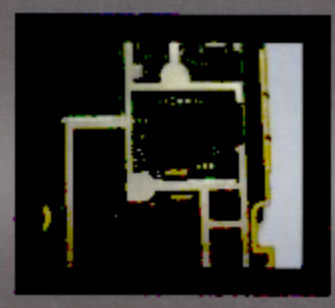
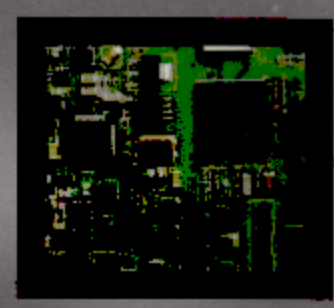




嵌入式Linux应用开发完全手册

- 基于ARM 9 + Linux 2.6平台
- 从基础讲起，引导读者快速入门
- 实例丰富，可直接应用于工程实践

本书全部实例代码及相关工具

韦东山 编著

人民邮电出版社
POSTS & TELECOM PRESS

嵌入式Linux应用开发完全手册

韦东山　编著

人民邮电出版社
北京

图书在版编目（CIP）数据

嵌入式 Linux 应用开发完全手册 / 韦东山编著. —北京：人民邮电出版社，2008.8
ISBN 978-7-115-18262-3

Ⅰ. 嵌… Ⅱ. 韦… Ⅲ. Linux 操作系统—程序设计—技术手册 Ⅳ. TP316.89-62

中国版本图书馆 CIP 数据核字（2008）第 082991 号

内 容 提 要

本书全面介绍了嵌入式 Linux 系统开发过程中，从底层系统支持到上层 GUI 应用的方方面面，内容涵盖 Linux 操作系统的安装及相关工具的使用、配置，嵌入式编程所需要的基础知识（交叉编译工具的选项设置、Makefile 语法、ARM 汇编指令等），硬件部件的使用及编程（囊括了常见硬件，比如 UART、I^2C、LCD 等），U-Boot、Linux 内核的分析、配置和移植，根文件系统的构造（包括移植 busybox、glibc、制作映象文件等），内核调试技术（比如添加 kgdb 补丁、栈回溯等），驱动程序编写及移植（LED、按键、扩展串口、网卡、硬盘、SD 卡、LCD 和 USB 等），GUI 系统的移植（包含两个 GUI 系统：基于 Qtopia 和基于 X），应用程序调试技术。

本书从最简单的点亮一个 LED 开始，由浅入深地讲解，使读者最终可以配置、移植、裁剪内核，编写驱动程序，移植 GUI 系统，掌握整个嵌入式 Linux 系统的开发方法。

本书由浅入深，循序渐进，适合刚接触嵌入式 Linux 的初学者学习，也可作为大、中专院校嵌入式相关专业本科生、研究生的教材。

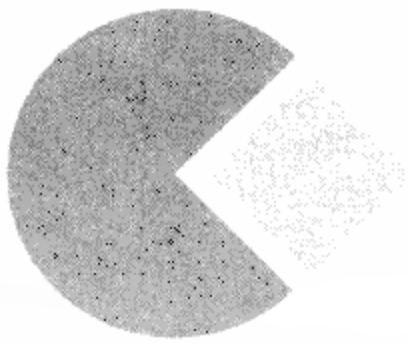
嵌入式 Linux 应用开发完全手册

◆ 编　　著　韦东山
　责任编辑　黄　焱
◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京昌平百善印刷厂印刷
◆ 开本：787×1092　1/16
　印张：37.25
　字数：908 千字　　2008 年 8 月第 1 版
　印数：1 – 4 000 册　　2008 年 8 月北京第 1 次印刷

ISBN 978-7-115-18262-3/TP

定价：69.00 元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154



前　言

背景知识

嵌入式 Linux 在嵌入式领域发展迅速、需求旺盛，但是嵌入式 Linux 的入门很难。初学者多是自己琢磨，效率不高。学习过程中碰到的问题千奇百怪，解决后却往往发现是极其低级的错误，以作者为例，初学时在论坛疯狂发帖求教，现在回头一看不免感叹：怎么会提出这么弱智的问题？但是，当时就是被这类问题折磨得寝食难安。

相对于嵌入式 Linux 常识的匮乏，更大的困难是缺乏完善的知识结构：只了解硬件，或是只了解软件。对于有志于从事底层系统开发（比如改造 Bootloader、钻研内核、为新硬件编写驱动程序）的人，对于想从上层软件开发转到底层软件开发的人，应该看得懂电路原理图，看得懂芯片数据手册，清楚地知道软件是怎样和硬件发生作用的。

同样，对于想从硬件岗位转到软件岗位的人，对于想从传统单片机（比如 51 单片机）编程进一步学习“有操作系统的”嵌入式编程的人，需要找到一个学习的切入点：先掌握各个硬件部件的简单编程，再将它们组合起来构成一个相对复杂的软件系统——比如 Bootloader，进而编写基于操作系统的驱动程序，最后深入钻研操作系统内核。

对于尚未参加工作的在校生来说，缺乏实际的操作经验可能是就业的最大障碍。很多人买了开发板想进一步练习，却发现不知从何入手。

鉴于上述种种困难及需求，作者结合自己的学习经历、工作心得写成此书，期望能帮助读者加快嵌入式 Linux 的入门速度，并体会到深入学习嵌入式 Linux 的乐趣。

关于本书

本书以 S3C2410、S3C2440 开发板为例，从分析硬件上电执行的第一条指令开始，到构造出一个类似 PDA、基于 Linux 的桌面 GUI 系统，带领读者学习、掌握从最底层到最高层的软件编写方法。

本书主要涉及以下主题：

- 开发环境的搭建（包括安装 Linux 系统及日常使用的工具）；
- 开发板上各硬件部件的使用方法及实际的编程操作；

- 嵌入式 Linux 系统的构造（包括 Bootloader、内核、文件系统等）;
- 嵌入式 Linux 驱动程序的编写方法及大量实例;
- GUI 系统的移植（两个 GUI 系统：基于 Qtopia 和基于 X）;
- 调试技术（包括内核调试技术和应用程序调试技术）。

本书所有章节都以理论结合代码的方式进行讲解，并可按照书中说明进行实际操作，力求让读者“知其然，也知其所以然”。

本书内容及组织方式

本书按照嵌入式 Linux 初学者的学习过程，从简单到复杂，从底层软件到上层软件进行讲解，全书分 5 篇，共 27 章。

第 1 篇（第 1 章至第 4 章）为嵌入式 Linux 开发环境构建篇，主要讲解以下内容。

- 第 1 章介绍基于 ARM 的嵌入式 Linux 系统的基本概念。
- 第 2 章讲解嵌入式开发环境的建立，包括在 PC 上安装、配置 Linux 操作系统，安装随书光盘。
- 第 3 章介绍交叉编译工具的选项、Makefile 的语法以及本书用到的 ARM 汇编指令及相关知识，这章可以当作阅读后续章节时的参考手册。
- 第 4 章介绍了一些日常工作要用到工具，比如源码阅读、编辑工具等。

第 2 篇（第 5 章至第 14 章）为 ARM9 嵌入式系统基础实例篇，具体内容如下。

本篇首先根据 S3C2410、S3C2440 的数据手册介绍各硬件部件的使用方法，然后介绍怎样编写程序来操作它们。文中穿插介绍了连接器的很多使用技巧，读者可以由此接触到“程序的内部结构”，这是单纯的上层开发人员所缺乏的。通过读写各个硬件部件的寄存器来操作硬件，读者还可以深刻体会到“软件”和“硬件”是怎样发生作用的，是第 3 篇、第 4 篇的基础。

第 3 篇（第 15 章至第 18 章）为嵌入式 Linux 系统移植篇，主要讲解以下内容。

- 第 15 章深入分析 U-Boot（它负责引导内核）的代码结构，并详细介绍了将它移植到开发板上的方法。
- 第 16 章首先分析了内核的代码结构，然后深入分析它的启动过程，最后将它移植到开发板上。
- 第 17 章先从整体上介绍了 Linux 文件系统的目录结构——FHS 标准。然后构造文件系统：移植常用工具的集合 Busybox，移植 glibc 库，建立各个目录，建立配置文件。最后修改、编译一些工具，使用它们来制作 yaffs、jffs2 文件系统映象文件。
- 第 18 章介绍了 3 种内核调试技术：printk、kgdb 补丁、使用 Oops 信息进行栈回溯。

第 4 篇（第 19 章至第 24 章）为嵌入式 Linux 设备驱动开发篇，具体内容如下。

在第 19 章中总体介绍了驱动程序的编写、移植方法，在第 20 章介绍了内核的异常处理体系结构——就是怎样使用中断。

其他章节都是一些例子：先总体介绍相关硬件的驱动程序架构，然后根据开发板的特性进行修改。

第 5 篇（第 25 章至第 27 章）为嵌入式 Linux 系统应用开发篇，主要讲解以下内容。

- 第 25 章移植了一个基于 Qtopia 的 GUI 系统，并且以简单的“Hello, world”程序为例编写、调试 GUI 程序。

- 第 26 章移植了一个基于 X 的 GUI 系统，里面涉及众多软件，读者可以体会到上层应用的开发过程，并且获得移植大型软件的经验。这章还介绍了一个名为 Scratchbox 的交叉编译工具包，它虚拟出一个可以直接编译软件的目标机器，使得“交叉编译”变为“本地编译”，大幅减少了为非 x86 平台移植软件所需的工作量。
- 第 27 章介绍了几种简便的应用程序调试技术，包括使用 strace 工具跟踪系统调用和信号，使用 memwatch 检查程序的内存漏洞，使用库函数 backtrace 和 backtrace_symbols 来定位段错误。

本书特色

- 由浅入深，从最简单的点亮 LED 讲起直至移植 GUI 系统。
- 实例丰富，每个实例都详尽地介绍原理及分析代码。
- 结构合理，先总体介绍概念、架构，然后进行具体操作。
- 包括初学者所碰到的常见问题。

参与本书编写的人员

本书由韦东山负责编写并统编全部书稿，陈汉仪、于明俭对本书的写作提供了大力支持，在此表示感谢。

感谢我的父母和女友，在本书写作过程中给了我强大的精神支持，鼓励、支持我，使我能够坚持写完本书。

同时参与编写的还有柴作朋、单辉、丁鹏、冯发勇、付贤会、葛仕明、何国宝、何圆明、何化成、黄永华、李志宏、廖娟、林清妹、陆江萍、祁晓璐、谭爱华、魏明辉、张帮芹、周霜、朱旭琪等，在此一并表示感谢。

我们为本书开通了专用的网站，网址是 http://www.100ask.net，读者可以直接同我们交流，共同学习和提高。

由于水平有限，书中难免遗漏和不足之处，恳请广大读者提出宝贵意见。本书责任编辑的联系方式是 huangyan@ptpress.com.cn，欢迎来信交流。

编　者

2008 年 6 月

目　录

第 1 篇　嵌入式 Linux 开发环境构建篇

第 2 篇 ARM9 嵌入式系统基础实例篇

第 3 篇 嵌入式 Linux 系统移植篇

第 4 篇 嵌入式 Linux 设备驱动开发篇

第 5 篇 嵌入式 Linux 系统应用开发篇

第 1 篇　嵌入式 Linux 开发环境构建篇

- 嵌入式 Linux 开发概述
- 嵌入式 Linux 开发环境构建
- 嵌入式编程基础知识
- Windows、Linux 环境下相关工具、命令的使用

第 1 章　嵌入式 Linux 开发概述

本章目标

- 了解嵌入式系统的概念及发展历史
- 了解 ARM 处理器
- 了解各类嵌入式操作系统

1.1　嵌入式系统介绍

1.1.1　嵌入式系统的定义和特点

1. 嵌入式系统的定义

嵌入式系统的定义为：以应用为中心、以计算机技术为基础、软硬件可裁剪、适用于应用系统，对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。它的主要特点是嵌入、专用。

从 20 世纪 70 年代起，微型机以小型、价廉、高速数值计算等特点迅速走向市场，它所具备的智能化水平在工业控制领域发挥了作用，常被组装成各种形状，“嵌入”到一个对象体系中，进行某类智能化的控制。这样一来，计算机便失去了原来的形态与“通用”的功能，为区别于通用计算机系统，将这类为了某个“专用”的目的，而“嵌入”到对象体系中的计算机系统，称为嵌入式计算机系统，简称嵌入式系统。

含有嵌入式系统的设备称为嵌入式设备，这在生活中随处可见：电子表、手机、MP3 播放器、摇控器等，涵盖了生产、工业控制、通信、网络、消费电子、汽车电子、军工等领域。从通俗、广义的角度来说，除电脑、超级计算机等具备比较强大计算能力及系统资源（比如内存、存储器等）的电子系统之外，凡具备计算能力的设备都可称为嵌入式设备。随着技术的进步，嵌入式设备的性能越来越高，一个相对高级的 PDA 的性能并不弱于一般的电脑。

2. 嵌入式系统的特点

嵌入式设备常应用于“特定”场合，与“通用的”个人电脑相比，具备以下特点。

（1）软件、硬件可裁剪。

将市面上的手机拆开，会发现虽然它们的功能是相似的，但是所用芯片多种多样，所用的操作系统也有多种，操作界面更是千变万化，操作的便利性各有千秋。这不同于个人电脑，CPU 除了 INTEL 就是 AMD 公司的，操作系统多用 Windows。功能、成本、开发效率等条件决定了嵌入式设备的选材多样化，软件、硬件可裁剪，即当不需要某项功能时，可以去除相关的软硬件。

（2）对功能、可靠性、成本、体积、功耗严格要求。

功能、可靠性、功耗这 3 点是软件开发人员最关注的地方。仍以手机为例，当选定硬件平台之后，处理器的性能已经被限定了，怎样使得手机的操作更人性化、菜单响应更快捷、具备更多更好的功能，这完全取决于软件。需要驱动程序和应用程序配合，最大程度地发挥硬件的性能。

也许读者见过这类手机，它的屏幕总是经过很长时间才熄灭，这使得它的电池很快耗光，只要在编写软件时进行改进，就可能成倍地延长电池的使用时间。一个优秀的嵌入式系统，对硬件性能的“压榨”、对软件的细致调节，已经到了精益求益的地步。有时候甚至为了节省几秒的启时间而大动脑筋，调整程序的启动顺序让耗时的程序稍后运行、改变程序的存储方式以便更快地加载等，甚至通过显示一个进度条让用户觉得时间没那么长。

1.1.2 嵌入式技术的发展历史

嵌入式技术在 20 世纪 70 年代起源于微型机，从此之后，通用计算机与嵌入式计算机就走上了两条不同的道路。通用计算机系统的技术要求是高速、海量的数值计算；技术发展方向是总线速度的无限提升，存储容量的无限扩大。 而嵌入式计算机系统的技术要求则是对象的智能化控制能力；技术发展方向是与对象系统密切相关的嵌入性能、控制能力与控制的可靠性。

嵌入式技术的发展日新月异，经历了单片机（SCM）、微控制器（MCU）、系统级芯片（SoC）3 个阶段。

1. SCM（Single Chip Microcomputer）

又称单片微型计算机，简称单片机，随着大规模集成电路的出现及其发展，计算机的 CPU、RAM、ROM、定时数器和多种 I/O 接口集成在一片芯片上，形成芯片级的计算机。

这个阶段主要是“寻求”单片形态嵌入式系统的最佳体系结构，也是从这个阶段起，嵌入式计算机技术与通用计算机技术走上两条不同的道路。

2. MCU（Micro Controller Unit）

MCU 即微控制器阶段的特征是：“满足”各类嵌入式应用，根据对象系统要求扩展各种外围电路与接口电路，突显其对象的智能化控制能力。它所涉及的领域都与对象系统相关，因此，发展 MCU 的重任不可避免地落在电气、电子技术厂家的身上。

实际上，MCU、SCM 之间的概念在日常工作中并不严格区分，很多时候一概以“单片机”称呼。随着能够运行更复杂软件（比如操作系统）的 SoC 的出现，“单片机”通常是指不运行操作系统、功能相对单一的嵌入式系统，但这不是绝对的，比如 8051 上就可以运行 RTX51 实时操作系统，它的大小只有 6kB，相比于嵌入式 Linux、Windows CE

等操作系统而言比较简单。

3. SoC（System on a Chip）

随着设计与制造技术的发展，集成电路设计从晶体管的集成发展到逻辑门的集成，现在又发展到IP的集成，即SoC（System on a Chip）设计技术。SoC可以有效地降低电子/信息系统产品的开发成本，缩短开发周期，提高产品的竞争力，是未来工业界将采用的最主要的产品开发方式。

虽然SoC一词多年前就已出现，但到底什么是SoC则有各种不同的说法。在经过了多年的争论后，专家们就SoC的定义达成了一致意见。这个定义虽然不是非常严格，但明确地表明了SoC的特征。

① 实现复杂系统功能的VLSI；

② 采用超深亚微米工艺技术；

③ 使用一个以上嵌入式CPU/数字信号处理器（DSP）；

④ 外部可以对芯片进行编程；

⑤ 主要采用第三方IP进行设计。

从上述SoC的特征来看，SoC中包含了微处理器/微控制器、存储器以及其他专用功能逻辑，但并不是包含了微处理器、存储器以及其他专用功能逻辑的芯片就是SoC，8051就集成了微处理器、存储器时部件，它不属于SoC。SoC技术被广泛认同的根本原因，并不在于SoC可以集成多少个晶体管，而在于SoC可以用较短时间设计出来，这是SoC的主要价值所在——缩短产品的上市周期。

因此，SoC更合理的定义为：SoC是在一个芯片上由于广泛使用预定制模块IP（Intellectual Property）而得以快速开发的集成电路。

本书介绍的S3C2410/S3C2440就属于SoC，它们集成了处理器、内存管理单元（MMU）、NAND Flash控制器等部件，而处理器是基于ARM公司的IP设计的。

嵌入式软件随着硬件的发展，也发生了很大的变化。在SCM、MCU阶段，嵌入式软件的编写通常由相关行业的电气、电子技术专家编写，计算机专业队伍并没有真正进入单片机应用领域。因此，电子技术应用工程师以自己习惯性的电子技术应用模式从事单片机的应用开发。这种应用模式最重要的特点是：软、硬件的底层性和随意性；对象系统专业技术的密切相关性；缺少计算机工程设计方法。

随着嵌入式处理器性能的快速提高，网络、通信、多媒体技术得以发展，很多嵌入式设备都具备收发邮件、编写文档、视听等功能，计算机专业人士开始进入嵌入式领域。这形成了明显的技术特点：基于操作系统、以网络、通信为主的“非嵌入式底层”应用——除要完成的功能比较特殊、性能比较苛刻外，嵌入式应用软件的开发已经与普通软件开发没有差别。实际上，很多基于操作系统的嵌入式应用程序就是先在PC上模拟验证，最后才移入嵌入式设备的。

以一个简单的例子加以说明：以前基于单片机编写的软件，通常是在main函数中定义一个无限循环，然后在里面查询各类输入事件，并作出相应处理，它直接操作硬件；而基于SoC的软件多是在操作系统上面运行，通过驱动程序操作硬件，这使得软件开发以分工的形式进行。

1.2 基于 ARM 处理器的嵌入式 Linux 系统

1.2.1 ARM 处理器介绍

1. ARM 的概念

嵌入式处理器种类繁多，有 ARM、MIPS、PPC 等多种架构。ARM 处理器的文档丰富，各类嵌入式软件大多（往往首选）支持 ARM 处理器，使用 ARM 开发板来学习嵌入式开发是个好选择。基于不同架构 CPU 的开发是相通的，掌握 ARM 架构之后，在使用其他 CPU 时也会很快上手。当然，作为产品进行选材时，需要考虑的因素就非常多了，这不在本书的介绍范围之内。

ARM（Advanced RISC Machine），既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。ARM 公司是 32 位嵌入式 RISC 微处理器技术的领导者，自从 1990 年创办公司以来，基于 ARM 技术 IP 核的微处理器的销售量已经超过了 100 亿。

ARM 公司并不生产芯片，而是出售芯片技术授权。其合作公司针对不同需求搭配各类硬件部件，比如 UART、SDI、I^2C 等，设计出不同的 SoC 芯片。

ARM 公司在技术上的开放性使得它的合作伙伴既有世界顶级的半导体公司，也有各类中、小型公司。随着合作伙伴的增多，也使得 ARM 处理器可以得到更多的第三方工具、制造和软件支持，又使整个系统成本降低，使新品上市时间加快，从而具有更大的竞争优势。

基于 ARM 的处理器以其高速度、低功耗、价格低等优点得到非常广泛的应用，它可以应用于以下领域：

① 为无线通信、消费电子、成像设备等产品提供可运行复杂操作系统的开放应用平台；

② 在海量存储、汽车电子、工业控制和网络应用等领域提供实时嵌入式应用；

③ 安全系统，比如信用卡、SIM 卡等。

2. ARM 体系架构的版本

ARM 体系架构的版本就是它所使用的指令集的版本。ARM 架构支持 32 位的 ARM 指令集和 16 位的 Thumb 指令集，后者使得代码的存储空间大大减小。还提供了一些扩展功能，比如 Java 加速器（Jazelle）、用以提高安全性能的 TrustZone 技术、智能能源管理（IEM，Intelligent Energy Manager）、SIMD 和 NEONTM 等技术。

还在使用的 ARM 指令集（ISA，Instruction Set Architecture）有以下版本。

（1）ARMv4。

这是当今市场上最老的版本，ARMv4 只支持 32 位的指令集，支持 32 位的地址空间。一些 ARM7 系列的处理器和 Intel 公司的 StrongARM 处理采用 ARMv4 指令集。

（2）ARMv4T。

增加了 16 位的 Thumb 指令集，它可以产生更紧凑的代码，与相同功能的 ARM 代码相

比，可以节省超过35%的存储空间，同时具备32位代码的所有优点。

（3）ARMv5TE。

在1999年，ARMv5TE版本改进了Thumb指令集，增加了一些“增强型DSP指令”，简称为E指令集。

这些指令用于增强处理器对一些典型的DSP算法的处理性能，使得音频DSP应用可以提升70%的性能。许多系统在使用微控制器来进行各类控制的同时，还需要具备数据处理能力，传统的做法要么是使用更高级的处理器（这使得成本增加），要么是使用多个处理器（这使得系统复杂度增高）。通过E指令集可以在一个普通CPU中增加DSP的功能，这在成本、性能、简化设计等方面都有优势。

（4）ARMv5TEJ。

在2000年，ARMv5TEJ版本中增加了Jazelle技术用于提供Java加速功能。相比于仅用软件实现的Java虚拟机，Jazelle技术使得Java代码的运行速度提高8位，而功耗降低80%。

Jazelle技术使得可以在一个单核的处理器上运行Java程序、已经建立好的操作系统和应用程序。

（5）ARMv6。

在2001年，ARMv6问世。它在很多方面都有改进：存储系统、异常处理，最重要的是增加了对多媒体功能的支持。ARMv6中包含了一些媒体指令以支持SIMD媒体功能扩展。SIMD媒体功能扩展为音频/视频的处理提供了优化功能，可以使音频/视频的处理性能提高4倍。

ARMv6中还引入了Thumb-2和TrustZone技术，这是两个可选的技术。之前的版本中，ARM指令和Thumb指令分别运行于不同的处理器状态，执行不同指令集的指令前要进行切换。Thumb-2技术增加了混合模式的功能，定义了一个新的32位指令集，可以运行32位指令与传统16位指令的混合代码。这能够提供“ARM指令级别的性能”与“Thumb指令级别的代码密度”。TrustZone技术在硬件上提供了两个隔离的地址空间：安全域（secure world）和非安全域（non-secure world），给系统提供了一个安全机制。

（6）ARMv7。

ARMv7架构使用Thumb-2技术，还使用了NEON技术，将DSP和媒体处理能力提高了近4倍，并支持改良的浮点运算，满足下一代3D图形、游戏物理应用以及传统嵌入式控制应用的需求。

总结 版本名中的T表示Thumb指令集，E表示增强型DSP指令，J表示Java加速器。

3. ARM处理器系列

在相同指令集下，搭配不同部件就可以组装出具有不同功能的处理器，比如有无内存管理单元、有无调试功能等。它们可以分为8个系列，系列名中有7个后缀，这些后缀可以组合，含义如下。

① T：表示支持Thumb指令集。

② D：表示支持片上调试（Debug）。

③ M：表示内嵌硬件乘法器（Multiplier）。

④ I：支持片上断点和调试点。

⑤ E：表示支持增强型 DSP 功能。

⑥ J：表示支持 Jazelle 技术，即 Java 加速器。

⑦ S：表示全合成式（full synthesizable）。

这 8 个系列中，ARM7、ARM9、ARM9E 和 ARM10 为通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。

下面简要说明它们的特点，要了解更详细的信息请参考 ARM 公司的网站。（http://www.arm.com）

（1）ARM7。

ARM7 系列处理器是低功耗的 32 位 RISC 微处理器，它主要用于对成本、功耗特别敏感的产品。最高可以达到 130MIPS，支持 Thumb 16 位指令集和 ARM 32 位指令集。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ-S。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 处理器，属于低端 ARM 处理器核。

ARM7 系列的处理器没有内存管理单元（MMU）。

（2）ARM9。

与 ARM7 相比，ARM9 的最大差别在于：有 MMU 和 Cache。它的指令执行效率较 ARM7 有较大提高，最高可达到 300MIPS。

ARM7 系列微处理器有 ARM920T 和 ARM922T 两种类型。

（3）ARM9E。

ARM9E 系列微处理器在单一的处理器内核上提供了微控制器、DSP、Java 应用系统的解决方案，极大地减少了芯片的面积和系统的复杂程度。ARM9E 系列微处理器提供了增强的 DSP 处理能力，适合于那些需要同时使用 DSP 和微控制器的应用场合。

ARM9E 系列微处理器有 ARM926EJ-S、ARM946E-S、ARM966E-S、ARM968E-S 和 ARM996HS 共 5 种类型。

（4）ARM10E。

ARM10E 系列微处理器具有更加杰出的高性能、低功耗特点，由于使用了新的体系结构，它拥有所有 ARM 系列中最高的主频。ARM10E 系列微处理器采用了一种新的省电模式，支持“64-bit load-store micro-architecture”，含有浮点运算协处理器（符合 IEEE 754 标准，支持向量运算）。

ARM10E 系列微处理器有 ARM1020E、ARM1022E 和 ARM1026EJ-S 共 3 种类型。

（5）ARM11。

ARM11 系列微处理器是 ARM 公司近年推出的新一代 RISC 处理器，它是 ARM 新指令架构——ARMv6 的第一代设计实现。ARM11 的媒体处理能力和低功耗特点特别适用于无线和消费类电子产品，其高数据吞吐量和高性能的结合非常适合网络处理应用。另外，在实时性能和浮点处理等方面 ARM11 可以满足汽车电子应用的需求。基于 AMRv6 体系结构的 ARM11 系列处理器将在上述领域发挥巨大的作用。

ARM11 系列微处理器有这 4 种类型：ARM11 MPCore，ARM1136J(F)-S，ARM1156T2(F)-S 和 ARM1176JZ(F)-S。

（6）Cortex。

Cortex 系列处理器是基于 ARMv7 架构的，分为 Cortex-A、Cortex-R 和 Cortex-M 共 3 类。Cortex-A 为传统的、基于虚拟存储的操作系统和应用程序而设计，支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-R 针对实时系统设计，支持 ARM、Thumb 和 Thumb-2 指令集；Cortex-M 为对价格敏感的产品设计，只支持 Thumb-2 指令集。

（7）SecurCore。

SecurCore 系列微处理器专为安全需要而设计，提供了完善的 32 位 RISC 技术的安全解决方案，因此，SecurCore 系列微处理器除了具有 ARM 体系结构的低功耗、高性能的特点外，还具有其独特的优势，即提供了对安全解决方案的支持。

SecurCore 系列微处理器有如下类型：SecurCoreSC100、SecurCore SC200。

（8）OptimoDE Data Engines。

这是一个新的 IP 核，针对高性能的嵌入式信号处理应用而设计。

另外，Intel 公司的 StrongARM、Xscale 系列处理器也属于 ARM 架构。Intel StrongARM 处理器是便携式通信产品和消费类电子产品的理想选择，已成功应用于多家公司的掌上电脑系列产品。Xscale 处理器是基于 ARMv5TE 体系结构的解决方案，是一款全性能、高性价比、低功耗的处理器。它支持 16 位的 Thumb 指令和 DSP 指令集，已使用在数字移动电话、个人数字助理和网络产品等场合。

本书使用 S3C2410、S3C2440 芯片，它们的处理器都属于 ARM920T 系列，版本为 ARMv4T。

1.2.2 在嵌入式系统中选择嵌入式 Linux 的理由

随着技术的发展及人们需求的增加，各种消费类电子产品的功能越来越强大，随身携带的电子设备变得“等同于 PC”：上面有键盘、触摸屏、LCD 等输入、输出设备，可以观看视频、听音乐，可以浏览网站、接收邮件，可以查看、编辑文档等。在工业控制领域，系统级芯片（SoC）以更低廉的价格提供了更丰富的功能，使得一个嵌入式系统可以同时完成更多的控制功能。

当系统越来越大、应用越来越多，使用操作系统很有必要。操作系统的作用有：统一管理系统资源、为用户提供访问硬件的接口、调度多个应用程序、管理文件系统等。在嵌入式领域可以选择的操作系统有很多，比如：嵌入式 Linux、VxWorks、Windows CE、μC/OS-II 等。

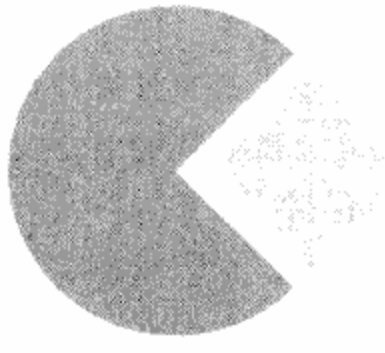
- VxWorks 是美国 WindRiver 公司开发的嵌入式实时操作系统。单就性能而言，它是非常优秀的操作系统，具有可裁剪的微内核结构、高效的任务管理、灵活的任务间通信、微秒级的中断处理，支持 POSIX 1003.1b 实时扩展标准，支持多种物理介质及标准、完整的 TCP/IP 网络协议等。缺点是它支持的硬件相对较少，并且源代码不开放，需要专门的技术人员进行开发和维护，并且授权费比较高。
- Windows CE 是微软公司针对嵌入式设备开发的 32 位、多任务、多线程的操作系统。它支持 x86、ARM、MIPS、SH 等架构的 CPU，硬件驱动程序丰富，比如支持 WiFi、USB 2.0 等新型设备，并具有强大的多媒体功能；可以灵活裁剪以减小系统体积；与 PC 上的 Windows 操作系统相通，开发、调试工具使用方便，应用程序的开发流程与 PC 上的 Windows 程序的开发流程相似，就开发的便利性而言（特别是对于习惯在

Windows 下开发的程序员），Windows CE 是最好的，但是，其源代码没有开放（目前开放了一小部分），开发人员难以进行更细致的定制；占用比较多的内存，整个系统相对庞大；版权许可费用也比较高。

- μC/OS-II 是 Micrium 公司开发的操作系统，可用于 8 位、16 位和 32 位处理器。可裁剪，对硬件要求较低；可以运行最多 64 个任务；调度方式为抢占式，即总是运行最高优先级的就绪任务。用户可以获得μC/OS-II 的全部代码，但它不是开放源码的免费软件，作为研究和学习，可以通过购买相关书籍获得源码，用于商业目的时，必须购买其商业授权。相对于其他按照每个产品收费的操作系统，μC/OS-II 采用一次性的收费方式，价格低廉。需要说明的是，μC/OS-II 仅是一个实时内核，用户需要完成其他更多的工作，比如编写硬件驱动程序、实现文件系统操作（使用文件的话）等。
- Linux 是遵循 GPL 协议的开放源码的操作系统，使用时无需交纳许可费用。内核可任意裁剪，几乎支持所有的 32 位、64 位 CPU；内核中支持的硬件种类繁多，几乎可以从网络上找到所有硬件驱动程序；支持几乎所有网络协议；有大量的应用程序可用，从编译工具、调试工具到 GUI 程序，几乎都有遵循 GPL 协议的相关版本；有庞大的开发人员群体，有数量众多的技术论坛，大多问题都可以得到快速而免费的解答。

Linux 的缺点在于实时性，虽然 2.6 版本的 Linux 在实时性方面有较大改进，但是仍无法称为实时操作系统。有不少变种 Linux 在实时性方面做了很大改进，比如 RTLinux 达到了硬实时，TimeSys Linux 提高了实时性。这些改进的 Linux 版本既有遵循 GPL 协议的免费版本，也有要付费的商业版本。

正是由于 Linux 开放源代码、易于移植、资源丰富、免费等优点，使得它在嵌入式领域越来越来流行。更重要的一点，由于嵌入式 Linux 与 PC Linux 源于同一套内核代码，只是裁剪的程度不一样，这使得很多为 PC 开发的软件再次编译之后，可以直接在嵌入式设备上运行，这使得软件资源“极大”非富，比如各类实用的函数库、小游戏等。



第 2 章　嵌入式 Linux 开发环境构建

本章目标

- 了解嵌入式 Linux 开发的交叉开发模式
- 搭建硬件、软件开发环境
- 掌握制作工具链的方法

2.1　硬件环境构建

2.1.1　主机与目标板结合的交叉开发模式

开发 PC 机上的软件时，可以直接在 PC 机上编辑、编译、调试软件，最终发布的软件也在 PC 机上运行。对于嵌入式开发，最初的嵌入式设备是一个空白的系统，需要通过主机为它构建基本的软件系统，并烧写到设备中；另外，嵌入式设备的资源并不足以用来开发软件。所以需要用到交叉开发模式：在主机上编辑、编译软件，然后在目标板上运行、验证程序。

主机指 PC 机，目标板指嵌入式设备，在本书中，目标板就是 S3C2410、S3C2440 开发板。

对于 S3C2410、S3C2440 开发板，进行嵌入式 Linux 开发时一般可以分为以下 3 个步骤。

（1）在主机上编译 Bootloader，然后通过 JTAG 烧入单板。

通过 JTAG 接口烧写程序的效率非常低，它适用于烧写空白单板。为方便开发，通常选用具有串口传输、网络传输、烧写 Flash 功能的 Bootloader，它可以快速地从主机获取可执行代码，然后烧入单板，或者直接运行。

（2）在主机上编译嵌入式 Linux 内核，通过 Bootloader 烧入单板或直接启动。

一个可以在单板上运行的嵌入式 Linux 内核是进行后续开发的基础，为方便调试，内核应该支持网络文件系统（NFS），即将应用程序放在主机上，单板启动嵌入式 Linux 内核后，通过网络来获取程序，然后运行。

（3）在主机上编译各类应用程序，单板启动内核后通过 NFS 运行它们，经过验证后再烧入单板。

烧写、启动 Bootloader 后，就可以通过 Bootloader 的各类命令来下载、烧写、运行程序了。启动嵌入式 Linux 后，也是通过执行各种命令来启动应用程序的。怎么输入这些命令、查看命令运行的结果呢？一般通过串口来进行输入/输出。所以交叉开发模式中，主机与目标板通常需要 3 种连接：JTAG、串口、网络，如图 2.1 所示。

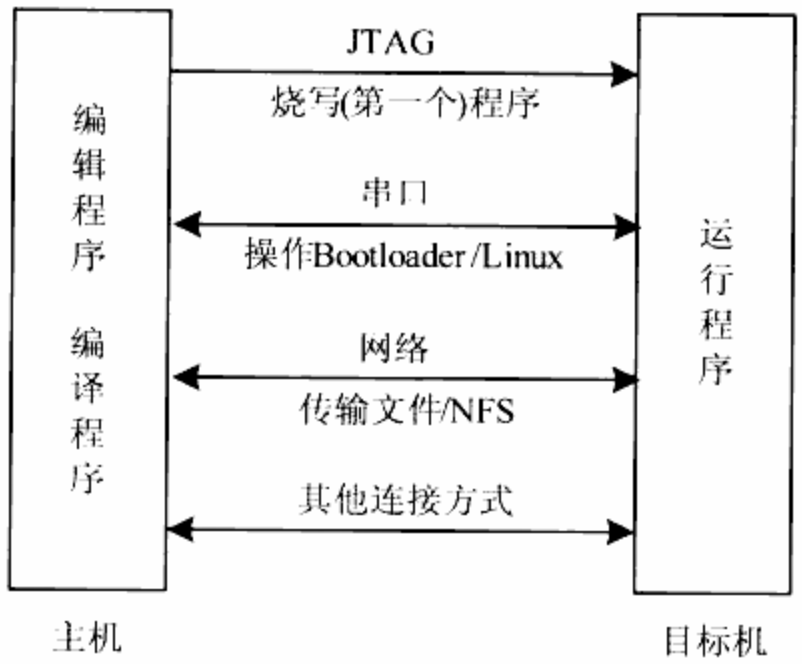


图 2.1 交叉开发模式

注意 JTAG 接口本来是用来调试程序的，但是本书中是通过它来烧写程序。

2.1.2 硬件要求

1. 主机要求

一般的 PC 就可以用来进行嵌入式 Linux 开发，它应该满足以下要求。

① 有一个 25 针的并口接口，它用来接 JTAG 连线；

② 有一个 9 针的 RS-232 串行接口；

③ 支持网络；

④ 至少 20GB 的硬盘。

对于台式机，这 3 项要求一般都能满足；对于笔记本电脑，如果没有并口则需要使用其他电脑来烧写程序，如果没有串口，则可以使用 USB-串口转换器。幸运的是，如果不是调试 Bootloader，使用 JTAG 接口的次数会很少：只需要烧写一次 Bootloader——当它启动后，就可以通过串口或网络下载程序，然后烧入单板。

2. 目标板要求

本书的例子适用于两种开发板：S3C2410 或 S3C2440。市场上 S3C2410、S3C2440 开发板的原理图、配置基本相同，即使在细节上有所差异，理解代码后稍作修改即可使用。

开发板上对于 S3C2410/S3C2440 的每个硬件部件（UARG、NAND Flash 控制器、I^2C 接口等）基本都外接了相关器件，有如下部件。

① 64MB SDRAM；
② 1MB NOR Flash；
③ 64MB NAND Flash；
④ 两个网卡（10MB和100MB）；
⑤ 5个串口（内置3个，外扩2个）；
⑥ 音频输入输出（本书没有介绍）；
⑦ 2.5寸IDE接口；
⑧ 标准SD/MMC卡座；
⑨ 4个GPIO按键/4个GPIO按键；
⑩ 外接I^2C接口的实时时钟（RTC）芯片。

硬件开发环境搭建很简单，将主机与目标板通过JTAG、串口线（接单板上的串口0）、网线（接单板上的网卡0）连接起来，将各类设备连接到目标板上去即可。

2.2 软件环境构建

2.2.1 主机Linux操作系统的安装

1. 在Windows上安装虚拟机

本书基于Ubuntu 7.10进行开发，它是一个很容易安装和使用的Linux发行版。光盘映象文件的下载地址为http://relenses.ubuntu.com/7.10。

安装方法有好几种：将映象文件刻录成光盘后安装，通过网络安装等。不熟悉Linux的读者可以通过VMware虚拟机软件使用映象文件安装，这样可以在Windows中使用Linux（反过来也是可以的），安装Linux后，再使用VMware安装windows，这样就可以在Linux中同时使用Windows。

这几种安装方法基本相同，下面介绍在Windows中通过VMware来安装Linux的方法。不管是哪种方法，都建议单独使用一个分区来存放本书所涉及的源码、编译结果，这样可以避免当系统出错、系统重装时破坏学习成果。

本书使用3个分区：1GB的交换分区（swap）、5GB的根分区（root）、15GB的工作分区（work）。swap分区被用来暂时存储数据，它可以提高系统性能；root分区被用来存放整个Linux系统；work分区的内容来自本书光盘，以后将在这个分区上编辑、编译、调试软件。

从VMware的官方网站http://www.vmware.com下载到VMware工具，安装后，参照以下方法安装、设置Linux。

建立一个虚拟机器需要指定硬盘、内存、网络。在VMware中可以使用实际的硬盘，也可以使用文件来模拟硬盘。本书使用文件来模拟硬盘：一个表示交换分区（swap），一个表示根分区（root），一个表示工作分区（work）。依照下面的一系列图形就能建立虚拟机。

① 启动 VMware，如图 2.2 所示。

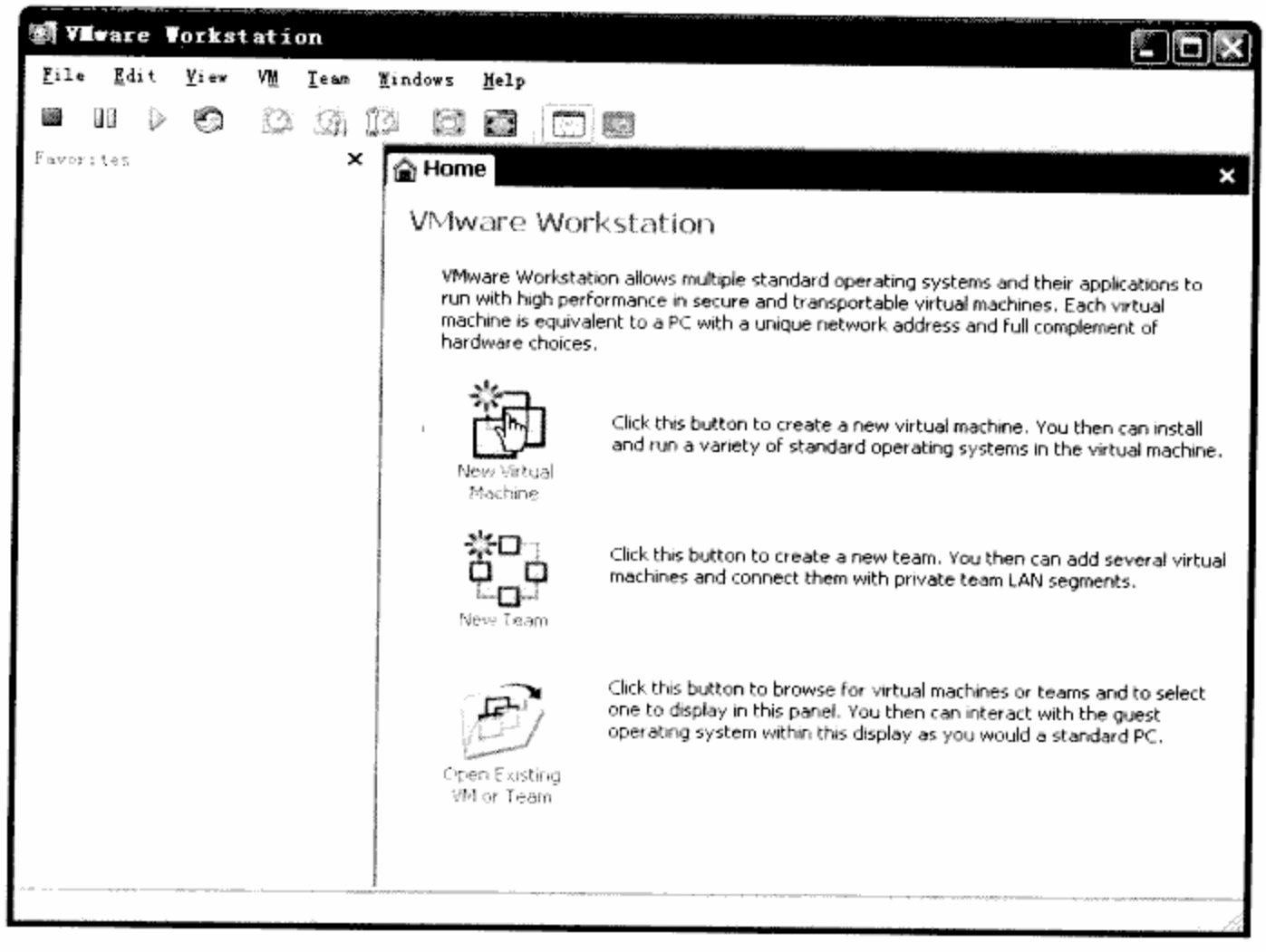


图 2.2 启动 VMware

② 在后续界面中使用默认选项，直到出现如图 2.3 所示的界面，选择“Custom”自己定制虚拟机。

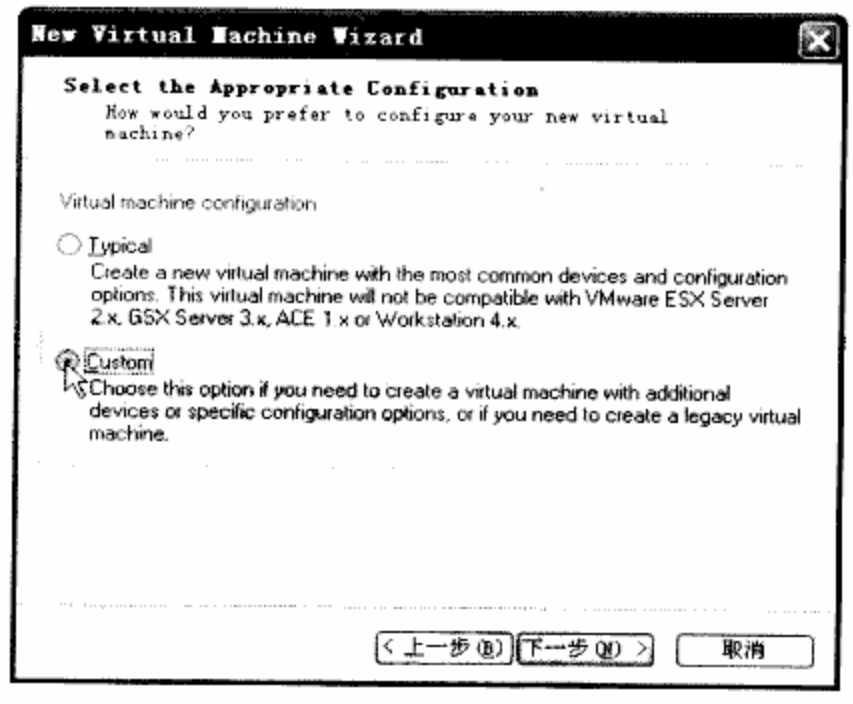


图 2.3 选择定制虚拟机

③ 选择虚拟机的格式，使用默认选项即可，如图 2.4 所示。

④ 在 Windows 中使用 VMware 安装 Linux，Windows 被称为“Host Operatins System”（主机操作系统），Linux 被称为“Gest Operatins System”（客户操作系统）。选择 Linux 作为客户操作系统，版本为“Other Linux 2.6.x kernel”，如图 2.5 所示。

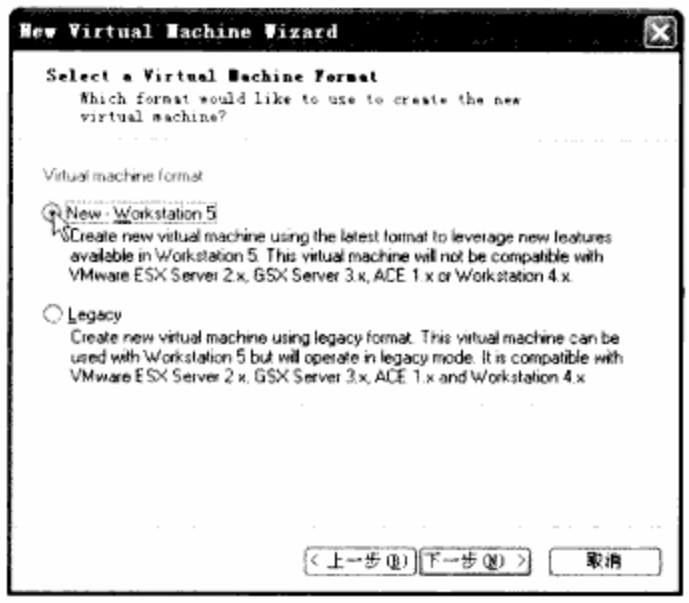


图 2.4　选择虚拟机的格式

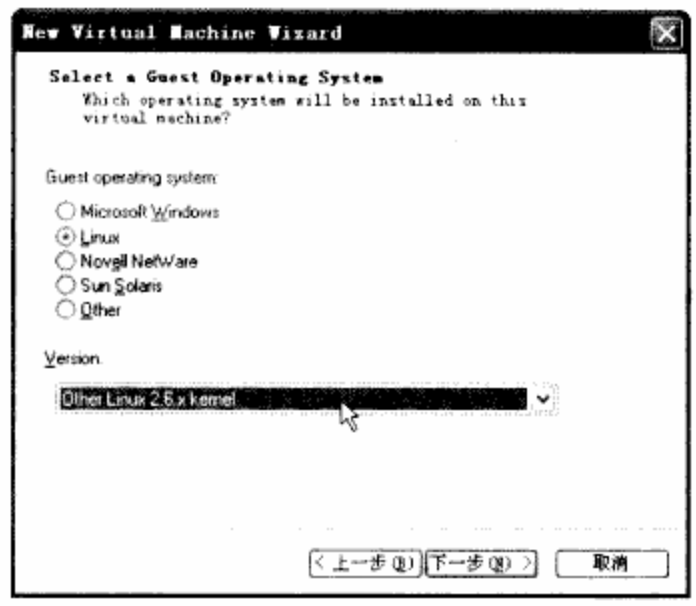


图 2.5　选择客户操作系统

⑤ 设置虚拟机的名称及存储位置，如图 2.6 所示。

⑥ 指定虚拟机的内存容量，在图 2.7 所示的界面中有推荐值及取值范围。

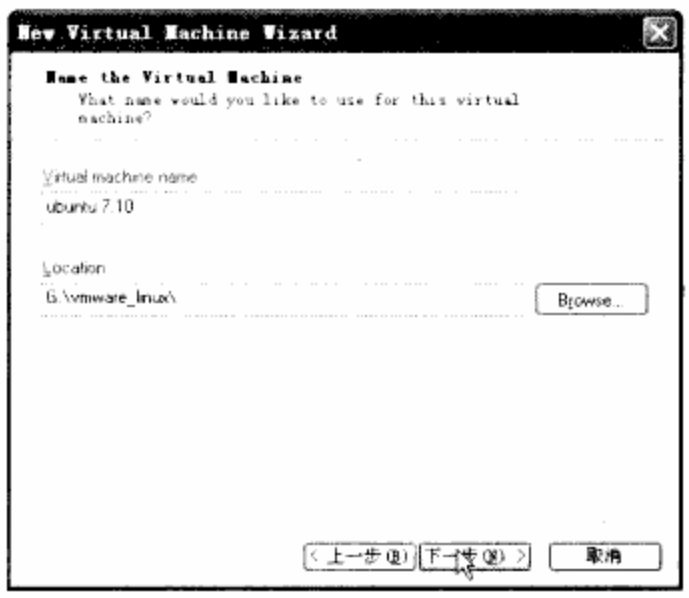


图 2.6　设置虚拟机的名字及存储位置

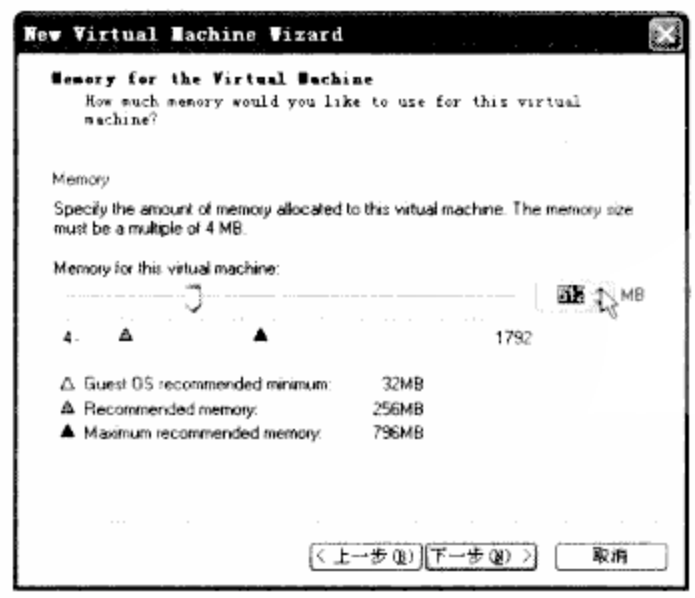


图 2.7　指定虚拟机的内存容量

⑦ 指定虚拟机的网络连接类型，一般使用桥接方式（bridged networking），如图 2.8 所示。安装完毕后可以再进行修改。

⑧ 选择“I/O Adapter”，使用默认值，如图 2.9 所示。

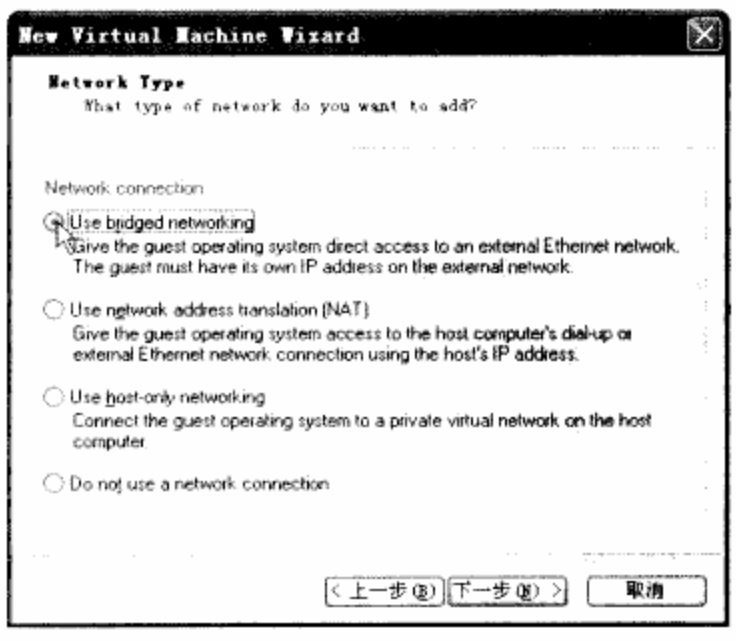


图 2.8　指定虚拟机的网络连接类型

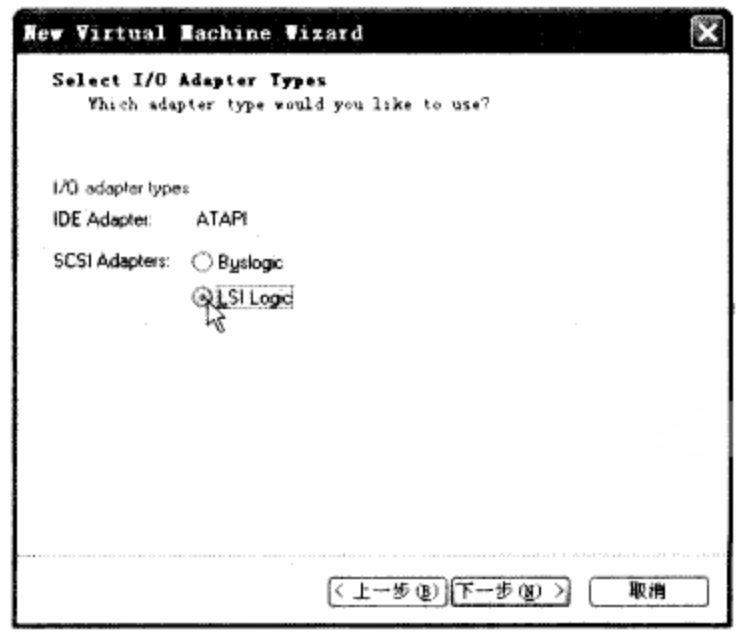


图 2.9　选择“I/O Adapter”

⑨ 图 2.10～图 2.13 所示为创建虚拟硬盘的过程，在创建完虚拟机后，还会重复这 4 个步骤创建另外两个虚拟硬盘。在图 2.12 中，“Split disk into 2GB files 选项表示使用多个 2GB 的文件来表示一个很大的虚拟硬盘。如果 Windows 的碰盘格式为 FAT32，因为它支持的最大文件只有 4GB，所以要选择这个选项；如果是 NTFS 格式，则无需选择这个选项。

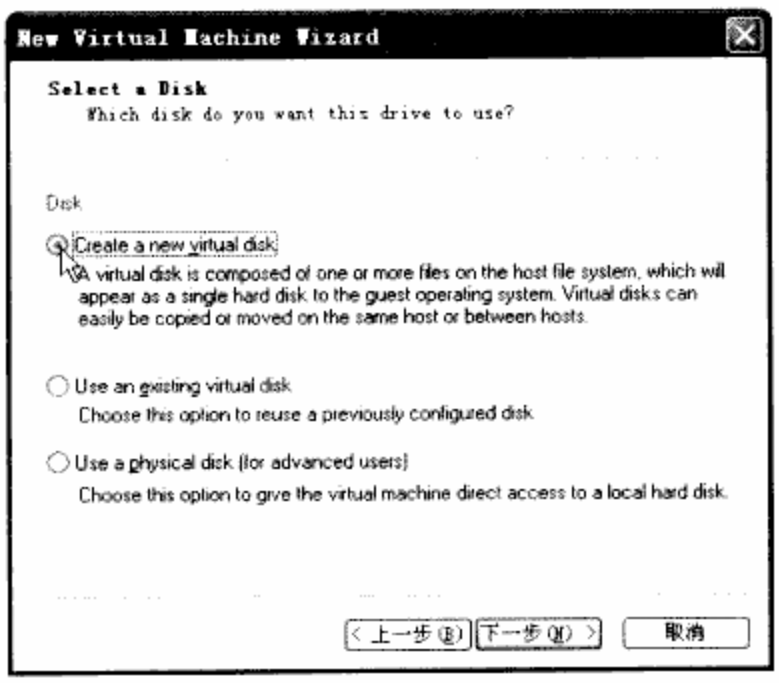


图 2.10　选择创建新的虚拟硬盘

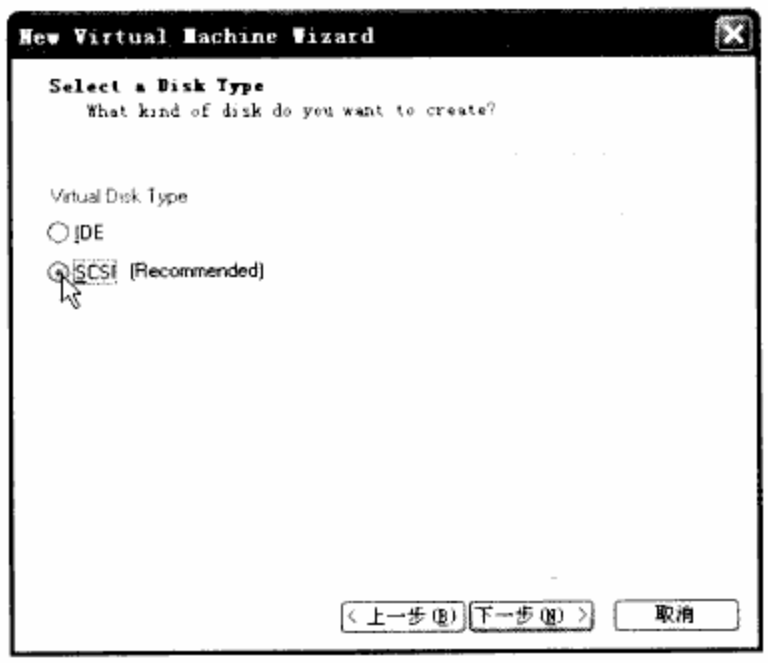


图 2.11　选择硬盘（使用默认类型）

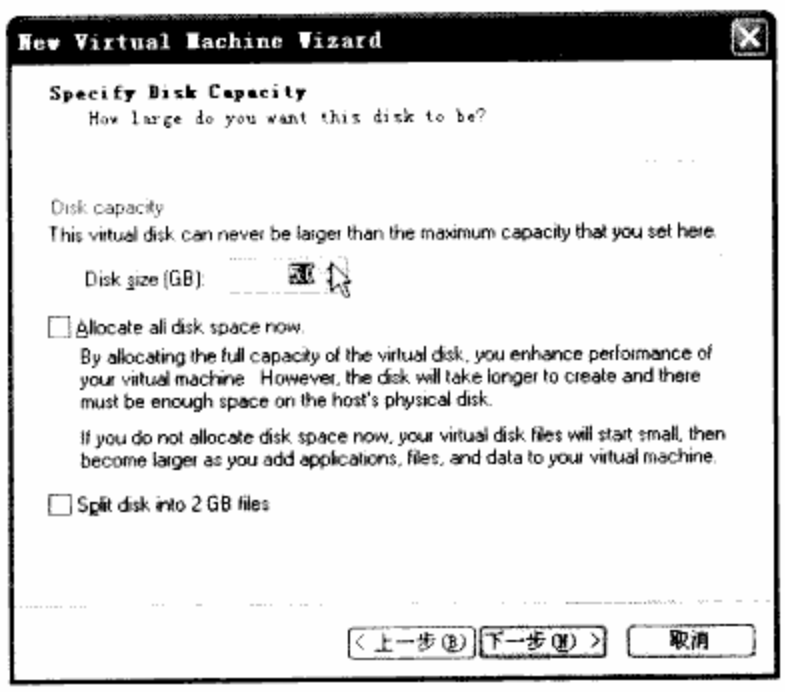


图 2.12　指定硬盘容量（这个硬盘将用来安装 Linux，推荐大小为 5GB）

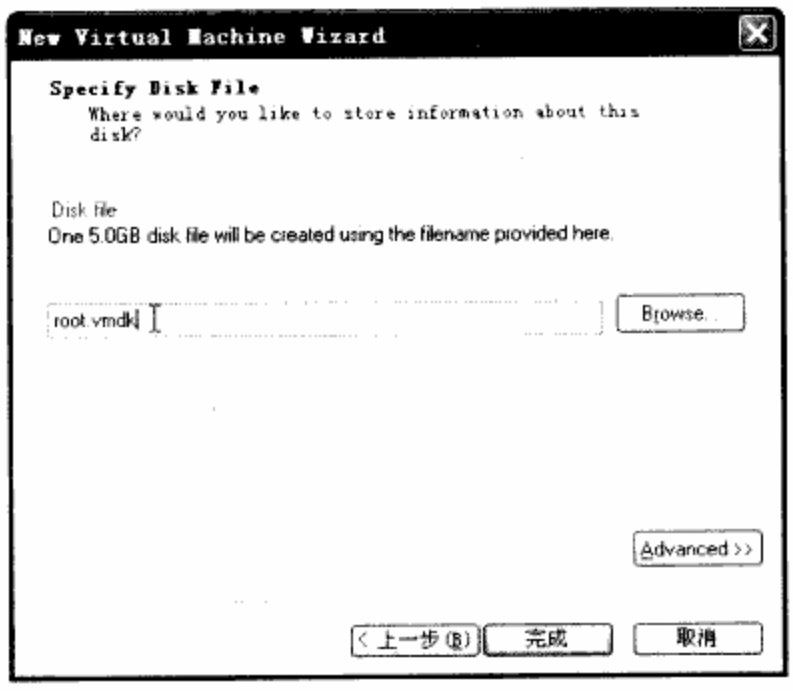


图 2.13　设置虚拟硬盘的名字（在 Windows 下将新建一个文件来表示这个虚拟硬盘）

⑩ 单击“完成”按钮后，就创建了一个虚拟机，得到如图 2.14 所示的界面。

还要创建两个硬盘，单击图 2.14 中的“Edit virtual machine settings”进行设置。参照图 2.14～图 2.16 及步骤⑨连续增加两个虚拟硬盘，一个容量为 15GB，命名为 work.vmdk（用来作为工作硬盘）；另一个容量为 1GB，命名为 swap.vmdk（在上面创建交换分区）。

以后的步骤就和前面的步骤⑨一样了。

2. 在虚拟机上安装 Linux

本书使用 Ubuntu 7.10 的光盘文件 ubuntu-7.10-desktop-i386.iso 进行安装。下面简单介绍

关键步骤，其他步骤可以参看安装时出现的说明。

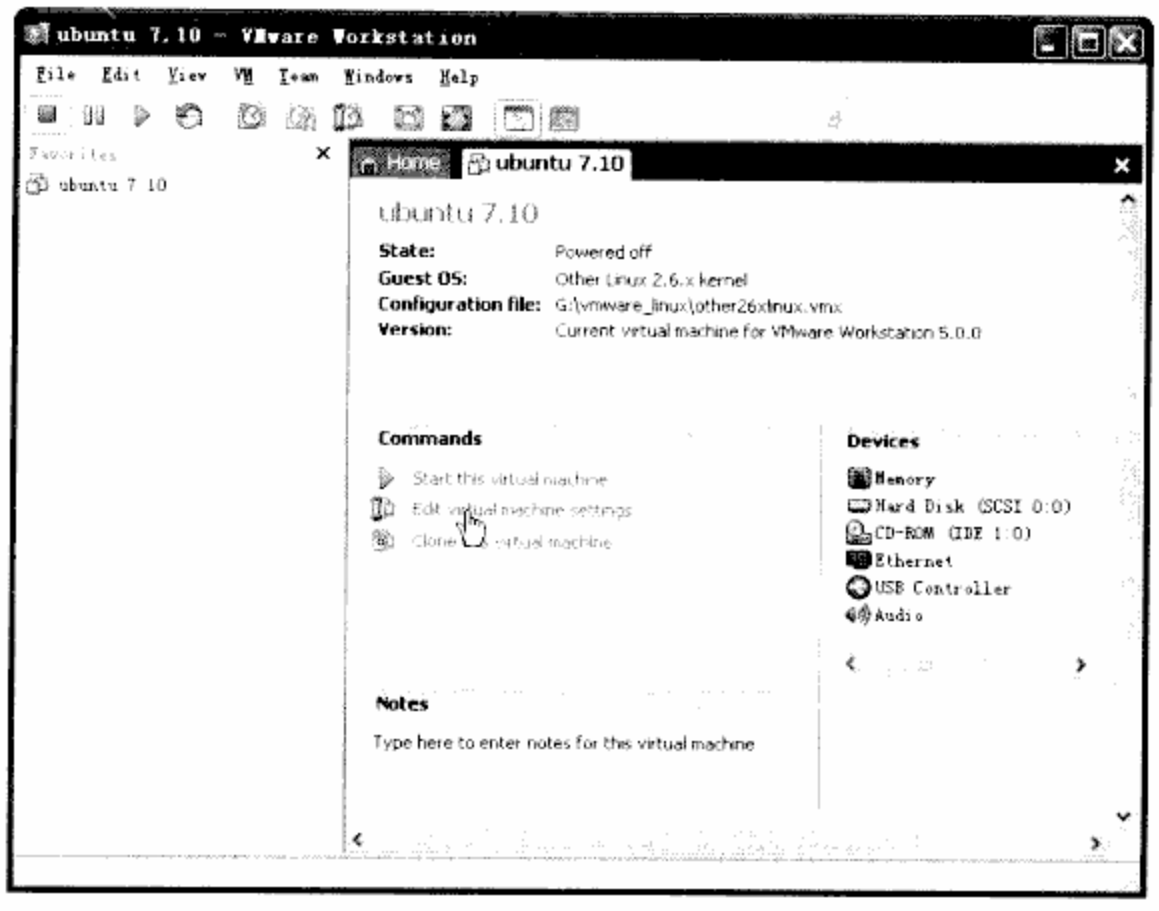


图 2.14　修改虚拟机的设置

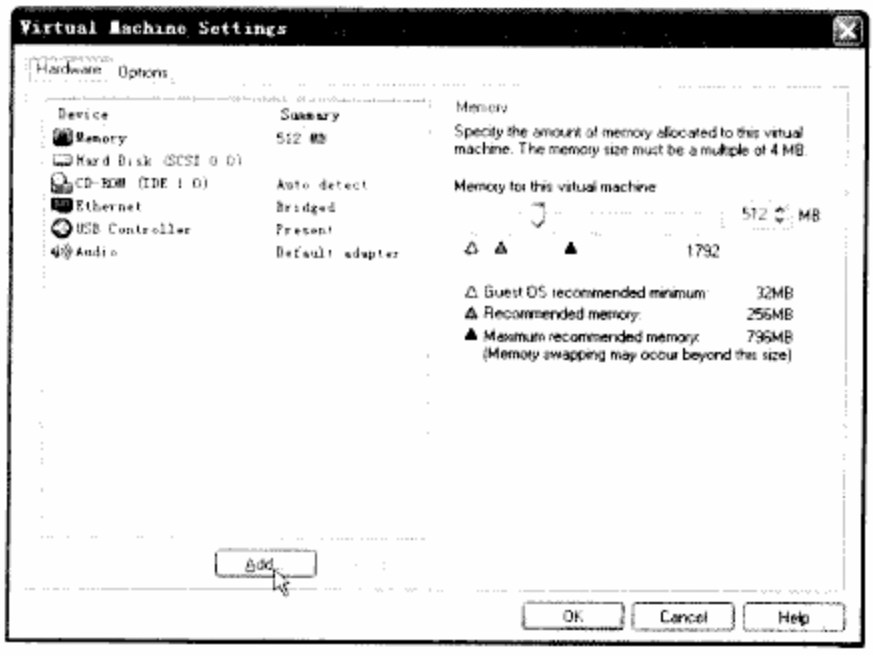


图 2.15　增加新硬件

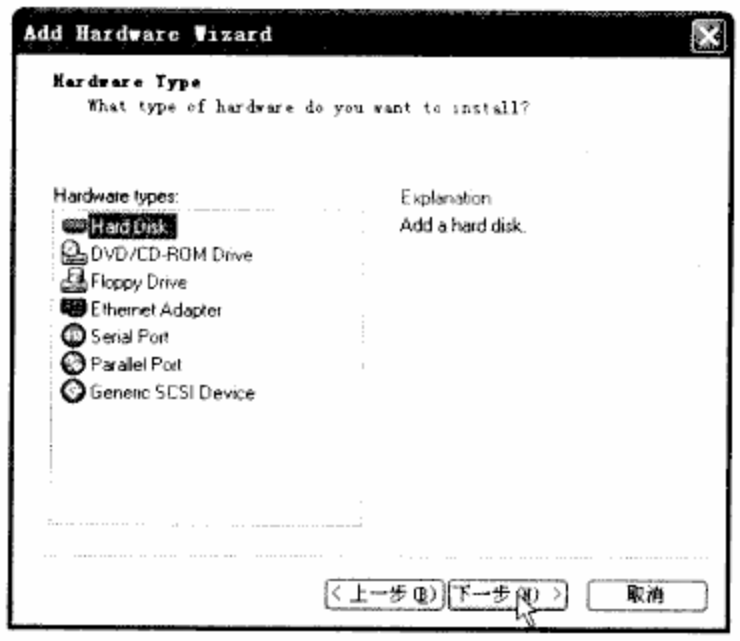


图 2.16　选择增加硬盘

（1）在虚拟机上使用光盘文件。

如图 2.17 所示，进入虚拟机的编辑界面，选中“CD-ROM”，在右边的界面中选择“Connect at power on”（表示开启虚拟机时就连接光盘）；然后选择“Use ISO image”，如果有实际的光盘，可以选择“Use physical drive”（在安装随书光盘中的代码时就是这样设置的）。

（2）启动虚拟机，它使用前面设置的光盘文件启动，这时候即可开始安装 Linux。

在虚拟机启动后，桌面有个名为“install”的图标，双击它进行安装。前面几个步骤是选择语言、键盘等，使用默认设置，先不要使用中文（以后远程登录时，中文不好显示）。当出现如图 2.18 所示的界面时，选择“Manual”。

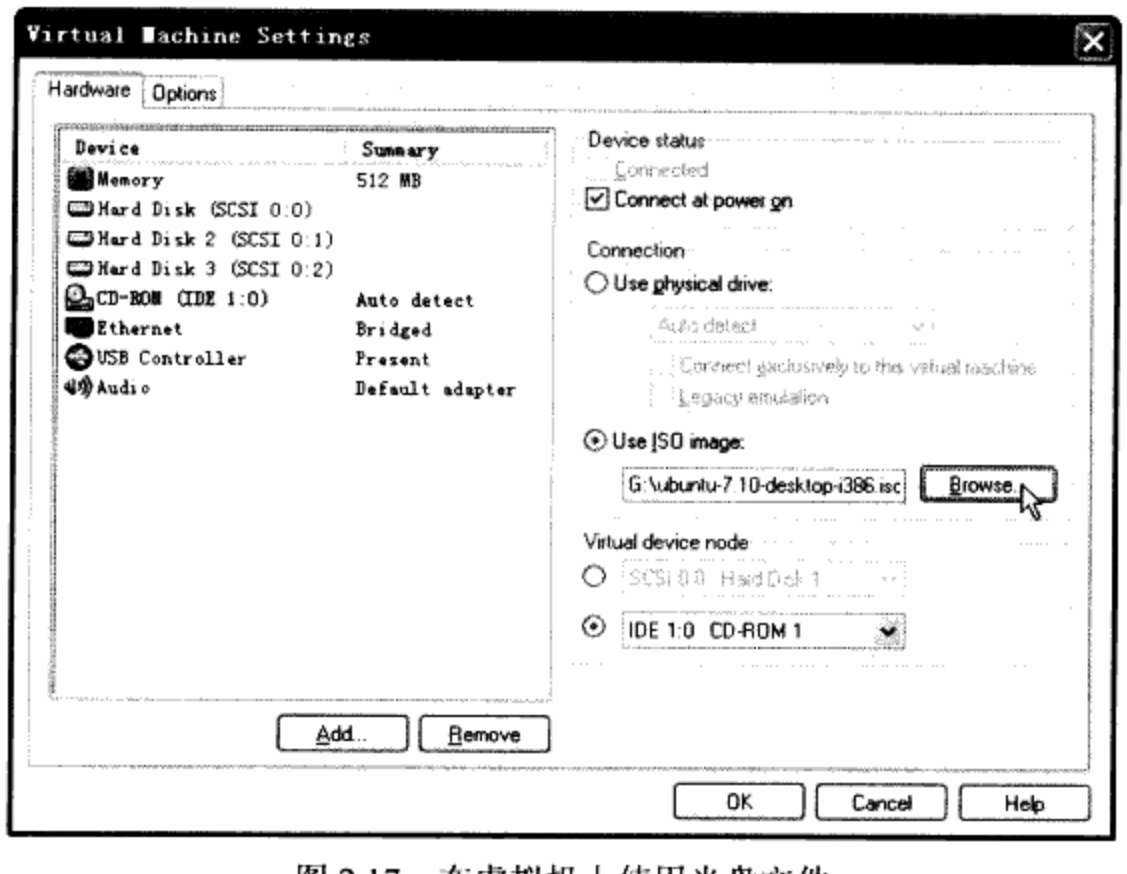


图 2.17 在虚拟机上使用光盘文件

注意 在 VMware 的操作系统中，要将鼠标光标释放出来（回到 Windows 中），按“Ctrl+Alt”键即可。

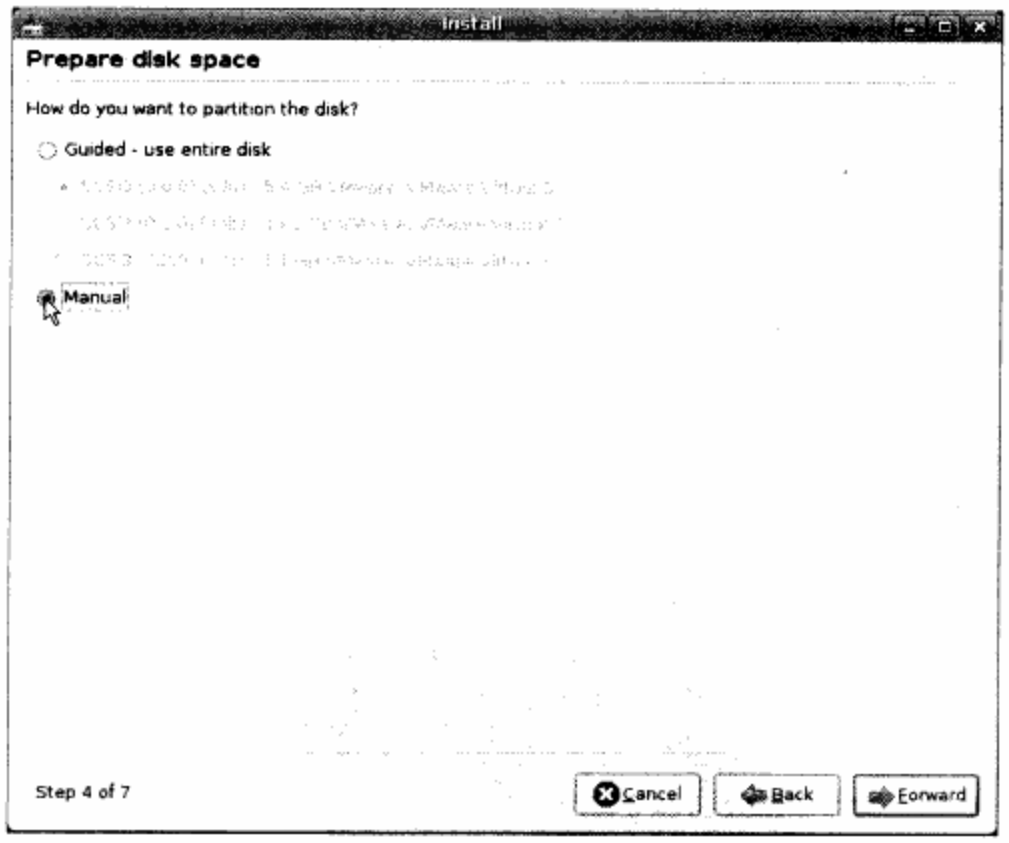


图 2.18 通过“手动”来使用硬盘

（3）在紧接着出现的界面中对硬盘进行分区、选择文件格式、选择挂载点。

本书设置的虚拟机使用 3 个虚拟硬盘，每个硬盘只划分一个分区。设置其中容量为 5GB 的分区的挂载点为“/”，在里面安装 Linux 系统；设置容量为 15GB 的分区的挂载点为“/work”，以后的代码、编译结果都放在/work 目录下；设置容量为 1GB 的分区为 swap 分区，它不需要设置挂载点，也不需要设置文件格式。设置的结果如图 2.19 所示。

