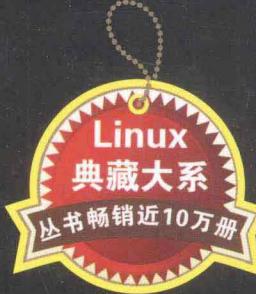
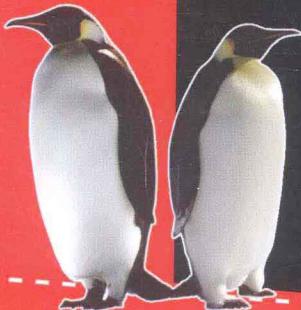


国内第一本Linux系统移植图书全面升级，ChinaUnix社区鼎力推荐  
通过15个典型案例，全面、系统、深入地剖析Linux系统移植的方法



# Linux

刘刚  
赵剑川 等编著

## 系统移植 (第2版)

赠送17.5小时高清语音教学视频及教学PPT

下载网址：<http://www.wanjuanchina.net>

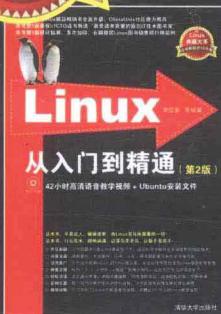
- **内容新颖：**书中的开发环境、编译工具和软件都是当前流行的稳定版本
- **内容全面：**涵盖Linux内核、文件系统、驱动及数据库等各种系统的移植
- **循序渐进：**遵循原理分析→代码分析→编译→测试→移植的学习顺序
- **讲解详细：**所有编译过程都附有编译命令，并对复杂的命令给出了详细说明
- **技巧性强：**穿插了大量的经验和技巧，并对容易出错的地方给出了专门的提示
- **注重实战：**通过典型案例，让读者深入体验Linux系统移植的方法和全过程



清华大学出版社

Linux  
典藏大系

精品畅销丛书，一线人员打造，有口皆碑，Linux爱好者的不二选择！



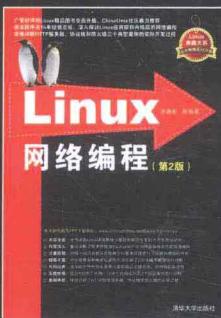
Linux

从入门到精通（第2版）

42小时高清语音教学视频+Ubuntu安装文件

本书将帮助读者全面掌握Linux系统的基本操作、配置与管理，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-31272-7

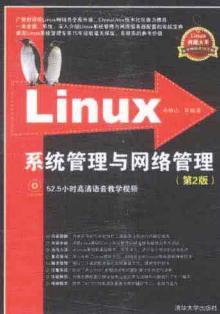


Linux

网络编程（第2版）

本书将帮助读者全面掌握Linux网络编程技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-33528-3



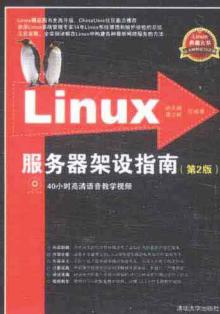
Linux

系统管理与网络管理（第2版）

62小时高清语音教学视频

本书将帮助读者全面掌握Linux系统管理与网络管理技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-32018-0



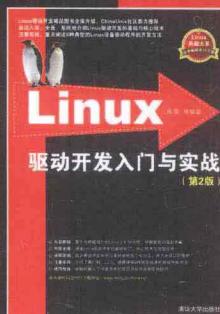
Linux

服务器架设指南（第2版）

40小时高清语音教学视频

本书将帮助读者全面掌握Linux服务器架设技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-31957-3



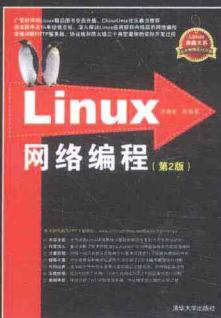
Linux

驱动开发入门与实战（第2版）

40小时高清语音教学视频

本书将帮助读者全面掌握Linux驱动开发技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-33776-8

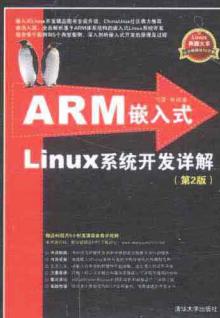


Linux

C程序设计（第2版）

399分钟高清语音教学视频

ISBN 978-7-302-34792-7

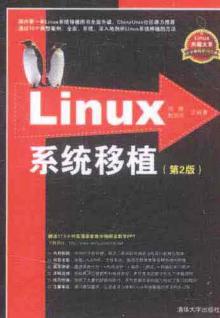


ARM嵌入式

Linux系统开发详解（第2版）

本书将帮助读者全面掌握ARM嵌入式Linux系统开发技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-34052-2

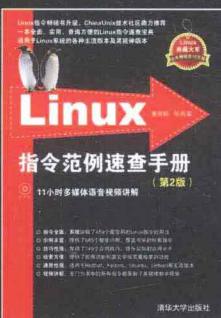


Linux

系统移植（第2版）

本书将帮助读者全面掌握Linux系统移植技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-34426-1



Linux

指令范例速查手册（第2版）

11小时多媒体语音视频讲解

本书将帮助读者全面掌握Linux命令行操作技巧，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-34425-4



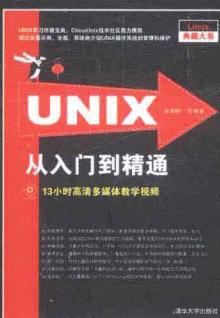
Shell

从入门到精通（第2版）

15小时配套教学视频+46小时Linux教学视频

本书将帮助读者全面掌握Shell脚本编写技术，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-33807-9



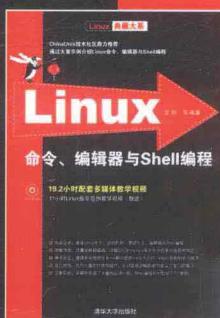
UNIX

从入门到精通（第2版）

13小时高清多媒体教学视频

本书将帮助读者全面掌握UNIX系统操作技巧，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-30735-8



Linux

命令、编辑器与Shell编程（第2版）

19.2小时配套多媒体教学视频

本书将帮助读者全面掌握Linux命令行操作技巧，深入学习各种应用软件的使用方法。通过大量的实践案例和丰富的经验分享，使读者能够熟练地运用Linux进行日常工作和项目开发。

ISBN 978-7-302-27615-9



清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

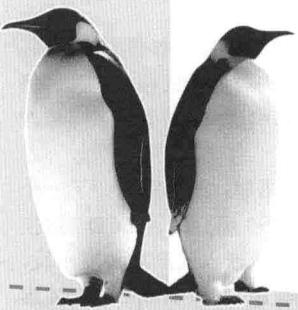
上架建议：计算机/Linux

ISBN 978-7-302-34426-1



9 787302 344261 >

定价：69.00元



典藏大系

# Linux

刘刚  
赵剑川 等编著

# 系统移植

(第2版)

清华大学出版社

北京

## 内 容 简 介

本书是获得了大量读者好评的“Linux 典藏大系”中的《Linux 系统移植》的第 2 版。本书由浅入深，全面、系统地介绍了 Linux 系统移植的各方面知识。书中的每个章节都有相应的实例编译或移植过程，每个移植实例都具有代表性，在实际应用和开发中有很高的参考价值。另外，本书赠送 17.5 小时多媒体教学视频及书中涉及的实例源程序，便于读者高效、直观地学习。

本书分为 4 篇。第 1 篇简单介绍了 Linux 内核和嵌入式 Linux 系统开发环境搭建；第 2 篇介绍了一个最基本的嵌入式系统的组成部分、Bootloader 移植、内核移植和文件系统移植；第 3 篇介绍了 LCD、触摸屏、USB、网卡、音频、SD 卡、NandFlash 等流行的设备驱动移植过程；第 4 篇从嵌入式产品角度出发，介绍了 GUI、Qtopia、嵌入式数据库 Berkeley DB 和 SQLite、嵌入式 Web 服务器 BOA 和 Thhttpd、JVM 虚拟机的移植及目前流行的 VoIP 技术和相关协议。

本书适合嵌入式 Linux 系统入门人员、Linux 系统开发和移植、系统分析师等相关人员阅读，也适合作为大中专院校相关专业的实验教材使用。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

### 图书在版编目 (CIP) 数据

Linux 系统移植 / 刘刚等编著. —2 版. —北京：清华大学出版社，2014

(Linux 典藏大系)

ISBN 978-7-302-34426-1

I. ①L… II. ①刘… III. ①Linux 操作系统 IV. ①TP316.89

中国版本图书馆 CIP 数据核字 (2013) 第 265921 号

责任编辑：夏兆彦

封面设计：欧振旭

责任校对：胡伟民

责任印制：沈 露

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：清华大学印刷厂

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185mm×260mm 印 张：31.5 字 数：790 千字

版 次：2011 年 1 月第 1 版 2014 年 2 月第 2 版 印 次：2014 年 2 月第 1 次印刷

印 数：5001～8000

定 价：69.00 元

# 前　　言

随着各种芯片技术的发展，各种嵌入式产品也如雨后春笋一般地出现了。目前，嵌入式产品应用领域涉及移动通信、汽车、医疗、家电等很多领域。而且，如今的嵌入式硬件的速度和容量越来越接近于 PC，因此在这些嵌入式产品上运行操作系统就成为了可能。一直以来，很多企业花费了巨大成本研发了大量运行在 PC 上的软件产品。如果将这些优秀的软件应用在嵌入式系统中，将会成为快速开发嵌入式系统，降低嵌入式产品开发成本，提高软件稳定性和安全性的重要途径。

目前，国内图书市场上专门介绍 Linux 系统移植的图书只有笔者 2011 年初出版的《Linux 系统移植》一书。该书也是获得了大量读者好评的“Linux 典藏大系”中的一个分册，出版后填补了该领域的空白。随着时间的推移和技术的发展，这本书已经逐渐不能适应读者的需求，因此笔者对这本书进行了升级和改版，便有了这本书的第 2 版。

本书在第 1 版的基础上进行了全新改版，升级了操作系统版本和 Linux 内核版本，并将编程环境和各种开发工具升级到了当前最新版本，也对第 1 版书中的一些疏漏进行了修订，对书中的一些实例和代码进行了重新表述，使得本书更加易读。相信升级后的图书易读性更强。

本书是笔者从事嵌入式开发的经验总结，希望能给目前从事嵌入式研发和学习的读者提供最有效的帮助，能使读者的嵌入式系统最快地运行起来，使读者在最短的时间内成功移植开源软件。

本书使用的源代码均为开源代码，读者可以从对应的官方网站获得。本书对于源码的重要部分进行了详细的分析，建议读者在阅读时对应源码进行阅读效果会更好。

## 关于“Linux 典藏大系”

“Linux 典藏大系”是清华大学出版社自 2010 年 1 月以来陆续推出的一个图书系列，截止 2012 年，已经出版了 10 余个品种。该系列图书涵盖了 Linux 技术的方方面面，可以满足各个层次和各个领域的读者学习 Linux 技术的需求。该系列图书自出版以来获得了广大读者的好评，已经成为了 Linux 图书市场上最耀眼的明星品牌之一。其销量在同类图书中也名列前茅，其中一些图书还获得了“51CTO 读书频道”颁发的“最受读者喜爱的原创 IT 技术图书奖”。该系列图书在出版过程中也得到了国内 Linux 领域最知名的技术社区 ChinaUnix（简称 CU）的大力支持和帮助，读者在 CU 社区中就图书的内容与活跃在 CU 社区中的 Linux 技术爱好者进行广泛交流，取得了良好的学习效果。

## 关于本书第 2 版

本书第 1 版出版后深受读者好评，并被 ChinaUNIX 技术社区所推荐。但是随着技术的发展，本书第 1 版内容已经无法满足读者的学习需求。应广大读者的要求，我们结合 Linux 系统移植的最新技术推出了本书的第 2 版。相比第 1 版，第 2 版图书在内容上的变化主要体现在以下几个方面：

- (1) 操作系统版本由 Fedora 6 升级为 Fedora 19。
- (2) 系统移植内核版本统一升级为 2.6.32。
- (3) 编译工具 GCC 版本升级到 4.4.3。
- (4) MiniGUI、Qt、Berkeley DB、SQLite、Linphone 等均升级为最新版本。
- (5) 将一些表达不准确的地方表述得更加准确。

## 本书特色

### 1. 内容全面、选材具有特点

本书介绍了最小系统的引导程序移植、内核移植、文件系统移植、各种驱动移植等内容。另外，本书还专门介绍了嵌入式数据库、嵌入式 GUI、嵌入式 Web 服务器、嵌入式 JVM、VoIP 技术等内容。对于数据库、GUI、Web 服务器分别选择了两种进行介绍，读者可以从性能上进行对比，然后应用在自己的项目中。

### 2. 内容由浅入深、循序渐进，可操作性强

本书按照由浅入深、循序渐进的梯度安排内容，适合各个层次的读者阅读。书中每章内容都遵循原理分析代码、分析编译测试移植的学习顺序，具有较强的可操作性。

### 3. 编译过程详细

本书的编译过程都附有详细的编译命令，对于复杂的命令均给出了说明，方便读者实际操作。读者可以边阅读本书，边动手进行实验。

### 4. 贯穿了大量的编译技巧，可迅速提升移植水平

本书在讲解编译过程时贯穿了大量的编译技巧，并针对移植过程中的编译错误介绍了如何发现错误的源头，同时给出了解决方法。这有利于读者解决类似的编译问题，提升系统移植的水平。

### 5. 提供多媒体教学视频和源文件

本书赠送 17.5 小时多媒体教学视频和实例源文件，便于读者高效、直观地学习。这些学习资料需要读者按照封底的提示自行下载。

## 本书内容概述

本书注重实践，包含了丰富的移植实例，这些实例各具特点，从基础的系统组成到设备驱动，再到高级应用，适合各个层面的读者学习和研究。本书中的实例是笔者根据实际项目中嵌入式产品的功能需求，专门选择的具有代表性的开源软件进行移植，包含了常见的嵌入式产品的最小系统组成部分移植，同时选择了应用比较多的数据库、Web 服务器、GUI 等进行移植。笔者通过亲自体会，在每次编译和移植过程中详细说明移植的细节，对移植过程中遇到的问题也给出了解决方法。本书最后还介绍了 VoIP 技术，并结合源码分析了 VoIP 的实现，同时还介绍了 VoIP 的详细编译过程。本书共 20 章，分为 4 篇。简单介绍如下。

### 第 1 篇 系统移植基础篇（第 1~2 章）

本篇介绍了系统移植的基础。首先对 Linux 内核进行了简单介绍，然后介绍了系统移植环境的搭建。通过对本篇内容的学习，读者可以对 Linux 系统有初步认识，能掌握嵌入式 Linux 开发工具的使用，能正确搭建开发平台，能够制作编译好的嵌入式系统。

### 第 2 篇 系统移植技术篇（第 3~5 章）

本篇介绍了一个最基本的嵌入式系统的组成部分、Bootloader、内核和文件系统的移植。学习完本篇内容后，读者能够动手独立编译和移植一个基本的嵌入式系统。

### 第 3 篇 系统移植驱动篇（第 6~12 章）

本篇介绍了各种驱动的移植，包括 LCD、触摸屏、USB、网卡、音频、SD 卡、NandFlash 等流行的设备驱动的移植过程。通过对本篇内容的学习，读者对嵌入式 Linux 驱动移植将会有一定的认识，可以基本掌握驱动的移植步骤，能完成简单的驱动移植。

### 第 4 篇 系统移植高级篇（第 13~20 章）

本篇从嵌入式产品的角度出发，介绍了系统移植中各种类型的高层软件移植，包括 GUI、数据库、Web 服务器、虚拟机的移植，最后还介绍了目前流行的 VoIP 技术，并结合源码介绍了 VoIP 相关协议和编译方法。通过学习本篇内容，读者可以掌握很多移植技巧，能够将这些实例应用到自己的项目中。

## 本书读者对象

- 嵌入式移植人员；
- 嵌入式专业的学生；
- 嵌入式实验指导老师；
- 嵌入式培训学员和老师；
- 系统分析师；
- 项目研发人员。

## 本书作者

本书由刘刚和赵剑川主笔编写，其他参与编写的人员有崔久、韩峰、胡国庆、刘智慧、

张照、梁洋洋、陈超、陈嵩、崔金英、樊丽、高宏静、高塬浚、郭金尚、郭维松、胡春杰、  
黄克、冀小幸、贾占领、阚言芳、兰海珍、李高畅、李海慧。

阅读本书时有任何疑问，都可通过以下方式联系我们，我们会及时答复。

E-mail: book@wanjuanchina.net (技术服务)

bookservice2008@163.com (编辑)

论坛网址: <http://www.wanjuanchina.net>

QQ 群: 336212690

编者

# 目 录

## 第 1 篇 系统移植基础篇

<b>第 1 章 Linux 内核介绍 .....</b>	<b>2</b>
1.1 系统调用接口 .....	2
1.1.1 Linux 系统调用 .....	2
1.1.2 用户编程接口 .....	2
1.1.3 系统调用与服务例程的对应关系 .....	3
1.1.4 系统调用过程 .....	3
1.1.5 系统调用传递的参数 .....	4
1.2 进程管理 .....	4
1.2.1 进程 .....	4
1.2.2 进程描述符 .....	5
1.2.3 进程状态 .....	6
1.2.4 进程调度 .....	6
1.2.5 进程地址空间 .....	8
1.3 内存管理 .....	10
1.3.1 内存管理技术 .....	10
1.3.2 内存区管理 .....	12
1.3.3 内核中获取内存的几种方式 .....	13
1.4 虚拟文件系统 .....	14
1.4.1 虚拟文件系统作用 .....	14
1.4.2 文件系统的注册 .....	15
1.4.3 文件系统的安装和卸载 .....	16
1.5 设备驱动程序 .....	17
1.5.1 字符设备驱动程序 .....	17
1.5.2 块设备驱动程序 .....	18
1.5.3 网络设备驱动程序 .....	21
1.5.4 内存与 I/O 操作 .....	22
1.6 小结 .....	23
<b>第 2 章 嵌入式 Linux 开发环境搭建 .....</b>	<b>24</b>
2.1 虚拟机及 Linux 安装 .....	24
2.1.1 虚拟机的安装 .....	24
2.1.2 虚拟机和主机通信设置 .....	31
2.1.3 虚拟机与主机共享文件 .....	35

2.1.4	虚拟机与主机文件传输	36
2.2	交叉编译工具	38
2.2.1	交叉编译工具安装	38
2.2.2	交叉编译器测试	42
2.3	超级终端和 Minicom	44
2.3.1	超级终端软件的安装	44
2.3.2	配置 Minicom	44
2.3.3	配置 SecureCRT	47
2.4	内核、文件系统加载工具	47
2.4.1	烧写 Bootloader	47
2.4.2	内核和文件系统下载	50
2.4.3	应用程序和文件传输	50
2.5	在开发中使用网络文件系统（NFS）	54
2.5.1	虚拟机设置	54
2.5.2	虚拟机的 IP 地址设置	55
2.5.3	验证网络连接	58
2.5.4	设置共享目录	59
2.5.5	启动 NFS 服务	59
2.5.6	修改共享配置后	60
2.5.7	挂载 NFS	60
2.5.8	双网卡挂载 NFS	60
2.6	小结	61

## 第 2 篇 系统移植技术篇

第 3 章	Bootloader 移植	64
3.1	Bootloader 介绍	64
3.1.1	Bootloader 与嵌入式 Linux 系统的关系	64
3.1.2	Bootloader 基本概念	64
3.1.3	Bootloader 启动过程	66
3.2	Bootloader 之 U-Boot	67
3.2.1	U-Boot 优点	67
3.2.2	U-Boot 的主要功能	67
3.2.3	U-Boot 目录结构	68
3.2.4	U-Boot 的编译	69
3.3	小结	69
第 4 章	Linux 内核裁剪与移植	70
4.1	Linux 内核结构	70
4.1.1	内核的主要组成部分	70
4.1.2	内核源码目录介绍	71
4.2	内核配置选项	73
4.2.1	一般选项	73

4.2.2 内核模块加载方式支持选项 .....	74
4.2.3 系统调用、类型、特性、启动相关选项 .....	75
4.2.4 网络协议支持相关选项 .....	76
4.2.5 设备驱动支持相关选项 .....	76
4.2.6 文件系统类型支持相关选项 .....	77
4.2.7 安全相关选项 .....	78
4.2.8 其他选项 .....	78
4.3 内核裁剪及编译 .....	79
4.3.1 安装内核源代码 .....	79
4.3.2 检查编译环境设置 .....	79
4.3.3 配置内核 .....	80
4.3.4 编译内核 .....	89
4.4 内核升级 .....	90
4.4.1 准备升级内核文件 .....	90
4.4.2 移植过程 .....	90
4.5 小结 .....	93
<b>第 5 章 嵌入式文件系统制作 .....</b>	<b>94</b>
5.1 文件系统选择 .....	94
5.1.1 Flash 硬件方案比较 .....	94
5.1.2 嵌入式文件系统的分层结构 .....	95
5.2 基于 Flash 的文件系统 .....	95
5.2.1 JFFS 文件系统 (Journalling Flash File System) .....	96
5.2.2 YAFFS 文件系统 (Yet Another Flash File System) .....	98
5.2.3 Cramfs 文件系统 (Compressed ROM File System) .....	101
5.2.4 Romfs 文件系统 (ROM File System) .....	102
5.3 基于 RAM 的文件系统 .....	103
5.4 文件系统的制作 .....	103
5.4.1 制作 Ramdisk 文件系统 .....	104
5.4.2 制作 YAFFS2 文件系统 .....	112
5.4.3 制作 JFFS2 文件系统 .....	117
5.4.4 其他文件系统制作 .....	119
5.5 小结 .....	120

### 第 3 篇 系统移植与驱动篇

<b>第 6 章 LCD 驱动移植 .....</b>	<b>122</b>
6.1 认识 LCD 相关硬件原理 .....	122
6.1.1 LCD 概述 .....	122
6.1.2 LCD 控制器 .....	123
6.1.3 LCD 控制器方块图 .....	123
6.1.4 LCD 控制器操作 .....	124
6.1.5 LCD 控制寄存器 .....	129

6.2 LCD 参数设置	132
6.3 内核 LCD 驱动机制	133
6.3.1 FrameBuffer 概述	133
6.3.2 FrameBuffer 设备驱动的结构	133
6.4 Linux 2.6.32 的 LCD 驱动源码分析	137
6.4.1 LCD 驱动开发的主要工作	137
6.4.2 s3c2410fb_init()函数分析	138
6.4.3 s3c2410fb_probe()函数分析	139
6.4.4 s3c2410fb_remove()函数分析	144
6.5 移植内核中的 LCD 驱动	145
6.5.1 LCD 硬件电路图	145
6.5.2 修改 LCD 源码	145
6.5.3 配置内核	150
6.6 小结	153
<b>第 7 章 触摸屏驱动移植</b>	<b>154</b>
7.1 触摸屏概述	154
7.1.1 触摸屏工作原理	154
7.1.2 触摸屏的主要类型	154
7.2 S3C2440 ADC 接口使用	157
7.2.1 S3C2440 触摸屏接口概述	157
7.2.2 S3C2440 触摸屏接口操作	158
7.3 2.6 内核触摸屏驱动源码分析 (s3c2410_ts.c 源码分析)	162
7.4 Linux 内核输入子系统介绍	167
7.4.1 Input 子系统概述	168
7.4.2 输入设备结构体	168
7.4.3 输入链路的创建过程	171
7.4.4 使用 Input 子系统	172
7.4.5 编写输入设备驱动需要完成的工作	174
7.5 触摸屏驱动移植和内核编译	175
7.5.1 修改初始化源码	175
7.5.2 修改硬件驱动源码 s3c2440_ts.c	177
7.5.3 修改 Kconfig 和 Makefile	179
7.5.4 配置编译内核	180
7.5.5 触摸屏测试程序设计	181
7.6 小结	182
<b>第 8 章 USB 设备驱动移植</b>	<b>183</b>
8.1 USB 协议	183
8.1.1 USB 协议的系统主要组成部分	183
8.1.2 总线物理拓扑结构	185
8.1.3 USB 设备、配置、接口、端点	185
8.1.4 USB 设备状态	188
8.1.5 USB 枚举过程	189
8.1.6 USB 请求块 (URB)	192

---

8.2	USB 主机驱动 .....	196
8.2.1	USB 主机驱动结构和功能 .....	196
8.2.2	主机控制器驱动（usb_hcd） .....	197
8.2.3	OHCI 主机控制器驱动 .....	199
8.2.4	S3C24XX OHCI 主机控制器驱动实例 .....	200
8.3	USB 设备驱动 .....	203
8.3.1	USB 骨架程序分析 .....	203
8.3.2	USB 驱动移植的时钟设置 .....	207
8.4	USB 鼠标键盘驱动 .....	208
8.4.1	USB 鼠标驱动代码分析 .....	208
8.4.2	USB 键盘驱动代码分析 .....	211
8.4.3	内核中添加 USB 鼠标键盘驱动 .....	214
8.5	U 盘驱动 .....	215
8.5.1	内核配置 .....	215
8.5.2	移植和测试 .....	216
8.6	小结 .....	218
<b>第 9 章 网卡驱动程序移植 .....</b>		219
9.1	以太网概述 .....	219
9.1.1	以太网连接 .....	219
9.1.2	以太网技术概述 .....	220
9.1.3	以太网的帧结构 .....	222
9.2	网络设备驱动程序体系结构 .....	224
9.2.1	嵌入式 Linux 网络驱动程序介绍 .....	224
9.2.2	Linux 网络设备驱动的体系结构 .....	225
9.2.3	网络设备驱动程序编写方法 .....	225
9.2.4	网络设备驱动程序应用实例 .....	227
9.3	net_device 数据结构 .....	228
9.3.1	全局信息 .....	228
9.3.2	硬件信息 .....	229
9.3.3	接口信息 .....	229
9.3.4	设备方法 .....	232
9.3.5	公用成员 .....	234
9.4	DM9000 网卡概述 .....	234
9.4.1	DM9000 网卡总体介绍 .....	235
9.4.2	DM9000 网卡的特点 .....	235
9.4.3	内部寄存器 .....	236
9.4.4	功能描述 .....	240
9.5	DM9000 网卡驱动程序移植 .....	241
9.5.1	DM9000 网卡连接 .....	241
9.5.2	驱动分析——硬件的数据结构 .....	242
9.5.3	驱动分析——数据读写函数 .....	243
9.5.4	驱动分析——重置网卡 .....	243
9.5.5	驱动分析——初始化网卡 .....	243
9.5.6	驱动分析——打开和关闭网卡 .....	248

---

9.5.7 驱动分析——数据包的发送与接收 .....	249
9.5.8 DM9000 网卡驱动程序移植 .....	251
9.6 小结 .....	254
<b>第 10 章 音频设备驱动程序移植 .....</b>	<b>255</b>
10.1 音频设备接口 .....	255
10.1.1 PCM（脉冲编码调制）接口 .....	255
10.1.2 IIS（Inter-IC Sound）接口 .....	255
10.1.3 AC97（Audio Codec 1997）接口 .....	255
10.1.4 Linux 音频设备驱动框架 .....	256
10.2 Linux 音频设备驱动——OSS 驱动框架 .....	257
10.2.1 OSS 驱动架构硬件 .....	257
10.2.2 OSS 驱动架构代码 .....	257
10.2.3 OSS 初始化函数 oss_init() .....	259
10.2.4 OSS 释放函数 oss_cleanup() .....	260
10.2.5 打开设备文件函数 sound_open() .....	261
10.2.6 录音函数 sound_read() .....	262
10.2.7 播放函数 sound_write() .....	263
10.2.8 控制函数 sound_ioctl() .....	263
10.3 Linux 音频设备驱动——ALSA 驱动框架 .....	265
10.3.1 card 和组件 .....	265
10.3.2 PCM 设备 .....	269
10.3.3 控制接口 .....	272
10.3.4 AC97 API 音频接口 .....	274
10.4 音频设备应用程序编写 .....	278
10.4.1 DSP 接口编程 .....	278
10.4.2 MIXER 接口编程 .....	281
10.4.3 ALSA 应用程序编程 .....	282
10.5 音频设备驱动移植 .....	284
10.5.1 添加 UDA1341 结构体 .....	284
10.5.2 修改录音通道 .....	285
10.5.3 内核中添加 UDA1341 驱动支持 .....	286
10.5.4 移植新内核并进行测试 .....	287
10.6 音频播放程序 madplay 的移植 .....	288
10.6.1 准备移植需要的源文件 .....	288
10.6.2 交叉编译 .....	288
10.6.3 移植和测试 .....	290
10.6.4 编译中可能遇到的问题 .....	290
10.7 小结 .....	290
<b>第 11 章 SD 卡驱动移植 .....</b>	<b>291</b>
11.1 SD 卡简介 .....	291
11.1.1 SD 卡系统概念 .....	291
11.1.2 SD 卡寄存器 .....	291
11.1.3 SD 功能描述 .....	292

---

11.2 SD 卡驱动程序分析.....	295
11.2.1 host 驱动部分 .....	296
11.2.2 core 驱动部分 .....	302
11.2.3 card 驱动部分 .....	307
11.3 SD 卡移植步骤.....	310
11.3.1 添加延时和中断.....	310
11.3.2 配置内核 .....	310
11.3.3 烧写新内核 .....	312
11.4 小结.....	313
<b>第 12 章 NandFlash 驱动移植.....</b>	<b>314</b>
12.1 NandFlash 介绍.....	314
12.1.1 NandFlash 命令介绍.....	314
12.1.2 NandFlash 控制器.....	315
12.2 NandFlash 驱动介绍.....	316
12.2.1 Nand 芯片结构 .....	316
12.2.2 NandFlash 驱动分析.....	317
12.3 NandFlash 驱动移植.....	323
12.3.1 内核的修改.....	323
12.3.2 内核的配置和编译.....	325
12.4 小结.....	326

## 第 4 篇 系统移植高级篇

<b>第 13 章 MiniGUI 与移植.....</b>	<b>328</b>
13.1 MiniGUI 在上位机中的安装.....	328
13.1.1 安装需要的安装文件 .....	328
13.1.2 MiniGUI 的运行模式 .....	328
13.1.3 编译并安装 MiniGUI .....	329
13.1.4 编译安装 MiniGUI 需要的图片支持库 .....	331
13.1.5 编译 MiniGUI 应用程序例子 .....	331
13.2 MiniGUI 的交叉编译和移植 .....	332
13.2.1 交叉编译 MiniGUI .....	332
13.2.2 移植 MiniGUI 程序 .....	334
13.3 小结.....	336
<b>第 14 章 Qt 开发与 Qtopia 移植 .....</b>	<b>337</b>
14.1 Qt 安装与编程 .....	337
14.1.1 下载安装 Qt .....	337
14.1.2 Qt 编程 .....	338
14.1.3 使用 qmake 生成 Makefile .....	340
14.2 Qtopia Core 在 X86 平台上的安装和应用 .....	341
14.2.1 Qtopia Core 安装准备 .....	341
14.2.2 编译 Qtopia Core .....	343

14.2.3 Qtopia 在 X86 平台上的应用开发	343
14.3 Qtopia Core 在嵌入式 Linux 上的移植	347
14.3.1 Qtopia Core 移植准备	347
14.3.2 交叉编译 Qtopia Core	348
14.3.3 编译内核	350
14.3.4 应用程序开发	351
14.3.5 应用程序移植	354
14.4 小结	354
<b>第 15 章 嵌入式数据库 Berkeley DB 移植</b>	<b>355</b>
15.1 数据库的基本概念	355
15.1.1 利用文档和源代码	355
15.1.2 创建环境句柄	355
15.1.3 创建数据库句柄	356
15.1.4 打开数据库	357
15.1.5 DBT 结构	357
15.1.6 存取数据	358
15.1.7 关闭数据库	359
15.2 Berkeley DB 数据库安装	359
15.2.1 安装成 C 库	359
15.2.2 安装成 C++库	360
15.2.3 交叉编译安装 Berkeley DB	360
15.3 使用 Berkeley DB 数据库	362
15.3.1 代码分析	362
15.3.2 编译运行程序	365
15.4 移植 Berkeley DB 数据库	366
15.4.1 数据库设计	366
15.4.2 编写应用程序	366
15.4.3 调试和交叉编译应用程序	368
15.4.4 数据库的移植和测试	369
15.5 小结	369
<b>第 16 章 嵌入式数据库 SQLite 移植</b>	<b>370</b>
16.1 SQLite 支持的 SQL 语句	370
16.1.1 数据定义语句	370
16.1.2 数据操作语句	371
16.2 SQLite 数据库编译、安装和使用	371
16.2.1 安装 SQLite	372
16.2.2 利用 SQL 语句操作 SQLite 数据库	372
16.2.3 利用 C 接口访问 SQLite 数据库	373
16.3 移植 SQLite	376
16.3.1 交叉编译 SQLite	376
16.3.2 交叉编译应用程序	377
16.4 移植 SQLite 数据库	377
16.4.1 文件移植	378

---

16.4.2 运行应用程序.....	378
16.4.3 测试 sqlite3 .....	379
16.5 小结.....	380
<b>第 17 章 嵌入式 Web 服务器 BOA 移植 .....</b>	<b>381</b>
17.1 BOA 介绍 .....	381
17.1.1 BOA 的功能.....	381
17.1.2 BOA 流程分析.....	382
17.1.3 BOA 配置信息.....	385
17.2 BOA 编译和 HTML 页面测试 .....	387
17.2.1 编译 BOA 源代码.....	387
17.2.2 设置 BOA 配置信息.....	388
17.2.3 测试 BOA .....	388
17.3 CGI 脚本测试.....	390
17.3.1 编写测试代码.....	390
17.3.2 编译测试程序.....	390
17.3.3 测试 CGI 脚本 .....	390
17.4 BOA 交叉编译与移植.....	391
17.4.1 交叉编译 BOA.....	391
17.4.2 准备测试程序.....	391
17.4.3 配置 BOA .....	392
17.4.4 测试 .....	392
17.5 BOA 与 SQLite 结合 .....	393
17.5.1 通过 CGI 程序访问 SQLite.....	393
17.5.2 编译和测试 .....	395
17.6 小结.....	396
<b>第 18 章 嵌入式 Web 服务器 Thhttpd 移植 .....</b>	<b>397</b>
18.1 Thhttpd 介绍 .....	397
18.1.1 Web 服务器比较 .....	397
18.1.2 Thhttpd 的特点 .....	397
18.1.3 Thhttpd 核心代码分析 .....	398
18.2 Thhttpd 编译和 HTML 页面测试 .....	401
18.2.1 配置文件介绍 .....	401
18.2.2 Thhttpd 编译 .....	402
18.2.3 运行和测试 Thhttpd .....	402
18.3 CGI 脚本测试 .....	405
18.3.1 编写测试代码 .....	405
18.3.2 编译测试程序 .....	406
18.3.3 测试 CGI 脚本 .....	406
18.4 Thhttpd 交叉编译与移植 .....	406
18.4.1 交叉编译 Thhttpd .....	406
18.4.2 交叉编译 CGI 程序 .....	407
18.4.3 移植 Thhttpd .....	407
18.4.4 测试 .....	408

18.5 Thttpd 与嵌入式数据库结合 .....	408
18.5.1 通过 CGI 程序访问 SQLite.....	409
18.5.2 编译和测试.....	411
18.6 小结.....	411
<b>第 19 章 JVM 及其移植.....</b>	<b>412</b>
19.1 JVM 介绍.....	412
19.1.1 JVM 原理.....	412
19.1.2 JVM 支持的数据类型.....	413
19.1.3 JVM 指令系统.....	414
19.1.4 JVM 寄存器.....	414
19.1.5 JVM 栈结构.....	414
19.1.6 JVM 碎片回收堆.....	417
19.1.7 JVM 异常抛出和异常捕获.....	417
19.2 类装载.....	418
19.2.1 装载类的结构体.....	418
19.2.2 装载类的操作.....	419
19.3 垃圾回收.....	421
19.3.1 mark-and-sweep 回收算法.....	422
19.3.2 分代回收算法.....	423
19.3.3 增量收集.....	424
19.4 解析器.....	424
19.4.1 函数 Interpret().....	424
19.4.2 函数 FastInterpret().....	425
19.4.3 函数 SlowInterpret () .....	427
19.5 Java 编程浅析.....	428
19.5.1 Java 程序命令.....	428
19.5.2 Java 构造函数.....	428
19.5.3 Java 主函数.....	428
19.5.4 Java 程序编译与运行.....	429
19.6 KVM 执行过程.....	429
19.6.1 KVM 启动过程.....	429
19.6.2 KVM 用到的计数器清零.....	432
19.6.3 KVM 初始化内存管理.....	433
19.6.4 KVM 中的哈希表初始化.....	434
19.6.5 KVM 中的事件初始化.....	435
19.6.6 KVM 中的资源释放.....	435
19.7 PC 机安装 JVM.....	435
19.7.1 JVM 在 Windows 上的安装.....	436
19.7.2 JVM 在 Linux 上的安装.....	437
19.8 KVM 移植和测试.....	438
19.8.1 SDK 安装和环境变量设置.....	438
19.8.2 修改 Makefile 和代码.....	438
19.8.3 KVM 编译.....	439
19.8.4 测试.....	439

---

19.8.5 移植.....	441
19.9 小结.....	443
<b>第 20 章 VoIP 技术与 Linphone 编译 .....</b>	<b>444</b>
20.1 VoIP 介绍 .....	444
20.1.1 VoIP 基本原理.....	444
20.1.2 VoIP 的基本传输过程.....	445
20.1.3 VoIP 的优势.....	445
20.1.4 VoIP 的实现方式.....	445
20.1.5 VoIP 的关键技术.....	446
20.2 oSIP 协议概述 .....	446
20.3 oSIP 状态机 .....	447
20.3.1 ICT (Invite Client (outgoing) Transaction) 状态机 .....	447
20.3.2 NICT (Non-Invite Client (outgoing) Transaction) 状态机 .....	457
20.3.3 IST (Invite Server (incoming) Transaction) 状态机 .....	457
20.3.4 NIST (Non-Invite Server (incoming) Transaction) 状态机 .....	458
20.4 oSIP 解析器 .....	459
20.4.1 初始化解析类型函数 osip_body_init().....	459
20.4.2 释放函数 osip_body_free().....	459
20.4.3 字符串到 body 类型转换函数 osip_body_parse().....	460
20.4.4 body 类型到字符串类型转换函数 osip_body_to_str() .....	460
20.4.5 克隆函数 osip_body_clone() .....	462
20.4.6 oSIP 解析器分类 .....	464
20.5 oSIP 事务层 .....	464
20.6 SIP 建立会话的过程 .....	467
20.7 RTP 协议 .....	468
20.7.1 RTP 基本概念.....	468
20.7.2 发送 RTP.....	470
20.7.3 接收 RTP.....	472
20.8 Linphone 编译与测试 .....	474
20.8.1 编译 Linphone 需要的软件包 .....	474
20.8.2 X86 平台上编译和安装 .....	475
20.8.3 Linphone 测试 .....	478
20.8.4 进一步的测试和开发 .....	481
20.9 Linphone 交叉编译 .....	482
20.9.1 Linphone 的交叉编译 .....	482
20.9.2 Linphone 的测试 .....	484
20.10 小结 .....	486

# 第1篇 系统移植基础篇

- ▶ 第1章 Linux 内核介绍
- ▶ 第2章 嵌入式 Linux 开发环境搭建

# 第1章 Linux 内核介绍

系统移植的核心是内核移植。内核移植不仅影响内核的功能，而且还影响到整个系统的性能。因此，了解 Linux 内核，有利于开发人员进行系统裁剪和移植。下面主要针对 Linux 内的 5 个重要部分（系统调用接口、进程管理、内存管理、虚拟文件系统和设备驱动程序）进行介绍。

## 1.1 系统调用接口

系统调用是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口获得操作系统内核提供的服务。例如，用户可以通过文件系统相关的调用请求系统打开文件、关闭文件或读写文件；通过时钟相关的系统调用获得系统时间或设置系统时间；通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

### 1.1.1 Linux 系统调用

所有的操作系统在内核里都有一些内建的函数，这些函数完成对硬件的访问和对文件的打开、读、写、关闭等操作。Linux 系统中称这些函数为“系统调用”（即 `syscall`）。这些函数实现了将操作从用户空间转换到内核空间，有了这些接口函数，用户就可以方便地访问硬件。例如，在用户空间调用 `open()` 函数，则会在内核空间调用 `sys_open()`。一个已经安装的系统所支持的系统调用都可以在 `/usr/include/bits/syscall.h` 文件里看到。

为了对系统进行保护，Linux 系统定义了内核模式和用户模式。内核模式可以执行一些特权指令并进入用户模式，而用户模式不能进入内核模式。Linux 将程序的运行空间也分为内核空间和用户空间，它们分别运行在不同的级别上。在逻辑上，它们是相互隔离的。系统调用规定用户进程进入内核空间的具体位置。在执行系统调用时，程序运行空间将会从用户空间转移到内核空间，处理完毕后再返回到用户空间。

### 1.1.2 用户编程接口

用户编程接口是为用户编程过程提供的各种功能库函数，如分配空间、拷贝字符、打开文件等。Linux 用户编程接口（API）遵循了在 UNIX 中最流行的应用编程界面标准——POSIX 标准。它与系统调用之间存在一定的联系和区别。不同的语言和平台为用户提供了丰富的编程接口，包括网络编程接口、图形编程接口、数据库编程接口等，但这些不是系统调用。

系统调用与用户编程接口之间存在联系。一个或者多个系统调用会对应到一个具体的应用程序使用的 API；但是，并非所有的 API 都需要使用到系统调用。

根据《深入理解 Linux 内核》一书中对用户编程接口（API）和系统调用两者区别的描述为，前者只是一个函数定义，说明了如何获得一个给定的服务；而后者是通过软中断向内核态发出一个明确的请求。如果按照层次关系来分，系统调用为底层，而用户编程接口为上层。一个用户编程接口由 0 个或者多个系统调用组成。

### 1.1.3 系统调用与服务例程的对应关系

为了通过系统调用号来调用不同的内核服务例程，系统必须创建并管理好一张系统调用表。该表用于系统调用号与内核服务函数的映射，Linux 用数组 `sys_call_table` 表示这个表。在这个表的每个表项中存放着对应内核服务例程的指针，而该表项的下标就是该内核服务例程的系统调用号。Linux 规定，在 i386 体系中，处理器的寄存器 `eax` 用来传递系统调用号。

### 1.1.4 系统调用过程

通常情况下，`abc()` 系统调用对应的服务例程的名字是 `sys_abc()`。图 1.1 表示了系统调用和应用程序、对应的封装例程、系统调用处理程序及系统调用服务例程之间的关系。下面使用一个例子来简单说明系统调用过程。

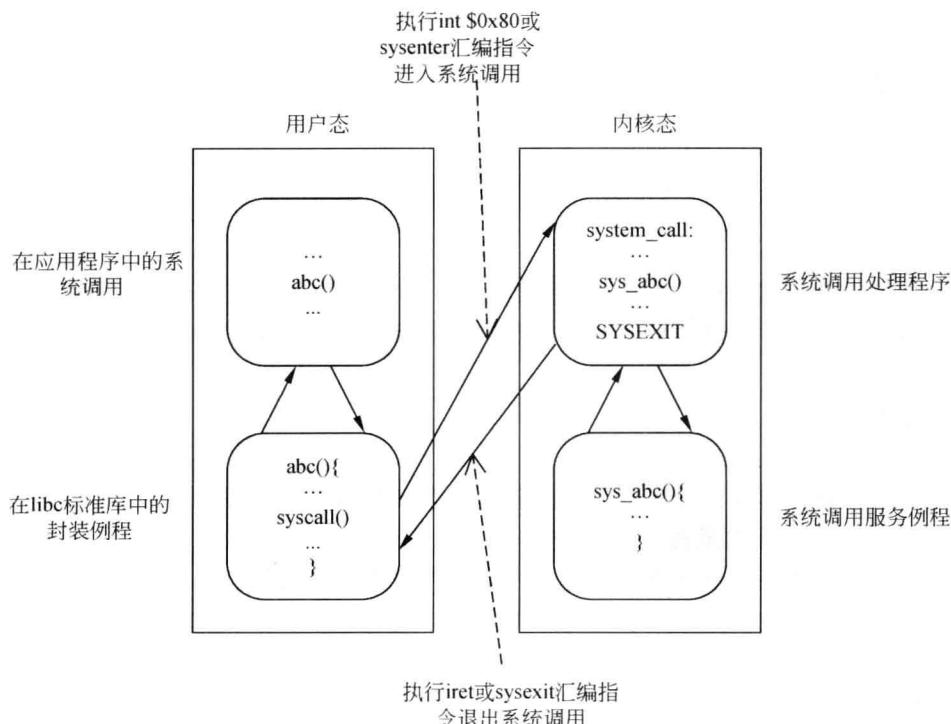


图 1.1 系统调用的处理过程

- (1) 用户程序中调用库函数 `abc()`。
- (2) 系统加载 `libc` 库调用索引和参数后，执行 `int $0x80` 或者 `sysenter` 汇编指令进入系统调用，执行 `system_call()` 函数。
- (3) `system_call()` 函数根据传递过来的参数处理所有的系统调用。使用 `system_call_table[参数]` 执行系统调用。
- (4) 系统调用返回。
- (5) 执行 `iret` 或者 `sysexit` 汇编指令两种方式退出系统调用，并调用 `resume_userspace()` 函数进入用户空间。
- (6) 继续在 `libc` 库中执行，执行完成后返回到用户应用程序中。

执行 `int $0x80` 或者 `sysenter` 汇编指令两种方式进入系统调用，在老版本的 Linux 内核中只支持 `int $0x80` 中断方式。执行 `iret` 或者 `sysexit` 汇编指令两种方式退出系统调用，如图 1.1 中的虚线指引线所示。

### 1.1.5 系统调用传递的参数

系统调用中输入输出的参数为实际传递的值或者是用户态进程的地址，或者是指向用户态函数指针的数据结构地址。传递的参数放在寄存器 `eax` 中，即系统调用号。寄存器传递参数的个数满足两个条件：

- 参数的长度不超过寄存器的长度，如果是 32 位平台不超过 32 位，64 位平台不超过 64 位。
- 不包括 `eax` 中的系统调用号，参数的个数不超过 6 个。

内核在为用户请求提供服务时，会检查所有的系统调用参数。检查参数时，主要对参数的权限和参数表示的地址空间的有效性进行验证。

## 1.2 进程管理

进程管理包括创建进程、管理进程及删除进程。进程管理是 Linux 内核的重要部分，对系统的核心资源进行管理。要做好系统移植就需要对这部分知识有一定的了解。

### 1.2.1 进程

进程是程序执行时的一个实体。程序包含指令和数据，而进程包含程序计数器和全部 CPU 寄存器的值。进程的堆栈中存储着一些数据，如子程序参数、返回地址及变量之类的临时数据。当前的执行程序（进程）包含当前处理器中的活动状态。

Linux 是一个多处理操作系统，进程拥有独立的权限和单一职责。如果系统中某个进程发生崩溃，它不会影响到另外的进程。每个进程都运行在各自独立的虚拟地址空间中，只有通过核心控制下可靠的进程通信机制，它们之间才能发生通信。进程通信机制包括管道、信号、信号量、消息队列等。

从内核的观点看，进程的目的就是担当分配系统资源的实体。系统资源包括 CPU 时间、

内存等。因此，进程管理的最终目的就是在进程畅通执行的条件下，合理分配系统资源给不同的进程。

当一个进程创建时，它几乎与父进程相同，即是父进程地址空间的一个（逻辑）备份。从进程创建系统调用的下一条指令开始，执行与父进程相同的代码。虽然父子进程含有共享的程序代码，但是分别拥有独立的数据备份。因此子进程对堆和栈中的数据进行修改时，对父进程的数据是不会有影响的。

## 1.2.2 进程描述符

内核对进程的优先级、进程的状态、地址空间等采用进程描述符表示。在 Linux 内核中，进程用一个相当大的称为 `task_struct` 的结构表示。下面是从 `linux-2.6.29\include\linux\sched.h` 中摘抄出来的进程描述的部分信息。

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth; /* BKL lock depth */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
    unsigned int policy;
    cpumask_t cpus_allowed;
    struct list_head tasks;
    struct mm_struct *mm, *active_mm;

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */

    unsigned int personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;

    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */
    struct list_head ptraced;
    struct list_head ptrace_entry;
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    struct completion *vfork_done; /* for vfork() */
}
```

```

int __user *set_child_tid;      /* CLONE_CHILD_SETTID */
int __user *clear_child_tid;    /* CLONE_CHILD_CLEARTID */
...
};

```

下面简单对上述内容进行描述。

- **state**: 表示进程的状态，-1 代表“不能运行”，0 代表“运行”，>0 代表“停止”。
- **flags**: 定义了很多指示符，表明进程是否正在被创建 (PF\_STARTING) 或退出 (PF\_EXITING)，或是进程当前是否在分配内存 (PF\_MEMALLOC)。可执行程序的名称（不包含路径）占用 comm (命令) 字段。
- 每个进程都会被赋予优先级（称为 static\_prio），但进程的实际优先级是基于加载及其他几个因素动态决定的。优先级值越低，实际的优先级越高。
- **tasks** 字段提供了链接列表的能力。它包含一个 prev 指针（指向下一个任务）和一个 next 指针（指向下一个任务）。

进程的地址空间由 mm 和 active\_mm 字段表示。mm 代表的是进程的内存描述符，而 active\_mm 则是前一个进程的内存描述符（为改进上下文切换时间的一种优化）。

### 1.2.3 进程状态

进程描述符中 state 字段描述进程当前的状态。它由一组标志组成，其中每个标志描述一种可能的进程状态。在 2.6 内核中，进程只能处于这些状态中的一种。下面分别对这些状态进行描述。

- 可运行状态 (TASK\_RUNNING)：进程处于运行（它是系统的当前进程）或者准备运行状态（它在等待系统将 CPU 分配给它）。
- 等待状态 (WAITING)：进程在等待一个事件或者资源。Linux 将等待进程分成两类：可中断的等待状态 (TASK\_INTERRUPTIBLE) 与不可中断的等待状态 (TASK\_UNINTERRUPTIBLE)。可中断等待进程可以被信号中断；不可中断等待进程直接在硬件条件等待，并且任何情况下都不可中断。
- 暂停状态 (TASK\_STOPPED)：进程被暂停，通常是通过接收一个信号 (SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU) 转为暂停状态。正在被调试的进程可能处于停止状态。
- 僵死状态 (EXIT\_ZOMBIE)：进程的执行被终止，但是，父进程还没有发布 wait4() 或 waitpid() 系统调用返回有关死亡进程的信息。

### 1.2.4 进程调度

Linux 进程调度的目的就是调度程序运行时，要在所有可运行状态的进程中选择最值得运行的进程投入运行。每个进程的 task\_struct 结构中有 policy、priority、counter 和 rt\_priority 这 4 项，是选择进程的依据。

其中，policy 为进程调度策略，用于区分普通进程和实时进程，实时进程优先于普通进程运行；priority 是进程（包括实时和普通）的静态优先级；counter 是进程剩余的时间片，它的起始值就是 priority 的值；因为 counter 用于计算一个处于可运行状态的进程值的

运行的程度 goodness，所以 counter 也被看做是进程的动态优先级。rt\_priority 是实时进程特有的优先级别，用于实时进程间的选择。

## 1. Linux进程分类

Linux 在执行进程调度的时候，对不同类型的进程采取的策略也不同，一般将 Linux 分为以下 3 类。

- 交互式进程：这类进程经常和用户发生交互，所以花费一些时间等待用户的操作。当有用户输入时，进程必须很快地激活。通常要求延迟在 50~150 毫秒。典型的交互式进程有控制台命令、文本编辑器、图形应用程序等。
- 批处理进程（Batch Process）：这类进程不需要用户交互，一般在后台运行。所以不需要非常快地反应，它们经常被调度期限制。典型的批处理进程有编译器、数据库搜索引擎和科学计算等。
- 实时进程：这类进程对调度有非常严格的要求，这种类型的进程不能被低优先级进程阻塞，并且在很短的时间内做出反应。典型的实时进程有音视频应用程序、机器人控制等。

## 2. Linux进程优先级

Linux 系统中每一个普通进程都有一个静态优先级。这个值会被调度器作为参考来调度进程。在内核中调度的优先级区间为[100,139]，数字越小，优先级越高。一个新的进程总是从它的父进程继承此值。Linux 进程优先级还包括动态优先级、实时优先级等，各个进程优先级描述如下。

- 静态优先级（priority）：被称为“静态”是因为它不随时间而改变，只能由用户进行修改。它指明了在被迫和其他进程竞争 CPU 之前该进程所被允许的时间片的最大值（20）。
- 动态优先级（counter）：counter 即系统为每个进程运行而分配的时间片。Linux 用它来表示进程的动态优先级。当进程拥有 CPU 时，counter 就随着时间不断减小。当它递减为 0 时，标记该进程将重新调度。它指明了在当前时间片中所剩余的时间量（最初为 20）。
- 实时优先级（rt\_priority）：它的变化范围是从 0~99。任何实时进程的优先级都高于普通的进程。
- Base time quantum：是由静态优先级决定，当进程耗尽当前 Base time quantum，kernel 会重新分配一个 Base time quantum 给它。静态优先级和 Base time quantum 的关系如下所述。

(1) 当静态优先级小于 120：

```
Base time quantum (in millisecond) = (140 - static priority) * 20
```

(2) 当静态优先级大于等于 120：

```
Base time quantum (in millisecond) = (140 - static priority) * 5
```

## 3. Linux进程的调度算法

在 Linux 系统中，进程作为程序的实体始终运行在系统中，并且进程占用系统资源，

所以进程调度算法优劣将会严重影响到系统的性能。为提高系统性能设计进程调度算法的原则，应该遵循进程响应尽量快，后台进程的吞吐量尽量大，尽量避免进程“饿死”现象，低优先级和高优先级进程需要尽可能调和。以下为常见的进程调度算法。

- 时间片轮转调度算法（round-robin）：SCHED\_RR 用于实时进程。系统使每个进程依次地按时间片轮流执行的方式。
- 优先权调度算法：SCHED\_NORMAL 用于非实时进程。每次系统都会选择队列中优先级最高的进程运行。Linux 采用抢占式的优级算法，即系统中当前运行的进程永远是可运行进程中优先权最高的进程。
- FIFO（先进先出）调度算法：SCHED\_FIFO 用于实时进程。采用 FIFO 调度算法选择的实时进程必须是运行时间较短的进程，因为这种进程一旦获得 CPU 就只有等到它运行完或因等待资源主动放弃 CPU 时，其他进程才能获得运行机会。

## 1.2.5 进程地址空间

Linux 的虚拟地址空间为 0~4GB。虚拟的 4GB 空间被 Linux 内核分为内核空间和用户空间两部分。将最高的 1GB（从虚拟地址 0xC0000000~0xFFFFFFFF）留给内核使用，称为“内核空间”。将较低的 3GB（从虚拟地址 0x00000000~0xBFFFFFFF）留给用户进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核空间被系统的所有进程共享，实际上对于某个进程来说，它仍然可以拥有 4GB 的虚拟空间。

其中，很重要的一点是虚拟地址空间，并不是实际的地址空间。在为进程分配地址空间时，根据进程需要的空间进行分配，4GB 仅仅是最大限额而已，并非一次性将 4GB 分配给进程。一般一个进程的地址空间总是小于 4GB 的，可以通过查看/proc/pid/maps 文件来获悉某个具体进程的地址空间。但进程的地址空间并不对应实际的物理页，Linux 采用 Lazy 的机制来分配实际的物理页（Demand paging 和“写时复制（Copy On Write）的技术”），从而提高实际内存的使用率，即每个虚拟内存页并不一定对应于物理页。虚拟页和物理页的对应是通过映射机制来实现的，即通过页表进行映射到实际的物理页。因为每个进程都有自己的页表，因此可以保证不同进程的相同虚拟地址可以映射到不同的物理页，从而为不同的进程都可以同时拥有 4GB 的虚拟地址空间提供了可能。

内核是系统中优先级最高的部分。内核函数申请动态内存时系统不会推迟这个请求；而进程申请内存空间时，进程的可执行文件被装入后，进程不会立即对所有的代码进行访问。因此内核总是尽量推迟给用户进程分配动态空间。

内核分配空间时，可以通过`_get_free_pages()`或`alloc_pages`从分区页框分配器中获得页框；通过`kmem_cache_alloc()`或`kmalloc()`函数使用 slab 分配器为专用或通用对象分配块；通过`vmalloc()`或`vmalloc32()`函数获得一块非连续的内存区。

与进程地址空间有关的全部信息都包含在一个叫做内存描述符（memory descriptor）的数据结构中，其结构类型为`mm_structs`进程描述符的`mm`字段就是指向这个结构的。

### 1. 创建进程地址空间

`copy_mm()`函数通过建立新进程的所有页表和内存描述符，来创建进程的地址空间。

```

01 static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
02 {
03     struct mm_struct * mm, *oldmm;
04     int retval;
05
06     tsk->min_flt = tsk->maj_flt = 0;
07     tsk->nvcsw = tsk->nivcsw = 0;
08
09     tsk->mm = NULL;
10     tsk->active_mm = NULL;
11
12 /*如果是内核线程的子线程 (mm=NULL) , 则直接退出, 此时内核线程 mm 和 active_mm 均
为 NULL*/
13     oldmm = current->mm;
14     if (!oldmm)
15         return 0;
16
17 /*内核线程, 只是增加当前进程的虚拟空间的引用计数*/
18     if (clone_flags & CLONE_VM) {
19         /*如果共享内存, 将 mm 由父进程赋值给子进程, 两个进程将会指向同一块内存*/
20         atomic_inc(&oldmm->mm_users);
21         mm = oldmm;
22         goto good_mm;
23     }
24
25     retval = -ENOMEM;
26     mm = dup_mm(tsk);           /*完成了对 vm_area_struct 和页面表的复制*/
27     if (!mm)
28         goto fail_nomem;
29
30 good_mm:
31     /* Initializing for Swap token stuff */
32     mm->token_priority = 0;
33     mm->last_interval = 0;
34
35 /*内核线程的 mm 和 active_mm 指向当前进程的 mm_struct 结构*/
36     tsk->mm = mm;
37     tsk->active_mm = mm;
38     return 0;
39
40 fail_nomem:
41     return retval;
42 }

```

## 2. 删除进程地址空间

内核调用 `exit_mm()` 函数释放进程的地址空间。

```

01 static void exit_mm(struct task_struct * tsk)
02 {
03     m_release(tsk, mm);
04     /*得到读写信号量*/
05     down_read(&mm->mmap_sem);
06     core_state = mm->core_state;
07     if (core_state) {
08         struct core_thread self;
09     /*释放读写信号量*/
10     up_read(&mm->mmap_sem);

```

```

11         self.task = tsk;
12         self.next = xchg(&core_state->dumper.next, &self);
13
14         if (atomic_dec_and_test(&core_state->nr_threads))
15             complete(&core_state->startup);
16
17         for (;;) {
18             set_task_state(tsk, TASK_UNINTERRUPTIBLE);
19             if (!self.task) /*take 字段可以查看函数 coredump_
20                           finish()*/
21                 break;
22             schedule();
23         }
24         set_task_state(tsk, TASK_RUNNING);
25         down_read(&mm->mmap_sem);
26     }
27     atomic_inc(&mm->mm_count);
28     BUG_ON(mm != tsk->active_mm);
29     /* more a memory barrier than a real lock */
30     task_lock(tsk);
31     tsk->mm = NULL;
32     up_read(&mm->mmap_sem);
33     enter_lazy_tlb(mm, current);
34     /*释放用户虚拟空间的数据结构*/
35     clear_freeze_flag(tsk);
36     task_unlock(tsk);
37     mm_update_next_owner(mm);
38     /*递减 mm 的引用计数并是否为 0, 如是, 则释放 mm 所代表的映射*/
39     mmput(mm);
40 }

```

## 1.3 内存管理

RAM 的一部分被静态地划分给了内核，用来存放内核代码和静态数据结构。RAM 的其余部分称为动态内存（dynamic memory），这不仅是运行用户进程所需的宝贵资源，也是内核所需的宝贵资源。事实上，整个系统的性能取决于如何有效地管理动态内存。

### 1.3.1 内存管理技术

页表（page tables）：进程在读取指令和存取数据时都要访问内存。在一个虚拟内存系统中，所有的地址都是虚拟地址而非物理地址。操作系统维护虚拟地址和物理地址转换的信息，处理器通过这组信息将虚拟地址转换为物理地址。虚拟内存和物理内存被分为适当大小的块，叫做页。为了将虚拟地址转换为物理地址，首先，处理器要找到虚拟地址的页编号和页内偏移量；然后，处理器根据虚拟地址和物理地址的映射关系将虚拟页编号转换为物理页；最后，根据偏移量访问物理页的确定偏移位置。每个物理页面都有一个 struct page 结构，位于 include/linux/mm.h，该结构体包含了管理物理页面时的所有信息，下面给出该结构体的具体描述：

