

深入浅出

【博客藏经阁丛书】

嵌入式底层软件开发

杨铸 唐攀 编著



北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS



策划编辑：胡晓柏

封面设计：runsign 视觉设计



嵌入式底层软件开发

内容简介

本书包含ARM裸机程序开发、嵌入式Linux系统建构、Linux驱动程序开发三部分。从软硬件的分界面开始，循序渐进，逐一详细讲解嵌入式底层软件开发的各个技术要点，技术体系全面；既有一定的理论，又强调实战性；深入浅出，能让读者以最少的时间成本代价获得嵌入式底层软件开发的技术精髓。

读者对象

本书适合硬件工程师、软件工程师、嵌入式软件的从业人员、教授嵌入式软件开发课程的老师、意欲从事嵌入式软件开发工作的大学生阅读。

上架建议：单片机/嵌入式系统

ISBN 978-7-5124-0382-6



9 787512 403826 >

定价：79.00（含光盘）

【博客藏经阁丛书】

深入浅出

嵌入式底层软件开发

杨铸 唐攀 编著



北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

内 容 简 介

本书包含 ARM 裸机程序开发、嵌入式 Linux 系统建构、Linux 驱动程序开发三部分。从软硬件的分界面开始,循序渐进,逐一详细介绍嵌入式底层软件开发的各个技术要点,技术体系全面;既有一定的理论,但更加强调实战性;深入浅出,能让读者以最少的时间成本代价获得嵌入式底层软件开发的技术精髓。

本书适合硬件工程师、软件工程师、嵌入式软件的从业人员、教授嵌入式软件开发课程的老师、意欲从事嵌入式软件开发工作的大学生阅读。

图书在版编目(CIP)数据

深入浅出嵌入式底层软件开发 / 杨铸等编著. —北京:
北京航空航天大学出版社, 2011. 5
ISBN 978 - 7 - 5124 - 0382 - 6

I. ①深… II. ①杨… III. ①微处理器—系统开发
IV. ①TP332

中国版本图书馆 CIP 数据核字(2011)第 044741 号

版权所有,侵权必究。

深入浅出嵌入式底层软件开发

杨 铸 唐 攀 编著

责任编辑 刘 晨

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191) <http://www.buaapress.com.cn>

发行部电话:(010)82317024 传真:(010)82328026

读者信箱:emsbook@gmail.com 邮购电话:(010)82316936

北京时代华都印刷有限公司印装 各地书店经销

*

开本:787×960 1/16 印张:42 字数:941 千字

2011 年 5 月第 1 版 2011 年 5 月第 1 次印刷 印数:4 000 册

ISBN 978 - 7 - 5124 - 0382 - 6 定价:79.00 元(含光盘 1 张)

前言

创作动机

还在学生时代,就曾听一位老师感叹:学硬件的人搞不懂软件,学软件的人搞不懂硬件。似乎计算机软硬件之间有一道难以逾越的鸿沟。因此学计算机软件专业的我当时就有一种冲动,要在裸机设备上,做一些有意思的编程,从而让自己能够从整体上,自下而上地了解、进而贯通计算机的软硬件体系知识。然而 20 世纪 90 年代初期国内硬件的缺乏,软件技术资料的匮乏,让这样的想法举步维艰,几经尝试后,终至放弃。

随着微软 VB、VC 可视化 IDE 开发在国内的兴起, JAVA 开发的兴盛,国内的计算机软件教育越来越倾向于快速拖拽控件的应用程序开发,这使得上下贯通的梦想愈来愈遥不可及。正当梦想远去之时,国内嵌入式产业开始兴起,由于嵌入式本身的特性,使得必然要同时横跨软件和硬件,更为重要的是,ARM CPU 以它的开放性和易学性一统嵌入式硬件的江湖,国内基于 ARM CPU 的优秀嵌入式硬件设备层出不穷,而互联网在国内的普及更使得各种嵌入式技术资料的获得和技术知识的交流变得非常地容易,于是终于再次有机会了却学生时代的梦想。

埋头钻研多年后,方有所心得,在其间更为 Linux 的人人为我、我为人人的理念所打动,故而萌发了要把自己的心得落于纸上,以降低嵌入式软件开发的学习门槛、平滑其陡峭的学习曲线,让更多的人受惠。然及至动笔,方才发现自己会和让别人会完全是两码事情,要想做到深入浅出,让读者以最少的时间成本代价获得嵌入式底层软件开发的技术精髓,何其难哉!本书的创作算是一种尽力的尝试吧。

本书内容及组织方式

第 1 篇(第 1~3 章)以 ARM CPU 及其汇编语言为背景,深入浅出地讲解软件是如何控制硬件的。

- 第 1 章学习 ARM 的汇编指令、伪操作和开发环境,使读者能在短时间内掌握和使用 ARM 汇编语言进行编程。

并为本书制作了大部分的插图,对他踏实刻苦的钻研精神和认真负责的敬业精神,在此表示深深的谢意。

感谢我的父母,是你们从小对我朴实无华的谆谆教导,在我心灵的深处种下了要勤奋学习、要努力工作、要懂得感恩的火种,你们给了我强大的精神鼓励和支持,这才使得本书得以顺利完成。

感谢来自中国台湾的中原大学生物医学工程系蔡育秀教授和全美教育的田本和先生,在我学习和研究的过程中,给予了很大帮助。

感谢北京航空航天大学出版社胡晓柏主任对本书的支持和关怀,正是他耐心的鼓励和支持,才使得本书在最短的时间内与读者见面。

感谢安博中程的孙夏玉、李奎、成宝宗、柳斌、刘鹏、张云和、关东升、关杰、葛红艳、肖瑶,重庆东方的马伯骊、马林,安博教育的马锋,北京软件出口中心的王柱经理、刘志强先生,威盛(中国)的修宸,神州数码的杨建光,对本书的写作和出版提供的帮助。

感谢姚文凯、韩林利对本书提出的宝贵意见,为读者提供了更前沿、更注重实践的案例。

感谢广州友善之臂科技有限公司,他们出品的开发板和相关资料质量很高,使得本书的写作有了个很好的硬件平台,事半功倍。

另外,由于版权问题,书中涉及的部分软件和资料不能收录在随书光盘中,请读者上网搜索自行下载,也可访问作者博客:<http://user.qzone.qq.com/308337370/>; <http://scyangzhu.wordpress.com/>,作者提供相关内容的下载链接。

限于笔者水平有限,书中难免有遗漏和不足之处,恳请广大读者批评指正,联系 E-mail 是:scyz@263.net(杨铸);tangpan09@gmail.com(唐攀),并开通了 QQ 技术讨论群:47753328。

作者 2011 年于
北京维亚大厦
成都少城公园
重庆西永微电园
洛阳师范学院
山西大学商务学院

资源分享网
PDG

目 录

第 1 篇 ARM 体系结构与编程

| | |
|------------------------------------|----|
| 第 1 章 ARM 汇编编程基础 | 3 |
| 1.1 ARM CPU 寄存器 | 3 |
| 1.1.1 普通寄存器 R0~R15 | 4 |
| 1.1.2 状态寄存器 CPSR 与 SPSR | 5 |
| 1.1.3 流水线对 PC 的值的影响 | 6 |
| 1.2 基本寻址方式与基本指令 | 9 |
| 1.2.1 最常见寻址方式精解 | 9 |
| 1.2.2 最常见指令精解 | 10 |
| 1.3 ARM 汇编伪操作 | 12 |
| 1.3.1 汇编伪操作在汇编程序中的使用范例 | 12 |
| 1.3.2 最常见汇编伪操作精解 | 14 |
| 1.3.3 汇编伪操作列表 | 16 |
| 1.4 ADS 开发环境的使用 | 19 |
| 1.4.1 在 ADS 中进行裸机程序的编辑、编译、运行 | 19 |
| 1.4.2 在 AXD 中进行裸机程序调试的方法与步骤 | 27 |
| 1.5 RealView MDK 开发环境的使用 | 34 |
| 1.5.1 在 MDK 开发环境下编写裸机程序 | 35 |
| 1.5.2 MDK 调试裸机程序的方法与步骤 | 49 |
| 1.6 其他常见寻址模式与常见指令 | 52 |
| 1.6.1 其他常见寻址模式 | 52 |
| 1.6.2 其他常见指令 | 57 |

| | |
|-----------------------------------|-----|
| 第 2 章 ARM 编程进阶 | 60 |
| 2.1 ARM 汇编伪指令 | 60 |
| 2.1.1 精解 ldr 伪指令 | 60 |
| 2.1.2 精解 adr | 63 |
| 2.1.3 精解 adrl 伪指令 | 64 |
| 2.1.4 nop 伪指令 | 64 |
| 2.2 ATPCS 与混合编程 | 65 |
| 2.2.1 ATPCS 规则精解 | 65 |
| 2.2.2 精解 C 和 ARM 汇编程序间的相互调用 | 69 |
| 2.3 裸机硬件的控制方法与例程 | 72 |
| 2.3.1 建立真实硬件的开发和调试环境 | 73 |
| 2.3.2 软件控制(驱动)硬件的编程原理 | 85 |
| 2.3.3 裸机硬件控制程序实例 | 86 |
| 2.3.4 启动例程 | 90 |
| 2.4 看门狗定时器 | 97 |
| 2.4.1 看门狗定时器的用途 | 97 |
| 2.4.2 看门狗工作原理 | 98 |
| 2.4.3 看门狗实验 | 101 |
| 2.5 系统时钟 | 102 |
| 2.5.1 系统工作时钟频率 | 103 |
| 2.5.2 时钟驱动实验 | 108 |
| 2.6 SDRAM 内存 | 112 |
| 2.6.1 S3C2440 存储器地址段(Bank) | 114 |
| 2.6.2 SDRAM 内存工作原理 | 116 |
| 2.6.3 SDRAM 的读操作 | 120 |
| 2.6.4 SDRAM 预充电操作 | 122 |
| 2.6.5 SDRAM 突发操作 | 122 |
| 2.6.6 SDRAM 写操作 | 123 |
| 2.6.7 SDRAM 的刷新 | 123 |
| 2.6.8 内存驱动实验 | 130 |
| 2.7 UART 串口 | 134 |
| 2.7.1 同步通信和异步通信 | 134 |
| 2.7.2 数据的串行和并行通信方式 | 135 |

| | | |
|--------------|---------------------------------|------------|
| 2.7.3 | 数据通信传输模式 | 136 |
| 2.7.4 | S3C2440 UART 控制器 | 137 |
| 2.7.5 | S3C2440 UART 串口工作原理 | 138 |
| 2.7.6 | UART 串口驱动实验 | 148 |
| 第 3 章 | ARM 体系结构 | 152 |
| 3.1 | ARM 处理器工作模式 | 152 |
| 3.1.1 | ARM 处理器不同模式下的寄存器 | 153 |
| 3.1.2 | ARM 处理器模式切换(含 MRS、MSR 指令) | 154 |
| 3.2 | ARM 处理器异常处理 | 157 |
| 3.2.1 | 异常分类 | 157 |
| 3.2.2 | 异常发生时的硬件操作 | 158 |
| 3.2.3 | 异常返回地址 | 158 |
| 3.2.4 | 异常向量表 | 160 |
| 3.2.5 | 异常处理的返回 | 162 |
| 3.3 | S3C2440 系统中断 | 163 |
| 3.3.1 | 中断的产生-中断源 | 164 |
| 3.3.2 | 中断优先级 | 167 |
| 3.3.3 | 中断控制器相关寄存器 | 168 |
| 3.3.4 | 系统中断流程 | 172 |
| 3.3.5 | 按键控制 LED 灯实验 | 176 |
| 3.4 | semihosting 与硬件重定向 | 184 |
| 3.4.1 | semihosting 半主机调试 | 185 |
| 3.4.2 | 硬件重定向 | 191 |
| 3.5 | 系统调用与软件中断 SWI 的实现 | 197 |
| 3.5.1 | 系统调用 | 197 |
| 3.5.2 | 软件中断 | 198 |
| 3.5.3 | 软件中断处理 | 200 |
| 3.5.4 | LED 系统调用实验 | 201 |
| 3.6 | 进程切换的实现 | 208 |
| 3.6.1 | 进 程 | 208 |
| 3.6.2 | 进程控制块 PCB | 209 |
| 3.6.3 | 进程创建 | 210 |
| 3.6.4 | 进程队列 | 210 |

| | | |
|-------|-------------------------------|-----|
| 3.6.5 | 进程调度 | 211 |
| 3.6.6 | 上下文切换 | 215 |
| 3.7 | MMU 与内存保护的实现 | 218 |
| 3.7.1 | 存储管理单元 MMU | 219 |
| 3.7.2 | cache | 233 |
| 3.7.3 | CP15 协处理器 | 236 |
| 3.8 | 实战:小型多任务操作系统 miniOS 的实现 | 240 |
| 3.8.1 | miniOS 代码分析 | 241 |
| 3.8.2 | miniOS 应用程序接口 | 279 |
| 3.8.3 | miniOS 应用程序系统调用接口 | 280 |

第 2 篇 嵌入式 Linux 系统建构

| | | |
|-------|--|-----|
| 第 4 章 | 嵌入式 Linux 软件开发环境搭建 | 284 |
| 4.1 | 体验嵌入式 Linux 系统 | 284 |
| 4.2 | Linux 操作系统安装 | 286 |
| 4.2.1 | 在 Windows 上安装虚拟机 | 286 |
| 4.2.2 | 在虚拟机上安装 Linux 操作系统 ubuntu 9.10 | 297 |
| 4.3 | 在 ubuntu 9.10 中安装基本的开发环境 | 299 |
| 4.4 | ubuntu 9.10 上网络服务的安装与配置 | 302 |
| 4.4.1 | 设置 vmware 网络 | 302 |
| 4.4.2 | 安装、配置和使用 FTP 服务 | 306 |
| 4.4.3 | 安装、配置 NFS 服务 | 307 |
| 第 5 章 | 建构 BootLoader | 308 |
| 5.1 | 准备工作 | 308 |
| 5.1.1 | 嵌入式 Linux 系统概述 | 308 |
| 5.1.2 | 构建交叉编译工具链 | 309 |
| 5.1.3 | BootLoader 概述 | 310 |
| 5.2 | 深入剖析 u-boot 代码 | 315 |
| 5.2.1 | 安装和使用源代码阅读工具 Source Insight | 316 |
| 5.2.2 | u-boot 的编译初步 | 318 |
| 5.2.3 | 分析 u-boot 的第一阶段代码(cpu/arm920t/start.S) | 320 |
| 5.2.4 | 分析 u-boot 的第二阶段代码 | 324 |

| | | |
|--------------|--|------------|
| 5.2.5 | 继续移植、编译 u-boot | 326 |
| 5.2.6 | u-boot 常用命令使用简介 | 331 |
| 5.2.7 | u-boot 命令实现框架的分析 | 334 |
| 5.2.8 | u-boot 引导 Linux 操作系统的过程分析 | 340 |
| 5.2.9 | 让 u-boot 支持从 USB slave 接口获得数据 | 346 |
| 第 6 章 | 建构嵌入式 Linux 内核 | 348 |
| 6.1 | Linux 内核简介 | 348 |
| 6.1.1 | Linux 内核版本历史 | 348 |
| 6.1.2 | 内核源码目录结构 | 349 |
| 6.1.3 | Linux 内核构造系统简介 | 349 |
| 6.2 | 移植、裁减及配置 Linux 内核到 S3C2440 开发板 | 351 |
| 6.2.1 | 体验 Linux 内核配置、编译与使用 | 351 |
| 6.2.2 | 为 S3C2440 移植内核 | 354 |
| 6.2.3 | 配置并裁减内核 | 355 |
| 6.2.4 | 运行内核并验证内核被配置的功能 | 359 |
| 6.3 | 内核 Kconfig 与 Makefile 文件分析 | 361 |
| 6.3.1 | 内核构造系统简介 | 361 |
| 6.3.2 | Kconfig 文件精解 | 361 |
| 6.3.3 | .config 文件说明 | 363 |
| 6.3.4 | Makefile 文件精解 | 364 |
| 6.3.5 | 实战:修改 Kconfig 和 Makefile,完成向内核中添加新的 功能组件——网卡、声卡、LCD、触摸屏驱动 | 365 |
| 第 7 章 | 建构嵌入式 Linux 文件系统 375 | |
| 7.1 | 嵌入式 Linux 文件系统简介 | 375 |
| 7.1.1 | 嵌入式文件系统概述 | 375 |
| 7.1.2 | MTD 设备与 Flash 文件系统简介 | 376 |
| 7.1.3 | 嵌入式 Linux 系统中的 tmpfs 文件系统 | 378 |
| 7.2 | 详解制作根文件系统 | 379 |
| 7.2.1 | FHS 标准介绍 | 379 |
| 7.2.2 | 编译/安装 busybox,生成/bin、/sbin、/usr/bin、/usr/sbin 目录 | 380 |
| 7.2.3 | 利用交叉编译工具链,构建/lib 目录 | 382 |
| 7.2.4 | 手工构建/etc 目录 | 384 |

| | | |
|-------|--|-----|
| 7.2.5 | 手工构建最简化的/dev 目录 | 386 |
| 7.2.6 | 使用启动脚本完成/proc、/sys、/dev、/tmp、/var 等目录的完整构建 | 387 |
| 7.2.7 | 制作根文件系统的 jffs2 映像文件 | 392 |
| 7.3 | 建构嵌入式 Linux 应用程序系统 | 393 |
| 7.3.1 | 辅助处理工具的移植 | 393 |
| 7.3.2 | MP3 播放器 madplay 的移植 | 397 |
| 7.3.3 | 主要网络服务器的移植与使用 | 401 |
| 7.3.4 | 数据库程序的移植与使用 | 407 |
| 7.4 | 建构 GUI 系统 | 410 |
| 7.4.1 | 移植 tslib 库 | 410 |
| 7.4.2 | 移植 qtopia | 412 |

第 3 篇 Linux 驱动程序开发

| | | |
|-------|---------------------------|-----|
| 第 8 章 | Linux 驱动程序开发基础 | 416 |
| 8.1 | Linux 设备驱动程序简介 | 416 |
| 8.1.1 | 设备驱动分类和内核模块 | 417 |
| 8.1.2 | 设备文件和设备驱动 | 418 |
| 8.1.3 | 内核模块的编译和使用 | 419 |
| 8.2 | 字符设备驱动基本编程 | 427 |
| 8.2.1 | 字符设备驱动体验 | 428 |
| 8.2.2 | 实现字符设备驱动的工作 | 428 |
| 8.3 | 驱动程序中的并发控制方法 | 443 |
| 8.3.1 | 并发控制原理简介 | 443 |
| 8.3.2 | 信号量的编程实战 | 444 |
| 8.3.3 | 自旋锁的编程实战 | 446 |
| 8.3.4 | Linux 内核提供的其他并发控制方法 | 451 |
| 8.4 | 驱动程序中的阻塞与非阻塞编程 | 453 |
| 8.4.1 | 体验阻塞 I/O | 453 |
| 8.4.2 | 如何在驱动程序中实现阻塞 I/O | 454 |
| 8.4.3 | 体验非阻塞 I/O | 457 |
| 8.4.4 | 如何在驱动程序中实现非阻塞 I/O | 460 |
| 8.5 | 字符设备驱动程序对一些高级特性的实现 | 460 |
| 8.5.1 | non - seekable 的实现 | 460 |

| | |
|---|------------|
| 8.5.2 select 的实现 | 462 |
| 第 9 章 Linux 字符设备驱动开发实战 | 467 |
| 9.1 I/O 内存与硬件通信 | 467 |
| 9.1.1 驱动中的内存分配 | 467 |
| 9.1.2 使用 I/O 端口地址空间与硬件进行通信的内核 API 介绍 | 468 |
| 9.1.3 使用 I/O 内存地址空间与硬件进行通信的内核 API 介绍 | 470 |
| 9.1.4 通过 I/O 内存驱动硬件的实战——LED 灯驱动 | 471 |
| 9.1.5 驱动程序对 ioctl 的规范实现 | 480 |
| 9.2 内核 misc 设备架构分析 | 483 |
| 9.2.1 定义全局变量 | 483 |
| 9.2.2 注册主设备号为 10 的 misc 设备 | 484 |
| 9.2.3 导出内核 API——misc_register 函数 | 485 |
| 9.2.4 实施“乾坤大挪移”的 misc 设备 open 函数 | 486 |
| 9.2.5 导出内核 API——misc_deregister 函数 | 489 |
| 9.3 Watchdog 驱动 | 489 |
| 9.3.1 相关概念 | 489 |
| 9.3.2 Watchdog 硬件结构分析 | 492 |
| 9.3.3 Watchdog 驱动的初始化和卸载 | 492 |
| 9.3.4 探测函数 watchdog_probe 的实现 | 493 |
| 9.3.5 实现 misc 设备中对设备文件的操作 | 496 |
| 9.3.6 Watchdog 平台驱动的设备移除、挂起和恢复接口函数的实现 | 499 |
| 9.3.7 测试 Watchdog 驱动 | 501 |
| 9.4 内核编码规范与风格 | 502 |
| 9.4.1 缩进、长行、{} 与空格的使用规范 | 502 |
| 9.4.2 变量和函数 | 503 |
| 9.4.3 注释、macros 和 enums | 505 |
| 9.4.4 快乐使用内核提供的实现常用功能的宏 | 505 |
| 第 10 章 Linux 驱动中的中断编程 | 507 |
| 10.1 驱动程序调测方法与技巧 | 507 |
| 10.1.1 利用 printk | 507 |
| 10.1.2 详解 OOP 消息 | 509 |
| 10.1.3 利用 strace | 513 |

| | | |
|--------|--|-----|
| 10.1.4 | 利用内核内置的 hacking 选项 | 515 |
| 10.1.5 | 其他调测方法简介 | 518 |
| 10.2 | 驱动程序中的中断处理 | 519 |
| 10.2.1 | 中断简述 | 519 |
| 10.2.2 | 驱动程序中进行中断处理涉及的最基本的内核 API | 520 |
| 10.2.3 | 驱动程序进行中断处理的实例代码分析 | 520 |
| 10.2.4 | 其他关于中断的内核 API | 523 |
| 10.3 | 内核时间与内核定时器 | 523 |
| 10.3.1 | 内核中如何记录时间 | 523 |
| 10.3.2 | 内核定时器 API | 524 |
| 10.3.3 | 内核定时器与内核时间的应用案例——按键消抖 | 525 |
| 10.3.4 | 如何在内核中实现延时 | 528 |
| 10.4 | 中断顶半部与底半部 | 529 |
| 10.4.1 | 区分和使用中断顶半部与底半部的原因 | 529 |
| 10.4.2 | tasklet 机制与编程实例 | 531 |
| 10.4.3 | workqueue 机制与编程实例 | 534 |
| 10.4.4 | tasklet 与 workqueue 的区别和不同应用环境总结 | 543 |
| 10.5 | Linux 中断处理系统的架构与共享中断 | 544 |
| 10.5.1 | 裸机程序中的中断编程与有操作系统下的中断编程的区别 | 544 |
| 10.5.2 | Linux 中断处理系统的架构 | 544 |
| 10.5.3 | 关于共享中断的说明 | 545 |
| 10.5.4 | 共享中断实例 | 545 |

第 11 章 Linux 网络设备驱动开发实战 547

| | | |
|--------|--------------------------------|-----|
| 11.1 | 网络设备驱动基础 | 547 |
| 11.1.1 | 体验网卡驱动 | 547 |
| 11.1.2 | 网卡驱动的基本知识——2 个结构体和 5 个函数 | 548 |
| 11.1.3 | 虚拟网卡 snull 驱动代码分析 | 554 |
| 11.1.4 | 网卡驱动的编写主要内容总结 | 559 |
| 11.2 | 网络设备驱动实例——cs8900 | 559 |
| 11.2.1 | 虚拟网卡驱动与真实网卡驱动的主要区别 | 559 |
| 11.2.2 | 真实网卡驱动的整体框架分析 | 560 |
| 11.2.3 | 驱动中关于 cs8900 硬件操作的探讨 | 569 |

| | |
|--|------------|
| 第 12 章 其他重要设备驱动开发实战 | 582 |
| 12.1 块设备驱动初步(以 ramdisk 为例) | 582 |
| 12.1.1 体验块设备驱动 | 582 |
| 12.1.2 块设备驱动框架介绍 | 583 |
| 12.1.3 块设备的简单读写实现代码分析 | 586 |
| 12.1.4 块设备的高效读写实现代码分析 | 587 |
| 12.1.5 块设备的其他操作接口 fops | 590 |
| 12.2 LCD 驱动 | 593 |
| 12.2.1 LCD 裸机驱动 | 593 |
| 12.2.2 帧缓冲设备驱动框架结构 | 602 |
| 12.2.3 LCD 驱动实例代码 | 609 |
| 12.2.4 LCD 驱动代码的主干结构的总结 | 624 |
| 12.2.5 测试 LCD 驱动程序 | 624 |
| 12.3 触摸屏驱动 | 625 |
| 12.3.1 触摸屏裸机驱动 | 625 |
| 12.3.2 Linux 输入子系统 | 630 |
| 12.3.3 Linux 下触摸屏驱动的实现步骤 | 631 |
| 12.3.4 测试触摸屏驱动程序 | 641 |
| 12.4 USB 驱动初步 | 641 |
| 12.4.1 Linux 下 4 种 USB 驱动简介与功能体验 | 641 |
| 12.4.2 USB 接口与规范 | 643 |
| 12.4.3 USB 设备驱动基本知识 | 645 |
| 12.4.4 USB 设备驱动实例 | 648 |
| 参考文献 | 655 |

第 1 篇

ARM 体系结构与编程

ARM 是 Advanced RISC Machines 的缩写,它是一家微处理器行业的知名企业,于 20 世纪 90 年代初期,据说成立于英国的某个谷仓中,似乎是与后来叱咤风云的 Linux 差不多同时出现。

该企业设计了大量高性能、廉价、耗能低的 RISC(精简指令集)处理器。ARM 公司的特点是只设计芯片,而不生产。它将技术授权给世界上许多著名的半导体、软件和 OEM 厂商(例如 Freescale、Intel、Samsung、TI 等),并提供服务。其隐藏在众多知名企业的背后,故不为大多数人所知,但却是靠收取这些大公司的费用过得很滋润,在这一点上,它很像通讯领域的高通公司。

ARM 体系结构从最初开发到现在有了巨大的改进,并仍在完善和发展。为了清楚地表达每个 ARM 应用实例所使用的指令集,ARM 公司定义了 7 种主要的 ARM 指令集体系结构版本,以版本号 V1~V7 表示。目前正在使用的版本是 V4、V5 和 V6、V7,V4 版本之前的版本已经很少看到。

对于 V4 版而言,不再为了与以前的版本兼容而支持 26 位体系结构,并明确了哪些指令会引起未定义指令异常发生,它相对 V3 版本做了以下的改进:

- 半字加载/存储指令。
- 字节和半字的加载和符号扩展指令。
- 具有可以转换到 Thumb 状态的指令。
- 新增特权处理器模式——系统模式,使用与用户模式相同的一套寄存器。

对于 V5 版而言,在 V4 版本的基础上,对现在指令的定义进行了必要的修正,对 V4 版本的体系结构进行了扩展并增加了指令,具体如下:

- 改进了 ARM/Thumb 状态之间的切换效率。
- 允许非 T 变量和 T 变量一样,使用相同的代码生成技术。
- 增加计数前导零指令和软件断点指令。
- 对乘法指令如何设置标志做了严格的定义。

对于 V6 版而言,最大的改变在于:

增加了 SIMD(Single Instruction Multiple Data)功能扩展;提供高性能的同时降低了功

耗;为音频、视频处理提供优化功能,使其性能提高4倍;首先在ARM11中使用。

V7体系定义了如下独立的内核型:

- A型应用于复杂、基于虚拟内存的操作系统和用户应用软件。
- R型应用于实时操作系统
- M型为微控制器和低成本的应用优化。

包括NEON技术扩展。增加了DSP和媒体处理功能,达到400帧/s,并提供了改进的浮点支持,适用于下一代的3D图形和游戏,同样改进传统的嵌入式控制应用。

指令集版本的升级,主要是软件的概念。而在硬件方面,就像Intel不断推出了286、386、486、奔腾、P2、P3等CPU一样,ARM也不断升级它的CPU,目前市面上常见到的ARM CPU主要有ARM7、ARM9、ARM10、ARM11系列,以及其变种:SecurCore、Xscale等,目前ARM7之前的CPU已经很少见。

ARM9系列包括ARM9TDMI、ARM920T和带有高速缓存处理器宏单元的ARM940T。除了兼容ARM系列,而且能够更加灵活的设计。ARM9系列主要应用于引擎管理、仪器仪表、安全系统和机顶盒等领域。相对于ARM7而言,具备MMU硬件单元,采用五级流水线。

ARM10系列包括ARM1020E处理器核,其核心在于使用向量浮点(VFP)单元VFP10提供高性能的浮点解决方案,从而极大提高了处理器的整型和浮点运算性能,可以用于视频游戏机和高性能打印机等场合。相对ARM9而言,可选VFP10浮点处理协处理器,包括了DSP指令集,6级流水线。

ARM11则是ARM公司较新的CPU,目前正在流行的基于android的3G手机,很多都采用ARM11系列的CPU,看来ARM11很有可能在不久的将来会横扫中国三网合一的终端设备。因此ARM11系列的CPU和V6版的指令集将是ARM未来重要的发展方向。

Cortex是基于V7架构新的产品系列,目前已经成为ARM公司主推的ARM处理器内核,根据ARM内核的最新描述,从另一个方面将其分为3种类型:

- 经典(Classic)内核。
- 应用(Application)内核。
- 嵌入式(Embedded)内核。

其中经典内核为Cortex以及以前的内核,应用内核和嵌入式内核分别指带有MMU和无MMU的Cortex内核。

SecurCore系列涵盖了SC100、SC110、SC200和SC210处理核。该系列处理器主要针对新兴的安全市场,以一种全新的安全处理器设计为智能卡和其他安全IC开发提供独特的32位系统设计,并具有特定反伪造方法,从而有助于防止对硬件和软件的盗版。

Intel Xscale微控制器则提供全性能、高性价比、低功耗的解决方案,支持16位Thumb指令并集成数字信号处理(DSP)指令。

第一章

ARM 汇编编程基础

1.1 ARM CPU 寄存器

ARM 的汇编编程,本质上就是针对 CPU 寄存器的编程,所以首先要弄清楚 ARM 有哪些寄存器?这些寄存器都是如何使用的?

ARM 寄存器分为两类:普通寄存器和状态寄存器,如表 1-1 所列。

表 1-1 ARM 寄存器

[illegible]

续表 1-1

| 寄存器类别 | 寄存器在汇编中的名称 | 各模式下实际访问的寄存器 | | | | | |
|-------|------------|--------------|----------|----------|----------|----------|----------|
| | | 用户 | 系统 | 管理 | 中止 | 未定义 | 中断 |
| 状态寄存器 | CPSR | CPSR | | | | | |
| | SPSR | 无 | SPSR_abt | SPSR_abt | SPSR_und | SPSR_irq | SPSR_fiq |

请看表 1-1 的第 2 列,普通寄存器总共 16 个,分别为 R0—R15;状态寄存器共两个,分别为 CPSR 和 SPSR。

1.1.1 普通寄存器 R0~R15

普通寄存器中特别要提出来的的是 R13、R14、R15。

R15 别名 PC(Program Counter),中文称为程序计数器,它的值是当前正在执行的指令在内存中的位置(不考虑流水线的影响,参见“流水线对 PC 值的影响”),而当指令执行结束后,CPU 硬件会自动将 PC 值加上一个单位,从而使得 PC 值为下一条即将执行的指令在内存中的位置,这样 CPU 硬件就可以根据 PC 值自动完成取指的操作。正是由于有 PC 的存在,以及 CPU 硬件会自动增加 PC 值,并根据 PC 值完成取指操作,才使得 CPU 一旦上电就永不停歇地运转,由此可见 PC 寄存器对于计算机的重要性。对于汇编程序编写而言,PC 寄存器也是十分重要,因为当程序员通过汇编指令完成了对 PC 寄存器的赋值操作的时候,其实就是完成了一次无条件跳转,这一点非常重要,请务必牢记。

LDR PC, LR

R14 别名 LR(Linked Register),中文称为链接寄存器,与子程序调用密切相关,用于存放子程序的返回地址,是 ARM 程序实现子程序调用的关键所在。下面用 C 语言中对子程序调用的实现细节来说明 LR 是如何被使用的。

```

1 int main(void)
2 {
3     int k, i = 1, j = 2;
4     addsub(i, j);
5     k = 3;
6 }
7 int addsub(int a, int b)
8 {
9     int c;
10    c = a + b;
11    return c;
12 }
```

对于上面的程序,编译器会将第 4 行编译为指令:BL addsub,将第 11 行编译为指令:

MOV pc, lr。(关于 BL 和 MOV 指令详见“基本寻址模式与基本指令”。)

在这里,关键指令 BL addsub 会完成两件事情:

- (1) 将子程序的返回地址(也就是第 5 行代码在内存中的位置)保存到寄存器 LR 中;
- (2) 跳转到子程序 addsub 的第一条指令处。

这样就完成了子程序的调用。

而指令 MOV pc, lr 则将保存在 lr 中的返回地址赋给 pc,这样就完成了从子程序的返回。由此可见,lr 是专门用于存放子程序的返回地址的。

另外一个要引起注意的问题是,如果子程序又调用了孙子程序,那么根据前面的分析,在调用孙子程序时,LR 寄存器中的值将从子程序的返回地址变为孙子程序的返回地址,这将导致从孙子程序返回子程序没有问题,但从子程序返回父程序则会出错。那么这个问题如何解决呢?其实,如果我们编写的是 C 程序,那么我们一点也不用担心,因为编译器会为我们考虑一切,针对这个问题,编译器会在孙子程序的入口处增加入栈操作将 LR 的值入栈,然后在孙子程序的返回处增加出栈操作,将 LR 的值恢复,从而解决这个难题。不过我们一定要保持头脑的清醒,因为我们现在是在编写汇编子程序,此时编译器已经不能在这方面给我们提供保障,所以在编写汇编子程序的时候,发现该子程序还要再调用孙子程序,那么请务必记住,一定要在子程序的入口处保存 LR 寄存器的值。

好了,现在轮到寄存器 R13 了,R13 又名 SP(stack pointer),中文名称栈指针寄存器。顾名思义,它是用于存放堆栈的栈顶地址的。也就是说,每次进行出栈和入栈的时候,都将根据该寄存器的值来决定访问内存的位置(即:出入栈的内存位置),同时在出栈和入栈操作完成后,SP 寄存器的值也应该相应增加或减少。这里要特别说明的是,其实在 32 位的 ARM 指令集中没有专门的入栈指令和出栈指令,所以并不是一定要用 SP 来作为栈指针寄存器,除了 PC 外,任何普通寄存器均可作为栈指针寄存器,只不过约定俗成,都使用 SP 罢了。我们将在“其他寻址模式与其他指令”中见到 ARM 中使用 SP 作为栈指针寄存器来实现出入栈的汇编指令。

寄存器 R0~R12 是普通的数据寄存器,可用于任何地方。在不涉及 ATPCS 规则(在“ATPCS 与混合编程”中详细介绍)的情况下,它们并没有什么特别的用法。

1.1.2 状态寄存器 CPSR 与 SPSR

1. 状态寄存器 CPSR(Current Program Status Register)

中文名称:当前程序状态寄存器,顾名思义它是用于保存程序的当前状态的。那么,程序的哪些状态是需要保存的呢?

图 1-1 是 CPSR 寄存器的内容,主要由以下部分组成:

- (1) 条件代码标志位。它们是 ARM 指令条件执行的依据。

① N:运算结果的最高位反映在该标志位。对于有符号二进制补码,结果为负数时 N=

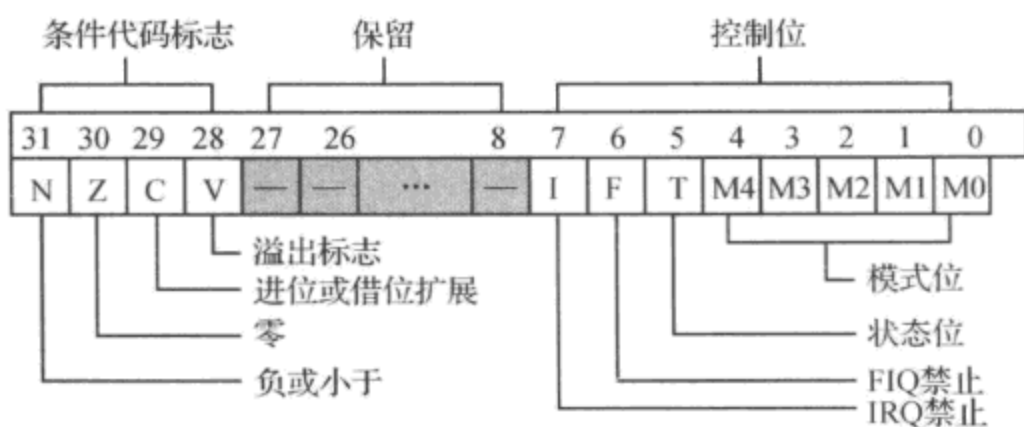


图 1-1 CPSR 寄存器

1, 结果为正数或零时 $N=0$ 。

② Z: 指令结果为 0 时 $Z=1$ (通常表示比较结果“相等”), 否则 $Z=0$;

③ C: 当进行加法运算 (包括 CMN 指令), 并且最高位产生进位时 $C=1$, 否则 $C=0$ 。当进行减法运算 (包括 CMP 指令), 并且最高位产生借位时 $C=0$, 否则 $C=1$ 。对于结合移位操作的非加法/减法指令, C 为从最高位最后移出的值, 其他指令 C 通常不变。

④ V: 当进行加/减法运算, 并且发生有符号溢出时 $V=1$, 否则 $V=0$, 其他指令 V 通常不变。

(2) 控制位。它们将控制 CPU 是否响应中断。

I: 中断禁止位, 当 I 位置位时, IRQ 中断被禁止。

F: 快中断禁止位, 当 F 位置位时, FIQ 中断被禁止。

T: 反映了 CPU 当前的状态。当 T 位置位时, 处理器正在 Thumb 状态下运行; 当 T 位清零时, 处理器正在 ARM 状态下运行。

(3) 模式位。包括 M4、M3、M2、M1 和 M0, 这些位决定了处理器的模式 (关于处理器模式详见“ARM 处理器模式与异常初步”)。

总共有 7 种模式: 用户、快中断、中断、管理、中止、未定义、系统, 分别会用于不同的情况和异常。由此可见, 不是所有模式位的组合都定义了有效的处理器模式, 如果使用了错误的设置, 将引起一个无法恢复的错误。

2. SPSR (Saved Program Status Register)

中文名称: 保存的程序状态寄存器。

该寄存器的结构与 CPSR 完全一样, 在异常发生时 (关于异常, 请参见“ARM 处理器模式与异常初步”), 由硬件自动将异常发生前的 CPSR 的值存放到 SPSR 中, 以便将来在异常处理结束后, 程序能恢复原来 CPSR 的值。

1.1.3 流水线对 PC 的值的影响

从图 1-2 中我们看到 CPU 内部有 3 个主要组成部分: 指令寄存器, 指令译码器, 指令执

行单元(包括 ALU 和通用寄存器组)。

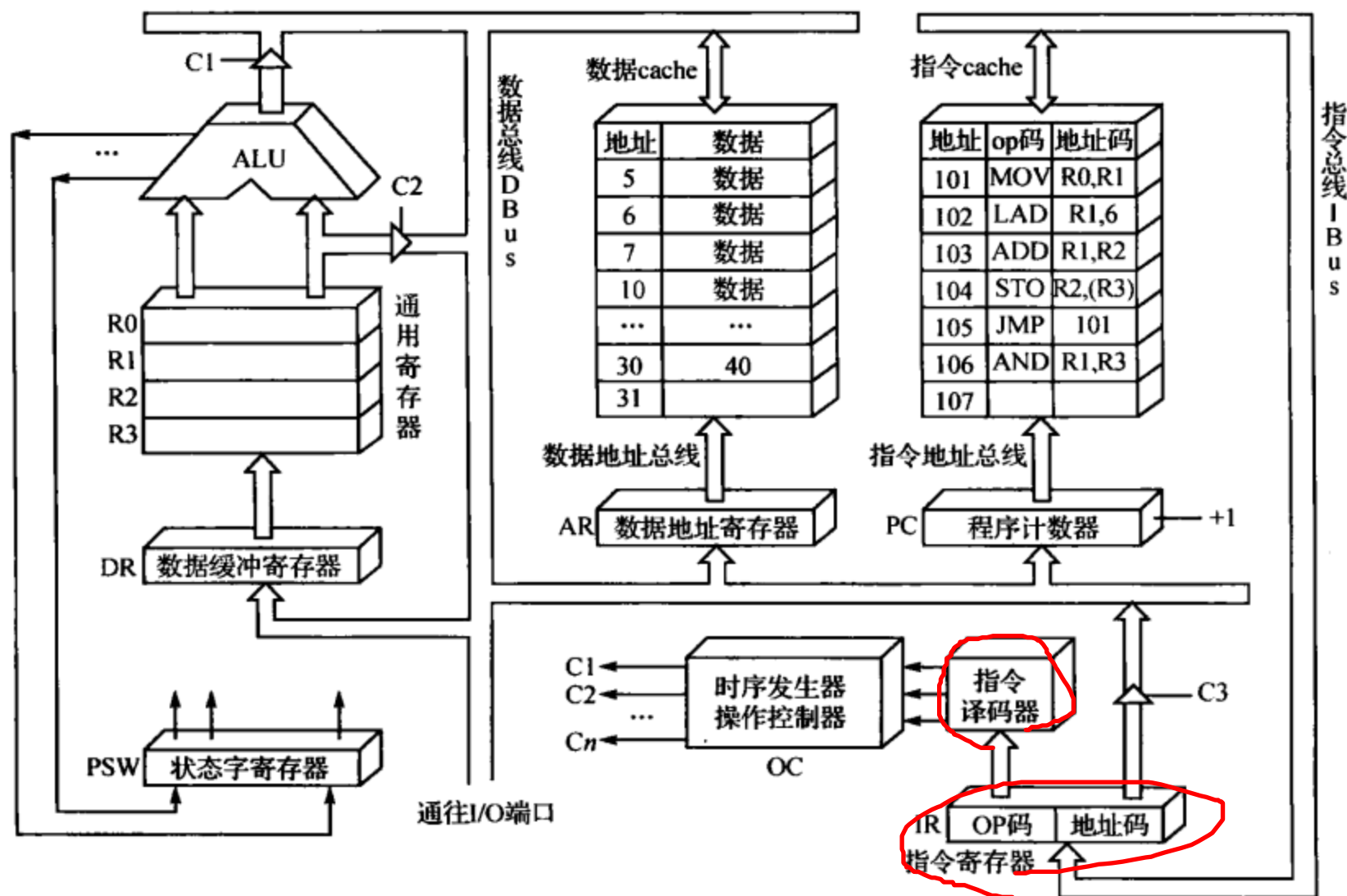


图 1-2 CPU 内部结构框图

CPU 在执行一条指令的时候,主要有 3 个步骤:取指(将指令从内存或指令 cache 中取入指令寄存器);译码(指令译码器对指令寄存器中的指令进行译码操作,从而辨识出该指令是要执行 add,或是 sub,或是其他操作,从而产生各种时序控制信号);执行(指令执行单元根据译码的结果进行运算并保存结果)。

现在我们假设一下:CPU 串行执行程序(即:执行完一条指令后,再执行下一条指令);指令执行的 3 个步骤中每个步骤都耗时 1 s;整个程序共 10 条指令。那么,这个程序总的执行时间是多少呢?显然,是 30 s。但这个结果令我们非常不满意,因为它太慢了。有没有办法让它坐上京津高铁提速 3 倍呢?当然有!仔细观察图 1-2,我们发现:取指阶段占用的 CPU 硬件是指令通路和指令寄存器;译码阶段占用的 CPU 硬件是指令译码器;执行阶段占用的 CPU 硬件是指令执行单元和数据通路。三者占用的 CPU 硬件完全不同,这样就使得如下的操作得以同时进行:在对第一条指令进行译码的时候,可以同时第二条指令进行取指操作;在对第 1 条指令进行执行的时候,可以同时第 2 条指令进行译码操作,对第 3 条指令进行取指操作。显然,这样就可以将该程序的运行总时间从 30 s 缩减为 12 s,提速近 3 倍。上面所述并行

第1章 ARM 汇编编程基础

运行指令的方式称为流水线操作。可见：流水线操作的本质是利用指令运行的不同阶段使用的 CPU 硬件互不相同，并发的运行多条指令，从而提高时间效率，如图 1-3 所示。

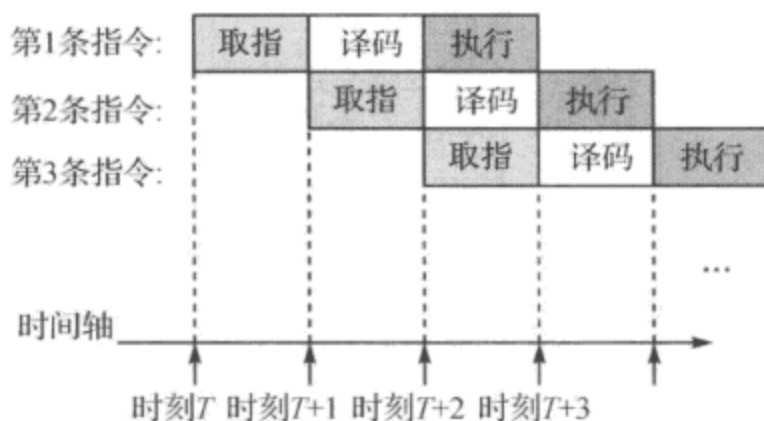


图 1-3 流水线指令执行图

流水线的引入的确提高了 CPU 运行指令的时间效率，但却为汇编程序编写引入了新的问题。请看下面的分析：

寄存器 PC 的值是即将被取指的指令的地址，正常情况下，在该条指令被取入 CPU 后执行期间，PC 的值保持不变，在该条指令执行完成的时间点上，硬件会自动将 PC 的值增加一个单位的大小，这样 PC 就指向了下一条将被取指和执行的指令。而在引入流水线后，PC 值的情况发生了变化，假定第一条指令的内存地址为 X ，则在时刻 T ，PC 值变为 X ，并在时刻 T 至时刻 $T+1$ 期间维持不变；在时刻 $T+1$ ，PC 的值变为 $X+1$ 个单位，并在时刻 $T+1$ 至时刻 $T+2$ 期间维持不变；在时刻 $T+2$ ，PC 的值变为 $X+2$ 个单位，并在时刻 $T+2$ 至时刻 $T+3$ 期间维持不变；在时刻 $T+3$ ，PC 的值将变为 $X+3$ 个单位。由此可见，在第一条指令的执行阶段，PC 的值不再是该指令在内存中的位置，而是该指令在内存中的位置 $+2$ 个单元。对于 ARM 指令集而言，每条指令的长度为 32bit，占 4B，所以一条指令在内存中需要 4B 存储。因此，结论如下：

指令执行时，PC 值 = 当前正在执行指令在内存中的地址 $+8$ 。

请牢记以上结论。虽然目前我们并不明白这个结论有何作用，但在后续的课程中，特别是通过查看反汇编代码的方式理解伪指令和编译器行为的时候，这个结论将会很有帮助。

最后说明一点：其实 ARM 现在的 CPU 的流水线级数早已经突破了 3 级。但我仍然以 3 级流水线来进行讲解，是因为：①较之多级流水线，3 级流水线最简单，因此也最便于初学者理解；②虽然存在多种级别的流水线，但 ARM 出于统一和前后兼容的考虑，PC 的值 = 当前正在执行指令在内存中的地址 $+8$ 这个结论在所有的流水线级别上都是相同的。作为编程人员而言，只需要知道这个结论即可。

1.2 基本寻址方式与基本指令

要想进行 ARM 的汇编编程,首当其冲要知道最基本、最常用的指令,而要了解指令则必须要了解寻址方式。所以这里将聚焦在——基本寻址方式和基本指令。

首先,来看一看我们已经见过的两条指令:“MOV pc, lr”和“BL addsub”。

最简单的汇编指令格式是操作码(例如 MOV、BL)和操作数(例如 pc、lr、addsub)。操作码易于理解,例如 MOV 表示将某个值从一处传送到另一处,BL 表示跳转到某处;而操作数则表示一处和另一处到底是哪里(是在寄存器中还是内存中),要跳转的位置在哪里(是绝对地址或者是相对地址)。

操作数部分要解决的问题是:到哪里去获得操作数?因此就有了寻址方式的分类。基本上来讲,ARM 共有 8 种寻址方式,这里先了解其中最基本的 3 种寻址方式:寄存器寻址、立即数寻址、寄存器间接寻址。

1.2.1 最常见寻址方式精解

1. 寄存器寻址

“MOV pc, lr”表示操作数来源于寄存器(PC 和 LR)。对于这种寻址方式而言,在指令的 32 位机器码中的地址码部分,存放的是寄存器(PC 和 LR)的编号,故称为寄存器寻址。

2. 立即数寻址

“MOV pc, #64”表示将常数 64 放入寄存器 PC,其中常数 64 称为立即数。立即数寻址指令中的地址码部分就是操作数本身,也就是说数据就包含在指令当中取出指令也就取出了可以立即使用的操作数(故称为立即数)。

这里,可能大家会看出一个问题:由于立即数是位于 32 位机器码中的,而 32 位机器码中除了操作数外还有操作码,这就意味着不可能用全部 32bit 来表示立即数。事实上,ARM 机器指令中,仅用了最低的 12bit 来表示立即数,如图 1-4 所示。那么立即数的范围是 $-2048 \sim 2047$,这意味着“MOV pc, #8192”这样的指令是非法的。但事实情况并非如此,“MOV pc, #8192”是合法且能正常运行的。真实情况是,ARM 机器指令可以表示的立即数范围为 $-2^{31} \sim 2^{31} - 1$,只不过它只能表示这其中的 2^{12} 个数字而已。ARM 是这样用 12bit 来表示一个立即数的:将 12bit 划分为两部分——高 4 位和低 8 位,将低 8 位补 0 扩展为 32 位,然后循环右移 X 位($X = \text{高 4 位表示的无符号整数} \times 2$),例如:如果 32 位机器码中低 12bit 为 0x512,则其表示的立即数为 0x04800000。

这里,请大家不妨现在先思考两个问题,我们将在后续章节中予以解答:

(1) 为什么 ARM 要这样设计,而不是按照我们最常见的想法(即:12 位就表示 $-2^{11} \sim$

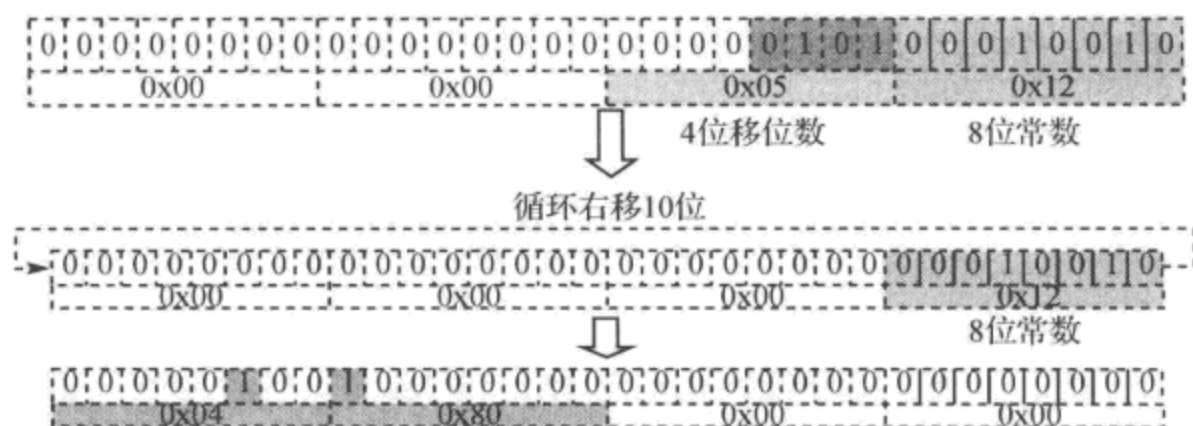


图 1-4 12bit 立即数

$2^{11}-1$ 中的数)。

(2) 如果需要“mov r0, #10000”这样的指令,应该怎么办?(常数 10 000 不能按照如上的方法进行表示。)

3. 寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号,所需的操作数保存在寄存器指定地址的存储单元中,即寄存器中存放的是操作数的地址指针。例如:“LDR R0, [R2]”表示将 R2 中存放的数作为内存地址,到该内存处取出存放的数,放到寄存器 R0 中,如图 1-5、图 1-6 所示。

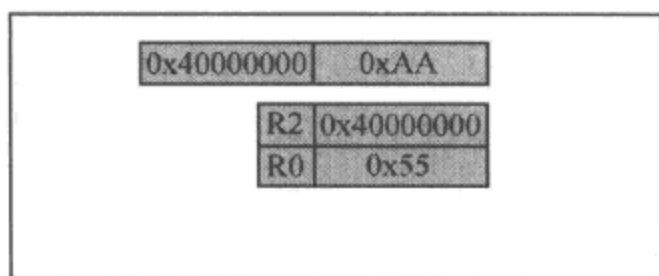


图 1-5 执行 LDR R0, [R2]前的情况

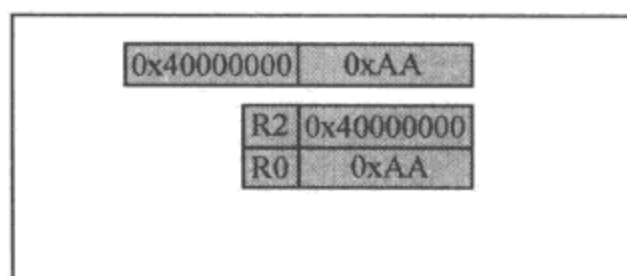


图 1-6 执行 LDR R0, [R2]后的情况

1.2.2 最常见指令精解

了解了基本的寻址方式后,现在来看一看最常用的汇编指令。

1. 单寄存器加载指令

加载字指令:LDR r0, [r1],将内存中的一个字(4 字节)加载到寄存器 r0 中。

加载字节指令:LDRB r0, [r1],将内存中的1字节加载到寄存器 r0 中。

有符号数加载字节指令:LDRSB r0, [r1],这条指令与上一条指令的不同之处在于,由于加载的是 1 字节,而不是一个字,所以需要确定寄存器 r0 的高 24bit 是什么。对于上一条指令,r0 的高 24bit 补 0,而本条指令,r0 的高 24bit 补符号位,也就是补 r0 的 bit7。

2. 单寄存器存储指令

存储字指令:STR r0, [r1],将 r0 中的值存储到内存的 4 字节中。

存储字节指令:STRB r0, [r1],将 r0 的低 8bit 存储到内存的 1 字节中。

3. 分支指令

共 3 条:B、BL、BX。

B label:跳转到标号 label 处,也就是说在该条 b 指令执行后,下一条执行的指令是标号 label 处的指令。

BL label:与 B 指令的功能相同,也实现跳转,不同之处在于,bl 在跳转的同时还要将返回地址(bl 指令的下一条指令的地址)保存到 lr 中。

BX r0:将 r0 的值作为地址,跳转到该地址处,并根据 r0 的值决定是否在 ARM 和 thumb 态之间进行切换。

特别说明:

B 和 BL 指令,其跳转范围限制在当前指令的 $\pm 32\text{MB}$ 地址内(ARM 指令为字对齐,最低 2 位地址固定为 0)。

4. 数据处理指令

MOV r0, r1:将 r1 的值赋给 r0。

ADD(SUB):r0, r1, r2:将 r1 的值加上(减去)r2 的值,结果存放到 r0 中。

AND(ORR, EOR):r0, r1, r2:将 r1 的值与(或、异或)r2 的值,结果存放到 r0 中。

CMP r1, r2:比较 r1 与 r2 值的大小。

特别需要说明的问题:

指令 CMP r1, r2,其运行细节是:执行 r1-r2 的操作,如果结果为负数,则置位 CPSR 的 N 位,清零 Z 位;结果为 0,则清零 CPSR 的 N 位,置位 Z 位;结果为正,则清零 CPSR 的 N 位,清零 Z 位。但 r1-r2 的结果并不保存。CMP 指令通常用于分支跳转。例如,如下的 C 程序:

```
int i, j;
if (i == j) {
    i++;
} else {
    j++;
}
```

如果使用汇编语句改写的话,就应该写为:

```
;使用 ldr 指令将变量 i 的值放入 r0
;使用 ldr 指令将变量 j 的值放入 r1
cmp r0, r1
```

$\pm 2^{26}$

新华书店
PDG

```

addeq r0, r0, #1
;使用 streq 指令将 r0 的值放入变量 i 中
beq label
add r1, r1, #1
;使用 str 指令将 r1 的值放入变量 j 中
label    其他代码
.....

```

其中 addeq, streq, beq 这几条指令,是 add, str, b 指令的条件执行版本。讲到这里就不得不讲解一下什么是条件执行了。ARM 指令集的所有指令均支持条件执行,条件执行指的是,指令可以根据执行时的情况(CPSR 的条件代码标志位)决定自身是否被执行。eq 表示如果 CPSR 的 Z 位为 1(对于本程序,实际上就是 r0 的值与 r1 的值相等,因为 cmp 会根据 r0 与 r1 的值设置 Z 位)的情况下,该指令要执行,否则不执行。

其他条件助记符如表 1-2 所列。

表 1-2 条件助记符

| 条件助记符 | 标志 | 含义 | 条件助记符 | 标志 | 含义 |
|-------|-----|-----------|-------|----------|---------------|
| EQ | Z=1 | 相等 | HI | C=1,Z=0 | 无符号数大于 |
| NE | Z=0 | 不相等 | LS | C=0,Z=1 | 无符号数小于或等于 |
| CS/HS | C=1 | 无符号数大于或等于 | GE | N=V | 有符号数大于或等于 |
| CC/LO | C=0 | 无符号数小于 | LT | N!=V | 有符号数小于 |
| MI | N=1 | 负数 | GT | Z=0,N=V | 有符号数大于 |
| PL | N=0 | 正数或零 | LE | Z=1,N!=V | 有符号数小于或等于 |
| VS | V=1 | 溢出 | AL | 任何 | 无条件执行(指令默认条件) |
| VC | V=0 | 没有溢出 | NV | 任何 | 从不执行(不要使用) |

1.3 ARM 汇编伪操作

1.3.1 汇编伪操作在汇编程序中的使用范例

掌握了基本的 ARM 汇编指令后,要写出简单的 ARM 汇编程序,还必须要掌握基本的 ARM 汇编伪操作(directive)。现在来看一个简单的汇编程序,该程序调用子程序完成了加法操作。

- 1 ;文件名:TEST.S
- 2 ;功能:实现两个寄存器相加

```

3      AREA Example, CODE, READONLY ; 声明代码段 Example
4      ENTRY ; 标识程序入口
5      CODE32 ; 声明 32 位 ARM 指令
6  START  MOV R0, #0 ; 设置参数
7          MOV R1, #10
8          BL  ADD_SUB ; 调用子程序 ADD_SUB
9  LOOP   B LOOP ; 跳转到 LOOP
10 ADD_SUB
11      ADD R0, R0, R1 ; R0 = R0 + R1
12      MOV PC, LR ; 子程序返回
13      END ; 文件结束

```

第 6、7 行将传递给子程序的参数存放在 r0 和 r1 中,第 8 行调用子程序。第 11、12 行是子程序的代码,完成了两个参数相加,并将结果放在 r0 后返回主程序。第 6、9、10 行的 START、LOOP、ADD_SUB 是标号,最经常用于跳转指令 B 和 BL,由于汇编语法要求的缘故,标号必须顶格写(即:不能在行首有空格),否则编译器会报错。与之对应的是,汇编指令一定不能顶格写。

很明显分号(;)在汇编程序中是注释符号,相当于 C 语言的// 号。除此之外,当然大家注意到了第 3、4、5、13 行是我们没学习过的符号,其实它们就是本文的重点——ARM 汇编伪操作。首先先来解释这几个伪操作,第 3 行定义了一个代码段。汇编伪操作 AREA 表示定义一个段,其段名为 Example, CODE 表明是代码段(而不是数据段),属性为只读(READONLY),从而表示第 6~12 行是程序代码(而不是程序数据)。第 4 行的 ENTRY 表示整个程序的入口点(即:程序运行的第一条指令。注 1)是第 6 行的 MOV 指令。第 5 行的 CODE32 表示第 6~12 行的程序代码是 ARM 指令,而不是 thumb 指令。第 13 行的 END 表示源代码文件结束,其背后的含义就是:如果程序员在第 13 行后还写有汇编指令,编译器也根本不会理会这些代码,更不会去编译它们,当然这些代码也就不可能出现在最后的可执行文件中。请务必记住、在 END 伪操作的后面再写代码,那是无用功。根据经验,初学者总是会犯这样的错误。

特别说明:第 9 行的含义是要让程序在运行结束后,在第 9 行进行死循环,从而让整个程序定格在第 9 行。这一点也许让人很困惑:在写应用程序时,程序结束就结束了,源代码根本不需要再去写个死循环。但现在要弄清楚:在写应用程序时,有 OS 为处理程序结束后的若干事情。可是,现在已经得不到 OS 服务。如果不自己写第 9 行的代码,那么当认为程序已经运行结束(第 8 行执行完成)的时候, CPU 不会聪明地停下来,它会继续任劳任怨地去取指第 11 行,继续运行,这不是大家所希望的。其实这还不是最糟糕的,最糟糕的是,如果程序没有 11~13 行,那么 CPU 任劳任怨取出的指令其实是内存中的随机数,但 CPU 却会把它当作指令来执行,那么,此时会出现什么情况呢?

注 1: ENTRY 的本意并非如此,此处的含义仅是 ENTRY 的副作用而以。关于其本意,

后续章节将予以解释。

1.3.2 最常见汇编伪操作精解

当然,伪操作远不止这几条,下面再来介绍经常使用的若干伪操作。

(1) GBLA:定义全局算术变量(准确说,应该是全局符号),例如:GBLA testval。

(2) SETA:对全局算术符号进行赋值,例如:testval SETA 9; testval SETA testval + 1。

(3) DCD:在编译时为整数分配字存储空间,例如:DCD 0x123456ab,这条伪操作将导致编译器在最终的二进制可执行文件中分配一个字的空间,并在该空间中存放整数 0x123456ab。

(4) DCB:在编译时为数据分配字节存储空间,例如:DCB 'a',这条伪操作将导致编译器在最终的二进制可执行文件中分配一个字节的空間,并在该空间中存放字符 a 的 ASCII 码。

(5) IF, ELSE 及 ENDIF:相当于 C 语言的条件编译,例如:

```
GBLA testval
testval SETA 9
IF testval < 5
    mov r0, #testval
ELSE
    mov r1, #testval
ENDIF
IF :DEF:testval
    mov r2, #testval
ELSE
    INFO 4, "you should define testval"
ENDIF
```

编译器编译该段代码的结果是:

```
mov r1, #9
mov r2, #9
```

(6) WHILE 及 WEND:例如

```
GBLA testval
testval SETA 1
WHILE testval <= 3
testval SETA testval + 1
    mov r0, #testval
WEND
```

直到等于4才能停下

编译器编译该段代码的结果是：

```
mov r0, #2
mov r0, #3
mov r0, #4
```

(7) MACRO、MEND 及 MEXIT：相当于 C 语言的宏替换，例如：

```
MACRO
$ label xmac $ p1, $ p2
; code1
$ label.loop1
; code2
    BGE $ label.loop1
$ label.loop2
; code3
    BL $ p1
    BGT $ label.loop2
; code4
    ADR r0, $ p2
; code5
MEND
; 主程序
abc xmac subr1, de
```

编译器编译该段代码的结果是：

```
; code1
abc.loop1
; code2
    BGE abc.loop1
abc.loop2
; code3
    BL subr1
    BGT abc.loop2
; code4
    ADR r0, de
; code5
```

(8) EQU：相当于 C 语言的宏定义，例如：testval EQU 4。

(9) EXPORT：参见“ATPCS 与混合编程”。→

(10) IMPORT：参见“ATPCS 与混合编程”。←

资源分享
PDG

第1章 ARM 汇编编程基础

非常重要的一点是：必须深刻理解汇编伪操作是给编译器提供某些必要的信息，以帮助编译器正确完成程序的编译。当编译完成后，汇编伪操作就完成了它的历史使命，它不可能在最终的可执行程序的二进制代码中留下哪怕是一点点痕迹，当然也就不可能在程序运行时受到CPU的“青睐”。总之记住一句话，汇编伪操作是给编译器看的，而不是给CPU看的。这是汇编伪操作与汇编指令最大的区别。

1.3.3 汇编伪操作列表

为了保持内容的完整，下面给出较为完整的汇编伪操作列表。如需完整的列表，请自行查阅ads自带的“Online Books”相关章节。

(1) 符号定义(Symbol Definition)伪操作如表1-3所列。

表1-3 符号定义伪操作

| 伪操作 | 语法格式 | 作用 |
|-------|-------------------------------|-------------------------------|
| GBLA | GBLA Variable | 声明一个全局的算术变量，并将其初始化成0 |
| GBLL | GBLL Variable | 声明一个全局的逻辑变量，并将其初始化成{FALSE} |
| GBLS | GBLS Variable | 声明一个全局的字符串变量，并将其初始化成空串“” |
| LCLA | LCLA Variable | 声明一个局部的算术变量，并将其初始化成0 |
| LCLL | LCLL Variable | 声明一个局部的逻辑变量，并将其初始化成{FALSE} |
| LCLS | LCLS Variable | 声明一个局部的串变量，并将其初始化成空串“” |
| SETA | Variable SETA expr | 给一个全局或局部算术变量赋值 |
| SETL | Variable SETL expr | 给一个全局或局部逻辑变量赋值 |
| SETS | Variable SETS expr | 给一个全局或局部字符串变量赋值 |
| RLIST | name LIST (list of registers) | 为一个通用寄存器列表定义名称 |
| CN | name CN expr | 为一个协处理器的寄存器定义名称 |
| CP | name CP expr | 为一个协处理器定义名称 |
| DN/SN | name DN/SN expr | DN/SN 为一个双精度/单精度的 VFP 寄存器定义名称 |
| FN | name FN expr | 为一个 FPA 浮点寄存器定义名称 |

(2) 数据定义(Data Definition)伪操作如表1-4所列。

表1-4 数据定义伪操作

| 伪操作 | 语法格式 | 作用 |
|-------|---------------------------|------------------------------|
| LTORG | LTORG | 声明一个数据缓冲池(也称为文字池)的开始 |
| MAP | MAP expr {,base_register} | 定义一个结构化的内存表(Storage Map)的首地址 |

续表 1-4

| 伪操作 | 语法格式 | 作用 |
|------------|--|---|
| FIELD | {label} FIELD expr | 定义一个结构化内存表中的数据域 |
| SPACE | {label} SPACE expr | 分配一块连续内存单元,并用 0 初始化 |
| DCB | {label} DCB expr {, expr} | 分配一段字节内存单元,并用 expr 初始化 |
| DCD/DCDU | {label} DCD {U} expr {, expr}... | 分配一段字(对齐)的内存单元,DCD 可能在分配的第 1 个内存单元前插入填补字节(padding),以保证分配的内存是字对齐的,DCDU 不需要对齐 |
| DCFD/DCFDU | {label} DCFD{U} fpliteral{, fpliteral}... | 为双精度的浮点数分配字对齐的内存单元 |
| DCFS/DCFSU | {label} DCFS{U} fpliteral{, fpliteral}... | 为单精度的浮点数分配字对齐的内存单元 |
| DCI | {label} DCI expr{, expr}... | 在 ARM 代码中分配一段字对齐的内存单元;在 Thumb 代码中,分配一段半字对齐的半字内存单元 |
| DCQ/DCQU | {label} DCQ {U}{-} Literal{, {-} literal}... | 分配一段以双字(8 个字节)为单位的内存 |
| DCW/ DCWU | {label} DCW {U} expr{, expr}... | DCW 用于分配一段半字对齐的半字内存单元 |

(3) 汇编控制(Assembly Control)伪操作如表 1-5 所列。

表 1-5 汇编控制伪操作

| 伪操作 | 语法格式 | 作用 |
|------------------------|---|--|
| IF, ELSE 及 ENDIF | IF logical expr ... {ELSE ...} ENDIF | 能够根据条件把一段源代码包括在汇编语言程序内或者将其排除在程序之外 |
| WHILE 及 WEND | WHILE logical expr ... WEND | 能够根据条件重复汇编相同的一段源代码 |
| MACRO, MEND 及 MEXIT | MACRO { \$label } macroname{ \$ param {, \$ param}...} ...;宏代码 MEND | MACRO 标识宏定义的开始,MEND 标识宏定义的结束。MEXIT 用于从宏中跳转出去。用 MACRO 和 MEND 定义的一段代码,称为宏定义体。通过宏名称来调用宏 |

(4) 信息报告(Reporting)伪操作如表 1-6 所列。

表 1-6 信息报告伪操作

| 伪操作 | 语法格式 | 作用 |
|--------|--------------------------------|--|
| ASSERT | ASSERT logical expr | 对汇编程序的第二遍扫描中,如果其中 ASSERT 中条件不成立,ASSERT 伪操作将报告该错误信息 |
| INFO | INFO numeric-expr, string-expr | 对汇编程序的第一遍扫描或者第二遍扫描时 INFO 伪操作报告诊断信息 |
| OPT | OPT n | 通过 OPT 伪操作可以在源程序中设置列表选项 |
| TTL | TTL title | 在列表文件的每一页的开头插入一个标题 |

(5) 其他(Miscellaneous)伪操作如表 1-7 所列。

表 1-7 其他伪操作

| 伪操作 | 语法格式 | 作用 |
|-----------------------|--|---|
| CODE16 | CODE16 | 告诉汇编编译器后面的指令序列为 16 位的 Thumb 指令 |
| ✓ CODE32 | CODE32 | 告诉汇编编译器后面的指令序列为 32 位的 ARM 指令 |
| ✓ EQU | name EQU expr{, type} | 为数字常量,基于寄存器的值和程序中的标号(基于 PC 的值)定义一个字符名称,类似于 C 语言中的 #define 宏定义 |
| ✓ AREA | AREA sectionment {, attr} {, attr}... | 定义一个代码段或者数据段 |
| ✓ ENTRY | ENTRY | 指定程序的入口点 |
| ✓ END | END | 告诉编译器已经到了源程序结尾 |
| ✓ ALIGN | ALIGN {expr{, offset}} | <u>通过添加补丁字节使当前位置满足一定的对齐方式</u> |
| EXPORT/ ✓ GLOBAL | EXPORT symbol{[WEAK]} | 声明一个符号可以被其他文件引用 |
| ✓ IMPORT/ ✓ EXTERN | IMPORT/ EXTERN symbol{(WEAK)} | 告诉编译器当前的符号不是在本源文件中定义的,而是在其他源文件中定义的,在本源文件中可能引用该符号 |
| ✓ GET/INCLUDE | GET filename | 将一个文件包含到当前源文件中,并将被包含的文件在其当前位置进行汇编处理 |
| INCBIN | INCBIN filename | 将一个文件包含到当前源文件中,被包含的文件不进行汇编处理 |
| KEEP | KEEP{symbol} | 告诉编译器将局部符号包含在目标文件的符号表中 |
| NOFP | NOFP | 禁止源程序中包含浮点运算指令 |
| REQUIRE | REQUIRE lable | 指定段之间的相互依赖关系 |

1.4 ADS 开发环境的使用

掌握了基本的汇编指令和伪操作后,就具备了编写简单 ARM 汇编程序的基本理论能力,不过要实战得到真实可执行的程序,还需要可以对程序进行编辑和编译的开发环境(命令行编译器或 IDE)的支持,同时程序在开发过程中免不了要进行调试,这就需要调试器的支持。一般而言,会有供应商将程序的编辑器、编译器、调试器以及其他一些辅助工具组合在一起,形成程序的开发、调试集成开发环境(IDE)软件,提供给程序开发人员使用。对 ARM 程序开发而言,目前比较流行的 IDE 有两套:运行于 windows 平台的 ADS 和运行于 Linux 平台的 gcc 等交叉编译工具链。总的来看,ADS 在程序的编译和调试方面要比 gcc 使用起来方便很多,也更容易掌握和使用,因此针对初学者,本书现在将展示 ADS 的使用。

在进行下面的学习之前,请自行在 Windows 主机上安装 ARM 公司生产的 ads1.2 这个集成开发环境软件。

1.4.1 在 ADS 中进行裸机程序的编辑、编译、运行

ADS 集成开发环境主要由编辑、编译器 Code Warrior 和调试器 AXD 组成。

首先使用 Code Warrior 对程序进行编辑和编译,过程如下:

1. 建立工程(图 1-7)

- (1) 在磁盘里新建一个目录“D:\arm”。
- (2) 打开 ADS 软件。
- (3) 单击工具栏 File 菜单,在下拉菜单中单击 New 命令。

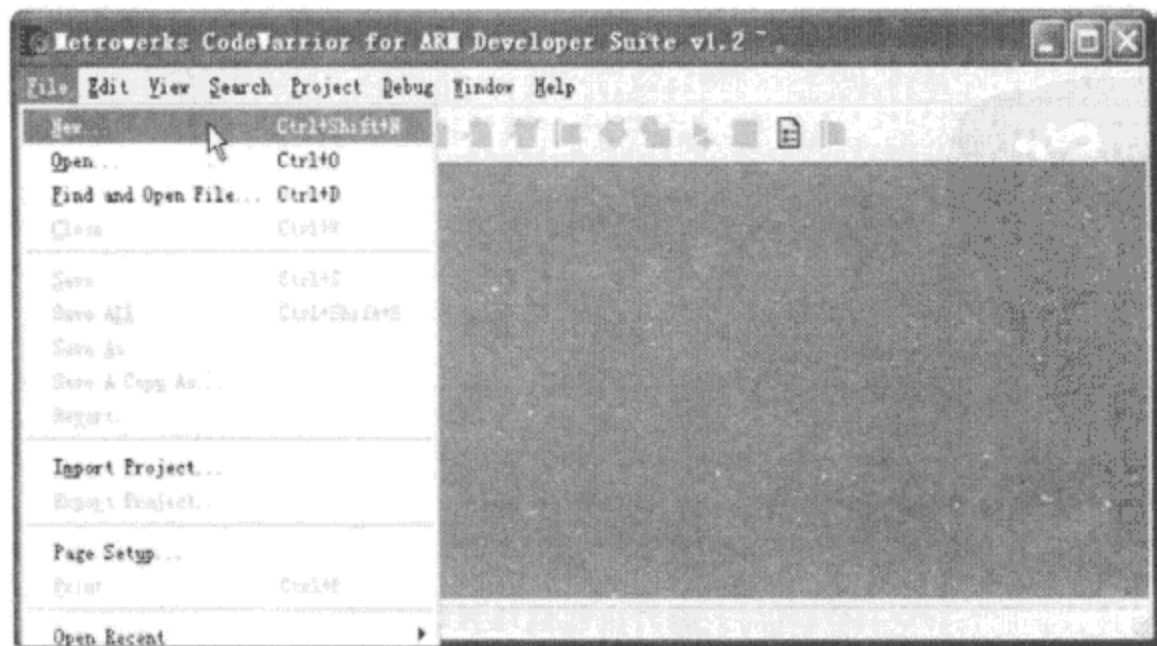


图 1-7 新建工程

2. 选择工程类型(图 1-8)

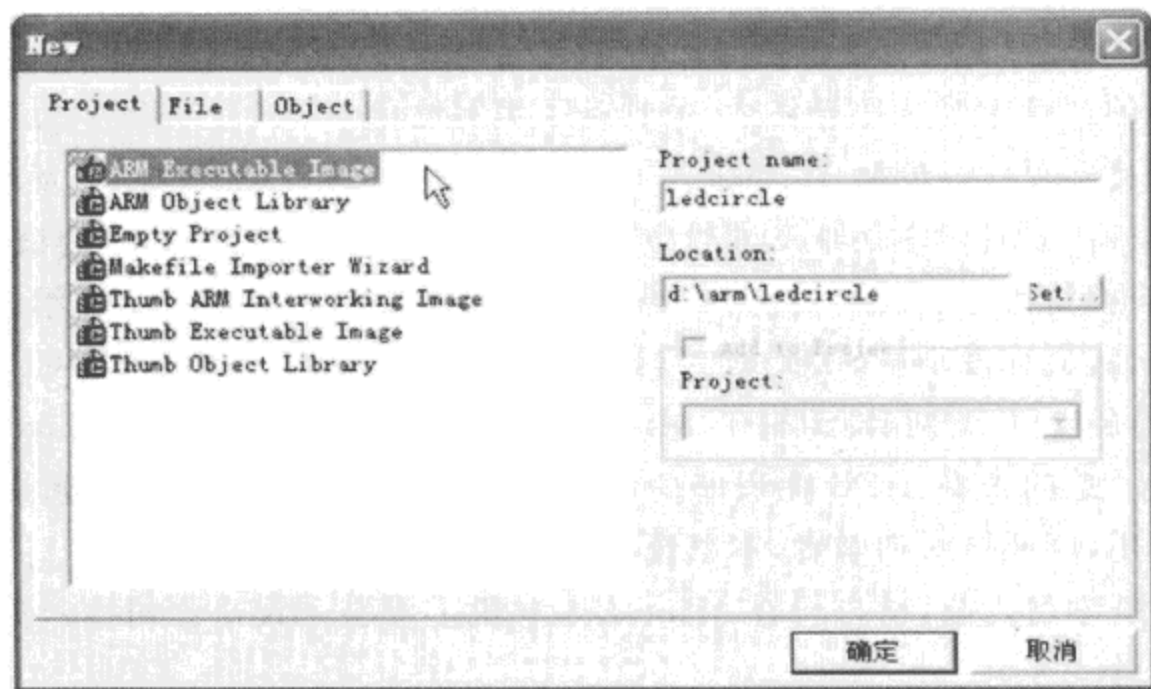


图 1-8 选择工程类型

3. 输入工程名称及其目录(图 1-9)

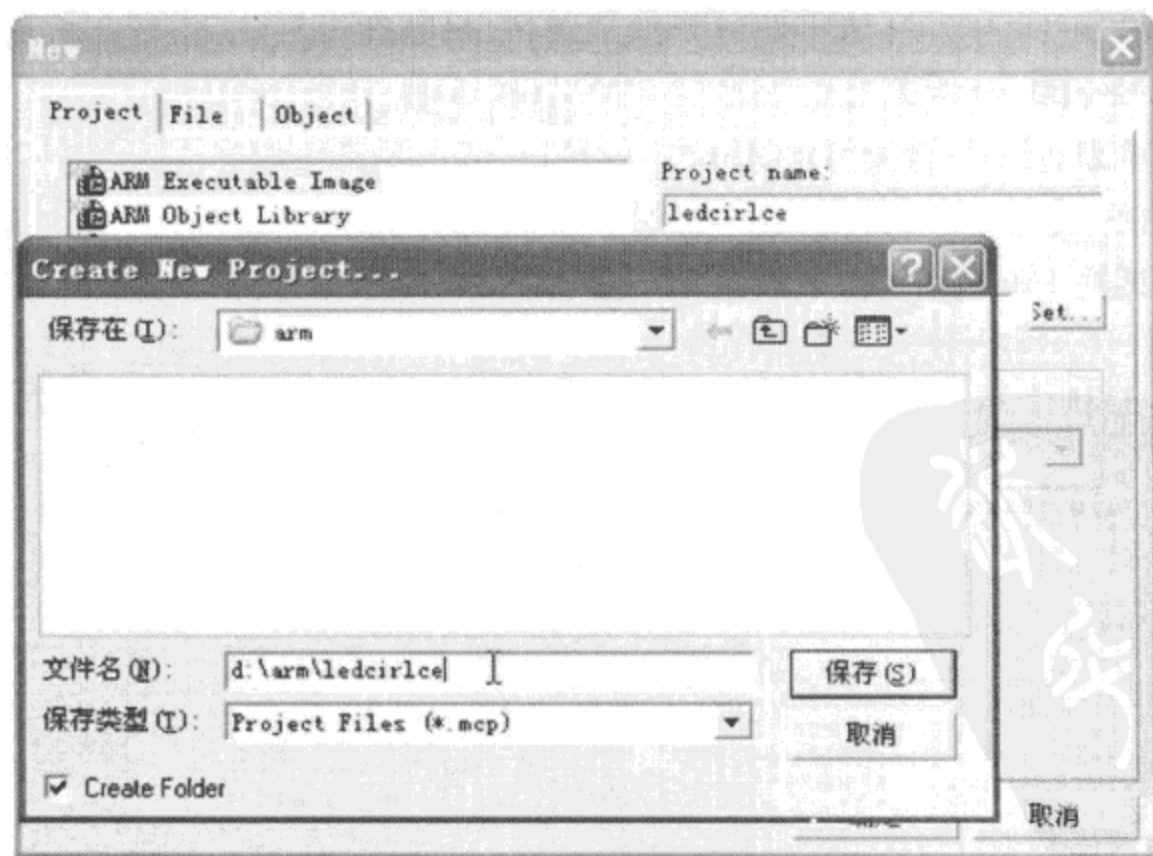


图 1-9 输入工程名称及其目录

4. 工程建立后的情形(图 1-10)

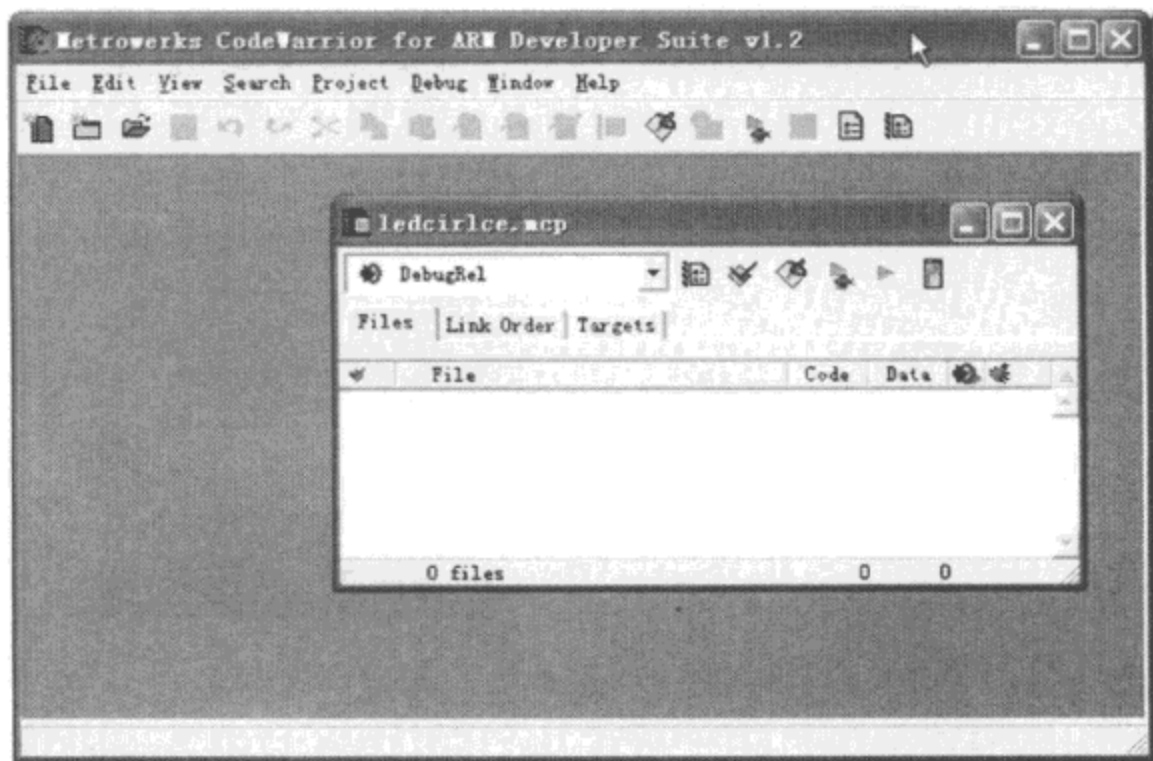


图 1-10 工程建立后的情形

5. 新建工程后的目录(图 1-11)

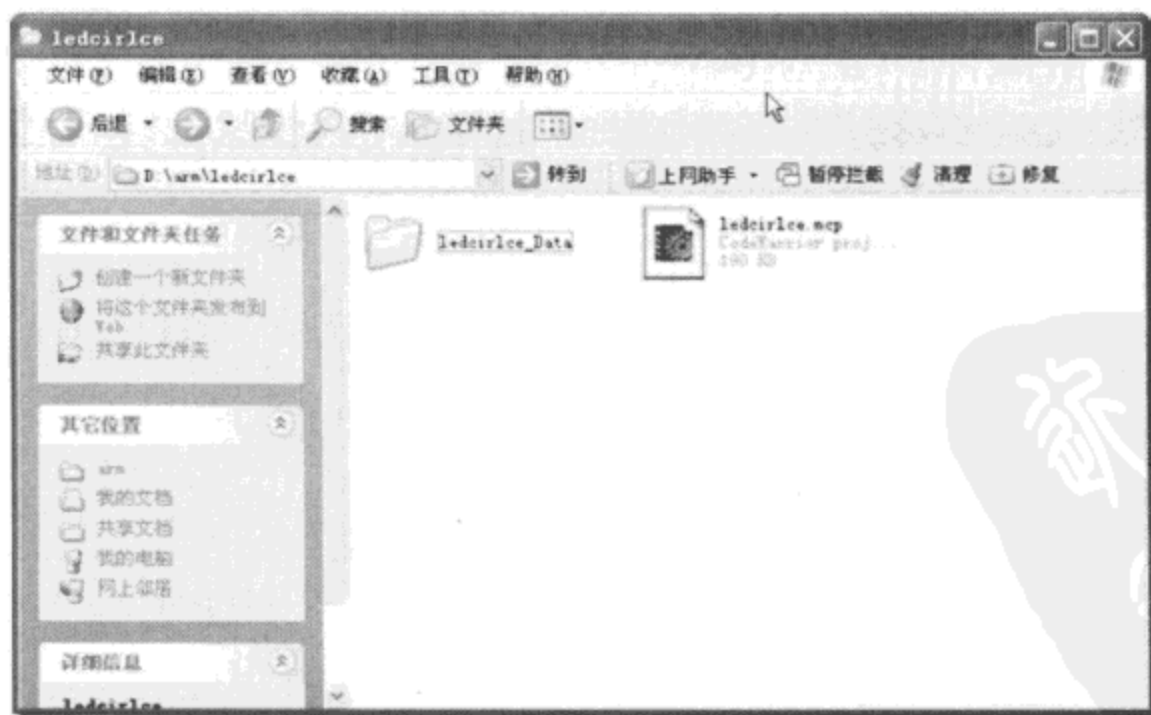


图 1-11 新建工程后的目录

6. 新建源文件并加入工程与 target(图 1-12)



图 1-12 新建源文件并加入工程与 target

7. 编辑汇编和 C 的源代码(图 1-13、图 1-14)

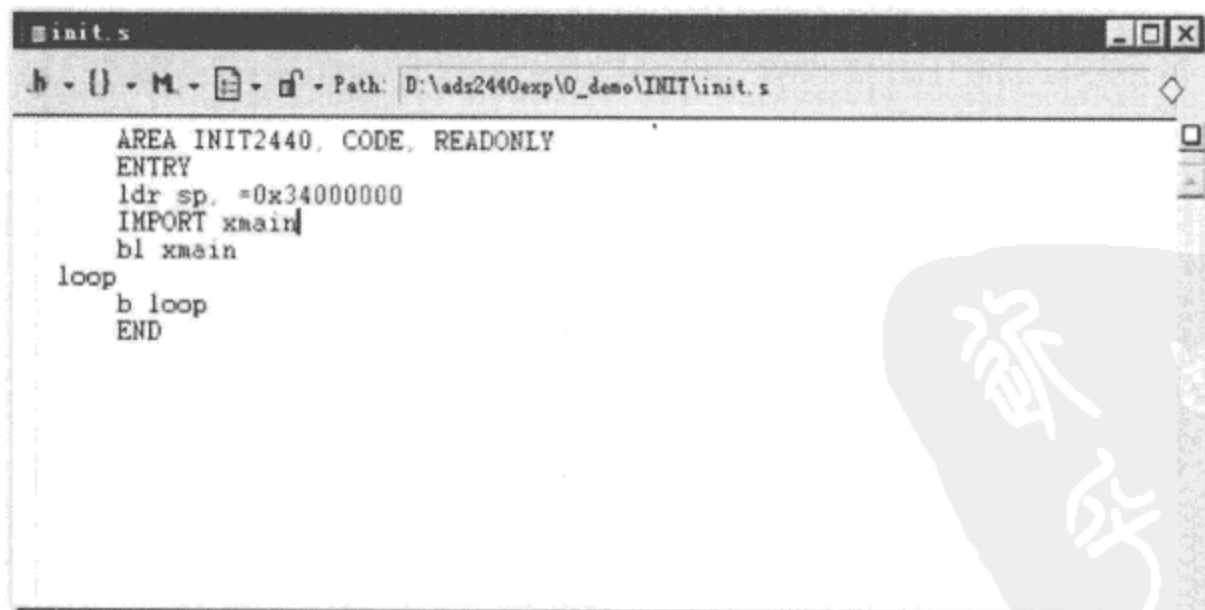


图 1-13 编辑汇编源代码

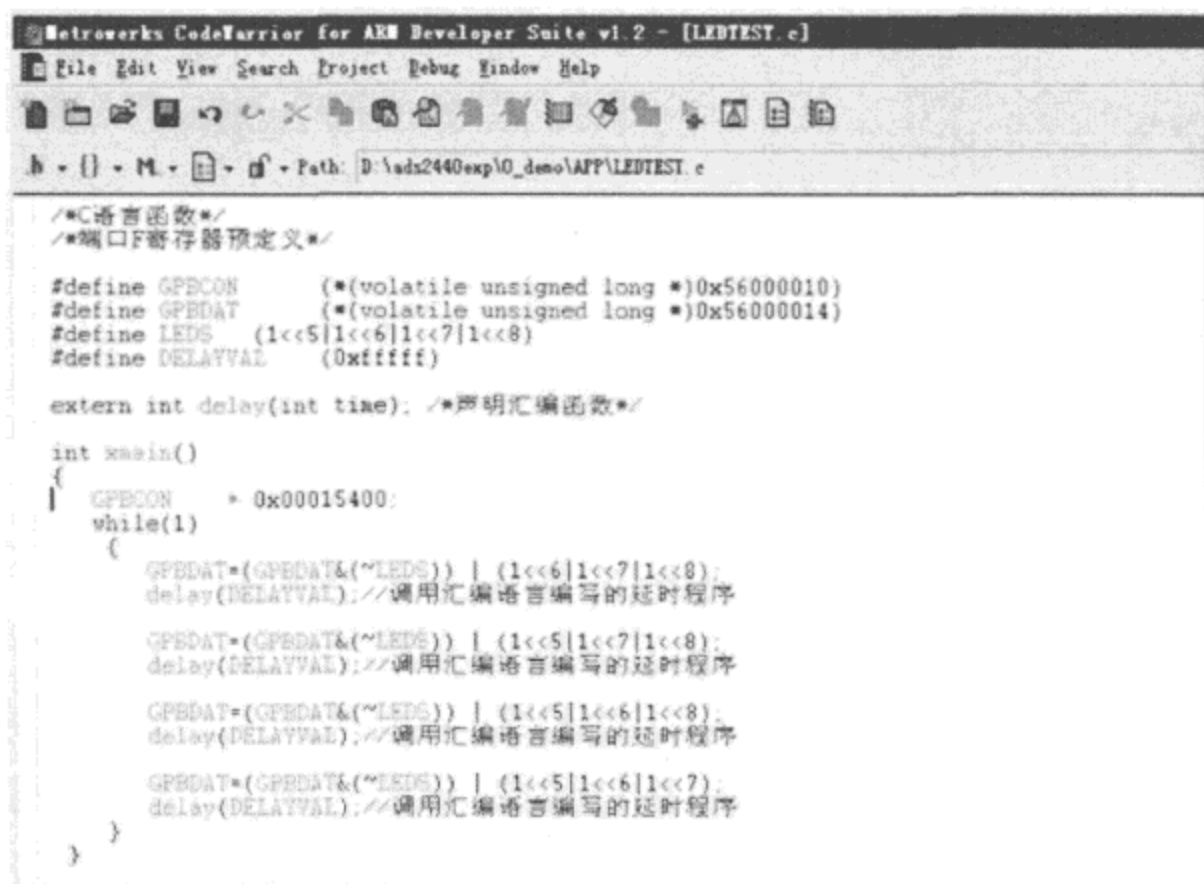


图 1-14 编辑 C 源代码

8. 编译源代码及其结果显示(图 1-15、图 1-16)

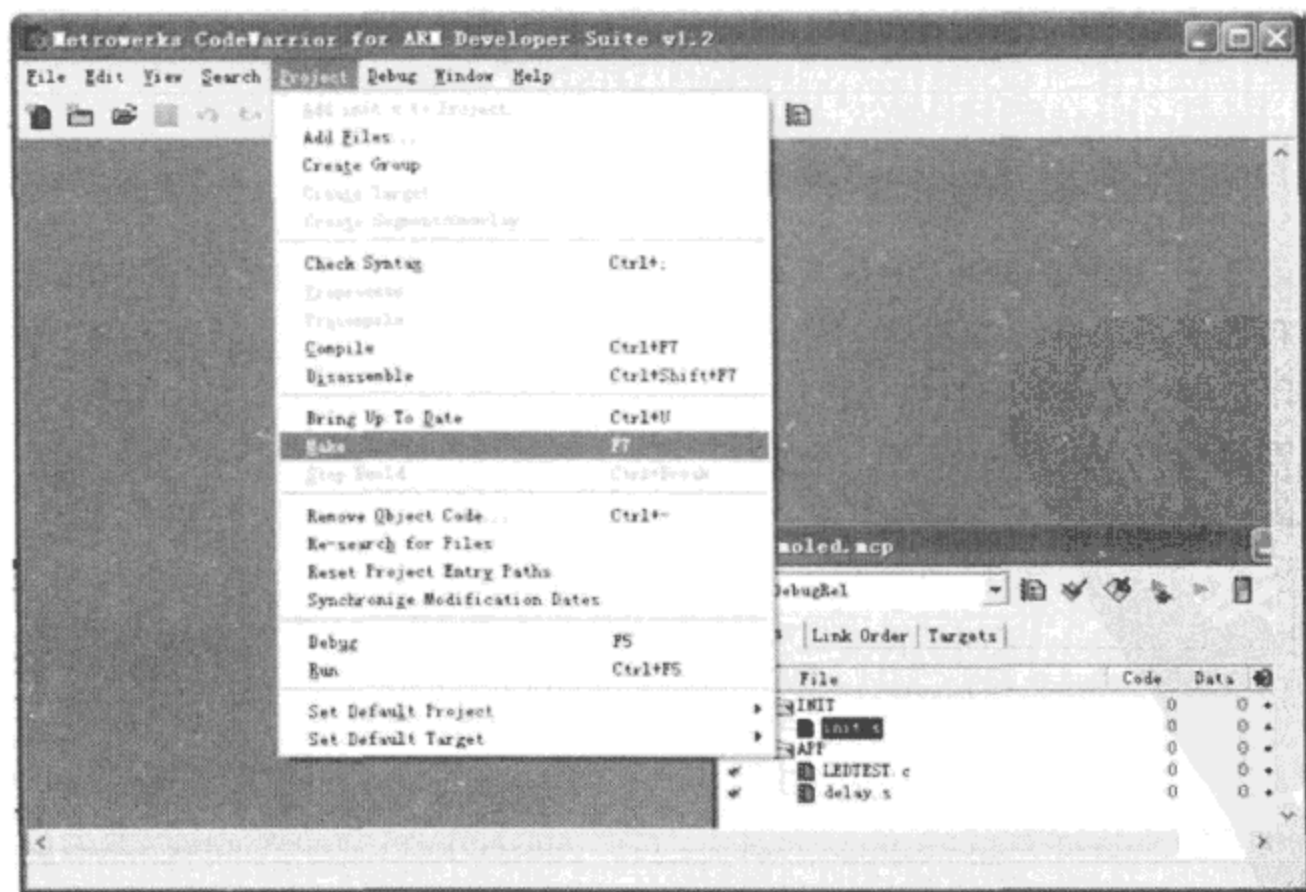


图 1-15 编译(make)菜单

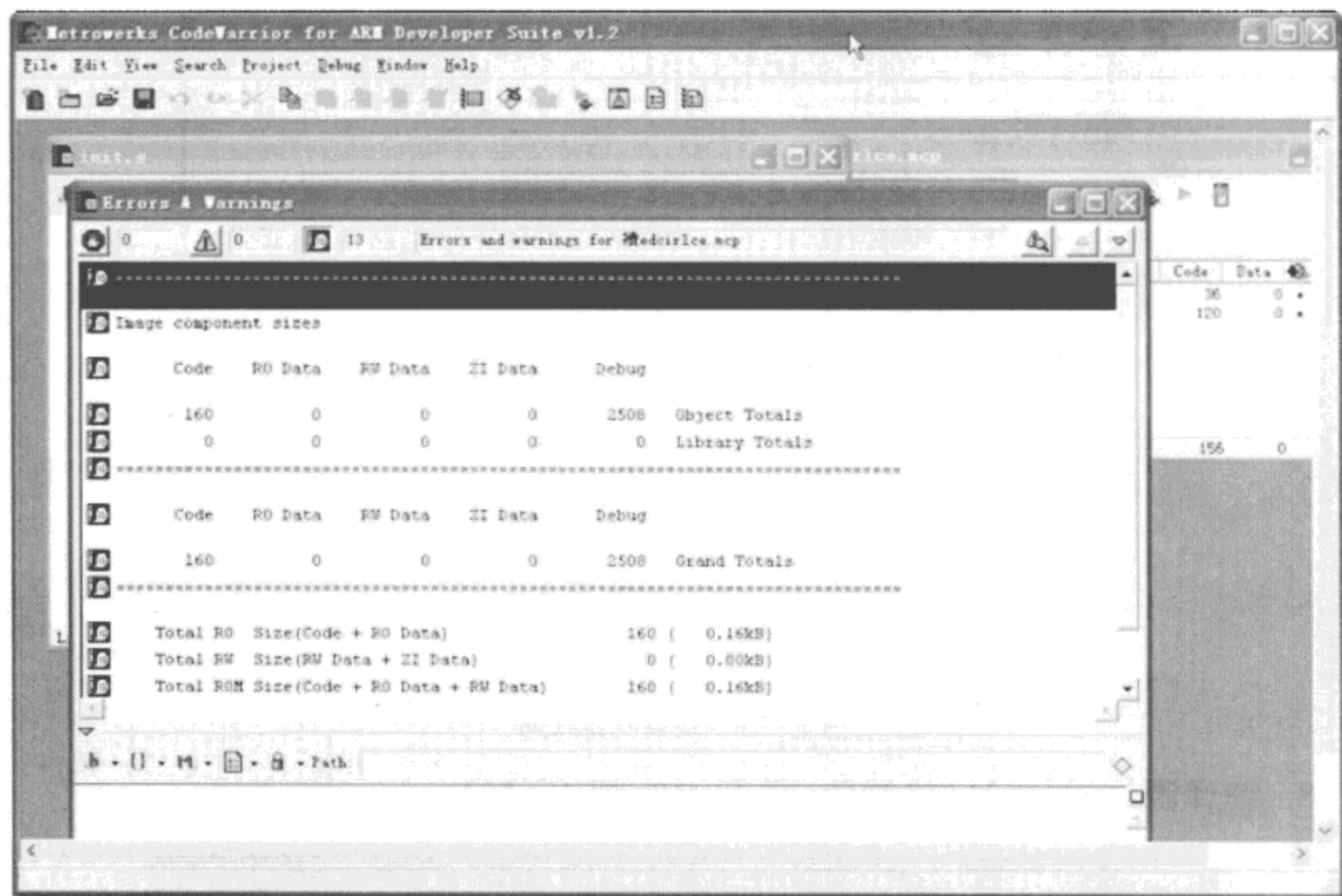


图 1-16 编译结果

以上是最普通(也是最简单)的源代码编辑和编译过程。但在很多时候,在 Make 之前都需要对编译和链接选项进行设置,下面对常用的设置进行演示。

进入编译链接设置如图 1-17、图 1-18 所示。

目标设置如图 1-18 所示。

汇编选项设置如图 1-19 所示。

C 编译器选项设置如图 1-20 所示。

文件输出设置如图 1-21 所示。



图 1-17 进入编译链接设置

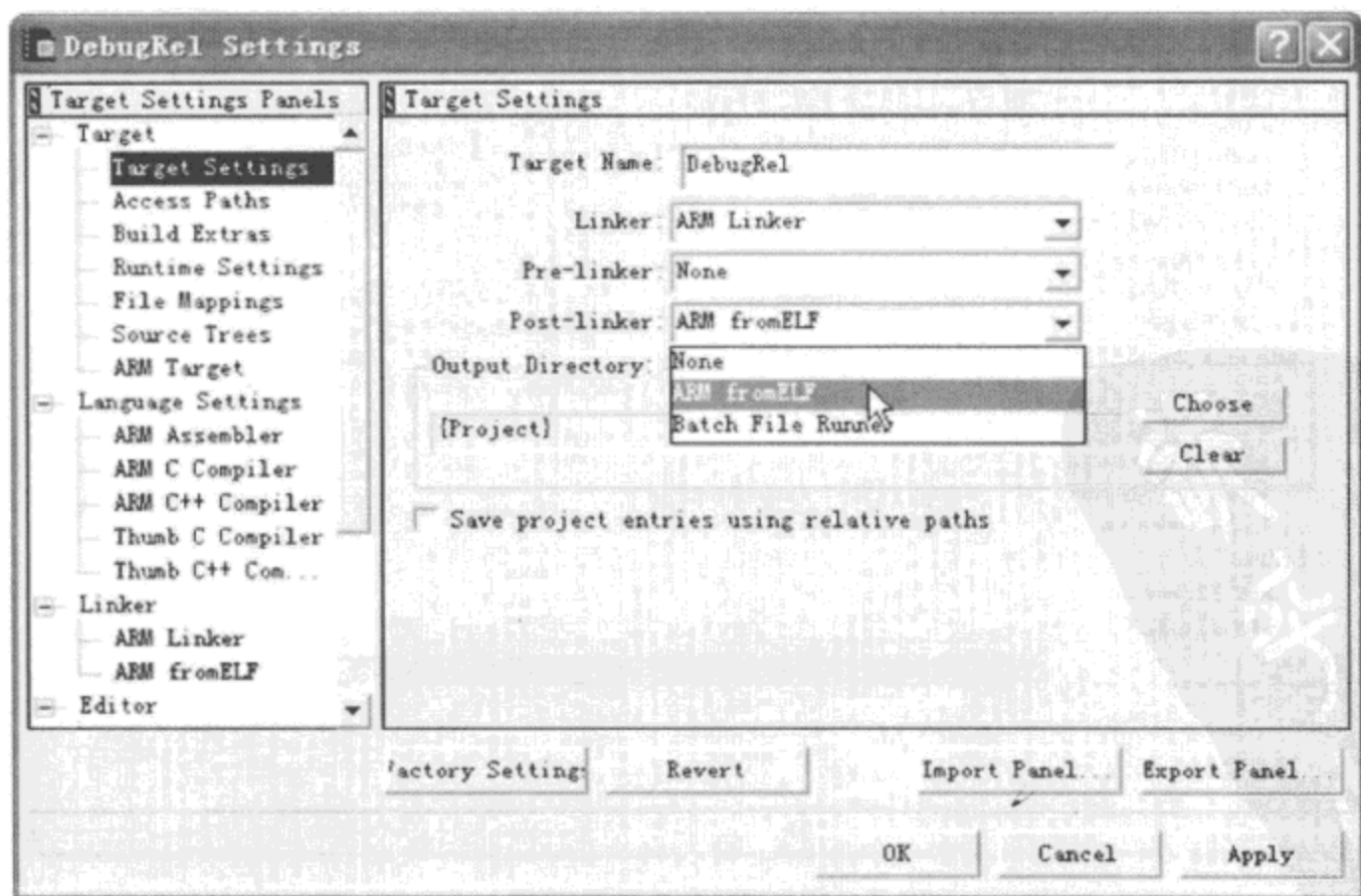


图 1-18 目标设置

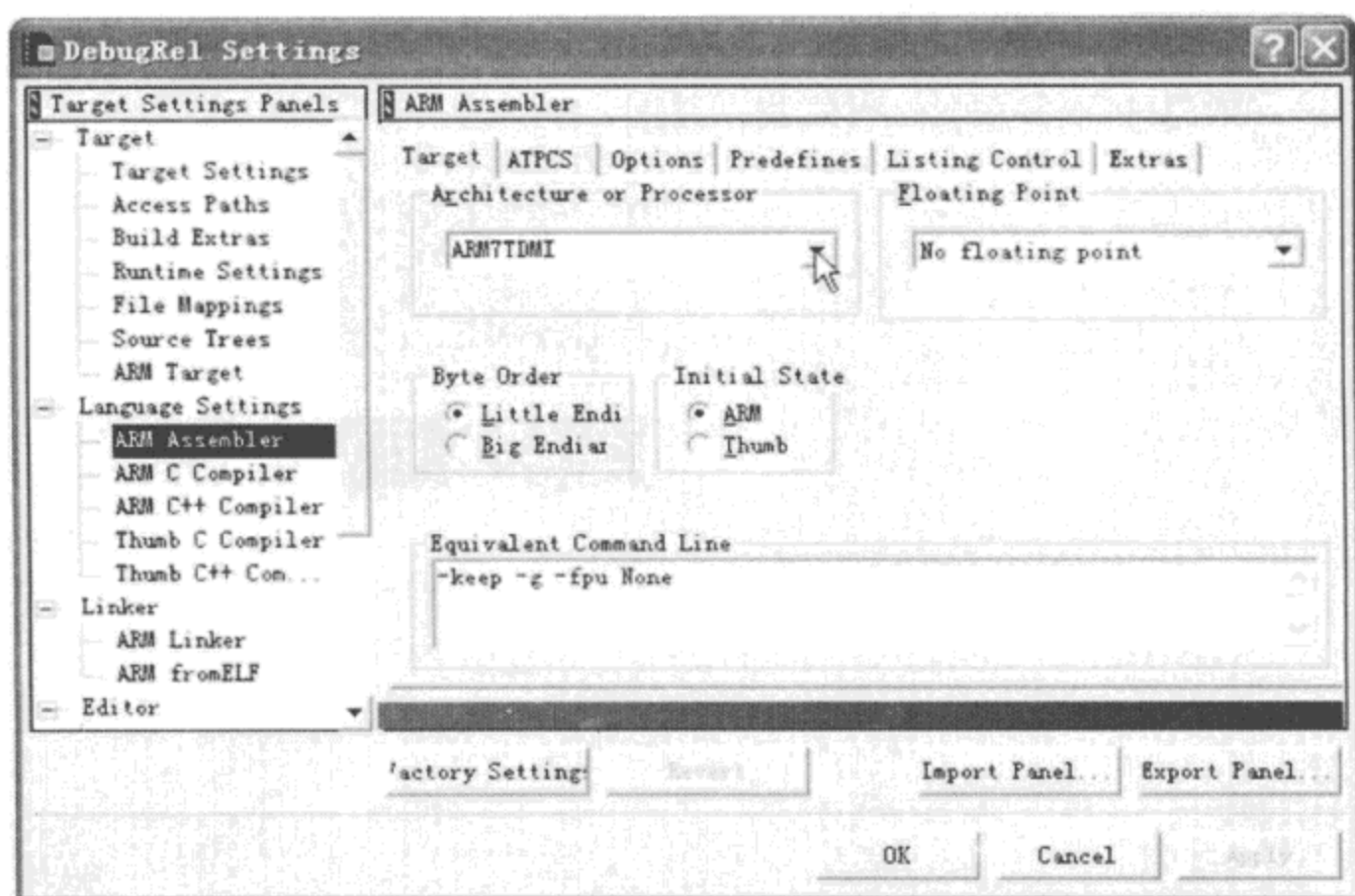


图 1-19 汇编选项设置

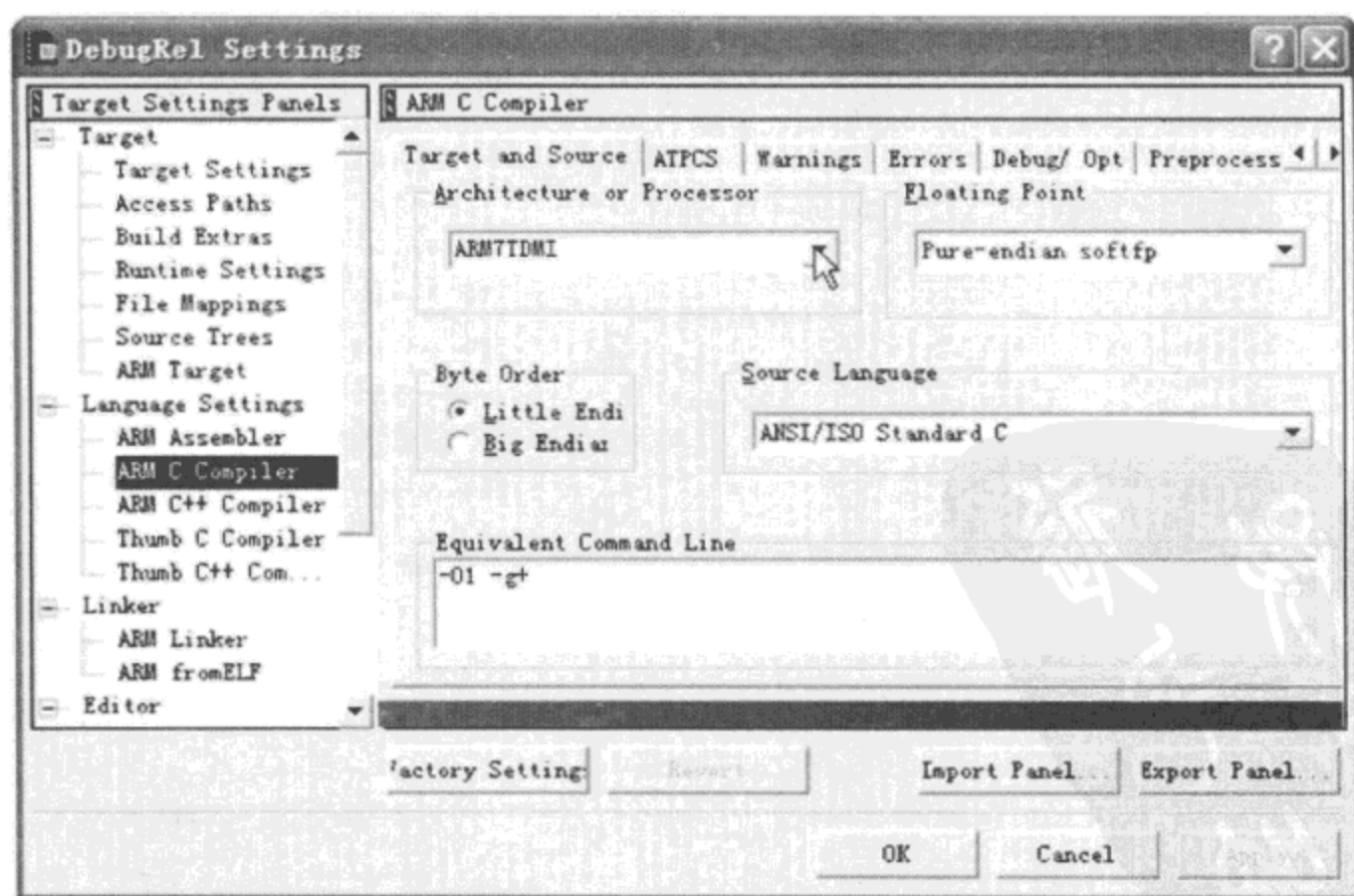


图 1-20 C 编译器选项设置

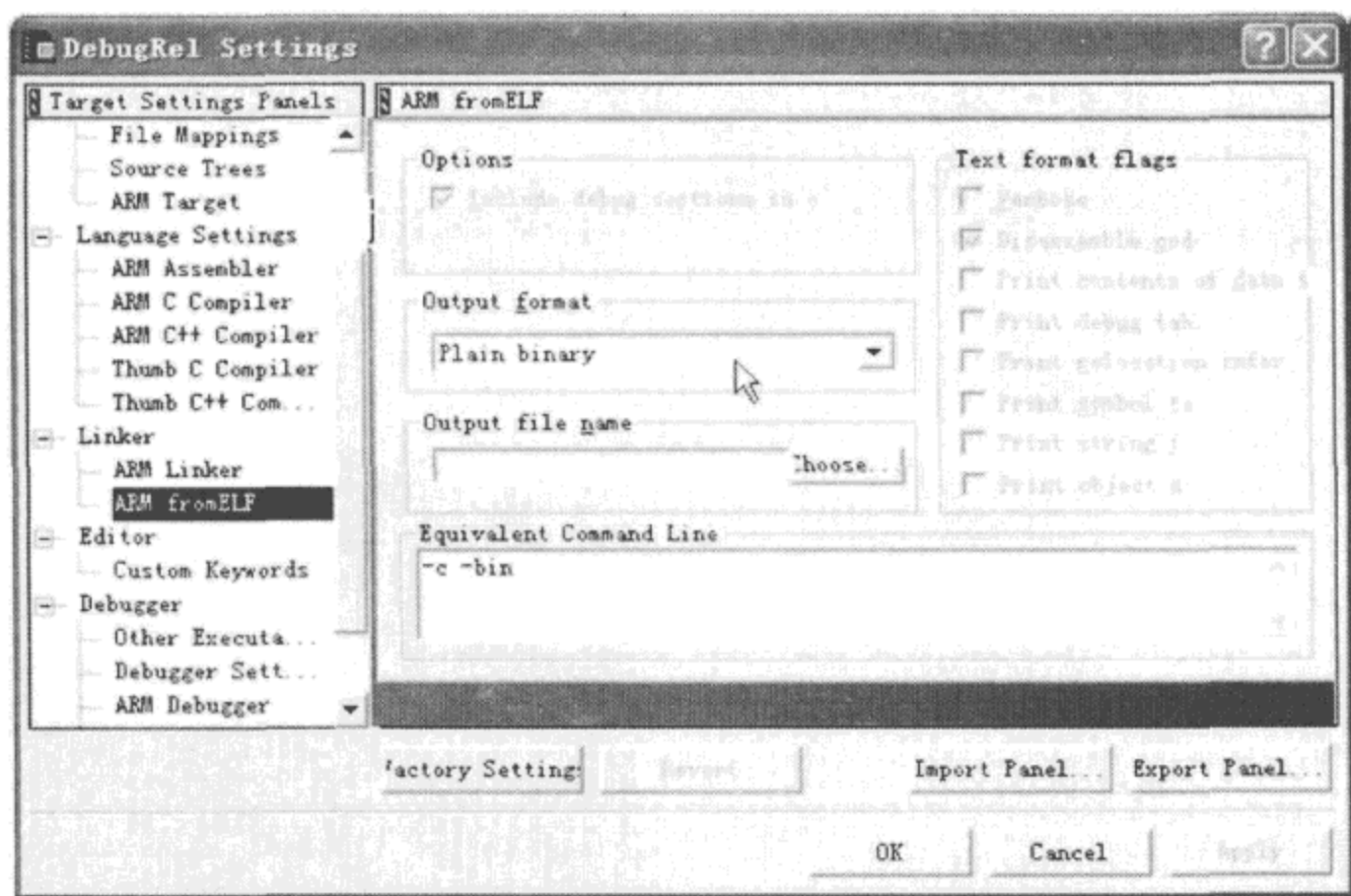


图 1-21 文件输出设置

1.4.2 在 AXD 中进行裸机程序调试的方法与步骤

在完成程序的编译、链接生成二进制可执行文件后,很多情况下需要对程序进行调试,ADS 采用的调试工具是 AXD,在调试中可以完成:

- 设置断点及单步运行与调试。
- 查看各种模式下的 CPU 寄存器。
- 查看内存内容。
- 查看 C 语言变量。

先对调试器 AXD(从 ARM Developer Suite v1.2→AXD Debugger 进入)进行设置,如图 1-22、图 1-23 所示,以使用正确的模拟开发板 ARMUL。

完成上面的设置后,下面就展示如何使用 AXD 对程序进行调试(图 1-24):(示例代码在光盘\work\armarch\ledcircle 中)

- (1) 使用 ADS 打开 ledcircle.mcp 工程。
- (2) 单击菜单 project→debug。
- (3) 单击 Porcessor Views→registers,注意窗口左上部分(图 1-25)。

(4) 展开 Current,可发现当前模式处于 SVC 模式,并可查看当前模式下 CPU 的各个寄存器的值,如图 1-26 所示,例如 r13=0,pc=0x8000。

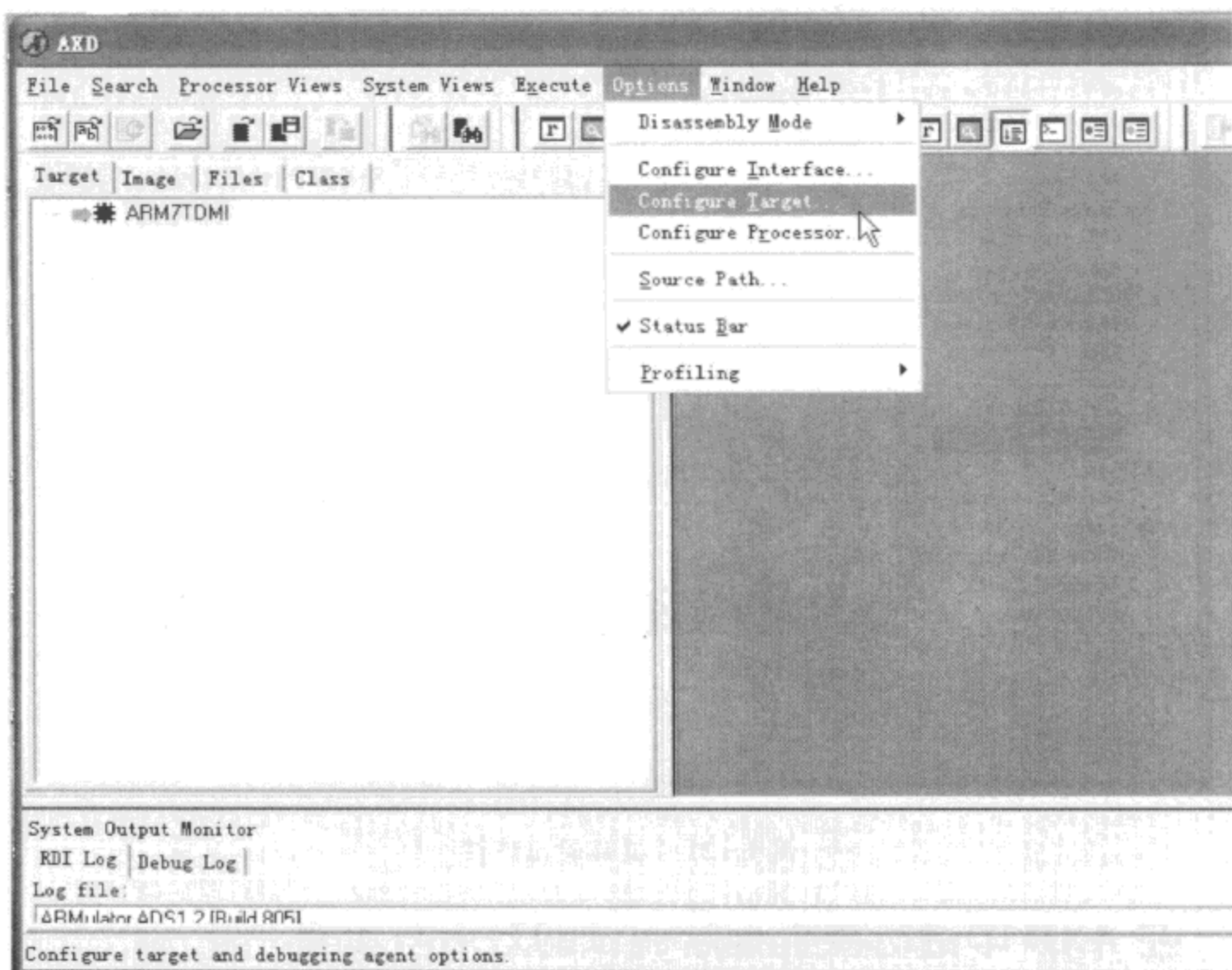


图 1-22 配置目标菜单

还可查看其他模式下,CPU 各寄存器的值。

(5) 单击 Execute→step,可以单步运行程序,如图 1-27 所示。

此时发现 PC 值增加了 4,而 r13 的值也由于 ldr 指令的赋值操作变为了 0x34000000。

(6) 单击 Execute→step-in,可以单步跟踪进入子程序,如图 1-28 所示。

(7) 按一次 F8,单击 Processor Views→Memory,如图 1-29 所示。

(8) 在 Memory Start addr 处输入 0x56000010,再按一次 F8,可以看到内存 0x56000010 处的值变为了 0x00015400,如图 1-30 所示。

(9) 单击 Processor View→Variables 可查看各种 C 变量(例如局部变量 i)的值,如图 1-31 所示。

(10) 单击 Execute→Toggle Breakpoint,设置断点,如图 1-32 所示。

(11) 继续单步执行程序,当程序再次回到断点是,观察 i 的值已经变为 1 了,如图 1-33 所示。

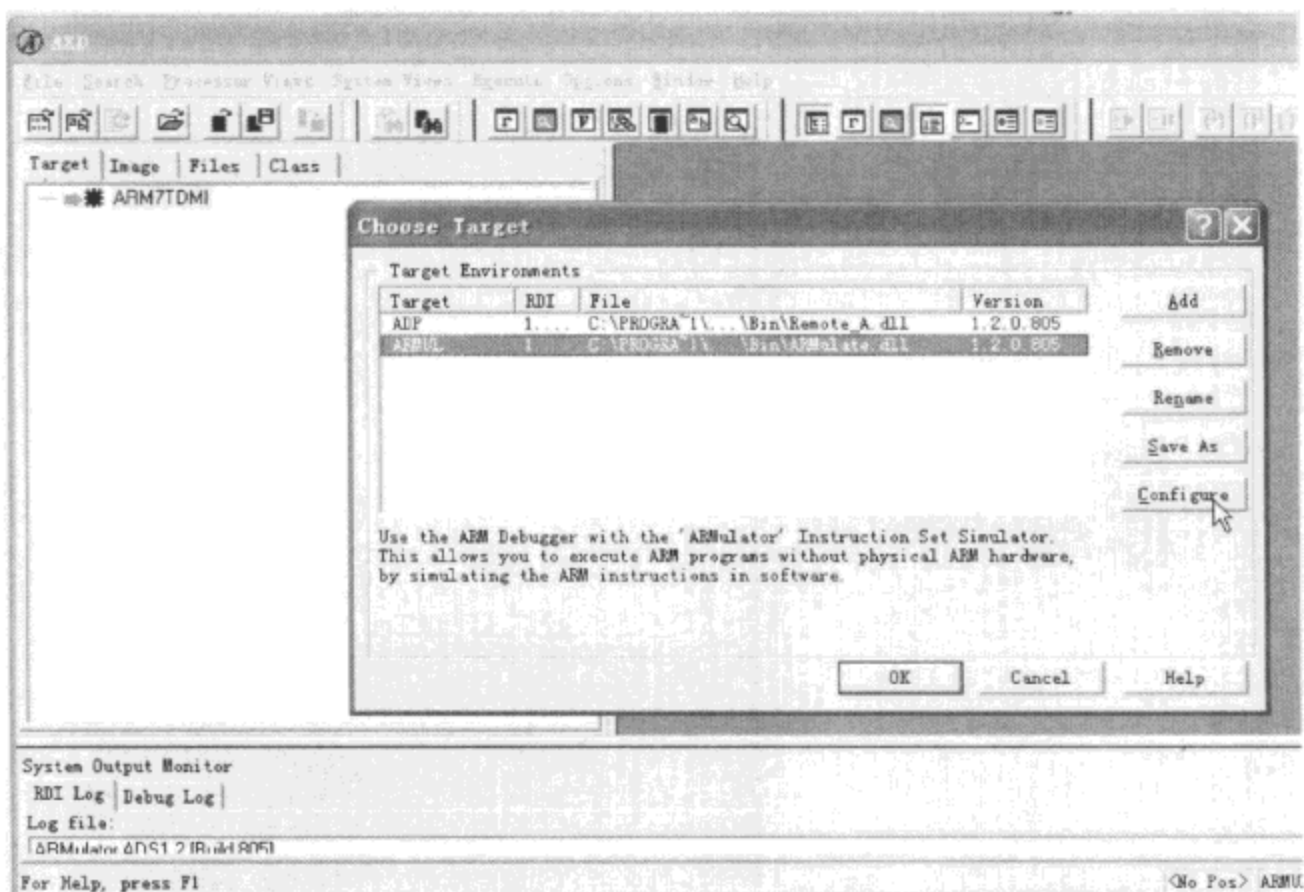


图 1-23 配置目标

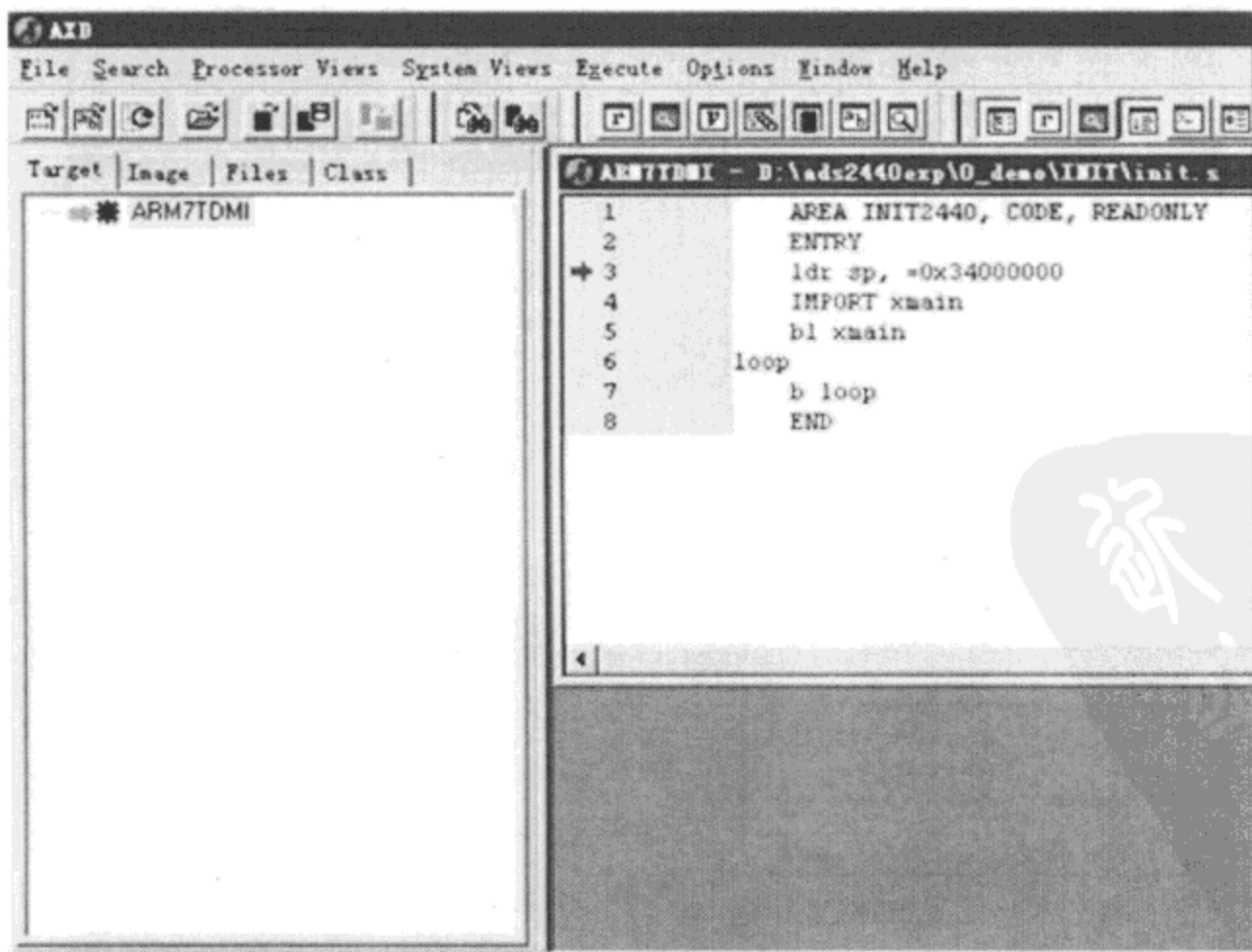


图 1-24 调试界面

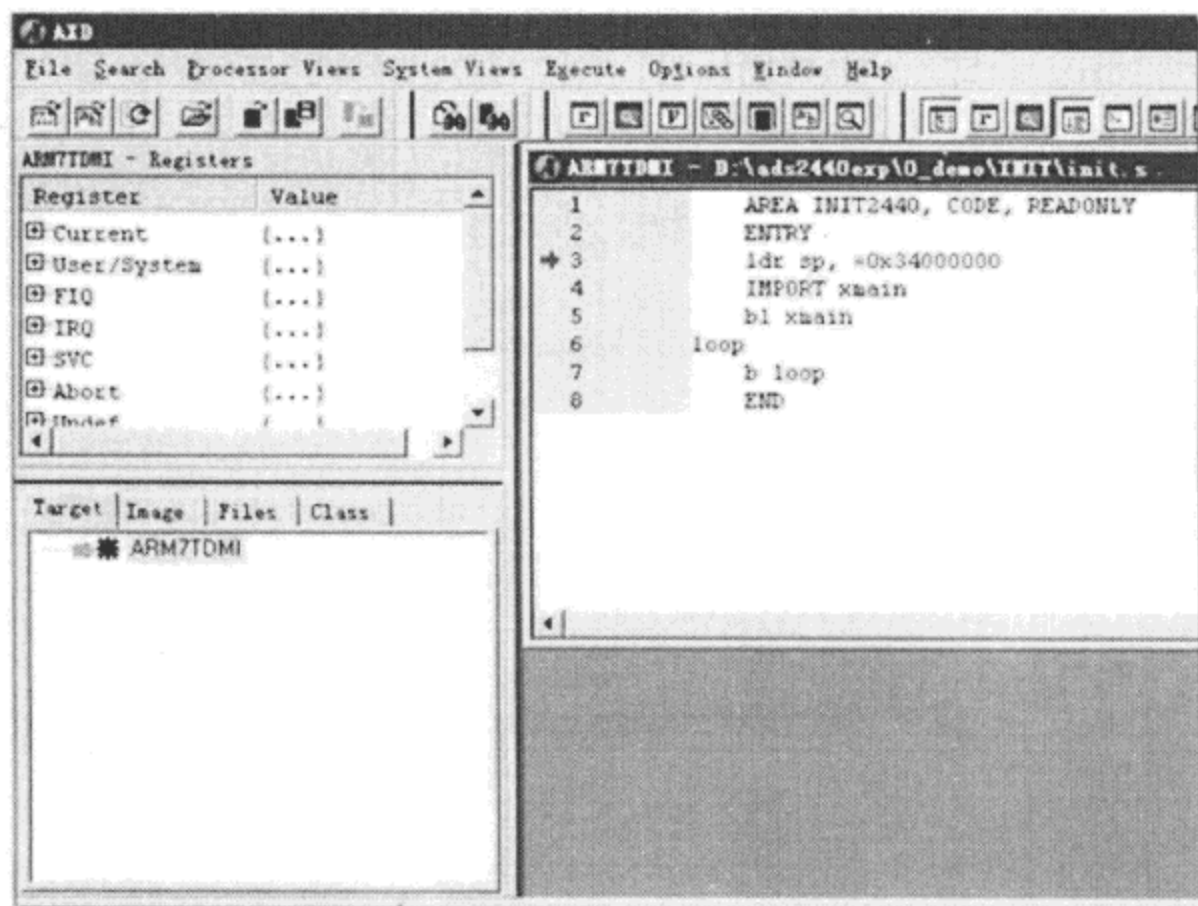


图 1-25 进入查看 CPU 寄存器

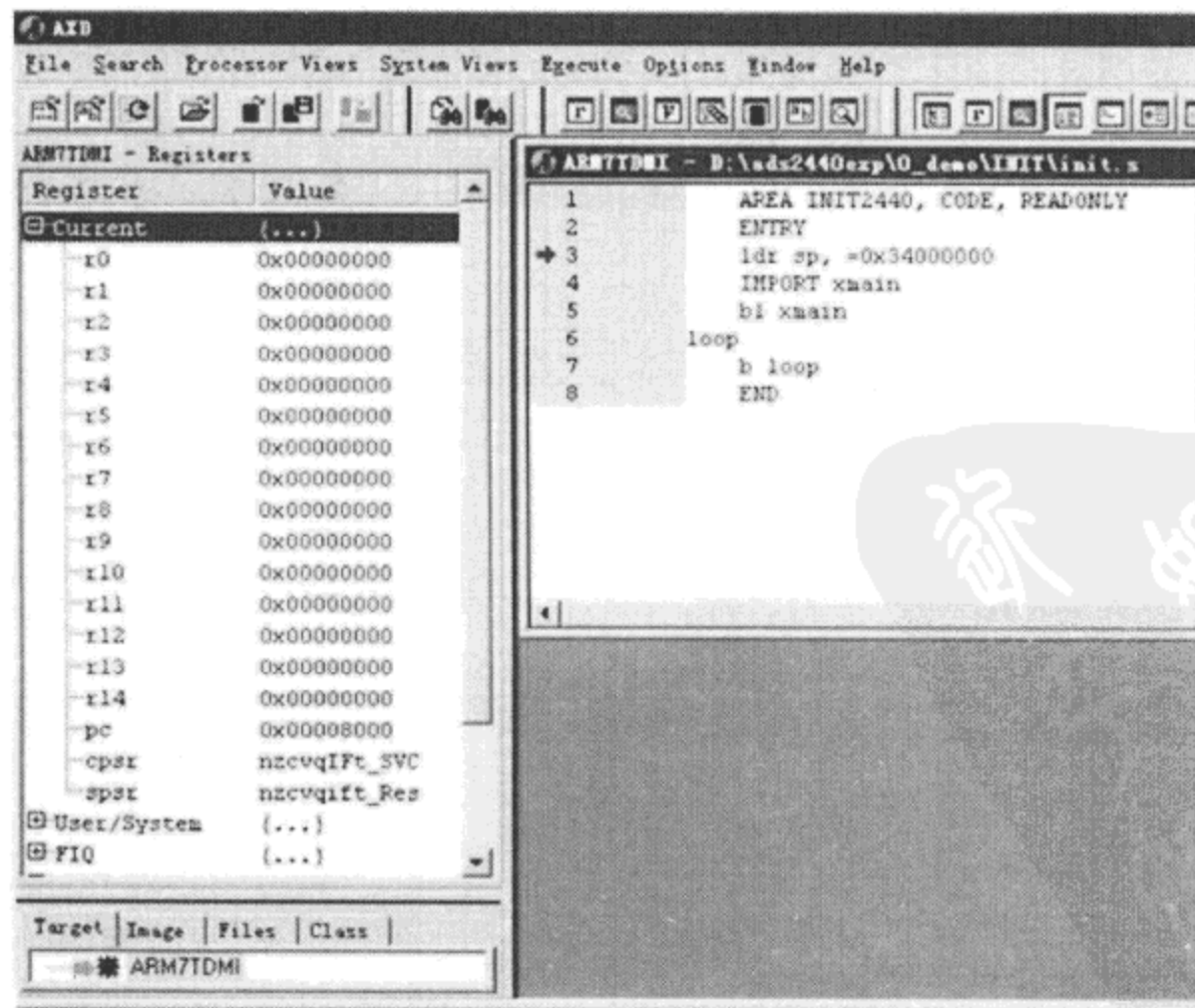


图 1-26 查看 CPU 寄存器

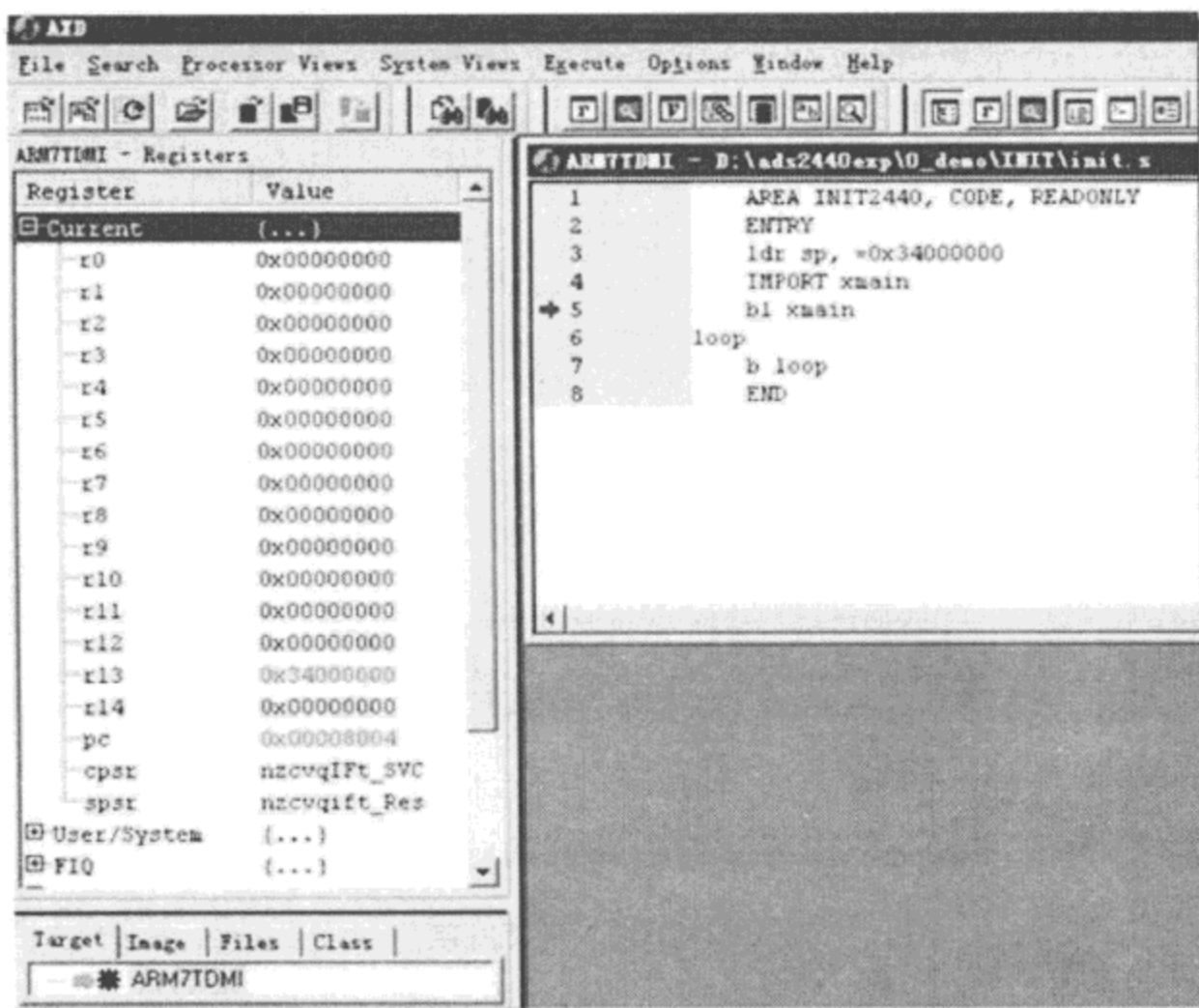


图 1-27 单步执行结果

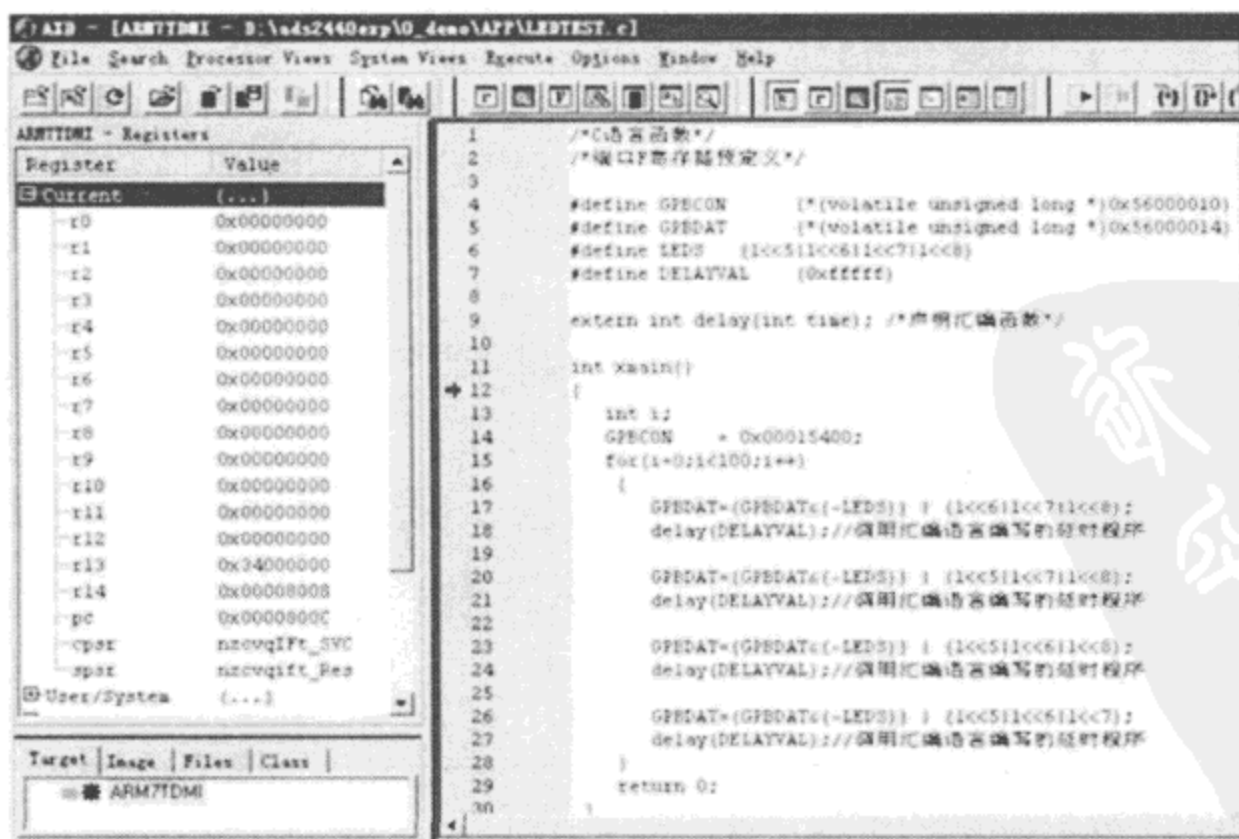


图 1-28 单步跟踪进入子程序

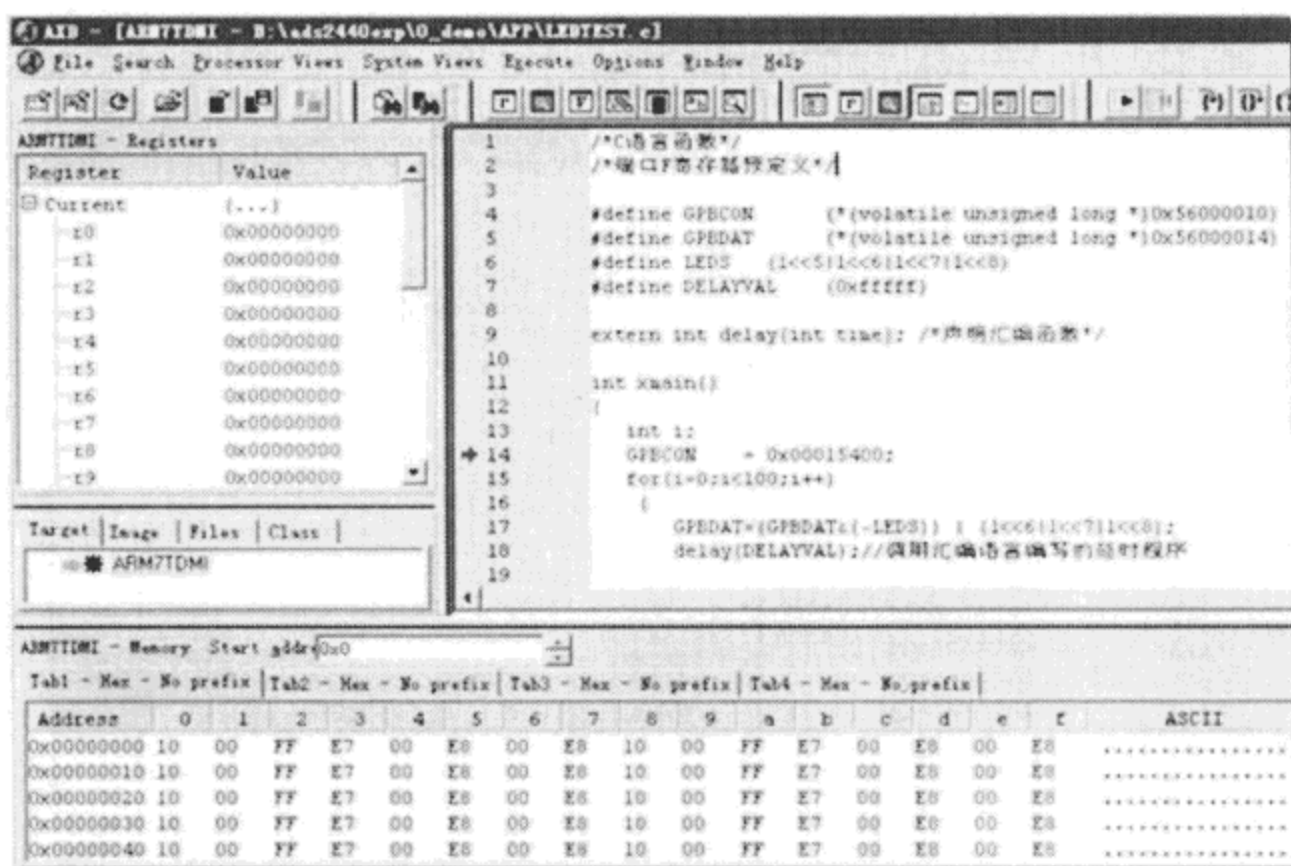


图 1-29 进入查看内存

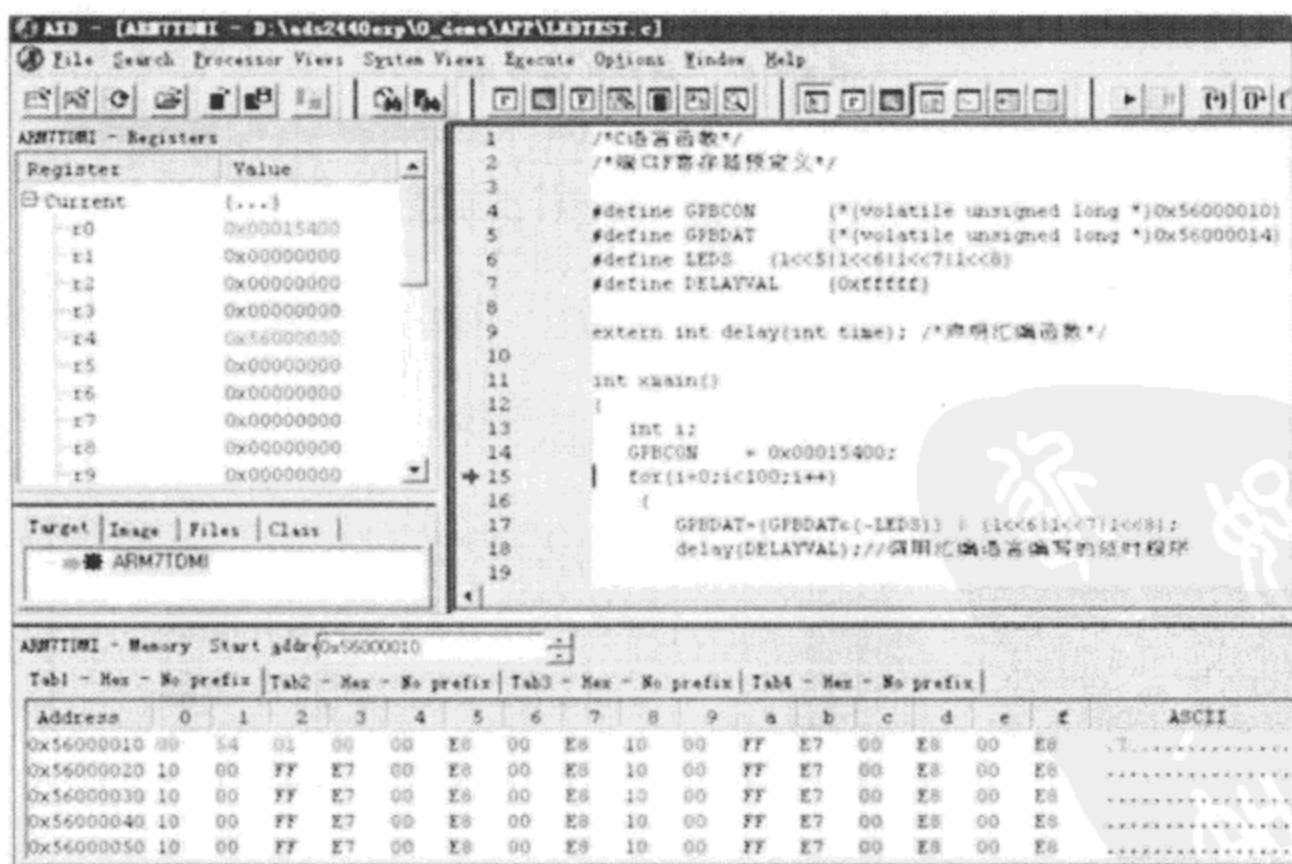


图 1-30 查看内存变化

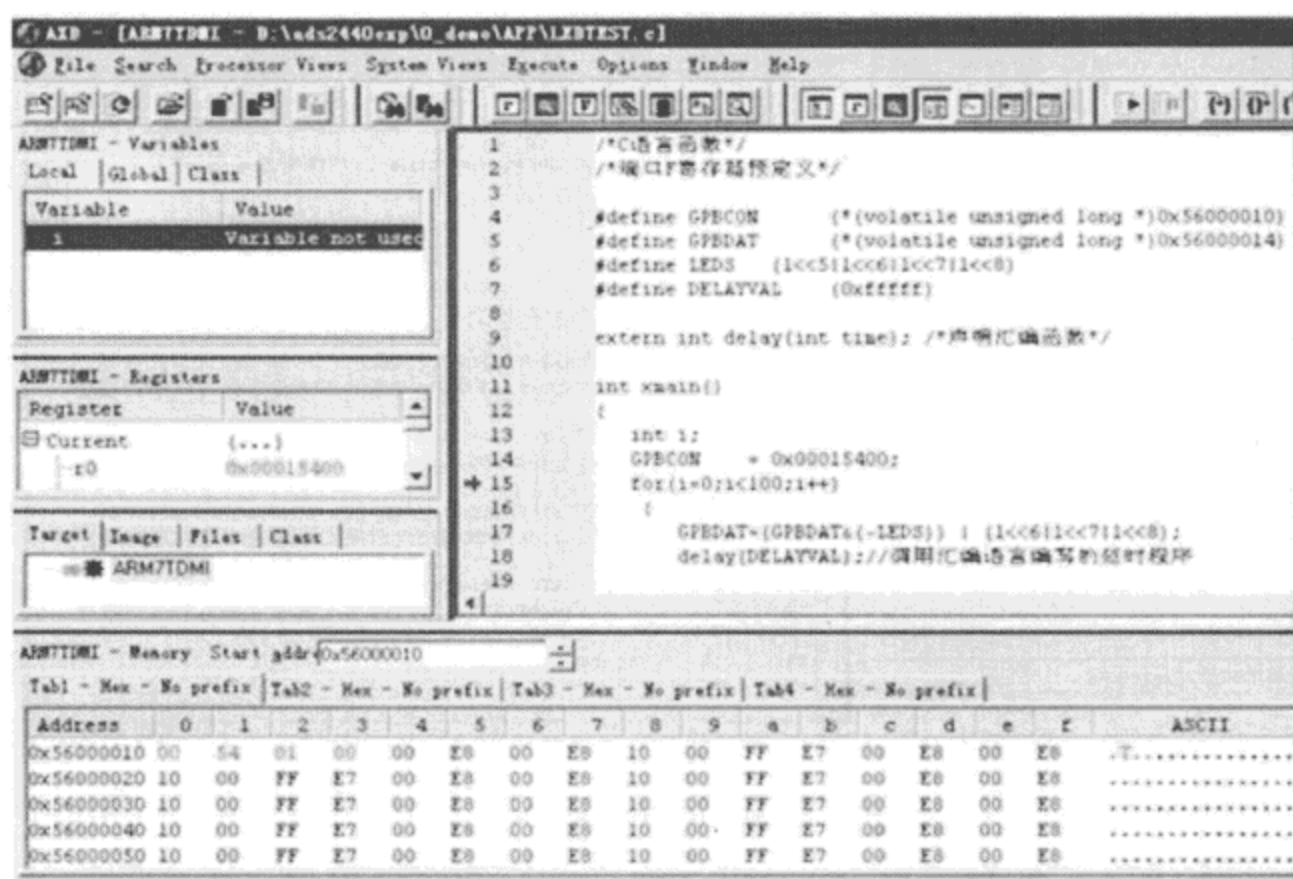


图 1-31 查看变量

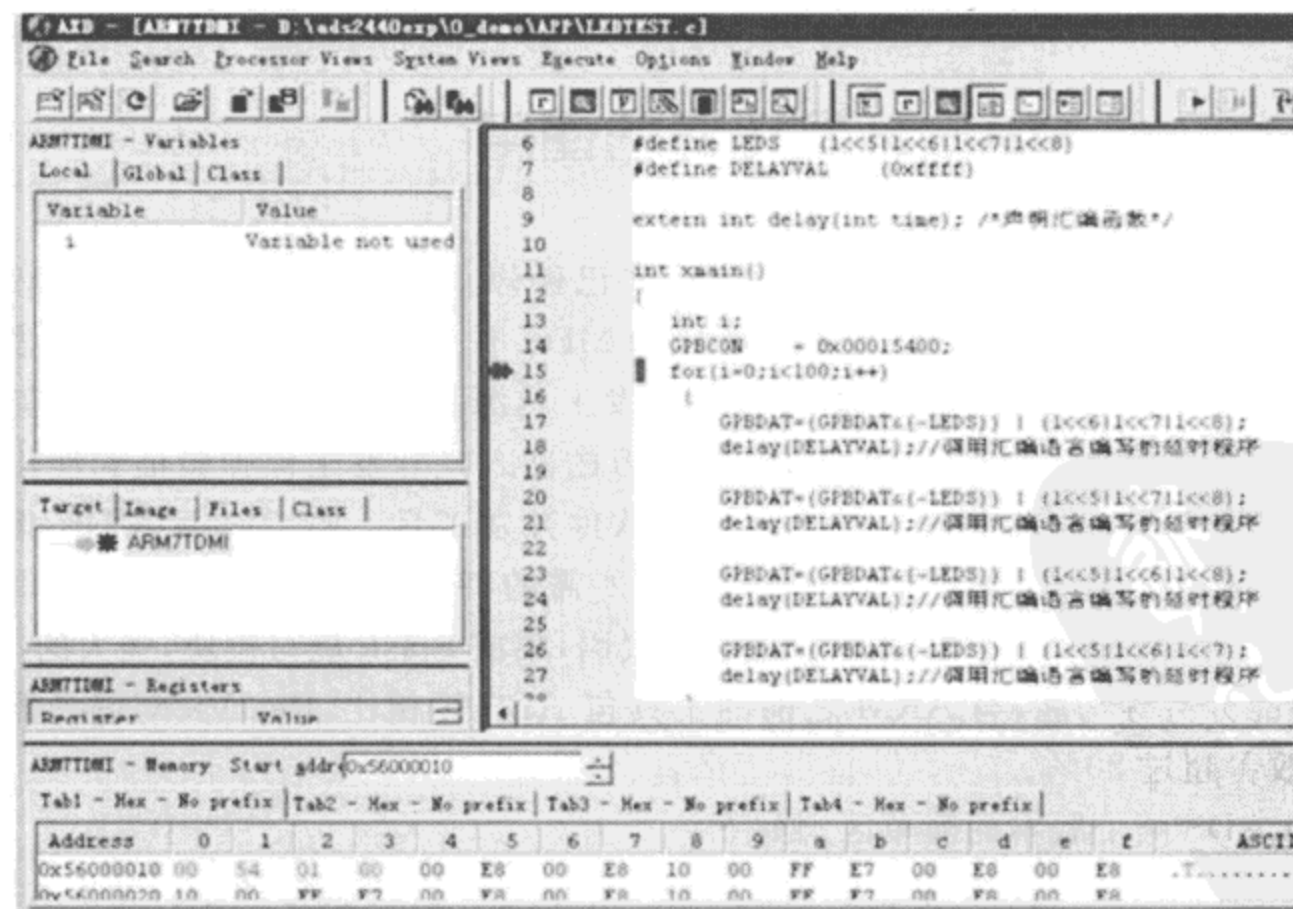


图 1-32 设置断点

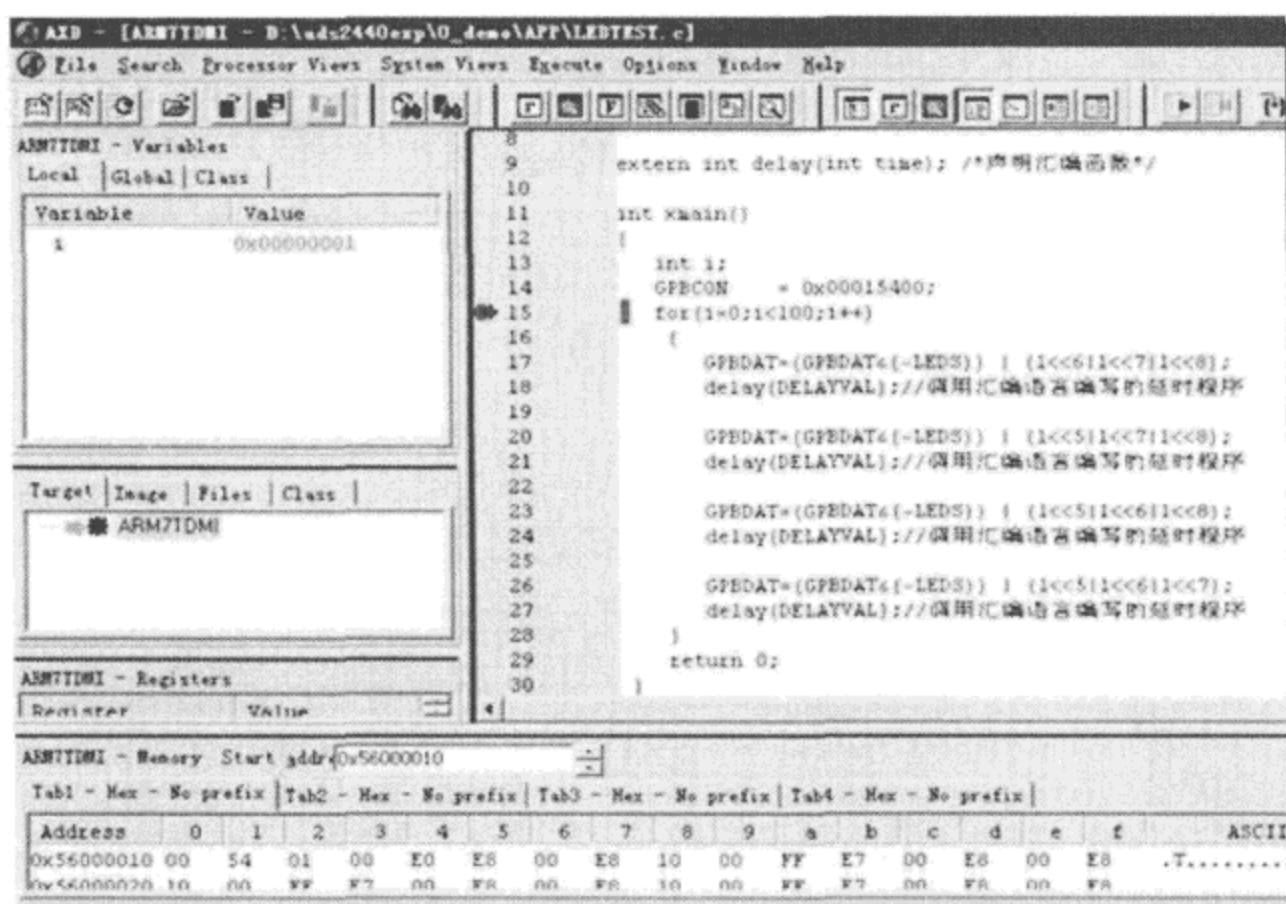


图 1-33 在断点处查看变量

1.5 RealView MDK 开发环境的使用

ADS1.2 集成开发工具 ARM 公司在 2001 年已经停止对其维护和支持,ARM 公司目前推出了新的、功能更加强大的、用户界面更友好的 MDK 作为其硬件平台的集成开发工具。

MDK(Microcontroller Development Kit)开发工具源自德国 Keil 公司,由于其拥有流畅的用户界面与强大的仿真功能而被全球超过 10 万的嵌入式开发工程师验证和使用,是 ARM 公司目前最新推出的针对各种嵌入式处理器的软件开发工具。RealView MDK 集成了业内最领先的技术,融合了中国多数软件开发工程师所需的特点和功能。它支持 ARM7、ARM9 和最新的 Cortex-M3 核处理器,自动配置启动代码,集成 Flash 烧写模块,强大的 Simulation 设备模拟,性能分析等功能,与 ARM 之前的工具包 ADS 等相比,RealView 编译器的最新版本可将性能改善超过 20%。

MDK 与 ADS 两个版本的简单区别如下:

(1) 技术支持。ADS 2000 年已被淘汰,2001 年 ARM 公司就停产了,ADS 从 2001 年开始已经停止对新 ARM 内核的支持。

(2) 模拟器功能。ADS 模拟器只能模拟指令集,而 MDK 的模拟器能提供指令集、启动代码、外设、中断等整个 MCU 行为的模拟。

(3) 性能分析器。MDK 提供性能分析器,而 ADS 没有。

(4) 方便的代码生成向导。MDK 提供启动代码生成向导,轻松完成启动生成,而 ADS 没有此功能。

(5) CM3 的支持。ADS 不支持 CM3,MDK 支持。

(6) 方便快捷的操作。MDK 的项目管理窗口、编译窗口、调试窗口等都在同一个界面,操作方便,上手更易;而 ADS 的项目管理窗口、编译窗口、调试窗口等需要在不同界面操作,相对不便。

(7) Flash 烧写支持。ADS 不支持 Flash 烧写,MDK 支持。

笔者使用 MDK 版本为 μ Vision V4.10,请读者自行下载并安装。

1.5.1 在 MDK 开发环境下编写裸机程序

MDK 集成开发环境的界面主要由工程管理区、编辑区和信息输出区组成,如图 1-34 所示。

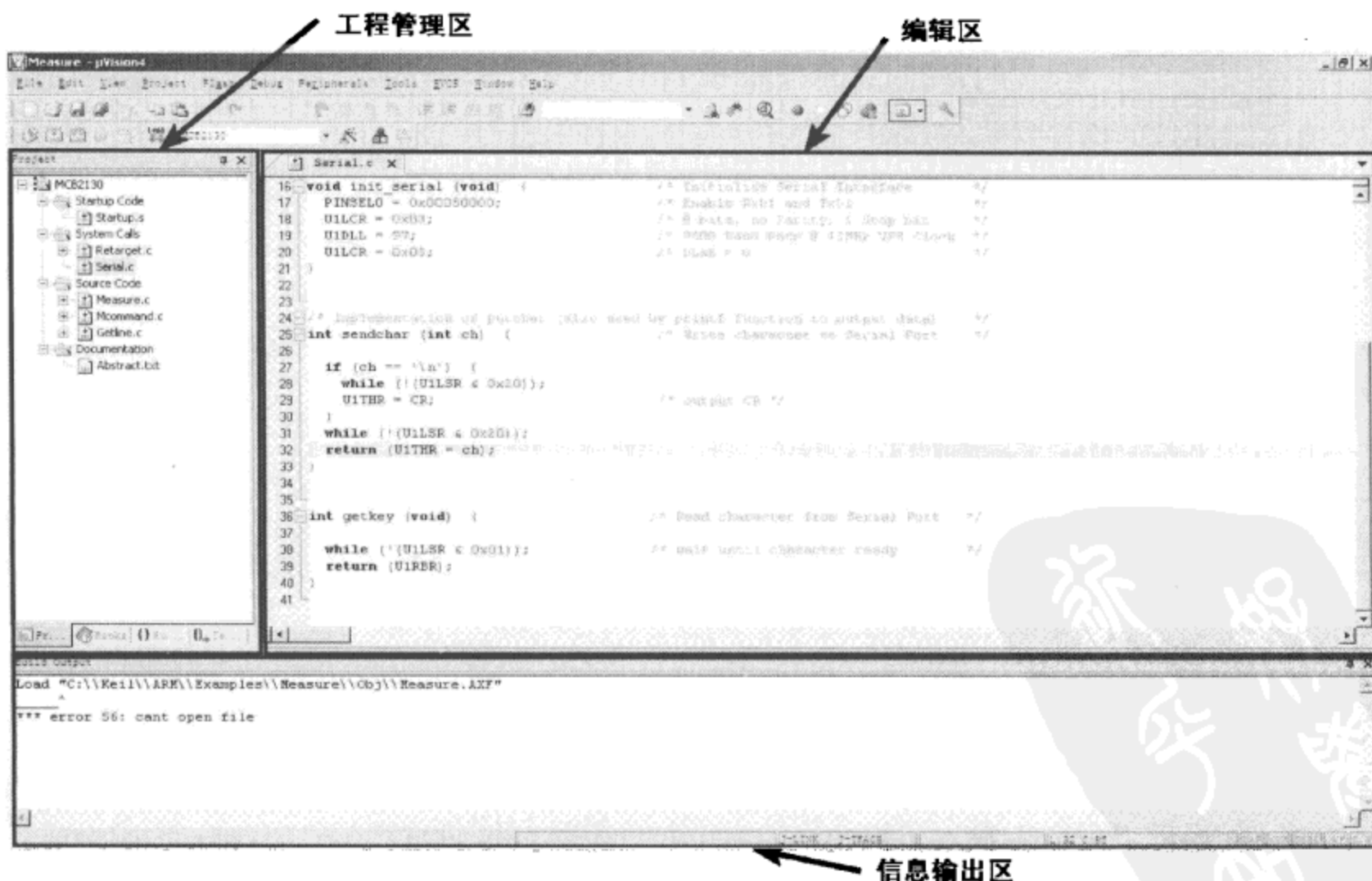


图 1-34 MDK 启动界面

(1) 工程管理区:主要显示工程文件资源,帮助文档信息等。

第1章 ARM 汇编编程基础

(2) 编辑区:是占据了整个开发环境的大半部分,主要用于显示,编辑和调试代码。

(3) 信息输出区:用来显示工程编辑和调试过程中产生的信息,主要包括调试信息、内存信息、符号信息、函数栈、局部变量、命令行、查找等相关信息。

默认打开 MDK 时,会打开上一次的操作工程,首先新建一个工程,对程序进行编辑和编译,过程如下:

(1) 建立工程(图 1-35)。

① 在磁盘里新建一个目录 D:\mdk。

② 打开 MDK 软件。

③ 单击工具栏 Project,在下拉菜单中单击 New μ Vision Project 命令。

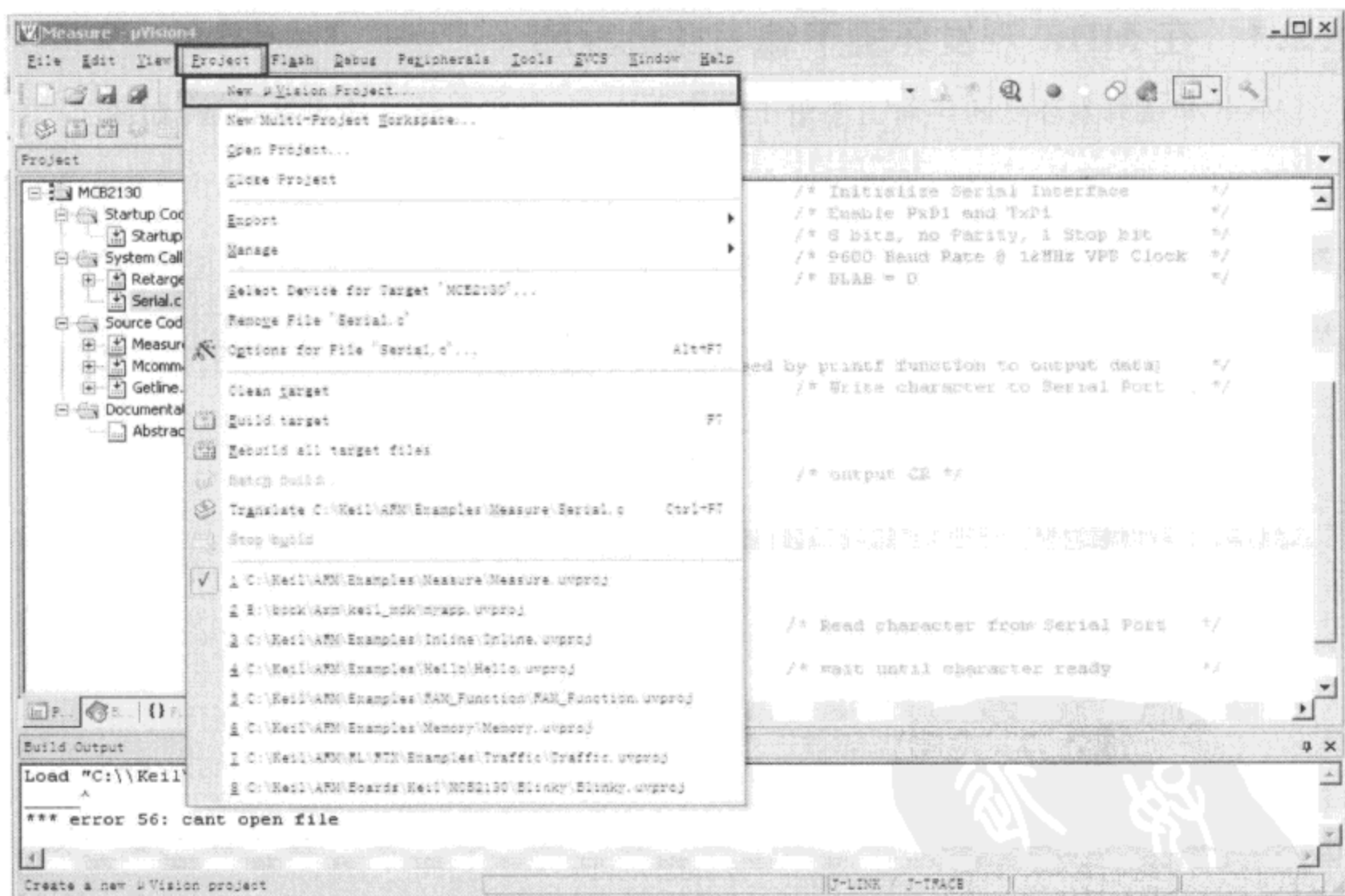


图 1-35 新建工程

(2) 如图 1-36 所示,输入工程名称“ledcircle”,选择 mdk 目录。

(3) 单击“保存”按钮后,会自动弹出 Select Device for Target...窗口,它是用来为新建工程选择使用设备型号。可以根据你使用的处理器来选择,如果在列表中找到,也可以找一款与您使用相兼容的型号来代替。我们选择 Samsung→S3C2440A 型号设备,如图 1-37 所示,



图 1-36 选择工程目录

因为 mini2440 使用三星的 S3C2440A 型号 CPU。

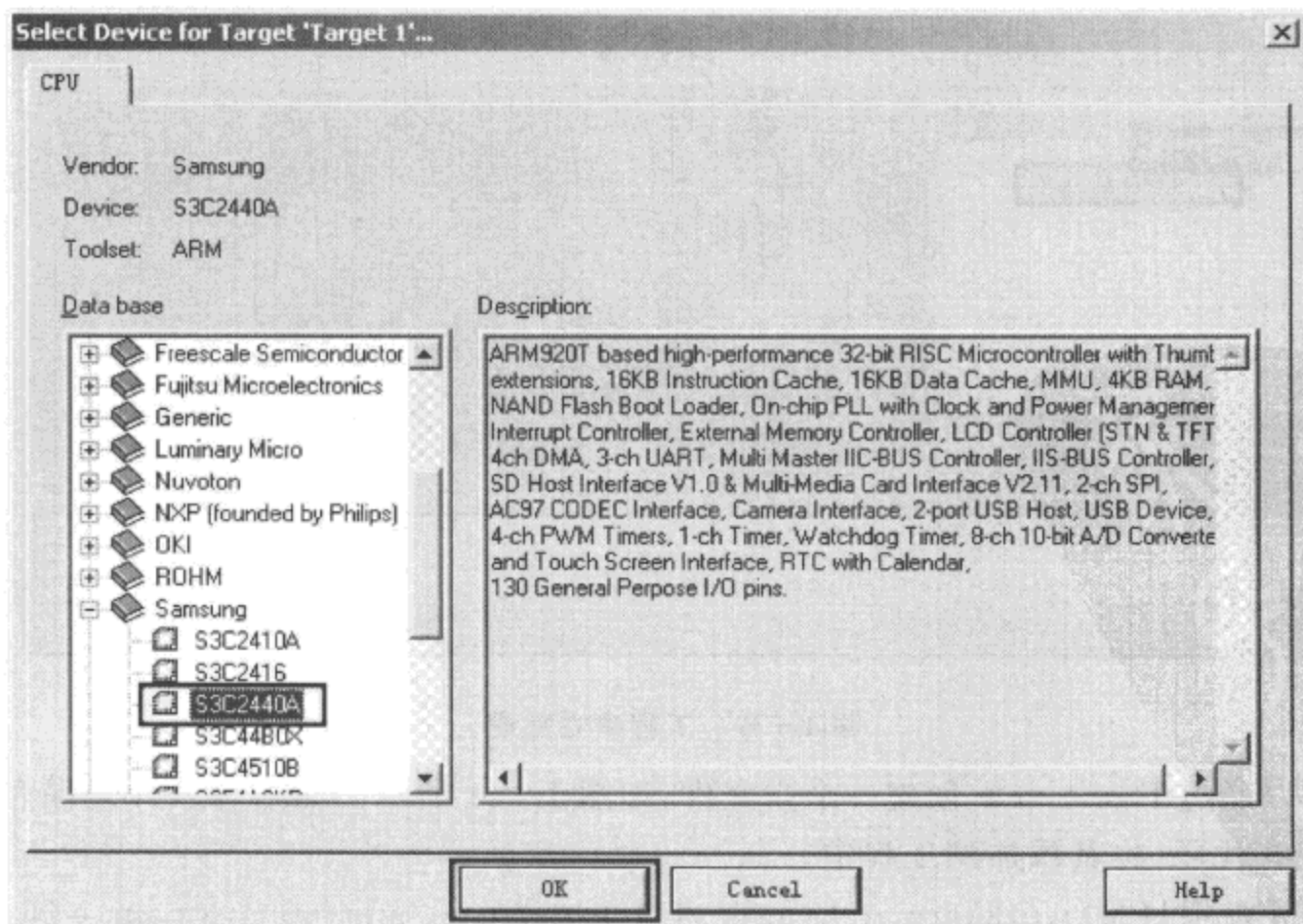


图 1-37 设备型号

第1章 ARM 汇编编程基础

(4) 这时会弹出让用户选择是否复制启动代码对话框,MDK 开发环境为所有常见型号设备提供了一个启动代码文件,用户可以方便地修改启动代码文件对设备进行初始化,我们不使用它提供的启动代码,因为 LED 跑马灯程序中已经包含启动代码。单击“否”按钮,如图1-38 所示。

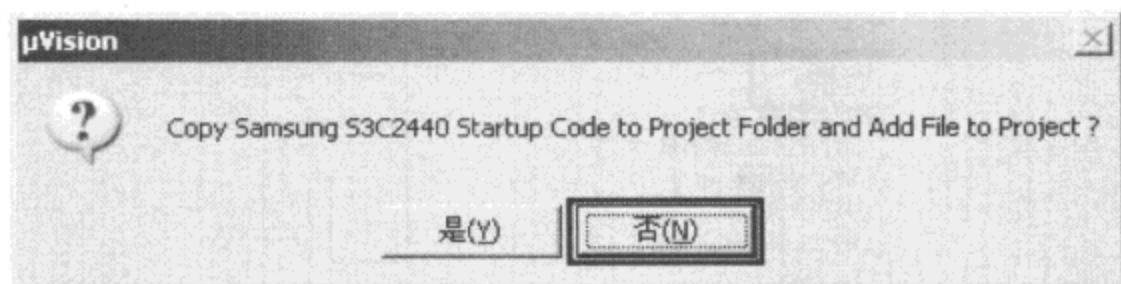


图 1-38 是否复制启动代码

(5) 这时工程就建立完了,但是工程里没有任何的文件,如图 1-39 所示,在工程管理可以看到目标设备名为:Target1,源码文件组名:SourceGroup1,可以通过右击 Target1,调出菜单选择 Manage Component 选项,在弹出窗口中双击修改目标设备名和源码组名,如图 1-40 所示。

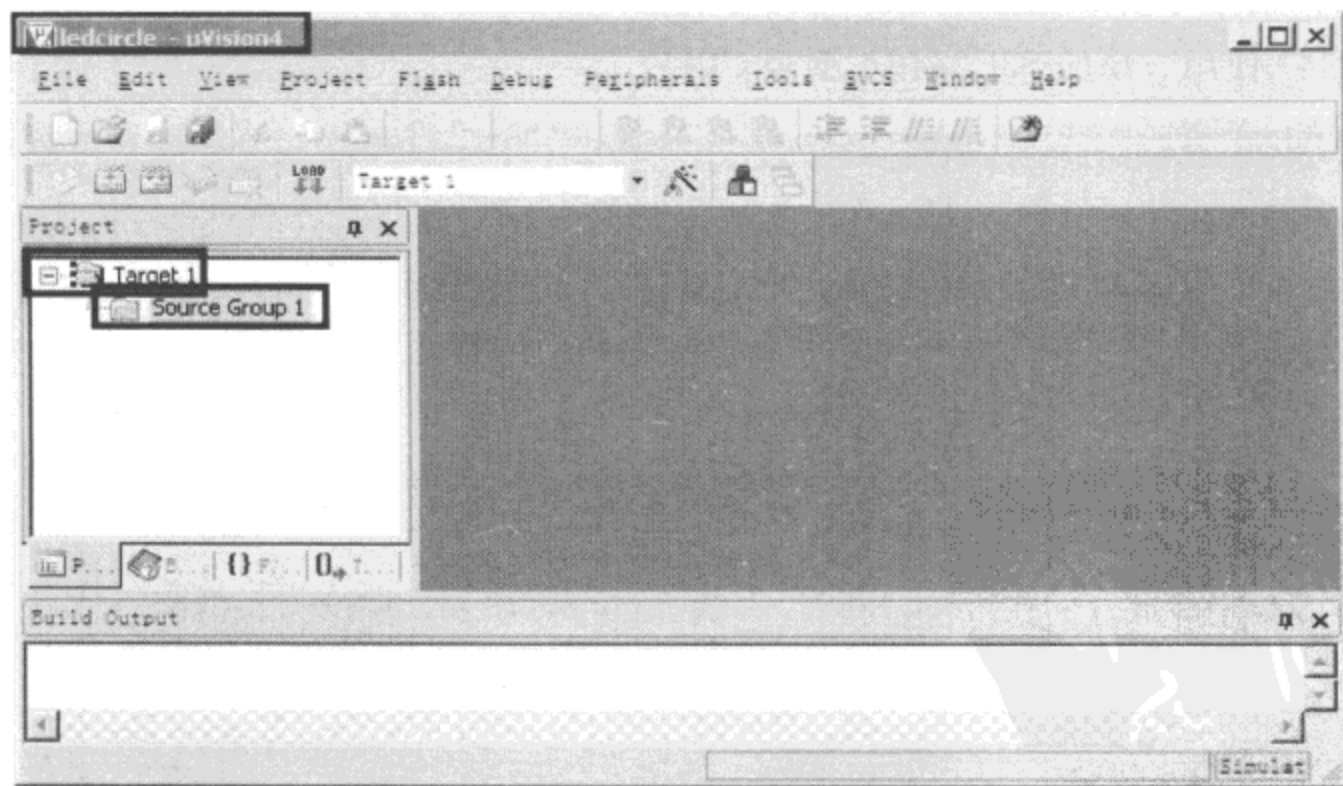


图 1-39 工程建立完成

(6) 可以通过 File→New 创建一个空文件,如图 1-41 所示。然后写入代码保存,也可以直接使用现有代码,将其添加到工程中。

(7) 将光盘根目录\work\armarch\mdk 目录下 delay.s, startup.s, xmain.c 这 3 个文件复制到新建的工程目录 D:\mdk 中,如图 1-42 所示。

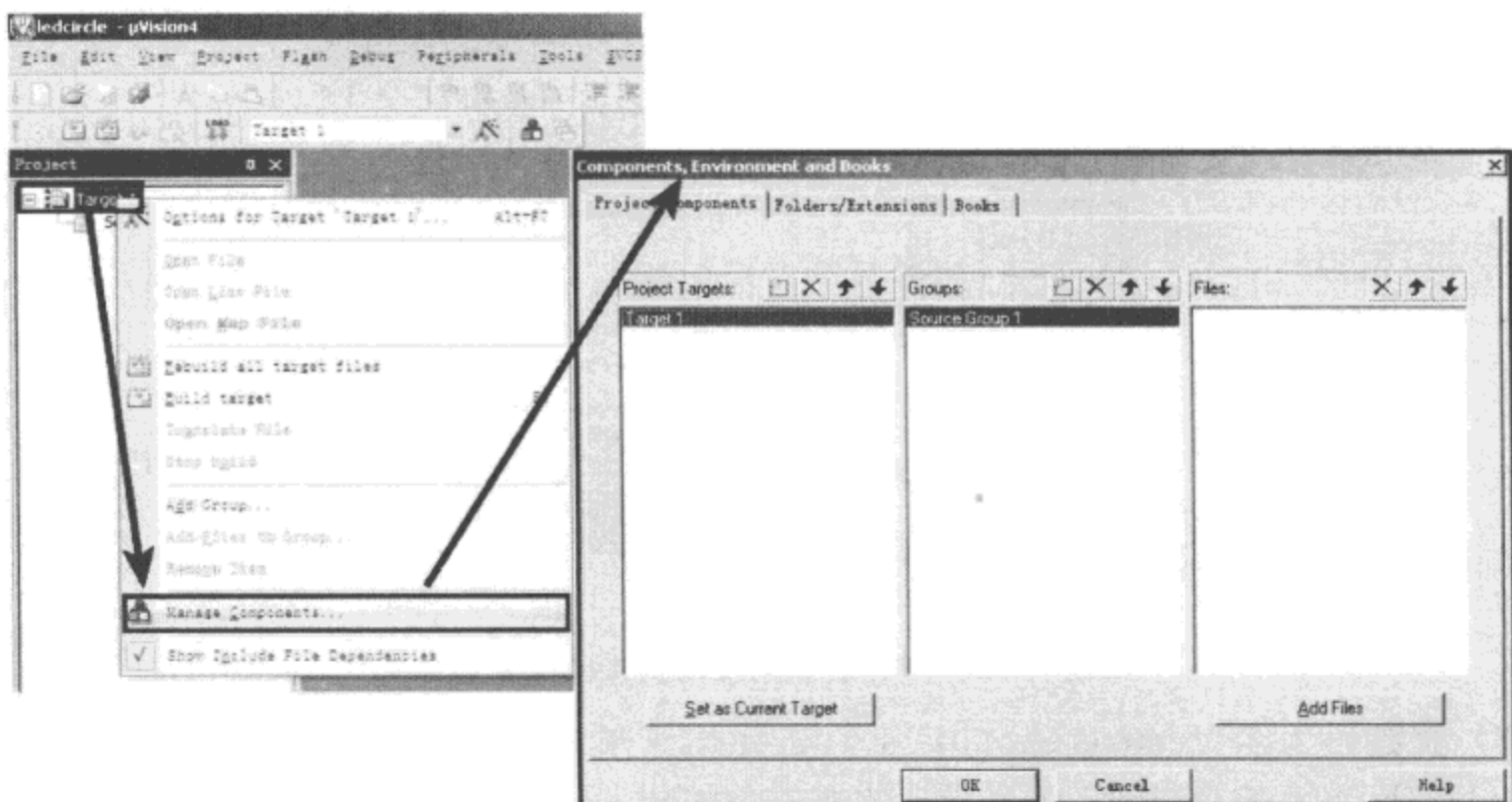


图 1-40 修改目标设备名和源码组名

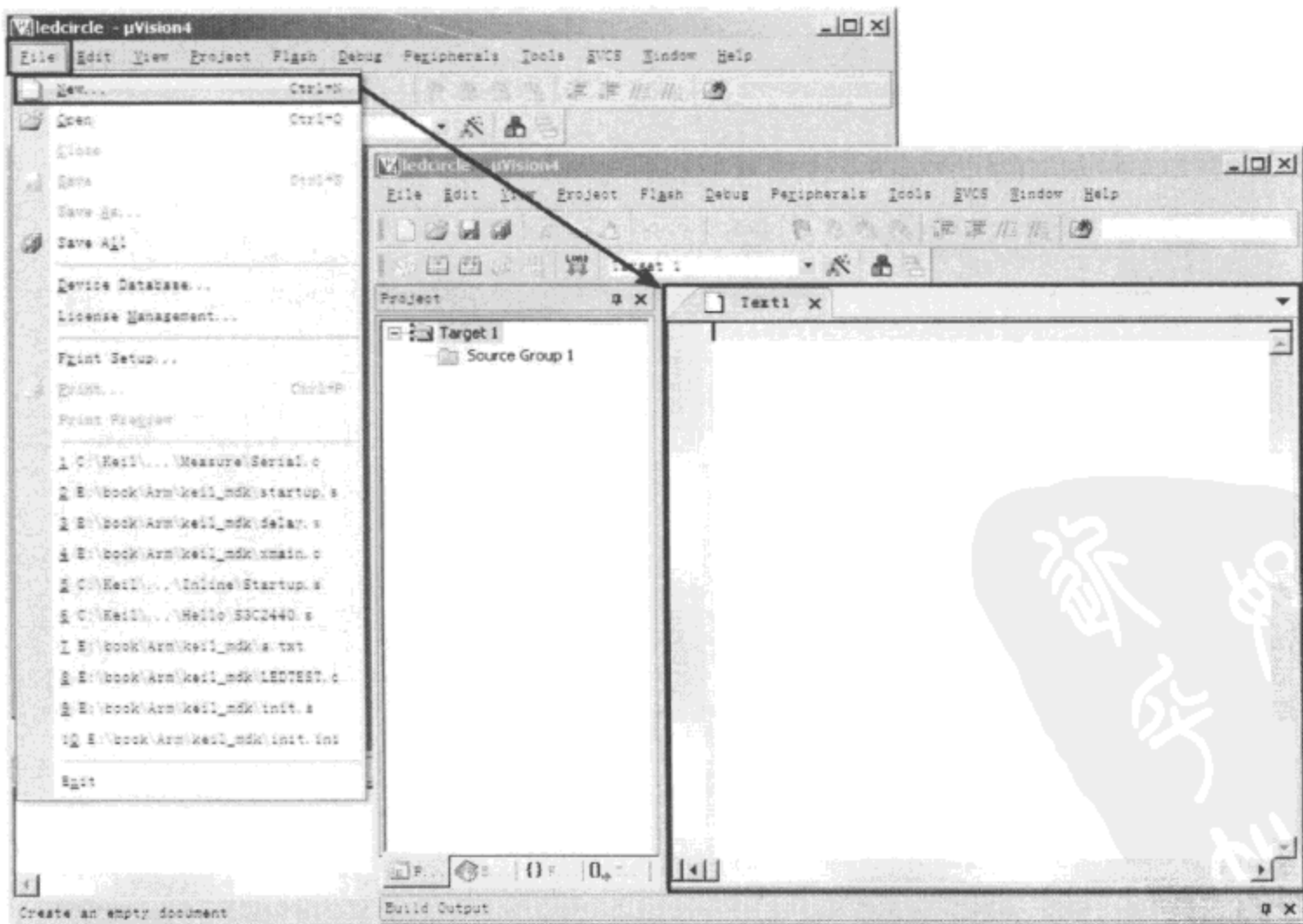


图 1-41 新建文件

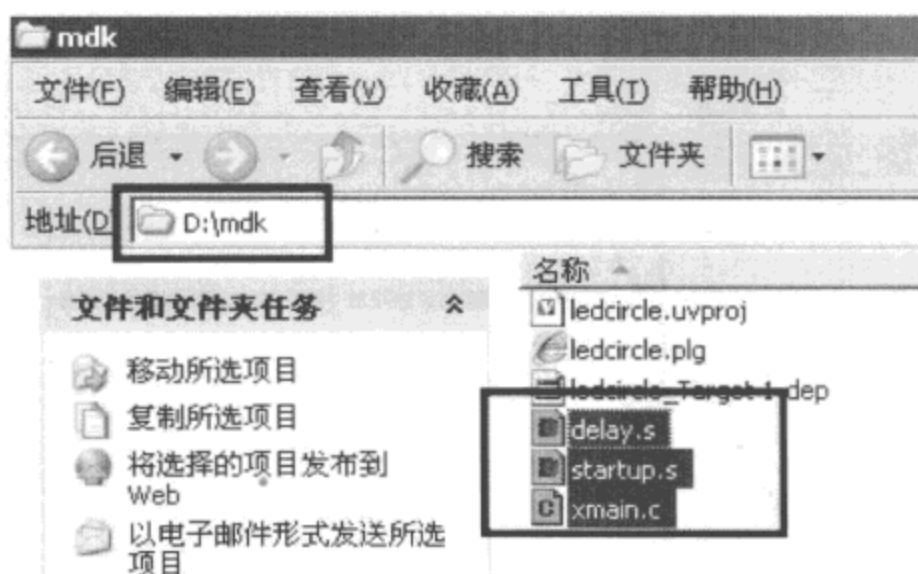


图 1-42 将示例代码复制到工程目录下

(8) 在源码组名“Source Group1”上右击弹出菜单,选择 Add Files to ‘Group Source Group1’选项,将新的复制到工程目录下的文件添加到工程中,如图 1-43、图 1-44 所示,添加完文件之后,文件就出现在工程窗口下,如图 1-45 所示。

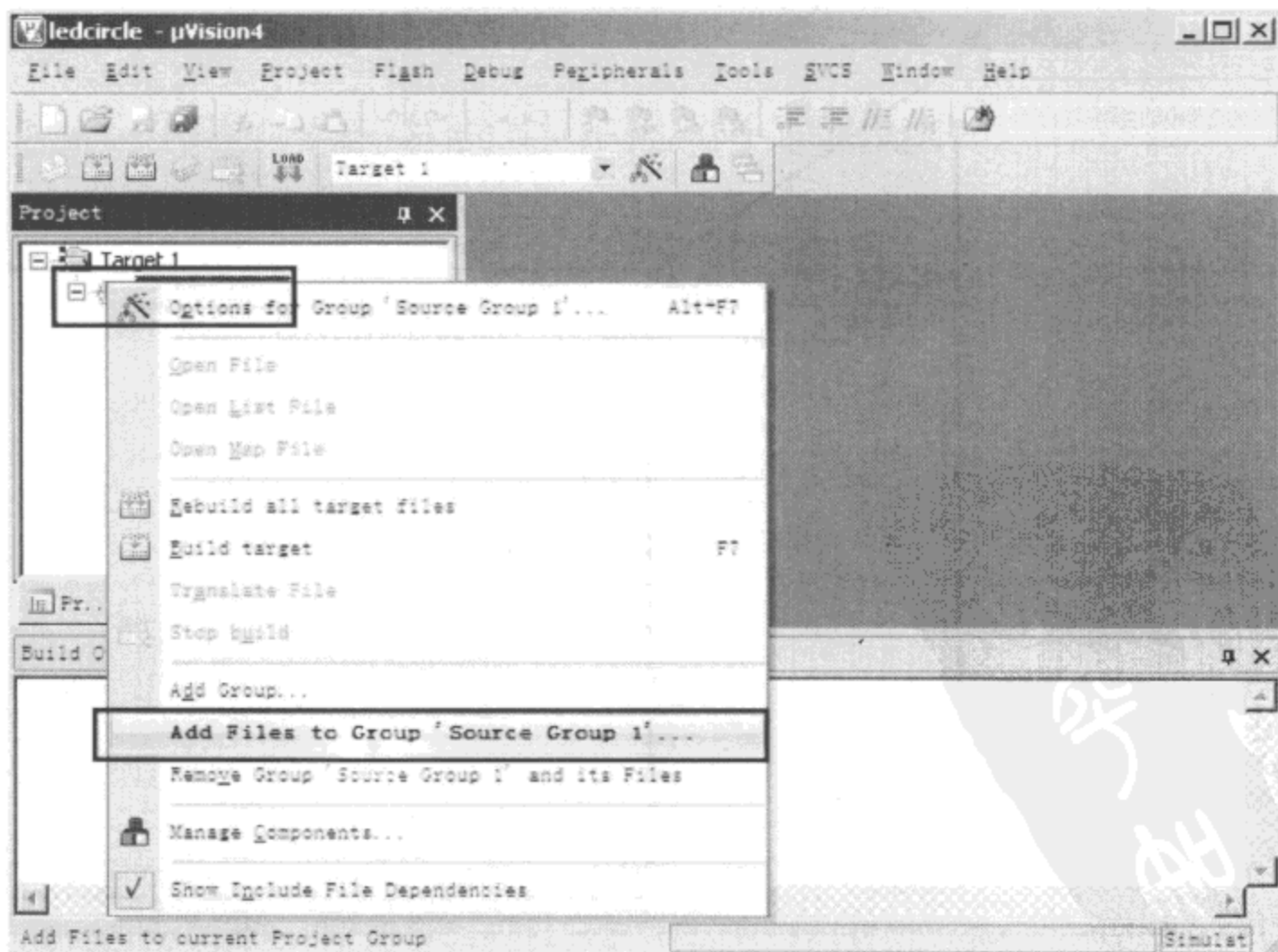


图 1-43 添加示例代码

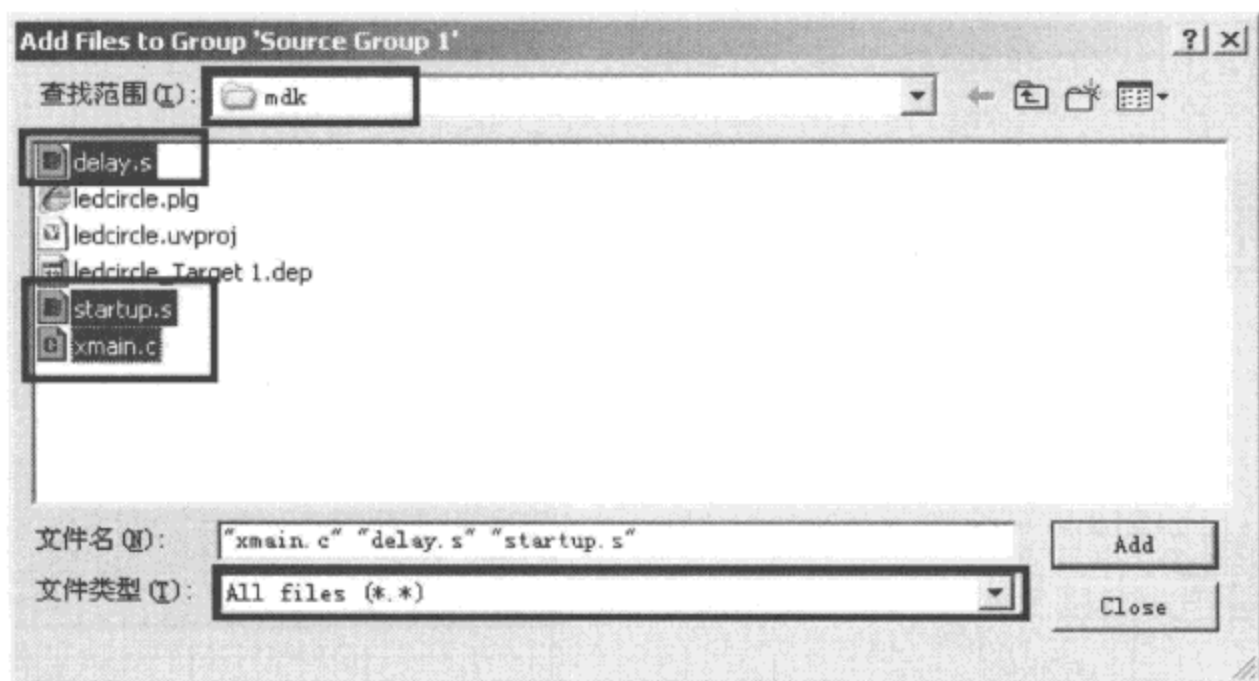


图 1-44 选择示例代码

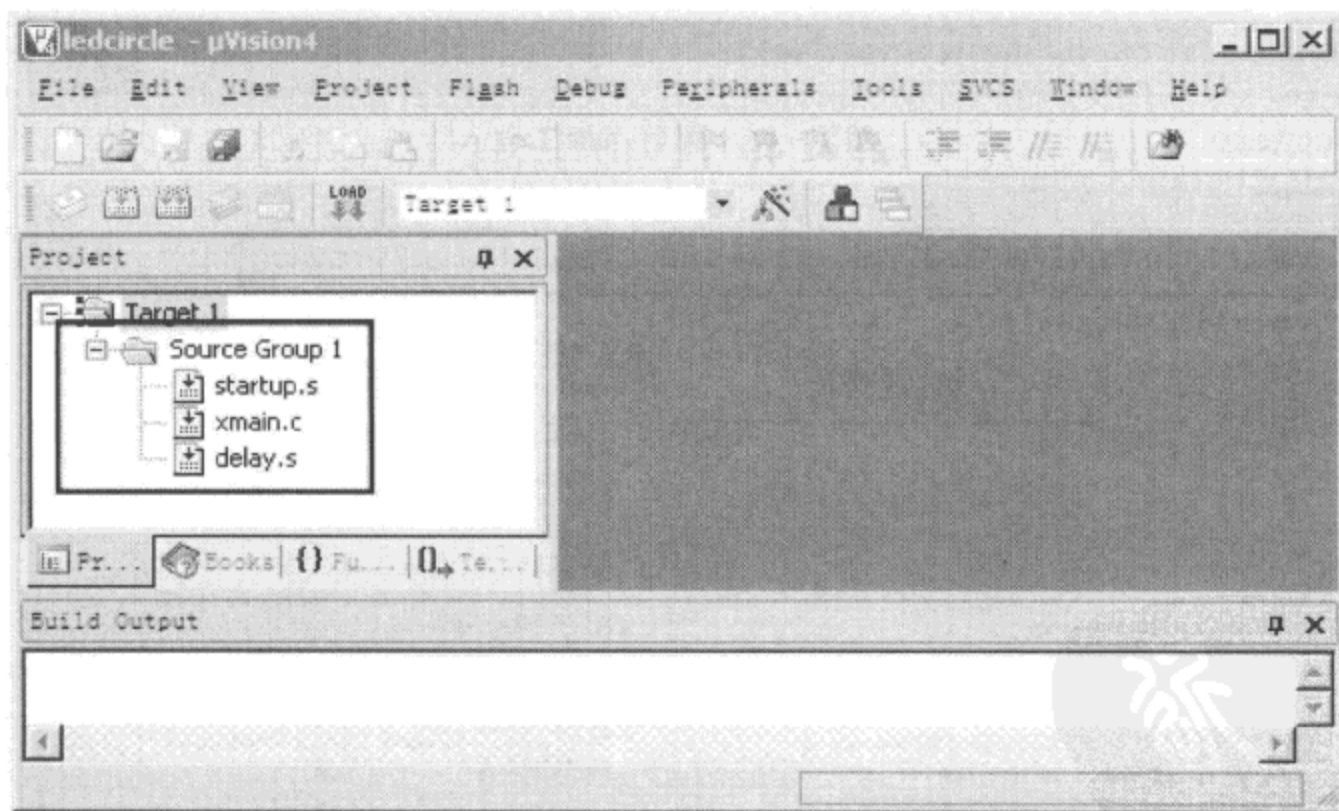


图 1-45 代码添加完毕

(9) 这时虽然文件程序源文件已经添加到了工程中,通常情况下,还不能正常编译运行,还需要对工程进行配置。右击工程目标名调出菜单,如图 1-46 所示,选择 Options for Target 'Target 1'选项,调出图 1-47 所示选项配置页面。

(10) 对工程的配置信息项较多,不过大部分不用修改,下面分别介绍每页配置信息。

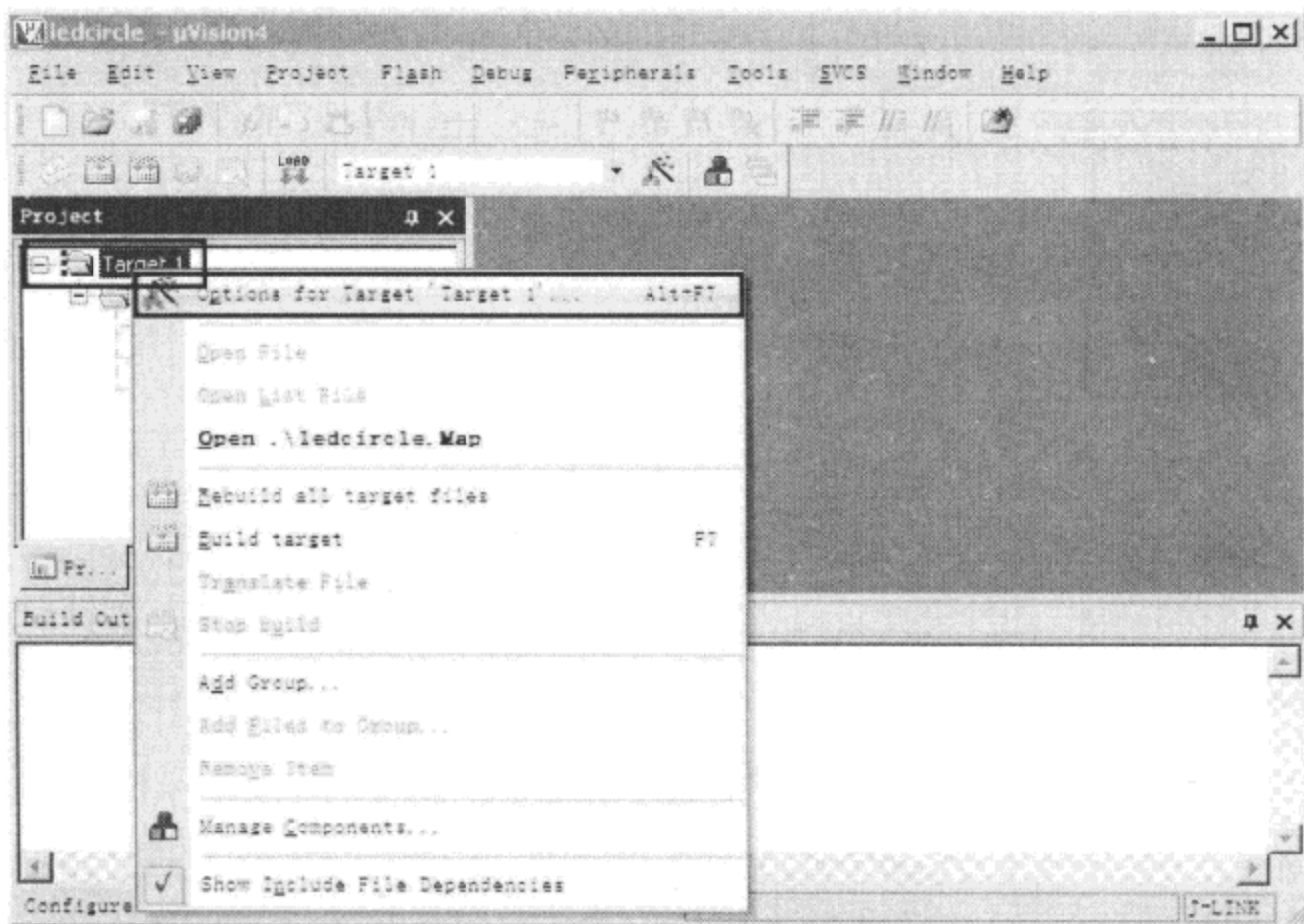


图 1-46 配置目标工程

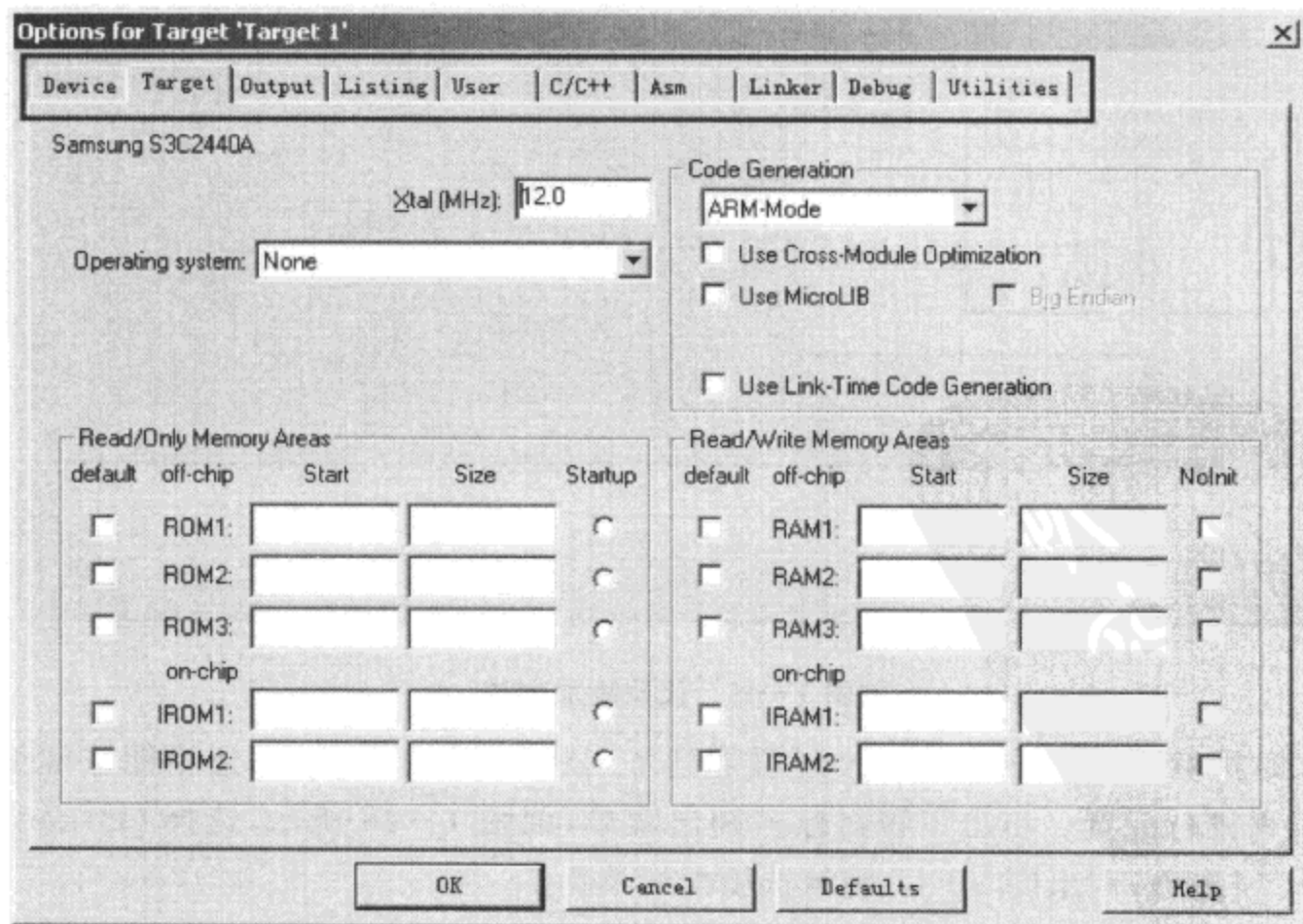


图 1-47 工程配置页面

Device:设备型号选择选项,如图 1-48 所示,如果在新建工程时已经指定,在这就不需要再指定。

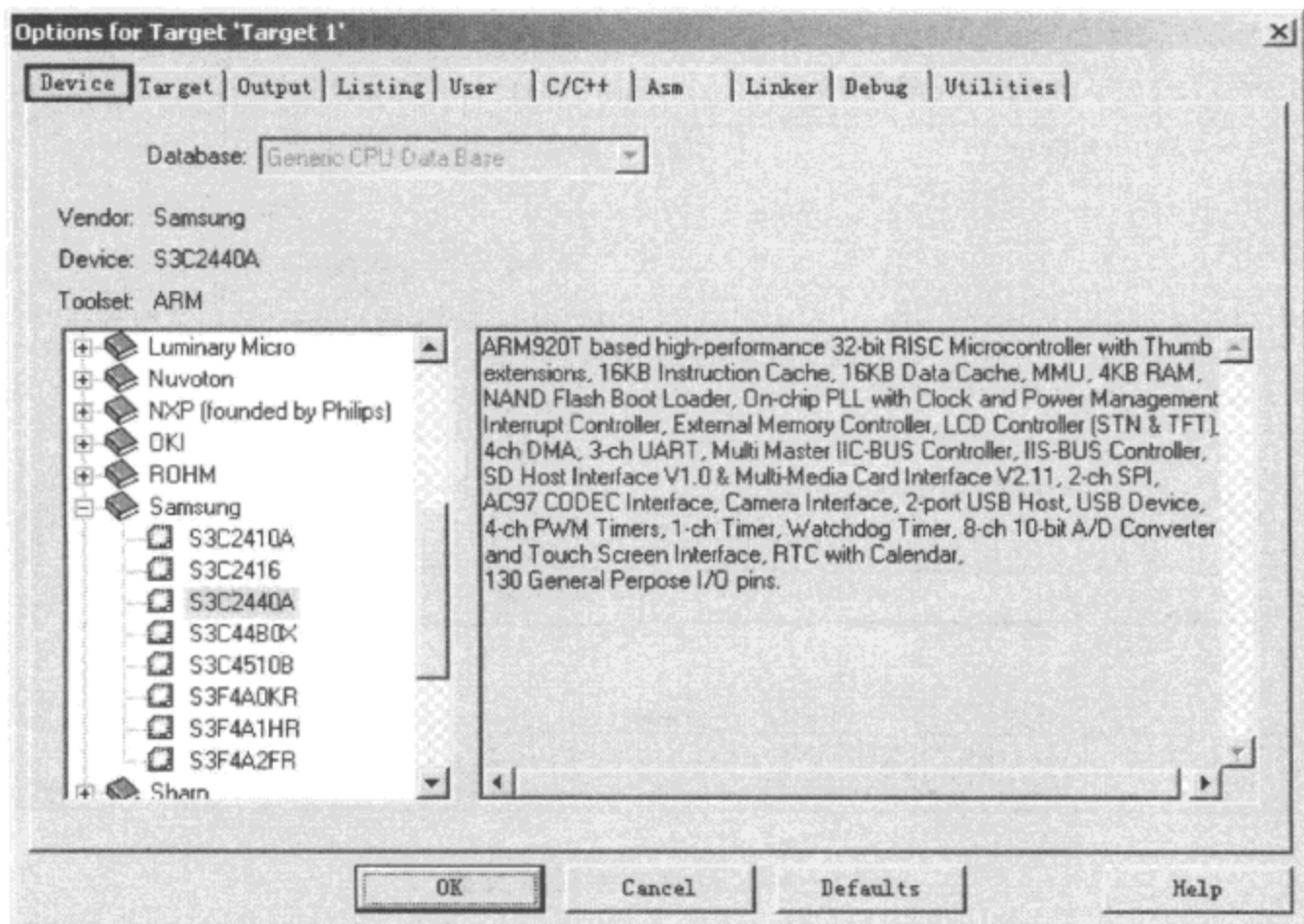


图 1-48 设备型号选择页面

Target:设置目标文件相关配置信息,主要修改 Flash 区域和内存区域,mini2440 使用地址空间为 0x0~0x200000 的 2MB 的 Nor Flash 和地址空间为 0x30000000~0x34000000 的 64MB 内存,所以设置其地址空间,如图 1-49 所示。

Output:生成目标文件配置信息页,如图 1-50 所示,Select Folder for Objects...用来指定目标映像文件目录,Name of Executable 设置生成的目标文件名,Create HEX File 选项,默认是没有选择的,要将其选择上,因为如果要将生成映像写入到芯片中,并且正常执行,则必须要生成 HEX 文件,才可正常使用。

Listing:中间生成的信息文件配置选项,如图 1-51 所示,这里保持默认即可。

User:配置在执行编译工程之前的操作,可以选择在编译工程之前先运行指定命令或脚本。如图 1-52 所示,本页保持默认即可。

C/C++, Asm:用来设置对 C/C++ 和汇编语言的编译配置信息。如图 1-53 所示,保存默认即可。

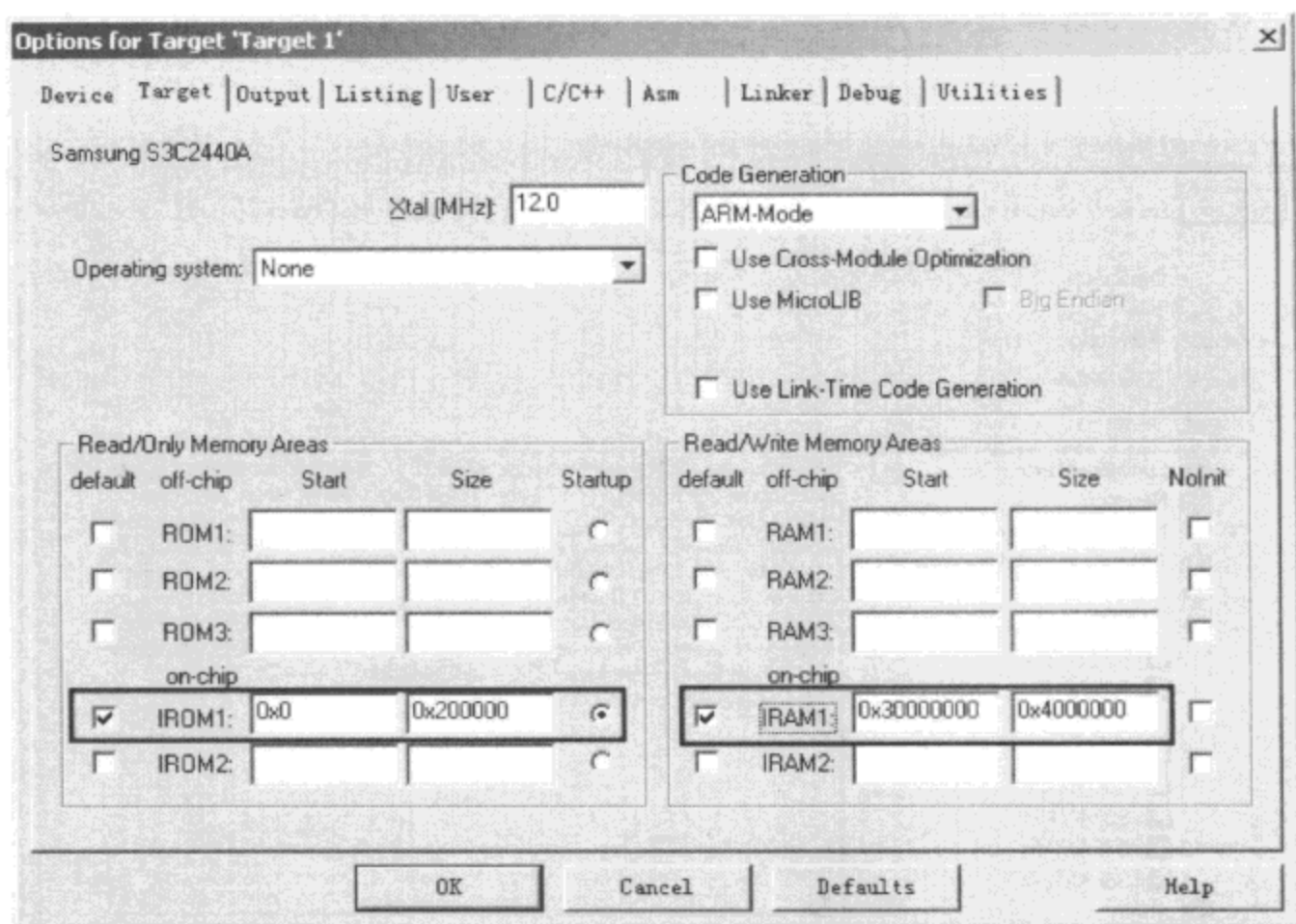


图 1-49 目标文件配置页面

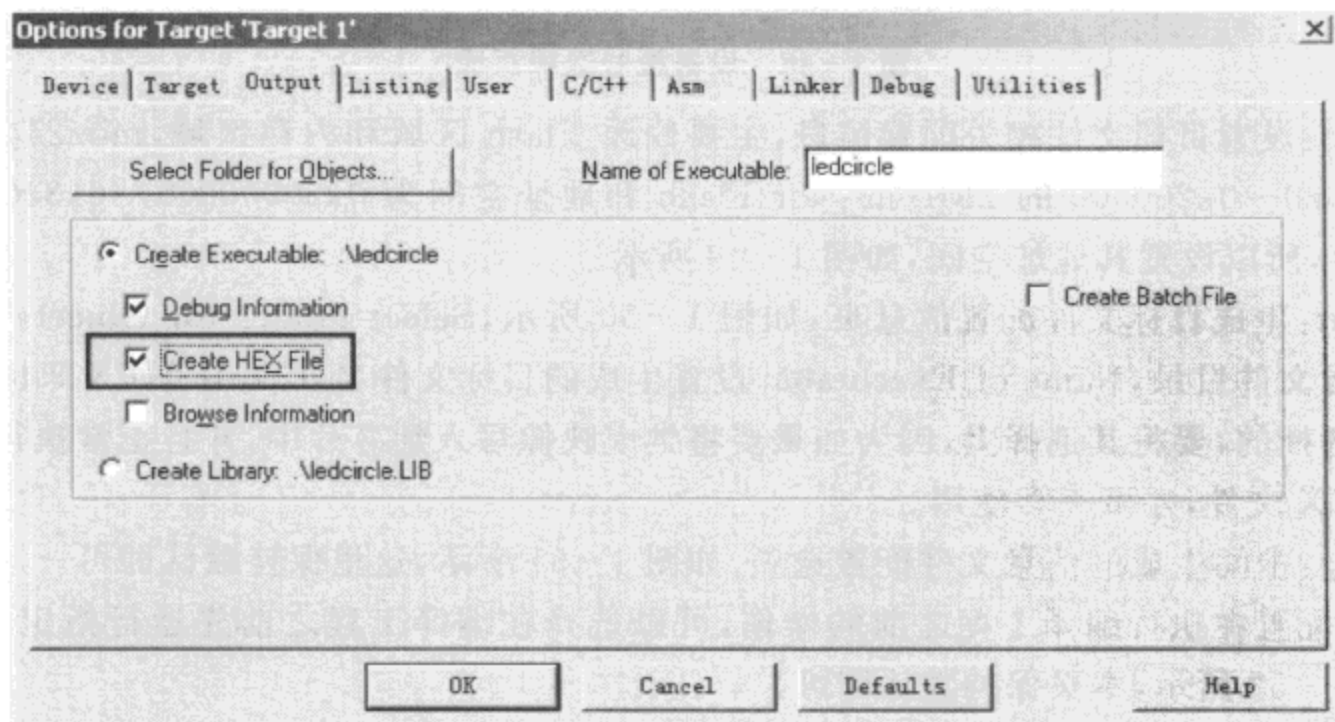


图 1-50 目标文件配置页面

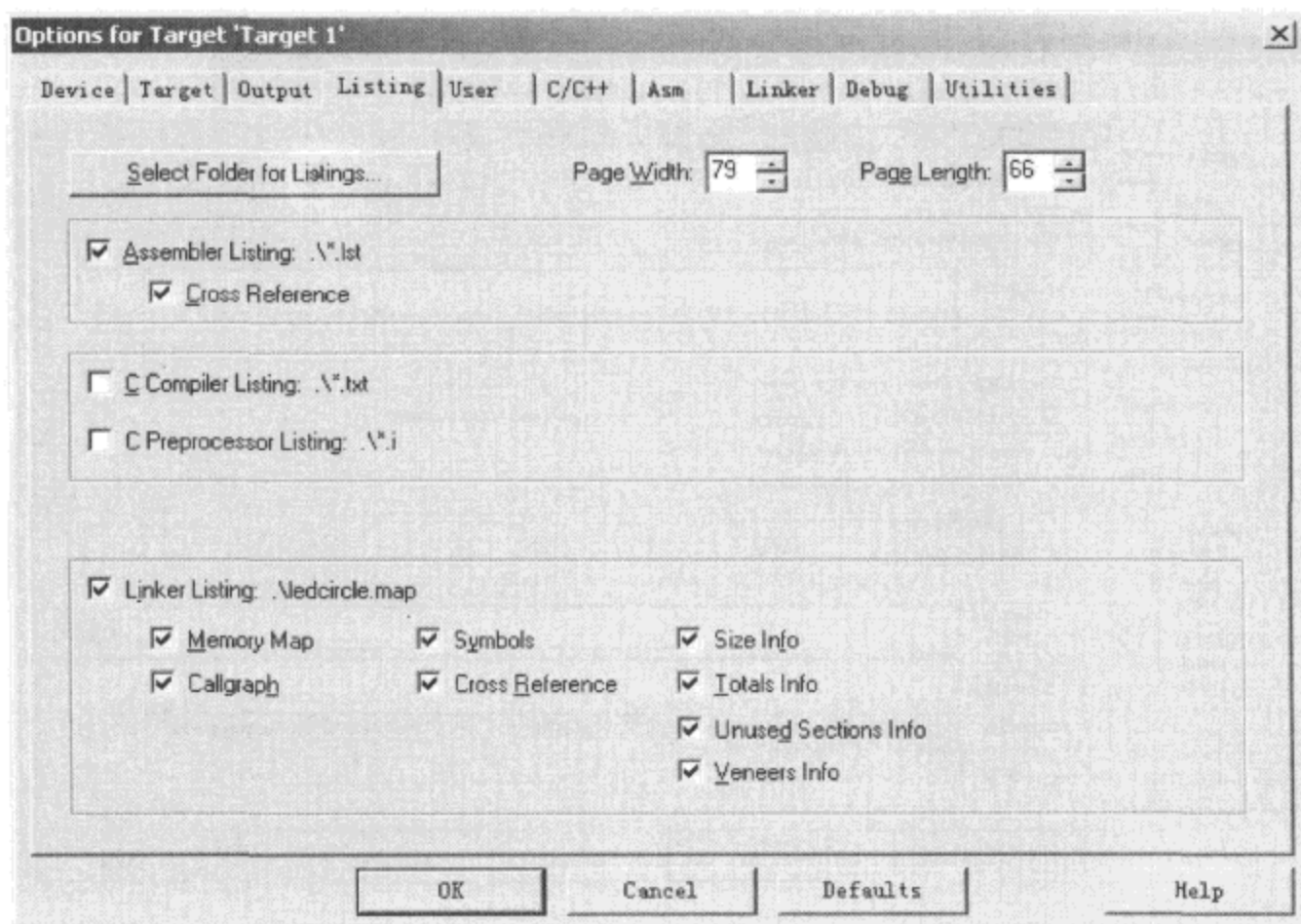


图 1-51 中间生成信息文件配置页面

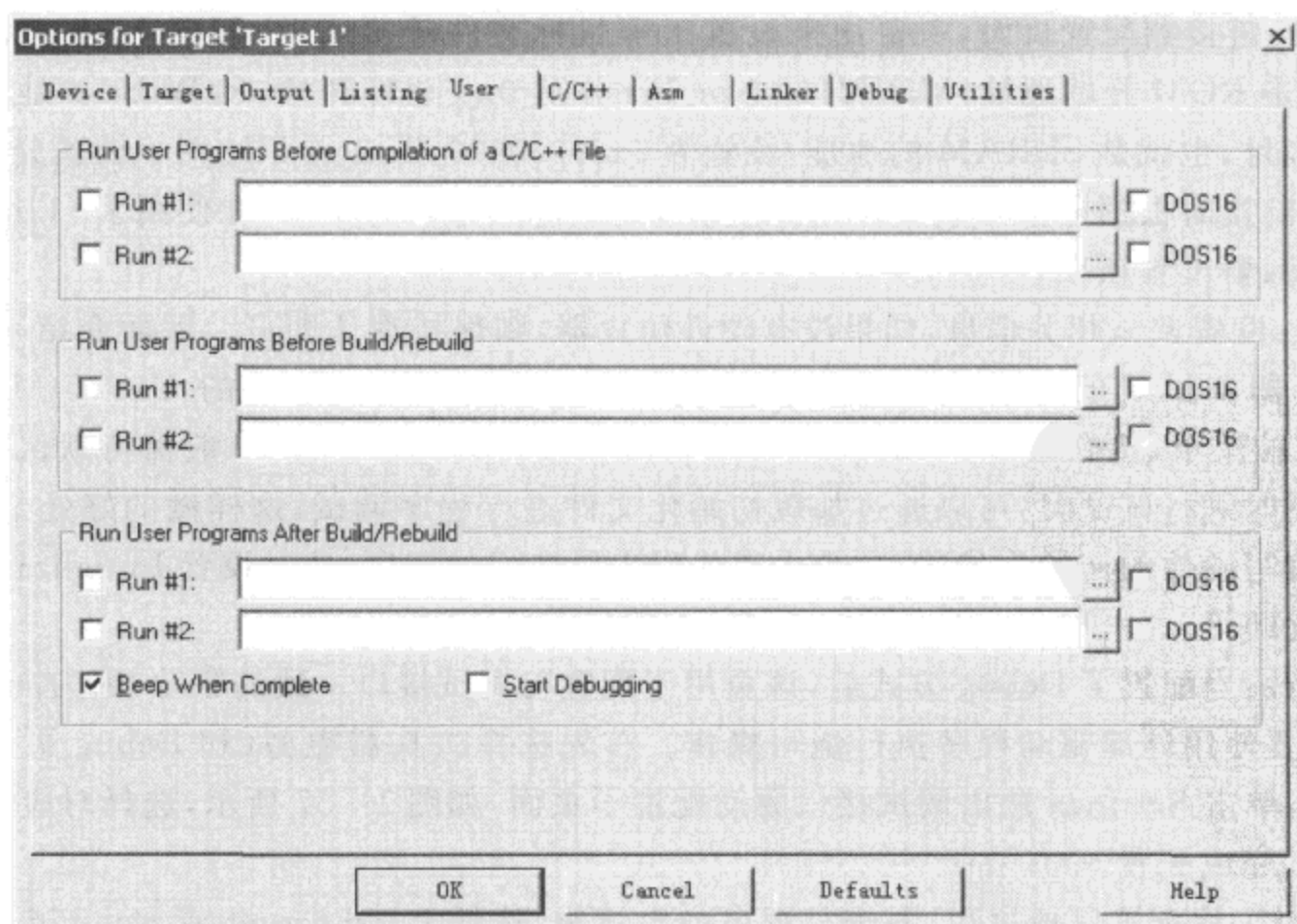


图 1-52 用户指定命令脚本配置页面

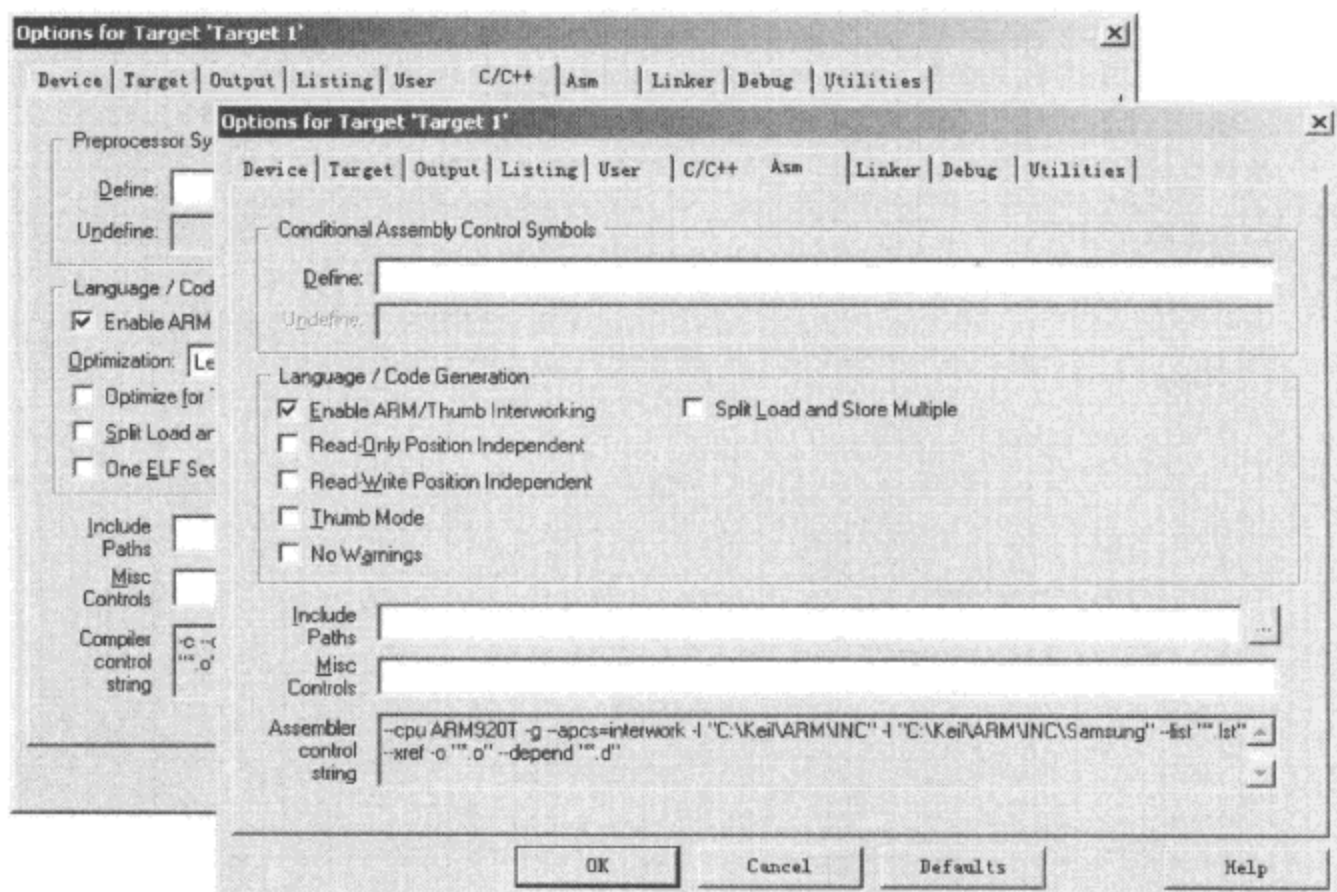


图 1-53 编译器配置页面

Linker:链接器配置页面,主要用来设置工程目标文件链接选项,如图 1-54 所示,其中 R/O Base 是 ROM 开始地址,MINI2440 Nor Flash 从 0x0 地址开始,R/W Base 是可读写存储器开始地址,也就是 SDRAM 的地址,设置为 0x30000000,Scatter File 用于设置分散加载文件选项,在链接时根据分散加载文件配置信息加载目标程序的各个段,在使用 J-Link 调试本光盘程序时,要设置该项。