（4）然后在后续的界面中使用默认值，安装程序会进行格式化虚拟硬盘等操作。当出现

如图 2.20 所示的界面时，在里面设置用户名及密码。

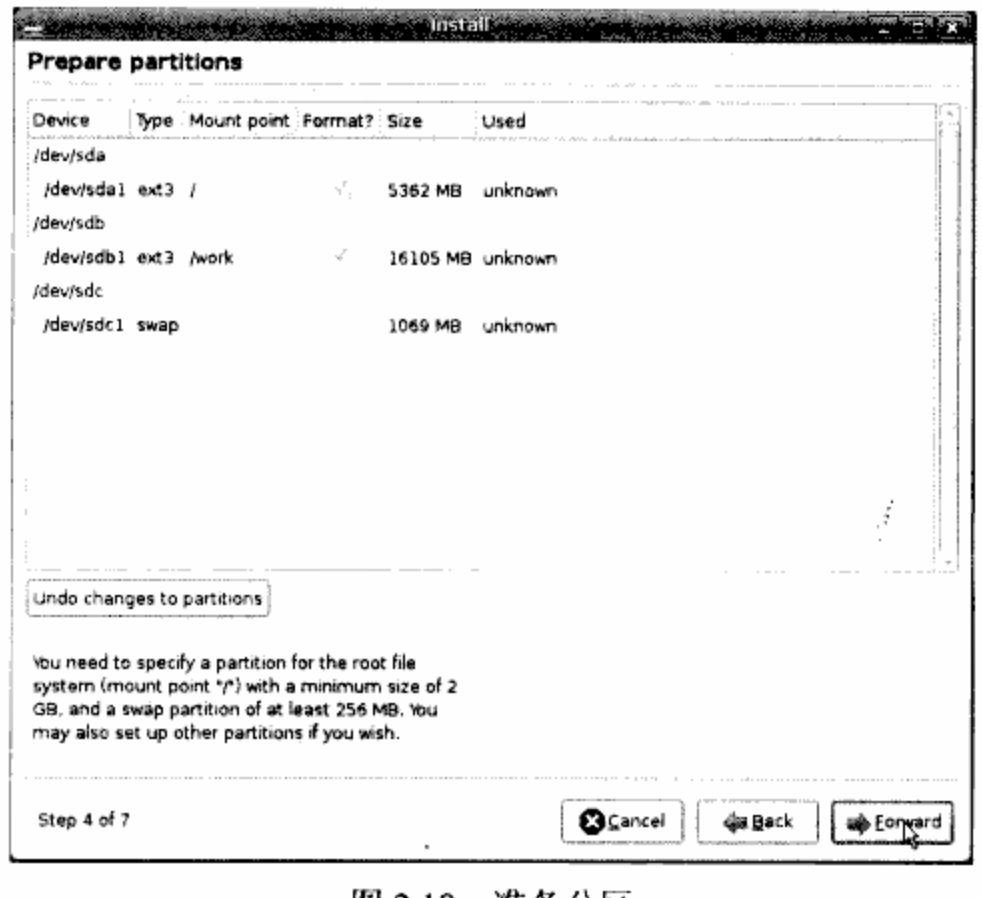


图 2.19　准备分区

（5）开始安装系统，当安装完成时，出现如图 2.21 所示的界面。

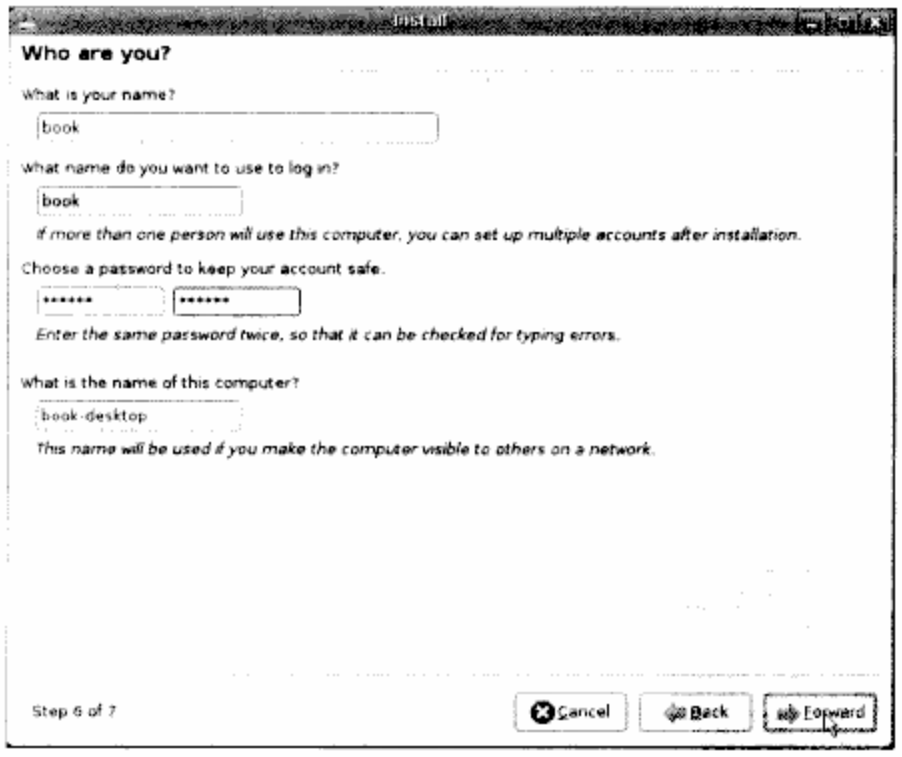


图 2.20　设置用户名和密码

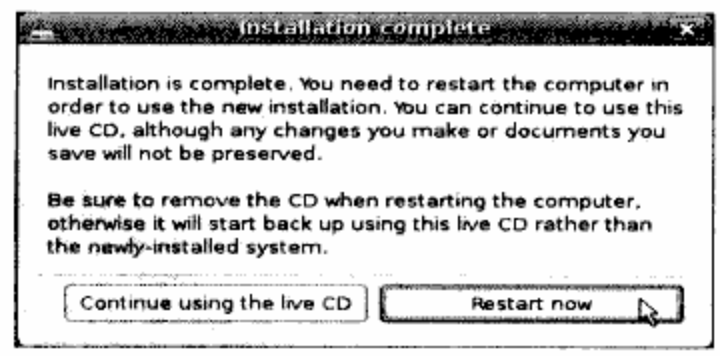


图 2.21　安装结束

如果从 VMware 的菜单“VM”→“Setting...”进入虚拟机的设置界面时，在“CD-ROM”的设置界面去掉“Connect at power on”选项（参考图 2.17），此时单击图 2.21 所示的“Restart now”按钮即可（如果不能重启，直接关闭 VMware 后再启动）。

2.2.2　主机 Linux 操作系统上网络服务的配置与启动

下面配置 Linux，启动 FTP、SSH、NFS 这 3 个服务。如果不是通过远程登录 Linux，而是直接在 Linux 中进行开发，则 FTP、SSH 这两个服务不用开启。

1. 设置网络

这涉及 3 方面的设置：主操作系统 Windows、VMware、客户操作系统 Linux。

本书假设：主操作系统 Windows 的 IP 为 192.168.1.11，客户操作系统 Linux 的 IP 为 192.168.1.57。

VMware 提供 4 种网络连接方式：网桥网络（Bridged）、网络地址翻译网络（NAT）、仅为主机网络（Host-only）和客户网络。常用的方式是前两种，网桥网络需要接上网线才可使用，当主机与目标板间需要进行网络通信时使用这种方式，它相当于 3 台处于同一网段的计算机：主机（Windows）、虚拟机（Linux）、目标板。没有接网线时，可以使用 NAT 网络在主操作系统 Windows 与客户操作系统 Linux 间进行通信。

（1）设置客户操作系统 Linux 的 IP 地址。

选择 Linux 的启动栏执行“System”→“Administration”→“Network”命令，在对话框中选中“Connections”选项卡的“Wired connection”项，然后单击“Properties”按钮。设置 eth0 的 IP 为 192.168.1.57，网关为 192.168.1.2，然后重新启动即可，如图 2.22 所示。

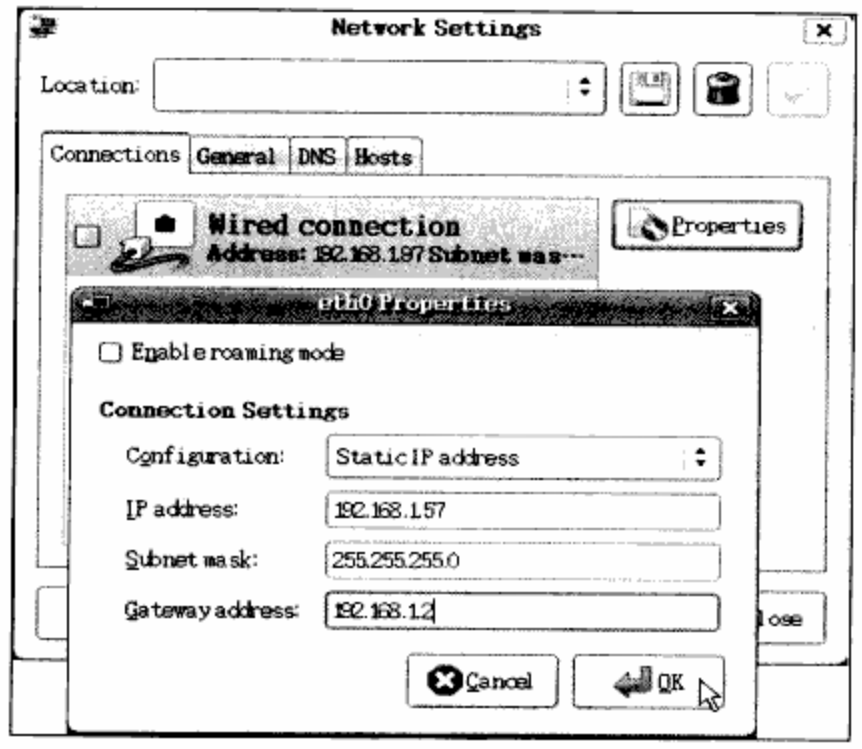


图 2.22 设置 Linux 的 IP 地址

（2）使用网桥网络时，设置主操作系统 Windows 和 VMware。

使用网桥网络时，主操作系统 Windows 和 Linux 的 IP 必须属于同一个网段。

- 将主操作系统 Windows 的网卡 IP 设为 192.168.1.11。
- 在 VMware 中，执行“VM”→“Setting”命令，然后参照图 2.23 设置以使用网桥网络（“Connect at power on”一定要选上）。

（3）使用 NAT 网络时，设置 VMware。

使用 NAT 网络时，首先要确保主操作系统 Windows 的 IP 和 Linux 的 IP 不在同一个网段。由于 Linux 的 IP 已经设为 192.168.1.57，需要修改 Windows 的 IP 让它不处于 192.168.1.x 网段。

然后分 3 步设置：修改虚拟机的设置使它使用 NAT 网络，修改 NAT 网络的 IP 地址范围，

设置 NAT 网络的网关地址。VMware 提供 9 个虚拟网卡：VMnet0～VMnet8，VMnet8 用于 NAT 网络，其他的在本书中都不需要自己设置。

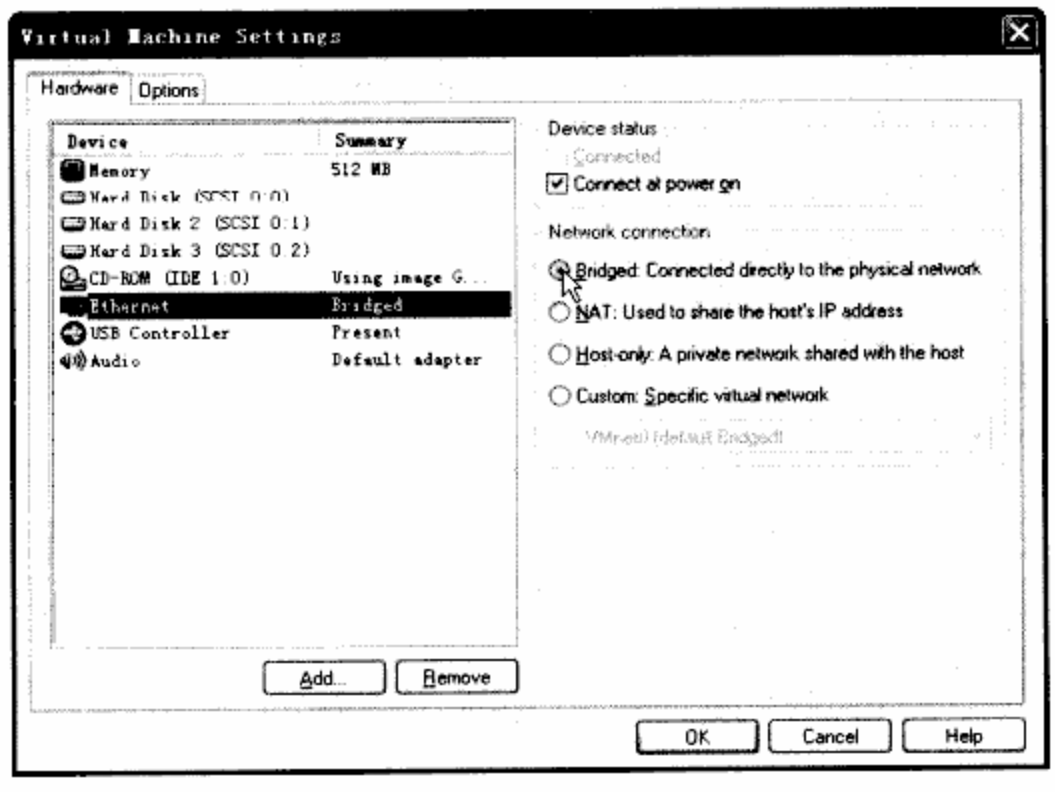


图 2.23 设置虚拟机以使用网桥网络

- 设置虚拟机以使用 NAT 网络。

选择 VMware 的菜单“VM”→“Setting”，然后参照图 2.24 进行设置（“Connect at power on”一定要选上）。

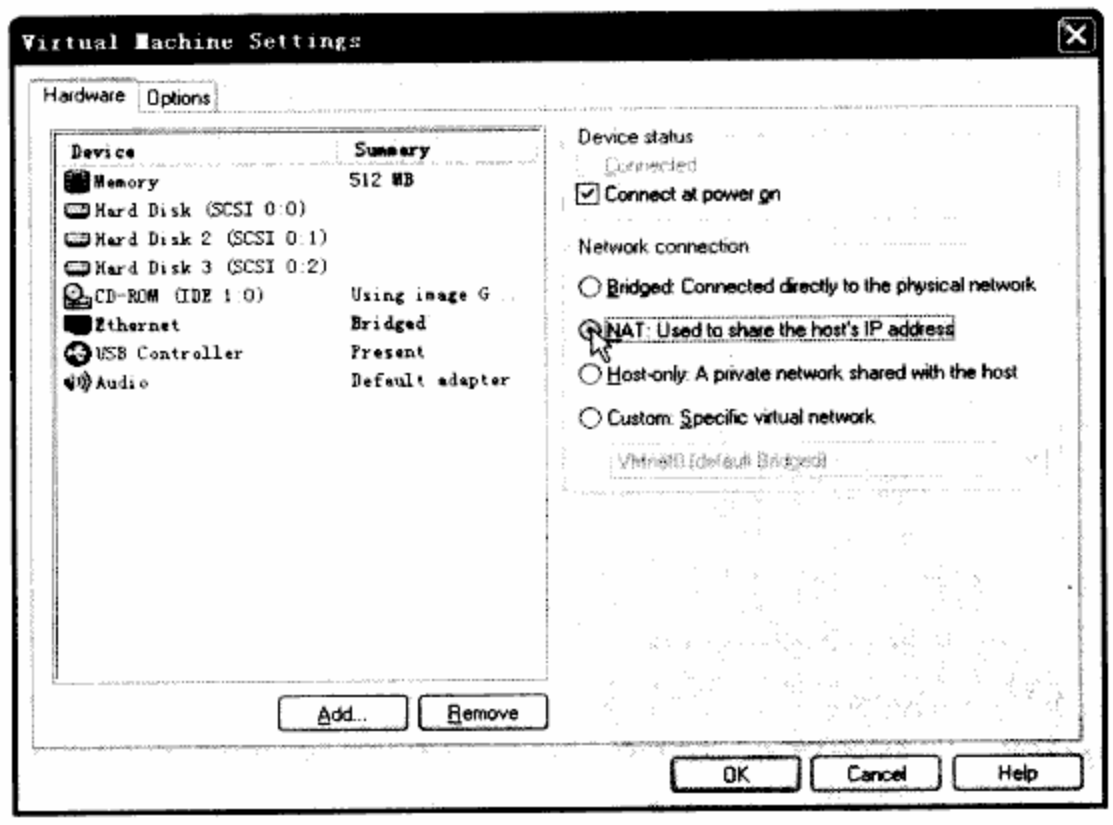


图 2.24 设置虚拟机以使用 NAT 网络

- 修改 NAT 网络的 IP 地址范围。

选择 VMware 的菜单“Edit”→“Virtual Network Setting...”，然后参照图 2.25 设置。

- 设置 NAT 网络的网关地址。

选择 VMware 的菜单“Edit”→“Virtual Network Setting...”，然后参照图 2.26 进行

设置。

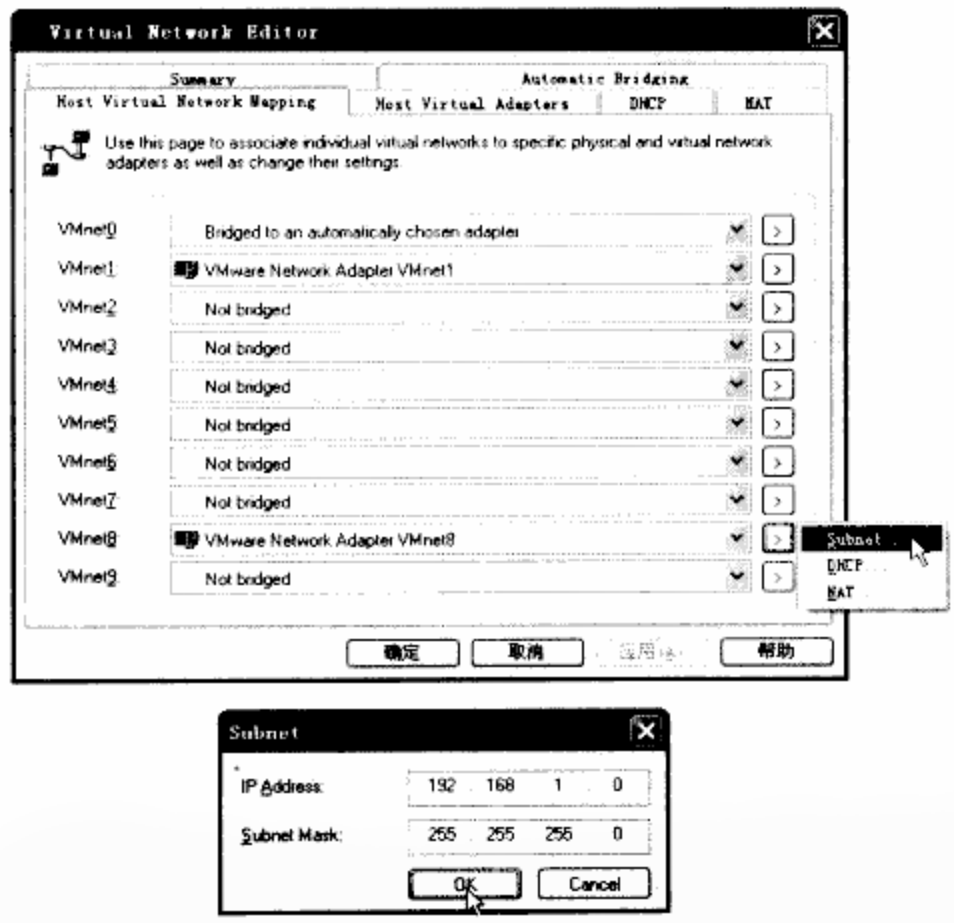


图 2.25 修改 NAT 网络的 IP 地址范围

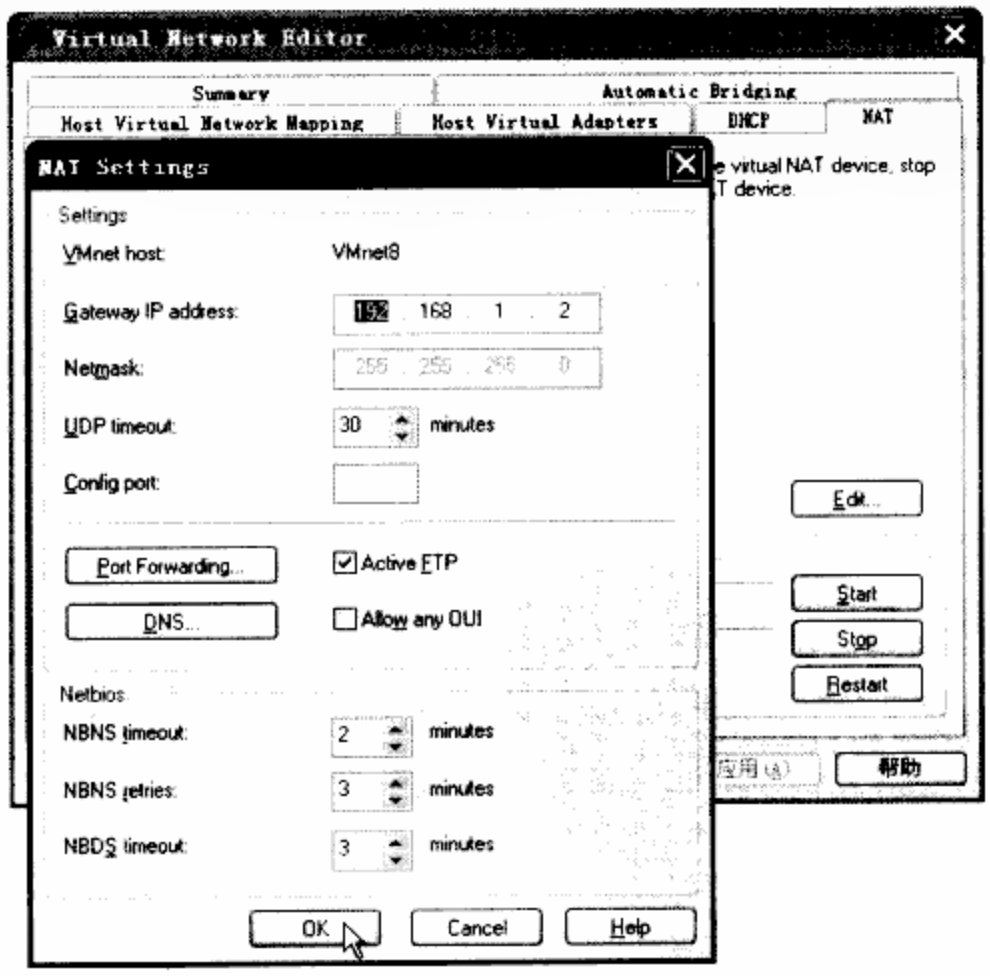


图 2.26 设置 NAT 网络的网关地址

2. 安装、配置、启动 FTP、SSH 或 NFS 服务

（1）准备使用软件维护工具 apt-get。

Ubuntu 7.10 中没有安装 FTP、SSH、NFS 服务器软件，它提供了一个很方便的安装、升

级、维护软件的工具 apt-get。apt-get 从光盘、网络上下载程序并安装。

第一次使用 apt-get 来安装程序之前，要先完成以下两件事。

① 修改/etc/apt/sources.list，将其中注释掉的网址打开。

在安装 Ubuntu 7.10 的时候，如果网络无法使用，它会自动将/etc/apt/sources.list 中各项注释去掉。比如需要将以下两行开头的“#”号去掉：

```
#deb http://cn.archive.ubuntu.com/ubuntu/ gutsy main restricted
#deb-src http://cn.archive.ubuntu.com/ubuntu/ gutsy main restricted
……
```

注意 由于/etc/apt/sources.list 属于 root 用户，而 Ubuntu 7.10 中屏蔽了 root 用户的使用，要修改它，需要使用 sudo 命令。比如可以使用“sudo vi /etc/apt/sources.list”来修改它，或者使用“sudo gedit &”命令启动图形化的文本编辑器，再打开、编辑它。

② 更新可用的程序列表。

执行如下命令即可，它只是更新内部的数据库以确定哪些程序已经安装、哪些没有安装、哪些有新版本。apt-get 程序将使用这个数据库来确定怎样安装用户指定的程序，并找到和安装它所依赖的其他程序。示例程序如下。

```
$ sudo apt-get update
```

（2）安装、配置、启动服务。

首先说明，Ubuntu 7.10 中隐藏了 root 用户，也就是说不能使用 root 用户登录，这可以避免不小心使用 root 权限而导致系统崩溃。当需要使用 root 权限时，使用“sudo”命令，比如要修改/etc/exports 文件时，修改如下所示：

```
# sudo vi /etc/exports
```

现在可以使用 apt-get 来安装软件了，以下的安装、配置、启动方法在 Ubuntu 7.10 自带的帮助文档中都有说明。

① 安装、配置、启动 FTP 服务。

执行以下命令安装，安装后即会自动运行：

```
$ sudo apt-get install vsftpd
```

修改 vsftpd 的配置文件/etc/vsftpd.conf，将下面几行前面的“#”去掉。

```
#local_enable=YES
#write_enable=YES
```

上面第一行表示是否允许本地用户登录，第二行表示是否允许上传文件。

修改完毕之后，执行以下命令重启 FTP 服务：

```
$ sudo /etc/init.d/vsftpd restart
```

② 安装、配置、启动 ssh 服务。

执行以下命令安装 ssh 服务，安装后即会自动运行：

```
$ sudo apt-get install openssh-server
```

它的配置文件为/etc/ssh/sshd_config，使用默认配置即可。

③ 安装、配置、启动 NFS 服务。

执行以下命令安装 nfs 服务，安装后即会自动运行：

```
$ sudo apt-get install nfs-kernel-server portmap
```

它的配置文件为/etc/exports，在里面增加以下内容，以后将通过网络文件系统访问/work/nfs_root 目录。

```
/work/nfs_root  *(rw,sync,no_root_squash)
```

修改完华之后，执行以下命令重启 NFS 服务：

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

以上 3 个服务使用 apt-get 安装后已经启动，并且以后每次开机都会自动启动。如果要取消某个服务，可以在 Linux 的启动栏菜单执行“System”→“系统管理”→“网络”命令，在对话框中取消。

2.2.3 在主机 Linux 操作系统中安装基本的开发环境

使用光盘安装的Ubuntu 7.10是一个比较精简的Linux发行版，它缺乏一些开发用的工具、文件，比如标准 C 库的头文件、g++编译器等。

先按照图 2.17 所示挂接上 Ubuntu 7.10 的安装光盘，然后使用以下命令安装基本的开发环境：

```
$ sudo apt-get install build-essential
```

还要安装工具 bison、flex，它们分别是语法、词法分析器：

```
$ sudo apt-get install bison flex
```

安装 C 函数库的 man 手册，以后就可以通过类似“man read”的命令查看函数的用法了：

```
$ sudo apt-get install manpages-dev
```

2.2.4 光盘的内容结构及安装

1. 光盘的内容结构

光盘内容分为 8 部分：硬件实验（hardware）、系统移植（system）、驱动和测试程序（drivers_and_test）、GUI、工具（tools）、scratchbox（这也是一个工具）、网络文件系统（nfs_root）、调试工具（debug），目录结构如图 2.27 所示。

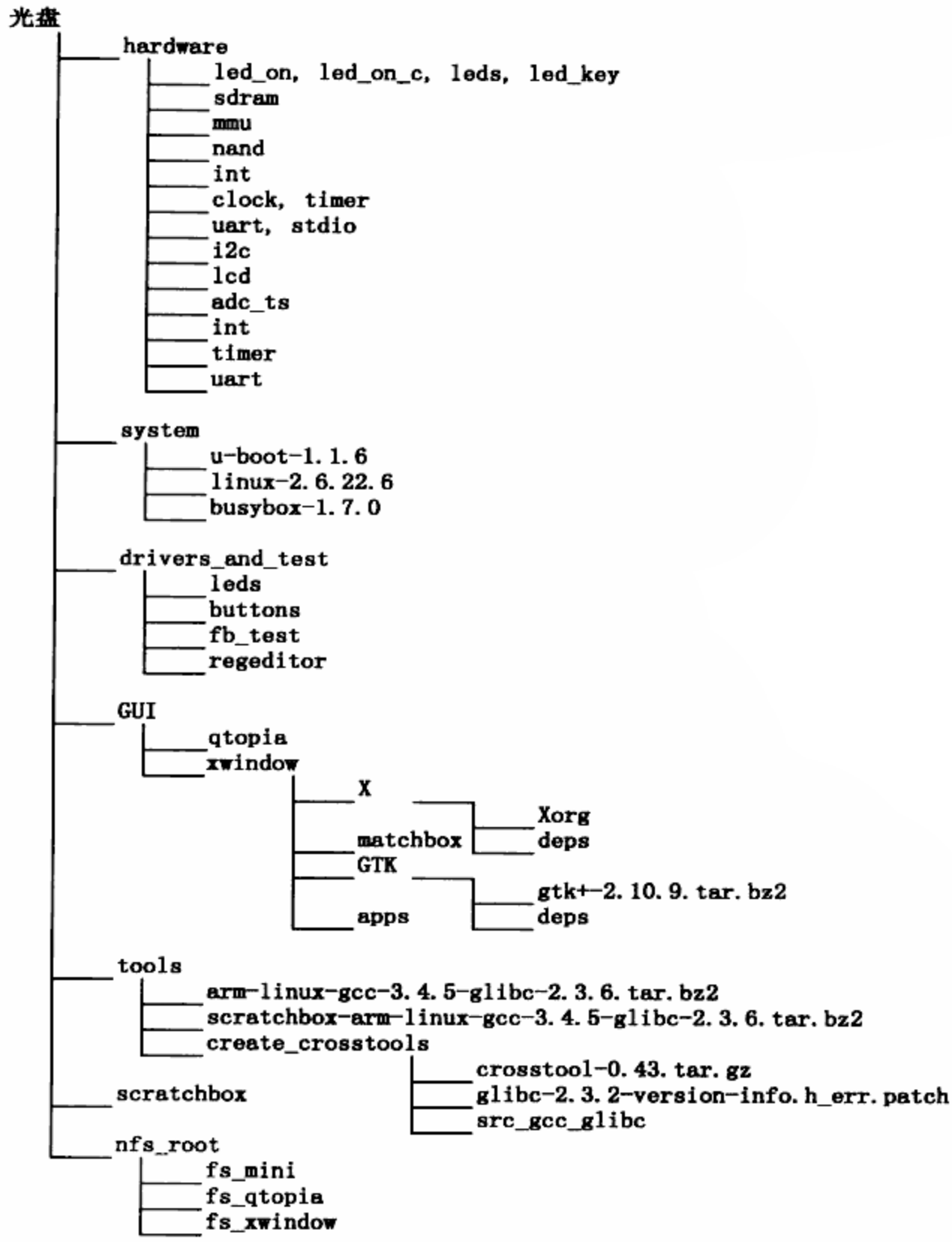


图 2.27 光盘内容结构

注意 光盘中几乎所有文件都制作成压缩包，比如光盘根目录中有一个文件 hardware.tar.bz2，它对应图 2.27 中的 hardware 目录，解压即得此目录。

hardware 目录下是各个硬件部件的实验代码，进入子目录后执行 make 命令即可生成可执行二进制程序（名字为子目录名加上后缀“.bin”），烧入开发板的 NAND Flash 中即可运行。

U-Boot、Linux 内核、根据 busybox 创建的文件系统这 3 部分构成了基本的、最小的嵌入式 Linux 系统，它们的代码在 system 目录下。本书中介绍的驱动也全部包含在 system/linux-2.6.22 内核代码中，drivers_and_test 主要是测试程序。

本书介绍两种 GUI 系统：qtopia、X window。GUI/xwindow 目录下有 4 个子目录。

- X 目录中主要是 Xorg 的代码，它提供 X server;
- matchbox 是一个基于 X 的、用于嵌入式系统的小型 GUI 环境，它包括一个窗口管理

器、一个面板、一个桌面、一个共享功能程序库和一些小的面板应用程序；

- GTK 目录下是 GTK+库，GTK+是一个用于创建图形用户界面的多平台工具，它包含有基本的控件和一些很复杂的控件，比如文件选择控件和颜色选择控件；
- apps 目录下是几个基于 X、GTK+的应用程序。

tools 目录下是运行于主机上的工具，主要是交叉编译工具。arm-linux-gcc-3.4.5-glibc-2.3.6.tar.bz2 和 scratchbox-arm-linux-gcc-3.4.5-glibc-2.3.6.tar.bz2 都是使用 create_crosstools 目录中的软件编译出来的交叉编译工具链，前一个是在主机上直接运行；后一个是当主机上启动 scratchbox 后，在 scratchbox 里面运行。这两个工具编译出来的代码是完全一样的。

scratchbox 是一个交叉编译工具包，它的目的是使嵌入式 Linux 开发变得像开发 PC Linux 软件一样容易。在移植 X window 时将用到它。

nfs_root 目录下是 4 个子目录：最小的根文件系统 fs_mini、使用 mdev 机制的根文件系统 fs_min_mdev、含有 qtopia 图形程序的根文件系统 fs_qtopia、含有 X window 图形程序的根文件系统 fs_xwindow。在开发阶段，单板上的内核启动后可以通过 NFS 挂接它们中的某一个，然后执行其中的程序；开发完成后，将所用的整个子目录制作为映象文件，烧入开发板。

2. 安装光盘

安装光盘前先使用以下的命令改变/work 目录的拥有者及所属组名：

```
$ sudo chown book:book /work -R
```

然后将光盘中所有内容复制到 Linux 的/work 目录中，有以下 3 种方法：

① 在 Windows 下，可以通过 cuteFTP 等工具登录 Linux，然后上传文件；

② 如果是通过 VMware 运行 Linux，则在 VMware 中挂接光盘后（如图 2.17 所示），Linux 中会自动弹出光盘的目录，将它的内容复制到/work 目录下；

③ 如果是直接运行 Linux，在光驱中放入光盘后，Linux 中也会自动弹出光盘的目录，将它的内容复制到/work 目录下。

2.2.5 安装交叉编译工具链

首先约定：在主机上执行的命令提示符为“$”；在主机中启动 scratchbox，然后在 scratchbox 里执行的命令，提示符为“>”；在单板上执行的命令，提示符为“#”。比如同是执行“ls”命令，用下面 3 行表示：

```
$ ls
> ls
# ls
```

1. 使用制作好的工具链

刚开始学习时，建议使用已经制作好工具链，使用以下命令解压得到 gcc-3.4.5-glibc-2.3.6 目录。

```
$ cd /work/tools
$ tar xjf arm-linux-gcc-3.4.5-glibc-2.3.6.tar.bz2
```

然后在环境变量PATH中增加路径，如下所示：

```
$ export PATH=$PATH:/work/tools/gcc-3.4.5-glibc-2.3.6/bin
```

这使得可以直接运行这个目录下的程序，而不需要指定目录位置。

为了不要每次使用时都手工设置PATH，可以在/etc/environment中修改PATH的值如下所示：

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games: /work/tools/gcc-3.4.5-glibc-2.3.6/bin "
```

安装ncurses，ncurses是一个能提供功能键定义（快捷键）、屏幕绘制以及基于文本终端的图形互动功能的动态库。如果没有它，在执行“make menuconfig”命令配置程序时会出错。使用以下命令安装：

```
$ cd /work/tools/
$ tar xzf ncurses.tar.gz
$ cd ncurses-5.6
$ ./configure --with-shared --prefix=/usr
$ make
$ make install
```

2. 自己制作工具链

用户也可以自己编译工具链。如果要基于gcc和glibc来制作工具链，可以使用crosstool来进行编译；如果要基于gcc和uClibc来制作工具链，可以使用buildroot来进行编译。如果不借助于这些工具，编译过程是非常繁锁的。uClibc比glibc小，在已有的接口上是兼容的，更适用于嵌入式系统。但是uClibc并没有包括glibc中的所有接口实现，因此有些应用可能在uClibc中不能编译。基于这个原因，本书使用glibc，当对系统很熟悉后，或是在开发资源很受限制的产品时，可以使用uClibc。

下面将使用/work/tools/create_crosstools目录下的crosstool-0.43.tar.gz工具来编译工具链，它运行时，会自动从网上下载源码，然后编译。也可以先自己下载源码，再运行crosstool。本书已经将源码放在src_gcc_glibc目录下。

crosstool官方网站为http://kegel.com/crosstool/，可以参考其中的crosstool-how to.html选择、配置、编译工具链。

下面分步讲解。

（1）修改crosstool脚本。

执行以下命令解压缩：

```
$ tar xzf crosstool-0.43.tar.gz
```

glibc-2.3.6-version-info.h_err.patch是一个补丁文件，它修改glibc-2.3.6/csu/Makefile，里面的一个小错误，导致自动生成的version-info.h文件编译出错。将它复制到crosstool的补丁目录下：

```
$ cp glibc-2.3.6-version-info.h_err.patch crosstool-0.43/patches/glibc-2.3.6/
```

注意 另一个补丁文件 ld-2.15-scratchbox_NATIVE.patch 在第 26 章制作运行在 scratchbox 上的交叉编译工具链时才用到。

后面将执行 crosstool-0.43 目录下的 demo-arm-softfloat.sh 脚本来进行编译，摘取它的部分内容如下：

```
07 TARBALLS_DIR=$HOME/downloads
08 RESULT_TOP=/opt/crosstool
09 export TARBALLS_DIR RESULT_TOP
10 GCC_LANGUAGES="c,c++"
……
26 #eval 'cat arm-softfloat.dat gcc-3.3.6-glibc-2.3.2-tls.dat' sh all.sh --notest
27 #eval 'cat arm-softfloat.dat gcc-3.4.5-glibc-2.2.5.dat' sh all.sh --notest
28 #eval 'cat arm-softfloat.dat gcc-3.4.5-glibc-2.3.5.dat' sh all.sh --notest
29 eval 'cat arm-softfloat.dat gcc-3.4.5-glibc-2.3.6.dat' sh all.sh --notest
```

第 7 行的 TARBALLS_DIR 表示源码存放的位置。

第 8 行的 RESULT_TOP 表示编译结果存放的位置。

第 10 行的 GCC_LANGUAGES 表示制作出来的工具链支持 C、C++语言，如果要支持其他语言，可以在里面增加。比如下面一行表示支持 Java：

```
GCC_LANGUAGES="c,c++,java"
```

从第 26～29 行可知，可以选择多种 gcc、glibc 版本，本书使用默认版本：gcc-3.4.5 和 glibc-2.3.6。执行 demo-arm-softfloat.sh 脚本后，它将根据 arm-softfloat.dat、gcc-3.4.5-glibc-2.3.6.dat 这两个文件中定义的环境变量调用 all.sh 脚本进行编译。gcc-3.4.5-glibc-2.3.6.dat 文件指明了要下载或使用的文件。

需要修改 demo-arm-softfloat.sh、arm-softfloat.dat、all.sh 这 3 个文件。

① 修改 demo-arm-softfloat.sh，修改后的内容如下：

```
07 TARBALLS_DIR=/work/tools/create_crosstools/src_gcc_glibc
08 RESULT_TOP=/work/tools
```

② 修改 arm-softfloat.dat，修改如下：

```
02 TARGET=arm-softfloat-linux-gnu
改为:
02 TARGET=arm-linux
```

它表示编译出来的工具样式为 arm-linux-gcc、arm-linux-ld 等，这是常用的名字。

③ 修改 all.sh。

如果现在就执行 demo-arm-softfloat.sh，最终结果将存放在/work/tools/gcc-3.4.5-glibc-2.3.6/arm-linux 目录下。为简洁起见，修改 all.sh，将结果存放在/work/tools/gcc-3.4.5-glibc-2.3.6 目录下。

```
70 PREFIX=${PREFIX-$RESULT_TOP/$TOOLCOMBO/$TARGET}
改为:
70 PREFIX=${PREFIX-$RESULT_TOP/$TOOLCOMBO}
```

（2）编译、安装工具链。

执行以下命令：

```
$ cd crosstool-0.43/
$ ./demo-arm-softfloat.sh
```

编译2、3个小时后，将在/work/tools/目录下生成gcc-3.4.5-glibc-2.3.6子目录，交叉编译器、库、头文件都包含在里面。设置PATH环境变量即可使用。使用下面命令测评一下：

```
$ arm-linux-gcc -v
```

现在，基本的开发环境已经建立，在后续开发过程中，要使用到其他工具时，再进行安装。

2.2.6 书中写作风格的约定

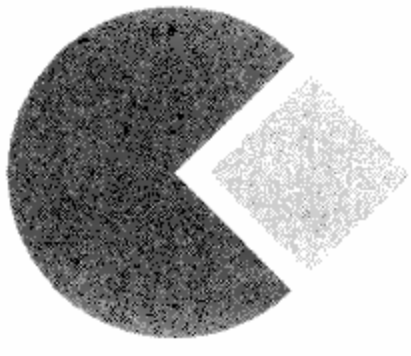
2.2.5小节开头部分约定：使用“$”、“>”、“#”这3个提示符分别表示在主机、scratchbox、或者在开发板上执行命令。

另外一个约定是关于“函数调用”的。阅读Linux源代码最大的困难是各个函数之间的调用关系很复杂，内核中常常有这种调用方式：函数A调用函数B，不是在A的代码里直接调用B，而是通过某个函数指针（它在其他地方被赋值为B）间接调用。

编者在阅读代码时，常使用以下的方式标记函数调用关系，在书中也使用同样的方式。假设有A、B、C、D共4个函数，它们分别处于a.c、b.c、c.c、d.c这4个文件中，A调用B和C，B调用D，则使用以下的缩进方式表示：

```
A(a.c) ->
    B(b.c) //可以加一些注释->
        D(d.c)
    C(c.c)
```

B比A缩进一格，表示B在A中被调用；相同的缩进表示同处于一个函数中，比如B、C都是在A中被调用；“//”号后可以加一些注释；“->”起辅助作用，表示“调用”。



第 3 章　嵌入式编程基础知识

本章目标

- 了解交叉编译工具链的各种选项
- 掌握连接脚本的编译方法
- 了解 Makefile 文件中常用的函数
- 了解几个常用的 ARM 汇编指令
- 了解汇编程序调用 C 函数所遵循的 ATPCS 规则

3.1　交叉编译工具选项说明

源文件需要经过编译才能生成可执行文件。在 Windows 下进行开发时，只需要单击几个按钮即可编译，集成开发环境（比如 Visual studio）已经将各种编译工具的使用封装好了。Linux 下也有很优秀的集成开发工具，但是更多的时候是直接使用编译工具；即使使用集成开发工具，也需要掌握一些编译选项。

PC 上的编译工具链为 gcc、ld、objcopy、objdump 等，它们编译出来的程序在 x86 平台上运行。要编译出能在 ARM 平台上运行的程序，必须使用交叉编译工具 arm-linux-gcc、arm-linux-ld 等，下面分别介绍。

3.1.1　arm-linux-gcc 选项

一个 C/C++文件要经过预处理（preprocessing）、编译（compilation）、汇编（assembly）和连接（linking)等 4 步才能变成可执行文件，如图 3.1 所示。在日常交流中通常使用“编译”统称这 4 个步骤，如果不是特指这 4 个步骤中的某一个，本书也依惯例使用“编译”这个统称。

（1）预处理。

C/C++源文件中，以“#”开头的命令被称为预处理命令，如包含命令“#include”、宏定义命令“#define”、条件编译命令“#if”、“#ifdef”等。预处理就是将要包含（include）的文件插入原文件中、将宏定义展开、根据条件编译命令选择要使用的代码，最后将这些代码输出到一个“.i”文件中等待进一步处理。预处理将用到 arm-linux-cpp 工具。

（2）编译。

编译就是把C/C++代码（比如上述的“.i”文件）“翻译”成汇编代码，所用到的工具为cc1（它的名字就是cc1，不是arm-linux-cc1）。

（3）汇编。

汇编就是将第二步输出的汇编代码翻译成符合一定格式的机器代码，在Linux系统上一般表现为ELF目标文件（OBJ文件），用到的工具为arm-linux-as。“反汇编”是指将机器代码转换为汇编代码，这在调试程序时常常用到。

（4）连接。

连接就是将上步生成的OBJ文件和系统库的OBJ文件、库文件连接起来，最终生成可以在特定平台运行的可执行文件，用到的工具为arm-linux-ld。

编译器利用这4个步骤中的一个或多个来处理输入文件，源文件的后缀名表示源文件所用的语言，后缀名控制着编译器的默认动作，如表3.1所示。

表3.1 文件后缀名与编译器的默认动作对应表

后缀名	语言种类	后期操作
.c	C源程序	预处理、编译、汇编
.C	C++源程序	预处理、编译、汇编
.cc	C++源程序	预处理、编译、汇编
.cxx	C++源程序	预处理、编译、汇编
.m	Objective-C源程序	预处理、编译、汇编
.i	预处理后的C文件	编译、汇编
.ii	预处理后的C++文件	编译、汇编
.s	汇编语言源程序	汇编
.S	汇编语言源程序	预处理、汇编
.h	预处理器文件	通常不出现在命令行上

其他后缀名的文件被传递给连接器（linker），通常包括以下两种。

.o：目标文件（Object file，OBJ文件）。

.a：归档库文件（Archive file）。

在编译过程中，除非使用了“-c”，“-S”或“-E”选项（或者编译错误阻止了完整的过程），否则最后的步骤总是连接。在连接阶段中，所有对应于源程序的.o文件、“-l”选项指定的库文件、无法识别的文件名（包括指定的“.o”目标文件和“.a”库文件）按命令行中的顺序传递给连接器。

以一个简单的“Hello,world!”C程序为例，代码在/work/hardware/hello目录下。它的代码如下，功能为打印“Hello World!”字符串。

```
01 /* File: hello.c */
02 #include <stdio.h>
03 int main(int argc, char *argv[])
04 {
```

```
05      printf("Hello World!\n");
06      return 0;
07 }
```

使用 arm-linux-gcc，只需要一个命令就可以生成可执行文件 hello，它包含了上述 4 个步骤：

```
$ arm-linux-gcc -o hello hello.c
```

加上“-v”选项，即使用“arm-linux-gcc -v -o hello hello.c”命令可以观看编译的细节，下面摘取关键部分：

```
cc1 hello.c -o /tmp/cctETob7.s
as -o /tmp/ccvv2KbL.o /tmp/cctETob7.s
collect2 -o hello crt1.o crti.o crtbegin.o /tmp/ccvv2KbL.o crtend.o crtn.o
```

以上 3 个命令分别对应于编译步骤中的预处理+编译、汇编和连接，ld 被 collect2 调用来连接程序。预处理和编译被放在了一个命令（cc1）中进行，可以把它再次拆分为以下两步：

```
cpp -o hello.i hello.c
cc1 hello.i -o /tmp/cctETob7.s
```

可以通过各种选项来控制 arm-linux-gcc 的动作，下面介绍一些常用的选项。

1. 总体选项（Overall Option）

（1）-c。

预处理、编译和汇编源文件，但是不作连接，编译器根据源文件生成 OBJ 文件。默认情况下，GCC 通过用“.o”替换源文件名的后缀“.c”，“.i”，“.s”等，产生 OBJ 文件名。可以使用“-o”选项选择其他名字。GCC 忽略“-c”选项后面任何无法识别的输入文件。

（2）-S。

编译后即停止，不进行汇编。对于每个输入的非汇编语言文件，输出结果是汇编语言文件。默认情况下，GCC 通过用“.s”替换源文件名后缀“.c”，“.i”等，产生汇编文件名。可以使用“-o”选项选择其他名字。GCC 忽略任何不需要汇编的输入文件。

（3）-E。

预处理后即停止，不进行编译。预处理后的代码送往标准输出。GCC 忽略任何不需要预处理的输入文件。

（4）-o file。

指定输出文件为 file。无论是预处理、编译、汇编还是连接，这个选项都可以使用。如果没有使用“-o”选项，默认的输出结果是：可执行文件为“a.out”；修改输入文件的名称是“source.suffix”，则它的 OBJ 文件是“source.o”，汇编文件是“source.s”，而预处理后的 C 源代码送往标准输出。

（5）-v。

显示制作 GCC 工具自身时的配置命令；同时显示编译器驱动程序、预处理器、编译器的版本号。

以一个程序为例，它包含 3 个文件，代码在/work/hardware/options 目录下。下面列出源码：

```
File: main.c
01 #include <stdio.h>
02 #include "sub.h"
03
04 int main(int argc, char *argv[])
05 {
06         int i;
07         printf("Main fun!\n");
08         sub_fun();
09         return 0;
10 }
11

File: sub.h
01 void sub_fun(void);
02

File: sub.c
01 void sub_fun(void)
02 {
03         printf("Sub fun!\n");
04 }
05
```

arm-linux-gcc、arm-linux-ld 等工具与 gcc、ld 等工具的使用方法相似，很多选项是一样的。本节使用 gcc、ld 等工具进行编译、连接，这样可以在 PC 上直接看到运行结果。使用上面介绍的选项进行编译，命令如下：

```
$ gcc  -c  -o  main.o  main.c
$ gcc  -c  -o  sub.o  sub.c
$ gcc  -o  test  main.o  sub.o
```

其中，main.o、sub.o 是经过了预处理、编译、汇编后生成的 OBJ 文件，它们还没有被连接成可执行文件；最后一步将它们连接成可执行文件 test，可以直接运行以下命令：

```
$ ./test
Main fun!
Sub fun!
```

现在试试其他选项，以下命令生成的 main.s 是 main.c 的汇编语言文件：

```
$ gcc  -S  -o  main.s  main.c
```

以下命令对 main.c 进行预处理，并将得到的结果打印出来。里面扩展了所有包含的文件、

所有定义的宏。在编写程序时，有时候查找某个宏定义是非常繁琐的事，可以使用“-dM-E”选项来查看。命令如下：

```
$ gcc -E main.c
```

2. 警告选项（Warning Option）

“-Wall”选项基本打开了所有需要注意的警告信息，比如没有指定类型的声明、在声明之前就使用的函数、局部变量除了声明就没再使用等。

上面的 main.c 文件中，第 6 行定义的变量 i 没有被使用，但是使用“gcc –c –o main.o main.c”进行编译时并没有出现提示。

可以加上“-Wall”选项，例子如下：

```
$ gcc -Wall -c main.c
```

执行上述命令后，得到如下警告信息：

```
main.c: In function 'main":
main.c:6: warning: unused variable 'i"
```

这个警告虽然对程序没有坏的影响，但是有些警告需要加以关注，比如类型匹配的警告等。

3. 调试选项（Debugging Option）

-g：以操作系统的本地格式（stabs、COFF、XCOFF 或 DWARF）产生调试信息，GDB 能够使用这些调试信息。在大多数使用 stabs 格式的系统上，“-g”选项加入只有 GDB 才使用的额外调试信息。可以使用下面的选项来生成额外的信息：“-gstabs+”，“-gstabs”，“-gxcoff+”，“-gxcoff”，“-gdwarf+”和“-gdwarf”，具体用法请读者参考 GCC 手册。

4. 优化选项（Optimization Option）

（1）-O 或-O1。

优化：对于大函数，优化编译的过程将占用较长时间和相当大的内存。不使用“-O”选项的目的是减少编译的开销，使编译结果能够调试、语句是独立的。如果在两条语句之间用断点中止程序，可以对任何变量重新赋值，或者在函数体内把程序计数器指到其他语句，以及从源程序中精确地获取所期待的结果。

不使用“–O”或“–O1”选项时，只有声明了 register 的变量才分配使用寄存器。

使用了“-O”或“–O1”选项时，编译器会试图减少目标码的大小和执行时间。如果指定了“-O”或“-O1”选项，“-fthread-jumps”和“-fdefer-pop”选项将被打开。在有 delay slot 的机器上，“-fdelayed-branch”选项将被打开。在既没有帧指针（frame pointer）又支持调试的机器上，“-fomit-frame-pointer”选项将被打开。某些机器上还可能会打开其他选项。

（2）-O2。

多优化一些。除了涉及空间和速度交换的优化选项，执行几乎所有的优化工作。例如不进行循环展开（loop unrolling）和函数内嵌（inlining）。和“-O”选项相比，这个选项既增加了编译时间，也提高了生成代码的运行效果。

（3）-O3。

优化的更多。除了打开“-O2”所做的一切，它还打开了“-finline-functions”选项。

（4）-O0。

不优化。

如果指定了多个“-O”选项，不管带不带数字，生效的是最后一个选项。

在一般应用中，经常使用“-O2”选项，比如对于 options 程序：

```
$ gcc -O2 -c -o main.o main.c
$ gcc -O2 -c -o sub.o sub.c
$ gcc -o test main.o sub.o
```

5. 连接器选项（Linker Option）

下面的选项用于连接 OBJ 文件，输出可执行文件或库文件。

（1）object-file-name。

如果某些文件没有特别明确的后缀（a special recognized suffix），GCC 就认为它们是 OBJ 文件或库文件（根据文件内容，连接器能够区分 OBJ 文件和库文件）。如果 GCC 执行连接操作，这些 OBJ 文件将成为连接器的输入文件。

比如上面的“gcc -o test main.o sub.o”中，main.o、sub.o 就是输入的文件。

（2）-llibrary。

连接名为 library 的库文件。

连接器在标准搜索目录中寻找这个库文件，库文件的真正名字是“liblibrary.a”。搜索目录除了一些系统标准目录外，还包括用户以“-L”选项指定的路径。一般说来用这个方法找到的文件是库文件——即由 OBJ 文件组成的归档文件（archive file）。连接器处理归档文件的方法是：扫描归档文件，寻找某些成员，这些成员的符号目前已被引用，不过还没有被定义。但是，如果连接器找到普通的 OBJ 文件，而不是库文件，就把这个 OBJ 文件按平常方式连接进来。指定“-l”选项和指定文件名的唯一区别是，“-l”选项用“lib”和“.a”把 library 包裹起来，而且搜索一些目录。

即使不明显地使用“-llibrary”选项，一些默认的库也被连接进去，可以使用“-v”选项看到这点：

```
$ gcc -v -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386
-dynamic-linker /lib/ld-linux.so.2
-o test
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crt1.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crti.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtbegin.o
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
```

```
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
main.o
sub.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/crtend.o
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../../crtn.o
```

可以看见，除了 main.o、sub.o 两个文件外，还连接了启动文件 crt1.o、crti.o、crtend.o 、crtn.o，还有一些库文件（-lgcc、-lgcc_eh、-lc–lgcc、-lgcc_eh）。

（3）-nostartfiles。

不连接系统标准启动文件，而标准库文件仍然正常使用：

```
$ gcc -v -nostartfiles -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386
-dynamic-linker
/lib/ld-linux.so.2
-o test
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
main.o
sub.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

可以看见启动文件 crt1.o、crti.o、crtend.o 、crtn.o 没有被连接进去。需要说明的是，对于一般应用程序，这些启动文件是必需的，这里仅是作为例子（这样编译出来的 test 文件无法执行）。在编译 bootloader、内核时，将用到这个选项。

（4）-nostdlib。

不连接系统标准启动文件和标准库文件，只把指定的文件传递给连接器。这个选项常用于编译内核、bootloader 等程序，它们不需要启动文件、标准库文件。

仍以 options 程序作为例子：

```
$ gcc -v -nostdlib -o test main.o sub.o
```

输出的信息如下：

```
/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/collect2 --eh-frame-hdr -m elf_i386
-dynamic-linker /lib/ld-linux.so.2
-o test
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2
-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/../../..
```

```
main.o
sub.o
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048074
main.o(.text+0x19): In function `main" :
: undefined reference to `printf"
sub.o(.text+0xf): In function `sub_fun" :
: undefined reference to `printf"
collect2: ld returned 1 exit status
```

出现了一大堆错误，因为 printf 等函数是在库文件中实现的。在编译 bootloader、内核时，用到这个选项，它们用到的很多函数是自包含的。

（5）-static。

在支持动态连接（dynamic linking）的系统上阻止连接共享库。

仍以 options 程序为例，使用和不使用“-static”选项编译出来的可执行程序大小相差巨大：

```
$ gcc -c -o main.c
$ gcc -c -o sub.c
$ gcc -o test main.o sub.o
$ gcc -o test_static main.o sub.o -static
$ ls -l test test_static
-rwxr-xr-x 1 book book   6591 Jan 16 23:51 test
-rwxr-xr-x 1 book book 546479 Jan 16 23:51 test_static
```

其中 test 文件为 6591 字节，test_static 文件为 546479 字节。当不使用“-static”编译文件时，程序执行前要连接共享库文件，所以还需要将共享库文件放入文件系统中。

（6）-shared。

生成一个共享 OBJ 文件，它可以和其他 OBJ 文件连接产生可执行文件。只有部分系统支持该选项。

当不想以源代码发布程序时，可以使用“-shared”选项生成库文件，比如对于 options 程序，可以如下制作库文件：

```
$ gcc -c -o sub.o sub.c
$ gcc -shared -o sub.a sub.o
```

以后要使用 sub.c 中的函数 sub_fun 时，在连接程序时，将 sub.a 加入即可，比如：

```
$ gcc -o test main.o ./sub.a
```

可以将多个文件制作为一个库文件，比如：

```
$ gcc -shared  -o sub.a  sub.o  sub2.o  sub3.o
```

（7）-Xlinker option。

把选项 option 传递给连接器。可以用来传递系统特定的连接选项，GCC 无法识别这些选项。如果需要传递携带参数的选项，必须使用两次“-Xlinker”，一次传递选项，另一次传递

其参数。例如，如果传递“-assert definitions”，要写成“-Xlinker -assert -Xlinker definitions”，而不能写成-“Xlinker”-assert definitions“”，因为这样会把整个字符串当做一个参数传递，显然这不是连接器期待的。

（8）-Wl，option。

把选项 option 传递给连接器。如果 option 中含有逗号，就在逗号处分割成多个选项。连接器通常是通过 gcc、arm-linux-gcc 等命令间接启动的，要向它传入参数时，参数前面加上“-Wl,”。

（9）-u symbol。

使连接器认为取消了 symbol 的符号定义，从而连接库模块以取得定义。可以使用多个“-u”选项，各自跟上不同的符号，使得连接器调入附加的库模块。

6. 目录选项（Directory Option）

下列选项指定搜索路径，用于查找头文件、库文件或编译器的某些成员。

（1）-Idir。

在头文件的搜索路径列表中添加 dir 目录。

头文件的搜索方法为：如果以“#include <>”包含文件，则只在标准库目录开始搜索（包括使用-Idir 选项定义的目录）；如果以“#include”包含文件，则先从用户的工作目录开始搜索，再搜索标准库目录。

（2）-I-。

任何在“-I-”前面用“-I”选项指定的搜索路径只适用于“#include"file"”这种情况；它们不能用来搜索“#include <file>”包含的头文件。如果用“-I”选项指定的搜索路径位于“-I-”选项后面，就可以在这些路径中搜索所有的“#include”指令（一般说来“-I-”选项就是这么用的）。“-I”选项能够阻止当前目录（存放当前输入文件的地方）成为搜索“#include "file"”的第一选择。

“-I-”不影响使用系统标准目录，因此，“-I-”和“-nostdinc”是不同的选项。

（3）-Ldir。

在“-I”选项的搜索路径列表中添加 dir 目录。

仍使用 options 程序进行说明，先制作库文件 libsub.a：

```
$ gcc -c -o sub.o sub.c
$ gcc -shared -o libsub.a sub.o
```

编译 main.c：

```
$ gcc  -c -o  main.o  main.c
```

连接程序，下面的指令将出错，提示找不到库文件：

```
$ gcc  -o  test  main.o  -lsub
/usr/bin/ld: cannot find -lsub
collect2: ld returned 1 exit status
```

可以使用“-Ldir”选项将当前目录加入搜索路径，如下则连接成功：

```
$ gcc -L. -o test main.o -lsub
```

（4）-Bprefix。

这个选项指出在何处寻找可执行文件、库文件以及编译器自己的数据文件。编译器驱动程序需要使用某些工具，比如：cpp、cc1 （或C++的cc1plus）、as和ld。它把prefix当作欲执行的工具的前缀，这个前缀可以用来指定目录，也可以用来修改工具名字。

对于要运行的工具，编译器驱动程序首先试着加上“-B”前缀（如果存在），如果没有找到文件或没有指定“-B”选项，编译器接着会试验两个标准前缀/usr/lib/gcc/和/usr/local/lib/gcc-lib/。如果仍然没能够找到所需文件，编译器就在“PATH”环境变量指定的路径中寻找没加任何前缀的文件名。如果有需要，运行时（run-time）支持文件libgcc.a也在“-B”前缀的搜索范围之内。如果这里没有找到，就在上面提到的两个标准前缀中寻找。如果上述方法没有找到这个文件，就不连接它了。多数情况的多数机器上，libgcc.a并非必不可少。

可以通过环境变量GCC_EXEC_PREFIX获得近似的效果；如果定义了这个变量，其值就和上面说的一样被用作前缀。如果同时指定了“-B”选项和GCC_EXEC_PREFIX变量，编译器首先使用“-B”选项，然后才尝试环境变量值。

3.1.2 arm-linux-ld 选项

arm-linux-ld用于将多个目标文件、库文件连接成可执行文件，它的大多数选项已经在3.1.1小节中介绍了。

本小节介绍“-T”选项，可以直接使用它来指定代码段、数据段、bss段的起始地址，也可以用来指定一个连接脚本，在连接脚本中进行更复杂的地址设置。

“-T”选项只用于连接Bootloader、内核等“没有底层软件支持”的软件；连接运行于操作系统之上的应用程序时，无需指定“-T”选项，它们使用默认的方式进行连接。

1. 直接指定代码段、数据段、bss段的起始地址

格式如下：

```
-Ttext startaddr
-Tdata startaddr
-Tbss  startaddr
```

其中的“startaddr”分别表示代码段、数据段和bss段的起始地址，它是一个十六进制数。

在/work/hardware/led_on/Makefile中，有如下语句：

```
arm-linux-ld -Ttext 0x0000000 -g led_on.o -o led_on_elf
```

它表示代码段的运行地址为0x0000000，由于没有定义数据段、bss段的起始地址，它们被依次放在代码段的后面。

以一个例子来说明“-Ttext”选项的作用。在/work/hardware/link目录下有个link.s文件，内容如下：

```
01 .text
02 .global _start
03 _start:
```

```
04          b   step1  .
05 step1:
06          ldr pc, =step2
07 step2:
08   b step2
```

使用下面的命令编译、连接、反汇编：

```
arm-linux-gcc  -c -o link.o link.s
arm-linux-ld -Ttext 0x00000000  link.o -o link_elf_0x00000000
arm-linux-ld -Ttext 0x30000000  link.o -o link_elf_0x30000000
arm-linux-objdump -D link_elf_0x00000000 > link_0x00000000.dis
arm-linux-objdump -D link_elf_0x30000000 > link_0x30000000.dis
```

link.s 中用到两种跳转方法：b 跳转指令、直接向 pc 寄存器赋值。先列出不同“-Ttext”选项下生成的反汇编文件，再详细分析由于不同运行地址带来的差异及影响。这两个反汇编文件如下：

```
link_0x00000000.dis:                        link_0x30000000.dis
0:  eaffffff  b   0x4                       0:  eaffffff  b   0x4
4:  e59ff000  ldr pc, [pc, #0] ; 0xc        4:  e59ff000  ldr pc, [pc, #0] ; 0xc
8:  eaffffe  b   0x8                        8:  eaffffe  b   0x8
c:  00000008  andeq r0, r0, r8              c:  30000008  tsteq r0, #8  ; 0x8
```

先看 link.s 中第一条指令“b step1”。b 跳转指令是个相对跳转指令，其机器码格式如下：

Cond	1	0	1	L	Offset

① [31:28]位是条件码。

② [27:24]位为“1010”时，表示 b 跳转指令；为“1011”时，表示 bl 跳转指令。

③ [23:0]表示偏移地址。

使用 b 或 bl 跳转时，下一条指令的地址是这样计算的：将指令中 24 位带符号的补码扩展为 32 位（扩展其符号位）；将此 32 位数左移两位；将得到的值加到 pc 寄存器中，即得到跳转的目标地址。

第一条指令“b step1”的机器码为 eaffffff。

① 24 位带符号的补码为 0xffffff，将它扩展为 32 位得到 0xffffffff。

② 将此 32 位数左移两位得到 0xfffffffc，其值就是-4。

③ pc 的值是当前指令的下两条指令的地址，加上步骤②得到的-4，这恰好是第二条指令 step1 的地址。

请读者不要被反汇编代码中的“b 0x4”迷惑，它不是指跳到绝对地址 0x4 处执行，绝对地址需要按照上述 3 个步骤计算。可以发现，b 跳转指令依赖于当前 pc 寄存器的值，这个特性使得使用 b 指令的程序不依赖于代码存储的位置——即不管这条代码放在什么位置，b 指令都可以跳到正确的位置。这类指令被称为位置无关码，使用不同的“-Ttext”选项，生成的

代码仍是一样的。

再看第二条指令“ldr pc, =step2”。从汇编码“ldr pc, [pc, #0]”可以看出，这条指令从内存的某个位置读出数据，并赋给 pc 寄存器。这个位置的地址是当前 pc 寄存器的值加上偏移值 0，其中存放的值依赖于连接命令的“-Ttext”选项。执行这条指令后，对于 link_0x00000000.dis，pc=0x00000008；对于 link_0x30000000.dis， pc=0x30000008。执行第三条指令“b step2”后，程序的运行地址就不同了：分别是 0x00000008、0x30000008。

Bootloader、内核等程序刚开始执行时，它们所处的地址通常不等于运行地址。在程序的开头，先使用 b、bl、mov 等“位置无关”的指令将代码从 Flash 等设备中复制到内存的“运行地址”处，然后再跳到“运行地址”去执行。

2. 使用连接脚本设置地址

以/work/source/hardware/timer 目录下的程序为例，它的 Makefile 中有以下代码：

```
arm-linux-ld -Ttimer.lds -o timer_elf $^
```

其中的“$^”表示“head.o init.o interrupt.o main.o”（为何如此暂时不用管），所以这句代码就变为：

```
arm-linux-ld -Ttimer.lds -o timer_elf head.o init.o interrupt.o main.o
```

它使用连接脚本 timer.lds 来设置可执行文件 timer_elf 的地址信息，timer_elf 文件内容如下：

```
01 SECTIONS {
02      . = 0x30000000;
03      .text          :   { *(.text) }
04      .rodata ALIGN(4) : {*(.rodata)}
05      .data ALIGN(4) : { *(.data) }
06      .bss ALIGN(4)  : { *(.bss)  *(COMMON) }
07 }
```

解析 timer_elf 文件之前，先讲解连接脚本的格式。连接脚本的基本命令是 SECTIONS 命令，它描述了输出文件的“映射图”：输出文件中各段、各文件怎么放置。一个 SECTIONS 命令内部包含一个或多个段，段（Section）是连接脚本的基本单元，它表示输入文件中的某部分怎么放置。

完整的连接脚本格式如下，它的核心部分是段（Section）：

```
SECTIONS {
...
secname start ALIGN(align) (NOLOAD) : AT(ldadr)
  { contents } >region :phdr =fill
...
}
```

secname 和 contents 是必需的，前者用来命名这个段，后者用来确定代码中的什么部分

放在这个段中。

start 是这个段重定位地址，也称为运行地址。如果代码中有位置无关的指令，程序在运行时，这个段必须放在这个地址上。

ALIGN（align）：虽然 start 指定了运行地址，但是仍可以使用 BLOCK（align）来指定对齐的要求——这个对齐的地址才是真正的运行地址。

（NOLOAD）：用来告诉加载器，在运行时不用加载这个段。显然，这个选项只有在有操作系统的情况下才有意义。

AT（ldadr）：指定这个段在编译出来的映象文件中的地址——加载地址（load address）。如果不使用这个选项，则加载地址等于运行地址。通过这个选项，可以控制各段分别保存输出文件中不同的位置，便于把文件保存到到单板上：A 段放在 A 处，B 段放在 B 处，运行前再把 A、B 段分别读出来组装成一个完整的执行程序。

后面的 3 个选项“>region :phdr =fill”在本书中没有用到，这里不再介绍。

现在，可以明白前面的连接脚本 timer.lds 的含义了。

① 第 2 行表示设置“当前运行地址”为 0x30000000。

② 第 3 行定义了一个名为“.text”的段，它的内容为“*（.text）”，表示所有输入文件的代码段。这些代码段被集合在一起，起始运行地址为 0x30000000。

③ 第 4 行定义了一个名为“.rodata”的段，在输出文件 timer_elf 中，它紧挨着“.text”段存放。其中的“ALIGN（4）”表示起始运行地址为 4 字节对齐。假设前面“.text”段的地址范围是 0x30000000～0x300003f1，则“.rodata”段的地址是 4 字节对齐后的 0x300003f4。

④ 第 5、6 行的含义与第 4 行类似。

3.1.3 arm-linux-objcopy 选项

arm-linux-objcopy 被用来复制一个目标文件的内容到另一个文件中，可以使用不同于源文件的格式来输出目的文件，即可以进行格式转换。

在本书中，常用 arm-linux-objcopy 来将 ELF 格式的可执行文件转换为二进制文件。

arm-linux-objcopy 的使用格式如下：

```
arm-linux-objcopy   [ -F bfdname | --target=bfdname ]
                    [ -I bfdname | --input-target=bfdname ]
                    [ -O bfdname | --output-target= bfdname ]
                    [ -S | --strip-all ] [ -g | --strip-debug ]
                    [ -K symbolname | --keep-symbol= symbolname ]
                    [ -N symbolname | --strip-symbol= symbolname ]
                    [ -L symbolname | --localize-symbol= symbolname ]
                    [ -W symbolname | --weaken-symbol= symbolname ]
                    [ -x | --discard-all ] [ -X | --discard-locals ]
                    [ -b byte | --byte= byte ]
                    [ -i interleave | --interleave= interleave ]
                    [ -R sectionname | --remove-section= sectionname ]
                    [ -p | --preserve-dates ] [ --debugging ]
```

```
[ --gap-fill= val ] [ --pad-to= address ]
[ --set-start= val ] [ --adjust-start= incr ]
[ --change-address= incr ]
[ --change-section-address= section{=,+,-} val ]
[ --change-warnings ] [ --no-change-warnings ]
[ --set-section-flags= section= flags ]
[ --add-section= sectionname= filename ]
[ --change-leading char ] [--remove-leading-char ]
[ --weaken ]
[ -v | --verbose ] [ -V | --version ] [ --help ]
input-file [ outfile ]
```

下面讲解常用的选项。

1. input-file、outfile

参数 input-file 和 outfile 分别表示输入目标文件（源目标文件）和输出目标文件（目的目标文件）。如果在命令行中没有明确地指定 outfile，那么 arm-linux-objcopy 将创建一个临时文件来存放目标结果，然后使用 input-file 的名字来重命名这个临时文件（这时候，原来的 input-file 将被覆盖）。

2. -I bfdname 或--input-target=bfdname

用来指明源文件的格式，bfdname 是 BFD 库中描述的标准格式名。如果不指明源文件格式，arm-linux-objcopy 会自己去分析源文件的格式，然后去和 BFD 中描述的各种格式比较，从而得知源文件的目标格式名。

3. -O bfdname 或--output-target=bfdname

使用指定的格式来输出文件，bfdname 是 BFD 库中描述的标准格式名。

4. -F bfdname 或--target=bfdname

同时指明源文件、目的文件的格式。将源目标文件中的内容复制到目的目标文件的过程中，只进行复制不做格式转换，源目标文件是什么格式，目的目标文件就是什么格式。

5. -R sectionname 或--remove-section=sectionname

从输出文件中删掉所有名为 sectionname 的段。这个选项可以多次使用。

注意 不恰当地使用这个选项可能会导致输出文件不可用。

6. -S 或--strip-all

不从源文件中复制重定位信息和符号信息到目标文件中去。

7. -g 或--strip-debug

不从源文件中复制调试符号到目标文件中去。

在编译 bootloader、内核时，常用 arm-linux-objcopy 命令将 ELF 格式的生成结果转换为二进制文件，比如：

```
$ arm-linux-objcopy -O binary -S elf_file bin_file
```

3.1.4 arm-linux-objdump 选项

arm-linux-objdump 用于显示二进制文件信息，本书中常用来查看反汇编代码。

使用格式如下：

```
arm-linux-objdump   [-a] [-b bfdname | --target=bfdname]
                    [-C] [--debugging]
                    [-d] [-D]
                    [--disassemble-zeroes]
                    [-EB|-EL|--endian={big|little}] [-f]
                    [-h] [-i|--info]
                    [-j section | --section=section]
                    [-l] [-m machine ] [--prefix-addresses]
                    [-r] [-R]
                    [-s|--full-contents] [-S|--source]
                    [--[no-]show-raw-insn] [--stabs] [-t]
                    [-T] [-x]
                    [--start-address=address] [--stop-address=address]
                    [--adjust-vma=offset] [--version] [--help]
                    objfile...
```

下面讲解常用的选项：

1. -b bfdname 或--target=bfdname

指定目标码格式。这不是必须的，arm-linux-objdump 能自动识别许多格式。可以使用“arm-linux-objdump –i”命令查看支持的目标码格式列表。

2. --disassemble 或-d

反汇编可执行段（executable sections）。

3. --disassemble-all 或-D

与“-d”类似，反汇编所有段。

4. -EB 或-EL 或--endian={big|little}

指定字节序。

5. --file-headers 或-f

显示文件的整体头部摘要信息。

6. --section-headers、--headers 或-h

显示目标文件各个段的头部摘要信息。

7. --info 或-i

显示支持的目标文件格式和 CPU 架构，它们在“-b”、“-m”选项中用到。

8. --section=name 或-j name

仅显示指定 section 的信息。

9. --architecture=machine 或-m machine

指定反汇编目标文件时使用的架构，当待反汇编文件本身没有描述架构信息的时候（比如 S-records），这个选项很有用。可以用“-i”选项列出这里能够指定的架构。

在调试程序时，常用 arm-linux-objdump 命令来得到汇编代码。本书使用这两个命令：

① 将 ELF 格式的文件转换为反汇编文件：

```
$ arm-linux-objdump -D elf_file > dis_file
```

② 将二进制文件转换为反汇编文件：

```
$ arm-linux-objdump -D -b binary -m arm bin_file > dis_file
```

3.1.5 汇编代码、机器码和存储器的关系以及数据的表示

即使使用 C/C++或者其他高级语言编程，最后也会被编译工具转换为汇编代码，并最终作为机器码存储在内存、硬盘或者其他存储器上。在调试程序时，经常需要阅读它的汇编代码，以下面的汇编代码为例：

```
4bc:    e3a0244e    mov r2, #1308622848 ; 0x4e000000
4c0:    e3a0344e    mov r3, #1308622848 ; 0x4e000000
4c4:    e5933000    ldr r3, [r3]
```

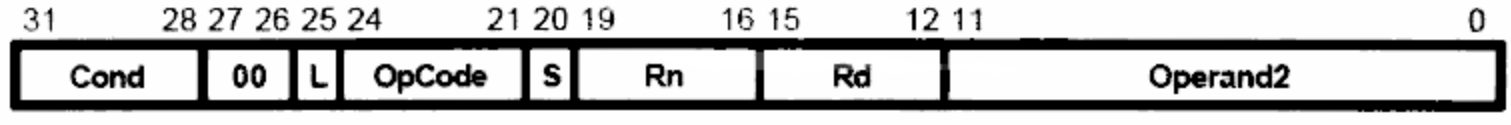
4bc、4c0、4c4 是这些代码在的运行地址，就是说运行前，这些指令必须位于内存中的这些地址上；e3a0244e、e3a0344e、e5933000 是机器码。运行地址、机器码都以十六进制表示。CPU 用到的、内存中保存的都是机器码，图 3.1 是这几条指令在内存中的示意图。

	……
0x4bc	0xe3a0244e
0x4c0	0xe3a0344e
0x4c4	0xe5933000
	……

图 3.1 内存中的机器码

“mov r2, #1308622848”、“mov r3, #1308622848”、“ldr r3, [r3]”是这几个机器码的汇编代码——所谓汇编代码仅仅是为了方便读、写而引入的，机器码和汇编代码之间也仅仅是简单的转换关系。

参考 CPU 的数据手册可知，ARM 的数据处理指令格式为：



以机器码 0xe3a0244e 为例。

① [31:28] = 0b1110，表示这条指令无条件执行。

② [25] = 0b1，表示 Operand2 是一个立即数。

③ [24:21] = 0b1101，表示这是 MOV 指令，即 Rd : = Op2。

④ [20] = 0b0，表示这条指令执行时不影响状态位。

⑤ [15:12] = 0b0010，表示 Rd 就是 r2。

⑥ [11:0] = 0x44e，这是一个立即数。

立即数占据机器码中的低 12 位表示：最低 8 位的值称为 immed_8，高 4 位称为 rotate_imm。立即数的数值计算方法为：<immediate>=immed_8 循环右移（2*rotate_imm）。对于“[11:0] = 0x44e”，其中 immed_8=0x4e、rotate_imm=0x4，所以此立即数等于 0x4e000000。

综上所述，机器码 0xe3a0244e 的汇编代码为：

```
mov r2, #0x4e000000
即
mov r2, #1308622848
```

上面的 0x4e000000 和 1308622848 是一样的，之所以强调这点，是因为很多初学者问这样的问题：“计算机中怎么以十六进制保存数据？以十六进制、十进制保存数据有什么区别？”这类问题与如下问题相似：桌子上有 12 个苹果，吃了一个，请问现在还有几个？你可以回答 11 个、0xb 个、十一个、eleven 个、拾壹个。所谓十六进制、十进制、八进制、二进制，都仅仅是对同一个数据的不同表达形式而已，这些不同的表达形式也仅仅是为了方便读写而已，它们所表示的数值及它在计算机中的保存方式是完全一样的。

3.2 Makefile 介绍

在 Linux 中使用 make 命令来编译程序，特别是大程序；而 make 命令所执行的动作依赖于 Makefile 文件。最简单的 Makefile 文件如下：

```
hello: hello.c
    gcc -o hello hello.c
clean:
    rm -f  hello
```

将上述 4 行存为 Makefile 文件（注意必须以 Tab 键缩进第 2、4 行，不能以空格键缩进），放入/work/hardware/hello 目录下，然后直接执行 make 命令即可编译程序，执行“make clean”即可清除编译出来的结果。

make 命令根据文件更新的时间戳来决定哪些文件需要重新编译，这使得可以避免编译已经编译过的、没有变化的程序，可以大大提高编译效率。

要想完整地了解 Makefile 的规则，请参考《GNU Make 使用手册》，以下仅粗略介绍。

3.2.1 Makefile 规则

一个简单的 Makefile 文件包含一系列的“规则”，其样式如下：

```
目标(target)…: 依赖(prerequiries)…
<tab>命令(command)
```

目标（target）通常是要生成的文件的名称，可以是可执行文件或 OBJ 文件，也可以是一个执行的动作名称，诸如“clean”。

依赖是用来产生目标的材料（比如源文件），一个目标经常有几个依赖。

命令是生成目标时执行的动作，一个规则可以含有几个命令，每个命令占一行。

注意 每个命令行前面必须是一个 Tab 字符，即命令行第一个字符是 Tab。这是容易出错的地方。

通常，如果一个依赖发生了变化，就需要规则调用命令以更新或创建目标。但是并非所有的目标都有依赖，例如，目标“clean”的作用是清除文件，它没有依赖。

规则一般是用于解释怎样和何时重建目标。make 首先调用命令处理依赖，进而才能创建或更新目标。当然，一个规则也可以是用于解释怎样和何时执行一个动作，即打印提示信息。

一个 Makefile 文件可以包含规则以外的其他文本，但一个简单的 Makefile 文件仅仅需要包含规则。虽然真正的规则比这里展示的例子复杂，但格式是完全一样的。

对于上面的 Makefile，执行“make”命令时，仅当 hello.c 文件比 hello 文件新，才会执行命令“arm-linux-gcc –o hello hello.c”生成可执行文件 hello；如果还没有 hello 文件，这个命令也会执行。

运行“make clean”时，由于目标 clean 没有依赖，它的命令“rm -f hello”将被强制执行。

3.2.2 Makefile 文件里的赋值方法

变量的定义语法形式如下：

```
immediate = deferred
immediate ?= deferred
immediate := immediate
immediate += deferred or immediate
define immediate
deferred
endef
```

在 GNU make 中对变量的赋值有两种方式：延时变量、立即变量。区别在于它们的定义方式和扩展时的方式不同，前者在这个变量使用时才扩展开，意即当真正使用时这个变量的值才确定；后者在定义时它的值就已经确定了。使用“=”、“?=”定义或使用 define 指令定义的变量是延时变量；使用“:=”定义的变量是立即变量。需要注意的一点是“?=”仅仅在变量还没有定义的情况下有效，即“?=”用来定义第一次出现的延时变量。

对于附加操作符“+=”，右边变量如果在前面使用（:=）定义为立即变量则它也是立即变量，否则均为延时变量。

3.2.3 Makefile 常用函数

函数调用的格式如下：

```
$(function arguments)
```

这里“function”是函数名，“arguments”是该函数的参数。参数和函数名之间用空格或Tab隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。

内核的Makefile中用到大量的函数，现在介绍一些常用的。

1. 字符串替换和分析函数

（1）$（subst from,to,text）。

在文本“text”中使用“to”替换每一处“from”。

比如：

```
$(subst ee,EE,feet on the street)
```

结果为“fEEt on the street”。

（2）$（patsubst pattern,replacement,text）。

寻找“text”中符合格式“pattern”的字，用“replacement”替换它们。“pattern”和“replacement”中可以使用通配符。

比如：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

结果为：“x.c.o bar.o”。

（3）$（strip string）。

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。

比如：

```
$(strip a   b c )
```

结果为“a b c”。

（4）$（findstring find,in）。

在字符串“in”中搜寻“find”，如果找到，则返回值是“find”，否则返回值为空。

比如：

```
$(findstring a,a b c)
$(findstring a,b c)
```

将分别产生值“a”和“　”。

（5）$（filter pattern...,text）。

返回在“text”中由空格隔开且匹配格式“pattern...”的字，去除不符合格式“pattern...”的字。

比如：

```
$(filter %.c %.s,foo.c bar.c baz.s ugh.h)
```

结果为“foo.c bar.c baz.s”。

（6）$（filter-out pattern...,text）。

返回在“text”中由空格隔开且不匹配格式“pattern...”的字，去除符合格式“pattern...”