```

typedef struct page {
    struct list_head list;          //指向链表中的下一页
    struct address_space *mapping;  //用来指定我们正在映射的索引节点 (inode)
    unsigned long index;            //在映射表中的偏移
    struct page *next_hash;         //指向页高速缓存哈希表中下一个共享的页
    atomic_t count;                //引用这个页的个数
    unsigned long flags;            //页面各种不同的属性
    struct list_head lru;           //用在 active_list 中
    wait_queue_head_t wait;         //等待这一页的页队列
    struct page **pprev_hash;       //指向页高速缓存哈希表中前一个共享的页与 next_hash 相对应
    struct buffer_head * buffers;   //把缓冲区映射到一个磁盘块
    void *virtual;
    struct zone_struct *zone;       //页所在的内存管理区
} mem_map_t;

```

## 1. 请求页面调度 (Demand Paging)

为了节省物理内存，只加载执行程序正在使用的虚拟页，这种进行访问时才加载虚拟页的技术叫做 Demand Paging。当一个进程试图访问当前不在内存中的虚拟地址时，处理器无法找到引用的虚拟页对应的页表条目。当处理器无法将虚拟地址转换为物理地址时，处理器通知操作系统发生 page fault。出错的虚拟地址无效，则意味着进程试图访问它不应该访问的虚拟地址。这种情况下，操作系统会中断它，从而保护系统中的其他进程。

如果出错的虚拟地址有效，而只是它所在的页当前不在内存中，操作系统应该从磁盘映像中将对应的页加载到内存中。相对内存存取来讲，磁盘存取需要更长的时间，所以进程一直处于等待状态直到该页被加载到内存中。如果系统当前有其他进程可以运行，操作系统将选择其中一个运行；接着将取到的页写进一个空闲页面，并将一个有效的虚拟页条目加到进程的页表中；然后，这个等待的进程重新执行发生内存出错地方的机器指令。本次虚拟内存存取进行时，处理器能够将虚拟地址转换为物理地址，使得进程能够继续运行。Linux 使用 demand paging 技术将可执行映像加载到进程的虚拟内存中，在执行命令时，包含命令的文件被打开，将该文件的内容映射到进程的虚拟内存中。这个过程通过修改描述进程内存映射的数据结构来实现，也叫做内存映射 (memory mapping)，但实际上只有映像的第一部分真正放在了物理内存中，映像的剩余部分仍然在磁盘上。当映像执行时，它产生 page fault，Linux 使用进程的内存映像表来确定映像的哪一部分需要加载到内存中执行。

## 2. 页面置换技术 (Swapping)

如果进程需要将虚拟页放到物理内存中，而此时已经没有空闲的物理页，操作系统必须废弃物理空间中的另一页，为该页让出空间。如果物理内存中需要废弃的页来自磁盘上的映像或者数据文件，并且该页没有被写过不需要存储，则该页被废弃。如果进程又需要该页，它可以从映像或数据文件中再次加载到内存中。但如果该页已经被改变，操作系统必须保留它的内容以便以后进行访问。这种也叫做 dirty page，当它从物理内存中废弃时，被存到一种叫做交换文件的特殊文件中。由于访问交换文件与访问处理器、物理内存的速

度相比较慢，操作系统必须判断是将数据页写到磁盘上还是将它们保留在内存中以便下次访问。

如果判断哪些页将被废弃或者交换的算法效率不高，则会发生颠簸（thrashing），这时页不停地被写到磁盘上，然后又被读回，操作系统频繁地处理此读写任务而无法执行实际的工作。Linux 使用 LRU (Least Recently Used, 最近最少使用置换算法) 的页面技术公平地选择需要从系统中废弃的页面。

伙伴系统算法用以解决外碎片问题。把所有的空闲页框分为 11 个块链表，每个块链表分别包含 1、2、4、8、16、32、64、128、256、512 和 1024 个连续的页框。对于 1024 个页框的最大请求对应着 4MB 大小的连续 RAM 块。每个块的第一个页框的物理地址是该块大小的整数倍。

非连续内存管理，当对内存区的请求不是很频繁的时候，通过连续的线性地址访问非连续的页框，该方法可以避免外碎片，但是其带来的负面因素打乱了内核表。非连续内存区的大小必须是 4096 字节的倍数。非连续内存区应用的场合分别有分配数据结构给活动的交换区、分配空间给模块和分配缓冲区给某些 I/O 驱动程序。

### 1.3.2 内存区管理

内存区 (memory area) 是具有连续的物理地址和任意长度的内存单元序列。伙伴系统采用的是页框作为基本内存区，适合于大内存的请求，对小内存的请求容易造成内碎片。为了解决内碎片的问题，将内存区大小按几何分布划分，也就是将内存区划分成 2 的幂的大小。不论请求的大小为多大时，总能保证内碎片小于内存区的 50%。为此，内核建立了 13 个按几何分布的空闲内存区链表，大小从 32~131 072 字节。伙伴系统的调用是为了获得存放新内存区所需的额外页框，同时也为了释放不再包含内存区的页框，用一个动态链表来记录每个页框所包含的空闲内存区。

物理内存被划分为 3 个区来管理，它们是 ZONE\_DMA、ZONE\_NORMAL 和 ZONE\_HIGHMEM。每个区都用 struct zone\_struct 结构表示，定义于 include/linux/mmzone.h:

```
typedef struct zone_struct {
    /* Commonly accessed fields:*/
    spinlock_t lock;
    unsigned long free_pages;
    unsigned long pages_min, pages_low, pages_high;
    int need_balance;
    /* free areas of different sizes*/
    free_area_t free_area[MAX_ORDER];
    /* Discontig memory support fields.*/
    struct pglist_data *zone_pgdat;
    struct page *zone_mem_map;
    unsigned long zone_start_paddr;
    unsigned long zone_start_mapnr;
    /*
     * rarely used fields:
     */
    char *name;
    unsigned long size;
} zone_t;
```

各个字段的含义如下。

- ❑ lock：用来保证对该结构中其他域的串行访问。
- ❑ free\_pages：在这个区中现有空闲页的个数。
- ❑ pages\_min、pages\_low 及 pages\_high 是对这个区最少、次少及最多页面个数的描述。
- ❑ need\_balance：与 kswapd 合在一起使用。
- ❑ free\_area：在伙伴分配系统中的位图数组和页面链表。
- ❑ zone\_pgdat：该管理区所在的存储结点。
- ❑ zone\_mem\_map：该管理区的内存映射表。
- ❑ zone\_start\_paddr：该管理区的起始物理地址。
- ❑ zone\_start\_mapnr：在 mem\_map 中的索引（或下标）。
- ❑ name：该管理区的名字。

### 1.3.3 内核中获取内存的几种方式

操作系统的内存管理方案优劣是决定其效率高低的重要因素，时间与空间常常作为内存管理方案优劣的衡量指标。首先，分配/释放内存是一个发生频率很高的操作，所以它要求有一定的实时性，另外，内存又是一种非常宝贵的资源，所以要尽量减少内存碎片的产生。下面介绍内核获取内存的几种方式。

#### 1. 伙伴算法分配大片物理内存

- ❑ alloc\_pages(gfp\_mask, order)：获得连续的页框，返回页描述符地址，是其他类型内存分配的基础。
- ❑ \_get\_free\_pages(gfp\_mask, order)：获得连续的页框，并返回页框对应的线性地址。线性地址与物理地址是内核直接映射方式。该方法不能用于大于 896MB 的高端内存。

#### 2. slab缓冲区分配小片物理内存

内核提供了后备高速缓存机制，称为“slab 分配器”。slab 分配器实现的高速缓存具有 kmem\_cache\_t 类型，可通过调用 kmem\_cache\_create 创建。

- ❑ kmem\_cache\_create：建立 slab 的高速缓冲区。
- ❑ kmem\_cache\_alloc：试图从本地高速缓存获得一个空闲的对象。
- ❑ kmalloc(gfp\_mask, size)：获得连续的以字节为单位的物理内存，返回线性地址。

#### 3. 非连续内存区分配

vmalloc(size)：分配非连续内存区，其线性地址连续，物理地址不连续，减少了外碎片，但是其性能低，因为要打乱内核页表。通常只是分配大内存，例如为活动的交互区分配数据结构、加载内核模块时分配空间、为 I/O 驱动程序分配缓冲区。

#### 4. 高端内存映射

- ❑ kmap(struct page \* page)：用于获得高端内存永久内核映射的线性地址。

- `kmap_atomic` (`struct page * page, enum km_type type`): 用于获得高端内存临时内核映射的线性地址。

## 5. 固定线性地址映射

- `set_fixmap(idx, phys)`: 把一个物理地址映射到一个固定的线性地址上。
- `set_fixmap_nocache(idx, phys)`: 把一个物理地址映射到一个固定的线性地址上, 禁用该页高速缓存。

# 1.4 虚拟文件系统

虚拟文件系统的思想是把不同种类的文件系统的共同信息放入内核。其中一个字段或函数来支持 Linux 所支持的各种文件系统提供的操作。对所调用的读、写或其他函数, 内核都能把它们替换成支持 Linux 文件系统、NFS 文件系统, 或者其他文件系统的实际函数。在第 2 章中, 会讲到 Linux 的安装, 在虚拟机上安装 Linux, 同时实现 Linux 和 Windows 文件共享, 即实现在 Linux 环境下能够直接访问 Windows 的 FAT32 文件系统。

## 1.4.1 虚拟文件系统作用

虚拟文件系统 (Virtual Filesystem), 实际上是对各种文件系统的一种封装, 为各种文件系统提供了一个通用的接口。通常情况下, 为了实现不同操作系统下的文件访问, 例如, 复制`/usr/local/arm`目录下的`zImage`文件到`/mnt/hgfs/Windows`目录下。

```
$ scp /usr/local/arm/zImage /mnt/hgfs/Windows/
```

在不同文件系统中实现文件复制, 其执行的原理如图 1.2 所示。

VFS 支持的文件系统可分为以下 3 个主要类型。

### 1. 磁盘文件系统

这些文件系统管理本地磁盘中可用的存储空间或者其他可以起到磁盘作用的设备 (如 USB 闪存或硬盘)。这些文件系统包括:

- Linux 使用的第二扩展文件系统 (Ext2), 第三扩展文件系统 (Ext3) 及 Reiser 文件系统 (ReiserFS)。
- UNIX 家族文件系统, 如 sysv 文件系统 (System V、Coherent、Xenix)、UFS (BSD、Solaris、NEXTSTEP), MINIX 文件系统及 VERITAS VxFS (SCO UnixWare)。
- Windows 支持的文件系统, 如 MS-DOS、

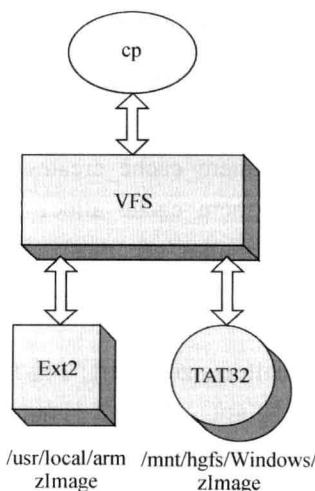


图 1.2 不同文件系统中实现文件复制

FAT、FAT32、NTFS 等文件系统。

- ISO9660CD-ROM 文件系统和通用磁盘的 DVD 文件系统。
- 其他文件系统，如 HPFS（IBM 公司的 OS/2）、HFS（苹果公司的 Macintosh）、AFFS（Amiga 公司的快速文件系统）及 ADFS（Acorn 公司的磁盘文件系统）。

## 2. 网络文件系统（NFS）

网络文件系统最主要的功能就是让网络上的主机可以共享目录及资料。将远端所共享出来的系统，挂载（mount）在本地端的系统上，然后就可以很方便地使用远端的资料，而操作起来就像在本地操作一样。使用 NFS 有相当多的好处，例如文档可以集中管理、节省磁盘空间、资源共享等。

## 3. 特殊文件系统

特殊文件系统可以为系统程序员和管理员提供一种容易的方式，来操作内核的数据结构并实现操作系统的特殊特征。

### 1.4.2 文件系统的注册

每个注册的文件系统是指可能会被挂载到目录树中的各个实际文件系统。实际文件系统，是指 VFS 中的实际操作最终要通过它们来完成而已，并不表示它们必须要存在于特定的某种存储设备上。注册过程实际上是将表示各实际文件系统的 struct file\_system\_type 数据结构的实例化，接着形成一个链表，内核中用一个名为 file\_systems 的全局变量来指向该链表的表头。下面为 file\_system\_type 数据结构。

```
struct file_system_type {
    const char *name;
    int fs_flags;
    int (*get_sb) (struct file_system_type *, int, const char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    struct lock_class_key s_lock_key;
    struct lock_class_key s_umount_key;
};
```

- name：文件系统名，如 ext2。
- fs\_flags：文件系统类型标志。
- get\_sb：读超级块的方法。
- kill\_sb：删除超级块的方法。
- owner：指向实现文件模块的文件指针。
- next：指向文件系统类型链表中下一个文件系统的指针。
- fs\_supers：具有相同文件系统类型的超级块对象链表的头。

### 1.4.3 文件系统的安装和卸载

在 Linux 系统中，同一个文件系统可以被多次安装。如果一个文件系统被安装多次，那么就可以通过这多个安装点来访问文件系统。尽管可以通过这多个安装点来访问，但是文件系统却只有一个。不论文件系统被安装了多少次，都只有一个超级块对象。安装一个文件系统遵循的步骤：

(1) Linux 系统内核必须首先检查有关参数的有效性。VFS 首先应找到准备安装的文件系统。查找的方式是，通过查找 `file_systems` 指针指向的链表中的 `file_system_type` 数据结构项，来搜索已知的文件系统（该结构中包含文件系统的名字和指向 VFS 超级块读取程序地址的指针），当找到一个匹配的名字时，就可以得到读取文件系统超级块的程序地址。

(2) 查找作为新文件系统安装点的 VFS 索引结点，并且同一目录下只能安装一个文件系统；VFS 安装程序必须分配一个 VFS 超级块 (`super_block`)，并且向它传递一些有关文件系统安装的信息。

(3) 申请一个 `vfsmount` 数据结构（其中包括存储文件系统的块设备的设备号、文件系统安装的目录和一个指向文件系统的 VFS 超级块的指针），并使它的指针指向所分配的 VFS 超级块。

(4) 当文件系统安装以后，该文件系统的根索引结点就一直保存在 VFS 索引结点缓存中。

**卸载文件系统：**验证被卸载文件系统是否为可卸载的，如果该文件系统中的文件当前正被使用，则该文件系统不能被卸载；如果文件系统中的文件或目录正在使用，则 VFS 索引结点高速缓存中可能包含对应的 VFS 索引结点；如果相应的结点标志为“被修改过”，则该文件系统不能被卸载。如果验证被卸文件系统为可卸载的，就释放相应的 VFS 超级块和安装点，从而卸载该文件系统。

`vfsmount` 数据结构如下：

```
struct vfsmount
{
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent;           /* fs we are mounted on */
    struct dentry *mnt_mountpoint;        /* dentry of mountpoint */
    struct dentry *mnt_root;              /* root of the mounted tree */
    struct super_block *mnt_sb;           /* pointer to superblock */
    struct list_head mnt_mounts;          /* list of children, anchored here */
    struct list_head mnt_child;           /* and going through their mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    int mnt_expiry_mark;                /* true if marked for expiry */
    char *mnt_devname;                  /* Name of device e.g. /dev/dsk/hda1 */
    struct list_head mnt_list;
    struct list_head mnt_fslink;          /* link in fs-specific expiry list */
    struct namespace *mnt_namespace;      /* containing namespace */
};
```

- `mnt_hash`: 用于散列表链表的指针。
- `mnt_parent`: 指向父文件系统，这个文件系统安装在其上。

- ❑ `mnt_mountpoint`: 指向这个文件系统安装点目录的 `dentry`。
- ❑ `mnt_root`: 指向这个文件系统根目录的 `dentry`。
- ❑ `mnt_sb`: 指向这个文件系统的超级块对象。
- ❑ `mnt_mounts`: 包含所有文件系统描述符的链表头。
- ❑ `mnt_child`: 用于已安装文件系统 `mnt_mounts` 的指针。
- ❑ `mnt_count`: 引用计数器, 增加该值禁止文件系统被卸载。
- ❑ `mnt_flags`: 安装标志。
- ❑ `mnt_expiry_mark`: 如果文件系统到期就设置该标志为 `true`。
- ❑ `mnt_devname`: 设备文件名。
- ❑ `mnt_list`: 已安装文件描述符的 `namespace` 链表的指针。
- ❑ `mnt_fslink`: 具体文件系统到期链表的指针。
- ❑ `mnt_namespace`: 指向安装了文件系统的 `namespace` 链表的指针。

## 1.5 设备驱动程序

设备驱动, 实际上是硬件功能的一个抽象。针对同一个硬件不同的驱动可以将硬件封装成不同的功能。设备驱动是硬件层和应用程序(或者操作系统)的媒介, 能够让应用程序或者操作系统使用硬件。在 Linux 操作系统下有 3 类主要的设备文件类型: 块设备、字符设备和网络设备。设备驱动程序是指管理某个外围设备的一段代码, 它负责传送数据、控制特定类型的物理设备的操作, 包括开始和完成 I/O 操作, 检测和处理设备出现的错误。

### 1.5.1 字符设备驱动程序

字符设备是一种能像字节流一样进行串行访问的设备, 对设备的存取只能按顺序按字节存取而不能随机访问。字符设备没有请求缓冲区, 必须按顺序执行所有的访问请求。应用程序对字符设备的访问是通过字符设备结点来完成的。字符设备是 Linux 中最简单的设备, 可以像文件一样访问。应用程序使用标准系统调用打开、读、写和关闭字符设备, 完全可以把它们当做普通文件一样进行操作, 甚至被 PPP 守护进程使用, 用于将一个 Linux 系统连接到网上的 modem, 也被看做一个普通文件。当字符设备初始化时, 它的设备驱动程序向 Linux 内核注册, 向 `chrdevs` 向量表中增加一个 `device_struct` 数据结构项。通常一种类型设备的主设备标识符是固定的。设备的主设备标识符(例如对于 `tty` 设备是 4), 用作该向量表的索引。`chrdevs` 向量表中的每一项, 即 `device_struct` 数据结构, 包括两个元素: 一个是指向登记的设备驱动程序名字的指针; 另一个是指向一组文件操作的指针。这组文件操作本身位于这个设备的字符设备驱动程序中, 每一个都处理一个特定的文件操作, 如打开、读、写和关闭。常见的字符设备有鼠标、键盘、串口、控制台等。

用户进程通过设备文件对硬件进行访问, 对设备文件的操作方式通过一些系统调用来实现, 如 `open`、`read`、`write` 和 `close` 等。下面通过一个关键的数据结构 `file_operations`, 将系统调用和驱动程序关联起来。

```

struct file_operations {
    int (*seek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char, int);
    int (*write) (struct inode *, struct file *, off_t, int);
    int (*readdir) (struct inode *, struct file *, struct dirent *, int);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (struct inode *, struct file *);
    int (*revalidate) (dev_t dev);
};


```

该结构中每一个成员的名字都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如 read/write 操作时，系统调用根据设备文件的主设备号找到对应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

编写驱动程序就是针对上面相应的函数编写具体的实现，然后将它们对应上。编写完驱动后，把驱动程序嵌入内核。驱动程序可以采用两种方式进行编译。一种是编译进内核，驱动被静态加载；另一种是编译成模块（modules），驱动模块需要动态加载。在模块被调入内存时，init()函数向系统的字符设备表登记了一个字符设备：

```

int __init chr_dev_init(void)
{
    if (devfs_register_chrdev(CHR_MAJOR, "chr_name", &chr_fops))
        printk("unable to get major %d for chr devs\n", MEM_MAJOR);
    ...
    return 0;
}

```

当 cleanup\_chr\_dev() 函数被调用时，它释放字符设备 chr\_name 在系统字符设备表中占有的表项。

```

void cleanup_chr_dev(void)
{
    unregister_chrdev(CHR_MAJOR, "chr_name");
}

```

## 1.5.2 块设备驱动程序

块设备具有请求缓冲区，从块设备读取数据时，可以从任意位置读取任意长度，即块设备支持随机访问而不必按照顺序存取数据。例如，可以先存取后面的数据，然后再存取前面的数据，字符设备则不能采用该方式存取数据。Linux 下的磁盘设备均为块设备，应用程序访问 Linux 下的块设备结点是通过文件系统及其高速缓存来访问块设备的，并非直接通过设备结点读写块设备上的数据。

块设备既可以用做普通的裸设备存放任意数据，也可以将块设备按某种文件系统类型

的格式进行格式化，然后读取块设备上的数据，读取时根据该文件系统类型的格式进行读取。无论使用哪种方式，访问设备上的数据都必须通过调用设备本身的操作方法实现。两者的区别在于前者直接调用块设备的操作方法，而后者则间接调用块设备的操作方法。常见的块设备有各种硬盘、flash 磁盘、RAM 磁盘等。

块设备也可以与字符设备 register\_chrdev、unregister\_chrdev()函数类似的方法进行设备的注册与释放。但是，字符设备的 register\_chrdev()函数使用一个 file\_operations 结构的指针，而块设备的 register\_blkdev()函数则使用 block\_device\_operations 结构的指针，其中定义的 open、release 和 ioctl 方法和字符设备的对应方法相同，但没有对 read 和 write 操作定义。这是因为所有涉及块设备的 I/O 通常由系统进行缓冲处理。

块驱动程序最终必须提供完成实际块 I/O 操作的机制，在 Linux 中，用于这些 I/O 操作的方法称为 request（请求）。注册块设备时，需要对 request 队列进行初始化，这一动作通过 blk\_init\_queue 来完成，blk\_init\_queue 函数创建队列，并将该驱动程序的 request 函数关联到队列。在模块的清除阶段，应调用 blk\_cleanup\_queue 函数。

初始化块设备的时候，将块设备注册到内核中，下面为块设备的注册函数 mtdblock\_release()的实现：

```

01 int register_blkdev(unsigned int major, const char *name)
02 {
03     struct blk_major_name **n, *p;
04     int index, ret = 0;
05     mutex_lock(&block_class_lock);
06     /*为块设备指定主设备号，如果指定为 0 则表示由系统来分配*/
07     if (major == 0) {
08         for (index = ARRAY_SIZE(major_names)-1; index > 0; index--) {
09             if (major_names[index] == NULL)
10                 break;
11         }
12         if (index == 0) {
13             printk("register_blkdev: failed to get major for %s\n",
14                   name);
15             ret = -EBUSY;
16             goto out;
17         }
18         major = index;
19         ret = major;
20     }
21     /*为块设备名字分配空间*/
22     p = kmalloc(sizeof(struct blk_major_name), GFP_KERNEL);
23     if (p == NULL) {
24         ret = -ENOMEM;
25         goto out;
26     }
27     p->major = major;
28     strlcpy(p->name, name, sizeof(p->name));
29     p->next = NULL;
30     index = major_to_index(major);
31     for (n = &major_names[index]; *n; n = &(*n)->next) {
32         if ((*n)->major == major)
33             break;
34     }
35     if (!*n)
36         *n = p;

```

```

37     else
38         ret = -EBUSY;
39     if (ret < 0) {
40         printk("register_blkdev: cannot get major %d for %s\n",
41               major, name);
42         kfree(p);
43     }
44 out:
45     mutex_unlock(&block_class_lock);
46     return ret;
47 }

```

块设备被注册到系统后，访问硬件的操作 `open` 和 `release` 等就能够被对应的系统调用指针所绑定，应用程序使用系统调用就可以对硬件进行访问了。下面是块设备主要的操作函数 `open()` 和 `release()`。

下面为块设备 `open()` 操作函数。

```

01 static int mtdblock_open(struct mtd_blktrans_dev *mbd)
02 {
03     struct mtdblk_dev *mtdblk;
04     struct mtd_info *mtd = mbd->mtd;
05     int dev = mbd->devnum;
06     DEBUG(MTD_DEBUG_LEVEL1, "mtdblock_open\n");
07     if (mtdblks[dev]) {
08         /*如果设备已经打开，则只需要增加其引用计数*/
09         mtdblks[dev]->count++;
10         return 0;
11     }
12     /*为设备创建 mtdblk_dev 对象保存 mtd 设备的信息*/
13     mtdblk = kzalloc(sizeof(struct mtdblk_dev), GFP_KERNEL);
14     if (!mtdblk)
15         return -ENOMEM;
16     mtdblk->count = 1;
17     mtdblk->mtd = mtd;
18     mutex_init(&mtdblk->cache_mutex);
19     mtdblk->cache_state = STATE_EMPTY;
20     if (!(mtdblk->mtd->flags & MTD_NO_ERASE) && mtdblk->mtd->erasesize)
21     {
22         mtdblk->cache_size = mtdblk->mtd->erasesize;
23         mtdblk->cache_data = NULL;
24     }
25     mtdblks[dev] = mtdblk;
26     DEBUG(MTD_DEBUG_LEVEL1, "ok\n");
27     return 0;
28 }

```

释放时递减用户计数，当用户计数递减为 0 时，释放缓存中的数据，并释放为设备分配的空间。

```

01 static int mtdblock_release(struct mtd_blktrans_dev *mbd)
02 {
03     int dev = mbd->devnum;
04     struct mtdblk_dev *mtdblk = mtdblks[dev];
05     DEBUG(MTD_DEBUG_LEVEL1, "mtdblock_release\n");
06     mutex_lock(&mtdblk->cache_mutex);
07     write_cached_data(mtdblk);
08     mutex_unlock(&mtdblk->cache_mutex);
09     if (!--mtdblk->count) {

```

```

10     /*用户计数递减为0时释放设备*/
11     mtdblk->mtd->sync);
12     mtdblk->mtd->sync(mtdblk->mtd);
13     vfree(mtdblk->cache_data);
14     kfree(mtdblk);
15 }
16 DEBUG(MTD_DEBUG_LEVEL1, "ok\n");
17 return 0;
18 }
19 }
```

### 1.5.3 网络设备驱动程序

网络设备与字符设备的区别是，网络设备是面向数据报文的，而字符设备是面向字符流的。网络设备与块设备的区别是，网络设备不支持随机访问，也没有请求缓冲区。在Linux里网络设备也可以被称为网络接口，如eth0，应用程序是通过Socket（套接字），而不是设备结点来访问网络设备，在系统中不存在网络设备结点。

网络设备用来与其他设备交换数据，它可以是硬件设备，也可以是纯软件设备，如loopback接口。网络设备由内核中的网络子系统驱动，负责发送和接收数据包，但它不需要了解每项事务如何映射到实际传送的数据包。许多网络连接（例如使用TCP协议的连接）是面向流的，但网络设备围绕数据包的传输和接收设计。网络驱动程序不需要知道各个连接的相关信息，它只需处理数据包。字符设备和块设备都有设备号，而网络设备没有设备号，只有一个独一无二的名字，例如eth0、eth1等，这个名字也无须与设备文件结点对应。内核利用一组数据包传输函数与网络设备驱动程序进行通信，它们不同于字符设备和块设备的read()和write()方法。

Linux网络设备驱动程序从下到上分为4层，依次为网络设备与媒介层、设备驱动功能层、网络设备接口层和网络协议接口层。在设计具体的网络设备驱动程序时，需要完成的主要工作是编写设备驱动功能层的相关函数以填充net\_device数据结构的内容，并将net\_device注册入内核。

下面以DM9000代码为例说明网络设备驱动的注册、注销等主要过程。其驱动的注册过程在设备初始化时被调用。

```

01 static int __init dm9000_init(void)
02 {
03     printk(KERN_INFO "%s Ethernet Driver, %s\n", CARDNAME,
04            DRV_VERSION);
05     return platform_driver_register(&dm9000_driver);
06 }
```

驱动注册函数。

```

01 int platform_driver_register(struct platform_driver *drv)
02 {
03     drv->driver.bus = &platform_bus_type;
04     if (drv->probe)
05         drv->driver.probe = platform_drv_probe;
06     if (drv->remove)
07         drv->driver.remove = platform_drv_remove;
08     if (drv->shutdown)
```

```

09         drv->driver.shutdown = platform_drv_shutdown;
10     if (drv->suspend)
11         drv->driver.suspend = platform_drv_suspend;
12     if (drv->resume)
13         drv->driver.resume = platform_drv_resume;
14     return driver_register(&drv->driver);
15 }

```

在网络设备被清除时调用注销网络设备驱动函数。

```

01 static void __exit dm9000_cleanup(void)
02 {
03     platform_driver_unregister(&dm9000_driver);
04 }

```

驱动注销过程。驱动注销的过程中还包括将设备从系统中移除和将驱动从总线上移植。

```

01 void platform_driver_unregister(struct platform_driver *drv)
02 {
03     driver_unregister(&drv->driver);
04 }
05 void driver_unregister(struct device_driver *drv)
06 {
07     driver_remove_groups(drv, drv->groups);
08     bus_remove_driver(drv);
09 }
10 static void driver_remove_groups(struct device_driver *drv, struct
attribute_group **groups)
11 {
12     int i;
13     if (groups)
14         for (i = 0; groups[i]; i++)
15             sysfs_remove_group(&drv->p->kobj, groups[i]);
16 }
17 void bus_remove_driver(struct device_driver *drv)
18 {
19     if (!drv->bus)
20         return;
21     remove_bind_files(drv);
22     driver_remove_attrs(drv->bus, drv);
23     driver_remove_file(drv, &driver_attr_uevent);
24     klist_remove(&drv->p->knode_bus);
25     pr_debug("bus: '%s': remove driver %s\n", drv->bus->name,
drv->name);
26     driver_detach(drv);
27     module_remove_driver(drv);
28     kobject_put(&drv->p->kobj);
29     bus_put(drv->bus);
}

```

有关网络设备驱动的详细接口函数解析和驱动移植将在后面的章节中叙述。

#### 1.5.4 内存与 I/O 操作

一般来说，在系统运行时，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址，预定义虚拟地址

范围，驱动程序并不能直接通过物理地址访问 I/O 内存资源，只能先将它们映射到内核的虚拟地址空间内（通过页表），然后才能根据映射的内核虚拟地址范围，通过访内指令访问这些 I/O 内存资源。Linux 在 io.h 头文件中声明了函数 ioremap()，用来将 I/O 内存资源的物理地址映射到核心虚拟地址空间（3GB~4GB）中，原型如下：

```
void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags);
```

iounmap()函数用于取消 ioremap()所做的映射，原型如下：

```
void iounmap(void * addr);
```

在将 I/O 内存资源的物理地址映射成内核的虚拟地址后，理论上可以像读写 RAM 那样直接读写 I/O 内存资源了。为了保证驱动程序跨平台的可移植性，应该使用 Linux 中特定的函数访问 I/O 内存资源，而不应该通过指向内核虚拟地址的指针来访问。如在 ARM 平台上，读写 I/O 的函数如下：

```
#define __raw_base_writeb(val,base,off) __arch_base_putb(val,base,off)
#define __raw_base_writew(val,base,off) __arch_base_putw(val,base,off)
#define __raw_base_writel(val,base,off) __arch_base_putl(val,base,off)

#define __raw_base_readb(base,off) __arch_base_getb(base,off)
#define __raw_base_readw(base,off) __arch_base_getw(base,off)
#define __raw_base_readl(base,off) __arch_base_getl(base,off)
```

驱动程序中 mmap()函数的实现原理是，用 mmap 映射一个设备，表示将用户空间的一段地址关联到设备内存上，这样当程序在分配的地址范围内进行读取或者写入时，实际上就是对设备的访问。这一映射原理类似于 Linux 下 mount 命令，将一种类型的文件系统或设备挂载到另外一个文件系统或者目录下时，挂载成功后，对挂载点的任何操作实际上是对被挂载的文件系统和设备的操作。

## 1.6 小 结

Linux 内核是一个比较庞大的系统，深入理解内核可以减少在系统移植中的障碍。在系统移植中设备驱动开发是一项很复杂的工作，由于 Linux 内核提供了一部分源代码，同时还提供了对某些公共部分的支持，例如，USB 驱动对读写 U 盘、键盘、鼠标等设备提供了通用驱动程序，一般情况可以直接使用内核提供的驱动。但是对于复杂的 USB 设备没有现成的驱动，就需要读者对驱动开发过程有一定的认识，必要时参考 Linux 源码重新开发驱动程序。

# 第 2 章 嵌入式 Linux 开发环境搭建

进行嵌入式项目开发，需要建立嵌入式开发环境。建立嵌入式 Linux 开发环境包括安装 Bootloader 工具；针对不同平台的交叉编译器（在本书中都是针对 ARM 平台）arm-linux-gcc；需要编译配置内核时还要安装内核源码树；在调试时使用的一些终端软件、TFTP 软件、FTP 软件，有内核和文件系统的烧写工具，一般硬件厂家会提供这样的工具。本章主要介绍嵌入式 Linux 系统移植过程中用到的交叉编译环境建立，以及各种工具的安装和配置。

## 2.1 虚拟机及 Linux 安装

很多工具都是 Windows 版本的，而要求的开发环境是 Linux 环境。在 Windows 系统中安装虚拟机，然后再虚拟一个 Linux 环境，使 Linux 和 Windows 能够互相通信。这种方案解决了很多软件不兼容两种平台的问题。

### 2.1.1 虚拟机的安装

虚拟机软件 Vmware 的安装和普通软件的安装过程一样，此处就不详细介绍了。这里主要介绍在虚拟机中安装 Linux 系统的过程。正确安装 VMware 后启动时的界面如图 2.1 所示。

下面介绍安装 Linux 的主要步骤，省略了一些只需要单击“下一步”按钮的步骤。

- (1) 准备安装系统软件 Fedora-Live-Desktop-i686-19-1.iso，可以在网上下载。
- (2) 运行 VMware，选择“文件”|“新建虚拟机”命令，或者直接单击“新建虚拟机”图标。
- (3) 在“新建虚拟机向导”窗口中选择“自定义”选项，如图 2.2 所示。
- (4) 虚拟机硬件兼容性窗口按默认选择安装。
- (5) 在“安装客户机操作系统”窗口中，选择“我以后再安装操作系统”选项。
- (6) 在选择客户机操作系统的时候选择 Linux 选项。在“版本”下拉菜单中选择准备安装的 Linux 版本。这里使用的是 Fedora 选项，如图 2.3 所示。
- (7) 单击“下一步”按钮，进入主题为虚拟机取名称的对话框。在此对话框中为安装的虚拟机取名字，同时确定安装路径，注意选择安装的分区应该有足够的空间安装 Linux 系统。因为在后面还要安装 Linux 源码树，所以建议安装在一个有 8GB~10GB 空闲空间的分区上。
- (8) 单击“下一步”按钮，进入处理器配置对话框，在其中根据实际情况选择处理器

的个数。

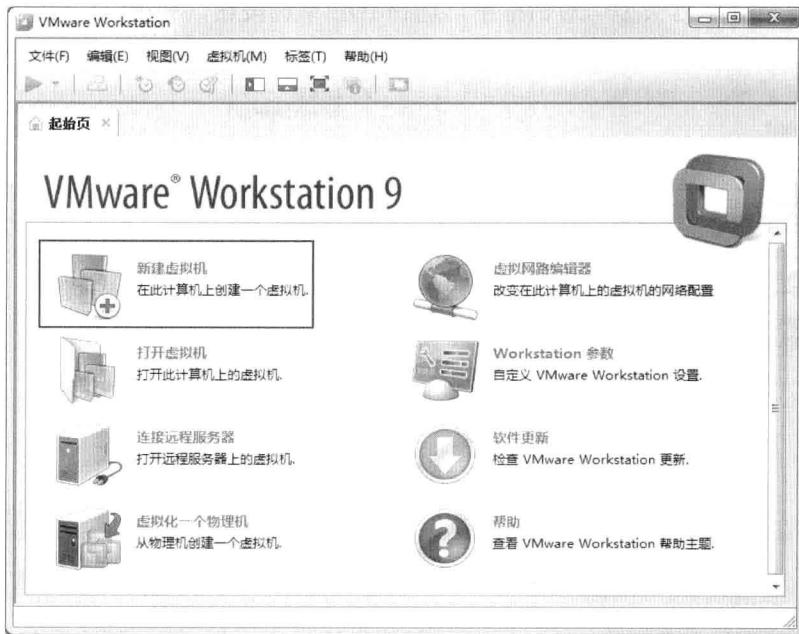


图 2.1 VMware 启动界面

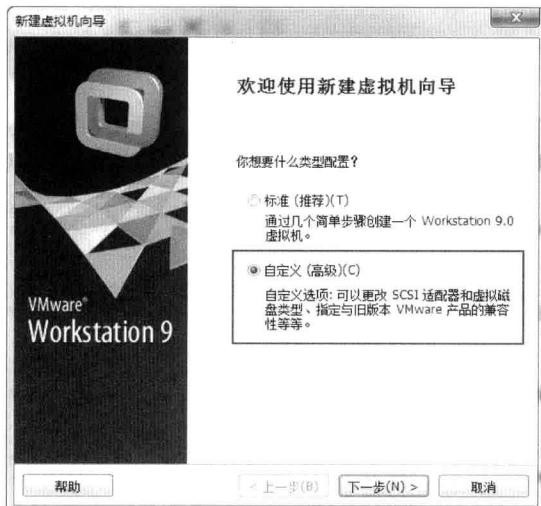


图 2.2 虚拟机配置

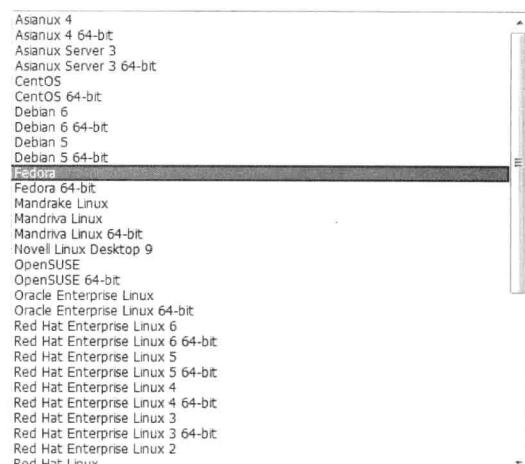


图 2.3 Linux 安装版本选择

(9) 在对虚拟机内存进行划分时，可以根据实际主机硬件的配置进行划分，一般可以按默认的配置安装。如果实际主机配置比较高，可以给虚拟机多分配点内存。

(10) 在网络连接类型中可以选择任意一种类型，该设置在后面需要修改的时候可以进行修改，此处可以按默认选项进行安装。

(11) I/O适配器窗口可以按照默认配置安装。

(12) 在“选择磁盘”对话框中，选择“创建一个新的虚拟磁盘”单选按钮，如图 2.4

所示。

(13) 在“选择磁盘类型”对话框中，选择 IDE 单选按钮，如图 2.5 所示。

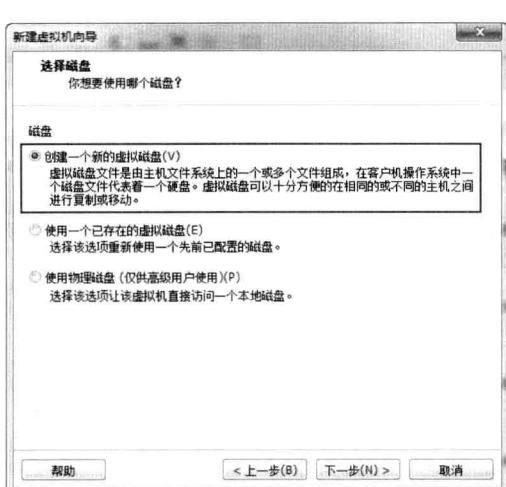


图 2.4 选择创建一个新的虚拟磁盘



图 2.5 虚拟磁盘类型

(14) 在“指定磁盘容量”对话框中，确定磁盘大小，为 Linux 系统预留 20GB 空间，并且选择“立即分配所有磁盘空间”复选框，如图 2.6 所示。

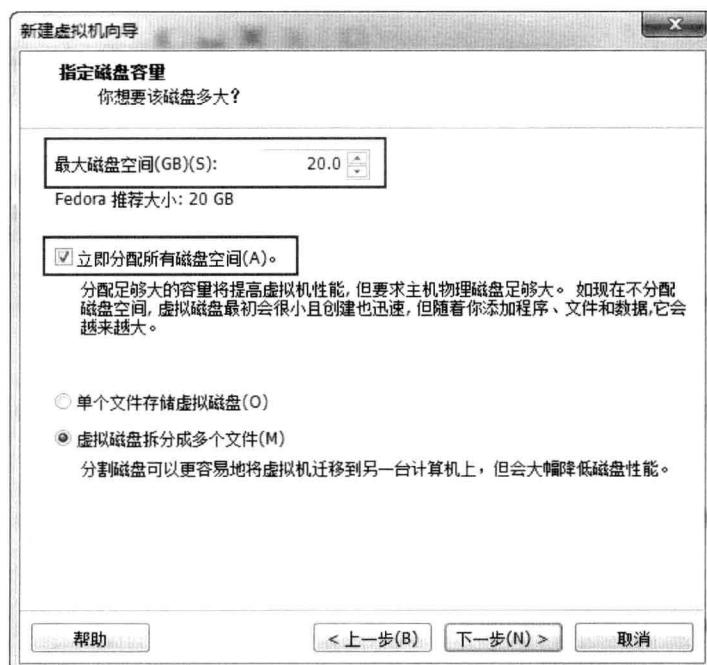


图 2.6 指定磁盘容量

(15) 指定磁盘文件对话框按默认名字和路径进行安装，单击“下一步”按钮，确认虚拟机设置。然后单击“完成”按钮，创建虚拟磁盘。结果如图 2.7 所示。

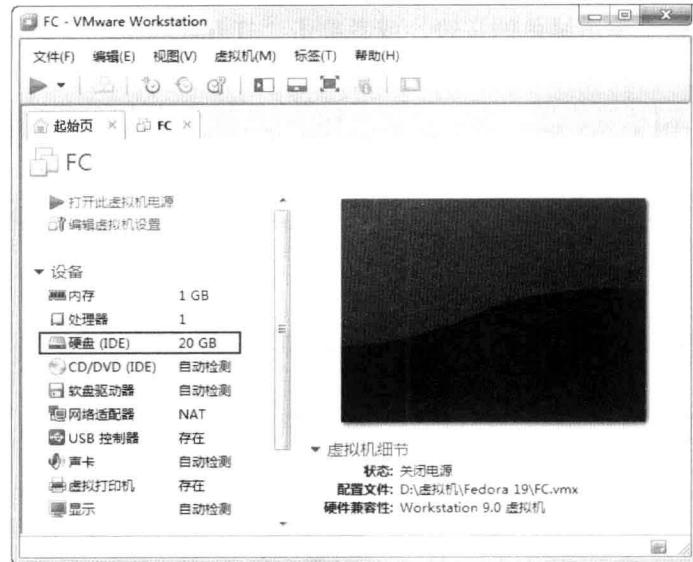


图 2.7 创建虚拟机

(16) 单击“CD/DVD(IDE) 自动检测”选项，打开“虚拟机设置”对话框，如图 2.8 所示。

(17) 选择“使用 ISO 镜像文件”选项，然后单击“浏览”按钮，选择下载好的 Fedora 镜像文件。然后单击“确定”按钮，关闭对话框。



图 2.8 “虚拟机设置”对话框

(18) 在工具栏中单击绿色的按钮，或者单击“打开此虚拟机电源”选项，如图 2.9 所示。

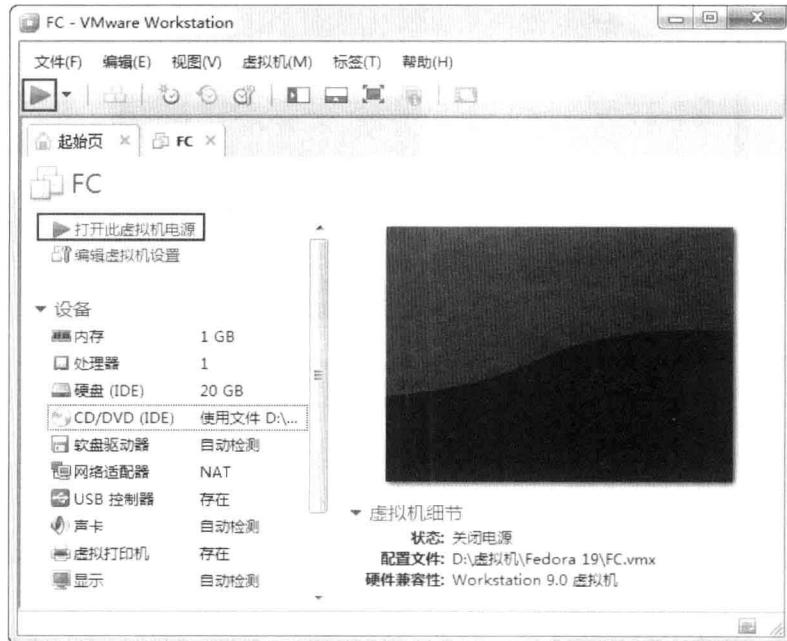


图 2.9 启动虚拟机

(19) 进入虚拟机安装界面，如图 2.10 所示。用户不需要操作，系统会自动加载光盘，进入 Fedora Live 界面，如图 2.11 所示。

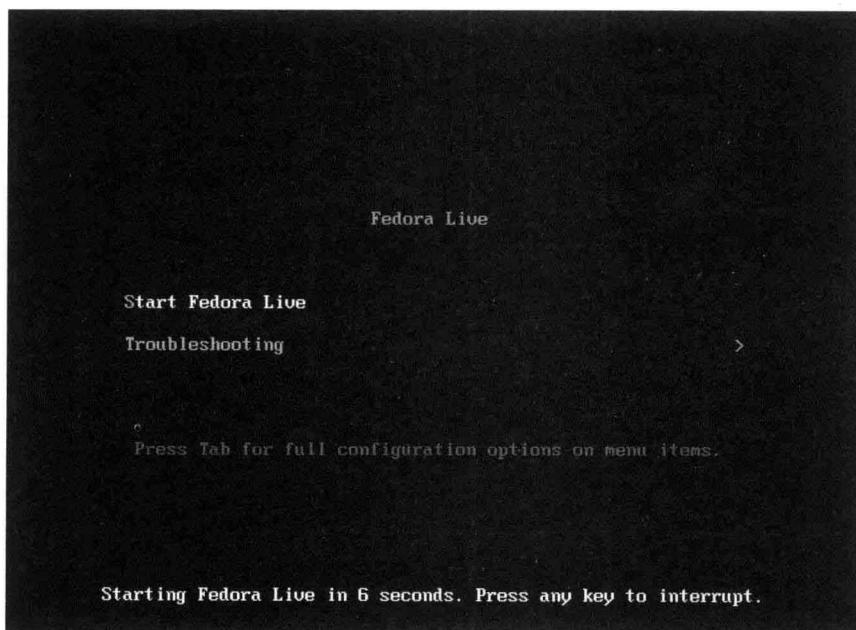


图 2.10 Fedora 启动界面

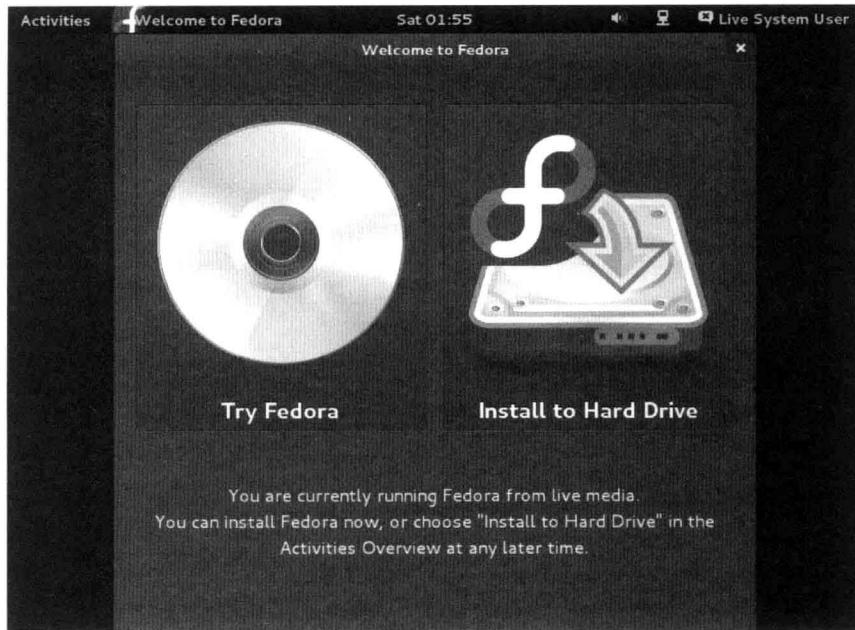


图 2.11 安装开始界面

- (20) 单击右侧的 Install to Hard Drive 文字，进入 Fedora 安装界面。
- (21) 在选择语言选项的时候，建议选择“简体中文”选项。然后单击“继续”按钮，进入“安装信息摘要”界面，如图 2.12 所示。



图 2.12 “安装信息摘要”界面

- (22) 在该界面提供本地化、软件、存储、键盘四项。依次确认或设置后，“开始安

装”按钮就不再是灰色。然后单击“开始安装”按钮，进入“配置”界面，如图 2.13 所示。



图 2.13 “配置”界面

(23) 进入该界面后，系统就默认开始安装。同时，该界面提供 ROOT 密码设置和创建用户功能。依次设置后，等待 Fedora 自动安装。安装完成后，系统提示重新启动计算机，如图 2.14 所示。



图 2.14 安装完成

(24) 在VMware软件中，选择“虚拟机”|“发送Ctrl+Alt+Del”命令，重新启动Fedora操作系统，进入Fedora启动界面，如图2.15所示。

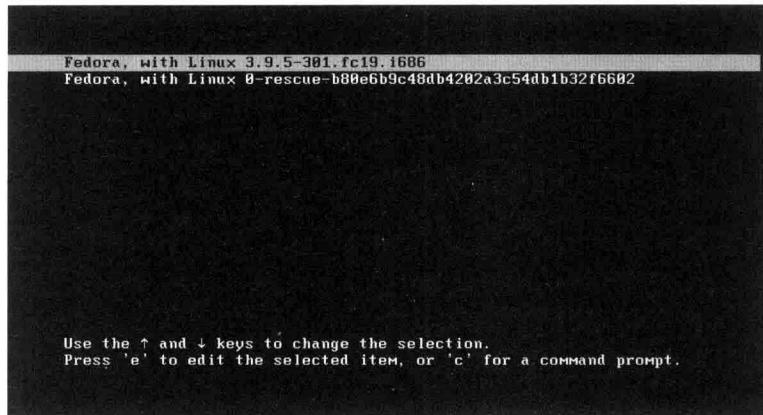


图2.15 Fedora启动界面

(25) 选择第一项，按回车键后，进入系统。由于是第一次登录，需要对语言、输入源、在线账号做一个简单设置。设置完成后，进入操作系统桌面，如图2.16所示。



图2.16 进入Fedora桌面

**注意：**在设置账号密码的时候，如果密码安全等级较低，系统会显示警告信息。如果确定使用该密码，就需要单击两次左上角的“完成”按钮，强制密码生效。

## 2.1.2 虚拟机和主机通信设置

很多资料和软件往往都放在主机上，当需要在虚拟机环境下对这些资料进行访问时，

或者将虚拟机编译好的文件传送到主机上时，就需要建立两者之间的通信。建立虚拟机和主机通信的过程如下所述。

(1) 选择“虚拟机”|“设置”命令，打开“虚拟机设置”对话框。选择“硬件”选项卡中的“网络适配器”选项。在“网络连接”选项区域中选择“自定义：指定虚拟网络”单选按钮，在下拉列表框中设置网络连接为VMnet8(NAT)。然后单击“确定”按钮，如图2.17所示。

(2) 选择“编辑”|“虚拟网络编辑器”命令，进入“虚拟网络编辑器”对话框。选择NAT标签，将VMnet host设置为VMnet8。然后单击“确定”按钮，如图2.18所示。

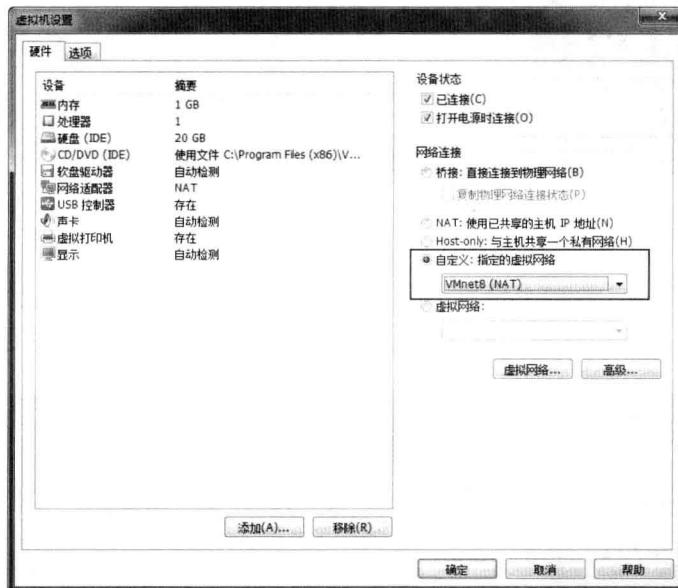


图2.17 “虚拟机设置”对话框



图2.18 “虚拟网络编辑器”对话框

(3) 在该对话框中，显示NAT子网的IP和子网掩码分别为192.168.81.0和255.255.255.0。

(4) 设置网卡连接状态。双击虚拟机窗口右下角的网络适配器标志，如图2.19所示。在“设备状态”选项框中，选择“已连接”和“打开电源时连接”复选框，如图2.20所示。

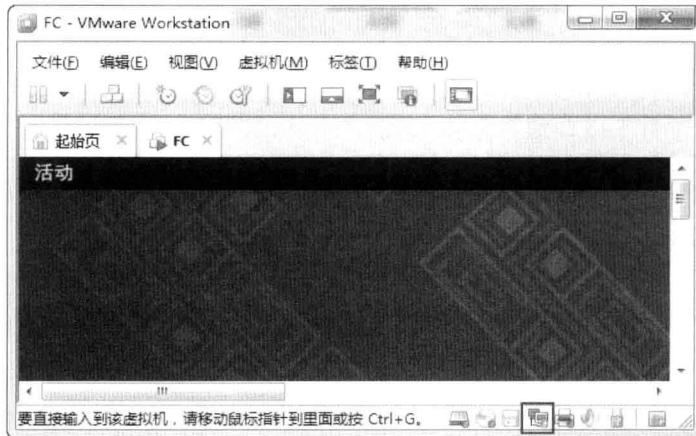


图2.19 双击网卡标志



图2.20 设置网卡状态

(5) 打开一个终端，通过ifconfig查看虚拟机的IP地址，如图2.21所示。图中显示网卡地址为192.168.81.136。

```
test@localhost:~$ ifconfig
lo: flags=7 mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        inet6 fe80::2c:9ff febc:b94 prefixlen 64 scopeid 0x20<link>
            ether 00:0c:29:bc:0b:94 txqueuelen 1000 (Ethernet)
            RX packets 1456 bytes 121776 (118.9 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 1456 bytes 121776 (118.9 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

p3p1: flags=4163 mtu 1500
    inet 192.168.81.136 netmask 255.255.255.0 broadcast 192.168.81.255
        inet6 fe80::20c:29ff febc:b94 prefixlen 64 scopeid 0x20<link>
            ether 00:0c:29:bc:0b:94 txqueuelen 1000 (Ethernet)
            RX packets 88551 bytes 130783074 (124.7 MiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 52956 bytes 2923365 (2.7 MiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
            device interrupt 19 base 0x2000

test@localhost:~$
```

图 2.21 虚拟机 IP 地址

(6) 确定主机的 VMware net8 网卡为已连接状态，在主机端 ping 虚拟机，结果如图 2.22 所示。查看主机 VMware net8 网卡的 IP 地址，一般和虚拟机网关为一个网段，并且其 IP 地址为 192.168.81.1。在虚拟机中 ping 该 IP，结果如图 2.23 所示。互相都可以 ping 通说明主机和虚拟机通信成功。

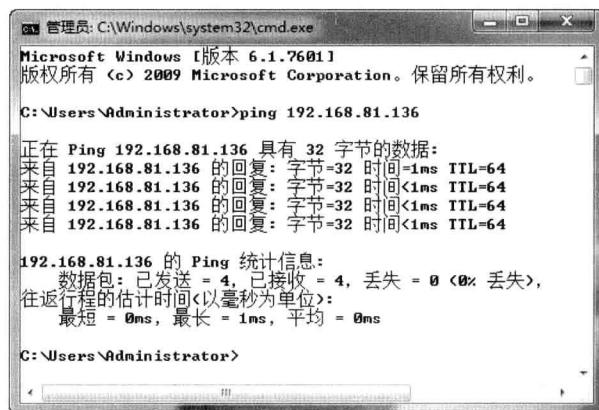


图 2.22 主机 ping 虚拟机

```
test@localhost:~$ ping 192.168.81.1
PING 192.168.81.1 (192.168.81.1) 56(84) bytes of data.
64 bytes from 192.168.81.1: icmp_seq=1 ttl=128 time=0.602 ms
64 bytes from 192.168.81.1: icmp_seq=2 ttl=128 time=0.304 ms
64 bytes from 192.168.81.1: icmp_seq=3 ttl=128 time=0.277 ms
64 bytes from 192.168.81.1: icmp_seq=4 ttl=128 time=0.280 ms
^C
--- 192.168.81.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 0.277/0.365/0.602/0.138 ms
test@localhost:~$
```

图 2.23 虚拟机 ping 主机

### 2.1.3 虚拟机与主机共享文件

设置文件共享后，能够在主机和虚拟机之间进行文件传输。

(1) 选择“虚拟机”|“设置”命令打开虚拟机设置（Virtual Machine Setting）对话框。选择“选项”标签，在其中选择“共享文件夹”选项。在“文件夹共享”选项框选择“总是启用”单选按钮，如图 2.24 所示。

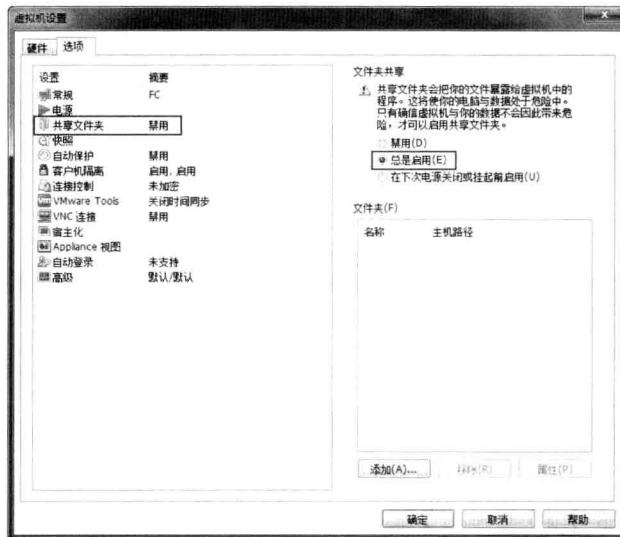


图 2.24 设置共享文件夹

(2) 单击“添加”按钮，弹出“添加共享文件夹向导”对话框。单击“下一步”按钮，弹出“共享文件夹名称”对话框，如图 2.25 所示。



图 2.25 “共享文件夹名称”对话框

(3) 设置完成后，单击“下一步”按钮，进入“指定共享文件夹属性”对话框，如图 2.26 所示。选择“启用该共享”选项，然后单击“完成”按钮，保存设置。进入 /mnt 目录下，会发现多了一个目录 share。进入 share，可以看到在 Windows 系统下的文件。



图 2.26 “指定共享文件夹属性”对话框

#### 2.1.4 虚拟机与主机文件传输

某些版本的虚拟机或者 Linux 系统对文件共享支持不够完美，此时可以选择 FTP 方式进行文件传输，该方法操作方便，在实际开发中被普遍使用。该方法包括服务器端（虚拟机）和客户端（主机）两部分安装，并且包括服务器端和客户端的配置。下面将分别进行介绍。

(1) 安装 vsftpd 软件。Fedora 默认没有安装 FTP，需要用户手动安装。在“软件”程序中，搜索 vsftpd，并安装，如图 2.27 所示。



图 2.27 “软件”对话框

(2) 启动FTP服务。在终端中，输入以下命令启动vsftpd服务。

```
service vsftpd start
Redirecting to /bin/systemctl start vsftpd.service
```

(3) 在防火墙中设置FTP策略，使得实体机可以访问虚拟机中的FTP服务，设置如图2.28所示。



图2.28 “防火墙配置”对话框

(4) 在实体机中使用IE等客户端工具，访问FTP服务器，就可以看到FTP所共享的文件，如图2.29所示。

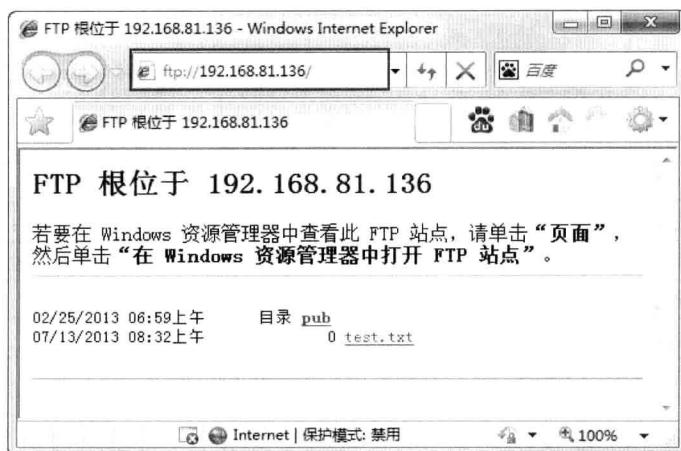


图2.29 实体机访问虚拟机的FTP服务

## 2.2 交叉编译工具

交叉编译工具是为了使在上位机中编译的文件能够在不同平台的目标机中执行。本书介绍的目标平台均为 ARM 平台。

### 2.2.1 交叉编译工具安装

交叉编译工具的安装主要包括编译 GNU binutils、获得 Linux 内核头文件、安装 Glibc 头文件、安装 GCC 第一阶段、安装 Glibc 和完全安装 GCC。一次成功安装的过程需要长达数小时的时间，因此在没有成功编译此工具经验时，可以参考一些稳定交叉编译器的版本。下面将详细介绍安装的过程，并提供相关的命令和解释供编译时参考。

(1) 建立存放工具、源码目录和设置相关的环境变量。

