9 ;              // DCR 数据配制寄存器 16 位远端数据
dma
Temp = Reset-Reg ;
Delay(1)
Reset- reg = temp ;         // 复位 8019
Delay(100)
                            // 关闭对于配置存储器的支持
```

```
reg00 = 0xel ;              // 选中第 3 页
reg01 = 0xe0 ;              // 9346CR EEM1 = EEM1 = 1
reg04 = 0xel ;              // config1 set IRQ bit
reg01 = 0xel ;              // 9346CR EEM1 = EEM1 = 0
                            // 3 3 3 3 3 3 3 停止 8019 3 3 3 3 3 3 3
reg00 = 0x21 ;              // CR = 0x21 ;   // STOP| NO-DMA
reg0b = 0 ;                 // 清除远程 DMA 计数器的 MSB
reg0a = 0 ;                 // 清除远程 DMA 计数器的 LSB
                            // 读取 ISR ,等待 ISR ,等待 ISR &ISR-RESET ,1. 6ms
for ( I = 0 ; I < 0xfff ; I + + )
{
    temp = reg07 ;
    if (temp &RTL8019- ISR-RST)
    {
        reg07 = temp ;
        break ;
    }
Delay(200) ;
}
                        // 如果超时没有收到 8019 的复位信号,表示 8019 芯片有硬件问题
if ( I = = 0xfff)
{
    Uart- Printf ("Reset RTL8019 Fail.
    Please check your hardware. \ n') ;
    return ;
}
reg0d = 0xe2 ;          // TCR = 0x02 ;开启回环模式
reg00 = 0x22 ;          // CR = 0x22 ;// START| NO-DMA
Delay(20000) ;          // 延时
                        // 3 3 3 3 3 3 3 配置 8019// 3 3 3 3 3 3 3
reg00 = 0x21 ;          // 选择页 0 的寄存器,网卡停止运行,因为还没有初始化
temp = reg00 ;
                        // 测试 8019 的寄存器能否工作
if ( (temp &0x21) ! = 0x21)
{
    Uart- Printf ("Write RTL8019 Fail \ n") ;
    return ;
}
reg0e = 0x49 ;          // DCR 数据配置寄存器 16 位数据 dma
reg0b = 0 ;             // 清除远程 DMA 计数器的 MSB
reg0a = 0 ;             // 清除远程 DMA 计数器的 LSB
reg0c = 0x04 ;          // RCR
reg0d = 0x02 ;          // TCR = 0x02 回环模式
reg03 = 0x4c ;          // BNRY
reg01 = 0x4C;           // 寄存器 Pstart
reg02 = 0x80 ;          // Pstop
reg07 = 0xff ;          // ISP
reg0f = 0x01 ;          // IMR open rx interrupt interrupt 1 = =IRQ2/ 9
reg04 = 0x45 ;          // TPSR
reg00 = 0x61 ;          // CR- STOP| CR-NO-DMA| CR- PAGE1//选择 1 的寄存器
                        // 初始化物理地址
GetMac (mac) ;
GetMac (mac) ;
                        // 初始化组播地址 Ether InitMar ( ) ;
reg07 = 0x4d ;          // CURR
```

```
reg00 = 0x21 ;                  // move back to page 0
reg00 = 0x22 ;                  // 选择页 0 寄存器,网卡执行命令
reg0d = 0x00 ;                  // TCR-NO-LOOPBACK
reg07 = 0xff ;                  // ISR
                                // 配置结束
EtherNetOpenInterrupt ( ) ;     // 设置中断
EtherSetRegPage (0) ;           // 选择页面 0
```

2. 数据包的发送与接收

（1）传送数据包

发送部分只要把数据写入缓冲区，启动执行命令，RTL8019 自动发送。一般在 RAM 内开辟两个以太网数据包的空间作为发送缓冲区。作为一个集成的以太网芯片，数据的发送校验、总线数据包的碰撞检测与避免是由芯片自己完成的，只需要配置发送数据的物理层地址、源地址、目的地址、数据包类型以及发送的数据。

当写发送命令时，RTL8019 将从 TPSR<<8 地址开始发送 size 个字节的数据。命令为：

```
WriteReg(CR,((PrePage&0xC0) | CR_ABORT_COMPLETE_DMA | CR_TXP | CR_START_
COMMAND));
```

如果发送的数据包存储如图 15-8 所示的黑色区域，RTL8019 不能自动连接两个区域，即当前发送页为 RECEIVE_START_PAGE 时，它不会转到 SEND_START_PAGE，而是发送阴影部分的内容。

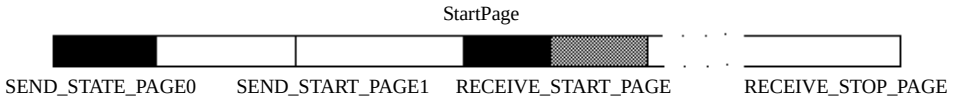


图 15-8　发送的数据包存储

（2）接收数据包

接收部分完成数据接收任务。RTL8019 接收到以太网数据包后自动存在接收缓冲区并发出中断信号，CPU 在中断程序中通过 DMA 即可接收到数据，亦即通过远端 DMA 把数据从 RTL8019 的 RAM 空间读回 ARM 中处理。这里主要是对一些相关的寄存器进行操作。

RTL8019 通过 LocalDMA 把接收的数据写入接收缓冲区，并且自动改变 CURR，同时识别缓冲区的界限，这些过程都不用用户的干预。接下来是用户如何读取接收缓冲区内的数据包。

当一个无错的数据接收完毕，则触发中断处理函数。接下来就是要读取数据包到分配的内存中，读取多少个可以从 ReceiveByteCount 得知。这里要处理一种情况，如果接收的数据包存储不是连续的，如图 15-9 所示，最后网卡通过中断控制器向嵌入式处理器响应中断，中断完毕清除中断标志，使得后来的同级和低级中断能够相应。

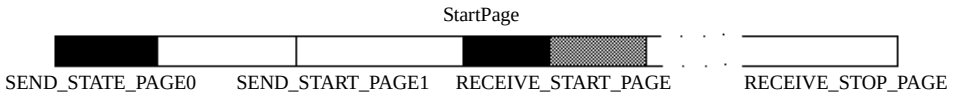


图 15-9　数据包存储不连续

以下是以太网控制器 RTL8019 工作的详细代码程序。

```
#include "GloblDef.h"
#include "MMenage.h"
#include "RTL8019.h"

extern BYTE MemAllocation(WORD size);
extern void FreePage(BYTE page);
extern BYTE xdata *MemPageToPoint(BYTE page);
extern BYTE WriteQueue(BYTE page,struct Queue xdata * pQueue);

BYTE xdata LocalMACAddr[6]={0x52,0x54,0x4c,0x30,0x2e,0x2f};
struct Queue xdata QueueNetPacketIn;
BYTE StartPageOfPacket;
                                    // 接收头文件信息
struct RTLReceiveHeader
{
    BYTE ReceiveStatus;
    BYTE NextPacketStartPage;
    BYTE PacketSizeLow;
    BYTE PacketSizeHigh;
}Head   ;/*Head 须为全局变量*/
BYTE xdata Head[4];
                                    // 上一次传输起始页
BYTE LastSendStartPage;

sbit RTLResetPin = RTL_RESET_PIN;
                                    // 读 rtl8019 寄存器端口
BYTE ReadReg(WORD port)
{
    BYTE xdata * p;
    p = (BYTE xdata *)port;
    return *p;
}
                                    // 写寄存器
void WriteReg(WORD port,BYTE value)
{
    BYTE xdata * p;
    p = (BYTE xdata *)port;
    *p = value;
}
                                    // 选择寄存器页使用
void RTLPage(BYTE Index)
{
                                    // 设置 CR, CR_TXP 7-6 位为 0(为 1 ，包重传)
    BYTE temp;
    temp = ReadReg(CR);
    temp = temp & 0x3B;             //set 7-6 and 3 bit to 0
    Index = Index<<6;
    temp = temp | Index;
    WriteReg(CR,temp);
}
                                    // PRA 为物理地址
void RTLInitial()
{
    BYTE temp;
    int i;

                                    // 硬件重启
```

```
    RTLResetPin = 1;
    for(i = 0;i<255;i++);
    RTLResetPin = 0;
                                    // 如果硬件重启时延很大， rtl 自我初始化
    for(i=0;i<DELAY_AFTER_HARDWARE_RESET;i++);

                                    // 写重启口
    temp = ReadReg(RESET_PORT);
    WriteReg(RESET_PORT,temp);

                                    // 初始化 RTL 寄存器
    WriteReg(CR,(CR_PAGE0 | CR_ABORT_COMPLETE_DMA | CR_STOP_COMMAND));
                                    // 设置 page0, stop command

    WriteReg(PSTART_WPAGE0, RECEIVE_START_PAGE);        // Pstart
    WriteReg(PSTOP_WPAGE0,  RECEIVE_STOP_PAGE);         // Pstop
    WriteReg(BNRY_WPAGE0,       RECEIVE_START_PAGE);    // BNRY
    WriteReg(TPSR_WPAGE0,       SEND_START_PAGE0);      // TPSR

    WriteReg(RCR_WPAGE0,        0xCE);  // RCR: 在 RTL8019.h 定义
    WriteReg(TCR_WPAGE0,        0xE0);  // TCR: 在 RTL8019.h 定义
    WriteReg(DCR_WPAGE0,        0xC8);  // DCR: 在 RTL8019.h 定义
    WriteReg(IMR_WPAGE0,0);             // RTL 接收中断使能
    WriteReg(ISR_WPAGE0,        0xFF);  // 写 FF 清除所有中断标志

    RTLPage(1);

    WriteReg(CURR_WPAGE1,RECEIVE_START_PAGE + 1);
                                        // MAR0
    WriteReg(0x08,0x00);
    WriteReg(0x09,0x41);
    WriteReg(0x0a,0x00);
    WriteReg(0x0b,0x80);
    WriteReg(0x0c,0x00);
    WriteReg(0x0d,0x00);
    WriteReg(0x0e,0x00);
    WriteReg(0x0f,0x00)
                                        // 设置物理地址
    WriteReg(PRA0_WPAGE1,LocalMACAddr[0]);
    WriteReg(PRA1_WPAGE1,LocalMACAddr[1]);
    WriteReg(PRA2_WPAGE1,LocalMACAddr[2]);
    WriteReg(PRA3_WPAGE1,LocalMACAddr[3]);
    WriteReg(PRA4_WPAGE1,LocalMACAddr[4]);
    WriteReg(PRA5_WPAGE1,LocalMACAddr[5]);
                                        // 传输起始页
    LastSendStartPage = SEND_START_PAGE0;
    StartPageOfPacket = RECEIVE_START_PAGE + 1;
                                        // 初始化结束
    WriteReg(CR,(CR_PAGE0 | CR_ABORT_COMPLETE_DMA | CR_STOP_COMMAND));
}
                                        // 写 buffer 到 rlt ram
void RTLWriteRam(WORD address, WORD size, BYTE xdata * buff)
{
    WORD i;
    BYTE PrePage;   /* store page */
    PrePage = ReadReg(CR);
```

```
    RTLPage(0);
    WriteReg(RSARH_WPAGE0,(BYTE)((address>>8)&0x00ff));
    WriteReg(RSARL_WPAGE0,(BYTE)address);
    WriteReg(RBCRH_WPAGE0,(BYTE)((size>>8)&0x00ff));
    WriteReg(RBCRL_WPAGE0,(BYTE)size);
    WriteReg(CR,(0x00 | CR_REMOTE_WRITE | CR_START_COMMAND));
    for(i=0;i<size;i++)
    {
        WriteReg(REMOTE_DMA_PORT,buff[i]);
    }
                                            // 完成 dma
    WriteReg(RBCRH_WPAGE0,0);
    WriteReg(RBCRL_WPAGE0,0);
    WriteReg(CR,((PrePage&0xC0)|CR_ABORT_COMPLETE_DMA|CR_START_COMMAND));
}
                                            // 读 rlt ram 数据到 buffer
void RTLReadRam(WORD address,WORD size,BYTE xdata * buff)
{
    WORD i;
    BYTE PrePage;                           // 存储页

    PrePage = ReadReg(CR);
    RTLPage(0);
    WriteReg(RSARH_WPAGE0,(BYTE)((address>>8)&0x00ff));
    WriteReg(RSARL_WPAGE0,(BYTE)address);
    WriteReg(RBCRH_WPAGE0,(BYTE)((size>>8)&0x00ff));
    WriteReg(RBCRL_WPAGE0,(BYTE)size);
    WriteReg(CR,(0x00 | CR_REMOTE_READ | CR_START_COMMAND));
    for(i=0;i<size;i++)
    {
        buff[i] = ReadReg(REMOTE_DMA_PORT);
    }
                                            // 完成 dma
    WriteReg(RBCRH_WPAGE0,0);
    WriteReg(RBCRL_WPAGE0,0);
    WriteReg(CR,((PrePage&0xC0)|CR_ABORT_COMPLETE_DMA|CR_START_COMMAND));
}

BYTE RTLSendPacket(BYTE xdata * buffer,WORD size)
{
    BYTE StartPage;
                                            // 存储页
    BYTE PrePage;
    PrePage = ReadReg(CR);
                                            // 检查包大小
    if(size < MIN_PACKET_SIZE || size > MAX_PACKET_SIZE)
        return FALSE;
                                            // 写包到 ram
    if(LastSendStartPage == SEND_START_PAGE0)
    {
        StartPage = SEND_START_PAGE1;
        LastSendStartPage = SEND_START_PAGE1;
    }
    else
    {
        StartPage = SEND_START_PAGE0;
        LastSendStartPage = SEND_START_PAGE0;
```

```
        }
        RTLWriteRam(((WORD)StartPage)<<8,size,buffer);
                                                // 等待上一次传输结束
        while((ReadReg(CR) & CR_TXP) == CR_TXP);
                                                // 写传输起始页和大小
        RTLPage(0);

        WriteReg(TPSR_WPAGE0,StartPage);                    // TPSR
        WriteReg(TBCRL_WPAGE0,(BYTE)size);                  //low
        WriteReg(TBCRH_WPAGE0,(BYTE)((size>>8)&0x00ff));    //high
        WriteReg(CR,((PrePage&0xC0)|CR_ABORT_COMPLETE_DMA | CR_TXP | CR_START_
COMMAND));
        return TRUE;
    }

    void RTLReceivePacket()
    {
        BYTE curr,bnry;
        WORD address;
        WORD PacketSize;
        BYTE MemPage;
        struct MemHeader xdata *pMemHead;

        RTLPage(1);
        curr = ReadReg(CURR_RPAGE1);
        RTLPage(0);
                                            //在接收缓存中读所有包
        while(TRUE)
        {
                                            // 检验起始页是否未知错误
            if(StartPageOfPacket >= RECEIVE_STOP_PAGE || StartPageOfPacket <
RECEIVE_START_PAGE)
            {
                                            // 用 curr 作为 StartPageOfPacket
                StartPageOfPacket = curr;
                break;
            }
                                            //检查是否有包读到
            if(StartPageOfPacket == curr)
                break;
                                            //读一个包
                                            // 读包头信息
            address = ((WORD)StartPageOfPacket)<<8;
            RTLReadRam(address,4,Head);

                                            // 校验 rsr
            if(Head[0] & RSR_RECEIVE_NO_ERROR)
            {
                                            //好包
                                            // 得到 MAC 校验和
                PacketSize = ((WORD)Head[3])*256 + Head[2] - 4;
                                            // 分配 buffer ， 读包到 buffer

                MemPage = MemAllocation(PacketSize);
                if(MemPage != PAGE_NOT_FOUND)
                {
```

```
                    pMemHead= (struct MemHeader xdata *)MemPageToPoint(MemPage);

                    pMemHead->StartPos = (BYTE xdata *)pMemHead + sizeof(struct
MemHeader);
                                                        // pos 起始有效地址
                    pMemHead->StopPos = pMemHead->StopPos + PacketSize;
                                                        // 停止 pos

                    address += 4;
                    if(StartPageOfPacket > Head[1] && Head[1] != RECEIVE_START_
PAGE)
                    {
                        RTLReadRam(address,(((WORD)RECEIVE_STOP_PAGE)<<8)  -
address,pMemHead->StartPos);                            // 读 rtl
                        RTLReadRam(((WORD)RECEIVE_START_PAGE)<<8,PacketSize
- ((((WORD)RECEIVE_STOP_PAGE)<<8) - address),
                            pMemHead->StartPos + ((((WORD)RECEIVE_STOP_PAGE)
<<8) - address));                                       // 读 rtl
                    }
                    else
                    {
                        RTLReadRam(address,PacketSize,pMemHead->StartPos);
                                                        // 读 rtl
                    }
                   if(WriteQueue(MemPage,&QueueNetPacketIn) == PAGE_NOT_FOUND)
                                                        // 写到对列
                    {
                                                        // 对列满
                        #ifdef  DEBUG
                        printf("\n-------queue full-------");
                        #endif
                        FreePage(MemPage);
                        break;
                    }
                }
                else
                {
                                                        // mem over
                    #ifdef  DEBUG
                    printf("\n-------mem over-------");
                    #endif
                    break;
                }
            }
                                                        // 得到下一包的起始页
            StartPageOfPacket = Head[1];
        }
                                                        // 重置 bnry
        bnry = StartPageOfPacket - 1;
        if(bnry < RECEIVE_START_PAGE)
            bnry = RECEIVE_STOP_PAGE - 1;
        WriteReg(BNRY_WPAGE0,bnry);
    }
```

以下这两个函数也很重要，当调用 Start8019 函数以后，网卡就开始接收数据，也可以让其发送数据。调用 Stop8019 函数以后，网卡停止接收数据，也不能发送数据。在初始

化工作全部结束以后，如果想让网卡开始工作，则调用 Start8019 函数，程序退出以前一定要调用 Stop8019 函数。否则在程序退出以后，网卡将继续接收数据到接收缓存中，但是却没有相应的中断服务程序提取数据包。这样在下次运行程序时，8019 的接收缓冲区中就有大量的数据包。比如，现在本机向另外一台计算机发送建立连接的请求，对方应答了，但是在这个应答数据包的前面有大量的不相关的数据包，这样就有可能出错。

```
void Start8019()
{
    WriteReg(CR,CR_ABORT_COMPLETE_DMA | CR_START_COMMAND);
}

void Stop8019()
{
    WriteReg(CR,CR_ABORT_COMPLETE_DMA | CR_STOP_COMMAND);
}
```

15.3.4 系统调试

RTL8019 网卡芯片不能单独工作，还必须有一个网络变压器在 RJ-45 接口和网卡芯片中间进行电平变换。这部分各元器件焊接完成后，就可以进行测试，在测试前编写了测试程序代码，因为各种初始化代码很长，这里就不列举出来，需要说明一点，在这部分网卡芯片有两个 LED 指示是用于指示接收和发送状态的，如果有网络连接并且正常收发数据包时，LED 会闪烁，在判断网卡芯片是否工作正常时，有两个依据，一是看状态指示 LED 是否有闪烁，二是用专用网络监听工具软件进行监听，如 sniffer，监听到网卡不断发送出来的特定测试数据包就表明网卡正常工作。

以下是网卡芯片 RTL8019 是否正常工作的测试程序，电路板上需要 RAM 和正确结果：使用网络监听工具 sniffer 或者 netxray 监听 MAC 地址 52544c302e2f 的数据包，程序运行以后监听工具应该监听到一个数据包。

```
void main(void)
{
    BYTE temp;
    WORD port = 1001;
    LocalMACAddr[0]=0x52;
    LocalMACAddr[1]=0x54;
    LocalMACAddr[2]=0x4c;
    LocalMACAddr[3]=0x30;
    LocalMACAddr[4]=0x2e;
    LocalMACAddr[5]=0x2f;
    LocalIPAddress = 0xc0a8020d;        // 本地地址 192.168.2.14
    ServerIPAddress = 0xc0a8020e;       // 目的地址 192.168.2.13

                                        //初始化
    SerialInitial();
    MemInitial();
    NetInInitial();
    RTLInitial();
    Start8019();
    InterruptInitial();

                                        // 建立一个 ARP 包
```

```
    p[0] =0xff;
    p[1] =0xff;
    p[2] =0xff;
    p[3] = 0xff;
    p[4] = 0xff;
    p[5] = 0xff;
    p[6] = 0x52;
    p[7] =0x54;
    p[8] =0x4c;
    p[9] =0x30;
    p[10] =0x2e;
    p[11] =0x2f;
    p[12] = 0x08;
    p[13] = 0x06;
    p[14] = 0x00;
    p[15] = 0x01;
    p[16] = 0x08;
    p[17] = 0x00;
    p[18] = 0x06;
    p[19] = 0x04;
    p[20] = 0x00;
    p[21] = 0x01;

                                // 发送 ARP 包
    RTLSendPacket(p,60);
    while(1);

#ifdef  DEBUG
    printf("\n-------bigine-------");
#endif
                                // 处理
    TCPBind(port);
    if(TCPConnect(ServerIPAddress,1001) == TRUE)
    {
        while(UserFunc());
    }

                                // 延时
    for(temp;temp<255;temp++);
#ifdef DEBUG
    printf("\n run over!");
#endif
                                // 存储
    Stop8019();
    while(1);
}
```

15.4　本章总结

本章详细讲述了以嵌入式 Linux 为核心、以 RTL8019 为网络接口芯片的嵌入式以太网接口的软硬件设计方法。RTL8019 芯片具有性价比高、连接方便等特点，是进行嵌入式以太网设计时出色的控制芯片。由于 RTL8019 性能高，价格适中，不仅可用于工业现场实现现场节点的自动上网功能，而且可以用于信息家电的以太网接口，实现远程控制，即在网上就可以控制家中的电器，因此该芯片技术具有很好的发展前景。

第 16 章

无线网络数据传输系统开发实例

无线网络有着广阔的市场前景，正在成为嵌入式应用中的一个快速增长点，其应用涵盖数据终端、掌上电脑（PDA）等领域。基于 IEEE 802.11b 协议的无线局域网 WLAN 是数据通信里的新兴领域，它所提供的无线接入功能在很大程度上满足了用户在移动情况下对无线数据传输接入的需求。对于中长距离的无线接入，IEEE 802.11b 是一个理想的方案，它可以方便地接入局域网，在与有线网络的互联的过程中，完成接入点 AP（Access Point）的任务。

采用 PCMCIA 接口的无线网卡是实现无线网络传输系统的主要手段，其一个典型应用 D-Link650 PCMCIA 无线网卡，如图 16-1 所示。D-Link650 支持 802.11 无线标准，提供 108Mbps 的无线传输速率。



图 16-1　基于 D-Link 650 无线网卡的实物图

PCMCIA 是个人计算机存储卡国际协会的简称，由于 PC 卡具有重量轻、方便灵活的特点，所以应用广泛，同时也是嵌入式平台下开发无线网络的很好的产品。基于 Linux 的 PCMCIA 驱动有专门的 PCMCIA-CS 系列的驱动程序开放源代码，该系列的源代码包括许多便携计算机的驱动和某些嵌入式系统的驱动，经过修改，与 Linux 内核编译在一起后便可使用。

16.1　无线网络传输系统简介

嵌入式实时操作系统是嵌入式系统应用软件开发的支撑平台，网络化是目前的主要趋势之一。在各种嵌入式操作系统中，Linux 凭借其在结构清晰、源代码的开放性等方面的

优势，在基于监控系统、手持设备等嵌入式系统领域中的应用广泛。

如图 16-2 所示是基于 PCMCIA 的无线数据传输系统，利用嵌入式 Linux 设备能够与 PC 服务器一起组成局域无线网络，可以满足在不具备架设网线的环境下的网络通信，实现了 802.11b 无线网络，并能够在该网络上加上一些语音网关功能实现 VoIP 与 WLAN 的结合。

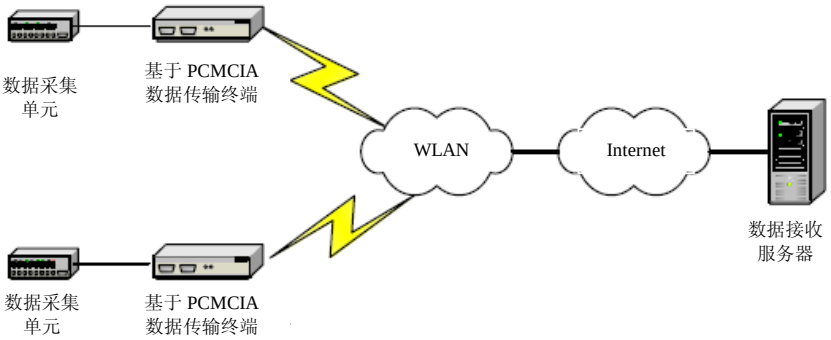


图 16-2　基于 PCMCIA 的无线数据传输系统

802.11 无线网络的实现框图如图 16-3 所示，通过 PC 端，将语音信号转换为 WAV 格式的文件，然后通过 802.11 无线网络发送给目标硬件板，目标板通过 PCMIA 接口的无线网卡，将文件接收，然后转换为 WAV 格式的文件保存在 SDRAM 中，再调用 S-Player 软件播放出来，实现通过无线网络的语音传输。

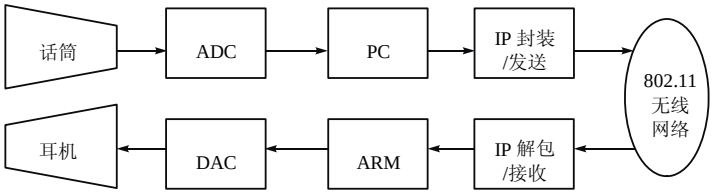


图 16-3　两个节点的 802.11 无线网络框图

利用无线局域网 WLAN 传输数据是在通信行业谈得最多的事情之一。其实，与有线网络一样，WLAN 同样可以传输语音，实现 VoIP 无线化，也就是无线语音传输。

虽然，无线局域网 WLAN 并不是传输 VoIP 的首选网络结构，但在线缆无法铺设的地区和移动性的工作地点，要使人们与 IP 电话保持连接，将 802.11 和 VoIP 两种技术结合起来是一个不错的选择。利用无线网络设备加语音网关，可以在 IP 链路中传输数据和语音，基于无线设备的 QoS，以确保 IP 话音质量。这样的架构相当于内部电话的互通，不产生任何费用，还可以解决分节点连接因特网的问题。

16.2　相关开发技术

16.2.1　无线网络接入技术

近年来，随着网络及通信技术的不断发展，无线通信技术得到了迅速发展，无线通信技术在人们生活中发挥着越来越重要的作用。

1．Wi-Fi（802.11）

Wi-Fi（Wireless Fidelity）是一种无线通信协议，正式名称是 IEEE 802.11（b/a/g），Wi-Fi 速率最高可达 54Mbps，传输距离在 100 m 左右。

802.11 系列标准是 IEEE 推出 WLAN 标准，最早的 802.11 协议速率只有 2Mbps，但随后推出的 802.11b 和 802.11a 则分别将速率提高到了 11Mbps 和 54Mbps，传输距离也得到了加强。此外，由于作为下一代标准的 802.11a 与 802.11b 无法进行互通，因此 IEEE 又推出了 802.11g 作为两个阶段之间的过渡，其构筑在 802.11b 的基础上，并提供了 54Mbps 的传输速率。改进后的 802.11b/g 协议以其成本低、灵活性高、移动性强、吞吐量高、通信可靠等诸多特性，迅速得到了广大厂商的支持，目前已成为了无限局域网通信的主要技术。

IEEE 802.11b/g 的工作频段为 2.4GHz，与蓝牙使用的频段相同，使用直接序列扩频技术传输数据，分频方式为正交频分多路复用（OFDM），虽然在数据安全性、通信稳定性、设备体积、功耗等方面略逊于蓝牙技术，但它的传输距离和通信速率都比较高，将其功率提高到一定程度后，覆盖面积可达数百米，很适合用于无线局域网。由于 Intel 和微软对 Wi-Fi 的支持，802.11b/g 得以迅速普及，广泛应用于个人电脑、PDA、数字终端等各种设备中。

2．UWB 超宽带无线电

现代意义上的超宽带 UWB（Ultra Wide Band）无线电，又称冲激无线电（Impulse Radio）技术，但早期其应用一直仅限于军事、灾害救援搜索雷达定位及测距等方面。2002 年，这项无线技术首次获得了美国联邦通信委员会（FCC）的批准用于民用通信，从而引起了世界各国的广泛关注，自 1998 年起，FCC 对超宽带无线设备对原有窄带无线通信系统的干扰及其相互共容的问题开始广泛征求业界意见，在有美国军方和航空界等众多不同意见的情况下，FCC 仍开放了 UWB 技术在短距离无线通信领域的应用许可，这充分说明此项技术所具有的广阔应用前景和巨大的市场诱惑力。

由于 UWB 是一种无载波通信技术，它不采用正弦载波，而是利用纳秒至微秒级的非正弦波窄脉冲传输数据，因此其所占的频谱范围很宽。UWB 的带宽非常宽，目前 FCC 开放的频段是 3.1~10.6 GHz，故 UWB 系统发射的功率谱密度可以非常低，甚至低于 FCC 规定的电磁兼容背景噪声电平（–41.3dBm，FCC Part15），由此可见，短距离 UWB 无线通信系统与其他窄带无线通信系统可以共存。

UWB 的传输速率可达几十 Mbps 至几 Gbps；其收发信机结构简单，成本低于全数字化；并且其固有的抗多径衰落功能很强。

目前，国际上关于 UWB 在短距离无线通信领域的研究与开发已经进入制定标准的阶段，IEEE802.15.3a 工作组已收到多项提案。UWB 目前采用的主要调制方式有脉冲幅度调制 PAM、通断键控调制 OOK、跳时脉位调制 TH-PPM 和跳时/直扩二进制移频键控调制 TH/DS-BPSK。

3．无线网络接入方案选择

为实现无线通信要求，这里进行各种方案的讨论。若不使用现有的通信协议来实现，如采用 NORDIC 公司的 nRF401 芯片，自定通信协议，虽然价格便宜，但系统就不能直接与现有的无线网络互连，而且 nRF401 的传送速率只能达到 20Kbps。

为此可以考虑采用已有的无线通信协议，其中可以选择的方案有蓝牙（Bluetooth）、HomeRF、WLAN（802.11a/b/g）等。由于考虑到蓝牙的通信距离只有 10 米左右，HomeRF 在国内使用比 WLAN 要少，最后采用了 WLAN（802.11b）作为无线方案。

目前校园和家庭的 WLAN 的使用越来越多，这为该方案的推广提供了便利。实现 WLAN 有两个方案。

一个方案是购买 802.11b 芯片自己设计电路，这个方案可以将 802.11b 芯片集成到电路板上，减小电路板的大小。可以采用的芯片有 Intersil 公司的 Prism2/2.5/3 芯片组、Realtek 公司的 RTL8180L 芯片、TI 公司的 ACX100 芯片等。但是这些公司的网站上不提供这些芯片的详细使用文档，难以进行开发。这些芯片的 WLAN 解决方案是需要购买的，且价格比较高。

另外一个可以采用现有的无线网卡的方案，采用该方案有利于系统的升级，即如果 802.11b 协议升级以后不用重新制作硬件电路板，只要购买新的网卡即可。

16.2.2　基于 PCMCIA 的无线网卡接口

无线网卡的主要接口形式有用于台式机的 PCI 接口、CF 卡接口、MiniPCI 和用于笔记本电脑的 PCMCIA 接口、USB 接口。由于考虑到 PCI 卡接口体积过大，CF 卡和 MiniPCI 使用不是很普及，USB 接口的无线网卡虽然体积比较小，但是相对 PCMCIA 的价格贵一些。这里可以使用 PCMCIA 接口形式。

PCMCIA 接口驱动是针对 802.11b 芯片组的，大部分芯片驱动支持 802.11b。PCMCIA 全称为 Personal Computer Memory Card International Association，中文是“国际个人电脑存储卡协会”。凡符合此协会定义的界面规定技术所设计的界面卡，便可称为 PCMCIA 卡或简称为 PC 卡。以前这项技术标准只适用于存储器扩充卡，但后来还扩展到存储器以外的外部设备，如网络卡、视频会议卡及调制解调器等。

PCMCIA 接口信号定义有 3 种模式：存储器模式（Memory Mode）、输入/输出模式（I/O Mode）、卡总线模式（CardBus Mode）。每种模式下的接口信号定义都有所不同。PCMCIA 无线网卡接口信号定义属于 I/O 模式。如图 16-4 所示给出了 PCMCIA 的体系结构，PC 卡接口（基于 PD6710 芯片）主要实现两种功能：一是与 ARM 上的扩展总线相连，二是对无线网卡内部的操作，包括对缓冲 RAM 的读写、MAC 芯片的控制、读写存储空间以及 I/O 空间等。

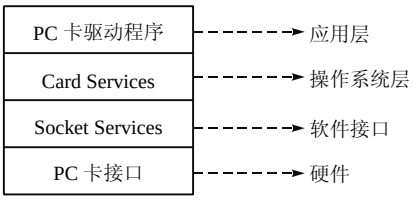


图 16-4　PCMCIA 体系结构

PCMCIA 卡共分成 4 种规格，分别是 TypeⅠ、TypeⅡ、TypeⅢ以及 CardBus。由于 CardBus 属于需要高频宽外设的界面规格，而且不常见。而 TypeⅠ、TypeⅡ及 Type Ⅲ，它们常被应用于一般的外设规格上。

- TypeⅠ的规格：面积为 8.56×5.4cm，厚度则为 0.33cm；适用于一般存储器扩充卡。

- TypeⅡ的规格：面积为 8.56×5.4cm，厚度则为 0.5cm；应用范围包括 Modem 卡、Network 卡、视频会议卡等。
- TypeⅢ的规格：面积为 8.56×5.4cm，厚度为 1.05cm；应用范围为硬盘。

从外观上看，这 3 种 PCMCIA 卡的尺寸都是 8.56cm×5.4cm，实际上，它们的区别在于卡的厚度。TypeⅠ的厚度最薄，最适合用于存储器扩充卡；TypeⅡ则常用于数据传输、网络连接等产品，Modem 卡和 Network 卡都是TypeⅡ规格的；Type ⅢI因为较厚，所以它适合取代机械式的储存媒体，如硬盘。

PCMCIA 卡除了轻巧、方便携带外，它有和 USB（Universal Serial Bus）外设相同的特色，就是“热插拔”（Hot Plugging）功能。所以 PCMCIA 规格的设备可于电脑开机状态时安装插入，并能自动通知操作系统做设备的更新，省去了安装的麻烦。

16.2.3 PCMCIA 驱动程序

在 PCMCIA 驱动程序的编写中，主要涉及 Linux 内核的内存管理机制、PC 卡的时序要求、PCMCIA 接口的信号选择。

首先，要在编写 PCMCIA 驱动程序之前定义一个新的存储空间；其次，要编写 init 函数，该函数要在内核启动期间完成 PCMCIA 各个寄存器参数的填写和正确描述；再次，在 init 函数正确的前提下，编写读写函数；最后，编写 interrupt 函数，这类的函数只对个别寄存器的情况。

硬件上系统采用 CS 片选标志来选中 PCMCIA 的空间，这一点必须在 arch/目录下的 head. s 文件中重新描述，否则，系统不能合法地访问这个空间。然后在编写驱动之前，要在 init.c 文件中的--initfunc（void MMU-init (void) ）函数里加入如下代码。