Debug:配置调试相关信息,如果没有硬件仿真器,则保持默认即可。下面介绍下使用 J-LINK 仿真器调试时的配置。如图 1-55、图 1-56 所示,选择右侧的 Use: J-LINK/J-TRACE 下拉菜单,然后单击 Settings 可以对仿真器做相关详细配置,一般保持默认即可。当使用仿真器时进行调试时,可以通过加载初始化文件进行程序调试,这样做的好处是,不需要在工程中编写硬件的初始化代码,减少由于代码出错造成的问题,通过设置 Initialization File,指定初始化即可。

Utilities:当配置了 Debug 方式后,该页用于配置与调试接口一致的驱动,因为向 Flash 中烧写时,需要使用外部驱动程序执行烧写操作。首先选择仿真器类型(和 Debug 页调试方式保存一致),单击 Settings 调出调试接口驱动配置子页面,如图 1-57 所示,选择对应开发板上 Flash 型号,完成配置。

(11) 对工程配置完成之后,右击工程名弹出菜单,选择 Build target 或 Rebuild all target files 进行编译,生成目标文件了,编译信息通过下面的输入窗口显示出来,如图 1-58 所示。

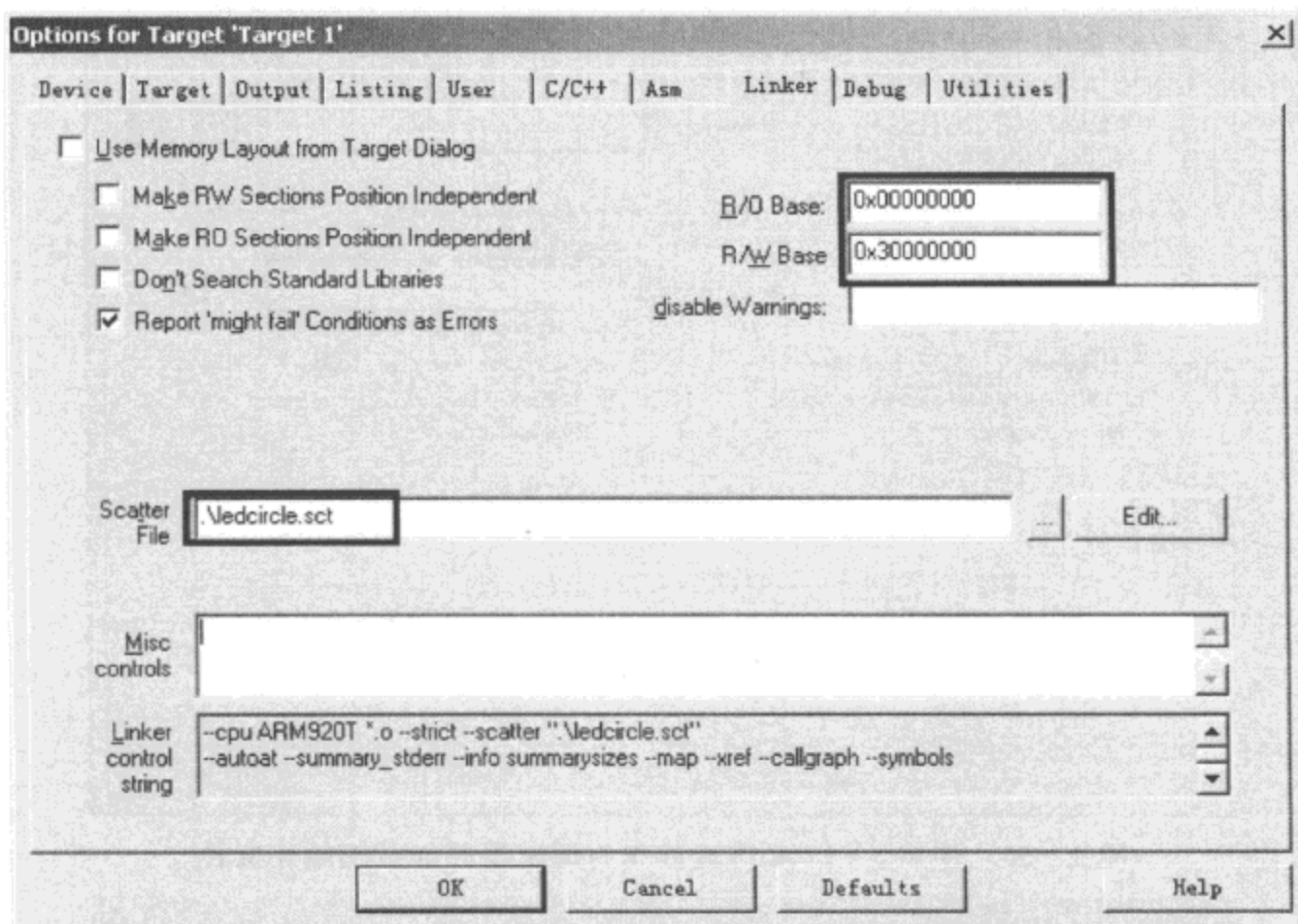


图 1-54 链接器配置选项页面

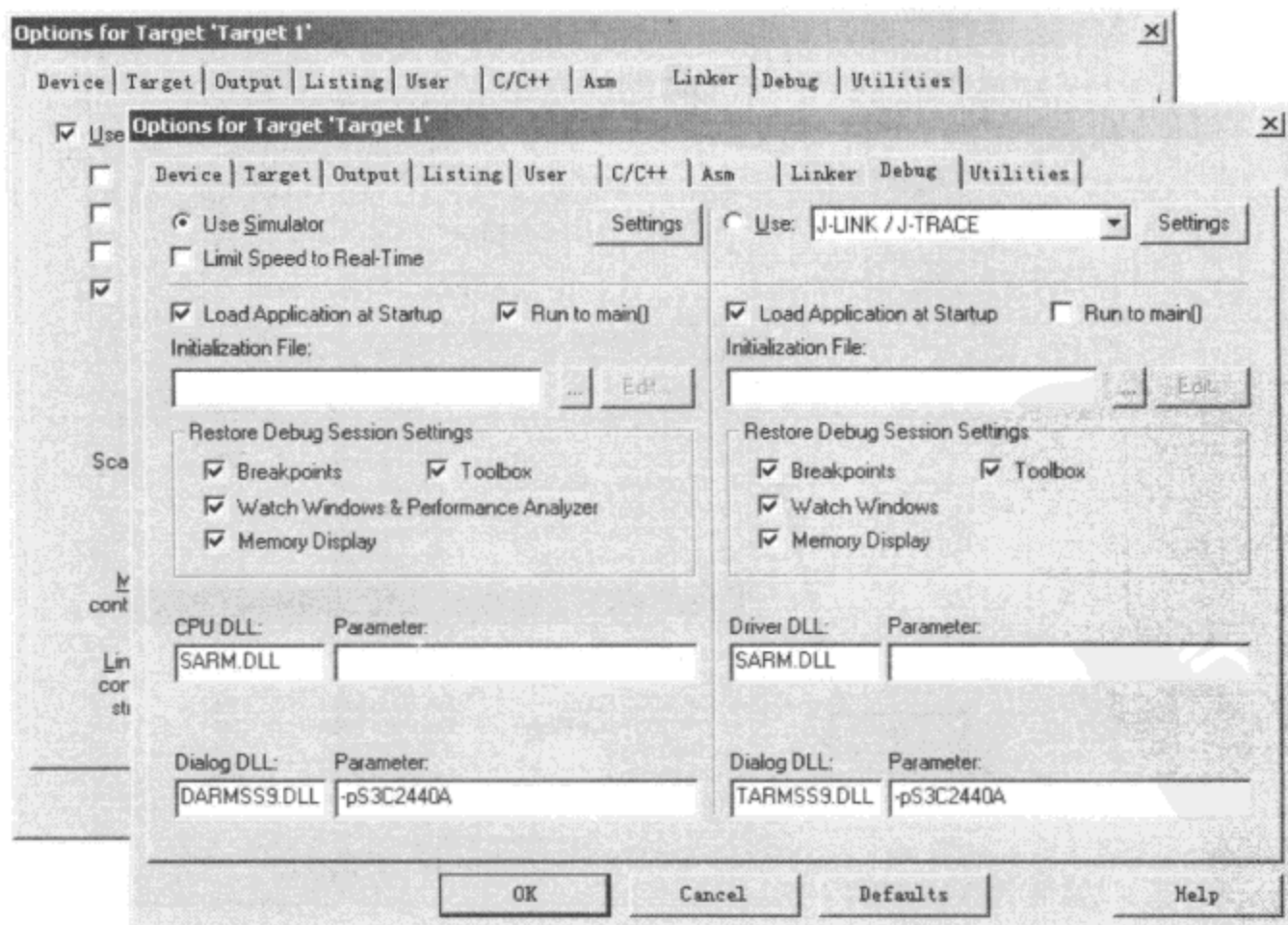


图 1-55 链接器与调试器配置页面

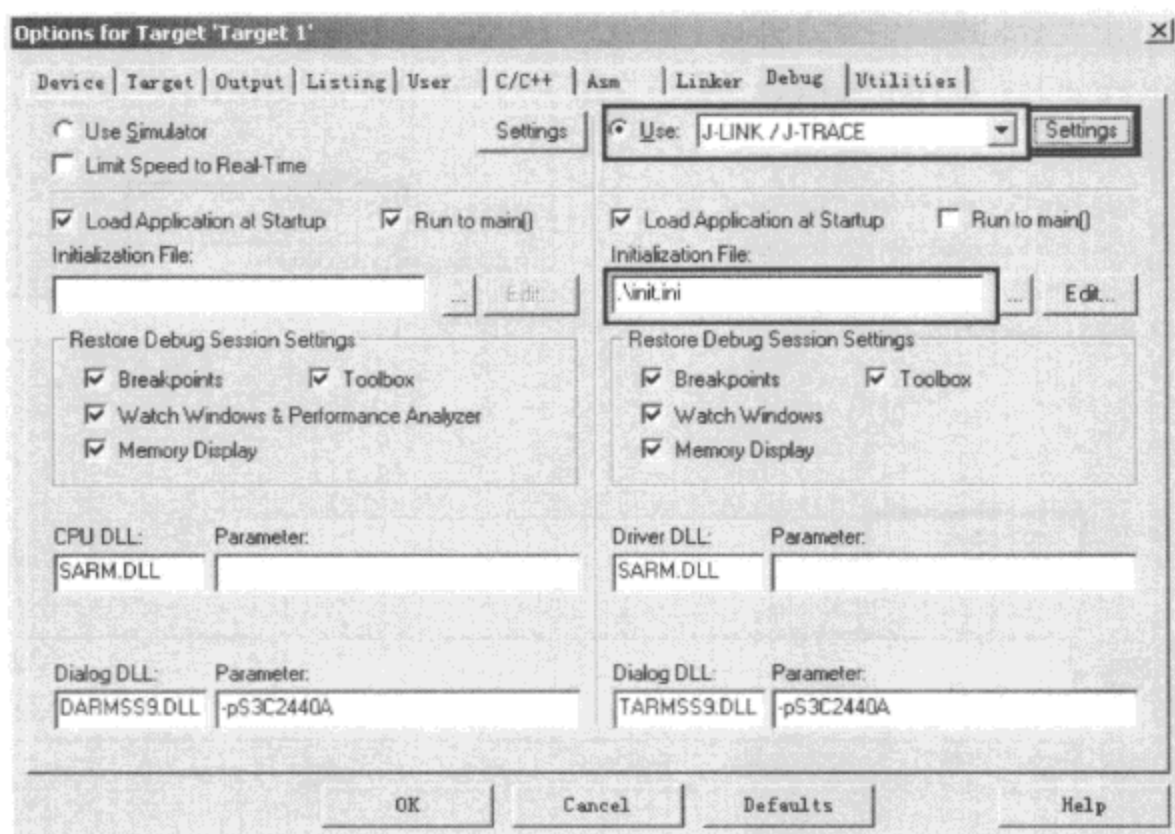


图 1-56 使用 J-Link 仿真器作为调试器并加载初始化文件

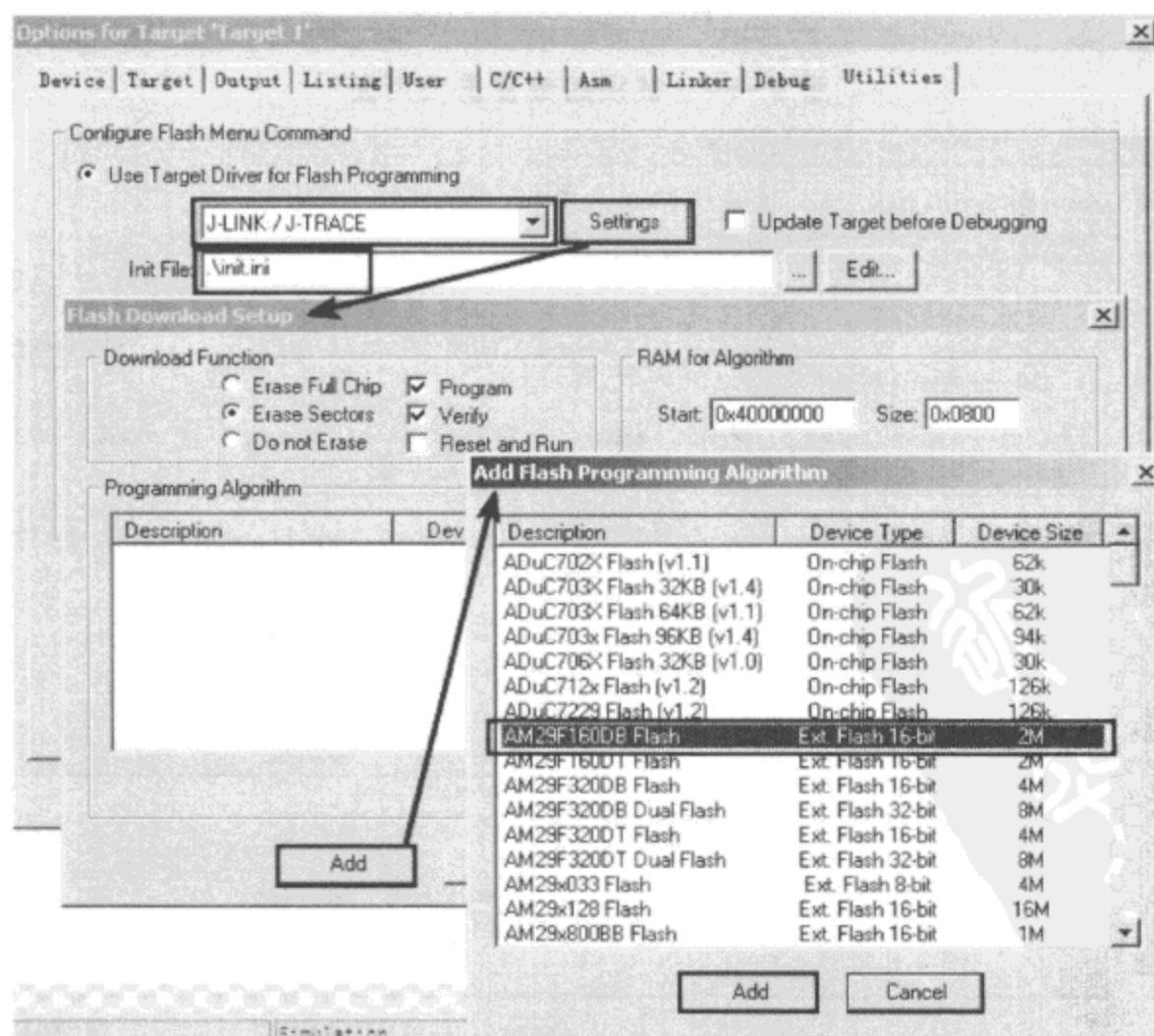


图 1-57 调试器工具配置页面

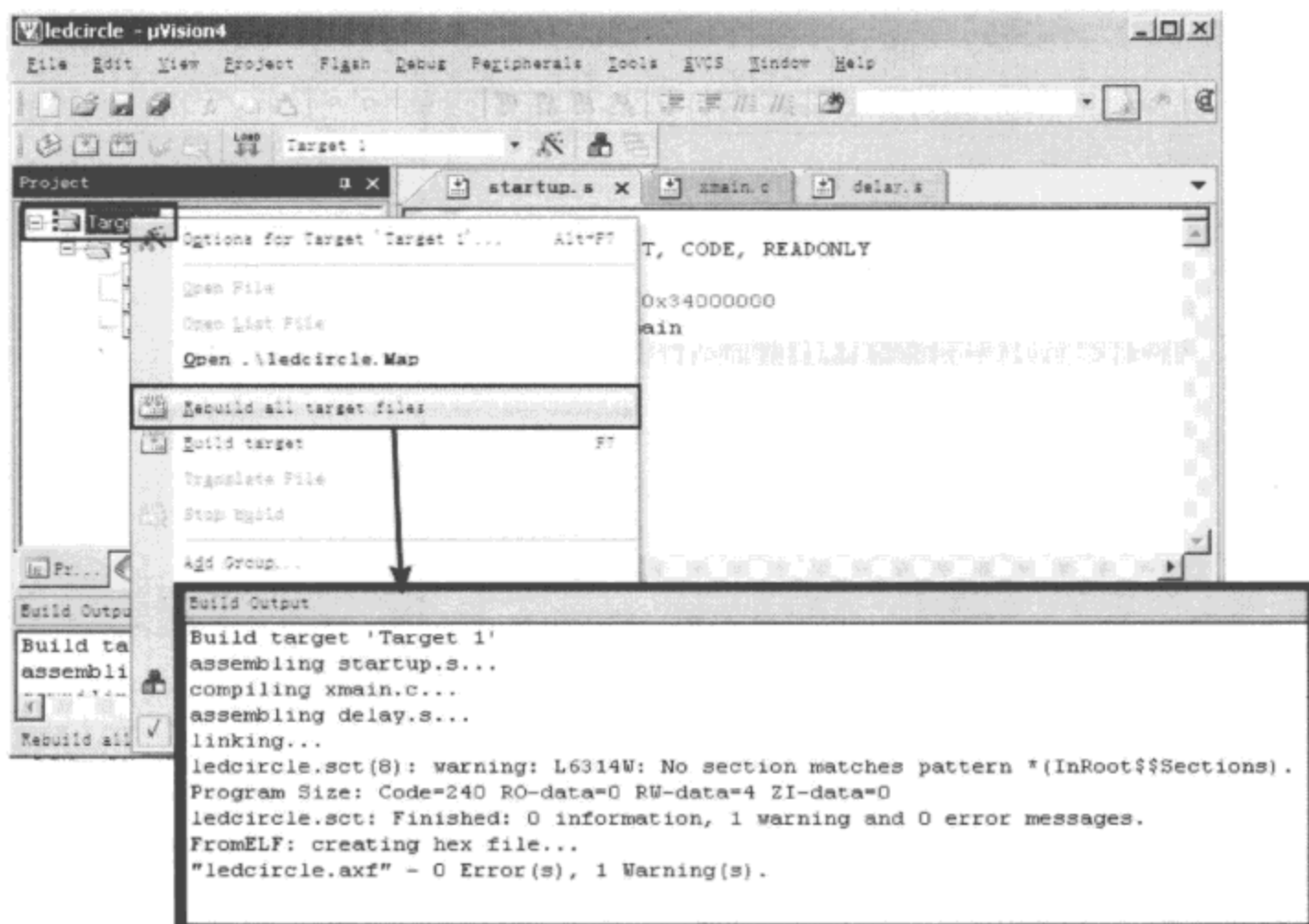


图 1-58 编译工程

1.5.2 MDK 调试裸机程序的方法与步骤

通常写完程序代码之后,进行编译,在修正了编译时出现的错误后,很多时候执行结果和预想结果不一致,这时只有通过调试才能找到最终原因。和 ADS 类似,MDK 调试时,可以完成:

- (1) 设置断点及单步运行与调试。
- (2) 查看各种模式下的 CPU 寄存器。
- (3) 查看内存内容和栈信息。
- (4) 查看 C 语言变量。

1. 进入调试模式

在对工程成功进行编译,连接之后,通过 Debug→Start/Stop Debug Session 来进入调试状态。如图 1-59 所示,进入调试状态后,界面与编辑状态相比没有明显的变化。可以看到在编辑模式下很多和调试相关的按钮都被激活了,如图 1-60 所示。

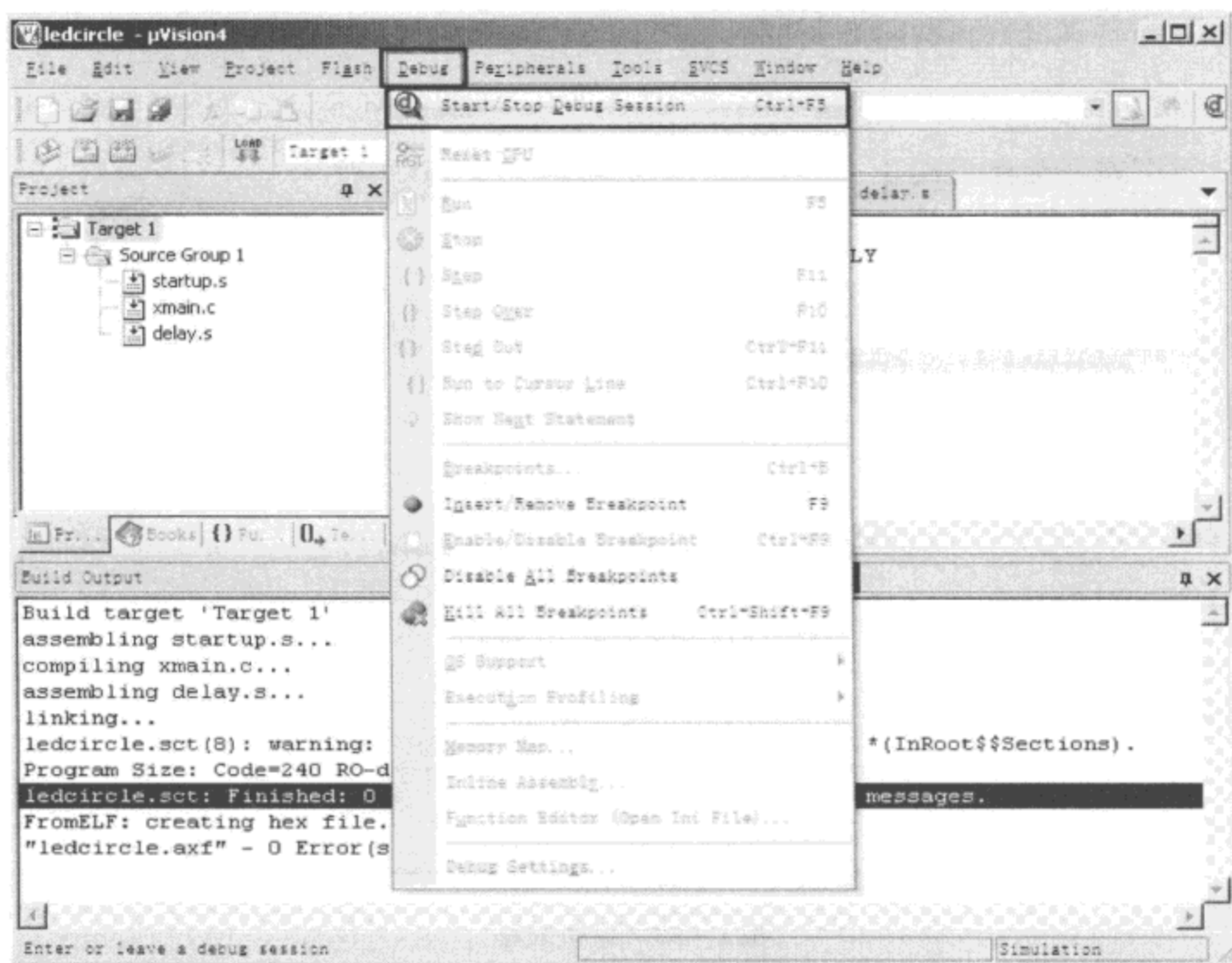


图 1-59 启动调试器

2. 设置断点(图 1-61)

打开对应代码文件,在需要让程序暂停的代码行号位置处双击,可以看到该行最前面出现红色框标记,说明断点已经设置,则当程序在执行过程中,遇到该断点即暂停。

3. 查看寄存器值、内存值、变量值等

在程序在断点处停止后,可以打开寄存器窗口,内存窗口,变量查看窗口来查看跟踪其数据变化,如图 1-62 所示。

4. 单步调试

当需要程序单步执行时,可以通过单击工具栏图标按钮或调出菜单中命令或使用快捷键来执行单步调试,如图 1-63 所示。

Reset CPU:重置 CPU,重新开始调试。

Run:向下顺序执行,直到结束或遇到断点。

Stop:程序停止运行。

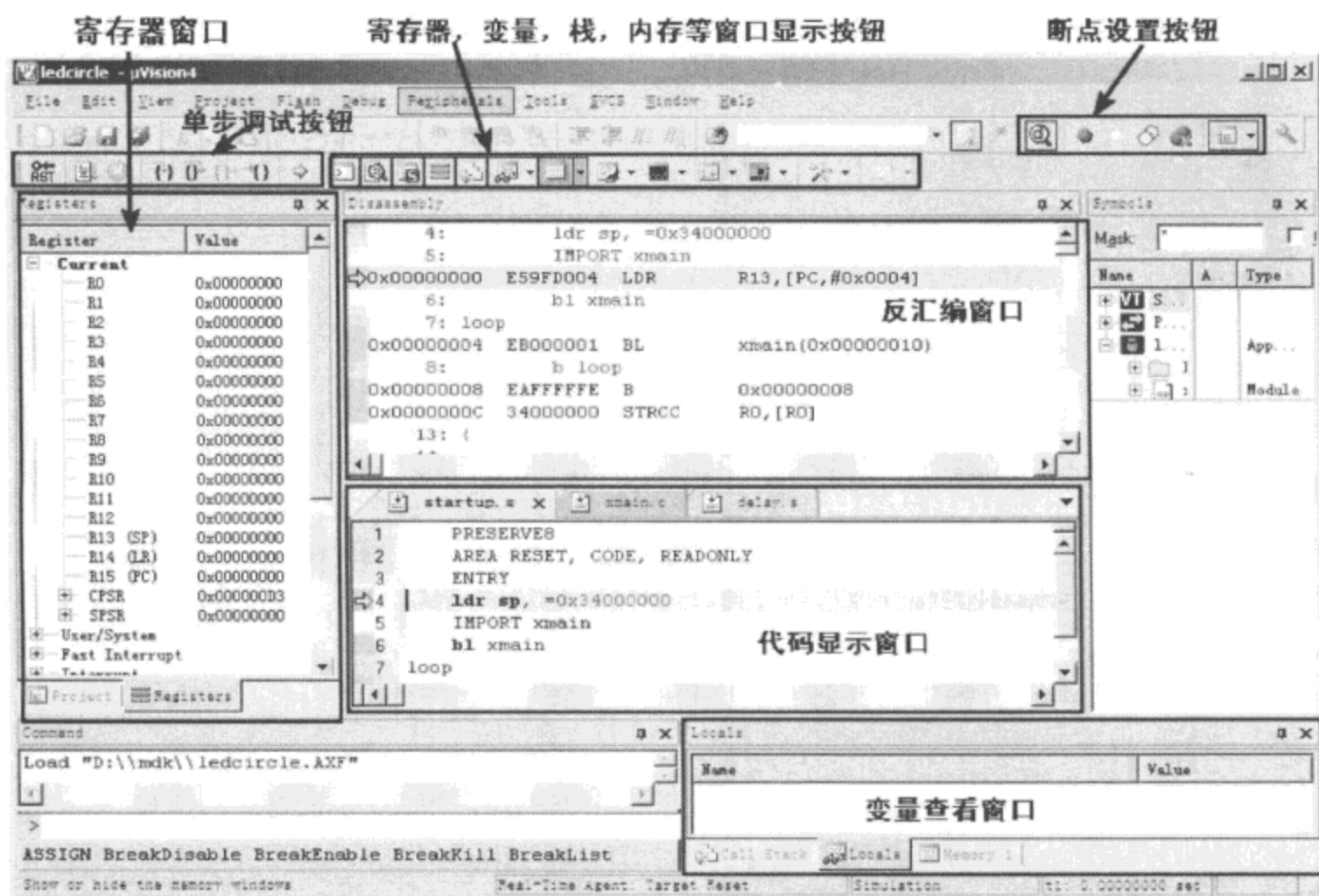


图 1-60 调试器操作界面

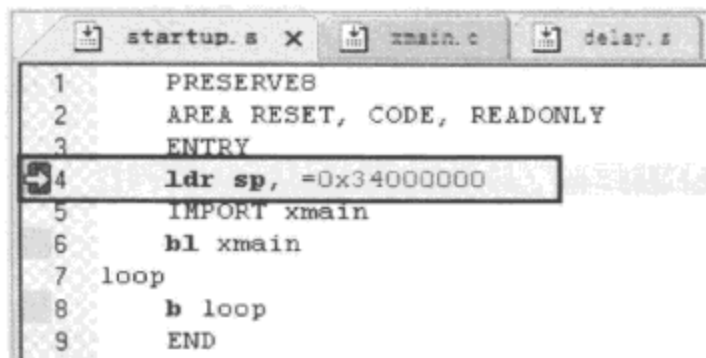


图 1-61 设置断点

Step: 单步跟踪执行, 当遇到函数时, 会进入函数内部执行。

Step Over: 过程单步执行, 当遇到函数时, 会跳过函数执行。

Step Out: 跳出当前函数。

Run to Cursor Line: 运行到游标位置。

和 ADS 一样, 在单步调试过程中, 可以随时查看寄存器、内存、变量、栈中数据的值。

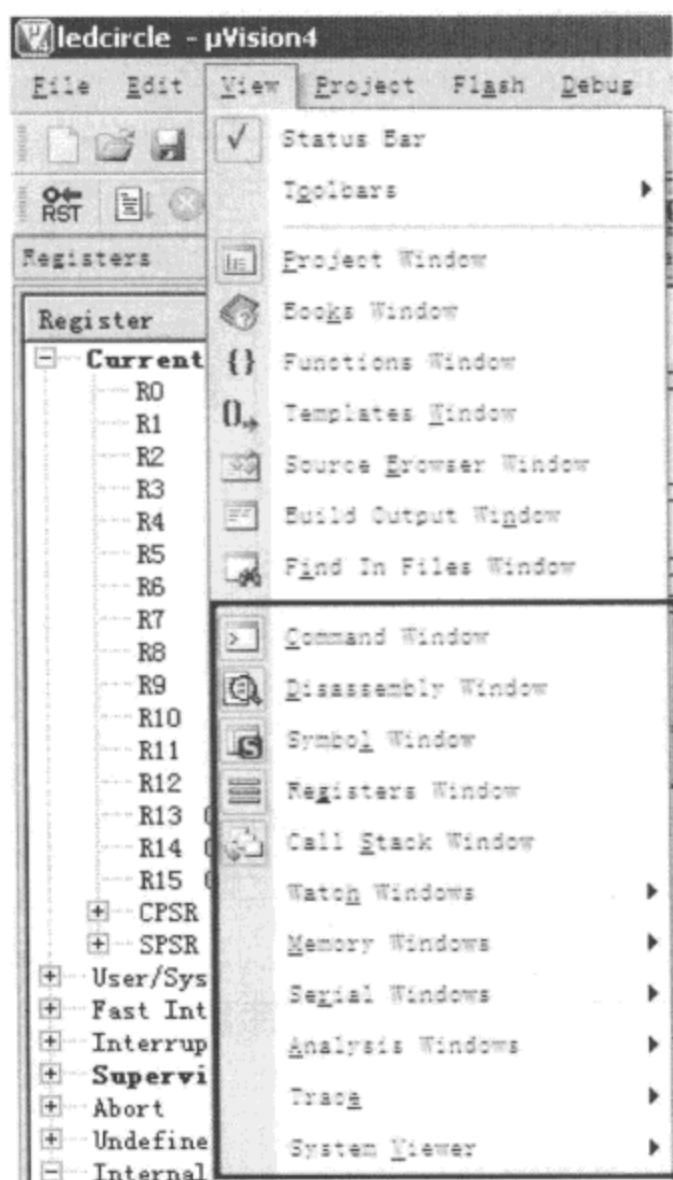


图 1-62 内存、变量、栈信息、命令行反汇编窗口控制按钮

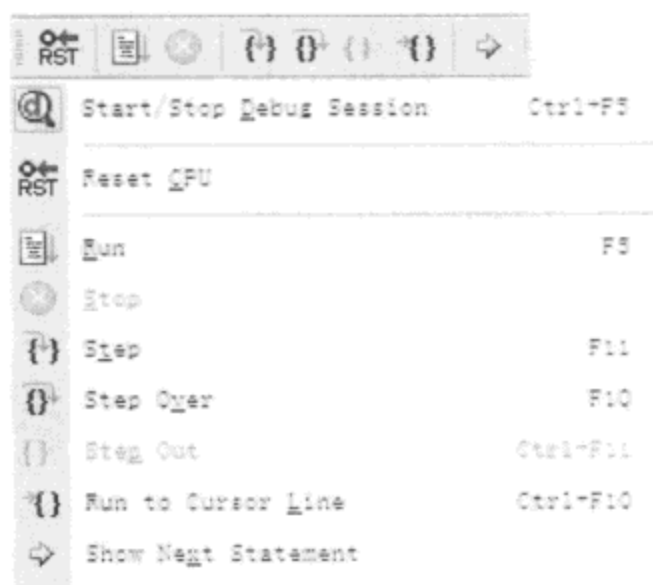


图 1-63 单步调试控制工具按钮和菜单

1.6 其他常见寻址模式与常见指令

现在我们已经掌握了所有知识,可以编写简单的 ARM 汇编程序,但如果要编写较为复杂的 ARM 程序,就必须掌握更多的寻址模式和指令,这就是本节的重点所在。

我们在“基本寻址模式与基本指令”中学习了最常用的 3 种寻址方式。下面介绍其他寻址方式。

1.6.1 其他常见寻址模式

1. 基址寻址

基址寻址就是将基址寄存器的内容与指令中给出的偏移量相加,形成操作数的有效地址。基址寻址用于访问基址附近的存储单元,常用于查表、数组操作、功能部件寄存器访问等。基

址寻址指令举例如下:

```
LDR R1,[R2,#0x0C]
```

R2 的值+0x0C 形成内存地址,读取内存中该地址上的内容,放入 R1。

其他额外需要了解的内容:

- 零偏移。如:LDR R0,[R1]
- 前索引偏移。如:LDR R0,[R1,#0x04]!,表示将 R1 的值加上 4 后作为内存地址,并且指令执行结束时,R1 本身的值也要加 4。这里! 表示要回写。
- 程序相对偏移。

如 LDR R0,label,表示将标号 label 所代表的地址处存放的内容放入 R0,相当于 LDR R0,[PC,#某个常数]。

```
ldr r0, label
```

```
.....
```

```
label DCD 0x12345678
```

- 后索引偏移。如 LDR R0,[R1],#0x04,表示将 R1 的值作为内存地址,并且指令执行结束时,R1 本身的值要加 4。

2. 多寄存器寻址

多寄存器寻址一次可传送几个寄存器值,允许一条指令传送 16 个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下:

LDMIA R1!,{R2-R4,R6},它是 ldr 的多寄存器版本,将内存中连续存放的 4 个字加载到寄存器 R2,R3,R4,R6 中,如图 1-64、图 1-65 所示。

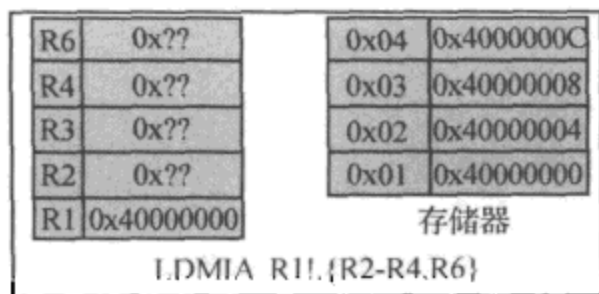


图 1-64 LDMIA 指令执行前

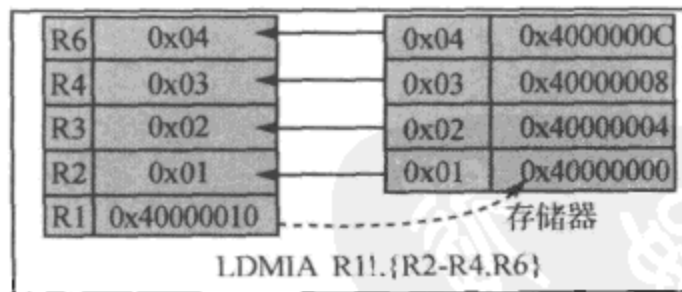


图 1-65 LDMIA 指令执行后

两点说明:

- (1) R1! 中的! 号表示在指令执行完成后,要改变(回写)基址寄存器(R1)的值。
- (2) 寄存器列表{R2-R4,R6}中的顺序并不要紧。最终寄存器与内存地址的对应关系是:编号小的寄存器与内存的低地址相对应。

两点问题:

- (1) 为什么内存起地址是 0x40000000,而不是 0x40000004;

(2) 为什么内存地址是从 $0x40000000 \sim 0x4000000C$, 而不是从 $0x3FFFFFF4 \sim 0x40000000$ 。

要解释上面两个问题, 其实也很简单。其实多寄存加载指令 `ldm` 总共有 4 个: `ldmia`、`ldmib`、`ldmda`、`ldmdb`。ia 的意思是 increase after, ib 的意思是 increase before, da 的意思是 decrease after, db 的意思是 decrease before。以 `LDMIA R1!, {R2-R4, R6}` 为例子, 这里的 ia 是指办事(将内存中的数加载到寄存器)之后增加基址寄存器(R1)的值。这条指令的执行过程从逻辑上看, 如下:

(1) 先办事: 将 R1 的值($0x40000000$)作为内存地址, 到该地址处取得数($0x01$), 加载到寄存器 R2 中。

(2) 后增加: 将 R1 的值从 $0x40000000$ 增加为 $0x40000004$, 再重复上面的操作 3 次, 分别将内存中的数 $0x02$ 、 $0x03$ 、 $0x04$ 放到寄存器中 R3、R4、R6 中, 最后 R1 的值变为 $0x40000010$ 。

这个例子中, 如果将 `ldmia` 改为 `ldmib`, 则 R2、R3、R4、R6 中存放的是 $0x02$ 、 $0x03$ 、 $0x04$ 、内存 $0x40000010$ 处的内容, 最后 R1 的值为 $0x40000010$ 。

除了 4 条多寄存器加载指令外, 还有 4 条类似的多寄存器存储指令, 分别是 `stria`、`strib`、`strda`、`strdb`。

3. 堆栈寻址

由于 ARM 指令集没有专门的出栈和入栈指令, 所以 ARM 汇编程序是采用 SP 作为栈指针, 以 `stm` 指令完成入栈操作, 以 `ldm` 指令完成出栈操作。

以入栈后 SP 的值是增加还是减少为依据, 可将堆栈类型划分为递增堆栈(向上生长)和递减堆栈(向下生长), 如图 1-66 所示。

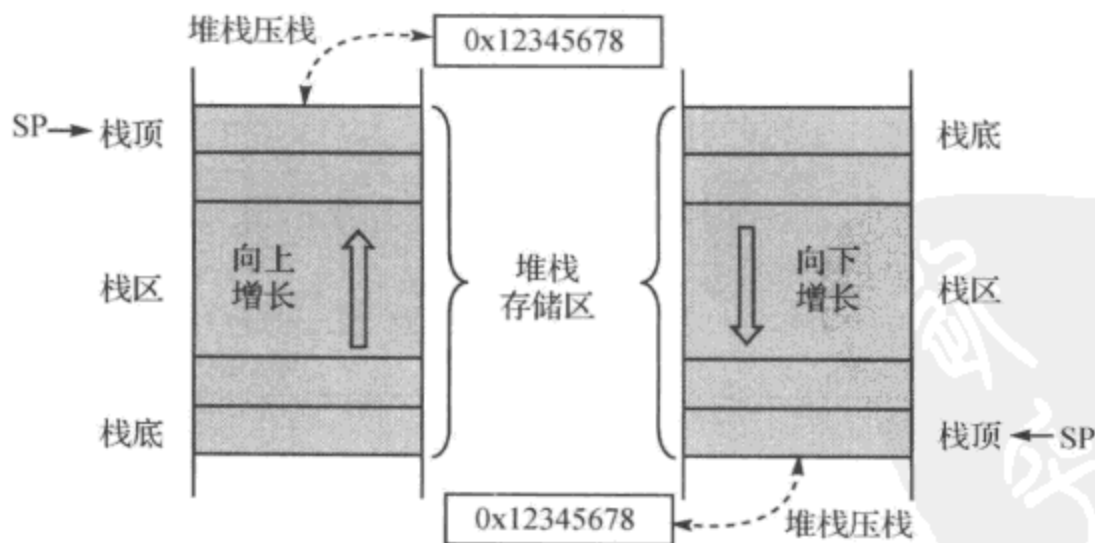


图 1-66 递增堆栈与递减堆栈

以 SP 所指向的内存处存放的是栈顶元素还是下一次要入栈的元素, 可将堆栈类型划分为满堆栈和空堆栈, 如图 1-67 所示。

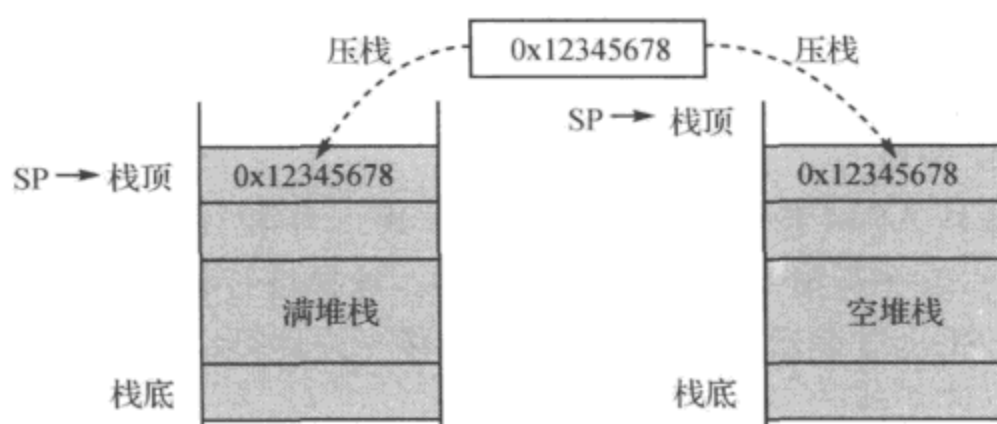


图 1-67 满堆栈与空堆栈

那么当堆栈类型为空递减堆栈时候,入栈操作应该使用什么指令?出栈操作应该使用什么指令?进一步,如果堆栈类型为空递增、满递增、满递减堆栈,又将如何呢?如表 1-8 所列。如果不看下面的答案,可能会感叹 ARM 指令集的设计者太不为的程序员考虑了,给了程序员本不应该承担的负荷。但事实上正相反,ARM 指令集的设计者充分理解了作为程序员的苦恼,如表 1-8 所列。

表 1-8 堆栈类型与堆栈操作

| 数据块传送(存储) | 堆栈操作(入栈) | 说 明 |
|--------------|--------------|-----|
| STMDA | STMED | 空递减 |
| STMIA | STMEA | 空递增 |
| STMDB | STMFD | 满递减 |
| STMIB | STMFA | 满递增 |
| 数据块传送(加载) | 堆栈操作(出栈) | 说 明 |
| LMDA | LDMFA | 满递增 |
| LDMIA | LDMFD | 满递减 |
| LDMDB | LDMEA | 空递增 |
| LDMIB | LDMED | 空递减 |

这张表的第一、三列回答了前面的问题。而第二列则体现了 ARM 指令集的设计者对作为程序员的充分体贴。第二列中的 ED、EA、FD、FA 分别表示 empty descend(空递减)、empty ascend(空递增)、full descend(满递减)、full ascend(满递增),其含义是说,如果采用的是空递减(空递增、满递减、满递增)堆栈的话,入栈操作则使用指令 STMED(STMEA、STMFD、STMFA),出栈操作则使用指令 LDMED(LDMEA、LDMFD、LDMFA)。从此再也不会为应该使用 ia、ib、da 还是 db 来实现出、入栈操作而苦恼了。

STMED、STMEA、STMFD、STMFA 和 LDMED、LDMEA、LDMFD、LDMFA 就是所谓

第1章 ARM 汇编编程基础

的堆栈寻址指令。由此可见:为了对程序员体贴入微,ARM 指令集的设计者设计了堆栈寻址指令,其实质就是多寄存寻址指令的快捷方式。

4. 寄存器移位寻址

寄存器移位寻址是 ARM 指令集特有的寻址方式。当第二个操作数是寄存器移位方式时,第二个寄存器操作数在与第一个操作数结合之前,选择进行移位操作。例如:

MOV R0,R2,LSL #3 表示将 R2 的值逻辑左移 3 位,结果放入 R0,即是 $R0 = R2 \times 8$ 。

移位的方式有以下几种:

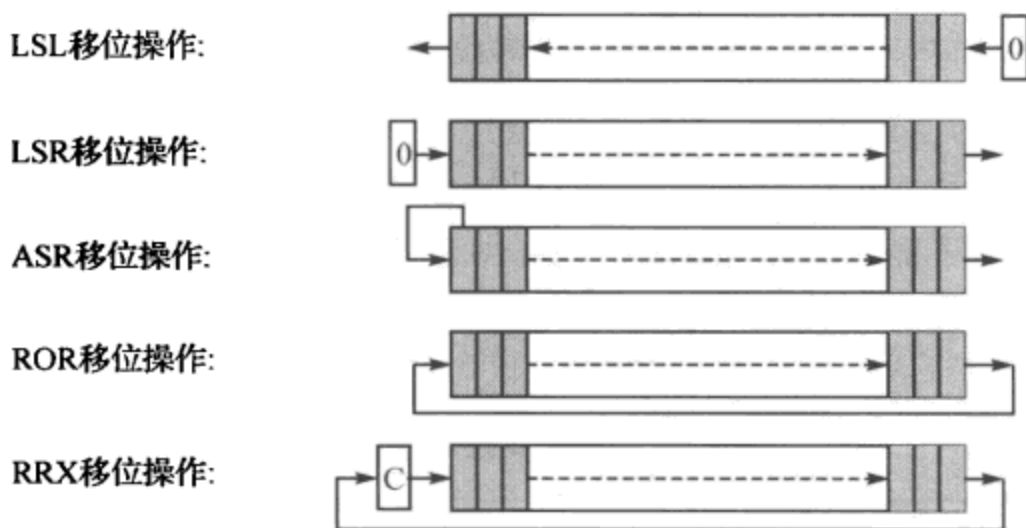


图 1-68 移位操作类型

LSL(logic shift left):逻辑左移;

LSR(logic shift right):逻辑右移;

ASR(arithmetic shift right):算术右移;

ROR(rotate shift right):循环右移;

RRX(rotate shift right with extend):带扩展的循环右移。其中的 C 指的是 CPSR 的 C 位。

5. 相对寻址

相对寻址是基址寻址的一种变通。由程序计数器 PC 提供基准地址,指令中的地址码字段作为偏移量,两者相加后得到的地址即为操作数的有效地址。例如:

```
B LOOP
...
LOOP MOV R6, #1
```

该条 B 指令的意思是要跳转到标号 LOOP 所代表的指令处,其含义相当明显,但要明白 CPU 根本不明白标号是个什么东西(事实上在指令的机器码中根本就没有标号这种东西),那么 B LOOP 这条指令的机器码会是什么呢? 答案是:高 8 bit 是操作码相关内容,低 24 bit 是

一个常数,表示从 B 指令到 MOV 指令之间的内存地址的差值(如果不考虑流水线的影响的话)。由此可见,B LOOP 这条指令相当于 `add pc, pc, # 偏移量常数`,典型地相对于 PC(当前指令地址)的相对寻址。由于是相对于当前指令地址进行相对寻址,所以无论程序最终运行在内存的何处(即使运行的地址不是它预期的位置),这条 B 指令都能正确运行。关于相对寻址、程序期望的运行地址等,将在“ARM 汇编伪指令”中详细描述。

顺便说一下,前面学到 B 指令的跳转范围是当前指令的先后 32MB,为什么是这个范围呢?因为 24bit 常数用 1bit 区别正负,还剩 23bit,同时由于 ARM 指令在内存中的地址的最低 2bit 一定是 0(为什么?请自行思考一下),因此 23bit 中可以不必表示这 2 个 0,所以 23bit 可以表示的范围是 $0 \sim 2^{25}$,即: $0 \sim 32\text{MB}$ 。

关于指令的机器码编码格式,请参阅技术文档“ARM Architecture Reference Manual”。

1.6.2 其他常见指令

在“基本寻址模式与基本指令”中学习了最常用的指令。下面介绍其他较为常用的指令。

1. 访存指令

LDRH(半字加载);

LDRSH(有符号半字加载);

STRH(半字存储);

交换指令如表 1-9 所列。

表 1-9 两个交换指令

| 助记符 | 说 明 | 操 作 |
|-----------------|-----------------|---|
| SWP Rd,Rm,[Rn] | 寄存器和存储器字进行数据交换 | 同时完成 $Rd \leftarrow [Rn], [Rn] \leftarrow Rm$ ($Rn \neq Rd$ 或 Rm) |
| SWPB Rd,Rm,[Rn] | 寄存器和存储器字节进行数据交换 | 同时完成 $Rd \leftarrow [Rn], [Rn] \leftarrow Rm$ ($Rn \neq Rd$ 或 Rm) |

2. 数据处理指令(表 1-10、表 1-11、表 1-12、表 1-13)

表 1-10 数据传送指令

| 助记符 | 说 明 | 操 作 |
|-----------------|-------|--|
| MVN Rd,operand2 | 数据非传送 | $Rd \leftarrow (\sim \text{operand2})$ |

表 1-11 算术运算指令

| 助记符 | 说 明 | 操 作 |
|----------------------|-----------|--|
| RSB Rd, Rn, operand2 | 逆向减法指令 | $Rd \leftarrow operand2 - Rn$ |
| ADC Rd, Rn, operand2 | 带进位加法 | $Rd \leftarrow Rn + operand2 + Carry$ |
| SBC Rd, Rn, operand2 | 带进位减法指令 | $Rd \leftarrow Rn - operand2 - (NOT)Carry$ |
| RSC Rd, Rn, operand2 | 带进位逆向减法指令 | $Rd \leftarrow operand2 - Rn - (NOT)Carry$ |

这里要特别提到,ADC 指令结合 CPSR,可以实现 64 位整数加法。

表 1-12 逻辑运算指令

| 助记符 | 说 明 | 操 作 |
|----------------------|--------|---------------------------------------|
| BIC Rd, Rn, operand2 | 按位清除指令 | $Rd \leftarrow Rn \& (\sim operand2)$ |

其实现功能是:将 Rn 中对应于 operand2 中为 1 的 bit 位全部清 0,其他 bit 位保持不变,然后将结果保存到 Rd 中。

表 1-13 比较指令

| 助记符 | 说 明 | 操 作 |
|------------------|--------|---------------------------------------|
| CMN Rn, operand2 | 负数比较指令 | 标志 N、Z、C、V $\leftarrow Rn + operand2$ |
| TST Rn, operand2 | 位测试指令 | 标志 N、Z、C $\leftarrow Rn \& operand2$ |
| TEQ Rn, operand2 | 相等测试指令 | 标志 N、Z、C $\leftarrow Rn - operand2$ |

TST 指令测试的是:Rn 中所有指定 bit 位是否全为 0(指定的 bit 位是 operand2 中为 1 的所有位)。

TEQ 指令测试的是:Rn 和 operand2 是否相等。这点与 CMP 指令一样,区别在于 CMP 指令除了可以比较两个数是否相等外,也可以比较两个数谁大谁小,但 TEQ 不行。

3. 乘法指令(表 1-14)

表 1-14 乘法指令

| 助记符 | 说 明 | 操 作 |
|--------------------------|-------------|---|
| MUL Rd, Rm, Rs | 32 位乘法指令 | $Rd \leftarrow Rm * Rs$ ($Rd \neq Rm$) |
| MLA Rd, Rm, Rs, Rn | 32 位乘加指令 | $Rd \leftarrow Rm * Rs + Rn$ ($Rd \neq Rm$) |
| UMULL RdLo, RdHi, Rm, Rs | 64 位无符号乘法指令 | $(RdLo, RdHi) \leftarrow Rm * Rs$ |

续表 1-14

| 助记符 | 说 明 | 操 作 |
|-----------------------|-------------|--|
| UMLAL RdLo,RdHi,Rm,Rs | 64 位无符号乘加指令 | $(RdLo, RdHi) \leftarrow Rm * Rs + (RdLo, RdHi)$ |
| SMULL RdLo,RdHi,Rm,Rs | 64 位有符号乘法指令 | $(RdLo, RdHi) \leftarrow Rm * Rs$ |
| SMLAL RdLo,RdHi,Rm,Rs | 64 位有符号乘加指令 | $(RdLo, RdHi) \leftarrow Rm * Rs + (RdLo, RdHi)$ |

4. 协处理器指令

参见“MMU 与内存保护的实现”。

5. 杂项指令

SWI: 软中断指令, 参见“swi 与 system call 的实现”;

MRS、MSR: 程序状态寄存器操作指令, 参见“ARM 异常处理”。

第 2 章

ARM 编程进阶

2.1 ARM 汇编伪指令

到目前为止,我们已经具备编写较为复杂的 ARM 汇编程序的能力,但要编写较为复杂且实用的程序,我们就不得不掌握 ARM 汇编的伪指令(pseudo-instruction)。千万别把汇编伪操作(directive)与汇编伪指令(pseudo-instruction)弄混了,directive 不会被编译器编译为机器指令,但 pseudo-instruction 会。而 pseudo-instruction 与指令(instruction)的区别在于,一条 instruction 与一条机器指令对应,而编译器会把一条 pseudo-instruction 编译为一条或多条机器指令。

ARM 汇编伪指令共 4 条:ldr、adr、adrl、nop。

2.1.1 精解 ldr 伪指令

首先我们来回答“基本寻址模式与基本指令”中提出的问题。“如果我们需要‘mov r0, #10000’这样的指令,应该怎么办?(常数 10 000 不能在机器指令 32bit 中的低 12bit 中被表示出来。)”当进行编译的时候,“Error: All70E”的错误就会出现,如图 2-1 所示。

其实,这个问题很容易解决,只需要将“mov r0, #10000”换为“ldr r0, =10000”即可。为什么这样就可以了呢?因为,这里的“ldr r0, =10000”并非 ldr 指令,而是一条伪指令,编译器会将这条伪指令替换为:

```
ldr r0, [pc, # -4]
DCD 10000
```

DCD 所分配的内存空间中存放了整数 10000,该内存空间被称为 literal pool,中文名称文字池。由于整个程序都是由编译器编译的(包括文字池的分配),所以很显然编译器能够知道 ldr 指令在内存中的地址与文字池在内存中的位置之间的偏移量,因此编译器就可以正确地使用以 pc 为基址,采用相对寻址的 ldr 指令将文字池中的数取出加载到寄存器 r0 中。由此可见,编译器对于“ldr r0, =10000”这条伪指令的处理,其实质是:在汇编源程序时,ldr 伪指令被编译器替换成一条合适的指令和存放常数的文字池。汇编器将常量放入文字池,并使用一

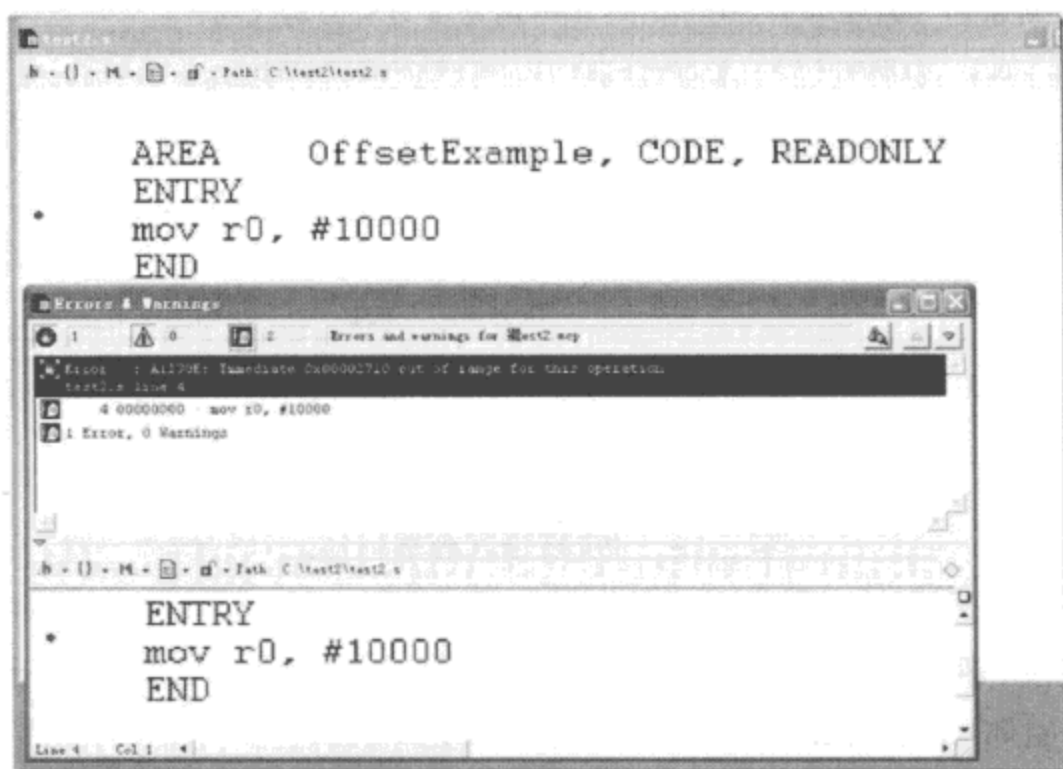


图 2-1 立即数不能被表示

条程序相对偏移的 ldr 指令从文字池读出常量。

由于,4 字节可以存放任何 int 型整数,这样一来,常数就可以是任何 int 型整数,而不再受制于 12 位的限制。当然此时的常数是存放在内存中的,而不是存放在机器指令的 32 位编码中的。

额外说明:

- ldr r0, [pc, #-4] 中是 -4,而不是 +4,是由于流水线的原因(参见图 1-3 流水线指令执行图)。今后对于流水线的这种影响,将不再予以特别说明。
- 从指令位置到文字池的偏移量必须小于 4 KB。
- 从语法上来看,与 ARM 指令的 ldr 相比,伪指令 ldr 的参数有“=”号,不是“#”号。
- 如果常数能够被 12 位表示出来,例如:ldr r0, =0x100,那么,编译器对该伪指令的处理,是使用 mov(或者 MVN)指令代替该 ldr 伪指令,例如:mov r0, #0x100,而不会采用 ldr 指令+文字池的方式。

除了“ldr 寄存器, =常数”这种形式外,还有“ldr 寄存器, =标号”这种形式也经常被使用,下面就来讲解这种形式的 ldr 伪指令,如图 2-2 所示。

由图 2-2 可见:“ldr pc, =InitStack”这条伪指令的作用是将标号 InitStack 所代表的地址赋予 pc。这里会使我们产生几个疑问:

(1) 为什么不使用“bl InitStack”,而要使用“ldr pc, =InitStack”?

这是因为 bl 指令的跳转范围是 $\pm 32\text{MB}$,而 InitStack 所代表的位置有可能距离“ldr pc, =InitStack”超过 32 MB,此时 bl 就无能为力了。竟然存在这么大范围跳转的程序(这似乎意味着编译出来的二进制可执行文件的大小会超过 32 MB),这一点似乎令我们感到非常震惊。

应用示例（源程序）：

```

...
LDR    pc,=InitStack
...
InitStack
MOV    R0, LR
...

```

使用伪指令将程序标号
InitStack的地址存入R1

编译后的反汇编代码：

```

...
0x60 LDR pc,[pc,#?] ;pc+=0xb4
...
0x64 MOV    R0, LR
...
0xb4 DCD    0x64

```

LDR伪指令被汇编成一条LDR指令，
并在文字池中定义了一个常量，该
常量为InitStack标号的地址

图 2-2 ldr 伪指令实现机制

事实上是：大小超过 32 MB 的可执行程序的确几乎不可能出现，但即使是很小的二进制程序中也可能进行大范围（超过 32 MB）的跳转，这一点在 Boot Loader 程序中几乎是必然的。

（2）编译器是如何得出 InitStack 所代表的地址是 0x64？

编译器知道“MOV R0, LR”这条指令相对于整个程序的第一条指令的偏移量为 0x64；同时又知道这个程序将来在内存中的运行地址为 0x0，所以编译器在编译的时候（不是程序运行的时候）就可以确定 InitStack 所代表的地址为 $0x64 + 0x0 = 0x64$ 。那么编译器又是如何知道“程序将来在内存中的运行地址”的呢？其实，这个“程序将来在内存中的运行地址”，通常称为程序的期望运行地址，简称运行地址，如图 2-3 所示。它其实是在编译程序前由程序员告知编译器的。

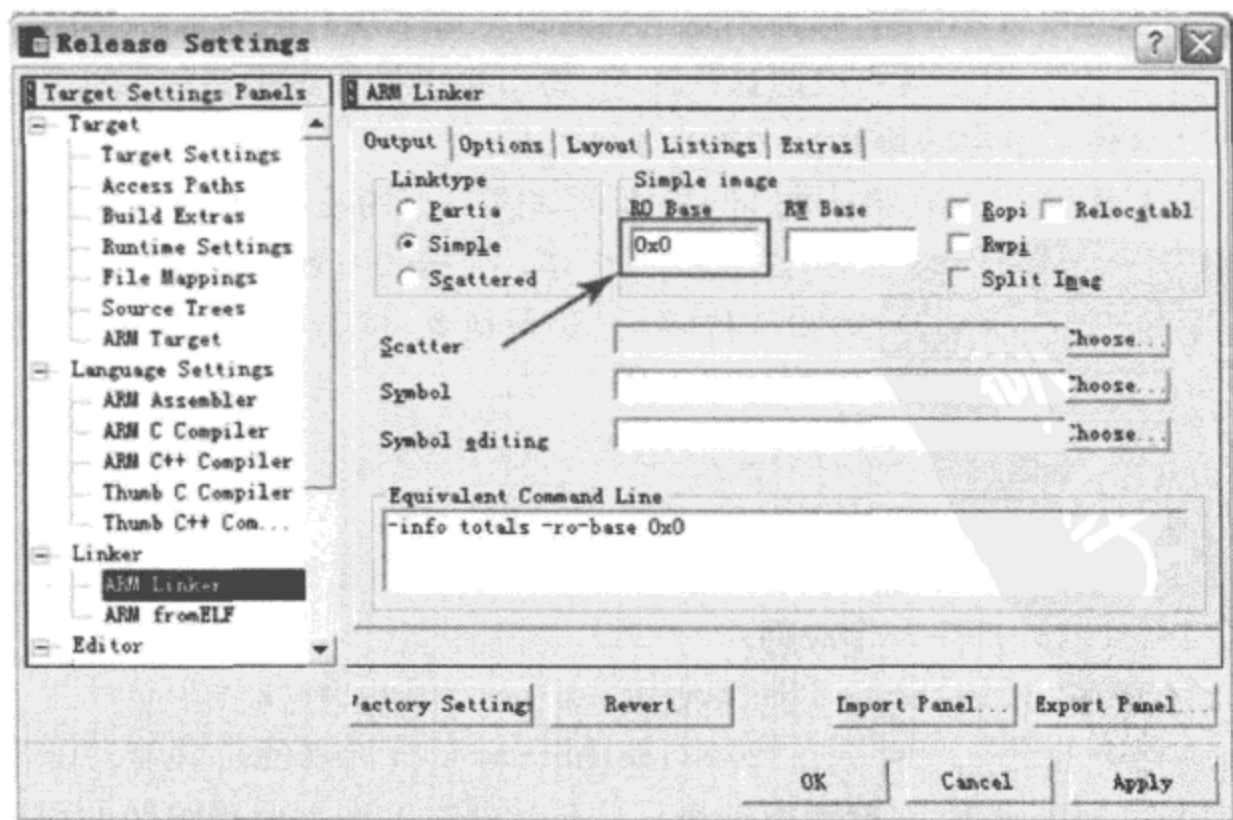


图 2-3 设置代码段运行地址

(3) 如果将来程序并没有运行在它的运行地址处,很显然这个程序就会出问题。如何解决?

出问题的原因,显然是由于 `ldr` 伪指令使用的是绝对地址。所以,解决的办法就是使用相对地址,进行相对寻址。这就要用到下面要介绍的另一条伪指令 `adr`。

2.1.2 精解 `adr`

`adr` 伪指令的作用与 `ldr` 伪指令的作用相同,都是将标号所代表的地址赋予寄存器,不过二者的实现机制是完全不同的: `ldr` 采用绝对地址, `adr` 采用相对地址。

`adr` 伪指令将基于 PC 相对偏移的地址值读取到寄存器中。在汇编源程序时, `adr` 伪指令被编译器替换成一条合适的指令。通常,编译器用一条 `add` 指令或 `sub` 指令来实现该 `adr` 伪指令的功能,如图 2-4 所示。

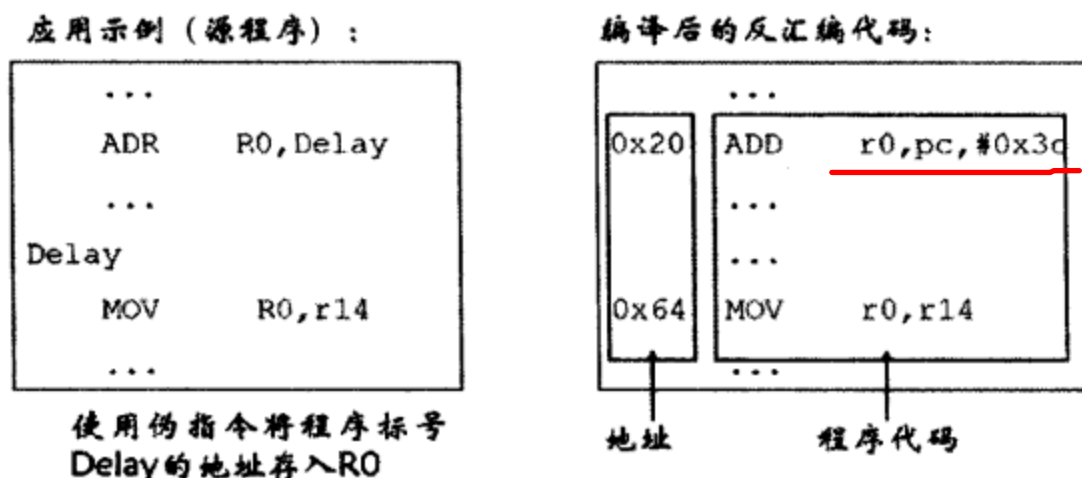


图 2-4 `adr` 伪指令实现机制

很显然,由于将“`ADR R0, Delay`”替换为“`ADD r0, pc, #0x3c`”,而它是以当前指令的地址(PC 的值)进行相对地址计算的。所以即使将来程序没有实际运行在运行地址处,也不会有问题。

当然,这里还有两个问题:

(1) 既然 `adr` 和 `ldr` 完成类似的功能, `adr` 又能避免绝对地址的问题,还要 `ldr` 伪指令有何用?

这主要是因为, `adr` 伪指令要求标号与 `adr` 伪指令必须在同一个段中(段的概念参见“ARM 汇编伪操作”),而 `ldr` 伪指令则没有这样的要求。

(2) “`add r0, pc, #0x3c`”中的常数 `0x3c` 是放在机器指令 12bit 中的立即数,这个立即数有可能不能被 12bit 表示出来。此时编译会产生错误。如果出现这样的情况,又应该怎么办?

使用下面要讲的伪指令 `adrl`。

2.1.3 精解 adrl 伪指令

在汇编源程序时,adrl 伪指令被编译器替换成两条合适的指令。其本质是:将偏移量这个立即数(可能不能被 12bit 表示出来)拆分为两个可以被 12bit 表示的立即数,然后用两条 add (或 sub)指令来替换 adrl 伪指令,如图 2-5 所示。

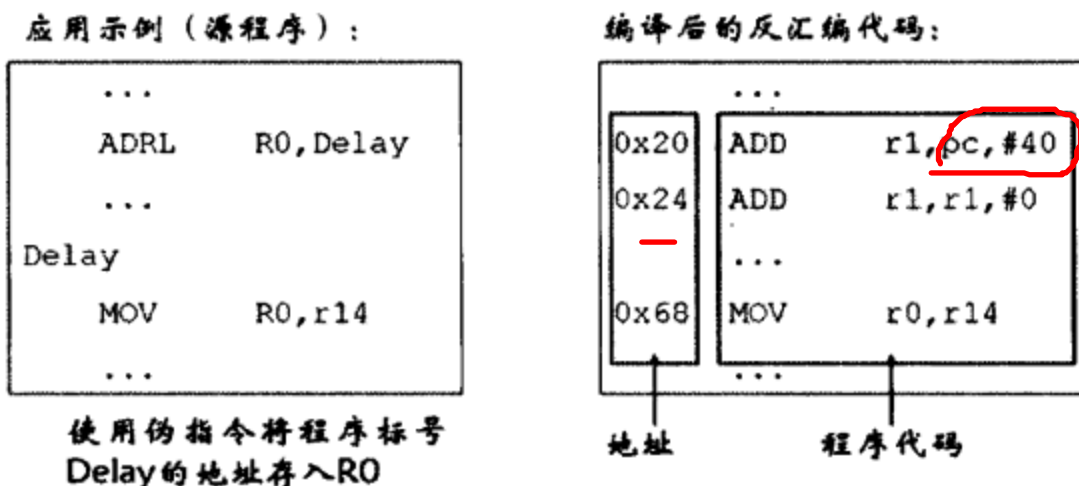


图 2-5 adrl 伪指令实现机制

当然你会问,如果那个立即数非常特殊,无论如何也拆分不成两个可以被 12bit 表示的立即数(也就是说需要拆分为 3 个甚至更多的数),那又应该怎么办? 关于这个问题,在这里不予回答,不过如果机器智能到啥都能做的话,程序员就失业了!

2.1.4 nop 伪指令

nop 是 no operation 的意思,就是 CPU 不做任何事的意思。这里千万要明白,CPU 一旦上电就将永不停歇地运行,绝不可能有一条指令能令 CPU 什么都不做。所以,该伪指令在汇编时将会被代替成“MOV R0,R0”指令。

nop 主要用于短延时操作。与应用程序不同,汇编程序通常要用于控制硬件,例如,你需要使用汇编程序要求某个硬件执行某个操作(例如:初始化 Nand Flash),并要在该操作完成后才能进行后面的操作,而硬件从接到命令到完成该操作需要一段时间(该时间很短,但对 CPU 而言却较长,通常需要几个指令周期)。此时,程序员就必须在发出命令的指令和后续操作之间做延时,通常的做法就是加入几个 nop 伪操作。

附:ldr 指令与 ldr 伪指令的 4 种形式如表 2-1 所示。(这 4 种形式,极其容易让初学者困惑,所以在此集中列出)

表 2-1 ldr 指令与伪指令的 4 种形式

| 形 式 | 描 述 |
|----------------|---|
| ldr r0, [r1] | 指令, 将 r1 指向的内存存放的内容加载到 r0 中 |
| ldr r0, label | 指令, 将标号 label 所代表的内存地址处存放的内容加载到 r0 中 |
| ldr r0, =10000 | 伪指令, 将常数 10 000 赋予 r0(采用 ldr 指令+文字池的方式实现) |
| ldr r0, =lable | 伪指令, 将标号 label 所代表的内存地址赋予 r0 |

2.2 ATPCS 与混合编程

完全使用汇编语言来编写程序会非常的繁琐, 因此通常情况下, 只是使用汇编程序来完成少量必须由汇编程序才能完成的工作, 而其他工作则由 C 语言程序来完成。这样一来, 实际上就是在进行汇编和 C 的混合编程, 甚至同一个程序的汇编源文件和 C 源文件是由不同的程序员编写的。在这种情况下, 要想使不同程序员编写的汇编代码和 C 代码能耦合得很好, 则必须有一个双方都必须遵守的规则, 这就是 ATPCS 规则。

2.2.1 ATPCS 规则精解

ATPCS(ARM - Thumb Procedure Call Standard)是 ARM 程序和 Thumb 程序中子程序调用的基本规则, 目的是使单独编译的 C 语言程序和汇编程序之间能够相互调用。这些基本规则包括子程序调用过程中寄存器的使用规则、数据栈的使用规则和参数的传递规则。

1. 寄存器的使用规则(表 2-2)

表 2-2 寄存器别名与特殊用途

| 寄存器 | 别 名 | 特殊名 | 使用规则 |
|-----|-----|-----|--------------------------------|
| R0 | a1 | | 参数/结果/scratch 寄存器 1 |
| R1 | a2 | | 参数/结果/scratch 寄存器 2 |
| R2 | a3 | | 参数/结果/scratch 寄存器 3 |
| R3 | a4 | | 参数/结果/scratch 寄存器 4 |
| R4 | v1 | | ARM 状态局部变量寄存器 1 |
| R5 | v2 | | ARM 状态局部变量寄存器 2 |
| R6 | v3 | | ARM 状态局部变量寄存器 3 |
| R7 | v4 | wr | ARM 状态局部变量寄存器 4, Thumb 状态工作寄存器 |
| R8 | v5 | | ARM 状态局部变量寄存器 5 |

| 寄存器 | 别名 | 特殊名 | 使用规则 |
|-----|----|-----|---|
| R9 | v6 | sb | ARM 状态局部变量寄存器 6, 在支持 RWPI 的 ATPCS 中为静态基址寄存器 ✓ |
| R10 | v7 | sl | ARM 状态局部变量寄存器 7, 在支持 RWPI 的 ATPCS 中为数据栈限制指针 ✓ |
| R11 | v8 | fp | ARM 状态局部变量寄存器 8/帧指针 ✓ |
| R12 | | ip | 子程序内部调用的 scratch 寄存器 |
| R13 | | sp | 数据栈指针 |
| R14 | | lr | 连接寄存器 |
| R15 | | pc | 程序计数器 |

(1) 寄存器 R0~R11 被分为两组: a1~a4, v1~v8。所以对于兼容 ATPCS 的编译器而言, 在编程的时候可以使用 a1~a4 替换 R0~R3。

(2) 除了 R13~R15 有别名外, 对于兼容 ATPCS 的编译器而言, 也可以使用其他寄存器的别名: wr, sb, sl, fp, ip, 它们都有自己的一些特殊用法。

(3) 寄存器 R0~R3 用于传递子程序的参数和返回结果(详见本节后部)。

(4) 寄存器 a1~a4 和 ip 是 scratch 寄存器(即: 临时寄存器), 其值在进行子程序调用时不需要保存和恢复。(详见本节后部)

2. 数据栈的使用规则

在“其他寻址模式与其他指令”中, 栈有 4 种类型:

- (1) FD(Full Descending)满递减。
- (2) ED(Empty Descending)空递减。
- (3) FA(Full Ascending)满递增。
- (4) EA(Empty Ascending)空递增。

ATPCS 规定数据栈为 FD(满递减)类型, 并且对数据栈的操作是 8 字节对齐的。这意味着在编写汇编子程序时, 如果要进行出栈和入栈操作, 则必须使用 ldmfd 和 stmfd 指令(或者 ldmia 和 stmdb); 而兼容 ATPCS 的编译器在编译 C 代码时, 也必须这样做。

3. 参数的传递规则

(1) 参数个数固定的子程序参数传递规则:

前 4 个整数参数, 通过寄存器 R0~R3 来传递。其他参数通过数据栈传递。

(2) 子程序结果返回规则:

结果为一个 32 位的整数时, 必须通过寄存器 R0 返回; 结果为一个 64 位整数时, 通过寄存器 R0 和 R1 返回, 依次类推。

下面看一下编译器对于这几个规则的遵循(实现)情况。

(1) 参数传递(4 个参数)以及结果返回:如图 2-6 所示,很显然,主调程序在调用子程序(即:bl func1)之前,把将要传递给子程序的 4 个参数准备在了 R0~R3 中,从而使得子程序可以通过该 4 个寄存器获得转递给它的参数(即:4 个参数是通过寄存器 R0~R3 来传递的);子程序在返回之前,将返回值放在了寄存器 R0 中,从而使得主调函数可以通过 R0 来获得子程序的返回值(即:结果为一个 32 位的整数时,通过寄存器 R0 返回)。

| | |
|---|--|
| int func1(int a, int b, int c, int d) { return a+b+c+d; } | func1 0x000000 : ADD r0,r0,r1 0x000004 : ADD r0,r0,r2 0x000008 : ADD r0,r0,r3 0x00000c : MOV pc,lr |
| int caller1(void) { return func1(1,2,3,4); } | caller1 0x000014 : MOV r3,#4 0x000018 : MOV r2,#3 0x00001c : MOV r1,#2 0x000020 : MOV r0,#1 0x000024 : BL func1 |

图 2-6 参数传递规则

(2) 多于 4 个参数,前 4 个参数通过寄存器 R0~R3 来传递,其他参数通过数据栈传递。7 个参数的情景。下面是程序的反汇编结果。通过它来分析一下当参数超过 4 个的时候,所谓“通过数据栈传递其他参数”是什么含义。

```
int func(int a, int b, int c, int d, int e, int f, int g)
{
    return(a+b+c+d+e+f+g);
}

int main()
{
    int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
    return func(a,b,c,d,e,f,g);
}

1 int func(int a, int b, int c, int d, int e, int f, int g)
2 {
3 func [0xe92d4010] stmfd r13!,{r4,r14}
4 000080ac [0xe59d4010] ldr r4,[r13,#0x10] //r4 为第 7 个参数的值
5 000080b0 [0xe28de008] add r14,r13,#8 //r14 指向了存放传入参数在栈中的位置
6 000080b4 [0xe89e5000] ldmba r14,{r12,r14} //r12 为第 5 个参数的值,r14 为第 6 个参数的值
7 return(a+b+c+d+e+f+g);
```



```

8 000080b8 [0xe0800001] add r0,r0,r1
9 000080bc [0xe0800002] add r0,r0,r2
10 000080c0 [0xe0800003] add r0,r0,r3
11 000080c4 [0xe080000c] add r0,r0,r12
12 000080c8 [0xe080000e] add r0,r0,r14
13 000080cc [0xe0800004] add r0,r0,r4
14 }
15 000080d0 [0xe8bd8010] ldmfd r13!,{r4,pc}
16 int main()
17 {
18 main [0xe92d401e] stmfd r13!,{r1 - r4,r14}
19 int a = 1,b = 2,c = 3,d = 4,e = 5,f = 6,g = 7;
20 000080d8 [0xe3a00001] mov r0,#1
21 000080dc [0xe3a0c002] mov r12,#2
22 000080e0 [0xe3a0e003] mov r14,#3
23 000080e4 [0xe3a04004] mov r4,#4
24 000080e8 [0xe3a01005] mov r1,#5
25 000080ec [0xe3a02006] mov r2,#6
26 000080f0 [0xe3a03007] mov r3,#7
27 return func(a,b,c,d,e,f,g);
28 000080f4 [0xe88d000e] stmia r13,{r1 - r3} //main 处的人栈操作,r1~r3 实为占位符,是替第 5、6、
//7 个参数预先在栈内占位置的
29 000080f8 [0xe1a03004] mov r3,r4
30 000080fc [0xe1a0200e] mov r2,r14
31 00008100 [0xe1a0100c] mov r1,r12
32 00008104 [0xebffffe7] bl func
33 }
34 00008108 [0xe8bd801e] ldmfd r13!,{r1 - r4,pc}

```

第 3 行采用的是 stmfd 指令实施入栈,这是因为要满足 ATPCS 中的“数据栈的使用规则”。而入栈的寄存器是 r4 和 r14,r4 入栈是因为 r4 在子程序中被破坏(使用)了,因此必须在子程序的入口入栈保存,在子程序的出口处出栈恢复(第 15 行);而 r14 要入栈则是因为 r14 存放的是子程序的返回地址,而 r14 又在子程序中被破坏(使用)了,如果不保存的话,在子程序返回(第 15 行)的时候,将不会正确地返回到主调程序。当然,也许发现了 r0、r1、r2、r3、r12 同样在子程序中被破坏了,为什么它们不需要保存和恢复呢?这是因为“寄存器 a1~a4 和 ip 是 scratch 寄存器(即:临时寄存器),其值在进行子程序调用时不需要保存和恢复。”(见前文)。也就是说,对于主调程序的编写者而言,应该很清楚他必须遵循 ATPCS 规则,所以他不会期望在子程序返回后,寄存器 r0、r1、r2、r3、r12 的值一定会维持原样。因此子程序的编写者也就

不必保存和恢复这几个寄存器了,即使子程序破坏了它们的值。随便说一句,这条 `stmfd` 指令是由编译器自动加在子函数的第一条语句之前的,所以类推一下就应该明白, `main` 函数运行时的第一条指令并不是程序员书写 `main` 函数的第一条语句,而是编译器添加的入栈指令。更进一步,为什么编译器要加这条入栈指令呢? 因为 `main` 函数本质上也是个子函数而已,它也会被别人调用,也就是说,程序运行起来后, `main` 函数并不是首先运行的。那么,是谁首先运行呢? 当然是调用 `main` 函数的代码,这段代码被称为 例行启动程序 (boot routine), 或称启动例程。它是由编译器在编译程序时自动加入的。

第 20、21、22、23、29、30、31 行显然是在准备(传递)前 4 个参数;第 18、24、25、26、28 行的执行,显然将后 3 个参数放到了栈中,而第 4、5、6 行完成后,子程序则将栈中的 3 个参数取出了。这样就完成了“多于 4 个的参数通过数据栈来传递”这个操作,如图 2-7 所示。

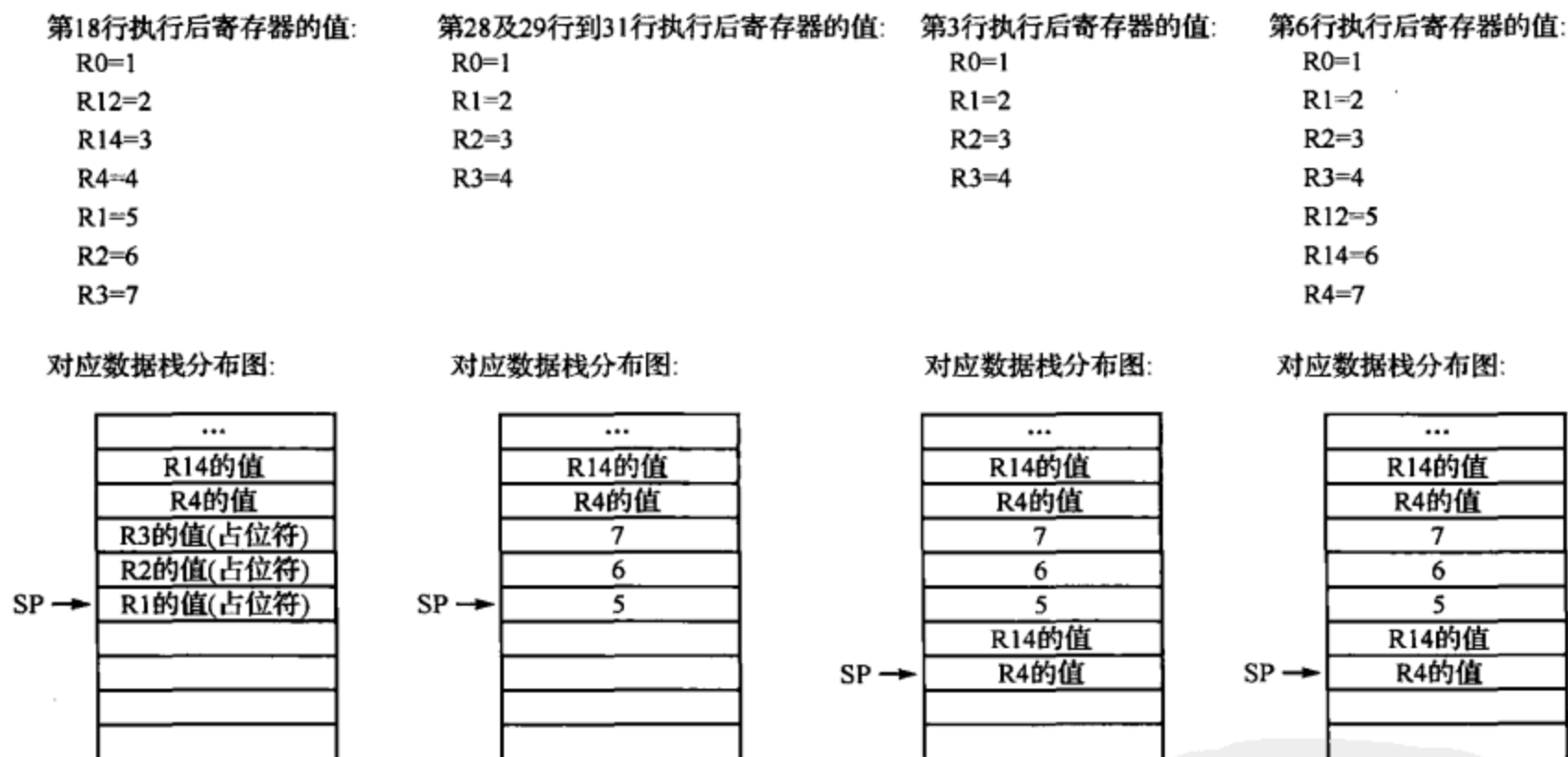


图 2-7 通过栈传参

由此可以得到关于程序优化的一个结论:开始 4 个字大小的参数直接使用寄存器的 `R0~R3` 来传递(快速且高效的);如果需要更多的参数,将使用堆栈。(需要额外的指令和慢速的存储器操作);所以通常限制参数的个数,使它为 4 或更少,如果不可避免,把常用的参数前 4 个放在 `R0~R3` 中。

2.2.2 精解 C 和 ARM 汇编程序间的相互调用

在 C 和 ARM 汇编程序之间相互调用必须遵守 ATPCS 规则。C 和汇编之间的相互调用可以从以下这五方面来说明:

- 在 C 语言程序中调用汇编程序。
- 在汇编程序中调用 C 语言程序。
- 汇编程序对 C 全局变量的访问。
- C 程序对汇编全局变量的访问。
- C 程序中内嵌汇编。

在下面的学习中请参照示例代码(位于光盘\work\armarch\mix_prog):

1. 在 C 语言程序中调用汇编子程序

为了保证程序调用时参数的正确传递,汇编子程序的设计要遵守 ATPCS。在汇编程序中需要使用 EXPORT 伪操作来声明汇编子程序,使得该子程序可以被其他程序调用。同时,在 C 程序调用该汇编程序之前需要在 C 语言程序中使用 extern 关键词来声明该汇编子程序。

参阅示例代码中 xmain 函数(在 ledtest.c 中)对 delay 函数(在 delay.s 中)的调用;

ledtest.c 中 extern int delay(int time);

delay.s 中 EXPORT delay。

2. 在汇编程序中调用 C 语言子程序

为了保证程序调用时参数的正确传递,汇编程序的设计要遵守 ATPCS。在 C 程序中不需要使用任何关键字来声明将被汇编语言调用的 C 程序(只要该程序的声明前不要加 static 关键字),但是在汇编程序调用该 C 程序之前需要在汇编语言程序中使用 IMPORT 伪操作来声明该 C 程序。在汇编程序中通过 bl 指令来调用子程序。

参阅示例代码中 init.s 文件中的代码对 xmain 函数的调用。

init.s 中存在代码:

```
IMPORT xmain
```

```
bl xmain
```

ledtest.c 中存在代码:

```
int xmain(int val)
```

3. 汇编程序访问全局 C 变量

汇编程序可以通过 C 全局变量的地址来间接访问 C 语言程序中定义的全局变量。在汇编程序中,通过使用 IMPORT 关键词引入 C 全局变量,该 C 全局变量的名称在汇编程序中被认为是一个标号,从而汇编程序可以利用 ldr 和 str 指令访问该标号所代表的地址处存放的内容(即:C 全局变量的值)。

参阅示例代码中 init.s 文件中的如下几行:

```
IMPORT i
```

```
ldr r0, i
```



```
sub r0, r0, #1
str r0, i
```

对于不同类型的变量,需要采用不同选项的 ldr 和 str 指令,如下所示:

```
unsigned char LDRB/STRB
unsigned short LDRH/STRH
unsigned int LDR/STR
char LDRSB/STRB
short LDRSH/STRH
```

4. C 程序对汇编全局变量的访问

汇编程序中用 DCD 为全局变量分配空间并赋初值,并定义一个标号代表该存储位置,用 EXPORT 导出该标号。C 程序将会将该标号视为全局变量的名称,在 C 程序中用 extern 声明该全局变量,之后就可以按正常的方式访问该全局变量了。

参阅示例代码中 delay.s 文件中的代码和 xmain 函数的代码:

```
EXPORT DELAYVAL
DELAYVAL
DCD 0xffff
```

```
extern int DELAYVAL;
```

5. C 程序中内嵌汇编

有些操作 C 语言程序是做不了的,例如:改变 CPSR 寄存器的值、初始化堆栈指针寄存器 SP 等,它们只能由汇编程序完成。但出于编程简洁以及其他一些因素的考虑,有时需要在 C 源代码中实现上述的操作,此时我们就必须采用在 C 源代码中嵌入少量汇编代码的方法来实现,这就是 C 程序中的内嵌汇编。

内嵌的汇编指令包括大部分的 ARM 指令和 Thumb 指令,但是不能直接引用 C 的变量定义,数据交换必须通过 ATPCS 进行,不支持诸如直接修改 PC 实现跳转等底层功能。嵌入式汇编语句在形式上是独立定义的函数体,其语法格式为:

```
__asm
{
指令[;指令]
...
[指令]
}
```

其中“__asm”为内嵌汇编语句的关键字,需要特别注意的是前面有两个下划线。同一行

第2章 ARM 编程进阶

如有多条指令,则指令之间用分号分隔,如果一条指令占据多行,除最后一行外都要使用续行符“\”。

例如,如果需要在 C 程序中禁用中断,那么内嵌的汇编代码如下:

```
__asm
{
    MRS    R0 CPSR
    ORR    R0, R0, # 0x80
    MSR    CPSR_c, R0
}
```

出于完整性的考虑,最后将内嵌汇编相对于一般汇编的一些不同的特点罗列如下:

- 操作数可以是寄存器、常量或 C 表达式。它们可以是 char、short 或者 int 类型,而且是作为无符号数进行操作。
- 内嵌的汇编指令中使用物理寄存器有一些限制。
- 常量前的符号“#”可以省略。
- 只有指令 B 可以使用 C 程序中的标号,指令 BL 不能使用 C 程序中的标号。
- 不支持汇编语言中用于内存分配的伪操作。
- 指令中如果包含常量操作数,该指令可能会被汇编器展开成几条指令。
- 内嵌汇编器不支持通过“.”指示符或 PC 获取当前指令地址。
- 不支持“LDR Rn, = expression”伪指令,而使用“MOV Rn, expression”指令向寄存器赋值。
- 不支持标号表达式。
- 不支持 ADR 和 ADRL 伪指令。
- 不支持 BX 和 BLX 指令。
- 不可以向 PC 赋值。
- 使用 0x 前缀替代“&”表示十六进制数。
- 必须小心使用物理寄存器,如 R0~R3、LR 和 PC。
- 不要使用寄存器寻址变量。
- 使用内嵌汇编时,编译器自己会保存和恢复它可能用到的寄存器,用户无须保存和恢复寄存器。
- ldm 和 stm 指令的寄存器列表只允许物理寄存器。

2.3 裸机硬件的控制方法与例程

到目前为止,我们已经能够编写较复杂的 ARM 汇编程序了,遗憾的是这些程序是运行在

ads 自带的虚拟开发板 ARMUL 下的(在 axd 界面下,单击 options→configure target,可见到如图 2-8 所示的目标板配置界面)

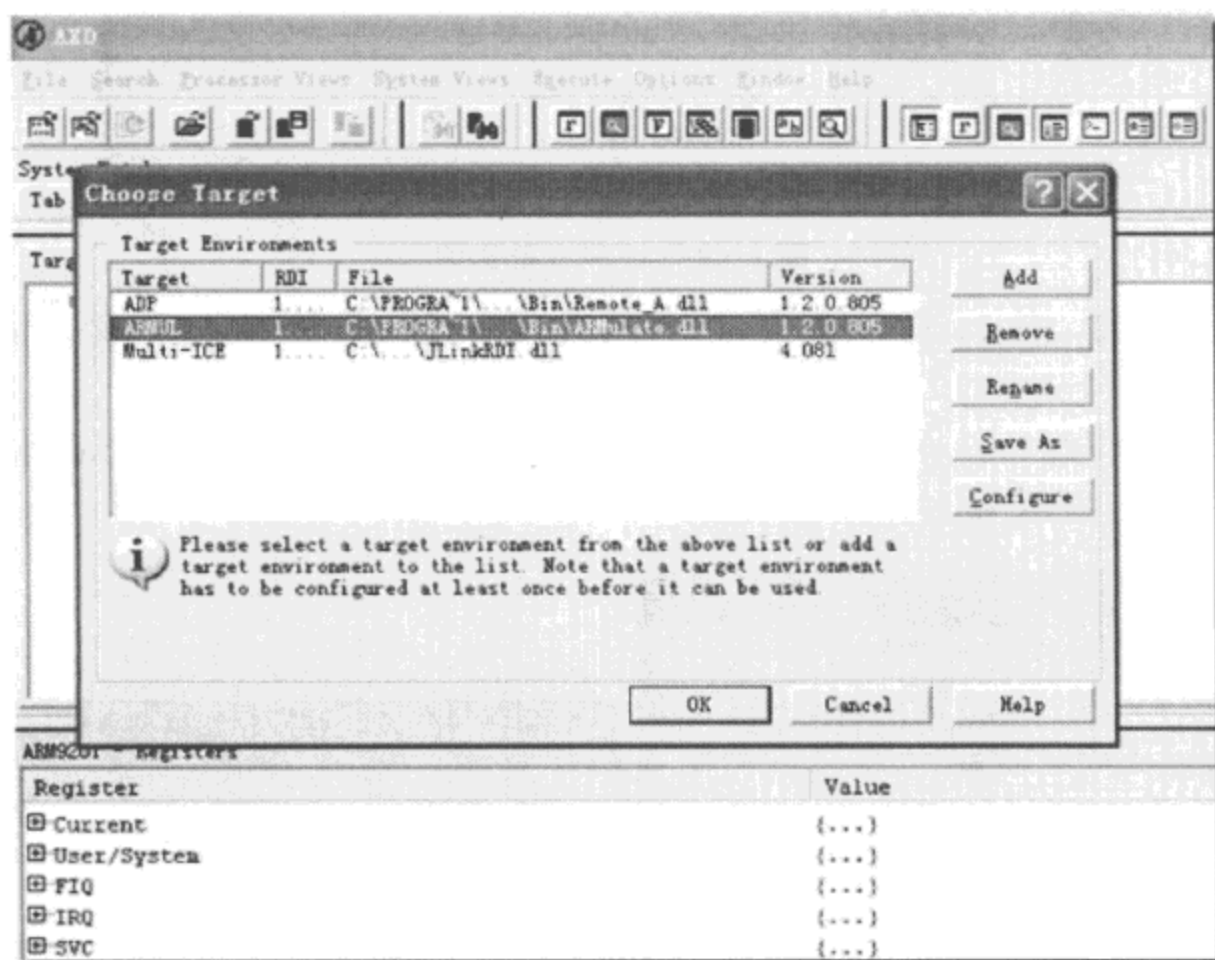


图 2-8 目标板配置

而我们最终的目的是要让程序运行在实际的硬件产品上,并能控制硬件。本节将初步介绍如何建立真实硬件的开发和调试环境,编写控制硬件的程序的方法。

2.3.1 建立真实硬件的开发和调试环境

由于基于 ARM 的嵌入式开发板种类众多,硬件仿真器、调试代理软件也是种类繁多,使用方法各异,这就为学习编写 ARM 裸机程序控制硬件带来了较大的难度。为便于初学者快速入门,有必要选择一套成熟、易于学习和实践的软硬件环境。本书与硬件相关的实验和代码均使用的是带有 Nor Flash 的 S3C2440 嵌入式开发板(自带简易的 JTAG 调试仿真器小板)和 H-JTAG 调试器代理软件。这样的嵌入式开发板目前在市面均比较容易买到。

1. 安装并设置 H-JTAG

(1) 安装 H-JTAG。先上网下载 H-JTAG 后,双击其安装文件运行,按照其提示安装即可。安装完毕,会在桌面生成 H-JTAG 和 H-Flasher 快捷方式,双击运行 H-JTAG,程序将自动检测是否连接了 JTAG 设备,因为之前我们还没有做任何设置,所以会跳出一个提示窗口,如图 2-9 所示。

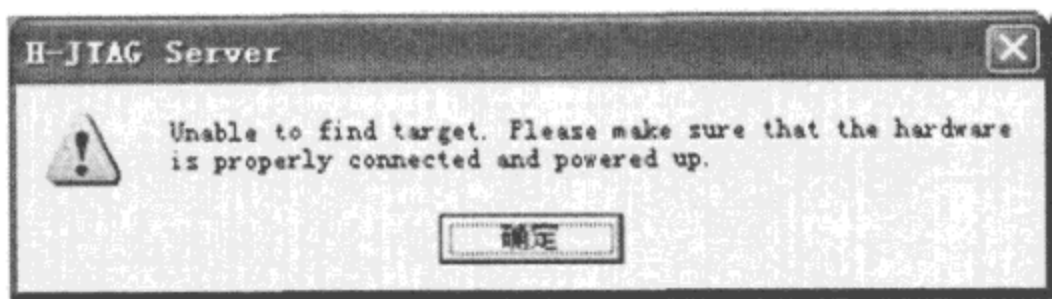


图 2-9 检测设备告警

单击“确定”按钮,进入程序主界面,因为没有连接任何目标器件,因此显示如图 2-10 所示。

(2) 设置 JTAG 端口。在 H-JTAG 主界面的菜单里点 Setting→Jtag Settings,作图 2-11 所示设置,单击 OK 按钮返回主界面。

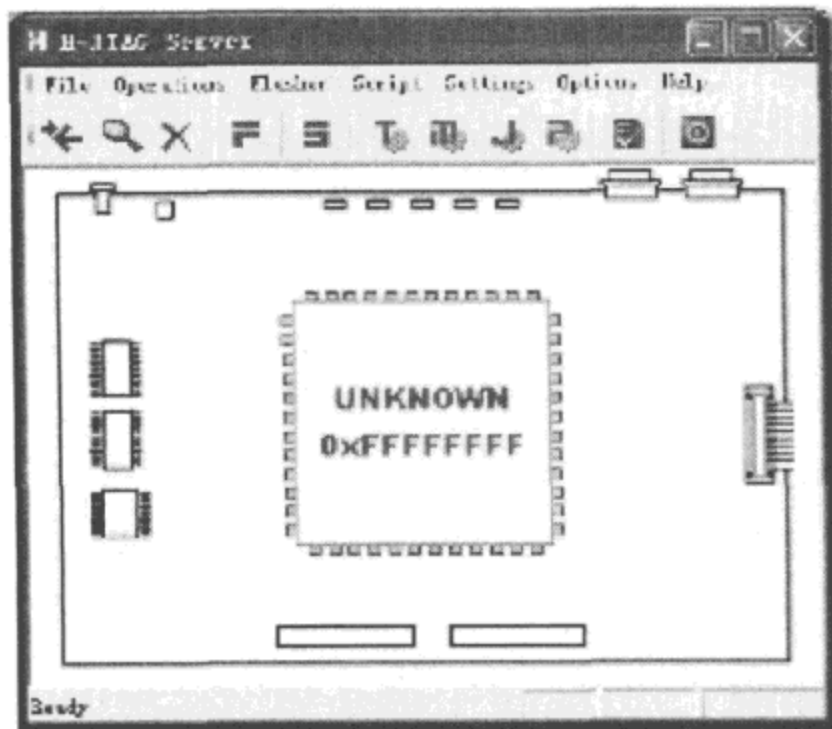


图 2-10 h-jtag 未检测到设备

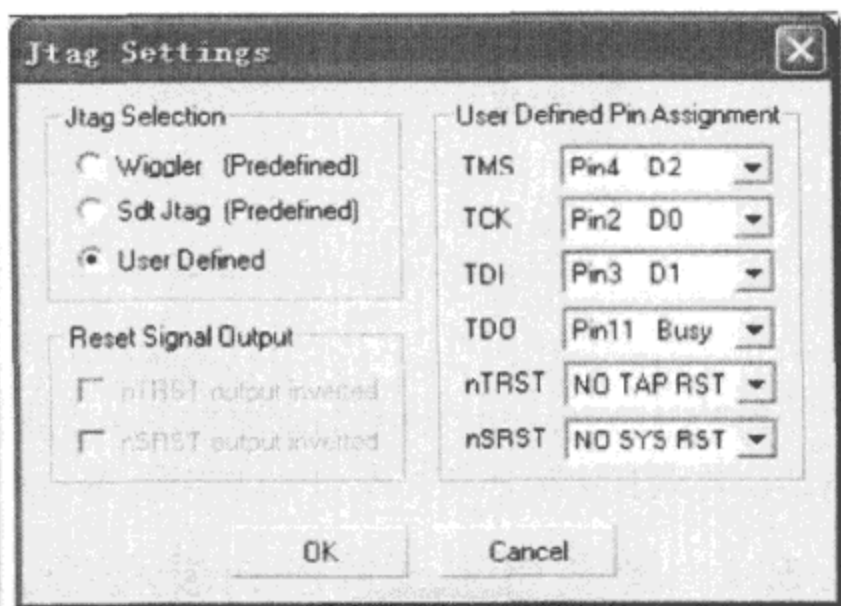


图 2-11 配置 JTAG 连接参数

(3) 设置初始化脚本。把 h-jtag 中的 H-Flasher_my2440.hfc、H-Flasher_my2440_sst.hfc 和 my2440.his 文件复制到 H-JTAG 的安装目录(默认安装的话,该目录是 c:\Program Files\H-JTAG)。

在 H-JTAG 的主界面,单击 Script→Init Script,跳出 Init Script 窗口,单击该窗口下面的 Load 按钮,找到并选择打开刚刚复制的 my2440.his 文件,如图 2-12 所示。

这时,Init Script 窗口会被载入的脚本填充,如图 2-13 所示,注意要点选 Enable Auto Init 单选按钮,单击 OK 按钮退回 H-JTAG 主界面。



图 2-12 配置 H-JTAG 初始化脚本

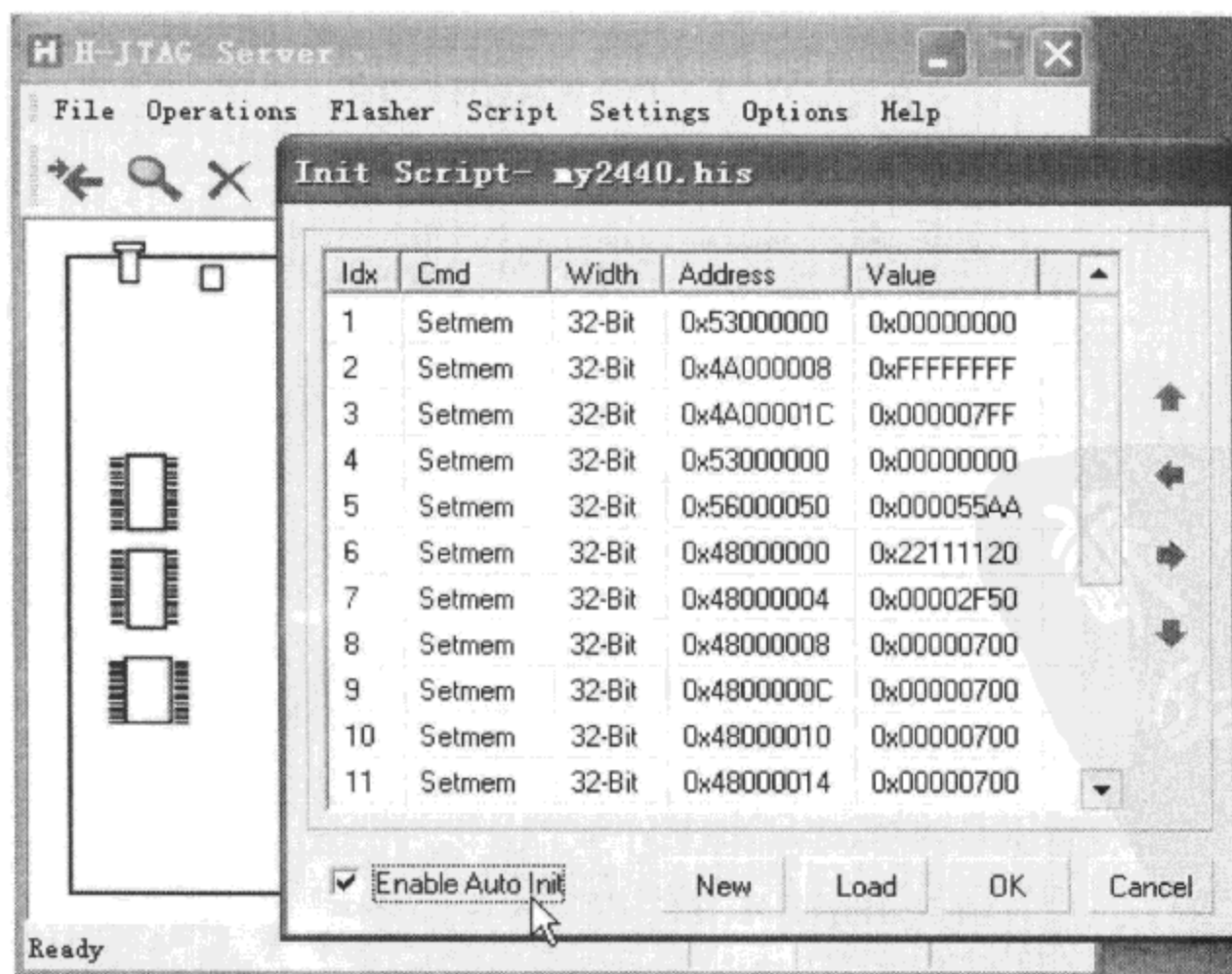


图 2-13 配置自动初始化

第2章 ARM 编程进阶

(4) 检测目标器件(图 2-14)。使用开发板附带的 JTAG 小板连接开发板的 JTAG 接口,并接上打开电源。单击主菜单 Operations→Detect Target,或者单击工具栏相应的图标也可以,这时就可以看到已经检测到目标器件了。

特别说明:如果没有设置初始化脚本,也可以检测到 CPU,但无法进行下面的单步调试。



图 2-14 H-JTAG 检测到目标设备

(5) 设置自动下载程序到 Nor Flash。如图 2-15 所示,单击 Flasher,点选 Auto Download 单选按钮。

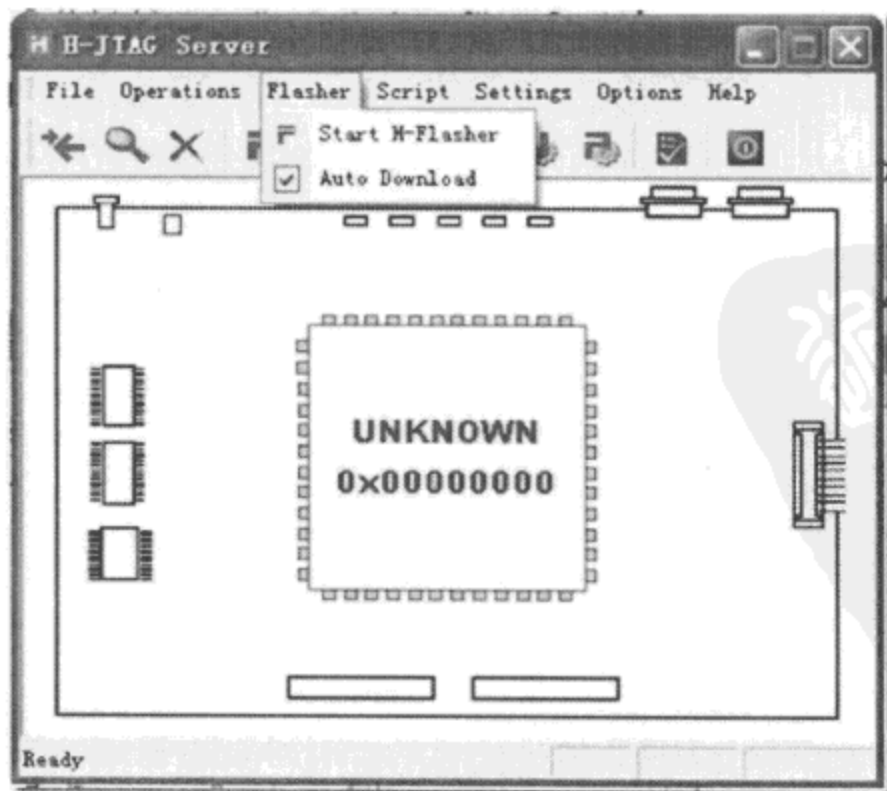


图 2-15 设置 H-Flasher 自动下载程序

(6) 设置 Nor Flash 型号。如图 2-16 所示,在出现的 H-Flasher 配置窗口中,单击菜单项 Load,找到并选择打开刚才复制的 H-Flasher_my2440_sst.hfc(或者 H-Flasher_my2440.hfc)文件(根据所用开发板上配置的 Nor Flash 型号而定)。

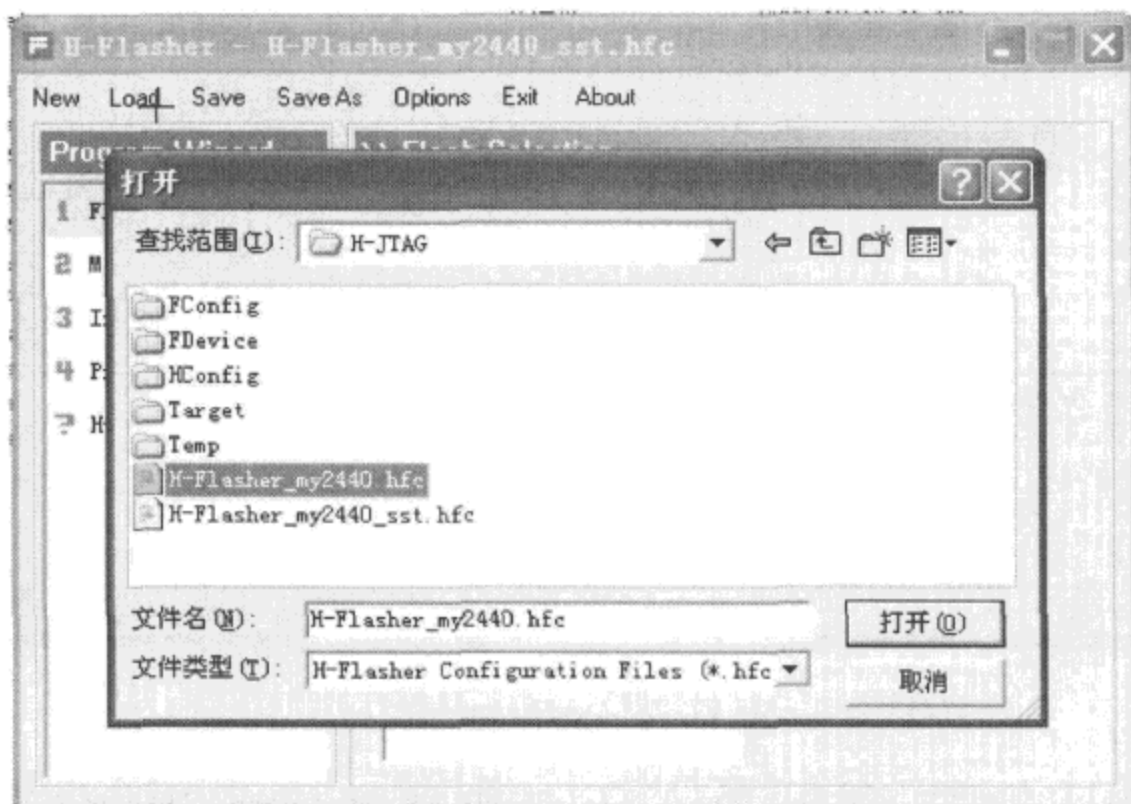


图 2-16 加载 H-Flasher 配置文件

2. 为 H-JTAG 配置 AXD DEBUGGER

(1) 运行 AXD Debugger。如图 2-17 所示打开运行 ADS1.2 软件的调试软件—AXD Debugger:

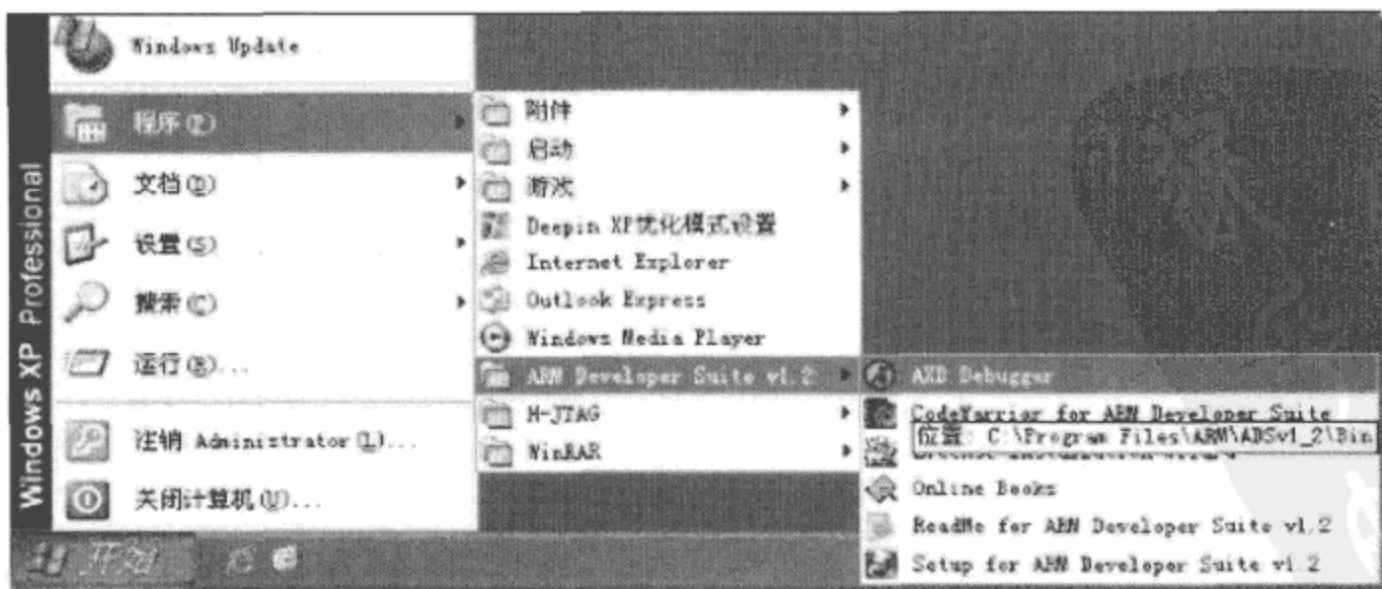


图 2-17 启动调试器 AXD

第2章 ARM 编程进阶

AXD Debugger 主界面如图 2-18 所示。

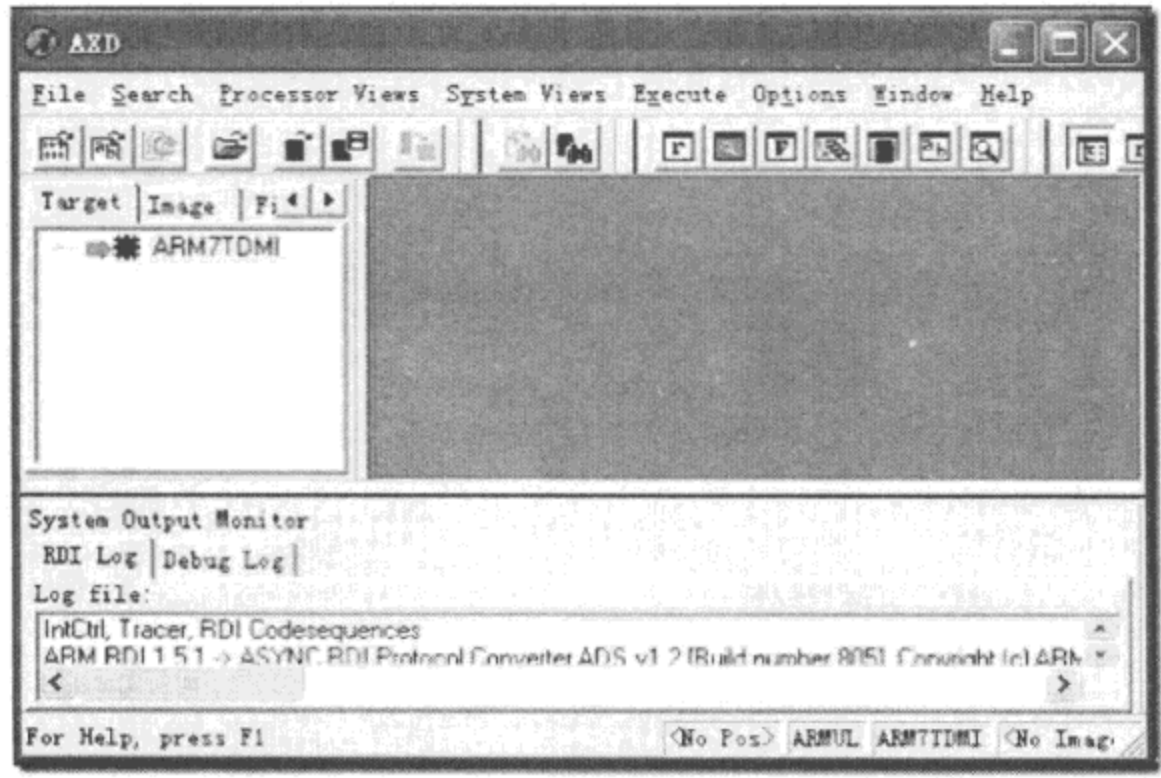


图 2-18 AXD 主界面

(2) 设置 AXD Debugger。单击菜单 Options→Configuer Target…命令,出现如图 2-19 所示窗口。

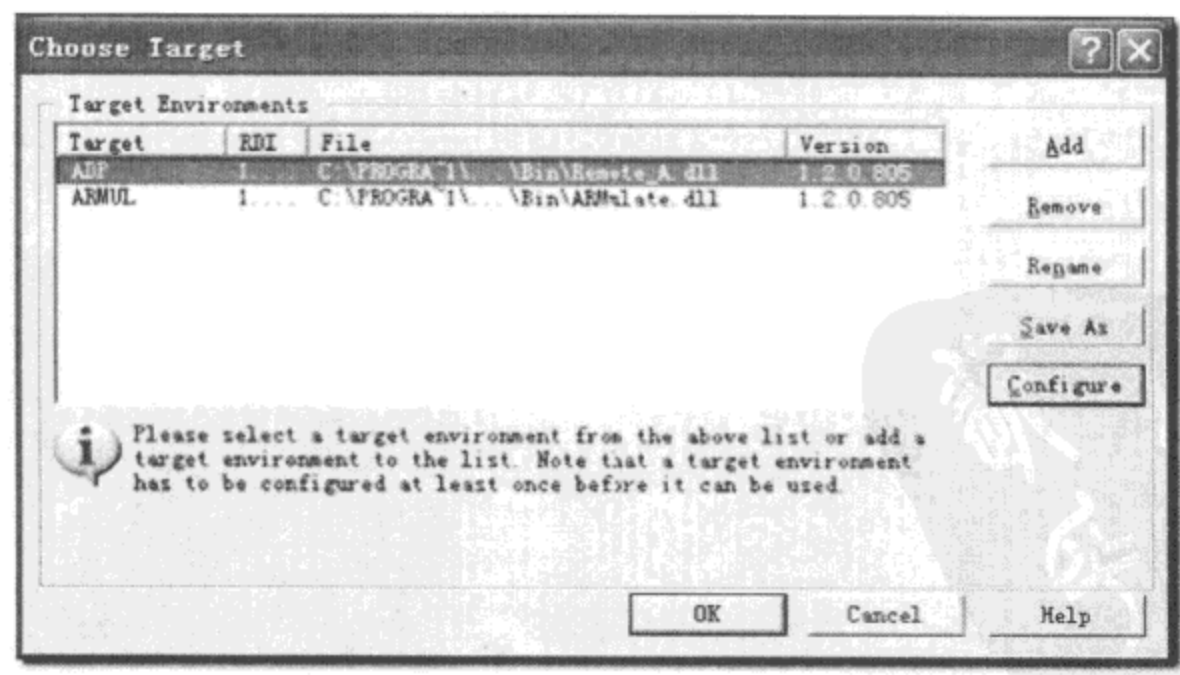


图 2-19 AXD 目标初始情况

在该窗口中单击 Add 按钮,跳出选择文件对话框,找到 H-JTAG 安装目录,选择并打开里面的 H-JTAG.dll 文件,如图 2-20 所示。

此时会在 Choose Target 窗口中多了一项 H-JTAG,如图 2-21 所示。然后单击 Con-

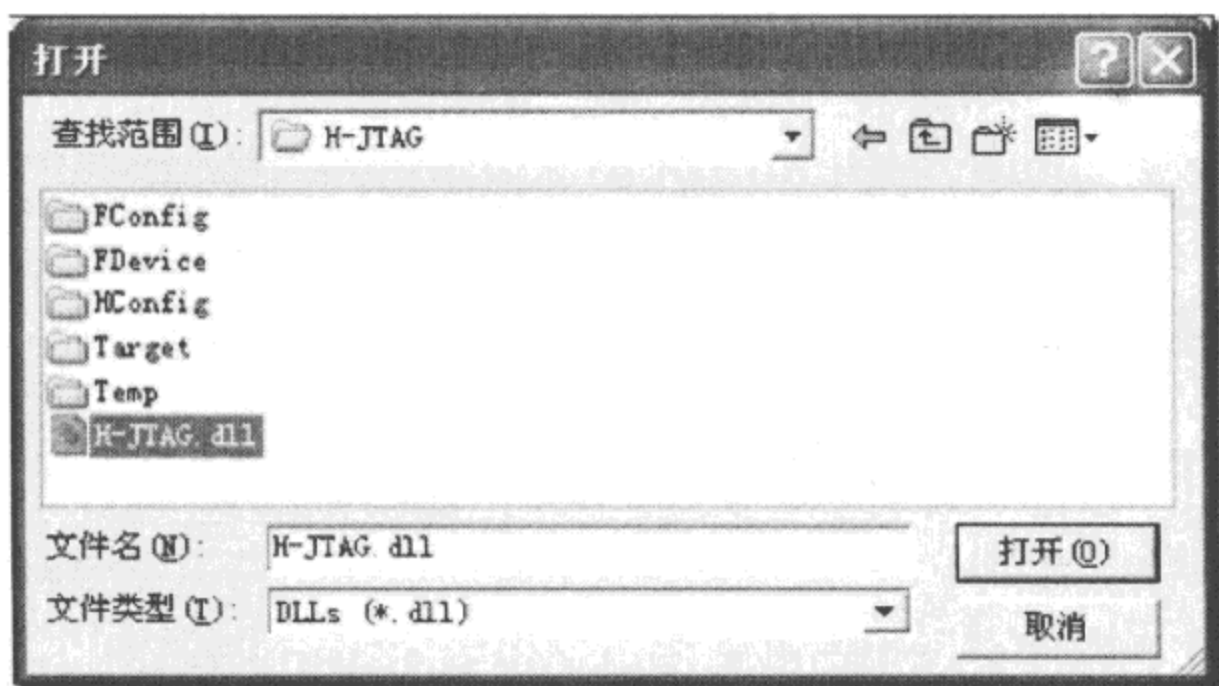


图 2-20 加载 H-JTAG 调试代理插件

figure 按钮。然后单击 OK 按钮返回 AXD Debugger 主界面。

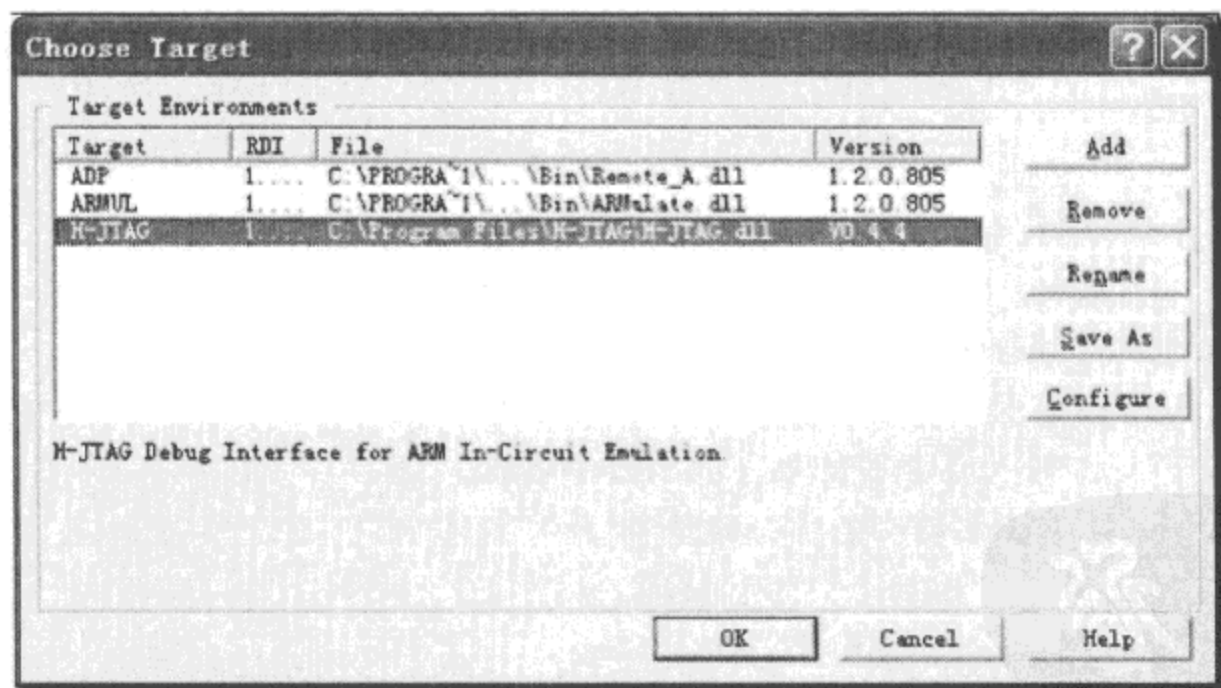


图 2-21 H-JTAG 成为 AXD 目标初始

(3) 使用 H-JTAG 在 ADS1.2 环境下进行仿真调试。关闭 AXD Debugger。在 CodeWarrior 中单击菜单 Project→Debug, 它将自动启动 AXD Debugger, AXD Debugger 会启动目标映像, 并通过 JTAG 下载至目标板, 这时在 AXD Debugger 底部的状态栏会出现下载过程提示, 下载完毕就可以进行单步或者全速调试了, 调试过程中可以看到 CPU 各个寄存器的值, 也可以设置断点等, 详细的用法请参考 ADS 附带的英文说明手册。

3. 使用 H-JTAG 中的 H-Flasher 烧写裸机程序到开发板的 Nor Flash 中

(1) 启动 H-Flasher 如图 2-22 所示。

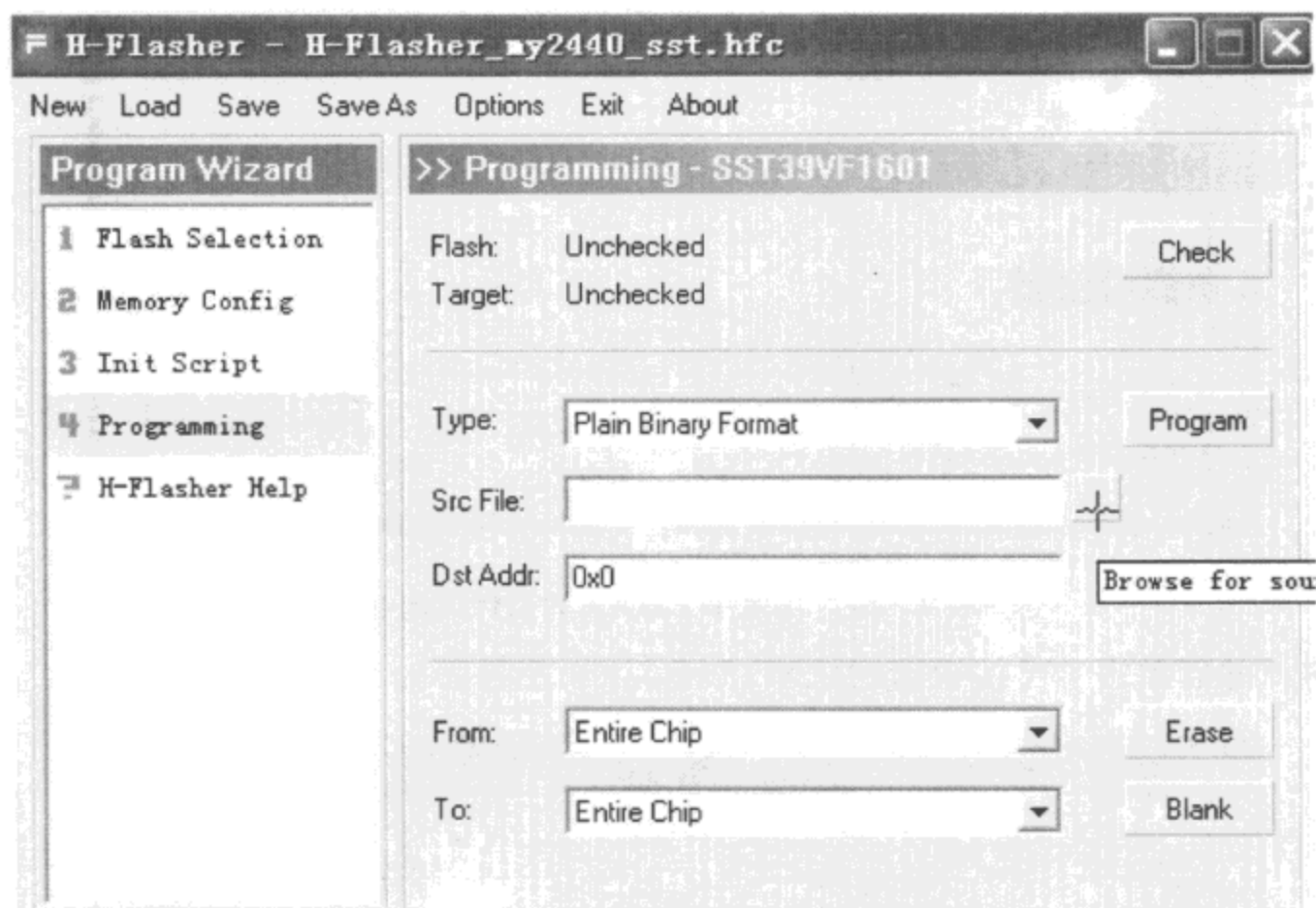


图 2-22 启动 H-Flasher

(2) 单击 Check 按钮检测 Nor Flash 型号。

(3) 在 Type 下拉列表中选择 Plain Binary Format 选项。

(4) 在 Src File 中填入要烧写的程序的路径和名称。

(5) 在 Dst Addr 中填入烧写的目标地址为 0x0。

(6) 单击 Program 按钮即可开始进行烧写。

注:单击 Erase 按钮可以擦除整个 Nor Flash。

如果使用 H-Flasher 不能进行烧写或者烧写出错,请按下述 4 的步骤将其恢复。

4. 安装 GIVEIO 驱动,烧写裸机程序到开发板的 Nor Flash 中

(1) 请以系统管理员的身份登录 WindowsXP,复制 sjf24x0\giveio.sys(sjf24x0 软件需要读者自行上网免费下载)到 C:\WINDOWS\system32\drivers 目录。

(2) 打开“控制面板→添加硬件→”,按照向导进行操作:

Step1:开始安装,如图 2-23 所示。

Step2:如图 2-24 所示,单击“是,我已经连接了此硬件”单选按钮,这时不必连接实际的 JTAG 板。

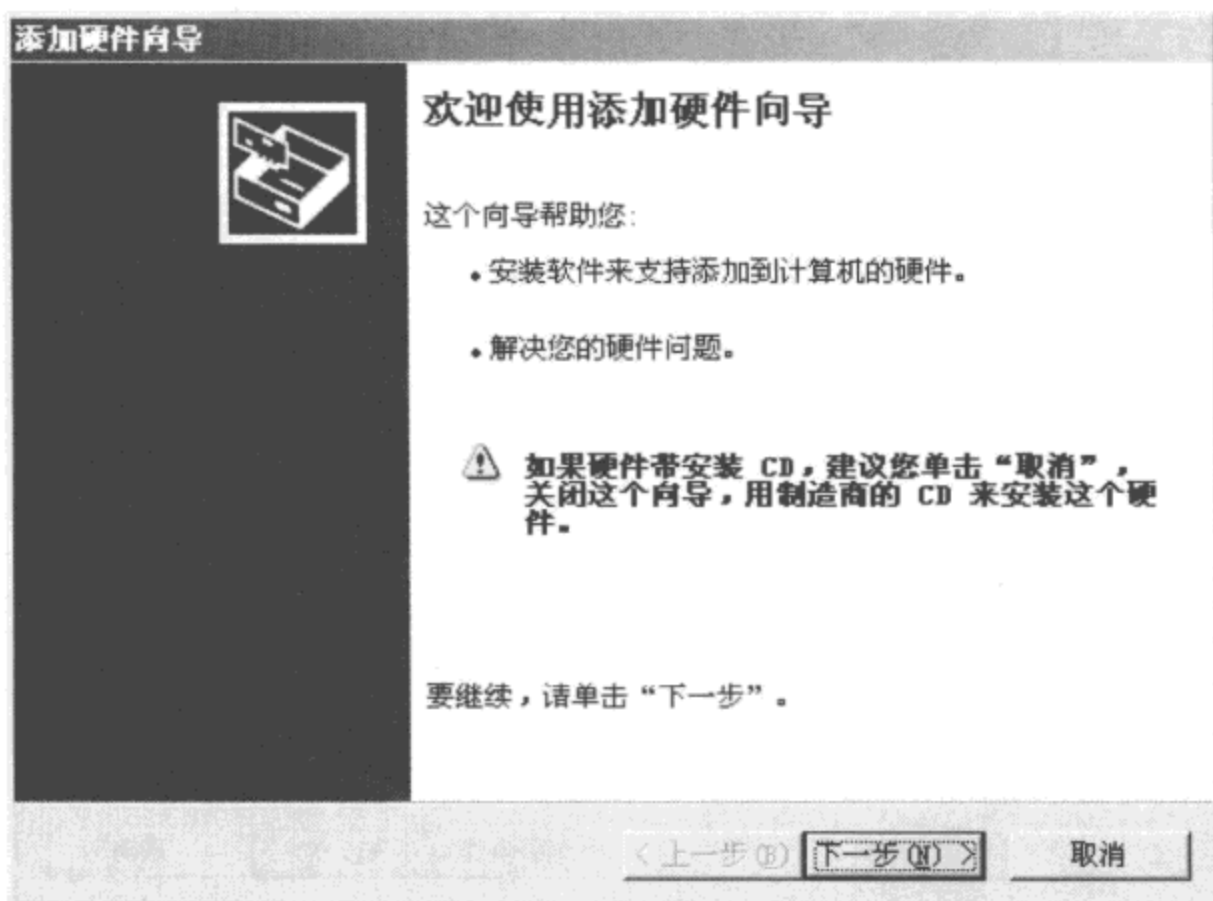


图 2-23 添加硬件

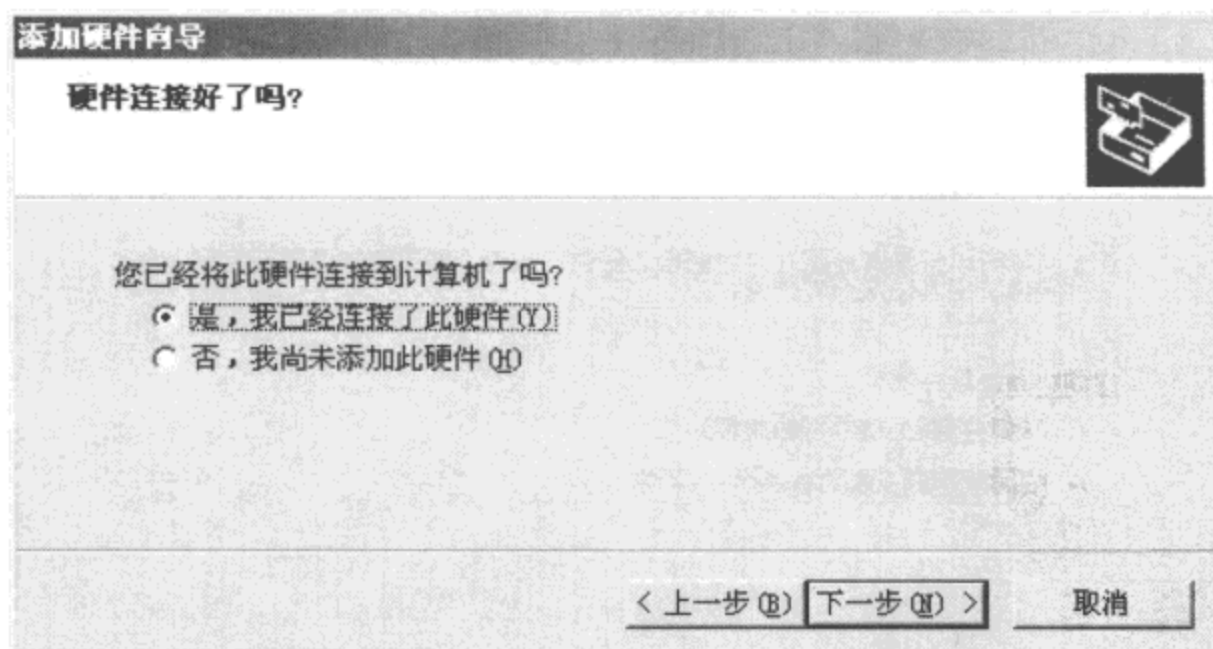


图 2-24 确认已连接硬件

Step3: 如图 2-25 所示，选择“添加新的硬件设备”选项。

Step4: 点选“安装我手动从列表选择的硬件”单选按钮，如图 2-26 所示。

Step5: 不选任何选项，直接单击“下一步”按钮，如图 2-27 所示。

Step6: 不选左右两侧列表中的任何选项，直接单击“从磁盘安装”按钮，如图 2-28 所示。

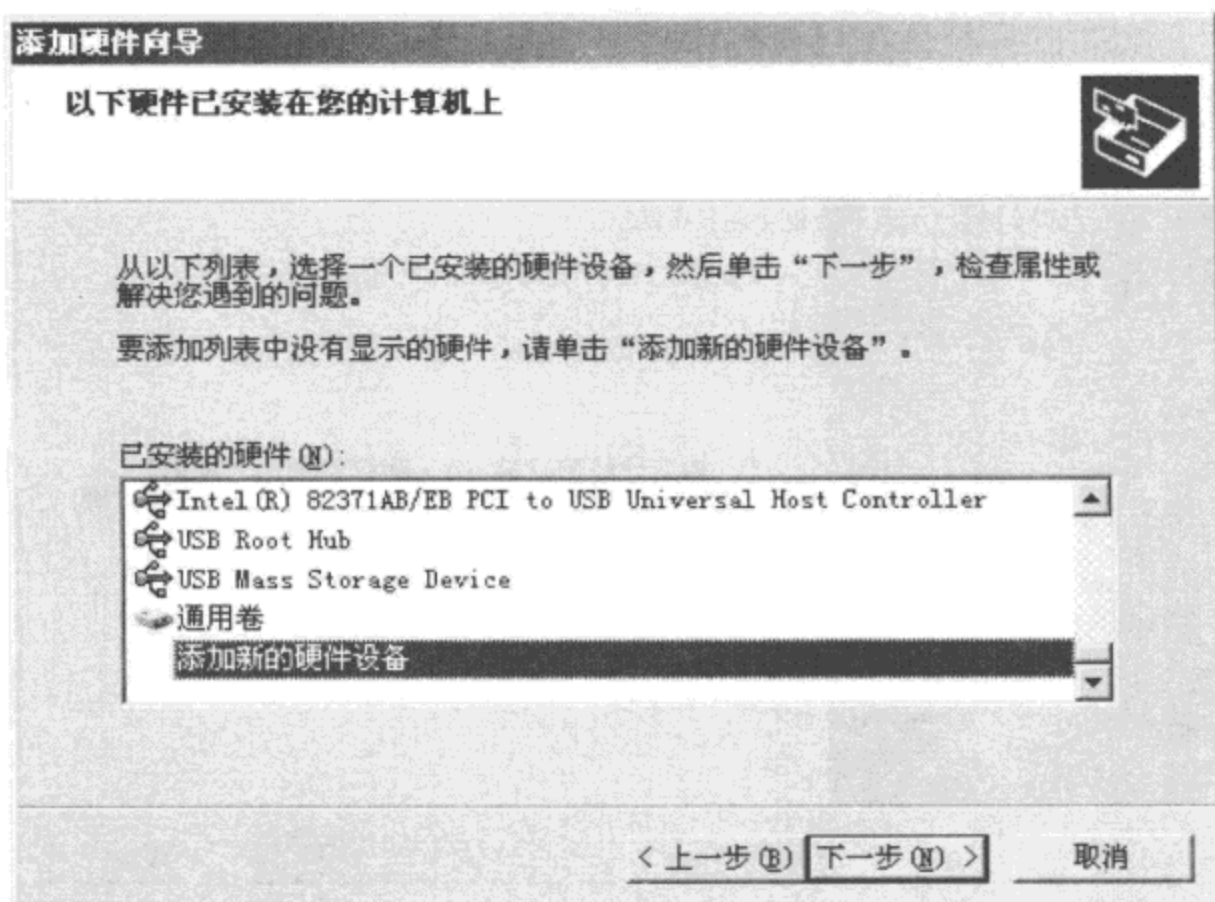


图 2-25 确认添加硬件

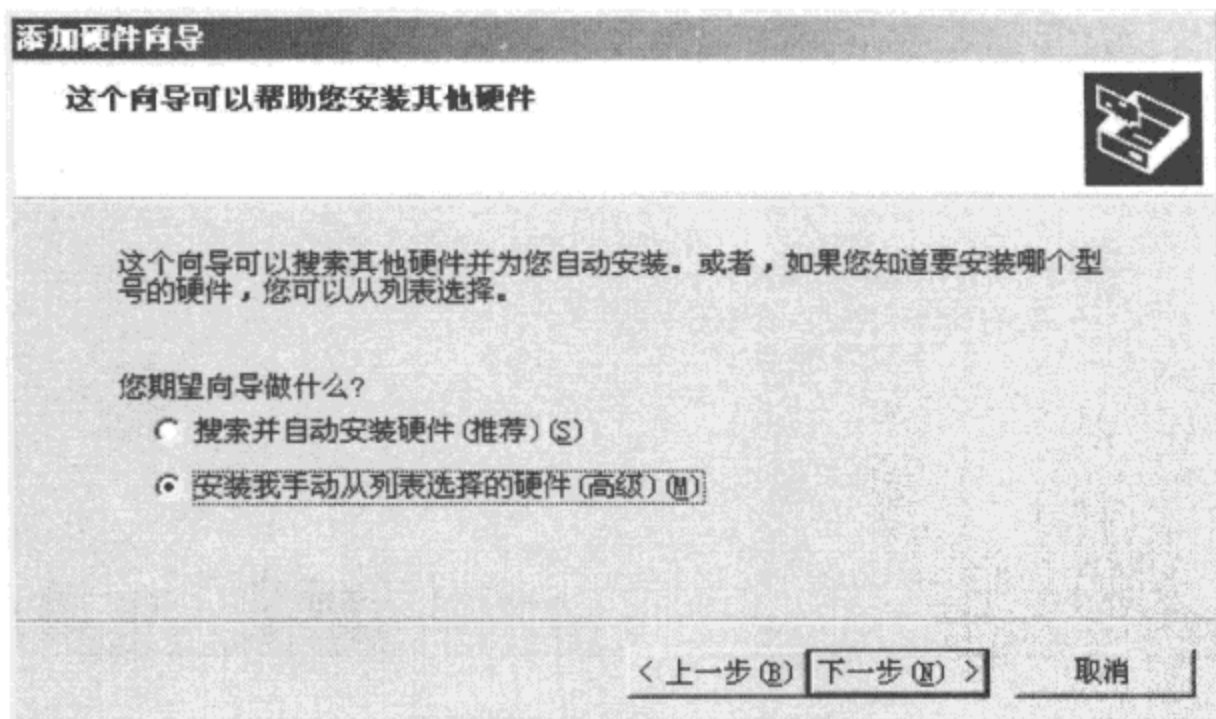


图 2-26 确认手动安装

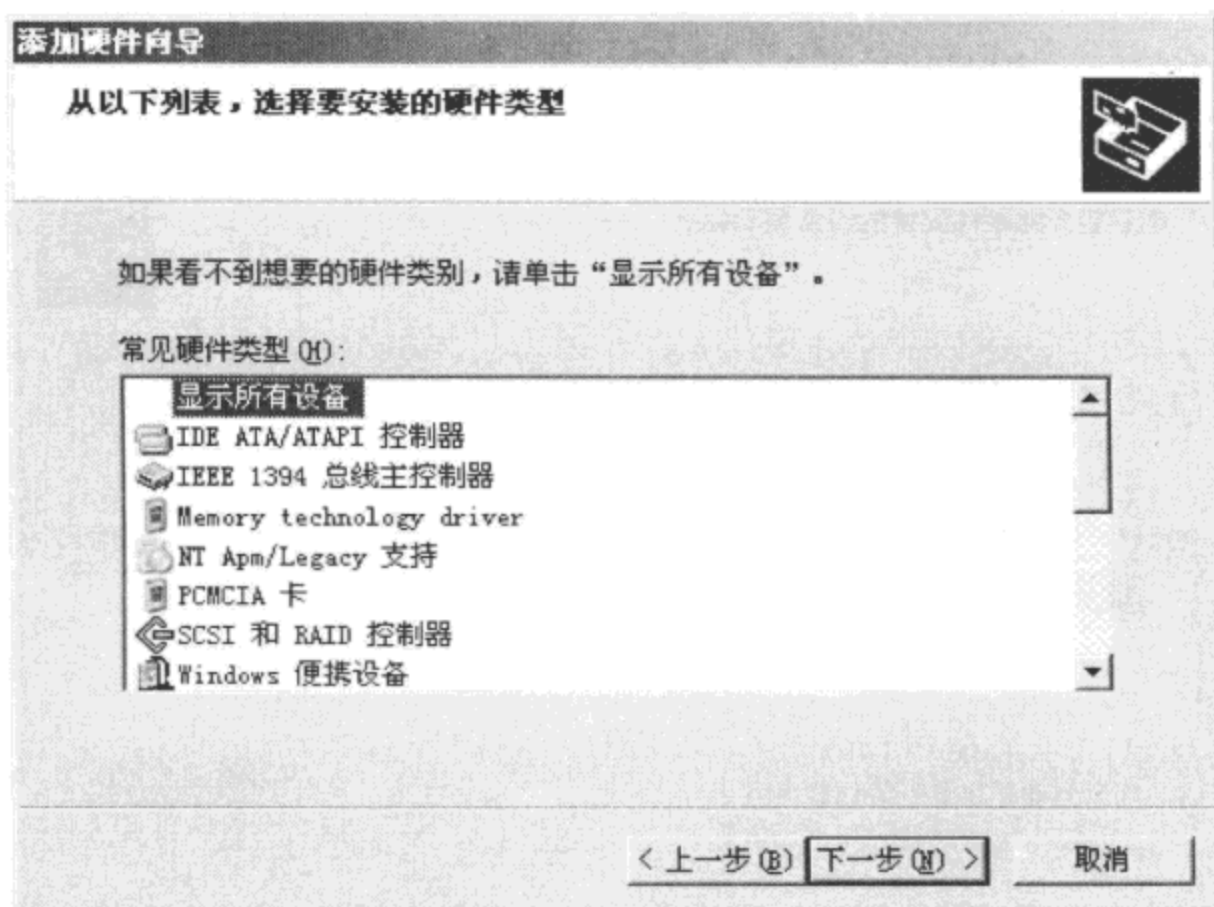


图 2-27 选择硬件类型

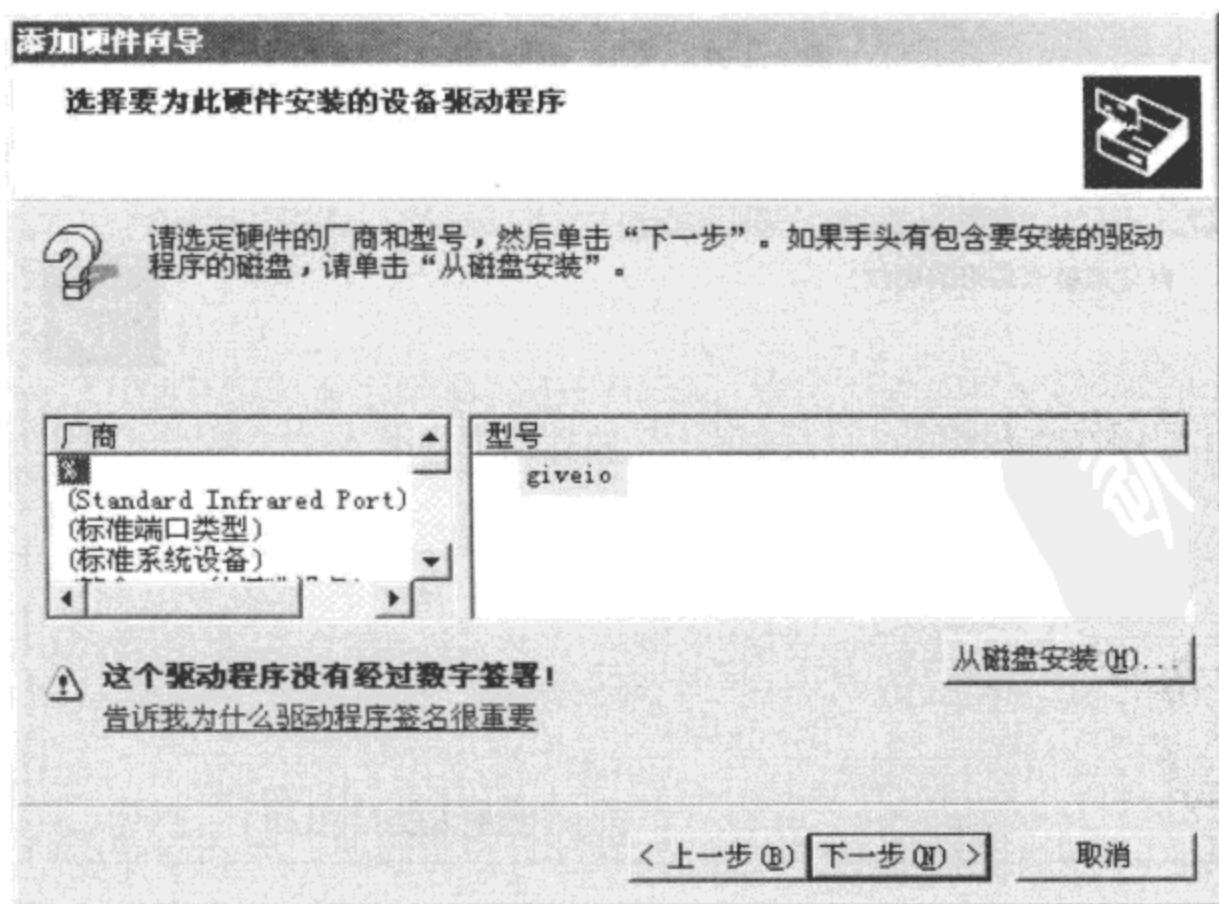


图 2-28 选择设备

Step7:选择要安装的驱动文件 giveio. ini(位于 sjf24x0 目录)。

Step8:单击“下一步”按钮,如图 2-29 所示。

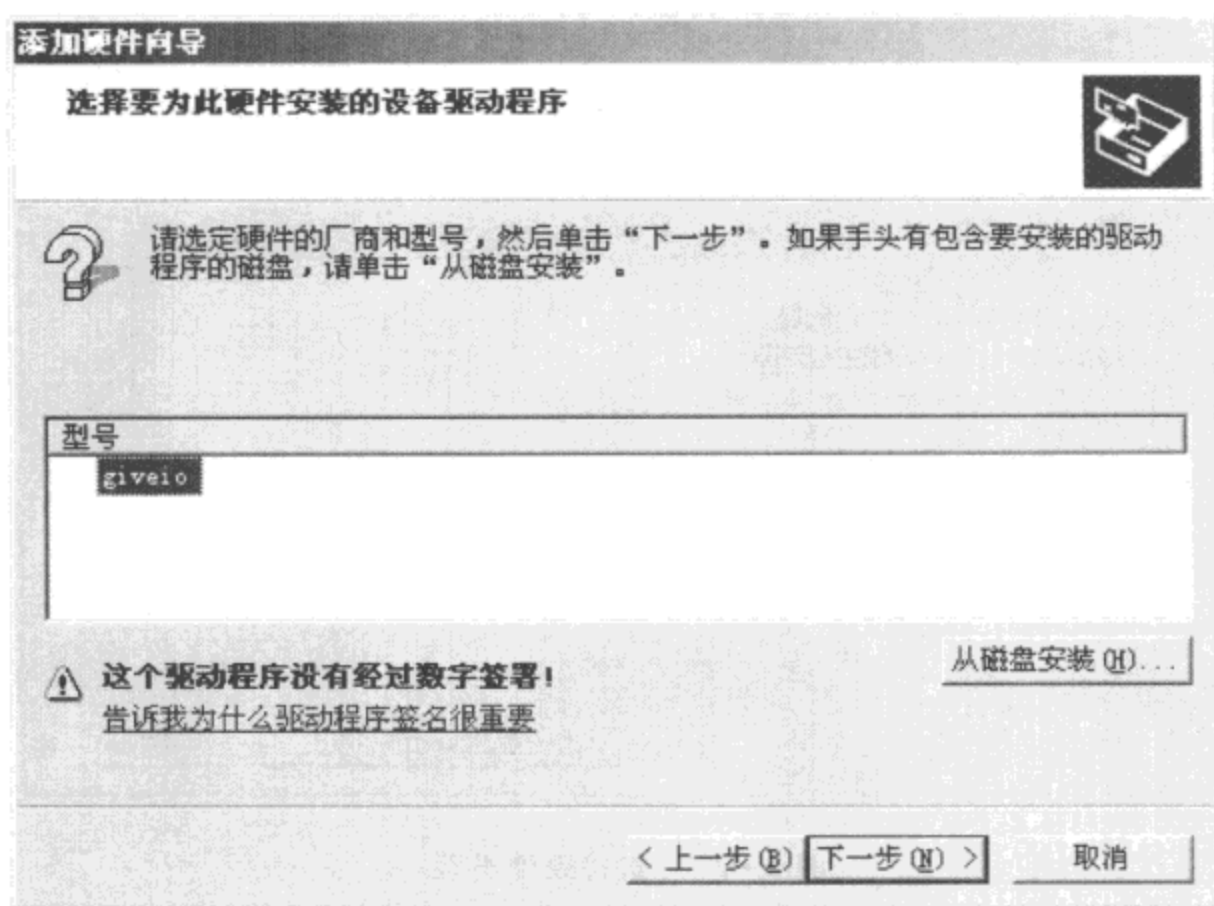


图 2-29 选择 giveio 硬件

Step9:单击“下一步”按钮,如图 2-30 所示。

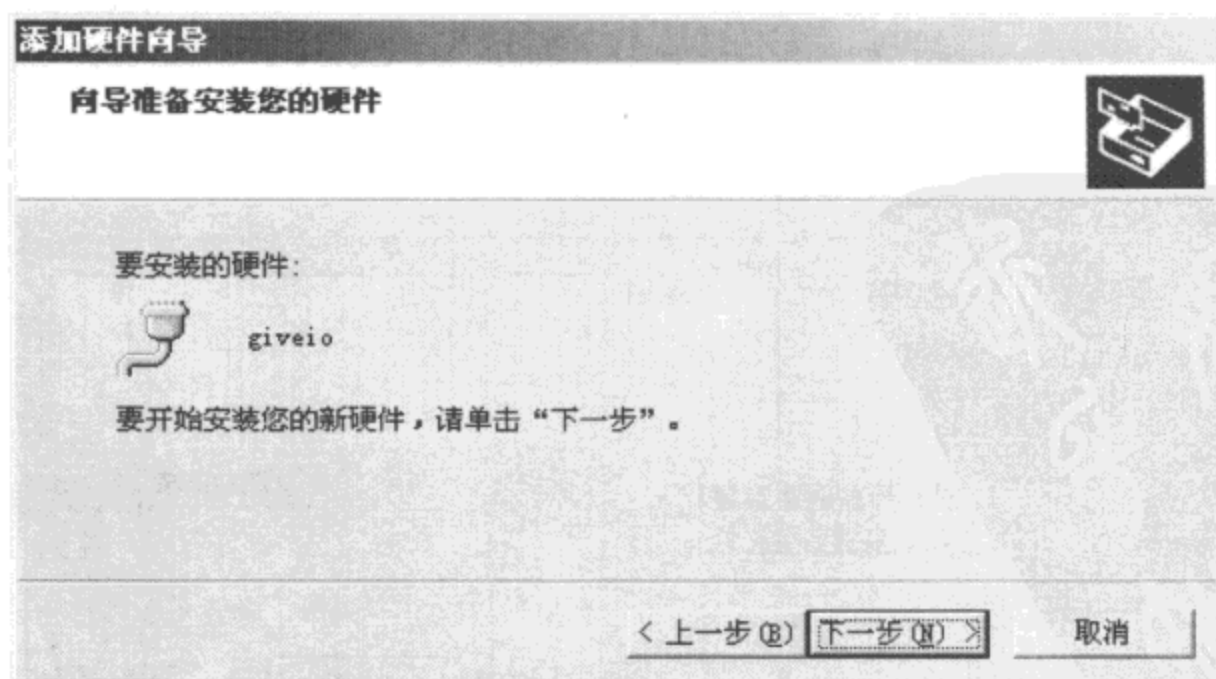


图 2-30 确认选择

Step10: 安装成功, 如图 2-31 所示。

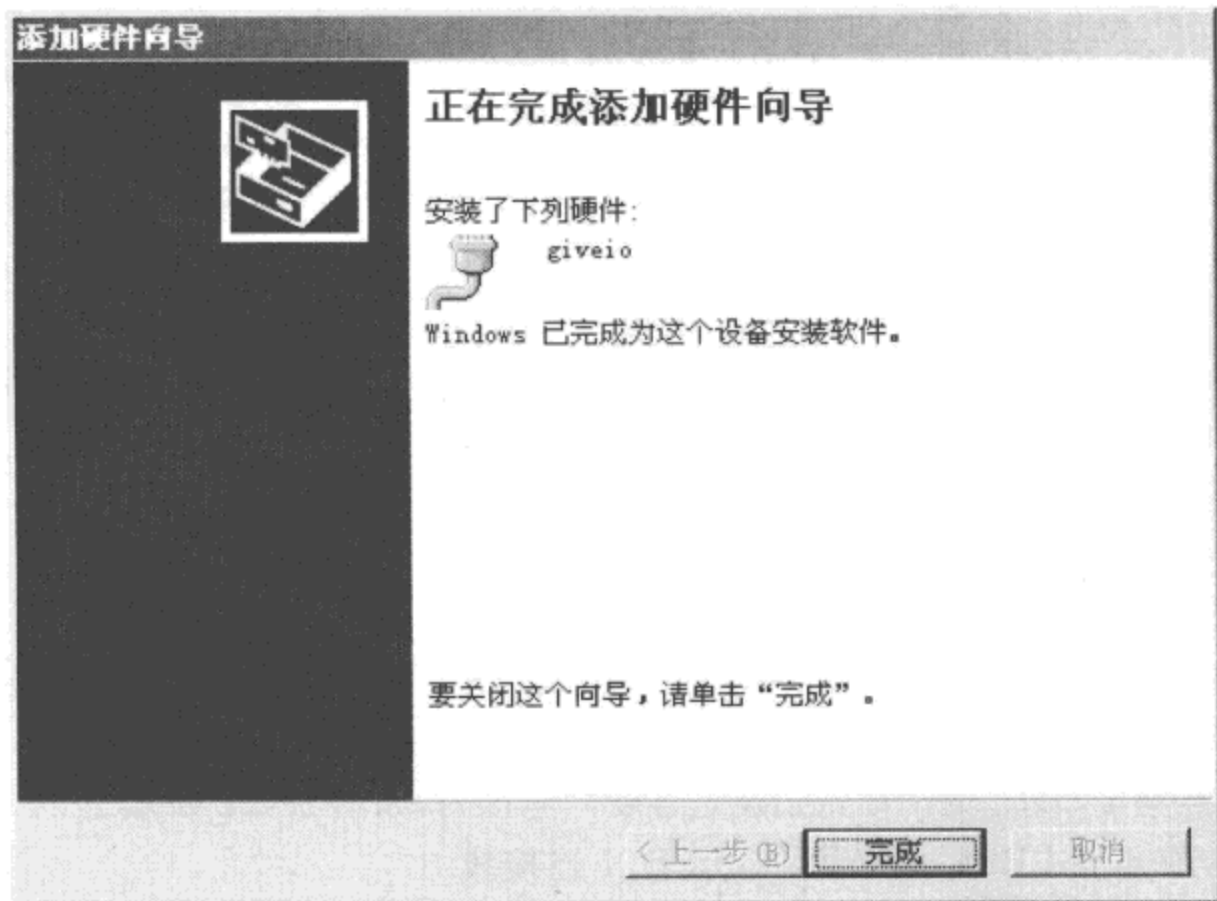


图 2-31 完成安装

(3) 烧写可执行文件到开发板的 nor flash 中。

- ① 在命令行提示符下, 进入 sjf24x0 目录。
- ② 在命令行提示符下, 输入 oflash led_on, 执行烧写命令。
- ③ 依次输入:
 - 1
 - 2
 - 1
 - 0

将程序通过 PC 并口烧写到开发板的 nor flash 的 0 地址。

2.3.2 软件控制(驱动)硬件的编程原理

每一种硬件在其控制器芯片上都会有物理的寄存器(注意这里的寄存器不是指的 CPU 内部的寄存器 R1 等, 而是指的硬件芯片上的存储单元, 在 ARM 体系下, 这些存储单元与内存进行统一编址, 可以被 CPU 通过访存指令, 像访问内存一样去访问), 这些寄存器通常分为 3 种类型: 命令寄存器、状态寄存器、数据寄存器。程序控制硬件的办法通常是: 程序通过 str 指令向命令寄存器写入合适的内容, 就可以完成对硬件进行配置的操作或者要求硬件进行某种

物理操作。到此为止,软件就完成了所有它该做的事情,之后硬件会自动完成相应操作,在硬件完成操作后,程序又可以通过 ldr 指令从数据寄存器中获得想要的数,或者从状态寄存器中获知硬件的状态。可见,程序控制硬件,简单地说,其实就是程序对硬件的寄存器进行读写操作,命令硬件完成操作,获取硬件状态和数据,仅此而已。这里的关键是:某个硬件寄存器的内存地址是多少?为使硬件执行某个操作,应当向哪个寄存器写入什么值?这些都是程序员需要解决的问题,而这些问题的解决,关键在于程序员能:

(1) 理解要控制的硬件的运作机制。

(2) 能熟练查阅硬件的手册(手册中会指明寄存器的内存地址以及寄存器各种取值的含义)。

(3) 能看懂硬件的连线原理图。

2.3.3 裸机硬件控制程序实例

现在我们来看一个最简单的裸机硬件控制程序(位于光盘的\work\armarch\ledcircle),它可以控制 LED 灯的亮灭。

如何才能点亮 LED 灯呢?首先必须先看硬件连接图,如图 2-32 所示。

显而易见,要点亮 LED1,则必须在 nLED_1 连接线上输出低电平,如图 2-33 所示。

要在 nLED_1 连接线上输出低电平,就必须让 CPU 的 GPB5 引脚输出低电平。

如何才能让 CPU 的 GPB5 为低电平?通过查阅 S3C2440 的硬件手册的第 9 章可知,需要将地址为 0x56000010 的这个寄存器的 bit11 和 10 设置为 01,从而将 GPB5 这个引脚配置为输出,然后将地址为 0x56000014 的这个寄存器的 bit5 写为 0,这样 CPU 的 GPB5 引脚就会输出低电平。

这就对应于示例代码中的如下关键代码。

```
#define GPBCON      (*(volatile unsigned long *)0x56000010)
#define GPBDAT      (*(volatile unsigned long *)0x56000014)
#define LEDS (1<<5|1<<6|1<<7|1<<8)
GPBCON = 0x00015400;
GPBDAT = (GPBDAT&(~LEDS)) | (1<<6|1<<7|1<<8);
```

表 2-3 为寄存器 GPBCON 与 GPBDAT 描述。

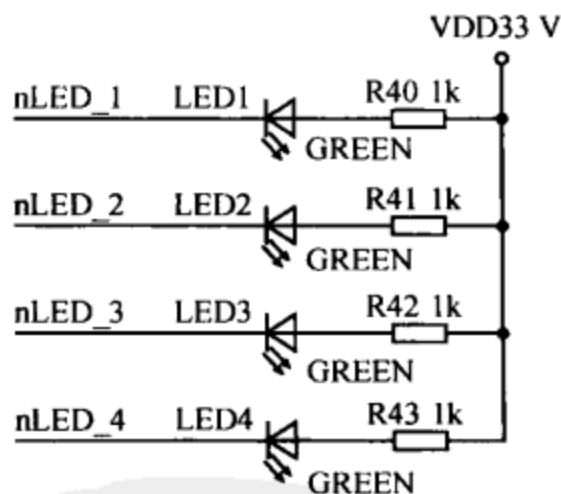


图 2-32 LED 灯硬件连接图

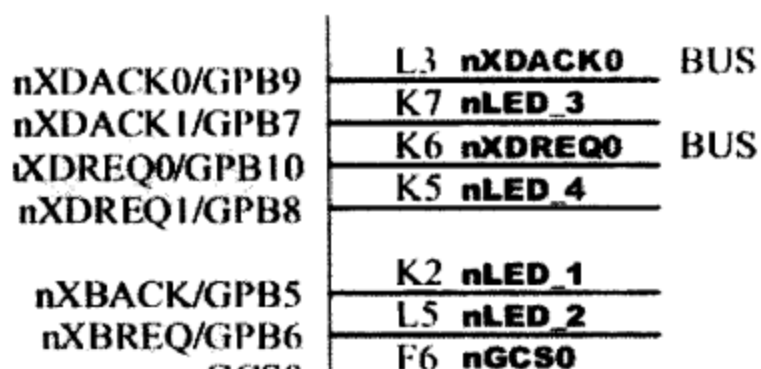


图 2-33 LED 灯的 CPU 引脚连接图

表 2-3 寄存器 GPBCON 与 GPBDAT

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|--|-------------|-------|
| GPBCON | 0x56000010 | R/W | GPB 端口配置寄存器 | 0x0 |
| GPBDAT | 0x56000014 | R/W | GPB 端口数据寄存器 | 未定义 |
| GPBCON | 位 | 描 述 | | |
| ... | ... | ... | | |
| GPB8 | [17:16] | 00 = 输入 01 = 输出 10 = nXDREQ1 11 = 保留 | | |
| GPB7 | [15:14] | 00 = 输入 01 = 输出 10 = nXDACK1 11 = 保留 | | |
| GPB6 | [13:12] | 00 = 输入 01 = 输出 10 = nXBREQ 11 = 保留 | | |
| GPB5 | [11:10] | 00 = 输入 01 = 输出 10 = nXBACK 11 = 保留 | | |
| ... | ... | ... | | |
| GPBDAT | 位 | 描 述 | | |
| GPB[10:0] | [10:0] | 当 GPB 端口设置为输入时,GPB 寄存器中对应的位的值就是对应引脚端口的状态。当 GPB 端口设置为输出时,设置引脚端口对应位就是设置对应状态,当引脚设置为功能引脚时,寄存器中值不能确定。 | | |

示例代码中的 readme.txt 中,说到:

for running on real board, you need do following 3 things:

1. Change Target from ARM7TDMI to ARM920t.
2. Change load address from 0x8000 to 0x30000000.
3. Change image layout for placing init.o(INIT2440) at the first.

做1的原因是开发板的CPU——S3C2440是ARM920t的内核,所以编译器编译时必须匹配,如图2-34、图2-35所示。

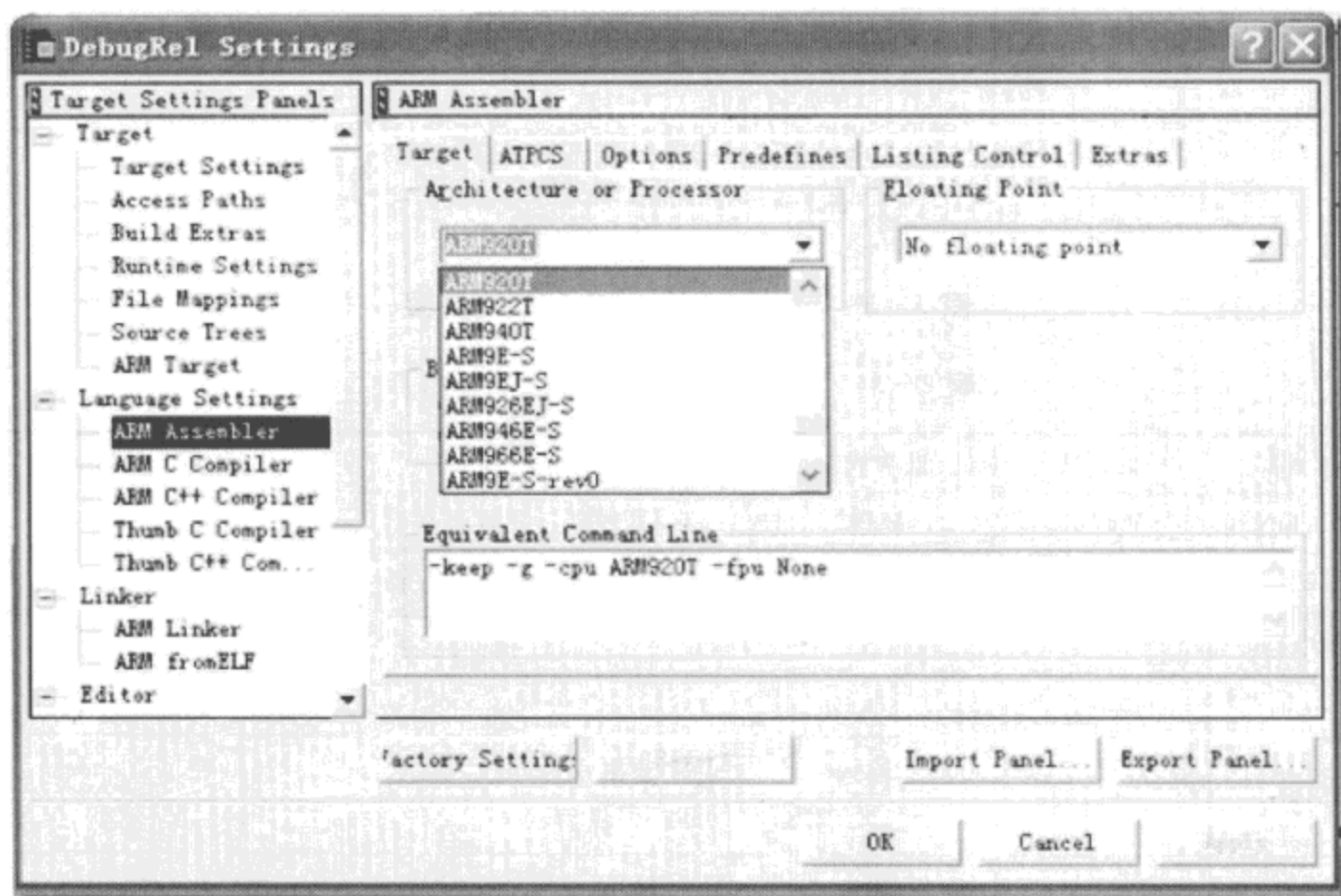


图2-34 设置汇编器目标

做2的原因是开发板的RAM位于0x30000000~0x34000000地址(共64MB),程序必须被调试器加载到RAM才能运行。

特别说明:配置中的0x30000000称为程序的期望加载地址,简称加载地址。运行地址与加载地址是很重要的两个概念,请大家一定要弄清楚。运行地址是给编译器看的,通过看运行地址编译器就能计算出程序中各个标号、变量、函数等在内存中的绝对地址,从而完成涉及地址的指令的正确编译,如图2-36所示。而“加载地址”是给调试器或者操作系统看的,它的作用是让调试器或者操作系统将程序加载到正确(期望)的内存地址。通常情况下,代码段的这两个地址是相等的,数据段则不一定。

做3的原因是init.o的INIT2440段的代码是首先运行的代码,因此必须放在整个二进制程序文件的最开头,如图2-37所示。

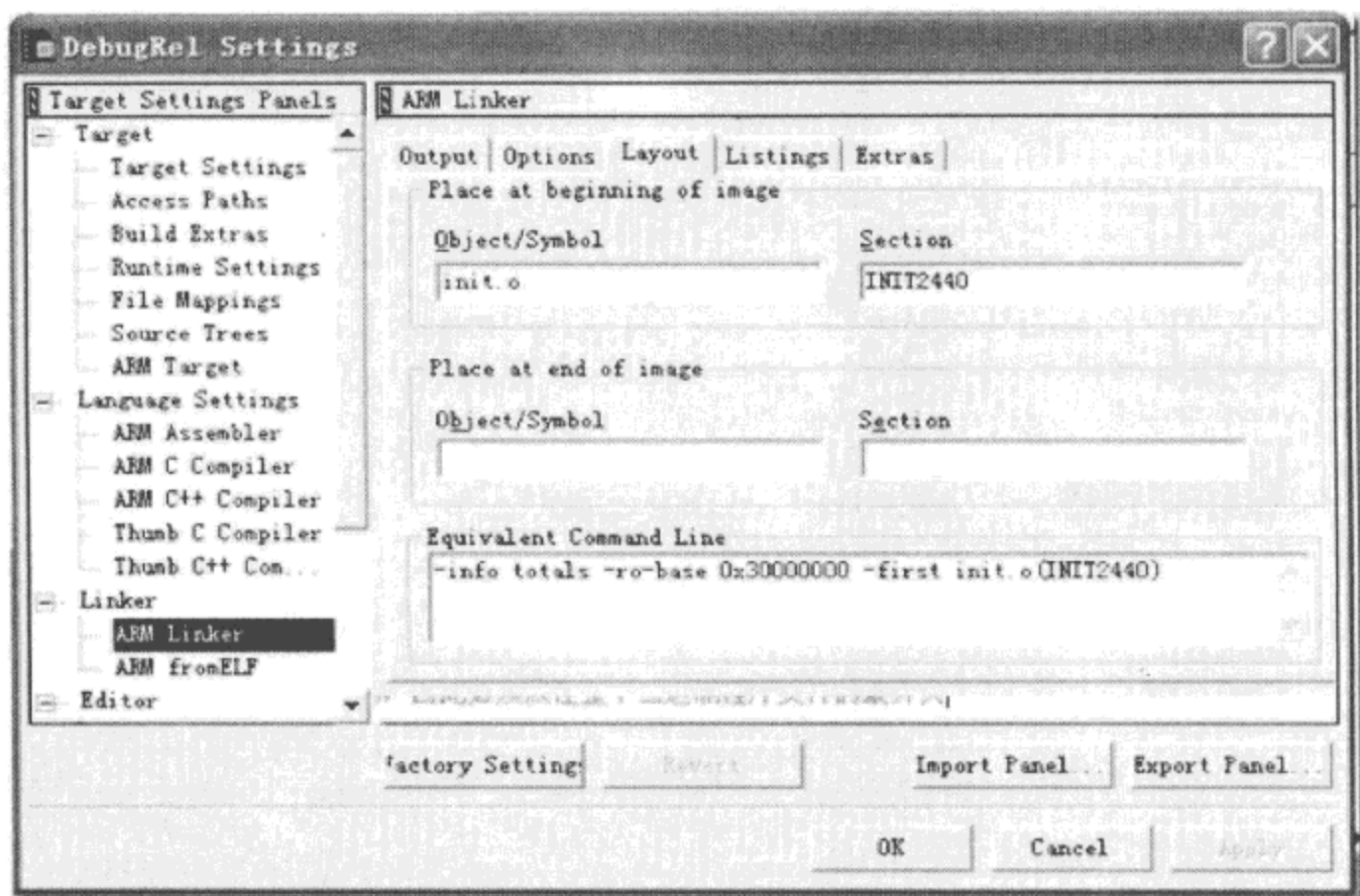


图 2-37 设置链接选项——二进制文件布局

2.3.4 启动例程

通常 PC 在开机之后,会进入带有 PC 厂商信息的 BIOS 画面,并且会显示出当前 PC 的硬件信息,比如内存大小、CPU 信息等,它其实是 PC 启动之后运行的第一段程序,它主要完成一些基本硬件初始化操作和硬件检测工作,保证拥有操作系统正常运行的软硬件环境,随后会加载并且启动操作系统。该段小程序是烧制到主板上的 BIOS 存储硬件里的。由此可见计算机系统在启动过程中必须依赖软硬件,在嵌入式系统中同样需要软硬件互相协调来完成启动过程。

1. 嵌入式系统启动——硬件支持

在嵌入式系统中,通常并没有像 BIOS 那样的固件程序,而是将用于引导系统的二进制映像文件(image 文件)烧写到只读的 ROM 中,系统启动之后从 ROM 里加载并执行该映像文件。根据只读 ROM 类型的不同,启动方式也不尽相同。

嵌入式系统通常有两种启动方式:

(1)“硬盘”方式启动。“硬盘”方式启动是指启动程序被烧写在“硬盘”中,系统上电后 CPU 读取并执行“硬盘”中的指令数据,由于“硬盘”不支持随机寻址(支持向任意有效地址进行读写操作,硬盘里最小的寻址单元是扇区),并且读写速度太慢,通常是将“硬盘”中的引导程

序指令数据复制到 RAM(随机访问存储器)中,然后执行这些指令。

嵌入式系统中通常不使用 PC 的硬盘作为文件存储器,而是使用类似硬盘的一些小容量 Flash,如 Nandflash。S3C2440A 处理器支持从 Nandflash 启动,这是由于 S3C2440A Nandflash 控制器中含有一个特殊的 RAM - Stepping stone,如图 2-38 所示。

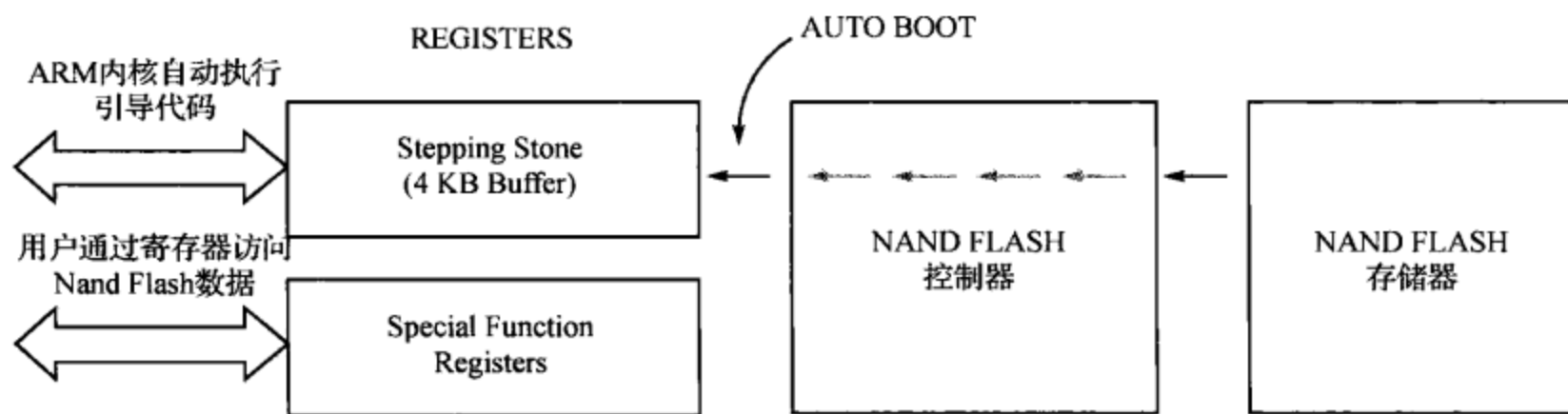


图 2-38 Stepping stone 原理图

当系统选择从 Nand Flash 启动时,硬件会完成以下操作:

- ① 通过 Nand Flash 控制器将 Nand Flash 中前 4KB 的指令数据复制到 Stepping stone 中。
- ② 将 0x0 地址映射到 Stepping stone 所在地址 0x40000000。
- ③ PC 从 0x0 地址处取址执行。

(2) ROM 方式启动。以这种方式启动的系统通常会有一个专门用来存放启动程序的存储固件,当系统以该方式启动时,CPU 直接从存储固件里运行启动程序。启动程序一般不会被擦除并且要求掉电时数据不能丢失,因此该存储固件通常是 XIP(eXecute In Place 片内可执行,代码不被压缩,并且支持随机寻址)类型的 ROM(掉电非易失只读存储器)。嵌入式系统中经常使用 Nor Flash 作为启动程序存储固件。S3C2440 同样也支持这种启动方式。

2. 嵌入式系统启动——软件支持

在嵌入式系统启动时,通常也有类似于 BIOS 的启动程序,该启动程序被称为 Boot Loader,由于 Boot Loader 是被编译生成的,并且它的正确执行通常和系统硬件密切相关。

(1) 二进制映像文件(image)。二进制映像文件是指烧写到 ROM 中的 bin 文件,也称为 image 文件。通常 image 文件是由编译器将源码编译而成的可执行二进制文件。源码中的语句,初始化变量,未初始化变量,和初始化为 0 的变量分别被编译成对应的操作指令和变量数据域,这些编译成的操作指令和变量数据域被链接器统一编址进 image 文件,作为 image 文件的输入。用户可以指定这些 image 文件的输入数据的属性,可以为只读的 RO(Read - Only),可读写的 RW(Read - Write)和初始化为 0 的 ZI(Zero - Initial)。通常源码文件中含有以下属性数据段:

- ① 只读属性 RO 的操作指令,如控制语句、表达式。
- ② 只读属性 RO 的数据段,如常量。
- ③ 可读写属性 RW 的数据段,如变量。
- ④ 初始化为 0 的数据段 ZI 属性,如:初始化为 0 的结构体。

多个代码源文件具有相同属性的数据段为了方便存储器对其存储空间访问权限进行管理,通常被链接器放置在 image 的相同位置,具有相同属性的输入数据段组成 image 文件的输出域。链接器允许用户指定每个 image 文件的输出域的起始地址,如图 2-39 所示。

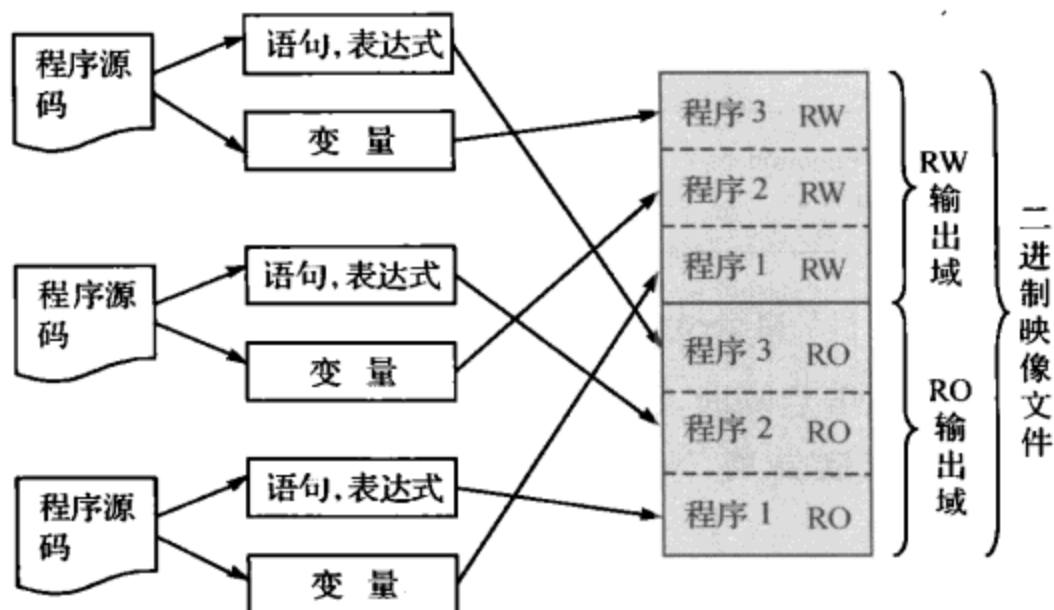


图 2-39 image 文件结构图

(2) RAM 中的执行程序。外部存储设备上的 image 没有被加载到内存中时,和普通文件一样,是不能被执行的, image 要想被 CPU 执行,必须要被程序加载器 Load 到内存中,并且设置好执行环境。

由于执行程序是内存中的 image 文件,因此二者内容是相同的,但还是有一些差别:

① image 文件是存储在外部存储设备里的(嵌入式系统中通常是 ROM,如:Nandflash, Norflash),而执行程序只能运行在内存 RAM 中。

② 程序中初始化为 0 的变量(ZI 数据段,也称为 BSS 数据段)。在 image 文件里是不存在的变量,要想被当前程序访问必须指定一个有效的内存地址,而未初始化的变量通常被设置为 0,因此全部为 0 的数据是没有必要放到 image 文件里,这样反而增加了 image 文件的体积。

由此可见, image 文件被加载到内存之后,还要为 ZI 段准备地址空间来存放初始化为 0 的 ZI 数据段。

ZI 段准备操作如下:

- ① 准备 ZI 数据段地址空间。
- ② 将空间内容清 0。

对于运行在操作系统中的程序,该项工作可由操作系统的运行环境(程序加载器)来帮之

实现,而对于没有操作系统的裸机程序,这项工作只能由 RO 段的代码自己实现,它也是启动程序最主要的一项工作之一。

3. 嵌入式系统启动过程

嵌入式系统启动过程就是指处理器从复位进入到操作系统或程序能够运行的状态的过程。该过程是由系统加载引导程序(Bootloader)来实现的,它主要完成以下工作:

- 初始化必要硬件,如关闭看门狗,初始化内存 SDRAM,提供硬件执行环境。
- 初始化 C 程序软件执行环境。
- 将启动代码从 ROM 复制到 RAM 中。
- 跳转到 RAM 里继续执行启动代码。

(1) 初始化必要硬件。

① 关闭看门狗。看门狗是许多嵌入式系统里带有的硬件装置,主要用于在外界环境,如噪声、电磁干扰、系统错误等造成的系统故障时自动重启系统,恢复系统正常工作。通常在系统启动过程中,直接将其关闭,详情查看看门狗控制器章节。

② 初始化内存。虽然程序可以在外部存储器 ROM 里执行,但是由于 ROM 是只读的,不能像内存一样执行写入操作,而在 C 程序中必须要求将变量安放到可以执行写入操作的内存中,因此在系统启动之前必须要对内存进行初始化。

(2) 初始化 C 程序软件执行环境。

① 初始化 C 程序栈指针。C 程序执行过程中,要进行出栈、入栈操作,在 C 程序执行之前,必须要将栈指针初始化为有效内存地址。

② 清零 ZI 段。因为 ZI 段并不存在 image 中,所以需要程序根据编译器给出的 ZI 段地址及大小来将相应的 RAM 区域清零。RO 段中的指令只有完成了上述两项工作之后,C 程序才能正常执行访问变量,否则只能执行不含变量的只读指令代码。

(3) 将启动代码从 ROM 复制到 RAM(内存)中。

由前面知识可知,启动程序通常存放在 ROM 中。ROM(只读存储器)的特点是里面数据是只读的,不能执行修改写入操作,在程序执行时,对变量的赋值操作其实就是对内存执行写入操作,如果该程序在 ROM 里执行,变量的赋值操作是不能实现的。RAM(随机访问存储器)可以支持随机地址读写,因此程序代码通常是在 RAM 里被执行的,而内存是典型的 RAM,这样就要将代码从 ROM 复制到内存中。

在执行 ROM 里数据指令复制时需要知道下面 3 个参数:

- ROM 中数据指令开始地址。
- RAM 中数据指令目标存放地址。
- 要复制启动程序的大小。

其中,第一个参数是由 CPU 体系结构决定的,像 S3C2440 CPU,它在开机上电后会自动从 0x0 地址处取指运行,第二个参数由用户自己定义,可以将 ROM 里数据指令存放到任意有

第2章 ARM 编程进阶

效内存区域(保证有足够空间存放),第三个参数则需要借助编译器来实现了。

ADS1.2 集成开发环境下实现 ROM 到 RAM 复制。

① image 文件的加载时地址与运行时地址。image 文件的加载地址是指由程序源文件编译生成的 image 文件被加载到内存时的地址,该地址由链接器的参数选项指定,在 ADS1.2 开发环境下,可以在图 2-40 中方框中位置中设置。

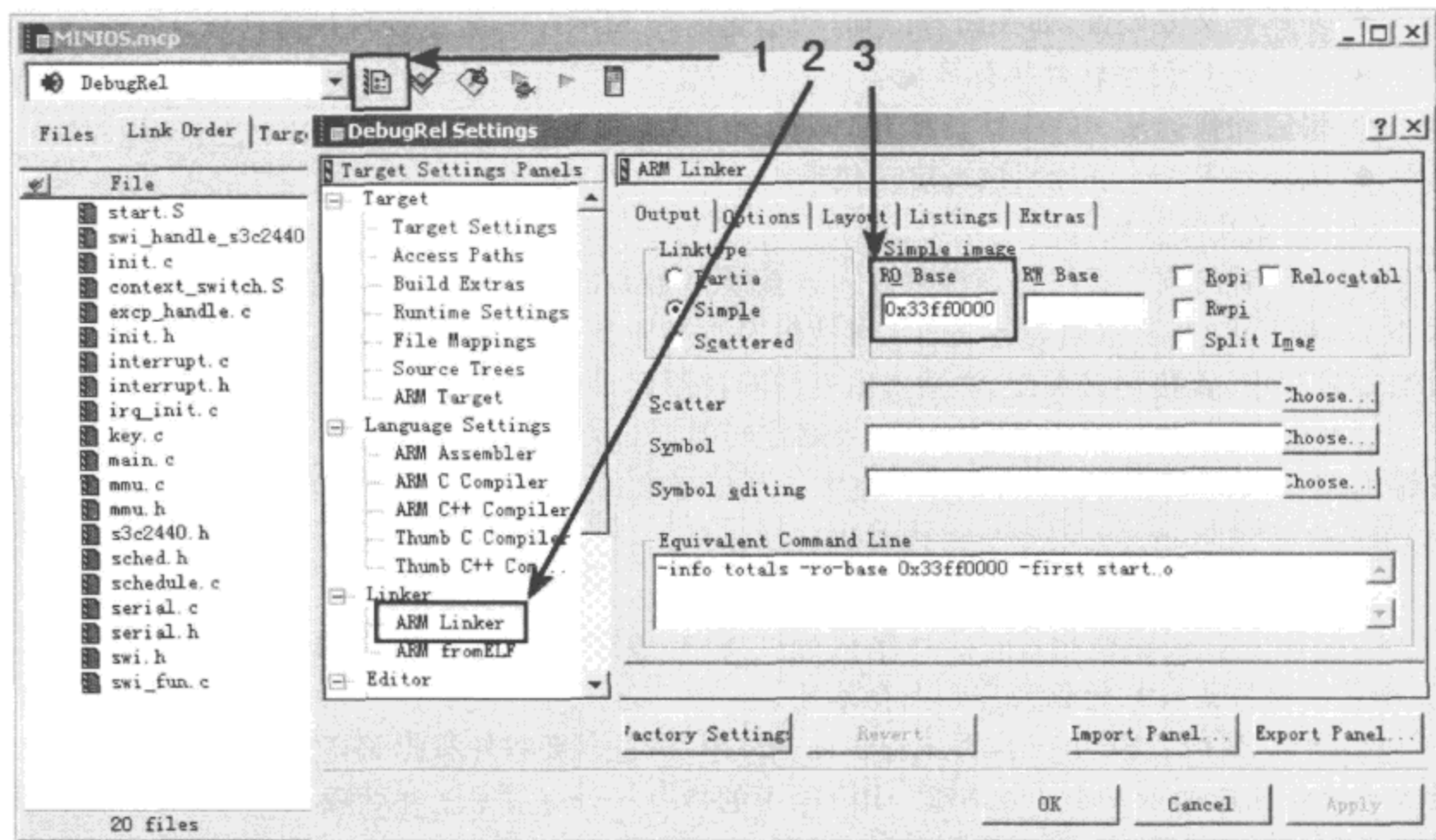


图 2-40 设置 image 文件加载地址

image 文件的运行时地址是指,当前 image 文件被加载到内存后,运行时的地址,也就是指令真正的内存地址。通常情况下,加载时地址和运行时地址是一致的,只要设置了其加载时地址其运行时地址也就确定了。但是通常编译生成的 image 文件会被烧写到 ROM 中,而在 CPU 上电后,会将 ROM 地址映射为 0x0 地址,CPU 自动从 ROM 开始处取指执行,也就是说指令在 ROM 中执行时,指令的实际地址并非真正的运行时地址,由于 ARM 指令中大部分指令是相对寻址的,这并不会影响大部分指令的执行。

② 链接器自动生成的符号变量。ADS1.2 链接器在对各数据段链接完成之后,会自动生成一些符号变量,用户程序可以通过引用这些符号变量取得 image 文件各输出域的信息,如表 2-4 所列。

表 2-4 ADS 中输出域符号变量

| 符 号 | 含 义 |
|----------------------------|---------------|
| Image \$ \$ RO \$ \$ Base | RO 输出域运行时起始地址 |
| Image \$ \$ RO \$ \$ Limit | RO 输出域运行时结束地址 |
| Image \$ \$ RW \$ \$ Base | RW 输出域运行时起始地址 |
| Image \$ \$ RW \$ \$ Limit | RW 输出域运行时结束地址 |
| Image \$ \$ ZI \$ \$ Base | ZI 输出域运行时起始地址 |
| Image \$ \$ ZI \$ \$ Limit | ZI 输出域运行时结束地址 |

有了以上链接器提供的符号变量,在程序里就可以动态获得链接后的各个输出域的起始、结束地址了,也就能得到每个输出域的大小了。在 ADS1.2 开发环境下使用链接器符号变量时要将符号变量使用“|”括起来,可以通过下面代码实现从 ROM 到 RAM 的复制。

```

IMPORT |Image $ $ RO $ $ Base|      ; 引入链接器自动生成符号变量 Image $ $ RO $ $ Base
IMPORT |Image $ $ ZI $ $ Limit|     ; 引入链接器自动生成符号变量 Image $ $ RO $ $ Base
copy_code                          ; 代码复制开始符号
mov r0, # 0x0                      ; R0 中为数据开始地址 (ROM 数据保存在 0 地址开始处)
ldr r1, = |Image $ $ RO $ $ Base| ; R1 中存放 RO 输出域运行地址
                                   ; 该值由符号变量 Image $ $ RO $ $ Base 取得
ldr r2, = |Image $ $ ZI $ $ Limit| ; R2 中存放 ZI 输出域结束地址
                                   ; 该值由符号变量 Image $ $ ZI $ $ Limit 取得
sub r2, r2, r1                     ; R2 = R2 - R1, 得出待复制数据长度
bl  CopyCode2Ram                   ; 将 R0, R1, R2 三个参数传递给 CopyCode2Ram 函数执行复制

```

(4) 跳转到 RAM 里继续执行启动代码。将代码从 ROM 复制到 RAM 之后,当前启动代码就有了两份,这时要让 CPU 去执行 RAM 里的启动代码,通常这会使用伪指令 ldr 来实现该功能:

```

LDR PC, = label
label
MOV R0, # 0      ; 内存中代码

```

上述加载伪指令 LDR PC, = label 是将 label 标签处的运行时地址赋值给 PC(运行时地址由用户指定),同时该指令的跳转范围可以寻址 4GB 地址空间,可以实现从 ROM 地址空间跳转到 RAM 地址空间。上述指令执行完之后 CPU 就会跳转到内存中启动代码 label 处取指执行。

4. S3C2440 的两种启动方式

由前面知识可知,通常嵌入式系统有两种启动方式:硬盘启动和 ROM 启动。S3C2440 内

核支持上述两种启动方式。

(1) S3C2440 的 Nandflash 启动。当系统选择从 Nandflash 启动之后,硬件自动将 0x0 地址映射到 Stepping stone,同时将 Nandflash 前 4 KB 代码复制到 Stepping stone,由于 Stepping stone 大小只有 4 KB,而系统启动程序大小往往超过 4KB,这就需要将全部的启动代码从 Stepping stone 搬运到空间更大的内存中运行。又由于内存、Nandflash 等硬件还没有被驱动起来,不能正常使用,这就要保证 Stepping stone 里的代码必须要先对搬运过程中相关硬件进行初始化,然后执行搬运工作,搬运完成之后,跳入到内存中继续运行,以上工作都要在 Stepping stone 里实现,如图 2-41 所示。

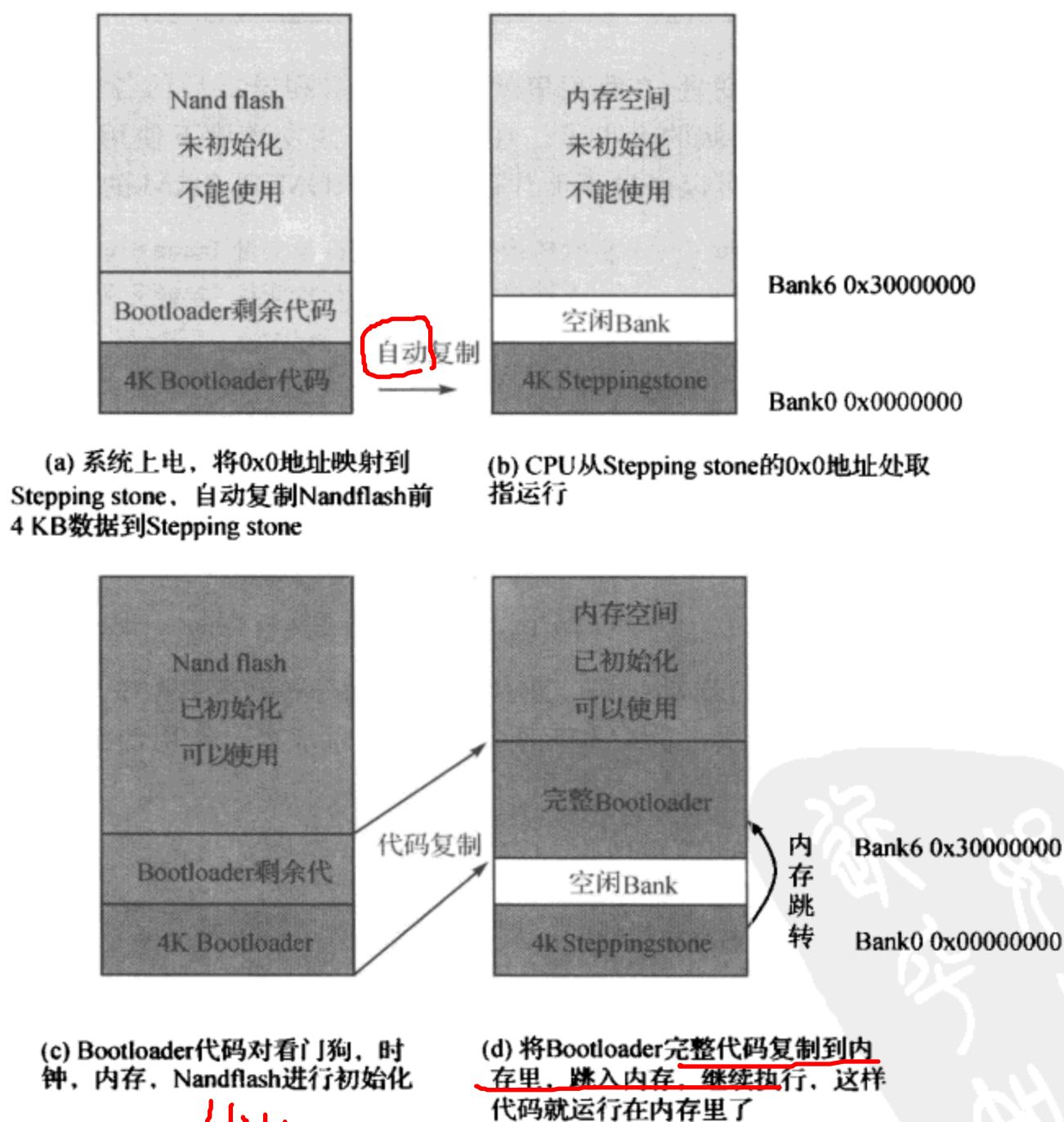


图 2-41 Nandflash 启动过程

(2) S3C2440 的 Norflash 启动。通常开发板上通常焊接 Norflash 来实现 ROM 启动方

式,由于 Norflash 支持随机寻址和片内执行 XIP,并且 Norflash 的容量一般足够大来存放 Bootloader,但是由于 Norflash 是 ROM,虽然可以像内存一样进行读操作,却不可以像内存一样支持写入操作,Bootloader 中 image 文件的 RW 段和 ZI 段,只有放入到 RAM 中才可以正常执行写入,因此从 Norflash 启动也需要代码搬运工作。

2.4 看门狗定时器

相信大家都看过中国移动做的一个广告,从城市到山村,到青藏高原,在哪儿都有中国移动的网络,到哪儿都能打电话,由此可以联想到中国移动在全国有无数个信号基站,很多基站建设在环境比较恶劣的地方,我们来思考一个问题:假如,有一天某个基站出了问题不能正常工作了,毫无疑问,移动的工作人员会带各种检测设备去进行修理,如果是出现非硬件故障(如用户电话服务突然巨增,造成繁忙死机或电磁干扰造成 CPU 运行出错等),导致基站服务器出现异常死机,工作人员只需要进行一个操作,重启一下即可。如果该基站安装在青藏高原上,这样一次上去,成本是很大的。退一步讲,这种情况虽然成本很高,但是还是可以修复的,如果这种情况出现在月球探测器身上,问题就更严重了。因此针对这种特殊情况的设备,我们期待有一种设备能够在机器出现非正常情况下进行自动重启来恢复工作状态,使用看门狗定时器就可以完美解决这个问题了。

看门狗(WatchDog)的名字形象地描述了它的工作原理,看门狗每隔一段时间(比如:3 个小时)它就会饥饿,每次饥饿时都叫,如果不想让它叫,只要我们保证在 3 个小时内喂狗一次就行。因此我们要及时地对看门狗控制器执行喂狗操作。

看门狗定时器内部有一个递减计数器,当该计数器递减为 0 的时候,就会自动重启控制器(图 2-42),如果写有这样的程序,该程序在定时器计数器递减为 0 之前,将其递减计数器重新设置一下(喂狗),那么就不会产生重启操作。假如机器设备出现异常情况下如死机,CPU 执行出错,程序跑飞等情况,CPU 就会陷入非正常的执行流程,就不会去执行重置计数器的程序,当计数器递减为 0 时,会产生复位控制器信号,机器就会重新启动,恢复正常执行流程。这样的设计原理就解决了很多环境恶劣的情况下,对服务器进行重启的任务。上面的重置倒计数的操作通常称做喂狗。

2.4.1 看门狗定时器的用途

(1) 用于解决远程控制器在出现电磁干扰、噪声、系统错误等外界条件造成的系统死机等不正常运行的问题。

(2) 它不仅可以产生复位信号,还可以通过设置,产生定时的中断信号。

看门狗定时器从单片机时代就开始广泛使用,是嵌入式产品中的一大特色,很明显上述用途如果都用不到,就没有必要打开看门狗的功能,在 miniOS 操作系统里不使用它来定时产生

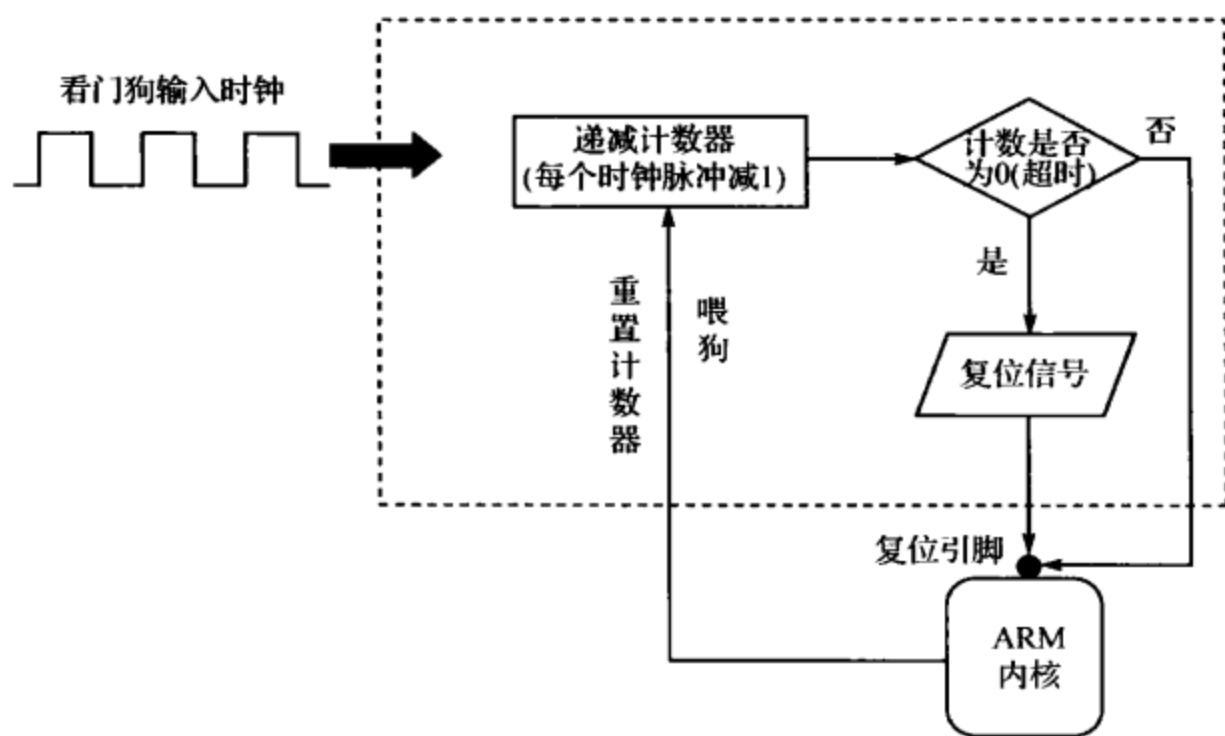


图 2-42 看门狗定时器

中断,也不用它来自动重启异常的开发板(因为出现异常的可能性几乎可以忽略),因为实现 miniOs 的第一步就是关闭看门狗,在开始裸板驱动之前,先来了解 S3C2440 看门狗定时器。

2.4.2 看门狗工作原理

S3C2440 看门狗定时器工作原理如图 2-43 所示。

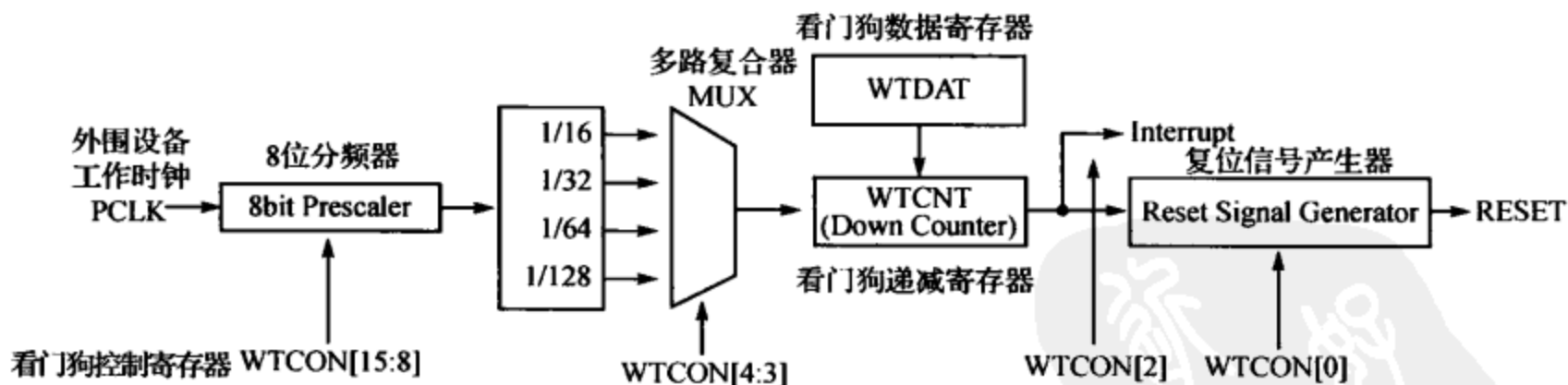


图 2-43 S3C2440 看门狗定时器工作原理

名词解释:

PCLK(Peripherals Clock): APB(Advanced Peripherals? Bus)高级外围设备总线上外设的工作时钟频率,它主要为开发板上的外围低速设备提供工作时钟,如串口、I2C 等。

8bit Prescaler: 8 位分频器,由程序来控制对 PCLK 进行分频,它由 WTCN 寄存器来控制设置。

MUX: 多路复合器,它支持多种时钟源,通过设置它来选择一种时钟频率,看门狗控制器

里会产生 4 种时钟频率,分别是 1/16、1/32、1/64、1/128 的已分频时钟,可以通过 WTCN 寄存器第 3、4 位进行选择设置。

WTCNT:看门狗递减寄存器,一旦看门狗使能,WTCNT 里的数据(由 WTDAT 寄存器在使能之前设置)就开始在输入时钟频率下递减。

Interrupt:中断信号,由 WTCN 寄存器的第二位控制是否产生中断信号。

Reset Signal Generator:Reset 信号产生器,默认情况下当递减寄存器里的倒计数为 0 时,通过 Reset 信号产生器产生 Reset 信号,让控制器重启,可以由 WTCN 寄存器的第 0 位进行设置。

当看门狗使能时,外围时钟 PCLK 为看门狗定时器的输入时钟,MINI2440 在没有开启时钟时,采用外部晶振(又称高频振荡器,它每秒钟产生固定频率时钟)提供的 12 MHz 输入时钟频率,PCLK 首先进入 8 位分频器 Prescaler,其对 PCLK 进行分频,可以理解为对 PCLK 的除法运算,这样 PCLK 的时钟频率就变成 $PCLK/Prescaler\ value$,其中分频器 Prescaler 的值可以在一定范围内自定义,然后经过分频的时钟经过除数因子选择器,这是在分频器的基础上再次对时钟信号进行降频,通过除数因子选择器,选择一个除数因子,然后经过 MUX 复合器,这时输入时钟变成 $PCLK/Prescaler\ value/除数因子$,得到想要的输入时钟,每个时钟周期都会将看门狗定时计数器 WTCNT 里的值减 1,当计数器 WTCNT 里的值变为 0 时开始执行超时操作,首先,判断看门狗控制寄存器里 bit2 WTCN[2]设置情况,如表 2-5 所列。如果为 1 则产生中断信号,引起系统中断,如果为 0 不做任何操作,进入复位信号产生器,如果 WTCN[0]位为 1,则产生控制器复位信号,否则不做任何操作。每次超时操作之后,看门狗 WTCN 会自动加载看门狗数据寄存器 WTDAT 里的用户设置值,继续执行递减操作,如表 2-6 所列。

表 2-5 看门狗定时器控制寄存器 (WTCN)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------------|------------|------|--|--------|
| WTCN | 0X53000000 | R/W | 看门狗定时器控制寄存器 | 0x8021 |
| WTCN | 位 | | 描 述 | 初始值 |
| Prescaler value | [15 : 8] | | 8 位分频器预设值,有效范围 $0 \sim 255(2^8 - 1)$ | 0x80 |
| Reserved | [7 : 6] | | 保留,正常情况下必须为 00 | 00 |
| Watchdog timer | [5] | | 看门狗定时器使能位 0 = 无效,1 = 有效 | 1 |
| Clock select | [4 : 3] | | 选择除数因子来决定输入时钟 00 = 16,01 = 32 10 = 64,11 = 128 | 00 |

续表 2-5

| WTCON | 位 | 描 述 | 初始值 |
|----------------------|-----|--|-----|
| Interrupt generation | [2] | 中断使能位 0 = 无效, 1 = 有效 | 0 |
| Reserved | [1] | 保留, 正常情况下该位必须为 0 | 0 |
| Reset enable/disable | [0] | 复位使能位 1: 看门狗定时器超时, 发出 CPU 复位信号 0: 看门狗定时器复位无效 | 1 |

表 2-6 看门狗定时器数据寄存器 (WTDAT)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|--------------------|------------|----------|-------------------|--------|
| WTDAT | 0x53000004 | R/W | 看门狗定时器数据寄存器 | 0x8000 |
| WTCNT | | 位 | 描 述 | 初始值 |
| Count Reload value | | [15 : 0] | 看该值在超时后,自动载入计数寄存器 | 0x8000 |

注: WTDAT 寄存器用于指定计数寄存器的初始值, 也就是它的超时时间, 系统上电之后硬件自动的将 0x8000 的初始值载入到 WTCNT 里, 在发生了第一次超时操作时, WTDAT 的值才会载入到 WTCNT 寄存器, 如表 2-7 所列。

表 2-7 看门狗定时器计数寄存器 (WTCNT)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-------------|------------|----------|--------------|--------|
| WTCNT | 0x53000008 | R/W | 看门狗定时器计数寄存器 | 0x8000 |
| WTCNT | | 位 | 描 述 | 初始值 |
| Count value | | [15 : 0] | 看门狗定时器的当前计数值 | 0x8000 |

通过对上面寄存器描述分析, 可以得出以下结论:

- 通过设置 WTCON[5]来设置看门狗定时器的使能。
- 在开启看门狗定时器后, 通过设置 WTCON[15 : 8]和[4 : 3]位来设置看门狗控制器的工作时钟频率, 具体看门狗的递减时间间隔可以通过下面的公式算出:

$$t_{\text{watchdog}} = 1 / (PCLK / (\text{Prescaler value} + 1) / \text{Division_factor})$$

式中, t_{watchdog} 为看门狗控制器递减时间间隔(单位秒);

PCLK 为 APB 总线工作时钟(单位 Hz);

Prescaler value 为 8 位分频器预设值;

Division_factor 为除数因子。

- 通过设置 WTCON[2]和 WTCON[0]位, 来使能产生中断和复位信号。
- 通过设置 WTDAT 来设置计数值。

2.4.3 看门狗实验

MINI2440 在没有初始化系统时钟时,整个开发板由一个 12 MHz 的外部晶振提供频率,PCLK 工作频率也是 12 MHz,WTCON[15:8]设置为 74,除数因子选择 16,通过上面公式可以计算出,看门狗控制器递减时间间隔 0.1 ms。将 WTCNT 里的值设置为 0x2710(十进制 10000),那么看门狗会每过一秒钟产生一次超时。

```
; 关闭看门狗实验
ldr r0, = 0x53000000          ; WTCON 寄存器地址加载到 r0
mov r1, #0                    ; r1 = 0
str r1, [r0]                  ; 将 r1 里的值存到 r0 里地址里
```

通过分析前面的寄存器的设置位可知,只要设置 WTCON[5]为 0 即可,上述代码里,直接将整个 WTCON 寄存器里的位设置为 0。

开启看门狗实验:

本实验源码存放在光盘目录\work\armarch\watchdog_enable 工程目录下。

```
WTCON      EQU      0x53000000    ; 看门狗控制寄存器
WTCNT      EQU      0x53000008    ; 看门狗计数寄存器
```

```
AREA  WATCHDOG_ENABLE, CODE, READONLY
```

```
ENTRY
```

```
; 设置看门狗控制寄存器
```

```
ldr r0, = WTCON          ; 加载 WTCON 寄存器地址
; 0x4a21 = [15:8] = 74, [5] = 1, [0] = 1
ldr r1, = 0x4a21         ; 将 0x4a21 保存到 r1 里
str r1, [r0]             ; 将 r1 里的值存入 r0 指向的地址
; 设置看门狗计数寄存器,该寄存器的值在上电后被加载,1 秒超时
ldr r2, = WTCNT          ; 加载 WTCNT 寄存器地址
ldr r3, = 0x2710         ; 将 0x2710 保存到 r1 里
str r3, [r2]             ; 将 r3 里的值存入 r2 指向的地址
```

```
IMPORT led_on            ; 引入 led_on 符号
bl led_on                ; 调用 led_on 代码
```

```
loop
```

```
b loop                  ; 死循环
```

```
END
```

程序开始定义两个常量地址,分别是 WTCON, WTCNT 的地址,ENRTY 后面是程序的

第2章 ARM 编程进阶

正文部分,首先通过 `ldr` 指令加载 `WTCNT` 的地址到 `r0` 里,将 `0x4a21` 这个数加载到 `r1` 里, `0x4a21` 是我们通过设置 `WTCNT[15:8]=74` `[7:6]=0`, `[5]=1`, `[4:1]=0`, `[0]=1` 得到的十六进制数,将其值存入到 `r0` 地址里,这样 `WTCNT` 里就是我们设置各位的值,然后同样道理再将 `0x2710` 存到 `WTCNT` 数据寄存器里,设置数据寄存器的值为 `0x2710`。

为了看到看门狗的重启效果,我们加入了一个小程序,用来点亮 LED 灯,将上述代码在 ADS 下编译完后,通过 H-JTAG 烧写到 NOR FLASH 里可以看到每过 1 s,开发板的 LED 灯就闪一下。

知识扩展

开启了看门狗之后,控制器会定时的复位,为了防止不停的复位,就要进行“喂狗”操作,喂狗操作相对比较简单,只要在 `WTCNT` 里的计数减为 0 之前,将其值重置一个非 0 的数值即可,看下面的函数: `feed_dog` (该代码仅供读者参考,光盘源码中没有给出具体例子)

; 喂狗程序

`feed_dog`

```
ldr r0, = WTCNT
ldr r1, = 0x2710
str r1, [r0]
mov pc, lr
```

喂狗程序对喂狗的时机必须要合适,否则在定时器还没来得及发生中断调用 `feed_dog` 函数之前, `watchdog` 已经超时了,也将引起系统复位重启,通常系统里会开启另外一个时钟来为整个系统服务,它会定时的“告知”系统,在看门狗定时器超时之前,自动地调用喂狗程序。

2.5 系统时钟

MINI2440 开发板在没有开启时钟前,整个开发板全靠一个 12 MHz 的晶振提供频率来运行,也就是说 CPU,内存, UART 等需要用到时钟频率的硬件都工作在 12 MHz 下,而 S3C2440A 可以正常工作在 400 MHz 下,两者速度相差可想而知,就好比牛车和动车。如果 CPU 工作在 12 MHz 频率下,开发板的使用效率非常低,所有依赖系统时钟工作的硬件,其工作效率也很低,比如,计算机的超频,超频就是让 CPU 工作在更高的频率下,让计算机运算速度更快,虽然频率是越高越好,但是由于硬件特性决定了任何一个设备都不可能无止境的超频,计算机超频时要考虑到 CPU 或主板发热过大,烧坏的危险,同样开发板的主板上的外设和 CPU 也有一个频率限度,ARM920T 内核的 S3C2440 的最高正常工作频率如下:

- FCLK: 400 MHz。
- HCLK: 100 MHz。
- PCLK: 50 MHz。

既然如此,那么怎样让 CPU 工作在 400 MHz,让牛车速度提高到动车的速度呢?