```
#mkdir /usr/local/arm                                //建立工作目录
#cd /usr/local/arm                                  //进入工作目录
#mkdir -p /usr/local/arm/src                         //建立安装文件的目录
#mkdir -p /usr/local/arm/sysroot                     //建立系统根目录
#mkdir -p /usr/local/arm/bin                         //建立生成工具存放的目录
#mkdir -p /usr/local/arm/build                       //建立编译目录
#export HOST=i686-pc-linux-gnu                      //设置 HOST 环境变量
#export TARGET=arm-linux-gnueabi                     //设置 TARGET 环境变量
#export PREFIX=/usr/local/arm                         //设置 PREFIX 环境变量
#export SYSROOT=/usr/local/arm/sysroot                //设置 SYSROOT 环境变量
#export PATH=$PATH:${PREFIX}/bin                      //添加生成工具目录到 PATH 中
```

 注意：在定义好环境变量后，应该使用 echo 命令查看环境变量是否与预计的变量相符合。在重新打开 shell 时，或者在第二次接着编译交叉环境时都应该检查环境变量。例如当 PATH 没有设置正确时，在编译 glibc 时，就会因 arm-liunx-gcc 不存在而导致编译失败。

(2) 从网站下载安装的源码包，需要下载的源码包如表 2.1 所示。

表 2.1 源码包列表

软件包名称	下 载 地 址
binutils-2.23.tar.bz2	<a href="http://ftp.gnu.org/gnu/binutils/">http://ftp.gnu.org/gnu/binutils/</a>
linux-2.6.33.3.tar.bz2	<a href="http://www.kernel.org/pub/linux/kernel/v2.6/">http://www.kernel.org/pub/linux/kernel/v2.6/</a>
gcc-4.6.0.tar.bz2	<a href="http://ftp.gnu.org/gnu/gcc/gcc-4.6.0/">http://ftp.gnu.org/gnu/gcc/gcc-4.6.0/</a>
glibc-2.17.tar.gz	<a href="http://ftp.gnu.org/gnu/glibc/">http://ftp.gnu.org/gnu/glibc/</a>
glibc-linuxthreads-2.3.6.tar.gz	<a href="http://ftp.gnu.org/gnu/glibc/">http://ftp.gnu.org/gnu/glibc/</a>

将获得的源码 mkdir 复制到/usr/local/arm/src 目录下。目录结构如下：

```
|-- arm
|   |-- bin      //工具目录
```

```

| |-- build      //编译目录
| |-- src        //存放源码 cd
| |-- sysroot    //编译过程生成系统的根目录

```

### (3) 编译 GNU binutils。

```

#cd ${PREFIX}/src
#tar xvfj binutils-2.23.tar.bz2
#mkdir -p /usr/local/arm/build/binutils-2.23
#cd /usr/local/arm/build/binutils-2.23
#../src/binutils-2.23/configure --prefix=${PREFIX} --target=${TARGET}
--with-sysroot=${SYSROOT} 2>&1 | tee configure.out
#make 2>&1 | tee make.out
#make install 2>&1 | tee -a make.out

```

编译选项注释：

--target=\${TARGET}

这个选项是跟--host一起表示编译生成的可执行文件运行在HOST上面，但这些可执行文件服务的对象是TARGET，也就是说，用这些可执行文件连接和汇编出来的程序运行在TARGET上面。这里，默认就会使用主机的GCC编译器，因此省略了--host选项。

--prefix=\${RESULT\_DIR}

告诉配置脚本当运行 make install 时，把编译好的东西安装在RESULT\_DIR目录下。

--with-sysroot=\${SYSROOT}

SYSROOT为系统根目录，生成相应的库放在SYSROOT/lib目录下，可执行文件放在SYSROOT/sbin下，配置文件放在SYSROOT/etc下，用户文件放在SYSROOT/usr下。

在上面的编译过程中如果出现问题，最好的方法就是，删除编译目录下的所有文件，删除Binutils目录，重新解压Binutils，再重新开始安装。安装成功后就会在/usr/local/arm/bin目录下生成下面的工具：

arm-linux-addr2line	arm-linux-c++filt	arm-linux-nm	arm-linux-ranlib
arm-linux-strings			
arm-linux-ar	arm-linux-gprof	arm-linux-objcopy	arm-linux-readelf
arm-linux-strip			
arm-linux-as	arm-linux-ld	arm-linux-objdump	arm-linux-size

Binutils是GNU工具之一，包括连接器、汇编器和其他用于目标文件和档案的工具，它是二进制代码的处理维护工具。安装Binutils工具包含的程序有addr2line、ar、as、c++filt、gprof、ld、nm、objcopy、objdump、ranlib、readelf、size、strings、strip、libiberty、libbfd和libopcodes。对这些程序的简单解释如下所述。

- **addr2line:** 把程序地址转换为文件名和行号。在命令行中给它一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件及行号。
- **ar:** 用来建立、修改和提取归档文件。归档文件是包括了其他多个文件的一个较大的文件，从该文件中可以恢复其他文件内容。
- **as:** 主要用来编译GNU C编译器gcc输出的汇编文件，编译产生的目标文件再由连接器ld进行连接操作。

- **c++filt:** 连接器使用它来过滤 C++ 和 Java 符号，防止发生重载函数冲突。
- **gprof:** 显示程序调用段的各种数据。
- **ld:** 是连接器，它把所有编译产生的目标文件和归档文件结合在一起，重新定位数据，并连接符号引用。一般建立一个新编译程序的最后一步就是调用 ld。
- **nm:** 列出目标文件中的符号。
- **objcopy:** 把一种目标文件中的内容复制到另一种类型的目标文件中。
- **objdump:** 显示一个或者更多目标文件的信息。使用选项来控制其显示的信息，它所显示的信息通常只有编写编译工具的人才感兴趣。
- **ranlib:** 产生归档文件索引，并将其保存到这个归档文件中。在索引中列出了归档文件各成员所定义的可重分配目标文件。
- **readelf:** 显示 elf 格式可执行文件的信息。
- **size:** 列出目标文件每一段的大小及总体的大小。默认方式下，目标文件或者归档文件中的单个模块只产生一行输出。
- **strings:** 打印某个文件的可打印字符串，这些字符串最少 4 个字符长，也可以使用选项-n 设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其他类型的文件它打印整个文件的可打印字符。这个程序对于了解非文本文件的内容很有帮助。
- **strip:** 丢弃目标文件中的全部或者特定符号。
- **liberty:** 包含许多 GNU 程序都会用到的函数，这些程序有 getopt、obstack、strerror、strtol 和 strtoul。
- **libbfd:** 二进制文件描述库。
- **libopcode:** 用来处理 opcodes 的库，在生成一些应用程序的时候也会用到它。

(4) 获得 Linux 内核头文件，并将头文件安装在 \${SYSROOT}/usr/include 目录下。

```
#cd ${PREFIX}/src
#tar xvzf linux-2.6.33.3.tar.bz2
#cd linux-2.6.33.3/
#make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
#make include/linux/version.h
#mkdir -p ${SYSROOT}/usr/include
#cp -a ${PREFIX}/src/linux-2.6.33.3/include/linux ${SYSROOT}/usr/include/linux
#cp -a ${PREFIX}/src/linux-2.6.33.3/arch/arm/include/asm ${SYSROOT}/usr/include/asm
#cp -a ${PREFIX}/src/linux-2.6.33.3/include/asm-generic ${SYSROOT}/usr/include/asm-generic
```

 注意：执行 make ARCH=arm CROSS\_COMPILE=arm-linux- menuconfig 时，主要是为了选择一个对应 CPU 的类型。选择 System Type → ARM system type (ARM Ltd. Versatile family) → (X) Samsung S3C2410, S3C2412, S3C2413, S3C2440, S3C2442, S3C2443，如图 2.30 所示。

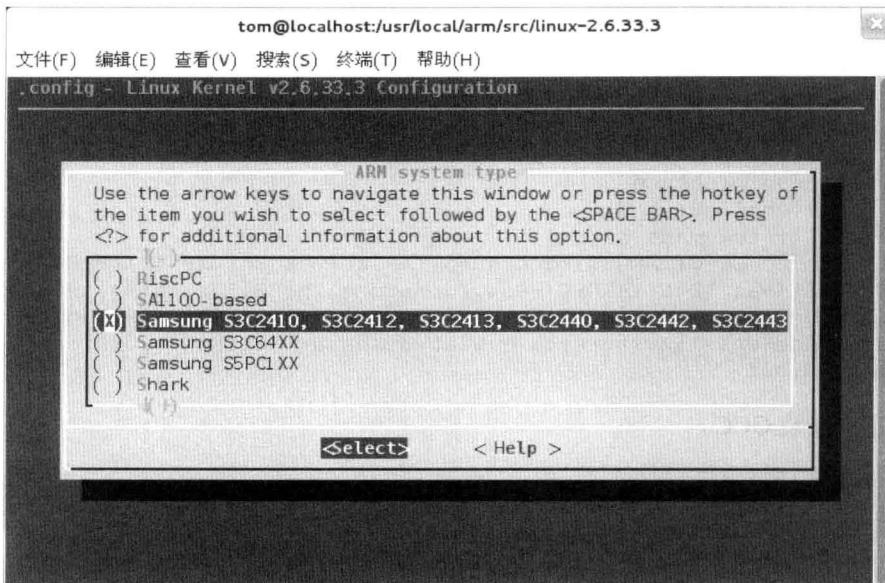


图 2.30 选择系统芯片类型

(5) 安装 Glibc 头文件。Glibc 是 GNU C 库，它是编译 Linux 系统程序的重要组成部分。在安装 GNU C 库前需要先安装其头文件。

```
#cd ${PREFIX}/src
#tar xvfz glibc-2.17.tar.gz
#cd glibc-2.17
#tar xvfz ./glibc-linuxthreads-2.3.6.tar.gz
#cd ..
mkdir -p /usr/local/arm/build/glibc-2.17-headers
#cd /usr/local/arm/build/glibc-2.17-headers
../../src/glibc-2.17/configure --prefix=/usr --host=${TARGET}
--enable-add-ons --with-headers=${SYSROOT}/usr/include 2>&1 | tee
configure.out
make cross-compiling=yes install_root=${SYSROOT} install-headers 2>&1 | tee
make.out
#touch ${SYSROOT}/usr/include/gnu/stubs.h
#touch ${SYSROOT}/usr/include/bits/stdio_lim.h
```

(6) 安装 GCC 第一阶段。安装 GCC 共分为两个阶段，第一阶段是为了安装 ARM 交叉编译工具没有支持 libc 库的头文件。

```
#cd ${PREFIX}/src
#tar xvfj gcc-4.6.0.tar.bz2
mkdir -p mkdir /usr/local/arm/build/gcc-4.6.0
#cd /usr/local/arm/build/gcc-4.6.0
../../src/gcc-4.6.0/configure --prefix=${PREFIX} --target=${TARGET}
--enable-languages=c --with-sysroot=${SYSROOT} 2>&1 | tee configure.out
make 2>&1 | tee make.out
make install 2>&1 | tee -a make.out
```

(7) 安装 GNU C 库。安装了 Glibc 后才能对 GCC 进行完全安装。

```
#cd ${PREFIX}/src
mkdir -p /usr/local/arm/build/glibc-2.17
#cd /usr/local/arm/build/glibc-2.17
```

```
#BUILD_CC=gcc CC=${TARGET}-gcc AR=${TARGET}-ar RANLIB=${TARGET}-ranlib
AS=${TARGET}-as LD=${TARGET}-ld ../../src/glibc-2.3.5/configure --prefix=
/usr
--build=i386-redhat-linux --host=arm-unknown-linux-gnu --target=arm-
unknown-linux-gnu
--without-_thread --enable-add-ons=linuxthreads --with-headers=
${SYSROOT}/usr/include
2>&1 | tee configure.out
#make 2>&1 | tee make.out
#BUILD_CC=gcc CC=${TARGET}-gcc AR=${TARGET}-ar RANLIB=${TARGET}-ranlib
AS=${TARGET}-as LD=${TARGET}-ld ../../src/glibc-2.17/configure --prefix=
/usr build=${HOST} --host=${TARGET} -target=${TARGET} --without-_thread
--enable-add-ons=linuxthreads --with-headers=${SYSROOT}/usr/include 2>&1
| tee make.out
```

### (8) 完全安装 GCC。

```
#cd ${PREFIX}/src
#mkdir -p /usr/local/arm/build/gcc-4.6.0
#cd /usr/local/arm/build/gcc-4.6.0
#../../src/gcc-4.6.0/configure --prefix=${PREFIX} --target=${TARGET}
--enable-languages=c --with-sysroot=${SYSROOT} 2>&1 | tee configure.out
#make 2>&1 | tee make.out
#make install 2>&1 | tee -a make.out
```

编译完成后会在/usr/local/arm/bin 目录下增加交叉编译工具，具体生成的工具如下：

arm-linux-addr2line	arm-linux-cpp	arm-linux-gcov	arm-linux-objdump
arm-linux-strings			
arm-linux-ar	arm-linux-gcc	arm-linux-ld	arm-linux-ranlib
arm-linux-strip			
arm-linux-as	arm-linux-gcc-4.4.0	arm-linux-nm	arm-linux-readelf
arm-linux-c++filt	arm-linux-gccbug	arm-linux-objcopy	arm-linux-size

### (9) 删除源码目录、临时目录和一些中间目录，得到 arm-linux、bin、lib、libexec 和 share 目录。

**▲注意：**编译交叉编译器是一个很耗时的工作，对于实际项目的作用并不大。除非在某些应用程序或者驱动模块已经通过测试进入成品库，而这些应用程序或驱动模块依赖某个版本 GCC 或 glibc，同时修改和测试应用程序或驱动模块的工作量相对非常复杂，此时可以选择需要的版本进行建立交叉编译环境。一般情况下，建议读者直接使用开发板厂商提供的交叉编译器，或者在网上下载稳定的交叉编译器。目前针对 2.4 内核的稳定版本为 2.95.3，针对 2.6 内核的稳定版本为 3.4.1。笔者目前还使用开发商提供的 4.4.3 版本。在后面的内核移植和驱动移植过程中使用的就是 4.4.3 版本。

## 2.2.2 交叉编译器测试

在使用新建立的交叉编译器前需要对其进行简单的测试，查看其生成的文件是否可以移植到 ARM 平台的开发板上运行。

(1) 对编译的交叉工具链进行简单的测试。将 arm-linux-gcc 添加到环境变量中。

```
# vi /etc/profile
```

找到# Path manipulation 部分，添加 arm-linux-gcc 所在目录。修改后保存配置重启系统配置生效。

```
# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
fi
```

修改如下：

```
# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
    pathmunge /usr/local/arm/bin
fi
```

(2) 编写简单的测试程序，查看程序适应的体系结构。

```
#include <stdio.h>
int main()
{
    printf("test arm-linux-gcc");
    return 0;
}
```

对上面的程序进行交叉编译，并查看其头文件信息，看其运行属于哪种平台体系结构。

```
#arm-linux-gcc -o test test.c          //交叉编译 test.c，生成 test
# readelf test -h                      //使用 readelf 命令查看头文件信息
ELF 头:
Magic: 7f 45 4c 46 01 01 00 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (可执行文件)
Machine: ARM
Version: 0x1
入口点地址: 0x8338
程序头起点: 52 (bytes into file)
Start of section headers: 4464 (bytes into file)
标志: 0x50000002, has entry point, Version5 EABI
本头的大小: 52 (字节)
程序头大小: 32 (字节)
Number of program headers: 10
节头大小: 40 (字节)
节头数量: 30
字符串表索引节头: 27
```

可以看出该文件运行的环境为 ARM。

## 2.3 超级终端和 Minicom

在对目标板进行查看、操作或目标板和上位机进行文件传输与通信时，需要安装终端软件。通过终端软件来对目标板进行配置，或者执行目标板上的程序与主机进行通信。

### 2.3.1 超级终端软件的安装

在 Windows 环境中，一般使用系统自带的超级终端软件，或者安装其他的终端软件。下面介绍超级终端软件的使用和设置。

在 Windows XP 上，可以直接执行“开始”|“所有程序”|“附件”|“通信”|“超级终端”命令，打开超级终端软件。但是在 Windows 7 中已经没有超级终端。用户需要从网上下载 Windows XP 的超级终端，然后放到 Windows 7 中运行。运行后，如图 2.31 所示。在“名称”文本框中，可以任意为其取名字，可以以开发板型号作为名称。



图 2.31 打开超级终端软件界面

开发板一般是通过串口线和 PC 连接。在选择连接使用的端口时，如果有计算机串行接口，则一般默认选择 COM1。图 2.32 是对连接端口的选择。如果第一次设置不清楚时，可以查看设备管理器，如图 2.33 所示。

对端口参数的设置包括比特率、奇偶校验、数据流控制等，其相应的选择如图 2.34 所示。最后单击“确定”按钮保存配置，下次可以直接使用。

### 2.3.2 配置 Minicom

Minicom 是 Linux 系统中的终端软件。在 Linux 系统中可以通过此软件访问目标板。

如果上位机中带有串口，那么ttyS0代表COM1，ttyS1代表的是COM2。如果在上位机中不带串口，那么需要编译加载usbserial模块。加载模块后可以在dev下面生成ttyUSB0。编译内核、驱动的方法和加载驱动的方法将在后面介绍。下面介绍Windows平台下minicom的设置。



图 2.32 超级终端连接端口选择



图 2.33 确定与开发板相连的端口

(1) 在终端中使用minicom -s命令进行配置界面，如图2.35所示。

```
#minicom -s
```

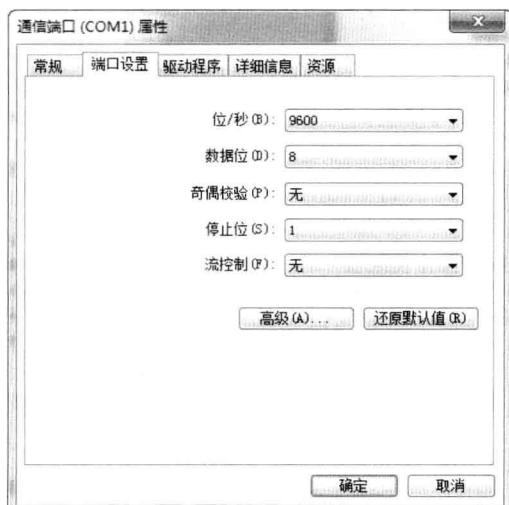


图 2.34 端口参数设置



图 2.35 Minicom 的配置界面

(2) 使用键盘的上下键对光标进行操作，选择Serial port setup项后按Enter键确定进入此项进行配置。如果选错选项可使用Esc键退出。进入串口的配置界面如图2.36所示。

(3) 通过按键盘的A、C、D、E、F、G键选择进入各个参数项的配置。配置完成后按Enter键确认配置。对波特率、数据位和停止位的参数配置，如图2.37所示，最终配置如图2.38所示。最后退出并保存配置。



图 2.36 串口的配置界面

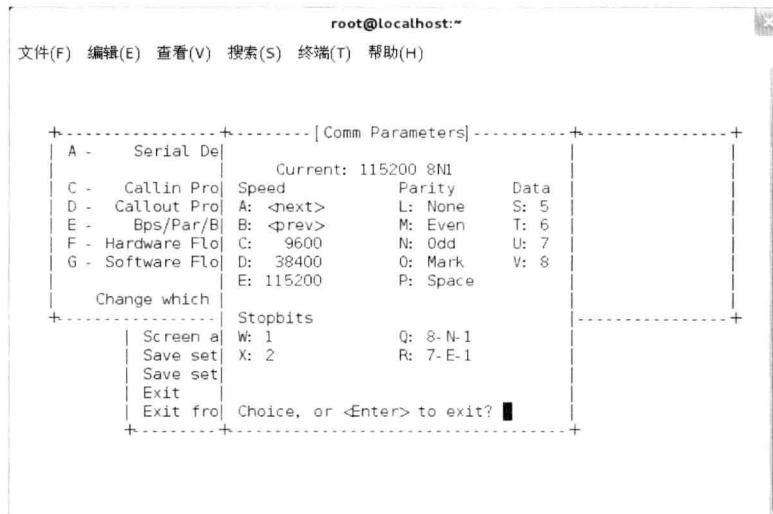


图 2.37 波特率、数据位和停止位配置



图 2.38 最终配置

### 2.3.3 配置 SecureCRT

SecureCRT也是一款功能强大的终端软件，其使用环境为Windows环境。它的安装过程比较简单。本节只介绍其第一次使用时的配置过程。配置一次后，以后直接连接即可。

(1) 通过菜单File | Connect或File | Quick Connect命令均可进入配置界面，或者通过工具栏的Connect按钮和Quick Connect按钮进入配置界面。两者的配置界面不同，但是参数是一样的。这里举例Quick Connect配置界面，如图2.39所示。

(2) 单击Connect按钮保存配置并进入连接状态。下次运行此软件时可以直接选择此连接，如图2.40所示，直接单击Connect按钮进入连接状态。

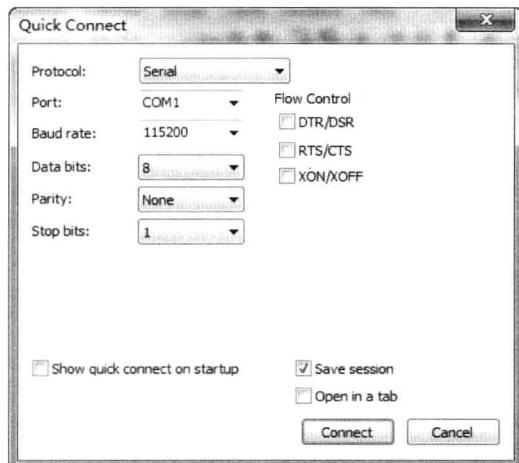


图2.39 SecureCRT串口配置

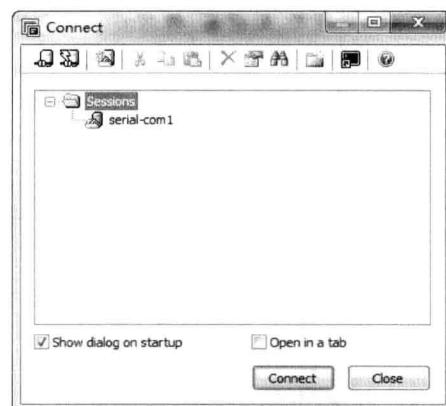


图2.40 重新运行SecureCRT界面

注意：以上3个终端软件任意使用一个即可，并非都要进行安装。

## 2.4 内核、文件系统加载工具

内核、文件系统加载工具是嵌入式开发必备的工具，在购买开发板时，会得到配套的这类工具。不同的公司提供的工具和方法略有不同。这里针对重要的地方加以介绍。

### 2.4.1 烧写Bootloader

烧写Bootloader可以使用超级终端的“传送”|“发送文件”命令进入发送文件对话框，使用Xmodem协议和Kermit协议发送Bootloader的各个文件。选择协议如图2.41所示。

除了比较通用的Bootloader之外，FriendlyARM公司专门为产品开发了MiniTools工具，该工具的使用方式非常简单，为入门读者提供了方便。使用MiniTools烧写Bootloader程序过程如下所述。

- (1) 确定开发板与主机相连，并且驱动运行正确。
- (2) 运行 MiniTools 软件，将开发板的 NOR/NAND 选择开关置于 NOR 位置，然后打开开发板电源。此时可以看到 MiniTools 工具已经正确连接到开发板，如图 2.42 所示。

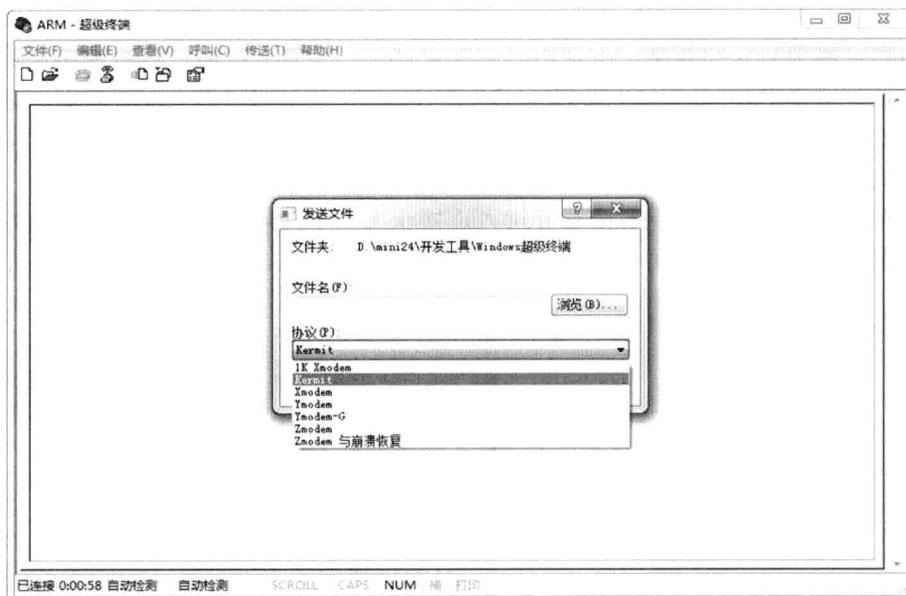


图 2.41 使用超级终端烧写 Bootloader



图 2.42 MiniTools 连接开发板

- (3) 正确连接开发板后，选择开发板的系统类型，如图 2.43 所示。



图 2.43 连接开发板

(4) 选择 Bootloader 文件。通过要烧写的文件对应项目后的“...”按钮选择 Bootloader 文件，并且选中对应项目的复选框，如图 2.44 所示。

(5) 烧写 Bootloader。单击软件界面右下角的“开始烧写”按钮开始烧写 Bootloader。烧写完成后会在“详细信息”窗口中显示成功信息，如图 2.45 所示。



图 2.44 选择 Bootloader 文件



图 2.45 烧写完成

## 2.4.2 内核和文件系统下载

有些公司提供网口下载内核和文件系统的方式；而有些公司采用 USB 方式下载文件系统和内核。采用网口方式下载时需要安装 TFTP 工具，然后设置正确的 IP 地址和下载文件路径，同时需要在 U-boot 中设置服务器的 IP 地址为上位机的 IP 地址。设置开发板的 IP 地址与上位机的 IP 地址为同一个网段，在上位机中建立 TFTP 服务器后，通过终端软件输入 tftp 命令下载内核和文件系统。

内核和文件系统也可以通过 MiniTools 工具烧写，烧写过程与烧写 Bootloader 类似。与烧写 Bootloader 不同的是，内核和文件系统下载需要选择正确的内核文件和文件系统文件，如图 2.46 所示。

在烧写完成后会在“详细信息”中显示操作步骤以及完成信息，如图 2.47 所示。

## 2.4.3 应用程序和文件传输

整个系统移植成功后，还有一些应用程序或者文件要在开发板和上位机之间进行传输。一般选择使用 SecureCRT 通过串口进行传输。SecureCRT 支持多种传输协议，通常使用 Zmodem 协议来传输。该协议的传输速度和成功率都是比较优秀的。在 SecureCRT 正确连接开发板后，通过 Transfer|Zmodem Upload List...命令来选择要传输的文件，如图 2.48 所示。



图 2.46 选择内核和文件系统文件



图 2.47 内核和文件系统烧写完成

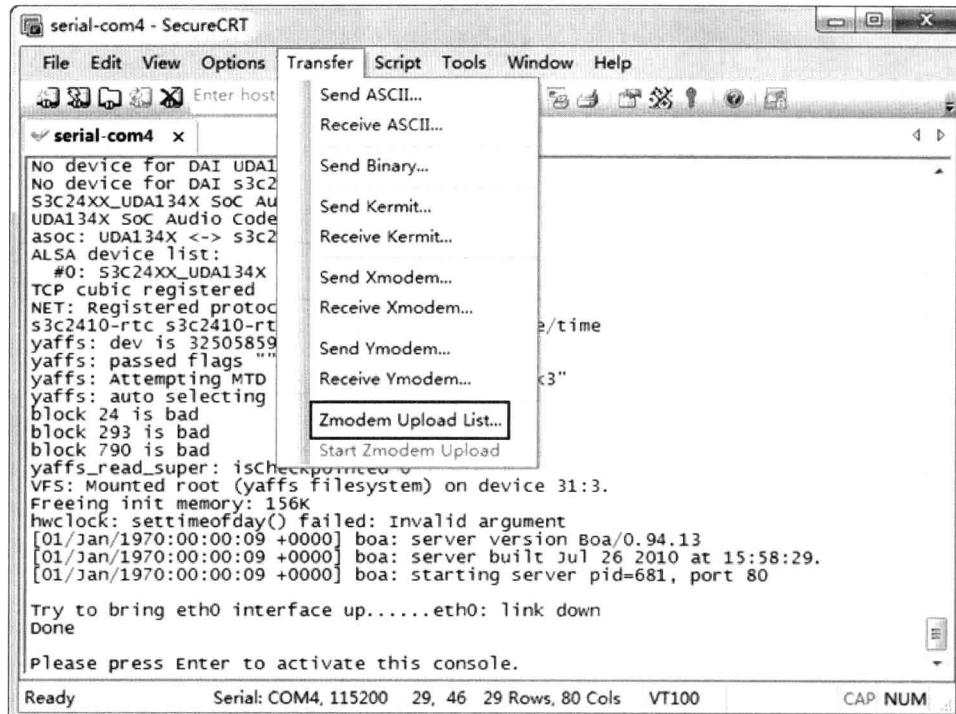


图 2.48 选择传输方式

在弹出的对话框中选择要传输的文件，并单击 Add 按钮添加到传输列表中，如图 2.49 所示。

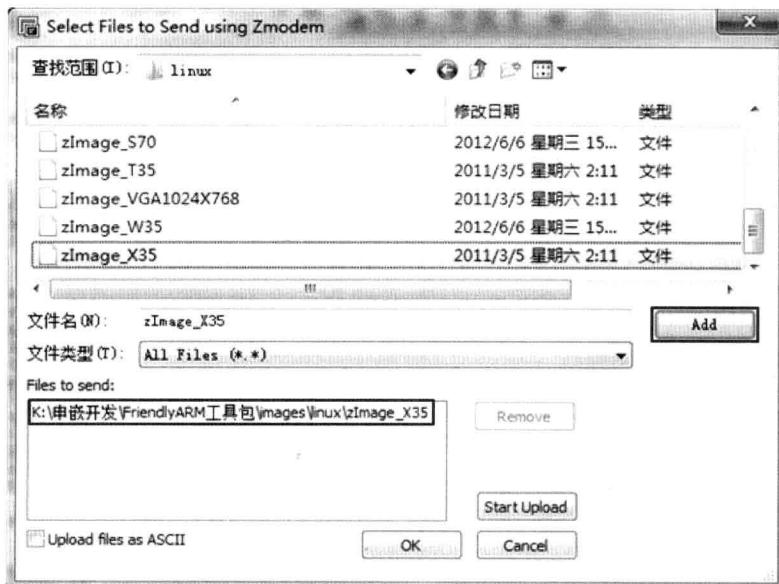


图 2.49 传输文件过程

可以单击 Start Upload 按钮开始传输文件，如图 2.50 所示。

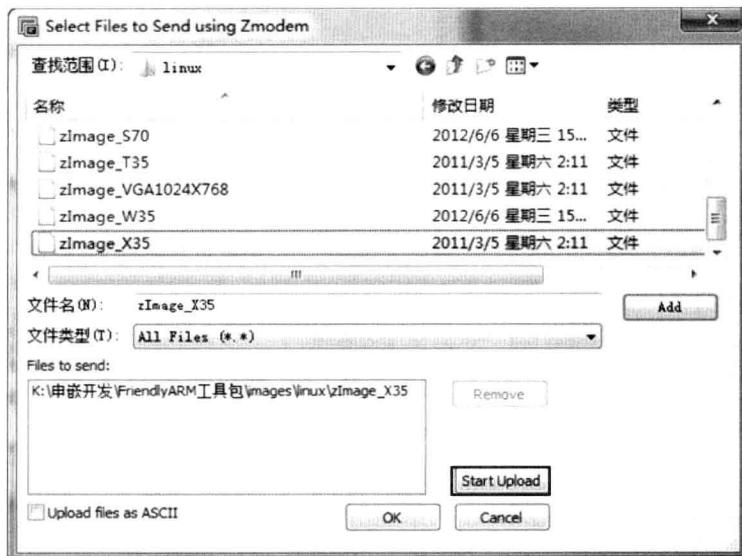


图 2.50 Start Upload

也可以单击 OK 按钮后通过 Transfer|Start Zmodem Upload 命令开始传输文件, 如图 2.51 所示。

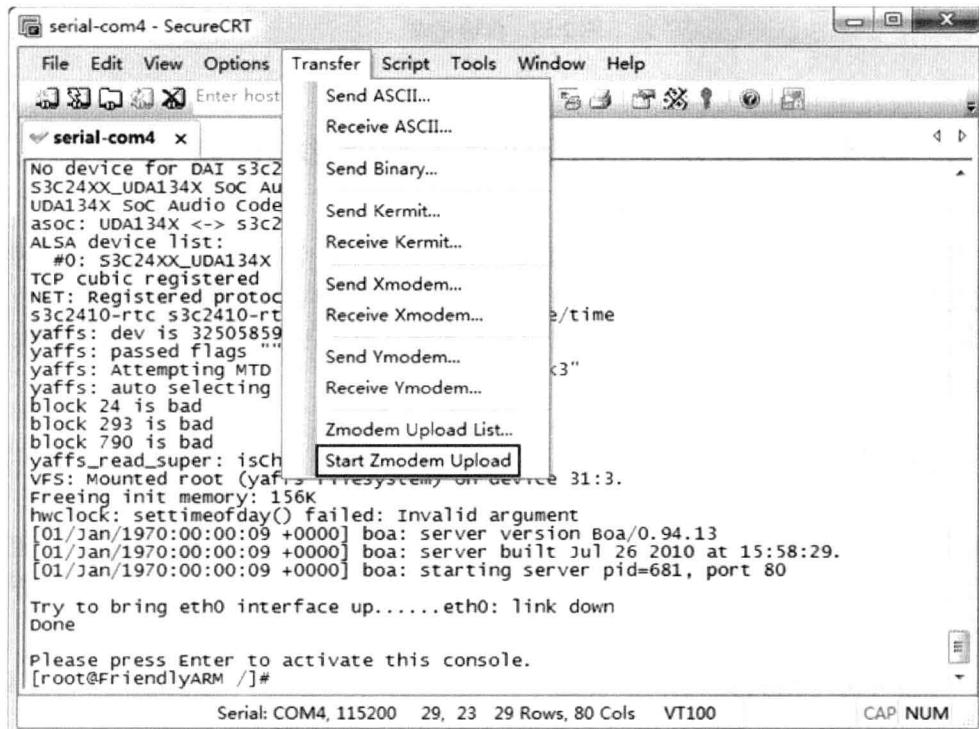


图 2.51 Start Zmodem Upload

文件会传输在登录用户的当前工作目录下, 如图 2.52 所示。

图 2.52 传输完成

## 2.5 在开发中使用网络文件系统（NFS）

在开发中使用 NFS 有两个优点：开发过程中不受开发板空间的限制，直接使用网络文件就像使用本地文件一样；调试过程中避免一一将编译后的应用程序和库文件复制到开发板上。在开发板中使用网络文件系统可以为开发和调试节省不少时间。下面介绍虚拟机环境下搭建 NFS 的过程。

### 2.5.1 虚拟机设置

这里需要配置虚拟机，让虚拟机能够直接访问局域网内的任何主机。前面为了能够让虚拟机与宿主机进行通信，将虚拟机的网络连接设置为 NAT 方式，下面主要介绍桥接模式和 NAT 模式。

#### 1. 桥接模式（Bridged Networking）

在桥接模式下，VMWare 虚拟出来的操作系统相当于局域网中一台独立的主机，它可以访问网内任何一台机器。在这种模式下，需要手动为虚拟系统配置和宿主机器处于同一网段的 IP 地址和子网掩码，这样虚拟系统就可以和宿主机器进行通信。如果配置好网关和 DNS 的地址，还可以通过局域网的网关或路由器访问互联网。

## 2. NAT模式（Network Address Translation）

在NAT模式下，虚拟系统借助NAT（网络地址转换）功能，通过宿主机器所在的网络来访问互联网。NAT模式下的虚拟系统的TCP/IP配置信息是由VMnet（NAT）虚拟网络的DHCP服务器提供，无法进行手动修改，因此虚拟系统和局域网中的其他真实主机无法通信。

为了使虚拟机、宿主机和开发板能达到互相通信的目的，虚拟机的网络连接方式应该采用桥接方式，通过选择菜单“虚拟机”，然后在下拉菜单中选择“设置”选项，在弹出的“虚拟机设置”窗口中进行设置，如图2.53所示。

**注意：**设置虚拟机网络连接时，应该在虚拟机没有启动时进行设置，否则无法设置或者设置无法生效。

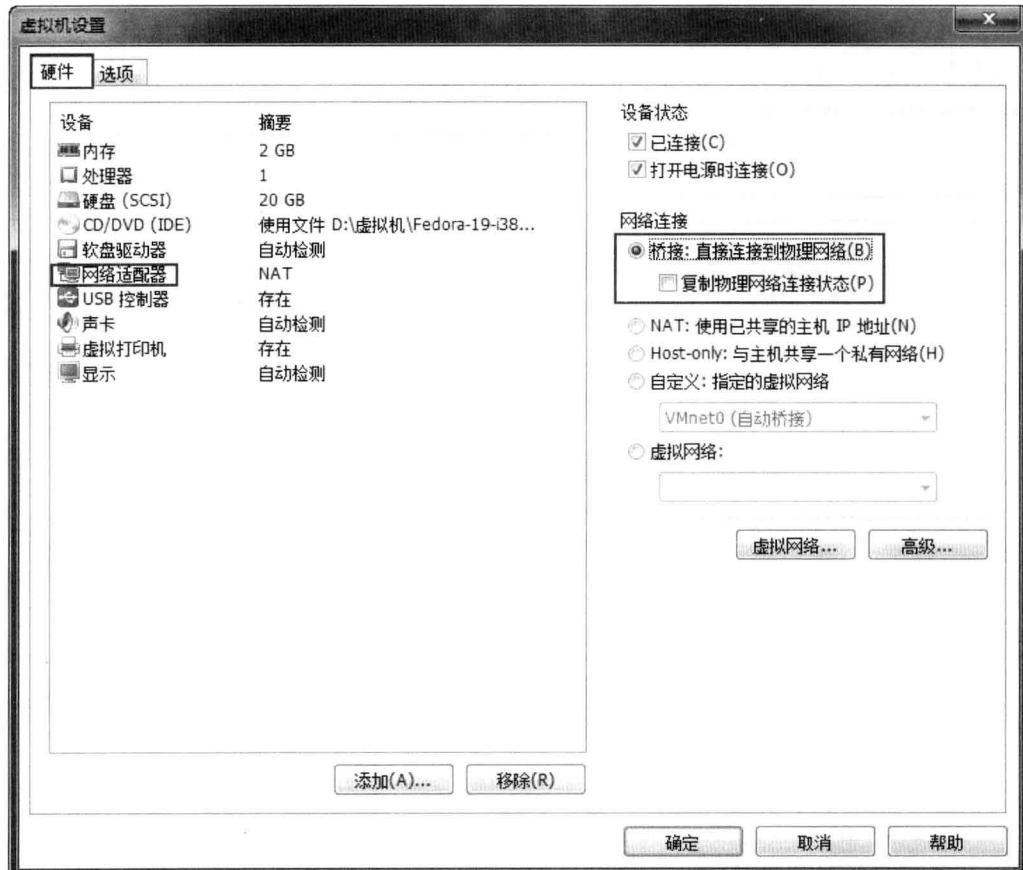


图2.53 设置网络连接方式为桥接方式

### 2.5.2 虚拟机的IP地址设置

启动虚拟机，查看虚拟机的IP地址和网络连接状态。在右下角查看虚拟网卡是否已连接，在终端输入ifconfig查看网卡是否已设置，如图2.54所示。



```

root@localhost ~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[ root@localhost ~ ]# ifconfig
lo  flags=73<IP,LOOPBACK,RUNNING> mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0
                inet6 ::1 prefixlen 128 scopedid 0x10<host>
                    loop txqueuelen 0 (Local Loopback)
                    RX packets 0 bytes 0 (0.0 B)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 0 bytes 0 (0.0 B)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

p3p1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 192.168.1.111 netmask 255.255.255.0 broadcast 192.168.1.255
                inet6 fe80::20c:29ff:fe04:3707 prefixlen 64 scopedid 0x20<link>
                    ether 00:0c:29:64:37:07 txqueuelen 1000 (Ethernet)
                    RX packets 5820 bytes 8642291 (8.2 MiB)
                    RX errors 0 dropped 0 overruns 0 frame 0
                    TX packets 2945 bytes 167853 (163.9 KiB)
                    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
                    device interrupt 19 base 0x2000

[ root@localhost ~ ]#

```

图 2.54 虚拟机的网络连接状态

或者可以单击右上角的网络图标，然后在弹出的面板中，单击“网络设置”按钮，打开“网络”对话框，如图 2.55 所示。单击右下角的设置图标进入网卡设置对话框，在该对话框中对虚拟机 IP 地址和网关进行设置，如图 2.56 所示。



图 2.55 虚拟机网络配置

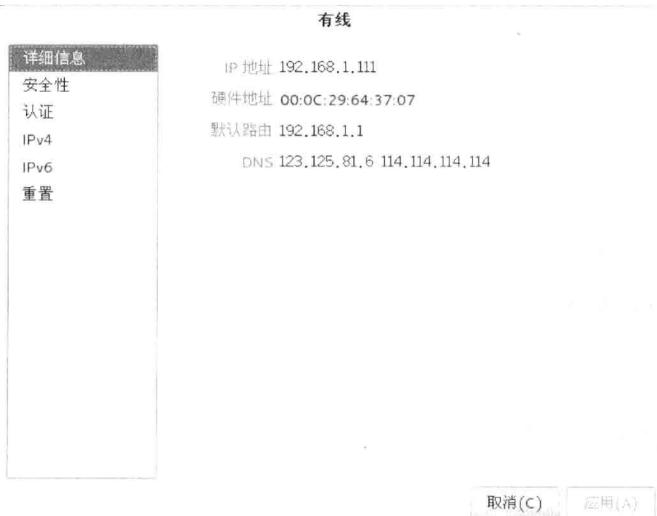


图 2.56 虚拟机 IP 地址设置

**注意：**虚拟机启动后看虚拟机右下角的虚拟网卡标识是否连接上，如果没有连接上则有可能是安装虚拟机时少选了一项VMware Bridge Protocol。如果连接上就不必进行下面的安装过程了。

可以打开主机网络连接的属性窗口，在该窗口中有本地连接VMnet1、VMnet8和网络连接。右击VMnet1或者VMnet8，选择“属性”进入“VMware Network Adapter VMnet1属性”对话框，单击“安装”按钮，如图2.57所示。进入“选择网络功能类型”对话框后，选择“服务”选项，并单击“添加”按钮，如图2.58所示。

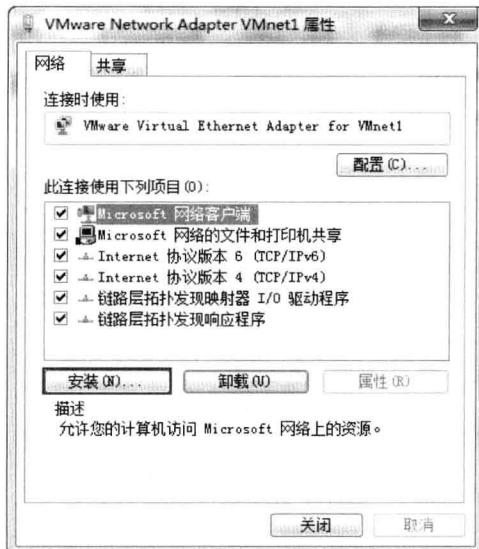


图2.57 进入“VMware Network Adapter VMnet1属性”窗口



图2.58 添加服务

在“选择网络服务”窗口，单击“从磁盘安装”按钮，如图2.59所示。在虚拟机安装路径下的VMware Workstation目录中找到netbridge.inf文件，然后“打开”按钮进行安装，如图2.60所示。



图2.59 选择从磁盘安装

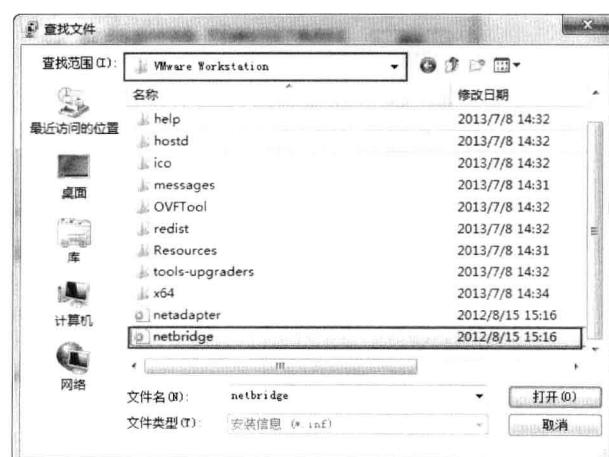


图2.60 打开安装文件netbridge.inf

安装过程完成后，在“VMware Network Adapter VMnet1 属性”窗口出现了 VMware Bridge Protocol 项，如图 2.61 所示。重新启动计算机，并且重启虚拟机。

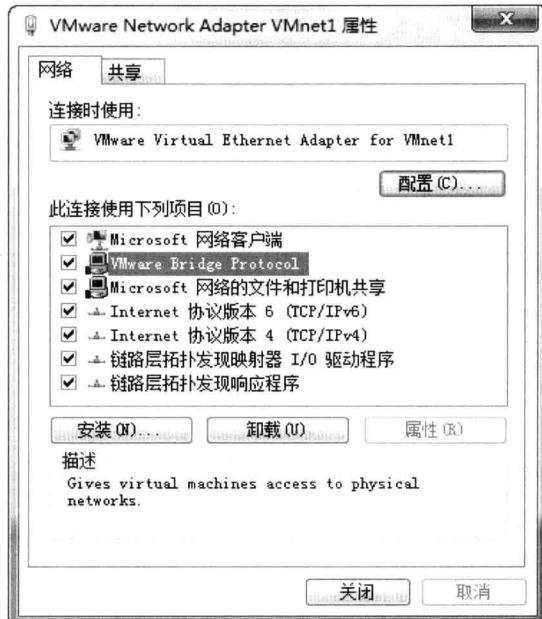


图 2.61 安装后出现 VMware Bridge Protocol 项

### 2.5.3 验证网络连接

主机的 IP 地址为 192.168.1.109，虚拟机的 IP 地址为 192.168.1.111，开发板的 IP 地址为 192.168.1.113。分别通过 ping 命令验证两两之间是否可以通信，正常情况下是可以互相 ping 通，但是如果物理网卡没有连接网线则无法实现通信（开发板与主机采用交叉网线连接），如图 2.62 是虚拟机 ping 主机的情况。

```
root@localhost:~ 
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[ root@localhost ~ ] # ping 192.168.1.109
PING 192.168.1.109 (192.168.1.109) 56(84) bytes of data.
64 bytes from 192.168.1.109: icmp_seq=1 ttl=28 time=1.14 ms
64 bytes from 192.168.1.109: icmp_seq=2 ttl=28 time=0.348 ms
64 bytes from 192.168.1.109: icmp_seq=3 ttl=28 time=0.305 ms
64 bytes from 192.168.1.109: icmp_seq=4 ttl=28 time=0.300 ms
64 bytes from 192.168.1.109: icmp_seq=5 ttl=28 time=0.292 ms
64 bytes from 192.168.1.109: icmp_seq=6 ttl=28 time=0.319 ms
64 bytes from 192.168.1.109: icmp_seq=7 ttl=28 time=0.299 ms
64 bytes from 192.168.1.109: icmp_seq=8 ttl=28 time=0.284 ms
64 bytes from 192.168.1.109: icmp_seq=9 ttl=28 time=0.315 ms
64 bytes from 192.168.1.109: icmp_seq=10 ttl=28 time=0.314 ms
^Z
[1] + 已停止          ping 192.168.1.109
[ root@localhost ~ ] #
```

图 2.62 虚拟机 ping 主机

## 2.5.4 设置共享目录

编辑文件/etc(exports，在文件中添加以下内容。

```
/home/nfs 192.168.1.*(rw, sync, no_root_squash)
```

- /home/nfs：表示共享给其他主机的共享目录；
- 192.168.1.\*：表示IP地址为192.168.1.2/254的主机都能够挂载/home/nfs目录；
- rw：表示挂接此目录的客户机对该目录具有读写的权力；
- sync：表示同步写入存储器；
- no\_root\_squash：表示允许挂接此目录的客户机享有该主机的root身份。

使用下面的命令查看共享目录：

```
# showmount -a
```

如果出现错误：showmount: can't get address for localhost.localdomain，则修改文件/etc/hosts，将

```
::1      localhost.localdomain  localhost
```

修改为：

```
127.0.0.1      localhost.localdomain  localhost
```

## 2.5.5 启动NFS服务

启动NFS服务之前，首先需要在防火墙中允许NFS和RPC-bind服务，并且启动RPC-bind服务。进入“防火墙配置”设置，如图2.63所示。



图2.63 禁止防火墙

启动 RPC-bing 服务和 NFS 服务的命令如下：

```
# service rpcbing start //服务启动
# service rpcbing restart //服务重启
# service nfs start
# service nfs restart
```

如果在启动过程中出现“启动 NFS 守护进程失败”等错误提示，则重新计算机后，再进行尝试。

## 2.5.6 修改共享配置后

修改/etc(exports 配置文件后，应该使配置文件重新生效。在启动了 NFS 之后又修改了 /etc(exports，此时就可以用 exportfs 命令使改动立刻生效，该命令格式如下：

```
exportfs [-aruv]
```

- a：全部 mount 或者 umount /etc(exports 中的内容；
- r：重新 mount /etc(exports 中共享出来的目录；
- u：umount 目录；
- v：在 export 的时候，将详细的信息输出到屏幕上。

例如：

```
exportfs -rv
```

该命令重新输出全部的共享目录信息。在每次修改了/etc(exports 文件后都要运行一次该命令，使共享配置生效。

## 2.5.7 挂载 NFS

在虚拟机上修改共享目录/home/nfs 的权限为 777。开发板与主机通过交叉网线连接后，虚拟机、主机及开发板三者可以进行互相通信。使用 mount 命令在开发板上挂载此目录。

```
# chmod 777 /home/nfs
# mount -t nfs 192.168.1.123:/home/nfs /mnt
或者使用
# mount -o noblock -t nfs 192.168.1.123:/home/nfs /mnt
```

## 2.5.8 双网卡挂载 NFS

当拥有两张物理网卡时，专门用一张网卡将 ARM 板和虚拟机相连，将两者的 IP 设置在一个 IP 段内。具体过程和单网卡类似，首先做到虚拟机和 ARM 能相互 ping 通，能正常启动 NFS 服务，最后挂载网络文件系统。

在搭建 NFS 时，给出一些错误情况解决的方法。

- 当启动 NFS 服务失败时，解决的办法通常是修改/etc(exports 文件，出错的原因通常是权限引起的。

- 当出现 RPC 等报错时，应该注意防火墙是否关闭。
- 当挂载 NFS 时，出现 Permission denied 报错时，检查/etc(exports)文件中的权限设置，另外检查共享目录的权限设置。

出现任何报错的情况，都应该查看错误日志/var/log/messages，对照错误日志查找问题。笔者在挂载的过程中遇到以下几个问题。

#### (1) mount: RPC: Timed out

该问题是由于主机的防火墙引起的，关闭了虚拟机的防火墙后请注意，主机的防火墙也可能对 RPC 的包进行拦截。遇到此类问题时，请读者注意虚拟机和主机两者的防火墙是否关闭。

(2) 在使用下面的 mount -t nfs 192.168.1.123:/home/nfs /mnt 命令进行挂载时，出现下面的错误。

```
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
rpcbind: server localhost not responding, timed out
RPC: failed to contact local rpcbind server (errno 5).
将挂载命令修改如下后挂载成功
#mount -o nolock 192.168.1.123:/home/nfs /mnt 或者
#mount -o nolock -t nfs 192.168.1.123:/home/nfs /mnt
```

## 2.6 小结

安装交叉编译工具是本章最复杂的一节，建议读者在有充足的时间及有必要的情况下才去编译，一般可以直接使用开发板公司提供的稳定的交叉编译工具。另外在编译前，应该参照一些编译成功的例子，与其版本最好一致。如果编译过程中出现问题，首先应该考虑版本是否兼容。更换新版本时应该清除旧版本编译过程中生成的文件。重新编译后应该核对生成工具的版本是否和更换的版本相符。因为初次编译交叉环境需要很长时间，所以在继续编译时有可能出现环境变量失效的问题。读者在安装的过程中直接使用全路径或者设置永久的环境变量的方式。一般而言，对编译安装路径变量不提倡设置为永久的环境变量，因为在以后的内核编译、驱动移植、文件系统移植等编译过程中都会有路径设置情况。



## 第2篇 系统移植技术篇

- ▶ 第3章 Bootloader 移植
- ▶ 第4章 Linux 内核裁剪与移植
- ▶ 第5章 嵌入式文件系统制作

# 第3章 Bootloader 移植

Bootloader 是在嵌入式系统运行之前运行的一段程序。运行 Bootloader 程序可以初始化硬件设备，建立内存空间的映射图，从而将系统的软硬件环境调整到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。本章主要介绍两种常见的 Bootloader 及其移植过程。

## 3.1 Bootloader 介绍

体系结构不同的 CPU 都有不同的 Bootloader，有些 Bootloader 支持多种不同类型体系结构的处理器，如 U-boot。通常，Bootloader 不但依赖于 CPU 的体系结构，而且依赖于特定的嵌入式板设备的配置，即对于两块不同的嵌入式板而言，即使它们是基于同一种 CPU 而构建的，要运行在一块开发板上的 Bootloader 程序能够运行在另一块开发板上，通常需要修改 Bootloader 的源程序以适应不同的开发板。

### 3.1.1 Bootloader 与嵌入式 Linux 系统的关系

从软件的角度可将嵌入式 Linux 系统划分成 4 个层次，4 个层次由低层到高层分别如下所述。

- 引导加载程序：包括固化在固件中（firmware）的 boot 代码（可选）和 Bootloader 两大部分。
- 内核：给具体类型开发板定制的内核及控制内核引导系统的参数。
- 文件系统：包括根文件系统和建立于 FLASH 内存设备上的文件系统。
- 用户应用程序：用户的应用程序，包括 GUI、Web 服务器、数据库、网络协议栈等。

### 3.1.2 Bootloader 基本概念

Bootloader 是在操作系统内核运行前执行的一段小程序，类似在启动 Windows 系统前运行的 BIOS 程序。通过这段小程序，完成了对必要硬件设备的初始化，创建内核需要的信息并将这些信息通过相关机制传递给内核，从而将系统的软硬件环境带到一个合适的状态，最终调用操作系统内核，起到引导和加载内核的作用。

## 1. Bootloader的安装媒介

系统每次加电或复位后，CPU 都会固定从预先设定的地址上取指令。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）被映射到这个预先设定的地址上。

一个同时安装有 Bootloader、内核的启动参数、内核映像和根文件系统映像的固态存储设备的典型空间分配结构图，如图 3.1 所示。

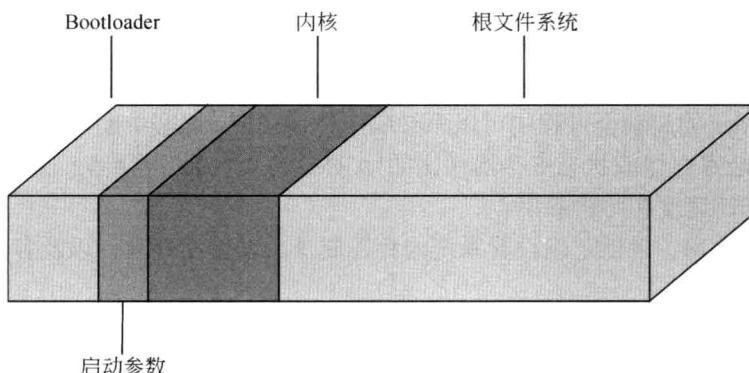


图 3.1 固态存储设备的典型空间分配结构图

## 2. Bootloader启动过程分类

Bootloader 启动过程分为单阶段和多阶段两种。相对单阶段 Bootloader 而言，多阶段 Bootloader 的功能更加复杂，可移植性更加优越。从固态存储设备上启动 Bootloader 一般可分为两个阶段的启动过程，即 stage 1 和 stage 2。

## 3. Bootloader的操作模式

绝大部分 Bootloader 均包含两种不同类型的操作模式，即启动加载模式和下载模式。

- 启动加载模式：这种模式也称为“自主”模式。即 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。启动加载模式为 Bootloader 的正常工作模式，因此在嵌入式产品发布时，Bootloader 只能工作在该模式下。
- 下载模式：这种模式下，目标机上的 Bootloader 将通过串口连接、网络连接或 USB 连接等通信手段从主机（Host）下载文件，如下载内核映像文件和文件系统映像文件等。从主机下载的文件通常首先被 Bootloader 保存到目标板的 ROM 中，然后再被 Bootloader 写到目标板上的 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，或者在系统更新时使用。Bootloader 工作在下载模式时，通常都会提供一个命令行接口给它的终端用户，以供用户通过命令行控制 Bootloader 的工作。

### 3.1.3 Bootloader 启动过程

Bootloader 的启动过程分为 stage1 和 stage2 两个阶段，通常 stage1 是用汇编语言完成，而 stage2 则用 C 语言来实现，以便于在 stage2 阶段实现更加复杂的功能和取得更好的代码可读性及可移植性。下面介绍两个阶段分别完成的不同工作。

#### 1. stage1完成的工作

(1) 基本的硬件初始化包括以下工作：

- 屏蔽所有中断。为中断提供服务通常是操作系统设备驱动程序的责任，因此在 Bootloader 的启动全过程中可以不必响应任何中断。屏蔽中断可以通过写 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 速度和时钟频率。
- 初始化 RAM。包括正确设置系统内存控制器的功能寄存器，以及各内存库控制寄存器等。
- 初始化 LED。典型地，通过 GPIO 来驱动 LED，其目的是检查当前系统的状态是 OK 还是 ERROR。如果板子上没有 LED，那么也可以通过初始化 UART 向串口打印 Bootloader 的 Logo 字符信息来完成这一点。
- 关闭 CPU 内部指令和数据 cache 灯。

(2) 准备 RAM 空间加载 stage2。为了获得更快的执行速度，通常把 stage2 加载到 RAM 空间中来执行，所以必须准备好一段可用的 RAM 空间范围用来加载 Bootloader 的 stage2。

(3) 复制 stage2 到 RAM 中。执行复制时要确定两类地址：第一，stage2 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；第二，RAM 空间的起始地址。

(4) 设置堆栈指针 sp。设置堆栈指针是为执行 C 语言代码做准备。在设置堆栈指针 sp 之前，也可以关闭 LED 灯，以提示用户我们准备跳转到 stage2。

(5) 跳转到 stage2 的 C 入口点。在 ARM 处理器中，实现跳转的方法是通过修改 PC 寄存器为合适的地址。

#### 2. stage2完成的工作

(1) 使用汇编语言跳转到 main()入口函数。用汇编语言写一段 trampoline 小程序，并将这段 trampoline 小程序作为 stage2 可执行映像的执行入口点。然后在 trampoline 汇编小程序中用 CPU 跳转指令跳入 main()函数中去执行；而当 main()函数返回时，CPU 执行路径再次回到 trampoline 程序。这种方法的思想是：用这段 trampoline 小程序作为 main()函数的外部包裹（external wrapper）。

(2) 初始化本阶段要使用到的硬件设备。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。

(3) 检测系统的内存映射。所谓内存映射是指在整个 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。

(4) 加载内核映像文件和根文件系统映像文件。包括规划内存分配布局和从 Flash 上

复制数据。规划内存分配布局包括内核映像所占用的内存范围和根文件系统所占用的内存范围。

(5) 设置内核的启动参数。在将内核映像和根文件系统映像复制到 RAM 空间中后，就可以准备启动 Linux 内核了。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。

## 3.2 Bootloader 之 U-Boot

U-Boot(全称 Universal Boot Loader)是遵循 GPL 条款的开放源码项目。从 FADSROM、8xxROM、PPCBOOT 逐步发展演化而来。其源码目录、编译形式与 Linux 内核很相似，实际上，在 U-Boot 的源码中很多是相应的 Linux 内核源程序的简化，尤其体现在一些设备驱动程序的源代码上，而且读者可以从 U-Boot 源码的注释中能够直接看到源码来自 Linux 内核源码。

U-Boot 不仅能够引导加载嵌入式 Linux 系统，还能引导加载 NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 嵌入式操作系统。其目前支持的目标操作系统是 OpenBSD、NetBSD、FreeBSD、4.4BSD、Linux、SVR4、Esix、Solaris、Irix、SCO、Dell、NCR、VxWorks、LynxOS、pSOS、QNX、RTEMS 和 ARTOS。U-Boot 中 Universal 的一方面是指支持前面提到的通用操作系统，另一方面是指 U-Boot 支持通用的处理器，包括 PowerPC、MIPS、x86、ARM、NIOS 和 XScale 等不同体系结构的常用系列处理器。

对大多数操作系统的支持和大多数处理器的支持是 U-Boot 项目的开发目标。从目前情况来看，U-Boot 对 PowerPC 系列处理器的支持最丰富，对 Linux 操作系统的支持最完善。

### 3.2.1 U-Boot 优点

U-Boot 在目前的嵌入式开发中被广泛采用，是因为其具有很多优点。其优点包括以下几点：

- 开放源码；
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS；
- 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale；
- 较高的可靠性和稳定性；
- 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等；
- 丰富的设备驱动源码，如串口、以太网、SDRAM、FLASH、LCD、NVRAM、EEPROM、RTC、键盘等；
- 较为丰富的开发调试文档与强大的网络技术支持。

### 3.2.2 U-Boot 的主要功能

U-Boot 的功能非常强大，其主要功能如下。

- 系统引导：支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统，支持 NFS 挂载，从 FLASH 中引导压缩或非压缩系统内核。
- 基本辅助功能：强大的操作系统接口功能；可灵活设置、传递多个关键参数给操作系统，满足系统不同阶段的开发调试和产品发布要求，特别是对 Linux 支持最完善；支持目标板环境参数多种存储方式，如 FLASH、NVRAM 和 EEPROM；CRC32 校验，可校验 FLASH 中内核、RAMDISK 镜像文件是否完好。
- 设备驱动：串口、SDRAM、FLASH、以太网、LCD、NVRAM、EEPROM、键盘、USB、PCMCIA、PCI 和 RTC 等驱动支持。
- 上电自检功能：SDRAM、FLASH 大小自动检测、SDRAM 故障检测、CPU 型号检测。
- 特殊功能：XIP 内核引导。

### 3.2.3 U-Boot 目录结构

以 U-Boot-1.1.6 为例介绍其目录结构。共有 27 个文件可以分为 3 类。

- 第 1 类目录与处理器体系结构或者开发板硬件直接相关；
- 第 2 类目录是一些通用的函数或者驱动程序；
- 第 3 类目录是 U-Boot 的应用程序、工具或者文档。

以 U-Boot-1.1.6 为例，其顶层目录下各级子目录的存放规则，如表 3.1 所示。

表 3.1 U-Boot各级子目录存放规则

目 录	特 性	说 明
board	平台依赖	存放已有开发板相关的目录文件，例如 RPXlite（mpc8xx）、smdk2410（arm920t）和 sc520_cdp（x86）等目录
cpu	平台依赖	存放与 CPU 相关的目录文件，例如 mpc8xx、ppc4xx、arm720t、arm920t、xscale 和 i386 等目录
lib_arm	平台依赖	存放 ARM 体系结构通用的相关文件，主要是实现 ARM 平台通用的函数
lib_avr32	平台依赖	avr32 系统结构通用的文件
lib_blackfin	平台依赖	存放对 ADI Blackfin6 体系结构通用的文件，主要用于实现 ADI Blackfin 平台通用的函数
lib_generic	平台依赖	通用库函数的实现
lib_i386	平台依赖	存放 X86 体系结构下通用的相关文件，主要是实现 X86 平台通用的函数
lib_m68k	平台依赖	存放对 m68k 体系结构通用的文件，主要用于实现 m68k 平台通用的函数
lib_microblaze	平台依赖	存放对赛灵思公司 32 位 MicroBlaze 处理架构体系结构通用的文件，主要用于实现 MicroBlaze 平台通用的函数
lib_mips	平台依赖	存放对 MIPS 体系结构通用的文件，主要用于实现 MIPS 平台通用的函数
lib_nios	平台依赖	存放对 NIOS 体系结构通用的文件，主要用于实现 NIOS 平台通用的函数
lib_nios2	平台依赖	存放对 Nios-II 体系结构通用的文件，主要用于实现 Nios-II 平台通用的函数
lib_ppc	平台依赖	存放 PowerPC 体系结构通用的相关文件，主要是实现 PowerPC 平台通用的函数
nand_spl	通用	Nand Flash boot 的程序

续表

目 录	特 性	说 明
net	通用	存放网络的程序，BOOTP 协议、TFTP 协议、RARP 协议和 NFS 文件系统的实现
post	通用	存放上电自检程序
rtc	通用	RTC 的驱动程序
tools	工具	存放制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage、crc 等
common	通用	通用的多功能函数实现
disk	通用	硬盘接口程序
doc	文档	存放开发帮助相关文档
drivers	通用	存放通用的设备驱动程序，主要有以太网接口的驱动
dtt	通用	存放数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如 hello_world
fs	通用	存放文件系统的程序
include	通用	存放头文件和开发板配置文件，所有开发板的配置文件都在 configs 目录下

### 3.2.4 U-Boot 的编译

本节主要以 u-boot-2013.07 为例介绍 U-Boot 编译的主要步骤，针对的开发板是 mini2440。其他版本的 U-Boot 移植主要步骤与其类似。

由于目前 U-Boot 对主流开发板的支持非常好，因此在 u-boot-2013.07 中已经支持直接将 U-Boot 编译为适用于 FriendlyARM 公司 mini2440 开发板的二进制程序。

u-boot 的下载地址 <ftp://ftp.denx.de/pub/u-boot/>。下载 u-boot-2013.07.tar.bz2，然后执行如下命令完成编译：