```
# ifdef CONFIG-
        ioremap (PCMCIA-ADDR , PCMCIA-SIZE) ;
# endif
```

这样做的意义是为 PCMCIA 定位一个空间，空间的地址是 PCMCIA-ADDR，大小是 PCMCIA-SIZE。

在嵌入式 Linux 系统中，CPU 不能按物理地址来访问存储空间，而必须使用虚拟地址，必须“反向”地从物理地址出发找到一片虚拟空间并建立起映射。Ioremap 的作用正是把物理空间映射成内核接受的虚拟内存空间，之后可以利用函数返回的地址访问系统需要的物理地址，但是要在这里说明的是，如果使用了上面给出的代码，PCMCIA 卡内存的物理地址和虚拟地址是一样的，不需使用返回值，而直接使用物理地址。

编写 PCMCIA 的 init 函数。init 函数所做的工作就是“打开”PCMCIA 的接口，使之能够完成通信。该 init 函数里应完成以下 4 个步骤。

（1）把所有的 PCMCIA base register 和 option register 全部置零，如果不这么做，有可能和别的 memory 冲突，造成读写错误；

（2）需要检测目标存储卡是否已经接上，这点可以在 PCMCIA interface Input Pis Register（PIPR）里读到。如果已经接上，清空 PCMCIA interface Status Change Register（PSCR），以免在初始化时，遇上中断；接着，使能 PCMCIA interface Enable Register（PER），这是为了启动接口。然后，根据目标卡的时序要求，通过改变 PCMCIA interface General Control Register B（PGCRB）的值，使 reset 引脚产生要求的 reset 信号，如果没有 reset 信

号，PC 卡不能工作；

（3）启动 nREG 作为 GPIO 口使用，用来作为 rd/w 信号的标志信号。也就是说，使这个引脚在写的时候置低，读的时候置高；

（4）注册 PCMCIA 的主设备号。

一旦正确完成了 init 函数，读写将变得十分简单，读者只要仔细阅读下面的例子就会明白是怎么做到的。比如说，如果已经在 pcmcia.h 文件里设定了一个未附值的词组 hyy[3]，只需要在 write ()里写上以下的代码就可以完成。

```
hyy = (char 3 ) PCMCIA-BASE;
for ( ii = 0 ; ii < 3 ; ii + + )
hyy[ ii ] = ii ;
```

然后在 read()中写入以下的代码。

```
for ( ii = 0 ; ii < 3 ; ii + + )
printk("the hyy[ %d ] is %d ",ii ,hyy[ ii ]) ;
```

会发现读出的数值和写入的数值相同，说明对 PC 卡的读写是正确的。但同时也发现，如果没有在 pcmcia.h 文件里预先设定这么一个数组，将无法做到对属于 PCMCIA 存储空间的地址的读写，容易产生错误。

16.3　实例——基于PCMCIA的无线网络嵌入式前端系统设计

PCMCIA 接口有支持无线网卡通信标准 PCMCIA 接口、CF 卡等。PCMCIA 定义的物理层包括主机（Host）和 PC 卡（PC Card），接口为 68 脚，称为一个“插槽”。在大多数提供了 PC 卡插槽的计算机或嵌入式系统中，同时采用了“卡和插槽服务”（Card and Socket Services）软件，即在计算机或嵌入式系统与 PC 卡之间提供一个标准化的软件接口，将“卡和插槽服务”的功能直接集成到 Linux 操作系统内部。

16.3.1　系统基本结构

1．ISA-to-PCMCIA 转接 PD6710

因为 ARM 嵌入式处理器没有 PCMCIA 接口，为了连接 PCMCIA 接口的无线网卡，必须采用一个专用的芯片作为转接。这里采用了 ISA-to-PC-Card 控制器 PD6710（如图 16-5 所示）。PD6710 能够控制一个 PC-Card 插槽，兼容 PC 卡标准、PCMCDk2.1 标准。基于专业的 CF/PCMCIA 卡控制逻辑芯片 PD6710 的无线网络系统，支持 DMA 模式，在 WinCE 和 Linux 操作系统上轻松实现文件管理功能，并访问 CF 卡，具有很强的实用功能。

PD6710 采用有效功率和混合电压技术减少系统的功耗，同时该芯片具有软控制挂起模式和硬件超级挂起模式以实现低功耗的控制。该芯片采用 82365SL 兼容的寄存器组，具有 5 个可编程的存储区和两个可编程的 I/O 区。PD6710 与 ARM 连接端采用 8 位或 16 位的类 ISA 系统总线，与无线网卡连接端采用 8 位或 16 位 PC 卡总线接口。

PD6710 提供了 SA 到 PCMCIA 接口的转化，无线网络控制器总线接口逻辑部分通过 PD6710 无线网络总线侧信号连接到无线网控制器的处理与控制单元。

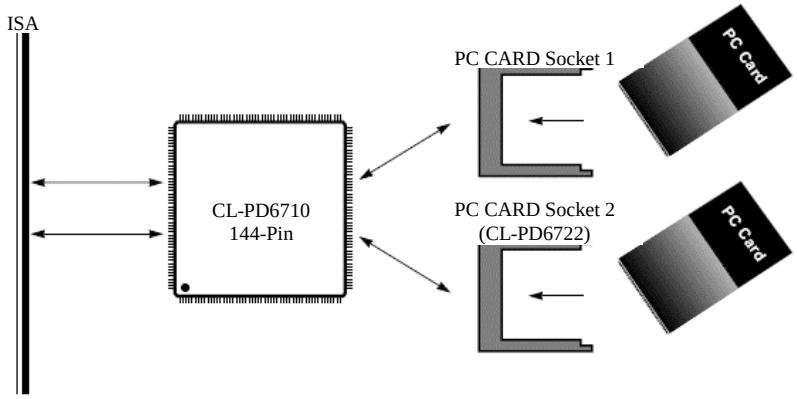


图 16-5　基于 PD6710 模块图

2．PD6710 功能框图

PD6710 功能框图如图 16-6 所示。其中主要的外部接口信号有以下几种。

- 电源：3.3V、5V、12V。
- 片选：A24、nOE、nWE。其中 nOE、nWE、A24 分别是 LnOE、LnWE、ADDR24 经过 245 后的信号。
- nRESET：直接来自核心板。
- EINT8，EINT3：给核心板 CPU 的中断。
- nWAIT：核心板 nWAIT 经过上拉以后的总线延迟信号。

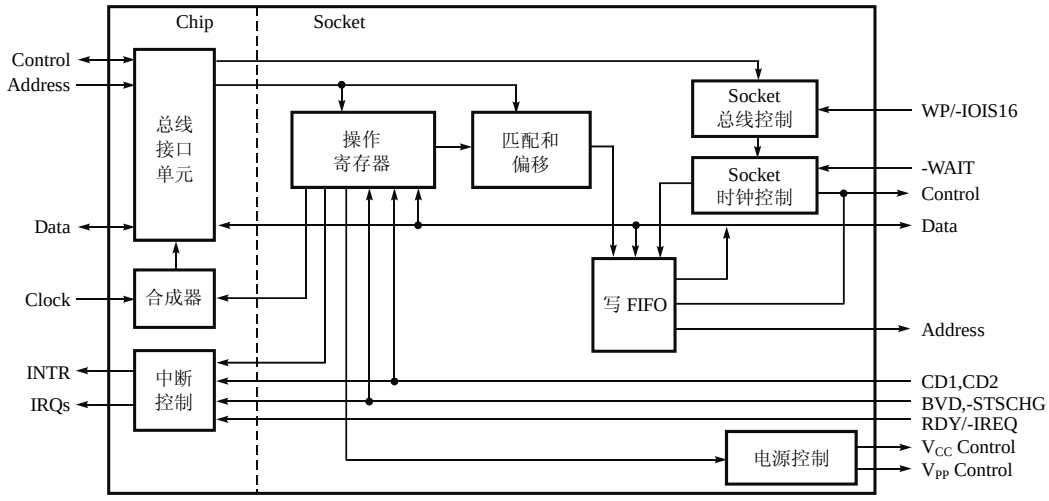


图 16-6　PD6710 功能框图

3．系统硬件电路结构图

在如图 16-7 所示的 PD6710 硬件电路图中，信号中既包括数据与地址线、中断请求与响应、读写选择、复位、片选等常用信号，还包括一些特殊用途的信号，如 16 位传输选择、写保护、状态改变输入、DMA 请求和响应、输入输出信道状态指示等，这些信号是 PCMCIA 接口所必需的。

另外，该接口芯片还需要一些外围电路，如芯片的配置电路（可采用 EEPROM）、复位电路、系统时钟等。

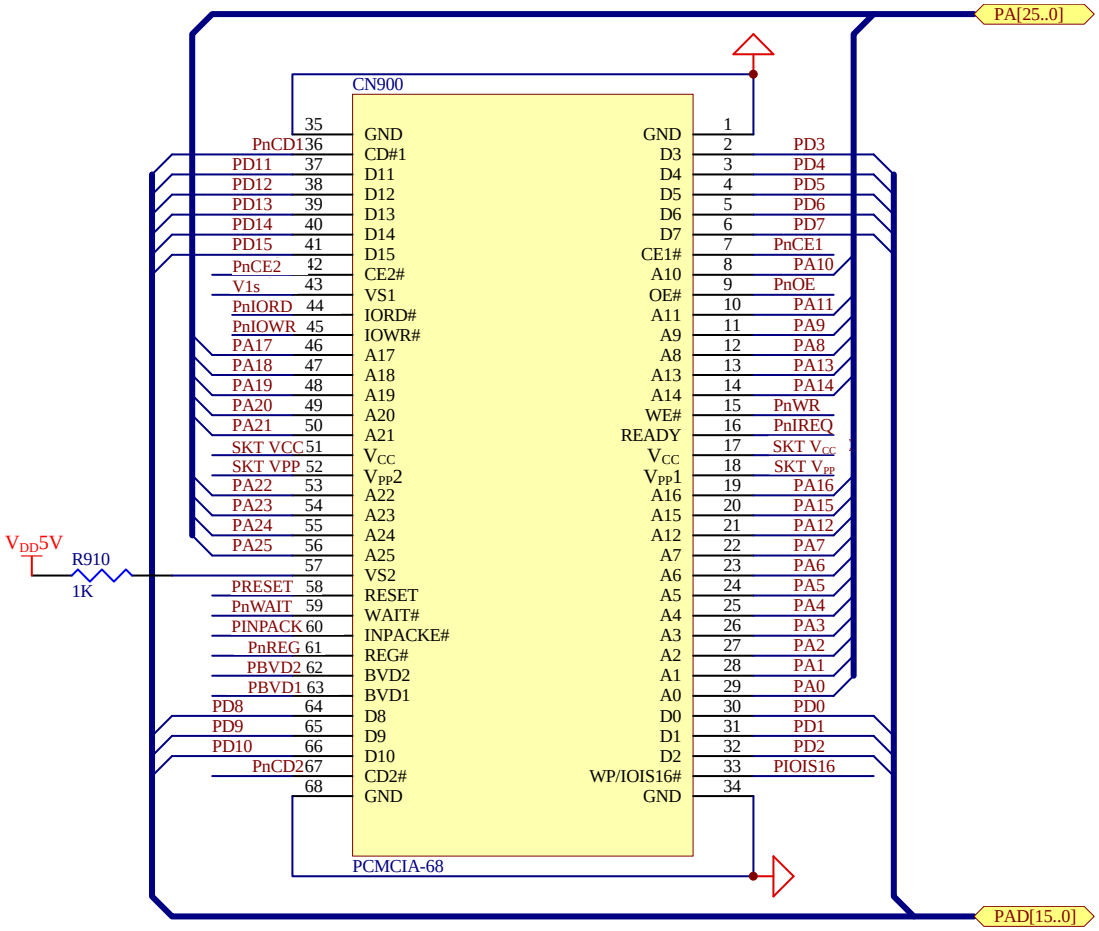


图 16-7　PD7610 硬件电路原理图

16.3.2　系统工作流程

嵌入式系统实现无线上网的技术关键是使驱动 PCMCIA 模块登录到 ISP，与 ISP 建立有效的 PPP 连接，进行数据传输。这一过程大致可以分为以下 3 个阶段。

（1）初始化。

系统上电后，首先对 PCMCIA 接口进行初始化，初始化流程如图 16-8 所示。系统加电后，系统将完成插槽初始化、PC Card 检测和 PC Card 配置。

插槽初始化使接口能够允许 PC Card 插入并对它进行配置。配置软件主要完成以下任务：设置插槽状态寄存器、触发状态改变中断、通知 PCMCIA 软件插槽的状态已改变等。

当系统检测到插入的 PC Card 后，它将产生一个状态改变中断，唤醒 PCMCIA 软件来读取插槽状态寄存器的内容，从而确定中断原因。

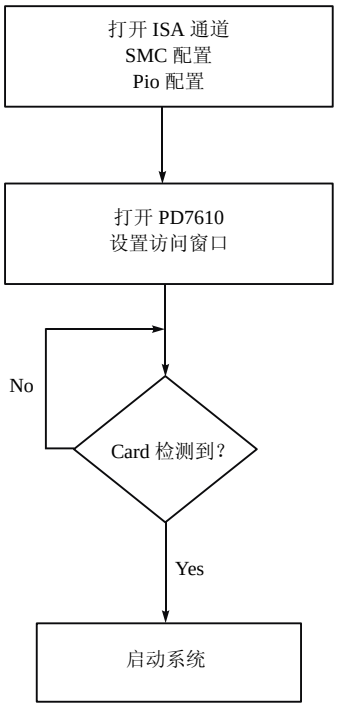


图 16-8　PD7610 初始化流程

如果初始化成功，则可以对 PCMCIA 模块使用命令进行配置。主要的 PCMCIA 配置包括以下几项：设置通信波特率；定义上下文，包括指定协议类型和 ISP 名称等；设置 ISP 的连接信息并拨号。

（2）建立网络连接。

拨号后，使用 PPP 协议与 ISP 进行连接。PPP 协议建立连接主要有以下几个步骤：进行协商，协商的内容主要包括 RFC1661 中所定义的选项；验证，以认证用户身份；网络阶段协商，在 IP 接入中主要是 IP 控制协议协商，如 IP 地址和 DNS 地址的协商等。如采用 IPv6，协商的内容与 IPv4 会有所不同。如表 16-1 所示是 IPv4 与 IPv6 协商内容的比较。

表 16-1　IPv4CP 与 IPv6CP 协商内容的比较

协　　议	地址内容	压缩协议
IPv4	IP Address	IP - Compression - Protocol
IPv6	Interface - Identifier	IPv6 - Compression - Protocol

如果协商成功，则链路建立成功，可以开始传输网络层数据报文。

（3）数据传输。

PPP 链路建立起来后，系统就可以通过 WLAN 与因特网上的目的主机交换 IP 数据报了。当数据传输结束后，通过 PPP 协议发出终止请求分组，请求终止链路连接，拆除建立的数据链接。整个实现流程如图 16-9 所示。

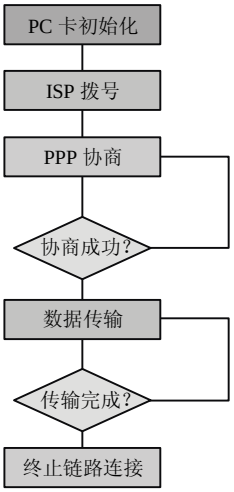


图 16-9　无线数据传输的实现流程

16.3.3　系统模块源代码实现

1．PCMCIA 模块配置

在内核代码中需要设置 makefile 文件，主要设置两个地方 ARCH 和 CROSS_COMPILE。

```
ARCH ：=arm ；  表示目标板为 arm
CROSS_COMPILE = 交叉编译工具的地址 ；设置交叉编译工具的地址
```

内核配置可采用图形界面和菜单界面，具体操作命令是 Make xconfig 和 makemenuconfig。

在设计中如果采用菜单界面来进行内核裁剪，在内核配置中，一般有 4 种选择：Y（选中），N（不选），M（模块）和数字。一般来说，用户可以根据需要进行配置。具体的操作命令如下。

```
# cd /linuette/target/box
# tar jxvf linux-2.4.18-rmk7-pxa1-mz4.tar.bz2
# cd 2.4.18-rmk7-pxa1-mz4
# make menuconfig
# make zImage
# make modules
# make modules_install
```

如果编译成功，那么将在 2.4.18-rmk7-pxa1-mz4/arch/arm/boot directory 文件夹下生成 zImage 文件。

```
Select "Load on Alternate Configuration File" menu, then write
"arch/arm/def-configs/sdk" or "arch/arm/def-configs/arm -tk"
                                        ;这也是 default 文件的选择好的选项
# tar cvf mod.tgz kernel pcmcia         ;这只是一个简单的映射
# cd 2.4.18-rmk7-pxa1-mz4/arch/arm/boot/
# cp zImage /image                      ;这里只是为了下载时方便
```

2．PCMCIA 接口程序

PCMCIA 软件在读取卡配置寄存器的内容后，确定 PC Card 所需资源并对它进行配置。

软件设计包括两个方面，一方面是 PC Card 侧的 ARM 控制器对 CIS 的初始化程序；另一方面是 PC 侧驱动程序。

PC Card 的 ARM 控制器对 CIS 的初始化程序的主要问题是：在 PC Card 正常工作前，必须对设备进行初始化，即将 CIS 信息写入属性空间，从而经过主机识别，使主机明确 PC Card 的应用类型。通常情况下，在将 CIS 写入 PC Card 前，可以利用程序测试 CIS 的兼容性。

详细程序如下：

```
#include "pd6710.h"
/*
    A24=1, I/O
    A24=0, 存储
    AEN=nGCS2,
    绝对地址：
    0x10000000~0x10FFFFFF: 存储区域
    0x11000000~0x11FFFFFF: I/O 区域
*/
int  PD6710_Init(void);
void PD6710_InitBoard(void);
void PD6710_InitInterrupt(void);
void PD6710_CardEnable(void);
void PD6710_Wr(U8 index, U8 data);
U8 PD6710_Rd(U8 index);
void PD6710_Modify(U8 index,U8 mask,U8 data);
void PD6710_CommonMemAccess(void);
void PD6710_AttrMemAccess(void);
U8 Card_RdAttrMem(U32 memaddr);
U8 Card_RdIO(U32 ioaddr);
void Card_WrIO(U32 ioaddr,U8 data);
void PrintCIS(void);
void __irq IsrPD6710Card(void);
void __irq IsrPD6710Management(void);
volatile int isCardInsert=0;
void _dbg(int i)
{
   Uart_Printf("<%d>",i);
}
void Test_PD6710(void)
{
   Uart_Printf(" [PD6710 test for reading pc_card CIS]\n");
   Uart_Printf("Insert PC card!!!\n");
   PD6710_InitBoard();
   PD6710_Init();
   PD6710_InitInterrupt();
                                    //如果 nCD1,2 为低，说明 PCMCIA 插入
                                    //这种情况下，不出现中断管理
   if((PD6710_Rd(INTERFACE_STATUS)&0xc)==0xc)
   {
        Uart_Printf("Card is inserted.\n");
        Delay(2000);
                                    //为了保持系统稳定，需要一定时延
                                    //如果没有时延，一些 CF card 可能检测不到
        PD6710_CardEnable();
```

```
        PrintCIS();
    }
    Uart_Printf("Waiting now.\n");

    while(1)
    {
        Uart_Printf(".");
        Delay(10000);
    }
}
                                                    //IRQ 堆栈扩展到 4096 byte
void PrintCIS(void)
{
    int i,j;
    U32 cisEnd=0;
    static U8 str[16];
    U8 c;
    Uart_Printf(" [Card Information Structure]\n");

    PD6710_AttrMemAccess();
                                                    //查找 CIS 尾部
    while(1)
    {
            c=Card_RdAttrMem((cisEnd)*2);
            Uart_Printf("c=%x,cisEnd=%d",c,cisEnd);
        if(c==0xff)                                 //0xff= 终止
            break;
        cisEnd++;
        cisEnd+=Card_RdAttrMem((cisEnd)*2)+1;
    }
    Uart_Printf("cisEnd=0~%x\n",cisEnd);
    for(i=0;i<=cisEnd*2;i+=2)
    {
        c=Card_RdAttrMem(i);
        str[(i%0x20)/2]=c;
        Uart_Printf("%2x,",c);
        if((i%0x20)>=0x1e)
        {
            Uart_Printf("//");
            for(j=0;j<0x10;j++)
                if(str[j]>=' ' && str[j]<=127)
                    Uart_Printf("%c",str[j]);
            else Uart_Printf(".");

            Uart_Printf("\n");
        }
    }
    Uart_Printf("\n");
}
#define B6710_Tacs  (0x0)               // 0clk
#define B6710_Tcos  (0x3)               // 4clk
#define B6710_Tacc  (0x7)               // 14clk
#define B6710_Tcoh  (0x1)               // 1clk
#define B6710_Tah   (0x0)               // 0clk
#define B6710_Tacp  (0x3)               // 6clk
#define B6710_PMC   (0x0)               // 正常(1data)
```

```
    void PD6710_InitBoard(void)
    {
                                                        //硬件板上初始化 PD6710
        rGPFCON=rGPFCON&~(3<<6)|(2<<6);
        rGPGCON=rGPGCON&~(3<<0)|(2<<0);
        rBWSCON=rBWSCON&~(0xf<<8)|(0xd<<8);
                                        //nGCS2=nUB/nLB(nSBHE),nWAIT,16bit
      rBANKCON2=((B6710_Tacs<<13)+(B6710_Tcos<<11)+(B6710_Tacc<<8)+(B6710_
Tcoh<<6)\
            +(B6710_Tah<<4)+(B6710_Tacp<<2)+(B6710_PMC));
    }
    void PD6710_InitInterrupt(void)
    {
        rEXTINT0=rEXTINT0&~(7<<12)|(2<<12);
                                                        //EINT3(GPF3)下降沿
        rEXTINT1=rEXTINT1&~(7<<0)|(4<<0);
                                                        //EINT8(GPG0)上升沿
        pISR_EINT3=(U32)IsrPD6710Management;            //nINT_P_CON
        pISR_EINT8_23=(U32)IsrPD6710Card;               //nINT_P_DEV
        rSRCPND = BIT_EINT3|BIT_EINT8_23;               //清除 pending 状态
        rINTPND = BIT_EINT3|BIT_EINT8_23;
        rINTMSK=~(BIT_EINT3|BIT_EINT8_23);
        rEINTMASK=rEINTMASK&~(1<<8)|(0<<8);             //EINTMASK[8]=中断使能
    }
    int PD6710_Init(void)
    {
                                                        //初始化 PD-6710
        PD6710_Wr(POWER_CTRL,(0<<7)|(1<<5)|(0<<4)|(0<<0));
                                                        //电源 Vpp1=0V

        PD6710_Wr(INT_GENERAL_CTRL,(1<<7)|(0<<5)|(1<<4)|(3<<0));

                                    //manage_int=-INTR pin, nINT_P_DEV=IRQ3
        PD6710_Wr(MANAGEMENT_INT_CONFIG,(1<<0)|(1<<3)|(3<<4));
        PD6710_Wr(IO_WINDOW_CTRL,0|(0<<3));
                                    //IO0=8bit,timing_set_0
                                    // I/O 窗口不能包括 3e0h and 3e1h
                                    // IO 内容=0x3f8~0x3ff(COM1) ->0x3f8~0x3ff
        PD6710_Wr(SYS_IO_MAP0_START_L,0xf8);
        PD6710_Wr(SYS_IO_MAP0_START_H,0x3);
        PD6710_Wr(SYS_IO_MAP0_END_L,0xff);
        PD6710_Wr(SYS_IO_MAP0_END_H,0x3);
        PD6710_Wr(CARD_IO_MAP0_OFFSET_L,0x0);
        PD6710_Wr(CARD_IO_MAP0_OFFSET_H,0x0);
                                                        //存储窗口, 最小 64KB
        PD6710_Wr(SYS_MEM_MAP0_START_L,0x0);            //MEM0=8bit 数据宽度

        PD6710_Wr(SYS_MEM_MAP0_START_H,0x0);
        PD6710_Wr(SYS_MEM_MAP0_END_L,0x0f);             //0x0 ~ 0xffff
        PD6710_Wr(SYS_MEM_MAP0_END_H,0x0|(0<<6));       //timing_set_0
        PD6710_Wr(CARD_MEM_MAP0_OFFSET_L,0x0);
        PD6710_Wr(CARD_MEM_MAP0_OFFSET_H,0x0|(1<<6));   //nREG=active
                                    //MEM AREA=0x0~0xFFFF ->0x0~0xFFFFFF
        PD6710_Wr(MAPPING_ENABLE,1|(1<<6));
                                        //memory map 0, 使能 I/O map 0 使能
        PD6710_Wr(MISC_CTRL1,(0<<7)|(1<<4)|(1<<3)|(1<<2)|(1<<1));
```

```
                                        // nVCC_3_使能(暂时)
        PD6710_Wr(MISC_CTRL2,1|(1<<1)|(1<<4));
                                        //25Mhz,IRQ12=drive_LED
        PD6710_Wr(FIFO_CTRL,0x80);      // FIFO
                                        //配置时钟寄存器前，FIFO 应清零
                                        //默认访问时间为 300ns

    PD6710_Wr(SETUP_TIMING0,0x2); PD6710_Wr(CMD_TIMING0,0x8); //25Mhz 时钟
        PD6710_Wr(RECOVERY_TIMING0,0x2);

        Delay(1000);
        Uart_Printf("INT_GENERAL_CTRL=%x\n",PD6710_Rd(INT_GENERAL_CTRL));
        Uart_Printf("MANAGEMENT_INT_CONFIG=%x\n",PD6710_Rd(MANAGEMENT_INT_
CONFIG));
        Uart_Printf("SYS_IO_MAP0_END_L=%x\n",PD6710_Rd(SYS_IO_MAP0_END_L));
        Uart_Printf("POWER_CTRL=%x\n",PD6710_Rd(POWER_CTRL));
        Uart_Printf("MISC_CTRL1=%x\n",PD6710_Rd(MISC_CTRL1));
        Uart_Printf("SYS_IO_MAP0_START_L=%x\n",PD6710_Rd(SYS_IO_MAP0_START_L));
        Uart_Printf("IO_WINDOW_CTRL=%x\n",PD6710_Rd(IO_WINDOW_CTRL));
        Uart_Printf("CARD_IO_MAP0_OFFSET_H=%x\n",PD6710_Rd(CARD_IO_MAP0_
OFFSET_H));
        Uart_Printf("CARD_IO_MAP0_OFFSET_L=%x\n",PD6710_Rd(CARD_IO_MAP0_
OFFSET_L));
        Uart_Printf("SYS_IO_MAP0_END_H=%x\n",PD6710_Rd(SYS_IO_MAP0_END_H));
        Uart_Printf("SYS_IO_MAP0_START_H=%x\n",PD6710_Rd(SYS_IO_MAP0_START
_H));
        Uart_Printf("SYS_MEM_MAP0_START_L=%x\n",PD6710_Rd(SYS_MEM_MAP0_START
_L));
        Uart_Printf("SYS_MEM_MAP0_END_L=%x\n",PD6710_Rd(SYS_MEM_MAP0_END_L));
        Uart_Printf("SYS_MEM_MAP0_END_H=%x\n",PD6710_Rd(SYS_MEM_MAP0_END_H));
        Uart_Printf("SYS_MEM_MAP0_START_H=%x\n",PD6710_Rd(SYS_MEM_MAP0_START
_H));
        Uart_Printf("CARD_MEM_MAP0_OFFSET_L=%x\n",PD6710_Rd(CARD_MEM_MAP0_
OFFSET_L));
        Uart_Printf("CARD_MEM_MAP0_OFFSET_H=%x\n",PD6710_Rd(CARD_MEM_MAP0_
OFFSET_H));

        Uart_Printf("MAPPING_ENABLE=%x\n",PD6710_Rd(MAPPING_ENABLE));
        Uart_Printf("MISC_CTRL2=%x\n",PD6710_Rd(MISC_CTRL2));
        Uart_Printf("FIFO_CTRL=%x\n",PD6710_Rd(FIFO_CTRL));
        Uart_Printf("SETUP_TIMING0=%x\n",PD6710_Rd(SETUP_TIMING0));
        Uart_Printf("CMD_TIMING0=%x\n",PD6710_Rd(CMD_TIMING0));
        Uart_Printf("RECOVERY_TIMING0=%x\n",PD6710_Rd(RECOVERY_TIMING0));