2.5.1 系统工作时钟频率

在对系统时钟进行提速之前,让我们先来了解下 S3C2440 上的工作时钟频率,FCLK,HCLK,PCLK,其中 FCLK 主要为 ARM920T 内核提供工作频率,如图 2-44 所示。

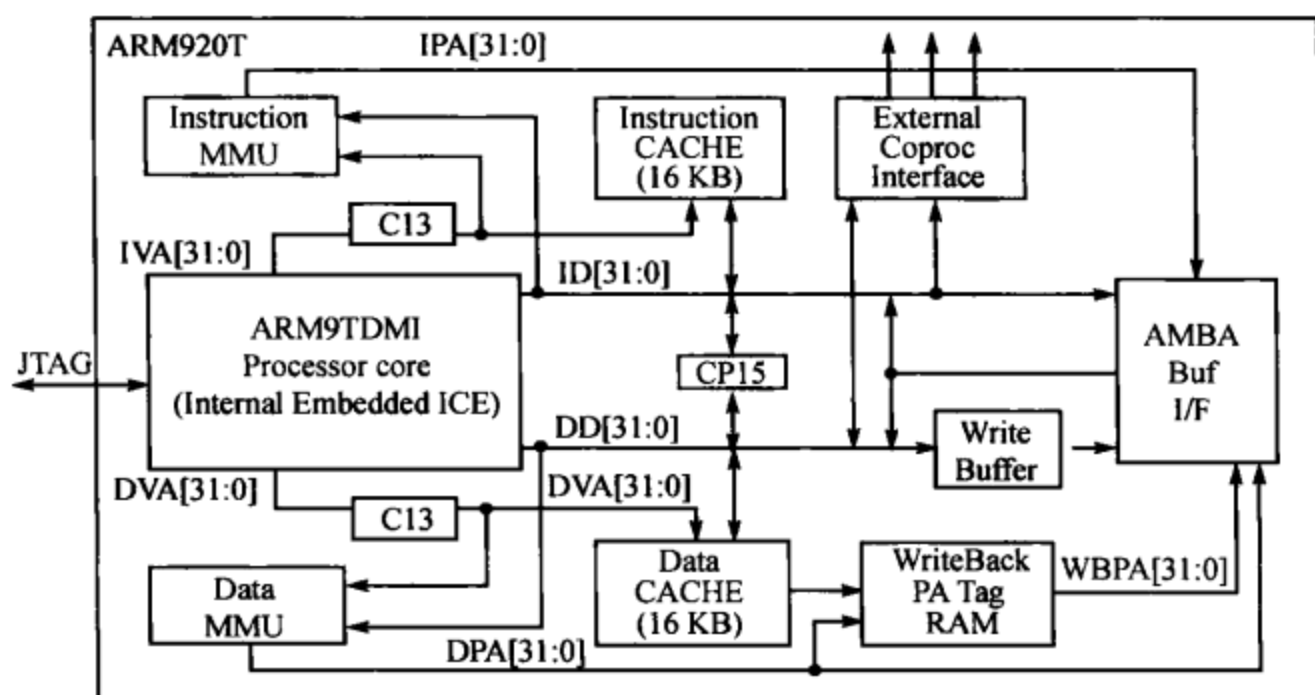


图 2-44 ARM920T 内核结构

HCLK 主要为 S3C2440 AHB 总线(Advanced High Performance Bus)上挂接的硬件提供工作频率,AHB 总线主要挂接有内存,NAND,LCD 控制器等硬件,如图 2-45 所示。

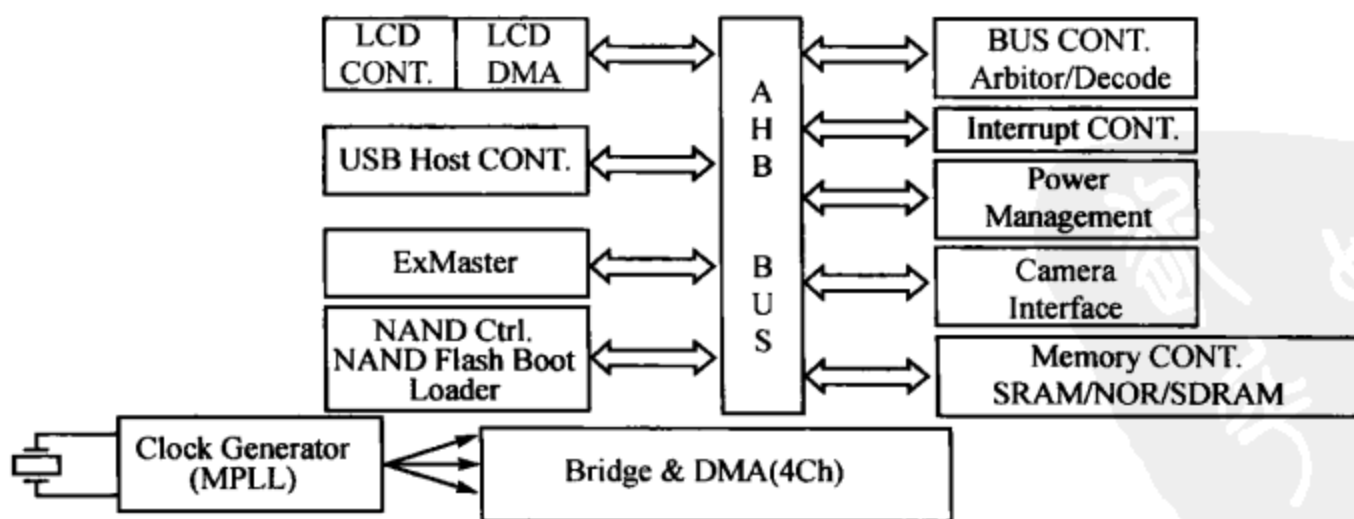


图 2-45 S3C2440 AHB 总线上挂接硬件

PCLK 主要为 APB 总线提供工作频率,如图 2-46 所示,APB 总线主要挂接 UART 串口,Watchdog 等硬件控制器。

也就是说,对于一些需要时钟工作的硬件,如果切断其时钟源,就不会再工作,从而达到降

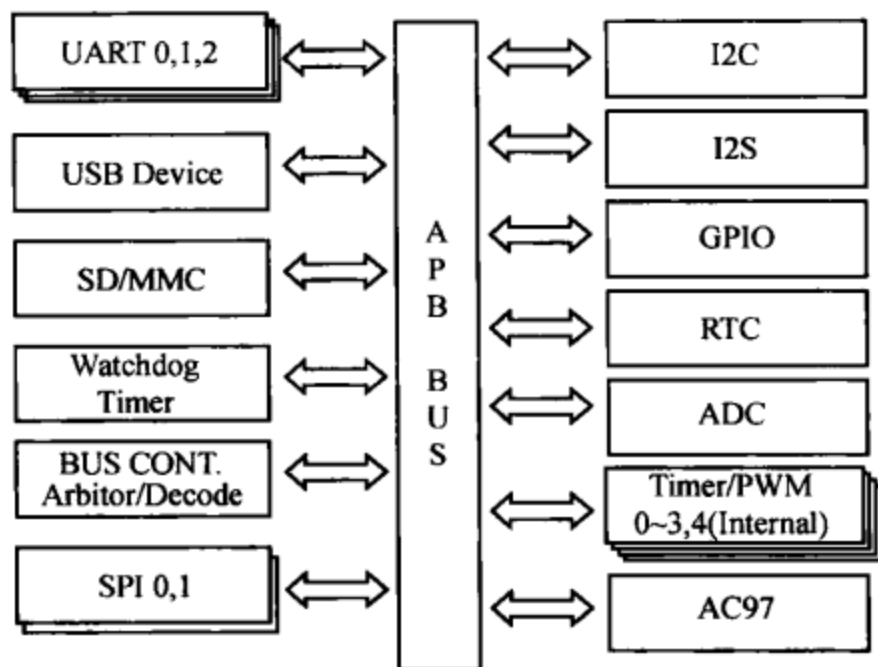


图 2-46 S3C2440 APB 总线挂接硬件

低功耗的目的,这也是便携嵌入式设备里的一个特点。

时钟源:为了减少外界环境对开发板电磁干扰,降低制作成本,通常开发板的外部晶振时钟频率都很低,MINI2440 开发板由 12MHz 的晶振来提供时钟源,要想让 CPU 运行在更高的频率就要通过时钟控制逻辑单元 PLL(锁相环)来提高主频。

S3C2440 里有两个 PLL:MPLL 和 UPLL。MPLL 用来产生 FCLK、HCLK、PCLK 的高频工作时钟;UPLL 用来为 USB 提供工作频率,如图 2-47 所示。

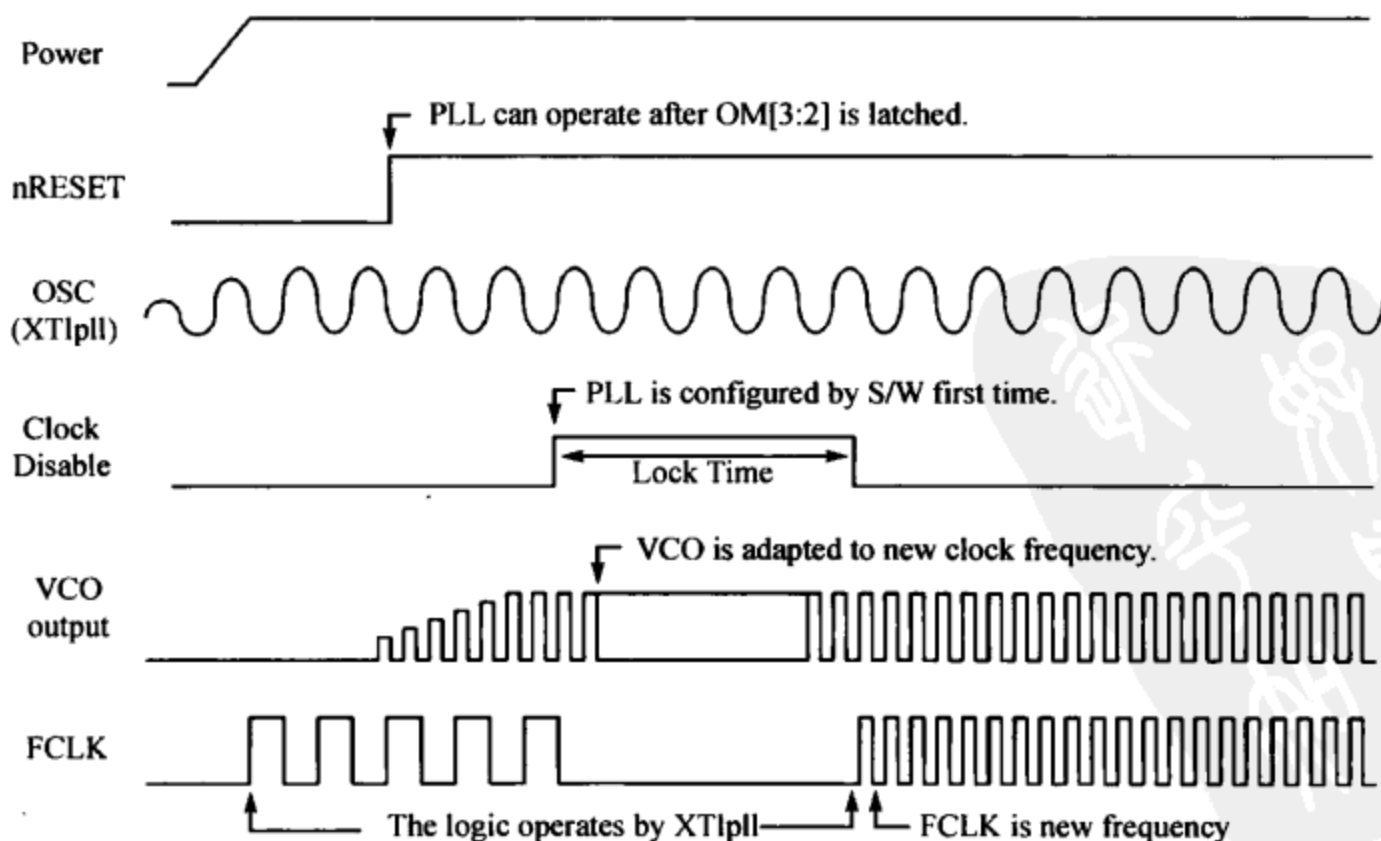


图 2-47 系统时钟初始化时序

开发板上电后,晶振 OSC 开始提供晶振时钟,由于系统刚刚上电,电压信号等都还不稳定,这时复位信号(nRESET)拉低,这时 MPLL 虽然默认启动,但是如果不向 MPLLCON 中写入值,那么外部晶振则直接作为系统时钟 FCLK,过几毫秒后,复位信号上拉,CPU 开始取指运行,这时可以通过代码设置启动 MPLL,MPLL 启动需要一定锁定时间(LockTime),这是因为 MPLL 输出频率还没有稳定,在这期间 FCLK 都停止输出,CPU 停止工作,过了 LockTime 后时钟稳定输出,CPU 工作在新设置的频率下,这时可以通过设置 FCLK,HCLK 和 PCLK 三者的频率比例来产生不同总线上需要的不同频率,下面详细介绍开启 MPLL 的过程:

- 设置 LockTime 变频锁定时间。
- 设置 FCLK 与晶振输入频率(Fin)的倍数。
- 设置 FCLK、HCLK、PCLK 三者之间的比例。

LockTime 变频锁定时间由 LOCKTIME 寄存器(见下表)来设置,如表 2-8 所列,由于变频后开发板所有依赖时钟工作的硬件都需要一小段调整时间,该时间计数通过设置 LOCKTIME 寄存器[31:16]来设置 UPLL(USB 时钟锁相环)调整时间,通过设置 LOCKTIME 寄存器[15:0]设置 MPLL 调整时间,这两个调整时间数值一般用其默认值即可。

表 2-8 变频锁定时间寄存器(LOCKTIME)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|----------|------------|------|--|------------|
| LOCKTIME | 0x4C000000 | R/W | 变频锁定时间寄存器 | 0xFFFFFFFF |
| LOCKTIME | 位 | | 描 述 | 初始值 |
| U_TIME | [31:16] | | UPLL 对 UCLK 的锁定时间值 (U_TIME: 300 μ s) | 0xFFFF |
| M_TIME | [15:0] | | MPLL 对于 FCLK,HCLK,PCLK 的锁定时间 值(M_TIME: 300 μ s) | 0xFFFF |

FCLK 与 Fin 的倍数通过 MPLLCON 寄存器设置,三者之间有以下关系:

$$\text{MPLL}(\text{FCLK}) = (2 \times m \times \text{Fin}) / (p \times 2^s)$$

其中: $m = \text{MDIV} + 8$, $p = \text{PDIV} + 2$, $s = \text{SDIV}$

当设置完 MPLL 之后,就会自动进入 LockTime 变频锁定期间,LockTime 之后,MPLL 输出稳定时钟频率,如表 2-9 所列。

表 2-9 MPLL 配置寄存器(MPLLCON)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|------|------------|------------|
| MPLLCON | 0x4C000004 | R/W | MPLL 配置寄存器 | 0x00096030 |

续表 2-9

| MPLLCON | 位 | 描述 | 初始值 |
|---------|---------|---------|------|
| MDIV | [19:12] | 主分频器控制位 | 0x96 |
| PDIV | [9:4] | 预分频器控制位 | 0x03 |
| SDIV | [1:0] | 后分频器控制位 | 0x0 |

通过上述算法比较难以找到合适的 PLL 值,表 2-10 给出了官方推荐的一些 MPLL 参考设置:

表 2-10 官方推荐 MPLL

| 输入频率/MHz | 输出频率/MHz | MDIV | PDIV | SDIV |
|----------|----------|-----------|------|------|
| 12.0000 | 48.00 | 56(0x38) | 2 | 2 |
| 12.0000 | 96.00 | 56(0x38) | 2 | 1 |
| 12.0000 | 271.50 | 173(0xad) | 2 | 2 |
| 12.0000 | 304.00 | 68(0x44) | 1 | 1 |
| 12.0000 | 405.00 | 127(0x7f) | 2 | 1 |
| 12.0000 | 532.00 | 125(0x7d) | 1 | 1 |
| 16.9344 | 47.98 | 60(0x3c) | 4 | 2 |
| 16.9344 | 95.96 | 60(0x3c) | 4 | 1 |
| 16.9344 | 266.72 | 118(0x76) | 2 | 2 |
| 16.9344 | 296.35 | 97(0x61) | 1 | 2 |
| 16.9344 | 399.65 | 110(0x6e) | 3 | 1 |
| 16.9344 | 530.61 | 110(0x56) | 1 | 1 |
| 16.9344 | 533.43 | 118(0x76) | 1 | 1 |

如表 2-11 所列,FCLK、HCLK、PCLK 三者之间的比例通过 CLKDIVN 寄存器进行设置(表 2-12),S3C2440 时钟设置时,还要额外设置 CAMDIVN 寄存器(表 2-13),如表 2-11 所列,HCLK4_HALF, HCLK3_HALF 分别与 CAMDIVN[9:8]对应,表中列出了各种时钟比例。

表 2-11 FCLK HCLK PCLK 设置比例

| HDIVN | PDIVN | HCLK3_HALF/HCLK4_HALF | FLCK | HCLK | PCLK | 分频比 |
|-------|-------|-----------------------|------|------|--------|-------|
| 0 | 0 | — | FLCK | FLCK | FLCK | 1:1:1 |
| 0 | 1 | --- | FLCK | FLCK | FLCK/2 | 1:1:2 |

续表 2-11

| HDIVN | PDIVN | HCLK3_HALF/HCLK4_HALF | FLCK | HCLK | PCLK | 分频比 |
|-------|-------|-----------------------|------|--------|---------|--------|
| 1 | 0 | — | FLCK | FLCK/2 | FLCK/2 | 1:2:2 |
| 1 | 1 | — | FLCK | FLCK/2 | FLCK/4 | 1:2:4 |
| 3 | 0 | 0/0 | FLCK | FLCK/3 | FLCK/3 | 1:3:3 |
| 3 | 1 | 0/0 | FLCK | FLCK/3 | FLCK/6 | 1:3:6 |
| 3 | 0 | 1/0 | FLCK | FLCK/6 | FLCK/6 | 1:6:6 |
| 3 | 1 | 1/0 | FLCK | FLCK/6 | FLCK/12 | 1:6:12 |
| 2 | 0 | 0/0 | FLCK | FLCK/4 | FLCK/4 | 1:4:4 |
| 2 | 1 | 0/0 | FLCK | FLCK/4 | FLCK/8 | 1:4:8 |
| 2 | 0 | 0/1 | FLCK | FLCK/8 | FLCK/8 | 1:8:8 |
| 2 | 1 | 0/1 | FLCK | FLCK/8 | FLCK/16 | 1:8:16 |

如果 HDIV 设置为非 0, CPU 的总线模式要进行改变, 默认情况下 FCLK = HCLK, CPU 工作在 fast bus mode 快速总线模式下, HDIV 设置为非 0 后, FCLK 与 HCLK 不再相等, 要将 CPU 改为 asynchronous bus mod 异步总线模式, 可以通过下面的嵌入汇编代码实现。

```
__asm{
    mrc p15, 0, r1, c1, c0, 0    /* 读取 CP15 C1 寄存器 */
    orr r1, r1, #0xc0000000     /* 设置 CPU 总线模式 */
    mcr p15, 0, r1, c1, c0, 0    /* 写回 CP15 C1 寄存器 */
}
```

关于 mrc 与 mcr 指令, 请查看 MMU 与内存保护的实现章节。

表 2-12 时钟分频器控制寄存器 (CLKDIVN)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|----------|------------|------|--|------------|
| CLKDIVN | 0x4C000014 | R/W | 时钟分频器控制寄存器 | 0x00000000 |
| CLKDIVN | | 位 | 描 述 | 初始值 |
| DIV_UPLL | | [3] | UCLK 选择寄存器 (UCLK 必须对 USB 提供 48MHz) 0: UCLK = UPLL clock 1: UCLK = UPLL clock/2 | 0 |

| CLKDIVN | 位 | 描 述 | 初始值 |
|---------|-------|--|-----|
| HDIVN | [2:1] | 00: HCLK = FCLK/1 01: HCLK = FCLK/2 10: HCLK = FCLK/4, 当 CAMIVN[9]=0 HCLK = FCLK/8, 当 CAMIVN[9]=1 11: HCLK = FCLK/3, 当 CAMIVN[8]=0 HCLK = FCLK/6, 当 CAMIVN[8]=1 | 0 |
| PDIVN | [0] | 0: PCLK 是和 HCLK/1 相同时钟 1: PCLK 是和 HCLK/2 相同时钟 | 0 |

表 2-13 摄像头时钟分频控制寄存器 (CAMDIVN)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|------------|------------|------|---|------------|
| CAMDIVN | 0x4C000018 | R/W | 摄像头时钟分频控制寄存器 | 0x00000000 |
| CAMDIVN | 位 | | 描 述 | 初始值 |
| ... | ... | | ... | ... |
| HCLK4_HALF | [9] | | HDIVN 分频因子选择位(当 CLKIVN[2:1] 位为 10b 时有效) 0: HCLK=FCLK/4 1: HCLK=FCLK/8 | 0 |
| HCLK3_HALF | [8] | | HDIVN 分频因子选择位(当 CLKIVN[2:1] 位为 11b 时有效) 0: HCLK=FCLK/3 1: HCLK=FCLK/6 | 0 |
| ... | ... | | ... | ... |

2.5.2 时钟驱动实验

系统时钟驱动可以分别用 ARM 汇编和 C 语言两个版本实现。

ARM 汇编版本:

; 以下为时钟相关寄存器地址

| | | |
|----------|-----|------------|
| LOCKTIME | EQU | 0x4c000000 |
| MPLLCON | EQU | 0x4c000004 |
| CLKDIVN | EQU | 0x4c000014 |
| CAMDIVN | EQU | 0x4c000018 |

```

clock_init                                ; 时钟初始化代码
    ; 设置变频锁定时间
    ldr r0, = LOCKTIME
    ldr r1, = 0x00ffffff
    str r1, [r0]

    ; 设置分频比 FCLK:HCLK:PCLK = 1:4:8
    ; 由于 CAMDIVN[9]位初始值为 0,寄存器 CAMDIVN 未使用,这儿不用再设置其值
    ldr r0, = CLKDIVN
    mov r1, # 0x05
    str r1, [r0]

    ; 修改 CPU 总线模式
    mrc    p15, 0, r1, c1, c0, 0
    orr     r1, r1, # 0xc0000000
    mcr     p15, 0, r1, c1, c0, 0

    ldr r0, = MPLLCON
    ldr r1, = 0x5c011                      ; MPLL = 400MHz
    str r1, [r0]
    mov pc, lr                            ; 函数调用返回

```

该汇编代码入口处先设置了变频锁定时间为 0x00ffffff,然后设置 FCLK:HCLK:PCLK 的分频比,由于系统时钟已经改变,需要修改 CPU 总线模式,最后设置系统时钟工作频率。

C 语言版本:

```

/* 通过 MPLL 计算公式可以算出:MDIV = 92,PDIV = 1,SDIV = 0 时,MPLL = 400MHz
#define MPLL_400MHz ((92 << 12)|(1 << 4)|(1 << 0))
void clock_init(void){
    /* 设置变频锁定时间 */
    LOCKTIME = 0x00ffffff;
    /* 设置分频比 FCLK:HCLK:PCLK = 1:4:8,CAMDIVN 初始值为 0,不用再对其设置 */
    CLKDIVN   = 0x05;
    /* 修改 CPU 总线模式 */
    __asm{
        mrc    p15, 0, r1, c1, c0, 0
        orr     r1, r1, # 0xc0000000
        mcr     p15, 0, r1, c1, c0, 0
    }
    MPLLCON = MPLL_400MHz;
}

```

}

C 语言版本与汇编版本一样,只是由于修改 CPU 总线模式时要使用 mrc 指令,因此只能使用 C 语言嵌入汇编方式来实现。

系统时钟驱动实验如下:

本实验源码存放在光盘目录\work\armarch\system_clock 工程目录下。

;

; 系统时钟初始化实验

;

| | | | |
|----------|-----|------------|-------------|
| WTCON | EQU | 0x53000000 | ; 看门狗控制寄存器 |
| WTDAT | EQU | 0x53000004 | ; 看门狗数据寄存器 |
| LOCKTIME | EQU | 0x4c000000 | ; 变频锁定时间寄存器 |
| MPLLCON | EQU | 0x4c000004 | ; MPLL 寄存器 |
| CLKDIVN | EQU | 0x4c000014 | ; 分频比寄存器 |
| GPBCON | EQU | 0x56000010 | ; LED 控制寄存器 |
| GPBDAT | EQU | 0x56000014 | ; LED 数据寄存器 |
| GPBUP | EQU | 0x56000018 | ; 上拉电阻设置寄存器 |
| DELAYVAL | EQU | 0x8fff | ; 延时数值 |

```
AREA    CLOCK, CODE, READONLY
```

```
ENTRY
```

```
start
```

```
    ldr r0, = 0x53000000          ; 看门狗关闭代码
```

```
    mov r1, #0
```

```
    str r1, [r0]
```

```
    bl clock_init                ; 调用时钟初始化函数
```

```
    bl led_on                    ; 调用点亮 LED 函数
```

```
clock_init                        ; 时钟初始化代码
```

```
    ; 设置锁频时间
```

```
    ldr r0, = LOCKTIME           ; 取得 LOCKTIME 寄存器地址
```

```
    ldr r1, = 0x00ffffff         ; LOCKTIME 寄存器设置数据
```

```
    str r1, [r0]                 ; 将 LOCKTIME 设置数据写入 LOCKTIME 寄存器
```

```
    ; 设置分频数
```

```
    ldr r0, = CLKDIVN            ; 取得 CLKDIVN 寄存器地址
```

```
    mov r1, #0x05                ; CLKDIVN 寄存器设置数据
```

```
    str r1, [r0]                 ; 将 CLKDIVN 设置数据写入 CLKDIVN 寄存器
```

; 修改 CPU 总线模式

```
mrc    p15, 0, r1, c1, c0, 0
orr     r1, r1, #0xc0000000
mcr     p15, 0, r1, c1, c0, 0
```

```
ldr r0, =MPLLCON
```

```
ldr r1, =0x5c011 ; MPLL is 400MHz
```

```
str r1, [r0]
```

```
mov pc, lr
```

led_on ; 亮点 LED 函数

; LED 初始化开始

```
ldr r0, =GPBCON
```

; 将 LED 控制寄存器地址放入 r0

```
ldr r1, [r0]
```

; 将控制寄存器里的值读出放入 r1

```
bic r1, r1, #0x3fc00
```

; 将 r1 里的值(控制寄存器里的值)

; bit[10]~bit[17]清除, 其他位不变

```
orr r1, r1, #0x15400
```

; 设置控制寄存器

```
str r1, [r0]
```

; 将 r1 里的值写入控制寄存器

; 禁止 GPF4 - GPF7 端口的上拉电阻

```
ldr r0, =GPBUP
```

```
ldr r1, [r0]
```

```
orr r1, r1, #0x1e0
```

```
str r1, [r0]
```

; LED 初始化结束

led_loop

; 循环点亮 LED

```
ldr r2, =GPBDAT
```

; 将 LED 数据寄存器的地址放入 r2

```
ldr r3, [r2]
```

; 将数据寄存器(r2)里的值放入 r3

```
bic r3, r3, #0x1e0
```

; 清除 bit[5]~bit[8], bit[n]代表 LED1~LED4

```
orr r3, r3, #0x1c0
```

; 清对应 LED 位 - 亮灯, 设置相应位 - 灭灯(点亮 LED1)

```
str r3, [r2]
```

; 将控制亮灯数据写入数据寄存器 r2

```
ldr r0, =DELAYVAL
```

; 设置延迟数

```
bl delay
```

; 调用延迟子程序

```
ldr r3, [r2]
```

; 将数据寄存器(r2)里的值放入 r3

```
bic r3, r3, #0x1e0
```

; 清除 bit[5]~bit[8], bit[n]代表 LED1~LED4

```
orr r3, r3, #0x1a0
```

; 清对应 LED 位 - 亮灯, 设置相应位 - 灭灯(点亮 LED2)

第2章 ARM 编程进阶

```

str r3,[r2]           ; 将控制亮灯数据写入数据寄存器 r2
ldr r0, = DELAYVAL    ; 设置延迟数
bl delay              ; 调用延迟子程序

ldr r3,[r2]           ; 将数据寄存器(r2)里的值放入 r3
bic r3,r3,#0x1e0       ; 清除 bit[5]~bit[8],bit[n]代表 LED1~LED4
orr r3,r3,#0x160       ; 清除对应 LED 位 - 亮灯,设置相应位 - 灭灯(点亮 LED3)
str r3,[r2]           ; 将控制亮灯数据写入数据寄存器 r2
ldr r0, = DELAYVAL    ; 设置延迟数
bl delay              ; 调用延迟子程序

ldr r3,[r2]           ; 将数据寄存器(r2)里的值放入 r3
bic r3,r3,#0x1e0       ; 清除 bit[5]~bit[8],bit[n]代表 LED1~LED4
orr r3,r3,#0xe0        ; 清除对应 LED 位 - 亮灯,设置相应位 - 灭灯(点亮 LED4)
str r3,[r2]           ; 将控制亮灯数据写入数据寄存器 r2
ldr r0, = DELAYVAL    ; 设置延迟数
bl delay              ; 调用延迟子程序

b led_loop

delay
    sub r0,r0,#1        ; r0 = r0 - 1
    cmp r0,#0x0         ; 将 r0 的值与 0 相比较
    bne delay           ; 比较的结果不为 0,继续调用 delay
    mov pc,lr           ; 返回
END                    ; 程序结束符

```

该实验首先关闭了看门狗定时器,然后修改系统时钟,将默认系统工作频率 12 MHz 提高到 400 MHz,由于 CPU 工作在较高频率下,其执行速度明显比未启动系统时钟时高得多,可以通过注释掉系统时钟初始化代码跳转指令 `bl clock_init`,对比 LED 的跑马灯效果可以证明。

2.6 SDRAM 内存

SDRAM(Synchronous Dynamic Random Access Memory,同步动态随机存储器)也就是通常所说的内存。内存的工作原理、控制时序、及相关控制器的配置方法一直是嵌入式系统学习、开发过程中的一个难点。我们从其硬件的角度来分析其原理,然后再引出 SDRAM 的驱动编写过程。

内存是代码的执行空间,以 PC 为例,程序是以文件的形式保存在硬盘里面的,程序在运

行之前先由操作系统装载入内存中,由于内存是 RAM(随机访问存储器),可以通过地址去定位一个字节的数, CPU 在执行程序时将 PC 的值设置为程序在内存中的开始地址, CPU 会依次地从内存里取址、译码、执行。在内存没有被初始化之前,内存好比是未建好的房子,是不能读取和存储数据的,因此要想让程序代码运行在内存里,必须进行内存的初始化。

通用存储设备:

在介绍内存工作原理之前有必要了解下存储设备的存储方式:ROM, RAM。

(1) ROM(Read-Only Memory):只读存储器是一种只能读出事先所存数据的固态半导体存储器。其特性是一旦储存资料就无法再将之改变或删除。通常用在不需经常变更资料的电子或电脑系统中,资料并且不会因为电源关闭而消失。如 PC 里面的 BIOS。

(2) RAM(Random Access Memory):随机访问存储器,存储单元的内容可按需随意取出或存入,且存取的速度与存储单元的位置无关的存储器。可以理解为,当给定一个随机有效的访问地址, RAM 会返回其存储内容(随机寻址),它访问速度与地址的无关。这种存储器在断电时将丢失其存储内容,故主要用于存储短时间内随机访问使用的程序。计算机系统里内存地址是一个 4 字节对齐的地址(32 位机), CPU 的取指、执行、存储都是通过地址进行的。

RAM 按照硬件设计的不同,随机存储器又分为 DRAM(Dynamic RAM)动态随机存储器和 SRAM(Static RAM)静态随机存储器。

① DRAM:它的基本原件是小电容,电容可以在两个极板上短时间内保留电荷,可以通过两极之间有无电压差代表计算机里的 0 和 1,由于电容的物理特性,要定期地为其充电,否则数据会丢失。对电容的充电过程称做刷新,但是其制作工艺较简单,体积小,便于集成化,经常作为计算机里内存制作原件。比如 PC 的内存, SDRAM, DDR, DDR2, DDR3 等,缺点:由于要定期刷新存储介质,存取速度较慢。

② SRAM:它是一种具有静止存取功能的内存,不需要刷新电路即能保存它内部存储的数据。因此其存取速度快,但是体积较大,功耗大,成本高,常用做存储容量不高,但存取速度快的场合,比如 CPU 的 L1 cache, L2 cache(一级,二级缓存),寄存器。

为了满足开发的需要 MINI2440 在出厂时搭载了 3 种存储介质:

(1) NOR FLASH(2MB):ROM 存储器,通常用来保存 BootLoader,引导系统启动。

(2) NAND FLASH(256MB,型号不一样, NAND FLASH 大小不一样):保存操作系统映像文件和文件系统。

(3) SDRAM(64MB):内存,执行程序。

另外, NOR FLASH 和 NAND FLASH 的特点如下:

(1) NOR FLASH:它的特点是支持 XIP 芯片内执行(eXecute In Place),这样应用程序可以直接在 Flash 闪存内运行,不必再把代码读到系统 RAM 中,也就是说可以随机寻址。NOR FLASH 的成本较高。

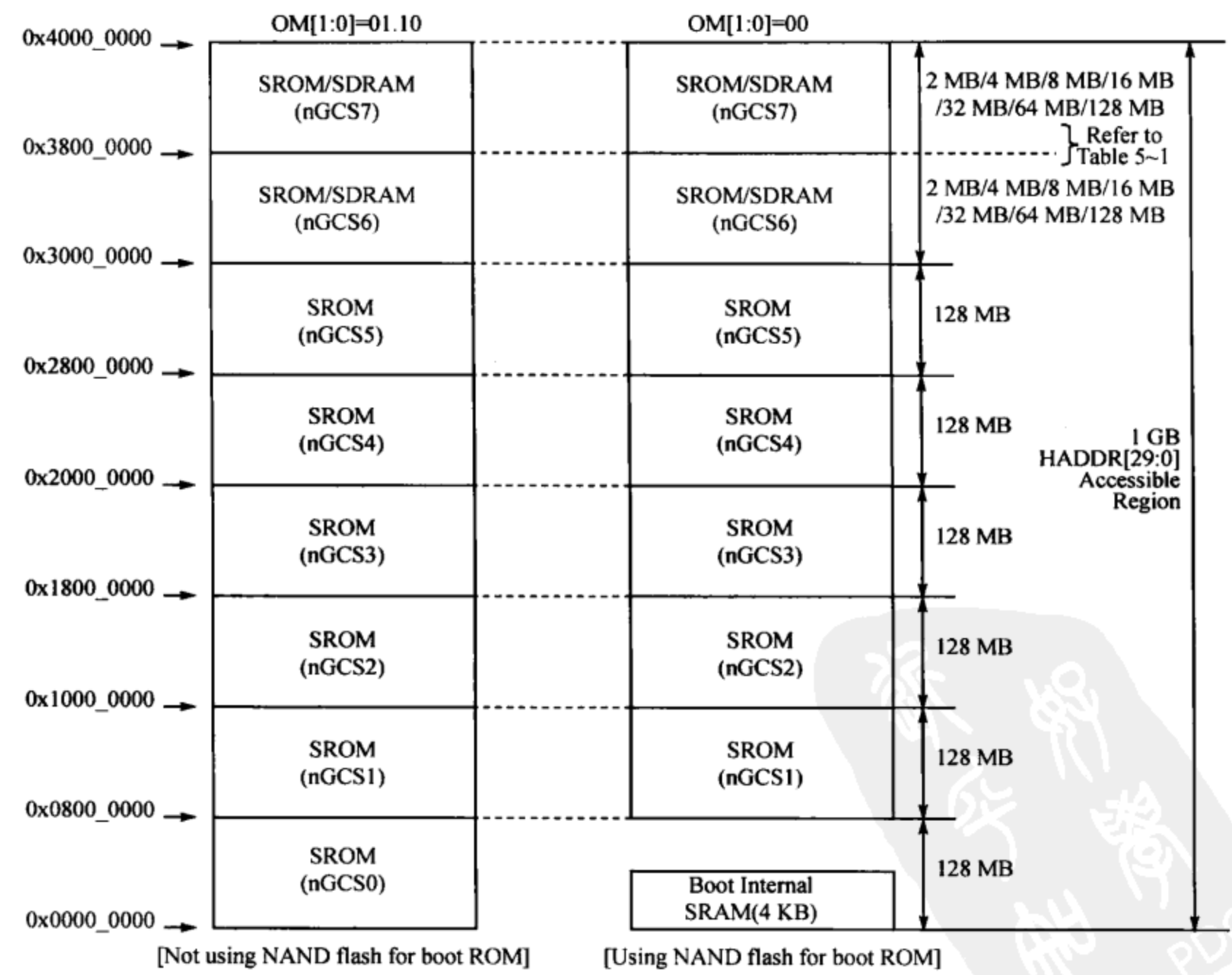
(2) NAND FLASH:它能提供极高的单元密度,可以达到高存储密度,并且写入和擦除

第2章 ARM 编程进阶

的速度也很快。其成本较低,不支持 XIP。可做嵌入式里的数据存储介质,如手机存储卡、SD 卡等。

2.6.1 S3C2440 存储器地址段(Bank)

S3C2440 对外引出了 27 根地址线 ADDR0~ADDR26,它最多能够寻址 128 MB,而 S3C2440 的寻址空间可以达到 1 GB,这是由于 S3C2440 将 1 GB 的地址空间分成了 8 个 BANKS(Bank0~Bank7),其中每一个 BANK 对应一根片选信号线 nGCS0~nGCS7,当访问 BANKx 的时候,nGCSx 引脚电平拉低,用来选中外接设备,S3C2440 通过 8 根选信号线和 27 根地址线,就可以访问 1 GB,如图 2-48 所示。



Note: SROM means ROM or SRAM type memory

图 2-48 S3C2440 存储器 Bank

如图 2-48 所示,左侧图对应不使用 NAND FLASH 启动时(通过跳线设置),存储器

Bank 分布图,通常在这种启动方式里选择 NOR FLASH 启动,将 NOR FLASH 焊接在 Bank0,系统上电后,CPU 从 Bank0 的开始地址 0x00000000 开始取指运行。

图右侧是选择从 NAND FLASH 引导启动(通过跳线设置),系统上电后,CPU 会自动将 NAND FLASH 里前 4KB 的数据复制到 S3C2440 内部一个 4KB 大小 SRAM 类型存储器里(称做 Stepping stone),然后从 Stepping stone 取指启动。

其中 Bank0~Bank5 可以焊接 ROM 或 SRAM 类型存储器,Bank6~Bank7 可以焊接 ROM,SRAM,SDRAM 类型存储器,也就是说,S3C2440 的 SDRAM 内存应该焊接在 Bank6~Bank7 上,最大支持内存 256MB,Bank0~Bank5 通常焊接一些用于引导系统启动小容量 ROM,具体焊接什么样存储器,多大容量,根据每个开发板生产商不同而不同,比如 MINI2440 开发板将 2MB 的 NOR FLASH 焊接在了 Bank0 上,用于存放系统引导程序 Bootloader,将两片 32MB,16Bit 位宽 SDRAM 内存焊接在 Bank6 和 Bank7 上,并联形成 64MB,32 位内存。

由于 S3C2440 是 32 位芯片,理论上讲可以达到 4GB 的寻址范围,除去上述 8 个 Bank 用于连接外围设备,还有一部分的地址空间是用于设备特殊功能寄存器,其余地址没有被使用,如表 2-14 所列。

表 2-14 S3C2440 设备寄存器地址空间

| 外接设备 | 起始地址 | 结束地址 |
|----------------|------------|------------|
| 存储控制器 | 0x48000000 | 0x48000030 |
| USB Host 控制器 | 0x49000000 | 0x49000058 |
| 中断控制器 | 0x4A000000 | 0x4A00001C |
| DMA | 0x4B000000 | 0x4B0000E0 |
| 时钟和电源管理 | 0x4C000000 | 0x4C000014 |
| LCD 控制器 | 0x4D000000 | 0x4D000060 |
| NAND FLASH 控制器 | 0x4E000000 | 0x4E000014 |
| 摄像头接口 | 0x4F000000 | 0x4F0000A0 |
| UART | 0x50000000 | 0x50008028 |
| 脉宽调制计时器 | 0x51000000 | 0x51000040 |
| USB 设备 | 0x52000140 | 0x5200026F |
| WATCHDOG 计时器 | 0x53000000 | 0x53000008 |
| IIC 控制器 | 0x54000000 | 0x5400000C |
| IIS 控制器 | 0x55000000 | 0x55000012 |
| I/O 端口 | 0x56000000 | 0x560000B0 |
| 实时时钟 RTC | 0x57000040 | 0x5700008B |
| A/D 转换器 | 0x58000000 | 0x58000010 |

续表 2-14

| 外接设备 | 起始地址 | 结束地址 |
|-------------|------------|------------|
| SPI | 0x59000000 | 0x59000034 |
| SD 接口 | 0x5A000000 | 0x5A000040 |
| AC97 音频编码接口 | 0x5B000000 | 0x5B00001C |

2.6.2 SDRAM 内存工作原理

SDRAM 的内部是一个存储阵列。阵列就如同表格一样,将数据“填”进去。在数据读写时和表格的检索原理一样,先指定一个行(Row),再指定一个列(Column),我们就可以准确地找到所需要的单元格,这就是内存芯片寻址的基本原理,如图 2-49 所示。



图 2-49 内存行、列地址寻址示意图

这个单元格(存储阵列)就称为逻辑 Bank(Logical Bank, L-Bank)。由于技术、成本等原因,不可能只做一个全容量的 L-Bank,而且最重要的是,由于 SDRAM 的工作原理限制,单一的 L-Bank 将会造成非常严重的寻址冲突,大幅降低内存效率。所以人们在 SDRAM 内部分割成多个 L-Bank,目前基本都是 4 个(这也是 SDRAM 规范中的最高 L-Bank 数量),由此可见,在进行寻址时就要先确定是哪个 L-Bank,然后在这个选定的 L-Bank 中选择相应的行与列进行寻址。因此对内存的访问,一次只能是一个 L-Bank 工作。如图 2-50 所示。

当对内存进行操作时,先要确定操作 L-Bank,因此要对 L-Bank 进行选择。在内存芯片的外部引脚上多出了两个引脚 BA0, BA1,用来片选 4 个 L-Bank。如前所述,32 位的地

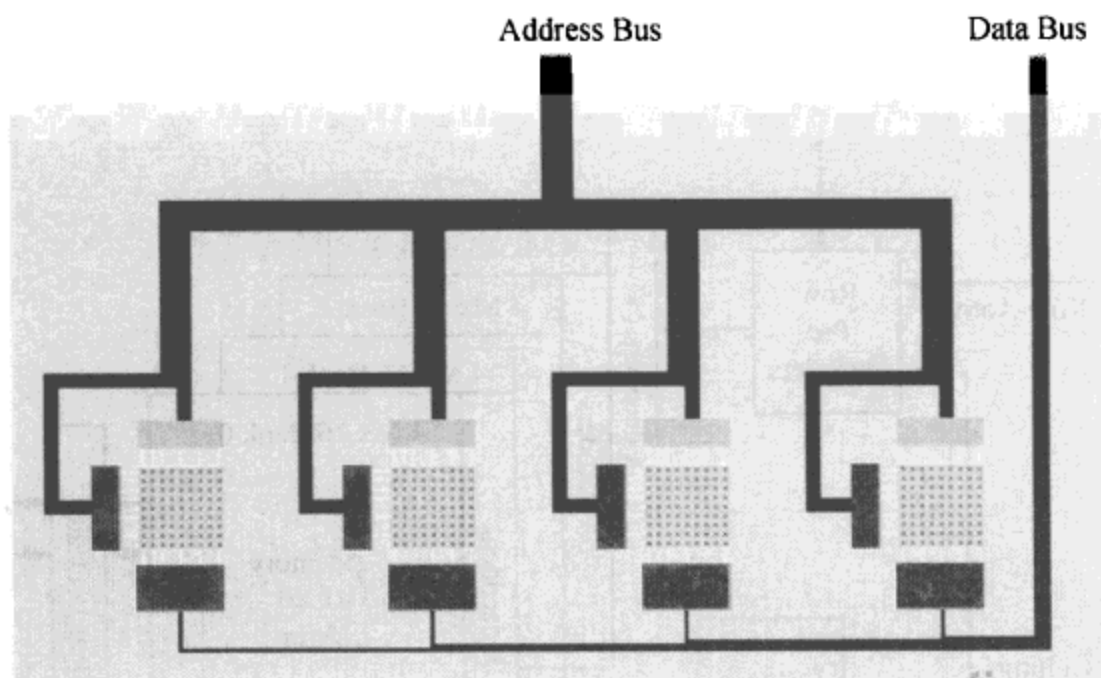


图 2-50 内存存储单元

址长度由于其存储结构特点,分成了行地址和列地址。通过图 2-51 所示的内存结构图可知,内存外接引脚地址线只有 13 根地址线 A0~A12,它最多只能寻址 8MB 内存空间,到底使用什么机制来实现对 64MB 内存空间进行寻址的呢? SDRAM 的行地址线和列地址线是分时复用的,即地址要分两次送出,先送出行地址(nSRAS 行有效操作),再送出列地址(nSCAS 列有效操作)。这样,可以大幅度减少地址线的数目,提高器件的性能和制作工艺复杂度。但寻址过程也会因此而变得复杂。实际上,现在的 SDRAM 一般都以 L-Bank 为基本寻址对象的。由 L-Bank 地址线 BAn 控制 L-Bank 间的选择,行地址线和列地址线贯穿连接所有的 L-Bank,每个 L-Bank 的数据的宽度和整个存储器的宽度相同,这样,可以加快数据的存储速度。同时,BAn 还可以使未被选中的 L-Bank 工作于低功耗的模式下,从而降低器件的功耗。

开发板内存控制器引脚接线(以 MINI2440 开发板为例):

(1)确定 BA0、BA1 的接线(表 2-15)。

表 2-15 BA0、BA1 接线

| Bank Size | Bus Width | Base Component | Memory Configuration | Bank Address |
|-----------|-----------|----------------|----------------------|--------------|
| 64MB | x32 | 128MB | (4M x 8 x 4B) x4 | A[25 : 24] |
| | x16 | 256MB | (8M x 8 x 4B) x2 | |
| | x32 | | (4M x 16 x 4B) x2 | |
| | x8 | 512MB | (16M x 8 x 4B) x1 | |

Bank Size: 外接内存容量大小(HY57561620 是 4Mbit * 16bit * 4Bank * 2Chips/8 = 64MB)

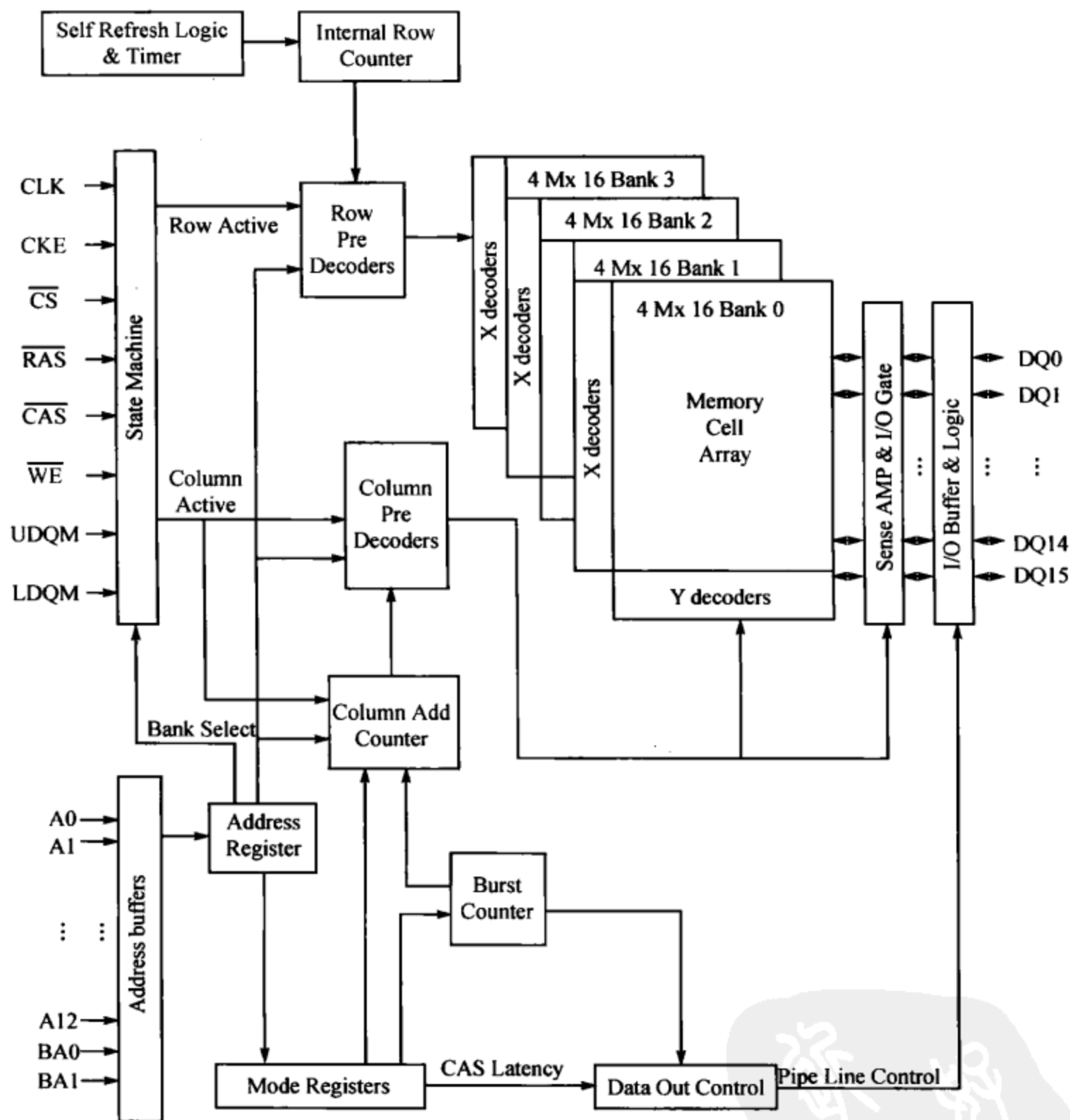


图 2-51 HY57561620 内部结构图

Bus Width: 总线宽度(两片 16 位 HY57561620, 并联成 32 位)

Base Component: 单个芯片容量(bit)(256MB)

Memory Configuration: 内存配置 ((4M * 16 * 4banks) * 2Chips)

由硬件手册 Bank Address 引脚连接配置表可知, 使用 A[25:24] 两根地址线作为 Bank 片选信号, 正好两根接线可以片选每个存储单元的 4 个 BANKS。

(2) 确定其他接线。SDRAM 内存是焊接在 BANK6~BANK7 上的, 其焊接引脚, 如图 2-52

所示。

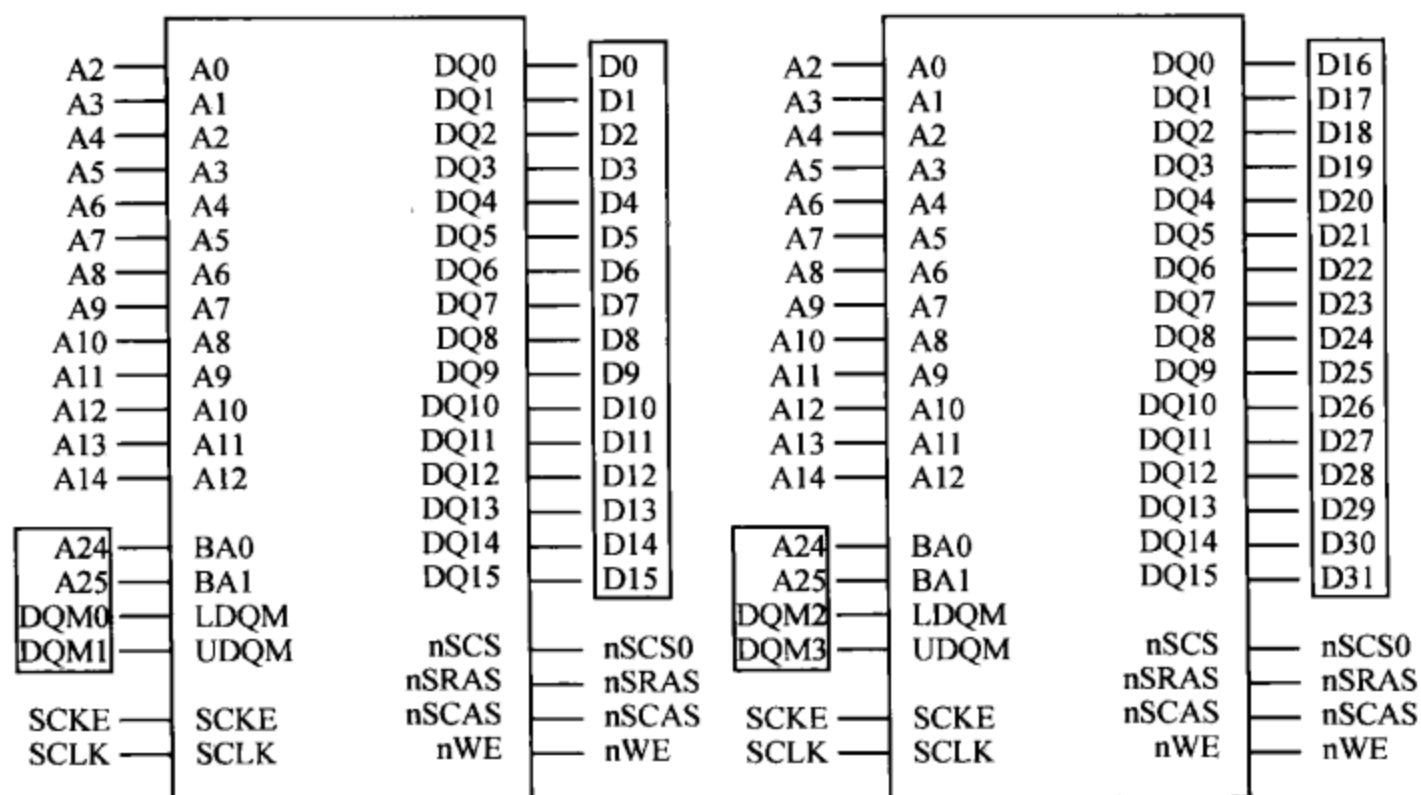


图 2-52 S3C2440 16 位宽内存芯片

图 2-52 是 S3C2440 提供的两片 16 位芯片并联连接示意图, A_n 是 CPU 地址总线, 其中 $A_2 \sim A_{14}$ 为内存芯片寻址总线, 之所以地址寻址总线从 A_2 开始是因为内存地址都是按字节对齐的, A_{24}, A_{25} 为 L-Bank 片选信号, D_n 为 CPU 数据总线, 其他为对应控制信号线, 如表 2-16 所列。

表 2-16 内存芯片各引脚说明

| 外接引脚名 | 内接引脚名 | 全 称 | 描 述 |
|--------------------|---------------------|------------------------|---------------------------|
| $A_2 \sim A_{14}$ | $A_0 \sim A_{12}$ | Address | 地址线 |
| $D_0 \sim D_{31}$ | $DQ_0 \sim DQ_{31}$ | Data Input/Output | 数据线 |
| A_{24}, A_{25} | BA_0, BA_1 | Bank Address | L-Bank 片选信号 |
| $DQM_0 \sim DQM_3$ | $LDQM, UDQM$ | Data Input/Output Mask | 高, 低字节数据掩码信号 |
| SCKE | SCKE | Clock Enable | 输入时钟有效信号 |
| SCLK | SCLK | Clock | 输入时钟 |
| nSCS0 | nSCS | General Chip Select | 片选信号(它与 nGCS6 是同一引脚的两个功能) |
| nSRAS | nSRAS | Row Address Strobe | 行地址选通信号 |
| nSCAS | nSCAS | Column Address Strobe | 列地址选通信号 |
| new | new | Write Enable | 写入有效信号 |

通过 S3C2440 16 位宽内存芯片接线图可以看出,两片内存芯片只有两个地方不一样,LDQM,UDQM 和数据总线 DQn 接线方式不一样。

由于存储芯片位宽为 16 位,一次可以进行 2 字节的读取。但是,通常操作系统里最小寻址单位是 1 字节,因此内存控制器必须要保证可以访问内存里每一个字节。UDQM,LDQM 分别代表 16 位数据的高,低字节读取信号,当读取数据时,LDQM/UDQM 分别用来控制 16 位数据中高低字节能否被读取,当 LDQM/UDQM 为低电平时,对应的高/低字节就可以被读取,如果 LDQM/UDQM 为高电平时,对应的高/低字节就不能被读取。当向内存里写入数据时,LDQM/UDQM 控制数据能否被写入,当 LDQM/UDQM 为低电平时,对应的高/低字节就可以被写入,如果 LDQM/UDQM 为高电平时,对应的高/低字节就不能被写入。通过对 LDQM/UDQM 信号的控制可以控制对两个存储芯片存储数据,由于两个存储单元的地址线是通用的,它们都能接收到 CPU 发出的地址信号,但是,发给两个存储单元的 LDQM/UDQM 信号是不同的,以此来区分一个字的高低字节。

S3C2440A 为 32 位 CPU,也就是说其数据总线和地址总线宽度都是 32 位(可以理解为 32 根线一端连接 CPU 内部,另外一端连接向内存控制器),那么内存数据的输入/输出端也要保证是 32 位总线,MINI2440 上采用两片 16 位宽总线内存芯片并联构成 32 位总线。其中一个芯片连接到 CPU 数据总线的低 16 位,另外一个芯片连接到数据总线上的高 16 位,并联成 32 位总线,因此两个芯片的输入/输出总线连接到 CPU 总线上的不同引脚上。

2.6.3 SDRAM 的读操作

如图 2-53 所示,SDRAM 进行读操作时,先向地址线上送上要读取数据的地址,通过前面的知识了解到,地址被分成三部分,行地址、列地址、L-Bank 片选信号。片选(L-Bank 的定址)操作和行有效操作可以同时进行。

在 CS、L-Bank 定址的同时,RAS(nSRAS 行地址选通信号)也处于有效状态。此时 An 地址线则发送具体的行地址。A0~A12,共有 13 根地址线(可表示 8192 行),A0~A12 的不同数值就确定了具体的行地址。由于行有效的同时也是相应 L-Bank 有效,所以行有效也可称为 L-Bank 有效。

行地址确定之后,就要对列地址进行寻址了。但是,地址线仍然是行地址所用的 A0~A12。没错,在 SDRAM 中,行地址与列地址线是复用的。列地址复用了 A0~A8,共 9 根(可表示 512 列)。那么,读/写的命令是怎么发出的呢?其实没有一个信号是发送读或写的明确命令的,而是通过芯片的可写状态的控制来达到读/写的目的。显然 WE 信号(nWE)就是一个关键。WE 无效时,当然就是读取命令。有效时,就是写命令。

SDRAM 基本操作命令,通过各种控制/地址信号的组合来完成(H 代表高电平,L 代表低电平,X 表示高,低电平均没有影响)。此表中,除了自刷新命令外,所有命令都是默认 CKE (SCKE1 输入时钟频率有效)有效。列寻址信号与读写命令是同时发出的。虽然地址线与行

寻址共用,但 CAS(nSCAS 列地址选通信号)信号则可以区分开行与列寻址的不同,配合 A0~A8,A9~A11 来确定具体的列地址。

读取命令与列地址一块发出(当 WE 为低电平是即为写命令)然而,在发送列读写命令时必须要与行有效命令有一个间隔,这个间隔被定义为 t_{RCD} ,即 RAS to CAS Delay(RAS 至 CAS 延迟),这个很好理解,在地址线上送完行地址之后,要等到行地址稳定定位后再送出列地址, t_{RCD} 是 SDRAM 的一个重要时序参数,相关数值参看对应芯片硬件手册。通常 t_{RCD} 以时钟周期(t_{CK} ,Clock Time)数为单位,比如笔者 MINI2440 所用内存芯片里面写到 t_{RCD} 为 20nst,如果将来内存工作在 100MHz,那么 RCD 至少要为两个时钟周期, $RCD=2$ 。

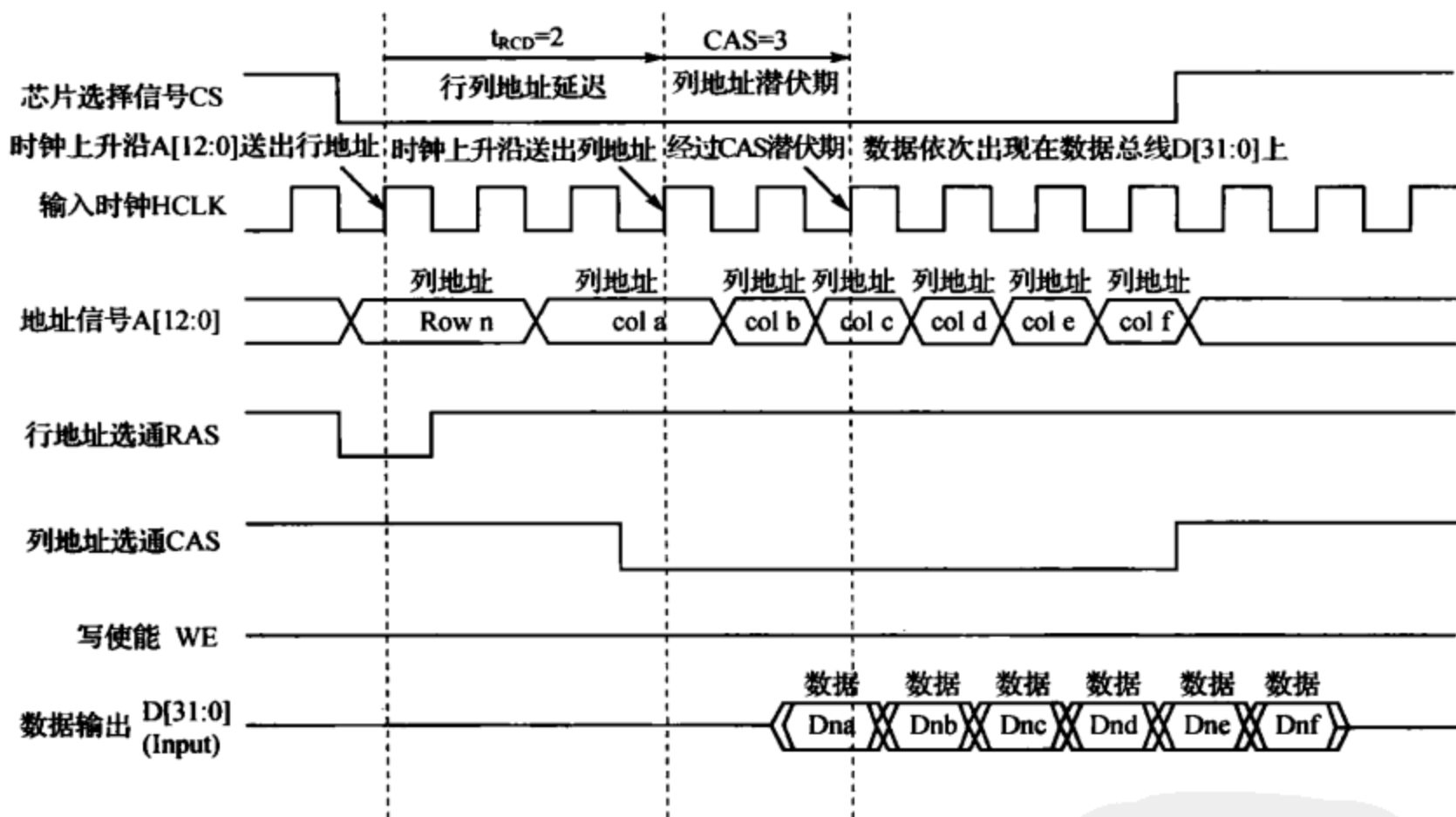


图 2-53 SDRAM 读操作时序图

在选定列地址后,就已经确定了具体的存储单元,剩下就是等待数据通过数据 I/O 通道(DQ)输出到内存数据总线上了。但是在列地址选通信号 CAS 发出之后,仍要经过一定的时间才能有数据输出,从 CAS 与读取命令发出到第一笔数据输出的这段时间,被定义为 CL (CAS Latency,CAS 潜伏期)。由于 CL 只在读取时出现,所以 CL 又称为读取潜伏期(Read Latency,RL)。CL 的单位与 t_{RCD} 一样,也是时钟周期数,具体耗时由时钟频率决定(笔者官方手册 CL=3)。不过,CAS 并不是在经过 CL 周期之后才送达存储单元。实际上 CAS 与 RAS 一样是瞬间到达的。由于芯片体积的原因,存储单元中的电容容量很小,所以信号要经过放大来保证其有效的识别性,这个放大/驱动工作由 S-AMP 负责。但它要有一个准备时

间才能保证信号的发送强度,这段时间称为 t_{AC} (Access Time from CLK, 时钟触发后的访问时间)。

2.6.4 SDRAM 预充电操作

从存储体的结构图上可以看出,原本逻辑状态为 1 的电容在读取操作后,会因放电而变为逻辑 0。由于 SDRAM 的寻址具有独占性,所以在进行完读写操作后,如果要对同一 L-Bank 的另一行进行寻址,就要将原先操作行关闭,重新发送行/列地址。在对原先操作行进行关闭时,DRAM 为了在关闭当前行时保持数据,要对存储体中原有的信息进行重写,这个充电重写和关闭操作行过程称做预充电,发送预充电信号时,意味着先执行存储体充电,然后关闭当前 L-Bank 操作行。预充电中重写的操作与刷新操作(后面详细介绍)一样,只不过预充电不是定期的,而只是在读操作以后执行的。

2.6.5 SDRAM 突发操作

突发(Burst)是指在同一行中相邻的存储单元连续进行数据传输的方式,连续传输所涉及存储单元(列)的数量就是突发长度(Burst Length, BL)。

在目前,由于内存控制器一次读/写 P-Bank 位宽的数据,也就是 8 字节,但是在现实中小于 8 字节的数据很少见,所以一般都要经过多个周期进行数据的传输,上文写到的读/写操作,都是一次对一个存储单元进行寻址,如果要连续读/写,还要对当前存储单元的下一单元进行寻址,也就是要不断地发送列地址与读/写命令(行地址不变,所以不用再对地寻址)。虽然由于读/写延迟相同可以让数据传输在 I/O 端是连续的,但是它占用了大量的内存控制资源,在数据进行连续传输时无法输入新的命令效率很低。为此,引入了突发传输机制,只要指定起始列地址与突发长度,内存就会依次自动对后面相应长度数据的数据存储单元进行读/写操作而不再需要控制器连续地提供列地址,这样,除了第一笔数据的传输需要若干个周期(主要是之间的延迟,一般的是 $t_{RCD} + CL$)外,其后每个数据只需一个周期即可。

总结如下:

SDRAM 的基本读操作需要控制线和地址线相配合地发出一系列命令来完成。先发出芯片有效命令(ACTIVE),并锁定相应的 L-BANK 地址(BA0、BA1 给出)和行地址(A0~A12 给出)。芯片激活命令后必须等待大于 t_{RCD} (SDRAM 的 RAS 到 CAS 的延迟指标)时间后,发出读命令。CL(CAS 延迟值)个时钟周期后,读出数据依次出现在数据总线上。在读操作的最后,要向 SDRAM 发出预充电(PRECHARGE)命令,以关闭已经激活的 L-BANK。等待 t_{RP} 时间(PRECHARGE 命令后,相隔 t_{RP} 时间,才可再次访问该行)后,可以开始下一次的读、写操作。SDRAM 的读操作支持突发模式(Burst Mode),突发长度为 1、2、4、8 可选。

2.6.6 SDRAM 写操作

如图 2-54 所示,SDRAM 的基本写操作也需要控制线和地址线相配合地发出一系列命令来完成。先发出芯片有效命令,并锁定相应的 L-Bank 地址(BA0、BA1 给出)和行地址(A0~A12 给出)。芯片有效命令发出后必须等待大于 t_{RCD} 的时间后,发出写命令数据,待写入数据依次送到 DQ(数据线)上。在最后一个数据写入后,延迟 t_{WR} 时间。发出预充电命令,关闭已经激活的页。等待 t_{RP} 时间后,可以展开下一次操作。写操作可以有突发写和非突发写两种。突发长度同读操作。

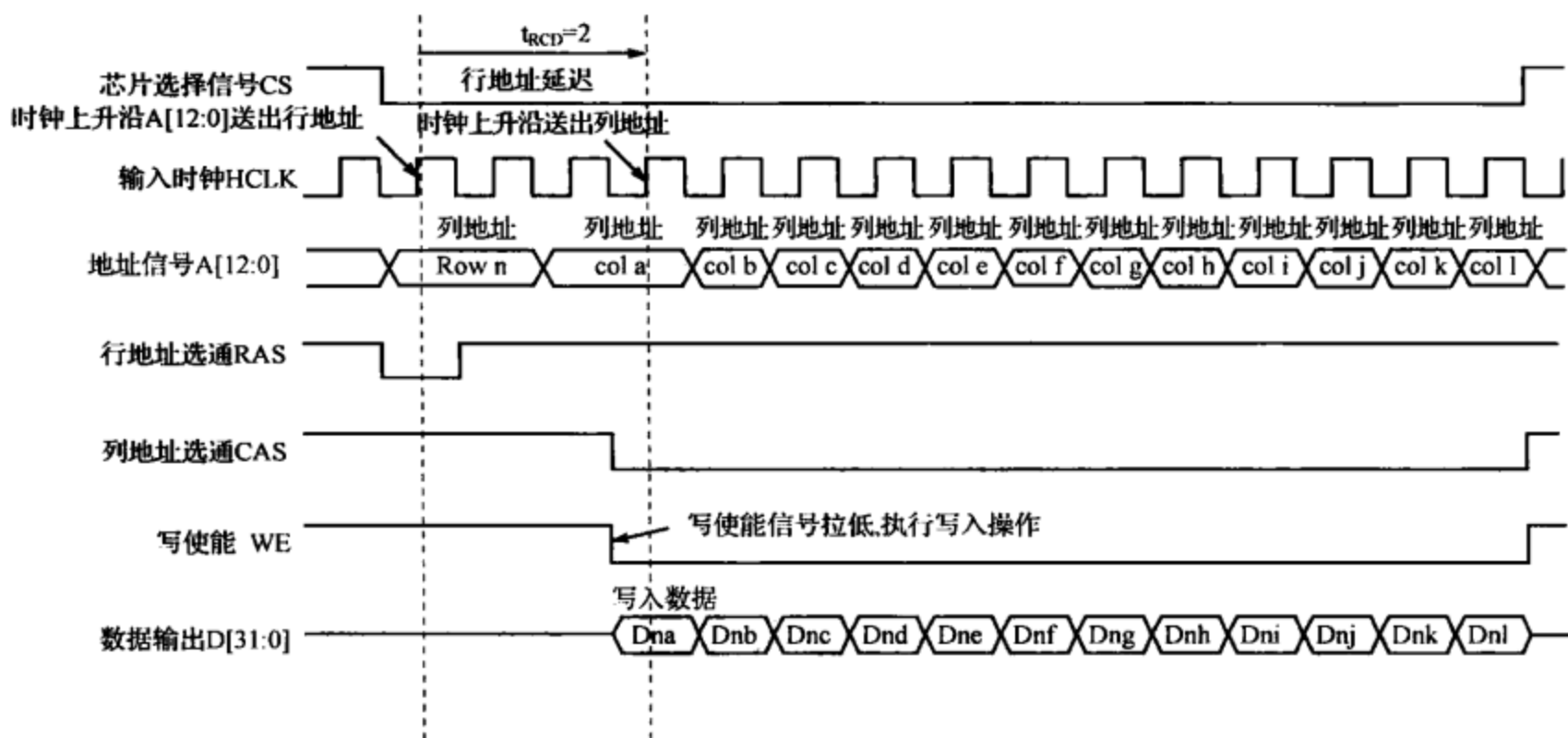


图 2-54 SDRAM 写操作时序图

2.6.7 SDRAM 的刷新

SDRAM 之所以成为 DRAM 就是因为它要不断进行刷新(Refresh)才能保留住数据,因此它是 SDRAM 最重要的操作。

刷新操作与预充电中重写的操作一样,都是用 S-AMP 先读再写。但为什么有预充电操作还要进行刷新呢?因为预充电是对一个或所有 L-Bank 中的工作行操作,并且是不定期的,而刷新则是有固定的周期,依次对所有行进行操作,以保留那些很长时间没经历重写的存储体中的数据。但与所有 L-Bank 预充电不同的是,这里的行是指所有 L-Bank 中地址相同的行,而预充电中各 L-Bank 中的工作行地址并不是一定是相同的。那么要隔多长时间重复一次刷新呢?目前公认的标准是,存储体中电容的数据有效保存期上限是 64ms(毫秒,1/1000 秒),也就是说每一行刷新的循环周期是 64ms。这样刷新速度就是:行数/64ms。在看内存规

格时,经常会看到 4096 Refresh Cycles/64ms 或 8192 Refresh Cycles/64ms 的标识,这里的 4096 与 8192 就代表这个芯片中每个 L-Bank 的行数。刷新命令一次对一行有效,发送间隔也是随总行数而变化,4096 行时为 $15.625\mu\text{s}$ (微秒,1/1000 毫秒),8192 行时即为 $7.8125\mu\text{s}$ 。刷新操作分为两种:Auto Refresh,简称 AR 与 Self Refresh,简称 SR。不论是何种刷新方式,都不需要外部提供行地址信息,因为这是一个内部的自动操作。对于 AR,SDRAM 内部有一个行地址生成器(也称刷新计数器)用来自动的依次生成行地址。由于刷新是针对一行中的所有存储体进行,所以无需列寻址,或者说 CAS 在 RAS 之前有效。所以,AR 又称 CBR(CAS Before RAS,列提前于行定位)式刷新。由于刷新涉及所有 L-Bank,因此在刷新过程中,所有 L-Bank 都停止工作,而每次刷新所占用的时间为 9 个时钟周期(PC133 标准),之后就可进入正常的工作状态,也就是说在这 9 个时钟期间内,所有工作指令只能等待而无法执行。64ms 之后则再次对同一行进行刷新,如此周而复始进行循环刷新。显然,刷新操作肯定会对 SDRAM 的性能造成影响,但这是没办法的事情,也是 DRAM 相对于 SRAM(静态内存,无需刷新仍能保留数据)取得成本优势的同时所付出的代价。SR 则主要用于休眠模式低功耗状态下的数据保存,这方面最著名的应用就是 STR(Suspend to RAM,休眠挂起于内存)。在发出 AR 命令时,将 CKE 置于无效状态,就进入了 SR 模式,此时不再依靠系统时钟工作,而是根据内部的时钟进行刷新操作。在 SR 期间除了 CKE 之外的所有外部信号都是无效的(无需外部提供刷新指令),只有重新使 CKE 有效才能退出自刷新模式并进入正常操作状态。

SDRAM 相关寄存器:

(1)BWSCON 寄存器(BUS WIDTH & WAIT CONTROL REGISTER)如表 2-17 所列。

表 2-17 SDRAM 控制寄存器(BWSCON)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|--------|------------|-----|--|----------|
| BWSCON | 0x48000000 | R/W | Bus width & wait status control register | 0x000000 |
| BWSCON | 位 | | 描 述 | 初始值 |
| ST7 | [31] | | Determines SRAM for using UB/LB for bank 7. 0=Not using UB/LB(The pins are dedicated nWBE[3:0]) 1=Using UB/LB(The pins are dedicated nBE[3:0]) | 0 |
| WS7 | [30] | | Determines WAIT status for bank 7. 0=WAIT disable 1=WAIT enable | 0 |
| DW7 | [29:28] | | Determines data bus width for bank 7. 00=8-bit 01=16-bit, 10=32-bit 11=reserved | 0 |

续表 2-17

| BWSCON | 位 | 描 述 | 初始值 |
|----------|---------|--|-----|
| ST6 | [27] | Determines SRAM for using UB/LB for bank 6. 0=Not using UB/LB(The pins are dedicated nWBE[3:0]) 1=Using UB/LB(The pins are dedicated nBE[3:0]) | 0 |
| WS6 | [26] | Determines WAIT status for bank 6. 0=WAIT disable. 1=WAIT enable | 0 |
| DW6 | [25:24] | Determines data bus width for bank 6. 00=8-bit 01=16-bit, 10=32-bit 11=reserved | 0 |
| DW0 | [2:1] | Indicate data bus width for bank 0(read only). 01=16-bit, 10=32-bit The states are selected by OM[1:0] pins | — |
| Reserved | [0] | Reserve to 0 | 0 |

根据开发板的存储器配置和芯片型号,设置每个 Bank 焊接芯片的位宽和等待状态 BWSCON,每 4 位对应一个 Bank,这 4 位分别表示:

① ST_x:启动/禁止 SDRAM 的数据掩码引脚(UB/LB),SDRAM 没有高低位掩码引脚,此位为 0,SRAM 连接有 UB/LB 引脚,设置为 1。

② 注:UB/LB 数据掩码引脚用来控制芯片读取/写入的高字节和低字节(对比硬件手册 SDRAM 和 SRAM 的接线图)。

③ WS_x:是否使用存储器的 WAIT 信号,通常设为 0。

④ DW_x:设置焊接存储器芯片的位宽,笔者开发板使用两片容量为 32MB,位宽为 16 的 SDRAM 组成 64MB,32 位存储器,因此 DW7,DW6 位设置为 0b10,其他 BANK 不用设置采用默认值即可。

⑤ BANK0 对应的是系统引导 BANK,这 4 位比较特殊,它的设置是由硬件跳线决定的,因此不用设置。

⑥ BWSCON 设置结果:0x22000000。

(2)BANKCON0~BANKCON5 (BANK CONTROL REGISTER)如表 2-18 所列。

表 2-18 BANKCON0~BANKCON5 控制寄存器(BANKCON0~BANKCON5)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|-----|-------------------------|--------|
| BANKCON0 | 0x48000004 | R/W | Bank 0 control register | 0x0700 |
| BANKCON1 | 0x48000008 | R/W | Bank 1 control register | 0x0700 |
| BANKCON2 | 0x4800000C | R/W | Bank 2 control register | 0x0700 |
| BANKCON3 | 0x48000010 | R/W | Bank 3 control register | 0x0700 |

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|-----|-------------------------|--------|
| BANKCON4 | 0x48000014 | R/W | Bank 4 control register | 0x0700 |
| BANKCON5 | 0x48000018 | R/W | Bank 5 control register | 0x0700 |

这 6 个寄存器用来设置对应 BANK0~BANK5 的访问时序,采用默认值 0x700 即可。

(3) BANKCON6~BANKCON7 (BANK CONTROL REGISTER)如表 2-19 所列。

表 2-19 BANKCON6~BANKCON7 控制寄存器 (BANKCON6~BANKCON7)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|---------------------------------|------------|--|-------------------------|---------|
| BANKCON6 | 0x4800001C | R/W | Bank 6 control register | 0x18008 |
| BANKCON7 | 0x48000020 | R/W | Bank 7 control register | 0x18008 |
| BANKCONn | 位 | 描 述 | | 初始值 |
| MT | [16 : 15] | Determine the memory type for bank6 and bank7. 00=ROM or SRAM 01=Reserved (Do not use) 10=Reserved(Do not use) 11=Sync. DRAM | | 11 |
| Memory Type=SDRAM [MT=11](4bit) | | | | |
| Trcd | [3 : 2] | RAS to CAS delay 00=2 clocks 01=3 clocks 10=4 clocks | | 10 |
| SCAN | [1 : 0] | Column address number 00=8bit 01=9bit 10=10bit | | 00 |

由于内存都焊接在这两个 Bank 上,因此内存驱动主要是对这两个寄存器进行设置。

① MT:设置 BANK6~BANK7 的存储器类型,

00=ROM or SRAM 01=保留

10=保留 11=SDRAM

内存为 SDRAM,设置为 0b11,对应的应该设置 Trcd 和 SCAN 位,其他位和 SDRAM 无关。

② Trcd:RAS to CAS Delay 行地址选通到列地址选通延迟,这个参数请看后面的内存工作原理扩展部分解释,笔者内存芯片为 HY57V561620,由其芯片手册可知其 Trcd 为最少 20ns,如果内存工作在 100MHz,则该值至少要为 2 个时钟周期,通常设置为 3 个时钟周期,因此设置为 0b01。

③ SCAN:SDRAM Column Address Number SDRAM 的列地址数,内存芯片为 HY57V561620,列地址数为 9,设置为 0b01。

④ BANK6, BANK7 设置结果为: 0x18005。

(4) REFRESH (REFRESH CONTROL REGISTER) 如表 2-20 所列。

表 2-20 刷新频率设置寄存器 (REFRESH)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|-----------------|------------|-----|--|----------|
| REFRESH | 0x48000024 | R/W | SDRAM refresh control register | 0xac0000 |
| REFRESH | 位 | | 描 述 | 初始值 |
| REFEN | [23] | | SDRAM Refresh Enable 0=Disable 1=Enable(self of CBR/auto refresh) | 1 |
| TREFMD | [22] | | SDRAM Refresh Mode 0=CBR/Auto Refresh 1=Self Refresh In self-refresh time, the SDRAM control signals are driven to the appropriate level. | 0 |
| Trp | [21:20] | | SDRAM RAS pre-charge Time 00=2 clocks 01=3 clocks 10=4 clocks 11=Not support | 10 |
| Tsrc | [19:18] | | SDRAM Semi Row cycle time 00=4 clocks 01=5 clocks 10=6 clocks 11=7 clocks SDRAM Row cycle time: $T_{rc} = T_{src} + T_{rp}$ If $T_{rp} = 3\text{clocks}$ & $T_{src} = 7\text{clocks}$, $T_{rc} = 3 + 7 = 10\text{clocks}$. | 11 |
| Reserved | [17:16] | | Not used | 00 |
| Reserved | [15:11] | | Not used | 0000 |
| Refresh Counter | [10:0] | | SDRAM refresh count value. Refer to chapter 6 SDRAM refresh controller bus priority section. Refresh period = $(2^{11} - \text{refresh_count} + 1) / \text{HCLK}$ Ex) If refresh period is 7.8 us and HCLK is 100 MHz, the refresh count is as follows: Refresh count = $2^{11} + 1 - 100 \times 7.8 = 1269$ | 0 |

SDRAM 的刷新有效, 刷新频率设置寄存器(刷新)。

① REFEN: 开启/关闭刷新功能, 设置为 1, 开启刷新。

② TREFMD: SDRAM 刷新模式, 0=CBR/AutoRefresh, 1=Self Refresh, 设置为 0, 自动刷新。

③ Trp: 行地址选通预充电时间, 一般设置为 0b00 即可。

④ Tsrc: 单行刷新时间, 设置为 0b11 即可。

⑤ Refresh Counter: 内存存储单元刷新数, 它通过下面公式计算出:

$\text{Refresh Counter} = 2^{11} + 1 - \text{SDRAM 时钟频率(MHz)} \times \text{SDRAM 刷新周期}(\mu\text{s})$

SDRAM 的刷新周期,也就是内存存储单元间隔需要多久进行一次刷新,前面内存工作原理分析可知电容数据保存上限为 64ms,笔者使用内存芯片每个 L-Bank 共有 8192 行,因此每次刷新最大间隔为: $64\text{ms}/8192 = 7.8125\mu\text{s}$,如果内存工作在外部晶振频率 12MHz 下, $\text{Refresh Counter} = 1955$,如果内存工作在 100MHz 下,那么 $\text{Refresh Counter} = 1269$ (取最大整数)。

⑥ REFRESH 寄存器设置为:

$0x8e0000 + 1269 = 0x008e04f5$ (HCLK = 100MHz)

$0x8e0000 + 1955 = 0x008e07a3$ (HCLK = 12MHz)

(5) BANKSIZE 寄存器(BANKSIZE REGISTER)如表 2-21 所列。

表 2-21 BANKSIZE 寄存器(BANKSIZE)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|---|----------------------------|-------|
| BANKSIZE | 0x48000028 | R/W | Flexiblebank size register | 0x0 |
| BANKSIZE | 位 | 描 述 | | 初始值 |
| BURST_EN | [7] | ARM core burst operation enable. 0=Disable burst operation. 1=Enable burst operation. | | 0 |
| Reserved | [6] | Not used | | 0 |
| SCKE_EN | [5] | SDRAM power down mode enable control by SCKE 0=SDRAM power down mode disable 1=SDRAM power down mode enable | | 0 |
| SCLK_EN | [4] | SCLK is enabled only during SDRAM access cycle for reducing power consumption. When SDRAM is not accessed, SCLK becomes 'L' level. 0=SCLK is always active. 1=SCLK is active only during the access(recommended). | | 0 |
| Reserved | [3] | Not used | | 0 |
| BK76MAP | [2:0] | BANK6/7 memory map 010=128MB/128MB 001=64MB/64MB 000=32M/32M 111=16M/16M 110=8M/8M 101=4M/4M 100=2M/2M | | 010 |

设置内存的突发传输模式,省电模式和内存容量。

① BURST_EN:是否开启突发模式,0=ARM 内核禁止突发传输,1=开启突发传输,设

置为 1, 开启突发传输。

② SCKE_EN: 是否使用 SCKE 信号作为省电模式控制信号, 0 = 不使用 SCKE 信号作为省电模式控制信号, 1 = 使用 SCKE 信号作为省电模式控制信号, 通常设置为 1。

③ SCLK_EN: 设置向存储器输入工作频率, 0 = 一直输入 SCLK 频率, 即使没有内存操作也会输入, 1 = 仅当进行内存数据操作时才输入 SCLK 频率, 通常设置为 1。

④ BK76MAP: 设置 Bank6/7 的内存容量, 使用开发板内存为两片 32MB 内存芯片并联成 64MB, 它们全部都外接到 Bank6 上, 因此选择 0b001。

⑤ BANKSIZE 寄存器设置为: 0xb1。

(6) SDRAM 模式设置寄存器 MRSR_x (SDRAM MODE REGISTER SET REGISTER) 如表 2-22 所列。

表 2-22 SDRAM 模式设置寄存器 (MRSR_x)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|--|----------------------------------|-------|
| MRSRB6 | 0x4800002C | R/W | Mode register set register bank6 | xxx |
| MRSRB7 | 0x48000030 | R/W | Mode register set register bank7 | xxx |
| MRSR | 位 | Description | | 初始值 |
| Reserved | [11 : 10] | Not used | | — |
| WBL | [9] | Write burst length 0: Burst(Fixed) 1: Reserved | | x |
| TM | [8 : 7] | Test mode 00: Mode register set(Fixed) 01, 10 and 11: Reserved | | xx |
| CL | [6 : 4] | CAS latency 000 = 1 clock, 010 = 2 clocks, 011 = 3 clocks Others: Reserved | | xxx |
| BT | [3] | Burst type 0: Sequential(Fixed) 1: Reserved | | x |
| BL | [2 : 0] | Burst length 000: 1(Fixed) Others: Reserved | | xxx |

该寄存器用于设置 CAS 潜伏周期, 可以手动设置的位只有 CL[6 : 4]位, 通过前面内存工作原理可知, 使用开发板 CL = 3, 即 0b011。

● MRSR6, MRSR7 设置为: 0x00000030。

2.6.8 内存驱动实验

本实验源码存放在光盘目录\work\armarch\mem_init 工程目录下。
设置该工程加载时运行时地址为 0x30000000,如图 2-55 所示。

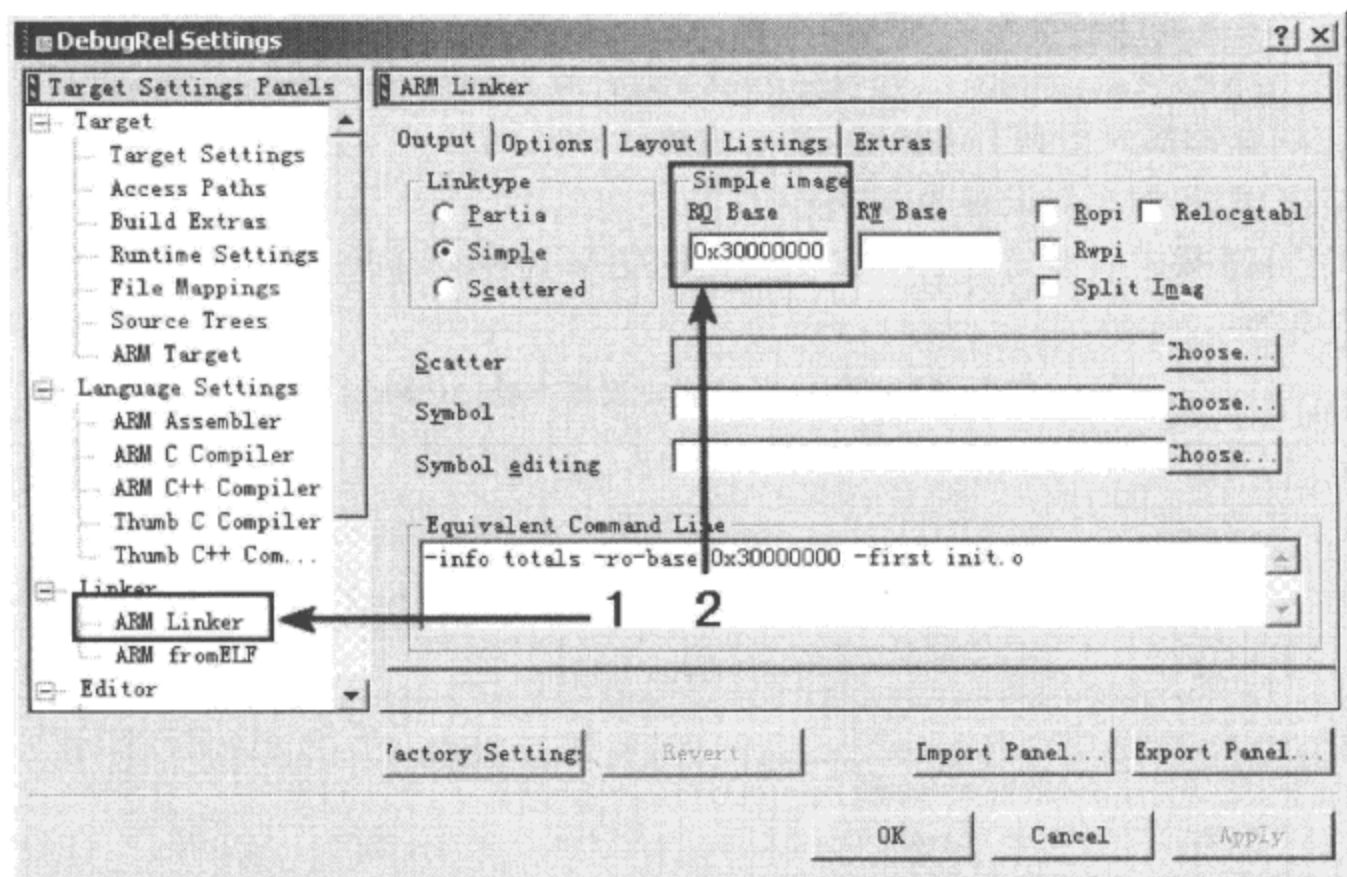


图 2-55 设置加载时运行时地址

init.s:本程序文件主要实现了:关闭看门狗,初始化内存,复制 ROM 数据到内存中,然后跳往内存中执行 xmain 函数,从 xmain 函数返回之后,将全部 LED 点亮,进入死循环。

```

;
; 内存初始化实验
;
    AREA Init, CODE, READONLY
    ENTRY
start
; close watchdog
    ldr r0, = 0x53000000          ; 将看门狗控制寄存器地址放入 r0
    mov r1, #0
    str r1, [r0]                 ; 设置看门狗控制寄存器的值为 0

    bl initmem                   ; 跳转到 initmem 代码段,初始化内存

```

资源库
PDG

```

bl copyall

; 跳转到数据复制代码段,将 ROM 中数据复制到内存中

IMPORT xmain
ldr sp, = 0x34000000
ldr lr, = endxmain
ldr pc, = xmain

; 引入 main.c 中的 xmain 函数
; 调用 C 程序之前先初始化栈指针
; 设置 xmain 函数的返回地址
; 跳转到 C 程序中的 xmain 函数的入口处执行

endxmain
ldr r0, = 0x56000010
ldr r1, = 0x00015400
str r1, [r0]
ldr r0, = 0x56000014
ldr r1, = 0x0
str r1, [r0]

; LED 的 GPIO 接口配置寄存器
; GPIO 配置数据
; 设置 GPIO
; LED 控制寄存器地址
; 全部 LED 亮

loop
b loop

; 死循环

copyall
IMPORT |Image$ $RO$ $Base|
IMPORT |Image$ $RW$ $Limit|
ldr r0, = |Image$ $RO$ $Base|
ldr r1, = |Image$ $RW$ $Limit|
ldr r2, = 0x0

; 引入编译器 Image$ $RO$ $Base 符号变量
; 引入编译器 Image$ $RW$ $Limit 符号变量
; 取得 Image$ $RO$ $Base 域基址的值
; 取得 Image$ $RW$ $Base 域结束地址的值
; 数据复制源地址

copyallloop
teq r0,r1
beq quitcopyallloop
ldr r3, [r2], #4
str r3, [r0], #4
b copyallloop

; 测试是否复制完成
; 复制完成,跳往 quitcopyallloop 退出
; 4 字节加载
; 4 字节存储
; 返回继续执行

quitcopyallloop
mov pc, lr

; 调用返回

initmem
ldr r0, = 0x48000000
ldr r1, = 0x48000034
adr r2, memdata

; 内存初始化
; 加载内存相关寄存器首地址 r0
; 加载内存相关寄存器尾地址到 r1
; 将寄存器配置数据地址段首地址加载到 r2

initmemloop
ldr r3, [r2], #4
str r3, [r0], #4

; 循环设置寄存器

```

资源分享网
PDG


```

teq r0, r1
bne initmemloop           ; 循环到最后一个寄存器时退出函数
mov pc, lr

```

```

memdata
DCD    0x22000000          ;BWSCON
DCD    0x00000700          ;BANKCON0
DCD    0x00000700          ;BANKCON1
DCD    0x00000700          ;BANKCON2
DCD    0x00000700          ;BANKCON3
DCD    0x00000700          ;BANKCON4
DCD    0x00000700          ;BANKCON5
DCD    0x00018005          ;BANKCON6
DCD    0x00018005          ;BANKCON7
DCD    0x008e07a3          ;REFRESH
DCD    0x000000b1          ;BANKSIZE
DCD    0x00000030          ;MRSRB6
DCD    0x00000030          ;MRSRB7

```

END

main.c:本程序文件主要实现 LED 灯的初始化,然后 4 个 LED 灯循环滚动亮 5 遍,xmain 函数返回。

```

/* C 语言函数 */
/* 端口 F 寄存器预定义 */
#define GPBCON      (*(volatile unsigned long *)0x56000010)
#define GPBDAT      (*(volatile unsigned long *)0x56000014)
#define LEDS        (1<<5|1<<6|1<<7|1<<8)
#define DELAYVAL    (0x1ffff)
extern int delay(int time); /* 声明外部声明汇编函数 */

int i = 5;
int xmain()
{
    GPBCON = 0x00015400; //GPF4~GPF7 设置为 output
    while(i > 0) {
        //第一个 LED 灯亮
        GPBDAT = (GPBDAT & (~LEDS)) | (1<<6|1<<7|1<<8);
        delay(DELAYVAL); //调用汇编语言编写的延时程序
    }
}

```

```

//第二个 LED 灯亮
GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<7|1<<8);
delay(DELAYVAL);           //调用汇编语言编写的延时程序

//第三个 LED 灯亮
GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<6|1<<8);
delay(DELAYVAL);           //调用汇编语言编写的延时程序

//第四个 LED 灯亮
GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<6|1<<7);
delay(DELAYVAL);           //调用汇编语言编写的延时程序
i--;
}
return 0;
}

```

delay.s: 本程序文件主要通过汇编指令来实现延时功能。

; 汇编指令延时程序

EXPORT delay

AREA DELAY, CODE, READONLY ; 该伪指令定义了一个代码段, 段名为 Init, 属性只读

; 下面是延迟子程序

delay

```

sub r0, r0, #1      ; r0 = r0 - 1
cmp r0, #0x0        ; 将 r0 的值与 0 相比较
bne delay           ; 比较的结果不为 0 (r0 不为 0), 继续调用 delay, 否则执行下一条语句
mov pc, lr          ; 返回

```

END ; 程序结束符

内存的初始化也可以用下面的 C 程序实现:

C 语言版本:

```

#define MEM_CTL_BASE 0x48000000
#define MEM_CTL_END 0x48000034
/* SDRAM 13 个寄存器的值 */
unsigned long const mem_cfg_val[] = { // 声明数组存放内存控制器设置数据
    0x22000000, // BWSCON
    0x00000700, // BANKCON0
    0x00000700, // BANKCON1
    0x00000700, // BANKCON2

```

```

        0x00000700,      //BANKCON3
        0x00000700,      //BANKCON4
        0x00000700,      //BANKCON5
        0x00018005,      //BANKCON6
        0x00018005,      //BANKCON7
        0x008e07a3,      //REFRESH(HCLK = 12MHz,该值为 0x008e07a3
                          //HCLK = 100MHz 0x008e04f5)
        0x000000b1,      //BANKSIZE
        0x00000030,      //MRSRB6
        0x00000030,      //MRSRB7
    };
    void mem_init(void)
    {
        int i = 0;
        unsigned long *p = (unsigned long *)MEM_CTL_BASE;
        for(; i < (MEM_CTL_END - MEM_CTL_BASE)/4; i++)
            p[i] = mem_cfg_val[i];
    }

```

2.7 UART 串口

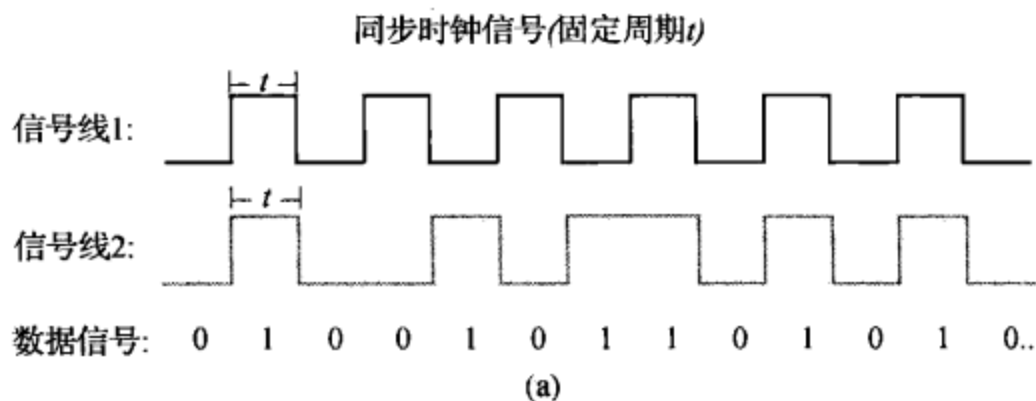
通用异步接收器和发送器(Universal Asynchronous Receiver and Transmitter, UART)。通常是嵌入式设备中默认都会配置的通信接口。这是因为,很多嵌入式设备没有显示屏,无法获得嵌入式设备实时数据信息,通过 UART 串口和超级终端相连,打印嵌入式设备输出信息。并且在对嵌入式系统进行跟踪和调试时,UART 串口是必要的通信手段。比如网络路由器、交换机等都要通过串口来进行配置。UART 串口还是许多硬件数据输出的主要接口,如 GPS 接收器就是通过 UART 串口输出 GPS 接收数据的。

2.7.1 同步通信和异步通信

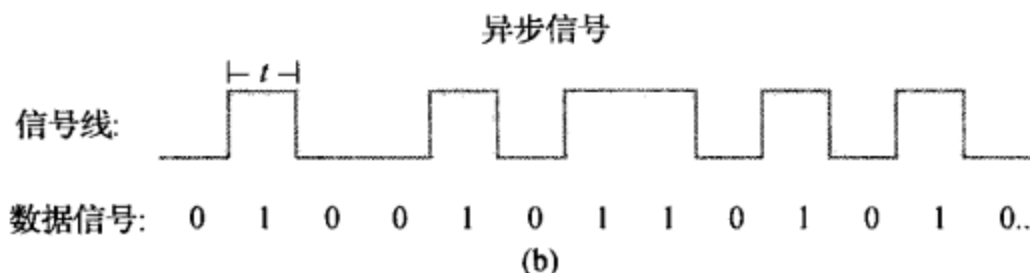
同步通信与异步通信如图 2-56 所示。

1. 同步通信技术

在发送数据信号的时候,会同时送出一根同步时钟信号,用来同步发送方和接收方的数据采样频率。如图 2-56 所示,同步通信时,信号线 1 是一根同步时钟信号线,以固定的频率进行电平的切换,其频率周期为 t ,在每个电平的上升沿之后进行对同步送出的数据信号线 2 进行采样(高电平代表 1,低电平代表 0),根据采样数据电平高低取得输出数据信息。如果双方没有同步时钟的话,那么接收方就不知道采样周期,也就不能正常的取得数据信息。



注: 时钟信号电平跳变时开始取数据信号, 其周期 t 与时钟频率有关



注: 发送端、接收端使用同一速率(波特率 b/s)来取数据信号, 其周期为 $t=1/b/s$

图 2-56 同步信号与异步信号

2. 异步通信技术

在异步通信技术中, 数据发送方和数据接收方没有同步时钟, 只有数据信号线, 只不过发送端和接收端会按照协商好的协议(固定频率)来进行数据采样。数据发送方以 57 600 b/s 的速度发送数据, 接收方也以 57 600 b/s 的速度去接收数据, 这样就可以保证数据的有效和正确。通常异步通信中使用波特率来规定双方传输速度, 其单位为 bps(bit per second 每秒传输位数)。

2.7.2 数据的串行和并行通信方式

串行通信好比是一列纵队, 每个数据元素依次纵向排列。如图 2-57 所示, 传输时一个比特一个比特地串行传输, 每个时钟周期传输一个比特, 这种传输方式相对比较简单, 但是使用总线数较少, 通常一根接收线, 一根发送线即可实现串行通信。它的缺点是要增加额外的数据来控制一个数据帧的开始和结束。

并行通信好比一排横队, 齐头并进同时传输。这种通信方式每个时钟周期传输的数据量和其总线宽度成正比, 但是实现较为复杂。UART 通信采用的是串行方式进行通信的。

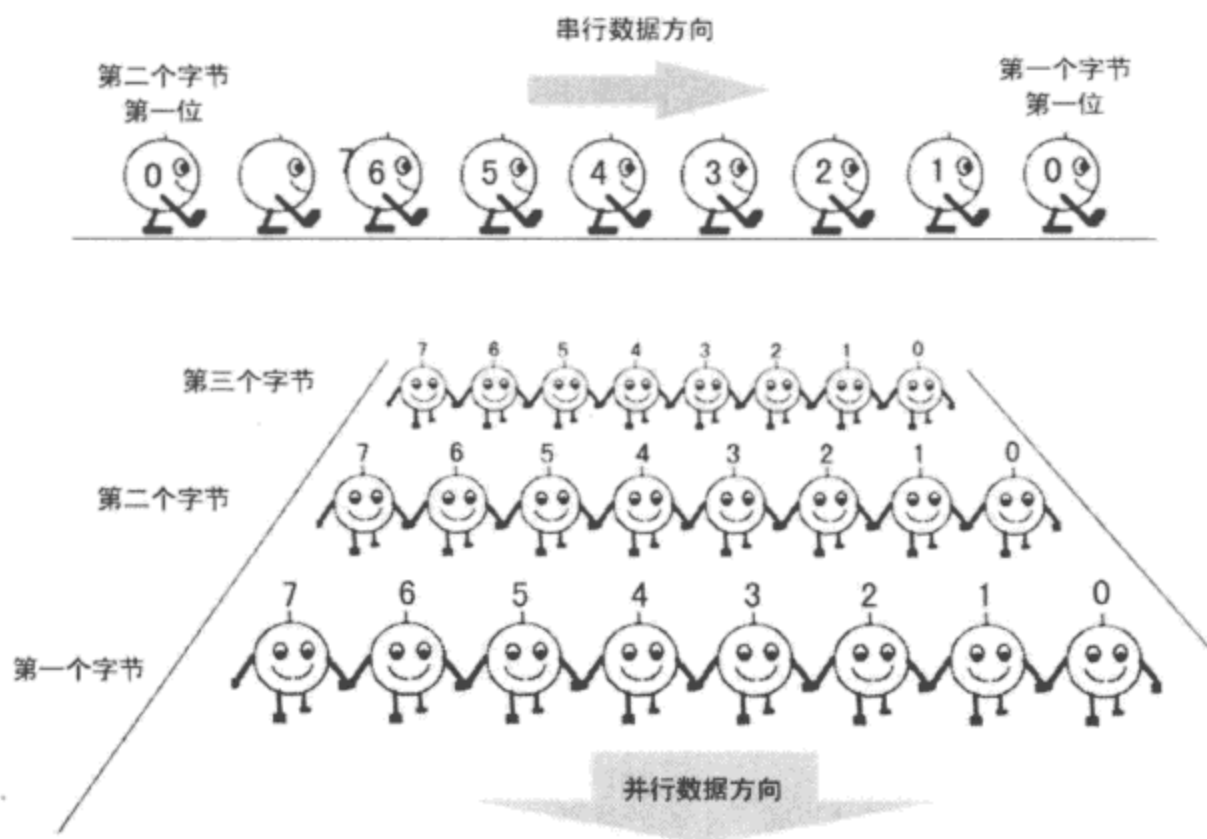


图 2-57 串行数据通信与并行数据通信

2.7.3 数据通信传输模式

在数据通信过程中,发送方和接收方为了实现数据的正确发送和接收,通常会有一个状态寄存器来描述当前数据的接收和发送状态,当发送方有数据发送时,会查看发送状态寄存器,看是否允许发送数据(如果上一次数据还没有发送完毕,不允许继续数据发送),在发送允许情况下再送出新数据。同样,接收端通过查看接收状态寄存器,确定是否有新数据到达,如果有数据到达,将去接收数据缓冲区读取数据。

(1) 轮询模式。通过程序执行流,不停地检测状态寄存器的结果,如果当前可发送或接收,则发送或接收数据。其过程可以用下面伪代码来表示。

；轮询方式实现数据发送伪代码

```
Send(){
    While(1){
        if(发送状态 == 可发送)
            执行数据发送操作;
    }
}
```

；轮询方式实现数据接收伪代码

```
Receive(){
```

```
While(1){  
    if(接收状态 == 有数据到达)  
        执行数据接收操作;  
}
```

由程序可知,这种方式实现简单,但在进行数据接收和发送时都要进入循环检查状态寄存器的值,当没有数据到达或数据不可发送时,CPU 会一直空转,其他程序又得不到 CPU 的执行权,很影响系统的效率。

(2) 中断模式。中断方式是指,当数据到达或数据可发送时,产生中断,通知 CPU 去发送或接收数据,这种方式将通信硬件和 CPU 独立出来,通信硬件只有在发送或接收条件准备好之后中,才通知 CPU 去处理数据,在通信条件没有准备好的时候,CPU 去处理其他程序,显然这种方式更合理,这种方式要求通信硬件要求比较高,需要支持产生中断信号。

(3) DMA 模式。通常实现数据的转移或复制时,CPU 将从源地址处复制数据到寄存器,然后将寄存器数据再写入目的地址处,该复制过程需要 CPU 来执行。S3C2440 支持 DMA 传输模式,DMA 传输是指在 CPU 不干涉的情况下,DMA 硬件自动实现数据的转移和复制,在 DMA 传输过程中,CPU 几乎不用干涉,这样可以让 CPU 安心的去做自己的事情。虽然如此,但是 DMA 在传输数据过程中要占用总线,在大批数据传输时,系统总线会被 DMA 通道占用,也会影响系统的效率。S3C2440 UART 控制器支持 DMA 方式传输串口通信数据。

2.7.4 S3C2440 UART 控制器

S3C2440 UART 控制器,提供了 3 个独立的异步串行 I/O 端口,每个端口都可以在中断模式或 DMA 模式下工作,换言之,UART 可以生成中断或 DMA 请求用于 CPU 和 UART 之间的数据传输。UART 串口挂接在 APB 总线上,APB 总线最高可以达到 50MHz 工作频率,在使用 APB 时钟频率时可以达到最高 115.2 kb/s 波特率的通信速度。如果 UART 串口接收外部设备提供 UEXTCLK(外部时钟),UART 可以在更高的速度下工作。每个 UART 串口在接收装置和发送装置里分别包含一个 64B 的 FIFO 缓冲区,用于缓存发送数据和接收数据。

由于 UART 是串行异步通信方式,因此在 UART 通信过程中每次只能传输 1 位(bit),若干位组成一个数据帧(frame),帧是 UART 通信中最基本单元,它主要包含开始位、数据位、校验位(如果开启了数据校验,要包含校验位)和停止位,帧结构如图 2-58 所示。

UART 在通信之前要在发送端和接收端约定好帧结构,也就是约定好传输数据帧格式。

- (1) 开始位:必须包含在数据帧中,表示一个帧的开始。
- (2) 数据位:可选 5、6、7、8 位,该位长度可由编程人员指定。
- (3) 校验位:如果在开启了数据校验时,该位必须指定。

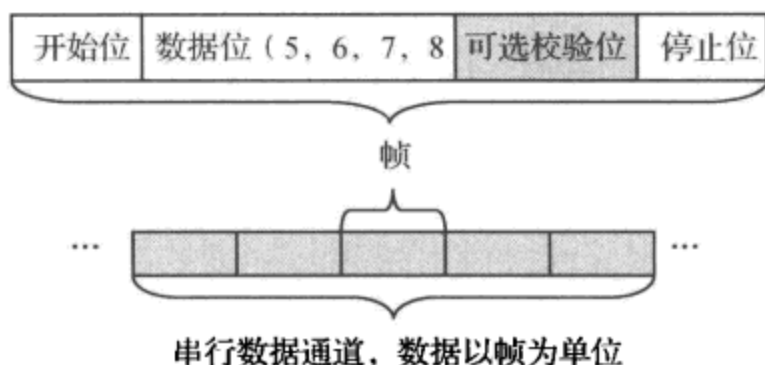


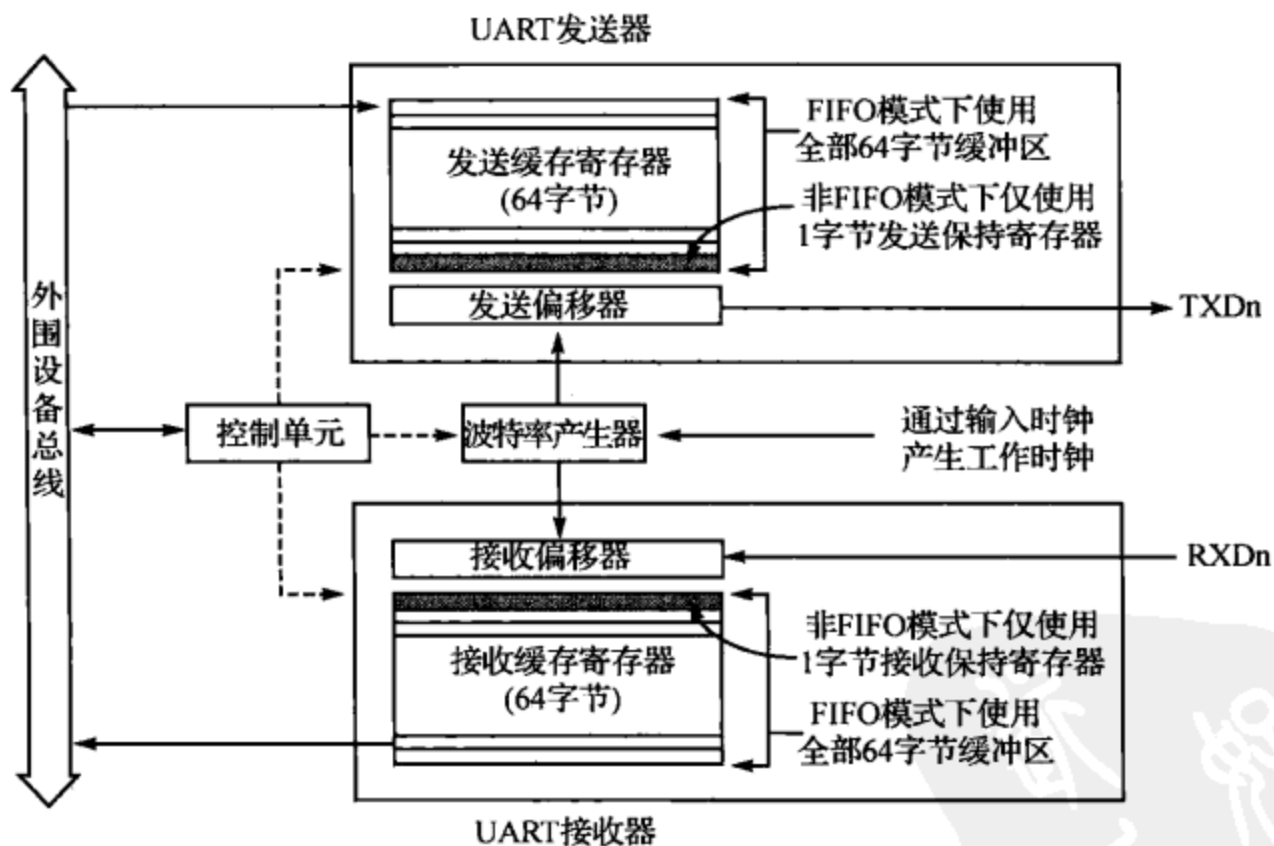
图 2-58 UART 数据帧结构

(4) 停止位: 可选 1、2 位, 该位长度可由编程人员指定。

通信双方约定好帧格式后, 指定同一波特率, 以保证双方数据传输的同步。

2.7.5 S3C2440 UART 串口工作原理

每个 UART 包含一个波特率产生器、发送器、接收器和一个控制单元, 如图 2-59 所示。



当选择 FIFO 模式时, 64 字节寄存器全部用来缓存数据
当选择非 FIFO 模式时, 仅使用 1 字节的数据保持寄存器
来发送或接收数据

图 2-59 UART 硬件结构

UART 是以异步方式实现通信的, 其采样速度由波特率决定, 波特率产生器的工作频率可以由 PCLK(外围设备频率), FCLK/n(CPU 工作频率的分频), UEXTCLK(外部输入时钟)

三个时钟作为输入频率,波特率设置寄存器是可编程的,用户可以设置其波特率决定发送和接收的频率。发送器和接收器包含了 64B 的 FIFO 和数据移位器。UART 通信是面向字节流的,待发送数据写到 FIFO 之后,被复制到数据移位器(1 字节大小)里,数据通过发送数据引脚 TXDn 发出。同样道理,接收数据通过 RXDn 引脚来接收数据(1 字节大小)到接收移位器,然后将其复制到 FIFO 接收缓冲区里。

(1) 数据发送。发送的数据帧可编程的,它的一个帧长度是用户指定的,它包括一个起始位,5~8 个数据位,一个可选的奇偶校验位和 1~2 个停止位,数据帧格式可以通过设置 ULCONn 寄存器来设置。发送器也可以产生一个终止信号,它是由一个全部为 0 的数据帧组成。在当前发送数据被完全传输完以后,该模块发送一个终止信号。在终止信号发送后,它可以继续通过 FIFO(FIFO)或发送保持寄存器(NON-FIFO)发送数据。

(2) 数据接收。同样接收端的数据也是可编程的,接收器可以侦测到溢出错误奇偶校验错误,帧错误和终止条件,每个错误都可以设置一个错误标志。

- 溢出错误是指在旧数据被读取到之前,新数据覆盖了旧数据。
- 奇偶校验错误是指接收器侦测到了接收数据校验结果失败,接收数据无效。
- 帧错误是指接收到的数据没有一个有效的停止位,无法判定数据帧结束。
- 终止条件是指 RxDn 接收到保持逻辑 0 状态持续长于一个数据帧的传输时间。

(3) 自动流控 AFC(Auto Flow Control)。UART0 和 UART1 支持有 nRTS 和 nCTS 的自动流控,UART2 不支持流控。在 AFC 情况下,通信双方 nRTS 和 nCTS 引脚分别连接对方的 nCTS 和 nRTS 引脚。通过软件控制数据帧的发送和接收。

在开启 AFC 时,发送端接收发送前要判断 nCTS 信号状态,当接收到 nCTS 激活信号时,发送数据帧。该 nCTS 引脚连接对方 nRTS 引脚。接收端在准备接收数据帧前,其接收器 FIFO 有大于 32 字节的空闲空间,nRTS 引脚会发送激活信号,当其接收 FIFO 小于 32 字节的空闲空间,nRTS 必须置非激活状态,如图 2-60 所示。

(4) 波特率。在 UART 中波特率发生器为发送器和接收器提供工作时钟。波特率发生器的时钟源可以选择 S3C2440A 的内部系统时钟(PCLK,FCLK/n)或 UEXTCLK(外部时钟源),可以通过设置 UCONn 寄存器来设置波特率发生器的输入时钟源。通常我们选择使用 PCLK 作为 UART 工作时钟。

UART 控制器中没有对波特率进行设置的寄存器,而是通过设置一个除数因子,来决定其波特率。其计算公式如下:

$$\text{UART 除数(UBRDIVn)} = (\text{int})(\text{CLK}/(\text{baud rate} * 16)) - 1$$

其中:UBRDIVn 的取值范围应该为 $1 \sim 2^{16} - 1$ 。例如:波特率为 115 200 b/s,PCLK 时钟为其工作频率,采用 50MHz,UBRDIVn 为:

$$\text{UBRDIVn} = (\text{int})(50 \times 10^6 / (115200 \times 16)) - 1 = 26$$

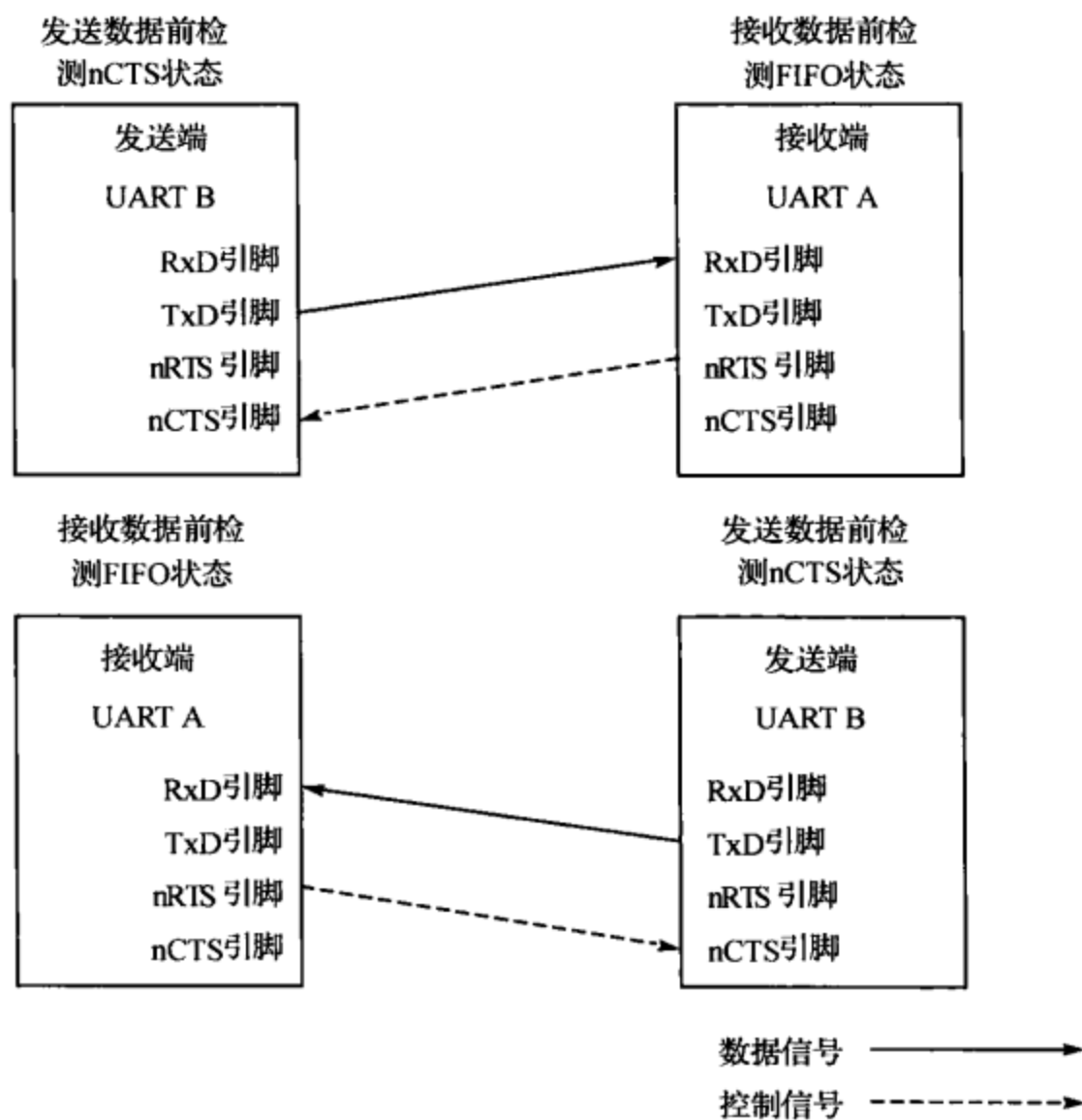


图 2-60 自动流控数据传输

在系统时钟未初始化时, $PCLK = 12\text{MHz}$, 如果比特率采用 $57\,600\text{ b/s}$, 那么 $UBRDIVn$ 为:

$$UBRDIVn = (\text{int})(12 \times 10^6 / (57600 \times 16)) - 1 = 12$$

当使用外部时钟源时, 如果外部时钟小于 $PCLK$ 时钟, 则 $UEXTCLK$ 应该设置为 0。

(5) 波特率的错误容忍率(Baud - Rate Error Torlerance)。数据信号在传输过程中由于外界电磁干扰, 信号减弱等原因, 当时钟频率较低, 传输速率较高时会产生误差, 当误差达到一定值时, 会出现数据信号不能正常识别, 造成通信异常。好比如, 在普通列车轨道上试图行驶高速列车一样, 由于高速列车对轨道要求很高, 当速度达到一定程度, 很可能造成事故。业界的波特率的错误容忍率为 1.86% ($3 / 160$), 如果大于该值则应该选择较低的比特率或提高输入时钟频率。

错误容忍率计算公式为:

$$\text{UART Error} = (\text{tUPCLK} - \text{tUEXACT}) / \text{tUEXACT} * 100\%$$

注: tUPCLK 为 UART 的真实工作时钟频率: $\text{tUPCLK} = (\text{UBRDIV}_n + 1) \times 16 \times 1\text{Frame} / \text{PCLK}$

tUEXACT 为 UART 理想工作时钟频率: $\text{tUEXACT} = 1\text{Frame} / \text{baud-rate}$

其中: 1Frame 为数据帧的长度 = 开始位 + 数据位 + 可选校验位 + 停止位

假如, 波特率采用 115 200 b/s, PCLK 时钟为 50MHz, 波特率除数因子 UBRDIV_n 为 26 (通过前面 UBRDIV_n 计算公式算出), 采用 1 个停止位, 8 个数据位, 无校验的 8N1 方式通信时, 其错误容忍率为:

$$\text{tUPCLK} = 27 \times 16 \times 10 / 50 \times 10^6 = 0.0000864$$

$$\text{tUEXACT} = 10 / 115200 = 0.0000868$$

$$\text{UART Error} = |0.0000864 - 0.0000868| / 0.0000868 = 0.46\%$$

在开发板没有初始化系统时钟前, 开发板工作在 12MHz 下, 假如我们将波特率设置为 115 200 b/s, 采用 PCLK 为系统默认时钟 12MHz, 8N1 数据帧格式通信, 那么:

$$\text{UBRDIV}_n = (\text{int})(12 \times 10^6 / (115200 \times 16)) - 1 = 6$$

其错误容忍率:

$$\text{tUPCLK} = 7 \times 16 \times 10 / 12 \times 10^6 = 0.0000933$$

$$\text{tUEXACT} = 10 / 115200 = 0.0000868$$

$$\text{UART Error} = |0.0000933 - 0.0000868| / 0.0000868 = 7.5\%$$

其错误容忍率大于 1.86%, 因此在 12MHz 频率下, 波特率不能设置为 115 200, 现在将波特率设置为 56 700 b/s, 采用 8N1 数据帧格式通信, 那么:

$$\text{UBRDIV}_n = (\text{int})(12 \times 10^6 / (56700 \times 16)) - 1 = 12$$

$$\text{tUPCLK} = 13 \times 16 \times 10 / 12 \times 10^6 = 0.000173$$

$$\text{tUEXACT} = 10 / 56700 = 0.0001736$$

$$\text{UART Error} = |0.000173 - 0.0001736| / 0.0001736 = 0.345\%$$

采用波特率为 56 700 b/s, 8N1 数据帧格式通信时, 其错误容忍率小于标准的 1.86%, 因此可以正常工作。

UART 的接口

图 2-61 所示为 mini2440 开发板引出 UART 串口接线图, 它采用 DB9 接口公头 (有接线柱的端口, 只有接线孔的为母头), 其有 9 根信号线, UART 通信过程中用到了信号线 2 RSTXD0 (数据发送引脚) 它和串口线母头 TXD_x 信号线相接 (x 代表 0 号, 1 号, 2 号串口), 信号 3 RSRXD0 (数据接收引脚) 和串口线母头 RXD_x 相接 (x 代表 0 号, 1 号, 2 号串口), 信号线 5 (接地引脚), 信号线 7 RSCTS0 (数据发送流控制引脚) 和串口线母头 nCTS_x 相接, 信号线 8 RSRTS0 (数据接收流控制引脚) 和串口线母头 nRTS_x 相接。如果 UART 中没有开启 AFC

第2章 ARM 编程进阶

流控的话,只要用到信号线 2,信号线 3 和信号线 5。

通过图 2-62 所示的 mini2440 硬件 CPU 引脚图可以看出,RSTXD0 和 RSRXD0 连接到 CPU 的 GPH2 和 GPH3 引脚上的,而 GPH2 和 GPH3 是 CPU 复用引脚,因此要对 GPH2 和 GPH3 对应寄存器进行设置,其对应寄存器为 GPHCON 如表 2-23 所列。

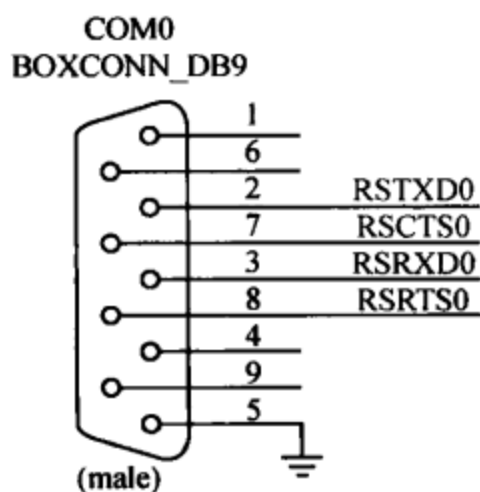


图 2-61 MINI2440 开发板串口硬件图

| | | |
|-------|-----|-----------------|
| nCTS0 | K11 | nCTS0/GPH0 |
| nRTS0 | L17 | nRTS0/GPH1 |
| TXD0 | K13 | TXD0/GPH2 |
| RXD0 | K14 | RXD0/GPH3 |
| TXD1 | K16 | TXD1/GPH4 |
| RXD1 | K17 | RXD1/GPH5 |
| TXD2 | J11 | nRTS1/TXD2/GPH6 |
| RXD2 | J15 | nCTS1/RXD2/GPH7 |
| WP_SD | K15 | UCLK/GPH8 |

图 2-62 mini2440 串口引脚接线

表 2-23 GPIO 端口 H 设置寄存器(GPHCON)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|--------|------------|------|--|-------|
| GPHCON | 0x56000070 | R/W | GPIO 端口 H 配置寄存器 | 0x0 |
| GPHDAT | 0x56000074 | R/W | GPIO 端口 H 数据寄存器 | 未定义 |
| GPHUP | 0x56000078 | R/W | GPIO 端口 H 上拉无效寄存器 | 0x000 |
| GPHCON | 位 | | 描 述 | 初始值 |
| ... | ... | | ... | ... |
| GPH3 | [7:6] | | 设置当前引脚功能: 00 = 输入端口 01 = 输出端口 10 = RXD[0]配置为串口 0 的接收数据引脚 11 = 保留 | 0 |
| GPH2 | [5:4] | | 设置当前引脚功能: 00 = 输入端口 01 = 输出端口 10 = RXD[0]配置为串口 0 的接收数据引脚 11 = 保留 | 0 |
| ... | ... | | ... | ... |

GPHCON[7:6]和 GPHCON[5:4]为 RSTXD0 和 RSRXD0 引脚设置位,将其功能设置为了 UART 专用通信引脚,因此应该设置其为 0b10,分别用于 UART 数据的接收和发送。

GPHCON | = 0xa0;

GPHUP = 0x0;

表 2-24 GPIO 端口 H 上拉电阻设置寄存器(GPHUP)

| GPHUP | 位 | 描 述 | 初始值 |
|-----------|--------|---|-----|
| GPH[10:0] | [10:0] | 设置对应引脚 GPHn 的是否启用上拉功能 0 = 启用上拉功能 1 = 禁用上拉功能 | 0 |

如表 2-24 所列,GPHUP 上拉电阻设置寄存器:上拉电阻用来稳定电平信号,保障传输数据的正确,GPHUP 里设置其内部上拉。

如表 2-25 所列,通过设置 ULCON0 来设置 UART0 通信方式,ULCON0[6]选择通信方式为一般通信模式或红外通信模式,ULCON0[5:3]设置串口 0 校验方式,ULCON0[2]设置串口 0 停止位数,ULCON0[1:0]设置串口 0 的数据位数,如表 2-26 所列。

表 2-25 UART0 串行控制寄存器(ULCON0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|--------|------------|------|---|-------|
| ULCON0 | 0x50000000 | R/W | 串口 0 串行控制寄存器 | 0x00 |
| ULCON0 | 位 | | 描 述 | 初始值 |
| 保留 | [7] | | | 0 |
| 红外模式 | [6] | | 选择串口 0 是否使用红外模式: 0 = 正常通信模式 1 = 红外通信模式 | 0 |
| 校验模式 | [5:3] | | 设置串口 0 在数据接收和发送时采用的校验方式: 0xx = 无校验 100 = 奇校验 101 = 偶校验 110 = 强制校验/检测是否为 1 111 = 强制校验/检测是否为 0 | 000 |
| 停止位 | [2] | | 设置串口 0 停止位数: 0 = 每个数据帧一个停止位 1 = 每个数据帧二个停止位 | 0 |
| 数据位 | [1:0] | | 设置串口 0 数据位数: 00 = 5 个数据位 01 = 6 个数据位 10 = 7 个数据位 11 = 8 个数据位 | 00 |

我们选择一般通信模式,无校验位,1个停止位,8个数据位的数据通信方式。因此:

ULCON0 = 0x03;

表 2-26 UART0 串口控制寄存器(UCON0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|----------------|------------|------|---|-------|
| UCON0 | 0x50000004 | R/W | 串口 0 控制寄存器 | 0x00 |
| UCON0 | 位 | | 描 述 | 初始值 |
| FCLK 分频因子 | [15:12] | | 当 UART0 选择 FCLK 作为时钟源时,设置其 FCLK 的分频因子 UART0 工作时钟频率 = FCLK/ FCLK 分频因子 + 6 | 0000 |
| UART 时钟源 选择 | [11:10] | | 选择 UART0 的工作时钟 PCLK, UEXTCLK, FCLK/n: 00, 10 = PCLK 01 = UEXTCLK 11 = FCLK/n 当选择 FCLK/n 作为 UART0 工作时钟时还要做其他设置, 具体请读者自行查看硬件手册 | 00 |
| 发送数据中断 产生类型 | [9] | | 设置 UART0 中断请求类型,在非 FIFO 传输模式下,一旦 发送数据缓冲区为空,立即产生中断信号,在 FIFO 传输模 式下达到发送数据触发条件时立即产生中断信号: 0 = 脉冲触发 1 = 电平触发 | 0 |
| 接收数据中断 产生类型 | [8] | | 设置 UART0 中断请求类型,在非 FIFO 传输模式下,一旦 接收到数据,立即产生中断信号,在 FIFO 传输模式下达到 接收数据触发条件时立即产生中断信号: 0 = 脉冲触发 1 = 电平触发 | 0 |
| 接收数据超时 | [7] | | 设置当接收数据时,如果数据超时,是否产生接收中断: 0 = 不开启超时中断 1 = 开启超时中断 10 = 7 个数据位 11 = 8 个数据位 | 0 |
| 接收数据 错误中断 | [6] | | 设置当接收数据时,如果产生异常,如传输中止,帧错误, 校验错误时,是否产生接收状态中断信号: 0 = 不产生错误状态中断 1 = 产生错误状态中断 | 0 |
| 回送模式 | [5] | | 设置该位时 UART 会进入回送模式,该模式仅用于测试 0 = 正常模式 1 = 回送模式 | 0 |

续表 2-26

| UCON0 | 位 | 描 述 | 初始值 |
|--------|-------|--|-----|
| 发送终止信号 | [4] | 设置该位时, UART 会发送一个帧长度的终止信号, 发送完毕后, 该位自动恢复为 0 0 = 正常传输 1 = 发送终止信号 | 0 |
| 发送模式 | [3:2] | 设置采用哪个方式执行数据写入发送缓冲区 00 = 无效 01 = 中断请求或查询模式 10 = DMA0 请求 | 00 |
| 接收模式 | [1:0] | 设置采用哪个方式执行数据写入接收缓冲区 00 = 无效 01 = 中断请求或查询模式 10 = DMA0 请求 | 00 |

如表 2-27~表 2-32 所列, 通常 UART 串口采用 PCLK 作为输入工作时钟, 采用简单的轮询方式进行数据接收和发送, 不开启数据接收超时, 数据产生错误时不产生错误状态中断, 因此:

UCON0 = 0x05;

表 2-27 UART FIFO 控制寄存器(UFCON0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|--------------|------------|------|--|-------|
| UFCON0 | 0x50000008 | R/W | 串口 0 FIFO 控制寄存器 | 0x00 |
| UFCON0 | 位 | | 描 述 | 初始值 |
| 发送数据 触发级别 | [7:6] | | 设置 FIFO 发送模式的触发级别: 00 = FIFO 为空触发 01 = 16 字节触发 10 = 32 字节触发 11 = 48 字节触发 | 00 |
| 接收数据 触发级别 | [5:4] | | 设置 FIFO 接收模式的触发级别: 00 = FIFO 为空触发 01 = 16 字节触发 10 = 32 字节触发 11 = 48 字节触发 | 00 |
| 保留 | [3] | | | 0 |
| 发送 FIFO 重置 | [2] | | 在重置 FIFO 后自动清除发送缓冲区 0 = 正常模式 1 = 自动清除 | 0 |
| 接收 FIFO 重置 | [1] | | 在重置 FIFO 后自动清除接收缓冲区 0 = 正常模式 1 = 自动清除 | 0 |
| 启用 FIFO | [0] | | 0 = 不启用 FIFO 1 = 启用 FIFO | 0 |

表 2-28 UART MODEM 控制寄存器(UMCON0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|----------|------------|---|------------------|-------|
| UMCON0 | 0x5000000C | R/W | 串口 0 MODEM 控制寄存器 | 0x00 |
| UMCON0 | 位 | 描 述 | | 初始值 |
| 保留 | [7:5] | 必须全部置 0 | | 000 |
| AFC 自动流控 | [4] | 0 = 不开启流控 1 = 开启流控 | | 0 |
| 保留 | [3:1] | 必须全部置 0 | | 000 |
| 请求发送 | [0] | 如果启用 AFC, 该位无效, S3C2440 会自动控制 nRTS, 如果不启用 AFC, nRTS 必须由软件控制 0 = 高电平激活 nRTS 1 = 低电平激活 nRTS | | 0 |

表 2-29 UART 发送/接收状态寄存器(UTRSTAT0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|---|-----------------|-------|
| UTRSTAT0 | 0x50000010 | R/W | 串口 0 发送/接收状态寄存器 | 0x06 |
| UTRSTAT0 | 位 | 描 述 | | 初始值 |
| 发送器为空 | [2] | 当发送缓存寄存器中没有数据要发送且发送移位寄存器为空时, 自动置 1 0 = 非空 1 = 发送器为空(发送缓存和移位寄存器) | | 1 |
| 发送缓存寄存器为空 | [1] | 当发送缓存寄存器为空时, 自动置 1 0 = 发送缓存寄存器非空 1 = 发送缓存寄存器为空 | | 1 |
| 接收缓存寄存器为空 | [0] | 当接收缓存寄存器有数据到达时, 自动置 1 0 = 接收缓存寄存器为空 1 = 缓存寄存器接收数据 | | 0 |

表 2-30 UART 发送缓存寄存器(UTXH0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-------|--------------------------------|------|--------------|-------|
| UTXH0 | 0x50000020(L) 0x50000023(B) | W | 串口 0 发送缓存寄存器 | — |

表 2-31 UART 接收缓存寄存器(URXH0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-------|--------------------------------|------|--------------|-------|
| URXH0 | 0x50000024(L) 0x50000027(B) | R | 串口 0 接收缓存寄存器 | — |

表 2-32 UART 比特率除数寄存器(UBRDIV0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|-----------------------------|---------------|-------|
| UBRDIV0 | 0x50000028 | R/W | 串口 0 波特率除数寄存器 | — |
| UBRDIV0 | 位 | | 描 述 | 初始值 |
| 比特率除数 | [15:0] | 设置比特率除数(大于 0)使用外部输入时钟时可以置 0 | | — |

上述寄存器是和 UART 通信相关寄存器,使用简单的无 FIFO,无自动流控 AFC 时,设置如下:

```

UFCON0 = 0x00;      // 不使用 FIFO
UMCON0 = 0x00;      // 不使用流控
UBRDIV0 = 26;        // 波特率为 115200, PCLK = 50MHz
UBRDIV0 = 53;        // 波特率为 57600, PCLK = 50MHz
UBRDIV0 = 12;        // 波特率为 57600, PCLK = 12MHz

```

UTXH0 和 URXH0 分别是数据发送和接收寄存器,发送数据时通过轮询方式判断发送状态寄存器的状态,当可以发送数据时,执行 UTXH0 寄存器写入操作,接收数据时,以轮询方式检测接收状态寄存器状态,当有数据到达时,读取 URXH0 寄存器里的数据即可取得串口数据。

```

#define TXDREADY    (1<<2) //发送数据状态 OK
#define RXDREADY    (1)     //接收数据状态 OK

```

```

/* UART 串口单个字符打印函数 */
extern void putc(unsigned char c)
{
    while( !(UTRSTAT0 & TXDREADY) );
    UTXH0 = c;
}

```

```

/* UART 串口接受单个字符函数 */

```




```
extern unsigned char getc(void)
{
    while( !(UTRSTAT0 & RXD0READY) );
    return URXH0;
}
```

2.7.6 UART 串口驱动实验

本实验源码存放在光盘目录\work\armarch\uart_init 工程目录下。

init.s:本程序文件对看门狗,内存等基本硬件做初始化,然后跳入到 xmain.c 中的 xmain 函数执行。

```
;
; UART 串口实验
;

GPBCON      EQU      0x56000010
GPBDAT      EQU      0x56000014

AREA Init, CODE, READONLY
ENTRY

start
    ; close watchdog
    ldr r0, = 0x53000000      ; 将看门狗控制寄存器地址放入 r0
    mov r1, #0
    str r1, [r0]             ; 设置看门狗控制寄存器的值为 0

    bl initmem               ; 跳转到 initmem 代码段,初始化内存

IMPORT xmain                 ; 引入 main.c 中的 xmain 函数
ldr sp, = 0x34000000        ; 调用 C 程序之前先初始化栈指针
ldr lr, = loop              ; 设置 xmain 函数的返回地址
ldr pc, = xmain              ; 跳转到 C 程序中的 xmain 函数的入口处执行

loop
    b loop                  ; 死循环

initmem
    ; 内存初始化
    ldr r0, = 0x48000000    ; 加载内存相关寄存器首地址 r0
    ldr r1, = 0x48000034    ; 加载内存相关寄存器尾地址到 r1
    adr r2, memdata         ; 将寄存器配置数据地址段首地址加载到 r2
```

```

initmemloop
    ldr r3, [r2], #4           ; 循环设置寄存器
    str r3, [r0], #4
    teq r0, r1
    bne initmemloop           ; 循环到最后一个寄存器时退出函数
    mov pc, lr

memdata
    DCD    0x22000000           ; BWSCON
    DCD    0x00000700           ; BANKCON0
    DCD    0x00000700           ; BANKCON1
    DCD    0x00000700           ; BANKCON2
    DCD    0x00000700           ; BANKCON3
    DCD    0x00000700           ; BANKCON4
    DCD    0x00000700           ; BANKCON5
    DCD    0x00018005           ; BANKCON6
    DCD    0x00018005           ; BANKCON7
    DCD    0x008e07a3           ; REFRESH
    DCD    0x000000b1           ; BANKSIZE
    DCD    0x00000030           ; MRSRB6
    DCD    0x00000030           ; MRSRB7

    END

```

xmain.c:uart_init 函数对 UART0 进行初始化,然后进入死循环内,不停打印字符串“Uart 串口打印试验”。

```

/* xmain.c */

/* GPIO registers */
#define GPHCON    (*(volatile unsigned long *)0x56000070)
#define GPHDAT    (*(volatile unsigned long *)0x56000074)
#define GPHUP     (*(volatile unsigned long *)0x56000078)

/* UART registers */
#define ULCON0     (*(volatile unsigned long *)0x50000000)
#define UCON0      (*(volatile unsigned long *)0x50000004)
#define UFCON0     (*(volatile unsigned long *)0x50000008)
#define UMCON0     (*(volatile unsigned long *)0x5000000c)
#define UTRSTAT0   (*(volatile unsigned long *)0x50000010)

```

```

#define    UTXH0                (*(volatile unsigned char *)0x50000020)
#define    URXH0                (*(volatile unsigned char *)0x50000024)
#define    UBRDIV0              (*(volatile unsigned long *)0x50000028)

#define    TXDREADY             (1<<2)           //发送数据状态 OK
#define    RXDREADY             (1)               //接收数据状态 OK

/* UART 串口初始化 */
void uart_init( )
{
    GPHCON |= 0xa0;           //GPH2,GPH3 used as TXD0,RXD0
    GPHUP   = 0x0;           //GPH2,GPH3 内部上拉
    ULCON0  = 0x03;         //8N1
    UCON0   = 0x05;         //查询方式为轮询或中断;时钟选择为 PCLK
    UFCON0  = 0x00;         //不使用 FIFO
    UMCN0   = 0x00;         //不使用流控
    UBRDIV0 = 12;           //比特率为 57600,PCLK = 12MHz
}

/* UART 串口单个字符打印函数 */
extern void putc(unsigned char c)
{
    while( !(UTRSTAT0 & TXDREADY) );
    UTXH0 = c;
}

/* UART 串口接受单个字符函数 */
extern unsigned char getc(void)
{
    while( !(UTRSTAT0 & RXDREADY) );
    return URXH0;
}

/* UART 串口字符串打印函数 */
extern int printk(const char * str)
{
    int i = 0;
    while( str[i] ){
        putc( (unsigned char) str[i + +] );
    }
}

```

```
    }  
    return i;  
}  
  
__inline void delay(int msec)  
{  
    int i, j;  
    for(i = 1000; i > 0; i--)  
        for(j = msec * 10; j > 0; j--)  
            /* do nothing */;  
}  
  
/* xmain 通过 UART 串口打印字符串 */  
int xmain()  
{  
    uart_init();  
    while(1) {  
        delay(10);  
        printk("Uart 串口打印试验\r\n");  
    }  
    return 0;  
}
```

当编译并将生成 image 文件烧写到 Norflash, 在 PC 上打开: 开始—>所有程序—>附件—>通讯—>超级终端, 创建一个新的连接: mini2440, 在之后弹出的 COM 设置中, 设置 COM 波特率 57 600 b/s, 1 个停止位, 8 个数据位, 无校验方式, 通过串口线连接开发板和 PC 串口(笔记本电脑通常没有串口, 可以买一个 USB 转串口线), 打开电源可以看到超级终端会不停输出“Uart 串口打印试验”字符串。

资源分享
PDG

第 3 章

ARM 体系结构

3.1 ARM 处理器工作模式

ARM 处理器共有 7 种工作模式,如表 3-1 所列。

表 3-1 ARM 处理器工作模式

| 处理器工作模式 | 特权模式 | 异常模式 | 说 明 |
|----------------------|-------------------------|--------------------------|--------------------------|
| 用户(user)模式 | | | 用户程序运行模式 |
| 系统(system)模式 | | | 运行特权级的操作系统任务 |
| 一般中断(IRQ)模式 | 该组模式下 可以任意访问 系统资源 | 通常由系统异常 状态切换进 该组模式 | 普通中断模式 |
| 快速中断(FIQ)模式 | | | 快速中断模式 |
| 管理(supervisor)模式 | | | 提供操作系统使用的一种保护模式,swi 命令状态 |
| 中止(abort)模式 | | | 虚拟内存管理和内存数据访问保护 |
| 未定义指令终止(undefined)模式 | | | 支持通过软件仿真硬件的协处理 |

CPU 的模式可以简单地理解为当前 CPU 的工作状态,比如:当前操作系统正在执行用户程序,那么当前 CPU 工作在用户模式,这时网卡上有数据到达,产生中断信号,CPU 自动切换到一般中断模式下处理网卡数据(普通应用程序没有权限直接访问硬件),处理完网卡数据,返回到用户模式下继续执行用户程序。

除用户模式外,其他模式均为特权模式(Privileged Modes)。ARM 内部寄存器和一些片内外设在硬件设计上只允许(或者可选为只允许)特权模式下访问。此外,特权模式可以自由的切换处理器模式,而用户模式不能直接切换到别的模式。

特权模式中除系统(system)模式之外的其他 5 种模式又统称为异常模式。它们除了可以通过在特权下的程序切换进入外,也可以由特定的异常进入。比如硬件产生中断信号进入中断异常模式,读取没有权限数据时进入中止异常模式,执行未定义指令时进入未定义指令中止异常模式。其中管理模式也称为超级用户模式,是为操作系统提供软件中断的特有模式,正是

[illegible]

续表 3-2

| 寄存器类别 | 寄存器在汇编中的名称 | 各模式下实际访问的寄存器 | | | | | | |
|-------------|------------|--------------|----|----------|----------|----------|----------|----------|
| | | 用户 | 系统 | 管理 | 终止 | 未定义 | 中断 | 快中断 |
| 通用寄存器和程序计数器 | R8(v5) | R8 | | | | | | |
| | R9(SB,v6) | R9 | | | | | | |
| | R10(SL,v7) | R10 | | | | | | |
| | R11(FP,v8) | R11 | | | | | | |
| | R12(IP) | R12 | | | | | | |
| | R13(SP) | R13 | | R13_svc | R13_abt | R13_und | R13_irq | R13_fiq |
| | R14(LR) | R14 | | R14_svc | R14_abt | R14_und | R14_irq | R14_fiq |
| | R15(PC) | R15 | | | | | | |
| 状态寄存器 | CPSR | CPSR | | | | | | |
| | SPSR | 无 | | SPSR_abt | SPSR_abt | SPSR_und | SPSR_irq | SPSR_fiq |

其中 R0~R7 是在所有模式下都可以使用的共有寄存器, R8~R12 是快速中断模式下私有的寄存器, 其他模式下不能使用, 之所以称其快速中断, 是因为快速中断模式下, 这几个私有寄存器里数据在模式切换时可以不用入栈保存。

除了用户模式和系统模式共用一组 R13、R14, 其余每种模式都私有自己的 R13、R14, 因为在每种模式下都有自己的栈空间用于执行程序, 在执行程序过程中还要保存返回地址, 这样可以保证在进入不同模式时, 当前模式下栈空间不被破坏。比如: 网卡因为数据到达, 产生了中断进入中断模式, 在中断模式下有自己的中断处理例程(ISR), ISR 在执行时要用到栈空间, 因此要使用 R13、R14。中断处理完成后, 返回用户模式下, 要继续执行被网卡中断信号中断的执行程序。

用户模式和系统模式为什么要共用一组 R13、R14 呢? 这是因为, 在特权模式下可以自由切换工作模式, 但是如果切换到用户模式下, 就不能再切换到特权模式了, 这是 CPU 为操作系统提供的保护机制, 但是有的时候就需要切换到用户模式下去使用其 R13、R14 寄存器, 比如当操作系统的进程进行上下文切换时, 如果用户模式和系统模式共用一组寄存器, 那么可以切换到系统模式下(系统模式是特权模式)进行操作。

所有 7 种模式都共有 R15 和 CPSR, 单核 CPU 同时只能处理一条指令, 在取指时, 有一个 PC 就可以了, 同样用一个 CPSR 表示当前 CPU 的状态即可。

3.1.2 ARM 处理器模式切换(含 MRS、MSR 指令)

除了用户模式和系统模式, 其余模式下都有一个私有 SPSR 保存状态寄存器, 用来保存切换到该模式之前的执行状态, 之所以用户模式和系统模式没有 SPSR, 是因为通常 CPU 大部

分时间执行在用户模式下,当产生异常或系统调用时会分别切换进入另外几种模式,保存用户模式下的状态,当切换回原先模式时,直接回复 SPSR 的值到 CPSR 就可以了,因此,用户模式和系统模式下不需要 SPSR,其详细操作查看下节异常处理。以上几种模式通过 CPSR 里的 M[4:0]位进行区分,如图 3-1 所示。

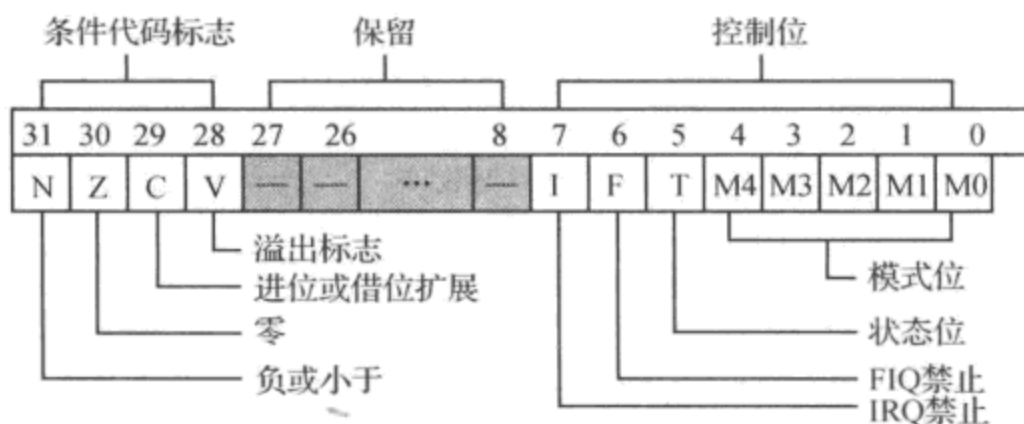


图 3-1 CPSR 控制位

通过向模式位 M[4:0]里写入相应的数据切换到不同的模式,在对 CPSR、SPSR 寄存器进行操作不能使用 mov、ldr 等通用指令,只能使用特权指令 MSR 和 MRS。

在 ARM 处理器中,只有 MRS(Move to Register from State register)指令可以对状态寄存器 CPSR 和 SPSR 进行读操作。通过读 CPSR 可以获得当前处理器的工作状态。读 SPSR 寄存器可以获得进入异常前的处理器状态(因为只有异常模式下有 SPSR 寄存器)。

例如:

```
MRS    R1,CPSR    ; 将 CPSR 状态寄存器读取,保存到 R1 中
MRS    R2,SPSR    ; 将 SPSR 状态寄存器读取,保存到 R2 中
```

通过 MRS 指令可以取得状态寄存器里的值,然后比较其模式位 M[4:0]的值判断当前所处模式,当然也可以比较其他相应位了解当前 CPU 的状态。

同样,在 ARM 处理器中,只有 MSR 指令可以对状态寄存器 CPSR 和 SPSR 进行写操作。与 MRS 配合使用,可以实现对 CPSR 或 SPSR 寄存器的读-修改-写操作,可以切换处理器模式、或者允许/禁止 IRQ/FIQ 中断等。

由于 xPSR 寄存器代表了 CPU 的状态,其每个位有特殊意义,在执行对 xPSR 状态寄存器写入时(读取时不存在该用法),为了防止误操作和方便记忆,将 xPSR 里 32 位分成 4 个区域,每个区域用小写字母表示:

- c 控制域屏蔽 PSR[7..0]
 - x 扩展域屏蔽 PSR[15..8]
 - s 状态域屏蔽 PSR[23..16]
 - f 标志域屏蔽 PSR[31..24]
- 注意:区域名必须为小写字母

第3章 ARM 体系结构

向对应区域进行执行写入时,使用 xPSR_x 可以指定写入区域,而不影响状态寄存器其他位,如:

使能 IRQ 中断:

ENABLE_IRQ

```
MRS    R0, CPSR           ; 将 CPSR 寄存器内容读出到 R0
BIC    R0, R0, # 0x80     ; 清掉 CPSR 中的 I 控制位
MSR    CPSR_c, R0         ; 将修改后的值写回 CPSR 寄存器的对应控制域
MOV    PC, LR             ; 返回上一层函数
```

禁用 IRQ 中断:

DISABLE_IRQ

```
MRS    R0, CPSR           ; 将 CPSR 寄存器内容读出到 R0
ORR    R0, R0, # 0x80     ; 设置 CPSR 中的 I 控制位
MSR    CPSR_c, R0         ; 将修改后的值写回 CPSR 寄存器的对应控制域
MOV    PC, LR             ; 返回上一层函数
```

表 3-3 列出了不同模式的二进制数表示。

表 3-3 不同工作模式对应二进制

| 模式名 | 用户 | 快中断 | 中断 | 管理 | 中止 | 未定义 | 系统 |
|--------|-------|-------|-------|-------|-------|-------|-------|
| M[4:0] | 10000 | 10001 | 10010 | 10011 | 10111 | 11011 | 11111 |

在对开发板进行初始化时,用对不同模式指定其栈空间,下面例子对各模式的栈指针 sp 进行初始化:

```
stack_init           ; 栈指针初始化函数
@ undefine_stack
msr cpsr_c, # 0xdb    ; 切换到未定义异常
ldr    sp, = 0x34000000 ; 栈指针为内存最高地址,栈为倒生的栈
                        ; 栈空间的最后 1M 0x34000000~0x33f00000

@ abort_stack
msr cpsr_c, # 0xd7    ; 切换到终止异常模式
ldr    sp, = 0x33f00000 ; 栈空间为 1MB, 0x33f00000~0x33e00000

@ irq_stack
msr    cpsr_c,      # 0xd2 ; 切换到中断模式
ldr    sp, = 0x33e00000 ; 栈空间为 1MB, 0x33e00000~0x33d00000

@ sys_stack
msr    cpsr_c,      # 0xdf ; 切换到系统模式
ldr    sp, = 0x33d00000 ; 栈空间为 1MB, 0x33d00000~0x33c00000
```



```
msr    cpsr_c,    # 0xd3    ; 切换回管理模式  
mov    pc, lr
```

3.2 ARM 处理器异常处理

所谓异常就是正常执行的用户程序被异常情况中止,处理器就进入异常模式的过程,例如响应一个来自外设的中断,或者当前程序非法访问内存地址都会进入相应异常模式。

3.2.1 异常分类

(1) 复位异常。当 CPU 刚上电时或按下 reset 重启键之后进入该异常,该异常在管理模式处理。

(2) 一般/快速中断请求。CPU 和外围设备是分别独立的硬件执行单元,CPU 对全部设备进行管理和资源调度处理,CPU 要想知道外围设备的运行状态,要么 CPU 定时的去查看外围设备特定寄存器,要么让外围设备在出现需要 CPU 干涉处理时“打断”CPU,让它来处理外围设备的请求,毫无疑问第二种方式更合理,可以让 CPU“专心”去工作。这里的“打断”操作就称做中断请求,根据请求的紧急情况,中断请求分一般中断和快速中断,快速中断具有最高中断优先级和最小的中断延迟,通常用于处理高速数据传输及通道的中数据恢复处理,如 DMA 等,绝大部分外设使用一般中断请求。

(3) 预取指令中止异常。该异常发生在 CPU 流水线取指阶段,如果目标指令地址是非法地址进入该异常,该异常在中止异常模式下处理。

(4) 未定义指令异常。该异常发生在流水线技术里的译码阶段,如果当前指令不能被识别为有效指令,产生未定义指令异常,该异常在未定义异常模式下处理。

(5) 软件中断指令(swi)异常。该异常是应用程序自己调用时产生的,用于用户程序申请访问硬件资源时,例如:printf()打印函数,要将用户数据打印到显示器上,用户程序要想实现打印必须申请使用显示器,而用户程序又没有外设硬件的使用权,只能通过使用软件中断指令切换到内核态,通过操作系统内核代码来访问外设硬件,内核态是工作在特权模式下,操作系统在特权模式下完成将用户数据打印到显示器上。这样做的目的无非是为了保护操作系统的安全和硬件资源的合理使用,该异常在管理模式处理。

(6) 数据中止访问异常。该异常发生在要访问数据地址不存在或者为非法地址时,该异常在中止异常模式下处理。

3.2.2 异常发生时的硬件操作

在异常发生后,ARM 内核会自动做以下工作:

- 保存执行状态:将 CPSR 复制到发生的异常模式下 SPSR 中。
- 模式切换:将 CPSR 模式位强制设置为与异常类型相对应的值,同时处理器进入到 ARM 执行模式,禁止所有 IRQ 中断,当进入 FIQ 快速中断模式时禁止 FIQ 中断。
- 保存返回地址:将下一条指令的地址(被打断程序)保存在 LR(异常模式下 LR_excep)中。
- 跳入异常向量表:强制设置 PC 的值为相应异常向量地址,跳转到异常处理程序中。

(1)保存执行状态。当前程序的执行状态是保存在 CPSR 里面的,异常发生时,要保存当前的 CPSR 里的执行状态到异常模式里的 SPSR 里,将来异常返回时,恢复回 CPSR,恢复执行状态。

(2)模式切换。硬件自动根据当前的异常类型,将异常码写入 CPSR 里的 M[4:0]模式位,这样 CPU 就进入了对应异常模式下。不管是在 ARM 状态下还是在 THUMB 状态下发生异常,都会自动切换到 ARM 状态下进行异常的处理,这是由硬件自动完成的,将 CPSR[5] 设置为 0。同时,CPU 会关闭中断 IRQ(设置 CPSR 寄存器 I 位),防止中断进入,如果当前是快速中断 FIQ 异常,关闭快速中断(设置 CPSR 寄存器 F 位)。

(3)保存返回地址。当前程序被异常打断,切换到异常处理程序里,异常处理完之后,返回当前被打断模式继续执行,因此必须要保存当前执行指令的下一条指令的地址到 LR_excep(异常模式下 LR,并不存在 LR_excep 寄存器,为方便读者理解加上_excep,以下道理相同),由于异常模式不同以及 ARM 内核采用流水线技术,异常处理程序里要根据异常模式计算返回地址。

(4)跳入异常向量表。该操作是 CPU 硬件自动完成的,当异常发生时,CPU 强制将 PC 的值修改为一个固定内存地址,这块固定地址称做异常向量(详见 3.2.4 小节)。

3.2.3 异常返回地址

由 1.1.3 小节可知,一条指令的执行分为取指、译码、执行三个主要阶段,CPU 由于使用流水线技术,造成当前执行指令的地址应该是 $PC - 8$ (32 位机一条指令 4 字节),那么执行指令的下条指令应该是 $PC - 4$ 。在异常发生时,CPU 自动会将 $PC - 4$ 的值保存到 LR 里,但是该值是否正确还要看异常类型才能决定。

各模式的返回地址说明如下:

(1)一般/快速中断请求。快速中断请求和一般中断请求返回处理是一样的。通常处理器执行完当前指令后,查询 FIQ/IRQ 中断引脚,并查看是否允许 FIQ/IRQ 中断,如果某个中断引脚有效,并且系统允许该中断产生,处理器将产生 FIQ/IRQ 异常中断,当 FIQ/IRQ 异常中断产生时,程序计数器 PC 的值已经更新,它指向当前指令后面第 3 条指令(对于 ARM 指

令,它指向当前指令地址加 12 字节的位置;对于 Thumb 指令,它指向当前指令地址加 6 字节的位置),当 FIQ/IRQ 异常中断产生时,处理器将值(PC-4)保存到 FIQ/IRQ 异常模式下的寄存器 lr_irq/lr_irq 中,它指向当前指令之后的第 2 条指令,因此正确返回地址可以通过下面指令算出:

```
SUBS    PC,LR_irq,#4           ; 一般中断
```

```
SUBS    PC,LR_fiq,#4          ; 快速中断
```

注:LR_irq/LR_fiq 分别为一般中断和快速中断异常模式下 LR,并不存在 LR_xxx 寄存器,为方便读者理解加上_xxx

(2) 预取中止异常。在指令预取时,如果目标地址是非法的,该指令被标记成有问题的指令,这时,流水线上该指令之前的指令继续执行,当执行到该被标记成有问题的指令时,处理器产生指令预取中止异常中断。发生指令预取异常中断时,程序要返回到该有问题的指令处,重新读取并执行该指令,因此指令预取中止异常中断应该返回到产生该指令预取中止异常中断的指令处,而不是当前指令的下一条指令。

指令预取中止异常中断由当前执行的指令在 ALU 里执行时产生,当指令预取中止异常中断发生时,程序计数器 PC 的值还未更新,它指向当前指令后面第二条指令(对于 ARM 指令,它指向当前指令地址加 8 字节的位置;对于 Thumb 指令,它指向当前指令地址加 4 字节的位置)。此时处理器将值(PC-4)保存到 lr_abt 中,它指向当前指令的下一条指令,所以返回操作可以通过下面指令实现:

```
SUBS    PC,LR_abt,#4
```

注:LR_abt 为中止模式下 LR,并不存在 LR_abt 寄存器,为方便读者理解加上_abt

(3) 未定义指令异常。未定义指令异常中断由当前执行的指令在 ALU 里执行时产生,当未定义指令异常中断产生时,程序计数器 PC 的值还未更新,它指向当前指令后面第二条指令(对于 ARM 指令,它指向当前指令地址加 8 字节的位置;对于 Thumb 指令,它指向当前指令地址加 4 字节的位置),当未定义指令异常中断发生时,处理器将值(PC-4)保存到 lr_und 中,此时(PC-4)指向当前指令的下一条指令,所以从未定义指令异常中断返回可以通过如下指令来实现:

```
MOV     PC, LR_und
```

注:LR_und 为未定义模式下 LR,并不存在 LR_und 寄存器,为方便读者理解加上_und

(4) 软中断指令(SWI)异常。SWI 异常中断和未定义异常中断指令一样,也是由当前执行的指令在 ALU 里执行时产生,当 SWI 指令执行时,PC 的值还未更新,它指向当前指令后面第二条指令(对于 ARM 指令,它指向当前指令地址加 8 字节的位置;对于 Thumb 指令,它指向当前指令地址加 4 字节的位置),当未定义指令异常中断发生时,处理器将值(PC-4)保存到 lr_svc 中,此时(PC-4)指向当前指令的下一条指令,所以从 SWI 异常中断处理返回的实现

第3章 ARM 体系结构

方法与从未定义指令异常中断处理返回一样：

MOV PC, LR svc

注:LR svc 为管理模式下 LR,并不存在 LR svc 寄存器,为方便读者理解加上 svc

(5) 数据中止异常。发生数据访问异常中断时,程序要返回到该有问题的指令处,重新访问该数据,因此数据访问异常中断应该返回到产生该数据访问中止异常中断的指令处,而不是当前指令的下一条指令。

数据访问异常中断由当前执行的指令在 ALU 里执行时产生,当数据访问异常中断发生时,程序计数器 PC 的值已经更新,它指向当前指令后面第 3 条指令(对于 ARM 指令,它指向当前指令地址加 12 字节的位置;对于 Thumb 指令,它指向当前指令地址加 6 字节的位置)。此时处理器将值(PC-4)保存到 lr_abt 中,它指向当前指令后面第 2 条指令,所以返回操作可以通过下面指令实现:

SUBS PC, LR abt, # 8

注:LR abt 为中止模式下 LR,并不存在 LR abt 寄存器,为方便读者理解加上 abt

上述每一种异常发生时,其返回地址都要根据具体异常类型进行重新修复返回地址,再次强调下,被打断程序的返回地址保存在对应异常模式下的 LR_{excep} 里。

3.2.4 异常向量表

异常向量表是一段特定内存地址空间,每种 ARM 异常对应一个字长空间(4B),正好是一条 32 位指令长度,当异常发生时,CPU 强制将 PC 的值设置为当前异常对应的固定内存地址。表 3-4 所列是 S3C2440 的异常向量表。

表 3-4 异常向量表

| 地 址 | Exception | Mode in Entry |
|------------|-----------------------|---------------|
| 0x00000000 | Reset | Supervisor |
| 0x00000004 | Undefined instruction | Undefined |
| 0x00000008 | Software Interrupt | Supervisor |
| 0x0000000C | Abort(prefetch) | Abort |
| 0x00000010 | Abort(data) | Abort |
| 0x00000014 | Reserved | Reserved |
| 0x00000018 | IRQ | IRQ |
| 0x0000001C | FIQ | FIQ |

注:异常向量也可以出现在高地址 0xFFFF0000 处,当今操作系统为了控制内存访问权限,通常会开启虚拟内存。开启了虚拟内存之后,内存的开始空间通常为内核进程空间和页表空间,异常向量表不能再安装在 0 地址处了。

ARM 的例外优先级从高到低依次为 Reset→Data abort→FIQ→IRQ→Prefetch abort→Undefined instruction/SWI。

跳入异常向量表操作是异常发生时,硬件自动完成的,剩下的异常处理任务完全交给了程序员。由上表可知,异常向量是一个固定的内存地址,我们可以通过向该地址处写一条跳转指令,让它跳向我们自己定义的异常处理程序的入口,就可以完成异常处理了。

正是由于异常向量表的存在,才让硬件异常处理和程序员自定义处理程序有机联系起来。异常向量表里 0x00000000 地址处是 Reset 复位异常,之所以它为 0 地址,是因为 CPU 在上电时自动从 0 地址处加载指令。由此可见将复位异常安装在此地址处也是前后结合起来设计的,不得不感叹 CPU 设计师的伟大;其后面分别是其余 7 种异常向量,每种异常向量都占有 4 字节,正好是一条指令的大小;最后一个异常是快速中断异常,将其安装在此也有它的意义,在 0x0000001C 地址处可以直接存放快速中断的处理程序,不用设置跳转指令,这样可以节省一个时钟周期,加快速度中断处理时间。

我们可以通过简单地使用下面的指令来安装异常向量表:

| | |
|-------------------|----------------|
| b reset | ;跳入 reset 处理程序 |
| b HandleUndef | ;跳入未定义处理程序 |
| b HandSWI | ;跳入软中断处理程序 |
| b HandPrefetchAbt | ;跳入预取指令处理程序 |
| b HandDataAbt | ;跳入数据访问中止处理程序 |
| b HandNoUsed | ;跳入未使用程序 |
| b HandleIRQ | ;跳入中断处理程序 |
| b HandleFIQ | ;跳入快速中断处理程序 |

通常,安装完异常向量表则跳到我们自己定义的处理程序入口,这时我们还没有保存被打断程序的现场,因此在异常处理程序的入口里先要保存打断程序现场。

保存执行现场:

异常处理程序最开始,要保存被打断程序的执行现场,程序的执行现场无非就是保存当前操作寄存器里的数据,可以通过下面的栈操作指令实现保存现场:

```
STMFD SP_excep!, {R0 - R12, LR_excep}
```

注:LR_abt,SP_excep 分别为对应异常模式下 LR 和 SP,为方便读者理解加上_abt

需要注意的是,在跳转到异常处理程序入口时,已经切换到对应异常模式下了,因此这里的 SP 是异常模式下的 SP_excep 了,所以被打断程序现场(寄存器数据)是保存在异常模式下的栈里,上述指令将 R0~R12 全部都保存到了异常模式栈,最后将修改完的被打断程序返回地址入栈保存;之所以保存该返回地址就是将来可以通过类似“MOV PC, LR”的指令,返回用户程序继续执行。

异常发生后,要针对异常类型进行处理,因此,每种异常都有自己的异常处理程序,异常处理过程通过下一小节的系统中断处理来进行分析。

3.2.5 异常处理的返回

异常处理完成之后,返回被打断程序继续执行,具体操作如下:

- 恢复被打断程序运行时寄存器数据。
- 恢复程序运行时状态 CPSR。
- 通过进入异常时保存的返回地址,返回到被打断程序继续执行。

异常发生后,进入异常处理程序时,将用户程序寄存器 R0~R12 里的数据保存在了异常模式下栈里面,异常处理完返回时,要将栈里保存的数据再恢复到原先 R0~R12 里。毫无疑问在异常处理过程中必须要保证异常处理入口和出口时栈指针 SP_excep 要一样,否则恢复到 R0~R12 里的数据不正确,返回被打断程序时执行现场不一致,出现问题,虽然将执行现场恢复了,但是此时还是在异常模式下,CPSR 里的状态是异常模式下状态,因此要恢复 SPSR_excep 里的保存状态到 CPSR 里。SPSR_excep 是被打断程序执行时的状态,在恢复 SPSR_excep 到 CPSR 的同时,CPU 的模式和状态从异常模式切换回了被打断程序执行时的模式和状态。此刻程序现场恢复了,状态也恢复了,但 PC 里的值仍然指向异常模式下的地址空间,我们要让 CPU 继续执行被打断程序,因此要再手动改变 PC 的值为进入异常时的返回地址,该地址在异常处理入口时已经计算好,直接将 PC = LR_excep 即可。

上述操作可以一步一步实现,但是通常可以通过一条指令实现上述全部操作:

```
LDMFD SP_excp!, {r0-r12, pc}^
```

注:SP_excep 为对应异常模式下 SP,^符号表示恢复 SPSR_excep 到 CPSR

以上操作可以用图 3-2 来描述。



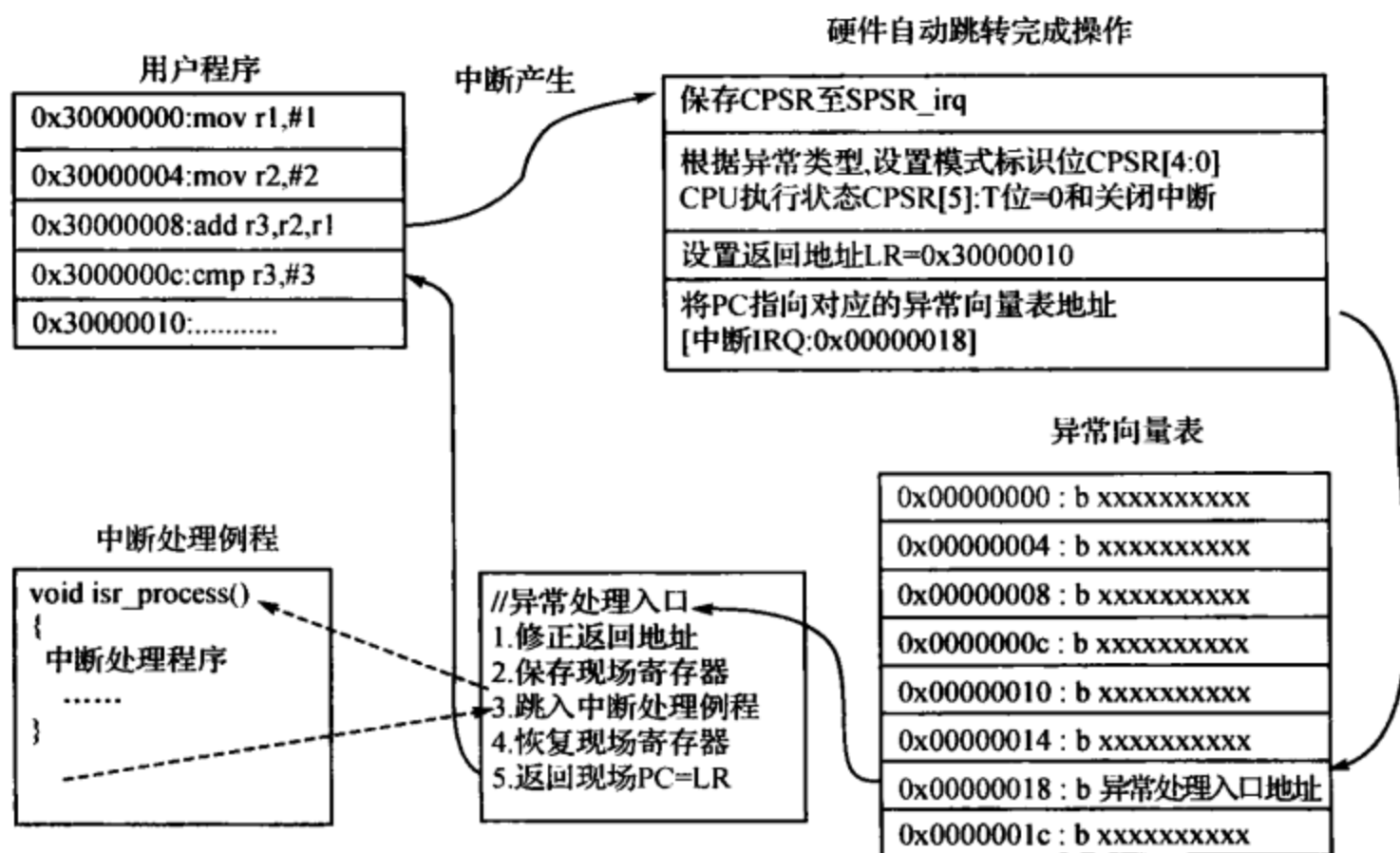


图 3-2 中断处理示意图

3.3 S3C2440 系统中断

CPU 和外设构成了计算机系统,CPU 和外设之间通过总线进行连接,用于数据通信和控制,CPU 管理监视计算机系统中所有硬件,通常以两种方式来对硬件进行管理监视:

(1) 查询方式:CPU 不停地去查询每一个硬件的当前状态,根据硬件的状态决定处理与否。好比是工厂里的检查员,不停的检查各个岗位工作状态,发现情况及时处理。这种方式实现起来简单,通常用在只有少量外设硬件的系统中,如果一个计算机系统有很多硬件,这种方式无疑是耗时、低效的,同时还大量占用 CPU 资源,并且对多任务系统反应迟钝。

(2) 中断方式:当某个硬件产生需要 CPU 处理的事件时,主动通过一根信号线“告知”CPU,同时设置某个寄存器里对应的位,CPU 一旦发现这根信号线上的电平有变化,就会中断当前程序,然后去处理发出该中断的请求。这就像是医院重危病房,病房每张病床床头有一个应急按钮,该按钮连接到病房监控室里控制台一盏指示灯,只要该张病床出现紧急情况病人按下按钮,病房监控室里电铃会响起,通知医护人员有紧急情况,医护人员这时查看控制台上的指示灯,找出具体病房,病床号,直接过去处理紧急情况。中断处理方式相对查询方式要复杂的多,并且需要硬件的支持,但是它处理的实时性更高,嵌入式系统里基本上都使用这种方式来处理。

第3章 ARM 体系结构

系统中断是嵌入式硬件实时地处理内部或外部事件的一种机制。对于不同 CPU 而言,中断的处理只是细节不同,大体处理流程都一样,S3C2440A 的中断控制器结构如图 3-3 所示:

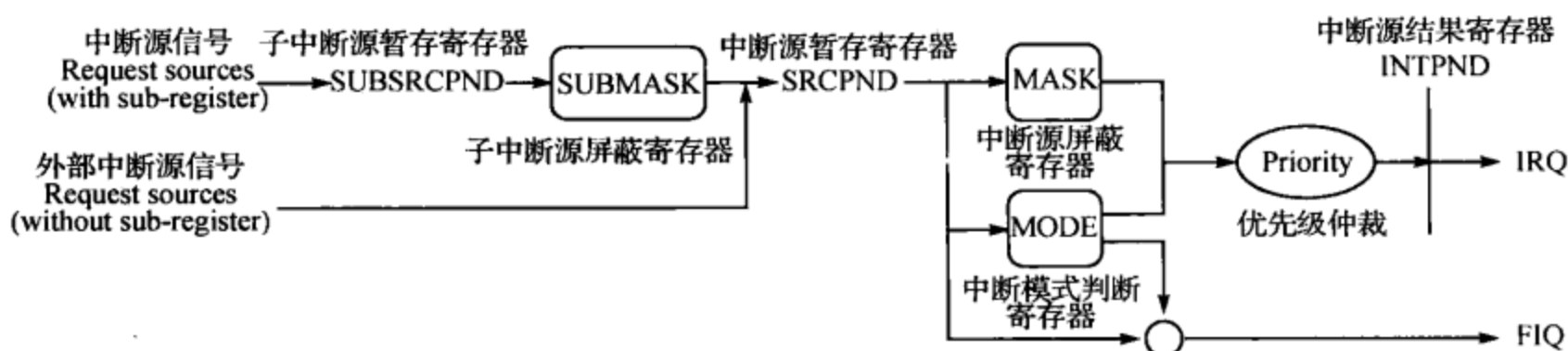


图 3-3 S3C2440 中断控制器

中断请求由硬件产生,根据中断源类型分别将中断信号送到 SUBSRCPND(SubSourcePending)和 SRCPND(SourcePending)寄存器,SUBSRCPND 是子中断源暂存寄存器,用来保存子中断源信号,SRCPND 是中断源暂存寄存器,用来保存中断源信号。中断信号可通过编程方式屏蔽掉,SUBMASK 是子中断源屏蔽寄存器,可以屏蔽指定的子中断信号,MASK 功能同 SUBMASK 用来屏蔽中断源信号。中断分为两种模式:一般中断和快速中断,MODE 是中断模式判断寄存器,用来判断当前中断是否为快速中断,如果为快速中断直接将快速中断信号送给 ARM 内核,如果不是快速中断,还要将中断信号进行仲裁选择。S3C2440A 支持多达 60 种中断,很有可能多个硬件同时产生中断请求,这时要求中断控制器做出裁决,Priority 是中断源优先级仲裁选择器,当多个中断产生时,选择出优先级最高的中断源进行处理,INTPND 是中断源结果寄存器,里面存放优先级仲裁出的唯一中断源。

3.3.1 中断的产生-中断源

S3C2440A 支持 60 种中断源,基本上满足了开发板内部、外围设备等对中断的需求。其中每一个中断源对应寄存器中的一位,显然要支持 60 种中断至少需要两个 32 位寄存器,SUBSRCPND 和 SRCPND 分别保存中断源信号。S3C2440A 对 60 种中断源的管理是按层级分的,如图 3-4 所示。

S3C2440A 将中断源分为两级:中断源和子中断源,中断源里包含单一中断源和复合中断源,复合中断源是子中断源的复合信号。如实时时钟中断,该硬件只会产生一种中断,它是单一中断源,直接将其中断信号线连接到中断源寄存器上。对于复合中断源,以 UART 串口为例进行说明,S3C2440A 可以支持 3 个 UART 串口,每个串口对应一个复合中断源信号 INT_UARTn,每个串口可以产生三种中断,也就是三个子中断:接收数据中断 INT_RXDn,发送数据中断 INT_TXDn,数据错误中断 INT_ERRn,这三个子中断信号在中断源寄存器复合为一

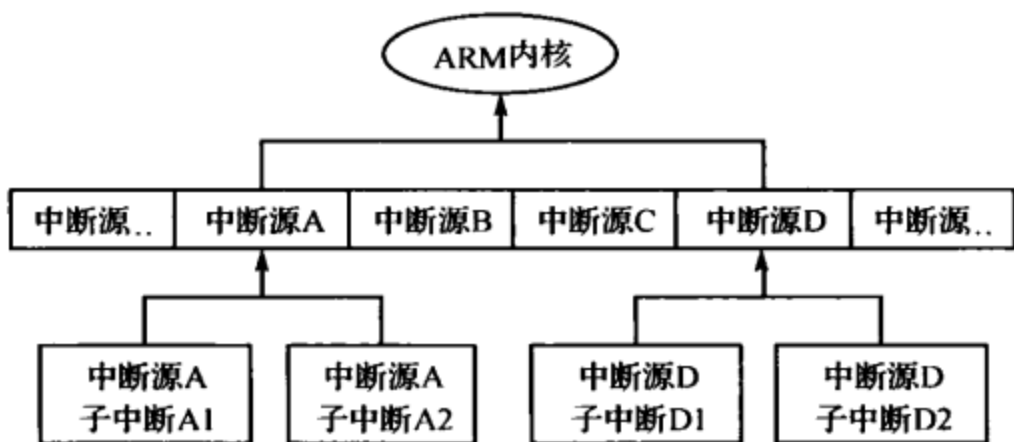


图 3-4 中断源信号复合示意图

个中断信号,三种中断任何一个产生都会将中断信号传递给对应的中断源 INT_UARTn,然后通过中断信号线传递给 ARM 内核,如图 3-5 所示。

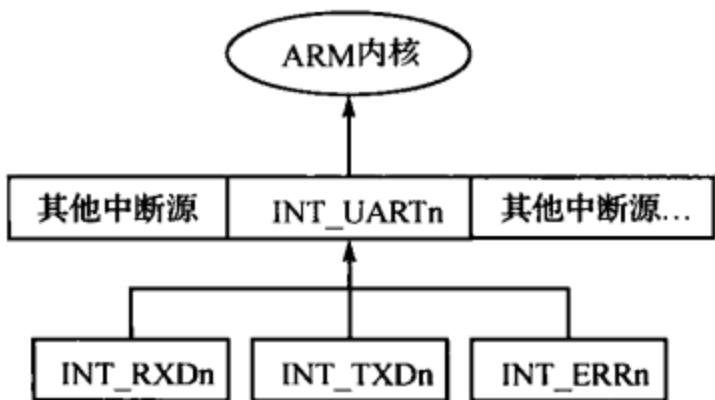


图 3-5 UART 串口中断源信号复合示意图

表 3-5 中列出了 S3C2440A 部分中断源,它分别对应中断源寄存器里某个位:详细中断源请查看 S3C2440A 硬件手册。

表 3-5 部分中断源信号

| 中断源 | 描 述 | 优先级仲裁分组 |
|--------------|--------------------|---------|
| INT_ADC | 数模转换和触摸屏中断 | ARB5 |
| INT_RTC | 实时时钟中断 | ARB5 |
| INT_UART0 | UART0 中断(包含子中断) | ARB5 |
| INT_NFCON | NandFlash 控制中断 | ARB4 |
| INT_WDT_AC97 | 看门狗中断 | ARB1 |
| EINT8-23 | 外部中断 8~23(包含外部子中断) | ARB1 |
| EINT4-7 | 外部中断 4~7(包含外部子中断) | ARB1 |
| EINT3 | 外部中断 3 | ARB0 |
| EINT2 | 外部中断 2 | ARB0 |

续表 3-5

| 中断源 | 描 述 | 优先级仲裁分组 |
|--------|--------|---------|
| EINT1 | 外部中断 1 | ARB0 |
| EINT 0 | 外部中断 0 | ARB0 |

中断信号除上述分法之外,还可以按照硬件位置分为外部中断源和内部中断源。

(1) 内部中断源:它是嵌入式系统中常见硬件产生的中断信号,比如 UART 串口中断源、时钟 Timer 中断源、看门狗中断源等。

(2) 外部中断源:有时嵌入式系统里要在外部接口上挂载一些外围设备,这些设备并不是一个通用嵌入式系统里必备硬件,比如蓝牙模块、各种传感器、WIFI 无线通信模块,这些硬件也要产生中断让 CPU 来处理数据,因此这些外设硬件通过中断信号线连接到中断控制器上,它们产生的中断称做外部中断信号。它们有着和内部中断一样的处理机制,只不过,它没有一个固定的中断号与之对应,硬件与嵌入式系统的连接方式与中断处理完全由系统硬件与软件设计者实现。

外设硬件通过输入输出接口 I/O Ports 挂接到嵌入式系统上,I/O Ports 向外设提供外部中断信号线、输出电源、频率时钟和输入输出信号线,外围硬件根据自己需要连接到 I/O Ports 上,产生中断时向外部中断信号线上送出中断信号,通过外部中断信号线传递到中断控制器。

按键 Key 可以认为最为简单的一种硬件设备了,如图 3-6 所示。

它功能很简单,按键 K1~K6 一端接地为低电平,另外一端接电源正极为高电平,EINT8、EINT11、EINT13、EINT14、EINT15、EINT19 这 6 根中断信号线分别和高电平端按键相连,当按键按下时电路接通,整个电路变成低电平,中断信号线上电压产生变化,通过设置中断触发方式,产生外部中断请求,输入到 CPU 内部,从而实现按键中断控制。

S3C2440A 可以支持 EINT0~EINT23 共 24 种外部中断,完全可以满足小型嵌入式设备外设硬件的需求。

外部中断源也分为外部中断源和外部子中断源,其处理方式和内部中断源基本一样。

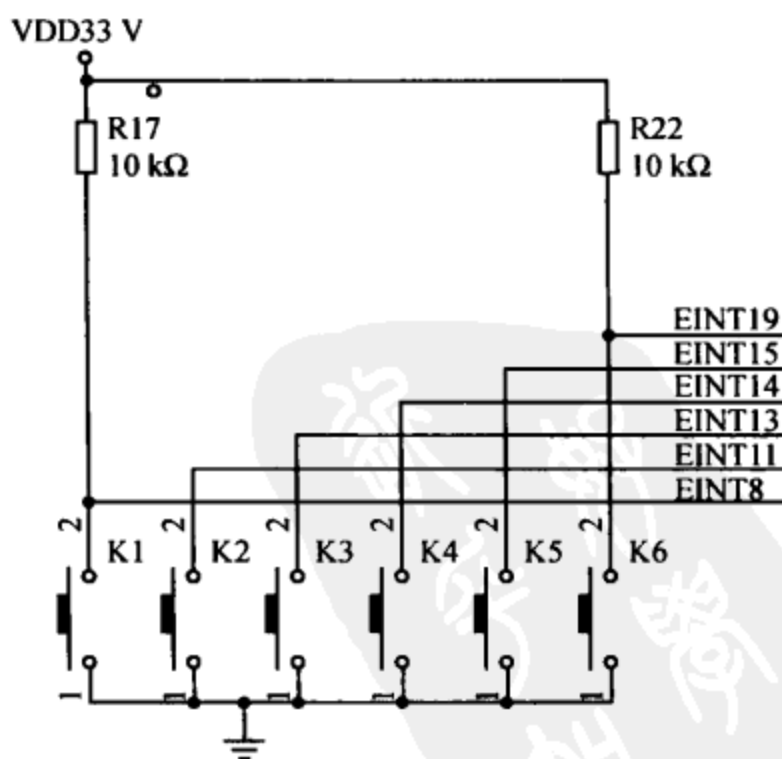


图 3-6 按键硬件接线原理图

3.3.2 中断优先级

S3C2440A 支持 60 种中断,多个硬件可能同时产生中断请求,由于 CPU 只能处理一个中断,中断控制器怎么选择一个最佳的中断,交给 ARM 内核进行处理呢? 中断控制器采用优先级仲裁比较的方式进行选择,找出优先级最高的中断源。中断控制器将 60 种中断源分成 7 组,如图 3-7 所示,它类似体育赛事里的比赛方式,所有参赛选手在小组赛 PK,选择出小组赛最优秀选手,然后进入决赛阶段和其他小组最优先选择再 PK,最后优胜者就是总冠军。其中 ARBITER0~ARBITER5 为“小组赛”阶段,中断源信号在各自小组里进行优先级仲裁,选择出最高优先级中断信号,每小组选出的中断信号送到 ARBITER6,也就是决赛阶段,选择出最高优先级中断信号,交给 ARM 内核。

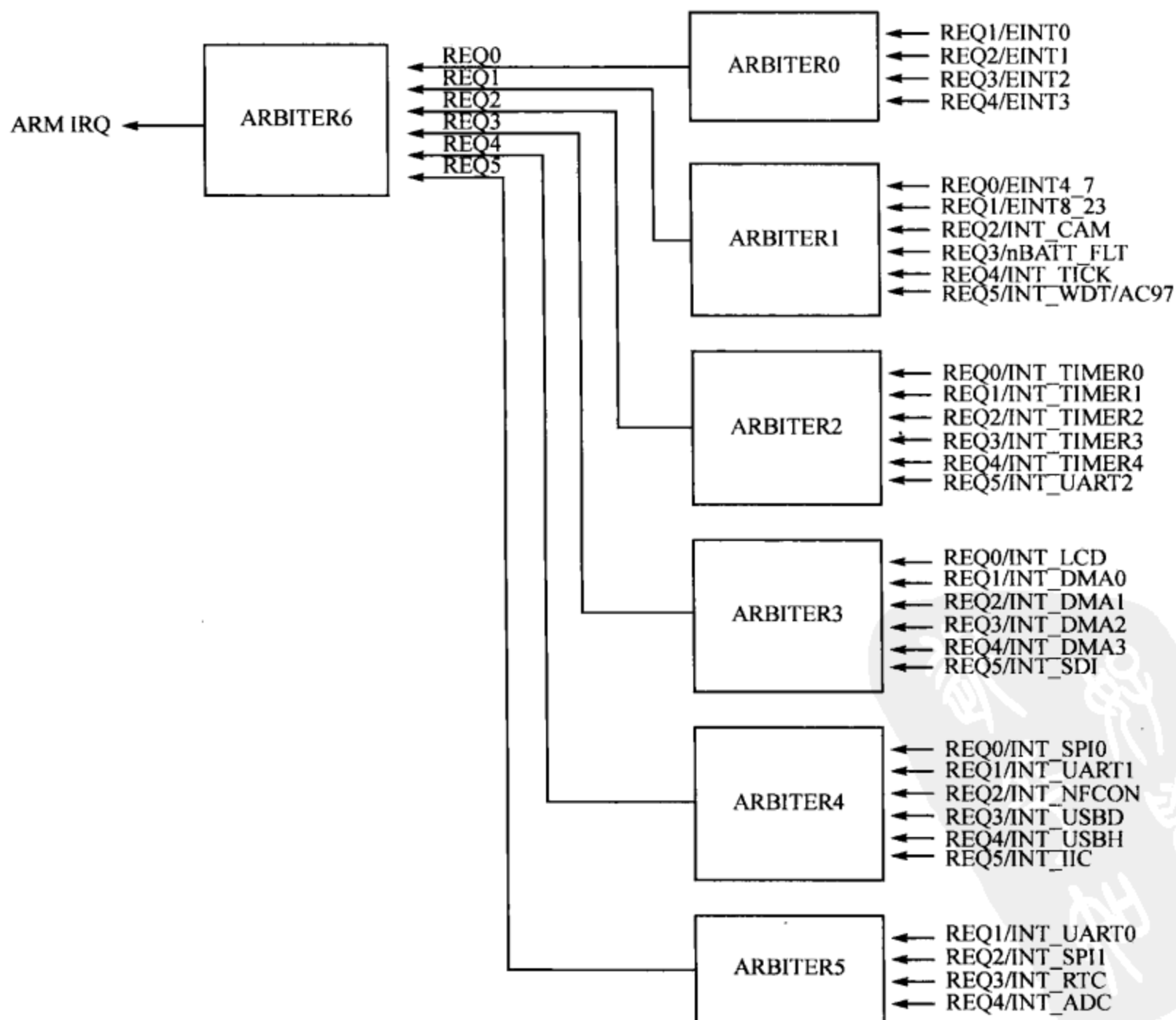


图 3-7 S3C2440 优先级仲裁示意图

中断信号在 7 个分组里 PK 时的优先级是可编程的,通过 PRIORITY 寄存器进行优先级设置。如表 3-6 所列(只列出 PRIORITY 寄存器部分位)。

表 3-6 中断优先级控制寄存器(PRIORITY)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|---|------------|-------|
| PRIORITY | 0x4A00000C | R/W | 中断优先级控制寄存器 | 0x7F |
| PRIORITY | 位 | 描 述 | | 初始值 |
| ARB_SEL6 | [20:19] | 仲裁组 6 优先级排序方式 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5 | | 0x00 |
| ARB_SEL5 | [18:17] | 仲裁组 5 优先级排序 00 = REQ 1-2-3-4 01 = REQ 2-3-4-1 10 = REQ 3-4-1-2 11 = REQ 4-1-2-3 | | 00 |
| ... | ... | ... | | ... |
| ARB_MODE6 | [6] | 仲裁组 6 优先级是否轮转: 0 = 不轮转, 1 = 轮转 | | 1 |
| ARB_MODE5 | [5] | 仲裁组 5 优先级是否轮转: 0 = 不轮转, 1 = 轮转 | | 1 |
| ... | ... | ... | | ... |

通过设置仲裁组 n 优先级排序方式位,设置每个仲裁组内中断信号的优先级顺序,比如: ARB_SEL5 分组时包含 4 个中断信号:REQ1 INT_UART0、REQ2 INT_SPI1、REQ3 INT_RTC、REQ4 INT_ADC、ARB_SEL5 位采用默认值:00,当 INT_UART0 和 INT_RTC 中断信号同时产生时,INT_UART0 会被选出,通过可编程方式改变优先级排序方式来改变中断信号优先级。

ARB_MODE0~ ARB_MODE6 为每个仲裁分组的优先级轮转设置位,采用默认值时,当前中断信号被选择处理之后,再次产生中断请求时,它的优先级自动轮转到该组最低,这样可以保证优先级低的中断信号可以被及时处理,不至于出现优先级高且中断请求频繁的中断每次都被优先处理,而优先级低的被“饿死”的情况。显然,这种方式更民主,实时性更佳。

3.3.3 中断控制器相关寄存器

(1) SUBSRCPND 子中断源暂存寄存器,如表 3-7 所列。

表 3-7 子中断源暂存寄存器 (SUBSRCPND)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|--------------|---------|---|------------|
| SUBSRCPND | 0x4A000018 | R/W | 子中断源暂存寄存器,保存中断请求状态: 0:没有中断请求信号 1:中断请求信号产生 | 0x00000000 |
| SUBSRCPND | 对应 SRCPND | 位 | 描 述 | 初始值 |
| Reserved | 无 | [31:15] | 未使用 | 0 |
| INT_AC97 | INT_WDT_AC97 | [14] | 0 = 未产生中断 1 = 产生中断 | 0 |
| ... | ... | ... | ... | ... |
| INT_RXD0 | INT_UART0 | [0] | 0 = 未产生中断 1 = 产生中断 | 0 |

该寄存器用来标识保存子中断源信号,当某个子中断信号产生之后,SUBSRCPND 对应位被自动置 1,该位会一直保持被置位,只到中断处理程序将其清除为止,需要注意一下,清除中断是通过向对应位写入 1 来清除,而不是写入 0,写入 0 无效。

(2) INTSUBMSK 子中断源屏蔽寄存器,如表 3-8 所列。

表 3-8 子中断源屏蔽寄存器 (INTSUBMSK)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|------------------|--|--------|
| INTSUBMSK | 0x4A00001C | R/W | 子中断源信号屏蔽存寄存器,设置相应位来屏蔽中断信号: 0:未屏蔽,中断可用 1:屏蔽中断信号 | 0xFFFF |
| INTSUBMSK | 位 | | 描 述 | 初始值 |
| Reserved | [31:15] | 未使用 | | 0 |
| INT_AC97 | [14] | 0 = 未屏蔽 1 = 屏蔽中断 | | 1 |
| ... | ... | ... | | ... |
| INT_RXD0 | [0] | 0 = 未屏蔽 1 = 屏蔽中断 | | 1 |

该寄存器用来屏蔽子中断源信号,默认值为全部子中断都被屏蔽掉,因此要想处理某个硬件中断,必须要打开中断屏蔽位,通过写入 0 来取消屏蔽中断。

(3) SRCPND 中断源暂存寄存器,如表 3-9 所列。

表 3-9 中断源暂存寄存器(SRCPND)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|--------------------|--|------------|
| SRCPND | 0x4A000000 | R/W | 中断源暂存寄存器,保存中断请求状态: 0:没有中断请求信号 1:中断请求信号产生 | 0x00000000 |
| SRCPND | 位 | | 描 述 | 初始值 |
| INT_ADC | [31] | 0 = 未产生中断 1 = 产生中断 | | 0 |
| ... | ... | ... | | ... |
| EINT0 | [0] | 0 = 未产生中断 1 = 产生中断 | | 0 |

该寄存器用来保存中断源信号,当某个中断信号产生之后,SRCPND 对应位被自动置 1,该位会一直保持被置位,只到中断处理程序将其清除为止,需要注意一下,清除中断是通过向对应位写入 1 来清除,而不是写入 0,写入 0 无效。

(4) INTMSK 中断源屏蔽寄存器,如表 3-10 所列。

表 3-10 中断源屏蔽寄存器(INTMSK)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|------------------|--|------------|
| INTMSK | 0x4A000008 | R/W | 中断源信号屏蔽寄存器,设置相应位来屏蔽中断信号: 0:未屏蔽,中断可用 1:屏蔽中断信号 | 0xFFFFFFFF |
| INTMSK | 位 | | 描 述 | 初始值 |
| INT_ADC | [31] | 0 = 未屏蔽 1 = 屏蔽中断 | | 1 |
| ... | ... | ... | | ... |
| EINT0 | [0] | 0 = 未屏蔽 1 = 屏蔽中断 | | 1 |

该寄存器用来屏蔽中断源信号,默认值为全部中断都被屏蔽掉,因此要想处理某个硬件中断,必须要打开中断屏蔽位,通过写入 0 来取消屏蔽中断。

(5) INTPND 最高优先级中断暂存寄存器,如表 3-11 所列。

表 3-11 最高优先级中断暂存寄存器(INTPND)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|--------------------|--|------------|
| INTPND | 0x4A000010 | R/W | 最高优先级中断暂存寄存器里面保存有经过优先级仲裁的结果： 0:没有中断请求信号 1:中断请求信号产生 | 0x00000000 |
| INTPND | 位 | | 描 述 | 初始值 |
| INT_ADC | [31] | 0 = 未产生中断 1 = 产生中断 | | 0 |
| ... | ... | ... | | ... |
| EINT0 | [0] | 0 = 未产生中断 1 = 产生中断 | | 0 |

该寄存器保存了经过优先级仲裁出的中断信号位,它是所有当前中断请求里优先级别最高的中断,因此该寄存器里的值最多有一位被置 1,通常中断处理程序中会通过读取该寄存器的值来获得当前正在处理的中断请求。中断处理完成之后,通过写入 1 来清除中断。

(6) INTOFFSET 中断号偏移量寄存器,如表 3-12 所列。

表 3-12 中断号偏移量寄存器(INTOFFSET)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|------|------------------------|------------|
| INTOFFSET | 0x4A000014 | R | 中断号偏移量寄存器,用来保存当前处理的中断号 | 0x00000000 |

该寄存器里存放的是经过优先级仲裁出的中断信号对应的中断号,是一个 0~31 之间的整数,其实它就是 INTPND 里对应的位号,比如:INT_UART0 产生了中断,INTPND 里第 28 位置 1,INTOFFSET 里保存的整数就是 28,多出来这个寄存器的目的主要是方便中断处理程序查询中断源,清除中断源:

```

#define TIMER0_IRQ_OFT      10           // 时钟 0 定时中断
#define EINT0_IRQ_OFT      0           // 开发板 K1 按键 1 对应外部中断 EINT0
void handle_irq()
{
    unsigned long irqOffSet = INTOFFSET; // 取得中断号
    switch(irqOffSet)
    {
        case TIMER0_IRQ_OFT:           // 当前中断为 0 定时中断
            do_timer();                 // 跳入定时器 0 处理程序
            break;
        case EINT0_IRQ_OFT:             // 当前中断为 K1 按键触发
    
```



```

do_key1_pressed();           // 处理 K1 按下事件
break;

}

SRCPND &= (1<<irqOffSet);    // 清除中断源
INTPND = INTPND;             // 清除最高优先级中断暂存寄存器中断
}

```

(7) INTMOD 中断模式寄存器,如表 3-13 所列。

表 3-13 中断模式寄存器(INTMOD)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|-----------------|---|-----------|
| INTMOD | 0x4A000004 | R/W | 中断模式寄存器,指定对应中断模式: 0 = IRQ 一般中断模式 1 = FIQ 快速中断模式 | 0x0000000 |
| INTMOD | 位 | | 描 述 | 初始值 |
| INT_ADC | [31] | 0 = IRQ 1 = FIQ | | 0 |
| ... | ... | ... | | ... |
| EINT0 | [0] | 0 = IRQ 1 = FIQ | | 0 |

通过设置 INTMOD 寄存器对应位,来指定对应中断模式,如果指定为一般中断,那么中断信号会进行优先级仲裁,如果指定为快速中断,那么中断信号直接送给 ARM 内核产生中断。需要注意的是,快速中断不存在优先级仲裁,只能有一位被设置为 FIQ 模式。

3.3.4 系统中断流程

中断源按照硬件位置分为外部中断源和内部中断源,外部中断源和内部中断源又包含子外部中断源和子内部中断源,如图 3-8 所示。

1. 子内部中断源的产生

以 UART0 接收数据产生 INT_RXD0 中断为例,INT_RXD0 产生后进入 SUBSRCPND 子中断源暂存寄存器,设置 INT_RXD0 对应的中断位,中断信号经过 INTSUBMSK 子中断屏蔽寄存器,如果 INT_RXD0 信号对应位没有被置位(屏蔽掉),中断信号继续向前传递,经过子内部中断源聚合器,将 INT_RXD0 聚合成对应的中断源信号 INT_UART0,设置 SRCPND 中断源暂存寄存器里 INT_UART0 位,经过 INTMSK 中断屏蔽寄存器,如果 INT_UART0 信号没有被屏蔽掉,中断信号进入 INTMOD 中断模式寄存器判断是否为快速中断,如果被编程为快速中断,直接打断 ARM 内核,进入中断处理,如果中断信号为一般中断,进入中断优先级仲裁器进入优先级仲裁,如果 INT_UART0 信号为最高优先级或只有 INT_UART0 中断信

号产生,则该中断信号被记录到 INTPND 最高优先级中断暂存寄存器,同时设置 INTOFFSET 的值为中断号 28,最终将中断信号打断 ARM 内核进行中断处理。如果同时产生多个中断且 INT_UART0 不是最高优先级,则该中断信号不会被处理,等最高优先级信号处理完后,再次进行优先级仲裁,也就是说中断信号不消失,一直保存在 SRCPND 里,只到被处理为止。

2. 内部中断源的产生

该过程在子内部中断处理过程中已经包含,中断信号产生后直接进入 SRCPND 里,然后经历上述子内部中断后期处理过程。

3. 子外部中断的产生

外部中断源共有 24 个,其中 EINT0~EINT3 为外部中断源,EINT4_7、EINT8_23 为复合中断源,它们包含有子外部中断源。

由于外部硬件直接挂接到 I/O Ports(详见 S3C2440A 硬件手册第 9 章)上的,要想让外设硬件中断得到处理先从 EINT0~EINT23 里选择中断信号,现以 EINT11 为例介绍子外部中断处理过程。

通常 CPU 内部引出引脚都是复用的,也就是说一根 CPU 引脚可以有多种功能,可以设置其为输入信号线、输出信号线或中断信号线,要想让硬件产生中断,首先要对可以产生中断的引脚进行编程,设置该引脚为中断信号线。EINT11 中断信号对应 CPU 引脚为 GPG3,通过设置 GPGCON[7:6] = 0b10,如表 3-14 所列,可以设置该引脚为中断信号线。

表 3-14 GPGCON 寄存器

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|--------|------------|-------------------------|-------------------------------|-------|
| GPGCON | 0x56000060 | R/W | Configures the pins of port G | 0x0 |
| GPG3 | [7:6] | 00=Input 10=EINT[11] | 01=Output 11=nSSI | |

如图 3-9 所示,设置了接线引脚为中断信号线之后,还要通过设置 EXTINT0 寄存器来指定中断信号的触发方式:高电平触发,低电平触发,电平上升沿,下降沿,双沿触发。

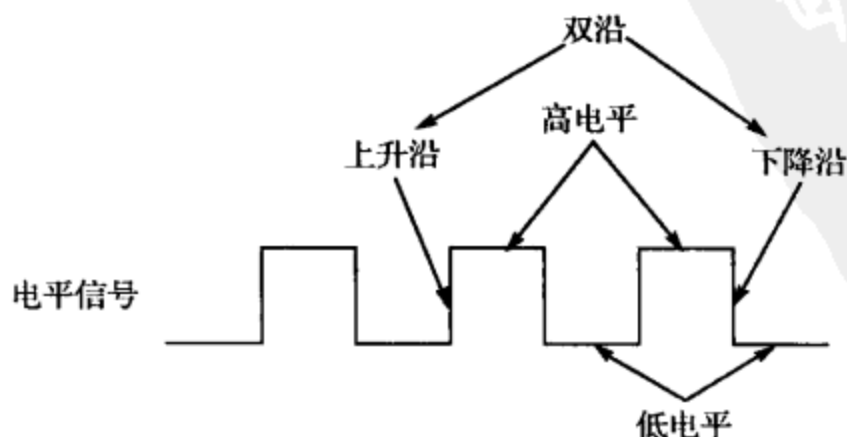


图 3-9 电平信号触发示意图

如表 3-15 所列,由于按键按下时让它产生中断,也就是从高电平变为低电平时产生(上节按键中断原理),因此设置 EINT11 的中断信号触发方式为下降沿触发,EXTINT1[14:12] = 0b01x

表 3-15 EXTINT1 寄存器

| EXTINT1 | 0x5600008c | R/W | External Interrupt control Register 1 | 0x000000 |
|---------|------------|---|---------------------------------------|----------|
| EINT11 | [14:12] | Setting the signaling method of the EINT11 000=Low level 001=High level 01x=Falling edge triggered 10x=Rising edge triggered 11x=Both edge triggered | | |

设置完触发方式之后,当外设中断信号线上的电平达到触发条件时,通过外部中断产生器产生中断信号,然后将子外部中断暂存寄存器 EINTPEND 中对应的 EINT11 位置 1,中断信号再进入 EINTMSK 子外部中断屏蔽寄存器,如果 EINT11 中断源信号没有被屏蔽,则 EINT11 中断信号进入子外部中断聚合器,复合成 EINT8_23 中断信号,然后再经历与前面子内部中断信号一样的处理机制。

(1) EINTPEND 外部中断暂存寄存器如表 3-6 所列。

表 3-16 外部中断暂存寄存器(EINTPEND)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|-----|---|------------|
| EINTPEND | 0x560000A8 | R/W | 外部中断信号暂存寄存器 0:没有中断请求信号 1:中断请求信号产生 | 0x00000000 |
| EINTPEND | 位 | | 描 述 | 初始值 |
| EINT23 | [23] | | 0 = 未产生中断 1 = 产生中断 | 0 |
| ... | ... | | ... | ... |
| EINT4 | [4] | | 0 = 未产生中断 1 = 产生中断 | 0 |
| 保留位 | [3:0] | 无 | | 0000 |

(2) EINTMASK 外部中断屏蔽寄存器如表 3-17 所列。

表 3-17 外部中断屏蔽寄存器(EINTMASK)

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|----------|------------|-----|---------------------------------------|--------------|
| EINTMASK | 0x560000A4 | R/W | 外部中断信号屏蔽寄存器 0:未屏蔽,中断可用 1:屏蔽中断信号 | 0x000FFFFFFF |

续表 3-17

| EINTMASK | 位 | 描 述 | 初始值 |
|----------|-------|------------------|------|
| EINT23 | [23] | 0 = 未屏蔽 1 = 屏蔽中断 | 1 |
| ... | ... | ... | ... |
| EINT4 | [4] | 0 = 未屏蔽 1 = 屏蔽中断 | 1 |
| 保留位 | [3:0] | 无 | 1111 |

4. 外部中断源的产生

外部中断产生过程读者可以根据上面中断图自行分析。

3.3.5 按键控制 LED 灯实验

本实验源码存放在光盘目录\work\armarch\sys_irq_开发板下,本实验分 3 个版本,分别针对 3 种开发板:友善之臂 QQ2440,友善之臂 MINI2440,天嵌 TQ2440。每种开发板对应工程在:“sys_irq_开发板名”目录下。下面实验内容为针对 MINI2440 开发板。

head.s:

主要实现安装异常向量表,处理复位异常,初始化必要硬件,中断入口处理等功能。

```

;*****
; 系统中断实验(MINI2440)
;*****
GPBCON    EQU        0x56000010
GPBDAT    EQU        0x56000014
EXPORT SYS_IRQ
AREA SYS_IRQ, CODE, READONLY
ENTRY

; *****
; 设置中断向量,除 Reset 和 HandleIRQ 外,其他异常都没有使用(如果不幸发生了,
; 将导致死机)
; *****
; 0x00: 复位 Reset 异常
b    Reset

; 0x04: 未定义异常(未处理)
HandleUndef
b    HandleUndef

; 0x08: 软件中断异常(未处理)

```


HandleSWI

b HandleSWI

; 0x0c: 指令预取异常(未处理)

HandlePrefetchAbt

b HandlePrefetchAbt

; 0x10: 数据访问中止异常(未处理)

HandleDataAbt

b HandleDataAbt

; 0x14: 未使用异常(未处理)

HandleNotUsed

b HandleNotUsed

; 0x18: 一般中断异常,跳往 HandleIRQ

b HandleIRQ

; 0x1c: 快速中断异常(未处理)

HandleFIQ

b HandleFIQ

Reset

; 复位异常处理入口

; 关闭看门狗

ldr r0, = 0x53000000

mov r1, #0

str r1, [r0]

bl initmem

ldr sp, = 0x32000000

; 设置管理模式栈指针

IMPORT uart_init

bl uart_init

; UART 串口初始化

IMPORT irq_init

bl irq_init

; 系统中断初始化

IMPORT key_init

资源解密
PDG

```

        bl    key_init                ; 按键初始化

        IMPORT led_init
        bl    led_init                ; LED 灯初始化

        msr    cpsr_cxsf, #0xd2      ; 切换到中断模式下
        ldr    sp,      = 0x31000000 ; 设置中断模式栈指针

        msr    cpsr_cxsf, #0x13      ; 返回管理模式

        ldr    lr,      = halt_loop   ; 设置管理模式返回地址
        IMPORT main
        ldr    pc,      = main        ; 跳入主函数 main 里执行
        ; *****
        ; 中断处理
        ; *****

HandleIRQ
        sub    lr,lr,#4                ; 修正返回地址
        stmdb  sp!,{r0-r12,lr}        ; 保存程序执行现场
        ldr    lr, = int_return        ; 设置中断处理程序返回地址
        IMPORT handle_irq
        ldr    pc, = handle_irq        ; 跳入中断处理程序

int_return                                ; 中断处理返回标签
        ldmbia sp!,{r0-r12,pc}~       恢复程序执行现场,返回继续执行

halt_loop
        b halt_loop

initmem
        ldr r0, = 0x48000000
        ldr r1, = 0x48000034
        ;ldr r2, = memdata
        adr r2, memdata

initmemloop
        ldr r3, [r2], #4
        str r3, [r0], #4
        teq r0, r1
        bne initmemloop

```



```

mov    pc,lr

memdata
DCD    0x22000000      ;BWSCON
DCD    0x00000700      ;BANKCON0
DCD    0x00000700      ;BANKCON1
DCD    0x00000700      ;BANKCON2
DCD    0x00000700      ;BANKCON3
DCD    0x00000700      ;BANKCON4
DCD    0x00000700      ;BANKCON5
DCD    0x00018005      ;BANKCON6
DCD    0x00018005      ;BANKCON7
DCD    0x008e07a3      ;REFRESH
DCD    0x000000b1      ;BANKSIZE
DCD    0x00000030      ;MRSRB6
DCD    0x00000030      ;MRSRB7

END                                ; 代码结束

```

该程序主要设置异常向量表,除了 Reset 异常和中断处理被处理以外,其他异常都未被处理,如果发生时,会产生死循环,Reset 异常里主要实现了硬件的基本初始化,如按键、LED 灯等,设置栈指针,用于执行 C 程序,最后跳入 C 程序的 main 函数。在中断处理异常处理中首先修正返回地址,保存用户执行现场,跳入到中断处理例程中执行。

sys_init.c:

硬件初始化文件,里面包含 LED,KEY 的初始化函数。

```

#include "register.h"
#include "comm_fun.h"

#define TXDREADY    (1<<2)      //发送数据状态 OK
#define RXDREADY    (1)          //接收数据状态 OK

/* UART 串口初始化 */
void uart_init()
{
    GPHCON |= 0xa0;              //GPH2,GPH3 used as TXD0,RXD0
    GPHUP   = 0x0;               //GPH2,GPH3 内部上拉
    ULCON0  = 0x03;              //8N1
    UCON0   = 0x05;              //查询方式为轮询或中断;时钟选择为 PCLK
    UFCON0  = 0x00;              //不使用 FIFO
}

```

```

        UMCON0 = 0x00;
        UBRDIV0 = 12;
    }

    /* UART 串口单个字符打印函数 */
    extern void putc(unsigned char c)
    {
        while( !(UTRSTAT0 & TXDORREADY) );
        UTXH0 = c;
    }

    /* UART 串口接受单个字符函数 */
    extern unsigned char getc(void)
    {
        while( !(UTRSTAT0 & RXDORREADY) );
        return URXH0;
    }

    /* UART 串口字符串打印函数 */
    extern int printk(const char * str)
    {
        int i = 0;
        while( str[i] ){
            utc( (unsigned char) str[i++] );
        }
        return i;
    }

    /* 按键初始化 */
    int key_init()
    {
        // 设置 K1,K2,K3,K4,K5,K6 对应控制寄存器为中断模式
        GPGCON = (2<<0) | (2<<6) | (2<<10) | (2<<12) | (2<<14) | (2<<22);
        /*
        01x falling edge triggered 下降沿触发
        10x Rising edge triggered 上升沿触发
        11x Both edge triggered 双沿触发
        */
        // 设置 K1,K2,K3,K4,K5 按键中断触发方式为上升沿触发
    }

```

```

EXTINT1 = (3<<0) | (3<<12) | (3<<20) | (3<<24) | (3<<28);
EXTINT2 = (3<<12);           // 设置 K6 按键中断触发方式为上升沿触
printk("按键初始化 OK\r\n");
return 0;
}

/* LED1~LED4 初始化 */
#define LED1      (1<<5)           //LED1 GPBDAT[5]
#define LED2      (1<<6)           //LED2 GPBDAT[6]
#define LED3      (1<<7)           //LED3 GPBDAT[7]
#define LED4      (1<<8)           //LED4 GPBDAT[8]

/* 点亮对应 num 号 LED 灯 */
extern int led_on(int num)
{
    switch(num)
    {
        case 1:
            GPBDAT = GPBDAT & ~LED1; break;
        case 2:
            GPBDAT = GPBDAT & ~LED2; break;
        case 3:
            GPBDAT = GPBDAT & ~LED3; break;
        case 4:
            GPBDAT = GPBDAT & ~LED4; break;
        default:
            return 0;
    }
    return num;
}

/* 关闭 num 号 LED 灯 */
extern int led_off(int num)
{
    switch(num)
    {
        case 1:
            GPBDAT = GPBDAT | LED1; break;
        case 2:

```

资源解密网
PDG


```

        GPBDAT = GPBDAT | LED2; break;
    case 3:
        GPBDAT = GPBDAT | LED3; break;
    case 4:
        GPBDAT = GPBDAT | LED4; break;
    default:
        return 0;
    }
    return num;
}

/* 关闭全部 LED 灯 */
extern int all_led_off(void)
{
    GPBDAT = GPBDAT | LED1 | LED2 | LED3 | LED4;
    return 0;
}

/* LED 灯初始化 */
int led_init(void)
{
    GPBCON = 0x15400;           //设置 GPB7 为输出口
    all_led_off();
    printf("led 初始化 OK\r\n");
    return 0;
}

/* 中断初始化 */
void irq_init(void)
{
    // 打开 KEY1~KEY6 的屏蔽位
    INTMSK &= ~(1<<5);
    EINTMASK &= ~((1<<8) | (1<<11) | (1<<13) | (1<<14) | (1<<15) | (1<<19));
    printf("中断初始化 OK\r\n");
}

```

该文件是相关硬件初始化程序,主要包含了看门狗驱动、按键驱动、系统中断驱动、LED 驱动。

handle_irq.c:

中断处理函数,查出中断源,中断处理,清除中断源。

```
#include "register.h"
#include "comm_fun.h"

#define EINT_Key_REQUEST    5           // Key 中断源中断号(6 个按键全部使用外部子中断)
#define K1_EINT_BIT        (1<<8)     // K1 外部子中断位
#define K2_EINT_BIT        (1<<11)    // K2 外部子中断位
#define K3_EINT_BIT        (1<<13)    // K3 外部子中断位
#define K4_EINT_BIT        (1<<14)    // K4 外部子中断位
#define K5_EINT_BIT        (1<<15)    // K5 外部子中断位
#define K6_EINT_BIT        (1<<19)    // K6 外部子中断位
/* 系统中断处理函数 */
void handle_irq()
{
    unsigned long irqOffSet = INTOFFSET; // 取得中断号
    all_led_off();                       // 关闭全部 LED 灯
    if(EINT_Key_REQUEST == irqOffSet){   // Key 中断产生(6 个按键使用一个总中断号)
        if(K1_EINT_BIT & EINTPEND){
            led_on(1);                   // 点亮 LED1
            printk("Key1 pressed\r\n");
            EINTPEND &= K1_EINT_BIT;     // 清除外部子中断源
        }else if(K2_EINT_BIT & EINTPEND){
            led_on(2);                   // 点亮 LED2
            printk("Key2 pressed\r\n");
            EINTPEND &= K2_EINT_BIT;     // 清除外部子中断源
        }else if(K3_EINT_BIT & EINTPEND){
            led_on(3);                   // 点亮 LED3
            printk("Key3 pressed\r\n");
            EINTPEND &= K3_EINT_BIT;     // 清除外部子中断源
        }else if(K4_EINT_BIT & EINTPEND){
            led_on(4);                   // 点亮 LED4
            printk("Key4 pressed\r\n");
            EINTPEND &= K4_EINT_BIT;     // 清除外部子中断源
        }else if(K5_EINT_BIT & EINTPEND){
            all_led_off(1);              // 熄灭全部 LED
            printk("Key5 pressed\r\n");
            EINTPEND &= K5_EINT_BIT;     // 清除外部子中断源
        }else if(K6_EINT_BIT & EINTPEND){
            all_led_on();                // 点亮全部 LED
        }
    }
}
```

```

        printk("Key6 pressed\r\n");
        EINTPEND &= K6_EINT_BIT;           // 清除外部子中断源
    }
}
SRCPEND &= (1<<irqOffSet);               // 清除中断源
INTPEND = INTPND;                         // 清除中断结果
}

main.c:
包含主函数和延时函数, 主要实现字符串的循环打印。

#include "register.h"
#include "comm_fun.h"

/* 延时 */
void delay(int msec)
{
    int i, j;
    for(i = 1000; i > 0; i--)
        for(j = msec * 10; j > 0; j--)
            /* do nothing */;
}

/* 主函数 */
int main()
{
    while(1)
    {
        printk("main 函数在运行...\r\n");
        delay(5);           //delay
    }
    return 0;
}

```

3.4 semihosting 与硬件重定向

应用程序在执行过程中经常会和主机有 I/O 交互请求, 例如 C 程序中的 printf, 该系统函数被执行时, 会通过软件中断将 printf 请求提交给操作系统内核, 内核将 printf 要打印的数据复制到内核空间, 通过调用显示器驱动程序接口, 将数据显示到显示器上, 如图 3-10 所示:

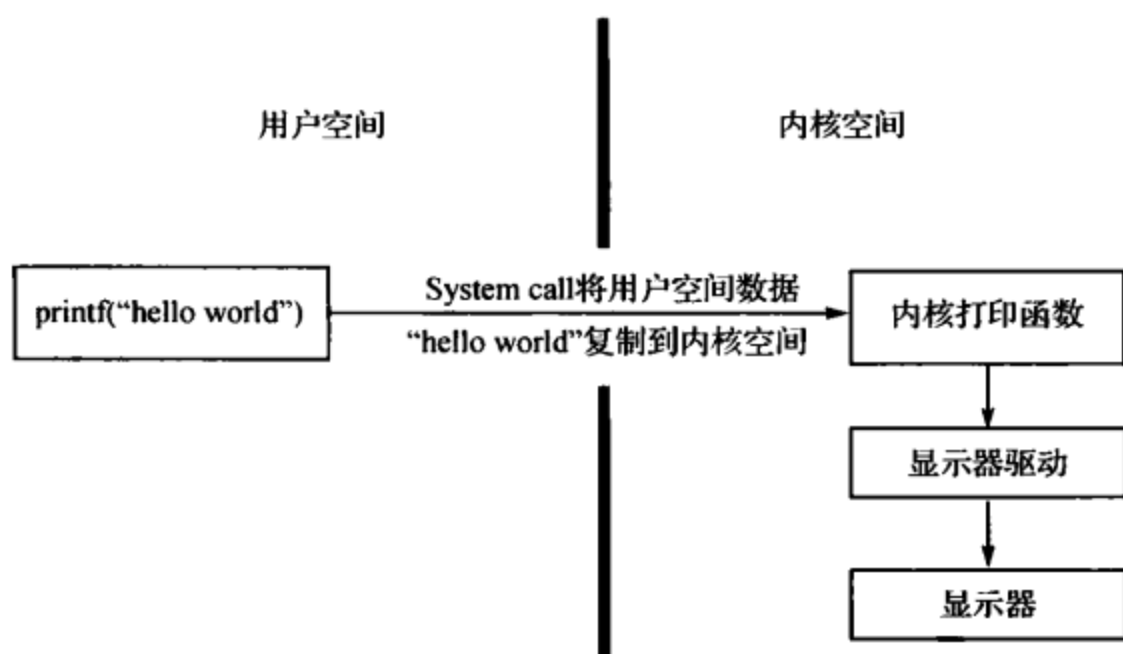


图 3-10 本地主机 I/O 请求示意图

上述应用程序 I/O 请求是最常见的一种本地主机请求方式,在嵌入式系统开发过程中还存在一种半主机请求模式 semihosting。

3.4.1 semihosting 半主机调试

semihosting 技术将应用程序中的 I/O 请求通过一定的通道传送到主机(host),由主机上的资源响应应用程序的 I/O 请求,而不是像在主机上执行本地应用程序一样,由应用程序所在的计算机响应应用程序 I/O 请求,也就是将目标板的输入/输出请求从应用程序代码传递到远程运行调试器的主机的一种机制。简单来说,目标开发板上通常不会有输入/输出这些外设,开发板运行的代码想要将结果打印出来,或者获得用户的输入,可以通过请求远程主机 I/O 设备来实现,如显示器、键盘等。目标开发板执行代码中加入对输入/输出设备进行访问函数,如 printf、scanf 等,这些函数并不是目标开发板的库函数,而是远程主机交叉编译器中带有的库函数,这些库函数被编译时,编译成一条软件中断指令。当目标开发板上电运行之后,执行到请求访问输入/输出设备指令时,产生特定中断号的软件中断 SWI(软件中断 SWI 指令请 3.5 系统调用与软件中断 SWI 的实现章节),与开发板相连的调试器会先截获目标板 SWI 请求,由于开发板程序中也可能存在用户自定义软件中断,为了区分二者,调试器会根据 SWI 的软中断号来判断是不是 semihosting 模式 I/O 请求,如果是,则取出 R0 寄存器里代表的具体请求号,然后使用远程主机来响应目标板具体 I/O 请求,而不是开发板本身去处理 semihosting 请求。

代表 semihosting 模式软中断号:

- 0x123456(ARM 模式)。
- 0xab(Thumb 模式)。

具体 semihosting 请求,通过寄存器 R0 里的值决定。

semihosting 仅仅是一种调试手段,它的工作原理就是利用调试器捕捉目标环境运行过程中产生的值为 0x123456 的 SWI 中断,然后向远程主机调试环境发送对应的调试信息。也就是说目标开发板通过特定的软件中断指令,借用了远程主机的输入输出设备实现 I/O 请求的访问,如图 3-11、图 3-12 所示。对于脱离调试环境开发板上运行的应用代码来说,是没有该机制的,也就是说是非 semihost 类型 I/O 请求。

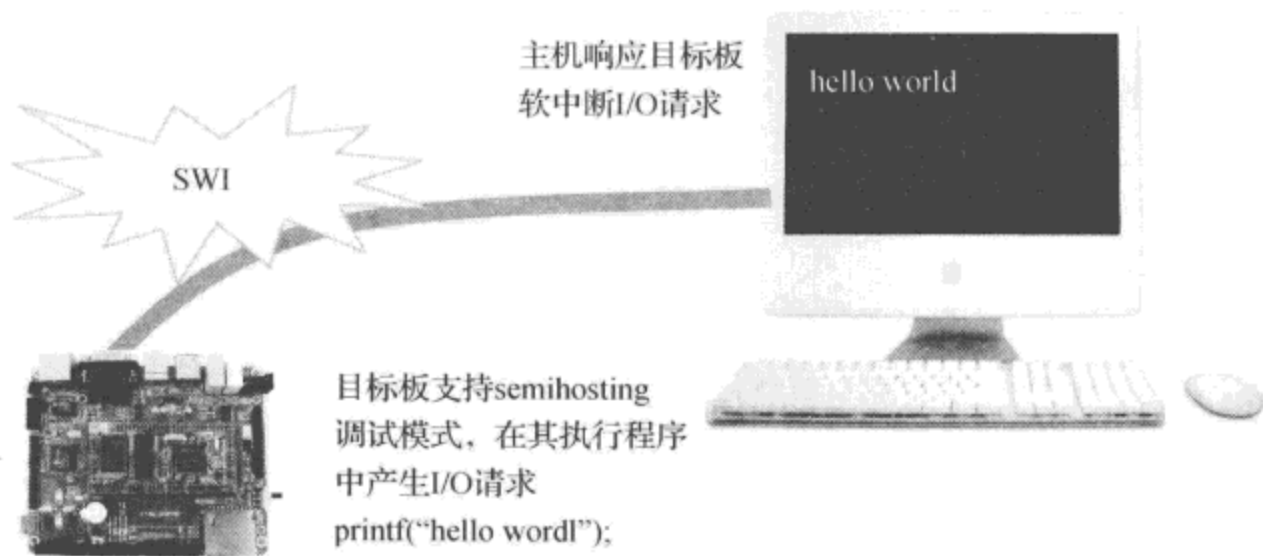


图 3-11 semihosting 远程主机 I/O 请求(一)

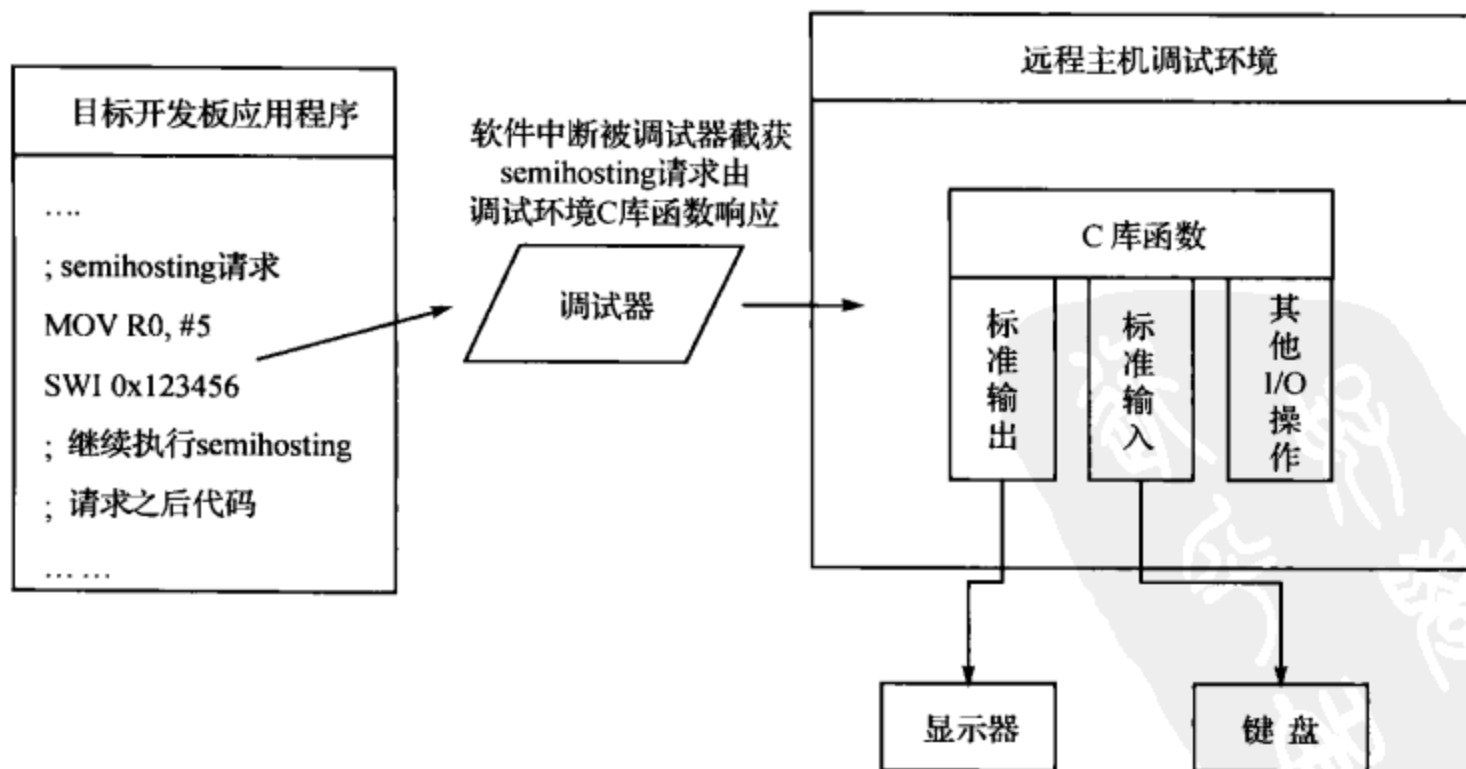


图 3-12 semihosting 远程主机 I/O 请求(二)

Semihosting 实验

本实验存放在随书光盘目录\work\armarch\semihosting_helloworld 工程。

打开 semihosting_helloworld.mcp 工程文件,由于当前工程内没有对看门狗,内存的初始化操作代码,为了保证程序正常运行,可以让调试器加载开发板初始化脚本(如果 H-JTAG 加载了初始化脚本,则调试器不加载初始化脚本也可以正常执行),通过图 3-13 进行设置。