的字。它是函数 filter 的反函数。

比如：

```
$(filter %.c %.s,foo.c bar.c baz.s ugh.h)
```

结果为“ugh.h”。

（7）$（sort list）。

将“list”中的字按字母顺序排序，并去掉重复的字。输出由单个空格隔开的字的列表。

比如：

```
$(sort foo bar lose)
```

返回值是“bar foo lose”。

2. 文件名函数

（1）$（dir names...）。

抽取“names...”中每一个文件名的路径部分，文件名的路径部分包括从文件名的首字符到最后一个斜杠（含斜杠）之前的一切字符。

比如：

```
$(dir src/foo.c hacks)
```

结果为“src/ ./”。

（2）$（notdir names...）。

抽取“names...”中每一个文件名中除路径部分外一切字符（真正的文件名）。

比如：

```
$(notdir src/foo.c hacks)
```

结果为“foo.c hacks”。

（3）$（suffix names...）。

抽取“names...”中每一个文件名的后缀。

比如：

```
$(suffix src/foo.c src-1.0/bar.c hacks)
```

结果为“.c .c”。

（4）$（basename names...）。

抽取“names...”中每一个文件名中除后缀外一切字符。

比如：

```
$(basename src/foo.c src-1.0/bar hacks)
```

结果为“src/foo src-1.0/bar hacks”。

（5）$（addsuffix suffix,names...）。

参数“names...”是一系列的文件名，文件名之间用空格隔开；suffix 是一个后缀名。将 suffix（后缀）的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用

单个空格隔开。

比如：

```
$(addsuffix .c,foo bar)
```

结果为“foo.c bar.c”。

（6）$（addprefix prefix,names...）。

参数“names”是一系列的文件名，文件名之间用空格隔开；prefix 是一个前缀名。将 preffix（前缀）的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。

比如：

```
$(addprefix src/,foo bar)
```

结果为“src/foo src/bar”。

（7）$（wildcard pattern）。

参数“pattern”是一个文件名格式，包含有通配符（通配符和 shell 中的用法一样）。函数 wildcard 的结果是一列和格式匹配且真实存在的文件的名称，文件名之间用一个空格隔开。

比如若当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(wildcard *.c)
```

结果为“1.c 2.c”。

3. 其他函数

（1）$（foreach var,list,text）。

前两个参数，“var”和“list”将首先扩展，最后一个参数“text”此时不扩展；接着，“list”扩展所得的每个字都赋给“var”变量；然后“text”引用该变量进行扩展，因此“text”每次扩展都不相同。

函数的结果是由空格隔开的“text”在“list”中多次扩展后，得到的新的“list”，就是说：“text”多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数 foreach 的返回值。

下面是一个简单的例子，将变量“files”的值设置为“dirs”中的所有目录下的所有文件的列表：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

这里“text”是“$(wildcard $(dir)/*）”，它的扩展过程如下。

① 第一个赋给变量 dir 的值是“a”，扩展结果为“$(wildcard a/*)”；

② 第二个赋给变量 dir 的值是“b”，扩展结果为“$(wildcard b/*)”；

③ 第三个赋给变量 dir 的值是“c”，扩展结果为“$(wildcard c/*)”；

④ 如此继续扩展。

这个例子和下面的例子有共同的结果：

```
files := $(wildcard a/* b/* c/* d/*)
```

（2）$（if condition,then-part[,else-part]）。

首先把第一个参数“condition”的前导空格、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件“condition”为真；如果扩展为空字符串，则条件“condition”为假。

如果条件“condition”为真，那么计算第二个参数“then-part”的值，并将该值作为整个函数 if 的值。

如果条件“condition”为假，并且第三个参数存在，则计算第三个参数“else-part”的值，并将该值作为整个函数 if 的值；如果第三个参数不存在，函数 if 将什么也不计算，返回空值。

注意 仅能计算“then-part”和“else-part”二者之一，不能同时计算。这样有可能产生副作用（例如函数 shell 的调用）。

（3）$（origin variable）。

变量“variable”是一个查询变量的名称，不是对该变量的引用。所以，不能采用“$”和圆括号的格式书写该变量，当然，如果需要使用非常量的文件名，可以在文件名中使用变量引用。

函数 origin 的结果是一个字符串，该字符串变量的定义如下：

```
undefined                    : 变量"variable"从来没有定义;
default                      : 变量"variable"是默认定义;
environment                  : 变量"variable"作为环境变量定义，选项"-e"没有打开;
environment override         : 变量"variable"作为环境变量定义，选项"-e"已打开;
file                         : 变量"variable"在 Makefile 中定义;
command line                 : 变量"variable"在命令行中定义;
override                     : 变量"variable"在 Makefile 中用 override 指令定义;
automatic                    : 变量"variable"是自动变量
```

（4）$（shell command arguments）。

函数 shell 是 make 与外部环境的通信工具。函数 shell 的执行结果和在控制台上执行“command arguments”的结果相似。不过如果“command arguments”的结果含有换行符（和回车符），则在函数 shell 的返回结果中将把它们处理为单个空格，若返回结果最后是换行符（和回车符）则被去掉。

比如当前目录下有文件 1.c、2.c、1.h、2.h，则：

```
c_src := $(shell ls *.c)
```

结果为“1.c 2.c”。

本节可以在阅读内核、Bootloader、应用程序的 Makefile 文件时作为手册来查询。下面以 options 程序的 Makefile 作为例进行演示，Makefile 的内容如下：

```
File: Makefile
01 src  := $(shell ls *.c)
02 objs := $(patsubst %.c,%.o,$(src))
03
04 test: $(objs)
05      gcc -o $@ $^
06
07 %.o:%.c
08      gcc -c -o $@ $<
09
10 clean:
11      rm -f test *.o
```

上述 Makefile 中“$@”、“$^”、“$<”称为自动变量。“$@”表示规则的目标文件名；“$^”表示所有依赖的名字，名字之间用空格隔开；“$<”表示第一个依赖的文件名。“%”是通配符，它和一个字符串中任意个数的字符相匹配。

options 目录下所有的文件为 main.c，Makefile，sub.c 和 sub.h，下面一行行地分析。

① 第 1 行 src 变量的值为“main.c sub.c”。

② 第 2 行 objs 变量的值为“main.o sub.o”，是 src 变量经过 patsubst 函数处理后得到的。

③ 第 4 行实际上就是：

```
test : main.o sub.o
```

目标 test 的依赖为 main.o 和 sub.o。开始时这两个文件还没有生成，在执行生成 test 的命令之前先将 main.o、sub.o 作为目标查找到合适的规则，以生成 main.o、sub.o。

④ 第 7、8 行就是用来生成 main.o、sub.o 的规则。

对于 main.o 这个规则就是：

```
main.o:main.c
    gcc -c -o main.o main.c
```

对于 sub.o 这个规则就是：

```
sub.o:sub.c
    gcc -c -o sub.o sub.c
```

这样，test 的依赖 main.o 和 sub.o 就生成了。

⑤ 第 5 行的命令在生成 main.o、sub.o 后得以执行。

在 options 目录下第一次执行 make 命令可以看到如下信息：

```
gcc -c -o main.o main.c
gcc -c -o sub.o sub.c
gcc -o test main.o sub.o
```

然后修改 sub.c 文件，再次执行 make 命令，可以看到如下信息：

```
gcc -c -o sub.o sub.c
gcc -o test main.o sub.o
```

可见，只编译了更新过的 sub.c 文件，对 main.c 文件不用再次编译，节省了编译的时间。

3.3 常用 ARM 汇编指令及 ATPCS 规则

3.3.1 本书使用的所有汇编指令

在嵌入式开发中，汇编程序常常用于非常关键的地方，比如系统启动时的初始化，进出中断时的环境保存、恢复，对性能要求非常苛刻的函数等。

在 S3C2410、S3C2440 的数据手册中，对各种汇编指令的作用及使用方法都有详细说明。本书用到的汇编指令比较少，下面分别介绍。

1. 相对跳转指令：b、bl

这两条指令的不同之处在于 bl 指令除了跳转之外，还将返回地址（bl 的下一条指令的地址)保存在 lr 寄存器中。

这两条指令的可跳转范围是当前指令的前后 32MB。

它们是位置无关的指令。

使用示例：

```
    b fun1
……
fun1:
    bl fun2
……
fun2:
……
```

2. 数据传送指令 mov，地址读取伪指令 ldr

mov 指令可以把一个寄存器的值赋给另一个寄存器，或者把一个常数赋给寄存器。例子如下：

```
mov r1, r2              /* r1=r2   */
mov r1, #4096           /* r1=4096 */
```

mov 指令传送的常数必须能用立即数来表示。

当不知道一个数能否用“立即数”来表示时，可以使用 ldr 命令来赋值。ldr 是伪指令，它不是真实存在的指令，编译器会把它扩展成真正的指令：如果该常数能用“立即数”来表示，则使用 mov 指令；否则编译时将该常数保存在某个位置，使用内存读取指令把它读出来。

例子如下：

```
ldr r1, =4097            /* r1=4097 */
```

ldr 本意为“大范围的地址读取伪指令”，上面的例子使用它来将常数赋给寄存器 r1。下面的例子是获得代码的绝对地址：

```
    ldr r1, =label
label:
……
```

3. 内存访问指令：ldr、str、ldm、stm

注意 ldr 指令既可能是前面所述的大范围的地址读取伪指令，也可能是内存访问指令。当它的第二个参数前面有“=”时，表示伪指令，否则表示内存访问指令。

ldr 指令从内存中读取数据到寄存器，str 指令把寄存器的值存储到内存中，它们操作的数据都是 32 位的。示例如下：

```
ldr r1, [r2, #4]     /* 将地址为 r2+4 的内存单元数据读取到 r1 中 */
ldr r1, [r2]         /* 将地址为 r2 的内存单元位数据读取到 r1 中 */
ldr r1, [r2], #4     /* 将地址为 r2 的内存单元数据读取到 r1 中，然后 r2=r2+4 */
str r1, [r2, #4]     /* 将 r1 的数据保存到地址为 r2+4 的内存单元中 */
str r1, [r2]         /* 将 r1 的数据保存到地址为 r2 的内存单元中 */
str r1, [r2], #4     /* 将 r1 的数据保存到地址为 r2 的内存单元中，然后 r2=r2+4 */
```

ldm 和 stm 属于批量内存访问指令，只用一条指令就可以读写多个数据。它们的格式如下：

```
ldm{cond}<addressing_mode> <rn>{!} <register list>{^}
stm{cond}<addressing_mode> <rn>{!} <register list>{^}
```

其中{cond}表示指令的执行条件，参考表 3.2。

<addressing_mode>表示地址变化模式，有以下 4 种方式。

① ia（Increment After）：事后递增方式。

② ib（Increment Before）：事先递增方式。

③ da（Decrement After）：事后递减方式。

④ db（Decrement Before）：事先递减方式。

<rn>中保存内存的地址，如果后面加上了感叹号，指令执行后，rn 的值会更新，等于下一个内存单元的地址。

<register list>表示寄存器列表，对于 ldm 指令，从<rn>所对应的内存块中取出数据，写入这些寄存器；对于 stm 指令，把这些寄存器的值写入<rn>所对应的内存块中。

{^}有两种含义：如果<register list>中有 pc 寄存器，它表示指令执行后，spsr 寄存器的值将自动复制到 cpsr 寄存器中——这常用于从中断处理函数中返回；如果<register list>中没有 pc 寄存器，{^}表示操作的是用户模式下的寄存器，而不是当前特权模式的寄存器。

指令中寄存器列表和内存单元的对应关系为：编号低的寄存器对应内存中的低地址单

元，编号高的寄存器对应内存中的高地址单元。

ldm 和 stm 指令示例如下：

```
HandleIRQ:                          @ 中断入口函数
    sub lr, lr, #4                  @ 计算返回地址
    stmdb   sp!,    { r0-r12,lr }   @ 保存使用到的寄存器
                                    @ r0-r12,lr 被保存在 sp 表示的内存中
                                    @ "!" 使得指令执行后 sp=sp-14*4

     ldr lr, =int_return            @ 设置调用 IRQ_Handle 函数后的返回地址
     ldr pc, =IRQ_Handle            @ 调用中断分发函数
int_return:
     ldmia   sp!,    { r0-r12,pc }^ @ 中断返回,"^"表示将 spsr 的值复制到 cpsr
                                    @ 于是从 irq 模式返回被中断的工作模式
                                    @ "!" 使得指令执行后 sp=sp+14*4
```

4. 加减指令：add、sub

例子如下：

```
add r1, r2, #1      /* 表示 r1=r2+1，即寄存器 r1 的值等于寄存器 r2 的值加上 1 */
sub r1, r2, #1      /* 表示 r1=r2-1 */
```

5. 程序状态寄存器的访问指令：msr、mrs

ARM 处理器有一个程序状态寄存器（cpsr），它用来控制处理器的工作模式、设置中断的总开关。示例如下：

```
msr cpsr, r0        /* 复制 r0 到 cpsr 中 */
mrs r0, cpsr        /* 复制 cpsr 到 r0 中 */
```

在第 9 章会描述 cpsr 寄存器的格式。

6. 其他伪指令

在本书的汇编程序中，常常见到如下语句：

```
.extern     main
.text
.global _start
_start:
```

".extern" 定义一个外部符号（可以是变量也可以是函数），上面的代码表示本文件中引用的 main 是一个外部函数。

".text" 表示下面的语句都属于代码段。

".global" 将本文件中的某个程序标号定义为全局的，比如上面的代码表示_start 个全局函数。

7. 汇编指令的执行条件

大多数 ARM 指令都可以条件执行，即根据 cpsr 寄存器中的条件标志位决定是否执行该指令：如果条件不满足，该指令相当于一条 nop 指令。

每条 ARM 指令包含 4 位的条件码域，这表明可以定义 16 个执行条件。可以将这些执行条件的助记符附加在汇编指令后，比如 moveq、movgt 等。这 16 个条件码和它们的助记符、含义如表 3.2 所示。

表 3.2　　指令的条件码

条件码（cond）	助 记 符	含　义	cpsr 中条件标志位
0000	eq	相等	Z = 1
0001	ne	不相等	Z = 0
0010	cs/hs	无符号数大于/等于	C = 1
0011	cc/lo	无符号数小于	C = 0
0100	mi	负数	N = 1
0101	pl	非负数	N = 0
0110	vs	上溢出	V = 1
0111	vc	没有上溢出	V = 0
1000	hi	无符号数大于	C = 1 且 Z = 0
1001	ls	无符号数小于等于	C = 0 或 Z = 1
1010	ge	带符号数大于等于	N = 1，V = 1 或 N = 0，V = 0
1011	lt	带符号数小于	N = 1，V = 0 或 N = 0，V = 1
1100	gt	带符号数大于	Z = 0 且 N = V
1101	le	带符号数小于/等于	Z = 1 或 N! = V
1110	al	无条件执行	-
1111	nv	从不执行	-

表中的 cpsr 条件标志位 N、Z、C、V 分别表示 Negative、Zero、Cary、oVerflow。影响条件标志位的因素比较多，比如比较指令 cmp、cmn、teq 及 tst 等。

3.3.2 ARM-THUMB 子程序调用规则 ATPCS

为了使 C 语言程序和汇编程序之间能够互相调用，必须为子程序间的调用制定规则，在 ARM 处理器中，这个规则被称为 ATPCS：ARM 程序和 Thumb 程序中子程序调用的规则。基本的 ATPCS 规则包括寄存器使用规则、数据栈使用规则、参数传递规则。

1. 寄存器使用规则

ARM 处理器中有 r0～r15 共 16 个寄存器，它们的用途有一些约定的习惯，并依据这些用途定义了别名，如表 3.3 所示。

表 3.3 ATPCS 中各寄存器的使用规则及其名称

寄存器	别名	使用规则
r15	pc	程序计数器
r14	lr	连接寄存器
r13	sp	数据栈指针
r12	ip	子程序内部调用的 scratch 寄存器
r11	v8	ARM 状态局部变量寄存器 8
r10	v7、sl	ARM 状态局部变量寄存器 7、在支持数据栈检查的 ATPCS 中为数据栈限制指针
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
r8	v5	ARM 状态局部变量寄存器 5
r7	v4、wr	ARM 状态局部变量寄存器 4、Thumb 状态工作寄存器
r6	v3	ARM 状态局部变量寄存器 3
r5	v2	ARM 状态局部变量寄存器 2
r4	v1	ARM 状态局部变量寄存器 1
r3	a4	参数/结果/scratch 寄存器 4
r2	a3	参数/结果/scratch 寄存器 3
r1	a2	参数/结果/scratch 寄存器 2
r0	a1	参数/结果/scratch 寄存器 1

寄存器的使用规则总结如下。

- 子程序间通过寄存器 r0～r3 来传递参数，这时可以使用它们的别名 a0～a3。被调用的子程序返回前无需恢复 r0～r3 的内容。
- 在子程序中，使用 r4～r11 来保存局部变量，这时可以使用它们的别名 v1～v8。如果在子程序中使用了它们的某些寄存器，子程序进入时要保存这些寄存器的值，在返回前恢复它们；对于子程序中没有使用到的寄存器，则不必进行这些操作。在 Thumb 程序中，通常只能使用寄存器 r4～r7 来保存局部变量。
- 寄存器 r12 用作子程序间 scratch 寄存器，别名为 ip。
- 寄存器 r13 用作数据栈指针，别名为 sp。在子程序中寄存器 r13 不能用作其他用途。它的值在进入、退出子程序时必须相等。
- 寄存器 r14 称为连接寄存器，别名为 lr。它用于保存子程序的返回地址。如果在子程序中保存了返回地址（比如将 lr 值保存到数据栈中），r14 可以用作其他用途。
- 寄存器 r15 是程序计数器，别名为 pc。它不能用作其他用途。

2. 数据栈使用规则

数据栈有两个增长方向：向内存地址减小的方向增长时，称为 DESCENDING 栈；向内

地址增加的方向增长时，称为 ASCENDING 栈。

所谓数据栈的增长就是移动栈指针。当栈指针指向栈顶元素（最后一个入栈的数据）时，称为 FULL 栈；当栈指针指向栈顶元素（最后一个入栈的数据）相邻的一个空的数据单元时，称为 EMPTY 栈。

综合这两个特点，数据栈可以分为以下 4 种。

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 类型，并且对数据栈的操作是 8 字节对齐的。使用 stmdb/ldmia 批量内存访问指令来操作 FD 数据栈。

使用 stmdb 命令往数据栈中保存内容时，先递减 sp 指针，再保存数据，使用 ldmia 命令从数据栈中恢复数据时，先获得数据，再递增 sp 指针，sp 指针总是指向栈顶元素，这刚好是 FD 栈的定义。

3. 参数传递规则

一般来说，当参数个数不超过 4 个时，使用 r0～r3 这 4 个寄存器来传递参数；如果参数个数超过 4 个，剩余的参数通过数据栈来传递。

对于一般的返回结果，通常使用 a0～a3 来传递。

示例：

假设 CopyCode2SDRAM 函数是用 C 语言实现的，它的数据原型如下：

```
int CopyCode2SDRAM(unsigned char *buf, unsigned long start_addr, int size)
```

在汇编代码中，使用下面的代码调用它，并判断返回值。

```
01 ldr r0, =0x30000000      @ 1. 目标地址 = 0x30000000，这是 SDRAM 的起始地址
02 mov r1, #0               @ 2. 源地址   = 0
03 mov r2, #16*1024         @ 3. 复制长度 = 16K
04 bl  CopyCode2SDRAM       @ 调用 C 函数 CopyCode2SDRAM
05 cmp a0, #0               @ 判断函数返回值
……
```

第 1 行将 r0 设为 0x30000000，则 CopyCode2SDRAM 函数执行时，它的第一个参数 buf 的指向的内存地址为 0x30000000。

第 2 行将 r1 设为 0，CopyCode2SDRAM 函数的第二个参数 start_addr 等于 0。

第 3 行将 r2 设为 16*1024，CopyCode2SDRAM 函数的第三个参数 start_addr 等于 16*1024。

第 5 行判断返回值。

第4章 Windows、Linux环境下相关工具、命令的使用

本章目标

- 掌握 Windows 下的代码阅读、编辑工具 Source Insight
- 掌握在 Windows 下与 Linux 进行交互的工具：Cuteftp、SecureCRT
- 掌握 Linux 下的代码阅读、编辑工具 KScope，串口工具 C-kermit
- 掌握一些常用的 Linux 命令

4.1 Windows 环境下的工具介绍

仅仅将 Linux 作为服务器来进行开发时（比如使用 VMware 运行 Linux 以提供编译环境，日常工作仍然在 Windows 下进行），可以使用 Windows 下的几款优秀工具提高工作效率。比如使用 Source Insight 阅读、编辑代码；使用 Cuteftp 与 Linux 服务器进行文件传输；使用 SecureCRT 远程登录 Linux 进行各类操作。

4.1.1 代码阅读、编辑工具 Source Insight

Source Insight 是一款极具革命性的代码阅读、编辑工具，它内建了 C/C++、C#、Java 等多种编程语言的分析器。Source Insight 会自动分析源代码，动态地生成、更新一个数据库，并通过丰富而有效的表现形式使得阅读、编辑代码非常方便、高效。比如它会将 C 语言中全局变量、局部变量标上不同的颜色；光标移到某个变量、函数上时，窗口下方会自动显示它们的定义；借助于不断更新的数据库，可以快速地找到函数的调用关系；编辑代码时，变量名、函数名会自动补全。

基于这些功能，在 Windows 下阅读 Linux 内核源码这类庞大的软件时，使用 Source Insight 有助于理清各类综错复杂的变量、函数之间的关系。

从网址 http://www.sourceinsight.com/上可以下载一个试用版本，它具有正式版的全部功能，试用期为 30 天。

下面以 Linux 内核源码为例介绍 Source Insight 的使用。

1. 创建一个 Source Insight 工程

启动 Source Insight 之后，它默认的支持文件中没有以“.S”结尾的汇编语言文件，单击菜单“Options”→“Document Options”，在弹出的对话框中选择“Document Type”为“C Source File”，在“File filter”中添加“*.S”类型，如图 4.1 所示。

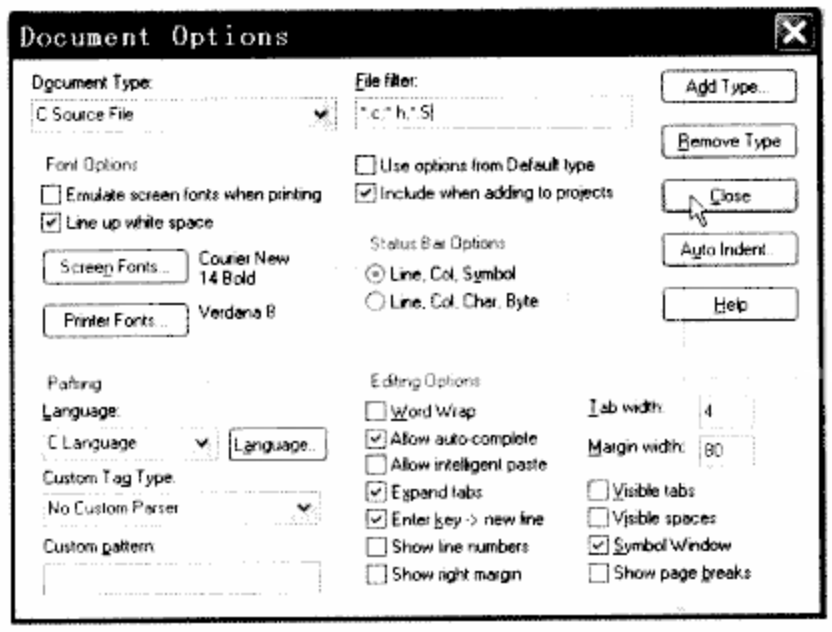


图 4.1 设置 Source Insight 支持的文件类型

然后单击菜单“Project”→“New Project”，开始建立一个新的工程，界面如图 4.2 所示。

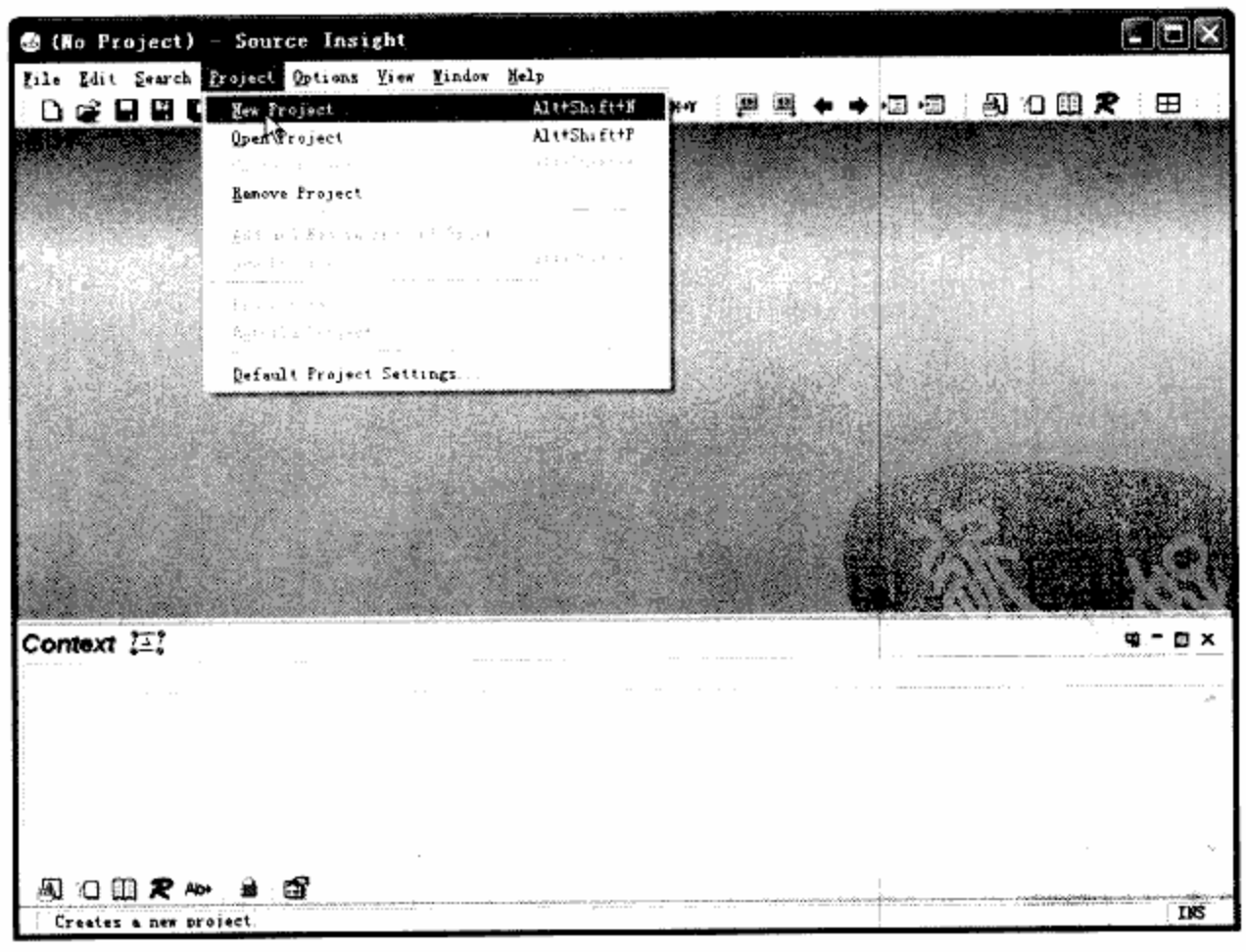


图 4.2 新建一个 Source Insight 工程

在随后出现的界面中，输入工程的名称和工程数据的存放位置。本小节中，假设内核源码位置为 E:\kernel_projects\linux-2.6.22.6，将要建立的 Source Insight 工程命名为

linux-2.6.22.6，在 E:\kernel_projects\sc 目录下存放工程数据，则如图 4.3 一样设置，然后单击“OK”按钮（如果 E:\kernel_projects\sc 目录还不存在，会提示是否创建这个目录）。

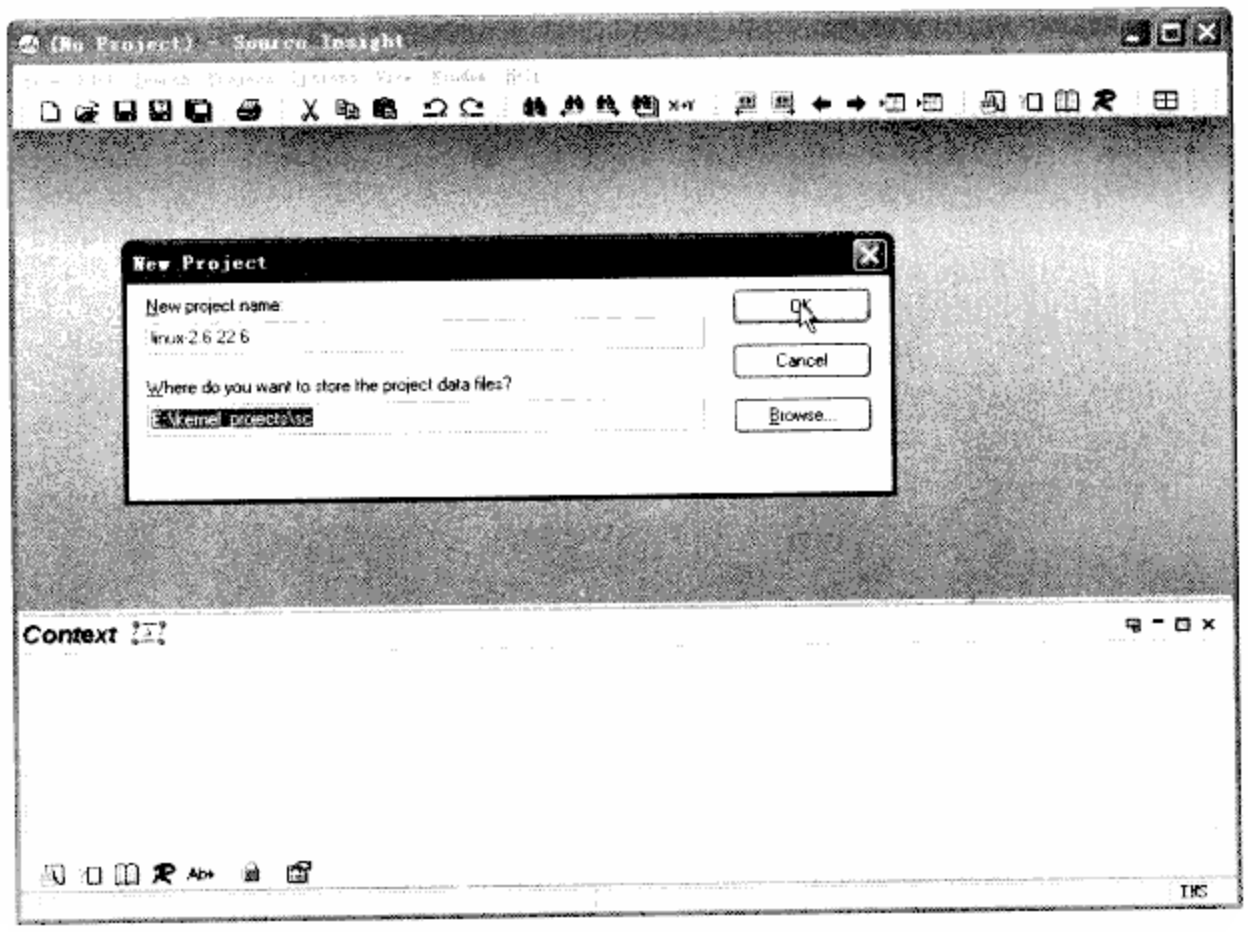


图 4.3 输入 Source Insight 工程名称及保存位置

接下来的步骤是指定源码的位置及添加源文件。如图 4.4 所示，指定内核源码位置后，单击“OK”按钮进入下一个设置界面。

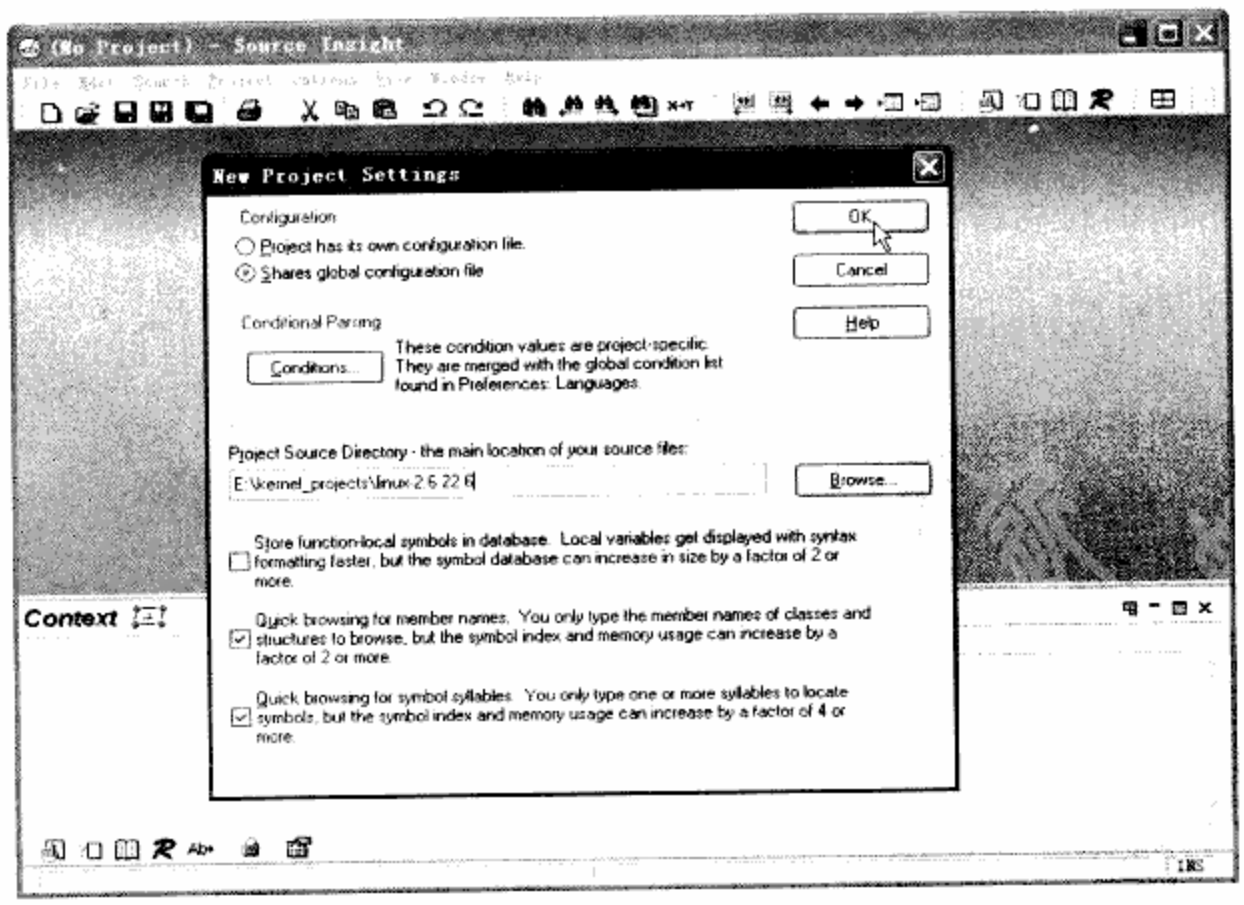


图 4.4 指定源码位置

图 4.5 是添加源文件的操作界面：先单击“Add All”按钮，在弹出的对话柜中选中“Include top level sub-directories”（表示将添加第一层子目录中的文件）、“Recursively add lower

sub-directories”（表示递归地加入底层的子目录，即加入所有子目录中的文件）；然后单击“OK”按钮控制开始加入内核的所有源文件。

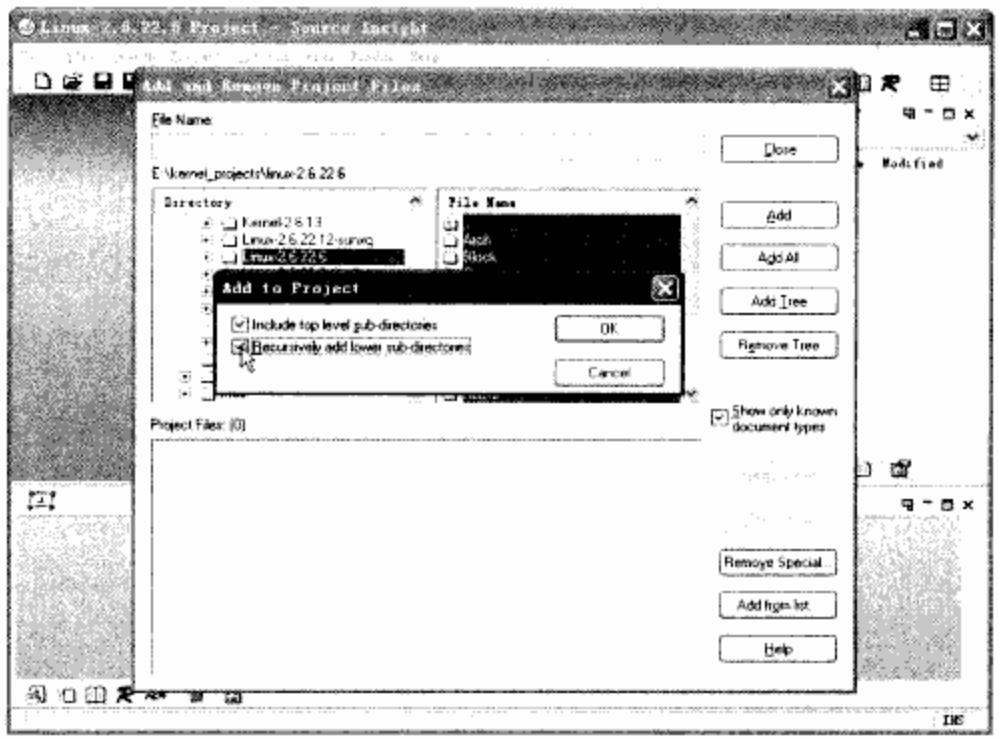


图 4.5 添加源文件

实际上，由于内核支持多个架构的 CPU、多个型号的目标板，而本书只关心 S3C2410、S3C2440 目标板，所以可以在 Source Insight 工程中去除其他不相关的文件。仍然在与图 4.5 相似的界面中（可以单击菜单“Projects”→“Add and Remove Project Files”进入），选择某个目录后，单击“Remove Tree”按钮将整个目录下的文件从工程中移除。

要移除的目录如下，操作的示例界面如图 4.6 所示。

① arch 目录下除 arm 外的所有子目录；

② arch/arm 目录下以“mach-”开头的目录（mach-s3c2410、mach-s3c2440 除外）；

③ arch/arm 目录下以“plat-”开头的目录（plat-s3c24xx 除外）；

④ include 目录下以“asm-”开头的目录（asm-arm、asm-generic 除外）；

⑤ include/asm-arm 目录下以“arch-”开头的目录（arch-s3c2410 除外）。

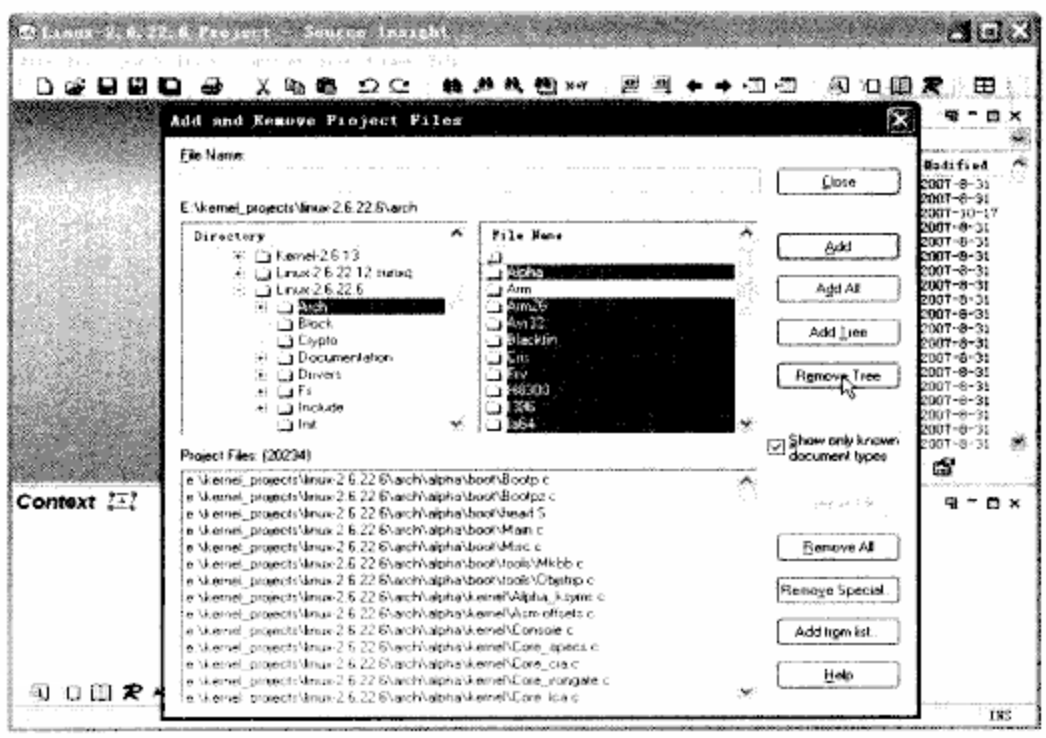


图 4.6 移除源文件

至此，Source Insight 工程建立完毕。

2．“同步”源文件

所谓“同步”源文件就是在 Source Insight 工程中建立一个数据库，它里面保存有源文件中各变量、函数之间的关系，使得阅读、编辑代码时能快速地提供各种辅助信息（比如以不同颜色显示不同类型的变量等）。

这个数据库会自动建立，但是对于比较庞大的源码的工程，建议初次使用时手工建立数据库，这使得 Source Insight 工程很快地建立所有源码的、全面的关系图。

单击菜单“Project”→“Synchronize Files”，会弹出如图 4.7 所示的对话框，选中其中的“Force all files to be re-parsed”（表示“强制分析所有文件”），然后单击“OK”按钮即可生成数据库。

图 4.7　同步源文件

3．Source Insight 工程使用示例

在 Source Insight 右边的文件列表中选择打开 s3c2410fb.c 文件，可以得到如图 4.8 所示的界面，它的中间是主窗口，可以在里面阅读、编辑代码；左边是“Symbol window”（符号窗口），可以从中快速地找到当前文件中的变量、函数，宏定义等；下边是“Context window”（上下文窗口），在主窗口中将光标放在某个变量、函数、宏上面时，会在这个窗口中显示它们的定义，比如在图 4.8 中，这个窗口中显示了 request_irq 函数的定义。

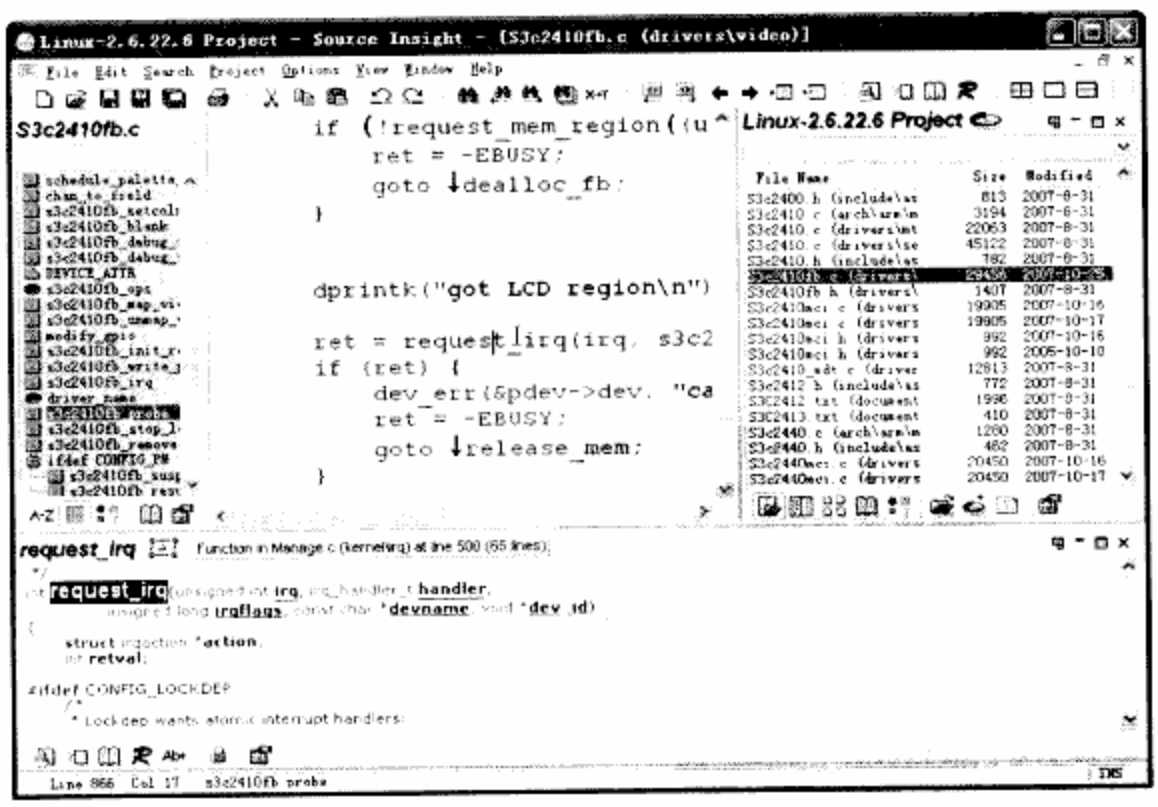


图 4.8　Source Insight 的使用界面

在主窗口中，按住“Ctrl”键的同时，单击某个变量、函数、宏，就可以跳到定义它们的位置；双击上下文窗口也可以达到同样的效果。

同时按住“Alt”、“，”键可以令主窗口退回上一画面，同时按住“Alt”、“.”键可以令主窗口前进到前一个画面。

在某个变量、函数、宏上单击右键，在弹出的菜单中选择“Lookup References”，可以快速地在所有源文件中找到对它们的引用，这比搜索整个源码目录快多了。

Source Insight 还有很多使用技巧，上面只介绍了几种常用的技巧，读者在使用过程中可以通过各个菜单了解更多。

4.1.2 文件传输工具 Cuteftp

Cuteftp 是一款 FTP 客户端软件，只要在 Linux 上安装、启动了 FTP 服务，就可以使用 Cuteftp 在 Windows 与 Linux 之间进行文件传输。

Cuteftp 专业版可以从网址 http://www.cuteftp.com/downloads/cuteftppro.aspx 中下载，有 30 天的免费试用期。

安装、启动 Cuteftp 后，跳过前面的设置向导。假设 Linux 服务器的 IP 为 192.168.1.57，用户名为 book，密码为 123456，如图 4.9 所示在“Host”、“Username”、“Password”中分别填入 192.168.1.57、book、123456，然后回车就可以连接上 Linux 服务器了。

在如图 4.9 所示界面中，可以在左、右两边的窗口中拖拽文件进行上传、下载。

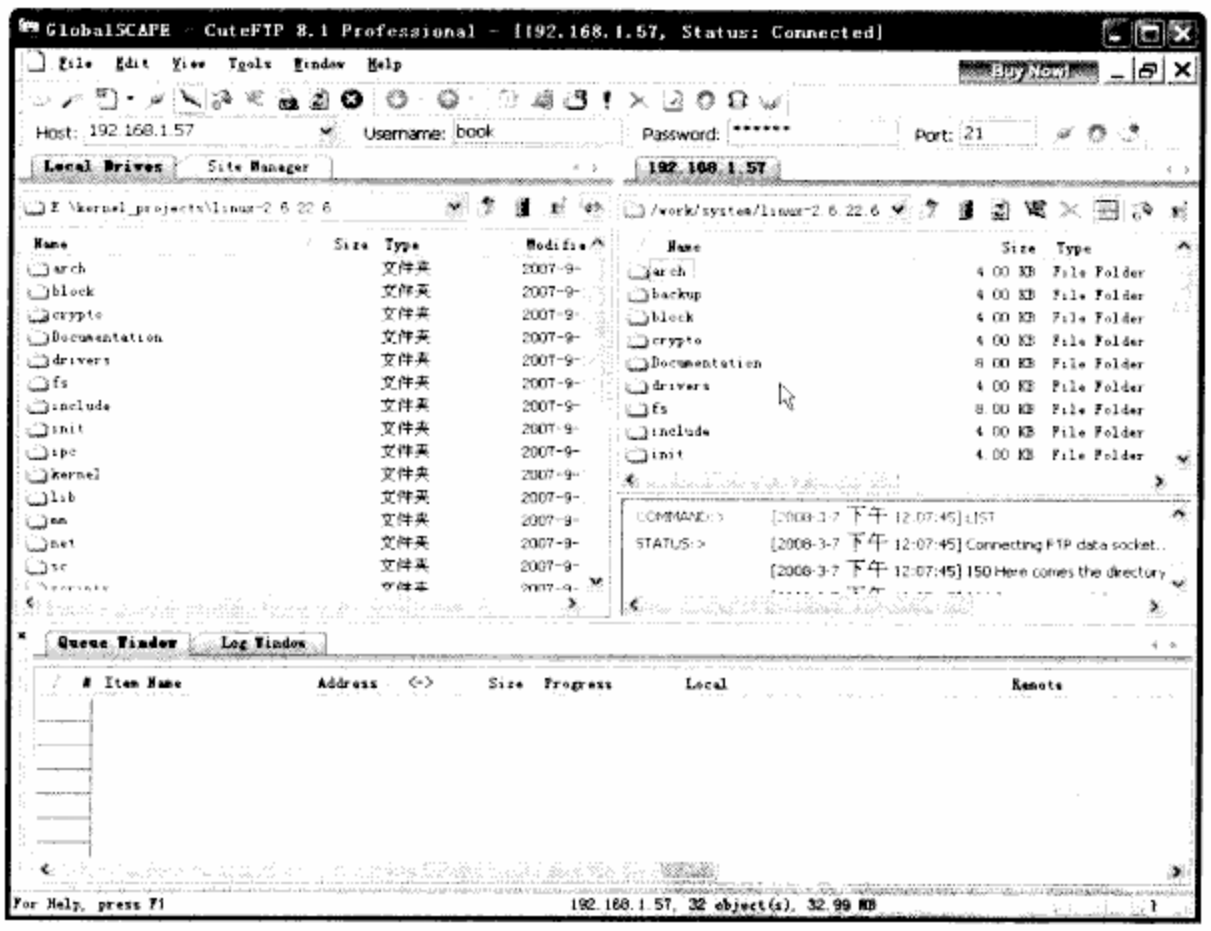


图 4.9 Cuteftp 的使用界面

4.1.3 远程登录工具 SecureCRT

SecureCRT 支持多种协议，比如 SSH2、SSH1、Telnet、Serial 等。可以用它来连接 Linux 服务器，作为一个远程控制台进行各类操作；也可以用它来连接串口，操作目标板。

SecureCRT 可以从网址 http://www.vandyke.com/products/securecrt/中下载，有 30 天的免费试用期。

安装、启动 SecureCRT 后，跳过初始设置界面。单击菜单“File”→“connect”，出现如图 4.10 所示的界面，单击“New Session”按钮，开始建立新的连接。

在随后出现的对话框中选择传输协议：SSH2 或 Serial，如图 4.11 所示。

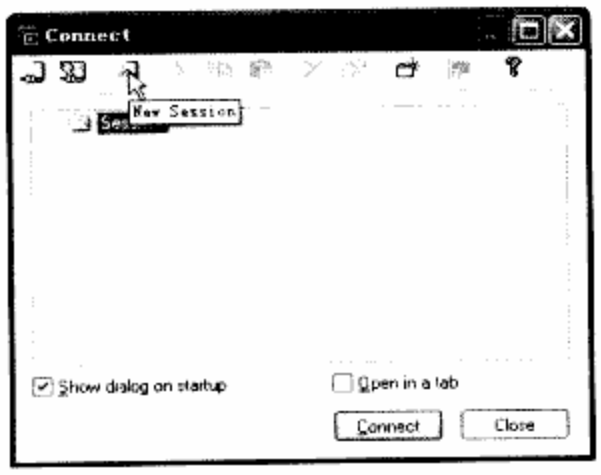


图 4.10　SecurtCRT 新建连接的界面

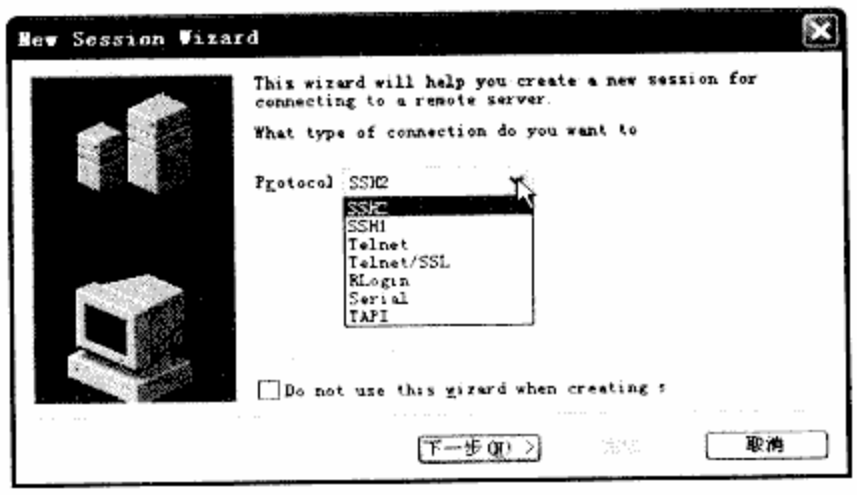


图 4.11　选择传输协议

在图 4.11 所示的对话框中单击“下一步”按钮，进行更详细的设置。对于 SSH2、Serial 协议，请分别参考图 4.12、图 4.13 进行设置。

图 4.12：输入服务器 IP（Hostname）、用户名（Username）。

图 4.13：选择串口 1（COM1），设置波特率（Baud rate）为 115200、数据位（Data bits）为 8、不使用校验位（Parity）、停止位（Stop bits）为 1，不使用流控（Flow Control）。

按照图 4.12、图 4.13 设置完成后，单击“下一步”按钮，在后续对话框中可以设置这个新建的连接的名字。

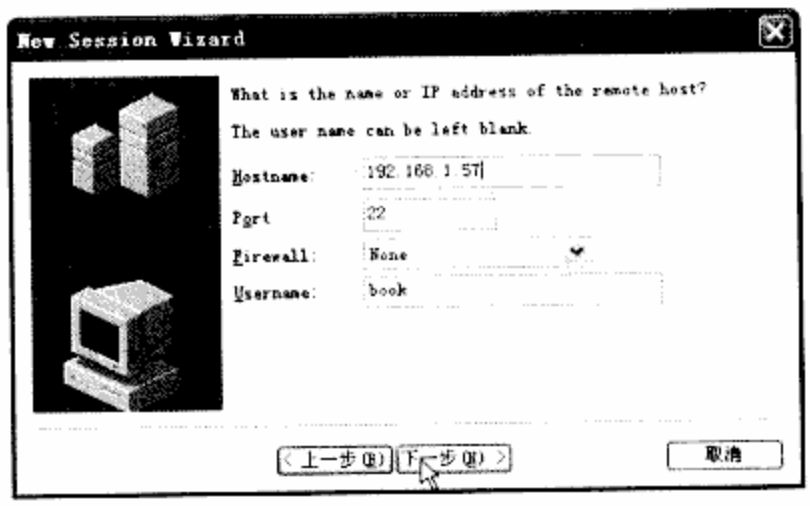


图 4.12　设置 SSH2 连接

当建立完新的连接后，可以看到如图 4.14 所示的对话框，也可以通过点击菜单“File”→“connect”启动这个对话框。在里面双击某个连接，即可启动它。

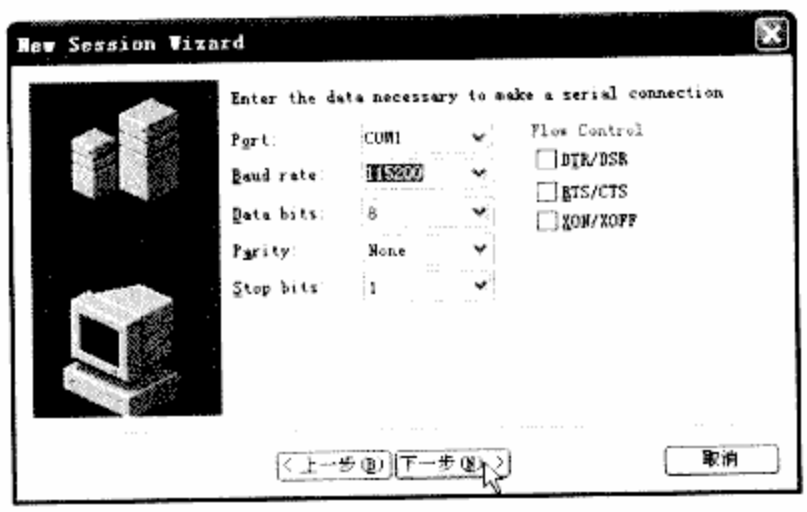


图 4.13　设置 Serial 连接

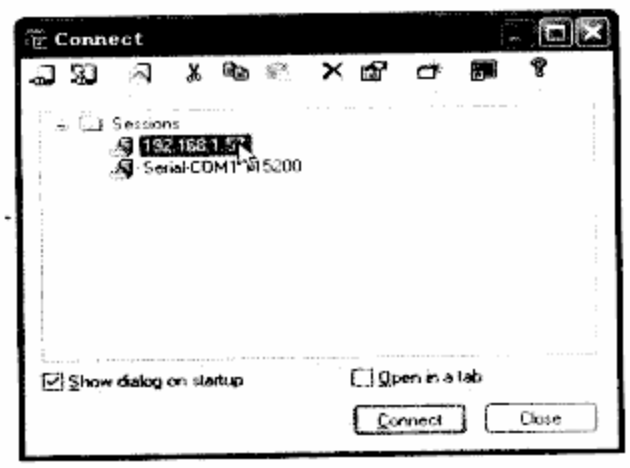


图 4.14　启动连接

4.1.4　TFTP 服务器软件 Tftpd32

Tftpd32 是一款轻便的 DHCP、TFTP、SNTP 和 Syslog 服务器软件，同时也是一款 TFTP 客户端软件。使用 U-Boot 时可以使用它的 TFTP 服务器功能下载软件到目标板中（也可以使用 Linux 中的 NFS 服务代替）。

Tftpd32 可以从网址 http://tftpd32.jounin.net/tftpd32_download.html 中下载。它可以直接运行，参考图 4.15 进行设置：选择服务器的目录（要传输的文件放在这个目录中）、选择 IP 地址（对于有多个 IP 的系统而言，要从中选择一个）。

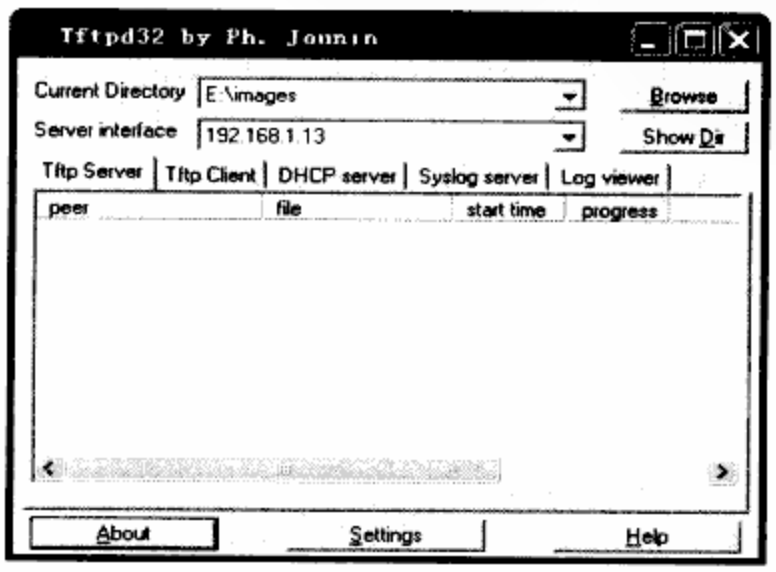


图 4.15 设置 Tftpd32 的 TFTP 服务

设置完毕后，以后 U-Boot 就可以通过 tftp 命令从 Windows 系统中获取文件了。

4.2 Linux 环境下的工具、命令介绍

如果读者直接在 Linux 环境下工作，可以使用本节介绍的几个工具、命令，它们与 4.1 节中介绍的工具功能相似，它们是免费的。

4.2.1 代码阅读、编辑工具 KScope

KScope 的作用与 Source Insight 几乎一样，它也是一款源代码阅读、编辑工具。KScope 使用 Cscope 作为源代码的分析引擎，可以为编码人员提供一些有价值的信息，特别适用于使用 C 语言编写的大型项目。

下面依次介绍 KScope 的安装、使用方法。

1. 安装 KScope

确保 Linux 能连上网络，然后使用以下命令进行安装，会得到一个 kscope 命令，并且在 Linux 桌面菜单“Applications”→“Programming”下生成了一个启动项“KScope”：

```
$ sudo apt-get install kscope
```

要启动 KScope，可以在控制台中运行 kscope 命令，或者单击菜单“Applications”→“Programming”→“KScope”。

2. 建立 KScope 工程

建立 KScope 工程的步骤与建立 Source Insight 工程的步骤相似，也分为这几步：设置工程名，指定工程数据的存放位置，设置支持的文件类型，指定源码的位置，添加、移除源文件，建立数据库。

以内核源码为例，假设内核源码位置为/work/system/linux-2.6.22.6，将要建立的KScope 工程命名为 linux-2.6.22.6，在 / work/kscope_projects/ linux-2.6.22.6 目录下存放工程数据。

先建立 / work/kscope_projects/linux-2.6.22.6 目录，在控制台中执行以下命令：

```
$ mkdir -p/ work/kscope_projects/ linux-2.6.22.6
```

然后启动 KScope 后，单击菜单“Project”→“Create Project”，弹出如图 4.16 所示的对话框。先在“Detail”选项页中仿照图 4.16 分别填入工程名、工程数据的存放位置、源码的位置；然后在“File Types”选项页中仿照图 4.17 设置支持的文件类型为：*.c、*.h 和*.S；最后单击“Create”按钮得到类似图 4.18 的对话框。

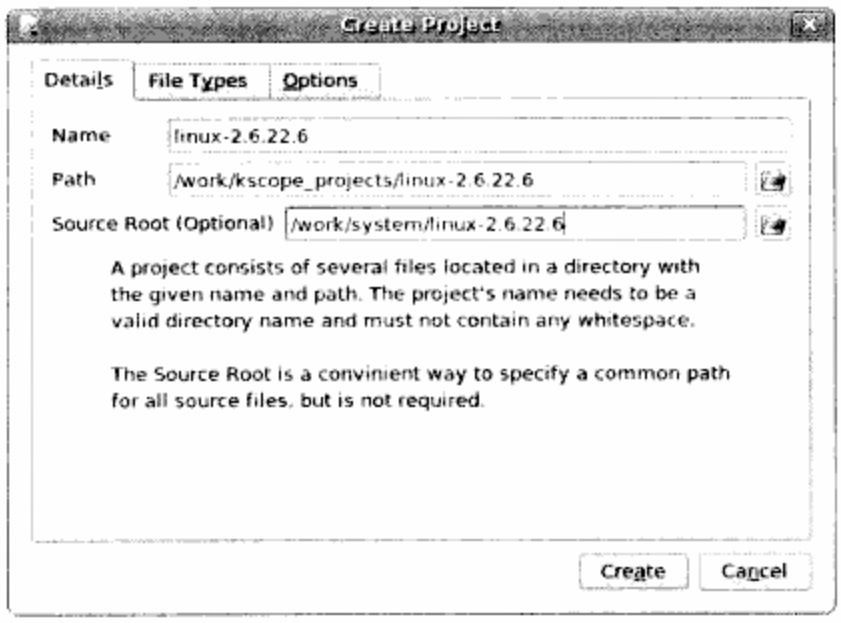


图 4.16 新建一个 KScope 工程

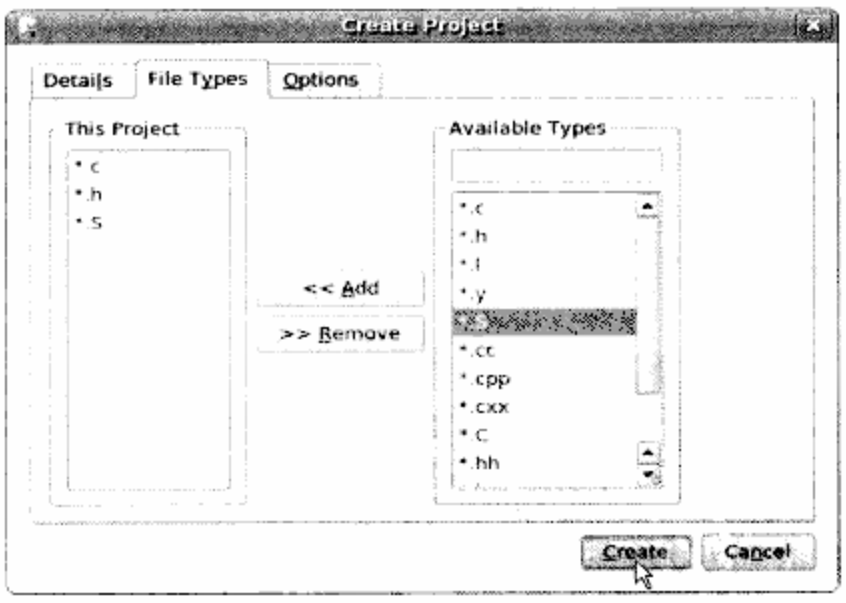


图 4.17 设置 KScope 工程支持的文件类型

类似图 4.18 的对话框，也可以通过菜单“Project”→“Add/Remove Files”来启动，在这个对话框中进行源文件的添加、移除。

图 4.18 中，“Files”、“Directory”、“Tree”按钮分别表示添加、移除的操作以文件、目录（表示目录下的文件，不包括它的子目录）、整个目录树为单位。移除操作中的“Dircectory”、“Tree”按钮还没有实现，可以在左边的文件框中选择要去除的文件，然后单击“Selected”按钮。为了方便，不妨在建立工程之前先删除不需要的目录、文件，以本书为例，这些不需

要目录如下。

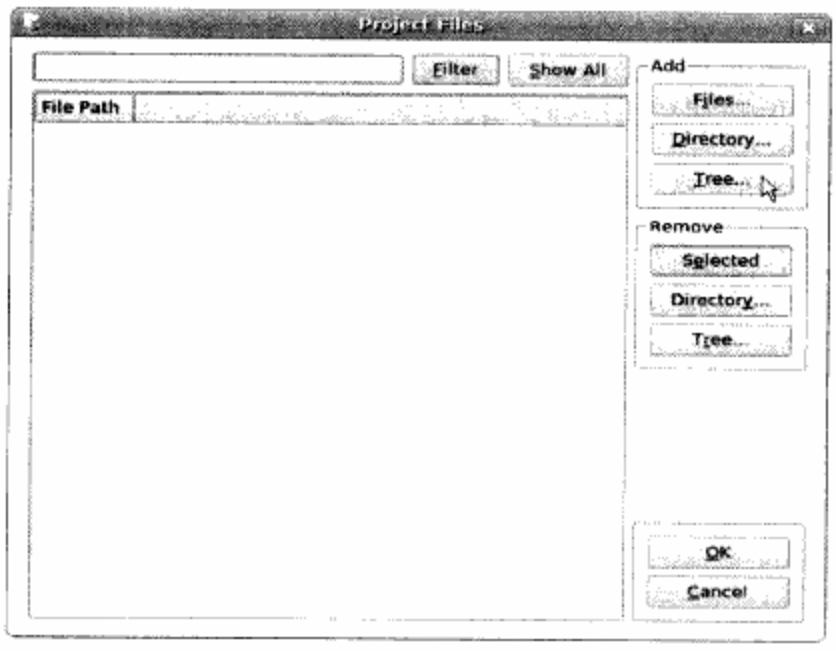


图 4.18 添加/移除源文件

① arch 目录下除 arm 外的所有子目录。

② arch/arm 目录下以“mach-”开头的目录（mach-s3c2410、mach-s3c2440 除外）。

③ arch/arm 目录下以“plat-”开头的目录（plat-s3c24xx 除外）。

④ include 目录下以“asm-”开头的目录（asm-arm、asm-generic 除外）。

⑤ include/asm-arm 目录下以“arch-”开头的目录（arch-s3c2410 除外）。

在图 4.18 中，单击“Add”框中的“Tree”按钮，弹出如图 4.19 所示的对话框。在“Folders”框中选择内核的根目录，然后单击“OK”按钮开始添加源文件。

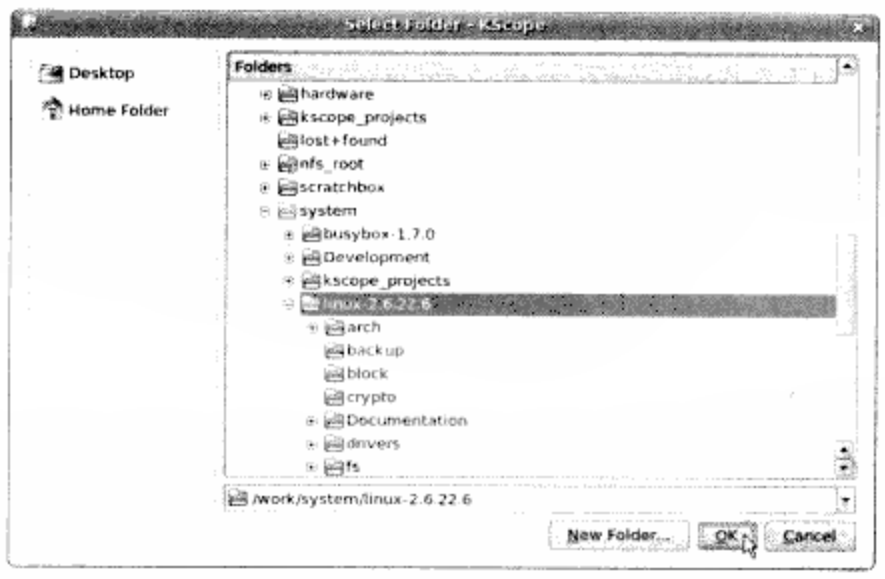


图 4.19 添加源文件目录树

源文件添加完毕后，单击图 4.18 中的“OK”按钮，KScope 即会自动生成数据库。

至此，KScope 工程建立完毕。

3. KScope 工程使用示例

在 KScope 右边的文件列表中选择打开 s3c2410fb.c 文件，可以得到如图 4.20 所示的界面，它的中间是主窗口，可以在里面阅读、编辑代码；左边是“Tag List”（Tag 列表），可以从中快速地找到当前文件中的变量、函数，宏定义等；下边是“Query Window”（查询结果窗口），

在主窗口中将光标放在某个变量、函数、宏上面，然后使用右键单击选择某些操作或者按住某些快捷键，会在这个窗口中显示这些操作的结果。

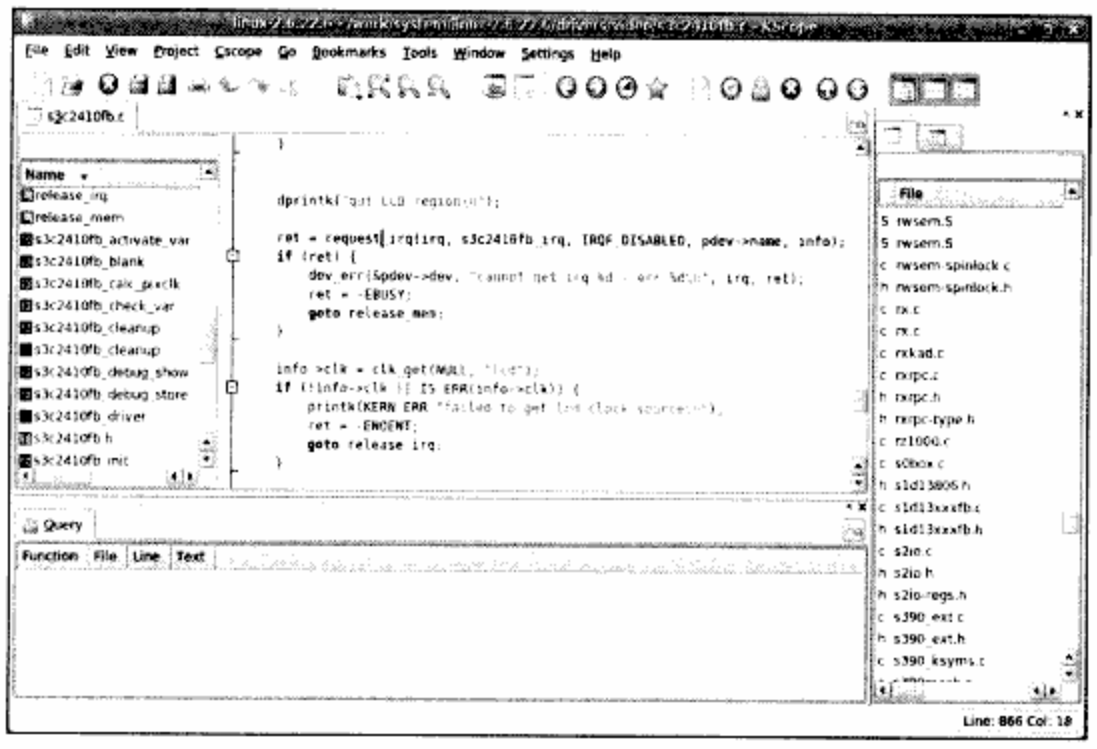


图 4.20　KScope 的使用界面

下面介绍一些简单的操作示例。

在主窗口中，单击某个函数，比如单击 s3c2410fb.c 中的“request_irq”字样，然后单击鼠标右键，将弹出一个菜单，选择其中的“Cscope”→“Quick Definition”即可快速找到它的定义。将光标移到“request_irq”字样上，然后按下快捷键“Ctrl+]”也可以达到同样的效果。

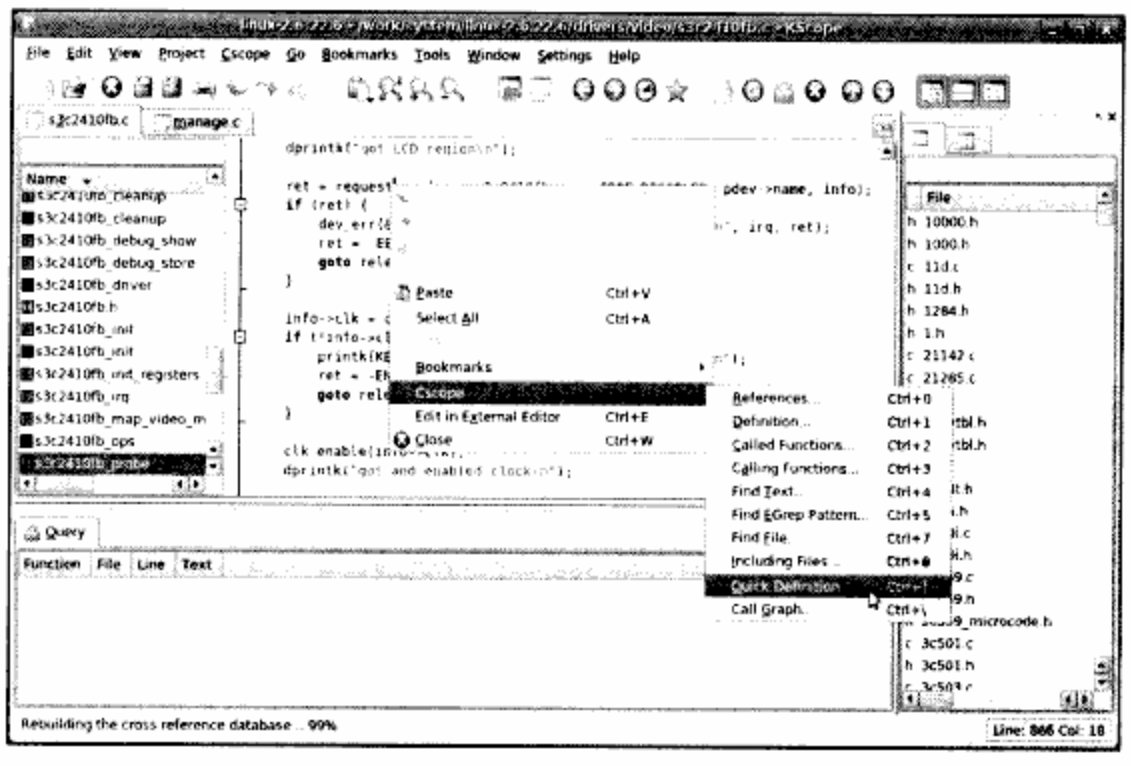


图 4.21　在 KScope 中查找函数定义

同时按住“Alt”、左箭头键可以令主窗口退回上一画面，同时按住“Alt”、右箭头键可以令主窗口前进到前一个画面。

KScope 还有很多使用技巧，上面只介绍了几种常用的技巧，读者在使用过程中可以通过各个菜单了解更多。

4.2.2 远程登录工具 C-kermit

C-Kermit 是一款集成了网络通信、串口通信的工具，它有如下多种功能（本书中，只用到它的串口通信功能）。

- 支持 Kermit 文件传输协议。
- 自定义了一种脚本语言，它强大而易于使用，可用于自动化工作。
- 无论是网络通信、还是串口通信，操作是一致的，并支持多种硬件、软件平台。
- 有安全认证、加密功能。
- 内建 FTP、HTTP 客户端功能及 SSH 接口。
- 支持字符集转换。

下面介绍在 Linux 下安装、使用 C-kermit 的方法。

确保 Linux 能连上网络，然后使用以下命令进行安装，会得到一个 kermit 命令：