        PD6710_Wr(CHIP_INFO,0x0);
        if((PD6710_Rd(CHIP_INFO)&0xc0)!=0xc0 || (PD6710_Rd(CHIP_INFO)&0xc0)!=
0x00 )
        {
             Uart_Printf("PD6710 hardware identification error!!!\n");
             return 0;
        }
        return 1;
    }
    void PD6710_CardEnable(void)            //中断之后
    {
```

```
    if(PD6710_Rd(MISC_CTRL1)&0x1)   //MISC_CTRL1[0] 0=3.3V  1=5.0V
    {
        PD6710_Modify(MISC_CTRL1,0x2,0x0);          //nVcc_5 使能
        Uart_Printf("5.0V card is detected.\n");
    }

     else
    {
        PD6710_Modify(MISC_CTRL1,0x2,0x2); //nVCC_3 使能
        Uart_Printf("3.3V card is detected.\n");
    }
    PD6710_Modify(POWER_CTRL,(1<<4)|3,(1<<4)|1);
                                    //VCC_POWER_on,Vpp=Vcc(3.3V 或 5.0V)
    Delay(100);

    Delay(10);                      //RESET 应 Hi-Z 状态最少 10ms
    PD6710_Modify(POWER_CTRL,(1<<7),(1<<7));

    PD6710_Modify(INT_GENERAL_CTRL,(1<<6),0);       //RESET=active(高电平)

    PD6710_Modify(INT_GENERAL_CTRL,(1<<6),(1<<6));  //RESET=inactive(低电平)

    Delay(200);                     //等待 200ms

     PD6710_Modify(INT_GENERAL_CTRL,(1<<5),(1<<5));
                                    //mem_card -> mem_io_card

    Delay(5000);                    //等待 500ms

    Uart_Printf("Card Enable!\n");
    Uart_Printf("INT_GENERAL_CTRL=%x\n",PD6710_Rd(INT_GENERAL_CTRL));
    Uart_Printf("POWER_CTRL=%x\n",PD6710_Rd(INT_GENERAL_CTRL));

                                    //如果没有时延，一些 CF card 可能检测不到
}
void PD6710_Wr(U8 index, U8 data)
{
                                    //nREG 为低
    rPD6710_INDEX=index;
    rPD6710_DATA=data;
}
U8 PD6710_Rd(U8 index)
{
                                    //nREG 为低
    rPD6710_INDEX=index;
    return rPD6710_DATA;
}
void PD6710_Modify(U8 index,U8 mask,U8 data)
{
                                    //nREG 为低
    rPD6710_INDEX=index;
    rPD6710_DATA=(rPD6710_DATA)&~mask|data;
}
U8 Card_RdAttrMem(U32 memaddr)
{
```

```
                                                    //nREG 为低
        return *((volatile U8 *)(PD6710_MEM_BASE_ADDRESS+memaddr));
    }
    U8 Card_RdIO(U32 ioaddr)
    {
        return *((volatile U8 *)(PD6710_IO_BASE_ADDRESS+ioaddr));
    }
    void Card_WrIO(U32 ioaddr,U8 data)
    {
        *((volatile U8 *)(PD6710_IO_BASE_ADDRESS+ioaddr))=data;
    }
    /*
         访问 I/O: nREG 为低 (自动)
         访问 contribute 存储: nREG 为低
         访问 common 存储: nREG 为高
    */
    void PD6710_CommonMemAccess(void)
    {
        PD6710_Modify(CARD_MEM_MAP0_OFFSET_H,(1<<6),(0<<6)); //nREG=inactive, H
    }
    void PD6710_AttrMemAccess(void)
    {
        PD6710_Modify(CARD_MEM_MAP0_OFFSET_H,(1<<6),(1<<6)); //nREG=active, L
        Uart_Printf("CARD_MEM_MAP0_OFFSET_H=%x\n",PD6710_Rd(CARD_MEM_MAP0_
OFFSET_H));
    }
    void __irq IsrPD6710Management(void)    //nINT_P_CON
    {
        U8 cardStat;
                                            //ClearPending(BIT_EINT7);
        Uart_Printf("\nPD6710 interrupt is occurred.\n");

        Delay(2000);
                                            //为了保持系统稳定，需要一定时延
                                            //如果没有时延，一些 CF card 可能检测不到

        cardStat=PD6710_Rd(CARD_STAT_CHANGE);
                                            //PCMCIA Card 状态改变否
        if(cardStat&0x8)
        {
                                            //校验 CD1,2 是否低电平
            if((PD6710_Rd(INTERFACE_STATUS)&0xc)==0xc)
            {
                Uart_Printf("Card is inserted.\n");
                PD6710_CardEnable();
                PrintCIS();
            }
            else
            {
                Uart_Printf("Card is ejected.\n");
                PD6710_Init();              //可以拔出
            }
        }
        ClearPending(BIT_EINT3);
                                            // 中断，int pending 在 ISR 结束时清零
    }
```

```
void __irq IsrPD6710Card(void)          //nINT_P_DEV
{
    rEINTPEND=(1<<8);                   //EINTPEND[8]清空
    ClearPending(BIT_EINT8_23);
    Uart_Printf("PC card interrupt is occurred.\n");
}
```

16.3.4 系统调试

DNW 是一款基于 Windows 的开发工具，提供图形界面，利用该软件包可以通过串口和 USB 将编译后的程序下载到目标板上调试 PCMCIA 程序。其工作界面如图 16-10 所示。

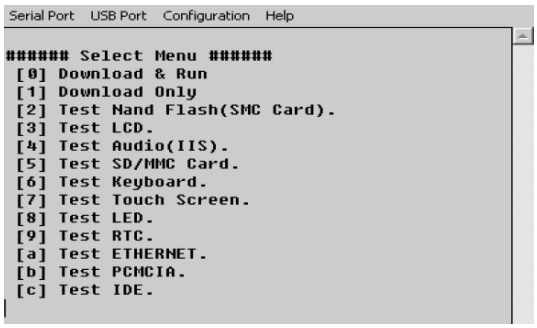


图 16-10　DNW 开发环境

16.4 本章总结

随着功能强大的便携式数据终端以及多媒体终端的应用逐渐深入，为了实现在任何时间、任何地点均能实现数据通信的目标，就要求传统的计算机网络由有线向无线，由固定向移动，由单一业务向多媒体发展，也就更进一步推动了 WLAN 的发展。

鉴于人们对嵌入式设备在智能化和互连性上的需求，嵌入式 Linux 系统在无线网络控制和通信领域的应用也将越来越广泛。本章详细介绍了 PCMCIA 无线网络数据传输系统开发的过程，重点通过利用外部 PD6710 电路设计 PCMCIA 接口，实现与无线网卡的软、硬件接口。通过本章的学习，读者可以领会设计思想，从而自行使用基于嵌入式 Linux 系统来开发无线网络产品。

第 17 章

基于 PDIUSBD12 的数据传输系统实例

常用的主机与嵌入式外设的高速通信接口有 LPT 并行口、USB 接口、1394 接口及 10/100M 以太网接口等。RS-232 接口不适合高速数据传送，1394 接口需要专门的适配器接口，成本过高，一般较少使用。而 USB（Universal Serial Bus，通用串行总线）接口是 1994 年 Intel、Microsoft 等多家公司联合推出的计算机外设互连总线协议。

USB 接口现在已被广泛用于高、中、低不同速度设备与主机通信，USB2.0 的最高速度可达 480Mbps，可传送高清晰数字视频码流，完全可以替代 1394 接口。USB 接口与以太网接口相比，采用主从节点。USB 接口方案对比 RS-232 串行口，大大提高了嵌入式系统的数据吞吐能力，与以太网接口相比有即插即用特性、有块和同步等多种数据传输模式，更适合音视频码流传送。

如图 17-1 所示是基于嵌入式 Linux 的 USB 音频应用系统模型。

从普通计算机用户、计算机工程师到硬件芯片生产厂商，都已经完全认可了 USB。厂商对于 USB 的硬件和软件支持也越来越完备，现在开发一个 USB 外设产品，所需要投入的成本和时间大大降低了，几年前没有办法做到这一点。但是，随着 USB 应用领域的逐渐扩大，人们对于 USB 的期望也越来越高，希望 USB 能应用在各种计算机领域中，尤其是在移动通信领域中，希望能通过 PDA 等移动设备直接和 USB 外设通信，使 USB 能应用在没有 PC 的领域中。

但在非 PC 领域应用正是 USB 一个致命的弱点。USB 的拓扑结构中居于核心地位的是 Host（主机），任何一次 USB 的数据传输都必须由 Host 发起和控制，所有的 USB 外设都只能和 Host 建立连接，任何两个外设之间或是两个 Host 之间无法直接通信。而目前大量扮演 Host 角色的是 PC。由此可见，“如何将 USB 应用到嵌入式领域？如何实现 USB 点对点的通信？”等问题，开始进入 USB 开发的讨论议程。正是在这种新的需求之下，USB Host 的嵌入式应用成为 USB 领域新的兴奋点。

传统意义的 USB 开发，仅仅是对 USB 外设的开发，USB 底层驱动程序和 USB 主控制器驱动程序都由 Windows 等操作系统提供，有关这些驱动程序的细节过程都不开放源代码。要设计 USB Host，就必须设计这两部分驱动程序，而 Windows 源码不公开，这些细节资料就无从得到，所以只有从开放源代码 Linux 的角度设计。

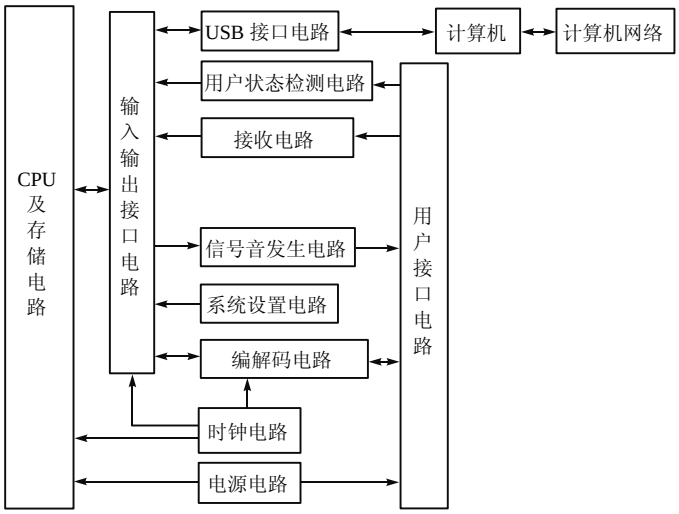


图 17-1　基于嵌入式系统的 USB 接口

17.1　USB 应用环境与硬件设计概要

以 Linux 为操作系统的嵌入式系统已大量普及，在其上开发 USB 驱动的需求也越来越大。在基于嵌入式 Linux 的 USB 开发中，要求同时提供 Host 和 Device 两种接口。通常 Host 主机端控制所有的传输，而外设（如数码相机等）作为 Device 端实现不同的功能。

如图 17-2 所示是嵌入式 USB 系统框图，图中虚线箭头表示逻辑连接；实线箭头表示物理连接和实际通信。操作系统是 Linux 2.4.18 内核，而 ARM 本身带有支持 USB1.1 的 Host 和 Device 硬件接口。只要实现 USB Host 总线和 USB Device 总线驱动，就可以在 USB Device 总线驱动上实现 USB 存储功能以及 USB 数据传输。USB 接口支持 1.5Mbps、12Mbps 和 480Mbps 的数据传输速率，支持控制、中断、批量与实时 4 种数据传输模式，让外围设备可以有弹性地选择。不管是交换少量或是大量的数据，还是有无时效的限制，都有合适的传输类型。因此 USB 的实时同步数据传输模式适合于高速实时音视频数据流的传送。

图 17-2 系统由以下几个部分组成。

（1）核心控制器部分，主要包括以下几个功能模块。

- USB Host 模块：包括 USB Host 驱动程序（用于配置和实现 USB Host 功能）和 USB Host 控制器接口和驱动程序（用于物理连接和通信，并进行配置）；
- USB 设备模块：包括 USB 设备用户应用程序（在 ARM 上实现对外围 USB 设备的读取，通常是一些 API（应用程序接口），用于读、写、格式化等命令）、USB 海量存储 FAT32 文件系统（按照 Windows 标准，编写外围连接的 USB Flsah 存储器的文件系统，增强了系统的通用性和扩展性）和 USB Flsah 设备驱动程序（通常 PC 平台上开发的 USB 外设的驱动程序配置 USB 外设，实现 USB 通信，实现 USB Host 要求的各项配置和数据传输的要求，接受 USB Host 的命令来管理 USB 外设）。

（2）USB Host 控制器：（核心逻辑控制部分，实现各种 USB Host 动作，协调内容各项功能，与外围主控制器通信），RAM 缓冲区和系统寄存器（存放传输的 USB 数据，设置 USB 传输特性）。

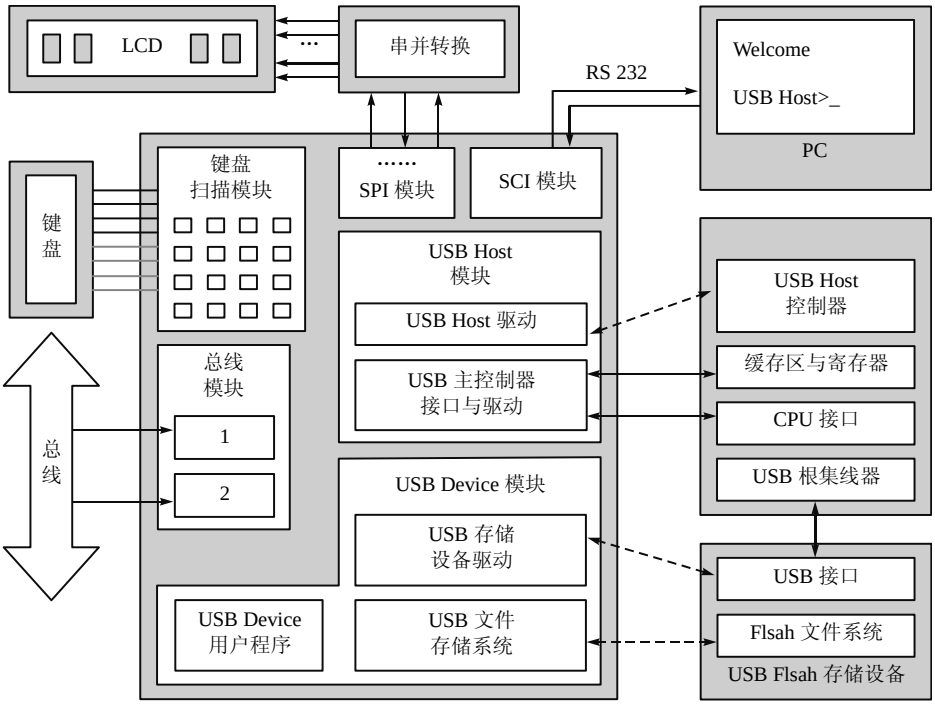


图 17-2　基于 USB 的应用系统

（3）键盘模块：通过中断方式、并口同 CPU 连接。用于系统独立运行时，键入命令，控制整个 USB Host 的工作。比如，按键 1 代表在 LCD 上显示 USB 外设的信息，按键 2 代表从系统中删除 USB 外设等。

（4）LCD 模块：显示 USB Host 系统的信息。通过 SPI 来控制。

（5）SCI 模块：通过 RS-232 与 PC 通信，用于系统调试。

17.2　相关开发技术——USB 系统与总线驱动

Linux 中每一个外围物理设备，如键盘、显示器、鼠标、网络适配器等，都有一个专门用于控制该设备的驱动程序。驱动程序实际上是函数和数据结构的集合，目的是实现设备管理接口，Linux 内核通过这个接口请求驱动程序控制设备进行 I/O 操作。Linux 系统中将设备分成字符设备、块设备和网络设备。块设备将数据分成可寻址的块来处理，块的大小通常为 512B 到 32 KB 不等。大多数块设备允许随机访问，并且通常使用缓存技术，USB 属于块设备。

17.2.1　USB 系统组成

简单来说，USB 系统包括两部分，模型如图 17-3 所示。

- USB Host，即 USB 主机；
- USB Device，即 USB 外设。

USB 系统的通信是由 USB Host 控制的。USB Host 不单纯指的是硬件，而是嵌入式或 PC 系统的软件和硬件的集合。这与以太网中的主机的概念有所不同。以太网中的主机

是一台负责控制网络通信，为网络终端提供服务的计算机。

可见，USB Host 是一个全新的概念。而设计嵌入式的 USB Host 尤其要深入把握和理解 USB 系统和主机的通信。

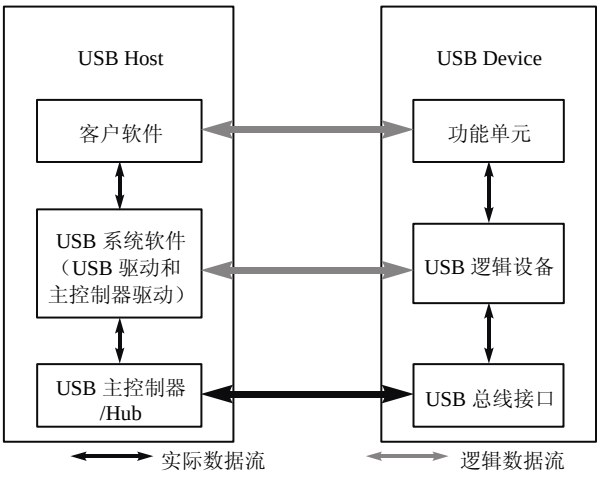


图 17-3　USB Host 与 Device 系统模型

USB 总线包括 4 根信号线，用于传送信号和提供电源，如图 17-4 所示。其中，D+和 D-为信号线，是一对双绞线；Vbus 和 GND 是电源线，提供电源（+5 V）。

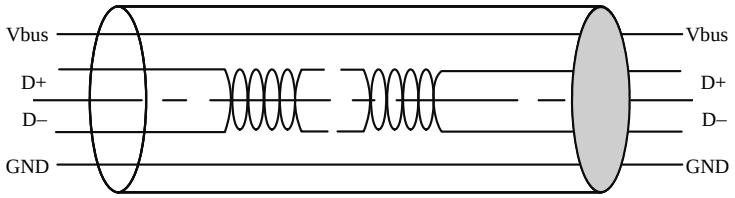


图 17-4　USB 总线

17.2.2　USB Host 总线驱动

USB 协议没有明确定义软件、硬件的分界面，其中 Host 在发展过程中形成了 OHCI（Open Host Controller Interface）和 UHCI（Universal Host Controller Interface）两种标准。OHCI 由 Compaq 等提出，大多数便携式计算机使用 OHCI 标准芯片；UHCI 由 Intel 公司开发，大多数台式 PC 使用这种接口标准。

ARM 的 USB Host 支持 OHCI 标准。OHCI 规定了一系列连续的寄存器作为硬件与软件交互的界面。虽然 USB OHCI 的标准在软件上实现非常复杂，但 Linux 内核已经包括了 OHCI 的机制实现部分。针对 ARM 嵌入式芯片，唯一需要对内核修改的就是指定 ARM 的 OHCI 寄存器基地址。

USB Host 端在 Linux 中的位置如图 17-5 所示。

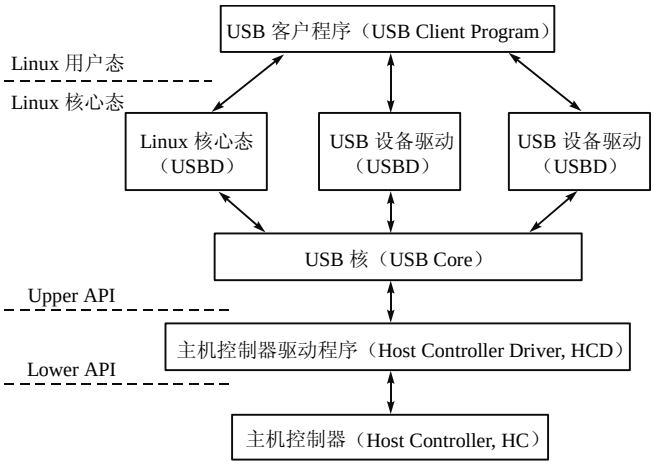


图 17-5　Linux 下的 USB 分层结构

其中，HCD 向 Linux 系统的 USBD 注册 USB 总线资源时提供了一个 usb_bus 数据结构，该结构中包含指向另一个非常重要的数据结构 usb_operations 的指针。usb_operations 结构定义了 HCD 所实现的 USB 总线操作接口，包括资源管理接口和数据传输（USB 总线上的数据都以 URB 传输）接口，USBD 正是借助这些接口来实现该总线上的设备管理和数据传输的。此结构的定义如下：

```
struct usb_operations
{
    int (*allocate)(struct usb_device*);        // HCD 资源分配函数
    int (*deallocate)(struct usb_device*);      // HCD 资源释放函数
    int (*get_frame_number)(struct usb_device *);
                                                // 读取 HC 帧号
    int (*submit_urb)(struct urb *);            // 提交 URB 给 HCD
    int (*unlink_urb)(struct urb *)             // 取消已提交的 URB
}
```

HCD 中的中断服务程序首先清除相应的中断悬挂位，以免被重复调用，然后根据产生中断的 USB 事件类型调用驱动程序中的相应的功能函数。

17.2.3　USB Device 总线驱动

在标准的 Linux 内核中不支持 USB Device 总线驱动，针对使用的 PDIUSBD12 芯片，必须开发全新的驱动模块。USB Device 分为驱动程序的注册和注销、设备的打开与释放、读写函数。

1．USB 设备注册与注销

进行注册（Register）和取消注册（Deregister）的函数如下：

```
int usb_register (struct usb_driver *drv);
/*用来向系统注册新的 USB 设备驱动。参数 drv 指向下面将要提到的结构 usb_driver */
void usb_deregister(struct usb_driver *drv);
/*每种 USB 设备都要有个 usb_driver 数据结构，这是对 USB 设备的最高层次上的抽象*/
struct usb_driver
{
```

```
        const char *name;                                   /* 模块名称*/
        void *(*probe) (struct usb_device *, unsigned int, const struct

        usb_device_id *id_table);                           /* 接入函数*/
        void (*disconnect)(struct usb_device *, void *);    /*移出函数*/

        struct list_head driver_list;
        struct file_operations *fops;       /* 驱动程序的文件操作列表*/

        int minor;                          /* 次设备号基数,次设备号为 16*基数*/
        struct semaphore serialize;

        int (*ioctl) (struct usb_device *dev, unsigned int code, void*buf);
        const struct usb_device_id *id_table;
    }
    /*以上的 USB 驱动框架为普通的设备驱动还设置了两个接入点*/
    void *(*probe) (struct usb_device *, unsigned int, const struct usb_device_id
*id_table);
    void  *probe(struct  usb-device  *dev,  unsigned  int  interface,const
structusb-device-id *id_table)
    {
        struct driver_ context *context;
        if(dev->descriptor.idVendor==0x0547 && dev->descriptor. idProduct==
0x2131 && interface==1)
        MOD-INC-USE-COUNT;
                /*为实例分配资源*/
        context=allocate_ driver_ resources();
        return context;

        return NULL;
    }
```

当一个新的设备被接到 USB 总线时，这个接入点的接入函数将被调用，然后设备驱动为新设备创建一个内部数据结构的实例。参数 dev 说明了设备内容，包含了所有 USB 描述符的指针。参数 interface 给出了接口号，如果一个 USB 驱动想要与某一设备和接口绑定，它必须返回一个指针，这个指针通常与设备驱动程序的内容结构有关。

```
    void (*disconnect)(struct usb_device *, void *);
    static void dabusb-disconnect (struct usb-device *usbdev, void*drv-context)
    {
                                                        /*取得指针*/
        struct driver_ context *s=drv_ context;
                                                        /*设置移出标志*/
        s ->remove-pending=1;
                                                        /*唤醒整个驱动程序*/
        wake-up (&s->wait);
        sleep-on (&s ->remove-ok);
                                                        /*释放资源*/
        free-driver-resources(s);
        MOD-DEC-USE-COUNT;
    }
```

2．打开（open）与释放（release）

open 方法是驱动程序用来为以后的操作完成初始化准备工作的。此外，open 还会增

加设备计数，以便防止文件在关闭前模块被卸载出内核。release 方法的作用正好与 open 相反。这个设备方法有时也称为 close。

```
static int open_mydevice (struct inode *inode,struct file *file)
static void release_mydevice (struct inode *inode, struct file*file)
```

3．读和写

read 方法的任务是将数据从设备复制到用户空间，write 方法则必须将数据从用户空间复制到设备。每一个 read 或 write 系统调用请求传输一定量的字节，但驱动程序可以随意传送其中一部分数据。

```
    static ssize_t read_mydevice (struct file *file, char *buffer,size_tcount,
loff t *ppos);
    static ssize_t write mydevice (struct file *file, const char *buffer,size_t
count, loff_t *ppos);
```

4．USB Device 设置

ARM 的 USB Device 接口支持 1 个 16 字节的双向控制端点（0 号端点）和 4 个 64 字节的双向批量（中断）传输端点。为了实现控制功能，它提供了大量的寄存器。对于 4 个批量（中断）传输端点，提供的寄存器都是一样的；而对于 0 号端点，除了不支持 DMA 传输以外，与其他端点的寄存器也基本一样。这些寄存器主要分为下面几类。

（1）控制状态寄存器。用于设定端点的各种属性，包括工作模式（批量还是中断），是否采用 DMA 传输，以及其他在传输过程中的细节设定。读取这类寄存器就可以得到当前端点传输的各种状态。

（2）中断设定（使能、屏蔽、等待），也可获知究竟是哪个端点发生了中断。

（3）FIFO 数据寄存器。需要传输的数据将连续写入该寄存器，实际上被保存到了一个 FIFO（先进先出）的缓冲区中，将被自动发送。同时，读取该寄存器将获得接收到的数据。

（4）DMA 传输寄存器。主要设定是否工作在 DMA 模式以及相应的细节。但是，具体 DMA 需要在系统的 DMA 控制器驱动里实现。

同时，还提供了一条中断请求线，用于在接收到数据或者发送完数据（以及其他定义需要中断的情况）时向系统发出中断请求。

在能够使用各个端点以前，必须通过 0 号端点进行配置，这也是 USB Device 接口与 PC 的 USB Host 接口连接后的第一个动作。通常使用有限状态自动机来完成这项配置工作（具体配置过程可以参见 USB1.1 规范）。

5．USB Device 数据发送

配置好通道以后，一个典型的数据发送过程如下。

ARM 的 1 号通道 FIFO 缓冲区寄存器为 EP1 _FIFO，长度设为最大值 64 字节，等待发送的数据开始地址为 buf，长度为 len。假设 len>64（len≤64 可当成以下过程的特例）。

（1）将 buf 开始的 64 字节依次赋值给 EP1_F IFO。

（2）设定控制寄存器开始发送。

（3）本次硬件发送完毕以后会发出中断，在中断处理函数中检查（len - = 64）是否大于 0。如果大于 0（未发送完），则发送下面的不超过 64 字节的数据，并返回（这会引起下一次中断）；如果已经发送完毕，那么调用用户给予的一个回调函数来完成用户指定的

工作（如开始接受回应的数据）。

通道已经建立，Device 总线驱动也就完成了。

17.3 实例——基于 PDIUSBD12 的数据传输设计

PDIUSBD12 芯片是飞利浦公司一款性能价格比很高的 USB 器件，通常用于基于微控制器的系统并与微控制器通过高速通用并行接口进行通信，也支持本地 DMA 传输。该器件采用模块化的方法实现一个 USB 接口，允许在众多可用的微控制器中选择最合适的作为系统微控制器，允许使用现存的体系结构并使固件投资减到最小。这种灵活性减少了开发时间、风险和成本，是开发低成本且高效的 USB 外围设备解决方案的一种有效途径。由于 PDIUSBD12 符合 USB1.1 规范，能适应大多数设备类规范的设计，如成像类、大容量存储类、通信类、打印类和人工输入设备等，因此 PDIUSBD12 非常适合做很多外围设备，如打印机、扫描仪、外部大容量存储器（Zip 驱动器）和数码相机等。现在用 SCSI 实现的很多设备如果用 USB 来实现可以直接降低成本。

17.3.1 系统基本结构

1．PDIUSBD12 组成

PDIUSBD12 功能强大，成本较低，应用比较广泛，8 位并行口可以与处理器直接连接，硬件上实现 USB 的底层协议能够满足系统的要求。

如图 17-6 所示，PDIUSBD12 是一个 28 脚的芯片，它的封装形式有两种：TSSOP28（塑料极小封装）28 脚，本体宽度 4.4mm。另一种封装是 S028（塑料小型封装）28，本体宽度 7.5mm，其管脚具体说明如表 17-1 所示。

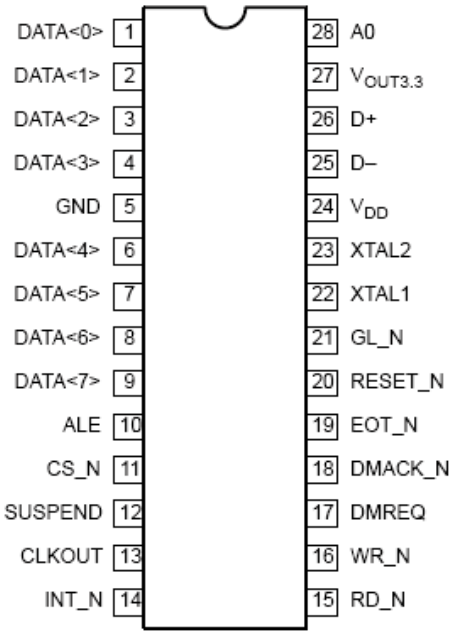


图 17-6　DS1820 管脚图

表 17-1　部分 PDIUSBD12 管脚列表

管　脚	符　号	类　型	描　述
1	DATA<0>	IO2	双向数据位 0
2	DATA<1>	IO2	双向数据位 1
3	DATA<2>	IO2	双向数据位 2
4	DATA<3>	IO2	双向数据位 3
5	GND	P	地
6	DATA<4>	IO2	双向数据位 4
7	DATA<5>	IO2	双向数据位 5
8	DATA<6>	IO2	双向数据位 6
9	DATA<7>	IO2	双向数据位 7
10	ALE	I	地址锁存使能。在多路地址/数据总线中，下降沿关闭地址信息锁存。将其固定为低电平用于单地址/数据总线配置
11	CS_N	I	片选（低有效）
12	SUSPEND	I,OD4	器件处于挂起状态
13	CLKOUT	O2	可编程时钟输出（MCU 时钟输出）
14	INT_N	OD4	中断（低有效）
15	RD_N	I	读选通（低有效）
16	WR_N	I	写选通（低有效）
17	DMREQ	O4	DMA 请求
18	DMACK_N	I	DMA 应答（低有效）
19	EOT_N	I	DMA 传输结束（低有效）。EOT_N 仅当 DMACK_N 和 RD_N 或 WR_N 一起激活时才有效
20	RESET_N	I	复位（低有效且不同步）。片内上电复位电路，该管脚可固定 V_{CC}
21	GL_N	OD8	GoodLink LED 指示器
22	XTAL1	I	晶振连接端 1（6MHz）
23	XTAL2	O	晶振连接端 2（6MHz）
24	V_{CC}	P	电源电压 4.0~5.5V，要使器件工作在 3.3V，对 V_{CC} 和 VOUT3.3 都提供 3.3V
25	D+	A	USB 上行端口 D+数据线，在全速模式下，要求使用 1.5kΩ电阻接至 3.3V 电压输出端
26	D-	A	USB 上行端口 D-数据线，在低速模式下，要求使用 1.5kΩ电阻接至 3.3V 电压输出端
27	VOUT3.3	P	3.3V 调整输出。要使器件工作在 3.3V，对 V_{CC} 和 VOUT3.3 脚都提供 3.3V
28	A0	I	地址位。A0=1 选择命令指示，A0=0 选择数据。该位在多路地址/数据总线配置时，可忽略应将其接高电平

其中：

O2 为 2mA 驱动输出；

OD4 为 4mA 驱动开漏输出；

OD8 为 8mA 驱动开漏输出；

IO2 为 4mA 输出。

2．功能结构

PDIUSBD12 芯片的结构主要包括以下几部分。

- 电压调整器：片内集成了一个 3.3V 的调整器用于模拟收发器的供电。该电压还作为输出连接到外部 1.5kΩ的上拉电阻。PDIUSBD12 提供集成 1.5kΩ上拉电阻的 SoftConnect 技术。
- PLL：片上集成了 6MHz 到 48MHz 时钟乘法，允许使用低成本的 6MHz 晶振，EMI 也由于使用低频晶振而降低。PLL 的使用不需要外部元件。
- 位时钟恢复：位时钟恢复电路采用 4 倍过采样原理，从输入的 USB 数据流中恢复时钟。它能跟踪 USB 规定范围内的抖动和频漂。
- PHILIPS 串行接口引擎（PSIE）：PHILIPS SIE 实现了全部的 USB 协议层。完全由硬件实现而不需要固件的参与。该模块的功能包括同步模式的识别、并行/串行转换、位填充/解除填充、CRC 校验/产生、PID 校验/产生、地址识别和握手评估/产生。
- SoftConnect：高速设备与 USB 的连接是通过 1.5kΩ上拉电阻将 D+职位高实现的。1.5kΩ上拉电阻集成在 PDIUSBD12 片内，默认状态下不与 Vcc 相连。连接的建立通过外部系统微处理器发送命令来实现的。这就允许 CPU 在决定与 USB 建立连接之前完成初始化时序。USB 总线连接可以重新初始化而不需要拔出电缆。
- GoodLink：GoodLink 技术可提供良好的 USB 连接指示。在枚举中，LED 指示灯根据通信的状况间歇闪烁。当 PDIUSBD12 成功的枚举和配置后，LED 指示灯将一直点亮。在 PDIUSBD12 的数据传输过程中，LED 将闪烁；在挂起期间，LED 熄灭。这种特性为 USB 器件、集线器和 USB 通信状态提供了用户友好的指示。作为一个诊断工具，它对隔离故障的设备是很有用的。该特性降低了现场支持和热线的成本。
- 并行和 DMA 接口：对一个微控制器而言，PDIUSBD12 看起来就像一个带 8 位数据总线和一个地址位（占用 2 个位置）的存储器件。PDIUSBD12 支持独立的和分时复用的地址和数据总线。还支持主端点与本地共享 RAM 之间直接读取的 DMA 传输，支持单周期和突发模式的 DMA 传输。

它的功能结构如图 17-7 所示。

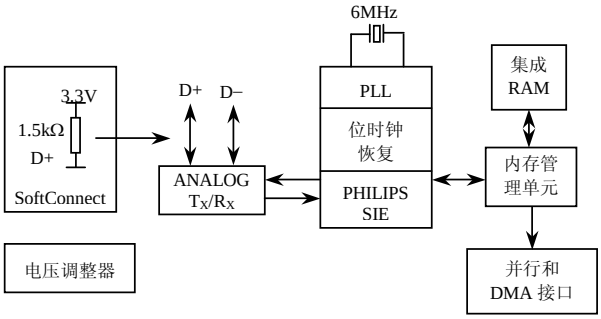


图 17-7　PDIUSBD12 的功能框图

3．USB 端口模式

USB 控制器的端口适用于不同类型的设备，例如图像、打印机、海量存储器和通信设备。端口可通过“Set Mode”命令配置为 4 种不同的模式，对应表 17-2~表 17-5 所示。

- 模式 0（non-ISO 模式）：非同步传输；

- 模式 1（ISO-OUT 模式）：同步输出模式；
- 模式 2（ISO-IN 模式）：同步输入模式；
- 模式 3（ISO-IO 模式）：同步输入输出传输。

表 17-2　模式 0（非同步模式）

端点数	端点索引	传输类型	端点类型	方　　向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	普通输出	普通	输出	64
	5	普通输入	普通	输入	64

表 17-3　模式 1（同步输出模式）

端点数	端点索引	传输类型	端点类型	方向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输出	同步	输出	128

表 17-4　模式 2（同步输入模式）

端点数	端点索引	传输类型	端点类型	方向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输入	同步	输入	128

表 17-5　模式 3（同步输入/输出模式）

端点数	端点索引	传输类型	端点类型	方向	最大信息包规格（字节）
0	0	控制输出	默认	输出	16
	1	控制输入		输入	16
1	2	普通输出	普通	输出	16
	3	普通输入	普通	输入	16
2	4	同步输出	同步	输出	64
	5	同步输入	同步	输入	64

这些端点描述符分别定义如下：