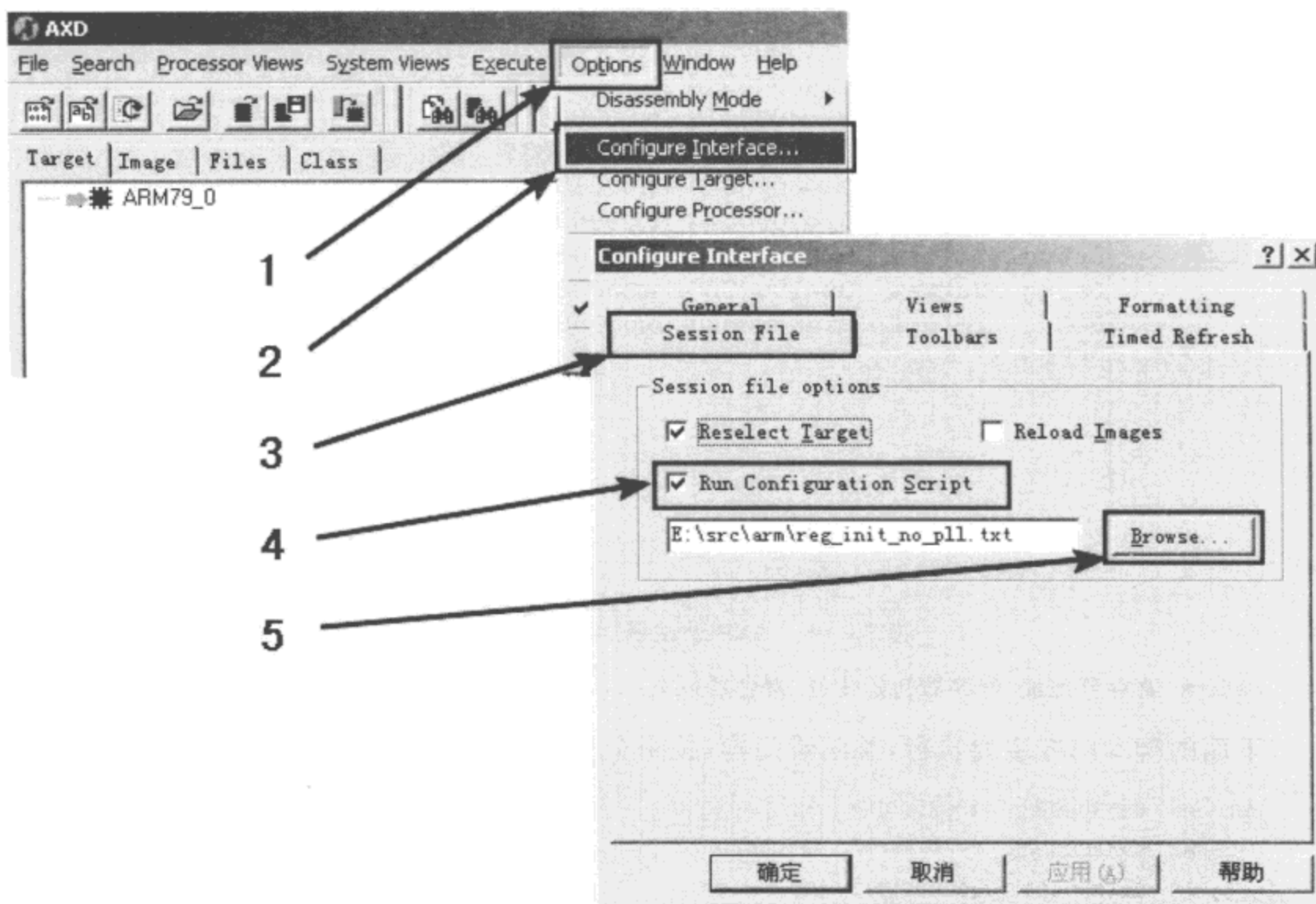


图 3-13 设置开发板初始化脚本

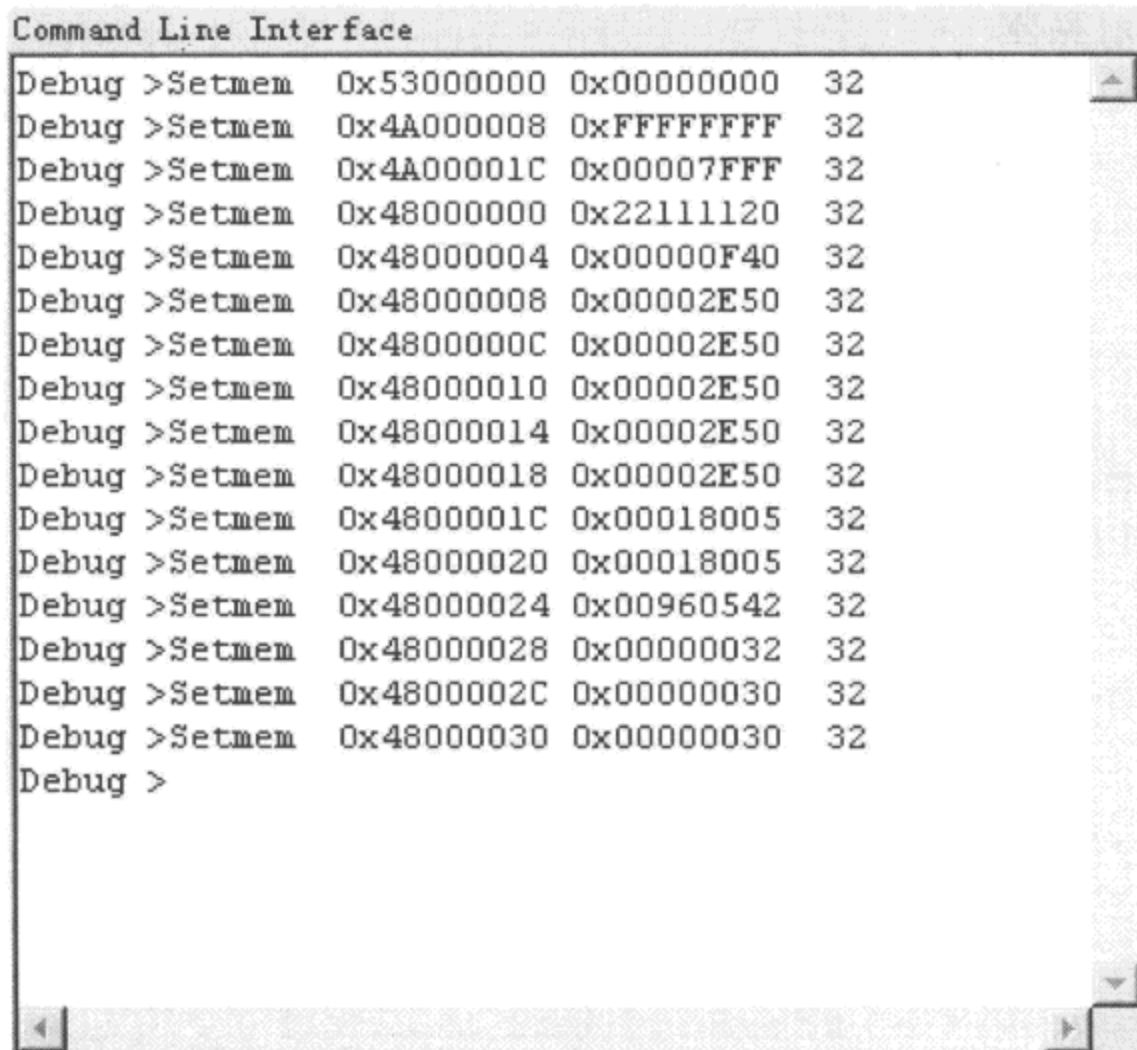
设置调试器初始化脚本

如图 3-14 所示,AXD 调试器初始化脚本设置完成后,再重启 AXD 调试环境,可以看到窗口 Command Line Interface 弹出脚本信息。

reg_init_no_pll.txt(随书光盘目录:\work\armarch\reg_init_no_pll.txt):

该文件是 AXD 调试器对目标开发板初始化配置脚本文件,为了保证开发板内程序正常执行,下面的脚本内容主要对目标开发板必要寄存器进行直接写入操作。

脚本格式:



```

Command Line Interface
Debug >Setmem 0x53000000 0x00000000 32
Debug >Setmem 0x4A000008 0xFFFFFFFF 32
Debug >Setmem 0x4A00001C 0x00007FFF 32
Debug >Setmem 0x48000000 0x22111120 32
Debug >Setmem 0x48000004 0x00000F40 32
Debug >Setmem 0x48000008 0x00002E50 32
Debug >Setmem 0x4800000C 0x00002E50 32
Debug >Setmem 0x48000010 0x00002E50 32
Debug >Setmem 0x48000014 0x00002E50 32
Debug >Setmem 0x48000018 0x00002E50 32
Debug >Setmem 0x4800001C 0x00018005 32
Debug >Setmem 0x48000020 0x00018005 32
Debug >Setmem 0x48000024 0x00960542 32
Debug >Setmem 0x48000028 0x00000032 32
Debug >Setmem 0x4800002C 0x00000030 32
Debug >Setmem 0x48000030 0x00000030 32
Debug >
  
```

图 3-14 AXD 调试器加载初始化脚本

Setmem 寄存器地址 寄存器初始化值 寄存器位宽

下面的脚本内容主要执行,关闭看门狗,关闭全部系统中断,初始化内存操作。

```

Setmem 0x53000000 0x00000000 32
Setmem 0x4A000008 0xFFFFFFFF 32
Setmem 0x4A00001C 0x00007FFF 32
Setmem 0x48000000 0x22111120 32
Setmem 0x48000004 0x00000F40 32
Setmem 0x48000008 0x00002E50 32
Setmem 0x4800000C 0x00002E50 32
Setmem 0x48000010 0x00002E50 32
Setmem 0x48000014 0x00002E50 32
Setmem 0x48000018 0x00002E50 32
Setmem 0x4800001C 0x00018005 32
Setmem 0x48000020 0x00018005 32
Setmem 0x48000024 0x00960542 32
Setmem 0x48000028 0x00000032 32
  
```

资源解密

PDG

```
Setmem 0x4800002C 0x00000030 32
Setmem 0x48000030 0x00000030 32
```

AXD 调试器会在启动时自动调用上面的配置脚本,初始化目标开发板相关寄存器。

运行该工程,可以在 ADS 调试器 AXD 下 ARM79_0-Console 窗口看到 helloworld 字样。该工作生成的 image 文件不能直接烧写到 Norflash 中运行,它只能在调试环境下运行。

helloworld.c

```
#include <stdio.h>
int main(){
    printf("hello world");
}
```

由于上述代码是纯 C 语言代码,而运行 C 程序必须要设置栈指针和堆指针,通过下面代码实现 C 程序堆栈空间的设置。

initstkhp.s

```
AREA initstkhp, CODE, READONLY
EXPORT __user_initial_stackheap      ; 导出__user_initial_stackheap 符号
__user_initial_stackheap            ; 堆栈设置入口函数
    LDR r0, = 0x31000000              ; 堆地址为 0x31000000
    LDR r1, = 0x33000000              ; 栈地址为 0x33000000
    mov PC, LR
END
```

其中__user_initial_stackheap 符号为 semihosting 程序运行时自动调用函数名,该函数通过 R0 和 R1 取得栈空间和堆空间地址。

Semihosting 控制 LED 灯实验(随书光盘:\work\armarch\led_semihosting):

该实验要加载 reg_init_no_pll.txt 脚本文件。

initstkhp.s:堆栈初始化函数

```
AREA initstkhp, CODE, READONLY
EXPORT __user_initial_stackheap      ; 导出__user_initial_stackheap 符号
__user_initial_stackheap            ; 堆栈设置入口函数
    LDR r0, = 0x31000000              ; 堆地址为 0x31000000
    LDR r1, = 0x33000000              ; 栈地址为 0x33000000
    mov PC, LR
END
```

main.c:

本程序首先初始化 LED 控制寄存器,使用标准 C 库里的 printf,scanf 分别在 AXD 终端

窗口打印信息和获得用户键盘输入,根据用户输入数字,点亮对应的 LED 灯。

```
#include <stdio.h>
#include <stdlib.h>

#define GPBCON      ( *(volatile unsigned long *)0x56000010)
#define GPBDAT      ( *(volatile unsigned long *)0x56000014)
#define LEDS        (1<<5|1<<6|1<<7|1<<8)

int main()
{
    int i;
    GPBCON = 0x00015400;
    while (1)
    {
        printf("please input led number: ");
        scanf("%d", &i);                // 取得用户输入
        switch (i)
        {
            case 1:                        // 输入 1, 点亮 LED1
                GPBDAT = (GPBDAT & (~LEDS)) | (1<<6|1<<7|1<<8);
                printf("led1 on\n");
                break;
            case 2:                        // 输入 2, 点亮 LED2
                GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<7|1<<8);
                printf("led2 on\n");
                break;
            case 3:                        // 输入 3, 点亮 LED3
                GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<6|1<<8);
                printf("led3 on\n");
                break;
            case 4:                        // 输入 4, 点亮 LED4
                GPBDAT = (GPBDAT & (~LEDS)) | (1<<5|1<<6|1<<7);
                printf("led4 on\n");
                break;
            default:
                printf("input error, please input 1 to 4\n");
                break;
        }
    }
}
```

}

总结:

Semihosting 半主机调试模式,只能使用在开发板和调试主机通过仿真器连接的情况下,也就是说脱离了主机调试环境上述代码不能正常运行。目标开发板上执行的 I/O 实际上是交给了远程主机来处理实现,正是因为如此,这种方式只适合在调试模式下,真正的嵌入式系统不可能依赖于主机实现 I/O 处理的,嵌入式系统要想独立出来实现 I/O 请求的处理,这就需要将输入/输出库函数的底层相关硬件实现重定向。

3.4.2 硬件重定向

由 semihosting 知识可知,semihosting 只是将目标系统中的 I/O 请求交给了调试环境来处理,但是在嵌入式系统实际应用中,往往嵌入式系统和主机调试环境是独立的,而嵌入式系统又想使用标准输入/输出中的库函数,这时就要使用硬件重定向技术。

应用程序中对外设的 I/O 请求实际是对底层最基本 I/O 硬件的封装,例如 printf() 函数,其实是对将数据写入到显示器相应寄存器的抽象封装,用户不用关心具体使用了什么硬件机制,也不用关心具体怎么将其打印到屏幕上。在 ADS 开发环境中,semihosting 低层也进行了封装。在嵌入式应用系统中,常常需要重新实现一些低级的 I/O 功能,以适应目标系统的具体情况。像这种将底层 I/O 由其他硬件来实现的重定向机制称做硬件重定向。如图 3-15 所示。

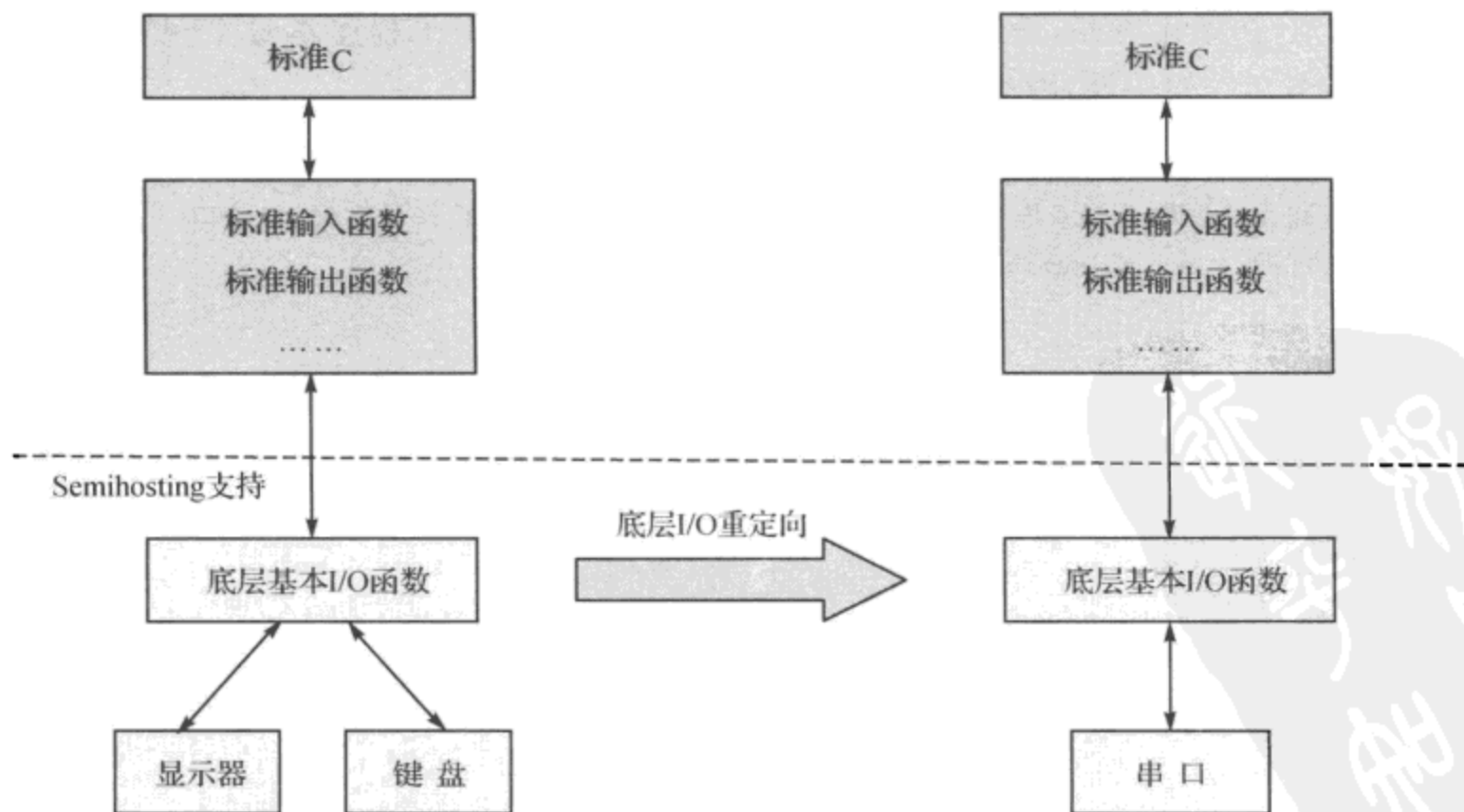


图 3-15 底层 I/O 重定向

Semihosting 将底层基本 I/O 函数进行了封装, 默认的 semihosting 模式下, 底层基本 I/O 都是针对显示器, 键盘等硬件进行了封装, 实现了对显示器, 键盘输入等硬件的驱动, I/O 操作等。用户可以自己定义底层的基本 I/O 函数, 来实现目标开发板上 I/O 硬件的驱动和 I/O 操作, 并且告之连接器, 在连接程序时连接用户自己定义底层基本 I/O 函数, 而不是默认的调试环境下的底层基本 I/O 函数。这样目标开发板运行应用程序中的 I/O 操作就被重新定向到了自己的硬件上了。例如: 标准输出函数 `printf()` 的底层基本 I/O 函数是 `fputc()`, 它是向硬件里写入一个字符函数, 用户自己将该函数重写, 在 `fputc()` 里面实现向 UART 串口打印字符, 而不是打印到标准输出显示器上。这样一来, `printf()` 系列函数的输出都被重定向到 UART 串口上去了。

实现硬件重定向时有以下几点需要注意:

- (1) 声明不使用 semihosting SWI 来请求 host 主机 I/O 操作, 而是使用自定义 I/O 操作。
- (2) 驱动重定向硬件设备。
- (3) 重写低级 I/O 函数。

硬件重定向实验

本实验源码存放在光盘目录 \work\armarch\ led_retargert 目录下。

init. s;

该程序文件主要用于启动处理, 声明不使用 semihosting SWI 来请求 host 主机 I/O 操作, 关闭看门狗, 初始化内存, 最后跳入到 main 函数中执行。由于程序本身实现了 `reg_init_no_pll.txt` 脚本文件的功能, 因此本实验不需要加载初始化脚本。

```
AREA Init, CODE, READONLY
```

```
; 确保不使用系统 C 库中的底层 I/O 函数接口, 而是使用用户自己定义 IO 接口
```

```
IMPORT __use_no_semihosting_swi
```

```
ENTRY
```

```
EXPORT Reset_Handler
```

```
Reset_Handler
```

```
; 关闭看门狗
```

```
ldr r0, = 0x53000000
```

```
mov r1, #0
```

```
str r1, [r0]
```

```
bl initmem
```

```
IMPORT __main
```



```

        B      __main      ; 使用 B 指令跳入 main,而不使用 BL 指令,因为不需要返回
initmem
        ldr r0, = 0x48000000
        ldr r1, = 0x48000034
        adr r2, memdata
initmemloop
        ldr r3, [r2], #4
        str r3, [r0], #4
        teq r0, r1
        bne initmemloop
        mov pc,lr

memdata
        DCD      0x22111110      ;BWSCON
        DCD      0x00000700      ;BANKCON0
        DCD      0x00000700      ;BANKCON1
        DCD      0x00000700      ;BANKCON2
        DCD      0x00000700      ;BANKCON3
        DCD      0x00000700      ;BANKCON4
        DCD      0x00000700      ;BANKCON5
        DCD      0x00018005      ;BANKCON6
        DCD      0x00018005      ;BANKCON7
        DCD      0x008e07a3      ;REFRESH
        DCD      0x000000b1      ;BANKSIZE
        DCD      0x00000030      ;MRSRB6
        DCD      0x00000030      ;MRSRB7
        END

```

serial.c:

该程序文件主要实现了对 UART 串口的驱动和基本 I/O 操作。由于 I/O 请求硬件重定向最终要实现数据的输出和输入,通过读取 UART 串口寄存器读取串口数据取得用户输入信息,通过写入 UART 串口寄存器实现数据输出操作。

```

#include "s3c2410.h"

#define TXDREADY    (1<<2)
#define RXDREADY    (1)

void init_serial_A( )
{

```

```
//初始化 UART
GPHCON |= 0xa0;           //GPH2,GPH3 used as TXD0,RXD0
GPHUP    = 0x0c;          //GPH2,GPH3 内部上拉

ULCON0    = 0x03;         //8N1
UCON0     = 0x05;         //查询方式
UFCON0    = 0x00;         //不使用 FIFO
UMCON0    = 0x00;         //不使用流控
UBRDIV0   = 12;           //波特率为 57600
}

/* 通过串口字符发送函数 */
void sendchar2(char c)
{
    while( !(UTRSTAT0 & TXD0READY) );
    UTXH0 = c;
}

/* 通过串口字符接收函数 */
char recvchar( )
{
    while( !(UTRSTAT0 & RXD0READY) );
    return URXH0;
}

```

retarget.c;

本程序文件主要重新实现了底层 I/O 的基本函数:

(1) int fgetc(FILE *f): 从文件描述符 f 中取得单个字符输入, scanf 的底层实现函数, 在 C 语言中所有的设备都被抽象成一个可以读写的文件, f 参数就是具体 I/O 设备, 如 stdout 标准输出指显示器, stdin 标准输入指键盘, 由于本实验用串口重定向了标准输入, 因此 fgetc 的功能主要返回从串口取得一个字符。

(2) int fputc(int ch, FILE *f): 向文件描述符 f 里写入一个字符 ch, 它是 printf 的底层实现函数, 本实验是用串口重定向标准输出, 因此 fputc 的功能主要是向串口里写入字符 ch。

(3) int ferror(FILE *f): 标准错误的底层实现函数, 本实验直接返回 EOF, 没有实现具体功能。

(4) void _ttywrch(int ch): 终端数据输出的底层实现, 本实验里用串口实现其功能, 同样是向串口里写入字符 ch。

(5) void _sys_exit(int return_code): 系统退出的底层实现, 这儿直接是用死循环来模拟

最后程序的退出。

由于底层 I/O 操作实现被重定向,基于这些底层 I/O 的库也没有被引入, __FILE, __stdout, __stdin 这些使用的结构体和变量就需要自己重新定义。其中 __FILE 就是 FILE 文件描述符。 __stdout 是标准输出文件描述符, __stdin 是标准输入文件描述符。

```
#include <stdio.h>

extern void sendchar2( char ch );
extern char recvchar(void);

struct __FILE { int handle; };
FILE __stdout;
FILE __stdin;

/* 底层 I/O 函数,从串口取得一个字符 */
int fgetc(FILE * f)
{
    char ch;
    ch = recvchar();
    if ((ch == '\r') || (ch == '\n'))           // 接收到返回行首符与换行符时实现换行显示
    {
        fputc('\n', NULL);
        fputc('\r', NULL);
    }
    fputc(ch, NULL);
    return ch;
}

/* 底层 I/O 函数,打印一个字符,将字符打印到串口 */
int fputc(int ch, FILE * f)
{
    char tempch = ch;
    if (ch == '\n')           // 发送换行符时,先发送返回行首符号\r,再发送换行符
        sendchar2('\r');
    sendchar2(tempch);
    return ch;
}
```

```

/* 底层 I/O 函数 */
int ferror(FILE *f)
{
    return EOF;
}

/* 底层 I/O 函数,终端打印函数 */
void _ttywrch(int ch)
{
    char tempch = ch;
    sendchar2( tempch );
}

/* 底层 I/O 函数,程序退出处理函数 */
void _sys_exit(int return_code)
{
    label: goto label;
}

```

main.c:

本程序文件主要调用串口驱动程序、驱动串口、驱动 LED,获得用户输入点亮对应 LED 灯。

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
extern void init_serial_A(void);
#define GPBCON      ( *(volatile unsigned long *)0x56000010)
#define GPBDAT      ( *(volatile unsigned long *)0x56000014)
#define LEDS      (1<<5|1<<6|1<<7|1<<8)

int main(void)
{
    int a,b;
    int i;
    float c,d;
    void *p1, *p2;
    #pragma import(__use_no_semihosting_swi) // 不使用软件中断响应 semihosting 请求

    init_serial_A();

```



```
GPBCON      = 0x00015400;
while (1)
{
    printf("please input led number: ");
    scanf("%d", &i);
    switch (i)
    {
        case 1:
            GPBDAT = (GPBDAT & (~LEDS)) | (1 << 6 | 1 << 7 | 1 << 8);
            printf("led1 on \n");
            break;
        case 2:
            GPBDAT = (GPBDAT & (~LEDS)) | (1 << 5 | 1 << 7 | 1 << 8);
            printf("led2 on \n");
            break;
        case 3:
            GPBDAT = (GPBDAT & (~LEDS)) | (1 << 5 | 1 << 6 | 1 << 8);
            printf("led3 on \n");
            break;
        case 4:
            GPBDAT = (GPBDAT & (~LEDS)) | (1 << 5 | 1 << 6 | 1 << 7);
            printf("led4 on\n");
            break;
        default:
            printf("input error, please input 1 to 4\n");
            break;
    }
}
return 0;
}
```

3.5 系统调用与软件中断 SWI 的实现

3.5.1 系统调用

操作系统的主要功能是为应用程序的运行创建良好的环境,保障每个程序都可以最大化利用硬件资源,防止非法程序破坏其他应用程序执行环境,为了达到这个目的,操作系统会将

硬件的操作权限交给内核来管理,用户程序不能随意使用硬件,使用硬件(对硬件寄存器进行读写)时,要先向操作系统发出请求,操作系统内核帮助用户程序实现其操作,也就是说用户程序不会直接操作硬件,而是提供给用户程序一些具备预定功能的内核函数,通过一组称为系统调用的(system call)的接口呈现给用户,系统调用把应用程序的请求传给内核,调用相应的内核函数完成所需的处理,将处理结果返回给应用程序。这好比我们去银行取款,用户自己的银行账户不可能随意操作,必须要有一个安全的操作流程和规范,银行里的布局通常被分成两部分,中间用透明玻璃分隔开,只留一个小窗口,面向用户的是用户服务区,工作人员所在区域为内部业务操作区,取款时,将银行卡或存折通过小窗口交给业务员,并且告诉他要取多少钱,具体取钱的操作你是不会直接接触的,业务员会将银行账户里减掉取款金额,将现金给你。上述操作流程可以很好保护银行系统,银行系统的操作全部由业务员来实现,用户只能向业务员提出自己的服务请求。银行里的小窗口就类似与操作系统的系统调用接口,是将用户请求传递给内核的接口,如图 3-16 所示。

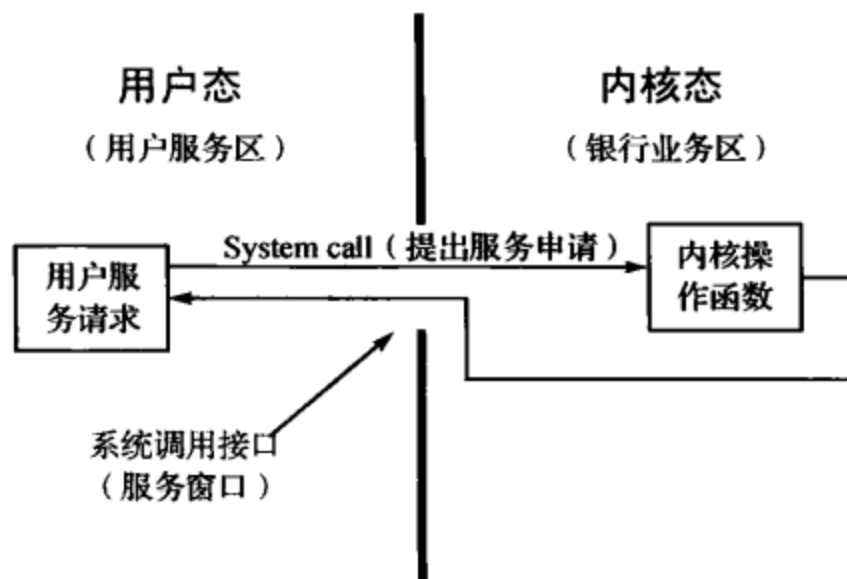


图 3-16 系统调用接口示意图

操作系统里将用户程序运行在用户模式下,并且为其分配可以使用内存空间,其他内存空间不能访问,内核态运行在特权模式下,对系统所有硬件进行统一管理和控制。从前面所学知识可以了解到,用户模式下没有权限进行模式切换,这也就意味着用户程序不可能直接通过切换模式去访问硬件寄存器,如果用户程序试图访问没有权限的硬件,会产生异常。这样用户程序被限制起来,如果用户程序想要使用硬件时怎么办呢? 用户程序使用硬件时,必须调用操作系统提供的 API 接口才可以,而操作系统 API 接口通过软件中断方式切换到管理模式下,实现从用户模式下进入特权模式。

3.5.2 软件中断

软件中断是利用硬件中断的概念,用软件方式进行模拟,实现从用户模式切换到特权模式

并执行特权程序的机制。

硬件中断是由电平的物理特性决定,在电平变化时引发中断操作,而软件中断是通过一条具体指令 SWI,引发中断操作,也就是说用户程序里可以通过写入 SWI 指令来切换到特权模式,当 CPU 执行到 SWI 指令时会从用户模式切换到管理模式下,执行软件中断处理。由于 SWI 指令由操作系统提供的 API 封装起来,并且软件中断处理程序也是操作系统编写者提前写好的,因此用户程序调用 API 时就是将操作权限交给了操作系统,所以用户程序还是不能随意访问硬件。

先来了解下 SWI 指令。

SWI 软件中断号 `immed_24`

软件中断指令相对比较简单,只有一个操作数: `immed_24`, SWI 指令编码格式如图 3-17 所示。

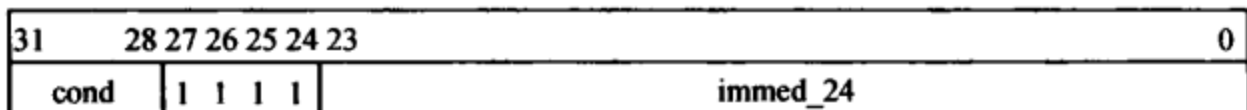


图 3-17 SWI 指令编码格式

SWI 指令编码中 `immed_24` 为 24 位任意有效立即数(范围 $0 \sim 2^{24} - 1$), 当该指令被执行时系统产生软件中断异常,切换到管理模式下。用户程序切换到管理模式下后,进入到软件中断处理程序,通常软件中断异常处理程序都是系统开发人员提前写好的,SWI 切换到了特权模式,执行的是系统开发人员写好的异常处理程序,只要该处理程序没有问题,那么用户程序还是不能为所欲为的。

SWI 指令后面的 24 立即数是干什么用的呢? 用户程序通过 SWI 指令切换到特权模式,进入软件中断处理程序,但是软件中断处理程序不知道用户程序到底想要做什么? SWI 指令后面的 24 位用来做用户程序和软件中断处理程序之间的接头暗号。通过该软件中断立即数来区分用户不同操作,执行不同内核函数。如果用户程序调用系统调用时传递参数,根据 ATPCSC 语言与汇编混合编程规则将参数放入 `R0~R3` 即可。下面的例子通过系统调用函数 `int led_on(int led_no)` 实现点亮第 `led_no` 个 LED 灯,由于 C 语言里没有 SWI 指令对应的语句,因此这儿要用到 C 语言与汇编混合编程, `led_on` 函数里将参数 `led_no` 的值传递给 `R0`, 通过软件中断 SWI 指令切换到软件中断管理模式,同时 `R0` 软件中断方式点亮 LED 灯,用户通过 SWI #1 指令可以点灯,具体点亮哪个灯,通过 `R0` 保存参数传递,如果亮灯成功返回对应 LED 号。

系统调用接口函数 `led_on`:

```
#define __led_on_swi_no 1 // 软件中断号 1,调用管理模式下的 do_led_on 函数
int led_on(int led_no){
```

```

        int ret;                                // 返回值
        __asm{                                  // 由于 C 程序中没有 SWI 对应表达式,所以
                                                // 使用混合编程

        mov    r0, led_no                      // 根据 ATPCS 规则, r0 存放第一个参数
        swi     __led_on_swi_no                // 产生 SWI 软中断, 中断号为 __led_on_swi_no
        mov    ret, r0                         // 软件中断处理结束, 取得中断处理返回值,
                                                // 传递给 ret 变量

    }
    return ret;                                // 将 ret 返回给调用 led_on 的语句
}

```

3.5.3 软件中断处理

CPU 执行到 `swi xxx` 执行后, 产生软件中断, 由异常处理部分知识可知, 软件中断产生后 CPU 将强制将 PC 的值置为异常向量表地址 `0x08`, 在异常向量表 `0x08` 处安放跳转指令 `b HandleSWI`, 这样 CPU 就跳往我们自己定义的 `HandleSWI` 处执行。

首先, 软件中断处理中通过 `STMFD SP!, {R0-R12, LR}` 要保存程序执行现场, 将 `R0~R12` 通用寄存器数据保存在管理模式下的 `SP` 栈内, `LR` 由硬件自动保存软件中断指令下一条指令的地址(后面利用 `LR` 的地址取得 `SWI` 指令编码), 该寄存器值也保存在 `SP` 栈内, 将来处理完毕之后返回。由 `SWI` 指令编码知识可知, `SWI` 指令低 24 位保存有软件中断号, 通过 `LDR R4, [LR, #-4]` 指令, 取得 `SWI` 指令编码(`LR` 为硬件自动保存 `SWI xxx` 指令的下一条指令地址, `LR - 4` 就是 `SWI` 指令地址), 将其保存在 `R4` 寄存器中。通过 `BIC R4, R4, #0xFF000000` 指令将 `SWI` 指令高 8 位清除掉, 只保留低 24 位立即数, 再根据 24 位立即数中的软件中断号判断用户程序的请求操作。如果 24 位立即数为 1, 表示 `led_on` 系统调用产生的软件中断, 则在管理模式下调用对应的亮灯操作 `do_led_on`。如果 24 位立即数为 2, 表示 `led_off` 系统调用产生的软件中断, 则调用灭灯操作 `do_led_on`, 根据 ATPCS 调用规则, `R0~R3` 作为参数传递寄存器, 在软件中断处理中没有使用这 4 个寄存器, 而是使用 `R4` 作为操作寄存器的。执行完系统调用操作之后, 返回到 `swi_return` (在调用对应系统操作时, 通过 `LDREQ LR, =swi_return` 设置了返回地址), 执行返回处理, 通过 `LDMIA SP!, {R0-R12, PC}` 指令将用户寄存器数据恢复到 `R0~R12`, 将进入软件中断处理时保存的返回地址 `LR` 的值恢复给 `PC`, 实现程序返回, 同时还恢复了状态寄存器。切换回用户模式下程序中继续执行。

```

; 异常向量表开始
; 0x00: 复位 Reset 异常
b    Reset

```

```

; 0x04: 未定义异常(未处理)
HandleUndef
    b    HandleUndef

; 0x08: 软件中断异常,跳往软件中断处理函数 HandleSWI
    b    HandleSWI
    ... ..
; 省略其他异常向量和对应处理
    ... ..

; *****
; 软件中断处理
; *****
IMPORT do_led_on
IMPORT do_led_off

HandleSWI
    STMFDB    SP!, {R0 - R12, LR}    ; 保存程序执行现场
    LDR R4, [LR, #-4]                ; LR - 4 为指令" swi xxx" 的地址,低 24 位是软件中断号
    BIC      R4, R4, #0xFF000000      ; 取得 ARM 指令 24 位立即数

    CMP      R4, #1                  ; 判断 24 位立即数,如果为 1,调用 do_led_on 系统调用
    LDREQ    LR, = swi_return         ; 软件中断处理返回地址
    LDREQ    PC, = do_led_on          ; 软件中断号 1 对应系统调用处理

    CMP      R4, #2                  ; 判断 24 位立即数,如果为 2,调用 do_led_off 系统调用
    LDREQ    LR, = swi_return         ; 软件中断处理返回地址
    LDREQ    PC, = do_led_off         ; 软件中断号 2 对应系统调用处理

    MOVNE    R0, #-1                 ; 没有该软件中断号对应函数,出错返回 -1

swi_return
    LDMIA    SP!, {R0 - R12, PC}^    ; 中断返回,^表示将 spsr 的值复制到 cpsr

```

3.5.4 LED 系统调用实验

本实验源码存放在光盘目录\work\armarch\swi_led 工程目录下。本实验通过 LED 跑马灯效果来模拟系统调用,本程序提供了两个系统调用接口 led_on 和 led_off,用户程序 main.c 通过引入头文件 led.h 使用系统调用接口,用户调用 led_on 和 led_off 时通过软件中断指令切换到管理模式,在管理模式下调用内核 LED 操作系统 do_led_on 和 do_led_off,实现 LED 的亮灭。实验源码适用于 QQ2440, TQ2440, MINI2440 开发板。

head.s:

本程序文件主要用于安装异常向量表, Reset 异常处理, 软件中断处理和必要硬件初始化。

```

;*****
; 系统调用实验(QQ2440, MINI2440, TQ2440)
;*****
GPBCON    EQU        0x56000010
GPBDAT    EQU        0x56000014
SYS_STACK_BASE EQU    0x33000000
EXPORT    SWI_LED
AREA      SWI_LED ,CODE,READONLY
ENTRY

;*****
; 设置中断向量,除了 Reset 和 HandleSWI 外,其他异常都没有使用(如果不幸发生了,
; 将导致死机)
;*****
; 0x00: 复位 Reset 异常
b        Reset

; 0x04: 未定义异常(未处理)
HandleUndef
b        HandleUndef

; 0x08: 软件中断异常,跳往软件中断处理函数 HandleSWI
b        HandleSWI

; 0x0c: 指令预取异常(未处理)
HandlePrefetchAbt
b        HandlePrefetchAbt

; 0x10: 数据访问中止异常(未处理)
HandleDataAbt
b        HandleDataAbt

; 0x14: 未使用异常(未处理)
HandleNotUsed
b        HandleNotUsed

; 0x18: 一般中断异常(未处理)

```

HandleIRQ

b HandleIRQ

; 0x1c: 快速中断异常(未处理)

HandleFIQ

b HandleFIQ

Reset

; 关闭看门狗

ldr r0, = 0x53000000

mov r1, #0

str r1, [r0]

bl initmem

ldr sp, = 0x32000000

; LED 灯初始化

ldr r0, = GPBCON

ldr r1, = 0x00015400

str r1, [r0]

ldr r0, = GPBDAT

ldr r1, = 0x1e0

str r1, [r0]

msr cpsr_c, # 0xdf

ldr sp, = SYS_STACK_BASE

msr cpsr_c, # 0x50

ldr lr, = halt_loop

IMPORT xmain

ldr pc, = xmain

halt_loop

b halt_loop

; 复位异常处理入口

; 设置管理模式栈指针

; LED 的 GPIO 接口配置寄存器

; GPIO 配置数据

; 设置 GPIO

; LED 数据寄存器

; 熄灭所有 LED

; 开启系统中断, 进入用户模式, 该指令执行完

; 就进入用户空间, 执行用户程序 xmain

; 设置管理模式下返回地址

; 跳入主函数 main 里执行

```

;*****
; 软件中断处理
;*****

IMPORT do_led_on
IMPORT do_led_off

HandleSWI
    STMFD    SP!, {R0 - R12, LR}    ; 保存程序执行现场
    LDR      R4, [LR, # - 4]         ; LR - 4 为指令“swi xxx”的地址,指令低 24 位软件中断号
    BIC      R4, R4, # 0xFF000000    ; 取得 ARM 指令 24 位立即数

    CMP      R4, # 1                ; 判断 24 位立即数的值,如果为 1,调用 do_led_on 系统调用
    LDREQ    LR, = swi_return        ; 软件中断处理返回地址
    LDREQ    PC, = do_led_on         ; 软件中断号 1 对应系统调用处理

    CMP      R4, # 2                ; 判断 24 位立即数的值,如果为 2,调用 do_led_off 系统调用
    LDREQ    LR, = swi_return        ; 软件中断处理返回地址
    LDREQ    PC, = do_led_off        ; 软件中断号 2 对应系统调用处理

    MOVNE    R0, # - 1              ; 没有该软件中断号对应函数,出错返回 - 1

swi_return
    LDMIA    SP!, {R0 - R12, PC}~    ; 中断返回,~表示将 spsr 的值复制到 cpsr

initmem
    ldr      r0, = 0x48000000        ; 内存控制寄存器起始地址
    ldr      r1, = 0x48000034        ; 内存控制寄存器结束地址
    adr      r2, memdata            ; 加载寄存器设置数据区首地址

initmemloop
    ldr      r3, [r2], # 4
    str      r3, [r0], # 4
    teq      r0, r1
    bne      initmemloop            ; 循环设置每一个寄存器
    mov      pc, lr

memdata
    DCD      0x22000000              ; BWSCON
    DCD      0x00000700              ; BANKCON0
    DCD      0x00000700              ; BANKCON1
    DCD      0x00000700              ; BANKCON2

```

```

DCD    0x00000700    ;BANKCON3
DCD    0x00000700    ;BANKCON4
DCD    0x00000700    ;BANKCON5
DCD    0x00018005    ;BANKCON6
DCD    0x00018005    ;BANKCON7
DCD    0x008e07a3    ;REFRESH
DCD    0x000000b1    ;BANKSIZE
DCD    0x00000030    ;MRSRB6
DCD    0x00000030    ;MRSRB7

```

```

END                ; 代码结束

```

main.c:

本程序文件是用户程序 xmain, 主要实现跑马灯效果, 通过使用系统调用 LED_on, led_off 实现 LED 控制。

```

#include "led.h"

/* 亮灯延时 */
void delay(int msec)
{
    int i, j;
    for(i = 1000; i > 0; i--)
        for(j = msec * 10; j > 0; j--)
            /* do nothing */;
}

/* 主函数跑马灯效果 */
int xmain()
{
    while(1)
    {
        led_on(1);
        delay(5);        //delay

        led_on(2);
        delay(5);        //delay

        led_on(3);
        delay(5);        //delay
    }
}

```

蘇子知覺
PDG

```

        led_on(4);
        delay(5);          //delay

        led_off(1);
        delay(5);          //delay

        led_off(2);
        delay(5);          //delay

        led_off(3);
        delay(5);          //delay

        led_off(4);
        delay(5);          //delay
    }
    return 0;
}

```

led_lib.c

本程序文件是系统调用函数 led_on, led_off 的具体实现,通过 swi 软件中断提交硬件访问请求,将具体请求以软件中断号的方式通过参数传递给内核空间。

```

#include "led.h"

#define __led_on_swi_no    1    // 软件中断号 1,调用管理模式下的 do_led_on 函数
#define __led_off_swi_no  2    // 软件中断号 2,调用管理模式下的 do_led_off 函数

int led_on(int led_no){
    int ret;                // 返回值
    __asm{                  // 由于 C 程序中没有 SWI 对应表达式,所以使用混合编程
        mov     r0, led_no  // 根据 ATPCS 规则,r0 存放第一个参数
        swi     __led_on_swi_no // 产生 SWI 软件中断,中断号为 __led_on_swi_no
        mov     ret, r0      // 软件中断处理结束,取得中断处理返回值,传递给 ret 变量
    }
    return ret;             // 将 ret 返回给调用 led_on 的语句
}

int led_off(int led_no){

```



```

int ret;                // 返回值
__asm{                  // 由于 C 程序中没有 SWI 对应表达式, 所以使用混合编程
    mov    r0, led_no    // 根据 ATPCS 规则, r0 存放第一个参数
    swi     __led_off_swi_no // 产生 SWI 软件中断, 中断号为 __led_off_swi_no
    mov    ret, r0       // 软件中断处理结束, 取得中断处理返回值, 传递给 ret 变量
}
return ret;             // 将 ret 返回给调用 led_off 的语句
}

```

led.h:

LED 系统调用头文件。

```

extern int led_on(int num);
extern int led_off(int num);

```

sys_call.c:

本程序文件主要是系统调用接口内核空间 do_led_on, do_led_off 函数的实现。

```

#include "register.h"
/* Led1~Led4 初始化 */
#define LED1    (1<<5)        //LED1 GPBDAT[5]
#define LED2    (1<<6)        //LED2 GPBDAT[6]
#define LED3    (1<<7)        //LED3 GPBDAT[7]
#define LED4    (1<<8)        //LED4 GPBDAT[8]

/* 点亮对应 num 号 Led */
extern int do_led_on (int num)
{
    switch(num)
    {
        case 1;
            GPBDAT = GPBDAT & ~LED1; break;
        case 2;
            GPBDAT = GPBDAT & ~LED2; break;
        case 3;
            GPBDAT = GPBDAT & ~LED3; break;
        case 4;
            GPBDAT = GPBDAT & ~LED4; break;
        default;
            return 0;
    }
}

```

新华书店
PDG

```
        return num;
    }

    /* 关闭对应 num 号 Led */
    extern int do_led_off(int num)
    {
        switch(num)
        {
            case 1:
                GPBDAT = GPBDAT | LED1; break;
            case 2:
                GPBDAT = GPBDAT | LED2; break;
            case 3:
                GPBDAT = GPBDAT | LED3; break;
            case 4:
                GPBDAT = GPBDAT | LED4; break;
            default:
                return 0;
        }
        return num;
    }
}
```

3.6 进程切换的实现

从本节起开始介绍关于第 3.8 节-小型多任务操作系统 miniOS 的实现相关知识,该部分不属于嵌入式底层开发,而是属于操作系统知识点,这些知识点是围绕 miniOS 的实现而介绍,主要目的是将底层硬件相关知识点和上层操作系统相关知识接合起来帮助大家更好理解整个嵌入式系统结构。

3.6.1 进 程

我们平时使用计算机时,一边开着浏览器浏览网页,一边打开播放器听着歌曲,这些都是运行程序,操作系统帮我们对这些程序进行管理和监督,同时操作系统还管理着全部系统资源,可见操作系统可以同时运行多个程序,执行多个任务,像 Windows、Linux、UNIX 这些操作系统,它们都支持同时运行多个任务程序,实现不同的功能,还可以同时让多个用户登录使用同一个操作系统,这是多用户多任务操作系统的典型特点。在现代操作系统里,任务通常以进程为表现形式。

进程是操作系统一个任务运行的基础,是一个正在执行的程序,是一个活动的实体,并不是磁盘上的程序,它表示当前 CPU 的执行状态和一组相关的系统资源所描述的活动单元。例如正在打开的音乐播放器是一个进程,它在不停地读取音乐文件,播放出歌曲。

进程不仅仅是程序在内存中的执行代码,它还包含当前进程运行的状态和相关使用的寄存器等。

如图 3-18 所示,每个进程都有一个运行状态来表示当前进程当前的运行情况,每个进程可以包含下列状态之一:

- (1) 新建状态:进程正在新建,分配系统资源。
- (2) 运行状态:当前进程正在被 CPU 执行。
- (3) 等待状态:进程自己主动或被动进入等待状态,例如,进程自己主动进入睡眠状态,或等待某个 I/O 资源或某个事件的产生而被动进入等待状态。
- (4) 就绪状态:当前进程所需资源已经准备好,等待 CPU 调度执行。
- (5) 终止状态:进程已经执行完成。

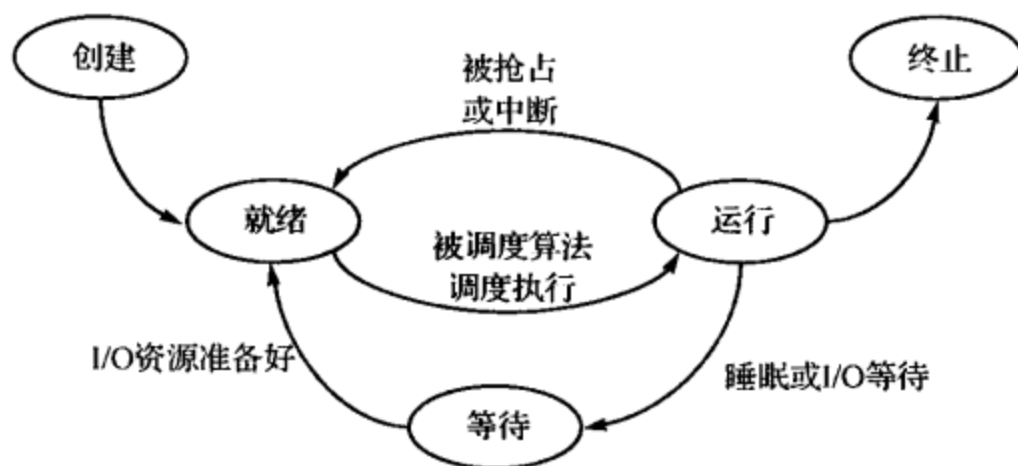


图 3-18 进程状态图

每一个进程在一个时刻只能存在一种状态,在单核 CPU 上一个时刻最多只能有一个进程处于运行状态,可以有多个进程处于就绪状态、等待状态。

3.6.2 进程控制块 PCB

在操作系统内,对每一个进程进行管理的数据结构称做进程控制块(Process Control Block, PCB),它主要描述当前进程状态和进程正在使用的资源。

PCB 主要包括以下内容:

- (1) 进程 ID:用来标识进程。
- (2) 进程状态:描述当前进程运行状态。
- (3) 进程调度信息:包含进程优先级、当前进程可用时间片和调度有关信息。
- (4) 进程使用寄存器:包含当前进程使用的操作寄存器、状态寄存器和栈指针寄存器等。

第3章 ARM 体系结构

(5) 进程 I/O 信息:当前进程正在使用的硬件资源,打开的文件等。

从本章节起,开始以 miniOS 代码为依据介绍小型多任务操作系统相关知识,下面代码是 miniOS 的 PCB 结构体:

```
typedef struct task_struct
{
    int    pid;           // 进程 ID
    unsigned long state;  // 进程状态
    unsigned long count;  // 进程时间片数
    unsigned long timer;  // 进程休眠时间
    unsigned long priority; // 进程默认优先级
    unsigned long content[20]; // 进程执行现场保存区
} PCB;
```

miniOS 采用直接从硬盘读取应用程序并启动进程的简单方式,而不是像 Windows, Linux 操作系统一样,启动进程时通过文件系统来寻找程序文件,再加载到内存中,因此 miniOS 没有使用对文件进程管理的文件系统,所以相关的 I/O 信息在 PCB 中没有体现。

3.6.3 进程创建

一个进程要想被执行,完成特定任务,首先要被创建。通常,进程需要一定的系统资源,如 CPU 时间片、内存空间、操作文件、硬件设备等,因此进程创建通常包含以下操作:

- (1) 初始化当前进程 PCB:分配有效进程 ID,设置进程优先级和 CPU 时间片。
- (2) 为进程准备内存执行空间:为进程分配内存空间。
- (3) 加载任务程序到内存空间:将进程代码从用户空间复制到内核空间。
- (4) 设置进程执行状态为就绪态:准备好所运行需资源,等待 CPU 调度执行。

由于操作系统可以同时运行多个进程,每个进程 PCB 描述了一个进程的所有相关信息,为了方便操作系统对每个进程进行管理,通常将进程 PCB 放到进程队列中。

进程的启动是由用户的操作引起的,比如在命令行下面,输入一个执行命令,pwd (打印出当前的绝对路径),那么这时,命令行解释程序(windows 下是 cmd.exe,linux 下是 shell)会将“pwd”这个字符串进行解释,将“pwd”作为一个程序名,在环境变量 PATH 设置的路径中,通过文件系统去硬盘等外部存储介质中寻找所有以“pwd”为名字的应用程序,如果没有找到,提示用户,是一个非法命令,如果找到,在内核空间为其分配 PCB 结构体,加载到内存中其对应地址空间,启动它作为一个进程,如果命令有参数的话,当然在启动之前还要将命令的参数复制到进程地址空间中。

3.6.4 进程队列

PCB 是在内核空间描述每一个进程的数据结构,当前操作系统内有多个进程时,也就是

意味着当前操作系统内存在多个 PCB 结构体,每个 PCB 对应用户地址空间里的一个进程。操作系统内核为了方便对所有进程进行管理,进程的 PCB 通常被放到队列里面,新创建进程放入队尾,当进程执行完成后进入终止状态从队列中剔除。引入进程队列也是为了方便操作系统调度程序进行进程调度,如图 3-19 所示。

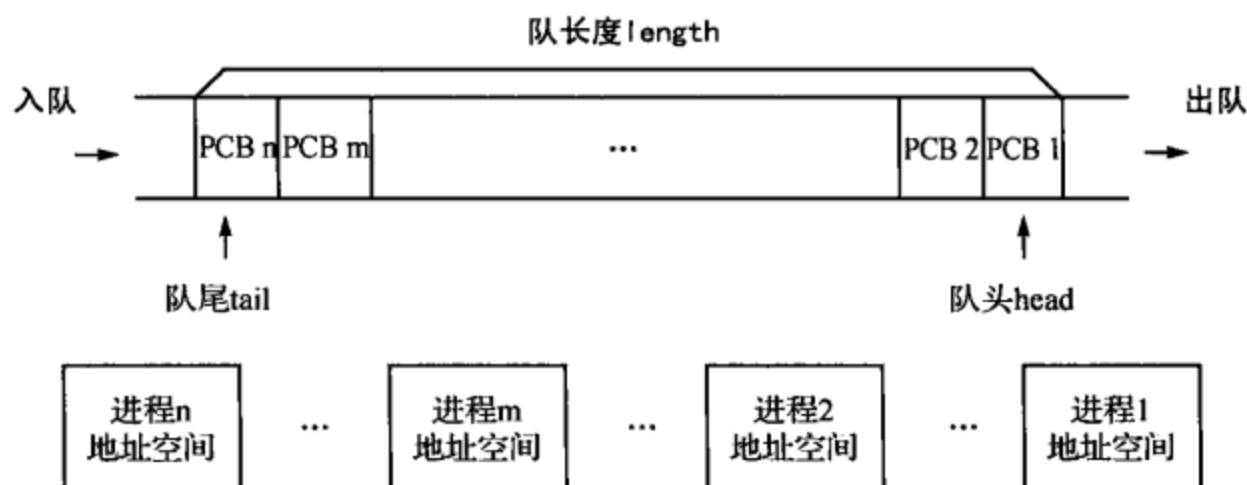


图 3-19 进程队列

进程被创建后,被插入当前进程队列中。该队列包含当前系统中的所有进程 PCB。通常操作系统中存在多个队列,处于就绪状态的进程放在就绪队列中,对于 I/O 设备也会有对应进程队列等待该硬件 I/O 的访问。通常队列是以动态链表形式封装对应结构体的 PCB。

3.6.5 进程调度

无论是在实时操作系统还是分时操作系统中,用户进程数一般都多于 CPU 个数,这将导致它们互相争夺 CPU 处理器来执行。另外,系统进程也同样需要使用 CPU。这就要求进程要按照一定的策略去被调度执行,让每个进程都能够取得 CPU 的执行权,来增加系统实时性和交互性,进程被调度时机通常出现在以下情况:

- 当前运行进程执行完毕时。
- 当前进程进入睡眠状态时(调用 sleep())。
- 当前进程等待请求资源时(例如等待 UART 串口接收数据等)。
- 当前进程时间片用完时或被优先级更高进程对当前进程抢占。

1. 进程调度时机

其中进程执行完毕时要退出,通过系统调用接口 _exit() 和软件中断指令切换到内核空间,将进程使用资源和内核中保存的 PCB 释放掉。程序通过调用 sleep() 自己主动进入睡眠状态,这时也是通过软件中断方式进入到内核的特权模式下,将当前进程挂起,放弃 CPU 执行权限,执行进程调度。这两个调度时机都发生在管理模式下的软件中断异常处理中。由请求硬件资源而阻塞的进程会被内核自动挂起,这是由于内核定期检查进程的运行状态时,内核

定期检查程序又是由定时器中断产生时调用的;同样定时器会定期的产生中断信号,在定时器中断处理程序中,对当前执行进程时间片 count 进行递减操作,当其时间片用尽时,意味着进程要被挂起,进行新进程调度。进程 I/O 请求阻塞和时间片用尽这两种情况都是发生在中断模式下,因此执行进程切换时机一般都是发生在管理模式和中断模式下。

以上情况发生时,进程调度程序会启动从进程队列里选出新进程,执行进程上下文切换(将 CPU 执行权交给新选进程,保存挂起进程的执行状态的操作称做进程上下文切换,如果挂起进程已经结束,则不需要保存其执行状态。上下文切换时要将当前进程现场保存到其 PCB 里,将被选出进程 PCB 内容复制到寄存器里准备执行),执行被选出新进程。

2. 进程调度算法

通常操作系统会按照一定的调度算法调度进程,常用的进程调度算法有多种,它们大多被目前流行操作系统所采纳,下面介绍其中最常见的几种:

(1) 先来先服务调度算法 FCFS。FCFS(First Come First Service)是一种最简单的进程调度算法。采用该调度算法的系统按照进程进入就绪队列的时间先后将就绪进程排列在队列中,进程从队尾入队,从队头出队,先入队进程先得到 CPU 执行权被执行。每次需要进行进程调度时,进程调度程序从队头取出 PCB,为其分配资源并将其交给 CPU 执行,直到当前进程执行完或发生某种情况被阻塞。

(2) 最短作业/进程优先调度算法。SJ(P)F(Shortest Job/Process First)是指以作业/进程的运行时间为依据,对短作业/进程进行优先调度的进程调度算法。由于作业/进程在执行完毕之前,它的运行时间是不可被预测的,所以这里的运行时间只能是事先估计的。SJ(P)F 可分别用于作业调度和进程调度。

(3) 时间片轮转调度算法。为了保证用户交互性,通常操作系统一般都采用时间片轮转算法进行进程调度。系统将用户创建的进程按一定原则(如先来先服务原则)组织在进程就绪队列中,把 CPU 的运行时间分割成一个个长度相等的微小时间区间(例如 10ms),即时间片(timelice)。每次调度时把 CPU 使用权分配给队首进程,并令其独占 CPU 执行一个时间片。如果该时间片内进程执行完毕,它将释放 CPU,退出就绪队列,调度器从就绪队列再选择一个进程让它占用 CPU 来运行;如果在时间片结束时进程还未运行完成,则该进程将被剥夺 CPU 的使用权,由计时器发出时钟中断,调度程序将它送入进程队列末尾等待下次执行,然后把 CPU 分配给进程队列中另外一个进程。如果当前执行进程在时间片结束前进入了阻塞状态,例如要等待一个串口数据,等待从外围存储设备读取数据,因为阻塞状态持续时间不可预知,当前 CPU 将立即进行进程调度切换。

(4) 优先级调度算法。优先级调度算法为每一个进程分配一个优先级,当对系统进行进程调度切换时,将 CPU 的使用权分配给进程队列里面优先级别最高的进程,而优先级较低的进程则等待下次调度。当前最高优先级进程执行完毕之后,再从进程队列里挑选最高优先级进程,当然优先级最低的进程因为很难被调度执行,因此很容易出现低优先级进程被“饿死”的

情况,为了弥补该情况,通常进程的优先级是动态改变的,并且和时间片轮转调度算法结合起来使用。

3. 简单调度算法实例

下面的调度算法采用优先级和时间片调度方式(借鉴自 Linux 内核 0.11 调度程序),描述一个简单的进程调度算法。进程 PCB 结构采用第 3.6.2 节代码所示结构。进程 PCB 里有一个 count 属性,它表示当前进程的可用时间片数,它的数目越大表明该进程可以使用的时间片越多,优先级越高,其初始值由进程 PCB 的优先级属性 priority 指定。当前进程如果没有执行完毕,每被调度执行一次,其 count 就减一。

每次执行调度程序时,遍历进程队列,从进程就绪队列里选择出 count 值最大(优先级最高)的进程,如果存在优先级最高进程。将其与当前进程进行上下文切换。如果当前进程队列所有进程 count 值都等于 0 意味着所有进程的时间片已经用完,需要将每个进程的 count 设置为初始值 priority。如果当前进程队列没有就绪进程,则执行系统空闲进程 0。

上述算法可以用下面代码实现:

```
#define TASK_UNALLOCATE    -1           // 进程 PCB 未分配
#define TASK_RUNNING       0           // 进程正在进行或已经准备就绪
#define TASK_INTERRUPTIBLE 1           // 进程处于可中断等待状态
#define TASK_UNINTERRUPTIBLE 2         // 进程处于不可中断等待状态
#define TASK_ZOMBIE        3           // 进程处于僵死状态,未用到
#define TASK_STOPPED       4           // 进程已经停止
#define TASK_SLEEPING      5           // 进程进入睡眠状态
#define TASK_SZ             62         // 进程最大数

typedef struct task_struct
{
    int      pid;                // 进程 ID
    unsigned long state;         // 进程状态
    unsigned long count;         // 进程时间片数
    unsigned long timer;         //
    unsigned long priority;      // 进程默认优先级
    unsigned long content[20];   // 进程执行现场保存区
} PCB;

PCB task[TASK_SZ];              // 进程队列
PCB * current;                  // 当前执行进程指针

void schedule(void)
{
    /*
```

```

* max 用来保存当前进程队列里最高优先级进程 count
* p_tsk 保存当前进程 PID 副本
*/
long max = -1; // max 初始值为 -1, 后面作判断
long i = 0, next = 0; // next 保存最高优先级 PID
PCB * p_tsk = NULL; // 临时进程结构体指针
// 进程调度循环
while(1){
    /*
    * 循环找出进程队列里,就绪状态最高优先级进程,也就是 count 值最大进程,
    * count 越大说明其被执行时间越短,CPU 需求越高,
    * 同时保存其 PID(进程队列数组下标)到 next 里
    */
    for(i = 0; i < TASK_SZ; i++){
        if( (task[i].state == TASK_RUNNING) && (max < (long)task[i].count) ) {
            max = (long)task[i].count;
            next = i;
        }
    }
    // 如果 max 为非 0,跳出循环,说明选出了调度进程
    // 如果 max 为 0,说明 count 值最大进程 count 为 0,说明全部进程分配时间片已执行完,
    // 需要重新分配,执行 break 后面 for 语句
    // 如果 max 为 -1 说明没有就绪状态进程可被调度,退出循环,继续执行 0 进程

    if(max) break; // max = 0 时,选出新进程,跳出循环
    // max = 0,即进程队列中 count 值最大为 0,全部进程时间片用尽,需要重新分配
    for(i = 0; i < TASK_SZ; i++){
        if( task[i].state == TASK_RUNNING ) {
            // 时间片数为其默认优先级
            task[i].count = task[i].priority;
        }
    }
}
// 当前进程为选出进程,说明当前进程优先级还是最高,返回继续执行
if(current == &task[next])
    return;
// 无效 PID
if(task[next].pid < 0)
    return;

```

```
// 保存当前进程副本到 p_tsk,将选出进程设置为当前运行进程
p_tsk = current;
current = &task[next];
// 调用上下文切换函数
__switch_to(p_tsk, current);
}
```

3.6.6 上下文切换

进程调度程序选出新进程之后,CPU 就要进行进程上下文件切换,将新进程切换成当前执行进程,同时还要保存当前执行进程执行现场,为下一次调度执行做准备,该过程称做进程的上下文切换(context switch),这里的上下文就是指进程的执行状态等。

当发生上下文切换时,要将旧进程的相关状态信息保存在 PCB 中,然后再装载新调度进程的 PCB 状态信息。上下文切换是进程调度过程中的额外开销,因为它没有对任何一个进程完成任何工作。上下文切换速度因机器而不同,它依赖于内存读取速度和要复制寄存器数量等。

由上节进程调度程序可知,进行上下文切换操作时有两种情况:

- 如果当前进程执行结束,进入调度程序执行上下文切换,将当前进程 PCB 释放掉(通常操作系统 PCB 是随后释放的),然后调度新进程,恢复新进程执行现场。
- 如果当前进程因为可用时间片耗尽或休眠阻塞而进行进程调度,则要先保存当前进程执行现场,然后调度新进程,恢复新进程执行现场。

1. 执行上下文切换时 ARM 的 CPU 模式

由上节知识点可知,4 种情况下会调用调度程序,调度时机发生时,按照 CPU 所在模式可以分为管理模式下调度和中断模式下调度,其中管理模式下调度主要指软件中断中的函数调度(_sleep() _exit()),中断模式下调度主要指用户空间执行程序时产生时钟中断而进行的调度(定时器中断),由于上下文切换和硬件体系相关紧密,两者在处理上下文切换时机制稍有不同。

2. 上下文切换详解

(1) 上下文切换中保存进程 PCB。进行上下文切换时,保存当前进程执行状态到 PCB 里。由前面知识可知,一般只有中断方式或软件异常方式进入进程调度程序然后执行上下文切换,这两种方式都是通过异常处理进入的。由前面异常处理知识可知,异常产生后硬件自动完成一系列操作,通过异常向量表,进入异常处理,保存程序执行现场,开始执行异常处理。因此,当产生进程调度时,那么用户模式下寄存器值已经入栈保存了,这时就要从异常模式的栈里将用户程序状态取出来保存到 PCB 中。

该被挂起进程下次被重新调度执行时,就从进程 PCB 里将保存数据恢复到寄存器中,然

第3章 ARM 体系结构

后再返回用户程序执行。

为了实现进程上下文切换,miniOS 中 PCB 结构体中声明有一个 `content[16]` 数组,用来保存用户进程状态,每个元素对应一个寄存器,如图 3-20 所示。

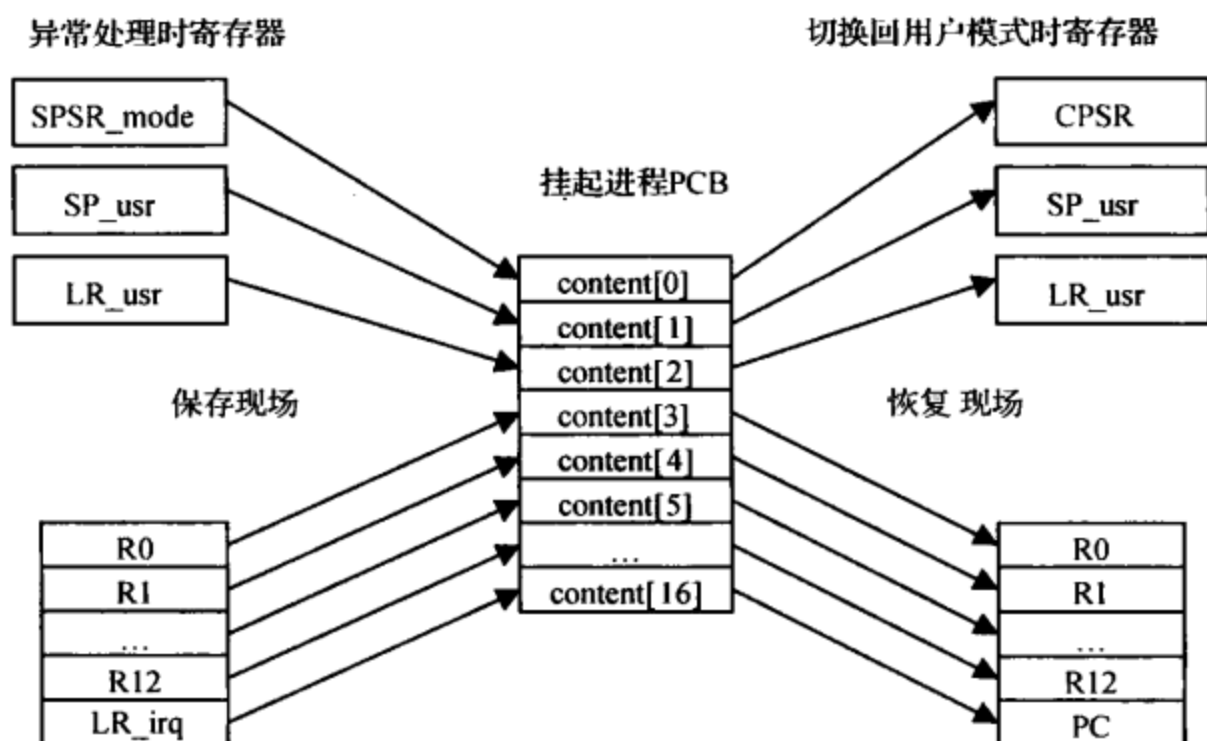


图 3-20 上下文切换保存,恢复进程 PCB

miniOS 上下文切换操作由 `__switch_to(PCB * pcur, PCB * pnext)` 函数完成,它有两个参数, `pcur` 是当前执行进程 PCB 结构体指针,也就是 PCB 的内存地址, `pnext` 是新调度进程 PCB 地址,根据 ATPCS 准则, `pcur` 保存在 R0 中, `pnext` 保存在 R1 中传递给下面的汇编程序。

`content[16]` 数组,主要用来保存用户执行状态(也就是寄存器),由于进入上下文切换函数时,CPU 可能处于不同模式,而不同模式下的栈不一样,因此,先根据 CPSR 中的 M[4:0] 位判断当前所处模式,然后加载不同模式下的栈空间基址,然后再从栈空间中取出对应保存的执行现场。保存 PCB 前,要知道保存位置,先将 `pcur` 地址偏移 20 位,指向 `content[0]`,然后将 `SPSR_mode` 保存到 `content[0]`, `SPSR_mode` 是异常模式下保存状态寄存器, `SPSR_mode` 中保存的是在进入异常状态前 CPU 的 CPSR 的值,也就是用户程序在被挂起前的 CPSR,保存该值用于将来恢复到用户程序 CPSR 寄存器中。 `content[1-2]` 分别保存 `SP_usr`、`LR_usr` 寄存器,用于将来恢复用户 SP、LR 寄存器,在异常处理时不会保存用户 SP、LR,这是由于每种模式下都有自己的 SP、LR,而多进程在执行每个程序时都运行在用户模式下,不同进程的 SP、LR 不一样,因此要保存它们,不然其他用户进程会覆盖掉当前进程 SP、LR。关于保存用户模式下的 SP、LR,这儿需要注意一下,由于执行到该代码时,已经处于特权模式,访问 SP、LR 是读取的特权模式下的对应栈指针和返回地址,要想得到用户模式下的 SP、LR,可以通过下面的指令实现:


```
stmia    r0, {sp}^      ; 保存用户模式下的 sp 到 r0
stmia    r0, {lr}^      ; 保存用户模式下的 lr 到 r0
```

上面的“^”符号在前面中断处理章节也出现过:

```
ldmia    sp!, {r0-r12, pc}^      ; 中断返回, ^表示将 spsr 的值复制到 cpsr
```

上面指令中的“^”符号表示要恢复当前模式下的 spsr 到 cpsr。

二者区别在于:

当执行多寄存器加载操作时(ldmxx)带有“^”,如果操作寄存器中含有 pc 寄存器,则表示该指令在执行完毕后,实现程序跳转同时恢复 spsr 到 cpsr,如果操作寄存器中不含有 pc 寄存器,表示从内存加载到用户模式下寄存器。

当执行多寄存器存储操作时(stmxx)带有“^”,表示将用户模式下寄存器列表的值存储到内存中和寄存器列表中有无 pc 无关。

最后保存通用寄存器 R0-R13 到 content[3-16]。其中 R13 也就是 LR 是对应异常模式下返回地址寄存器。

; __switch_to 上下文切换代码片段

```
__switch_to      ; __switch_to(PCB * pcur, PCB * pnext), pcur 是当前进程 PCB 地址,
                  ; pnext 是新进程 PCB 地址
```

```
mrs      r2, cpsr      ; 判断当前 CPU 执行模式
bic      r2, r2, #0xfffffe0      ; 取出低 5 位模式位
cmp      r2, #0x12      ; 判断是否为 irq 模式
; 设置中断模式下的 SP 栈指针
ldreq    sp, = IRQ_STACK_BASE      ; 中断模式
; 设置管理模式下的 SP 栈指针
ldrne    sp, = SVR_STACK_BASE      ; 管理模式

add      r0, r0, #CONTENT_OFT      ; r0 指向当前进程 PCB, 偏移到 PCB 结构体 content 处
mrs      r2, spsr      ; 取出 SPSR
stmia    r0!, {r2}      ; 保存 SPSR 中值到 PCB 结构体 content[0]中
stmia    r0!, {sp}^      ; 保存挂起的用户空间程序 SP, LR 到 content[1-2]
stmia    r0!, {lr}^      ; 用于将来恢复用户 SP, LR 寄存器
; 把中断处理时保存的寄存器(R0~R13)转移到当前 PCB 中
ldmia    sp!, {r2-r8}      ; 加载 SP 中保存的 R0~R6
stmia    r0!, {r2-r8}      ; 存入 r0 指向的 PCB 结构体中 content[3~9]
ldmia    sp!, {r2-r8}      ; 加载 SP 中保存的 R7~R13
stmia    r0!, {r2-r8}      ; 存入 r0 指向的 PCB 结构体中 content[10~16]
```

(2) 上下文切换中恢复进程 PCB。与保存进程 PCB 信息相反,将保存在 PCB 结构体中的

第3章 ARM 体系结构

数据恢复到寄存器中。按照保存时顺序进行恢复。content[0]保存的是 SPSR, 因此将其恢复到当前 SPSR 中。content[1-2]保存的是用户程序 LR 和 SP, 因此要手动将其恢复到用户模式下 LR 和 SP 寄存器中, 如果直接切换回用户模式, 那么从用户模式就不可能再返回当前特权模式了, 可以进入系统模式(系统模式为特权模式, 并且与用户模式共享全部寄存器, 这也是为什么用户模式和系统共享全部寄存器的主要原因之一), 设置完 LR 和 SP 之后, 返回到当前模式, 然后通过 ldmia r1, {r0-r12, pc} 恢复 content[3-16]中保存的 R0~R13 的值, 到 R0~R12 和 PC, 同时将 SPSR(保存在 content[0]中)恢复到 CPSR, 切换回用户程序。

; __switch_to 上下文切换代码片段

```
__switch_to      ; __switch_to(PCB * pcur, PCB * pnext), pcur 是当前进程 PCB 地址
                  ; pnext 是新进程 PCB 地址

add      r1, r1, #CONTENT_OFT      ; r1 指向新进程 PCB
ldmia    r1!, {r2-r4}              ; r2 是用户模式下 CPSR, R3 是 PCB 中保存的用户模式下 SP
                                          ; r4 是用户模式下 LR

msr      spsr_cxsf, r2              ; 先将 CPSR 保存到 SPSR 中, 将来一并恢复
mrs      r2, cpsr                  ; 取得当前执行状态到 r2 中, 设置完用户 SP, LR 再返回
; 切换到系统模式下, 加载用户模式下 SP, LR

msr      cpsr_c, #0xdf
mov      sp, r3
mov      lr, r4

msr      cpsr_cxsf, r2              ; 切换回当前模式
ldmia    r1, {r0-r12, pc}~         ; 将 PCB 保存数据恢复到 R0~R12 和 PC, 同时返回用户程序
```

3.7 MMU 与内存保护的实现

在常见的中、低档单片机中, 嵌入式系统的存储系统中地址空间的分配是固定的, 各个软硬件直接使用物理地址, 将程序全部装载到物理内存中, CPU 对数据和指令进行操作时也是使用物理地址。这样的设计方式虽然简单、实用, 但是如果存在以下情况时, 这种简单存储管理方式就不能胜任了。

- (1) 程序文件超过物理内存容量, 导致不可能将全部程序代码装载到内存中运行。
- (2) 在多任务操作系统中, 通常运行有多个应用程序, 多个应用程序的大小往往超过系统内存总容量。

实际上, 在程序运行时, 没有必要将全部程序代码装载到内存中, 只需将当前要执行的部分先装载入内存, 其余部分在被执行时再从磁盘装载。当物理内存不足时, 可以通过内存交换技术将暂时不运行的程序部分交换到磁盘中, 腾出物理内存供其他程序使用。

通常在内存容量保持不变时,通过虚拟地址空间来扩大程序的可访问空间。虚拟存储器从逻辑上扩大了地址空间,让用户感觉到大容量可用内存空间。在 32 位 CPU 系统中,虚拟内存地址空间为 $0 \sim 0xFFFFFFFF$ (4GB),指向该空间区域的地址称做虚拟地址。

虚拟地址与物理地址之间通过映射建立关系,最终的虚拟地址要映射到具体物理地址上才可以访问数据。通常将物理内存空间划分成同样大小的一小块块空间,然后为这些小空间建立映射关系。如图 3-21 所示,由于虚拟地址空间远大于物理空间,有可能多个虚拟地址空间映射到一块物理地址空间。

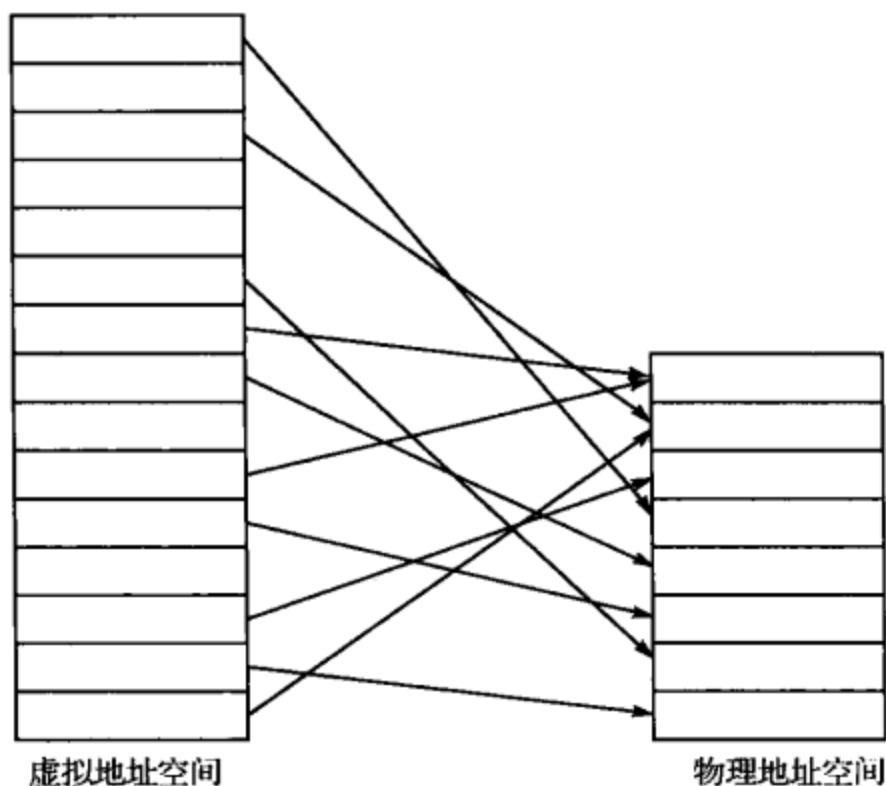


图 3-21 虚拟地址映射图

3.7.1 存储管理单元 MMU

现代的操作系统都支持虚拟内存,这样可以让操作系统内核更合理的保护内存区,控制硬件地址访问权限,切换用户的进程空间。这些机制都是通过 CPU 内部的 MMU 来实现的。在 ARM 系统中,MMU 主要完成以下工作:

- (1) 虚拟存储空间地址到物理存储空间地址的映射。
- (2) 控制存储器访问权限。
- (3) 设置虚拟存储空间的缓冲特性。

1. 虚拟存储空间地址到物理存储空间地址的映射

如图 3-22 所示,在虚拟存储空间地址到物理存储空间地址进行映射时,存在 3 种内存地址:

(1) 物理内存地址 PA(Physical Address)。物理内存地址 PA 对应真正内存控制器中每个存储单元,通过物理内存地址直接可以找到对应内存数据空间。

(2) 虚拟内存地址 VA(Virtual Address)。所谓虚拟内存地址 VA,其实就是一个在物理内存空间里不存在的地址,通过映射规则将该地址映射到其对应的物理地址 PA,这样 CPU 通过这个不存在的地址和它们之前的映射关系进行物理内存的操作。

(3) 修正虚拟内存地址 MVA(Modified Virtual Address)。MMU 启动之后,CPU 内核对外围设备访问的都是 VA 地址,VA 经过 CP15 的 C13 寄存器修正之后变成 MVA 供 MMU 和 cache 使用,经过 MMU 的地址映射,MVA 变成 PA,通过 PA 访问外围硬件设备。

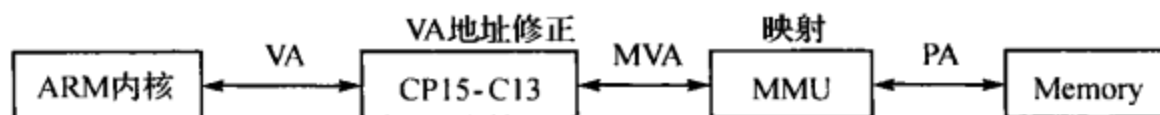


图 3-22 ARM CPU 内部地址转换

- ARM 内核所有参加运算的地址都是 VA 虚拟地址,ARM 内核不用关心 VA 与 PA 的映射。

- 进入 MMU 的是 MVA 地址,出来 MMU 的是 PA 物理地址,MMU 看不到 VA 地址。

- 对外围设备的访问都是实际物理地址,外围设备不用关心自己的地址被映射成什么虚拟地址。

开启了 MMU 之后,CPU 工作时用的所有地址(数据地址/指令地址)都是虚拟地址。在操作系统中,由于每个用户进程的运行地址在程序编译时就确定了,并且都是相同的运行地址(从 0 地址开始)。那么在进程执行时,所有用户进程使用的虚拟地址空间就会有重叠,这就无法对每个进程地址空间进行单独管理,同样,在进程切换时,要将重叠的 VA 地址映射到不同的 PA 上(因为进程是保存在 PA 地址空间内的,不同进程肯定要保存在不同地址空间里),这样的话要重建页表,使无效 caches 和 TLBS 等,代价非常大。为了解决这个问题,使用 MMU 中一个特殊寄存器和修正虚拟地址来将不同的用户进程进行区分。

在 ARM 中每个进程最大可使用内存空间为 32MB,当一个 CPU 访问的 VA 地址小于 32MB 时,MMU 认为它是一个用户进程地址,而不同进程其地址空间是重叠的,每个进程有一个唯一 PID 用来区分,为了将不同用户进程地址空间区分开,则将 PID 逻辑左移 25 位,成为一个大于 32MB 的地址空间的基址(低 25 位全部为 0),然后再加上 VA(小于 32MB)作为偏移地址,形成一个新的地址。

```

if(VA < 32M)
    MVA = PID << 25 + VA;
else
    MVA = VA;
  
```

为什么会通过 PID 左移 25 位形成一个地址基址？ARM 是 32 位机，地址宽度为 32 位，32MB 地址空间占用 25 位，剩余 7 位可以作为不同进程空间的基址，所以让 PID 逻辑左移 25 位，形成地址空间基址，再加上低于 32M 的 VA，形成一个 32 位 MVA 地址。然后再对这个 MVA 进行映射。

ARM 内核支持运行多个进程，每个进程执行空间为 0~32MB，则 4GB 地址空间被分成 128 个 32MB 空间，也就是说 ARM 内核最多支持 128 个进程同时运行，每个进程的地址空间为 $PID * 0x2000000 \sim PID * 0x2000000 + 0x01FFFFFF$ 。

上述由 VA 到 MVA 转换操作是由硬件自动完成，在使用 VA 地址时，只要给定其 PID，则硬件自动找到其物理地址 PA。毫无疑问，PID 是通过设置对应寄存器来告诉 MMU 硬件的，在需要进行地址转换时将 PID 的值写入 CP15 协处理器的 C13 寄存器即可。

通常在用户进程执行过程中，其地址访问操作都在其对应地址空间，只要不切换进程，C13 寄存器的值不需要重新设置。当进程进行切换时，必须重新设置 C13 的 PID，这样才能切换到对应进程的地址空间。

下面通过 ARM 内核结构图再来看下 VA 到 PA 转换过程。如图 3-23 所示，系统在开启了 MMU 之后，CPU 内核操作的都是虚拟内存地址 VA (ARM920T 采用哈佛结构，其指令总线 and 数据总线是分开的，虚拟内存地址包含指令地址和数据访问地址)，虚拟地址 VA 经过 C13 (CP15 里的寄存器 13) 后变成修正虚拟地址 (MVA)，然后 MMU 对 MVA 进行地址映射，对通过 MMU 处理之后的地址都是实际物理内存地址 (PA)，将物理地址送上地址总线，交给内存控制器去寻址。

VA 经过 CP15 的 C13 寄存器转换成 MVA 之后，就被送往 MMU 进行 MVA 到 PA 地址的映射。其映射过程可以用 C 语言中的指针数组来理解。

指针数组首先是一个数组，里面的每一个元素是一个指针也就是一个内存地址，地址指向一个有效内存空间。每个数组元素都有一个对应的下标 (C 语言是从 0 开始)，下标其实就是数组首地址的相对偏移位置，如图 3-24 所示。

假如有一个 32 位虚拟地址，其高 12 位为基址位，低 20 位为偏移位，如图 3-25 所示。

将虚拟地址的高 12 位取出来，作为指针数组的下标，指针数组下标对应的元素是一个内存地址，这个地址作为一段地址的基址，将该基址加上虚拟地址的低 20 位偏移地址，形成了一个新的地址，这就是虚拟地址与物理地址内存映射的基本原理。可以用下面语句表示：

$$PA = \text{tabMap}[VA \gg 20] + VA \& 0xfffff;$$

因此可以根据自己的需要将整个内存空间分成不同的段，将段基址作为 tabMap 的元素。元素中地址加上虚拟地址偏移位形成物理地址，从而实现地址的映射。

在 MMU 里，对应指针数组 tabMap 有个专有名词：页表，它是存放在内存中的一块映射区域，主要用来做虚拟地址 VA 和物理地址 PA 的映射，页表里的每一个元素称为页表项，其作用类似 tabMap 里的数组元素。由于页表是内存中的一块区域并且类似数组，因此该区域

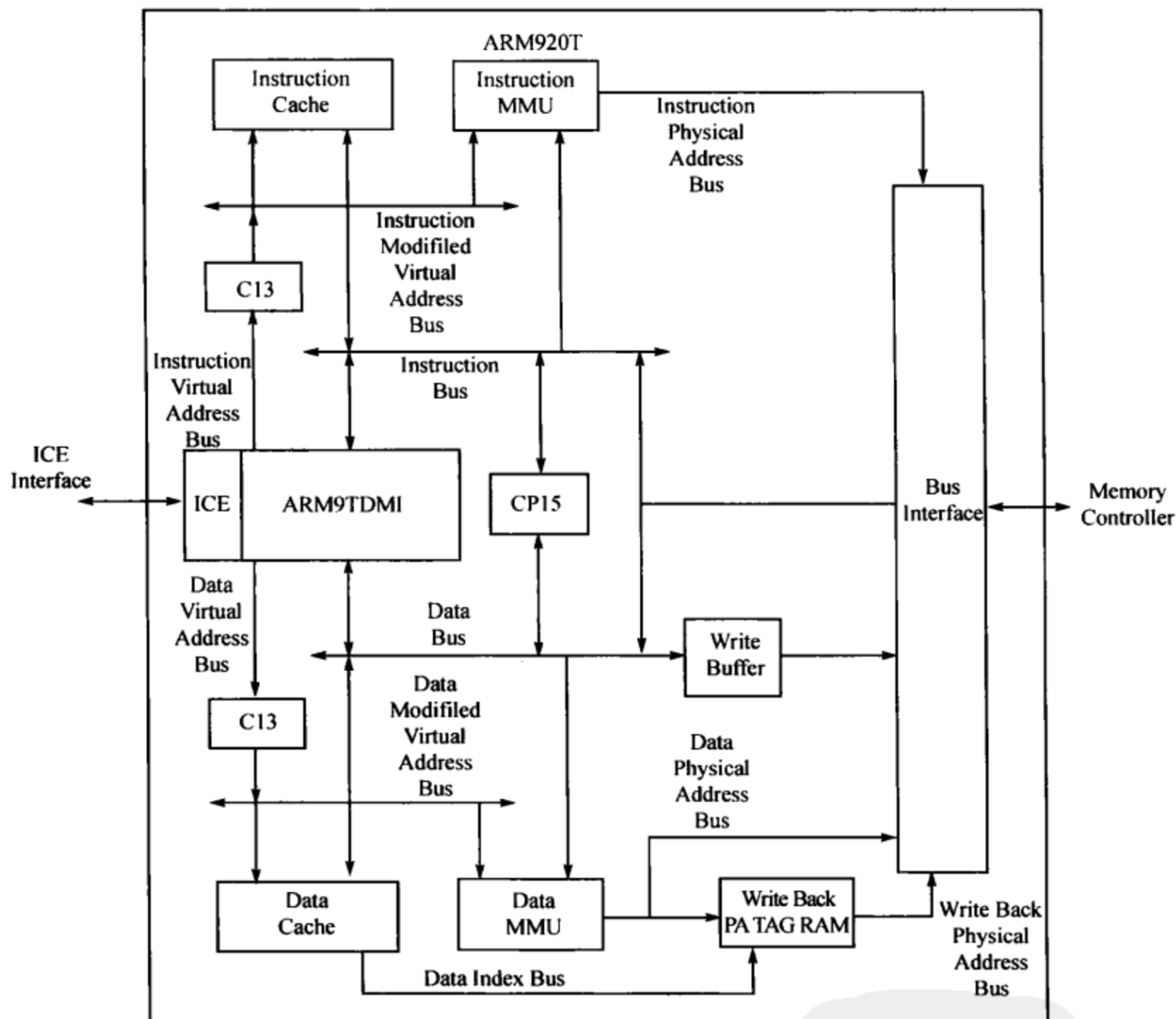


图 3-23 ARM CPU 虚拟地址映射

有一个首地址,称为页表转换基址 Translation Table Base(TTB),简称页表基址,只要指定了页表基址,就可以找到 VA 与 PA 的映射关系。页表基址可以通过 CP15 的 C2 寄存器来设置。TTB 基址最多可以指向 4096 个页表项(每个页表项占 32 位),每个页表项可以描述 1MB 的地址空间,来形成 4GB 的虚拟地址空间。因此 CP15 的 C2 寄存器中值是 16KB 对齐的,在读取 C2 寄存器时,其低 14 位全部被置为 0,如图 3-26 所示。

TTB 指向一块 16KB 对齐的内存空间,虚拟地址高 12 位作为 TTB 指向页表空间的偏移地址,从而可以找到 4B 对齐的一个页表项(TTB 页表基址低 14 位为 0,16KB 对齐,MVA 高 12 位为偏移位,最多寻址 4096 个项),每个页表项用来描述 1MB 大小地址空间。经过一级映

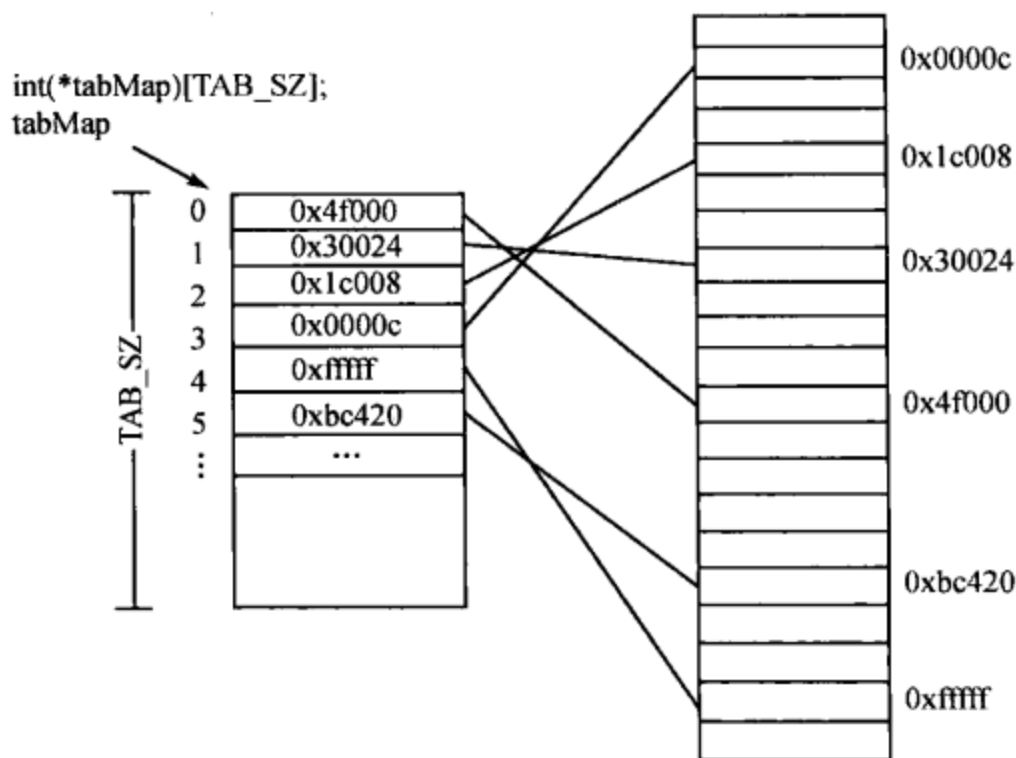


图 3-24 指针数组



图 3-25 虚拟地址

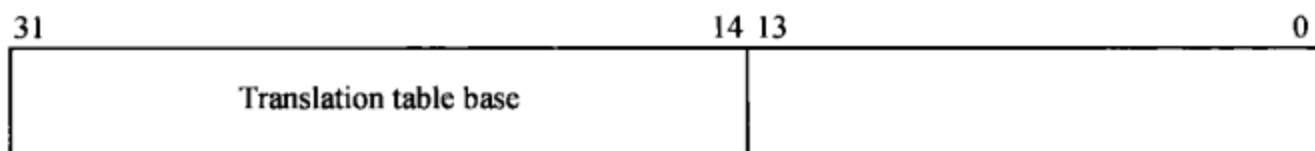


图 3-26 页表结构

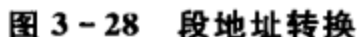
射后找到的页表项,又称为一级映射描述符。它可以对 1MB 大小的地址空间进行描述,决定是用来直接指向一个物理地址或对 1MB 空间再次进行映射,这是其[1:0]位决定的,如图 3-27 所示。

根据一级映射描述符[1:0]位,页表项主要包括无效页表项、粗页表页表项、段页表项和细页表页表项,每个页表项对其对应粗页表、段、细页表进行描述。

(1) 无效页表项(Fault)。当一级描述符 [1:0]位未设置时,该描述符为无效描述符。

(2) 段(Section)。一级映射描述符 [1:0]位为 0b10 时,该页表项表示对段地址空间进行描述,通过[31:20]共 12 位的段基址指向 1MB 物理内存空间首地址,再加上 MVA[19:0]低 20 位偏移位,指向物理内存地址。段可以表示 1MB 大小的地址空间。

段地址转换(图 3-28)过程如下:

图 3-27 页表项描述符结构

(b) 将 TTB 页表基址[31:14]与 MVA 地址[31:20]相加,组成低 2 位为 0 的 32 位地址,定位到具体页表项。

(c) 由页表项[1:0] = 0b10 位可知,该页表项存放一级映射描述符为段描述符,根据[31:20]定位到 1MB 大小物理地址空间首地址(段基址)。

(d) 将段基址[31:20]与 MVA 地址[19:0]相加,组成 32 位地址,相加得到具体物理地址。

(3) 粗页表(Coarse page table)。一级描述符 [1:0] 位为 0b01 时, 该页表项表示对粗页表进行描述, 如图 3-29 所示, 粗页表采用二级映射方式, 粗页表项中 [31:10] 作为二级页表基址指向粗页表, 将 1MB 空间进行再次映射。每个二级页表包含 256 个二级页表项 (大小为 1KB), 每个二级页表项可以描述 4KB 大小的物理地址空间。因此粗页表也可以表示 1MB 物理地址空间。根据粗页表项 [1:0] 位的不同, 又可分为: 无效描述符, 大页描述符 (支持 64KB 大小地址空间) 和小页描述符 (支持 4KB 大小地址空间)。

| 31 | | | | | | | | | | | | | | | | 16 15 | | 12 11 | | 10 9 | | 8 7 | | 6 5 | | 4 3 | | 2 1 | | 0 | | |
|-------------------------|--|--|--|--|--|--|--|--|--|--|-----|-----|-----|-----|-----|-------|---|-------|------------|-------|------------|-----|--|-----|--|-----|--|-----|--|---|--|--|
| | | | | | | | | | | | | | | | | | | 0 | 0 | Fault | | | | | | | | | | | | |
| Large page base address | | | | | | | | | | | | | ap3 | ap2 | ap1 | ap0 | C | B | 0 | 1 | Large page | | | | | | | | | | | |
| Small page base address | | | | | | | | | | | ap3 | ap2 | ap1 | ap0 | C | B | 1 | 0 | Small page | | | | | | | | | | | | | |

图 3-29 粗页表中二级描述符

① 无效描述符。当一个二级映射描述符[1:0]位为 0b00 时,该描述符为无效描述符。

② 大页描述符(Large page descriptor)。二级映射描述符[1:0]位为 0b01 时,该描述符为大页描述符。其中[31:16]为大页基址(large page base address),它可以表示一个 64KB 大小的物理地址空间首地址。粗页表中每个二级页表项只能表示 4KB 大小地址空间,如果大页描述符保存在粗页表中,则连续 16 个二级页表项都保存同一个大页表描述符。类似地,细页表中每个二级页表项只能表示 1KB 地址空间,如果大页表描述符保存在细页表中,则连续 64 个二级页表项都保存同一个大页描述符。

大页地址转换(图 3-30)过程如下:

(a) 从 CP15 寄存器 C2 中取出 TTB 页表基址。

(b) 将 TTB 页表基址[31:14]与 MVA 地址[31:20]相加,组成低 2 位为 0 的 32 位地址,定位到具体页表项。

(c) 由页表项[1:0] = 0b01 位可知,该页表项存放一级映射描述符为粗页描述符,根据[31:10]定位到粗页表基址。

(d) 将粗页表基址[31:10]与 MVA 地址[19:12]相加,组成低 2 位为 0 的 32 位地址,得到具体粗页表项地址,定位到粗页表项。

(e) 粗页表项内存放二级映射描述符,通过 $[1:0] = 0b01$ 可知,二级映射描述符为大页描述符,取出 $[31:16]$ 位作为页基址。

(f) 将页基址[31:16]与 MVA[15:0]相加得到具体物理地址。

③ 小页描述符(Small page descriptor)。二级映射描述符[1:0]位为 0b10 时,该描述符为小页描述符。其中[31:12]为小页基址(small page base address),它可以表示一个 4KB 大小

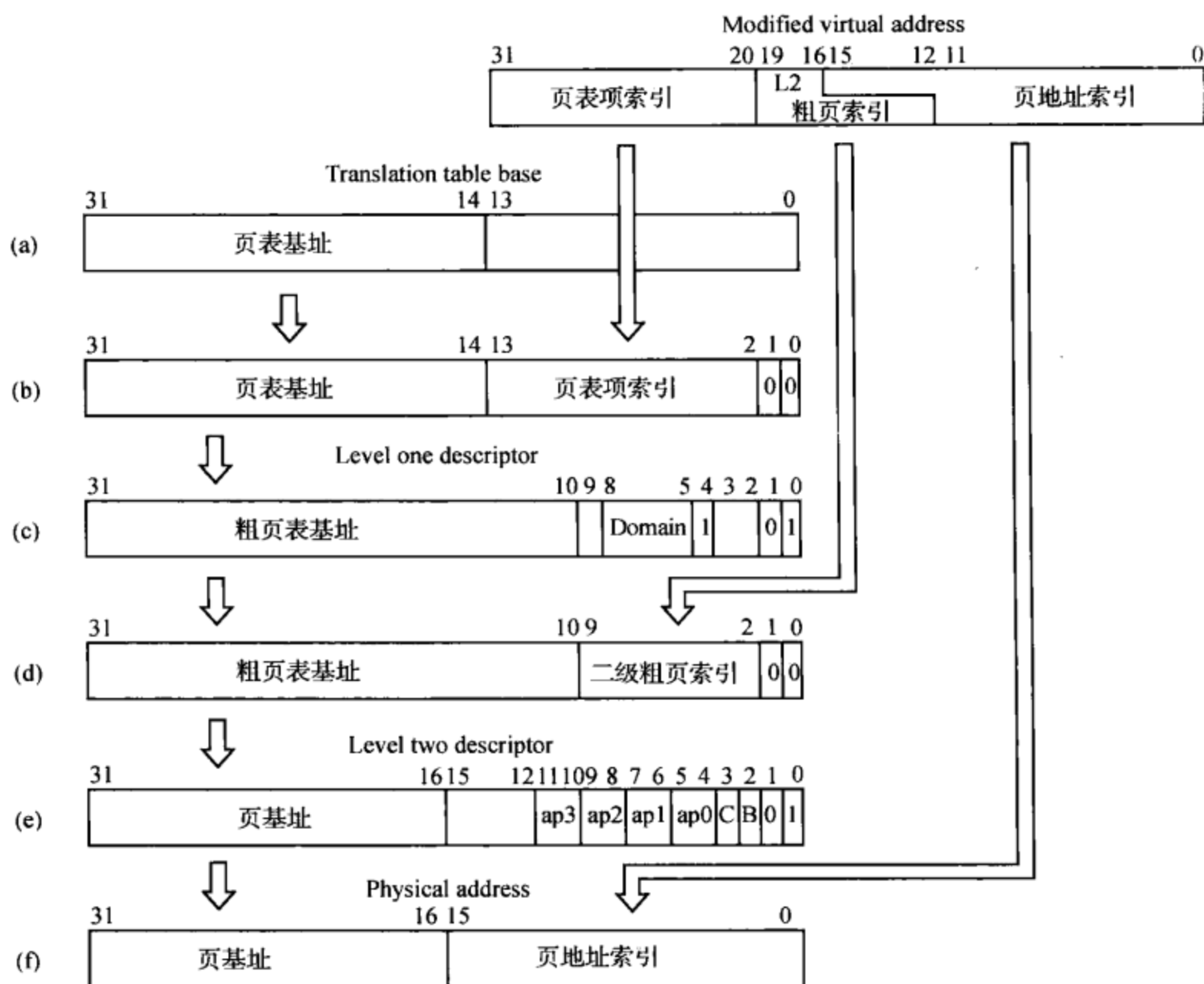


图 3-30 大页地址转换

的物理地址空间首地址。粗页表中每个二级页表项只能表示 4KB 大小地址空间,如果小页描述符保存在粗页表中,则只需要一个二级页表项来保存一个小页描述符。类似地,细页表中每个二级页表项只能表示 1KB 地址空间,如果小页表描述符保存在细页表中,则连续 4 个二级页表项都保存同一个小页描述符。

小页地址转换(图 3-31)过程如下:

(a) 从 CP15 寄存器 C2 中取出 TTB 页表基址。

(b) 将 TTB 页表基址[31:14]与 MVA 地址[31:20]相加,组成低 2 位为 0 的 32 位地址,定位到具体页表项。

(c) 由页表项[1:0] = 0b01 位可知,该页表项存放一级映射描述符为粗页描述符,根据[31:10]定位到粗页表基址。

(d) 将粗页表基址[31:10]与 MVA 地址[19:12]相加,组成低 2 位为 0 的 32 位地址,得

到具体粗页表项地址,定位到粗页表项。

(e) 粗页表项内存放二级映射描述符,通过 $[1:0] = 0b10$ 可知,二级映射描述符为小页描述符,取出 $[31:12]$ 位作为小页基址。

(f) 将小页基址 $[31:12]$ 与 MVA $[11:0]$ 相加得到具体物理地址。

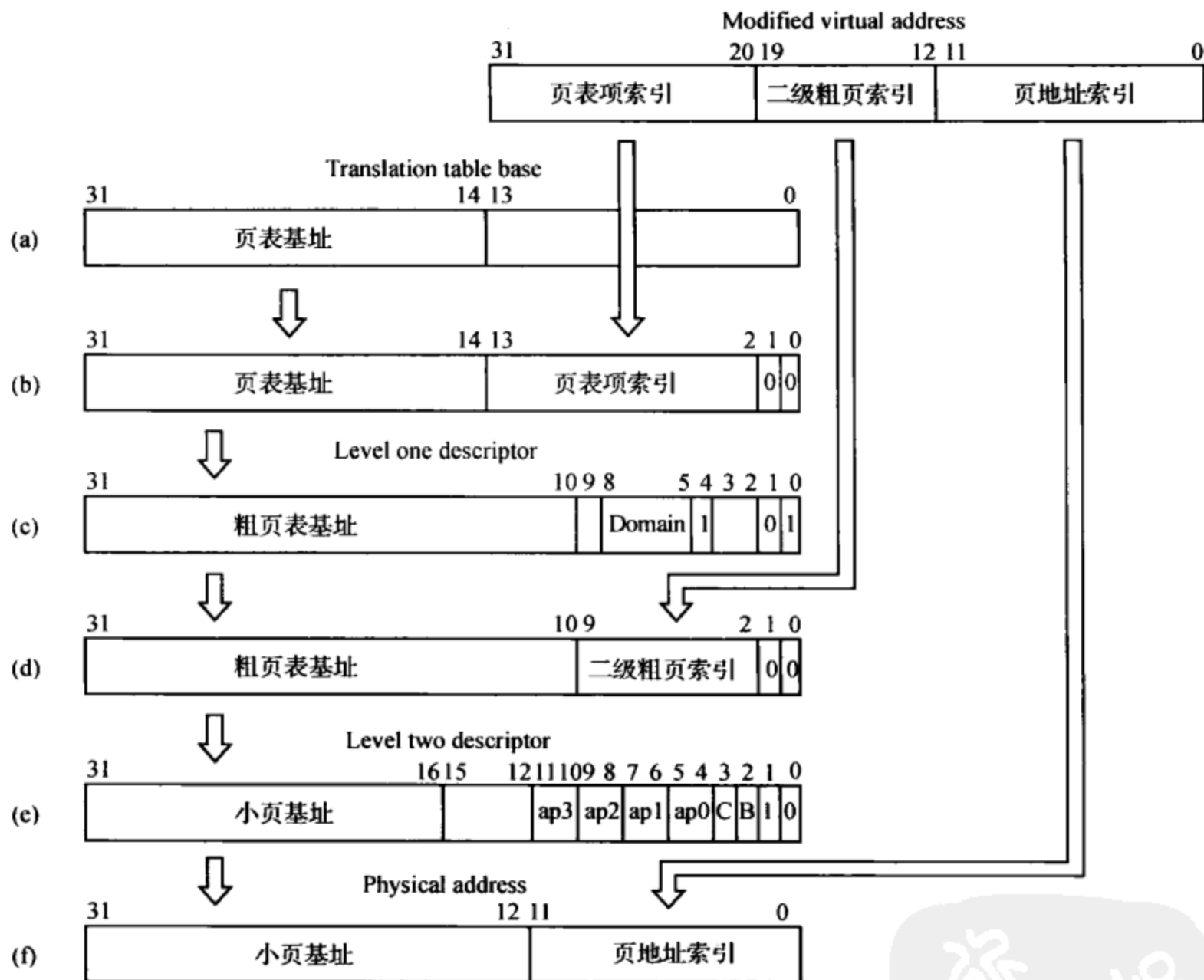


图 3-31 小页地址转换

(4) 细页表(fine page table)。一级描述符 $[1:0]$ 位为 $0b11$ 时,该页表项表示对细页表进行描述,细页表采用二级映射方式, $[31:12]$ 作为二级页表基址指向细页表,将1MB空间进行再次映射。每个二级页表最多支持1024个二级页表项,因此它可以描述1KB大小的物理地址空间。

极小页描述符(Tiny page descriptor)如图3-32所示。

二级映射描述符 $[1:0]$ 位为 $0b10$ 时,该描述符为极小页描述符。其中 $[31:10]$ 为极小页基址(tiny page base address),它可以表示一个1KB大小的物理地址空间首地址。极小页描述

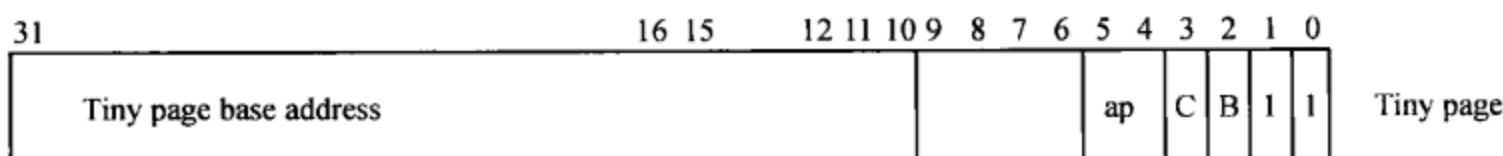


图 3-32 极小页描述符

符只能保存在细页表中,用一个细页表项来保存一个极小页描述符。

极小页地址转换(图 3-33)过程如下:

(a) 从 CP15 寄存器 C2 中取出 TTB 页表基址。

(b) 将 TTB 页表基址[31:14]与 MVA 地址[31:20]相加,组成低 2 位为 0 的 32 位地址,定位到具体页表项。

(c) 由页表项[1:0] = 0b11 位可知,该页表项存放一级映射描述符为细页描述符,根据 [31:12]定位到细页表基址。

(d) 将细页表基址[31:12]与 MVA 地址[19:10]相加,组成低 2 位为 0 的 32 位地址,得到具体细页表项地址,定位到细页表项。

(e) 细页表项内存放二级映射描述符,通过[1:0] = 0b11 可知,二级映射描述符为极小页描述符,取出[31:10]位作为极小页基址。

(f) 将极小页基址[31:10]与 MVA[9:0]相加得到具体物理地址。

由段,大页,小页,极小页的地址转换过程可知:

(1) 以段进行映射时,MVA 地址被分成两部分,通过 MVA[31:20]结合页表基址得到一段大小(1MB)的起始物理地址,通过 MVA[19:0]定位到具体物理地址。

(2) 以大页进行映射时,采用二级映射方式,通过 MVA[31:16]结合页表基址得到一个大页(64KB)的起始物理地址,通过 MVA[15:0]定位到具体物理地址。

(3) 以小页进行映射时,采用二级映射方式,通过 MVA[31:12]结合页表基址得到一个小页(4KB)的起始物理地址,通过 MVA[11:0]定位到具体物理地址。

(4) 以极小页进行映射时,采用二级映射方式,通过 MVA[31:10]结合页表基址得到一个极小页(1KB)的起始物理地址,通过 MVA[9:0]定位到具体物理地址。

2. 控制存储器访问权限

操作系统通过 MMU 可以实现虚拟内存对物理地址映射,同时也提供了对存储空间权限的管理控制。这样可以保护内存空间,防止非法程序随意破坏重要数据信息。

内存的访问权限是由 CP15 寄存器 C1 的 R, S, A 控制位和页表项中描述符的 AP 位和 CP15 寄存器 C3 域访问控制联合作用来控制的。

CP15 寄存器 C1 中的 A 位表示是否对访问地址进行对齐检查,所谓对齐检查就是指每次访问字数据时地址是否是 4 字节对齐的(32 位地址中低 2 位为 0),访问半字数据时地址是否为 2 字节对齐(32 位地址中最低位是否为 0)。如果访问地址未对齐,则产生“Alignment

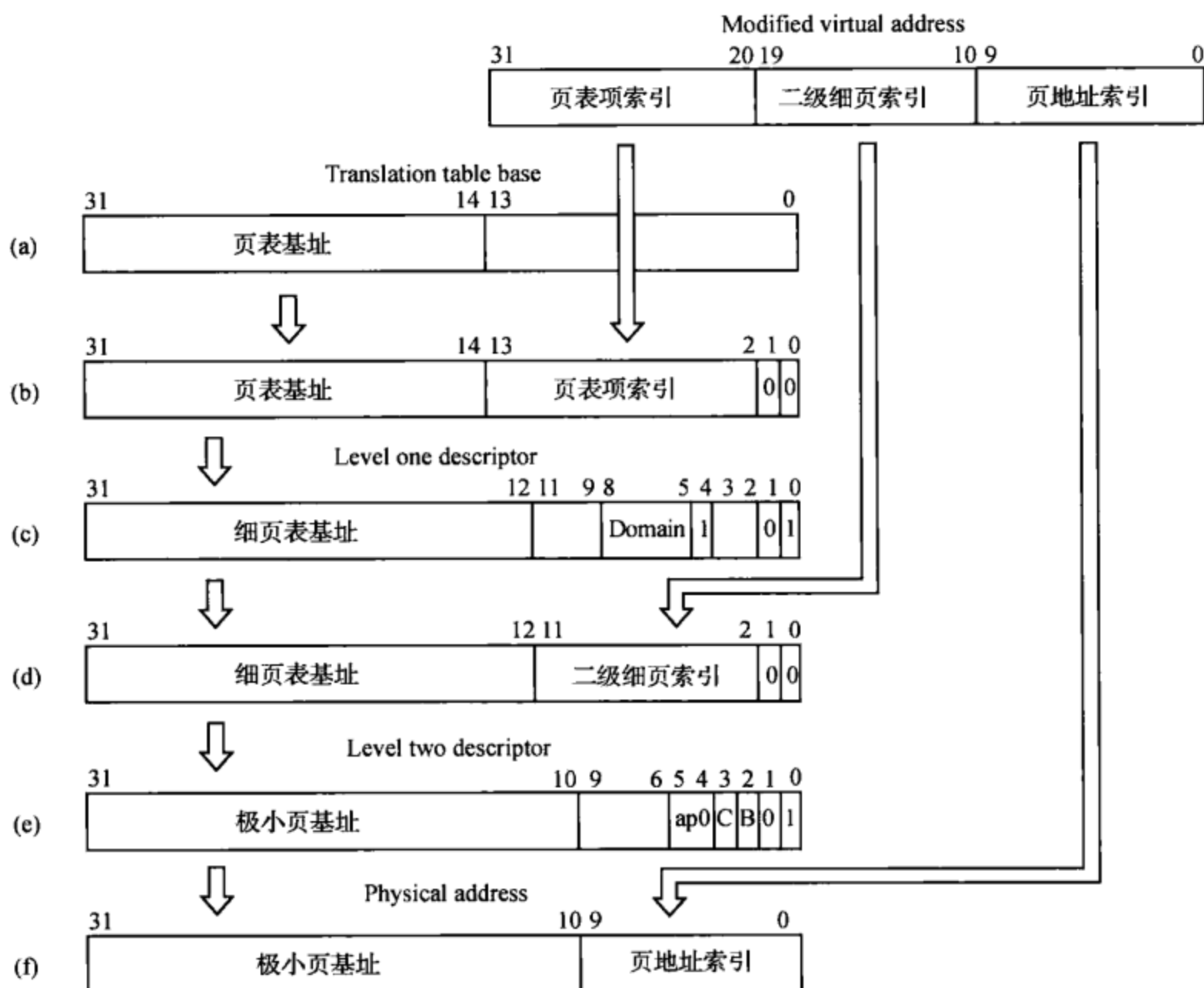


图 3-33 极小页地址转换

fault”异常。无论 MMU 是否开启都可以设置是否对齐检查。地址对齐检查是在 MMU 权限检查和地址映射之前最先做的工作。

为了对存储器进行统一权限管理,MMU 将存储空间分为最多 16 个域(domain)。该区域具有相同的访问控制权限。CP15 寄存器 R3 用于控制与域相关的操作。这样就能很方便地将某个域的地址空间包含在虚拟地址空间中,或者排除在虚拟地址空间。

CP15 寄存器 R3 的格式如表 3-18 所列。

表 3-18 ARM 存储器中的域

| | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|-----|-----|-----|-----|
| 31:30 | 29:28 | 27:26 | 25:24 | 23:22 | 21:20 | 19:18 | 17:16 | 15:14 | 13:12 | 11:10 | 9:8 | 7:6 | 5:4 | 3:2 | 1:0 |
| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

第3章 ARM 体系结构

其中每个域通过两位来进行控制,其控制编码如表 3-19 所列。

表 3-19 域访问类型

| 控制编码 | 访问类型 | 意义 |
|------|-------|--------------------------|
| 0b00 | 无访问权限 | 访问时会产生访问失效 |
| 0b01 | 用户类型 | 根据页表项里访问权限位,决定是否允许访问地址空间 |
| 0b10 | 保留 | 设置为该值会产生不可预知结果 |
| 0b11 | 管理员权限 | 不考虑页表项里的访问权限位,拥有全部访问权限 |

如果对应域权限为:无访问权限(0b00)或保留类型(0b10),则会产生 Section domain fault 异常,如果为管理员权限(0b11),则直接访问物理内存,如果为用户类型(0b01),则要检查 MVA 所在页表项中描述符 AP 位权限。

页表项描述符中 AP 位和 CP15 寄存器 C1 的 S,R 位组合可以产生多种访问权限。ARM CPU 存在 7 种工作模式,其中 6 种属于特权模式,1 种属于用户模式,在特权模式和用户模式下,相同的 AP 位,S 位和 R 位的组合,其访问权限也不相同,如表 3-20 所列。

表 3-20 AP 访问权限

| AP | S R | 特权模式时访问权限 | 用户模式时访问权限 |
|------|-----|-----------|-----------|
| 0b00 | 0 0 | 无访问权限 | 无访问权限 |
| 0b00 | 1 0 | 只读 | 无访问权限 |
| 0b00 | 0 1 | 只读 | 只读 |
| 0b00 | 1 1 | 不可预知 | 不可预知 |
| 0b01 | X X | 读/写 | 无访问权限 |
| 0b10 | X X | 读/写 | 只读 |
| 0b11 | X X | 读/写 | 读/写 |

页表项描述符对应权限位,如图 3-34 所示。

其中[8:5]这 4 位,用来表示这块内存区域的所属域(正好可以用 4 位表示 16 个域)。段描述符中[11:10]两位表示该段地址空间的访问权限,如图 3-35 所示。

粗页表项中描述符中存在四组 AP 权限 AP0~AP3,每组对应页中的 1/4 地址空间访问权限,如对于大页描述符,AP0 对应 64KB 空间中开始 16KB 空间访问权限,AP3 对应 64KB 中最后 16KB 空间访问权限。对于小页描述符,AP0 对应 4KB 开始 1KB 空间访问权限,AP3 对应 4KB 中最后 1KB 空间访问权限。

在开启了 MMU 之后,任何经过 MMU 的 MVA 地址都要进行有效性检查,其检查顺序如下:

- (1) 对齐检查(如果开启对齐检查),未对齐产生 Alignment fault 异常。
- (2) 取得当前 MVA 地址一级映射描述符,如果是无效描述符,产生 Section translation fault 异常。
- (3) 如果是段映射描述符,检查 MVA 所属域状态,如果所属域访问类型为无访问权限(0b00)或保留类型(0b10),产生 Section domain fault 异常,如果为管理员权限(0b11),则直接访问物理内存,如果为用户类型(0b01),则要检查段描述符 AP 位权限。如果 AP 位没有对应操作权限,产生 Section permission fault。如果拥有对应操作权限,经过 MMU 地址映射,访问物理内存。
- (4) 如果是页映射描述符(粗页描述符和细页描述符),检查二级页表项,如果是无效描述符,产生 Section translation fault 异常,如果是有效描述符(大页描述符,小页描述符或极小页描述符),检查描述符 Domain 位所属域状态,如果所属域访问类型为无访问权限(0b00)或保留类型(0b10),产生 Section domain fault 异常,如果为管理员权限(0b11),则直接访问物理内存,如果为用户类型(0b01),则要检查对应页描述符 AP 位权限。如果 AP 位没有对应操作权限,产生 Section permission fault。如果拥有对应操作权限,经过 MMU 地址映射,访问物理内存。

第3章 ARM 体系结构

内存,如图 3-36 所示。

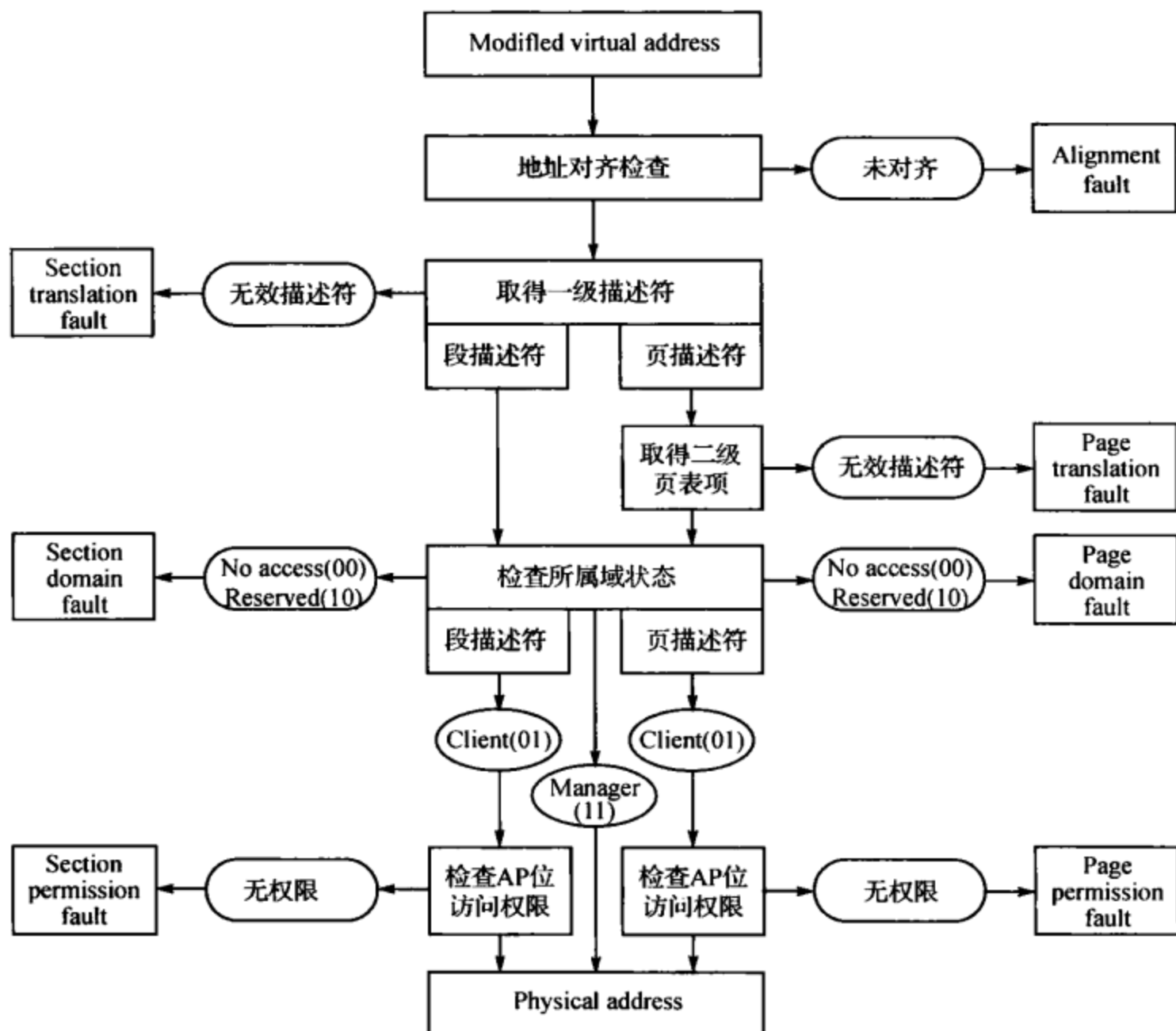


图 3-36 MMU 地址权限检查流程

3. TLB(Translation Lookaside Buffers)转换后备缓存

由于页表是存放在内存的,在虚拟地址映射时,查询页表的代价是非常大的,使用一级页表进行地址映射转换时,每次读/写数据需要访问两次内存,第一次访问一级页表获得物理地址,第二次才是真正的读/写数据。使用二级页表,每次读/写数据需要访问 3 次内存,访问两次页表(一级页表和二级页表)获得物理地址,第三次都真正读/写数据。

虽然 MMU 提供了虚拟地址映射功能,可以运行更大程序,支持多进行执行和相关权限检查,但是地址的转换过程大大降低了 CPU 的性能。为了保留 MMU 的优点又提高 CPU 的性能,CPU 中通常使用 TLB 转换后备缓存解决上述问题。

通常在程序执行过程中,所用到的指令,数据的地址往往集中在很小的范围内,对页表的

存储空间访问并不是完全随机的,也就是说,对页表的访问在一定时间内,往往只是局限在少数页表项中。根据这一特点,在 CPU 内采用了容量更小(通常为 8~16 个字)访问速度较快(和 CPU 寄存器一样速度)的存储器来存放当前需要访问的页表项(段/大页/小页/极小页描述符)。当对存储器地址进行访问时,先去该存储器中去查找,如果能查找到就不用再通过一级映射和二级映射方式计算物理地址了。这样无疑会大大提高内存映射速度。

当 CPU 需要访问内存时,先在 TLB 中查找需要映射用到的页表项,如果该页表项不存在,则 CPU 从内存中的页表中进行查找,并将查询结果保存到 TLB 中。当下次 CPU 又需要该页表项时,可以直接从 TLB 中读取,从而使虚拟地址映射速度大大加快。

当内存中的页表内容发生改变,或者通过修改 CP15 寄存器 C2 的页表基地址寄存器时,TLB 中内容需要全部被清除,MMU 提供了相关硬件支持,CP15 寄存器 C8 用来控制清除 TLB 操作。

MMU 还可以将某些 TLB 中的页表项进行锁定(looked down),从而使得进行该页表项地址变换时速度很快。在 MMU 中寄存器 C10 用于控制 TLB 内容锁定。

3.7.2 cache

基于程序执行过程中的局部性,在主存和 CPU 通用寄存器之间设置一个高速的,容量相对较小的存储器,它会将正在执行的指令地址附近的一部分指令或数据从主存调入到这个存储器中。这个介于主存和 CPU 之间的高速小容量存储器称为高速缓冲存储器(cache)。它的作用类似于工厂中的临时仓库,如图 3-37 所示。工厂在生产过程中用到的原材料存放在郊区仓库 A 中,由于距离比较远,每次工厂加工需要材料都需要较长时间,而工厂一般短期内都会生产同一种产品,于是在工厂附近新建一个小仓库(城市地价太高,成本太大,不能建大仓库),每次从 A 仓库运输材料时,都多运输一些相关材料,暂时放到小仓库中,如果工厂在需要材料时先去 B 仓库看看有没有需要的材料,如果有,直接拿来使用,如果没有再去 A 仓库调货。

CPU 在启用 cache 之后,CPU 读取数据时,如果 cache 中有这个数据的备份,则直接返回,否则从主存中读入数据,并存入 cache,下次再使用这个数据时,直接可以使用 cache 中的备份。

CPU 在执行写操作时有写通式和回写式两种方式。

(1) 写通式(Write Through)。任一从 CPU 发出的写操作在送到 cache 的同时,也写入到主存中,以保证主存数据与 cache 中数据同步。它的优点是操作简单,但由于主存访问速度较慢,降低了写操作性能并且占用总线时间。

(2) 回写式(Write Back)。为了克服写通式中每次数据写入时都要写入主存,从而造成性能降低,可以采用回写式写入方式。

回写式是指,要执行写入操作的数据先将其写入到 cache 中,而暂时不对主存进行写入更

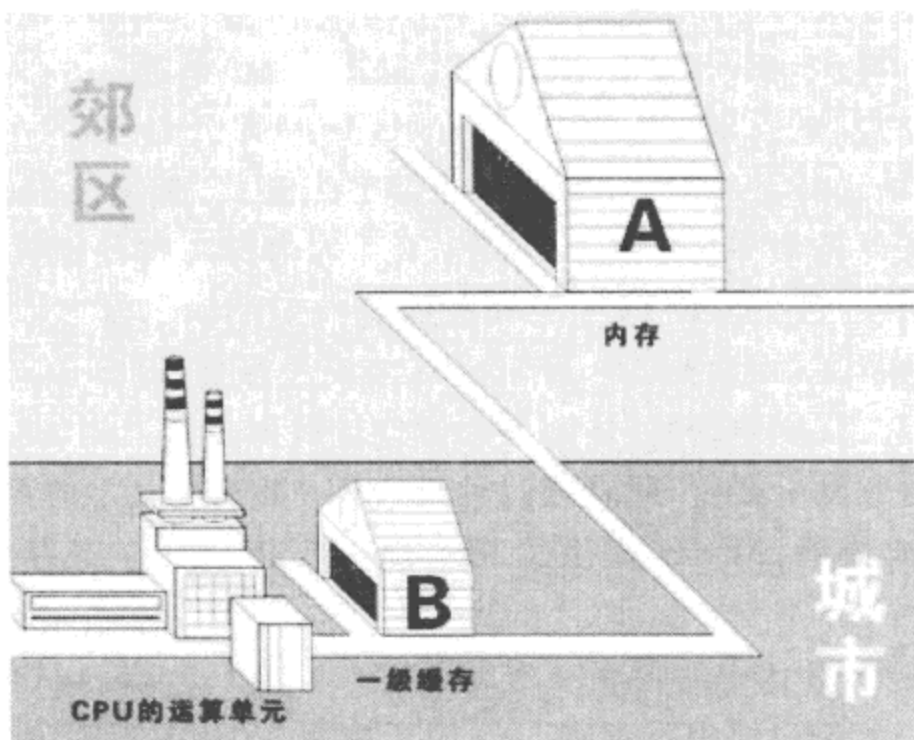


图 3-37 cache

新操作。这样会造成 cache 中数据和主存中数据不一致情况。可以在 cache 中设置一个标志及数据不一致信息，只有当 cache 中的数据被换出（cache 已满，该数据较长时间未访问）或强制执行“清空 cache”操作时，才会将原来更新的数据写入到主存中。

cache 清空操作如下：

① “清空”(clean): 把 cache 或 Write buffer 中的“脏”数据(cache 已更新, 未写入主)写入主存, 让其变干净。

② “使无效”(Invalidate): 使 cache 不能再使用, 并将脏数据写入到主存中。

S3C2440 内置了指令 Cache(ICaches)、数据 Cache(DCaches)、写缓存(Write buffer)等缓存技术。下面的内容用到了页表项中描述符的 C、B 位。如图 3-38 所示。为了区分 CP15 寄存器中 C 位, 描述符 C 称为 Ctt, B 位称为 Btt。

1. 指令 Cache(ICaches)

ICaches 的使用比较简单, 系统刚上电或复位时, ICaches 中的内容是无效的, 并且 ICaches 功能是关闭的, 将 C1 寄存器 I 位(CP15 寄存器 C1 第 12 位)写 1 置位, 可以启动 ICaches, 写 0 可以停止 ICaches。

ICaches 一般在 MMU 开启之后被使用, 此时页表中的描述符的 Ctt 位用来表示一段内存是否可以被 Cache, 若 Ctt 位=1, 则允许 Cache, 否则不允许被 Cache。但是, 即使 MMU 没有开启 ICaches, ICaches 也是可以被使用的, 这时 CPU 取指时所涉及的内存都被是允许 Cache 的。

ICaches 被关闭时, CPU 每次取指都要读取主存, 性能非常低。所以通常尽早启动 ICaches。

| | | | | | | | | | | | | | | |
|-------------------------|----|----|----|-----|-----|--------|-----|----|---|---|---|---|---|------------|
| 31 | 20 | 19 | 12 | 11 | 10 | 9 | 8 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Section base address | | | | AP | | Domain | | | 1 | C | B | 1 | 0 | Section |
| Large Page base address | | | | ap3 | ap2 | ap1 | ap0 | C | B | 0 | 1 | | | Large page |
| Small Page base address | | | | ap3 | ap2 | ap1 | ap0 | C | B | 1 | 0 | | | Small page |
| Tiny page base address | | | | | | | | ap | C | B | 1 | 1 | | Tiny page |

图 3-38 页表项描述符

ICaches 被开启后, CPU 每次取指都会先在 ICaches 中查看是否能找到所要的指令, 而不管 Ctt 位是否置位, 如果找到了, 称为 cache 命中(cache hit); 如果找不到, 称为 cache 丢失(cache miss)。ICache 被开启后, CPU 分为如下 3 种情况。

(1) Cache 命中且 Ctt 位为 1 时, 从 ICaches 中取指, 返回给 CPU。

(2) Cache 丢失, 且 Ctt 位为 1 时, CPU 从主存中读取指令, 同时, 一个被称为“8word line-fill”的动作将发生, 它会将当前指令所在区域的 8 个 word 写进 ICaches 的某个条目中。同样, 它可能会覆盖掉原先存在的条目, 可以通过一定算法选出某个未被锁定的条目进行覆盖。

2. 数据 Cache(DCaches)

与 ICaches 相似, 系统刚上电或复位时, DCaches 中的内容是无效的, 并且 DCaches 功能也是关闭着的, 而 Write buffer 中的内容也是被废弃不用的。往 CP15 寄存器 C1 Ccr 位 (CP15 寄存器 C13 第 2 位) 写 1 置位, 可以启动 DCaches, 写 0 可以停止 DCaches, Write buffer 与 DCaches 紧密结合, 没有专门的控制位来开启, 停止它。

与 ICaches 不同, DCaches 功能必须在 MMU 开启之后才能被使用, 因为开启 MMU 之后, 才能使用页表中的描述符来定义一块内存如何使用 DCaches 和 Write buffer。

DCaches 被关闭时, CPU 每次读写数据都要操作主存, DCaches 和 Write buffer 被完全忽略。

DCaches 被开启后, CPU 每次读写数据都会先在 DCaches 中查看是否能找到所要的数据, 而不管 Ctt 位是否置位, 如果找到了, 称为 Caches 命中(Cache hit); 如果找不到, 称为 Cache 丢失(Cache miss)。

与 TLB 类似, 使用 Cache 时需要保证 Cache、Write buffer 的内容和主存内容一致, 需要遵循如下两个原则:

- (1) 清空 DCaches, 使得主存数据得到更新。
- (2) 使无效 ICaches, 使得 CPU 取指时重新读取主存。

在编写程序时, 要注意以下几点:

(1) 开启 MMU 前,使无效 ICaches、DCaches 和 Write buffer。

(2) 关闭 MMU 前,清空 ICaches、DCaches,将“脏”数据写到主存中。

(3) 如果代码有更新,使无效 ICaches,这样 CPU 取指时会重新读取主存。

(4) 使用 DMA 操作可以被 Cache 的内存时,将内存的数据发送出去时,要清空 Cache,将内存的数据读入时,要使无效 Cache。

(5) 改变页表中地址映射关系时,也要慎重考虑。

(6) 开启 ICaches 或 DCaches 时,要考虑 ICaches 或 DCaches 中的内容是否与主存保持一致。

(7) 对于 I/O 地址空间,不使用 Cache 和 Write buffer。所谓 I/O 地址空间,就是对于其中的地址连续两次的写操作不能合并在一起,每次读/写操作都必须直接访问设备,否则程序运行结果无法预料。比如状态寄存器、非内存外设(扩展串口,网卡等)。

3.7.3 CP15 协处理器

在基于 ARM 的嵌入式应用系统中,对存储系统和管理通常是通过设置系统控制协处理器 CP15 来实现的。CP15 可以包含 16 个 32 位寄存器,其编号由 0~15。对于某些编号的寄存器可能对应多个物理寄存器,在指令中指定特定的标志位来区分这些物理寄存器。这种机制有些类似于 ARM 中的某些寄存器,当处于不同处理器模式时,某些 ARM 寄存器可能对应不同的物理寄存器,比如对于寄存器 SPSR,每一种处理器模式下都对应一个独立的物理寄存器(用户模式和系统模式对应同样的物理寄存器,这是个例外)。

CP15 中的寄存器可能是只读的也可能是只写的,还有一些是可以读写的,对于每一种寄存器下面将详细介绍:

- 寄存器的访问类型(只读/只写/读写)。
- 寄存器中每位的作用。
- 寄存器是否对应多个物理寄存器。
- 寄存器的具体作用。

(1) 访问 CP15 寄存器的指令有下面两种:

- MCR 从 ARM 寄存器到协处理器的数据传送指令。
- MRC 协处理器寄存器到 ARM 寄存器的数据传递指令。

MCR 和 MRC 指令只能在 CPU 特权模式下执行,在用户模式下执行 MCR 和 MRC 指令将会触发未定义指令的异常中断。

如果需要在用户模式下访问 CP15 中寄存器,必须先切换到特权模式下,再通过 MCR 和 MRC 对 CP15 寄存器进行操作。一种常见的作法是由操作系统定义的一些 SWI 调用接口,这些 SWI 调用完成对 CP15 的操作。在用户模式下可以调用这些 SWI 调用。

与常用 ARM 指令相比,MCR,MRC 指令格式相对比较复杂,指令格式如下:

MCR/MRC {cond} p15, opcode_1, Rd, CRn, CRm {, opcode_2}

cond 为指令执行条件码。当 cond 忽略时指令为无条件执行。

opcode_1 为协处理器将执行操作的操作码。对于 CP15 协处理器来说, opcode_1 永远为 0b000, 当 opcode 不为 0b000 时, 该指令操作结果不可预知。

Rd 为目标通用 ARM 寄存器, Rd 不能为 PC, 当其为 PC 时, 指令结果不可预知。

CRn 为协处理器的操作目标寄存器, 其编号可以为 C0~C15。

CRm 为附加的目标寄存器或源操作寄存器, 用于区分同一编号的不同物理寄存器。当指令中不需要提供附加信息时, 指令 CRm 为 C0 即可, 否则, 结果不可预知。

opcode_2 提供附加信息, 用于区别同一编号的不同物理寄存器。当指令中没有指定附加信息时, 省略 opcode_2 将其指令为 0, 否则操作结果不可预知。

(2) 协处理器 CP15 中的寄存器。CP15 提供了 16 个 32 位额外寄存器, 如表 3-21 所列。

表 3-21 CP15 寄存器说明

| 寄存器 | 名 称 | 访问方式 |
|-----|-------------------------|------|
| 0 | ID 寄存器 | 只读 |
| 1 | 控制 | 读/写 |
| 2 | 置换表 | 读/写 |
| 3 | 访问控制范围 | 读/写 |
| 4 | 保留位 | 无 |
| 5 | 默认状态 | 读/写 |
| 6 | 默认地址 | 读/写 |
| 7 | 缓存工作 | 只写 |
| 8 | TLB ⁽¹⁾ 运行 | 只写 |
| 9 | 缓存上锁 | 读/写 |
| 10 | TLB 上锁 | 读/写 |
| 11 | 保留位 | 无 |
| 12 | 保留位 | 无 |
| 13 | FCSE PID ⁽²⁾ | 读/写 |
| 14 | 保留位 | 无 |
| 15 | 测试结构 | 无 |

下面列出常用的几个 CP15 的寄存器。

● CP15 寄存器 0 - C0

其值为只读的, 里面包含有当前 CPU 的 ID 代码和缓存类型, 可以通过读取该寄存器里

的 ID 号,判断出当前 CPU 的型号。

当 opcode_2 域置 0 后,C0 里的值为 CPU 的 ID,如图 3-39 所示。

| | | | | | | | |
|------------|----|----|----|-------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| imp | | | | | | | |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| SRev | | | | archi | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| PNumber | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Layout Rev | | | | | | | |

图 3-39 CP15 寄存器 C0 (opcode_2 = 0)

Layout Rev[3:0]:包含处理器版本号

PNumber[15:4]:处理器部分序号,ARM920T 处理器的值为 0x920

Archi[19:16]:CPU 架构详细码,S3C2440 为 ARMV4 架构,其值为 0x2

SRev[23:20]:详细版本号,S3C2440 该值为 0x1

Imp[31:24]:设备代码,0x41(=A),表示 ARM 公司

当 opcode_2 域置 1 后,C0 里的值为缓存类型和架构信息,如图 3-40 所示。

| | | | | | | | |
|-------|----|----|-------|-------|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| 0 | 0 | 0 | ctype | | | | S |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| SRev | | | | DSize | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ISize | | | | | | | |

图 3-40 CP15 寄存器 C0(opcode_2 = 1)

ISize[11:0]:指令缓存大小

DSize[23:12]:数据缓存大小

S[24]:缓存

ctype[28:25]:缓存类型

● CP15 寄存器 1 - C1

C1 为 ARM920T 的控制寄存器,可以进行读写操作,其结构如图 3-41、图 3-42 所示。

M[0]:MMU 使能

0 = MMU 禁用 1 = MMU 使能

A[1]:地址对齐检查 0 = 对齐检查禁用 1 = 对齐检查使能

C[2]:DCache 使能 0 = DCache 禁用 1 = DCache 使能

B[7]:大小端设置 0 = 小端模式 1 = 大端模式

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| iA | nF | - | - | - | - | - | - |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| iA | RR | V | 1 | 0 | 0 | R | S |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| B | 1 | 1 | 1 | 1 | C | A | M |

图 3-41 CP15 寄存器 C1

S[8], R[9]:和页表项中描述符一起确定内存访问仅限

I[12]:ICache 控制 0 = ICache 禁用 1 = ICache 使能

V[13]:异常向量表地址 0 = 低地址, 0x0 1 = 高地址, 0xFFFF0000

RR[14]:Round Robin 置换 0 = 随机置换 1 = Round Robin 置换

[31:30]:时钟模式, 开启一系统时钟之后, FCLK 和 PCLK 时钟不在相等, 这时要修改时钟模式为异步模式, 因此将 C1[31:30]位设置为 0b11 即可。

| iA | nF | 时钟模式 |
|----|----|------|
| 0 | 0 | 快速总线 |
| 0 | 1 | 同步 |
| 1 | 0 | 保留 |
| 1 | 1 | 异步 |

图 3-42 CP15 寄存器 C1(系统总线模式)

● CP15 寄存器 2 - C2

C2 为页表基址寄存器 TTB, 里面存放的是页表的基址, 通过 MMU 虚拟地址转换机制的分析可以知道, 页表基址被放入到该寄存器中, MMU 会自动去该寄存器取出基址做 VA 地址到 MVA 地址的映射。

● CP15 寄存器 3 - C3(图 3-43)

该寄存器为域访问控制寄存器, 定义允许域访问, 每二位对应一个域, 总共包含 16 个域。

| | | | | | | | |
|-----|-----|-----|-----|-----|-----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

图 3-43 CP15 寄存器 C3

● CP15 寄存器 7 - C7(图 3-44)

C7 为缓存工作寄存器,用以管理指令缓存(I-Cache)与数据缓存(D-Cache)。每个缓存工作功能由 opcode_2 及使用写 CP15 C7 的 MCR 指令的 CRm 域选定。

| 功能 | 数据 | CRm | opcode_2 |
|-----------------------|-------|-----|----------|
| 等待中断 | SBZ | c0 | 4 |
| 使ICache无效 | SBZ | c5 | 0 |
| 使ICache单入口(使用MVA)无效 | MVA格式 | c5 | 1 |
| 使DCache无效 | SBZ | c6 | 0 |
| 使DCache单入口(使用MVA)无效 | MVA格式 | c6 | 1 |
| 使ICache与DCache无效 | SBZ | c7 | 0 |
| 清除DCache单入口(使用MVA) | MVA格式 | c10 | 1 |
| 清除DCache单入口(使用索引) | 索引格式 | c10 | 2 |
| 耗写缓冲器 | SBZ | c10 | 4 |
| 预取ICache线(使用MVA) | MVA格式 | c13 | 1 |
| 清除并使DCache入口无效(使用MVA) | MVA格式 | c14 | 1 |
| 清除并使DCache入口无效(使用索引) | 索引格式 | c14 | 2 |

图 3-44 CP15 寄存器 C7(相关操作)

● CP15 寄存器 8 - C8

C8 为 TLB 工作寄存器(转换后备缓冲器工作寄存器),用于管理指令 TLB 与数据 TLB。

● CP15 寄存器 13 - C13(图 3-45)

C13 为快速上下文切换寄存器 FCSE PID。其内容为进程的 PID,用于用户进程虚拟地址映射。

| | | | | | | | |
|---------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| FCSEPID | | | | | | | - |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| - | - | - | - | - | - | - | - |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| - | 1 | - | - | - | - | - | - |

图 3-45 CP15 寄存器 C13

3.8 实战:小型多任务操作系统 miniOS 的实现

本实验源码包含三部分:

miniOS 源码:存放在光盘目录\work\armarch\miniOS\miniOS_XXXX(开发板名)工程目录下。

miniOS 应用程序跑马灯:存放在光盘目录\work\armarch\miniOS\miniOS_app_led 工程目录下。

miniOS 应用程序打印程序:存放在光盘目录\work\armarch\miniOS\miniOS_app_print 工程目录下。

所有可执行二进制文件在\work\armarch\miniOS\image 目录下。

操作步骤:

(1) 在 ADS 下编译 miniOS_xxxx 工程,选择 Norflash 启动,然后通过 H-JTAG 将 minios_org.bin 烧写到 Norflash 中 0 地址处。

(2) 编译应用程序 miniOS_app_led 和 miniOS_app_print,选择 Norflash 启动,通过 H-Jtag 将 miniOS_app_print.bin 烧写到 0x10000 地址处,将 miniOS_app_led.bin 烧写 0x20000 地址处。

(3) 启动开发板,可以看到 miniOS 启动信息。miniOS 最多同时支持 60 个进程执行,miniOS 启动完毕之后,默认没有系统内核进程在执行,可以通过按 K1 键,新创建进程,最多创建 60 个,按 K2 键,随机杀死一个进程,按 K6 键可以启动跑马灯进程。

(4) 如果读者想写一个程序,让 miniOS 启动它,则根据编写规则,编写程序,如果需要系统调用,还要自己实现系统调用接口。

3.8.1 miniOS 代码分析

该程序文件,是 miniOS 运行时最先执行的代码,它主要用来安装异常向量表,初始化必要的硬件,然后将代码从 Norflash 中复制到 SDRAM,然后跳转到 SDRAM 中去执行,最后调用 main 函数,开始 miniOS 启动第二阶段。

```

;
; start.S:主要安装异常向量表,初始化必要的硬件,然后将代码从 Norflash 中复制到 SDRAM,
; 然后跳转到 SDRAM 中去执行,最后调用 main 函数,开始 miniOS 启动第二阶段
INCLUDE common_asm.h ; 定义了开发板内存起始地址和系统栈指针
; 以下为时钟相关寄存器地址
LOCKTIME EQU 0x4c000000
MPLLCON EQU 0x4c000004
CLKDIVN EQU 0x4c000014
RUN_BASE EQU 0x33ff0000 ; OS 内存运行地址
MEM_REG_BASE EQU 0x48000000 ; 内存控制寄存器开始地址,用于进行循环进行内存初始化
MEM_REG_END EQU 0x48000034 ; 内存控制寄存器线束地址

IMPORT |Image$ $RO$ $Base| ; 引入链接器自动生成符号变量 Image$ $RO$ $Base
IMPORT |Image$ $ZI$ $Base| ; 引入链接器自动生成符号变量 Image$ $ZI$ $Base
IMPORT |Image$ $ZI$ $Limit| ; 引入链接器自动生成符号变量 Image$ $ZI$ $Limit

```



```

IMPORT HandleSWI
IMPORT disable_watch_dog
IMPORT CopyCode2Ram
IMPORT xmain
IMPORT handle_irq
IMPORT undef_excp
IMPORT prefetch_abt
IMPORT data_abt

AREA    Start, CODE, READONLY

ENTRY
b    Reset

ldr    pc, = HandleUndef
ldr    pc, = HandleSWI
ldr    pc, = HandlePrefetchAbort
ldr    pc, = HandleDataAbort
HandleNotUsed
b    HandleNotUsed
b    HandleIRQ
HandleFIQ
b    HandleFIQ

Reset
bl    clock_init
bl    mem_init
ldr    sp, = SVC_STACK
bl    disable_watch_dog

copy_code

```

; 以下引入其他文件中声明函数名
 ; 代码段开始
 ; 异常向量表,其运行地址为 0,pc 自动由硬件设置
 ; 该地址处指令为一跳转指令,跳往 reset 异常处理
 ; 未定义异常处理跳转指令,跳往 HandleUndef 处
 ; 软件中断异常处理跳转指令,跳往 HandleSWI 处
 ; 预取指令中止异常处理跳转指令,
 ; 跳往 HandlePrefetchAbort 处
 ; 数据中止异常处理跳转指令,跳往 HandleDataAbort 处
 ; 未使用异常处理跳转指令,没有处理
 ; 一般中断异常处理跳转指令,跳往本源文件中
 ; HandleIRQ 符号处
 ; 快速中断异常处理跳转指令,没有处理
 ; Reset 异常处理符号
 ; 跳往时钟初始化处理
 ; 跳往内存初始化处理
 ; 设置管理模式栈指针
 ; 关闭看门狗
 ; 代码复制,将代码复制到内存里去运行,如果从 Nandflash
 ; 启动运行,其 RAM steppingstone 只有 4KB,不足已运行全部代
 ; 码,如果从 Norflash 启动,其硬件特性决定其运行速度较慢,
 ; 因此,将代码复制到内存里去运行
 ; 代码复制开始符号

```

mov    r0, #0x0                                ; R0 中为数据开始地址 (ROM 数据保存在 0 地址开始处)

ldr    r1, = |Image$ $ RO$ $ Base| ; R1 中存放 R0 输出域运行地址,
                                           ; 该值由符号变量 Image$ $ RO$ $ Base 取得

ldr    r2, = |Image$ $ ZI$ $ Limit| ; R2 中存放 ZI 输出域结束地址,
                                           ; 该值由符号变量 Image$ $ ZI$ $ Limit 取得

sub    r2, r2, r1                                ; R2 = R2 - R1, 得出待复制数据长度
bl     CopyCode2Ram                             ; 将 R0, R1, R2 三个参数传递给 CopyCode2Ram 函数执行复制

ldr    r0, = |Image$ $ ZI$ $ Base|
ldr    r1, = |Image$ $ ZI$ $ Limit|
bl     clear_bss_region

bl     stack_init                               ; 跳往栈初始化代码处, 初始化所有模式下栈指针

msr    cpsr_c, #0x5f                            ; 开启系统中断, 进入系统模式
ldr    lr, = halt_loop                          ; 设置返回地址
ldr    pc, = xmain                              ; 跳往 main 函数, 进入 OS 启动处理
halt_loop                                     ; OS 返回地址, 其实这儿永远不可能被执行到,
                                           ; 因为只要 OS 工作, 它就会运行到世界末日

b      halt_loop                               ; 死循环

HandleIRQ                                     ; 系统中断处理
sub    lr, lr, #4                               ; 修正返回地址
ldr    sp, = IRQ_STACK_BASE                    ; 设置中断模式下栈指针
stmdb  sp!, {r0-r12, lr}                       ; 保存现场
ldr    lr, = int_return                        ; 设置中断处理程序的返回地址
ldr    pc, = handle_irq                        ; 跳往中断处理程序
int_return                                     ; 返回地址
ldmia  sp!, {r0-r12, pc}~                      ; 恢复被中断打断现场

clock_init                                   ; 时钟初始化代码
; Set lock time
ldr    r0, = LOCKTIME
ldr    r1, = 0x00ffffff
str    r1, [r0]

```

资源解密网
PDG

```
; Set clock divison
```

```
ldr r0, = CLKDIVN
```

```
mov r1, # 0x05
```

```
str r1, [r0]
```

```
; 设置系统总线为异步模式
```

```
mrc    p15, 0, r1, c1, c0, 0
```

```
orr    r1, r1, # 0xc0000000
```

```
mcr    p15, 0, r1, c1, c0, 0
```

```
ldr r0, = MPLLCON
```

```
ldr r1, = 0x5c011
```

```
; MPLL is 400MHz
```

```
str r1, [r0]
```

```
mov pc, lr
```

```
mem_init
```

```
; 内存初始化代码
```

```
ldr r0, = MEM_REG_BASE
```

```
ldr r1, = MEM_REG_END
```

```
adr r2, memdata
```

```
mem_init_loop
```

```
ldr r3, [r2], # 4
```

```
str r3, [r0], # 4
```

```
teq r0, r1
```

```
bne mem_init_loop
```

```
mov pc,lr
```

```
memdata
```

```
DCD 0x22111110 ;BWSCON
```

```
DCD 0x00000700 ;BANKCON0
```

```
DCD 0x00000700 ;BANKCON1
```

```
DCD 0x00000700 ;BANKCON2
```

```
DCD 0x00000700 ;BANKCON3
```

```
DCD 0x00000700 ;BANKCON4
```

```
DCD 0x00000700 ;BANKCON5
```

```
DCD 0x00018005 ;BANKCON6
```

```
DCD 0x00018005 ;BANKCON7
```

```
DCD 0x008e04f4 ;REFRESH
```

```
DCD 0x000000b1 ;BANKSIZE
```

```
DCD 0x00000030 ;MRSRB6
```

蘇子知覺
PDG

```
DCD 0x00000030      ;MRSRB7
```

```
clear_bss_region
```

```
    mov r2, #0
```

```
clear_loop
```

```
    cmp r0, r1
```

```
    beq quit_loop
```

```
    str r2, [r0], #4
```

```
    b clear_loop
```

```
quit_loop
```

```
    mov pc, lr
```

```
stack_init
```

```
; 栈指针初始化
```

```
    ; undefine_stack
```

```
; 未定义异常
```

```
    msr    cpsr_c, #0xdb
```

```
    ldr    sp, =EXCPT_STACK_BASE
```

```
    ; abort_stack
```

```
; 未定义异常模式
```

```
    msr    cpsr_c, #0xd7
```

```
    ldr    sp, =EXCPT_STACK_BASE
```

```
    ; irq_stack
```

```
; 中断模式
```

```
    msr    cpsr_c,    #0xd2
```

```
    ldr    sp, =IRQ_STACK_BASE
```

```
    ; sys_stack
```

```
; 系统模式
```

```
    msr    cpsr_c,    #0xdf
```

```
    ldr    sp, =SYS_STACK_BASE
```

```
    msr    cpsr_c,    #0xd3
```

```
    mov    pc, lr
```

```
HandleUndef
```

```
; 未定义异常处理
```

```
    add lr, lr, #4
```

```
; 修正返回地址
```

```
    stmdb  sp!, {r0 - r15}
```

```
; 保存现场
```

```
    mrs r0, cpsr
```

```
; 发生异常时,将状态寄存器里的数据保存在栈里
```

```
    mrs r1, spsr
```

```
    stmdb  sp!, {r0, r1}
```

```
    mov r0, sp
```

```
; 发生异常时,将栈指针传递给异常处理函数
```

```
; 用于打印异常现场信息
```

```
    ldr pc, =undef_excp
```

```
    b    halt_loop
```

资源解密

PDG

```

HandlePrefetchAbort                                ; 未定义异常处理
    sub lr, lr, #4                                  ; 修正返回地址
    stmdb sp!, {r0-r15}                             ; 保存现场
    mrs r0, cpsr                                    ; 发生异常时,将状态寄存器里的数据保存在栈里
    mrs r1, spsr
    stmdb sp!, {r0, r1}
    mov r0, sp                                       ; 发生异常时,将栈指针传递给异常处理函数
                                                    ; 用于打印异常现场信息

    ldr pc, = prefetch_abt
    b    halt_loop

HandleDataAbort                                    ; 未定义异常处理
    sub lr, lr, #8                                  ; 修正返回地址
    stmdb sp!, {r0-r15}                             ; 保存现场
    mrs r0, cpsr                                    ; 发生异常时,将状态寄存器里的数据在栈里
    mrs r1, spsr
    stmdb sp!, {r0, r1}
    mov r0, sp                                       ; 发生异常时,将栈指针传递给异常处理函数
                                                    ; 用于打印异常现场信息

    ldr pc, = data_abt
    b    halt_loop

END

```

init.c:

本程序源码主要用于被 start.S 调用,实现开发板启动例程操作,关闭看门狗,从 ROM 复制代码到 SDRAM。

```

#include "serial.h"
#include "init.h"
#include "mmu.h"

/* 关闭看门狗,默认看门狗是打开的 */
void disable_watch_dog(void)
{
    WTCON = 0;
}

/*****
* 本函数用来检测是否是从 Norflash 启动

```

资源解密

PDG

- * 常见的启动介质有 Norflash 和 Nandflash 两种,当开发板
- * 选择从 Nand 启动时,由于 Nand 控制器中将 Nand 前 4KB 代码复制到
- * SRAM 类型的 Stepping stone,SRAM 是可读写存储器,可以通过指令
- * 实现写入操作,而 Norflash 是 ROM 类型存储器,只能读取数据,执行
- * 执行写入操作时,要借助特定程序实现,因此可以通过向介质中写入数据
- * 然后再读取出来,看是否一致,即可判断是否是从 Norflash 启动
- * 0 地址处由于安装有异常向量表,而"b Reset"指令对应的二进行码为:0xEA00000B
- * 通过向 0 地址处写入数据 0x12345678,然后再读取出来,看是否是 0x12345678
- * 如果是 0x12345678 则说明是从 Nand 启动,如果是 0xEA00000B 说明,没有写入成功,则
- * 是从 Norflash 启动

*****/

```
int isBootFrmNORFlash(void)
```

```
{
    volatile unsigned long * pdw = (volatile unsigned long *)0;
    unsigned long dwVal;

    dwVal = *pdw;
    *pdw = 0x12345678;
    if (*pdw != 0x12345678) {
        return 1;
    } else {
        *pdw = dwVal;
        return 0;
    }
}
```

/*****/

* 代码复制函数

- * 功能:判断当前启动方式,如果是从 Norflash 启动,则将代码复制到 SDRAM 内存中
- * Norflash 是 ROM,只能读取数据不能执行写入操作,并且在 Norflash 中执行速度
- * 较慢,因此必须要将其复制到 SDRAM 中

* 参数:

- * unsigned long start_addr; 开启复制地址
- * unsigned char * buf; 目标地址(内存区域)
- * int size; 复制数据大小(字节)

*****/

```
extern int CopyCode2Ram(unsigned long start_addr, unsigned char * buf, int size)
```

```
{
```

```

unsigned long * pdwDest;
unsigned long * pdwSrc;
unsigned long i;

if (isBootFrmNORFlash()) {
    pdwDest = (unsigned long *)buf;
    pdwSrc  = (unsigned long *)start_addr;
    /* copy code from NOR Flash to RAM */
    for (i = 0; i < size / 4; i++) {
        pdwDest[i] = pdwSrc[i];
    }
    return 0;
} else {
    return 0;
}
}

```

mmu.c

pgtb_init()函数:

本函数主要建立 miniOS 整个地址空间映射关系,用于支持多进程和对不同进程进行权限管理。

miniOS 采用段描述符,每段控制管理 1MB 地址空间,页表存放在物理内存最开始的 1MB 空间范围内(0x300F0000 — 0x300F3FFF),共 16KB,如图 2-46 所示。

(1) 最高地址 1MB 用来存放异常向量表和 miniOS 代码,它在 miniOS 第一阶段从 Nor-flash 中复制到 SDRAM 0x33FF0000 地址处。

(2) 为了方便 miniOS 对整个内存区域进行管理,将 0x30000000~0x34000000 虚拟地址映射向本身物理地址,也就是说虚拟地址和物理地址是一样的,这样的话,在 MMU 启动前后,miniOS 代码不用跳转,直接可以正常执行。

(3) 除了开始 1MB 和最后 1MB 分别给了页表和异常向量表,其余 62MB 空间分配作为用户进程空间。由 MMU 章节,可知用户程序执行地址空间为 0~32MB,为了区分不同进程的地址空间,通过 PID 左移 25 位(乘以 32MB)方式区分不同地址空间,因此,将每个进程空间映射向物理内存 1MB 地址空间,这样最多可能支持 63 个进程(包含内核进程),其映射关系如下:

PID * 32MB 虚拟地址空间 -> PID * 1MB 物理地址空间

当 PID = 24 和 25 时,虚拟地址空间为 0x30000000~0x33FFFFFFF,物理地址空间为 0x31800000~0x319FFFFFFF,由于该映射关系在前面已经存在(第二条),所以该虚拟地址空间

对应 PID 进程不可用,因此其对应 2MB 物理空间可以留作它用。

(4) 特殊功能寄存器地址空间为 $0x48000000 \sim 0x60000000$,其地址空间和用户进程虚拟地址空间有重合,因此将寄存器地址空间做映射,将其物理地址映射向 $register_address + 0x80000000$ 地址处。这样其可以操作的虚拟地址空间为 $0xC8000000 \sim 0xE0000000$ 。

| 虚拟地址 | | 物理地址 | |
|------------|-------------------------------|------------|-------|
| 0xFFFFFFFF | miniOS代码 (64K, 含0号进程代码和异常向量表) | 0x33FFFFFF | 1 MB |
| 0xFFFF0000 | | 0x33FF0000 | |
| 0xFFFEFFFF | | 0x33FEFFFF | |
| 0xFFF00000 | | 0x33F00000 | |
| | 异常模式堆栈区 (960K) | | |
| | 进程 62 物理空间 | 0x33EFFFFF | 64 MB |
| 0x7C000000 | | 0x33E00000 | |
| | 进程 61 物理空间 | 0x33DFFFFF | |
| 0x7A000000 | | 0x33D00000 | |
| | 进程 27~60 物理空间 | 0x33CFFFFF | |
| 0x36000000 | | 0x31B00000 | |
| | 进程 26 物理空间 | 0x31AFFFFF | |
| 0x34000000 | | 0x31A00000 | |
| | 未使用的进程 25 空间 | 0x319FFFFF | |
| 0x32000000 | | 0x31900000 | |
| | 未使用的进程 24 空间 | 0x318FFFFF | |
| 0x30000000 | | 0x31800000 | |
| | 进程 23 物理空间 | 0x317FFFFF | |
| 0x2E000000 | | 0x31700000 | |
| | 进程 2~22 物理空间 | 0x316FFFFF | |
| 0x04000000 | | 0x30200000 | |
| | 进程 1 物理空间 | 0x301FFFFF | |
| 0x02000000 | | 0x30100000 | |
| | 空闲 48K, 页表 16K | 0x300FFFFF | |
| | 系统模式堆栈区 (960K) | 0x30000000 | |

图 3-46 miniOS 地址空间映射

由 MMU 权限管理知识可知,段描述符低 12 位为对应段地址空间的权限,其主要包括: [11:10]为 AP(控制访问权限),[8:5]域, [3:2]=CB(cache 和缓存设置位),[1:0]页表项描述符标识位, AP 权限为特权与用户模式下读写权限设定,如果段对应域设置为 0b01 的用户模式,则要设置对应段描述符中的 AP 位,我们使用域 1,域 0 设置为管理员权限域,域 0 设置为用户权限域(见 mmu_init 函数)因此 [8:5] = 0b0,开启 Write Buffer,设置 CB 位为

0b1,页表项描述符选择段描述符 0b10。

mmu_init()函数:

本函数主要设置异常向量表位置,开启 ICaches,DCaches,写入页表基址,开启 MMU 映射等。

(1) 异常向量表默认在 0 地址处,由于开启 MMU 之后,CPU 看到的都是虚拟地址,0 地址被看做是进程空间地址,不同进程空间被 MMU 映射到不同内存,为了解决开启 MMU 之后,能够正常处理异常,ARM920T 提供了,修改异常向量表位置功能,当设置了 CP15 寄存器 C1 的 V 位后,异常发生时,CPU 会自动去 0xFFFF0000 地址处执行异常处理。而开启 MMU 后,根据虚拟地址与物理地址映射规则可知,0xFFFF0000 地址高 12 位 0xFFF 为页表偏移地址,低 20 位 0xF0000 为页表映射基址的偏移地址,因此,0xFFFF0000 映射到 0x33FF0000 内存地址处,miniOS 将全部代码复制到 0x33FF0000(还有 64Kb 就到达内存总容量,64K 空间足够存放内核代码)。

(2) miniOS 页表放置在 0x300F0000 处,因此要将页表地址告诉 ARM920T 用于地址映射,通过设置 CP15 寄存器 C2,设置页表。

(3) 内存权限域控制设置,通过设置 CP15 寄存器 C3,来设置最多 16 个域用于管理不同时地址空间。代码中设置 Domain0 为用户模式,其他域为管理员模式,miniOS 使用简单的管理员模式 0b11。

```
#include      "s3c2440.h"
#include      "serial.h"
#include      "mmu.h"
#define MANAGER_AP      (0x3<<10)
#define USER_AP      (0x1<<10)
#define MANAGER_DOMAIN      (0x1<<5)
#define USER_DOMAIN      (0x0<<5)
#define CB      (0x1<<2)
#define DESC_FLAG      (0x2<<0)
#define SEC_DESC      (MANAGER_AP | MANAGER_DOMAIN | CB | DESC_FLAG)
static unsigned long * mmu_tlb_base = (unsigned long *) MMU_TABLE_BASE;

/*****
* 页表建立函数
* -----
* 段页表项 entry:[31:20]段基址,[11:10]为 AP(控制访问权限),[8:5]域,
* [3:2] = CB(decide cached & buffered),[1:0] = 0b10 --> 页表项为段描述符
* 1. 将页表放在最开始 1M 地址空间的最后 64K,即:MMU_TABLE_BASE = 0x300F0000(in mmu.h)
* 2. 对于 64M SDRAM,其物理地址为 0x30000000~0x33f00000,令其虚拟地址 = 物理地址
```

```

* 这样内核空间代码可以正常运行,栈指针等不用再次映射
* 3. 对于 SFR,其物理地址为 0x48000000~0x60000000,令其虚拟地址 = 物理地址 + 0x80000000
* 4. 由于虚拟地址 0x0 为用户进程的虚拟地址空间,不能再在该处存放异常向量表,所以需要将
* 异常向量表放置到高端虚拟地址 0xFFFF0000 处。异常向量表通过设置 CP15,C1 寄存器,选择
* 使用高端地址,当异常发生时,会跳转到 0xFFFF0000。0xFFFF0000 是虚拟地址,将虚拟
* 地址 0xFFFF0000 映射到物理地址 0x33FF0000
* 5. 每个进程使用 1MB 内存空间,除去最高 1MB 空间留给内核代码
* 其余以 PID 为进程号的进程空间块的虚拟地址为:PID * 0x02000000
* 到 PID * 0x02000000 + 0x01FFFFFF
* 6. 进程 0 为内核进程,其位于物理地址:0x33FF0000~0x33FFFFFF
*
*****/
void pgth_init()
{
    unsigned long entry_index, SFR_base;

    /* 建立到 Norflash 的 2MB 的地址空间的映射 */
    /* 0xA0000000 映射到 0 开始的 1MB 地址空间 */
    * (mmu_tlb_base + (0xA0000000 >> 20)) = 0x0 | SEC_DESC;
    /* 0xA0100000 映射到 0x100000~0x1FFFFFF 的 1MB 地址空间 */
    * (mmu_tlb_base + (0xA0100000 >> 20)) = 0x100000 | SEC_DESC;

    /*
    * 令 0x30000000~0x34000000 的 64MB 虚拟地址等于物理地址空间,
    * 方便 miniOS 内部进程管理
    */
    for(entry_index = 0x30000000; entry_index < 0x34000000; entry_index += 0x100000) {
        * (mmu_tlb_base + (entry_index >> 20)) = entry_index | SEC_DESC;
    }

    /* 寄存器地址空间 0x48000000~0x60000000 映射到虚拟地址 0xC8000000~0xE0000000 */
    for(entry_index = 0x48000000 + 0x80000000, SFR_base = 0x48000000;
        SFR_base < 0x60000000; entry_index += 0x100000, SFR_base += 0x100000) {
        * (mmu_tlb_base + (entry_index >> 20)) = SFR_base | SEC_DESC;
    }

    /*
    * 进程 1~23 号进程地址空间,每个进程 32MB,miniOS 允许进程使用 32MB 虚拟地址空间,
    * 但是只分配其 1MB 的实际物理空间
    */

```



```

* 进程 1:物理地址空间 0x30100000~0x301ffffff,对应 MVA(修正虚拟地址,
* 进程 PID<<25 形成)
*      MVA 地址空间:0x02000000~0x021ffffff
* 进程 2:物理地址空间 0x30200000~0x302ffffff
*      MVA 地址空间:0x04000000~0x041ffffff
*      ...
* 进程 23:物理地址空间 0x31700000~0x317ffffff
*      MVA 地址空间:0x2E000000~0x2E1ffffff
* 对应进程 24 的 MVA 地址空间 0x30000000 已被映射,因此该 MVA 不能再次被映射,
* 跳过进程 24,同样道理,跳过进程 25
*
* /
for(entry_index = 1; entry_index < 24; entry_index++){
    *(mmu_tlb_base + ((entry_index * 0x02000000)>>20)) =
        (entry_index * 0x00100000 + SDRAM_BASE) | SEC_DESC;
}

/*
* 进程 26:物理地址空间 0x31A00000~0x31Affffff
*      MVA 地址空间:0x34000000~0x35ffffff
*      ...
* 进程 62:物理地址空间 0x33E00000~0x33Effffff
*      MVA 地址空间:0x7C000000~0x7Dffffff
* /
for(entry_index = 26; entry_index < TASK_SZ; entry_index++){
    *(mmu_tlb_base + ((entry_index * 0x02000000)>>20)) =
        (entry_index * 0x00100000 + SDRAM_BASE) | SEC_DESC;
}

/*
* 异常向量表
* 0xFFFF0000 为高地址异常向量表,可以通常设置 CP15,C1 寄存器 V 位,当异常产生时,
* 由硬件自动去 0xFFFF0000 地址处执行异常跳转指令,而不是之前的 0 地址处异常向量
* 表跳转,将该虚拟地址映射到 0x33F00000 这 1MB 地址空间,0xFFFF0000 虚拟地址会
* 通过 MMU 的映射关系映射为 0x33FF0000,而在 start.S 中已经将 miniOS 的代码复制到了
* 该地址处,这样当异常产生时,能够正常执行异常向量表的跳转指令。
* /
*(mmu_tlb_base + (0xffff0000>>20)) = ((VECTORS_PHY_BASE) | SEC_DESC);
}

```

```

/*****
* MMU 初始化函数
*****/
#define VECTOR      (1<<13)    // 设置异常向量表位置,0 = 低地址 0x0 1 = 高地址 0xFFFF0000
#define ICACHE      (1<<12)    // 设置 ICACHE,0 = 禁用 1 = 使用
#define R_S_BIT     (3<<8)     // 和页表项中描述符一起确定内存访问权限
#define ENDIAN      (1<<7)     // 确定系统使用大,小端字节序,0 = 小端模式 1 = 大端模式
#define DCACHE      (1<<2)     // 设置 DCACHE,0 = 禁用 1 = 使用
#define ALIGN       (1<<1)     // 设置地址对齐检查,0 = 禁用 1 = 使用
#define MMU_ON      (1<<0)     // 设置 MMU,0 = 禁用 1 = 使用

void mmu_init()
{
    unsigned long ttb = MMU_TABLE_BASE;
    /* reg1 待清除位 */
    int reg0, reg1 = (VECTOR | ICACHE | R_S_BIT | ENDIAN | DCACHE | ALIGN | MMU_ON);
    /*
    * CP15,C1 设置位:异常向量表设置在高地址,使用 ICACHE,系统采用小端模式,
    * 使用 DCACHE,使用地址对齐检查,开启 MMU
    */
    int CP15_C1_set = (VECTOR | ICACHE | DCACHE | ALIGN | MMU_ON);
    __asm{
        mov     reg0, #0
        /* 使 ICaches 和 DCaches 无效 */
        mcr     p15, 0, reg0, c7, c7, 0
        /* 使能写入缓冲器 */
        mcr     p15, 0, reg0, c7, c10, 4
        /* 使指令,数据 TLB 无效无效 */
        mcr     p15, 0, reg0, c8, c7, 0
        /* 页表基址写入 C2 */
        mcr     p15, 0, ttb, c2, c0, 0
        /* 将 0x2 取反变成 0xFFFFFFF, Domain0 = 0b01 为用户模式,其他域为 0b11 管理模式 */
        mvn     reg0, #0x2
        /* 写入域控制信息 */
        mcr     p15, 0, reg0, c3, c0, 0
        /* 取出 C1 寄存器中值给 reg0 */
        mrc     p15, 0, reg0, c1, c0, 0
        /* 先清除不需要的功能,现开启 */
        bic     reg0, reg0, reg1
        /* 设置相关位并开启 MMU */
    }
}

```

```

    orr    reg0, reg0, CP15_C1_set
    mcr    p15, 0, reg0, c1, c0, 0
}
}

```

main.c

本程序文件,实现 miniOS 启动第二阶段,主要实现建立页表,开启 MMU,和一些 miniOS 运行过程中使用的硬件初始化。由于特殊寄存器的地址在开启 MMU 之后只能使用映射后的虚拟地址进行访问,因此对一些硬件的初始化操作放到了 MMU 开启之后,像 UART, KEY, LED, Timer0 定时器等,它们在 miniOS 启动之后都会被频繁使用。

```

#include "s3c2440.h"
#include "serial.h"
#include "swi.h"
#include "init.h"
#include "led.h"
#include "mmu.h"
#include "interrupt.h"

```

```

extern void key_init(void);
extern void sched_init(void);
__inline void wait(unsigned long dly)
{
    unsigned long i;
    for( i = dly; i > 0; i-- );
}

```

```
int xmain(void)
```

```
{
```

```

    pgtb_init();
    mmu_init();
    uart_init();
    irq_init();
    Timer0_init();
    key_init();
    led_init();
    PRINT_OS_INRO();
    OS_ENTER_CRITICAL();
    sched_init();
    OS_EXIT_CRITICAL();

```

```

// 建立页表
// MMU 初始化
// 串口初始化
// 中断初始化
// 定时器 0 初始化
// 按键初始化
// LED 灯初始化
// 打印 OS 信息
// 关闭中断,准备进入进程初始化函数
// 进程调度初始化
// 开启中断

```

```
// 调试信息
DPRINTK(KERNEL_DEBUG, "kernel: Enter user mode to run");
ENTER_USR_MODE();           // 进入用户模式
// 进程 0 执行内容
while(1){
    DPRINTK(KERNEL_DEBUG, "kernel: process 0");
    printk("process 0, idle");
    wait(1000000);
}
return 0;
}
```

serial.c:

本程序文件实现了串口 0 的初始化,用于打印系统和用户信息,由于串口在开启 MMU 之后初始化,所以相关寄存器使用的虚拟地址对应寄存器。

```
#include "s3c2440.h"
#include "serial.h"

#define TXD0READY (1<<2)
#define RXD0READY (1)

/*****
* 串口初始化代码
*****/
void uart_init( )
{
    VA_GPHCON |= 0xa0;           //GPH2,GPH3 used as TXD0,RXD0
    VA_GPHUP   = 0x0;           //GPH2,GPH3 内部上拉

    VA_ULCON0   = 0x03;         //8N1
    VA_UCON0    = 0x05;         //查询方式为轮询或中断;时钟选择为 PCLK
    VA_UFCON0   = 0x00;         //不使用 FIFO
    VA_UMCON0   = 0x00;         //不使用流控
    VA_UBRDIV0  = 26;           //比特率为 115200,PCLK = 50MHz
//    UBRDIV0 = 53;             //比特率为 57600,PCLK = 50MHz
//    UBRDIV0 = 6;              //比特率为 115200,PCLK = 12MHz,会超出 baudrate 的误差容忍范围
//    UBRDIV0 = 12;             //比特率为 57600,PCLK = 12MHz
    DPRINTK(KERNEL_DEBUG, "init Uart OK");
}
```

```

/*****
 * 从串口打印单个字符
 *****/
extern void putc(unsigned char c)
{
    while( !(VA_UTRSTAT0 & TXDREADY) );
    VA_UTXH0 = c;
}

/*****
 * 从串口接收单个字符
 *****/
extern unsigned char getc(void)
{
    while( !(VA_UTRSTAT0 & RXDREADY) );
    return VA_URXH0;
}

/*****
 * 打印字符串
 *****/
extern int printk(const char * str)
{
    int i = 0;
    while( str[i] ){
        putc( (unsigned char) str[i++] );
    }
    putc('\n');
    putc('\r');
    return i;
}

```

key.c:

按键初始化程序代码。使用虚拟寄存器地址,分别初始化 mini2440 的 6 个按键。

```

#include "s3c2440.h"
#include "serial.h"

```




```

/*****
 * 按键初始化代码
 *****/
void key_init(void)
{
    VA_GPFCON = (2 << 0) | (2 << 4);
    VA_GPGCON = (2 << 6) | (2 << 22);
    VA_EXTINT0 = (3 << 8) | (3 << 0);
    VA_EXTINT1 = (3 << 12);
    VA_EXTINT2 = (3 << 12);
    DPRINTK(KERNEL_DEBUG, "key_init OK");
}

```

led.c:

LED 灯初始化程序代码, 使用虚拟寄存器地址, 分别初始化 mini2440 的 4 个 LED 灯, 提供了几个控制 LED 亮, 灭的函数。

```

#include    "s3c2440.h"
#include    "serial.h"
#include    "led.h"

#define    ALL_LED_ON    (1 << 5 | 1 << 6 | 1 << 7 | 1 << 8)
#define    ALL_LED_OFF    (0)
#define    LED_BIT    (1 << 5 | 1 << 6 | 1 << 7 | 1 << 8)
#define    LED1_ON    (1 << 6 | 1 << 7 | 1 << 8)
#define    LED2_ON    (1 << 5 | 1 << 7 | 1 << 8)
#define    LED3_ON    (1 << 5 | 1 << 6 | 1 << 8)
#define    LED4_ON    (1 << 5 | 1 << 6 | 1 << 7)

```

/* led 初始化 */

```

void led_init(void){
    VA_GPBCON = 0x15400;
    all_led_on();
    DPRINTK(KERNEL_DEBUG, "led_init");
}

```

/* 根据参数 led_no 点亮对应灯 */

```

void led_on(int led_no){
    switch(led_no){
        case 1:

```

资源解密 PDG

```

        VA_GPB DAT    = (VA_GPB DAT & ~LED_BIT) | LED1_ON;
        break;
    case 2:
        VA_GPB DAT    = (VA_GPB DAT & ~LED_BIT) | LED2_ON;
        break;
    case 3:
        VA_GPB DAT    = (VA_GPB DAT & ~LED_BIT) | LED3_ON;
        break;
    case 4:
        VA_GPB DAT    = (VA_GPB DAT & ~LED_BIT) | LED4_ON;
        break;
    default:
        printk("error led number!!");
    }
}

```

/* 点亮全部 LED 灯 */

```

void all_led_on(void){
    VA_GPB DAT    = (VA_GPB DAT & ~ALL_LED_ON);
}

```

/* 关闭全部 LED 灯 */

```

void all_led_off(void){
    VA_GPB DAT    = (VA_GPB DAT | ~ALL_LED_OFF);
}

```

irq_init.c:

系统中断初始化程序代码,使用虚拟寄存器地址,开启了按键中断,定时器 0 中断。

```

#include "init.h"
#include "serial.h"
#include "s3c2440.h"

```

```

void irq_init(void)
{
    VA_INTMSK = 0xffffffff;
    VA_EINTMASK = 0xffffffff;
    VA_INTMSK &= ~(1<<10) | (1<<5) | (1<<2) | (1<<0);
    VA_EINTMASK &= ~(1<<11) | (1<<19);
    DPRINTK(KERNEL_DEBUG, "Interrupt init OK");
}

```

}

time0_init.c:

定时器 0 初始化程序代码,使用虚拟寄存器地址,每 10ms 产生一次时钟中断,用来为 OS 调度程序提供进程运行时间。由于 miniOS 使用优先级和时间片轮转接合方式进行进程调度,而 CPU 每个时间片超时,是通过定时器产生中断信号通知操作系统的,如果没有定时器操作系统无法确定一个 CPU 时间片的长度,也就无法对进程进行调度。miniOS 采用 PWM Timer 来为操作系统提供时间计数,如图 3-47 所示。

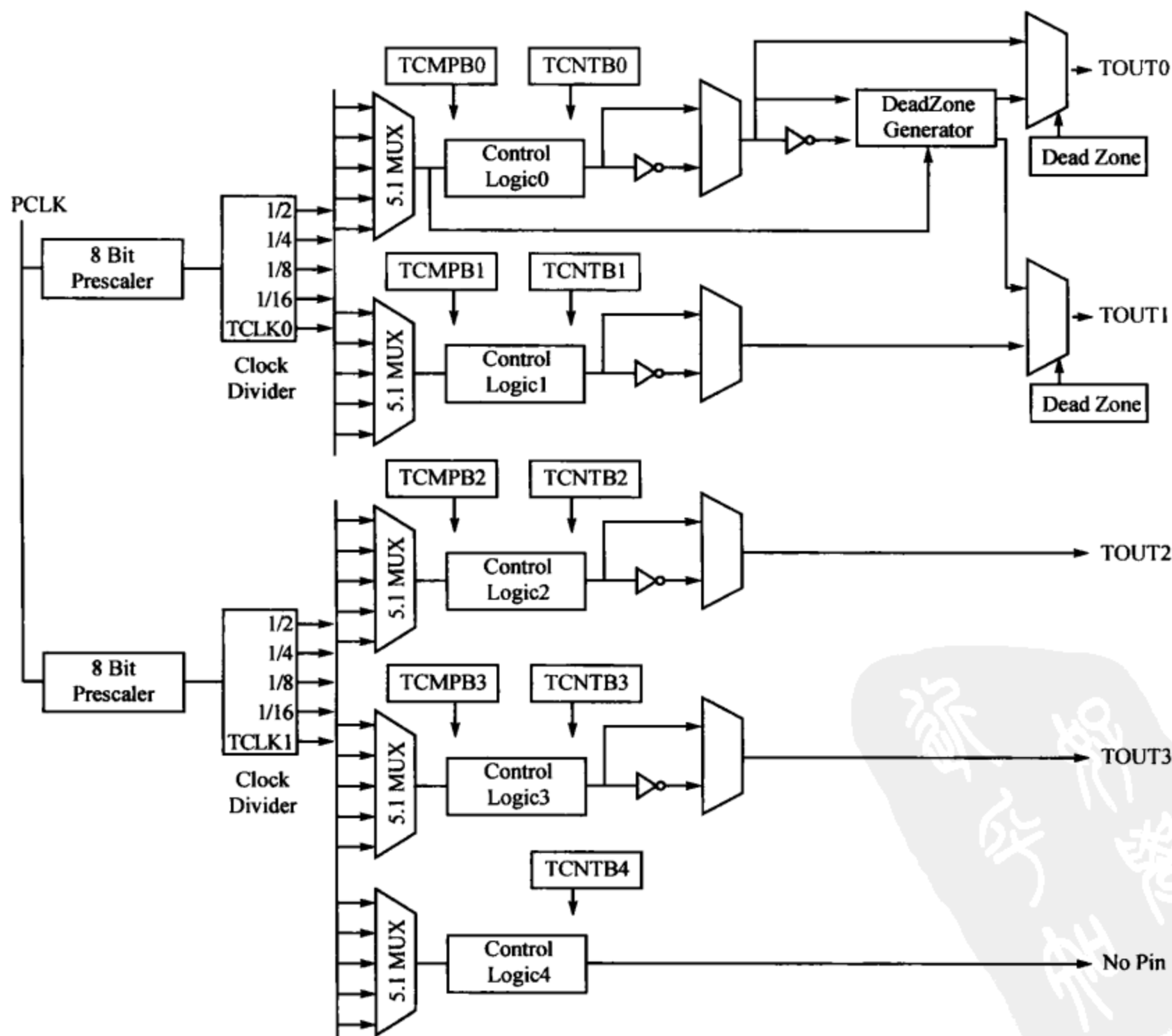


图 3-47 PWM 定时器模块图

1. 定时器工作原理

定时器的工作原理非常简单,它有两个寄存器,TCNTB0 定时器计数缓存寄存器,TCNTO0 定时器观察寄存器,用户可以向 TCNTB0 中放入一个初始计数,在定时器开启时,硬件自动将 TCNTB0 的值复制到 TCNTO0 里,然后在输入时钟下 TCNTO0 里的值递减,当 TCNTO0 的值减为 0 时,产生中断(如果中断开启的话),然后硬件再自动将 TCNTB0 的值复制到 TCNTO0 里(如果设置了自动载入位的话)。

S32440A 支持 5 个 16 位的定时器。定时器 0 和 1 共享一个 8 位的预分频器(Prescaler),定时器 2、3、4 共享另一个 8 位预分频器,预分频器先对输入时钟进行预分频,经过预分频的时钟再进入时钟分频器,每一个定时器有一个时钟分频器,通过设置寄存器 TCFG1 可将其分频成 5 种不同的分频信号(1/2,1/4,1/8,1/16 和 TCLK,其实就是将 PCLK 分频)。在 MUX 时钟信号复用选择器中根据 TCFG1 寄存器设置的分频比分频时钟,经过 MUX 后(已经被两次分频)的时钟就是定时器的工作时钟,每一个时钟周期定时器硬件会自动执行减 1 操作。

当定时器使能时,用户可编程设置定时器计数缓存寄存器(TCNTBn)来设置定时器的初始值计数。在定时器开启时,它被装载到递减计数器中。

每一个定时器有一个自己由定时器时钟驱动的 16 位递减计数器($2^{16}=64\text{KB}$)。当递减计数器为 0 时,定时器产生中断请求,通知 CPU 去处理定时器中断,同时硬件自动复制相应的 TCNTBn 的值到递减计数器里(如果自动重载使能的话),继续下一个定时操作。如果定时器运行时,将 TCONn 的定时器使能位清零,TCNTBn 里的值不会被复制到递减计数器里。

在定时器开启时,可以通过查看 TCNTO0 寄存器来观察当前递减计数器中的数值。

由上面的知识可以得出表 3-22 所列的定时器定时范围(PCLK=50MHz)。

表 3-22 定时器定时范围

| 4-bit divider settings | Minimum resolution (prescaler=0) | Maximum resolution (prescaler=255) | Maximum interval (TCNTBn=65535) |
|------------------------|-------------------------------------|---------------------------------------|------------------------------------|
| 1/2(PCLK=50 MHz) | 0.0400 μs (25.0000 MHz) | 10.2400 μs (97.6562 kHz) | 0.6710 s |
| 1/4(PCLK=50 MHz) | 0.0800 μs (12.5000 MHz) | 20.4800 μs (48.8281 kHz) | 1.3421 s |
| 1/8(PCLK=50 MHz) | 0.1600 μs (6.2500 MHz) | 40.9601 μs (24.4140 kHz) | 2.6843 s |
| 1/16(PCLK=50 MHz) | 0.3200 μs (3.1250 MHz) | 81.9188 μs (12.2070 kHz) | 5.3686 s |

2. 定时器的初始化

当递减计数器为 0 将出现定时器自动重载操作。因此递减计数器的初始值必须由用户预定义。在定时器初始化时,初始值必须手动装载,通过设置手动更新位来实现。以下步骤描述了如何开启定时器:

(1) 设置初始值到 TCNTBn 和 TCMPBn。

(2) 设置相应定时器的手动更新位。

(3) 设置相应定时器的开始位,开启定时器(同时要清除手动更新位)。

(4) 如果定时器被强行停止,递减计数器里的值被保留不变,且不会再从 TCNTBn 里重载。如果必须设置新值(比如要修改定时器的时间),则要再次执行手动更新。

定时器工作时钟换算公式:

$$\text{定时器工作时钟} = \text{PCLK} / (\text{prescaler value} + 1) / \text{分频因子}$$

miniOS 的任务调度时间片时间为 10ms, PCLK 为 50MHz, 当 prescaler value 设置为 49, divider value 选择为 1/16 分频, 定时器输出时钟频率为 62500Hz, 时钟周期为 $16\mu\text{s}$, 设置 TCNTB0 为 625, 定时器就每过 10ms 产生一次定时器中断。

如表 3-23、表 3-24、表 3-25、表 3-26 所列, 具体操作如下:

(1) TCFG0 bit[7]~[0] 设置为 49。

(2) TCFG1 bit[3]~bit[0] 为定时器 0 的分频因数, 将其选择为 1/16, 即 $\text{TCFG1} = 0x03$ 。定时器开启时会以 62500Hz 的频率提供给递减计数器工作, 每一时钟周期执行减 1 操作。

(3) 将 TCNTB0 的值设置为 625, 即 $625 \times 16\mu\text{s} = 10\text{ms}$, 定时器会每 10ms 产生一次中断(如果开启中断)。

(4) 通过执行 $\text{TCON} | = (1 \ll 1)$, 手动装载 TCNTB0 的值 625 到递减计数器里, 装载后, 执行 $\text{TCON} = 0x09$, 关闭手动装载功能, 开启计数器和自动装载功能。

表 3-23 定时器配置寄存器 0(TCFG0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|------|-------------------|------------|
| TCFG0 | 0x51000000 | R/W | 用于配置 8 位预分频器 | 0x00000000 |
| TCFG0 | 位 | | 描 述 | 初始值 |
| ... | ... | ... | ... | ... |
| 8 位预分频器 0 | [7:0] | | 设置预分频器 0 的 8 位分频数 | 0 |

表 3-24 定时器配置寄存器 0(TCFG1)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|---------|------------|------|---|------------|
| TCFG1 | 0x51000004 | R/W | 用于配置多路复用分频除数与 DMA | 0x00000000 |
| TCFG1 | 位 | | 描 述 | 初始值 |
| ... | ... | ... | ... | ... |
| 多路复用器 0 | [3:0] | | 设置定时器 0 的多路复用器输入时钟分频: 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = 外部输入时钟 | 0 |

表 3-25 定时器控制寄存器(TCON)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-------------------------|------------|--|----------|------------|
| TCON | 0x51000008 | R/W | 定时器控制寄存器 | 0x00000000 |
| TCON | 位 | 描 述 | 初始值 | |
| ... | ... | ... | ... | ... |
| 定时器 0 开启/关闭 重新加载 | [3] | 设置定时器 0 是否开启重新加载计数: 0 = 只执行一次 1 = 超时后重新加载 | 0 | |
| 定时器 0 开启/关闭 PWM 输出变换 | [2] | 设置定时器 0 是否开启 PWM 输出变换 | 0 | |
| 定时器 0 手动更新 | [1] | 设置是否定时器 0 手动更新 0 = 无操作 1 = 更新 TCNTB0 和 TCMPB0 | 0 | |
| 开启/关闭定时器 0 | [0] | 设置是否开启定时器 0 0 = 停止 1 = 开启 | 0 | |

表 3-26 定时器 0 计数缓存寄存器(TCNTB0)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|------------|---------------|------------|
| TCNTB0 | 0x5100000C | R/W | 定时器 0 计数缓存寄存器 | 0x00000000 |
| TCNTB0 | 位 | 描 述 | 初始值 | |
| 定时器 0 计数值 | [15:0] | 设置定时器 0 计数 | 0x0000 | |

表 3-27 定时器 0 计数观察寄存器(TCNT00)

| 寄存器名 | 地 址 | 是否读写 | 描 述 | 复位默认值 |
|-----------|------------|--------------------|---------------|------------|
| TCNT00 | 0x51000014 | R | 定时器 0 计数观察寄存器 | 0x00000000 |
| TCNT00 | 位 | 描 述 | 初始值 | |
| 定时器 0 计数值 | [15:0] | 显示定时器 0 计数,该寄存器为只读 | 0x0000 | |

miniOS 定时器初始化代码:

```
#include "serial.h"
#include "s3c2440.h"
```

```
/*
*****
* 定时器 0 初始化代码
* 定时器 0 输入时钟频率 = PCLK / (预分频器值 + 1) / (分频因子)
*/
```

```

*      其中:    预分频器值 = 0~255
*              分频因子 = 2, 4, 8, 16
*              输入时钟频率 = 50MHz/(49+1)/(16) = 62500Hz
*              设置定时器计数器 TCNTB0[15:0] = 625, 则每 10ms 产生一次定时器中断
*****/
void Timer0_init(void)
{
    VA_TCFG0 = 49;           // Prescaler0 = 49
    VA_TCFG1 = 0x03;         // Select MUX input for PWM Timer0; divider = 16
    //TCNTB0 = 62;           // Request PWM Interrput per 1ms
    VA_TCNTB0 = 625;         // Request PWM Interrput per 10ms
    //TCNTB0 = 6250;         // Request PWM Interrput per 100ms
    //TCNTB0 = 62500;        // Request PWM Interrput per 1s used to Debug
    VA_TCON |= (1<<1);      // Timer 0 manual update
    /* Timer 0 auto reload on Timer 0 output inverter off */
    VA_TCON = 0x09;
    DPRINTF(KERNEL_DEBUG, "Timer0_init OK");
}

```

interrupt. c:

中断处理程序,使用虚拟地址,主要处理按键中断和定时器中断。

```

#include "s3c2440.h"
#include "serial.h"
#include "interrupt.h"
#include "sched.h"

void handle_irq()
{
    unsigned long irqOffSet = VA_INTOFFSET;
    char str[2];
    switch(irqOffSet) {
        case K1_IRQ_OFT:
            printk("KEY 1 to create new Task");
            str[0] = 'A';
            str[1] = 0;
            if(OSCreateProcess(0xA0010000, 1024, str, 5) == -1)
                DPRINTF(KERNEL_DEBUG, "Process Create fault!!");
            break;
    }
}

```

资源解密

PDG

```

        case K2_IRQ_OFT:
            printk("KEY 2 to kill a random Task");
            kill_task(0);
            break;

        case K3_K4_IRQ_OFT:
            if(K3_EINT_BIT & VA_EINTPEND) {
                VA_EINTPEND &= K3_EINT_BIT;
                printk("KEY 3 pressed");
            } else {
                VA_EINTPEND &= K4_EINT_BIT;
                printk("KEY 4 pressed");
                if(OSCreateProcess(0xA0020000, 1024, NULL, 5) == -1)
                    DPRINTK(KERNEL_DEBUG, "Process Create fault!!");
            }
            break;
        case TIMER_OFT:
            do_timer();
            break;
        default:
            DPRINTK(KERNEL_DEBUG, "Unknown Interrupt.");
    }

    VA_SRCPEND &= (1<<irqOffSet);
    VA_INTPND = VA_INTPND;
}

```

sched.h:

进程调度操作相关头文件,里面定义了进程状态,进程结构体进程队列数组,和进程操作相关函数。

```

/* 条件编译 */
#ifndef SCHED
#define SCHED

#include "s3c2440.h"

#define TASK_SZ 63 // 支持最大进程个数
#define VALID_TASK_INDEX(x) ((x) >= 1 && (x) <= 23) || ((x) >= 26 && (x) < TASK_SZ)

```

```

#define TASK_UNALLOCATE    -1    // PCB 未分配
#define TASK_RUNNING       0    // 进程正在进行或已经准备就绪
#define TASK_INTERRUPTIBLE  1    // 进程处于可中断等待状态
#define TASK_UNINTERRUPTIBLE 2    // 进程处于不可中断等待状态
#define TASK_ZOMBIE        3    // 进程处于僵死状态,未用到
#define TASK_STOPPED       4    // 进程已经停止
#define TASK_SLEEPING      5    // 进程进入睡眠状态

#define PID_OFT             0    // PID 相对偏移位
#define STATE_OFT          4    // 状态相对偏移位
#define COUNT_OFT          8    // 时间片数相对偏移位
#define PRIORITY_OFT       16   // 优先级别相对偏移位
#define CONTENT_OFT        20   // 上下文保存数据区相对偏移位
#define NULL               0

```

```
typedef struct task_struct
```

```

{
    long pid;                // 进程 ID
    long state;              // 进程状态
    long count;              // 进程时间片数
    long timer;              // 进程休眠时间
    unsigned long priority;   // 进程优先级
    unsigned long content[20]; // 进程执行现场保存区(寄存器的值)
    /* *
     * content[0]:用户进程状态寄存器 CPSR 的值
     * content[1]:保存用户进程 SP 栈指针寄存器的值
     * content[2]:保存用户进程 LR 返回地址寄存器的值
     * content[3~15]:保存 R0~R12 寄存器的值
     * content[16]:保存 PC 程序计数器寄存器的值
     */
} PCB;

// 进程队列数组
extern PCB task[TASK_SZ];
// 当前进程结构体指针
extern PCB * current;
// 检查当前 CPU 工作模式,如果是用户模式返回非 0
int is_in_user_space(void);
// 定时器处理
void do_timer(void);

```

资源解密

PDG

```

// 进程调度
extern void schedule(void);
// 杀死进程
extern void kill_task(int);
// 进程调度初始化
extern void sched_init(void);
// 创建进程
extern int OSCreateProcess(unsigned long start_addr, unsigned long len, char * parameters, long
priority);
#endif

```

schedule.c:

进程调度相关程序代码, 主要包含进程队列初始化、杀死进程、进程调度和创建新进程。

新进程执行空间为 1MB 大小, 在进程执行之前, 要先将外部存储设备上的程序文件加载到执行空间中, 最后 1KB 用来存放当前进程执行参数, 程序在启动后会从其参数空间内读取启动参数。

```

#include "sched.h"
#include "s3c2440.h"
#include "init.h"
#include "mmu.h"
#include "string.h"
#include "interrupt.h"
#include "serial.h"
#define SYS_MODE_STACK_BASE (KERNEL_LIMIT - 0x10000)

PCB task[TASK_SZ]; // 进程队列(PCB 数组), 最多支持 62 个进程
PCB * current = NULL; // 声明当前执行进程指针
extern void __switch_to(PCB * pcur, PCB * pnext); // 引入外部声明上下文切换函数

/*****
 * 进程队列初始化函数
 *****/
void sched_init(void)
{
    PCB * p = &task[0]; // 0 号进程为内核进程
    int i;
    /* 循环为每个进程 PCB 初始化 */
    for(i = 0; i < TASK_SZ; i++, p++){
        p->pid = -1; // pid = -1, 表示未分配 pid
    }
}

```



```

    p->state = TASK_UNALLOCATE; // 设置初始进程状态为未分配状态
    p->count = 0;                // 设置时间片计数为 0, 表示没有时间片
    p->priority = 0;             // 初始进程优先级为 0
}
/* 初始化 0 号进程 */
p = &task[0];                  // p 指向 0 号进程 PCB
p->pid = 0;                     // 设置 0 号进程 pid
p->state = TASK_RUNNING;       // 设置其运行状态为就绪态
p->count = 5;                   // 设置其时间片为 5
p->priority = 5;                // 设置优先级为 5
p->content[0] = 0x5f;           // 保存状态寄存器 cpsr 值, 表示为系统模式, 开启中断
p->content[1] = SYS_MODE_STACK_BASE; // 设置当前进程栈指针
p->content[2] = 0;
p->content[16] = 0;             // 设置 PC 寄存器的值为 0, 该进程起始地址被 MMU 映射为 0 地址

current = &task[0];            // 当前运行进程为 0 号进程
DPRINTK(KERNEL_DEBUG, "kernel:sched_init, all task init OK");
}

/*****
* 杀死进程函数
* 参数: int pid; 0: 随机杀死一个进程, pid 不为 0: 杀死指定进程 id 号为 pid 进程
*****/
void kill_task(int pid){
    int i;
    DPRINTK(KERNEL_DEBUG, "kernel:kill_task");
    for(i = 1; i < TASK_SZ; i++){
        if(task[i].state != TASK_UNALLOCATE){
            if(pid == 0 || pid == task[i].pid){
                task[i].pid = -1;
                task[i].state = TASK_UNALLOCATE;
                task[i].count = 0;
                task[i].priority = 0;
                break;
            }
        }
    }
}

// 杀死进程后, 重新调度
schedule();

```

资源解密
PDG

```

}

/*****
 * 进程调度函数
 * 从当前进程就绪队列中挑选出优先级最高的进程执行,如果没有就绪进程,执行内核进程
 * 如果当前进程还是最高优先级,则继续执行当前进程
 * 如果有进程比当前进程优先级高,则进行上下文切换
 *****/
void schedule(void)
{
    /*
     * max 用来保存当前进程队列里最高优先级进程 count
     * p_tsk 保存当前进程 PID 副本
     */
    long max = -1;           // max 初始值为 -1,后面做判断
    long i = 0, next = 0;    // next 保存最高优先级 PID
    PCB * p_tsk = NULL;      // 临时进程结构体指针
    // 进程调度循环

    DPRINTK(KERNEL_DEBUG, "kernel:schedule");
    while(1){
        /*
         * 循环找出进程队列里,就绪状态最高优先级进程,也就是 count 值最大进程,
         * count 越大说明其被执行时间越短,CPU 需求越高,
         * 同时保存其 PID(进程队列数组下标)到 next 里
         */
        for(i = 0; i < TASK_SZ; i++){
            if( (task[i].state == TASK_RUNNING) && (max < (long)task[i].count) ){
                max = (long)task[i].count;
                next = i;
            }
        }
        // 如果 max 为非 0,跳出循环,说明选出了调度进程
        // 如果 max 为 0,说明 count 值最大进程 count 为 0,说明全部进程分配时间片已执行完,
        // 需要重新分配,执行 break 后面 for 语句
        // 如果 max 为 -1 说明没有就绪状态进程可被调度,退出循环,继续执行 0 进程

        if(max) break;        // max = 0 时,选出新进程,跳出循环
    }
}

```

```

// max = 0,即进程队列中 count 值最大为 0,全部进程时间片用尽,需要重新分配
for(i = 0; i < TASK_SZ; i++){
    if( task[i].state == TASK_RUNNING ) {
        // 时间片数为其默认优先级
        task[i].count = task[i].priority;
    }
}
// 当前进程为选出进程,说明当前进程优先级还是最高,返回继续执行
if(current == &task[next])
    return;
// 无效 PID
if(task[next].pid < 0)
    return;
// 保存当前进程副本到 p_tsk,将选出进程设置为当前运行进程
p_tsk = current;
current = &task[next];
DPRINTK(KERNEL_DEBUG, "__switch_to");
// 调用上下文切换函数
__switch_to(p_tsk, &task[next]);
}

/*****
 * 检查当前 CPU 工作模式,如果是用户模式,返回 1,如果不是返回 0
 *****/
int is_in_user_space(void) {
    unsigned long __rtn;
    __asm{
        mrs __rtn, SPSR
        and __rtn, __rtn, #0x1F
        cmp    __rtn,    #0x10
        moveq __rtn, #0x1
        movne __rtn, #0x0
    }
    return __rtn;
}

/*****
 * 定时器处理函数
 *****/

```

- * 主要用于进程时间片处理和睡眠时间处理,每次定时器中断产生后,调用该函数,对进程时间片
- * 进行递减操作,如果时间片用完,则进行调度,如果用户进程主动进入睡眠状态,则
- * 该函数在睡眠时间到达后,将其唤醒

```

*****/
void do_timer(void)
{
    int i = 0;
    // 没有当前进程,说明进程还未创建,返回
    if(! current){
        DPRINTK(KERNEL_DEBUG,"kernel:leaving do_timer,hasnt init task");
        return;
    }
    // 递减睡眠进程,睡眠时间到了,将其状态改为就绪态
    for(i = 1; i < TASK_SZ; i++){
        if(task[i].state == TASK_SLEEPING){           // 检查其睡眠时间
            if(! (--task[i].timer)){
                task[i].state = TASK_RUNNING;         // 如果睡眠时间为 0,唤醒它
            }
        }
    }
    // 对当前执行进程时间片递减,每 10ms 递减一次
    if(current->count){
        current->count--;
    }
    // 如果当前进程时间片已经用完,或当前进程状态为非就绪态,则尝试调度新进程
    if((current->state != TASK_RUNNING) || current->count <= 0){
        // 保障内核空间执行进程不会被抢占打断
        if(is_in_user_space())
            schedule();
    }
}

/*****/
* 进程代码复制函数,用于在创建新进程后,将程序代码复制到其对应执行空间
*****/
int CopyCode2Ram2(unsigned long * src, unsigned long * dest, int size)
{
    unsigned long i;
    /* copy code from NOR Flash to RAM */

```

```

    for (i = 0; i < size / 4; i++) {
        dest[i] = src[i];
    }
    return 0;
}

/*****
 * 创建新进程函数
 * 参数: unsigned long start_addr 新进程的运行地址
 *       unsigned long len 新进程代码长度
 *       char * parameters 程序执行参数,以空格格开
 *       long priority 进程指定优先级
 *****/
int OSCreateProcess(unsigned long start_addr, unsigned long len, char * parameters, long priority)
{
    unsigned long i, j, pid = -1, argc = 0;
    unsigned long * p_VA;
    char * pDes;

    DPRINTK(KERNEL_DEBUG, "kernel: Enter OSCreateProcess");
    /*
     * 检查用户程序是否符合程序调用规格,所有用户程序第一条指令为:ldr    r0, [sp]
     * 其对应机器码为:0xe59d0000,如果加载的是非法程序,则出错退出
     */
    if( *((unsigned long *)start_addr) != 0xe59d0000 ){
        printk("user program error!!");
        return -1;
    }
    // 为新进程挑选可用 pid
    for(i = 0; i < TASK_SZ; i++){
        if((task[i].state == TASK_UNALLOCATE && VALIDE_TASK_INDEX(i)) ) {
            pid = i;
            break;
        }
    }
    // 如果没有可用 Pid,出错,退出
    if(pid == -1){
        printk("task has to max number!");
        return -1;
    }

```



```

}
// 进入进程创建,此过程中不能被中断打断
OS_ENTER_CRITICAL();
/*
 * 新进程执行空间首地址,为其对应 Pid 号对应 MB 处,比如 pid = 4,则其进程空间为
 * 内存基址 + 4MB,使用了段映射每个进程对应 1MB 执行空间
 */
p_VA = (unsigned long *)(SDRAM_BASE + (pid << 20));
// 将用户执行空间清零
or(j = 0; j < (0x100000 >> 2); j++)
    p_VA[j] = 0;
// 从 Norflash 将程序代码复制到新进程执行空间
CopyCode2Ram2((unsigned long *)start_addr, p_VA, len);

// -----以下为程序执行时所带参数处理 -----
// 程序参数保存在新进程空间最高 1KB 空间内,以下简称参数空间
p_VA = (unsigned long *)(SDRAM_BASE + (pid << 20) + 0x100000 - 1024);
// 参数个数变量
argc = 0;
j = 0;

/* 根据 parameters,计算出程序参数个数,以空格格开 */
if(parameters){
    while(parameters[j]){
        while(parameters[j] == ' ') j++;           // 吃掉空格
        if(! parameters[j]) break;                 // 到达字符串结束,退出
        argc++;                                     // 参数个数++
        while(parameters[j] && (parameters[j] != ' ')) j++;
                                                // 如果是有效参数,跳过
        if(! parameters[j])
            break;
    }
}
// 现在 argc 里面保存的是 parameters 传递过来的参数个数,以空格格开

* p_VA++ = argc + 1;                             /* 将参数个数保存到参数空间内第一个位置 */
* p_VA++ = pid;                                    /* 将新进程 pid 号保存到参数空间第二个位置 */

// pDes 用来复制每个参数字符串,首先跳过 argc + 1 个位置,每个位置用来放参数字符串指针

```

```

pDes = (char *)(p_VA + argc + 1);
// 循环复制每一个参数
for(i = 0, j = 0; i < argc; i++){
    // 将每一个参数字符串地址保存到 p_VA 指向的参数空间
    *p_VA++ = (unsigned long)pDes - (SDRAM_BASE + (pid << 20));
    while(parameters[j] == ' ') j++;           // 吃掉空格
    if(! parameters[j]) break;                 // 到达字符串结束,退出
    // 执行参数字符串复制
    while(parameters[j] && (parameters[j] != '\0')){
        *pDes = parameters[j];
        pDes++;
        j++;
    }
    // 最后加上字符串结束符
    *pDes = '\0';
    // 保证每个参数开始位置是 4 字节对齐
    pDes = (char *)(((unsigned long)pDes + 4)&(~0x3));
    if(! parameters[j])
        break;
}
// -----以上为程序执行时所带参数处理-----

task[pid].pid = pid;           // 新进程 PID
task[pid].state = TASK_RUNNING; // 新进程执行状态
task[pid].count = 5;           // 新进程时间片
task[pid].priority = priority;  // 新进程优先级
task[pid].content[0] = 0x5f;    // CPSR
task[pid].content[1] = 0x100000 - 1024; // SP 栈指针
task[pid].content[2] = 0;       // LR 返回地址
task[pid].content[16] = 0;      // PC
// 打开中断
OS_EXIT_CRITICAL();
return pid;
}

```

context_switch. S;

本程序文件主要实现了进程的上下文切换,由于在上下文切换时,主要用来保存寄存中的数据,因此这儿全部用汇编实现。具体详解请看第 3.6.6 节——进程上下文切换。

INCLUDE common_asm.h

```

PID_OFT      EQU      0      ; 进程 PID 相对偏移
STATE_OFT    EQU      4      ; 进程状态相对偏移
CONTENT_OFT   EQU      20     ; 进程执行现场保存区(寄存器的值)
SPSR_OFT     EQU      CONTENT_OFT      ; CPSR 相对偏移
USER_SP_LR_OFT EQU      CONTENT_OFT + 1      ; 用户模式 SP,LR 偏移
R0_R13_OFT   EQU      USER_SP_LR_OFT + 1      ; 用户能用寄存器偏移
; 由于在异常发生后,要求异常处理之前保存执行现场,其入栈寄存器的值依次保存起来
; 为了取得在进入异常时的执行现场,将其对应模式下栈指针向下偏移 14,得到 R0~R13 寄存器的值
IRQ_SAVE_BASE EQU      (IRQ_STACK_BASE - 14 * 4) ; 中断模式下 R0~R13 寄存器保存地址
SVR_SAVE_BASE EQU      (SVC_STACK - 14 * 4)      ; 管理模式下 R0~R13 寄存器保存地址

```

```

AREA SWITCH, CODE, READONLY

```

```

EXPORT __switch_to

```

```

__switch_to      ; __switch_to 有可能从 irq 和 svc 模式调用

```

```

mrs      r2, cpsr

```

```

bic      r2, r2, #0xffffffe0

```

```

cmp      r2, #0x12      ; 判断是否为 irq 模式

```

```

; 设置中断模式下的 sp 栈指针

```

```

ldreq    sp, = IRQ_SAVE_BASE

```

```

; 设置管理模式下的 sp 栈指针

```

```

ldrne    sp, = SVR_SAVE_BASE

```

```

add      r0, r0, #CONTENT_OFT ; r0 指向当前进程 PCB,偏移到 PCB 结构体 content[16]处

```

```

mrs      r2, spsr

```

```

stmia    r0!, {r2}      ; 保存 SPSR 中值到 PCB 结构体 content[0]中

```

```

stmia    r0!, {sp}^     ; 保存挂起的用户空间程序 SP,LR 到 content[1-2]

```

```

stmia    r0!, {lr}^     ; 用于将来恢复用户 SP, LR 寄存器(异常处理时没有保存
; 是因为,每种模式下都有自己的 SP,LR,而多进程在执行
; 每个程序都运行在用户模式下,不同进程的 SP,LR 不一样,
; 因此要保存,不然其他用户进程会覆盖掉当前进程 SP,LR)

```

```

; 把中断处理时保存的寄存器(R0~R13)转移到当前 PCB 中

```

```

ldmia    sp!, {r2-r8}   ; 加载 SP 中保存的 R0~R6

```

```

stmia    r0!, {r2-r8}   ; 存入 r0 指向的 PCB 结构体中 content[3~9]

```

```

ldmia    sp!, {r2-r8}   ; 加载 SP 中保存的 R7~R13

```

```

stmia    r0!, {r2-r8}   ; 存入 r0 指向的 PCB 结构体中 content[10~16]

```

```

; 清除中断

```

```

mov    r0, #0xCA000000    ; 使用虚拟地址
mov    r2, #0x400
str    r2, [r0]
add    r0, r0, #0x10
ldr    r2, [r0]
str    r2, [r0]

```

do_switch

```

ldr    r0, [r1, #PID_OFT]  ; 要切换到的进程的 PID
mov    r0, r0, lsl #25      ; PID 左移 25 位, 存在寄存器的最高 7 位
mcr    p15, 0, r0, c13, c0, 0; 写 next_pid, 从此, VA<32M 的取址计算公式就变了,
                                ; 不过现在的 VA 总是大于 32M 的

add    r1, r1, #CONTENT_OFT; r1 = pnext->content

ldmia  r1!, {r2-r4}         ; sp_another_state, sp_int, spsr_int
msr    spsr_cxsf, r2         ; 前挂起程序的 cpsr
mrs    r2, cpsr
; 切换到系统模式下, 加载用户模式下 SP, LR
msr    cpsr_c, #0xdf
mov    sp, r3
mov    lr, r4
; 切换回当前模式
msr    cpsr_cxsf, r2
ldmia  r1, {r0-r12, pc}^    ; 执行进程切换
END

```

swi_handle_s3c2440.C:

软中断主要用来实现操作系统提供给用户程序的接口的, miniOS 提供了 6 个软中断接口, 分别是: exit(), sleep(), write(), led_on(), all_led_on(), all_led_off()。当用户程序使用这些系统调用时, 通过软中断指令和软中断号切换到内核空间中, 由特权级别的内核程序执行具体的操作, 最后返回用户空间。本程序文件主要处理上述 6 种软中断。如果读者想加系统调用要修改宏 MAX_SWI_NUM 系统调用个数, swi_table(swi_fun.h) 函数指针数组, 将系统调用函数名进行初始化最后还要定义一个软中断号(swi.h), 代码如下:

```

#define    __NR_exit          0
#define    __NR_sleep         1
#define    __NR_write         2
#define    __NR_led_on        3

```

```
#define __NR_all_led_on 4
#define __NR_all_led_off 5
```

在软中断处理过程,最主要的就是取得软中断号,软中断号其实是包含在软中断指令机器码中的,由软中断指令格式可知,其低 24 位存放的软中断号,所以取出软中断指令机器码,将其低 24 位取出来即可。这儿还用到了一个小技巧,将全部处理系统调用函数名放到了 swi_table 函数指针数组中,通过数组下标方式可以调用对就系统调用函数。

```
INCLUDE common_asm.h
MAX_SWI_NUM EQU 6

EXTERN swi_table
AREA SWI, CODE, READONLY
EXPORT HandleSWI

HandleSWI
    ldr sp, = SVC_STACK
    stmdb sp!, {r0-r12, lr} ; 保存使用到的寄存器和返回地址,不必保存那么多,
                                ; 以后要提高实时性时再改吧
    ldr r4, [lr, #-4] ; lr-4 为指令"swi n"的地址,此指令低 24 位就是 n
    bic r4, r4, #0xff000000
    ldr r6, = MAX_SWI_NUM
    cmp r4, r6
    ldrls r5, = swi_table ; swi 跳转表基址
    ldrls lr, = swi_return ; 返回地址
    ldrls pc, [r5, r4, lsl #2] ; 跳转到相应汇编处理函数
    mov r0, #-1 ; 出错,返回 -1

swi_return
    ldmia sp!, {r0-r12, pc}^ ; 中断返回, ^表示将 spsr 的值复制到 cpsr

; 进程退出处理函数,比如处理 main 函数退出
EXPORT sys_exit
IMPORT do_exit

sys_exit
    ; 将调用 C 函数 do_exit(int error_code)
    ; r0 = error_code
    ldr pc, = do_exit ; do_exit 返回到 swi_return

EXPORT sys_sleep
IMPORT do_sleep
```



```

sys_sleep
    ; 将调用 C 函数 int sleep(int time)
    ; r0 = sleep time
    ldr    pc, = do_sleep          ; do_sleep 返回到 swi_return

    EXPORT sys_write
    IMPORT do_write

sys_write
    ; 将调用 C 函数 int write(char * str)
    ; r0 = 字符串首地址
    ldr    pc, = do_write          ; do_write 返回到 swi_return

    EXPORT sys_led_on
    IMPORT do_led_on

sys_led_on
    ; 将调用 C 函数 void led_on(int led_no)
    ; r0 = led_no
    ldr    pc, = do_led_on         ; do_led_on 返回到 swi_return

    EXPORT sys_all_led_off
    IMPORT do_all_led_off

sys_all_led_off
    ; 将调用 C 函数 void all_led_off(void)
    ldr    pc, = do_all_led_off    ; do_all_led_off 返回到 swi_return

    EXPORT sys_all_led_on
    IMPORT do_all_led_on

sys_all_led_on
    ; 将调用 C 函数 void all_led_on(void)
    ldr    pc, = do_all_led_on     ; do_all_led_on 返回到 swi_return

    END

```

swi_fun.c:

本程序文件主要是具体系统调用内核空间实现部分,该部分函数以用户程序的软中断指令中的中断号为标识进行调用,因此,用户程序和内核系统调用实现,双方要将软中断号统一。

```

#include "swi.h"
#include "serial.h"
#include "sched.h"

```

```

#include "led.h"

/*****
 * 所有系统调用函数指针数组,每个数组成员都是一个指向函数的指针
 *****/
fn_ptr swi_table[] = {sys_exit, sys_sleep, sys_write,
                      sys_led_on, sys_all_led_off, sys_all_led_on};

/*****
 * exit 系统调用内核空间实现代码
 *****/
void do_exit(int error_code)
{
    DPRINTK(KERNEL_DEBUG, "kernel:do_exit\n\r");
    current->state = TASK_UNALLOCATE;
    schedule();
}

/*****
 * sleep 系统调用内核空间实现代码
 *****/
void do_sleep(unsigned long time)
{
    DPRINTK(KERNEL_DEBUG, "kernel:do_sleep\n\r");
    current->state = TASK_SLEEPING;
    current->timer = time;
    schedule();
}

/*****
 * write 系统调用内核空间实现代码
 *****/
int do_write(char * str)
{
    return printk(str);
}

/*****
 * led_on 系统调用内核空间实现代码
 *****/

```

```

*****/
int do_led_on(int led_no)
{
    led_on(led_no);
    return led_no;
}

/*****/
* all_led_off 系统调用内核空间实现代码
*****/
void do_all_led_off(void)
{
    all_led_off();
}

/*****/
* all_led_on 系统调用内核空间实现代码
*****/
void do_all_led_on(void)
{
    all_led_on();
}

```

3.8.2 miniOS 应用程序接口

应用程序被加载到内存之后,当取得 CPU 权限(被调度)后,准备运行,新进程其开始运行地址都是从 0 地址开始,由于 MMU 的映射机制,会被自动映射到当前进程对应的执行空间,因此当前进程执行空间最开始处也就是应用程序最先运行代码。

用户 C 程序都是从 main 函数开始的,main 函数是一个特殊的函数,它是会被系统最先调用的函数,在进程启动时,main 函数被调用之前,还要有以下工作要做:

(1) 传递给用户程序的参数。传递给用户程序参数,根据 ATPCS 规范,R0 是保存 argc,R1 中保存 argv 地址即可。而在内核代码启动用户程序时,内核会将启动用户程序的参数保存在其栈空间中,(miniOS 中进程空间 1MB 最后 1KB 作为栈空间),然后通过栈传递给用户程序。通过下面代码可以取出栈中参数个数和参数列表,传递给 main。

crt0.S:用户程序启动时,参数传递和返回处理代码

```

*****/
; File:crt0.s
; 功能:通过它转入 C 程序

```

```

;*****
AREA    crt0, CODE, READONLY
ENTRY
ldr     r0, [sp]           ; main 函数参数 argc
add     r1, sp, #4         ; main 函数参数 argv
IMPORT main
bl      main               ; 调用 main 函数
IMPORT exit
bl      exit               ; 用户程序 main 结束后,调用 exit,在 lib.c 中
halt_loop
b       halt_loop
END

```

(2) 用户程序返回结束处理。用户程序 main 函数返回之后就意味着用户程序结束,但是用户程序真正结束是在返回到内核空间之后,对用户程序的 main 函数的善后处理工作也是有必要做的,这也是 main 函数返回值的主要意义。

3.8.3 miniOS 应用程序系统调用接口

由 3.8.2 节可知,miniOS 提供了 6 个系统调用接口,用户程序在使用这些系统调用时,要注意以下两点:

- (1) 通过软中断指令切换到内核。
- (2) 用户系统调用接口与内核系统调用处理的软中断号要一致。

lib.c:

本程序文件为用户程序提供系统调用接口。

```

#include "lib.h"

/*****
 * exit 系统调用接口函数,提供给用户程序
 *****/
int exit(int error_code)
{
    int __rtn;
    __asm{
        mov     r0, error_code
        swi     __NR_exit
        mov     __rtn, r0
    }
    return __rtn;
}

```

```

}

/*****
 * sleep 系统调用接口函数,提供给用户程序
 *****/
int sleep(int time)
{
    int __rtn;
    __asm{
        mov    r0, time
        swi    __NR_sleep
        mov    __rtn, r0
    }
    return __rtn;
}

/*****
 * write 系统调用接口函数,提供给用户程序
 *****/
int write(char * str)
{
    int __rtn;
    __asm{
        mov    r0, str
        swi    __NR_write
        mov    __rtn, r0
    }
    return __rtn;
}

/*****
 * led_on 系统调用接口函数,提供给用户程序
 *****/
int led_on(int led_no)
{
    int __rtn;
    __asm{
        mov    r0, led_no
        swi    __NR_led_on
    }
}

```

PDF
PDG

```

        mov    __rtn, r0
    }
    return __rtn;
}

/*****
 * all_led_off 系统调用接口函数,提供给用户程序
 *****/
void all_led_off(void)
{
    __asm{
        swi    __NR_all_led_off
    }
}

/*****
 * all_led_on 系统调用接口函数,提供给用户程序
 *****/
void all_led_on(void)
{
    __asm{
        swi    __NR_all_led_on
    }
}

```

跑马灯用户程序 main.c:

```

#include    "lib.h"
int main(int argc, char * * argv)
{
    int i = 0;
    while(1){
        led_on((i++) % 4 + 1);
        write("user program leds will sleep 0.5s\n\r");
        sleep(50);    //系统时钟周期为 10ms,休眠 0.5s
    }
    return 0;
}

```

资源解密
PDG

第2篇

嵌入式 Linux 系统建构

特别提醒

本篇的示例中，>提示符表示在开发板的 Boot Loader 上操作；#提示符表示在开发板的 Linux 操作系统上操作；\$提示符表示在 PC 的 Linux 操作系统上操作，\$提示符前的是当前目录。例如：

```
Dennis Yang > usbslave 1 0x32000000  
# madplay /music/pianpianxihuanni.mp3  
/work/studydriver/examples/scull$ cat /proc/devices
```



第4章

嵌入式 Linux 软件开发环境搭建

4.1 体验嵌入式 Linux 系统

- (1) 获得光盘 image 目录提供的映像文件。
- (2) 使用 H-JTAG 将 u-boot.bin 烧写进 NOR FLASH 中。
- (3) 启动超级终端,设置 baudrate 为 115200,8IN1。重启开发板,将会进入 u-boot 的命令界面。表明 Boot Loader 已正常运行。

Dennis Yang >

- (4) 在 PC 上安装 USB 驱动(安装文件为 USB Download Driver.exe,需读者上网搜索后下载)。
- (5) 用 USB 线将 PC 与开发板连接起来。
- (6) 在 PC 上启动 USB 传输软件 dnw(需读者上网搜索后下载)。
- (7) 在开发板上输入 usbslave 命令,让开发板进入等待接收数据状态。

Dennis Yang > usbslave 1 0x32000000

USB host is connected. Waiting a download

- (8) 在 dnw 软件界面,单击“USB Port→Transmit/Restore”,找到 image 目录中的 kernel 文件 uImage(如果使用的是天嵌 tq2440,请选择 uImage-embsky),进行传输。这将导致 uImage 被传输到开发板内存的 0x32000000 处。

Dennis Yang > usbslave 1 0x32000000

USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:32000000h,TOTAL:1518826]

RECEIVED FILE SIZE: 1518826 (741KB/S, 2S)

- (9) 在 u-boot 命令行输入命令,将 NAND FLASH 的 0x100000~0x400000 区间擦除(格式化)。

Dennis Yang > nand erase 0x100000 0x300000

NAND erase: device 0 offset 0x100000, size 0x300000

Erasing at 0x120000 -- 100% complete.

OK

(10) 在 u-boot 命令行输入命令, 将内存 0x32000000 处的 kernel 烧写到 NAND FLASH 的 0x100000~0x400000 区间。

```
Dennis Yang > nand write.jffs2 0x32000000 0x100000 0x300000
```

```
NAND write: device 0 offset 0x100000, size 0x300000
```

```
Writing data at 0x3ff800 -- 100% complete.
```

```
3145728 bytes written: OK
```

(11) 使用相同的方法将根文件系统(myfs-128M.jffs2, 如果使用 64MB 的 NAND FLASH, 请选择 myfs-64M.jffs2) 烧写到 NAND FLASH 的 0x400000~0x3c00000 区间。

```
Dennis Yang > usbslave 1 0x30000000
```

```
USB host is connected. Waiting a download.
```

```
Now, Downloading [ADDRESS:30000000h,TOTAL:24594994]
```

```
RECEIVED FILE SIZE:24594994 (667KB/S, 36S)
```

```
Dennis Yang > nand erase 0x400000 0x3c00000
```

```
NAND erase: device 0 offset 0x400000, size 0x3c00000
```

```
Erasing at 0x3fe0000 -- 100% complete.
```

OK

```
Dennis Yang > nand write.jffs2 0x30000000 0x400000 $(filesize)
```

```
NAND write: device 0 offset 0x400000, size 0x1774a28
```

```
Writing data at 0x1b74800 -- 100% complete.
```

```
24594984 bytes written: OK
```

(12) 输入启动操作系统命令, 将进入 Linux 操作系统。

```
Dennis Yang > boot
```

(13) 使用触笔进行屏幕校正后, 将进入 qtopia 图形系统。

如果由于屏幕校正不准确而不能正常使用触摸屏的话, 请删除/etc/pointercal 文件后, 重启开发板, 重新进行校正。

(14) 在 Linux 命令提示符下, 播放歌曲:

```
# madplay /music/pianpianxihuanni.mp3
```

```
MPEG Audio Decoder 0.15.2 (beta) - Copyright (C) 2000 - 2004 Robert Leslie et al.
```