```
#tar xvjf u-boot-2013.07.tar.bz2
#cd u-boot-2013.07
#make mini2440_config
#make all
```

编译生成的 u-boot.bin 在当前目录下。当然，读者可以使用如下的命令指定生成文件的位置：

```
#make mini2440_config O=/bin/dir
```

## 3.3 小 结

本章主要介绍了 Bootloader 的启动流程，以及这个启动流程在 U-Boot 和 vivi 上是如何体现的。本章还介绍了两种 Bootloader 在 ARM 平台的移植，通过介绍移植的主要步骤，使读者大致了解移植 Bootloader 时需要注意的方面；通过代码分析让读者清楚 Bootloader 执行流程和功能。

# 第4章 Linux 内核裁剪与移植

内核，即操作系统。它为底层的可编程部件提供服务，为上层应用程序提供执行环境。内核裁剪就是对这些功能进行裁剪，选取满足特定平台和需求的功能。不同的硬件平台对内核的要求也不同，因此从一个平台到另一个平台需要对内核进行重新配置和编译。操作系统从一个平台过渡到另一个平台称为移植。Linux 是一款平台适应性强且容易裁剪的操作系统，因此 Linux 在嵌入式系统得到了广泛的应用。本章将详细讲解内核裁剪与移植的各项技术。

## 4.1 Linux 内核结构

Linux 内核采用模块化设计，并且各个模块源码以文件目录的形式存放，在对内核裁剪和编译时非常方便。下面介绍内核的主要部分及其文件目录。

### 4.1.1 内核的主要组成部分

在第 1 章中已经介绍了 Linux 内核主要的 5 个部分：进程调度、内存管理、虚拟文件系统、网络接口、进程通信。在系统移植的时候，它们是内核的基本元素，这 5 个部分之间的关系，如图 4.1 所示。

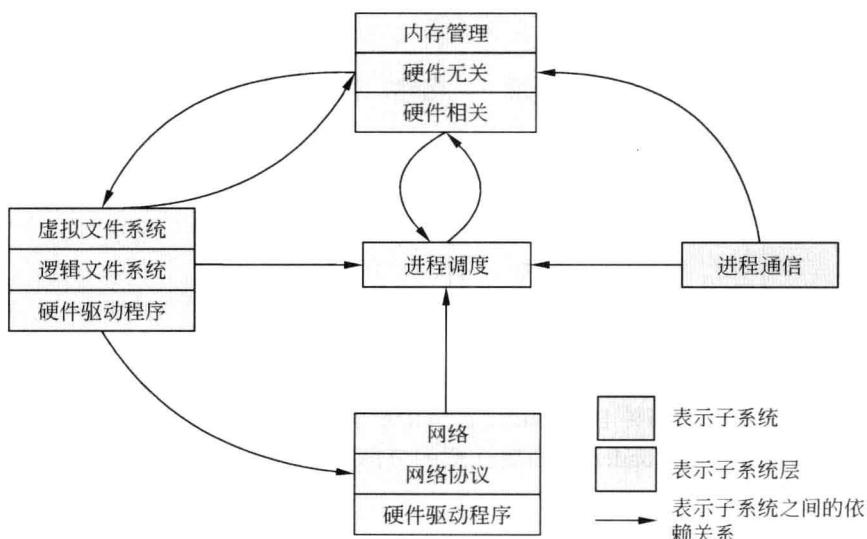


图 4.1 Linux 内核子系统及其之间的关系

进程调度部分负责控制进程对 CPU 的访问。内存管理允许多个进程安全地共享主内存区域。内存管理从逻辑上分为硬件无关部分和硬件相关部分。硬件无关部分提供了进程的映射和逻辑内存的对换；硬件相关部分为内存管理硬件提供了虚拟接口。虚拟文件系统隐藏了不同类型硬件的具体细节，为所有的硬件设备提供了一个标准的接口，VFS 提供了十多种不同类型的文件系统。网络接口提供了对各种网络标准的存取和各种网络硬件的支持。进程通信部分用于支持进程间各种不同的通信机制。进程调度处于核心位置，内核的其他子系统都要依赖它，因为每个子系统都存在进程挂起或恢复过程。

- 进程调度与内存管理之间的关系：这两个子系统为互相依赖关系。在多道程序环境下，程序要运行必须为之创建进程，而创建进程首先就是要将程序和数据装入内存。另外，内存管理子系统也存在进程的挂起和恢复过程。
- 进程间通信与内存管理之间的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，通过对共同的内存区域进行操作来达到通信的目的。
- 虚拟文件系统与网络接口之间的关系：虚拟文件系统通过依赖网络接口支持网络文件系统（NFS），也通过依赖内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程定期地由调度程序调度，这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时，内存管理将会向文件系统发出请求，同时，挂起当前正在运行的进程。

除了上面 5 个主要部分，下面将介绍 Linux 代码的整体分区结构。

### 4.1.2 内核源码目录介绍

Linux 内核代码以源码树的形式存放，如果在安装系统的时候已经安装了源码树，其源码树就在 /usr/src/linux 下，源码树结构如图 4.2 所示。

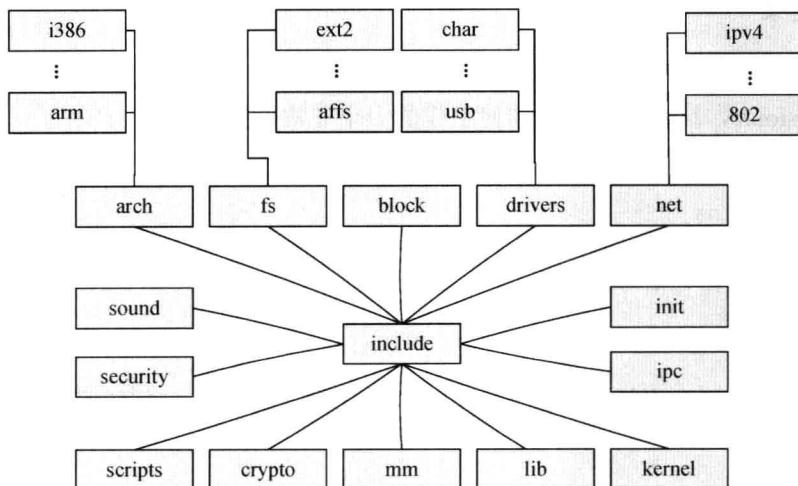


图 4.2 Linux 内核源码树结构

下面分别针对图 4.2 中各个部分进行介绍，各个目录的主要的功能分别介绍如下。

## 1. arch目录

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 arm 子目录是关于 ARM 平台下各种芯片兼容的代码。

## 2. include目录

include 子目录包括内核编译时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，include/scsi 目录则是有关 scsi 设备的头文件目录，与 arm 相关的头文件在 include/asm-arm 子目录下。

## 3. drivers目录

drivers 子目录放置系统所有的设备驱动程序。有些驱动是与硬件无关的，而有些驱动是与硬件平台相关。例如，在 USB 驱动中，主机控制器有 3 种规格：

- OHCI 主要为非 PC 系统上及带有 SiShe ALi 芯片组的 PC 主板上的 USB 芯片，嵌入式系统一般使用该驱动。
- UHCI 大多为 Intel 和 Via 主板上的 USB 控制器芯片。相对 OHCI 而言 UHCI 的硬件电路比较简单，同时其成本也比较低，但驱动复杂，但它们都是与 USB 1.1 规范同时提出的。
- EHCI 由 USB 2.0 规范所提出，它兼容 OHCI 和 UHCI。

## 4. fs目录

fs 子目录列出了 Linux 支持的所有文件系统，目前 Linux 支持 ext3、vfat、ntfs、yaffs2、ramfs、cramfs 和 romfs 等多种文件系统。在嵌入式系统中常用的闪存设备的文件系统有 cramfs、romfs、ramfs、jffs2、yaffs 等文件系统。

## 5. init目录

init 子目录包含核心的初始化代码（注意，不是系统的引导代码）。它包含两个文件 main.c 和 version.c，这是研究核心如何工作的一个非常好的起点。

## 6. ipc目录

ipc 子目录包含核心进程间的通信代码。Linux 下进程间通信机制主要包括管道、信号、消息队列、共享内存、信号量、套接口。

## 7. kernel目录

kernel 子目录包含内核管理的核心代码。与处理器结构相关代码都放在 arch/\*/kernel 目录下。

## 8. net目录

net 子目录里是核心的网络部分代码，其每个子目录存放一个具体的网络协议或者网络模型代码。

## 9. mm目录

mm 子目录包含了所有的内存管理代码。与具体硬件体系结构相关的内存管理代码位于 arch/\*/mm 目录下。

## 10. scripts目录

scripts 子目录包含用于配置核心的脚本文件。

## 11. lib目录

lib 子目录包含了核心的库代码，与处理器结构相关的库代码被放在 arch/\*/lib/目录下。

## 4.2 内核配置选项

内核配置通常是对内核支持的各个功能进行取舍配置，将配置的方案保存到 configure 文件中。在编译内核的时候，就会根据此配置对内核进行取舍编译。在源码目录下通过 make menuconfig 命令进入内核的配置界面，如图 4.3 所示。在对内核功能进行配置时，使用键盘的方向键移动光标位置，使用 Enter 键选择菜单，使用空格键修改配置选项。

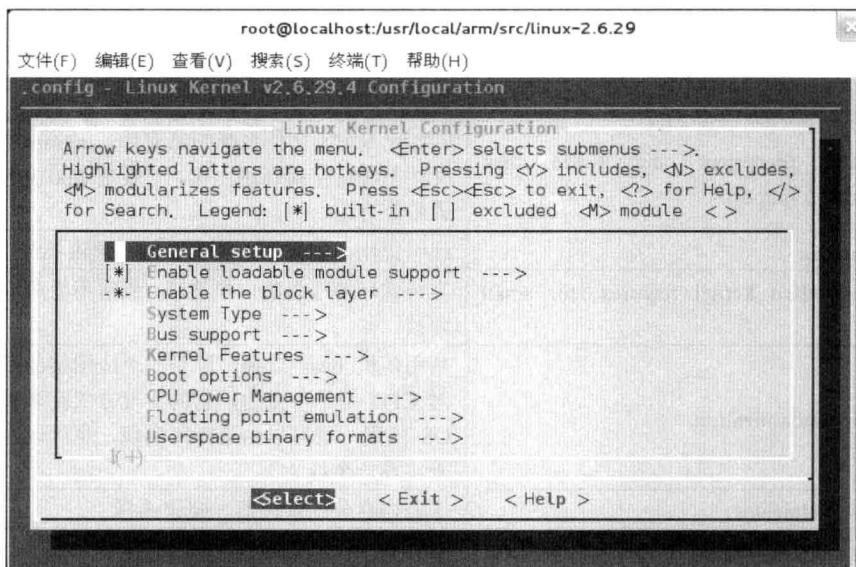


图 4.3 内核配置界面

Linux 配置选项的基本分类和含义介绍如下。

### 4.2.1 一般选项

菜单选项（General setup）的子菜单中包含一些内核通用配置选项，如表 4.1 所示。在

一般配置选项中如果对系统没有特殊要求，可以只选择 System V IPC 配置。

表 4.1 一般选项

选 项 名	说 明
Automatically append version information to the version string	自动在版本后添加版本信息，编译时需要有 perl 及 git 仓库支持，通常可以不选
Support for paging of anonymous memory (swap)	支持交换内存，通常选择
System V IPC	进程间通信，通常需要配置
POSIX Message Queues	POSIX 消息队列，通常需要配置
BSD Process Accounting	可以将行程资料记录下来，通常建议配置
Export task/process statistics through netlink	通过 netlink 接口向用户空间导出任务/进程的统计信息
Auditing support	审计支持，某些内核模块（例如 SELinux）需要配置
RCU subsystem	同步机制
Kernel .config support	提供.config 配置文件支持
Kernel log buffer size (16=>64KB, 17=>128KB)	内核日志缓冲区大小(16 代表 64KB, 17 代表 128KB)
Group CPU scheduler	CPU 组调度
Control Group support	控制组支持
Create deprecated sysfs layout for older userspace tools	为旧的用户空间工具创建过时的文件系统风格
Kernel->user space relay support (formerly relayfs)	在某些文件系统上（比如 debugfs）提供从内核空间向用户空间传递大量数据的接口
Namespace support	命名空间支持
Initial RAM filesystem and RAM disk (initramfs/initrd) support	初始化 RAM 文件系统的源文件。initramfs 可以将根文件系统直接编译进内核，一般是 cpio 文件。对嵌入式系统有用
Optimize for size	代码优化。如果不了解编译器，建议不选
Configure standard kernel features (for small systems)	为特殊环境准备的内核选项，通常不需要这些非标准内核
Disable heap randomization	禁用随机 heap (heap 堆是一个应用层的概念，即堆对 CPU 是不可见的，它的实现方式有多种，可以由 OS 实现，也可以由运行库实现，也可以在一个栈中来实现一个堆)
Choose SLAB allocator	选择内存分配管理器，建议选择
Profiling support	支持系统评测，建议不选
Kprobes	探测工具，开发人员可以选择，非开发人员建议不选

## 4.2.2 内核模块加载方式支持选项

菜单选项 (Loadable module support) 的子菜单中包含一些内核模块加载方式选项，如表 4.2 所示。如果对模块的加载方式有特殊要求，如可以强制卸载正在使用的模块的要求，那么可以配置相关的模块加载方式。

表 4.2 内核模块加载方式

选 项 名	说 明
Forced module loading	允许强制加载模块驱动
Module unloading	允许卸载已经加载的模块, 建议选择
Forced module unloading	允许强制卸载正在运行的模块, 该功能危险, 建议不选
Module versioning support	允许使用其他内核版本的模块, 建议不选
Source checksum for all modules	为所有的模块校验源码, 可以不选

### 4.2.3 系统调用、类型、特性、启动相关选项

菜单选项 (Block layer) 的子菜单中包含一些系统调用方式选项, 如表 4.3 所示。在配置内核时可以不选该菜单选项。

表 4.3 系统调用方式

选 项 名	说 明
Support for Large Block Devices and files	使用大容量块设备时选择
Support for tracing block io actions	支持块队列 I/O 跟踪
Block layer SG support v4	支持通用 scsi 块设备第 4 版
Block layer data integrity support	支持块设备数据完整性
IO Schedulers	I/O 调度器

菜单选项 (System Type) 的子菜单中包含一些系统类型选项, 在配置内核时直接选择对应的芯片类型即可。对特定的平台选择相应的支持类型。

菜单选项 (Kernel Features) 的子菜单中包含一些系统特性选项, 如表 4.4 所示。在嵌入式系统中, 一般不对这些选项进行配置。

表 4.4 系统特性

选 项 名	说 明
Preemptible Kernel	抢占式内核。建议采用
Use the ARM EABI to compile the kernel	使用 ARM EABI 编译内核
Allow old ABI binaries to run with this kernel	使内核支持旧版本的 ABI 程序
Memory model	只有 Flat Memory 供选择
Add LRU list to track non-evictable pages	对没有使用的页采用最近最少使用算法, 建议选择

菜单选项 (Boot Options) 的子菜单中包含一些系统启动选项, 如表 4.5 所示。

表 4.5 系统启动

选 项 名	说 明
(0)Compressed ROM boot loader base address	xImage 存放的基址
(0)Compressed ROM boot loader BSS address	BSS 地址
()Default Kernel command string	内核启动参数
Kernel Execute-In-Place from ROM	从 ROM 中直接运行内核, 该内核使用 make xipImage 编译
(0x00080000)XIP Kernel Physical Location	选择 XIP 后, 内核存放的物理地址
Kexec system call	Kexec 系统调用

#### 4.2.4 网络协议支持相关选项

菜单选项（Networking Support）的子菜单中包含一些网络协议支持的选项，如表 4.6 所示。基本只需要在 Networking options 子菜单中选择具体所需的网络协议即可。

表 4.6 网络协议

选 项 名	说 明
Networking options	该菜单的子菜单包含支持的各种具体网络协议，在开发中可以根据需要进行配置
Amateur Radio support	业余无线电支持，一般不选
CAN bus subsystem support	CAN 总线子系统支持
IrDA (infrared) subsystem support	红外线支持
Bluetooth subsystem support	蓝牙支持
RxRPC session sockets	RxRPC 会话套接字支持
Phonet protocols family	Phonet 协议族支持
Wireless	无线电协议支持
WiMAX Wireless Broadband support	WiMAX 无线宽带支持
RF switch subsystem support	RF 交换子系统支持
Plan 9 Resource Sharing Support (9P2000)	9 计划资源共享支持

#### 4.2.5 设备驱动支持相关选项

菜单选项（Device drivers）的子菜单中包含一些设备驱动的选项，如表 4.7 所示。重点说明了 MTD 设备相关的驱动。需要支持设备驱动时可以配置相关的选项。

表 4.7 设备驱动

选 项 名	说 明
Connector - unified userspace <-> kernelspace linker	用户空间和内核空间的统一连接器
Memory Technology Devices (MTD) support	MTD 设备支持，嵌入式系统使用
Debugging	调试功能
MTD concatenating support	连接多个 MTD 设备，例如使用 JFFS2 文件系统管理多片 Flash 的情形。只有一片 Flash 时不选
MTD partitioning support	Flash 分区支持，建议选择
MTD tests support	MTD 测试支持
RedBoot partition table parsing	使用 RedBoot 解析 Flash 分区表，如果需要读取这个分区表的信息，选择此项
Command line partition table parsing	允许通过内核命令行传递 MTD 分区表信息
ARM Firmware Suite partition parsing	使用 AFS 分区信息
TI AR7 partitioning support	AR7 分区支持
Direct char device access to MTD devices	将系统中的 MTD 设备看作字符设备进行读/写

续表

选 项 名	说 明
Caching block device access to MTD devices	文件系统挂载后，模拟块设备进行访问。常用于只读文件系统。如果是 DiskOnChip 使用 NFTL 方式
FTL (Flash Translation Layer) support	提供对 Flash 翻译层支持，可以不选
NFTL (NAND Flash Translation Layer) support	NAND Flash 翻译层支持，可以不选
INFTL (Inverse NAND Flash Translation Layer) support	提供 INFTL 支持，DiskOnChip 使用
Resident Flash Disk (Flash Translation Layer) support	提供 RFD 支持，为嵌入式系统提供类似 BIOS 功能
NAND SSFDC (SmartMedia) read only translation layer	NAND SSFDC 只读翻译层
Log panic/oops to an MTD buffer	MTD 缓冲区日志
RAM/ROM/Flash chip drivers	RAM/ROM/Flash 芯片驱动
Mapping drivers for chip access	为芯片的访问方式选择 Mapping 驱动
Self-contained MTD device drivers	自身包含 MTD 设备驱动，一般不选
NAND Device Support	NAND Flash 支持
OneNAND Device Support	One NAND 相关驱动
LPDDR flash memory drivers	LPDDR Flash 内存驱动
UBI - Unsorted block images	只提供 UBI 支持
Parallel port support	并口支持
Block devices	红外线支持
Bluetooth subsystem support	蓝牙支持
RxRPC session sockets	RxRPC 会话套接字支持
Phonet protocols family	Phonet 协议族支持
Wireless	无线电协议支持
WiMAX Wireless Broadband support	WiMAX 无线宽带支持
RF switch subsystem support	RF 交换子系统支持
Plan 9 Resource Sharing Support (9P2000)	9 计划资源共享支持

#### 4.2.6 文件系统类型支持相关选项

菜单选项 (File Systems) 的子菜单中包含一些文件系统配置的选项，如表 4.8 所示。内核移植完成后，通常需要制作文件系统，可以在此部分选择内核支持的文件系统格式。

表 4.8 文件系统

选 项 名	说 明
Second extended fs support	Ext2 文件系统支持
Ext3 journalling file system support	Ext3 文件系统支持
The Extended 4 (ext4) filesystem	Ext4 文件系统支持
Reiserfs support	Reiserfs 文件系统支持

续表

选 项 名	说 明
JFS filesystem support	JFS 文件系统支持
XFS filesystem support	XFS 文件系统支持
OCFS2 file system support	OCFS2 文件系统支持
Btrfs filesystem (EXPERIMENTAL)	Btrfs 文件系统, 不稳定, 建议不选择
Unstable disk format	
Dnotify support	文件系统变化通知机制支持
Inotify file change notification support	Inotify 是 Dnotify 的替代者, 在高版内核中默认支持
Quota support	磁盘限额支持
Kernel automounter support	自动挂载远程文件系统, 如 NFS
Kernel automounter version 4 support (also supports v3)	自动挂载远程文件系统, 对版本 4 和版本 3 都支持
FUSE (Filesystem in Userspace) support	在用户空间挂载文件系统, 建议选择
CD-ROM/DVD Filesystems	ISO 9660, UDF 等文件系统支持
DOS/FAT/NT Filesystems	FAT/NTFS 文件系统支持。如果用于访问存储设备, 并且包含像 Windows 文件时选择该选项
Pseudo filesystems	伪操作系统, 多指内存中的操作系统
Miscellaneous filesystems	杂项文件系统, 包括 ADFS、BFS、BeFS、HPFS 等, 比较少用, 建议不选
Network File Systems	网络文件系统。其中只有 NFS 在产品开发过程中用。在开发过程可以选用
Partition Types	分区类型。该菜单下提供很多中类型, 但在嵌入式产品中很少用, 建议不选
Distributed Lock Manager (DLM)	分布式锁管理器

#### 4.2.7 安全相关选项

菜单选项 (Security options) 的子菜单中包含一些安全配置选项。很少用, 建议不选。菜单选项 (Kernel hacking) 的子菜单中包含内核黑客配置选项。建议不选。菜单选项 (Cryptographic API) 的子菜单中包含内核加密算法配置选项。很少用, 建议不选。

#### 4.2.8 其他选项

菜单选项 (Bus Support) 的子菜单中包含一些总线接口支持, 嵌入式系统可以不选。菜单选项 (CUP Power Management) 的子菜单中包含电源管理选项, 嵌入式系统可以不选。菜单选项 (Floating) 的子菜单中包含一些总线接口支持, 嵌入式系统可以不选。菜单选项 (Library routines) 的子菜单中包含一些库配置选项, 主要提供 CRC 支持, 在开发通信类产品时可以选择对应的 CRC。

## 4.3 内核裁剪及编译

经过对内核的认识和对裁剪配置项的了解，接下来实际操作。针对 S3C2440 开发板进行裁剪 Linux 内核。

### 4.3.1 安装内核源代码

在前面章节中已经介绍了建立交叉编译环境。如果还没有建立编译环境，请参考相关章节。获得源码可以直接从网上下载开发板对应的源码。该源码相比 Linux 基本内核源码增加了对应平台相关的内容。将源代码压缩包复制到 /usr/local/arm 目录下，使用 tar 命令解压源码。

```
tar -jxvf linux-2.6.32.tar.bz2
```

tar 命令带上 zxvf 参数可以看到详细的解压过程，如图 4.4 所示。

```
tom@localhost:/usr/local/arm
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
linux-2.6.32/arch/arm/mach-bcmring/include/csp/
linux-2.6.32/arch/arm/mach-bcmring/include/csp/cache.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/delay.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/dmachw.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/errno.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/intchw.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/module.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/reg.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/sechw.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/stdint.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/string.h
linux-2.6.32/arch/arm/mach-bcmring/include/csp/tmrhw.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/
linux-2.6.32/arch/arm/mach-bcmring/include/mach/clkdev.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/cap.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/cap_inline.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/chipchw_def.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/chipchw_inline.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/chipchw_reg.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/ddrcreg.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/dmachw_priv.h
linux-2.6.32/arch/arm/mach-bcmring/include/mach/csp/dmachw_reg.h
```

图 4.4 内核解压过程

### 4.3.2 检查编译环境设置

源代码解压完成后，进入 linux-2.6.32 目录下，然后使用 VI 命令编辑 Mackfile。确定编译环境为 arm 交叉编译工具与本机安装的路径一致。

```
ARCH = arm
CROSS_COMPILE = /usr/local/arm/4.4.3/bin/arm-linux-
```

### 4.3.3 配置内核

使用 make menuconfig 命令进入内核配置界面，如图 4.3 所示。注意，在 linux-2.6.32 目录下，执行 make menuconfig 命令才能正确进入配置界面。下面给出一个内核的基本配置。

(1) 在一般 General setup 配置项中选择子项 System V IPC。由于要支持处理器在程序之间同步和交换信息，如果不选该项，很多程序将运行不起来，所以选择 General setup 配置项中的子项 System V IPC，其他可以不选，如图 4.5 所示。在此配置界面中还有一个选项[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support 在制作 Ramdisk 文件系统时，应该选择该选项，如图 4.6 所示。

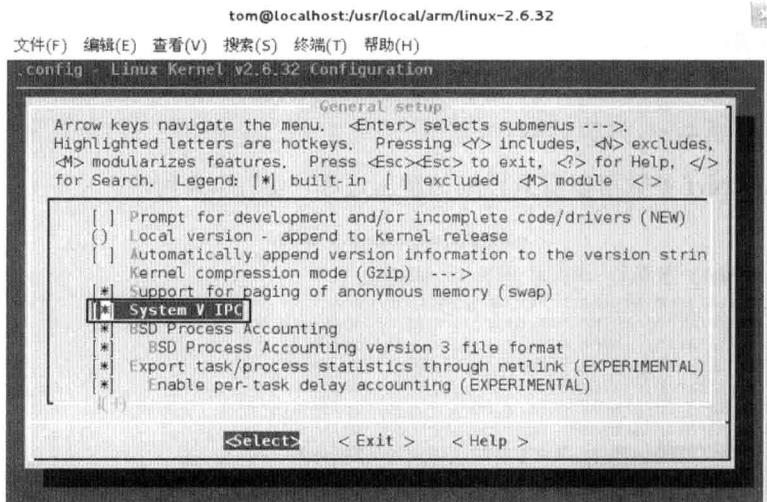


图 4.5 配置 System V IPC

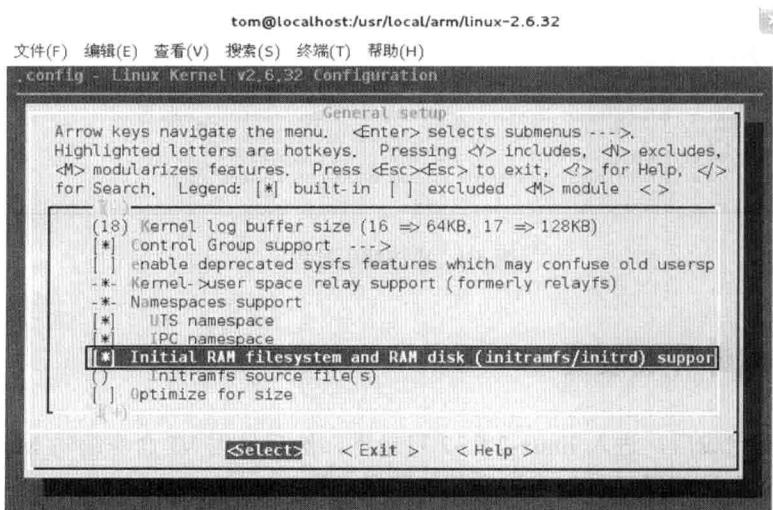


图 4.6 配置 RAM disk 支持

(2) 在模块加载方式中, 只选择子项 Module unloading, 其他可以不选。因为 Force module loading 和 Force module unloading 会造成安全隐患, 所以一般不选。Module unloading 支持动态卸载模块, 减少内核占用的资源。如图 4.7 所示为模块加载方式选项配置。

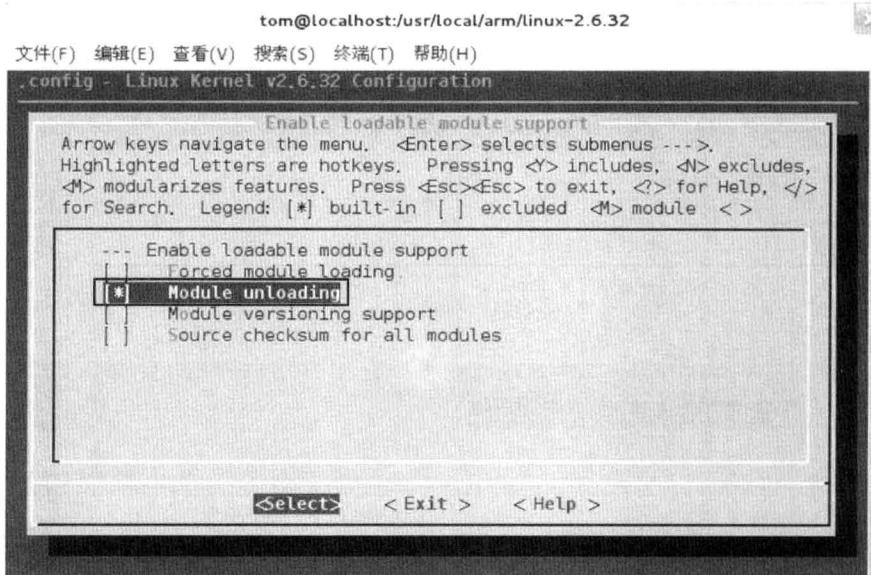


图 4.7 模块加载方式选项配置

(3) 如果系统没有对磁盘调度方式有特殊的要求, 对 block layer 可以不做任何配置。

(4) 在系统类型中选择 S3C2410 DMA support 和 Force UART FIFO on during boot process。选择 DMA support 选项是为了支持 2440 直接内存访问。选择 UART FIFO 可以支持一般的串口通信协议。如图 4.8 所示为系统类型选项配置。

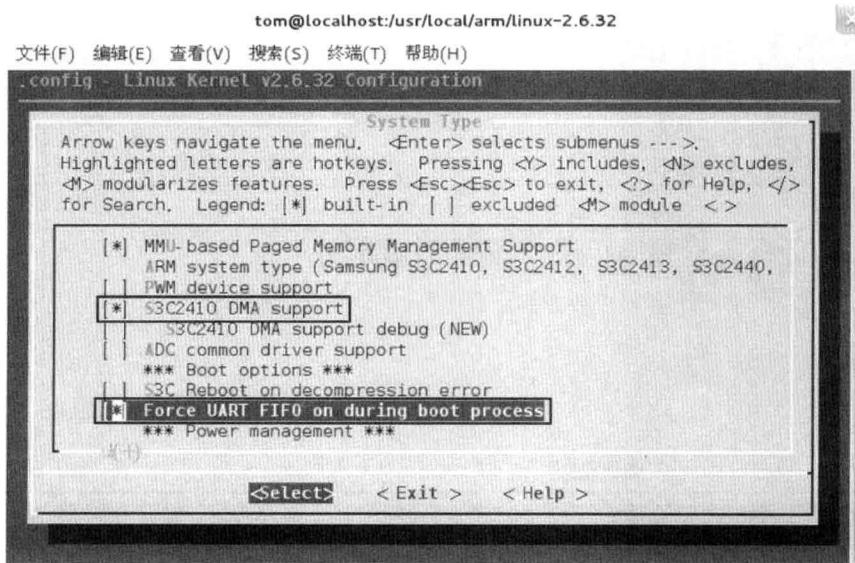


图 4.8 系统类型选项配置

选择 S3C2440 Machines 进入 S3C2440 Machines 的配置界面，选择对应开发板类型的  
支持，笔者的开发板为 Mini2440，则对应的配置如图 4.9 所示。

(5) 对于总线支持 Bus support 配置，一般情况下该选项可以不做配置，除非在开发对  
应的驱动时配置该选项。

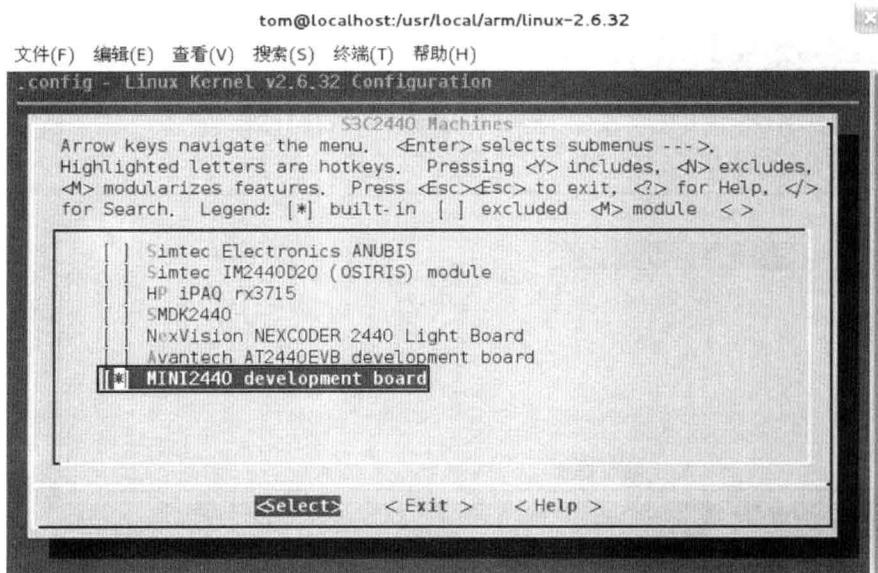


图 4.9 选择对应的开发板类型

(6) 在对系统特性选项进行配置时，建议对选项 Use the ARM EABI to compile the kernel  
进行配置，如图 4.10 所示。

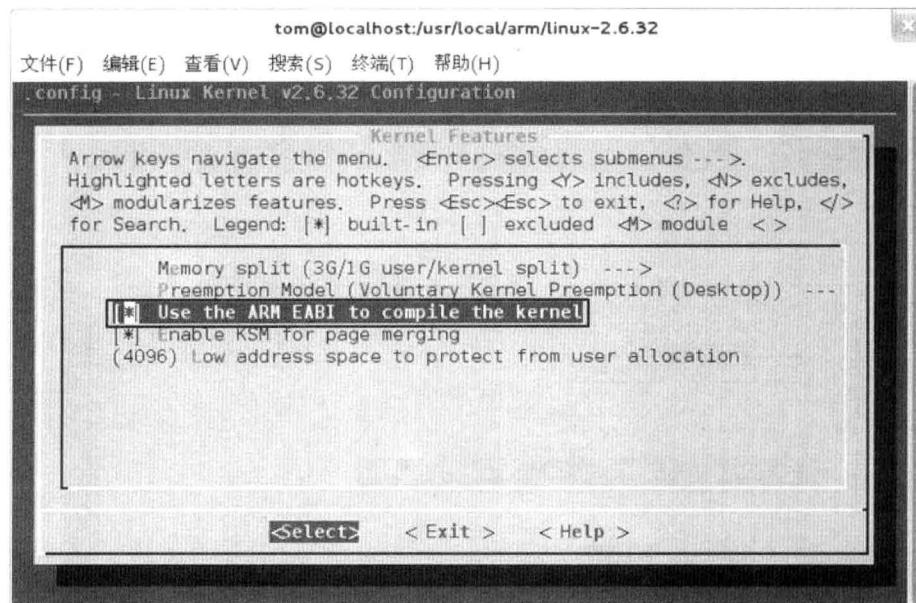


图 4.10 系统类型选项配置

**注意：**ARM EABI 有许多革新之处，其中最突出的改进就是 Float Point Performance，它使用 Vector Float Point（矢量浮点），因此可以极大提高涉及浮点运算程序的运算速度。如果编译内核的编译器支持 EABI，则在内核中也应该选择对该项的支持。

(7) 对启动参数的配置，Bootloader 启动后会将板子的信息、Ramdisk 大小、命令行字符串等信息传递给内核，然后开始启动内核。文件系统为 Ramdisk 时一般要配置该选项，对选项的具体地址和参数应该根据具体板子、内核大小、文件系统大小来定，该配置界面如图 4.11 所示。

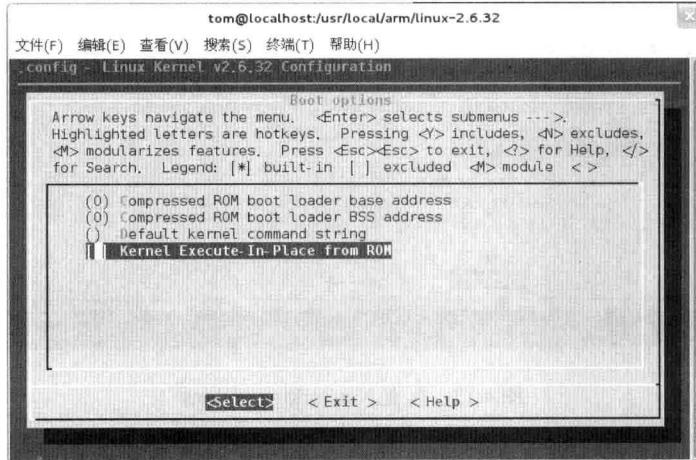


图 4.11 启动参数配置

- (8) 选项 CPU Power Management 一般不做配置。  
 (9) 选项 Floating point emulation 一般不做配置。  
 (10) 选项 Userspace binary formats，配置 Kernel support for ELF binaries，如图 4.12 所示。  
 (11) 对于电源管理选项一般不做配置。

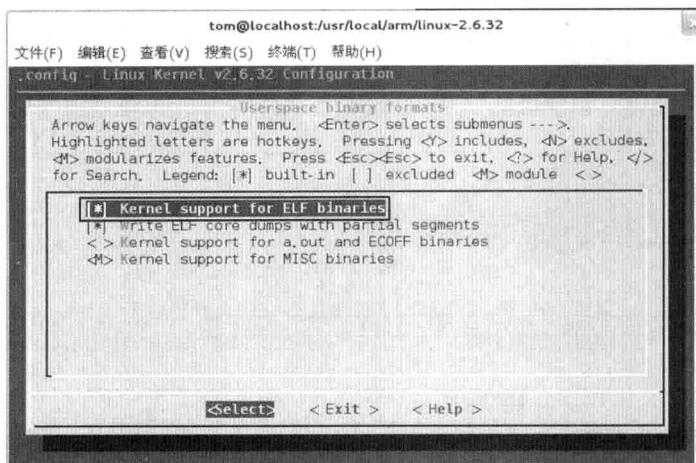


图 4.12 配置 Kernel support for ELF binaries 选项

(12) 对于网络选项的支持，配置 Networking options 中的 TCP/IP networking 和 Unix domain sockets，配置如图 4.13 所示。在 Networking support 下的其他选项，在开发对应的驱动时选择对应的选项。

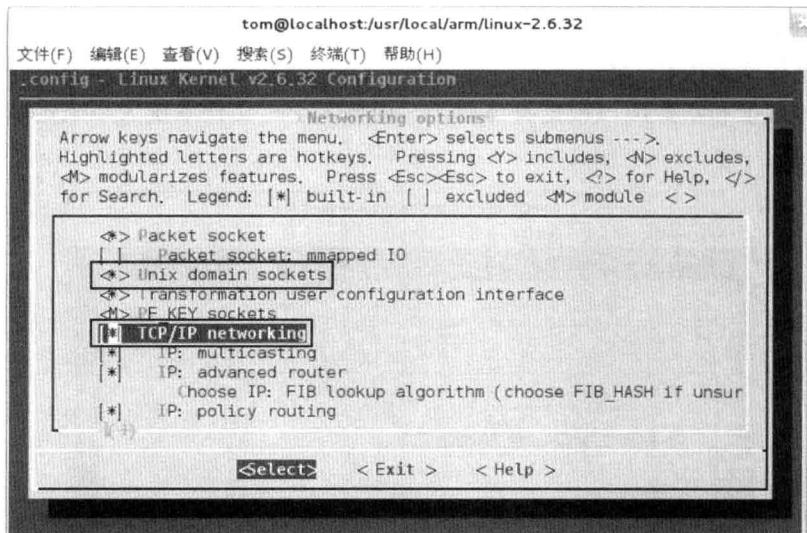


图 4.13 配置 Networking options

(13) 设备驱动选择，设备驱动选项是最复杂也是用得最多的配置选项，特别是在开发驱动和系统移植的时候。

在设备驱动选项中添加 MTD 支持，配置 MTD partitioning support 和 Direct char device access to MTDdevices。配置 MTD partitioning support 是对 Flash 分区的支持，配置 Direct char device access to MTDdevices 是支持将系统中的 MTD 设备当做字符设备进行读/写，如图 4.14 所示为驱动选项配置。

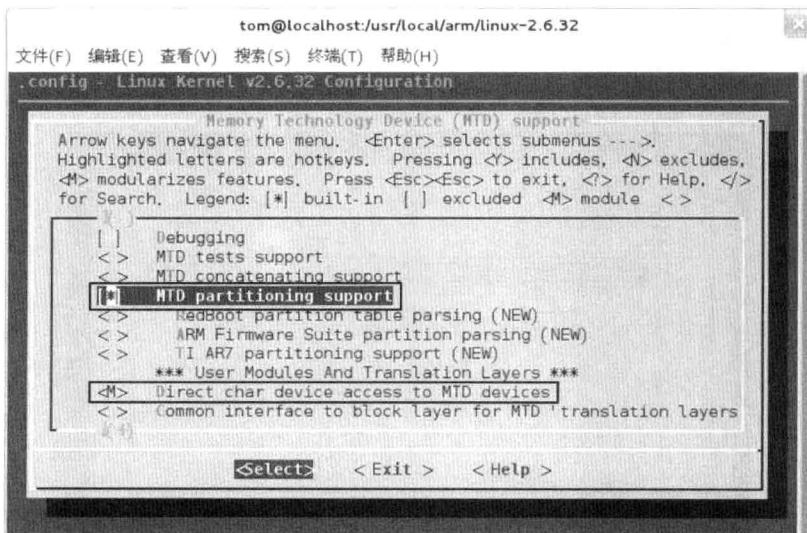


图 4.14 驱动选项配置

在网络设备驱动配置窗口中，为了使开发板支持网卡驱动选择 Ethernet (10 or 100Mbit)，如果希望支持 ppp 拨号还可以选择对 ppp 协议的支持等，可以根据具体的开发进行配置，如图 4.15 所示，进入 Ethernet (10 or 100Mbit) 配置中选择对应的网卡驱动，如图 4.16 所示。

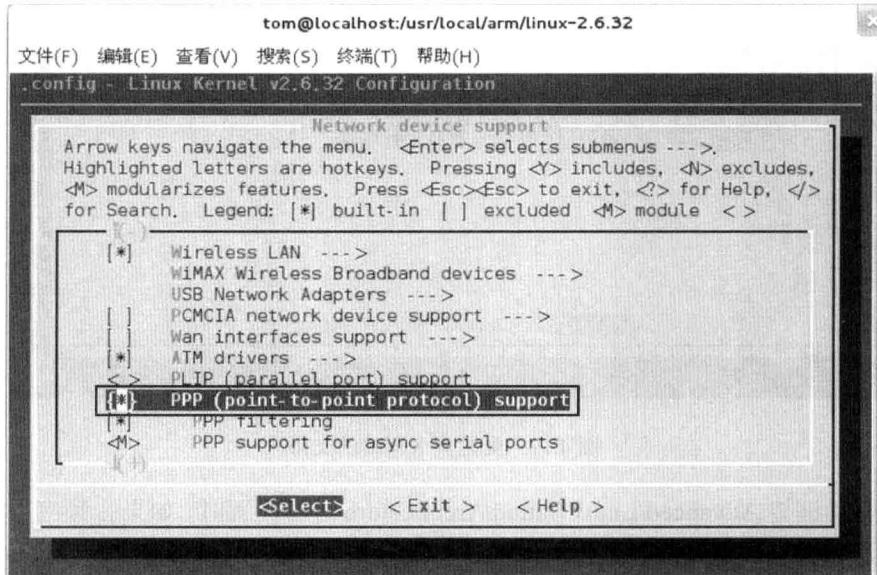


图 4.15 配置网络协议的支持

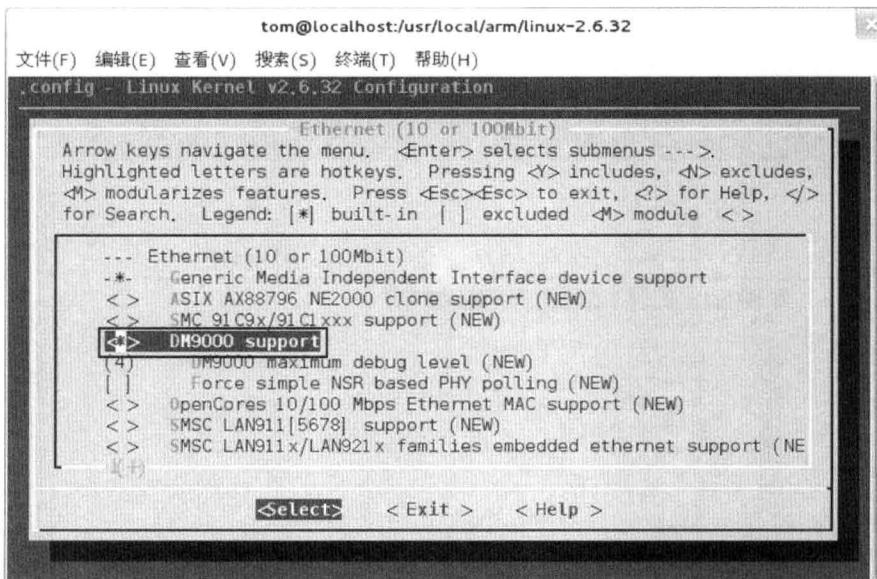


图 4.16 配置对 DM9000 支持

在音频设备驱动时，应该配置 Sound card support，在该配置窗口下有 OSS 驱动框架和 ALSA 驱动框架，其配置界面如图 4.17 所示。

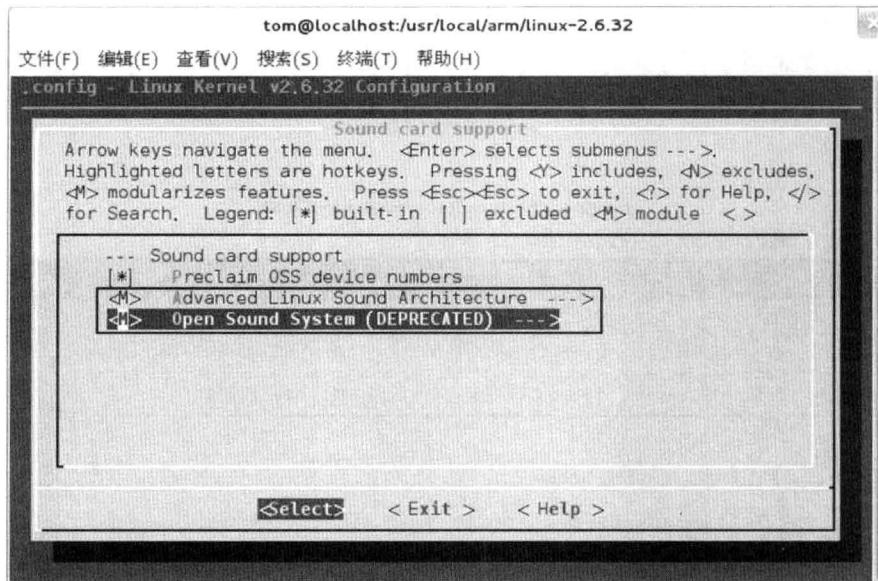


图 4.17 配置声卡驱动的支持

在对驱动框架 Advanced Linux Sound Architecture 进行配置时, 如果音频应用程序需要支持数字音频接口、混音接口, 则需要配置 OSS Mixer API 和 OSS PCM (digital audio) API, 如图 4.18 所示。另外, 还要对开发板具体的芯片支持, 如 Mini2440 采用的 UDA134x, 则还要对具体的芯片驱动进行配置, 如图 4.19 所示。当然在配置具体音频驱动支持前应该先在内核代码中添加相应的驱动。

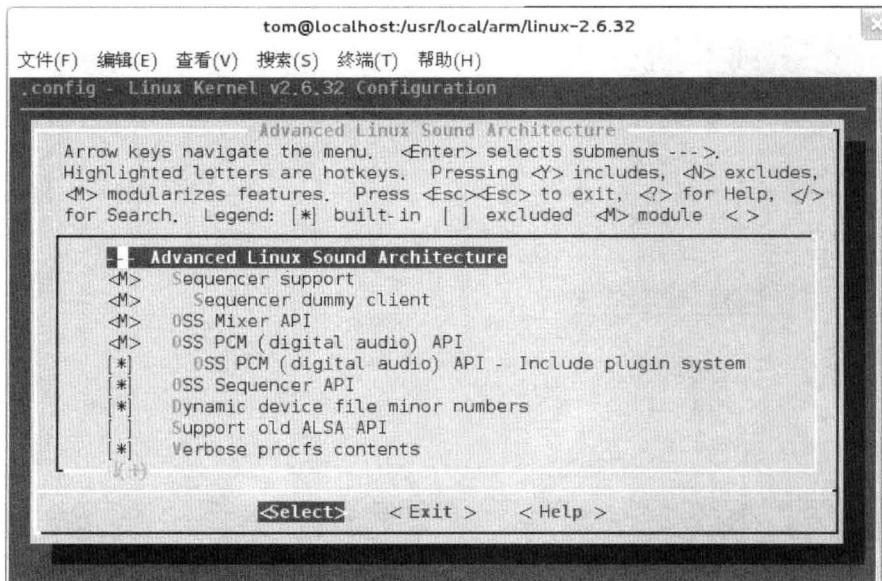


图 4.18 对数字音频接口和混音接口支持

USB 设备驱动, 也是应该要用到的内核配置选项, 在开发 USB 主机驱动时应该配置 OHCI HCD support 选项, 在开发 USB 存储设备驱动时配置 USB Mass Storage support 选项,

如图 4.20 所示。

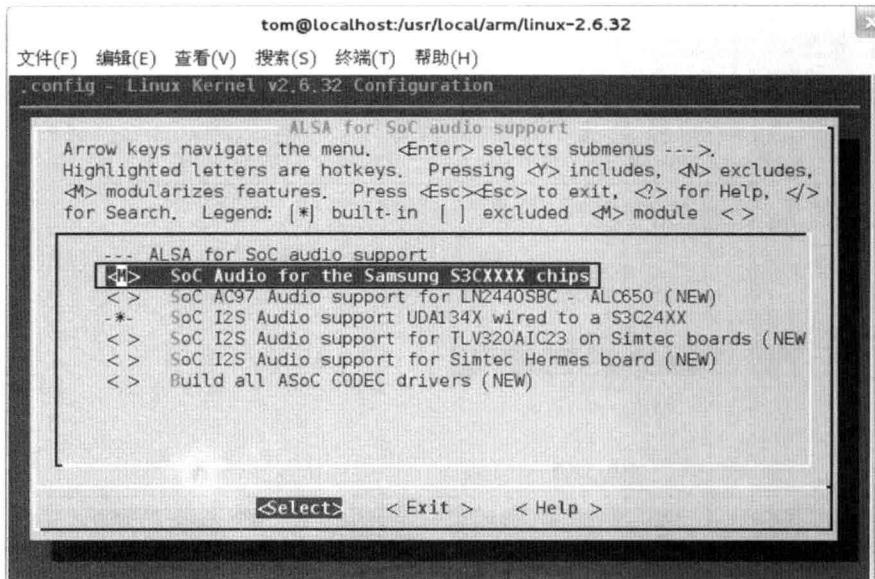


图 4.19 对具体芯片和驱动的支持

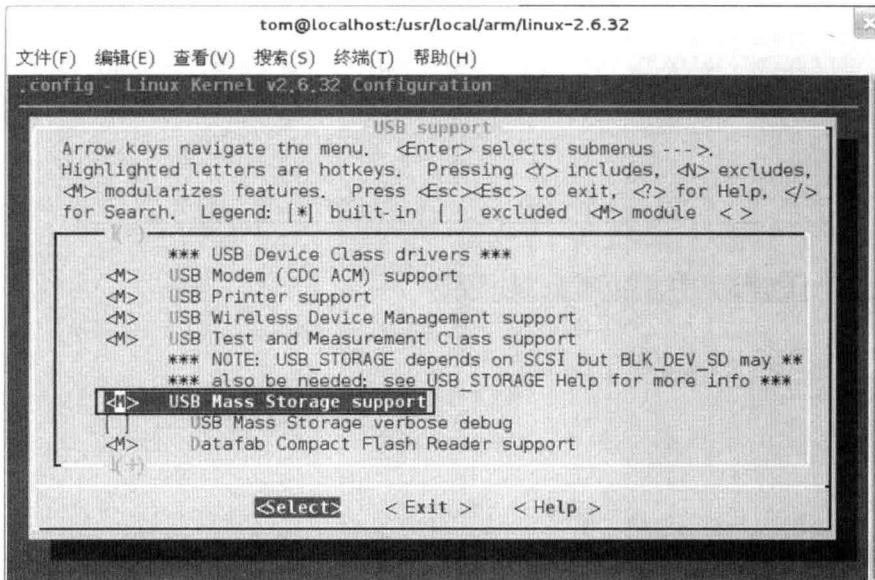


图 4.20 USB 设备驱动配置

在开发键盘、鼠标等输入设备驱动时，应该配置 HID Devices 选项。在开发 SD 卡驱动时应该配置 MMC/SD/SDIO card support 选项。

(14) 文件系统选择也是比较重要的部分，在文件系统配置选项时，应该根据所用的文件系统来添加对应的文件系统支持。笔者用到了网络文件系统和 YAFFS2 文件系统，在内核中添加对 NFS 和 YAFFS2 文件系统的支持，如图 4.21 和图 4.22 所示。

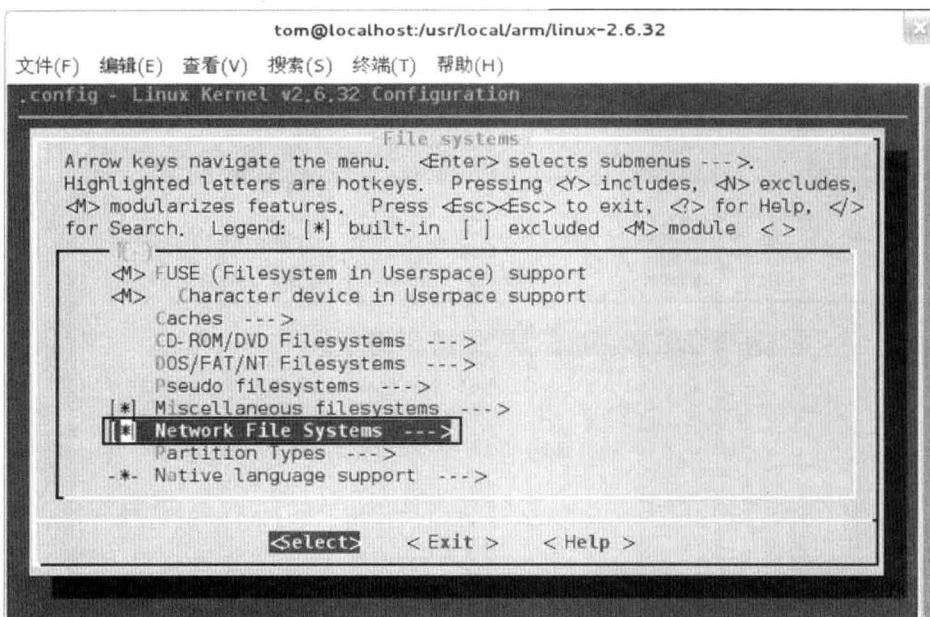


图 4.21 对 NFS 的支持

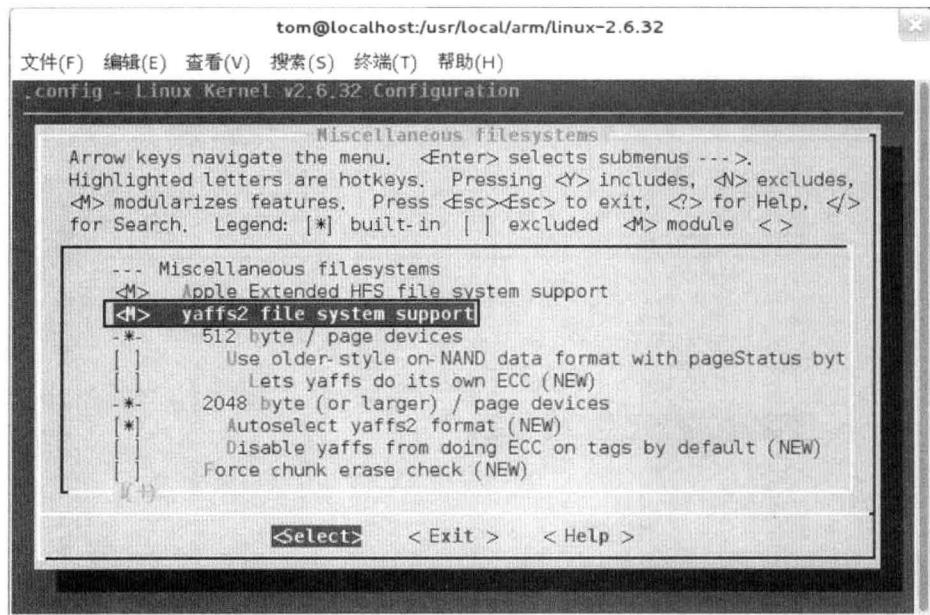


图 4.22 对 YAFFS2 的支持

如果开发板挂载其他存储设备时，这些存储设备还包含中文时，为了正确挂载这些设备，则应该在 Native language support 中添加对字符编码的设置，如图 4.23 所示为支持简体中文的配置。

(15) 剩下的内核选项一般不做配置。退出内核的配置界面并保存配置。

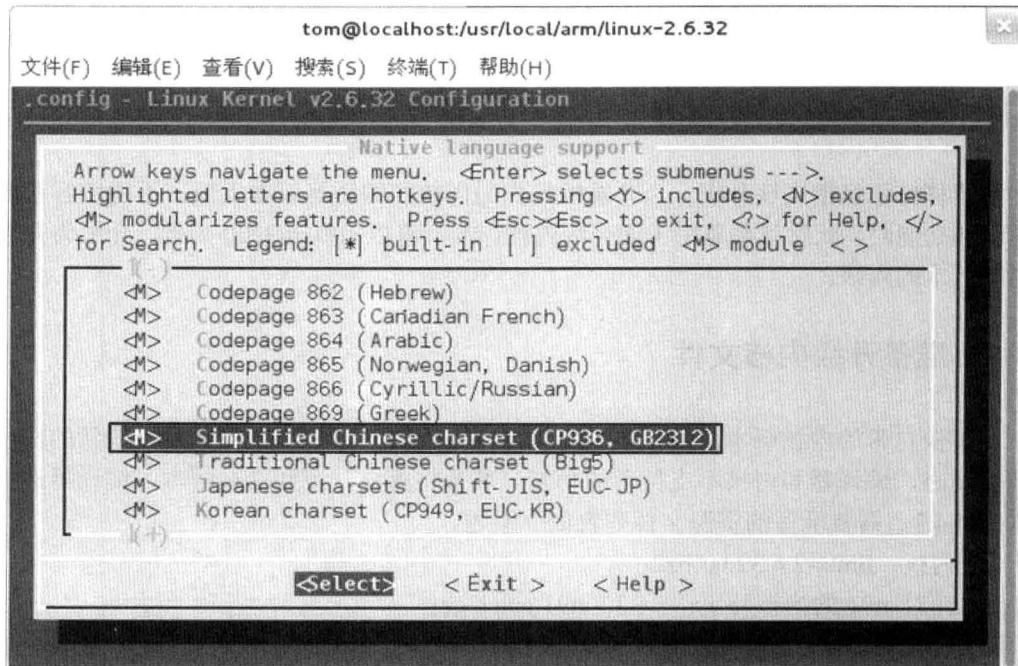


图 4.23 对字符编码的支持

#### 4.3.4 编译内核

如果是第一次编译内核就不用清理以前的映像文件。否则可以使用 make clean 命令清理以前编译的结果。在 linux-2.6.32 目录下使用 make dep 和 make zImage 命令生成内核映像文件。