```
//端点 1 发送端点描述符
code USB ENDPOINT DESCRIPTOR EP1 TXDescr =
{
    sizeof （USB ENDPOINT DESCRIPTOR） ,
    USB ENDPOINT DESCRIPTOR TYPE ,
    0x81 ,
```

```
    USB ENDPOINT TYPE INTERRUPT ,
    SWAP( EP1 PACKET SIZE) ,
    1
} ;
//端点 1 接收端点描述符
code USB ENDPOINT DESCRIPTOR EP1 RXDescr =
{
    sizeof (USB ENDPOINT DESCRIPTOR) ,
    USB ENDPOINT DESCRIPTOR TYPE ,
    0x1 ,
    USB ENDPOINT TYPE INTERRUPT ,
    SWAP( EP1 PACKET SIZE) ,
    1
} ;

//端点 2 发送端点描述符
code USB ENDPOINT DESCRIPTOR EP2 TXDescr =
{
    sizeof (USB ENDPOINT DESCRIPTOR) ,
    USB ENDPOINT DESCRIPTOR TYPE ,
    0x82 ,
    USB ENDPOINT TYPE BUL K,
    SWAP( EP2 PACKET SIZE) ,
    10
} ;
//端点 2 接收端点描述符
code USB ENDPOINT DESCRIPTOR EP2 RXDescr =
{
    sizeof (USB ENDPOINT DESCRIPTOR) ,
    USB ENDPOINT DESCRIPTOR TYPE ,
    0x2 ,
    USB ENDPOINT TYPE BUL K,
    SWAP( EP2 PACKET SIZE) ,
    10
} ;
```

其中主端口（端口 2）在有些方面是比较特别的，它是吞吐大数据的主要端口。同时，它执行主端口的特性以减轻传输大数据的任务。

（1）双缓冲：允许 USB 与 ARM 之间进行并行读写操作，这样就增加了数据的吞吐量。缓冲区切换是自动处理的，这导致了透明的缓冲区操作。

（2）支持 DMA（直接存储器访问）操作：可以与其他端点的正常 I/O 操作交叉进行。

（3）DMA 操作中的自动指针处理：在跨过缓冲区边界时不需要 ARM 的干预。

（4）可配置为同步传输或非同步（批量和中断）传输。

4．系统硬件电路结构图

USB 接口主芯片电路如图 17-8 所示，在枚举中，LED 指示灯根据通信的状况间歇闪烁。当 PDIUSBD12 成功地枚举和配置后，LED 指示灯将一直点亮。在 PDIUSBD12 的数据传输过程中，LED 将闪烁；在挂起时 LED 熄灭。

由于 PDIUSBD12D12 采用的是独立地址方式，因此将 ALE 引脚接地；ARM 的 nGCS2 作为 PDIUSBD12 的片选信号，这是使用 ARM 的 External I/O BANK2 的地址范围；PDIUSBD12 的 A0 脚与 ARM 的 ADDR0 口相连用以控制 PDIUSBD12 的命令和数据状态；PDIUSBD12 的中断脚 INT_N 连接到 ARM 外部中断 2 脚，使用外部中断 2。

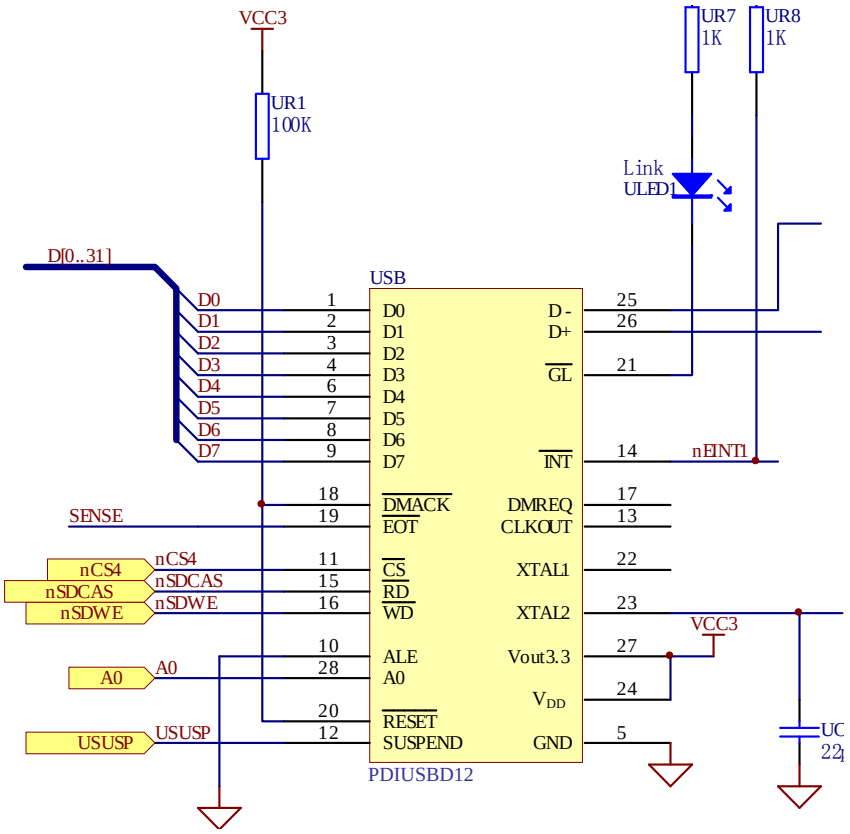


图 17-8　PDIUSBD12 连接原理图

PDIUSBD12 的晶振采用 6MHz 的普通晶振，经过 PDIUSBD12 的内部倍频以后，实际的内部时钟为 24MHz。晶振电路使用的电容值是不一样的，一个是 22pf，一个是 68pf。工作时 GL 引脚所接的发光二极管将会发光，表示正常工作。用并行接口方式连接 USB 和 CPU，类似一个带 8 位数据总线和一个地址位占用 2 个位置的存储器件。

如图 17-9 所示是 USB 总线接口图，一个 USB 口作为 Host 方式用于下载代码。

USB 总线控制器 PDIUSBD12 的 D+和 D-引脚分别串接 22Ω的电阻和电感，其中电感起到电源滤波的作用，并且 D+和 D-线必须用 1MΩ的下拉电阻。USB 总线接口由总线来供电。

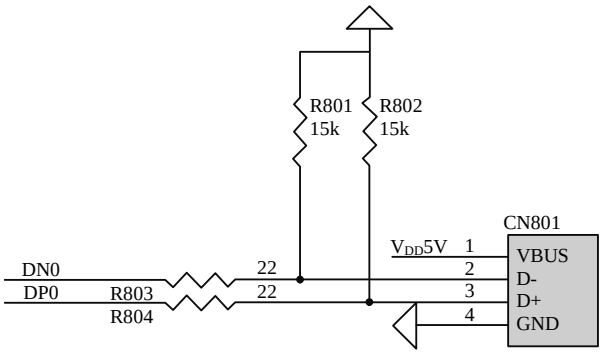


图 17-9　USB 总线接口电路图

17.3.2 系统工作流程

1．USB 数据传输

系统中 USB 接口数据传输的主要任务是从 PC 机上下载内核（.bin 文件）或应用程序（.exe 文件）。

主 CPU 系统板上的 PDUD12USB 接口芯片支持 USB 传输协议 1.1 版本，但由于它只能作为从设备使用，所以必须和上位机（PC 机）之类带有 USB 主设备的系统进行数据传输。本系统中设计的 USB 传输是与 PC 机进行传输的，它相对于 PC 机是从设备。在此 USB 接口数据的正确传输依靠以下 3 个部分。

（1）Device 设备中的固件程序；

（2）Host 端的驱动程序；

（3）Host 端的应用程序。

这三者的关系如图 17-10 所示。

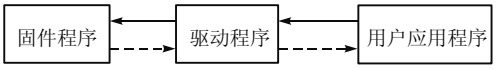


图 17-10　USB 数据传输图

USB 数据传输提供的功能有：

- 管理 Host 和 Device 之间进行的 USB 标准流量控制；
- 管理 Host 和外设之间的数据流；
- 搜集系统状态和性能的统计信息。

2．Device 端固件程序的设计

结合 USB1.1 协议和 PDIUSBD12 USB 接口芯片的特性设计固件程序。固件程序主要是对 ARM 中的 USB 接口设备进行配置和 USB 数据传输的读写操作。其中，USB Device 配置包括设备描述符、配置描述符、接口描述符、端点描述符和字符串描述符（可选）等。配置过程在 USB 设备插入 PC 机时完成，在此程序设计中，通过控制端点 0 和 PC 机交换信息来配置 USB 从设备；然后，通过 USB 读写端点 2 来传输数据。整个固件程序的流程如图 17-11 所示。

17.3.3 系统模块源代码实现

1．Host 端控制程序设计

新一代通用串行接口 USB 的优良特性提供了一个的解决方案。CUSB 的封装性和继承性如图 17-12 所示。将 USB 函数封装成一个 CUSB 类在使用时提供了很大的方便，它主要包括 2 个成员变量和 4 个成员函数，通过继承父类得到 2 个派生类，这 2 个派生类公有地继承了 CUSB 类，通过对 4 个成员函数使用完成对 USB 的读写操作，类的封装性方便了程序的结构和安全性，而继承性则大大地提高软件的开发效率。同时，封装性和继承性的结合大大提高了系统的可靠性和软件的重用性。

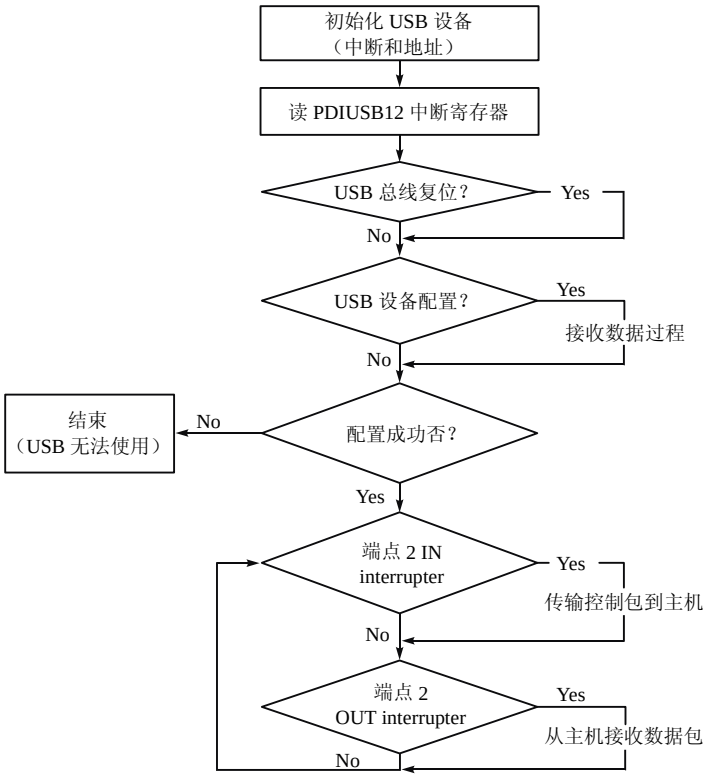


图 17-11　USB 固件程序流程图

以下为 USB 与嵌入式主机进行数据通信的关键程序。

```
unsigned char device_descriptor[] =
{
  0x12,                  // 描述表大小 (18 bytes)
```

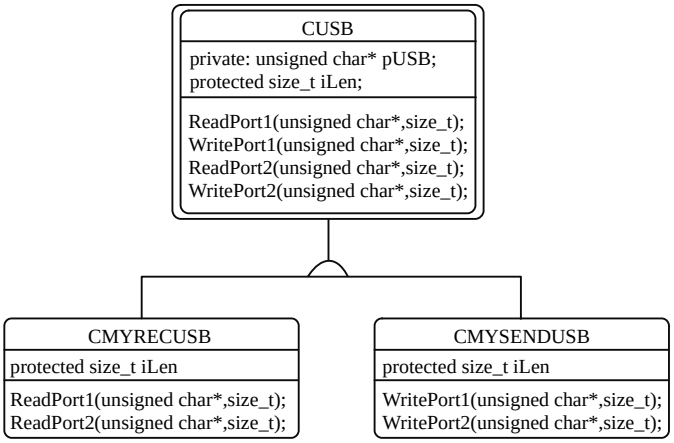


图 17-12　CUSB 的继承特性

```
  0x01,                  // 设备描述表类型
  0x10, 0x01,            // 兼容设备版本号(BCD 码)
  0x00,
```

```
  0x00,
  0x00,
  0x10,                  // 端点 0 最大包大小(8,16,32,64)
  0x84, 0x05,            // 厂商 ID
  0x01, 0x00,            // 产品 ID
  0x00, 0x01,            // 设备发行号(BCD 码)
  0x00,                  // 厂商信息字符串索引
  0x00,                  // 产品信息字符串索引
  0x00,                  // 设备序列号字符串索引(不支持设为 0)
  0x01                   // 可能配置数
};
unsigned char Configuration_Descriptor[] =
{
  0x09,                  // 配置描述表长度
  0x02,                  // 配置描述表类型
  0x22, 0x00,            // 配置描述表及附带表长度
  0x01,                  // 接口配置数
  0x01,                  // 配置描述表标识
  0x00,                  // 配置描述表字符串描述表索引
  0xc0,                  // 配置属性
  0x32                   // 总线供电最大值(*2mA)
};
unsigned char Interface_Descriptor[] =
{
  0x09,                  // 接口描述表长度(9)
  0x04,                  // 接口描述表类型
  0x00,                  // 接口号(0)
  0x00,                  // 轮寻设置(0 号端点)
  0x01,                  // 端点数 (1)
  0x03,                  // 接口类属
  0x00,                  // 子类码
  0x00,                  // 协议码
  0x00                   // 接口字符串描述表索引(不支持)
};
unsigned char Endpoint1_out_Descriptor[] =
{
  0x07,                  // 端点描述表长度(7)
  0x05,                  // 端点描述表类型
  0x81,                  // OUT 端点(1)
  0x03,                  // 传输模式(0 控制、1 同步、2 批、3 中断传输)
  0x10, 0x00,            // 最大包大小(16)
  0x0A                   // 轮寻时间 (10ms)
};
unsigned char TotalConf[] =
{
  0x09,                  // 配置描述表长度
  0x02,                  // 配置描述表类型
  0x22, 0x00,            // 配置描述表及附带表长度
  0x01,                  // 接口配置数
  0x01,                  // 配置描述表标识
  0x00,                  // 配置描述表字符串描述表索引
  0xc0,                  // 配置属性
```

```
  0x32,              // 总线供电最大值(*2mA)
  0x09,              // 接口描述表长度(9)
  0x04,              // 接口描述表类型
  0x00,              // 接口号(0)
  0x00,              // 轮寻设置(0 号端点)
  0x01,              // 端点数 (1)
  0x03,              // 接口类属
  0x00,              // 子类码
  0x00,              // 协议码
  0x00,              // 接口字符串描述表索引(不支持)
  0x09,
  0x21,
  0x00, 0x01,
  0x00,
  0x01,
  0x22,
  0x34,
  0,
  0x07,              // 端点描述表长度(7)
  0x05,              // 端点描述表类型
  0x81,              // OUT 端点(1)
  0x03,              // 传输模式(0 控制、1 同步、2 批、3 中断传输)
  0x10, 0x00,        // 最大包大小(16)
  0x0A
};
unsigned char USBStringLanguageDescription[] =
{
  0x04,              // 字符串描述表长度
  0x03,              // 字符串描述表类型
  0x09,              // 语言标识(9 英语)
  0x04               // 子语言标识: Default
};
…
…
void usb_interrupt()
{
  unsigned short lIrq;
  char iIrqUsb;
  unsigned char tmp;
  int i;
    while (1)
    {
         lIrq = *(unsigned short*)0x80000240;
         lIrq &= 0x0020;
         if (lIrq == 0x0020)
         {
         if (bIsOrig)
         {
          XmtBuff.pNum = 16;
         }
       SETADDR = 0xF4;            //读 IRQ 寄存器
       iIrqUsb = SETDATA;
       tmp = SETDATA;
       if (iIrqUsb & 0x01)        //EP0 OUT
         {
```

```
    XmtBuff.out = 0;
    XmtBuff.in = 1;
    SETADDR = 0x40;
    tmp = SETDATA;
  if (tmp & 0x20)
   {
    tx_0 ();
   }
    else
   {
    SETADDR = 0x00;    // 选择端点 0(指针指向 0 位置)
    SETADDR = 0xF0;    // 读标准控制码
    tmp = SETDATA;
    tmp = SETDATA;
    for (i = 0; i < 8; i++)
    {
       XmtBuff.b[i] = SETDATA;
    }
    if (bSetReport)
    {
       for (i = 0; i < 8; i++)
      {
       HIDData[i] = XmtBuff.b[i];
      }
       bSetReport = 0;
     }
    SETADDR = 0xF1;    //应答 SETUP 包,使能(清 OUT 缓冲区、使能 IN 缓冲区)命令
    SETADDR = 0xF2;    // 清 OUT 缓冲区
    SETADDR = 0x01;    // 选择端点 1(指针指向 0 位置)
    SETADDR = 0xF1;    //应答 SETUP 包,使能(清 OUT 缓冲区、使能 IN 缓冲区)命令
  }
 }
else if (iIrqUsb & 0x02)  //EP0 IN
{
   XmtBuff.in = 1;
   SETADDR = 0x41;        //读 in 最后状态
   tmp = SETDATA;
   rx_0 ();
}
else if (iIrqUsb & 0x04)  //EP1 OUT
{
   XmtBuff.out = 2;
   XmtBuff.in = 3;
   SETADDR = 0x42;        //读 out 最后状态
   tmp = SETDATA;
   tx_1 ();
}
else if (iIrqUsb & 0x08)  //ENDP1_IN
{
   XmtBuff.in = 3;
   SETADDR = 0x43;        //读 in 最后状态
   tmp = SETDATA;
   XmtBuff.b[0] = 5;
   XmtBuff.wrLength = 8;
   XmtBuff.p = HIDData;
   rx_0 ();
```

```
        }
        else if (iIrqUsb & 0x10)  //EP2 OUT
        {
            XmtBuff.pNum = 64;
            XmtBuff.out = 4;
            XmtBuff.in = 5;
            SETADDR = 0x44;        //读 out 最后状态
            tmp = SETDATA;
            read_out ();
        }
        else if (iIrqUsb & 0x20)  //EP2 IN
        {
            XmtBuff.pNum = 64;
            XmtBuff.in = 5;
            SETADDR = 0x45;        //读 in 最后状态
            tmp = SETDATA;
            XmtBuff.b[0] = 5;
            XmtBuff.wrLength = 1;
            XmtBuff.p = XmtBuff.b;
            rx_0 ();
        }
        else if (iIrqUsb & 0x80)
        {
        }
        else if (iIrqUsb & 0x40)
        {
        }
      }
     }
  }
  void MLsup_StallEP0 (void)
  {
    SETADDR = 0x40;       // 0 端点停止(用于发送 Stall 包)
    SETDATA = 0x01;
    SETADDR = 0x41;       // 1 端点停止(用于发送 Stall 包)
    SETDATA = 0x01;
    SETADDR = 0xF1;       // 应答 SETUP 包,使能(清 OUT 缓冲区、使能 IN 缓冲区)命令
  }
  void GetStatus (void)
  {
    XmtBuff.b[1] = 0x00;
    switch (XmtBuff.b[0])
    {
      case 0x80:          //返回设备状态
    //发送两个字节数据：第一字节 D1 为 1 支持远程唤醒，D0 为 0 是总线供电,其他位为 0；第二
字节为 0
       XmtBuff.b[0] = 0x03;
       break;
      case 0x81:          //返回接口状态
       //发送两个字节数据：第一字节为 0；第二字节为 0
       XmtBuff.b[0] = 0x00;
       break;
      case 0x82:          //返回端点状态
       //发送两个字节数据：第一字节 D0 为 1 端点处于暂停，否则 D0 为 0,其他位为 0；第二字节为 0
       //XmtBuff.b[5] D7 为方向，D3~0 为端点号
```

```
      XmtBuff.b[0] = 0x00;
      break;
    }
  XmtBuff.wrLength = 2;
  XmtBuff.p = XmtBuff.b;
  rx_0 ();
}

void ClearFeature (void)
{
  switch (XmtBuff.b[0])
  {
    case 0x00:                                  //清设备唤醒功能
      break;
    case 0x01:                                  //清接口状态
      break;
    case 0x02:                                  //启用端点
      // D7 为方向，D3~0 为端点号
      if (XmtBuff.b[5] < 0x80)
     {
       SETADDR = 0x40 + XmtBuff.b[5] * 2;     // 启用 OUT 端点
       SETDATA = 0x00;
     }
      else
     {
       XmtBuff.b[5] = XmtBuff.b[5] & 0x0f;
       SETADDR = 0x41 + XmtBuff.b[5] * 2;     // 启用 IN 端点
       SETDATA = 0x00;
     }
      break;
    }
  XmtBuff.wrLength = 0;
  rx_0 ();
}
void SetFeature (void)
{
  switch (XmtBuff.b[0])
  {
    case 0x00:                                  //设置设备唤醒功能
      break;
    case 0x01:                                  //设置接口状态
      break;
    case 0x02:                                  //停止端点
      // D7 为方向，D3~0 为端点号

     if (XmtBuff.b[5] < 0x80)
     {
       SETADDR = 0x40 + XmtBuff.b[5] * 2;     // 停止 OUT 端点
       SETDATA = 0x01;
     }
      else
     {
       XmtBuff.b[5] = XmtBuff.b[5] & 0x0F;
       SETADDR = 0x41 + XmtBuff.b[5] * 2;     // 停止 IN 端点
       SETDATA = 0x01;
     }
```

```
      break;
    }
  XmtBuff.wrLength = 0;
  rx_0 ();
}
void SetAddress (void)
{
  SETADDR = 0xD0;       //设置新地址使能
  SETDATA = 0x80 | XmtBuff.b[2];
  XmtBuff.wrLength = 0;
  rx_0 ();
}
void GetConfiguration (void)
{
  XmtBuff.b[0] = 1;     //返回是否被配置(非 0 为配置)
  XmtBuff.wrLength = 1;
  XmtBuff.p = XmtBuff.b;
  rx_0 ();
}
void SetConfiguration (void)
{
  if (XmtBuff.b[0] == 0x00)
  {
     XmtBuff.wrLength = 0;
     rx_0 ();
     SETADDR = 0xD8;
     if (XmtBuff.b[2] == 0x00)
    {
      SETDATA = 0x00;   // 停止普通/同步端点
    }
     else if (XmtBuff.b[2] == 0x01)
    {
      SETDATA = 0x01;   // 使能普通/同步端点
      SETADDR = 0x03;
      SETADDR = 0xFA;   // 设置 IN 缓冲区有效(满标志)
      SETADDR = 0x05;
      SETADDR = 0xFA;   // 设置 IN 缓冲区有效(满标志)
    }
   }
  else
    MLsup_StallEP0 ();
}
void rx_0 ()
{
  INT8 tmp;
  if (XmtBuff.pNum > XmtBuff.wrLength)
  {
     XmtBuff.b[6] = XmtBuff.wrLength;
  }
  else
  {
     XmtBuff.b[6] = XmtBuff.pNum;
     bIsOrig = 0;
  }
  tmp =  XmtBuff.in;
  SETADDR = tmp;        // 选择 IN 端点(指针指向 0 位置)
```

```
  SETADDR = 0xF0;       // 写缓冲区
  SETDATA = 0x00;
  tmp =  XmtBuff.b[6];
  SETDATA = tmp;
  for (XmtBuff.b[7] = 0; XmtBuff.b[7] < XmtBuff.b[6]; XmtBuff.b[7]++)
  {
     tmp = *(XmtBuff.p++);
     SETDATA = tmp;
  }
  XmtBuff.wrLength -= XmtBuff.b[6];
  if (XmtBuff.wrLength <= 0)
    bIsOrig = 1;
  SETADDR = 0xFA;       // 设置 IN 缓冲区有效(满标志)
  tmp = XmtBuff.in;
  SETADDR = tmp | 0x40;
  tmp = SETDATA;
}
void tx_0 ()
{
  int i;
  SETADDR = 0x00;
  SETADDR = 0xF0;
  XmtBuff.b[0] = SETDATA;
  XmtBuff.b[1] = SETDATA;
  for (i = 0; i < 8; i++)
  {
     XmtBuff.b[i] = SETDATA;
  }
  SETADDR = 0xF1;       // 应答 SETUP 包,使能(清 OUT 缓冲区、使能 IN 缓冲区)命令
  SETADDR = 0xF2;       // 清 OUT 缓冲区
  SETADDR = 0x01;       // 选择端点 1(指针指向 0 位置)
  SETADDR = 0xF1;       // 应答 SETUP 包,使能(清 OUT 缓冲区、使能 IN 缓冲区)命令
  if (XmtBuff.b[0] & 0x20)  //厂商请求跳转表
  {
     (*NonStandardDeviceRequest[XmtBuff.b[1]]) ();
     return;
  }
  else
    {
     if (XmtBuff.b[1] <= 0x0B) //标准请求跳转表
    {
      (*StandardDeviceRequest[XmtBuff.b[1]]) ();
      return;
    }
    }
}

void tx_1 ()
{
  int i;
  SETADDR = XmtBuff.out;    // 选择端点 0(指针指向 0 位置)
  SETADDR = 0xF0;           // 读标准控制码
  XmtBuff.b[0] = SETDATA;
  XmtBuff.b[1] = SETDATA;
  for (i = 0; i < 8; i++)
  {
```