```
Title: Track 1
```

```
Artist: 陳百強
```

```
Orchestra: 陳百強
```

Album: Best Memory

Track: 15

Genre: Other

4.2 Linux 操作系统安装

4.2.1 在 Windows 上安装虚拟机

本书基于 ubuntu 9.10 进行开发,它是一个容易安装和使用的 Linux 发行版,光盘映像文件可以自由从互联网上获得,在配套光盘中提供了该文件(software/ubuntu-9.10-desktop-i386.iso)。下面介绍在 Windows 中通过 vmware 来安装 ubuntu 的方法。

特别说明:本书在虚拟机中使用两个硬盘,40GB 的硬盘用于挂载 root 分区(/)并制作 snapshot,这样可以在系统损坏时,快速的一键恢复;80GB 硬盘用于挂载 work 分区(/work),并设置为不受 snapshot 影响的独立硬盘,以后将在这个分区上编辑、编译软件,这样可以避免当系统出错后使用 snapshot 恢复时,不破坏学习成果。

请按照如下的一系列操作建立虚拟机。

(1) 在 Windows 上安装 VMware workstation 7.0 软件。

该软件可以从 VMware 的官方网站 <http://www.vmware.com> 下载。

(2) 启动 VMware,如图 4-1~图 4-20 所示,新建客户虚拟机。选择 File→New→Virtual Machine。

① 在主机能联通互联网的情况下,图 4-9 选择虚拟机与主机的互联方式为 NAT,使得虚拟机可以通过主机联通互联网。

② 图 4-13 中选择 Split virtual disk into 2GB files 选项,表示使用多个小于 2 GB 的文件来表示一个很大的硬盘。如果 Windows 的硬盘格式为 FAT32,请务必选择此选项,因为 FAT32 支持的最大文件大小为 4 GB,否则虚拟机将无法启动;如果是 NTFS 格式,就无需选择这个选项。

③ 图 4-16 中选择“Edit virtual machine settings”,可以增减、修改虚拟机的设备。

④ 图 4-18 用于新增第 2 个硬盘,大小 80GB,用于将来挂载 work 分区。

⑤ 图 4-19 设置新增的硬盘不受 snapshot 影响,即该硬盘上修改的内容不会被一键恢复。

⑥ 图 4-20 设置虚拟机使用光盘映像文件,这相当于将 ubuntu 的安装光盘插入了虚拟机的光驱。请务必确保 Connect at power on 选项被选中,这样当虚拟机启动时就能够从光盘启动,以便可以安装 Linux 操作系统。

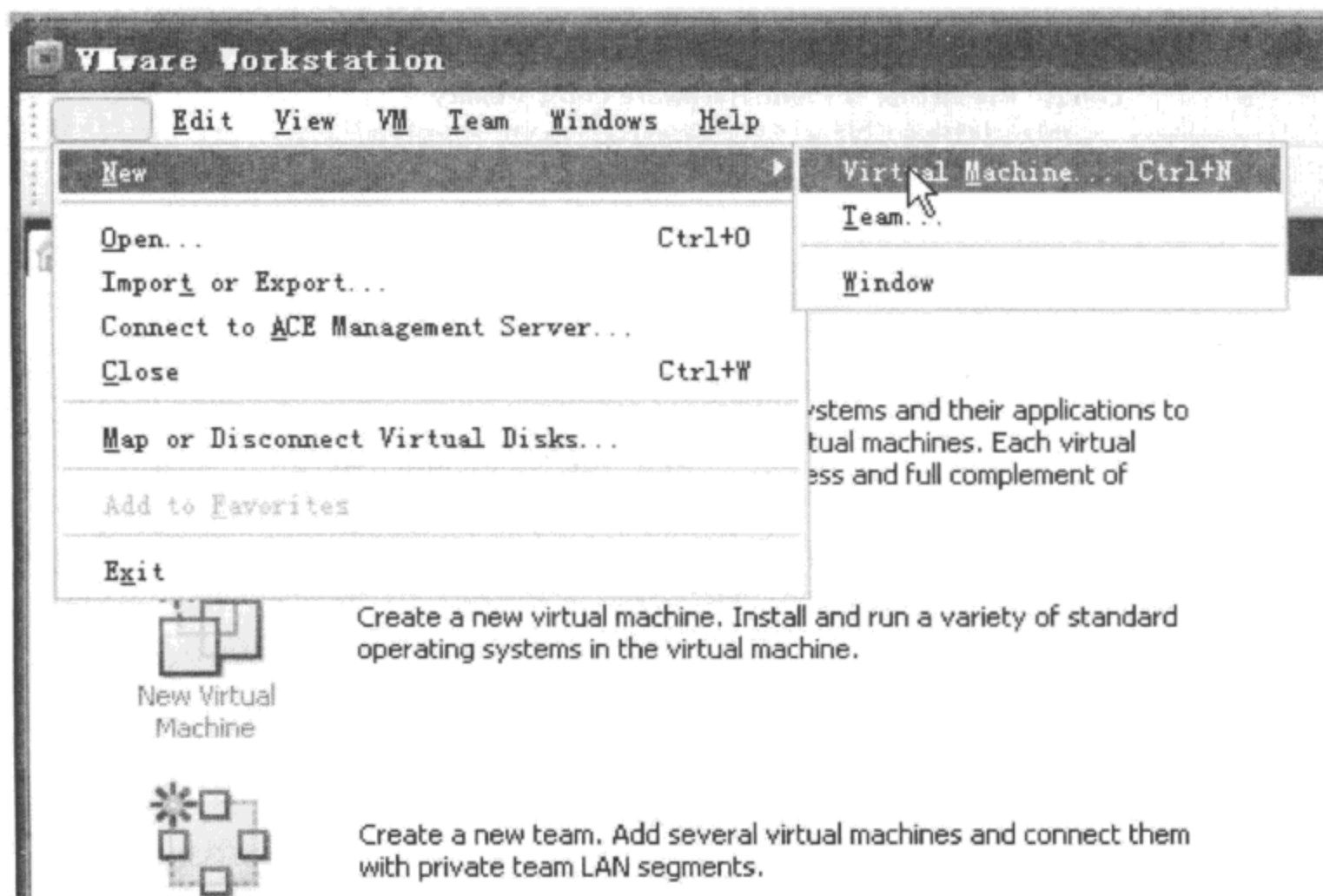


图 4-1 选择新建虚拟机

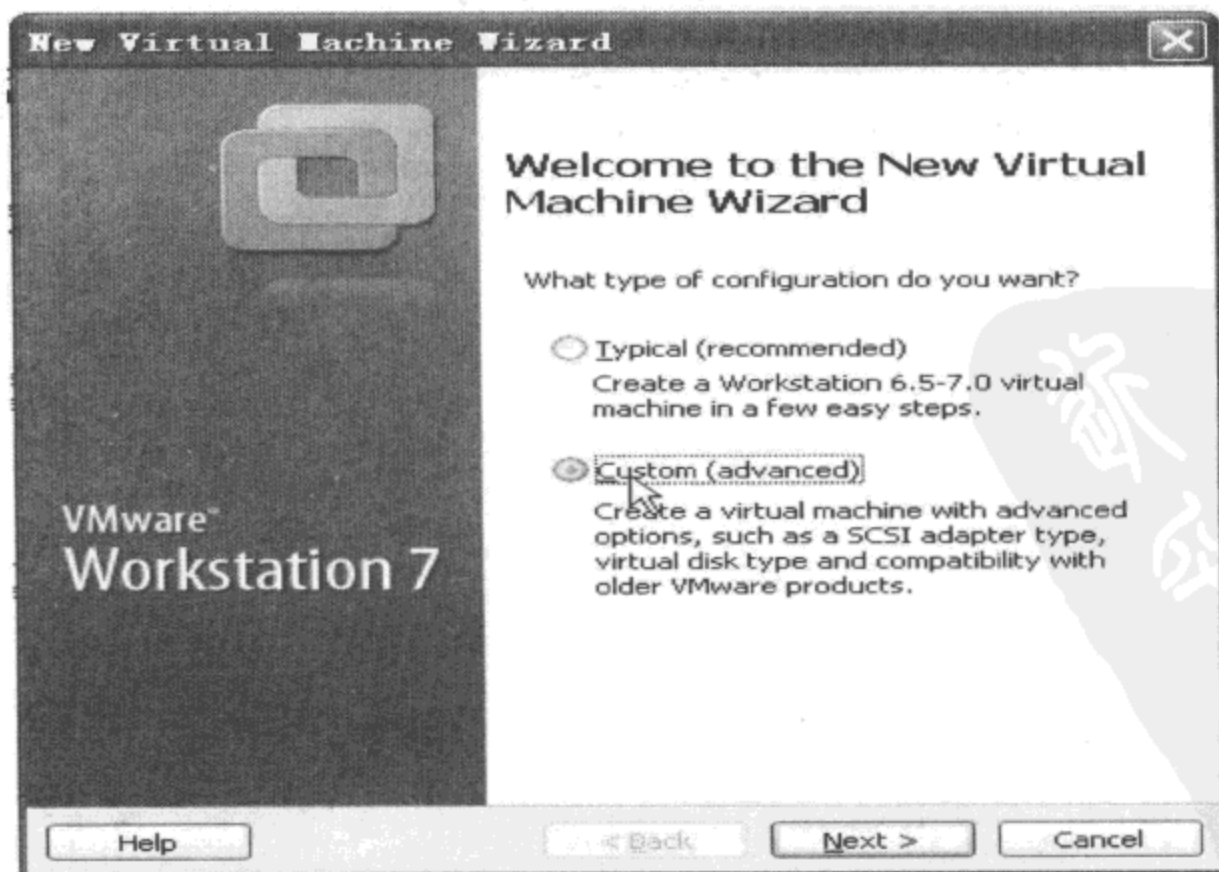


图 4-2 选择定制虚拟机

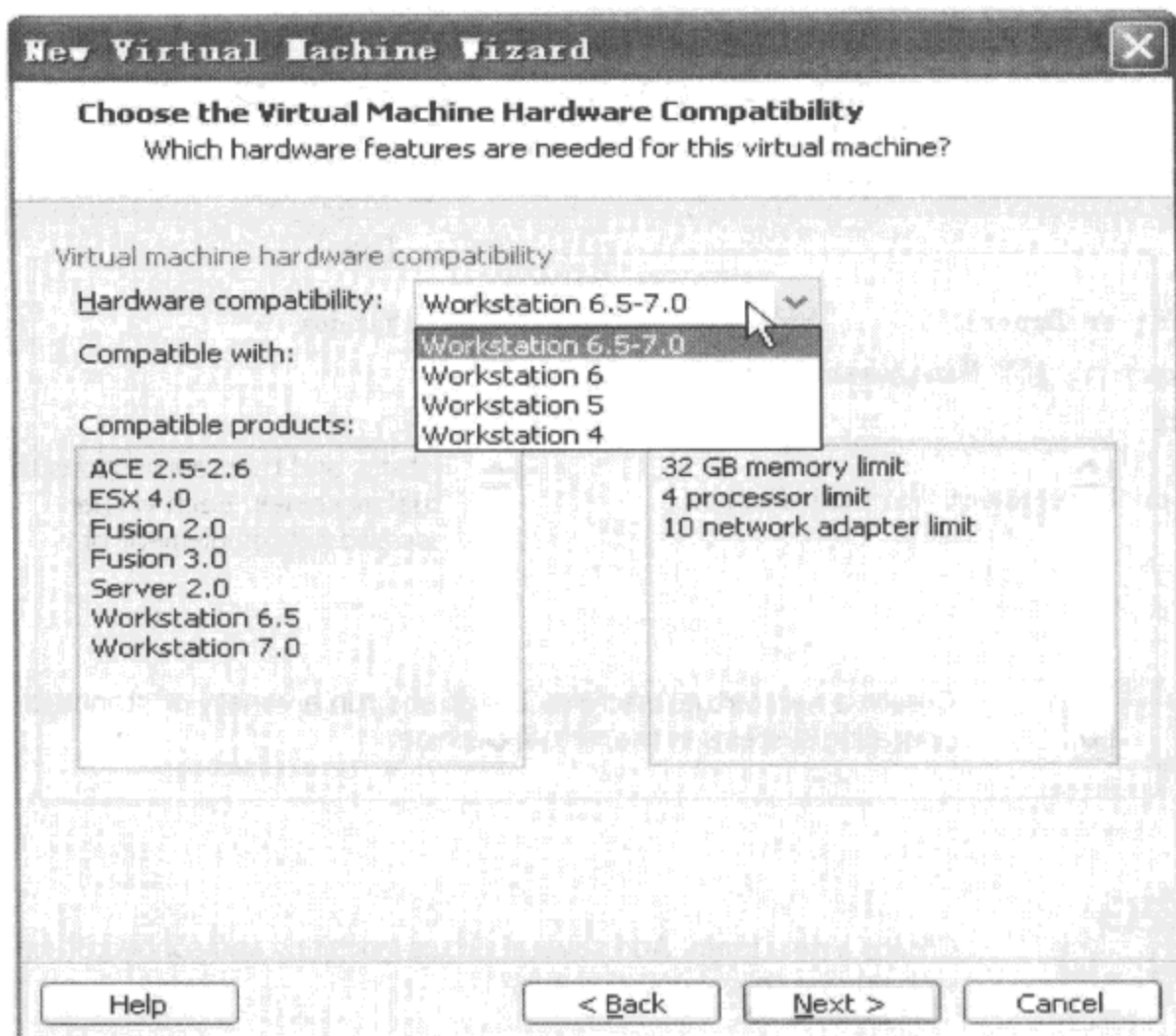


图 4-3 选择虚拟机版本

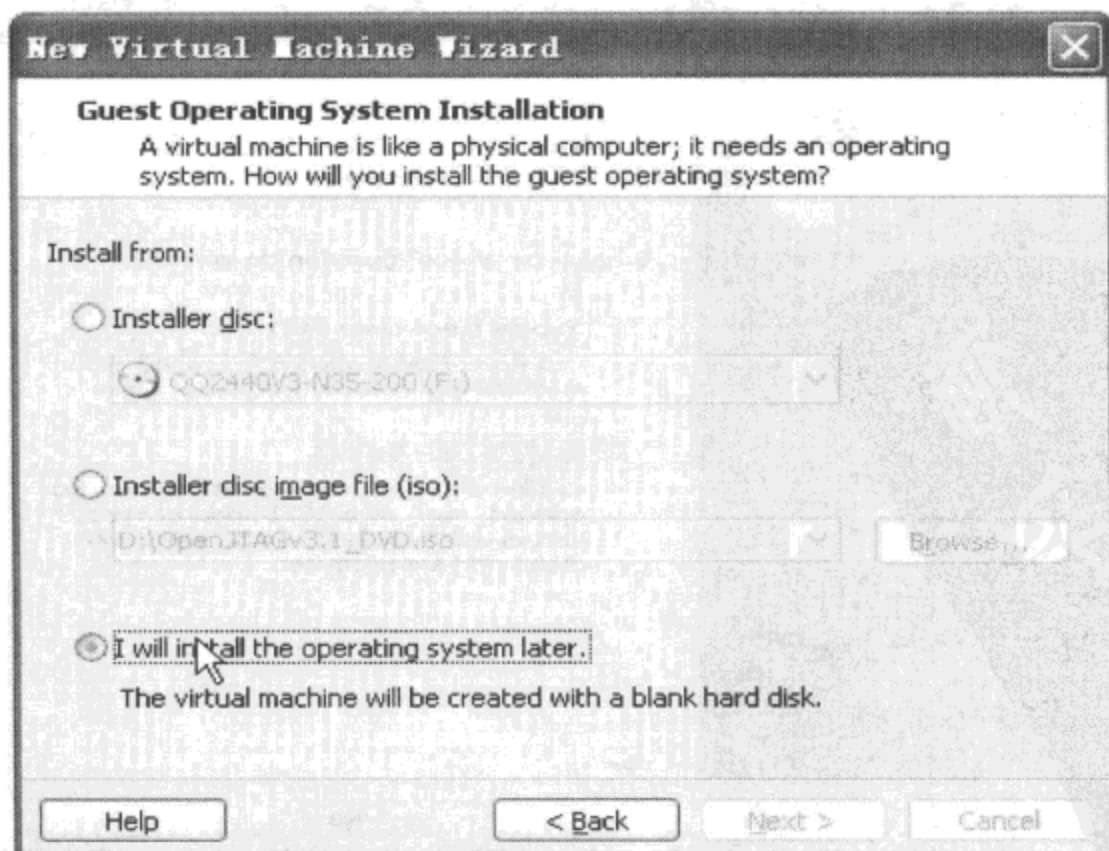


图 4-4 选择虚拟机安装操作系统方式

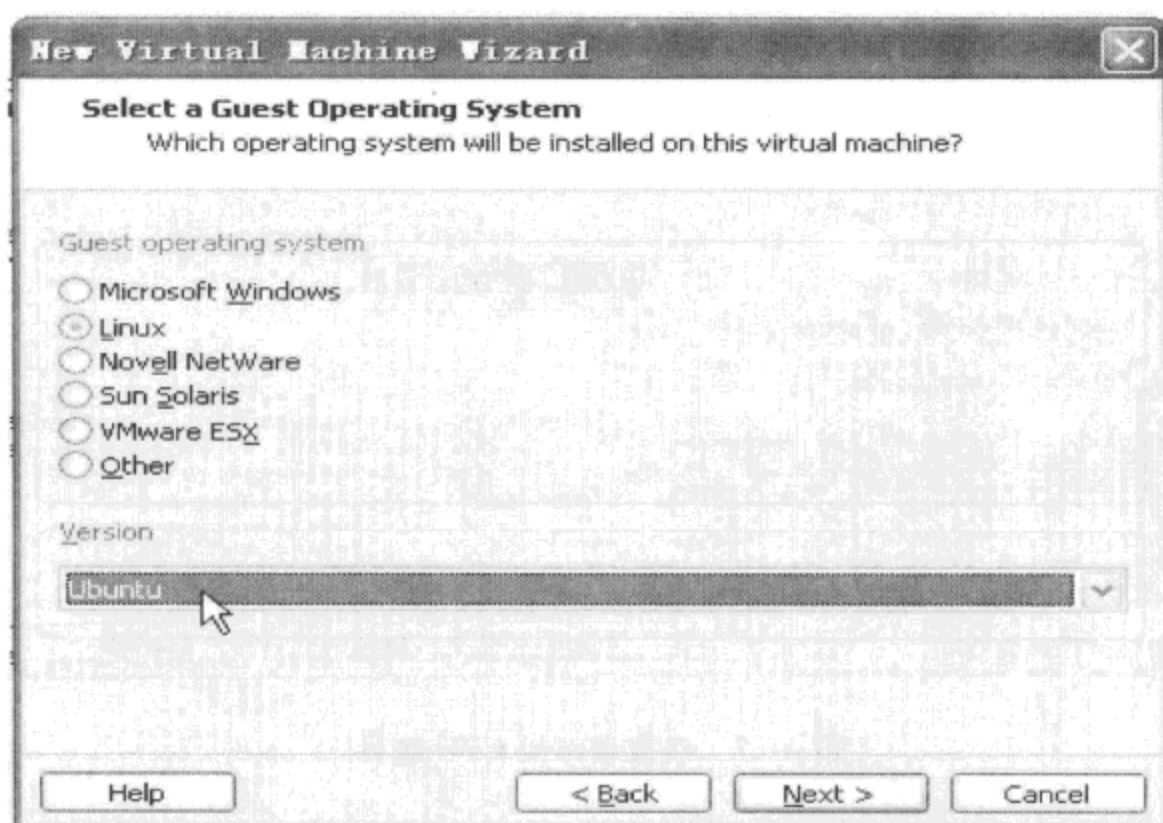


图 4-5 选择虚拟机操作系统

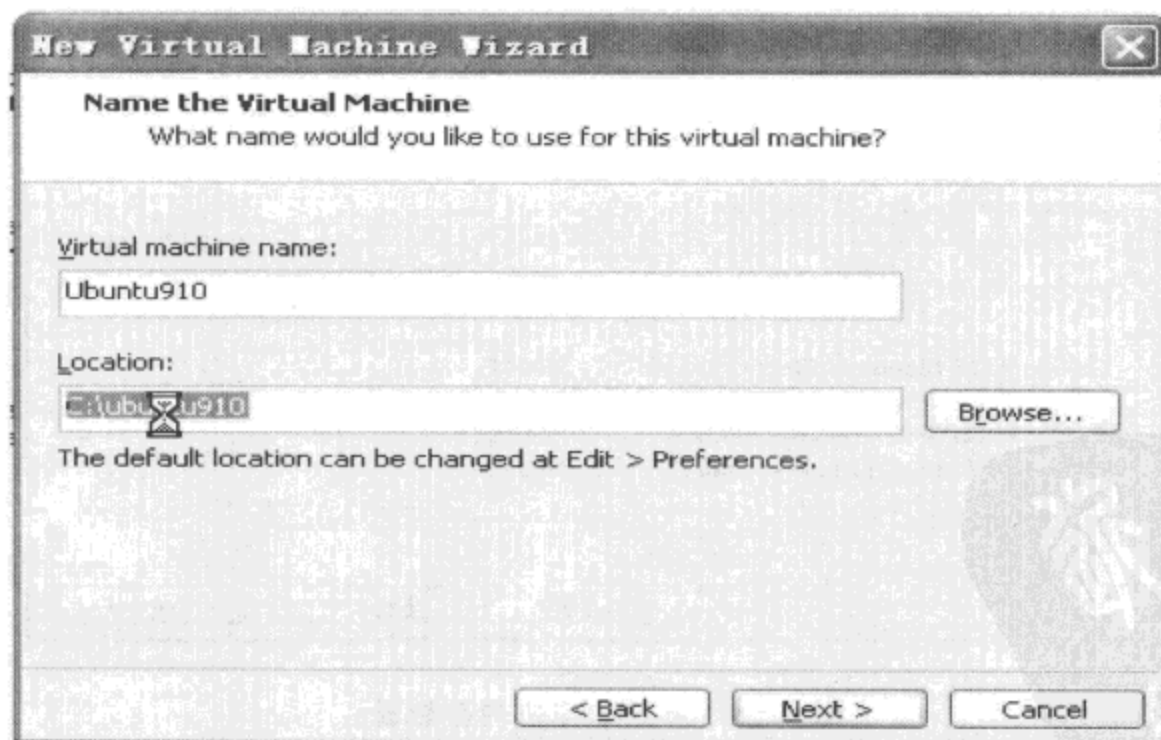


图 4-6 设置虚拟机名字及存储位置

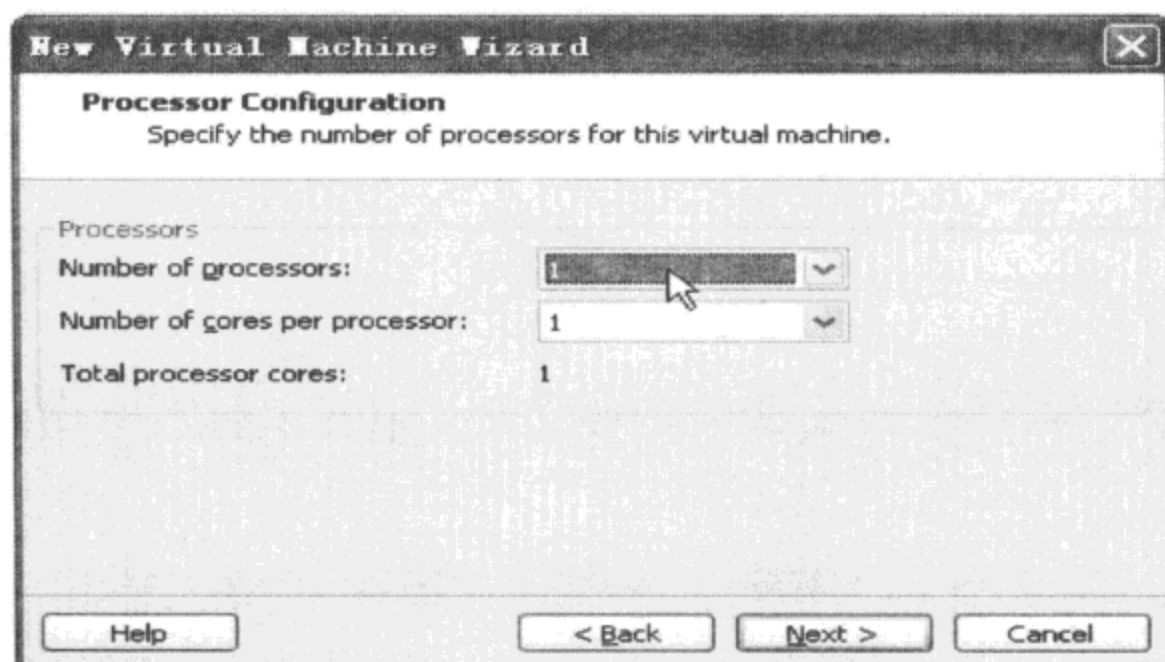


图 4-7 设置虚拟机 CPU 数目

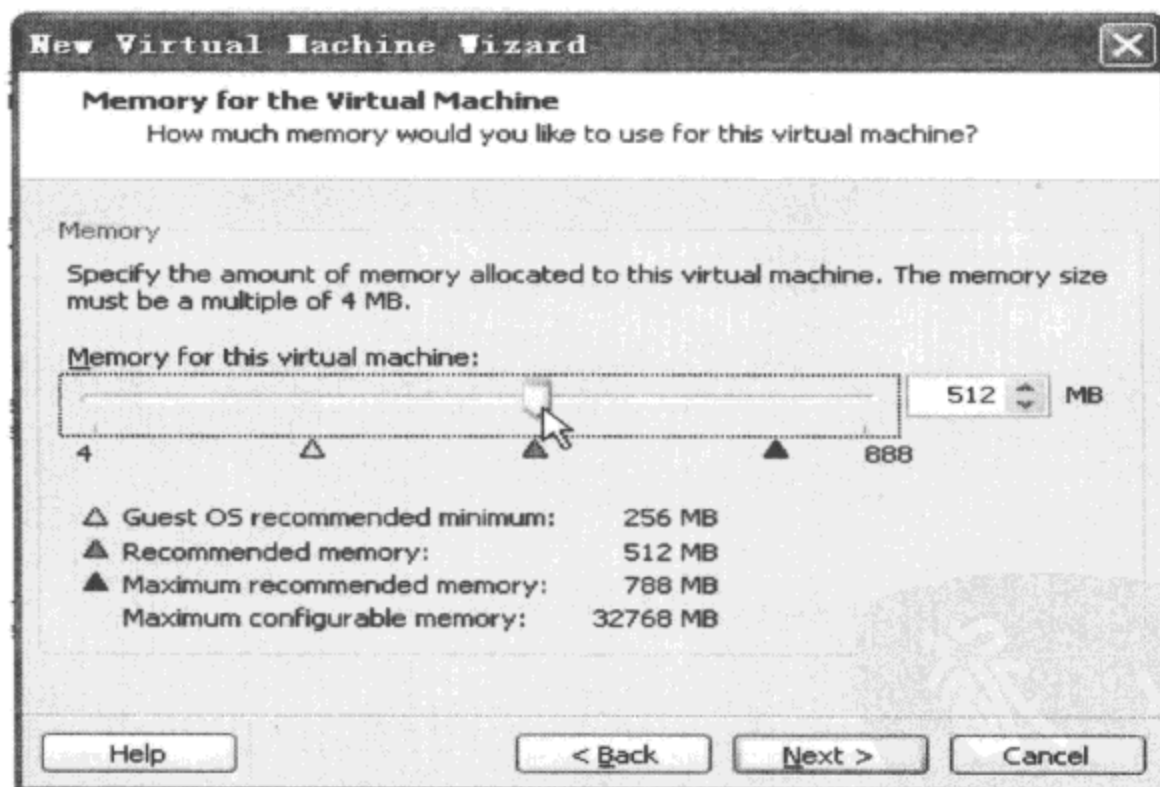


图 4-8 设置虚拟机内存数量

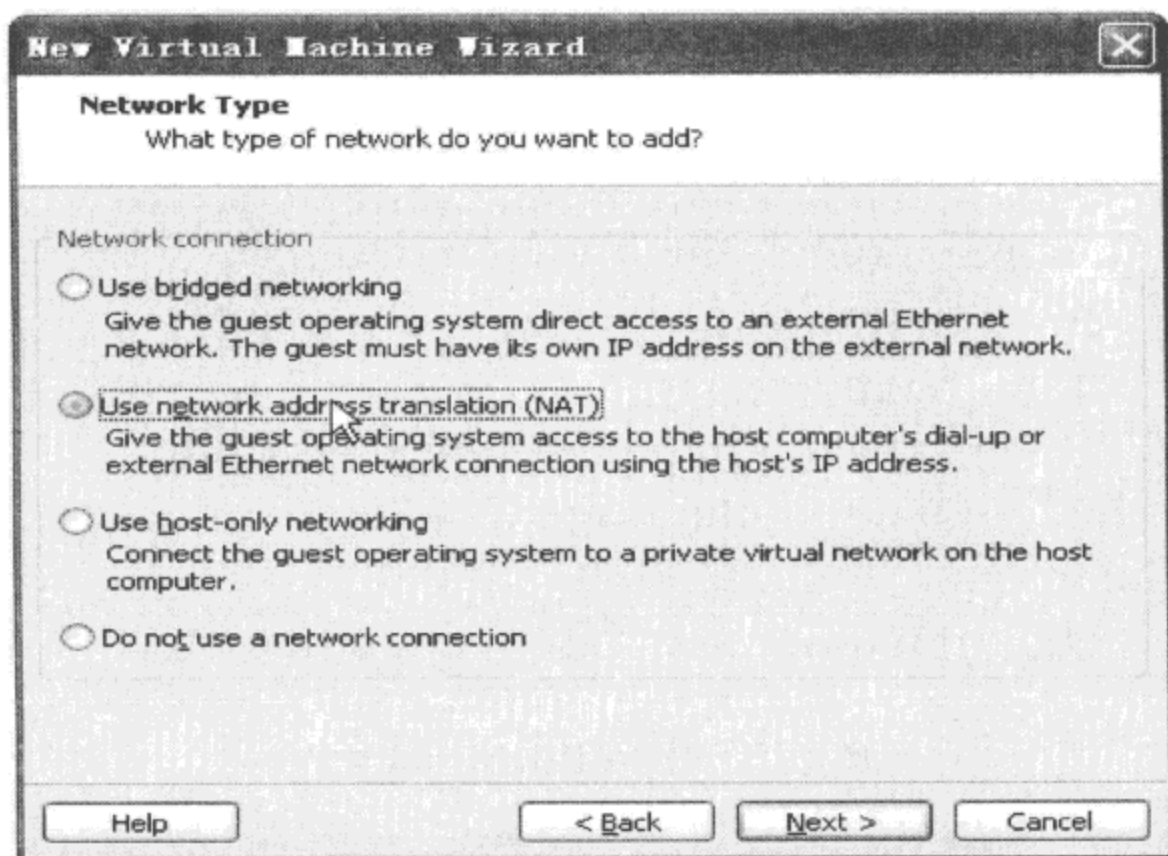


图 4-9 指定虚拟机同主机互联的方式

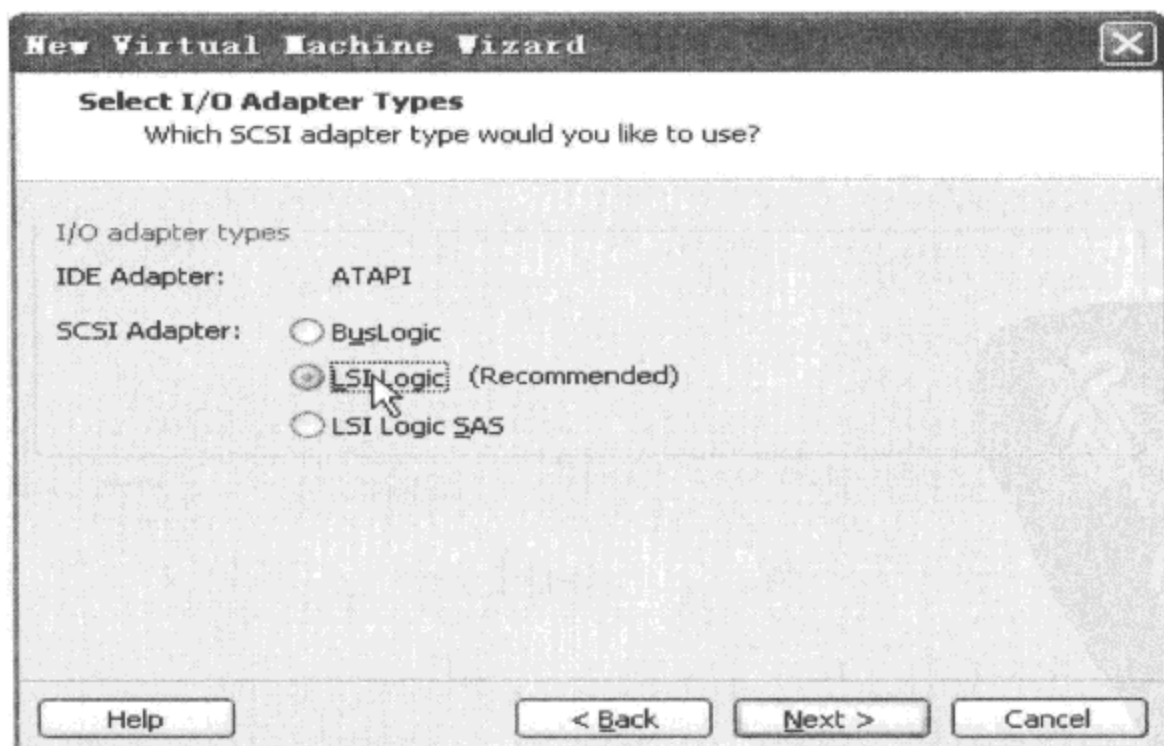


图 4-10 指定虚拟机硬盘控制器类型

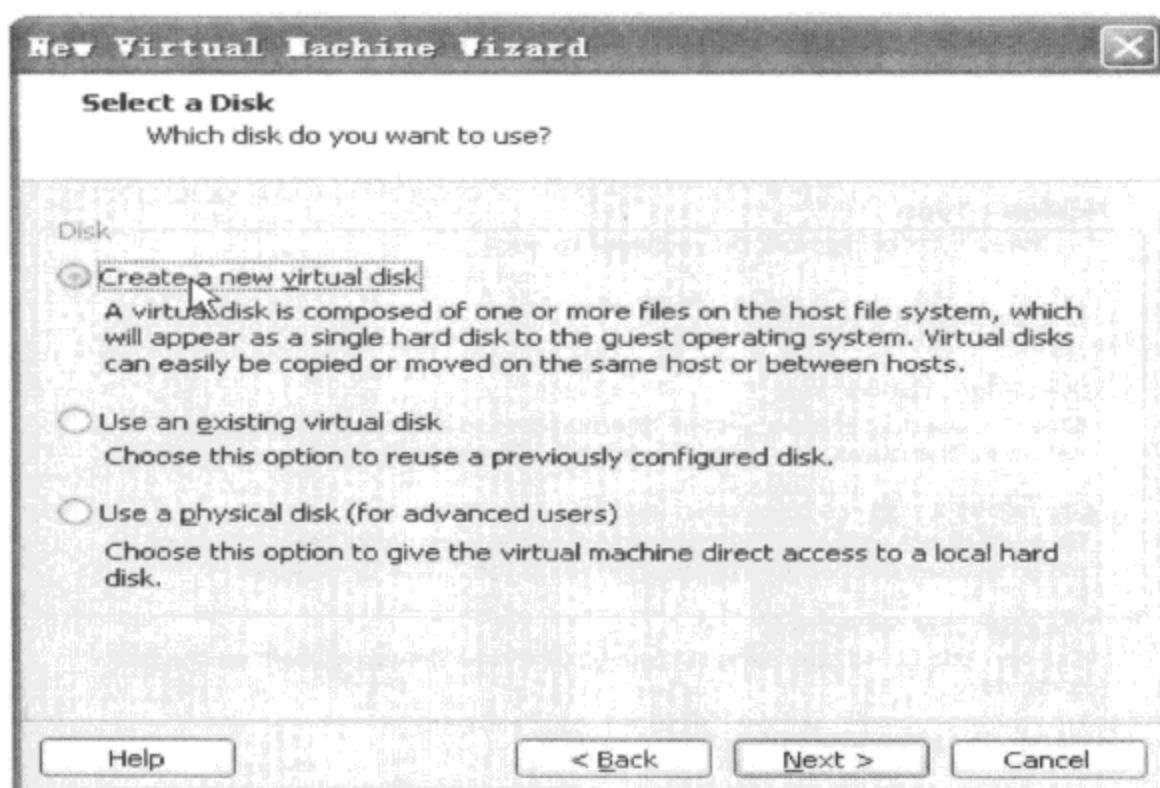


图 4-11 选择创建新的虚拟硬盘

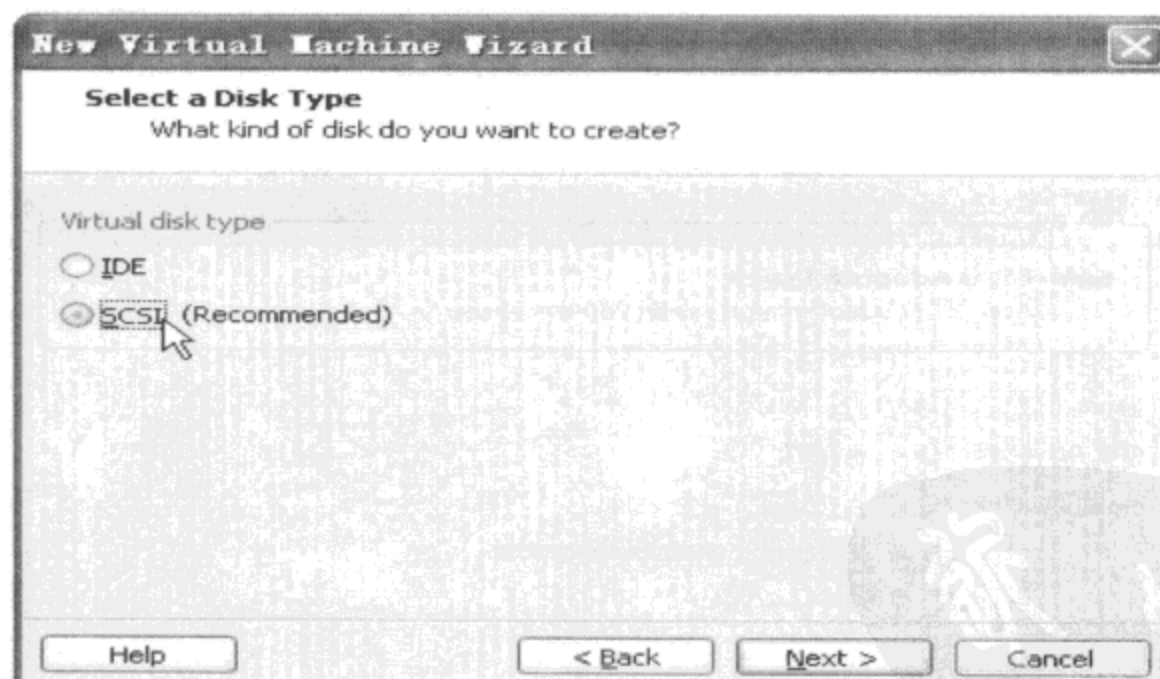


图 4-12 选择硬盘类型

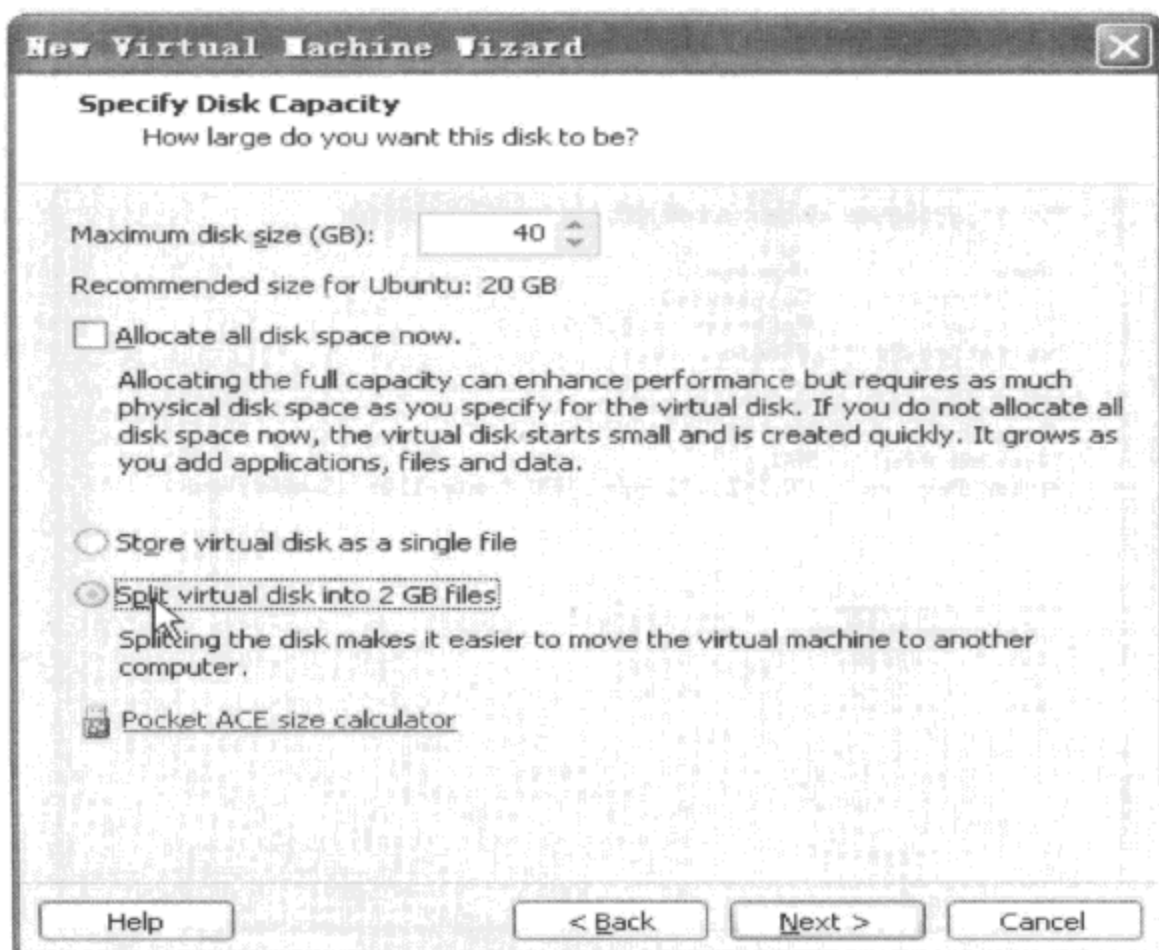


图 4-13 指定虚拟硬盘容量

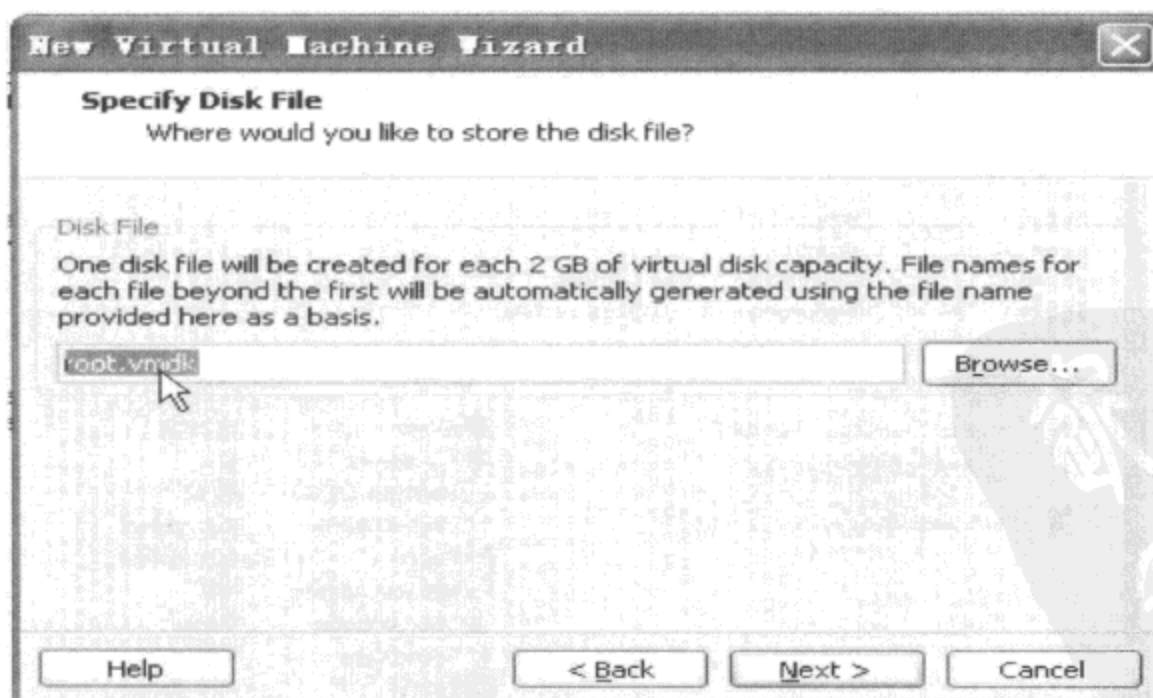


图 4-14 设置虚拟硬盘文件的名字(在 Windows 下将新建一个文件来表示这个虚拟硬盘)

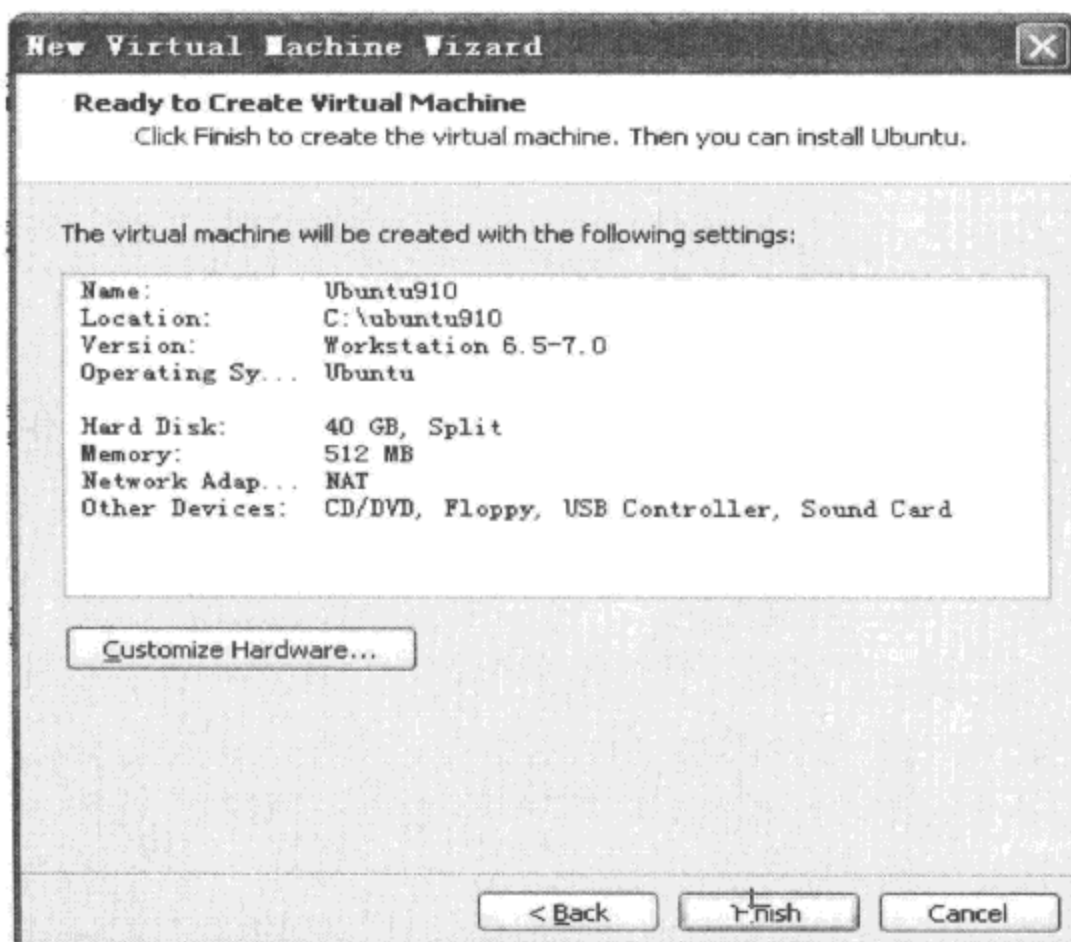


图 4-15 虚拟机设置总结

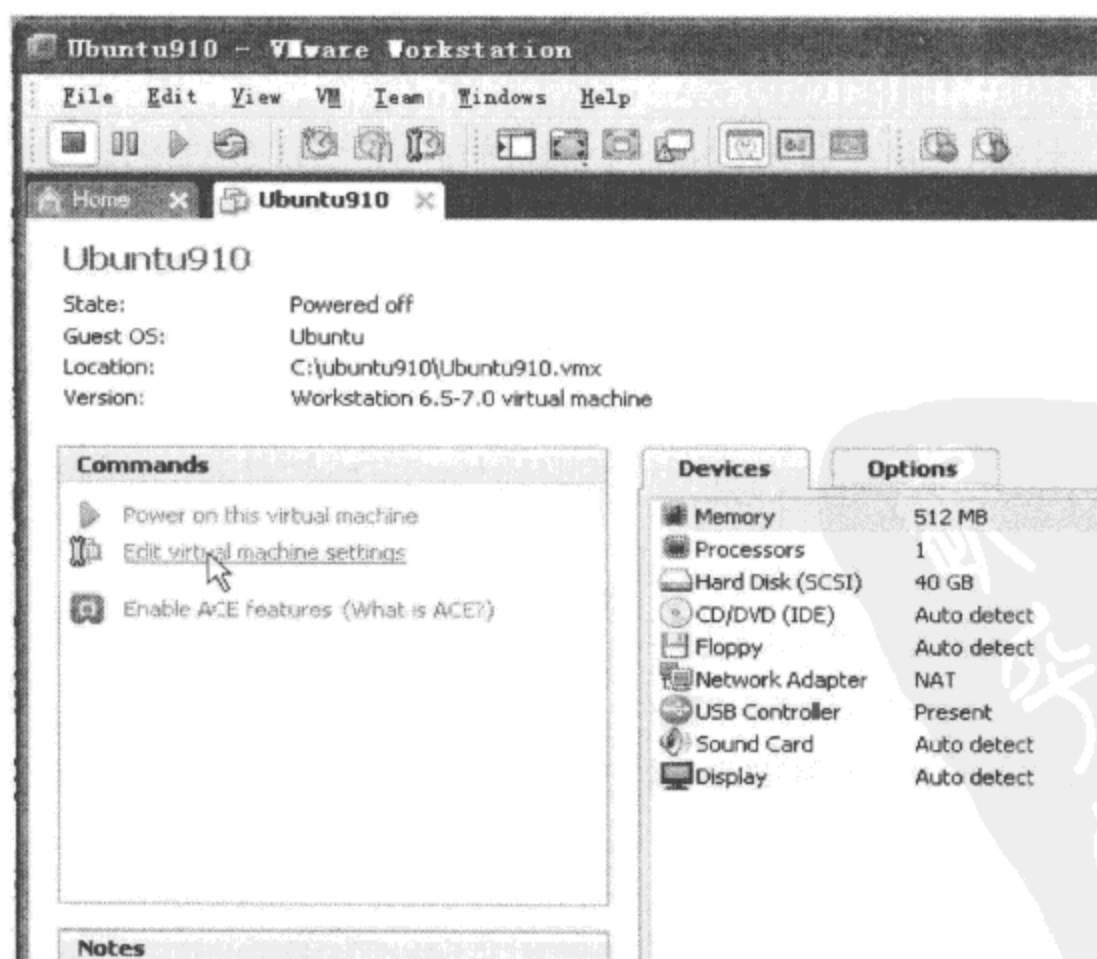


图 4-16 修改虚拟机属性

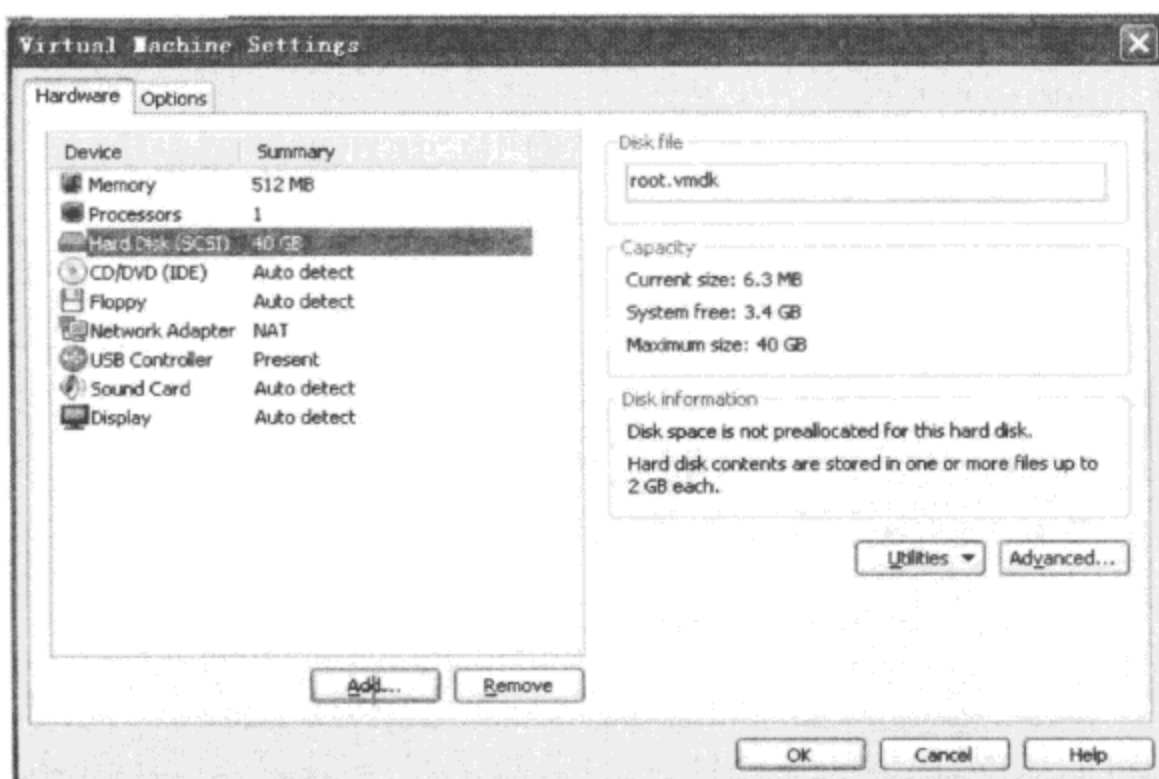


图 4-17 增加新硬件

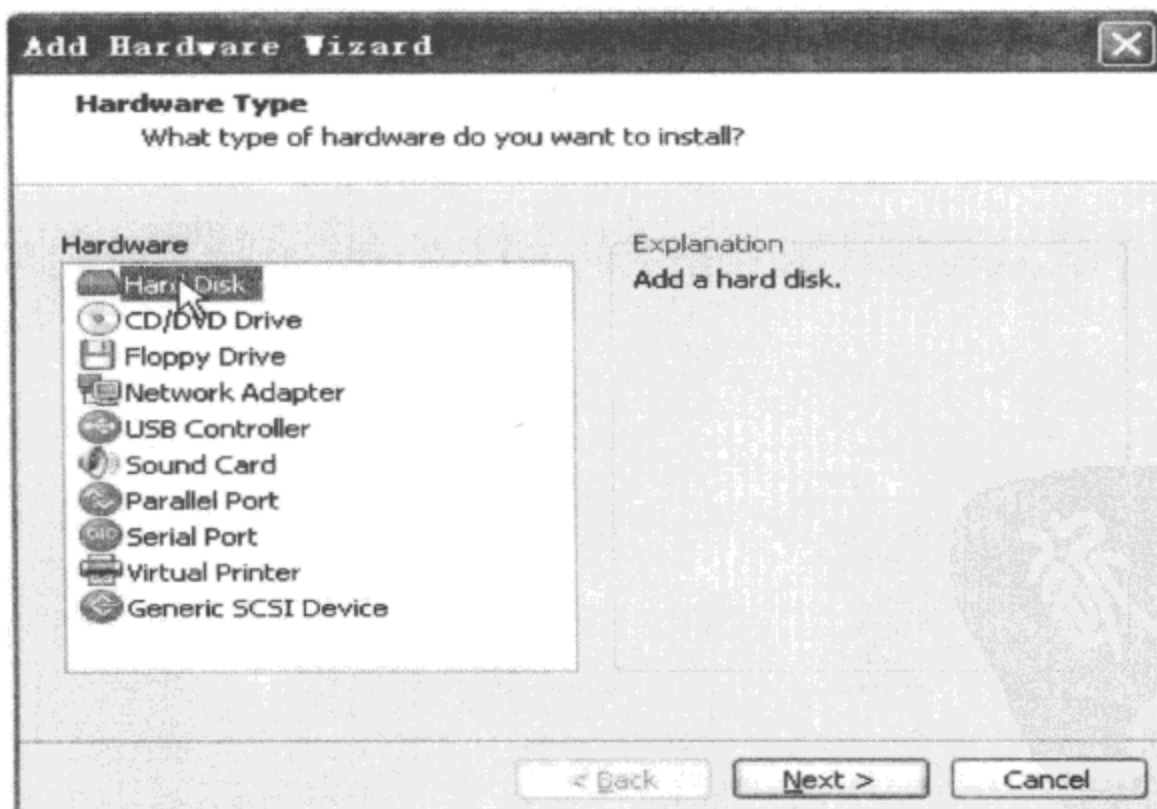


图 4-18 选择增加新硬盘

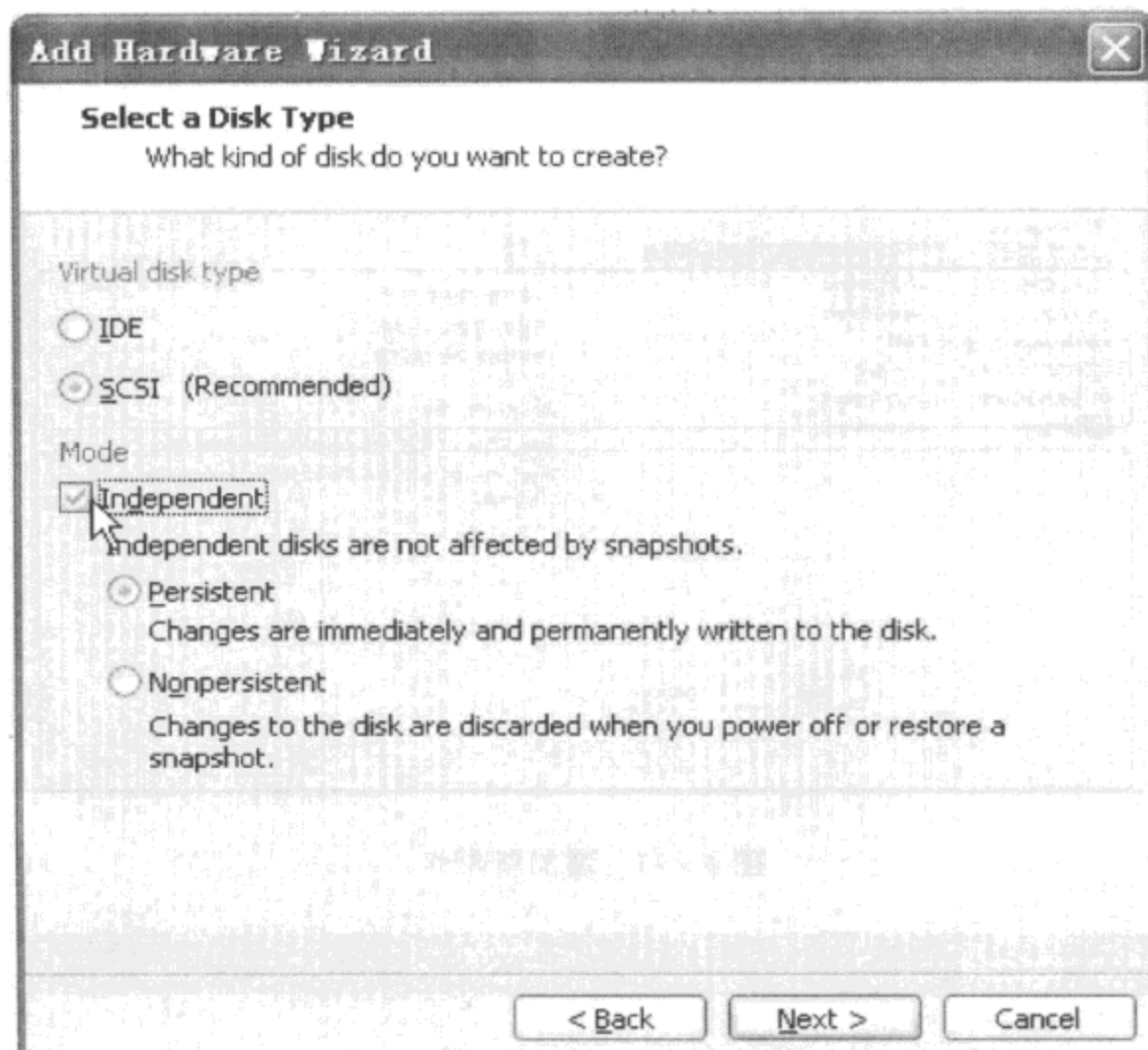


图 4-19 设置硬盘不受 snapshot 影响

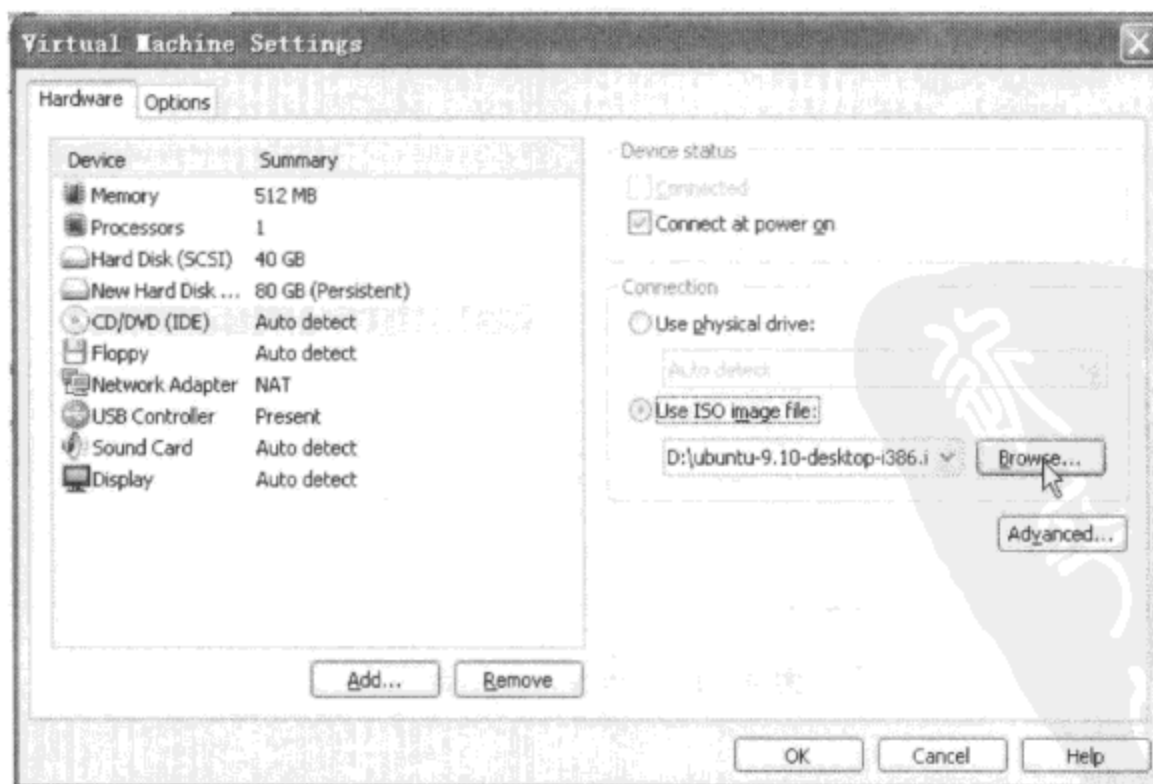


图 4-20 在虚拟机光驱上使用光盘映像文件

4.2.2 在虚拟机上安装 Linux 操作系统 ubuntu 9.10

本书使用 ubuntu 9.10 的光盘映像文件 ubuntu 9.10 - desktop - i386. iso 进行安装。下面介绍关键步骤,其他步骤可以参见安装时出现的说明。

(1) 单击 vmware 7.0 的主菜单:VM→power→power on,启动虚拟机。此时虚拟机会从 ubuntu 9.10 的安装光盘启动,进入安装 ubuntu 的界面,如图 4-21 所示。

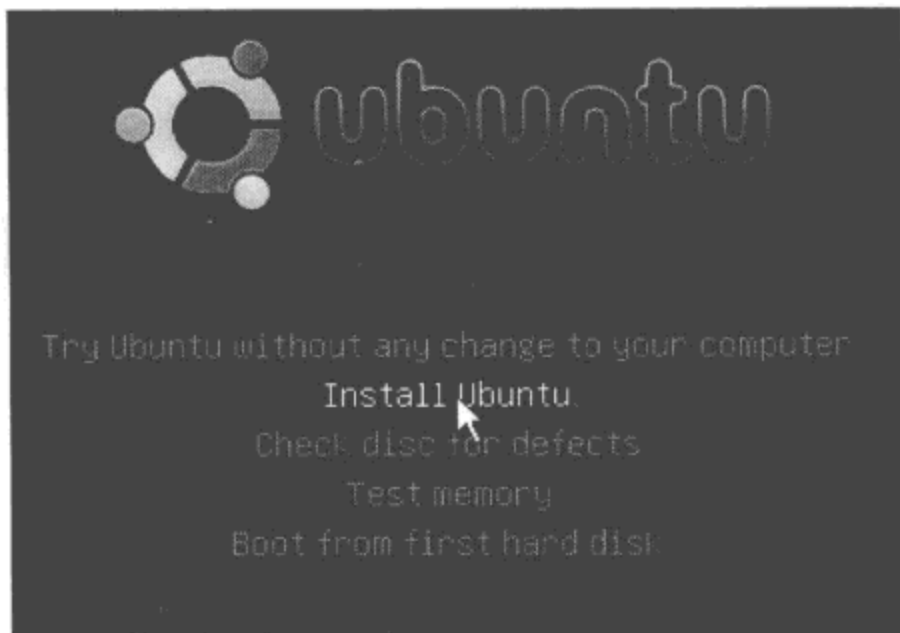


图 4-21 选择安装 ubuntu

(2) 在图 4-21 中选择 Install Ubuntu 选项,会进入 ubuntu 安装的图形界面,以后各个步骤中,大多数情况下只需选择“下一步”即可。

特别说明:此时鼠标和键盘被虚拟机接管,将无法操作 Windows 主机。如想从虚拟机退出到 Windows 主机,按 ctrl+alt 组合键即可;之后如想重新操控虚拟机,请用鼠标单击虚拟机的安装界面。

(3) 安装过程中,当出现图 4-22 时,需要选择 specify partitions manually(advanced)选项,以便手动对两个硬盘进行分区。

(4) 在分区界面中,将第一个硬盘(/dev/sda)分为两个区:/dev/sda1 分区大小 39GB,挂载 root 目录(/),文件系统为 ext3;/dev/sda2 分区大小 1GB,挂载交换分区.swap)。将第二个硬盘(/dev/sdb)划分为一个分区(/dev/sdb1),大小 80GB,挂载/work 目录,文件系统为 ext3。如图 4-23 所示。

特别说明:

- 在 Linux 操作系统中,对于 SCSI 磁盘,用 sdx 来表示,第一个磁盘 x 为 a,第二个磁盘 x 为 b,依次类推。
- 磁盘上的第一个分区编号为 1,第二个分区编号为 2,依次类推。

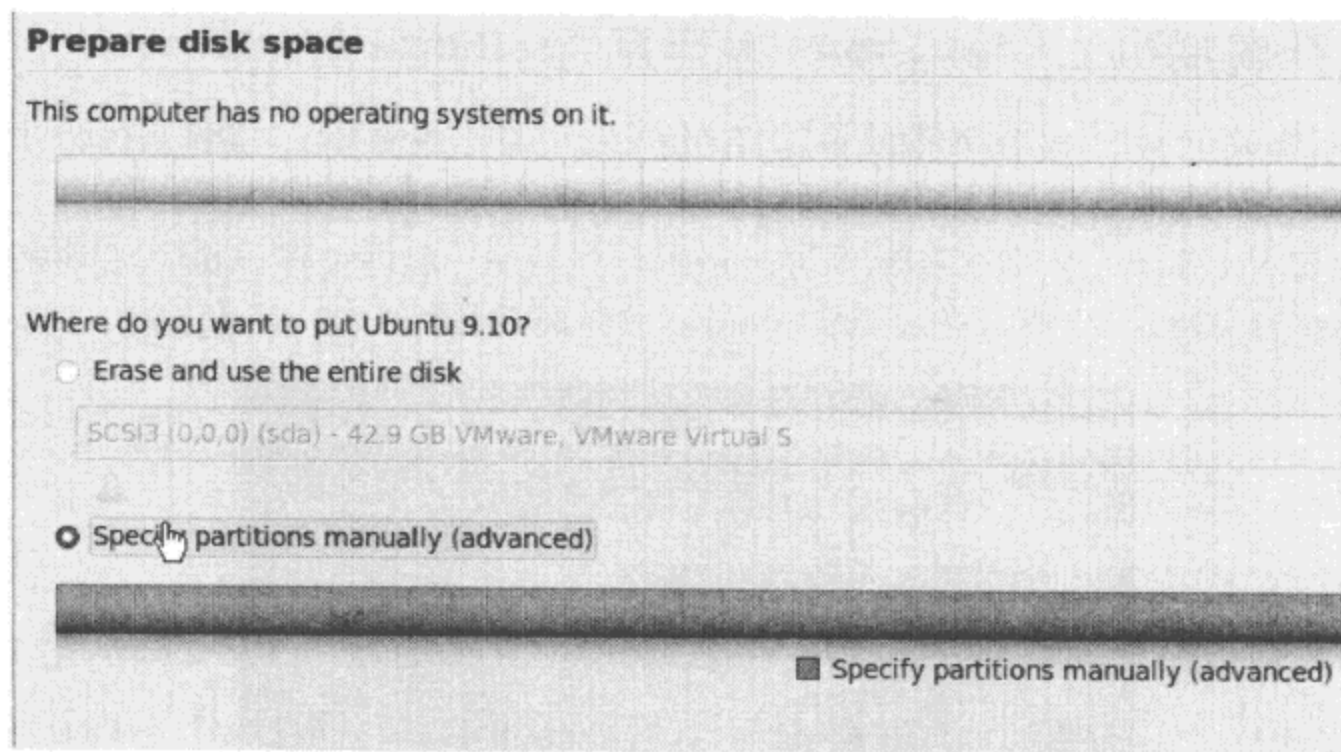


图 4-22 指定手动对硬盘进行分区

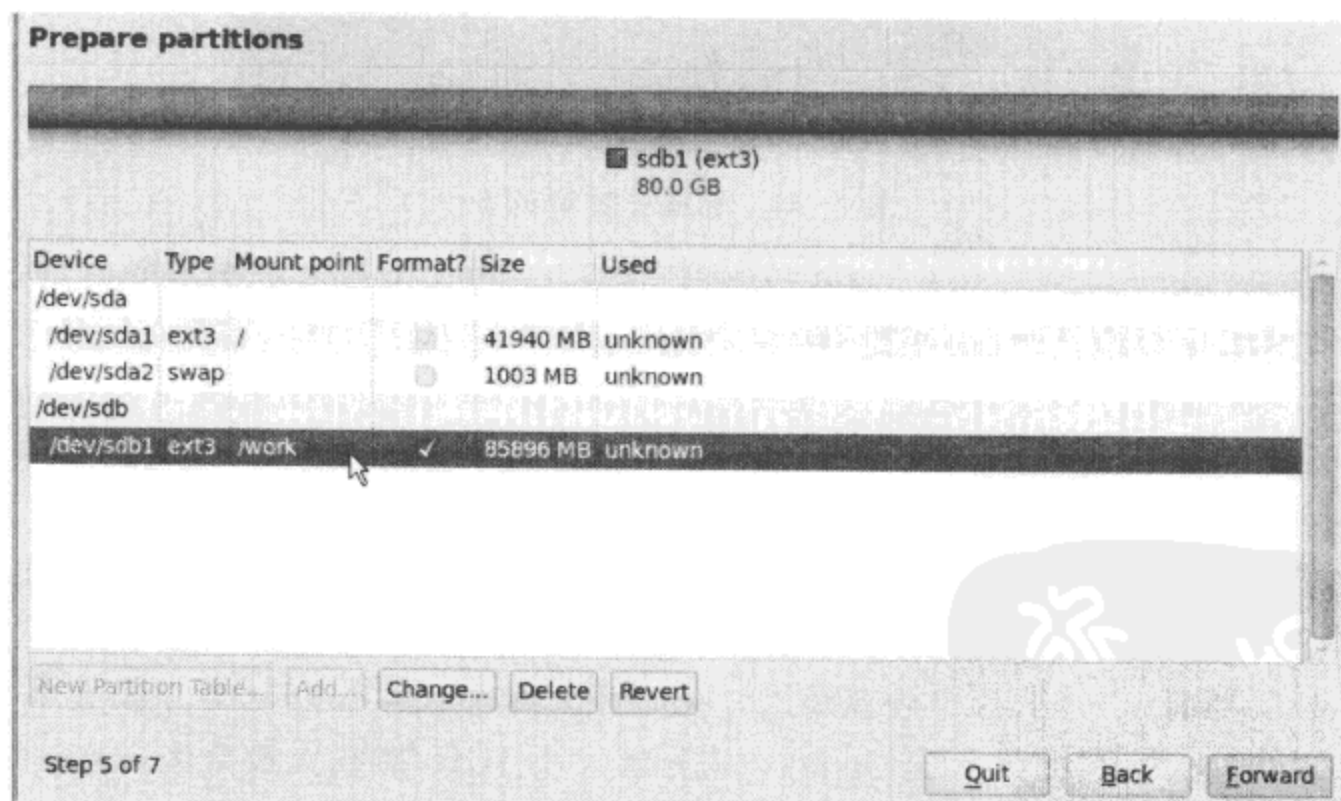


图 4-23 对虚拟机硬盘进行分区

- ext3 文件系统是 Linux 在 PC 上最常用的硬盘文件系统。在嵌入式设备上则常用 jffs2 文件系统和 yaffs2 文件系统。
- swap 分区,用于 Linux 在运行期间的虚拟内存使用,其作用类似 Windows 中的交换文件 pagefile.sys。

(5) 当出现图 4-24 时,设置 ubuntu 的第一个普通用户登录名为 dennis,密码为 1234。用户 dennis 将成为使用 ubuntu 操作系统的主要用户。

Who are you?

What is your name?

What name do you want to use to log in?

If more than one person will use this computer, you can set up multiple accounts after installation.

Choose a password to keep your account safe.

Enter the same password twice, so that it can be checked for typing errors. A good password will contain a mixture of letters, numbers and punctuation, should be at least eight characters long, and should be changed at regular intervals.

What is the name of this computer?

This name will be used if you make the computer visible to others on a network.

☐ Log in automatically
☒ Require my password to log in
☐ Require my password to log in and to decrypt my home folder

Step 6 of 7

Quit Back Forward

图 4-24 设置 ubuntu 中第一个普通用户的登录名和密码

(6) 安装完成后,请务必在图 4-20 中取消点选 Connect at power on 选项,以便使得虚拟机重启后从硬盘启动,而不是从光盘启动。

4.3 在 ubuntu 9.10 中安装基本的开发环境

使用光盘安装的 ubuntu 9.10 是一个比较精简的 Linux 发行版,大部分软件和工具都没有安装,这些工具和软件都需要从互联网上获得和安装,这正是 ubuntu 的一大优点之一,因为只要能联通互联网就能安装数量不断增加、版本不断更新的优质软件,而不必考虑从哪里去获得它们。但对习惯了 Windows setup.exe 的初学者而言,则是一大缺点,因为在使用 ubuntu 的过程中总是完成不了想完成的工作,原因是该工具软件没有安装,而初学者总是无所适从,因为不知道应该如何安装该工具软件。

为了避免在学习本书的过程中遇到这样恼人的尴尬,请按如下步骤在 ubuntu 中安装学习本书的过程中要用到的所有软件。当然要是想快速学习本书,不想浪费时间安装这些工具软件的话,可以直接使用配套光盘中已经制作好的虚拟机(位于光盘的 ubuntu 910 目录中),它

第4章 嵌入式 Linux 软件开发环境搭建

已经安装了学习本书所需的所有软件。使用已制作好的虚拟机的方法是:将配套光盘的整个 ubuntu 910 目录全部复制到 Windows 主机的 C 盘根目录;去掉该目录中所有文件的只读属性;在 vmware 7.0 主菜单中选择 File→Open,找到 C:\ubuntu910 目录下的 Ubuntu 910. vmx 文件,单击打开即可。登录 ubuntu 的用户名为 dennis,密码为 1234。

特别提醒:在安装软件前,务必确保 Windows 主机能够联通互联网,虚拟机与 Windows 主机的联接方式为 NAT。

1. 安装编辑器 vim

~\$ sudo apt-get update

~\$ sudo apt-get install vim

特别说明:

(1) 出于操作安全考虑,默认情况下 ubuntu 不允许 root 用户登录,所以一般情况下都使用普通用户 dennis 登录系统进行操作,而安装软件的操作需要 root 权限,所以需要在执行 apt-get 前加上 sudo 命令,表示以管理员的身份执行 apt-get 命令。此后系统会要求你输入密码,你只需要输入 dennis 的密码 1234 即可。

(2) apt-get 是 ubuntu 系统提供的从互联网上安装软件的命令行工具,install 是 apt-get 的参数,表示安装。常用的参数还有 uninstall 表示卸载软件,update 表示更新软件数据库,以获得最新可安装软件的列表。

(3) vim 是要安装的软件的名字。

2. 安装基本的开发环境

~\$ sudo apt-get install build-essential

3. 安装语法、词法分析器

~\$ sudo apt-get install bison flex

4. 安装 C 以及 C++ 函数库 man 手册

~\$ sudo apt-get install manpages-dev manpages-posix-dev glibc-doc

5. 安装串口工具 minicom 和 ckermi

~\$ sudo apt-get install minicom ckermi

6. 安装已经制作好的 arm 平台的交叉编译工具链

交叉编译工具链是进行嵌入式开发必须具备的基础工具。它的获得,最原始的方法是下载编译器 gcc、C 库 glibc、二进制工具 binutil 的源代码,然后进行复杂的编译过程得到。该方法需要制作者具备很强的关于编译器和 C 库的知识和经验,而且制作过程极其不易成功。因此本书光盘中提供了制作好的交叉编译工具链(wok/sysbuild/arm-Linux-gcc-3.4.5-glibc-2.3.6.tar.bz2),只需要解压即可。该文件如何从 Window 主机上传到 linux 虚拟机,请参见

“ubuntu 9.10 上网络服务的安装与配置”的FTP服务器部分。

(1) 将交叉编译工具链解压到/usr/local/arm:

```
$ sudo tar xjvf arm-linux-gcc-3.4.5-glibc-2.3.6.tar.bz2 -C /usr/local/arm
```

说明:

① tar 是 Linux 下的打包、解包命令,命令中的选项 x 表示解包,对应的打包的选项是 c。

② 选项 j 表示在解包(打包)过程中调用 bzip2 进行解压缩(压缩),对应的 z 选项表示调用 gzip。后缀名为 bz2 的文件通常是由 bzip2 进行压缩的,而后缀名为 gz 的文件通常是由 gzip 压缩的。

③ 选项 f 表示后面跟的参数是要打包或解包的文件的名称。

④ -C 后跟的参数表示要解包到的目标位置。

(2) 修改交叉编译工具链目录权限:

```
~ $ sudo chown dennis:dennis /usr/local/arm/gcc-3.4.5-glibc-2.3.6-R
```

执行这个操作的原因是:本书学习过程中要编译一些新的 C 库文件并将其复制到交叉编译工具链的相关目录中。

(3) 修改 PATH 环境变量,使其包含交叉编译器的路径:

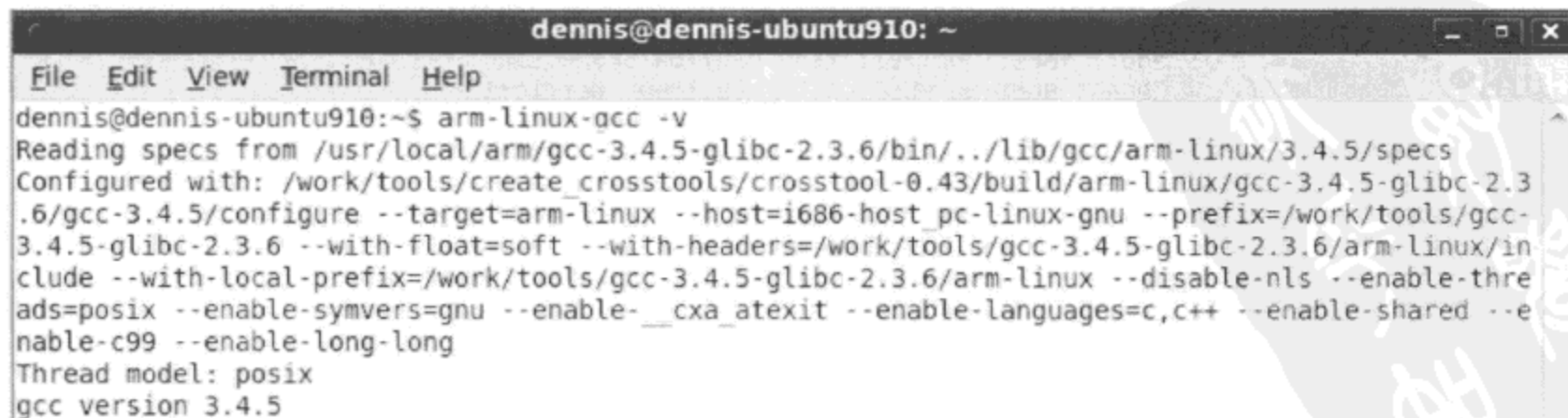
使用 vim,将/etc/environment 文件中的内容

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games"
```

改为

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/arm/gcc-3.4.5-glibc-2.3.6/bin"
```

退出系统重新登录后,输入 arm-linux-gcc-v,如出现图 4-25 所示的内容。则表示交叉编译工具链安装成功。



```
dennis@dennis-ubuntu910: ~
File Edit View Terminal Help
dennis@dennis-ubuntu910:~$ arm-linux-gcc -v
Reading specs from /usr/local/arm/gcc-3.4.5-glibc-2.3.6/bin/../lib/gcc/arm-linux/3.4.5/specs
Configured with: /work/tools/create_crosstools/crosstool-0.43/build/arm-linux/gcc-3.4.5-glibc-2.3.6/gcc-3.4.5/configure --target=arm-linux --host=i686-host_pc-linux-gnu --prefix=/work/tools/gcc-3.4.5-glibc-2.3.6 --with-float=soft --with-headers=/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux/include --with-local-prefix=/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux --disable-nls --enable-threads=posix --enable-symvers=gnu --enable-__cxa_atexit --enable-languages=c,c++ --enable-shared --enable-c99 --enable-long-long
Thread model: posix
gcc version 3.4.5
```

图 4-25 验证交叉编译工具链安装成功

7. 安装移植 qtopia 的时候所需的工具

```
$ sudo apt-get install x-dev libx11-dev x11proto-xext-dev libxext-dev libqt3-mt-dev
```

uuid uuid-dev

8. 安装配置内核(make menuconfig)时需要的 ncurses 库

```
$ sudo apt-get install libncurses5 libncursesw5 libncurses5-dev ncurses-base
ncurses-bin
```

9. 安装制作根文件系统的 jffs2 映像时需要的工具软件

```
$ sudo apt-get install mtd-utils
```

10. 安装编译 tslib 时需要的工具软件

```
$ sudo apt-get install m4 autoconf automake libtool
```

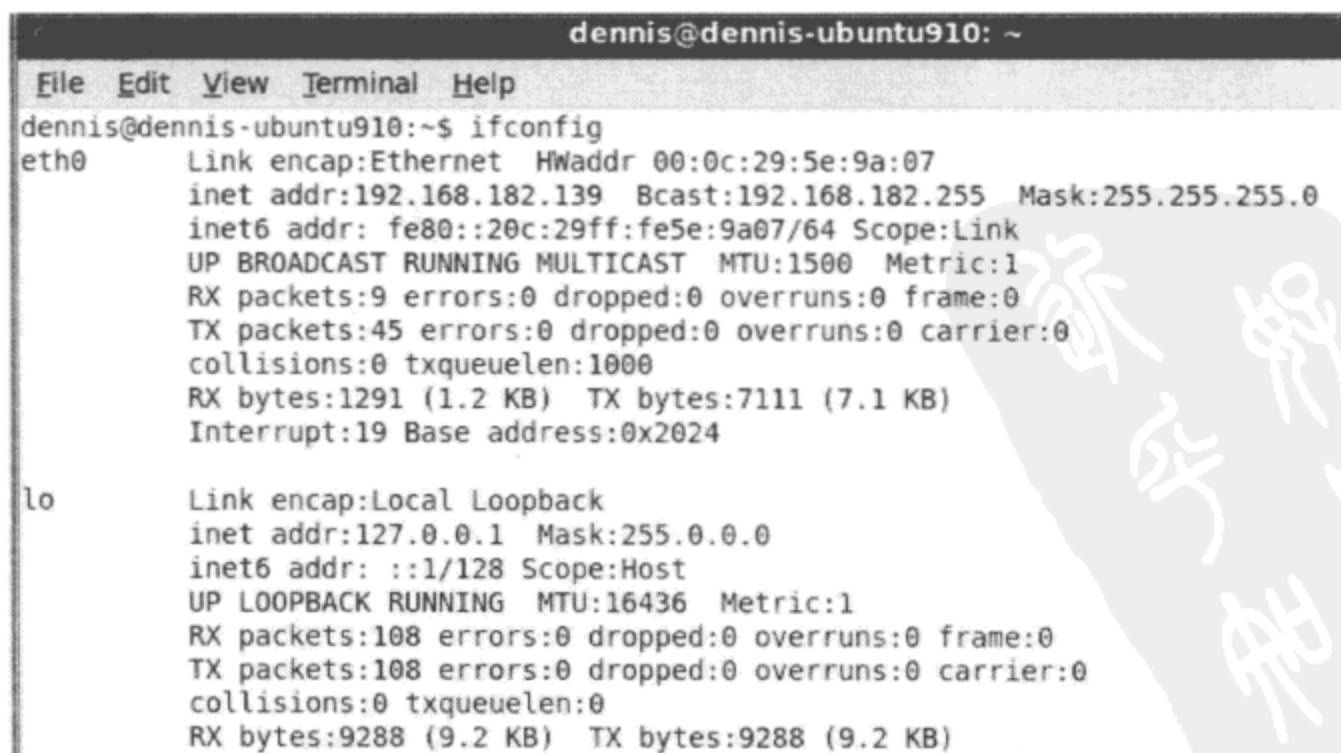
4.4 ubuntu 9.10 上网络服务的安装与配置

在 ubuntu 上配置两个网络服务:FTP 服务和 NFS 服务。前者用于在 Windows 主机与 Linux 虚拟机之间交换数据,后者用于在开发板和 Linux 虚拟机之间交换数据。

4.4.1 设置 vmware 网络

目前虚拟机只有一张网卡,其被设置为 NAT 的连接方式,可以通过它实现 Windows 主机与 Linux 虚拟机的通信;此外,还需要为虚拟机增加一张网卡,并将其桥接到 Windows 主机的物理网卡上,通过它可实现 Linux 虚拟机与开发板的通信。

(1) Linux 虚拟机未添加第二张网卡前(图 4-26)。



```
dennis@dennis-ubuntu910: ~
File Edit View Terminal Help
dennis@dennis-ubuntu910:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:5e:9a:07
          inet addr:192.168.182.139  Bcast:192.168.182.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe5e:9a07/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9 errors:0 dropped:0 overruns:0 frame:0
          TX packets:45 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1291 (1.2 KB)  TX bytes:7111 (7.1 KB)
          Interrupt:19 Base address:0x2024

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:108 errors:0 dropped:0 overruns:0 frame:0
          TX packets:108 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9288 (9.2 KB)  TX bytes:9288 (9.2 KB)
```

图 4-26 添加第二张网卡前

(2) 为虚拟机添加第二张网卡,并设置为桥接(图4-27~图4-30)。

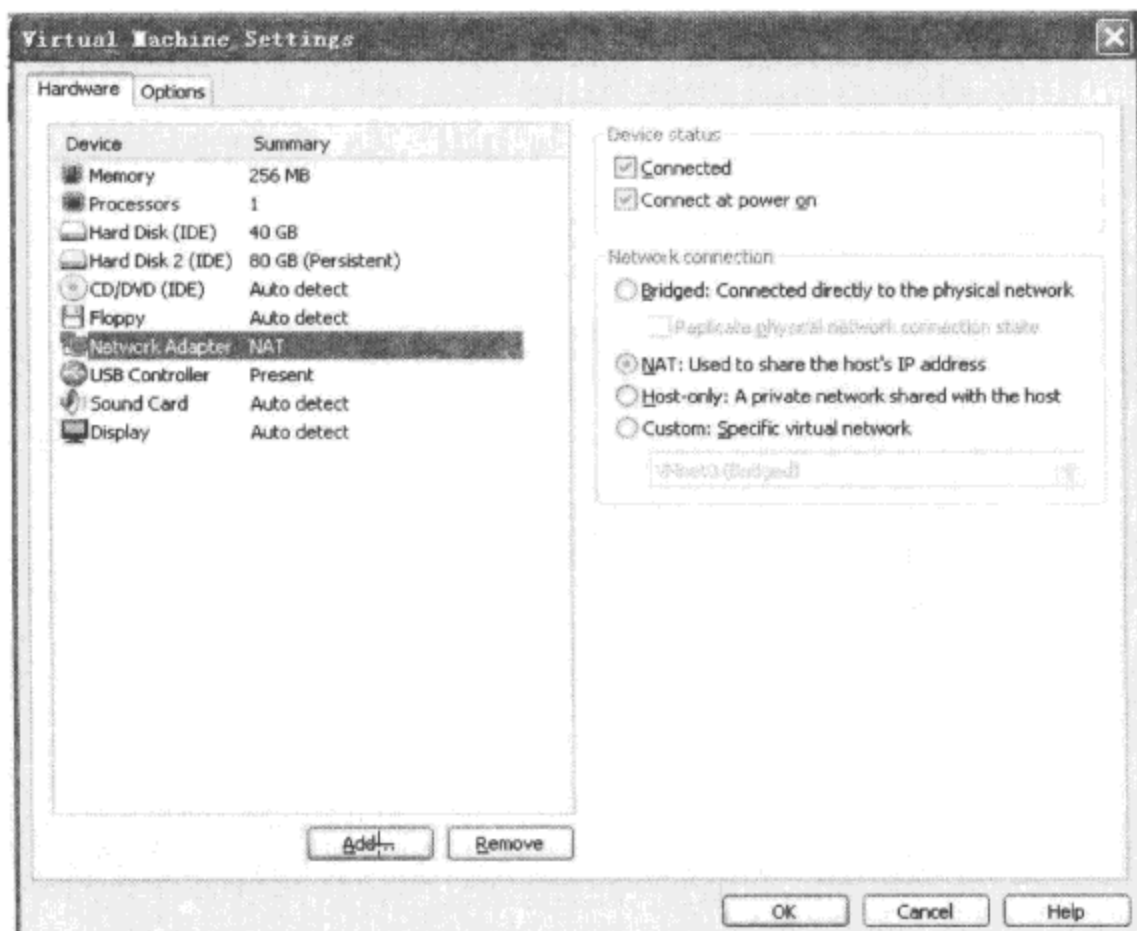


图 4-27 虚拟机设备页

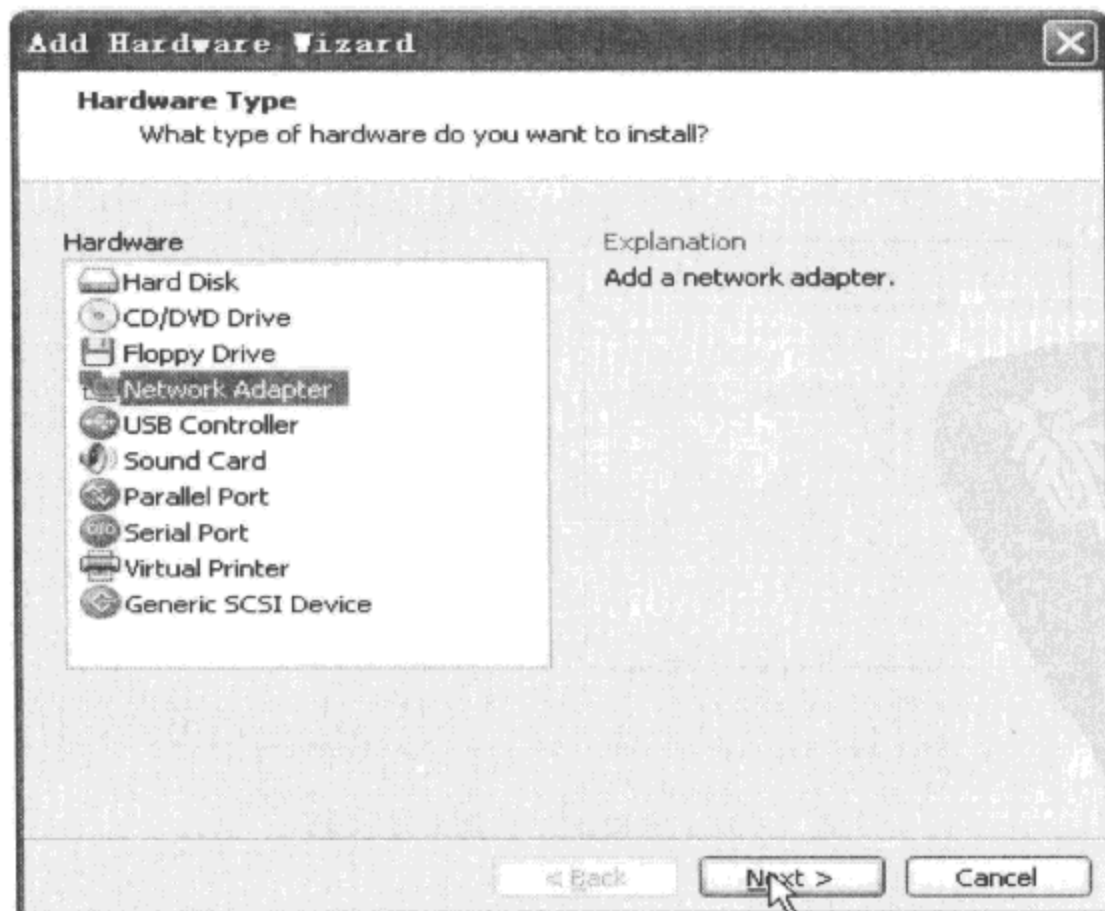


图 4-28 添加网卡

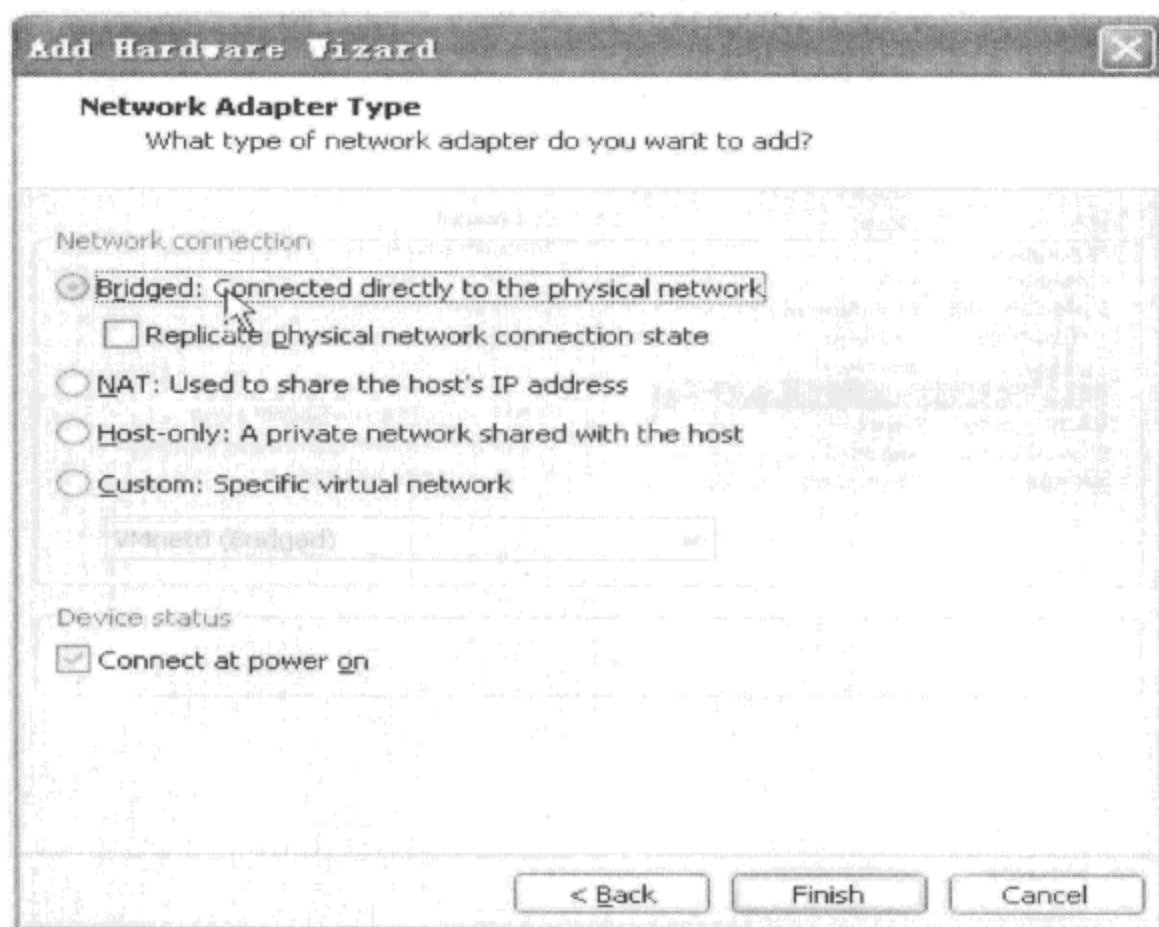


图 4-29 指定新网卡桥接到主机物理网卡

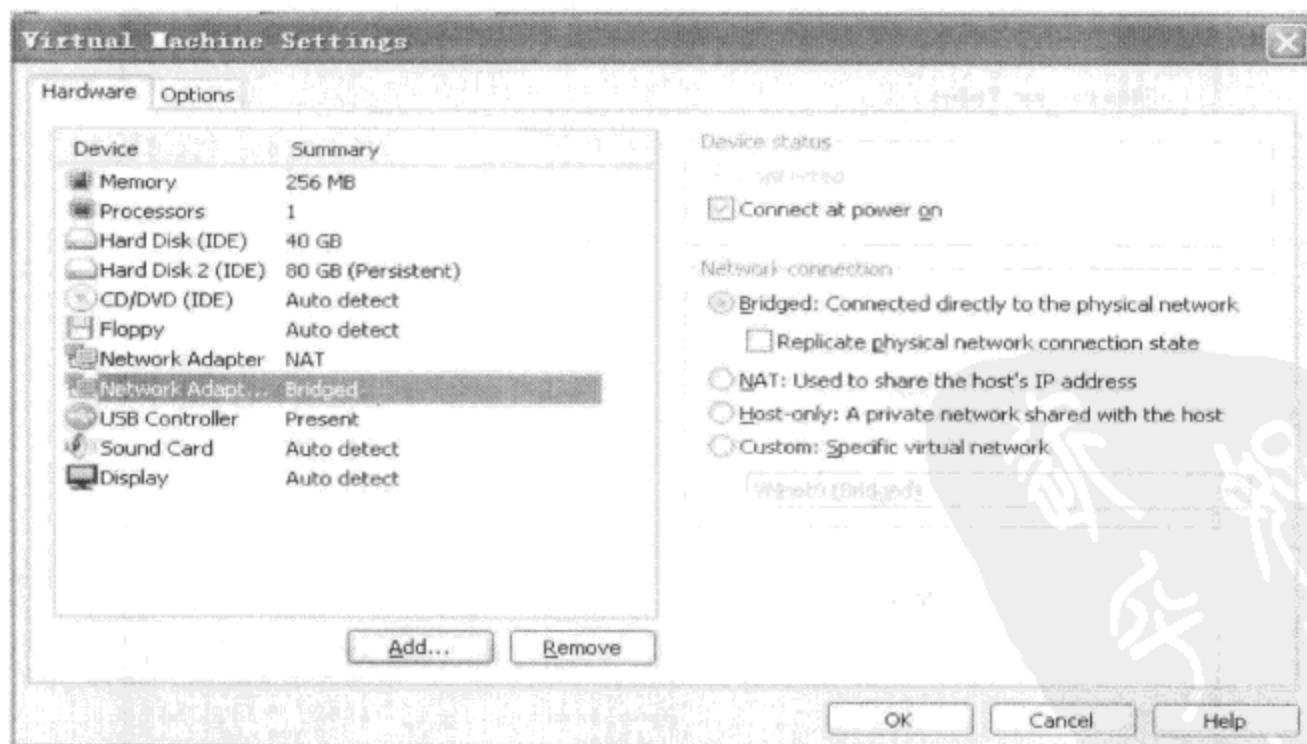


图 4-30 完成第二张网卡添加后

(3) 在Linux虚拟机中为第二张网卡设置IP地址。

本书假设:Windows主机的IP为192.168.1.12, Linux虚拟机第二张网卡的IP地址为192.168.1.11, 开发板的IP地址为192.168.1.17。

按照图4-31~图4-33所示, 设置第二张网卡的IP为192.168.1.11, 第一张网卡的IP保持自动获得不变。

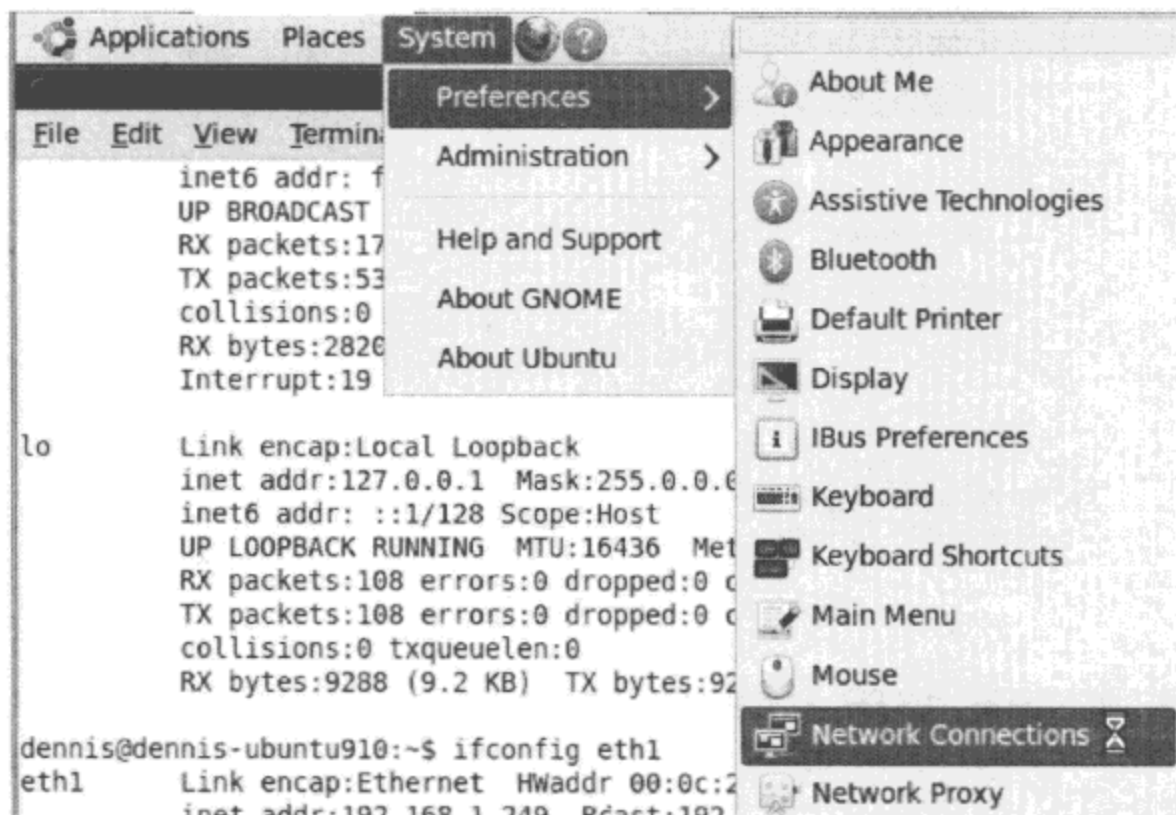


图4-31 网络配置工具

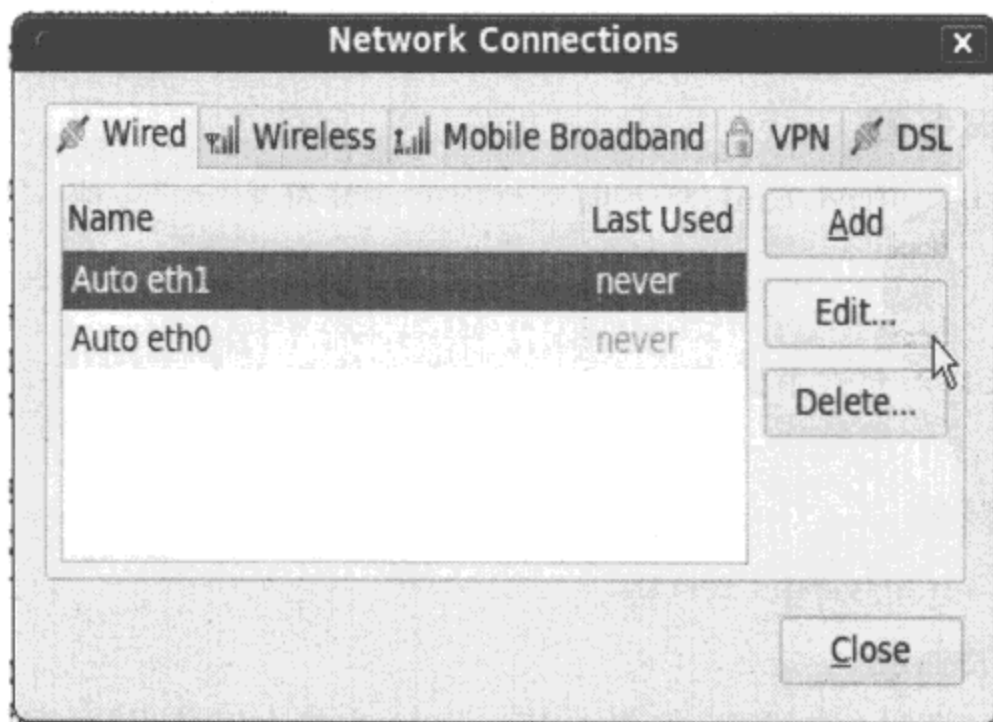


图4-32 网卡基本情况列表



图 4-33 设置 eth1 的 IP 地址

4.4.2 安装、配置和使用 FTP 服务

执行下面操作之前,再次使 Windows 主机能联通互联网。

1. 安装 FTP 服务器 vsftpd

```
~$ sudo apt-get install vsftpd
```

2. 配置 FTP 服务器

修改配置文件/etc/vsftpd.conf,将下面两行前的注释符号“#”去掉

```
# local_enable=YES
```

```
# write_enable=YES
```

上面第一行表示是否允许本地用户(dennis 就是本地用户之一)登录,第二行表示是否允许上传文件。

3. 重启 FTP 服务

```
~$ sudo /etc/init.d/vsftpd restart
```

4. 使用 FTP 服务器

从 FTP 服务器下载和上传数据,需要在 Windows 主机上安装 FTP 客户端。可以使用任何熟悉的 FTP 客户端,读者可上网搜索一个 FTP 客户端软件(uestudio.rar),下载使用。

4.4.3 安装、配置 NFS 服务

1. 安装 NFS 服务器

~\$ sudo apt-get install nfs-kernel-server portmap

2. 配置 NFS 服务器

修改配置文件/etc/exports,添加如下内容,以后就可以通过 NFS 文件系统访问/work 目录了

/work * (rw, sync, no_root_squash)

3. 重启 NFS 服务器

~\$ sudo /etc/init.d/nfs-kernel-server restart



第 5 章

建构 BootLoader

5.1 准备工作

5.1.1 嵌入式 Linux 系统概述

嵌入式系统是以应用为中心,以计算机技术为基础,且软硬件可裁减,适应应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。

以上定义是官方定义。其实嵌入式系统本质上必然是计算机系统,不严格地讲,当看到可以干活(运行程序)的东西,但它又没有显示器或者显示器很小的时候,大体上见到的就是嵌入式系统。不过嵌入式有别于通用的 PC,主要体现在功耗要小(想想如果一个手机的电池只能支撑 2 个小时,谁会用它),体积要小(砖头式的 MP3 你会用吗),成本要低(S3C2440 才卖 40 元人民币),可靠性要高(中国移动在山顶上的基站如果几天就坏一次的话,恐怕王同学和李同学早就下课了),硬件可裁减(如果只是做 MP3,会让硬件有网卡、显卡吗),软件可裁减(如果只做 MP3,会让 Linux 含有摄像头驱动吗?你会让系统含有 PowerPoint 吗)。

嵌入式系统最初的应用是基于单片机的,嵌入式系统的历史可以追溯到 20 世纪 70 年代,当时 Intel 公司生产了全球第一款单片机(my god, Intel 竟然是单片机的鼻祖)。20 世纪 80 年代嵌入式操作系统出现,最著名的是 Wind River 公司荣誉出品的 VxWorks。商业嵌入式实时内核包含传统操作系统的特征,使得开发周期缩短、成本降低、效率提高,促使嵌入式系统有了更为广阔的应用空间。20 世纪 90 年代, Linux、 μ C/OS 等可用于嵌入式的操作系统横空出世。进入 21 世纪, Windows CE、Symbian、Blackberry、iPhone、Android 也跑来凑热闹,好不快活。值得一提的是我的母校也弄出了个完整的商用嵌入式操作系统 DeltaOS,不过主要是军用,民用还没看到。

如图 5-1 所示,嵌入式系统的构架从下往上,一般分为硬件、驱动程序、操作系统、API 接口、应用程序。

对硬件来说,最重要的是 CPU,目前在嵌入式设备上使用最多的通用 CPU 是 ARM、MIPS、PowerPC、M68k 等。同时,带 DSP 功能的 CPU 也在逐渐兴起。存储系统除了常见的

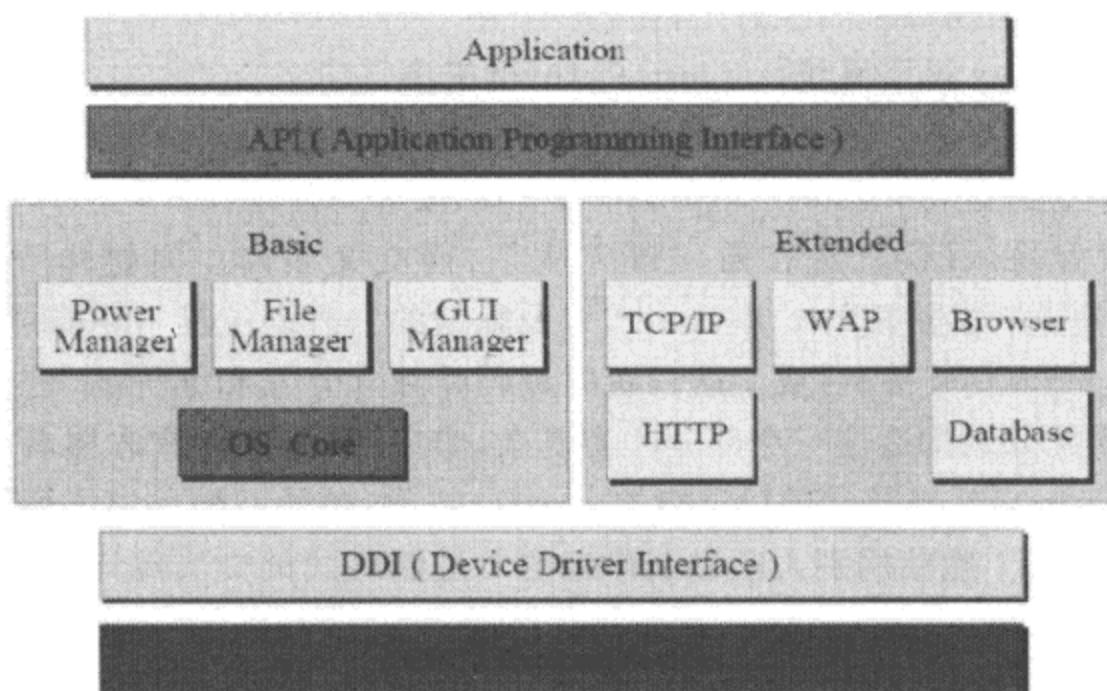


图 5-1 嵌入式系统构架

SDRAM 外,通常还使用 Nor flash 充当 ROM 以存放启动程序(BootLoader),Nand flash 充当硬盘。通信接口一般有 RS-232 接口(串口 UART)、USB 接口(通用串行总线接口)、IrDA (Infra Red Data Association,红外线接口)、SPI(串行外围设备接口)、I2C、CAN 总线接口、蓝牙接口(Bluetooth)、Ethernet(以太网接口)、IEEE1394 接口、通用可编程接口 GPIO、LCD 和触摸屏、扩展卡(各种 CF 卡、SD 卡、Memory Stick 等)。在便携式嵌入式系统的应用中,还必须特别关注电源装置等辅助设备。

对于嵌入式 Linux 系统而言,其软件系统的组成主要有:

- BootLoader;vivi、u-boot。
- 内核(kernel)。
- 根文件系统(yaffs、jffs2、cramfs、ramdisk...)。
- 系统应用程序(web server...)。
- 图形界面系统(QTE、MINIGUI...)。

开发工具,主要是:

- 交叉编译工具链。
- 图形界面开发工具。

5.1.2 构建交叉编译工具链

嵌入式系统的开发过程有别与普通的 PC 程序的开发,主要表现在程序的编辑、编译与程序的运行不在同一台设备上,通常前者在开发机(PC)上,后者在嵌入式设备(ARM 开发板)上。这就要求在进行实际开发前必须要搭建基于 ARM 的嵌入式 Linux 交叉开发环境,这

个过程主要包含以下几方面:

- 开发主机 Linux 操作系统(ubuntu 9.10)的安装。
- 开发主机基本服务(NFS、vsftpd)及程序(minicom)的安装、配置与使用。
- 交叉编译工具链(binutils\gcc\glibc)的生成、安装与使用。

前两者比较容易,我们已经在前一章完成了,而交叉编译工具链由交叉编译器(arm-linux-gcc)、供 ARM 平台使用的 C 库(glibc)以及辅助的二进制工具 binutils(包括 arm-linux-readelf、arm-linux-objcopy 等)组成。这三者的源码是独立开发和维护的,它们之间的版本兼容性较为繁杂,因此分别构造这三个部分较为复杂,难以成功,好在有志愿者为我们制作好了工具软件,例如 crosstool 以及它的后继版本 crosstool-ng,使用它们可以方便快捷地制作出交叉编译工具链。或者干脆使用别人已经制作好的二进制格式的交叉编译工具链,这样就只需要简单地解包安装即可,就像前一章那样操作。

为什么要制作交叉编译工具链呢?

因为程序的编译是在 PC 上完成的,如果使用 gcc 来编译程序,得到的二进制程序是 for X86 的,而不是 for ARM 的,它根本就不能运行在 ARM 平台上,因此需要交叉编译器 arm-linux-gcc,它编译出来的二进制程序是 for ARM 的,而不是 for X86 的。arm-linux-gcc 真是很有特点,它自己本身运行在 X86 平台上,但它编译出来的程序则运行在 ARM 平台上,所以称它为杂种编译器。光有 arm-linux-gcc 还是不够的,因为应用程序中会调用 printf 之类的库函数,编译器在编译时需要去库中链接这些库函数代码,由于最终程序是运行在 ARM 平台上的,所以被链接的库也必须是 for ARM 的。因此除了制作 arm-linux-gcc 外,还必须制作 for ARM 的 C 库(glibc)。

最后看一看光盘中提供的交叉编译工具链的 3 个组成部分在硬盘上的位置。交叉编译工具链安装于/usr/local/arm/gcc-3.4.5-glibc-2.3.6/目录:

- 交叉编译器位于 bin 目录,主要有:C 编译器 arm-linux-gcc、C++ 编译器 arm-linux-g++、汇编器 arm-linux-as、链接器 arm-linux-ld。
- 交叉编译用到的 C 库位于 arm-linux/lib;头文件位于 arm-linux/include。
- binutil 工具位于 bin 目录,主要有 arm-linux-readelf、arm-linux-ar、arm-linux-ranlib、arm-linux-objcopy 等。

5.1.3 BootLoader 概述

引导加载程序是系统加电后运行的第一段软件代码。回忆一下 PC 的体系结构可以知道,PC 中的引导加载程序由 BIOS(其本质就是一段固件程序)和位于硬盘 MBR 中的 OS BootLoader(比如,LILO 和 GRUB 等)一起组成。BIOS 在完成硬件检测和资源分配后,将硬盘 MBR 中的 BootLoader 读入到系统的 RAM 中,然后将控制权交给 OS BootLoader。BootLoader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中,然后跳转到内核的入口点去

运行,即开始启动操作系统。

而在嵌入式系统中,通常并没有像 BIOS 那样的固件程序(注,有的嵌入式 CPU 也会内嵌一段短小的启动程序),因此整个系统的加载启动任务就完全由 BootLoader 来完成。比如在一个基于 ARM9 核的嵌入式系统中,系统在上电或复位时通常都从地址 0x00000000 处开始执行,而在这个地址处安排的通常就是系统的 BootLoader 程序。

下面从 BootLoader 的概念、BootLoader 的主要任务、BootLoader 的框架结构来讨论嵌入式系统的 BootLoader。

1. BootLoader 的概念

简单地说,BootLoader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序,可以初始化硬件设备、建立内存空间的映射图,从而将系统的软硬件环境带到一个合适的状态,以便为最终调用操作系统内核准备好正确的环境。

通常,BootLoader 是严重地依赖于硬件而实现的,特别是在嵌入式世界。因此,在嵌入式世界里建立一个通用的 BootLoader 几乎是不可能的。尽管如此,仍然可以对 BootLoader 归纳出一些通用的概念来,以指导用户设计与实现特定的 BootLoader。

(1) BootLoader 所支持的 CPU 和嵌入式板。每种不同的 CPU 体系结构都有不同的 BootLoader。有些 BootLoader 也支持多种体系结构的 CPU,比如 u-boot 就同时支持 ARM 体系结构和 MIPS 体系结构。除了依赖于 CPU 的体系结构外,BootLoader 实际上也依赖于具体的嵌入式板级设备的配置。这也就是说,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,要想让运行在一块板子上的 BootLoader 程序也能运行在另一块板子上,通常也都需要修改 BootLoader 的源程序。

(2) BootLoader 的安装媒介(Installation Medium)。系统加电或复位后,所有的 CPU 通常都从某个由 CPU 制造商预先安排的地址上取指令。比如,基于 ARM 核的 CPU 在复位时通常都从地址 0x00000000 取它的第一条指令。而基于 CPU 构建的嵌入式系统通常都有某种类型的固态存储设备(比如 ROM、EEPROM 或 FLASH 等)被映射到这个预先安排的地址上。因此在系统加电后,CPU 将首先执行 BootLoader 程序。

(3) 用来控制 BootLoader 的设备或机制。主机和目标机之间一般通过串口建立连接,BootLoader 软件在执行时通常会通过串口来进行 I/O,比如:输出打印信息到串口、从串口读取用户控制字符等。

(4) BootLoader 的启动过程是单阶段(Single Stage)还是多阶段(Multi-Stage)。通常多阶段的 BootLoader 能提供更为复杂的功能,以及更好的可移植性。从固态存储设备上启动的 BootLoader 大多都是两阶段的启动过程,也即启动过程可以分为 stage 1 和 stage 2 两部分。而至于在 stage 1 和 stage 2 具体完成哪些任务将在下面讨论。

(5) BootLoader 的操作模式(Operation Mode)。大多数 BootLoader 都包含两种不同的操作模式:“启动加载”模式和“下载”模式,这种区别仅对于开发人员才有意义。但从最终用户

第5章 建构 BootLoader

的角度看, BootLoader 的作用就是用来加载操作系统, 而并不存在所谓的启动加载模式与下载工作模式的区别。

启动加载(BootLoading)模式:这种模式也称为“自主”(Autonomous)模式。也即 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。这种模式是 BootLoader 的正常工作模式, 因此在嵌入式产品发布的时候, BootLoader 显然必须工作在这种模式下。

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

像 Blob 或 u-boot 等这样功能强大的 BootLoader 通常同时支持这两种工作模式, 而且允许用户在这两种工作模式之间进行切换。比如, u-boot 在启动时处于正常的启动加载模式, 但是它会延时 10 s 等待终端用户按下任意键而将 u-boot 切换到下载模式。如果在 10 s 内没有用户按键, 则 u-boot 继续启动 Linux 内核。

(6) BootLoader 与主机之间进行文件传输所用的通信设备及协议。最常见的情况就是, 目标机上的 BootLoader 通过串口与主机之间进行文件传输, 传输协议通常是 xmodem/ymodem/zmodem 协议中的一种。但是, 串口传输的速度是有限的, 因此通过以太网连接并借助 TFTP 协议(或者 NFS 协议)来下载文件是个更好的选择。

此外, 在论及这个话题时, 主机方所用的软件也要考虑。比如, 在通过以太网连接和 TFTP 协议(或者 NFS 协议)来下载文件时, 主机方必须有一个软件用来提供 TFTP(或者 NFS)服务。

2. BootLoader 的主要任务与典型结构框架

从操作系统的角度看, BootLoader 的总目标就是正确地调用内核来执行。

另外, 由于 BootLoader 的实现依赖于 CPU 的体系结构, 因此大多数 BootLoader 都分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码, 比如设备初始化代码等, 通常都放在 stage1 中, 而且通常都用汇编语言来实现, 以达到短小精悍的目的。而 stage2 则通常用 C 语言来实现, 这样可以实现更复杂的功能, 而且代码会具有更好的可读性和可移植性。

BootLoader 的 stage1 通常包括以下步骤(以执行的先后顺序):

- (1) 硬件设备初始化。
- (2) 为加载 BootLoader 的 stage2 准备 RAM 空间。
- (3) 复制 BootLoader 的 stage2 到 RAM 空间中。
- (4) 设置好堆栈。

(5) 跳转到 stage2 的 C 入口点。

BootLoader 的 stage2 通常包括以下步骤(以执行的先后顺序):

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

3. BootLoader 的 stage1 完成的主要任务

(1) 基本的硬件初始化。这是 BootLoader 一开始就执行的操作,其目的是为 stage2 的执行以及随后的 kernel 的执行准备好一些基本的硬件环境。它通常包括以下步骤(以执行的先后顺序):

- 为中断提供服务通常是 OS 设备驱动程序的责任,因此在 BootLoader 的执行全过程中可以不必响应任何中断。中断屏蔽可以通过写 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 的速度和时钟频率。
- RAM 初始化包括正确地设置系统的内存控制器的功能寄存器等。
- 关闭 CPU 内部指令/数据 Cache。

(2) 复制 stage2 到 RAM 中。

复制时要确定两点: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; RAM 空间的起始地址。

(3) 设置堆栈指针 sp。堆栈指针的设置是为了执行 C 语言代码作好准备。

(4) 跳转到 stage2 的 C 入口点。在上述一切都就绪后,就可以跳转到 BootLoader 的 stage2 去执行了。比如,在 ARM 系统中,这可以通过修改 PC 寄存器为合适的地址来实现。

4. BootLoader 的 stage2 完成的主要任务

stage2 的代码通常用 C 语言来实现,以便于实现更复杂的功能和取得更好的代码可读性和可移植性。但是与普通 C 语言应用程序不同的是,在编译和链接 BootLoader 这样的程序时,不能使用 glibc 库中的任何支持函数。其原因是显而易见的。

(1) 初始化本阶段要使用到的硬件设备。这通常包括:初始化至少一个串口,以便和终端用户进行 I/O 输出信息。设备初始化完成后,可以输出一些打印信息,程序名字字符串、版本号等。

(2) 加载内核映像。从 Nor Flash 或 Nand Flash(需要编写 Nand Flash 裸驱动)上将内核映像复制到 RAM 中。

(3) 设置内核的启动参数。应该说,在将内核映像复制到 RAM 空间中后,就可以准备启

动 Linux 内核了。但是在调用内核之前,应该做一步准备工作,即设置 Linux 内核的启动参数。

Linux 2.4.x 以后的内核都期望以标记列表(tagged list)的形式来传递启动参数。启动参数标记列表以标记 ATAG_CORE 开始,以标记 ATAG_NONE 结束。每个标记由标识被传递参数的 tag_header 结构以及随后的参数值数据结构来组成。数据结构 tag 和 tag_header 定义在 Linux 内核源码的 include/asm/setup.h 头文件中。在嵌入式 Linux 系统中,通常需由 BootLoader 设置的常见启动参数有 ATAG_CORE、ATAG_MEM、ATAG_CMDLINE。

(4) 调用内核。BootLoader 调用 Linux 内核的方法是直接跳转到内核的第一条指令处,也即直接跳转到 MEM_START+0x8000 地址处。在跳转时,下列条件要满足:

- CPU 寄存器的设置:

R0=0;

R1=机器类型 ID;关于 Machine Type Number,可以参见 linux/arch/arm/tools/machine-types。

R2=启动参数标记列表在 RAM 中起始基地址;

- CPU 模式:

必须禁止中断(IRQs 和 FIQs);

CPU 必须处于 SVC 模式;

- Cache 和 MMU 的设置:

MMU 必须关闭;

指令 Cache 可以打开也可以关闭;

数据 Cache 必须关闭;

如果用 C 语言,可以像下列示例代码这样来调用内核:

```
void (* theKernel)(int zero, int arch, u32 params_addr) = (void (*)(int, int, u32))KERNEL_RAM_
BASE;
```

```
.....
```

```
theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

注:到此为止,绝大多数内容均来源于詹荣开所著《嵌入式系统 BootLoader 技术内幕》,只做了少量修改,可视为对该文的转载。

5. 常见的嵌入式 BootLoader

表 5-1 显示的是几种不同的 BootLoader 以及特性。

表 5-1 各种 BootLoader

| BootLoader | Monitor | 描 述 | x86 | ARM | PowerPC |
|------------|---------|--------------------------|-----|-----|---------|
| LILO | 否 | Linux 磁盘引导程序 | 是 | 否 | 否 |
| GRUB | 否 | GNU 的 LILO 替代程序 | 是 | 否 | 否 |
| Loadlin | 否 | 从 DOS 引导 Linux | 是 | 否 | 否 |
| ROLO | 否 | 从 ROM 引导 Linux 而不需要 BIOS | 是 | 否 | 否 |
| Etherboot | 否 | 通过以太网卡启动 Linux 系统的固件 | 是 | 否 | 否 |
| LinuxBIOS | 否 | 完全替代 BIOS 的 Linux 引导程序 | 是 | 否 | 否 |
| BLOB | 否 | LART 等硬件平台的引导程序 | 否 | 是 | 否 |
| u-boot | 是 | 通用引导程序 | 是 | 是 | 是 |
| RedBoot | 是 | 基于 eCos 的引导程序 | 是 | 是 | 是 |

(1) X86 的工作站和服务器上一般使用 LILO 和 GRUB。

(2) ARM 处理器的芯片商很多,所以每种芯片的开发板都有自己的 BootLoader。结果 ARM BootLoader 也变得多种多样。早期有为 ARM720 处理器开发板编写的固件,后来有了 armboot,以及 StrongARM 平台的 blob,还有 S3C2410 处理器开发板上的 vivi 等。现在 armboot 已经并入了 u-boot,所以 u-boot 也支持 ARM/XScale 平台。u-boot 已经成为 ARM 平台事实上的标准 BootLoader。

(3) PowerPC 平台的处理器有标准的 BootLoader,就是 ppcboot。ppcboot 在合并 armboot 等之后,创建了 u-boot,成为各种体系结构开发板的通用引导程序。u-boot 仍然是 PowerPC 平台的主要 BootLoader。

(4) MIPS 公司开发的 YAMON 是标准的 BootLoader,也有许多 MIPS 芯片商为自己的开发板写了 BootLoader。现在,u-boot 也已经支持 MIPS 平台。

(5) SH 平台的标准 BootLoader 是 sh-boot。Redboot 在这种平台上也很好用。

(6) M68K 平台没有标准的 BootLoader。Redboot 和 u-boot 能够支持 M68K 系列的系统。

5.2 深入剖析 u-boot 代码

本节将以 S3C2440 开发板作为硬件实验平台,ubuntu 9.10 作为开发机,u-boot-1.1.6 作为 BootLoader(位于光盘\work\sysbuild\u-boot-1.1.6.tar.bz2),Linux-2.6.22.6 作为内核,gcc-3.4.5 和 glibc-2.3.6 作为交叉编译工具,来深入剖析 u-boot。

5.2.1 安装和使用源代码阅读工具 Source Insight

如果没有强大的源码阅读工具,阅读庞大的系统软件代码无疑是一场梦魇。由于即将开始阅读 u-boot 源代码之旅,所以必须先学习使用源码阅读工具。在 Linux 下最著名的是 Kscope,而在 Windows 下则是的 Source Insight。下面就来学习 Source Insight 的使用。

1. 安装 Source Insight

从网址 <http://www.sourceinsight.com> 上下载一个试用版,并进行安装。

2. 增加对分析汇编源文件的支持

启动 Source Insight 之后,它默认的支持文件中没有以“.S”结尾的汇编文件。单击菜单 Options→Document Options,在弹出对话框中选择 Document Type 为 C Source File,在 File filter 中添加“*.S”和“*.s”类型,如图 5-2 所示。务必注意:*.h 和 *.S 和 *.s 之间的分隔符是英文的分号。

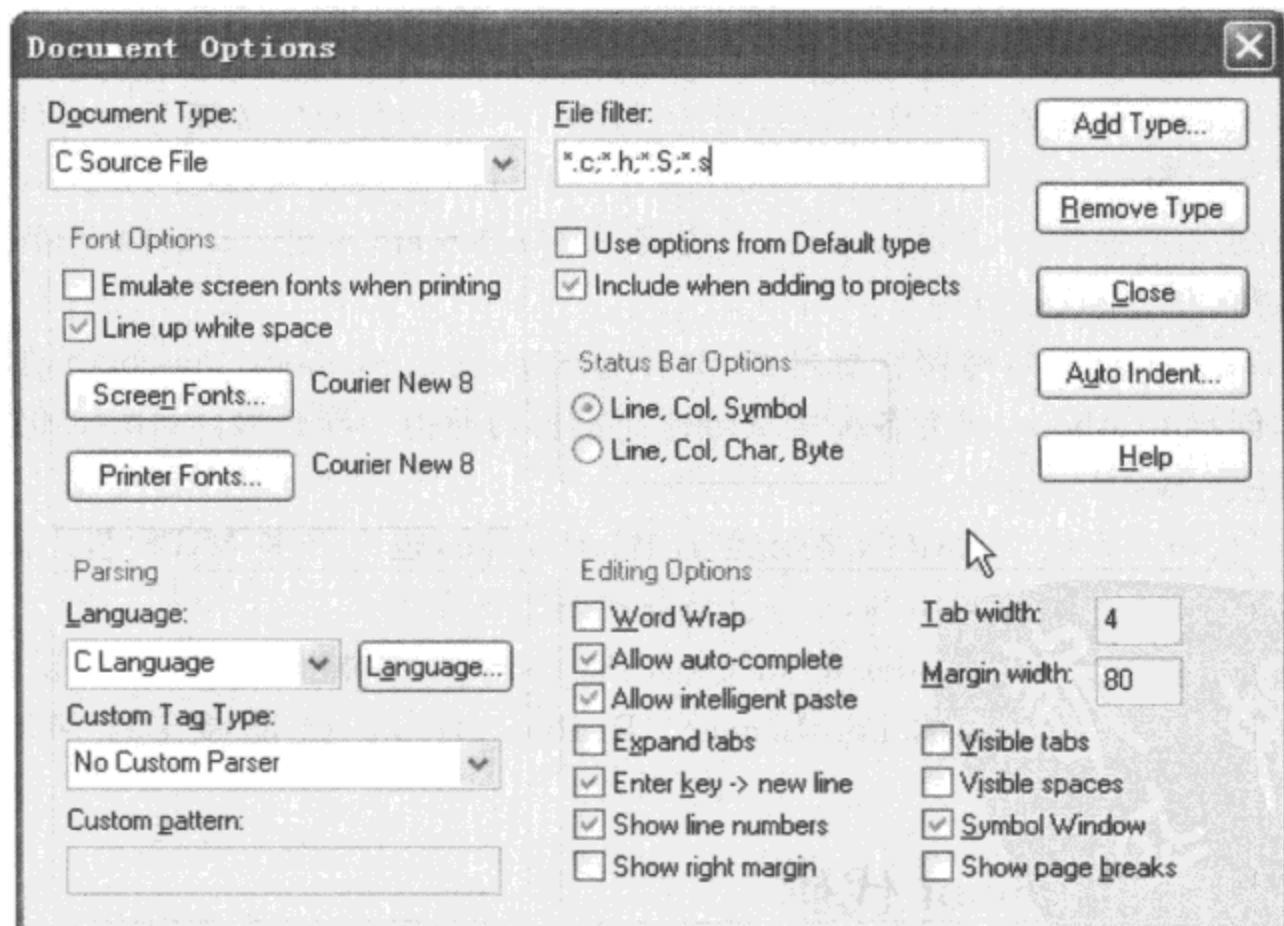


图 5-2 添加对.s 文件的支持

3. 快速创建 u-boot-1.1.6 的 source insight 工程(图 5-3)

单击菜单 Project→New Project,开始新建一个新的工程,将 u-boot-1.1.6 的源代码加入该工程。在图 5-3 中选中“u-boot-1.1.6”源码目录后,单击 Add Tree 按钮将 u-boot 全部的

源代码加入该工程。

特别说明:其实不必将所有源码文件都加入工程,只需要将与使用的开发板相关的源文件加入即可。

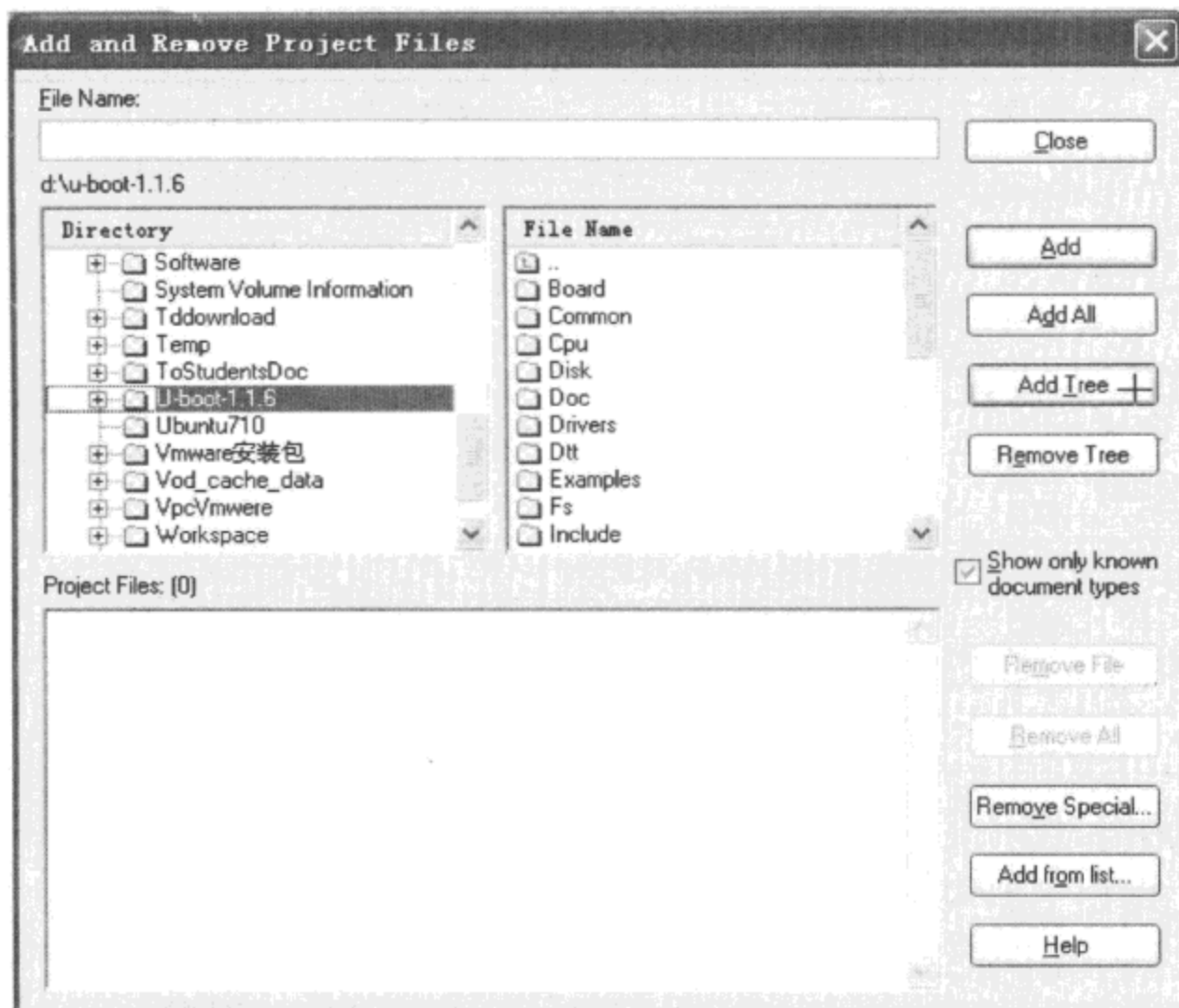


图 5-3 新建 Source Insight 工程

4. 同步源文件

单击 Project→Synchronize Files,会弹出图 5-4 所示对话框,选中其中的 Force all files to be re-parsed,然后单击 OK 按钮即可生成保存源文件中各变量、函数之间关系的数据库。

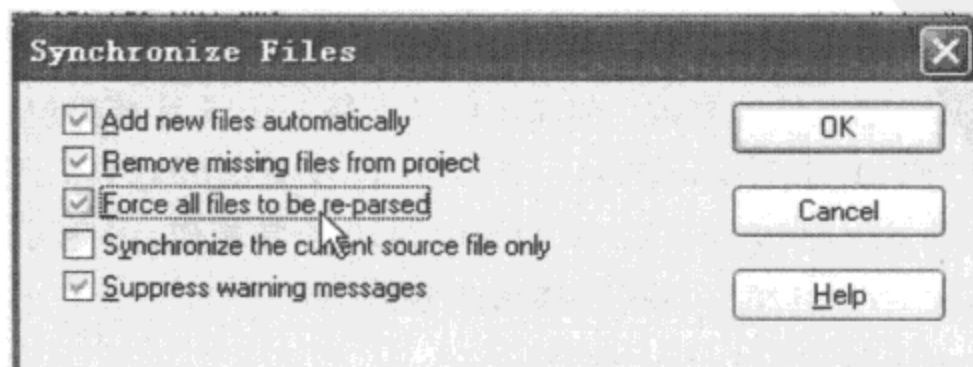


图 5-4 同步文件

5.2.2 u-boot 的编译初步

表 5-2 是 u-boot 源代码的目录结构。

表 5-2 u-boot 源代码目录结构

| 目 录 | 特 性 | 解释说明 |
|-------------|------|---|
| board | 平台依赖 | 存放电路板相关的目录文件,例如:RPXlite(mpc8xx)、smdk2410(arm920t)、sc520_cdp(x86)等目录 |
| cpu | 平台依赖 | 存放 CPU 相关的目录文件,例如 mpc8xx、ppc4xx、arm720t、arm920t、xscale、i386 等目录 |
| lib_arm | 平台依赖 | 存放对 ARM 体系结构通用的文件,主要用于实现 ARM 平台通用的函数 |
| include | 通用 | 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 |
| common | 通用 | 通用的多功能函数实现,即:u-boot 的命令 |
| lib_generic | 通用 | 通用库函数的实现,如 printf |
| Net | 通用 | 存放网络的程序 |
| Fs | 通用 | 存放文件系统的程序 |
| Post | 通用 | 存放上电自检程序 |
| drivers | 通用 | 通用的设备驱动程序,主要有以太网接口的驱动以及 Nand Flash 驱动 |
| Disk | 通用 | 硬盘接口程序 |
| Rtc | 通用 | RTC 的驱动程序 |
| Dtt | 通用 | 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 helloworld |
| tools | 工具 | 存放制作 S-Record 或者 u-boot 格式的映像等工具,例如 mkimage |
| Doc | 文档 | 开发使用文档 |

使用 tar 命令解压源代码包后,cd /work/sysbuild/u-boot-1.1.6。

u-boot-1.1.6 的代码,其目录结构主要分为与体系结构有关的代码目录以及与体系结构无关(通用)的代码目录。前者主要包含 board 目录和 CPU 目录,移植 u-boot 的工作主要就集中在对这些目录里面特定文件的修改。board 目录下每一个子目录都包含一个 u-boot 支持的硬件开发板的支持代码,其中有 smdk2410 子目录,但没有 smdk2440 目录。这表明 u-boot 支持 S3C2410,但不支持 S3C2440。S3C2440 与 S3C2410 非常相似,因此可以用 S3C2410 的支持代码作为基础,为 S3C2440 移植 u-boot。移植步骤如下:

- (1) cp -R board/smdk2410/ board/my2440
- (2) mv board/my2440/smdk2410.c board/my2440/my2440.c

(3) 将 board/my2440/Makefile 的第 28 行 COBJS := smdk2410.o flash.o 改为 COBJS := my2440.o flash.o (注:关于 Makefile 的详细写法,请参阅技术文档“GNU make 中文手册”或者“GNU make 英文手册”)。

(4) cp include/configs/smdk2410.h include/configs/my2440.h。

(5) 在 Makefile 的第 1882 行处模仿 smdk2410_config 目标增加新目标 my2440_config (共两行)。

```
my2440_config : unconfig
```

```
@ $(MKCONFIG) $(@:_config=) arm arm920t my2440 NULL
s3c24x0
```

(6) make my2440_config。

(7) make。

最终得到 u-boot.bin,但该程序是 for S3C2410 的,不能运行在 S3C2440 上。这就需要修改源代码,但应该修改哪里呢?这就必须阅读源代码,但从哪个文件开始阅读呢?当然是程序的第一条指令所在文件,但这个文件是谁呢?查看 make 过程最后的链接命令:

```
arm-linux-ld -Bstatic -T /work/system/u-boot-1.1.6/board/my2440/u-boot.lds -Ttext 0x33F80000?
$ UNDEF_SYM cpu/arm920t/start.o \
    --start-group lib_generic/libgeneric.a board/my2440/libmy2440.a cpu/arm920t/li-
barm920t.a cpu/arm920t/s3c24x0/lib_s3c24x0.a lib_arm/libarm.a fs/cramfs/libcramfs.a fs/fat/libfat.
a fs/fdos/libfdos.a fs/jffs2/libjffs2.a fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a net/libnet.a
disk/libdisk.a rtc/librtc.a dtb/libdtb.a drivers/libdrivers.a drivers/nand/libnand.a drivers/nand_
legacy/libnand_legacy.a drivers/sk98lin/libsk98lin.a post/libpost.a post/cpu/libcpu.a common/lib-
common.a --end-group -L /work/tools/gcc-3.4.5-glibc-2.3.6/lib/gcc/arm-linux/3.4.5 -lgcc \
    -Map u-boot.map -o u-boot
```

其中的-T /work/system/u-boot-1.1.6/board/my2440/u-boot.lds 可知链接脚本是 /work/system/u-boot-1.1.6/board/my2440/u-boot.lds。查看该链接脚本。

```
SECTIONS
```

```
{
. = 0x00000000;
. = ALIGN(4);
.text :
{
cpu/arm920t/start.o (.text)
* (.text)
}
```

可知程序第一条指令所在源代码文件是 cpu/arm920t/start.S。

资源解密

PDG

注:关于 ld 的用法以及链接脚本的详细写法请参阅技术文档“GNU ld 英文手册”。

5.2.3 分析 u-boot 的第一阶段代码(cpu/arm920t/start.S)

注:start.S 中有很多 GNU 语法的 ARM 汇编伪操作,这些伪操作的含义可查阅“GNU Assembler 英文手册-简单版”。GNU 的 ARM 指令和伪指令与 ADS 的完全一样。

(1) 第一条指令在 42 行(reset 异常向量),是一条 b 指令,跳转到 114 行。

附带说明:42~49 为异常向量区,51~57 为 literal pool,供异常向量的跳转指令使用。57~102 定义了若干全局变量供后续代码使用,其中 76 行的 TEXT_BASE 由 gcc 编译时传入 (arm-linux-gcc -D__ASSEMBLY__ -g -Os -fno-strict-aliasing -fno-common -ffixed-r8 -msoft-float -malignment-traps -D__KERNEL__ -DTEXT_BASE=0x33F80000? -I/work/system/u-boot-1.1.6/include -fno-builtin -ffreestanding -nostdinc -isystem /work/tools/gcc-3.4.5-glibc-2.3.6/lib/gcc/arm-linux/3.4.5/include -pipe -DCONFIG_ARM -D__ARM__ -march=armv4 -mapcs-32 -c -o start.o start.S),由-DTEXT_BASE=0x33F80000 可知,TEXT_BASE 的值为 0x33F80000。

(2) 114~117 切换模式到管理模式。由于 ARM 上电就处于管理模式,故此代码可以省略。

(3) 125~128,132~145 关闭 Watch Dog,屏蔽所有中断。150~152 设置频率,此段代码针对 2410 是对的,对 2440 来说是错误的,但由于后续还有代码进行正确的设置,所以可以不理睬这段错误代码。

(4) 160 行调用子程序 cpu_init_crit 完成清除 CPU cache(245~247),禁用 MMU(252~257),初始化内存控制器(265)。

(5) 初始化内存控制器(board/my2440/lowlevel_init.S)。

① 请思考 139 行为何要让 r0 减去 r1 的值?

回答:

137~149 行的目的是将 155~167 行存储的数赋给内存控制器的 13 个寄存器,因此必须要正确寻址 155~167 行存储的数据在内存中的位置。因为整个 u-boot.bin 是位于 Nor Flash 中的,所以 155 行存储的数据在内存中的地址为 0x100(假设 155 行存储的数据在 u-boot.bin 二进制文件中的位置为 0x100)。我们需要由 r0 来正确定位 155 行存储的数据的地址(即:0x100),而 ldr r0, =SMRDATA 这条伪指令取的地址是标号的绝对地址(也就是程序的运行地址加上 _SMRDATA 标号与程序第一条指令的偏移量),这将导致 137 行执行后 r0 = 0x33f80100 + 0x100,因此我们需要 139 行来将 r0 的值从 0x33f80100 变为 0x100(138 行执行后,r1=0x33f80000)。

事实上 137~139 行可以由 `adr r0, SMRDATA` 代替,而且这种替换还更好,因为这样一来就不需要 `start.S` 的第 159 行的 `CONFIG_SKIP_LOWLEVEL_INIT` 条件编译了。

② 请务必将 126 行 `#define REFCNT 1113` 改为 `#define REFCNT 1268`

思考为何要做这样的修改?

回答:

图 5-5 是 SDRAM 的一个 bit,存储电容上有电荷表示 1,无电荷表示 0。但由于存储电容有自动放电的物理属性,因此要维持 SDRAM 正确运行就必须定期给电容充电,这个周期性的充电操作称为内存刷新,它是由内存控制器自动完成的。因此内存控制器必须知道刷新周期(即:多长时间刷新一次),刷新周期最终取决于 SDRAM 芯片的物理特性,通过查阅现代公司提供的 SDRAM 的芯片手册可知:64ms 内最少需要刷新 8192 次。再根据 samsung 提供的 S3C2440 硬件手册关于内存控制器的说明,如表 5-3 所列。

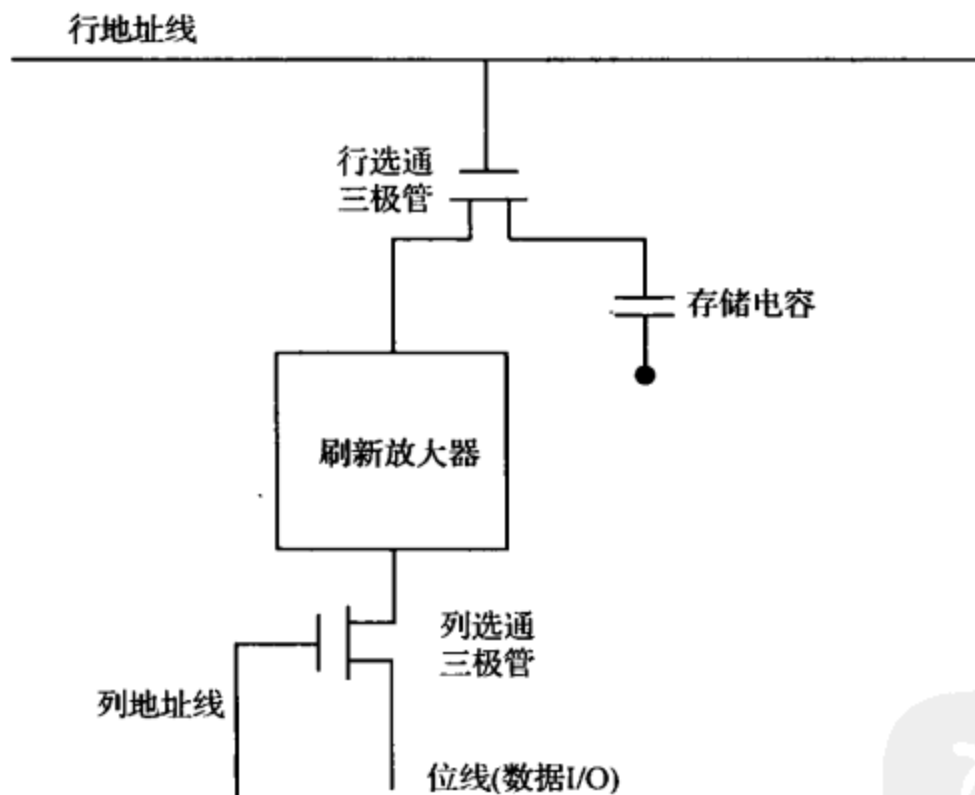


图 5-5 SDRAM 硬件原理图

表 5-3 内存控制器 REFRESH 寄存器

| 寄存器名 | 地 址 | R/W | 描 述 | 复位默认值 |
|---------|------------|-----|--------------------------------|----------|
| REFRESH | 0x48000024 | R/W | SDRAM refresh control register | 0xac0000 |

续表 5-3

| REFRESH | 位 | 描 述 | 初始值 |
|-----------------|----------|--|-----|
| Refresh Counter | [10 : 0] | SDRAM refresh count value. Refer to chapter 6 SDRAM refresh controller bus priority section. Refresh period = $(2^{11} - \text{refresh_count} + 1) / \text{HCLK}$ Ex) if refresh period is 7.8 us and HCLK is 100 MHz, the refresh count is as follows. Refresh count = $2^{11} + 1 - 100 \times 7.8 = 1269$ | 0 |

可知地址为 0x48000024 的寄存器的低 11bit 应该写入值 X。X 需要满足： $64\text{ms}/8192 = (2^{11} - X + 1) / 100\text{M}$ 。由此可以计算出 $X = 1268$ 。

(6) 165~179 将 Nor Flash 中的 u-boot 代码段(text 段)以及已初始化数据段(data 段)复制到 RAM 中(0x33f80000~?)。

思考:165 行后 r0 的值是多少? 为何有 167、168 行,可不可以删掉它们,为什么? 为何要将 u-boot 的 text 和 data 段从 Nor Flash 复制到 RAM 中?

回答:

“adr r0, _start”这条伪指令取的地址是相对于当前 PC 进行计算的地址。当程序在 Nor Flash 里运行(这正是当前的情况)时,假如 adr 这条指令相对于整个二进制的程序的第一条指令的偏移量是 0x100,则 _start 标号所代表的地址相对于 adr 伪指令所在地址的偏移量必然为 -0x100。所以编译器编译时就会把 adr 这条伪指令编译为 add r0, pc, #-0x100(忽略流水线对 pc 的影响),程序运行到 add r0, pc, #-0x100 这条指令时,pc 的值为 0x100(忽略流水线对 pc 的影响),所以该指令执行后 r0 的值为 0。

167、168 行必须有。因为 170~179 是将 Nor Flash 中的 u-boot 代码段(text 段)以及已初始化数据段(data 段)复制到 RAM 中。当 u-boot. bin 是被烧写到 Nor Flash 的情况下, $r0=0, r1=0x33f80000$, 167~168 行执行的结果将会使 170~179 行被执行,这是我们期望的结果。在这种情况下,不要 167~168 行,结果也一样。但是如果 u-boot. bin 不是烧写到 Nor Flash 中,而是由下载器直接下载到 RAM 的 0x33f80000 处(这种情况出现在 u-boot 的开发调试过程中)运行,此时 $r0=0x33f80000, r1=0x33f80000$ 。这种情况下,如果不要 167 和 168 行将会导致 170~179 行执行,而此时 Nor Flash 中的内容(也就是将被复制到 RAM 0x33f80000 处的内容)不是 u-boot. bin 的二进制代码,而是随机的乱码。这样一来,在复制后,原先位于 RAM 0x33f80000 处正确的程序代码就被改变了,程序将不会再正确运行。反过来看,如果有 167、168 行,则 170~179 行不会被执行,程序就能继续正确运行。

.data 段要复制的原因是:由于 CPU 上电时 $PC=0$,这使得我们必须要将 u-boot. bin 放在 Nor Flash 中(当然这就导致 .data 段也放在 Nor Flash 中)。而 .data 段包含的是已初始化全局变量,这些变量的值很有可能在程序中会被改变,这就要求这些变量的存储位置必须在可

以读写的 RAM 中,而不能在只读的 Nor Flash 中。所以必须将位于 Nor Flash 中的 .data 段从 Nor Flash 复制到 RAM 中(为了保证程序运行后能正确地访问到位于 RAM 中的已初始化全局变量,还必须将程序的运行地址设置为 0x33f80000)。

.text 段要复制的原因是:从理论上讲,只要把 .text 段的运行地址设为 0x0,就不需要复制 .text 段。但由于代码在 RAM 中的运行速度要远快于在 Nor Flash 中的运行速度,所以从效率的角度考虑,要把 .text 段从 Nor Flash 复制到 RAM 中(同样,为了保证程序运行后能正确地跳转,还必须将程序的运行地址设置为 0x33f80000)。

(7) 初始化栈指针 sp (184 ~ 190)。设置栈指针 sp 为 0x33f4df74 (启用中断)或 0x33f4ff74 (不启用中断)。

思考:为何要设置 SP 的初值? 为何 SP 的值要设置为比 0x33f80000 小?

回答:

因为将来 u-boot 还要运行 C 程序代码,这会导致出入栈操作。为了使出入栈操作正确完成,需要设置正确的栈顶位置(即:SP 的初值)。

如图 5-6 所示,因为满足 ATPCS 规则的 ARM 程序采用的是递减满堆栈,将 SP 的值设为小于 0x33f80000,可以防止入栈操作对存放在 0x33f80000 处的 u-boot 程序和数据造成破坏。并且在存放 u-boot 代码和数据的区域前,将来还会存放 u-boot 环境变量、堆区、全局信息,预留给 Abort-stack 的区域,所以 SP 的值要在以上那些区域的底部。

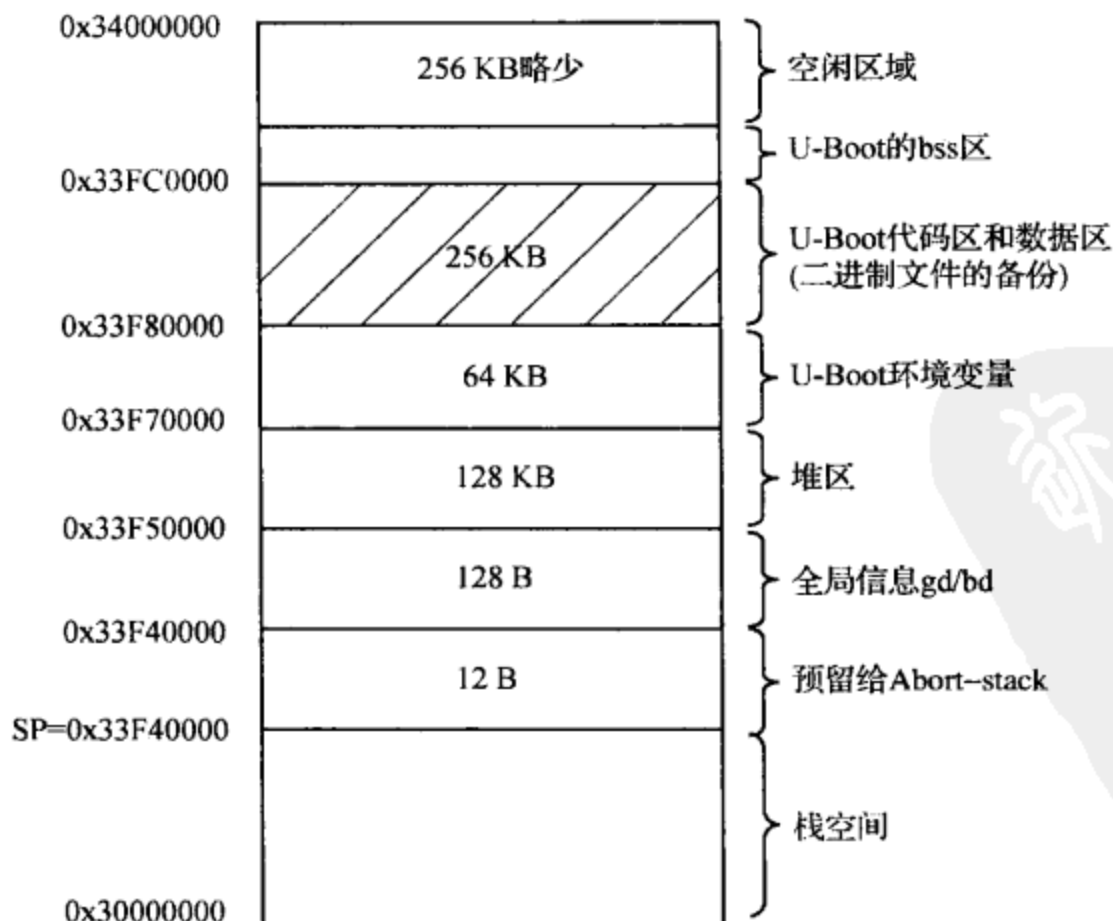


图 5-6 U-Boot 内存使用图

(8) 193~200 清零 RAM 中的未初始化数据段(bss 段)。

思考:data 段在 Nor Flash 和 RAM 中各有一份备份,为何 bss 段仅在 RAM 中有一份备份,而在 Nor Flash 中没有? 为何 bss 段要被清为 0,而不是置为 1?

回答:

bss 段是未初始化全局变量段,该段的变量的值是用户可以修改的,而 Nor Flash 是只读存储器,不能对其中的变量进行修改,因此 bss 段最终必须在 RAM 中。要想让 bss 段最终在 RAM 中,有两个办法。一种办法是像 data 段一样(即:先在 Nor Flash 中有一份备份,然后由程序将其复制到 RAM 中);另一种办法是在 Nor Flash 中根本就不放置 bss 段的备份,而是直接由程序在 RAM 中对 bss 段进行初始化。不采用第一种办法的原因是,未初始化全局变量的初值一定是 0,所以未初始化全局变量不必在编译时就为其在二进制文件中分配空间(这样做的好处是可以减小二进制文件的大小),这样一来,当将二进制文件烧写到 Nor Flash 中时,bss 段自然也就在 Nor Flash 中没有备份。但要注意,由于最终 bss 段必须出现在 RAM 中,所以要想办法让 bss 段的运行地址位于 RAM 中(这可以通过设置 text 段的运行地址位于 RAM 中,再由链接脚本文件指定 data 段位于 text 段后,bss 段位于 data 后来实现。链接脚本文件请参见 board/my2440 /u-boot.lds)。

C 语言标准规定,未初始化的全局变量的初值必须为 0,所以要将 bss 段清 0,而不是置 1。

(9) 223 行,神奇一跳到了 RAM 中,跳转到位于 RAM 中的第二阶段代码的 C 入口点(调用 lib_arm/board.c 中的 start_armboot 函数)。

思考:为何是跳到了 RAM 中,而不是仍在 Nor Flash 中? 为何要用 ldr 指令,而不是 b 指令?

回答:

_start_armboot 处存放的 start_armboot 是 lib_arm/board.c 文件中的函数 start_armboot 的名称,所以 ldr pc, _start_armboot 执行之后,PC 的值就是函数 start_armboot 在内存中的首地址。start_armboot 在内存中的首地址在编译时就被编译器确定。编译器在确定某个函数的内存地址时是根据函数相对于整个程序的偏移量+程序的运行地址(0x33f80000),因此 start_armboot 代表的地址就位于 RAM 中。

b 指令的实现机制是:将机器码的低 24 位作为偏移量,跳转到当前指令位置+偏移量,而 b 指令执行时程序还在 Nor Flash 中运行,这样一来 b 指令跳转后,程序仍然在 Nor Flash 中运行,并不能跳转到 RAM 中。

5.2.4 分析 u-boot 的第二阶段代码

u-boot 的第二阶段代码开始于 lib_arm/board.c 中的 start_armboot 函数。

(1) 216~234,258~262,执行系统初始化工作。

① 其中 board_init(board/my2440/my2440.c)函数完成了。

● 系统频率设定。

S3C2410 CPU 需要 4 个频率,分别是 CPU 核 arm920t 使用的 FCLK、快速设备控制器(例如内存控制器)使用的 HCLK、慢速设备控制器(例如串口控制器)使用的 PCLK、USB 控制器使用的 UPLL。这 4 个频率均是由晶振频率(12MHz)通过频率控制器芯片倍频和分频所得,它们分别为 200MHz、100MHz、50MHz、48MHz。由于 S3C2440 的频率分别是 400MHz、100MHz、50MHz、48MHz,并且 S3C2440 与 S3C2410 的频率控制器芯片的设定方法也略有不同,所以必须修改该段代码以匹配 s3c2440。

将 77 行改为:

```
#define S3C2440_MPLL_400MHz ((0x5c<<12)|(0x01<<4)|(0x01))
#define S3C2440_UPLL_48MHz ((0x38<<12)|(0x02<<4)|(0x02))
#define S3C2440_CLKDIV      0x05 /* FCLK;HCLK;PCLK = 1:2:4 */
/* FCLK;HCLK;PCLK = 1:4:8 */
clk_power->CLKDIVN = S3C2440_CLKDIV;
/* Change to Asynchronous bus mode */
__asm__( "mrc p15, 0, r1, c1, c0, 0\n" /* read ctrl register */
        "orr r1, r1, #0xc0000000\n" /* Asynchronous */
        "mcr p15, 0, r1, c1, c0, 0\n" /* write ctrl register */
        ::: "r1"
    );
/* configure MPLL */
clk_power->MPLLCON = S3C2440_MPLL_400MHz;
```

将 83 行改为:

```
clk_power->UPLLCON = S3C2440_UPLL_48MHz;
```

注:请查看“GCC 使用手册—英文”了解 GNU 内嵌汇编语法。

● 初始化 GPIO(89~103)。

● 设定机器类型 ID,将在调用 kernel 时传给 kernel。

需将 `gd->bd->bi_arch_number = MACH_TYPE_SMDK2410` 改为 `gd->bd->bi_arch_number = MACH_TYPE_S3C2440`,从匹配 s3c2410 变为匹配 s3c2440。

● 109 行,设定 kernel 启动参数在内存中的存放地址,将在调用 kernel 时传给 kernel。

② 其中 `serial_init` 调用 `serial_setbrg(cpu/arm920t/s3c24x0/serial.c)`。

初始化串口以便对外输出文本。该函数需要将串口比特率设置为 115200,因此需要获取 PCLK(58 行通过 `get_PCLK` 获得)。cpu/arm920t/s3c24x0/speed.c 中的 `get_HCLK`、`get_PCLK`、`get_PLLCLK` 函数获得 HCLK 和 PCLK,但由于 2410 与 2440 计算 PLLCLK 和 HCLK、PCLK 的方法不一样,需要修改这 3 个函数。因此需要将 cpu/arm920t/s3c24x0 下的 speed.c 的源代码全部重写。

第5章 建构 BootLoader

光盘\work\sysbuild\speed.c 文件含有重写后的代码,只需要简单将该文件拷贝覆盖 cpu/arm920t/s3c24x0 下的 speed.c 即可。

此外,由于 s3c2440 频率控制器比 s3c2410 的频率控制器多了一个寄存器 CAMDIVN,并且上述程序使用了该寄存器,因此还需要在 include/s3c24x0.h 中的结构体 S3C24X0_CLOCK_POWER 的最后增加一个字段 CAMDIVN(增加第 129 行:S3C24X0_REG32?? CAMDIVN;)

(2) 266 行初始化 Nor Flash(flash_init)。

(3) 299~302 初始化 Nand Flash(需自行实现与开发板相关的 board_nand_init 函数)。由于该段代码是条件编译,故需修改 include/configs/my2440.h 文件,将第 81 行的注释去掉。这样做同时也条件编译了 u-boot 的 nand 命令。

(4) 368 行初始化网卡 cs8900。

(5) 398 行调用 main_loop,进入 u-boot 的命令下载模式(或者在其中检查 bootdelay 和 bootcmd 环境参数,超时进入启动加载模式执行 bootcmd 环境变量中的命令启动 Linux)

5.2.5 继续移植、编译 u-boot

(1) 到此为止,再次使用 make 编译 u-boot,会出现如下错误:

```
In file included from /work/system/u-boot-1.1.6/include/nand.h:29,
    from nand.c:28,
/work/system/u-boot-1.1.6/include/linux/mtd/nand.h:412: error: `NAND_MAX_CHIPS'
undeclared here (not in a function)
nand.c:35: error: `CFG_MAX_NAND_DEVICE' undeclared here (not in a function)
nand.c:38: error: `CFG_NAND_BASE' undeclared here (not in a function)
nand.c:35: error: storage size of `nand_info' isn't known
nand.c:37: error: storage size of `nand_chip' isn't known
nand.c:38: error: storage size of `base_address' isn't known
nand.c:37: warning: `nand_chip' defined but not used
nand.c:38: warning: `base_address' defined but not used
make[1]: *** [nand.o] Error 1
make[1]: Leaving directory /work/system/u-boot-1.1.6/drivers/nand
make: *** [drivers/nand/libnand.a] Error 2
```

根据提示,在 include/configs/my2440.h 倒数第二行增加宏定义:

```
#define CFG_MAX_NAND_DEVICE 1
#define NAND_MAX_CHIPS 1
#define CFG_NAND_BASE 0
```

再次 make, 会出现如下错误:

```
drivers/nand/libnand.a(nand.o)(.text+0x24): In function `nand_init':
/work/system/u-boot-1.1.6/drivers/nand/nand.c:50: undefined reference to `board_nand_init'
make: *** [u-boot] Error 1
```

这是由于 u-boot 的编写者并不知道最终开发板所使用的 Nand Flash 的类型, 因此他也就不能编写具体 Nand Flash 的裸驱动, 这个任务留给了移植者。因此需要自行编写 nand flash 的裸驱动。

(2) 新建 cpu/arm920t/s3c24x0/nand_flash.c 文件(光盘中\work\sysbuild\nand_flash.c 已写好), 以实现与具体开发板相关的 board_nand_init 函数。

(3) 在 include/s3c24x0.h 文件, 仿照 S3C2410_NAND 定义 2440 的 Nand Flash 控制器寄存器的数据结构, 以供 board_nand_init 函数使用。

```
typedef struct {
    S3C24X0_REG32    NFCONF;
    S3C24X0_REG32    NFCONT;
    S3C24X0_REG32    NFCMD;
    S3C24X0_REG32    NFADDR;
    S3C24X0_REG32    NFDATA;
    S3C24X0_REG32    NFMECCD0;
    S3C24X0_REG32    NFMECCD1;
    S3C24X0_REG32    NFSECCD;
    S3C24X0_REG32    NFSTAT;
    S3C24X0_REG32    NFESTAT0;
    S3C24X0_REG32    NFESTAT1;
    S3C24X0_REG32    NFMECC0;
    S3C24X0_REG32    NFMECC1;
    S3C24X0_REG32    NFSECC;
    S3C24X0_REG32    NFSBLK;
    S3C24X0_REG32    NFEBLK;
} /* __attribute__((__packed__)) */ S3C2440_NAND;
```

(4) 在 include/s3c2410.h 文件中仿照 S3C2410_GetBase_NAND 函数定义 S3C2440_GetBase_NAND 函数, 以供 board_nand_init 函数调用。

```
static inline S3C2440_NAND * const S3C2440_GetBase_NAND(void)
{
    return (S3C2440_NAND * const)S3C2410_NAND_BASE;
}
```

第5章 建构 BootLoader

(5) 修改 `cpu/arm920t/s3c24x0/Makefile` 第 29 行为 `usb_ohci.o nand_flash.o`, 以将 `nand flash` 裸驱动编译进 `u-boot`。

(6) 移植的其他辅助工作。到此为止, 我们已经得到一个可以在 `s3c2440` 开发板上运行的 `u-boot`。不过这个 `u-boot` 还有一些不足, 例如: 提示符不是 `logo`; 不能执行 `ping` 命令; 不能在命令行进行编辑、不能记忆命令历史; 不能自动加载 Linux 操作系统。下面就一一解决, 其实只需要修改 `include/configs/my2440.h` 即可。

```
#define CONFIG_ETHADDR    08:00:3e:26:0a:5b
#define CFG_PROMPT        "YangZhu > "
#define CONFIG_AUTO_COMPLETE
#define CONFIG_CMDLINE_EDITING
#define CONFIG_BOOTCOMMAND "nand read. jffs2 0x32000000 0x100000 0x300000;
bootm 0x32000000"
#define CONFIG_BOOTARGS    "noinitrd root = /dev/mtdblock2 console = ttySAC0
rootfstype = jffs2"
```

第 82 行增加: `CFG_CMD_PING`

(7) 使用 `H-JTAG`(或其他 `Nor Flash` 烧写软件)将 `Nor Flash` 全部擦除, 如图 5-7 所示, 以删除其中存放的 `u-boot` 环境变量。然后重新将 `u-boot` 烧写到 `Nor Flash` 中。

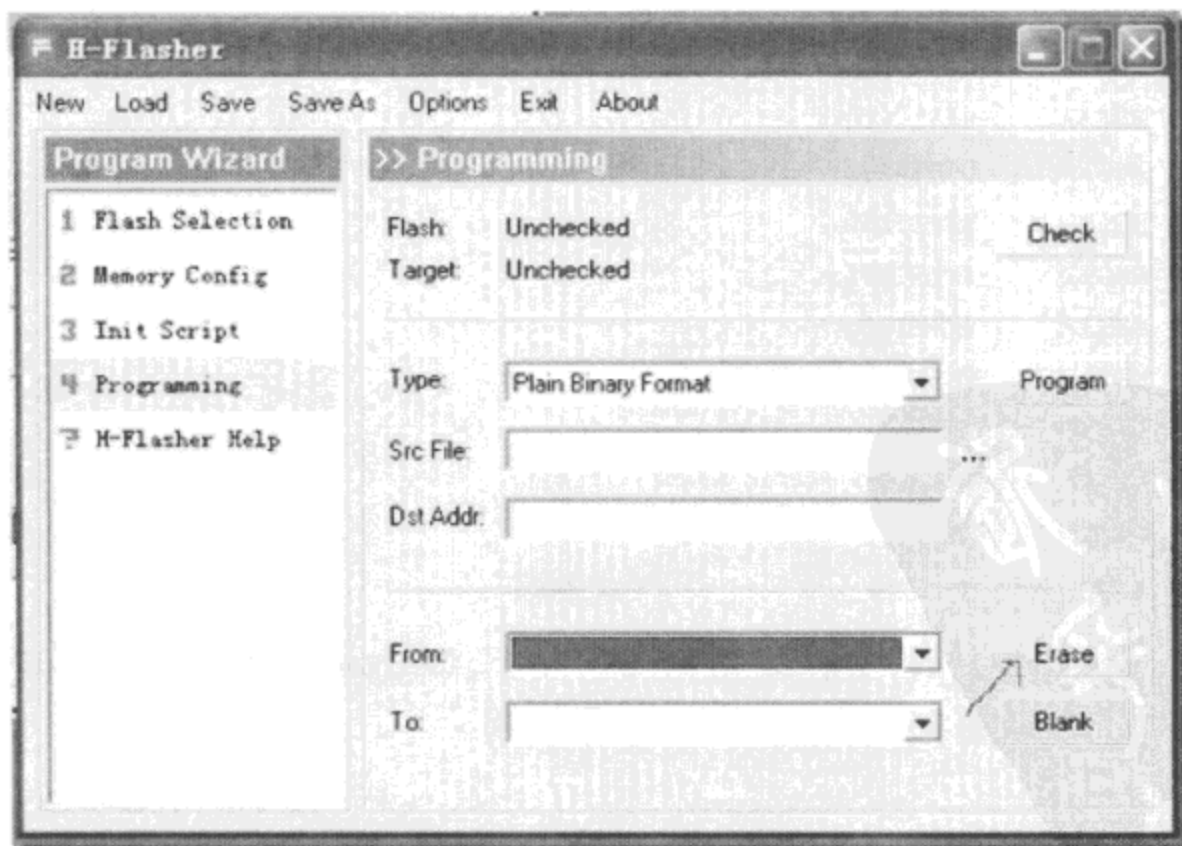


图 5-7 擦除 Nor Flash

(8) 进行配置, 以支持 `dm9000` 网卡。目前移植出来的 `u-boot` 使用的是 `cs8900` 的网卡。

目前市面上买到的 S3C2440 开发板有不少使用的是 dm9000 网卡,因此如果开发板使用的是 dm9000 网卡,就需要修改配置将支持的网卡从 cs8900 改为 dm9000。由于 u-boot 已经内置了 dm9000 的网卡驱动,所以只需要进行简单地配置就可以了。

① 去除对 cs8900 的支持。在 include/configs/my2440.h 中将如下三行注释掉:

```
#define CONFIG_DRIVER_CS8900    1           /* we have a CS8900 on-board */
#define CS8900_BASE              0x19000300
#define CS8900_BUS16             1 /* the Linux driver does accesses as shorts */
```

② 加入对 dm9000 的支持。在 include/configs/my2440.h 中定义如下宏:

```
1  #define CONFIG_DRIVER_DM9000 1
2  #define CONFIG_DM9000_BASE 0x20000300
3  #define DM9000_IO CONFIG_DM9000_BASE
4  #define DM9000_DATA (CONFIG_DM9000_BASE + 4)
5  #define CONFIG_DM9000_USE_16BIT 1
6  #undef CONFIG_DM9000_DEBUG
```

第一行定义的宏会启用 dm9000 网卡,即:编译 dm9000 网卡驱动。u-boot 总是用宏作为开关来控制是否编译某个源代码文件,不妨看看 dm9000 驱动源码:drivers/dm9000x.c

```
#ifdef CONFIG_DRIVER_DM9000 //在文件的最前面
.....
#endif /* CONFIG_DRIVER_DM9000 */ //此行是文件的最后一行
```

在 drivers/cs8900.c 和 drivers/dm9000x.c 中均实现了 eth_send、eth_rx 等操作具体网卡硬件的发送、接收函数,供上层程序(如 ping)调用。如果 CONFIG_DRIVER_DM9000 有定义,而 CONFIG_DRIVER_CS8900 没有定义,则 dm9000x.c 中的函数被编译进 u-boot.bin 中,cs8900.c 中的函数则没有被编译。这就是 u-boot 中多个同类型设备驱动并存的原理。

第 2 行定义 dm9000 网卡的访问基地址,这也许是你为不同开发板移植 dm9000 驱动,可能唯一需要修改的源代码,其值是多少取决于网卡芯片与 s3c2440 芯片的连接方式,这需要查看硬件连接图。

第 3、4 行定义 dm9000 网卡的 I/O 和 DATA 访问基地址。

第 5 行定义 dm9000 数据总线的宽度为 16bit。

(9) 让 u-boot 支持能从 Nand Flash 启动。s3c2440 除了可以从 Nor Flash 启动外,还可以从 Nand Flash 启动。当从 Nand Flash 启动的时候,s3c2440 硬件会自动将 Nand Flash 的前 4KB 内容复制到你内部的一个 4KB 的 RAM(被称为 stepping stone)中,且该 RAM 的内存地址被映射为 0~4095。

目前得到的 u-boot 只能烧入 Nor Flash 中才能使用,请进行如下移植操作,使得既可以

从 Nor Flash 又可以从 Nand Flash 中启动。

① 在 cpu/arm920t/start.S 的 191 行之后加入以下代码：

```
/* Test boot from Nand flash or Nor flash, Add by Dennis Yang */
1   ldr r1, = 4092
2   ldr r2, = 0x55555555
3   str r2, [r1]
4   ldr r0, [r1]
5   teq r0, r2
6   bne clear_bss
7   ldr r2, = 0xaaaaaaaa
8   str r2, [r1]
9   ldr r0, [r1]
10  teq r0, r2
11  bne clear_bss
/* Load U-boot from Nand flash to RAM, Add by Dennis Yang */
12  ldr r0, _TEXT_BASE
13  mov r1, #0
14  ldr r2, _bss_start
15  sub r2, r2, r0          /* r2 <- size of armboot */
16  bl nand_read_ll
```

这段代码的 1~11 行是在测试目前 u-boot 是从 Nor Flash 中还是从 Nand Flash 中启动。其原理是：当从 Nand Flash 启动时，内存 0~4KB 的 stepping stone 是 RAM，可写入；而从 Nor Flash 启动，内存 0~4KB 是 Nor Flash，不可写入。因此如果往内存 4092 写入一个字 0x55555555，之后再从 4092 读出数据，如果该数据是 0x55555555 的话，就说明 0~4KB 是可写入的，也就说明是从 Nand Flash 启动的。当然可能 4092 处本来就存放的是 0x55555555，所以为了应对这种情况，7~11 行执行了对内存 4092 进行写入 0xaaaaaaaa，再读出、比较。至于选择 4KB RAM 的后部作为写入目的地，是为了避免破坏 4KB RAM 中的 u-boot 代码。

在确认 u-boot 是从 Nand Flash 启动后，第 16 行调用 C 子函数 nand_read_ll，将存放在 Nand Flash 上的 u-boot.bin 复制到内存 0x33f80000 处。

② 实现 nand_read_ll 函数。

- 将光盘中的\work\sysbuild\nand_read_ll.c(该文件实现了 nand_read_ll 函数)放到 u-boot-1.1.6 源代码的 cpu/arm920t/s3c24x0 中

- 修改 cpu/arm920t/s3c24x0/Makefile，将第 29 行改为
usb_ohci.o nand_flash.o nand_read_ll.o
以使 nand_read_ll 能被编译进 u-boot.bin

- 修改 board/my2440/u-boot.lds，增加第 36 行
cpu/arm920t/s3c24x0/nand_read_ll.o (.text)

以确保 nand_read_ll 能被编译进 u-boot.bin 的前 4KB。

③ 修改 include/configs/my2440.h 里面的配置参数,使得 u-boot 的环境变量保存到 Nand Flash 中而不是 Nor Flash 中。注释掉下面二行,添加下面三行。

```
// #define CFG_ENV_IS_IN_FLASH 1
// #define CFG_ENV_SIZE 0x10000 /* Total Size of Environment Sector */

#define CFG_ENV_IS_IN_NAND 1
#define CFG_ENV_OFFSET 0x60000
#define CFG_ENV_SIZE 0x20000 /* Total Size of Environment Sector */
```

5.2.6 u-boot 常用命令使用简介

在串口中进入 u-boot 控制界面后,可以运行各种命令,比如下载文件到内存,擦除、读写 Flash,运行内存、Nor Flash、Nand Flash 中的程序,查看、修改、比较内存中的数据等。

使用各种命令时,可以使用其开头的若干个字母代替它。比如 tftpboot 命令,可以使用 t、tf、tft、tftp 等字母代替,只要其他命令不以这些字母开头即可。

当运行一个命令之后,如果它是可重复执行的(代码中使用 U_BOOT_CMD 定义这个命令时,第 3 个参数是 1),若想再次运行可以直接按 Enter 键。

u-boot 接受的数据都是十六进制,输入时可以省略前缀 0x、0X。

下面介绍常用的命令:

1. 帮助命令 help

运行 help 命令可以看到 u-boot 中所有命令的作用,如果要查看某个命令的使用方法,运行“help 命令名”,比如 help bootm。

可以使用? 来代替 help,比如直接输入?、? bootm。

2. 下载命令

u-boot 支持串口下载、网络下载,USB 下载,相关命令有 loadb、loads、loadx、loady 和 tftpboot、nfs、usbslave。

前几个串口下载命令使用方法相似,以 loadx 命令为例,它的用法为“loadx [off] [baud]”。中括号“[]”表示里面的参数可以省略,off 表示文件下载后存放的内存地址,baud 表示使用的比特率。如果 baud 参数省略,则使用当前的比特率;如果 off 参数省略,存放的地址为配置文件中定义的宏 CFG_LOAD_ADDR。

tftpboot 命令使用 TFTP 协议从服务器下载文件,服务器的 IP 地址为环境变量 serverip。用法为“tftpboot [loadAddress] [bootfilename]”,loadAddress 表示文件下载后存放的内存地址,bootfilename 表示要下载的文件名称。如果 loadAddress 省略,存放的地址为配置文件

第5章 建构 BootLoader

中定义的宏 `CFG_LOAD_ADDR`; 如果 `bootfilename` 省略, 则使用单板的 IP 地址构造一个文件名, 比如单板 IP 为 192.168.1.17, 则缺省的文件名为 `C0A80711.img`。

`nfs` 命令使用 NFS 协议下载文件, 用法为 `nfs [loadAddress] [host ip addr: bootfilename]`。 `loadAddress`、`bootfilename` 的意义与 `tftpboot` 命令一样, `host ip addr` 表示服务器的 IP 地址, 默认为环境变量 `serverip`。

`usbslave` 命令: 在 PC 上使用 `dnw` 工具发送文件, `u-boot` 通过 USB Device 接口接收文件。用法为 `usbslave [wait] [loadAddress]`, `wait` 可以取值 1 或 0, 表示是否等得数据传输完成。当 `wait` 取 0 时, 在后台进行下载, 这时在 `u-boot` 仍可执行其他操作。

下载文件成功后, `u-boot` 会自动创建或更新环境变量 `filesize`, 它表示下载的文件长度, 可以在后续命令中使用 `$ (filesize)` 来引用它。

3. 内存操作命令

常用的命令有查看内存命令 `md`、修改内存命令 `mm`、填充内存命令 `mw`、复制命令 `cp`。这些命令都可以带上扩展名 `.b`、`.w` 或 `.l`, 表示以字节、字(2 字节)、双字(4 字节)为单位进行操作。比如 `cp.l 30000000 31000000 2` 将从开始地址 `0x30000000` 处, 复制 2 双字到开始地址为 `0x31000000` 的地方。

`md` 命令用法为 `md[.b, .w, .l] address [count]`, 表示以字节、字或双字(默认为双字)为单位, 显示从地址 `address` 开始的内存数据, 显示的数据个数为 `count`。

`mm` 命令用法为 `mm[.b, .w, .l] address`, 表示以字节、字或双字(默认为双字)为单位, 从地址 `address` 开始修改内存数据。执行 `mm` 命令后, 输入新数据后按 Enter 键, 地址会自动增加, `Ctrl+C` 退出。

`mw` 命令用法为 `mw[.b, .w, .l] address value [count]`, 表示以字节、字或双字(默认为双字)为单位, 往开始地址为 `address` 的内存中填充 `count` 个数据, 数据值为 `value`。

`cp` 命令用法为 `cp[.b, .w, .l] source target count`, 表示以字节、字或双字(默认为双字)为单位, 从源地址 `source` 的内存复制 `count` 个数据到目的地址的内存。

4. Nor Flash 操作命令

常用的命令有查看 Flash 信息的 `flinfo` 命令、加/解写保护命令 `protect`、擦除命令 `erase`。由于 Nor Flash 的接口与一般内存相似, 所以一些内存命令可以在 Nor Flash 上使用, 比如读 Nor Flash 时可以使用 `md`、`cp` 命令, 写 Nor Flash 时可以使用 `cp` 命令(`cp` 根据地址分辨出是 Nor Flash, 从而调用 Nor Flash 驱动完成写操作)。

直接运行 `flinfo` 即可看到 Nor Flash 的信息, 有 Nor Flash 的型号、容量、各扇区的开始地址、是否只读等信息。比如对于本书基于的开发板, `flinfo` 命令的结果如下:

```
Bank # 1: AMD: 1x Amd29LV800BB (8Mbit)
Size: 1 MB in 19 Sectors
```


Sector Start Addresses:

```
00000000 (RO) 00004000 (RO) 00006000 (RO) 00008000 (RO) 00010000 (RO)
00020000 (RO) 00030000 00040000 00050000 00060000
00070000 00080000 00090000 000A0000 000B0000
000C0000 000D0000 000E0000 000F0000 (RO)
```

其中的 RO 表示该扇区处于写保护状态,只读。

对于只读的扇区,在擦除、烧写它之前,要先解除写保护。最简单的命令为 protect off all,解除所有 Nor Flash 的写保护。

erase 命令常用的格式为 erase start end ^{offset}——擦除的地址范围为 start~end、erase start + len——擦除的地址范围为 start~(start + len - 1),erase all——表示擦除所有 Nor Flash。

注意:其中的地址范围,刚好是一个扇区的开始地址到另一个(或同一个)扇区的结束地址。比如要擦除 Amd29LV800BB 的前 5 个扇区,执行的命令为 erase 0 0x2ffff,而非 erase 0 0x30000。

5. Nand Flash 操作命令

Nand Flash 操作命令只有一个:nand,它根据不同的参数进行不同操作,比如擦除、读取、烧写等。

nand info 查看 Nand Flash 信息。

nand erase [clean] [off size]擦除 Nand Flash。加上 clean 时,表示在每个块的第一个扇区的 OOB 区加写入清除标记;off、size 表示要擦除的开始偏移地址和长度,如果省略 off 和 size,表示要擦除整个 Nand Flash。

nand read[.jffs2] addr off size 从 Nand Flash 偏移地址 off 处读出 size 个字节的数据,存放到开始地址为 addr 的内存中。是否加扩展名.jffs 的差别只是读操作时的 ECC 校验方法不同。

nand write[.jffs2] addr off size 把开始地址为 addr 的内存中的 size 个字节数据,写到 Nand Flash 的偏移地址 off 处。是否加后缀“.jffs”的差别只是写操作时的 ECC 校验方法不同。

nand read.yaffs addr off size 从 Nand Flash 偏移地址 off 处读出 size 个字节的数据(包括 OOB 区域),存放到开始地址为 addr 的内存中。

nand write.yaffs addr off size 把开始地址为 addr 的内存中的 size 个字节数据(其中有要写入 OOB 区域的数据),写到 Nand Flash 的偏移地址 off 处。

nand dump off,将 Nand Flash 偏移地址 off 的一个扇区的数据打印出来,包括 OOB 数据。

6. 环境变量命令

printenv 命令打印全部环境变量,printenv name1 name2 ... 打印名字为 name1、

name2,... 的环境变量。

setenv name value 设置名字为 name 的环境变量的值为 value。

setenv name 删除名字为 name 的环境变量。

上面的设置、删除操作只是在内存中进行, saveenv 将更改后的所有环境变量写入 Nor Flash(或 Nand Flash)中。

7. 启动命令

不带参数的 boot、bootm 命令都是执行环境变量 bootcmd 所指定的命令。

“bootm [addr [arg ...]]”命令启动存放在地址 addr 处的 u-boot 格式的映像文件(使用 u-boot 目录 tools 下的 mkimage 工具制作得到), [arg ...] 表示参数。如果 addr 参数省略, 映像文件所在地址为配置文件中定义的宏 CFG_LOAD_ADDR。

go addr [arg ...] 与 bootm 命令类似, 启动存放在地址 addr 处的二进制文件, [arg ...] 表示参数。

nboot [[[loadAddr] dev] offset] 命令将 Nand Flash 设备 dev 上偏移地址 off 处的映像文件复制到内存 loadAddr 处, 然后, 如果环境变量 autostart 的值为 yes, 就启动这个映像。

如果 loadAddr 参数省略, 存放地址为配置文件中定义的宏 CFG_LOAD_ADDR; 如果 dev 参数省略, 则它的取值为环境变量 bootdevice 的值; 如果 offset 参数省略, 则默认为 0。

5.2.7 u-boot 命令实现框架的分析

lib_arm/board.c 的第 398 行调用 main_loop 函数(位于 common/Main.c)进入命令行模式。简化后的 main_loop 函数, 如下:

```
1 void main_loop(void)
2 {
3     static char lastcommand[CFG_CBSIZE] = { 0, };
4     int len;
5     int rc = 1;
6     int flag;
7
8     #if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
9         char *s;
10        int bootdelay;
11    #endif
12
13    #if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
14        s = getenv("bootdelay");
15        bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;
```

```

16     s = getenv("bootcmd");
17     if (bootdelay >= 0 && s && ! abortboot(bootdelay)) {
18         run_command(s, 0);
19     }
20 #endif /* CONFIG_BOOTDELAY */
21 /*
22  * Main Loop for Monitor Command Processing
23  */
24 for (;;) {
25     len = readline(CFG_PROMPT);
26     flag = 0; /* assume no special flags for now */
27     if (len > 0)
28         strcpy(lastcommand, console_buffer);
29     else if (len == 0)
30         flag |= CMD_FLAG_REPEAT;
31     if (len == -1)
32         puts("<INTERRUPT>\n");
33     else
34         rc = run_command(lastcommand, flag);
35     if (rc <= 0) {
36         /* invalid command or not repeatable, forget it */
37         lastcommand[0] = 0;
38     }
39 }
40 }
41
42 static __inline__ int abortboot(int bootdelay)
43 {
44     int abort = 0;
45
46     printf("Hit any key to stop autoboot: %2d ", bootdelay);
47     while ((bootdelay > 0) && (! abort)) {
48         int i;
49
50         --bootdelay;
51         /* delay 100 * 10ms */
52         for (i = 0; ! abort && i < 100; ++i) {
53             if (tstc()) { /* we got a key press */
54                 abort = 1; /* dont auto boot */

```

资源分享网
PDG

```

55         bootdelay = 0; /* no more delay      */
56         (void) getc(); /* consume input      */
57         break;
58     }
59     udelay (10000);
60 }
61 printf ("\b\b\b%2d ", bootdelay);
62 }
63 putc ('\n');
64 return abort;
65 }

```

显然其功能是从终端读入用户输入的命令字符串,然后将该字符串作为参数调用 34 行的 `run_command` 函数(位于 `common/Main.c`)执行该命令,然后循环读取下一个用户输入的命令。简化后的 `run_command` 函数如下:

```

1 int run_command (const char * cmd, int flag)
2 {
3     cmd_tbl_t * cmdtp;
4     char cmdbuf[CFG_CBSIZE]; /* working copy of cmd */
5     char * token; /* start of token in cmdbuf */
6     char * sep; /* end of token (separator) in cmdbuf */
7     char finaltoken[CFG_CBSIZE];
8     char * str = cmdbuf;
9     char * argv[CFG_MAXARGS + 1]; /* NULL terminated */
10    int argc, inquotes;
11    int repeatable = 1;
12    int rc = 0;
13    clear_ctrlc(); /* forget any previous Control C */
14    strcpy (cmdbuf, cmd);
15    /* Process separators and check for invalid
16     * repeatable commands
17     */
18    while (* str) {
19        /*
20         * Find separator, or string end
21         * Allow simple escape of ';' by writing "\;"
22         */
23        for (inquotes = 0, sep = str; * sep; sep++) {
24            if ((* sep == '\\') &&

```




```

25         (* (sep - 1) != '\\')
26         inquotes = ! inquotes;
27
28     if (! inquotes &&
29         (* sep == ';' && /* separator */
30         (sep != str) && /* past string start */
31         (* (sep - 1) != '\\') /* and NOT escaped */
32         break;
33     }
34     /*
35     * Limit the token to data between separators
36     */
37     token = str;
38     if (* sep) {
39         str = sep + 1; /* start of command for next pass */
40         * sep = '\0';
41     }
42     else
43         str = sep; /* no more commands for next pass */
44     /* find macros in this token and replace them */
45     process_macros (token, finaltoken);
46     /* Extract arguments */
47     if ((argc = parse_line (finaltoken, argv)) == 0) {
48         rc = -1; /* no command at all */
49         continue;
50     }
51     /* Look up command in command table */
52     if ((cmdtp = find_cmd(argv[0])) == NULL) {
53         printf ("Unknown command '%s' - try 'help'\n", argv[0]);
54         rc = -1; /* give up after bad command */
55         continue;
56     }
57     /* found - check max args */
58     if (argc > cmdtp->maxargs) {
59         printf ("Usage: %s\n", cmdtp->usage);
60         rc = -1;
61         continue;
62     }
63     /* OK - call function to do the command */

```



```

64         if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {
65             rc = -1;
66         }
67         repeatable &= cmdtp->repeatable;
68         /* Did the user stop this? */
69         if (had_ctrlc())
70             return 0;          /* if stopped then not repeatable */
71     }
72     return rc ? rc : repeatable;
73 }

struct cmd_tbl_s {
    char        * name;          /* Command Name      */
    int         maxargs;         /* maximum number of arguments */
    int         repeatable;      /* autorepeat allowed? */
    /* Implementation function */
    int         (* cmd)(struct cmd_tbl_s *, int, int, char *[]);
    char        * usage;         /* Usage message    (short) */
};

typedef struct cmd_tbl_s  cmd_tbl_t;

```

很明显,第50行之前均是在解析用户命令字符串。例如,如果用户输入的是 ping 192.168.1.11,则当程序运行到50行时,argv[0]为字符串 ping。52行以字符串“ping”作为参数调用 find_cmd 函数,得到一个指向结构体的指针 cmdtp,cmdtp 所指向的结构体存放的内容是:{"ping", 2, 1, do_ping, "ping\t-send ICMP ECHO_REQUEST to network host\n"}(为何是这样的值,请见后面的分析)。查看第3和64行可知:64行实际上是执行 do_ping 函数。查阅 common/Cmd_net.c 的224~243可知,函数 do_ping 就是 ping 命令对应的实现代码。现在只剩下一个问题,就是:为什么 find_cmd 函数仅根据参数“ping”就能找到一个结构体,且该结构体的第4个字段存放的就是实现 ping 命令的函数的函数指针。

查阅 common/cmd_net.c 的245~249行:

```

U_BOOT_CMD(
    ping,      2,      1,      do_ping,
    "ping\t- send ICMP ECHO_REQUEST to network host\n",
    "pingAddress\n"
);

```

以及 include/common.h 相关行:

```

#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help)
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage}

```

宏替换后变为：

```
cmd_tbl_t __u_boot_cmd_ping Struct_Section = {"ping", 2, 1, do_ping, "ping\t- send ICMP ECHO_
REQUEST to network host\n"};
```

```
#define Struct_Section __attribute__((unused,section(".u_boot_cmd")))
```

宏替换后变为：

```
cmd_tbl_t __u_boot_cmd_ping __attribute__((unused,section(".u_boot_cmd"))) =
{"ping", 2, 1, do_ping, "ping\t-send ICMP ECHO_REQUEST to network host\n"};
```

这句代码：①定义了一个全局已初始化变量；②变量名为__u_boot_cmd_ping；③变量类型是结构体类型 cmd_tbl_t；④该结构体变量的第一个字段存放的是用户将来会在 u-boot 命令行中输入的命令名称字符串——"ping"；⑤该结构体变量的第 4 个字段存放的是实现该命令的函数的函数名称；⑥该变量存放在 u-boot.bin 的二进制代码的 .u_boot_cmd 段。

再查看链接脚本：

```
48      . = .;
49      __u_boot_cmd_start = .;
50      .u_boot_cmd : { *(.u_boot_cmd) }
51      __u_boot_cmd_end = .;
```

可知：

- u-boot 所支持的每个命令均有一个结构体与其相对应，该结构体的第一个字段存放的是命令名字符串，第 4 个字段存放的是命令实现函数的函数指针。
- 所有结构体依次存放在 u-boot 最终的二进制代码中。
- 结构体集合在内存中的起始地址由符号 __u_boot_cmd_start 表示；结束地址由符号 __u_boot_cmd_end 表示。

简化后的 find_cmd 函数如下：

```
1 cmd_tbl_t * find_cmd (const char * cmd)
2 {
3     cmd_tbl_t * cmdtp;
4     cmd_tbl_t * cmdtp_temp = &__u_boot_cmd_start;    /* Init value */
5     const char * p;
6     int len;
7     int n_found = 0;
8     /*
9      * Some commands allow length modifiers (like "cp.b");
10     * compare command name only until first dot.
11     */
12     len = ((p = strchr(cmd, '.')) == NULL) ? strlen(cmd) : (p - cmd);
```

```

13     for (cmdtp = &__u_boot_cmd_start;
14         cmdtp != &__u_boot_cmd_end;
15         cmdtp++) {
16         if (strncmp(cmd, cmdtp->name, len) == 0) {
17             if (len == strlen(cmdtp->name))
18                 return cmdtp; /* full match */
19             cmdtp_temp = cmdtp; /* abbreviated command? */
20             n_found++;
21         }
22     }
23     if (n_found == 1) { /* exactly one match */
24         return cmdtp_temp;
25     }
26     return NULL; /* not found or ambiguous command */
27 }

```

由 13~22 不难看出, find_cmd 函数以命令名字符串为查找标准遍历整个结构体集合, 从而找到与该命令对应的结构体。

由此可得, 如果需要向 u-boot 中增加自定义命令(以点亮或熄灭 LED 灯的命令为例), 需要执行以下步骤:

- 编写实现 leds 命令的函数以及 U_BOOT_CMD 宏。形成源代码文件 cmd_leds.c(光盘\work\sysbuild\cmd_leds.c)。
- 将该文件放到 common 目录下, 并修改 common/Makefile 以使该 c 文件能被编译。
- 在 include/cmd_confdefs.h 第 98 行增加宏定义 CFG_CMD_LEDS, 以便使 leds 命令能被用户通过 include/configs/my2440.h 进行裁减。# define CFG_CMD_LEDS 0x8000000000000000ULL /* Control LEDs, Add by Dennis Yang */。
- 修改 include/configs/my2440.h 中对 CONFIG_COMMANDS 宏的定义, 使其包含 CFG_CMD_LEDS 宏。

特别说明, 由于对 LED 的控制, 需要增加对 GPIO(GPB5-8)进行一次性的设置的代码, 这段代码放在 board_init 函数中最合适:

```

gpio->GPBCON = (gpio->GPBCON & ~(0xff<<10)) | (0x55<<10); //configure GPB5 - 8 as out-
put for LEDs, Add by Dennis Yang

```

在 u-boot 命令行中可以使用下面的命令点亮第 4 号灯:

```
Dennis Yang > leds 3 1
```

5.2.8 u-boot 引导 Linux 操作系统的过程分析

在 u-boot 命令提示符下, 输入 boot 后就可以看到成功引导了 Linux, 但遗憾的是在

Linux 启动的最后阶段不能够成功地挂载根文件系统,这是因为 Linux 没有获得正确的启动参数,该参数应由 u-boot 在引导 Linux 前准备好。

在 u-boot 命令提示符下,输入 boot 就可以引导 Linux,这是因为 bootcmd 环境变量的定义为:nand read.jffs2 0x32000000 0x100000 0x300000; bootm 0x32000000,输入 boot 时,u-boot 将执行 bootcmd 环境变量中定义的命令。第一条命令是将 linux kernel 从 Nand Flash 上 0x100000——0x400000 这个位置读入到 RAM 的 0x32000000 这个位置,第二条命令则是使用已经位于 RAM 地址 0x32000000 处的 Linux kernel 启动操作系统。所以,分析 u-boot 引导 Linux 操作系统的过程就从分析 bootm 命令的实现开始。

bootm 命令的实现函数是 do_bootm(位于 common/cmd_bootm.c)。简化的 do_bootm 函数如下:

```
1 int do_bootm (cmd_tbl_t * cmdtp, int flag, int argc, char * argv[])
2 {
3     ulong    iflag;
4     ulong    addr;
5     ulong    data, len, checksum;
6     int      i, verify;
7     char      * name, * s;
8     image_header_t * hdr = &header;
9     s = getenv ("verify");
10    verify = (s && (*s == 'n')) ? 0 : 1;
11    if (argc < 2) {
12        addr = load_addr;
13    } else {
14        addr = simple_strtoul(argv[1], NULL, 16);
15    }
16    printf ("## Booting image at %08lx ... \n", addr);
17    /* Copy header so we can blank CRC field for re-calculation */
18    memmove (&header, (char *)addr, sizeof(image_header_t));
19    if (ntohl(hdr->ih_magic) != IH_MAGIC) {
20        puts ("Bad Magic Number\n");
21        return 1;
22    }
23    data = (ulong)&header;
24    len = sizeof(image_header_t);
25    checksum = ntohl(hdr->ih_hcrc);
26    hdr->ih_hcrc = 0;
27    if (crc32 (0, (uchar *)data, len) != checksum) {
```

```
28     puts ("Bad Header Checksum\n");
29     return 1;
30 }
31 print_image_hdr ((image_header_t *)addr);
32 data = addr + sizeof(image_header_t);
33 len  = ntohl(hdr->ih_size);
34 if (verify) {
35     puts ("    Verifying Checksum ... ");
36     if (crc32 (0, (uchar *)data, len) != ntohl(hdr->ih_dcrc)) {
37         printf ("Bad Data CRC\n");
38         return 1;
39     }
40     puts ("OK\n");
41 }
42 if (hdr->ih_arch != IH_CPU_ARM)
43 {
44     printf ("Unsupported Architecture 0x%x\n", hdr->ih_arch);
45     return 1;
46 }
47 switch (hdr->ih_type) {
48 case IH_TYPE_KERNEL;
49     name = "Kernel Image";
50     break;
51 }
52 /*
53  * We have reached the point of no return; we are going to
54  * overwrite all exception vector code, so we cannot easily
55  * recover from any failures any more...
56  */
57 iflag = disable_interrupts();
58 switch (hdr->ih_comp) {
59 case IH_COMP_NONE;
60     if(ntohl(hdr->ih_load) == addr) {
61         printf ("    XIP %s ... ", name);
62     } else {
63         memmove ((void *) ntohl(hdr->ih_load), (uchar *)data, len);
64     }
65     break;
66 }
```



```

67     puts ("OK\n");
68     switch (hdr->ih_os) {
69     default:                                /* handled by (original) Linux case */
70     case IH_OS_LINUX:
71         do_bootm_linux (cmdtp, flag, argc, argv,
72             addr, len_ptr, verify);
73         break;
74     }
75     return 1;
76 }

typedef struct image_header {
    uint32_t    ih_magic;                    /* Image Header Magic Number */
    uint32_t    ih_hcrc;                    /* Image Header CRC Checksum */
    uint32_t    ih_time;                    /* Image Creation Timestamp */
    uint32_t    ih_size;                    /* Image Data Size */
    uint32_t    ih_load;                    /* Data Load Address */
    uint32_t    ih_ep;                      /* Entry Point Address */
    uint32_t    ih_dcrc;                    /* Image Data CRC Checksum */
    uint8_t     ih_os;                      /* Operating System */
    uint8_t     ih_arch;                    /* CPU architecture */
    uint8_t     ih_type;                    /* Image Type */
    uint8_t     ih_comp;                    /* Compression Type */
    uint8_t     ih_name[IH_NMLEN];         /* Image Name */
} image_header_t;

```

可用于 u-boot 的 kernel 是在常规内核 zImage 前加了 64B 个头信息的 uImage, 这个头包含了: ① kernel 的加载和运行地址 0x30008000; ② zImage 的 CRC 校验码等信息(见上面 struct image_header 的定义)。8、14、18 行将 uImage 中的 64B 头暂存在 header 所指向的内存中供后续程序校验 kernel 是否完整正确。19~51 校验 64B 头以及 zImage 是否完整正确。63 行将 zImage 从 0x32000040 复制到 kernel 的正确加载地址 0x30008000。71 行调用 do_bootm_linux 函数(位于 lib_arm/amlinux.c)启动 Linux。简化后的 do_bootm_linux 如下:

```

1 void do_bootm_linux (cmd_tbl_t * cmdtp, int flag, int argc, char * argv[],
2     ulong addr, ulong * len_ptr, int verify)
3 {
4     ulong data;
5     void (* theKernel)(int zero, int arch, uint params);
6     image_header_t * hdr = &header;
7     bd_t * bd = gd->bd;
8     char * commandline = getenv ("bootargs");

```

第5章 建构 BootLoader

```
9   theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
10   setup_start_tag (bd);
11 # ifdef CONFIG_SETUP_MEMORY_TAGS
12   setup_memory_tags (bd);
13 # endif
14 # ifdef CONFIG_CMDLINE_TAG
15   setup_commandline_tag (bd, cmdline);
16 # endif
17   setup_end_tag (bd);
18   /* we assume that the kernel is in place */
19   printf ("\nStarting kernel ... \n\n");
20   cleanup_before_linux ();
21   theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
22 }
```

第9行实际上是 theKernel=0x30008000

第21行实际上相当于:mov r0, #0; mov r1, #362 (362是Linux对s3c2440的编号);
mov r2, #0x30000100(内核的启动参数在内存中的存放位置);ldr pc, =0x30008000

由于在执行21行时,内存0x30008000处已经存放了内核zImage,所以执行21行就相当于跳转到Linux内核的第一条指令,同时将0、362、0x30000100这3个参数传递给了内核。从此u-boot一去不复还,Linux操作系统到来了。

遗憾的是,Linux虽然能够启动,却不能正确挂载根文件系统。这是为什么呢?

应该说,在将内核映像复制到RAM空间中后,就可以准备启动Linux内核了。但是在调用内核之前,应该作一步准备工作,即:设置Linux内核的启动参数(根文件系统在Nand Flash上的位置是其中的一个参数)。u-boot必须将正确的内核的启动参数放到内存0x30000100处,然后将存放地址(0x30000100)告知内核(这通过21行完成)。内核启动后,到0x30000100处去取得参数,内核必须取得正确的根文件系统在Nand Flash上的位置这个参数后,才能挂载根文件系统。

Linux 2.4.x以后的内核都期望以标记列表(tagged list)的形式来传递启动参数。启动参数标记列表以标记ATAG_CORE开始,以标记ATAG_NONE结束。每个标记由标识被传递参数的tag_header结构以及随后的参数值数据结构来组成。数据结构tag和tag_header定义在Linux内核源码的include/asm/setup.h头文件(u-boot复制了该定义,放在include/asm-arm/setup.h)中:

```
struct tag {
    struct tag_header hdr;
    union {
        struct tag_core core;
```

```

struct tag_mem32    mem;
struct tag_videotext videotext;
struct tag_ramdisk  ramdisk;
struct tag_initrd   initrd;
struct tag_serialnr serialnr;
struct tag_revision revision;
struct tag_videolfb videolfb;
struct tag_cmdline  cmdline;
}u;
};
struct tag_header {
    u32 size;
    u32 tag;
};

```

内核至少需要 4 个 tag: start_tag、end_tag、memory_tags、commandline_tag 才能正确启动,这 4 个 tag 分别由 10、17、12、15 行将其复制到 RAM 的 0x30000100 处。

```

static void setup_start_tag (bd_t * bd)
{
    params = (struct tag *) bd->bi_boot_params;
    params->hdr.tag = ATAG_CORE;
    params->hdr.size = tag_size(tag_core);
    params->u.core.flags = 0;
    params->u.core.pagesize = 0;
    params->u.core.rootdev = 0;
    params = tag_next(params);
}

static void setup_memory_tags (bd_t * bd)
{
    int i;
    for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
        params->hdr.tag = ATAG_MEM;
        params->hdr.size = tag_size(tag_mem32);
        params->u.mem.start = bd->bi_dram[i].start;
        params->u.mem.size = bd->bi_dram[i].size;
        params = tag_next(params);
    }
}

```

新华书店
PDG

```

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

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

```

但遗憾的是:12、15 行是条件编译,它并没有被编译进 u-boot,这样一来内核就得不到正确的根文件系统位置,当然就不能正确挂载根文件系统了。所以我们要做的就是,在 include/configs/my2440.h 中定义两个宏:

```

#define CONFIG_SETUP_MEMORY_TAGS 1
#define CONFIG_CMDLINE_TAG 1

```

附带说明:

第 20 行的 cleanup_before_linux () 完成了:①关闭中断;②清空数据 cache;③关闭数据 cache。第 21 行传递了 3 个参数:①0;②Linux 对 s3c2440 的编号;③内核的启动参数在内存中的存放位置。这些都是 Linux 操作系统要求的必须满足的条件,这些要求详见 Linux 源代码的帮助文档:Documentation/arm/Bootimg。

5.2.9 让 u-boot 支持从 USB slave 接口获得数据

通过 USB slave 接口下载数据到开发板是最快的方法,非常具有实用性。光盘中提供了

补丁文件\work\sysbuild\ u-boot-1.1.6-usbslave-dm9000.patch(如果你的开发板使用的是cs8900网卡,请使用 u-boot-1.1.6-usbslave-cs8900.patch)。用该文件给原始的 u-boot-1.1.6 源码打上补丁后,就具备了通过 USB slave 接口下载数据到开发板的功能

```
/work/sysbuild$ patch -p0 < u-boot-1.1.6-usbslave-dm9000.patch
```

将开发板与 PC 通过 USB 线连接起来后,在 u-boot 命令行输入:

```
Dennis Yang > usbslave 1 0x32000000
```

USB host is connected.