```
$ sudo apt-get install ckermit
```

使用 kermit 之前，先在/homt/book（假设用户名为 book）目录下创建一个名为.kermrc 的配置文件，内容如下：

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

然后，运行“$ sudo kermit -c”命令即可启动串口；要想关闭串口，先同时按住“Ctrl”和“\”键，然后松开再按“C”键，最后输入“exit”并回车。

在 Linux 中，可以使用 kermit 连接串口以操作目标板。

4.2.3 编辑命令 vi

vi 命令是字符终端下的一个文本编辑工具。对文本进行少量修改时使用 vi 命令很方便，特别是在使用 SecureCRT 等工具远程登录 Linux 时。

vi 可以执行输出、删除、查找、替换、块操作等众多文本操作，它没有菜单，只有命令，且命令繁多。

在控制台中输入“vi”或“vi filename”就可以启动 vi，后者将打开或新建文件。它有 3 种基本工作模式：命令行模式、文本输入模式和末行模式。

1. 命令行模式

vi 一被启动，它就处于命令行模式；另外，任何时候、任何模式下，只要按“Esc”键，即可使 vi 进入命令行模式。在“命令行模式”下，可以直接使用某些按键完成相应操作。

常用的命令如表 4.1 所示。

表 4.1　　vi 中常用的命令

	命令（按键）	作　用	命令（按键）	作　用
光标移动命令	Ctrl+f	向文件尾翻一屏	Ctrl+b	向文件首翻一屏
	n+	光标下移 n 行（n 为数字）	n-	光标上移 n 行(n 为数字)
	0（数字零）	光标移至当前行首	$	光标移至当前行尾
	nG	光标移至第 n 行的行首（n 为数字）	:n	光标移至第 n 行的行首（n 为数字）
文本插入命令	i	在光标前开始插入文本	a	在光标后开始插入文本
	o	在当前行之下新开一行	O	在当前行之上新开一行
文本删除命令	d0	删至行首	d$或者 D	删至行尾
	x	删除光标后的一个字符	X	删除光标前的一个字符
	ndd	删除当前行及其后 n-1 行		
搜索及替换命令	/pattern	从光标开始处向文件尾搜索 pattern	?pattern	从光标开始处向文件首搜索 pattern
	n	在同一方向重复上一次搜索命令	N	在反方向上重复上一次搜索命令
	:s/p1/p2/g	将当前行中所有 p1 均用 p2 替代	:n1,n2s/p1/p2/g	将第 n1 至 n2 行中所有 p1 均用 p2 替代
	:g/p1/s//p2/g	将文件中所有 p1 均用 p2 替换		
退出/保存命令	:w	保存文件	:wq	保存文件并退出 vi
	:q	退出 vi	:q!	退出 vi，但是不保存文件

注：（1）“搜索及替换命令”中的“pattern”、“p1”、“p2”表示一个正则表达式，可以用来匹配某些字符串，比如“[0-9][0-9]”表示两位数。通常直接使用字符串，比如使用命令“/lib”在文件中查找“lib”字样。

（2）“:”开头的命令是“末行模式”中的用法，这里是为了方便读者参考才放在一起。

2. 文本输入模式

在命令模式下输入表 4.1 中的文本插入命令时，就会进入文本输入模式。在该模式下，用户输入的任何字符都被 vi 当做文件内容保存起来，并在屏幕上显示。在文本输入过程中，按“Esc”键即可回到命令模式。

3. 末行模式

在 vi 中，命令通常只包含几个按键，如表 4.1 所示；要想输入更长的命令，要进入“末

行模式”。在命令模式下，用户按“:”键即可进入末行模式，此时 vi 会在显示窗口的最后一行显示一个“:”作为末行模式的提示符，等待用户输入命令。输入完成后回车，命令即会执行，然后 vi 自动回到命令模式。

末行模式下常用的命令如表 4.1 所示。

4.2.4 查找命令 grep、find 命令

在 Linux 下，常用 grep 命令列出含有某个字符串的文件，常用 find 命令查找匹配给定文件名的文件。

1. grep 命令

grep 命令的用法为：

```
grep [options] PATTERN [FILE...]
```

以几个例子介绍它的常用格式。

（1）在内核目录下查找包含“request_irq”字样的文件。

```
$ cd /work/system/linux-2.6.22.6/
// *表示查找当前目录下的所有文件、目录，-R 表示递归查找子目录
$ grep "request_irq" * -R
```

（2）在内核的 kernel 目录下查找包含“request_irq”字样的文件。

```
$ cd /work/system/linux-2.6.22.6/
// kernel 表示在当前目录的 kernel 子目录下查找，-R 表示递归查找它的所有子目录
$ grep "request_irq" kernel -R
```

2. find 命令

find 命令的用法为：

```
find [-H] [-L] [-P] [path...] [expression]
```

以几个例子介绍它的常用格式。

（1）在内核目录下查找文件名中包含“fb”字样的文件。

```
$ cd /work/system/linux-2.6.22.6/
$ find -name "*fb*"
```

（2）在内核的 drivers/net 目录下查找文件名中包含“fb”字样的文件。

```
$ cd /work/system/linux-2.6.22.6/
$ find drivers/net -name "*fb*"          // “drivers/net”必须是 find 命令的第一个参数
```

依照上面介绍的 grep、find 命令的使用例子，基本可以满足在 Linux 下对代码、文件的查找工作。

4.2.5 在线手册查看命令 man

Linux 中包含了种类繁多的在线手册，从各种命令、各种函数的使用，到一些配置文件的设置。可以使用 man 命令查看这些手册，比如执行“man grep”命令即可看到 grep 命令的使用方法。

man 命令的基本用法为：

```
man [section] name
```

其中的“section”被称为区号，当直接使用“man name”命令没有查到需要的手册时，可以指定区号。比如想查看“open”函数的用法，使用“man open”命令得到的却是一个名为 openvt 的程序的用法，这时可以使用“man 2 open”命令，表示要查看第 2 区（它表示系统调用）中的手册。

Linux 在线手册按照区号进行分类，如表 4.2 所示。

表 4.2　　Linux 在线手册的区号及类别

区　号	类　别
1	命令，比如 ls、grep、find 等
2	系统调用，比如 open、read、socket 等
3	库调用，比如 fopen、fread 等
4	特殊文件，比如/dev/目录下的文件等
5	文件格式和惯例，比如/etc/passwd 等
6	游戏
7	其他
8	系统管理命令，类似 mount 等只有系统管理员才能执行的命令
9	内核例程（这个区号基本没被使用）

最后介绍使用 man 命令的阅读技巧，即启动 man 命令后，可以通过一些热键进行翻页等操作，如表 4.3 所示。

表 4.3　　man 命令的热键

热　键	作　用
h	显示帮助信息
j	前进一行
k	后退一行
空格或 f	向前翻页
b	向后翻页
g	跳转到手册的第一行
G	跳转到手册的最后一行
?string	向后搜索字符串 string

续表

热 键	作 用
/string	向前搜索字符串 string
r	刷屏
q	退出

4.2.6 其他命令：tar、diff、patch

1. tar 命令

tar 命令具有打包、解包、压缩、解压缩 4 种功能，在本书中使用的频率很高。它常用的压缩、解压缩方式有两种：gzip、bzip2。一般而言，以“.gz”、“z”结尾的文件是用 gzip 方式进行压缩的，以“.bz2”结尾的文件是用 bzip2 方式进行压缩的，后缀名中有“tar”字样时表示这是一个文件包。

tar 命令有 5 个常用的选项。

① “c”：表示创建，用来生成文件包。

② “x”：表示提取，从文件包中提取文件。

③ “z”：使用 gzip 方式进行处理，它与“c”结合就表示压缩，与“x”结合就表示解压缩。

④ “j”：使用 bzip2 方式进行处理，它与“c”结合就表示压缩，与“x”结合就表示解压缩。

⑤ “f”：表示文件，后面接着一个文件名。

以例子说明 tar 命令的使用方法。

① 将某个目录 dirA 制作为压缩包。

```
$ tar czf dirA.tar.gz dirA      // 将目录 dirA 压缩为文件包 dirA.tar.gz，以 gzip 方式进行压缩
$ tar cjf dirA.tar.bz2 dirA     // 将目录 dirA 压缩为文件包 dirA.tar.bz2，以 bzip2 方式进行压缩
```

② 将某个压缩包文件 dirA.tar.gz 解开。

```
$ tar xzf dirA.tar.gz    // 在当前目录下解开 dirA.tar.gz，先使用 gzip 方式解压缩，然后解包
$ tar xjf dirA.tar.bz2   // 在当前目录下解开 dirA.tar.bzip2，先使用 bzip2 方式解压缩，然后解包
$ tar xzf dirA.tar.gz  -C <dir>  // 将 dirA.tar.gz 解开到<dir>目录下
$ tar xjf dirA.tar.bz2 -C <dir>  // 将 dirA.tar.bz2 解开到<dir>目录下
```

2. diff、patch 命令

diff 命令常用来比较文件、目录，也可以用来制作补丁文件。所谓“补丁文件”就是“修改后的文件”与“原始文件”的差别。

常用的选项如下。

① “-u”：表示在比较结果中输出上下文中一些相同的行，这有利于人工定位。
② “-r”：表示递归比较各个子目录下的文件。
③ “-N”：将不存在的文件当作空文件。
④ “-w”：忽略对空格的比较。
⑤ “-B”：忽略对空行的比较。

例如：假设 linux-2.6.22.6 目录中是原始的内核，linux-2.6.22.6_ok 目录中是修改过的内核，可以使用以下命令制作补丁文件 linux-2.6.22.6_ok.diff（原始目录在前，修改过的目录在后）。

```
$ diff -urNwB linux-2.6.22.6 linux-2.6.22.6_ok > linux-2.6.22.6_ok.diff
```

由于 linux-2.6.22.6 是标准的代码，可以从网上自由下载，要发布 linux-2.6.22.6_ok 中所做的修改时，只需要提供补丁文件 linux-2.6.22.6_ok.diff（它通常是很小的）。

patch 命令被用来打补丁——就是依据补丁文件来修改原始文件。比如对于上面的例子，可以使用以下命令将补丁文件 linux-2.6.22.6_ok.diff 应用到原始目录 linux-2.6.22.6 上去。假设 linux-2.6.22.6_ok.diff 和 linux-2.6.22.6 位于同一个目录下。

```
$ cd linux-2.6.22.6
$ patch -p1 < ../ linux-2.6.22.6_ok.diff
```

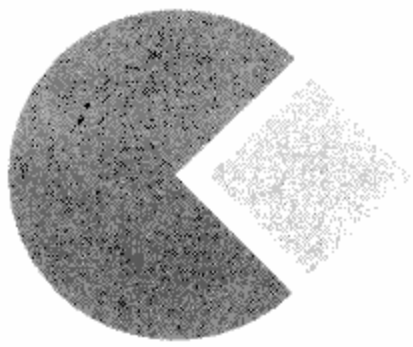
patch 命令中最重要的选项是“-pn”：补丁文件中指明了要修改的文件的路径，“-pn”表示忽略路径中第 n 个斜线之前的目录。假设 linux-2.6.22.6_ok.diff 中有如下几行：

```
diff -urNwB linux-2.6.22.6/A/B/C.h linux-2.6.22.6_ok/A/B/C.h
--- linux-2.6.22.6/A/B/C.h 2007-08-31 02:21:01.000000000 -0400
+++ linux-2.6.22.6_ok/A/B/C.h  2007-09-20 18:11:46.000000000 -0400
……
```

使用上述命令打补丁时，patch 命令根据“linux-2.6.22.6/A/B/C.h”寻找源文件，“-p1”表示忽略第 1 个斜线之前的目录，所以要修改的源文件是当前目录下的：A/B/C.h。

第 2 篇　ARM9 嵌入式系统基础实例篇

- GPIO 接口
- 存储控制器
- 内存管理单元 MMU
- NAND Flash 控制器
- 中断体系结构
- 系统时钟和定时器
- 通用异步收发器 UART
- I^2C 接口
- LCD 控制器
- ADC 和触摸屏接口



第 5 章　GPIO 接口

本章目标

- 掌握嵌入式开发的步骤：编程、编译、烧写程序、运行
- 通过 GPIO 的操作了解软件如何控制硬件

5.1　GPIO 硬件介绍

GPIO（General Purpose I/O Ports）意思为通用输入/输出端口，通俗地说，就是一些引脚，可以通过它们输出高低电平或者通过它们读入引脚的状态——是高电平还是低电平。

S3C2410 有 117 个 I/O 端口，共分为 A～H 共 8 组：GPA、GPB、...、GPH。S3C2440 有 130 个 I/O 端口，分为 A～J 共 9 组：GPA、GPB、...、GPJ。可以通过设置寄存器来确定某个引脚用于输入、输出还是其他特殊功能。比如可以设置 GPH6 作为一般的输入、输出引脚，或者用于串口。

GPIO 的操作是所有硬件操作的基础，由此扩展开来可以了解所有硬件的操作，这是底层开发人员必须掌握的。

5.1.1　通过寄存器来操作 GPIO 引脚

既然一个引脚可以用于输入、输出或其他特殊功能，那么一定有寄存器用来选择这些功能；对于输入，一定可以通过读取某个寄存器来确定引脚的电平是高还是低；对于输出，一定可以通过写入某个寄存器来让这个引脚输出高电平或低电平；对于其他特殊功能，则有另外的寄存器来控制它。

对于这几组 GPIO 引脚，它们的寄存器是相似的：GPxCON 用于选择引脚功能，GPxDAT 用于读/写引脚数据；另外，GPxUP 用于确定是否使用内部上拉电阻。x 为 A、B、...、H/J，没有 GPAUP 寄存器。

1. GPxCON 寄存器

从寄存器的名字即可看出，它用于配置（Configure）——选择引脚的功能。

PORT A 与 PORT B~PORT H/J 在功能选择方面有所不同，GPACON 中每一位对应一根引

脚（共 23 根引脚）。当某位被设为 0 时，相应引脚为输出引脚，此时我们可以在 GPADAT 中相应位写入 0 或 1 让此引脚输出低电平或高电平；当某位被设为 1 时，相应引脚为地址线或用于地址控制，此时 GPADAT 无用。一般而言 GPACON 通常被设为全 1，以便访问外部存储器件。本章不使用 PORT A。

PORT B~PORT H/J 在寄存器操作方面完全相同。GPxCON 中每两位控制一根引脚：00 表示输入、01 表示输出、10 表示特殊功能、11 保留不用。

2. GPxDAT 寄存器

GPxDAT 用于读/写引脚：当引脚被设为输入时，读此寄存器可知相应引脚的电平状态是高还是低；当引脚被设为输出时，写此寄存器相应位可令此引脚输出高电平或低电平。

3. GPxUP 寄存器

GPxUP：某位为 1 时，相应引脚无内部上拉电阻；为 0 时，相应引脚使用内部上拉电阻。上拉电阻、下拉电阻如图 5.1 所示：

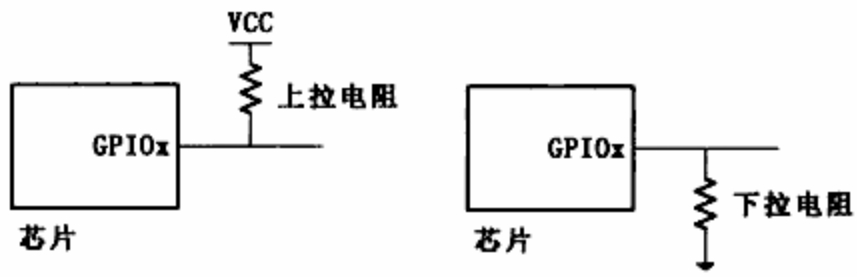


图 5.1 上拉电阻和下拉电阻

上拉电阻、下拉电阻的作用在于，当 GPIO 引脚处于第三态（既不是输出高电平，也不是输出低电平，而是呈高阻态，即相当于没接芯片）时，它的电平状态由上拉电阻、下拉电阻确定。

5.1.2 怎样使用软件来访问硬件

1. 访问单个引脚

单个引脚的操作无外乎 3 种：输出高低电平、检测引脚状态、中断。对某个引脚的操作一般通过读、写寄存器来完成。

比如对于如图 5.2 所示的电路，可以设置 GPBCON 寄存器将 GPB5、GPB6、GPB7 和 GPB8 设为输出功能，然后写 GPBDAT 寄存器的相应位使得这 4 个引脚输出高电平或低电平：输出高电平时，相应的 LED 熄灭；输出低电平时，相应的 LED 点亮。

还可以设置 GPFCON 寄存器将 GPF0、GPF2、GPG3 和 GPG11 设为输入功能，然后通过读出 GPFDAT/GPGDAT 寄存器并判断相应位是 0 还是 1 来确定各个按键是否被按下：某个按键被按下时，相应引脚电平为低，GPFDAT/GPGDAT 寄存器中相应位为 0；否则为 1。

那么，怎么访问这些寄存器呢？通过软件，读写它们的地址。比如，S3C2410 和 S3C2440 的 GPBCON、GPBDAT 寄存器地址都是 0x56000010、0x56000014，可以通过如下的指令让 GPB5 输出低电平，点亮 LED1。

```
#define GPBCON          (*(volatile unsigned long *)0x56000010)
#define GPBDAT          (*(volatile unsigned long *)0x56000014)
#define GPB5_out        (1<<(5*2))
GPBCON  = GPB5_out;     // GPB5 引脚设为输出
GPBDAT  &= ~(1<<5);     // GPB5 输出低电平
```

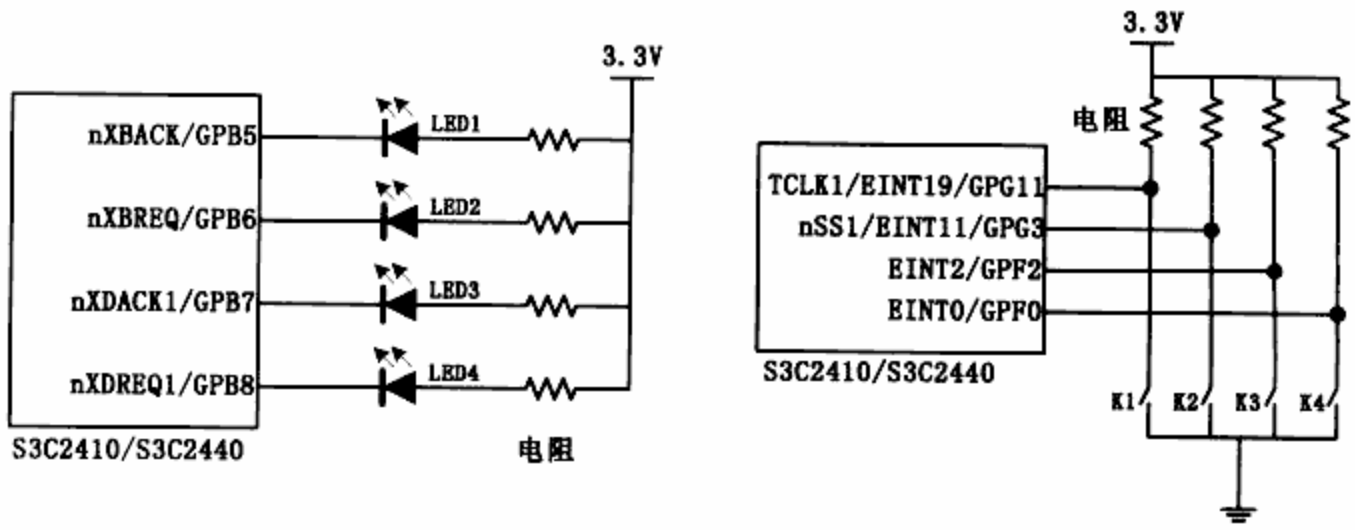


图 5.2 LED 与按键连线图

2. 以总线方式访问硬件

并非只能通过寄存器才能发出硬件信号，实际上，通过访问总线的方式控制硬件更常见。以 NOR Flash 的访问为例，如图 5.3 所示是 S3C2410/S3C2440 与 NOR Flash 的连线图。

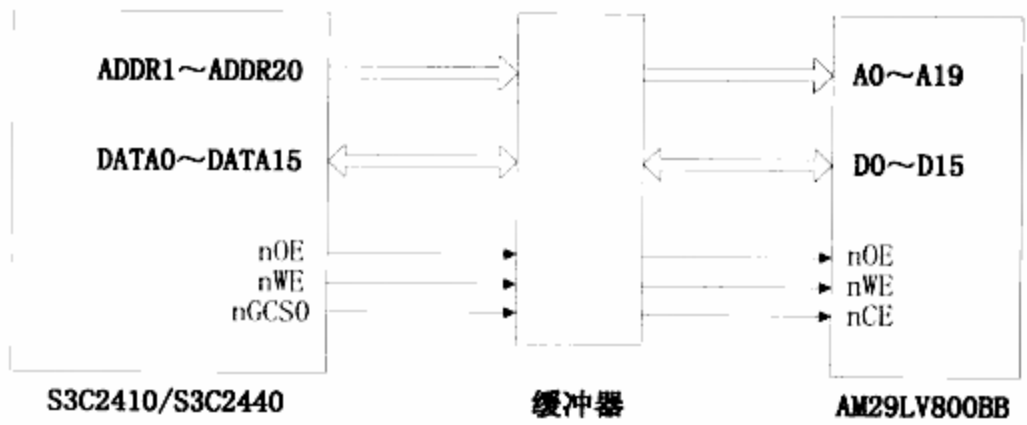


图 5.3 S3C2410/S3C2440 与 NOR Flash 的连线图

图中，缓冲器的作用为提高驱动能力、隔离前后级信号。NOR Flash AM29LV800BB 的片选信号使用 S3C2410/S3C2440 的 nGCS0 信号，当 CPU 发出的地址信号处于 0x00000000～0x07FFFFFF 之间时，nGCS0 信号有效（为低电平），于是 NOR Flash 被选中。这时，CPU 发出的地址信号传到 NOR Flash；进行写操作时，nWE 信号为低，数据信号从 CPU 发给 NOR Flash；进行读操作时，nWE 信号为高，数据信号从 NOR Flash 发给 CPU。如图 5.3 所示的硬件连线决定了读写操作都是以 16 位为单位的。

软件如何发起读写操作呢？下面通过几个实例的代码进行讲解。

【例 5.1】 地址对齐的 16 位读操作。

```
unsigned short * pwAddr = (unsigned short *)0x2;
unsigned short wVal;
wVal = *pwAddr;
```

上述代码就会向 NOR Flash 发起读操作：CPU 发出的读地址为 0x2，则地址总线 ADDR1～ADDR20、A0～A19 的信号都是 1、0、…、0（CPU 的 ADDR0 为 0，不过 ADDR0 没有接到 NOR Flash 上）。NOR Flash 接收到的地址就是 0x1，NOR Flash 在稍后的时间里将此址上的 16 位数据取出，并通过数据总线 D0～D15 发给 CPU。

【例 5.2】 地址不对齐的 16 位读操作。

```
unsigned short * pwAddr = (unsigned short *)0x1;
unsigned short wVal;
wVal = *pwAddr;
```

由于地址为 0x1，不是 2 对齐的，但是 BANK0 的位宽被设为 16，这将导致异常。我们可以设置异常处理函数来处理这种情况。在异常处理函数中，使用 0x0、0x2 发起两次读操作，然后将两个结果组合起来：使用地址 0x0 的两字节数据 D0、D1；再使用地址 0x2 读到 D2、D3；最后，D1、D2 组合成一个 16 位的数返回给 wVal。如果没有设置地址不对齐的异常处理函数，那么上述代码将出错。如果某个 BANK 被的位宽被设为 *n*，访问此 BANK 时，在总线上永远只会看到地址对齐的 *n* 位操作。

【例 5.3】 8 位读操作。

```
unsigned char * pucAddr = (unsigned char *)0x6;
unsigned char ucVal;
ucVal = *pucAddr;
```

CPU 首先使用地址 0x6 对 NOR Flash 发起 16 位的读操作，得到两字节的数据，假设为 D0、D1；然后将 D0 取出赋给变量 ucVal。在读操作期间，地址总线 ADDR1～ADDR20、A0～A19 的信号都是 1、1、…、0（CPU 的 ADDR0 为 0，不过 ADDR0 没有接到 NOR Flash 上）。CPU 会自动丢弃 D1。

【例 5.4】 32 位读操作。

```
unsigned int * pdwAddr = (unsigned int *)0x6;
unsigned int udwVal;
dwVal = *pdwAddr;
```

CPU 首先使用地址 0x6 对 NOR Flash 发起 16 位的读操作，得到两字节的数据，假设为 D0、D1；再使用地址 0x8 发起读操作，得到两字节的数据，假设为 D2、D3；最后将这 4 个数据组合后赋给变量 dwVal。

由于 NOR Flash 的特性，使得对 NOR Flash 的写操作比较复杂——比如要先发出特定的地址信号通知 NOR Flash 准备接收数据，然后才发出数据等。不过，其总线上的电信号与软件指令的关系与读操作类似，只是数据的传输方向相反，如例 5.5 所示。

【例 5.5】 16 位写操作。

```
unsigned short * pwAddr = (unsigned short *)0x6;
*pwAddr = 0x1234;
```

CPU 发起一次对 NOR Flash 的写操作，地址总线 ADDR1～ADDR20、A0～A19 的信号都

是1、1、…、0（CPU的ADDR0为0，不过ADDR0没有接到NOR Flash上）；数据总线DATA0～DATA15、D0～D15的信号为0、0、1、0、1、1、0、0、 0、1、0、0、1、0、0、0。

由此可见，CPU使用某个地址进行访问时，这个32位的地址值和ADDR0～ADDR31一一对应（也许CPU没有引出那么多地址信号，比如S3C2410/S3C2440只引出了ADDR0～ADDR26共27根地址线）。外接的设备可以以8位、16位、32位进行操作——这取决于硬件设计：如果以8位进行操作，则数值出现在数据信号DATA0～DATA7上；如果以16位进行操作，则数值出现在数据信号DATA0～DATA15上；如果以32位进行操作，则数值出现在数据信号DATA0～DATA31上。

5.2 GPIO操作实例：LED和按键

从本节开始，将涉及在开发板上运行程序了，下面用几个例子由简到繁地介绍。

首先约定：本文的所有程序均在hardware目录下的各级子目录中，生成的可执行文件名为相应目录名的加上后缀“.bin”，比如leds目录下的可执行文件为leds.bin。

5.2.1 硬件设计

LED和按键与处理器的电路连接如图5.2所示。

5.2.2 程序设计及代码详解

本小节有3个实例，通过读写GPIO寄存器来驱动LED、获得按键状态。先使用汇编程序编写一个简单的点亮LED的程序，然后使用C语言实现了更复杂的功能。

1．实例1：使用汇编代码点亮一个LED

源程序为/work/hardware/led_on/led_on.S。它只有7条指令，只是简单地点亮发光二极管LED1。本实例的目的是让读者对开发流程有个基本概念。

操作步骤：

（1）把PC并口和开发板JTAG接口连起来，确保插上NAND_BOOT跳线、上电。

（2）进入led_on目录后，执行如下命令生成可执行文件led_on.bin：

```
$ make
```

（3）执行如下命令将led_on.bin写入NAND Flash。

sjf2410.exe、sjf2440.exe是Windows烧写开发板的JTAG工具，嵌入式Linux下的工具名为Jflash-s3c2410、Jflash-s3c2440，它们的用法相似，直接执行这些命令就可以看到使用说明。这些工具位于/work/tools/jtag目录下，在Windows上使用前要先安装驱动程序，它在/work/tools/jtag/for_windows/jtag_driver目录下。

以Windows下的工具为例。

① 对于S3C2410，执行以下命令：

```
$ sjf2410.exe /f:led_on.bin /d=5
```

对于S3C2440，执行以下命令：

```
$ sjf2440.exe /f:led_on.bin /d=5
```

注意 在 Linux 下，Jflash-s3c2410、Jflash-s3c2440 还不支持对 NOR Flash 的操作。幸运的是：使用 JTAG 工具对 NOR Flash 进行烧写只需要进行一次——烧写 U-Boot 时，这时可以借助 Windows 下的工具。另外，使用 Jflash-s3c2410、Jflash-s3c2440 时，要在前面加上“sudo”。

当出现如图 5.4 所示的提示时，输入“0”选择 NAND Flash K9F1208，并回车。

```
[SJF Main Menu]
 0:K9S1208 prog     1:28F128J3A prog   2:AM29LV800 Prog   3:Memory Rd/Wr
 4:k9f2gxxu0m Program 5:Exit
Select the function to test:0
```

图 5.4 提示 1

② 当出现如图 5.5 所示的提示时，输入“0”表示烧写 NAND Flash K9F1208，并回车：

```
[K9S1208 NAND Flash JTAG Programmer]
K9S1208 is detected. ID=0xec76
 0:K9S1208 Program      1:K9S1208 Pr BlkPage   2:Exit
Select the function to test :0
```

图 5.5 提示 2

③ 当出现如图 5.6 所示的提示时，输入“0”表示从第 0 块开始烧写，并回车：

```
[SMC(K9S1208V0M) NAND Flash Writing Program]

Source size:0h~23h

Available target block number: 0~4095
Input target block number:0
```

图 5.6 提示 3

④ 当再次出现与步骤②相同的提示时，输入“2”并按“回车”键退出。

以后不再重复讲述如何烧写程序，直接运行 Jflash-s3c2410 或 Jflash-s3c2440 即可看到提示。

（4）按开发板上的复位键后可看见 LED1 被点亮了。

实例分为 4 个步骤：编写源程序、生成可执行程序、烧写程序、运行程序。

先看看源程序 led_on.S：

```
01 .text
02 .global _start
03 _start:
04          LDR     R0,=0x56000010      @ R0 设为 GPBCON 寄存器，此寄存器
05                                      @ 用于选择端口 B 各引脚的功能
06                                      @ 是输出、是输入、还是其他
07          MOV     R1,# 0x00000400
08          STR     R1,[R0]             @ 设置 GPB5 为输出口，位[10:9]=0b01
09          LDR     R0,=0x56000014      @ R0 设为 GPBDAT 寄存器，此寄存器
10                                      @ 用于读/写端口 B 各引脚的数据
11          MOV     R1,#0x00000000      @ 此值改为 0x00000020，
12                                      @ 可让 LED1 熄灭
13          STR     R1,[R0]             @ GPB5 输出 0，LED1 点亮
```

```
14 MAIN_LOOP:
15          B       MAIN_LOOP
```

程序很简单，第4、7、8行3条指令用于将LED1对应的引脚GPB5设成输出引脚；第9、11、13行这3条指令让这个引脚输出0；第15行的指令是个死循环。

操作步骤（2）中，指令make的作用就是编译led_on.S源程序。Makefile的内容如下：

```
01 led_on.bin:led_on.S
02          arm-linux-gcc -g -c -o led_on.o led_on.S
03          arm-linux-ld -Ttext 0x0000000 -g led_on.o -o led_on_elf
04          arm-linux-objcopy -O binary -S led_on_elf led_on.bin
05 clean:
06          rm -f   led_on.bin led_on_elf *.o
```

make指令比较第1行中文件led_on.bin和文件led_on.S的时间，如果led_on.S的时间比led_on.bin的时间新（led_on.bin未生成时，此条件默认成立），则执行第2、3、4行的命令重新生成led_on.bin。也可以不用指令make，而直接一条一条地执行2、3、4行的指令，但是这样效率比较低。

第2行的指令是编译，第3行是连接，第4行是把ELF格式的可执行文件led_on_elf转换成二进制格式文件led_on.bin。

执行“make clean”时强制执行第6行的删除命令。

注意 Makefile文件中相应的命令行前一定要有一个制表符（TAB）。

汇编语言可读性差，实例2用C语言来实现了同样的功能，以后的实例也尽量用C语言实现。

2. 实例2：使用C语言代码点亮一个LED

源程序位于/work/hardware/led_on_c目录下。

C语言程序执行的第一条指令，并不在main函数中。生成一个C程序的可执行文件时，编译器通常会在我们的代码中加上几个被称为启动文件的代码——crt1.o、crti.o、crtend.o、crtn.o等，它们是标准库文件。这些代码设置C程序的堆栈等，然后调用main函数。它们依赖于操作系统，在裸板上这些代码无法执行，所以需要自己写一个。

这段代码很简单，只有6条指令。自己编写的crt0.S文件内容如下：

```
01 @*****************************************************************
02 @ File: crt0.S
03 @ 功能: 通过它转入C程序
04 @*****************************************************************
05
06 .text
07 .global _start
08 _start:
```

```
09              ldr   r0, =0x56000010    @ WATCHDOG 寄存器地址
10              mov   r1, #0x0
11              str   r1, [r0]           @ 写入 0，禁止 WATCHDOG，否则 CPU 会不断重启
12
13              ldr    sp, =1024*4       @ 设置堆栈，注意:不能大于 4KB，因为现在可用的内存只有 4KB
14                                       @ NAND Flash 中的代码在复位后会移到内部 ram(只有 4KB)
15              bl     main              @ 调用 C 程序中的 main 函数
16 halt_loop:
17              b      halt_loop
```

它在第 13 行设置好栈指针后，就可以通过第 15 行调用 C 函数 main 了。C 函数执行前，必须设置栈。

现在，可以很容易写出控制 LED 的程序了。main 函数在 led_on_c.c 文件中，代码如下：

```
01 #define GPBCON      (*(volatile unsigned long *)0x56000010)
02 #define GPBDAT      (*(volatile unsigned long *)0x56000014)
03
04 int main()
05 {
06     GPBCON = 0x00000400;    // 设置 GPB5 为输出端口，位[10:9]=0b01
07     GPBDAT = 0x00000000;    // GPB5 输出 0，LED1 点亮
08
09     return 0;
10 }
```

最后来看看 Makefile:

```
01 led_on_c.bin : crt0.S  led_on_c.c
02     arm-linux-gcc -g -c -o crt0.o crt0.S
03     arm-linux-gcc -g -c -o led_on_c.o led_on_c.c
04     arm-linux-ld -Ttext 0x0000000 -g  crt0.o led_on_c.o -o led_on_c_elf
05     arm-linux-objcopy -O binary -S led_on_c_elf led_on_c.bin
06     arm-linux-objdump -D -m arm  led_on_c_elf > led_on_c.dis
07 clean:
08     rm -f led_on_c.dis led_on_c.bin led_on_c_elf *.o
```

第 2、3 行分别编译源程序 crt0.S、led_on_c.c（还没有连接）。

第 4 行将编译得到的结果连接起来。

第 5 行把连接得到的 ELF 格式可执行文件 led_on_c_elf 转换成二进制格式文件 led_on_ c.bin。

第 6 行将结果转换为汇编码以供查看。

操作步骤如下。

（1）进入 led_on_c 目录后，执行如下命令生成可执行文件 led_on_c.bin：

```
$ make
```

（2）如同上面烧写 led_on.bin 一样，使用 JTAG 工具将 led_on_c.bin 写入 NAND Flash。

（3）按开发板上复位键后可看见 LED1 被点亮了。

目录 leds 中的程序是使用 4 个 LED 从 0～15 轮流计数，操作步骤如下。

（1）进入 leds 目录后执行 make 命令。

（2）烧写程序到 NAND Flash：

```
$ sjf2410.exe /f:leds.bin /d=5
或
$ sjf2440.exe /f:leds.bin /d=5
```

（3）按开发板上复位键运行。

另外，如果有兴趣，可以使用如下命令看看二进制可执行文件的汇编码：

```
$ arm-linux-objdump -D -b binary -m arm  xxxxx(二进制可执行文件名)
```

3. 实例 3：使用按键来控制 LED

目录/work/hardware/key_led 中的程序功能为：当 K1～K4 中某个按键被按下时，点亮 LED1～LED4 中相应的 LED。

key_led.c 代码如下：

```
01 #define GPBCON      (*(volatile unsigned long *)0x56000010)
02 #define GPBDAT      (*(volatile unsigned long *)0x56000014)
03
04 #define GPFCON      (*(volatile unsigned long *)0x56000050)
05 #define GPFDAT      (*(volatile unsigned long *)0x56000054)
06
07 #define GPGCON      (*(volatile unsigned long *)0x56000060)
08 #define GPGDAT      (*(volatile unsigned long *)0x56000064)
09
10 /*
11  * LED1-4 对应 GPB5、GPB6、GPB7、GPB8
12  */
13 #define GPB5_out        (1<<(5*2))
14 #define GPB6_out        (1<<(6*2))
15 #define GPB7_out        (1<<(7*2))
16 #define GPB8_out        (1<<(8*2))
17
18 /*
```

```
19  * K1-K4 对应 GPG11、GPG3、GPF2、GPF0
20  */
21 #define GPG11_in      ~(3<<(11*2))
22 #define GPG3_in       ~(3<<(3*2))
23 #define GPF2_in       ~(3<<(2*2))
24 #define GPF0_in       ~(3<<(0*2))
25
26 int main()
27 {
28         unsigned long dwDat;
29         // LED~1LED4 对应的 4 根引脚设为输出
30         GPBCON = GPB5_out | GPB6_out | GPB7_out | GPB8_out ;
31
32         // K1~K2 对应的 2 根引脚设为输入
33         GPGCON = GPG11_in & GPG3_in ;
34
35         // K3~K4 对应的 2 根引脚设为输入
36         GPFCON = GPF2_in & GPF0_in ;
37
38         while(1){
39             //若 Kn 为 0(表示按下)，则令 LEDn 为 0(表示点亮)
40             dwDat = GPGDAT;              // 读取 GPG 管脚电平状态
41
42             if (dwDat & (1<<11))         // K1 没有按下
43                 GPBDAT |= (1<<5);        // LED1 熄灭
44             else
45                 GPBDAT &= ~(1<<5);       // LED1 点亮
46
47             if (dwDat & (1<<3))          // K2 没有按下
48                 GPBDAT |= (1<<6);        // LED2 熄灭
49             else
50                 GPBDAT &= ~(1<<6);       // LED2 点亮
51
52             dwDat = GPFDAT;              // 读取 GPF 管脚电平状态
53
54             if (dwDat & (1<<2))          // K3 没有按下
55                 GPBDAT |= (1<<7);        // LED3 熄灭
56             else
57                 GPBDAT &= ~(1<<7);       // LED3 点亮
```

```
58
59          if (dwDat & (1<<0))          // K4没有按下
60              GPBDAT |= (1<<8);        // LED4熄灭
61          else
62              GPBDAT &= ~(1<<8);       // LED4点亮
63      }
64
65      return 0;
66  }
```

第30行将LED1～4对应的引脚GPB5～GPB8设为输出引脚。

第33行将K1、K2对应的引脚GPG11、GPG3设为输入引脚。

第36行将K3、K4对应的引脚GPF2、GPF0设为输入引脚。

下面就是个无穷循环，读取GPGDAT、GPFDAT寄存器，从中判断K1、K2、K3、K4是否按下，若按下则点亮相应的LED，否则熄灭相应的LED。

操作步骤如下。

（1）进入目录key_led，运行make命令生成key_led.bin。

（2）烧写程序到NAND Flash：

```
$ sjf2410.exe /f:key_led.bin /d=5
```

或

```
$ sjf2440.exe /f:key_led.bin /d=5
```

（3）按下复位键运行程序后，按下、松开按钮K1～K4可以看见LED1～LED4相应亮灭。

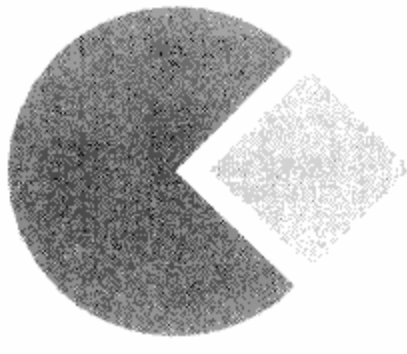
5.2.3 实例测试

本章所有程序的测试方法在上节都有所说明了，在烧写程序时一再强调“烧写到NAND Flash”中去，这是有原因的：S3C2410、S3C2440中有被称为“Steppingstone”的4KB内部RAM；当选择从NAND Flash启动CPU时，CPU会通过内部的硬件将NAND Flash开始的4KB字节数据复制到这4KB的内部RAM中（此时内部RAM的起始地址为0），然后跳到地址0开始执行。

通过设置S3C2410/S3C2440的OM1、OM0引脚，可以选择从NAND Flash还是从NOR Flash启动。开发板插上NAND_BOOT跳线时，CPU从NAND Flash启动；否则从NOR Flash启动。NOR Flash虽然可以像内存一样进行读操作，却不可以像内存一样进行写操作，所以从NOR Flash启动时，一般先在代码的开始部分使用汇编指令初始化外接的内存器件（外存），然后将代码复制到外存中，最后跳到外存中继续执行。对于小程序，一般将它烧入NAND Flash中，借助CPU的内部RAM直接运行。

“ARM9 嵌入式系统基础实验篇”的所有实例都是在没有操作系统的裸板上运行的，只能通过JTAG工具烧写。如果读者想在Windows环境下进行这部分实验，在光盘中也提供了可以通过Windows下的工具ADS1.2进行编译的源程序，在/work/hardware/source_ads目录下。

Windows、Linux下的JTAG工具使用相似，直接运行它们即可看到使用方法，此处不再赘述。



第 6 章　存储控制器

本章目标

- 了解 S3C2410/S3C2440 地址空间的布局
- 掌握如何通过总线形式访问扩展的外设，比如内存、NOR Flash、网卡等

总线的使用方法是嵌入式底层开发的基础，了解它之后，再根据外设的具体特性，就可以驱动该外设了。

6.1　使用存储控制器访问外设的原理

6.1.1　S3C2410/S3C2440 的地址空间

S3C2410/S3C2440 的“存储控制器”提供了访问外部设备所需的信号，它有如下特性：

- 支持小字节序、大字节序（通过软件选择）;
- 每个 BANK 的地址空间为 128MB，总共 1GB（8 BANKs）;
- 可编程控制的总线位宽（8/16/32-bit），不过 BANK0 只能选择两种位宽（16/32-bit）;
- 总共 8 个 BANK，BANK0 ~ BANK 5 可以支持外接 ROM、SRAM 等，BANK6 ~ BANK7 除可以支持 ROM、SRAM 外，还支持 SDRAM 等;
- BANK0 ~ BANK6 共 7 个 BANK 的起始地址是固定的;
- BANK7 的起始地址可编程选择;
- BANK6、BANK7 的地址空间大小是可编程控制的;
- 每个 BANK 的访问周期均可编程控制;
- 可以通过外部的“wait”信号延长总线的访问周期;
- 在外接 SDRAM 时，支持自刷新（self-refresh）和省电模式（power down mode）。

S3C2410/S3C2440 对外引出的 27 根地址线 ADDR0～ADDR26 的访问范围只有 128MB，那么如何达到上面所说的 1GB 的访问空间呢？CPU 对外还引出了 8 根片选信号 nGCS0～nGCS7，对应于 BANK0～BANK7，当访问 BANKx 的地址空间时，nGCSx 引脚输出低电平

用来选中外接的设备。这样，每个 nGCSx 对应 128MB 地址空间，8 个 nGCSx 信号总共就对应了 1GB 的地址空间。这 8 个 BANK 的地址空间如图 6.1 所示。

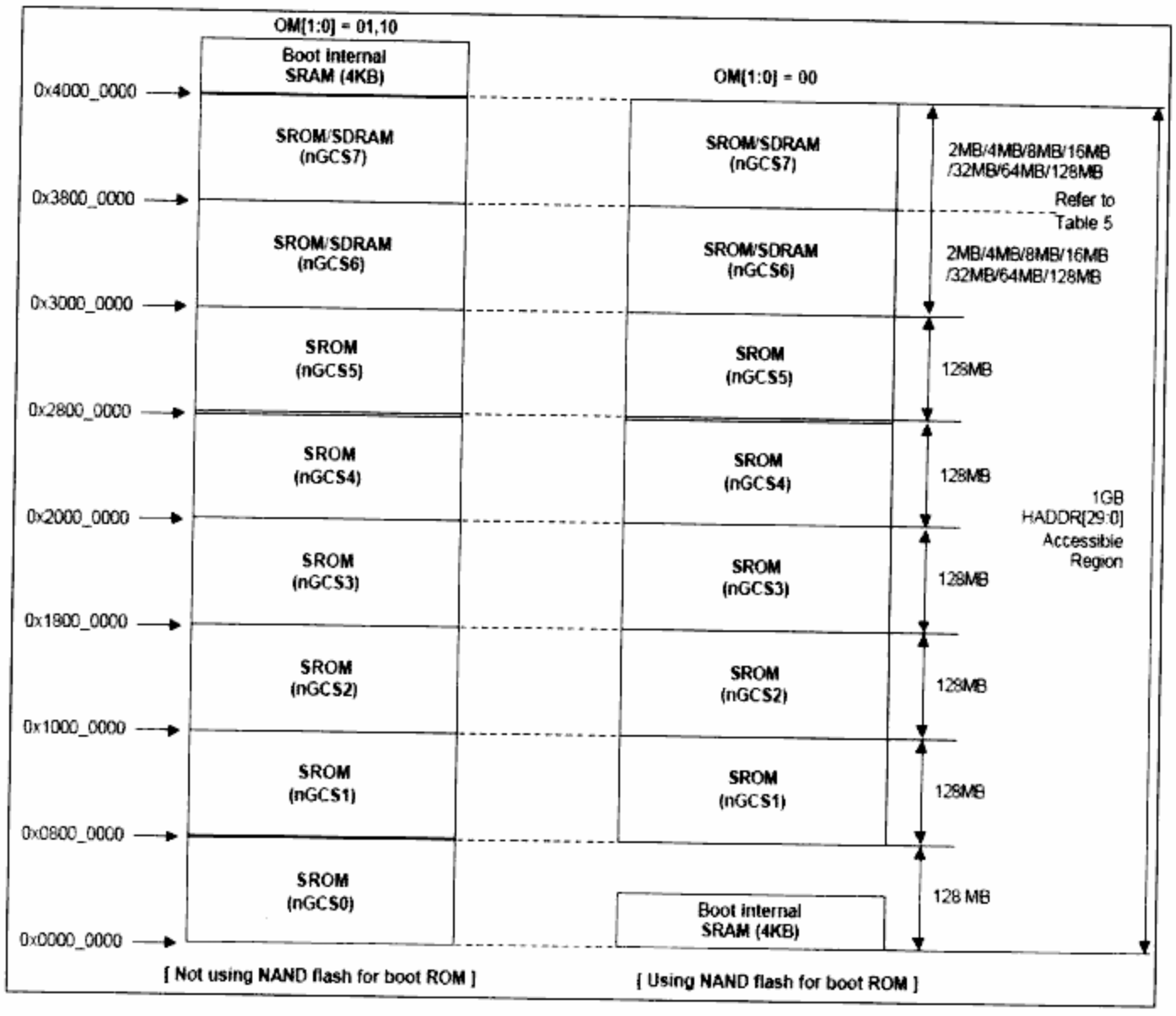


图 6.1 S3C2410/S3C2440 存储控制器的地址空间分布图

如图 6.1 所示，左边对应不使用 NAND Flash 作为启动设备（单板上不接 NAND_BOOT 跳线）时的地址空间布局，右边对应使用 NAND Flash 作为启动设备（单板上接 NAND_BOOT 跳线）时的地址空间布局。

S3C2410/S3C2440 作为 32 位的 CPU，可以使用的地址范围理论上达到 4GB。除去上述用于连接外设的 1GB 地址空间外，还有一部分是 CPU 内部寄存器的地址，剩下的地址空间没有使用。

注意 这里说的是物理地址。

S3C2410/S3C2440 的寄存器地址范围都处于 0x4800000～0x5FFFFFFF，各功能部件的寄存器大体相同，如表 6.1 所示。

表 6.1 S3C2410/S3C2440 功能部件的寄存器地址范围对照表

功能部件	S3C2410		S3C2440	
	起始地址	结束地址	起始地址	结束地址
存储控制器	0x48000000	0x48000030	0x48000000	0x48000030
USB Host 控制器	0x49000000	0x49000058	-	-
中断控制器	0x4A000000	0x4A00001C	-	-

续表

功能部件	S3C2410		S3C2440	
	起始地址	结束地址	起始地址	结束地址
DMA	0x4B000000	0x4B0000E0	-	-
时钟和电源管理	0x4C000000	0x4C000014	-	0x4C000018
LCD 控制器	0x4D000000	0x4D000060	-	-
NAND Flash 控制器	0x4E000000	0x4E000014	-	0x4E00003C
摄像头接口	无	无	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	-	0x54000010
IIS 控制器	0x55000000	0x55000012	-	-
I/O 端口	0x56000000	0x560000B0	-	0x560000CC
实时时钟 RTC	0x57000040	0x5700008B	-	-
A/D 转换器	0x58000000	0x58000010	-	0x58000014
SPI	0x59000000	0x59000034	-	-
SD 接口	0x5A000000	0x5A000040	-	0x5A000043
AC97 音频编码接口	无	无	0x5B000000	0x5B00001C

注："-"表示 S3C2440 的寄存器地址与 S3C2410 相同；"无"表示没有相应功能部件。

6.1.2 存储控制器与外设的关系

本书所用开发板使用了存储控制器的 BANK0~BANK6，分别外接了如下设备：NOR Flash、IDE 接口、10M 网卡 CS8900A、100M 网卡 DM9000、扩展串口芯片 16C2550、SDRAM，连线方式如图 6.2 所示。

根据图 6.1 可以知道各个 BANK 的起始地址，但还需要结合图 6.2 中用到的地址线才能确定相关外设的访问地址。这些地址线所确定的地址值，再加上这个 BANK 的起始地址，就是这个外设的访问地址。

选一个相对复杂的 BANK——扩展串口作为例子。

（1）它使用 nGCS5，起始地址为 0x28000000。

（2）nCSA=ADDR24 || nGCS5，nCSB = !ADDR24 || nGCS5。当 ADDR24 和 nGCS5 均为低电平时选中扩展串口 A；当 ADDR24 为高电平、nGCS5 为低电平时选中扩展串口 B。

（3）CPU 的 ADDR0～ADDR2 连接到扩展串口的 A0～A2，所以访问空间为 8 字节。

综上所述，扩展串口 A 的访问空间为：0x28000000～0x28000007；扩展串口 B 的访问空间为：0x29000000～0x29000007（bit24 为 1）。

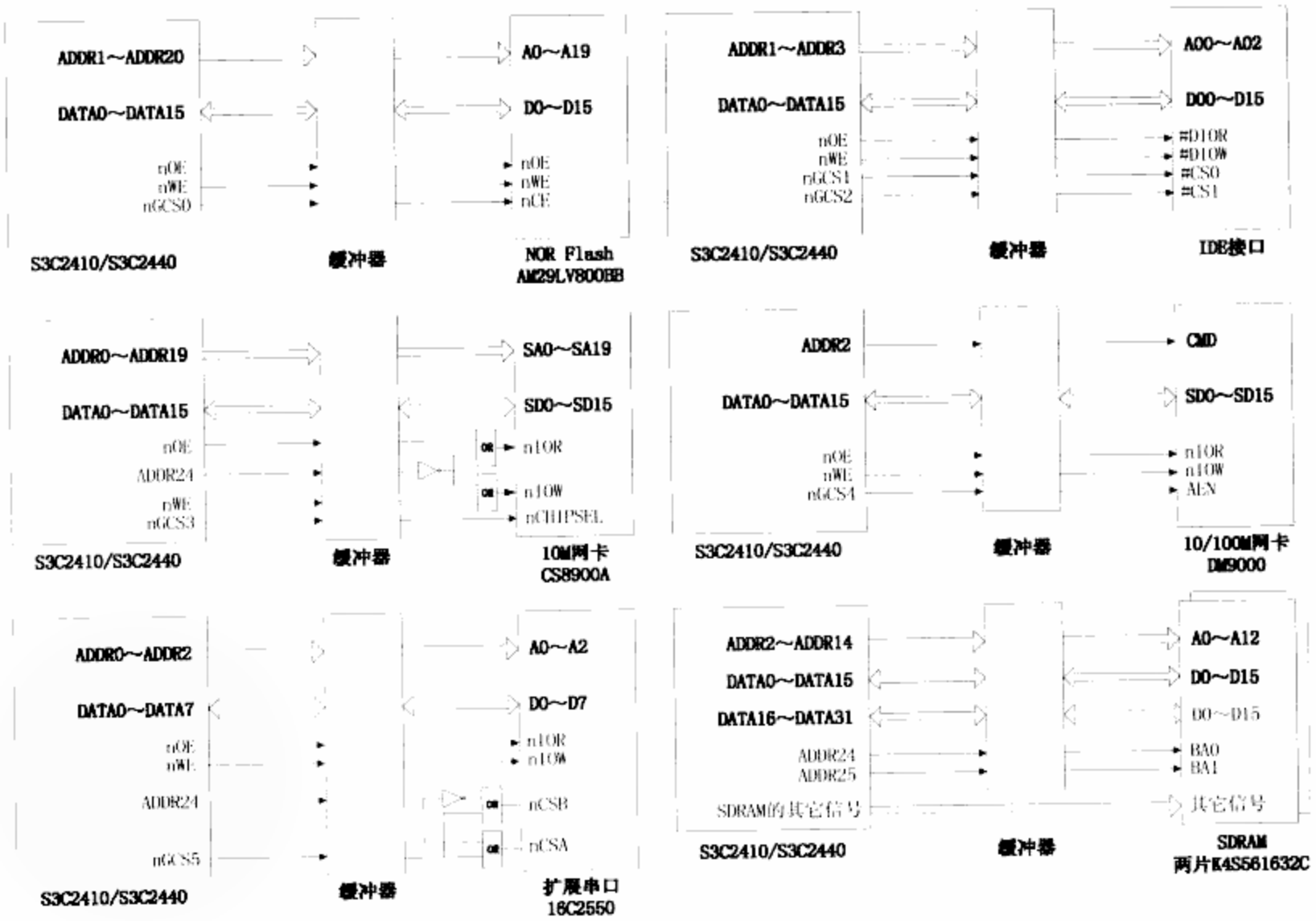


图 6.2 S3C2410/S3C2440 与外设的连线图

BANK0～BANK5 的连接方式都是相似的，BANK6 连接 SDRAM 时复杂一点，CPU 提供了一组用于 SDRAM 的信号。

- SDRAM 时钟有效信号 SCKE;
- SDRAM 时钟信号 SCLK0/SCLK1;
- 数据掩码信号 DQM0/DQM1/DQM2/DQM3;
- SDRAM 片选信号 nSCS0（它与 nGCS6 是同一引脚的两个功能）;
- SDRAM 行地址选通脉冲信号 nSRAS;
- SDRAM 列地址选通脉冲信号 nSCAS;
- 写允许信号 nWE（它不是专用于 SDRAM 的）。

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。和表格的检索原理一样，先指定一个行（Row），再指定一个列（Column），就可以准确地找到所需要的单元格，这就是 SDRAM 寻址的基本原理。这个单元格被称为存储单元，这个表格（存储阵列）就是逻辑 Bank（Logical Bank，下文简称 L-Bank），SDRAM 一般含有 4 个 L-Bank。SDRAM 的逻辑结构如图 6.3 所示。

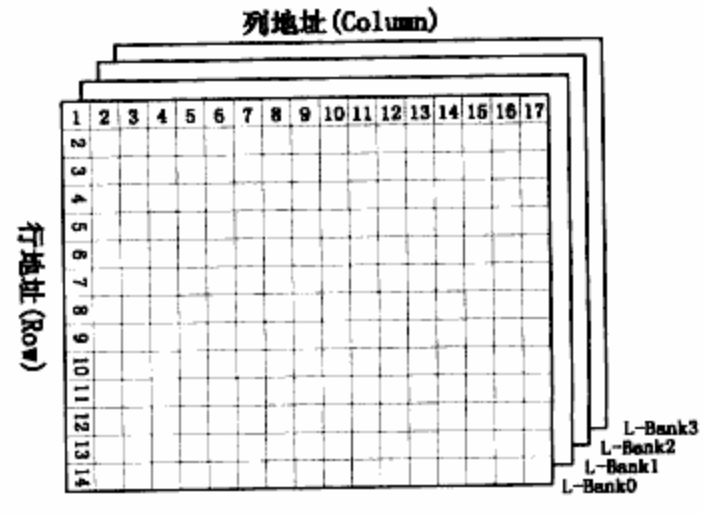


图 6.3 SDRAM 存储结构逻辑图

可以想象，对 SDRAM 的访问可以分为如下 4

个步骤。

（1）CPU 发出的片选信号 nSCS0 有效，它选中 SDRAM 芯片。

（2）SDRAM 中有 4 个 L-Bank，需要两根地址信号来选中其中一个，从图 6.2 可知使用 ADDR24、ADDR25 作为 L-Bank 的选择信号。

（3）对被选中的芯片进行统一的行/列（存储单元）寻址。

根据 SDRAM 芯片的列地址线数目设置 CPU 的相关寄存器后，CPU 就会从 32 位的地址中自动分出 L-Bank 选择信号、行地址信号、列地址信号，然后先后发出行地址信号、列地址信号。L-Bank 选择信号在发出行地址信号的同时发出，并维持到列地址信号结束。

在图 6.2 中，行地址、列地址共用地址线 ADDR2～ADDR14（BANK6 位宽为 32，ADDR0/1 没有使用），使用 nSRAS、nSCAS 两个信号来区分它们。比如本开发板中，使用两根地址线 ADDR24、ADDR25 作为 L-Bank 的选择信号；SDRAM 芯片 K4S561632 的行地址数为 13，列地址数为 9，所以当 nSRAS 信号有效时，ADDR2～ADDR14 上发出的是行地址信号，它对应 32 位地址空间的 bit[23:11]；当 nSCAS 信号有效时，ADDR2～ADDR10 上发出的是列地址信号，它对应 32 位地址空间的 bit[10:2]；由于图 6.2 中 BANK6 以 32 位的宽度外接 SDRAM，ADDR0、ADDR1 恒为 0，不参与译码。

（4）找到了存储单元后，被选中的芯片就要进行统一的数据传输了。

开发板中使用两片 16 位的 SDRAM 芯片并联组成 32 位的位宽，与 CPU 的 32 根数据线（DATA0～DATA31）相连。

BANK6 的起始地址为 0x30000000，所以 SDRAM 的访问地址为 0x30000000～0x33FFFFFF，共 64MB。

对图 6.2 中连接的外设，它们的访问地址（物理地址）如表 6.2 所示。

表 6.2　　存储控制器所接外设的访问地址

BANKx	外设名称	起始地址	结束地址	大小（字节）	位宽
BANK0	NOR Flash	0x00000000	0x001FFFFF	2M	16
BANK1	IDE 接口命令块寄存器	0x08000000	0x0800000F	16	16
BANK2	IDE 接口控制块寄存器	0x10000000	0x1000000F	16	16
BANK3	10M 网卡 CS8900A	0x19000000	0x190FFFFF	1M	16
BANK4	10/100M 网卡 DM9000	只有两个地址 0x20000000 和 0x20000004			16
BANK5	扩展串口 A	0x28000000	0x28000007	8	8
BANK5	扩展串口 B	0x29000000	0x29000007	8	8
BANK6	SDRAM	0x30000000	0x33FFFFFF	64M	32

注：10M 网卡 CS8900A 使用 nIOR、nIOW 作为读/写使能信号时，ADDR24 必须为 1。

6.1.3 存储控制器的寄存器使用方法

存储控制器共有 13 个寄存器，BANK0～BANK5 只需要设置 BWSCON 和 BANKCON*x*（*x* 为 0～5）两个寄存器；BANK6、BANK7 外接 SDRAM 时，除 BWSCON 和 BANKCON*x*

（x 为 6、7）外，还要设置 REFRESH、BANKSIZE、MRSRB6、MRSRB7 等 4 个寄存器。下面分类说明（“[y:x]”表示占据了寄存器的位 x、x+1、……、y）：

1．位宽和等待控制寄存器 BWSCON（BUS WIDTH & WAIT CONTROL REGISTER）

BWSCON 中每 4 位控制一个 BANK，最高 4 位对应 BANK7、接下来 4 位对应 BANK6，依此类推。

（1）STx：启动/禁止 SDRAM 的数据掩码引脚，对于 SDRAM，此位为 0；对于 SRAM，此位为 1。

（2）WSx：是否使用存储器的 WAIT 信号，通常设为 0。

（3）DWx：使用两位来设置相应 BANK 的位宽，0b00 对应 8 位，0b01 对应 16 位，0b10 对应 32 位，0b11 表示保留。

比较特殊的是 BANK0，它没有 ST0 和 WS0，DW0（[2:1]）只读——由硬件跳线决定：0b01 表示 16 位，0b10 表示 32 位，BANK0 只支持 16、32 两种位宽。

对于本开发板，没有使用 BANK7，根据表 6.1 可以确定 BWSCON 寄存器的值为：0x22011110。

2．BANK 控制寄存器 BANKCON*x*（BANK CONTROL REGISTER，*x* 为 0～5）

这几个寄存器用来控制 BANK0～BANK5 外接设备的访问时序，使用默认的 0x0700 即可满足本开发板所接各外设的要求。

3．BANK 控制寄存器 BANKCON*x*（BANK CONTROL REGISTER，*x* 为 6～7）

在 8 个 BANK 中，只有 BANK6 和 BANK7 可以外接 SRAM 或 SDRAM，所以 BANK CON6～BANKCON7 与 BANKCON0～BANK5 有点不同。

（1）MT（[16:15]）：用于设置本 BANK 外接的是 ROM/SRAM 还是 SDRAM。SRAM-0b00，SDRAM-0b11。

当 MT=0b00 时，此寄存器与 BANKCON0～BANKCON5 类似，不再赘述。

当 MT=0b11 时，此寄存器其他值如下设置。

（2）Trcd([3:2])：RAS to CAS delay，设为推荐值 0b01。

（3）SCAN([1:0])：SDRAM 的列地址位数，对于本开发板使用的 SDRAM K4S561632，列地址位数为 9，所以 SCAN = 0b01。如果使用其他型号的 SDRAM，需要查看其数据手册来决定 SCAN 的取值。0b00 表示 8 位，0b01 表示 9 位，0b10 表示 10 位。

综上所述，本开发板中 BANKCON6/7 均设为 0x00018005。

4．刷新控制寄存器 REFRESH（REFRESH CONTROL REGISTER）：设为 0x008C0000 + R_CNT

（1）REFEN([23])：0 = 禁止 SDRAM 的刷新功能，1 = 开启 SDRAM 的刷新功能。

（2）TREFMD([22])：SDRAM 的刷新模式，0 = CBR/Auto Refresh，1 = Self Refresh（一般在系统休眠时使用）。

（3）Trp([21:20])：设为 0 即可。

（4）Tsrc([19:18])：设为默认值 0b11 即可。

（5）Refresh Counter([10:0])：即上述的 R_CNT

R_CNT 可如下计算（SDRAM 时钟频率就是 HCLK)：

```
R_CNT = 2^11 + 1 - SDRAM 时钟频率(MHz) * SDRAM 刷新周期(uS)
```

SDRAM 的刷新周期在 SDRAM 的数据手册上有标明，在本开发板使用的 SDRAM K4S561632 的数据手册上，可看见这么一行“64ms refresh period (8K Cycle)”。所以，刷新周期 = 64ms/8192 = 7.8125 μs。

在未使用 PLL 时，SDRAM 时钟频率等于晶振频率 12MHz。

现在可以计算：R_CNT = 2^11 + 1 −12 * 7.8125 = 1955。

所以，在未使用 PLL 时，REFRESH = 0x008C0000 + 1955 = 0x008C07A3。

5. BANKSIZE 寄存器 REFRESH（BANKSIZE REGISTER）

（1）BURST_EN([7])。

0 = ARM 核禁止突发传输，1 = ARM 核支持突发传输。

（2）SCKE_EN([5])。

0 = 不使用 SCKE 信号令 SDRAM 进入省电模式，1=使用 SCKE 信号令 SDRAM 进入省电模式。

（3）SCLK_EN([4])。

0 = 时刻发出 SCLK 信号，1 = 仅在访问 SDRAM 期间发出 SCLK 信号（推荐）。

（4）BK76MAP([2:0])：设置 BANK6/7 的大小。

BANK6/7 对应的地址空间与 BANK0～5 不同。BANK0～5 的地址空间大小都是固定的 128MB，地址范围是（*x**128M）到（*x* + 1）*128M−1，*x* 表示 0 到 5。BANK6/7 的大小是可变的，以保持这两个空间的地址连续，即 BANK7 的起始地址会随它们的大小变化。BK76MAP 的取值意义如下：

```
0b010 = 128MB/128MB, 0b001 = 64MB/64MB, 0b000 = 32M/32M,
0b111 = 16M/16M, 0b110 = 8M/8M, 0b101 = 4M/4M, 0b100 = 2M/2M
```

本开发板 BANK6 外接 64MB 的 SDRAM，令[2:0]=0b001（64M/64M），表示 BANK6/7 的容量都是 64MB，虽然 BANK7 没有使用。

综上所述，本开发板 BANKSIZE 寄存器的值可算得 0xB1。

6. SDRAM 模式设置寄存器 MRSRB*x*（SDRAM MODE REGISTER SET REGISTER，*x* 为 6～7）

能修改的只有位 CL([6:4])，这是 SDRAM 时序的一个时间参数：

```
[work]0b000 = 1 clock, 0b010 = 2 clocks, 0b011=3 clocks
```

SDRAM K4S561632 不支持 CL = 1 的情况，所以位[6:4]取值为 0b010（CL = 2）或 0b011（CL = 3）。本开发板取最保守的值 0b011，所以 MRSRB6/7 的值为 0x30。

6.2 存储控制器操作实例：使用 SDRAM

6.2.1 代码详解及程序的复制、跳转过程

从 NAND Flash 启动 CPU 时，CPU 会通过内部的硬件将 NAND Flash 开始的 4KB 数据复制到称为“Steppingstone”的 4KB 的内部 RAM 中（起始地址为 0），然后跳到地址 0 开始执行。

本实例先使用汇编语言设置好存储控制器，使外接的 SDRAM 可用；然后把程序本身从 Steppingstone 复制到 SDRAM 处；最后跳到 SDRAM 中执行。

源代码在/work/hardware/sdram 目录中，包含两个文件 head.S、leds.c。其中 leds.c 和第 5 章中的 leds 的代码完全一样，也是让 4 个 LED 从 0～15 轮流计数。

重点在 head.S，它的作用是设置 SDRAM，将程序复制到 SDRAM，然后跳到 SDRAM 继续执行。head.S 的代码如下：

```
01 @*********************************************************************
02 @ File: head.S
03 @ 功能：设置 SDRAM，将程序复制到 SDRAM，然后跳到 SDRAM 继续执行
04 @*********************************************************************
05
06 .equ      MEM_CTL_BASE,      0x48000000
07 .equ      SDRAM_BASE,        0x30000000
08
09 .text
10 .global _start
11 _start:
12     bl  disable_watch_dog                  @ 关闭 WATCHDOG，否则 CPU 会不断重启
13     bl  memsetup                           @ 设置存储控制器
14     bl  copy_steppingstone_to_sdram        @ 复制代码到 SDRAM 中
15     ldr pc, =on_sdram                      @ 跳到 SDRAM 中继续执行
16 on_sdram:
17     ldr sp, =0x34000000                    @ 设置栈
18     bl  main
19 halt_loop:
20     b   halt_loop
21
22 disable_watch_dog:
23     @ 往 WATCHDOG 寄存器写 0 即可
24     mov r1,     #0x53000000
```

```
25      mov r2,     #0x0
26      str r2,     [r1]
27      mov pc,     lr      @ 返回
28
29 copy_steppingstone_to_sdram:
30      @ 将 Steppingstone 的 4KB 数据全部复制到 SDRAM 中去
31      @ Steppingstone 起始地址为 0x00000000，SDRAM 中起始地址为 0x30000000
32
33      mov r1, #0
34      ldr r2, =SDRAM_BASE
35      mov r3, #4*1024
36 1:
37      ldr r4, [r1],#4         @ 从 Steppingstone 读取 4 字节的数据，并让源地址加 4
38      str r4, [r2],#4         @ 将此 4 字节的数据复制到 SDRAM 中，并让目地地址加 4
39      cmp r1, r3              @ 判断是否完成：源地址等于 Steppingstone 的末地址？
40      bne 1b                  @ 若没有复制完，继续
41      mov pc,     lr          @ 返回
42
43 memsetup:
44      @ 设置存储控制器以便使用 SDRAM 等外设
45
46      mov r1,     #MEM_CTL_BASE           @ 存储控制器的 13 个寄存器的开始地址
47      adrl    r2, mem_cfg_val             @ 这 13 个值的起始存储地址
48      add r3,     r1, #52                 @ 13*4 = 52
49 1:
50      ldr r4,     [r2], #4                @ 读取设置值，并让 r2 加 4
51      str r4,     [r1], #4                @ 将此值写入寄存器，并让 r1 加 4
52      cmp r1,     r3                      @ 判断是否设置完所有 13 个寄存器
53      bne 1b                              @ 若没有写成，继续
54      mov pc,     lr                      @ 返回
55
56
57 .align 4
58 mem_cfg_val:
59      @ 存储控制器 13 个寄存器的设置值
60      .long   0x22011110      @ BWSCON
61      .long   0x00000700      @ BANKCON0
62      .long   0x00000700      @ BANKCON1
63      .long   0x00000700      @ BANKCON2
```

```
64      .long   0x00000700      @ BANKCON3
65      .long   0x00000700      @ BANKCON4
66      .long   0x00000700      @ BANKCON5
67      .long   0x00018005      @ BANKCON6
68      .long   0x00018005      @ BANKCON7
69      .long   0x008C07A3      @ REFRESH
70      .long   0x000000B1      @ BANKSIZE
71      .long   0x00000030      @ MRSRB6
72      .long   0x00000030      @ MRSRB7
```

第 12～18 行是程序的主体，为了使得程序结构明了，主要使用了函数调用的方式。

（1）第 12 行禁止 WATCHDOG，否则 WATCHDOG 会不断地重启系统。往 WTCON 寄存器（地址 0x53000000）写入 0 即可禁止 WATCHDOG。

（2）第 13 行设置存储控制器的 13 个寄存器，以便使用 SDRAM。

（3）第 14 行将 Steppingstone 中的代码复制到 SDRAM 中（起始地址为 0x30000000）。

（4）第 15 行向 pc 寄存器直接赋值跳到 SDRAM 中执行下一条指令“ldr sp, =0x34000000”。

（5）第 17 行设置栈，调用 C 函数之前必须设好栈。

（6）第 18 行调用 C 函数 main。

程序是如何从 Steppingstone 跳到 SDRAM 中去执行的呢？

这是通过第 15 行的“ldr pc,=on_sdram”指令来完成的。程序标号“on_sdram”这个地址值在连接程序时被确定为 0x30000010（这是 SDRAM 的地址），执行“ldr pc, =on_sdram”后，程序一下子就跳到 SDRAM 中去了。

“on_sdram”这个地址值为什么等于 0x30000010？

Makefile 中连接程序的命令为“arm-linux-ld -Ttext 0x30000000 head.o sdram.o -o sdram_elf”，意思就是代码段的起始地址为 0x30000000，即程序的第一条指令（第 12 行）的连接地址为 0x30000000，第二条指令（第 13 行）的连接地址为 0x30000004，…，第五条指令（第 17 行）的连接地址为 0x30000010，其程序标号“on_sdram”的值即为 0x30000010。

虽然第 12～14 行指令的连接地址都在 SDRAM 中，但是由于它们都是位置无关的相对跳转指令，所以可以在 Steppingstone 里执行。

Makefile 如下（注意第 4 行，“-Ttext 0x30000000”指定了代码段的起始地址）：