```
        XmtBuff.b[i] = SETDATA;
    }
    SETADDR = 0xF2;            // 清 OUT 缓冲区
}

void read_out ()
{
    INT8 i;
    SETADDR = XmtBuff.out;     // 选择端点 0(指针指向 0 位置)
    SETADDR = 0xF0;            // 读标准控制码
    XmtBuff.b[0] = SETDATA;
    XmtBuff.b[1] = SETDATA;
    for (i = 0; i < 8; i++)
    {
        XmtBuff.b[i] = SETDATA;
    }
    SETADDR = 0xF2;            // 清 OUT 缓冲区
}
```

2．USB Device 驱动程序设计

目前驱动程序开发的工具主要有 Windiver、微软的 DDK 和 Compuware 公司的 DriverStudio 等。

这里使用 DriverStudio 驱动程序开发工具进行开发，对于熟悉面向对象编程的软件开发员，DriverStudio 是一个良好的驱动开发工具，并且开发时间比较短。DriverStudio 工具包中的 DriverWorks 提供了 KDriver、KPnpDevice 和 KpnpLowerDevice3 个类，这 3 个类用于实现 WDM 驱动程序的框架结构。其中，KDriver 类提供设备驱动程序的基本框架结构。它负责驱动程序的初始化和将 IRP 分发到目标设备对象。由于 KDriver 是抽象类，必须创建一个 KDriver 的派生类，并重载 DriverEntry 例程，在 DriverEntry 例程中做一些初始化工作。每当 Pnp 子系统检测到驱动程序所负责的设备时，就调用 AddDevice 例程，UnLoad 例程负责最后的清除工作。

对于 KPnpDevice 类，它是 KDevice 类的派生类，在驱动程序中只作为基类使用，它支持即插即用和电源管理，主要处理 IRP_MJ_PNP 和 IRP_MJ_POWER 请求包。

KPnpLowerDevice 类提供了一个物理设备对象的模型，当驱动程序创建或初始化一个 KPnpLowerDevice 类实例时，它就将一个设备对象连向了一个物理设备对象。

除用到以上类外，开发 USB 驱动程序还用到了 DriverWorks 提供的 3 个用于实现 USB 设备操作的类：KUsbLowerDevice、KUsbInterface 和 KUsbPipe 类。其中，KUsbLowerDevice 实例代表端点 0，允许 USB 驱动程序通过默认控制管道控制 USB 设备，如配置 USB 设备，传输各种控制和状态请求；KUsbInterface 类的作用更多是结构上的而非功能上的，其成员函数几乎不与实际物理设备交互作用，驱动程序用这个类获取接口和管道信息；KUsbPipe 类对应于管道，管道是主机和一个端点的信息连接，这个类用于初始化管道信息和管道操作控制。以下是部分驱动程序代码。

（1）设备驱动程序的 DriverEntry 和 AddDevice 例程。

```
NTSTATUS Asgccusb::DriverEntry(PUNICODE_STRING RegistryPath)  //程序入口
{
    KRegistryKey Params(RegistryPath, L"Parameters");
    if ( NT_SUCCESS(Params.LastError()) )
```

```
    {
        LoadRegistryParameters(Params);
    }
    m_Unit = 0;
    return STATUS_SUCCESS;
}
NTSTATUS Asgccusb::AddDevice(PDEVICE_OBJECT Pdo)            //加载设备
{
    AsgccusbDevice * pDevice = new (
            static_cast<PCWSTR>(KUnitizedName(L"AsgccusbDevice", m_Unit)),
            FILE_DEVICE_UNKNOWN,
            NULL,
            0,
            DO_DIRECT_IO
                | DO_POWER_PAGABLE
            )
        AsgccusbDevice(Pdo, m_Unit);
    if (pDevice == NULL)
    {
       return STATUS_INSUFFICIENT_RESOURCES;
    }

    NTSTATUS status = pDevice->ConstructorStatus();
    if ( !NT_SUCCESS(status) )                    //不成功则删除该设备
    {
        delete pDevice;
    }
    else
    {
        m_Unit++;
        pDevice->ReportNewDevicePowerState(PowerDeviceD0);
    }
    return status;
}
```

（2）数据读写（read 和 write）例程，如果固件程序中对 USB 设备配置选择的是哪个端点读写数据，驱动程序中便使用相应的那个端点来读写数据。具体的代码如下。

```
NTSTATUS AsgccusbDevice::Read(KIrp I)                       //读进程
{
   if (!NT_SUCCESS(I.Information()))
    {
        I.Information() = 0;
        return I.PnpComplete(this, STATUS_INVALID_PARAMETER);
    }
    if (I.ReadSize() == 0)                                  //读是否完成
    {
        I.Information() = 0;
        return I.PnpComplete(this, STATUS_SUCCESS);
    }
    KMemory Mem(I.Mdl());
   ULONG dwTotalSize = I.ReadSize(CURRENT);
    ULONG dwMaxSize = m_Endpoint2IN.MaximumTransferSize();

    if (dwTotalSize > dwMaxSize)
    {
```

```
            ASSERT(dwMaxSize);
            dwTotalSize = dwMaxSize;
        }
        USB_COMPLETION_INFO*pCompInfo=new(NonPagedPool)USB_COMPLETION_INFO;
        if (pCompInfo == NULL)                                    //空间不足
        {
            I.Information() = 0;
            return I.PnpComplete(this, STATUS_INSUFFICIENT_RESOURCES);
        }
        PURB pUrb = m_Endpoint2IN.BuildBulkTransfer(
                    Mem,
                        dwTotalSize,
                        TRUE,
                        NULL,
                        TRUE
                        );
        if (pUrb == NULL)                                         //空间不足
        {
            delete pCompInfo;
            I.Information() = 0;
            return I.PnpComplete(this, STATUS_INSUFFICIENT_RESOURCES);
        }
        pCompInfo->m_pClass = this;
        pCompInfo->m_pUrb = pUrb;
        NTSTATUS status;
        status  =  m_Endpoint2IN.SubmitUrb(I,  pUrb,  LinkTo(ReadComplete),
pCompInfo, 0);
        return status;
    }
```

关于写数据的例程和上述的读数据的例程相似，只要相应地把读的函数换成写的函数即可，在此不详细列出。

3．USB 应用程序开发

应用程序是使用 Visual C++开发的，主要是调用 Windows 提供的一系列 API 函数来实现与驱动程序交互。主要使用的 Win32API 函数有以下几个。

（1）CreatFile()：打开设备文件；

（2）CloseHandle()：关闭设备文件；

（3）ReadFile()：从设备文件中读数据；

（4）WriteFile()：向设备文件中写入数据。

具体实现在此不详细列出。其中，应用程序界面如图 17-13 所示。

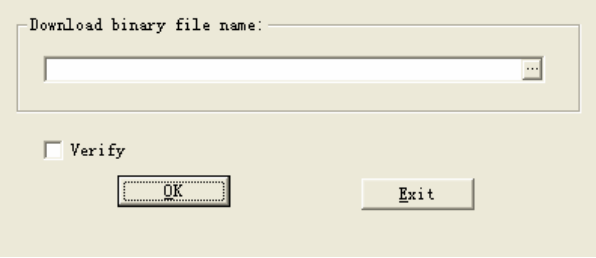


图 17-13　USB 应用程序界面图

17.4 本章总结

本章详细讲述了基于 USB 设备驱动开发的应用实例，它采用 USB 通信协议编写了 PC 机与嵌入式系统的 USB 通信接口，利用 USB 灵活、稳定、即插即用、主从结构、速度快（12Mbps）、成本低廉的特点，可以解决数据包小、传输速度快的问题；用 USB 接口将 ARM 和计算机相连，可以满足系统对于传输速率和安装方便的要求，实现实时的数据传输和调试。

作为开源系统，在 Linux 上开发设备驱动程序有着其他嵌入式系统不可比拟的优势，大量的开放源码无疑可以加速开发的进程，并使得其应用更加广泛。由此可见，USB 作为一种新型的高速外设总线，在嵌入式 Linux 领域必定有着广阔的应用前景。

第 18 章

家庭安全监控系统设计实例

在信息网络技术高速发展的今天，安全监控系统已逐步成为家庭防护的基本设备。由于以前通过传统闭路电视实现的安全监控系统暴露出通用性差、不易扩展、不能网络化等缺点，因此现在安全系统正在向基于 IP 网络的数字视频监控转变。在整个数字时代大背景下，通过改善传输条件，逐渐走向网络化，成为安全监控市场最新的发展方向。

在智能家居系统中，运用到许多无线网络技术，采用无线网络可以提供更大的灵活性、流动性，省去花在综合布线上的费用和精力，因此随着无线网络技术的进一步发展和应用深入，将大大促进家庭网络智能化的进程。

18.1 应用环境与硬件设计概要

本章家庭安全监控系统中，Linux 客户端系统采用 Intel Xscale PXA255 开发板，传感器系统采用 EDUKIT-II S3C2410 开发板并且基于摄像头和红外传感器。

18.1.1 系统功能和组成

家庭安全监控系统的目标是使用户可以随时通过移动通信设备（手机、PDA 等）、通过网络查看被监控地点的摄像设备捕捉到的视频信息。

如图 18-1 所示，家里的摄像设备进行监视，将视频信息传送到服务器进行处理和保存。当用户向服务器请求查看时，服务器将数据通过无线网络传送至用户的移动通信终端。

整个家庭安全服务系统包括服务器、传感器和客户端（基于 T-Engine、Symbian、嵌入式 Linux 三种嵌入式移动终端）三部分组成。各部分分工如下：

（1）传感器系统通过无线网络实现对用户家庭安全情况的实时监测。本系统使用无线技术在家庭内组建无线局域网络，并通过网络中的传感器设备（如无线摄像头、无线红外探头及无线门磁等）对用户室内情况监测，将监测信息通过无线局域网传送数据至服务器，供服务器进行相应处理。摄像头负责采集家庭室内视频及图像信号，并传输至服务器。红外探头负责监控家庭内是否有陌生人入侵，并将信号传输至服务器。

（2）Linux 客户端系统，通过有线网络实现对用户家庭安全情况的实时监测。其中屏幕负责显示图片或者视频。

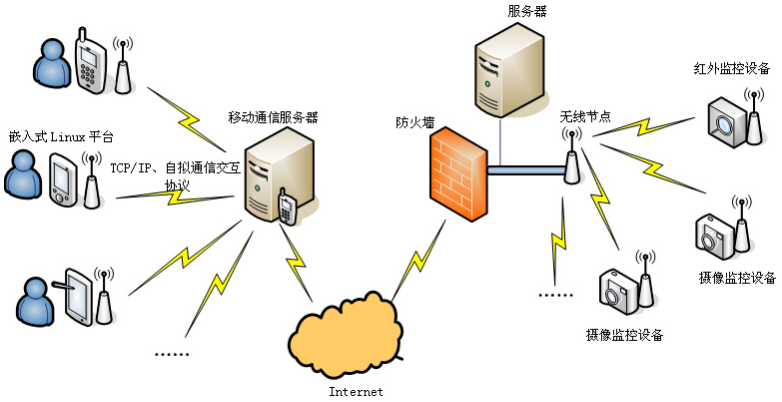


图 18-1　家庭安全服务系统工作网络图

18.1.2　系统模块功能描述

1．Linux 客户端主要功能

（1）监控功能

用户通过服务终端向服务器发送监控请求，明确需要进行的监控类型。服务器端收到请求后，向终端发送相关的多媒体信息，包括图片或视频。过程如图 18-2 所示。

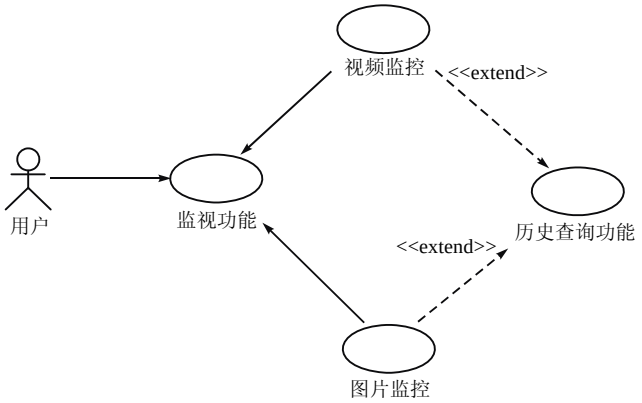


图 18-2　监控功能用例图

- 视频监控功能

用户向服务器端发起视频监控请求。服务器端通过摄像头捕捉室内的视频信息后，将这些信息通过无线网络发送到移动通信终端。用户通过手持设备上的软件查看这些视频。视频监控活动图如图 18-3 所示。

- 图片监控功能

用户向服务器端发送视频监控请求。服务器端通过摄像头对室内进行拍照后，将图片通过无线网络发送到移动通信终端。用户通过终端上的软件查看这些图像。图片监控活动图如图 18-4 所示。

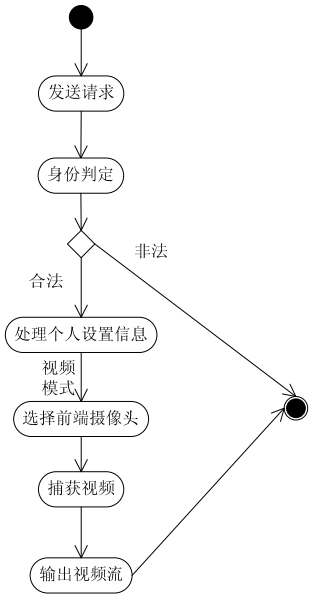


图 18-3　视频监控功能活动图

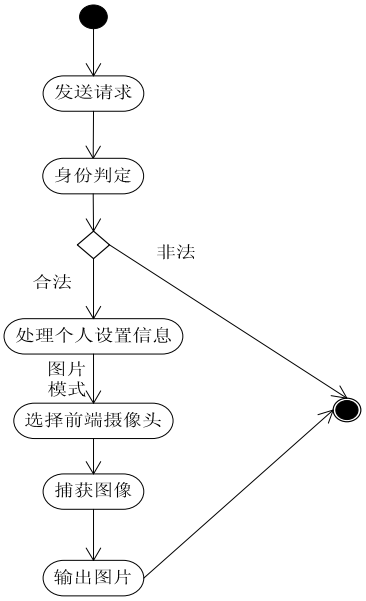


图 18-4　图片监控功能活动图

（2）历史查询功能

用户可以通过发送一个历史视频浏览的请求，将想浏览的历史视频的时间发送到服务器端。服务器端找到这个时间段的视频后，通过网络将视频数据发送到客户端。用户通过手持设备查看该视频数据。如果在被请求的时间段中，用户没有将视频捕捉模式设置为实时捕捉功能，或者不是定时拍照时间，则向客户端发送一个错误信息。图 18-5 为历史查询功能活动图。

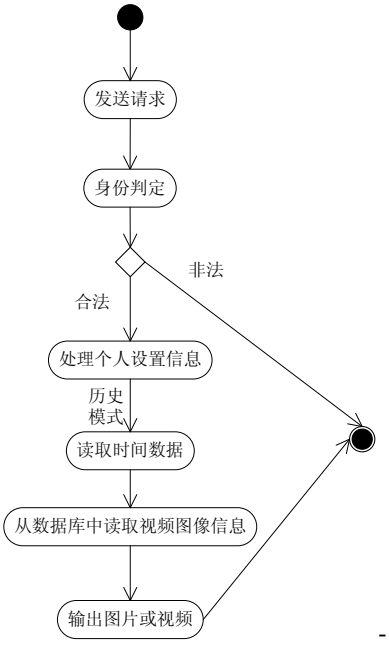


图 18-5　历史查询功能活动图

（3）数据捕捉功能

数据捕捉功能用例图如图 18-6 所示，数据捕捉功能活动情况如图 18-7 所示。

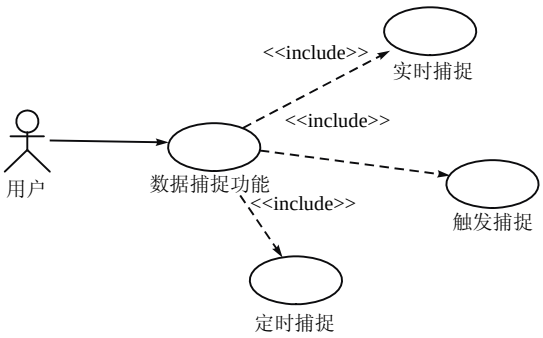


图 18-6　数据捕捉功能用例图

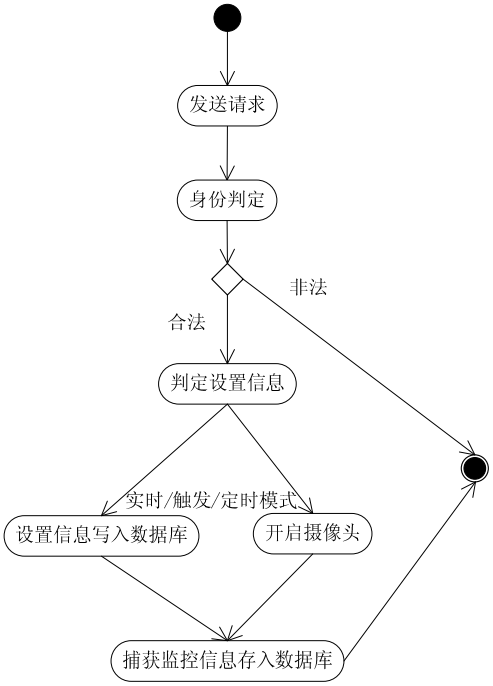


图 18-7　数据捕捉功能活动图

- 实时捕捉功能

用户可以在终端上访问服务器端的功能设置页面，将视频捕捉模式设置为实时捕捉功能。在该模式下，摄像头会 24 小时不间断地拍摄室内的影像，并将这些视频数据存入服务端的数据服务器中。

- 触发捕捉功能

用户通过客户端发送监控请求时，服务器端才开始接收摄像头捕捉的视频数据。如果用户发送的是视频监控请求，则将捕捉的视频数据发送给手持终端，同时存入数据库。如果是图像监控请求，只需要从视频中抽取一幅图片，发送给远程用户。

- 定时捕捉功能

用户可以在终端上访问服务器端的功能设置页面，将视频捕捉模式设置为定时捕捉，并需要明确定时拍照，或者定时拍摄一段时间的视频。拍摄视频的时间用户可以设置。

（4）自动提醒功能

通过在用户室内的大门、窗台、厨房和卫生间里安装传感器，实现对非法入室、煤气泄漏和火灾监控。传感器捕捉到报警信号后，将该信号传到服务器。然后服务器将该信息转换为文本信息发送到终端用户。信息保存的最长时间为一个月。图 18-8 为自动提醒功能用例图。

（5）管理功能

管理功能用例图如图 18-9 所示。



图 18-8　自动提醒功能用例图

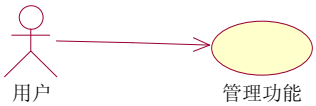


图 18-9　管理功能用例图

用户可以通过浏览器登录服务器端的网页，设置需要获得功能模式和管理个人信息，如密码、绑定的手机号等。图 18-10 为管理功能活动图。

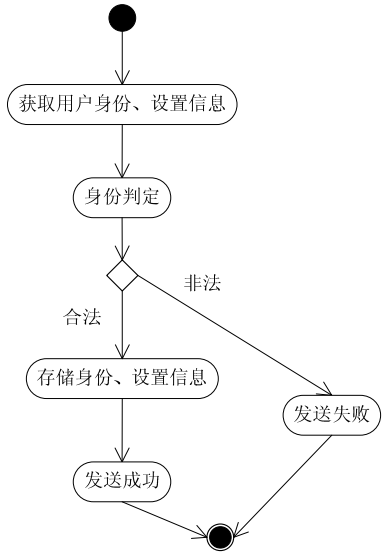


图 18-10　管理功能活动图

- 设置监控模式

用户登录后，可以在网页上设置是否需要自动提醒、是否需要实时拍摄、是否需要定时捕捉信息等功能。此类信息可以被保存在服务器端的数据库中。

- 设置数据安全传输模式

用户登录后，可以在网页上设置是否开启数据流加密功能，如果需要，以哪种方式进行加密。此类信息可以被保存在服务器端的数据库中。

- 个人信息管理

用户登录后，可以在网页上修改自己的个人信息，包括登录密码、绑定的手机号、电子邮箱以及一些可选的个人信息。此类信息可以被保存在服务器端的数据库中。

2．传感器系统主要功能

传感器系统主要的功能是：

（1）用户可以通过此子系统感应外界入侵；

（2）通过此子系统可以拍摄室内的情况，起到实时监控的作用；

（3）用户手动设置开启或关闭自动提醒功能。

系统的主要输入包括如下三部分来源。

- 当有外界入侵时红外感应器感应的外界热能；
- 来自服务器端的命令；
- 用户手动输入的控制命令。

系统的主要输出包括如下两部分。

- 摄像头所拍摄的图像；
- 异常提醒信息。

18.2 系统硬件结构

18.2.1 Linux 客户端系统硬件结构

硬件环境列表如表 18-1 所示。

表 18-1 硬件环境列表

硬件平台	名 称
手持设备（针对模拟机）	Intel Xscale PXA255 开发板
网络环境（针对模拟机）	校园 Internet 局域网

硬件规格如表 18-2 所示。

表 18-2 Intel Xscale PXA255 硬件规格

项 目	规格说明
Processor	Intel® XScale PXA255 400MHz
SDRAM	Samsung 64Mbyte
Flash	Intel® strata Flash 32MByte
Ethernet	CS8900A 10BaseT
Audio	AC’97 Stereo audio
Display	LG TFT LCD 6.4”(640 * 480)
Touch	ADS7843 touch screen
USB	USB Slave
PCMCIA	1 Slot
RTC	Real time clock RTC4513
IrDA	HDSL3600
CF	1 Slot
MMC	1 Slot

XScale 的 RISC 微处理器核心可以作为符合 ASSP（Applications Specific Standard Products）标准的嵌入式系统控制核心。它不但具有优异的数据处理功能，而且很适合大量多媒体数据信号的传输与处理。此外，XScale 处理器还具有协同微处理器接口，可以直接扩充 DSP 微处理器。利用 Intel 公司的先进制造技术，XScale 微处理器能够在片上集成高达 4 MB 的 SDRAM 和 32 MB 的闪存。

1．处理器特点

- 32 位 Xscale RISC 核心，兼容 ARM v.5TE 指令；
- 工作频率 200、300、400 MHz；
- 0.18 微米制程；
- 超标量执行；

- 特殊 40 位存储器，16 位 SIMD 指令（视频、音频处理）；
- Intel Strata 闪存高速同步接口；
- 32 KB 指令缓存，32KB 数据缓存；
- 多媒体流数据专用 2KB 缓存；
- 内存控制器：4 bank（最大 256MB），工作频率 100MHz，支持 2.5V 和 3.3V SDRAM、SRAM、ROM、闪存等，16 位、32 位总线带宽；
- 双通道 PCMCIA、CF 卡控制器位；
- MMC/SD 存储卡控制器；
- 15 个单位通用输入/输出接口，支持中断；
- 集成可编程频率合成器、计时器；
- 16 条 DMA 通道；
- LCD 显示控制器，支持填充、矩形单元变换硬件加速；
- AC97 音频；USB 接口（非主机接口）；
- UART（一个用于内部设备，一个提供全功能硬件传输控制）；
- 蓝牙、红外接口；
- I2C 和 I2S 总线；
- SSP 接口；
- 256 针 PBGA 封装，核心大小 17 mm×17 mm。

PXA255 功能框图如图 18-11 所示。

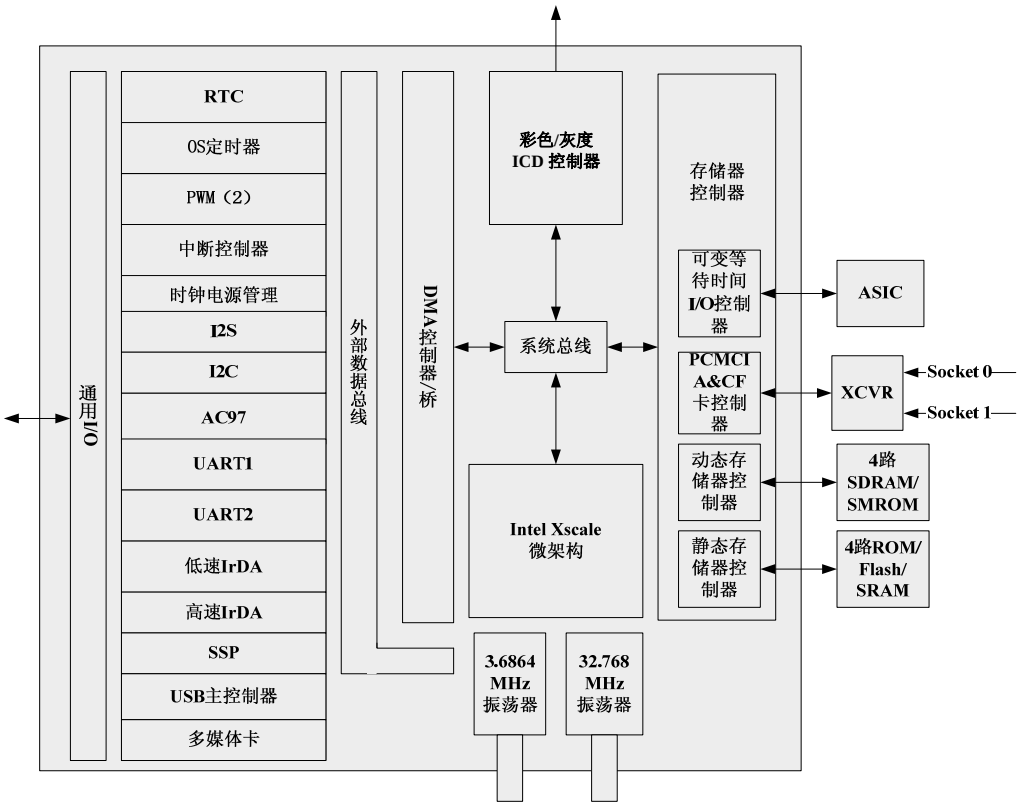


图 18-11　PXA255 功能框图

2．Intel PXA255 处理器功能特点

（1）存储器控制器

存储器控制器可以控制多种配置和组合的存储器，并提供了可编程定时的 glueless 控制信号。支持多达 4 MB 的 SDRAM 和 6 片静态存储器（SRAM、S DRAM、Flash、ROM、S ROM 等），支持 2 个 PCMCIA 或 CF 卡槽。

（2）时钟和电源控制器

PXA 255 处理器的各个功能模块由一个 3.6864MHz 晶振和一个可选的 32.768 kHz 晶振驱动。3.6864MHz 晶振驱动内部和外部锁相环，由锁相环为特定的功能模块提供指定的时钟频率。32.768kHz 晶振必须在硬复位后被选中，它负责驱动实时时钟、电源管理控制器和中断控制器。当处理器处于休眠状态时，32. 768 kHz 晶振还作为独立的电源模块为微处理器提供活动时钟。

（3）通用串行总线（USB）主控制器

USB 主控制器模块基于 USB 1.1 标准，支持多达 16 个终端设备并提供内部 48 MHz: 时钟。USB 设备控制器还提供先入先出的 DMA 内存直接读写功能。

（4）DMA 控制器（DMAC）

DMA 控制器提供 16 个优先级通道以支持内部设备的传输请求和外部配套片子的数据传输请求。基于描述符的 DMA 控制器还支持指令链和循环结构和 DMA 控制器。在进行外设到存储器、存储器到外设、存储器到存储器的数据传输时，工作于数据流模式。支持字节、字和双字数据传输模式。

（5）LCD 控制器

LCD 控制器支持被动（DSTN）和主动（TFT）LCD 屏显示方式，最高分辨率为 640×480 时显示 16 位增强色。2 个专用 DMA 通道使 LCD 控制器能够进行单屏或双屏显示。被动单色模式支持 256 阶灰度，被动彩色模式支持 64K 彩色，主动彩色模式也支持 64K 彩色。

（6）AC 97 控制器

AC 97 控制器支持 AC972 .0CODEC，能够以 48kHz 进行采样操作。控制器还提供独立的支持 DMA 先入先出的 16 位通道，以支持立体声 PCM 输入、立体声 PCM 输出、Modem 输入、Modem 输出和麦克风输入。

（7）多媒体卡（MMC）控制器

多媒体卡控制器提供到标准存储器卡的串行接口。控制器能以 20 Mbps 速率进行串行数据传输，能够控制两片 MMC.或 SPI 卡，并支持先入先出的 DMA 内存访问功能。

（8）快速红外（FIR）通信端口

基于 4 M bpsIrDA 的快速红外通信端口工作于半双工模式，支持先入先出的 DMA 内存访问。端口通过 STUART 的发射接收引脚直接与外部的 IrDA LED 收发器相连。

（9）同步串行协议（SSP）控制器

SSP 控制器提供全双工的串行接口，波特率为 7.2 kHz~1.84 MHz。支持 NationalSemiconductor 的 Microwirs，TI 的同步串行协议和 MOTOROLA 的串行外围设备接口协议。支持先入先出的 DMA 内存访问功能。

（10）通用目的 I/O（GPIO）

每一个 I/O 引脚都可以单独编程作为输入或输出线。其中输入线可以设定上升沿或下

降沿触发中断。主 GPIO 引脚不能与外设公用，次 GPIO 引脚可与外设通过映像公用。

（11）通用异步收发口（UART）

PX A 255 处理器提供 3 个通用异步收发口（UART），每个 UART 都可以用做低速红外数据（IrDA）传输口，并符合 Infrared DataAssociation 的串行红外物理层连接规范。

（12）实时时钟（RTC）

实时时钟可以使用两个晶振中的任何一个，当系统处于休眠状态时，使用 32.7 68kHz 晶振比使用 3. 6864 MHz 晶振功耗更小，但为了节省系统成本也可以不选用 32. 768 kHz 晶振。使用可编程的闹钟寄存器 RTC 可以提供连续的频率输出。闹钟寄存器也可以将处理器从睡眠状态中唤醒。

（13）脉宽调制器（PWM）

PWM 具有两个独立输出 C1，可以编程驱动两路 GPIO，且可以编程调节其频率和占空比。例如，2 路 PWM 输出可以分别用来控制 LCD 的对比度和亮度。

（14）中断控制

中断控制器引入内核的 IRQ 中断请求和 FIO 输入。使用 MASK 寄存器能够使能或关闭某个独立的中断源。

（15）网络同步串行协议端口（NSSP）

PXA255 处理器拥有专为连接到其他网络的 ASIC 而优化的 SSP 端口。此网络端口新增了针对 TXD 的 Hi-Z 功能，能够在 Hi-Z 发生时对 TXD/RXD 引脚进行控制和转换。

表 18-3 给出了软件支持环境列表，终端软硬件结构图如图 18-12 所示。

表 18-3　软件支持环境列表

软件平台	名　　称	作　　用
OSGi 框架	OSGi Framework	OSGi 为本系统服务器端软件的实现提供了安全框架
嵌入式操作系统	嵌入式 Linux	运行于手持设备、管理硬件，支持上层系统应用软件

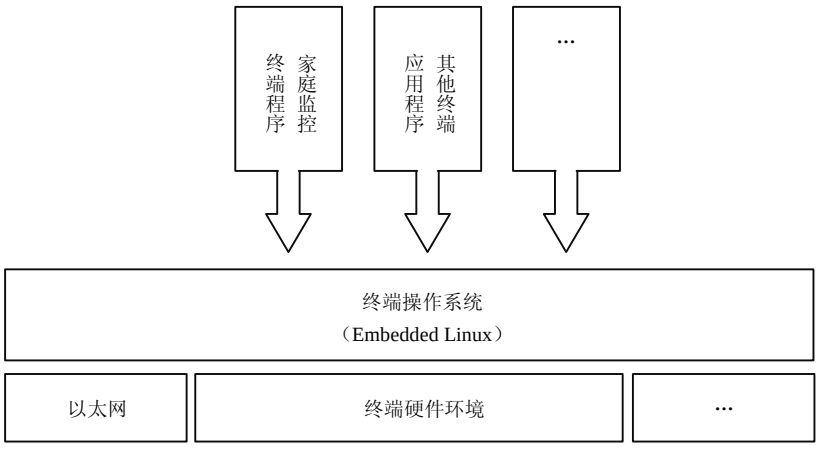


图 18-12　终端软硬件结构图

18.2.2　传感器系统硬件结构

图 18-13 为传感器系统硬件结构图。

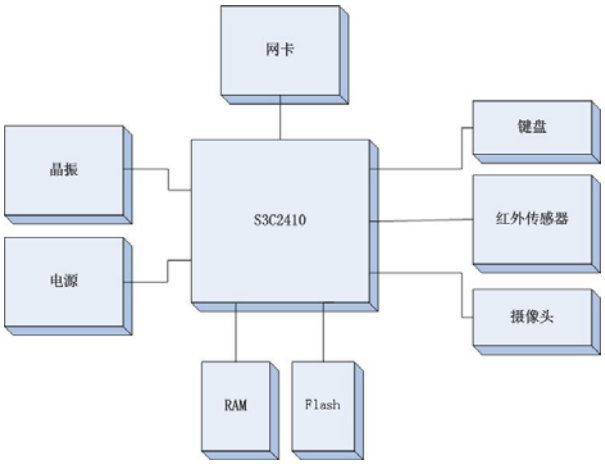


图 18-13　传感器系统硬件结构图

室内摄像设备：由安装于室内重点防区的摄像头组成。负责采集室内视频图像信号，并通过无线网络传输至主机，进行压缩后传输至服务器存储。

无线红外探头：任何物体因表面热度的不同，都会辐射出强弱不等的红外线。因物体的不同，其辐射的红外线波长亦有差异。红外探测主要用来探测人体和其他一些入侵的移动物体，当人体进入探测区域，稳定不变的热辐射被破坏，产生一个变化的热辐射，红外传感器接收后放大、处理，发出报警信号。

无线门磁探测器：是一种广泛使用、成本低、安装方便，而且不需要调整和维修的探测器。门磁开关分为可移动部件和输出部件。可移动部件安装在活动的门窗上；输出部件安装在相应的门窗上，两者安装距离不超过 10 毫米。输出部件上有两条线，正常状态为常闭输出，门窗开启超过 10 毫米，输出转换成为常开。

1．影像监控器

名称：台电 USB 摄像头

规格说明如下：

传感器	301PL+7131R Color
有效像素	30 万像素
支援解像度	VGA(640×480)
支援速度	VGA:15fps
最低照明度	5.0
镜头	F=2.0/4
镜头角度	60°
焦距	30mm
曝光	自动/手动
白平衡	自动/手动
信噪比	48dB 以上
接口	USB2.0
电源	DC5V +/-5% USB Power

2．红外探头

名称：SPIDER 红外探头 SP-9923

规格说明如下：

工作电压	10-16VDC
工作电流	12VDC/15mA
传 感 器	双元低噪声热电传感器
报警周期	2~5 秒
工作温度	-10~50 度
RFI 保护	20V/M-1000MHZ

3．EDUKIT-II S3C2410 开发板

名称：EDUKIT-II S3C2410 开发板

规格说明如下：

32M NandFlash	直流电机、步进电机模块
4Kbit IIC BUS 的串行 EEPROM	双以太网接口
2 个串口	8 段数码管
两个中断按钮	双 CAN 总线模块
4 个 LED	A/D、D/A 模块
320×240 STN 彩色 LCD 及 TSP 触摸屏	IDE 硬盘+CF 卡模块
4 × 5 键盘	SD 卡接口
20 针 JTAG 接口	Microphone 输入口
PS/2 接口	IIS 音频信号输出口
2 个 USB 主口和 1 个 USB 从口	固态硬盘 16M × 8bit
4 个 2 × 20PIN CPU 扩展接口	GPS 模块（选配）
红外线模块	GPRS 模块（选配）
蜂鸣器	全功能硬件仿真器，支持 ARM7 和 ARM9

18.3　系统软件结构

18.3.1　Linux 客户端系统软件结构

客户端系统划分为通信模块、显示模块、用户管理模块与系统设置模块四个功能模块。

其中，通信模块主要负责与家庭安全监控系统服务器端的通信，包括发送信息、接收数据和监听异常三个子功能模块。

显示模块主要负责监控数据的显示，包括图片显示和历史数据查询两个子功能模块。

用户管理模块主要负责用户信息的确认与维护，包括登录、注销和修改密码三个子功能模块。

系统设置模块主要负责对监控设备的设置，包括监控模式的设置和监控状态更改两个子功能模块。

图 18-14 给出了客户端中各个模块的结构图，指出了各个程序模块的层次关系。

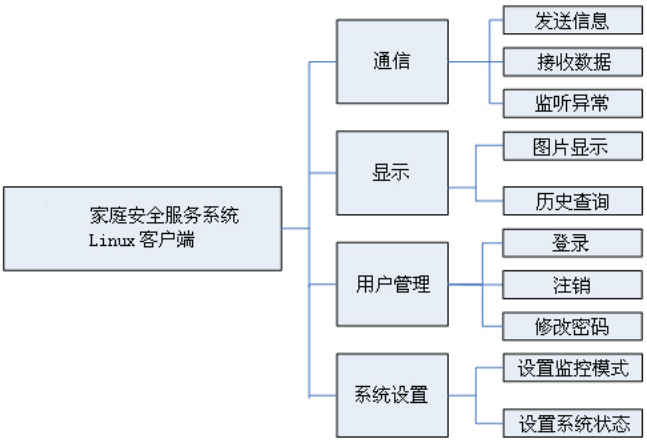


图 18-14　系统模块结构图

1．软件系统分析

Intel Xscale PXA255 软件规格具体如表 18-4 所示。

表 18-4　Intel Xscale PXA255 软件规格

项　　目	描　　述
O/S	Linux 2.4.18 kernel
Device Driver	CS8900 Ethernet
	AC’97 Stereo audio
	Framebuffer
	ADS7843 touch screen
	USB
	PCMCIA(包括 CF driver)
	RTC4513(Real Time Clock)
	IrDA
	MMC
File System	JFFS2
GUI+Applications software	事先指定的

2．Linux 操作系统

作为候选的嵌入式操作系统，Linux 有一些很吸引人的优势，它已经被成功移植到多个不同结构的 CPU 和硬件平台上，具有很好的稳定性和持续升级能力，而且对各种硬件有良好的驱动支持。它有下面几个特点。

（1）嵌入式 Linux 由于代码开放性，不存在黑箱技术，遍布全球的众多 Linux 爱好者就是 Linux 开发的强大技术后盾；

（2）嵌入式 Linux 拥有强大的网络功能，在嵌入式联网日趋成熟的今天，这个优势非常明显，Linux 甚至支持蓝牙等新技术，其他任何操作系统对于新技术的支持远没有 Linux 那么迅速；

（3）Linux 的内核小，功能强大，运行稳定，系统健壮，效率高；

（4）Linux 沿用了 UNIX 的发展模式，遵循国际标准，可以方便地获得第三方软硬件厂商的支持；

（5）Linux 有非常优秀的完整开发工具链，有十几种集成开发环境，其中很多是免费的，大大降低了开发费用。Linux 的自由精神吸引了成千上万的程序员投入到 Linux 的开发和测试中来，使得 Linux 在短时间内就成为一个功能强大的操作系统。

在使用上，根据 PXA255 开发板手册选择合适的硬件驱动模块。同时，将内存管理单元、文件系统和网络通信协议栈作为模块编译入内核。

目前，国际上一些著名的公司已经选中了嵌入式 Linux 作为开发嵌入式产品的工具，如三星公司的 Linux PDA、Transmeta 公司的 Linux 智能手机、NetGem 的机顶盒等。由此不难看出 Linux 作为开发嵌入式产品的操作系统所具备的巨大潜力。

3．应用软件

应用软件在调用 Linux 系统接口函数下，具备以下模块。

（1）数据接收模块。能够接收来自服务器端发送的数据。

（2）数据分析模块。对接收的数据进行分析，区分出视频、图像还是文本信息。

（3）视频数据解码播放模块。对接收的视频数据进行解码并能播放。

（4）图片数据解码显示模块。对接收的图片数据进行解码并能显示。

（5）数据发送模块。对于向服务器端发送的数据请求和服务端设置数据要能够发送给服务器。

系统软件结构如图 18-15 所示，移动终端软件系统包括 Linux 操作系统和应用软件系统。应用软件系统负责接收视频数据和服务器反馈信息，并能发送相关操作指令，该系统由用户直接使用，要求易用、稳定、低耗。

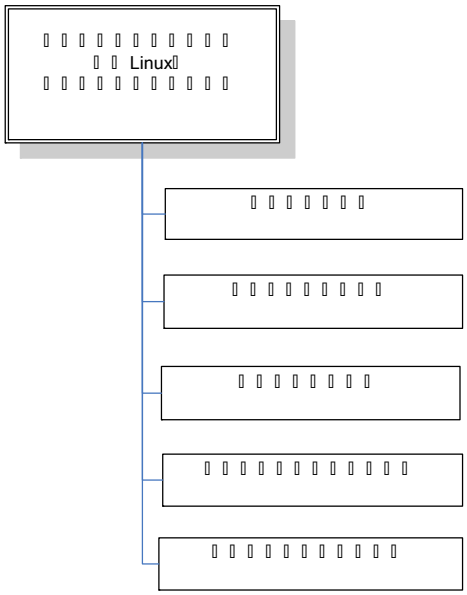


图 18-15　系统软件结构图

4．系统工作流程

本系统为家庭安全监控服务系统的基于 Linux 的移动通信服务终端子系统。当用户向服务器请求查看时，移动通信服务器将数据通过无线网络传送至用户的移动通信终端。用户也可以通过该终端进行各项系统参数的设置。

该终端的数据处理流程如图 18-16 所示。

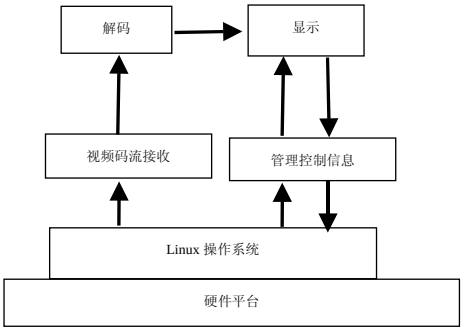


图 18-16　基于 Linux 的移动通信服务终端子系统的系统模型图

18.3.2　传感器系统软件结构

系统软件结构如图 18-17 所示。

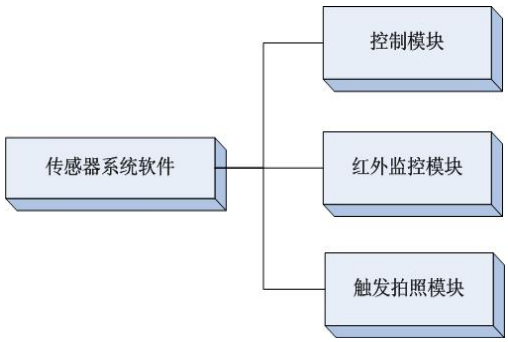


图 18-17　传感器软件结构图

1．嵌入式操作系统

Linux 是嵌入式系统的可靠主力，其能更快地支持新的 IP 协议和其他协议。例如，用于 Linux 的设备驱动程序要比用于商业操作系统的设备驱动程序多，如网络接口卡（NIC）驱动程序以及并口和串口驱动程序。

最终采用内核版本为 2.4 的嵌入式 Linux 作为传感器设备的操作系统。根据 2410 开发板手册选择合适的硬件驱动模块。操作系统对 2410 开发板上的软硬件资源进行管理，主要负责红外传感器信号收发、处理，图片数据采集以及与服务器通信。

2．开发环境说明

传感器系统采用了英蓓特公司 Embest EduKit2410 （ARM920T）作为硬件平台。其软件开发环境亦采用了该系统的配套开发环境。

以下为该环境的简要说明，详细使用说明请参考开发板光盘中的用户手册。

（1）Linux 开发环境

本项目采用 Cygwin 作为 Linux 开发环境。Cygwin 是 Cygwin 公司（http://cygwin.com/）的产品，它提供了 Windows 操作系统下的一个 UNIX 环境，它可以帮助程序开发人员把应用程序从 UNIX/Linux 移植到 Windows 平台，是一个功能强大的工具集。Cygwin 可以从其网站 http://www.cygwin.com 上下载并安装最新版本。本项目中使用的版本为 Cygwin，1.5.10，版本的发布日期为 2004 年 5 月。

（2）Embest IDE 开发环境

传感器系统开发宏采用 Embest online Flash Programmer for ARM 烧写启动文件、内核映像和文件系统到 Nor Flash 中。

（3）文件传输服务器

在传感器系统开发过程中，需要经常在开发板系统与 PC 之间传送文件，这里采用了较常用的 tftp 小型文件传送协议，来实现与开发板系统进行驱动程序安装及应用程序测试。

首先需要在 PC 上运行一个 tftp 服务器，并进行各种工作状态、权限以及本地 tftp 工作目录的设置。

在传感器系统最终的实现中，也是通过 tftp 服务器进行系统的驱动程序及应用程序的安装。

（4）编译工具

在传感器系统开发过程中，采用了 Embest 2410 开发板光盘中提供的编译工具包 cross-armtools-linux-edukit2410。

3．系统工作流程

（1）陌生人入侵时触发红外传感器，系统判断采集到的红外信号，若有陌生人入侵则将报警信息发送到服务器；

（2）系统根据服务器命令驱动摄像头拍照，并将图片数据传递至服务器；

（3）用户通过键盘控制自动提醒功能的开启与关闭。

工作流程示意图如图 18-18 所示。

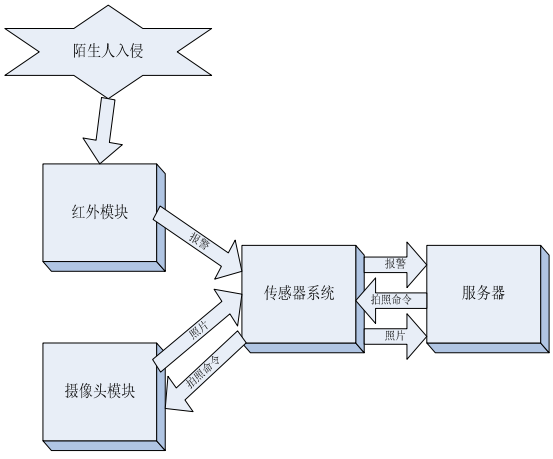


图 18-18　工作流程示意图

18.4 Linux 客户端系统设计实现

用户的客户端操作（客户端软件界面切换和文本信息显示），由界面、文件管理与显示模块处理。

对于监视功能：由界面、文件管理与显示模块接收命令，传给管理控制信息编码与传递模块，再发送给服务器。从服务器获取的数据传给图片和视频码接收模块，经图片和视频码解码模块解析数据，再由界面、文件管理与显示接口显示信息。

对于其他功能：由界面、文件管理与显示模块接收命令，传给管理控制信息编码与传递模块，再发送给服务器。

从键盘和鼠标得到外部控制数据和信号，交给界面、文件管理与显示模块处理，如果需要发送给服务器端，则再经管理控制信息编码与传递模块发送给服务器。

客户端系统划分为通信模块、显示模块、用户管理模块与系统设置模块四个功能模块。

- 通信模块主要负责与家庭安全监控系统服务器端的通信，包括发送信息、接收数据和监听异常三个子功能模块。
- 显示模块主要负责监控数据的显示，包括图片显示和历史查询两个子功能模块。
- 用户管理模块主要负责用户信息的确认与维护，包括登录、注销和修改密码三个子功能模块。
- 系统设置模块主要负责对监控设备的设置，包括设置监控模式和设置系统状态两个子功能模块。

18.4.1 系统数据结构设计

1. 逻辑结构设计要点

本客户端系统用到的关键数据结构如下。

用户账户信息：