在 PC 运行 dnw,就可将用户指定的 PC 上的文件下载到开发板内存 0x32000000 地址处。

要想让 u-boot 支持 USB slave,主要需要完成以下 4 个工作:

- (1) 增加 USB slave 的驱动以及 dma2、usbd 的 ISR。
- (2) 启用 u-boot 对中断的支持(因为默认情况下,u-boot 禁用了中断)。
- (3) 初始化并设置 DMA 控制器(因为 USB 批量传输使用的是 DMA 通道)。
- (4) 增加 usbslave 命令作为用户接口。

感兴趣的读者,可以自行分析补丁文件和 u-boot-1.1.6 源码。



第 6 章

建构嵌入式 Linux 内核

6.1 Linux 内核简介

6.1.1 Linux 内核版本历史

内核源码可以从 www.kernel.org 下载, ARM 体系结构的内核源码补丁可从 www.arm.linux.org.uk/developer 获得。Linux 内核的版本号可以从源码顶层目录下的 Makefile 中看到, 下面几行构成 Linux 版本号: 2.6.22.6。

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 22
EXTRAVERSION = .6
```

内核版本的 PATCHLEVEL, 稳定内核为偶数, 实验内核为奇数。

Linux 内核最初在 1991 年发布, 是 Linus Torvalds 为 386 开发的一个类 Minix 的操作系统。

Linux 1.0 官方版本发行于 1994 年 3 月, 仅支持 386, 仅支持单 CPU 系统。

Linux 1.2 发行于 1995 年 3 月, 是第一个支持多平台 (Alpha\ Sparc\ Mips 等) 的版本。

Linux 2.0 发行于 1996 年 6 月, 包含很多新平台的支持, 但最重要的是支持 SMP。

Linux 2.2 在 1999 年 1 月发行, 极大提升了 SMP 系统性能, 同时支持更多的硬件。

Linux 2.4 在 2001 年 1 月发行, 进一步提升了 SMP 系统的扩展性, 同时集成了很多用于支持桌面系统的特性: USB、PC 卡 (PCMCIA)、内置的即插即用等。

Linux 2.6 发布于 2003 年 12 月, 特点如下:

- (1) 支持更多的平台, 从小规模的嵌入式到服务器级的 64 位系统。
- (2) 使用新的调度器, 进程的切换效率更高。
- (3) 内核服务可被抢占, 使得用户操作可得到更快的响应。
- (4) I/O 子系统进行了大修改, 使得在各种工作负荷下都更具响应性。

- (5) 模块子系统、文件系统都做了大量的改进。
- (6) 合并了 μ Clinux 的功能, 以支持没有 MMU 的 CPU。

6.1.2 内核源码目录结构

Documentation: 内核帮助文档之源头。

init: 内核初始化代码(非引导代码), 相当于应用程序的 main 函数, 包含 main.c 和 version.c, 是研究核心如何工作的好起点。

kernel: 主核心代码, 主要包括进程管理和 irq。与平台相关的代码在 arch/* /kernel 目录下。

mm: 内存管理代码。与平台相关的代码(如 MMU)在 arch/* /mm 中。

ipc: 进程间通信代码。

net: 网络协议栈代码(ipv4、ipx、802、bluetooth、atm), 每个子目录对应网络的一个方面。

fs: 文件系统(如 ext3、fat、ntfs、yaffs2)代码。

lib: 内核代码用到的库函数(如 strlen、memcpy), 与平台相关的代码在 arch/* /lib。

scripts: 在配置内核时用到, 存放了配置内核的一些脚本文件, 如 make menuconfig 命令。

drivers: 设备驱动程序(如 Nand Flash、串口、cs8900), 占整个内核代码的一半以上, 有些是与平台相关的, 有些是平台无关的。

arch/arm/boot: 内核引导代码(compressed/head.S)。

arch/arm/kernel、lib、mm。

arch/arm/mach-s3c2410: 机器平台相关代码。

arch/arm/tools: 存有 mach-types(该文件保存了 Linux 支持的所有开发板机器编号)。

include/linux: 平台无关头文件。

include/asm-arm: ARM 平台相关头文件。

include/asm-arm/arch-s3c2410: 机器平台相关头文件。

6.1.3 Linux 内核构造系统简介

使用 tar 命令解压 Linux 源代码包后, 进入其顶层目录, 使用 make menuconfig 命令可对内核进行配置和裁减, 如图 6-1 所示。保存退出后, 可以使用 make zImage 生成内核 zImage。

整个内核的生成过程非常复杂, 但只使用了 make menuconfig 和 make zImage 两个命令就搞定了, 这是由于庞大而设计精巧的内核构造系统在背后完成了大量工作。内核构造系统由三部分组成:

1. Kconfig 文件

其作用是: ①控制 make menuconfig 时, 出现的配置选项; ②根据用户的选择, 生成 .config 配置文件。

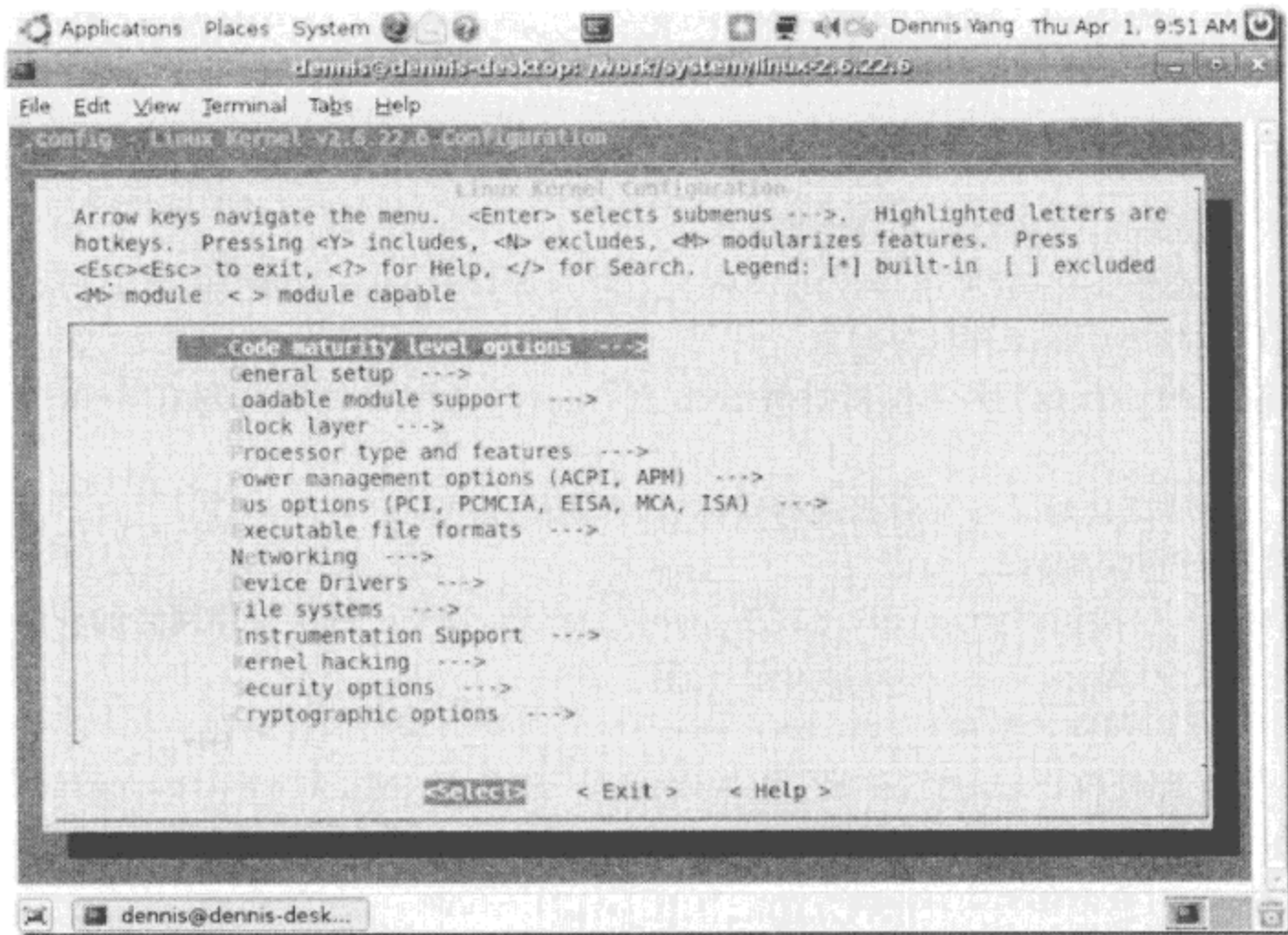


图 6-1 配置内核

主(初始)Kconfig 文件是 arch/\$(ARCH)/Kconfig。

详细信息请参阅 Documentation/kbuild/kconfig-language.txt。

2. .config 文件

该配置文件定义了一系列变量(每一个变量对应一个内核组件), Makefile 将结合它来决定哪些文件被编译进内核、哪些文件被编为模块、进入哪些子目录递归编译。

3. Makefile 文件

(1) 顶层 Makefile 和 arch/\$(ARCH)/Makefile:

- ① 决定根目录下哪些子目录、arch/\$(ARCH)/目录下哪些文件和目录将被编进内核。
- ② 设置可影响所有文件的编译、链接选项: CFLAGS、AFLAGS、LDFLAGS、ARFLAGS。
- ③ 根据链接脚本 arch/\$(ARCH)/kernel/vmlinux.lds, 按照一定顺序组织文件生成映像文件 vmlinux。

(2) 各级子目录下的 Makefile(Kbuild):

- ① 决定所在目录下哪些文件被编进内核, 哪些文件被编成模块, 进入哪些子目录继续调用子目录的 Makefile。

② 设置能够影响当前目录下所有文件的编译、链接选项: EXTRA_CFLAGS、EXTRA_AFLAGS、EXTRA_LDFLAGS、EXTRA_ARFLAGS。

③ 设置可以影响某个文件的编译选项: CFLAGS_\$@、AFLAGS_\$@。

详细信息请参阅 Documentation/kbuild/makefiles.txt。

6.2 移植、裁减及配置 Linux 内核到 S3C2440 开发板

6.2.1 体验 Linux 内核配置、编译与使用

(1) 使用 tar xjvf 命令解压 Linux-2.6.22.6.tar.bz2 后, cd /work/sysbuild/linux-2.6.22.6。

(2) 修改顶层 Makefile 的 185 和 186 行, 以指定特定 CPU 体系结构和交叉编译工具。

```
185 ARCH          = arm
186 CROSS_COMPILE = arm-linux-
```

(3) 内核配置选项有几百个, 配置者不可能一一配置, 因此先输入 make s3c2410_defconfig, 以使用标准模板的配置:

```
/work/system/linux-2.6.22.6$ make s3c2410_defconfig
```

(4) 使用 make menuconfig 进入图形配置界面, 如图 6-2 所示, 浏览基本配置。

单击 System Type—S3C2440 Machines, 如图 6-3 所示。

SMDK2440 和 SMDK2440 with S3C2440 CPU module 已被选择(方括弧内有 * 号), 表明编译出来的内核将支持由 S3C2440 组成的开发板。

(5) 保存退出后, 输入 make zImage, 大约半小时后将在 arch/arm/boot 目录生成内核文件 zImage。

(6) 将制作 uImage 的程序 mkimage(位于编译后的 u-boot 源代码目录的 tools 子目录中)复制到 /usr/bin 目录 sudo cp /work/sysbuild/u-boot-1.1.6/tools/mkimage /usr/bin/。

(7) 输入 make uImage 后, 内核构造系统将调用 mkimage, 通过 zImage 生成 uImage(位于 arch/arm/boot 目录)。uImage 是可供 u-boot 引导的 Linux 内核, 它在 zImage 前增加了 64B 的头(这 64B 的头有什么作用、里面包含什么内容, 请参见前面“深入剖析 u-boot”一节)

```
/work/sysbuild/linux-2.6.22.6$ ls -l arch/arm/boot/[uz]Image
```

```
-rw-r--r-- 1 dennis dennis 1511140 2010-04-01 10:43 arch/arm/boot/uImage
```

```
-rwxr-xr-x 1 dennis dennis 1511076 2010-04-01 10:41 arch/arm/boot/zImage
```

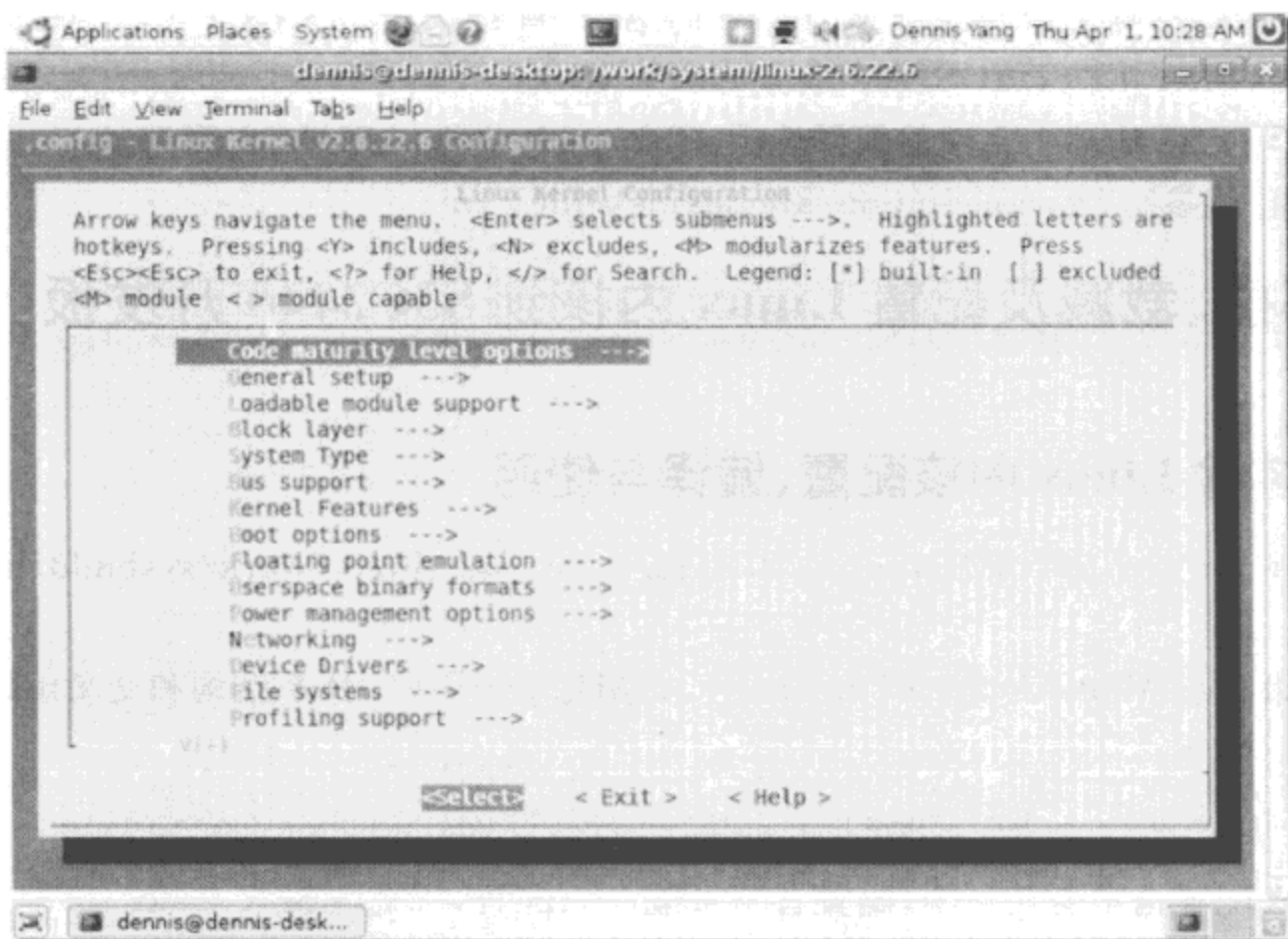


图 6-2 内核配置主界面

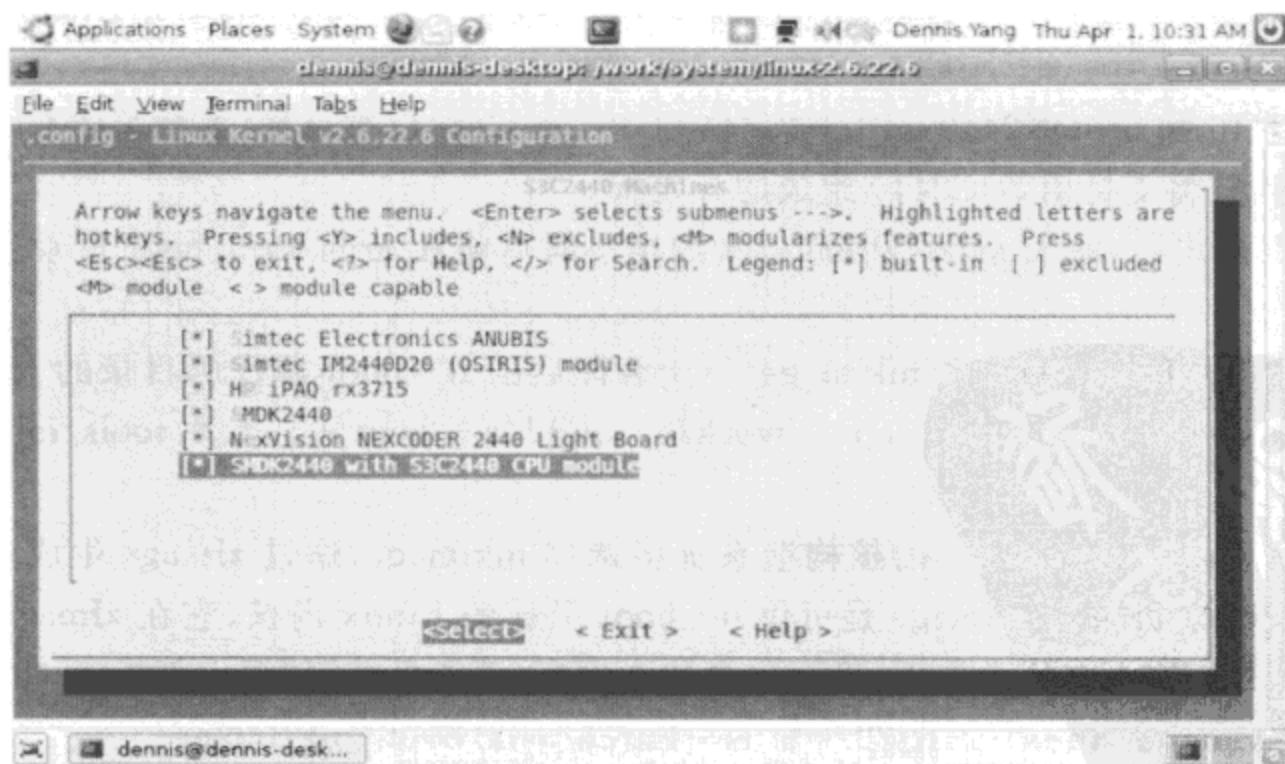


图 6-3 S3C2440 Machines 配置

(8) 将 uImage 烧写到开发板的 Nand Flash 上:

- ① 使用网线将开发板与 Linux 机器物理连通。
- ② 配置 Linux 机器的 IP 地址为 192.168.1.11。
- ③ 配置开发板的 IP 地址为 192.168.1.17:

```
Dennis Yang > setenv ipaddr 192.168.1.17
```

```
Dennis Yang > saveenv
```

```
Saving Environment to NAND...
```

```
Erasing Nand...Writing to Nand... done
```

- ④ 确认开发板能与 Linux 机器通信:

```
Dennis Yang > ping 192.168.1.11
```

```
host 192.168.1.11 is alive
```

- ⑤ 配置 Linux 机器上 NFS 服务器的配置文件/etc/exports,使 NFS 服务器能共享/work/sysbuild/linux-2.6.22.6 目录后,重启 NFS 服务:

```
sudo /etc/init.d/nfs-kernel-server restart
```

- ⑥ 将 uImage 下载到开发板 RAM 地址 0x32000000 处

```
DennisYang > nfs 0x32000000 192.168.1.11:/work/sysbuild/linux-2.6.22.6/arch/arm/boot/uImage
```

- ⑦ 格式化 Nand Flash 某区段(起始地址为 0x100000,长度为 0x300000 字节),并将位于 RAM 地址 0x32000000 的 uImage 烧写到该区段:

```
Dennis Yang > nand erase 0x100000 0x300000
```

```
Dennis Yang > nand write.jffs2 0x32000000 0x100000 0x300000
```

- ⑧ 启动 Linux。可以看到内核正确并被启动,但旋即输出乱码。这说明需要对 Linux 进行移植(修改源代码):

```
Dennis Yang > boot
```

```
NAND read: device 0 offset 0x80000, size 0x180000
```

```
Reading data from 0x1ffe00 -- 100% complete.
```

```
1572864 bytes read; OK
```

```
## Booting image at 32000000 ...
```

```
Image Name: Linux-2.6.22.6
```

```
Created: 2010-04-01 2:43:59 UTC
```

```
Image Type: ARM Linux Kernel Image (uncompressed)
```

```
Data Size: 1511076 Bytes = 1.4 MB
```

```
Load Address: 30008000
```

```
Entry Point: 30008000
```

```
Verifying Checksum ... OK
```




```

OK
Starting kernel ...
Uncompressing Linux..... done,
booting the kernel.

```

鯨筵"甃 7 记净 77 悄 3:癖;? 虞? 薜儻{ \$ 剌# "84 梧 f 九鄢九? f 痼 C? 吾 G4 鞞[

6.2.2 为 S3C2440 移植内核

1. 修改晶振频率

S3C2440 支持两种晶振频率:12MHz 和 16MHz。目前市面上出售的开发板大多使用的是 12MHz 的晶振,而内核源代码则采用的是 16MHz 频率,从而产生了错误的 PCLK,因此导致内核向串口输出数据时使用了错误的比特率(正确的应是 115200),这样在超级终端中看到的的就是乱码。因此我们只须修改内核源代码中的晶振频率即可。

将 arch/arm/mach-s3c2440/mach-smdk2440.c 的第 180 行

```
s3c24xx_init_clocks(16934400);
```

改为:

```
s3c24xx_init_clocks(12000000);
```

2. 修改 Nand Flash 分区表

这次内核正常启动,不再出现乱码,但未能成功挂载根文件系统。出现错误如下:

```

VFS: Mounted root (jffs2 filesystem).
Freeing init memory: 132K
Warning: unable to open an initial console.
Kernel panic - not syncing: No init found. Try passing init= option to kernel.

```

仔细查看内核出错前的输出,发现内核在 Nand Flash 上创建了 8 个分区:

```

Creating 8 MTD partitions on "NAND 64MiB 3,3V 8-bit":
0x00000000 - 0x00004000 : "Boot Agent"
0x00000000 - 0x00200000 : "S3C2410 flash partition 1"
0x00400000 - 0x00800000 : "S3C2410 flash partition 2"
0x00800000 - 0x00a00000 : "S3C2410 flash partition 3"
0x00a00000 - 0x00e00000 : "S3C2410 flash partition 4"
0x00e00000 - 0x01800000 : "S3C2410 flash partition 5"
0x01800000 - 0x03000000 : "S3C2410 flash partition 6"
0x03000000 - 0x04000000 : "S3C2410 flash partition 7"

```

内核从 u-boot 获得的启动参数(root=/dev/mtdblock2)表明,根文件系统放在 Nand Flash 的第 3 个分区上:

Kernel command line: noinitrd root = /dev/mtdblock2 console = ttySAC0 rootfstype = jffs2

这样一来,内核会到 Nand Flash 的 0x00400000~0x00800000 区间来挂载根文件系统,但事实上通过 u-boot 烧写根文件系统时,是将其烧写在 Nand Flash 的 0x400000~0x4000000 这个区间的。所以内核不能成功挂载根文件系统。因此,必须修改内核对 Nand Flash 的分区定义,让第 3 个分区位于 0x400000~0x4000000 这个区间。

修改 arch/arm/plat-s3c24xx/common-smdk.c 文件,将 109~150 行改为如下所示。这样一来,整个 Nand Flash 被划分为 4 个分区。第 1 分区大小 1MB,存放 u-boot;第 2 分区大小 3MB,存放 Linux kernel;第 3 分区大小 60MB,存放根文件系统;第 4 分区从第 64M 开始一直到 Nand Flash 的最后,可用于扩展。

```
static struct mtd_partition smdk_default_nand_part[] = {
    [0] = {
        .name      = "Bootloader",
        .size      = SZ_1M,
        .offset    = 0,
    },
    [1] = {
        .name      = "Kernel",
        .offset    = SZ_1M,
        .size      = SZ_2M + SZ_1M,
    },
    [2] = {
        .name      = "Root filesystem",
        .offset    = SZ_4M,
        .size      = SZ_64M - SZ_4M,
    },
    [3] = {
        .name      = "Extended",
        .offset    = SZ_64M,
        .size      = MTDPART_SIZ_FULL,
    }
};
```

6.2.3 配置并裁减内核

现在内核虽然可以正常工作,但作为嵌入式操作系统的 Linux,是可以被裁减以适应我们的具体需要。下面就通过 make menuconfig 对内核进行配置和裁减,使得最终的内核尽量小,并实现支持:① Nand Flash 驱动;② jffs2 文件系统;③ ELF 格式的应用程序;④ 串口驱动;

第6章 建构嵌入式 Linux 内核

⑤ram disk(将内存当硬盘使用);⑥ext2 文件系统;⑦环回设备(将物理文件当硬盘使用、类似 windows 下的虚拟光驱);⑧将目录挂载到内存(将内存当目录使用);⑨能识别 U 盘 FAT 分区和 NTFS 分区上的中英文文件;⑩更新和获取实时时钟;⑪ watchdog 驱动;⑫ I2C 驱动;⑬ SPI 驱动;⑭ framebuffer;⑮ TCP/IP 协议栈;⑯开发板充当 NFS 客户端;⑰将 NFS 服务器上的共享目录挂载为根文件系统;⑱动态加载和卸载模块;⑲开发板充当 PC 的 U 盘;⑳ USB 网卡 DM9601。

不支持:①网卡驱动;②声卡驱动;③LCD 驱动。

请参见图 6-2,表 6-1 是输入 make menuconfig 后,出现在 kernel 配置主菜单界面上的选项说明。

表 6-1 内核配置主界面选项说明

| 选项名 | 说 明 |
|-----------------------------|---|
| Code maturity level options | 选中则使未定型的功能组件出现在配置选项中,这样就可以显示更多的配置选项 |
| General setup | 杂项设置 |
| Block layer | 块设备层:用于设置块设备的一些总体参数,比如是否支持大于 2TB 的块设备、是否支持大于 2TB 的文件、设置 I/O 调度器等。一般使用默认值即可 |
| System Type | 系统类型:选择 CPU 架构、开发板类型等开发板相关的配置选项 |
| Bus support | PCMCIA/CardBus 总线的支持,不用设置 |
| Kernel Features | 用于设置内核的一些参数,比如是否支持内核抢占,是否支持动态修改系统时钟等 |
| Boot options | 启动参数:比如设置默认的命令行参数等,一般不用理会 |
| Floating point emulation | 浮点运算仿真功能,目前 Linux 还不支持硬件浮点运算,所以要选择一个浮点仿真器,一般选择 NWFPE math emulation 选项 |
| Userspace binary formats | 可执行文件格式,一般选择 ELF |
| Power management options | 电源管理选项 |
| Networking | 网络协议选项:一般都选择 Networking support 选项以支持网络功能,选择 Packet socket 选项,以支持 raw socket 接口功能,选择 TCP/IP networking 选项以支持 TCP/IP 网络协议。通常在选择 Networking support 选项后使用默认配置 |
| Device Drivers | 设备驱动程序:几乎包含 Linux 的所有驱动程序 |
| File systems | 文件系统 |
| profiling support | 对系统的活动进行分析,仅供内核开发者使用 |
| Kernel hacking | 调试内核时的各种选项 |
| Security options | 安全选项,一般使用默认选项 |

续表 6-1

| 选项名 | 说 明 |
|-----------------------|--|
| Cryptographic options | 加密选项 |
| Library routines | 库子程序:比如 CRC32 检验函数、zlib 压缩函数等。不包含在内核源码中的第三方内核模块可能需要这些库,可以全不选,若内核中其他部分依赖它,会自动选上 |

在配置界面上对 Linux 进行配置时,对于[]选项可以按下 Y 键表示要编译进内核(即:该功能组件的二进制代码会进入最终的内核映像文件 zImage 中),N 键表示不编译;对于< >选项,除了 Y 和 N 外,还可以按 M 键表示编译为模块(即:该功能组件的二进制代码并不进入 zImage,但会单独形成一个扩展名为 .ko 的文件。该 KO 文件就被称为模块,模块可以在操作系统启动后通过“insmod 模块文件名”将其加载进内核),以减小内核的大小。

执行以下配置:

(1) 选中 Code maturity level options → Prompt for development and/or incomplete code/drivers 以便显示更多的配置选项。

(2) 保持 General setup 的默认选择。

(3) 保持 Loadable module support 的默认选择,以支持动态加载和卸载模块。

(4) 保持 Block layer 的默认选择。

(5) System Type 默认配置下,保留 S3C2440 Machines 下的 SMDK2440 和 SMDK2440 with S3C2440 CPU module,把其他类型的 Machines 全部去掉。

(6) 保持 Bus support 的默认选择。

(7) 保持 Kernel Features 的默认选择,不要选择 Preemptible Kernel (EXPERIMENTAL)和 Use the ARM EABI to compile the kernel。

(8) 保持 Boot options 的默认选择。

(9) 保持 Floating point emulation 的默认选择。

(10) 去掉 Userspace binary formats 下的“Kernel support for a.out and ECOFF binaries”,只保留“Kernel support for ELF binaries”。

(11) 保持 Power management options 的默认选择。

(12) 去掉 Networking—Networking options 下的“IP: IPsec transport mode”。“IP: IPsec tunnel mode”。“IP: IPsec BEET mode”。

(13) Device Drivers:

① Memory Technology Device (MTD) support:

(a)去掉 RedBoot partition table parsing。

(b)去掉 RAM/ROM/Flash chip drivers 的全部选项。

(c)保持 NAND Device Support—NAND Flash support for S3C2410/S3C2440 SoC,以支持 Nand Flash 驱动。

② 去掉 Parallel port support 的全部选项。

③ Block devices:

(a)保留 Loopback device support 以支持环回设备。

(b)保留 RAM disk support 以支持 ram disk。

④ 去掉对 ATA/ATAPI/MFM/RLL support 的支持。

⑤ 选中 SCSI device support—SCSI device support—SCSI disk support,以支持 U 盘(U 盘在 Linux 中被识别为 SCSI 硬盘)。

⑥ Network device support:

(a)去掉 Ethernet (10 or 100Mbit)—DM9000 support,现在暂不支持网卡驱动。

(b)去掉 Ethernet (1000 Mbit)和 Ethernet (10 000 Mbit)。

(c)USB Network Adapters 下,选中 Multi - purpose USB Networking Framework 和 Davicom DM9601 based USB 1.1 10/100 ethernet devices 以支持 USB 网卡 DM9601;去掉其他选项。

⑦ Input device support:

(a)去掉对 Keyboards 的支持。

(b)去掉对 Mice 的支持。

⑧ Character devices 去掉 Non - standard serial port support 和 Legacy (BSD) PTY support。Watchdog Timer Support 下,仅保留 S3C2410 Watchdog,以支持看门狗驱动;保留 Serial drivers 下的默认配置,以支持串口。

⑨ I2C support 保持默认配置,以支持 I2C 驱动。

⑩ SPI support 选中 Samsung S3C24XX series SPI、Samsung S3C24XX series SPI by GPIO 和 Bitbanging SPI master,以支持 SPI 驱动。

⑪ 去掉 Hardware Monitoring support。

⑫ LED devices 去掉 LED Support。

⑬ Multimedia devices 去掉 DAB adapters。

⑭ Graphics support 下,按默认保留 Support for frame buffer devices 及其相关选项以支持 frame buffer,去掉 S3C2410 LCD framebuffer support,现在暂不支持 LCD 驱动。

⑮ HID Devices 清除所有选项,这将导致不支持 USB 鼠标、键盘等。

⑯ USB support:

(a)选中 USB Mass Storage support,以支持 U 盘。

(b)USB Gadget Support,选中 Support for USB Gadgets,选择将 USB Gadget Drivers—File - backed Storage Gadget 编译为模块。以支持将开发板充当 PC 的 U 盘。(需修改 udc 驱

动源代码。)

⑰ Real Time Clock 保持默认设置,以支持更新和获取实时时钟(需修改 RTC 驱动源代码)。

修改 arch/arm/mach-s3c2440/mach-smdk2440.c,在其第 175 行处增加一行:

&s3c_device_rtc,

(14) File systems:

① 保留 Second extended fs support 以支持 Ext2 文件系统。去掉 Ext3 journalling file system support、ROM file system support。

② 在 DOS/FAT/NT Filesystems 下选中除 NTFS debugging support 之外的全部选项,以支持 FAT 文件系统和 NTFS 文件系统。

③ 选中 Pseudo filesystems—Virtual memory file system support (former shm fs),以支持将目录挂载到内存。

④ Miscellaneous filesystems:

(a)保持 Journalling Flash File System v2 (JFFS2) support,以支持 jffs2 文件系统。

(b)去掉 Compressed ROM file system support (cramfs)。

⑤ Network File Systems:

(a)选中 Provide NFSv3 client support,以支持开发板充当 NFS 客户端。

(b)保持 Root file system on NFS,以支持将 NFS 服务器上的共享目录挂载为根文件系统。

⑥ Partition Types,去掉全部选项。

⑦ Native Language Support。

(a)选中 Codepage 437 (United States, Canada),以支持英文文件。

(b)Simplified Chinese charset (CP936, GB2312),以支持中文文件。

(15) Kernel hacking,清除全部选项。

6.2.4 运行内核并验证内核被配置的功能

(1) Nand Flash 驱动。

(2) jffs2 文件系统。由于根文件系统是 jffs2 文件系统并位于 Nand Flash 上,现在能正确加载根文件系统,说明内核已经支持这两个功能。

(3) ELF 格式的应用程序。ls 命令就是 ELF 格式的应用程序,现在能执行 ls 这样的命令,说明内核已经支持这个功能。

(4) 串口驱动。已能通过超级终端接收和输入数据,说明内核已经支持这个功能。

(5) ram disk。

(6) ext2 文件系统。

```
# mke2fs /dev/ram3
# mount-t ext2 /dev/ram3 /mnt
# touch /mnt/yangzhu
```

执行 `ls /mnt` 显示存在 yangzhu 这个文件。reboot 后,重新执行 `mount-t ext2 /dev/ram3`,报错,因为需要重新格式化/dev/ram3。说明内核已经支持这两个功能。

(7) 环回设备。

```
# dd if=/dev/zero of=/test.img bs=512 count=8190
# losetup /dev/loop0 /test.img
# mke2fs /dev/loop0
# mount -t ext2 /dev/loop0 /mnt
# touch /mnt/yangzhu
```

执行 `ls /mnt` 可以看到存在文件 yangzhu

```
# umount /dev/loop0
```

reboot 后执行:

```
# losetup /dev/loop0 /test.img
# mount -t ext2 /dev/loop0 /mnt
```

执行 `ls /mnt` 可以看到文件 yangzhu 仍然存在。说明内核已经支持这个功能。

(8) 将目录挂载到内存。

```
# mount -t tmpfs none /mnt
# touch /mnt/yangzhu
```

执行 `ls /mnt` 显示存在文件 yangzhu。reboot 后重新执行

```
# mount -t tmpfs none /mnt
```

执行 `ls /mnt` 显示文件 yangzhu 不存在。说明内核已经支持这个功能。

(9) 能识别 U 盘 FAT 分区和 NTFS 分区上的中英文文件。

插入格式化为 FAT 文件系统的 U 盘后,执行

```
# mount /dev/sda1 /mnt (或者需要输入的是 mount /dev/sda /mnt)
# ls /mnt
```

可看到 U 盘上的中英文文件。

插入为格式化为 NTFS 文件系统的 U 盘后,执行

```
mount -t ntfs -o ro,nls=cp936 /dev/sda1 /mnt
ls /mnt
```

可看到 U 盘上的中英文文件。

说明内核已经支持这个功能。

注:之所以要加入 `-o ro` 这个选项,是因为目前 Linux 对于 NTFS 分区的写入操作的支

持很有限,并且不太完善。

(10) 更新和获取实时时钟。

```
date -s 040210102010 # 设置时间为 2010-04-02 10:10
```

```
hwclock -w # 把刚刚设置的时间存入 S3C2440 内部的 RTC
```

reboot 后使用 date 显示当前时间,变为了 2010-04-02 10:11:02。说明内核已经支持这个功能。

(11) Watchdog 驱动。

```
# watchdog -t 1-T 5 /dev/watchdog
```

```
# killall -9 watchdog
```

```
s3c2410 - wdt: Unexpected close, not stopping watchdog!
```

```
#
```

```
u-boot 1.1.6 (Sep 8 2009 - 11:22:15)
```

```
DRAM: 64 MB
```

```
Flash: 1 MB
```

```
NAND:
```

在开启 Watchdog 功能后停止喂狗,结果系统重新启动。这说明内核已经支持这个功能。

注:暂不验证 I2C 驱动、SPI 驱动、framebuffer 等功能,其将在下节中验证。

6.3 内核 Kconfig 与 Makefile 文件分析

6.3.1 内核构造系统简介

内核是个复杂庞大的系统,对它进行配置、裁减、编译原本非常地复杂和困难,但现在却只需要简单的两个命令 make menuconfig 和 make uImage 就搞定了,原因是在其背后有一个设计精巧的内核构造系统帮我们精确完成了各项任务。内核构造系统最关键的组成元素就是各个目录下的 Kconfig 文件和 Makefile 文件,本节将对这两类文件进行介绍,以使大家了解内核构造系统的基本情况,从而能够修改它们,以完成向内核中添加功能组件的目的。

6.3.2 Kconfig 文件精解

Kconfig 文件的作用是:控制 make menuconfig 时,出现的配置选项;并根据用户在配置界面中的选择,将配置结果保存在 .config 配置文件(该文件将供 Makefile 使用,以决定要编译的内核组件以及如何编译)。

主(初始)Kconfig 文件是 arch/arm/Kconfig(打开来看看吧!并试着改一改,看看当 make menuconfig 的时候会出现什么样的变化)。

Kconfig 文件的语法和语义,详情可查阅内核源码中的 Documentation/kbuild/kconfig-language.txt 文件。

1. Kconfig 文件的基本要素:config 条目(entry)

```
config REISERFS_FS_POSIX_ACL
```

```
    bool
```

```
    prompt "ReiserFS POSIX Access Control List"
```

```
    default y
```

```
    depends on REISERFS_FS_XATTR
```

```
    select FS_POSIX_ACL
```

```
    help
```

```
    Posix Access Control Lists (ACLs) support
```

```
    If you don't know what Access Control Lists are, say N
```

上面的 config 条目,各个部分的含义是:

(1) REISERFS_FS_POSIX_ACL 为变量名,将在.config 中以 CONFIG_REISERFS_FS_POSIX_ACL=y 或 n 的形式出现。

(2) bool 为变量取值的类型,可为 y 或 n。

(3) prompt 为出现在配置菜单中的文字,没有它,将使得用户不能在配置界面中显示并配置它。

(4) default 为变量缺省值,可被用户设置值覆盖。

(5) depends on 表示该变量必须在 REISERFS_FS_XATTR 被设置的情况下才能进行设置,否则取值为 n,即使 default 为 y。

(6) select 表示它将影响到变量 FS_POSIX_ACL,使得 FS_POSIX_ACL 至少应该配置为 y 或 m(如果它最终取值为 y 或 m)。

(7) help 中的文字将作为配置界面中的帮助信息。

附加说明:

(1) 无 depends on,default 为 y:默认为 y。一般用于必须要设置的选项,此时不要设置 prompt。

(2) 有 depends on,default 为 y:所依赖的条目已设置,则默认为 y;所依赖的条目未设置,则为 n。

(3) 有 depends on,default 为 n:所依赖的条目已设置,则默认为 n;所依赖的条目未设置,则为 n。

(4) 无 depends on,default 为 n:默认为 n。在未设置 prompt 的情况下,此选项想要被设置,需要由其他选项来 select 它。

2. Kconfig 中变量的取值类型

Kconfig 中变量的取值类型总共有 5 种。其中最常见的是 tristate 和 bool, 分别对应于 make menuconfig 配置界面中 `< >` 和 `[]` 选项。

(1) tristate: 可取 y、n、m。

(2) bool(其为 tristate 的变体): 可取 y、n。

(3) string: 取值为字符串, 如: `CONFIG_CMDLINE="root=/dev/hda1 ro init=/bin/bash console=ttySAC0"`。

(4) hex(其为 string 的变体): 取值为十六进制数据, 如: `CONFIG_VECTORS_BASE=0xffff0000`。

(5) int(其为 string 的变体): 取值为十进制数据, 如: `CONFIG_SPLIT_PTLOCK_CPUS=4096`。

3. Kconfig 文件的要素: menu

在 menu 和 endmenu 中间可配置若干 config 条目;

体现在配置菜单上为 System type `---`, 按下该条目后, 将出现各个 config 条目

4. Kconfig 文件的要素: choice

在 choice 和 endchoice 之间可定义若干 config 条目;

体现在配置菜单上为 ARM system type `---`, 按下该条目后, 将出现各个 config 条目;

choice 中的 config 条目变量只能有两种类型: bool 或 tristate, 且不能同时有这两种类型的变量。对于 bool 型变量只能在多个选择中选择一个为 y; 对于 tristate 型变量要么将多个 (当然也可以是一个) 设为 m, 要么仅将一个设为 y, 其余为 n。这好比一个硬件有多个驱动, 要么选择一个编入内核, 要么把多个全编为模块。

5. Kconfig 文件的要素: comment

用于定义帮助信息, 将出现在配置界面的第一行; 并且还会出现在配置文件 .config 中 (作为注释)。

6. Kconfig 文件的要素: source

由于内核源代码中大多数目录下都有各自的 Kconfig 文件, 因此需要一种手段将所有的 Kconfig 文件组织为一个整体。这就是 source 的功能, 它用于引入另一个 Kconfig 文件, 有点类似于 C 语言中的 `#include`。

6.3.3 .config 文件说明

make menuconfig 配置完成退出时, 选择保存, 则用户所作的选择将保存在内核源代码顶

第6章 建构嵌入式 Linux 内核

层目录的.config文件中。下面.config文件的片断显示内核配置者作了如下选择:将BLK_DEV_LOOP、CONFIG_BLK_DEV_RAM功能编译进zImage;不编译BLK_DEV_COW_COMMON、BLK_DEV_CRYPTOLOOP、BLK_DEV_UB功能;将BLK_DEV_NBD功能编译为模块。

```
484 #
485 # Block devices
486 #
487 # CONFIG_BLK_DEV_COW_COMMON is not set
488 CONFIG_BLK_DEV_LOOP = y
489 # CONFIG_BLK_DEV_CRYPTOLOOP is not set
490 CONFIG_BLK_DEV_NBD = m
491 # CONFIG_BLK_DEV_UB is not set
492 CONFIG_BLK_DEV_RAM = y
```

6.3.4 Makefile 文件精解

下面是drivers/net/Makefile文件的片断:

```
12 obj- $(CONFIG_ATL1) += atl1/
13 obj- $(CONFIG_GIANFAR) += gianfar_driver.o
14
15 gianfar_driver-objs := gianfar.o \
16     gianfar_ethtool.o \
17     gianfar_mii.o \
18     gianfar_sysfs.o

26 obj- $(CONFIG_PLIP) += plip.o
```

它的含义是:

(1) 第26行,如果.config文件中变量CONFIG_PLIP=y,那么将编译本目录下的plip.c文件并将其功能集成进zImage;如果.config文件中变量CONFIG_PLIP=m,那么将编译本目录下的plip.c文件生成模块plip.ko;否则,将不编译plip.c。

(2) 第13~18行,如果.config文件中变量CONFIG_GIANFAR=y,那么将编译本目录下的gianfar.c、gianfar_ethtool.c、gianfar_mii.c、gianfar_sysfs.c文件并将其功能集成进zImage;如果.config文件中变量CONFIG_GIANFAR=m,那么将编译本目录下的gianfar.c、gianfar_ethtool.c、gianfar_mii.c、gianfar_sysfs.c文件生成模块gianfar_driver.ko;否则,将不编译gianfar.c、gianfar_ethtool.c、gianfar_mii.c、gianfar_sysfs.c。

(3) 第12行,如果.config文件中变量CONFIG_ATL1=y,将递归进入本目录的子目录

atl1,并根据该子目录下的 Makefile 文件的内容决定该子目录如何进行编译;否则,将不进入本目录的子目录 atl1 进行编译。

6.3.5 实战:修改 Kconfig 和 Makefile,完成向内核中添加新的功能组件——网卡、声卡、LCD、触摸屏驱动

目前内核还不支持 dm9000 网卡的驱动,下面就来解决这个问题。

注:如果开发板是 cs8900 的网卡,请使用光盘提供的 cs8900.c 和 cs8900.h,仿照下面的步骤即可。

(1) 首先,先将 dm9000 的驱动源代码文件 mydm9000.c 和 mydm9000.h(位于光盘\work\sysbuild\目录)放到 drivers/net 目录(其实放到任何目录都可以,只是根据惯例,网络设备驱动都放置于 drivers/net)。

(2) 在 drivers/net/Kconfig 增加 config 条目,以使 mydm9000 驱动的配置选项能够出现在 make menuconfig 的配置界面中,如图 6-4 所示。

```

878 config MYDM9000
879         tristate "MYDM9000 support"
880         default y
881         depends on ARCH_S3C2410 && NET_ETHERNET
882         select CRC32
883         select MII
884         help
885         Support for dm9000 chipset on S3C2440. Added by Dennis Yang.
```

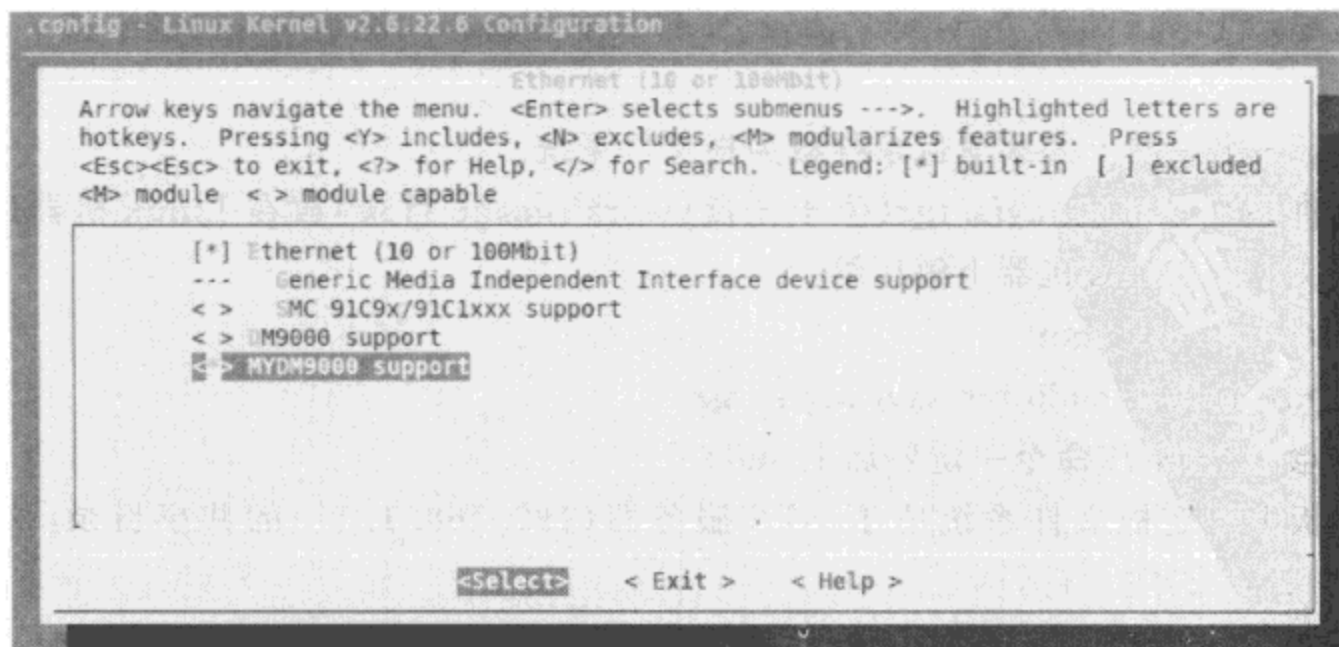


图 6-4 网卡驱动配置

(3) 在 drivers/net/Makefile(第 198 行)增加条目,以使 mydm9000 驱动能被编译进 zImage 中:

```
198 obj-$(CONFIG_MYDM9000) += mydm9000.o
```

(4) 测试新内核

① 支持 dm9000 网卡驱动

```
# ifconfig eth0 192.168.1.17
```

```
# ping 192.168.1.11
```

```
PING 192.168.1.11 (192.168.2.11): 56 data bytes
```

```
64 bytes from 192.168.1.11: seq=0 ttl=64 time=4.275 ms
```

```
64 bytes from 192.168.1.11: seq=1 ttl=64 time=3.154 ms
```

```
64 bytes from 192.168.1.11: seq=2 ttl=64 time=3.938 ms
```

```
--- 192.168.1.11 ping statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
```

```
round-trip min/avg/max = 3.154/3.789/4.275 ms
```

顺带测试“移植、裁剪及配置 Linux 内核到 s3c2440 开发板”中还未测试的内核功能。

② TCP/IP 协议栈。

③ 开发板充当 NFS 客户端。

```
# mount -t nfs -o nolock 192.168.1.11:/work/nfs_root /mnt
```

```
# ls /mnt
```

```
fs_mini fs_mini.jffs2 fs_mini3.jffs2
```

ls 时能显示 Linux 机器上/work/nfs_root 目录下的内容。说明内核已经支持了上述两项功能。

④ 将 NFS 服务器上的共享目录挂载为根文件系统。

将根文件系统压缩包 myfs.tgz(位于光盘\work\image 目录)放在 Linux 机器的/work/nfs_root 目录。在 Linux 机器上解压缩:

```
/$ cd /work/nfs_root
```

```
/work/nfs_root$ sudo tar xzvf myfs.tgz
```

特别注意:上述解压命令一定要加上 sudo

在开发板上指定根文件系统位于 NFS 服务器(192.168.1.11)的共享目录/work/nfs_root/myfs 中:

```
Dennis Yang > setenv bootargs noinitrd root = /dev/nfs nfsroot = 192.168.1.11:/work/nfs_root/
myfs ip = 192.168.1.17;192.168.1.11;192.168.1.11;255.255.255.0;www.ifl.com;eth0:off console =
ttySAC0
```



```
Dennis Yang > saveenv
Dennis Yang > boot
IP - Config: Complete;
    device = eth0, addr = 192.168.1.17, mask = 255.255.255.0, gw = 192.168.1.11,
    host = www, domain = , nis-domain = ifl.com,
    bootserver = 192.168.1.11, rootserver = 192.168.1.11, rootpath =
Looking up port of RPC 100003/2 on 192.168.1.11
Looking up port of RPC 100005/1 on 192.168.1.11
VFS: Mounted root (nfs filesystem).
```

这表明:开发板 Linux kernel 已将网络服务器上 NFS 共享文件目录作为根文件系统挂载成功,kernel 已支持了将 NFS 服务器上的共享目录挂载为根文件系统。

⑤ 动态加载和卸载模块。

⑥ 开发板充当 PC 的 U 盘。

在 Linux 机器上编译内核模块(输入命令 `make modules`),并将 `g_file_storage.ko` 复制到开发板根文件系统中:

```
/work/nfs_root$ cd /work/sysbuild/linux-2.6.22.6
/work/sysbuild/linux-2.6.22.6$ make modules
/$ mkdir /work/nfs_root/myfs/lib/modules
/$ cp /work/system/linux-2.6.22.6/drivers/usb/gadget/g_file_storage.ko /work/nfs_root/myfs
/lib/modules/
```

重启开发板,然后动态加载模块:

```
# cd /lib/modules
# insmod g_file_storage.ko file = /dev/mtdblock2 stall = 0 removable = 1
g_file_storage gadget: File-backed Storage Gadget, version: 28 November 2005
g_file_storage gadget: Number of LUNs = 1
g_file_storage gadget - lun0: ro = 0, file: /dev/mtdblock2
# usb 2-1: new high speed USB device using dummy_hcd and address 2
usb 2-1: configuration #1 chosen from 1 choice
g_file_storage gadget: high speed config #1
scsi0 : SCSI emulation for USB Mass Storage devices
```

将开发板与运行 windows 的 PC 通过 USB 线相连(PC 为主设备,开发板为从设备),将看到 PC 上产生一个可移动硬盘。

```
# rmmod g_file_storage
```

可卸载该模块。这说明内核已经支持了上述两项功能。



⑦ 支持 USB 网卡 DM9601。

未插入 USB 网卡时:

```
# ifconfig
eth0      Link encap:Ethernet HWaddr 08:00:3E:26:0A:5B
          inet addr:192.168.1.17 Bcast:192.168.1.255
          Mask:255.255.255 0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2288 errors:28 dropped:28 overruns:0 frame:0
          TX packets:1139 errors:0 dropped:0 overruns:0 carrier:0
          collisions:801 txqueuelen:1000
          RX bytes:2518046 (2.4 MiB)  TX bytes:169064 (165.1 KiB)
          Interrupt:53 Base address:0x2300
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

将 USB 网卡 DM9601 插入开发板的 usb 接口,终端界面出现:

```
# usb 1-1: new full speed USB device using s3c2410-ohci and address 7
usb 1-1: configuration #1 chosen from 1 choice
eth1: register 'dm9601' at usb - s3c24xx-1, Davicom DM9601 USB Ethernet, 00:80:14:00:02:00
```

配置 IP 地址后,再查看:

```
# ifconfig eth1 192.168.3.11
eth1: link down# ifconfig
eth0      Link encap:Ethernet HWaddr 08:00:3E:26:0A:5B
          inet addr:192.168.1.17 Bcast:192.168.1.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2343 errors:28 dropped:28 overruns:0 frame:0
          TX packets:1192 errors:0 dropped:0 overruns:0 carrier:0
          collisions:801 txqueuelen:1000
          RX bytes:2525700 (2.4 MiB)  TX bytes:176186 (172.0 KiB)
          Interrupt:53 Base address:0x2300
eth1      Link encap:Ethernet HWaddr 00:80:14:00:02:00
          inet addr:192.168.3.11 Bcast:192.168.3.255 Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
```

```

RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

这说明内核已经支持了该功能。

目前内核还不支持声卡驱动,下面就来解决这个问题。

(1) 首先,先将声卡 ual341 的驱动源代码文件 s3c_ual341.c(位于光盘\work\sysbuild 目录)放到 sound/oss 目录。

(2) 在 sound/oss/Kconfig 的最后增加 config 条目,以使 ual341 驱动的配置选项能够出现在 make menuconfig 的配置界面中,如图 6-5 所示。

```

742 config SOUND_S3C2410_UDA1341
743     tristate "S3C2410 UDA1341 support"
744     default y
745     depends on SOUND_PRIME && ARCH_S3C2410
746     help
747     oss sound driver for S3C2440, Added by Dennis Yang

```

配置界面中,单击 Device Drivers→Sound

选择 Sound card support 选项后,单击 Open Sound System,选择 Open Sound System (DEPRECATED)和 S3C2410 UDA1341 support (NEW)选项,如图 6-6 所示。

(3) 在 sound/oss/Makefile 最后(第 141 行)增加条目,以使 ual341 驱动能被编译进 zImage 中。

```
141 obj-$(CONFIG_SOUND_S3C2410_UDA1341) += s3c_ual341.o
```

(4) 由于 linux-2.6.22.6 这个版本对 s3c24XX 芯片的 DMA 驱动有一个小 bug,会导致播放歌曲时不能出声音,现在就来修复这个小 bug。

将 arch/arm/plat-s3c24xx/dma.c 的第 528 行:

```
s3c2410_dma_ctrl(chan->number, S3C2410_DMAOP_START);
```

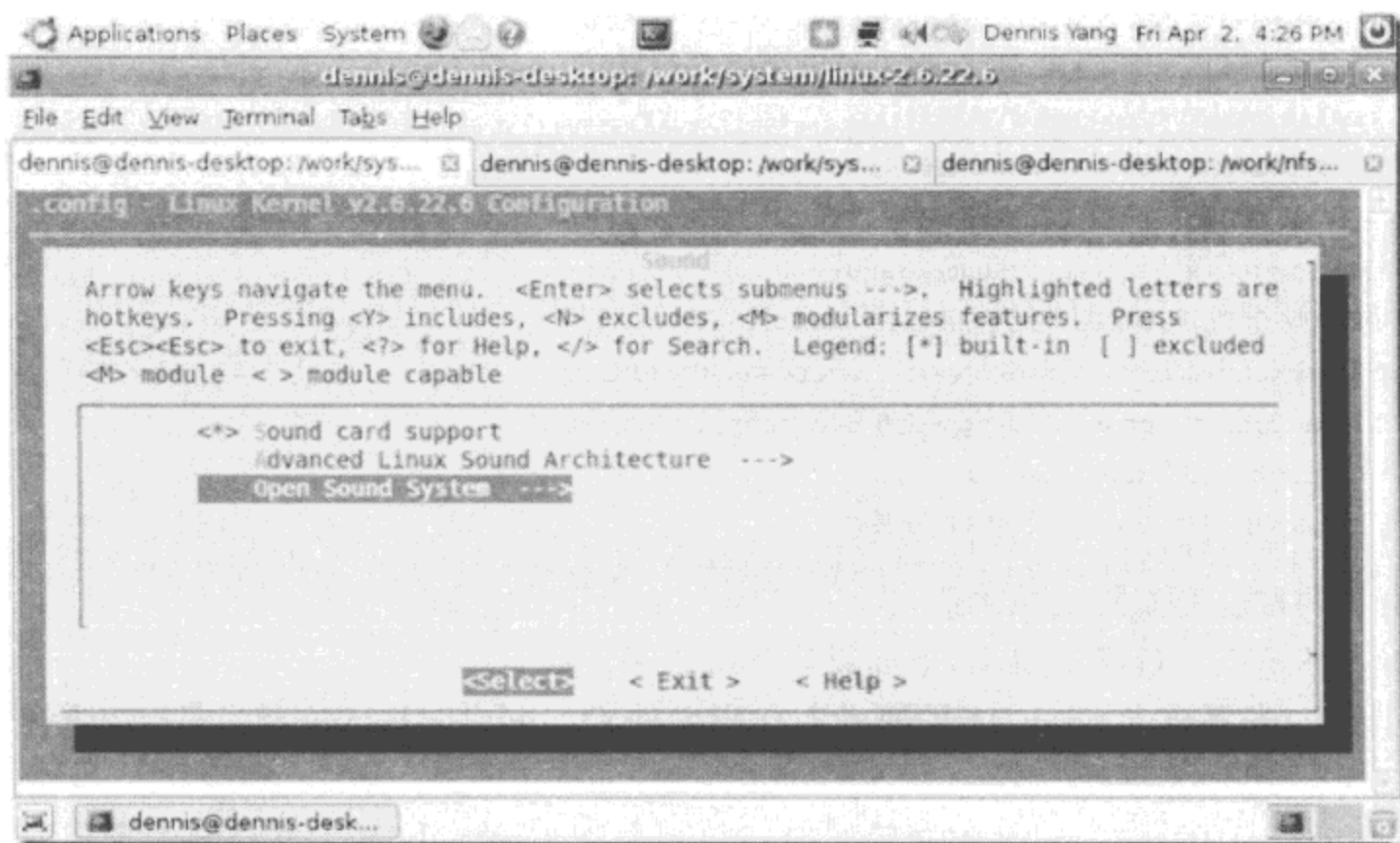


图 6-5 声卡驱动配置 1

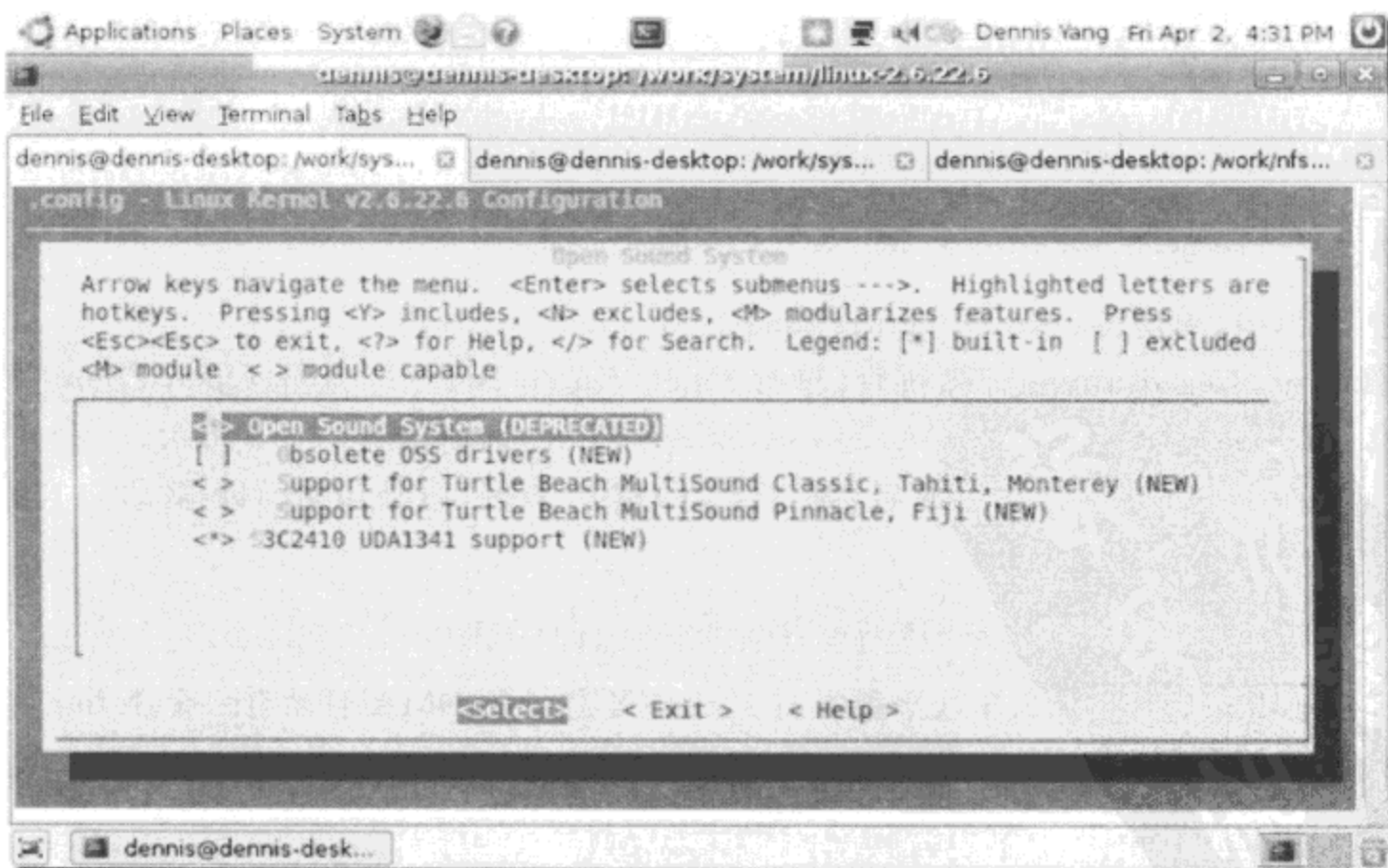


图 6-6 声卡驱动配置 2

修改为:

```
s3c2410_dma_ctrl(chan->number | DMACH_LOW_LEVEL, S3C2410_DMAOP_START);
```

(5) 测试声卡驱动。重启开发板后,运行播放器软件播放歌曲:

```
# madplay /music/pianpianxihuanni.mp3
```

MPEG Audio Decoder 0.15.2 (beta) — Copyright (C) 2000—2004 Robert Leslie et al.

Title: Track 1

Artist: 陳百強

Orchestra: 陳百強

Album: Best Memory

Track: 15

Genre: Other

目前内核还不支持 LCD 驱动,如果能在 LCD 显示屏上显示图像,的确是一件美妙的事情。下面就来解决这个问题。

(1) 首先,先将 LCD 的驱动源代码文件 mys3c2410fb.c、mys3c2410fbdata.h、regs-lcd.h (位于光盘\work\sysbuild 目录)放到 driveres/video 目录。

注:如使用的是天嵌的 tq2440,请使用 mys3c2410fbdata.h.embsky 文件替换 mys3c2410fbdata.h

(2) 在 drivers/video/Kconfig 中增加 config 条目,以使 mys3c2410fb 驱动的配置选项能够出现在 make menuconfig 的配置界面中,如图 6-7 所示。

```
1743 config MYFB_S3C2410
1744     tristate "MYS3C2410 LCD framebuffer support"
1745     depends on FB && ARCH_S3C2410
1746     select FB_CFB_FILLRECT
1747     select FB_CFB_COPYAREA
1748     select FB_CFB_IMAGEBLIT
1749     --- help ---
1750     Frame buffer driver for the built-in LCD controller in the Samsung
1751     S3C2440 processor. Added by Dennis Yang.
```

在 make menuconfig 配置界面中,单击 Device Drivers→Graphics support。

选择 MYS3C2410 LCD framebuffer support 选项。

(3) 在 drivers/video/Makefile 的第 109 行后增加一行,以使 mys3c2410fb 驱动能被编译进 zImage 中:

```
110 obj-$(CONFIG_MYFB_S3C2410) += mys3c2410fb.o
```

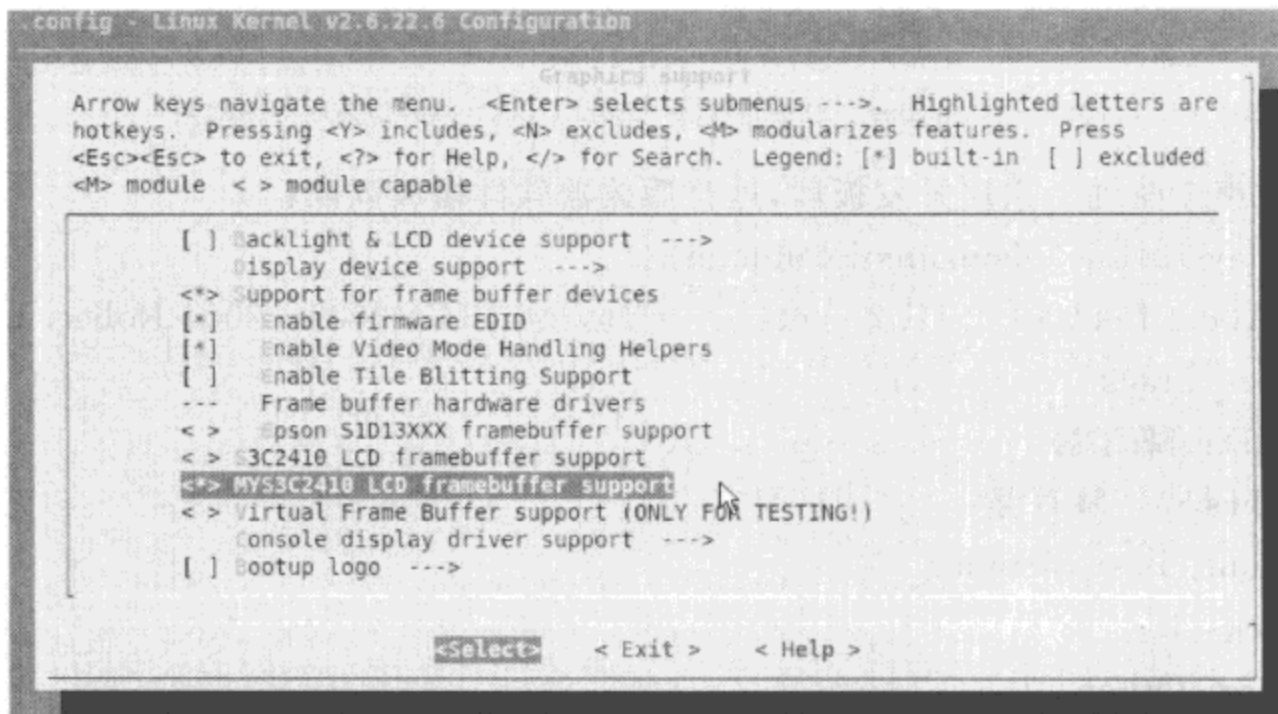


图 6-7 LCD 驱动配置

(4) 测试 LCD 驱动。

① 用 `arm - linux - gcc fbtest.c -o fbtest` 编译光盘中提供的 LCD 驱动测试程序 (`work/sysbuild\fbtest.c` 和 `fb.h`), 将生成的测试程序 `fbtest` 复制到根文件系统的 `/usr/bin` 目录

② 使用新内核重启开发板后, 可以看到在 `dev` 目录下自动生成了 `lcd` 设备文件 `fb0`

```
# ls -l /dev/fb *
```

```
crw-rw---- 1 root root 29, 0 Aug 26 15:53 /dev/fb0
```

③ 运行测试程序, 可以看到 LCD 显示屏上显示出同心圆图像

```
# fbtest /dev/fb0
```

目前的内核还不支持触摸屏驱动。下面就来解决这个问题。

(1) 首先, 先将触摸屏的驱动源代码文件 `mys3c2440ts.c` (位于光盘 `work\sysbuild` 目录) 放到 `drivers/input/touchscreen` 目录。

(2) 在 `drivers/input/touchscreen/Kconfig` 中增加 `config` 条目, 以使 `mys3c2440ts` 驱动的配置选项能够出现在 `make menuconfig` 的配置界面中, 如图 6-8 所示。

```
222 config TOUCHSCREEN_S3C2440
223     default y
224     tristate "MYS3C2440 touchscreen"
225     depends on ARCH_S3C2410
226     help
227         enable S3C2440 touchscreen, Added by Dennis Yang
```

(3) 在 `make menuconfig` 配置界面中, 单击 `Device Drivers` → `Input device support`, 选择

Event interface 和 Touchscreens 选项。

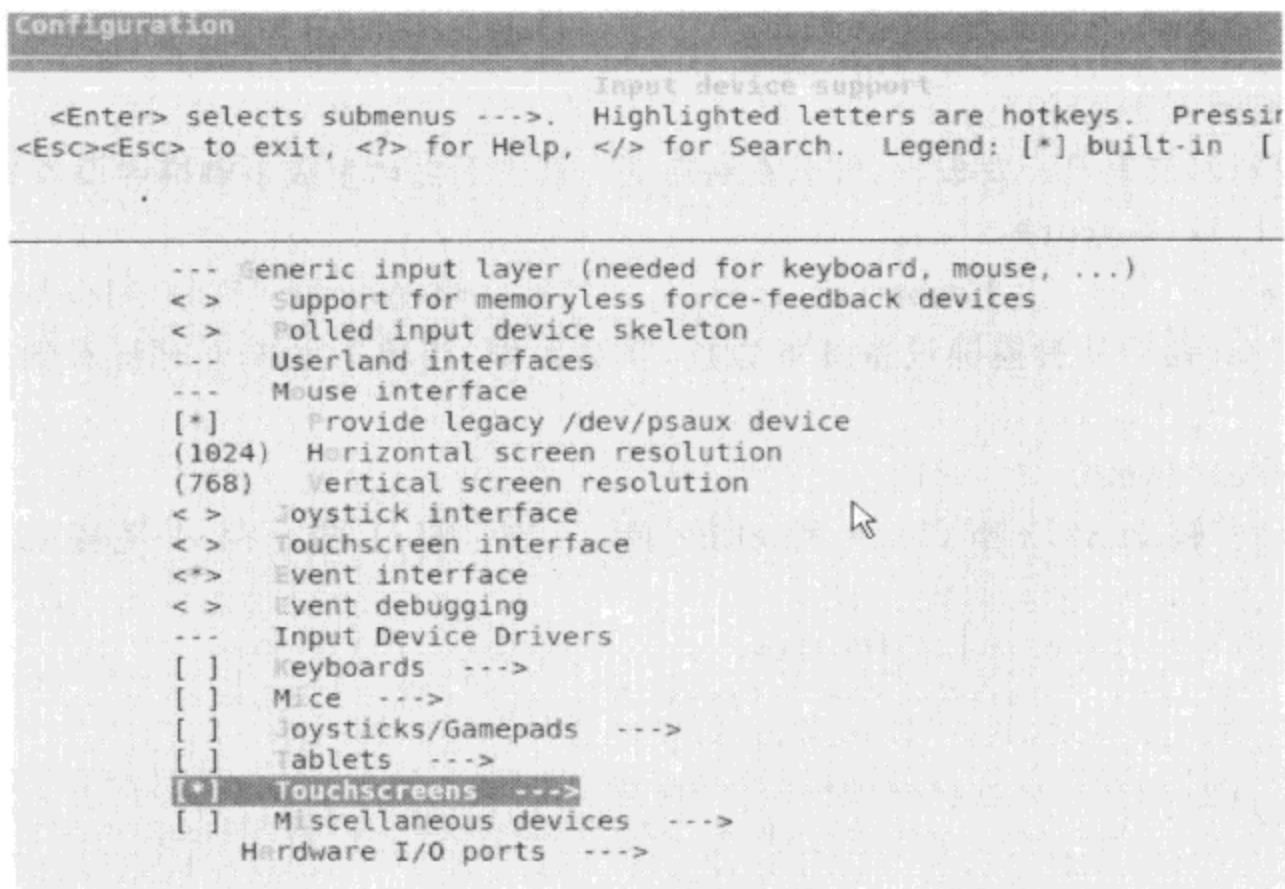


图 6-8 触摸屏驱动配置 1

(4) 继续单击 Touchscreens, 选择 MYS3C2440 touchscreen (NEW), 如图 6-9 所示。

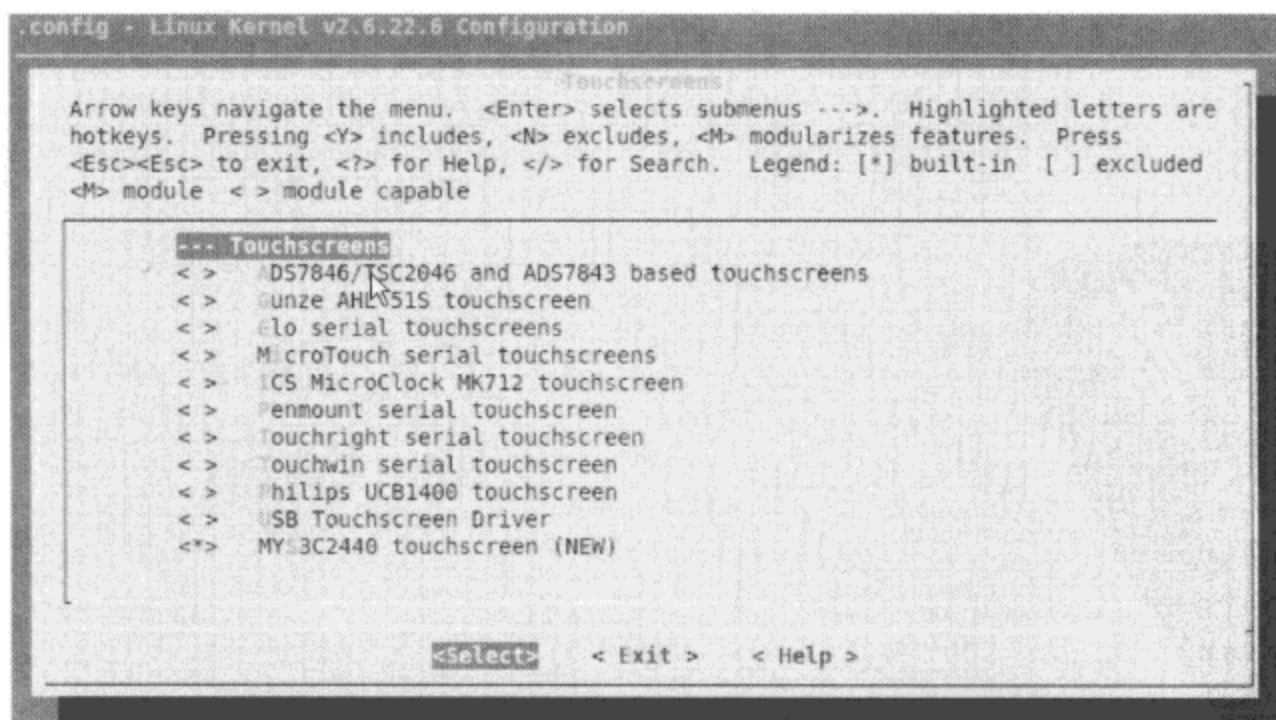


图 6-9 触摸屏驱动配置 2

(5) 在 drivers/input/touchscreen/Makefile 的最后增加一行, 以使 mys3c2440ts 驱动能

被编译进 zImage 中:

```
20 obj - $(CONFIG_TOUCHSCREEN_S3C2440) + = mys3c2440ts.o
```

(6) 测试触摸屏驱动。

① 使用新内核重启开发板后,可以看到在 dev 目录下自动生成了触摸屏设备文件 event0

```
# ls -l /dev/event *
```

```
crw-rw---- 1 root root 13, 64 Aug 26 15:53 /dev/event0
```

② 执行 cat 命令从触摸屏设备读取数据,将会阻塞,当单击触摸屏时将看到 cat 读到了数据

```
# cat /dev/event0
```

```
翱 vL#?? 翱 vL8? h 翱 vL<? 翱 vLB? 翱 vL 雏? 翱 vL 喳 g 翱 vL 芜 翱 vLF
```



第7章

建构嵌入式 Linux 文件系统

7.1 嵌入式 Linux 文件系统简介

7.1.1 嵌入式文件系统概述

1. 嵌入式 Linux 对文件系统的要求

(1) 要求文件系统在频繁的文件操作(例如新建、删除、截断)下能够保持较高的读写性能,要求低碎片化。

(2) Linux 下的日志文件系统(XFS, ReiserFS, Ext3 等)能保持数据的完整性,但消耗过多系统资源的弱点使之不能成为嵌入式系统中的主流应用。并且这些都是专门为硬盘这类的存储设备优化,对于 Flash 这类的存储介质并不适用。

(3) 嵌入式文件系统的载体是以 Flash 为主的存储介质,Flash 的擦除次数是有限的,所以为了延长 Flash 的使用寿命,应该尽量减少对 Flash 的写入操作。

(4) 嵌入式文件系统的载体是以 Flash 为主的存储介质,Flash 的擦除次数是有限的,所以为了延长 Flash 的使用寿命,应该尽量使对 Flash 的写入操作均匀分布在整个 Flash 上。

2. 嵌入式 Linux 常用的文件系统

(1) RomFS 文件系统的特点如下:

① 可以放在 ROM 空间,也可以在系统的 RAM 中。嵌入式 Linux 中常用来作根文件系统。

② Romfs 是只读的文件系统,禁止写操作,因此系统同时需要虚拟盘(RAMDISK)支持临时文件和数据文件的存储。

③ Romfs 是一种相对简单、占用空间较少的文件系统。空间的节约来自于两个方面:首先内核支持 Romfs 文件系统比支持 ext2 文件系统需要更少的代码;其次 Romfs 文件系统相对简单,在建立文件系统超级块(Superblock)需要更少的存储空间。

④ μ Clinux 系统多采用 Romfs 文件系统。

(2) CRAMFS 文件系统特点如下:

- ① CRAMFS 中的数据已被压缩,属于只读性文件系统,不能在闪存中修改。
- ② 用户想获取数据时,CRAMFS 先把数据送到 RAM 中,用户从 RAM 中读取。一般 CRAMFS 的上层为 RAMFS 文件系统,经修改过的文件都保存在 RAM 中。
- ③ RAMFS 和 CRAMFS 结合的缺陷在于,一旦出现掉电等特殊情况,保存在 RAMFS 中的修改数据将全部丢失。

(3) jffs2 文件系统特点如下:

- ① jffs2 的数据压缩方式和 CRAMFS 一样。
- ② 它允许在闪存中直接进行修改。
- ③ 其数据可存放在全部的闪存区域中,数据的写入和删除分布在很大一片区域中以防止同样的块会被重复使用。
- ④ jffs2 提供了比 Ext2 拥有更好的崩溃/掉电安全保护。
- ⑤ jffs2 是专门为 Flash 芯片那样的嵌入式设备创建的,所以它的整个设计提供了更好的闪存管理。

(4) yaffs 文件系统特点如下:

- ① 专门为 Nand Flash 设计的日志文件系统,提供磨损平衡和掉电恢复。
- ② 适合大容量的 Nand Flash。

■ jffs2 的日志通过 `jffs_node` 建立在 RAM 中,占用 RAM 空间:对于 128MB 的 Nand Flash 大概需要 4MB 的空间来维护节点。yaffs 需要的 RAM 空间要小很多。

■ jffs2 文件系统启动的时候需要扫描整个日志节点,因此对大容量 Nand Flash 而言,启动时间太长。yaffs 没有这个问题。

③ 非常适合作为嵌入式 Linux 的根文件系统。遗憾的是到目前为止,它还没有被集成进主线内核。要使用它,需要打内核补丁。

7.1.2 MTD 设备与 Flash 文件系统简介

MTD 设备与 Flash 文件系统如图 7-1 所示。

1. Flash 硬件驱动层

在 init 时驱动 Flash 硬件,Nand 型 Flash 的驱动程序位于 `/driver/mtd/nand` 子目录下。

2. MTD 原始设备

它是内核的一个功能组件,用于屏蔽下层各种不同的 Flash 硬件驱动,向上提供统一的操作 Flash 的接口。其代码由两部分组成:

- (1) MTD 原始设备的通用代码。
- (2) 各个特定 Flash 的数据,例如分区数据(`mtd_info`、`mtd_table`(`mtdcore.c`)、`mtd_part`

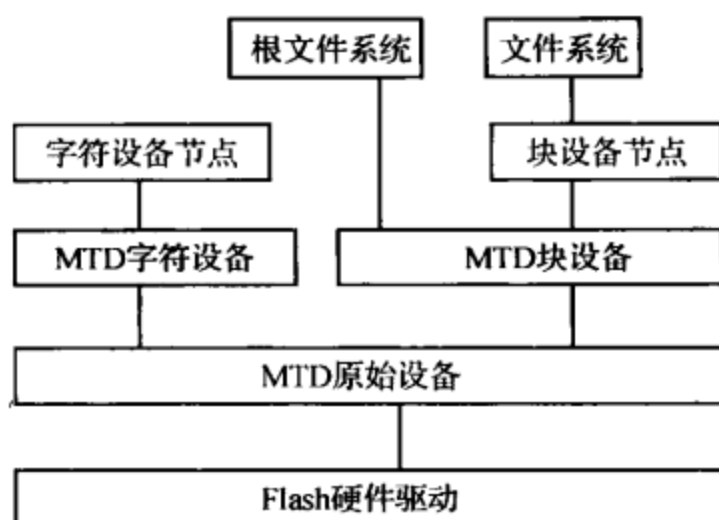


图 7-1 MTD 设备与 Flash 文件系统

(mtd_part.c))。

3. MTD 设备层

Linux 系统定义出 MTD 的块设备(主设备号 31)和字符设备(设备号 90)。通过访问此设备节点(例如 `cat /dev/mtdblock1`)就可以访问 MTD 块设备(但请注意,MTD 块设备一般不是由用户程序直接访问,而是由操作系统中的文件系统代码直接访问的。用户程序通过读写文件,借由文件系统代码间接访问 MTD 块设备)和字符设备。

```
# ls -l /dev/mtd *
crw-rw---- 1 0 0 90, 0 May 2 16:44 /dev/mtd0
crw-rw---- 1 0 0 90, 1 May 2 16:44 /dev/mtd0ro
crw-rw---- 1 0 0 90, 2 May 2 16:44 /dev/mtd1
crw-rw---- 1 0 0 90, 3 May 2 16:44 /dev/mtd1ro
crw-rw---- 1 0 0 90, 4 May 2 16:44 /dev/mtd2
crw-rw---- 1 0 0 90, 5 May 2 16:44 /dev/mtd2ro
crw-rw---- 1 0 0 90, 6 May 2 16:44 /dev/mtd3
crw-rw---- 1 0 0 90, 7 May 2 16:44 /dev/mtd3ro
brw-rw---- 1 0 0 31, 0 May 2 16:44 /dev/mtdblock0
brw-rw---- 1 0 0 31, 1 May 2 16:44 /dev/mtdblock1
brw-rw---- 1 0 0 31, 2 May 2 16:44 /dev/mtdblock2
brw-rw---- 1 0 0 31, 3 May 2 16:44 /dev/mtdblock3
```

4. 根文件系统

在 Boot Loader 中将文件系统映像烧录到 Flash 的某一个分区中,在启动的时候,将该分区(第 3 个分区)作为根文件系统(根文件系统是 jffs2)挂载。

```
Dennis Yang > printenv bootargs
```

```
bootargs = noinitrd root = /dev/mtdblock2 console = ttySAC0 rootfstype = jffs2
```

5. 文件系统

在块设备之上,就可以编写各种各样的逻辑文件系统(例如 jffs2、yaffs 等)的内核代码。内核启动后,可以通过 mount 命令和设备文件挂载相应 Flash 分区:

```
# mount -t jffs2 /dev/mtdblock1 /mnt
```

7.1.3 嵌入式 Linux 系统中的 tmpfs 文件系统

1. tmpfs 文件系统的功能

tmpfs 是基于内存的文件系统(将内存作为目录使用),它主要用于减少对闪存不必要的写操作这一唯一目的。因为 tmpfs 驻留在 RAM 中,所以写/读/擦除的操作发生在 RAM 中而不是在闪存中。因此,当将日志消息写入挂载为 tmpfs 文件系统的目录时,是将其写入 RAM 而不是闪存中,在重新引导时不会保留它们。

2. tmpfs 文件系统的优缺点

(1) 动态目录大小。目录大小可以根据被复制、创建的文件或目录的大小和数量来缩放,使得能够最理想地使用内存。并且 tmpfs 还可使用磁盘交换空间来存储,并且当为存储文件而请求页面时,使用虚拟内存(VM)子系统。

(2) 速度快。因为 tmpfs 驻留在 RAM,所以读和写几乎都是瞬时的。即使以交换的形式存储文件,I/O 操作的速度仍非常快。

(3) tmpfs 的一个缺点是当系统重新引导时会丢失所有数据。因此,重要的数据不能存储在 tmpfs 上。

tmpfs 与 ramfs 的区别:

① ramfs 实现机制是将 cache 在物理内存的文件占用的 page 不标记为可释放(freeable),这样 VM(虚拟内存管理)就不会将这些 page 释放或交换到 swap,从而实现文件总在物理内存中。

② tmpfs 也是存放于内存中,但它可以被 VM 交换到 swap。它其实是 ramfs 的一个变体。

③ 详细请参阅 Documentation/filesystems/ramfs-rootfs-initramfs.txt 和 tmpfs.txt。

3. tmpfs 在嵌入式 Linux 的用途

当 Linux 运行于嵌入式设备上时,设备就成为功能齐全的单元。许多守护进程(例如 ftpd、httpd 等)会在后台运行并生成许多日志消息;另外,所有内核日志记录机制(例如 syslogd、dmesg 和 klogd)也会在后台运行并生成许多消息。这些消息会被写入/var 和/tmp 目录下的

文件中。由于这些进程产生了大量数据,所以允许将所有这些写操作都发生在 flash 上会快速消耗 flash 的使用寿命。由于在重新引导时这些消息不需要持久存储,所以这个问题的解决方案是使用 tmpfs。

4. tmpfs 文件系统的配置位置

如图 7-2 所示,File systems→Pseudo filesystems→Virtual memory file system support (former shm fs)。

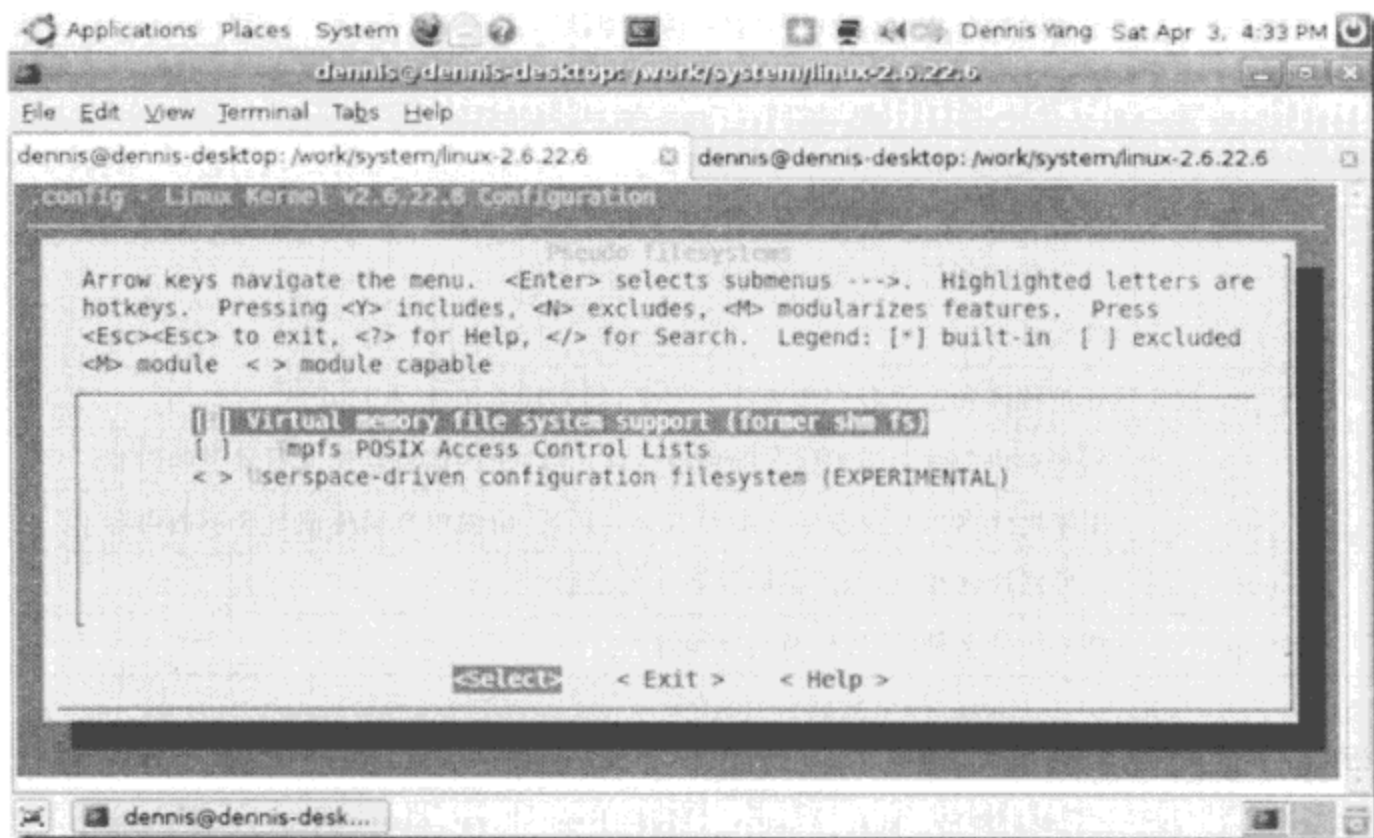


图 7-2 tmpfs 配置

7.2 详解制作根文件系统

7.2.1 FHS 标准介绍

当我们在 Linux 下输入 `ls /` 的时候,见到的目录结构以及这些目录下的内容都大同小异,这是因为所有的 Linux 发行版在对根文件系统布局上都遵循 FHS 标准的建议规定。

该标准规定了根目录下各个子目录的名称及其存放的内容,如表 7-1 所列。

表 7-1 FHS 标准目录结构

| 目录名 | 存放的内容 |
|-----------|----------------------------------|
| /bin | 必备的用户命令,例如 ls、cp 等 |
| /sbin | 必备的系统管理员命令,例如 ifconfig、reboot 等 |
| /dev | 设备文件,例如 mtblock0、tty1 等 |
| /etc | 系统配置文件,包括启动文件,例如 inittab 等 |
| /lib | 必要的链接库,例如 C 链接库、内核模块 |
| /home | 普通用户主目录 |
| /root | root 用户主目录 |
| /usr/bin | 非必备的用户程序,例如 find、du 等 |
| /usr/sbin | 非必备的管理员程序,例如 chroot、inetd 等 |
| /usr/lib | 库文件 |
| /var | 守护程序和工具程序所存放的可变,例如日志文件 |
| /proc | 用来提供内核与进程信息的虚拟文件系统,由内核自动生成目录下的内容 |
| /sys | 用来提供内核与设备信息的虚拟文件系统,由内核自动生成目录下的内容 |
| /mnt | 文件系统挂接点,用于临时安装文件系统 |
| /tmp | 临时性的文件,重启后将自动清除 |

制作根文件系统就是要建立以上的目录,并在其中建立完整目录内容。其过程大体包括:

- (1) 编译/安装 busybox,生成/bin、/sbin、/usr/bin、/usr/sbin 目录及其内容。
- (2) 利用交叉编译工具链,构建/lib 目录。
- (3) 手工构建/etc 目录。
- (4) 手工构建最简化的/dev 目录。
- (5) 创建其他空目录。
- (6) 配置系统自动生成/proc 和/sys 目录。
- (7) 利用 udev 构建完整的/dev 目录。
- (8) 制作根文件系统的 jffs2 映像文件。

下面就来详细介绍这个过程。

7.2.2 编译/安装 busybox,生成/bin、/sbin、/usr/bin、/usr/sbin 目录

这些目录下存放的主要是常用命令的二进制文件。如果要自己编写这几百个常用命令的源程序将非常困难,好在有嵌入式 Linux 系统的瑞士军刀——busybox,事情就简单多了。

- (1) 从 <http://www.busybox.net/> 下载 busybox-1.13.3.tar.bz2(光盘中已提供该文件

\work\sysbuild\busybox-1.13.3.tar.bz2), 上传到 Linux 虚拟机的 /work/sysbuild/ 目录。

(2) tar xjvf busybox-1.13.3.tar.bz2 解包。

(3) cd busybox-1.13.3 后, 执行 make defconfig 对 busybox 进行缺省配置。

(4) 执行 make menuconfig 对 busybox 进行详细配置, 如图 7-3 所列。

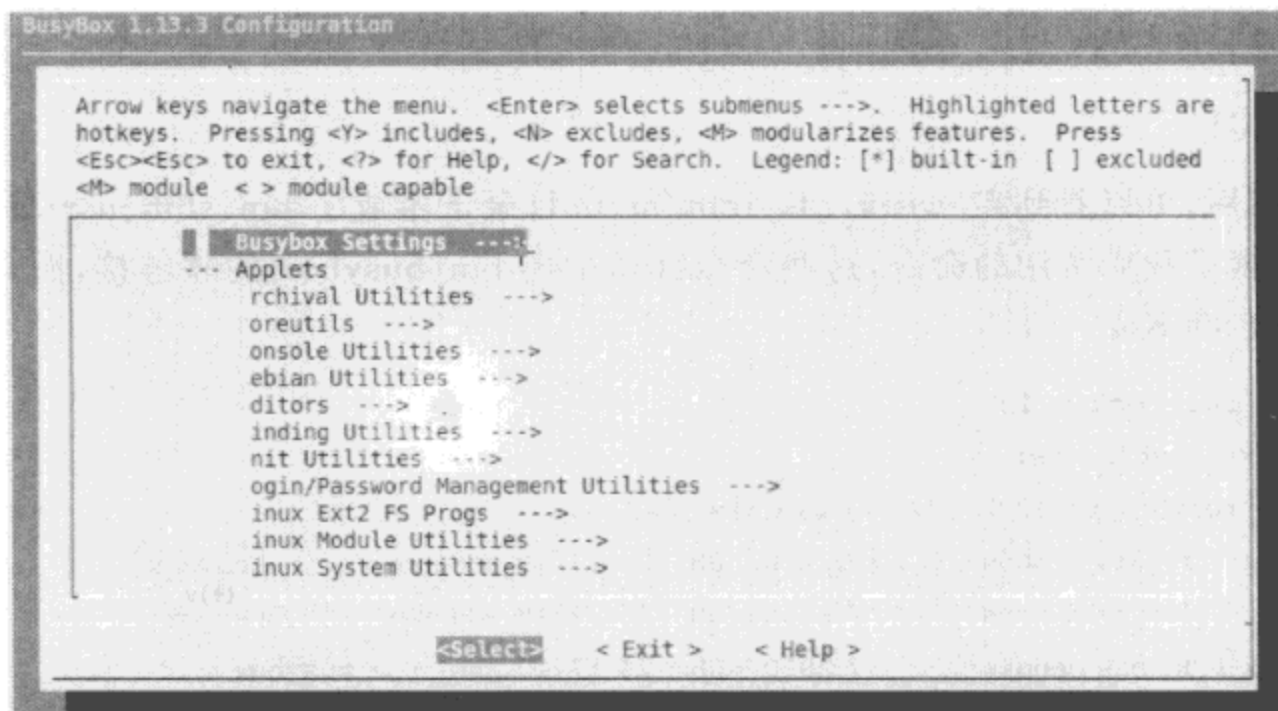


图 7-3 配置 busybox

busybox 配置主要分两部分。

第一部分是 Busybox Settings, 主要是编译和安装 busybox 的一些选项。这里主要需要配置:

① Build Options → Build BusyBox as a static binary (no shared libs), 表示编译 busybox 时, 是否静态链接 C 库。我们选择动态链接 C 库。

② Build Options → Cross Compiler prefix, 表示编译时使用的交叉编译器。我们输入 arm-linux-。

③ Installation Options → Applets links (as soft-links) → (X) as soft-links, 表示安装 busybox 时, 将各个命令安装为指向 busybox 的软链接还是硬链接。我们选择软链接。

④ Installation Options → (/work/nfs_root/myfs) BusyBox installation prefix, 表示 busybox 的安装位置。我们选择 /work/nfs_root/myfs。

⑤ Busybox Library Tuning。保留 Command line editing 以支持命令行编辑; 保留 History size 和 History saving 以支持记忆历史命令; 选中 Tab completion 和 Username completion 以支持命令自动补全。

第二部分是 Applets, 它将 busybox 的支持的几百个命令分门别类。只要在各个门类下选择想要的命令即可。这里基本保持默认设置。

去除 Linux Module Utilities 下的 Simplified modutils, 选中 insmod、rmmod、lsmod、modprobe、depmod。

(5) 编译 busybox:

```
make
```

(6) 安装 busybox:

```
make install
```

安装完成后, 可以看到在 /work/nfs_root/myfs 目录下生成了 bin、sbin、usr/bin、usr/sbin 目录, 其下包含了我们常用的命令, 这些命令都是指向 bin/busybox 的软链接, 而 busybox 本身的大小约为 800KB:

```
/work/nfs_root/myfs$ ls
bin linuxrc sbin usr
/work/nfs_root/fs_mini3$ ls -l bintotal 820
lrwxrwxrwx 1 dennis dennis      7 2010-08-26 17:10 addgroup -> busybox
lrwxrwxrwx 1 dennis dennis      7 2010-08-26 17:10 adduser -> busybox
lrwxrwxrwx 1 dennis dennis      7 2010-08-26 17:10 ash -> busybox
-rwxr-xr-x 1 dennis dennis 834912 2010-08-26 17:10 busybox
```

而普通 PC 上的 ls 命令就有差不多 110KB 的大小:

```
/work/nfs_root/myfs$ ls -l /bin/ls
-rwxr-xr-x 1 root root 112720 2009-10-06 19:07 /bin/ls
```

busybox 以它娇小的身躯容纳了数以百计的命令代码, 实在是让人佩服不已, 其不愧嵌入式系统瑞士军刀之美誉。据说, busybox 的作者身患绝症, 这更让人钦佩 GNU 开源软件的作者们, 大家积极加入开源软件队伍吧!

7.2.3 利用交叉编译工具链, 构建 /lib 目录

光有应用程序(命令)是不够的, 因为应用程序本身需要使用 C 库的库函数, 因此还必需制作 for ARM 的 C 库, 并将其放置于 /lib 目录。要自己写 C 库的源代码吗? 不用! 还记得交叉编译工具链的 3 个组成部分吗? 交叉编译器、for ARM 的 C 库和二进制工具。for ARM 的 C 库是现成的, 只需要复制过来就可以了。遗憾的是: 整个 C 库目录下的文件总大小有 29MB。而嵌入式设备的 Nand Flash 空间不会太大, 根本不会容忍一个 lib 库就占用这么大的空间。怎么办呢?

```
/work/nfs_root/myfs$ du -s --si /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
29M    /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
```


需要 C 库目录下所有的文件吗？当然不是！让我们来分析一下 glibc 库目录下 内容的组成。该目录下的子目录和文件共分 7 类：

- (1) 目标文件，如 `crt.o`，用于 `gcc` 链接可执行文件。
- (2) `libtool` 库文件（`.la`），在链接库文件时这些文件会被用到，比如它们列出了当前库文件所依赖的其他库文件，程序运行时无需这些文件。
- (3) `gconv` 目录，里面是各种链接脚本，在编译应用程序时，它们用于指定程序的运行地址、各段的位置等。
- (4) 静态库文件（`.a`），例如 `libm.a`、`libc.a`。
- (5) 动态库文件（`.so`、`.so.[0-9]*`）。
- (6) 动态链接库加载器 `ld-2.3.6.so`、`ld-linux.so.2`。
- (7) 其他目录及文件。

很显然，第 1、2、3、4、7 类文件和目录是不需要复制的。

由于动态链接的应用程序本身并不含有它所调用的 C 库函数的代码，因此执行时需要动态链接库加载器来为它加载相应的 C 库文件，所以第 6 类文件是需要复制的。

除此之外，第 5 类文件当然要复制。但第 5 类文件的大小也相当大。

```
/work/nfs_root/myfs$ du -c --si /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/
* .so *
8.3M    total
```

需要全部复制吗？非也！其实，需要哪些库完全取决于要运行的应用程序使用了哪些库函数。如果只制作最简单的系统，那么只需要运行 `busybox` 这一个应用程序即可。通过执行：

```
/work/nfs_root/myfs$ arm-linux-readelf -a bin/busybox | grep 'Shared'
0x00000001 (NEEDED)             Shared library: [libm.so.6]
0x00000001 (NEEDED)             Shared library: [libc.so.6]
```

可知：`busybox` 只用到了两个库：通用 C 库（`libc`）、数学库（`libm`），因此我们只需要复制这两个库的库文件即可。但是 每个库都有 4 个文件，4 个文件都要复制吗？ 当然不是。

```
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ ls -l libm[.-]*
-rwxr-xr-x 1 dennis dennis 779096 2008-01-22 05:31 libm-2.3.6.so
-rw-r--r-- 1 dennis dennis 1134282 2008-01-22 05:32 libm.a
lrwxrwxrwx 1 dennis dennis      9 2008-12-22 15:38 libm.so -> libm.so.6
lrwxrwxrwx 1 dennis dennis    13 2008-12-22 15:38 libm.so.6 -> libm-2.3.6.so
/usr/local/arm /gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ ls -l libc[.-]*
-rwxr-xr-x 1 dennis dennis 1435660 2008-01-22 05:48 libc-2.3.6.so
-rw-r--r-- 1 dennis dennis 2768280 2008-01-22 05:31 libc.a
```

```
-rw-r--r-- 1 dennis dennis      195 2008-01-22 05:34 libc.so
lrwxrwxrwx 1 dennis dennis      13 2008-12-22 15:38 libc.so.6 -> libc-2.3.6.so
```

● 4 个文件中的 .a 文件是静态库文件,是不需要复制的。另外 3 个文件是:

● 实际的共享链接库:libLIBRARY_NAME-GLIBC_VERSION.so。当然需要复制。

● 主修订版本的符号链接,指向实际的共享链接库:libLIBRARY_NAME.so.MAJOR_REVISION_VERSION,程序一旦链接了特定的链接库,将会参用该符号链接。程序启动时,加载器在加载程序前,会检索该文件。所以需要复制。

● 与版本无关的符号链接,指向主修订版本的符号连接(libc.so 是唯一的例外,它是一个链接命令行:libLIBRARY_NAME.so,是为编译程序时提供一个通用条目)。这些文件在程序被编译时会被用到,但在程序运行时不会被用到,所以不必复制它。

关于共享库的两个符号链接的作用的特别说明:

当我们使用 gcc hello.c -o hello -lm 编译程序时,gcc 会根据 -lm 的指示,加头(lib)添尾(.so)得到 libm.so,从而沿着与版本无关的符号链接(libm.so -> libm.so.6)找到 libm.so.6 并记录在案(hello 的 ELF 头中),表示 hello 需要使用 libm.so.6 这个库文件所代表的数学库中的库函数。而当 hello 被执行的时候,动态链接库加载器会从 hello 的 ELF 头中找到 libm.so.6 这个记录,然后沿着主修订版本的符号链接(libm.so.6 -> libm-2.3.6.so)找到实际的共享链接库 libm-2.3.6.so,从而将其与 hello 作动态链接。可见,与版本无关的符号链接是供编译器使用的,主修订版本的符号链接是供动态链接库加载器使用的,而实际的共享链接库则是供应用程序使用的。

通过以上分析,只需要复制两个库(每个库各一个主修订版本的符号链接和一个实际的共享链接库)以及动态链接库加载器(一个符号链接和一个实体文件)。步骤如下:

```
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ mkdir /work/nfs_root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libm- * /work/nfs_root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libm.so. * /work/nfs_root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libc- * /work/nfs_root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libc.so. * /work/nfs_root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P ld- * /work/nfs_root/myfs/lib
```

7.2.4 手工构建 /etc 目录

/etc 目录存放的是系统程序的主配置文件,因此需要哪些配置文件取决于要运行哪些系统程序。即使最小的系统也一定会运行 1 号用户进程 init,所以我们至少要手工编写 init 的主

配置文件 `inittab`, 该配置文件用于决定 `init` 进程要启动哪些子进程以及如何启动这些子进程。`busybox` 的 `inittab` 文件的语法、语义与传统的 `SYSV` 的 `inittab` 有所不同。

`inittab` 文件中每个条目用来定义一个需要 `init` 启动的子进程, 并确定它的启动方式, 格式为 `<id>:<runlevel>:<action>:<process>`。例如: `ttySAC0::askfirst:-/bin/sh`

- `<id>` 表示子进程要使用的控制台, 若省略则使用与 `init` 进程一样的控制台。
- `<runlevel>` 表示运行级别, `busybox init` 程序这个字段没有意义。
- `<action>` 表示 `init` 进程如何控制这个子进程。
 - `sysinit`: 以该方式启动的子进程最先被 `init` 启动, 该子进程只会被启动一次, 如该子进程结束, `init` 将不会重新启动它(请与 `respawn` 对照查看)。 `init` 进程必须等待该子进程结束后才能继续执行启动其他子进程的动作。
 - `wait`: 系统执行完 `sysinit` 条目后才启动该子进程, 该子进程只执行一次, `init` 进程必须等待该子进程结束后才能继续执行启动其他子进程的动作。
 - `once`: 系统执行完 `wait` 条目后才启动该子进程, 该子进程只执行一次, `init` 进程不必等待该子进程的结束就可以执行启动其他子进程的动作。
 - `respawn`: 系统执行完 `once` 条目后才启动该子进程, `init` 进程会持续监测该子进程的状态, 若发现该子进程退出, 会重新启动它。
 - `askfirst`: 系统启动完 `respawn` 条目后才启动该子进程, 与 `respawn` 类似, 不过 `init` 进程先输出 `Please press Enter to activate this console`, 等用户按 `Enter` 后才启动子进程。
 - `shutdown`: 当系统关机时启动该子进程。
 - `restart`: `Busybox` 中配置了 `CONFIG_FEATURE_USE_INITAB`, 并且 `init` 进程接收到 `SIGUP` 信号时执行, 先重新读取、解析 `/etc/inittab` 文件, 再执行 `restart` 程序。
 - `ctrlaltdel`: 按 `Ctrl+Alt+Del` 键时启动该子进程。不过在串口控制台中无法输入它。
- `<process>` 表示进程对应的二进制文件。如果前面有一号, 表示该程序是“可以与用户进行交互的”。

制作最简单的 `/etc/inittab` 文件, 其内容如下:

```

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

```

制作最简单的脚本程序文件 `/etc/init.d/rcS`, 其内容如下:

```

#!/bin/sh
ifconfig eth0 192.168.1.17

```

修改 shell 脚本文件 `etc/init.d/rcS` 的权限, 以使其可被执行:

```
/work/nfs_root/myfs/etc/init.d$ chmod a+x rcS
```

7.2.5 手工构建最简化的 /dev 目录

在 Linux 机器上, 执行 `ls /dev` 可看到几百个设备文件, 需要手工创建它们吗? 只需要手工创建几个设备文件! 我怎么知道我应该创建哪几个设备文件呢? 先看看开发板上 Linux 的反应再说。

启动 Linux 操作系统, 显示如下:

```
VFS: Mounted root (nfs filesystem).
```

```
Freeing init memory: 112K
```

```
Warning: unable to open an initial console.
```

这说明, 内核已经成功挂载根文件系统, 但却未能成功启动第一个用户进程 `init`。通过错误消息 `unable to open an initial console` 搜索内核源代码, 找到 `init/main.c` 文件。

```
748 static int noinline init_post(void)
749 {
750     free_initmem();
751     unlock_kernel();
752     mark_rodata_ro();
753     system_state = SYSTEM_RUNNING;
754     numa_default_policy();
755
756     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
757         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
758
759     (void) sys_dup(0);
760     (void) sys_dup(0);
761
762     if (ramdisk_execute_command) {
763         run_init_process(ramdisk_execute_command);
764         printk(KERN_WARNING "Failed to execute %s\n",
765                ramdisk_execute_command);
766     }
767
768     /*
769     * We try each of these until one succeeds.
770     *
771     * The Bourne shell can be used instead of init if we are
```

```

772  * trying to recover a really broken machine.
773  * /
774  if (execute_command) {
775      run_init_process(execute_command);
776      printk(KERN_WARNING "Failed to execute %s.  Attempting "
777                  "defaults...\n", execute_command);
778  }
779  run_init_process("/sbin/init");
780  run_init_process("/etc/init");
781  run_init_process("/bin/init");
782  run_init_process("/bin/sh");
783
784  panic("No init found.  Try passing init = option to kernel.");
785 }

```

显然,内核错误是由 757 行不能打开/dev/console 所致。通过查看已经安装好的 Linux 机器的/dev/console 设备文件,可知其是字符设备文件,主设备号为 5,次设备号为 1:

```

/work/nfs_root/myfs/etc$ ls -l /dev/console
crw-r--r-- 1 root root 5, 1 2010-04-08 08:40 /dev/console

```

因此,使用下面的命令创建它:

```

/work/nfs_root/myfs/dev$ sudo mknod console c 5 1

```

还需要创建其他设备文件吗? 再看看 Linux 的反应。

再次重启开发板上的 Linux,显示

```

VFS: Mounted root (nfs filesystem).
Freeing init memory: 112K
Please press Enter to activate this console.
#

```

7.2.6 使用启动脚本完成 /proc、/sys、/dev、/tmp、/var 等目录的完整构建

```

/work/nfs_root/myfs $ mkdir home root proc sys tmp mnt var

```

再次重启开发板上的 Linux 时,似乎有些问题。

```

VFS: Mounted root (nfs filesystem).
Freeing init memory: 112KB
Please press Enter to activate this console.

```



```
# ps
PID USER      VSZ STAT COMMAND
```

ps 竟然看不到任何进程的存在！ps 的机制是通过查看 /proc 中的内容来获得进程信息的。那么，目前 /proc 里有哪些内容呢？

```
# ls /proc
#
```

竟然什么也没有！（这可如何是好？）

其实 /proc 是用来提供内核与进程信息的虚拟文件系统，由内核自动生成目录下的内容。不过需要设置一下，将 /etc/init.d/rcS 修改为：

```
#! /bin/sh
ifconfig eth0 192.168.1.17
mount -t proc none /proc
```

对于 mount -t proc none /proc 的解释：通常情况下 mount 命令应该写为 mount -t ext2 /dev/hdb1 /proc。但由于现在挂载的 /proc 是虚拟文件系统，它不与任何物理硬盘分区相对应，因此在表示物理硬盘分区的位置用占位符 none 来表示。

重启开发板上的 Linux，显示成功了：

Please press Enter to activate this console.

```
# ps
PID USER      VSZ STAT COMMAND
  1 0          2960 S    init
  2 0           0 SW<   [kthreadd]
  3 0           0 SWN   [ksoftirqd/0]
  4 0           0 SW<   [events/0]
  5 0           0 SW<   [khelper]
 41 0           0 SW<   [kblockd/0]
 42 0           0 SW<   [ksuspend_usbd]
 45 0           0 SW<   [khubd]
 47 0           0 SW<   [kseriod]
 59 0           0 SW    [pdflush]
 60 0           0 SW    [pdflush]
 61 0           0 SW<   [kswapd0]
 62 0           0 SW<   [aio/0]
179 0           0 SW<   [mtdblockd]
231 0           0 SW<   [rpciod/0]
236 0          2964 S    - /bin/sh
```



237 0 2964 R ps

插入 U 盘, 内核显示识别到了 U 盘:

```
# usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: not running at top speed; connect to a high speed hub
usb 1-1: configuration #1 chosen from 1 choice
scsi0 : SCSI emulation for USB Mass Storage devices
scsi 0:0:0:0: Direct-Access      SanDisk    Cruzer                      8.02 PQ: 0 ANSI: 0 CCS
sd 0:0:0:0: [sda] 7856127 512-byte hardware sectors (4022 MB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] 7856127 512-byte hardware sectors (4022 MB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sda: sda1
sd 0:0:0:0: [sda] Attached SCSI removable disk
```

但当要使用的时候, 却找不到设备文件:

```
# mount /dev/sda1 /mnt
mount: mounting /dev/sda1 on /mnt failed: No such file or directory
# ls /dev/sda1
ls: /dev/sda1: No such file or directory
```

/dev 目录下只有一个设备文件。

```
# ls /dev
console
```

在虚拟机 Linux 操作系统下, 执行 `ls /dev` 可看到几百个设备文件, 难道要查看这几百个设备文件的主、次设备号, 然后再手工使用 `mknod` 命令来生成这几百个设备文件吗? 其实构建 `/dev` 目录有 3 种方法:

- 创建静态设备文件: 需要使用 `mknod` 命令事先创建很多设备文件, 麻烦大了。
- 使用 `devfs`: 使用上存在一些问题, 在 2.6.13 内核版本后已被废弃。
- 使用 `udev` (user dev), `mdev` 是 `busybox` 中对 `udev` 的简化实现。我们采用该方法。

`mdev` 的原理是: 操作系统启动的时候会将识别到的所有设备的信息自动导出到 `/sys` 目录。在此基础上, 用户态的应用程序 `mdev -s` 就可以扫描 `/sys/class` 和 `/sys/block` 中所有的类设备目录, 如果在目录中含有名为 `dev` 的文件, 且文件中包含的是设备号, 则 `mdev` 就利用这些信息为这个设备在 `/dev` 下创建设备节点文件。

因此我们要做的就是: 配置自动生成 `/sys` 目录下的内容并调用 `mdev`。

```
# ls /sys
# mount -t sysfs none /sys
# ls /sys
block      class      firmware  kernel     power
bus        devices   fs         module
# cat /sys/block/sda/dev
8:0
# ls /dev
console
# mdev -s
# ls -l dev | wc -l
140
# ls /dev/sda* -l
brw-rw---- 1 0      0          8,  0 Aug 26 11:51 /dev/sda
brw-rw---- 1 0      0          8,  1 Aug 26 11:51 /dev/sda1
```

可是,当我们将 U 盘拔出后,发现/dev/sda1 并不自动消失;手工删除/dev/sda1 后,再重新插入 U 盘,/dev/sda1 也不会自动生成。怎么办呢?

先让我们来了解一下 Linux 系统下实现热插拔的机制:当有热插拔事件产生时,内核就会调用位于/sbin 目录的 mdev。这时 mdev 通过环境变量中的 ACTION 和 DEVPATH(这两个变量是系统自带的)来确定此次热插拔事件的动作以及影响了/sys 中的哪个目录。接着会看看这个目录中是否有 dev 的属性文件,如果有就利用这些信息为这个设备在/dev 下创建或删除设备节点文件。

由此可知:需告知操作系统,当它发现热插拔事件时应调用 mdev,而不是别的程序。

```
# echo /sbin/mdev > /proc/sys/kernel/hotplug
```

将上述工作放到 rcS 中:

```
#!/bin/sh
ifconfig eth0 192.168.1.17
mount -t proc none /proc
mount -t sysfs none /sys
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

似乎我们的根文件系统已经相当完善了。但仔细想一想 Nand Flash 的擦写寿命是有限的这个事实,我们就应该明白,还需将/dev、/tmp、/var 三个目录挂载为 tmpfs 文件系统。修改 rcS 如下:

```
#!/bin/sh
```



```

ifconfig eth0 192.168.1.17
mount -t proc none /proc
mount -t sysfs none /sys
mount -t tmpfs none /dev
mount -t tmpfs none /var
mount -t tmpfs none /tmp
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s

```

到此为止,我们的根文件系统基本定型,但还有一个问题需要考虑。现在处在产品研发阶段,所以根文件系统采用 NFS 挂载以方便开发,由于在 u-boot 中我们指定了开发板的 IP 地址为 192.168.1.17,所以我們必須在 rcS 中指定相同的 IP 地址,否則在启动过程中,会因为 IP 地址的缘故,在执行 rcS 后,导致开发板无法和 NFS 服务器通信,也就无法进入 shell。但在最终产品的根文件系统中,也许要求开发板的 IP 地址是别的 IP,而不是 192.168.1.17。应该如何做,才能做出一个即可以用于开发阶段,又可以用于最终产品的统一的根文件系统呢?

答案是:查看 /proc/mounts。

因为如果 kernel 是通过 NFS 挂载根文件系统的话,会在虚拟文件 /proc/mounts 中产生如下下一行:

```
/dev/root / nfs rw,vers = 2, rsize = 4096, wsize = 4096, hard, nolock, proto = udp, addr = 192.168.1.11
```

从而可以据此判断是否是 NFS 挂载。是,则在 rcS 中不设置 IP 地址;不是,则在 rcS 中设置最终产品需要的 IP 地址。因此将 rcS 改写为如下:

```

#! /bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t tmpfs none /dev
mount -t tmpfs none /var
mount -t tmpfs none /tmp
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
/sbin/ifconfig lo 127.0.0.1
/etc/init.d/ifconfig-eth0

```

在 /etc/init.d/ 下新建脚本文件 ifconfig-eth0 如下:

```

1 #! /bin/sh
2 echo -n Try to bring eth0 interface up.....
3 if [ -f /etc/eth0 - setting ]; then
4     source /etc/eth0 - setting

```



```

5      if grep -q "~dev/root / nfs " /proc/mounts ; then
6          echo -n NFS root
7      else
8          ifconfig eth0 down
9          ifconfig eth0 hw ether $ MAC
10         ifconfig eth0 $ IP netmask $ Mask up
11         route add default gw $ Gateway
12     fi
13     echo nameserver $ DNS > /etc/resolv.conf
14 else
15     if grep -q "~dev/root / nfs " /proc/mounts ; then
16         echo -n NFS root ...
17     else
18         /sbin/ifconfig eth0 192.168.1.222 netmask 255.255.255.0 up
19     fi
20 fi
21 echo Done

```

其中,18行的192.168.1.222就是最终产品的IP地址。这样做的好处还在于,在最终产品运行期间,用户还可以通过修改配置文件/etc/eth0-setting,来定制IP地址、子网掩码、MAC地址、网关地址、DNS服务器地址。

一个eth0-setting文件范例如下:

```

IP = 192.168.1.223
Mask = 255.255.255.0
Gateway = 192.168.1.1
DNS = 192.168.1.1
MAC = 08:20:90:50:90:50

```

cut X

7.2.7 制作根文件系统的 jffs2 映像文件

根文件系统已经制作完毕,最后一个步骤是将其打包为 jffs2 映像文件,以供 bootloader 将其烧录到 Nand Flash 上。这只需要执行命令:

```
/work/nfs_root$ sudo mkfs.jffs2 -n -s 2048 -e 128KiB -d myfs -o myfs.jffs2
```

这其中:

- -n 表示不要在每个擦除块上都加上清除标记。
- -s 2048 指明一页大小为 2048 字节。
- -e 128KiB 指明一个擦除块大小为 128KB。
- -d fs_mini3 指明要打包的目录。

- -o fs_mini3.jffs2 指明最终的映像文件名。

以上命令是针对大页 Nand Flash 的。如果很不幸,开发板的 Nand Flash 总容量为 64MB 或更小,则一般而言,Nand Flash 是小页的。此时,应当使用如下的命令:

```
/work/nfs_root$ mkfs.jffs2 -n -s 512 -e 16KiB -d myfs -o myfs.jffs2
```

7.3 建构嵌入式 Linux 应用程序系统

目前我们得到了一个光秃秃的根文件系统,几乎没有任何实际用途。这就好比 iPhone 手机上面几乎没有安装多少应用程序,使用这样的手机是不是非常无趣呢? 所以,我们必须为嵌入式 Linux 设备配上功能强大的应用系统。

7.3.1 辅助处理工具的移植

在某些情况下,在嵌入式设备上可能需要处理 ext2、fat 文件系统分区,或者 mtdblock 分区。这就需要相应的文件系统工具,这些工具 busybox 中并不包含,需要通过 tar 包进行交叉编译得到。

1. 移植 ext2 文件系统所需工具

(1) 可以到 <http://sourceforge.net/projects/e2fsprogs/> 下载 e2fsprogs 源码(光盘\work\sysbuild\e2fsprogs-1.40.2.tar.gz),放到虚拟机 Linux 系统的/work/sysbuild 目录。

(2) 解压、编译以及安装:

- cd /work/sysbuild/
- tar xzvf e2fsprogs-1.40.2.tar.gz
- cd e2fsprogs-1.40.2
- ./configure --with-cc=arm-linux-gcc --with-linker=arm-linux-ld --enable-elf-shlibs --host=arm --prefix=/work/sysbuild/e2fsprogs-1.40.2/result
- make
- make install-libs
- make install

对上述命令的解释:

采用 tar 包编译应用,一般有 3 个步骤:configure 用于配置软件以生成 Makefile 文件;make 编译软件;make install 安装软件。其中最重要的步骤是 configure。configure 的参数因软件不同而不同,但通常需要--prefix 参数指定安装目录。对于交叉编译,通常还需要--host 参数指定软件将来会运行在何种主机平台(本软件是 ARM)之上,对于 ARM 而言,一般其值为 arm-linux 或 arm。此外可能还需指定交叉编译器,如上的--with-cc=arm-linux-gcc。如果需要编译出动态链接库,还需指定相应参数,如上的--enable-elf-shlibs。

(3) 将/work/sysbuild/e2fsprogs - 1.40.2/result/sbin/mke2fs 复制至根文件系统的 usr/sbin 目录:

```
/work/sysbuild/e2fsprogs - 1.40.2/result/sbin$ cp mke2fs /work/nfs_root/myfs/usr/sbin
```

(4) 将 mke2fs 要用到的 5 个相关库文件(位于 result/lib 目录)复制至根文件系统的 usr/lib 目录,并创建相应软链接文件:

```
/work/tools/e2fsprogs - 1.40.2$ mkdir
/work/nfs_root/myfs/usr/lib
/work/tools/e2fsprogs - 1.40.2$ cd result/lib
/work/tools/e2fsprogs - 1.40.2/result/lib$ cp libblkid.so.1.0 libcom_err.so.2.1 libe2p.so.2.3
libext2fs.so.2.4 libuuid.so.1.2
/work/nfs_root/myfs/usr/lib
/work/tools/e2fsprogs - 1.40.2/result/lib$ cd
/work/nfs_root/myfs/usr/lib
/work/nfs_root/myfs/usr/lib$ ln -s libblkid.so.1.0 libblkid.so.1
/work/nfs_root/myfs/usr/lib$ ln -s libcom_err.so.2.1 libcom_err.so.2
/work/nfs_root/myfs/usr/lib$ ln -s libe2p.so.2.3 libe2p.so.2
/work/nfs_root/myfs/usr/lib$ ln -s libext2fs.so.2.4 libext2fs.so.2
/work/nfs_root/myfs/usr/lib$ ln -s libuuid.so.1.2 libuuid.so.1
```

(5) 测试 ext2 文件系统工具

启动开发板 Linux 后:

- 必要时,使用 fdisk /dev/sda 分区。
- mke2fs /dev/sda1。
- mount -t ext2 /dev/sda1 /mnt。

(6) 由于将来我们在编译 qtopia 时,需要链接 libuuid 库。因此现在将 libuuid 库的头文件和库文件复制到交叉编译工具链的 C 库中,并创建库文件相应的两个软链接。

```
/work/sysbuild/e2fsprogs - 1.40.2$ cp -R result/include/uuid
/usr/local/arm/gcc - 3.4.5 - glibc - 2.3.6/arm - linux/include/
/work/sysbuild/e2fsprogs - 1.40.2$ cp result/lib/libuuid.so.1.2
/usr/local/arm/gcc - 3.4.5 - glibc - 2.3.6/arm - linux/lib
/work/sysbuild/e2fsprogs - 1.40.2$ cd /usr/local/arm/gcc - 3.4.5 - glibc - 2.3.6/arm -
linux/lib
/usr/local/arm/gcc - 3.4.5 - glibc - 2.3.6/arm - linux/lib$ ln -s libuuid.so.1.2 libuuid.so.1
/usr/local/arm/gcc - 3.4.5 - glibc - 2.3.6/arm - linux/lib$ ln -s libuuid.so.1 libuuid.so
```

2. 移植 dos 文件系统所需工具

(1) 可以到 <http://ftp.debian.org/debian/pool/main/d/dosfstools/> 下载 dosfstools 源

码(光盘\work\sysbuild\dosfstools_2.11.orig.tar.gz),放到虚拟机 Linux 系统的/work/sysbuild 目录。

(2) 解压、编译:

- cd /work/sysbuild/
- tar xzvf dosfstools-2.11.orig.tar.gz
- cd dosfstools-2.11
- 修改 Makefile,将 CC=gcc 改为 CC=arm-linux-gcc
- make

特别说明:

此处由于 Makefile 已经存在,所以没有使用 configure 命令,但由于是交叉编译,因此仍然需要指定交叉编译器,这里采用直接修改 Makefile 中 CC 变量的方式指定交叉编译器。

很多的交叉编译移植所要修改的环境变量是:

CC 编译器,系统默认为 gcc,需要修改为 arm-linux-gcc

AR 库工具,用以创建和修改库,需要修改为 arm-linux-ar

LD 链接器,系统默认为 LD,需要修改为 arm-linux-ld

RANLIB 随机库创建器,系统默认为 ranlib,需要修改为 arm-linux-ranlib

AS 汇编器,系统默认为 as,需要修改为 arm-linux-as

NM 库查看工具,系统默认为 nm,需要修改为 arm-linux-nm

还有一些不常用的其他环境变量,在此就不一一列举了。需要注意的是,并不是每个移植都需要做全面的环境变量修改,有些是不需要改的,这要根据实际情况,也就是系统提示信息来调整。

(3) 将./mkdosfs/mkdosfs 复制至根文件系统的 usr/bin 目录/work/sysbuild/dosfstools-2.11 \$ cp ./mkdosfs/mkdosfs /work/nfs_root/myfs/usr/bin。

(4) 测试 dos 文件系统工具。

启动开发板 Linux:

- 必要情况下,使用 fdisk /dev/sda 分区。
- mkdosfs -F 32 /dev/sda1。
- mount -t msdos /dev/sda1 /mnt。

3. 移植 mtd 工具程序

(1) 由于 mtd 工具程序会用到 zlib 库,因此先移植 zlib 库:

- cd /work/sysbuild/。
- tar xzvf zlib-1.2.3.tar.gz。
- cd zlib-1.2.3。
- ./configure --shared --prefix=/work/sysbuild/zlib-1.2.3/result。注意此处生成动



态链接库使用的参数是`-shared`,对比“移植 ext2 文件系统所需工具”指定的参数,可知:即使是相同作用,不同软件使用的 `configure` 参数也是不同的。不过一般都可以通过 `./configure --help` 获得该软件的 `configure` 所能使用的参数说明。

- 修改 Makefile,将 19、28、29、36、37 全部加上 `arm-linux` 这个前缀,例如:19 行原为 `CC=gcc`,改为 `CC=arm-linux-gcc`。

- `make`。
- `make install`。

(2) 移植 mtd 工具程序:

- `cd /work/sysbuild/`。
- `tar xjvf mtd-utils-05.07.23.tar.bz2`。
- `cd mtd-utils-05.07.23/util`。

- 修改 Makefile:

■ 增加第 5 行 `DESTDIR=/work/sysbuild/mtd-utils-05.07.23/util/result`。

■ 第 9 行改为 `CROSS=arm-linux-`,此处直接修改变量指定交叉编译器。

■ 第 11 行改为 `CFLAGS := -I../include -I/work/sysbuild/zlib-1.2.3/result/include -O2-Wall`,此处 `-I` 选项指定编译过程中查找 `zlib` 库的头文件的位置。

■ 增加第 12 行 `LDFLAGS := -L/work/sysbuild/zlib-1.2.3/result/lib`,此处 `-L` 选项指定编译过程中查找 `zlib` 库的库文件的位置。

■ 第 63 行改为 `install -m0755 ${TARGETS} ${DESTDIR}/${SBINDIR}/`。

■ 第 67 行改为 `install -m0644 ../include/mtd/*.h ${DESTDIR}/${INCLUDEDIR}/mtd/`

- `make`。
- `make install`。

(3) 将 `result` 目录下的 `mtd` 工具程序复制到根文件系统的 `usr/bin` 目录:

```
/work/sysbuild/mtd-utils-05.07.23/util $ cp result/usr/sbin/* /work/nfs_root/myfs/usr/bin
```

(4) 将 `mtd` 工具程序所需的 `zlib` 库复制到根文件系统的 `usr/lib` 目录:

```
/work/tools/mtd-utils-05.07.23/util $ cd /work/sysbuild/zlib-1.2.3/result/lib/
/work/sysbuild/zlib-1.2.3/result/lib $ cp -P libz.so.1* /work/nfs_root/myfs/usr/lib
```

(5) 测试 `mtd` 工具程序。启动开发板 Linux:

- 可以运行 `mkfs.jffs2` 在开发板上制作 `jffs2` 文件映像。
- 可以运行 `nandwrite` 将 `jffs2` 或 `yaffs` 文件映像写入 Nand Flash 的某个分区。
- 可以运行 `nanddump` 将 Nand Flash 的某个分区的内容制作为文件映像。

4. strace 的移植

在以后的驱动开发的学习中, strace 是调测驱动程序的手段之一, 但 busybox 中不含有 strace, 因此需要从 tar 包进行交叉编译而得。

先从 strace 的官方网站 <http://sourceforge.net/projects/strace/> 下载(光盘\work\sysbuild\strace-4.5.20.tar.bz2), 将 strace-4.5.20.tar.bz2 放到虚拟机 Linux 系统的 /work/sysbuild 目录。然后执行以下命令:

- cd /work/sysbuild。
- tar xjvf strace-4.5.20.tar.bz2。
- cd strace-4.5.20。
- ./configure --prefix=/work/sysbuild/strace-4.5.20/result --host=arm-linux。
- make。
- make install。
- cp result/bin/strace /work/nfs_root/myfs/usr/bin/。

7.3.2 MP3 播放器 madplay 的移植

在测试声卡驱动时, 使用的播放器就是 madplay, 本节介绍一下该播放器的详细移植过程。

目前 madplay 的官方网站是 <http://www.underbit.com/products/mad/>, 透过该网站的介绍可以得知, 它还需要 libmad 和 libid3tag 两个库, 从该网站找到下载连接 http://sourceforge.net/project/showfiles.php?group_id=12349

这样就得到了移植 madplay 所需要的关键的 3 个文件(光盘\work\sysbuild 目录中已提供):

- madplay-0.15.2b.tar.gz;
- libmad-0.15.1b.tar.gz;
- libid3tag-0.15.1b.tar.gz。

它还会用到其他文件吗? 一般都会遇到一些小麻烦, 让我们继续吧。

1. 准备目录及文件

(1) 在 /work 目录下建立工作目录 madplay, 在该目录下创建 3 个子目录: tarball 用于存放源码包, src-arm 用于存放解包后的源代码, target-arm 用于安装编译成功的内容。

(2) 将上述 3 个源码包复制到 /work/madplay/tarball 目录。

(3) 将 3 个源码包解包到 /work/madplay/src-arm 目录:

```
/work/sysbuild/madplay/tarball$ tar xzvf libid3tag-0.15.1b.tar.gz -C
/work/sysbuild/madplay/src-arm
```

```
/work/sysbuild/madplay/tarball$ tar xzvf libmad-0.15.1b.tar.gz -C
/work/sysbuild/madplay/src-arm
/work/sysbuild/madplay/tarball$ tar xzvf madplay-0.15.2b.tar.gz -C
/work/sysbuild/madplay/src-arm
```

2. 指定交叉编译器

```
/work/sysbuild/madplay$ export CC=arm-linux-gcc
```

3. 交叉编译 libid3tag 库

(1) 配置:

```
/work/sysbuild/madplay$ cd src-arm/libid3tag-0.15.1b/
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ ./configure --host=arm-linux --
prefix=/work/sysbuild/madplay/target-arm
```

结果出现如图 7-4 所示的错误。

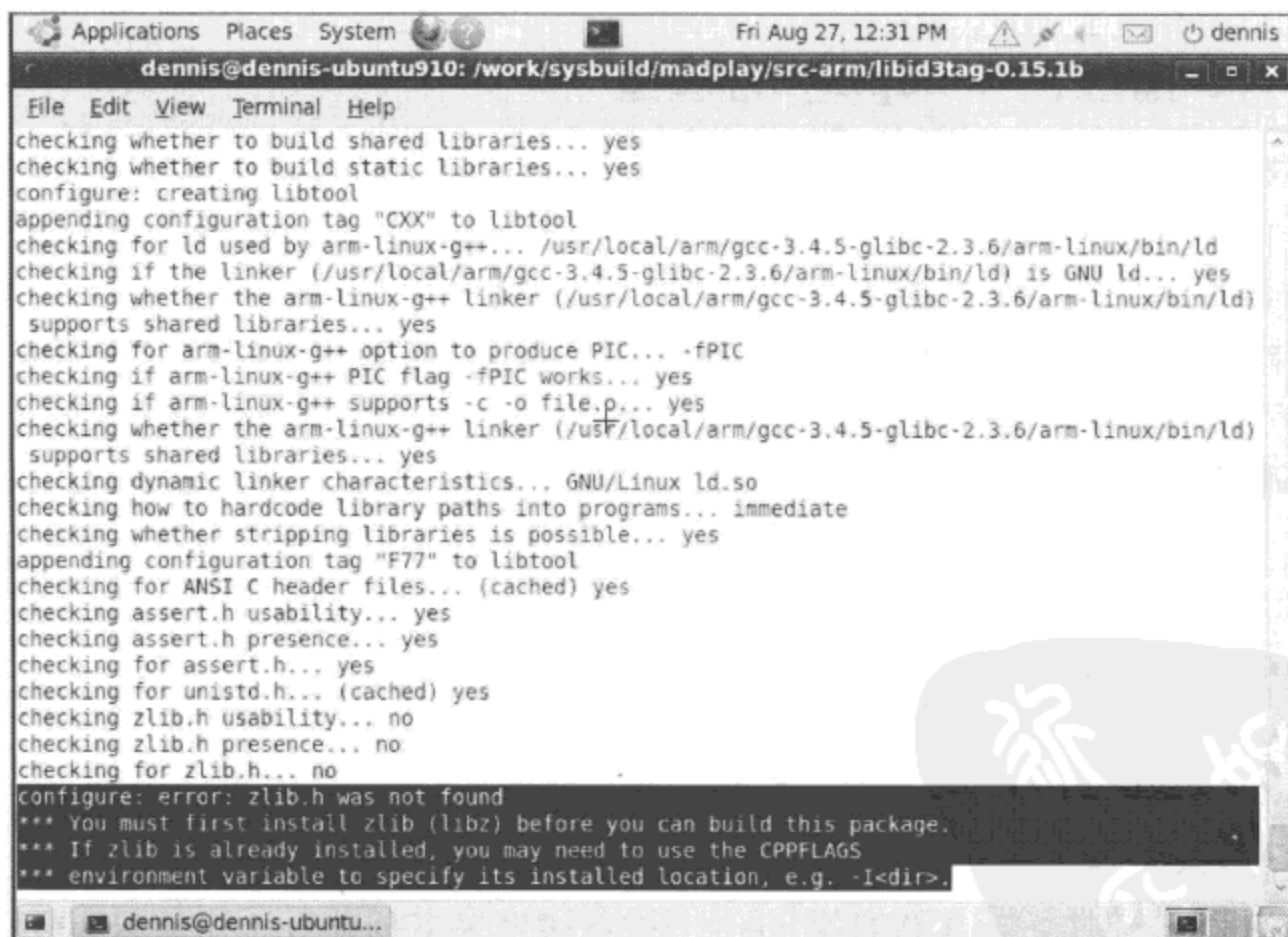


图 7-4 配置 libid3tag 库 1

由错误提示可知, libid3tag 库依赖于 zlib 库。我们已经在“移植 mtd 工具程序”时生成了 for ARM 的 zlib 库, 根据提示, 指定 zlib 库的头文件的位置即可。

(2) 更正配置如图 7-5 所示。

```
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ ./configure --host=arm-linux
--prefix=/work/sysbuild/madplay/target-arm
CPPFLAGS=-I/work/sysbuild/zlib-1.2.3/result/include/
```



图 7-5 配置 libid3tag 库 2

由错误提示可知,libid3tag 库依赖于 zlib 库。我们已经在“移植 mtd 工具程序”时生成了 for ARM 的 zlib 库,根据提示,指定 zlib 库的库文件的位置即可。

(3) 再次更正配置,成功:

```
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ ./configure --host=arm-linux
--prefix=/work/sysbuild/madplay/target-arm
CPPFLAGS=-I/work/sysbuild/zlib-1.2.3/result/include/
LDFLAGS=-L/work/sysbuild/zlib-1.2.3/result/lib
```

(4) 编译、安装:

```
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ make
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ make install
```

此时查看安装目录,可以看到 libid3tag 的库文件和头文件

```
/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ ls /work/sysbuild/madplay/target-arm/
```

```
lib -l
total 356
-rw-r--r-- 1 dennis dennis 199170 2010-08-27 12:47 libid3tag.a
-rwxr-xr-x 1 dennis dennis 884 2010-08-27 12:47 libid3tag.la
lrwxrwxrwx 1 dennis dennis 18 2010-08-27 12:47 libid3tag.so -> libid3tag.so.0.3.0
lrwxrwxrwx 1 dennis dennis 18 2010-08-27 12:47 libid3tag.so.0 -> libid3tag.so.0.3.0
-rwxr-xr-x 1 dennis dennis 147596 2010-08-27 12:47 libid3tag.so.0.3.0

/work/sysbuild/madplay/src-arm/libid3tag-0.15.1b$ ls /work/sysbuild/madplay/target-arm/
include -l
```

4. 交叉编译 libmad 库

```
/work/sysbuild/madplay/src-arm/libmad-0.15.1b$ ./configure --host=arm-linux
--prefix=/work/sysbuild/madplay/target-arm
CPPFLAGS=-I/work/sysbuild/madplay/target-arm/include
LDFLAGS=-L/work/sysbuild/madplay/target-arm/lib
/work/sysbuild/madplay/src-arm/libmad-0.15.1b$ make
/work/sysbuild/madplay/src-arm/libmad-0.15.1b$ make install
```

5. 交叉编译 madplay 应用程序(注意:madplay 依赖于 libz、libid3tag、libmad 库)

```
/work/sysbuild/madplay/src-arm/madplay-0.15.2b$ ./configure --host=arm-linux
--prefix=/work/sysbuild/madplay/target-arm
CPPFLAGS=-I/work/sysbuild/madplay/target-arm/include
LDFLAGS=-L/work/sysbuild/madplay/target-arm/lib -L/work/sysbuild/zlib-1.2.3/result/
lib
/work/sysbuild/madplay/src-arm/madplay-0.15.2b$ make
/work/sysbuild/madplay/src-arm/madplay-0.15.2b$ make install
```

6. 下载 madplay 到开发板运行测试

把它以及依赖库下载到开发板,并作如下放置:

执行文件: madplay 放在 /usr/bin/目录

库文件: libid3tag.so.0、libid3tag.so.0.3.0、libmad.so.0、libmad.so.0.2.1 放在 /usr/lib 目录。(注: libid3tag 和 madplay 依赖的 zlib 库已经放到了开发板根文件系统中,因此不需复制 zlib 库。)

执行结果如下：

```
# madplay /music/pianpianxihuanni.mp3
MPEG Audio Decoder 0.15.2 (beta) - Copyright (C) 2000 - 2004 Robert Leslie et al.
    Title: Track 1
    Artist: 陳百強
    Orchestra: 陳百強
    Album: Best Memory
    Track: 15
    Genre: Other
```

7.3.3 主要网络服务器的移植与使用

1. telnet 服务器

busybox 中已经包含了 telnet 服务器 telnetd, 所以只需要对它进行配置就可以了。(1~4 步均在开发板上执行)

(1) telnet 时需要输入登录用户名和密码, 所以首先创建用户。

① adduser dennis, 这将创建 dennis 用户。

② 在 /etc/passwd 中第一行新增: root:x:0:0:Root,,,:/root:/bin/sh。

③ passwd root, 设置 root 的密码为 1234。

④ passwd dennis, 设置 dennis 的密码 1234。

⑤ chown 0:0/bin/busybox, 改变 busybox 的属主, 否则将来 u+s 后第一个用户进程 init 的权限将不是 root 的权限, 而是 dennis 权限。

⑥ chmod u+s /bin/busybox, 这将使普通用户能使用 passwd 修改自己的密码, 同时也使将来要使用的 login 程序能正常工作。

(2) 创建组 audio, 并将 dennis 加入到 audio 组中, 为 dennis 能播放音乐做准备。

① addgroup -g 0 root, 这将增加一个组, 组名为 root, 组号为 0。

② addgroup audio, 这将增加一个组, 组名为 audio, 组号为 1。

③ addgroup dennis audio, 这将用户 dennis 加入到 audio 组中。

(3) 修改 /etc/inittab, 使得 telnetd 开机自动启动:

```
::sysinit:/etc/init.d/rcS
::once:/usr/sbin/telnetd
::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

注意此处必须使用 once, 一定不能用 respawn, 因为 telnetd 是守护进程, 其实现会 fork 自

己后让自己结束。

(4) 修改 rcS:

- ① 创建并挂载/dev/pts,它将供 telnetd 服务使用。
- ② 更改/dev/dsp 的组,以使 audio 组的用户可以播放音乐。
- ③ 更改/dev/tty 和/dev/console 的权限,以使普通用户登录系统时也能读写控制终端。

修改后的 rcS 如下:

```
# ! /bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
mount -t tmpfs none /dev
mount -t tmpfs none /var
mount -t tmpfs none /tmp
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir /dev/pts
mount -t devpts devpts /dev/pts
chown root:audio /dev/dsp
chmod 666 /dev/tty
chmod 600 /dev/console
```

(5) 测试 telnet 服务器。在 Linux 机器上执行 telnet 192.168.1.17,输入正确的用户名和密码登录开发板后,播放开发板上的 mp3 音乐。

```
~$ telnet 192.168.1.17
Trying 192.168.1.17...
Connected to 192.168.1.17.
Escape character is '^]'.
www login: dennis
Password:
$ madplay /music/pianpianxihuanni.mp3
MPEG Audio Decoder 0.15.2 (beta) - Copyright (C) 2000 - 2004 Robert Leslie et al.
Title: Track 1
Album: Best Memory
Track: 15
Genre: Other
```

(6) 顺便让控制台登录也需要输入用户名和密码。将 etc/inittab 中的::askfirst:-/bin/sh 换为::respawn:-bin/login

2. FTP 服务器

(1) 从 vsftpd 官方网站: <http://vsftpd.beasts.org/> 下载版本 vsftpd-2.0.6.tar.gz (光盘\work\sysbuild\vsftpd-2.0.6.tar.gz), 并解压:

```
/work/sysbuild$ tar xzvf vsftpd-2.0.6.tar.gz
/work/sysbuild$ cd vsftpd-2.0.6
```

(2) 交叉编译(for ARM-Linux)。

① 修改 Makefile, 指定交叉编译器。

将第二行改为 CC = arm-linux-gcc

② 修改 vsf_findlibs.sh

将所有的 /lib 和 /usr/lib 前面加上 /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux, 以使库目录指向交叉编译工具链的库位置。

(3) make 后生成 vsftpd, 将 vsftpd 复制到开发板根文件系统相应目录:

```
/work/system/vsftpd-2.0.6$ make
/work/system/vsftpd-2.0.6$ cp vsftpd /work/nfs_root/myfs /usr/sbin/
```

(4) 复制 vsftpd 依赖的动态库文件到开发板根文件系统相应目录:

```
/work/system/vsftpd-2.0.6$ arm-linux-readelf -a ./vsftpd | grep 'Shared
0x00000001 (NEEDED)           Shared library: [libcrypt.so.1]
0x00000001 (NEEDED)           Shared library: [libdl.so.2]
0x00000001 (NEEDED)           Shared library: [libnsl.so.1]
0x00000001 (NEEDED)           Shared library: [libresolv.so.2]
0x00000001 (NEEDED)           Shared library: [libc.so.6]
```

可见 vsftpd 依赖 5 个库, 其中第 5 个库在前面已经复制过, 现在只需要复制另外 4 个库。

```
/work/sysbuild/vsftpd-2.0.6$ cd /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libdl.so.2 libdl-2.3.6.so
/work/nfs_root/myfs/lib/
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libnsl.so.1 libnsl-2.3.6.so
/work/nfs_root/myfs/lib/
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libresolv.so.2 libresolv-2.
3.6.so
/work/nfs_root/myfs/lib/
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libcrypt.so.1 libcrypt-2.3.
6.so
/work/nfs_root/myfs/lib/
```

(5) 创建配置文件 vsftpd.conf

将模板配置文件复制到开发板 etc 目录

```
/work/sysbuild/vsftpd-2.0.6 $ cp ./vsftpd.conf /work/nfs_root/myfs/etc/
```

修改该配置文件,使有效配置如下:

```
anonymous_enable = YES
local_enable = YES
write_enable = YES
dirmesssage_enable = YES
connect_from_port_20 = YES
nopriv_user = ftp
#listen entry enable standalone mode
listen = YES
```

(6) 新建相应用户及目录。由于 vsftpd 源代码程序一定要使用一个用户 ftp(供匿名用户 anonymous 映射);同时默认情况下还要使用一个非特权用户 nobody。(5)中已经设置了非特权用户为 ftp。因此只需新建一个用户 ftp 即可。

```
# adduser ftp
```

再修改匿名用户主目录的属主和权限:

```
# chown root:root /home/ftp
# chmod 755 /home/ftp
```

由于 vsftpd 源代码程序一定要使用一个目录/usr/share/empty,所以必须预先创建它。

```
/work/nfs_root/myfs $ mkdir -p usr/share/empty
```

(7) 复制辅助库文件到开发板。由于 vsftpd 的源代码程序在寻找 ftp 用户时,调用了 getpwnam 库函数去解析/etc/passwd 文件。而 getpwnam 库函数若要正确运行,需要一些其他辅助库函数。

```
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib $ cp -P libnss_* /work/nfs_root/myfs/lib/
```

注:这样操作会使根文件系统变大。另一个解决方案是修改 vsftpd 的源代码,使用 busybox 中的账户管理 API,而不用标准 libc 中的账户管理 API,参见 http://hi.baidu.com/hzau_wall_e/blog/item/1c2bd462d458a5680c33facd.html

(8) 修改/etc/inittab,使 vsftpd 在启机时自动启动。

```
::sysinit:/etc/init.d/rcS
::once:/usr/sbin/telnetd
::respawn:/usr/sbin/vsftpd
::respawn:- bin/login
::ctrlaltdel:/sbin/reboot
```

```
::shutdown:/bin/umount -a -r
```

(9) 测试 vsftpd

```
/ $ ftp 192.168.1.17
Connected to 192.168.1.17.
220 (vsFTPd 2.0.6)
Name (192.168.1.17;dennis): dennis
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> quit
221 Goodbye.

/ $ ftp 192.168.1.17
Connected to 192.168.1.17.
220 (vsFTPd 2.0.6)
Name (192.168.1.17;dennis): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> quit
221 Goodbye.
```

3. HTTP 服务器

busybox 中已经包含了 HTTP 服务器 httpd, 所以只需要对它进行配置就可以了。

(1) 出于安全考虑, 应该让 httpd 运行在非 root 权限下, 因此先创建专用于 httpd 服务的普通用户 www (没有主目录和密码, 不能用于交互式登录)

```
# adduser -S -D -H www
```

(2) 修改 /etc/inittab 文件, 以在系统启动时自动启动 httpd, 并指定主目录为 /www, 服务运行账户是普通用户 www:

```
::sysinit:/etc/init.d/rcS
::once:/usr/sbin/telnetd
::once:/usr/sbin/httpd -h /www -u www
```

(3) httpd 运行时会以普通用户 www 的身份访问 /dev/null 设备, 因此需在 rcS 脚本中修

改/dev/null 的权限:

```
# ! /bin/sh
ifconfig eth0 192.168.2.17
mount -t proc none /proc
mount -t sysfs none /sys
mount -t tmpfs none /dev
mount -t tmpfs none /var
mount -t tmpfs none /tmp
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir /dev/pts
mount -t devpts devpts /dev/pts
chown root:audio /dev/dsp
chmod 666 /dev/tty
chmod 600 /dev/console
chmod 666 /dev/null
```

(4) 创建 HTTP 服务器的主目录和主文件。

```
mkdir /www
echo "this is my first web site" > /www/index.html
```

(5) 测试 http 服务器。

① 重新启动开发板 Linux。可见 HTTP 服务器自动启动并运行在 www 用户权限下。

```
VFS: Mounted root (nfs filesystem).
Freeing init memory: 112K
starting pid 230, tty ": '/etc/init.d/rcS'
starting pid 244, tty ": '/usr/sbin/telnetd'
starting pid 245, tty ": '/usr/sbin/httpd'
Please press Enter to activate this console.
starting pid 246, tty ": '/bin/sh'
# ps
  PID  Uid          VSZ Stat Command
    1  root        3092 S   init
    2  root                SW< [kthreadd]
    3  root                SWN [ksoftirqd/0]
    4  root                SW< [events/0]
    5  root                SW< [khelper]
   42  root                SW< [kblockd/0]
   43  root                SW< [ksuspend_usbd]
```



```

46 root          SW< [khubd]
48 root          SW< [kseriod]
60 root          SW  [pdflush]
61 root          SW  [pdflush]
62 root          SW< [kswapd0]
63 root          SW< [aio/0]
178 root         SW< [mtdblockd]
227 root         SW< [rpciod/0]
246 root         3096 S   - sh
251 root         3092 S   /usr/sbin/telnetd
252 www          3092 S   /usr/sbin/httpd -h /www -u www
253 root         3096 R   ps

```

② 在 Linux 机器上启动浏览器访问 HTTP 服务器,看到最终正确的结果如图 7-6 所示。

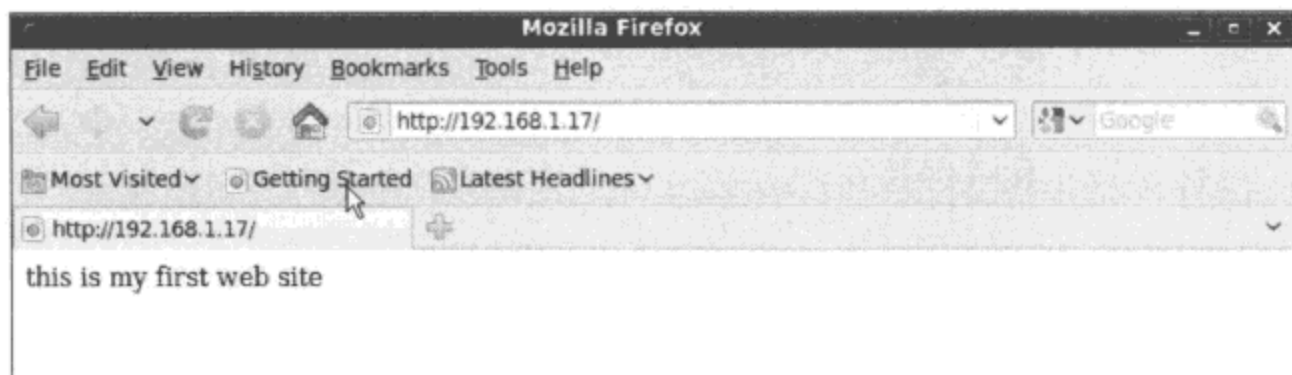


图 7-6 测试 http 服务器

7.3.4 数据库程序的移植与使用

1. SQLite 数据库的介绍

SQLite 数据库是一种嵌入式数据库,它的目标是尽量简单,因此它抛弃了传统企业级数据库的种种复杂特性,只实现对于数据库而言必备的功能。尽管简单性是 SQLite 追求的首要目标,但是其功能和性能都非常出色,它具有这样一些特性:

- 支持 ACID 事务 (ACID 是 Atomic、Consistent、Isolated 和 Durable 的缩写)。
- 零配置,不需要任何管理性的配置过程。
- 支持 SQL92 标准。
- 所有数据存放到单独的文件中,支持的最大文件可达 2TB。
- 数据库可以在不同字节的机器之间共享。
- 体积小。
- 系统开销小,检索效率高。

- 简单易用的 API 接口。
- 可以和 Tcl、Python、C/C++、Java、Ruby、Lua、Perl、PHP 等多种语言绑定。
- 自包含,不依赖于外部支持。
- 良好注释的代码。
- 代码测试覆盖率达 95% 以上。
- 开放源码,可以用于任何合法用途。

由于 SQLite 具有功能强大、接口简单、速度快、体积小等一系列优点,因此特别适合应用在嵌入式系统中。

总而言之,SQLite 很像 windows 下的 Access 数据库,非常适合在嵌入式领域使用。

2. 移植 SQLite

(1) 从 <http://sqlite.org/download.html> 下载最新的 SQLite 源代码(光盘\work\sysbuild\sqlite-amalgamation-3.6.23.1.tar.gz 提供)。

(2) 解压源代码 `tar xzvf sqlite-amalgamation-3.6.23.1.tar.gz`,进入源代码顶层目录 `cd sqlite-3.6.23.1`。

(3) 配置并进行交叉编译和安装:

```
/work/sysbuild/sqlite-3.6.23.1 $ ./configure --enable-shared
--prefix=/work/sysbuild/sqlite-3.6.23.1/result --host=arm-linux
/work/sysbuild/sqlite-3.6.23.1 $ make
/work/sysbuild/sqlite-3.6.23.1 $ make install
```

最终在 result 目录下得到 for ARM 的数据库管理程序 `sqlite3`(相当于 Windows 下 Access 这个应用程序),提供编程所需的 API 的动态库 `libsqlite3.so.0.8.6`,编程所需的头文件 `sqlite3ext.h` 和 `sqlite3.h`。

```
/work/sysbuild/sqlite-3.6.23.1/result $ ls
bin include lib share
/work/sysbuild/sqlite-3.6.23.1/result $ ls bin
sqlite3
/work/sysbuild/sqlite-3.6.23.1/result $ ls -l lib
total 2252
-rw-r--r-- 1 dennis dennis 1274962 2010-08-27 16:57 libsqlite3.a
-rwxr-xr-x 1 dennis dennis 855 2010-08-27 16:57 libsqlite3.la
lrwxrwxrwx 1 dennis dennis 19 2010-08-27 16:57 libsqlite3.so -> libsqlite3.so.0.8.6
lrwxrwxrwx 1 dennis dennis 19 2010-08-27 16:57 libsqlite3.so.0 -> libsqlite3.so.0.8.6
-rwxr-xr-x 1 dennis dennis 1009316 2010-08-27 16:57 libsqlite3.so.0.8.6
drwxr-xr-x 2 dennis dennis 4096 2010-08-27 16:57 pkgconfig
/work/sysbuild/sqlite-3.6.23.1/result $ ls include
```



```
sqlite3ext.h  sqlite3.h
```

(4) 将数据库管理程序 sqlite3、提供编程所需的 API 的动态库 libsqlite3.so.0.8.6 及其 1 个软链接复制到开发板根文件系统相应位置:

```
/work/sysbuild/sqlite-3.6.23.1/result$ cp bin/sqlite3 /work/nfs_root/myfs/usr/bin/
/work/sysbuild/sqlite-3.6.23.1/result$ cp -P lib/libsqlite3.so.* /work/nfs_root/myfs/usr/lib
```

(5) 由于 sqlite3 以及 libsqlite3.so.0.8.6 本身还要用到 C 标准库中 dl 和 pthread 库,因此需要将这两个库及其软链接(位于交叉编译工具链的 C 库所在目录/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib/)复制到开发板根文件系统中的/lib 目录。

(6) 为了能在开发机上编译调用了 SQLite 数据库 API 的应用程序,需要将动态库 libsqlite3.so.0.8.6 及其两个软链接,两个头文件复制到交叉编译工具链所在目录的适当位置

```
/work/sysbuild/sqlite-3.6.23.1/result$ cp -P lib/libsqlite3.so.*
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/usr/lib
/work/sysbuild/sqlite-3.6.23.1/result$ cp include/*.h
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/include/
```

3. 在开发板上,使用数据库管理程序 sqlite3 操作数据库

```
# cd /root
# sqlite3 test.db
SQLite version 3.6.23.1
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table student(name varchar(10),age INTEGER);
sqlite> insert into student values('TOM',23);
sqlite> select * from student;
TOM|23
sqlite> .quit
# ls -l test.db
-rw-r--r-- 1 root root 2048 Apr 5 17:08 test.db
```

4. 编写 c 程序 testdb.c 操作数据库

```
1 #include <stdio.h>
2 #include <sqlite3.h>
3 int main(void)
4 {
5     sqlite3 *db = NULL;
6     char *zErrMsg = NULL;
```

```

7   int rc;
8   rc = sqlite3_open("test.db", &db);
9   if (rc) {
10      fprintf(stderr, "cant open database: %s\n", sqlite3_errmsg(db));
11      sqlite3_close(db);
12      return 1;
13   } else {
14      printf("open a sqlite3 database named test.db successfully! \n");
15   }
16   char *sql = "create table student(id integer primary key,name varchar(10),age varchar
(10),sex varchar(6))";
17   sqlite3_exec(db, sql, 0, 0, &zErrMsg);
18   printf(" %s\n", zErrMsg);
19   sqlite3_free(zErrMsg);
20   sqlite3_close(db);
21   return 0;
22 }

```

第二行包含 sqlite 头文件,第 8 行打开数据库,第 17 行执行 sql 命令后错误消息存放在 zErrMsg 变量中,第 20 行关闭数据库。

交叉编译 arm-linux-gcc testdb.c -o testdb-lsqlite3 后,运行结果如下:

```

# ./testdb
open a sqlite3 database named test.db successfully!
table student already exists

```

7.4 建构 GUI 系统

在嵌入式 Linux 操作平台上,有各种各样的 GUI 系统,目前最常见的有:

(1) QTE 和 QPE,它由 nokia 的子公司奇趣公司出品,是嵌入式 Linux 上使用最广泛的 GUI 系统。

(2) Android,它由 google 公司开发,主要用于智能手机。

(3) MiniGUI,它由我国飞漫公司自主研发的小型 GUI 系统。

本书将在这里介绍 qtopia 的移植。

7.4.1 移植 tslib 库

qtopia 不仅能在 LCD 上显示图形界面,更有意思的是能支持触摸操作(试想一下如果智能手机不能支持触摸操作,那是多么的无趣!)。qtopia 支持触摸操作的功能依赖于底层触摸

屏函数库 tslib。

tslib 是一个开源的触摸屏支持库,它是在 handhelds.org 上开发的,作者是 Russul King, Douglas Lowder 和 Chris Larson。它给上层的应用程序,为不同的触摸屏提供了一个统一的接口。它提供诸如滤波、去抖、校准之类的功能。

因此首先要移植 tslib 库。移植步骤如下:

(1) 在 Linux 虚拟机的 /work/sysbuild 目录下解压源代码包(光盘\work\sysbuild\tslib-1.4.tar.gz)

```
$ cd /work/sysbuild
$ tar xzvf tslib-1.4.tar.gz
$ cd tslib
```

(2) 配置、编译、安装

```
$ ./autogen.sh
$ echo "ac_cv_func_malloc_0_nonnull=yes" >arm-linux-yz.cache
$ ./configure --host=arm-linux --prefix=/work/sysbuild/tslib/result --cache-file=arm-linux-yz.cache
$ make
$ make install
```

(3) 复制相关库文件和头文件到交叉编译工具链

完成第二步后,在 /work/sysbuild/tslib/result 会生成 4 个目录:bin、etc、include、lib。由于编译 qtopia 时会使用 and 链接 tslib 的库,所以需要将 lib 目录下 tslib 库文件及其两个软链接文件复制到交叉编译工具链的库目录,将 include 下的头文件复制到交叉编译工具链的头文件目录:

```
/work/sysbuild/tslib/result$ cp -P lib/libts* /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib
/work/sysbuild/tslib/result$ cp include/* /usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/include
```

(4) 复制相关库文件、配置文件、应用程序到开发板根文件系统

① 由于 qtopia 运行时会使用到 tslib 库,因此需将 lib 目录下的 tslib 库文件及其一个软链接文件复制到开发板根文件系统的库目录

```
/work/sysbuild/tslib/result$ cp -P lib/libts-0.0.so.0 lib/libts-0.0.so.0.1.1
/work/nfs_root/myfs/usr/lib
```

② qtopia 运行时有可能会调用触摸屏校准程序,因此要将此应用程序复制到开发板根文件系统的指定目录。

```
/work/sysbuild/tslib/result$ mkdir -p /work/nfs_root/myfs/usr/local/bin
/work/sysbuild/tslib/result$ cp bin/ts_calibrate /work/nfs_root/myfs/usr/local/bin
```

注:ts_calibrate 要复制到/usr/local/bin 下,这是在 qtopia 源代码中写死的,如要改变该目录需修改 qtopia 源代码。

③ 由于 tslib 库在运行时,会根据配置文件加载相关插件库(完成诸如滤波、去抖、校准之类的功能),所以需要在根文件系统中创建 tslib 的配置文件,并将插件库复制到根文件系统中

将 etc/ts.conf 第 2 行(# module_raw input)的注释符去掉后,将其复制到根文件系统的/etc 目录下:

```
/work/sysbuild /tslib/result$ cp etc/ts.conf /work/nfs_root/myfs/etc/ts.conf
/work/sysbuild /tslib/result$ mkdir -p /work/nfs_root/myfs/usr/tslib/lib/ts
/work/sysbuild /tslib/result$ cp lib/ts/* /work/nfs_root/myfs/usr/tslib/lib/ts
```

特别说明:根据 ts.conf 的内容其实只需要复制 4 个插件库即可

④ 在根文件系统的/bin 目录下创建 qtopia 脚本文件,在其中定义 tslib 库需要用到的环境变量,其内容如下:

```
1 #! /bin/sh
2 export TSLIB_TSDEVICE = /dev/event0
3 export TSLIB_CONFFILE = /etc/ts.conf
4 export TSLIB_PLUGINDIR = /usr/tslib/lib/ts
5 export TSLIB_CALIBFILE = /etc/pointercal
6 export QTDIR = /opt/Qtopia
7 export QPEDIR = /opt/Qtopia
8 export LD_LIBRARY_PATH = $QTDIR/lib: $LD_LIBRARY_PATH
9 export QWS_MOUSE_PROTO = "TPanel:/dev/event0"
10 export HOME = /root
11 exec $QPEDIR/bin/qpe
```

其中第 3 行指定 tslib 配置文件的位置和名称;第 4 行指定 tslib 插件库的位置;第 5 行指定存放校准结果的文件的位置和名称;第 2 行指定触摸屏的设备文件名;其他的环境变量是供 qtopia 使用的,参见下一小节的讲解。

7.4.2 移植 qtopia

(1) 编译、安装 qtopia 所依赖的库。

qtopia 依赖 jpeg、zlib、uuid 三个库,可以使用 qtopia 自带的 zlib 库,因此只需要编译、安装 jpeg 和 uuid。

① 编译、安装 jpeg 库。

从 <http://www.ijg.org/files> 下载源代码 `jpegsrc.v6b.tar.gz` (光盘\work\sysbuild\jpegsrc.v6b.tar.gz), 进行配置

```
$ ./configure --enable-shared --enable-static --prefix=/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux --build=i386 --host=arm
```

然后修改生成的 Makefile, 如下:

将 `CC=gcc` 改为 `CC=arm-linux-gcc`

将 `AR=ar rc` 改为 `AR=arm-linux-ar rc`

将 `AR2=ranlib` 改为 `AR2=arm-linux-ranlib`

最后执行如下命令进行编译、安装:

```
$ make
$ make install-lib
```

这将在 `/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux` 中的 `include` 目录中生成 4 个头文件 `jconfig.h`, `jerror.h`, `jmorecfg.h`, `jpeglib.h`; 在 `lib` 目录中生成 `libjpeg` 库文件。

② 编译、安装 `uuid` 库。本书在“辅助处理工具的移植”中的“移植 `ext2` 文件系统所需工具”部分已经安装并编译了 `uuid` 库。

(2) 解压源代码包并打补丁。

```
/work/sysbuild$ tar xzvf qtopia-2.2.0.tgz
/work/sysbuild$ patch -p0 < qtopia-2.2.0-my.patch
```

(3) 配置、编译、安装 `qtopia`。

```
/work/sysbuild$ cd qtopia-2.2.0
/work/sysbuild/qtopia-2.2.0$ echo yes | ./configure -qte'-embedded -no-xft -qconfig qpe
-depths 16,32 -system-jpeg -qt-zlib -qt-libpng -gif -no-g++ -exceptions -no-qvfb -
xplatform linux-arm-g++ -tslib' -qpe'edition pda -displaysize 240x320 -fontfamilies "helvet-
ica fixed micro smallsmooth smoothtimes unifont" -xplatform linux-arm-g++ -luuid' -qt2'-no-
opengl -no-xft' -dqt'-no-xft -thread'
/work/sysbuild/qtopia-2.2.0$ make
/work/sysbuild/qtopia-2.2.0$ make install
```

(4) 将字体文件(位于光盘\work\sysbuild\fonts.tar.gz)解压到相应目录。

```
/work/sysbuild/qtopia-free-2.2.0$ tar xzvf ../fonts.tar.gz -C
/work/sysbuild/qtopia-2.2.0/qtopia/image/opt/Qtopia/lib/fonts
```

(5) 将 `opt` 目录的全部内容复制到根文件系统中。

```
/work/sysbuild/qtopia-free-2.2.0$ cp qtopia/image/opt /work/nfs_root/myfs/ -R
```

(6) 在根文件系统中增加时区数据。

```
/work/sysbuild/qtopia-free-2.2.0$ sudo chmod 777 /work/nfs_root/myfs/usr/share
/work/sysbuild/qtopia-free-2.2.0$ tar xzvf ../zoneinfo.tar.gz -C /work/nfs_root/myfs/
usr/share
```

(7) 还有几个 qtopia 需要用到的库没有被复制,现在把它们复制到根文件系统:

```
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libstdc++.so.* /work/nfs_
root/myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp libgcc_s.so.1 /work/nfs_root/
myfs/lib
/usr/local/arm/gcc-3.4.5-glibc-2.3.6/arm-linux/lib$ cp -P libjpeg.so.* /work/nfs_
root/myfs/usr/lib
```

(8) 前面已经在根文件系统的/bin目录下创建 qtopia 脚本文件,其中:第6、7行是 qtopia 需要的环境变量;第8行定义 qtopia 加载库文件时的查找路径;第9行指定 qtopia 使用的输入设备是触摸屏,而不是鼠标;第11行真正启动 qtopia。

当然不要忘了给该脚本加上可执行权限。

```
/work/sysbuild/qtopia-2.2.0$ chmod a+x /work/nfs_root/myfs/bin/qtopia
```

(9) 在根文件系统中的/etc/init.d/rcS脚本文件的最后加入一行:

```
/bin/qtopia &>/dev/null &
```

以便系统启动后自动进入 GUI 图形系统。



第 3 篇

Linux 驱动程序开发

特别提醒

在本篇的示例中,>提示符表示在开发板的 Boot Loader 上操作;#提示符表示在开发板的 Linux 操作系统上操作;\$提示符表示在 PC 的 Linux 操作系统上操作,\$提示符前的是当前目录。例如:

```
Dennis Yang > usblave 1 0x32000000  
# madplay /music/pianpianxihuanni.mp3  
/work/studydriver/examples/scull$ cat /proc/devices
```



第 8 章

Linux 驱动程序开发基础

8.1 Linux 设备驱动程序简介

Linux kernel 系统架构图如图 8-1 所示。

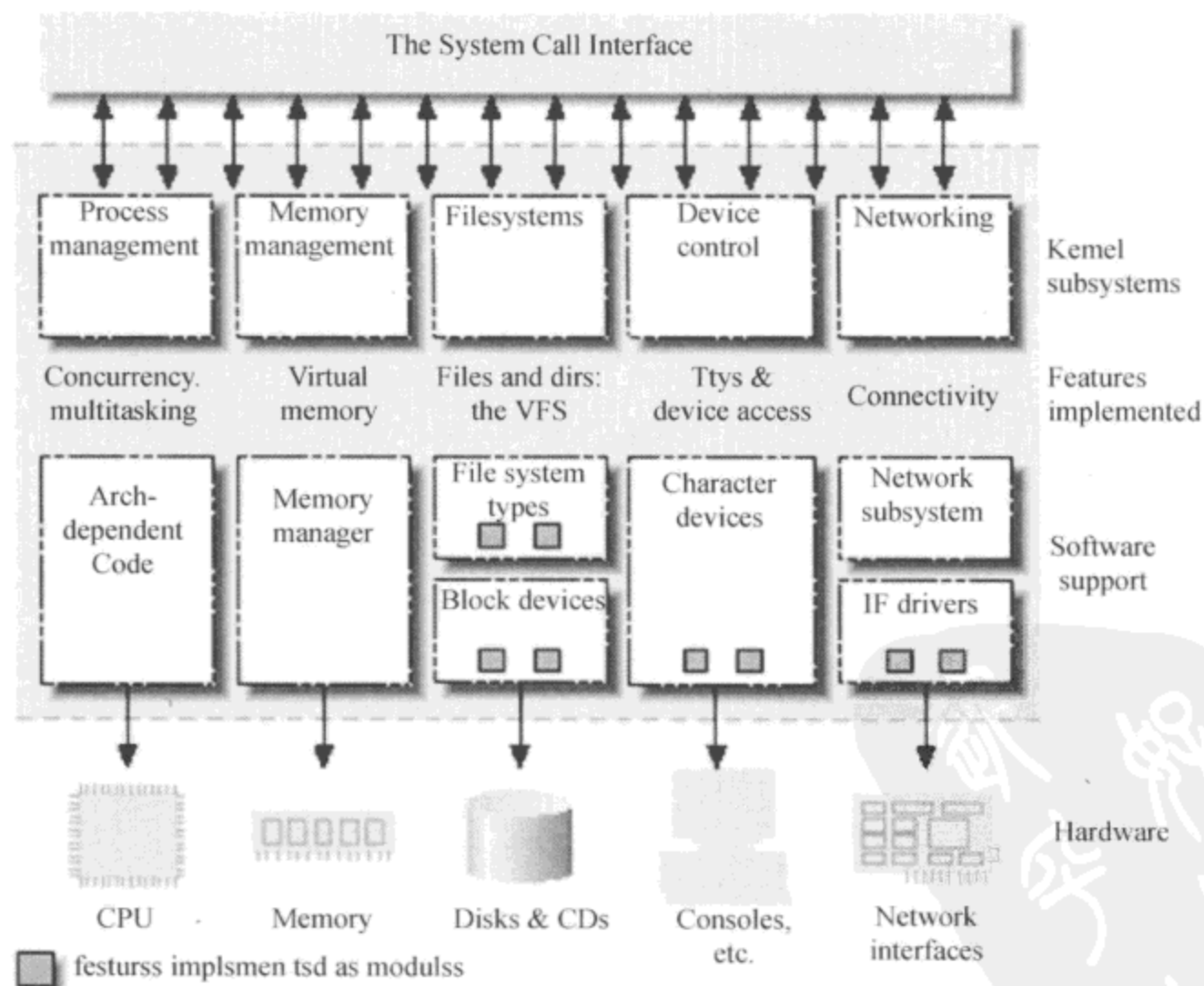


图 8-1 Linux kernel 系统架构图

驱动程序的特点如下：

(1) 是应用和硬件设备之间的一个软件层。

(2) 这个软件层一般在内核中实现。

(3) 设备驱动程序的作用在于提供机制,而不是提供策略,编写访问硬件的内核代码时不要给用户强加任何策略:

① 机制:驱动程序能实现什么功能。

② 策略:用户如何使用这些功能。

只提供协议或者功能就可以,不要限制其他人如何用,这就是机制和策略

8.1.1 设备驱动分类和内核模块

Linux 系统将设备驱动分成三种类型。

(1) 字符设备。

(2) 块设备。

(3) 网络设备。

内核模块是内核提供的一种可以动态加载功能单元来扩展内核功能的机制,类似于软件中的插件机制。这种功能单元称为内核模块。

通常为每个驱动创建一个不同的模块,而不在一个模块中实现多个设备驱动,从而实现良好的伸缩性和扩展性。

1. 字符设备

字符设备是个能够像字节流(比如文件)一样访问的设备,由字符设备驱动程序来实现这种特性。通过/dev 下的字符设备文件来访问。字符设备驱动程序通常至少需要实现 open、close、read 和 write 等系统调用所对应的、对该硬件进行操作的功能函数。

应用程序调用 system call(系统调用),例如 read、write,将会导致操作系统执行上层功能组件的代码,这些代码会处理内核的一些内部事务,为操作硬件做好准备,然后就会调用驱动程序中实现的对硬件进行物理操作的函数,从而完成对硬件的驱动,然后返回操作系统上层功能组件的代码,做好内核内部的善后事务,最后返回应用程序。

由于应用程序必须使用/dev 目录下的设备文件(参见 open 调用的第一个参数),所以该设备文件必须事先创建。谁创建设备文件呢?

大多数字符设备是个只能顺序访问的数据通道,不能前后移动访问指针,这一点和文件不同。比如串口驱动,只能顺序地读写设备。然而,也存在和数据区或者文件特性类似的字符设备,访问它们时可前后移动访问指针。例如 framebuffer 设备就是这样一个设备,应用程序可以用 mmap 或 lseek 访问图像的各个区域。

2. 块设备

块设备通常是按照块为单位来访问数据,比如一块为 512KB。

块设备也是通过 /dev 目录下的文件系统节点来访问。块设备和字符设备的区别仅仅在

第8章 Linux 驱动程序开发基础

于内核内部管理数据的方式,也就是内核和驱动程序的接口不同。

块设备除了给内核提供和字符设备一样的接口外,还提供了专门面向块设备的接口,块设备的接口必须支持挂装文件系统,通过此接口,块设备能够容纳文件系统,因此应用程序一般通过文件系统来访问块设备上的内容,而不是直接和块设备打交道。

文件系统可能是除驱动程序外 Linux 系统中最重要的模块类型,与块设备驱动程序联系紧密。

3. 网络设备驱动和网络接口

网络设备驱动不同于字符设备和块设备,不在/dev下以文件节点为代表,而是通过单独的网络接口(eth0、eth1)来代表。

任何网络事务都要经过一个网络接口,即一个能够和其他主机交换数据的设备。通常接口代表一个硬件设备(如网卡),但也可能是个纯软件设备。

内核和网络驱动程序间的通信完全不同于内核和字符设备以及块设备驱动程序之间的通信,内核调用一套与数据包传输相关的函数。

8.1.2 设备文件和设备驱动

设备文件是文件系统上的一个节点,是一种特殊的文件,称做设备文件。每个设备文件在用户空间代表了一个设备。设备文件一般存在/dev目录下,用mknod命令创建。设备文件有主、次设备号与其关联。设备文件是用户应用程序和设备驱动的接口。应用程序一般只能通过设备文件来使用设备驱动的功能。字符和块设备驱动必须有相应的设备文件来对应,如图8-2所示。

很明显,操作系统内部不可能用设备文件名来与物理设备及其驱动进行绑定。其实,操作系统内部是用设备号来与物理设备及其驱动进行绑定的。习惯上,用主设备号与驱动进行关联,用次设备号与具有相同驱动的不同物理设备关联(例如两个硬盘)。

```
~$ ls -l /dev/sd[a-c]
brw-rw---- 1 root disk 8, 0 2010-04-13 13:38 /dev/sda
brw-rw---- 1 root disk 8, 16 2010-04-13 13:38 /dev/sdb
brw-rw---- 1 root disk 8, 32 2010-04-13 13:38 /dev/sdc
```

当用户程序运行 `open("/dev/ttyS0",...)` 时,由于设备文件 `/dev/ttyS0` 有一个设备号与其关联,因此操作系统可以获知应用程序想操控的设备的设备号,而操作系统内部又将设备号与物理设备及其驱动进行了绑定,因此操作系统就可以知道应该调用哪一个驱动去控制哪一个设备。当然这一切的前提是,操作系统内部要将设备号与物理设备及其驱动进行绑定,那么操作系统内部是用什么手段完成这种绑定关系的呢? 实际上,在操作系统内部存在一个结构体链表(就是图8-2中的 Char device list,以后称它为设备链表),链表的每个节点代表一个绑

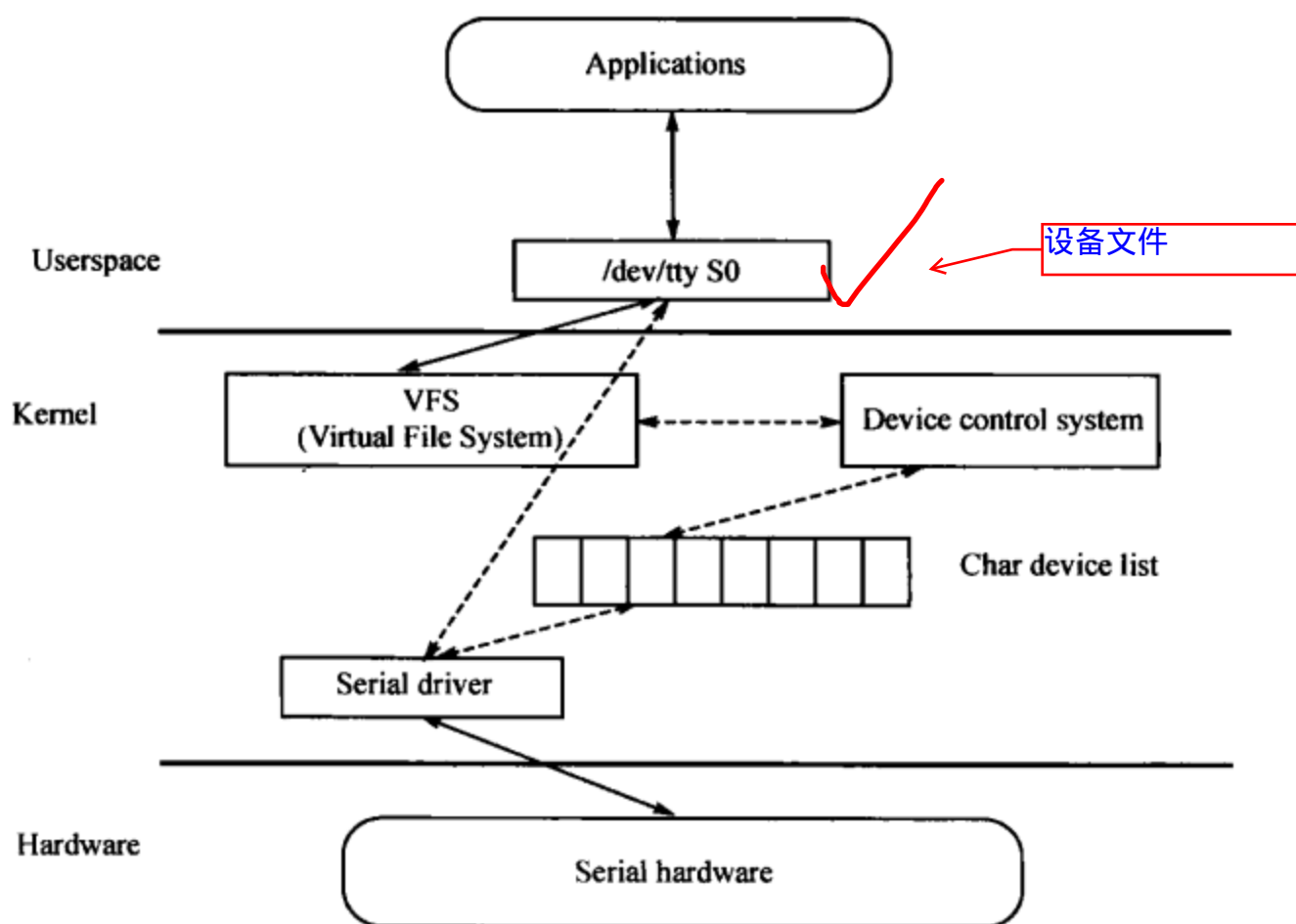


图 8-2 字符设备文件与设备驱动

定关系(也就是说:节点至少含有 2 个字段,1 个用于记录设备号,另 1 个用于记录寻找驱动的信息,通常是一个指向驱动中的功能函数的结构体的指针)。那么是谁生成该节点并将它链入链表的呢?当然是驱动程序!

8.1.3 内核模块的编译和使用

1. 构造和运行模块

(1) Kernel Module 的特点如下:

- ① 模块只是先注册自己以便服务于将来的某个请求,然后就立即结束。
- ② 模块可以是实现驱动程序、文件系统或者其他功能。
- ③ 加载模块后,模块运行在内核空间,和内核链接为一体。

(2) 模块与内核的接口函数(除掉 read、write 等功能函数)。

生成节点并将它链入设备链表这个操作由驱动中的函数实现,这些函数什么时机运行呢?当然最合适的时机是内核加载模块(insmod 模块)的时候。

① 函数 module_init:内核加载模块的时候调用。主要功能是:为以后使用模块里的函数和变量预先做准备。

② 函数 module_exit:模块的第二个入口点,内核在模块即将卸载之前调用它。

(3) 操作模块的相关命令如下:

① insmod: 加载模块。后面参数是模块文件名。

```
# insmod /lib/modules/hello.ko
Hello, world
```

② rmmod: 卸载模块。后面参数是模块名称。

```
# rmmod hello
Goodbye, cruel world
```

③ lsmod: 列出当前内核使用的模块。或者查看 `/proc/modules` 文件。

④ depmod: 扫描 `/lib/modules/<kernel version>/` 目录下的所有内核模块, 从而给内核模块生成依赖文件。

生成 `/lib/modules/<kernel version>/modules.dep` 文件, 其中 `<kernel version>` 是当前运行内核的版本号。

⑤ modprobe: 根据 `modules.dep` 文件探测并加载内核模块。只需要给出模块名称, 自动寻找适合的模块文件, 并进行加载。注意和 `insmod` 的不同之处。

- 可以自动寻找模块文件并加载。
- 自动寻找并加载依赖的模块。

```
# cat /lib/modules/2.6.22.6/modules.dep/
lib/modules/s3c24xx_buttons.ko; /lib/modules/leds.ko
/lib/modules/leds.ko;
# lsmod
Module                Size  Used by    Not tainted
# modprobe s3c24xx_buttons
leds initialized
buttons initialized
# lsmod
Module                Size  Used by    Not tainted
s3c24xx_buttons        5944   0
leds                   3592   1 s3c24xx_buttons
# rmmod leds
rmmod; leds: Resource temporarily unavailable
# rmmod s3c24xx_buttons
buttons driver unloaded
# lsmod
Module                Size  Used by    Not tainted
leds                   3592   0
```




```
# rmmod leds
leds driver unloaded
# lsmod
Module                Size  Used by    Not tainted
# insmod s3c24xx_buttons
s3c24xx_buttons: Unknown symbol ledoff
s3c24xx_buttons: Unknown symbol ledon
insmod: cannot insert /lib/modules/s3c24xx_buttons.ko: Unknown symbol in module (-1): No
such file or directory
```

⑥ modinfo:查看模块文件的基本信息。

```
/work/studydriver/buttons$ modinfo s3c24xx_buttons.ko
filename:    s3c24xx_buttons.ko
license:     GPL
description:  S3C2410/S3C2440 BUTTON Driver
author:      YangZhu E-mail: scyz@263.net
depends:
vermagic:    2.6.22.6 mod_unload ARMv4
```

(4) 内核模块的编译方法。

内核源码树:指的是内核源代码 tar 包解压缩后形成的目录(包含其下级所有目录和文件)。

已编译内核源码树:指的是已经成功生成过内核的内核源码树(即已经成功执行过 make uImage 的内核源码树)。

驱动大多都编译为模块,2.6 内核中要想编译模块,必须先存在已经成功编译了的内核源码树(即已编译内核源码树),且该源码树编译出来的内核就是该模块即将运行在其上的内核。

编译方法 1:

① 编写 Makefile:obj-m := hello.o

② 编译命令:make -C 内核源码树目录 M=`pwd` modules。例如:

```
/work/studydriver/examples/misc-modules$ make -C /work/sysbuild/linux-2.6.22.6/ M=`pwd` modules
```

对该 make 命令的解释:

要想编译内核模块,只需要在内核源码树的顶层目录下输入 make modules 来编译 Makefile 中的 modules 目标即可,剩下的事情由内核构造系统全权替我们处理。但由于目前不处于内核源码树的顶层目录,并且当前目录下的 Makefile 也没有 modules 目标,因此使用 -C 参数来告知 make 程序需要在执行之前切换到 /work/sysbuild/linux-2.6.22.6/ 目录。此外,由于模块的源代码在当前目录中,不在内核源码树中,因此需要使用 M 变量(该变量是内核构造

系统的变量)告知内核构造系统,编译模块所需的源代码以及 Makefile 在当前目录(/work/studydriver/examples/misc-modules)中来找,而且最终生成的模块 KO 文件也要放在当前目录中。

编译方法 2:

① 编写 Makefile 如下:

```
ifeq ($ (KERNELRELEASE),)
    KERNELDIR ? = /work/sysbuild/linux-2.6.22.6
    PWD := $ (shell pwd)
modules:
    $ (MAKE) -C $ (KERNELDIR) M = $ (PWD) moduleselse
    obj-m := hello.oendif
```

② 编译命令 make。

对该 Makefile 的解释:

当 make 时,由于变量 KERNELRELEASE 尚未赋值,因此 ifeq (\$ (KERNELRELEASE),)为真,于是变量 KERNELDIR 被赋值为内核源码树目录/work/sysbuild/linux-2.6.22.6,变量 PWD 被赋值为当前目录/work/studydriver/examples/misc-modules,然后执行找到的第 1 个目标 modules,从而执行命令 make -C /work/sysbuild/linux-2.6.22.6 M=/work/studydriver/examples/misc-modules modules,而当该命令执行以调用内核构造系统的时候,内核构造系统会为变量 KERNELRELEASE 赋值,从而它不再为空,从而当前目录下的 Makefile 就变成了只有一行:“obj-m := hello.o”。此时情况与编译方法 1 的情况完全相同,因此 2 种编译方法得到了相同的结果。

最后将得到编译好的模块 hello.ko。

(5)简单的内核模块例子。

- 初始化函数:hello_init,使用宏 module_init 来声明。
- 销毁函数:hello_exit,使用宏 module_exit 来声明。
- 模块 LICENSE 信息,使用宏 MODULE_LICENSE 来说明。

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4
5 static int hello_init(void)
6 {
7     printk(KERN_ALERT "Hello, world\n");
8     return 0;
9 }
```

```
10
11 static void hello_exit(void)
12 {
13     printk(KERN_ALERT "Goodbye, cruel world\n");
14 }
15
16 module_init(hello_init);
17 module_exit(hello_exit);
```

模块执行结果:

```
# insmod hello.ko
Hello, world
# rmmod hello
Goodbye, cruel world
```

(6) 带参数的内核模块例子。

① 可以给模块在加载的时候传递参数。

② 使用宏 MODULE_PARM(变量名, 变量类型, 权限)来声明参数。变量有如下类型:
short、ushort、int、uint、long、ulong、charp、bool。

③ 使用方法: insmod hellp.ko howmany=3 whom="YangZhu"。

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 #include <linux/moduleparam.h>
4 static char *whom = "world";
5 static int howmany = 1;
6 module_param(howmany, int, S_IRUGO | S_IWUSR);
7 module_param(whom, charp, S_IRUGO);
8
9 static int hello_init(void)
10 {
11     int i;
12     for (i = 0; i < howmany; i++)
13         printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
14     return 0;
15 }
16
17 static void hello_exit(void)
18 {
19     printk(KERN_ALERT "howmany is %d, whom is %s\n", howmany, whom);
```



```

20     printk(KERN_ALERT "Goodbye, cruel world\n");
21 }
22
23 module_init(hello_init);
24 module_exit(hello_exit);

```

模块执行结果:

```

# insmod hello.ko
hello: module license 'unspecified' taints kernel.
(0) Hello, world ✓
# insmod hello.ko howmany = 3 whom = "YangZhu"
(0) Hello, YangZhu
(1) Hello, YangZhu
(2) Hello, YangZhu
# rmmod hello
howmany is 3, whom is YangZhu
Goodbye, cruel world

```

④ 模块运行期间,参数变量会以文件的形式出现在/sys 目录,MODULE_PARM 宏中的权限指定了该文件的权限。可以通过改变/sys 目录下文件的内容来改变参数变量的值。

```

# insmod hello.ko howmany = 3 whom = "YangZhu"
(0) Hello, YangZhu
(1) Hello, YangZhu
(2) Hello, YangZhu
# ls /sys/module/hello/parameters/ -l
-rw-r--r-- 1 root root 4096 May 3 22:09 howmany
-r--r--r-- 1 root root 4096 May 3 22:09 whom
# cat /sys/module/hello/parameters/howmany
3
# cat /sys/module/hello/parameters/whom
YangZhu
# echo 10 >/sys/module/hello/parameters/howmany
# echo YangYong >/sys/module/hello/parameters/whom
-sh: cannot create /sys/module/hello/parameters/whom: Permission denied
# rmmod hello
howmany is 10, whom is YangZhu
Goodbye, cruel world

```

(7) 编程注意事项如下:

① license 问题。Linux 内核源码以 GPL 许可发布,模块如果不声明自己使用的 license,

则加载的时候警告,这时可以使用 `MODULE_LICENSE("GPL")` 来避免。

② 避免“名字空间污染”:因为模块动态链接到内核里,最好不要输出内核中已有的全局函数或全局变量。否则会后者会影响前者。可以通过查看 `/proc/kallsyms` 来查看内核符号列表。解决方法:

- `EXPORT_NO_SYMBOLS`:使用此宏,这个模块不输出任何符号,除了使用下面的宏定义的符号。
- `EXPORT_SYMBOL(name)`:使用此宏定义的符号,强制输出。需要在 `EXPORT_NO_SYMBOLS` 使用之前使用才能输出。可以输出 `static` 的符号。

③ 模块之间的依赖问题:有的模块依赖于其他模块的函数或者变量,在加载前需要先加载所依赖的所有模块后,才能成功加载。卸载模块时要先卸载被依赖的所有模块后,才能成功卸载。

(8) 集成模块到内核步骤:

① 使用 `module_init` 和 `module_exit` 宏来定义内核模块接口函数,并确保模块工作正常(hello.c)。

② 把模块文件 `hello.c` 复制到内核的选定目录(例如 `drivers/char` 目录)。

③ 修改选定目录(例如: `drivers/char` 目录)下的 `Kconfig` 文件和 `Makefile` 文件。

- 修改 `Kconfig` 文件,增加如下:

```
config HELLO
    tristate New Hello
```

- 修改 `Makefile` 文件,增加如下:

```
obj-$(CONFIG_HELLO) += hello.o
```

④ 重新配置内核,选中要将该功能编译进内核,而不是编译为模块。

⑤ 重新编译内核,如果成功,则得到新内核。

⑥ 测试此内核,确保内核模块已集成。

2. 查看系统支持的设备

(1) `/proc/devices` 虚拟文件:系统支持的字符设备驱动、块设备驱动及其对应的主设备号。

```
# cat /proc/devices
Character devices:
    1 mem
    10 misc
    29 fb
    400 leds
```

```
232 buttons
Block devices,
    1 ramdisk
31 mtdblock
254 sbull
```

(2) dmesg 命令:查看系统的启动信息。可以看到系统支持的驱动的一些打印信息。

```
# dmesg
leds initialized
snull: snull initialized
buttons initialized
snull: enter snull_open
```

(3) /proc/ioports(或/proc/iomem)虚拟文件:查看设备的 I/O 内存物理地址。

```
# cat /proc/iomem
19000300 - 19000310 : cs8900
    19000300 - 19000310 : cs8900
30000000 - 33ffffff : System RAM
    30024000 - 30293fff : Kernel text
    30294000 - 302f1f97 : Kernel data
49000000 - 490fffff : s3c2410 - ohci
    49000000 - 490fffff : ohci_hcd
4d000000 - 4d0fffff : s3c2410 - lcd
4e000000 - 4e0fffff : s3c2440 - nand
    4e000000 - 4e0fffff : s3c2440 - nand
50000000 - 50003fff : s3c2440 - uart.0
    50000000 - 500000ff : s3c2440 - uart
50004000 - 50007fff : s3c2440 - uart.1
    50004000 - 500040ff : s3c2440 - uart
50008000 - 5000bfff : s3c2440 - uart.2
    50008000 - 500080ff : s3c2440 - uart
52000000 - 520fffff : s3c2440 - usb gadget
53000000 - 530fffff : s3c2410 - wdt
    53000000 - 530fffff : s3c2410 - wdt
54000000 - 540fffff : s3c2440 - i2c
    54000000 - 540fffff : s3c2440 - i2c
55000000 - 550fffff : s3c2410 - iis
    55000000 - 550fffff : s3c2410 - iis
56000010 - 5600001b : qq2440_leds
```

PDF


```
56000054 - 56000057 : qq2440_button34
56000064 - 56000067 : qq2440_button12
57000000 - 570000ff : s3c2410 - rtc
    57000000 - 570000ff : s3c2410 - rtc
f0300000 - f03fffff : s3c2410 - lcd
```

(4) /proc/interrupts 虚拟文件,查看正在使用的中断号。

```
# cat /proc/interrupts
          CPU0
16:         4      s3c - ext0  KEY4
18:         0      s3c - ext0  KEY3
30:    461932          s3c  S3C2410 Timer Tick
32:         0          s3c  s3c2410 - lcd
34:         0          s3c  I2SSDI
35:         0          s3c  I2SSDO
42:         0          s3c  ohci_hcd:usb1
43:         0          s3c  s3c2440 - i2c
53:    11257      s3c - ext  eth0
55:         0      s3c - ext  KEY2
63:         0      s3c - ext  KEY1
70:         990    s3c - uart0  s3c2440 - uart
71:    2206      s3c - uart0  s3c2440 - uart
83:         0          -      s3c2410 - wdt
```

3. 驱动目录

Linux 驱动相关代码放在 drivers/ 目录下,常见驱动目录介绍如下:

- block/: 常见块设备驱动。
- char/, serial/: 虚拟终端, 串口。
- net/: 网络设备驱动。
- video/: VGA 和 framebuffer 设备驱动。
- Ide/, scsi/: IDE 和 SCSI 设备驱动。
- 顶层目录的子目录 sound/: 声卡驱动。

8.2 字符设备驱动基本编程

本章将使用内存来虚拟 4 个同类型字符设备 scull,并以该字符设备为例来进行字符设备驱动基本编程的讲解。本章的素材和源代码(做了部分修改)均来源于《Linux Device Driver》

一书的第3版,因此本章可视为对该书相关章节的阅读理解。

8.2.1 字符设备驱动体验

(1) 下载 scull 设备的驱动源码(位于光盘\work\studydriver\examples\scull)到虚拟机 linux 系统/work/studydriver/examples,make 后可得到 scull.ko。将其加载进内核:insmod scull.ko。

(2) 创建设备节点文件:

```
/work/studydriver/examples/scull$ cat /proc/devices|grep scull
252 scull
252 scullop
/work/studydriver/examples/scull$ sudo mknod scull0 c 252 0
/work/studydriver/examples/scull$ sudo mknod scull1 c 252 1
/work/studydriver/examples/scull$ sudo mknod scull2 c 252 2
/work/studydriver/examples/scull$ sudo mknod scull3 c 252 3
/work/studydriver/examples/scull$ sudo chmod 666 scull[0-3]
```

(3) 体验 scull 设备。向该字符设备写入内容后再将内容读出。

```
/work/studydriver/examples/scull$ cat scull0
/work/studydriver/examples/scull$ sudo echo yangzhu > scull0
/work/studydriver/examples/scull$ cat scull0
```

8.2.2 实现字符设备驱动的工作

1. 确定主设备号和次设备号

(1) 什么是主设备/次设备号。

① 主设备号是内核识别一个设备属于哪一个驱动的标识。是一个整数,范围为 0~(4096-1),但是一般使用 1~255。

② 次设备号是驱动程序自己用来区别多个设备的。是一个整数,范围为 0~(1048576-1),但是一般使用 0~255。

③ 预定义的设备号:详见 Documentation/devices.txt。

④ 查看设备号:\$ ls -l /dev。

(2) 设备编号的内部表示。

① 内核用 32bit 表示设备号:

```
typedef unsigned long dev_t;
```

② 其中高 12bit 为主设备号,低 20bit 为次设备号。要想获得一个 dev_t 类型的变量中包

含的主或者次设备号,使用内核定义的宏:MAJOR(dev_t dev); 和 MINOR(dev_t dev);

```
#define MINORBITS    20
#define MINORMASK    ((1U << MINORBITS) - 1)
#define MAJOR(dev)    ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)    ((unsigned int) ((dev) & MINORMASK))
```

③ 主次设备号转换为一个 dev_t,使用内核定义的宏: MKDEV(int major, int minor);

```
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

④ 分配主设备号/次设备号的方法和内核 API。

静态分配设备号:请求操作系统分配驱动程序要求的特定设备号。first 为要求分配的第一个设备号(包含主、次设备号),count 为请求的设备号数量,name 为驱动名称(出现在 /proc/devices 中)。失败返回负数,成功则操作系统将 first 到 first+count-1,总共 count 个设备号分配给驱动。例如:如果 register_chrdev_region((200:2), 4, "test")成功,则分配到的设备号为(200:2)-(200:5)。请求分配的时机应该在驱动程序的初始化函数中。注:(200:2)表示一个设备号,该设备号的主设备号为 200,次设备号为 2。在不引起混淆的情况下,今后均采用这种方法表示设备号。

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

动态申请设备号。dev 用于存放结果,其最终存放的是分配到的 count 个设备号中的第一个设备号,firstminor 为期望分配到的第一个设备号的次设备号,count 为请求的设备号数量,name 为驱动名称(出现在 /proc/devices 中)。失败返回负数,成功则操作系统将分配的第一个设备号放在 dev 中,并且分配出去的设备号是从 dev 到 dev+count-1,共 count 个,而且保证第一个设备号(存放在 dev 中)的次设备号是 firstminor。例如:如果 alloc_chrdev_region(&dev, 3, 4, "test2")执行成功,则分配到的设备号为(201:3)-(201:6),并且 dev 的值变为(201:3)。申请的时机应该在驱动程序的初始化函数中。

```
int alloc_chrdev_region(dev_t * dev, unsigned int firstminor, unsigned int count, char * name);
```

以下是 main.c 中获取主设备号的代码:

```
44 int scull_major = SCULL_MAJOR; //宏 SCULL_MAJOR 在 scull.h 中被定义为 0
45 int scull_minor = 0;
688 if (scull_major) {
689     dev = MKDEV(scull_major, scull_minor);
690     result = register_chrdev_region(dev, scull_nr_devs, "scull");
691 } else {
692     result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
693     "scull");
```



```

694     scull_major = MAJOR(dev);
695 }
696 if (result < 0) {
697     printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
698     return result;
699 }

```

⑤ 释放主设备号/次设备号的方法和内核 API。

释放的时机应该在驱动程序的销毁函数中。

```

652     unregister_chrdev_region(devno, scull_nr_devs);

```

⑥ 早期的分配、释放主设备号/次设备号的内核 API。

由于目前内核源码中还有不少驱动使用早期的分配、释放主设备号/次设备号的内核 API, 所以这里也将早期的内核 API 做个列出, 以帮助大家在阅读内核源码时能够理解它们。

```

int register_chrdev(unsigned int major, const char * name, struct file_operations *
fops)

```

```

int unregister_chrdev(unsigned int major, const char * name)

```

2. 确定设备文件名称并创建设备文件作为用户程序与驱动的接口界面

(1) 设备文件名称是一个合法的文件名称即可。一般是设备名称, 或者是设备名称+数字。比如设备名称为 scull, 设备文件名可以为 scull0, scull1。

(2) 设备类型主要有 c(字符设备类型)和 b(块设备类型)。

(3) 创建设备文件 mknod /dev/scull0 c 252 0。

3. 将字符设备注册进操作系统

(1) 字符设备的注册时机在驱动程序的初始化函数中, 如图 8-3 所示, 注销时机在驱动程序的销毁函数中:

```

705     scull_devices = kmalloc(4 * sizeof(struct scull_dev), GFP_KERNEL); // kmalloc 的作用
//相当于应用程序中 malloc

```

```

710     memset(scull_devices, 0, 4 * sizeof(struct scull_dev));

```

```

721     scull_setup_cdev(&scull_devices[i], i); // i 是 4 个设备中第 i 个设备的编号

```

```

740 module_init(scull_init_module); //函数 scull_init_module 的代码位于 678-738 行

```

```

642     cdev_del(&scull_devices[i].cdev);

```

```

741 module_exit(scull_cleanup_module); //函数 scull_cleanup_module 的代码位于 633-658 行

```

(2) 字符设备是如何在操作系统中被注册和注销的? 如图 8-4 所示。

图 8-4 是一个 scull 设备的逻辑图(虚拟设备 scull 以内存作为设备存储数据的地方, 其设计详情, 请参阅《Linux Device Driver》第 3 版的 3.1 和 3.6 节)。

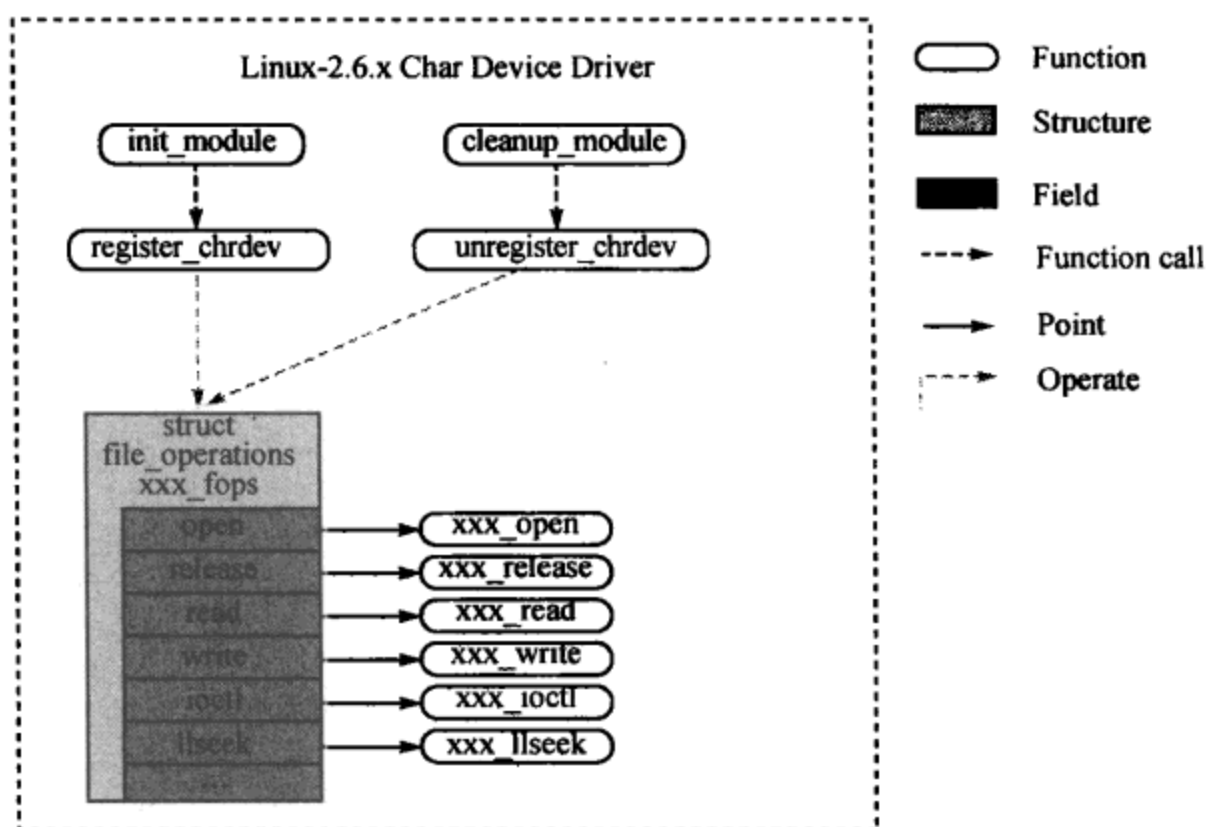


图 8-3 字符设备驱动架构图

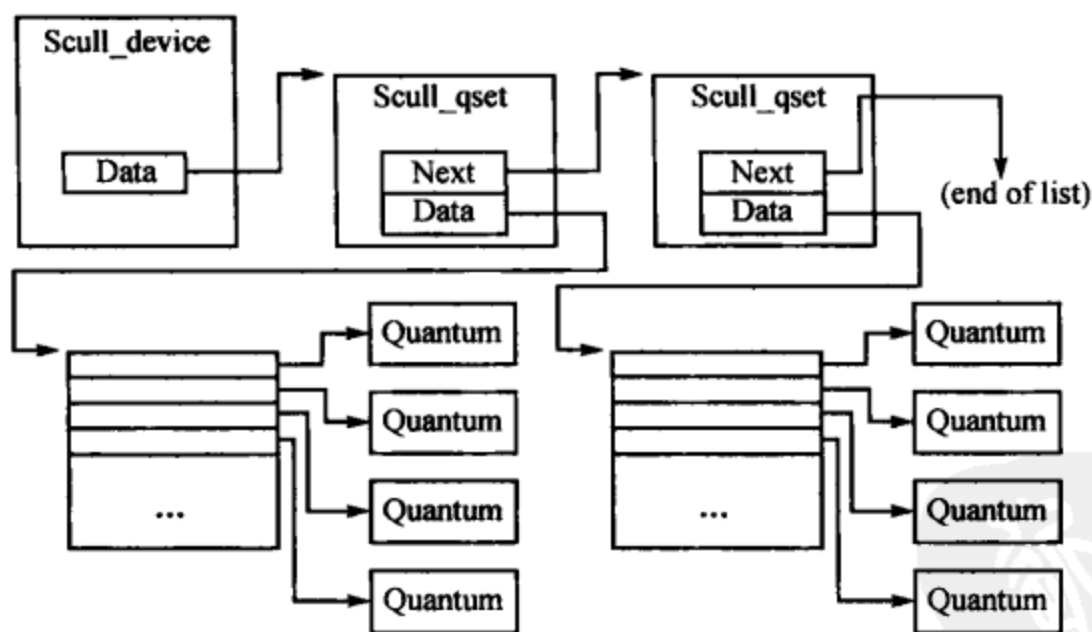


图 8-4 scull 设备示意图

图 8-4 中的 `Scull_device` 是一个 `scull_dev` 结构体，其中最重要的是 `struct cdev` 结构体，它被内核用来在内部表示一个字符设备。

```
struct scull_dev {
    struct scull_qset *data; /* 指向设备存储区的第一个 quantum 集合 */
    int quantum;           /* 表明一个 quantum 的大小 */
    int qset;              /* 表明一个 quantum 集合中有几个 quantum */
};
```

```

    unsigned long size;          /* 当前设备中总的数量字节数 */
    struct cdev cdev;            /* 内核内部用于表示一个字符设备的结构体 */
};

struct cdev {
    struct kobject kobj;
    struct module * owner;
    const struct file_operations * ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};

664 static void scull_setup_cdev(struct scull_dev * dev, int index)
665 {
666     int err, devno = MKDEV(scull_major, scull_minor + index);
667
668     cdev_init(&dev->cdev, &scull_fops);
669     dev->cdev.owner = THIS_MODULE;
670     ✓ dev->cdev.ops = &scull_fops;
671     err = cdev_add(&dev->cdev, devno, 1);
672     /* Fail gracefully if need be */
673     if (err)
674         printk(KERN_NOTICE "Error %d adding scull %d", err, index);
675 }

```

668 行初始化了 `struct cdev` 结构体(在不引起混淆的情况下,以后将称其为 `cdev`)的各个字段,其中最重要的是把 `ops` 初始化为了 `scull_fops`。这样 `cdev` 结构体就与 `scull_fops`(记录了驱动中实现的操作硬件的全部功能函数)建立了关联的关系。671 行将 `scull` 设备的设备号 `devno(252:0)` 和 `cdev` 注册进操作系统(在将 `cdev` 的 `dev`、`ops`、`count` 字段正确填写后,链入操作系统的字符设备链表),这样一来操作系统内部就建立了设备号-`cdev`-`scull_fops` 三者之间的关联关系,如图 8-5 所示。

```

642     cdev_del(&scull_devices[i].cdev);

```

而要从操作系统中注销字符设备,只需要执行内核 API `cdev_del` 即可。它将 `cdev` 从操作系统的字符设备链表中移除。

(3) 应用程序调用 `open` 打开一个设备时,操作系统做了什么? 如图 8-6 所示。

由于操作系统内部已经建立了设备号-`cdev`-`scull_fops` 三者之间的关联关系,所以当用户程序调用 `open(fd, "/dev/scull0")` 打开设备文件的时候,操作系统就可以根据设备文件名得到设备号,再根据设备号找到 `cdev`,进而找到 `fops`,从而为该设备在内核空间中建立 3 张

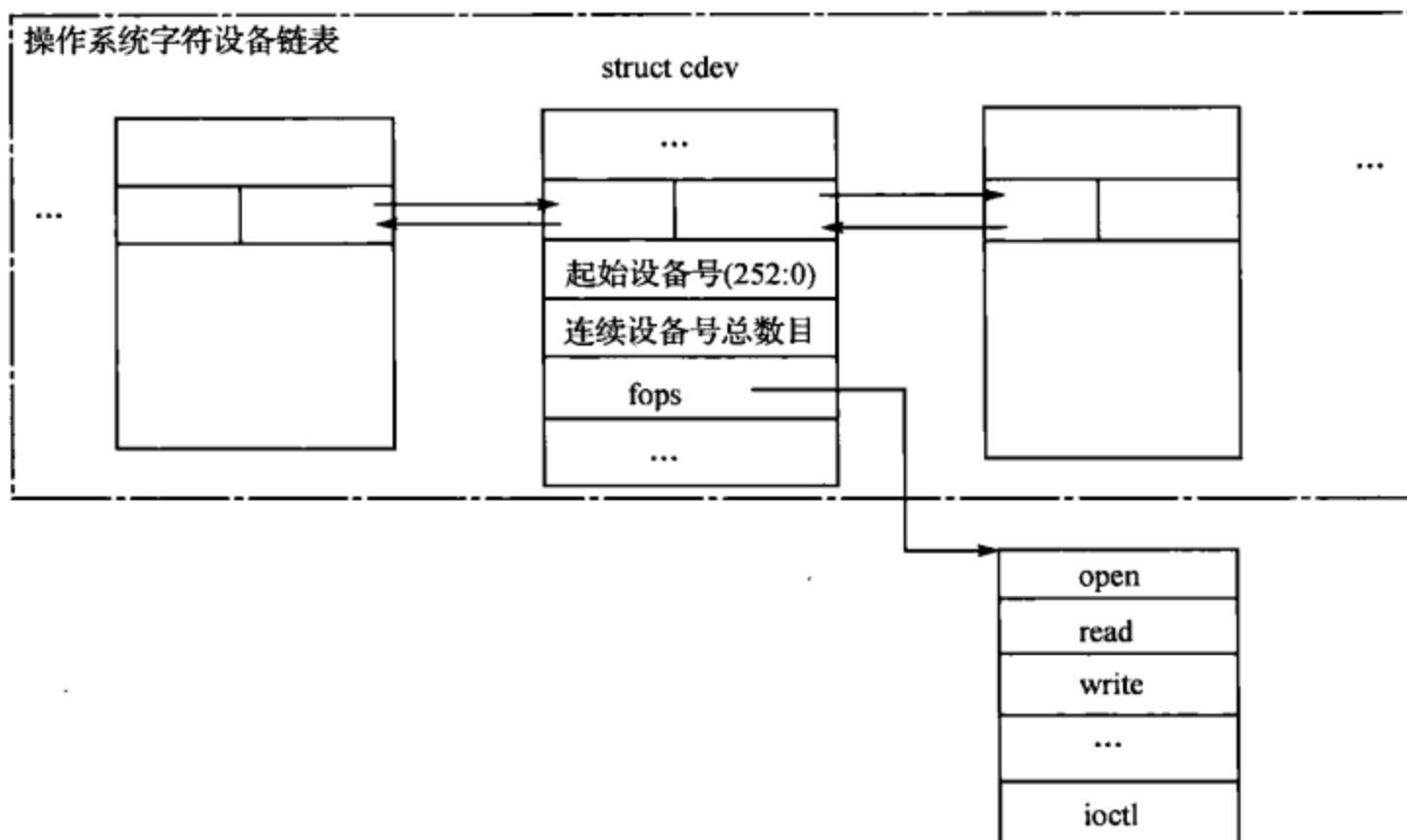


图 8-5 字符设备注册后

表: 文件描述符表 (file descriptor table)、文件表 (file table)、i 节点表 (i-node table)。关于 3 张表的关系和作用, 请参见《Linux Device Driver》第 3 版的 3.3 节。

i 节点表中含有:

- ① `i_rdev` 字段代表实际的设备号 (open 调用中设备文件对应的设备号)。
- ② `i_cdev` 字段指向字符设备 `cdev`。

文件表中含有:

- ① `f_op` 字段指向 `fops`。
- ② `f_pos` 字段表示设备的当前读写位置。
- ③ `f_flags` 字段标识文件打开时是否可读或可写。
- ④ `private_data` 字段指向私有数据指针, 驱动程序可以将这个成员用于任何目的或者忽略这个成员。

(4) 应用程序调用 `read(fd, buff)` 读取设备时, 操作系统做了什么?

操作系统根据 `fd` 和文件描述符表找到文件表, 再根据文件表中的 `f_op` 字段找到 `fops`, 而 `fops` (`scull_fops`) 存放的函数指针就是对应于操作物理设备 (例如 `read` 或 `write` 等) 的各类驱动函数的函数名, 这些函数就组成了驱动程序源代码的主体。所以操作系统就可以根据用户程序想执行 `read` 这个 system call, 在 `scull_fops.read` 中找到函数指针 `scull_read`, 进而调用它完成对物理设备的读操作。

所以我们写驱动程序很大一部分工作就是要实现这些直接操作设备硬件的函数。

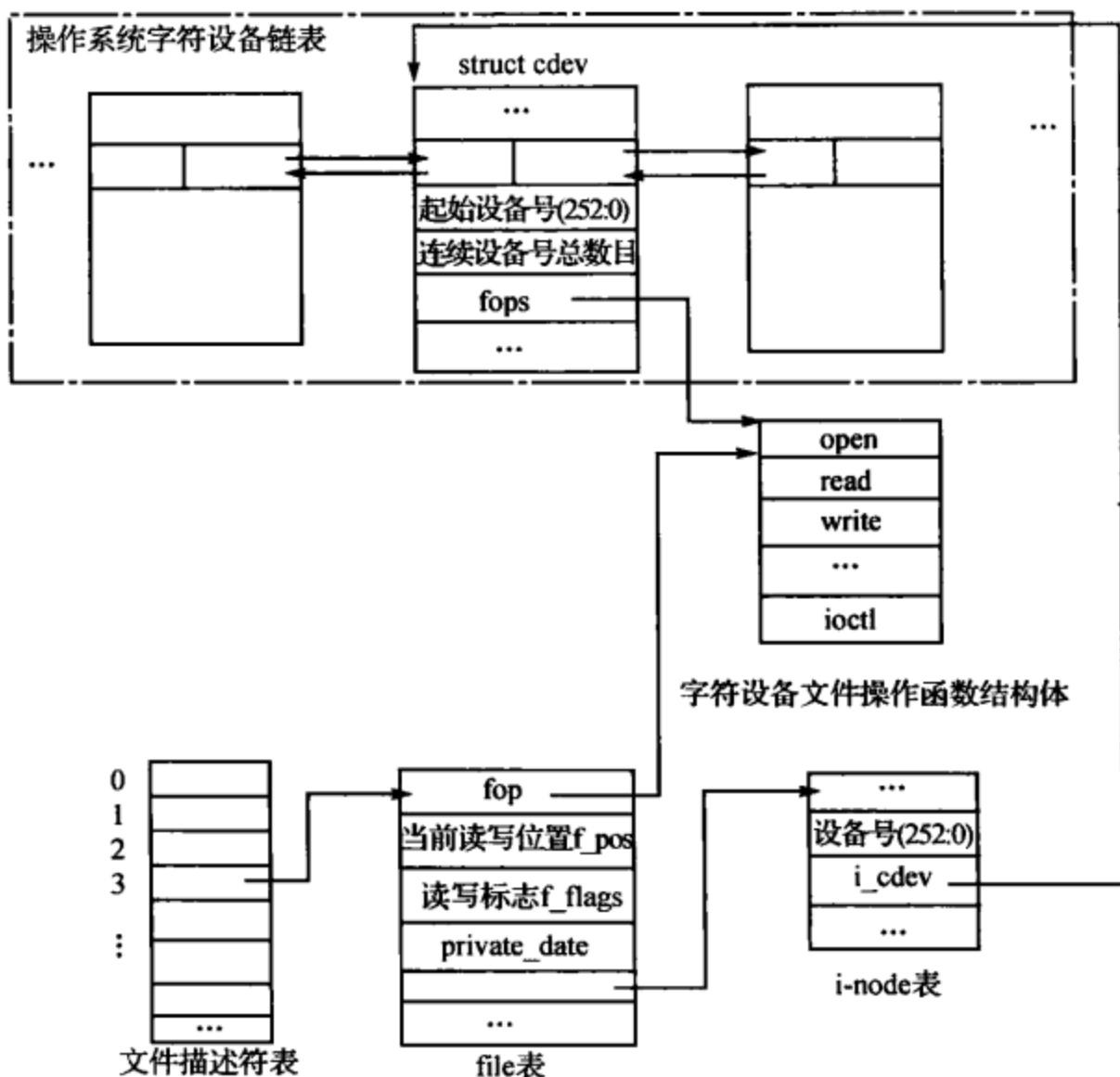


图 8-6 应用程序完成 open 调用后,内核的状况图

```

613 struct file_operations scull_fops = {
614     .owner = THIS_MODULE,
615 // .llseek = scull_llseek,
616     .llseek = no_llseek,
617     .read = scull_read,
618     .write = scull_write,
619     .ioctl = scull_ioctl,
620     .open = scull_open,
621     .release = scull_release,
622 };

```

file_operations 的主要字段:

- ① struct module * owner: 指向模块自身。
- ② open: 打开设备。
- ③ release: 关闭设备。

- ④ read:从设备上读数据。
- ⑤ write:向设备上写数据。
- ⑥ ioctl:操作设备函数。
- ⑦ llseek:定位读写指针。
- ⑧ mmap:映射设备空间到进程的地址空间。

4. 实现驱动中的重要功能函数

(1) 实现 open 函数:

```

257 int scull_open(struct inode * inode, struct file * filp)
258 {
259     struct scull_dev * dev; /* device information */
261     dev = container_of(inode->i_cdev, struct scull_dev, cdev);
265     filp->private_data = dev; /* for other methods */
267     /* now trim to 0 the length of the device if open was write-only */
268     if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
275         scull_trim(dev);
281     }
283     return 0;          /* success */
284 }

```

用户程序调用 open 时,操作系统会在建立并初始化好前述的 3 张表后,再调用驱动程序中的 scull_open 函数,传入的参数 inode 是 i 节点表指针,filp 是文件表指针。

261 行使用内核提供的宏 container_of,根据字符设备结构体 cdev 的地址(inode->i_cdev)推算出驱动程序定义的设备结构体地址(scull_devices),并将其赋给文件表中的 private_data 字段,以便驱动的其他功能函数将来比较容易地找到驱动程序定义的设备结构体。

268 行根据文件表中的读写标志(由操作系统已经根据用户程序 open 时指定的标志设置好了该标志),决定是否要清空设备中存放的数据。目前,这个驱动对于 echo zhu >> /dev/scull0 不能得到预定的结果。请思考,如何才能让 echo zhu >> /dev/scull0 能得到预定的结果?

出于演示的目的,所以本驱动中的 open 函数比较简单。其实 open 函数需要做的事情很多,总结如下:

① 模块使用计数加 1。

② 识别次设备号,如有必要更新 f_op 指针并调用新 f_op 指向的结构体中的 open 函数,以支持驱动拥有相同主设备号但却分属不同类型的设备(不同类型的设备驱动是不一样的)(这主要用于 misc 类型的设备。有兴趣的话,请参阅内核源码的 drivers/char/misc.c 中的 misc_open 函数、misc_register 函数和 drivers/char/watchdog/s3c2410_wdt.c 的 Line418 或

者阅读“字符设备驱动实战——内核 misc 设备框架分析”一节)。

③ 填写 `filp->private_data` 字段。

④ 硬件操作：

- 检查设备相关错误(诸如设备未就绪或类似的硬件问题)。
- 如果设备是首次打开,则对其初始化。
- 如果有中断操作,申请中断处理程序。

(2) 实现 `release` 函数。用户程序调用 `close` 时,操作系统一般都会调用驱动程序中的 `scull_release` 函数,传入的参数 `inode` 是 `i` 节点表指针, `filp` 是文件表指针。

出于演示的目的,所以本驱动中的 `release` 函数简单到不能再简单了。其实 `release` 函数需要做很多与 `open` 函数逆向的事情,总结如下:

① 模块使用计数减 1。

② 释放由 `open` 分配的,保存在 `filp->private_data` 中的所有内容。

③ 硬件操作：

- 如果申请了中断,则释放中断处理程序。
- 在最后一次关闭操作时关闭设备。

特别说明, `release` 函数被调用的时机:

④ 当文件表被释放时, `release` 函数被调用。

⑤ 当用户程序调用 `close`,但文件表并不被释放时(由于用户程序曾调用 `fork`、`dup`,从而导致 `FILE` 结构体引用计数 > 1), `release` 函数不会被调用。由此可见, `release` 函数与 `open` 函数的被调用次数,应该是相等的,等于设备被 `open` 的次数,但小于或等于被 `close` 的次数。

(3) 实现 `read` 函数。

```
323 ssize_t scull_read(struct file * filp, char __user * buf, size_t count,
324                    loff_t * f_pos)
325 {
326     struct scull_dev * dev = filp->private_data;
327     struct scull_qset * dptr; /* the first listitem */
328     int quantum = dev->quantum, qset = dev->qset;
329     int itemsize = quantum * qset; /* how many bytes in the listitem */
330     int item, s_pos, q_pos, rest;
331     ssize_t retval = 0;
349     if (*f_pos >= dev->size)
350         goto out;
351     if (*f_pos + count > dev->size)
352         count = dev->size - *f_pos;
354     /* find listitem, qset index, and offset in the quantum */
355     item = (long) *f_pos / itemsize;
```

```

356     rest = (long) * f_pos % itemsize;
357     s_pos = rest / quantum; q_pos = rest % quantum;
359     /* follow the list up to the right position (defined elsewhere) */
360     dptr = scull_follow(dev, item);
362     if (dptr == NULL || ! dptr->data || ! dptr->data[s_pos])
363         goto out; /* dont fill holes */
365     /* read only up to the end of this quantum */
366     if (count > quantum - q_pos)
367         count = quantum - q_pos;
369     if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
370         retval = -EFAULT;
371         goto out;
372     }
373     * f_pos += count;
374     retval = count;
376 out;
383     return retval;
384 }

```

当用户程序调用 read 时,操作系统会调用驱动中的 scull_read 函数。传入的参数 flip 指向文件表, buf、count 是用户程序调用 read 时传入的第 2、3 个参数,内核原封不动地把它们传给了驱动程序, f_pos 指向了文件表中的 f_pos 字段(表示设备的当前读写位置)。

326 行从文件表私有数据字段获得 scull_device 结构体地址以供稍后使用。

355 行计算出需要读取的数据位于第几个 quantum 集(item); 357 行计算出了需要读取的数据位于 quantum 集合中的第几个 quantum(s_pos) 以及 quantum 中的第几个字节(q_pos)。

最重要的是 369 行调用内核 API copy_to_user 将内核地址 dptr->data[s_pos] + q_pos 处开始的 count 个字节复制到用户空间 buf, 如图 8-7 所示。

373 行将文件表中记录设备当前读写位置的字段增加已经读写的字节数, 其目的是使下次读取设备时从本次读取的数据后接着读取。

374、383 行驱动程序将已成功读取的数据的总字节数返回给操作系统。操作系统将会把该值原封不动地返回给用户程序, 作为 read 函数的返回值。可见用户程序调用 read 时得到的返回值是多少, 最终取决于驱动程序实际读取的字节数。

关于 copy_to_user 的特别说明:

不能用

```

for (i = 0; i < count; i++)
    buf[i] = (dptr->data[s_pos] + q_pos)[i];

```

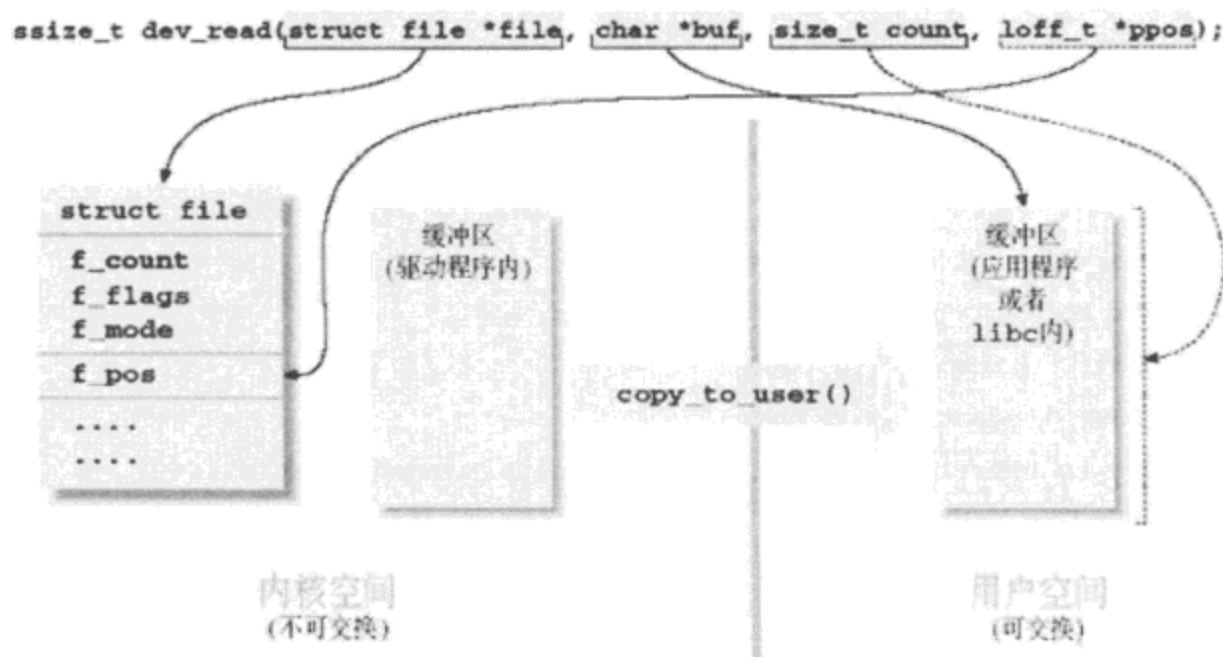



图 8-7 内核 API copy_to_user 示意图

替代 kernel API copy_to_user 的原因如下：

① 内核空间地址与用户空间地址，在不同的体系结构下的映射关系是不一样的。也就是说内核的地址 buf 与用户空间的地址 buf 未必是指同一个物理位置。

② 即使在某些体系结构下，其映射关系是一样的，也不能进行替换。因为，恶意用户会通过调用 read 时，指定恶意的 buf 地址，来达到破坏内核核心数据结构的目的。

③ 内核 API copy_to_user 会在复制前检查目的地址是否是用户程序可写的位置。

(4) 实现 write 函数。scull_write 函数代码在 386~447 行，它的参数和实现都与 scull_read 函数非常类似。最主要的区别是用 copy_from_user 替代了 copy_to_user，以完成从用户空间向内核地址复制数据的工作。

(5) 实现 llseek 函数。

```

583 loff_t scull_llseek(struct file *filp, loff_t off, int whence)
584 {
585     struct scull_dev *dev = filp->private_data;
586     loff_t newpos;
587     switch(whence) {
588         case 0: /* SEEK_SET */
589             newpos = off;
590             break;
591         case 1: /* SEEK_CUR */
592             newpos = filp->f_pos + off;
593             break;
594         case 2: /* SEEK_END */

```

```

598     newpos = dev->size + off;
599     break;
601     default: /* can't happen */
602         return -EINVAL;
604 }
605 if (newpos < 0) return -EINVAL;
606 filp->f_pos = newpos;
607 return newpos;
609 }

```

用户程序调用 `lseek` 时,操作系统会最终调用驱动的 `llseek` 函数。传入的参数 `flip` 指向文件表, `off` 和 `whence` 是用户程序调用 `lseek` 时传入的第 2、3 个参数,内核原封不动地把它传给了驱动程序。

594、606 行完成设备读写指针的移动。607 行将设备读写的新位置返回给操作系统,操作系统将原封不动地将这个值返回给应用程序。

课后作业:

① 按照 `scull_read` 的分析,完成 `scull_write` 的分析报告。

答:简化后的函数如下:

```

387 ssize_t scull_write(struct file * filp, const char __user * buf, size_t count, loff_t * f_
pos)
389 {
390     struct scull_dev * dev = filp->private_data;
404     /* find listitem, qset index and offset in the quantum */
405     item = (long) * f_pos / itemsize;
406     rest = (long) * f_pos % itemsize;
407     s_pos = rest / quantum; q_pos = rest % quantum;
409     /* follow the list up to the right position */
410     dptr = scull_follow(dev, item);
413     if (! dptr->data) {
414         dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
417         memset(dptr->data, 0, qset * sizeof(char *));
418     }
419     if (! dptr->data[s_pos]) {
420         dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
423     }
424     /* write only up to the end of this quantum */
425     if (count > quantum - q_pos)
426         count = quantum - q_pos;

```

```

428     if (copy_from_user(dp->data[s_pos] + q_pos, buf, count)) {
429         retval = -EFAULT;
430         goto out;
431     }
432     *f_pos += count;
433     retval = count;
434     /* update the size */
435     if (dev->size < *f_pos)
436         dev->size = *f_pos;
437 out:
438     return retval;
439 }

```

当用户程序调用 `write` 时,操作系统会调用驱动中的 `scull_write` 函数。传入的参数 `flip` 指向文件表, `buf`、`count` 是用户程序调用 `write` 时传入的第 2、3 个参数,内核原封不动地把它传给了驱动程序, `f_pos` 指向了文件表中的 `f_pos` 字段(表示设备的当前读写位置)。

390 行从文件表私有数据字段获得 `scull_device` 结构体地址以供稍后使用。

405~407 行计算出需要写入的位置位于第几个 quantum 集(item)中的第几个 quantum (`s_pos`)以及 quantum 中的第几个字节(`q_pos`)。410~423 为即将写入的位置申请空间。

425~426 说明每次只写到 quantum 尾,一次最多写一个 quantum 的大小。

428 行调用内核 API `copy_from_user` 将 `buf` 里的 `count` 个字节复制到内核地址 `dp->data[s_pos] + q_pos` 处。

最后更新 `*f_pos`(433 行), `dev->size`(437~438 行)和返回写入字节的个数(434、446 行)。

② 编写测试程序,以验证 `scull_llseek` 功能正确,提交测试程序源码和测试报告。

答:程序源代码参见光盘 `testllseekscull.c`。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <sys/types.h>
7 #include <sys/stat.h>
8 #include <fcntl.h>
9 int main()
10 {
11     int fd, retval;

```



```
12 char buf[2] = {0};
13 if ((fd = open("./scull0", O_RDWR)) < 0)
14 {
15     perror("open scull0\n");
16     exit(-1);
17 }
18 retval = write(fd, "1234567890", 10);
19 lseek(fd, 0, SEEK_SET);
20 read(fd, buf, sizeof(buf) - 1);
21 printf("%s\n", buf);
22 bzero(buf, sizeof(buf) - 1);
23 lseek(fd, 2, SEEK_CUR);
24 read(fd, buf, sizeof(buf) - 1);
25 printf("%s\n", buf);
26 bzero(buf, sizeof(buf) - 1);
27 lseek(fd, -2, SEEK_END);
28 read(fd, buf, sizeof(buf) - 1);
29 printf("%s\n", buf);
30 if ((retval = lseek(fd, -20, SEEK_CUR)) < 0)
31     printf("error 1 num is %d\n", retval);
32 else {
33     bzero(buf, sizeof(buf) - 1);
34     read(fd, buf, sizeof(buf) - 1);
35     printf("%s\n", buf);
36 }
37 if ((retval = lseek(fd, 10, SEEK_CUR)) < 0)
38     printf("error 2 num is %d\n", retval);
39 else {
40     printf("current position is %d\n", retval);
41     bzero(buf, sizeof(buf) - 1);
42     retval = read(fd, buf, sizeof(buf) - 1);
43     printf("have read %d chars, content is : %s\n", retval, buf);
44 }
45 close(fd);
46 }
```

运行结果如下:

```
/work/studydriver/examples/scull$ ./testlseekscull
```

```
1
```

数字水印
PDG

```

4
9
error 1 num is -1
current position is 19
have read 0 chars, content is :

```

③ 修改驱动程序,以实现 `echo zhu >> /dev/scull0` 的预期结果。提交报告说明修改了什么? 为什么那样修改?

答:

将 275 行 `scull_trim(dev); /* ignore errors */` 改为:

```

if (filp->f_flags & O_APPEND) {
    scull_llseek(filp, 0, 2);
} else {
    scull_trim(dev); /* ignore errors */
}

```

原因:

```

#define O_ACCMODE    00000003
#define O_RDONLY     00000000
#define O_WRONLY     00000001
#define O_RDWR       00000002
#define O_TRUNC      00001000
#define O_APPEND     00002000

```

`echo zhu >> /dev/scull0` 实现的是以追加和只写的方式(`O_WRONLY | O_APPEND`)将内容写入文件,因此应用程序会以 `O_APPEND` 和 `O_WRONLY` 标志调用 `open` 函数,此时操作系统在建立与本次 `open` 系统调用相对应的文件表条目时,会将文件表条目中的 `f_flags` 字段打上 `O_APPEND` 和 `O_WRONLY` 标志,然后调用底层 `scull_open` 驱动程序进行相应初始化工作。所以 `scull_open` 需要判断 `f_flags` 标志中是否设置了 `O_APPEND` 位。如果该位已被设置,则需要将文件表中表示当前文件读写位置的字段(`filp->f_pos`)更新至文件的末尾,而不是清空文件内容。

测试结果如下:

```

/work/system/scull$ echo 123 > scull0
/work/system/scull$ cat scull0
123
456
/work/system/scull$ echo 456 >> scull0
/work/system/scull$ cat scull0

```



```
123
456
/work/system/scull$ echo 456 > scull0
/work/system/scull$ cat scull0
456
```

其实这种做法也是有问题的。上面的测试结果正确,那是因为 shell 对于 `>>` 的解释是 `O_WRONLY | O_APPEND`。如果自行编写用户程序去执行 `open("./scull0", O_RDWR | O_APPEND)`,则结果不会正确。那么又应该如何更改驱动呢?这就留给大家继续思考吧。提示:既要更改 `scull_open`,还要更改 `scull_write`。更进一步,如何让驱动实现 `O_TRUNC` 标志呢?

8.3 驱动程序中的并发控制方法

8.3.1 并发控制原理简介

1. 驱动中产生并发控制需求的原因

一个硬件会并发的被多个进程使用,因此驱动中的读写函数会被不同的进程并发执行。例如 `scull` 设备就有可能在 A 进程正在执行 `scull_read` 函数(但尚未执行完)的时候,就被 B 进程打断,而 B 进程执行的是 `scull_write` 函数,当 A 进程再次被执行的时候它读到的东西就不再是它以前应该读到的东西。这还不是最严重的,如果是两个进程并发执行 `scull_write`,就有可能导致内存泄露以及丢失某个进程写入的数据。为什么会这样?请在分析 `scull_write` 函数的时候,假想 A 进程在执行该函数时有可能随时被 B 进程打断。要解决这个问题,就必须保证当 `scull_read` 或 `scull_write` 在执行时不被打断,这就需要并发控制。

2. 并发控制的原理

在任何给定时间只有一个线程可以执行的代码段被称为临界区(例如 `scull_write` 中不能被打断的代码)。

使用信号量和 P、V 操作来保护临界区。

一个信号量的核心是一个整型数,结合有一对函数 P 和 V。一个想进入临界区的进程将在相关信号量上调用 P:如果信号量的值大于零,这个值递减 1 并且进程继续;相反,如果信号量的值是 0(或更小),进程必须等待直到别人释放信号量。释放一个信号量通过调用 V 完成:这个函数递增信号量的值,如果需要(即:信号量的值为 1 或更大),唤醒等待的进程。

Linux 操作系统提供给驱动程序进行并发控制的手段主要有 5 种,最常用的是信号量和自旋锁:

- (1) 信号量。
- (2) 自旋锁。
- (3) 读/写信号量。
- (4) 读/写自旋锁。
- (5) Completions 机制。

8.3.2 信号量的编程实战

1. 信号量操作方法

- (1) 定义及初始化。普通信号量定义以及初始化：

```
struct semaphore  
void sema_init(struct semaphore * sem, int val);
```

互斥锁(2 值信号量)的声明与初始化：

```
DECLARE_MUTEX(name);  
DECLARE_MUTEX_LOCKED(name);
```

如果互斥锁必须在运行时间初始化使用下列宏：

```
void init_MUTEX(struct semaphore * sem);  
void init_MUTEX_LOCKED(struct semaphore * sem);
```

- (2) 信号量操作函数。

① void down (struct semaphore * sem), 递减信号量值, 如果需要则深度睡眠。它其实就是前面讲的 P 操作。

② int down_interruptible (struct semaphore * sem), 含义同上, 但是是浅度睡眠。

③ down_trylock 从不睡眠, 如果信号量在调用时不可用, down_trylock 立刻返回一个非零值。

④ void up(struct semaphore * sem); 一旦 up 被调用, 调用者就不再拥有信号量, 如有需要则唤醒睡眠在该信号量上的进程。它其实就是前面讲的 V 操作。

特别说明：

① ps 中用 D+ 表示深度睡眠, 用 S+ 表示浅度睡眠。

② 深度睡眠不能被信号中断(即使 SIGKILL 也不行); 浅度睡眠能被信号中断。

③ down_interruptible 返回值为 int, 可以通过返回值判定进程被唤醒的原因是由于其他进程执行了 up, 还是由于收到了信号; down 返回值为 void, 因为 down 不能被信号中断。

2. scull 设备驱动中是如何使用信号量进行并发控制的

- (1) 定义(位于 scull.h 中):

```

88 struct scull_dev {
89     struct scull_qset *data; /* Pointer to first quantum set */
95     struct semaphore sem;    /* mutual exclusion semaphore */
100    struct cdev cdev;        /* Char device structure */
101 };

```

(2) 初始化(位于 main.c 的驱动初始化函数中):

```

713 for (i = 0; i < scull_nr_devs; i++) {
717     init_MUTEX(&scull_devices[i].sem);
721     scull_setup_cdev(&scull_devices[i], i);
722 }

```

(3) 获得信号量(位于 main.c 的 scull_write 和 scull_read 函数中):

```

397 if (down_interruptible(&dev->sem))
398     return - ERESTARTSYS;

```

(4) 释放信号量:

```

441 up(&dev->sem);

```

3. 测试信号量进行并发控制的效果

(1) 增加测试代码(main.c),进行编译后,insmod scull.ko 去掉 333、345 行的注释:

```

333 ssleep(5);
345 ssleep(5);

```

(2) 写入数据:echo yang > ./scull0。

(3) 并发读出数据:

```

cat ./scull0 &
cat ./scull0

```

(4) 查看驱动的输出:tail /var/log/syslog。

```

1 pr  9 16:26:09 dennis - desktop kernel: [25131.489859] enter scull_open in process 11307
2 Apr  9 16:26:09 dennis - desktop kernel: [25131.509388] enter scull_open in process 11307
3 Apr  9 16:26:09 dennis - desktop kernel: [25131.545269] enter read in process 11307
4 Apr  9 16:26:11 dennis - desktop kernel: [25133.232751] enter scull_open in process 11308
5 Apr  9 16:26:11 dennis - desktop kernel: [25133.250795] enter scull_open in process 11308
6 Apr  9 16:26:11 dennis - desktop kernel: [25133.250860] enter read in process 11308
7 Apr  9 16:26:14 dennis - desktop kernel: [25136.567816] before get semaphore in process 11307
8 Apr  9 16:26:15 dennis - desktop kernel: [25136.568074] semaphore get, and begin sleep 5 second
in process 11307

```

```
9 Apr  9 16:26:16 dennis - desktop kernel: [25138.252521] before get semaphore in process 11308
10 Apr  9 16:26:19 dennis - desktop kernel: [25141.571120] end sleep in process 11307
11 Apr  9 16:26:19 dennis - desktop kernel: [25141.571247] release lock
12 Apr  9 16:26:19 dennis - desktop kernel: [25141.571318] semaphore get, and begin sleep 5 second
in process 11308
13 Apr  9 16:26:24 dennis - desktop kernel: [25146.573040] end sleep in process 11308
14 Apr  9 16:26:24 dennis - desktop kernel: [25146.573055] release lock
```

结合 `scull_read` 的代码分析上述输出,第 8~12 行很明显显示:进程 11308 必须要等待 5s,直到进程 11307 释放了信号量后才能获得信号量进入临界区。可见信号量很好地完成了并发控制的任务。

4. 读写信号量

读写信号量,在控制读和写不能并发执行的前提下,使得多个读可以并发执行。

定义: `struct rw_semaphore`

初始化: `void init_rwsem(struct rw_semaphore * sem);`

获得读信号量:

- `void down_read(struct rw_semaphore * sem);`
- `int down_read_trylock(struct rw_semaphore * sem);`
- 由于不存在 `down_read_interruptable`,因此可能导致深度休眠。

释放读信号量:

- `void up_read(struct rw_semaphore * sem);`

写信号量与读信号量相似。

8.3.3 自旋锁的编程实战

1. 自旋锁的特点及其与信号量的区别

(1) 一个自旋锁是一个互斥设备,只能有两个值:“上锁”和“解锁”。

(2) 如果锁是可用的,这个“上锁”位被置位并且代码继续进入临界区。通过 `spin_lock` 持有自旋锁期间,os 不会进行进程切换,只会响应中断,且在中断结束后,也不进行进程调度,而是回到持有自旋锁的进程。

(3) 相反,如果这个锁已经被别人获得,代码就进入一个紧凑的循环中反复检查这个锁,直到它变为可用。此时进程将持续占有 CPU,而不会进入休眠。这个循环就是自旋锁的“自旋”部分。

(4) 自旋锁常用于临界区代码很短的情况下,这样在多 CPU 的情况下,可以避免想要获得信号量的进程进入休眠继而又被立即唤醒。从而与信号量相比,更能提高系统效能。

(5) 在单 CPU 并且非抢占内核的情况下,自旋锁退化为空操作。

(6) 持有自旋锁的进程千万不要主动休眠(即:主动放弃 CPU。例如调用休眠函数 `ssleep`), 否则可能造成系统死锁。这是与信号量最大的区别之一。

2. 自旋锁操作方法

(1) 定义以及初始化。初始化可以在编译时完成, 代码如下:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者在运行时使用:

```
void spin_lock_init(spinlock_t * lock);
```

(2) 自旋锁操作函数。

```
void spin_lock(spinlock_t * lock), 获得锁
```

```
void spin_unlock(spinlock_t * lock), 释放锁
```

其他函数:

```
void spin_lock_irqsave(spinlock_t * lock, unsigned long flags);
```

```
void spin_lock_irq(spinlock_t * lock);
```

```
void spin_lock_bh(spinlock_t * lock);
```

```
int spin_trylock(spinlock_t * lock);
```

```
int spin_trylock_bh(spinlock_t * lock);
```

说明:

`spin_lock_irq` disable interrupts; `spin_lock_bh` disable interrupts bottom half (soft interrupt), but leave hardware interrupts enabled; (请参阅“中断顶半部与底半部”一节) `spin_lock_irqsave` 功能同 `spin_lock_irq`, 但要保存 interrupt 的 flag。

3. scull 设备驱动中是如何使用自旋锁进行并发控制的

(1) 定义(位于 `scull.h` 中):

```
88 struct scull_dev {
89     struct scull_qset * data; /* Pointer to first
90     spinlock_t spin;
91     struct cdev cdev;          /* Char device structure */
92 };
```

(2) 初始化(位于 `main.c` 的驱动初始化函数中):

```
713 for (i = 0; i < scull_nr_devs; i++) {
714     spin_lock_init(&scull_devices[i].spin);
715     scull_setup_cdev(&scull_devices[i], i);
716 }
```

资源解密
PDG

(3) 获得自旋锁(位于 main.c 的 scull_write 和 scull_read 函数中):

```
400 spin_lock(&dev->spin);
```

(4) 释放自旋锁:

```
443 spin_unlock(&dev->spin);
```

4. 测试自旋锁进行并发控制的效果(图 8-8、图 8-9、图 8-10)

(1) 增加测试代码,进行编译后,insmod scull.ko 去掉 scull.h87 行的注释:

```
87 #define usespin
```

(2) 写入数据:echo yang > ./scull0。

(3) 并发读出数据:

```
cat ./scull0 &
```

```
cat ./scull0
```

(4) 结果系统死锁,观察看到 vmware.exe 在双核 CPU 的机器上占有 50% 的 CPU 使用率。我们只能重新启动系统。



图 8-8 windows 机器是双核处理器

这证实了自旋锁的机制以及要特别注意的问题,参见:1. 自旋锁的特点及其与信号量的区别。

作业:请分析这个实验证实了 1 中自旋锁的哪些机制以及要特别注意的问题? 为什么说证实了那些机制和问题?

(5)重新启动系统后,增加测试代码(main.c)并编译后,insmod scull.ko 注释掉 345 行;去掉 346、347 行的注释:

```
345 // ssleep(5);
```

```
346 volatile long long tmp = 0xffffffff;
```

```
347 for (;tmp>0;tmp--);
```

(6)写入数据:echo yang > ./scull0。

(7)并发读出数据:

```
cat ./scull0 &
```

```
cat ./scull0
```

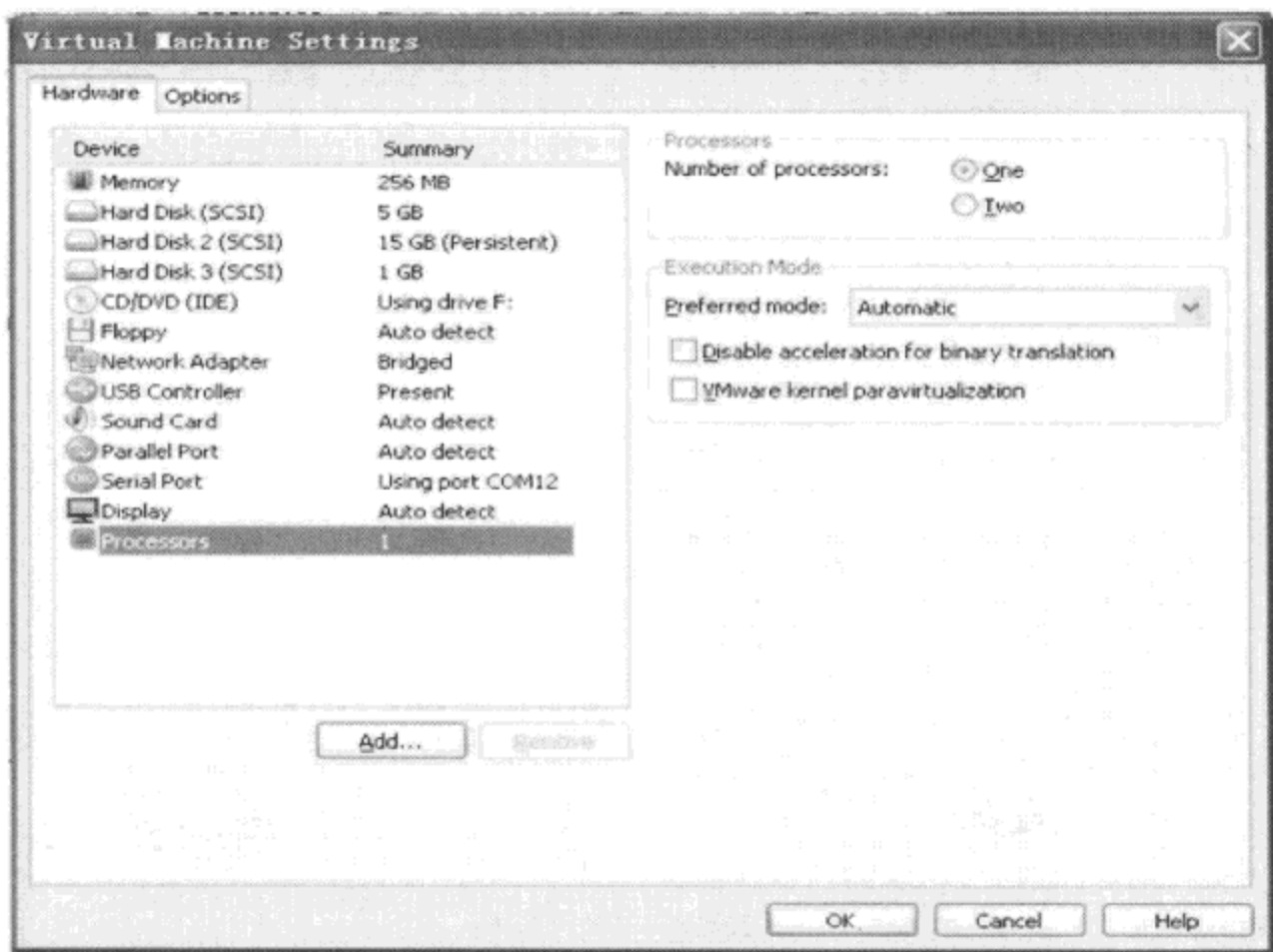


图 8-9 虚拟机使用一个 CPU

(8) 查看驱动的输出: `tail /var/log/syslog`。

结合 `scull_read` 的代码分析下述输出,第 4~11 行很明显显示:进程 11797 必须等待进程 11796 释放了自旋锁后才能获得自旋锁,才能进入临界区。可见自旋锁很好地完成了并发控制的任务。

```

1 Apr  9 17:07:26 dennis - desktop kernel: [27607.284734] enter scull_open in process 11796
2 Apr  9 17:07:26 dennis - desktop kernel: [27607.285429] enter read in process 11796
3 Apr  9 17:07:27 dennis - desktop kernel: [27608.663831] enter scull_open in process 11797
4 Apr  9 17:07:27 dennis - desktop kernel: [27608.664352] enter read in process 11797
5 Apr  9 17:07:32 dennis - desktop kernel: [27612.286445] before get semaphore in process 11796
6 Apr  9 17:07:32 dennis - desktop kernel: [27612.286522] semaphore get, and begin sleep 5 second
in process 11796
7 Apr  9 17:07:32 dennis - desktop kernel: [27613.512063] end sleep in process 11796
8 Apr  9 17:07:32 dennis - desktop kernel: [27613.519998] release lock
9 Apr  9 17:07:32 dennis - desktop kernel: [27613.520130] enter read in process 11796
10 Apr  9 17:07:33 dennis - desktop kernel: [27613.665567] before get semaphore in process 11797
11 Apr  9 17:07:33 dennis - desktop kernel: [27613.665576] semaphore get, and begin sleep 5 second
in process 11797

```

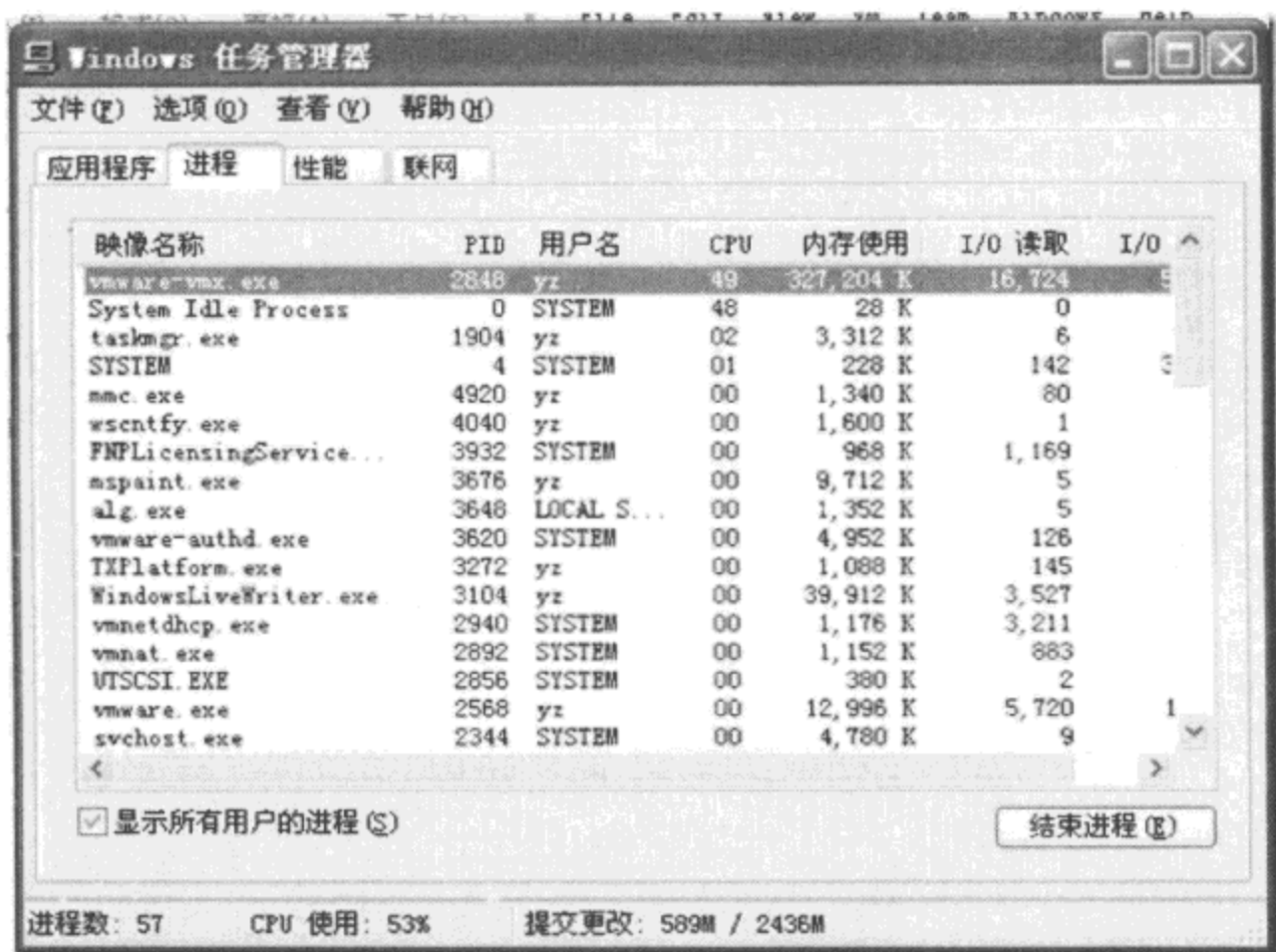


图 8-10 虚拟机使用了一个 CPU 的全部时间

```

12 Apr  9 17:07:33 dennis - desktop kernel: [27614.795063] end sleep in process 11797
13 Apr  9 17:07:33 dennis - desktop kernel: [27614.797880] release lock
14 Apr  9 17:07:33 dennis - desktop kernel: [27614.798073] enter read in process 11797
15 Apr  9 17:07:40 dennis - desktop kernel: [27618.520476] before get semaphore in process 11796
16 Apr  9 17:07:40 dennis - desktop kernel: [27618.520500] semaphore get, and begin sleep 5 second
in process 11796
17 Apr  9 17:07:40 dennis - desktop kernel: [27619.744942] end sleep in process 11796
18 Apr  9 17:07:40 dennis - desktop kernel: [27619.744981] release lock
19 Apr  9 17:07:40 dennis - desktop kernel: [27619.808457] before get semaphore in process 11797
20 Apr  9 17:07:40 dennis - desktop kernel: [27619.808463] semaphore get, and begin sleep 5 second
in process 11797
21 Apr  9 17:07:40 dennis - desktop kernel: [27621.084813] end sleep in process 11797
22 Apr  9 17:07:40 dennis - desktop kernel: [27621.084819] release lock

```

5. 读写自旋锁

读写自旋锁,在控制读和写不能并发执行的前提下,使得多个读可以并发执行声明和初始化:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */
```

```
rwlock_t my_rwlock; rwlock_init(&my_rwlock); /* Dynamic way */
```

操作方法:

```
void read_lock(rwlock_t * lock);
void read_lock_irqsave(rwlock_t * lock, unsigned long flags);
void read_lock_irq(rwlock_t * lock); void read_lock_bh(rwlock_t * lock); void read_unlock
(rwlock_t * lock);
void read_unlock_irqrestore(rwlock_t * lock, unsigned long flags); void read_unlock_irq(rwlock_
t * lock);
void read_unlock_bh(rwlock_t * lock);
```

8.3.4 Linux 内核提供的其他并发控制方法

1. 使用 Completions 机制的原因

信号量的同步机制

- struct semaphore sem;
- init_MUTEX_LOCKED(&sem);
- start_external_task(&sem);
- down(&sem)。

A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete.