```
make clean
make dep
make zImage
```

注意： make dep 命令是当程序之间有依赖关系的时候，程序发生更新时，依赖的程序会自动更新。

如果编译成功，最后会打印生成内核映像文件 zImage 及其目录。

```
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP   arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

将内核映像文件烧写到开发板的知识可以参考 2.4 节。

## 4.4 内核升级

系统移植还包括内核升级。当开发板提供的内核和编译器版本太低，不能兼容很多新的驱动和功能时，此时就要着手考虑升级内核。本节将以 at91rm9200 为例，介绍为开发板移植高版本的内核。

### 4.4.1 准备升级内核文件

开发板自带的内核版本为 Linux 2.4.27，编译器版本为 2.95.3。在开发一些新的应用程序和驱动时，编译器和内核不支持新的功能。准备将内核升级到 2.6 版本，编译器选择的版本为 4.4.3。需要准备的资源文件列表如下所述。

- 内核：linux-2.6.32.tar.bz2；
- 针对 at91 的内核补丁：2.6.32-at91.patch.gz；
- 交叉编译器：arm-linux-gcc-4.4.3.tar.gz。

### 4.4.2 移植过程

下面详细介绍下移植过程。

- (1) 将所有文件复制到工作目录下，然后解压内核文件和编译器文件。
- (2) 为内核打补丁。

```
#cd linux-2.6.32
#patch -p1 < 2.6.32-at91.patch.gz
```

- (3) 修改 Makefile，修改编译环境。

```
ARCH = arm
CROSS_COMPILE = /usr/local/arm/4.4.3/bin/arm-linux-
```

(4) 修改 machine ID。如果这一步省略，会在移植到开发板后 Bootloader 引导时出现机器 ID 错误的现象。出错的 ID 号将以十六进制给出，将其转化为十进制，替换 mach-types 文件中的对应项。这里移植后报的错误是 0xFB，即对应十进制 251。

```
#vi /usr/local/arm/linux-2.6.32/arch/arm/tools/mach-types
```

找到

at91rm9200dk	ARCH_AT91RM9200DK	AT91RM9200DK	262
--------------	-------------------	--------------	-----

将其修改为：

at91rm9200dk	ARCH_AT91RM9200DK	AT91RM9200DK	251
--------------	-------------------	--------------	-----

(5) 制作 uImage 文件。在内核目录下创建一个名为 mkimage 的文件，其内容如下：

```
/usr/local/arm/4.4.3/bin/arm-linux-objcopy -O binary -S vmlinux linux.bin
gzip -v9 linux.bin
```

```
./mkimage -A arm -O linux -T kernel -C gzip -a 0x20008000 -e 0x20008000 -d
linux.bin.gz uImage
```

(6) 对内核进行配置。执行 make at91rm9200dk\_defconfig 实际上就是完成对内核的配置。

```
#make at91rm9200dk_defconfig
```

其具体配置如下：

```
* Plug and Play support
* Block devices
RAM disk support (BLK_DEV_RAM) [Y/n/m/?] y
    Default number of RAM disks (BLK_DEV_RAM_COUNT) [16] 16
    Default RAM disk size (kbytes) (BLK_DEV_RAM_SIZE) [8192] 8192
    Initial RAM disk (initrd) support (BLK_DEV_INITRD) [Y/n/?] y
Source directory of cpio_list (INITRAMFS_SOURCE) []
Packet writing on CD/DVD media (CDROM_PKTCDVD) [N/m/y/?] n
* IO Schedulers
Anticipatory I/O scheduler (IOSCHED_AS) [Y/n/m/?] y
* Multi-device support (RAID and LVM)
* Networking support
Networking support (NET) [Y/n/?] y
    * Networking options
        Packet socket (PACKET) [Y/n/m/?] y
        Unix domain sockets (UNIX) [Y/n/m/?] y
            TCP/IP networking (INET) [Y/n/?] y
                IP: kernel level autoconfiguration (IP_PNP) [Y/n/?] y
                IP: BOOTP support (IP_PNP_BOOTP) [Y/n/?] y
                IP: TCP socket monitoring interface (IP_TCPCDIAG) [Y/n/m/?] y
        * Network packet filtering (replaces ipchains)
        * SCTP Configuration (EXPERIMENTAL)
        * QoS and/or fair queueing
        * Network testing
* Amateur Radio support
* IrDA (infrared) subsystem support
* Bluetooth subsystem support
Network device support (NETDEVICES) [Y/n/?] y
    * Ethernet (10 or 100Mbit)
        Ethernet (10 or 100Mbit) (NET_ETHERNET) [Y/n/?] y
        Generic Media Independent Interface device support (MII) [Y/?] y
        AT91RM9200 Ethernet support (ARM_AT91_ETHER) [Y/n/m/?] y
            RMII interface (ARM_AT91_ETHER_RMII) [Y/n/?] y
    * Ethernet (1000 Mbit)
    * Ethernet (10000 Mbit)
    * Token Ring devices
    * Wireless LAN (non-hamradio)
    * Wan interfaces
* SCSI device support
* Fusion MPT device support
* IEEE 1394 (FireWire) support
* I2O device support
* ISDN subsystem
* Input device support
* Userland interfaces
Mouse interface (INPUT_MOUSEDEV) [Y/?] (NEW) y
    Horizontal screen resolution (INPUT_MOUSEDEV_SCREEN_X) [1024] 1024
    Vertical screen resolution (INPUT_MOUSEDEV_SCREEN_Y) [768] 768
* Input I/O drivers
```

```

* Input Device Drivers
* Character devices
* Serial drivers
* Non-8250 serial port support
AT91RM9200 serial port support (SERIAL_AT91) [Y/n/m/?] y
    Support for console on AT91RM9200 serial port (SERIAL_AT91_CONSOLE) [Y/n/?]
y
Legacy (BSD) PTY support (LEGACY_PTYS) [Y/n/?] y
    Maximum number of legacy PTY in use (LEGACY_PTY_COUNT) [256] 256
* IPMI
* Watchdog Cards
Watchdog Timer Support (WATCHDOG) [Y/n/?] y
    Disable watchdog shutdown on close (WATCHDOG_NOWAYOUT) [Y/n/?] y
* Watchdog Device Drivers
AT91RM9200 watchdog (AT91_WATCHDOG) [Y/n/m/?] y
    * USB-based Watchdog Cards
* Ftape, the floppy tape device driver
SPI driver for AT91 processors (AT91_SPI) [Y/n/?] y
    SPI device interface for AT91 processors (AT91_SPIDEV) [Y/n/?] y
* I2C support
I2C support (I2C) [Y/n/m/?] y
    I2C device interface (I2C_CHARDEV) [Y/n/m/?] y
    * I2C Algorithms
    * I2C Hardware Bus support
    Atmel AT91RM9200 I2C Two-Wire interface (TWI) (I2C_AT91) [Y/n/m/?] y
    * Hardware Sensors Chip support
    * Other I2C Chip support
* Multimedia devices
* Digital Video Broadcasting Devices
* File systems
Second extended fs support (EXT2_FS) [Y/n/m/?] y
* CD-ROM/DVD Filesystems
* Pseudo filesystems
/proc file system support (PROC_FS) [Y/n/?] y
/dev file system support (OBSOLETE) (DEVFS_FS) [Y/n/?] y
    Automatically mount at boot (DEVFS_MOUNT) [Y/n/?] y
    Debug devfs (DEVFS_DEBUG) [N/y/?] n
Virtual memory file system support (former shm fs) (TMPFS) [Y/n/?] y
* Miscellaneous filesystems
Compressed ROM file system support (cramfs) (CRAMFS) [Y/n/m/?] y
* Network File Systems
* Partition Types
* Native Language Support
* Profiling support
* Graphics support
* Console display driver support
* Sound
* Misc devices
* USB support
Support for Host-side USB (USB) [Y/n/m/?] y
    USB verbose debug messages (USB_DEBUG) [Y/n/?] y
    * Miscellaneous USB options
* USB Host Controller Drivers
SL811HS HCD support (USB_SL811_HCD) [N/m/y/?] n
* USB Device Class drivers
USB Mass Storage support (USB_STORAGE) [N/m/y/?] n
* USB Input Devices
    * USB HID Boot Protocol drivers
* USB Imaging devices

```

```
* USB Multimedia devices
* Video4Linux support is needed for USB Multimedia device support
* USB Network Adapters
* USB port drivers
* USB Serial Converter support
* USB Miscellaneous drivers
* USB ATM/DSL drivers
* USB Gadget Support
* MMC/SD Card support
* Kernel hacking
Kernel debugging (DEBUG_KERNEL) [Y/n/?] y
* Security options
* Cryptographic options
* Library routines
CRC32 functions (CRC32) [Y/?] y
```

上面已经对内核做了详细的配置，考虑到内容比较多，省略了没有配置的选项。可以通过 make menuconfig 查看对 System Type(系统类型)的修改情况以确认进行正确的配置，如图 4.24 所示。

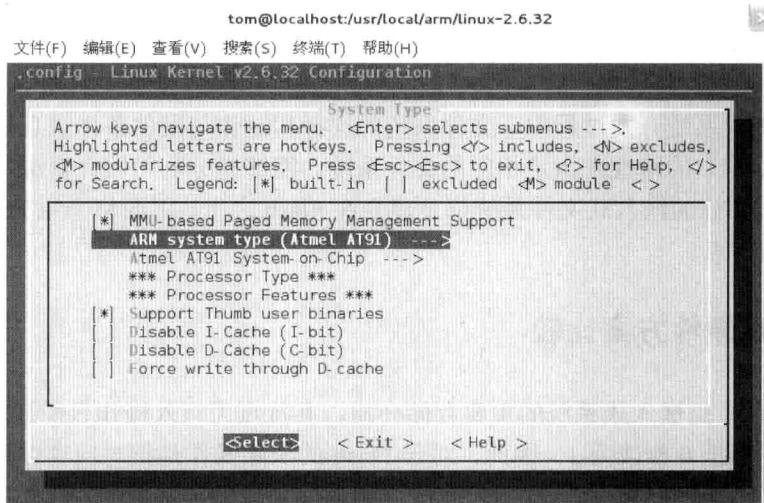


图 4.24 系统类型已经被设置为 AT91RM9200

### (7) 编译内核生成映像文件。

```
#make clean
#make dep
#./mkimage
```

## 4.5 小结

本章主要讲解 Linux 内核的目录结构、Linux 内核配置选项及裁剪内核、编译内核。最后结合实例讲解内核移植和内核升级的具体过程。在开始接触内核移植时，不提倡初学者拿到源码就直接进行裁剪配置，这样经常会由于忽略了某个选项导致移植失败。最好的办法是首先导入内核自带的配置，在这些配置的基础上根据自己的需要进行裁剪。

# 第 5 章 嵌入式文件系统制作

Linux 支持多种文件系统，包括 Ext2、Ext3、Vfat、Ntfs、Iso9660、Jffs、Romfs 和 Nfs 等。为了对各种不同类型的文件系统进行统一管理，Linux 引入了虚拟文件系统 VFS（Virtual File System），为各类文件系统提供一个统一的操作界面和应用编程接口。本章主要介绍各种嵌入式文件系统的特点和嵌入式文件系统的制作过程。嵌入式文件系统包括 Ramdisk、Jffs2、Yaffs、Cramfs、Romfs 和 Ramfs/Tmpfs。了解各种嵌入式文件系统的特点，有利于读者选择适合自己的硬件条件的文件系统。

## 5.1 文件系统选择

在进行嵌入式系统开发过程中，文件系统的选择和制作与硬件条件息息相关。根据硬件（Flash 或 RAM）的特性来指定相应的文件系统，能够充分利用硬件资源及提高系统效率。因为目前大部分的嵌入式文件系统都是建立在 Flash 之上，下面将介绍 Flash 硬件方案比较与 Flash 的特点。

### 5.1.1 Flash 硬件方案比较

Flash（闪存）是嵌入式系统的主要存储介质，其特点为写入操作只能把对应位置的 1 修改为 0，而不能把 0 修改为 1。因此，对于 Flash 的擦除操作是把对应存储块的内容恢复为 1。一般情况下，向 Flash 写入内容时，首先必须擦除对应的存储区间，擦除是以块（block）为单位进行。闪存技术主要有 NOR 和 NAND 两种技术。Flash 存储器的擦写次数是有限的，NAND 闪存设备有特殊的硬件接口和读写时序。因此，必须根据 Flash 硬件特性设计符合应用要求的文件系统。下面介绍选择硬件方案的原则。

硬件方案的总体原则是：用于数据存储采用 NAND Flash，用于代码存储采用 NOR Flash。依据这一总则，系统架构师或者硬件设计师在硬件选型阶段可以灵活地将两种闪存结合使用，用 NOR Flash 存放引导程序和根文件系统，用 NAND Flash 存放用户文件系统，使两种闪存进行优势互补。目前在手机、PocketPC、PDA 及电子词典等设备的设计中基本采用类似的方案。

在选择存储解决方案时，为了获得最高的性价比，设计师在速度、存储密度、成本、开发周期等多种因素之间进行权衡。以手机的存储解决方案为例：NOR Flash 采用支持 XIP（eXecute In Place，芯片内执行）技术能够直接运行操作系统，速度快，既简化了设计，又降低了成本，所以许多手机硬件方案都采用 NOR Flash+RAM。

NOR Flash 的缺点是存储密度较低，为了提高手机的存储容量也有方案采用 NAND

Flash+RAM。而同时追求速度和容量的手机硬件方案则会采用 NOR Flash+NAND Flash+RAM。使用 NAND Flash 的技术难度超过 NOR Flash，几乎每个 NAND 器件都存在坏的扇区，需要纠错码来维持数据。而且，在 NAND 器件上运行代码，需要存储技术驱动程序 MTD (Memory Technology Device) 技术的支持。

在表 5.1 中给出了两种 Flash 的特性进行对比，在具体的硬件选型阶段读者可以参考该表进行具体的硬件选型。

表 5.1 NOR Flash与NAND Flash比较

磁盘类型	NOR Flash	NAND Flash
使用难易程度	接口时序同 SRAM，易使用	地址/数据线复用，数据位较窄
读速度	读取速度比较快	读取速度比较慢
擦除速度	擦除速度慢，以 64KB~128KB 的块为单位	写(编程)和擦除操作的速率快，以 8KB~32KB 的块为单位
写速度	写入速度慢 (因为一般要先擦除)	写入速度快
应用场合	随机存取速度较快，支持 XIP(eXecute In Place，芯片内执行)，适用于代码存储。在嵌入式系统中，常用于存放引导程序、根文件系统等	顺序读取速度较快，随机存取速度慢，适用于数据存储 (如大容量的多媒体应用)。用于嵌入式系统中时，通常存放用户文件系统等
存储密度	单片容量较小，1KB~32MB	单片容量较大，8MB~128MB，提高了单元密度
使用成本	成本高	成本低
使用寿命	NOR 的擦写次数为十万次	NAND 闪存中每个块的最大擦写次数为一百万次
软件支持	在 NOR 器件上运行代码时不需要其他驱动程序支持	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 驱动程序支持

### 5.1.2 嵌入式文件系统的分层结构

不同类型的文件系统具有不同的特点，根据系统需求、采用的存储设备的硬件特性等，采用不同的文件系统或者文件系统组合。在嵌入式 Linux 应用中，主要的存储设备为 RAM (DRAM, SDRAM) 和 ROM (常采用 Flash 存储器)，常见的基于存储设备的文件系统类型包括 Ramdisk、JFFS2、YAFFS、Cramfs、Romfs 和 Ramfs/Tmpfs 等。下面给出 Linux 系统下文件系统的分层结构，如图 5.1 所示。

从图 5.1 中可以看出，嵌入式文件系统主要有基于 Flash 的文件系统和基于 RAM 的文件系统，接下来将分别基于这两种硬件的文件系统特点和原理进行介绍。

## 5.2 基于 Flash 的文件系统

基于 Flash 的文件系统主要包括 JFFS2、YAFFS、Cramfs 和 Romfs 等。各种文件系统

具有不同的特点，下面将分别进行介绍。

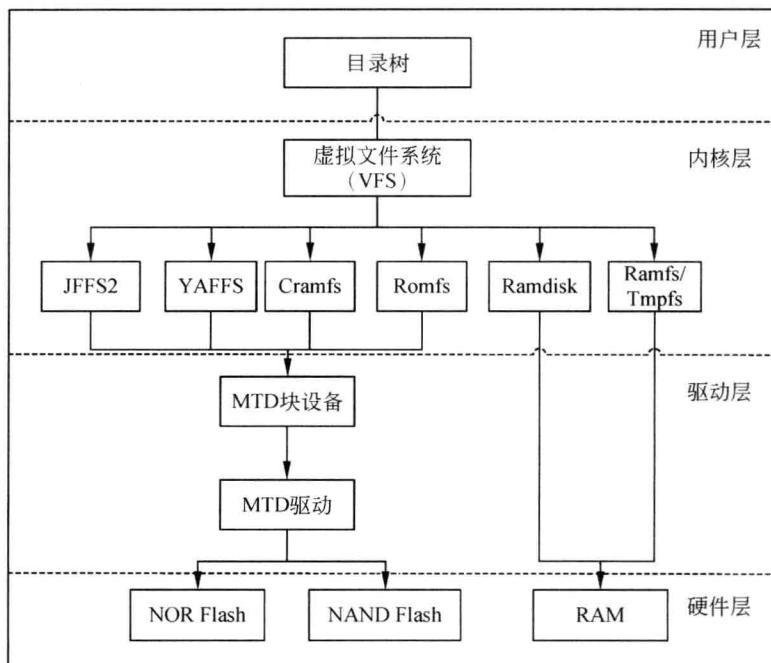


图 5.1 文件系统的分层结构

### 5.2.1 JFFS 文件系统 (Journalling Flash File System)

JFFS 系列日志文件系统包括 JFFS1、JFFS2 和 JFFS3。JFFS3 正在开发中，JFFS2 比 JFFS1 有很多改进的地方，所以目前通常使用 JFFS2。

#### 1. JFFS2的工作原理

当文件系统加载时扫描整个 Flash 的内容，将信息读入日志结点 `jffs2_raw_inode`，然后根据该信息建立文件系统。修改操作是先分配新结点 `jffs2_raw_inode`，将内容写入新结点，然后将原来的结点标记为脏数据。当系统接近满或者已满时就要进行垃圾收集，需要扫描整个 Flash 中的结点，将标记为脏的结点进行回收。

```

struct jffs2_raw_inode
{
    jint16_t magic;           //固定的魔数 magic number
    jint16_t nodetype;        //结点类型设置为 JFFS2_NODETYPE_INODE
    jint32_t totlen;          //包括有效数据在内的结点总长度
    jint32_t hdr_crc;         //jffs2_unknown_node 部分的 CRC 校验
    jint32_t ino;              //结点数
    jint32_t version;         //版本数
    jmode_t mode;             //文件类型
    jint16_t uid;              //文件的属主
    jint16_t gid;              //文件的属组
}

```

```

jint32_t isize;           //实际长度
jint32_t atime;          //上一次访问时间
jint32_t mtime;          //上一次修改时间
jint32_t ctime;          //创建时间
jint32_t offset;         //结点对应的数据在文件中的起始地址
jint32_t csize;          //压缩数据的长度
jint32_t dsize;          //压缩后数据的有效长度
uint8_t compr;           //当前使用的压缩算法
uint8_t usercompr;       //用户指定的压缩算法
jint16_t flags;          //标志位
jint32_t data_crc;       //数据 CRC
jint32_t node_crc;       //头结点 CRC
uint8_t data[0];
};

```

## 2. JFFS2的主要特点

JFFS2，日志闪存文件系统版本2（Journalling Flash FileSystem v2）。主要用于NOR Flash，基于MTD驱动层。JFFS2的主要特点如下：

- 可读写；
- 使用基于哈希表的日志结点结构，大大提高了对结点的操作速度；
- 支持数据压缩；
- 提供了“写平衡”支持；
- 支持多种结点类型（数据I结点、目录I结点等），JFFS只支持一种结点；
- 提高了对闪存的利用率，降低了内存的消耗。

 注意：“写平衡”是在垃圾收集中实现的。垃圾收集的时候会读取系统时间，通过系统时间产生一个伪随机数。使用这个伪随机数结合不同的待回收链表选择要进行回收的链表。使用写平衡策略能提供较好的写平衡效果。

JFFS2与JFFS1相比，加快了对结点的操作速度；支持更多的结点类型；提高了对闪存的利用率，降低了内存的利用率。JFFS2与JFFS3相比，根本区别在于JFFS3将索引信息放在闪存上，而JFFS2将索引信息放在内存上。

## 3. JFFS2的挂载过程

JFFS2的挂载过程主要分为4个过程：

- (1) JFFS2扫描闪存介质，检查每个结点jffs2\_raw\_inode的CRC校验码是否合法，同时分配struct jffs2\_inode\_cache和struct jffs2\_raw\_node\_ref。
- (2) 扫描每个结点的物理结点链表，标识出过时的物理结点；将每个合法的dentry结点相应的jffs2\_inode\_cache中的字段nlink加1。
- (3) 找到nlink为0的jffs2\_inode\_cache，释放对应的结点。
- (4) 释放扫描过程中的临时信息。

## 4. JFFS2的优点和缺点

JFFS2的优点有：

- 碎片收集对象是基于一个扇区而不是基于整个文件系统，删除的对象也是扇区，

因此删除操作的时间短。

- 遇到坏扇区时进行标记而使用可用扇区，延长了设备的写生命周期。

JFFS 系列文件系统存在下面的缺点：

- 文件系统已满或者接近满时，系统无法分配新的结点就必须进行垃圾收集；
- 垃圾收集就是从头开始扫描日志结点 (jffs2\_raw\_inode) 标记脏数据结点，这样使文件系统变得非常缓慢。

由于 JFFS 系列文件系统存在上述缺点，不适用于 NAND Flash。因为 NAND Flash 的容量一般较大，导致 JFFS 系列文件系统为维护日志结点所占用的内存空间迅速增大；其次，JFFS 系列文件系统在挂载时需要扫描整个 Flash 的内容，以找出所有的日志结点，建立文件结构，对于大容量的 NAND Flash 会耗费大量时间。

## 5.2.2 YAFFS 文件系统 (Yet Another Flash File System)

YAFFS 文件系统包括 YAFFS 和 YAFFS2。YAFFS/YAFFS2 是专门为嵌入式系统使用 NAND Flash 而设计的一种日志型文件系统，适用于大容量的存储设备。与 JFFS 相比，它减少了一些功能（例如不支持数据压缩），所以速度更快，挂载时间较短，对内存的占用较小。

### 1. YAFFS文件系统的特点

YAFFS/YAFFS2 自带 NAND 芯片驱动，提供了嵌入式系统直接访问文件系统的 API，这样用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统操作。当然，用户也可以通过 MTD 驱动程序来访问文件系统。

### 2. YAFFS与YAFFS2的区别

YAFFS 与 YAFFS2 的主要区别在于，YAFFS 仅支持小页 (512 Bytes) NAND Flash，而 YAFFS2 能够支持大页 (2KB) NAND Flash。另外，YAFFS2 在内存空间占用、垃圾回收速度、读/写速度等方面都有较大的改进。

### 3. YAFFS/YAFFS2的工作原理

YAFFS/YAFFS2 根据 NAND Flash 的存取特点，将文件组织成固定大小 (512 Bytes/2KB) 的数据段。对文件系统上的所有内容（比如正常文件、目录、链接、设备文件等）都统一当做文件来处理，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、Parent Object ID 等信息。根据 NAND Flash 的特点，NAND Flash 上的每一页数据都留有额外的空间用于存储附加信息，一般 NAND 驱动只占有该空间的一部分，YAFFS 文件系统正是利用了这部分空间中剩余的部分来存储文件系统相关的内容。为了了解 YAFFS 工作的原理，下面通过分析源码 (fs/yaffs\_guts.c) 认识 YAFFS 是如何进行分配和删除页（代码中表示为 chunk）的。

函数 yaffs\_AllocateChunk() 用来从 block 中分配存储空间。其代码如下：

```
static int yaffs_AllocateChunk(yaffs_Device *dev, int useReserve,
                               yaffs_BlockInfo **blockUsedPtr)
```

```

/*
dev 该指针用来记录 NAND 器件属性和使用情况，并且维护一组 NAND 操作函数指针。
useReserve 表示是否使用保留空间。
blockUsedPtr 记录 block 内还有多少空闲页信息。
*/
{
    int retVal;
    yaffs_BlockInfo *bi;
    //当 block 内的所有页都分配完时 dev->allocationBlock 的值为 -1
    if (dev->allocationBlock < 0) {
        /*在下一个 block 中进行分配*/
        dev->allocationBlock = yaffs_FindBlockForAllocation(dev);
        dev->allocationPage = 0;
    }
    if (!useReserve && !yaffs_CheckSpaceForAllocation(dev)) {
        /*没有足够的空间进行分配时，就要等到允许使用保留空间*/
        return -1;
    }
    if (dev->nErasedBlocks < dev->nReservedBlocks
        && dev->allocationPage == 0) {
        T(YAFFS_TRACE_ALLOCATE, (TSTR("Allocating reserve" TENDSTR)));
    }
    /*分配到页*/
    if (dev->allocationBlock >= 0) {
        bi = yaffs_GetBlockInfo(dev, dev->allocationBlock);
        //获得 block 信息
        retVal = (dev->allocationBlock * dev->nChunksPerBlock) + dev->
            allocationPage;                                //计算分配的页大小
        bi->pagesInUse++;                            //使用的页数加 1
        yaffs_SetChunkBit(dev, dev->allocationBlock, dev->allocationPage); //在分配的 block 中标记分配的页
        dev->allocationPage++;                      //更新分配页数加 1
        dev->nFreeChunks--;                         //更新空闲块数减 1
        /*如果 block 已满则设置其状态为 full */
        if (dev->allocationPage >= dev->nChunksPerBlock) {
            bi->blockState = YAFFS_BLOCK_STATE_FULL;
            dev->allocationBlock = -1;
        }
        if (blockUsedPtr)
            *blockUsedPtr = bi;                      //更新使用的页数
        return retVal;
    }
    T(YAFFS_TRACE_ERROR,
        (TSTR("!!!!!!!!!! Allocator out !!!!!!!!!!!!!!!" TENDSTR)));
    return -1;
}

```

函数 yaffs\_DeleteChunk() 用来释放 block 中分配的空间。其代码如下：

```

void yaffs_DeleteChunk(yaffs_Device *dev, int chunkId, int markNAND, int
lyn)
/*
dev 该指针用来记录 NAND 器件属性和使用情况，并且维护一组 NAND 操作函数指针。
chunkId 是要删除 chunk 的序号。
markNAND 在 yaffs 中使用 yaffs2 中不使用。
lyn 设置为当前行号用于调试。
*/

```

```

{
    int block;
    int page;
    yaffs_ExtendedTags tags;
    yaffs_BlockInfo *bi;

    if (chunkId <= 0)
        return;

    dev->nDeletions++;
    block = chunkId / dev->nChunksPerBlock;      //要删除 chunk 指向哪个 block
    page = chunkId % dev->nChunksPerBlock;
                                                //要删除 chunk 具体指向 block 中的哪个 page

    if (!yaffs_CheckChunkBit(dev, block, page))
                                                //检查该 block 和 page 是否为可删除
        T(YAFFS_TRACE_VERIFY,
           (TSTR("Deleting invalid chunk %d" TENDSTR),
            chunkId));

    bi = yaffs_GetBlockInfo(dev, block);           //获得 block 信息

    T(YAFFS_TRACE_DELETION,
       (TSTR("line %d delete of chunk %d" TENDSTR), lyn, chunkId));

    if (markNAND &&
        bi->blockState != YAFFS_BLOCK_STATE_COLLECTING && !dev->isYaffs2) {
        /*函数 yaffs_InitiateTags() 通过调用 memset 来设置 tags 为 0, 同时将 tags
         中的 validMarker0 设置为 0xAAAAAAA, validMarker1 设置为 0x55555555;*/
        yaffs_InitiateTags(&tags);
        tags.chunkDeleted = 1;                      //标记为被删除
        yaffs_WriteChunkWithTagsToNAND(dev, chunkId, NULL, &tags);
                                                //将标记 tags 写入块
        yaffs_HandleUpdateChunk(dev, chunkId, &tags); //更新处理
    } else {
        dev->nUnmarkedDeletions++;                //如果是 yaffs2 只增加计数
    }
    if (bi->blockState == YAFFS_BLOCK_STATE_ALLOCATING ||
                                                //状态为正在分配
        bi->blockState == YAFFS_BLOCK_STATE_FULL ||          //状态为满
        bi->blockState == YAFFS_BLOCK_STATE_NEEDS_SCANNING || //扫描 block 真正的状态
        bi->blockState == YAFFS_BLOCK_STATE_COLLECTING) {   //状态为正在回收
        dev->nFreeChunks++;                            //空闲块计数加 1
        yaffs_ClearChunkBit(dev, block, page);           //清楚块标记设置
        bi->pagesInUse--;                            //使用页计数减 1
        if (bi->pagesInUse == 0 &&
            !bi->hasShrinkHeader &&
            bi->blockState != YAFFS_BLOCK_STATE_ALLOCATING &&
            bi->blockState != YAFFS_BLOCK_STATE_NEEDS_SCANNING) {
            yaffs_BlockBecameDirty(dev, block);           //标记为脏数据
        }
    }
}
}

```

#### 4. YAFFS与JFFS的比较

YAFFS 和 JFFS 都提供了写均衡、垃圾收集等操作。同时在稳定性、垃圾收集速度、储存容量等特性方面具有以下区别：

- JFFS 是一种日志文件系统，采用日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了日志系统的部分思想，不提供日志机制，所以稳定性不如 JFFS，但是资源占用少。
- JFFS 中使用多级链表管理需要回收的脏块，采用系统生成伪随机变量计算要回收的块，这种方法能提高硬件的写均衡，在 YAFFS 中是从头到尾对块进行扫描，所以在垃圾收集上 JFFS 的速度较慢，但是能延长 NAND 器件的寿命。
- JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量大的系统。

#### 5.2.3 Cramfs 文件系统（Compressed ROM File System）

Cramfs 是一个压缩式的文件系统，不必一次性将文件系统中的全部内容解压缩到内存中，而只是在系统需要访问某个位置的数据时，先计算出该数据压缩后在 Cramfs 中所存的位置，再将该数据即时解压缩到 RAM 中，最后通过访问内存来读取文件系统中需要的数据。Cramfs 中的解压缩及解压缩之后的内存中数据存放位置，都是由 Cramfs 文件系统本身进行维护的，用户不需要了解具体的实现细节，因此这种方式增强了透明度，既方便，又节省了存储空间。

##### 1. Cramfs文件系统的特点

在 Cramfs 文件系统中，每一页（4KB）被单独压缩，可以随机页访问，其压缩比高达 2: 1，节省了嵌入式系统 Flash 存储空间，系统使用低容量的 Flash 存储相同的文件，因此降低了系统的成本。另外，它的速度快，效率高，其只读特性有利于保护文件系统遭受破坏，提高了系统的可靠性。

Cramfs 的特性如下：

- 系统访问数据时采用实时解压缩方式，其解压缩算法复杂，因此解压缩过程有延迟。
- Cramfs 的数据都是经过处理、打包的，对数据进行写操作比较困难。所以 Cramfs 不支持写操作，这一特性适合嵌入式应用中使用 Flash 存储文件系统的场合。
- 在 Cramfs 中文件最大不能超过 16MB。
- 支持组标识（gid）。mkcramfs 处理掉 gid 的高 8 位，保留 gid 的低 8 位，因此只有 gid 的低 8 位是有效的。
- 支持硬链接，但是 Cramfs 不能处理多条链接，硬链接的文件属性中，链接数始终为 1。
- Cramfs 的目录中，没有“.”和“..”这两项。因此，Cramfs 中的目录链接数通常也仅有一个。
- Cramfs 中不保存文件的时间戳（timestamps）信息。正在使用的文件由 inode 保存

在内存中，其时间可以暂时变更为最新时间，但是不会保存到 Cramfs 文件系统中。

- 当前版本的 Cramfs 只支持 PAGE\_CACHE\_SIZE 为 4096 的内核。因此，如果发现 Cramfs 不能正常读写的时候，可以修改 mkcramfs.c 中的宏定义。

 注意：对于上面特性的描述，具体细节可以查看 Cramfs.txt 文档中的 Usage Notes 描述。

## 2. Cramfs文件系统的优点和缺点

Cramfs 文件系统的优点有：压缩比较高，占用内存空间少；其缺点就是只能进行读操作，不支持写操作。

### 5.2.4 Romfs 文件系统（ROM File System）

Romfs 是一种简单的、紧凑的、只读的文件系统，不支持动态擦写保存功能，采用顺序存储方式，所有的数据包括目录、链接等都按照目录树的顺序进行存放。与 EXT2 等较大型的文件系统而言，Romfs 非常节省空间。通常 Romfs 用在嵌入式设备中作为根文件系统，或者用于保存 boot loader 以便引导系统启动。

因为 Romfs 是一种只读的文件系统，使用顺序存储方式，所有数据都是顺序存放的。它的数据存储方式决定了无法对 Romfs 进行写操作。因此 Romfs 中的数据一旦确定就无法修改，Romfs 只能作为一种只读文件系统。由于采取了顺序存放策略，Romfs 中每个文件的数据都能连续存放，读取过程中只需要一次寻址操作，就可以对整块数据进行读取，因此 Romfs 中读取数据效率很高。

Romfs 有两个结构，代码比较简单，在 romfs\_fs.h 中定义如下：

```
struct romfs_super_block {
    __be32 word0;
    __be32 word1;
    __be32 size;
    __be32 checksum;
    char name[0];      /* volume name */
};
```

结构体 romfs\_super\_block 用于识别 Romfs 文件系统，大小为 512 字节，字段 word0 的初始值为 ‘-’，‘r’，‘o’，‘m’，字段 word1 的初始值为 ‘-’，‘1’，‘f’，‘s’，通过这两个字段系统可以确定这是一个 Romfs 文件系统。字段 size 记录整个文件系统的大小，理论上 Romfs 大小最多可以达到 4GB。checksum 字段是前 512 字节的校验和，用于确认整个文件系统结构数据的正确性。前 4 个字段占 16 字节，剩下的都可以用做文件系统的卷名，如果整个首部不足 512 字节部分采用 0 填充，保证首部遵循 16 字节对齐原则。

```
struct romfs_inode {
    __be32 next;        /* low 4 bits see ROMFH_ */
    __be32 spec;
    __be32 size;
    __be32 checksum;
    char name[0];
};
```

结构体 `romfs_inode` 是 Romfs 的文件结构。`next` 字段指定下一个文件的偏移地址，该地址的后 4 位是保留的，用于记录文件模式信息，其中前两位标识文件类型，后两位标识该文件是否为可执行文件。因此 Romfs 用于文件寻址的 bit 数实际上只有 28bit，所以 Romfs 中文件大小不能超过 256MB ( $2^{28}$ )。`spec` 字段标识该文件类型，目前 Romfs 支持的文件类型包括普通文件、目录文件、符号链接、块设备和字符设备文件。`size` 字段指明文件大小；`checksum` 字段是文件名和填充字段的校验和；`name` 字段是文件名首地址，文件名长度必须保证遵循 16 字节对齐原则，不足部分可用 0 填充。

对于 Romfs 文件系统的注册过程，读者可以查看 `fs/romfs/inode.c` 文件。其注册过程类似简单设备驱动注册过程。

## 5.3 基于 RAM 的文件系统

基于 RAM 文件系统的优点就是读写速度非常快，而缺点就是当系统复位后会丢失所有数据。下面分别简单介绍基于 RAM 的文件系统的特点。

### 1. Ramdisk

Ramdisk 是划分一块固定大小的内存作分区来使用，它不是一个实际的文件系统，而是将实际的文件系统装入内存的一种策略，并且可以作为根文件系统。将一些经常被访问而又不会更改的文件（如只读的根文件系统）通过 Ramdisk 放在内存中，可以明显地提高系统的性能。

在 Linux 的启动阶段，`initrd` 提供了一套机制，将内核映像和根文件系统一起加载到内存中。在 `initrd` 机制中还会指定文件系统的起始地址、大小等参数，这些参数会通过 Bootloader 传递给内核。

### 2. Ramfs/Tmpfs

Ramfs 是 Linus Torvalds 开发的一种基于内存的文件系统，工作于虚拟文件系统（VFS）层，不能进行格式化，可以创建多个，在创建时可以指定其最大使用的内存大小（VFS 可看成是一种内存文件系统，统一了文件在内核中的表示方式，并对磁盘文件系统进行了缓冲）。

Ramfs/Tmpfs 文件系统把所有的文件都放在 RAM 中，所以读/写操作发生在 RAM 中，可以用 Ramfs/Tmpfs 来存储一些临时性或经常要修改的数据，例如/`tmp` 和/`var` 目录，这样既避免了对 Flash 存储器的读写损耗，又提高了数据读写速度。

Ramfs/Tmpfs 相对于传统的 Ramdisk 的不同之处主要在于其不能被格式化，文件系统大小可随所含文件内容大小变化。

## 5.4 文件系统的制作

5.3 节介绍了常用的文件系统的特点，以及如何根据硬件方案选择合适的文件系统。

本节将介绍如何制作选择的文件系统。另外，Busybox 集合了很多工具，编译起来也非常方便。在讲解制作文件系统的时候，也会介绍 busybox 的编译和安装过程；介绍制作文件系统时，会详细介绍 Ramdisk 和 YAFFS 2 文件系统的制作。

### 5.4.1 制作 Ramdisk 文件系统

制作根文件系统需要有如下目录：/dev、/bin、/usr、/sbin、/lib、/etc、/proc 和/sys。下面分别简单介绍各个目录中存放的文件。

(1) /dev 目录下存放的是设备文件，用于访问系统资源或设备，如串口、U 盘、硬盘、系统内存等。在 Linux 中所有的设备都被抽象成了文件，用户访问设备就像访问普通文件一样。在/dev 目录下，每个文件可用 mknod 建立。/dev 目录下主要的设备文件包括以下几个。

- /dev/console：系统控制台设备文件。
- /dev/hd IDE：接口硬盘设备文件。
- /dev/fd：软驱设备文件。
- /dev/sd：SCSI 接口磁盘驱动器文件。
- /dev/tty：设备虚拟控制台。
- /dev/ttys\*：串口设备文件。

(2) /bin、/usr/bin、/usr/sbin、/sbin 存放的是二进制可执行文件，这部分内容通常通过编译 busybox 获得。

- (3) /lib 用于存放动态链接库。

(4) /etc 是用来存放初始化脚本和其他配置文件的。启动脚本位于/etc/rc.d/init.d 中，系统最先运行的服务是那些放在/etc/rc.d 目录下的文件，运行级别在文件/etc/inittab 中指定。

(5) /proc 是用来挂载存放系统信息虚拟文件的系统，不保存在系统硬盘中，是内存的映射。它包含一些和系统相关的信息，如 CPU 的信息。

(6) /sys 该目录下安装了 2.6 内核中新出现的 sysfs 文件系统，sysfs 集成了 3 种文件系统的信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统及针对伪终端的 devpts 文件系统。sysfs 是内核设备树的一个直观反映。当一个内核对象被创建时，会在内核对象子系统中创建对应的文件和目录。

下面将详细介绍 Ramdisk 的制作过程。

#### 1. 建立根文件目录

前面提到过根文件目录主要包括/dev、/bin、/usr、/sbin、/lib、/etc、/proc、/sys、/var 和/tmp。下面给出建立根文件目录的命令：

```
#cd /usr/local
#mkdir rootfs
#cd rootfs
#mkdir bin dev etc lib proc sbin tmp usr var sys
#chmod 777 tmp
#mkdir usr/bin usr/lib usr/sbin
#mkdir var/lib var/lock var/log var/run var/tmp
#chmod 777 var/tmp
```

## 2. 编译Busybox

编译 Busybox 可以得到绝大多数目录和工具，可以简化设计和开发时间。在下载和使用 busybox 时，注意要使用稳定版本(stable)。例如，Busybox 1.21.1 是稳定版本，而 Busybox 1.21.0 是非稳定版本，建议读者在初学时使用稳定版本。

编译 Busybox 前必须对需要的工具进行配置，通过图形界面选择工具，选择的原则是尽量选择必要的工具。下面是解压和进入配置界面命令：

```
#tar jxvf busybox-1.21.1.tar.bz2
#cd busybox-1.21.1
#make menuconfig
```

(1) 进入配置界面后，依次选择 Busybox Settings-->Build Options-->，在该窗口中设置将 Busybox 编译成静态库，选择交叉编译器，如图 5.2 所示。

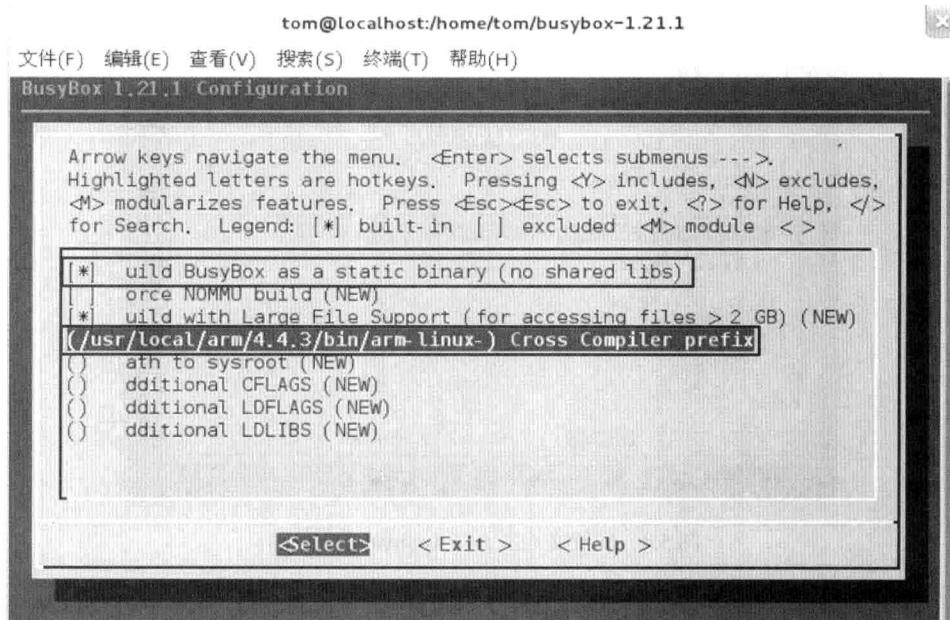


图 5.2 设置编译选项 Build Options

(2) 配置安装选项，依次选择 Busybox Settings-->Installation Options-->，进入 Installation Options 窗口后设置 busybox 的安装目录为 /usr/local/rootfs，即前面创建的根文件目录，如图 5.3 所示。

(3) 配置关于档案工具选项 (Archival Utilities)，该窗口中有常用的压缩 (bzip2)、解压 (bunzip2) 和安装软件包工具 (rpm) 等。可以选择常用的工具，也可以按照默认的选择进行配置，如图 5.4 所示。

(4) 配置核心工具选项 (Coreutils)，该窗口中包括打印日历 (cal)、修改权限 (chmod)、复制 (cp)、移动文件 (mv) 等，可以选择常用的工具，也可以按照默认的选择进行配置，如图 5.5 所示。

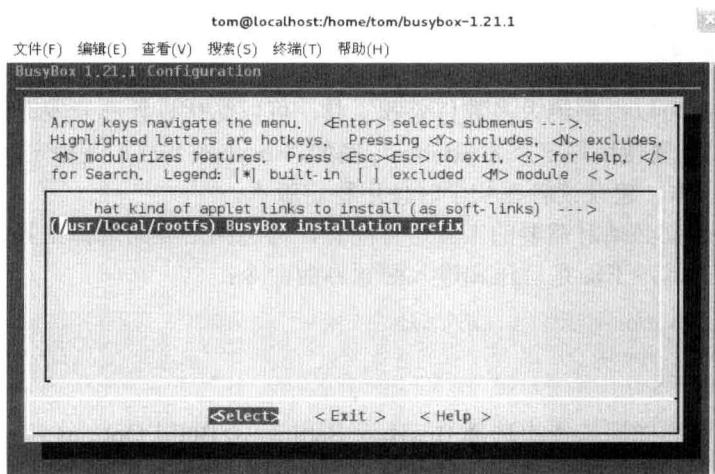


图 5.3 设置安装选项 (Installation Options)

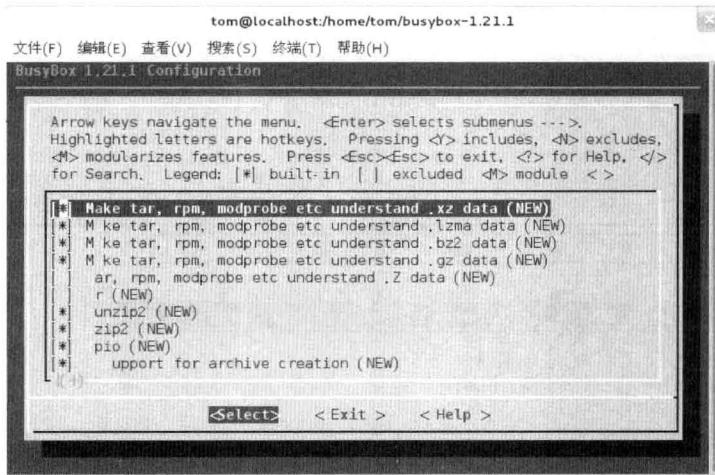


图 5.4 配置文档工具 (Archival Utilities)

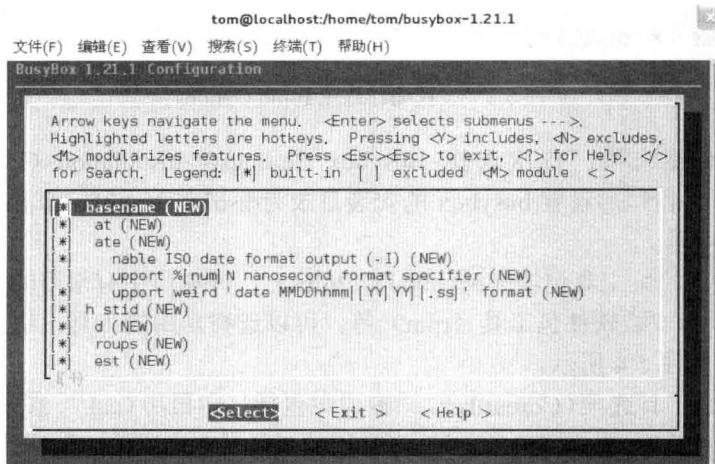


图 5.5 配置核心工具选项 (Coreutils)

(5) 配置控制台工具 (Console Utilities)，该窗口中的工具在实际中用的比较少，常用的有清除控制台 (clear)、重置 (reset) 控制台等，读者可以根据需要选择，如图 5.6 所示。

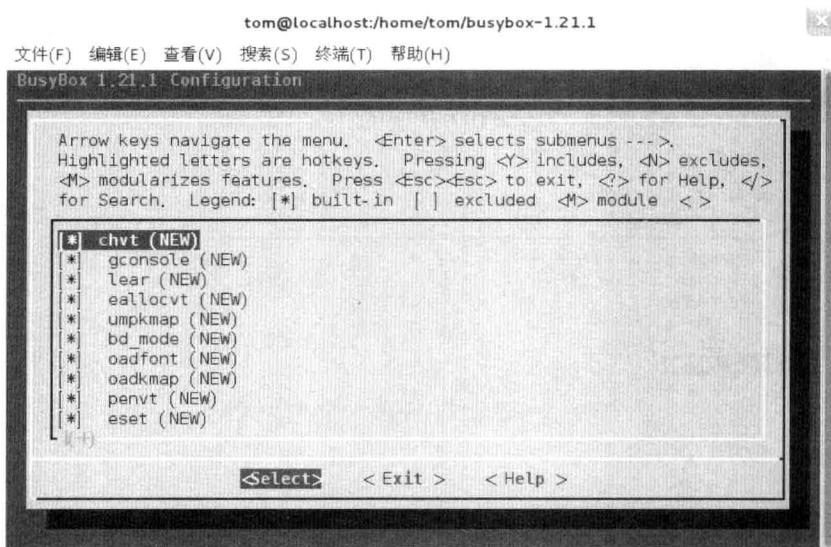


图 5.6 配置控制台工具 (Console Utilities)

(6) Debian Utilities 和 Mail Utilities，这两项工具在嵌入式系统中基本没有用到，读者可以不用配置这两个选项。

(7) 配置 Editors 时，可以只选 VI 和 diff 工具。

(8) 必须配置初始化工具 (Init Utilities)，并且在该窗口中一定要选择 Support reading an inittab file，支持 init 读取/etc/inittab 配置文件，如图 5.7 所示。

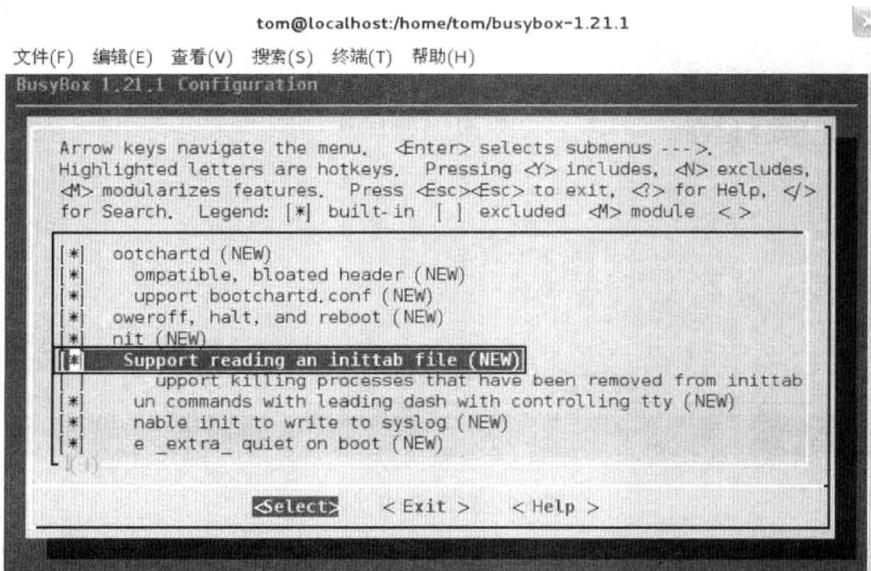


图 5.7 配置初始化工具 (Init Utilities)

(9) 必须配置网络工具（Networking Utilities），要与开发板进行通信，或者上传文件到开发板上时，需要通过网络进行传输。因此，需要设置 IP 工具（ifconfig）、文件传输工具（FTP）等，可以不用支持 IPv6、ARP 等工具，如图 5.8 所示。

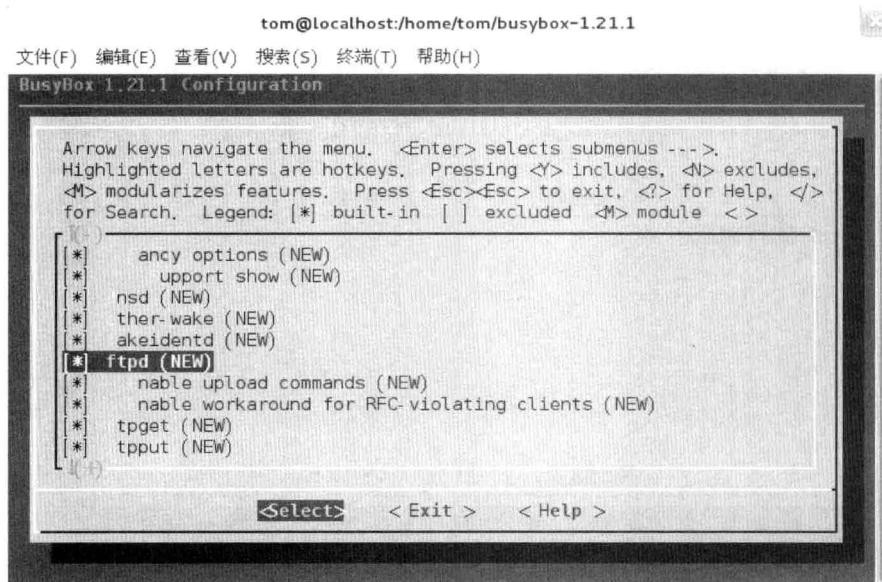


图 5.8 配置网络工具（Networking Utilities）

(10) 必须配置 Shell 工具，选择命令 Shell 进入 Shell 窗口，选择 ash，如图 5.9 所示。

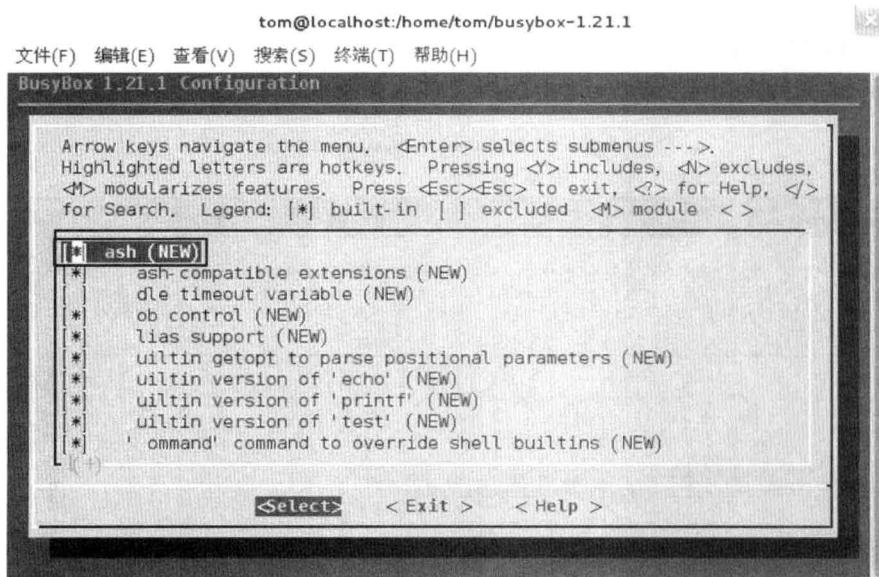


图 5.9 配置 Shell 工具

(11) 保存配置，选择 Save Configuration to an Alternate File，退出配置窗口后执行下面的命令进行编译安装 busybox 到/usr/local/rootfs 目录下。

```
#make install
```

### 3. 将交叉编译器库复制到 rootfs/lib下

(1) 将交叉编译器目录下库文件复制到 rootfs/lib 中时, 注意查看所复制的目录下是否有 libm、libpthread 等常用库。进入/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib 下, 查看目录下的库文件, 是否存在需要的库文件。

```
#cd /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib
#ls
ld-2.9.so          libmudflap.a      libresolv-2.9.so
ld-linux.so.3       libmudflap.la     libresolv.so.2
libanl-2.9.so      libmudflap.so     librt-2.9.so
libanl.so.1        libmudflap.so.0   librt.so.1
libBrokenLocale-2.9.so libmudflap.so.0.0.0 libSegFault.so
libBrokenLocale.so.1 libmudflapth.a   libssp.a
libc-2.9.so        libmudflapth.la  libssp.la
libcrypt-2.9.so    libmudflapth.so   libssp_nonshared.a
libcrypt.so.1      libmudflapth.so.0 libssp_nonshared.la
libc.so.6          libmudflapth.so.0.0.0 libssp.so
libdl-2.9.so       libnsl-2.9.so    libssp.so.0
libdl.so.2         libnsl.so.1      libssp.so.0.0.0
libgcc_s.so        libnss_compat-2.9.so libstdc++.a
libgcc_s.so.1      libnss_compat.so.2 libstdc++.la
libgomp.a          libnss_dns-2.9.so  libstdc++_pic.a
libgomp.la         libnss_dns.so.2   libstdc++.so
libgomp.so         libnss_files-2.9.so libstdc++.so.6
libgomp.so.1       libnss_files.so.2  libstdc++.so.6.0.13
libgomp.so.1.0.0   libnss_hesiod-2.9.so libsupc++.a
libgomp.spec       libnss_hesiod.so.2 libsupc++.la
libiberty.a        libnss_nis-2.9.so  libthread_db-1.0.so
libid3tag.so       libnss_nisplus-2.9.so libthread_db.so.1
libid3tag.so.0     libnss_nisplus.so.2 libts-0.0.so.0
libid3tag.so.0.3.0 libnss_nis.so.2   libts-0.0.so.0.1.1
libjpeg.so         libpcprofile.so   libts.so
libjpeg.so.62       libpng12.so     libutil-2.9.so
libjpeg.so.62.0.0  libpng12.so.0   libutil.so.1
libm-2.9.so        libpng12.so.0.35.0 libuuid.so
libmad.so          libpng.so       libuuid.so.1
libmad.so.0         libpng.so.3     libuuid.so.1.2
libmad.so.0.2.1    libpng.so.3.35.0 libz.a
libmemusage.so    libpthread-2.9.so
libm.so.6          libpthread.so.0
```