```
Struct Account{              //账户信息，81 个字节长
char telephoneNum[11];       //手机号，11 个字节长
    char password[20];       //密码，20 个字节长
char mailAddress[50];        //电子邮箱地址，50 个字节长
}
```

命令字：

```
unsigned char command;       //1 个字节长
```

JPEG 格式：

JPEG 的每个标记都是由 2 个字节组成，其前一个字节是固定值 0xFF。每个标记之前还可以添加数目不限的 0xFF 填充字节（fill byte）。下面是其中的 8 个标记。

```
SOI  0xD8              图像开始
APP0 0xE0              JFIF 应用数据块
APPn 0xE1 - 0xEF       其他的应用数据块（n,1~15）
DQT  0xDB              量化表
SOF0 0xC0              帧开始
DHT  0xC4              霍夫曼（Huffman）表
SOS  0xDA              扫描线开始
EOI  0xD9              图像结束
```

2．物理结构设计要点

用户账户信息和命令字在客户端无存储要求。图片使用 JPEG 格式，在客户端至少要求存储最近查看的 10 张图片，存取单位为张。视频流在客户端无存储要求。

18.4.2　通信模块设计说明

1．程序描述

通信模块主要负责与家庭安全监控系统服务器端的通信，包括发送信息、接收数据和监听异常三个子功能模块。各部分功能及特点对照如表 18-5 所示。

表 18-5　通信模块子功能模块特点对照

	发送信息	接收数据	监听异常
功能	将数据发送到服务器	接收服务器发送到客户端的数据	不断查询服务器端发送的异常信号
是否常驻内存	否	否	是
是否子程序	是	是	是
是否可重入	是	是	是
有无覆盖要求	无	无	无
顺序或并发处理	顺序	顺序	并发

2．功能

发送信息子模块将客户端的数据准备好后通过 socket 传输到服务器，如图 18-19 所示。

接收数据子模块将服务器端通过 socket 发送到客户端的数据接收到本地，如图 18-20 所示。



图 18-19　发送信息子模块 IPO 图



图 18-20　接收数据子模块 IPO 图

监听异常子模块不断向服务器端发送监听请求，从服务器端获取是否有异常的信号，如图 18-21 所示。

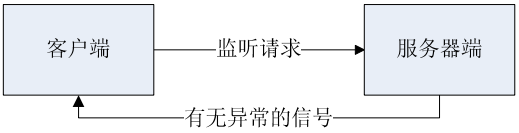


图 18-21　监听异常子模块 IPO 图

3．输入项

表 18-6 列出了通信模块输入项。

表 18-6　通信模块输入项

功 能 项	命令码标识	命令码宏对应值	数据格式和长度
操作成功	E_SUCCESS	2	根据需要加载数据。目前只有图片的成功反馈信息需要加载数据，格式为大小（32 位，单位字节）+数据（单位字节），表示单张图片

续表

功 能 项	命令码标识	命令码宏对应值	数据格式和长度
用户名或者密码错误	E_ACCOUNT	4	无
服务超时	E_TIMEOUT	6	无
请求的资源不存在	E_NOEXIST	8	无
无法开启相应设备	E_DEVICE	10	无
自动提醒提示	E_WARNNING	12	格式：提示码（8 位，单位字节） 注：目前只有家里入侵提示（值为 1）
保留	E_UNDEFINE	14	无

4．输出项（如表 18-7 所示）

表 18-7　通信模块输出项

功 能 项	命令码标识	命令码宏对应值	数据格式和长度
用户登录	C_LOGIN	1	用户名（64 位）+密码（64 位）
用户注销	C_LOGOUT	3	用户名（64 位）
触发监控	C_PIC_TRIGGER	5	无
查看历史	C_PIC_HISTORY	7	起始时刻（32 位，从 1970.01.01 00：00：00 算起，单位 s）+终止时间（32 位，从 1970.01.01 00：00：00 算起，单位 s）
定时拍照	C_PIC_PERIOD	9	起始时刻（32 位，从 1970.01.01 00：00：00 算起，单位 s）+间隔时间（32 位，单位 s）
修改密码	C_PASSWORD	11	旧密码（64 位）+新密码（64 位）
开启监控	C_OPENSYSTEM	13	无
关闭监控	C_CLOSESYSTEM	15	无
开启自动提醒	C_OPENWARNNING	17	无
关闭自动提醒	C_CLOSEWARNNING	19	无
收到自动提醒应答	C_ACKWARNNING	21	无

5．算法

发送信息子模块和接收数据子模块，采用顺序处理的方式，属于主进程，并为监听异常子模块创建一个子进程，使其常驻内存。

6．流程逻辑

三个子模块均采用 socket 方式与服务器通信，符合 socket 的标准通信流程。参照通信模块的 IPO 图。

18.4.3　显示模块设计说明

1．程序描述

图像显示子模块用于提供系统接口给用户，使用户能够主动查看当前时刻的监控图像，也可以在开启自动监控的情况下当有异常发生时，显示获取的异常情况的图像数据。历史查询子模块用于提供系统接口给用户，使用户能够查看数据库中某一历史时刻的图像。表 18-8 列出了显示模块子模块特点。

表 18-8　显示模块子模块特点

	图片显示	历史查询
功能	主动查询当前时刻的监控图像，被动显示异常情况的图像数据	查看数据库中某一历史时刻的图像
是否常驻内存	否	否
是否子程序	是	是
是否可重入	是	是
有无覆盖要求	无	无
顺序或并发处理	顺序	顺序

2．功能

显示模块的 IPO 图如图 18-22 所示。

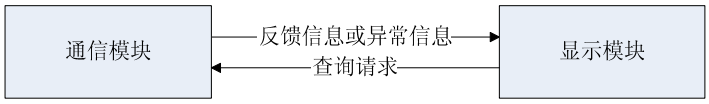


图 18-22　显示模块 IPO 图

3．输入项

- 通信模块获得的图像数据以全局数组的形式存放，可供通信模块和显示模块读写。
- 从通信模块获取图像失败的反馈信息。具体信息标准参照通信模块的输入项。

4．输出项

向通信模块传递时间信息和查询请求，时间信息采用日期型。

5．算法

顺序执行，图像数据采用全局数组。

6．流程逻辑

参照显示模块的 IPO 图。

18.4.4　用户管理模块设计说明

1．程序描述

用户管理模块用于提供系统接口给用户，使用户能够登录系统、手动设置密码以及注销用户，暂不可以添加用户（服务器端的工作）。本程序对全局变量用户名进行读操作，对全局变量用户密码进行读写操作，因此是不可重入的。表 18-9 列出了用户管理模块子模块特点。

表 18-9　用户管理模块子模块特点

	登　录	注　销	修改密码
功能	输入用户名和密码，登录到服务器	注销当前用户	修改当前用户的密码
是否常驻内存	否	否	否
是否子程序	是	是	是

续表

	登　　录	注　　销	修改密码
是否可重入	否	否	否
有无覆盖要求	无	无	无
顺序或并发处理	顺序	顺序	顺序

2．功能

登录子模块 IPO 图如图 18-23 所示。

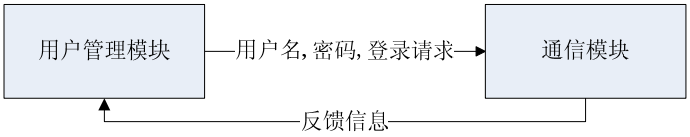


图 18-23　登录子模块 IPO 图

注销子模块 IPO 图如图 18-24 所示。

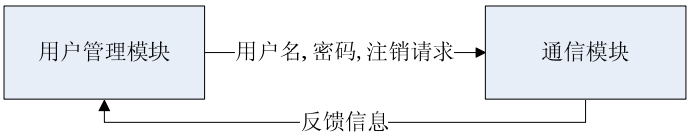


图 18-24　注销子模块 IPO 图

修改密码子模块 IPO 图如图 18-25 所示。

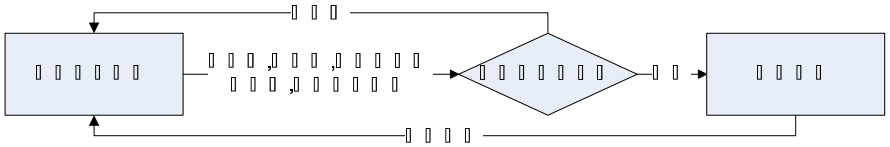


图 18-25　修改密码子模块 IPO 图

3．输入项

从通信模块获取的反馈信息，具体信息标准参照通信模块的输入项。此外还有用户信息的输入项，如表 18-10 所示。

表 18-10　用户信息的输入项

名　　称	标 识 符	备　　注	数据类型	有效范围	输入方式
用户名	user	手机号码	char[]	11 位	手机键盘
密码	password	数字或大小写字母	char[]	6 位	手机键盘

4．输出项

向通信模块传递登录、注销和修改密码请求及用户信息的输入项。

5．算法

顺序执行。

6．流程逻辑

参照用户管理模块的 IPO 图。

18.4.5　系统设置模块设计说明

1．程序描述

设置监控模式子模块用于提供系统接口给用户，使用户能够手动设置实时或定时监控模式。本程序会读写全局变量监控模式，因此是不可重入的。

设置系统状态子模块用于提供系统接口给用户，使用户能够手动开关本监控系统提供给客户端的监控的服务。本程序会读写全局变量系统状态，因此是不可重入的。

系统设置模块将用户输入的参数通过通信模块传送至服务器。表 18-11 列出了系统设置模块子模块特点。

表 18-11　系统设置模块子模块特点

	设置监控模式	设置系统状态
功能	手动设置实时或定时监控模式	手动开关本监控系统提供给客户端的监控服务
是否常驻内存	否	否
是否子程序	是	是
是否可重入	否	否
有无覆盖要求	无	无
顺序或并发处理	顺序	顺序

2．功能

系统设置模块 IPO 图如图 18-26 所示。

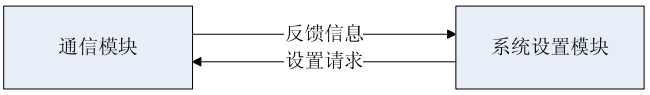


图 18-26　系统设置模块 IPO 图

3．输入项

从通信模块获取的反馈信息，具体信息标准参照通信模块的输入项。

4．输出项

将用户的设置请求传递给通信模块。

5．算法

顺序执行。

6．流程逻辑

参照系统设置模块的 IPO 图。

18.4.6　客户端主要代码与注释

1．主函数

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <unistd.h>
#include <qapplication.h>
#include <qdialog.h>
#include <qmessagebox.h>
#include <pthread.h>
#include "CMainForm.h"
#include "CWarningForm.h"
void* waitForWarning(void* parent);
int main(int argc, char *argv[])
{
   int ret;
   pthread_t pid;
    QApplication app(argc, argv);
    CMainForm form;
       QFont font1( "unifont",16,50,FALSE,QFont::Unicode);
       app.setFont(font1);
    pthread_create(&pid, NULL, &waitForWarning, NULL);//建立监控线程
    app.setMainWidget(&form);
    form.setFont(font1);
       form.show();
    ret = app.exec();
    pthread_cancel(pid);
   return ret;
}
```

2. 监控程序

```
void* waitForWarning(void* args)//开启监控
{
       args = NULL;
    qWarning("The monitoring process starts.\n");
    int connfd, sockfd;
    struct sockaddr_in servaddr;
    struct sockaddr_in cliaddr;
    socklen_t clilen;//客户端 socket
    int ret;
    char buf[3];
    //建立 socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd == -1){
        qWarning("socket init failed.\n");
        return NULL;
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(8093);
    //bind 绑定 socket
    ret = bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    if(ret == -1){
        qWarning("bind error.\n");
        return NULL;
    }
```

```
    //listen 监听
    ret = listen(sockfd, 10);
    if(ret == -1){
        qWarning("listen error.\n");
        return NULL;
    }
    for(;;){
        clilen = sizeof(cliaddr);
//客户端接收服务器发的流
    connfd = accept(sockfd, (struct sockaddr*)&cliaddr, &clilen);
if(connfd == -1)
{
            qWarning("receive client connection failed.\n");
            continue;
        }

        ret = read(connfd, buf, 2);
        if(ret == -1){
            qWarning("read error.\n");
            continue;
        }
        buf[ret] = '\0';
        if((buf[0] == 12) && (buf[1] == 1)){
           int fd;
           fd = vfork();
           if(fd == 0){
              execl("/home/leek84/warning","warning",NULL);
           }else{
            qWarning("invalid intruding haha");
           }

        }else{
           qWarning("invalid command");
        }
        buf[0] = 21;
        write(connfd, buf, 1);

        close(connfd);
    }
       qWarning("The monitoring process ends.\n");

}
```

3．Qt 图形用户界面

限于篇幅这里省略，请读者参考光盘中代码。

18.5　系统主要模块设计实现

18.5.1　红外监控模块设计说明

红外模块是一个分布式的程序，由红外信号采集模块及报警模块两个子模块组成，如图 18-27 所示，其中红外信号采集模块周期性采集外界红外信息，并不断对采集到的信息进行处理；当系统根据采集到的红外信息判断有陌生人入侵时，调用报警模块，向服务器