```
01 sdram.bin : head.S  leds.c
02        arm-linux-gcc  -c -o head.o head.S
03        arm-linux-gcc -c -o leds.o leds.c
04        arm-linux-ld -Ttext 0x30000000 head.o leds.o -o sdram_elf
05        arm-linux-objcopy -O binary -S sdram_elf sdram.bin
06        arm-linux-objdump -D -m arm  sdram_elf > sdram.dis
07 clean:
08        rm -f   sdram.dis sdram.bin sdram_elf *.o
```

为了更形象地了解本程序，下面用图 6.4 来演示程序的复制、跳转过程。

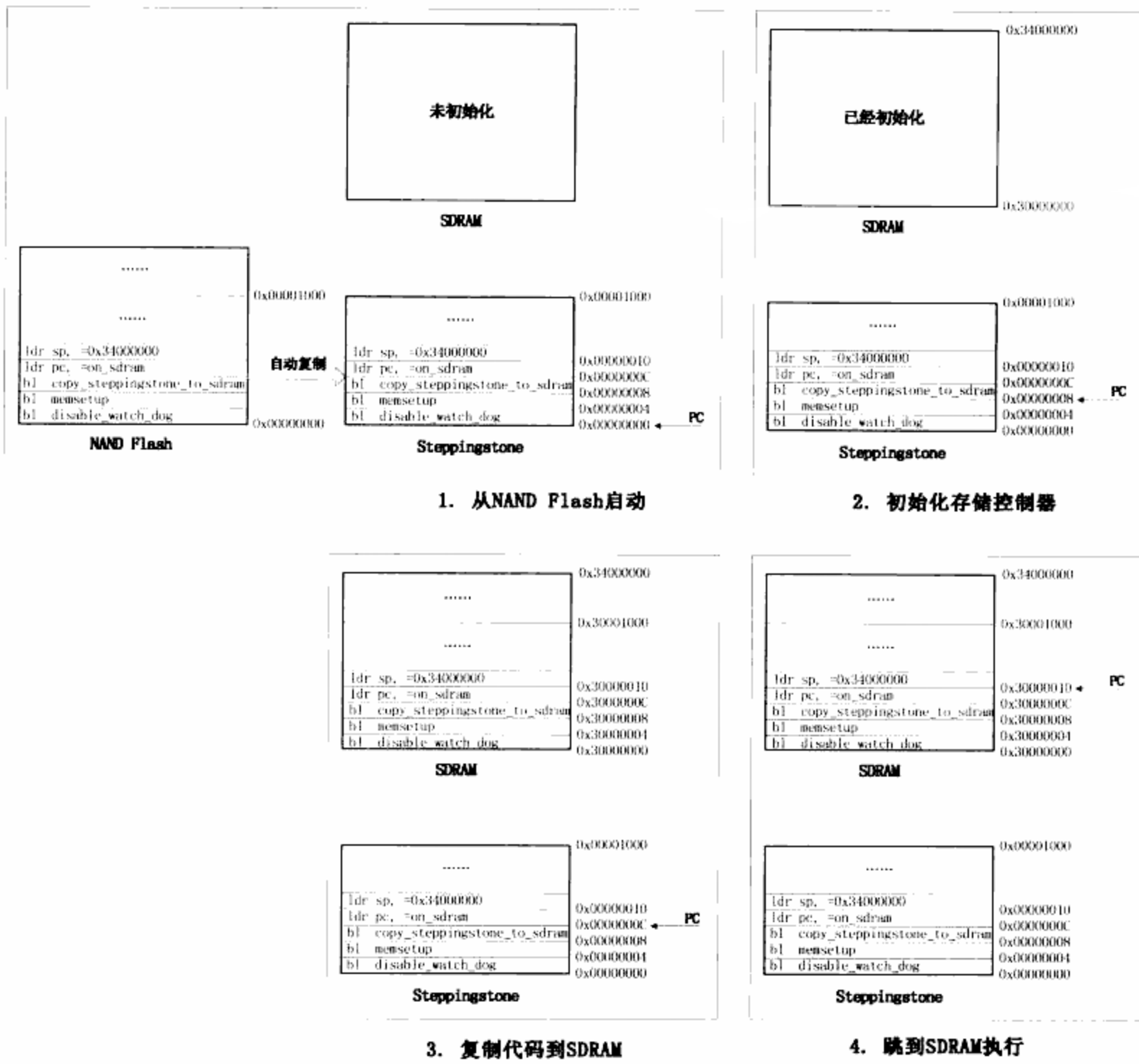
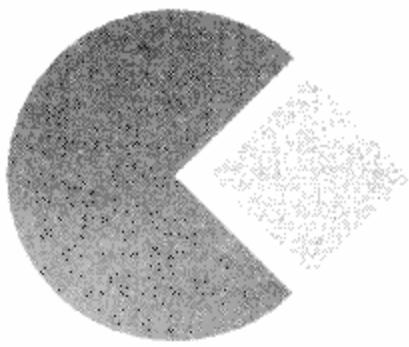


图 6.4 程序从 Steppingstone 到 SDRAM 的执行过程

6.2.2 实例测试

在 sdram 目录下执行 make 指令生成可执行文件 sdram.bin 后，下载到板子上运行。可以发现与 leds 程序相比，LED 闪烁得更慢，原因是外部 SDRAM 的性能比内部 SRAM 差一些。

把程序从性能更好的内部 SRAM 移到外部 SDRAM 中去，是否多此一举呢？内部 SRAM 只有 4KB 大小，如果程序大于 4KB，那么就不能指望完全利用内部 SRAM 来运行了，得想办法把存储在 NAND Flash 中的代码复制到 SDRAM 中去。对于 NAND Flash 中的前 4KB，芯片自动把它复制到内部 SRAM 中，可以很轻松地再把它复制到 SDRAM 中（实验代码中的函数 copy_steppingstone_to_sdram 就有此功能），要复制4KB后面的代码需要使用NAND Flash 控制器来读取 NAND Flash，这是第 8 章的内容。



第 7 章　内存管理单元 MMU

本章目标

- 了解虚拟地址和物理地址的关系
- 掌握如何通过设置 MMU 来控制虚拟地址到物理地址的转化
- 了解 MMU 的内存访问权限机制
- 了解 TLB、Cache、Write buffer 的原理，使用时的注意事项
- 通过实例深刻掌握上述要点

7.1　内存管理单元 MMU 介绍

7.1.1　S3C2410/S3C2440 MMU 特性

内存管理单元（Memory Management Unit）简称 MMU，它负责虚拟地址到物理地址的映射，并提供硬件机制的内存访问权限检查。现代的多用户多进程操作系统通过 MMU 使得各个用户进程都拥有自己独立的地址空间：地址映射功能使得各进程拥有“看起来”一样的地址空间，而内存访问权限的检查可以保护每个进程所用的内存不会被其他进程破坏。

S3C2410/S3C2440 有如下特性。

- 与 ARM V4 兼容的映射长度、域、访问权限检查机制。
- 4 种映射长度：段（1MB）、大页（64kB）、小页（4kB）、极小页（1kB）。
- 对每段都可以设置访问权限。
- 大页、小页的每个子页（sub-page，即被映射页的 1/4）都可以单独设置访问权限。
- 硬件实现的 16 个域。
- 指令 TLB（含 64 个条目）、数据 TLB（含 64 个条目）。
- 硬件访问页表（地址映射、权限检查由硬件自动进行）。
- TLB 中条目的替换采用 round-robin 算法（也称 cyclic 算法）。
- 可以使无效整个 TLB。
- 可以单独使无效某个 TLB 条目。
- 可以在 TLB 中锁定某个条目，指令 TLB、数据 TLB 互相独立。

本章的重点在于地址映射：页表的结构与建立、映射的过程，对于访问权限、TLB、Cache 只粗略介绍。

7.1.2 S3C2410/S3C2440 MMU 地址变换过程

1. 地址的分类

以前的程序是非常小的，可以全部装入内存中。随着技术的发展，出现了以下两种情况。

（1）有的程序很大，它所要求的内存空间超过了内存总容量，不能一次性装入内存；

（2）多道系统中有很多程序需要同时执行，它们要求的内存空间超过了内存总容量，不能把所有程序全部装入内存。

实际上，一个程序在运行之前，没有必要全部装入内存，而仅需要将那些当前要运行的部分先装入内存，其余部分在用到时再从磁盘调入，而当内存耗光时再将暂时不用的部分调出到磁盘。这使得一个大程序可以在较小的内存空间中运行，也使得内存中可以同时装入更多的程序并发执行，从用户的角度看，该系统所具有的内存容量将比实际内存容量大得多，人们把这样的存储器称为虚拟存储器。

虚拟存储器从逻辑上对内存容量进行了扩充，用户看到的大容量只是一种感觉，是虚的，在 32 位的 CPU 系统中，这个虚拟内存地址范围为 0～0xFFFFFFFF，我们把这个地址范围称为虚拟地址空间，其中的某个地址称为虚拟地址。与虚拟地址空间、虚拟地址对应的是物理地址空间、物理地址，它们对应实际的内存。

虚拟地址最终需要转换为物理地址才能读写实际的数据，这通过将虚拟地址空间、物理地址空间划分为同样大小的一块块小空间(称为段或页)，然后为这两类小空间建立映射关系。由于虚拟地址空间远大于物理空间，有可能多块虚拟地址空间映射到同一块物理地址空间，或者有些虚拟地址空间没有映射到具体的物理地址空间上去（可以在使用到时再映射）。如图 7.1 所示为这些映射关系。

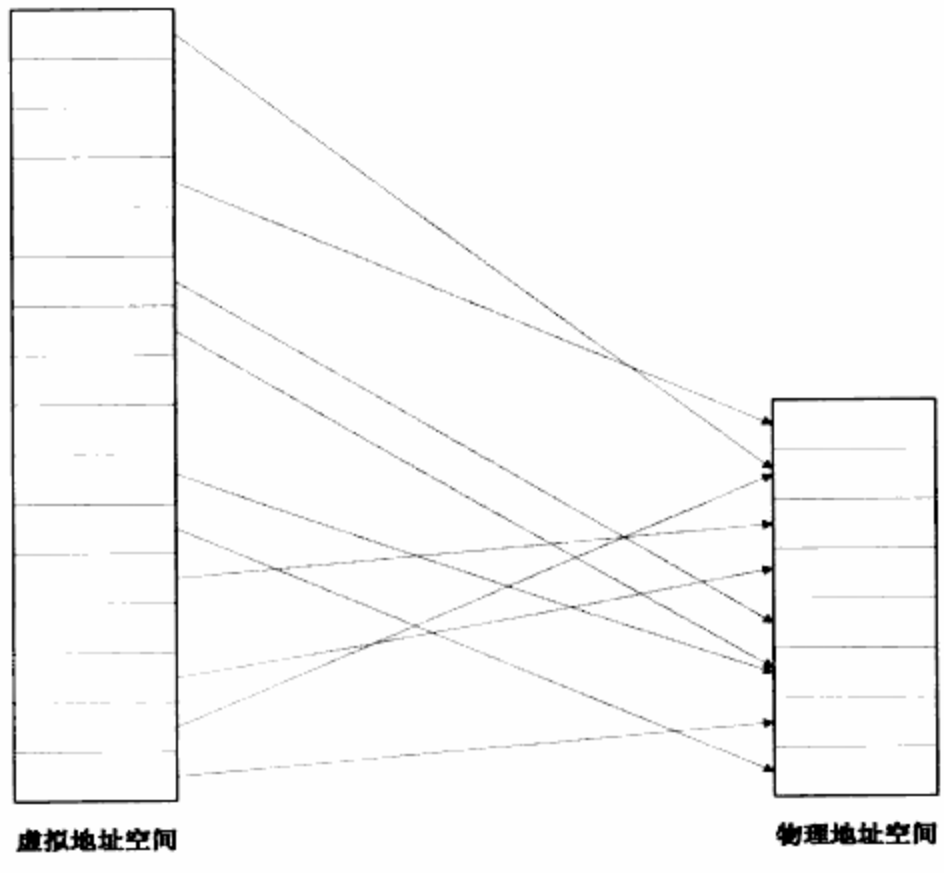


图 7.1 虚拟地址空间和物理地址空间的映射

ARM CPU 上的地址转换过程涉及 3 个概念：虚拟地址（VA，Virtual Address）、变换后的虚拟地址（MVA，Modified Virtual Address）、物理地址（PA，Physical Address）。

没启动 MMU 时，CPU 核、cache、MMU、外设等所有部件使用的都是物理地址。

启动 MMU 后，CPU 核对外发出虚拟地址 VA；VA 被转换为 MVA 供 cache、MMU 使用，在这里 MVA 被转换为 PA；最后使用 PA 读写实际设备（S3C2410/S3C2440 内部寄存器或外接的设备）：

（1）CPU 核看到的、用到的只是虚拟地址 VA，至于 VA 如何最终落实到物理地址 PA 上，CPU 核是不理会的。

（2）而 caches 和 MMU 也是看不见 VA 的，它们利用由 MVA 转换得到 PA。

（3）实际设备看不到 VA、MVA，读写它们时使用的是物理地址 PA。

MVA 是除 CPU 核外的其他部分看见的虚拟地址，VA 与 MVA 之间的变换关系如图 7.2 所示。

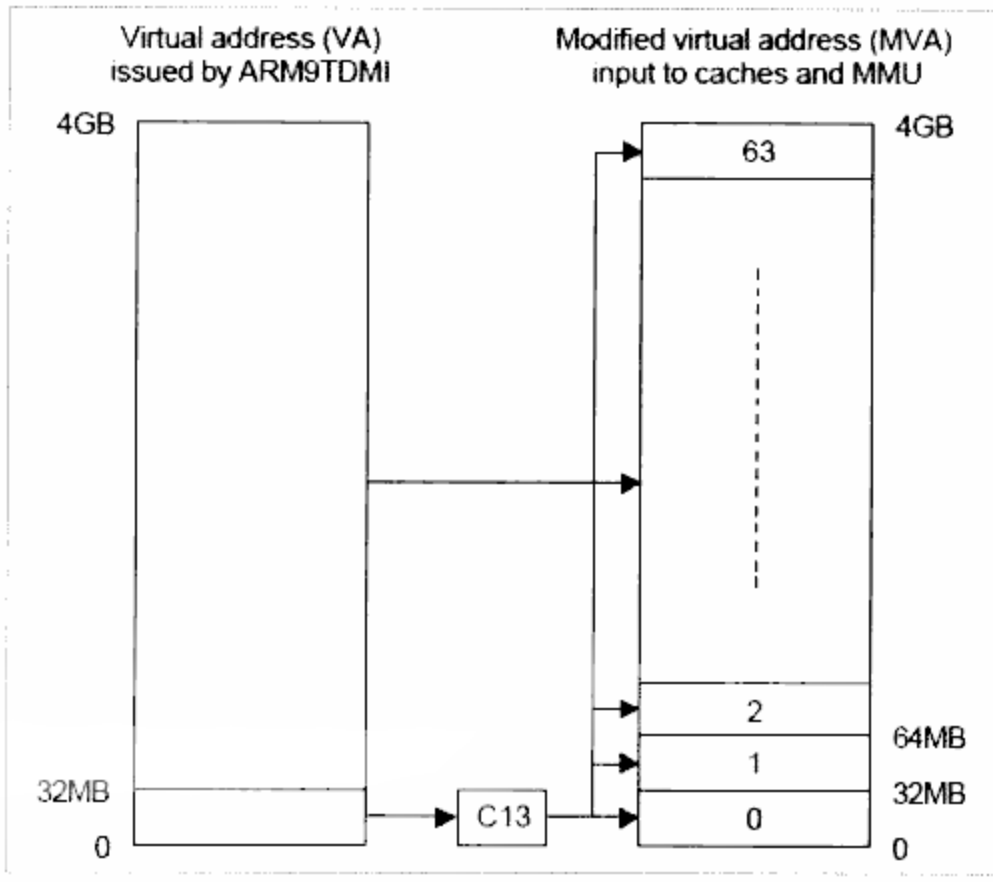


图 7.2 VA 与 MVA 的关系

如果 VA<32M，需要使用进程标识号 PID（通过读 CP15 的 C13 获得）来转换为 MVA。VA 与 MVA 的转换方法如下（这是硬件自动完成的）：

```
if(VA < 32M) then
    MVA = VA | (PID << 25)        // VA < 32M
else
    MVA = VA                      // VA >= 32M
```

利用 PID 生成 MVA 的目的是为了减少切换进程时的代价：不使用 MVA 而直接使用 VA 的话，当两个进程所用的虚拟地址空间（VA）有重叠时，在切换进程时为了把重叠的 VA 映射到不同的 PA 上去，需要重建页表、使无效 caches 和 TLBS 等，代价非常大。使用 MVA 后，进程切换就省事多了：假设两个进程 1、2 运行时的 VA 都是 0～（32M-1），但是它们的 MVA 并不重叠，

分别是 0x02000000～0x03fffff、0x04000000～0x05fffff，这样就不必进行重建页表等工作了。

下面说到的虚拟地址，若没有特别指出，就是指 MVA。

2. 虚拟地址到物理地址的转换过程

将一个虚拟地址转换为物理地址，一般有两个办法：用一个确定的数学公式进行转换或用表格存储虚拟地址对应的物理地址。这类表格称为页表（Page table），页表由一个个条目（Entry）组成；每个条目存储了一段虚拟地址对应的物理地址及其访问权限，或者下一级页表的地址。

在 ARM CPU 中使用第二种方法。S3C2410/S3C2440 最多会用到两级页表：以段（Section，1MB）的方式进行转换时只用到一级页表，以页（Page）的方式进行转换时用到两级页表。页的大小有 3 种：大页（64KB）、小页（4KB）、极小页（1KB）。条目也称为“描述符”（Descriptor），有：段描述符、大页描述符、小页描述符、极小页描述符——它们保存段、大页、小页或极小页的起始物理地址；粗页表描述符、细页表描述符——它们保存二级页表的物理地址。

大概的转换过程如下，请参考图 7.3、图 7.4（图 7.3 是通用的转换过程，图 7.4 是针对 ARM CPU 细化的转换过程）：

（1）根据给定的虚拟地址找到一级页表中的条目；

（2）如果此条目是段描述符，则返回物理地址，转换结束；

（3）否则如果此条目是二级页表描述符，继续利用虚拟地址在此二级页表中找到下一个条目；

（4）如果这第二个条目是页描述符，则返回物理地址，转换结束；

（5）其他情况出错。

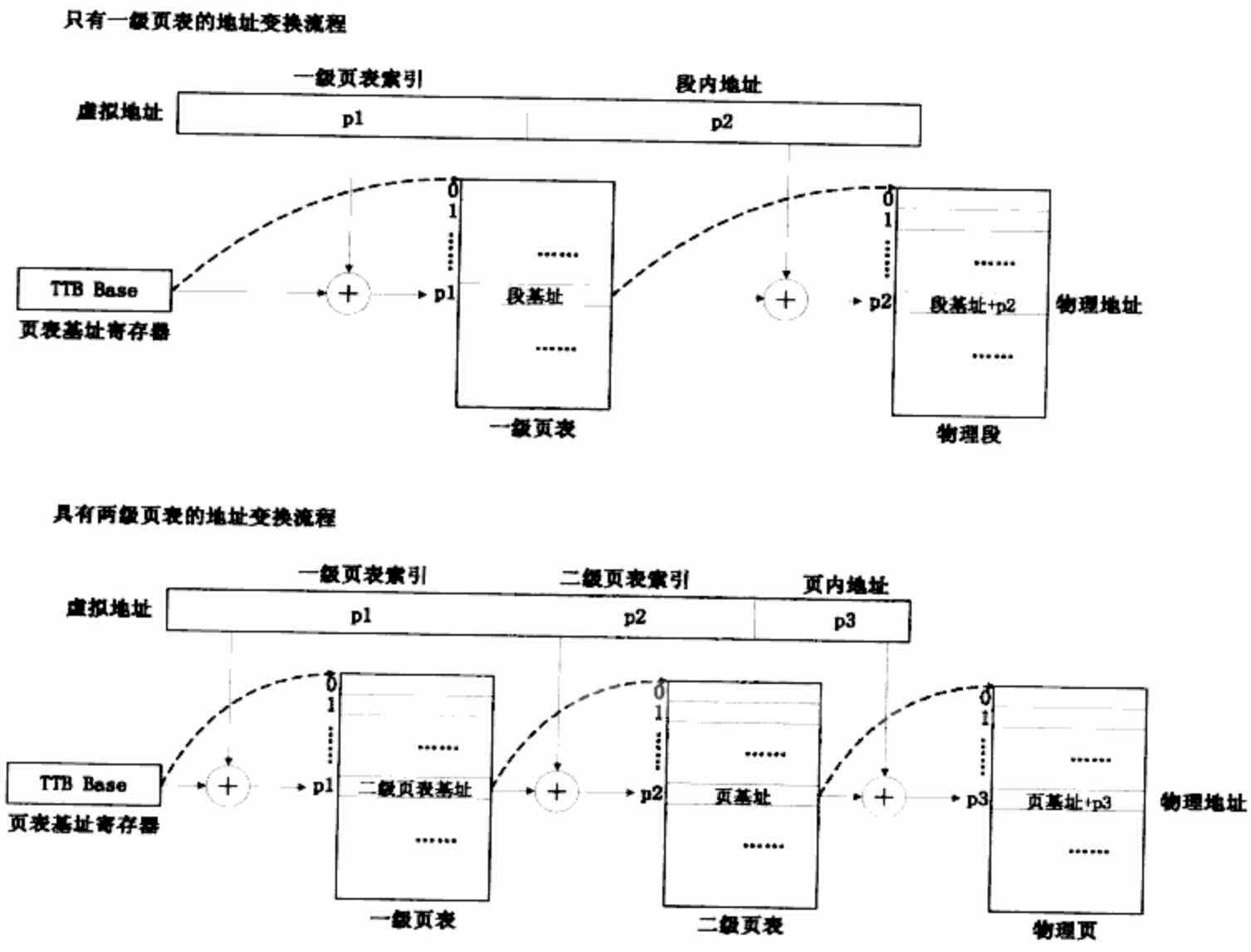


图 7.3 使用页表的地址转换图

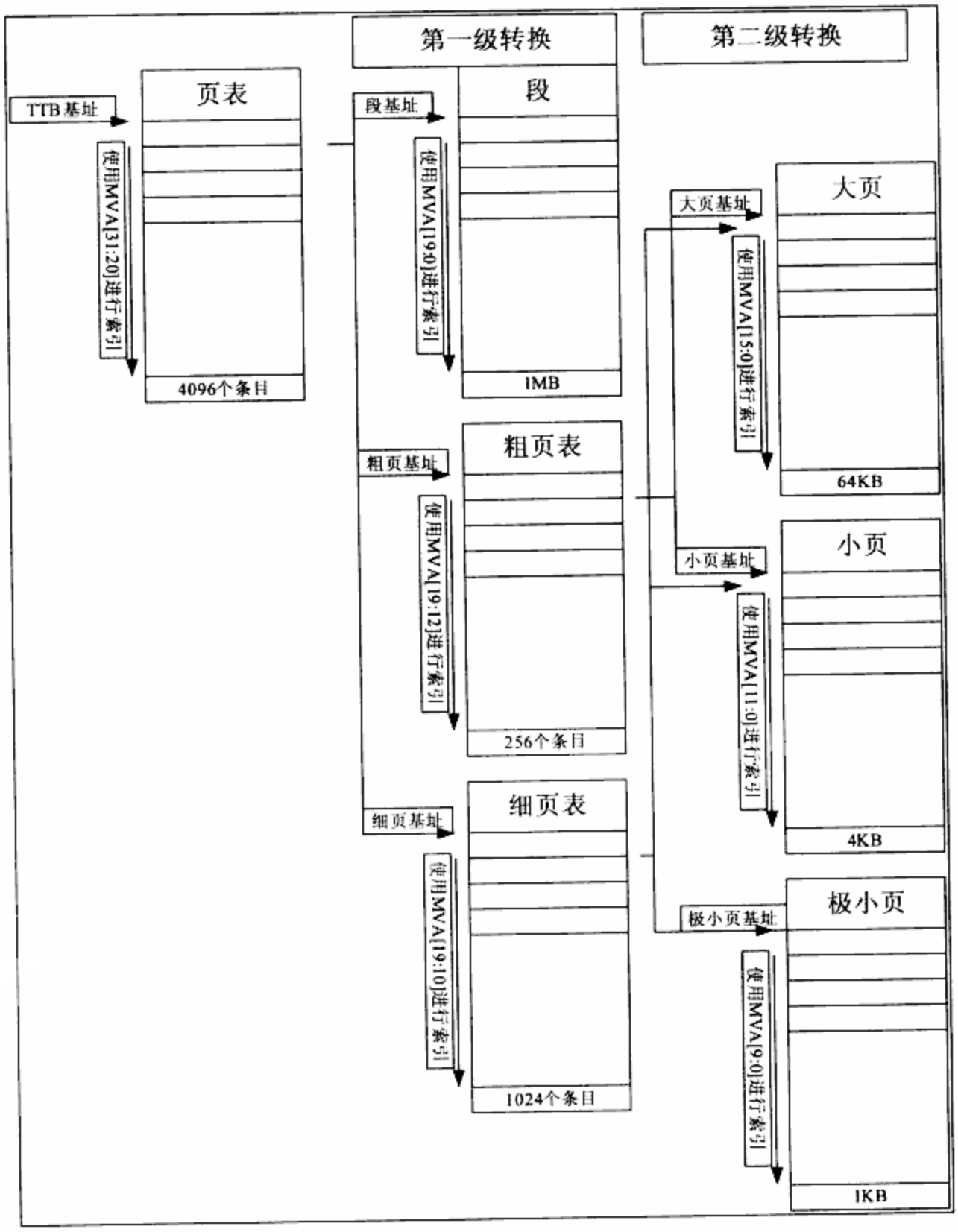


图 7.4 S3C2410/S3C2440 的地址转换图

图 7.3/7.4 中的“TTB base”代表一级页表的地址，将它写入协处理器 CP15 的寄存器 C2（称为页表基址寄存器）即可，如图 7.5 所示，一级页表的地址必须是 16K 对应的（位[14:0]为 0）。

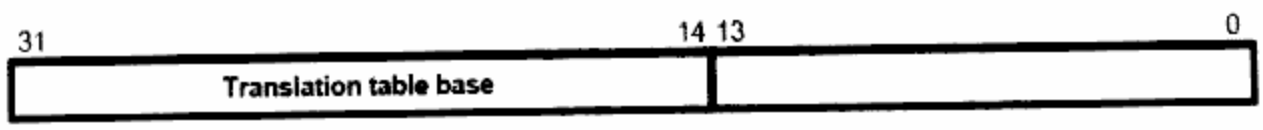


图 7.5 页表基址寄存器

现在先介绍一级页表。32 位 CPU 的虚拟地址空间达到 4GB，一级页表中使用 4096 个描述符来表示这 4GB 空间——每个描述符对应 1MB 的虚拟地址，要么存储了它对应的 1MB 物理空间的起始地址，要么存储了下一级页表的地址。使用 MVA[31:20]来索引一级页表，得到一个描述符，每个描述符占据 4 字节，格式如图 7.6 所示。

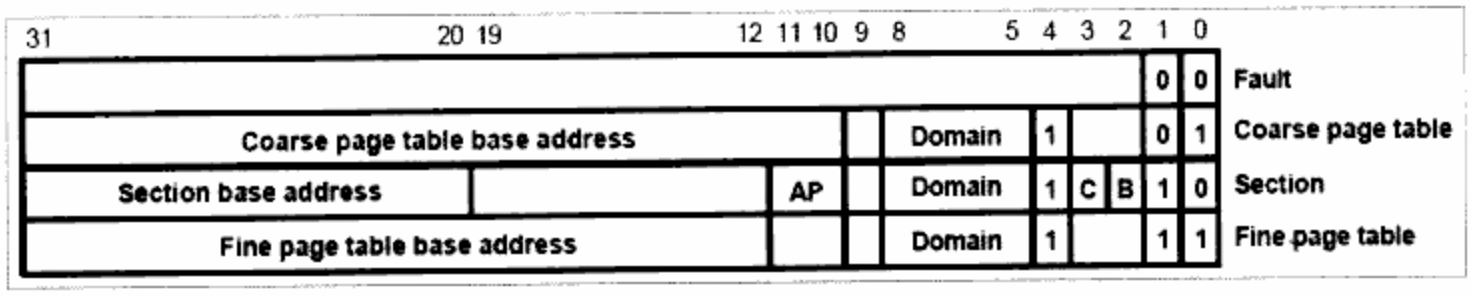


图 7.6 一级页表的描述符格式

根据一级描述符的最低两位，可分为以下 4 种。

（1）0b00：无效。

（2）0b01：粗页表（Coarse page table）。

位[31:10]称为粗页表基址（Coarse page table base address），此描述符的低 10 位填充 0 后就是一个二级页表的物理地址。此二级页表含 256 个条目（所以大小为 1KB），称为粗页表（Coarse page table，见图 7.4）。其中每个条目表示大小为 4KB 的物理地址空间，所以一个粗页表表示 1MB 的物理地址空间。

（3）0b10：段（Section）。

位[31:20]称为段基址（Section base），此描述符的低 20 位填充 0 后就是一块 1MB 物理地址空间的起始地址。MVA[19:0]用来在这 1MB 空间中寻址。所以，描述符的位[31:20]和 MVA[19:0]就构成了这个虚拟地址 MVA 对应的物理地址。

以段的方式进行映射时，虚拟地址 MVA 到物理地址 PA 的转换过程如下（参考图 7.7）。

① 页表基址寄存器位[31:14]和 MVA[31:20]组成一个低两位为 0 的 32 位地址，MMU 利用这个地址找到段描述符。

② 取出段描述符的位[31:20]——即段基址，它和 MVA[19:0]组成一个 32 位的物理地址——这就是 MVA 对应的 PA。

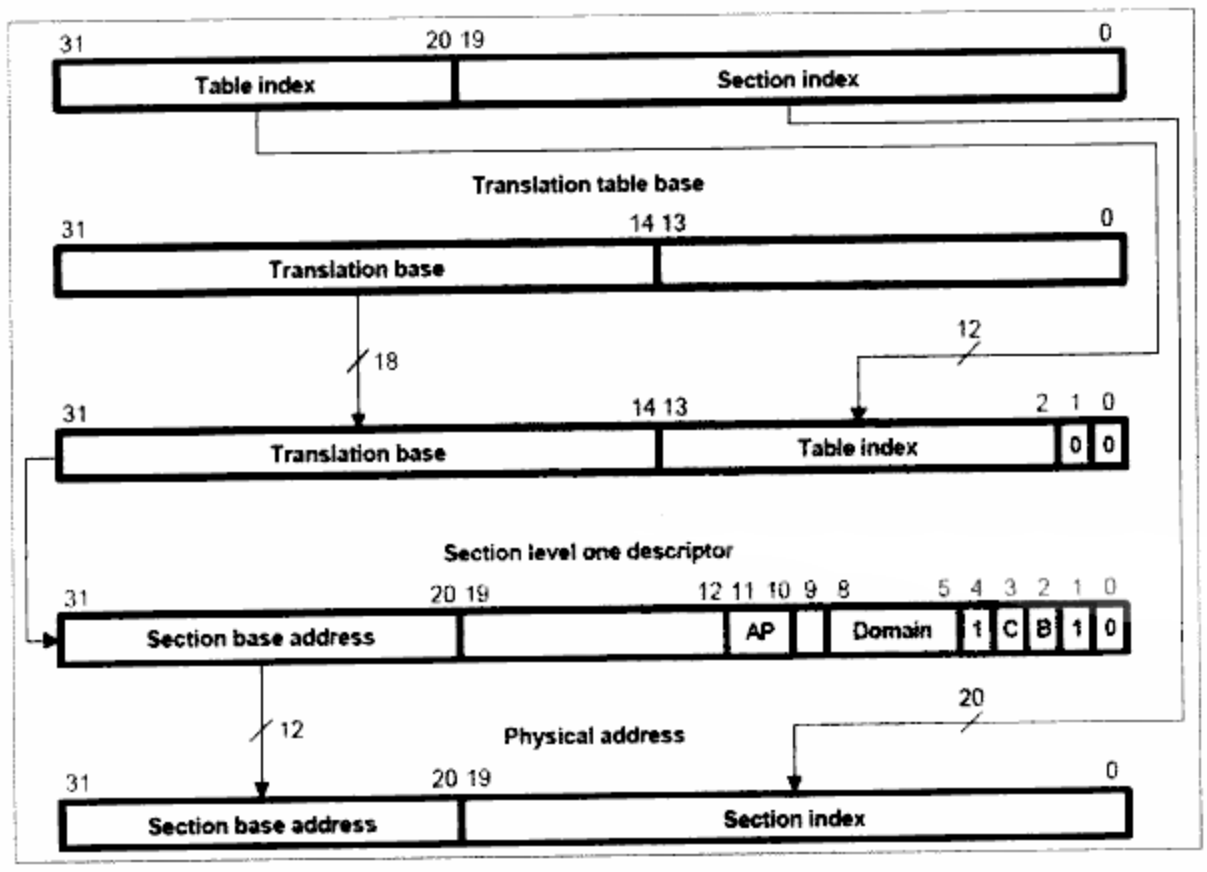


图 7.7 段的地址转换过程

（4）0b11：细页表（Fine page table）

位[31:12]称为细页表基址（Fine page table base address），此描述符的低 12 位填充 0 后就是一个二级页表的物理地址。此二级页表含 1024 个条目（所以大小为 4kB），称为细页表（Fine page table，如图 7.4 所示）。其中每个条目表示大小为 1kB 的物理地址空间，所以一个细页表表示 1MB 的物理地址空间。

以大页（64kB）、小页（4kB）或极小页（1kB）进行地址映射时，需要用到两级页表。二级页表有粗页表、细页表两种，图 7.4 中“Coarse page table”和“Fine page table”就是这两种页表。二级页表中描述符的格式如图 7.8 所示。

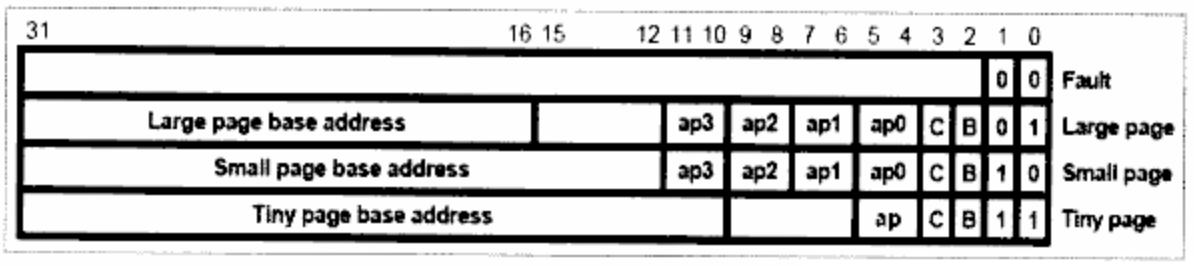


图 7.8 二级页表的描述符格式

根据二级描述符的最低两位，可分为以下 4 种情况。

（1）0b00：无效。

（2）0b01：大页描述符。

位[31:16]称为大页基址（Large page base address），此描述符的低 16 位填充 0 后就是一块 64kB 物理地址空间的起始地址。粗页表中每个条目只能表示 4KB 的物理空间，如果大页描述符保存在粗页表中，则连续 16 个条目都保存同一个大页描述符。类似的，细页表中每个条目只能表示 1kB 的物理空间，如果大页描述符保存在细页表中，则连续 64 个条目都保存同一个大页描述符。

下面以保存在粗页表中的大页描述符为例，说明地址转换的过程（参考图 7.9）。

① 页表基址寄存器位[31:14]和 MVA[31:20]组成一个低两位为 0 的 32 位地址，MMU 利用这个地址找到粗页表描述符。

② 取出粗页表描述符的位[31:10]——即粗页表基址，它和 MVA[19:12]组成一个低两位为 0 的 32 位物理地址——据此即可找到大页描述符。

③ 取出大页描述符的位[31:16]——即大页基址，它和 MVA[15:0]组成一个 32 位的物理地址——这就是 MVA 对应的 PA。

上面步骤②和③中，用于在粗页表中索引的 MVA[19:12]、用于在大页内寻址的 MVA[15:0]有重合的位：位[15:12]。当位[15:12]从 0b0000 变化到 0b1111 时，步骤②返回的大页描述符相同——所以，粗页表中连续 16 个条目都保存同一个大页描述符。

大页描述符保存在细页表中时，地址转换过程与上面类似，如图 7.9 所示，详细过程不再赘述。

（3）0b10：小页描述符。

位[31:12]称为小页基址（Small page base address），此描述符的低 12 位填充 0 后就是一块 4kB 物理地址空间的起始地址。粗页表中每个条目表示 4kB 的物理空间，如果小页描述符保存在粗页表中，则只需要用一个条目来保存一个小页描述符。类似的，细页表中每个条目只能表示 1kB 的物理空间，如果小页描述符保存在细页表中，则连续 4 个条目都保存同一个

小页描述符。

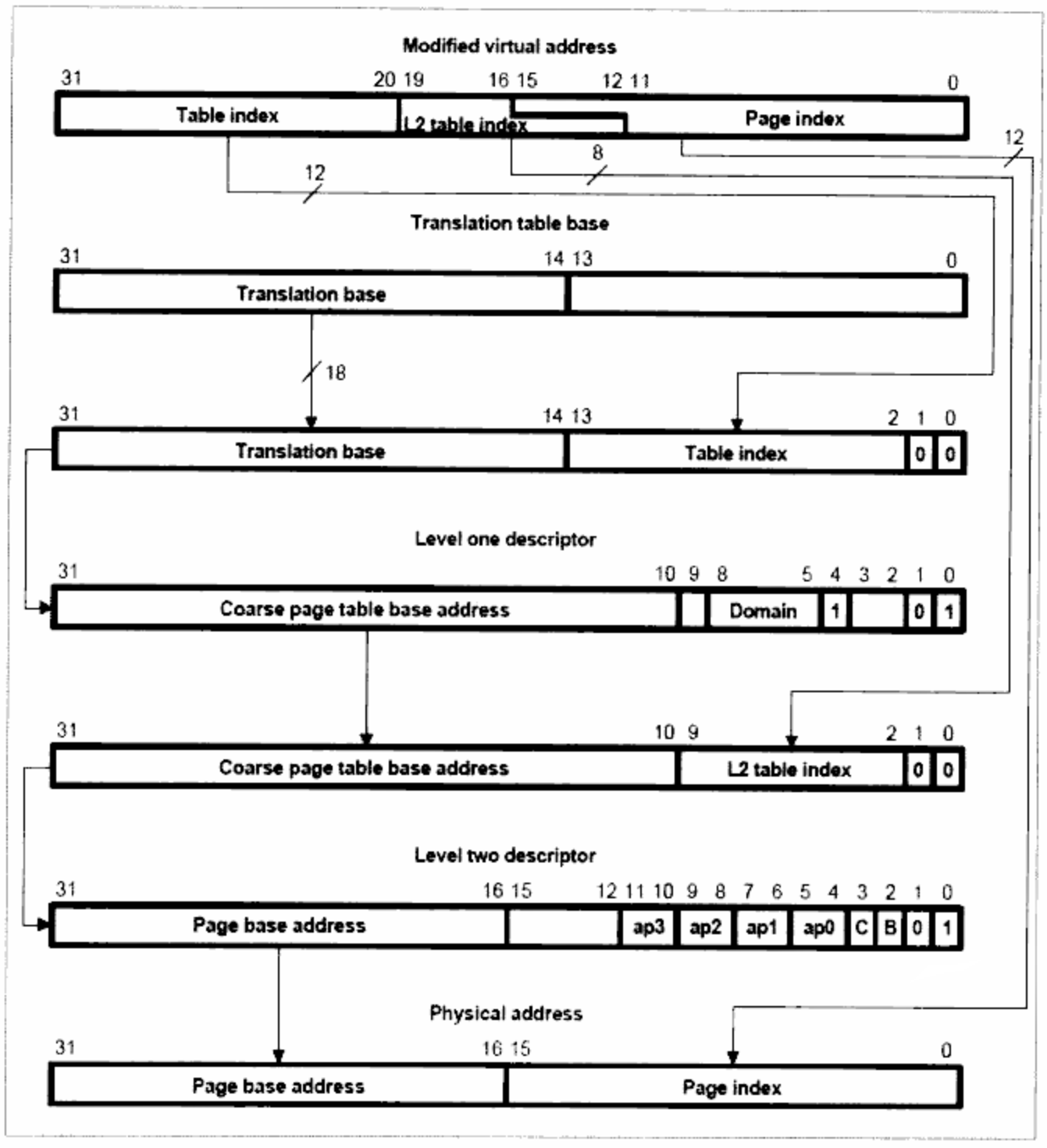


图 7.9 大页的地址转换过程（大页描述符保存在粗页表中）

下面以保存在粗页表中的小页描述符为例，说明地址转换的过程（参考图 7.10）。

① 页表基址寄存器位[31:14]和 MVA[31:20]组成一个低两位为 0 的 32 位地址，MMU 利用这个地址找到粗页表描述符。

② 取出粗页表描述符的位[31:10]——即粗页表基址，它和 MVA[19:12]组成一个低两位为 0 的 32 位物理地址——据此即可找到小页描述符。

③ 取出大页描述符的位[31:12]——即小页基址，它和 MVA[11:0]组成一个 32 位的物理地址——这就是 MVA 对应的 PA。

小页描述符保存在细页表中时，地址转换过程与上面类似，不再赘述。

（4）0b11：极小页描述符。

位[31:10]称为极小页基址（Tiny page base address），此描述符的低 10 位填充 0 后就是一块 1KB 物理地址空间的起始地址。极小页描述符只能保存在细页表中，用一个条目来保存一个极小页描述符。

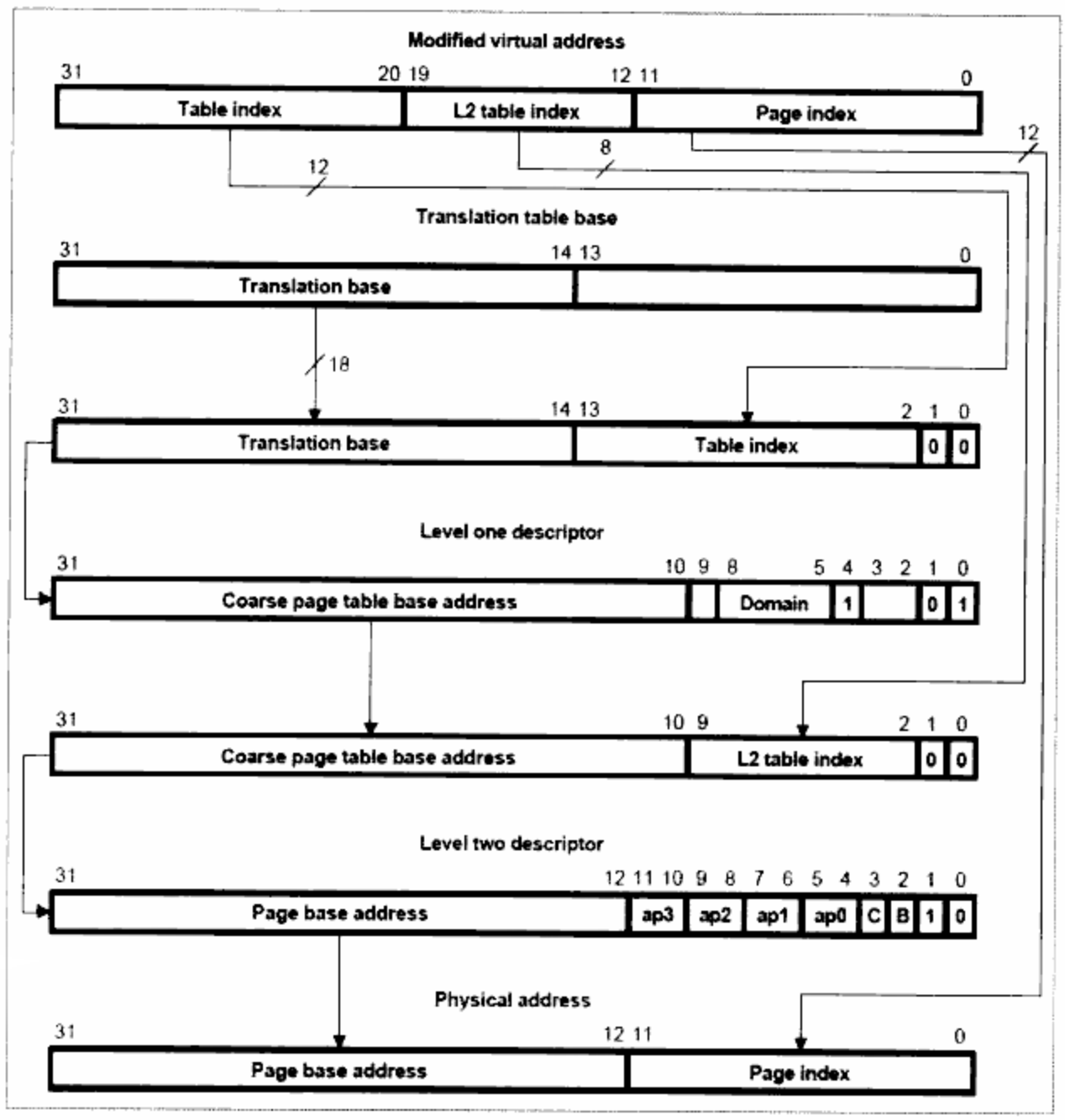


图 7.10 小页的地址转换过程（小页描述符保存在粗页表中）

下面是极小页的地址转换的过程（如图 7.11 所示）。

① 页表基址寄存器位[31:14]和 MVA[31:20]组成一个低两位为 0 的 32 位地址，MMU 利用这个地址找到细页表描述符。

② 取出细页表描述符的位[31:12]——即细页表基址，它和 MVA[19:10]组成一个低两位为 0 的 32 位物理地址——据此即可找到极小页描述符。

③ 取出极小页描述符的位[31:10]——即极小页基址，它和 MVA[9:0]组成一个 32 位的物理地址——这就是 MVA 对应的 PA。

从段、大页、小页、极小页的地址转换过程可知。

（1）以段进行映射时，通过 MVA[31:20]结合页表得到一段（1MB）的起始物理地址，MVA[19:0]用来在段中寻址。

（2）以大页进行映射时，通过 MVA[31:16]结合页表得到一个大页（64kB）的起始物理地址，MVA[15:0]用来在大页中寻址。

（3）以小页进行映射时，通过 MVA[31:12]结合页表得到一个小页（4kB）的起始物理地址，MVA[11:0]用来在小页中寻址。

（4）以极小页进行映射时，通过 MVA[31:10]结合页表得到一个极小页（1kB）的起始物理地址，MVA[9:0]用来在极小页中寻址。

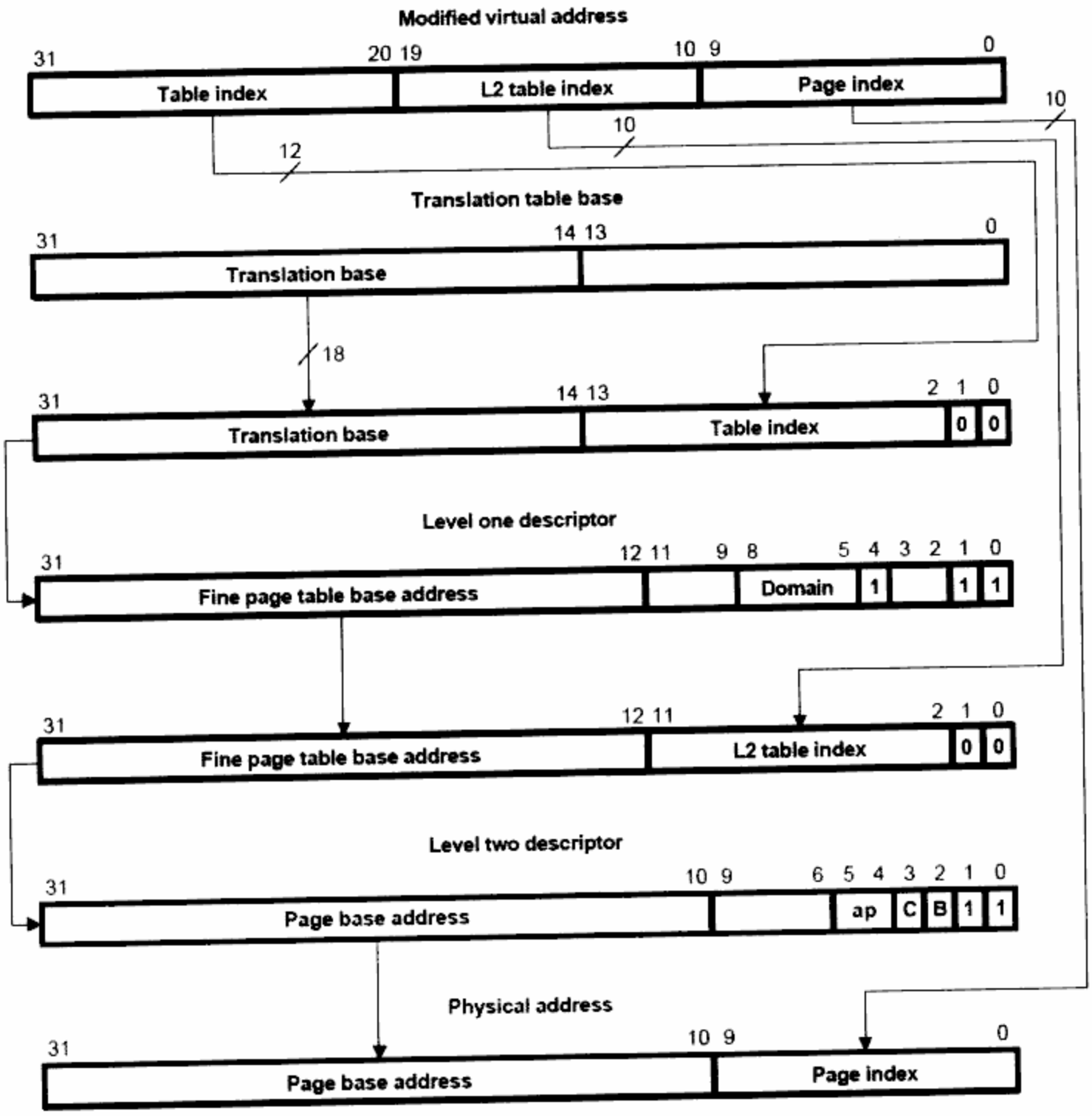


图 7.11 极小页的地址转换过程（极小页描述符保存在粗页表中）

7.1.3 内存的访问权限检查

内存的访问权限检查是 MMU 的主要功能之一，简单地说，它就是决定一块内存是否允许读、是否允许写。这由 CP15 寄存器 C3（域访问控制）、描述符的域（Domain）、CP15 寄存器 C1 的 R/S/A 位、描述符的 AP 位等联合作用。

CP15 寄存器 C1 中的 A 位表示是否对地址进行对齐检查。所谓对齐检查就是，访问字（4 字节的数据）时地址是否为 4 字节对齐，访问半字（2 字节的数据）时地址是否 2 字节对齐，如果地址不对齐则产生“Alignment fault”异常。无论 MMU 是否被开启，都可以进行对齐检查。CPU 读取指令时不进行对齐检查，以字节为单位访问时也不进行对齐检查。对齐检查在 MMU 的权限检查、地址映射前进行。

内存的访问权限检查可以概括为以下两点。

（1）“域”决定是否对某块内存进行权限检查。

（2）“AP”决定如何对某块内存进行权限检查。

如图 7.12 所示，S3C2410/S3C2440 有 16 个域，CP15 寄存器 C3 中每两位对应一个域，用来表示这个域是否进行权限检查。图 7.12 表示 CP15 寄存器 C3 中哪两位对应哪个域，表 7.1 给出了 CP15 寄存器 C3 中这些“两位的数据”的含义。

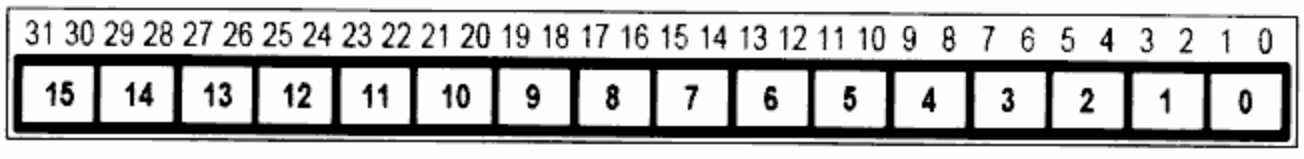


图 7.12 CP15 寄存器 C3（域访问控制寄存器）的格式

表 7.1 域访问控制寄存器中每两位数据的含义

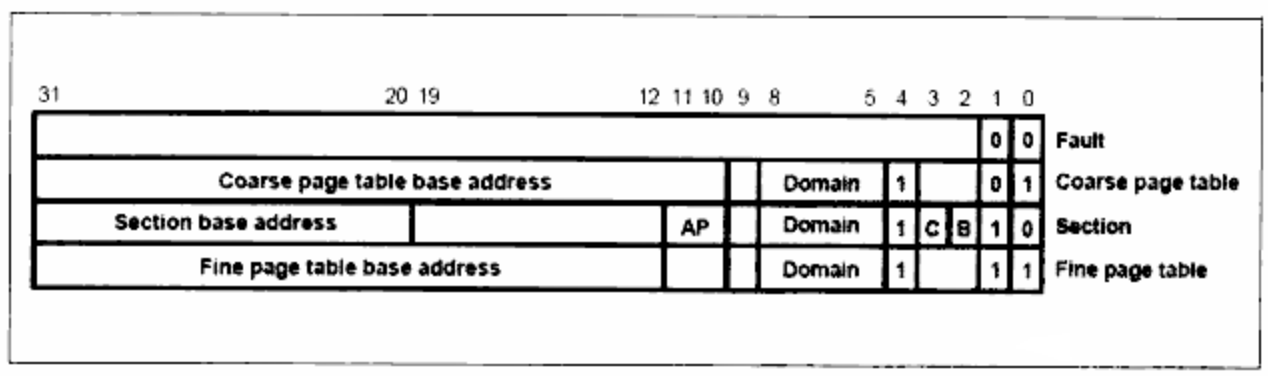
值	含 义	说 明
00	无访问权限	任何访问都将导致“Domain fault”异常
01	客户模式	使用段描述符、页描述符进行权限检查
10	保留	保留，目前相当于“无访问权限”
11	管理模式	不进行权限检查，允许任何访问

图 7.13 中的“Domain”占据 4 字节，用来表示这块内存属于上面定义的 16 个域中的哪一个。举例如下。

（1）段描述符中的“Domain”为 0b0000 时，表示这 1MB 内存属于域 0，如果域访问控制寄存器的位[1:0]等于 0b00，则访问这 1MB 空间时都会产生“Domain fault”的异常，如果域访问控制寄存器的位[1:0]等于 0b11，则使用描述符中的“AP”位进行权限检查。

（2）粗页表中的“Domain”为 0b1111 时，表示这 1MB 内存属于域 15，如果域访问控制寄存器的位[31:30]等于 0b00，则访问这 1MB 空间时都会产生“Domain fault”的异常，如果域访问控制寄存器的位[31:30]等于 0b11，则使用二级页表中的大页/小页描述符中的“ap3”、“ap2”、“ap1”、“ap0”位进行权限检查。

1. 一级页表中的描述符



2. 二级页表中的描述符

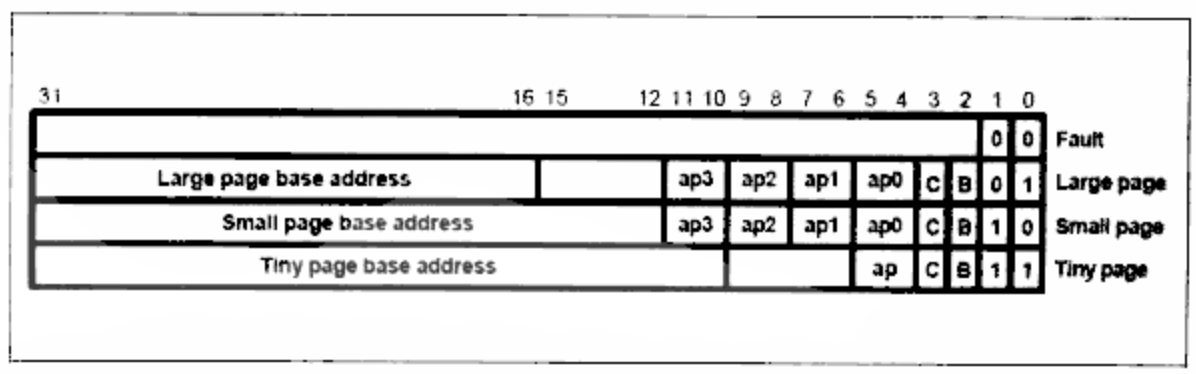


图 7.13 页表描述符中的访问权限控制

图 7.13 中的“AP”、“ap3”、“ap2”、“ap1”、“ap0”结合 CP15 寄存器 C1 的 R/S 位，决定如何进行访问权限检查。首先说明，段描述符中的“AP”控制整个段（1MB）的访问权限；大页描述符中的每个“apx”（x 为 0～3）控制一个大页（64kB）中 1/4 内存的访问权限，即“ap3”对应大页高端的 16KB，“ap0”对应大页低端的 16kB；小页描述符与大页描述符相似，每个“apx”控制一个小页（4kB）的 1/4 内存的访问权限；极小页中的“ap”就控制整个极小页（1kB）的访问权限。

如表 7.2 所示，AP 位、S 位和 R 位的组合，可以产生多种访问权限。需要指出的是，ARM CPU 有 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。

表 7.2　　AP 位、S、R 位的访问权限对照表

AP	S	R	特权模式	用户模式	说明
00	0	0	无访问权限	无访问权限	任何访问将产生“Permission fault”异常
00	1	0	只读	无访问权限	在超级权限下可以进行读操作
00	0	1	只读	只读	任何写操作将产生“Permission fault”异常
00	1	1	保留	—	—
01	x	x	读/写	无访问权限	只允许在超级模式下访问
10	x	x	读/写	只读	在用户模式下进行写操作将产生“Permission fault”异常
11	x	x	读/写	读/写	在所有模式下允许任何访问
xx	1	1	保留	—	—

7.1.4 TLB 的作用

从虚拟地址到物理地址的转换过程可知：使用一级页表进行地址转换时，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；使用两级页表时，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，第三次才是真正的读/写数据。

上述的地址转换过程大大降低了 CPU 的性能，有没有办法改进呢？程序执行过程中，所用到的指令、数据的地址往往集中在一个很小的范围内，其中的地址、数据经常多次使用，这称为程序访问的局部性。由此，通过使用一个高速、容量相对较小的存储器来存储近期用到的页表条目（段/大页/小页/极小页描述符），以避免每次地址转换时都到主存去查找，这样可以大幅度地提高性能。这个存储器用来帮助快速地进行地址转换，称为“转译查找缓存”（Translation Lookaside Buffers，TLB）。

当 CPU 发出一个虚拟地址时，MMU 首先访问 TLB。如果 TLB 中含有能转换这个虚拟地址的描述符，则直接利用此描述符进行地址转换和权限检查；否则 MMU 访问页表找到描述符后再进行地址转换和权限检查，并将这个描述符填入 TLB 中（如果 TLB 已满，则利用 round-robin 算法找到一个条目，然后覆盖它），下次再使用这个虚拟地址时就可以直接使用 TLB 中的描述符了。

使用 TLB 需要保证 TLB 中的内容与页表一致，在启动 MMU 之前、在页表中的内容发生变化后，尤其要注意这点。S3C2410/S3C2440 可以使无效（Invalidate）整个 TLB，或者通过某个虚拟地址使无效 TLB 中的某个条目。一般的做法是：在启动 MMU 之前使无效整个 TLB，改变页表时，使无效所涉及的虚拟地址对应的 TLB 中的条目。

7.1.5 Cache 的作用

同样基于程序访问的局部性，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，把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这个存储器，供 CPU 在一段时间内使用，这对提高程序的运行速度有很大的作用。这个介于主存和 CPU 之间的高速小容量存储器称作高速缓冲存储器（Cache）。

启用 Cache 后，CPU 读取数据时，如果 Cache 中有这个数据的复本则直接返回，否则从主存中读入数据，并存入 Cache 中，下次再使用（读/写）这个数据时，可以直接使用 Cache 中的复本。

启用 Cache 后，CPU 写数据时有写穿式和回写式两种方式。

（1）写穿式（Write Through）。

任一从 CPU 发出的写信号送到 Cache 的同时，也写入主存，以保证主存的数据能同步地更新。它的优点是操作简单，但由于主存的慢速，降低了系统的写速度并占用了总线的时间。

（2）回写式（Write Back）。

为了克服贯穿式中每次数据写入时都要访问主存，从而导致系统写速度降低并占用总线时间，尽量减少对主存的访问次数，又有了回写式。

它是这样工作的：数据一般只写到 Cache，这样有可能出现 Cache 中的数据得到更新而主存中的数据不变（数据陈旧）的情况。但此时可在 Cache 中设一标志地址及数据陈旧的信息，只有当 Cache 中的数据被换出或强制进行“清空”操作时，才将原更新的数据写入主存相应的单元中。这样保证了 Cache 和主存中的数据保持一致。

先介绍 Cache 的两个操作。

（1）“清空”（clean）：把 Cache 或 Write buffer 中已经脏的（修改过，但未写入主存）数据写入主存。

（2）“使无效”（Invalidate）：使之不能再使用，并不将脏的数据写入主存。

S3C2410/S3C244 内置了指令 Cache（ICaches）、数据 Cache（DCaches）、写缓存（Write buffer）。下面的内容需要用到页表中描述符的 C、B 位，为了方便读者，先把这些描述符用图 7.14 表示出来。下文中，描述符中的 C 位称为 Ctt，B 位称为 Btt。

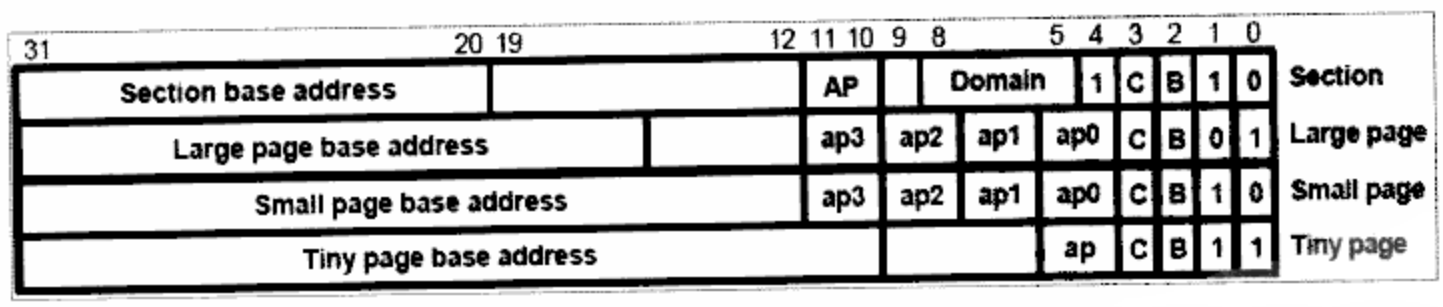


图 7.14 段/大页/小页/极小页描述符

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

功能是关闭着的。往 Icr 位（即 CP15 协处理器中寄存器 1 的第 12 位）写 1 可以启动 ICaches，写 0 可以停止 ICaches。

ICaches 一般在 MMU 开启之后被使用，此时页表中描述符的 C 位（称为 Ctt）用来表示一段内存是否可以被 Cache。若 Ctt=1,则允许 Cache,否则不允许被 Cache。但是，即使 MMU 没有开启，ICaches 也是可以被使用的，这时 CPU 读取指令（以后简称“取指”）时所涉及的内存都被当作是允许 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。

ICaches 被开启后，CPU 每次取指时都会先在 ICaches 中查看是否能找到所要的指令，而不管 Ctt 是 0 还是 1。如果找到了，称为 Cache 命中（Cache hit）；如果找不到，称为 Cache 缺失（Cache miss）。ICaches 被开启后，CPU 的取指分为如下 3 种情况。

（1）Cache 命中且 Ctt 为 1 时，从 ICaches 中取出指令，返回 CPU。

（2）Cache 缺失且 Ctt 为 1 时，CPU 从主存中读出指令。同时，一个称为“8-word linefill”的动作将发生，这个动作把该指令所处区域的 8 个 word 写进 ICaches 的某个条目中。这有可能会覆盖某个条目，可以使用 Pseudo-random 算法或 round-robin 算法在 ICaches 中选出某个没有被锁定的条目。可以通过 CP15 协处理器中寄存器 1 的第 14 位来选择使用哪种算法。

（3）Ctt 为 0 时，CPU 从主存中读出指令。

2. 数据 Cache（DCaches）

与 ICaches 相似，系统刚上电或复位时，DCaches 中的内容也是无效的，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 Ccr 位（即 CP15 协处理器中寄存器 1 的第 2 位）写 1 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 每次读写数据时都会先在 DCaches 中查看是否能找到所要的数据，而不管 Ctt 是 0 还是 1。如果找到了，称为 Cache 命中（Cache hit）；如果找不到，称为 Cache 缺失（Cache miss）。

通过表 7.3 可以知道 DCaches 和 Write buffer 在 CCr、Ctt 和 Btt 的各种取值下，如何工作。表中“Ctt and Ccr”一项里面的值是 Ctt 与 Ccr 进行逻辑与之后的值（Ctt && Ccr）。

表 7.3　　DCaches 和 Write buffer 配置表

Ctt and Ccr	Btt	DCaches、Write buffer 和主存的访问方式
0	0	Non-cached, non-buffered （NCNB） 读写数据时都是直接操作主存，并且可以被外设中止； 写数据时不使用 Write buffer，CPU 会等待写操作完成； 不会出现 Cache 命中

续表

Ctt and Ccr	Btt	DCaches、Write buffer 和主存的访问方式
0	1	Non-cached buffered （NCB） 读写数据时都是直接操作主存； 不会出现 Cache 命中； 写数据时，数据先存入 Write buffer，并在随后写入主存； 数据存入 Write buffer 后，CPU 立即继续执行； 读数据时，可以被外设中止； 写数据时，无法被外设中止
1	0	Cached, write-through（WT，写通）mode 读数据时，如果 Cache 命中则从 Cache 中返回数据，不读取主存； 读数据时，如果 Cache 缺失则从读主存中返回数据，并导致“linefill”的动作； 写数据时，数据先存入 Write buffer，并在随后写入主存； 数据存入 Write buffer 后，CPU 立即继续执行； 写数据时，如果 Cache 命中则新数据也写入 Cache 中； 写数据时，无法被外设中止
1	1	Cached, write-back （WB，写回） mode 读数据时，如果 Cache 命中则从 Cache 中返回数据，不读取主存； 读数据时，如果 Cache 缺失则从读主存中返回数据，并导致“linefill”的动作； 写数据时，如果 Cache 缺失则将数据先存入 Write buffer，存储完毕后 CPU 立即继续执行，这些数据在随后写入主存； 写数据时，如果 Cache 命中则在 Cache 中更新数据，并设置这些数据为“脏的”，但是不会写入主存； 无论 Cache 命中与否，写数据都无法被外设中止

与 TLB 类似，使用 Cache 时需要保证 Cache、Write buffer 的内容和主存内容保持一致，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。

在实际编写程序时，要注意如下几点。

（1）开启 MMU 前，使无效 ICaches、DCaches 和 Write buffer。

（2）关闭 MMU 前，清空 ICaches、DCaches，即将“脏”数据写到主存上。

（3）如果代码有变，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，否则程序的运行结果无法预料。比如寄存器、非内存的外设（扩展串口、网卡等）。

S3C2410/S3C2440 提供了相关指令来操作 Cache 和 Write buffer，可以使无效整个 ICaches

或其中的某个条目，可以清空、使无效整个 DCaches 或其中的某个条目。这些指令在后面介绍。

7.1.6 S3C2410/S3C2440 MMU、TLB、Cache 的控制指令

S3C2410/S3C2440 中，除了有一个 ARM920T 的 CPU 核外，还有若干个协处理器。协处理器也是一个微处理器，它被用来帮助主 CPU 完成一些特殊功能，比如浮点计算等。对 MMU、TLB、Cache 等的操作就涉及协处理器。CPU 核与协处理器间传送数据时使用这两条指令：MRC 和 MCR，它们的格式如下：

```
<MCR|MRC>{cond} p#,<expression1>,Rd,cn,cm{,<expression2>}
MRC             //从协处理器获得数据，传给 ARM920T CPU 核的寄存器
MCR             //数据从 ARM920T CPU 核的寄存器传给协处理器
{cond}          //执行条件，省略时表示无条件执行
p#              //协处理器序号
<expression1>   //一个常数
Rd              //ARM920T CPU 核的寄存器
cn 和 cm        //协处理器中的寄存器
<expression2>   //一个常数
```

其中，<expression1>、cn、cm、<expression2>仅供协处理器使用，它们的作用如何取决于具体的协处理器。

7.2 MMU 使用实例：地址映射

7.2.1 程序设计

程序源码位于/work/hardware/mmu 目录下。

本开发板 SDRAM 的物理地址范围处于 0x30000000～0x33FFFFFF，S3C2410/S3C2440 的寄存器地址范围都处于 0x4800000～0x5FFFFFFF。在第 5 章中，通过往 GPBCON 和 GPBDAT 这两个寄存器的物理地址 0x56000010、0x56000014 写入特定的数据来驱动 4 个 LED。

本章的实例将开启 MMU，并将虚拟地址空间 0xA0000000～0xA0100000 映射到物理地址空间 0x56000000～0x56100000 上，这样，就可以通过操作地址 0xA0000010、0xA0000014 来达到驱动这 4 个 LED 的同样效果。

另外，将虚拟地址空间 0xB0000000～0xB3FFFFFF 映射到物理地址空间 0x30000000～0x33FFFFFF 上，并在连接程序时将一部分代码的运行地址指定为 0xB0004000（这个数值有点奇怪，看下去就会明白），看看能否令程序跳转到 0xB0004000 处执行。

本章程序只使用一级页表，以段的方式进行地址映射。32 位 CPU 的虚拟地址空间达到 4GB，一级页表中使用 4096 个描述符来表示这 4GB 空间（每个描述符对应 1MB 的虚拟地址），每个描述符占用 4 字节，所以一级页表占 16KB。本实例使用 SDRAM 的开始 16KB 来存放一级页表，所以剩下的内存开始物理地址为 0x30004000。

将程序代码分为两部分：第一部分的运行地址设为 0，它用来初始化 SDRAM、复制第二部分的代码到 SDRAM 中（存放在 0x30004000 开始处）、设置页表、启动 MMU，最后跳到 SDRAM 中（地址 0xB0004000）去继续执行；第二部分的运行地址设为 0xB0004000，它用来驱动 LED。

根据上面的叙述，程序流程如图 7.15 所示。

图 7.15 MMU 实例程序流程

7.2.2 代码详解

1. 第一部分代码分析

程序源码分 3 个文件：head.S、init.c、leds.c。

（1）head.S 代码详解。

head.S 文件如下：

```
01 @*****************************************************************************
02 @ File: head.S
03 @ 功能: 设置 SDRAM，将第二部分代码复制到 SDRAM，设置页表，启动 MMU，
04 @       然后跳到 SDRAM 继续执行
05 @*****************************************************************************
06
```

```
07 .text
08 .global _start
09 _start:
10     ldr sp, =4096              @ 设置栈指针，以下都是 C 函数，调用前需要设好栈
11     bl  disable_watch_dog      @ 关闭 WATCHDOG，否则 CPU 会不断重启
12     bl  memsetup               @ 设置存储控制器以使用 SDRAM
13     bl  copy_2th_to_sdram      @ 将第二部分代码复制到 SDRAM
14     bl  create_page_table      @ 设置页表
15     bl  mmu_init               @ 启动 MMU
16     ldr sp, =0xB4000000        @ 重设栈指针，指向 SDRAM 顶端(使用虚拟地址)
17     ldr pc, =0xB0004000        @ 跳到 SDRAM 中继续执行第二部分代码
18 halt_loop:
19      b   halt_loop
20
```

head.S 中调用的函数都在 init.c 中实现。

值得注意的是，在第 15 行开启 MMU 之后，无论是 CPU 取指还是 CPU 读写数据，使用的都是虚拟地址。

在第 14 行设置页表时，在 create_page_table 函数中令 head.S、init.c 程序所在内存的虚拟地址和物理地址一样，这使得 head.S 和 init.c 中的代码在开启 MMU 后能够没有任何障碍地继续运行。

（2）init.c 代码详解。

init.c 中的 disable_watch_dog、memsetup 函数实现的功能在前面两章已经讨论过，不再重复，下面列出代码以方便读者查阅。

```
001 /*
002  * init.c: 进行一些初始化，在 Steppingstone 中运行
003  * 它和 head.S 同属第一部分程序，此时 MMU 未开启，使用物理地址
004  */
005
006 /* WATCHDOG 寄存器 */
007 #define WTCON          (*(volatile unsigned long *)0x53000000)
008 /* 存储控制器的寄存器起始地址 */
009 #define MEM_CTL_BASE    0x48000000
010
011
012 /*
013  * 关闭 WATCHDOG，否则 CPU 会不断重启
014  */
015 void disable_watch_dog(void)
016 {
```

```
017     WTCON = 0;  // 关闭 WATCHDOG 很简单，往这个寄存器写 0 即可
018 }
019
020 /*
021  * 设置存储控制器以使用 SDRAM
022  */
023 void memsetup(void)
024 {
025     /* SDRAM 13 个寄存器的值 */
026     unsigned long  const   mem_cfg_val[]={ 0x22011110,      //BWSCON
027                                            0x00000700,      //BANKCON0
028                                            0x00000700,      //BANKCON1
029                                            0x00000700,      //BANKCON2
030                                            0x00000700,      //BANKCON3
031                                            0x00000700,      //BANKCON4
032                                            0x00000700,      //BANKCON5
033                                            0x00018005,      //BANKCON6
034                                            0x00018005,      //BANKCON7
035                                            0x008C07A3,      //REFRESH
036                                            0x000000B1,      //BANKSIZE
037                                            0x00000030,      //MRSRB6
038                                            0x00000030,      //MRSRB7
039                                   };
040     int     i = 0;
041     volatile unsigned long *p = (volatile unsigned long *)MEM_CTL_BASE;
042     for(; i < 13; i++)
043         p[i] = mem_cfg_val[i];
044 }
045
```

copy_2th_to_sdram 函数用来将第二部分代码（即由 leds.c 编译得来的代码）从 Steppingstone 中复制到 SDRAM 中。在连接程序时，第二部分代码的加载地址被指定为 2048，重定位地址为 0xB0004000。所以系统从 NAND Flash 启动后，第二部分代码就存储在 Steppingstone 中地址 2048 之后，需要把它复制到 0x30004000 处（此时尚未开启 MMU，虚拟地址 0xB0004000 对应的物理地址在后面设为 0x30004000）。Steppingstone 总大小为 4KB，不妨把地址 2048 之后的所有数据复制到 SDRAM 中，所以源数据的结束地址为 4096。copy_2th_to_sdram 函数的代码如下：

```
046 /*
047  * 将第二部分代码复制到 SDRAM
```

```
048 */
049 void copy_2th_to_sdram(void)
050 {
051     unsigned int *pdwSrc  = (unsigned int *)2048;
052     unsigned int *pdwDest = (unsigned int *)0x30004000;
053
054     while (pdwSrc < (unsigned int *)4096)
055     {
056         *pdwDest = *pdwSrc;
057         pdwDest++;
058         pdwSrc++;
059     }
060 }
061
```

剩下的 create_page_table、mmu_init 就是本章的重点了，前者用来设置页表，后者用来开启 MMU。

先看看 create_page_table 函数。它用于设置 3 个区域的地址映射关系。

（1）将虚拟地址 0～（1M-1）映射到同样的物理地址去，Steppingstone（从 0 地址开始的 4KB 内存）就处于这个范围中。使虚拟地址等于物理地址，可以让 Steppingstone 中的程序（head.s 和 init.c）在开启 MMU 前后不需要考虑太多的事情。

（2）GPIO 寄存器的起始物理地址范围为 0x56000000，将虚拟地址 0xA0000000～(0xA0000000+ 1M-1)映射到物理地址 0x56000000～(0x56000000+1M-1)。

（3）本开发板中 SDRAM 的物理地址范围为 0x30000000～0x33FFFFFF，将虚拟地址 0xB0000000～0xB3FFFFFF 映射到物理地址 0x30000000～0x33FFFFFF。

create_page_table 函数代码如下：

```
062 /*
063  * 设置页表
064  */
065 void create_page_table(void)
066 {
067
068 /*
069  * 用于段描述符的一些宏定义
070  */
071 #define MMU_FULL_ACCESS     (3 << 10)   /* 访问权限 */
072 #define MMU_DOMAIN          (0 << 5)    /* 属于哪个域 */
073 #define MMU_SPECIAL         (1 << 4)    /* 必须是 1 */
074 #define MMU_CACHEABLE       (1 << 3)    /* cacheable */
```

```
075 #define MMU_BUFFERABLE      (1 << 2)    /* bufferable */
076 #define MMU_SECTION         (2)         /* 表示这是段描述符 */
077 #define MMU_SECDESC         (MMU_FULL_ACCESS | MMU_DOMAIN | MMU_SPECIAL | \
078                             MMU_SECTION)
079 #define MMU_SECDESC_WB      (MMU_FULL_ACCESS | MMU_DOMAIN | MMU_SPECIAL | \
080                             MMU_CACHEABLE | MMU_BUFFERABLE | MMU_SECTION)
081 #define MMU_SECTION_SIZE   0x00100000
082
083     unsigned long virtuladdr, physicaladdr;
084     unsigned long *mmu_tlb_base = (unsigned long *)0x30000000;
085
086     /*
087      * Steppingstone 的起始物理地址为 0，第一部分程序的起始运行地址也是 0，
088      * 为了在开启 MMU 后仍能运行第一部分的程序，
089      * 将 0～1M 的虚拟地址映射到同样的物理地址
090      */
091     virtuladdr = 0;
092     physicaladdr = 0;
093     *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
094                                            MMU_SECDESC_WB;
095
096     /*
097      * 0x56000000 是 GPIO 寄存器的起始物理地址，
098      * GPBCON 和 GPBDAT 这两个寄存器的物理地址 0x56000010、0x56000014，
099      * 为了在第二部分程序中能以地址 0xA0000010、0xA0000014 来操作 GPBCON、GPBDAT，
100      * 把从 0xA0000000 开始的 1MB 虚拟地址空间映射到从 0x56000000 开始的 1MB 物理地址空间
101      */
102     virtuladdr = 0xA0000000;
103     physicaladdr = 0x56000000;
104     *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
105                                            MMU_SECDESC;
106
107     /*
108      * SDRAM 的物理地址范围是 0x30000000～0x33FFFFFF，
109      * 将虚拟地址 0xB0000000～0xB3FFFFFF 映射到物理地址 0x30000000～0x33FFFFFF 上，
110      * 总共 64MB，涉及 64 个段描述符
111      */
112     virtuladdr = 0xB0000000;
113     physicaladdr = 0x30000000;
```

```
114     while (virtuladdr < 0xB4000000)
115     {
116         *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
117                                                 MMU_SECDESC_WB;
118         virtuladdr += 0x100000;
119         physicaladdr += 0x100000;
120     }
121 }
122
```

mmu_tlb_base 被定义为 unsigned long 指针，所指向的内存为 4 字节，刚好是一个描述符的大小。在 SDRAM 的开始存放页表——第 81 行令 mmu_tlb_base 指向 SDRAM 的起始地址 0x30000000。其中最能体现页表结构的代码是第 93、104、116 行。

```
093     *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
094                                             MMU_SECDESC_WB;

104     *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
105                                             MMU_SECDESC;