在编写驱动时,经常会作如下的操作:在当前线程中初始化并启动另外的伙伴线程,并且当前线程必须等待被启动的伙伴线程帮助自己完成某些工作后,自己才能继续执行

Semaphores have been heavily optimized for the "available" case. When used to communicate task completion in the way shown above, however, the thread calling down will almost always have to wait; performance will suffer accordingly.

内核的信号量 API(例如:down、down_interruptible)针对可获得的情况(即:当想获取信号量时,以高概率获得信号量,而不是阻塞)进行了专门的优化。当按照上述程序的方式来使用信号量 API 完成工作,则会导致第 1 个线程几乎必然会阻塞在 down 调用处,因此会导致性能遭受极大的下降。

2. Completions 同步机制

(1) 定义、初始化:

```
struct completion my_completion; /* ... */
init_completion(&my_completion);
```

或者:DECLARE_COMPLETION(my_completion);

(2) 等待完成量:

```
void wait_for_completion(struct completion *c);
```

(3) 唤醒完成量:

```
void complete(struct completion *c);  
void complete_all(struct completion *c);
```

作业答案:

这个实验证实了 1 中的如下机制:

- (1) 一个自旋锁是一个互斥设备,只能有两个值:“上锁”和“解锁”。
- (2) 通过 spin_lock 持有自旋锁期间,OS 不会进行进程切换。
- (3) 相反,如果这个锁已经被别人获得,代码就进入一个紧凑的循环中反复检查这个锁,直到它变为可用。此时进程将持续占有 CPU,而不会进入休眠。
- (4) 持有自旋锁的进程千万不要主动休眠(即:主动放弃 CPU。例如调用休眠函数 `ssleep`),否则可能造成系统死锁。

分析:

```
cat ./scull0 &  
cat ./scull0
```

上面两个进程在并发访问临界资源时,进程“`cat ./scull0 &`”获得自旋锁资源且调用 `ssleep()` 休眠函数进入休眠状态,并释放占用的 CPU 资源,此时进程“`cat ./scull0`”将被 OS 调用获得 CPU 资源,之后其进入紧凑的自旋检测状态,检测“被进程‘`cat ./scull0 &`’使用的锁资源”是否被释放,这些说明自旋锁只有“上锁”和“解锁”两种状态,它是一个互斥设备;在“`cat ./scull0 &`、`cat ./scull0`”两个进程并发运行过程中查看任务管理器,任务管理器中类容显示(双核)CPU 资源的占用率达 50%(单核近 100%)之多,而且数值一直维持在这个数据上下,同时查看虚拟机配置,其内容显示只有一个处理器配置,这些数据说明进程“`cat ./scull0`”在被 OS 调度获得 CPU 资源之后,一直在占用 CPU 资源进行自旋检测,而不会进入睡眠状态;此时在进程“`cat ./scull0`”进行自旋检测期间,没有其他任何线程或进程能够获得 CPU 资源,来释放这个锁(包括进程“`cat ./scull0 &`”自身也不能得到 CPU 资源来进行锁的释放),一直维持这种“`cat ./scull0 &`”进程独占 CPU 资源自旋检测的僵持状态,也就出现了系统死锁现象,透过这种系统死锁现象,也能反映出:在自旋锁未被释放期间(也即:OS 认为是自旋锁持有期间),OS 不会进行进程切换。假设 OS 在此期间会进行进程切换,则这种系统死锁现象也可避免。上述分析也就验证了 1 中如上(实验报告)所列取的机制和需要特别注意的问题。

8.4 驱动程序中的阻塞与非阻塞编程

8.4.1 体验阻塞 I/O

1. 准备工作

准备工作包括加载模块；创建设备文件。

```
/work/studydriver/examples/scull $ sudo insmod scull.ko  
[sudo] password for dennis:  
/work/studydriver/examples/scull $ cat /proc/devices|grep scull  
252 scull  
252 scullp  
/work/studydriver/examples/scull $ sudo mknod scullp0 c 252 4
```

2. 进程工作

在设备 scullp0 中没有数据时 `cat ./scullp0`, 进程将阻塞等待数据；当 scullp0 中一旦有了数据, cat 进程就被唤醒并读出数据, 如图 8-11 所示。

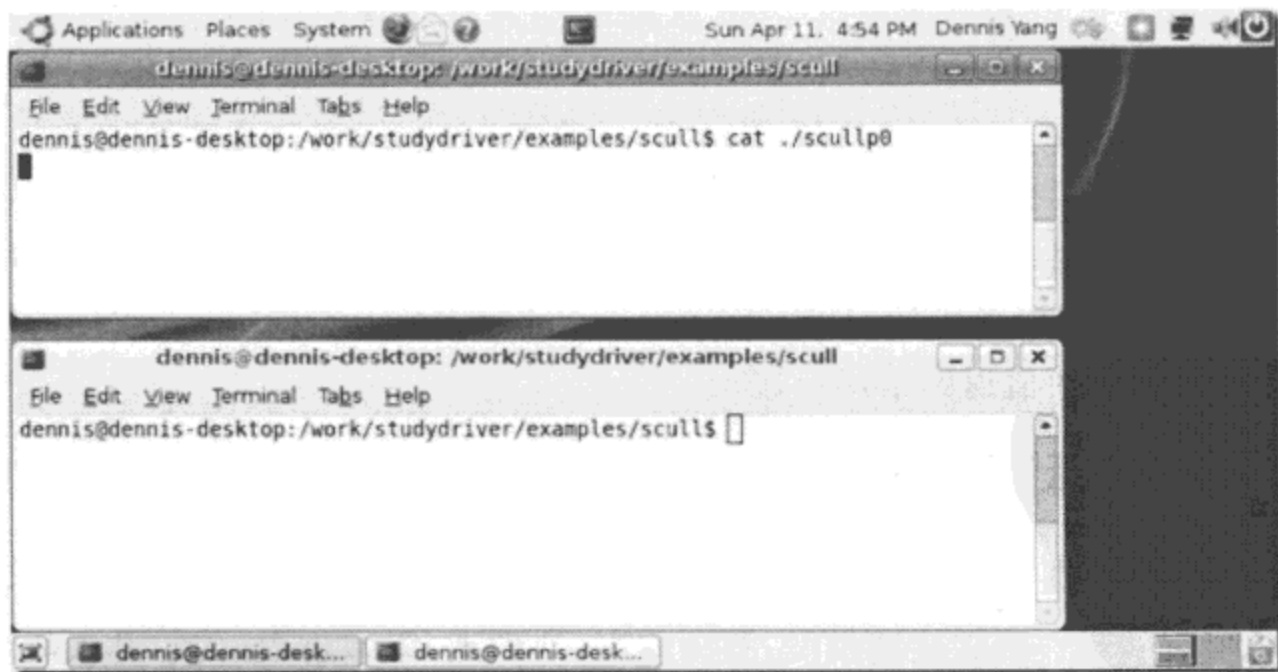


图 8-11 体验阻塞 I/O 图 1

在设备 scullp0 中没有数据时 `cat ./scullp0`, cat 进程将阻塞等待数据, 如图 8-12 所示。当 scullp0 中一旦有了数据, cat 进程就被唤醒并读出数据。

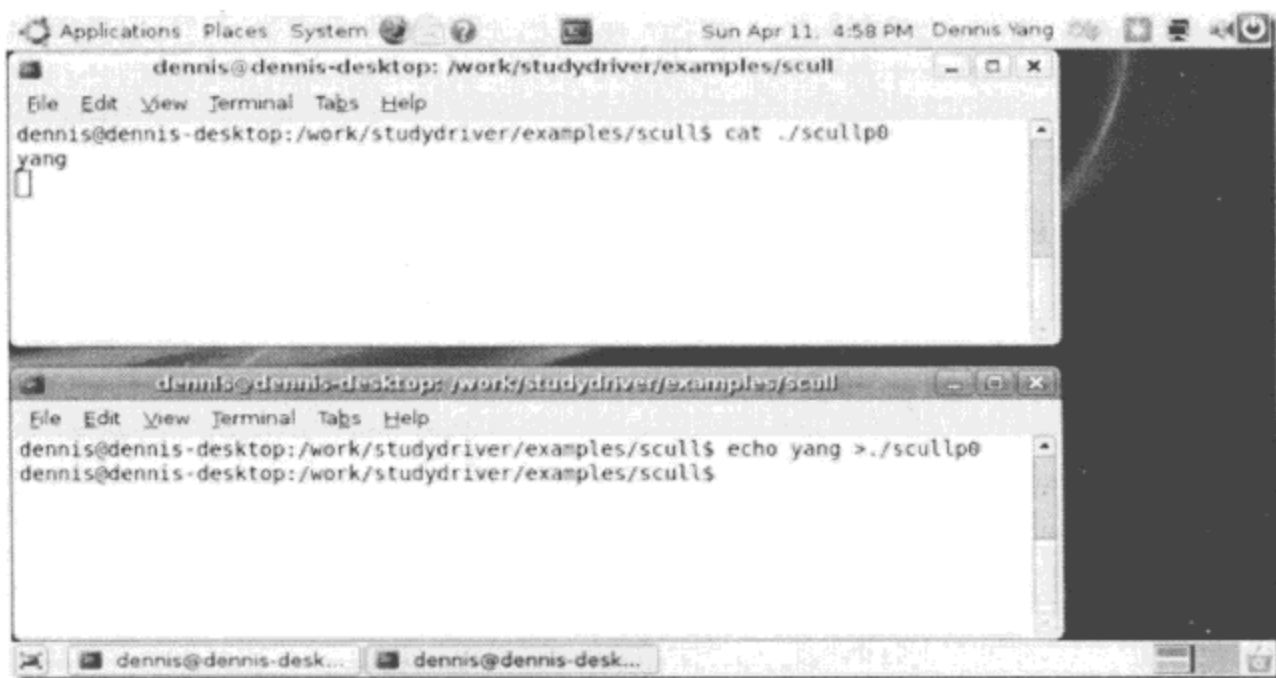


图 8-12 体验阻塞 I/O 图 2

3. 阻塞 I/O 的操作原则

(1) 如果一个进程调用 `read`, 但是没有数据可用, 这个进程必须阻塞。这个进程在有数据达到时被立刻唤醒, 并且那个数据被返回给调用者, 即便小于给 `read` 系统调用传入的 `count` 参数中请求的数量。实际被读取数据的字节数将作为 `read` 系统调用的返回值返回给用户进程。

(2) 如果一个进程调用 `write` 准备写入 `count` 字节, 并且在设备中没有空间, 这个进程必须阻塞, 并且它必须在一个与用作 `read` 的不同的等待队列中。当一些数据被从硬件设备读出, 从而使得设备中的空间变空闲, 那么这个写进程被唤醒并且 `write` 调用成功, 尽管数据可能只被部分写入 (如果在设备中的空闲空间不足 `count` 字节)。实际被写入数据的字节数将作为 `write` 系统调用的返回值返回给用户进程。

8.4.2 如何在驱动程序中实现阻塞 I/O

1. 实现原理

一个驱动当它无法立刻满足请求时, 驱动应当 (缺省地) 阻塞进程, 使它进入睡眠。直到请求可继续时再将它唤醒。

当驱动发现无法满足进程的请求时, 会调用内核 API 将该进程挂入等待队列 (等待链表) 后主动放弃 CPU, 从而使得进程睡眠; 当驱动发现已睡眠进程被唤醒的条件已经满足时, 将会唤醒睡眠进程, 其实质就是调用内核 API 将挂入等待队列的进程从等待队列中摘除, 然后调度该进程运行。

2. 需要调用的内核 API

(1) 定义以及初始化等待队列头。在 Linux 中,一个等待队列由一个“等待队列头”来管理,它是 `wait_queue_head_t` 类型的结构体。

一个等待队列头可使用如下方式来定义和初始化:

① 静态方式: `DECLARE_WAIT_QUEUE_HEAD(name);`

② 动态方式: `wait_queue_head_t my_queue; init_waitqueue_head(&my_queue);`

(2) 睡眠函数。当驱动发现无法立即满足用户进程请求时,驱动会调用睡眠函数 `wait_event(queue, condition)` 将用户进程阻塞。`wait_event` 函数会生成一个代表用户进程的等待队列节点,并将其链入等待队列头 `queue` 所表示的等待队列中,然后将进程的状态从 `TASK_RUNNING` 改为 `TASK_INTERRUPTIBLE` 或者 `TASK_UNTERRUPTIBLE`,然后调用 `schedule` 函数进行进程切换。`condition` 表示用户进程被唤醒(准确讲:是可以继续运行)的条件,其主要目的是要尽可能(但不可能完全)阻止惊群效应(即多个等待进程被同时唤醒,继而大多数又重新睡眠所带来的系统开销)。

除了 `wait_event` 外还有另外几个变体 API:

① `wait_event_interruptible(queue, condition);`

② `wait_event_timeout(queue, condition, timeout);`

③ `wait_event_interruptible_timeout(queue, condition, timeout);`

(3) 唤醒函数。当驱动发现可以满足睡眠用户进程的请求时,就会调用 `wake_up(wait_queue_head_t * queue)` 来唤醒挂在等待队列头 `queue` 上的睡眠进程们。不过特别要提请注意的是,其实 `wakeup` 函数只做了一件事情,就是通过遍历 `queue` 队列,将睡眠进程们的状态重新置为 `TASK_RUNNING`。至于调度睡眠进程重新运行是由下一次操作系统运行调度程序 `schedule` 的时候完成的,而将代表睡眠用户进程的节点从等待队列头 `queue` 所表示的等待队列中移除,则是由睡眠函数 `wait_event(queue, condition)` 的后部代码(`schedule` 之后的代码)完成的。参见(5)的描述。

(4) `sculnp` 的实现代码。

① 初始化等待队列头(位于模块初始化函数 `scull_p_init` 中):

```
362     scull_p_devices = kmalloc(scull_p_nr_devs * sizeof(struct scull_pipe), GFP_KERNEL);
367     memset(scull_p_devices, 0, scull_p_nr_devs * sizeof(struct scull_pipe));
368     for (i = 0; i < scull_p_nr_devs; i++) {
369         init_waitqueue_head(&(scull_p_devices[i].inq));    // 初始化读等待队列
370         init_waitqueue_head(&(scull_p_devices[i].outq));    // 初始化写等待队列
371         init_MUTEX(&scull_p_devices[i].sem);
372         scull_p_setup_cdev(scull_p_devices + i, i);
373     }
```

② 睡眠与唤醒：

```

124 static ssize_t scull_p_read(struct file * filp, char __user * buf, size_t count,
125     loff_t * f_pos)
126 {
132     while (dev->rp == dev->wp) { /* nothing to read */
137         if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
138             return - ERESTARTSYS; /* signal: tell the fs layer to handle it */
139         /* otherwise loop, but first reacquire the lock */
140         if (down_interruptible(&dev->sem))
141             return - ERESTARTSYS;
142     }
143     /* ok, data is there, return something */
148     if (copy_to_user(buf, dev->rp, count)) {
149         up(&dev->sem);
150         return - EFAULT;
151     }
152     dev->rp += count;
155     up(&dev->sem);
157     /* finally, awake any writers and return */
158     wake_up_interruptible(&dev->outq);
160     return count;
161 }

```

132 行发现无数据可读的情况下,将调用 `wait_event_interruptible` 将读进程投入睡眠(同时将读进程对应的等待队列节点链入读等待队列 `inq`), `dev->rp != dev->wp` 用于减小惊群效应。在 `scull_p_write` 函数唤醒读进程:

```

225     wake_up_interruptible(&dev->inq); /* blocked in read() and select() */

```

之后,读进程读取数据(148 行),然后唤醒写进程(158 行)。

尽管 `wait_event_interruptible` 已经尽力减少了惊群效应,但惊群效应的产生不可能完全避免,132、140、155 就用于处理惊群效应。例如两个读进程(A、B)同时被唤醒,A 进程通过 140 行获得信号量,当 B 进程试图在 140 行获得信号量时将睡眠;而 A 进程在将数据全部读完后,于 155 行唤醒 B 进程,当 B 进程再度运行通过 140 进行到 132 行时将发现无数据可读,将于 137 行再次调用 `wait_event_interruptible` 将自己投入睡眠。

(5)深入了解进程是如何睡眠及醒来的(即:`wait_event` 的实现,参见 `scull_p_write` 调用的 `scull_getwritespace` 函数)。

① 准备工作——分配及初始化一个等待队列:

- `wait_queue_head_t` 结构。

- 其由一个自旋锁和一个链表组成。

- 是 wait_queue_t(表征一个等待进程的结构体)的入口。

② 第一步——分配一个 wait_queue_t 结构,并用自身(current)初始化:

```
DEFINE_WAIT(wait);
```

③ 第二步——将表征自己的等待结构体挂入等待队列;同时设置进程的状态来标志自己为睡眠:

```
prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
```

④ 第三步——放弃 CPU:

```
schedule();
```

⑤ 第四步——被唤醒(或决定不睡眠)后,设置进程的状态来标志自己为运行(可能会出现重复设置,但不影响系统的功能);将表征自己的等待结构体从等待队列中摘出:

```
finish_wait(&dev->outq, &wait);
```

```
165 static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
166 {
167     while (spacefree(dev) == 0) { /* full */
168         DEFINE_WAIT(wait);
174         prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
175         if (spacefree(dev) == 0)
176             schedule();
177         finish_wait(&dev->outq, &wait);
182     }
183     return 0;
184 }
```

8.4.3 体验非阻塞 I/O

1. 非阻塞 I/O 的操作原则

(1) 如果一个进程调用 read,但是没有数据可读,这个 read 应立即返回而不是阻塞,read 的返回值为负数,并通过 errno 为 EAGAIN,表示需要再次读取。

(2) 如果一个进程调用 write 并且在设备缓冲中没有空间,这个 write 应立即返回而不是阻塞,write 的返回值为负数,并通过 errno 为 EAGAIN,表示需要再次写入。

2. 体验一下非阻塞 I/O

testreadscullp.c 如下:

```
1 #include <stdio.h>
2 #include <fcntl.h>
```



```
3 #include <errno.h>
4 #include <stdlib.h>
5 int main(int argc, char * * argv)
6 {
7     int fd, retnum;
8     char buf[100];
9     if ((fd = open("./scullp0", O_RDWR | O_NONBLOCK)) < 0) {
10         perror("open");
11         return -1;
12     }
13     while (1) {
14         retnum = read(fd, buf, 2);
15         if (retnum < 0) {
16             if (errno == EAGAIN)
17                 printf("have no data to read\n");
18             perror("read");
19             exit(-1);
20         }
21         write(1, buf, retnum);
22         write(1, "\n", 1);
23     }
24 }
```

testwritescullp.c 如下:

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <errno.h>
6 int main(int argc, char * * argv)
7 {
8     int fd;
9     if (argc != 2) {
10         printf("usage: ./testwritescullp content\n");
11         exit(-1);
12     }
13     if ((fd = open("./scullp0", O_RDWR | O_NONBLOCK)) < 0) {
14         perror("open");
15         exit(-1);
```

PDF

```

16     }
17     if (write(fd, argv[1], strlen(argv[1])) < 0) {
18         if (errno == EAGAIN)
19             printf("full, cant write data to device\n");
20         perror("write");
21         exit(-1);
22     }
23     printf("write: %s success\n", argv[1]);
24     return 0;
25 }

```

testreadscullp 与 testwritescullp 运行的结果如下:

1~3 行显示非阻塞读在没有数据时果然没有阻塞;4~10 行显示非阻塞写在设备没写满的情况下果然成功写入数据,非阻塞读在有数据时成功读出数据;12~18 行显示非阻塞写在设备已被写满的情况下果然没有阻塞;20~25 行显示非阻塞写在设备已被写满的情况下果然没有写入任何数据。

```

1 /work/studydriver/examples/scull$ ./testreadscullp
2 have no data to read
3 read: Resource temporarily unavailable
4 /work/studydriver/examples/scull$ ./testwritescullp abc
5 write: abc success
6 /work/studydriver/examples/scull$ ./testreadscullp
7 ab
8 c
9 have no data to read
10 read: Resource temporarily unavailable
11
12 /work/studydriver/examples/scull$ cat scull.h >./scullp0
13
14 ctrl+C
15
16 /work/studydriver/examples/scull$ ./testwritescullp abc
17 full, cant write data to device
18 write: Resource temporarily unavailable
19
20 /work/studydriver/examples/scull$ cat ./scullp0
21
22 .....
23

```

资源
分享
PDG

24 * S means "Set" through a ptr,
25 * T means "Tell" directly with

8.4.4 如何在驱动程序中实现非阻塞 I/O

非阻塞 I/O 由 `filp->f_flags` 中的 `O_NONBLOCK` 标志来指示。因此驱动只需要判明该标志是否设置,如果设置,说明用户进程以非阻塞的方式打开了该设备,如果该设备目前不能读取或写入,驱动就直接向操作系统返回 `EAGAIN` 即可,不需作任何其他操作。

```
124 static ssize_t scull_p_read(struct file * filp, char __user * buf, size_t count,
125     loff_t * f_pos)
126 {
132     while (dev->rp == dev->wp) { /* nothing to read */
134         if (filp->f_flags & O_NONBLOCK)
135             return -EAGAIN;
137         if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))

165 static int scull_getwritespace(struct scull_pipe * dev, struct file * filp)
166 {
167     while (spacefree(dev) == 0) { /* full */
171         if (filp->f_flags & O_NONBLOCK)
172             return -EAGAIN;
174         prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
```

作业:请在阅读本节、分析驱动程序代码和内核 API 源码后,给出各个主要内核 API 被调用后,内核内部以及驱动程序内部都产生了什么样的变化(最好画出示意图,更能说明问题)。并在此基础上,详细说明内核实现进程睡眠和唤醒进程的原理和过程。

8.5 字符设备驱动程序对一些高级特性的实现

8.5.1 non - seekable 的实现

1. non - seekable 设备的行为特点

由于 `scullp` 设备模拟的是管道,所以不可以对它调用 `lseek` 系统调用。对不可 `seek` 的设备调用 `lseek` 系统调用,`lseek` 将返回负值并将 `errno` 设为 `ESPIPE`。

2. 体验 non - seekable

`testlseekscullp.c` 如下:

```

8 int main(void)
9 {
10     int fd, retval;
11     if ((fd = open("./scullp0", O_RDWR)) < 0) {
12         perror("open");
13         exit(-1);
14     }
15     if ((retval = lseek(fd, 1, SEEK_SET)) < 0) {
16         if (errno == ESPIPE)
17             printf("error num is %d\n", errno);
18         perror("lseek");
19         exit(-1);
20     }
21     printf("lseek successful, current position is %d\n", retval);
22     return 0;
23 }
/work/studydriver/examples/scull$ ./testlseekscullp
error num is 29
lseek: Illegal seek

```

3. 驱动如何实现 non - seekable

驱动要实现设备的 non - seekable 特性,只需要执行以下两种操作中的任何一种即可。

(1) 在 open 功能函数中调用 nonseekable_open(inode, filp),告知 OS 本设备不支持 lseek。其实 nonseekable_open 将会设置 filp->f_mode 以向操作系统表明本设备不支持 seek 操作。

```

93     return nonseekable_open(inode, filp);
int nonseekable_open(struct inode * inode, struct file * filp)
{
    filp->f_mode &= ~(FMODE_LSEEK | FMODE_READ | FMODE_WRITE);
    return 0;
}

```

(2) 将 file_operations 中的 .llseek 设置为 no_llseek。其实 no_llseek 会向操作系统返回 ESPIPE。

```

321     .llseek =      no_llseek,
loff_t no_llseek(struct file * file, loff_t offset, int origin)
{
    return - ESPIPE;
}

```

}

8.5.2 select 的实现

1. select 的行为特点

应用程序可以通过 select 系统调用同时监控多个设备。当任意一个设备就绪(可读写)时,select 就返回并指明是哪一个设备就绪;如果没有任何设备就绪,select 将会阻塞(select 的第 5 个参数为 NULL)。

2. 体验 select(图 8-13~图 8-15)

testselectread.c(testselectsculp.c)如下:

```
9 int main(void)
10 {
11     int fd0, fd1, maxfd, retval;
12     fd_set readset;
13     char buf[100];
14     if (((fd0 = open("./scullp0", O_RDWR)) < 0) ||
15         ((fd1 = open("./scullp1", O_RDWR)) < 0)) {
16         perror("open scullp");
17         exit(-1);
18     }
19     maxfd = ((fd0 > fd1) ? fd0 : fd1);
20     FD_ZERO(&readset);
21     FD_SET(fd0, &readset);
22     FD_SET(fd1, &readset);
23     select(maxfd + 1, &readset, NULL, NULL, NULL);
24     if (FD_ISSET(fd0, &readset)) {
25         printf("scullp0 have data:");
26         if ((retval = read(fd0, buf, 100)) < 0) {
27             perror("read fd0\n");
28             exit(-1);
29         }
30         buf[retval] = '\0';
31         printf("%s\n", buf);
32     } else if (FD_ISSET(fd1, &readset)) {
33         printf("scullp1 have data:");
34         if ((retval = read(fd1, buf, 100)) < 0) {
35             perror("read fd1\n");
```




```

36         exit(-1);
37     }
38     buf[retval] = '\0';
39     printf(" %s\n", buf);
40 } else {
41     printf("select error\n");
42     exit(-1);
43 }
44

```

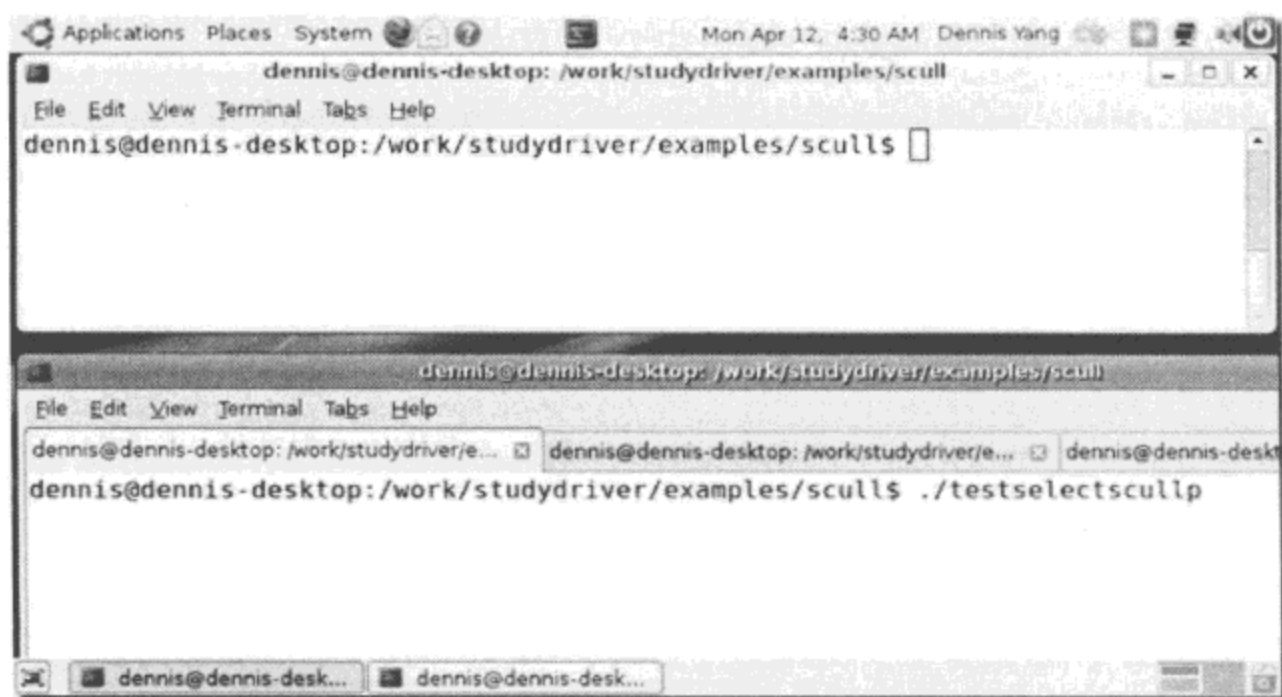


图 8-13 体验 select 1

3. 驱动如何实现 select

(1) 操作系统对 select 的实现

当用户程序调用 select 时,操作系统会依次轮询 select 中指定的所有文件描述符对应的设备,即:调用所有设备驱动程序中的 .poll 函数。

poll 函数会做两件事情:

- ① 告知操作系统如果设备暂时不可用的话,进程应该阻塞在哪个(或哪些)等待队列上。
- ② 根据设备的具体状况告知操作系统设备是否可用。

当有至少一个 poll 函数告知操作系统设备可用的话,select 返回。当所有 poll 都告知操作系统设备不可用时,操作系统会将进程阻塞在 poll 函数报告的所有等待队列上。当其中的任何一个等待队列发生了变化,从而导致该进程被唤醒后,操作系统将循环回去,再度依次轮询 select 中指定的所有文件描述符对应的设备。

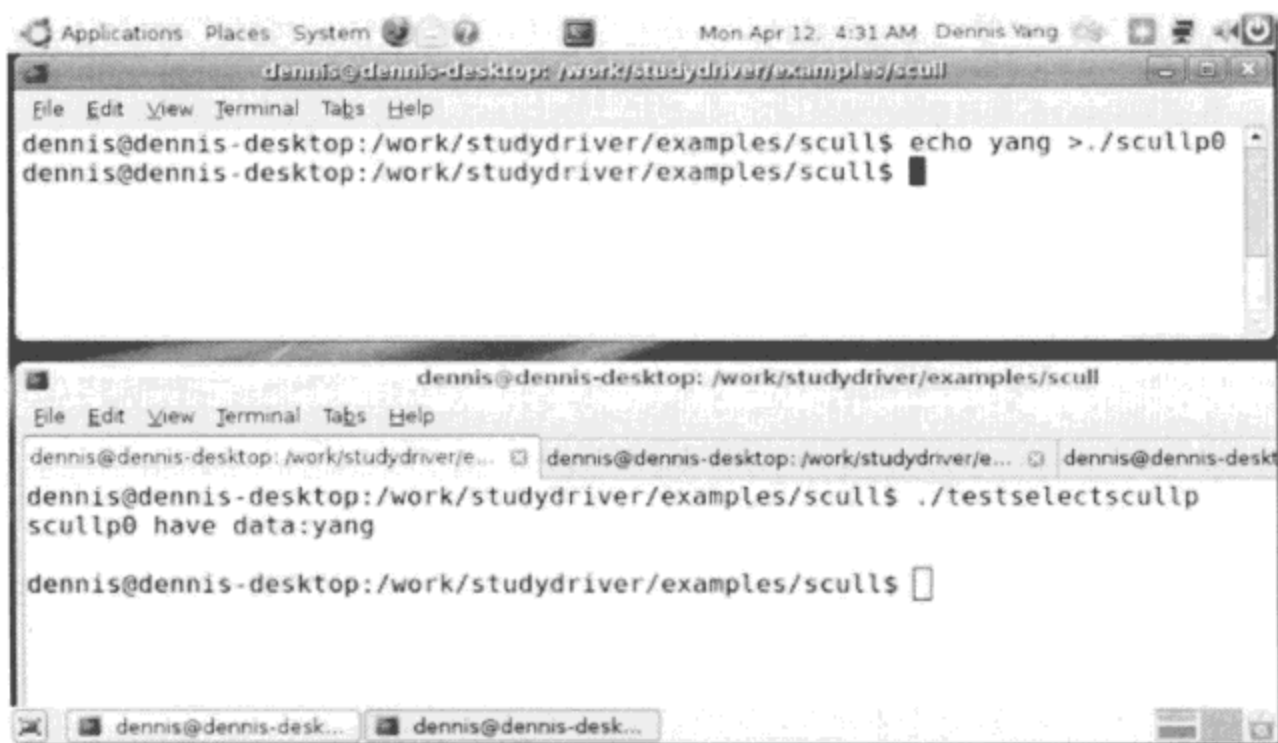


图 8-14 体验 select 2

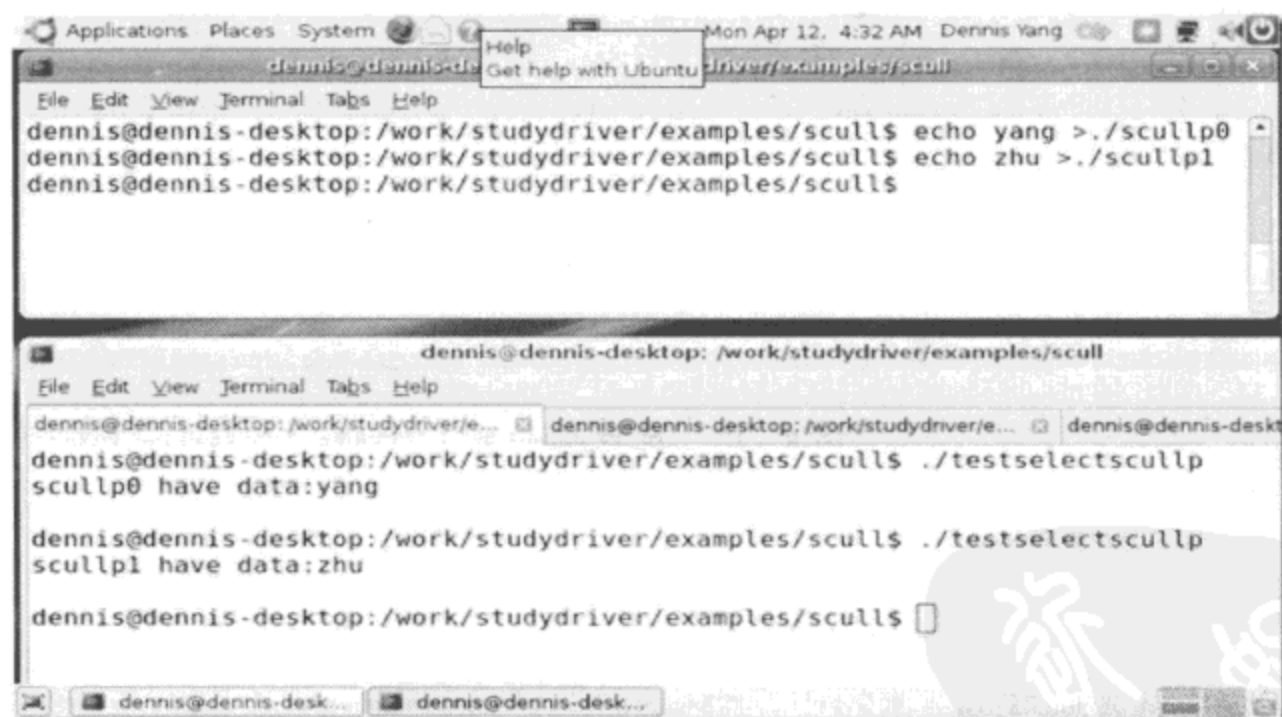


图 8-15 体验 select 3

(2) 驱动实现 poll 函数, 完成上述的两件事情:

- ① 246、247 行的 poll_wait 完成第一件事情。
- ② 249、251、254 行完成第二件事情。
 - POLLIN 表示可以无阻塞地读。
 - POLLRDNORM 表示有正常数据可读。

- POLLOUT 表示可以无阻塞地写。

- POLLWRNORM 与 POLLOUT 含义相同。

- 注意,对于数据到达设备末尾这种情况, poll 函数应当返回 POLLUP(hang-up),此时操作系统会向调用 select 的进程告知设备是可读的,如同 select 功能所规定的一样。不过这种情况不会发生在 scullp 设备中。

```

234 static unsigned int scull_p_poll(struct file * filp, poll_table * wait)
235 {
236     struct scull_pipe * dev = filp->private_data;
237     unsigned int mask = 0;
245     down(&dev->sem);
246     poll_wait(filp, &dev->inq, wait);
247     poll_wait(filp, &dev->outq, wait);
248     if (dev->rp != dev->wp)
249         mask |= POLLIN | POLLRDNORM;    /* readable */
250     if (spacefree(dev))
251         mask |= POLLOUT | POLLWRNORM;    /* writable */
252     up(&dev->sem);
254     return mask;
255 }

```

(3) 操作系统如何将进程阻塞在 poll 函数报告的所有等待队列上,如图 8-16 所示。

```
void poll_wait(struct file *, wait_queue_head_t *, poll_table *);
```

. poll 函数中的 poll_wait 会生成一个 poll_table_entry(其中包含指向文件表的指针 filp、指向等待队列头的指针 inq 或 outq、新建的代表本进程的等待队列节点),并将其链入 wait 指向的 poll_table 链表。当 poll 函数返回后,如果操作系统被告知没有任何设备可用,则操作系统会根据各个 poll_table_entry 中记录的等待队列头,将该各个 poll_table_entry 中包含的等待队列节点分别链入各自相应的等待队列,然后睡眠,等待被唤醒;如果操作系统被告知至少有一个设备可用,则操作系统先将各个 poll_table_entry 中包含的等待队列节点从各自的等待队列中删除,然后释放 poll_table 链表中的全部 poll_table_entry 后返回用户程序。

由于阻塞在 select 上的进程和阻塞在 read(write)上的进程有可能同时被唤醒(惊群效应),所以必须等到 read(write)进程读(写)完数据后才能够去判断设备是否可用(或者反之,判断完设备是否可用后,再运行读写进程),因此 245、252 行的并发控制(与 read、write 的并发控制相配合)就是必须的了。

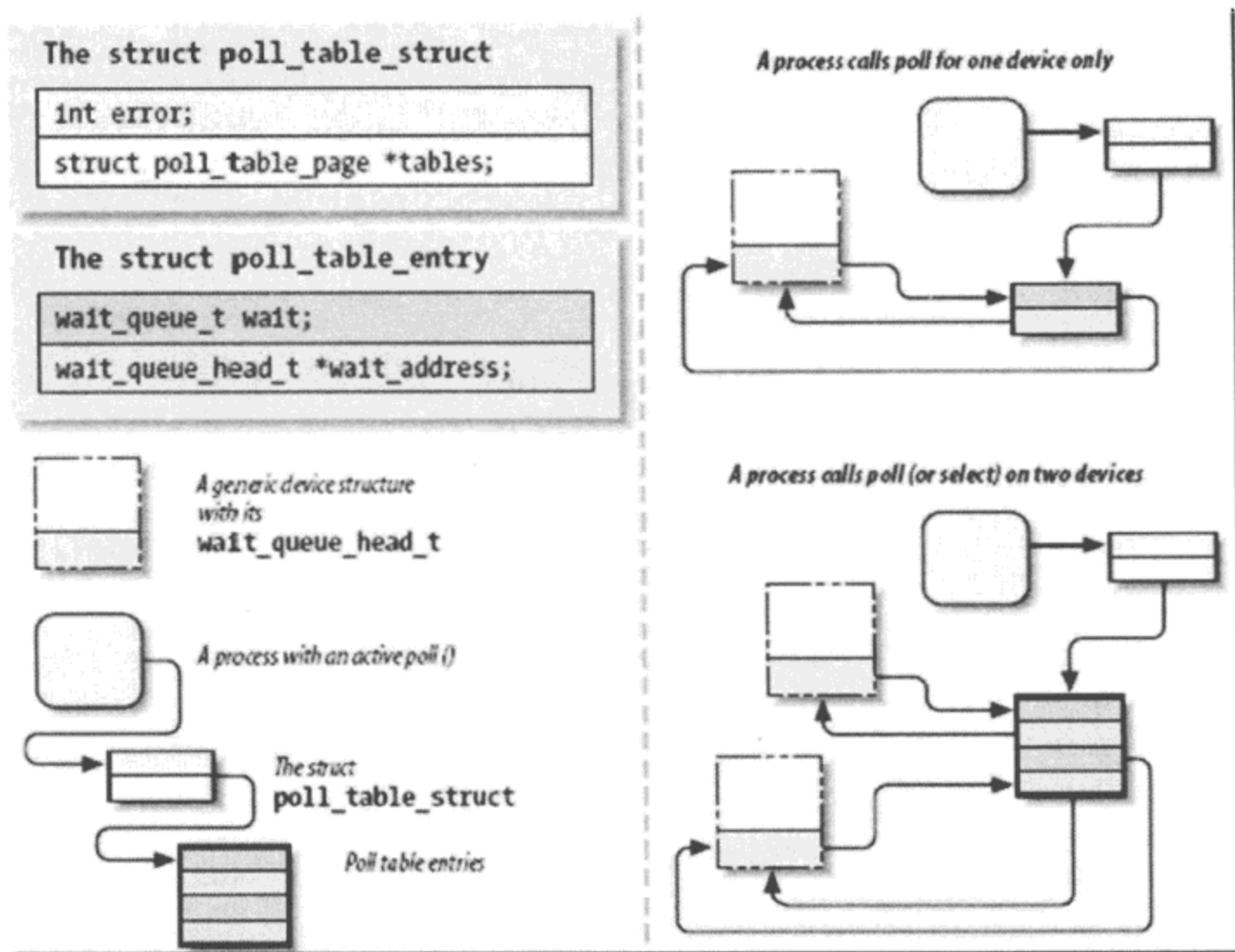


图 8-16 操作系统如何将进程阻塞在 poll 函数报告的所有等待队列上

第 9 章

Linux 字符设备驱动开发实战

9.1 I/O 内存与硬件通信

9.1.1 驱动中的内存分配

1. kmalloc

(1) 原型:

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

(2) 特点:

- ① 速度快。
- ② 不清零,分配的区仍然持有它原来的内容。
- ③ 物理内存中连续。
- ④ 用于分配的小内存,不能用于分配大内存。
- ⑤ 使用 kfree 释放分配的内存。

(3) flags 参数:

① GFP_KERNEL:最常用的内核内存分配选项,代表运行在内核空间的进程而进行分配内存,内部最终通过调用 __get_free_pages 来进行,能够使当前进程在少内存的情况下睡眠来等待一页。

② GFP_ATOMIC:用于中断处理、tasklet 和内核定时器(关于中断处理、tasklet 和内核定时器参见后文),不睡眠。

- ③ 可以与上述标志或操作的标志。
- ④ __GFP_DMA:在 DMA 区内分配内存。
- ⑤ __GFP_HIGHMEM:分配的内存可以位于高端内存。

(4) 内存区分为三类:

① 可用于 DMA 的内存:位于特别地址范围的内存,外设可以在这里进行 DMA 存取,在 x86, DMA 区用在 RAM 的前 16 MB。

② 普通内存。

③ 高端内存:用于存取大量内存。

(5) size 参数。

① 内核使用面向页的分配技术来最好地利用系统 RAM。

② 创建一套固定大小的内存对象池,处理分配请求时,进入一个持有足够大的对象的池子并且将整个内存块递交给请求者。

③ 分配小于 128 KB 的内存。

2. get_free_page

一个模块需要分配大块的内存,它最好是使用一个面向页的技术。为分配页,下列函数可用:

get_zeroed_page(unsigned int flags):返回一个指向新页的指针并且用零填充了该页。

__get_free_page(unsigned int flags):类似于 get_zeroed_page,但是没有清零该页。

__get_free_pages(unsigned int flags, unsigned int order):分配并返回一个指向一个内存区第一个字节的指针,内存区可能是几个(物理上连续)页长,但是没有清零。

① flags 参数同 kmalloc。

② order 是在请求的或释放的页数的以 2 为底的对数(即, $\log_2 N$),如果要一个页 order 为 0,如果请求 8 页就是 3。

当一个程序用完这些页,它可以使用下列函数之一来释放它们:

```
void free_page(unsigned long addr);
```

```
void free_pages(unsigned long addr, unsigned long order);
```

3. vmalloc

它在虚拟内存空间分配一块连续的内存区。尽管这些页在物理内存中不连续(使用一个单独的对 alloc_page 的调用来获得每个页),内核将它们作为一个连续的地址范围:

```
#include <linux/vmalloc.h>
```

```
void * vmalloc(unsigned long size);
```

```
void vfree(void * addr);
```

9.1.2 使用 I/O 端口地址空间与硬件进行通信的内核 API 介绍

1. 与硬件进行通信的原理

每个外设都是通过读写它芯片上的寄存器来控制。大部分时间一个设备有几个寄存器,

并且是在连续地址空间上存取它们,这个空间在内存地址空间或者在 I/O 地址空间。在硬件级别上,内存地址空间区域和 I/O 地址空间区域没有概念上的区别:它们都是通过在地总线和控制总线上发出电信号来存取(即,读写信号)并且读自或者写到数据总线。

2. 使用 I/O 地址空间与硬件进行通信的内核 API

(1) I/O 端口分配:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const char *name);
```

这个函数告诉内核,要使用 n 个端口,从 $first$ 开始, $name$ 参数是设备的名字,它会出现在 `/proc/ioports` 中。可以通过 `/proc/ioports` 接口查看系统中 I/O 端口的分配情况。

还有一个函数可以允许驱动检查一个给定的 I/O 端口组是否可用:

```
int check_region(unsigned long first, unsigned long n);
```

(2) I/O 端口释放。当用完一组 I/O 端口(在模块卸载时,也许)时,应当返回它们给系统使用:

```
void release_region(unsigned long start, unsigned long n);
```

(3) 单次操作 I/O 端口。在硬件请求了在它的活动中需要使用的 I/O 端口范围之后,驱动必须读且/或写到这些端口。为此,大部分硬件区别 8 位、16 位和 32 位端口。

- `unsigned inb(unsigned port);`
- `void outb(unsigned char byte, unsigned port);`

读或写字节端口(8 位宽)。 $port$ 参数在某些平台定义为 `unsigned long` 以及在其他上定义为 `unsigned short`。

- `unsigned inw(unsigned port);`
- `void outw(unsigned short word, unsigned port);`

这些函数存取 16 位端口(一个字宽)。

- `unsigned inl(unsigned port);`
- `void outl(unsigned longword, unsigned port);`

这些函数存取 32 位端口, $longword$ 声明为或者 `unsigned long` 或者 `unsigned int`。

(4) 重复操作 I/O 端口。

- `void insb(unsigned port, void * addr, unsigned long count);`
- `void outsb(unsigned port, void * addr, unsigned long count);`

读或写从内存地址 $addr$ 开始的 $count$ 字节,数据读自或者写入单个 $port$ 端口。

- `void insw(unsigned port, void * addr, unsigned long count);`
- `void outsw(unsigned port, void * addr, unsigned long count);`

读或写 16 位值到一个 16 位端口。

- void insl(unsigned port, void * addr, unsigned long count);
 - void outsl(unsigned port, void * addr, unsigned long count);
- 读或写 32 一位值到一个 32 位端口。

9.1.3 使用 I/O 内存地址空间与硬件进行通信的内核 API 介绍

I/O 内存实际位置是外设控制器芯片上的物理寄存器,其地址空间与普通内存进行统一编址,位于 0~4GB 的某个位置(如果是 32 位机的话)。程序对 I/O 内存进行读写操作就等于是对外设控制器芯片上的物理寄存器进行读写操作,从而就完成了对外设的驱动控制。ARM 体系结构下只有 I/O 内存,没有 9.1.2 小节所述的 I/O 端口,所以对外设的访问均使用 I/O 内存,而不是 I/O 端口。而在 X86 体系结构下存在 I/O 端口,所以在 X86 下可以访问 I/O 端口,但由于 I/O 端口并不是与内存进行统一编址的(它独立编址),所以 X86 下会提供一套有别于访存指令的 I/O 端口访问指令。

1. I/O 内存分配和释放

(1) I/O 内存区必须在使用前分配。分配内存区的接口是(在 `<linux/ioport.h>` 定义):

```
struct resource * request_mem_region(unsigned long start, unsigned long len, char * name);
```

这个函数分配一个 len 字节的内存区,从 start (物理地址)开始。如果一切顺利,一个非 NULL 指针返回;否则,返回值是 NULL。name 是申请到的区域名称,会在 `/proc/iomem` 中列出。可以通过 `/proc/iomem` 接口查看系统中 I/O 内存的分配情况。

(2) 当内存区不再需要时,应当释放:

```
void release_mem_region(unsigned long start, unsigned long len);
```

2. I/O 内存映射

I/O 内存分配和释放函数使用的是物理地址,而在 Linux 操作系统下都必须使用虚拟地址才能对内存进行合法访问,因此在使用申请到的 I/O 内存区之前,必须对该区域进行实地址到虚地址的映射成功后,才能使用虚地址对该区域进行访问。

```
#include <asm/io.h>
```

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

将长度为 size,起始地址为 phys_addr 的物理内存地址映射到虚拟地址,虚拟地址的首地址作为返回值返回。其本质是在 MMU 的页表中新建条目。

```
void * ioremap_nocache(unsigned long phys_addr, unsigned long size);
```

同 ioremap,区别在于不允许映射的内存区域的内容可在 CPU 的 cache 中缓存。其本质是在新建 MMU 的页表条目时,在该条目的相应域指定不可缓存。但由于对外设寄存器的映射都不应该允许缓存,所以内核对这两个 API 的实现是一样的。

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

取消 ioremap 建立的虚实地址映射。其本质是在 MMU 的页表中删除条目。

3. 简单存取 I/O 内存

从 I/O 内存地址 addr 处读取字节、半字、字：

```
unsigned int ioread8(void * addr);  
unsigned int ioread16(void * addr);  
unsigned int ioread32(void * addr);
```

向 I/O 内存地址 addr 处写入字节、半字、字：

```
void iowrite8(u8 value, void * addr);  
void iowrite16(u16 value, void * addr);  
void iowrite32(u32 value, void * addr);
```

4. 重复存取 I/O 内存

```
void ioread8_rep(void * addr, void * buf, unsigned long count);  
void ioread16_rep(void * addr, void * buf, unsigned long count);  
void ioread32_rep(void * addr, void * buf, unsigned long count);  
void iowrite8_rep(void * addr, const void * buf, unsigned long count);  
void iowrite16_rep(void * addr, const void * buf, unsigned long count);  
void iowrite32_rep(void * addr, const void * buf, unsigned long count);
```

5. 存取 I/O 内存的老接口

我们在编写驱动时,应该使用 3、4 介绍的新 API,但由于内核中有很多以前编写的驱动,它们使用的是老 API。我们需要能读懂老驱动,因此在这里也介绍一下老的 API。能猜出它们各自与哪些新 API 对应吧!

作业:请写出老的 API 与新的 API 的对应关系。

```
unsigned readb(address);  
unsigned readw(address);  
unsigned readl(address);  
void writeb(unsigned value, address);  
void writew(unsigned value, address);  
void writel(unsigned value, address);  
void __raw_writel(unsigned value, address);
```

9.1.4 通过 I/O 内存驱动硬件的实战——LED 灯驱动

在 ARM 体系结构与编程课程中,已经学习了如何在裸机(实地址)下驱动(点亮、熄灭)LED 灯,现在就来学习如何在 Linux 操作系统下驱动 LED 灯。将 LED 灯的驱动源码以及测

试程序(位于光盘\work\studydriver\ledsdriver)放到虚拟机 Linux 系统的/work/studydriver/ledsdriver 目录,然后执行 make 将驱动和测试程序进行编译。

1. 体验 LED 驱动

```
# mount -t nfs -o nolock 192.168.1.11:/work /mnt
# cd /mnt/studydriver/ledsdriver/
# insmod leds.ko
leds initialized
```

此时会看到 4 个 LED 灯全部被点亮。

```
# mknod /dev/leds c 400 0
# ./leds_test 3 off
leds conditon = 0x8
```

此时会看到第 4 个灯熄灭,其他 3 个灯仍然点亮。如果灯亮为 0,灯灭为 1 的话,此时的状况为 0b1000=0x8。

```
# ./leds_test 2 off
leds conditon = 0xc
# ./leds_test 3 on
leds conditon = 0x4
# rmmod leds
leds driver unloaded
```

此时会看到 4 个 LED 灯全部熄灭。

2. LED 灯驱动程序分析(leds.c)

(1) 设备初始化

133~142 申请设备号;146~152 为 LED 灯的自定义设备结构体分配空间并清 0;153 注册字符设备;155 申请 I/O 内存 0x56000010~0x5600001C,这个区域正是控制 LED 灯的 GPIO 控制器的寄存器物理地址;159 映射该 I/O 内存,得到的虚拟地址 virtaddr 就可被驱动程序用来读写该 I/O 内存,从而实现驱动 LED 灯;165 行可被分解为 166~171 行,其目的是初始化 GPBCON,将 GPB5~8 设置为输出引脚(详情参见 ARM 体系结构与编程的相关章节);174 行可被分解为 175~178 行,其目的是设置 GPBDAT 的 5~8bit 为 0,以点亮 4 个 LED 灯(详情参见 ARM 体系结构与编程的相关文章);184~189 则是在进行错误的善后处理工作,以使驱动更加健壮。顺便提一句,看到 goto 语句在驱动中是用来做什么的了!吧!

作业:请说出你对 goto 在驱动中的用法这个问题的看法。

特别说明:有的人认为由于物理内存 0x56000010~0x5600001C 实际已经存在,在 ARM 裸机编程中可以直接使用,因此现在只要进行虚实地址映射后就可使用,所以 155 行申请 I/O

内存的 API 没有任何意义,可以不要。这种看法是完全不对的,个人认为持这种看法的人是典型的自由主义者。打个比喻,假定物理存在一座房子(相当于物理内存 0x56000010~0x5600001C),你(相当于驱动)在不向房管局(相当于操作系统)申请(相当于调用 155 行的 API)并获得批准(相当于该 API 返回非 NULL)的情况下,可以住进去吗?当然可以!但是你不要忘了,此时房管局并不知道你住了该房子,在房管局的信息系统中显示的是该房子没有住人,所以当我(相当于另一个驱动)向房管局提出申请要住你现在住的房子的时候,会得到批准,于是我就会与你分享同一间房子(相当于两个驱动同时操控同一个硬件),这是不允许出现的情况。出现这样的情况,归根结底是你太自由主义化了,不遵守规则!所以,编程的时候,一定要调用 155 行的 API 向操作系统申请该 I/O 内存,并在操作系统同意的情况下(该 API 返回非 NULL),才能进行后续操作:映射和使用该 I/O 内存区。

随便说一下,155 行的 API 本质上执行的操作是:遍历操作系统的一个内部链表(称它为 I/O 内存链表吧),查看申请的内存是否已经分配。如果是,直接返回 NULL。反之,生成一个代表该内存的节点(含有内存地址范围、这个范围的名字等信息)并将它链入 I/O 内存链表,最后返回非 NULL(应该就是节点的地址)。我们 cat /proc/iomem 其实就是在遍历 I/O 内存链表,并显示每个节点的相关信息。

```

37 #define LEDS_MAJOR 400
40 #define GPBCON 0x56000010
42 struct leds_dev {
43     unsigned char value; /* When LED lighted, its value bit is 0, otherwise 1 */
44     struct cdev cdev;
45 };
47 static struct leds_dev * leds_devp;
48 static int leds_major = 400;
49 static void * virtaddr;

130 static int leds_init(void)
131 {
132     int result;
133     dev_t devno = MKDEV(leds_major, 0);
137     if (leds_major)
138         result = register_chrdev_region(devno, 1, "leds");
139     else {
140         result = alloc_chrdev_region(&devno, 0, 10, "leds");
141         leds_major = MAJOR(devno);
142     }
146     leds_devp = kmalloc(sizeof(struct leds_dev), GFP_KERNEL);
147     if (! leds_devp) {

```

新华书店
PDG

```
148     result = - ENOMEM;
149     goto fail_malloc;
150 }
152     memset(leds_devp, 0, sizeof(struct leds_dev));
153     leds_setup_cdev(leds_devp, 0);
155     if (! request_mem_region(GPBCON, 0xc, "qq2440_leds")) {
157         goto fail_request_mem_region;
158     }
159     if (! (virtaddr = ioremap(GPBCON, 0xc))) {
161         goto fail_ioremap;
162     }
165     iowrite32((ioread32(virtaddr) & ~(0xff<<(2 * 5))|(0x55<<(2 * 5)), virtaddr);
166 /*     tmp2 = ~(0xff<<(2 * 5));
167     tmp = ioread32(virtaddr);
168     tmp = tmp & tmp2;
169     tmp2 = 0x55<<(2 * 5);
170     tmp = tmp | tmp2;
171     iowrite32(tmp, virtaddr);
172 */
174     iowrite32(ioread32(virtaddr + 4)&~(0xf<<5), virtaddr + 4);
175 /*     tmp2 = ~(0xf<<5);
176     tmp = ioread32(virtaddr + 4);
177     tmp = tmp & tmp2;
178     iowrite32(tmp, virtaddr + 4);
179 */
182     return 0;
184 fail_ioremap:
185     release_mem_region(GPBCON, 0xc);
186 fail_request_mem_region:
187     kfree(leds_devp);
188 fail_malloc:
189     unregister_chrdev_region(devno, 1);
190     return result;
191 }

206 module_init(leds_init);
```

(2) 驱动(点亮或熄灭)LED 灯

当应用(测试)程序(leds_test.c)要点亮第 4 个 LED 灯时,将会执行:




```
32    ioctl(fd, IOCTL_LED_ON, led_no);
```

其实就是执行 `ioctl(fd, 0, 3)`, 表示要驱动执行的命令是 0 号命令(即点亮 LED 灯), 传给该命令的参数是 3(即第 4 个 LED 灯)。注: 某个编号的命令代表什么命令以及传给命令的参数代表什么含义, 一般而言是由驱动编写者在编写驱动时予以确定, 并编写一个头文件, 在其中定义将会在 `ioctl` 的第二个及后续参数使用的各种宏以及对这些宏的说明。最后将该头文件提供给应用程序编写者使用。

`ioctl` 的执行最终将导致操作系统调用驱动的 `leds_fops.ioctl` 函数(即 `leds_ioctl` 函数, 参见 `leds.c` 的第 114 行), 并将 0、3 这两个参数原封不动地传给该函数。`leds_ioctl` 函数将会检查传入的两个参数是否合法, 检查不通过, 将向操作系统返回对应的错误代码(94、105 行), 操作系统将向应用程序返回 -1, 并将 `errno` 设置为错误代码; 检查通过, 则将使用得到的参数去执行指定的命令(98、102 行)。

```
90 static int leds_ioctl(struct inode * inode, struct file * filp, unsigned int cmd, unsigned
long arg)
91 {
92     if (arg > 3)
93         return -EINVAL;
94     switch (cmd) {
95         case 0:
96             ledon(arg);
97             break;
98         case 1:
99             ledoff(arg);
100            break;
101        default:
102            return -ENOTTY;
103    }
104    return 0;
105 }
106
107 ledon(arg);
108 }
```

这样一来, 执行的就是“`ledon(3);`”。

`ledon` 函数的 76 行点亮第 4 个灯; 78 行修改 LED 灯的自定义设备结构体中的 `value` 字段, 以反映 4 个 LED 灯最新的亮灭状况, 供将来应用程序读取。

```
74 static void ledon(unsigned long arg)
75 {
76     iowrite32(ioread32(virtaddr + 4) & ~(0x1 << (arg + 5)), virtaddr + 4);
```

```

78     leds_devp->value &= ~(0x1<<arg);
80     return;
81 }

```

熄灭 LED 灯的程序与点亮 LED 灯类似。

至于 82 和 89 行是什么意思,就留待后文介绍吧。

```

82 EXPORT_SYMBOL(ledon);
89 EXPORT_SYMBOL(ledoff);

```

(3) 获得 LED 灯的亮灭情况

这是由驱动的 `leds_read` 函数完成的。

作业:请仿照 2(1)那样,简单分析一下 `leds_read` 函数的工作。指出在并发控制方面这个驱动存在什么不足,并设法展示出这个不足(可修改源代码),最后修正这个不足。

3. 完善该驱动

当我们将 LED 驱动从内核中卸载后,再加载的时候必须手工建立设备文件 `/dev/leds`。这时可用 `ledsv2.ko` 来完善。想知道怎么实现的吗?

其实很简单。参见“详解制作根文件系统”一节可知,只要在 `/sys` 目录下导出了有关设备的基本信息(主要就是设备号),`udev` 就可以自动在 `/dev` 目录下创建正确的设备文件。在 `/sys` 目录下导出了有关设备的基本信息是谁的责任呢?当然是驱动。

```

# ls /sys/class
graphics    misc        scsi_device  spi_master   usb_host
i2c-adapter mtd         scsi_disk    tty           vc
input       net         scsi_host     usb_device    vtconsole
mem         rtc         sound         usb_endpoint

# insmod ledsv2.ko
leds initialized

# ls /sys/class
graphics      mem           rtc           sound         usb_endpoint
i2c-adapter   misc          scsi_device   spi_master    usb_host
input         mtd           scsi_disk     tty           vc
leds_class    net           scsi_host     usb_device    vtconsole

# ls /sys/class/leds_class/
leds

# ls /sys/class/leds_class/leds
dev          subsystem    uevent

# cat /sys/class/leds_class/leds/dev
400:0

```

下面展示了在 ledsv2.c 中与 leds.c 主要的不同:

177 行将在 /sys 目录下导出一个类 leds_class; 180 行将在 /sys 目录下导出属于 leds_class 类的设备 leds 的基本信息, 其中就包含设备名称“leds”, 设备号 devno。

这之后, udev 就会自动在 /dev 目录下创建设备文件 leds 了, 并将它与正确设备号相关联。

232 行在 leds_class 类下删除设备号为 devno 的设备; 233 行删除设备类 leds_class。

```
28 #include <linux/device.h>      /* for sys mdev */
56 static struct class * leds_class;

152 static int leds_init(void)
153 {
177     leds_class = class_create(THIS_MODULE, "leds_class");
178     if (IS_ERR(leds_class))
179         printk(KERN_WARNING "failed in creating class\n");
180     class_device_create(leds_class, NULL, devno, NULL, "leds"); // devno 是设备号
220 }

222 void leds_cleanup(void)
223 {
232     class_device_destroy(leds_class, MKDEV(leds_major, 0));
233     class_destroy(leds_class);
235 }
```

4. 使用 misc(混杂)设备和 s3c24x0 BSP 提供的内核 API 简化 LED 驱动的编写

在 ledsv2.c 中有大量的代码用于生成控制 4 个 LED 灯引脚的虚拟地址, 并通过该虚拟地址配置引脚为输出, 设置引脚以点亮(或熄灭)LED 灯。其实 s3c24x0 BSP 已经映射产生了这些引脚的虚拟地址, 也提供了相应的内核 API 和宏来配置和设置这些引脚。在 ledsv3.c 使用了 BSP 提供的宏和内核 API 简化了 LED 驱动的编写。

代码 29~32 行使用了 BSP 定义的宏来表示 GPB5、6、7、8 这 4 个引脚; 36~39 行使用 BSP 定义的宏来表示 GPB5、6、7、8 这 4 个引脚的 output 状态。这些宏可在 include/asm-arm/arch-s3c2410/regs-gpio.h 中查到。

代码 114 行使用 BSP 提供的内核 API s3c2410_gpio_cfgpin 来设置 4 个引脚的状态为 output; 115、44、51 行使用 BSP 提供的内核 API s3c2410_gpio_setpin 来设置引脚的输出, 从而控制 LED 灯的亮灭。这两个内核 API 可在 arch/arm/plat-s3c24xx/gpio.c 中查到。

ledsv2.c 中还编写了大量代码来为 LED 申请字符设备号、初始化代表字符设备的 cdev、注册 cdev 以及在 SYS 文件系统中导出设备信息。其实, 内核中已经提供了混杂(misc)设备

的基础结构,可以利用该基础结构来简化简单字符设备的驱动编写。

在 ledsv3.c 中已经没有了申请字符设备号、初始化代表字符设备的 cdev、注册 cdev 以及在 SYS 文件系统中导出设备信息的代码。这是因为混杂(misc)设备的基础结构已经在操作系统中生成并注册了主设备号为 10 的 misc 字符设备,需要做的仅仅是将字符设备注册为 misc 设备中的一员,并建立字符设备的文件操作函数结构体与 misc 设备基础结构之间的联系即可。

代码 101~105 行定义的结构体是字符设备与 misc 设备基础结构之间的纽带,宏 MISC_DYNAMIC_MINOR 表示字符设备使用 misc 设备基础结构中能动态提供的次设备号,name 字段则是字符设备将来导出到 SYS 文件系统中的设备名,fops 字段则用于将字符设备的文件操作函数告知 misc 设备基础结构。使用 misc 设备基础结构导出的内核 API misc_register 在 119 行将它注册进 misc 设备基础结构,从而将字符设备注册为 misc 设备中的一员。相应的 128 行使用 misc_deregister 进行逆向操作。

关于 misc 设备基础结构的具体实现,请参阅“内核 misc 设备架构分析”一节。

```

26 static unsigned char value;      /* When LED lighted, its value bit is 0, otherwise 1 */
28 static unsigned long led_table[] = {
29     S3C2410_GPB5,
30     S3C2410_GPB6,
31     S3C2410_GPB7,
32     S3C2410_GPB8,
33 };
35 static unsigned int led_cfg_table[] = {
36     S3C2410_GPB5_OUTP,
37     S3C2410_GPB6_OUTP,
38     S3C2410_GPB7_OUTP,
39     S3C2410_GPB8_OUTP,
40 };
42 static void ledon(unsigned long arg)
43 {
44     s3c2410_gpio_setpin(led_table[arg], 0);
45     value &= ~(0x1<<arg);
46     return;
47 }
48 EXPORT_SYMBOL(ledon);
49 static void ledoff(unsigned long arg)
50 {
51     s3c2410_gpio_setpin(led_table[arg], 1);
52     value |= (0x1<<arg);

```

```
53         return;
54     }
55     EXPORT_SYMBOL(ledoff);
56
57     static int leds_ioctl(struct inode * inode, struct file * filp, unsigned int cmd, unsigned
long arg)
58     {
59         if (arg > 3)
60             return -EINVAL;
61         switch (cmd) {
62             case 0:
63                 ledon(arg);
64                 break;
65             case 1:
66                 ledoff(arg);
67                 break;
68             default:
69                 return -ENOTTY;
70         }
71         return 0;
72     }
73
74     static int leds_open(struct inode * inode, struct file * filp)
75     {
76         printk(KERN_INFO "in leds_open\n");
77         return 0;
78     }
79
80     static int leds_release(struct inode * inode, struct file * filp)
81     {
82         printk(KERN_INFO "in leds_release\n");
83         return 0;
84     }
85
86     static ssize_t leds_read(struct file * filp, char __user * buf, size_t count, loff_t * f_pos)
87     {
88         if (copy_to_user(buf, &value, 1))
89             return -EFAULT;
90         return 1;
91     }
92
93     static struct file_operations leds_fops =
```

```
93 {
94     .owner      = THIS_MODULE,
95     .read       = leds_read,
96     .ioctl      = leds_ioctl,
97     .open       = leds_open,
98     .release    = leds_release
99 };
101 static struct miscdevice misc = {
102     .minor = MISC_DYNAMIC_MINOR,
103     .name = "leds",
104     .fops = &leds_fops,
105 };
106
107 static int __init dev_init(void)
108 {
109     int ret;
110
111     int i;
112     for (i = 0; i < 4; i++) {
113         s3c2410_gpio_cfgpin(led_table[i], led_cfg_table[i]);
114         s3c2410_gpio_setpin(led_table[i], 0);
115     }
116     value = 0;
117     ret = misc_register(&misc);
118     printk("leds initialized\n");
119     return ret;
120 }
121
122 static void __exit dev_exit(void)
123 {
124     misc_deregister(&misc);
125     printk("leds unloaded\n");
126 }
127
128 module_init(dev_init);
129 module_exit(dev_exit);
```

9.1.5 驱动程序对 ioctl 的规范实现

在 LED 灯的驱动中,学习了如何在驱动中实现 ioctl 功能函数,但这种简化的实现存在一些潜在的隐患。例如,有两个设备,一个是 LED 灯,另一个是硬盘。这两个设备的驱动由不同的程序员编写。LED 灯驱动定义命令 0 表示亮灯,而硬盘驱动定义命令 0 表示格式化硬盘。

这样一来就存在潜在的威胁,例如,如果应用程序员本想通过 `ioctl(fd, 0)` 点亮 LED 灯,但在调用 `open` 的时候,误将硬盘的设备文件当成了 LED 灯的设备文件来打开,那么当应用程序执行 `ioctl(fd, 0)` 的时候就会导致硬盘被格式化。这个问题如何解决呢?最好的解决方案就是想办法让不同设备的命令的编号落在不同的区间上,例如 LED 的命令编号为 0~10,硬盘的命令编号为 20~100,以此类推,这样即使应用程序误将硬盘当作 LED 灯来打开,但由于其执行的是 `ioctl(fd, 0)` 或者 `ioctl(fd, 1)`,命令 0 和 1 均不是硬盘对应的命令,这样只会使得 `ioctl` 执行失败,而不会导致硬盘被格式化。

正是基于这样的考虑,内核编写者给出了一个规范,使得不同设备的命令的编号处于不同的区间,所以在编写驱动程序时应该遵循该规范。下面就来介绍该规范。

```
int scull_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
```

`ioctl` 的第 3 个参数(cmd)是命令编号,第 4 个参数(arg)是命令的参数。

1. 规范 `ioctl` 的第 3 个参数

(1) 规范将 cmd 的 32bit 从高位到低位分为 4 个部分:

① DIR:2 位。表示如果 `ioctl` 的第 4 个参数是指针的话,该指针所指向的数据是用于读还是写(读写是站在应用程序的角度而言)。

```
#define _IOC_NONE 0U
#define _IOC_WRITE 1U
#define _IOC_READ 2U
```

② SIZE:14 位。表示如果 `ioctl` 的第 4 个参数是指针的话,该指针所指向的数据的长度。

③ TYPE:8 位。表示统一分配的 `ioctl` 的类型编号。

```
#define SCULL_IOC_MAGIC 'k'
```

④ NR:8 位。表示自己定义的命令编号(都位于同一个 TYPE 下)

```
#define SCULL_IOCTLQUANTUM _IO(SCULL_IOC_MAGIC, 3)
```

(2) 系统头文件中定义的几个规范的宏:

```
#define _IOC(dir,type,nr,size) (((dir)<<30) | ((type)<<8) | ((nr)<<0) | ((size)<<16))
#define _IO(type,nr) _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,datatype) _IOC(_IOC_READ,(type),(nr),(sizeof(datatype)))
#define _IOW(type,nr,datatype) _IOC(_IOC_WRITE,(type),(nr),(sizeof(datatype)))
#define _IOWR(type,nr,datatype) \
    _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),(sizeof(datatype)))
```

这样一来 `#define SCULL_IOCTLQUANTUM _IO(SCULL_IOC_MAGIC, 3)` 使得命令编号 `SCULL_IOCTLQUANTUM` 为二进制的 00 00000000000000 字符 k 的 ASCII 码 00000011。由于字符 k 是专门分给 `scull` 驱动所使用(其他驱动不会使用 k),所以只要 `scull`

驱动自己在定义自己的命令编号(NR)时保证不冲突,则这个编号在所有驱动中都不会冲突。

2. 规范化的 ioctl 的第 3、4 个参数的应用(_IOC_TYPE、_IOC_NR 等都是内核定义的宏)

(1) if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;

可以通过比较 cmd 的 TYPE 段与分配给本驱动的 TYPE 段是否相等来确定应用程序传入的命令是否是本驱动支持的命令。

(2) if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

可以通过比较 cmd 的 NR 段是否大于本驱动支持的命令的总数量来确定应用程序传入的命令是否是本驱动支持的命令(当然前提是命令的编号从 0 开始顺次编号,并且宏 SCULL_IOC_MAXNR 是命令的总数量)

(3) if (_IOC_DIR(cmd) & _IOC_READ) err = ! access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));

可以通过查看 cmd 的 DIR 段来得知 ioctl 是否用来从内核读、写数据到用户空间,进而调用 access_ok 来测试该用户缓冲区是否安全可写(或可读)。

注:内核 API access_ok 用于测试用户缓冲区(arg 指向的内存区,大小为_IOC_SIZE(cmd))是否安全可写(或可读)。

(4) __get_user 利用 ioctl 的第 4 个参数。

```
switch(cmd)
case SCULL_IOC_SQUANTUM:
    if (!capable(CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_quantum, (int __user *)arg);
```

注:__get_usr 用于从 arg 所指向的用户内存区复制 sizeof(*arg)大小的数据到 scull_quantum,用于替换 copy_from_user;__put_usr 反之;capable(CAP_SYS_ADMIN)表示进程有没有管理员权限。

3. scull 驱动对 ioctl 规范化实现的代码分析

请查看 scull 驱动中的 scull_ioctl 函数的实现来体会 ioctl 的规范化实现;

请查看 scull 驱动中 scull.h 头文件以了解对 ioctl 各个命令的编号的规范化定义:

```
#define SCULL_IOC_SQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
#define SCULL_IOC_SQSET _IOW(SCULL_IOC_MAGIC, 2, int)
#define SCULL_IOC_TQUANTUM _IO(SCULL_IOC_MAGIC, 3)
#define SCULL_IOC_TQSET _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOC_GQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOC_GQSET _IOR(SCULL_IOC_MAGIC, 6, int)
```

```
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 7)
#define SCULL_IOCQOSET _IO(SCULL_IOC_MAGIC, 8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQOSET _IOWR(SCULL_IOC_MAGIC, 10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQOSET _IO(SCULL_IOC_MAGIC, 12)
```

9.2 内核 misc 设备架构分析

一点声明:本节使用 misc 设备指代 misc 设备基础结构中率先注册的主设备号为 10 的设备;使用 misc 字符设备指代使用 misc_register 注册到 misc 设备基础结构中的真实的字符设备。

内核已经提供了 misc 设备基础结构(其源代码在 drivers/char/misc.c 文件中),它预先生成了主设备号为 10 的 misc 设备,驱动可以让自己的 misc 字符设备(例如,前面讲到的 leds,后面要讲到的 watchdog)使用该主设备号下的某一个次设备号,将自己注册进 misc 设备基础结构。

内核 misc 设备基础结构主要完成以下几个操作。

9.2.1 定义全局变量

(1) 如图 9-1 所示,全局链表 misc_list 用于将使用 misc_register 注册的所有 misc 字符设备串接起来。结构体 miscdevice 代表一个 misc 字符设备,其中 minor 为次设备号, name 为 sys 文件系统中的设备名, fops 指向 misc 字符设备的文件操作函数结构体, list 用于链表串接。

```
56 static LIST_HEAD(misc_list);
struct miscdevice {
    int minor;
    const char * name;
    const struct file_operations * fops;
    struct list_head list;
    struct device * parent;
    struct device * this_device;
};
```

(2) 全局字符数组的每一个 bit 位(共 64bit)代表一个可供动态分配的次设备号(未分配为 0,已分配为 1)。可供动态分配的次设备号为 0~63。

```
62 #define DYNAMIC_MINORS 64 /* like dynamic majors */
```

```
63 static unsigned char misc_minors[DYNAMIC_MINORS / 8];
```



| | | | | | | | |
|------|------|------|------|------|------|------|------|
| 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00 |
|------|------|------|------|------|------|------|------|

全局数组misc_minors[8]
共64 bit,用于记录已动态分配的次设备号(0~63)

图 9-1 定义全局变量后

9.2.2 注册主设备号为 10 的 misc 设备

289 行生成并向操作系统注册了主设备号为 10 的 misc 设备,该设备仅定义了 open 函数,该函数用于将来 open 真正的 misc 字符设备时,修正真正的 misc 字符设备的 file 表。

285 行在 SYS 虚拟文件系统中建立了名为 misc 的类,以便在其下建立真正 misc 字符设备的信息,如图 9-2 所示。

```
276 static int __init misc_init(void)
277 {
285     misc_class = class_create(THIS_MODULE, "misc");
289     if (register_chrdev(MISC_MAJOR, "misc", &misc_fops)) { //MISC_MAJOR = 10
292         class_destroy(misc_class);
293         return -EIO;
294     }
295     return 0;
296 }
297 subsys_initcall(misc_init);

175 static const struct file_operations misc_fops = { //misc_fops 仅定义了 open 函数
176     .owner          = THIS_MODULE,
177     .open            = misc_open,
178 };

```

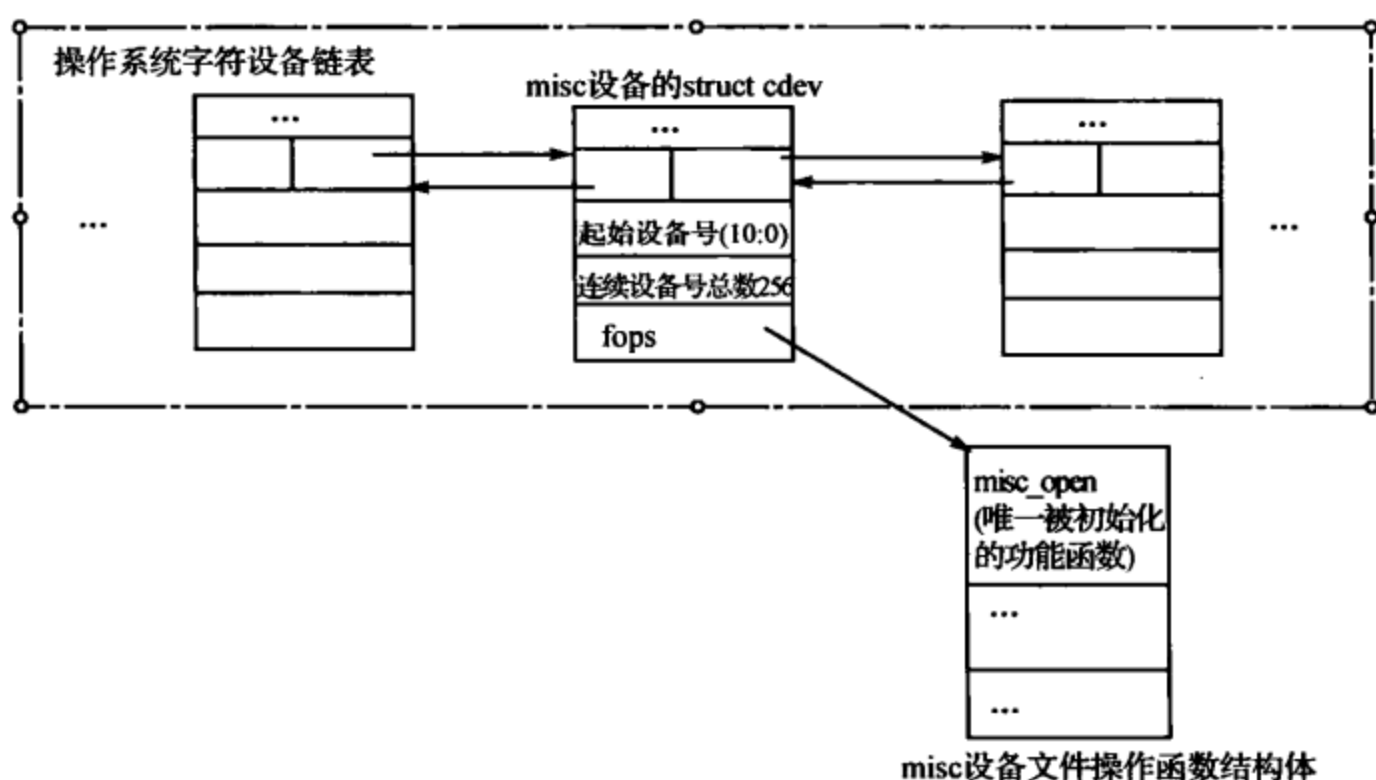


图 9-2 注册主设备号为 10 的 misc 设备后

9.2.3 导出内核 API——misc_register 函数

导出 misc_register 函数供其他驱动调用,来注册真实的 misc 字符设备。当别的驱动调用 misc_register 时,使用全局链表 misc_list:

判定是否已注册特定次设备号(206~211 行);

如果是使用动态次设备号注册(213 行),则从 63 开始向下查找未分配的次设备号(214~217),找到则分配之,并更新 miscdevice 结构体中的次设备号(222、225~226);

使用主设备号 10、miscdevice 结构体中的次设备号、设备名在 misc 类下创建 sys 文件系统下的设备(227、229)

将注册的 misc 字符设备链入全局链表 misc_list(240 行),如图 9-3 所示。

```

197 int misc_register(struct miscdevice * misc)
198 {
199     struct miscdevice * c;
200     dev_t dev;
203     INIT_LIST_HEAD(&misc->list);
206     list_for_each_entry(c, &misc_list, list) {
207         if (c->minor == misc->minor) {
209             return -EBUSY;
210         }
211     }

```

```

213     if (misc->minor == MISC_DYNAMIC_MINOR) {
214         int i = DYNAMIC_MINORS;
215         while (--i >= 0)
216             if ((misc_minors[i >> 3] & (1 << (i & 7))) == 0)
217                 break;
218         if (i < 0) {
219             return -EBUSY;
220         }
221         misc->minor = i;
222     }
223     if (misc->minor < DYNAMIC_MINORS)
224         misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
225     dev = MKDEV(MISC_MAJOR, misc->minor);
226     misc->this_device = device_create(misc_class, misc->parent, dev,
227                                     "%s", misc->name);
228     list_add(&misc->list, &misc_list);
229 }

```

```

273 EXPORT_SYMBOL(misc_register);

```

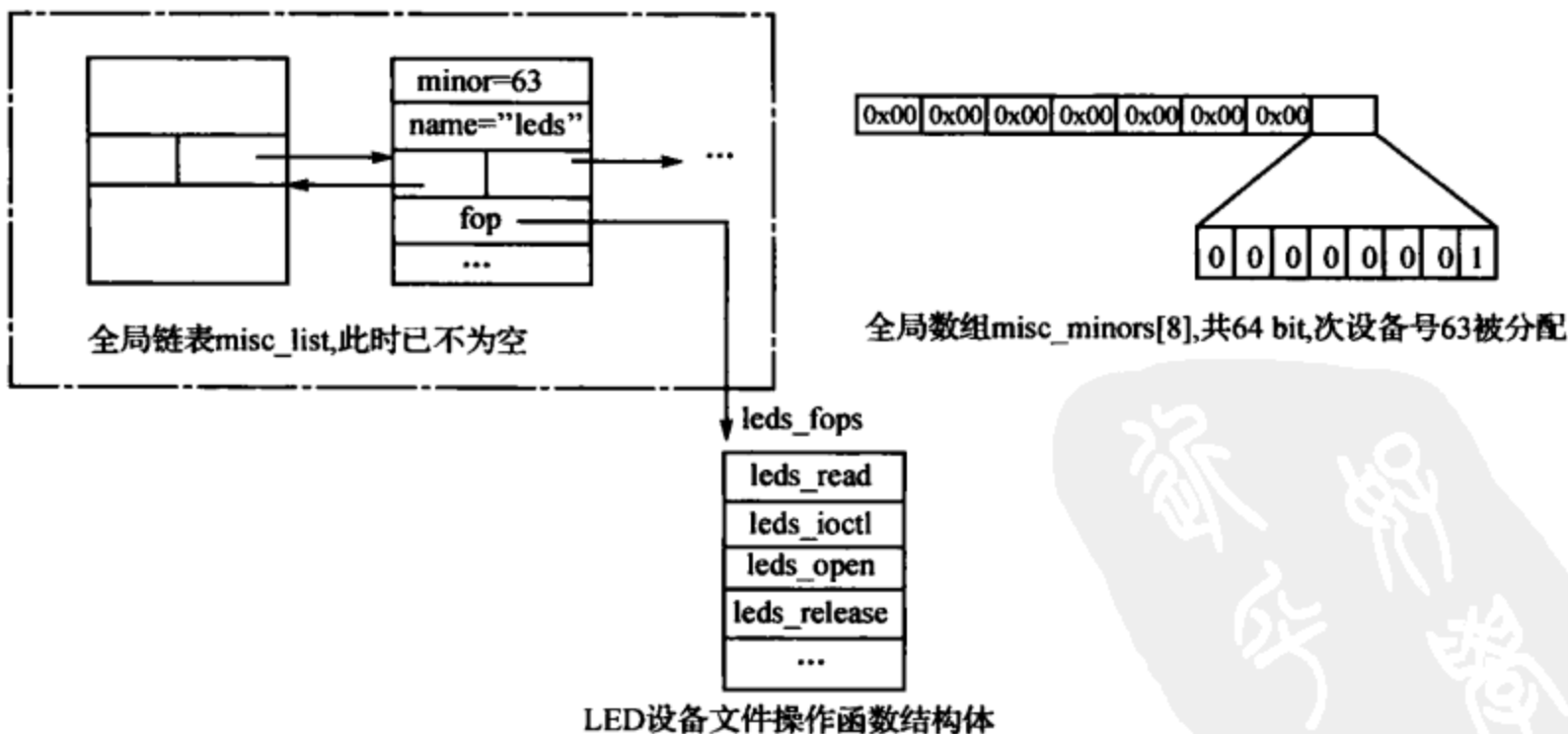


图 9-3 LEDs 驱动调用 misc_register 后(将 leds 这个 misc 字符设备注册进 misc 设备链表)

9.2.4 实施“乾坤大挪移”的 misc 设备 open 函数

open misc 字符设备时,巧借 misc 设备的 open 函数,修正 misc 字符设备的 file 表。

当应用程序 open misc 字符设备时,操作系统会通过主设备号 10 在操作系统的字符设备链表中找到 misc 设备,进而为 misc 字符设备,使用主设备号 10 和次设备号建立 i-node 表,使用 misc 设备的 fops 建立 file 表。请注意此时的 file 表 fops 字段是错误的,它是 misc 设备的 fops,而不是 misc 字符设备的 fops,如图 9-4 所示。

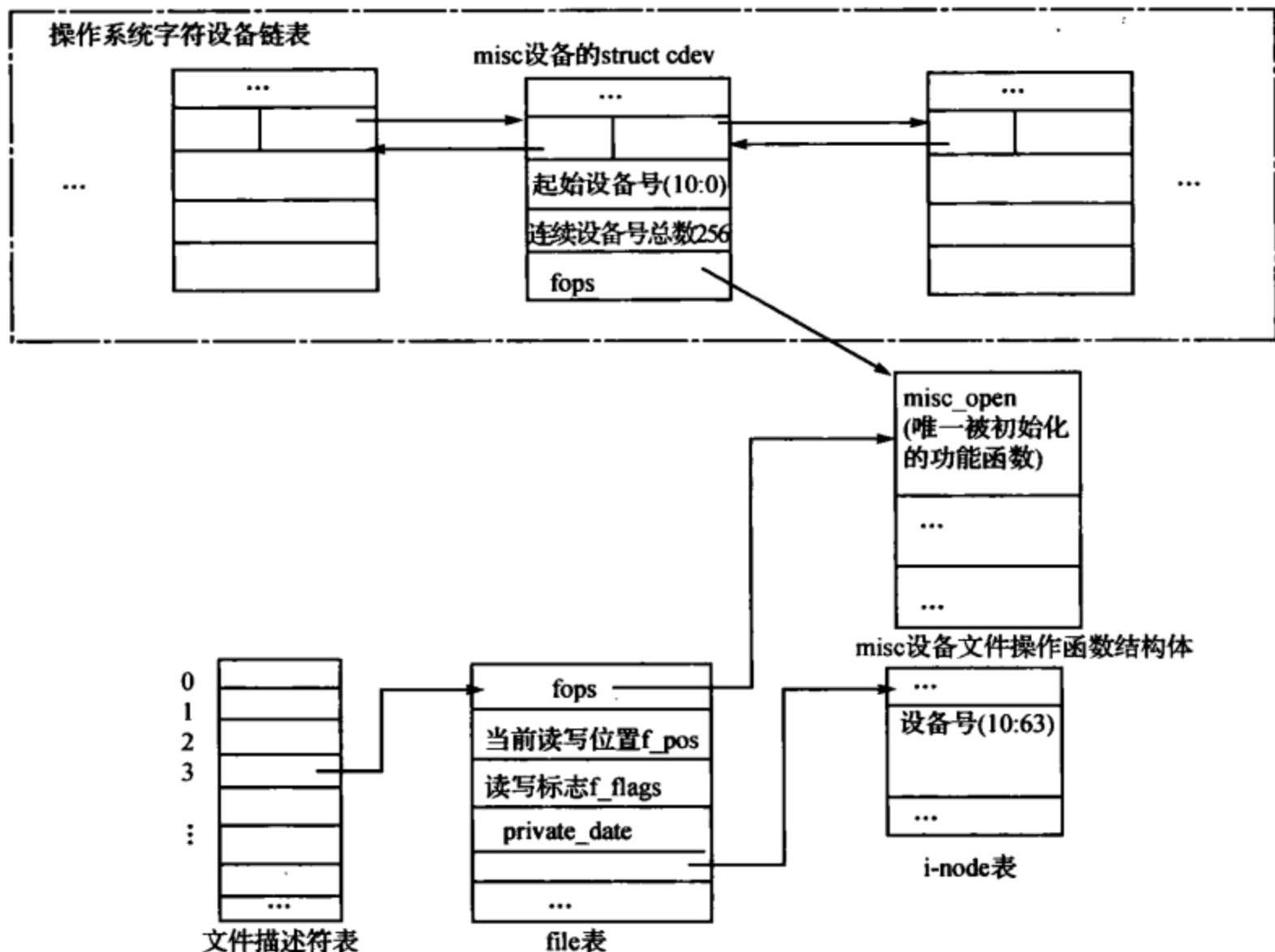


图 9-4 应用程序调用 open, OS 执行相关操作后

然后调用 misc 设备的 open 函数——misc_open。

获得 misc 字符设备的次设备号(128 行)。

使用该次设备号遍历全局 misc_list 链表,查找该 misc 字符设备是否已注册,是则获得该 misc 字符设备的 fops(135~140 行)。

用找到的 fops 更正该 misc 字符设备的 file 表(159 行),从此之后 misc 字符设备的功能函数(read、write 等)就能被正确找到并执行。

如果该 misc 字符设备定义了自己的 open 函数,则执行之(160~161 行),如图 9-5 所示。

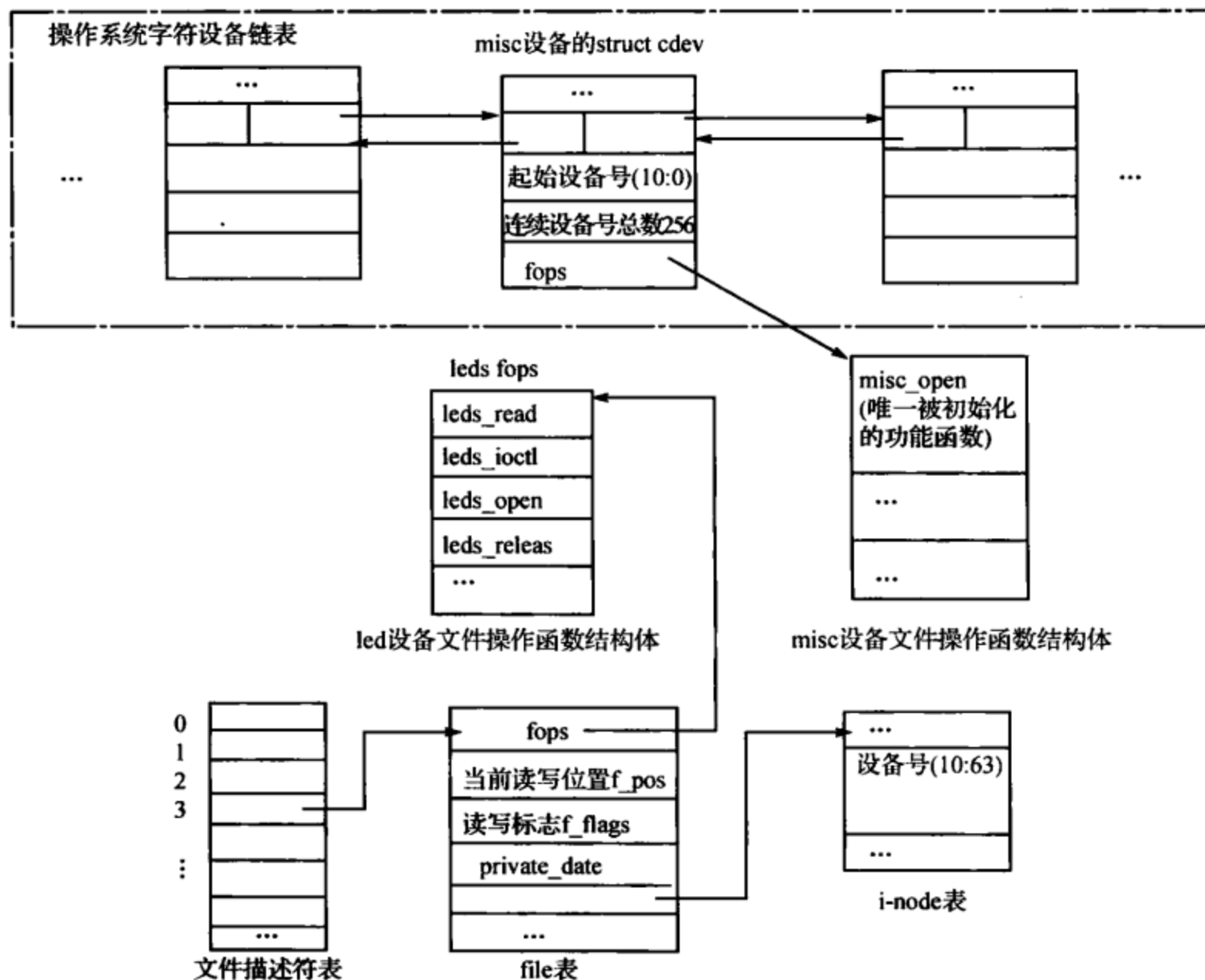


图 9-5 OS 调用 misc_open 后

```

126 static int misc_open(struct inode * inode, struct file * file)
127 {
128     int minor = iminor(inode);
129     struct miscdevice * c;
130     const struct file_operations * old_fops, * new_fops = NULL;
131     list_for_each_entry(c, &misc_list, list) {
132         if (c->minor == minor) {
133             new_fops = fops_get(c->fops);
134             break;
135         }
136     }
137     if (! new_fops) {
138         goto fail;
139     }
140 }
141
142 if (! new_fops) {
143     goto fail;
144 }

```

```

155     }
158     old_fops = file->f_op;
159     file->f_op = new_fops;
160     if (file->f_op->open) {
161         file->f_op->open(inode, file);
162     }
163     fops_put(old_fops);
164 fail:
165 }

```

9.2.5 导出内核 API——misc_deregister 函数

导出 misc_deregister 函数供其他驱动调用,以注销真实的 misc 字符设备。当别的驱动调用 misc_deregister 时:

将注销的 misc 字符设备从全局链表 misc_list 中删除(264 行)

使用主设备号 10、miscdevice 结构体中的次设备号在 misc 类下删除 SYS 文件系统中的设备(265 行)

如果是使用动态次设备号注册的,则将 misc_minors 数组中的相应 bit 位清 0,收回该动态次设备号(258、266~268)

```

256 int misc_deregister(struct miscdevice * misc)
257 {
258     int i = misc->minor;
264     list_del(&misc->list);
265     device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
266     if (i < DYNAMIC_MINORS && i > 0) {
267         misc_minors[i >> 3] &= ~(1 << (misc->minor & 7));
268     }
269 }
270
271 EXPORT_SYMBOL(misc_deregister);

```

9.3 Watchdog 驱动

9.3.1 相关概念

1. 平台设备及平台设备驱动

通常在 Linux 中,把 SoC 系统中集成的独立外设单元(如:I2C、IIS、RTC、看门狗等)都当

作平台设备来处理。在 Linux 中用 platform_device 结构体来描述一个平台设备,在内核中定义在:include/linux/platform_device.h 中,如下:

```
16 struct platform_device {
17     const char      * name; //设备名称,用于匹配平台设备驱动
18     u32              id;
19     struct device    dev;
20     u32*             num_resources; //设备使用各类资源的数量
21     struct resource * resource; //设备使用的资源,主要包括设备使用的内存地址和中断号
22 };
```

在 arch/arm/plat-s3c24xx/devs.c 中就定义了很多平台设备,watchdog 就是其中的一种的:

```
/* 定义了 Watchdog 平台设备 */
struct platform_device s3c_device_wdt = {
    .name          = "s3c2410-wdt", /* 设备名称 */
    .id            = -1,
    .num_resources = ARRAY_SIZE(s3c_wdt_resource), /* 资源数量 */
    .resource       = s3c_wdt_resource, /* 引用上面定义的资源 */
};

EXPORT_SYMBOL(s3c_device_wdt);

/* 定义了 Watchdog 平台设备使用的资源,这些资源在驱动程序中都会用到 */
static struct resource s3c_wdt_resource[] = {
    [0] = { /* Watchdog 所使用 IO 端口资源范围 */
        .start = S3C24XX_PA_WATCHDOG, // 该宏为 0x53000000,是 watchdog 寄存器组的内存首地址
        .end   = S3C24XX_PA_WATCHDOG + S3C24XX_SZ_WATCHDOG - 1,
        .flags = IORESOURCE_MEM, //表示该资源是内存资源
    },
    [1] = { /* Watchdog 中断资源 */
        .start = IRQ_WDT, // 该宏表示 watchdog 的中断号
        .end   = IRQ_WDT,
        .flags = IORESOURCE_IRQ, //表示该资源是中断号
    }
};
```

定义了平台设备,那系统是怎么来使用它的呢?我们打开:arch/arm/mach-s3c2440/mach-smdk2440.c 这个 ARM 2440 平台的系统入口文件,可以看到在系统初始化函数 smdk2440_machine_init 中是使用 platform_add_devices 这个函数将一些平台设备添加到系

统中的,代码如下:

```
static struct platform_device * smdk2440_devices[] __initdata = {
    &s3c_device_usb,
    &s3c_device_lcd,
    &s3c_device_wdt,
    &s3c_device_i2c,
    &s3c_device_iis,
};

static void __init smdk2440_machine_init(void)
{
    s3c24xx_fb_set_platdata(&smdk2440_lcd_cfg);
    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}
```

这其实是在操作系统内部建立了一个平台设备链表,将所有平台设备作为节点链入该链表。每个节点由 struct platform_device 中的 name 字段唯一标识(watchdog 这个设备的 name 就是“s3c2410-wdt”),将来平台设备驱动与平台设备的匹配正是通过二者都定义相同的 name 来完成的。

在 Linux 中,系统还为平台设备定义了平台设备驱动结构体 platform_driver,平台设备驱动需要实现 platform_driver 结构体中定义的 probe、remove、suspend、resume 等接口函数。下面的代码来源于光盘中的 watchdog 驱动(\work\studydriver\watchdog\mys3c2410_wdt.c),该结构体中最重要的是 driver.name 字段和 probe 字段。

```
480 static struct platform_driver s3c2410wdt_driver = {
481     .probe          = s3c2410wdt_probe,
482     .remove         = s3c2410wdt_remove,
483     .shutdown       = s3c2410wdt_shutdown,
484     .suspend        = s3c2410wdt_suspend,
485     .resume         = s3c2410wdt_resume,
486     .driver          = {
487         .owner       = THIS_MODULE,
488         .name        = "s3c2410-wdt",
489     },
490 };
```

当平台设备驱动的初始化函数调用 platform_driver_register 的时候,系统会使用 s3c2410wdt_driver 中 driver.name 字段的值去遍历前面提到的平台设备链表。如果找到对应

的平台设备, 则用该平台设备结构体的指针作为参数, 回调 probe 字段填入的函数——s3c2410wdt_probe。该函数会对平台设备进行初始化等操作, 其中会使用内核 API platform_get_resource 从平台设备结构体中获得所需要的内存地址和中断号。

```

495 static int __init watchdog_init(void)
496 {
497     printk(banner);
498     return platform_driver_register(&s3c2410wdt_driver);
499 }
506 module_init(watchdog_init);

313 static int s3c2410wdt_probe(struct platform_device *pdev)

324     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
347     wdt_irq = platform_get_resource(pdev, IORESOURCE_IRQ, 0);

```

2. 混杂设备(misc 设备)

Watchdog 在驱动中是作为 misc(混杂)设备实现的, 关于 misc 设备的具体分析请参阅“内核 misc 设备架构分析”一节和“通过 I/O 内存驱动硬件的实战——LED 灯驱动”一节。

9.3.2 Watchdog 硬件结构分析

请参阅第 1 篇中的“看门狗工作原理”和“看门狗实验”两小节。

这里补充说明一点, 看门狗硬件除了可以产生 reset 信号外, 也能通过配置来产生(或者禁止产生)Watchdog 中断信号。

9.3.3 Watchdog 驱动的初始化和卸载

```

480 static struct platform_driver s3c2410wdt_driver = {
481     .probe          = s3c2410wdt_probe, /* Watchdog 探测函数, 真正的设备初始化是
由它完成的 */
482     .remove         = s3c2410wdt_remove,
483     .shutdown        = s3c2410wdt_shutdown,
484     .suspend         = s3c2410wdt_suspend,
485     .resume          = s3c2410wdt_resume,
486     .driver          = {
487         .owner       = THIS_MODULE,

```

/* 注意这里的名称一定要和系统中定义平台设备的地方一致, 这样才能把平台设备与该平台设备的驱动关联起来 */


```

488         .name      = "s3c2410-wdt",
489     },
490 };

495 static int __init watchdog_init(void)
496 {
497     printk(banner);
/* 将 Watchdog 注册成平台设备驱动 */
498     return platform_driver_register(&s3c2410wdt_driver);
499 }
500
501 static void __exit watchdog_exit(void)
502 {
/* 注销 Watchdog 平台设备驱动 */
503     platform_driver_unregister(&s3c2410wdt_driver);
504 }
505
506 module_init(watchdog_init);
507 module_exit(watchdog_exit);

```

9.3.4 探测函数 watchdog_probe 的实现

当 Watchdog 平台设备在系统的平台设备链表中存在时,系统将使用 Watchdog 平台设备的结构体指针作为参数,调用 s3c2410wdt_probe。

324 行获取 Watchdog 平台设备的物理内存地址,330~338 将该物理地址映射为虚拟地址。

347 行获取 Watchdog 平台设备的中断号,354 行注册 Watchdog 中断处理程序 s3c2410wdt_irq。

360 行获取 Watchdog 需要的时钟结构体,367 行启用该时钟信号。

372 行调用 s3c2410wdt_set_heartbeat 设置 watchdog 的超时值为 tmr_margin 秒。如果不成功的话,373 行设置确认值——15s。

383 行将 Watchdog 注册为混杂设备。

390~392 行启动 Watchdog(tmr_atboot 表示在驱动加载时,是否启动 Watchdog)。

393~398 行停止 Watchdog。

```

313 static int s3c2410wdt_probe(struct platform_device *pdev)
314 {
316     int cfg_err = 0;

```

```

324     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
330     size = (res->end-res->start)+1;
331     wdt_mem = request_mem_region(res->start, size, pdev->name);
338     wdt_base = ioremap(res->start, size);
347     wdt_irq = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
354     ret = request_irq(wdt_irq->start, s3c2410wdt_irq, 0, pdev->name, pdev);
360     wdt_clock = clk_get(&pdev->dev, "watchdog");
367     clk_enable(wdt_clock);
372     if (s3c2410wdt_set_heartbeat(tmr_margin)) {
373         cfg_err = s3c2410wdt_set_heartbeat(15);
381     }
383     ret = misc_register(&s3c2410wdt_miscdev);
390     if (tmr_atboot && cfg_err == 0) {
391         printk(KERN_INFO PFX "Starting Watchdog Timer\n");
392         s3c2410wdt_start();
393     } else if (! tmr_atboot) {
398         s3c2410wdt_stop();
399     }
401     return 0;

```

s3c2410wdt_set_heartbeat 设置 watchdog 的超时值为 timeout 秒:

121 行获取 Watchdog 所使用的时钟频率,结果 freq=50MHz

130、164 将 Watchdog 的多路复合器固定为 1/128。

131~145 则在计算 watchdog 的 8 位分频器的值——divisor,

29、153 将超时值保存在全局变量 tmr_margin 中,以便其他函数使用

130、131、158 行在 PCLK(freq=50 MB)、分频系数(divisor、1/128)确定的情况下,计算出 timeout 秒对应的时钟方波数目。159 行将该值保存到全局变量 wdt_count 中,以便其他函数使用。

166 行设置 WTDAT 寄存器,162、163、164、167 行设置 WTCON 寄存器,最终完成 watchdog 超时值的设定。

```

29 static int tmr_margin    = CONFIG_S3C2410_WATCHDOG_DEFAULT_TIME;
62 static unsigned int     wdt_count;
119 static int s3c2410wdt_set_heartbeat(int timeout)
120 {
121     unsigned int freq = clk_get_rate(wdt_clock);
123     unsigned int count;
124     unsigned int divisor = 1;
125     unsigned long wtcon;

```

```

130     freq /= 128;
131     count = timeout * freq;
141     if (count >= 0x10000) {
142         for (divisor = 1; divisor <= 0x100; divisor++) {
143             if ((count / divisor) < 0x10000)
144                 break;
145         }
147         if ((count / divisor) >= 0x10000) {
148             printk(KERN_ERR PFX "timeout %d too big\n", timeout);
149             return -EINVAL;
150         }
151     }
153     tmr_margin = timeout;
158     count /= divisor;
159     wdt_count = count;
162     wtcon = readl(wdt_base + S3C2410_WTCON);
163     wtcon &= ~0xff00;
164     wtcon |= (divisor - 1) << 8 | (3 << 3);
166     writel(count, wdt_base + S3C2410_WTDAT);
167     writel(wtcon, wdt_base + S3C2410_WTCON);
170 }

```

s3c2410wdt_stop 停止 Watchdog:

```

#define S3C2410_WTCON_ENABLE (1<<5)
#define S3C2410_WTCON_RSTEN (0x01)
80 static int s3c2410wdt_stop(void)
81 {
82     unsigned long wtcon;
84     wtcon = readl(wdt_base + S3C2410_WTCON);
85     wtcon &= ~(S3C2410_WTCON_ENABLE | S3C2410_WTCON_RSTEN);
86     writel(wtcon, wdt_base + S3C2410_WTCON);
89 }

```

s3c2410wdt_start() 启动 Watchdog:

```

#define S3C2410_WTCON_INTEN (1<<2)
91 static int s3c2410wdt_start(void)
92 {
93     unsigned long wtcon;
95     s3c2410wdt_stop();

```

PDF

```

97      wtcon = readl(wdt_base + S3C2410_WTCON);
98      wtcon |= S3C2410_WTCON_ENABLE;
101     if (soft_noboot) {
102         wtcon |= S3C2410_WTCON_INTEN;
103         wtcon &= ~S3C2410_WTCON_RSTEN;
104     } else {
105         wtcon &= ~S3C2410_WTCON_INTEN;
106         wtcon |= S3C2410_WTCON_RSTEN;
107     }
112     writel(wdt_count, wdt_base + S3C2410_WTDAT);
113     writel(wdt_count, wdt_base + S3C2410_WTCNT);
114     writel(wtcon, wdt_base + S3C2410_WTCON);
117 }

```

注:soft_noboot 表示是否禁用 Watchdog 的 reboot 的功能。如果禁用 reboot 功能,则启用 Watchdog 中断。

Watchdog 中断处理程序 s3c2410wdt_irq 完成喂狗操作(s3c2410wdt_keepalive):

```

304 static irqreturn_t s3c2410wdt_irq(int irqno, void * param)
305 {
306     printk(KERN_INFO PFX "Watchdog timer expired! \n");
308     s3c2410wdt_keepalive();
309     return IRQ_HANDLED;
310 }

74 static int s3c2410wdt_keepalive(void)
75 {
76     writel(wdt_count, wdt_base + S3C2410_WTCNT);
77     return 0;
78 }

```

9.3.5 实现 misc 设备中对设备文件的操作

```

287 static const struct file_operations s3c2410wdt_fops = {
288     .owner          = THIS_MODULE,
289     .llseek        = no_llseek,
290     .write          = s3c2410wdt_write,
291     .ioctl         = s3c2410wdt_ioctl,
292     .open          = s3c2410wdt_open,
293     .release       = s3c2410wdt_release,

```

资源解密

PDG

```
294 };
```

```
296 static struct miscdevice s3c2410wdt_miscdev = {
297     .minor          = WATCHDOG_MINOR,
298     .name           = "watchdog",
299     .fops           = &s3c2410wdt_fops,
300 };
```

应用打开/dev/watchdog 设备文件时,启动 Watchdog:

```
51 typedef enum close_state {
52     CLOSE_STATE_NOT, //表示不允许关闭 Watchdog
53     CLOSE_STATE_ALLOW = 0x4021 //表示允许关闭 Watchdog
54 } close_state_t;
63 static close_state_t    allow_close;

176 static int s3c2410wdt_open(struct inode * inode, struct file * file)
177 {
181     if (nowayout)
182         __module_get(THIS_MODULE);
184     allow_close = CLOSE_STATE_NOT;
187     s3c2410wdt_start();
188     return nonseekable_open(inode, file);
189 }
```

注:nowayout 表示 make menuconfig 时,指定了一旦 Watchdog 启动,就决不允许关闭。

应用程序关闭/dev/watchdog 设备文件时,如果允许关闭 Watchdog,则关闭它:

```
191 static int s3c2410wdt_release(struct inode * inode, struct file * file)
192 {
198     if (allow_close == CLOSE_STATE_ALLOW) {
199         s3c2410wdt_stop();
200     } else {
201         printk(KERN_CRIT "Unexpected close, not stopping watchdog! \n");
202         s3c2410wdt_keepalive();
203     }
205     allow_close = CLOSE_STATE_NOT;
208 }
```

当应用程序向/dev/watchdog 设备文件写入时,则喂狗:

```
247 static int s3c2410wdt_ioctl(struct inode * inode, struct file * file,
248         unsigned int cmd, unsigned long arg)
```



```

249 {
250     void __user * argp = (void __user *)arg;
251     int __user * p = argp;
252     int new_margin;
253     switch (cmd) {
254         default;
255             return -ENOTTY;
256         case WDIOC_GETSUPPORT;
257             return copy_to_user(argp, &s3c2410_wdt_ident,
258                 sizeof(s3c2410_wdt_ident)) ? -EFAULT : 0;
259         case WDIOC_GETSTATUS;
260         case WDIOC_GETBOOTSTATUS;
261             return put_user(0, p);
262         case WDIOC_KEEPAWAKE;
263             s3c2410wdt_keeplive();
264             return 0;
265         case WDIOC_SETTIMEOUT;
266             if (get_user(new_margin, p))
267                 return -EFAULT;
268             if (s3c2410wdt_set_heartbeat(new_margin))
269                 return -EINVAL;
270             s3c2410wdt_keeplive();
271             return put_user(tmr_margin, p);
272         case WDIOC_GETTIMEOUT;
273             return put_user(tmr_margin, p);
274     }
275 }
276 }

```

9.3.6 Watchdog 平台驱动的设备移除、挂起和恢复接口函数的实现

```

/* Watchdog 平台驱动的设备移除接口函数的实现 */
420 static int s3c2410wdt_remove(struct platform_device * dev)
421 {
422     /* 释放获取的 Watchdog 平台设备的 IO 资源 */
423     release_resource(wdt_mem);
424     kfree(wdt_mem);
425     wdt_mem = NULL;
426     /* 同 watchdog_probe 中中断的申请相对应,在那里申请中断,这里就释放中断 */
427     free_irq(wdt_irq->start, dev);

```

```
427         wdt_irq = NULL;
        /* 释放获取的 Watchdog 平台设备的时钟 */
429         clk_disable(wdt_clock);
430         clk_put(wdt_clock);
431         wdt_clock = NULL;
        /* 释放 Watchdog 设备虚拟地址映射空间 */
433         iounmap(wdt_base);
        /* 注销 misc 设备 */
434         misc_deregister(&s3c2410wdt_miscdev);
435         return 0;
436 }

/* Watchdog 平台驱动的设备关闭接口函数的实现 */
438 static void s3c2410wdt_shutdown(struct platform_device * dev)
439 {
        /* 停止看门狗定时器 */
440         s3c2410wdt_stop();
441 }

/* 对 Watchdog 平台设备驱动电源管理的支持。CONFIG_PM 这个宏定义在内核中, */
/* 当配置内核时选上电源管理,则 Watchdog 平台驱动的设备挂起和恢复功能均有效, */
443 #ifdef CONFIG_PM
/* 定义两个变量来分别保存挂起时的 WTCON 和 WTDAT 值,到恢复的时候使用 */
445 static unsigned long wtcon_save;
446 static unsigned long wtdat_save;

/* Watchdog 平台驱动的设备挂起接口函数的实现 */
448 static int s3c2410wdt_suspend(struct platform_device * dev, pm_message_t state)
449 {
        /* 保存挂起时的 WTCON 和 WTDAT 值 */
451         wtcon_save = readl(wdt_base + S3C2410_WTCON);
452         wtdat_save = readl(wdt_base + S3C2410_WTDAT);
453
        /* 停止看门狗定时器 */
455         s3c2410wdt_stop();
457         return 0;
458 }

/* Watchdog 平台驱动的设备恢复接口函数的实现 */
```

```

460 static int s3c2410wdt_resume(struct platform_device * dev)
461 {
    /* 恢复挂起时的 WTCNT 和 WTDAT 值,注意这个顺序 */
464     writel(wtdat_save, wdt_base + S3C2410_WTDAT);
465     writel(wtdat_save, wdt_base + S3C2410_WTCNT); /* Reset count */
466     writel(wtcon_save, wdt_base + S3C2410_WTCNT);
471     return 0;
472 }
473
474 #else /* 配置内核时没选上电源管理,Watchdog 平台驱动的设备挂起和恢复功能均无效, */
/* 这两个函数也就无需实现了 */
475 #define s3c2410wdt_suspend NULL
476 #define s3c2410wdt_resume NULL
477 #endif /* CONFIG_PM */

```

9.3.7 测试 Watchdog 驱动

加载 Watchdog 驱动后,请交叉编译光盘中的 test_wdt.c 进行测试:

/* 在 sigint 信号处理程序中向 watchdog 写入 MAGICCLOSE 字符'V',从而能够在关闭 watchdog 设备文件时顺利停止 watchdog */

```

void sigint(int signum)
{
    write(fd, "V", 1);
    close(fd);
    exit(-1);
}

```

```

int main(int argc, char * argv[])
{
    int rst_peroid, feddog_peroid;
    signal(SIGINT, sigint);
    if ((fd = open(argv[1], O_WRONLY)) == -1) { // 打开 watchdog 设备文件时,会启动 watchdog
        perror("open");
        exit(-1);
    }
    feddog_peroid = atoi(argv[2]);

    ioctl(fd, WDIOC_GETTIMEOUT, &rst_peroid); //获取当前 watchdog 的超时时间
    printf("watchdog timeout is %d\n", rst_peroid);
}

```

```

rst_peroid = atoi(argv[3]);
ioctl(fd, WDIOC_SETTIMEOUT, &rst_peroid); //设置 watchdog 的超时时间
if (feeddog_peroid != 0) {
    while (1) {
        ioctl(fd, WDIOC_KEEPLIVE); //喂狗
        sleep(feeddog_peroid);
    }
}
close(fd);
return 0;
}

```

(1) 输入下面命令,表示设置 Watchdog 的喂狗周期为 3s,重启周期为 5s,结果机器不重启。按下 Ctrl+C 中止该程序,机器也不重启,这是由于 sigint 信号处理程序关闭 Watchdog 设备文件前向 Watchdog 设备写入了 MAGICCLOSE 字符‘V’,因而可以顺利停止 Watchdog

```
# ./test_wdt /dev/watchdog 3 5
```

(2) 输入下面命令,表示设置 Watchdog 的喂狗周期为 5s,重启周期为 3s,结果机器重启:

```
# ./test_wdt /dev/watchdog 5 3
```

(3) 输入下面命令,后台执行 test_wdt,表示设置 Watchdog 的喂狗周期为 3s,重启周期为 5s。然后在输入 killall 命令结束 test_wdt,结果机器重启,这是因为 killall 杀死了喂狗程序,而 Watchdog 有没有被停止,最终在 5s 超时后,机器重启:

```
# ./test_wdt /dev/watchdog 3 5 &
```

```
# killall test_wdt
```

9.4 内核编码规范与风格

到目前为止,已经阅读了一些内核的驱动源代码。如果继续阅读更多的内核驱动源码,就会发现这些源码的作者都自觉的采用了一些编码的规范,或者说是风格。因此,作为编写驱动程序的初学者,有必要一开始就了解并严格遵守这些编码风格。

9.4.1 缩进、长行、{}与空格的使用规范

- 缩进用 tab,不使用空格作为缩进符。
- 超过 3 层缩进,说明代码需要修订。
- case 与 switch 并排,不需要缩进。

```
switch (TID_TIMER_ID) {
```

```

case 0:
    hwp->cpm_vec = CPMVEC_TIMER1;
    break;
case 1:
    hwp->cpm_vec = CPMVEC_TIMER2;
    break;
}

```

除非你想隐藏一些东西,否则不要在一行上书写多条语句。

不要在一行上出现多个赋值符,因为要避免出现晦涩难懂的表达式。

一行一定不允许超过 80 个字符。

长行断开时,尽可能保障后面的行缩进放在第一行的右边。

```

if (! index &&
    (! cascade(base, &base->tv2, INDEX(0))) &&
    (! cascade(base, &base->tv3, INDEX(1))) &&
    ! cascade(base, &base->tv4, INDEX(2)))
    cascade(base, &base->tv5, INDEX(3));

```

{放于行尾,}放于行首。函数例外,{放于行首。

if-else 语句一定要加{ },例外情况是 if 只有一条语句且没有 else 分支。

关键字 if, switch, case, for, do, while 后加一个空格,紧挨()内壁不加空格。

看上去像函数的关键字 sizeof, typeof, alignof, __attribute__ 后不加空格。

指针号 * 应紧挨变量名或函数名,即:二者之间不要加空格。

二元和三元运算符左右各加一个空格,一元运算符不加空格。++,.,->不加空格。

```
#define GET_TI(count,i) (((count) - i->last_counter) & (i->mask) >> (i->shift))
```

9.4.2 变量和函数

局部变量要使用容易理解的简写,例如 tmp,不要使用长名。

在不引起歧义的情况下,局部变量应该尽量简写,例如循环变量 i。

全局变量和全局函数不要使用简写,例如应当使用 count_active_users() 而不是 cntusr()。

全局变量和全局函数命名不要采用大小写混合。

不要把变量的类型放进变量名,即:不要采用 Hungarian notation。

让程序员直接看到数据类型的具体含义是最直接也是最好的选择,所以应避免使用 typedef。

但在以下几种情况应该考虑使用 typedef。

当需要完全隐藏变量与体系结构相关的实现细节。不过此时应该提供相应的访问函数供

调用,例如 `dev_t`。

当某种类型将来可能发生变化,而现在的程序必须要考虑到向后兼容的问题,例如 `pid_t`。
明确整数变量的类型,例如 `u8/u16/u32`。

正常情况下,一个函数的源码不能超过两屏(一屏为 80×24)。一定坚持一个函数只做一件事并做好。例如:`schedule` 函数。

函数功能越复杂,越要保持短小精悍,因此对复杂功能的函数一定要拆分成多个逻辑简单的短函数。

对于逻辑简单的函数,可以较长。例如:分支很多但功能简单的 `case`。

1 个函数的局部变量,不要超过 10 个。因为人的大脑难以同时跟踪 7 个以上的不同事情。

用一个空行分割不同的函数定义。但 `EXPORT_SYMBOL` 紧接函数的定义体,不留空行。

函数原形声明,需完整包含参数的名称和类型。这样源码阅读者可以获得更多有价值的信息。

```
extern long sched_setaffinity(pid_t pid, cpumask_t new_mask);
```

将函数的退出点通过 `goto` 语句进行集中化。

这主要用在函数需要从多个点退出,并在退出前需要做相关的清理工作。

使用 `p = kmalloc(sizeof(*p), ...)`,而不要使用 `sizeof(结构名)`。因为结构名即使能改变,也不会出错。

滥用 `inline` 会导致内核增大,从而导致整个系统性能下降。

一般的规则是,超过 3 行代码的函数不要使用 `inline`。

只被调用一次的 `static` 函数前不要加 `inline`,因为 `gcc` 会自动进行这样的优化。而且当该函数变为被调用两次后,`gcc` 会自动调整优化,而不需要进行人工去除 `inline` 的负担。

执行任务的函数的返回值。

0 表示成功。

-Exxx 表示失败。

例如:`add_work()`。

判定真假的函数的返回值。

0 表示 `false`。

1 表示 `true`。

例如:`pci_dev_present()`。

有实际返回结果值的函数:返回结果集外的值表示失败。例如 `kmalloc` 返回 `NULL` 表示失败。

9.4.3 注释、macros 和 enums

注释放在函数头部,不要放在函数体中间,否则应该考虑拆分函数以变简单。

只注释函数完成了什么功能以及为什么,不要注释函数是如何完成功能的,因为简单的函数如何完成功能是显而易见的。

对于函数中特别有技巧的地方,可以予以注释,但一定要短小精悍。

单行注释请采用 C89 风格,即: `/* ... */`,而不采用 C99 风格,即: `// ...`。

多行注释采用如下风格:

```
/*
 * This is the preferred style for multi-line
 * comments in the Linux kernel source code.
 */
```

重要变量的注释很重要,采用紧随变量进行单行注释的风格。因此要求一行一个变量,并要求注释短小精悍。

注释内核 api 请使用 kernel-doc 格式,详见 Documentation/kernel-doc-nano-HOWTO.txt 和 scripts/kernel-doc

全局数据结构需要有引用计数,例如:struct mm_struct 有 mm_users 和 mm_count。

宏名和枚举名一般要全部大写。

定义几个相关常量时,优先使用枚举。

类似函数的宏的名称要使用小写,例如:wait_event。

尽可能使用 inline 函数替代类似函数的宏。

多语句宏应该放置在 do-while(0)块中,例如:wait_event。

宏的定义不能改变程序流程,例如下面的宏定义是错误的:

```
#define FOO(x) \
do { \
if (blah(x) < 0) \
return -EBUGGERED; \
} while(0)
```

宏定义中的表达式要用括弧括起来,例如: `#define CONSTEXP (CONSTANT | 3)`。

9.4.4 快乐使用内核提供的实现常用功能的宏

获得数组的元素个数:



```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

获得特定结构体中指定域的大小：

```
#define FIELD_SIZEOF(t, f) (sizeof(((t *)0) ->f))
```

计算包裹结构体地址：

```
container_of(ptr, type, member)
```

除此之外,还有类似于 min、max 这样的大量的宏,请细读 include/linux/kernel.h。

第 10 章

Linux 驱动中的中断编程

10.1 驱动程序调测方法与技巧

驱动程序开发的一个重大难点就是不易调试。本节目的就是介绍驱动开发中常用的以下几种直接和间接的调试手段：

利用 printk。

查看 OOP 消息。

利用 strace。

利用内核内置的 hacking 选项。

利用 ioctl 方法。

利用 /proc 文件系统。

使用 kgdb。

10.1.1 利用 printk

这是驱动开发中最朴实无华,同时也是最常用的和有效的手段。scull 驱动的 main.c 第 338 行如下,就是使用 printk 进行调试的例子,这样的例子相信大家在阅读驱动源码时随处可见。

```
338 //      printk(KERN_ALERT "wakeup by signal in process %d\n", current->pid);
```

printk 的功能与经常在应用程序中使用的 printf 是一样的,不同之处在于 printk 可以在打印字符串前面加上内核定义的宏,例如上面例子中的 KERN_ALERT(注意:宏与字符串之间没有逗号)。

```
#define KERN_EMERG "<0>";  
#define KERN_ALERT "<1>";  
#define KERN_CRIT "<2>";  
#define KERN_ERR "<3>";
```

```
# define KERN_WARNING "<4>";
# define KERN_NOTICE "<5>";
# define KERN_INFO "<6>";
# define KERN_DEBUG "<7>";
# define DEFAULT_CONSOLE_LOGLEVEL 7.
```

这个宏是用来定义需要打印的字符串的级别。值越小,级别越高。内核中有个参数用来控制是否将 printk 打印的字符串输出到控制台(屏幕或者/sys/log/syslog 日志文件)。

```
# cat /proc/sys/kernel/printk
6    4    1    7
```

第一个 6 表示级别高于(小于)6 的消息才会被输出到控制台,第二个 4 表示如果调用 printk 时没有指定消息级别(宏)则消息的级别为 4,第三个 1 表示接受的最高(最小)级别是 1,第四个 7 表示系统启动时第一个 6 原来的初值是 7。

因此,如果发现在控制台上看不到程序中某些 printk 的输出,请使用 echo 8 > /proc/sys/kernel/printk 来解决。

在复杂驱动的开发过程中,为了调试,会在源码中加入成百上千的 printk 语句。而当调试完毕形成最终产品的时候必然会将这些 printk 语句删除(为什么?想想自己是驱动的使用者而不是开发者吧,这个工作量是不小的。最要命的是,如果将调试用的 printk 语句删除后,用户又报告驱动有 bug,所以又不得不手工将这些上千条的 printk 语句再重新加上。所以,我们需要一种能方便地打开和关闭调试信息的手段。哪里能找到这种手段呢?下面看看 scull 驱动或者 leds 驱动的源代码吧!

```
# define LEDS_DEBUG
# undef PDEBUG      /* undef it, just in case */
# ifdef LEDS_DEBUG
#   ifdef __KERNEL__
/* This one if debugging is on, and kernel space */
#   define PDEBUG(fmt, args...) printk( KERN_EMERG "leds: " fmt, ## args)
#   else
/* This one for user space */
#   define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#   endif # else
#   define PDEBUG(fmt, args...) /* not debugging: nothing */
# endif
# undef PDEBUGG
# define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

这样一来,在开发驱动的过程中,如果想打印调试消息,“就可以用 PDEBUG("address of

`i_cdev is %p\n", inode->i_cdev);`”，如果不想看到该调试消息，就只需要简单地将 PDE-BUG 改为 PDEBUGG 即可。而当调试完毕形成最终产品时，只需要简单地将第一行注释掉即可。

上边那一段代码中的 `__KERNEL__` 是内核中定义的宏，当编译内核（包括模块）时，它会被定义。当然如果不明白代码中的 `...` 和 `##` 是什么意思的话，就请认真查阅一下 gcc 关于预处理部分的资料吧！

10.1.2 详解 OOP 消息

OOP 意为惊讶。当驱动有问题，内核将打印 OOP 消息。下面就来看看内核是如何惊讶的。

根据 `faulty.c`（位于光盘\work\studydriver\examples\misc-modules）编译出 `faulty.ko`，并 `insmod faulty.ko`。执行 `echo yang >/dev/faulty`，结果内核就惊讶了。内核为什么会惊讶呢？因为 `faulty` 驱动的 `write` 函数执行了 `*(int *)0 = 0`，向内存 0 地址写入，这是内核绝对不会容许的。

```
52 ssize_t faulty_write (struct file * filp, const char __user * buf, size_t count,
53         loff_t * pos)
54 {
55     /* make a simple fault by dereferencing a NULL pointer */
56     *(int *)0 = 0;
57     return 0;
58 }
```

```
1 Unable to handle kernel NULL pointer dereference at virtual address 00000000
2 pgd = c3894000
3 [00000000] * pgd = 33830031, * pte = 00000000, * ppte = 00000000
4 Internal error: Oops: 817 [#1] PREEMPT
5 Modules linked in: faulty scull
6 CPU: 0      Not tainted (2.6.22.6 #4)
7 PC is at faulty_write+0x10/0x18 [faulty]
8 LR is at vfs_write+0xc4/0x148
9 pc : [<bf00608c>]    lr : [<c0088eb8>]    psr: a0000013
10 sp : c3871f44   ip : c3871f54   fp : c3871f50
11 r10: 4021765c  r9 : c3870000   r8 : 00000000
12 r7 : 00000004  r6 : c3871f78   r5 : 40016000   r4 : c38e5160
13 r3 : c3871f78  r2 : 00000004   r1 : 40016000   r0 : 00000000
14 Flags: NzCv  IRQs on  FIQs on  Mode SVC_32  Segment user
15 Control: c000717f  Table: 33894000  DAC: 00000015
```





```

16 Process sh (pid: 745, stack limit = 0xc3870258)
17 Stack: (0xc3871f44 to 0xc3872000)
18 1f40:          c3871f74 c3871f54 c0088eb8 bf00608c 00000004 c38e5180 c38e5160
19 1f60: c3871f78 00000000 c3871fa4 c3871f78 c0088ffc c0088e04 00000000 00000000
20 1f80: 00000000 00000004 40016000 40 215730 00000004 c002c0e4 00000000 c3871fa8
21 1fa0: c002bf40 c0088fc0 00000004 40016000 00000001 40016000 00000004 00000000
22 1fc0: 00000004 40016000 40215730 00000004 00000001 00000000 4021765c 00000000
23 1fe0: 00000000 bea60964 0000266c 401adb40 60000010 00000001 00000000 00000000
24 Backtrace:
25 [<bf00607c>] (faulty_write + 0x0/0x18 [faulty]) from [<c0088eb8>] (vfs_write + 0xc4/
0x148)
26 [<c0088df4>] (vfs_write + 0x0/0x148) from [<c0088ffc>] (sys_write + 0x4c/0x74)
27  r7:00000000 r6:c3871f78 r5:c38e5160 r4:c38e5180
28 [<c0088fb0>] (sys_write + 0x0/0x74) from [<c002bf40>] (ret_fast_syscall + 0x0/0x2c)
29 r8:c002c0e4 r7:00000004 r6:402157 30 r5:40016000 r4:00000004
30 Code: e1a0c00d e92dd800 e24cb004 e3a00000 (e5800000)

```

1 行是惊讶的原因,也就是报告出错的原因。

2~4 行是 OOP 信息序号。

5 行是出错时内核已加载模块。

6 行是发生错误的 CPU 序号。

7~15 行是发生错误的位置,以及当时 CPU 各个寄存器的值,这最有利于找出问题所在地。

16 行是当前进程的名字及进程 ID。

17~23 行是出错时栈内的内容。

24~29 行是栈回溯信息,可看出直到出错时的函数递进调用关系(确保 CONFIG_FRAME_POINTER 被定义)。

30 行是出错指令及其附近指令的机器码,出错指令本身在小括号中。

反汇编 faulty.ko(arm-linux-objdump -D faulty.ko > faulty.dis;cat faulty.dis)可以看到如下的语句:

```

0000007c <faulty_write>:
   7c:  e1a0c00d    mov  ip, sp
   80:  e92dd800    stmdb sp!, {fp, ip, lr, pc}
   84:  e24cb004    sub  fp, ip, #4      ; 0x4
   88:  e3a00000    mov  r0, #0      ; 0x0
   8c:  e5800000    str  r0, [r0]
   90:  e89da800    ldmia sp, {fp, sp, pc}

```


定位出错位置以及获取相关信息的过程：

```

9 pc : [<bf00608c>] lr : [<c0088eb8>] psr: a0000013
25 [<bf00607c>] (faulty_write + 0x0/0x18 [faulty]) from [<c0088eb8>] (vfs_write + 0xc4/
0x148)
26 [<c0088df4>] (vfs_write + 0x0/0x148) from [<c0088ffc>] (sys_write + 0x4c/0x74)

```

出错代码是 faulty_write 函数中的第 5 条指令((0xbf00608c-0xbf00607c)/4+1=5),该函数的首地址是 0xbf00607c,该函数总共 6 条指令(0x18),该函数是被 0xc0088eb8 的前一条指令调用的(即:函数返回地址是 0xc0088eb8。这一点可以从出错时 lr 的值正好等于 0xc0088eb8 得到印证)。调用该函数的指令是 vfs_write 的第 49 条((0xc4-4)/4+1=49)指令。

达到出错处的函数调用流程是:write(用户空间的系统调用)——>sys_write——>vfs_write——>faulty_write

OOP 消息不仅让我们定位了出错的地方,更让我们惊喜的是,它让我们知道了一些秘密:①gcc 中 fp 到底有何用处?②为什么 gcc 编译任何函数的时候,总是要把 3 条看上去傻傻的指令放在整个函数的最开始?③内核和 gdb 是如何知道函数调用栈顺序,并使用函数的名字而不是地址?④如何才能知道各个函数入栈的内容?下面再看一次内核惊讶吧。

执行 cat /dev/faulty,内核又再一次惊讶!

```

1 Unable to handle kernel NULL pointer dereference at virtual address 0000000b
2 pgd = c3a88000
3 [0000000b] * pgd = 33a79031, * pte = 00000000, * ppte = 00000000
4 Internal error: Oops: 13 [#2] PREEMPT
5 Modules linked in: faulty
6 CPU: 0 Not tainted (2.6.22.6 #4)
7 PC is at vfs_read+0xe0/0x140
8 LR is at 0xffffffff
9 pc : [<c0088c84>] lr : [<ffffffff>] psr: 20000013
10 sp : c38d9f54 ip : 0000001c fp : ffffffff
11 r10: 00000001 r9 : c38d8000 r8 : 00000000
12 r7 : 00000004 r6 : ffffffff r5 : ffffffff r4 : ffffffff
13 r3 : ffffffff r2 : 00000000 r1 : c38d9f38 r0 : 00000004
14 Flags: nzCv IRQs on FIQs on Mode SVC_32 Segment user
15 Control: c000717f Table: 33a88000 DAC: 00000015
16 Process cat (pid: 767, stack limit = 0xc38d8258)
17 Stack: (0xc38d9f54 to 0xc38da000)
18 9f40: 00002000 c3c105a0 c3c10580
19 9f60: c38d9f78 00000000 c38d9fa4 c38d9f78 c0088f88 c0088bb4 00000000 00000000

```

```

20 9f80: 00000000 00002000 bef07c80 00000003 00000003 c002c0e4 00000000 c38d9fa8
21 9fa0: c002bf40 c0088f4c 00002000 bef07c80 00000003 bef07c80 00002000 00000000
22 9fc0: 00002000 bef07c80 00000003 00000000 00000000 00000001 00000001 00000003
23 9fe0: 00000000 bef07c6c 0000266c 401adab0 60000010 00000003 00000000 00000000
24 Backtrace: invalid frame pointer 0xffffffff
25 Code: ebf8ff86 e3500000 e1a07000 da000015 (e594500c)
26 Segmentation fault

```

不过这次惊讶却令人大为不解。OOP 竟然说出错的地方在 `vfs_read` (要知道它可是大牛们千锤百炼的内核代码), 万能的内核也不能追踪函数调用栈了, 这是为什么? 其实问题出在 `faulty_read` 的 43 行, 它导致入栈的 `r4`、`r5`、`r6`、`fp` 全部变为了 `0xffffffff`, `ip`、`lr` 的值未变, 这样一来 `faulty_read` 函数能够成功返回到它的调用者——`vfs_read`。但是可怜的 `vfs_read` (忠实的 APTCS 规则遵守者) 并不知道它的 `r4`、`r5`、`r6` 已经被万恶的 `faulty_read` 改变, 这样下去 `vfs_read` 命运就可想而知了——必死无疑! 虽然内核很有能力, 但缺少了正确的 `fp` 的帮助, 它也无法追踪函数调用栈。

```

36 ssize_t faulty_read(struct file * filp, char __user * buf,
37                     size_t count, loff_t * pos)
38 {
39     int ret;
40     char stack_buf[4];
41
42     /* Lets try a buffer overflow */
43     memset(stack_buf, 0xff, 20);
44     if (count > 4)
45         count = 4; /* copy 4 bytes to the user */
46     ret = copy_to_user(buf, stack_buf, count);
47     if (!ret)
48         return count;
49     return ret;
50 }

```

00000000 <faulty_read>:

```

0:  e1a0c00d      mov     ip, sp
4:  e92dd870      stmdb   sp!, {r4, r5, r6, fp, ip, lr, pc}
8:  e24cb004      sub     fp, ip, #4 ; 0x4
c:  e24dd004      sub     sp, sp, #4 ; 0x4, 这里为 stack_buf[] 在栈上分配一个字的空间, 局部

```

变量 `ret` 使用寄存器存储, 因此就不在栈上分配空间了

```

10: e24b501c      sub     r5, fp, #28 ; 0x1c

```

```

14: e1a04001    mov     r4, r1
18: e1a06002    mov     r6, r2
1c: e3a010ff    mov     r1, #255 ; 0xff
20: e3a02014    mov     r2, #20 ; 0x14
24: e1a00005    mov     r0, r5
28: ebfffffe    bl      28 <faulty_read+0x28> //这里在调用 memset
78: e89da878    ldmia   sp, {r3, r4, r5, r6, fp, sp, pc}

```

这次 OOP, 让我深刻地认识到以下几点:

(1) 内核能力超强, 但它不是也不可能是万能的。所以即使你能力再强, 也要和你的 team member 搞好关系, 否则在关键时候你会倒霉的。

(2) 出错的是 `faulty_read`, `vfs_read` 却做了替罪羊。所以人不要被表面现象所迷惑, 要深入看本质。

(3) 内核本来超级健壮, 可是你写的驱动是内核的组成部分, 由于它出错, 结果整体崩盘。所以当你加入一个团队的时候一定要告诫自己, 虽然你的角色也许并不重要, 但你的疏忽大意将足以令整个非常棒的团队崩盘。反过来说, 当你是 team leader 的时候, 在选团队成员的时候一定要慎重、慎重、再慎重, 即使他只是一个角色。

(4) 千万别惹堆栈, 它一旦出问题, 定位错误将会是一件非常困难的事情。

10.1.3 利用 strace

有时小问题可以通过监视程序监控用户应用程序的行为来追踪, 同时监视程序也有助于建立对驱动正确工作的信心。例如, 在看了它的读实现如何响应不同数量数据的读请求之后, 我们能够对 `scull` 正在正确运行感到有信心。

有几个方法来监视用户空间程序运行。可以运行一个调试器来通过单步方式运行它的函数, 增加打印语句, 或者在 `strace` 下运行程序。这里将讨论最后一个技术, 因为当真正目的是检查内核代码时, 它是最有用的。

`strace` 命令是一个有力工具, 它能显示用户空间程序发出的所有系统调用。它不仅显示系统调用, 还以符号形式显示系统调用的参数和返回值。当一个系统调用失败, 错误的符号值 (例如, `ENOMEM`) 和对应的字符串 (out of memory) 都将显示。`strace` 有很多命令行选项, 其中最有用的是 `-t` 来显示每个调用执行的时间, `-T` 来显示调用中花费的时间, `-e` 来限制被跟踪调用的类型 (例如 `strace -e read, write ls` 表示只监控 `ls` 运行时的 `read` 和 `write` 调用), 以及 `-o` 来重定向输出到一个文件。缺省情况下, `strace` 打印调用信息到 `stderr`。

`strace` 从内核自身获取信息。这意味着可以跟踪一个程序, 不管它是否带有调试支持编译 (对 `gcc` 是 `-g` 选项) 以及不管它是否被 `strip` 过。此外, 你也可以追踪一个正在运行中的进程, 这类似于调试器连接到一个运行中的进程并控制它。

跟踪信息常用来支持发给应用程序开发者的故障报告,但是对内核程序员也是很有价值的。我们已经看到驱动代码运行如何响应系统调用, `strace` 允许我们检查每个调用的输入和输出数据的一致性。

例如,运行命令 `strace ls /dev > /dev/scull0` 将会在屏幕上显示如下的内容:

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode = S_IFDIR|0755, st_size = 24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
getdents64(3, /* 141 entries */ , 4096) = 4088
[...]
getdents64(3, /* 0 entries */ , 4096) = 0
close(3) = 0
[...]
fstat64(1, {st_mode = S_IFCHR|0664, st_rdev = makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"... , 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"... , 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"... , 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"... , 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"... , 673) = 673
close(1) = 0
exit_group(0) = ?
```

从第一个 `write` 调用看,明显地,在 `ls` 结束查看目标目录后,它试图写 4KB。但奇怪的是,只有 4000 字节被成功写入,并且操作被重复。但当我们查看 `scull` 中的 `write` 实现,发现它一次最多只允许写一个 `quantum`(共 4000 字节),可见驱动本来就是期望部分写。几步之后,所有东西清空,程序成功退出。正是通过 `strace` 的输出,使我们确信驱动的部分写功能运行正确。

作为另一个例子,让我们读取 `scull` 设备(使用 `wc scull0` 命令):

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode = S_IFCHR|0664, st_rdev = makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"... , 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"... , 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"... , 16384) = 865
read(3, "", 16384) = 0
fstat64(1, {st_mode = S_IFCHR|0620, st_rdev = makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17) = 17
close(3) = 0
exit_group(0) = ?
```

如同期望的, read 一次只能获取 4000 字节, 但是数据总量等同于前个例子写入的。这个例子, 意外的收获是: 可以肯定, wc 为快速读进行了优化, 它因此绕过了标准库 (没有使用 fscanf), 而是直接一个系统调用以读取更多数据。这一点, 可从跟踪到的读的行里看到 wc 一次试图读取 16 KB 的数据而确认。

10.1.4 利用内核内置的 hacking 选项

内核开发者在 make menuconfig 的 Kernel hacking 中提供了一些内核调试选项。这些选项有助于调试驱动程序, 因为当启用某些调试选项的时候, 操作系统会在发现驱动运行有问题时给出一些错误提示信息, 而这些信息非常有助于驱动开发者找出驱动中的问题所在。下面就举几个简单例子。

先启用如下选项:

- General setup——Configure standard kernel features (for small systems)——Load all symbols for debugging/ksymoops (NEW)

- Kernel hacking——Kernel debugging

- Device Drivers——Generic Driver Options——Driver Core verbose debug messages

(1) Kernel debugging——Spinlock and rw-lock debugging: basic checks (NEW) 可以检查到未初始化的自旋锁。

(2) Kernel debugging——Mutex debugging: basic checks (NEW) 可以检查到未初始化的信号量。

```
717 //init_MUTEX(&scull_devices[i].sem);
```

例如, 如果忘记了初始化 scull 驱动中的信号量 (将 main.c 的第 717 行注释掉), 则在 open 设备 scull 时只会产生 OOP, 而没有其他信息提示我们有信号量未初始化, 因此此时很难定位问题。相反, 如果启用了上述选项, 操作系统则会产生相关提示信息, 使我们知道有未初始化的信号量或者自旋锁。从而, 就可以去驱动代码中初始化信号量和自旋锁的地方修正程序。

这个测试, 我们的意外收获是: 信号量的实现, 其底层仍然是自旋锁。这与我们之前的大胆推测一致。

```
process 751 enter scull_open
BUG: spinlock bad magic on CPU#0, sh/751
lock: c38ac1e4, .magic: 00000000, .owner: <none>/-1, .owner_cpu: 0
[<c002fe70>] (dump_stack+0x0/0x14) from [<c0130b5c>] (spin_bug+0x90/0xa4)
[<c0130acc>] (spin_bug+0x0/0xa4) from
[<c0130b98>] (_raw_spin_lock+0x28/0x160)
r5:40000013 r4:c38ac1e4
[<c0130b70>] (_raw_spin_lock+0x0/0x160) from [<c025276c>] (_spin_lock_irqsave+0x2c/
```

```

0x34)[<c0252740>] (_spin_lock_irqsave + 0x0/0x34) from [<c0053d28>]
(add_wait_queue_exclusive + 0x24/0x50)
r5;c38ac1e4 r4;c38a1e1c
[<c0053d04>] (add_wait_queue_exclusive + 0x0/0x50) from [<c024fcf0>]
(__down_interruptible + 0x5c/0x16c)
r5;c38a0000 r4;c38ac1dc
[<c024fc94>] (__down_interruptible + 0x0/0x16c) from [<c024fb4c>]
(__down_interruptible_failed + 0xc/0x20)
[<bf000530>] (scull_open + 0x0/0xd8 [scull]) from [<c0088eb8>] (chrdev_open + 0x1b4/0x1d8)
r6;c3ef0300 r5;c38ac1fc r4;bf0045a0

```

(3) Kernel debugging——Spinlock debugging: sleep — inside — spinlock checking (NEW)可以检查出驱动在获取自旋锁后又睡眠以及死锁等状况。

```

345    ssleep(5);
87 #define usespin

```

例如,如果第一个进程在获得自旋锁的情况下睡眠(去掉 main.c 第 345 行的注释,去掉 scull.h 第 87 行的注释),当第二个进程试图获得自旋锁时将死锁系统。但如果启用了上面的选项,则在死锁前操作系统可以给出提示信息。

```

process 763 enter read
semaphore get, and begin sleep 5 second in process 763
BUG: scheduling while atomic; cat/0x00000001/763
[<c002fe70>] (dump_stack + 0x0/0x14) from [<c024fe64>] (schedule + 0x64/0x778)
[<c024fe00>] (schedule + 0x0/0x778) from [<c02510a8>] (schedule_timeout + 0x8c/0xbc)
process 764 enter read
BUG: spinlock cpu recursion on CPU # 0, cat/764
lock: c3ae7014, .magic: dead4ead, .owner: cat/763, .owner_cpu: 0
[<c002fe70>] (dump_stack + 0x0/0x14) from [<c0130b5c>] (spin_bug + 0x90/0xa4)
[<c0130acc>] (spin_bug + 0x0/0xa4) from [<c0130bcc>] (_raw_spin_lock + 0x5c/0x160)
r5;beed2c70 r4;c3ae7014
[<c0130b70>] (_raw_spin_lock + 0x0/0x160) from [<c025273c>] (_spin_lock + 0x20/0x24)
[<c025271c>] (_spin_lock + 0x0/0x24) from [<bf000610>] (scull_read + 0x64/0x210 [scull])
r4;c3949520
[<bf0005ac>] (scull_read + 0x0/0x210 [scull]) from [<c0085eac>] (vfs_read + 0xc0/0x140)
BUG: spinlock lockup on CPU # 0, cat/764, c3ae7014
[<c002fe70>] (dump_stack + 0x0/0x14) from [<c0130c94>] (_raw_spin_lock + 0x124/0x160)
[<c0130b70>] (_raw_spin_lock + 0x0/0x160) from [<c025273c>] (_spin_lock + 0x20/0x24)
[<c025271c>] (_spin_lock + 0x0/0x24) from [<bf000610>] (scull_read + 0x64/0x210 [scull])
r4;c3949520

```



```
[<bf0005ac>] (scull_read + 0x0/0x210 [scull]) from [<c0085eac>] (vfs_read + 0xc0/0x140)
```

(4) Magic SysRq key 可以在已经死锁的情况下,打印一些有助于定位问题的信息。

魔键 sysrq 在大部分体系上都可用,它是用 PC 键盘上 alt 和 sysrq 键组合来发出的,或者在别的平台上使用其他特殊键(详见 documentation/sysrq.txt),在串口控制台上也可用(通过发送 break 后 5 秒内发送第 3 键)。一个第三键,与这两个一起按下,进行许多有用的动作中的一个:

r 关闭键盘原始模式;用在一个崩溃的应用程序(例如 X 服务器)可能将键盘搞成一个奇怪的状态。

k 调用“安全注意键”(SAK)功能。SAK 杀掉在当前控制台的所有运行的进程,提供一个干净的终端。

s 进行一个全部磁盘的紧急同步。

u umount. 试图重新加载所有磁盘在只读模式。这个操作,常常在 s 之后马上调用,可以节省大量的文件系统检查时间,在系统处于严重麻烦时。

b boot. 立刻重启系统。确认先同步和重新加载磁盘。

p 打印处理器消息。

t 打印当前任务列表。

m 打印内存信息。

例如,在系统死锁的情况下,期望能知道 CPU 寄存器的值,则可以按如下方法使用该魔法键。(假定你使用的是 minicom):

- 按下 ctrl+a,松开
- 立即按下 f 键,松开
- 5 秒内按下 p 键

将显示如下信息:

```
SysRq : Show Regs
Pid: 764, comm:      cat
CPU: 0      Not tainted (2.6.22.6 #6)
PC is at _raw_spin_lock + 0xbc/0x160
LR is at _raw_spin_lock + 0xcc/0x160
pc : [<c0130c2c>]      lr : [<c0130c3c>]      psr: 60000013
sp : c3b11ecc  ip : c3b11e08  fp : c3b11efc
r10: c3b10000  r9 : 00000000  r8 : 055b131f
r7 : c3ae7014  r6 : 00000000  r5 : 05f1e000  r4 : 00000000
r3 : 00000000  r2 : c3b10000  r1 : 00000001  r0 : 00000001
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  Segment user
Control: c000717f  Table: 33b48000  DAC: 00000015
```



```

[<c002cdb0>] (show_regs + 0x0/0x4c) from [<c015ab00>] (sysrq_handle_showregs + 0x20/0x28)
r4;c0310c34
[<c015aae0>] (sysrq_handle_showregs + 0x0/0x28) from [<c015ad50>] (__handle_sysrq + 0xa0/
0x148)
[<c015acb0>] (__handle_sysrq + 0x0/0x148) from [<c015ae28>] (handle_sysrq + 0x30/0x34)
[<c015adf8>] (handle_sysrq + 0x0/0x34) from [<c016477c>] (s3c24xx_serial_rx_chars +
0x1b0/0x2d4)
r5;00000000 r4;c03111e4
[<c01645cc>] (s3c24xx_serial_rx_chars + 0x0/0x2d4) from [<c0061474>] (handle_IRQ_event +
0x44/0x80)
[<c0061430>] (handle_IRQ_event + 0x0/0x80) from [<c00629a8>] (handle_level_irq + 0xd0/
0x134)
r7;c03073e8 r6;c3e52940 r5;00000046 r4;c03073bc
[<c00628d8>] (handle_level_irq + 0x0/0x134) from [<c0038118>] (s3c_irq_demux_uart + 0x50/
0x90)
r7;00000000 r6;00000046 r5;00000001 r4;c03073bc
[<c00380c8>] (s3c_irq_demux_uart + 0x0/0x90) from [<c003816c>] (s3c_irq_demux_uart0 +
0x14/0x18)
r6;c0336650 r5;0000002c r4;c0306cd4
[<c0038158>] (s3c_irq_demux_uart0 + 0x0/0x18) from [<c002b044>] (asm_do_IRQ + 0x44/0x5c)
[<c002b000>] (asm_do_IRQ + 0x0/0x5c) from [<c002ba78>] (__irq_svc + 0x38/0xb0)
Exception stack(0xc3b11e84 to 0xc3b11ecc)
1e80;          00000001 00000001 c3b10000 00000000 00000000 05f1e000 00000000
1ea0; c3ae7014 055b131f 00000000 c3b10000 c3b11efc c3b11e08 c3b11ecc c0130c3c
1ec0; c0130c2c 60000013 ffffffff
r7;00000002 r6;10000000 r5;f0000000 r4;fffffff
[<c0130b70>] (_raw_spin_lock + 0x0/0x160) from [<c025273c>] (_spin_lock + 0x20/0x24)
[<c025271c>] (_spin_lock + 0x0/0x24) from [<bf000610>] (scull_read + 0x64/0x210 [scull])
r4;c3949520
[<bf0005ac>] (scull_read + 0x0/0x210 [scull]) from [<c0085eac>] (vfs_read + 0xc0/0x140)
[<c0085dec>] (vfs_read + 0x0/0x140) from [<c00861d0>] (sys_read + 0x4c/0x74)
r7;00000000 r6;c3b11f78 r5;c3949520 r4;c3949540
[<c0086184>] (sys_read + 0x0/0x74) from [<c002bf00>] (ret_fast_syscall + 0x0/0x2c)
r8;c002c0a4 r7;00000003 r6;00000003 r5;beed2c70 r4;00002000

```

(5) Debug shared IRQ handlers 可用于调试共享中断。

10.1.5 其他调测方法简介

1. 利用 ioctl 方法

由于驱动中的 ioctl 函数可以将驱动的一些信息返回给用户程序,也可以让用户程序通过

ioctl 系统调用设置一些驱动的参数。所以在驱动的开发过程中,可以扩展一些 ioctl 的命令用于传递和设置调试驱动时所需各种信息和参数,以达到调试驱动的目的。如何在驱动中实现 ioctl,请参见“驱动程序对 ioctl 的规范实现”一节。

2. 利用 /proc 文件系统

/proc 文件系统用于内核向用户空间暴露一些内核的信息。因此出于调试的目的,我们可以在驱动代码中增加向 /proc 文件系统导出有助于监视驱动的信息的代码。这样一来,我们就可以通过查看 /proc 中的相关信息来监视和调试驱动。如何在驱动中实现向 /proc 文件系统导出信息,请参见《Linux Device Driver》的 4.3 节。

3. 使用 kgdb

kgdb 是在内核源码中打上(patch 上)用于调试内核的补丁,然后通过相应的硬件和软件,就可以像 gdb 单步调试应用程序一样来调试内核(当然包括驱动)。至于 kgdb 如何使用,就请你 google 吧,实在不行,百度一下也可以。

10.2 驱动程序中的中断处理

10.2.1 中断简述

(1) 使用中断原因

处理器速度远快于外设速度,因此不希望一直等待外设,而是希望能有一种机制,在外设需要处理器关注(处理)外设的时候,能通知处理器。这种机制就是中断,一个中断是一个硬件在它需要处理器的注意时发出的信号。

(2) 体验中断(源码位于光盘\work\studydriver 目录,buttons—mini2440.tgz、buttons—qq2440.tgz、buttons—tq2440.tgz 三个文件根据开发板进行正确选择)

① make 后,insmod s3c24xx_buttons.ko。

② 编译测试程序 button_test.c 后运行它。此时没有按键被按下,CPU 去做别的事情,测试进程进入睡眠。CPU(测试程序)并不需要不停地轮询(等待)按键是否被按下。

③ 按下任意按钮。此时按键这个外设向 CPU 发出中断信号引起 CPU 对它的注意,CPU 执行按键对应的中断处理程序,并唤醒睡眠的测试程序,测试程序读出并显示被按下的键:

```
# ./button_test
open success
read buttons successfully, begin print the result:
K1 has been pressed 1 times!
```

10.2.2 驱动程序中进行中断处理涉及的最基本的内核 API

驱动程序中进行中断处理涉及的最基本的内核 API 主要用于申请和释放中断。

模块使用中断前请求一个中断通道(即:启用某个中断),并且当结束时释放它(即:禁止某个中断)。在有些情况下,也希望模块能够与其他驱动共享中断线。

(1) 调用中断处理涉及的内核 API,需要包含中断注册接口头文件, <linux/interrupt.h>

(2) 申请中断,使用 `int request_irq(unsigned int irq, irqreturn_t (* handler)(int, void *), unsigned long flags, const char * dev_name, void * dev_id);`

① unsigned int irq, 请求的中断号。

② irqreturn_t (* handler), 中断处理函数指针。当中断产生时,该函数被操作系统回调,传入的第一个参数是中断号(也就是 request_irq 的第一个参数),传入的第二个参数是 request_irq 的第 5 个参数。

③ unsigned long flags, 一个与中断管理相关的选项的位掩码,例如低电平触发。

● IRQF_DISABLED(SA_INTERRUPT), 表示一个“快速”中断处理,即:ISR 运行期间,禁止中断(即:将 CPSR 的 bit7 置 1)。

● IRQF_SHARED(SA_SHIRQ), 表示中断可以在设备间共享。

● IRQF_TRIGGER_FALLING(SA_TRIGGER_FALLING), 表示下降沿触发。

④ const char * dev_name, 这个传递给 request_irq 的字符串用在 /proc/interrupts 来显示中断的拥有者。

⑤ void * dev_id。

如果存在中断共享,则它可用来区分共享中断。当释放中断时,需要指定这个参数。如果中断没有被共享,dev_id 可以设置为 NULL。

该值会作为第二个参数,传递给中断处理例程 handler,因此如果没有中断共享,则它可以被驱动用来指向它自己的私有数据区。

⑥ 成功返回 0,失败返回非 0

(3) 释放中断, `void free_irq(unsigned int irq, void * dev_id);`

(4) 示例代码:

```
request_irq(IRQ_EINT2, buttons_interrupt, IRQF_TRIGGER_FALLING | IRQF_SHARED,
"KEY3", (void *)&press_cnt[2]);
free_irq(IRQ_EINT2, (void *)&press_cnt[2]);
```

10.2.3 驱动程序进行中断处理的实例代码分析

现在对照 s3c24xx_buttons_v1.c 和 button_test.c 中的代码和注释,并根据实验的现象,进行分析。

s3c24xx_buttons_v1.c:

```

65 static irqreturn_t buttons_interrupt(int irq, void * dev_id)
66 {
67     volatile int * press_cnt = (volatile int *)dev_id;
69     * press_cnt = * press_cnt + 1; /* 按键计数加 1 */
70     buttons_dev.ev_press = 1; /* 表示中断发生了 */
71     wake_up_interruptible(&(buttons_dev.buttons_waitqueue)); /* 唤醒休眠的进程 */
74     return IRQ_RETVAL(IRQ_HANDLED);
75 }

80 static int s3c24xx_buttons_open(struct inode * inode, struct file * file)
81 {
85     for (i = 0;
86         i < sizeof(buttons_dev.button_irqs)/sizeof(buttons_dev.button_irqs[0]);
87         i++) {
88         // 注册中断处理函数
89         err = request_irq(buttons_dev.button_irqs[i].irq, buttons_interrupt,
90             buttons_dev.button_irqs[i].flags,
91             buttons_dev.button_irqs[i].name,
92             (void *)&(buttons_dev.press_cnt[i]));
95     }
105 }

110 static int s3c24xx_buttons_close(struct inode * inode, struct file * file)
111 {
114     for (i = 0;
115         i < sizeof(buttons_dev.button_irqs)/sizeof(buttons_dev.button_irqs[0]);
116         i++) {
117         // 释放已经注册的中断
118         free_irq(buttons_dev.button_irqs[i].irq, (void *)&(buttons_dev.press_cnt[i]));
119     }
121 }

126 static int s3c24xx_buttons_read(struct file * filp, char __user * buff, size_t count, loff_t * offp)
127 {
130     /* 如果 ev_press 等于 0,休眠 */
131     wait_event_interruptible(buttons_dev.buttons_waitqueue, (buttons_dev.ev_press != 0));
133     /* 执行到这里时, ev_press 等于 1,将它清 0 */
134     buttons_dev.ev_press = 0;

```

```

136  /* 将按键状态复制给用户,并清 0 */
137  err = copy_to_user(buff, (const void *)buttons_dev.press_cnt,
138                    min(sizeof(buttons_dev.press_cnt), count));
139  memset((void *)buttons_dev.press_cnt, 0, sizeof(buttons_dev.press_cnt));
142 }

```

button_test.c 如下:

```

13  fd = open("/dev/buttons", 0); // 打开设备
20  // 这是个无限循环,进程有可能在 read 函数中休眠,当有按键被按下时,它才返回
21  while (1) {
22      // 读出按键被按下的次数
23      ret = read(fd, press_cnt, sizeof(press_cnt));
28      printf("read buttons successfully, begin print the result:\n");
29      for (i = 0; i < sizeof(press_cnt)/sizeof(press_cnt[0]); i++) {
30          // 如果被按下的次数不为 0,打印出来
31          if (press_cnt[i])
32              printf("K %d has been pressed %d times! \n", i + 1, press_cnt[i]);
33      }
35  }
37  close(fd);

```

当测试程序执行 13 行打开按键设备时,驱动将执行 s3c24xx_buttons_open 函数:89 行将中断处理函数 buttons_interrupt 注册进操作系统,并指定将来中断产生时操作系统传给 buttons_interrupt 的第二个参数是记录对应按键被按下次数的变量的地址。

当测试程序执行 23 行读取按键设备时,驱动将执行 s3c24xx_buttons_read 函数:131 执行时由于尚无按键被按下,因此驱动将测试程序投入睡眠。

当某个按键被按下时,操作系统将调用中断处理函数 buttons_interrupt 执行,从而完成:69 行将对应按键次数加 1(从 0 变为 1);70、71 行唤醒测试程序;74 行中断处理程序将 IRQ_HANDLED 返回操作系统,告知操作系统本次中断处理成功完成。

之后操作系统将调度测试程序再次运行,这将使 s3c24xx_buttons_read 函数继续执行:134 行清除表示已经按下键的标志,以便为接收下一次按键做好准备;137 行 copy_to_user 将 4 个按键被按下的次数返回用户空间。

测试程序继续运行,29~33 行将驱动返回的 4 个按键被按下的次数分别打印出来。然后循环回去读取下次按键的情况,这将使测试程序再度睡眠到下次用户按下按键。

当测试程序退出时(如收到 sigint),类似 37 行的操作将会执行,这将导致驱动执行 s3c24xx_buttons_close 函数:118 行 free_irq 将释放在 s3c24xx_buttons_open 中用 request_irq 注册的中断。这样一来,操作系统将不会再响应任何按键中断。

总结：

要进行中断处理的驱动程序一般要实现 3 类函数：模块初始化、注销函数；设备驱动的功能函数（open、read、write、release）；中断处理函数。中断处理函数的注册和释放一般放在模块初始化、注销函数中，或者放在功能函数 open、release 中。当功能函数发现某个条件不成立时，一般会调用 wait 将进程投入睡眠。中断处理函数在中断产生时被操作系统调用，它将对产生中断的设备进行处理、更新数据，当发现条件成立时将唤醒睡眠进程。

10.2.4 其他关于中断的内核 API

1. 启用和禁用单个中断

- void enable_irq(int irq);
- void disable_irq(int irq); //禁用中断后，要等到其他正在进行处理的同类型中断全部处理完成后，才返回。这就是所谓的同步。

- void disable_irq_nosync(int irq); //禁用中断后，立即返回

2. 启用和禁用全部中断

- void local_irq_enable(void);
- void local_irq_restore(unsigned long flags);
- void local_irq_disable(void);
- void local_irq_save(unsigned long flags);

10.3 内核时间与内核定时器

10.3.1 内核中如何记录时间

任何程序都需要时间控制，其主要目的是：

- 测量时间流逝和比较时间。
- 知道当前时间。
- 指定时间量的延时操作。

为达到这个目的，应用程序使用日历时间（年月日时分秒）或者自 1970 年 1 月 1 日零时零分零秒到当前的秒数来度量时间的流逝，但内核中需要更加有精度的时间度量，因此内核使用时钟滴答来记录时间。时钟中断发生后内核内部时间计数器增加 1（即：增加 1 个时钟滴答），系统引导时为 0，当前值为自上次系统引导以来的时钟滴答数，程序可通过内核定义的全局变量 jiffies_64 或 jiffies 来访问。真实硬件上每秒的滴答数从 50~1200 不等，x86 上默认为 1000，而 s3c2440 上默认为 200，出于统一编程接口的考虑，内核定义了一个宏 HZ，它表示 1s

第10章 Linux 驱动中的中断编程

的嘀嗒数,可供程序使用。

(1) Jiffies 和 jiffies_64 是 unsigned long 类型只读变量,其用法如下:

- #include <linux/jiffies.h>
- unsigned long j, stamp_1, stamp_half, stamp_n;
- j = jiffies; /* read the current value */
- stamp_1 = j + HZ; /* 1 second in the future */
- stamp_half = j + HZ/2; /* half a second in the future */
- stamp_n = j + n * HZ / 1000; /* n milliseconds in the future */

注意:32 位平台上当 HZ 是 1000 时,计数器只是每 50 天溢出一次,必要时代码应当准备处理这个事件。

(2) 比较两个时间的大小,常用内核的如下宏定义:

- #include <linux/jiffies.h>
- int time_after(unsigned long a, unsigned long b);
- int time_before(unsigned long a, unsigned long b);
- int time_after_eq(unsigned long a, unsigned long b);
- int time_before_eq(unsigned long a, unsigned long b);

它们分别在时间 a 在时间 b 之后、之前、之后或相等、之前或相等的时候为真,反之为假

(3) 求两个 jiffies 实例之间的差:

- diff = (long)t2 - (long)t1;

(4) 可以转换一个 jiffies 差为毫秒:

- msec = diff * 1000 / HZ;

(5) jiffies 与日历时间的转换函数:

- #include <linux/time.h>
- unsigned long timespec_to_jiffies(struct timespec □value);
- void jiffies_to_timespec(unsigned long jiffies, struct timespec □value);
- unsigned long timeval_to_jiffies(struct timeval □value);
- void jiffies_to_timeval(unsigned long jiffies, struct □timeval)

10.3.2 内核定时器 API

1. 概述

(1) 无论何时你需要调度一个动作以后发生,就可以使用内核定时器,如:当硬件无法发出中断时,可通过使用内核定时器,以定期的间隔检查一个设备的状态。

(2) 一个内核定时器是一个数据结构,它指导内核在一个用户定义的时间,使用一个用户定义的参数,执行一个用户定义的函数。

(3) 由内核线程——软中断(ksoftirqd/0)调度执行。

- 一个 CPU, 一个 ksoftirqd
- ksoftirqd 属于 atomic context
- ksoftirqd 运行时, 不禁用 irq

2. 定时器 API

(1) 包含的头文件: #include <linux/timer.h>

(2) 最主要数据结构体:

```
struct timer_list {
    unsigned long expires;
    void (* function)(unsigned long);
    unsigned long data;
    其他字段
};
```

(3) 静态初始化定时器结构体:

```
struct timer_list timerval = TIMER_INITIALIZER(_function, _expires, _data);
```

(4) 动态初始化定时器结构体:

```
setup_timer(struct timer_list * timer, _function, _data); //初始化 function 和 data 后, 调用
init_timer
void init_timer(struct timer_list * timer); //初始化其他字段
timer->expires = jiffies + HZ/10; //最后手工指定触发时间
```

(5) 将已初始化定时器加入系统定时器链表:

```
void add_timer(struct timer_list * timer);
```

注: 定时器执行后会自动退出系统链表, 如需再次执行, 需要更新 expires 后, 再次加入系统链表

(6) 更新一个定时器的超时时间, 同时将其加入系统定时器链表:

```
int mod_timer(struct timer_list * timer, unsigned long expires);
```

(7) 将已链入但还未执行的定时器退出系统定时器链表:

```
int del_timer(struct timer_list * timer);
```

10.3.3 内核定时器与内核时间的应用案例——按键消抖

在“驱动程序中的中断处理”一节的实验中, 如果将编译 s3c2440_buttons_v2.5.c 改为编

译 s3c2440_buttons_v1.c (rm s3c2440_buttons.c; ln -s s3c2440_buttons_v1.c s3c2440_buttons.c; make), 可能注意到了, 当仅按一次按键的时候, 结果显示有多次按键, 这跟我们理想的状态有些差异, 出现这种情况是由于存在按键抖动。以下的显示是由于一次人工按键共产生了 4 次中断: 第 1 次中断唤醒了测试进程, 并且测试进程在第 2 次中断产生前成功读取了按键次数; 第 2 次中断唤醒了测试进程, 但在测试进程尚未成功读取按键次数前, 第 3 次中断迅速产生, 因此当测试进程赶在第 4 次中断产生前读取按键次数时, 该值已经变为了 2; 第 4 次中断唤醒了测试进程, 同时由于按键已经平稳, 不再产生新的中断, 从而测试进程成功读取了按键次数。

```
# ./button_test
open success
read buttons successfully, begin print the result:
K1 has been pressed 1 times!
read buttons successfully, begin print the result:
K1 has been pressed 2 times!
read buttons successfully, begin print the result:
K1 has been pressed 1 times!
```

按键的物理特性决定它肯定会存在抖动。因此要消除抖动, 只能采用软件消抖的方法。请编译 s3c24xx_buttons_v2.5.c, 可得到消除了按键抖动的驱动版本。

```
# ./button_test
open success
read buttons successfully, begin print the result:
K1 has been pressed 1 times!
```

s3c24xx_buttons_v2.5.c 是怎么做到的呢? 其消抖方案是: 只在第一次中断产生时才记录按键次数; 在第一次中断产生后, 多次延时 0.1s 直到检测到按键已被放开, 最后再做一次延迟 0.1s 的操作, 然后允许中断可以再次记录按键次数。这样一来, 第一次延迟 0.1s 消除了按下键时的抖动, 最后一次延迟 0.1s 消除了放开键时的抖动。这个方案用到的技术主要就是内核定时器和内核时间。

下面对主要实现代码进行说明:

```
58 static struct buttons_dev_t buttons_dev =
59 {
60     .ev_press    = 0,
61     .press_cnt    = {0, 0, 0, 0},
62     .button_irqs  = {
63         {IRQ_EINT8, IRQF_TRIGGER_FALLING, "KEY1"}, /* K1 */
64         {IRQ_EINT11, IRQF_TRIGGER_FALLING, "KEY2"}, /* K2 */
```

```

65     {IRQ_EINT13,  IRQF_TRIGGER_FALLING, "KEY3"}, /* K3 */
66     {IRQ_EINT14,  IRQF_TRIGGER_FALLING, "KEY4"} /* K4 */
67 },
68 .firstint      = 0
69 };

79     if (buttons_dev.firstint == 1)
80         return IRQ_RETVAL(IRQ_NONE);
82     buttons_dev.firstint = 1;

```

中断处理函数如果发现不是第一次中断,就不累加按键次数。

```

048     struct buttons_dev_t {
049         wait_queue_head_t buttons_waitqueue;
050         volatile int ev_press;
051         volatile unsigned int press_cnt[BUTTON_NUM];
052         struct button_irq_desc button_irqs[BUTTON_NUM];
053         struct cdev cdev;
054         struct timer_list button_timers[BUTTON_NUM]; /* buttons delay timer */
055         int firstint;
056     };

277     for (i = 0; i < BUTTON_NUM; i++)
278         setup_timer(&(buttons_dev.button_timers[i]), buttons_timer_handler, i);

```

这是在模块的初始化函数中对 4 个定时器(分别对应 4 个按键)进行初始化。

```

105     buttons_dev.button_timers[butno-1].expires = jiffies + HZ/10; /* delay 0.1s */
106     add_timer(&(buttons_dev.button_timers[butno-1]));

```

中断处理函数做第一次 0.1s 的延时。

```

152 static void buttons_timer_handler(unsigned long data)
153 {
154     static int shouldfinish = 0;
155     if (shouldfinish) {
156         shouldfinish = 0;
157         buttons_dev.firstint = 0;
158         return;
159     }
160     if (! keydown(data))
161         shouldfinish = 1;

```



```

164     mod_timer(&(buttons_dev.button_timers[data]), jiffies + HZ/10); /* delay 0.1s */
165 }

```

定时器处理函数进行多次 0.1s 的延时(164 行),并在判定出按键已被放开的情况下作最后一次 0.1s 延迟(162~164 行),之后允许中断处理函数可以再次记录按键次数(159 行)。

```

41 #define GPGDAT      0x56000064
296 if (! (button1234virtaddr = ioremap(GPGDAT, 0x4))) {
297     printk(KERN_NOTICE "ioremap failed\n");
298     result = -ENOMEM;
299     goto fail_ioremap_GPGDAT;
300 }
112 static int keydown(unsigned long data)
113 {
114     int result;
115     switch (data) {
116         case 0:
117             PDEBUGG("K1\n");
118             if ((ioread32(button12virtaddr) & (1<<0)) == 0)
119                 result = 1;
120             else
121                 result = 0;
122             break;
123         case 1:
147     }
149     return result;
150 }

```

keydown 函数用于判定某个按键是否按下。按下返回真,放开返回假。关于获得按键是否被按下的机制,类似于使用 GPIO 控制 LED 灯,请参阅“ARM 体系结构与编程”中的相关内容。

10.3.4 如何在内核中实现延时

设备驱动常常需要延后一段时间执行一个特定片段的代码,以便允许硬件完成某个任务。延时一般区分为短延时和长延时。

1. 短延时

当一个设备驱动需要等待硬件的反应时间,涉及的延时常常是最多几个毫秒。此种延时就是短延时,一般采用忙等待。相关函数如下:

- #include <linux/delay.h>

- `void ndelay(unsigned long nsecs);`
- `void udelay(unsigned long usecs);`
- `void mdelay(unsigned long msecs);`

2. 长延时

如果需要延后较长时间,就可以采用长延时。长延时可分为忙等待和让出 CPU 两种方式。

(1) 忙等待:

```
unsigned long j1 = jiffies + 2 * HZ;
while (time_before(jiffies, j1))
    cpu_relax();
```

注: `cpu_relax` 的调用使用了一个特定于体系的方式,此时没有用处理器做事情。

(2) 让出处理器:

```
unsigned long j1 = jiffies + 3600 * HZ;
while (time_before(jiffies, j1))
    schedule();
```

忙等待强加了一个重负载给系统总体,通过释放 CPU 改变这种状况。

(3) 此外,如果驱动使用一个等待队列来等待某些其他事件,但是你也想确保它在一个确定时间段内能够运行,而不是永久等待,那么可以使用超时:

- `#include <linux/wait.h>`
- `long wait_event_timeout(wait_queue_head_t q, condition, long timeout);`
- `long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);`

10.4 中断顶半部与底半部

10.4.1 区分和使用中断顶半部与底半部的原因

- One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.
- 中断处理中存在的一个最主要问题是:如何在中断处理中执行很耗时的操作。一方

面,在响应外部设备的硬件中断时,通常情况下都会做大量的操作,很耗时间;另一方面,中断处理程序需要尽快结束,以避免阻塞其他中断过长时间。这样两个需求(即:需要做大量工作,又需要迅速完成并退出)相互冲突,给硬件驱动程序的编写者带来了很大的困难。

- Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt—the one you register with `request_irq`. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits; this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.
- Linux 以及其他许多操作系统解决这个难题的方法是,把中断处理需要完成的工作划分为两部分:顶半部和底半部。顶半部是由驱动编写者通过内核 API——`request_irq` 注册进操作系统的例程序,用于快速响应中断,并会调度底半部程序在稍后安全的时间里运行,而由底半部程序去完成耗时的操作。顶半部例程和底半部程序最大的区别是:在底半部程序运行期间,所有的中断都没有被阻塞。这也是为什么说底半部运行在一个更安全的时间。中断顶半部和底半部典型的分工是,顶半部将硬件设备中的数据保存到特定的内存缓冲区后,调度与它配对的底半部运行,然后立即退出(这个操作的完成非常快);稍后底半部完成其他耗时的操作,例如:唤醒进程、启动另一个 I/O 操作等等。这样分工的好处在于:当底半部在工作的时候,顶半部可以服务新到来的硬件中断,而不受影响。
- The Linux kernel has two different mechanisms that may be used to implement bottom-half processing. Tasklets are often the preferred mechanism for bottom-half processing; they are very fast, but all tasklet code must be atomic. The alternative to tasklets is workqueues, which may have a higher latency but that are allowed to sleep.
- Linux 内核有两种机制用于实施中断底半部,它们是:tasklet 和 workqueue。tasklet 通常是优先选择的方案,因为它非常快,但缺点是 tasklet 要求所有的操作都必须是原子操作,即:在 tasklet 的代码中不允许出现睡眠等非原子的操作。而 workqueue 虽然比 tasklet 更慢些,但允许睡眠。

可见,操作系统将中断处理区分为上、下半部的目的是:尽可能缩短中断被阻塞的时间,以提高系统的实时性。中断底半部的 2 种实现机制(tasklet 和 workqueue)的最重要区别是:能否在底半部中睡眠。2 种机制各有其不同应用环境和优缺点,它们的区别和不同应用环境请参见后续的实例分析和 10.4.4 节的总结。

10.4.2 tasklet 机制与编程实例

1. 原 理

- (1) 类似内核定时器,调度一个任务稍后执行。主要用于中断的底半部。
- (2) Tasklet 是一个数据结构,它指导内核在稍后 CPU 空闲的时间(但不能指定确定时间),使用一个用户定义的参数,执行一个用户定义的函数。
- (3) 由内核线程——软中断(ksoftirqd/0)调度执行。在 tasklet 内不能睡眠。
- (4) 运行在调度它们的同一个 CPU 上。

2. tasklet 使用的内核 API

- (1) 需要包含头文件:include <linux/interrupt.h>

- (2) 主要数据结构体:

```
struct tasklet_struct {
    void (* func)(unsigned long);
    unsigned long data;
    其他字段
};
```

- (3) 完全初始化 tasklet

- void tasklet_init(struct tasklet_struct * t, void (* func)(ulong), ulong data);
- DECLARE_TASKLET(name, func, data);
- DECLARE_TASKLET_DISABLED(name, func, data);

- (4) void tasklet_schedule(struct tasklet_struct * t);

- 将 tasklet 加入系统链表,即调度 tasklet 运行。
- tasklet 执行后会自动退出系统链表,如需再次执行,需再次加入系统链表。

- (5) void tasklet_hi_schedule(struct tasklet_struct * t);

- 以高优先级调度 tasklet

- (6) void tasklet_kill(struct tasklet_struct * t);

- 将 tasklet 退出系统链表

- (7) void tasklet_disable(struct tasklet_struct * t);

- (8) void tasklet_enable(struct tasklet_struct * t);

- tasklet 在 disable 的状态下可以被 tasklet_schedule, 但该 tasklet 必须等到 tasklet_enable 后才能真正运行。
- tasklet_enable 的次数必须与 tasklet_disable 的次数相等。

3. tasklet 实例(s3c24xx_buttons_v3.5.c)

(1) 实例功能描述。按下 K1(K2、K3、K4)键时, LED1(LED2、LED3、LED4)闪烁一次。由于 LED 灯闪烁这个操作比较耗时(需 2s), 因此不能在中断顶半部中执行, 所以由顶半部调度它在中断底半部(本实例在 tasklet)中运行。

(2) 主要实现代码分析与说明:

```
314     for (i = 0; i < BUTTON_NUM; i++) /* initialize tasklets */
315         tasklet_init(&(buttons_dev.button_tasklets[i]), button_do_tasklet, i);
```

模块初始化时, 完成对 tasklet 的初始化:

```
150     tasklet_schedule(&(buttons_dev.button_tasklets[butno - 1])); /* schedule tasklet */
```

在中断处理程序(顶半部)中, 调度 tasklet(底半部)在稍后安全的时间(非顶半部时间)运行:

```
82 static void button_do_tasklet(unsigned long data)
83 {
84     PDEBUG("enter tasklet, current pid is %d, button number is %li\n", current->pid, data
+ 1);
85     switch (data) {
86         case 0:
87             ledon(0);
88             mdelay(1000);
89             ledoff(0);
90             mdelay(1000);
91             ledon(0);
92             break;
93         case 1:
116     }
117     PDEBUG("exit tasklet, current pid is %d, button number is %li\n", current->pid, data + 1);
118 }
```

在 tasklet 中调用 mdelay 完成了毫秒级的延时, 由于 mdelay 是忙等待, 而不是睡眠, 所以可以在 tasklet(中断上下文)中使用。

在 tasklet 中, 调用 ledon 和 ledoff 完成亮灯和灭灯的操作。但 ledon 和 ledoff 并非是在本程序中定义的, 而是在其他驱动内核模块中定义的, 所以在编译本模块时会有警告, 告知这

两个函数在本模块中尚未定义。

```
/work/studydriver/buttons $ make
make -C /work/sysbuild/linux-2.6.22.6/ M=/work/studydriver/buttons modules
make[1]: Entering directory ~/work/system/linux-2.6.22.6
CC [M] /work/studydriver/buttons/s3c24xx_buttons.o
Building modules, stage 2.
MODPOST 1 modules
WARNING: "ledoff" [/work/studydriver/buttons/s3c24xx_buttons.ko] undefined!
WARNING: "ledon" [/work/studydriver/buttons/s3c24xx_buttons.ko] undefined!
CC /work/studydriver/buttons/s3c24xx_buttons.mod.o
LD [M] /work/studydriver/buttons/s3c24xx_buttons.ko
make[1]: Leaving directory ~/work/system/linux-2.6.22.6
```

也正是由于这个原因,必须在加载本模块之前,先加载 ledon 和 ledoff 被定义的模块:

```
# insmod s3c24xx_buttons.ko
s3c24xx_buttons: Unknown symbol ledoff
s3c24xx_buttons: Unknown symbol ledon
insmod: cannot insert 's3c24xx_buttons.ko': Unknown symbol in module (-1): No such file or
directory
# insmod ../ledsdriver/ledsv2.ko
leds initialized
# insmod s3c24xx_buttons.ko
buttons initialized
```

而且在 ledon 和 ledoff 被定义的模块(ledsv2.c)中,还必须用内核定义的 EXPORT_SYMBOL 宏将这两个函数导出到内核全局符号表,以供别的模块使用。

```
93 EXPORT_SYMBOL(ledon);
100 EXPORT_SYMBOL(ledoff);
```

由于这两个函数是被导出到内核全局符号表的,因此不能与内核全局符号表中的其他函数(内核 API)重名,否则就造成了内核名字空间污染。

(3) 运行结果分析

```
1 # ./button_test
2 open success
3 buttons: who(63) enter interrupt
4 buttons: quit interrupt
5 buttons: who(63) enter interrupt
6 buttons: enter tasklet, current pid is 0, button number is 1
```

```

7 buttons: who(63) enter interrupt
8 buttons: who(63) enter interrupt
9 buttons: who(63) enter interrupt
10 buttons: who(63) enter interrupt
11 buttons: who(63) enter interrupt
延迟时长约 2s
12 buttons: exit tasklet, current pid is 0, button number is 1
13 read buttons successfully, begin print the result;
14 K1 has been pressed 1 times!

```

第 6 行说明 tasklet 的确是在稍后安全的时间(即:在顶半部退出后)运行的。

第 5 行说明 tasklet 的运行时间的确不能精确确定(即:并不是顶半部退出后立即运行),但会在顶半部退出后(此时 CPU 空闲)非常快就运行。

第 7~11 行出现在第 6 行和第 12 行之间,说明底半部运行期间,的确不禁用中断(顶半部),因此 tasklet 运行的时间段的确比较安全(这期间可以服务其他中断)。

第 13~14 行验证了 tasklet 的确是运行在中断上下文中。虽然测试进程在第 4 行之前就已经被唤醒,但它却要等到 tasklet 执行完(需要约 2s 的时间),到 12 行后才有机会运行。这是因为 tasklet 运行在中断上下文中,期间 OS 除了响应中断外,不会进行进程调度。

10.4.3 workqueue 机制与编程实例

1. 原理

(1) 类似 tasklet,提交一个 work 到 OS 的特定 workqueue——events/0 上,以便稍后执行。主要用于中断的底半部。

(2) work 是一个数据结构,它指导内核在稍后的时间(但不能指定确定时间)或至少延迟一个确定的时间后,执行一个用户定义的函数(不能指定参数)。

(3) 由内核线程——events/0 调度执行(注:内核线程——events/0 不处于 atomic context)。

(4) 缺省运行在调度它们的同一个 CPU 上。

```

# ps
  PID  Uid    VSZ Stat Command
   1 root    3092 S   init
   2 root         SW< [kthreadd]
   3 root         SWN [ksoftirqd/0] //N 表示低优先级, <表示高优先级
   4 root         SW< [events/0]
   5 root         SW< [khelper]
  41 root         SW< [kblockd/0]

```



```

42 root      SW< [ksuspend_usbd]
45 root      SW< [khubd]
47 root      SW< [kseriod]
59 root      SW  [pdflush]
60 root      SW  [pdflush]
61 root      SW< [kswapd0]
62 root      SW< [aio/0]
177 root     SW< [mtdblockd]
226 root     SWN [jffs2_gcd_mtd2]
248 root     1952 S   /usr/sbin/vsftpd
249 root     3096 S   - sh
250 root     3092 S   /usr/sbin/telnetd
253 www      3092 S   /usr/sbin/httpd -h /www -u www
256 root     3096 R   ps

```

2. workqueue API

- DECLARE_WORK(name, void (* function)(work_struct *))
 - 编译时完全初始化一个 work_struct
- INIT_WORK(struct work_struct * work, void (* function)(work_struct *))
 - 完全初始化一个 work_struct
- int schedule_work(struct work_struct * work)
 - 将 work 提交给默认的 kevent
- int schedule_delayed_work(struct work_struct * work, ulong delay)
 - 将 work 提交给默认的 kevent,但延后 delay 后调度
- int cancel_delayed_work(struct work_struct * work);
 - 取消提交的 work
- void flush_scheduled_work(void)
 - 将 kevent 中的 work 全部取消
- struct workqueue_struct * create_workqueue(const char * name)
- struct workqueue_struct * create_singlethread_workqueue(const char * name)
 - 创建自己的 workqueue,并与每个 CPU(或单个 CPU)绑定
- int queue_work(struct workqueue_struct * queue, struct work_struct * work)
- int queue_delayed_work(struct workqueue_struct * queue, struct work_struct * work, ulong delay)
 - 将 work 提交给 queue
- void flush_workqueue(struct workqueue_struct * queue)

■ 将 queue 中的 work 全部取消

● void destroy_workqueue(struct workqueue_struct * queue)

■ 删除自己创建的 queue

3. workqueue 实例(s3c24xx_buttons_v4.5.c)

(1) 实例功能描述。按下 K1(K2、K3、K4)键时,LED1(LED2、LED3、LED4)闪烁一次。由于 LED 灯闪烁这个操作比较耗时(需 6s),因此不能在中断顶半部中执行,所以由顶半部调度它在中断底半部(本实例在 workqueue)中运行。

(2) 主要实现代码分析与说明:

```
397 INIT_WORK(&(buttons_dev.buttons_workqueue), button_do_workqueue);
```

模块初始化时,完成对 workqueue 的初始化:

```
229 buttons_dev.dkeynum = butno - 1;
230 schedule_work(&(buttons_dev.buttons_workqueue)); /* schedule workqueue */
```

在中断处理程序(顶半部)中,调度 workqueue(底半部)在稍后安全的时间(非顶半部时间)运行:

```
88 static void button_do_workqueue(struct work_struct * wq)
89 {
90     PDEBUG("enter workqueue, current pid is %d, button number is %d\n", current->pid, but-
tons_dev.dkeynum + 1);
101     switch (buttons_dev.dkeynum) {
102         case 0;
103             ledon(0);
104             ssleep(3);
105             ledoff(0);
106             ssleep(3);
107             ledon(0);
108             break;
123         case 3;
124             ledon(3);
125             mdelay(3000);
126             ledoff(3);
127             mdelay(3000);
128             ledon(3);
129             break;
132     }
133     PDEBUG("exit workqueue, current pid is %d, button number is %d\n", current->pid, but-
```

```
tons_dev.dkeynum + 1);
134 }
```

在 workqueue 与 tasklet 一个重大区别就是可以睡眠,所以它可以调用 mdelay 进行忙等待,也可以安全地调用 ssleep 睡眠。

(3) 运行结果分析:

```
1 # ./button_test
2 open success
```

按下 K1 键

```
4 buttons: who(63) enter interrupt, current pid is 0
5 buttons: in interrupt, CPSR = 60000093, SPSR = 80000013
6 buttons: who(63) quit interrupt, current pid is 0, button number is 1
7 buttons: enter workqueue, current pid is 4, button number is 1
8 buttons: in workqueue, CPSR = 60000013, SPSR = 60000013
9 read buttons successfully, begin print the result;
10 K1 has been pressed 1 times!
11 buttons: who(63) enter interrupt, current pid is 0, button number is 0
12 buttons: who(63) enter interrupt, current pid is 0, button number is 0
13 buttons: who(63) enter interrupt, current pid is 0, button number is 0
14 延迟时长约 6 秒
15 buttons: exit workqueue, current pid is 4, button number is 1
```

按下 K4 键

```
17 buttons: who(16) enter interrupt, current pid is 0
18 buttons: in interrupt, CPSR = 60000013, SPSR = 60000013
19 buttons: who(16) quit interrupt, current pid is 0, button number is 4
20 buttons: who(16) enter interrupt, current pid is 0
21 buttons: who(16) enter interrupt, current pid is 0
22 buttons: enter workqueue, current pid is 4, button number is 4
23 buttons: in workqueue, CPSR = 60000013, SPSR = 60000013
24 buttons: who(16) enter interrupt, current pid is 4
25 buttons: who(16) enter interrupt, current pid is 4
26 延迟时长约 6 秒
27 buttons: exit workqueue, current pid is 4, button number is 4
28 read buttons successfully, begin print the result;
29 K4 has been pressed 1 times!
```

第 5 行说明本次中断处理程序(顶半部)运行时,禁用了 IRQ 异常(请比对第 18 行)。这是注册中断时指定中断选项的结果:

```

75  .button_irqs = {
76      {IRQ_EINT19, IRQF_TRIGGER_FALLING|IRQF_DISABLED, "KEY1"}, /* K1 */
77      {IRQ_EINT11, IRQF_TRIGGER_FALLING, "KEY2"}, /* K2 */
78      {IRQ_EINT2, IRQF_TRIGGER_FALLING, "KEY3"}, /* K3 */
79      {IRQ_EINT0, IRQF_TRIGGER_FALLING, "KEY4"} /* K4 */
80  }

```

不过为何处于 SVC 模式而不是 IRQ 模式,是因为内核处理中断时为了能够中断嵌套进行了模式切换。

第 7 行验证了由内核线程——events/0(它的 pid 是 4)调度执行 workqueue。

第 9、10 行出现在第 7 与 15 行之间,说明 workqueue 的确可以睡眠,从而使得操作系统可以调度其他用户进程运行。

第 20、21 行说明 workqueue 的运行时间的确不能精确确定(即:并不是顶半部退出后立即运行),但会在顶半部退出后(此时 CPU 空闲)非常快就运行。

第 24~25 行出现在第 22 行和第 27 行之间,说明底半部运行期间,的确不禁用中断(顶半部),因此 workqueue 运行的时间段的确比较安全(这期间可以服务其他中断)。

测试进程在 19 行之后就已经被唤醒,但第 28~29 出现在 27 行之后,说明了虽然 workqueue 运行在进程上下文中,但它是高优先级内核线程,所以只要它不主动让出 CPU,操作系统也几乎不会(不是肯定不会)调度普通用户进程运行。这一点与 tasklet 不同,tasklet 运行在中断上下文中,期间 OS 除了响应中断外,肯定不会进行进程调度。

(4) 验证在 tasklet 中不能睡眠。将第 54 行注释掉,从而使用 tasklet。

```
54 // #define USEWORKQUEUE
```

```

136 static void button_do_tasklet(unsigned long data)
137 {
149     switch (data) {
150         case 0:
151             ledon(0);
152             mdelay(1000);
153             ledoff(0);
154             mdelay(1000);
155             ledon(0);
156             break;
171         case 3:
172             ledon(3);
173             mdelay(1000);
174             ledoff(3);

```



```

175         ssleep(1);
176         //mdelay(1000);
177         ledon(3);
178         break;
181     }
183 }

```

运行的结果如下,验证了在 tasklet 的确不能睡眠(有时间的话,就好好看看 OOP 的输出吧,也许你能发现一些内核的小秘密!):

```

# ./button_test
open success

```

按下 K1 键

```

buttons: who(63) enter interrupt, current pid is 0
buttons: in interrupt, CPSR = 60000093, SPSR = 93
buttons: who(63) quit interrupt, current pid is 0, button number is 1
buttons: who(63) enter interrupt, current pid is 0
buttons: enter tasklet, current pid is 0, button number is 1
buttons: in tasklet, CPSR = 60000013, SPSR = 60000013
buttons: who(63) enter interrupt, current pid is 0
buttons: who(63) enter interrupt, current pid is 0
buttons: who(63) enter interrupt, current pid is 0
buttons: who(63) enter interrupt, current pid is 0
buttons: who(63) enter interrupt, current pid is 0
buttons: exit tasklet, current pid is 0, button number is 1
read buttons successfully, begin print the result:
K1 has been pressed 1 times!

```

按下 K4 键

```

buttons: who(16) enter interrupt, current pid is 0
buttons: in interrupt, CPSR = 60000013, SPSR = 80000013
buttons: who(16) quit interrupt, current pid is 0, button number is 4
buttons: enter tasklet, current pid is 0, button number is 4
buttons: in tasklet, CPSR = 60000013, SPSR = 60000013
BUG: scheduling while atomic: swapper/0x00000100/0
[<c0028c90>] (dump_stack+0x0/0x14) from [<c0200cf0>] (schedule+0x50/0x750)
[<c0200ca0>] (schedule+0x0/0x750) from [<c0201dec>] (schedule_timeout+0x8c/0xbc)
[<c0201d60>] (schedule_timeout+0x0/0xbc) from [<c0201e68>] (schedule_timeout_uninter-
ruptible+0x24/0x28)

```

```

r8;3001f48c r7;c02ce280 r6;00000004 r5;c0282000 r4;ffffffff
[<c0201e44>] (schedule_timeout_uninterruptible + 0x0/0x28) from [<c00442f4>] (msleep +
0x1c/0x28)
[<c00442d8>] (msleep + 0x0/0x28) from [<bf00208c>] (button_do_tasklet + 0x8c/0x1c8
[s3c24xx_buttons])
[<bf002000>] (button_do_tasklet + 0x0/0x1c8 [s3c24xx_buttons]) from [<c00404c0>] (tasklet
_action + 0x88/0xdc)
r6;0000000a r5;c02ce2a4 r4;00000000
[<c0040438>] (tasklet_action + 0x0/0xdc) from [<c0040000>] (__do_softirq + 0x5c/0xc8)
r5;c02ce2e4 r4;00000001
[<c003ffa4>] (__do_softirq + 0x0/0xc8) from [<c0040224>] (irq_exit + 0x44/0x4c)
r7;00000000 r6;c02d2edc r5;c028b0a8 r4;00000010
[<c00401e0>] (irq_exit + 0x0/0x4c) from [<c002404c>] (asm_do_IRQ + 0x4c/0x60)
[<c0024000>] (asm_do_IRQ + 0x0/0x60) from [<c0024a24>] (__irq_svc + 0x24/0xa0)
Exception stack(0xc0283f54 to 0xc0283f9c)
3f40;                                00000000 ffffffff f020000c
3f60; 80000013 c0025974 c0282000 c0020f28 c02dfb58 3001f48c 41129200 3001f458
3f80; c0283fa8 c0283f9c c0283f9c c00259d4 c00259e0 80000013 ffffffff
r7;c02dfb58 r6;00000001 r5;f0000000 r4;ffffffff
[<c0025974>] (default_idle + 0x0/0x78) from [<c0025a34>] (cpu_idle + 0x48/0x64)
[<c00259ec>] (cpu_idle + 0x0/0x64) from [<c02006c4>] (rest_init + 0x48/0x58)
r5;c02bc328 r4;c02d12d4
[<c020067c>] (rest_init + 0x0/0x58) from [<c0008938>] (start_kernel + 0x27c/0x2e4)
[<c00086bc>] (start_kernel + 0x0/0x2e4) from [<30008030>] (0x30008030)
bad; scheduling from the idle thread!
[<c0028c90>] (dump_stack + 0x0/0x14) from [<c0200d3c>] (schedule + 0x9c/0x750)
[<c0200ca0>] (schedule + 0x0/0x750) from [<c0201dec>] (schedule_timeout + 0x8c/0xbc)
[<c0201d60>] (schedule_timeout + 0x0/0xbc) from [<c0201e68>] (schedule_timeout_uninter-
ruptible + 0x24/0x28)
r8;3001f48c r7;c02ce280 r6;00000004 r5;c0282000 r4;ffffffff
[<c0201e44>] (schedule_timeout_uninterruptible + 0x0/0x28) from [<c00442f4>] (msleep +
0x1c/0x28)
[<c00442d8>] (msleep + 0x0/0x28) from [<bf00208c>] (button_do_tasklet + 0x8c/0x1c8
[s3c24xx_buttons])
[<bf002000>] (button_do_tasklet + 0x0/0x1c8 [s3c24xx_buttons]) from [<c00404c0>] (tasklet
_action + 0x88/0xdc)
r6;0000000a r5;c02ce2a4 r4;00000000
[<c0040438>] (tasklet_action + 0x0/0xdc) from [<c0040000>] (__do_softirq + 0x5c/0xc8)
r5;c02ce2e4 r4;00000001

```



```

[<c003ffa4>] (__do_softirq + 0x0/0xc8) from [<c0040224>] (irq_exit + 0x44/0x4c)
r7:00000000 r6:c02d2edc r5:c028b0a8 r4:00000010
[<c00401e0>] (irq_exit + 0x0/0x4c) from [<c002404c>] (asm_do_IRQ + 0x4c/0x60)
[<c0024000>] (asm_do_IRQ + 0x0/0x60) from [<c0024a24>] (__irq_svc + 0x24/0xa0)
Exception stack(0xc0283f54 to 0xc0283f9c)
3f40;                                00000000 ffffffff f020000c
3f60; 80000013 c0025974 c0282000 c0020f28 c02dfb58 3001f48c 41129200 3001f458
3f80; c0283fa8 c0283f9c c0283f9c c00259d4 c00259e0 80000013 ffffffff
r7:c02dfb58 r6:00000001 r5:f0000000 r4:ffffffff
[<c0025974>] (default_idle + 0x0/0x78) from [<c0025a34>] (cpu_idle + 0x48/0x64)
[<c00259ec>] (cpu_idle + 0x0/0x64) from [<c02006c4>] (rest_init + 0x48/0x58)
r5:c02bc328 r4:c02d12d4
[<c020067c>] (rest_init + 0x0/0x58) from [<c0008938>] (start_kernel + 0x27c/0x2e4)
[<c00086bc>] (start_kernel + 0x0/0x2e4) from [<30008030>] (0x30008030)
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = c0004000
[00000000] * pgd = 00000000
Internal error: Oops: 17 [#1]
Modules linked in: s3c24xx_buttons leds
CPU: 0 Not tainted (2.6.22.6 #22)
PC is at dequeue_task + 0xc/0x84
LR is at deactivate_task + 0x34/0x40
pc : [<c0036b44>] lr : [<c0036ec8>] psr: 60000093
sp : c0283e14 ip : c0283e28 fp : c0283e24
r10: 004c4b18 r9 : 894f8c40 r8 : c0284ea0
r7 : c0283e74 r6 : 000000c9 r5 : c0282000 r4 : c0284ea0
r3 : 00000080 r2 : 00000080 r1 : 00000000 r0 : c0284ea0
Flags: nZCv IRQs off FIQs on Mode SVC_32 Segment kernel
Control: c000717f Table: 338c8000 DAC: 00000017
Process swapper (pid: 0, stack limit = 0xc0282258)
Stack: (0xc0283e14 to 0xc0284000)
3e00;                                c0284ea0 c0283e38 c0283e28
3e20; c0036ec8 c0036b48 004c4b18 c0283e70 c0283e3c c0200e50 c0036ea4 c0284edc
3e40; c0284fac 3b9aca00 ffff7c22 c0282000 000000c9 c0283e74 c028a4d0 41129200
3e60; 3001f458 c0283eac c0283e74 c0201dec c0200cb0 c02cee88 c02cee88 ffff7c22
3e80; c00440a8 c0284ea0 c02ce4a0 ffffffff c0282000 00000004 c02ce280 3001f48c
3ea0; c0283ebc c0283eb0 c0201e68 c0201d70 c0283ecc c0283ec0 c00442f4 c0201e54
3ec0; c0283ee8 c0283ed0 bf00208c c00442e8 00000000 c02ce2a4 0000000a c0283f00
3ee0; c0283eec c00404c0 bf002010 00000001 c02ce2e4 c0283f20 c0283f04 c0040000

```

```

3f00: c0040448 00000010 c028b0a8 c02d2edc 00000000 c0283f30 c0283f24 c0040224
3f20: c003ffb4 c0283f50 c0283f34 c002404c c00401f0 ffffffff f0000000 00000001
3f40: c02dfb58 c0283fa8 c0283f54 c0024a24 c0024010 00000000 ffffffff f020000c
3f60: 80000013 c0025974 c0282000 c0020f28 c02dfb58 3001f48c 41129200 3001f458
3f80: c0283fa8 c0283f9c c0283f9c c00259d4 c00259e0 80000013 ffffffff c0283fc0
3fa0: c0283fac c0025a34 c0025984 c02d12d4 c02bc328 c0283fd0 c0283fc4 c02006c4
3fc0: c00259fc c0283ff4 c0283fd4 c0008938 c020068c c0008324 c0020f28 c0007175
3fe0: c02bc7e4 c0285c9c 00000000 c0283ff8 30008030 c00086cc 00000000 00000000

```

Backtrace:

```

[<c0036b38>] (dequeue_task + 0x0/0x84) from [<c0036ec8>] (deactivate_task + 0x34/0x40)
r4:c0284ea0
[<c0036e94>] (deactivate_task + 0x0/0x40) from [<c0200e50>] (schedule + 0x1b0/0x750)
r4:004c4b18
[<c0200ca0>] (schedule + 0x0/0x750) from [<c0201dec>] (schedule_timeout + 0x8c/0xbc)
[<c0201d60>] (schedule_timeout + 0x0/0xbc) from [<c0201e68>] (schedule_timeout_uninter-
ruptible + 0x24/0x28)
r8:3001f48c r7:c02ce280 r6:00000004 r5:c0282000 r4:ffffffff
[<c0201e44>] (schedule_timeout_uninterruptible + 0x0/0x28) from [<c00442f4>] (msleep +
0x1c/0x28)
[<c00442d8>] (msleep + 0x0/0x28) from [<bf00208c>] (button_do_tasklet + 0x8c/0x1c8
[s3c24xx_buttons])
[<bf002000>] (button_do_tasklet + 0x0/0x1c8 [s3c24xx_buttons]) from [<c00404c0>] (tasklet
_action + 0x88/0xdc)
r6:0000000a r5:c02ce2a4 r4:00000000
[<c0040438>] (tasklet_action + 0x0/0xdc) from [<c0040000>] (__do_softirq + 0x5c/0xc8)
r5:c02ce2e4 r4:00000001
[<c003ffa4>] (__do_softirq + 0x0/0xc8) from [<c0040224>] (irq_exit + 0x44/0x4c)
r7:00000000 r6:c02d2edc r5:c028b0a8 r4:00000010
[<c00401e0>] (irq_exit + 0x0/0x4c) from [<c002404c>] (asm_do_IRQ + 0x4c/0x60)
[<c0024000>] (asm_do_IRQ + 0x0/0x60) from [<c0024a24>] (__irq_svc + 0x24/0xa0)
Exception stack(0xc0283f54 to 0xc0283f9c)
3f40: 00000000 ffffffff f020000c
3f60: 80000013 c0025974 c0282000 c0020f28 c02dfb58 3001f48c 41129200 3001f458
3f80: c0283fa8 c0283f9c c0283f9c c00259d4 c00259e0 80000013 ffffffff
r7:c02dfb58 r6:00000001 r5:f0000000 r4:ffffffff
[<c0025974>] (default_idle + 0x0/0x78) from [<c0025a34>] (cpu_idle + 0x48/0x64)
[<c00259ec>] (cpu_idle + 0x0/0x64) from [<c02006c4>] (rest_init + 0x48/0x58)
r5:c02bc328 r4:c02d12d4
[<c020067c>] (rest_init + 0x0/0x58) from [<c0008938>] (start_kernel + 0x27c/0x2e4)

```

```
[<c00086bc>] (start_kernel + 0x0/0x2e4) from [<30008030>] (0x30008030)
Code: c02bcffc e1a0c00d e92dd810 e24cb004 (e5913000)
Kernel panic - not syncing: Fatal exception in interrupt
```

10.4.4 tasklet 与 workqueue 的区别和不同应用环境总结

(1) tasklet 与 workqueue 的区别如表 10-1 所列。

表 10-1 tasklet 与 workqueue 的区别

| Tasklet | Workqueue |
|-----------------------------|--------------------------------------|
| 处于 atomic context, 不能 sleep | 不处于 atomic context, 可以 sleep |
| 处于中断上下文, OS 不可以进行进程调度 | 处于进程上下文, OS 可以进行进程调度 |
| 运行在调度它们的同一个 CPU 上 | 默认运行在调度它们的同一个 CPU 上 |
| 不能指定确定时间进行调度 | 不能指定确定时间进行调度或指定至少延迟一个确定的时间后调度 |
| 只能提交给 ksoftirqd/0 | 可以提交给 events/0, 也可以提交给自定义的 workqueue |
| tasklet 函数带参数 | work 函数不带参数 |

(2) tasklet 与 workqueue 的不同应用环境总结如下:

① 必须立即进行紧急处理的极少量任务放在顶半部中。此时屏蔽了与自己同类型的中断, 由于任务量极少, 所以可以迅速不受打扰地处理完紧急任务。

② 需要较少时间处理的中等数量的急迫任务放在 tasklet 中。此时不会屏蔽任何中断 (包括与自己的顶半部同类型的中断), 所以不影响顶半部对紧急任务的处理; 同时又不会进行用户进程调度, 从而保证了自己的急迫任务得以迅速完成。

③ 需要较多时间处理且并不急迫 (允许被操作系统剥夺运行权) 的大量任务放在 workqueue 中。此时操作系统会尽量快速处理完这个任务, 但如果任务量太大, 期间操作系统也有机会调度别的用户进程运行, 从而保证了不会因为这个任务需要运行的时间太长而将其其他用户进程无法进行。

④ 可能引起睡眠的任务放在 workqueue 中。因为在 workqueue 中睡眠是安全的。



10.5 Linux 中断处理系统的架构与共享中断

10.5.1 裸机程序中的中断编程与有操作系统下的中断编程的区别

(1) 当硬件发生中断时, CPU 硬件会自动执行 4 个操作, 从而无条件跳转到异常向量处 (参见 ARM 体系结构与编程中关于异常和中断的内容)。

(2) 异常向量处放置了一条跳转指令, 跳转到中断异常处理程序。编写裸机程序需要想办法将这条指令放置到异常向量处, 而操作系统已经帮我们做了。

(3) 编写裸机程序当然需要编写中断异常处理程序去完成诸如环境保护 (CPU 寄存器入栈)、获取中断号、中断异常结束返回之类的操作, 而操作系统已经帮我们全部都做了, 然后调用中断处理程序 (也就是通常所说的 ISR)。

(4) 不同硬件的中断处理程序当然肯定不同, 所以要求操作系统把这个也帮你写了, 实在是太过分了。所以, 编写 ISR 就是驱动编写者的责任了! 当然, 裸机程序的编写者肯定也要编写 ISR。

(5) 编写 ISR, 对于裸机程序编写者而言百无禁忌, 想怎么搞就怎么搞。但对于有操作系统的情况下则要受到一些限制, 例如: 不能睡眠, 要用虚地址访问硬件等。

(6) 特别说明: 操作系统在启动过程中会初始化中断子系统, 这会在操作系统内部将所有中断的逻辑上的中断号 (整数) 与物理的中断线进行关联。因此当驱动调用 `request_irq` 告知操作系统将中断号与中断处理程序进行关联时, 操作系统就能够将物理的中断线与中断处理程序进行关联, 从而当某个物理中断产生时, 操作系统有能力知道应该回调哪一个中断处理程序。

10.5.2 Linux 中断处理系统的架构

详情参阅《嵌入式 linux 应用开发完全手册》P401~P403:

3 种结构: `irq_desc`、`irq_chip`、`irqaction`。

发生中断时, CPU 执行中断异常向量 `vector_irq` 的代码。

`vector_irq` 最终会调用中断处理的总入口函数 `asm_do_IRQ`。

`asm_do_IRQ` 会以中断号为下标来调用 `irq_desc` 数组中对应元素中的 `handle_irq` 函数。

`handle_irq` 会使用 `chip` 成员 (指向 `irq_chip` 结构体) 中的函数来设置硬件, 例如清除中断等。

`handle_irq` 逐个调用 `action` (`irq_desc` 数组元素的一个字段) 链表中注册的处理函数 (位于 `irqaction` 结构体的 1 个函数指针字段)。

中断处理系统的初始化就是构造 `irq_desc` 数组元素中的 `handle_irq`、`chip` 等成员。

用户注册中断就是构造 action 链表;释放中断就是从 action 链表中去除不需要的结点。

10.5.3 关于共享中断的说明

(1) 共享同一中断号的 ISR,通过 request_irq 中的同一个中断号注册到系统中的同一个中断链表(action 链表)中,而 dev_id(request_irq 的第 5 个参数)则用于区分该中断链表中不同的 ISR,因此注册时 dev_id 不能相同,也不能为 NULL。

(2) 共享中断的所有注册,其第 3 个参数都必须指明 IRQF_SHARED,以表示是注册共享中断。

(3) 中断发生时,系统会顺次调用中断链表中所有的 ISR。因此 ISR 一定要根据自己是否真的发生了中断,来返回 IRQ_HANDLED 或 IRQ_NONE。

(4) 释放中断 free_irq 的第二个参数 dev_id,正是用来查找中断链表中要被释放的中断。

10.5.4 共享中断实例

dummyisr.c 如下:

```
11 static irqreturn_t dummy_isr(int irq, void * dev_id)
12 {
13     static int haveint = 0;
14     if (haveint == 0) {
15         printk(KERN_NOTICE "dummy_isr will return IRQ_NONE");
16         haveint = 1;
17         return IRQ_RETVAL(IRQ_NONE);
18     } else {
19         printk(KERN_NOTICE "dummy_isr will return IRQ_HANDLED");
20         haveint = 0;
21         return IRQ_RETVAL(IRQ_HANDLED);
22     }
23 }
24
25 struct cdev dummy_cdev;
26 static int __init my_init(void)
27 {
28     int err;
29     err = request_irq(IRQ_EINT8, dummy_isr, IRQF_TRIGGER_FALLING | IRQF_SHARED, "dummy_isr",
30 (void *)&dummy_cdev);
31     if (err) {
32         printk(KERN_WARNING "request_irq IRQ_EINT8 for dummy_isr failed, error number is %d
33 \n", err);
```

```

32     } else {
33         printk(KERN_NOTICE "register dummy_isr succeed\n");
34     }
35     return 0;
36 }
37 static void __exit my_fini(void)
38 {
39     free_irq(IRQ_EINT8, (void *)&dummy_cdev);
40     printk(KERN_NOTICE "unregister dummy_isr succeed\n");
41 }

```

s3c24xx_buttons_v2.5.c 如下:

```

63     {IRQ_EINT8, IRQF_TRIGGER_FALLING | IRQF_SHARED, "KEY1"}, /* K1 */

175     for (i = 0; i < BUTTON_NUM; i++) {
177         err = request_irq(buttons_dev.button_irqs[i].irqno, buttons_interrupt,
178             buttons_dev.button_irqs[i].flags, buttons_dev.button_irqs[i].name,
179             (void *)&buttons_dev);

```

s3c24xx_buttons_v2.5.c 与 dummy_isr.c 注册了共享中断 IRQ_EINT8,且均遵循了“关于共享中断的说明”中的 1、2 两点。

当按下 K1 键时,OS 会逐个调用这两个中断处理函数:

```

# insmod s3c24xx_buttons.ko
buttons initialized
# ./button_test &
# open success
按 Enter 键
# insmod dummy_isr.ko
register dummy_isr succeed

```

按 K1 键

```

# dummy_isr will return IRQ_NONE
read buttons successfully, begin print the result:
K1 has been pressed 1 times!