发出报警信息。

1．功能

包括如下功能。

- 接收红外信号。
- 将红外信号转换成控制信息。

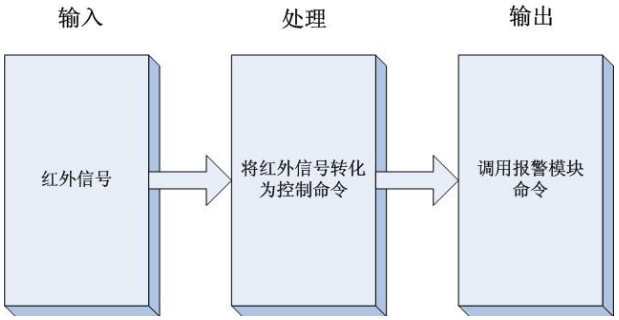


图 18-27　红外模块流程

2．性能

从接收红外信号到最后发送出去，时间不超过 1s。

3．输入项（如表 18-12 所示）

表 18-12　红外监控模块输入项

输入项名称	频　　度	输入数据来源	输入方式
红外信号	触发式，无固定频率	红外感应器	内部电路

4．输出项（如表 18-13 所示）

表 18-13　红外监控模块输出项

输出项名称	频　　度	输出目的地	输出方式
控制信号	触发式，无固定频率	报警模块	全局变量

5．算法

本模块无特殊的算法，无限循环方式采集红外信息，并将采集到的数据进行处理。

6．流程逻辑（如图 18-28 所示）

18.5.2　报警模块（warnning）

报警模块处理过程如图 18-29 所示。

1．功能

- 红外信号接收模块发送过来的数据。
- 对数据进行进行处理（解包/压包）。
- 处理后的数据通过串口发送到服务器。

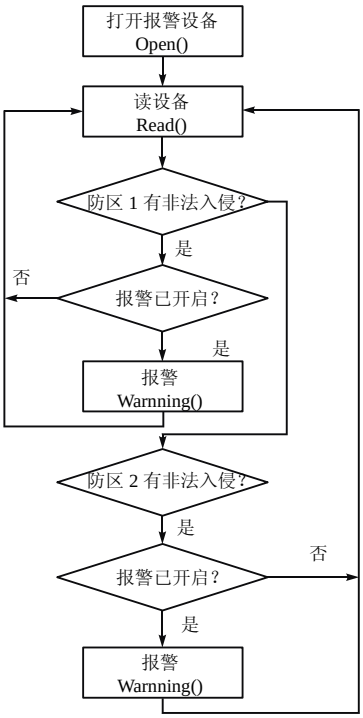


图 18-28　流程逻辑图

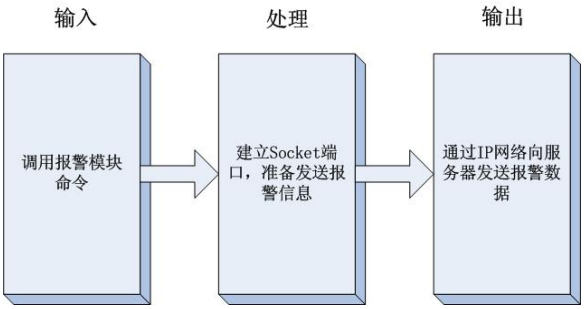


图 18-29　报警模块处理过程

2．性能

整个输入、处理、输出过程不能超过 1s。

3．输入项（如表 18-14 所示）

表 18-14　报警模块输入项

输入项名称	频　　度	输入数据来源	输入方式
控制信号	触发式，无固定频率	红外感应模块	程序内部

4．输出项（如表 18-15 所示）

表 18-15　报警模块输出项

输出项名称	频　　度	输出目的地	输出方式
报警信息	触发式，无固定频率	服务器	IP 网络

5．算法

无特殊算法，顺序执行，发送报警数据。

6．流程逻辑（如图 18-30 所示）

18.5.3　触发监控模块

触发监控模块处理过程如图 18-31 所示。

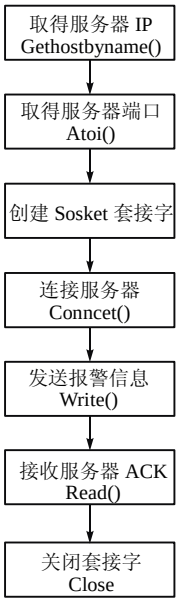


图 18-30　流程逻辑图

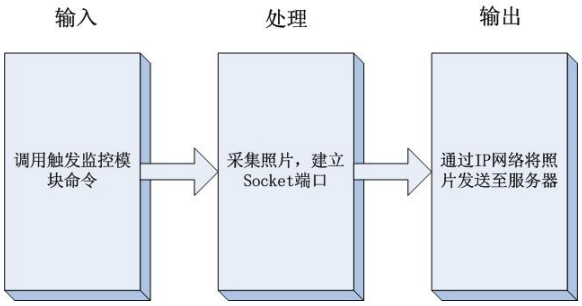


图 18-31　触发监控模块处理过程

1．功能

服务器发送触发监控请求时，系统调用该模块，拍摄室内照片并将照片发送至服务器。

2．性能

从接收拍照命令到将照片传输至服务器，时间不能超过 1s。

3．输入项（如表 18-16 所示）

表 18-16　触发监控模块输入项

输入项名称	频　　度	输入数据来源	输入方式
触发拍照命令	触发式，无固定频率	服务器	Socket

4．输出项（如表 18-17 所示）

表 18-17　触发监控模块输出项

输出项名称	频　　度	输出目的地	输出方式
图像数据数据包	触发式，无固定频率	服务器	Socket

5．算法

监听 8093 端口，接收数据若为有效数据，则进行拍照并将照片发送至服务器。

6．流程逻辑（如图 18-32 所示）

18.5.4　管理模块

1．功能描述

功能 1：用户按下键盘 0 键，开启自动提醒功能，如图 18-33 所示。

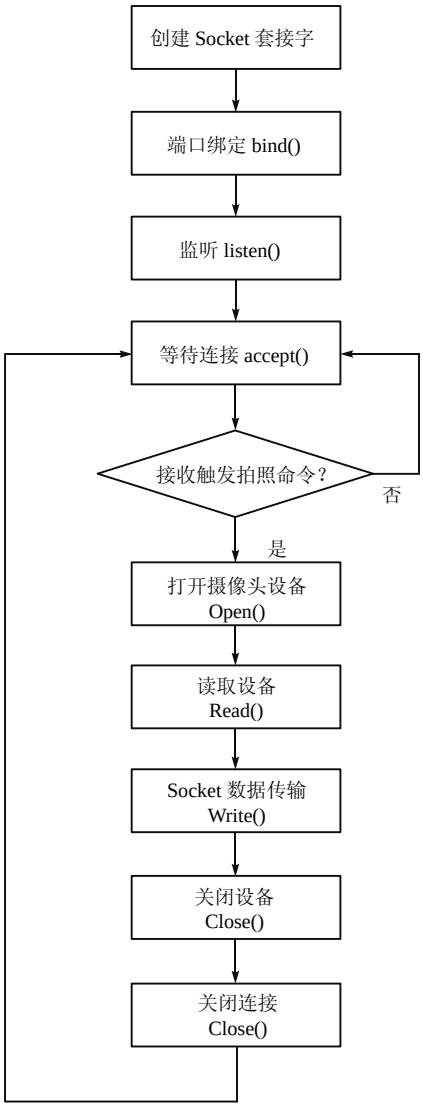


图 18-32　流程逻辑图

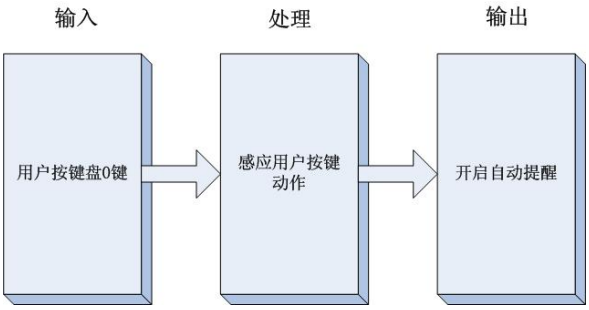


图 18-33　功能 1 处理示意图

功能 2：用户按下键盘 1 键，关闭自动提醒功能，如图 18-34 所示。

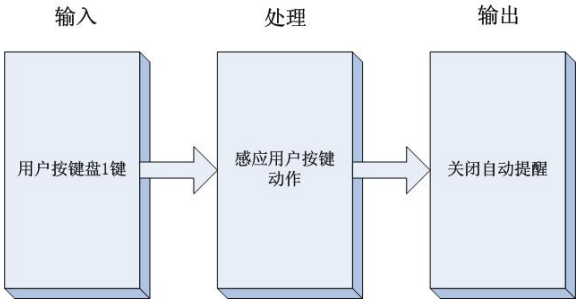


图 18-34　功能 1 处理示意图

2．性能

- 从用户按下开启键，到系统开启自动提醒功能，时间不能超过 1s。
- 从用户按下关闭键，到系统关闭自动提醒功能，时间不能超过 1s。

3．输入项（如表 18-18 所示）

表 18-18　管理模块输入项

输入项名称	类型与格式	输入数据来源	输入方式
开启命令	二进制数据	开启按键	按键
关闭命令	二进制数据	关闭按键	按键

4．输出项（如表 18-19 所示）

表 18-19　管理模块输出项

输出项名称	类型与格式	输出目的地	输出方式
开启状态标志	布尔型变量	传感器系统	全局变量
关闭状态标志	布尔型变量	传感器系统	全局变量

5．算法

键盘采用行列扫描法，采集到按键动作后判断，若为开启命令，则设置工作状态全局变量为 ON；若为关闭命令，则设置工作状态全局变量为 OFF。

6．流程逻辑（如图 18-35 所示）

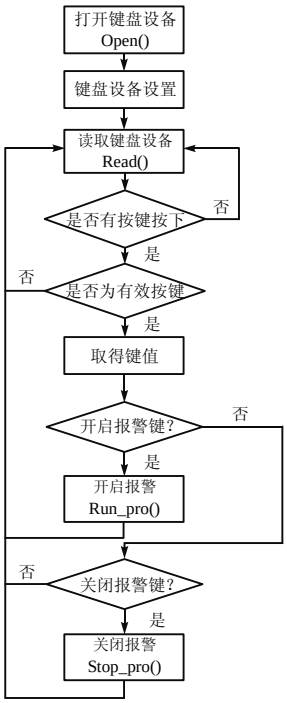


图 18-35　流程逻辑图

18.5.5　主要代码与注释

1. 图片采集

（1）代码段

```
int v4lopen(char *name, v4ldevice * vd)                //打开 V4L 视频设备
{
    if((vd->fd = open(name, O_RDWR)) < 0){
        v4lperror("open error!\n");
        return -1;
    }
    if(ioctl(vd->fd, VIDIOCGCAP, &(vd->capability)) < 0){
        v4lperror("Get capability error!\n");
        return -1;
    }else{
        printf("Name = %s\n", vd->capability.name);
        printf("maxwidth=%d, maxheight=%d, minwidth=%d,minheight=%d\n",
                vd->capability.maxwidth, vd->capability.maxheight,
                vd->capability.minwidth, vd->capability.minheight);
    }
    return 0;
}
int v4lgrabinit(v4ldevice *vd, int width, int height) // V4L 视频设备初始化
{
    vd->picture.palette = 21;
    if(ioctl(vd->fd, VIDIOCGPICT, &(vd->picture)) < 0){
        v4lperror("v4lgetpicture error!\n");
        return -1;
    }
    vd->picture.palette = 21;
    if(ioctl(vd->fd, VIDIOCGMBUF, &(vd->mbuf)) < 0){
        v4lperror("v4lgetmbuf error!\n");
        return -1;
    }
    vd->mbuf.size = width * height;
    if((vd->map  =  (unsigned  char*)mmap(0,  vd->mbuf.size,  PROT_READ  |
PROT_WRITE,MAP_SHARED, vd->fd, 0)) < 0){
        printf("mvuf.size1=%d\n", vd->mbuf.size);
        v4lperror("mmap failed!\n");
        return -1;
    }
    vd->mmap.width = width;
    vd->mmap.height = height;
    vd->mmap.format = vd->picture.palette;
    printf("vd->mbuf.szie = %d\n", vd->mbuf.size);
    vd->frame = 0;
    vd->framestat[0] = 0;
    vd->framestat[1] = 0;
    return 0;
}
int v4lclose(v4ldevice *vd)                          //关闭 V4L 视频设备
{
    close(vd->fd);
    return 0;
}
read(device.fd, buf, bufsize);                       //读取一张照片
```

（2）特点说明（关键技术、算法等）：

- 市场上应用最广泛的是采用中芯微公司生产的 zc301 芯片的摄像头，在传感器系统中的摄像头采用的是台电 WE-MK02-G35A1 摄像头，其采用的是 zc301 芯片。而 Linux 无该芯片的摄像头驱动，首先需要移植 zc301 摄像头驱动。

摄像头驱动程序采用了开源项目 spca5xx，将驱动程序放至/kernel/driver/usb 下，重新编译内核，生成了 301 摄像头所需的所有驱动程序。

- 图片采集程序参考了 V4L 编程。Video4Linux 是为市场现在常见的电视捕获卡和并口及 USB 口的摄像头提供的编程接口，同时也提供无线电通信和数字电视广播解码和垂直消隐的数据接口。本小节主要针对 USB 摄像头设备文件/dev/video0，进行视频图像采集方面的设计。

视频编程的流程如下。

- 打开视频设备；
- 读取设备信息；
- 进行视频采集，有内存映射和直接从设备读取两种方法；
- 对采集的视频进行处理；
- 关闭视频设备。

2．红外信号采集

（1）代码段

```
ret = read (buttons_fd, &key_value, sizeof key_value);
if ( key_value==128) {
                    printf("Invalid Intrude!\n");
                    key_value =  0;
                    if (state==ON)
                    warnning();

            }
            else if( key_value==167){
                    printf("Fire Warnning!\n");
                    key_value =  0;
                    if(state==ON)
                    warnning();
                }
```

（2）特点说明：

常见的传感器设备的输出表现为继电器特性，即无报警状况下电压为 0，输出电阻为无穷大；报警状态下输出电压为 0，输出电阻为 0。而 EDUKIT-II S3C2410 开发板上没有为该特性设备预留的输入接口。可以观察到该开发板上的 button 按键有与传感器设备同样的继电器输出特性。针对该特点，将传感器设备连接到开发板上的 button 按钮，根据读取 button 按钮键值的方法读取传感器设备，弥补了开发接口不足的问题。

3．控制功能

（1）代码段

```
ioctl(fd, I2C_SET_DATA_ADDR, REG_Sys);
read(fd, &key, 1);
if(key & 0x01) {
```

```
        ioctl(fd, I2C_SET_DATA_ADDR, REG_Key);
        read(fd, &key, 1);
        if(key > 0){
            ioctl(fd, I2C_SET_DATA_ADDR, REG_Key);
            read(fd, &key, 1);
        if(key > 0){
            cx = ((key-1) % 8) + 1; cy = ((key-1) / 8) + 1;
            key = key_set(key);
            printf("down\n");
            if(key != 0xFE){
                    if(key<16){
                        if(key<10)  key += 0x30;
                        else        key += 0x37;
                    }
                    printf("get key!\n");
                    if (key=='0'){
                        printf("run\n");
                        run_pro();
                    }
                    if(key=='1'){
                        printf("stop\n");
                        stop_pro();
                    }
                }else{
                    printf("else\n");
                }
            }
          }
        }
```

（2）特点说明：

算法是：键盘通过常用的行列扫描法判断键值。

4. 键盘驱动

```
#include "butt-s3c2410.h"
#ifdef CONFIG_PROC_FS
static int buttons_proc(char *page, char **start, off_t off,
                     int count, int *eof, void *data)
{
    char *p = page;
    *eof =0;
    *start = page + off;
    return 0;
}
#endif
#define BUTT_DEVICE_NAME "buttons"              /* 设备名 */
#define BUTTON_MAJOR 232                        /* 主设备号 */
wait_queue_head_t buttons_wait;
 static int flag = 0;
static struct butt_info{
  /* 分配设备使用的 CPU 资源*/
    int irq_no;
    unsigned int gpio_port;
    int butt_no;
}butt_info_tab[2]={
    {IRQ_EINT0, GPIO_F0, 0},
```

```
        {IRQ_EINT11, GPIO_G3,39},
        };
    static int ready = 0;
    static int key_value = 0;
    /*-----------------------------------*
     *         中断      *
     *-----------------------------------*/
    static void butt_isr_handler (int irq, void *dev_id, struct pt_regs *reg)
    {
        struct butt_info *k;
        int found = 0;
        int up;
        int flags;
                                                /*将中断号分配给两个按钮*/
        if (butt_info_tab[0].butt_no == irq) {
            found = 1;
            k=&(butt_info_tab[0]);
        }
        else if (butt_info_tab[1].butt_no == irq){
            found =1;
            k=&(butt_info_tab[1]);
        }
        if (!found) {
            printk ("bad irq %d in button!\n",irq);
            return;
        }
        save_flags (flags);                     /* 保存现在的中断状态*/
        cli();                                  /*  设置 cspr */
    //重用中断端口,读取端口等级
          GPCON(GRAB_PORT((k->gpio_port)))&=~(0x3<<(GRAB_OFS((k->gpio_
port))*2));
        GPCON(GRAB_PORT((k->gpio_port)))|=((GPIO_MODE_IN)<<(GRAB_OFS
((k->gpio_port))*2));
        up = read_gpio_bit(k->gpio_port);
        set_external_irq(k->irq_no, EXT_BOTH_EDGES, GPIO_PULLUP_DIS);
        restore_flags(flags);                   /*储存中断状态 */
                                                /* 保存中断后的信息*/
        if (up) {
            key_value = k->butt_no + 0x80;
        }
        else{
            key_value = k->butt_no;
        }
        ready = 1;                              /*设置全局状态*/
          wake_up_interruptible(&buttons_wait); /*  唤醒进程 */
    }
    /*-----------------------------------*
     *    寄存器中断         *
     *-----------------------------------*/
    static int request_irqs (void)
    {
        struct butt_info *k;
        int i;
        for (i=0;i<=1;i++) {
            k = &butt_info_tab[i];
                                                /* 使用外部中断*/
```

```
            set_external_irq (k->irq_no, EXT_BOTH_EDGES, GPIO_PULLUP_DIS);
                                                    /*  申请中断资源   */
            if(request_irq(k->irq_no,&butt_isr_handler,SA_INTERRUPT,
BUTT_DEVICE_NAME, 0))
            {
                return -1;
            }
        }
        return 0;
    }
    /*-----------------------------------*
     *    释放中断         *
     *-----------------------------------*/
    static void free_irqs (void)
    {
        struct butt_info *k;
        int i;
        for (i=0;i<=1;i++) {
            k = &butt_info_tab[i];
            free_irq (k->irq_no, &butt_isr_handler);
        }
    }
    /*-------------文件系统---------------------------------*/
    /*-----------------------------------*
     *  文件系统 I/O   *
     *-----------------------------------*/
        static int buttons_read (struct file *file, char *buffer, size_t count,
loff_t *ppos)
        {
            static int key;
            int flags;
            int flag;
            int repeat;
            if (!ready) -EAGAIN;
            if (count != sizeof key_value) return -EINVAL;
            save_flags (flags);
            if (key != key_value) {
                key = key_value;
                repeat = 0;
            }
            else{
                repeat = 1;
            }
            restore_flags (flags);
            if (repeat) {
                return -EAGAIN;
            }
            /*  使用 copy_to_user 函数发送按钮键值到用户空间 */
            wait_event_interruptible(buttons_wait, flag != 0);
            copy_to_user (buffer, &key, sizeof key);
            ready = 0;
            flag = 0;
            return sizeof key_value;
        }
    /*-----------------------------------*
     *    文件系统 I/O 控制 *
```

```
     *---------------------------------*/
    static int buttons_ioctl(struct inode *inode, struct file *file, unsigned
int cmd, unsigned long arg)
    {
        switch (cmd) {
           default:
           return -EINVAL;
        }
    }
    /*  结构化操作   */
    static struct file_operations buttons_fops = {
                 owner:  THIS_MODULE,
                 ioctl:  buttons_ioctl,     /*  注册但是没有使用过 */
                 poll:   buttons_select,
                 read:   buttons_read,
    };
    /*---------------------------------系统入口------------------*/
    /*  初始化键值  */
    #ifdef CONFIG_DEVFS_FS
        static devfs_handle_t devfs_buttons_dir,devfs_handle;
    #endif
    static int __init buttons_init (void)
    {
        int ret;
        ready = 0;
        init_waitqueue_head(&buttons_wait);
    #ifdef CONFIG_DEVFS_FS
        /*  向文件系统注册设备 */
         ret = devfs_register_chrdev(0, BUTT_DEVICE_NAME, &buttons_fops);
         if(ret < 0) {
             printk(BUTT_DEVICE_NAME "Unable to get major for button device!\n");
             return ret;
         }
         /*建立设备 */
         devfs_buttons_dir = devfs_mk_dir(NULL, BUTT_DEVICE_NAME, NULL);
        devfs_handle = devfs_register (devfs_buttons_dir, BUTT_DEVICE_NAME,
                    DEVFS_FL_DEFAULT, BUTTON_MAJOR, 0, S_IFCHR | S_IRUSR |
S_IWUSR, &buttons_fops, NULL);
    #else
        /*  注册按钮设备  */
        ret = register_chrdev (BUTTON_MAJOR, BUTT_DEVICE_NAME, &buttons_fops);
        if (ret <0) {
           printk (BUTT_DEVICE_NAME "Can't register major number!\n");
           return ret;
        }
    #endif
        /*  申请中断资源 */
        ret = request_irqs ();
        if (ret) {
           unregister_chrdev (BUTTON_MAJOR, BUTT_DEVICE_NAME);
           printk (BUTT_DEVICE_NAME "Can't request irqs!\n");
           return ret;
        }
        return 0;
    }
    /*  注销按钮 */
```

```
static void __exit buttons_exit (void)
{
   free_irqs();
   unregister_chrdev (BUTTON_MAJOR, BUTT_DEVICE_NAME);
#ifdef CONFIG_DEVFS_FS
    devfs_unregister(devfs_handle);
    devfs_unregister(devfs_buttons_dir);
#endif
   printk ("Release buttons.\n");
}
module_init (buttons_init);
module_exit (buttons_exit);
MODULE_DESCRIPTION("EduKitII-2410 button driver");
MODULE_AUTHOR("lewis");
```

5. I2C 驱动

限于篇幅这里省略，请读者自行参考光盘中代码。

18.6　本章总结

本章介绍了家庭安全监控系统的设计实现过程，整个家庭安全服务系统包括服务器子系统、传感器子系统和客户端子系统。读者在学习过程中，需要注意以下几点。

（1）硬件的选型很重要，有些硬件平台本身带有很多支持，有利于用户开发。

（2）嵌入式系统在移植时，需要进行一些裁剪的工作。

（3）程序的健壮性和鲁棒性在嵌入式系统中非常重要，因为嵌入式系统往往拥有的硬件资源有限，会出现很多意外情况。

第 19 章

移动校园系统设计实例

如今手持设备（如手机、PDA 等）的处理能力日益强大，可以给人们的生活带来更多便捷。本章将介绍移动校园系统设计过程，本系统旨在让手持 NOKIA N800 的同学们通过这个系统方便地获取自己想要的诸如课表、课件、公告、位置、论坛主题等校园信息。用户并不一定要在校园内才可使用这个系统提供的服务，只要是有 Wi-Fi 信号覆盖的地区都可以使用。

19.1 应用环境与硬件设计概要

本系统中客户端系统采用 NOKIA N800 PDA。 服务器端采用 MySQL 数据库。

19.1.1 系统功能和组成

从整体上讲，此项目是 C/S 架构的，NOKIA N800 是其中的客户端。在服务器端则有一个 MySQL 数据库以及访问该数据库的 API 层。数据库中存储有提供服务所需的大量信息，客户端通过 API 层来与数据库进行交互。图 19-1 的系统模型展示了整个项目的架构。

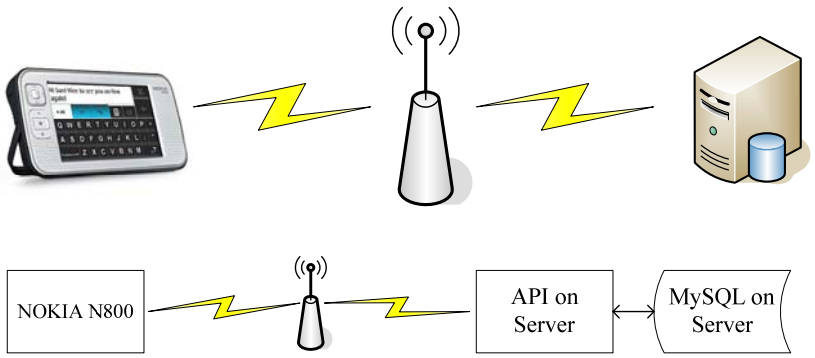


图 19-1 系统模型

19.1.2 系统模块功能和软件图

系统模块功能描述如表 19-1 所示。

图 19-2 和图 19-3 分别为软件的顶级用例图和总体用例图。

表 19-1　系统模块功能描述说明

功能类别	子 功 能
A 课表信息服务	A1．查询一门课的具体信息（课程内容、授课教师、上课时间及地点）
	A2．获得用户所感兴趣的课程的列表
B 教室信息服务	B1．查询一个教室的具体信息（位置、教室参数描述、教室被占用的情况）
	B2．获得用户所感兴趣的教室的列表
C BBS 功能	C1．登录
	C2．浏览主题
	C3．发表主题
	C4．回复主题
D 校园位置信息服务	D1．获取地点在地图上的位置
	D2．缩放地图
	D3．在地图上测距
E 即时消息服务	E1．获知即时的校园消息、新闻、公告
	E1．获知系统升级信息，并可让系统自动升级
	E3．查看所有历史的系统发来的消息
F 流媒体播放	F1．获得可观看的流媒体列表
	F2．观看流媒体

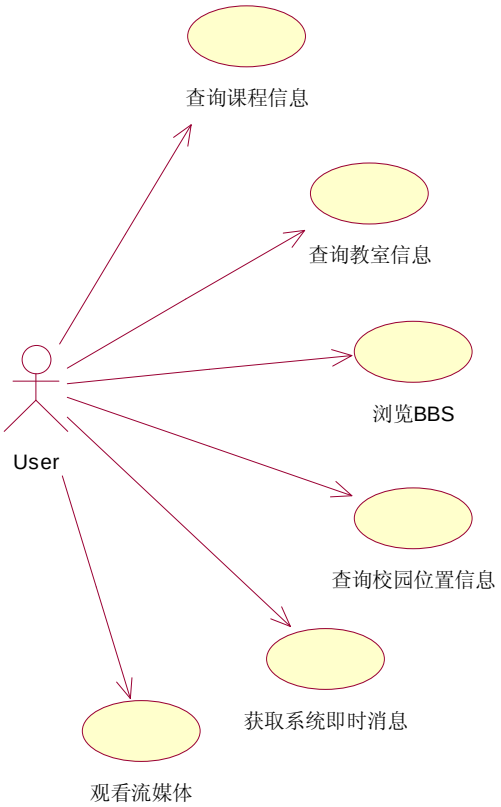


图 19-2　软件的顶级用例图

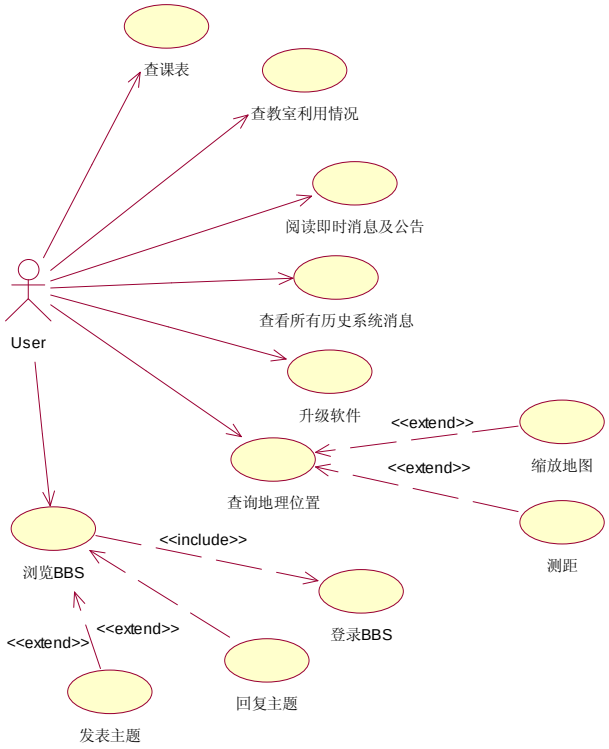


图 19-3　软件的总体用例图

19.2　系统硬件结构

本系统需要的硬件设施如下。

1．处理器：TI OMAP2420 320MHz。

2．内存：128MB RAM + 256MB ROM + 128MB 存储卡。

3．操作系统：meamo(Linux), 2007 Tablet Edition。

4．输出设备

- 显示屏：TFT 彩屏，800×480 像素，4.1 英寸，触摸屏；
- 立体声耳机；
- 小音箱。

5．输入设备

- 触摸屏、手写笔一根；
- Microphone；
- 30 万像素摄像头。

6．数据通信设备

- USB 数据线；
- WLAN：支持 802.11b/g；
- 蓝牙：支持蓝牙 v2.0。

7．视频播放：支持 MPEG1、MPEG4、Real Video、H.263、AVI、3GP 等视频格式。

19.3　系统软件结构

19.3.1　软件整体结构

如图 19-4 为软件的整体功能模块图，其中最主要的功能模块有以下 4 个。

- Syllabus，完成课表查询相关功能；
- BBS，完成浏览论坛相关功能；
- Map，完成校园电子地图相关功能；
- Message，完成软件更新及校园消息获取相关功能。

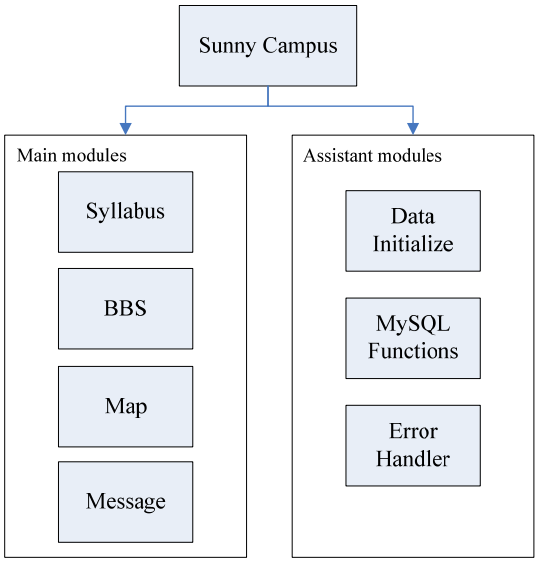


图 19-4　软件功能模块图

另外软件还有 3 个辅助模块，分别为：

- Data Initialize，完成读取配置文件，对程序的初始化参数进行设置；
- MySQL Functions，完成有关数据库的操作；
- Error Handler，完成对程序运行中错误和异常的处理。

图 19-5 是软件界面的一张示意图，是用 VISIO 画的 XP 风格的界面图。此图主要为展示软件界面的整体结构，而不在乎其风格。

整体界面中以每个标签页来呈现每一个主要功能模块的界面，其中的“首页”标签页将会显示一些欢迎信息和统计信息。Syllabus 模块的子界面，其左侧为用户提供查询条件设定的控件，右上方为查询后所得的列表，右下方则为查询列表中当前项的详细信息。

在实际实现过程中，将采用 GTK+工具库来实现界面与事件响应。

图 19-6 为程序运行的大体流程，主要的步骤是初始化程序数据、创建界面，然后以事件响应的方式与用户交互，最后释放内存并退出。相应地，软件中的那 4 个主要的功能模块均包含初始化与销毁数据结构子模块、创建界面子模块、事件响应子模块。每个功能模块的主要功能实现都集中在事件响应子模块当中。