(2) 执行库文件的复制过程。复制完成后进入/usr/local/rootfs/lib 查看是否复制了需要的库文件。

```
#cd /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib
#for file in libc libcrypt libdl libm libpthread libresolv libutil
>do
>cp $file-*.* /usr/local/rootfs/lib
>cp -d $file.so.[*0-9] /usr/local/rootfs/lib
>done
#cp -d ld*.so* /usr/local/rootfs/lib
#cd /usr/local/rootfs/lib
#ls
ld-2.9.so          libcrypt.so.1   libm-2.9.so      libresolv-2.9.so
ld-linux.so.3       libc.so.6      libm.so.6       libresolv.so.2
```

```
libc-2.9.so      libdl-2.9.so   libpthread-2.9.so  libutil-2.9.so
libcrypt-2.9.so  libdl.so.2    libpthread.so.0   libutil.so.1
```

#### 4. 建立所需设备文件

需要的设备文件结点包括控制台 console、内存 mem 等。建立各个设备结点的参数包括设备类型、主设备号和次设备号。建立结点命令如下：

```
# cd /usr/local/rootfs/dev
# mknod console c 5 1
# mknod full c 1 7
# mknod kmem c 1 2
# mknod mem c 1 1
# mknod null c 1 3
# mknod port c 1 4
# mknod random c 1 8
# mknod urandom c 1 9
# mknod zero c 1 5
# for i in `seq 0 7`; do mknod loop$ i b 7 $ i; done
# for i in `seq 0 9`; do mknod ram$ i b 1 $ i; done
# ln -s ram1 ram
# mknod tty c 5 0
# for i in `seq 0 9`; do mknod tty$ i c 4 $ i; done
# for i in `seq 0 9`; do mknod vcs$ i b 7 $ i; done
# ln -s vcs0 vcs
# for i in `seq 0 9`; do mknod vcsa$ i b 7 $ i; done
# ln -s vcsa0 vcsa
```

 注意：符号``并非键盘上的单引号，而是键盘左上方的波浪号对应的键。建立完成后可以查看在/usr/local/rootfs/dev 目录下建立的设备结点有：

console	loop3	null	ram3	ram9	tty3	tty9	vcs3	vcs9	vcsa4	zero
full	loop4	port	ram4	random	tty4	urandom	vcs4	vcsa	vcsa5	
kmem	loop5	ram	ram5	tty	tty5	vcs	vcs5	vcsa0	vcsa6	
loop0	loop6	ram0	ram6	tty0	tty6	vcs0	vcs6	vcsa1	vcsa7	
loop1	loop7	ram1	ram7	tty1	tty7	vcs1	vcs7	vcsa2	vcsa8	
loop2	mem	ram2	ram8	tty2	tty8	vcs2	vcs8	vcsa3	vcsa9	

#### 5. 建立文件系统映像文件

准备目标系统启动所需要的文件 rcS、inittab 和 fstab。这 3 个文件是制作文件系统最重要的文件。下面给出各个文件的内容。

(1) /etc/init.d/rcS：挂载/etc/fstab 指定的文件系统。

```
#!/bin/sh
/bin/mount -a
```

(2) /etc/inittab：init 进程的配置文件。

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/bash
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

(3) /etc/fstab: 指定需要挂载的文件系统。

```
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
var /dev tmpfs defaults 0 0
```

## 6. 建立文件系统映像文件

建立根文件系统挂载点:

```
# mkdir /mnt/ramdisk
```

建立大小为 8192 的根文件系统:

```
# mke2fs -vm0 /dev/ram 8192
细节中打印的细节信息中包括块的个数、块的大小、结点个数等信息。
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
2048 inodes, 2048 blocks
0 blocks (0.00%) reserved for the super user
First data block=0
1 block group
32768 blocks per group, 32768 fragments per group
2048 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 30 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
```

挂载根文件系统:

```
# mount -t ext2 /dev/ram /mnt/ramdisk
```

对文件系统进行操作，将制作的文件系统复制到挂载点:

```
# cp -af /usr/local/rootfs/* /mnt/ramdisk
```

退出/mnt/ramdisk 目录才能进行卸载:

```
# cd /
```

卸载文件:

```
# umount /mnt/ramdisk
```

文件系统生成:

```
# dd if=/dev/ram of=ramdisk bs=1k count=8192
```

制作文件系统映像:

```
# gzip -v9 ramdisk
```

生成的映像文件为 ramdisk，压缩后为 ramdisk.gz。

## 7. 内核中支持RAM文件系统的初始化

在编译内核时，在 General setup 窗口中选择[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support 如图 5.10 所示，同时在 Initramfs source 中传递初始化参数：

```
initrd=0x21100000,8000000 root=/dev/ram rw init=linuxrc console=ttyS0,
115200, mem=32M
```

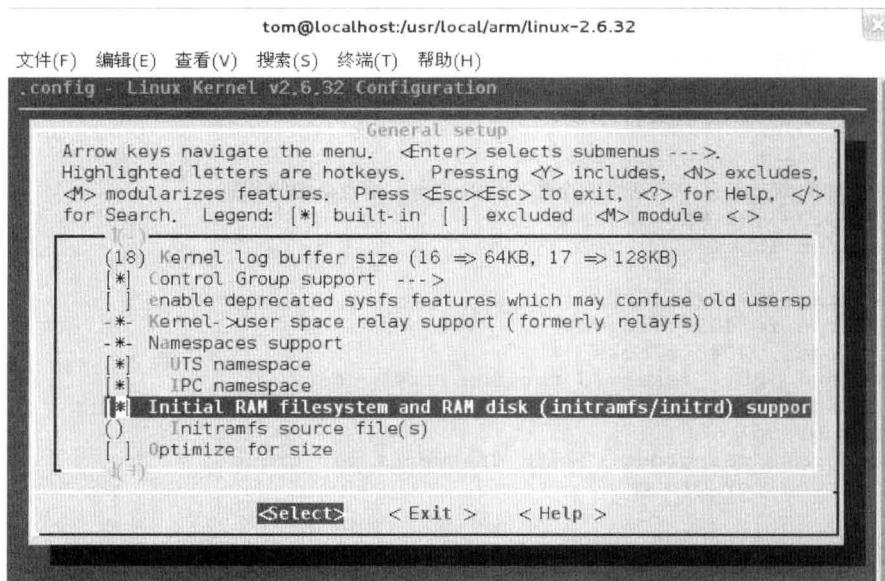


图 5.10 配置 RAM 文件系统的初始化

### 5.4.2 制作 YAFFS2 文件系统

如果开发板只有 Nand Flash，那么选择最合适的文件系统为 YAFFS 文件系统。mini2440 只有 Nand Flash 没有 Nor Flash，因此选择的文件系统为 YAFFS2 文件系统。

#### 1. 制作文件系统时准备的源代码

关于内核源代码和交叉编译器，读者可以根据自己的实际情况选择具体的对应版本，yaffs2.tar.gz 源码是必需的。

- linux-2.6.32.tar.bz2：内核源代码；
- yaffs2.tar.gz：YAFFS2 文件系统源代码；
- arm-linux-gcc-4.4.3.tar.gz：交叉编译工具；
- mkyaffs2image.tar：制作 YAFFS2 文件系统工具。

#### 2. 解压源码

解压内核源码和交叉编译器的源码，并将 yaffs2.tar.gz 复制到内核源码的 fs 目录下进

行解压。如果是第一次使用交叉编译器，那么应该在环境变量中添加交叉编译器的路径或者在/etc/profile 中添加交叉编译器路径，并重新启动计算机。文件/etc/profile 中的交叉编译器的设置，例如：

```
# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
    pathmunge /usr/local/arm/4.3.2/bin
fi
```

### 3. 修改内核顶层Makefile

在 Makefile 中设置目标平台为 arm，交叉编译器为 arm-linux。

```
# vi Makefile
ARCH ?= $(SUBARCH)
CROSS_COMPILE ?=
修改为
ARCH ?=arm
CROSS_COMPILE ?=arm-linux-
```

### 4. 修改机器码

在 vivi 启动时如果机器码与设置的不一致会出现提示，在文件 arch/arm/tools/mach-types 中进行下面的修改。

```
# vi arch/arm/tools/mach-types
s3c2440 ARCH_S3C2440 S3C2440 362
修改为
s3c2440 ARCH_S3C2440 S3C2440 782
```

### 5. 修改时钟频率

修改 arch/arm/mach-s3c2440/mach-smdk2440.c 中的时钟为 12MHz，具体修改如下：

```
static void __init smdk2440_map_io(void)
{
    s3c24xx_init_io(smdk2440_iodesc, ARRAY_SIZE(smdk2440_iodesc));
    //s3c24xx_init_clocks(16934400);
    s3c24xx_init_clocks(12000000); //将频率设置为 12MHz
    s3c24xx_init_uarts(smdk2440_uartcfgs, ARRAY_SIZE(smdk2440_uartcfgs));
}
```

### 6. 使内核支持YAFFS2

解压 yaffs2.tar.gz 后进入 YAFFS2 目录，在 YAFFS2 目录下有可执行文件 patch-ker.sh，执行如下命令：

```
# ./patch-ker.sh c /usr/local/arm/linux-2.6.29.6
```

执行该命令后，就会在 fs 的 Kconfig 和 Makefile 中增加对 YAFFS2 的编译选项的支持。在 fs/Kconfig 会自动添加：

```
# Patched by YAFFS
source "fs/yaffs2/Kconfig"
```

在 fs/Makefile 中会自动添加：

```
# Patched by YAFFS
obj-$(CONFIG_YAFFS_FS) += yaffs2/
```

 注意：这两部分内容也可以进行手动添加。添加的目的是在内核的文件系统选项中增加对 YAFFS2 的支持选项。

回到内核的一级目录下运行 make menuconfig，对内核进行配置，配置中多了对 YAFFS2 文件系统支持的选项。进入内核配置界面后，依次选择命令 File systems ---> Miscellaneous filesystems --->，进入 Miscellaneous filesystems 配置窗口，选择对 YAFFS2 文件系统的支持，如图 5.11 所示。

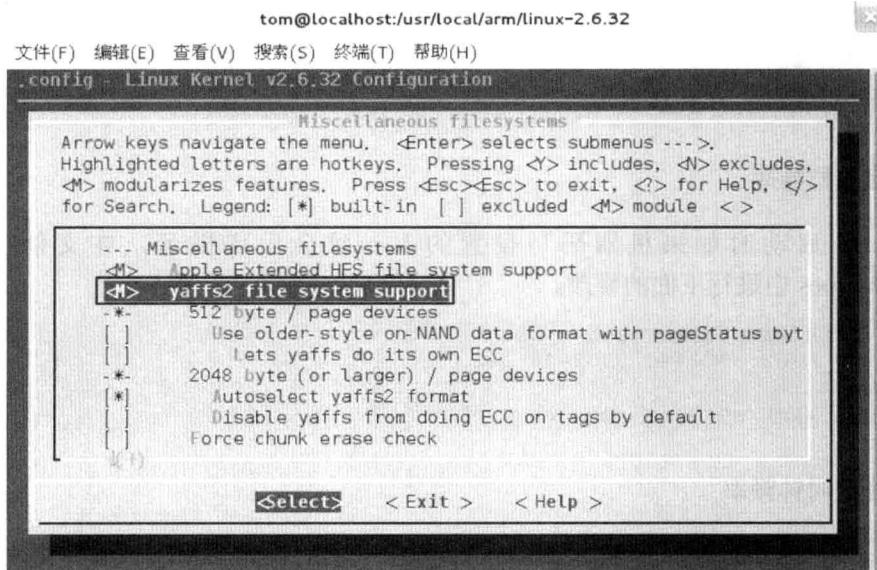


图 5.11 修改位置后的 YAFFS2 选项

## 7. 使内核支持Mini2440

在内核的 System Type-->ARM system type 选项下，选择 Samsung S3C24xx 系列，如图 5.12 所示。如果读者的开发板不是 mini2440，那么就应该选择对应的处理器类型。然后在 S3C2440 Machines 中选择 Mini2440 支持选项。

## 8. 编译内核映像文件

执行 make zImage 生成内核的映像文件，如果遇到下面的错误可以执行 make distclean 进行清理，然后重新生成映像文件。

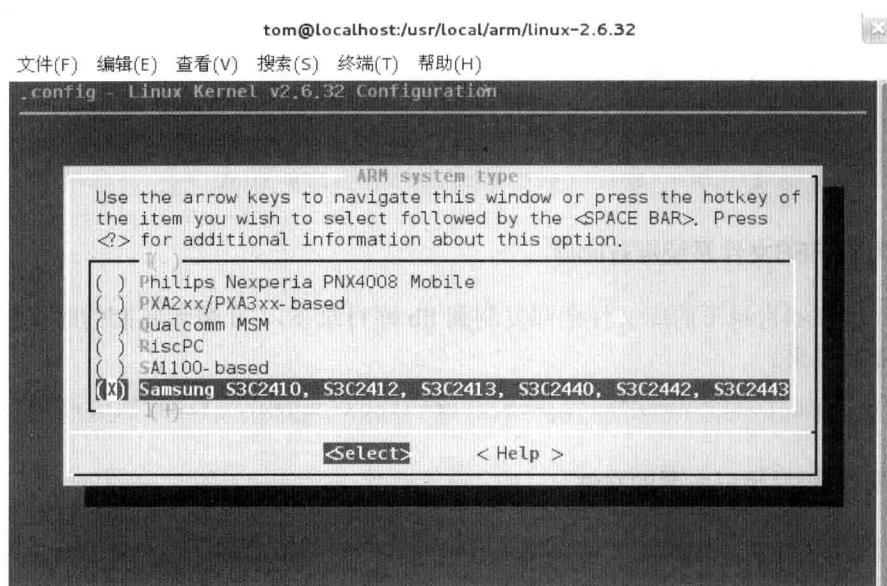


图 5.12 选择处理器类型

```
ERROR: the symlink include/asm points to asm-x86 but asm-arm was expected
      set ARCH or save .config and run 'make mrproper' to fix it
# make distclean
# make zImage
```

## 9. 编译Busybox

编译 Busybox 的配置细节可以参考 5.2.1 节。这里可以将 Busybox 交叉编译安装在 \_install 文件中，如图 5.13 所示为配置安装路径。

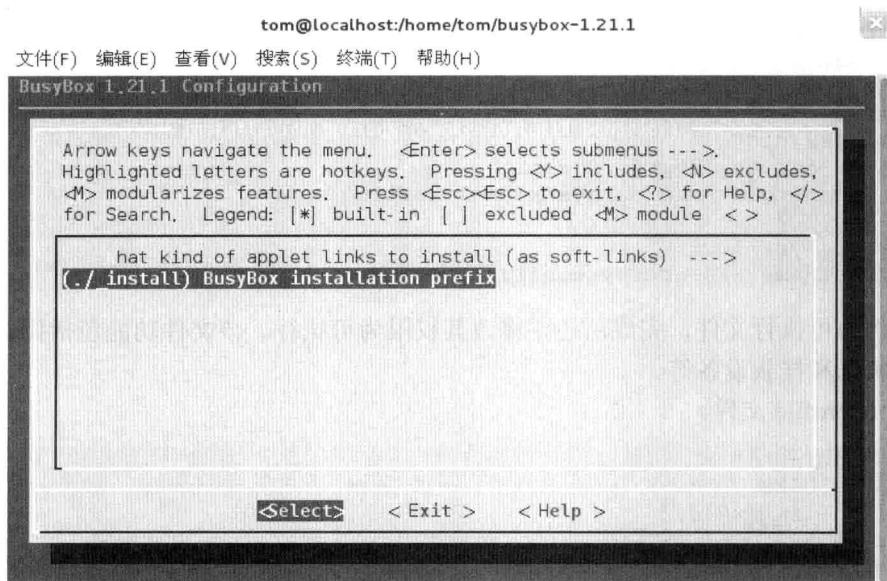


图 5.13 配置 Busybox 安装路径

在/usr/local 目录下新建 yaffs\_root 文件夹，将 Busybox 的安装目录\_install 中的文件 bin、linuxrc、sbin、usr 复制到 yaffs\_root 目录下。

```
# mkdir /usr/local/yaffs_root
# cd /usr/local/busybox-1.21.1/_install
# cp -rf bin linuxrc sbin usr /usr/local/yaffs_root
```

## 10. 为YAFFS文件系统准备lib库

将交叉编译器目录下的库文件全部复制到 lib 库目录下，-d 表示复制的时候包括链接文件一起复制过来。

```
#cp -d /usr/local/arm/4.3.2/arm-none-linux-gnueabi/libc/lib/*so* ./lib
```

## 11. 制作etc目录下必要的文件

etc 目录是文件系统中最重要的目录，系统配置的启动信息都在该目录下，下面分别给出必要的几个文件。

(1) /etc/inittab 文件。

```
::sysinit:/etc/init.d/rcS          #调用系统初始化文件
s3c2410_serial0::askfirst:-/bin/sh  #文件 drivers/serial/s3c2410.c 中指定了
串口驱动名字 s3c2410_serial
::ctrlaltdel:/sbin/reboot           #重启
::shutdown:/bin/umount -a -r         #关机
```

该文件为 init 进程的配置文件。

(2) etc/init.d/rcS 文件。

```
#!/bin/sh
PATH=/sbin:/bin:/usr/sbin:/usr/bin
runlevel=S
prevlevel=N
umask 022
export PATH runlevel prevlevel
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir -p /var/lock
/bin/hostname -F /etc/sysconfig/HOSTNAME
```

该文件为可执行文件，完成后还要修改其权限为可执行。该文件功能包括指定环境变量、运行级别和挂载设备等。

(3) etc/profile 文件。

```
USER=`id -un`
LOGNAME=$USER
PS1='[Yaffs_LiuG]# '
PATH=$PATH
HOSTNAME='/bin/hostname'
export USER LOGNAME PS1 PATH
```

如果不配置 profile，移植完文件系统后，进入系统命令行头为空，效果如下：

```
# ls
bin      home      lost+found  proc      sys      var
dev      lib       mnt        root      tmp
etc      linuxrc   opt        sbin      usr
```

配置 profile 文件后，其效果如下：

```
[Yaffs@LiuG]# ls
bin      home      lost+found  proc      sys      var
dev      lib       mnt        root      tmp
etc      linuxrc   opt        sbin      usr
```

(4) etc/sysconfig/HOSTNAME 文件。

```
Yaffs@LiuG          #指定 HOSTNAME, rcs 中调用该文件
```

(5) etc/fstab 文件。

```
proc /proc proc defaults 0 0
tmpfs /tmp tmpfs defaults 0 0
sysfs /sys sysfs defaults 0 0
tmpfs /dev tmpfs defaults 0 0
var /dev tmpfs defaults 0 0
```

该文件指明需要挂载的文件系统。

## 12. 制作YAFFS映像文件

解压制作文件系统工具 mkyaffs2image.tar，解压后该工具自动安装在目录 `usr/sbin/` 下。使用 `mkyaffs2image` 将上面制作的文件系统制作成映像文件。

```
# tar xvf mkyaffs2image.tar
# mkyaffs2image yaffs_root yaffs_root.img
```

### 5.4.3 制作 JFFS2 文件系统

制作 JFFS2 文件系统是通过工具 `mkfs.jffs2` 将文件系统目录录制成映像文件。制作工具 `mkfs.jffs2` 需要编译 `zhb` 库和 `mtd-utils`，下面详细介绍其制作过程。

#### 1. 内核配置MTD驱动支持和JFFS2支持

从图 5.1 中可以看出 YAFFS2 自带 MTD 驱动，而 JFFS2 文件系统则需要在内核中配置 MTD 驱动支持。内核也必须支持 JFFS2 文件系统。

在编译内核时选择 Device Drivers ---> Memory Technology Device (MTD) support --->, 进入 Memory Technology Device 配置窗口，如图 5.14 所示。

在编译内核时选择 File systems --->Miscellaneous filesystems--->, 进入 Miscellaneous filesystems 配置窗口，选择支持 JFFS2 文件系统，如图 5.15 所示。

#### 2. 制作工具mkfs.jffs2

制作工具 `mkfs.jffs2` 是用于制作 JFFS2 映像文件。制作 JFFS2 映像文件需要以下两个

文件：

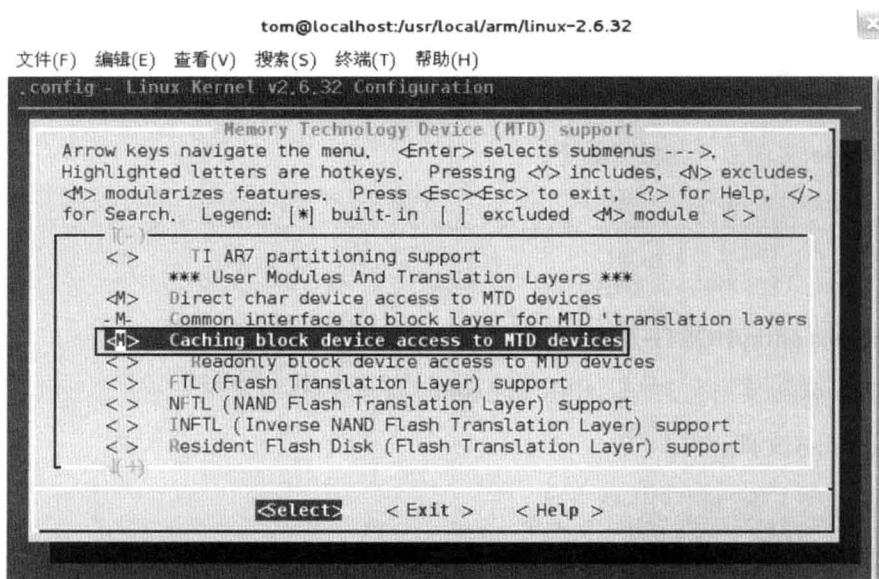


图 5.14 配置 MTD 驱动支持

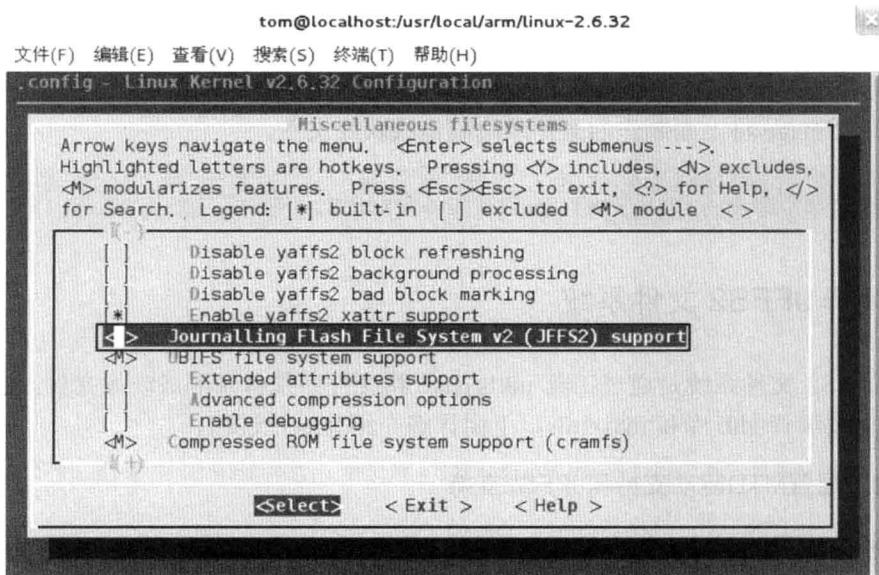


图 5.15 配置内核支持 JFFS2 文件系统

- zlib-1.2.8.tar.gz;
- mtd-utils-1.5.0.tar.bz2。

(1) 编译安装 zlib 库，用于文件压缩和解压。进入 zlib 的解压目录下，使用 configure 命令生成 Makefile。

```
# ./configure --prefix=/usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc
```

```
-shared
```

修改生成的 Makefile 如下：

```
CC=arm-linux-gcc
LDSHARED=                               arm-linux-gcc -shared
-Wl,-soname,libz.so.1,--version-script,zlib.map
CPP=arm-linux-gcc -E
AR=arm-linux-ar
RANLIB=arm-linux-ranlib
```

执行 make 和 make install 进行编译和安装。

```
#make
#make install
```

编译和安装完成后在目录 /usr/local/arm/4.4.3/arm-none-linux-gnueabi/libc/lib 下会生成动态和静态库文件 libz.a、libz.so、libz.so.1.2.8 和 libz.so.1。

(2) 编译工具 mtd-utils。进入 mtd-utils 的解压目录，执行 make 进行编译。完成编译后，新生成的工具 mkfs.jffs2、mkfs.jffs 等在目录 /usr/local/jffs\_root/tmp/mtd-utils-1.0.0 下。

```
# tar jxvf mtd-utils-1.5.0.tar.bz2
# cd mtd-utils-1.5.0
# make
# make install
```

将该路径添加到环境变量 PATH 中。

```
#PATH=$PATH: /usr/local/jffs_root/tmp/mtd-utils-1.0.0
```

(3) 制作 JFFS2 映像文件。制作 JFFS2 根文件系统的过程与其他文件系统的过程相同，制作 JFFS2 映像文件的命令如下：

```
#mkfs.jffs2 -r jffs_root -o jffs_root.jffs2 -e 0x4000 --pad=0x800000 -s 0x200
-n
```

各个参数的含义如下所述。

- -r：指定文件系统。
- -o：指定输出的映像文件名。
- -e：擦除块的大小（block size），不同的 flash，其 block size 不一样。
- --pad（-p）：指定输出文件的大小，也就是 jffs\_root.jffs2 的大小。重要的是，为了不浪费 flash 的空间，该值应该符合 flash driver 划分块的大小。
- -n：在每个擦除块中不添加 cleanmarker（消除标志）。

#### 5.4.4 其他文件系统制作

Cramfs 文件系统目录的制作方法与其他文件系统目录的制作方法相同，其映像文件的制作如下：

```
# mkfs.cramfs rootfs cram.img
```

- rootfs：Cramfs 文件系统目录；
- cram.img：生成映像文件名。

Romfs 文件系统的制作一般使用工具 genromfs。下载 genromfs-0.5.2.tar.gz 进行解压，进入解压目录进行编译生成 genromfs 工具。

```
# tar zxvf genromfs-0.5.2.tar.gz  
# cd genromfs-0.5.2  
# make  
# make install
```

使用 genromfs 工具制作 Cramfs 文件系统映像文件 romfs.img。使用如下命令可以看到制作 Cramfs 文件系统映像文件的细节。

```
# ./genromfs -V "xromfs" -f romfs.img -d ../rootfs/ -v
```

各个参数的含义如下所述。

- -V VOLUME: 指定卷标;
- -f IMAGE: 指定输出 romfs 映像的名字;
- -d DIRECTORY: 指定源目录（将该目录制作成 romfs 文件系统）;
- -v: 显示详细的创建过程。

## 5.5 小结

文件系统内容比较多，大致可以将其制作过程分为 3 个部分。工具程序，一般利用交叉编译 Busybox 获得；动态库文件，可以从交叉编译器的库目录下进行复制；配置和启动文件目录（etc），该目录下主要有 4 个文件，这 4 个文件可以参考生成旧的文件系统进行配置。文件系统的内容虽然很多，制作文件系统最简单有效的方式还是在原有的文件系统目录基础上进行增加和删除。

# 第3篇 系统移植与驱动篇

- ▶▶ 第6章 LCD 驱动移植
- ▶▶ 第7章 触摸屏驱动移植
- ▶▶ 第8章 USB 设备驱动移植
- ▶▶ 第9章 网卡驱动程序移植
- ▶▶ 第10章 音频设备驱动程序移植
- ▶▶ 第11章 SD 卡驱动移植
- ▶▶ 第12章 NandFlash 驱动移植

# 第 6 章 LCD 驱动移植

液晶显示器（Liquid Crystal Display，简称 LCD）是一种采用液晶控制透光度技术来实现色彩的显示器。它是嵌入式系统中常见的输出设备，也是现代掌上设备人机交互的重要组成部分。随着嵌入式技术的快速发展，LCD 在人们日常生活中可以说是无处不在，它在嵌入式产品中发挥着越来越重要的作用。本章将讲述怎样在开发板上移植 Linux 的 LCD 驱动程序。

## 6.1 认识 LCD 相关硬件原理

在编写 LCD 驱动程序前，驱动开发者应该对 LCD 的相关硬件原理有个大概的认识，弄清楚 LCD 显示屏相关参数的意义，如何在驱动程序中设置这些参数，从而根据相应的 LCD 显示屏型号编写相应的驱动程序。本节首先对 LCD 显示屏做大概的讲述，在对 LCD 显示屏的显示原理有一个认识之后，主要讲述 S3C2440 芯片的 LCD 控制器。

### 6.1.1 LCD 概述

LCD（液晶显示）模块满足了嵌入式系统日益提高的要求。它可以显示汉字、字符和图形，同时还具有低压、低功耗、体积小、重量轻和超薄等优点。随着嵌入式系统的应用越来越广泛，功能也越来越强大，对系统中的人机界面的要求也越来越高。在实际应用的驱使下，许多工作在 Linux 下的图形界面软件包的开发和移植工作中都涉及底层 LCD 驱动的开发问题。因此开发 LCD 驱动在嵌入式系统中得以广泛的运用。

#### 1. LCD 显示屏的分类

常见的液晶显示屏按物理结构可分为 4 种，即扭曲向列型（TN-LCD）、超扭曲向列型（STN-LCD）、双层超扭曲向列型（DSTN-LCD）和薄膜晶体管型（TFT-LCD）。其中，TN-LCD、STN-LCD 和 DSTN-LCD 的基本显示原理是一样的，只是液晶分子扭曲的角度不同而已。而 TFT-LCD 则采用与 TN 型系列 LCD 完全不同的显示方式。在写驱动程序时要根据不同的类型对 LCD 控制器进行控制。

#### 2. LCD 的常用参数

市场上的 LCD 显示屏从厂家、型号、规格等来说不尽相同，了解 LCD 的主要参数对进行 LCD 驱动开发非常有用，因为写驱动程序时就需要对 LCD 的参数进行设置。

- PPI (Pixel Per Inch) 是指每平方英寸所拥有的像素数目。由此可见，PPI 值越高就

意味着显示屏显示图像的密度越高，显示密度越高，拟真度也就越高，图像也就越清晰，显示效果也就越好。目前市面上通用的 TFT 液晶屏大部分是 100PPI 的。

- 分辨率：市面上的分辨率标准多种多样，主要有 VGA、SVGA、UXGA 和 SXGA+。其中，SXGA+所代表的显示分辨率为  $1400 \times 1050$ 。Quad-VGA 是三菱公司的一种新分辨率标准，它所代表的分辨率是  $1280 \times 960$ ，而一般标准 XGA 代表的分辨率是  $1280 \times 1024$ 。
- BPP (Bit Per Pixel)，即每个像素使用多少位来表示其颜色。比如，黑白只用 1bit (1BPP) 就可以表示黑白两种颜色，对于 4 阶灰度可用 2bit (2BPP) 来表示一个点，而对于 256 色的彩色要用 8bit (8BPP) 来表示一个点。

### 3. LCD的显示原理

对于内存中的一幅完整图像，它是怎样显示到屏幕上的呢？LCD 显示器沿用了以前 CRT 显示器的概念。一幅图像被称为一帧，每帧由多个行排列组成，每行又由多个像素组成，每个像素的色彩使用若干位数据来表示。对于单色（黑白）显示器，每个像素用 1 位来表示，称为 1BPP；对于 256 色显示器，每个像素使用 8 位来表示，称为 8BPP，以此类推。

显示器从屏幕的左上方开始，一行一行地取得每个像素的数据并显示出来，当显示到一行的最右边时，跳到下一行的最左边开始显示下一行；当显示完所有行后，重新跳到左上方开始下一帧的显示。显示器沿着“Z”字形的路线进行扫描，同时使用帧扫描信号和行扫描信号来同步每一帧和每一行。

#### 6.1.2 LCD 控制器

LCD 控制器的功能是产生控制时序和信号，从而驱动 LCD。用户只需要通过读写 LCD 控制器的一系列寄存器来完成配置。用户所要显示的内容皆是由 LCD 控制器从帧缓冲区中读出，然后再把读取到的数据发送到 LCD 驱动器进而显示到屏幕上。随着电子技术不断地发展，芯片的集成度也越来越高了，很多嵌入式处理器都集成了 LCD 控制器，如三星的 S3C2440。本节将以 S3C2440 的 LCD 控制器为例来分析如何设置 LCD 控制寄存器。

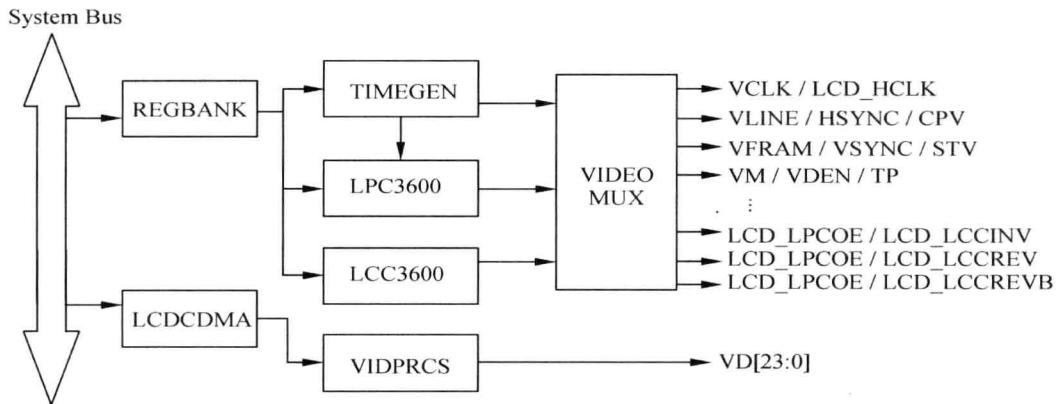
S3C2440 的 LCD 控制器由一个逻辑单元组成，它的作用是把 LCD 图像数据从一个位于系统内存的 buffer 传送到一个外部的 LCD 驱动器。LCD 控制器使用一个基于时间的像素抖动算法和帧速率控制思想，可以支持单色，2-bit per pixel (4 级灰度) 或者 4-bit-pixel (16 级灰度) 屏，并且它可以与 256 色 (8BPP) 和 4096 色 (12BPP) 的彩色 STN-LCD 连接。

LCD 控制器还支持 1BPP、2BPP、4BPP、8BPP 的调色板 TFT 彩色屏，并且支持 64K 色 (16BPP) 和 16M 色 (24BPP) 非调色板真彩显示。LCD 控制器可以编程满足不同的需求，如水平或者垂直方向的像素数目、数据接口的数据线宽度、接口时序和刷新速率等。

#### 6.1.3 LCD 控制器方块图

图 6.1 描述了 S3C2440 的 LCD 控制器结构。由于工作原理不同，LCD 控制器的接口

时序分为 STN 和 TFT 两种。S3C2440 的 LCD 控制器可以同时支持 STN 和 TFT 的 LCD 显示屏，根据实际需要对控制器进行不同的设置可以产生不同的时序。



LPC3600 是针对 LTX3500Q1-PD1 或 TS350Q1-PD2 的控制时序  
LCC3600 是针对 LTX3500Q1-PE1 或 TS350Q1-PE2 的控制时序

图 6.1 S3C2440 的 LCD 控制器结构

S3C2440 LCD 控制器被用来传送视频数据和生成必要的控制信号，比如 VFRAME、VLINE、VCLK 和 VM 等。除了控制信号外，S3C2440 还作为视频数据的数据端口，它们是如图 6.1 所示的 VD[23:0]。LCD 控制器由 REGBANK、LCDDMA、VIDPRCS、TIMEGEN 和 LPC3600 组成。

REGBANK 由 17 个可编程的寄存器组和一块  $256 \times 16$  的调色板内存组成，这些是用来配置 LCD 控制器的。LCDDMA 是一个专用的 DMA，它能自动地把在内存中的视频数据传送到 LCD 驱动器。通过使用这个 DMA 通道，视频数据在不需要 CPU 干预的情况下显示在 LCD 屏上。VIDPRCS 接收从 LCDDMA 到来的数据，将数据转换为合适的数据格式，比如说 4/8 位单扫，4 位双扫显示模式，然后通过数据端口 VD[23:0] 传送视频数据到 LCD 驱动器。TIMEGEN 由可编程的逻辑组成，支持不同的 LCD 驱动器接口时序和速率的需要，TIMEGEN 块可以产生 VFRAME、VLINE、VCLK 和 VM 等时钟信号。

具体的数据流描述如下：

LCDDMA 中存在 FIFO 存储器。当 FIFO 为空，或者部分为空的时候，LCDDMA 请求从存储器中取得数据，是用突发的存储传输模式取得数据的（每一个突发请求，连续的取 4 个字，即 16bytes，在总线传输过程中，不允许总线控制权交给另一个总线控制）。当传输请求被存储控制器中的总线仲裁器接收后，将会产生连续的 4 个字的数据传输从系统内存到内部的 FIFO。FIFO 的总共大小为 28 个字，由 12 个字的 FIFOL 和 16 个字的 FIFOH 分别组成。S3C2440 用 2 个 FIFO 支持双扫显示模式。假如是单扫模式，只有一个 FIFO 会被用到。

### 6.1.4 LCD 控制器操作

S3C2440 的 LCD 控制器分 STN 控制和 TFT 控制，目前市面上主流的 LCD 为 TFT-LCD，因

此本节将基于 TFT-LCD 介绍 LCD 控制器的使用。对于 STN-LCD 所涉及的操作与此类似。

首先来了解一下视频数据在内存中是怎么存储的。

显示屏上每个像素的色彩由 3 个部分组成，即红、绿、蓝，这就是我们经常说的三基色。这 3 种颜色混合可以表示人眼所能识别的所有颜色。如果每种颜色用 8bit 来表示，则每种颜色可分为 256 阶色，那么一个像素点就可以用 24bit（24BPP）来表示，每个像素就有 16M 阶。S3C2440 的 TFT LCD 控制器支持 1、2、4、8BPP 调色板彩色模式及 16BPP、24BPP 无调色板真彩模式。下面分别介绍各显示模式下图像数据的存储格式。

### 1. 24BPP显示

24BPP 显示模式使用 24 位表示一个像素点，每种颜色用 8 比特位来表示。LCD 控制器从内存中获得某个像素的 24 位颜色值后直接通过 VD[23:0]数据线发送给 LCD 驱动器，像素值与 VD[23:0]引脚的对应关系如表 6.1 所示。

表 6.1 像素值与VD[23:0]引脚的对应关系

VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
RED	7	6	5	4	3	2	1	0																	
GREEN									7	6	5	4	3	2	1	0									
BLUE																		7	6	5	4	3	2	1	0

为了方便 DMA 传输，在内存中使用 4 字节来表示一个像素，其中有一位是无效的。可以通过编程来选择最低字节无效还是最高字节无效。内存中像素的排列格式分两种情况，即高位无效和低位无效，如表 6.2 和表 6.3 所示，图 6.2 为像素在屏幕上的排列情况。

表 6.2 24BPP内存中像素的排列格式 (BSWP=0, SHSWP=0, BPP24BL=0)

	D[31:24]	D[23:0]
000H	Dummy Bit	P1
004H	Dummy Bit	P2
008H	Dummy Bit	P3
...		

表 6.3 24BPP内存中像素的排列格式 (BSWP=0, SHSWP=0, BPP24BL=1)

	D[31:8]	D[7:0]
000H	P1	Dummy Bit
004H	P2	Dummy Bit
008H	P3	Dummy Bit
...		

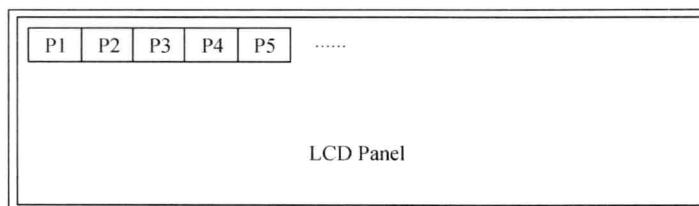


图 6.2 24BPP 显示模式时像素在屏幕上的排列情况

## 2. 16BPP

16BPP 模式用 16 位来表示一个像素点的值。这 16 位数据的格式分为两种，即 5:6:5 和 5:5:5:1，前者使用高 5 位表示红色，中间 6 位表示绿色，低 5 位表示蓝色；后者用高 15 位表示红、绿、蓝 3 种颜色，每种颜色用 5 位表示，最低位表示透明度。

内存数据与像素位置的关系如表 6.4 所示。NC 表示没有连接；5:5:5:1 格式下“I”表示透明度。

表 6.4 16BPP 像素值与 VD[23:0] 引脚的对应关系

(5:6:5)																									
VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
RED	4	3	2	1	0	NC										NC									
GREEN									5	4	3	2	1	0											
BLUE																		4	3	2	1	0			
(5:5:5:1)																									
VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
RED	4	3	2	1	0	I	NC										NC							NC	
GREEN									4	3	2	1	0	I											
BLUE																		4	3	2	1	0	I		

4 字节可以表示两个像素，使用高 2 字节还是低 2 字节来表示第一个像素也是可以编程选择的，如表 6.5 和表 6.6 所示，图 6.3 为 16BPP 模式时像素在屏幕上的排列情况。

表 6.5 16BPP 内存中像素的排列格式 (BSWP=0, SHSWP=0)

	D[31:16]	D[15:0]
000H	P1	P2
004H	P3	P4
008H	P5	P6
...		

表 6.6 16BPP 内存中像素的排列格式 (BSWP=0, SHSWP=1)

	D[31:16]	D[15:0]
000H	P2	P1
004H	P4	P3
008H	P6	P5
...		

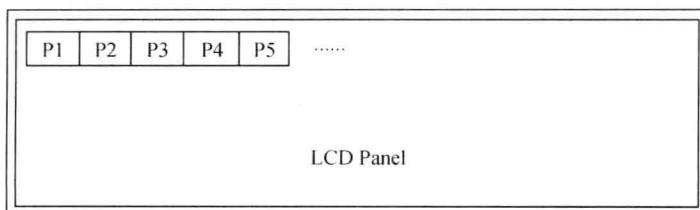


图 6.3 16BPP 显示模式时像素在屏幕上的排列情况

### 3. 8BPP

8BPP 显示模式使用 8 位来表示一个像素点，然而对 3 种基色平均下来，每种基色只能使用不到 3 位的数据来表示，即每种基色最多不过 8 阶，这不能表示更丰富的色彩。

4 字节可以表示 4 个 8BPP 的像素，字节与像素的对应顺序也是可以编程选择的，如表 6.7 和表 6.8 所示，图 6.4 为 8BPP 模式时像素在屏幕上的排列情况。

表 6.7 8BPP 内存中像素的排列格式 (BSWP=0, SHSWP=0)

	D[31:24]	D[23:16]	D[15:8]	D[7:0]
000H	P1	P2	P3	P4
004H	P5	P6	P7	P8
008H	P9	P10	P11	P12
...				

表 6.8 8BPP 内存中像素的排列格式 (BSWP=0, SHSWP=1)

	D[31:24]	D[23:16]	D[15:8]	D[7:0]
000H	P4	P3	P2	P1
004H	P8	P7	P6	P5
008H	P12	P11	P10	P9
...				

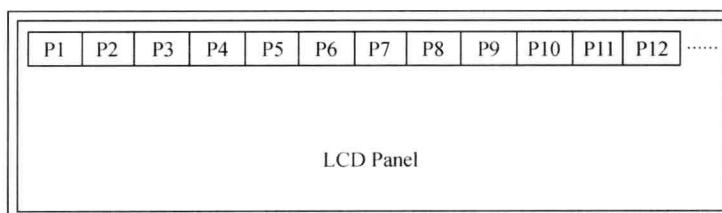


图 6.4 8BPP 显示模式时像素在屏幕上的排列情况

### 4. 256 色调色板

为了解决 8BPP 模式下显示能力弱的问题，需要使用调色板。每个像素对应的 8 位数据不是用来表示 RGB 3 种基色，而是表示它在调色板中的索引值；要显示这个像素时，使用这个索引值从调色板中取得其 RGB 值。这里说的调色板一块内存，可以对每个索引值设置颜色，可以使用 24BPP 或 16BPP。S3C2440 为 TFT 显示器提供 256 色调色板，调色板是一块  $256 \times 16$  的内存，使用 16BPP 的格式来表示 8BPP 模式下各索引值的颜色。8BPP 模式下每种基色只能使用不到 3 位，所以每种基色最大的索引值为 8，只能表示 256 阶中的 8 个阶值。这样，即使使用 8BPP 的显示模式，最终出现在 LCD 数据总线上的仍是 16BPP 的数据。

调色板中数据存放的格式与上面所描述的 16BPP 显示模式相似，也分两种格式，即 5:6:5 和 5:5:5:1。调色板中数据的格式及其与 LCD 数据线的对应关系如表 6.9 所示。

表 6.9 调色板中数据的格式及其与LCD数据线的对应关系

5:6:5 格式:

INDEX\Bit Pos	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Address
00H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000400
01H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000404
...																	
FFH	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D0007FC
Number of DV	23	22	21	20	19	15	14	13	12	11	10	7	6	5	4	3	

5:5:5:1 格式:

INDEX\Bit Pos	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Address
00H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000400
01H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000404
...																	
FFH	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D0007FC
Number of DV	23	22	21	20	19	15	14	13	12	11	10	7	6	5	4	3	

说明: 0x4D000400 是调色板的起始地址; VD18、VD10 和 VD2 有同样的输出值; .DATA[31:16]是无效的。

现在, 读者基本知道视频数据在内存中的存储格式了。下面再来了解一下这些数据是如何显示到 LCD 屏的, 这就要了解 LCD 的时序, 下面开始介绍 TFT-LCD 时序。

TFT-LCD 时序图如图 6.5 所示。

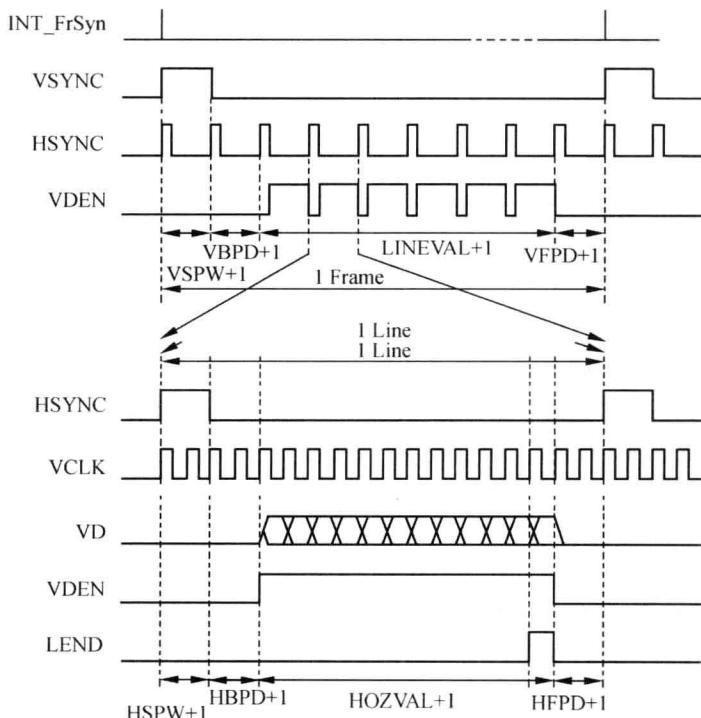


图 6.5 TFT-LCD 时序图

图 6.5 是 TFT 屏的典型时序。其中 VSYNC 是帧同步信号，VSYNC 每发出 1 个脉冲，意味着新的 1 屏视频资料开始发送。而 HSYNC 为行同步信号，每个 HSYNC 脉冲都表明新的 1 行视频资料开始发送。而 VDEN 则用来表明视频资料的有效，VCLK 是用来锁存视频资料的像素时钟。

在帧同步及行同步的头尾都必须留有回扫时间，例如对于 VSYNC 来说，前回扫时间就是  $(VSPW+1) + (VBPD+1)$ ，后回扫时间就是  $(VFPD+1)$ ；HSYNC 亦类同。这样的时序要求是当初 CRT 显示器由于电子枪偏转需要时间，但后来成了实际上的工业标准，甚至于后来出现的 TFT 屏为了在时序上与 CRT 兼容，也采用了这样的控制时序。

### 6.1.5 LCD 控制寄存器

S3C2440 提供了 LCD 控制器，其中有 17 个控制寄存器，包括 LCDCON1~LCDCON5 及 LCDSADDR1~LCDSADDR3 等，下面将分别介绍。

#### 1. LCD 控制寄存器 LCDCON1

LCDCON1 用于选择 LCD 类型、设置像素时钟、使能 LCD 输出信号等，其格式如表 6.10 所示。

表 6.10 LCDCON1 控制格式

功 能	位	描 述	初 始 状 态
LINECNT (只读)	[27:18]	每输出一个有效行其值减 1	0
CLKVAL	[17:8]	用于设置 VCLK 的值 STN: $VCLK=HCLK / (CLKVAL \times 2)$ TFT: $VCLK=HCLK / ((CLKVAL+1) \times 2)$	0
MMODE	[7]	用于设置 VM 信号的反转效率，只用于 TFT 屏	0
PNRMODE	[6:5]	设置 LCD 的类型 00: 4 位双扫描 (STN) 01: 4 位单扫描 (STN) 10: 8 位单扫描 (STN) 11: TFT 屏	0
BPPMODE	[4:1]	选择 BPP 模式，对于 TFT 屏 1000: 1BPP 1001: 2BPP 1010: 4BPP 1011: 8BPP 1100: 16BPP 1101: 24BPP (STN)	0
ENVID	[0]	信号输出使能位 0: 禁止, 1: 使能	0

## 2. LCD控制寄存器LCDCON2

用于设置垂直方向各信号的时间参数，其格式如表 6.11 所示。

表 6.11 LCDCON2 控制格式

功 能	位	描 述	初 始 值
VBPD	[31:24]	VSYNC 信号之后，还要经过 (VBPD+1) 个 HSYNC 信号周期才会出现有效行数据	0
LINEVAL	[23:14]	LCD 的行数：(LINEVAL+1) 行	0
VFPD	[13:6]	一帧中的有效数据完成后，到下一个 VSYNC 信号有效前的无效行数目：(VFPD+1)	0
VSPW	[5:0]	表示 VSYNC 信号的宽度为 (VSPW+1) 个 HSYNC 信号周期，这 (VSPW+1) 行的数据无效	0

## 3. LCD控制寄存器LCDCON3

用于设置水平方向各信号的时间参数，其格式如表 6.12 所示。

表 6.12 LCDCON3 控制格式

功 能	位	描 述	初 始 值
HBPD	[25:19]	HSYNC 信号脉冲之后，还要经过 (HBPD+1) 个 VCLK 信号周期，才出现有效数据	0
HOZVAL	[18:8]	LCD 的水平宽度：(HOZVAL+1) 个像素	0
HFPD	[7:0]	一行中的有效数据完后，到下一个 HSYNC 信号有效前的无效像素个数：HFPK+1	0

## 4. LCD控制寄存器LCDCON4

对于 TFT 屏，这个寄存器用来设置 HSYNC 信号的脉冲宽度，其格式如表 6.13 所示。

表 6.13 LCDCON4 控制格式

功 能	位	描 述	初 始 值
MVAL	[15:8]	STN 屏用	0
HSPW	[7:0]	表示脉冲宽度为 (HSPW+1) 个 VCLK 信号周期	0
WLH	[7:0]	STN 屏用	0

## 5. LCD控制寄存器LCDCON5

用于设置各个控制信号的极性，并可从中读取状态信息，其格式如表 6.14 所示。

表 6.14 LCDCON5 控制格式

功 能	位	描 述	初 始 值
VSTATUS	[16:15]	只读，垂直状态 00：处于 VSYNC 信号脉冲期间 01：处于 VSYNC 信号结束到行有效之间 10：处于行有效期间 11：处于行有效结束到下一个 VSYNC 信号之间	0

续表

功 能	位	描 述	初始值
HSTATUS	[14:13]	只读, 描述水平状态 00: 处于 HSYNC 信号脉冲期间 01: 处于 HSYNC 信号结束到像素有效之间 10: 处于像素有效期间 11: 处于像素有效结束到下一个 HSYNC 信号之间	0
BPP24BL	[12]	设置 TFT 屏的显示模式为 24BPP 时, 一个 4 字节中哪 3 个字节有效 0: 低 3 字节有效; 1: 高 3 字节有效	0
FRM565	[11]	设置 TFT 屏的显示模式为 16BPP 时, 数据的格式 0: 5:5:5:1; 1: 5:6:5	0
INVVCLK	[10]	设置 VCLK 信号有效沿的极性 0: 下降沿读取数据; 1: 在上升沿读取数据	0
INVVLIN	[9]	设置 VLINE/HSYNC 脉冲的极性 0: 正常极性; 1: 反转的极性	0
INVVFRA	[8]	设置 VFRA/VSYNC 脉冲的极性 0: 正常极性; 1: 反转的极性	0
INVD	[7]	设置 VD 数据线表示数据的极性 0: 正常极性; 1: 反转的极性	0
INVVDEN	[6]	设置 VDEN 信号的极性 0: 正常极性; 1: 反转的极性	0
INVPWREN	[5]	设置 PWREN 信号的极性 0: 正常极性; 1: 反转的极性	0
INVLEND	[4]	设置 LEND 信号的极性 0: 正常极性; 1: 反转的极性	0
PWREN	[3]	LCD_PWREN 信号输出使能 0: 禁止; 1: 使能	0
ENLEND	[2]	LEND 信号输出使能 0: 禁止; 1: 使能	0
BSWP	[1]	字节交换使能 0: 禁止; 1: 使能	0
HWSWP	[0]	半字交换使能 0: 禁止; 1: 使能	0

## 6. 帧内存地址寄存器

帧内存可以很大, 而真正要显示的区域称为视口, 它处于帧内存之内。s3c2440 有 3 个帧内存地址寄存器, 这 3 个寄存器用于确定帧内存的起始地址, 定位视口在帧内存的位置, 其格式如表 6.15 所示。

表 6.15 帧内存地址寄存器格式

LCDSADDR1			
功 能	位	描 述	初 始 值
LCDBANK	[29:21]	用于保存帧内存起始地址的 A [30: 22]，帧内存起始地址必须为 4M 对齐	0
LCDBASEU	[20:0]	对于双扫描，用于表示上半帧的内存起始地址 对于单扫描，用于表示帧的内存起始地址	0
LCDSADDR2			
功 能	位	描 述	初 始 值
LCDBASEL	[20: 0]	对于双扫描，用于保存下半帧的起始地址 对于单扫描，用于保存帧的结束地址 其值按如下公式计算： $LCDBASEL = CLDBASEU + (PAGEWIDTH + OFFSIZE) * (LINEVAL + 1)$	0
LCDSADDR3			
功 能	位	描 述	初 始 值
OFFSIZE	[21:11]	虚拟屏长度 表示上一行最后一个数据与下一行第一个数据间地址差值的一半，即以半字为单位的地址差	0
PAGEWIDTH	[10:0]	虚拟屏的宽度，这个值决定视口的宽度，以半字为单位	0

## 6.2 LCD 参数设置

LCD 驱动编写的主要任务就是根据所使用的 LCD 屏正确地设置对应的 LCD 寄存器参数。前面已经讲述了 S3C2440 LCD 各控制寄存器，本节将讲述如何设置这些寄存器参数。

### 1. 设置VFRAME、VLINE

VFRAME 和 VLINE 信号可以根据液晶屏的尺寸和显示模式来设置，它们对应 LCDCON2 寄存器的 HOZVAL 和 LINEVAL 值，设置方法如下：

HOZVAL= (水平尺寸/VD 数据位) -1  
 彩色液晶屏：水平尺寸=3 × 水平像素点数  
 VD 数据位=BBP 数（不分单双扫描）  
 LINVAL=垂直尺寸-1（单扫描）  
 LINVAL=垂直尺寸/2-1（双扫描）

### 2. 设置VCLK

LCD 控制器输出的 VCLK 是直接由系统总线（AHB）的工作频率 HCLK 直接分频得到的。

```
VCLK=HCLK/((CLKVAL+1)×2)
```

### 3. 帧速率

帧速率就是 VSYNC 信号的频率。帧速率与 VSYNC、VBPD、VFPD、LINEVAL、HSYNC、HBPD、HFDP、HOZVAL 和 CLKVAL 的域有关，它们是 LCDCON1/2/3/4。大多数 LCD 驱动器需要它们合适的帧速率。帧速率按如下公式计算：

```
Frame Rate=1/[(VSPW+1)+(VBPD+1)+(LINEVAL+1)+(VFPD+1)]*  
[(HSPW+1)+(HBPD+1)+(HFDP+1)+HOZVAL+1]*{2*(CLKVAL+1)/HCLK}]
```

其中，VSPW、VBPD、VFPD、HSPW、HBPD、HFDP 需要参考具体的 LCD 屏的手册（datasheet）来设置。

## 6.3 内核 LCD 驱动机制

本节开始，将介绍如何编写 LCD 驱动程序。实际上，Linux 已经为显示设备专门提供了一类驱动程序，叫做帧缓冲（FrameBuffer）设备驱动程序。实际工作中，工程师只需要在显示缓存中填写将要显示的数据，屏幕上就会显示出相应的图像，驱动的主要工作就是准确地得到这个显示缓存的地址，然后对它进行操作。

### 6.3.1 FrameBuffer 概述

帧缓冲区是出现在 Linux 2.2.xx 及以后版本内核中的一种驱动程序接口，这种接口把显示设备抽象成为帧缓冲区设备区。帧缓冲区为图像设备提供了一种抽象化的处理，它代表了一些视频设备，允许应用软件通过定义明确的界面来访问图像硬件设备。这样软件无需了解任何涉及硬件底层驱动的内容（如硬件寄存器）。

帧缓冲区允许上层应用程序在图形模式下直接对显示缓冲区进行读写和 I/O 控制等操作。通过专门的设备结点可对相应的设备进行访问，如/dev/fb0。应用程序可以将它看成显示内存的一个引用，将其映射到进程地址空间之后，就可以进行读写操作，而读写操作可以反映到具体的 LCD 设备上。

此外，FrameBuffer 驱动程序还考虑了支持控制台的字符显示。在 Linux 2.4 内核中，与 FrameBuffer 控制台有关的代码被放到了 fbcon 和其他相关目录中；在 Linux 2.6 内核中，这些代码被放到了 drivers/video/console 中，它们涵盖了各种格式显示缓冲的字符输出、字体定义文件，这样可以简化 FrameBuffer 控制台驱动程序的移植。

### 6.3.2 FrameBuffer 设备驱动的结构

FrameBuffer 设备驱动基于两个文件，linux/include/linux/fb.h 和 linux/drivers/video/fbmem.c。其中，fb.h 定义了 Framebuffer 驱动所要用到的几乎所有的结构体，这些结构主要包括 struct fb\_info、struct fb\_var\_screeninfo 和 struct fb\_fix\_screeninfo。下面分别讲述这

几个结构体。

### 1. struct fb\_info

struct fb\_info 记录了帧缓冲的全部信息，包括设置参数、状态、操作函数指针等。每个帧缓冲设备均由一个 struct fb\_info 体来描述它，struct fb\_info 所包含的参数、状态、操作函数指针都是面向特定的设备，驱动开发工程师的任务就是填写这些数值。这个结构体也是所有 FrameBuffer 相关结构中唯一一个在内核空间可见的，具体结构代码如下：

```
struct fb_info {
    int node;                                /*次设备号, 如 fb0 中的“0”*/
    int flags;
    struct fb_var_screeninfo var;             /*当前可变参数*/
    struct fb_fix_screeninfo fix;              /*当前固定参数*/
    struct fb_monspecs monspecs;             /*当前的显示器模式*/
    struct work_struct queue;                /*事件队列*/
    struct fb_pixmap pixmap;
    struct fb_pixmap sprite;
    struct fb_cmap cmap;                     /*当前 cmap*/
    struct list_head modelist;               /*模式列表*/
    struct fb_videomode *mode;                /*当前模式*/
    struct fb_ops *fbops;                    /*一些操作指针集, 下面会讲到*/
    struct device *device;                  /*指向 struct platform_device 中的
                                                dev 成员*/
    struct class_device *class_device;        /*sysfs 文件系统用到*/
#ifndef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops;            /*Tile Blitting*/
#endif
    char __iomem *screen_base;               /*硬件 I/O 的虚拟地址*/
    unsigned long screen_size;
    void *pseudo_palette;                  /*调色板*/
#define FBINFO_STATE_RUNNING 0
#define FBINFO_STATE_SUSPENDED 1
    u32 state;                            /*硬件状态*/
    void *fbcon_par;
    /*From here on everything is device dependent*/
    void *par;                            /*驱动定义的私有数据*/
};

};
```