116         *(mmu_tlb_base + (virtuladdr >> 20)) = (physicaladdr & 0xFFF00000) | \
117                                                 MMU_SECDESC_WB;
```

虚拟地址的位[31:20]用于索引一级页表，找到它所对应的描述符，对应于“(virtuladdr >> 20)”。

如图 7.13 所示，段描述符中位[31:20]中保存段的物理地址，对应于“(physicaladdr & 0xFFF00000)”。

位[11:0]中用来设置段的访问权限，包括所属的域、AP 位、C 位（是否可 Cache）、B 位（是否使用 Write buffer）——这对应“MMU_SECDESC”或“MMU_SECDESC_WB”，它们的域都被设为 0，AP 位被设为 0b11（根据表 7.2 可知它所在的域进行权限检查，则读写操作都被允许）。“MMU_SECDESC”中 C/B 位都没有设置，表示不使用 Cache 和 Write buffer，所以映射寄存器空间时使用“MMU_SECDESC”。“MMU_SECDESC_WB”中 C/B 位都设置了，表示使用 Cache 和 Write buffer，即所谓的写回式，在映射 Steppingstone 和 SDRAM 等内存时使用“MMU_SECDESC_WB”。

现在来看看 mmu_init 函数。create_page_table 函数设置好了页表，还需要把页表地址告诉 CPU，并且在开启 MMU 之前做好一些准备工作，比如使无效 ICaches、DCaches，设置域访问控制寄存器等。代码的注释就可以帮助读者很好地理解 mmu_init 函数，不再重复。代码如下：

```
123 /*
124  * 启动 MMU
125  */
```

```
126 void mmu_init(void)
127 {
128     unsigned long ttb = 0x30000000;
129
130 __asm__(
131     "mov    r0, #0\n"
132     "mcr    p15, 0, r0, c7, c7, 0\n"    /* 使无效 ICaches 和 DCaches */
133
134     "mcr    p15, 0, r0, c7, c10, 4\n"   /* drain write buffer on v4 */
135     "mcr    p15, 0, r0, c8, c7, 0\n"    /* 使无效指令、数据 TLB */
136
137     "mov    r4, %0\n"                   /* r4 = 页表基址 */
138     "mcr    p15, 0, r4, c2, c0, 0\n"    /* 设置页表基址寄存器 */
139
140     "mvn    r0, #0\n"
141     "mcr    p15, 0, r0, c3, c0, 0\n"    /* 域访问控制寄存器设为 0xFFFFFFFF,
142                                          * 不进行权限检查
143                                          */
144     /*
145      * 对于控制寄存器，先读出其值，在这基础上修改感兴趣的位，
146      * 然后再写入
147      */
148     "mrc    p15, 0, r0, c1, c0, 0\n"    /* 读出控制寄存器的值 */
149
150     /* 控制寄存器的低 16 位含义为: .RVI ..RS B... .CAM
151      * R : 表示换出 Cache 中的条目时使用的算法，
152      *     0 = Random replacement; 1 = Round robin replacement
153      * V : 表示异常向量表所在的位置，
154      *     0 = Low addresses = 0x00000000; 1 = High addresses = 0xFFFF0000
155      * I : 0 = 关闭 ICaches; 1 = 开启 ICaches
156      * R、S : 用来与页表中的描述符一起确定内存的访问权限
157      * B : 0 = CPU 为小字节序; 1 = CPU 为大字节序
158      * C : 0 = 关闭 DCaches; 1 = 开启 DCaches
159      * A : 0 = 数据访问时不进行地址对齐检查; 1 = 数据访问时进行地址对齐检查
160      * M : 0 = 关闭 MMU; 1 = 开启 MMU
161      */
162
163     /*
164      * 先清除不需要的位，往下若需要则重新设置它们
```

```
165     */
166                                                  /* .RVI ..RS B... .CAM */
167    "bic   r0, r0, #0x3000\n"                     /* ..11 ........... 清除V、I位 */
168    "bic   r0, r0, #0x0300\n"                     /* ....  ..11 .... .... 清除R、S位 */
169    "bic   r0, r0, #0x0087\n"                     /* ....  ....  1...  .111  清除
B/C/A/M */
170
171    /*
172     * 设置需要的位
173     */
174    "orr   r0, r0, #0x0002\n"                     /* .... .... .... ..1. 开启对齐检查 */
175    "orr   r0, r0, #0x0004\n"                     /* .... .... .... .1.. 开启DCaches */
176    "orr   r0, r0, #0x1000\n"                     /* ...1 .... .... .... 开启ICaches */
177    "orr   r0, r0, #0x0001\n"                     /* .... .... .... ...1 使能MMU */
178
179    "mcr   p15, 0, r0, c1, c0, 0\n"               /* 将修改的值写入控制寄存器 */
180    : /* 无输出 */
181    : "r" (ttb) );
182 }
```

mmu_init 函数在 C 语言中嵌入了汇编指令。

2. 第二部分代码分析

第二部分代码 leds.c 中只有两个函数：wait 和 main。wait 函数用来延迟时间，main 函数用来循环点亮 4 个 LED。与前面两章所用的 leds.c 有两点不同。

（1）操作 GPBCON、GPBDAT 两个寄存器时使用虚拟地址 0xA0000010、0xA0000014，在 init.c 中已经把虚拟地址 0xA0000000～(0xA0000000+ 1M–1)映射到物理地址 0x56000000～(0x56000000+1M–1)；

（2）在定义 wait 函数时使用了一点小技巧，将它定义成“static inline”类型，原因在源码的第 15 行给出。

leds.c 代码如下：

```
01 /*
02  * leds.c: 循环点亮 4 个 LED
03  * 属于第二部分程序，此时 MMU 已开启，使用虚拟地址
04  */
05
06 #define GPBCON      (*(volatile unsigned long *)0xA0000010)     // 物理地址
0x56000010
07 #define GPBDAT      (*(volatile unsigned long *)0xA0000014)     // 物理地址
```

```
0x56000014
08
09 #define GPB5_out    (1<<(5*2))
10 #define GPB6_out    (1<<(6*2))
11 #define GPB7_out    (1<<(7*2))
12 #define GPB8_out    (1<<(8*2))
13
14 /*
15  * wait 函数加上 "static inline" 是有原因的,
16  * 这样可以使得编译 leds.c 时, wait 嵌入 main 中, 编译结果中只有 main 一个函数。
17  * 于是在连接时, main 函数的地址就是由连接文件指定的运行地址。
18  * 而连接文件 mmu.lds 中, 指定了 leds.o 的运行时装载地址为 0xB0004000,
19  * 这样, head.S 中的 "ldr pc, =0xB0004000" 就是跳去执行 main 函数。
20  */
21 static inline void wait(unsigned long dly)
22 {
23     for(; dly > 0; dly--);
24 }
25
26 int main(void)
27 {
28     unsigned long i = 0;
29
30     // 将 LED1NLED4 对应的 GPB5GPB8 引脚设为输出
31     GPBCON = GPB5_out|GPB6_out|GPB7_out|GPB8_out;
32
33     while(1){
34         wait(3000000);
35         GPBDAT = (~(i<<5));     // 根据 i 的值, 点亮 LED1~LED4
36         if(++i == 16)
37             i = 0;
38     }
39
40     return 0;
41 }
42
```

3. Makefile 和连接脚本 mmu.lds

Makefile 内容如下:

```
01 objs := head.o init.o leds.o
02
03 mmu.bin: $(objs)
04        arm-linux-ld -Tmmu.lds -o mmu_elf $^
05        arm-linux-objcopy -O binary -S mmu_elf $@
06        arm-linux-objdump -D -m arm mmu_elf > mmu.dis
07
08 %.o:%.c
09        arm-linux-gcc -Wall -O2 -c -o $@ $<
10
11 %.o:%.S
12        arm-linux-gcc -Wall -O2 -c -o $@ $<
13
14 clean:
15        rm -f mmu.bin mmu_elf mmu.dis *.o
16
```

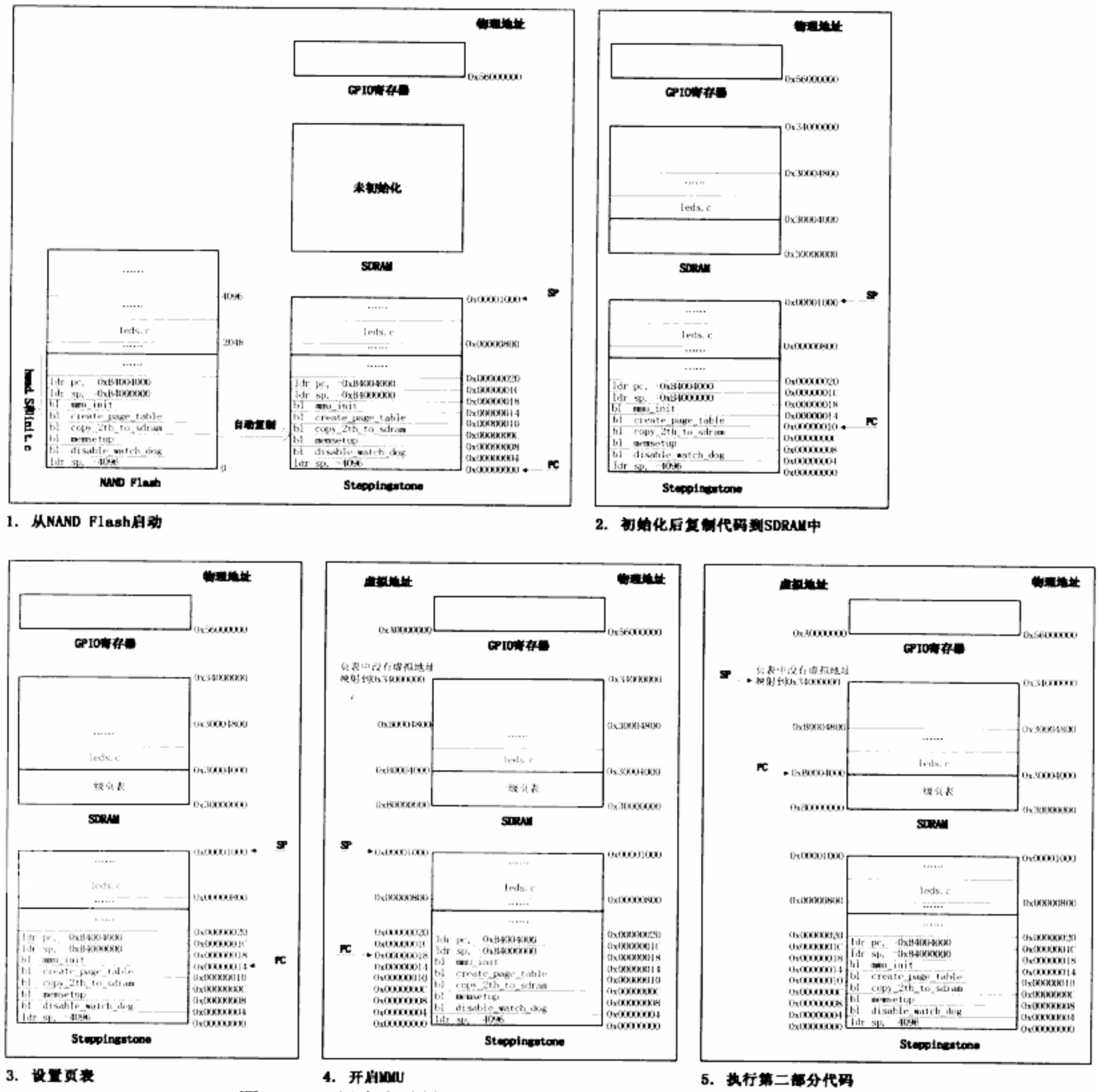
在源代码目录下执行 make 命令时，make 命令读取 Makefile 文件发现目标文件 mmu.bin 不存在，所以试图使用它的依赖文件 head.o、init.o、leds.o 来生成 mmu.bin；可是，这些依赖文件也不存在，于是先使用其他规则来生成这些文件：使用第 11、12 行的规则来编译 head.S 以生成 head.o，使用第 8、9 行的规则来编译 init.c 以生成 init.o、编译 leds.c 以生成 leds.o；最后，调用第 4、5、6 行的命令来生成 mmu.bin。

Makefile 中第 4 行命令用来连接程序，它使用连接脚本 mmu.lds 来控制连接器的行为。文件 mmu.lds 内容如下：

```
01 SECTIONS {
02    firtst    0x00000000 : { head.o init.o }
03    second    0xB0004000 : AT(2048) { leds.o }
04 }
05
```

连接脚本 mmu.lds 将程序分为两个段：first 和 second。前者由 head.o 和 init.o 组成，它的加载地址和运行地址都是 0，所以运行前不需要重新移动代码。后者由 leds.o 组成，它的加载地址为 2048，重定位地址为 0xB0004000，这表明段 second 存放在编译所得映象文件地址 2048 处，在运行前需要将它复制到地址 0xB0004000 处，这由 init.c 中的 copy_2th_to_sdram 函数完成（注意，此函数将代码复制开始地址为 0x30004000 的内存中，这是开启 MMU 后虚拟地址 0xB0004000 对应的物理地址）。

本实例程序中涉及代码的复制、开启 MMU 前使用物理地址寻址，开启 MMU 后使用虚拟地址寻址，相对复杂。为了更形象地讲解本程序，下面用图 7.16 来演示代码的执行过程。



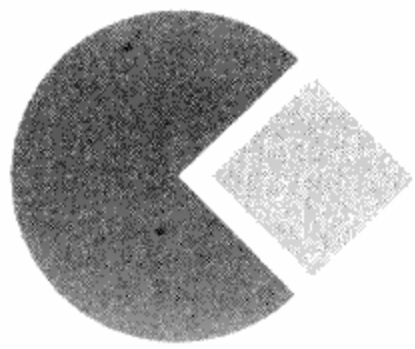
1. 从NAND Flash启动　　2. 初始化后复制代码到SDRAM中

3. 设置页表　　4. 开启MMU　　5. 执行第二部分代码

图 7.16　程序复制代码、设置页表、启动 MMU 的执行过程

7.2.3　实例测试

在源码目录下执行 make 命令生成可执行程序 mmu.bin，使用 JTAG 工具烧入 NAND Flash 后，按复位键启动系统。可以看见 4 个 LED 被轮流点亮，用来从 0～15 重复计数。LED 闪烁的速度比第 6 章的 SDRAM 实验快，这是因为开启了 Cache。



第 8 章　NAND Flash 控制器

本章目标

- 了解 NAND Flash 芯片的接口
- 掌握通过 NAND Flash 控制器访问 NAND Flash 的方法

8.1　NAND Flash 介绍和 NAND Flash 控制器使用

NAND Flash 在嵌入式系统中的地位与 PC 上的硬盘类似，用于保存系统运行所必需的操作系统、应用程序、用户数据、运行过程中产生的各类数据。与内存掉电后数据丢失不同，NAND Flash 中的数据在掉电后仍可永久保存。

8.1.1　Flash 介绍

常用的 Flash 类型有 NOR Flash 和 NAND Flash 两种。NOR Flash 由 Intel 公司在 1988 年发明，以替代当时在市场上占据主要地位的 EPROM 和 E^2PROM。NAND Flash 由 Toshiba 公司在 1989 年发明。两者的主要差别如表 8.1 所示。

表 8.1　NOR/NAND Flash 的差别

		NOR	NAND
容量		1MB～32MB	16MB～512MB
XIP		Yes	No
性能	擦除	非常慢（5s）	快（3ms）
	写	慢	快
	读	快	快
可靠性		比较高，位反转的比例小于 NAND Flash 的 10%	比较低，位反转比较常见，必需有校验措施，比如TNR必须有坏块管理措施
可擦除次数		10000～100000	100000～1000000
生命周期		低于 NAND Flash 的 10%	是 NOR Flash 的 10 倍以上

续表

	NOR	NAND
接口	与 RAM 接口相同	I/O 接口
访问方法	随机访问	顺序访问
易用性	容易	复杂
主要用途	常用于保存代码和关键数据	用于保存数据
价格	高	低

NOR Flash 支持 XIP，即代码可以直接在 NOR Flash 上执行，无需复制到内存中。这是由于 NOR Flash 的接口与 RAM 完全相同，可以随机访问任意地址的数据。在 NOR Flash 上进行读操作的效率非常高，但是擦除和写操作的效率很低；另外，NOR Flash 的容量一般比较小。NAND Flash 进行擦除和写操作的效率更高，并且容量更大。一般而言，NOR Flash 用于存储程序，NAND Flash 用于存储数据。基于 NAND Flash 的设备通常也要搭配 NOR Flash 以存储程序。

Flash 存储器件由擦除单元（也称为块）组成，当要写某个块时，需要确保这个块已经被擦除。NOR Flash 的块大小范围为 64kB～128kB；NAND Flash 的块大小范围为 8kB～64kB，擦/写一个 NOR Flash 块需 4s，而擦/写一个 NAND Flash 块仅需 2ms。NOR Flash 的块太大，不仅增加了擦写时间，对于给定的写操作，NOR Flash 也需要更多的擦除操作——特别是小文件，比如一个文件只有 1kB，但是为了保存它却需要擦除大小为 64kB～128kB 的 NOR Flash 块。

NOR Flash 的接口与 RAM 完全相同，可以随意访问任意地址的数据。而 NAND Flash 的接口仅仅包含几个 I/O 引脚，需要串行地访问。NAND Flash 一般以 512 字节为单位进行读写。这使得 NOR Flash 适合于运行程序，而 NAND Flash 更适合于存储数据。

容量相同的情况下，NAND Flash 的体积更小，对于空间有严格要求的系统，NAND Flash 可以节省更多空间。市场上 NOR Flash 的容量通常为 1MB～4MB（也有 32MB 的 NOR Flash），NAND Flash 的容量为 8MB～512MB。容量的差别也使得 NOR Flash 多用于存储程序，NAND Flash 多用于存储数据。

对于 Flash 存储器件的可靠性需要考虑 3 点：位反转、坏块和可擦除次数。所有 Flash 器件都遭遇位反转的问题：由于 Flash 固有的电器特性，在读写数据过程中，偶然会产生一位或几位数据错误（这种概率很低），而 NAND Flash 出现的概率远大于 NOR Flash。当位反转发生在关键的代码、数据上时，有可能导致系统崩溃。当仅仅是报告位反转，重新读取即可；如果确实发生了位反转，则必须有相应的错误检测/恢复措施。在 NAND Flash 上发生位反转的概率更高，推荐使用 EDC/ECC 进行错误检测和恢复。NAND Flash 上面会有坏块随机分布，在使用前需要将坏块扫描出来，确保不再使用它们，否则会使产品含有严重的故障。NAND Flash 每块的可擦除次数通常在 100000 次左右，是 NOR Flash 的 10 倍。另外，因为 NAND Flash 的块大小通常是 NOR Flash 的 1/8，所以 NAND Flash 的寿命远远超过 NOR Flash。

嵌入式 Linux 对 NOR、NAND Flash 的软件支持都很成熟。在 NOR Flash 上常用 jffs2 文件系统，而在 NAND Flash 上常用 yaffs 文件系统。在更底层，有 MTD 驱动程序实现对它们的读、写、擦除操作，它也实现了 EDC/ECC 校验。

8.1.2 NAND Flash 的物理结构

以 NAND Flash K9F1208U0M 为例，K9F1208U0M 是三星公司生产的容量为 64MB 的 NAND Flash，常用于手持设备等消费电子产品。它的封装及外部引脚如图 8.1 所示。

引脚名称	引脚功能
I/O0 ~ I/O7	数据输入/输出
CLE	命令锁存使能
ALE	地址锁存使能
$\overline{CE}$	芯片使能
$\overline{RE}$	读使能
$\overline{WE}$	写使能
$\overline{WP}$	写保护
R/$\overline{B}$	就绪/忙输出信号
Vcc	电源
Vss	地
N.C	不接

图 8.1 NAND Flash 封装及引脚

K9F1208U0M 的功能结构图如图 8.2 所示。

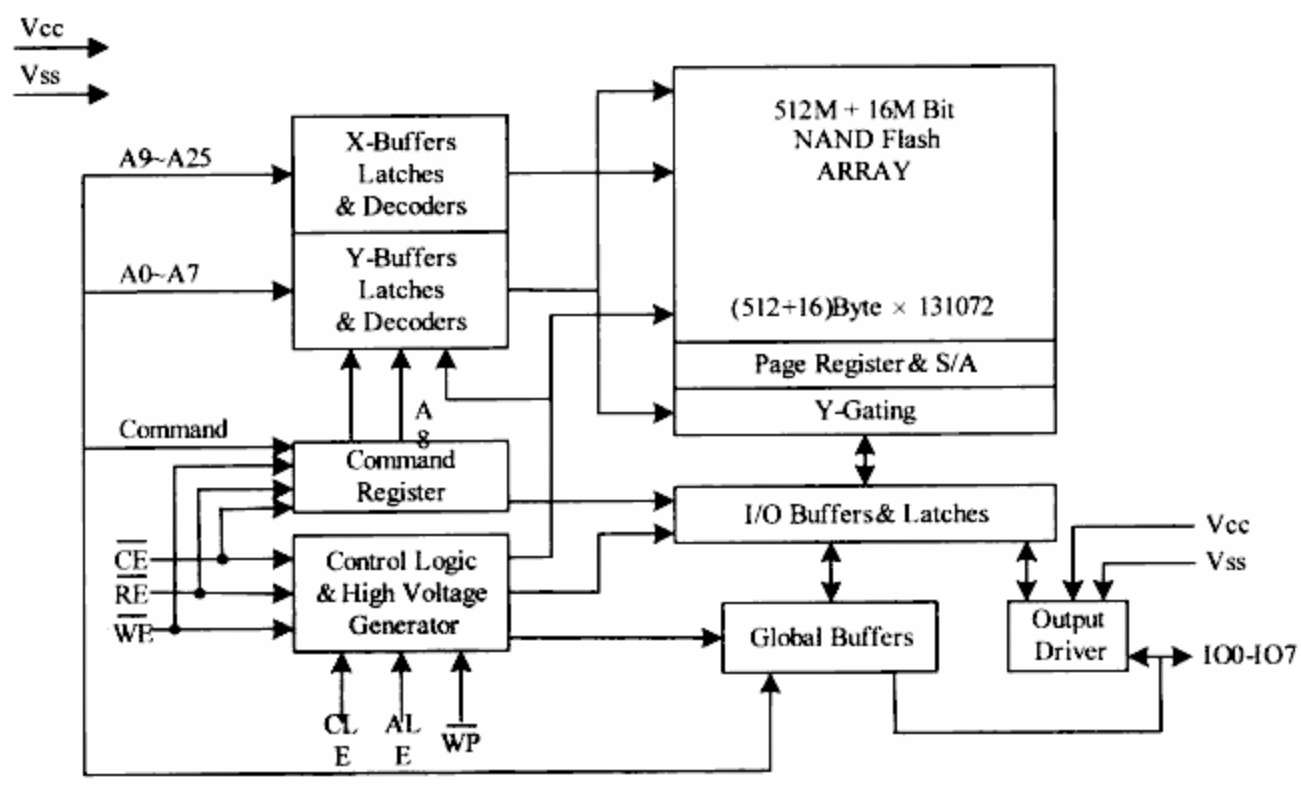


图 8.2 NAND Flash K9F1208U0M 功能结构

K9F1208U0M 的内部结构分为 10 个功能部件。

① X-Buffers Latche & Decoders：用于行地址。

② Y-Buffers Latche & Decoders：用于列地址。

③ Command Register：用于命令字。

④ Control Logic & High Voltage Generator：控制逻辑及产生 Flash 所需高压。

⑤ Nand Flash Array：存储部件。

⑥ Page Register & S/A：页寄存器，当读、写某页时，会将数据先读入/写入此寄存器，

大小为 528 字节。

⑦ Y-Gating。

⑧ I/O Buffers & Latches。

⑨ Global Buffers。

⑩ Output Driver。

NAND Flash 存储单元组织结构如图 8.3 所示。

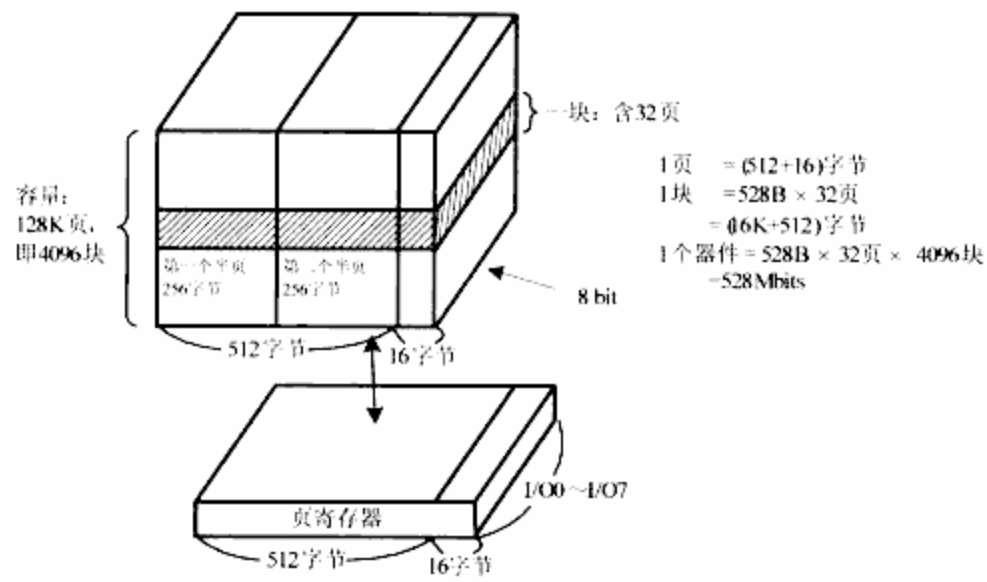


图 8.3 NAND Flash K9F1208U0M 存储单元组织结构

K9F1208U0M 容量为 528Mbit，分为 131072 行（页）、528 列；每一页大小为 512 字节，外加 16 字节的额外空间，这 16 字节额外空间的列地址为 512～527。

命令、地址、数据都通过 8 个 I/O 口输入/输出，这种形式减少了芯片的引脚个数，并使得系统很容易升级到更大的容量。写入命令、地址或数据时，都需要将 WE#、CE#信号同时拉低。数据在 WE#信号的上升沿被 NAND Flash 锁存；命令锁存信号 CLE、地址锁存信号 ALE 用来分辨、锁存命令或地址。K9F1208U0M 的 64MB 存储空间需要 26 位地址，因此以字节为单位访问 Flash 时需要 4 个地址序列：列地址、行地址的低位部分、行地址的高位部分。读/写页在发出命令后，需要 4 个地址序列，而擦除块在发出擦除命令后仅需要 3 个地址序列。

8.1.3 NAND Flash 访问方法

1. 硬件连接

NAND Flash 与 S3C2410/S3C2440 的硬件连接如图 8.4 所示。

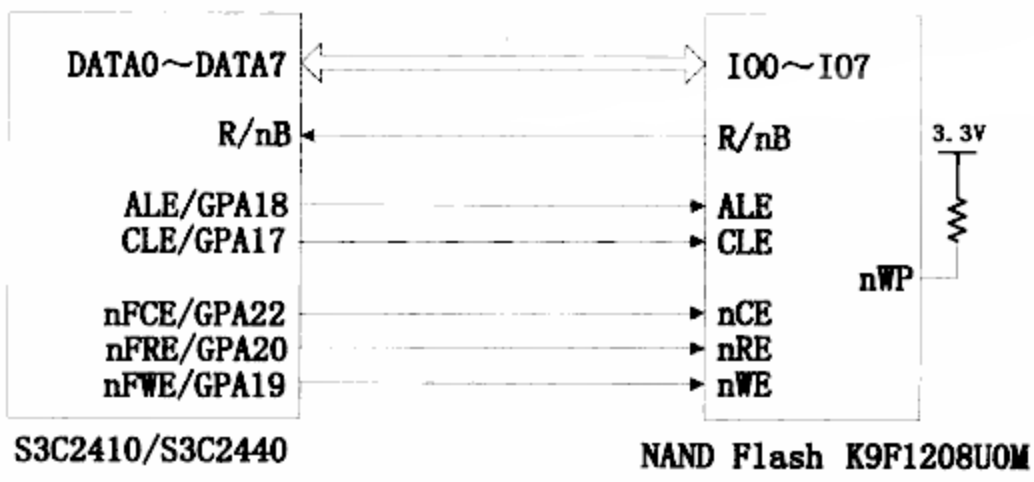


图 8.4 NAND Flash 与 S3C2410/S3C2440 的硬件连线

NAND Flash 与 S3C2410/SC32440 的连线比较少：8 个 I/O 引脚（IO0～IO7）、5 个使能信号（nWE、ALE、CLE、nCE、nRE）、1 个状态引脚（RDY/B）、1 个写保护引脚（nWP）。地址、数据和命令都是在这些使能信号的配合下，通过 8 个 I/O 引脚传输。写地址、数据、命令时，nCE、nWE 信号必须为低电平，它们在 nWE 信号的上升沿被锁存。命令锁存使能信号 CLE 和地址锁存信号 ALE 用来区分 I/O 引脚上传输的是命令还是地址。

2. 命令字及操作方法

操作 NAND Flash 时，先传输命令，然后传输地址，最后读/写数据，期间要检查 Flash 的状态。对于 K9F1208U0M，它的容量为 64MB，需要一个 26 位的地址。发出命令后，后面要紧跟着 4 个地址序列。比如读 Flash 时，发出读命令和 4 个地址序列后，后续的读操作就可以得到这个地址及其后续地址的数据。相应的命令字和地址序列如表 8.2、8.3 所示。

表 8.2　NAND Flash K9F1208U0M 的命令字

命　令	第 1 个访问周期	第 2 个访问周期	第 3 个访问周期
Read 1（读）	00h/01h	-	-
Read 2（读）	50h	-	-
Read ID（读芯片 ID）	90h	-	-
Page Program（写页）	80h	10h	-
Block Erase（擦除块）	60h	D0h	-
Read Status（读状态）	70h	-	-
Read Multi-Plane Status（读多层的状态）	71h	-	-
Reset（复位）	FFh	-	-
Page Program（Dummy）	80h	11h	-
Copy-Back Program（True）	00h	8Ah	10h
Copy-Back Program（Dummy）	03h	8Ah	11h
Multi-Plane Block Erase	60h-60h	D0h	-

表 8.3　NAND Flash K9F1208U0M 的地址序列

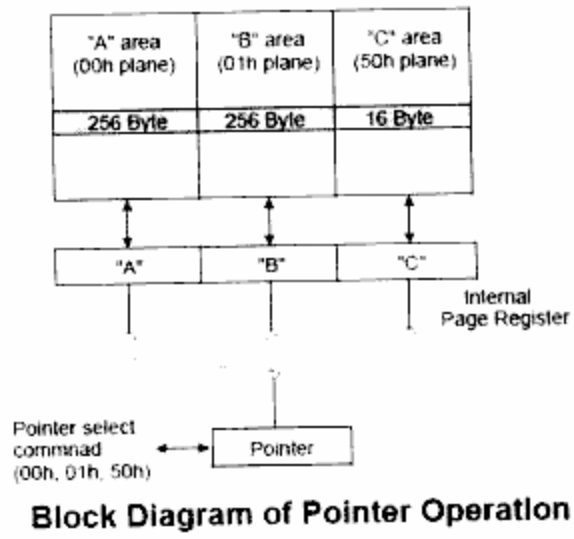
	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	备　注
第 1 个地址序列	A0	A1	A2	A3	A4	A5	A6	A7	列地址
第 2 个地址序列	A9	A10	A11	A12	A13	A14	A15	A16	行地址 即：页地址
第 3 个地址序列	A17	A18	A19	A20	A21	A22	A23	A24	
第 4 个地址序列	A25	L	L	L	L	L	L	L	

注：① K9F1208U0M 一页大小为 512 字节，分为两部分：上半部、下半部。
② 列地址用来在半页（256 字节）中寻址。
③ 当发出读命令 00h 时，表示列地址将在上半部寻址。
当发出读命令 01h 时，表示列地址将在下半部寻址。
④ A8 被读命令 00h 设为低电平，被 01h 设为高电平。
⑤ L 表示低电平。

K9F1208U0M 一页大小为 528 字节，而列地址 A0～A7 可以寻址的范围是 256 字节，所以必须辅以其他手段才能完全寻址这 528 字节。将一页分为 A、B、C 三个区：A 区为 0～255 字节，B 区为 256～511 字节，C 区为 512～527 字节。访问某页时，需要选定特定的区，这称为“使地址指针指向特定的区”。这通过 3 个命令来实现：命令 00h 让地址指针指向 A 区、命令 01h 让地址指针指向 B 区、命令 50h 让地址指针指向 C 区。命令 00h 和 50h 会使得访问 Flash 的地址指针一直从 A 区或 C 区开始，除非发出了其他的修改地址指针的命令。命令 01h 的效果只能维持一次，当前的读、写、擦除、复位或者上电操作完成后，地址指针重新指向 A 区。写 A 区或 C 区的数据时，必须在发出命令 80h 之前发出命令 00h 或者 50h；写 B 区的数据时，发出命令 01h 后必须紧接着就发出命令 80h。图 8.5 形象地表示了 K9F1208U0M 的这个特性。

Destination of the pointer

Command	Pointer position	Area
00h	0 - 255 byte	1st half array(A)
01h	256 - 511 byte	2nd half array(B)
50h	512 - 527 byte	spare array(C)



Block Diagram of Pointer Operation

图 8.5 SAMSUNG NAND Flash 地址指针操作

下面逐个讲解表 8.2 的命令字。

（1）Read 1：命令字为 00h 或 01h。

如图 8.5 所示，发出命令 00h 或 01h 后，就选定了读操作是从 A 区还是 B 区开始。从表 8.3 可知，列地址 A0～A7 可以寻址的范围是 256 字节，命令 00h 和 01h 使得可以在 512 字节大小的页内任意寻址——这相当于 A8 被命令 00h 设为 0，而被命令 01h 设为 1。

发出命令字后，依据表 8.3 发出 4 个地址序列，然后就可以检测 R/nB 引脚以确定 Flash 是否准备好。如果准备好了，就可以发起读操作依次读入数据。

（2）Read 2：命令字为 50h。

与 Read 1 命令类似，不过读取的是 C 区数据，操作序列为：发出命令字 50h、发出 4 个地址序列、等待 R/nB 引脚为高，最后读取数据。不同的是，地址序列中 A0～A3 用于设定 C 区（大小为 16 字节）要读取的起始地址，A4～A7 被忽略。

（3）Read ID：命令字为 90h。

发出命令字 90h，发出 4 个地址序列（都设为 0），然后就可以连续读入 5 个数据，分别表示：厂商代码（对于 SAMSUNG 公司为 Ech）、设备代码（对于 K9F1208U0M 为 76h）、保留的字节（对于 K9F1208U0M 为 A5h）、多层操作代码（C0h 表示支持多层操作）。

（4）Reset：命令字为 FFh。

发出命令字 FFh 即可复位 NAND Flash 芯片。如果芯片正处于读、写、擦除状态，复位命令会终止这些命令。

（5）Page Program(True)：命令字分两阶段，80h 和 10h。

它的操作序列如图 8.6 所示。

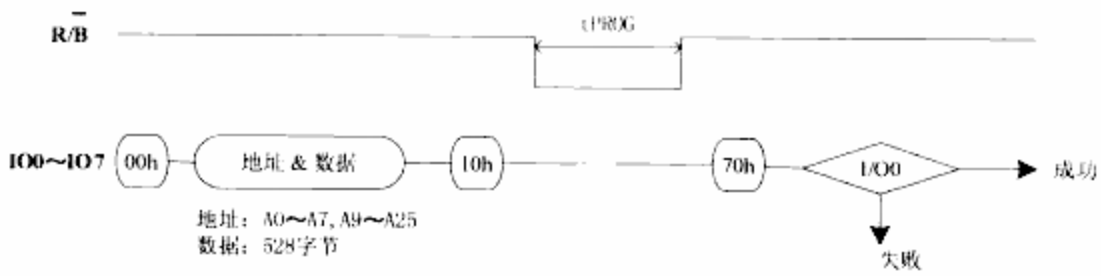


图 8.6 NAND Flash 写操作

NAND Flash 的写操作一般是以页为单位的，但是可以只写一页中的一部分。发出命令字 80h 后，紧接着是 4 个地址序列，然后向 Flash 发送数据（最大可以达到 528 字节），然后发出命令字 10h 启动写操作，此时 Flash 内部会自动完成写、校验操作。一旦发出命令字 10h 后，就可以通过读状态命令 70h 获知当前写操作是否完成、是否成功。

（6）Page Program(Dummy)：命令字分两阶段，80h 和 11h。

NAND Flash K9F1208U0M 分为 4 个 128Mbit 的存储层（plane），每个存储层包含 1024 个 block 和 528 字节的寄存器。这使得可以同时写多个页(page)或者同时擦除多个块(block)。块的地址经过精心安排，可以在 4 个连续的块内同时进行写或者擦除操作。

如图 8.7 所示为 K9F1208U0M 的块组织图。

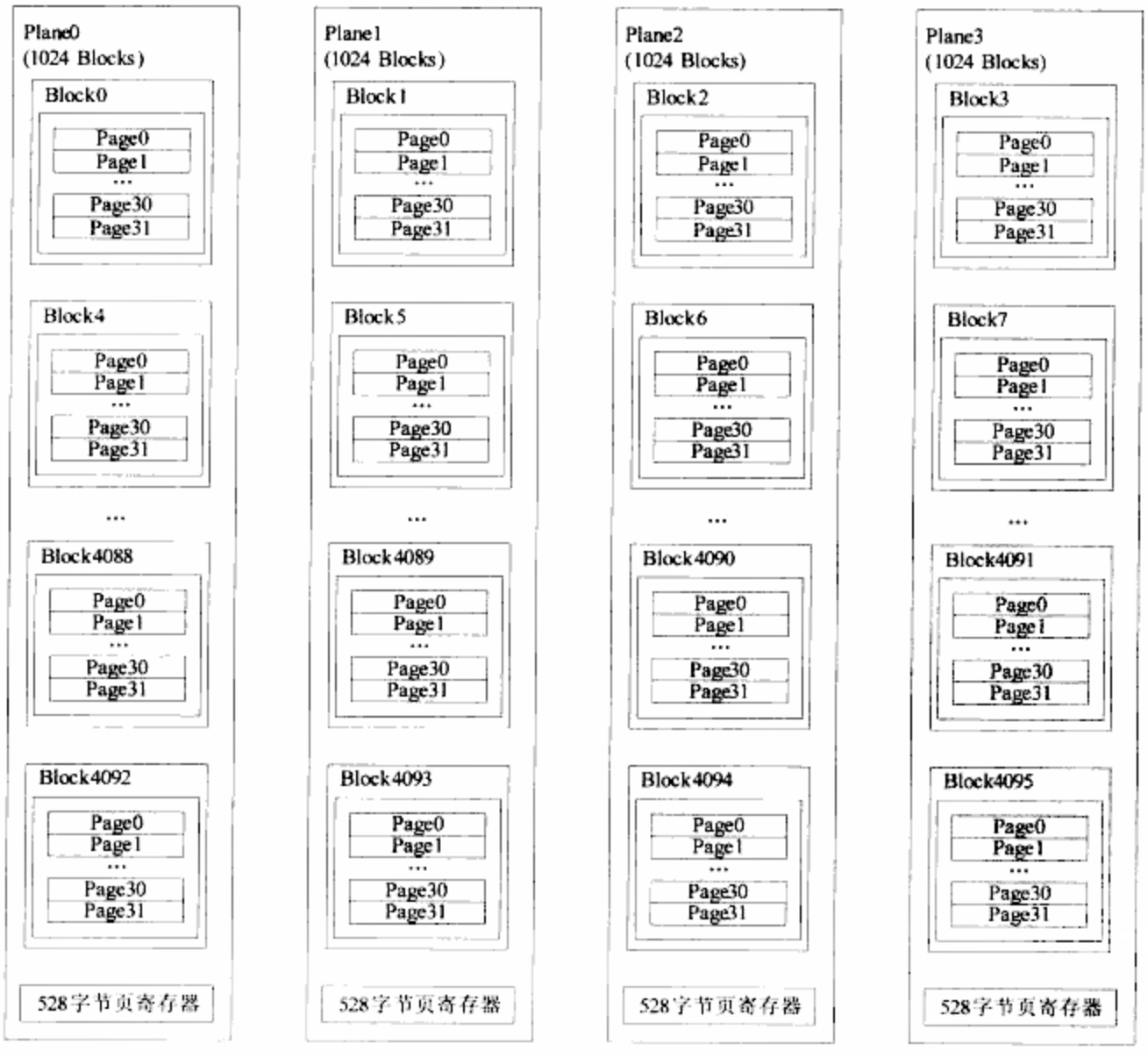


图 8.7 K9F1208U0M 的 block 组织结构

命令 Page Program(Dummy)正是在这种结构下对命令 Page Program(True)的扩展，后者仅能对一页进行写操作，前者可以同时写 4 页。命令 Page Program(Dummy)的操作序列如图 8.8 所示。

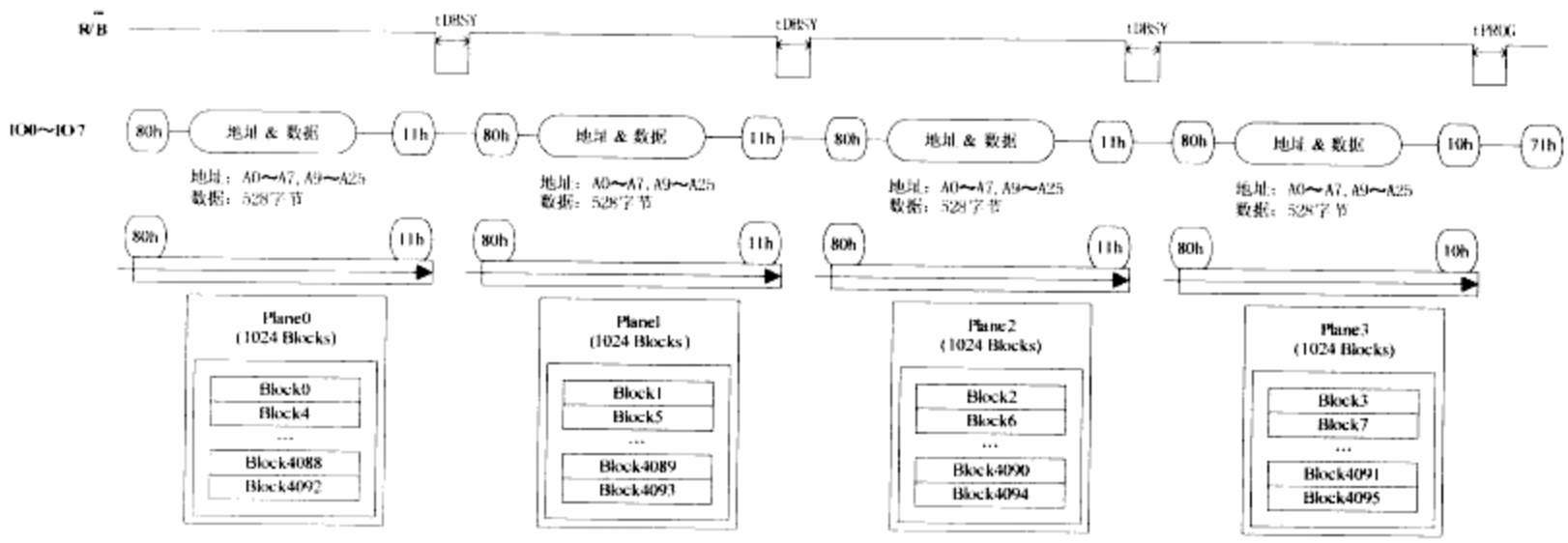


图 8.8 Four-Plane Page Program

发出命令字 80h、4 个地址序列及最多 528 字节的数据之后，发出命令字 11h（11h 称为“Dummy Page Program command”，相对地，10h 称为“True Page Program Command”）；接着对相邻层（plane）上的页进行同样的操作；仅在第 4 页的最后使用 10h 替代 11h，这样即可启动 Flash 内部的写操作。此时可以通过命令 71h 获知这些写操作是否完成、是否成功。

（7）Copy-Back Program(True)：命令字分 3 阶段，00h、8Ah、10h。

此命令用于将一页复制到同一层（plane）内的另一页，它省略了读出源数据、将数据重新载入 Flash，这使得效率大为提高。此命令有两个限制：源页、目的页必须在同一个层（plane）中，并且源地址、目的地址的 A14、A15 必须相同。

操作序列如图 8.9 所示。

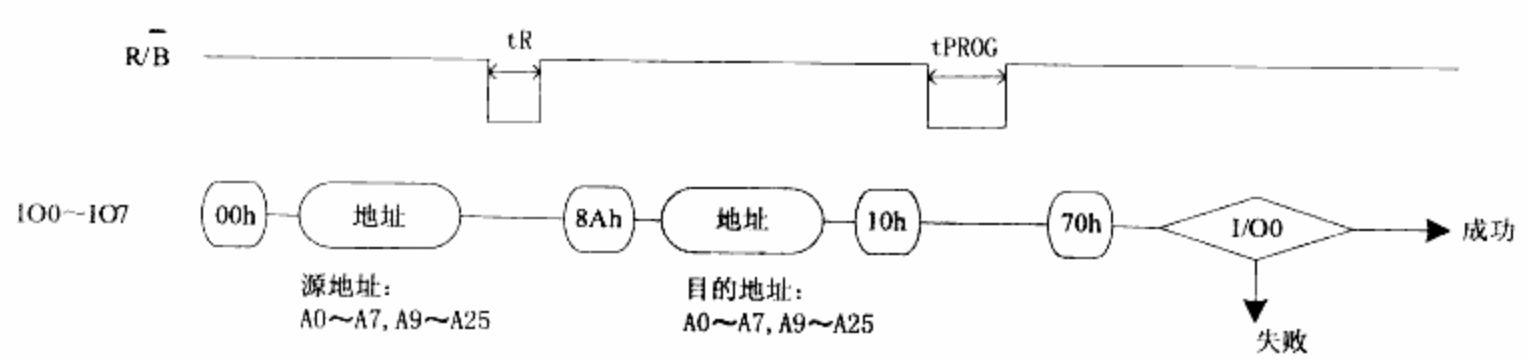


图 8.9 One Page Copy-Back program Operation

首先发出命令 Read 1(00h)、4 个源地址序列，此时源页的 528 字节数据很快就被全部读入内部寄存器中；接着发出命令字 8Ah(Page-Copy Data-input command)，随之发出 4 个目的地址序列；最后发出命令字 10h 启动对目的页的写操作。此后可以使用命令 70h 来查看此操作是否完成、是否成功。

（8）Copy-Back Program(Dummy)：命令字分 3 阶段，03h、8Ah、11h。

与命令 Page Program(Dummy)类似，Copy-Back Program(Dummy)可以同时启动对多达 4

个连续 plane 内的 Copy-Back Program 操作。操作序列如图 8.10 所示。

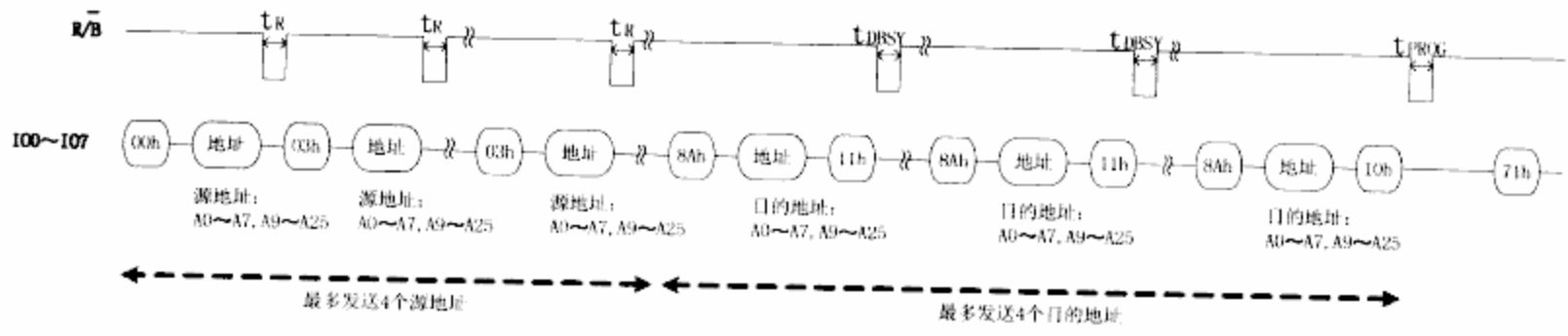


图 8.10 Four-Plane Copy-Back Page Program

从图 8.10 可知，首先发出命令字 00h、源页地址，这使得源页的 528 字节数据被读入所在 plane 的寄存器；对于随后的其他 plane 的源页，发出命令字 03h 和相应的源页地址将数据读入该 plane 的寄存器；按照前述说明读出最多 4 页的数据到寄存器后，发出命令字 8Ah、目的地址、命令字 11h，在发出最后一页的地址后，用 10h 代替 11h 以启动写操作。

（9）Block Erase：命令字分 3 阶段，60h、D0h。

此命令用于擦除 NAND Flash 块（block，大小为 16KB）。发出命令字 60h 之后，发出 block 地址——仅需要 3 个地址序列（请参考表 8.3，仅需要发出 2、3、4 cycle 所示地址），并且 A9～A13 被忽略。操作序列如图 8.11 所示。

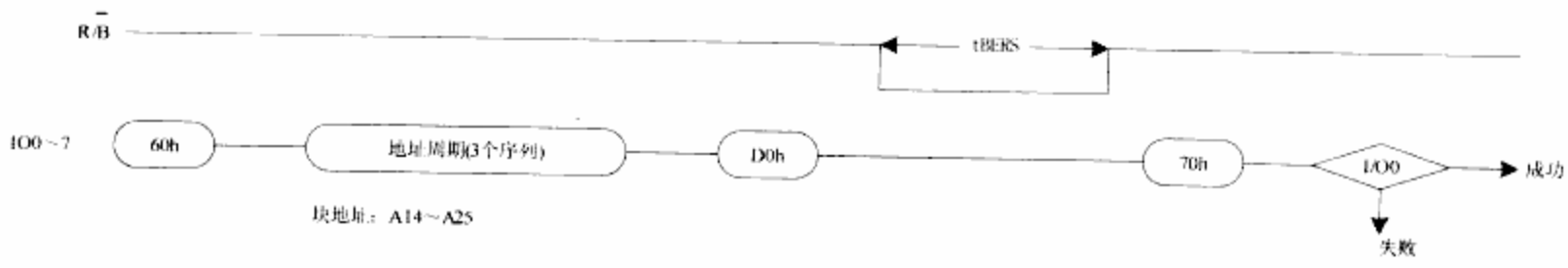


图 8.11 Block Erase Operation

（10）Multi-Plane Block Erase：60h----60h D0h

此命令用于同时擦除不同 plane 中的块。发出命令字 60h 之后，紧接着发出 block 地址序列，如此最多可以发出 4 个 block 地址，最后发出命令字 D0h 启动擦除操作。操作序列如图 8.12 所示。

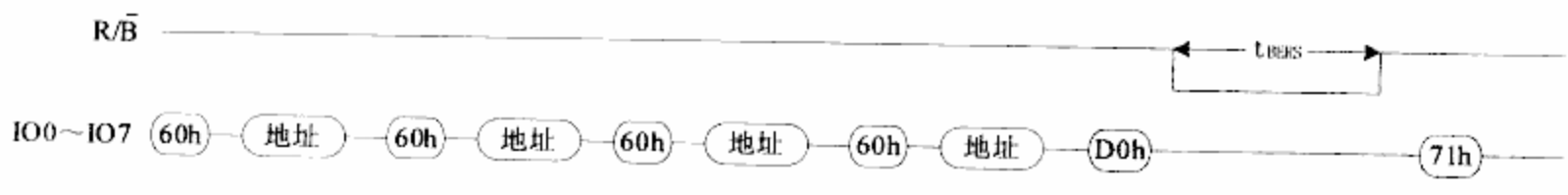


图 8.12 Four-Plane Block Erase Operation

（11）读状态命令有以下两种：

① Read Status：命令字为 70h。

② Read Multi-Plane Status：命令字为 71h。

Flash 中有状态寄存器，发出命令字 70h 或者 71h 之后，启动读操作即可读入此寄存器。状态寄存器中各位的含义如表 8.4 所示。

表 8.4 NAND Flash 读状态寄存器时各数据位的定义

I/O 引脚	所标识的状态	命令 70h 对应的定义	命令 71h 对应的定义
I/O0	总标记：成功/失败	成功：0，失败：1	成功：0，失败：1
I/O1	Plane0 的标记：成功/失败	忽略	成功：0，失败：1
I/O2	Plane1 的标记：成功/失败	忽略	成功：0，失败：1
I/O3	Plane2 的标记：成功/失败	忽略	成功：0，失败：1
I/O4	Plane3 的标记：成功/失败	忽略	成功：0，失败：1
I/O5	保留	忽略	忽略
I/O6	设备状态	忙：0，就绪：1	成功：0，失败：1
I/O7	写保护状态	保护：0，没有保护：1	保护：0，没有保护：1

注：① I/O0 是所有 Plane 的“总标记”，只要有一个 Plane 的操作是失败的，I/O0 就会被设为“失败”。
② I/O0～I/O4 引脚只标记它对应的 Plane。

8.1.4 S3C2410/S3C2440 NAND Flash 控制器介绍

NAND Flash 控制器提供几个寄存器来简化对 NAND Flash 的操作。比如要发出读命令时，只需要往 NFCMD 寄存器中写入 0 即可，NAND Flash 控制器会自动发出各种控制信号。

1. 操作方法概述

访问 NAND Flash 时需要先发出命令，然后发出地址序列，最后读/写数据；需要使用各个使能信号来分辨是命令、地址还是数据。S3C2410 的 NAND Flash 控制器提供了 NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT 和 NFECC 等 6 个寄存器来简化这些操作。S3C2440 的 NAND Flash 控制器则提供了 NFCONF、NFCONT、NFCMMD、NFADDR、NFDATA、NFSTAT 和其他与 ECC 有关的寄存器。对 NAND Flash 控制器的操作，S3C2410 和 S3C2440 有一点小差别：有些寄存器的地址不一样，有些寄存器的内容不一样，这在实例程序中会体现出来。

NAND Flash 的读写操作次序如下。

① 设置 NFCONF（对于 S3C2440，还要设置 NFCONT)寄存器，配置 NAND Flash。

② 向 NFCMD 寄存器写入命令，这些命令字可参考表 8.2。

③ 向 NFADDR 寄存器写入地址。

④ 读/写数据：通过寄存器 NFSTAT 检测 NAND Flash 的状态，在启动某个操作后，应该检测 R/nB 信号以确定该操作是否完成、是否成功。

2. 寄存器介绍

下面讲解这些寄存器的功能及具体用法。

（1）NFCONF：NAND Flash 配置寄存器。

这个寄存器在 S3C2410、S3C2440 上功能有所不同。

① S3C2410 的 NFCONF 寄存器。

被用来使能/禁止 NAND Flash 控制器、使能/禁止控制引脚信号 nFCE、初始化 ECC、设置 NAND Flash 的时序参数等。

TACLS、TWRPH0 和 TWRPH1 这 3 个参数控制的是 NAND Flash 信号线 CLE/ALE 与写控制信号 nWE 的时序关系，如图 8.13 所示。

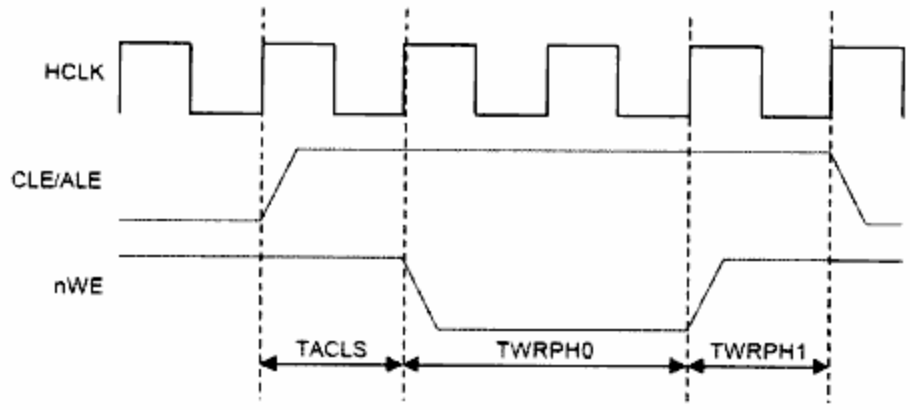


图 8.13 NAND FLASH MEMORY TIMING（TACLS = 0, TWRPH0 = 1, TWRPH1 = 0）

② S3C2440 的 NFCONF 寄存器。

被用来设置 NAND Flash 的时序参数 TACLS、TWRPH0、TWRPH1，设置数据位宽；还有一些只读位，用来指示是否支持其他大小的页（比如一页大小为 256/512/1024/2048 字节）。

它没有实现 S3C2410 的 NFCONF 寄存器的控制功能，这些功能在 S3C2440 的 NFCONT 寄存器里实现。

（2）NFCONT：NAND Flash 控制寄存器，S4C2410 没有这个寄存器。

被用来使能/禁止 NAND Flash 控制器、使能/禁止控制引脚信号 nFCE、初始化 ECC。它还有其他功能，在一般的应用中用不到，比如锁定 NAND Flash。

（3）NFCMD：NAND Flash 命令寄存器。

对于不同型号的 Flash，操作命令一般不一样。对于本板使用的 K9F1208U0M，请参考表 8.2。

（4）NFADDR：NAND Flash 地址寄存器。

当写这个寄存器时，它将对 Flash 发出地址信号。

（5）NFDATA：NAND Flash 数据寄存器。

只用到低 8 位，读、写此寄存器将启动对 NAND Flash 的读数据、写数据操作。

（6）NFSTAT：NAND Flash 状态寄存器。

只用到位 0，0:busy，1:ready。

8.2 NAND Flash 控制器操作实例：读 Flash

本实例讲述如何读取 NAND Flash，擦除、写 Flash 的操作与读 Flash 类似，读者可以自行编写程序。

8.2.1 读 NAND Flash 的步骤

下面讲述如何从 NAND Flash 中读出数据，假设读地址为 addr。

1. 设置 NFCONF（对于 S3C2440，还要设置 NFCONT）

（1）对于 S3C2410。

在本章实例中设为 0x9830——使能 NAND Flash 控制器、初始化 ECC、NAND Flash 片选信号 nFCE = 1（inactive，真正使用时再让它等于 0），设置 TACLS = 0，TWRPH0 = 3，TWRPH1 = 0。这些时序参数的含义为：TACLS = 1 个 HCLK 时钟，TWRPH0 = 4 个 HCLK 时钟，TWRPH1 = 1 个 HCLK 时钟。

K9F1208U0M 的时间特性如下：

```
CLE setup Time = 0 ns, CLE Hold Time = 10 ns,
ALE setup Time = 0 ns, ALE Hold Time = 10 ns,
WE Pulse Width = 25 ns
```

参考图 8.13，可以计算：即使在 HCLK=100MHz 的情况下，TACLS+TWRPH0+TWRPH1=6/100 μS = 60ns，也是可以满足 NAND Flash K9F1208U0M 的时序要求的。

（2）对于 S3C2440。

时间参数也设为：TACLS=0，TWRPH0=3，TWRPH1=0。NFCONF 寄存器的值如下：

```
NFCONF = 0x300
```

NFCONT 寄存器的取值如下，表示使能 NAND Flash 控制器、禁止控制引脚信号 nFCE、初始化 ECC。

```
NFCONT = (1<<4)|(1<<1)|(1<<0)
```

2. 在第一次操作 NAND Flash 前，通常复位一下 NAND Flash

（1）对于 S3C2410。

```
NFCONF &= ~(1<<11) (发出片选信号)
NFCMD  = 0xff  (reset 命令)
```

然后循环查询 NFSTAT 位 0，直到它等于 1。

最后禁止片选信号，在实际使用 NAND Flash 时再使能。

```
NFCONF |= (1<<11) (禁止 NAND Flash)
```

（2）对于 S3C2440。

```
NFCONT &= ~(1<<1) (发出片选信号)
NFCMD  = 0xff  (reset 命令)
```

然后循环查询 NFSTAT 位 0，直到它等于 1。

最后禁止片选信号，在实际使用 NAND Flash 时再使能。

```
NFCONT |= 0x2   (禁止 NAND Flash)
```

3. 发出读命令

先使能 NAND Flash，然后发出读命令。

（1）对于 S3C2410。

```
NFCONF &= ~(1<<11) (发出片选信号)
NFCMD  = 0 (读命令)
```

（2）对于 S3C2440。

```
NFCONT &= ~(1<<1) (发出片选信号)
NFCMD  = 0 (读命令)
```

4．发出地址信号

这步请注意，表 8.3 列出了在地址操作的 4 个步骤对应的地址线，没用到 A8（它由读命令设置，当读命令为 0 时，A8=0；当读命令为 1 时，A8=1），如下所示：

```
NFADDR = addr & 0xff
NFADDR = (addr>>9) & 0xff        (左移 9 位，不是 8 位)
NFADDR = (addr>>17) & 0xff       (左移 17 位，不是 16 位)
NFADDR = (addr>>25) & 0xff       (左移 25 位，不是 24 位)
```

5．循环查询 NFSTAT 位 0，直到它等于 1，这时可以读取数据了

6．连续读 NFDATA 寄存器 512 次，得到一页数据（512 字节）

循环执行第 3、4、5、6 这 4 个步骤，直到读出所要求的所有数据。

7．最后，禁止 NAND Flash 的片选信号

（1）对于 S3C2410。

```
NFCONF |= (1<<11)
```

（2）对于 S3C2440。

```
NFCONT |= (1<<1)
```

8.2.2 代码详解

实验代码在/work/hardware/nand 目录下，源文件为 head.S、init.c 和 main.c。本实例的目的是把一部分代码存放在 NAND Flash 地址 4096 之后，当程序启动后通过 NAND Flash 控制器将它们读出来、执行。以前的代码都小于 4096 字节，开发板启动后它们被自动复制进“Steppingstone”中。

连接脚本 nand.lds 把它们分为两部分，nand.lds 代码如下：

```
01 SECTIONS {
02   firtst     0x00000000 : { head.o init.o nand.o}
03   second 0x30000000 : AT(4096) { main.o }
```

```
04 }
05
```

第 2 行表示 head.o、init.o、nand.o 这 3 个文件的运行地址为 0，它们在生成的映象文件中的偏移地址也为 0（从 0 开始存放）。

第 3 行表示 main.o 的运行地址为 0x30000000，它在生成的映象文件中的偏移地址为 4096。

head.S 调用 init.c 中的函数来关 WATCH DOG、初始化 SDRAM；调用 nand.c 中的函数来初始化 NAND Flash，然后将 main.c 中的代码从 NAND Flash 地址 4096 开始处复制到 SDRAM 中；最后跳到 main.c 中的 main 函数继续执行。

由于 S3C2410、S3C2440 的 NAND Flash 控制器并非完全一样，这个程序要既能处理 S3C2410，也能处理 S3C2440，首先需要分辨出是 S3C2410 还是 S3C2440，然后使用不同的函数进行处理。读取 GSTATUS1 寄存器，如果它的值为 0x32410000 或 0x32410002，就表示处理器是 S3C2410，否则是 S3C2440。

nand.c 向外引出两个函数：用来初始化 NAND Flash 的 nand_init 函数、用来将数据从 NAND Flash 读到 SDRAM 的 nand_read 函数。

1. nand_init 函数分析

代码如下：

```
252 /* 初始化 NAND Flash */
253 void nand_init(void)
254 {
255 #define TACLS   0
256 #define TWRPH0  3
257 #define TWRPH1  0
258
259     /* 判断是 S3C2410 还是 S3C2440 */
260     if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
261     {
262         nand_chip.nand_reset            = s3c2410_nand_reset;
263         nand_chip.wait_idle             = s3c2410_wait_idle;
264         nand_chip.nand_select_chip      = s3c2410_nand_select_chip;
265         nand_chip.nand_deselect_chip    = s3c2410_nand_deselect_chip;
266         nand_chip.write_cmd             = s3c2410_write_cmd;
267         nand_chip.write_addr            = s3c2410_write_addr;
268         nand_chip.read_data             = s3c2410_read_data;
269
270         /* 使能 NAND Flash 控制器，初始化 ECC，禁止片选，设置时序 */
271         s3c2410nand->NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0
```

```
<<4)|(TWRPH1<<0);
272     }
273     else
274     {
275         nand_chip.nand_reset            = s3c2440_nand_reset;
276         nand_chip.wait_idle             = s3c2440_wait_idle;
277         nand_chip.nand_select_chip      = s3c2440_nand_select_chip;
278         nand_chip.nand_deselect_chip    = s3c2440_nand_deselect_chip;
279         nand_chip.write_cmd             = s3c2440_write_cmd;
280         nand_chip.write_addr            = s3c2440_write_addr;
281         nand_chip.read_data             = s3c2440_read_data;
282
283         /* 设置时序 */
284         s3c2440nand->NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
285         /* 使能 NAND Flash 控制器，初始化 ECC，禁止片选 */
286         s3c2440nand->NFCONT = (1<<4)|(1<<1)|(1<<0);
287     }
288
289     /* 复位 NAND Flash */
290     nand_reset();
291 }
```

第 260 行读取 GSTATUS1 寄存器来判断为 S3C2410 还是 S3C2440，然后分别处理：S3C2410、S3C2440 的 NAND Flash 控制器中，有一些寄存器的功能是相同的，但是它们的地址不一样；有一些寄存器的功能已经发生变化。所以使用两套函数来进行处理。

第 262～268 行设置 S3C2410 的 NAND Flash 处理函数，第 275～281 行设置 S3C2440 的 NAND Flash 处理函数，把这些函数赋给 nand_chip 结构，以后通过这个结构来调用。这只是一个编程技巧，代码的关键还是 s3c2410_nand_reset、s3c2440_nand_reset 等函数。

如果处理器是 S3C2410，则调用第 271 行的代码设置 NFCONF 寄存器：使能 NAND Flash 控制器，初始化 ECC，禁止片选，设置时序。如果处理器是 S3C2440，则使用第 284、286 两行代码来进行相同的设置（涉及两个寄存器：NFCONF 和 NFCONT）。

最后，第 290 行调用 nand_reset 函数复位 NAND Flash。在第一次使用之前通常复位一下。

其中涉及的各个函数都只有几行，主要是读写寄存器。

2. nand_read 函数分析

它的原型如下，表示从 NAND Flash 位置 start_addr 开始，将数据复制到 SDRAM 地址 buf 处，共复制 size 字节。

```
void nand_read(unsigned char *buf, unsigned long start_addr, int size)
```

代码如下：