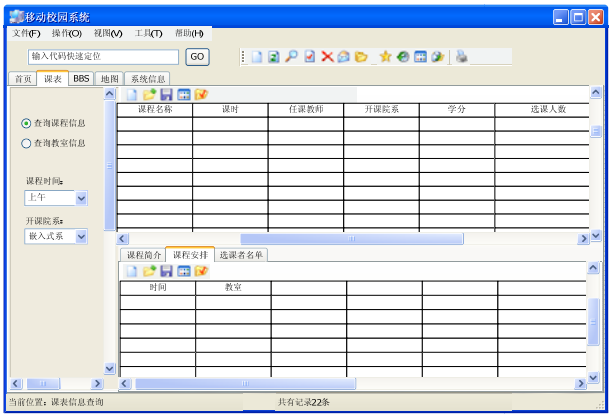


图 19-5　软件的整体界面设计图

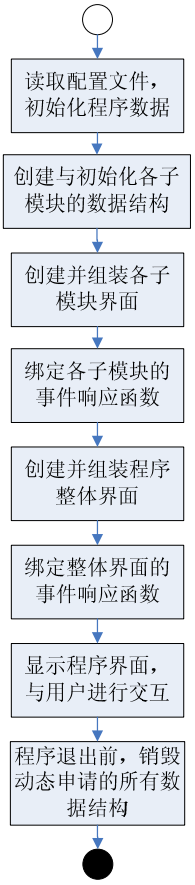


图 19-6　程序运行流程图

19.3.2　软件模块结构

本小节分别介绍软件中的 4 个功能模块。图 19-7 为 Syllabus 模块的内部结构图。

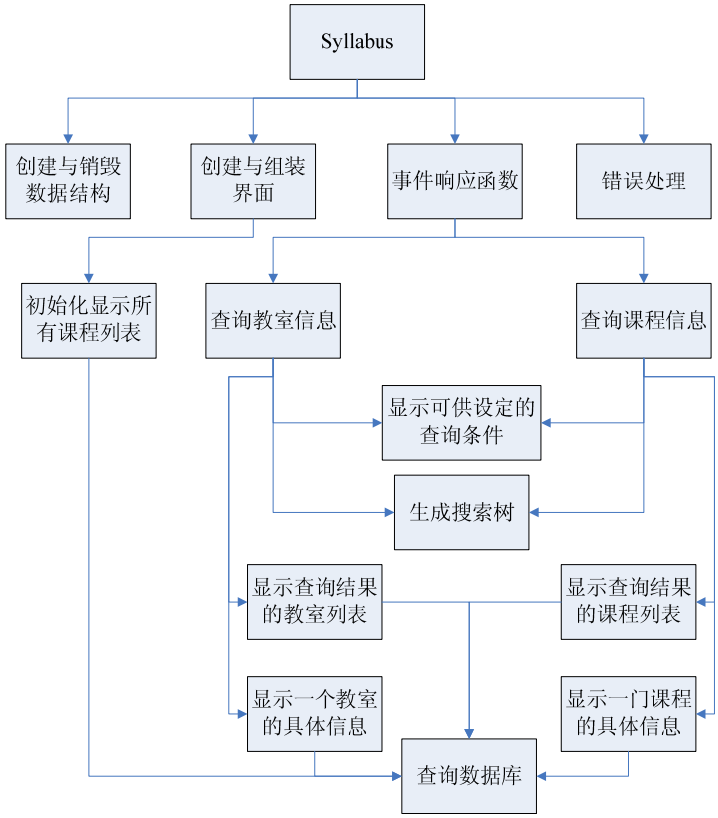


图 19-7　Syllabus 模块的内部结构图

图 19-8 为 BBS 模块的内部结构图。

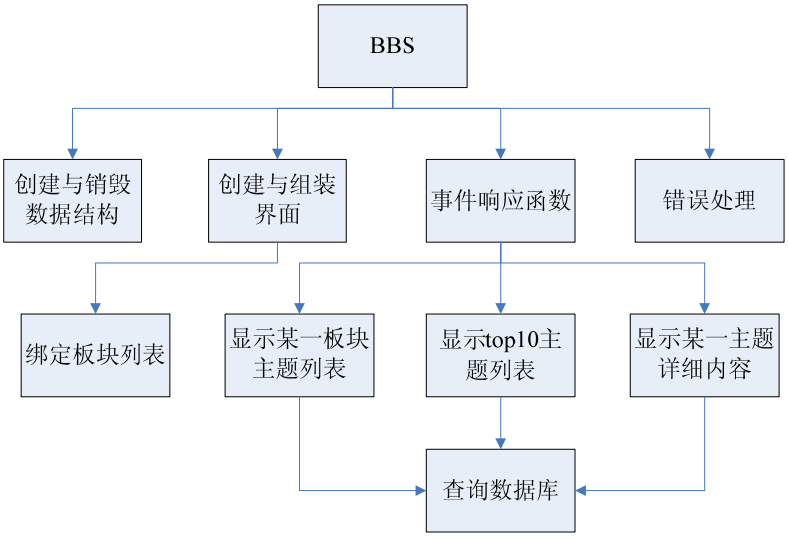


图 19-8　BBS 模块的内部结构图

图 19-9 为 Map 模块的内部结构图。

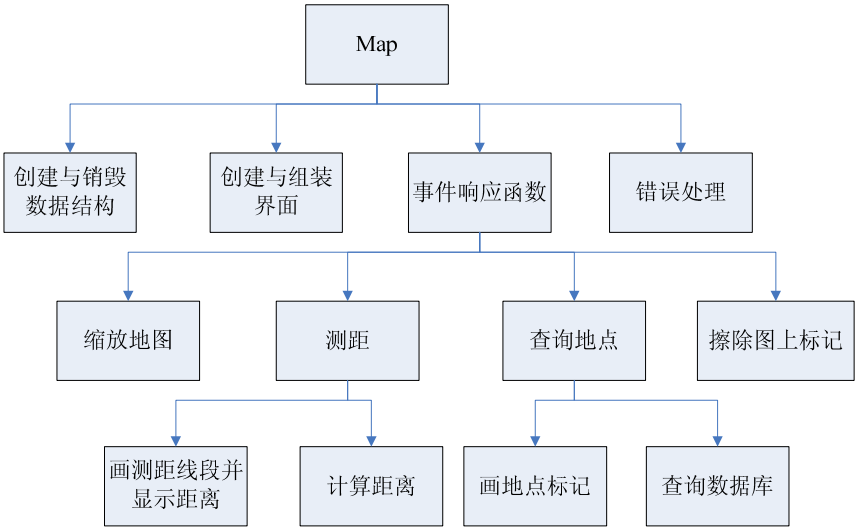


图 19-9　Map 模块的内部结构图

图 19-10 为 Message 模块的内部结构图。

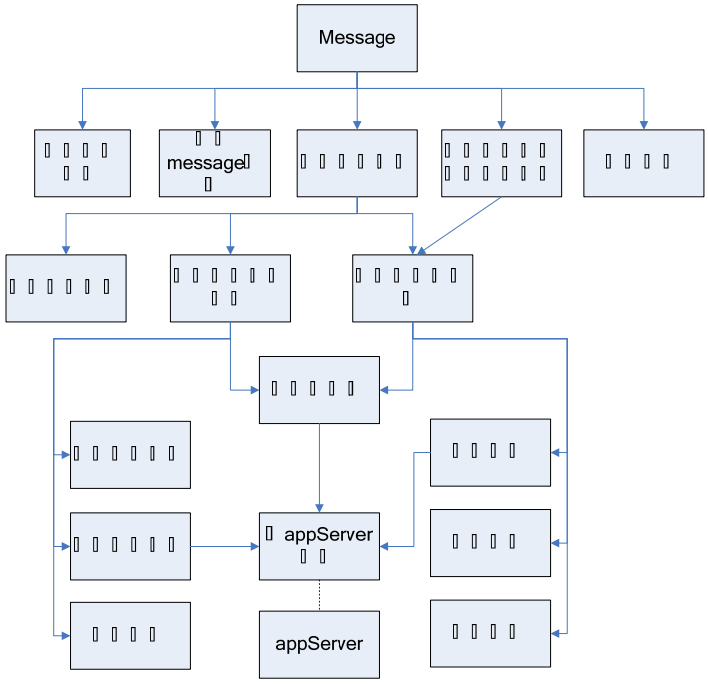


图 19-10　Message 模块的内部结构图

图 19-11 为启动程序时检测程序更新信息流程图。

图 19-12 为周期性检测新校园消息流程图。

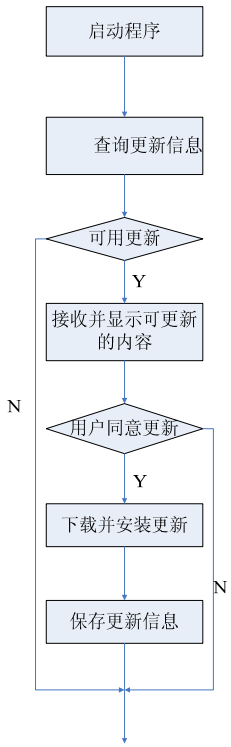


图 19-11　启动程序时检测程序更新信息流程图

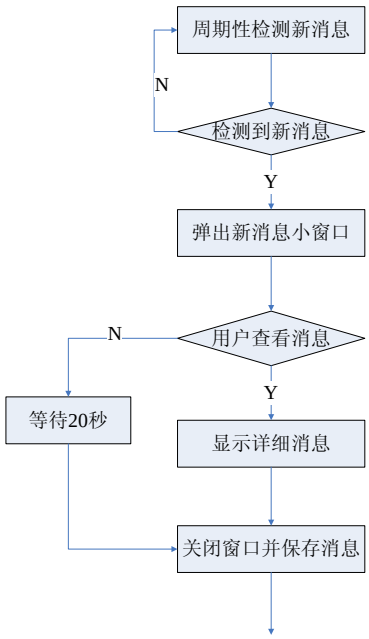


图 19-12　周期性检测新校园消息流程图

19.3.3　接口设计

1．用户接口

（1）用户通过操作界面上的控件，可以实现相应的功能。

（2）用户可以修改/usr/SunnyCampus 文件夹中的 config 文件来实现对软件的配置，其格式如下：

```
server_ip:192.168.*.*;
username:root;
password:123;
database:SunnyCampus;
message_log_path:/usr/sunnycampus/msg.log;
message_logo_path:/usr/sunnycampus/logo.jpg;
map_file_path:/usr/sunnycampus/map.jpg;
check_message_signal_interval:180;
```

其中：

server_ip 为服务器 IP；

username 为服务器上 MySQL 数据库登录名；

password 为服务器上 MySQL 数据库登录密码；

database 为软件在服务器上 MySQL 数据库中的数据库名；

message_log_path 为历史消息记录文件的路径；

message_logo_path 为 Message 标签页中的 logo 图片文件的路径；

map_file_path 为 Map 标签页中所用的地图图片文件的路径；

check_message_signal_interval 为周期性检测新校园消息的时间间隔。

2．外部接口

本软件需要有 mysqlclient 动态链接库，它应该在 PDA 上操作系统的/usr/lib 文件夹中。本软件要求 PDA 连入校园内的无线网络。在服务器端需要有 MySQL 数据库储存软件所需要的信息。数据库中论坛数据库实际上是一个因特网上校园论坛的基础数据库，因特网上用户可更新论坛数据库中数据，本软件用户可浏览论坛数据库中数据。

3．内部接口

（1）初始化各个主功能模块数据结构的函数分别为：

- syllabus_module_new（）
- bbs_module_new（）
- map_module_new（）
- message_module_new（）

（2）销毁各个主功能模块数据结构的函数分别为：

- syllabus_module_destroy（）
- bbs_module_destroy（）
- map_module_destroy（）
- message_module_destroy（）

（3）所有主功能模块将使用同一个“数据库连接结构”——MYSQL* CONN。

（4）每个主功能模块在创建好自己的整体界面后，将自己界面的最外层容器填充到主界面的相应标签页中，并将这个容器指针分别赋值给 app->homePageUI、app->syllabusPageUI、app->bbsPageUI、app->mapPageUI、app->messagePageUI，其中 app 是结构 App 的实例，是代表程序整体的数据结构。

（5）创建各个主功能模块界面的函数分别为：

- buildHomeUI ( )
- buildSyllabusUI ( )
- buildBBSUI ( )
- buildMapUI ( )
- buildMessageUI ( )

19.3.4 运行过程设计

程序的过程设计包括 Data Initialize 模块、各主功能模块中的数据结构初始化子模块、各主功能模块中的创建与组装界面子模块、MySQL Functions 模块、Message 模块中的检测软件更新信息子模块。

模块及其对应的功能详细说明如下。

查询课程信息：Syllabus 模块，MySQL Functions 模块；

查询教室信息：Syllabus 模块，MySQL Functions 模块；

浏览 BBS 中某一板块列表：BBS 模块，MySQL Functions 模块；

阅读 BBS 中某一主题：BBS 模块，MySQL Functions 模块；

获取 BBS 中 Top 10 列表：BBS 模块，MySQL Functions 模块；

缩放地图：Map 模块中的缩放地图子模块；

测距：Map 模块中的测距子模块；

查询地点：Map 模块中的地点查询子模块，MySQL Functions 模块；

检测软件更新：Message 模块中的检测软件更新信息子模块；

实施软件更新：Message 模块中的下载更新文件与实施自动更新子模块；

阅读历史消息：Message 模块中的查看历史消息子模块；

检测新校园消息：Message 模块中的检测新校园消息子模块；

周期性自动检测新校园消息：Message 模块中的“创建周期性自动检测新校园消息”子模块；

异常捕获与错误处理：Error Handler 模块；

正常退出程序的过程：各主功能模块中的销毁数据结构子模块。

19.3.5　系统数据结构设计

（1）整个程序中的全局变量（由 Data Initialize 模块进行初始化，所有模块均可访问）：

```
MYSQL* CONN;//数据库连接
char CONFIG_FILE_PATH[50];
char HOSTIP[16];
char USER[16];
char PWD[10];
char DB[20];
char MAP_FILE_PATH[50];
char MSG_LOG_PATH[50];
char MSG_LOGO_PATH[50];
int CHECK_MSG_SIGNAL_INTERVAL;//周期性自动检测新校园消息的时间间隔
```

（2）代表程序整体的结构 App：

```
typedef struct {
HildonProgram* program;
HildonWindow* window;
GtkWidget* notebook;//标签页整体框架
GtkWidget* homePageUI;
GtkWidget* syllubusPageUI;
GtkWidget* mapPageUI;
GtkWidget* bbsPageUI;
GtkWidget* messagesPageUI;
BBSModule * bbsModule;
MessageModule* messageModule;
MapModule * mapModule;
SyllabusModule* syllabusModule;
gboolean fullScreen;//是否为全屏显示状态
}App
```

（3）Map 模块中代表一个地点查询匹配项的标记的结构 Pot：

```
typedef struct{
    char name[20];
    int x;
```

```
    int y;
}Pot;
```

（4）Map 模块中代表一个测距点的结构 Node：

```
typedef struct{
    int x;
    int y;
}Node;
```

（5）Map 模块将用有限长度的 Pot 数组记录地点查询匹配项，用有限长度的 Node 数组记录所有当前的测距点。

19.3.6 搭建开发环境

需要采用的开发环境如下。

- 操作系统：Ubuntu 操作系统
- 交叉编译环境：scratchbox
- 开发工具：Meamo SDK
- 编译程序：gcc
- PDA 上的操作系统：maemo NOKIA OS 2007

（1）开发环境的搭建建议在 Ubuntu 上进行，Ubuntu 就是从 debian 衍生来的 Linux 操作系统，Maemo 组织推荐使用。

（2）将 Ubuntu 系统进行彻底的更新，如下所示。

```
#apt-get update;
#apt-get install;
```

也可以通过 Ubuntu 操作系统中的包管理器来对系统进行更新。

（3）搭建 maemo 开发平台。

这一过程请参阅“Development Platform install.txt”文件的内容完成。此文件的来源是 http://tablets-dev.nokia.com/3.1/INSTALL.txt。

（4）测试是否搭建成功。

在终端中输入命令：

```
$ Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 –ac
```

打开另一终端，输入命令：

```
$ scratchbox
[sbox-SDK_X86:~] > export DISPLAY=:2
[sbox-SDK_X86:~] > af-sb-init.sh start
```

若搭建成功，这时应该出现如下图形界面。

下面是有助于开发的参考网站。

- http://maemo.org/ ：指导如何进行 maemo 开发，是最为重要、针对性最强的开发指导网站。
- http://library.gnome.org/devel/gtk/unstable/：GTK+参考手册。



Sunny Campus 软件实现过程的大体流程如下。

（1）在 Scratchbox 中的 SDK_ARMEL 平台下（SDK_X86 平台也可以）用文本编辑器编写程序并运行、调试和修改。

（2）本软件在开发过程中需要在 Scratchbox 中的 SDK_ARMEL 平台中安装 MySQL 客户端（实际上，如果有现成的通过交叉编译 MySQL 源代码得到的 MySQL 的静态库和动态库，就无须安装）。安装过程请参阅 http://guymage.net/index.php/2006/04/25/45-nokia-770-mysql-remote-connection-with-libmysqlclient-for-arm，若想获取现成的 MySQL 的静态库和动态库，可以去 http://guymage.net/index.php/2006/06/15/49-mysql-libraries-for-nokia-770-it-2006-and-maemo-20 页面下载。

（3）当然，本软件在开发过程中要在服务器端安装包含 Server 的 MySQL 数据库。

（4）当程序调试成功后，请参照 http://maemo.org/中所讲的 debian 安装包打包过程，将程序制作成安装包，这一过程最终可以得到一个.deb 文件，即为软件安装文件。这一过程一定要在 Scratchbox 中的 SDK_ARMEL 平台上进行，因为只有在这一平台制作生成的.deb 安装文件才可以在 NOKIA N800 上安装运行。

（5）将生成的.deb 文件复制到 NOKIA N800 中，双击安装即可。程序在虚拟平台上的运行效果与在PDA上实际运行并不完全相同，所以在虚拟平台上调试成功并不说明在PDA上就会顺利运行，还需要在 PDA 上运行、观察、测试，当发现问题后需在 Scratchbox 中修改程序，再打包并安装在 PDA 上，这样反复多次直至程序可以在 PDA 上顺利运行。

（6）进行 NOKIA N800 上的软件开发需要破解此 PDA 上用户权限，这样才能够有足够的权限将软件所需的动态链接库放到 PDA 上适当的文件夹中，并且一些其他必要的操作也需要有足够的权限。获取 PDA 上足够权限的方法有以下两种（本小节采用的是第一种方法）。

- 去 maemo.org 网站上下载 becomeroot.deb 安装包并在 PDA 上安装。打开 PDA 上的终端 X-Term（如果 PDA 上没有安装 X-Term，则须先安装），输入命令#sudo gainroot，执行这条命令后即可获得 root 的 shell 权限。
- 使 PDA 进入开发模式，这一过程请参照 http://maemo.org/community/wiki/HowDoiBecomeRoot/ 中的叙述，实际上这里介绍了几乎所有的获取 PDA 上 root 权限的方法。

19.4 系统模块程序代码

19.4.1 主函数

```
/*主函数*/
int main(int argc, char** argv) {
   /* 初始化 DTK+库. */
  gtk_init(&argc, &argv);
 /*初始化程序*/
    App *app;
    app = create_data ();
    App_Initialize(app);
  /* 开启主界面. */
  g_print("main: calling gtk_main\n");
  gtk_main();
  /* 标准输出和退出显示 */
  g_print("main: returned from gtk_main and exiting with success\n");
destroy_data ( app );
  return EXIT_SUCCESS;
}
```

19.4.2 Syllabus 课表模块

（1）可以查询各院系的各门课程的详细信息。

（2）可以查询各教室在各时段的利用情况。

```
//查询课程信息
char* get_course_union_general_query(SyllabusView* view)
{
    if(view==NULL)
        return "";
    char ret_value[512]="";
    gchar* gcourseAcamedy=NULL;
    gchar* gcourseScore=NULL;
    gchar* entry_text;
    char result[64]="";
    gcourseAcamedy=gtk_combo_box_get_active_text(GTK_COMBO_BOX(view->co
urseAcamedy));
    sprintf(result,"and TS_Academy.s_AcademyName='%s' ",gcourseAcamedy);
    if(gcourseAcamedy!=NULL)
        g_free(gcourseAcamedy);
    strcat(ret_value,result);
    gcourseScore=gtk_combo_box_get_active_text(GTK_COMBO_BOX(view->cour
seScore));
    sprintf(result,"and TS_Course.n_Score='%s' ",gcourseScore);
    if(gcourseScore!=NULL)
        g_free(gcourseScore);
    strcat(ret_value,result);
    entry_text=gtk_entry_get_text(GTK_ENTRY(view->courseName));
    printf("courseName=%s\n",entry_text);
    strcat(ret_value,"and TS_Course.s_CourseName like '%");
    strcat(ret_value,entry_text);
    strcat(ret_value,"%' ");
```

```
        entry_text=gtk_entry_get_text(GTK_ENTRY(view->teacherName));
        printf("teacherName=%s\n",entry_text);
        strcat(ret_value,"and TS_Teacher.s_TeacherName like '%");
        strcat(ret_value,entry_text);
        strcat(ret_value,"%' ");
        printf("get_course_union_general_query=%s\n",ret_value);
        return (char*)&ret_value;
    }
    //查询教室信息
    char* get_classroom_union_general_query(SyllabusView* view)
    {
        if(view==NULL)
            return "";
        gchar* gtemp;
        gchar* entry_text;
        char result[64]="";
        char ret_value[512]="";
        gtemp=gtk_combo_box_get_active_text(GTK_COMBO_BOX(view->classroomTy
pe));
        sprintf(result,"and TS_ClassroomType.s_ClassroomTypeDesc='%s' ",gtemp);
        strcat(ret_value,result);
        gtemp=gtk_combo_box_get_active_text(GTK_COMBO_BOX(view->haveProject
or));
        sprintf(result,"and TS_Classroom.e_HaveProjector='%s' ",gtemp);
        strcat(ret_value,result);
        if(gtemp!=NULL)
            g_free(gtemp);
        gtemp=gtk_combo_box_get_active_text(GTK_COMBO_BOX(view->containerCo
ndition));
        sprintf(result,"and TS_Classroom.n_Container%s",gtemp);
        strcat(ret_value,result);
        entry_text=gtk_entry_get_text(GTK_ENTRY(view->container));
        sprintf(result,"'%s' ",entry_text);
        strcat(ret_value,result);
        entry_text=gtk_entry_get_text(GTK_ENTRY(view->classroomName));
        strcat(ret_value,"and TS_Classroom.s_ClassroomName like '%");
        strcat(ret_value,entry_text);
        strcat(ret_value,"%' ");
        printf("query=%s\n",ret_value);
        return (char*)&ret_value;
    }
    //创建查询树
    void create_query_tree(GtkWidget* widget,gpointer data)
    {
        g_print("enter create_query_tree\n");
        SyllabusView* view=NULL;
        view=(SyllabusView*)(data);
        g_assert(view != NULL);
        gbooleantimeQuery=gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON
(view->condition_time));
        if(timeQuery == TRUE )
        {
            QueryTree* week=init_queryTree();
            QueryTree* weekDesc=init_queryTree();
            QueryTree* section=init_queryTree();

    fill_queryTree(week,view->week_from,view->week_condition,view->week_to);
```

```
        fill_queryTree(weekDesc,view->weekDesc_from,view->weekDesc_conditio
n,view->weekDesc_to);
        fill_queryTree(section,view->section_from,view->section_condition,v
iew->section_to);
            create_left_down_tree(view,week,weekDesc,section);
            gtk_widget_show_all(GTK_WIDGET((view->frame_time)));
            free_queryTree(week);
            free_queryTree(weekDesc);
            free_queryTree(section);
        }
        else
        {
            gboolean generalQuery=gtk_toggle_button_get_active (GTK_TOGGLE_
BUTTON (view->condition_general));
            if(generalQuery == TRUE )
            {
                /*请求*/
                char* end="";
                char query[2048]="";
                gbooleancourseSelected=gtk_toggle_button_get_active
(GTK_TOGGLE_BUTTON (view->btnCourse));
                gbooleanclassroomSelected=gtk_toggle_button_get_active
(GTK_TOGGLE_BUTTON (view->btnClassroom));
                char* union_general="";
                QUERY_TYPE queryType;
                if(courseSelected == TRUE )
                {
                    end=" group by TS_Course.n_CourseID";
                    union_general=get_course_union_general_query(view);
                    queryType=COURSE_QUERY;
                }
                if(classroomSelected == TRUE )
                {
                    end=" group by TS_Classroom.n_ClassroomID";
                    union_general=get_classroom_union_general_query(view);
                    queryType=CLASSROOM_QUERY;
                }
                strcat(query,union_general);
                strcat(query,end);
                printf("query=%s\n",query);
                fill_right_up_table(&query,view,"All:",queryType);
            }
        }
    }
```

19.4.3 BBS 论坛模块

（1）可以浏览各个版块各个主题帖。

（2）可以浏览本日“十大热门主题”。

```
    void onBoardChanged (GtkComboBox *combo_box, gpointer data)
    {
        if(!combo_box||!data)
            return;
        gchar* boardName = gtk_combo_box_get_active_text(combo_box);
        GtkListStore* topicListStore = (GtkListStore*)data;
```

```
        gtk_topic_list_store_fill(topicListStore, boardName);
        printf("-----------------------------------1\n");
        return;
    }
    void onDisplayTop10 (GtkWidget *widget, gpointer data)
    {
        if(!widget || !data)
            return;
        GtkListStore* top10ListStore = (GtkListStore*)data;
        gtk_topic_list_store_fill(top10ListStore, "TOP10");
        printf("-----------------------------------2\n");
        return;
    }
    gboolean onTopicSelected(GtkTreeView* treeview,
        gpointer userdata)
    {
        if(!treeview || !userdata)
            return;
        printf("Event Occur\n");
        GtkListStore* model = gtk_tree_view_get_model (treeview);
        GtkTreeIter iter;
        gtk_tree_selection_get_selected        (gtk_tree_view_get_selection
(treeview),
            &model,
            &iter);
        if (TRUE)
        {
            gchar *topic;
            gtk_tree_model_get(model, &iter, 1, &topic, -1);
            printf("%s\n",topic);
            GtkTextBuffer* contentTextBuffer = userdata;
            gtk_content_text_buffer_fill(contentTextBuffer, topic);
            g_free(topic);
        }
        return TRUE;
    }
    //bbs 排名
    GtkWidget* gtk_topic_list_view_new()
    {
        GtkWidget* topicListView;
        GtkTreeViewColumn *column;
        GtkCellRenderer *renderer;
        topicListView = gtk_tree_view_new();
        gtk_tree_view_set_headers_visible(GTK_TREE_VIEW(topicListView),TRUE
);
        gtk_tree_selection_set_mode(gtk_tree_view_get_selection
            (GTK_TREE_VIEW(topicListView)),
            GTK_SELECTION_SINGLE);
        renderer = gtk_cell_renderer_text_new ();
        g_object_set (G_OBJECT (renderer),"foreground", "blue",NULL);
        /* 添加列 0 */
        column=gtk_tree_view_column_new_with_attributes ("TopicID", renderer,
"text", COL_TOPIC_ID,NULL);
        gtk_tree_view_column_set_resizable(column,TRUE);
        gtk_tree_view_column_set_visible (column,FALSE);
        gtk_tree_view_append_column (GTK_TREE_VIEW (topicListView), column);
        /*添加列 1 */
```

```
        column=gtk_tree_view_column_new_with_attributes ("Topic", renderer,
"text", COL_TOPIC,NULL);
        gtk_tree_view_column_set_resizable(column,TRUE);
        gtk_tree_view_append_column (GTK_TREE_VIEW (topicListView), column);
        /*添加列 2 */
        column=gtk_tree_view_column_new_with_attributes ("Author", renderer,
"text", COL_AUTHOR,NULL);
        gtk_tree_view_column_set_resizable(column,TRUE);
        gtk_tree_view_append_column (GTK_TREE_VIEW (topicListView), column);
        /*添加列 3 */
        column=gtk_tree_view_column_new_with_attributes ("Post", renderer,
"text", COL_POST,NULL);
        gtk_tree_view_column_set_resizable(column,TRUE);
        gtk_tree_view_append_column (GTK_TREE_VIEW (topicListView), column);
        /*添加列 4*/
     column=gtk_tree_view_column_new_with_attributes("Date",renderer,"text",
COL_DATE,NULL);
        gtk_tree_view_column_set_resizable(column,TRUE);
        gtk_tree_view_append_column (GTK_TREE_VIEW (topicListView), column);
        gtk_widget_show (topicListView);
        return topicListView;
    }
```

19.4.4 Map 地图模块

（1）可以在一张校园地图上查询用户所关心的地点的位置。

（2）可以缩放地图，在地图上测距。

```
    //数据库操作
    static MYSQL_RES* ExcuteSQL(gchar* sqlstring)
    {
         int   query_error;
         MYSQL_RES *query_result;
         query_error=mysql_query(CONN, sqlstring);
         if(query_error!=0)
            {
           printf(mysql_error(CONN));
           return NULL;
         }
         query_result=mysql_store_result(CONN);
         return query_result;
    }
    //两点间画线测距
    static void draw_line_and_show_distance(MapModule *mapModule)
    {
        GdkPixmap *pmap;
         GdkGC *gc;
         GdkColormap   *colormap;
        GdkColor   color;
        int x1,y1,x2,y2;
        int distance;
        int i;
        char show_text[20];
    pmap=gdk_pixmap_new(NULL,mapModule->map_width*mapModule->scales[mapModu
le->scales_index],mapModule->map_height*mapModule->scales[mapModule->scales
_index],16);
```

```
         gdk_drawable_set_colormap (pmap,gdk_colormap_get_system());
         gdk_draw_pixbuf
(pmap,NULL,mapModule->map2,0,0,0,0,-1,-1,GDK_RGB_DITHER_NORMAL, 0, 0);
        color.red=0xffff;
        color.blue=0x0000;
        color.green=0x0000;
        gdk_colormap_alloc_color(gdk_colormap_get_system(),  &color,  FALSE,
TRUE);
         gc=gdk_gc_new(pmap);
         gdk_gc_set_foreground(gc,&color);
         while(mapModule->nodes_visible<mapModule->nodes_num)
         {
            x1=mapModule->nodes[mapModule->nodes_visible-1]->x;
            y1=mapModule->nodes[mapModule->nodes_visible-1]->y;
            x2=mapModule->nodes[mapModule->nodes_visible]->x;
            y2=mapModule->nodes[mapModule->nodes_visible]->y;
            printf("%d %d %d %d\n",x1,y1,x2,y2);
            distance=sqrt(pow(x2-x1,2)+pow(y2-y1,2));
        gdk_draw_line(pmap,gc,x1*mapModule->scales[mapModule->scales_index]
,y1*mapModule->scales[mapModule->scales_index],x2*mapModule->scales[mapModu
le->scales_index],y2*mapModule->scales[mapModule->scales_index]);
            mapModule->cur_distance+=distance*mapModule->scale_to_real;
            sprintf(show_text, "%dm", (int)mapModule->cur_distance);
            draw_text(mapModule->map_image,
        pmap,show_text,mapModule->nodes[mapModule->nodes_visible]->x*mapMod
ule->scales[mapModule->scales_index]+5,
        mapModule->nodes[mapModule->nodes_visible]->y*mapModule->scales[map
Module->scales_index]-5);
            mapModule->nodes_visible++;
        }
    /*  gdk_pixbuf_unref(mapModule->map2);*/
        gdk_pixbuf_get_from_drawable(mapModule->map2,pmap,gdk_drawable_get_
colormap(pmap), 0,0,0,0,-1, -1);
        gtk_image_set_from_pixbuf(mapModule->map_image,mapModule->map2);
        g_object_unref(pmap);
    }
    //图片更新
    static void paint_locations_when_image_update(MapModule *mapModule)
    {
        GdkPixmap *pmap;
         GdkGC *gc;
         GdkColormap   *colormap;
        GdkColor   color;
        int i;
    pmap=gdk_pixmap_new(NULL,mapModule->map_width*mapModule->scales[mapModu
le->scales_index],mapModule->map_height*mapModule->scales[mapModule->scales
_index],16);
         gdk_drawable_set_colormap (pmap,gdk_colormap_get_system());
         gdk_draw_pixbuf(pmap,NULL,mapModule->map2,0,0,0,0,-1,
-1,GDK_RGB_DITHER_NORMAL, 0, 0);
        color.red=0xffe6;
        color.blue=0x0000;
        color.green=0x0000;
        gdk_colormap_alloc_color(gdk_colormap_get_system(),  &color,  FALSE,
TRUE);
         gc=gdk_gc_new(pmap);
         gdk_gc_set_foreground(gc,&color);
```

```
        for(i=0;i<mapModule->pots_num;i++){
          gdk_draw_arc(pmap, gc, TRUE,
        mapModule->match_pots[i]->x*mapModule->scales[mapModule->scales_ind
ex]-8,
        mapModule->match_pots[i]->y*mapModule->scales[mapModule->scales_ind
ex]-8,
                             16, 16,  (0 * 64), (360 * 64));
        draw_text(mapModule->map_image, pmap,
                                mapModule->match_pots[i]->name,
        mapModule->match_pots[i]->x*mapModule->scales[mapModule->scales_ind
ex]+8,
        mapModule->match_pots[i]->y*mapModule->scales[mapModule->scales_ind
ex]-12);
        }
        gdk_pixbuf_get_from_drawable(mapModule->map2,pmap,gdk_drawable_get_
colormap(pmap), 0,0,0,0,-1, -1);
        gtk_image_set_from_pixbuf(mapModule->map_image,mapModule->map2);
        g_object_unref(pmap);
    }
```

19.4.5 Message 系统消息模块

（1）在客户端即时显示服务器即时传来的消息和公告。

（2）在用户打开软件时查看是否有更新并可以自动更新。

（3）用户可以浏览所有历史系统消息。

限于篇幅具体代码省略，请读者参考光盘中代码进行学习。

19.5 本章总结

本章介绍了移动校园系统的设计实现过程。通过本系统手持 NOKIA N800 的同学们可以获取自己想要的诸如课表、课件、公告、位置、论坛主题等校园信息。用户并不一定要在校园内才可使用这个系统提供的服务，只要是有 Wi-Fi 信号覆盖的地区都可使用。

本系统比较复杂，实现的模块功能比较多，同时设计时用到的硬件设施较多，用户需要结合几种开发环境来实现。读者学习完后，还需要考虑以下两点。

（1）程序的扩展性，例如校园不断会有新的活动、新的设施和安排，可以动态地进行增删。

（2）从程序的友好性来说，针对每个校园最好在界面上加入特色元素，例如校徽等。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview®
www.broadview.com.cn

《嵌入式 Linux 驱动程序和系统开发实例精讲》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．短信

您只需编写如下短信：B07936+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2．电子邮件

您可以发邮件至**jsj@phei.com.cn**或**editor@broadview.com.cn**。

3．信件

您可以写信至如下地址：北京万寿路**173**信箱博文视点，邮编：**100036**。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B07936+退订”至10666666789即可，或者编写邮件“B07936+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（**www.broadview.com.cn**）上，或推荐至**CSDN.NET**等专业网站上发表，被发表的书评的作者将获得价值**50**元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：　dbqq@phei.com.cn
通信地址：北京市万寿路 173 信箱
　　电子工业出版社总编办公室
邮　　编：100036