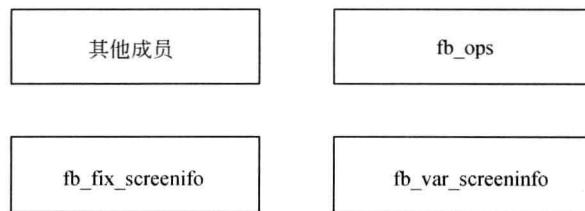
由上可见，struct fb\_var\_screeninfo 和 struct fb\_fix\_screeninfo 两个结构体被包含于 struct fb\_info，这两个结构记录了设备状态信息。它们和 struct fb\_info 的关系如图 6.6 所示。

由图可以看出 struct fb\_var\_screeninfo 与 fb\_fix\_screeninfo 的区别，struct fb\_var\_screeninfo 可以通过 fb\_ops 接口调用进行参数的设置。fb\_ops 定义了一些操作函数，用户应用程序可以使用 ioctl() 系统调用来操作设备，这个结构就是用于支持 ioctl() 的这些操作。

### 2. fb\_var\_screeninfo

struct fb\_var\_screeninfo 记录了帧缓冲设备和指定显示模式的可修改信息。它包括显示屏幕的分辨率、每个像素的比特数和一些时序变量。其中变量 xres 定义了屏幕中一行所占像素的数目，yres 定义了屏幕中一列所占像素的数目，bits\_per\_pixel 定义了每个像素用多

少个位来表示。这些都是可以通过应用程序来设置的，它通过 `fb_opt` 实现。

`fb_info`图 6.6 `fb_info` 结构图

```

struct fb_var_screeninfo {
    __u32 xres;                                /*实际分辨率*/
    __u32 yres;
    __u32 xres_virtual;                         /*虚拟分辨率*/
    __u32 yres_virtual;
    __u32 xoffset;                             /*虚拟分辨率到实际分辨率的偏移*/
    __u32 yoffset;

    __u32 bits_per_pixel;                      /*BPP*/
    __u32 grayscale;                           /*灰度级别*/

    struct fb_bitfield red;                    /*真彩色中的三基色*/
    struct fb_bitfield green;
    struct fb_bitfield blue;
    struct fb_bitfield transp;

    __u32 nonstd;                            /*!= 0 不是超标准格式*/
    __u32 activate;

    __u32 height;                            /*图像高度*/
    __u32 width;                             /*图像宽度*/

    __u32 accel_flags;                        /*加速标志*/

    /*下面是一些时序*/
    __u32 pixclock;                          /*像素时钟*/
    __u32 left_margin;                       /*帧的同步时钟 */
    __u32 right_margin;
    __u32 upper_margin;                      /*行的同步时钟 */
    __u32 lower_margin;
    __u32 hsync_len;                         /*一行的像素数 */
    __u32 vsync_len;                         /*一帧的行数*/
    __u32 sync;
    __u32 vmode;
    __u32 rotate;
    __u32 reserved[5];
};

```

### 3. fb\_fix\_screeninfo

`fb_fix_screeninfo` 定义了硬件的不可变属性，显示缓冲区的映射地址也被定义在这里，它表示缓冲区不应该被应用程序所改变。下面是这个结构体的代码：

```
struct fb_fix_screeninfo {
    char id[16];                      /*驱动中定义的设备名字*/
    unsigned long smem_start;          /*frame buffer 的开始地址, 这是物理地址*/
    __u32 smem_len;                   /*frame buffer 内存的长度*/
    __u32 type;                      /*类型*/
    __u32 type_aux;                  /*Interleave for interleaved Planes*/
    __u32 visual;                    /*see FB_VISUAL_**/
    __u16 xpanstep;                 /*如果硬件没有 panning, 那么填 0*/
    __u16 ypanstep;                 /*如果硬件没有 panning, 那么填 0*/
    __u16 ywrapstep;                 /*如果硬件没有 panning, 那么填 0*/
    __u32 line_length;               /*一行的字节表示*/
    unsigned long mmio_start;         /*frame buffer 的开始地址, 这是虚拟地址*/
    __u32 mmio_len;                  /*I/O 的大小*/
    __u32 accel;                     /*可用的加速类型 */
    __u16 reserved[3];               /*保留位*/
};
```

### 4. fbmem.c

`fbmem.c` 是 Framebuffer 设备驱动技术的关键。它为上层应用程序提供系统调用，也为下一层的特定硬件驱动提供编程接口；那些底层硬件驱动需要用到这里的接口来向系统内核注册它们自己。

`fbmem.c` 为所有支持 FrameBuffer 的设备驱动提供了通用的接口，避免了重复的工作。用户开发硬件驱动时，只要针对 `fbmem.c` 中提供的底层驱动接口函数分别加以实现即可。这些接口函数就是前面讲到的 `struct fb_ops` 函数指针组，具体代码如下：

```
struct fb_ops {
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);
    ssize_t (*fb_read)(struct fb_info *info, char __user *buf,
                      size_t count, loff_t *ppos);
    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,
                       size_t count, loff_t *ppos);
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);
    int (*fb_set_par)(struct fb_info *info);
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
                        unsigned blue, unsigned transp, struct fb_info *info);
    int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);
    int (*fb_blank)(int blank, struct fb_info *info);
    int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);
    void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect
```

```

*rect);
void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea
*region);
void (*fb_imageblit) (struct fb_info *info, const struct fb_image
*image);
int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);
void (*fb_rotate)(struct fb_info *info, int angle);
int (*fb_sync)(struct fb_info *info);
int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
               unsigned long arg);
int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
                       unsigned long arg);
int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);
void (*fb_save_state)(struct fb_info *info);
void (*fb_restore_state)(struct fb_info *info);
void (*fb_get_caps)(struct fb_info *info, struct fb_blt_caps *caps,
                    struct fb_var_screeninfo *var);
};

}

```

这个结构体内有很多函数，在这里不一一讲述了。驱动不用每个函数都去实现它，只要实现其中部分函数就可以了，比较多的是实现设备属性的设置和获得函数如 `fb_get_var`、`fb_set_var`。

`fbmem.c` 还实现了如下函数：

```

register_framebuffer(struct fb_info *fb_info);
unregister_framebuffer(struct fb_info *fb_info);

```

这两个是提供给下层 FrameBuffer 设备驱动的接口，设备驱动程序通过这两个函数向系统注册或注销自己。可以说，底层设备驱动所要做的所有事情就是填充 `fb_info` 结构，并向系统注册或注销它。

## 6.4 Linux 2.6.32 的 LCD 驱动源码分析

内核源码中已经有 LCD 的相关驱动了，驱动开发人员只要了解了内核的 LCD 驱动体系结构，然后参考内核中已有 LCD 驱动源码，再针对具体的显示屏型号和硬件资源做相关修改就可以了。因此，从本节开始将要分析 Linux 2.6.32 的 LCD 驱动源码。

### 6.4.1 LCD 驱动开发的主要工作

LCD 驱动开发的工作包括两个方面，分别是初始化函数的编写和填充 `fb_info` 结构体中的主要成员函数，下面分别讲述。

#### 1. 编写初始化函数

初始化函数首先初始化各个 LCD 控制器寄存器，通过写寄存器来设置显示的模式和颜色数，然后在内存中分配 LCD 显示缓冲区。在 Linux 中可以用 `kmalloc()` 函数分配一段连续的空间。缓冲区大小为：点阵行数×点阵列数×用于表示一个像素的比特数/8。缓冲

区通常分配在大容量的片外 SDRAM 中，起始地址保存在 LCD 控制寄存器中。本节采用的 LCD 显示方式为  $320 \times 240$ 、16 位彩色，则需要分配的显示缓冲区为  $320 \times 240 \times 2 = 15\text{kb}$ 。最后是初始化一个 `fb_info` 结构体，填充其中的成员变量，并调用 `register_framebuffer(&fb_info)`，将 `fb_info` 注册入内核。

## 2. 编写成员函数

对于嵌入式系统的简单实现，编写结构 `fb_info` 中函数指针 `fb_ops` 对应的成员函数只需要以下 3 个函数就可以了。

```
struct fb_ops{
...
int (*fb_get_fix)(struct fb_fix_screeninfo *fix, int con, struct fb_info
*info);
int (*fb_get_var)(struct fb_var_screeninfo *var, int con, struct fb_info
*info);
int (*fb_set_var)(struct fb_var_screeninfo *var, int con, struct fb_info
*info);
...
}
```

`Struct fb_ops` 在 `include/linux/fb.h` 中定义，前面已经讲述过了。这些函数都是用来设置/获取 `fb_info` 结构中的成员变量的。当应用程序对设备文件进行 `ioctl` 系统调用时会调用它们。对于 `fb_get_fix()`，应用程序把 `fb_fix_screeninfo` 这个结构传进来，在函数中对各个成员变量赋值，主要是 `smem_start`（缓冲区起始地址）和 `smem_len`（缓冲区长度），最终返回给应用程序。而 `fb_set_var()` 函数的传入参数是 `fb_var_screeninfo`，函数中需要对 `xres`、`yres` 和 `bits_per_pixel` 赋值。

对于 `/dev/fb`，对显示设备的操作主要有以下几种。

- 读/写（`read/write`）`/dev/fb`：相当于读/写屏幕缓冲区。
- 映射（`map`）操作：由于 Linux 工作在保护模式，每个应用程序都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲区地址的。为此，Linux 在文件操作 `file_operations` 结构中提供了 `mmap()` 函数，可将文件的内容映射到用户空间。帧缓冲设备则可通过映射操作，可将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址之内，之后用户就可以通过读写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。
- I/O 控制：帧缓冲设备对设备文件的 `ioctl` 操作可读取/设置显示设备及屏幕的参数，如分辨率、显示颜色数和屏幕大小等。`ioctl` 的操作是由底层的硬件驱动程序来完成的。在应用程序中，操作 `/dev/fb` 的一般步骤是：打开 `/dev/fb` 设备文件；用 `ioctl` 操作取得当前显示屏幕的参数，如屏幕分辨率以及每个像素的比特数，根据屏幕参数可计算屏幕缓冲区的大小；将屏幕缓冲区映射到用户空间；映射后即可直接读写屏幕缓冲区，进行绘图和图片显示了。

### 6.4.2 s3c2410fb\_init()函数分析

首先要分析的是 `s3c2410fb_init()` 函数，该函数内容相对简单，在 `s3c2410fb_init()` 函数

体内只是包装了对平台驱动注册函数 `platform_driver_register()` 的调用，它的参数是 `&s3c2410fb_driver`。这里是向内核注册一个 platform 设备驱动的意思，该 platform 设备是 LCD 设备。

platform 有两个比较重要的数据结构，分别是 `platform_device` 和 `platform_driver`，这里用到的就是 `platform_driver`。`platform_driver` 在 `include/linux/platform_device.h` 中有定义，它的成员无非是一些回调函数还有一个同 `platform_device` 一致的设备名字，在前面章节已经讲过了。在 LCD 驱动程序 (`drivers/video/s3c2410fb.c`) 中有 `platform_driver` 结构体的定义，内容如下：

```
static struct platform_driver s3c2410fb_driver = {
    .probe      = s3c2410fb_probe,           // 初始化函数
    .remove     = s3c2410fb_remove,
    .suspend    = s3c2410fb_suspend,
    .resume     = s3c2410fb_resume,
    .driver     = {
        .name   = "s3c2410-lcd", /*设备名字，这个名字一定要和 struct platform_
                                         device 中的 name 域一样才能把平台驱动和前面定义的
                                         平台数据联系起来*/
        .owner   = THIS_MODULE,
    },
};
```

由以上可以看到，该 platform 设备的驱动有 `s3c2410fb_probe`、`s3c2410fb_remove` 等回调函数。在通过 `platform_driver_register` 函数注册该设备的过程中，它会回调 `probe` 探测函数，也就是说 `s3c2410fb_probe` 是在 `platform_driver_register` 中被回调的。

### 6.4.3 s3c2410fb\_probe()函数分析

这个函数是驱动的关键函数，下面就一步步来分析它。因为函数太长，这里先列出源码，然后在后面再对源码进行分析。

```
799 static int __init s3c24xxfb_probe(struct platform_device *pdev,
800                                     enum s3c_drv_type drv_type)
801 {
802     struct s3c2410fb_info *info;
803     struct s3c2410fb_display *display;
804     struct fb_info *fbinfo;
805     struct s3c2410fb_mach_info *mach_info;
806     struct resource *res;
807     int ret;
808     int irq;
809     int i;
810     int size;
811     u32 lcdcon1;
812 }
```

第 802~811 行代码定义了本函数用到的变量，其中 `s3c2410fb_info` 是驱动自己定义的设备数据结构，可以说该结构记录了 `s3c2410fb` 驱动的所有信息；`s3c2410fb_display` 这个结构体定义了 LCD 屏的规格和时序；`fb_info` 是 `framebuffe` 里定义的设备数据结构，表示

一个显示设备，`s3c2410fb_probe` 的最终目的是填充该结构，并向内核注册；`s3c2410fb_mach_info` 结构里包含了一些 LCD 控制器的 GPIO，这个主要用来设置引脚的功能。Resource 指向设备用到的 I/O 资源，这是平台驱动的结构；irq 用来保存中断号。

```

813     mach_info = pdev->dev.platform_data;
814     if (mach_info == NULL) {
815         dev_err(&pdev->dev,
816                 "no platform data for lcd, cannot attach\n");
817         return -EINVAL;
818     }
819

```

第 813~818 行代码用于获得平台数据，这里需要深入说明一下。`mach_info` 是一个 `s3c2410fb_mach_info` 类型的指针，它描述了 LCD 的一些属性。值得读者注意的是，`s3c2410fb_mach_info` 和 `s3c2410fb_info` 结构是有区别的。同样是描述 LCD 的属性，`s3c2410fb_mach_info` 只是用于描述 LCD 初始化时所用的值，而 `s3c2410fb_info` 是描述整个 LCD 驱动的结构体。

`s3c2410fb_mach_info` 在 `include/asm-arm/arch-s3c2410/fb.h` 中定义，它不是内核所认知的数据结构，它只和平台相关，也就是说，这只是驱动程序设计者设计的结构。从后面的 if 语句可以知道，如果 `mach_info` 等于 `NULL`，整个驱动程序就会退出，这里读者应该会问，`pdev->dev.platform_data` 是在什么时候被初始化的呢？这个问题在后面将会讲到。

```

820     if (mach_info->default_display >= mach_info->num_displays) {
821         dev_err(&pdev->dev, "default is %d but only %d displays\n",
822                 mach_info->default_display, mach_info->num_displays);
823         return -EINVAL;
824     }
825
826     display = mach_info->displays + mach_info->default_display;

```

第 820~826 行代码用来查找当前显示屏的规格。这个规格是在 `arch/arm/mach-smdk2440.c` 文件中定义的平台数据结构体的一些参数。

```

827
828     irq = platform_get_irq(pdev, 0);
829     if (irq < 0) {
830         dev_err(&pdev->dev, "no irq for device\n");
831         return -ENOENT;
832     }

```

第 829~832 行代码获得设备的中断号。这个中断号也是在 `arch/arm/mach-smdk2440.c` 文件中定义的硬件所使用到的中断资源。

```

833
834     fbinfo = framebuffer_alloc(sizeof(struct s3c2410fb_info),
835                               &pdev->dev);
836     if (!fbinfo)
837         return -ENOMEM;

```

第 834~836 行代码的功能是向内核申请一段大小为 `sizeof(struct fb_info) + size` 的空间，其中 `size` 的大小代表设备的私有数据空间，并用 `fb_info` 的 `par` 域指向该私有空间。

```

837     platform_set_drvdata(pdev, fbinfo);

```

```

839         info = fbinfo->par;
840         info->dev = &pdev->dev;
841         info->drv_type = drv_type;

```

第 838~842 行代码设置驱动数据并填充驱动数据。

```

843             res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
844             if (res == NULL) {
845                 dev_err(&pdev->dev, "failed to get memory registers\n");
846                 ret = -ENXIO;
847                 goto dealloc_fb;
848             }
849
850             size = (res->end - res->start) + 1;
851             info->mem = request_mem_region(res->start, size, pdev->name);
852             if (info->mem == NULL) {
853                 dev_err(&pdev->dev, "failed to get memory region\n");
854                 ret = -ENOENT;
855                 goto dealloc_fb;
856             }
857
858             info->io = ioremap(res->start, size);
859             if (info->io == NULL) {
860                 dev_err(&pdev->dev, "ioremap() of registers failed\n");
861                 ret = -ENXIO;
862                 goto release_mem;
863             }
864
865             info->irq_base = info->io + ((drv_type == DRV_S3C2412) ?
866 S3C2412_LCDINTBASE : S3C2410_LCDINTBASE);

```

第 844~866 行代码主要是映射内核资源，如 I/O 口和中断号，平台数据不定期传递来的硬件资源地址都是物理地址，而在内核中对硬件进行访问都是要通过虚拟地址的，所以要用相关函数把物理地址映射为虚拟地址。如果在这里映射失败将退出程序。

```

867         dprintk("devinit\n");
868
869         strcpy(fbinfo->fix.id, driver_name);
870
871         /*Stop the video*/
872         lcdcon1 = readl(info->io + S3C2410_LCDCON1);
873         writel(lcdcon1 & ~S3C2410_LCDCON1_ENVID, info->io + S3C2410_
874 LCDCON1);
875
876         fbinfo->fix.type          = FB_TYPE_PACKED_PIXELS;
877         fbinfo->fix.type_aux     = 0;
878         fbinfo->fix.xpanstep    = 0;
879         fbinfo->fix.ypanstep    = 0;
880         fbinfo->fix.ywrapstep   = 0;
881         fbinfo->fix.accel       = FB_ACCEL_NONE;
882
883         fbinfo->var.nonstd      = 0;
884         fbinfo->var.activate     = FB_ACTIVATE_NOW;
885         fbinfo->var.accel_flags = 0;
886         fbinfo->var.vmode       = FB_VMODE_NONINTERLACED;
887
888         fbinfo->fbops           = &s3c2410fb_ops;

```

```

889     fbiinfo->flags          = FBINFO_FLAG_DEFAULT;
890     fbiinfo->pseudo_palette = &info->pseudo_pal;
891
892     for (i = 0; i < 256; i++)
893         info->palette_buffer[i] = PALETTE_BUFF_CLEAR;

```

第 872~893 行代码首先关闭显示屏，然后对 fbiinfo 进行赋值。

```

894
895     ret = request_irq(irq, s3c2410fb_irq, IRQF_DISABLED, pdev->
896                         name, info);
897     if (ret) {
898         dev_err(&pdev->dev, "cannot get irq %d - err %d\n", irq, ret);
899         ret = -EBUSY;
900         goto release_regs;
901     }
902
903     info->clk = clk_get(NULL, "lcd");
904     if (!info->clk || IS_ERR(info->clk)) {
905         printk(KERN_ERR "failed to get lcd clock source\n");
906         ret = -ENOENT;
907         goto release_irq;
908     }
909
910     clk_enable(info->clk);
911     dprintk("got and enabled clock\n");
912
913     msleep(1);

```

第 895~912 行代码申请中断并打开 LCD 时钟，使得各 LCD 相关引脚输出信号。

```

913
914     /*find maximum required memory size for display*/
915     for (i = 0; i < mach_info->num_displays; i++) {
916         unsigned long smem_len = mach_info->displays[i].xres;
917
918         smem_len *= mach_info->displays[i].yres;
919         smem_len *= mach_info->displays[i].bpp;
920         smem_len >>= 3;
921         if (fbiinfo->fix.smem_len < smem_len)
922             fbiinfo->fix.smem_len = smem_len;
923     }
924
925     /*Initialize video memory*/
926     ret = s3c2410fb_map_video_memory(fbiinfo);
927     if (ret) {
928         printk(KERN_ERR "Failed to allocate video RAM: %d\n", ret);
929         ret = -ENOMEM;
930         goto release_clock;
931     }

```

第 915~931 行代码计算显示内存容量并映射 DMA 资源，上层应用程序把要显示的图像存入这个内存区域，然后 LCD 控制器通过 DMA 从这个映射内容中取出要显示的图像。

```

932
933     dprintk("got video memory\n");
934
935     fbiinfo->var.xres = display->xres;
936     fbiinfo->var.yres = display->yres;
937     fbiinfo->var.bits_per_pixel = display->bpp;
938

```

```

939         s3c2410fb_init_registers(fbinfo);
940
941         s3c2410fb_check_var(&fbinfo->var, fbinfo);
942
943         ret = register_framebuffer(fbinfo);
944         if (ret < 0) {
945             printk(KERN_ERR "Failed to register framebuffer device: %d\n",
946                   ret);
947             goto free_video_memory;
948         }
949
950         /*create device files*/
951         device_create_file(&pdev->dev, &dev_attr_debug);
952
953         printk(KERN_INFO "fb%d: %s frame buffer device\n",

```

第 939~953 行代码初始化各个 LCD 控制寄存器、注册我们的 fbinfo 并为该设备创建一个在 sysfs 中的属性。

```

954             fbinfo->node, fbinfo->fix.id);
955
956         return 0;
957
958     free_video_memory:
959         s3c2410fb_unmap_video_memory(fbinfo);
960     release_clock:
961         clk_disable(info->clk);
962         clk_put(info->clk);
963     release_irq:
964         free_irq(irq, info);
965     release_regs:
966         iounmap(info->io);
967     release_mem:
968         release_resource(info->mem);
969         kfree(info->mem);
970     dealloc_fb:
971         platform_set_drvdata(pdev, NULL);
972         framebuffer_release(fbinfo);
973     return ret;
974 }

```

前面留下了一个问题，pdev->dev.platform\_data 是在什么时候被初始化的？其实在内核启动 init 进程之前就会执行 smdk2410\_map\_io() 函数，而在 smdk2410\_map\_io() 函数中加入了 s3c24xx\_fb\_set\_platdata (&smdk2410\_lcd\_platdata) 这条语句，s3c24xx\_fb\_set\_platdata() 的实现为：

```

void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)
{
    s3c_device_lcd.dev.platform_data = pd;
}

```

根据这些代码，可以清楚地看到 s3c\_device\_lcd.dev.platform\_data 指向了 smdk2410\_lcd\_platdata，而这个 smdk2410\_lcd\_platdata 就是一个 s3c2410fb\_mach\_info 的变量，它里面就存放了 LCD 驱动初始化需要的初始数据。当 s3c2410fb\_probe 被回调时，所传给它的参数实际就是 s3c\_device\_lcd 的首地址。

在 s3c2410fb\_probe 中最后调用了 s3c2410fb\_init\_registers() 和 s3c2410fb\_check\_var() 函数，这里应该将它们交代清楚。很明显，s3c2410fb\_init\_registers() 是用来初始化相关寄存

器的函数。那么后者呢？这里先说 `s3c2410fb_init_registers()`。`s3c2410fb_init_registers()` 的定义和实现如下，先根据它的程序流程，一步一步分析。

```

static int s3c2410fb_init_registers(struct s3c2410fb_info *fbi)
{
    unsigned long flags;
    /*Initialise LCD with values from haret*/
    local_irq_save(flags);      /*关闭中断，在关闭中断前，中断的当前状态被保存在
                                flags 中，对于关闭中断的函数，Linux 内核有很多种，
                                可以查阅相关的资料。*/
    /*下面的 modify_gpio() 函数是修改处理器 GPIO 的工作模式，它的实现很简单，将第 2 个参数
    的值与第 3 个参数的反码按位与操作后，写到第 1 个参数里。这里的第 1 个参数实际就是硬件的
    GPIO 控制器。*/
    modify_gpio(S3C2410_GPCUP, mach_info->gpcup, mach_info->gpcup_mask);
    modify_gpio(S3C2410_GPCCON, mach_info->gpccon, mach_info->gpccon_
mask);
    modify_gpio(S3C2410_GPDUP, mach_info->gpdup, mach_info->gpdup_mask);
    modify_gpio(S3C2410_GPDCON, mach_info->gpdccon, mach_info->gpdccon_
mask);
    local_irq_restore(flags);    /*使能中断，并恢复以前的状态*/
    /*下面的几个 writel() 函数开始初始化 LCD 控制寄存器，它的值就是在 smdk2410_lcd_
platdata(arch/arm/mach-s3c2410/mach-smdk2410.c) 中 regs 域的值。*/
    writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
    writel(fbi->regs.lcdcon2, S3C2410_LCDCON2);
    writel(fbi->regs.lcdcon3, S3C2410_LCDCON3);
    writel(fbi->regs.lcdcon4, S3C2410_LCDCON4);
    writel(fbi->regs.lcdcon5, S3C2410_LCDCON5);
    s3c2410fb_set_lcdaddr(fbi); /*该函数的主要作用是让处理器的 LCD 控制器的 3
                                个地址寄存器指向正确的位置，这个位置就是 LCD
                                的缓冲区，详细的情况可以参见 s3c2410 的用户手
                                册。*/
    /*下面的程序是打开 video，在 s3c2410fb_probe 中被关闭了，这里打开*/
    fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID;
    writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
    return 0;
}

```

`s3c2410fb_init_registers` 就简单分析到这里。下面看看 `s3c2410fb_check_var()` 函数要做什么事，要说到这个函数，还得提到 `fb_var_screeninfo` 这个结构类型，与它对应的是 `fb_fix_screeninfo` 结构类型。这两个类型分别代表了显示屏的属性信息，这些信息可以分为可变属性信息（如颜色深度、分辨率等）和不可变的信息（如帧缓冲的真实地址）。既然 `fb_var_screeninfo` 表示了可变的属性信息，那么这些可变的信息就应该有一定的范围，否则显示就会出问题，所以 `s3c2410fb_check_var()` 函数的功能就是要在 LCD 的帧缓冲驱动开始运行之前将这些值初始到合法的范围内。知道了 `s3c2410fb_check_var()` 函数要做什么，再去阅读 `s3c2410fb_check_var()` 函数的代码就没什么问题了。

#### 6.4.4 s3c2410fb\_remove()函数分析

现在解释 `s3c2410fb_driver` 中的最后一个关键函数 `s3c2410fb_remove()`。顾名思义该函

数要将这个 platform 设备从系统中移除，可以推测它的作用应该释放掉所有的资源，包括内存空间、中断线等。和前文一样，我们在它的实现代码中一步步分析。

```

static int s3c2410fb_remove(struct platform_device *pdev)
{
    struct fb_info     *fbinfo = platform_get_drvdata(pdev);
                           /*该函数从 platform_device 中获得 fb_info 信息*/
    struct s3c2410fb_info *info = fbinfo->par;      //得到私有数据
    int irq;
    s3c2410fb_stop_lcd(info);                      //该函数停止 LCD 控制器,
                                                       实现可以在 s3c2410fb.c 中
                                                       找到
    msleep(1); //等待 LCD 停止
    s3c2410fb_unmap_video_memory(info);             //该函数释放缓冲区
                                                       //停止时钟
    if (info->clk) {
        clk_disable(info->clk);
        clk_put(info->clk);
        info->clk = NULL;
    }
    irq = platform_get_irq(pdev, 0);                  //得到中断线, 以便释放
    free_irq(irq, info);                            //释放该中断
    release_mem_region((unsigned long)S3C24XX_VA_LCD, S3C24XX_SZ_LCD); //释放内存空间
    unregister_framebuffer(fbinfo);                //向内核注销该帧缓冲
    return 0;
}

```

## 6.5 移植内核中的 LCD 驱动

不同的开发者在设计同一种设备时都有其独特性，比如访问硬件资源所对应的端口地址等。所以驱动移植的首要工作是要明确所驱动的设备硬件连接情况，然后在此基础上对驱动源码进行修改以适应具体的硬件。

### 6.5.1 LCD 硬件电路图

写驱动程序前先了解一下电路连接情况。LCD 的像素同步时钟信号、水平同步信号、垂直同步信号直接连接到 LCD 的 VCLK、VLINE 和 VFRAME 上，用 GPG4 作为 LCD 的电源信号直接连接到 LCD\_PWREN 上。如图 6.7 所示为 TFT LCD 屏与 CPU 的连接图。

### 6.5.2 修改 LCD 源码

本章用的源码是 linux2.6.32 中的 LCD 驱动源码，文件为 drivers/video/s3c2410fb.c 和 drivers/video/s3c2410fb.h。

(1) 修改 arch/arm/mach-s3c2410/include/mach/fb.h 中的 s3c2410fb\_display 结构，在其中加入一个元素用于在初始化时设置 CLKVAL 参数。修改后结构如下，黑体为修改部分。

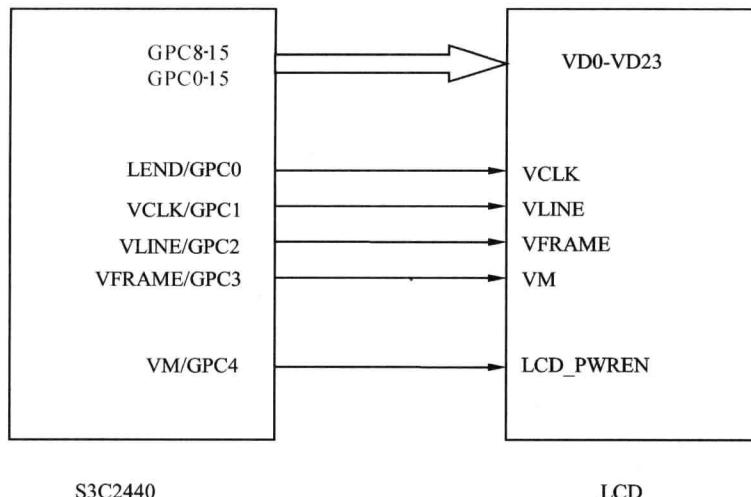


图 6.7 TFT LCD 屏与 CPU 的连接图

```

struct s3c2410fb_display {
    /*LCD type*/
    unsigned type;

    /*Screen size*/
    unsigned short width;
    unsigned short height;

    /*Screen info*/
    unsigned short xres;
    unsigned short yres;
    unsigned short bpp;

    unsigned pixclock;      //pixclock in picoseconds
    unsigned setclkval;   //在这里加入 setclkval，用来设置 CLKVAL 参数
    unsigned short left_margin; // (TFT) 或 HCLks (STN) 的像素值
    unsigned short right_margin; // (TFT) 或 HCLks (STN) 的像素值
    unsigned short hsync_len; // (TFT) 或 HCLks (STN) 的像素值
    unsigned short upper_margin; // (TFT) 或 0 (STN) 的行
    unsigned short lower_margin; // (TFT) 或 0 (STN) 的行
    unsigned short vsync_len; // (TFT) 或 0 (STN) 的行

    // lcd configuration registers
    unsigned long lcdcon5;
};

```

(2) 在源码中加入设置 CLKVAL 参数的代码，这里修改 drivers/video/s3c2410fb.c 中的函数 s3c2410fb\_activate\_var()，修改后结果如下，黑体为修改部分。

```

static void s3c2410fb_activate_var(struct fb_info *info)
{
    struct s3c2410fb_info *fbi = info->par;
    void __iomem *regs = fbi->io;
    int type = fbi->regs.lcdcon1 & S3C2410_LCDCON1_TFT;

```

```

struct fb_var_screeninfo *var = &info->var;
int clkdiv = s3c2410fb_calc_pixclk(fbi, var->pixclock) / 2;

/*获取平台数据并得到参数值*/
struct s3c2410fb_mach_info *mach_info=fbi->dev->platform_data;
struct s3c2410fb_display *default_display=mach_info->displays+mach_info->
default_display;

dprintk("%s: var->xres = %d\n", __FUNCTION__, var->xres);
dprintk("%s: var->yres = %d\n", __FUNCTION__, var->yres);
dprintk("%s: var->bpp = %d\n", __FUNCTION__, var->bits_per_pixel);

if (type == S3C2410_LCDCON1_TFT) {
    s3c2410fb_calculate_tft_lcd_regs(info, &fbi->regs);
    --clkdiv;
    if (clkdiv < 0)
        clkdiv = 0;
} else {
    s3c2410fb_calculate_stn_lcd_regs(info, &fbi->regs);
    if (clkdiv < 2)
        clkdiv = 2;
}

fbi->regs.lcdcon1 |= S3C2410_LCDCON1_CLKVAL(clkdiv);

/*写新的寄存器*/

dprintk("new register set:\n");
dprintk("lcdcon[1] = 0x%08lx\n", fbi->regs.lcdcon1);
dprintk("lcdcon[2] = 0x%08lx\n", fbi->regs.lcdcon2);
dprintk("lcdcon[3] = 0x%08lx\n", fbi->regs.lcdcon3);
dprintk("lcdcon[4] = 0x%08lx\n", fbi->regs.lcdcon4);
dprintk("lcdcon[5] = 0x%08lx\n", fbi->regs.lcdcon5);

	writel(fbi->regs.lcdcon1 & ~S3C2410_LCDCON1_ENVID,
       regs + S3C2410_LCDCON1);
	writel(fbi->regs.lcdcon2, regs + S3C2410_LCDCON2);
	writel(fbi->regs.lcdcon3, regs + S3C2410_LCDCON3);
	writel(fbi->regs.lcdcon4, regs + S3C2410_LCDCON4);
	writel(fbi->regs.lcdcon5, regs + S3C2410_LCDCON5);

/*set lcd address pointers/
s3c2410fb_set_lcdaddr(info);

/*注释源码中设置 CLKVAL 的代码并增加我们的设置代码*/
/*fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID,*/
fbi->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID (default_display->setclkval) ;

	writel(fbi->regs.lcdcon1, regs + S3C2410_LCDCON1);
}

```

(3)修改LCD的参数值。源码中的参数设置在arch/arm/mach-s3c2440/mach-smdk2440.c中,全局变量smdk2410\_lcd\_cfg就是配置LCD参数的地方,配置这些参数要根据具体LCD屏手册来配置,读者请参考自己所使用的LCD屏的datasheet来设置。本书使用的是型号为WXCAT35-TG3#001F的LCD屏,其参数如表6.16所示。

表 6.16 LCD屏的时序表

Signal	Item	Symbol	Min	Typ	Max	Unit
Dclk	Frequency	Dclk	-	6.4	-	MHZ
	Dclk-Period	Tosc	-	156	-	ns
Data	Setup Time	TSU	12	-	-	ns
	Hold Time	THD	12	-	-	ns
Hsync	Period	Th	-	408	-	DCLK
	Pulse Width	Thp	-	30	-	DCLK
	Back-Porch	Thb	-	38	-	DCLK
	Display Period	Thd	-	320	-	DCLK
	Front-Porch	Thf	-	20	-	DCLK
Vsync	Period	Tv	-	270	-	TH
	Pulse Width	Tvp	-	3	-	TH
	Back-Porch	Tvb	-	15	-	TH
	Display Period	Tvd	-	240	-	TH
	Front-Porch	Tvf	-	12	-	TH

修改后结果如下，黑体为修改的内容。

```

static struct s3c2410fb_display smdk2440_lcd_cfg __initdata = {

    .lcdcon5      = S3C2410_LCDCON5_FRM565 |
                      S3C2410_LCDCON5_INVVLINE |
                      S3C2410_LCDCON5_INVVFRAME |
                      S3C2410_LCDCON5_FWREN |
                      S3C2410_LCDCON5_HWSWP,

    .type         = S3C2410_LCDCON1_TFT,
    .width        = 320,           //这是 LCD 屏的分辨率
    .height       = 240,
    .pixclock     = 100000,        // HCLK 60 MHz, divisor 10
    .setclkval    = 0x3,          //这是前面加入的新变量
    .xres         = 320,
    .yres         = 240,
    .bpp          = 16,
    /*下面写参数要对照芯片手册来修改*/
    .left_margin   = 38,          //左边框
    .right_margin  = 20,          //右边框
    .hsync_len     = 30,          //水平时长
    .upper_margin  = 15,          //上边框
    .lower_margin  = 12,          //下边框
    .vsync_len     = 3,           //垂直时长
};

static struct s3c2410fb_mach_info smdk2440_fb_info __initdata = {
    .displays     = &smdk2440_lcd_cfg,
    .num_displays = 1,
    .default_display = 0,
};

#endif

```

```

/*currently setup by downloader*/
.gpccon      = 0xaa940659,
.gpccon_mask = 0xffffffff,
.gpcup       = 0x0000ffff,
.gpcup_mask = 0xffffffff,
.gpdcon     = 0xaa84aaa0,
.gpdcon_mask = 0xffffffff,
.gpdup      = 0x0000faff,
.gpdup_mask = 0xffffffff,
#endif
/*源码中把对引脚的功能设置屏蔽掉了，这样引脚的功能还是默认设置，所以要根据芯片手册
把相关引脚设置为第三功能，即 LCD 的相关功能，设置的结果如下*/
.gpccon      = 0aaaaaaaaa,
.gpccon_mask = 0xffffffff,
.gpcup       = 0xffffffff,
.gpcup_mask = 0xffffffff,
.gpdcon     = 0aaaaaaaaa,
.gpdcon_mask = 0xffffffff,
.gpdup      = 0xffffffff,
.gpdup_mask = 0xffffffff,
//我们用的不是 LPC 屏，所去掉这个设置
//.lpcsel      = ((0xCE6) & ~7) | 1<<4,
};

}

(4) 修改源码中一处有误的地方。在源码中，没有设置 LCD 的供电电源，这样 LCD 屏是点不起来的，所以要补充对 LCD 屏供电引脚的设置。代码如下，黑体为增加的代码。

```

```

/*
 * s3c2410fb_init_registers - Initialise all LCD-related registers
 */
static int s3c2410fb_init_registers(struct fb_info *info)
{
    struct s3c2410fb_info *fbi = info->par;
    struct s3c2410fb_mach_info *mach_info = fbi->dev->platform_data;
    unsigned long flags;
    void __iomem *regs = fbi->io;
    void __iomem *tpal;
    void __iomem *lpcsel;

    if (is_s3c2412(fbi)) {
        tpal = regs + S3C2412_TPAL;
        lpcsel = regs + S3C2412_TCONSEL;
    } else {
        tpal = regs + S3C2410_TPAL;
        lpcsel = regs + S3C2410_LPCSEL;
    }

    /* Initialise LCD with values from haret */

    local_irq_save(flags);

    /* modify the gpio(s) with interrupts set (bjd) */

    modify_gpio(S3C2410_GPCUP,                                mach_info->gpcup,
    mach_info->gpcup_mask);
}

```

```

    modify_gpio(S3C2410_GPCCON,
mach_info->gpccon_mask);
    modify_gpio(S3C2410_GPDUP,
mach_info->gpdup_mask);
    modify_gpio(S3C2410_GPDCON,
mach_info->gpdcon_mask);

//设置 LCD 的供电电源, GPG4 为上拉, 输出
modify_gpio(S3C2410_GPGUP, 0x00000010, 0x00000010);
modify_gpio(S3C2410_GPGCON, 0x00000300, 0x00000300);

local_irq_restore(flags);

dprintk("LPCSEL    = 0x%08lx\n", mach_info->lpcsel);
writel(mach_info->lpcsel, lpcsel);

dprintk("replacing TPAL %08x\n", readl(tpal));

/* ensure temporary palette disabled */
writel(0x00, tpal);

return 0;
}
}

```

### 6.5.3 配置内核

做完以上工作，我们就可以对 LCD 进行配置了。进入内核源码所在的目录，输入 make menuconfig 进入配置菜单后，进行如下配置：

(1)选择进入 Device Drivers。Linux 的所有设备驱动都放在 Device 文件夹里，而 Device 包含的所有设备驱动都会在 Device Drivers 这个选项里列出，要编译设备驱动程序都要进入这个选项然后根据具体情况配置，所以这里选择 Device Drivers，如图 6.8 所示。

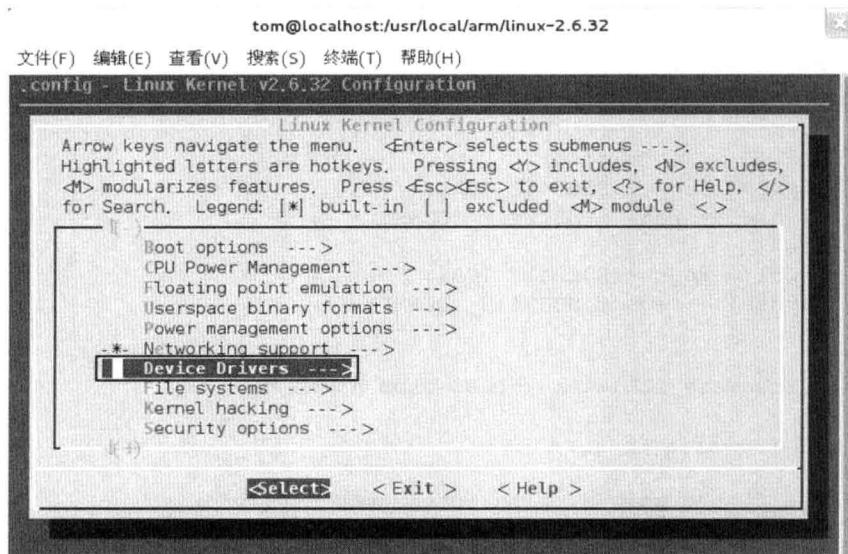


图 6.8 步骤 1

(2) 选择 Graphics support 选项，如图 6.9 所示。

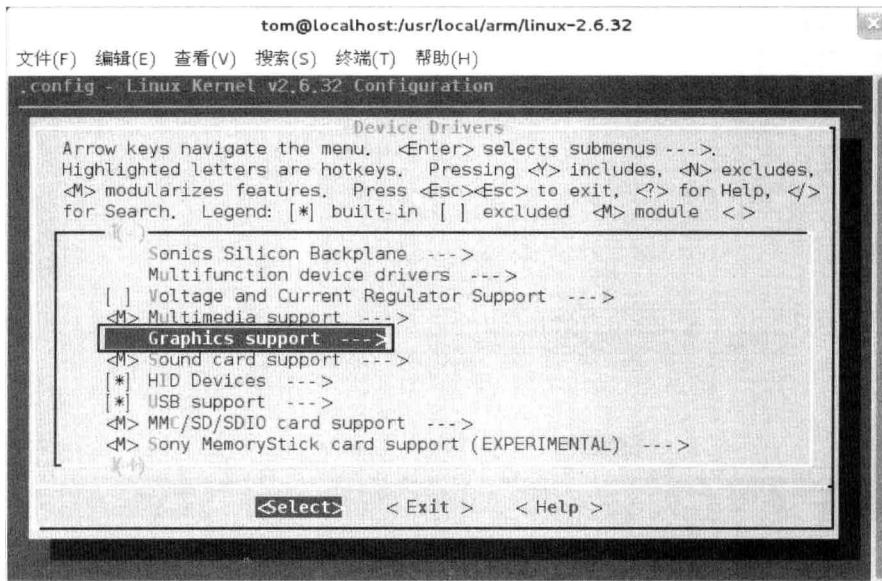


图 6.9 步骤 2

(3) 选择 Support for frame buffer devices 选项，如图 6.10 所示。

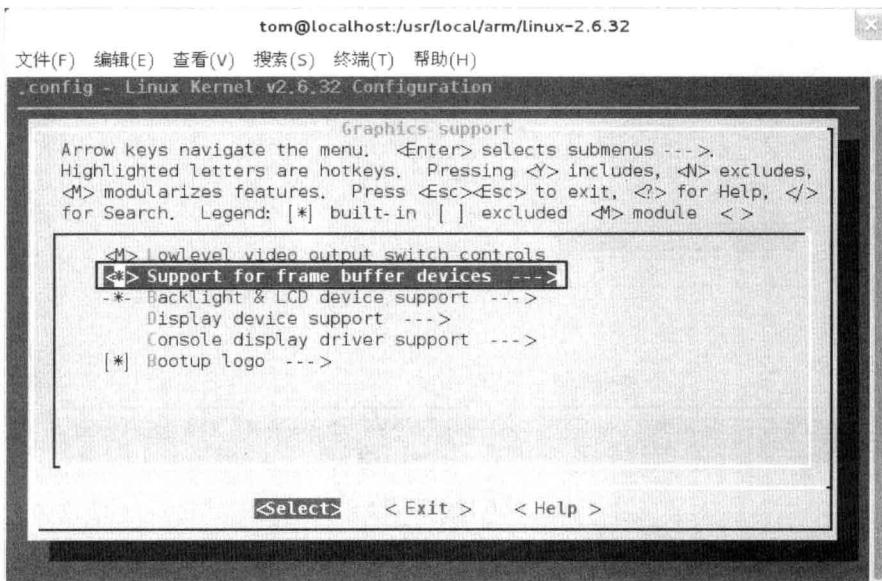


图 6.10 步骤 3

(4) 在驱动程序的调试阶段最好选中 S3C2410 lcd debug messages 选项，如图 6.11 所示。这样会在标准输出中打印出调试信息，有助于驱动程序的调试。

(5) 增加开机 LOGO，让系统开机时在 LCD 屏上显示一个小企鹅的图片。在步骤 (3) 中同时选择 Bootup logo 选项，如图 6.12 所示。

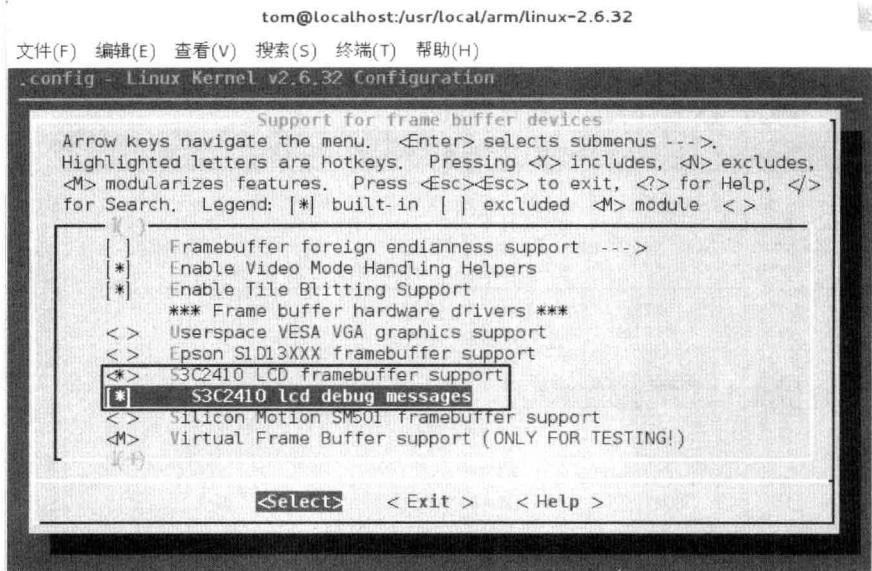


图 6.11 步骤 4

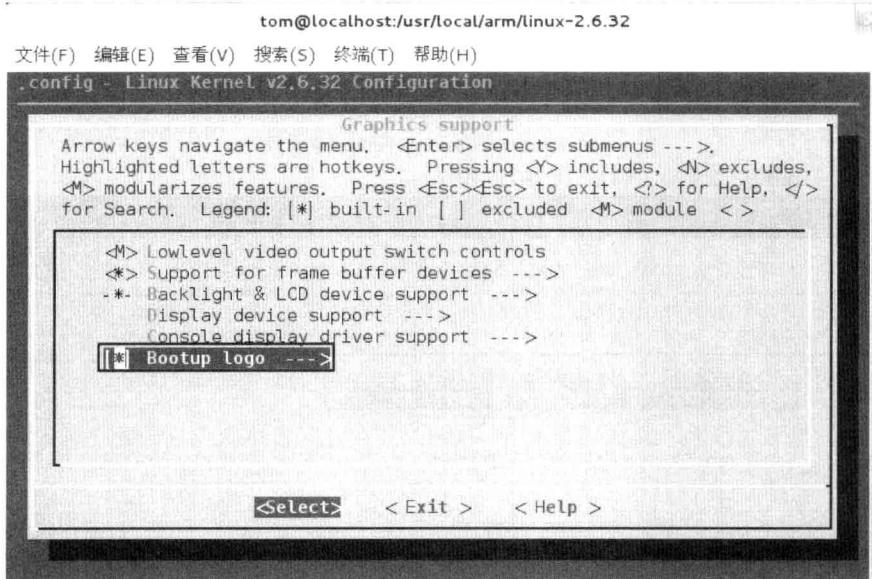


图 6.12 步骤 5

(6) 进入开机 LOGO 的设置选项 Bootup logo 目录后，里面有单色的 LOGO 图片选项，也有 16 色和 224 色的 LOGO 图片选项供选择，这里根据实际情况选择 Standard 224-color Linux logo，如图 6.13 所示。这是因为我们用的 LCD 屏是支持 224 色的。

做完以上配置后保存并退出配置界面，在源码目录上输入 make 重新编译内核。把内核下载到开发板上，然后开机就可以看到在 LCD 屏上有一个企鹅的图片了，说明 LCD 已经可以使用了。

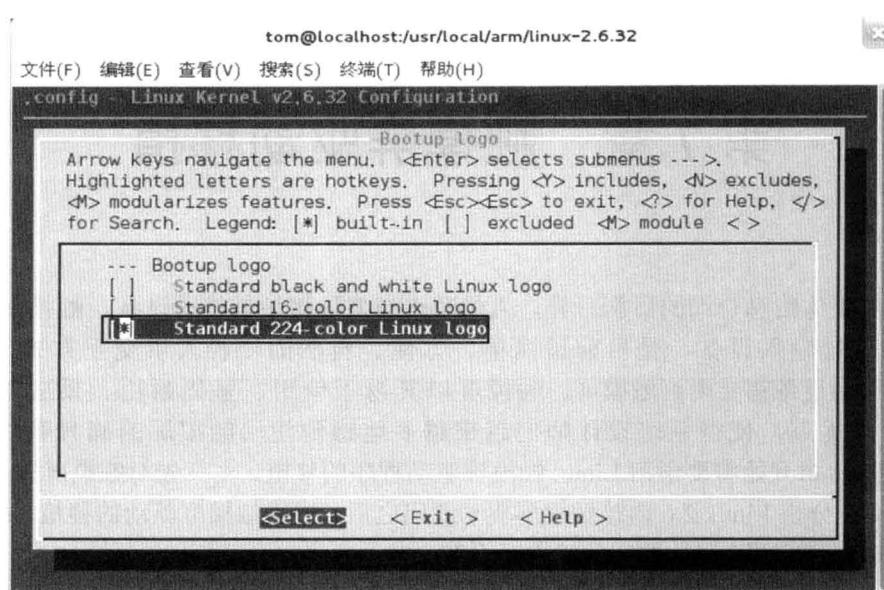


图 6.13 步骤 6

## 6.6 小 结

本章主要在分析了 S3C2440 LCD 控制寄存器的硬件操作、内核中自带的 LCD 驱动源程序的基础上讲述了 LCD 网卡驱动的移植。LCD 驱动程序移植主要是了解内核中 FrameBuffer 驱动程序的体系结构及如何操作 LCD 控制器。因此本章的 6.1 节和 6.2 节是后面移植工作的基础，读者要认真掌握，特别是对 FrameBuffer 数据结构的主要成员要掌握其代表的具体含义。

# 第 7 章 触摸屏驱动移植

随着多媒体信息查询的日新月异，人们越来越多地接触触摸屏设备。触摸屏作为一种新颖的电脑输入设备，是目前最简单、方便、自然的一种人机交互方式，许多现代手持或终端设备都要用到触摸屏。触摸屏以其易于使用、坚固耐用、反应速度快、节省空间等优点，使得系统设计师们越来越多地感到使用触摸屏的确具有相当大的优越性。从本章开始将要学习 Linux 触摸屏驱动程序的移植。本章先对触摸屏做一个简单概述，然后在分析 Linux 2.6 内核触摸屏驱动源码之后，讲述触摸屏驱动的移植，最后结合触摸屏驱动程序介绍 Linux 的内核输入子系统。

## 7.1 触摸屏概述

触控屏（Touch panel）又称为触控面板，它并不是人们日常所见的立方体屏，它只是覆盖显示屏表面的一层薄片，是一个可接收触头等输入信号的感应式液晶显示装置，它的工作原理比较简单。当接触到屏幕上的图形按钮时，屏幕上的触觉反馈系统可根据预先编写的程序驱动连接各种装置，可用以取代机械式的按钮面板，并借由液晶显示画面制造出生动的影音效果。

### 7.1.1 触摸屏工作原理

触摸屏简单地说就是一种特殊的输入设备。为了操作方便，人们用触摸屏取代鼠标或者键盘。在工作时，必须首先用手指或其他物体触摸安装在显示器前面的触摸屏，然后系统根据手指触摸的图标或菜单位置来定位选择信息输入。触摸屏由触摸检测部件和触摸屏控制器组成；触摸检测部件安装在显示器屏幕前面，用来检测用户触摸的位置，接收后送触摸屏控制器；而触摸屏控制器的主要作用是从触摸点检测装置上接收触摸信息，并将它转换成为触点坐标，再送给 CPU 处理，它同时能接收 CPU 发来的命令并加以执行。

### 7.1.2 触摸屏的主要类型

按技术原理来区别触摸屏，可分为以下 5 个基本种类：

- 矢量压力传感式触摸屏；
- 电阻式触摸屏；
- 电容式触摸屏；

- 红外线式触摸屏；
- 表面声波式触摸屏。

其中矢量压力传感式触摸屏已退出历史舞台，这里不再进行介绍；红外线技术触摸屏价格低廉，但它的外框易碎，比较容易产生光干扰，曲面情况下失真；电容技术触摸屏设计构思合理，但其图像失真问题很难得到根本解决；电阻技术触摸屏的定位准确，但其价格颇高，且怕刮易损；表面声波触摸屏解决了以往触摸屏的各种缺陷，清晰且不容易被损坏，适用于各种场合，缺点是屏幕表面如果有水滴和尘土会使触摸屏变得迟钝，甚至不工作。根据触摸屏的工作原理和传输信息的介质，可以把触摸屏分为4种，它们分别为电阻式、电容感应式、红外线式以及表面声波式。每一类触摸屏都有它们各自的优缺点，要了解各种触摸屏适用的场合，关键就在于要懂得每一类触摸屏技术的工作原理和特点。下面对后面4种类型的触摸屏进行简要介绍。

### 1. 电阻触摸屏

电阻触摸屏的屏体部分是一块与显示器表面相匹配的多层复合薄膜，由一层有机玻璃作为基层，表面还涂有一层透明的导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层透明导电层，在两层导电层之间有许多细小（小于千分之一英寸）的透明隔离点把它们隔开绝缘，如图7.1所示。

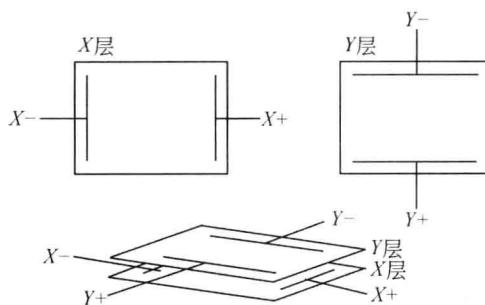


图7.1 电阻触摸屏示意图

当手指点击屏幕时，平时相互绝缘的两层导电层就在触摸点位置产生了一个接触，因为其中一面导电层接通Y轴方向的5V均匀电压场，使得侦测层的电压由0变成了非零，这种接通状态被控制器侦测到后，进行A/D转换，并将得到的电压值与5V相比就可得到触摸点的Y轴坐标，同理得出X轴的坐标，这就是所有电阻技术触摸屏共同的最基本原理。电阻类触摸屏的关键是材料科技。电阻屏根据引出的线数多少，可分为四线、五线、六线等多线电阻触摸屏。电阻式触摸屏在强化玻璃表面分别涂上了两层OTI透明氧化金属导电层，在最外面的OTI涂层作为导电体，第二层OTI则经过精密的网络附上横竖两个方向的+5V~0V的电压场，两层OTI之间用细小的透明隔离点隔开。当手指接触屏幕时，两层OTI导电层之间就会出现一个接触点，程序同时检测电压及电流，计算出触摸的位置，其反应速度为10~20ms。

五线式电阻触摸屏的外层导电层使用的是延展性较好的镍金涂层材料，由于频繁触摸

外导电层，使用延展性好的镍金材料可以延长使用寿命，但是工艺成本较为高昂。镍金导电层虽然延展性较好，但是只能作为透明导体，不适合作为电阻触摸屏的工作面，因为它导电率高，而且金属做到厚度非常均匀不容易，不适合做电压分布层，只能做探层。

电阻式触摸屏工作在一种对外界完全隔离的环境，不怕灰尘及水汽，其可以用任何物体来触摸，可以用来写字画画，比较适合工业控制领域及办公室内有限人的使用。电阻式触摸屏共同的缺点是复合薄膜的外层采用塑胶材料，不清楚原理的人太用力或使用锐器触摸可能划伤整个触控屏而导致报废。不过在限度之内，划伤只是伤及外导电层，外导电层的划伤对于五线电阻触摸屏来说是没有关系的，而对四线电阻触摸屏来说是致命的。

## 2. 电容式触摸屏

电容式触摸屏的构造主要是在玻璃屏幕上镀一层透明的薄膜层，再在导体层外加上了一块保护玻璃，双层玻璃设计能够很好地保护导体层及感应器。

电容式触摸屏在触摸屏的四边均镀上狭长的电极，在导体内形成一个低电压交流电场。用户接触屏幕时，由于人体电场，手指与导体层间会形成一个耦合电容，四个电极发出的电流会流向触点，而电流强弱跟手指到电极的距离成正比，位于触摸屏幕后的控制器便会计算电流的比例及强弱，准确算出触摸点的位置。电容触摸屏的双玻璃不仅能保护导体及感应器，而且更有效地防止了外在环境因素对触摸屏造成的影响，就算屏幕沾有污秽、尘埃或油渍，电容式触摸屏仍然可以准确地算出触摸位置。

电容式触摸屏在玻璃表面贴上一层透明的特殊金属导电物质。当手指接触在金属层上时，触点的电容就发生变化，使得与其相连的振荡器频率发生变化，通过测量频率变化可以确定触摸位置从而获得信息。由于电容随温度、湿度或接地情况的不同而变化，故其稳定性较差，经常会产生漂移现象。

## 3. 红外线式触摸屏

这种触摸屏由装在触摸屏外框上的红外线发射与接收感测元件构成，在屏幕表面上形成红外线探测网，任何触摸物体都可以改变触点上的红外线而实现触摸操作。红外触摸屏不受电流、电压和静电干扰，适宜某些恶劣的环境条件。其主要优点是价格低廉、安装方便、不需要卡或其他任何控制器，可以用在各档次的计算机上。此外，由于没有电容充、放电过程，其响应速度比电容式快，但分辨率较低。

红外线触摸屏原理也比较简单，只是在显示屏上加上光点距架框，不需要在屏幕表面上加涂层或接驳控制器。光点距架框的四边排列了红外线发射管和接收管，在屏幕的表面形成了一个红外线网。用户以手指触摸屏幕某一点，就会挡住经过该位置的横竖两条红外线，计算机程序便可即时算出触摸点位置。因为红外触摸屏不受电流、电压和静电干扰，所以适宜某些恶劣的环境条件。但是，因为只是在普通屏幕增加了框架，所以在使用过程中框架四周的红外线发射管及接收管很容易损坏。

## 4. 表面声波触摸屏

表面声波是一种沿着介质表面传播的机械波。这种触摸屏由触摸屏、声波发生器、反射器和声波接收器组成。其中，声波发生器能发送一种高频声波跨越屏幕的表面，当手指

触到屏幕时，触点上的声波就被阻止，由此确定触点坐标的位置。表面声波触摸屏不受温度、湿度等环境因素影响，分辨率很高，有很好的防刮性，寿命长（5000万次无故障）；透光率高（92%），能保持清晰透亮的图像质量；没有漂移，只需安装时一次校正；有第三轴（即压力轴）响应，最适合公共场所使用。表面声波触摸屏的触摸屏部分可以是一块平面、球面或是柱面的玻璃平板，安装在CRT、LED、LCD或是等离子显示器屏幕的前面。该玻璃平板只是一块纯粹的强化玻璃，区别于其他有触摸屏和覆盖层的触摸屏技术。玻璃屏的左上角和右下角各固定了竖直和水平方向的超声波发射换能器，右上角固定了两个相应的超声波接收换能器。玻璃屏的四个周边则刻有45°角由疏到密间隔非常精密的反射条纹。

## 7.2 S3C2440 ADC 接口使用

写设备驱动程序一般都是和具体的处理器芯片和接口打交道，本章所使用的依然是S3C2440芯片。S3C2440的触摸屏控制器是和ADC（模数转换控制器）结合在一起的，因此本节将介绍S3C2440的ADC及其触摸屏接口。

### 7.2.1 S3C2440 触摸屏接口概述

S3C2440具有8通道模拟输入的10位CMOS模数转换器（ADC），它将输入的模拟信号转换为10位的二进制数字码。在2.5MHz的A/D转换器时钟下，最大转化速率可以达到500KSPS。A/D器支持片上采样和保持功能，并支持掉电模式。