```
297 /* 读函数 */
298 void nand_read(unsigned char *buf, unsigned long start_addr, int size)
299 {
300     int i, j;
301
302     if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
303         return ;    /* 地址或长度不对齐 */
304     }
305
306     /* 选中芯片 */
307     nand_select_chip();
308
309     for(i=start_addr; i < (start_addr + size);) {
310       /* 发出READ0命令 */
311       write_cmd(0);
312
313       /* Write Address */
314       write_addr(i);
315       wait_idle();
316
317       for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
318           *buf = read_data();
319           buf++;
320       }
321     }
322
323     /* 取消片选信号 */
324     nand_deselect_chip();
325
326     return ;
327 }
```

可以看到，读 NAND Flash 的操作分为 6 步。

① 选择芯片（第 307 行）。

② 发出读命令（第 311 行）。

③ 发出地址（第 314 行）。

④ 等待数据就绪（第 315 行）。

⑤ 读取数据（第 318 行）。

⑥ 结束后，取消片选信号（第 324 行）。

流程如图 8.14 所示。

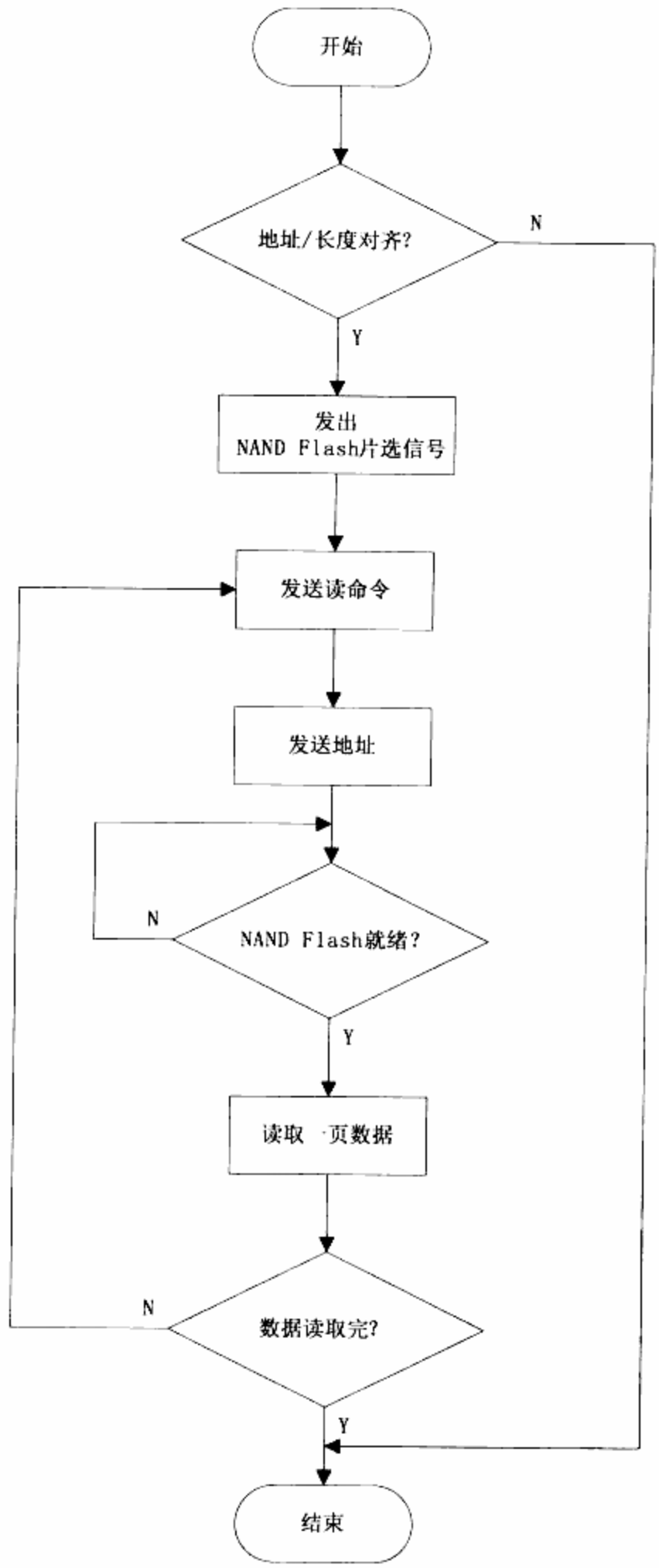
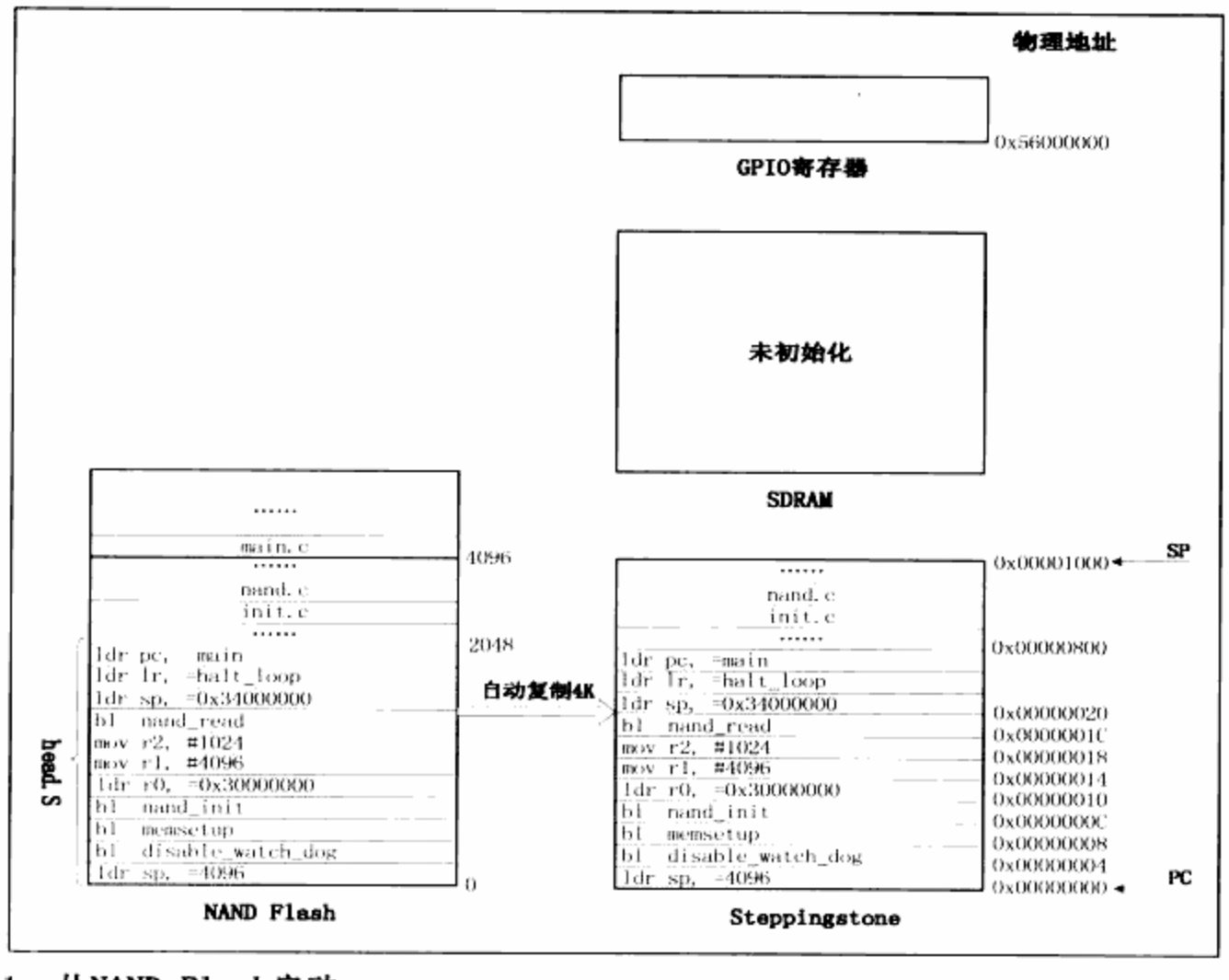
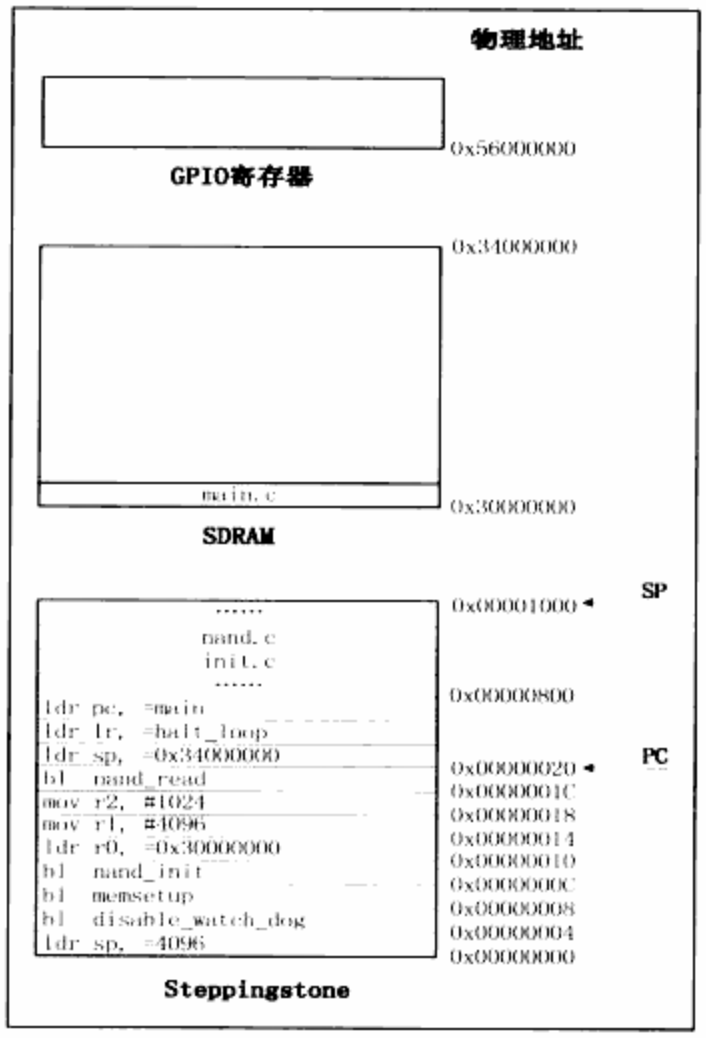


图 8.14 NAND Flash 读操作流程

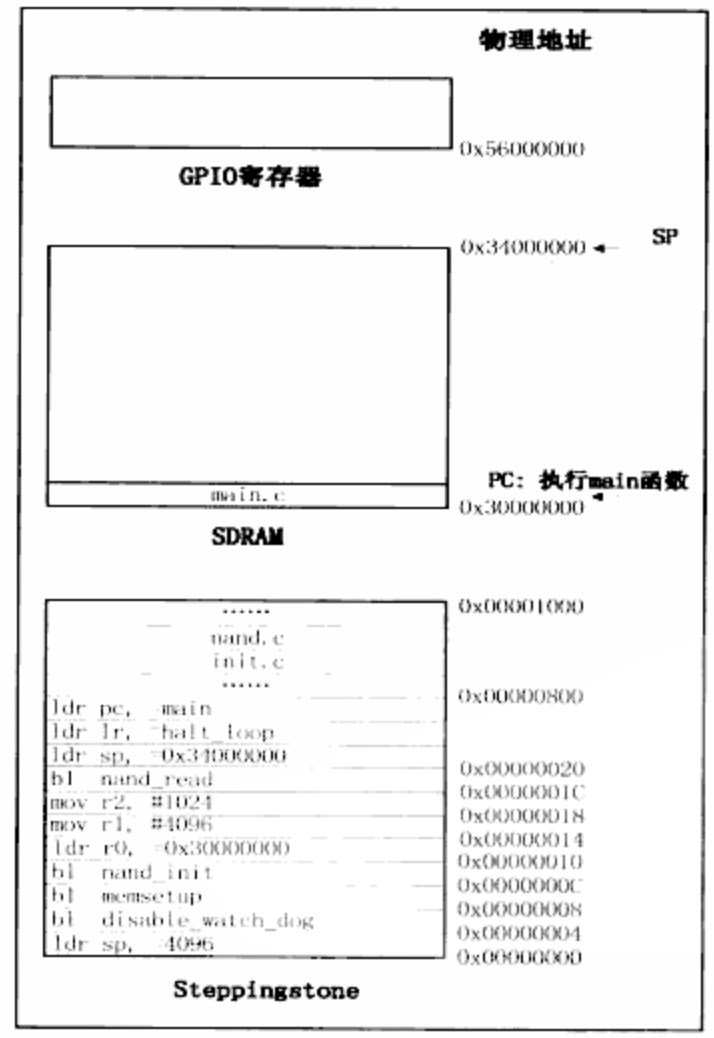
为了更形象地了解程序执行时代码复制、程序执行位置，请参考图 8.15。



1. 从NAND Flash启动

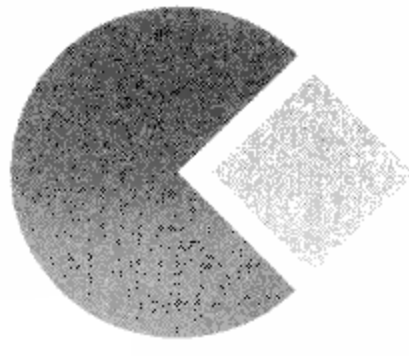


2. nand_read复制main.c的代码到SDRAM中



3. 执行SDRAM中的main函数

图 8.15 从 NAND Flash 复制代码到 SDRAM 并执行的过程



第 9 章　中断体系结构

本章目标

- 了解 ARM 体系 CPU 的 7 种工作模式
- 了解 S3C2410/S3C2440 中断体系结构
- 掌握 S3C2410/S3C2440 的中断服务程序的编写方法

9.1　S3C2410/S3C2440 中断体系结构

9.1.1　ARM 体系 CPU 的 7 种工作模式

ARM 体系的 CPU 有以下 7 种工作模式。

- 用户模式（usr）：ARM 处理器正常的程序执行状态。
- 快速中断模式（fiq）：用于高速数据传输或通道处理。
- 中断模式（irq）：用于通用的中断处理。
- 管理模式（svc）：操作系统使用的保护模式。
- 数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

可以通过软件来进行模式切换，或者发生各类中断、异常时 CPU 自动进入相应的模式。除用户模式外，其他 6 种工作模式都属于特权模式。大多数程序运行于用户模式，进入特权模式是为了处理中断、异常，或者访问被保护的系统资源。

另外，ARM 体系的 CPU 有以下两种工作状态。

- ARM 状态：此时处理器执行 32 位的字对齐的 ARM 指令。
- Thumb 状态：此时处理器执行 16 位的、半字对齐的 Thumb 指令。

实际上，本书所有程序都是在 ARM 状态下运行，而 CPU 一上电就处于 ARM 状态，所以无需关心 CPU 的工作状态。

ARM920T 有 31 个通用的 32 位寄存器和 6 个程序状态寄存器。这 37 个寄存器分为 7 组，进入某个工作模式时就使用它那组的寄存器。有些寄存器，不同的工作模式下有自己的副本，当切换到另一个工作模式时，那个工作模式的寄存器副本将被使用：这些寄存器被称为备份寄存器（图 9.1 中使用灰色三角形标记的寄存器）。

在 ARM 状态下，每种工作模式都有 16 个通用寄存器和 1 个（或 2 个，这取决于工作模式）程序状态寄存器。图 9.1 列出了 ARM 状态下不同工作模式所使用的寄存器。

图 9.1 ARM 状态下各工作模式使用的寄存器

图中 R0～R15 可以直接访问，这些寄存器中除 R15 外都是通用寄存器，即它们既可以用于保存数据也可以用于保存地址。另外，R13～R15 稍有特殊。R13 又被称为栈指针寄存器，通常被用于保存栈指针。R14 又被称为程序连接寄存器（subroutine Link Register）或连接寄存器，当执行 BL 子程序调用指令时，R14 中得到 R15（程序计数器 PC）的备份。而当发生中断或异常时，对应的 R14_svc、R14_irq、R14_fiq、R14_abt 或 R14_und 中保存 R15 返回值。R15 是程序计数器。

快速中断模式有 7 个备份寄存器 R8～R14（即 R8_fiq～R14_fiq），这使得进入快速中断模式执行很大部分程序时（只要它们不改变 R0～R7），甚至不需要保存任何寄存器。用户模式、管理模式、数据访问终止模式和未定义指令中止模式都含有两个独占的寄存器副本 R13、R14，这样可以令每个模式拥有自己的栈指针寄存器和连接寄存器。

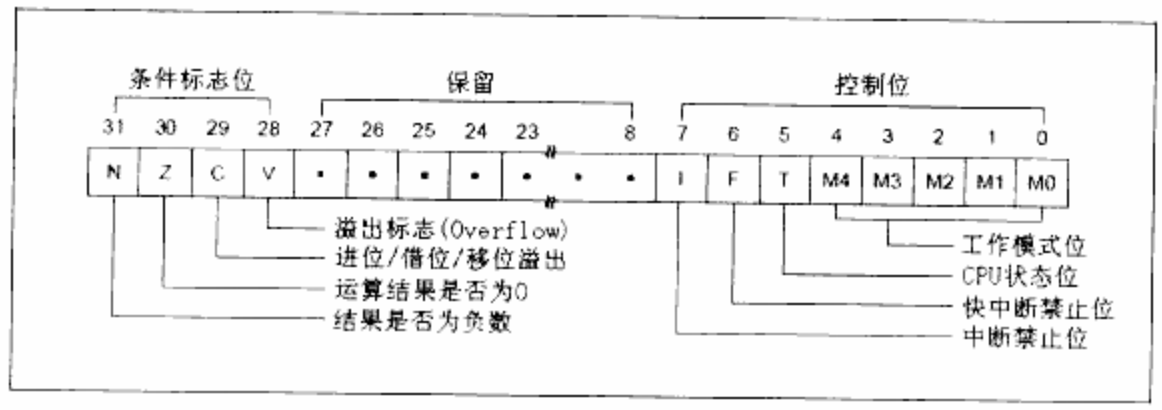
每种工作模式除 R0～R15 共 16 个寄存器外，还有第 17 个寄存器 CPSR，即“当前程序状态寄存器”（Current Program Status Register）。CPSR 中一些位被用于标识各种状态，一些位被用于标识当前处于什么工作模式。

CPSR 中各位意义如下，如图 9.2 所示。

（1）T 位：置位时，CPU 处于 Thumb 状态；否则处于 ARM 状态。

（2）中断禁止位：I 位和 F 位属于中断禁止位。它们被置位时，IRQ 中断、FIQ 中断分别

被禁止。



M[4:0]	工作模式	THUMB状态下可见的寄存器	ARM状态下可见的寄存器
10000	用户模式 (User)	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	快中断模式 (FIQ)	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	中断模式 (IRQ)	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	管理模式 (Supervisor)	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	数据访问中止模式 (Abort)	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	未定义指令中止模式 (Undefined)	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	系统模式 (System)	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

图 9.2 程序状态寄存器的格式

（3）工作模式位：表明 CPU 当前处于什么工作模式。可以编写这些位，使 CPU 进入指定的工作模式。

除 CPSR 外，还有快速中断模式、中断模式、管理模式、数据访问终止模式和未定义指令中止模式等 5 种工作模式和一个寄存器——SPSR，即“程序状态保存寄存器”（Saved Process Status Registers）。当切换进入这些工作模式时，在 SPSR 中保存前一个工作模式的 CPSR 值，这样，当返回前一个工作模式时，可以将 SPSR 的值恢复到 CPSR 中。

综上所述，当一个异常发生时，将切换进入相应的工作模式（为表述方便，下文中将它称为异常模式），这时 ARM920T CPU 核将自动完成如下事情。

（1）在异常工作模式的连接寄存器 R14 中保存前一个工作模式的下一条，即将执行的指令的地址。对于 ARM 状态，这个值是当前 PC 值加 4 或加 8（参考表 9.1）。

（2）将 CPSR 的值复制到异常模式的 SPSR。

（3）将 CPSR 的工作模式位设为这个异常对应的工作模式。

（4）令 PC 值等于这个异常模式在异常向量表中的地址，即跳转去执行异常向量表中的相应指令。

相反地，从异常工作模式退出回到之前的工作模式时，需要通过软件完成如下事情。

（1）前面进入异常工作模式时，连接寄存器中保存了前一工作模式的一个指令地址，将它减去一个适当的值（参考表 9.1）后赋给 PC 寄存器。

（2）将 SPSR 的值复制回 CPSR。

表 9.1 进入/退出异常模式时的 PC 地址

异常模式	退出异常模式时 PC 的计算方法	进入异常模式时 R14 中保存的值（ARM 状态）
管理模式(通过 SWI 指令进入)	MOVS PC，R14	PC + 4（1）
未定义指令终止模式	MOVS PC，R14	PC + 4（1）
快速中断模式	SUBS PC，R14，#4	PC + 4（2）
中断模式	SUBS PC，R14，#4	PC + 4（2）
数据访问终止模式	异常原因：指令预取终止 SUBS PC，R14，#4	PC + 4（1）
	异常原因：数据访问终止 SUBS PC，R14，#8	PC + 8（3）

注：

（1）PC 值是这些指令的地址：SWI、未定义的指令、在预取指时就失败的指令。

（2）PC 值是这些指令的地址：进入快速中断模式、中断模式前，被打断而未执行的指令。

（3）PC 值是这些指令的地址：导致数据访问终止的加载/存储指令（LDR、STR、LDM 和 STM）。

9.1.2 S3C2410/S3C2440 中断控制器

CPU 运行过程中，如何知道各类外设发生了某些不预期的事件，比如串口接收到了新数据、USB 接口中插入了设备、按下了某个按键等。主要有以下两个方法。

（1）查询方式：程序循环地查询各设备的状态并作出相应反应。它实现简单，常用在功能相对单一的系统中，比如在一个温控系统中可以使用查询方式不断检测温度的变化。缺点是占用 CPU 资源过高，不适用于多任务系统。

（2）中断方式：当某事件发生时，硬件会设置某个寄存器；CPU 在每执行完一个指令时，通过硬件查看这个寄存器，如果发现所关注的事件发生了，则中断当前程序流程，跳转到一个固定的地址处理这事件，最后返回继续执行被中断的程序。它的实现相对复杂，但是效率很高，是常用的方法。

参考图 9.3，不论何种 CPU，中断的处理过程是相似的。

（1）中断控制器汇集各类外设发出的中断信号，然后告诉 CPU。

（2）CPU 保存当前程序的运行环境（各个寄存器等），调用中断服务程序（ISR，Interrupt Service Routine）来处理这些中断。

（3）在 ISR 中通过读取中断控制器、外设的相关寄存器来识别这是哪个中断，并进行相应的处理。

（4）清除中断：通过读写中断控制器和外设的相关寄存器来实现。

（5）最后恢复被中断程序的运行环境（即上面保存的各个寄存器等），继续执行。

对于不同 CPU 而言，中断的处理只是细节的不同。S3C2410/S3C2440 的中断控制器结构如图 9.4 所示，可以从中看出中断的处理细节。

SUBSRCPND 和 SRCPND 寄存器表明有哪些中断被触发了，正在等待处理（pending）；SUBMASK（INTSUBMSK 寄存器）和 MASK（INTMSK 寄存器）用于屏蔽某些中断。

图中的“Request sources（with sub -register）”表示 INT_RXD0、INT_TXD0 等中断源

（S3C2410 中这类中断有 11 个，而 S3C2440 中有 15 个）。它们不同于“Request sources（without sub -register）”。

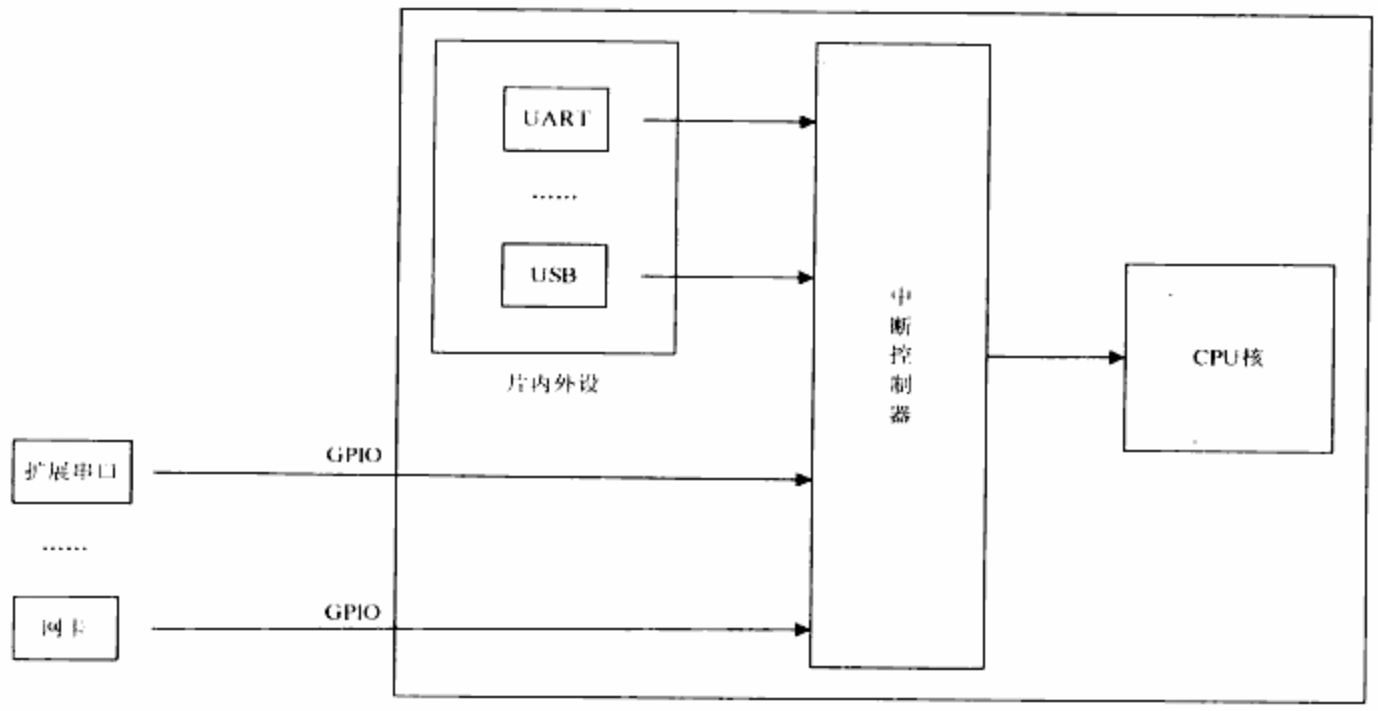


图 9.3 中断体系中外设、内部外设与 CPU 核的硬件框图

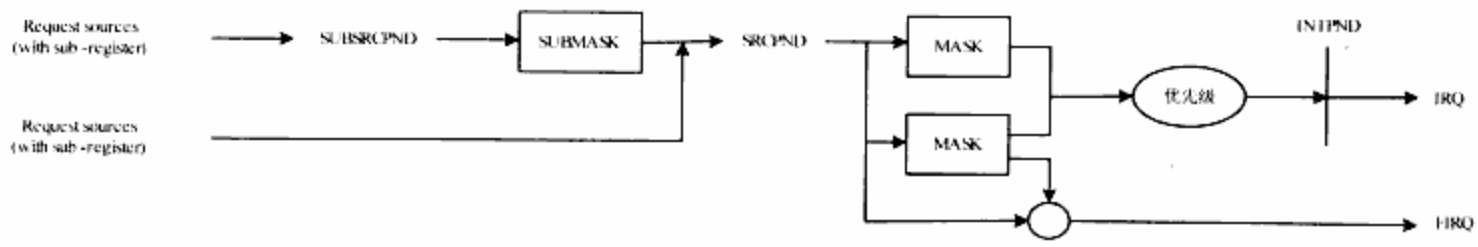


图 9.4 中断处理框图

（1）“Request sources（without sub -register）”中的中断源被触发之后，SRCPND 寄存器中相应位被置 1，如果此中断没有被 INTMSK 寄存器屏蔽或者快速中断（FIQ）的话，它将被进一步处理。

（2）对于“Request sources（with sub -register）”中的中断源被触发之后，SUBSRCPND 寄存器中的相应位被置 1，如果此中断没有被 INTSUBMSK 寄存器屏蔽的话，它在 SRCPND 寄存器中的相应位也被置 1，之后的处理过程就和“Request sources（without sub -register）”一样了。

继续沿着图 9.4 的箭头前进：在 SRCPND 寄存器中，被触发的中断的相应位被置 1，等待处理。

（1）如果被触发的中断中有快速中断（FIQ）——MODE（INTMOD 寄存器）中为 1 的位对应的中断是 FIQ，则 CPU 进入快速中断模式（FIQ Mode）进行处理。

注意 FIQ 只能分配一个，即 INTMOD 中只能有一位设为 1。

（2）对于一般中断 IRQ，可能同时有几个中断被触发，未被 INTMSK 寄存器屏蔽的中断经过比较后，选出优先级最高的中断，此中断在 INTPND 寄存器中的相应位被置 1，然后 CPU 进入中断模式（IRQ Mode）进行处理。中断服务程序可以通过读取 INTPND 寄存器或者 INTOFFSET 寄存器来确定中断源。

图 9.4 中的“Priority”表示中断的优先级判选，通过 PRIORITY 寄存器进行设置，这在 9.1.3 小节介绍。

综上所述，使用中断的步骤如下。

（1）设置好中断模式和快速中断模式下的栈：当发生中断 IRQ 时，CPU 进入中断模式，这时使用中断模式下的栈；当发生快速中断 FIQ 时，CPU 进入快速中断模式，这时使用快速中断模式下的栈。

（2）准备好中断处理函数。

① 异常向量。

在异常向量表中设置好当进入中断模式或快速中断模式时的跳转函数，它们的异常向量地址分别为 0x00000018、0x0000001c。

② 中断服务程序（ISR）。

IRQ、FIQ 的跳转函数，最终将调用具体中断的服务函数。

对于 IRQ，读取 INTPND 寄存器或 INTOFFSET 寄存器的值来确定中断源，然后分别处理。

对于 FIQ，因为只有一个中断可以设为 FIQ，无须判断中断源。

③ 清除中断：如果不清除中断，则 CPU 会误以为这个中断又一次发生了。

可以在调用 ISR 之前清除中断，也可以在调用 ISR 之后清除中断，这取决于在 ISR 执行过程中，这个中断是否可能继续发生、是否能够丢弃。如果在 ISR 执行过程中，这个中断可能发生并不能丢弃，则在调用 ISR 之前清除中断，这样在 ISR 执行过程中发生的中断能够被各寄存器再次记录并通知 CPU；如果在 ISR 执行过程中，这个中断不会发生或者可以丢弃，则在调用 ISR 之后清除中断。

清除中断时，从源头开始：首先，需要的话，操作具体外设清除中断信号；其次，清除 SUBSRCPND（用到的话）、SRCPND 寄存器中的相应位（往相应位写 1 即可）；最后，清除 INTPND 寄存器中的相应位（往相应位写 1 即可），最简单的方法就是“INTPND=INTPND”。

（3）进入、退出中断模式或快速中断模式时，需要保存、恢复被中断程序的运行环境。

① 对于 IRQ，进入和退出的代码如下：

```
sub lr, lr, #4                  @ 计算返回地址
stmdb   sp!, { r0-r12,lr }      @ 保存使用到的寄存器
……                              @ 处理中断
ldmia   sp!, { r0-r12,pc }^     @ 中断返回
                                @ ^表示将 spsr 的值赋给 cpsr
```

② 对于 FIQ，进入和退出的代码如下：

```
sub lr, lr, #4                  @ 计算返回地址
stmdb   sp!, { r0-r7,lr }       @ 保存使用到的寄存器
……                              @ 处理快速中断
ldmia   sp!, { r0-r7,pc }^      @ 快速中断返回
                                @ ^表示将 spsr 的值赋给 cpsr
```

（4）根据具体中断，设置相关外设。比如对于 GPIO 中断，需要将相应引脚的功能设为“外部中断”、设置中断触发条件（低电平触发、高电平触发、下降沿触发还是上升沿触发）等。一些中断拥有自己的屏蔽寄存器，还要开启它。

（5）对于“Request sources（without sub -register）”中的中断，将 INTSUBMSK 寄存器

中相应位设为 0。

（6）确定使用此中断的方式：FIQ 或 IRQ。

① 如果是 FIQ，则在 INTMOD 寄存器中设置相应位为 1。

② 如果是 IRQ，则在 RIORITY 寄存器中设置优先级。

（7）如果是 IRQ，将 INTMSK 寄存器中相应位设为 0（FIQ 不受 INTMSK 寄存器控制）。

（8）设置 CPSR 寄存器中的 I-bit（对于 IRQ）或 F-bit（对于 FIQ）为 0，使能 IRQ 或 FIQ。

9.1.3 中断控制器寄存器

SUBSRCPND、INTSUBMSK 这两个寄存器中相同的位对应相同的中断；SRCPND、INTMSK、INTMOD、INTPND 这 4 个寄存器中相同的位对应相同的中断。下面沿着图 9.4 的箭头方向讲解所涉及的寄存器。

1. SUBSRCPND 寄存器（SUB SOURCE PENDING）

SUBSRCPND 寄存器被用来标识 INT_RXD0、INT_TXD0 等中断（S3C2410 中这类中断有 11 个，而 S3C2440 中有 15 个）是否已经发生，每位对应一个中断。当这些中断发生并且没有被 INTSUBMSK 寄存器（下面介绍）屏蔽，则它们中的若干位将“汇集”出现在 SRCPND 寄存器（下面介绍）的一位上。比如 SUBSRCPND 寄存器中的 3 个中断 INT_RXD0、INT_TXD0、INT_ERR0，只要有一个发生了并且它没有被屏蔽，则 SRCPND 寄存器中的 INT_UART0 位被置 1。

要清除中断时，往 SUBSRCPND 寄存器中某位写入 1 即可令此位为 0；写入 0 无效果，数据保持不变。

SUBSRCPND 寄存器中各位对应的中断、SUBSRCPND 寄存器中哪几位“汇集”成 SRCPND 寄存器中的哪一位，请读者参考数据手册。

2. INTSUBMSK 寄存器（INTERRUPT SUB MASK）

INTSUBMSK 寄存器被用来屏蔽 SUBSRCPND 寄存器所标识的中断。INTSUBMSK 寄存器中某位被设为 1 时，对应的中断被屏蔽。

3. SRCPND 寄存器（SOURCE PENDING）

SRCPND 中每一位被用来表示一个（或一类）中断是否已经发生，即图 9.4 中输入“SRCPND”的两类中断：使用 SUBSRCPND/INTSUBMSK 控制的中断，不使用 SUBSRCPND/INTSUBMSK 控制的中断。

SRCPND 寄存器的操作与 SUBSRCPND 寄存器相似，若想清除某一位，往此位写入 1。

SRCPND 寄存器中各位对应哪个（或哪类）中断，请读者参考数据手册。

4. INTMSK 寄存器（INTERRUPT MASK）

INTMSK 寄存器被用来屏蔽 SRCPND 寄存器所标识的中断。INTMSK 寄存器中某位被设为 1 时，对应的中断被屏蔽。

INTMSK 寄存器只能屏蔽被设为 IRQ 的中断，不能屏蔽被设为 FIQ 的中断，请参考下面的 INTMOD 寄存器。

5. INTMOD 寄存器（INTERRUPT MODE）

当 INTMOD 寄存器中某位被设为 1 时，它对应的中断被设为 FIQ，即此中断发生时，CPU 将进入快速中断模式，这通常用来处理特别紧急的中断。

注意 同一时间里，INTMOD 寄存器中只能有一位被设为 1。

6. PRIORITY 寄存器

上面 INTMOD 寄存器中，将设为 1 的中断称为快速中断（FIQ），将其余设为 0 的中断称为普通中断（IRQ）。

当有多个普通中断同时发生时，中断控制器将选出最高优先级的中断，首先处理它。中断优先级的判选通过 7 个仲裁器来完成，包括 6 个一级仲裁器和 1 个二级仲裁器，结构如图 9.5 所示。

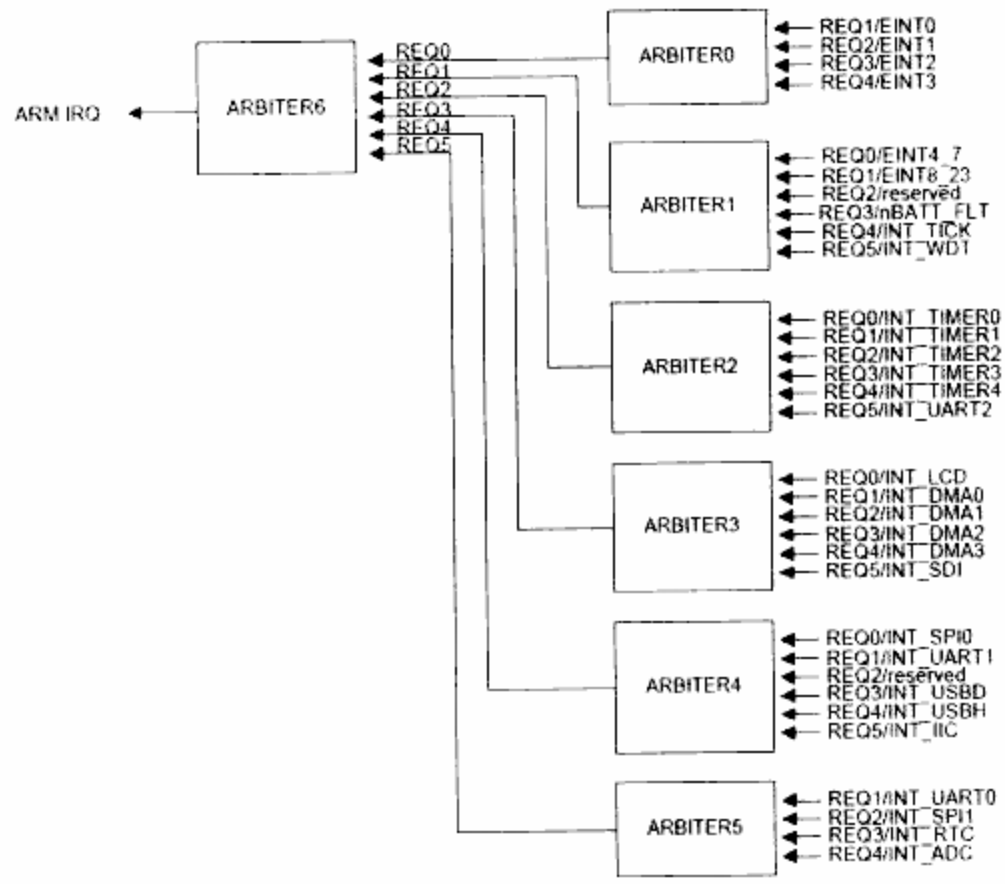


图 9.5 中断优先级仲裁器逻辑图

每个仲裁器含 6 个输入引脚 REQ0～REQ5。对于每个仲裁器，PRIORITY 寄存器使用三位来控制其行为：一位被用于选择仲裁器工作模式，称为 ARB_MODE；两位被用于控制各输入信号的优先级，称为 ARB_SEL。

ARB_SEL 的取值和 REQ0～REQ5 的优先级如表 9.2 所示。

表 9.2 中断优先仲裁器 ARB_SEL 取值和各输入信号的优先级对照表

ARB_SEL	优先级（从高到低）
00b	REQ0，REQ1，REQ2，REQ3，REQ4，REQ5
01b	REQ0，REQ2，REQ3，REQ4，REQ1，REQ5
10b	REQ0，REQ3，REQ4，REQ1，REQ2，REQ5
11b	REQ0，REQ4，REQ1，REQ2，REQ3，REQ5

注：REQ0 的优先级永远是最高的，REQ5 的优先级永远是最低的。

当某个仲裁器的 ARB_MODE 位被设为 0 时，它的 ARB_SEL 位是不会自动变化的，此时这个仲裁器的 6 个输入引脚的优先级固定不变（当然，可以通过软件修改 ARB_SEL 来改变它们的优先级）。当 ARB_MODE 位被设为 1 时，ARB_SEL 会随着“已经被服务的 REQx”（x 为 1～4）自动变化，变化顺序如表 9.3 所示。

表 9.3　中断优先级仲裁器的 ARB_SEL 变化规则

已经被服务的 REQx	ARB_SEL 的新值
REQ0/REQ5	维持不变
REQ1	01b
REQ2	10b
REQ3	11b
REQ4	00b

结合表 9.2、表 9.3 可知：当 ARB_MODE 为 1 时，某个 REQx（x 为 1～4）被服务之后，它的优先级变为 REQ0～REQ4 中的最低。

PRIORITY 寄存器中位[0:6]对应这 7 个仲裁器的 ARB_MODE 位（位[0]是 ARB_MODE0，依此类推），位[7:20]位对应这 7 个仲裁器的 ARB_SEL 位（[7:8]是 ARB_SEL0，依此类推）。

7. INTPND 寄存器（INTERRUPT PENDING）

经过中断优先级仲裁器选出优先级最高的中断后，这个中断在 INTPND 寄存器中的相应位被置 1，随后，CPU 将进入中断模式处理它。

同一时间内，此寄存器只有一位被置 1；在 ISR 中，可以根据这个位确定是哪个中断。清除中断时，往这个位写入 1。

8. INTOFFSET 寄存器（INTERRUPT OFFSET）

这个寄存器被用来表示 INTPND 寄存器中哪位被置 1 了，即 INTPND 寄存器中位[x]为 1 时，INTOFFSET 寄存器的值为 x（x 为 0～31）。

在清除 SRCPND、INTPND 寄存器时，INTOFFSET 寄存器被自动清除。

9.2 中断控制器操作实例：外部中断

9.2.1 按键中断代码详解

开发板上，K1～K4 四个按键所接的 CPU 引脚可以设成外部中断功能。本程序的 main 函数是一个不做任何事的无限循环，程序的功能完全靠中断来驱动：当按下某个按键时，CPU 调用其中断服务程序来点亮对应的 LED。

程序代码在/work/hardware/int 目录下，有 4 个源文件：head.S、init.c、interrupt.c 和 main.c，1 个头文件 s3c24xx.h。

1. head.S 代码详解

先看 head.S 文件：

```
01 @*****************************************************************
02 @ File: head.S
03 @ 功能: 初始化，设置中断模式、系统模式的栈，设置好中断处理函数
04 @*****************************************************************
05
06 .extern    main
07 .text
08 .global _start
09 _start:
10 @*****************************************************************
11 @ 中断向量，本程序中，除 Reset 和 HandleIRQ 外，其他异常都没有使用
12 @*****************************************************************
13     b  Reset
14
15 @ 0x04: 未定义指令终止模式的向量地址
16 HandleUndef:
17     b  HandleUndef
18
19 @ 0x08: 管理模式的向量地址，通过 SWI 指令进入此模式
20 HandleSWI:
21     b  HandleSWI
22
23 @ 0x0c: 指令预取终止导致的异常的向量地址
24 HandlePrefetchAbort:
25     b  HandlePrefetchAbort
26
27 @ 0x10: 数据访问终止导致的异常的向量地址
28 HandleDataAbort:
29     b  HandleDataAbort
30
31 @ 0x14: 保留
32 HandleNotUsed:
33     b  HandleNotUsed
```

```
34
35 @ 0x18: 中断模式的向量地址
36     b   HandleIRQ
37
38 @ 0x1c: 快中断模式的向量地址
39 HandleFIQ:
40     b   HandleFIQ
41
```

上面的第 17、21、25、29、33、36 和 40 行等共 7 条指令所在地址为 0x00、0x04、……、0x1c，这些地址上的指令被称为“异常向量”。当发生各类异常时，CPU 进入相应的工作模式，并跳转去执行它的“异常向量”。比如当复位时，CPU 进入系统模式，并跳到 0x00 地址开始执行；发生中断时，CPU 进入中断模式，并跳到 0x18 地址开始执行。

本程序中，只使用到“复位”和“中断”对应的异常向量，其他异常向量没有实际作用。

0x00 地址处的指令为“b　Reset”，在系统复位后，这条指令将跳去执行标号“Reset”开始的代码，它们完成一些初始化，代码如下：

```
42 Reset:
43     ldr sp, =4096              @ 设置栈指针，以下都是 C 函数，调用前需要设好栈
44     bl  disable_watch_dog      @ 关闭 WATCHDOG，否则 CPU 会不断重启
45
46     msr cpsr_c, #0xd2          @ 进入中断模式
47     ldr sp, =3072              @ 设置中断模式栈指针
48
49     msr cpsr_c, #0xdf          @ 进入系统模式
50     ldr sp, =4096              @ 设置系统模式栈指针
51                                @ 其实复位之后，CPU 就处于系统模式
52                                @ 前面的" ldr sp, =4096" 完成同样的功能，此句可省略
53
```

第 43、44 行读者已经很熟悉，不再介绍。第 46、47 行和第 49、50 行功能相似，前者用于设置中断模式的栈指针，后者用于设置系统模式的栈指针。注意，这时尚未完成所有初始化，所以还不能开中断——第 46、49 行代码中，CPSR 寄存器的 I 位、F 位都被设为 1。第 47、50 中 sp 寄存器并不是同一个寄存器，前者为 sp_irq，后者为 sp_sys。

继续往下看代码：

```
54     bl  init_led               @ 初始化 LED 的 GPIO 管脚
55     bl  init_irq               @ 调用中断初始化函数，在 init.c 中
56     msr cpsr_c, #0x5f          @ 设置 I-bit=0，开 IRQ 中断
57
58     ldr lr, =halt_loop         @ 设置返回地址
59     ldr pc, =main              @ 调用 main 函数
```

```
60 halt_loop:
61     b   halt_loop
62
```

第 54 行调用 init_led 函数来设置 LED1～LED4 这 4 个 LED 的 GPIO 为输出功能，第 55 行进行中断管脚的初始化（下面细述）。

当完成所有初始化后，第 56 行将 CPSR 寄存器中的 I 位设为 0，开 IRQ 中断。

最后，第 59 行调用 main 函数，main 函数是个不做任何事的无限循环。当按下按键时，这个循环被打断，CPU 进入中断模式，执行第 36 行的“b　HandleIRQ”的指令。

标号“HandleIRQ”开始的代码用于处理中断，如下所示：

```
62 HandleIRQ:
63     sub lr, lr, #4                    @ 计算返回地址
64     stmdb   sp!,    { r0-r12,lr }     @ 保存使用到的寄存器
65                                       @ 注意，此时的 sp 是中断模式的 sp
66                                       @ 初始值是上面设置的 3072
67
68    ldr lr, =int_return                @ 设置调用 ISR 即 EINT_Handle 函数后的返回地址
69     ldr pc, =EINT_Handle              @ 调用中断服务函数，在 interrupt.c 中
70 int_return:
71     ldmia   sp!,    { r0-r12,pc }^    @ 中断返回，^表示将 spsr 的值复制到 cpsr
72
```

第 63 行计算中断处理完毕后的返回地址，请参考表 9.1，lr 寄存器的值等于被中断指令的地址加 4，所以返回地址为 lr 的值减去 4。

第 64 行用于保存被中断程序的运行环境，即各个寄存器。其中的 sp 为中断模式的栈，在上面的第 47 行初始化。这样，r0～r12、lr 这 14 个寄存器被保存在中断模式的栈中。

第 68 行用于设置 EINT_Handle 函数执行完后的返回地址，这个地址为第 71 行指令的地址。

第 69 行调用中断服务函数 EINT_Handle（代码在 interrupt.c 中，下面详述）。

当 EINT_Handle 函数处理完所发生的中断后，返回第 71 行的指令。它恢复前面第 64 行保存的各个寄存器，即恢复被中断程序的运行环境：从栈中恢复 r0～r12、pc 这 14 个寄存器的值，同时，将 SPSR 寄存器的值复制到 CPSR（在进入中断模式时，CPU 自动将原来的 CPSR 值保存到 SPSR 中），这导致 CPU 切换到原来的工作模式。

2. init.c 中与中断相关的代码详解

下面详细讲述前面略过的 init_irq 函数，它在 init.c 中：

```
37 /*
38  * 初始化 GPIO 引脚为外部中断
39  * GPIO 引脚用作外部中断时，默认为低电平触发、IRQ 方式(不用设置 INTMOD)
40  */
```

```
41 void init_irq( )
42 {
43     GPFCON  = GPF0_eint | GPF2_eint;
44     GPGCON  = GPG3_eint | GPG11_eint;
45
46     // 对于 EINT11、EINT19，需要在 EINTMASK 寄存器中使能它们
47     EINTMASK &= (~(1<<11)) & (~(1<<19));
48
49     /*
50      * 设定优先级:
51      * ARB_SEL0 = 00b, ARB_MODE0 = 0: REQ1 > REQ3，即 EINT0 > EINT2
52      * 仲裁器 1、6 无需设置
53      * 最终:
54      * EINT0 > EINT2 > EINT11、EINT19、即 K4 > K3 > K1、K2
55      * EINT11 和 EINT19 的优先级相同
56      */
57     PRIORITY = (PRIORITY & ((~0x01) | (0x3<<7))) | (0x0 << 7) ;
58
59     // EINT0、EINT2、EINT8_23 使能
60     INTMSK   &= (~(1<<0)) & (~(1<<2)) & (~(1<<5));
61 }
```

第 43、44 行用于设置 K1～K4 对应的 GPIO 管脚为中断功能。使用 GPIO 的中断功能时，还需要确定它们的中断触发方式（是低电平触发、高电平触发、下降沿触发还是上升沿触发）。我们使用默认的低电平触发方式，无需额外设置。

第 47 行在 EINTMASK 寄存器中开启 EINT19、EINT11 中断，它们对应 K1 和 K2。K3、K4 对应的 EINT2、EINT0 中断不受 EINTMASK 寄存器控制。EINTMASK 寄存器可以屏蔽的中断请读者参考数据手册。

第 57 行用于设置中断优先级。本开发板中，外部中断 EINT19、EINT11、EINT2 和 EINT0 分别对应 K1、K2、K3 和 K4 这 4 个按键。请参考图 9.5，EINT0、EINT2 被接到仲裁器 0 的 REQ1、REQ3，程序中设置 ARB_SEL0 为 0（即 0b00，参考表 9.2），所以 REQ1 的优先级高于 REQ3，即 K4 的优先级高于 K3。程序中设置 ARB_MODE0 为 0，所以仲裁器 0 中各优先级保持不变。EINT8～EINT23 共用仲裁器 1 的 REQ1，所以 EINT11 和 EINT19 优先级相同。仲裁器 0、1 的输出接到仲裁器 6 的 REQ0、REQ1，而仲裁器中 REQ0 的优先级总是高于 REQ1，所以这 4 个控制的优先级如下：K4＞K3＞K1、K2；K1、K2 优先级相同。

本程序使用 GPIO 的默认中断方式——IRQ 方式（不是 FIQ 方式），所以不用设置 INTMOD 寄存器。

最后，在第 60 行将 INTMSK 寄存器中 EINT0、EINT2、EINT8_23 这 3 个（类）中断对应的位设为 0，使能中断（注意，即使到这里，中断仍未完全开启，head.S 第 56 行才打开了最后一个开关）。

3. interrupt.c 中的中断处理函数

上面讲解了有关的初始化、中断的进入和退出，真正的处理函数为 EINT_Handle，它被称为中断服务程序（ISR），代码在 interrupt.c 中，如下所示：

```
01 #include "s3c24xx.h"
02
03 void EINT_Handle()
04 {
05     unsigned long oft = INTOFFSET;
06     unsigned long val;
07
08     switch( oft )
09     {
10         // K4 被按下
11         case 0:
12         {
13             GPBDAT |= (0x0f<<5);   // 所有 LED 熄灭
14             GPBDAT &= ~(1<<8);     // LED4 点亮
15             break;
16         }
17
18         // K3 被按下
19         case 2:
20         {
21             GPBDAT |= (0x0f<<5);   // 所有 LED 熄灭
22             GPBDAT &= ~(1<<7);     // LED3 点亮
23             break;
24         }
25
26         // K1 或 K2 被按下
27         case 5:
28         {
29             GPBDAT |= (0x0f<<5);   // 所有 LED 熄灭
30
31             // 需要进一步判断是 K1 还是 K2 被按下，或是 K1、K2 被同时按下
32             val = EINTPEND;
33             if (val & (1<<11))
34                 GPBDAT &= ~(1<<6);     // K2 被按下，LED2 点亮
35             if (val & (1<<19))
```

```
36                GPBDAT &= ~(1<<5);      // K1 被按下，LED1 点亮
37
38            break;
39        }
40
41        default:
42            break;
43    }
44
45    //清除中断
46    if( oft == 5 )
47        EINTPEND = (1<<11) | (1<<19);   // EINT8~EINT23 合用 IRQ5
48    SRCPND = 1<<oft;
49    INTPND = 1<<oft;
50 }
```

以上代码主要关注中断的识别和清除，其余代码根据所识别出来的中断（即按键）点亮对应的 LED。第 5 行用来读取 INTOFFSET 寄存器，它的值被用来标识 INTPND 寄存器中哪位被设为 1。此值为 0 时表示 INTPND 寄存器的位[0]为 1，即 EINT0 中断发生了，这说明 K4 被按下；此值为 2 时表示 INTPND 寄存器的位[2]为 1，即 EINT2 中断发生了，这说明 K3 被按下；此值为 5 时表示 INTPND 寄存器的位[5]为 1，即 EINT8～EINT23 中至少一个中断发生了，在本程序中这表示 K1、K2 中至少被按下了一个，至于是哪个，需要进一步判断。

第 32 行用来读取 EINTPEND 寄存器，它的位 x 为 1 时，表示 EINTx 已经发生（x 为 4～23）。本程序就是通过读取 EINTPEND 寄存器的值来进一步判断是 EINT11 还是 EINT19 发生了，即是 K2 还是 K1 被按下了。

第 47 行用于清除 EINTPEND 寄存器，往某位写入 1 即可清除此位。

第 48、49 行用于清除 SRCPND、INTPND 寄存器。

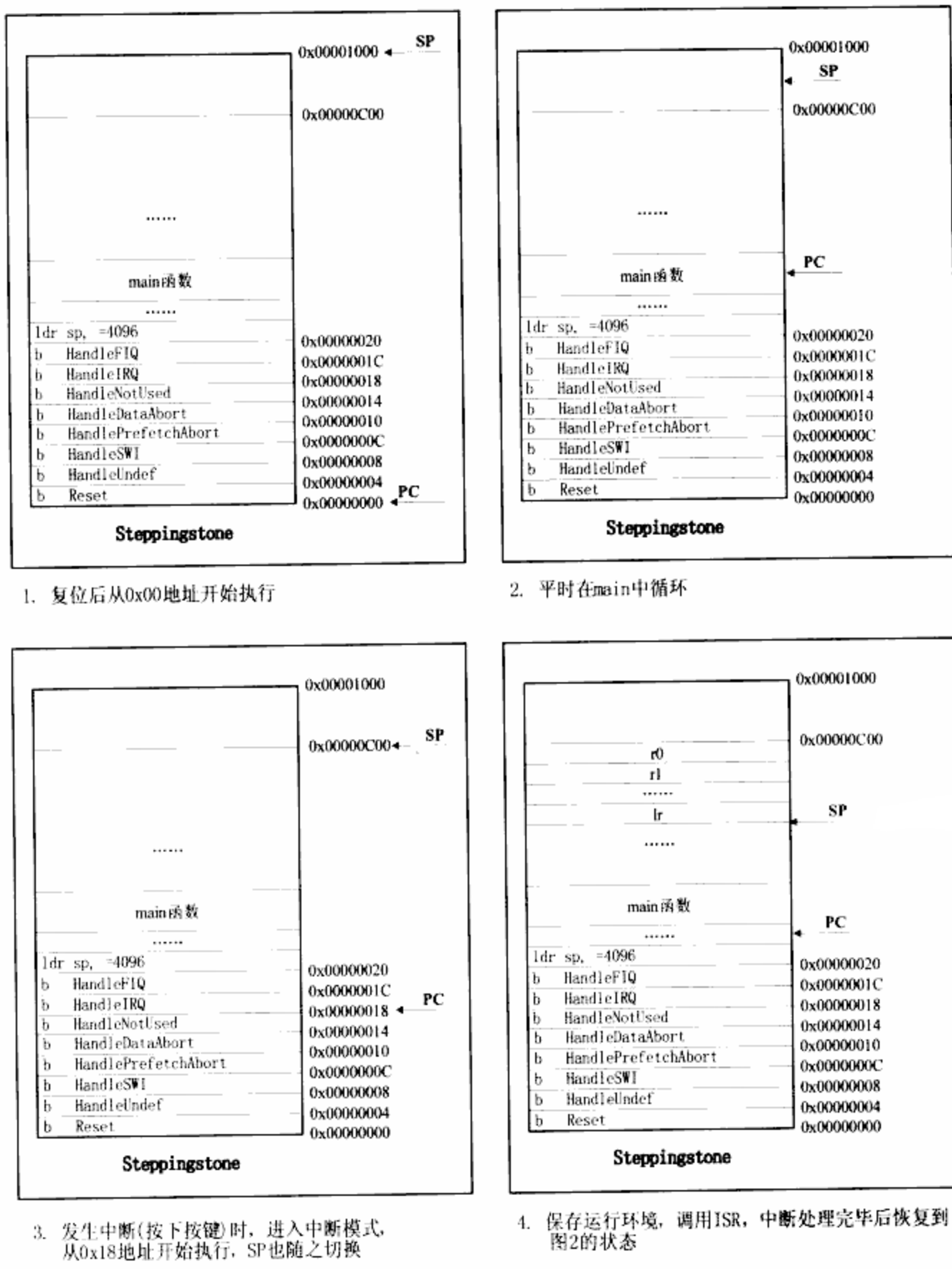
注意 清除顺序很重要：先是 EINTPEND，然后是 SRCPND，最后是 INTPND

4. 主函数

程序的主循环很简单，在 main.c 中，只是个不做任何事情的无限循环，如下所示：

```
01 int main()
02 {
03     while(1);
04     return 0;
05 }
```

为了更形象地了解本程序，下面用图 9.6 来演示代码的执行过程，注意其中 SP、PC 寄存器的变化。



1. 复位后从0x00地址开始执行

2. 平时在main中循环

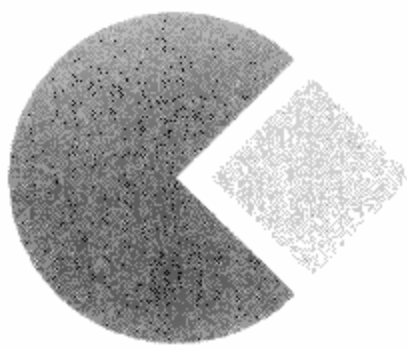
3. 发生中断(按下按键)时，进入中断模式，从0x18地址开始执行，SP也随之切换

4. 保存运行环境，调用ISR，中断处理完毕后恢复到图2的状态

图 9.6 中断程序的执行过程

9.2.2 实例测试

在源码目录 INT 下执行 make 命令生成可执行程序 int，使用 JTAG 工具烧入 NAND Flash 后，按复位键启动系统。轮流按下 K1～K4，可以看见 LED1～LED4 被轮流点亮；同时按下 K3、K4 时，只有 LED4 被点亮；同时按下 K1、K2 时，LED1、LED2 被同时点亮；同时按下所有按键时，只有 LED4 被点亮。读者可以同时按下几个按键，验证它们的优先级。



第 10 章　系统时钟和定时器

本章目标

- 了解 S3C2410/S3C2440 的时钟体系结构
- 掌握通过设置 MPLL 改变系统时钟的方法
- 掌握在不同的频率下设置存储控制器的方法
- 掌握 PWM 定时器的用法
- 了解 WATCHDOG 定时器的用法

10.1　时钟体系及各类时钟部件

10.1.1　S3C2410/S3C2440 时钟体系

S3C2410/S3C2440 的时钟控制逻辑既可以外接晶振，然后通过内部电路产生时钟源；也可以直接使用外部提供的时钟源，它们通过引脚的设置来选择。时钟控制逻辑给整个芯片提供 3 种时钟：FCLK 用于 CPU 核；HCLK 用于 AHB 总线上设备，比如 CPU 核、存储器控制器、中断控制器、LCD 控制器、DMA 和 USB 主机模块等；PCLK 用于 APB 总线上的设备，比如 WATCHDOG、IIS、I^2C、PWM 定时器、MMC 接口、ADC、UART、GPIO、RTC 和 SPI。

AHB（Advanced High performance Bus）总线主要用于高性能模块（如 CPU、DMA 和 DSP 等）之间的连接；APB（Advanced Peripheral Bus）总线主要用于低带宽的周边外设之间的连接，例如 UART、I^2C 等。

S3C2410 CPU 核的工作电压为 1.8V 时，主频可以达到 200MHz；工作电压为 2.0V 时，主频可达 266MHz。S3C2440 CPU 核的工作电压为 1.2V 时，主频可以达到 300MHz；工作电压为 1.3V 时，主频可达 400MHz。为了降低电磁干扰、降低板间布线的要求，S3C2410/S3C2440 外接的晶振频率通常很低，本开发板上为 12MHz，需要通过时钟控制逻辑的 PLL 提高系统时钟。

S3C2410/S3C2440 有两个 PLL：MPLL 和 UPLL。UPLL 专用于 USB 设备，MPLL 用于设置 FCLK、HCLK、PLCK。它们的设置方法相似，本书以 MPLL 为例。

上电时，PLL 没被启动，FCLK 即等于外部输入的时钟，称为 Fin。若要提高系统时钟，需要软件来启用 PLL，下面结合图 10.1 来介绍 PLL 的设置过程。请跟随 FCLK 的图像了解启动过程。

（1）上电几毫秒后，晶振（图中的 OSC）输出稳定，FCLK=Fin（晶振频率），nRESET

信号恢复高电平后，CPU 开始执行指令。

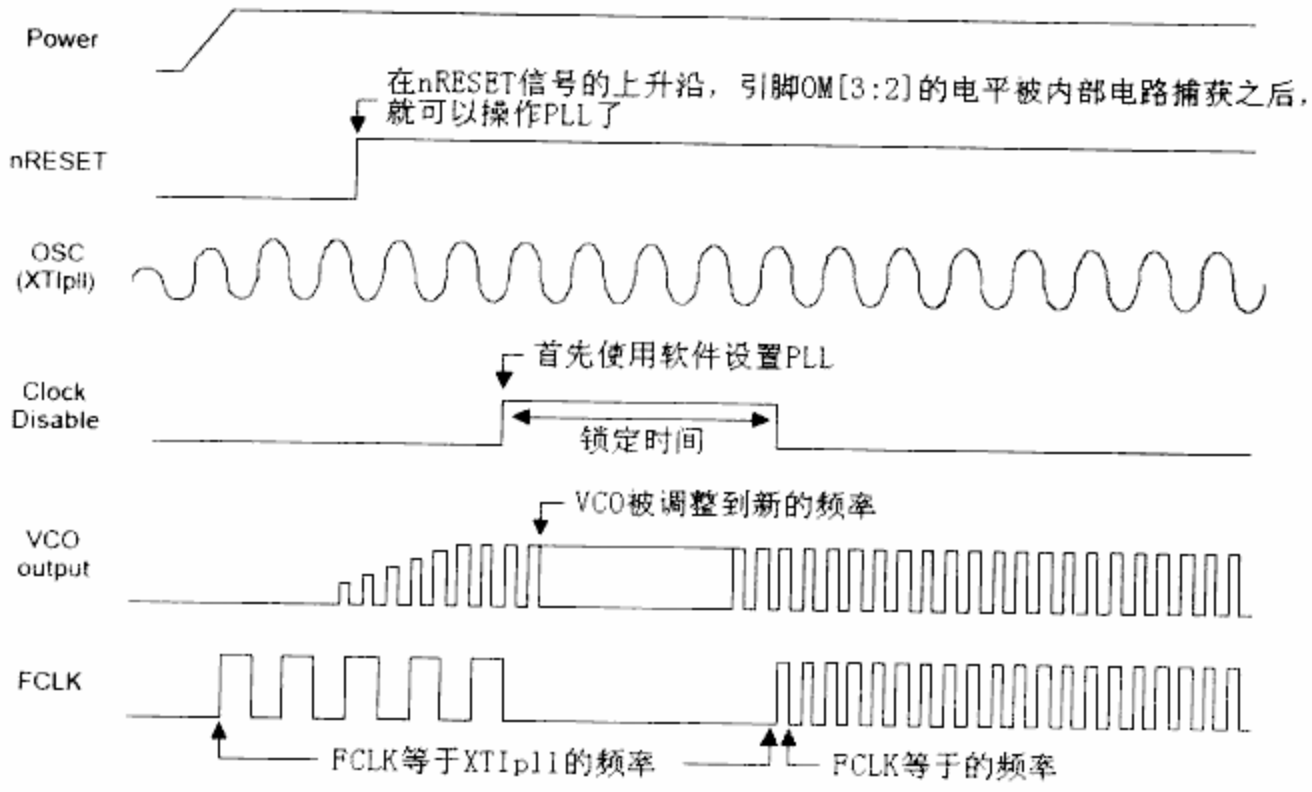


图 10.1 上电后 MPLL 的启动过程（使用晶振）

（2）可以在程序开头启动 MPLL，设置 MPLL 的几个寄存器后，需要等待一段时间（Lock Time），MPLL 的输出才稳定。在这段时间（称为 Lock Time）内，FCLK 停振，CPU 停止工作。Lock Time 的长短由寄存器 LOCKTIME 设定。

（3）Lock Time 之后，MPLL 输出正常，CPU 工作在新的 FCLK 下。

FCLK、HCLK 和 PCLK 的比例是可以改变的，设置它们三者的比例，启动 MPLL 只需要设置 3 个寄存器（对于 S3C2440 的一些时钟比例，还需要额外设置一个寄存器）。

（1）LOCKTIME 寄存器（LOCK TIME COUNT）：用于设置“Lock Time”的长度。

前面说过，MPLL 启动后需要等待一段时间（Lock Time），使得其输出稳定。S3C2410 中，位[23:12]用于 UPLL，位[11:0]用于 MPLL。S3C2440 中，位[31:16]用于 UPLL，位[15:0]用于 MPLL。一般而言，使用它的默认值即可，S3C2410 中默认值为 0x00FFFFFF，S3C2440 中默认值为 0xFFFFFFFF。

（2）MPLLCON 寄存器（Main PLL Control）：用于设置 FCLK 与 Fin 的倍数。

位[19:12]的值称为 MDIV，位[9:4]的值称为 PDIV，位[1:0]的值称为 SDIV。FCLK 与 Fin 的关系有如下计算公式。

① 对于 S3C2410：

MPLL(FCLK) = (m * Fin)/(p * 2^s)

其中：m = MDIV + 8，p = PDIV + 2，s = SDIV。

② 对于 S3C2440：

MPLL(FCLK) = (2 * m * Fin)/(p * 2^s)

其中：m = MDIV+8，p = PDIV+2，s = SDIV。

当设置 MPLLCON 之后——相当于图 10.1 中的“首先使用软件设置 PLL”，Lock Time 就被自动插入。Lock Time 之后，MPLL 输出稳定，CPU 工作在新的 FCLK 下。

（3）CLKDIVN 寄存器（CLOCK DIVIDER CONTROL）：用于设置 FCLK、HCLK、PCLK

三者的比例。

对于 S3C2410、S3C2440，这个寄存器表现稍有不同，请参考表 10.1 和图 10.2。

表 10.1　　S3C2410 CLKDIVN 寄存器格式

CLKDIVN	位	说　明
HDIVN1	2	0 表示保留 1 表示 FCLK:HCLK:PCLK = 1:4:4，此时 HDIVN、PDIVN 必须设为 0b00
HDIVN	1	HCLK 的分频系数，0-HCLK = FCLK，1-HCLK = FCLK/2
PDIVN	0	PCLK 的分频系数，0-PCLK = HCLK，1-PCLK = HCLK/2

对于 S3C2440 的一些时钟比例，还需要颇外设置一个寄存器 CAMDIVN。图 10.2 中，HDIVN 为 CLKDIVN 寄存器为位[2:1]，PDIVN 为位[0]；HCLK4_HALF、HCLK3_HALF 分别为 CAMDIVN 寄存器的位[9]、位[8]。各种时钟比例对应的寄存器设置如图 10.2 所示。

HDIVN	PDIVN	HCLK3_HALF/ HCLK4_HALF	FCLK	HCLK	PCLK	分频比
0	0	–	FCLK	FCLK	FCLK	1 : 1 : 1 (缺省)
0	1	–	FCLK	FCLK	FCLK / 2	1 : 1 : 2
1	0	–	FCLK	FCLK / 2	FCLK / 2	1 : 2 : 2
1	1	–	FCLK	FCLK / 2	FCLK / 4	1 : 2 : 4
3	0	0 / 0	FCLK	FCLK / 3	FCLK / 3	1 : 3 : 3
3	1	0 / 0	FCLK	FCLK / 3	FCLK / 6	1 : 3 : 6
3	0	1 / 0	FCLK	FCLK / 6	FCLK / 6	1 : 6 : 6
3	1	1 / 0	FCLK	FCLK / 6	FCLK / 12	1 : 6 : 12
2	0	0 / 0	FCLK	FCLK / 4	FCLK / 4	1 : 4 : 4
2	1	0 / 0	FCLK	FCLK / 4	FCLK / 8	1 : 4 : 8
2	0	0 / 1	FCLK	FCLK / 8	FCLK / 8	1 : 8 : 8
2	1	0 / 1	FCLK	FCLK / 8	FCLK / 16	1 : 8 : 16

图 10.2　S3C2440 FCLK/HCLK/PCLK 比例设置

对于 S3C2410，HDIVN 是 CLKDIVN 寄存器的位[1]；对于 S3C2440，HDIVN 是 CLKDIVN 寄存器的位[2:1]。如果 HDIVN 非 0，CPU 的总线模式应该从“fast bus mode”变为“asynchronous bus mode”，这可以通过如下指令来完成：

```
# MMU_SetAsyncBusMode
mrc p15, 0, r0, c1, c0, 0
orr r0, r0, #R1_nF:OR:R1_iA
mcr p15,0, r0, c1, c0, 0
```

其中的“R1_nF:OR:R1_iA”等于 0xC0000000。如果 HDIVN 非 0 时，而 CPU 的总线模式仍是“fast bus mode”，则 CPU 的工作频率将自动变为 HCLK，而不再是 FCLK。

10.1.2　PWM 定时器

S3C2410/S3C2440 的定时器部件完全一样，共有 5 个 16 位的定时器。其中定时器 0、1、2、3 有 PWM（Pulse Width Modulation）功能，即它们都有一个输出引脚，可以通过定时器来控制引脚周期性的高、低电平变化；定时器 4 没有输出引脚。

定时器部件的时钟源为 PCLK，首先通过两个 8 位的预分频器降低频率：定时器 0、1 共

用第一个预分频器，定时器 2、3、4 共用第二个预分频器。预分频器的输出将进入第二级分频器，它们输出 5 种频率的时钟：2 分频、4 分频、8 分频、16 分频或者外部时钟 TCLK0/TCLK1。每个定时器的工作时钟可以从这 5 种频率中选择。

这两个预分频都可以通过 TCFG0 寄存器来设置，每个定时器工作在哪种频率下也可以通过 TCFG1 寄存器来选择。如图 10.3 所示形象地说明定时器的结构。

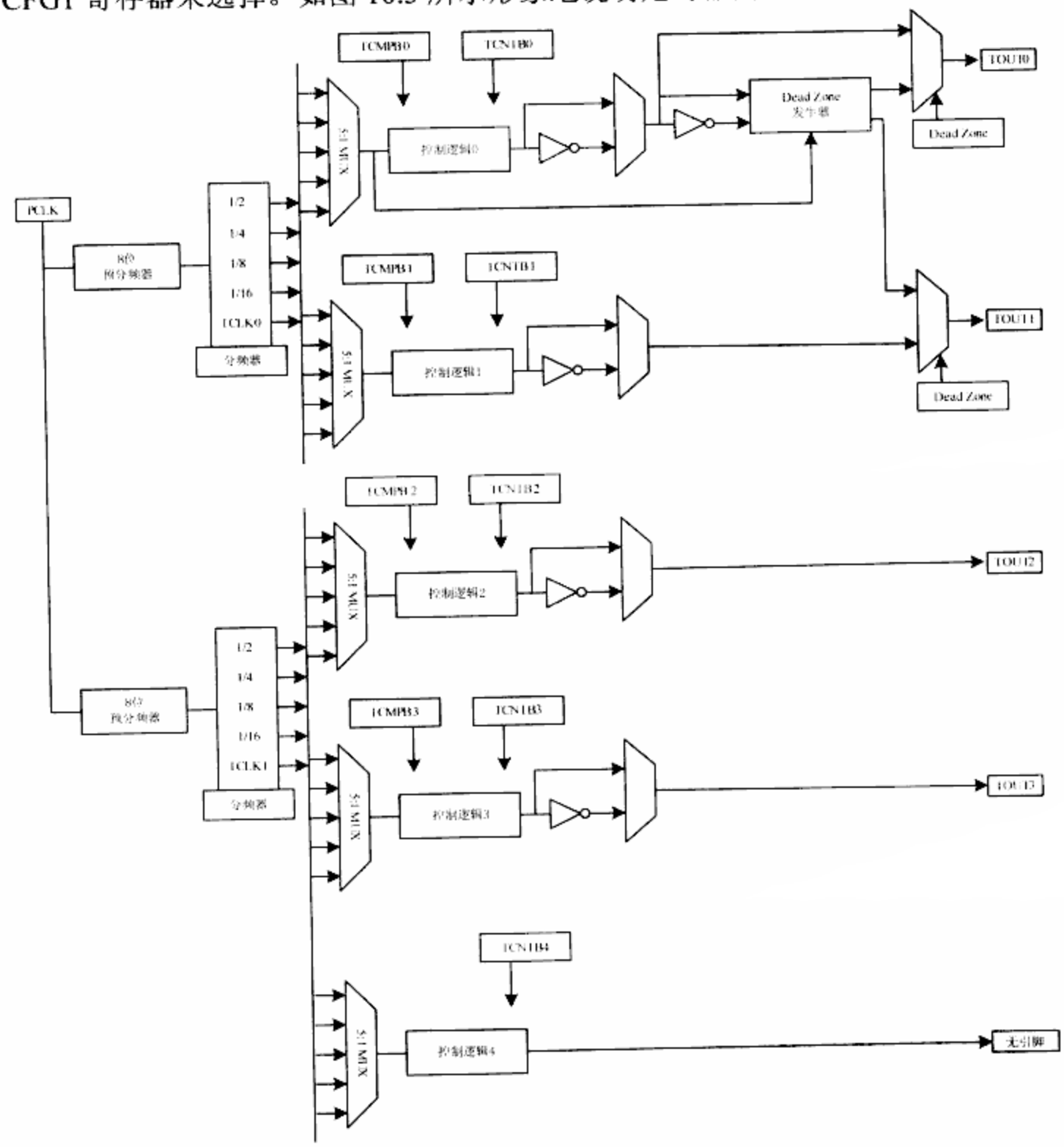


图 10.3 定时器结构图

上面只是确定了定时器的工作频率，至于定时器如何工作还得了解其内部结构，如图 10.4 所示。

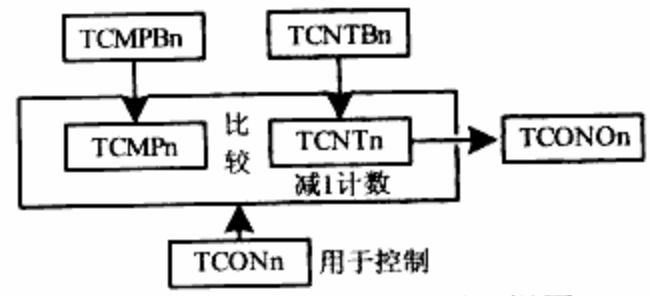


图 10.4 定时器内部控制逻辑图

定时器内部控制逻辑的工作流程如下。

（1）程序初始，设定 TCMPBn、TCNTBn 这两个寄存器，它们表示定时器 n 的比较值、初始计数值。

（2）随之设置 TCON 寄存器启动定时器 n，这时，TCMPBn、TCNTBn 的值将被装入其内部寄存器 TCMPn、

TCNTn 中。在定时器 n 的工作频率下，TCNTn 开始减 1 计数，其值可以通过读取 TCNTOn 寄存器得知。

（3）当 TCNTn 的值等于 TCMPn 的值时，定时器 n 的输出管脚 TOUTn 反转；TCNTn 继续减 1 计数。

（4）当 TCNTn 的值到达 0 时，其输出管脚 TOUTn 再次反转，并触发定时器 n 的中断（如果中断使能了的话）。

（5）当 TCNTn 的值到达 0 时，如果在 TCON 寄存器中将定时器 n 设为“自动加载”，则 TCMPB0 和 TCNTB0 寄存器的值被自动装入 TCMP0 和 TCNT0 寄存器中，下一个计数流程开始。

定时器 n 的输出管脚 TOUTn 初始状态为高电平，以后在 TCNTn 的值等于 TCMPn 的值、TCNTn 的值时反转。也可以通过 TCON 寄存器设置其初始电平，这样 TOUTn 的输出就完全反相了。通过设置 TCMPBn、TCNTBn 的值可以设置管脚 TOUTn 输出信号的占空比，这就是所谓的可调制脉冲（PWM，Pulse Width Modulation），所以这些定时器又被称为 PWM 定时器。

下面讲解定时器的寄存器使用方法。

（1）TCFG0 寄存器（TIMER CONFIGURATION）。

位[7:0]、位[15:8]分别被用于控制预分频器 0、1，它们的值为 0～255。经过预分频器出来的时钟频率为：PCLK / {prescaler value+1}。

（2）TCFG1 寄存器（TIMER CONFIGURATION）。

经过预分频器得到的时钟将被 2 分频、4 分频、8 分频和 16 分频，除这 4 种频率外，额外地，定时器 0、1 还可以工作在外接的 TCLK0 时钟下，定时器 2、3、4 还可以工作在外接的 TCLK1 时钟下。

通过 TCFG1 寄存器来设置这 5 个定时器，分别工作于这 5 个频率中哪一个之下，如表 10.2 所示。

这样，定时器 n 的工作频率或者是外接的 TCLK0 或 TCLK1 可以通过这个公式计算：

定时器工作频率 = PCLK / {prescaler value+1} / {divider value}

{prescaler value} = 0～255

{divider value} = 2, 4, 8, 16

表 10.2　　TCFG1 寄存器格式

定 时 器	位	值
0	[3:0]	0b0000 = 1/2，　0b0001 = 1/4，　0010 = 1/8 0b0011 = 1/16，　0b01xx = External TCLK1
1	[7:4]	0b0000 = 1/2，　0b0001 = 1/4，　0010 = 1/8 0b0011 = 1/16，　0b01xx = External TCLK1
2	[11:8]	0b0000 = 1/2，　0b0001 = 1/4，　0010 = 1/8 0b0011 = 1/16，　0b01xx = External TCLK1
3	[15:12]	0b0000 = 1/2，　0b0001 = 1/4，　0010 = 1/8 0b0011 = 1/16，　0b01xx = External TCLK0
4	[19:16]	0b0000 = 1/2，　0b0001 = 1/4，　0010 = 1/8 0b0011 = 1/16，　0b01xx = External TCLK0

（3）TCNTBn/TCMPBn 寄存器（COUNT BUFFER REGISTER & COMPARE BUFFER REGISTER）。

n 为 0～4，这两个寄存器都只用到位[15:0]，TCNTBn 中保存定时器的初始计数值，TCMPBn 中保存比较值。它们的值在启动定时器时，被传到定时器内部寄存器 TCNTn、TCMPn 中。

没有 TCMPB4，因为定时器 4 没有输出引脚。

（4）TCNTOn 寄存器（COUNT OBSERVATION）。

n 为 0～4，定时器 n 被启动后，内部寄存器 TCNTn 在其工作时钟下不断减 1 计数，可以通过读取 TCNTOn 寄存器得知其值。

（5）TCON 寄存器（TIMER CONTROL）。

它有以下 4 个作用：

① 第一次启动定时器时“手动”将 TCNTBn/TCMPBn 寄存器的值装入内部寄存器 TCNTn、TCMPn 中。

② 启动、停止定时器。

③ 决定在定时器计数到达 0 时是否自动将 TCNTBn/TCMPBn 寄存器的值装入内部寄存器 TCNTn、TCMPn 中。

④ 决定定时器的管脚 TOUTn 的输出电平是否反转。

TCON 寄存器位[3:0]、位[11:8]、位[15:12]、位[19:16]、位[22:20]分别用于定时器 0～4。除定时器因为没有输出引脚在没有“输出反转”位外，其他位的功能相似。表 10.3 以定时器 0 为例说明这些位的作用。

表 10.3　　TCON 寄存器格式

功　能	位	设　置
开启/停止	0	0：停止定时器 0；1：开启定时器 0
手动更新	1	0：无用 1：将 TCNTBn/TCMPBn 寄存器的值装入内部寄存器 TCNTn、TCMPn 中
输出反转	2	0：TOUT0 不反转；1：TOUT0 反转
自动加载	3	0：不自动加载； 1：在定时器 0 计数达到 0 时，TCNTBn/TCMPBn 寄存器的值自动装入内部寄存器 TCNTn、TCMPn 中

在第一次使用定时器时，需要设置“手动更新”位为 1 以使 TCNTBn/TCMPBn 寄存器的值装入内部寄存器 TCNTn、TCMPn 中。下一次如果还要设置这一位，需要先将它清 0。

定时器还有其他功能，比如 DMA、Dead zone 等，需要了解的读者请自行阅读数据手册，寄存器中涉及它们的部分这里也省略了。

10.1.3 WATCHDOG 定时器

WATCHDOG 定时器可以像一般 16 位定时器一样用于产生周期性的中断，也可以用于发出复位信号以重启失常的系统。它与 PWM 定时器结构类似，如图 10.5 所示。

同样，WATCHDOG 定时器的 8 位预分频器将 PCLK 分频后，被再次分频得到 4 种频率：16 分频、32 分频、64 分频、128 分频，WATCHDOG 定时器可以选择工作于哪种频率之下。WTCNT 寄存器按照其工作频率减 1 计数，当达到 0 时，可以产生中断信号，可以输出复位

信号。在第一次使用 WATCHDOG 定时器时，需要往 WTCNT 寄存器中写入初始计数值，以后在计数值到达 0 时自动从 WATDAT 寄存器中装入，重新开始下一个计数周期。

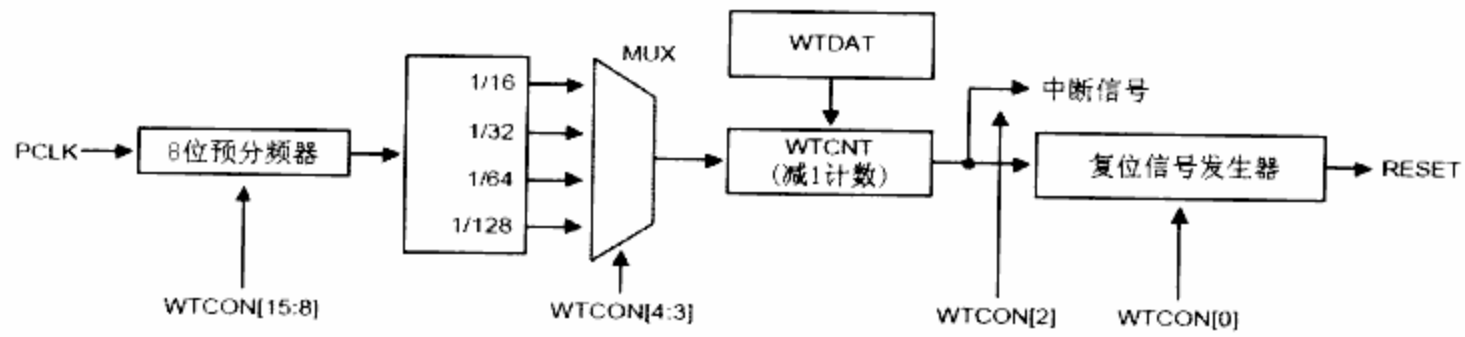


图 10.5 WATCHDOG 定时器结构图

使用 WATCHDOG 定时器的“WATCHDOG 功能”时，在正常的程序中，必须不断重新设置 WTCNT 寄存器使得它不为 0，这样可以保证系统不被重启，这称为“喂狗”；当程序崩溃时不能正常“喂狗”，计数值达到 0 后系统将被重启，这样程序将重新运行。为了克服各种干扰、避免各类系统错误时系统彻底死机，经常使用 WATDOG 功能。

WATCHDOG 定时器所涉及的寄存器如下。

（1）WTCON 寄存器（WATCHDOG TIMER CONTROL）。

用于设置预分频系数，选择工作频率，决定是否使能中断、是否启用 WATDOG 功能（即是否输出复位信号），各位的作用如表 10.4 所示。

表 10.4 WTCON 寄存器格式

功　能	位	说　明
WATCHDOG 功能	0	当定时器达到 0 时，0：不输出复位信号；1：输出复位信号
中断使能	1	0：禁止中断；1：使能中断
时钟选择	[4:3]	选择分频系数： 0b00: 16，0b01: 32 0b10: 64，0b11: 128
定时器启动	5	0：停止；1：启动
预分频系数	[15:8]	预分频系数，0～255

与 PWM 定时器相似，WATDOG 定时器的工作频率可以通过这个公式计算：

WATDOG 定时器工作频率 = PCLK / {prescaler value+1} / {divider value}

{prescaler value} = 0～255

{divider value} = 16, 32, 64, 128

（2）WTDAT 寄存器（WATCHDOG TIMER DATA）。

WTDAT 寄存器被用来决定 WATCHDOG 定时器的超时周期，在定时器启动后，当计数达到 0 时，WTDAT 寄存器的值会自动传入 WTCNT 寄存器。不过，第一次启动 WATDOG 定时器时，WTDAT 寄存器的值会自动传入 WTCNT 寄存器。

（3）WTCNT 寄存器（WATCHDOG TIMER COUNT）。

在启动 WATDOG 定时器前，必须往这个寄存器写入初始计数值。启动定时器后，它减 1 计数，当计数值达到 0 时：如果中断被使能的话发出中断，如果 WATCHDOG 功能被使能的话发出复位信号，装载 WTDAT 寄存器的值并重新计数。

10.2 MPLL 和定时器操作实例

10.2.1 程序设计

本实例讲解 MPLL、定时器的使用。首先启动 MPLL 提高系统时钟，初始化存储控制器使 SDRAM 工作在新的 HCLK 下，然后将定时器 0 设为 0.5s 产生一次中断，在中断程序里改变 LED 的状态。

10.2.2 代码详解

源码在/work/hardware/timer 目录下。

本实验的重点在 4 点：设置/启动 MPLL、根据 HCLK 设置存储控制器、初始化定时器 0、定时器中断。相关的函数都在 init.c 文件中。

1. 设置/启动 MPLL

clock_init 函数用于设置 MPLL，本开发板的输入时钟频率 Fin 为 12MHz，将 FCLK、HCLK、PCLK 分别设为 200MHz、100MHz 和 50MHz，代码如下：

```
23 #define S3C2410_MPLL_200MHZ     ((0x5c<<12)|(0x04<<4)|(0x00))
24 #define S3C2440_MPLL_200MHZ     ((0x5c<<12)|(0x01<<4)|(0x02))
25 /*
26  * 对于MPLLCON寄存器，[19:12]为MDIV，[9:4]为PDIV，[1:0]为SDIV
27  * 有如下计算公式:
28  *  S3C2410: MPLL(FCLK) = (m * Fin)/(p * 2^s)
29  *  S3C2440: MPLL(FCLK) = (2 * m * Fin)/(p * 2^s)
30  *  其中: m = MDIV + 8, p = PDIV + 2, s = SDIV
31  * 对于本开发板，Fin = 12MHz
32  * 设置CLKDIVN，令分频比为: FCLK:HCLK:PCLK=1:2:4
33  * FCLK=200MHz,HCLK=100MHz,PCLK=50MHz
34  */
35 void clock_init(void)
36 {
37     // LOCKTIME = 0x00ffffff;   // 使用默认值即可
38     CLKDIVN  = 0x03;            // FCLK:HCLK:PCLK=1:2:4, HDIVN=1,PDIVN=1
39
40     /* 如果HDIVN非0,CPU的总线模式应该从" fast bus mode" 变为" asynchronous bus
mode"  */
41 __asm__(
42     "mrc    p15, 0, r1, c1, c0, 0\n"        /* 读出控制寄存器 */
```

```
43     "orr   r1, r1, #0xc0000000\n"          /* 设置为" asynchronous bus mode"  */
44     "mcr    p15, 0, r1, c1, c0, 0\n"        /* 写入控制寄存器 */
45     );
46
47     /* 判断是 S3C2410 还是 S3C2440 */
48     if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
49     {
50         MPLLCON = S3C2410_MPLL_200MHZ;  /* 现在，FCLK=200MHz,HCLK=100MHz,
P  CLK=50MHz */
51     }
52     else
53     {
54         MPLLCON = S3C2440_MPLL_200MHZ;  /* 现在，FCLK=200MHz,HCLK=100MHz,
P  CLK=50MHz */
55     }
56 }
57
```

第 38 行设置 FCLK、HCLK、PCLK 三者分频比为 1:2:4。

当 HDIVN 非 0 时，需要将 CPU 总线模式从“fast bus mode”设为“asynchronous bus mode”，第 41～45 行的汇编代码即完成此事。

第 48 行判断是 S3C2410 还是 S3C2440，它们的 MPLL 计算公式稍有不同，需要区分开来。

如果处理器为 S3C2410，使用第 50 行设置 MPLL 寄存器，令 MDIV = 0x5c，PDIV = 0x04，SDIV = 0，所以：