```



第 11 章

Linux 网络设备驱动开发实战

11.1 网络设备驱动基础

11.1.1 体验网卡驱动

(1) 虚拟网卡驱动源码位于光盘\work\studydriver\snull 目录,执行 make 得到 snull.ko,加载驱动 `sudo insmod snull.ko`。

(2) 分别配置 2 张网卡(sn0 和 sn1)的 ip 地址:

```
/work/studydriver/snull$ sudo ifconfig sn0 192.168.140.1
/work/studydriver/snull$ sudo ifconfig sn1 192.168.141.2
```

(3) 测试 2 张网卡之间的通信:

```
/work/studydriver/snull$ ping 192.168.140.2
PING 192.168.140.2 (192.168.140.2) 56(84) bytes of data.
64 bytes from 192.168.140.2: icmp_seq=1 ttl=64 time=0.167 ms
64 bytes from 192.168.140.2: icmp_seq=2 ttl=64 time=0.112 ms
64 bytes from 192.168.140.2: icmp_seq=3 ttl=64 time=0.111 ms
64 bytes from 192.168.140.2: icmp_seq=4 ttl=64 time=0.139 ms
- - - 192.168.140.2 ping statistics - - -
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.111/0.132/0.167/0.024 ms
```

注:也许你感觉(2)中 ip 地址的分配和(3)中的 ping 的目标有些奇怪,甚至有些难以理解。这是由于我们要测试网卡 sn0 是否能 ping 通网卡 sn1,而 sn0 和 sn1 是本机的两张网卡,如果将它们 ip 地址配在同一网段,则测试时数据包根本不会外送。为解决测试的问题,snull 驱动对外发数据包作了一些额外处理,将目标 ip 的第 3 段加 1(即:将目标 ip 从 192.168.140.2 改为 192.168.141.2),将源 ip 的第 3 段加 1(即:将源 ip 从 192.168.140.1 改为 192.168.141.1)。详情请参见《Linux Device Driver》17.1 节。

11.1.2 网卡驱动的基本知识——2 个结构体和 5 个函数

1. 结构体 struct net_device

网络设备的注册方式与字符和块设备不同。网络设备没有主次设备号,驱动为每个刚刚探测到的网络设备(或称为接口)在一个全局的网络设备链表里插入一个节点,该节点就代表一个网络设备,它是 struct net_device 类型的结构体。

(1) net_device 结构体的内容(字段)很多,主要用于操作系统和驱动程序操控和查询网卡:

- 全局信息(例如:接口名称 eth0)
- 硬件信息(例如:中断号、I/O base)
- 接口信息(例如 MAC 地址、混杂模式标志)。
- 设备功能函数指针(例如数据发送函数)。
- 其他字段(例如 priv、自旋锁)。

(2) net_device 结构体的生成与初始化:

```
struct net_device * alloc_netdev(int sizeof_priv, const char * name, void (* setup)
(struct net_device *))
```

① sizeof_priv 是驱动的“私有数据”区的大小。

■ 对于网络驱动,这个区域是同 net_device 结构体一起分配的(逻辑上可认为“私有数据”区紧随 net_device 结构体之后);

■ net_device 结构体中有一个成员 priv,它就是指向“私有数据”区的指针。它的角色近似于我们用在字符驱动上的 private_data 指针,当一个驱动需要存取私有数据指针时,应当使用 netdev_priv 函数,例如:struct snull_priv * priv = netdev_priv(dev);

② name 是网络设备的名字,例如 eth0、eth1、sn0、sn1。

③ setup 是一个初始化函数的指针,被 alloc_netdev 调用以初始化 net_device 结构的大部分字段。网络子系统针对 alloc_netdev 为各种接口提供了一些封装函数,如:

■ 以太网设备:alloc_etherdev

◆ struct net_device * alloc_etherdev(int sizeof_priv)这个函数分配一个网络设备结构体,使用 eth%d 作为参数 name。它提供了自己的初始化函数(ether_setup)来设置许多 net_device 成员,使用对以太网设备合适的值。没有驱动提供的初始化函数给 alloc_etherdev。

◆ 通过 ether_setup 函数(由 alloc_etherdev 调用),内核负责了一些以太网范围中的缺省值。

■ 光纤通道设备:alloc_fcdev;

■ FDDI 设备: `alloc_fddidev`;

■ 令牌环设备: `alloc_trdev`。

(3) `net_device` 结构体的注册(插入全局的网络设备链表)。在对 `net_device` 结构体完成初始化后,传递这个结构体给 `int register_netdev(struct net_device * dev)`,以完成注册。

(4) 将 `net_device` 结构体从内核中注销:

`void unregister_netdev(struct net_device * dev)`。

(5) `net_device` 结构体的销毁:

`void free_netdev(struct net_device * dev)` 归还 `net_device` 结构给内核。

2. 网络接口的打开与关闭

当用户执行 `ifconfig` 命令时,内核会调用驱动的 `open` 功能函数打开或者调用驱动的 `stop` 功能函数关闭一个接口。

(1) 打开接口使用命令: `ifconfig eth0 192.168.10.1 up`。OS 会执行如下操作:

① 通过 `ioctl(SIOCSIFADDR)` (Socket I/O Control Set Interface Address) 来安排 ip 地址——内核实现。

② 调用 `dev->open` 方法——驱动实现。

③ 通过 `ioctl(SIOCSIFFLAGS)` (Socket I/O Control Set Interface Flags) 来设置 `dev->flag` 的 `IFF_UP` 位——内核实现。

(2) 接口关闭使用命令: `ifconfig eth0 down`。OS 会执行如下操作:

① 通过 `ioctl(SIOCSIFFLAGS)` 来清除 `dev->flag` 的 `IFF_UP` 位——内核实现。

② 调用 `dev->stop` 方法——驱动实现。

(3) 驱动的 `open` 方法,一般会执行如下操作:

① 把硬件(MAC)地址从硬件设备复制到 `dev->dev_addr`。

② 注册中断。

③ 启动接口传输队列 `void netif_start_queue(struct net_device * dev)`。

(4) 以下是 `snul` 驱动的 `open` 功能函数的示例:

```
int snul_open(struct net_device * dev) {
    memcpy(dev->dev_addr, "\0SNULO", ETH_ALEN);
    netif_start_queue(dev);
}
```

(5) 驱动的 `stop` 方法,一般会执行如下操作:

① 注销中断。

② 停止接口传输队列 `void netif_stop_queue(struct net_device * dev)`;

(6) 以下是 `snul` 驱动的 `stop` 功能函数的示例:

新华书店
PDG

```
int snull_release(struct net_device * dev) {  
    netif_stop_queue(dev);  
}
```

3. 数据包发送

(1) 无论何时内核需要传送一个数据包,它调用驱动的 `hard_start_xmit` 方法将数据放在外发队列上,成功时返回 0,不成功返回非 0。

(2) 每个内核处理的数据包都包含在一个 socket 缓存结构(结构 `sk_buff`)`skb` 里:

- ① `skb->data` 指向要传送的数据包。
- ② `skb->len` 指向要传送的数据包的长度。

(3) 控制发送并发。

`hard_start_xmit` 函数由一个 `net_device` 结构中的自旋锁(`xmit_lock`)来保护以避免并发调用,这样在 `hard_start_xmit` 函数未返回前不会出现对它的并发调用。

`hard_start_xmit` 函数一返回,它有可能被再次调用。当软件完成指示硬件发送数据包后,该函数返回,但此时硬件传送可能还没有完成。因此驱动需要告知协议栈不要再启动发送,直到硬件准备好接收新的数据。

① 该告知是驱动通过调用 `netif_stop_queue` 来实现的。

② 在中断处理程序中调用 `void netif_wake_queue(struct net_device * dev)` 告知协议栈可再次发送数据。

③ 如果从其他地方停止数据包的传送,不是 `hard_start_xmit` 函数,则使用的函数是:`void netif_tx_disable(struct net_device * dev)`;这个函数类似 `netif_stop_queue`,但是它还保证,当它返回时,`hard_start_xmit` 方法没有在另一个 CPU 上运行。队列能够用 `netif_wake_queue` 重启。

(4) 发送超时解决办法。设置定时器来处理这个问题。不过,网络驱动不需要自己去检测超时。它只需要设置:

① 一个超时值(设置在 `net_device` 结构的 `watchdog_timeo` 成员,以 jiffies 计)。

② 最后一个数据包的发送时间(设置在 `net_device` 结构的 `trans_start` 成员,以 jiffies 计)。

如果发送超时,协议栈的网络层最终会发现,并调用驱动的 `tx_timeout` 方法。(这个方法所做的工作是):

① 将引起发送超时的故障排除。

② 并且保证任何已经开始的发送正确地完成。特别地,驱动没有丢失追踪任何协议栈委托给它的 socket 缓存(即:需要发送的数据)。

③ 通知操作系统可以继续提交数据给网卡驱动程序的发送函数(重新启动发送队列)。

4. 中断处理

(1) 硬件芯片(例如网卡芯片)可能因为 3 种情况而触发中断:

- ① 硬件将一个外发数据包发送完成。
- ② 一个新数据包到达硬件。
- ③ 网络接口也能够产生中断来指示错误,例如状态改变。

(2) 驱动的中断处理程序可以通过检查硬件芯片中的状态寄存器,能够得知是 3 种触发中断中的哪一种情况,然后:

- ① 通知协议栈可重新启动发送队列 `netif_wake_queue(dev)`;
- ② 调用接收函数。
- ③ 处理错误。

5. 数据包接收

数据包接收有两种模式:中断驱动和轮询。这里只介绍中断驱动,在这种模式下,数据包接收函数是由中断处理程序来调用的。接收函数完成功能的流程如下:

(1) 分配一个缓存区来保存数据包:

① 缓存分配函数 (`dev_alloc_skb`) 需要知道数据长度,函数用这些信息来给缓存区分配空间。该信息来源于网卡的硬件寄存器。

② `dev_alloc_skb` 使用 `atomic` 优先级调用 `kmalloc`,因此它可以在中断时间安全使用。

(2) 一旦有一个有效的 `skb` 指针,通过调用 `memcpy`,报文数据被复制到缓存区。

● `memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen)`;

(3) `skb` 的 `dev`、`protocol`、`pkt_type` 成员必须在缓存向上传递前赋值,使协议栈知道数据包的一些信息。以太网支持代码输出一个辅助函数 `eth_type_trans(skb, dev)` 来完成这个工作:

① 通过 `skb->mac.raw` 的协议类型,发现一个合适的 `protocol` 值作为该函数的返回值。

② 通过 `skb->mac.raw` 的目标 `mac` 与 `dev` 中的本机 `mac` 地址比较,来赋值给 `skb->pkt_type`。

■ `PACKET_HOST`,是发给本机的包。

■ `PACKET_OTHERHOST`,不是发给本机的包。

■ `PACKET_BROADCAST`,广播包。

■ `PACKET_MULTICAST`,多播包。

③ 去掉 `mac` 头。

(4) 指出 IP 校验和要如何进行,即:对 `skb->ip_summed` 赋值。其取值有 3 种:

① `CHECKSUM_HW`,硬件已经进行校验。

② `CHECKSUM_NONE`,未进行校验,需要协议栈进行校验。

③ CHECKSUM_UNNECESSARY, 不必进行校验(loop 接口即是如此)。

(5) 驱动更新它的统计计数(存放于私有数据区)来记录收到一个报文。统计结构由几个成员组成, 最重要的是:

- ① rx_packet, 表示收到的报文数目。
- ② rx_bytes, 表示收到的字节总数。
- ③ tx_bytes, 表示发送的字节总数。

(6) 调用 `int netif_rx(struct sk_buff * skb)` 执行最后的接受数据包工作, 它递交 socket 缓存给 TCP/IP 协议上层。netif_rx 返回一个整数:

NET_RX_SUCCESS(0) 意思是报文成功接收。

任何其他值指示错误:

① NET_RX_CN_LOW, NET_RX_CN_MOD 和 NET_RX_CN_HIGH 指出网络子系统的递增的拥塞级别。

② NET_RX_DROP 意思是报文被丢弃。

6. 结构体 struct sk_buff 及相关内核 API

(1) 结构体 struct sk_buff 的组成如图 11-1 所示。

```
struct sk_buff {
    union { /* ... */ } h;
    union { /* ... */ } nh;
    union { /* ... */ } mac;
    uchar * head;
    uchar * data;
    uchar * tail;
    uchar * end;
    uint len; // 数据包长度
    uint truesize;
    uchar ip_summed;
    uchar pkt_type;
    ushort protocol
} * skb;
```

可用缓存空间是 `skb->end - skb->head`, 有效数据(即数据包)的空间是 `skb->tail - skb->data`。

truesize: 表示缓存区的整体长度, 置为 `sizeof(struct sk_buff)` 加上传入 `alloc_skb()` 函数的长度(或 `dev_alloc_skb` 分配的数据缓存区长度)。

(2) 分配 Socket 缓存的 API(对应的释放 API 用 `kfree` 替换 `alloc` 即可)。

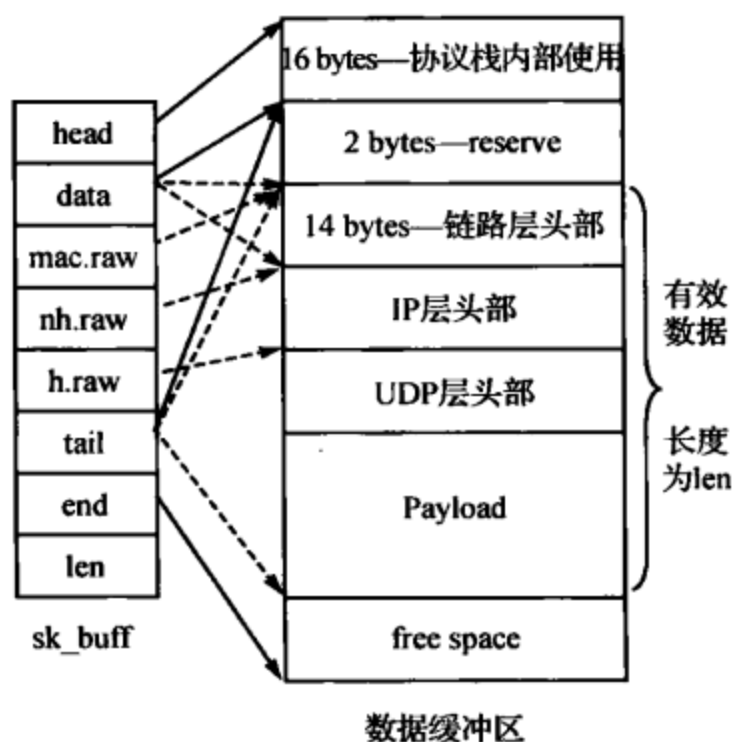


图 11-1 结构体 struct sk_buff 组成示意图

① `struct sk_buff * alloc_skb(unsigned int len, int priority)`

分配一个缓存区。`alloc_skb` 函数分配一个缓存并且将 `skb->data` 和 `skb->tail` 都初始化为 `skb->head`。

② `struct sk_buff * dev_alloc_skb(unsigned int len)`

`dev_alloc_skb` 函数是使用 `GFP_ATOMIC` 优先级调用 `alloc_skb` 的快捷方法，并且在 `skb->head` 和 `skb->data` 之间保留了一些空间。这个数据空间用在网络层之间的优化，驱动不要动它。

(3) 操纵 Socket 缓存的 API:

```
unsigned char * skb_put(struct sk_buff * skb, int len);
```

```
unsigned char * __skb_put(struct sk_buff * skb, int len);
```

更新 `sk_buff` 结构中的 `tail` 和 `len` 成员；它们用来增加数据到缓存的结尾，每个函数的返回值是 `skb->tail` 的前一个值（换句话说，它指向刚刚创建的数据空间）。两个函数的区别在于 `skb_put` 检查以确认数据适合缓存，而 `__skb_put` 省略这个检查：

```
unsigned char * skb_push(struct sk_buff * skb, int len);
```

```
unsigned char * __skb_push(struct sk_buff * skb, int len);
```

递减 `skb->data` 和递增 `skb->len` 的函数与 `skb_put` 相似，除了数据是添加到报文的开始而不是结尾。返回值指向刚刚创建的数据空间。这些函数用来在发送报文之前添加一个硬件头部。又一次，`__skb_push` 不同在它不检查空间是否足够：

```
void skb_reserve(struct sk_buff * skb, int len);
```

将 `data` 和 `tail` 的值加上 `len`。

11.1.3 虚拟网卡 snull 驱动代码分析

1. 设备生成与注册

```

688 int snull_init_module(void)
689 {
694     /* Allocate the devices */
695     snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
696                                 snull_init);
697     snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d",
698                                 snull_init);
703     for (i = 0; i < 2; i++)
704         if ((result = register_netdev(snull_devs[i])))
705             printk("snull: error %i registering device \"%s\"\n",
706                   result, snull_devs[i] -> name);
707     else
708         ret = 0;
714 }

```

在模块初始化函数中,695、697 行初始化 net_device 结构体。struct snull_priv 结构体用来存放一些私有数据,例如统计数据。函数 snull_init 则设置 net_device 的各个重要字段。704 行将代表网卡的 net_device 结构体注册进操作系统。至此,网卡就存在于系统中,可以被使用了。

模块的卸载函数完成相反的功能——将 net_device 结构体从内核中注销,以及销毁 net_device 结构体。

```

617 void snull_init(struct net_device *dev)
618 {
634     dev->open                = snull_open;
635     dev->stop                 = snull_release;
636     dev->set_config           = snull_config;
637     dev->hard_start_xmit      = snull_tx;
638     dev->do_ioctl             = snull_ioctl;
639     dev->get_stats             = snull_stats;
640     dev->change_mtu           = snull_change_mtu;
641     dev->rebuild_header       = snull_rebuild_header;
642     dev->hard_header          = snull_header;
643     dev->tx_timeout           = snull_tx_timeout;
644     dev->watchdog_timeo       = timeout; //timeout 为 5 个 jiffies

```

资源解密

PDG

663 }

特别说明:在较高版本的 Linux 内核中,会在 struct net_device 结构体中新增两个结构体字段——netdev_ops 和 header_ops,用于将上述的 open、stop 等功能函数移到这两个字段中,具体情况请参阅光盘提供的 snull 驱动源码文件。

snull_init 函数将 net_device 结构体的重要字段初始化以实现驱动中的各个功能函数,因此:

- ① 当 ifconfig sn0 up 打开网卡时,snull_open 将被调用。
- ② 当 ifconfig sn1 down 关闭网卡时,snull_release 将被调用。
- ③ 当 ifconfig 查看网卡统计数据时,snull_stats 将被调用。
- ④ 当传送数据包超过了 5 个 jiffies 仍然没有成功时,snull_tx_timeout 将被调用进行善后处理。
- ⑤ 当协议栈需要发送数据时,snull_tx 将被调用。

2. 设备打开、关闭与查询

snull_open 完成:填写 struct net_device 结构体的 dev_addr 字段(mac 地址);启动传输队列 netif_start_queue(dev)。

snull_release 完成:停止传输队列 netif_stop_queue(dev)。

snull_stats 完成:将网卡的统计数据返回给操作系统。之后操作系统将该信息返回给应用程序(例如:ifconfig)。

```

78 struct snull_priv {
79     struct net_device_stats stats;
86     struct sk_buff * skb;
87     spinlock_t lock;
88 };

557 struct net_device_stats * snull_stats(struct net_device * dev)
558 {
559     struct snull_priv * priv = netdev_priv(dev);
560     return &priv->stats;
561 }
```

3. 数据的发送

需要发送数据时,协议栈会调用 dev->hard_start_xmit。传入的第一个参数是 socket 缓存结构体(通过它可以找到需要发送的数据内容、长度等相关信息)的指针;第二个参数是对应于该网卡的 net_device 结构体。

509 行获得实际要发送的数据,510 行获得要发送数据的长度。

517 行在 `net_device` 结构体中设置发送开始时间,以便让协议栈能检测发送超时。

协议栈一旦调用 `dev->hard_start_xmit` 将 `skb` 递交给驱动,就不会在协议栈中再保留该 `skb`。因此 520 行将 `skb` 暂存起来,以便将来数据被硬件发送成功后能在中断处理程序中释放该 `skb`(不在发送函数中释放 `skb`,是因为此时不能保证硬件能发送数据成功)。

523 行将要发送的数据提交给硬件。

由于本程序驱动的是虚拟网卡,所以不会出现硬件发送数据不成功的情况。在真实的情况下,这种状况是有可能发生的,因此真实网卡驱动程序还会在 523 行调用 `netif_stop_queue` 通知协议栈暂停发送数据,直到硬件发送成功产生中断、在中断处理程序中调用 `netif_wake_queue` 通知协议栈重新启动发送。

525 行返回 0,表示发送成功。若不为零则表示失败。

注:511~516 行是预防协议栈下发的数据包长度未能达到以太网最小数据包的长度(`#define ETH_ZLEN 60`)。

```
501 int snull_tx(struct sk_buff * skb, struct net_device * dev)
502 {
503     int len;
504     char * data, shortpkt[ETH_ZLEN];
505     struct snull_priv * priv = netdev_priv(dev);
509     data = skb->data;
510     len = skb->len;
511     if (len < ETH_ZLEN) {
512         memset(shortpkt, 0, ETH_ZLEN);
513         memcpy(shortpkt, skb->data, skb->len);
514         len = ETH_ZLEN;
515         data = shortpkt;
516     }
517     dev->trans_start = jiffies; /* save the timestamp */
519     /* Remember the skb, so we can free it at interrupt time */
520     priv->skb = skb;
522     /* actual deliver of data is device-specific, and not shown here */
523     snull_hw_tx(data, len, dev);
525     return 0; /* Our simple device can not fail */
526 }
```

如果协议栈检测到发送超时(如果协议栈被 `netif_stop_queue(dev)` 通知暂停发送后,超过 `dev->watchdog_timeo` 个 `jiffies` 仍然没有被 `netif_wake_queue(dev)` 通知恢复传送,则协议栈认为在网卡 `dev` 上发生了发送超时),将会调用 `dev->tx_timeout`。

● 533、540 行更新网卡统计信息。

● 541 行恢复协议栈可以继续向网卡驱动提交数据。

● 注：如果你要追求完美的话，应该在 541 行之前添加代码调用 `snull_tx` 完成重发操作，并在重发前获得 `dev->xmit_lock` 自旋锁，重发后释放自旋锁，以避免与协议栈调用 `dev->hard_start_xmit` 产生竞态。

```
531 void snull_tx_timeout (struct net_device * dev)
532 {
533     struct snull_priv * priv = netdev_priv(dev);
540     priv->stats.tx_errors++;
541     netif_wake_queue(dev);
543 }
```

4. 中断处理程序

中断处理程序的注册一般放在驱动的 `open` 功能函数中。由于本程序驱动的是虚拟网卡，没有物理中断，所以就没有注册中断的代码。注：本程序使用定时器来模拟中断。

332~336 行说明，对于真实硬件而言，应该首先检查是否发生的中断。

由于数据包接收程序需要网卡对应的 `net_device` 结构体中的一些字段（例如：`dev`、`dev_addr`，具体参见接收程序），所以 337 行去获得它。当然 337 行要达到目的，要求我们在注册中断时要用网卡对应的 `net_device` 结构体指针作为 `request_irq` 的第 5 个参数（这一点不难做到，因为 `open` 被调用时，`net_device` 结构体指针是传入参数）。

348 行表明对于真实设备，此时应该读取硬件获得中断的原因（硬件接收完成？硬件发送完成？硬件错误？），本程序用 349 行模拟。

351~358 行处理接收中断。注：如果是真实硬件，只需要简单的调用 `snull_rx(dev)` 即可。

359~364 行处理发送中断。361~362 更新统计信息；由于硬件已经成功发送数据包，所以 363 行可以放心大胆地释放 `skb` 的空间（发送函数曾暂存该 `skb`）。

```
327 static void snull_regular_interrupt(int irq, void * dev_id, struct pt_regs * regs)
328 {
329     int statusword;
330     struct snull_priv * priv;
331     struct snull_packet * pkt = NULL;
332     /*
333      * As usual, check the "device" pointer to be sure it is
334      * really interrupting.
335      * Then assign "struct device * dev"
336      */
337     struct net_device * dev = (struct net_device *)dev_id;
348     /* retrieve statusword; real netdevices use I/O instructions */
```

```

349     statusword = priv->status;
350     priv->status = 0;
351     if (statusword & SNULL_RX_INTR) {
352         /* send it to snull_rx for handling */
353         pkt = priv->rx_queue;
354         if (pkt) {
355             priv->rx_queue = pkt->next;
356             snull_rx(dev, pkt);
357         }
358     }
359     if (statusword & SNULL_TX_INTR) {
360         /* a transmission is over; free the skb */
361         priv->stats.tx_packets++;
362         priv->stats.tx_bytes += priv->tx_packetlen;
363         dev_kfree_skb(priv->skb);
364     }
368     if (pkt) snull_release_buffer(pkt); /* Do this outside the lock! */
370 }

```

5. 数据的接收

当硬件成功接收数据后会产生中断,中断处理程序在判明是接收中断后会调用数据包接收程序,并将 struct net_device 结构体的指针作为参数传递:

258 行分配 skb 空间准备接收数据,+2 是为 265 行作准备。

259~263 行,发现如果不能分配内存则将该数据包丢弃,并更新统计信息。

265 行将数据包在 skb 中的存放位置下移 2 字节,目的是为了让数据包中的 ip 头开始在 16 字节对齐的内存位置上,以方便将来协议栈对数据包的处理。

266 行将接收到的数据拷贝到 skb 中(同时完成 skb 内部数据的更新,这由 skb_put 完成)。注意:对于真实硬件而言,266 行将被扩展为对硬件的物理操作代码(需要从硬件获得数据的长度,读取硬件寄存器以获得数据包的内容)。

269~271 行设置 skb 的一些字段,以使将来协议栈能获得数据包的一些基础信息。269 行设置网卡对应的 net_device 结构体;270 行设置包的目的类型(skb->pkt_type)和协议类型(skb->protocol,作为 eth_type_trans 的返回值);271 行表示协议栈的网络层不需要检查 ip 包的校验和(注意:loop 接口和虚拟网卡可以这样,但物理网卡必须要检查)。

272~273 更新统计信息。

274 行将 skb 上交协议栈处理。

```

249 void snull_rx(struct net_device *dev, struct snull_packet *pkt)
250 {

```



```

251 struct sk_buff * skb;
252 struct snull_priv * priv = netdev_priv(dev);
253 /*
254  * The packet has been retrieved from the transmission
255  * medium. Build an skb around it, so upper layers can handle it
256  */
257
258 skb = dev_alloc_skb(pkt->datalen + 2);
259 if (! skb) {
260     priv->stats.rx_dropped++;
261     goto out;
262 }
263
264 skb_reserve(skb, 2); /* align IP on 16B boundary */
265 memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
266 /* Write metadata, and then pass to the receive level */
267
268 skb->dev = dev;
269 skb->protocol = eth_type_trans(skb, dev);
270 skb->ip_summed = CHECKSUM_UNNECESSARY; /* dont check it */
271
272 priv->stats.rx_packets++;
273 priv->stats.rx_bytes += pkt->datalen;
274 netif_rx(skb);
275 out:
276 return;
277 }

```

11.1.4 网卡驱动的编写主要内容总结

2 个结构体：struct net_device 和 struct sk_buff。前者记录网卡设备相关信息，后者保存接收到和要发送的数据。

5 个函数：打开、关闭、发送、接收、中断。特别地，接收函数不是由协议栈调用的，而是由中断处理程序调用的，并向协议栈提供数据。

11.2 网络设备驱动实例——cs8900

11.2.1 虚拟网卡驱动与真实网卡驱动的主要区别

通过“网络设备驱动基础”一节，我们学习了网卡驱动编写的大致框架。对于真实网卡驱动，其驱动程序架构也是类似的，最大区别在于：

(1) 定义网卡使用的资源(I/O 基址和中断号等)；在模块的加载函数中调用内核 API 注

册平台设备和平台设备驱动;在模块的卸载函数中调用内核 API 注销平台设备驱动和平台设备;将原本在模块初始化函数中实现的网卡初始化代码移动到平台设备驱动的 probe 函数中,将原本在模块卸载函数中实现的网卡卸载代码移动到平台设备驱动的 remove 函数中。

(2) 对于集成在芯片上的设备通常称为平台设备,内核提供了一个基础架构(其实就是一系列数据结构和内核 API)供驱动使用,以简化对平台设备及其驱动的操作(例如:热插拔时自动调用平台设备的驱动的 probe 和 remove 函数;调用 API 获取平台设备的 I/O 基址和中断号等资源)。网卡一般集成在开发板上,所以在模块的初始化函数中,通常会调用平台设备注册 API 将网卡(连带它使用的资源)注册为平台设备,而后调用平台设备驱动注册 API 注册与平台设备关联的平台设备驱动结构体,从而通过平台设备驱动中的 probe 函数完成网卡设备在操作系统中的注册(以前这是在模块的初始化函数中实现的)。在模块的卸载函数中,会调用注销平台设备驱动的 API(这将导致平台设备驱动的 remove 运行,在该函数中通常会有以前在模块的卸载函数中执行的任务——将网卡设备从操作系统中注销),会调用注销平台设备的 API。

(3) 在真实网卡驱动的功能函数中,会增加对硬件进行操作的代码。主要有:

① 注册、释放中断。在 open、release 中实现。

② 启用和禁用物理网卡。在 open、release 中实现。

③ 物理上初始设置或改变网卡 MAC 地址。在 open、set_mac_address 中实现。

④ 物理上启用或禁用网卡混杂模式。在 cs8900_set_receive_mode 中实现。

⑤ 向物理网卡提交发送数据。在 cs8900_start_xmit 中实现。

⑥ 从物理网卡查询接收到的数据的长度、从物理网卡读取数据。在 cs8900_receive 中实现。

⑦ 从物理网卡中获得中断类型等。在中断处理程序中实现。

(4) 要对物理网卡进行上面的物理操作,需要知道物理网卡的寄存器地址和中断号,这二者都是由网卡与 CPU 的物理连接决定的。因此驱动编写者还必须要能看懂网卡的硬件连线图,同时配合查看网卡芯片生产商给出的网卡硬件手册,才能知道物理网卡的寄存器地址和中断号是多少。

11.2.2 真实网卡驱动的整体框架分析

1. 驱动接口简介

(1) 上层接口。

① 用户接口。

● 网络设备:ethx。

● 用户空间程序。

■ ifconfig:直接调用 open、close、ioctl 接口。

■ socket 程序:不直接调用 send 和 recv 接口,而是将请求提交 OS 的网络协议栈。

② 操作系统接口。

- 网络协议栈决定发送数据时,调用驱动中的.ndo_start_xmit(skb,ndev)发送函数向硬件发送数据:

■ ndo_start_xmit 通过内核 api——netif_stop_queue (ndev)禁止网络协议栈再度调用 ndo_start_xmit。

■ ndo_start_xmit 向硬件提交数据后立即返回。

■ 硬件在完成发送后,以中断的方式通知 OS,中断服务程序通过内核 api——netif_wake_queue (ndev)通知协议栈可再度调用 ndo_start_xmit。

- 硬件接收到数据后,以中断的方式通知 OS;中断服务程序调用驱动中的接收函数,该函数在从网卡硬件接收了数据后,调用内核 api——netif_rx (skb),向网络协议栈提交数据。

(2) 下层接口(硬件接口)。

- 驱动的.ndo_start_xmit(skb, net_dev)通过写网卡的硬件数据寄存器,将 skb 中的数据发送给硬件。
- 中断服务程序通过读取网卡的硬件状态寄存器,判别是接收中断还是发送中断。
- 驱动接收函数,通过读网卡的硬件数据寄存器,将数据读入内存,存放于 kmalloc 的 skb 中。

2. 网络设备结构体 struct net_device 中的重要字段

网络设备结构体 struct net_device 在操作系统内部代表一张网卡,含有很多重要字段:

- name:网卡名称,例如 eth0
- base_addr:I/O 基地址
- irq:中断号
- dev_addr[]:MAC 地址
- if_port:接口类型
- features:接口标志
 - IFF_UP
 - IFF_PROMISC
 - IFF_MULTICAST
- trans_start:发送开始时间
- last_rx:接收结束时间
- watchdog_timeo:发送超时时间
- net_device_ops:包含各个驱动函数指针的结构体
 - open



- stop
- hard_start_xmit
- tx_timeout
- get_stats
- do_ioctl
- set_multicast_list
- set_mac_address

- priv: 私有数据。由用户定义, 通常包括以下两种:
 - struct net_device_stats stats: 记录各种统计信息。
 - spinlock_t lock

3. 数据包结构体重要字段

数据包结构体, 代表要发送或发送的一个数据包。含有很多重要字段:

- mac、nh、h: MAC、IP、TCP 头指针。
- head、end: 数据缓冲区首、尾指针。
- data、tail: 数据包有效数据首、尾指针。
- len: 数据包长度。
- protocol: 协议(ip、ipx 等)。
- pkt_type: PACKET_HOST(给我的)、PACKET_OTHERHOST(不是给我的)、PACKET_BROADCAST、PACKET_MULTICAST。

4. 网络设备注册

cs8900 驱动在探测函数 cs8900_probe (struct platform_device * pdev) 中注册网络设备。

(1) 操作系统接口。

- ndev = alloc_etherdev(sizeof(struct cs8900_priv)); //kmalloc 了含 priv 数据在内的网络设备结构体
- SET_NETDEV_DEV(ndev, &pdev->dev); //生成 sys 链接
- ndev->dev_addr[0] = 0x08;
- ndev->if_port = IF_PORT_10BASET;
- priv = netdev_priv(ndev); //获取 priv 数据指针
- priv->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
- priv->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
- priv->addr_req = request_mem_region(priv->addr_res->start, iosize, pdev->name);
- priv->io_addr = ioremap(priv->addr_res->start, iosize);

- `ndev->base_addr = (unsigned long)priv->io_addr;`
- `ndev->irq = priv->irq_res->start;`
- `ndev->netdev_ops = &cs8900_netdev_ops;` // 驱动中各个实际函数的指针
- `ndev->watchdog_timeo = HZ;`
- `register_netdev(ndev);` // 将网络设备注册进操作系统(即将结构体链入系统链表)

(2) 硬件接口

① 内存控制器设置:

- `__raw_writel(0x2211d110, S3C2410_BWSCON);`
- `__raw_writel(0x1f7c, S3C2410_BANKCON3);`

② 检测是否是 cs8900 硬件:

- `cs8900_read(ndev, PP_ProductID) != EISA_REG_CODE`
- `value = cs8900_read(ndev, PP_ProductID + 2);`
- `if (VERSION(value) != CS8900A)`

③ 中断引脚 4 选 1

④ `cs8900_write(ndev, PP_IntNum, 0);`

⑤ 在网卡硬件寄存器中写入 mac 地址

- `for (i = 0; i < ETH_ALEN; i += 2)`
`cs8900_write(ndev, PP_IA + i, ndev->dev_addr[i] | (ndev->dev_addr[i + 1]`
`<< 8));`

5. 网络设备注销

cs8900 驱动在函数 `cs8900_drv_remove(struct platform_device *pdev)` 中注销网络设备:

- (1) `struct net_device *ndev = platform_get_drvdata(pdev);`
- (2) `struct cs8900_priv *priv = netdev_priv(ndev);`
- (3) `platform_set_drvdata(pdev, NULL);`
- (4) `unregister_netdev(ndev);` // 将网络设备结构体从系统链表中摘除
- (5) `cs8900_release_priv(pdev, priv);`
- ① `iounmap(priv->io_addr);`
- ② `release_resource(priv->addr_req);`
- ③ `kfree(priv->addr_req);`
- (6) `free_netdev(ndev);` // kfree 网络设备结构体(包括 priv 结构体)

6. 平台设备原理及其在 cs8900 驱动中的使用

(1) 平台设备及其驱动的数据结构定义。

① 单个平台设备的资源(I/O 基址、中断号)定义。

```
static struct resource cs8900_resource[] = {
    [0] = {
        .start = 0x19000300,
        .end   = 0x19000310,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_EINT9,
        .end   = IRQ_EINT9,
        .flags = IORESOURCE_IRQ,
    }
};
```

② 单个平台设备定义:

```
struct platform_device net_device_cs8900 = {
    .name = "cs8900",
    .id   = -1,
    .num_resources = ARRAY_SIZE(cs8900_resource),
    .resource      = cs8900_resource,
};
```

③ 一组平台设备定义:

```
static struct platform_device *network_devices[] __initdata = {
    &net_device_cs8900,
};
```

④ 单个平台设备驱动的定义:

```
static struct platform_driver cs8900_driver = {
    .driver = {
        .name = "cs8900",
        .owner = THIS_MODULE,
    },
    .probe = cs8900_probe,
    .remove = __devexit_p(cs8900_drv_remove)
};
```

(2) 平台设备及其驱动注册

① 将一组平台设备注册到操作系统中(即:将一组平台设备中的全部设备节点链入操作系统中的平台设备链表中)。


```
platform_add_devices(network_devices, ARRAY_SIZE(network_devices));
```

② 将平台设备驱动注册到操作系统中(即:将平台驱动节点链入操作系统中的平台设备驱动链表中,如果能匹配平台设备,就回调平台设备驱动中定义的 probe 函数)。

```
platform_driver_register(&cs8900_driver)
{
    将 cs8900_driver 插入平台驱动链表;
    扫描平台驱动链表以及平台设备链表,用二者的 name 字段进行匹配;
    if (匹配成功)
        回调平台驱动中定义的 probe 函数
}
```

③ 在注册成功后,如果发生平台设备的热插拔,操作系统会自动回调匹配的平台驱动的 probe 和 remove 函数。platform_driver_unregister 的调用也会引起平台驱动的 remove 函数被执行。这样就可以在 probe 函数中实现网卡设备的注册,在 remove 函数中实现网卡设备的注销。

④ 使用平台设备。在平台设备注册成功后,就可以在任何时候(通常在平台驱动的 probe 函数中)方便的获得平台设备的信息(资源)。

```
cs8900_probe(struct platform_device *pdev)
{
    priv->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0); //priv->addr_res 将会
    等于 cs8900_resource[0],其中有 I/O 基址 0x19000300
    priv->irq_res = platform_get_resource(pdev, IORESOURCE_IRQ, 0); // priv->irq_res 将会
    等于 cs8900_resource[1],其中有中断号 IRQ_EINT9
}
```

7. 打开、关闭、数据发送、数据发送超时处理、中断处理、数据接收处理、改变接收模式、改变 mac 地址等操作在物理网卡 cs8900 上的实现代码简述

(1) 打开网卡设备 cs8900_open(struct net_device *ndev)

① 被调用时机:ifconfig up

② 硬件相关操作 /* enable the ethernet controller */参阅代码详细分析以及网卡芯片硬件手册 4.4.5~4.4.23;4.10

```
cs8900_set(ndev, PP_RxCFG, RxOKiE | BufferCRC | CRCErroriE | RuntiE | ExtradataiE);
cs8900_set(ndev, PP_RxCTL, RxOKA | IndividualA | BroadcastA);
cs8900_set(ndev, PP_TxCFG, TxOKiE | Out_of_windowiE | JabberiE);
cs8900_set(ndev, PP_BufCFG, Rdy4Tx iE | RxMissiE | TxUnderruniE | TxColOvf iE | MissOvflo iE);
cs8900_set(ndev, PP_LineCTL, SerRxON | SerTxON);
```

```
cs8900_set (ndev, PP_BusCTL, EnableRQ);
```

③ 操作系统相关:注册中断、通知协议栈可启动输出。

```
set_irq_type(ndev->irq, IRQ_TYPE_EDGE_RISING);
request_irq (ndev->irq, cs8900_interrupt, 0, ndev->name, ndev);
netif_start_queue (ndev);
```

(2) 关闭网卡设备 cs8900_close(struct net_device * ndev)

① 被调用时机:ifconfig down

② 硬件相关操作

```
cs8900_write (ndev, PP_BusCTL, 0);
cs8900_write (ndev, PP_TestCTL, 0);
cs8900_write (ndev, PP_SelfCTL, 0);
cs8900_write (ndev, PP_LineCTL, 0);
cs8900_write (ndev, PP_BufCFG, 0);
cs8900_write (ndev, PP_TxCFG, 0);
cs8900_write (ndev, PP_RxCTL, 0);
cs8900_write (ndev, PP_RxCFG, 0);
```

● 系统相关操作。

```
free_irq (ndev->irq, ndev);
netif_stop_queue (ndev);
```

(3) 数据发送 cs8900_start_xmit (struct sk_buff * skb, struct net_device * ndev)

● 被调用时机:协议栈想发送数据,且发送队列被激活。

```
netif_stop_queue (ndev); //通知协议栈不要再调用自己发送数据
//通知硬件准备发送以及要发送的数据的长度后,向硬件提交数据 4.10.8
cs8900_write (ndev, PP_TxCMD, TxStart (After5));
cs8900_write (ndev, PP_TxLength, skb->len);
status = cs8900_read (ndev, PP_BusST);
if ((status & TxBidErr)) return 1; //数据包超长,返回 1 告知协议栈传输错误
if (! (status & Rdy4TxNOW)) return 1; //硬件未能成功分配容纳数据包的 buffer
cs8900_frame_write (ndev, skb); //向硬件提交数据
ndev->trans_start = jiffies; //记录发送时间,供协议栈检测超时使用
dev_kfree_skb (skb); //已发送成功,故 kfree skb
priv->txlen = skb->len; //暂存发送数据的长度,供将来硬件发送数据成功后,中断服务程序更新统计信息

return 0; //通知协议栈发送成功,协议栈将会将 skb 从系统链表中摘除,不再重传该 skb
```

- 如果硬件最终发送成功,将在中断服务程序中 `netif_wakeup_queue`
- 如果硬件未能成功发送,或由于此处返回 1,则由于没有 `netif_wakeup_queue`,协议栈不会再提交发送数据。这将导致最终协议栈发现超时,而调用 `ndo_tx_timeout`

(4) 数据发送超时处理 `cs8900_tx_timeout (struct net_device * ndev)`

- 被调用时机:协议栈发现发送已经超时的时候。

```
struct cs8900_priv * priv = netdev_priv(ndev);
//更新统计记录
priv->stats.tx_errors++;
priv->stats.tx_heartbeat_errors++;
//通知协议栈可重新提交发送数据
netif_wake_queue(ndev);
```

(5) 中断处理 `irqreturn_t cs8900_interrupt (int irq, void * id)`

读取硬件的中断类型指示寄存器,判定中断类型。若是发送成功中断则更新统计信息后 `netif_wakeup_queue`;若是接收成功中断则调用接收函数 `cs8900_receive`。

```
while ((status = cs8900_read(ndev, PP_ISQ))) {
    switch (RegNum(status)) {
        case TxEvent:
            if (RegContent(status) & TxOK) {
                priv->stats.tx_packets++;
                priv->stats.tx_bytes += priv->txlen;
                netif_wake_queue(ndev);
                break;
            }
        case RxEvent:
            cs8900_receive(ndev);
            break;
    }
}
```

(6) 数据接收处理 `cs8900_receive (struct net_device * ndev)`

```
status = cs8900_read(ndev, PP_RxStatus);
length = cs8900_read(ndev, PP_RxLength); //从硬件获知数据长度
if (!(status & RxOK)) {
    priv->stats.rx_errors++;
    return;
}
skb = dev_alloc_skb(length + 4); //kmalloc skb
```

资源解密

PDG

```

skb->dev = ndev;
skb_reserve(skb,2); //因 mac 头长度为 14, + 2 可保证 ip 头 16 字节对齐, 方便协议栈处理 skbcs8900_
frame_read(ndev,skb,length); //从硬件读取数据, 填充 skb
skb->protocol = eth_type_trans(skb,ndev); //设置正确的 skb->mac\ pkt_type\protocol, 去掉
mac 头
netif_rx(skb); //向协议栈提交 skb

```

(7) 改变接收模式 cs8900_set_receive_mode(struct net_device * ndev)

- 被调用时机: ifconfig 改变接口标志时。
- 功能: 启用(或禁用)网卡硬件的混杂、多播模式。

```

if ((ndev->flags & IFF_PROMISC))
    cs8900_set(ndev,PP_RxCTL,PromiscuousA);
else
    cs8900_clear(ndev,PP_RxCTL,PromiscuousA);
if ((ndev->flags & IFF_ALLMULTI) && ndev->mc_list)
    cs8900_set(ndev,PP_RxCTL,MulticastA);
else
    cs8900_clear(ndev,PP_RxCTL,MulticastA);

```

(8) 获取统计信息 cs8900_get_stats(struct net_device * ndev)

- 调用时机: ipconfig 获取统计信息时。

```

struct cs8900_priv * priv = netdev_priv(ndev);
struct net_device_stats * stats = &priv->stats;
return stats;

```

(9) 改变 mac 地址 int (* set_mac_address)(struct net_device * dev, void * addr)

- 调用时机: ifconfig eth0 hw ether xx:xx:xx:xx:xx:xx
- 此方法默认被置为 eth_mac_addr
 - eth_mac_addr 只在接口处于 down 的情况才执行操作。
 - eth_mac_addr 只完成将新 mac 地址 addr 拷贝到 dev->dev_addr 中这一个操作。
 - 因此驱动如需支持改变 mac 的操作, 需要在驱动的 open 方法中将硬件的 mac 地址寄存器的值设为 dev->dev_addr 中的值(参见 open 硬件控制代码解析)。
- 若驱动想要彻底支持更改 MAC 地址的操作, 需要实现自己的 set_mac_address 方法。
 - 修改 dev->dev_addr(给 ifconfig 看以及作为 arp 的响应)。
 - 修改硬件的 MAC 地址寄存器的值(使硬件获得修改后的 MAC 地址)。
 - 修改 EEPROM 中存储的 MAC 地址(使修改永久有效)。

11.2.3 驱动中关于 cs8900 硬件操作的探讨

“真实网卡驱动的整体框架分析”中的内容大多数与“网络设备驱动基础”中的内容一致，都比较容易理解。但其中有关硬件的操作，则是与 cs8900 网卡芯片紧密相关的，要理解与硬件相关的代码，必须了解如何访问和操纵硬件，这要求全面了解网卡芯片的操控机制及各个硬件寄存器的地址与访问方式（这需要阅读网卡芯片数据手册），此外还需要看懂网卡芯片与 CPU 的硬件连线图。下面对这个问题，予以简要说明。

如图 11-2 所示，cs8900 提供两类寄存器：可以用 I/O 内存地址直接访问的外部寄存器 8 个；不能用 I/O 内存地址直接访问，而必须通过外部寄存器间接访问的内部寄存器若干。

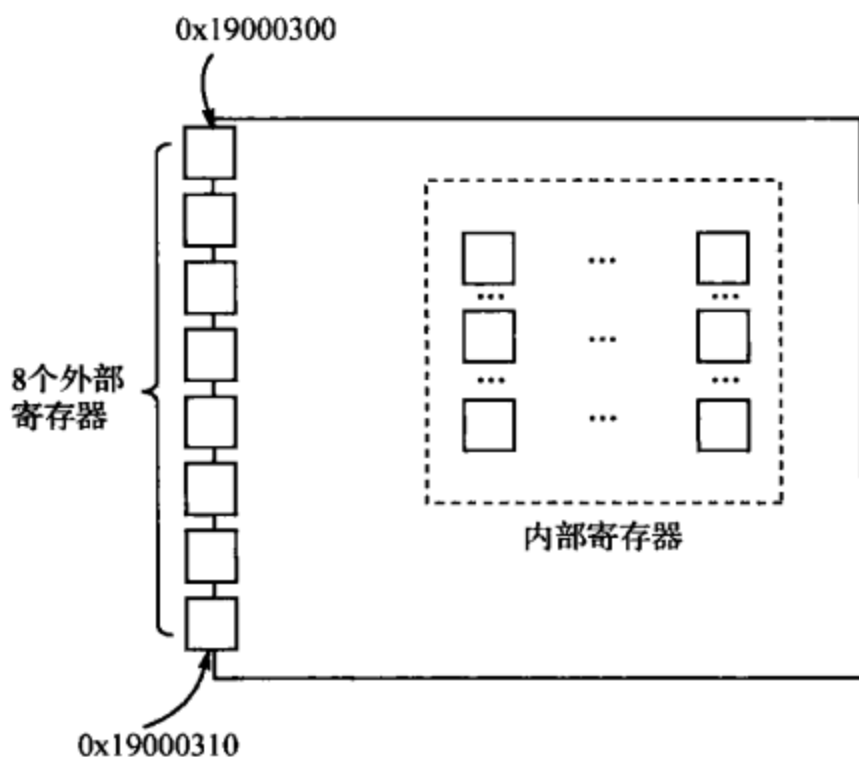


图 11-2 cs8900 的两类寄存器

1. 网卡寄存器的 I/O 内存地址范围为何是 0x19000300~0x19000310?

(1) CPU 的 nGCS3 连接 CS8900 的 nCHIPSEL 引脚（此引脚必须有效，才能选中 cs8900）。当地址为 0x18000000 时，nGCS3 有效。

(2) CPU 的 nGCS3 连接 CS8900 的 AEN 引脚。当 cs8900 启用为 io 模式时（而不是内存模式），AEN 必须为低电平。当地址为 0x18000000 时，nGCS3 为低电平。

(3) CPU 的 addr24 反相后与 LnOE (LnWE) 相或，其输出接在了 CS8900 的 nIOR (nIOW) 的引脚上。addr24 必须为 1，cs8900 的 nIOR (nIOW) 才有机会有效。

(4) cs8900 硬件已经确定了第一个 I/O 寄存器的地址偏移量为 0x300。

(5) cs8900 硬件共提供 8 个可以用 I/O 内存地址直接访问的外部寄存器（它们的地址连续，每个寄存器大小为 2 字节）。

查看 cs8900 芯片与 CPU 的硬件连线图可知 1、2、3,再加上 4,可得网卡寄存器的 I/O 内存地址的首地址是 0x19000300,再由 (5) 可知网卡寄存器的 I/O 内存地址的末地址是 0x19000310。

2. 如何访问内部寄存器

每个内部寄存器都有一个编号。要想访问某个内部寄存器,先将内部寄存器编号写入外部寄存器 PP_Address(地址:0x0a),再读取(或写入)外部寄存器 PP_Data(地址:0x0c)即可。例如:

```
120 static inline u16 cs8900_read (struct net_device * ndev,u16 reg)
121 {
122     outw (reg,ndev->base_addr + PP_Address);
123     return (inw (ndev->base_addr + PP_Data));
124 }
125
126 static inline void cs8900_write (struct net_device * ndev,u16 reg,u16 value)
127 {
128     outw (reg,ndev->base_addr + PP_Address);
129     outw (value,ndev->base_addr + PP_Data);
130 }
131
132 static inline void cs8900_set (struct net_device * ndev,u16 reg,u16 value)
133 {
134     cs8900_write (ndev,reg,cs8900_read (ndev,reg) | value);
135 }
136
137 static inline void cs8900_clear (struct net_device * ndev,u16 reg,u16 value)
138 {
139     cs8900_write (ndev,reg,cs8900_read (ndev,reg) & ~value);
140 }
```

3. 驱动的各个功能函数中操控硬件的代码解析

(1) cs8900_probe 中的硬件操控代码。

① 内部寄存器 PP_ProductID 存放了网卡芯片版本号信息,所以 547~557 行是通过该寄存器来确认芯片是否是 cs8900A。

② cs8900 芯片有 4 个中断引脚,查阅硬件连线图可知:cs8900 芯片的第一个中断引脚连接到了 CPU。559 行将 0 写入 PP_IntNum 寄存器,表示选择 cs8900 的第一个中断引脚来输出中断信号。

③ PP_IA 寄存器存放的是网卡运行时的 MAC 地址,所以 565~566 行将 net_device 中的 MAC 地址写入该寄存器。这一步必须要做,因为本开发板的 cs8900 芯片没有配置 EEPROM 来固化 MAC 地址,因此就必须在网卡被使用前,将指定的 MAC 地址写入硬件寄存器 PP_IA。

```

547     if ((value = cs8900_read (ndev,PP_ProductID)) != EISA_REG_CODE) {
548         printk (KERN_ERR "%s: incorrect signature 0x%.4x\n",ndev->name,value);
549         return -ENXIO;
550     }
554     if (VERSION (value) != CS8900A) {
555         printk (KERN_ERR "%s: unknown chip version 0x%.8x\n",ndev->name,VERSION
(value));
556         return -ENXIO;
557     }
559     cs8900_write (ndev,PP_IntNum,0);
565     for (i = 0; i < ETH_ALEN; i += 2)
566         cs8900_write (ndev,PP_IA + i,ndev->dev_addr[i] | (ndev->dev_addr[i
+ 1] << 8));

```

(2) 打开网卡设备 cs8900_open(struct net_device * ndev) 中的硬件操控代码

- 397~398 行将 MAC 地址写入 PP_IA 寄存器,这样一来就可以保证用户能用 ifconfig 改变网卡的运行时 MAC 地址(参见稍后的验证);
- 401 行设置 PP_RxCFG 寄存器(查阅 cs8900 硬件手册 P52):

■ RxOKiE When set, there is an RxOK Interrupt if a frame is received without errors. RxOK interrupt is not generated when DMA mode is used for frame reception. 正确接收数据帧则产生中断

■ BufferCRC When set, the received CRC is included with the data stored in the receive-frame buffer, and the four CRC bytes are included in the receive-frame length (PacketPage base + 0402h). When clear, neither the receive buffer nor the receive length include the CRC. 将 CRC 校验信息保留在数据帧中

■ CRCErroriE When set, there is a CRCError Interrupt if a frame is received with a bad CRC. 接收数据帧 CRC 校验错则产生中断

■ RuntiE When set, there is a Runt Interrupt if a frame is received that is shorter than 64 bytes. The CS8900A always discards any frame that is shorter than 8 bytes. 接收数据帧短于 64 字节则产生中断

■ ExtradataiE When set, there is an Extradata Interrupt if a frame is received that is longer than 1518 bytes. The operation of this bit is independent of the received packet integrity (good or bad CRC). 接收数据帧长于 1518 则产生中断

- 402 行设置 PP_RxCTL 寄存器(查阅 cs8900 硬件手册 P54),按此设置,将不会启用混杂模式
 - RxOKA When set, the CS8900A accepts frames with correct CRC and valid length (valid length is: 64 bytes \leq length \leq 1518 bytes). 要接收 CRC 正确且有正确长度的帧
 - IndividualA When set, receive frames are accepted if the Destination Address matches the Individual Address found at PacketPage base + 0158h to PacketPage base + 015Dh. 要接收发给自己的帧。
 - BroadcastA When set, receive frames are accepted if the Destination Address is FFFF FFFF FFFFh. 要接收广播帧。
- 403 行设置 PP_TxCFG 寄存器(查阅 cs8900 硬件手册 P55)
 - TxOKiE When set, an interrupt is generated if a packet is completely transmitted. 成功传送一个数据帧则产生中断。
 - Out-of-windowiE When set, an interrupt is generated if a late collision occurs (a late collision is a collision which occurs after the first 512 bit times). When this occurs, the CS8900A forces a bad CRC and terminates the transmission 发送数据帧的后部(发送 512bit 之后)时产生冲突,则产生中断(这说明发送时间窗有问题。因为如果局域网上所有机器都严格遵守以太网规范的话,如果有冲突的话,应该在前 512bit 的时间内就能检测到)。
 - JabberiE When set, an interrupt is generated if a transmission is longer than approximately 26 ms. 发送数据帧的时长超过 26ms 则产生中断。
- 404 行设置 PP_BufCFG 寄存器(查阅 cs8900 硬件手册 P58)。
 - Rdy4TxIE When set, there is an interrupt when the CS8900A is ready to accept a frame from the host for transmission. (See Section 5.7 on page 98 for a description of the transmit bid process.) 硬件可以接收驱动提交数据,就产生中断。
 - RxMissIE When set, there is an interrupt if one or more received frames is lost due to slow movement of receive data out of the receive buffer (called a receive miss). When this happens, the RxMiss bit (Register C, BufEvent, Bit A) is set. 因硬件接收缓冲区满而造成接收数据包丢失,就产生中断。
 - TxUnderrunIE When set, there is an interrupt if the CS8900A runs out of data before it reaches the end of the frame (called a transmit underrun). When this happens, event bit TXUnderrun (Register C, BufEvent, Bit 9) is set and the CS8900A makes no further attempts to transmit that frame. If the host still wants to transmit that particular frame, the host must go through the transmit request process

again. 要传输的数据帧长度小于指示的长度,就产生中断。

■ TxColOvfiE If set, there is an interrupt when the TxCOL counter increments from 1FFh to 200h. (The TxCOL counter (Register 18) is incremented whenever the CS8900A sees that the RXD+/RXD- pins (10BASE-T) or the CI+/CI- pins (AUI) go active while a packet is being transmitted.) 在发送期间有数据到达,这种情况累计发生 0x200 次,就产生中断(这说明网络很繁忙)。

■ MissOvfloiE If MissOvfloiE is set, there is an interrupt when the RxMISS counter increments from 1FFh to 200h. (A receive miss is said to have occurred if packets are lost due to slow movement of receive data out of the receive buffers. When this happens, the RxMiss bit (Register C, BufEvent, Bit A) is set, and the RxMISS counter (Register 10) is incremented.) 当 RxMissE 错误产生累计超过 0x200 次的时候,就产生中断(这说明硬件数据帧缓冲区太小,或者驱动接收数据的速度太慢)。

● 405 行设置 PP_LineCTL 寄存器(查阅 cs8900 硬件手册 P62)。

■ SerRxON When set, the receiver is enabled. When clear, no incoming packets pass through the receiver. If SerRxON is cleared while a packet is being received, reception is completed and no subsequent receive packets are allowed until SerRxON is set again. 启用接收。

■ SerTxON When set, the transmitter is enabled. When clear, no transmissions are allowed. If SerTxON is cleared while a packet is being transmitted, transmission is completed and no subsequent packets are transmitted until SerTxON is set again. 启用发送。

● 406 行设置 PP_BusCTL 寄存器(查阅 cs8900 硬件手册 P66)。

■ EnableRQ When set, the CS8900A will generate an interrupt in response to an interrupt event(Section 5.1). When cleared, the CS8900A will not generate any interrupts 启用中断。

● 412 行延时一段时间,以使网卡硬件能有时间处理完上述要完成的设置。

如果以上针对硬件的描述理解不了的话,就应该去了解一下 ISO 的 OSI 模型的数据链路层的相关知识了。

```
391 static int cs8900_open(struct net_device * ndev)
392 {
395     /* in case of ifconfig modify mac address */
397     for (i = 0; i < ETH_ALEN; i += 2)
398         cs8900_write(ndev, PP_IA + i, ndev->dev_addr[i] | (ndev->dev_addr[i + 1] << 8));
400     /* enable the ethernet controller */
```

```

401 cs8900_set (ndev,PP_RxCFG,RxOKiE | BufferCRC | CRCErroriE | RuntiE | ExtradataiE);
402 cs8900_set (ndev,PP_RxCTL,RxOKA | IndividualA | BroadcastA);
403 cs8900_set (ndev,PP_TxCFG,TxOKiE | Out_of_windowiE | JabberiE);
404 cs8900_set (ndev,PP_BufCFG,Rdy4TxIE | RxMissiE | TxUnderruniE | TxColOvfiE | MissOvfloiE);
405 cs8900_set (ndev,PP_LineCTL,SerRxON | SerTxON);
406 cs8900_set (ndev,PP_BusCTL,EnableRQ);
412 udelay(200);
423 }

```

验证用户可以用 ifconfig 改变网卡的运行时 MAC 地址如表 11-1 所列。

表 11-1 验证 ifconfig 改变网卡 MAC 地址

| 开发板 | Linux 机器 |
|---|---|
| <pre> # ifconfig eth0 Link encap:Ethernet HWaddr 08:00:3E:26:0A:5B inet addr:192.168.2.17 Bcast:192.168.2.255 Mask:255.255.255.0 # ping 192.168.2.11 PING 192.168.2.11 (192.168.2.11): 56 data bytes 64 bytes from 192.168.2.11: seq=0 ttl=64 time=3.932 ms </pre> | <pre> /\$ arp -a (192.168.2.17) at 08:00:3E:26:0A:5B [ether] on eth3 </pre> |
| <pre> # ifconfig eth0 down cs8900: close... # ifconfig eth0 hw ether 08:00:3E:26:0A:5D # ifconfig eth0 up # ifconfig eth0 Link encap:Ethernet HWaddr 08:00:3E:26:0A:5D inet addr:192.168.2.17 Bcast:192.168.2.255 Mask:255.255.255.0 # ping 192.168.2.11 PING 192.168.2.11 (192.168.2.11): 56 data bytes 64 bytes from 192.168.2.11: seq=0 ttl=64 time=4.612 ms </pre> | <pre> /\$ arp -a (192.168.2.17) at 08:00:3E:26:0A:5D [ether] on eth3 </pre> |

(3) 关闭网卡设备 `cs8900_close(struct net_device * ndev)` 中的硬件操控代码。

```
425 static int cs8900_close(struct net_device * ndev)
426 {
427     /* disable ethernet controller */
428     cs8900_write (ndev, PP_BusCTL, 0); //禁止产生中断
429     cs8900_write (ndev, PP_TestCTL, 0);
430     cs8900_write (ndev, PP_SelfCTL, 0);
431     cs8900_write (ndev, PP_LineCTL, 0); //禁止发送和接收
432     cs8900_write (ndev, PP_BufCFG, 0);
433     cs8900_write (ndev, PP_TxCFG, 0);
434     cs8900_write (ndev, PP_RxCTL, 0);
435     cs8900_write (ndev, PP_RxCFG, 0);
436 }
443 }
```

(4) 发送数据函数中的硬件操控代码。

273 行告知硬件芯片:hi,你好,我要向你提交发送数据了!

274 行告知硬件芯片:小子,我要发送的数据帧的长度为 `skb->len`

276 行读取硬件芯片的状态

278~285 行在硬件芯片拒绝了本次传送请求的情况下,放弃本次传送请求并更新统计信息

287~294 行在硬件芯片暂时未准备好接收本次传送请求的情况下(例如:暂时在硬件芯片内部找不到足够空闲空间来容纳本次请求传送的数据),放弃本次请求并更新统计信息。注:由于硬件芯片只是暂时没有空间,所以当稍后有空间了,硬件芯片将产生中断。所以如果你是追求完美的人的话,可以将 `skb` 暂存起来,然后在中断处理程序中提交这个数据帧。

296 行将要发送的数据帧提交给网卡硬件(如何操作,见稍后分析)。

```
264 static int cs8900_start_xmit (struct sk_buff * skb, struct net_device * ndev)
265 {
273     cs8900_write (ndev, PP_TxCMD, TxStart (After5));
274     cs8900_write (ndev, PP_TxLength, skb->len);
276     status = cs8900_read (ndev, PP_BusST);
278     if ((status & TxBidErr)) {
281         priv->stats.tx_errors++;
282         priv->stats.tx_aborted_errors++;
283         priv->txlen = 0;
284         return 1;
285     }
287     if (!(status & Rdy4TxNOW)) {
```

```

290     priv->stats.tx_errors++;
291     priv->txlen = 0;
292     /* FIXME: store skb and send it in interrupt handler */
293     return 1;
294 }
296     cs8900_frame_write (ndev,skb);
310 }

```

假设要发送的数据长度为 125 字节,则下面的函数 cs8900_frame_write 通过向 0x19000300 这个 I/O 内存地址连续写入 63 个半字来完成将要发送的数据(125 个有效的字节+1 个无效的字节)提交给硬件芯片。到此为止,驱动的任务就完成了。

```

147 static inline void cs8900_frame_write (struct net_device * ndev,struct sk_buff * skb)
148 {
149     outsw (ndev->base_addr,skb->data,(skb->len + 1) / 2);
150 }

```

(5) 中断处理 irqreturn_t cs8900_interrupt (int irq,void * id)中的硬件操控代码

328 行读取 PP_ISQ 寄存器以获得产生中断的原因;

若是由于接收到数据帧而产生中断,则 332—334 行调用驱动接收函数进行处理(当然这里也可能是接收到含有错误的数据帧而产生中断。不过不要紧,待会儿接收处理函数会处理出错的情况)。

若是由于发送数据帧而产生中断,则 336~349 行进行处理。

① 337 行更新统计信息,累计产生冲突的次数。

② 338~342 行对应传送不成功(时间窗口不对、传送超过 26ms),更新统计信息。这种情况下没有调用 netif_wake_queue,所以将最终导致 cs8900_tx_timeout 被协议栈调用,清除障碍的工作本来就是该 cs8900_tx_timeout 完成的。

● 343~349 行对应发送成功的情况。

352~356 行对应硬件缓冲区不足而导致正确接收到的数据帧被迫丢弃的情况。这种情况下,需要更新统计信息(表示 missed 的帧数)。

357~362 行对应硬件内部在处理要发送的数据帧的时候,就发现数据帧有问题,从而未执行真正发送的情况。这种情况下,应该调用 netif_wake_queue,因为这个错误不是硬件或发送链路出问题导致的,所以 cs8900_tx_timeout 无能为力。

```

312 static irqreturn_t cs8900_interrupt (int irq,void * id)
313 {
328     while ((status = cs8900_read (ndev, PP_ISQ))) {
331         switch (RegNum (status)) {
332             case RxEvent;

```



```
333     cs8900_receive (ndev);
334     break;
335 case TxEvent;
336     priv->stats.collisions += ColCount (cs8900_read (ndev,PP_TxCOL));
337     if (! (RegContent (status) & TxOK)) {
338         priv->stats.tx_errors++;
339         if ((RegContent (status) & Out_of_window)) priv->stats.tx_window_errors++;
340         if ((RegContent (status) & Jabber)) priv->stats.tx_aborted_errors++;
341         break;
342     } else if (priv->txlen) {
343         priv->stats.tx_packets++;
344         priv->stats.tx_bytes += priv->txlen;
345     }
346     priv->txlen = 0;
347     netif_wake_queue (ndev);
348     break;
349 case BufEvent;
350     if ((RegContent (status) & RxMiss)) {
351         ul6 missed = MissCount (cs8900_read (ndev,PP_RxMISS));
352         priv->stats.rx_errors += missed;
353         priv->stats.rx_missed_errors += missed;
354     }
355     if ((RegContent (status) & TxUnderrun)) {
356         priv->stats.tx_errors++;
357         priv->stats.tx_fifo_errors++;
358         priv->txlen = 0;
359         netif_wake_queue (ndev);
360     }
361     /* FIXME: if Rdy4Tx, transmit last sent packet (if any) */
362     break;
363 case TxCOL;
364     priv->stats.collisions += ColCount (cs8900_read (ndev,PP_TxCOL));
365     break;
366 case RxMISS;
367     status = MissCount (cs8900_read (ndev,PP_RxMISS));
368     priv->stats.rx_errors += status;
369     priv->stats.rx_missed_errors += status;
370     break;
371 }
```

```
377     }
379 }
```

(6) 数据接收处理 `cs8900_receive (struct net_device * ndev)` 中的硬件操控代码。

231 行读取寄存器 `PP_RxStatus` 以获知接收数据的状况,以供判断接收数据帧是否正确之用。

232 行读取寄存器 `PP_RxLength` 以获知接收到的数据帧长度,以供 copy 数据帧到 `skb` 之用。

234~238 行对应接收到错误的的数据帧(太短、太长、CRC 错),并更新相应统计信息。

249 行将硬件芯片接收到的数据帧 copy 到 `skb` 中(如何实现,参见稍后分析)。

```
224 static void cs8900_receive (struct net_device * ndev)
225 {
231     status = cs8900_read (ndev,PP_RxStatus);
232     length = cs8900_read (ndev,PP_RxLength);
234     if (! (status & RxOK)) {
235         priv->stats.rx_errors++;
236         if ((status & (Runt | Extradata))) priv->stats.rx_length_errors++;
237         if ((status & CRCError)) priv->stats.rx_crc_errors++;
238         return;
239     }
249     cs8900_frame_read (ndev,skb,length);
262 }
```

假设硬件芯片接收到的数据长度为 125 字节,则下面的函数 `cs8900_frame_read` 通过从 `0x19000300` 这个 I/O 内存地址连续读取 63 个半字来完成将接收到的数据(125 个有效的字节+1 个无效的字节)copy 到 `skb`。可见地址 `0x19000300` 这个寄存器实际上是硬件芯片内部帧缓存的外部读写窗口:

```
142 static inline void cs8900_frame_read (struct net_device * ndev,struct sk_buff * skb,u16
length)
143 {
144     insw (ndev->base_addr,skb_put (skb,length),(length + 1) / 2);
145 }
```

(7) 改变接收模式 `cs8900_set_receive_mode (struct net_device * ndev)` 中的硬件操控代码。

455~458 行通过操作 `PP_RxCTL` 寄存器来指令硬件芯片是否启用混杂模式。

460~463 行通过操作 `PP_RxCTL` 寄存器来指令硬件芯片是否接收多播包。

```

453 static void cs8900_set_receive_mode (struct net_device * ndev)
454 {
455     if ((ndev->flags & IFF_PROMISC))
456         cs8900_set (ndev, PP_RxCTL, PromiscuousA);
457     else
458         cs8900_clear (ndev, PP_RxCTL, PromiscuousA);
460     if ((ndev->flags & IFF_ALLMULTI) && ndev->mc_list)
461         cs8900_set (ndev, PP_RxCTL, MulticastA);
462     else
463         cs8900_clear (ndev, PP_RxCTL, MulticastA);
464 }

```

验证用户可以用 ifconfig 使网卡处于混杂模式如表 11-2 所列。

去掉 255、256 行(位于函数 cs8900_receive 中)的注释,以显示接收到的数据包的类型。

```

#define PACKET_HOST    0    /* To us      */
#define PACKET_BROADCAST 1    /* To all    */
#define PACKET_MULTICAST 2    /* To group  */
#define PACKET_OTHERHOST 3    /* To someone else */

255     if (((skb->pkt_type) == PACKET_HOST) || ((skb->pkt_type) == PACKET_OTHERHOST))
256         printk("received a frame which package type is %d\n", skb->pkt_type);

```

表 11-2 验证网卡驱动支持混杂模式

| 开发板 | Linux 机器 |
|---|---|
| # ifconfig eth0 192.168.2.17 # ifconfig eth0 promisc device eth0 entered promiscuous mode | |
| | / \$ ping -c 1 192.168.2.17 PING 192.168.2.17 (192.168.2.17) 56(84) bytes of data. 64 bytes from 192.168.2.17: icmp_seq=1 ttl=64 time=1.74 ms |
| # received a frame which package type is 0 | |

| 开发板 | Linux 机器 |
|--|--|
| | <pre> /\$ sudo arp -s 192.168.2.18 08:00:3E:26:0A:59 /\$ ping -c 1 192.168.2.18 PING 192.168.2.18 (192.168.2.18) 56(84) bytes of data. --- 192.168.2.18 ping statistics --- 1 packets transmitted, 0 received, 100% packet loss, time 0ms </pre> |
| received a frame which package type is 3 | |
| # ifconfig eth0 --promisc device eth0 left promiscuous mode | |
| | <pre> /\$ ping -c 1 192.168.2.17 PING 192.168.2.17 (192.168.2.17) 56(84) bytes of data. 64 bytes from 192.168.2.17: icmp_seq=1 ttl=64 time=1.74 ms </pre> |
| # received a frame which package type is 0 | |
| | <pre> /\$ ping -c 1 192.168.2.18 PING 192.168.2.18 (192.168.2.18) 56(84) bytes of data. --- 192.168.2.18 ping statistics --- 1 packets transmitted, 0 received, 100% packet loss, time 0ms </pre> |
| 没有任何显示 | |

4. 中断服务程序为何是 while 循环？

cs8900 会同时出现多个能引发中断的事件(例如:在双工模式下,发送数据的同时也接收到了数据),此时 cs8900 的 ISQ 寄存器就表明了这多个中断事件。

需要用循环来处理全部中断事件。

5. 中断为何是 EINT9? 为什么是上升沿触发?

```
76      .start = IRQ_EINT9,
414    set_irq_type(ndevice->irq, IRQ_TYPE_EDGE_RISING);
```

CPU 的 EINT9 引脚接在了 CS8900 的 INTRQ0 上。而 CS8900 的默认配置(无 EEPROM 时)启用的是 4 个中断中的 INTRQ0,即使不是这样,我们在程序中也明确指定的是 INTRQ0。

查 cs8900 硬件芯片手册可知:中断发生时,INTRQ0 从低电平变为高电平。

6. 为什么要设置内存控制器?

```
500    __raw_writel(0x2211d110,S3C2410_BWSCON);
501    __raw_writel(0x1f7c,S3C2410_BANKCON3);
```

cs8900 的 I/O 内存地址占用的是 BANK3 的地址空间:

```
__raw_writel(0x2211d110,S3C2410_BWSCON)
```

● 0xd=0b1101,表示 bank3 的配置为:

■ (0b1)使用 nBE,而不是 nWBE,即:nWBE/nBE 引脚在读写字节时均输出有效信号,而不是仅仅在写字节时才输出有效信号。这是由于 nWBE1/nBE1 引脚接的是 cs8900 的 nSBHE(system bus high enable)引脚,表示 16 位数据中的高 8 位有效,在读写字节中均有效,而不仅仅在写字节中有效。

■ (0b1)表示 enable wait 信号。CPU 的 nWAIT 信号接的是 cs8900 的 IOCHRDY 引脚。

■ (0b01)表示数据位宽 16bit。因为 cs8900 的数据总线位宽为 16bit。

● 0x2,是 bank6 正确的配置:

```
__raw_writel(0x1f7c,S3C2410_BANKCON3)
```

● bank3 的控制寄存器,配置是正确的,其中的各个参数的含义请查阅“s3c2440 硬件手册”第 5 章中的 BANKCON3 寄存器说明。



第 12 章

其他重要设备驱动开发实战

12.1 块设备驱动初步(以 ramdisk 为例)

本节的素材以及源代码(稍有改动)均来源于《Linux Device Driver》，因此本节可视为该书块设备驱动相关章节的阅读理解。

12.1.1 体验块设备驱动

本驱动模拟了一个硬盘。想象你去中关村买了一个硬盘，迫不及待地安装在 Linux 机器上，怎么样才能使用这个新硬盘呢？当然要先把它驱动起来。步骤如下：

1. **make 生成 sbull.ko 后加载驱动：**`# insmod sbull.ko`

2. 对硬盘分区

- 执行 `# fdisk /dev/sbulla`
- 输入 `x`, 进入高级菜单
- 输入 `h`, change number of heads 为 4
- 输入 `c`, change number of cylinders 为 4
- 输入 `r`, 退回主菜单
- 顺次输入 `n`、`p`、`1`、`1`、`4`, 创建一个主分区占有 cylinders 1—4
- 输入 `w`, 保存并退出

3. 输入 `# mke2fs /dev/sbulla1` 在硬盘分区上格式化 ext2 文件系统

4. 挂载新硬盘上的分区

- 输入 `# mkdir testdir` 创建空目录
- 输入 `# mount -t ext2 /dev/sbulla1 ./testdir`

之后, 就可以通过 `testdir` 目录来访问新硬盘分区了。

12.1.2 块设备驱动框架介绍

1. 驱动接口简介

(1) 上层接口。

① 用户接口。

- 块设备: /dev/sda、/dev/sda2、/dev/ram0

- 用户空间程序

- mkfs、mount、fdisk;

- 直接调用 open、release、ioctl 接口;

- 读写时,不直接调用读写接口,而是将读写 request 提交给 OS 的 block layer 层。

② 操作系统接口。

- block layer I/O scheduler 决定需要执行磁盘 I/O 时,调用驱动的读写接口。

- 抽象物理设备为 cylinder、header、sector 组成,视其为编号从 0 开始的 flat 型 sector 集合(注 1)。

- 总假定物理设备一个 sector 大小为 512B(注 2)。

(2) 下层接口

① 抽象物理设备为编号从 0 开始的 flat 型 sector 集合,但转换 sector 大小为物理设备的实际值,通过物理寄存器的读写,将数据写入指定的 sector 位置。

② 由物理设备的中断(读写完成)来唤醒用户进程(或内核线程)。

注 1:借助 minor number 辨别 partition 编号(也可能是整个磁盘),再借助分区表决定分区的起始 sector 号。

注 2: #define KERNEL_SECTOR_SHIFT 9

#define KERNEL_SECTOR_SIZE (1 << KERNEL_SECTOR_SHIFT)

2. 块设备结构体

块设备结构体 struct gendisk 是块设备驱动中最重要的数据结构,它在操作系统和驱动中代表一个物理磁盘。其主要字段有:

- major: 磁盘对应的主设备号,出现在 /proc/devices 中。

```
register_blkdev(sbull_major, "sbull");
```

- first_minor: 磁盘对应的第一个次设备号。

```
dev->gd->first_minor = which * SBULL_MINORS;
```

- minors: 磁盘拥有的次设备号的总数。

```
dev->gd = alloc_disk(SBULL_MINORS);
```

- `disk_name`: 磁盘的名称。出现在 `/proc/partitions` 中。
- `fops`: 块设备驱动中的功能函数。包括 `open`、`release`、`media_change`、`revalidate_disk`、`ioctl` 等。

- `queue`: OS 回调读写函数时使用的 request queue 队列(它绑定了读写函数)。
- `private_data`: 私有数据, 常用于存放包裹设备结构体。
- `capacity`: 512 字节大小的 sector 的总数量。

```
set_capacity(dev->gd, nsectors * (hardsect_size/KERNEL_SECTOR_SIZE));
```

3. 块设备结构体注册

(1) 申请 major number:

```
sbull_major = register_blkdev(sbull_major, "sbull"); //sbull 出现在/proc/devices 中
```

(2) 分配(初始生成)块设备结构体:

```
dev->gd = alloc_disk(SBULL_MINORS); //同时也关联设备号数量
```

(3) 将块设备结构体与设备号、设备操作函数、读写队列关联:

```
dev->gd->major = sbull_major; //关联主设备号
dev->gd->first_minor = which * SBULL_MINORS; //关联次设备号
dev->gd->fops = &sbull_ops; //关联功能函数
dev->queue = blk_init_queue(sbull_request, &dev->lock);
dev->gd->queue = dev->queue; //关联读写队列
```

(4) 将块设备结构体注册进 OS:

```
add_disk(dev->gd);
```

4. 块设备结构体注销

(1) 将块设备结构体从 OS 中注销:

```
del_gendisk(dev->gd);
```

(2) 销毁块设备结构体(移除对 kobject 的最后 ref., 释放结构体内存):

```
put_disk(dev->gd);
```

(3) 释放 major number:

```
unregister_blkdev(sbull_major, "sbull");
```

图 12-1 为块设备驱动的内核数据结构。



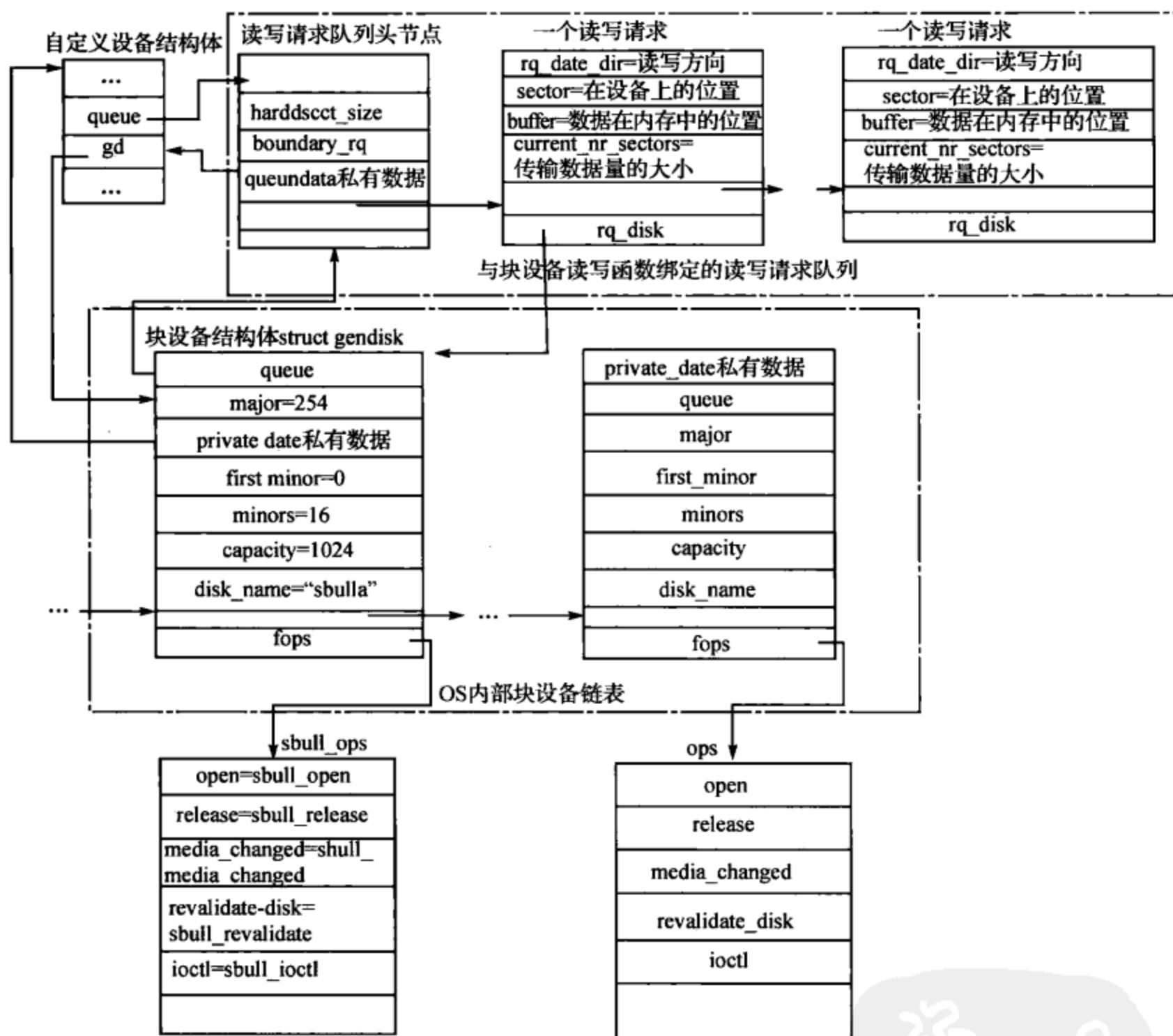


图 12-1 块设备驱动的内核数据结构

5. 块设备的简单读写-原理

- (1) 用户程序或内核组件提出读写 request 时,会将该 request 提交给 block layer 层。
- (2) block layer 层构造 request 结构,并将其链入块设备结构体绑定的 request queue 中。
- (3) 在适当的时候,block layer 层回调 request queue 中绑定的驱动中的读写函数,并将 request queue 的指针作为参数传给驱动中的读写函数。
- (4) 驱动中的读写函数根据传给它的 request queue 的指针,从 request queue 中取出一个 request,根据 request 中指定的方向(读或写)、数据在内存中的位置、在设备上的位置(起始

sector 编号)、传输数据量的大小(sector 数量)完成物理读写。

(5) 驱动通知 block layer 层实际完成的读写量, block layer 层据此更新其内部各个数据结构;并告知驱动是否整个 request 已经处理完成,若是,则驱动负责将 request 从 request queue 中摘下,并释放 request 结构的内存,唤醒等待该 request 完成的所有进程。

6. 读写队列结构体—注册与注销

(1) 分配(初始生成)读写队列结构体,并与读写函数绑定:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

block layer 在回调读写函数 sbull_request 时,会先获得自旋锁 dev->lock,这样 block layer 就可以与驱动的其他函数共享相同的临界区:

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

设置实际设备的扇区大小,这样 block layer 层就会根据实际设备能够处理的扇区大小来构造 request 结构体,从而不会出现实际设备处理不了的 request:

```
dev->queue->queuedata = dev
```

(2) 将块设备结构体与读写队列关联:

```
dev->gd->queue = dev->queue;
```

(3) 将读写队列结构体间接注册进 OS:

```
add_disk(dev->gd);
```

(4) 将读写队列结构体间接从 OS 中注销:

```
del_gendisk(dev->gd);
```

(5) 销毁读写队列结构体(移除对 kobject 的最后 ref., 释放结构体内存):

```
blk_cleanup_queue(dev->queue);
```

12.1.3 块设备的简单读写实现代码分析

```
108 static void sbull_request(request_queue_t *q)
109 {
110     struct request *req;
112     while ((req = elv_next_request(q)) != NULL) { //从 request queue 中取出一个 request,
//循环直到 request queue 中的所有 request 被传完
//因为读写队列与块设备结构体已关联,所以 block layer 层在将读写请求链入 request queue 时,能将
//rq_disk 字段指向块设备结构体
113     struct sbull_dev *dev = req->rq_disk->private_data;
```

```

114     if (! blk_fs_request(req)) {
115         end_request(req, 0);
116         continue;
117     }
118     sbull_transfer(dev, req->sector, req->current_nr_sectors, //根据 request 中指定的
//方向(rq_data_dir)、数据在内存中的位置( req->buffer)、
119         req->buffer, rq_data_dir(req)); //在设备上的位置( req->sector)、传输数据量的
//大小( req->current_nr_sectors),完成物理读写
120     end_request(req, 1); //通知 block layer 层;将 request 从 request queue 中摘下;释放
//request 结构的内存,唤醒等待该 request 完成的所有进程
121 }
122 }
void end_request(struct request * req, int uptodate)
{
    //驱动通知 block layer 层实际完成的读写量,block layer 层据此更新其内部各个数据结构;并告知
//驱动是否整个 request 已经处理完成
    if (! end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
        add_disk_randomness(req->rq_disk);
        blkdev_dequeue_request(req); //若是,驱动则负责将 request 从 request queue 中摘下
        end_that_request_last(req); //释放 request 结构的内存,唤醒等待该 request 完成的所有进程
    }
}

```

简单读写的不足:

(1) 一次只命令硬件传输一个数据块 segment(内存中不超过 1page 的连续单元),其只是 request 中的一小部分而已。

(2) 虽然简单读写采用 while 循环请求 elv_next_request,但 block 层会认为硬件可能由于某种原因不能一次性完成一个完整 request 的传输,因此相邻两次 elv_next_request 极有可能返回的是不同的 request。

(3) block layer 尽了很大努力,实施电梯调度算法(drivers/block/ll_rw_block.c and elevator.c),使得一个 request 结构体中包含多个在内存中离散,但在物理设备上却连续的数据块。

(4) 先后两次 elv_next_request 的简单读写,使得磁头必须寻道,产生较大延迟。

(5) 简单读写忽视 block layer 层的工作,对同一个 request 结构体中包含的多个 segment 在物理设备上连续,不予理睬,实在是暴殄天物。

12.1.4 块设备的高效读写实现代码分析

1. 请求队列

(1) 一个块设备的 I/O 请求的序列。

(2) 跟踪未完成的块 I/O 请求。

(3) 允许使用多 I/O 调度器,以最大化性能的方式提交 I/O 请求给你的驱动,I/O 调度器还负责合并邻近的请求。

(4) 请求队列的实现代码在内核的 `drivers/block/Ll_rw_block.c` 和 `elevator.c` 源码文件中。

2. 块设备的高效读写-原理与实现

一个 request 结构体中包含多个在内存中离散、但在物理设备上却连续的数据块,由 bio 和 bio_vec 结构体来表示,如图 12-2、图 12-3 所示。

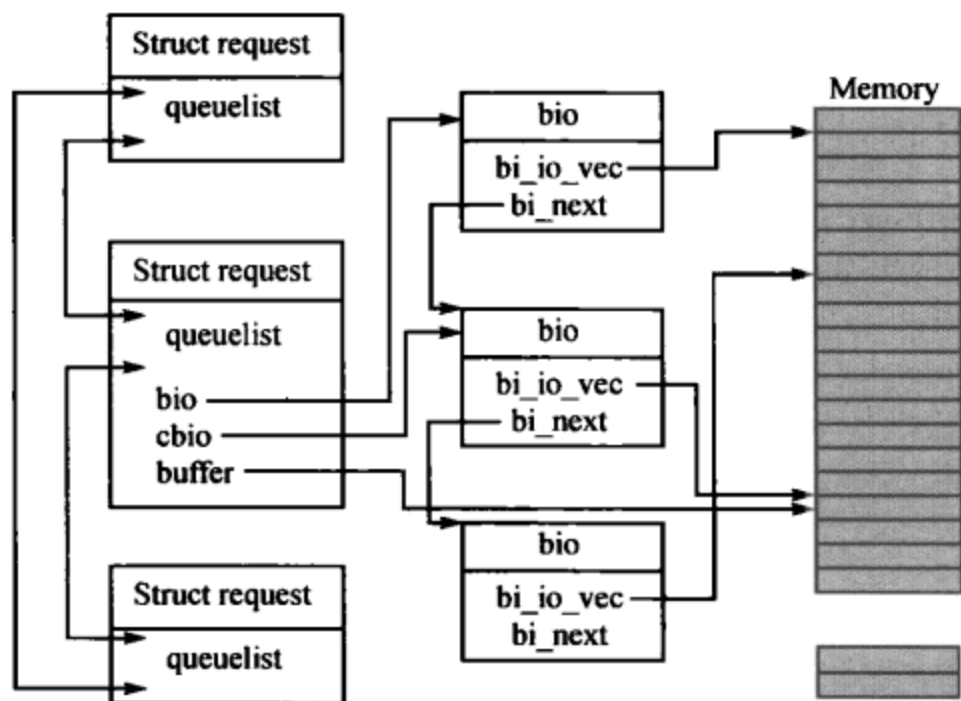


图 12-2 bio 结构体

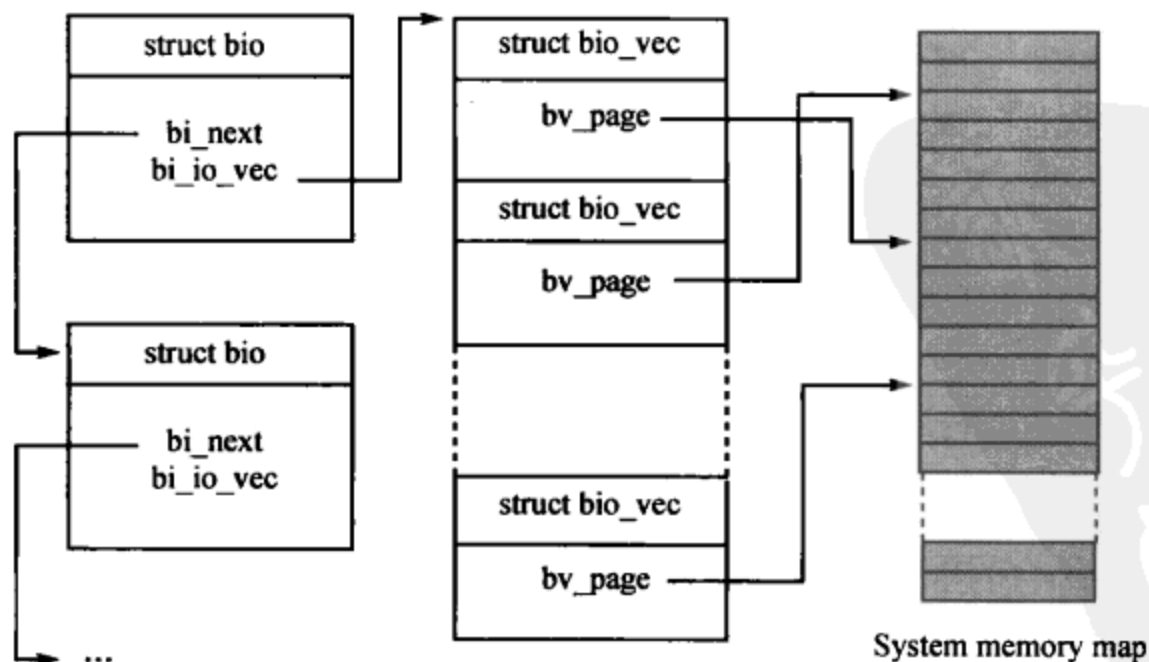


图 12-3 bio_vec 结构体


```

337 static void setup_device(struct sbull_dev * dev, int which)
362     switch (request_mode) {
370         case RM_FULL;
371             dev->queue = blk_init_queue(sbull_full_request, &dev->lock);
374             break;
385     }
387     dev->queue->queuedata = dev;
409 }

175 static void sbull_full_request(request_queue_t * q)
176 {
177     struct request * req;
178     int sectors_xferred;
179     struct sbull_dev * dev = q->queuedata;
181     while ((req = elv_next_request(q)) != NULL) { //每次取出读写队列中的一个 request
182         sectors_xferred = sbull_xfer_request(dev, req);
183         if (!end_that_request_first(req, 1, sectors_xferred)) {
184             blkdev_dequeue_request(req);
185             end_that_request_last(req, 1);
186         }
187     }
188 }
189 }
190 }

#define rq_for_each_bio(_bio, rq) \
    if ((rq->bio)) \
        for (_bio = (rq)->bio; _bio; _bio = _bio->bi_next)

157 static int sbull_xfer_request(struct sbull_dev * dev, struct request * req)
158 {
159     struct bio * bio;
160     int nsect = 0;
162     rq_for_each_bio(bio, req) { //while 循环,每次取出 req 中的一个 bio
163         sbull_xfer_bio(dev, bio);
164         nsect += bio->bi_size/KERNEL_SECTOR_SIZE; //bi_size 记录一个 bio 中数据的总字节数
165     }
167     return nsect;
168 }

#define bio_for_each_segment(bvl, bio, i) \

```

```

__bio_for_each_segment(bvl, bio, i, (bio) ->bi_idx)

#define __bio_for_each_segment(bvl, bio, i, start_idx) \
    for (bvl = bio_iovec_idx((bio), (start_idx)), i = (start_idx); \
         i < (bio) ->bi_vcnt; \
         bvl++, i++)

133 static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
134 {
135     int i;
136     struct bio_vec *bvec;
137     sector_t sector = bio->bi_sector; //bi_sector 记录 bio 中首字节应位于硬件的哪个
//扇区。bio 中的所有 segment 在硬件上的 sector 位置全部连续
138     /* Do each segment independently. */
139     bio_for_each_segment(bvec, bio, i) { //while 循环,每次取出 bio 中的一个 bio_vec 来传输
140         char *buffer = __bio_kmap_atomic(bio, i, KM_USER0); //获取 bv_page 的内核
//virtual address
141         sbull_transfer(dev, sector, bio_cur_sectors(bio), //bio_cur_sectors(bio) 求出
//bio 当前 segment 的大小(sector 数目)
142             buffer, bio_data_dir(bio) == WRITE);
143         sector += bio_cur_sectors(bio); //累进数据在硬件上的位置(sector)
144         __bio_kunmap_atomic(bio, KM_USER0);
145     }
146     return 0; /* Always "succeed" */
147 }
148
149
150
151
152

```

12.1.5 块设备的其他操作接口 fops

块设备的其他操作接口包括 open、release、media_change、revalidate_disk、ioctl 等。

1. open 与 release(本驱动可以模拟光盘从光驱中更换)

```

216 static int sbull_open(struct inode *inode, struct file *filp)
217 {
218     struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data; //inode->
//i_bdev->bd_disk 指向关联的 gendisk structure
219     if (!dev->users)
220         check_disk_change(inode->i_bdev); //check_disk_change 将导致对 media_change 的
//调用,若介质已改变,将导致调用 revalidate_disk
221     dev->users++;

```

228 }

media_changed 不为 NULL 时,则会被内核 API check_disk_change 所调用。

在 media_changed 返回 true 的情况下, revalidate_disk 会被内核 API check_disk_change 所调用。

```
int check_disk_change(struct block_device * bdev)
{
    struct gendisk * disk = bdev->bd_disk;
    struct block_device_operations * bdops = disk->fops;
    if (! bdops->media_changed) //media_changed 为 NULL
        return 0;
    if (! bdops->media_changed(bdev->bd_disk)) //media_changed 不为 NULL 时,则会被内核
//API check_disk_change 所调用
        return 0;
    if (__invalidate_device(bdev))
        printk("VFS: busy inodes on changed media.\n");
    if (bdops->revalidate_disk) // 在 media_changed 返回 true 的情况下, revalidate_disk 不为
//NULL 时,则会被内核 API check_disk_change 所调用
        bdops->revalidate_disk(bdev->bd_disk);
    if (bdev->bd_disk->minors > 1)
        bdev->bd_invalidated = 1;
    return 1;
}
```

```
230 static int sbull_release(struct inode * inode, struct file * filp)
231 {
232     struct sbull_dev * dev = inode->i_bdev->bd_disk->private_data;
235     dev->users--;
244 }
```

2. media_change 与 revalidate_disk

media_changed 不为 NULL 时,则会被内核 API check_disk_change 所调用。若磁盘介质已更改,应返回 true;否则返回 false。

在 media_changed 返回 true 的情况下, revalidate_disk 会被内核 API check_disk_change 所调用。应完成对新磁盘介质进行操作的准备工作。

```
249 int sbull_media_changed(struct gendisk * gd)
250 {
251     struct sbull_dev * dev = gd->private_data;
```

```

253     return dev->media_change;
254 }

260 int sbull_revalidate(struct gendisk *gd)
261 {
262     struct sbull_dev *dev = gd->private_data;
263     if (dev->media_change) {
264         dev->media_change = 0;
265         // memset (dev->data, 0, dev->size);
266     }
267 }
268 }

```

3. ioctl

大部分的 ioctl 命令都已经被 block layer 层所截获并处理, 到达不了驱动程序。

驱动 ioctl 函数中需要处理的命令是 HDIO_GETGEO(用户, 例如 fdisk, 要求获得磁盘的几何参数)。

```

291 int sbull_ioctl (struct inode * inode, struct file * filp,
292                 unsigned int cmd, unsigned long arg)
293 {
294     long size;
295     struct hd_geometry geo;
296     struct sbull_dev * dev = filp->private_data;
297     switch(cmd) {
298         case HDIO_GETGEO:
299             /*
300              * Get geometry; since we are a virtual device, we have to make
301              * up something plausible.  So we claim 16 sectors, four heads,
302              * and calculate the corresponding number of cylinders.  We set the
303              * start of data at sector four.
304              */
305             size = dev->size * (hardsect_size/KERNEL_SECTOR_SIZE);
306             size = dev->size / KERNEL_SECTOR_SIZE;
307             geo.cylinders = (size & ~0x3f) >> 6;
308             geo.heads = 4;
309             geo.sectors = 16;
310             geo.start = 4;
311             if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
312                 return -EFAULT;
313             return 0;

```

```

316     }
317     return -ENOTTY; /* unknown command */
318 }
319 }

```

12.2 LCD 驱动

12.2.1 LCD 裸机驱动

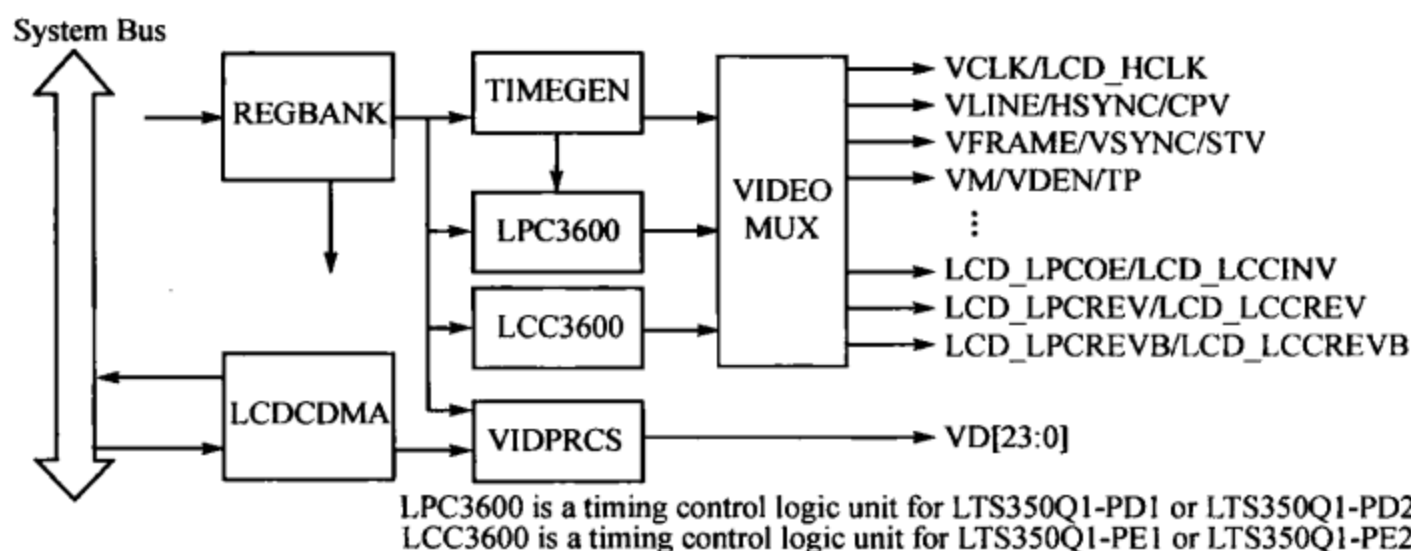


图 12-4 LCD 控制器硬件组成框图

1. S3C2440 内部 LCD 控制器硬件结构(图 12-4)

要使一块 LCD 正常的显示文字或图像,不仅需要 LCD 驱动器,而且还需要相应的 LCD 控制器。在通常情况下,生产厂商把 LCD 驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起,而 LCD 控制器则是由外部的电路来实现,现在很多的 MCU 内部都集成了 LCD 控制器,如 S3C2410/2440 等。通过 LCD 控制器就可以产生 LCD 驱动器所需要的控制信号来控制 STN/TFT 屏了。

根据 s3c2440a 硬件手册来描述一下这个集成在 S3C2440 内部的 LCD 控制器:

LCD 控制器由 REG BANK、LCD CDMA、TIME GEN、VIDPRCS 寄存器组成。

REG BANK 由 17 个可编程的寄存器和一块 256×16 的调色板内存组成,它们用来配置 LCD 控制器的。

LCD CDMA 是一个专用的 DMA,它能自动地把在帧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 的干预的情况下就可显示在 LCD 屏上。

VIDPRCS 接收来自 LCD CDMA 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,它生成 LCD 驱动器需要的控制信号,比如 VSYNC、HSYNC、VCLK 和 LEND 等等,而这些控制信号又与 REG BANK 寄存器组中的 LCDCON1/2/3/4/5 的配置密切相关,通过不同的配置,TIMEGEN 就能产生这些信号的不同形态,从而支持不同的 LCD 驱动器(即不同的 STN/TFT 屏)。

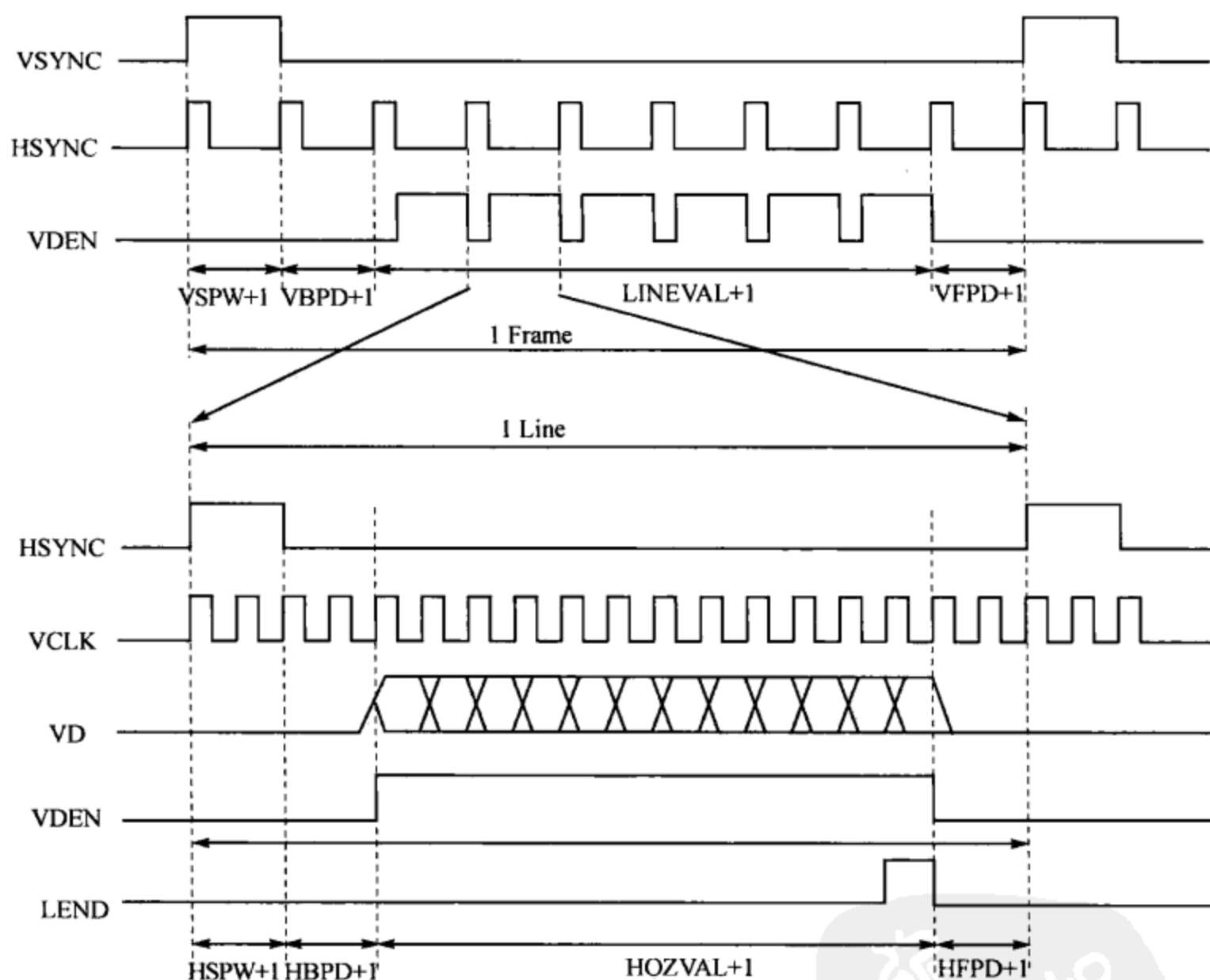


图 12-5 TFT 屏工作时序

2. 常见 TFT 屏工作时序分析(图 12-5)

(1) LCD 提供的外部接口信号:

- ① VSYNC/VFRAME/STV: 垂直同步信号(TFT)/帧同步信号(STN)/SEC TFT 信号。
- ② HSYNC/VLINE/CPV: 水平同步信号(TFT)/行同步脉冲信号(STN)/SEC TFT 信号。
- ③ VCLK/LCD_HCLK: 像素时钟信号(TFT/STN)/SEC TFT 信号。

④ VD[23:0]:LCD 像素数据输出端口(TFT/STN/SEC TFT)。

⑤ VDEN/VM/TP:数据使能信号(TFT)/LCD 驱动交流偏置信号(STN)/SEC TFT 信号。

⑥ LEND/STH:行结束信号(TFT)/SEC TFT 信号。

⑦ LCD_LPCOE:SEC TFT OE 信号。

⑧ LCD_LPCREV:SEC TFT REV 信号。

⑨ LCD_LPCREVB:SEC TFT REVB 信号。

(2) 所有显示器显示图像的原理都是从上到下,从左到右的。一幅图像可以看作一个矩形,由很多排列整齐的点一行一行组成,这些点称为像素。那么这幅图在 LCD 上的显示原理就是:

① 显示指针从矩形左上角的第一行第一个点开始,一个点一个点地在 LCD 上显示,在上面的时序图上用时间线表示就为 VCLK,称为像素时钟信号。

② 当显示指针一直显示到矩形的右边就结束这一行,那么这一行的动作在上面的时序图中就称为 1 Line。

③ 接下来显示指针又回到矩形的左边从第二行开始显示,注意,显示指针在从第一行的右边回到第二行的左边是需要一定的时间的,称之行切换。

④ 如此类推,显示指针就这样一行一行地显示至矩形的右下角才把一幅图显示完成。因此,这一行一行的显示在时间线上看,就是时序图上的 HSYNC。

⑤ 然而,LCD 的显示并不是对一副图像快速地显示一下,为了持续和稳定地在 LCD 上显示,就需要切换到另一幅图上(另一幅图可以和上一副图一样或者不一样,目的只是为了将图像持续地显示在 LCD 上)。那么这一幅一幅的图像就称为帧,在时序图上就表示为 1 Frame,因此从时序图上可以看出 1 Line 只是 1 Frame 中的一行。

⑥ 同样地,在帧与帧切换之间也是需要一定的时间的,称为帧切换,那么 LCD 整个显示的过程在时间线上看,就可表示为时序图上的 VSYNC。

(3) 上面时序图上各时钟延时参数的含义如下(这些参数的值,LCD 生产厂商会提供相应的数据手册):

① VBPD(vertical back porch):表示在一帧图像开始时,垂直同步信号以后的无效的行数,对应驱动中的 upper_margin。

② VFPPD(vertical front porch):表示在一帧图像结束后,垂直同步信号以前的无效的行数,对应驱动中的 lower_margin。

③ VSPW(vertical sync pulse width):表示垂直同步脉冲的宽度,用行数计算,对应驱动中的 vsync_len。

VSYNC(垂直同步信号)有效时,表示一帧数据的开始,此后需要经过(VPSW+1+VBPD+1)个无效的行,第一个有效的行才出现。随后连续发出(LINEVAL+1)行的有效数据。

最后是(VFPD+1)个无效的行,完整的一帧结束,紧接着就是下一个帧的数据了(即:下一个VSYNC信号)。

- HBPD(horizontal back porch):表示从水平同步信号开始到一行的有效数据开始之间的VCLK的个数,对应驱动中的left_margin。

- HFPD(horizontal front porch):表示一行的有效数据结束到下一个水平同步信号开始之间的VCLK的个数,对应驱动中的right_margin。

- HSPW(horizontal sync pulse width):表示水平同步信号的宽度,用VCLK计算,对应驱动中的hsync_len。

HSYNC(水平同步信号)有效时,表示一行数据的开始,此后需要经过(HSPW+1+HBPD+1)个无效的像素,第一个有效的像素才出现。随后连续发出(HOZVAL+1)个像素的有效数据。最后是(HFPD+1)个无效的像素,完整的一行结束,紧接着就是下一行的数据了(即:下一个HSYNC信号)。

3. S3C2440 LCD 控制器重要寄存器精解

对于以上这些参数的值将分别保存到 REG BANK 寄存器组中的 LCDCON1/2/3/4/5 寄存器中:(对寄存器各个位的含义请查看 S3c2440 数据手册 LCD 部分)

(1) LCDCON1:17~8 位 CLKVAL(与 HCLK 一起,决定 VCLK)

6~5 位扫描模式(11 代表 TFT 屏)

4~1 位色位模式(1BPP、8BPP、16BPP 等。1100 = 16 bpp for TFT)

0 ENVID(1 表示启用 video 数据输出)

(2) LCDCON2:31~24 位 VBP

23~14 位 LINEVAL

13~6 位 VFPD

5~0 位 VSPW

(3) LCDCON3:25~19 位 HBP

18~8 位 HOZVAL

7~0 位 HFPD

(4) LCDCON4:7~0 位 HSPW

(5) LCDCON5:其他一些重要参数设置

4. 帧缓冲(FrameBuffer)

FrameBuffer 是为显示设备提供的一段内存,应用程序通过向 FrameBuffer 写入显示点对应的颜色值,对应的颜色就会自动地在屏幕上显示,从而不用去关心具体的硬件细节。之所以能做到这一点,是因为 LCD 控制器中设置了 FrameBuffer 的起始地址,LCD 控制器会通过 DMA 的方式自动从 FrameBuffer 取得显示点的颜色信息,通过在数据总线 VD[23:0]输出相

应信息,由 LCD 驱动器解释这些数据,然后显示在 LCD 屏上。

下面来看一下在不同色位模式下,图像数据在 FrameBuffer 中如何存储。

显示器的每个像素的颜色由三部分组成:红(red)、绿(green)、蓝(blue)。它们被称为三原色,这三者的混合几乎可以表示人眼所能识别的所有颜色。比如可以根据颜色的浓烈程度将三原色都分为 256 个级别(0~255),则可以使用 255 级的红色、255 级的绿色、255 级的蓝色组合成白色,可以使用 0 级的红色、0 级的绿色、0 级的蓝色组合成黑色。

LCD 控制器可以支持单色(1BPP)、4 级灰度(2BPP)、16 级灰度(4BPP)、256 色(8BPP)的调色板模式,支持 64K(16BPP)和 16M(24BPP)非调色板显示模式。出于简化以便于初学者学习的考虑,这里只讲解 64K(16BPP)色显示模式下,图像数据的存储格式。

64K(16BPP)色的显示模式就是使用 16 位的数据来表示一个像素的颜色。这 16 位数据的格式又分为两种:5:6:5 和 5:5:5:1,前者使用高 5 位来表示红色,中间的 6 位来表示绿色,低 5 位来表示蓝色;后者的高 15 位从高到低分成 3 个 5 位来表示红色、绿色、蓝色,最低位表示透明度。5:5:5:1 的格式也被称为 RGBA 格式(A:Alpha,表示透明度)。

5. TFT LCD 裸机驱动实例分析

(1) 体验裸机驱动。

将光盘\work\studydriver\lcd\baredrv 下相应目录的程序 make 后,把 lcd. bin 烧写到开发板的 Nor Flash 中,重启开发板,将会在 lcd 屏上看到 8 根直线,分别是红色、绿色、蓝色、白色、黄色、紫色、银色、金色。

(2) 代码分析。

① main.c 第 6 行,调用 Test_Lcd_Tft_16Bit_240320,以 16BPP 颜色模式在 240×320 的 LCD 屏上显示图像。

```
4 int main()
5 {
6     Test_Lcd_Tft_16Bit_240320();
7     return 0;
8 }
```

② lcdlib.c 实现 Test_Lcd_Tft_16Bit_240320。它调用 Lcd_Port_Init 设置 GPIO 引脚用于 LCD,Lcd_Port_Init 以 240×320、16BPP(5:6:5 模式)、TFT 屏的方式初始化 LCD 控制器,Lcd_Port_Init 打开 LCD 电源,Lcd_EnvidOnOff 使能(或禁用)LCD 控制器输出信号,DrawLine 以不同颜色和起始点画线。

```
12 void Test_Lcd_Tft_16Bit_240320(void)
13 {
14     Lcd_Port_Init(); // 设置 LCD 引脚
15     Tft_Lcd_Init(MODE_TFT_16BIT_240320); // 初始化 LCD 控制器
```

```

16    Lcd_PowerEnable(0, 1); // 设置 LCD_PWREN 有效,它用于打开 LCD 的电源
17    Lcd_EnvidOnOff(1);      // 使能 LCD 控制器输出信号
18
19    ClearScr(0x0); // 清屏,黑色
20
21    DrawLine(0, 0, 239, 0, 0xff0000); // 红色
22    DrawLine(0, 0, 0, 319, 0x00ff00); // 绿色
23    DrawLine(239, 0, 239, 319, 0x0000ff); // 蓝色
24    DrawLine(0, 319, 239, 319, 0xffffffff); // 白色
25    DrawLine(0, 0, 239, 319, 0xffff00); // 黄色
26    DrawLine(239, 0, 0, 319, 0xff00ff); // 紫色
27    DrawLine(120, 0, 120, 319, 0xe6e8fa); // 银色
28    DrawLine(0, 160, 239, 160, 0xcd7f32); // 金色
29 }

```

③ lcddrv.c 实现了 Lcd_Port_Init、Lcd_Port_Init、Lcd_EnvidOnOff。各个宏的定义参见 lcddrv.h

598

```

void Tft_Lcd_Init(int type)
{
    switch(type)
    {
        case MODE_TFT_16BIT_240320:
            /*
             * 设置 LCD 控制器的控制寄存器 LCDCON1~5
             * 1. LCDCON1:
             *     设置 VCLK 的频率: VCLK(Hz) = HCLK/[(CLKVAL + 1)x2]
             *     选择 LCD 类型: TFT LCD
             *     设置显示模式: 16BPP
             *     先禁止 LCD 信号输出
             * 2. LCDCON2/3/4:
             *     设置控制信号的时间参数
             *     设置分辨率,即行数及列数
             *     现在,可以根据公式计算出显示器的频率:
             *     当 HCLK = 100MHz 时,
             *     Frame Rate = 1/[(VSPW + 1) + (VBPD + 1) + (LINEVAL + 1) + (VFPD + 1)]x
             *                   [(HSPW + 1) + (HBPD + 1) + (HFPD + 1) + (HOZVAL + 1)]x
             *                   [2x(CLKVAL + 1)/(HCLK)]
             *                   = 60Hz
             * 3. LCDCON5:

```

```

* 设置显示模式为 16BPP 时的数据格式: 5:6:5
* 设置 HSYNC、VSYNC 脉冲的极性(这需要参考具体 LCD 的接口信号): 反转
* 半字(2 字节)交换使能
* /
LCDCON1 = (CLKVAL_TFT_240320<<8) | (LCDTYPE_TFT<<5) | \
            (BPPMODE_16BPP<<1) | (ENVID_DISABLE<<0);
LCDCON2 = (VBPD_240320<<24) | (LINEVAL_TFT_240320<<14) | \
            (VFPD_240320<<6) | (VSPW_240320);
LCDCON3 = (HBPD_240320<<19) | (HOZVAL_TFT_240320<<8) | (HFPD_240320);
LCDCON4 = HSPW_240320;
LCDCON5 = (FORMAT8BPP_565<<11) | (HSYNC_INV<<9) | (VSYNC_INV<<8) | \
            (HWSWP<<0);
LDCSADDR1 = ((LCDFRAMEBUFFER>>22)<<21) | LOWER21BITS(LCDFRAMEBUFFER>>1);
fb_base_addr = LCDFRAMEBUFFER; // LCDFRAMEBUFFER = 0x30400000, 设置 FrameBuffer 起始地址
bpp = 16;
xsize = 240;
ysize = 320;

break;
default:
break;
}

/*
* 设置是否输出 LCD 电源开关信号 LCD_PWREN
* 输入参数:
*   invpwren: 0 - LCD_PWREN 有效时为正常极性
*             1 - LCD_PWREN 有效时为反转极性
*   pwren:    0 - LCD_PWREN 输出有效
*             1 - LCD_PWREN 输出无效
* /
void Lcd_PowerEnable(int invpwren, int pwren)
{
    GPGCON = (GPGCON & (~ (3<<8))) | (3<<8); // GPG4 用做 LCD_PWREN
    GPGUP   = (GPGUP & (~ (1<<4))) | (1<<4); // 禁止内部上拉

    LCDCON5 = (LCDCON5 & (~ (1<<5))) | (invpwren<<5); // 设置 LCD_PWREN 的极性: 正常/反转
    LCDCON5 = (LCDCON5 & (~ (1<<3))) | (pwren<<3);    // 设置是否输出 LCD_PWREN
}

```

```

/*
 * 设置 LCD 控制器是否输出信号
 * 输入参数:
 *  onoff:
 *      0 : 关闭
 *      1 : 打开
 */
void Lcd_EnvidOnOff(int onoff)
{
    if (onoff == 1)
        LCDCON1 |= 1;          // ENVID ON
    else
        LCDCON1 &= 0x3fffe;    // ENVID Off
}

```

④ framebuffer.c 实现了画点函数 PutPixel, 并通过 PutPixel 实现画线函数 DrawLine 和清屏函数 ClearScr:

```

void DrawLine(int x1, int y1, int x2, int y2, int color)
{
    int dx, dy, e;
    dx = x2 - x1;
    dy = y2 - y1;

    if(dx >= 0)
    {
        if(dy >= 0) // dy >= 0
        {
            if(dx >= dy) // 1/8 octant
            {
                e = dy - dx/2;
                while(x1 <= x2)
                {
                    PutPixel(x1, y1, color);
                    if(e > 0){y1 + = 1; e - = dx;}
                    x1 + = 1;
                    e + = dy;
                }
            }
            else

```

蘇子知覺
PDG


```

        { ...
        }
    }
    else // dy<0
    {
    }
}
else //dx<0
{.....
}
}

/*
 * 画点其实就是按照 5 : 6 : 5 的 RGB 格式将颜色值写入帧内存(起始地址在 fb_base_addr)
 * 中对应点的位置
 * 输入参数:
 *     x,y : 像素坐标
 *     color: 颜色值
 *     对于 16BPP: color 的格式为 0xAARRGGBB (AA = 透明度),
 *     需要转换为 5 : 6 : 5 格式
 */
void PutPixel(UINT32 x, UINT32 y, UINT32 color)
{
    UINT8 red,green,blue;

    switch (bpp){
        case 16:
        {
            UINT16 * addr = (UINT16 *)fb_base_addr + (y * xsize + x);
            red    = ((color >> 16) & 0xff) >> 3;
            green  = ((color >>  8) & 0xff) >> 2;
            blue   = ((color >>  0) & 0xff) >> 3;
            color = (red << 11) | (green << 5) | blue; // 格式 5:6:5
            * addr = (UINT16) color;
            break;
        }
        default:
            break;
    }
}

```

12.2.2 帧缓冲设备驱动框架结构

帧缓冲(FrameBuffer)设备为标准的字符型设备,在 Linux 中主设备号 29,定义在/include/linux/major.h 中的 FB_MAJOR,次设备号定义帧缓冲的个数,最大允许有 32 个 FrameBuffer,定义在/include/linux/fb.h 中的 FB_MAX,对应于文件系统下/dev/fb%d 设备文件。

1. 帧缓冲设备驱动在 Linux 子系统结构(图 12-6)

从下面这幅图可以看出,帧缓冲设备在 Linux 中也可以看做是一个完整的子系统,大体由 fbmem.c 和 xxxfb.c 组成。向上给应用程序提供完善的设备文件操作接口(即对 FrameBuffer 设备进行 read、write、ioctl 等操作),接口在 Linux 提供的 fbmem.c 文件中实现;向下

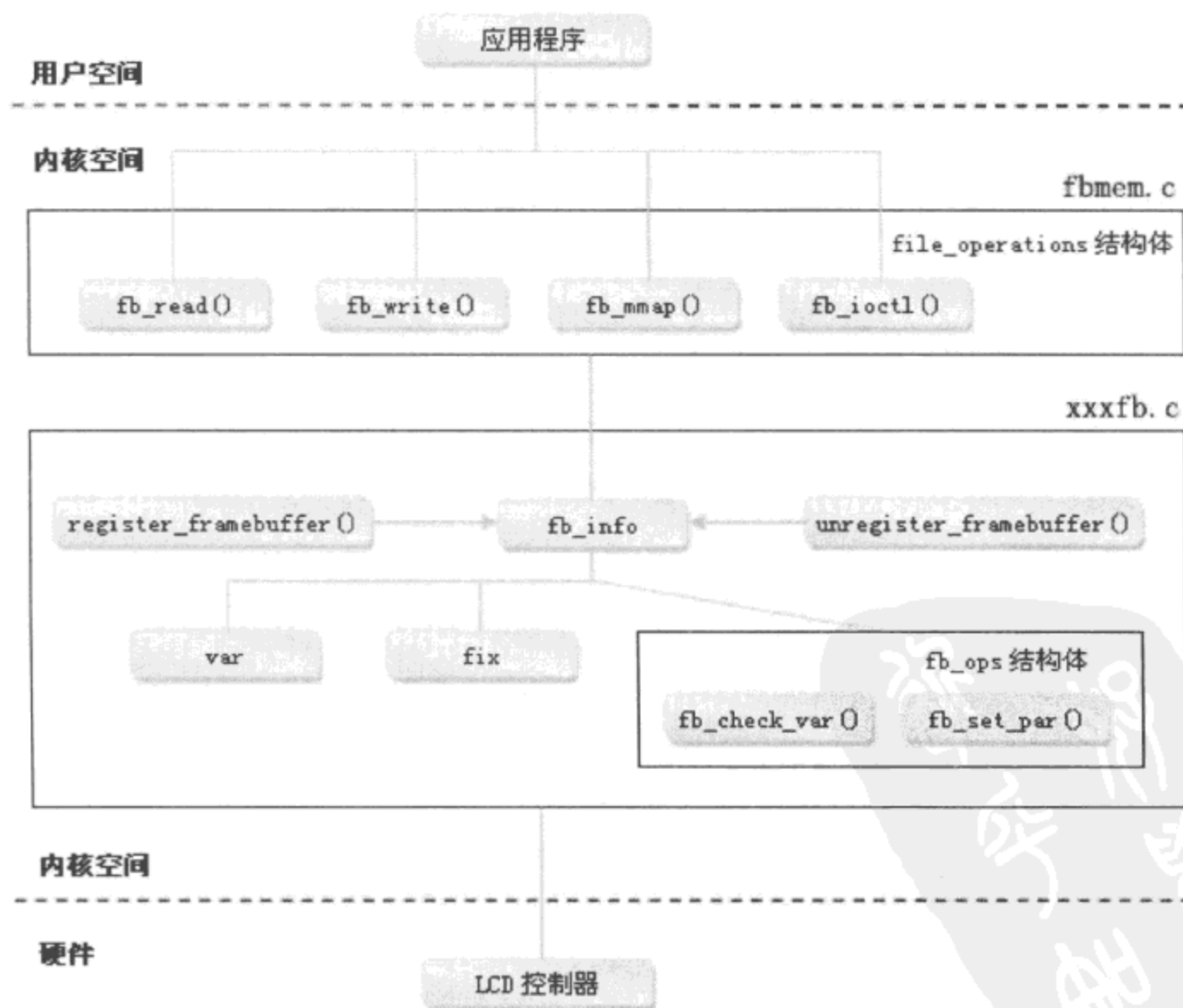


图 12-6 帧缓冲设备驱动程序结构图

提供了硬件操作的接口,只是这些接口 Linux 并没有提供实现,因为这要根据具体的 LCD 控制器硬件进行设置,所以这就是我们要做的事情了(即 xxxfb.c 部分的实现)。

2. 帧缓冲相关的重要数据结构

(1) 从帧缓冲设备驱动程序结构看,该驱动主要跟 fb_info 结构体有关,该结构体记录了帧缓冲设备的全部信息,包括设备的设置参数、状态以及对底层硬件操作的函数指针。在 Linux 中,每一个帧缓冲设备都必须对应一个 fb_info,fb_info 在 /linux/fb.h 中的定义如下:(只列出重要的一些)

```
struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var; /* LCD 可变参数结构体 */
    struct fb_fix_screeninfo fix; /* LCD 固定参数结构体 */
    struct fb_monspecs monspecs; /* LCD 显示器标准 */
    struct work_struct queue;    /* 帧缓冲事件队列 */
    struct fb_pixmap pixmap;    /* 图像硬件 mapper */
    struct fb_pixmap sprite;    /* 光标硬件 mapper */
    struct fb_cmap cmap;        /* 当前的颜色表 */
    struct fb_videomode * mode; /* 当前的显示模式 */

#ifdef CONFIG_FB_BACKLIGHT
    struct backlight_device * bl_dev; /* 对应的背光设备 */
    struct mutex bl_curve_mutex;
    u8 bl_curve[FB_BACKLIGHT_LEVELS]; /* 背光调整 */
#endif

#ifdef CONFIG_FB_DEFERRED_IO
    struct delayed_work deferred_work;
    struct fb_deferred_io * fbdefio;
#endif

    struct fb_ops * fbops; /* 对底层硬件操作的函数指针 */
    struct device * device;
    struct device * dev; /* fb 设备 */
    int class_flag;
#ifdef CONFIG_FB_TILEBLITTING
    struct fb_tile_ops * tileops; /* 图块 Blitting */
#endif

    char __iomem * screen_base; /* 虚拟基地址 */
};
```

资源分享 PDG

```

    unsigned long screen_size;    /* LCD IO 映射的虚拟内存大小 */
    void * pseudo_palette;        /* 伪 16 色颜色表 */
#define FBINFO_STATE_RUNNING    0
#define FBINFO_STATE_SUSPENDED 1
    u32 state;    /* LCD 的挂起或恢复状态 */
    void * fbcon_par;
    void * par;    /* 存放 FrameBuffer 设备的私有数据 */
};

```

其中,比较重要的成员有 struct fb_var_screeninfo var、struct fb_fix_screeninfo fix 和 struct fb_ops * fbops,他们也都是结构体。下面我们一个一个地来看。

(2) fb_var_screeninfo 结构体主要记录用户可以修改的控制器的参数,比如屏幕的分辨率和每个像素的比特数等,该结构体定义如下:

```

struct fb_var_screeninfo {
    __u32 xres;                /* 可见屏幕一行有多少个像素点 */
    __u32 yres;                /* 可见屏幕一列有多少个像素点 */
    __u32 xres_virtual;        /* 虚拟屏幕一行有多少个像素点 */
    __u32 yres_virtual;        /* 虚拟屏幕一列有多少个像素点 */
    __u32 xoffset;             /* 虚拟到可见屏幕之间的行偏移 */
    __u32 yoffset;             /* 虚拟到可见屏幕之间的列偏移 */
    __u32 bits_per_pixel;      /* 每个像素的位数即 BPP */
    __u32 grayscale;           /* 非 0 时,指的是灰度 */

    struct fb_bitfield red;     /* fb 缓存的 R 位域 */
    struct fb_bitfield green;   /* fb 缓存的 G 位域 */
    struct fb_bitfield blue;    /* fb 缓存的 B 位域 */
    struct fb_bitfield transp; /* 透明度 */

    __u32 nonstd;              /* != 0 非标准像素格式 */
    __u32 activate;
    __u32 height;              /* 高度 */
    __u32 width;               /* 宽度 */
    __u32 accel_flags;

    /* 定时:除了 pixclock 本身外,其他的都以像素时钟为单位 */
    __u32 pixclock;            /* 像素时钟(ps) */
    __u32 left_margin;         /* 行切换,从同步到绘图之间的延迟 */
    __u32 right_margin;        /* 行切换,从绘图到同步之间的延迟 */
    __u32 upper_margin;        /* 帧切换,从同步到绘图之间的延迟 */

```

```

__u32 lower_margin;      /* 帧切换,从绘图到同步之间的延迟 */
__u32 hsync_len;         /* 水平同步的长度 */
__u32 vsync_len;         /* 垂直同步的长度 */
__u32 sync;
__u32 vmode;
__u32 rotate;
__u32 reserved[5];       /* 保留 */
};

```

(3) 而 fb_fix_screeninfo 结构体又主要记录用户不可以修改的控制器的参数,比如屏幕缓冲区的物理地址和长度等,该结构体的定义如下:

```

struct fb_fix_screeninfo {
    char id[16];           /* 字符串形式的标示符 */
    unsigned long smem_start; /* fb 缓存的开始位置 */
    __u32 smem_len;        /* fb 缓存的长度 */
    __u32 type;            /* 看 FB_TYPE_ * */
    __u32 type_aux;        /* 分界 */
    __u32 visual;          /* 看 FB_VISUAL_ * */
    __u16 xpanstep;         /* 如果没有硬件 panning 就赋值为 0 */
    __u16 ypanstep;         /* 如果没有硬件 panning 就赋值为 0 */
    __u16 ywrapstep;        /* 如果没有硬件 ywrap 就赋值为 0 */
    __u32 line_length;     /* 一行的字节数 */
    unsigned long mmio_start; /* 内存映射 I/O 的开始位置 */
    __u32 mmio_len;        /* 内存映射 I/O 的长度 */
    __u32 accel;
    __u16 reserved[3];     /* 保留 */
};

```

(4) fb_ops 结构体是对底层硬件操作的函数指针,该结构体中定义了对硬件的操作有:(这里只列出了常用的操作)

```

struct fb_ops {

    struct module * owner;

    //检查可变参数并进行设置
    int (* fb_check_var)(struct fb_var_screeninfo * var, struct fb_info * info);

    //根据设置的值进行更新,使之有效
    int (* fb_set_par)(struct fb_info * info);

```



```

//设置颜色寄存器
int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
                    unsigned blue, unsigned transp, struct fb_info * info);

//显示空白
int (*fb_blank)(int blank, struct fb_info * info);

//矩形填充
void (*fb_fillrect)(struct fb_info * info, const struct fb_fillrect * rect);

//复制数据
void (*fb_copyarea)(struct fb_info * info, const struct fb_copyarea * region);

//图形填充
void (*fb_imageblit)(struct fb_info * info, const struct fb_image * image);
};

```

3. 帧缓冲设备作为平台设备

在 S3C2440 中, LCD 控制器被集成在芯片的内部作为一个相对独立的单元, 所以 Linux 把它看做是一个平台设备, 故在驱动的 mys3c2410fbdata.h 中定义有 LCD 相关的平台设备及资源, 代码如下:

```

static struct resource mys3c_lcd_resource[] = {
    [0] = {
        .start = S3C24XX_PA_LCD,
        .end   = S3C24XX_PA_LCD + S3C24XX_SZ_LCD - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = IRQ_LCD,
        .end   = IRQ_LCD,
        .flags = IORESOURCE_IRQ,
    }
};

static u64 mys3c_device_lcd_dmamask = 0xffffffffUL;

struct platform_device mys3c_device_lcd = {

```




```

.name           = "mys3c2410 - lcd",
.id             = -1,
.num_resources  = ARRAY_SIZE(mys3c_lcd_resource),
.resource       = mys3c_lcd_resource,
.dev            = {
    .dma_mask     = &mys3c_device_lcd_dmamask,
    .coherent_dma_mask = 0xffffffffUL
}
};

static struct platform_device *display_devices[] = {
    &mys3c_device_lcd,
};

```

除此之外,在驱动的 mys3c2410fbdata.h 文件中为 LCD 平台设备定义了一个 s3c2410fb_mach_info 结构体,该结构体主要是记录 LCD 的硬件参数信息(比如该结构体的 s3c2410fb_display 成员结构中就用于记录 LCD 的屏幕尺寸、屏幕信息、可变的屏幕参数、LCD 配置寄存器等),这样在写驱动的时候就直接使用这个结构体。

//LCD 硬件的配置信息,注意这里使用的 LCD 是统宝 3.5 寸 TFT 屏,这些参数要根据具体的 LCD 屏进行
//设置

```

#define LCD_WIDTH 240
#define LCD_HEIGHT 320
#define LCD_PIXCLOCK 170000

#define LCD_RIGHT_MARGIN 25
#define LCD_LEFT_MARGIN 0
#define LCD_HSYNC_LEN 4

#define LCD_UPPER_MARGIN 1
#define LCD_LOWER_MARGIN 4
#define LCD_VSYNC_LEN 1

```

```
static struct s3c2410fb_display my2440_lcd_cfg __initdata = {
```

//这个地方的设置是配置 LCD 寄存器 5,这些宏定义在 regs - lcd.h 中,计算后二进制为:
//111111111111,然后对照数据手册上 LCDCON5 的各位来看,注意是从右边开始

```

.lcdcon5      = S3C2410_LCDCON5_FRM565 | //设置为 16BPP 的 5:6:5 模式
                S3C2410_LCDCON5_INVVLINE |
                S3C2410_LCDCON5_INVVFRAME |
                S3C2410_LCDCON5_PWREN |

```



S3C2410_LCDCON5_HWSWP,

```
.type          = S3C2410_LCDCON1_TFT, // TFT 类型

.width         = LCD_WIDTH,
.height        = LCD_HEIGHT,

.pixclock      = LCD_PIXCLOCK, // 像素时钟 170000ps
.xres          = LCD_WIDTH,
.yres          = LCD_HEIGHT,
.bpp           = 16, // 色位模式 16BPP
.left_margin   = LCD_LEFT_MARGIN + 1,
.right_margin  = LCD_RIGHT_MARGIN + 1,
.hsync_len     = LCD_HSYNC_LEN + 1,
.upper_margin  = LCD_UPPER_MARGIN + 1,
.lower_margin  = LCD_LOWER_MARGIN + 1,
.vsync_len     = LCD_VSYNC_LEN + 1,
};
```

```
static struct s3c2410fb_mach_info my2440_fb_info __initdata = {
    .displays      = &my2440_lcd_cfg, //应用上面定义的配置信息
    .num_displays  = 1,
    .default_display = 0,
```

/* 配置 GPIO 引脚用于 LCD 屏 */

```
.gpcccon = 0xaa955699,
.gpcccon_mask = 0xffc003cc,
.gpcup = 0x0000ffff,
.gpcup_mask = 0xffffffff,

.gpdcon = 0xaa95aaa1,
.gpdcon_mask = 0xffc0fff0,
.gpdup = 0x0000faff,
.gpdup_mask = 0xffffffff,
```

```
.lpcsel      = 0xf82, // 这个是三星 TFT 屏的参数,这里不用
};
```

下面来看一下驱动是如果使用这个结构体的。在驱动的 `mys3c2410fb.c` 中定义有：

```

1106 static int __init lcd_init(void)
1107 {
1108     /* 在 Linux 中,帧缓冲设备被看做是平台设备,所以这里注册平台设备 */
1109     mys3c24xx_fb_set_platdata(&my2440_fb_info);
1110     platform_add_devices(display_devices, ARRAY_SIZE(display_devices));
1111     return platform_driver_register(&lcd_fb_driver);
1112 }

```

mys3c24xx_fb_set_platdata 定义在驱动的 mys3c2410fbdata.h 中:

```

146 /* set platform data in platform device -- LCD */
147 void mys3c24xx_fb_set_platdata(struct s3c2410fb_mach_info * pd)
148 {
149     struct s3c2410fb_mach_info * npd;
150     npd = kmalloc(sizeof(* npd), GFP_KERNEL);
151     if (npd) {
152         memcpy(npd, pd, sizeof(* npd));

```

//这里就是将前面中定义的 s3c2410fb_mach_info 结构体数据保存到 LCD 平台数据中,所以//在写驱动的时候就可以直接在平台数据中获取 s3c2410fb_mach_info 结构体的数据(即 LCD//各种参数信息)进行操作

```

153         mys3c_device_lcd.dev.platform_data = npd;
154     } else {
155         printk(KERN_ERR "no memory for LCD platfrom data\n");
156     }
157 }

```

12.2.3 LCD 驱动实例代码

1. FrameBuffer 驱动的初始化和卸载部分

```

static struct platform_driver lcd_fb_driver =
{
    .probe      = lcd_fb_probe,           /* FrameBuffer 设备探测 */
    .remove     = __devexit_p(lcd_fb_remove), /* FrameBuffer 设备移除 */
    .suspend    = lcd_fb_suspend,        /* FrameBuffer 设备挂起 */
    .resume     = lcd_fb_resume,         /* FrameBuffer 设备恢复 */
    .driver     =
    {

```

/* 注意这里的名称一定要和系统中定义平台设备的地方一致,这样才能把平台设备与该平台设备的驱动关联起来 */

```

        .name = "mys3c2410 - lcd",

```

```

        .owner = THIS_MODULE,
    },
};

static int __init lcd_init(void)
{
    /* 在 Linux 中,帧缓冲设备被看做是平台设备,所以这里注册平台设备 */
    mys3c24xx_fb_set_platdata(&my2440_fb_info);
    platform_add_devices(display_devices, ARRAY_SIZE(display_devices));
    return platform_driver_register(&lcd_fb_driver);
}

static void __exit lcd_exit(void)
{
    /* 注销平台设备 */
    platform_driver_unregister(&lcd_fb_driver);
    platform_device_unregister(display_devices[0]);
    kfree(mys3c_device_lcd.dev.platform_data);
}

module_init	lcd_init);
module_exit	lcd_exit);

```

platform_driver_register 注册平台驱动时,会导致 LCD 平台设备 probe 函数(lcd_fb_probe)被调用。

2. LCD 平台设备 probe 函数的实现

```

struct my2440fb_var
{
    int lcd_irq_no;           /* 保存 LCD 中断号 */
    struct clk * lcd_clock;    /* 保存从平台时钟队列中获取的 LCD 时钟 */
    struct resource * lcd_mem; /* LCD 的 I/O 空间 */
    void __iomem * lcd_base;   /* LCD 的 I/O 空间映射到虚拟地址 */
    struct device * dev;
    struct s3c2410fb_hw regs; /* 表示 5 个 LCD 配置寄存器,s3c2410fb_hw 定义在 mys3c2410fbdata.h 中 */

    /* 定义一个数组来充当调色板。
    据数据手册描述,TFT 屏色位模式为 8BPP 时,调色板(颜色表)的长度为 256,调色板起始地址为 0x4D000400 */
    u32 palette_buffer[256];

```

```

u32 pseudo_pal[16];
unsigned int palette_ready; /* 标识调色板是否准备好了 */
};

static int __devinit lcd_fb_probe(struct platform_device *pdev)
{
    int i;
    int ret;
    struct resource *res; /* 用来保存从 LCD 平台设备中获取的 LCD 资源 */
    struct fb_info *fbinfo; /* FrameBuffer 驱动所对应的 fb_info 结构体 */
    struct s3c2410fb_mach_info *mach_info; /* 保存从内核中获取的平台设备数据 */
    struct my2440fb_var *fbvar; /* 上面定义的驱动程序全局变量结构体 */
    struct s3c2410fb_display *display; /* LCD 屏的配置信息结构体, 该结构体定义在
mys3c2410fbdata.h 中 */

    /* 获取 LCD 硬件相关信息数据, 在前面讲过驱动使用 s3c24xx_fb_set_platdata 函数将 LCD 的硬
件相关信息保存到了 LCD 平台数据中, 所以这里就从平台数据中取出来在驱动中使用 */
    mach_info = pdev->dev.platform_data;

    /* 获得在驱动中定义的 FrameBuffer 平台设备的 LCD 配置信息结构体数据 */
    display = mach_info->displays + mach_info->default_display;

    /* 给 fb_info 分配空间, 私有数据的大小为 my2440fb_var 结构的大小, 由 fbinfo->par 指向该
私有数据。framebuffer_alloc 定义在 fb.h 中在 fbsysfs.c 中实现 */
    fbinfo = framebuffer_alloc(sizeof(struct my2440fb_var), &pdev->dev);

    platform_set_drvdata(pdev, fbinfo); /* 重新将 LCD 平台设备数据设置为 fbinfo, 好在后面的一
些函数中来使用 */

    /* 这里的用途其实就是将 fb_info 的成员 par(注意是一个 void 类型的指针)指向这里的私有变
量结构体 fbvar, 目的是到其他接口函数中再取出 fb_info 的成员 par, 从而能继续使用这里的私有变量 */
    fbvar = fbinfo->par;
    fbvar->dev = &pdev->dev;

    /* 在系统定义的 LCD 平台设备资源中获取 LCD 中断号, platform_get_irq 定义在 platform_de
vice.h 中 */
    fbvar->lcd_irq_no = platform_get_irq(pdev, 0);

    /* 获取 LCD 平台设备所使用的 I/O 端口资源, 注意这个 IORESOURCE_MEM 标志和 LCD 平台设备定义

```

中的一致 */

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

/* 申请 LCD I/O 端口所占用的 I/O 空间 */

```
fbvar->lcd_mem = request_mem_region(res->start, res->end - res->start + 1, pdev->name);
```

/* 将 LCD 的 I/O 端口占用的这段 I/O 空间映射到内存的虚拟地址, ioremap 定义在 io.h 中。注意: I/O 空间要映射后才能使用, 以后对虚拟地址的操作就是对 I/O 空间的操作 */

```
fbvar->lcd_base = ioremap(res->start, res->end - res->start + 1);
```

/* 从平台时钟队列中获取 LCD 的时钟, 这里为什么要取得这个时钟, 从 LCD 屏的时序图上看, 各种控制信号的延迟都跟 LCD 的时钟有关。系统的一些时钟定义在 arch/arm/plat-s3c24xx/s3c2410-clock.c 中 */

```
fbvar->lcd_clock = clk_get(NULL, "lcd");
```

/* 时钟获取后要使能后才可以使用, clk_enable 定义在 arch/arm/plat-s3c/clock.c 中 */

```
clk_enable(fbvar->lcd_clock);
```

/* 申请 LCD 中断服务 */

```
ret = request_irq(fbvar->lcd_irq_no, lcd_fb_irq, IRQF_DISABLED, pdev->name, fbvar);
```

/* 好了, 以上是对要使用的资源进行了获取和设置。下面就开始初始化填充 fb_info 结构体 */

/* 首先初始化 fb_info 中代表 LCD 固定参数的结构体 fb_fix_screeninfo */

/* 像素值与显示内存的映射关系有 5 种, 定义在 fb.h 中。现在采用 FB_TYPE_PACKED_PIXELS 方式, 在该方式下, 像素值与内存直接对应, 比如在显示内存某单元写入一个“1”时, 该单元对应的像素值也将是“1”, 这使得应用层把显示内存映射到用户空间变得非常方便。Linux 中当 LCD 为 TFT 屏时, 显示驱动管理显示内存就是基于这种方式 */

```
strcpy(fbinfo->fix.id, driver_name); /* 字符串形式的标识符 */
```

```
fbinfo->fix.type = FB_TYPE_PACKED_PIXELS;
```

fbinfo->fix.type_aux = 0; /* 以下这些根据 fb_fix_screeninfo 定义中的描述, 当没有硬件时都设为 0 */

```
fbinfo->fix.xpanstep = 0;
```

```
fbinfo->fix.ypanstep = 0;
```

```
fbinfo->fix.ywrapstep = 0;
```

```
fbinfo->fix.accel = FB_ACCEL_NONE;
```

/* 接着, 再初始化 fb_info 中代表 LCD 可变参数的结构体 fb_var_screeninfo */


```
fbinfo->var.nonstd      = 0;
fbinfo->var.activate     = FB_ACTIVATE_NOW;
fbinfo->var.accel_flags  = 0;
fbinfo->var.vmode        = FB_VMODE_NONINTERLACED;
fbinfo->var.xres         = display->xres;
fbinfo->var.yres         = display->yres;
fbinfo->var.bits_per_pixel = display->bpp;
```

/* 指定对底层硬件操作的函数指针, 参见 3 的说明 */

```
fbinfo->fbops           = &my2440fb_ops;
```

```
fbinfo->flags           = FBINFO_FLAG_DEFAULT;
```

```
fbinfo->pseudo_palette  = &fbvar->pseudo_pal;
```

/* 初始化色调色板(颜色表)为空 */

```
for(i = 0; i < 256; i++)
```

```
{
```

```
    fbvar->palette_buffer[i] = PALETTE_BUFF_CLEAR;
```

```
}
```

```
for (i = 0; i < mach_info->num_displays; i++) /* fb 缓存的长度 */
```

```
{
```

/* 计算 FrameBuffer 缓存的最大大小, 这里右移 3 位(即除以 8)是因为色位模式 BPP 是以位为单位 */

```
    unsigned long smem_len = (mach_info->displays[i].xres * mach_info->displays[i].
yres * mach_info->displays[i].bpp) >> 3;
```

```
    if(fbinfo->fix.smem_len < smem_len)
```

```
{
```

```
        fbinfo->fix.smem_len = smem_len;
```

```
}
```

```
}
```

/* 初始化 LCD 控制器之前要延迟一段时间 */

```
msleep(1);
```

/* 初始化完 fb_info 后, 开始对 GPIO 寄存器进行初始化, 其定义在后面讲到 */

```
my2440fb_init_registers(fbinfo); //仅仅需要做一次即可
```

资源解密

PDG

/* 初始化完 GPIO 寄存器后,开始检查 fb_info->var 与 fbinfo 支持的哪一种分辨率、色彩模式相匹配,并据此填充 fb_info->var 中的其他可变参数,其定义在后面讲到 */

```
my2440fb_check_var(&fbinfo->var, fbinfo);
```

/* 申请帧缓冲设备 fb_info 的显示缓冲区空间,并将其地址填写进 fbinfo 中。其定义在后面讲到 */

```
ret = my2440fb_map_video_memory(fbinfo);
```

/* 最后,注册这个帧缓冲设备 fb_info 到系统中。register_framebuffer 定义在 fb.h 中,在 fbmem.c 中实现,该内核 API 会回调 fbinfo->fbops.fb_set_par = my2440fb_set_par(fbinfo) 函数,此函数会根据 fbinfo->var 激活 fb_info 中的参数配置(即:使修改后的参数在硬件上生效) */

```
ret = register_framebuffer(fbinfo);
```

/* 对设备文件系统的支持。创建 framebuffer 设备文件,device_create_file 定义在 linux/device.h 中 */

```
ret = device_create_file(&pdev->dev, &dev_attr_debug);
```

```
return 0;
```

```
}
```

/* GPIO 寄存器进行初始化 */

```
static int my2440fb_init_registers(struct fb_info * fbinfo)
```

```
{
```

/* 从 lcd_fb_probe 探测函数设置的私有变量结构体中再获得 LCD 相关信息的数据 */

```
struct my2440fb_var * fbvar = fbinfo->par;
```

```
struct s3c2410fb_mach_info * mach_info = fbvar->dev->platform_data;
```

/* 这里就是把 GPIO 端口 C 和 D 配置成 LCD 模式 */

```
modify_gpio(S3C2410_GPCUP, mach_info->gpcup, mach_info->gpcup_mask);
```

```
modify_gpio(S3C2410_GPCCON, mach_info->gpcccon, mach_info->gpcccon_mask);
```

```
modify_gpio(S3C2410_GPDUP, mach_info->gpdup, mach_info->gpdup_mask);
```

```
modify_gpio(S3C2410_GPDCON, mach_info->gpdcon, mach_info->gpdcon_mask);
```

```
}
```

/* 该函数实现修改 GPIO 端口的值,注意第三个参数 mask 的作用是将要设置的寄存器值先清零 */

```
static inline void modify_gpio(void __iomem * reg, unsigned long set, unsigned long mask)
```

```
{
```

```
unsigned long tmp;
```

```
tmp = readl(reg) & ~mask;
```

```

writel(tmp | set, reg);
}

/* 检查 var 与 fbinfo 支持的哪一种分辨率、色彩模式相匹配,并据此填充 var 中的其他可变参数,*/
static int my2440fb_check_var(struct fb_var_screeninfo * var, struct fb_info * fbinfo)
{
    unsigned i;

    /* 从 lcd_fb_probe 探测函数设置的平台数据中再获得 LCD 相关信息的数据 */
    struct my2440fb_var * fbvar = fbinfo->par; /* 在 lcd_fb_probe 探测函数中设置的私有
    结构体数据 */
    struct s3c2410fb_mach_info * mach_info = fbvar->dev->platform_data; /* LCD 的配置结构
    体数据 */

    struct s3c2410fb_display * display = NULL;
    struct s3c2410fb_display * default_display = mach_info->displays + mach_info->de-
    fault_display;
    int type = default_display->type; /* LCD 的类型,type 赋值是 TFT 类型 */

    /* 验证 X/Y 解析度 */
    if (var->yres == default_display->yres &&
        var->xres == default_display->xres &&
        var->bits_per_pixel == default_display->bpp)
    {
        display = default_display;
    }
    else
    {
        for (i = 0; i < mach_info->num_displays; i++)
        {
            if (type == mach_info->displays[i].type &&
                var->yres == mach_info->displays[i].yres &&
                var->xres == mach_info->displays[i].xres &&
                var->bits_per_pixel == mach_info->displays[i].bpp)
            {
                display = mach_info->displays + i;
                break;
            }
        }
    }
}

```

```
}

if (! display)
{
    return -EINVAL;
}

/* 配置 LCD 配置寄存器 1 中的 5~6 位(配置成 TFT 类型)和配置 LCD 配置寄存器 5 */
fbvar->regs.lcdcon1 = display->type;
fbvar->regs.lcdcon5 = display->lcdcon5;

/* 设置屏幕的虚拟解析像素和高度宽度 */
var->xres_virtual = display->xres;
var->yres_virtual = display->yres;
var->height = display->height;
var->width = display->width;

/* 设置时钟像素,行、帧切换值,水平同步、垂直同步长度值 */
var->pixclock = display->pixclock;
var->left_margin = display->left_margin;
var->right_margin = display->right_margin;
var->upper_margin = display->upper_margin;
var->lower_margin = display->lower_margin;
var->vsync_len = display->vsync_len;
var->hsync_len = display->hsync_len;

/* 设置透明度 */
var->transp.offset = 0;
var->transp.length = 0;

/* 根据色位模式(BPP)来设置可变参数中 R、G、B 的颜色位域。 */
switch (var->bits_per_pixel)
{
    case 16: /* 16 bpp */
        if (display->lcdcon5 & S3C2410_LCDCON5_FRM565)
        {
            /* 565 format */
            var->red.offset = 11;
            var->green.offset = 5;
```

```

        var->blue.offset      = 0;
        var->red.length       = 5;
        var->green.length     = 6;
        var->blue.length      = 5;
    } else {
        /* 5551 format */
        var->red.offset        = 11;
        var->green.offset      = 6;
        var->blue.offset       = 1;
        var->red.length        = 5;
        var->green.length      = 5;
        var->blue.length       = 5;
    }
    break;
case 32: /* 24 bpp 888 and 8 dummy */
    var->red.length           = 8;
    var->red.offset           = 16;
    var->green.length         = 8;
    var->green.offset         = 8;
    var->blue.length          = 8;
    var->blue.offset          = 0;
    break;
}
return 0;
}

```

/* 申请帧缓冲设备 fb_info 的显示缓冲区空间 */

static int __init my2440fb_map_video_memory(struct fb_info *fbinfo)

{

 dma_addr_t map_dma; /* 用于保存 DMA 缓冲区总线地址 */

 struct my2440fb_var *fbvar = fbinfo->par; /* 获得在 lcd_fb_probe 探测函数中设置的私有结构体数据 */

 unsigned map_size = PAGE_ALIGN(fbinfo->fix.smem_len); /* 获得 FrameBuffer 缓存的大小, PAGE_ALIGN 定义在 mm.h 中 */

 /* 将分配的一个写合并 DMA 缓存区设置为 LCD 屏幕的虚拟地址(对于 DMA 请参考 DMA 相关知识)

 dma_alloc_writecombine 定义在 arch/arm/mm/dma-mapping.c 中 */

 fbinfo->screen_base = dma_alloc_writecombine(fbvar->dev, map_size, &map_dma, GFP_KERNEL);

```

    if (fbinfo->screen_base)
    {
        /* 设置这片 DMA 缓存区的内容为空 */
        memset(fbinfo->screen_base, 0x00, map_size);

        /* 将 DMA 缓冲区总线地址设成 fb_info 不可变参数中 framebuffer 缓存的开始位置 */
        fbinfo->fix.smem_start = map_dma;
    }
    return fbinfo->screen_base ? 0 : - ENOMEM;
}

```

3. 帧缓冲设备驱动对底层硬件操作的函数接口实现(即:my2440fb_ops 的实现)

```

/* Framebuffer 底层硬件操作各接口函数 */
static struct fb_ops my2440fb_ops =
{
    .owner          = THIS_MODULE,
    .fb_check_var   = my2440fb_check_var, /* 第二步中已实现 */
    .fb_set_par     = my2440fb_set_par, /* 激活 fb_info 中的参数配置(即:使修改后的参数在硬
件上生效) */
    .fb_blank       = my2440fb_blank, /* 显示空白(即:LCD 开关控制) */
    .fb_setcolreg   = my2440fb_setcolreg, /* 设置颜色表 */
    /* 以下三个函数是可选的,主要是提供 fb_console 的支持,在内核中已经实现,这里直接调用即可 */
    .fb_fillrect    = cfb_fillrect, /* 定义在 drivers/video/cfbfillrect.c 中 */
    .fb_copyarea    = cfb_copyarea, /* 定义在 drivers/video/cfbcopyarea.c 中 */
    .fb_imageblit   = cfb_imageblit, /* 定义在 drivers/video/cfbimgblt.c 中 */
};

/* 显示空白,blank mode 有 5 种模式,定义在 fb.h 中,是一个枚举 */
static int my2440fb_blank(int blank_mode, struct fb_info * fbinfo)
{
    struct my2440fb_var * fbvar = fbinfo->par;
    void __iomem * regs = fbvar->lcd_base;

    /* 根据显示空白的模式来设置 LCD 是开启还是停止 */
    if (blank_mode == FB_BLANK_POWERDOWN)
    {
        my2440fb_lcd_enable(fbvar, 0); /* 停止 LCD */
    }
}

```



```

else
{
    my2440fb_lcd_enable(fbvar, 1); /* 启动 LCD */
}

/* 根据显示空白的模式来控制临时调色板寄存器 */
if (blank_mode == FB_BLANK_UNBLANK)
{
    /* 临时调色板寄存器无效 */
    writel(0x0, regs + S3C2410_TPAL);
}
else
{
    /* 临时调色板寄存器有效 */
    writel(S3C2410_TPAL_EN, regs + S3C2410_TPAL);
}
return 0;
}

/* 启动(或停止)LCD 控制器的工作 */
static void my2440fb_lcd_enable(struct my2440fb_var * fbvar, int enable)
{
    if (enable)
    {
        fbvar->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID;
    }
    else
    {
        fbvar->regs.lcdcon1 &= ~S3C2410_LCDCON1_ENVID;
    }
    writel(fbvar->regs.lcdcon1, fbvar->lcd_base + S3C2410_LCDCON1);
}

/* 根据 fbinfo->var 激活 fb_info 中的参数配置(即:使修改后的参数在硬件上生效) */
static int my2440fb_set_par(struct fb_info * fbinfo)
{
    /* 获得 fb_info 中的可变参数 */
    struct fb_var_screeninfo * var = &fbinfo->var;

```

```

/* 判断可变参数中的色位模式,根据色位模式来设置色彩模式 */
switch (var->bits_per_pixel)
{
    case 32:
    case 16:
    case 12: /* 12BPP 时,设置为真彩色(分成红、绿、蓝三基色) */
        fbinfo->fix.visual = FB_VISUAL_TRUECOLOR;
        break;
    case 1: /* 1BPP 时,设置为黑白色(分黑、白两种色,FB_VISUAL_MONO01 代表黑,FB_VISUAL_MONO10 代表白) */
        fbinfo->fix.visual = FB_VISUAL_MONO01;
        break;
    default: /* 默认设置为伪彩色,采用索引颜色显示 */
        fbinfo->fix.visual = FB_VISUAL_PSEUDOCOLOR;
        break;
}

/* 设置 fb_info 中固定参数中一行的字节数,公式:1 行字节数 = (1 行像素个数 × 每像素位数 BPP)/8 */
fbinfo->fix.line_length = (var->xres_virtual * var->bits_per_pixel) / 8;

/* 修改以上参数后,重新激活 fb_info 中的参数配置(即:使修改后的参数在硬件上生效) */
my2440fb_activate_var(fbinfo);

return 0;
}

/* 重新激活 fb_info 中的参数配置 */
static void my2440fb_activate_var(struct fb_info * fbinfo)
{
    /* 获得结构体变量 */
    struct my2440fb_var * fbvar = fbinfo->par;
    void __iomem * regs = fbvar->lcd_base;

    /* 获得 fb_info 可变参数 */
    struct fb_var_screeninfo * var = &fbinfo->var;

    /* 计算 LCD 控制寄存器 1 中的 CLKVAL 值,根据数据手册中该寄存器的描述,计算公式如下:
     * TFT 屏:VCLK = HCLK / [(CLKVAL + 1) × 2], CLKVAL 要求 ≥ 0 */

```

```
int clkdiv = my2440fb_calc_pixclk(fbvar, var->pixclock) / 2;

/* 获得屏幕的类型 */
int type = fbvar->regs.lcdcon1 & S3C2410_LCDCON1_TFT;

if (type == S3C2410_LCDCON1_TFT)
{
    /* 根据数据手册按照 TFT 屏的要求配置 LCD 控制寄存器 1~5 */
    my2440fb_config_tft_lcd_regs(fbinfo, &fbvar->regs);
    -- clkdiv;
    if (clkdiv < 0)
    {
        clkdiv = 0;
    }
}
else
{
    /* 根据数据手册按照 STN 屏的要求配置 LCD 控制寄存器 1~5 */
    my2440fb_config_stn_lcd_regs(fbinfo, &fbvar->regs);

    if (clkdiv < 2)
    {
        clkdiv = 2;
    }
}

/* 设置计算的 LCD 控制寄存器 1 中的 CLKVAL 值 */
fbvar->regs.lcdcon1 |= S3C2410_LCDCON1_CLKVAL(clkdiv);

/* 将各参数值写入 LCD 控制寄存器 1~5 中 */
writel(fbvar->regs.lcdcon1 & ~S3C2410_LCDCON1_ENVID, regs + S3C2410_LCDCON1);
writel(fbvar->regs.lcdcon2, regs + S3C2410_LCDCON2);
writel(fbvar->regs.lcdcon3, regs + S3C2410_LCDCON3);
writel(fbvar->regs.lcdcon4, regs + S3C2410_LCDCON4);
writel(fbvar->regs.lcdcon5, regs + S3C2410_LCDCON5);

/* 配置帧缓冲起始地址寄存器 1~3 */
my2440fb_set_lcdaddr(fbinfo);
```



```

    fbvar->regs.lcdcon1 |= S3C2410_LCDCON1_ENVID,
    writel(fbvar->regs.lcdcon1, regs + S3C2410_LCDCON1);
}

/* 根据数据手册按照 TFT 屏的要求计算 LCD 控制寄存器 1~5 */
static void my2440fb_config_tft_lcd_regs(const struct fb_info * fbinfo, struct s3c2410fb_hw *
regs)
{
    const struct my2440fb_var * fbvar = fbinfo->par;
    const struct fb_var_screeninfo * var = &fbinfo->var;

    /* 根据色位模式计算 LCD 控制寄存器 1 和 5, 参考数据手册 */
    switch (var->bits_per_pixel)
    {
        case 1: /* 1BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT1BPP;
            break;
        case 2: /* 2BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT2BPP;
            break;
        case 4: /* 4BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT4BPP;
            break;
        case 8: /* 8BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT8BPP;
            regs->lcdcon5 |= S3C2410_LCDCON5_BSWP | S3C2410_LCDCON5_FRM565;
            regs->lcdcon5 &= ~S3C2410_LCDCON5_HSWP;
            break;
        case 16: /* 16BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT16BPP;
            regs->lcdcon5 &= ~S3C2410_LCDCON5_BSWP;
            regs->lcdcon5 |= S3C2410_LCDCON5_HSWP;
            break;
        case 32: /* 32BPP */
            regs->lcdcon1 |= S3C2410_LCDCON1_TFT24BPP;
            regs->lcdcon5 &= ~(S3C2410_LCDCON5_BSWP | S3C2410_LCDCON5_HSWP | S3C2410_
LCDCON5_BPP24BL);
            break;
        default: /* 无效的 BPP */

```

```

    dev_err(fbvar->dev, "invalid bpp %d\n", var->bits_per_pixel);
}

/* 计算 LCD 配置寄存器 2、3、4 */
regs->lcdcon2 = S3C2410_LCDCON2_LINEVAL(var->yres - 1) |
    S3C2410_LCDCON2_VBPD(var->upper_margin - 1) |
    S3C2410_LCDCON2_VFPD(var->lower_margin - 1) |
    S3C2410_LCDCON2_VSPW(var->vsync_len - 1);

regs->lcdcon3 = S3C2410_LCDCON3_HBPD(var->right_margin - 1) |
    S3C2410_LCDCON3_HFPD(var->left_margin - 1) |
    S3C2410_LCDCON3_HOZVAL(var->xres - 1);

regs->lcdcon4 = S3C2410_LCDCON4_HSPW(var->hsync_len - 1);
}

/* 配置帧缓冲起始地址寄存器 1-3, 参考数据手册 */
static void my2440fb_set_lcdaddr(struct fb_info *fbinfo)
{
    unsigned long saddr1, saddr2, saddr3;
    struct my2440fb_var *fbvar = fbinfo->par;
    void __iomem *regs = fbvar->lcd_base;

    saddr1 = fbinfo->fix.smem_start >> 1;
    saddr2 = fbinfo->fix.smem_start;
    saddr2 += fbinfo->fix.line_length * fbinfo->var.yres;
    saddr2 >>= 1;
    saddr3 = S3C2410_OFFSIZE(0) | S3C2410_PAGEWIDTH((fbinfo->fix.line_length / 2) & 0x3ff);

    writel(saddr1, regs + S3C2410_LCDSADDR1);
    writel(saddr2, regs + S3C2410_LCDSADDR2);
    writel(saddr3, regs + S3C2410_LCDSADDR3);
}

```

4. 其他函数的实现

中断处理函数 `lcd_fb_irq`, 以及 `my2440fb_setcolreg`、`my2440fb_write_palette`、`schedule_palette_update` 都与调色板有关, 有兴趣的读者可以自行学习、分析该代码。

12.2.4 LCD 驱动代码的主干结构的总结

lcd_init ——>间接

lcd_fb_probe ——>

分配 fb_info 结构体空间,并初始化 fb_info 结构体中的各参数

初始化 GPIO 控制器 my2440fb_init_registers(fbinfo) ——>

modify_gpio

检查并设置 fb_info 中可变参数 my2440fb_check_var(&fbinfo->var, fbinfo)

申请帧缓冲设备的显示缓冲区空间 my2440fb_map_video_memory(fbinfo)

注册 framebuffer 设备 register_framebuffer(fbinfo) ——>间接

my2440fb_set_par(fbinfo)

my2440fb_set_par(fbinfo) ——>

my2440fb_activate_var(fbinfo) ——>

计算 clkval my2440fb_calc_pixclk

计算 lcdcon1/2/3/4/5 的值 my2440fb_config_tft_lcd_regs

设置 LCD 寄存器 LCDCON1/2/3/4/5

设置帧缓存寄存器 my2440fb_set_lcdaddr

注:——>,表示直接调用;——>间接,表示间接调用。

12.2.5 测试 LCD 驱动程序

(1) 在 make menuconfig 中,配置将 LCD 驱动编译为模块。这样会在 drivers/video 目录生成 3 个模块文件:cfbcopyarea.ko、cfbimgblt.ko、cfbfillrect.ko。

(2) 在开发板上使用 insmod 命令将上述 3 个模块文件加载到内核中。

(3) 将光盘中提供的 LCD 驱动(位于\work\studydriver\lcd\linuxdriver\driver 目录)进行编译,得到 mys3c2410fb.ko。在开发板上使用 insmod 命令将该驱动加载到内核中。

(4) 将光盘中提供的 LCD 测试程序(\work\studydriver\lcd\linuxdriver 目录)进行编译。在开发板上运行该测试程序,你将会看到在 LCD 屏上显示同心圆。

```
# ./fbtest /dev/fb0
```

(5) 在开发板上执行下面的命令,将启动 qtopia,显示图形界面:

```
# qtopia &
```


12.3 触摸屏驱动

12.3.1 触摸屏裸机驱动

1. S3C2440 内部触摸屏控制器硬件结构

图 12-7 的 XP、XM、YP、YM 这 4 个引脚另一端接在物理的触摸屏上,当有人在触摸屏上按下触笔时,这 4 个引脚上会产生不同的电压值(根据按下的位置坐标不同而不同),这样触摸屏控制器就能检测到这种变化,从而产生 INT_TC 中断,表示触笔按下。紧接着,在得到 CPU 指示的情况下,触摸屏控制器还可以根据 4 个引脚上产生的不同电压值进行 A/D 转换,从而计算出 X 坐标和 Y 坐标的数值,并在将这两个值成功保存到其内部寄存器后,发出 INT_ADC 中断,表示坐标转换已经完成,从而软件就可以读取按下触笔的位置。

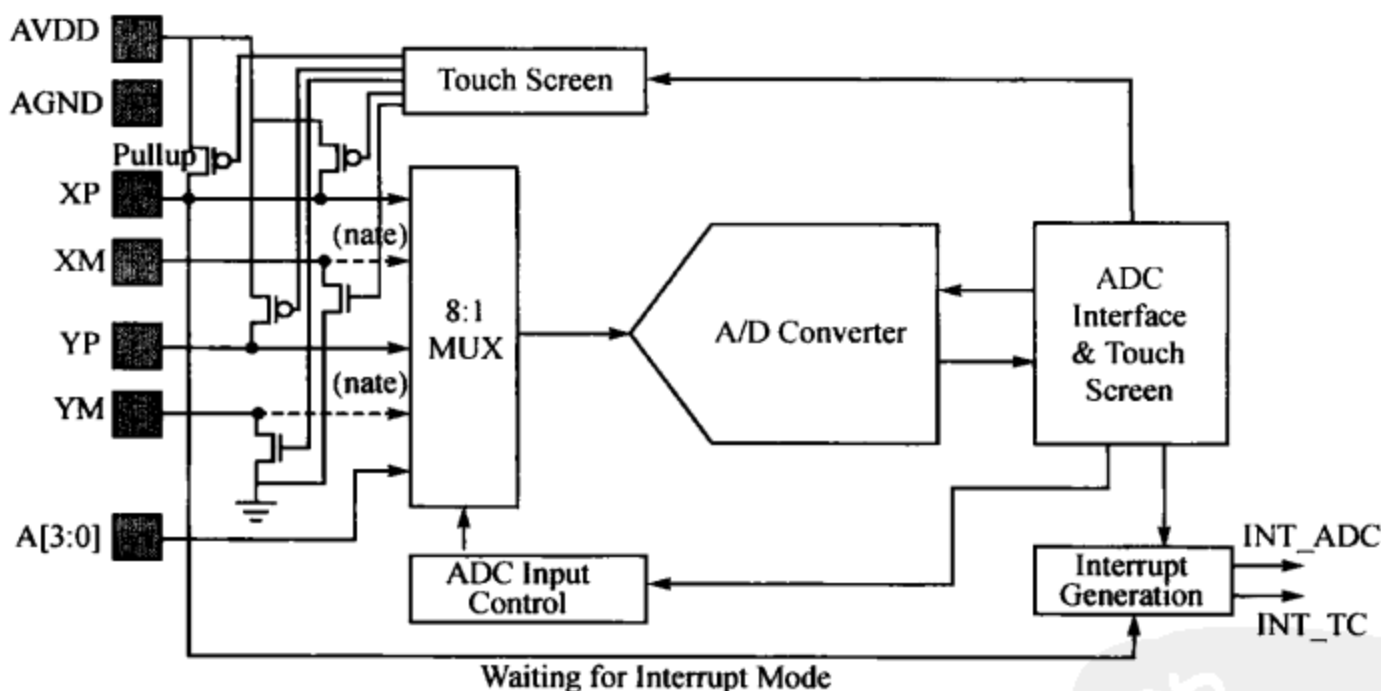


图 12-7 S3C2440 内部触摸屏控制器硬件结构

2. S3C2440 内部触摸屏控制器硬件运行机制(图 12-8)

- (1) 软件开启 INT_ADC 和 INT_TS 中断。
- (2) 软件初始化触摸屏控制器：①设置 ADCCON 寄存器,确定 A/D 转化需要的时钟频率;②设置 ADCDLY 寄存器,确定从得到命令到开始转换坐标的延时时长。
- (3) 软件设置 ADCTSC 寄存器,将触摸屏控制器设置在等待触笔按下状态。
- (4) 当触笔按下时,触摸屏控制器产生 INT_TC 中断,表示触笔按下。
- (5) 软件进入中断处理程序 Isr_Tc。Isr_Tc 首先判定中断产生的原因是触笔按下还是放开(通过查询寄存器 ADCDAT0 的 bit15),得到的答案是触笔按下,于是设置 ADCTSC 寄存

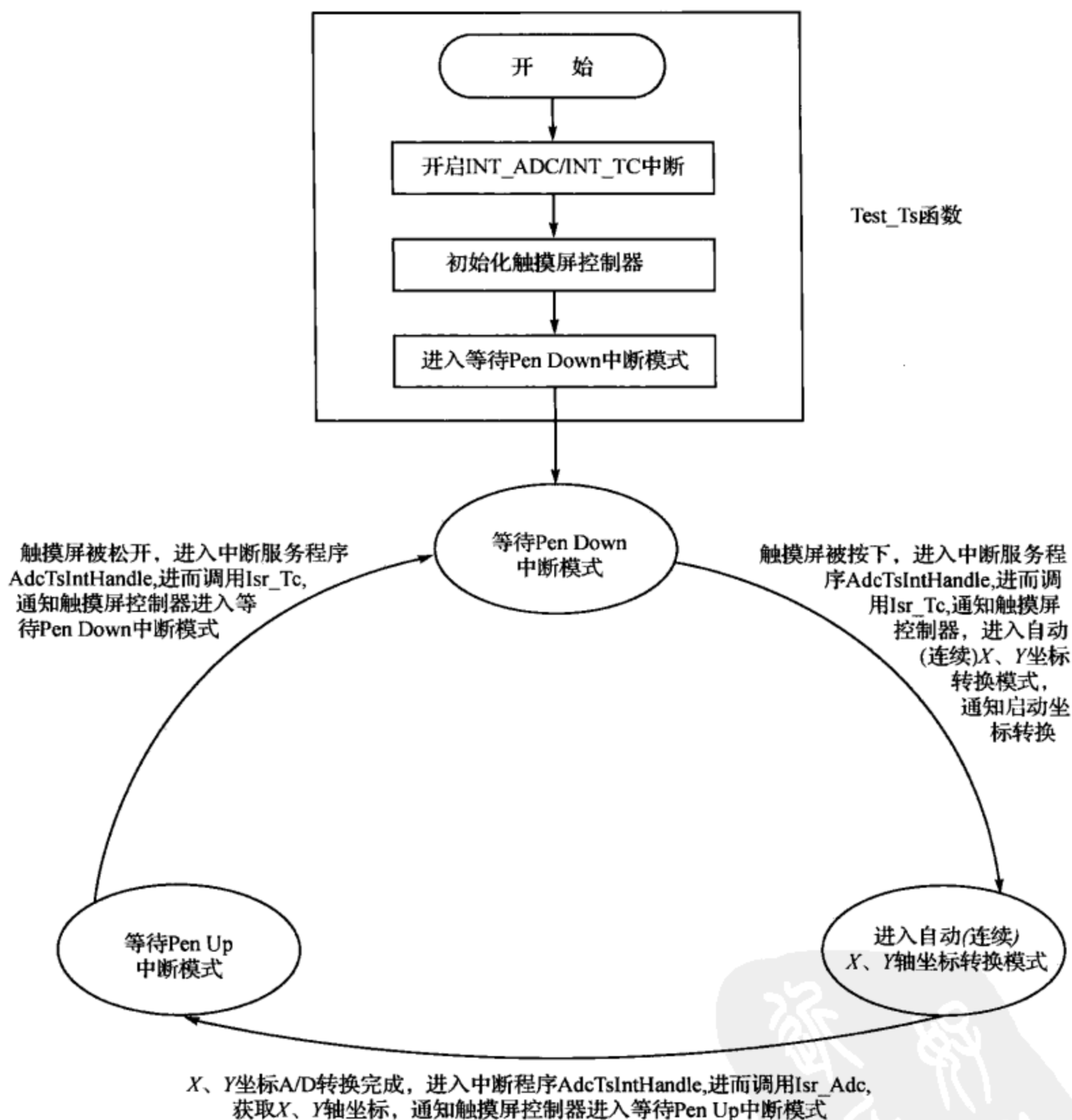


图 12-8 触摸屏控制器硬件运行机制

器,将触摸屏控制器设置在准备进行自动(连续)X/Y轴坐标转换状态,然后通知触摸屏控制器启动坐标转化(通过设置 ADCCON 的 bit0),Isr_Tc 运行结束。

(6) 触摸屏控制器在延迟指定时间后,开始转换 X/Y 坐标,并将 X 坐标保存在 ADC-DAT0 寄存器的 bit0~bit9,y 坐标保存在 ADCDAT1 寄存器的 bit0~bit9。完成后发出 INT_ADC 中断,表示转换完成,可以读取坐标值了。

(7) 软件进入中断处理程序 `Isr_Adc`。`Isr_Adc` 此时可以获取 X/Y 坐标, 然后设置 ADCTSC 寄存器, 将触摸屏控制器设置在等待触笔放开的状态, `Isr_Adc` 运行结束。

(8) 当触笔放开时, 触摸屏控制器产生 `INT_TC` 中断, 表示触笔放开。

(9) 软件进入中断处理程序 `Isr_Tc`。`Isr_Tc` 首先判定中断产生的原因是触笔按下还是放开(通过查询寄存器 `ADCDAT0` 的 `bit15`), 得到的答案是触笔放开, 于是设置 ADCTSC 寄存器, 将触摸屏控制器设置在等待触笔按下状态, `Isr_Tc` 运行结束。

(10) 回到第(4)步。

3. S3C2440 触摸屏控制器重要寄存器精解

(1) ADCCON 寄存器。

`bit[14]` 确定时候要将 PCLK 分频以得到 A/D 转化需要的时钟频率。

`bit[13:6]` 确定分频系数 PRSCVL。A/D 时钟 = $PCLK / (PRSCVL + 1)$ 。

`bit[0]` 为 1 时, 启动 A/D 转换。

(2) ADCDLY 寄存器。

`bit[15:0]` 确定得到命令到开始转换的延时值 D。延时时间为 $D \times (1/PCLK)$

(3) ADCTSC 寄存器。

`bit[8]` 为 0 表示检测触笔按下, 为 1 表示检测触笔放开。可见其作用是确定等待触笔按下还是等待触笔放开。

`bit[7:4]` 各个 bit 为 1 时, 表示使能 YM、YP、XM、XP 引脚。处于等待中断状态时, 这些位必须为 1。

`bit[3]` 为 0 表示使能上拉。处于等待中断状态时, 此位必须为 0。

`bit[2]`: 当处于准备进行自动(连续)X/Y 轴坐标转换状态, 该 bit 必须为 1; 当处于等待中断状态时, 该 bit 必须为 0。

`bit[1:0]`: 当处于准备进行自动(连续)X/Y 轴坐标转换状态, 该 bits 必须为 0b00; 当处于等待中断状态时, 该 bits 必须为 0b11。

(4) ADCDAT0 寄存器。

`bit[9:0]` 用于保存转换得到的 X 坐标, 可见 X 轴坐标的长度为 1024。

(5) ADCDAT1 寄存器。

`bit[9:0]` 用于保存转换得到的 Y 坐标, 可见 Y 轴坐标的长度为 1024。

4. 触摸屏裸机驱动实例分析

(1) 初始化。

```
void Test_Ts(void)
{
    isr_handle_array[ISR_ADC_OFT] = AdcTsIntHandle;    // 设置 ADC 中断服务程序
```

```

INTMSK &= ~BIT_ADC;           // 开启 ADC 总中断
INTSUBMSK &= ~(BIT_SUB_TC);    // 开启 INT_TC 中断,即触摸屏被按下或松开时产生中断
INTSUBMSK &= ~(BIT_SUB_ADC);   // 开启 INT_ADC 中断,即 A/D 转换结束时产生中断

// 使能预分频功能,设置 A/D 转换器的时钟 = PCLK/(49 + 1)
ADCCON = PRESCALE_EN | PRSCVL(49);

/* 设置延时时间 */
ADCDLY = 50000;

wait_down_int();    /* 进入“等待中断模式”,等待触摸屏被按下 */

printf("Touch the screem to test, press any key to exit\n\r");
getc();

// 屏蔽 ADC 中断
INTSUBMSK |= BIT_SUB_TC;
INTSUBMSK |= BIT_SUB_ADC;
INTMSK |= BIT_ADC;
}

```

(2) INT_TC 的中断服务程序。

```

/*
 * INT_TC 的中断服务程序
 * 当触摸屏被按下时,进入自动(连续) X/Y 轴坐标转换模式;
 * 当触摸屏被松开时,进入等待中断模式,再次等待 INT_TC 中断
 */
static void Isr_Tc(void)
{
    if (ADCDAT0 & 0x8000)
    {
        printf("Stylus Up!! \n\r");
        wait_down_int();    /* 进入“等待中断模式”,等待触摸屏被按下 */
    }
    else
    {
        printf("Stylus Down: ");

        mode_auto_xy();    /* 进入自动(连续) X/Y 轴坐标转换模式 */
    }
}

```

```

/* 设置位[0]为1,启动 A/D 转换
 * 注意:ADCDLY 为 50000,PCLK = 50MHz,
 *      要经过(1/50MHz) × 50000 = 1ms 之后才开始转换 X 坐标
 *      再经过 1ms 之后才开始转换 Y 坐标
 */
ADCCON |= ADC_START;
}

```

```

// 清 INT_TC 中断
SUBSRCPND |= BIT_SUB_TC;
SRCPND     |= BIT_ADC;
INTPND     |= BIT_ADC;
}

```

(3) INT_ADC 的中断服务程序。

```

/*
 * INT_ADC 的中断服务程序
 * A/D 转换结束时发生此中断
 * 先读取 X、Y 坐标值,再进入等待中断模式
 */
static void Isr_Adc(void)
{
    // 打印 X、Y 坐标值
    printf("xdata = %4d, ydata = %4d\r\n", (int)(ADCDAT0 & 0x3ff), (int)(ADCDAT1 & 0x3ff));

    /* 判断是 S3C2410 还是 S3C2440 */
    if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
    {
        // S3C2410
        wait_down_int(); /* 进入“等待中断模式”,等待触摸屏被松开 */
    }
    else
    {
        // S3C2440
        wait_up_int(); /* 进入“等待中断模式”,等待触摸屏被松开 */
    }

    // 清 INT_ADC 中断
    SUBSRCPND |= BIT_SUB_ADC;
    SRCPND     |= BIT_ADC;
    INTPND     |= BIT_ADC;
}

```

}

5. 测试驱动

将光盘提供的裸驱动(位于\work\studydriver\ts\baredrv 目录)make 后,把得到的 adc_ts.bin 文件烧写到 Nor Flash 中,重启开发板,在超级终端中输入 t 后,触摸触摸屏的某一位置,程序将会显示该点的坐标。显示如下:

```
##### Test ADC and Touch Screen #####
[T] Test Touch Scream
Enter your selection: Touch the screen to test, press any key to exit
Stylus Down: xdata = 845, ydata = 895
Stylus Up!!
Stylus Down: xdata = 169, ydata = 165
Stylus Up!!
```

12.3.2 Linux 输入子系统

1. 输入子系统结构

在 Linux 中,输入子系统是由输入子系统设备驱动层、输入子系统核心层(Input Core)和输入子系统事件处理层(Event Handler)组成。其中设备驱动层提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件,再通过核心层提交给事件处理层;而核心层对下提供了设备驱动层的编程接口,对上又提供了事件处理层的编程接口;而事件处理层就为用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理。所以这使得我们输入设备的驱动部分不用再关心对设备文件的操作,而是要关心对各硬件寄存器的操作和提交的输入事件。下面用图 12-9 来描述一下这三者的关系吧!

图 12-10 说明 Linux 输入子系统的结构,更加形象容易理解。

2. 输入子系统设备驱动层实现原理

在 Linux 中,Input 设备用 input_dev 结构体描述,定义在 input.h 中。设备的驱动只需按照如下步骤就可实现了。

- (1) 在驱动模块加载函数中设置 Input 设备支持 input 子系统的哪些事件。
- (2) 将 Input 设备注册到 input 子系统中。
- (3) 在 Input 设备发生输入操作时(如:键盘被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时等),提交所发生的事件及对应的键值/坐标等状态。

Linux 中输入设备的事件类型有(这里只列出了常用的一些,更多请看 Linux/input.h 中):

EV_SYN 0x00 同步事件

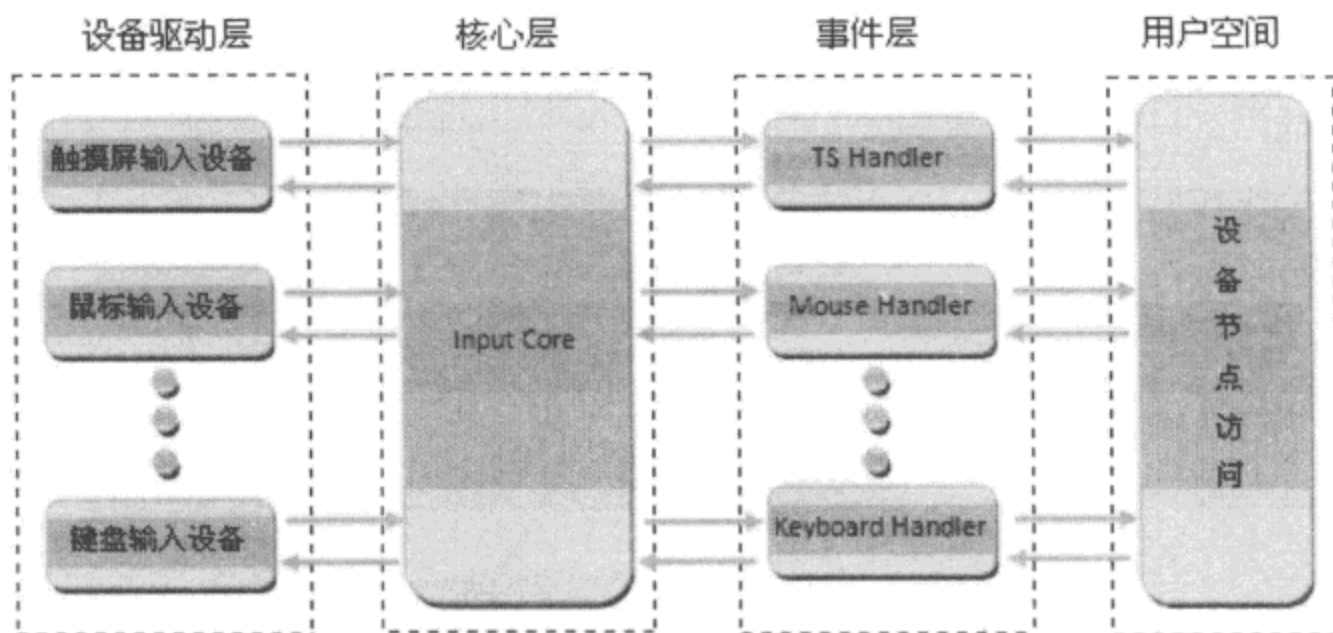


图 12-9 Linux 输入子系统结构图 1

| | | |
|--------|------|-----------------------------|
| EV_KEY | 0x01 | 按键事件 |
| EV_REL | 0x02 | 相对坐标(如鼠标移动,报告的是相对最后一次位置的偏移) |
| EV_ABS | 0x03 | 绝对坐标(如触摸屏和操作杆,报告的是绝对的坐标位置) |
| EV_MSC | 0x04 | 其他 |
| EV_LED | 0x11 | LED |
| EV_SND | 0x12 | 声音 |
| EV_REP | 0x14 | Repeat |
| EV_FF | 0x15 | 力反馈 |

用于提交较常用的事件类型给输入子系统的函数有:

`void input_report_key(struct input_dev * dev, unsigned int code, int value);` //提交按键事件的函数

`void input_report_rel(struct input_dev * dev, unsigned int code, int value);` //提交相对坐标事件的函数

`void input_report_abs(struct input_dev * dev, unsigned int code, int value);` //提交绝对坐标事件的函数

注意,在提交输入设备的事件后必须用下列方法使事件同步,让它告知 input 系统,设备驱动已经发出了一个完整的报告:

`void input_sync(struct input_dev * dev)`

12.3.3 Linux 下触摸屏驱动的实现步骤

(1) 先实现加载和卸载部分,在驱动加载部分,主要做的事情是:启用 ADC 所需要的时

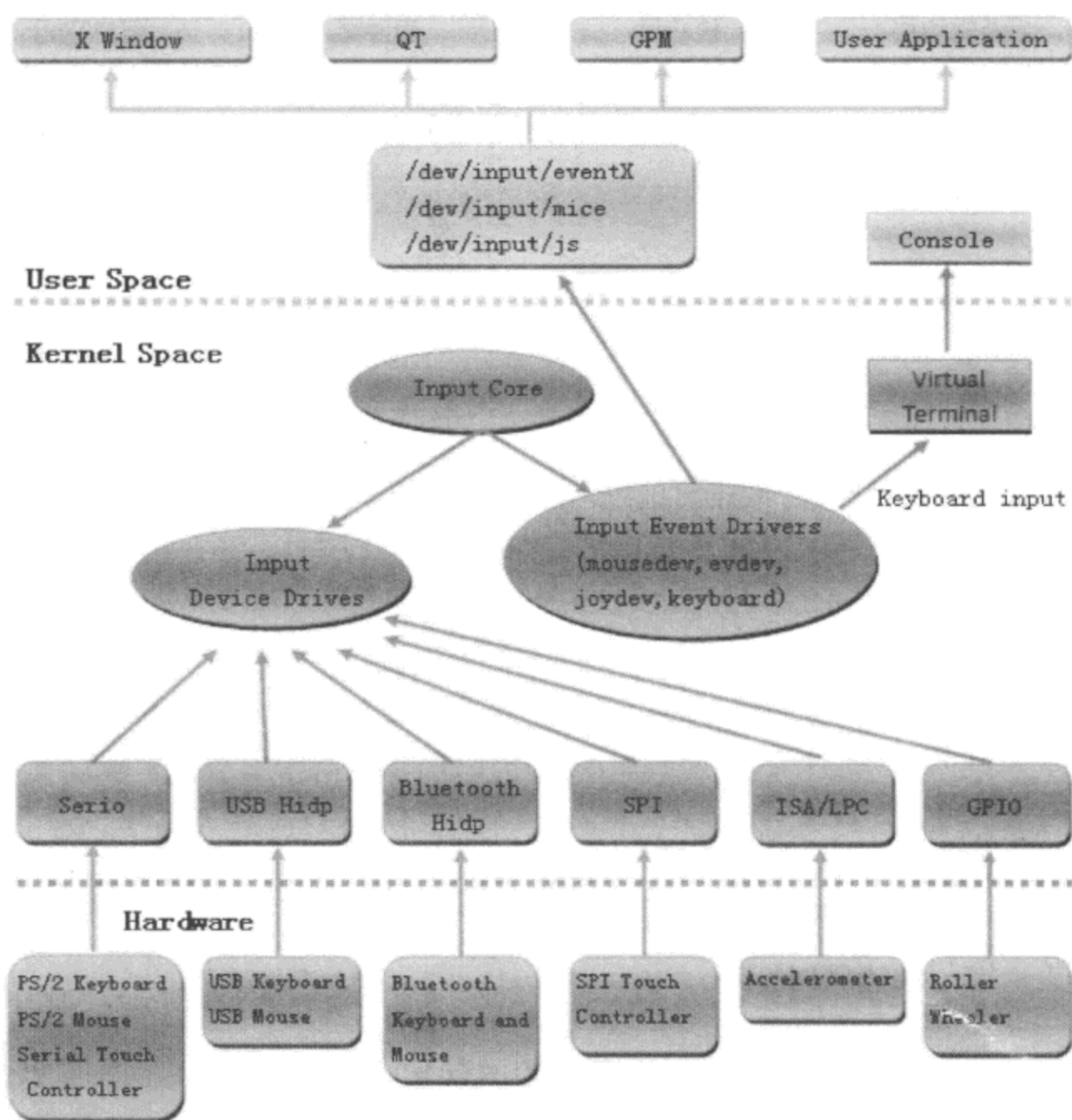


图 12-10 Linux 输入子系统结构图 2

钟、映射 I/O 口、初始化寄存器、申请中断、初始化输入设备、将输入设备注册到输入子系统。代码如下：

```
static int __init ts_init(void)
{
    int ret;
```

/* 从平台时钟队列中获取 ADC 的时钟，这里为什么要取得这个时钟，因为 ADC 的转换频率跟时钟有关。系统的一些时钟定义在 arch/arm/plat-s3c24xx/s3c2410-clock.c 中 */

```
adc_clk = clk_get(NULL, "adc");
```

/* 时钟获取后要使能后才可以使使用，clk_enable 定义在 arch/arm/plat-s3c/clock.c 中 */

```
clk_enable(adc_clk);
```

/* 将 ADC 的 I/O 端口占用的这段 I/O 空间映射到内存的虚拟地址, ioremap 定义在 io.h 中。注意: I/O 空间要映射后才能使用, 以后对虚拟地址的操作就是对 I/O 空间的操作, S3C2410_PA_ADC 是 ADC 控制器的基地址, 定义在 mach-s3c2410/include/mach/map.h 中, 0x20 是虚拟地址长度大小 */

```
adc_base = ioremap(S3C2410_PA_ADC, 0x20);
```

/* 初始化 ADC 控制寄存器和 ADC 触摸屏控制寄存器 */

```
adc_initialize();
```

/* 申请 ADC 中断, A/D 转换完成后触发。这里使用共享中断 IRQF_SHARED 是因为该中断号在 ADC 驱动中也使用了, 最后一个参数 2 是随便给的一个值, 因为如果不给值设为 NULL 的话, 中断就申请不成功 */

```
ret = request_irq(IRQ_ADC, adc_irq, IRQF_SHARED, DEVICE_NAME, (void *)2);
```

/* 申请触摸屏中断, 对触摸屏按下或提笔时触发 */

```
ret = request_irq(IRQ_TC, tc_irq, 0, DEVICE_NAME, (void *)2);
```

/* 给输入设备申请空间, input_allocate_device 定义在 input.h 中 */

```
ts_dev = input_allocate_device();
```

/* 下面初始化输入设备, 即给输入设备结构体 input_dev 的成员设置值。

evbit 字段用于描述支持的事件, 这里支持同步事件、按键事件、绝对坐标事件,

BIT 宏实际就是对 1 进行位操作, 定义在 Linux/bitops.h 中 */

```
ts_dev->evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) | BIT(EV_ABS);
```

/* keybit 字段用于描述按键的类型, 在 input.h 中定义了很多, 这里用 BTN_TOUCH 类型来表示触摸屏的点击 */

```
ts_dev->keybit[BITS_TO_LONGS(BTN_TOUCH)] = BIT(BTN_TOUCH);
```

/* 对于触摸屏来说, 使用的是绝对坐标系统。这里设置该坐标系统中 X 和 Y 坐标的最小值和最大值(0~1023)

ABS_X 和 ABS_Y 就表示 X 坐标和 Y 坐标, ABS_PRESSURE 就表示触摸屏是按下还是抬起状态 */

```
input_set_abs_params(ts_dev, ABS_X, 0, 0x3FF, 0, 0);
```

```
input_set_abs_params(ts_dev, ABS_Y, 0, 0x3FF, 0, 0);
```

```
input_set_abs_params(ts_dev, ABS_PRESSURE, 0, 1, 0, 0);
```

/* 以下是设置触摸屏输入设备的身份信息, 直接在这里写死。

这些信息可以在驱动挂载后在 /proc/bus/input/devices 中查看到 */

```
ts_dev->name = DEVICE_NAME; /* 设备名称 */
```

```

    ts_dev->id.bustype    = BUS_RS232;    /* 总线类型 */
    ts_dev->id.vendor     = 0xDEAD;       /* 经销商 ID 号 */
    ts_dev->id.product    = 0xBEEF;       /* 产品 ID 号 */
    ts_dev->id.version    = 0x0101;       /* 版本 ID 号 */

    /* 好了,一些都准备就绪,现在就把 ts_dev 触摸屏设备注册到输入子系统中 */
    input_register_device(ts_dev);

    return 0;
}

static void __exit ts_exit(void)
{
    /* 屏蔽和释放中断 */
    disable_irq(IRQ_ADC);
    disable_irq(IRQ_TC);
    free_irq(IRQ_ADC, (void *)2);
    free_irq(IRQ_TC, (void *)2);

    /* 释放虚拟地址映射空间 */
    iounmap(adc_base);

    /* 屏蔽和销毁时钟 */
    if(adc_clk)
    {
        clk_disable(adc_clk);
        clk_put(adc_clk);
        adc_clk = NULL;
    }

    /* 将触摸屏设备从输入子系统中注销 */
    input_unregister_device(ts_dev);
}

module_init(ts_init);
module_exit(ts_exit);

/* 初始化 ADC 控制寄存器和 ADC 触摸屏控制寄存器 */
static void adc_initialize(void)

```

```

{
    /* 计算结果为(二进制):111111111000000,再根据数据手册得知
    此处是将 A/D 转换预定标器值设为 255、A/D 转换预定标器使能有效 */
    writel(S3C2410_ADCCON_PRSCEN | S3C2410_ADCCON_PRSCVL(0xFF), adc_base + S3C2410_ADCCON);

    /* 对 ADC 开始延时寄存器进行设置,延时值为 0xffff */
    writel(0xffff, adc_base + S3C2410_ADCDLY);

    /* WAIT4INT 宏计算结果为(二进制):11010011,再根据数据手册得知
    此处是将 ADC 触摸屏控制寄存器设置成等待按下中断模式 */
    writel(WAIT4INT(0), adc_base + S3C2410_ADCTSC);
}

```

(2) 接下来要做的是,在两个中断服务程序中实现触摸屏状态的转换。

① 如果触摸屏感觉到触摸,则触发触摸屏中断进入 `tc_irq`,获取 `ADC_LOCK` 后判断触摸屏状态为按下,则调用 `touch_timer_fire` 将触摸屏置为自动坐标转换状态,并启动 ADC 转换;

② 当 ADC 转换启动后,触发 ADC 中断进入 `adc_irq`,如果这一次转换的次数小于 4,则重新启动 ADC 进行转换,如果 4 次完毕后,启动一个时间滴答的定时器去执行 `touch_timer_fire`,让它去完成费时的向 input core 报告坐标的工作。同时将触摸屏置于等待放开中断状态(此为非转换状态),这将停止 ADC 转换。

注:在这个时间滴答内,ADC 转换是停止的,这是为了防止屏幕抖动。

主体代码如下:

```

/* 申明并初始化一个信号量 ADC_LOCK,对 ADC 资源进行互斥访问
 * 这样就能保证 ADC 资源在 ADC 驱动和触摸屏驱动中进行互斥访问
 */
DECLARE_MUTEX(ADC_LOCK);

/* 导出信号量 ADC_LOCK 在 ADC 驱动中使用,因为触摸屏驱动和 ADC 驱动公用
   相关的寄存器,为了不产生资源竞态,就用信号量来保证资源的互斥访问 */
EXPORT_SYMBOL(ADC_LOCK);

/* 作为一个标签,只有对触摸屏操作后才对 X 和 Y 坐标进行转换 */
static int OwnADC = 0;

/* 用于记录转换后的 X 坐标值和 Y 坐标值 */
static long xp;
static long yp;

```

```

/* 用于计数对触摸屏压下或抬起时模拟输入转换的次数，
   需要 4 次转换结果的平均值，以提高精确度
   */
static int count;

/* 定义一个 AUTOPST 宏，将 ADC 触摸屏控制寄存器设置成自动转换模式 */
#define AUTOPST (S3C2410_ADCTSC_YM_SEN | S3C2410_ADCTSC_YP_SEN | S3C2410_ADCTSC_XP_SEN | \
                 S3C2410_ADCTSC_AUTO_PST | S3C2410_ADCTSC_XY_PST(0))

/* 定义一个输入设备来表示我们的触摸屏设备 */
static struct input_dev *ts_dev;

/* 定义一个 WAIT4INT 宏，该宏将对 ADC 触摸屏控制寄存器进行操作
   S3C2410_ADCTSC_YM_SEN 这些宏都定义在 regs - adc.h 中 */
#define WAIT4INT(x) (((x)<<8) | S3C2410_ADCTSC_YM_SEN | S3C2410_ADCTSC_YP_SEN | \
                     S3C2410_ADCTSC_XP_SEN | S3C2410_ADCTSC_XY_PST(3))

/* 触摸屏中断服务程序，对触摸屏按下或提笔时触发执行 */
static irqreturn_t tc_irq(int irq, void *dev_id)
{
    /* 用于记录这一次 A/D 转换后的值 */
    unsigned long data0;
    unsigned long data1;

    /* 用于记录触摸屏操作状态是按下还是抬起 */
    int updown;

    /* ADC 资源可以获取，即上锁 */
    if (down_trylock(&ADC_LOCK) == 0)
    {
        /* 标识对触摸屏进行了操作 */
        OwnADC = 1;

        /* 读取这一次 A/D 转换后的值，注意这次主要读的是状态 */
        data0 = readl(adc_base + S3C2410_ADCDAT0);
        data1 = readl(adc_base + S3C2410_ADCDAT1);

        /* 记录这一次对触摸屏是压下还是抬起，该状态保存在数据寄存器的第 15 位，所以与上

```



```

S3C2410_ADCDAT0_UPDOWN * /
    updown = (! (data0 & S3C2410_ADCDAT0_UPDOWN)) && (! (data1 & S3C2410_ADCDAT0_UP-
DOWN));

    /* 判断触摸屏的操作状态 */
    if (updown)
    {
        /* 如果是按下状态,则调用 touch_timer_fire 函数来启动 ADC 转换,该函数定义后面再讲 */
        touch_timer_fire(0);
    }
    else
    {
        /* 如果是抬起状态,就结束了这一次的操作,所以就释放 ADC 资源的占有 */
        OwnADC = 0;
        up(&ADC_LOCK);
    }
}

return IRQ_HANDLED;
}

```

```

/* 定义并初始化了一个定时器 touch_timer,定时器服务程序为 touch_timer_fire */
static struct timer_list touch_timer = TIMER_INITIALIZER(touch_timer_fire, 0, 0);

```

```

/* ADC 中断服务程序,A/D 转换完成后触发执行 */
static irqreturn_t adc_irq(int irq, void * dev_id)
{

```

```

    /* 用于记录这一次 A/D 转换后的值 */
    unsigned long data0;
    unsigned long data1;

    if(OwnADC)
    {
        /* 读取这一次 A/D 转换后的值,注意这次主要读的是坐标 */
        data0 = readl(adc_base + S3C2410_ADCDAT0);
        data1 = readl(adc_base + S3C2410_ADCDAT1);
    }

```

/* 记录这一次通过 A/D 转换后的 X 坐标值和 Y 坐标值,根据数据手册可知,X 和 Y 坐标转换数值分别保存在数据寄存器 0 和 1 的第 0~9 位,所以这里与上 S3C2410_ADCDAT0_XPDATA_MASK 就是取 0~9



位的值 */

```

xp += data0 & S3C2410_ADCDATA0_XPDATA_MASK;
yp += data1 & S3C2410_ADCDATA1_YPDATA_MASK;

/* 计数这一次 A/D 转换的次数 */
count++;

if (count < (1<<2))
{
    /* 如果转换的次数小于 4,则重新启动 ADC 转换。目的是求 4 次平均值提高精确度 */
    writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, adc_base + S3C2410_ADCTSC);
    writel(readl(adc_base + S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_START, adc_base
+ S3C2410_ADCCON);
}
else
{
    /* 否则,启动一个时间滴答的定时器,将来就会去执行定时器服务程序,上报事件和数据 */
    mod_timer(&touch_timer, jiffies + 1);
    /* 将 ADC 触摸屏控制寄存器设置成等待放开中断模式 */
    writel(WAIT4INT(1), adc_base + S3C2410_ADCTSC);
}
}
return IRQ_HANDLED;
}

```

(3) 实现 touch_timer_fire 函数,它在 3 种情况下分别完成 3 种不同任务。

① 若触屏按下且全局变量 count 为 0(意味着刚按下触笔,尚未开始转换坐标),则将触屏置于自动转换坐标状态,并启动转换。这个操作小,能迅速完成。这种情况源于 tc_irq 对它的直接调用。

② 若触屏按下且全局变量 count 不为 0(意味着 4 次坐标转换全部完成),这种情况源于 adc_irq 启动一个时间滴答的定时器来执行它,且此时用户还未放开触笔,此时是中断的底半部,正好适合。执行两个操作:

(a) 求出 4 次转换的平均值,向 input core 报告键被按下这个事件以及绝对坐标。这个操作不能很迅速完成。

(b) 再次将触屏置于自动转换坐标状态,并启动转换。从而触发持续向 input core 报告按下事件以及绝对坐标,直到放开触笔。

③ 若触屏放开,这种情况源于 adc_irq 启动一个时间滴答的定时器来执行它,且此时用户已经放开触笔。执行两个操作:

(a)向 input core 报告放开按键事件。

(b)最后将触屏置于等待按下中断状态。这就完成了一次按键周期的动作。

主体代码,如下:

```
static void touch_timer_fire(unsigned long data)
{
    /* 用于记录这一次 A/D 转换后的值 */
    unsigned long data0;
    unsigned long data1;

    /* 用于记录触摸屏操作状态是按下还是抬起 */
    int updown;

    /* 读取这一次 A/D 转换后的值,注意这次主要读的是状态 */
    data0 = readl(adc_base + S3C2410_ADCDATA0);
    data1 = readl(adc_base + S3C2410_ADCDATA1);

    /* 记录这一次对触摸屏是压下还是抬起,该状态保存在数据寄存器的第 15 位,所以与上 S3C2410_
    ADCDATA0_UPDOWN */
    updown = (! (data0 & S3C2410_ADCDATA0_UPDOWN)) && (! (data1 & S3C2410_ADCDATA0_UPDOWN));

    /* 判断触摸屏的操作状态 */
    if (updown)
    {
        /* 如果状态是按下,并且 ADC 已经转换了就报告事件和数据 */
        if (count != 0)
        {
            xp >>= 2;
            yp >>= 2;

            /* 报告 X、Y 的绝对坐标值 */
            input_report_abs(ts_dev, ABS_X, xp);
            input_report_abs(ts_dev, ABS_Y, yp);

            /* 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);

            /* 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);
        }
    }
}
```

```

        /* 等待接收方收到数据后回复确认,用于同步 */
        input_sync(ts_dev);
    }
    /* 报告按键位置后,用户如不松开按键,则继续进行转换,持续报告按键位置 */
    /* 如果状态是按下,并且 ADC 还没有开始转换就启动 ADC 进行转换 */
    xp = 0;
    yp = 0;
    count = 0;

    /* 设置触摸屏的模式为自动转换模式 */
    writel(S3C2410_ADCTSC_PULL_UP_DISABLE | AUTOPST, adc_base + S3C2410_ADCTSC);

    /* 启动 ADC 转换 */
    writel(readl(adc_base + S3C2410_ADCCON) | S3C2410_ADCCON_ENABLE_START, adc_base +
S3C2410_ADCCON);

}
else
{
    /* 否则是抬起状态 */
    count = 0;

    /* 报告按键事件,键值为 0(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);

    /* 等待接收方受到数据后回复确认,用于同步 */
    input_sync(ts_dev);

    /* 将触摸屏重新设置为等待按下中断状态 */
    writel(WAIT4INT(0), adc_base + S3C2410_ADCTSC);

    /* 如果触摸屏抬起,就意味着这一次的操作结束,所以就释放 ADC 资源的占有 */
    if (OwnADC)
    {
        OwnADC = 0;
        up(&ADC_LOCK);
    }
}

```



```

    }
}
}

```

12.3.4 测试触摸屏驱动程序

(1) 将光盘提供的触屏驱动(位于\work\studydriver\ts\linuxdriver 目录)make 后,将得到的 mys3c2440ts.ko 用 insmod 命令加载进 kernel。

(2) 执行 cat /dev/event0 后,触击触摸屏,将会有输出,这表示触屏驱动正常运行:

```
# cat /dev/event0
```

```
底触_? _Y_底触 4? __? 底触;? 底触? ____底触? ___? 底触? _底触 AV___底触 jV
```

(3) 再将 LED 驱动加载进 kernel 后,执行 qtopia 将进入图形界面,可在图形界面下使用触摸屏:

```
# qtopia &
```

12.4 USB 驱动初步

12.4.1 Linux 下 4 种 USB 驱动简介与功能体验

1. 体验 USB 驱动的功能

(1) 插入 4GB 的 U 盘到开发板的 usb host 接口,开发板将识别这个插入过程并出现设备/dev/sda 和/dev/sda1。其中 sda 代表整个 U 盘,sda1 代表 U 盘上的第一个分区。

```
usb 1-1: new full speed USB device
```

```
using s3c2410-ohci and address 2
```

```
usb 1-1: configuration #1 chosen from 1 choice
```

```
scsi0 : SCSI emulation for USB Mass Storage devices
```

```
scsi 0:0:0:0: Direct-Access SanDisk S3 UCL Mode 3.00 PQ: 0 ANSI: 0 CCS
```

```
sd 0:0:0:0: [sda] 32768 512-byte hardware sectors (17 MB)
```

```
sd 0:0:0:0: [sda] Write Protect is off
```

```
sd 0:0:0:0: [sda] Assuming drive cache: write through
```

```
sd 0:0:0:0: [sda] 32768 512-byte hardware sectors (17 MB)
```

```
sd 0:0:0:0: [sda] Write Protect is off
```

```
sd 0:0:0:0: [sda] Assuming drive cache: write through
```

```
sda: unknown partition table
```

```
sd 0:0:0:0: [sda] Attached SCSI removable disk
```

资源解密
PDG

```
# ls /dev/sda *
/dev/sda  /dev/sda1
```

这种情况是将开发板作为 USB host: 插入 U 盘到开发板 USB 主机端, 开发板 USB 主机端检测到插入 U 盘设备并完成枚举和初始化过程。然后调用一个具体的 USB device 驱动(如 storage 设备驱动)并产生一个设备节点/dev/sda1。

(2) 将开发板的 USB device 接口插入 windows USB 口, 使得开发板的本机 Nand Flash 的/dev/mtdblock3 在计算机上通过 U 盘形式来访问(出现一个盘符)。

这种情况是将开发板作为 USB device: usb 设备端驱动(USB gadget 驱动)在 windows 用户的要求下将一个设备(mtdblock3)作为 U 盘设备接入 windows USB 主机端。并对 windows 发起的枚举过程作出正确的响应, 返回这个设备的相关信息, 使得最终 Windows 能正确识别出这个设备, 并出现一个 U 盘盘符供用户方便的访问这个存储介质。

2. Linux USB 驱动层次(图 12-11)

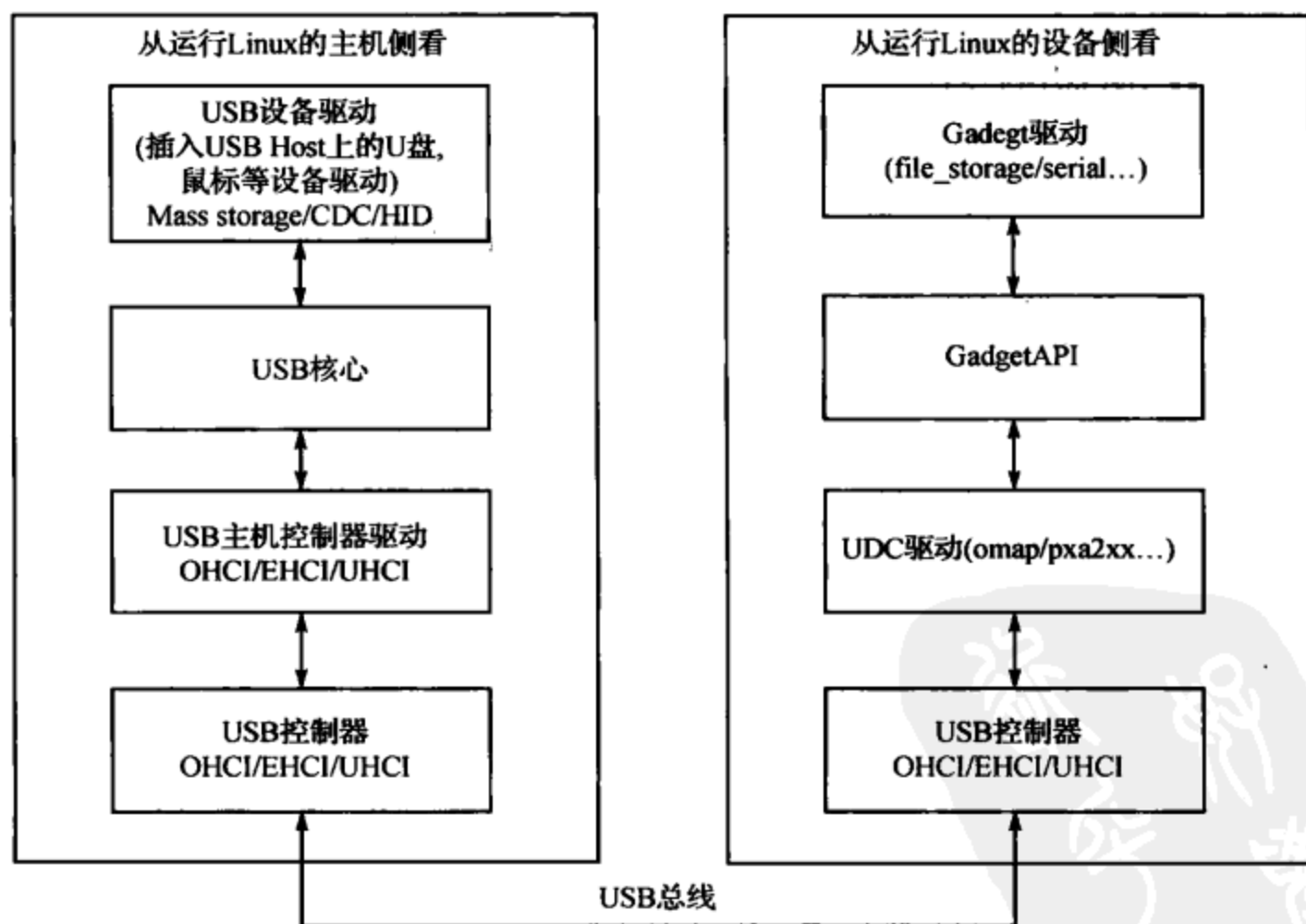


图 12-11 Linux USB 驱动层次

一个 Linux 系统既可以作为 USB host, 也可以作为 USB device。例如上面的例(1)的情况就是把开发板上的 Linux 系统作为 USB host 去访问别的 USB device; 例(2)的情况是把开发板上的 Linux 系统作为 USB device 供别的 USB host 访问。

(1) 作为 USB device。在硬件之上是 UDC 驱动,这通常由硬件厂家提供并集成在 Linux 内核中。

UDC 驱动之上,是由 Linux 内核提供的一组统一的 gadget API。

基于这些统一的 gadget API,就可以开发 USB gadget 驱动,例如:file_storage(该驱动将 Linux 系统作为 U 盘向 USB host 呈现)

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

USB 主机控制器驱动之上,是 USB 核心,它是 Linux 内核固有的一个重要子系统,如图 12-12 所示。主要完成两个功能:

① 实现复杂的 USB 传输和控制协议。

② 向下与具体的 USB 主机控制器驱动交互,屏蔽不同 USB 控制器硬件的差别,从而向上提供统一的内核 API 调用,供上层的 USB device 驱动调用以使用 USB 硬件。

由此可见,所谓 USB 驱动其实有 4 种类型:

- USB 主机控制器驱动,例如:drivers/usb/host/ohci-s3c2410.c。
- UDC 驱动,例如:drivers/usb/gadget/s3c2410_udc.c。
- USB gadget 驱动,例如:drivers/usb/gadget/file_storage.c。
- USB device 驱动,例如:drivers/usb/storage/usb-storage.c。

我们通常所说的 USB 驱动,大多是指的 USB device 驱动,也称做 USB 设备驱动。

12.4.2 USB 接口与规范

1. USB 接口

按照物理接口特性,USB 接口可以分为:

- (1) 主机(USB HOST)端。
- (2) USB 集线器(USB HUB)。
- (3) USB 设备(USB DEVICE)端。

USB 集线器其实就是一类特殊的 USB 设备。

在一个完整的 USB 拓扑结构上,必须有且仅有一个 USB 主机,一个或多个 USB 集线器和 USB 设备。

2. USB 规范

USB-通用串行总线是目前使用最广泛的外部总线。

USB 是采用单一的主从设备通信模式。总线上的唯一的主机负责轮询设备并发动各种传送,因此实现简单,成本相对低廉。

USB 从拓扑上讲类似于主机同外设之间点对点连接,设备连接汇集于集线器上。

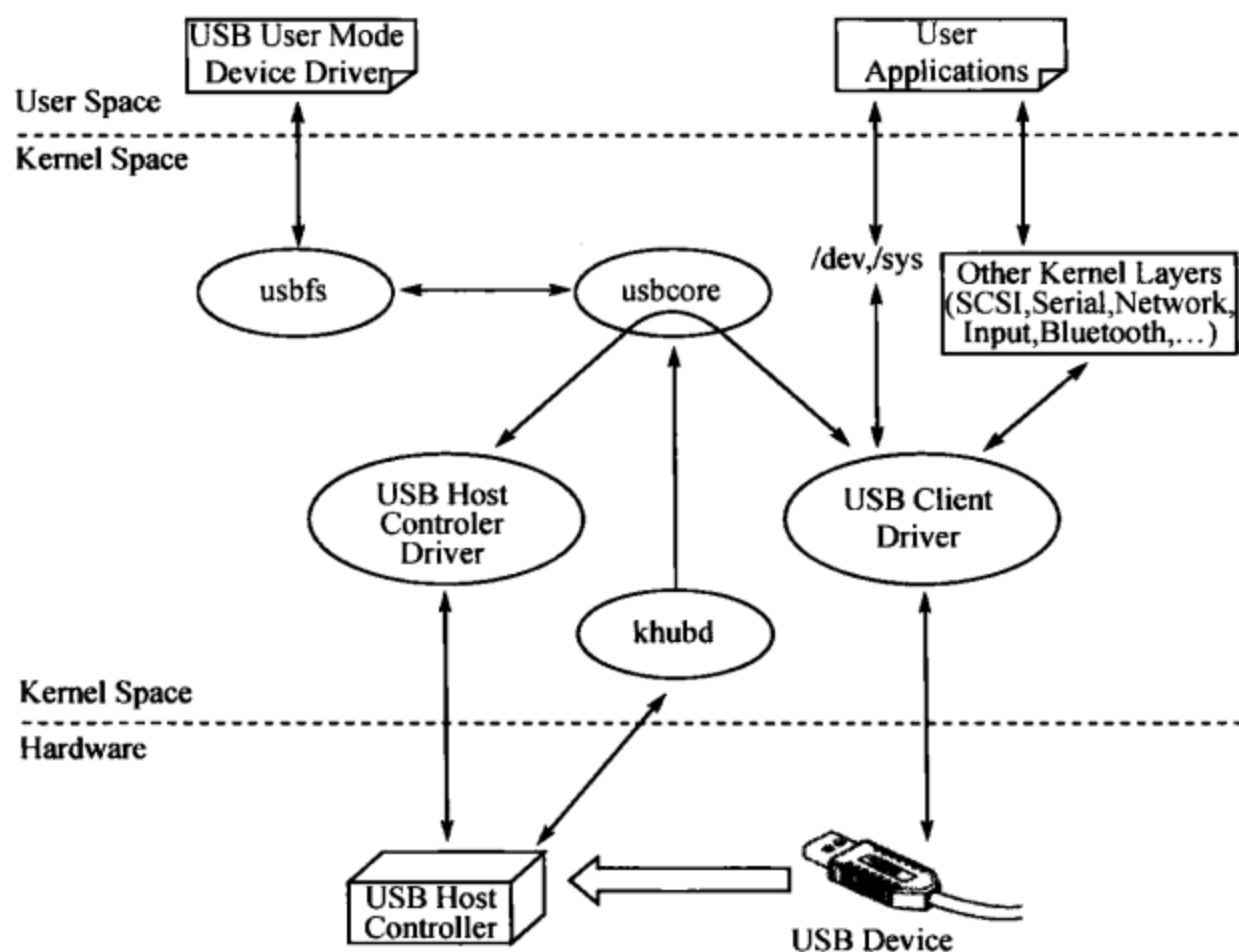


图 12-12 Linux-USB 子系统

USB 最新的规范是 USB 3.0,但使用最广泛的是 USB 2.0 版本,它定义了三种传输速率。

- Low speed——1.5Mb/s
- Full speed——12Mb/s
- High speed——480Mb/s

(1) USB 1.1 规范

USB1.1 规范支持低速(1.5Mb/s)和全速(12Mb/s)两种不同速率的数据传输和 4 种不同类型的数据传输方式:

- ① 控制传输(CONTROL TRANSFER)。
- ② 中断传输(INTERRUPT TRANSFER)。
- ③ 批量传输(BULK TRANSFER)。
- ④ 等时传输(ISOCHRONOUS TRANSFER)。

(2) USB 2.0 规范。USB 2.0 在兼容 USB1.1 低速(1.5Mb/s)和全速(12Mb/s)数据传输基础上,支持高速(480Mb/s)数据传输。

对于 USB 2.0 规范,同样支持控制传输、中断传输、批量传输和等时传输 4 种类型的数据传输方式。

在物理结构和拓扑结构上,USB 2.0 与 USB 1.1 也是完全相同的。

(3) USB OTG 规范。USB OTG 规范是作为对 USB 2.0 规范的补充而出现的,其目的是为了满足不同设备对 USB 接口性能的需求。

根据 USB OTG 规范,一个 USB 接口可同时具有 USB 主机和 USB 设备两种功能,根据其连接的其他设备属性,USBOTG 接口会自动转换成为适合 USB 总线需求的接口类型。

关于 USB HOST 接口,在符合 USB 规范的基础上,不同的厂商开发的 USB HOST 器件可能有着不同的结构特性。当前流行的 USB HOST 规范有:

- ① OHCI(OPEN HOST CONTROL INTERFACE)。
- ② UHCI(UNIVERSAL HOST CONTROL INTERFACE)。
- ③ EHCI(ENHANCED HOST CONTROL INTERFACE)。

ehci-hcd 模块支持的是 USB 2.0 控制器的高速模式,它本身并不支持全速或低速模式;ohci-hcd 或 uhci-hcd 模块提供对 USB 1.1 设备的支持。如果我们只配置了 EHCI,就没有办法使用 USB 的鼠标键盘。

12.4.3 USB 设备驱动基本知识

1. USB 规范中规定的标准概念由 Linus USB core 来实现处理(图 12-13)

(1) 端点(endpoints)。端点是 USB 总线传输最基本的概念,一个端点可以单方向传输数据。可以把端点看作是一个单方向的管道。端点有 4 种类型。

(2) 接口(interfaces)。若干端点可以捆绑起来,成为一个接口。接口可以作为完整的逻辑设备连接,例如鼠标设备、键盘设备。

需要指出的是,一个硬件设备可能包含多个逻辑设备。

接口可以有多个预设值,用来指定不同的参数。

(3) 配置(configurations)。接口捆绑起来成为配置。一个 USB 设备可以在不同的配置之间切换,一次只能激活一个配置。

2. USB 端点分类

USB 总线中的通信可以使用下面 4 种数据传输类型中的任意一种。

(1) 控制传输:这些是一些短的数据包,用于设备控制和配置,特别是在设备附加到主机上时。

(2) 批量传输:这些是数量相对大的数据包。像扫描仪或者 SCSI 适配器这样的设备使用这种传输类型。

(3) 中断传输:这些是定期轮询的数据包。主控制器会以特定的间隔自动发出一个中断。

(4) 等时传输:这些是实时的数据流,它们对带宽的要求高于可靠性要求。音频和视频设备一般使用这种传输类型。

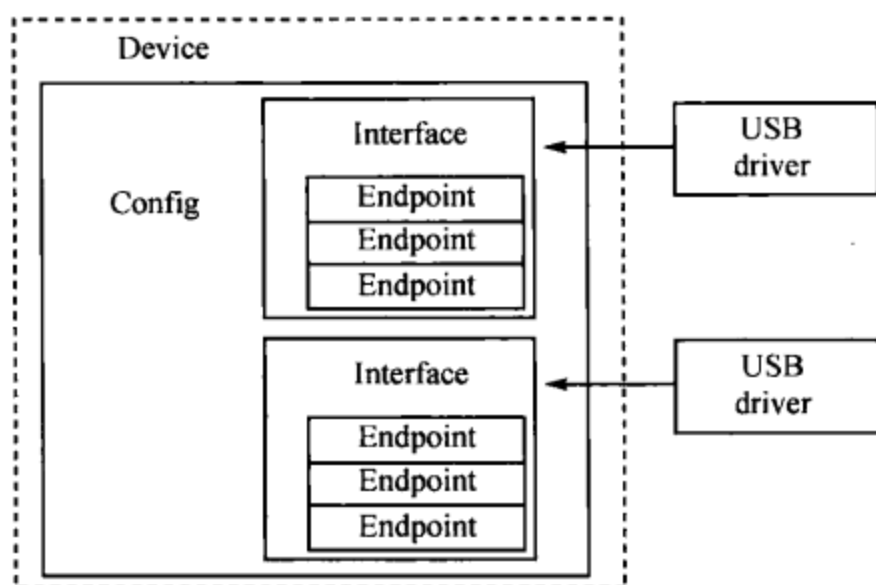


图 12-13 USB 设备基本概念示意图

3. 编写 USB 设备驱动所需内核 API 和数据结构简介

(1) USB 设备基础。

① 端点结构体(struct usb_host_endpoint)。

struct usb_endpoint_descriptor 包含上述结构体真实的端点信息，驱动相关的主要成员有：

- bEndpointAddress, 这是这个特定端点的 USB 地址。可以结合位掩码 USB_DIR_OUT 和 USB_DIR_IN 使用，来确定这个端点的数据传输方向是到设备还是到主机。
- bmAttributes, 这是端点的类型。可以和位掩码 USB_ENDPOINT_XFERTYPE_MASK 结合使用，来确定这个端点是否是 USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, 或者是类型 USB_ENDPOINT_XFER_INT。这些宏定义了等时，批量 和中断端点。
- wMaxPacketSize, 这是以字节计的这个端点可一次处理的最大大小。驱动可能发送大量的比这个值大的数据到端点，但是数据会被分为 wMaxPakcetSize 的块。
- bInterval, 如果这个端点是中断类型的，这个值是为这个端点设置的间隔，即在请求端点的中断之间的时间。这个值以毫秒表示。

② 接口。USB 端点被绑在接口中。USB 接口只处理一类 USB 逻辑连接，例如一个鼠标，一个键盘，或者一个音频流。接口数据结构是 struct usb_interface, 这个结构是 USB 核心传递给 USB 驱动的，并且是 USB 驱动接下来负责控制的。struct usb_interface 的成员：

- struct usb_host_interface * altsetting, 接口结构体数组，包含所有可能用于该接口的可选设置。每个 struct usb_host_interface 包含一套由 struct usb_host_endpoint 结构所定义的端点配置。
- unsigned num_altsetting, 由 altsetting 指针指向的预备设置的数目。

- `struct usb_host_interface * cur_altsetting`, 指向数组 `altsetting` 的一个指针, 表示这个接口当前的激活的设置。
- `int minor`, 如果绑定到这个接口的 USB 驱动使用 USB 主编号, 这个变量包含由 USB 核心安排给接口的次编号。

③ 配置。USB 接口本身被捆绑到配置。一个 USB 设备可有多个配置并且可能在它们之间转换以便改变设备的状态。

Linux 使用结构 `struct usb_host_config` 描述 USB 配置和使用结构 `struct usb_device` 描述整个 USB 设备。但 USB 设备驱动通常不会需要读写这些结构的任何值。

函数 `interface_to_usbdev`:

- 一个 USB 设备驱动通常需要把给定的 `struct usb_interface` 结构转换到 `struct usb_device` 结构, USB 核心在很多函数调用中需要该结构体。
- `struct usb_device` 需要转换为 `struct usb_interface` 时, 由核心完成, 不需要驱动程序做转换工作。

(2) USB core 与 USB 设备驱动程序的通信。在 Linux 内核中, USB 设备驱动程序与 USB core 之间通过 `urb` (USB request block) 来进行异步数据交换。USB 设备驱动程序只需要直接使用 USB core 提供的统一的内核 API 进行编程即可。

设备和多个端点之间可以使用同样或不同的 `urb`, 端点可以处理一个 `urb` 队列。使用 `urb` 的流程如下:

- ① USB 设备驱动创建一个 `urb`。
- ② 设置 `urb`, 将它关联到某个端点。
- ③ 设备驱动程序将 `urb` 提交给 USB core。
- ④ USB core 解析 `urb` 关联的设备, 并把它发送到适当的 USB 主控器。
- ⑤ 主控器按照 `urb` 的内容, 驱动总线设备完成传输。当传输完成时, 主控器将告知设备驱动程序。

- ⑥ 发送 `urb` 的设备驱动程序或 USB core 也可以取消 `urb`。

(3) USB core 提供的内核 API 主要就是用来处理 `urb` 的, 大体有如下一些 API。

- ① 创建和销毁 `urb`。

因为 USB core 要对 `urb` 进行引用计数, 所以驱动程序不能静态的创建 `urb`。应该使用下面的调用来动态的获得和释放 `urb`。

```
struct urb * usb_alloc_urb(int iso_packets, int mem_flags);
```

```
void usb_free_urb(struct urb * urb);
```

获得 `urb` 后, 在使用之前还要对它初始化。

```
usb_fill_int_urb
```

```
usb_fill_bulk_urb
```

usb_fill_control_urb

而等时 urb 需要手工地来填充各项。

② 提交和完成 urb。初始化好一个 urb 之后,驱动程序就可以将它提交给 USB core。

- `int usb_submit_urb(struct urb * urb, int mem_flags);`

提交过后,驱动程序在 complete 调用之前就不要再访问 urb

urb 完成时,该 urb 的 complete handler 就会被调用。完成分 3 种情况。

- urb 成功完成,设备返回正确的应答。这种情况下,urb 的 status 会被置 0。
- 发生了错误,urb 的 status 值将会被置为某个错误代号。
- urb 被取消,有可能是驱动程序或 USB core 取消了 urb,或者是该设备已经从总线上拔出。

取消一个 urb,可以通过下列调用来实现:

- `int usb_kill_urb(struct urb * urb);`
- `int usb_unlink_urb(struct urb * urb);`

(4) USB 设备驱动的注册

驱动程序需要创建一个 `usb_driver` 结构,该结构包含了下列项:

```
struct module * owner
const char * name
const struct usb_device_id * id_table
int (* probe) (struct usb_interface * intf, const struct usb_device_id * id)
void (* disconnect) (struct usb_interface * intf)
int (* ioctl) (struct usb_interface * intf, unsigned int code, void * buf)
int (* suspend) (struct usb_interface * intf, u32 state)
int (* resume) (struct usb_interface * intf)
```

使用 `usb_register(&myusb_driver)` 将驱动注册进 USB core;使用 `usb_deregister(&myusb_driver)` 将驱动从 usb core 中取消。

其中 `id_table` 含有表示 USB 设备的 vendor id 和 device id,提供了这个驱动支持的一个不同类型 USB 设备的列表。这个列表被 USB 核心用来决定设备该使用哪个驱动,热插拔脚本使用它来决定当特定设备被插入系统时哪个驱动自动加载。

当 USB core 检测到 USB 设备列表中的某个设备接入了系统,USB core 就会回调 `usb_driver` 结构体中的 `probe` 探测函数。

12.4.4 USB 设备驱动实例

1. 体验 USB 设备驱动

(1) 将开发板作为 USB device 与 Linux PC 相连

(2) 在开发板上进入 u-boot, 输入命令从 PC 下载数据:

```
Dennis Yang > usbslave 1 0x30000000
USB host is connected. Waiting a download.
```

(3) 编译光盘提供的 USB 设备驱动(位于\work\studydriver\usb 目录)后, 将得到的 usb-skeleton.ko 加载进 OS

```
/work/studydriver/usb$ sudo insmod usb-skeleton.ko
```

(4) 编译光盘提供的 PC 端传输测试软件(位于\work\studydriver\usb 目录)

```
/work/studydriver/usb$ gcc dnw.c -o dnw
```

(5) 使用 dnw 传输内核到开发板内存:

```
/work/studydriver/usb$ sudo ./dnw /work/sysbuild/linux-2.6.22.6/arch/arm/
boot/uImage
```

(6) 开发板显示得到传输来的数据:

```
Dennis Yang > usbslave 1 0x30000000
USB host is connected. Waiting a download.
Now, Downloading [ADDRESS:30000000h,TOTAL:1512234]
RECEIVED FILE SIZE: 1512234 (738KB/S, 2S)
```

(7) 验证下载的内核正确:

```
Dennis Yang > iminfo 0x30000000
## Checking Image at 30000000 ...
Image Name:   Linux-2.6.22.6
Created:      2010-10-13 13:59:26 UTC
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    1512160 Bytes = 1.4 MB
Load Address: 30008000
Entry Point:  30008000
Verifying Checksum ... OK
```

2. 实现代码

(1) 向 USB core 注册设备驱动:

```
/* Define these values to match your devices */
#define USB_SKEL_VENDOR_ID    0xffff0
#define USB_SKEL_PRODUCT_ID   0xffff0

/* table of devices that work with this driver */
static struct usb_device_id skel_table[] = {
```



```

        { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
        { USB_DEVICE(0x5345, 0x1234) },
        { } /* Terminating entry */
    };
MODULE_DEVICE_TABLE (usb, skel_table);

static struct usb_driver skel_driver = {
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};

static int __init usb_skel_init(void)
{
    int result;

    /* register this driver with the USB subsystem */
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}

static void __exit usb_skel_exit(void)
{
    /* deregister this driver with the USB subsystem */
    usb_deregister(&skel_driver);
}

module_init (usb_skel_init);
module_exit (usb_skel_exit);

```

(2) 当设备(开发板)接入 PC 时,usb core 将回调 skel_probe 函数,在该函数中将设备注册进 OS。skel_probe 的实现代码如下:

```

static struct file_operations skel_fops = {
    .owner = THIS_MODULE,
    .read = skel_read,
    .write = skel_write,
    .open = skel_open,
    .release = skel_release,
};

```

```

static struct usb_class_driver skel_class = {
    .name = "usb/skel %d",
    .fops = &skel_fops,
    .minor_base = USB_SKEL_MINOR_BASE,
};

static int skel_probe(struct usb_interface * interface, const struct usb_device_id * id)
{
    struct usb_skel * dev = NULL;
    struct usb_host_interface * iface_desc;
    struct usb_endpoint_descriptor * endpoint;
    size_t buffer_size;
    int i;
    int retval = - ENOMEM;

    /* allocate memory for our device state and initialize it */
    dev = kmalloc(sizeof(struct usb_skel), GFP_KERNEL);
    memset(dev, 0x00, sizeof (* dev));
    kref_init(&dev->kref);

    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    dev->interface = interface;

    /* set up the endpoint information */
    /* use only the first bulk-in and bulk-out endpoints */
    iface_desc = interface->cur_altsetting;
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc;

        if (! dev->bulk_in_endpointAddr &&
            (endpoint->bEndpointAddress & USB_DIR_IN) &&
            ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
             == USB_ENDPOINT_XFER_BULK)) {
            /* we found a bulk in endpoint */
            buffer_size = endpoint->wMaxPacketSize;
            dev->bulk_in_size = buffer_size;
            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        }
    }
}

```

资源解密网
PDG

```

        if (! dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }

    if (! dev->bulk_out_endpointAddr &&
        ! (endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
         == USB_ENDPOINT_XFER_BULK)) {
        /* we found a bulk out endpoint */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}

if (! (dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk - in and bulk - out endpoints");
    goto error;
}

/* save our data pointer in this interface device */
usb_set_intfdata(interface, dev);

/* we can register the device now, as it is ready */
retval = usb_register_dev(interface, &skel_class);

return 0;

error:
    if (dev)
        kref_put(&dev->kref, skel_delete);
    return retval;
}

```

(3) 这样一来, OS 中就存在字符设备 /dev/skel0, 当它被 open、read、write 时, 相应的 skel_open、skel_read、skel_write 就被执行:

```

static int skel_open(struct inode * inode, struct file * file)
{
    struct usb_skel * dev;
    struct usb_interface * interface;

```



```

int subminor;
int retval = 0;

subminor = iminor(inode);

interface = usb_find_interface(&skel_driver, subminor);

dev = usb_get_intfdata(interface);

/* increment our usage count for the device */
kref_get(&dev->kref);

/* save our object in the file's private structure */
file->private_data = dev;
}

static ssize_t skel_write(struct file * file, const char __user * user_buffer, size_t count,
loff_t * ppos)
{
    struct usb_skel * dev;
    int retval = 0;
    struct urb * urb = NULL;
    char * buf = NULL;

    dev = (struct usb_skel *)file->private_data;

    /* verify that we actually have some data to write */
    if (count == 0)
        goto exit;

    /* create a urb, and a buffer for it, and copy the data to the urb */
    urb = usb_alloc_urb(0, GFP_KERNEL);

    buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);

    if (copy_from_user(buf, user_buffer, count)) {
        retval = -EFAULT;
        goto error;
    }
}

```

```

/* initialize the urb properly */
usb_fill_bulk_urb(urb, dev->udev,
                  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
                  buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);

/* release our reference to this urb, the USB core will eventually free it entirely */
usb_free_urb(urb);

exit:
    return count;

error:
    usb_buffer_free(dev->udev, count, buf, urb->transfer_dma);
    usb_free_urb(urb);
    kfree(buf);
    return retval;
}

static void skel_write_bulk_callback(struct urb *urb)
{
    printk(KERN_ALERT "enter callback\n");
    /* sync/async unlink faults aren't errors */
    if (urb->status &&
        ! (urb->status == -ENOENT ||
            urb->status == -ECONNRESET ||
            urb->status == -ESHUTDOWN)) {
        printk(KERN_ALERT "%s - nonzero write bulk status received: %d",
            __FUNCTION__, urb->status);
    }

    /* free up our allocated buffer */
    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                    urb->transfer_buffer, urb->transfer_dma);
}

```


参考文献

- [1] 杜春雷. ARM 体系结构与编程[M]. 北京:清华大学出版社, 2003.
- [2] 詹荣开. 嵌入式系统 BootLoader 技术内幕. <http://www.ibm.com/developerworks/cn/linux/l-btloader/>.
- [3] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers 3rd.
- [4] 韦东山. 嵌入式 Linux 应用开发完全手册[M]. 北京:人民邮电出版社, 2008.
- [5] Robert Love. Linux 内核设计与实现[M]. 2 版. 陈莉君等译. 北京:机械工业出版社, 2009.
- [6] 广州友善之臂计算机科技有限公司. mini2440 用户手册.
- [7] ARM Ltd. ARM Architecture Reference Manual.
- [8] Samsung electronics. S3C2440A USER'S MANUAL.
- [9] Arthur Griffith. GCC: The Complete Reference. McGraw-Hill/Osborne.
- [10] ARM Ltd. ads book online.
- [11] 黄刚. S3C2440 上 LCD 驱动(FrameBuffer)实例开发讲解. <http://hbhuanggang.cublog.cn>.
- [12] GNU Make. Richard M. Stallman, Roland McGrath, Paul D. Smith.
- [13] 徐海兵. GNU make 中文手册.
- [14] 黄刚. S3C2440 上触摸屏驱动实例开发讲解. <http://hbhuanggang.cublog.cn>.
- [15] Steve Chamberlain, Ian Lance Taylor. Using ld. Red Hat Inc.
- [16] 广州友善之臂计算机科技有限公司. QQ2440 用户手册.
- [17] 赵炯. Linux 内核完全注释[M]. 北京:机械工业出版社, 2004.

数字资源
PDG