此外，S3C2440的AIN[7]和AIN[5]用于连接触摸屏的模拟信号输入。触摸屏接口电路一般由触摸屏、4个外部晶体管和1个外部电压源组成，如图7.2所示。

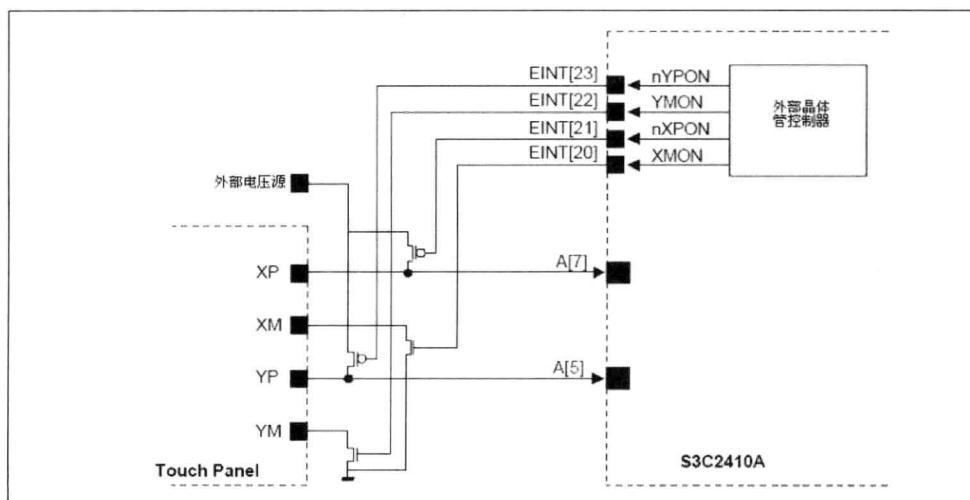


图7.2 触摸屏接口电路示意图

触摸屏接口的控制和选择信号有 nYPON、YMON、nXPONT 和 XMON，它们连接切换 X 坐标和 Y 坐标转换的外部晶体管。模拟输入引脚（AIN[7]、AIN[5]）则连接到触摸屏引脚。

触摸屏控制接口包括一个外部晶体管控制逻辑和具有路数产生逻辑的 ADC 接口逻辑。

### 7.2.2 S3C2440 触摸屏接口操作

图 7.3 是 S3C2440 上的 A/D 转换器和触摸屏接口的功能框图。这个 A/D 转换器是一个循环类型的。上拉电阻接在 VDDA-ADC 和 AIN[7]之间。因此，触摸屏的 X+脚应该接到 S3C2440 的 AIN[7]，Y+脚则接到 S3C2440 的 AIN[5]。

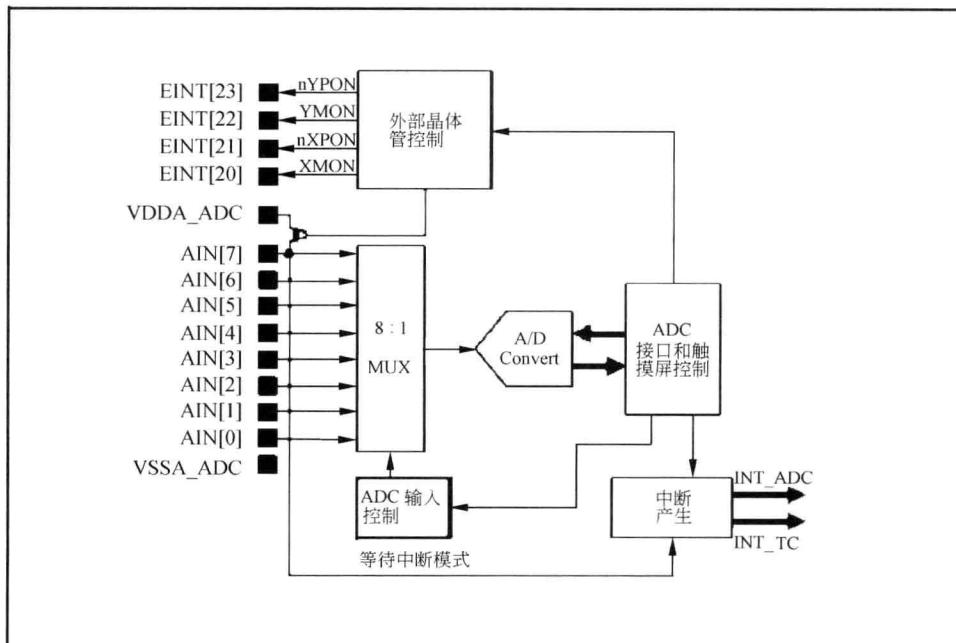


图 7.3 ADC 和触摸屏接口结构图

从图 7.3 可以知道，ADC 和触摸屏接口中只有一个 A/D 转换器，可以通过设置寄存器来选择对哪路模拟信号进行采样。图中有两个中断信号：INT\_ADC 和 INT\_TC，INT\_ADC 表示 A/D 转换器已经转换完毕，而 INT\_TC 则表示触摸屏被按下。

在使用触摸屏时，引脚 XP、XM、YP、YM 被用于和触摸屏直接相连，只剩下 AIN[3:0] 共 4 个引脚用于一般的 ADC 输入；当不使用触摸屏时，XP、XM、YP 和 YM 这 4 个引脚也可以用于一般的 ADC 输入。

#### 1. S3C2440 触摸屏控制器工作模式

S3C2440 的触摸屏控制器是和 A/D 转换控制器结合在一起的，触笔的位置通过模拟信号传递给 A/D 转换器，A/D 转换完成后，把结果保存在相应的寄存器。根据转换方式的不同，触摸屏控制器有以下 4 种工作模式。

### (1) 等待中断模式

设置 ADCTSC 寄存器为 0XD3 即可令触摸屏控制器处于等待中断模式。这时，它在等待触摸屏被按下。当触摸屏被按下时，触摸屏控制器就发出 INT\_TC 中断信号，这时触摸屏控制器要转入下面两种工作模式中的一种，来读取 x、y 方向的坐标。

当设置 ADCTSC 寄存器的位[8]为 0 时，表示等待 Pen Down 中断。

当设置 ADCTSC 寄存器的位[8]为 1 时，表示等待 Pen Up 中断。

等待中断模式下触摸屏引脚状况，如表 7.1 所示。

表 7.1 等待中断模式下的触摸屏引脚状况

	XP	SM	YP	YM
等待中断模式	上拉	高阻	AIN[5]	接地

### (2) 分离 x/y 轴坐标模式

设置 ADCTSC 寄存器为 0X69 进入 x 轴坐标转换模式，x 坐标值转换完毕后被写入 ADCDAT0，然后发出 INT\_ADC 中断；同样地，设置 ADCDAT0 寄存器为 0X9A 进入 y 轴坐标转换模式，y 坐标值转换完毕后被写入 ADCDAT1，接着发出 INT\_ADC 中断信号。分离 x/y 轴坐标模式下触摸屏引脚状况如表 7.2 所示。

表 7.2 分离 x/y 轴坐标模式下的触摸屏引脚状况

	XP	SM	YP	YM
X	接到外部电压	接地	接 AIN[5]	高阻
Y	接 AIN[7]	高阻	接外部电压	接地

### (3) 自动 x/y 轴坐标转换模式

设置 ADCTSC 寄存器为 0X0C，则进入自动 x/y 轴坐标转换模式，触摸屏控制器会自动转换触点的 x/y 坐标值，然后分别保存在 ADCDAT0 和 ADCDAT1 寄存器中，之后发出 INT\_ADC 中断信号。自动 x/y 轴坐标转换模式下的触摸屏引脚状况如表 7.3 所示。

表 7.3 自动 x/y 轴坐标转换模式下的触摸屏引脚状况

	XP	SM	YP	YM
X	接到外部电压	接地	接 AIN[5]	高阻
Y	接 AIN[7]	高阻	接外部电压	接地

### (4) 普通转换模式

这是一种普通的 A/D 转换，在不使用触摸屏时，触摸屏控制器处于这种工作模式。在这种模式下，可以通过设置 ADCCON 寄存器启动普通的 A/D 转换，转换结束后数据就被保存在 ADCDAT0 寄存器中。

## 2. S3C2440 触摸屏接口专用寄存器

S3C2440 触摸屏接口涉及的专用寄存器比较少，主要有 ADCCON、ADCTSC、ADCDAT0 和 ADCDAT1。下面分别对它们进行介绍。

### (1) ADCCON 控制寄存器

它主要用来设置触摸屏的 A/D 转换方式，普通的 A/D 转换有 8 个输入通道，这 8 个通道是 AIN0~AIN7。其中 AIN5 和 AIN7 用于作为触摸屏的 x 和 y 方向的输入通道，具体描述如表 7.4 所示。

表 7.4 ADCCON控制寄存器描述

ADCCON	位	描 述
DCFLG	[15]	A/D 转换结束标志 0: 正在转换; 1: 转换结束
PRSCEN	[14]	A/D 转换预分频使能 0: 不使能; 1: 使能
PRSCVLF	[13:6]	A/D 转换预分频器数值 数据值范围: 1~255 当预分频为 N 时, 则除数实际上为 (N+1) 注意: ADC 频率应该设置成小于 PLCK 的 5 倍 (例如: 如果 PCLK=10MHz, ADC 频率小于 2 MHz)
SEL_MUX	[5:3]	选择模拟输入通道 000: AIN0; 001: AIN1 010: AIN2; 011: AIN3 100: AIN4; 101: AIN5 110: AIN6; 111: AIN7
STDBM	[2]	选择静止模式 0: 正常模式; 1: 静止模式
READ_START	[1]	通过读取来启动 A/D 转换 0: 不启动; 1: 启动
ENABLE_START	[0]	通过设置该位来启动 A/D 操作。如果 READ_START 是使能的, 这个值就是无效的 0: 无操作; 1: A/D 转换启动, 启动后该位被清 0

## (2) ADC 触摸屏控制寄存器 ADCTSC

它主要用来选择触摸屏的工作模式。通过设置 ADCTSC 控制寄存器的值来设置触摸屏引脚状况, 从而可以设置触摸屏的工作模式。ADCTSC 控制寄存器的具体描述如表 7.5 所示。

表 7.5 ADCTSC控制寄存器描述

ADCTSC	位	描 述
Reserved	[8]	此位表示将检测哪类中断 0: 按下; 1: 松开
YM_SEN	[7]	选择 YM <sub>MON</sub> 的输出值 0: YM <sub>MON</sub> 输出是 0 (YM=高阻) 0: YM <sub>MON</sub> 输出是 1 (YM=GND)
YP_SEN	[6]	选择 nYP <sub>ON</sub> 的输出值 0: nYP <sub>ON</sub> 输出是 0 (YP=外部电压) 0: nYP <sub>ON</sub> 输出是 1 (YP 连接到 AIN[5])
XM_SEN	[5]	选择 XM <sub>MON</sub> 的输出值 0: XM <sub>MON</sub> 输出是 0 (XM=高阻) 0: XM <sub>MON</sub> 输出是 1 (XM=GND)
SP_SEN	[4]	选择 nXP <sub>ON</sub> 的输出值 0: nXP <sub>ON</sub> 输出是 0 (XP=外部电压) 0: nXP <sub>ON</sub> 输出是 1 (XP 连接到 AIN[7])
PULL_UP	[3]	上拉切换使能 0: XP 上拉使能 1: XP 上拉禁止

续表

ADCTSC	位	描 述
AUTO_PST	[2]	自动连续转换 X、Y 轴坐标 0: 普通 ADC 转换 1: 自动转换
XY_PST	[1:0]	手动测量 X、Y 轴坐标 00: 无操作模式; 01: 对 X 轴坐标进行测量 10: 对 Y 轴坐标测量; 11: 等待中断模式

### (3) ADCDAT0 和 ADCDAT1 寄存器

这两个寄存器用来设置要转换的坐标和保存坐标的转换结果。X 轴的坐标转换结果会写到 ADCDAT0 寄存器的 XPDAT 中, 等待转换完成后, 触摸屏控制器会产生相应的中断; Y 轴的坐标转换结果会写到 ADCDAT1 寄存器的 YPDAT 中, 等待转换完成后, 触摸屏控制器会产生相应的中断。ADCDAT0 和 ADCDAT1 寄存器的具体描述如表 7.6 和表 7.7 所示。

表 7.6 ADCDAT0 寄存器描述

ADCDAT0	位	描 述
UPDOWN	[15]	等待中断模式下触笔的点击或抬起状态 0: 触笔按下状态 1: 触笔抬起状态
AUTO_PST	[14]	自动 X/Y 轴坐标转换模式 0: 普通 ADC 转换 1: X/Y 轴坐标转换
XY_PST	[13:12]	手动 X/Y 坐标转换模式 00: 无操作; 01: X 轴坐标转换 10: Y 轴坐标转换; 11: 等待中断模式
保留	[11:10]	保留
XPDATA	[9:0]	X 坐标的转换数据值

表 7.7 ADCDAT1 寄存器描述

ADCDAT1	位	描 述
UPDOWN	[15]	等待中断模式下触笔的点击或抬起状态 0: 触笔按下状态 1: 触笔抬起状态
AUTO_PST	[14]	自动 X/Y 轴坐标转换模式 0: 普通 ADC 转换 1: X/Y 轴坐标转换
XY_PST	[13:12]	手动 X/Y 坐标转换模式 00: 无操作; 01: X 轴坐标转换 10: Y 轴坐标转换; 11: 等待中断模式
保留	[11:10]	保留
YPDATA	[9:0]	Y 坐标的转换数据值

### (4) ADC 起始延迟寄存器 (ADCDLY)

ADCDLY 只有前 16 位有效, 在正常转换模式, 独立 X/Y 位置转换模式和自动 X/Y 位

置转换模式下，X/Y 位置转换延迟值；当在等待中断模式中有触笔按下时，这个寄存器在间歇的几毫秒时间内，为自动 X/Y 位置转换产生中断信号（INT\_TC），好处在于在等待中断的时候还可以进行 AD 转换。

## 7.3 2.6 内核触摸屏驱动源码分析（s3c2410\_ts.c 源码分析）

Linux 2.6.33 的内核源码中已经包含了触摸屏的相驱动了，驱动开发人员只要了解了内核的 LCD 驱动体系结构，然后参考内核中已有 LCD 驱动源码，再针对具体的触摸屏型号和硬件资源做相关修改就可以了。因此，本节开始将要分析 Linux 2.6.33 的触摸驱动源码，对应的源代码在 drivers/input/touchscreen/s3c2410\_ts.c 中，这是一个对应于 S3C2410 片的触摸屏驱动代码，只要稍做修改就可以在 S3C2440 芯片上使用了。

文件 drivers/input/touchscreen/s3c2410\_ts.c 是内核针对 S3C2410 芯片而设计的驱动程序，这个驱动程序也是一个平台驱动结构，通过向内核注册 device\_driver 结构初始化触摸屏。device\_driver 结构必须实现两个函数，分别是 probe 和 remove，在这里分别对应于 s3c2410ts\_probe 和 s3c2410ts\_remove。s3c2410ts\_probe 是一个初始化函数，主要功能是完成资源的获取和对硬件初始化，下面将分别介绍。

### 1. s3c2410ts\_probe 分析

s3c2410ts\_probe 是一个探测函数，在这个函数中完成了硬件资源获取、GPIO 口的初始化、中断申请和注册驱动程序等操作。下面跟着程序流程一步步分析。

```
static int __init s3c2410ts_probe(struct device *dev)
{
    /*结构 s3c2410_ts_mach_info 用于保存触摸屏的特定数据，里面存放的是触摸屏需要的一些设置参数，如分频比和延时等参数*/
    struct s3c2410_ts_mach_info *info;

    info = (struct s3c2410_ts_mach_info *)dev->platform_data;
                    //从传进来的平台数据中获取硬件特定数据

    if (!info)
    {
        printk(KERN_ERR "Hm... too bad : no platform data for ts\n");
        return -EINVAL;           //出错返回
    }

#ifdef CONFIG_TOUCHSCREEN_S3C2410_DEBUG
    printk(DEBUG_LVL "Entering s3c2410ts_init\n"); //这是一些调试信息的打印
#endif

    adc_clock = clk_get(NULL, "adc");
    /*获取时钟，挂载 APB BUS 上的外围设备，需要时钟控制，ADC 就是这样的设备*/
    if (!adc_clock) {
        printk(KERN_ERR "failed to get adc clock source\n");
        return -ENOENT;
    }
    //clk_use(adc_clock);
    clk_enable(adc_clock);
```

```

#define CONFIG_TOUCHSCREEN_S3C2410_DEBUG
    printk(DEBUG_LVL "got and enabled clock\n");
#endif

/*开始映射 I/O 内存，I/O 内存是不能直接进行访问的，必须对其进行映射，为 I/O 内存分配虚拟地址，这些虚拟地址以__iomem 进行说明，但不能直接对其进行访问，需要使用专用的函数，如 iowrite320()*/
    base_addr=ioremap(S3C2410_PA_ADC,0x20);
    if (base_addr == NULL) {
        printk(KERN_ERR "Failed to remap register block\n");
        return -ENOMEM;
    }

/* Configure GPIOs */
s3c2410_ts_connect(); //设置触摸屏相关的 4 个 GPIO 为特殊功能

if ((info->presc&0xff) > 0)
    writel(S3C2410_ADCCON_PRSCEN | S3C2410_ADCCON_PRSCVL(info->presc
    &0xFF), \
            base_addr+S3C2410_ADCCON); //使能预计分频和设置分频系数
else
    writel(0,base_addr+S3C2410_ADCCON);

/* Initialise registers */
if ((info->delay&0xffff) > 0)
    writel(info->delay & 0xffff, base_addr+S3C2410_ADCDLY);
    /*设置 ADC 延时，在等待中断模式下表示产生 INT_TC 的间隔时间*/

writel(WAIT4INT(0), base_addr+S3C2410_ADCTSC);
    //按照等待中断的模式设置 TSC

```

下面开始初始化 s3c2410ts 结构体，s3c2410ts 结构体定义如下：

```

struct s3c2410ts {
    struct input_dev dev; //定义输入设备
    long xp; //X 轴坐标
    long yp; //Y 轴坐标
    int count; //统计采样次数
    int shift; //用于根据采样次数求平均值
    char phys[32]; //设备名称
};

memset(&ts, 0, sizeof(struct s3c2410ts));
init_input_dev(&ts.dev);

```

设置事件类型，下面是设置事件类型的代码，要理解这些代码，需先理解事件类型。常用的事件类型有 EV\_KEY、EV\_MOSSE，EV\_ABS（用来接收像触摸屏这样的绝对坐标事件），而每种事件又会有不同类型的编码 code，比如 ABS\_X、ABS\_Y，这些编码又会有相应的值，关于内核的输入子系统在后面详细介绍。

```

ts.dev.evbit[0] = ts.dev.evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) |
BIT(EV_ABS);
ts.dev.keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
input_set_abs_params(&ts.dev, ABS_X, 0, 0x3FF, 0, 0);
input_set_abs_params(&ts.dev, ABS_Y, 0, 0x3E8, 0, 0);
input_set_abs_params(&ts.dev, ABS_PRESSURE, 0, 1, 0, 0);

```

```

        sprintf(ts.phys, "ts0"); //填写设备名称
/*以上是输入设备的名称和 ID, 这些信息是输入设备的身份信息*/
        ts.dev.private = &ts;
        ts.dev.name = s3c2410ts_name;
        ts.dev.phys = ts.phys;
        ts.dev.id.bustype = BUS_RS232;
        ts.dev.id.vendor = 0xDEAD;
        ts.dev.id.product = 0xBEEF;
        ts.dev.id.version = S3C2410TSVERSION;

        ts.shift = info->oversampling_shift; //设置采样次数

```

下面开始注册中断处理进程。stylus\_action 和 stylus\_updown 是两个中断处理函数，当笔尖触摸时，会进入到 stylus\_updown。这里申请了触摸屏相关的两个中断，一个是 IRQ\_TC 中断，查阅了数据手册后了解到，这个中断在笔按下时，由 XP 管脚产生表示中断的低电平信号，而笔抬起是没有中断信号产生的。另一个是 IRQ\_ADC\_DONE 中断，该中断是当芯片内部 A/D 转换结束后，通知中断控制器产生中断，此时就可以去读取转换得到的数据。

```

//if (request_irq(IRQ_ADC, stylus_action, SA_SAMPLE_RANDOM | SA_SHIRQ,
if (request_irq(IRQ_ADC, stylus_action, SA_SAMPLE_RANDOM,
    "s3c2410_action", &ts.dev)) {
    printk(KERN_ERR "s3c2410_ts.c: Could not allocate ts IRQ_ADC !\n");
    iounmap(base_addr);
    return -EIO;
} //ADC 转换中断, 转换结束后触发
if (request_irq(IRQ_TC, stylus_updown, SA_SAMPLE_RANDOM,
    "s3c2410_action", &ts.dev)) {
    printk(KERN_ERR "s3c2410_ts.c: Could not allocate ts IRQ_TC !\n");
    iounmap(base_addr);
    return -EIO;
} //TSC 中断, 触笔动作触发

printk(KERN_INFO "%s successfully loaded\n", s3c2410ts_name);

/* All went ok, so register to the input system */
input_register_device(&ts.dev); //注册输入设备
printk(KERN_INFO "%s input_register_device\n", s3c2410ts_name);
return 0;
}

```

## 2. touch\_timer\_fire 分析

touch\_timer\_fire() 函数主要实现以下功能：

- stylus down 的时候，touch\_timer\_fire() 函数在中断函数 stylus\_updown 里被调用，此时缓存区没有数据，ts.count 的值为 0，所以只是简单地设置 A/D 转换的模式，然后开启 A/D 转换。
- 当 ADC 中断函数 stylus\_action() 把缓冲区填满时，作为中断后半段函数稍后被调用，此时 ts.count 等于 shift，算出其平均值后，交给事件处理器层（Event Handler）处理，主要是填写缓冲，然后唤醒等待输入数据的进程。
- stylus 抬起，等到缓冲区填满后（可能会包含一些无用的数据）被调用，这时判断出 stylus up，报告 stylus up 事件，重新等待 stylus down。

下面分析具体代码。

```

static void touch_timer_fire(unsigned long data)
{
    unsigned long data0;           //保存X轴的坐标
    unsigned long data1;           //保存Y轴的坐标
    int updown;                   //保存触笔按下或抬起，按下为1

    data0 = readl(base_addr+S3C2410_ADCDAT0);      //读X轴的坐标
    data1 = readl(base_addr+S3C2410_ADCDAT1);      //读Y轴的坐标

    updown = (!(data0 & S3C2410_ADCDAT0_UPDOWN)) && (!(data1 & S3C2410_ADCDAT0_UPDOWN));          //测试触笔是否按下

/*触笔按下后，开始对X和Y轴的坐标进行A/D转换，为了求得比较精确的坐标值，可以进行多次转换后取平均值。这里的ts.shift就是转换（采样）的次数*/
    if (updown) {                                //触笔按下
        if (ts.count != 0) {                      //多次采样已经完成
            long tmp;

            tmp = ts.xp;
            ts.xp = ts.yp;
            ts.yp = tmp;

            ts.xp >= ts.shift;                  //求平均值
            ts.yp >= ts.shift;
        }
    }

/*调试信息*/
#ifdef CONFIG_TOUCHSCREEN_S3C2410_DEBUG
{
    struct timeval tv;
    do_gettimeofday(&tv);
    printk(DEBUG_LVL "T: %06d, X: %03ld, Y: %03ld\n", (int)tv.tv_usec, ts.xp, ts.yp);
}
#endif

/*下面两句是报告X、Y的绝对坐标值*/
input_report_abs(&ts.dev, ABS_X, ts.xp);
input_report_abs(&ts.dev, ABS_Y, ts.yp);

/*报告按键事件，键值为1（代表触摸屏对应的按键被按下）*/
input_report_key(&ts.dev, BTN_TOUCH, 1);

/*报告触摸屏的状态，1表明触摸屏被按下*/
input_report_abs(&ts.dev, ABS_PRESSURE, 1);

/*等待接收方收到数据后回复确认，用于同步*/
input_sync(&ts.dev);
}

/*如果触笔是刚刚按下的，那么ts.count的值为0，此时要清空之前保存的数值。有一种情况，当触笔在屏幕上拖动时，会不停地采样并报告坐标值*/
    ts.xp = 0;
    ts.yp = 0;
    ts.count = 0;

    writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, base_addr+S3C2410_ADCTSC); //设置自动转换
    writel(readl(base_addr+S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_

```

```

        START, base_addr+S3C2410_ADCCON); //开始 A/D 转换
    } else { //这种情况是触笔抬起的时候

        ts.count = 0;

        /* 报告按键事件，键值为 1（代表触摸屏对应的按键被释放）*/
        input_report_key(&ts.dev, BTN_TOUCH, 0);

        /* 报告触摸屏的状态，0 表明触摸屏没被按下 */
        input_report_abs(&ts.dev, ABS_PRESSURE, 0);
        input_sync(&ts.dev);

        /* 进入 s3c2410 触摸屏提供的等待中断模式，等待触笔按下 */
        writel(WAIT4INT(0), base_addr+S3C2410_ADCTSC);
    }
}

```

### 3. stylus\_updown分析

当有触笔按下屏幕时，触摸屏会产生触摸屏中断，这时函数 stylus\_updown()就会被调用，从而进入中断服务，这是在 s3c2410ts\_probe()函数设置中断调用时就已设置好了的。stylus\_updown 主要完成触笔动作的检查工作，具体分析如下：

```

static irqreturn_t stylus_updown(int irq, void *dev_id, struct pt_regs
*regs)
{
    unsigned long data0;           //用于保存 ADCDAT0 的值
    unsigned long data1;           //用于保存 ADCDAT1 的值
    int updown;                   //用于保存触笔动作

    data0 = readl(base_addr+S3C2410_ADCDAT0); //读 ADCDAT0 的值
    data1 = readl(base_addr+S3C2410_ADCDAT1); //读 ADCDAT1 的值

    /*再次判断触笔是否真的按下，本来进入这个中断服务程序就说明触笔是已经按下了，这里有延时去抖动的作用*/
    updown = (!(data0 & S3C2410_ADCDAT0_UPDOWN)) && (!(data1 & S3C2410_
    ADCDAT0_UPDOWN));

    /* TODO we should never get an interrupt with updown set while
     * the timer is running, but maybe we ought to verify that the
     * timer isn't running anyways. */

    if (updown)
        touch_timer_fire(0);
    /*判断出 stylus down，调用 touch_timer_fire 函数，从而进入中断的底半部*/

    return IRQ_HANDLED;
}

```

### 4. stylus\_action分析

这是针对 A/D 转换完成的中断处理函数，当 A/D 转换完成时，判断采样是否完成，没完成则继续采样，用 mod\_timer 定时器来设置多次采样的时间间隔，具体分析如下：

```

static irqreturn_t stylus_action(int irq, void *dev_id, struct pt_regs
*regs)

```

```

{
    unsigned long data0;                                //用于保存 x 坐标的值
    unsigned long data1;                                //用于保存 y 坐标的值

    data0 = readl(base_addr+S3C2410_ADCDAT0);          //读 x 坐标的值
    data1 = readl(base_addr+S3C2410_ADCDAT1);          //读 y 坐标的值

    ts.xp += data0 & S3C2410_ADCDAT0_XPDATA_MASK;    //x 坐标的值累加
    ts.yp += data1 & S3C2410_ADCDAT1_YPDATA_MASK;    //y 坐标的值累加
    ts.count++;                                         //计数器递增

    if (ts.count < (1<<ts.shift)) {                  //判断是否完成采样次数
        writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, base_addr+S3C2410_ADCTSC);
        writel(readl(base_addr+S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_START, base_addr+S3C2410_ADCCON); //开始 A/D 转换
    } else {
        mod_timer(&touch_timer, jiffies+1);           //采样完成，调用 touch_timer_fire 报告坐标值
        writel(WAIT4INT(1), base_addr+S3C2410_ADCTSC);
    }

    return IRQ_HANDLED;
}

```

### 5. 3c2410ts\_remove分析

这是一个设备移除函数，当注销时该函数会被调用，它主要完成资源的释放，代码分析如下：

```

static int s3c2410ts_remove(struct device *dev)
{
    disable_irq(IRQ_ADC);                         //禁止 A/D 中断
    disable_irq(IRQ_TC);                          //禁止触摸屏中断
    free_irq(IRQ_TC,&ts.dev);                     //释放触摸屏中断号
    free_irq(IRQ_ADC,&ts.dev);                     //释放 A/D 中断号

    if (adc_clock) {
        clk_disable(adc_clock);                   //关闭时钟
        //clk_unuse(adc_clock);
        clk_put(adc_clock);
        adc_clock = NULL;
    }

    input_unregister_device(&ts.dev); //注销输入设备
    iounmap(base_addr);

    return 0;
}

```

## 7.4 Linux 内核输入子系统介绍

前面分析了 S3C2410 的触摸屏驱动，那么，这里的驱动是怎样和应用程序交互的呢？

这中间就用到 Linux 的输入子系统，现在深入到下一层，对 Linux 的输入子系统进行分析。

### 7.4.1 Input 子系统概述

Linux 系统提供了 Input 子系统，输入子系统由输入子系统核心层（input core）、驱动层和事件处理层（event handler）3 部分组成。输入事件（如鼠标移动、键盘按键被按下、joystick 的移动等）通过 driver ->inputcore -> eventhandler -> userspace 的顺序到达用户空间传给应用程序。按键、触摸屏、键盘、鼠标等输入都可以利用 Input 接口函数来实现设备驱动。

在 Linux 内核中，Input 设备用 input\_dev 结构体描述，使用 Input 子系统实现输入设备驱动程序的时候，驱动的核心工作是向系统报告按键、触摸屏、键盘、鼠标等输入事件（event，通过 input\_event 结构体描述），不需要再关心文件操作接口，因为 Input 子系统已经完成了文件操作接口。驱动报告的事件经过 InputCore 和 Eventhandler 最终到达用户空间。例如，触摸屏将检测到的所有按键都上报给了 Input 子系统。Input 子系统是所有 I/O 设备驱动的中间层，为上层提供了一个统一的接口界面。所以，在终端系统中，我们不需要去管有多少个键盘，多少个鼠标，它只要从 Input 子系统中去取对应的事件（按键、鼠标移位等）就可以了。

### 7.4.2 输入设备结构体

要了解输入设备子系统，就得先了解内核中输入设备的定义，这里先给出内核中 input\_dev 的定义，然后再对其中重要的成员进行描述。内核中 input\_dev 的定义如下：

```
struct input_dev {
    /* private: */
    void *private; /* do not use */
    /* public: */

    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    unsigned int keycodemax;
    unsigned int keycodesize;
    void *keycode;
    int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
    int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
};
```

```

struct ff_device *ff;

unsigned int repeat_key;
struct timer_list timer;

int sync;

int abs[ABS_MAX + 1];
int rep[REP_MAX + 1];

unsigned long key[BITS_TO_LONGS(KEY_CNT)];
unsigned long led[BITS_TO_LONGS(LED_CNT)];
unsigned long snd[BITS_TO_LONGS(SND_CNT)];
unsigned long sw[BITS_TO_LONGS(SW_CNT)];

int absmax[ABS_MAX + 1];
int absmin[ABS_MAX + 1];
int absfuzz[ABS_MAX + 1];
int absflat[ABS_MAX + 1];

int (*open)(struct input_dev *dev);
void (*close)(struct input_dev *dev);
int (*flush)(struct input_dev *dev, struct file *file);
int (*event)(struct input_dev *dev, unsigned int type, unsigned int code,
int value);

struct input_handle *grab;

spinlock_t event_lock;
struct mutex mutex;

unsigned int users;
int going_away;

struct device dev;

struct list_head h_list;
struct list_head node;
};

```

下面分析几个重要的字段，这是在驱动程序中经常用到的。

### 1. private字段

在 Input 结构中，这个字段可以被用来指向在输入设备驱动程序中的任何私有数据结构，例如在驱动处理多个设备时，在 open() 和 close() 函数中需要此字段。

### 2. ID和name字段

在注册输入设备前，驱动程序应该设置 dev->name。它是一个字符串，例如 Generic button device，包含了一个设备的名字。

ID 字段包含了总线 ID(PCI、USB, ...), 供应商 ID 和设备的设备 ID。总线 ID 在 input.h 文件中定义。供应商和设备 ID 在 pci\_ids.h、usb\_ids.h 和相似的头文件中被定义。这些字段应该在注册输入设备前被驱动程序设置。

ID 和 name 字段可以通过 evdev 接口传递给用户空间使用。

### 3. keycode、keycodemax、keycodesize字段

这3个字段可以用于所有输入设备，被用来报告将产生的数据作为扫描码。如果不是所有的扫描码可以被自动识别所辨别，它们可能需要通过用户空间应用程序设置。这样 keycode 数组被用来映射扫描码到输入系统的键码。keycodemax 包含了数组的大小。keycodesize 表示数组中数据的大小（单位为 bytes）。

### 4. evbit、keybit、relbit、absbit字段

这几个字段是用于设置输入设备的事件类型，EV\_KEY 是最简单的事件类型，用做按键的事件类型。这个事件通过下面函数报告给输入系统：

```
input_report_key(struct input_dev *dev, int code, int value)
```

可以通过文件 linux/input.h 了解所有允许的码值（从 0 到 KEY\_MAX）。参数 value 为布尔值，也就是任何非零值意味着键被按下，0 值表示键被释放。只有在 value 不同于以前的 value 的，输入码产生事件。

除了 EV\_KEY，还有两个基本的事件类型，即 EV\_REL 和 EV\_ABS。它们被用于表示设备提供的相对和绝对值，例如，一个相对值可以是鼠标在 X 轴上的移动距离。鼠标上报此值作为相对于此前最后位置的差值，因为鼠标没有采用任何绝对坐标系统。绝对事件用于摇杆和数字仪（joysticks and digitizers），此类设备在一个绝对坐标系统下工作。

让设备上报 EV\_REL 和上报 EV\_KEY 一样简单，只需设置对应位并调用下面函数，为非零值产生事件报告。

```
input_report_rel(struct input_dev *dev, int code, int value)
```

然而，EV\_ABS 需要一些特殊的处理。在调用 input\_register\_device() 函数之前，必须为设备具有的每个绝对坐标填充 input\_dev 数据结构中相关的字段。假如例子中的按键设备也有 ABS\_X 轴：

```
button_dev.absmin[ABS_X] = 0;
button_dev.absmax[ABS_X] = 255;
button_dev.absfuzz[ABS_X] = 4;
button_dev.absflat[ABS_X] = 8;
```

这个设置可能对摇杆 X 轴是适合的，最小值为 0，最大值为 255，数据误差是±4，中心平滑位置为 8。

如果不需 absfuzz 和 absflat，可以设置它们的值为 0。这样就表示数据是精确的，总是返回到中心位置。

除了介绍的事件类型，其他的事件类型还包括：

```
EV_LED: used for the keyboard LEDs.
EV SND: used for keyboard beeps.
```

这两个非常类似于 EV\_KEY 事件，但是报告方向是相反的，也就是说从系统到输入设备驱动。如果你的输入设备驱动能处理这些事件，则驱动程序必须在 evbit 字段设置相对应的位和设置回调函数：

```

button_dev.event = button_event;
int button_event(struct input_dev *dev, unsigned int type, unsigned int code,
int value);
{
    if (type == EV SND && code == SND_BELL) {
        outb(value, BUTTON_BELL);
        return 0;
    }
    return -1;
}

```

这个回调例程可以在中断上下文或者底半部 BH 中调用，因此不能休眠，也不能花费太长时间去完成。

### 7.4.3 输入链路的创建过程

输入链路的创建过程主要包括硬件设备注册和 input handler 两部分。下面依次讲解。

#### 1. 硬件设备的注册

驱动程序负责和底层的硬件设备打交道，将底层硬件对用户输入的响应转换为标准的输入事件以后再向上发送给 input core。驱动程序通过调用 `input_register_device` 函数和 `input_unregister_device` 函数来向输入子系统中注册和注销输入设备。

这两个函数调用的参数是一个 `input_dev` 结构，这个结构在 `driver/input/input.h` 中定义。驱动程序在调用 `input_register_device` 之前需要填充该结构中的部分字段。如 `s3c2410ts_probe` 函数中的如下代码：

```

init_input_dev(&ts.dev);
ts.dev.evbit[0] = ts.dev.evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) | BIT(EV_ABS);
ts.dev.keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
input_set_abs_params(&ts.dev, ABS_X, 0, 0x3FF, 0, 0);
input_set_abs_params(&ts.dev, ABS_Y, 0, 0x3E8, 0, 0);
input_set_abs_params(&ts.dev, ABS_PRESSURE, 0, 1, 0, 0);
sprintf(ts.phys, "ts0"); //填写设备名称
/*以上是输入设备的名称和 ID，这些信息是输入设备的身份信息了*/
ts.dev.private = &ts;
ts.dev.name = s3c2410ts_name;
ts.dev.phys = ts.phys;
ts.dev.id.bustype = BUS_RS232;
ts.dev.id.vendor = 0xDEAD;
ts.dev.id.product = 0xBEEF;
ts.dev.id.version = S3C2410TSVERSION;

```

#### 2. 注册input handler

驱动程序只是把输入设备注册到输入子系统中，在驱动层的代码中本身并不创建设备结点。应用程序用来与设备打交道的设备结点的创建由 event handler 层调用 input core 中的函数来实现。在创建具体的设备结点之前，event handler 层需要先注册一类设备的输入事件处理函数及相关接口，以 `evdev` 为例，代码在 `evdev.c` 文件中可以找到：

```

static struct input_handler evdev_handler = {
    .event      = evdev_event,

```

```

    .connect      = evdev_connect,
    .disconnect   = evdev_disconnect,
    .fops         = &evdev_fops,
    .minor        = EVDEV_MINOR_BASE,
    .name         = "evdev",
    .id_table     = evdev_ids,
};

static int __init evdev_init(void)
{
    return input_register_handler(&evdev_handler);
}

```

在 `evdev_init` 中调用 `input.c` 中定义的 `input_register_handler` 来注册一个事件类型的 `handler`。这里的 `handler` 不是具体用户可以操作设备，而是事件类设备统一的处理函数接口，如我们所说的触摸屏就是这种类型的设备。

总而言之，整个流程是硬件驱动向 `input` 子系统注册一个硬件设备后，在 `input_register_device` 中调用已经注册的所有类型的 `input handler` 的 `connect()` 函数。每一个具体的 `connect()` 函数会根据注册设备所支持的事件类型判断是否与自己相关，如果相关就调用 `input_register_minor()` 创建一个具体的设备结点。

```

void input_register_device(struct input_dev *dev)
{
    ...
    while (handler) {
        if ((handle = handler->connect(handler, dev)))
            input_link_handle(handle);
        handler = handler->next;
    }
}

```

此外，如果已经注册了一些硬件设备，此后再注册一类新的 `input handler`，则同样会对所有已注册的 `device` 调用新的 `input handler` 的 `connect()` 函数，以确定是否需要创建新的设备结点。

```

void input_register_handler(struct input_handler *handler)
{
    ...
    while (dev) {
        if ((handle = handler->connect(handler, dev)))
            input_link_handle(handle);
        dev = dev->next;
    }
}

```

从上面的分析中可以看到，一类 `input handler` 可以和多个硬件设备相关联，创建多个设备结点。而一个设备也可能与多个 `input handler` 相关联，创建多个设备结点。

#### 7.4.4 使用 Input 子系统

在内核自带的文档 `Documentation/input/input-programming.txt` 中。有一个使用了 `Input` 子系统的例子，并附带相应的说明。下面以这个为例来分析如何使用 `Input` 子系统。

```
#include <linux/input.h>
```

```

#include <linux/module.h>
#include <linux/init.h>

#include <asm/irq.h>
#include <asm/io.h>

static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) & 1);
    //报告按键事件
    input_sync(&button_dev); //等待接收方收到数据后回复确认，用于同步
}

static int __init button_init(void)
{
    if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
        printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
        return -EBUSY;
    }
    //注册中断处理函数

    button_dev.evbit[0] = BIT(EV_KEY); //设置输入设备是按键
    button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0); //有一个按键

    input_register_device(&button_dev); //注册输入设备
}

static void __exit button_exit(void)
{
    input_unregister_device(&button_dev); //注销输入设备
    free_irq(BUTTON_IRQ, button_interrupt); //释放中断
}

module_init(button_init);
module_exit(button_exit);

```

这个示例 module 代码相对比较简单，在初始化函数里注册了一个中断处理例程。然后注册了一个 input device. 在中断处理程序里，将接收到的按键上报到 Input 子系统。

首先，程序必须包含头文件<linux/input.h>，这个文件包含了输入子系统的接口，提供了输入设备所需要的所有定义。

在模块加载或内核启动时调用的\_init()函数中，它申请了所需的资源（它也应该检查设备的存在）。

然后，它设置了输入位域。这是设备驱动告诉输入系统其他部分它是什么的方法，也就是说哪些事件可以被输入设备产生和接收。例子中设备仅仅产生 EV\_KEY 类型事件，对应的事件码是 BTN\_0。这样，可以仅仅设置这两位。也可以通过下面语句完成此功能，当需要设置多位时，例子中使用的方式更简洁。

```

set_bit(EV_KEY, button_dev.evbit);
set_bit(BTN_0, button_dev.keybit);

```

在设置完输入位域后，例子驱动通过语句 input\_register\_device(&button\_dev); 注册输入设备数据结构。它增加 button\_dev 数据结构到输入驱动链表中，并调用设备处理模块 \_connect() 函数去通知它们一个新的输入设备已经被发现。因为 \_connect() 函数可能会调用

可以休眠的 `kmalloc(, GFP_KERNEL)` 函数，所以 `input_register_device()` 函数不能在中断上下文或者拥有自旋锁时调用。

在使用中，驱动中被调用的函数仅是 `button_interrupt()`。一旦从按键产生中断，此函数检查按键的状态，通过 `input_report_key()` 函数调用向输入系统报告。在此，不需要检查中断函数是否通知了两个相同的事件值（例如：按下，按下），因为 `input_report_*` 函数本身完成了此项功能。

通过 `input_sync()` 函数调用通知事件的接收者，已经发送了完整的报告。在一个按键情况下，这个函数看起来并不重要。但在鼠标移动下，此函数就非常重要了。因为你不希望 X 值和 Y 值被分开解释，否则，它们将导致不同的鼠标移动事件。

### 7.4.5 编写输入设备驱动需要完成的工作

从上面这个例子可以看出，通过 Input 子系统，具体的输入设备驱动只需要完成如下工作。

#### 1. 在模块加载函数中告知Input子系统它可以报告的事件

设备驱动通过 `set_bit()` 告诉 Input 子系统它支持哪些事件，如下所示。

```
set_bit(EV_KEY, button_dev.evbit);
set_bit(BTN_0, button_dev.keybit);
```

这两个函数分别用来设置设备所产生的事件及上报的按键值。Struct `input_dev` 中有两个成员，一个是 `evbit`，另一个是 `keybit`，分别用于表示设备所支持的动作和按键类型。也可以像下面一样直接赋值：

```
button_dev.evbit[0] = BIT(EV_KEY); // 设置输入设备是按键
button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0); // 有一个按键
```

#### 2. 在模块加载函数中注册输入设备

设备驱动可以通过 `input_register_device()` 注册一个输入设备，函数原型如下：

```
input_register_device(&button_dev);
```

这个函数用来向内核注册一个输入设备，它的参数类型是一个输入设备结构体，驱动程序在初始化输入设备结构体后调用该函数进行注册。

#### 3. 在键被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时，通过`input_report_xxx()`函数报告发生的事件及对应的键值/坐标等状态

主要的事件类型包括 `EV_KEY`（按键事件）、`EV_REL`（相对值，如光标移动，报告的是相对最后一次位置的偏移）和 `EV_ABS`（绝对值，如触摸屏、操纵杆，它们工作在绝对坐标系统）。

用于报告 `EV_KEY`、`EV_REL` 和 `EV_ABS` 事件的函数分别为：

```
void input_report_key(struct input_dev *dev, unsigned int code, int value);
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
```

```
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
input_sync()
```

其中 `input_sync` 用来告诉上层，本次的事件已经完成了，用做同步。例如，在触摸屏设备驱动中，一次坐标及按下状态的报告过程如下：

```
input_report_abs(input_dev, ABS_X, x); //X坐标
input_report_abs(input_dev, ABS_Y, y); //Y坐标
input_report_abs(input_dev, ABS_PRESSURE, pres); //压力
input_sync(input_dev); //同步
```

#### 4. 在模块卸载函数中注销输入设备

注销输入设备的函数如下：

```
void input_unregister_device(struct input_dev *dev);
```

是前一节内核源码的 `3c2410ts_remove` 函数中用下面函数注销触摸屏输入设备。

```
input_unregister_device(&ts.dev); //注销输入设备
```

## 7.5 触摸屏驱动移植和内核编译

在有些 Linux 内核版本中没有针对 S3C2440 芯片的触摸屏驱动程序(如 Linux 2.6.25)，所以要在 S3C2440 芯片使用触摸屏必须自己编写触摸屏驱动程序，然而不可能从头编写，那样对项目开发进度来说是缓慢的。Linux 的好处是开源，所以可以参考 Linux 已经有的相关驱动程序做简单修改。

当然我们所在的 Linux 2.6.33 中已经包含了针对 S3C2410 芯片的触摸屏驱动程序，前面也已经对该驱动源码进行了分析。这样，对于编写针对 S3C2440 芯片的触摸屏驱动程序就容易多了，简单的触摸屏驱动移植过程如下所述。

### 7.5.1 修改初始化源码

本节主要是几个关键文件，这些文件完成一些硬件初始化工作。读者看这里的时候可以对照内核源代码来看。

#### 1. 修改 `arch/arm/mach-s3c2440/mach-smdk2440.c`

这个文件夹的上部分写的都是各个硬件设备的初始化数据。比如串口的初始化数据定义如下：

```
static struct s3c2410_uartcfg smdk2440_uartcfgs[] __initdata = {
[0] = {
    .hwport     = 0,
    .flags      = 0,
    .ucon       = 0x3c5,
    .ulcon      = 0x03,
    .ufcon      = 0x51,
```

```

},
[1] = {
    .hwport      = 1,
    .flags       = 0,
    .ucon        = 0x3c5,
    .ulcon       = 0x03,
    .ufcon       = 0x51,
},
/* IR port */
[2] = {
    .hwport      = 2,
    .flags       = 0,
    .ucon        = 0x3c5,
    .ulcon       = 0x43,
    .ufcon       = 0x51,
}
;
}

```

该文件的上半部都是此类信息，这类信息多数是在该文件的下半部分会用得到的，对于 Linux 2.6.25.8 以后的内核来说，已经有了触摸屏初始化数定义了。但如果是其他没有触摸屏初始化数定义的内核版本，则必须在这里加入如下代码：

```

static struct s3c2410_ts_mach_info s3c2440_ts_info = {
    .delay = 10000,           //延迟
    .presc = 49,              //分频
    .oversampling_shift = 2,   //采样次数
};

```

在下面数组中加入关于触摸屏的平台数据，这里我们加入上面的 `s3c_device_ts`:

```

static struct platform_device *smdk2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
    &s3c_device_ts, //这个在 devs.c 中定义，看下文
};

```

在下面代码段中加入黑体部分，用于初始化设备：

```

static void __init smdk2440_machine_init(void)
{
    s3c24xx_fb_set_platdata(&smdk2440_fb_info);
    s3c_device_ts.dev.platform_data = &s3c2440_ts_info;
    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}

```

这样，`arch/arm/mach-s3c2440/mach-smdk2440.c` 文件部门的代码就算修改完成了。

## 2. 修改`arch/arm/plat-s3c24xx/devs.c`

这个文件里面全是设备信息。这里常出现的就是 `resource` 类型的数据结构，代表各类资源，在这里加入下面代码，注意，要在文件尾，至少要在 `s3c_adc_resource` 定义之后，因为我们要用到它。

```

/*touch screen*/
struct platform_device s3c_device_ts = {
    .name = "s3c2410-ts",
    .id = -1,
    .num_resources = ARRAY_SIZE(s3c_adc_resource),
    .resource = s3c_adc_resource,
};
EXPORT_SYMBOL(s3c_device_ts)

```

对于上面的 EXPORT\_SYMBOL 解释如下，当想用到 devs.c 中的数据结构时，应该怎么办？#include <devs.c>？没见过有人这样写。实际上真要在其他文件里用到 devs.c 中的东西，比如，我们想用到 s3c\_device\_ts，就在 devs.c 中写 EXPORT\_SYMBOL(s3c\_device\_ts)；正如上面的代码一样，然后，在 devs.h 中写这么一句：

```
extern struct platform_device s3c_device_ts
```

这样，在包含了头文件 devs.h 的 c 文件里，就可以使用这个通过 EXPORT\_SYMBOL 导出来的变量了。正如我们前面 arch/arm/mach-s3c2440/mach-smdk2440.c 中的代码一样；如果 devs.c 中的东西是一个函数的话，我们还要通过“#include <devs.h>”的方式，只需要在 devs.h 中加入函数的原型即可，也不用 EXPORT\_SYMBOL 这样的宏定义。

### 3. 添加头文件

如果使用的内核版本里面没有 reg-adc.h，则需要从其他版本复制 reg-adc.h，文件位置在 include/asm-arm/arch-s3c2410/regs-adc.h，Linux 2.6.25.8 版本是有这个文件的，只要在该文件内添加如下内容，这些内容是用来设置触摸屏的工作模式的。

```

#define S3C2410_ADCTSC_XY_PST_N      (0x0<<0)      //无操作模式
#define S3C2410_ADCTSC_XY_PST_X      (0x1<<0)      //对 X 坐标进行转换
#define S3C2410_ADCTSC_XY_PST_Y      (0x2<<0)      //对 Y 坐标进行转换
#define S3C2410_ADCTSC_XY_PST_W      (0x3<<0)      //等待中断模式

```

编写 s3c2440\_ts.h 文件复制到 include/asm/arch-s3c2410/ 目录下，这个文件用于 S3C2440 触摸屏的平台数据结构，其内容如下所示。

```

#ifndef __ASM_ARM_S3C2440_TS_H
#define __ASM_ARM_S3C2440_TS_H
struct s3c2440_ts_mach_info {
    int delay;           //延迟
    int presc;          //预分频
    int oversampling_shift; //采样次数
};
void __init set_s3c2440ts_info(struct s3c2440_ts_mach_info *hard_s3c2440-
ts_info); /*用于设置私有数据的函数*/
#endif /* __ASM_ARM_S3C2440_TS_H */

```

## 7.5.2 修改硬件驱动源码 s3c2440\_ts.c

将下载的 S3C2410 的驱动程序代码复制到 drivers/input/touchscreen/ 下，改名为 s3c2440\_ts.c。因为 S3C2410 和 S3C2440 的触摸屏接口基本相同，所以不用对代码做太大修改，只要改变一下名字即可。

## 1. 修改触摸屏的私有硬件结构体

这个结构体定义了S3C2440触摸屏的私有数据结构，包括坐标值和腰板计数器等信息，修改后的内容如下：

```
static char *s3c2440ts_name = "s3c2440 TouchScreen";
struct s3c2440ts {
    struct input_dev *dev;      //输入设备指针
    long xp;                  //保存X坐标
    long yp;                  //保存Y坐标
    int count;                //采样计数器
    int shift;                //要采样的次数
    char phys[32];
};
static struct s3c2440ts ts; //定义S3C2440触摸屏
```

## 2. 初始化触摸屏并注册该触摸屏输入设备

注册一个输入设备之前要先对这个输入设备的结构体进行初始化，触摸屏是一种事件输入设备，这里把它同时初始化为一个同步事件、按键事件、绝对坐标事件，具体代码如下：

```
memset(&ts, 0, sizeof(struct s3c2440ts));
ts.dev = input_allocate_device();           //分配一个输入设备到触摸屏结构中
...
ts.dev->evbit[0] = ts.dev->evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) | BIT(EV_ABS); /*同步事件、按键事件、绝对坐标事件*/
ts.dev->keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH); //一个按键
input_set_abs_params(ts.dev, ABS_X, 0, 0x3FF, 0, 0); //绝对坐标的X方向
input_set_abs_params(ts.dev, ABS_Y, 0, 0x3FF, 0, 0); //绝对坐标的Y方向
input_set_abs_params(ts.dev, ABS_PRESSURE, 0, 1, 0, 0); //按下还是抬起
sprintf(ts.phys, "ts0"); // dev文件夹中该触摸屏的名字
ts.dev->private = &ts;
ts.dev->name = s3c2440ts_name; //输入系统的名字
ts.dev->phys = ts.phys;
ts.dev->id.bustype = BUS_RS232;
ts.dev->id.vendor = 0xDEAD; //生产商代号
ts.dev->id.product = 0xBEEF;
ts.dev->id.version = S3C2440TSVERSION;
ts.shift = info->oversampling_shift;
...
printf(KERN_INFO "%s successfully loaded\n", s3c2440ts_name);
/* All went ok, so register to the input system */
input_register_device(ts.dev); //注册S3C2440ts输入设备
```

## 3. 注册S3C2440ts触摸屏驱动

下面是注册S3C2440ts触摸屏驱动的代码，系统初始化硬件设备时会调用s3c2440ts\_init()函数向系统注册这个驱动。

```
static struct device_driver s3c2440ts_driver = {
    .name        = "s3c2440-ts",
```

```

        .bus      = &platform_bus_type,
        .probe     = s3c2440ts_probe,
        .remove    = s3c2440ts_remove,
};

int __init s3c2440ts_init(void)
{
    return driver_register(&s3c2440ts_driver);
}
void __exit s3c2440ts_exit(void)
{
    driver_unregister(&s3c2440ts_driver);
}

```

### 7.5.3 修改 Kconfig 和 Makefile

要使驱动程序可以在内核的配置界面中显示并进行配置和编译，需要修改 drivers/input/touchscreen/Kconfig 和 drivers/input/touchscreen/Makefile。

#### 1. 修改drivers/input/touchscreen/Kconfig

在 drivers/input/touchscreen/Kconfig 文件中添加如下内容，这样在 make menuconfig 时就会在配置菜单中显示出 Samsung S3C2440 touchscreen input driver 这一项，选中它就可以编译驱动源码了。

```

config TOUCHSCREEN_S3C2440
tristate "Samsung S3C2440 touchscreen input driver"
depends on ARCH_S3C2440 && INPUT && INPUT_TOUCHSCREEN
select SERIO
help
    Say Y here if you have the s3c2440 touchscreen.
    If unsure, say N.
    To compile this driver as a module, choose M here: the
    module will be called s3c2440_ts.
config TOUCHSCREEN_S3C2440_DEBUG
boolean "Samsung S3C2440 touchscreen debug messages"
depends on TOUCHSCREEN_S3C2440
help
    Select this if you want debug messages

```

#### 2. 修改drivers/input/touchscreen/Makefile

在 drivers/input/touchscreen/Makefile 中加入要编译进内核的驱动源码，这样就可以把上面的触摸屏驱动编译进内核中了，修改后结果如下，黑体为加入的内容。

```

#
# Makefile for the touchscreen drivers.
#
# Each configuration option enables a list of files.

obj-$ (CONFIG_TOUCHSCREEN_ADS7846)      += ads7846.o
obj-$ (CONFIG_TOUCHSCREEN_BITSY)          += h3600_ts_input.o
obj-$ (CONFIG_TOUCHSCREEN_CORGI)          += corgi_ts.o
obj-$ (CONFIG_TOUCHSCREEN_GUNZE)          += gunze.o

```

```

obj-$ (CONFIG_TOUCHSCREEN_ELO)      += elo.o
obj-$ (CONFIG_TOUCHSCREEN_FUJITSU)   += fujitsu_ts.o
obj-$ (CONFIG_TOUCHSCREEN_MTTOUCH)   += mtouch.o
obj-$ (CONFIG_TOUCHSCREEN_MK712)     += mk712.o
obj-$ (CONFIG_TOUCHSCREEN_HP600)     += hp680_ts_input.o
obj-$ (CONFIG_TOUCHSCREEN_HP7XX)     += jornada720_ts.o
obj-$ (CONFIG_TOUCHSCREEN_USB_COMPOSITE)  += usbtouchscreen.o
obj-$ (CONFIG_TOUCHSCREEN_PENMOUNT)   += penmount.o
obj-$ (CONFIG_TOUCHSCREEN_TOUCHRIGHT) += touchright.o
obj-$ (CONFIG_TOUCHSCREEN_TOUCHWIN)   += touchwin.o
obj-$ (CONFIG_TOUCHSCREEN_UCB1400)    += ucb1400_ts.o
obj-$ (CONFIG_S3C2410_TOUCHSCREEN)    += s3c2440_ts.o

```

## 7.5.4 配置编译内核

接下来就可以进入内核配置界面把驱动编译入内核中了。

### 1. 把触摸屏驱动编译进内核

因为本章中在写触摸屏驱动程序时，是把触摸屏当成一种输入设备来设计的，所以在这里要进入 Device Drivers->Input device support->Touchscreen 选项中，然后选上 Samsung S3C2440 touchscreen input driver 和 Samsung S3C2440 touchscreen debug messages 这两个选项，选前一个选项的目的是为了把触摸屏驱动源码选择编译进内核中，选后一个选项的目的是为了在标准输出上输出驱动代码中的调试信息，选择的结果如图 7.4 所示。

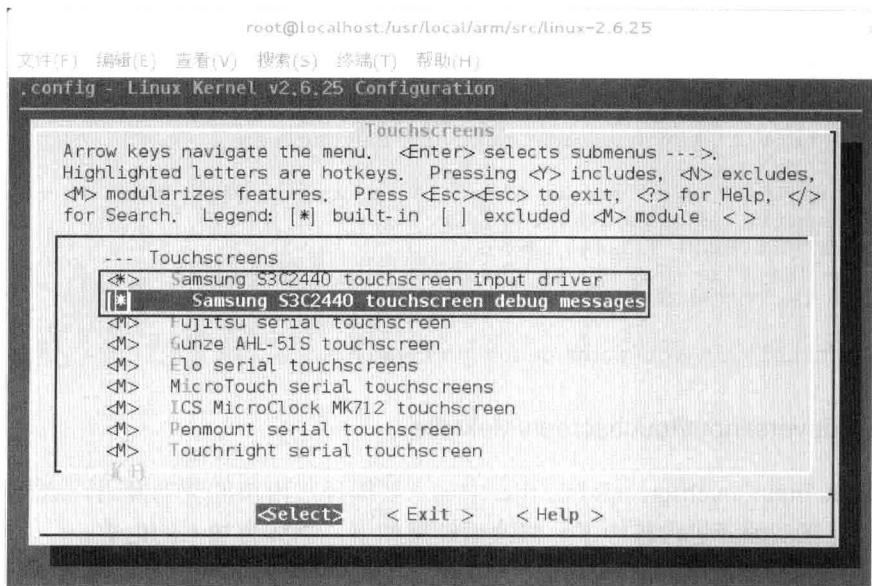


图 7.4 把触摸屏驱动编译进内核

### 2. 选择Event debugging进行内核调试

选择 Event debugging 进行内核调试，当调试没错误后可以取消该选项，如图 7.5 所示。

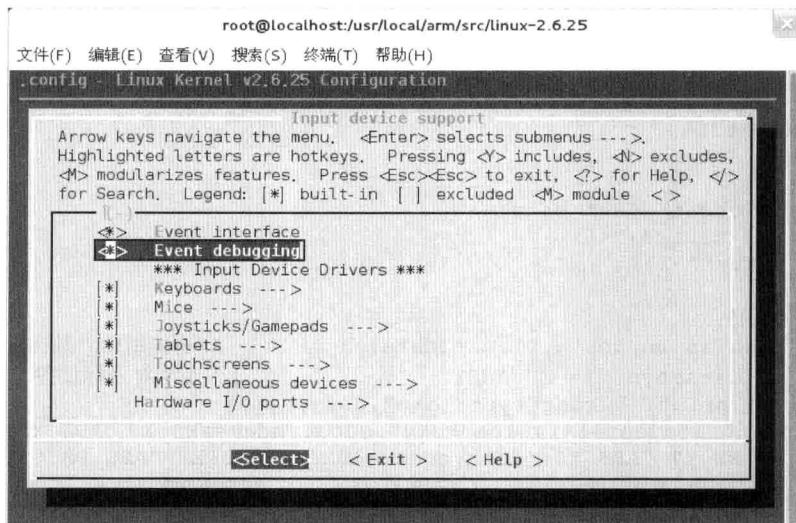


图 7.5 把触摸屏驱动编译进内核

### 7.5.5 触摸屏测试程序设计

对于输入事件接口的触摸屏设备，它使用的是输入设备的标准接口。这种接口传递的数据结构是 `struct input_event`，它的定义在 `include/linux/input.h` 中。

```
struct input_event {
    struct timeval time;
    unsigned short type;
    unsigned short code;
    unsigned int value;
};
```

其中，字段 `time` 是时间戳，利用结构体 `timeval` 记录了事件发生的时间；`type` 字段表明了事件的类型，对于触摸屏设备来说，在驱动程序中定义成了绝对输入设备（EV——ABS）；`code` 字段返回的是事件代码，在触摸屏设备中，只定义了 `ABS_PRESSURE`、`ABS_X`、`ABS_Y`；`value` 字段表示返回值，以下的代码可用于测试触摸屏驱动程序。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/ioctl.h>
#include <pthread.h>
#include <fcntl.h>
#include <linux/input.h>

#define TS_DEV "/dev/inut/event0"
static int ts_fd=-1;
static int init_device(void)
{
    if((ts_fd=open(TS_DEV,O_RDONLY))<0){ //打开触摸屏设备
        printf("open error:%s\n", TS_DEV);
        return -1;
    }
}
```