```
MPLL(FCLK) = (m * Fin)/(p * 2^s) = (0x5c+8)*12MHz/((0x04+2)*2^0) = 200MHz
HCLK = FCLK / 2 = 100MHz
PCLK = FCLK / 4 = 50MHz
```

如果处理器为 S3C2440，使用第 54 行设置 MPLL 寄存器，使用第 29 行的公式可以计算出 MPLL=200MHz，所以 FCLK、HCLK、PCLK 分别为 200MHz、100MHz 和 50MHz。

2. 设置存储控制器

memsetup 函数被用来设置存储控制器，代码如下：

```
58 /*
59  * 设置存储控制器以使用 SDRAM
60  */
61 void memsetup(void)
62 {
```

```
63    volatile unsigned long *p = (volatile unsigned long *)MEM_CTL_BASE;
64
65    /* 这个函数之所以这样赋值，而不是像前面的实验(比如 mmu 实验)那样将配置值
66     * 写在数组中，是因为要生成位置无关的代码，使得这个函数可以在被复制到
67     * SDRAM 之前就可以在 steppingstone 中运行
68     */
69    /* 存储控制器 13 个寄存器的值 */
70    p[0] = 0x22011110;     //BWSCON
71    p[1] = 0x00000700;     //BANKCON0
72    p[2] = 0x00000700;     //BANKCON1
73    p[3] = 0x00000700;     //BANKCON2
74    p[4] = 0x00000700;     //BANKCON3
75    p[5] = 0x00000700;     //BANKCON4
76    p[6] = 0x00000700;     //BANKCON5
77    p[7] = 0x00018005;     //BANKCON6
78    p[8] = 0x00018005;     //BANKCON7
79
80    /* REFRESH,
81     * HCLK=12MHz:  0x008C07A3,
82     * HCLK=100MHz: 0x008C04F4
83     */
84    p[9]  = 0x008C04F4;
85    p[10] = 0x000000B1;     //BANKSIZE
86    p[11] = 0x00000030;     //MRSRB6
87    p[12] = 0x00000030;     //MRSRB7
88 }
89
```

除 REFRESH 寄存器外，其他寄存器的值与第 6 章的实验程序一样。现在 HCLK 等于 100MHz， REFRESH 寄存器的值需要重新计算。参考第 6 章的公式可以计算：R_CNT = 2^11 + 1 - 100MHz * 7.8125uS = 0x04F4，所以 REFRESH = 0x008C0000 + R_CNT = 0x008C0000 + 0x04F4 = 0x008C04F4。公式如下：

```
REFRESH = 0x008C0000 + R_CNT
R_CNT   = 2^11 + 1 - SDRAM 时钟频率(MHz) * SDRAM 刷新周期(uS)
```

第 65～68 行解释了为什么使用第 70～87 行这样笨拙的方式“手工”地一个个设置存储控制器的 13 个寄存器。在连接脚本 timer.lds 中，全部代码的起始运行地址都被设为 0x30000000，但是在执行 memsetup 函数时，代码还在内部 SRAM（steppingstone）中，为了能够在 steppingstone 运行这个函数，它应该是“位置无关”的，上面的“手工赋值”可以达到这个目的。

3. 初始化定时器 0

timer0_init 函数用于初始化定时器 0，根据相关寄存器的格式并参考代码中的注释就可以理解这个函数，代码如下：

```
124 /*
125  * Timer input clock Frequency = PCLK / {prescaler value+1} / {divider value}
126  * {prescaler value} = 0~255
127  * {divider value} = 2, 4, 8, 16
128  * 本实验的 Timer0 的时钟频率=100MHz/(99+1)/(16)=62500Hz
129  * 设置 Timer0 0.5s 触发一次中断
130  */
131 void timer0_init(void)
132 {
133     TCFG0  = 99;          // 预分频器 0 = 99
134     TCFG1  = 0x03;        // 选择 16 分频
135     TCNTB0 = 31250;       // 0.5s 触发一次中断
136     TCON   |= (1<<1);     // 手动更新
137     TCON   = 0x09;        // 自动加载，清除“手动更新”位，启动定时器 0
138 }
139
```

4. 定时器中断

head.S 中调用 timer0_init 函数之后，定时器 0 即开始工作；调用 init_irq 函数使能定时器 0 中断、设置 CPSR 寄存器开启 IRQ 中断后，每当定时器 0 计数达到 0 时就会触发中断。init_irq 函数很简单，在 init.c 中，代码如下：

```
140 /*
141  * 定时器 0 中断使能
142  */
143 void init_irq(void)
144 {
145     // 定时器 0 中断使能
146     INTMSK   &= (~(1<<10));
147 }
```

发生定时器中断时，CPU 将调用其中断服务程序 Timer0_Handle，它在 interrupt.c 中：

```
03 void Timer0_Handle(void)
04 {
05     /*
```

```
06      * 每次中断令 4 个 LED 改变状态
07      */
08     if(INTOFFSET == 10)
09     {
10         GPBDAT = ~(GPBDAT & (0xf << 5));
11     }
12     //清除中断
13     SRCPND = 1 << INTOFFSET;
14     INTPND = INTPND;
15 }
```

定时器 0 的中断使用 SRCPND、INTPND 寄存器中的位 10 来表示。中断服务程序 Timer0_Handle 先判断是否定时器 0 的中断，若是则反转 4 个 LED 的状态。

10.2.3 实例测试

在 timer 目录下执行 make 命令生成 timer.bin 可执行程序，烧入 NAND Flash 之后运行，即可看到 4 个 LED 每 1s 闪烁 1 次。

将 head.S 中对 clock_init 函数的调用去掉，不启用 MPLL；并随之将 init.c 中的 memsetup 函数的 REFRESH 寄存器改为 12MHz 对应的 0x008C07A3。重新编译、烧写，可以看到差不多 8s 这 4 个 LED 才闪烁 1 次。

第 11 章　通用异步收发器 UART

本章目标

- 了解 UART 的原理
- 掌握 S3C2410/S3C2440 中 UART 的使用

11.1　UART 原理及 UART 部件使用方法

11.1.1　UART 原理说明

通用异步收发器简称 UART，即“Universal Asynchronous Receiver Transmitter”，它用来传输串行数据：发送数据时，CPU 将并行数据写入 UART，UART 按照一定的格式在一根电线上串行发出；接收数据时，UART 检测另一根电线上的信号，将串行收集放在缓冲区中，CPU 即可读取 UART 获得这些数据。UART 之间以全双工方式传输数据，最精简的连线方法只有 3 根电线：TxD 用于发送数据，RxD 用于接收数据，Gnd 用于给双方提供参考电平，连线如图 11.1 所示。

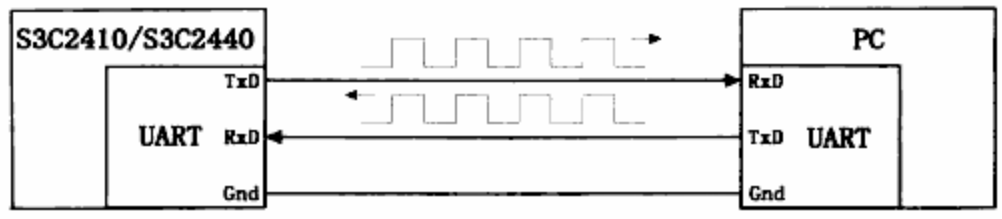


图 11.1　UART 连线图

UART 使用标准的 TTL/CMOS 逻辑电平（0～5V、0～3.3V、0～2.5V 或 0～1.8V）来表示数据，高电平表示 1，低电平表示 0。为了增强数据的抗干扰能力、提高传输长度，通常将 TTL/CMOS 逻辑电平转换为 RS-232 逻辑电平，3～12V 表示 0，−3～−12V 表示 1。

TxD、RxD 数据线以“位”为最小单位传输数据。帧（frame）由具有完整意义的、不可分割的若干位组成，它包含开始位、数据位、校验位（需要的话）和停止位。发送数据之前，UART 之间要约定好数据的传输速率（即每位所占据的时间，其倒数称为波特率）、数据的传输格式（即有多少个数据位、是否使用校验位、是奇校验还是偶校验、有多少个停止位）。

数据传输流程如下。

（1）平时数据线处于“空闲”状态（1 状态）。

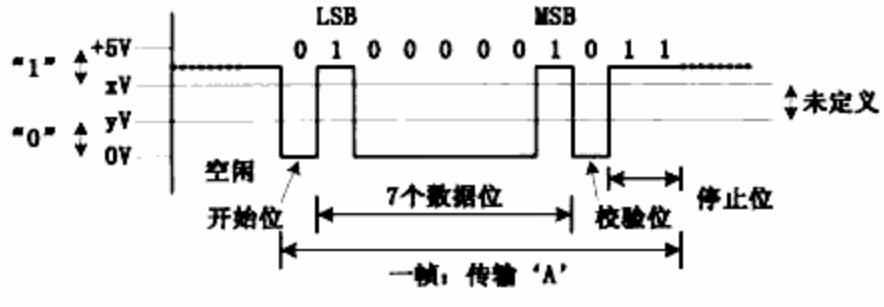
（2）当要发送数据时，UART 改变 TxD 数据线的状态（变为 0 状态）并维持 1 位的时间，这样接收方检测到开始位后，再等待 1.5 位的时间就开始一位一位地检测数据线的状态得到所传输的数据。

（3）UART 一帧中可以有 5、6、7 或 8 位的数据，发送方一位一位地改变数据线的状态将它们发送出去，首先发送最低位。

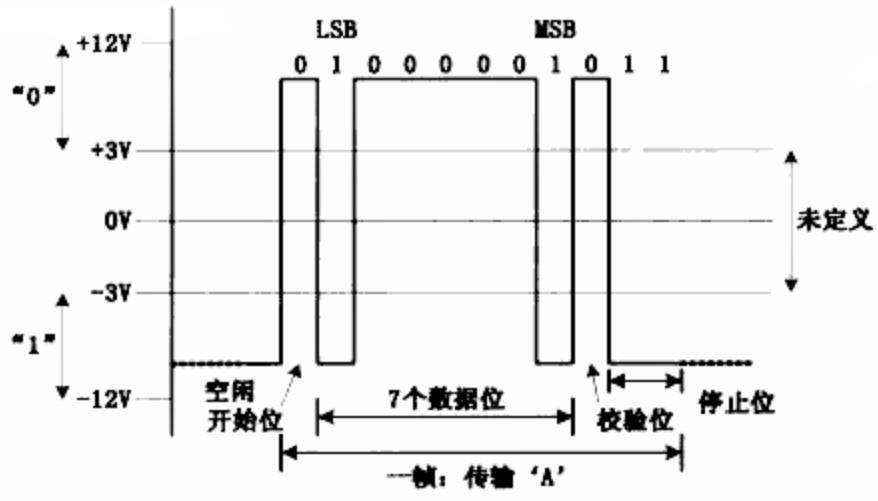
（4）如果使用校验功能，UART 在发送完数据位后，还要发送 1 个校验位。有两种校验方法：奇校验、偶校验——数据位连同校验位中，“1”的数目等于奇数或偶数。

（5）最后，发送停止位，数据线恢复到“空闲”状态（1 状态）。停止位的长度有 3 种：1 位、1.5 位、2 位。

图 11.2 演示了 UART 使用 7 个数据位、偶校验、2 个停止位的格式传输字符‘A’（二进制值为 0b1000001）时，TTL/CMOS 逻辑电平、RS-232 逻辑电平对应的波形。



1. TTL/CMOS逻辑电平下，传输‘A’时的波形



2. RS-232逻辑电平下，传输‘A’时的波形

图 11.2 UART 在 TTL/CMOS 逻辑电平和 RS-232 逻辑电平下的串行波形

11.1.2 S3C2410/S3C2440 UART 的特性

S3C2410/S3C2440 中 UART 功能相似，有 3 个独立的通道，每个通道都可以工作于中断模式或 DMA 模式，即 UART 可以发出中断或 DMA 请求以便在 UART、CPU 间传输数据。UART 由波特率发生器、发送器、接收器和控制逻辑组成。

使用系统时钟时，S3C2410 的 UART 波特率可以达到 230.4Kbit/s，S3C2440 则可以达到 115.2Kbit/s；如果使用 UEXTCLK 引脚提供的外部时钟，则可以达到更高的波特率。波特率可以通过编程进行控制。

S3C2410 UART 的每个通道都有 16 字节的发送 FIFO 和 16 字节的接收 FIFO，S3C2440

UART 的 FIFO 深度为 64。发送数据时，CPU 先将数据写入发送 FIFO 中，然后 UART 会自动将 FIFO 中的数据复制到“发送移位器”（Transmit Shifter）中，发送移位器将数据一位一位地发送到 TxDn 数据线上（根据设定的格式，插入开始位、校验位和停止位）。接收数据时，“接收移位器”（Receive Shifter）将 RxDn 数据线上的数据一位一位接收进来，然后复制到接收 FIFO 中，CPU 即可从中读取数据。

S3C2410/S3C2440 UART 的每个通道支持的停止位有 1 位、2 位，数据位有 5、6、7 或 8 位，支持校验功能，另外还有红外发送/接收功能。

S3C2410/S3C2440 UART 结构如图 11.3 所示。

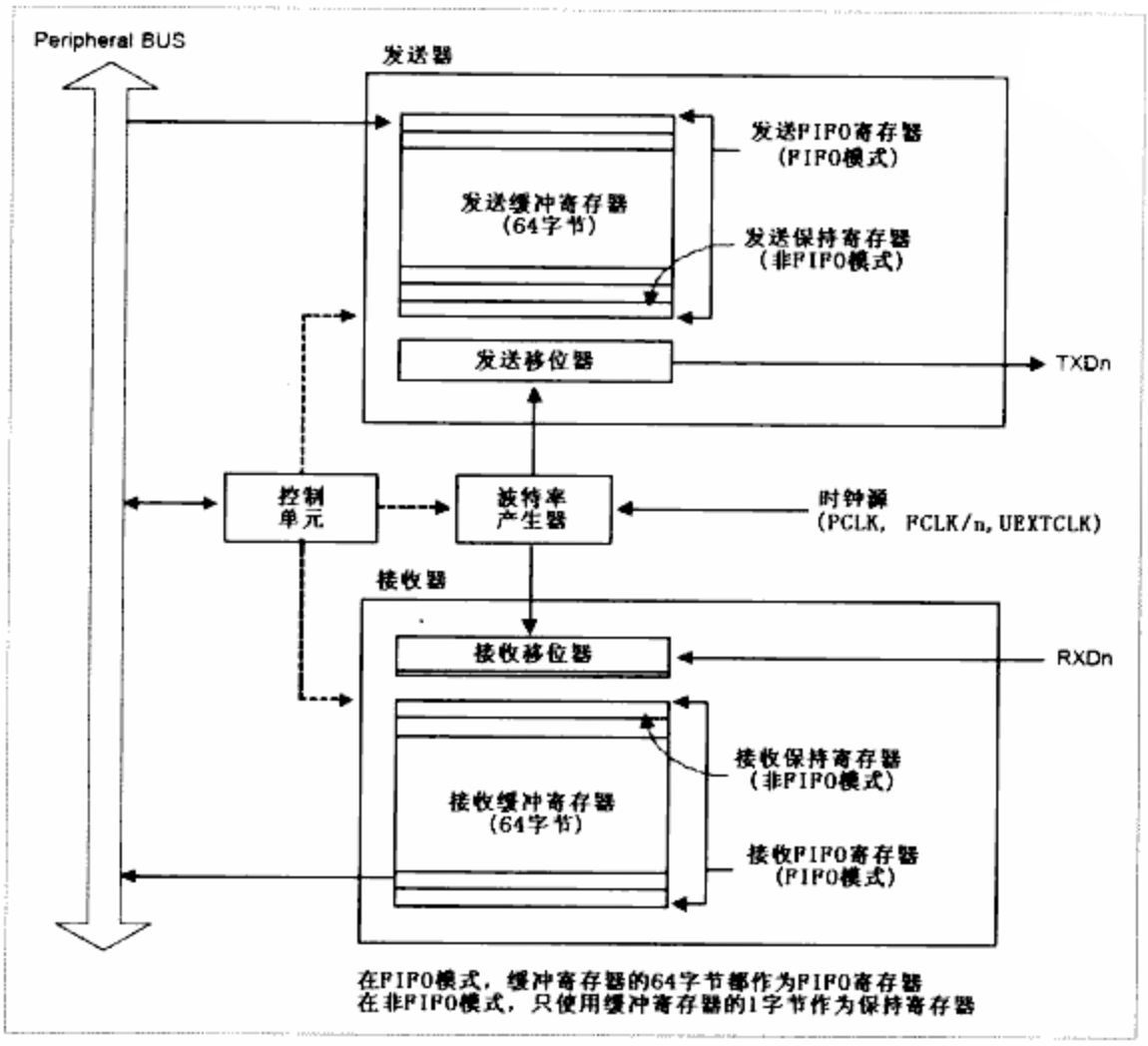


图 11.3 S3C2410/S3C2440 UART 结构图（对于 S3C2440，FIFO 深度为 64）

11.1.3 S3C2410/S3C2440 UART 的使用

在使用 UART 之前需要设置波特率、传输格式（有多少个数据位、是否使用校验位、是奇校验还是偶校验、有多少个停止位、是否使用流量控制）；对于 S3C2410/S3C2440，还要选择所涉及管脚为 UART 功能、选择 UART 通道的工作模式为中断模式或 DMA 模式。设置好之后，往某个寄存器写入数据即可发送，读取某个寄存器即可得到接收到的数据。可以通过查询状态寄存器或设置中断来获知数据是否已经发送完毕、是否已经接收到数据。

针对上述要点，下面一一说明。

1. 将所涉及的 UART 通道管脚设为 UART 功能

比如 UART 通道 0 中，GPH2、GPH3 分别用作 TXD0、RXD0，要使用 UART 通道 0 时，先设置 GPHCON 寄存器将 GPH2、GPH3 引脚的功能设为 TXD0、RXD0。

2. UBRDIVn 寄存器（UART BAUD RATE DIVISOR）：设置波特率

S3C2410 UART 的时钟源有两种选择：PCLK、UEXTCLK；S3C2440 UART 的时钟源有三种选择：PCLK、UEXTCLK、FCLK/n，其中 n 值通过 UCON0～UCON2 联合设置。

根据给定的波特率、所选择的时钟源的频率，可以通过以下公式计算 UBRDIVn 寄存器（n 为 0～2，对应 3 个 UART 通道）的值，如下所示：

```
UBRDIVn = (int)( UART clock / ( buad rate x 16) ) - 1
```

上式计算出来的 UBRDIVn 寄存器值不一定是整数，只要其误差在 1.87%之内即可。误差计算公式如下：

```
tUPCLK = (UBRDIVn + 1) x 16 x 1Frame / (UART clock)     // tUPCLK: 实际的UART
时钟
tUEXACT = 1Frame / baud-rate                            // tUEXACT: 理论的UART
时钟
UART error = (tUPCLK - tUEXACT) / tUEXACT x 100%        // 误差
```

3. ULCONn 寄存器（UART LINE CONTROL）：设置传输格式

ULCONn 寄存器（n 为 0～2）格式如表 11.1 所示。

表 11.1　　ULCONn 寄存器格式

功　能	位	说　明
数据位宽度	[1:0]	0b00：5 位，0b01：6 位 0b10：7 位，0b11：8 位
停止位宽度	[2]	0：一帧中有一个停止位 1：一帧中有两个停止位
校验模式	[5:3]	设置校验位的产生方法、检验方法。 0b0xx：无校验 0b100：奇校验 0b101：偶校验 0b110：发送数据时强制设为 1，接收数据时检查是否为 1 0b111：发送数据时强制设为 0，接收数据时检查是否为 0
红外模式	[6]	0：正常模式 1：红外模式

UART 通道被设为红外模式时，其串行数据的波形与正常模式稍有不同，有兴趣的读者请自行阅读数据手册。

4. UCONn 寄存器（UART CONTROL）

UCONn 寄存器用于选择 UART 时钟源、设置 UART 中断方式等。由于 S3C2410 的 UART 只有两个时钟源：PCLK 和 UEXTCLK；S3C2440 UART 有三个时钟源 PCLK、UEXTCLK、

FCLK/n，所以在时钟源的选择与设置（设置 n）方面稍有不同。

S3C2410 的 UCONn 寄存器格式如表 11.2 所示。

表 11.2　　S3C2410 的 UCONn 寄存器格式

功　能	位	说　明
接收模式	[1:0]	选择如何从 UART 接收缓冲区中读取数据。 0b00：禁止接收数据 0b01：中断方式或查询方式 0b10：DMA0 请求（只能用于 UART0） DMA3 请求（只能用于 UART2） 0b11：DMA1 请求（只能用于 UART1）
发送模式	[3:2]	选择如何将数据发送到 UART 发送缓冲区中。 0b00：禁止发送数据 0b01：中断方式或查询方式 0b10：DMA0 请求（只能用于 UART0） DMA3 请求（只能用于 UART2） 0b11：DMA1 请求（只能用于 UART1）
自环模式	[5]	自环模式就是将 TxDn 和 RxDn 在内部相连，用于自发自收。 0：正常模式，1：自环模式
接收错误状态中断使能	[6]	用于使能当发生接收错误（比如帧错误、溢出错误）时，产生中断。 0：出错时不产生中断 1：出错时产生中断
接收超时使能	[7]	当使用 UART FIFO 时，用于使能/禁止接收超时的中断。 0=禁止，1=使能
接收中断方式	[8]	如下情况发生时，将产生接收中断。 不使用 FIFO 时，接收到一个数据； 使用 FIFO 时，FIFO 中的数据达到 RxFIFO 的触发阈值。 中断方式如下设置。 0：脉冲 1：电平
发送中断方式	[9]	如下情况发生时，将产生发送中断。 不使用 FIFO 时，发送缓冲区变空； 使用 FIFO 时，FIFO 中的数据达到 TxFIFO 的触发阈值。 中断方式如下设置。 0：脉冲 1：电平
时钟选择	[10]	选择 UART 时钟源。 0：PCLK 1：UEXTCLK

S3C2440 的 UCONn 寄存器在 UART 时钟的选择方面与 S3C2410 有所不同，从位[10]往上的位含义不一样，并且原来的位[4]用于选择是否发出“break”信号，这些位的含义如表 11.3 所示。

表 11.3　S3C2440 的 UCONn 寄存器中与 S3C2410 的不同之处

功　能	位	说　明
“break”信号	[4]	设置此位时，UART 会在一帧的时间里面发出一个“break”信号。 0：正常发送，1：发出“break”信号
时钟选择	[11:10]	选择 UART 时钟源。 0b00/0b10：PCLK 0b01：UEXTCLK 0b11：FCLK/n
FCLK 分频率系数	[15:12]	用来设置“FCLK/n”中的 n 值

UCON0、UCON1、UCON2 这 3 个寄存器的位[15:12]一起用来确定 n 值，它们的意义如下。

（1）UCON2[15]：“FCLK/n”使能位。

它等于 0 时，禁止使用“FCLK/n”作为 UART 时钟源；等于 1 时，可以用作 UART 时钟源。

（2）n 值的设置。

UCON0[15:12]、UCON1[15:12]、UCON2[14:12]三者用于设置 n 值，当其中一个被设成非 0 值时，其他两个必须为 0。

① n 值处于 7～21 时，UART 时钟=FCLK/（divider+6），divider 为 UCON0[15:12]的值，大于 0。

② n 值处于 22～36 时，UART 时钟=FCLK/（divider+21），divider 为 UCON1[15:12]的值，大于 0。

③ n 值处于 37～43 时，UART 时钟=FCLK/（divider+36），divider 为 UCON2[14:12]的值，大于 0。

④ UCON0[15:12]、UCON1[15:12]、UCON2[14:12]都等于 0 时，UART 时钟：FCLK/44。

5．UFCONn 寄存器（UART FIFO CONTROL）、UFSTATn 寄存器（UART FIFO STATUS）

UFCONn 寄存器用于设置是否使用 FIFO，设置各 FIFO 的触发阈值，即发送 FIFO 中有多少个数据时产生中断、接收 FIFO 中有多少个数据时产生中断。并可以通过设置 UFCONn 寄存器来复位各个 FIFO。

读取 UFSTATn 寄存器可以知道各个 FIFO 是否已经满、其中有多少个数据。

不使用 FIFO 时，可以认为 FIFO 的深度为 1，使用 FIFO 时 S3C2410 的 FIFO 深度为 16，S3C2440 的 FIFO 深度为 64。这两类寄存器各位的含义请读者查阅数据手册，后面的实例部分没有使用 FIFO。

6．UMCONn 寄存器（UART MODEM CONTROL）、UMSTATn 寄存器（UART MODEM STATUS）

这两类寄存器用于流量控制，这里不介绍。

7. UTRSTATn 寄存器（UART TX/RX STATUS）

UTRSTATn 寄存器用来表明数据是否已经发送完毕、是否已经接收到数据，格式如表 11.4 所示。缓冲区其实就是图 11.3 中的 FIFO，只不过不使用 FIFO 功能时可以认为其深度为 1。

表 11.4　　UTRSTATn 寄存器格式

功　能	位	说　明
接收缓冲区数据就绪	[0]	当接收到数据时，此位被自动设为 1
发送缓冲区空	[1]	当发送缓冲区中没有数据时，此位被自动设为 1
发送器空	[2]	当发送缓冲区中没有数据，并且最后一个数据也已经发送出去时，此位被自动设为 1

8. UERSTATn 寄存器（UART ERROR STATUS）

用来表示各种错误是否发生，位[0]～位[3]为 1 时分别表示溢出错误、校验错误、帧错误、检测到“break”信号。读取这个寄存器时，它会自动清 0。

需要注意的是，接收数据时如果使用 FIFO，则 UART 内部会使用一个“错误 FIFO”来表明接收 FIFO 中哪个数据在接收过程中发生了错误。CPU 只有在读出这个错误的数据时，才会觉察到发生了错误。要想清除“错误 FIFO”，则必须读出错误的数据，并读出 UERSTATn 寄存器。

9. UTXHn 寄存器（UART TRANSMIT BUFFER REGISTER）

CPU 将数据写入这个寄存器，UART 即会将它保存到缓冲区中，并自动发送出去。

10. URXHn 寄存器（UART RECEIVE BUFFER REGISTER）

当 UART 接收到数据时，CPU 读取这个寄存器，即可获得数据。

11.2 UART 操作实例

11.2.1 代码详解

本实例代码在/work/hardware/uart 目录下。目的是在串口上输入一个字符，单板接收到后将它的 ASCII 值加 1 后，从串口输出。

首先设置 MPLL 提高系统时钟，令 PCLK 为 50MHz，UART 将选择 PCLK 为时钟源。将代码复制到SDRAM中之后，调用main函数。这些代码与第10章相似。重点在于main函数对UART0的初始化、收发数据，这由 3 个函数来实现：uart0_init、getc 和 putc，它们在 serial.c 文件中。

1. UART 初始化

uart0_init 函数代码如下，每行都有注释，结合上面的寄存器用法即可理解。

```
07 #define PCLK            50000000     // init.c 中的 clock_init 函数设置 PCLK
为 50MHz
08 #define UART_CLK        PCLK         // UART0 的时钟源设为 PCLK
09 #define UART_BAUD_RATE  115200       // 波特率
10 #define UART_BRD        ((UART_CLK   / (UART_BAUD_RATE * 16)) - 1)
11
12 /*
13  * 初始化 UART0
14  * 115200,8N1,无流控
15  */
16 void uart0_init(void)
17 {
18     GPHCON  |= 0xa0;        // GPH2、GPH3 用作 TXD0、RXD0
19     GPHUP   = 0x0c;         // GPH2、GPH3 内部上拉
20
21     ULCON0  = 0x03;         // 波特率为 115200，数据格式为:8 个数据位，没有流控，1
个数据位
22     UCON0   = 0x05;         // 查询方式，UART 时钟源为 PCLK
23     UFCON0  = 0x00;         // 不使用 FIFO
24     UMCON0  = 0x00;         // 不使用流控
25     UBRDIV0 = UART_BRD;     // 波特率为 115200
26 }
27
```

2. 发送字符的函数

本实例不使用 FIFO，发送字符前，首先判断上一个字符是否已经被发送出去。如果没有，则不断查询 UTRSTAT0 寄存器的位[2]，当它为 1 时表示已经发送完毕。于是，即可向 UTXH0 寄存器中写入当前要发送的字符。代码如下（宏 TXD0READY 被定义为“(1<<2)”）：

```
28 /*
29  * 发送一个字符
30  */
31 void putc(unsigned char c)
32 {
33     /* 等待，直到发送缓冲区中的数据已经全部发送出去 */
34     while (!(UTRSTAT0 & TXD0READY));
35
36     /* 向 UTXH0 寄存器中写入数据，UART 即自动将它发送出去 */
37     UTXH0 = c;
```

```
38 }
39
```

3. 接收字符的函数

试图读取数据前，先查询 UTRSTAT0 寄存器的位[1]，当它为 1 时表示接收缓冲区中有数据。于是，即可读取 URXH0 得到数据。代码如下（宏 RXD0READY 被定义为“(1)”）：

```
40 /*
41  * 接收字符
42  */
43 unsigned char getc(void)
44 {
45     /* 等待，直到接收缓冲区中的有数据 */
46     while (!(UTRSTAT0 & RXD0READY));
47
48     /* 直接读取 URXH0 寄存器，即可获得接收到的数据 */
49     return URXH0;
50 }
51
```

4. 主函数

在 main 函数中，初始化完 UART0 之后，即不断地读取串口数据，并判断它是否为数字或字母。如果是的话，就将它加 1 后从串口输出。代码如下：

```
01 #include "serial.h"
02
03 int main()
04 {
05     unsigned char c;
06     uart0_init();   // 波特率 115200，8N1(8 个数据位，无校验位，1 个停止位)
07
08     while(1)
09     {
10         // 从串口接收数据后，判断其是否为数字或字母，若是则加 1 后输出
11         c = getc();
12         if (isDigit(c) || isLetter(c))
13             putc(c+1);
14     }
15
16     return 0;
17 }
```

11.2.2 实例测试

1. PC 上的串口工具推荐

Windows 下推荐使用 SecureCRT 工具，Linux 下推荐使用 kermit。

下面介绍在 Linux 下安装、使用 kermit 的方法。

确保 Linux 能连上网络，然后使用以下命令进行安装，会安装一个 kermit 命令。

```
$ sudo apt-get install ckermit
```

使用 kermit 之前，先建立一个配置文件，在/homt/book（假设用户名为 book）目录下创建名为.kermrc 的文件，内容如下：

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

然后，运行“$ sudo kermit -c”命令即可启动串口；要想关闭串口，可以先输入“Ctrl+\”，然后按住“C”键，最后输入“exit”后回车。

2. 测试方法

首先使用串口将开发板的 COM0 和 PC 的串口相连，打开 PC 上的串口工具并设置其波特率为 115200、8N1。

然后，在 uart 目录下运行 make 命令生成可执行文件 uart.bin，将它烧入 NAND Flash 后运行。

最后，在 PC 上的串口工具中输入数字或字母，可以看到输出另一个字符（加了 1）；如果输入其他字符，则无输出。

/work/hardware/stdio 目录下的程序在串口 0 上实现 printf、scanf 等函数，它使用 scanf、sscanf 和 printf 等函数从串口接收一个十进制数字序列，然后将它转换为十六进制输出。步骤与 UART 实例相似，读者可自行操作。

第 12 章　I²C 接口

本章目标

- 了解 I²C 总线协议
- 掌握 S3C2410/S3C2440 中 I²C 接口的使用方法

12.1　I²C 总线协议及硬件介绍

12.1.1　I²C 总线协议

1. I²C 总线的概念

I²C（Inter-Integrated Circuit，又称 IIC）总线是一种由 PHILIPS 公司开发的串行总线，用于连接微控制器及其外围设备，它具有如下特点。

- 只有两条总线线路：一条串行数据线（SDA），一条串行时钟线（SCL）。
- 每个连接到总线的器件都可以使用软件根据它的惟一的地址来识别。
- 传输数据的设备间是简单的主/从关系。
- 主机可以用作主机发送器或主机接收器。
- 它是一个真正的多主机总线，两个或多个主机同时发起数据传输时，可以通过冲突检测和仲裁来防止数据被破坏。
- 串行的 8 位双向数据传输，位速率在标准模式下可达 100kbit/s，在快速模式下可达 400kbit/s，在高速模式下可达 3.4Mbit/s。
- 片上的滤波器可以增加抗干扰功能，保证数据的完整。
- 连接到同一总线上的 IC 数量只受到总线的最大电容 400pF 的限制。

图 12.1 是一条 I²C 总线上多个设备相连的例子。

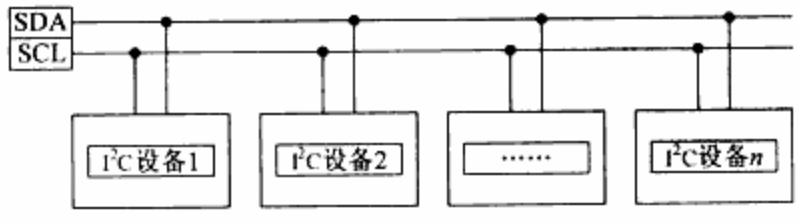


图 12.1　I²C 总线设备互连

先说明一些术语，如表 12.1 所示。

表 12.1 I²C 总线术语的定义

术 语	描 述
发送器	发送数据到总线的器件
接收器	从总线接收数据的器件
主机	发起/停止数据传输、提供时钟信号的器件
从机	被主机寻址的器件
多主机	可以有多个主机试图去控制总线，但是不会破坏数据
仲裁	当多个主机试图去控制总线时，通过仲裁可以使得只有一个主机获得总线控制权，并且它传输的信息不被破坏
同步	多个器件同步时钟信号的过程

2. I²C 总线的信号类型

I²C 总线在传送数据过程中共有 3 种类型信号：开始信号、结束信号和响应信号。

（1）开始信号（S）：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

（2）结束信号（P）：SCL 为低电平时，SDA 由低电平向高电平跳变，结束传送数据。

（3）响应信号（ACK）：接收器在接收到 8 位数据后，在第 9 个时钟周期，拉低 SDA 电平。

它们的波形如图 12.2、12.3 所示。

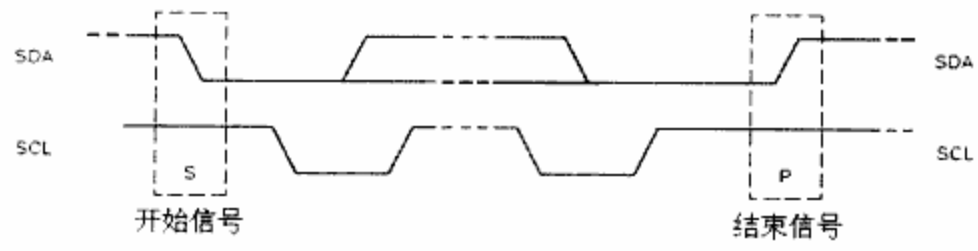


图 12.2 开始信号（S）和结束信号（P）

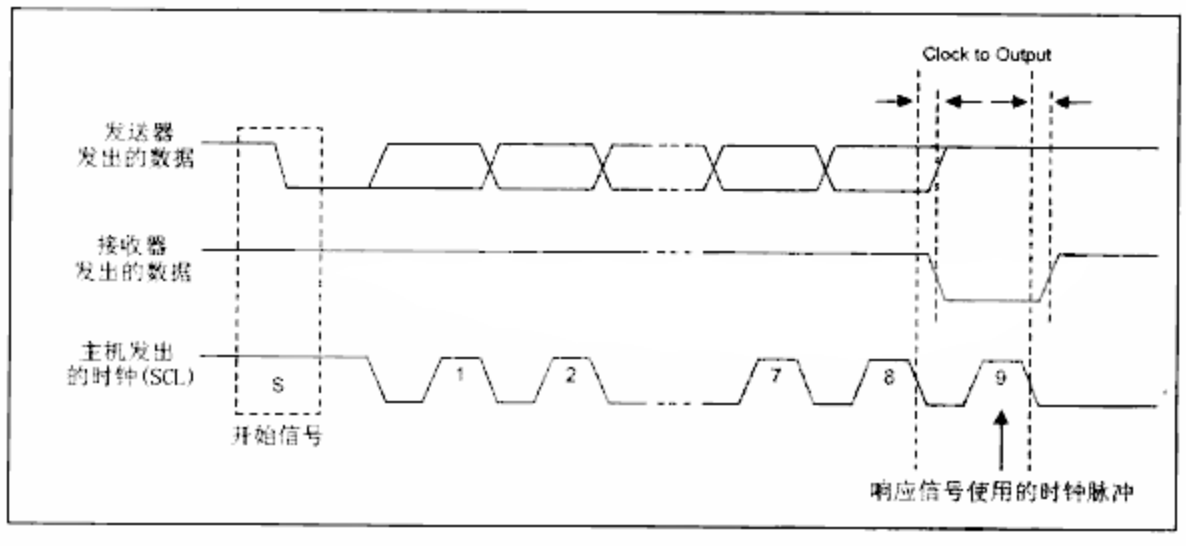


图 12.3 应答信号（ACK）

SDA 上传输的数据必须在 SCL 为高电平期间保持稳定，SDA 上的数据只能在 SCL 为低电平期间变化，如图 12.4 所示。

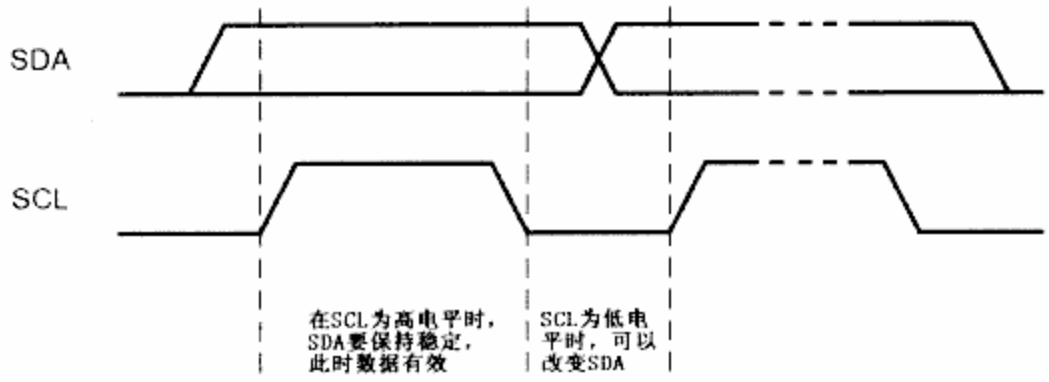


图 12.4 I²C 总线的位传输

3. I²C 总线的数据传输格式

发送到 SDA 线上的每个字节必须是 8 位的，每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位。首先传输的是数据的最高位（MSB）。如果从机要完成一些其他功能后（例如一个内部中断服务程序）才能继续接收或发送下一个字节，从机可以拉低 SCL 迫使主机进入等待状态。当从机准备好接收下一个数据并释放 SCL 后，数据传输继续。如果主机在传输数据期间也需要完成一些其他功能（例如一个内部中断服务程序）也可以拉低 SCL 以占住总线。

启动一个传输时，主机先发出 S 信号，然后发出 8 位数据。这 8 位数据中前 7 位为从机的地址，第 8 位表示传输的方向（0 表示写操作，1 表示读操作）。被选中的从机发出响应信号。紧接着传输一系列字节及其响应位。最后，主机发出 P 信号结束本次传输。

图 12.5 是几种 I²C 总线上数据传输的格式。

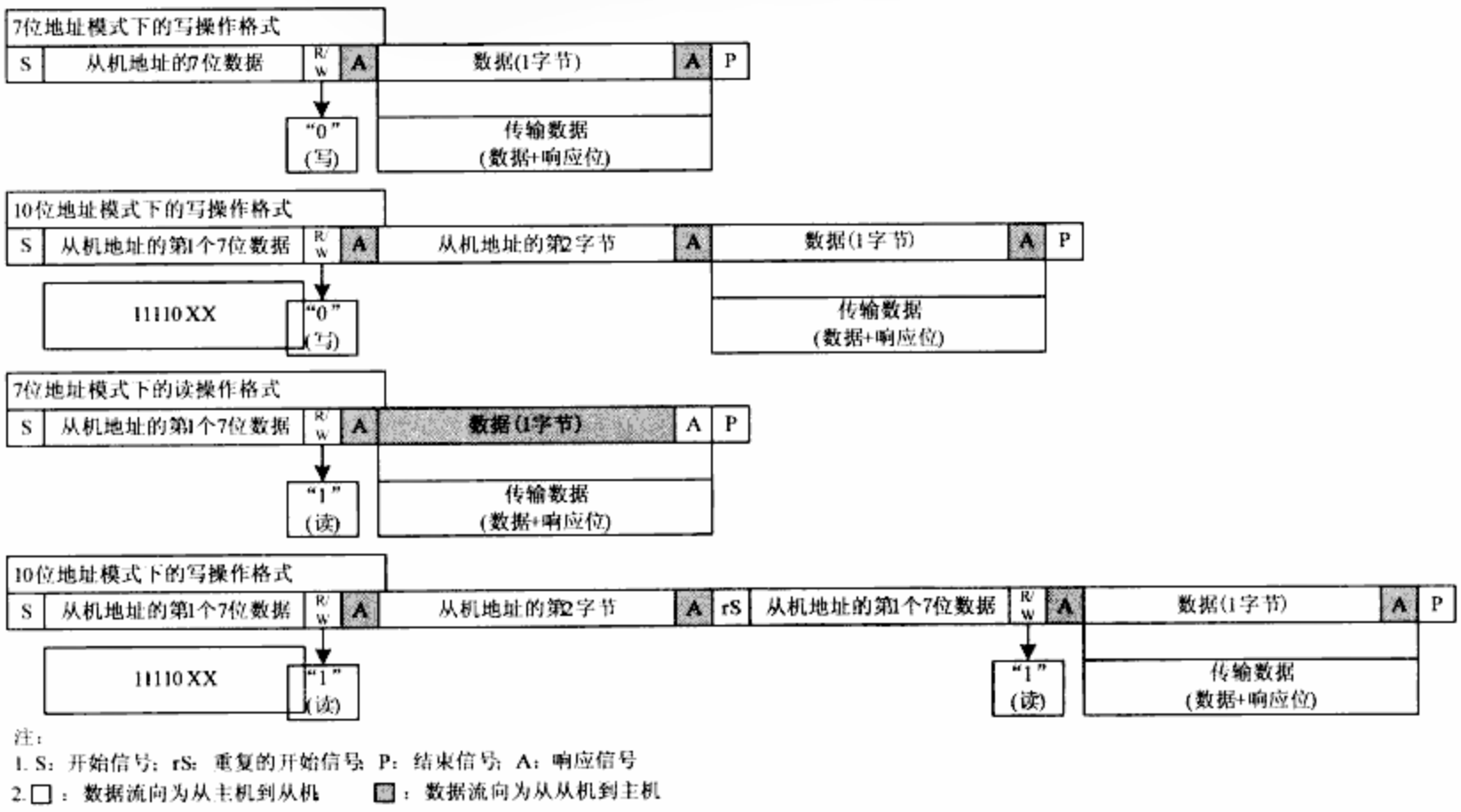


图 12.5 I²C 总线上数据传输的格式

并非每传输 8 位数据之后，都会有 ACK 信号，有以下 3 种例外。

（1）当从机不能响应从机地址时（例如它正忙于其他事而无法响应 I²C 总线的操作，或者这个地址没有对应的从机），在第 9 个 SCL 周期内 SDA 线没有被拉低，即没有 ACK 信号。这时，主机发出一个 P 信号终止传输或者重新发出一个 S 信号开始新的传输。

（2）如果从机接收器在传输过程中不能接收更多的数据时，它也不会发出 ACK 信号。这样，主机就可以意识到这点，从而发出一个 P 信号终止传输或者重新发出一个 S 信号开始新的传输。

（3）主机接收器在接收到最后一个字节后，也不会发出 ACK 信号。于是，从机发送器释放 SDA 线，以允许主机发出 P 信号结束传输。

12.1.2 S3C2410/S3C2440 I²C 总线控制器

1. S3C2410/S3C2440 I²C 总线控制器寄存器介绍

S3C2410/S3C2440 的 I²C 接口有 4 种工作模式：主机发送器、主机接收器、从机发送器、从机接收器。其内部结构如图 12.6 所示。

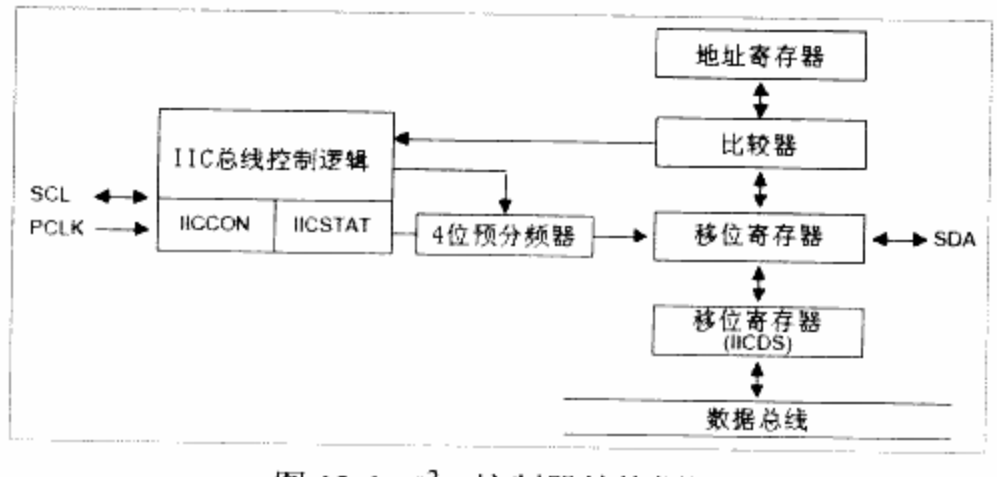


图 12.6 I²C 控制器结构框图

从图 12.6 可知，S3C2410/S3C2440 提供 4 个寄存器来完成所有的 I²C 操作。SDA 线上的数据从 IICDS 寄存器发出，或传入 IICDS 寄存器中；IICADD 寄存器中保存 S3C2410/S3C2440 当作从机时的地址；IICCON、IICSTAT 两个寄存器用来控制或标识各种状态，比如选择工作模式，发出 S 信号、P 信号，决定是否发出 ACK 信号，检测是否收到 ACK 信号。各寄存器的用法如下。

（1）IICCON 寄存器（Multi-master IIC-bus control）。

IICCON 寄存器用于控制是否发出 ACK 信号、设置发送器的时钟、开启 I²C 中断，并标识中断是否发生。它的各位含义如表 12.2 所示。

表 12.2 IICCON 寄存器的格式

功　能	位	说　明
ACK 信号使能	[7]	0 = 禁止，1 = 使能 在发送模式，此位无意义； 在接收模式，此位使能时，SDA 线在响应周期内将被拉低，即发出 ACK 信号
发送模式时钟源选择	[6]	0 = IICCLK 为 PCLK/16，1 = IICCLK 为 PCLK/512
发送/接收中断使能	[5]	0 = I²C 总线 Tx/Rx 中断禁止， 1 = I²C 总线 Tx/Rx 中断使能
中断标记	[4]	此位用来标识是否有 I²C 中断发生，读出为 0 时表示没有中断发生，读出为 1 时表示有中断发生。当此位为 1 时，SCL 线被拉低，此时所有 I²C 传输停止；如果要继续传输，需写入 0 清除它
发送模式时钟分频系数	[3:0]	发送器时钟 = IICCLK /（IICCON[3:0] + 1）

使用 IICCON 寄存器时，有如下注意事项。

① 发送模式的时钟频率由位[6]、位[3:0]联合决定。另外，当 IICCON[6]=0 时，IICCON[3:0]不能取 0 或 1。

② I²C 中断在以下 3 种情况下发生：当发出地址信息或接收到一个从机地址并且吻合时，当总线仲裁失败时，当发送/接收完一个字节的数据（包括响应位）时。

③ 基于 SDA、SCL 线上时间特性的考虑，要发送数据时，先将数据写入 IICDS 寄存器，然后再清除中断。

④ 如果 IICCON[5]=0，IICCON[4]将不能正常工作。所以，即使不使用 I²C 中断，也要将 IICCON[5]设为 1。

（2）IICSTAT 寄存器（Multi-master IIC-bus control/status）。

IICSTAT 寄存器用于选择 I²C 接口的工作模式，发出 S 信号、P 信号，使能接收/发送功能，并标识各种状态，比如总线仲裁是否成功、作为从机时是否被寻址、是否接收到 0 地址、是否接收到 ACK 信号等。

IICSTAT 寄存器的各位如表 12.3 所示。

表 12.3　IICSTAT 寄存器的格式

功　能	位	说　明
工作模式	[7:6]	0b00：从机接收器 0b01：从机发送器 0b10：主机接收器 0b11：主机发送器
忙状态位/S 信号、P 信号	[5]	读此位时 0：总线空闲，1：总线忙 写此位时 0：发出 P 信号，1：发出 S 信号。当发出 S 信号后，IICDS 寄存器中的数据将被自动发送。
串行输出使能位	[4]	0：禁止接收/发送功能，1：使能接收/发送功能
仲裁状态	[3]	0：总线仲裁成功，1：总线仲裁失败
从机地址状态	[2]	作为从机时，在检测到 S/P 信号时此位被自动清 0； 接收到的地址与 IICADD 寄存器中的值相等时，此位被置 1
0 地址状态	[1]	在检测到 S/P 信号时此位被自动清 0； 接收到的地址为 0b0000000 时，此位被置 1
最后一位的状态	[0]	0：接收到的最后一位为 0（接收到 ACK 信号）； 1：接收到的最后一位为 1（没有接收到 ACK 信号）

（3）IICADD 寄存器（Multi-master IIC-bus address）。

用到 IICADD 寄存器的位[7:1]，表示从机地址。IICADD 寄存器在串行输出使能位 IICSTAT[4]为 0 时，才可以写入；在任何时间都可以读出。

（4）IICDS 寄存器（Multi-master IIC-bus Tx/Rx data shift）。

用到 IICDS 寄存器的位[7:0]，其中保存的是要发送或已经接收的数据。IICDS 寄存器在串行输出使能位 IICSTAT[4]为 1 时，才可以写入；在任何时间都可以读出。

2. S3C2410/S3C2440 I²C 总线操作方法

启动或恢复 S3C2410/S3C2440 的 I²C 传输有以下两种方法。

（1）当 IICCON[4]即中断状态位为 0 时，通过写 IICSTAT 寄存器启动 I²C 操作。有以下两种情况。

① 在主机模式，令 IICSTAT[5:4]等于 0b11，将发出 S 信号和 IICDS 寄存器的数据（寻址），令 IICSTAT[5:4]等于 0b01，将发出 P 信号。

② 在从机模式，令 IICSTAT[4]等于 1 将等待其他主机发出 S 信号及地址信息。

（2）当 IICCON[4]即中断状态位为 1 时，表示 I²C 操作被暂停。在这期间设置好其他寄存器之后，向 IICCON[4]写入 0 即可恢复 I²C 操作。所谓“设置其他寄存器”，有以下 3 种情况。

① 对于主机模式，可以按照上面①的方法写 IICSTAT 寄存器，恢复 I²C 操作后即可发出 S 信号和 IICDS 寄存器的值（寻址），或发出 P 信号。

② 对于发送器，可以将下一个要发送的数据写入 IICDS 寄存器中，恢复 I²C 操作后即可发出这个数据。

③ 对于接收器，可以从 IICDS 寄存器中读出接收到的数据。最后向 IICCON[4]写入 0 的同时，设置 IICCON[7]以决定在接收到下一个数据后是否发出 ACK 信号。

通过中断服务程序来驱动 I²C 传输。

（1）当仲裁失败时发生中断——本次传输没有抢到总线，可以稍后继续。

（2）对于主机模式，当发出 S 信号、地址信息并经过一个 SCL 周期（对应 ACK 信号）后，发生中断——主机可在此时判断是否成功寻址到从机。

（3）对于从机模式，当接收到的地址与 IICADD 寄存器吻合时，先发出 ACK 信号，然后发生中断——从机可在此时准备后续的传输。

（4）对于发送器，当发送完一个数据并经过一个 SCL 周期（对应 ACK 信号）后，发生中断。这时可以准备下一个要发送的数据，或发出 P 信号以停止传输。

（5）对于接收器，当接收到一个数据时，先根据 IICCON[7]决定是否发出 ACK 信号后，然后发生中断。这时可以读取 IICDS 寄存器得到数据，并设置 IICCON[7]以决定接收到下一个数据后是否发出 ACK 信号。

对于 4 种工作模式，S3C2410/S3C2440 数据手册中都有它们的操作流程图。现在以主机发送器作为例子说明，它的工作流程如图 12.7 所示，其他的工作模式请读者自行查阅数据手册。

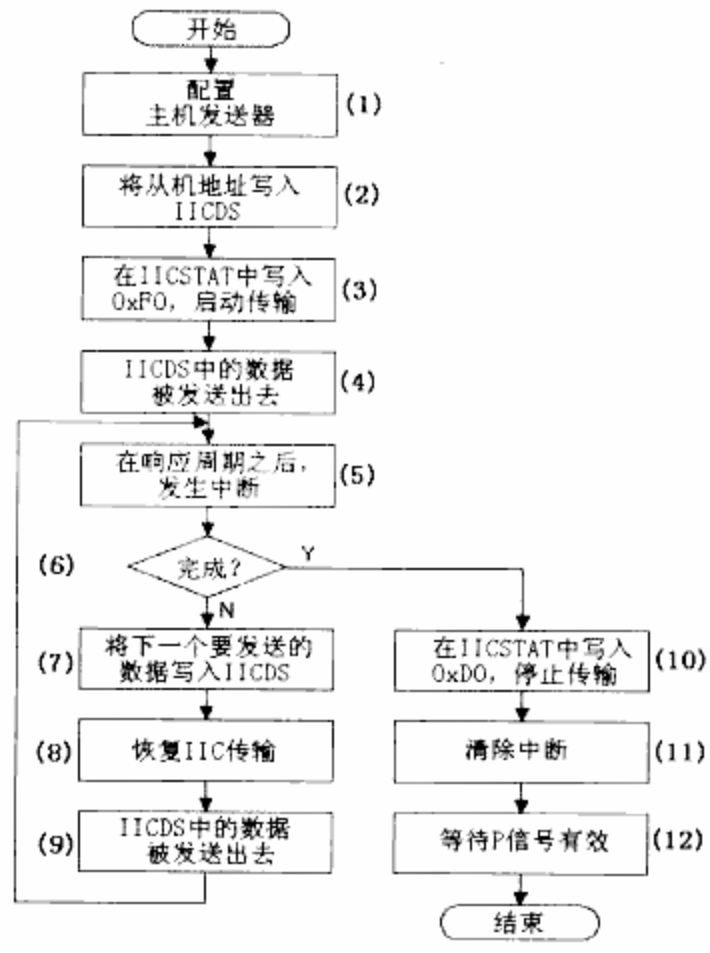


图 12.7 S3C2410/S3C2440 I²C 主机发送器的工作流程

下面结合 I²C 寄存器的用法，详细讲解图 12.7 中各步骤的含义。

（1）配置主机发送器的各类参数。

设置 GPE15、GPE14 引脚用于 SDA、SCL，设

置 IICCON 寄存器选择 I²C 发送时钟，最后，设置 IICSTAT[4]为 1，这样，后面才能写 IICDS 寄存器。

注意 初始时 IICCON[4]为 0，不能将 IICSTAT 设为主机模式，否则就会立刻发出 S 信号、发送 IICDS 寄存器的值。

（2）将要寻址的从机地址写入 IICDS 寄存器。

（3）将 0xFO 写入 IICSTAT 寄存器，即设为主机发送器、使能串行输出功能、发出 S 信号。

（4）发出 S 信号后，步骤（2）中设置的 IICDS 寄存器值也将被发出，它用来寻址从机。

（5）在响应周期之后，发生中断，此时 IICCON[4]为 1，I²C 传输暂停。

（6）如果没有数据要发送，则跳到步骤（10）；否则跳到步骤（7）。

（7）将下一个要发送的数据写入 IICDS 寄存器中。

（8）往 IICCON[4]中写入 0，恢复 I²C 传输。

（9）这时 IICDS 寄存器中的值将被一位一位地发送出去。当 8 位数据发送完毕，再经过另一个 SCL 周期（对应 ACK 信号）后，中断再次发生，跳到步骤（5）。

步骤（5）～（9）不断循环直到发出了所有数据。当要停止传输时，跳到步骤（10）。

（10）将 0xFO 写入 IICSTAT 寄存器，即：设为主机发送器、使能串行输出功能、发出 P 信号。

注意 这时的 P 信号并没有实际发出，只有清除了 IICCON[4]后才会发出 P 信号。

（11）清除 IICCON[4]，P 信号得以发出。

（12）等待一段时间，使得 P 信号完全发出。

12.2 I²C 总线操作实例

12.2.1 I²C 接口 RTC 芯片 M41t11 的操作方法

本书所用开发板中，通过 I²C 总线连接 RTC（实时时钟）芯片 M41t11，它使用电池供电，系统断电时也可以维持日期和时间。S3C2410/S3C2440 作为 I²C 主机向 M41t11 发送数据以设置日期和时间、读取 M41t11 以获得日期和时间。连接图如图 12.8 所示。

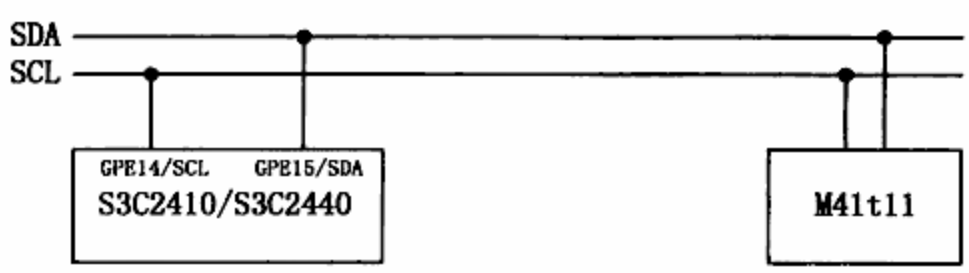


图 12.8 S3C2410/S3C2440 与 M41t11 连线图

M41t11 中有 8 个寄存器，分别对应秒、分、时、天（星期几）、日、月、年、控制寄存器，其中的数据都是以 BCD 格式保存（BCD 格式例子：0x15 表示数值 15），如表 12.4 所示。

表 12.4 M41t11 寄存器格式

地 址	数 据								功能/取值范围 BCD 编码	
	D7	D6	D5	D4	D3	D2	D1	D0		
0	ST	秒的十位数			秒的个位数				秒	00～59
1	X	分的十位数			分的个位数				分	00～59
2	CEB	CB	时的十位数		时的个位数				世纪位/时	0～1/00～23
3	X	X	X	X	X	天（星期几）			天	01～07
4	X	X	日的十位数		日的个位数				日	01～31
5	X	X	X	10 月	月的个位数				月	01～12
6	年的十位数				年的个位数				年	00～99
7	OUT	FT	S	校准					控制	

注：

①ST 为停止位，写入 1 时 M41t11 停止工作，写入 0 时开始工作。

②CEB 为“世纪使能位”，当它为 1 时，每过 100 年，CB 位就反转一次。

③“10 月”表示“月的十位数”。

④地址为 7 的寄存器有一些控制功能，本书没有使用。

除上表的 8 个寄存器（地址为 0～7）之外，M41t11 内部还有 56 字节的 RAM（地址为 8～63）。访问 M41t11 前，先设置寄存器地址，以后每次读/写操作完成后，M41t11 内部会自动将寄存器地址加 1。

所以读写 M41t11 分以下两个步骤。

（1）主机向 M41t11 发出要操作的寄存器起始地址（0～7）。

（2）要设置 M41t11 时，主机连续发出数据；要读取 M41t11 时，主机连续读取数据。

M41t11 的 I²C 从机地址为 0xD0。

12.2.2 程序设计

本实例将在串口上输出一个菜单，可以选择设置时间和日期，或者将它们读出来。将通过本实例验证 I²C 主机的发送、接收操作。

12.2.3 设置/读取 M41t11 的源码详解

本实例的源码在/work/hardware i2c 目录下。

文件 i2c.c 封装了 S3C2410/S3C2440 作为主机发送器、主机接收器的 4 个操作函数：i2c_init 用于初始化，i2c_write 用发起发送数据，i2c_read 用于发起读取数据，I2CIntHandle 是 I²C 中断服务程序，它用于完成后续的数据传输。

1. S3C2410/S3C2440 I²C 控制器初始化

i2c_init 函数对应于图 12.7 中步骤（1），它用来初始化 I²C，代码如下：

```
24 /*
25  * I²C 初始化
26  */
```

```
27 void i2c_init(void)
28 {
29     GPEUP  |= 0xc000;        // 禁止内部上拉
30     GPECON |= 0xa0000000;   // 选择引脚功能，GPE15:IICSDA, GPE14:IICSCL
31
32     INTMSK &= ~(BIT_IIC);
33
34     /* bit[7] = 1, 使能ACK
35      * bit[6] = 0, IICCLK = PCLK/16
36      * bit[5] = 1, 使能中断
37      * bit[3:0] = 0xf, Tx clock = IICCLK/16
38      * PCLK = 50MHz, IICCLK = 3.125MHz, Tx Clock = 0.195MHz
39      */
40     IICCON = (1<<7) | (0<<6) | (1<<5) | (0xf);  // 0xaf
41
42     IICADD  = 0x10;     // S3C24xx slave address = [7:1]
43     IICSTAT = 0x10;     // I²C 串行输出使能(Rx/Tx)
44 }
45
```

第 29、30 行将引脚 GPE15、GPE14 的功能选择用于 I²C：IICSDA、IICSCL。

第 32 行在 INTMSK 寄存器中开启 I²C 中断，这样，以后调用 i2c_read、i2c_write 启动传输时，即可以触发中断，进而可以在中断服务程序中进一步完成后续传输。

第 40 行用于选择发送时钟，并进行一些设置：使能 ACK、使能中断。

第 42 行用于设置 S3C2410/S3C2440 作为 I²C 从机时的地址，在本实例中没有用到。

第 43 行使能 I²C 串行输出（设置 IICSTAT[4]为 1），这样，在 i2c_write、i2c_read 函数中就可以写 IICDS 寄存器了。

2. S3C2410/S3C2440 I²C 主机发送函数

初始化完成后，就可调用 i2c_read、i2c_write 读写 I²C 从机了。它们的使用方法从参数名称即可看出。这两个函数仅仅是启动 I²C 传输，然后等待，直到数据在中断服务程序中传输完毕后再返回。

i2c_write 函数的实现如下：

```
46 /*
47  * 主机发送
48  * slvAddr : 从机地址，buf : 数据存放的缓冲区，len : 数据长度
49  */
50 void i2c_write(unsigned int slvAddr, unsigned char *buf, int len)
51 {
```

```
52      g_tS3C24xx_I²C.Mode = WRDATA;   // 写操作
53      g_tS3C24xx_I²C.Pt   = 0;        // 索引值初始为 0
54      g_tS3C24xx_I²C.pData = buf;     // 保存缓冲区地址
55      g_tS3C24xx_I²C.DataCount = len; // 传输长度
56
57      IICDS   = slvAddr;
58      IICSTAT = 0xf0;         // 主机发送，启动
59
60      /* 等待直至数据传输完毕 */
61      while (g_tS3C24xx_I²C.DataCount != -1);
62 }
63
```

第 52～55 行用于设置全局变量 g_tS3C24xx_I2C，它表明当前是写操作，并保存缓冲区地址、要传送数据的长度，将缓冲区索引值初始化为 0。

第 57 将从机地址写入 IICDS 寄存器，这样，在第 58 行启动传输并发出 S 信号后，紧接着就自动发出从机地址。

第 58 行设置 IICSTAT 寄存器，将 S3C2410/S3C2440 设为主机发送器，并发出 S 信号。后续的传输工作将在中断服务程序中完成。

第 61 行等待 g_tS3C24xx_I2C.DataCount 在中断服务程序中被设为-1，这表明传输完成，于是返回。

3. S3C2410/S3C2440 I²C 主机接收函数

i2c_read 函数的实现与 i2c_write 函数类似，代码如下：

```
64 /*
65  * 主机接收
66  * slvAddr : 从机地址，buf : 数据存放的缓冲区，len : 数据长度
67  */
68 void i2c_read(unsigned int slvAddr, unsigned char *buf, int len)
69 {
70      g_tS3C24xx_I2C.Mode = RDDATA;   // 读操作
71      g_tS3C24xx_I2C.Pt   = -1;       // 索引值初始化为-1，表示第 1 个中断时不接收
数据(地址中断)
72      g_tS3C24xx_I2C.pData = buf;     // 保存缓冲区地址
73      g_tS3C24xx_I2C.DataCount = len; // 传输长度
74
75      IICDS       = slvAddr;
76      IICSTAT     = 0xb0;     // 主机接收，启动
77
```

```
78      /* 等待直至数据传输完毕 */
79      while (g_tS3C24xx_I²C.DataCount != -1);
80 }
81
```

需要注意的是第 71 行将索引值设为-1，在中断处理函数中会根据这个值决定是否从 IICDS 寄存器中读取数据。读操作时，第 1 次中断发生时表示发出了地址，这时候还不能读取数据。

4. S3C2410/S3C2440 I²C 中断服务程序

I²C 操作的主体在中断服务程序，它分 3 部分：首先是在 SRCPND、INTPND 中清除中断，后面两部分分别对应于写操作、读操作。先看清除中断的代码：

```
82 /*
83  * I²C 中断服务程序
84  * 根据剩余的数据长度选择继续传输或者结束
85  */
86 void I2CIntHandle(void)
87 {
88     unsigned int iicSt,i;
89
90     // 清中断
91     SRCPND = BIT_IIC;
92     INTPND = BIT_IIC;
93
94     iicSt  = IICSTAT;
95
96     if(iicSt & 0x8){ printf("Bus arbitration failed\n\r"); }    // 仲裁
失败
```

第 91、92 行用来清除 I²C 中断的代码。需要注意的是，即使清除中断后，IICCON 寄存器中的位 4（中断标志位）仍为 1，这导致 I²C 传输暂停。

第 94 读取状态寄存器 IICSTAT，发生中断时有可能是因为仲裁失败，在第 96 行对它进行处理。本程序中忽略仲裁失败（只有一个 I²C 主机，不可能发生仲裁失败）。

接下来是一个 switch 语句，分别处理写操作、读操作。先看写操作：

```
98     switch (g_tS3C24xx_I2C.Mode)
99     {
100        case WRDATA:
101        {
102            if((g_tS3C24xx_I2C.DataCount--) == 0)
```

```
103             {
104                 // 下面两行用来恢复 I²C 操作，发出 P 信号
105                 IICSTAT = 0xd0;
106                 IICCON  = 0xaf;
107                 Delay(10000);  // 等待一段时间以便 P 信号已经发出
108                 break;
109             }
110
111             IICDS = g_tS3C24xx_I2C.pData[g_tS3C24xx_I2C.Pt++];
112
113             // 将数据写入 IICDS 后，需要一段时间才能出现在 SDA 线上
114             for (i = 0; i < 10; i++);
115
116             IICCON = 0xaf;      // 恢复 I²C 传输
117             break;
118         }
119
```

g_tS3C24xx_I2C.DataCount 表示剩余等待传输的数据个数，第 102 行判断数据是否已经全部发送完毕：若是，则通过第 105、106 行发出 P 信号，停止传输。

第 105 行设置 IICSTAT 寄存器以便发出 P 信号，但是由于这时 IICCON[4]仍为 1，P 信号还没有实际发出。当第 106 行清除 IICCON[4]后，P 信号才真正发出。第 107 行等待一段时间，确保 P 信号已经发送完毕。

如果数据还没发送完毕，第 111 行从缓冲区中得到下一个要发送的数据，将它写入 IICDS 寄存器中。稍加等待之后，即可在第 116 行清除 IICCON[4]以恢复 I²C 传输，这时，IICDS 寄存器中的数据就会发送出去，这将触发下一个中断。

I²C 读操作的处理与写操作类似，代码如下：

```
120         case RDDATA:
121         {
122             if (g_tS3C24xx_I2C.Pt == -1)
123             {
124                 // 这次中断是在发送 I²C 设备地址后发生的，没有数据
125                 // 只接收一个数据时，不要发出 ACK 信号
126                 g_tS3C24xx_I2C.Pt = 0;
127                 if(g_tS3C24xx_I2C.DataCount == 1)
128                     IICCON = 0x2f;   // 恢复 I²C 传输，开始接收数据，接收到数据时不
发出 ACK
129                 else
130                     IICCON = 0xaf;   // 恢复 I²C 传输，开始接收数据
```

```
131             break;
132         }
133
134         if ((g_tS3C24xx_I2C.DataCount--) == 0)
135         {
136             g_tS3C24xx_I2C.pData[g_tS3C24xx_I2C.Pt++] = IICDS;
137
138             // 下面两行恢复 I²C 操作，发出 P 信号
139             IICSTAT = 0x90;
140             IICCON  = 0xaf;
141             Delay(10000);  // 等待一段时间以便 P 信号已经发出
142             break;
143         }
144
145         g_tS3C24xx_I2C.pData[g_tS3C24xx_I2C.Pt++] = IICDS;
146
147         // 接收最后一个数据时，不要发出 ACK 信号
148         if(g_tS3C24xx_I2C.DataCount == 0)
149             IICCON = 0x2f;   // 恢复 I²C 传输，接收到下一数据时无 ACK
150         else
151             IICCON = 0xaf;   // 恢复 I²C 传输，接收到下一数据时发出 ACK
152         break;
153     }
```

读操作比写操作多了一个步骤：第 1 次中断发生时表示发出了地址，这时候还不能读取数据，在代码中要分辨这点。对应第 122～132 行：如果 g_tS3C24xx_I2C.Pt 等于-1，表示这是第 1 次中断，然后修改 g_tS3C24xx_I2C.Pt 为 0，并设置 IICCON 寄存器恢复 I²C 传输（第 127～130 行）。

当数据传输已经开始后，每接收到一个数据就会触发一次中断。后面的代码读取数据，判断所有数据是否已经完成：如果完成则发出 P 信号，否则继续下一次传输。

第 134 行判断数据是否已经全部接收完毕：若是，先通过第 136 行将当前数据从 IICDS 寄存器中取出存入缓冲区，然后通过第 139、140 行发出 P 信号停止传输。

第 139 行设置 IICSTAT 寄存器以便发出 P 信号，但是由于这时 IICCON[4]仍为 1，P 信号还没有实际发出。第 140 行清除 IICCON[4]后，P 信号才真正发出。第 141 行等待一段时间，确保 P 信号已经发送完毕。

第 145～151 行用来启动下一个数据的接收。

第 145 行将当前数据从 IICDS 寄存器中取出存入缓冲区中。

第 148～151 行判断是否只剩下最后一个数据了：若是，就通过第 149 行中清除 IICCON[4]、IICCON[7]，这样即可恢复 I²C 传输，并使得接收到数据后，S3C2410/S3C2440 不发出 ACK 信号（这样从机即可知道数据传输完毕）；否则，在第 151 行中只要清除 IICCON[4]

以恢复 I²C 传输。

中断服务程序中，当数据传输完毕时，g_tS3C24xx_I2C.DataCount 将自减为-1，这样，i2c_read 或 i2c_write 函数即可跳出等待，直接返回。

5. RTC 芯片 M41t11 特性相关的操作

m41t11.c 文件中提供两个函数 m41t11_set_datetime、m41t11_get_datetime，前者用来设置日期与时间，后者用来读取日期与时间。它们都通过调用 i2c_read 或 i2c_write 函数来完成与 M41t11 的交互。

前面说过，操作 M41t11 只需要两步骤：发出寄存器地址，发出数据或读取数据。m41t11_set_datetime 函数把这两个步骤合为一个 I²C 写操作，m41t11_get_datetime 函数先发起一个 I²C 写传输，再发起一个 I²C 读传输。

先看 m41t11_set_datetime 函数的代码：

```
29 /*
30  * 写 m41t11，设置日期与时间
31  */
32 int m41t11_set_datetime(struct rtc_time *dt)
33 {
34     unsigned char leap_yr;
35     struct {
36         unsigned char addr;
37         struct rtc_registers rtc;
38     } __attribute__ ((packed)) addr_and_regs;
       …… /* 设置 rtc 结构，即根据传入的参数构造各寄存器的值 */
76     i2c_write(0xD0, (unsigned char *)&addr_and_regs, sizeof(addr_and_regs));
77
78     return 0;
79 }
80
```

省略号表示的代码用来设置 addr_and_regs 结构。这个结构分为两部分：addr_and_regs.addr 表示 M41t11 寄存器地址（它被设为 0），addr_and_regs. rtc 表示 M41t11 的 8 个寄存器——秒、分、时、天（星期几）、日、月、年、控制寄存器。

根据传入参数填充好 addr_and_regs 结构之后，就可以启动 I²C 写操作了。第 38 行使用“__attribute__ ((packed))”设置这个结构为紧凑格式，使得它的大小为 9 个字节（否则大小为 12 字节）：1 个字节用来保存寄存器地址，8 个字节用来保存 8 个寄存器的值。

第 76 行发起一次 I²C 写操作，将 addr_and_regs 结构结构中的数据（寄存器地址、日期、时间）发送给 M41t11：M41t11 会把接收到的第 1 个数据当作寄存器的起始地址，随后是要写入寄存器的数据。

m41t11_get_datetime 函数的代码与 m41t11_set_datetime 函数类似，如下所示：

```
81 /*
82  * 读取m41t11，获得日期与时间
83  */
84 int m41t11_get_datetime(struct rtc_time *dt)
85 {
86     unsigned char addr[1] = { 0 };
87     struct rtc_registers rtc;
88
89     memset(&rtc, 0, sizeof(rtc));
90
91     i2c_write(0xD0, addr, 1);
92     i2c_read(0xD0, (unsigned char *)&rtc, sizeof(rtc));
93
       …… /* 根据读出的各寄存器的值，设置dr结构 */
110     return 0;
111 }
112
```

第 91 行发起一次 I²C 写传输，设置要操作的 M41t11 寄存器地址为 0。

第 92 行发起一次 I²C 读传输，读出 M41t11 各寄存器的值。

省略号对应的代码根据读出的各寄存器的值，设置 dr 结构。M41t11 中以 BCD 码表示日期与时间，需要转换为程序使用的一般二进制格式。

12.2.4 I²C 实例的连接脚本

本实例要用到第 8 章 NAND Flash 控制器的函数将代码从 NAND Flash 复制到 SDRAM 中。由于 nand 代码中用到了全局变量，而全局变量要运行于可读写的内存中，为了方便，使用连接脚本将这些初始化代码放在 Steppingstone 中。

连接脚本为 i2c.lds，内容如下：

```
01 SECTIONS {
02     . = 0x00000000;
03     .init : AT(0){ head.o init.o nand.o}
04     . = 0x30000000;
05     .text : AT(4096) { *(.text) }
06       .rodata ALIGN(4) : AT((LOADADDR(.text)+SIZEOF(.text)+3)&~(0x03))
{*(.rodata*)}
07     .data ALIGN(4)   : AT((LOADADDR(.rodata)+SIZEOF(.rodata)+3)&~(0x03))
{ *(.data) }
08     _ _bss_start = .;
09     .bss ALIGN(4)  : { *(.bss)  *(COMMON) }
```

```
10      _ _bss_end = .;
11 }
```

第2～3行将head.S、init.c和nand.c对应的代码的运行地址设为0，加载地址（存在NAND Flash上的地址）设为0。从NAND Flash启动时，这些代码被复制到Steppingstone后就可以直接运行。

第4行设置其余代码的运行地址为0x30000000；第5行将代码段的加载地址设为4096，表示代码段将存在NAND Flash地址4096处。

第6～7行的“AT(…)”设置rodata段、data段的加载地址依次位于代码段之后。“LOADADDR(…)”表示某段的加载地址，SIZEOF(…)表示它的大小。这两行的前面使用“ALIGN(4)”使得它们的运行地址为4字节对应，为了使各段之间加载地址的相对偏移值等于运行地址的相对偏移值，需要将“AT(…)”中的值也设为4字节对齐：先加上3，然后与~(0x03)进行与操作（将低两位设为0）。

12.2.5 实例测试

本程序在main函数中通过串口输出一个菜单，用于设置或读取时间，步骤如下。

（1）使用串口将开发板的COM0和PC的串口相连，打开PC的串口工具并设置其波特率为115200、8N1。

（2）在i2c目录下执行make命令生成i2c可执行程序，烧入NAND Flash后运行。

（3）在PC的串口工具上，可以看到如下菜单：

```
##### RTC Menu #####
Data format: 'year.month.day w hour:min:sec', 'w' is week day
eg: 2007.08.30 4 01:16:57
[S] Set the RTC
[R] Read the RTC
Enter your selection:
```

（4）要设置RTC，输入“s”或“S”，可以看到如下字符。

```
Enter date&time:
```

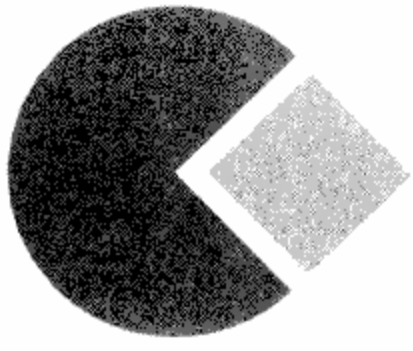
在串口中按照“year.month.day w hour:min:sec”的格式输入日期与时间，比如：“2007.08.30 4 01:16:57”，然后按回车键。

注意 只能输入2000.01.1至2099.12.31之内的日期与时间；年月日与星期几必须真实存在，否则RTC芯片无法正常工作。

（5）要可读取RTC，输入“r”或“R”，即可看到当前日期与时间，串口上会输出类似下面的结果。

```
*** Now is: 2007.08.30 4 01:16:57 ***
```

（6）断电后重启，输入“R”，仍可以看到正确的时间，只要RTC芯片M41t11没被断电，它就会一直工作。实际上，常使用电池给RTC芯片供电，这使得主电源关闭后系统仍可正确计时。



第 13 章　LCD 控制器

本章目标

- 了解 LCD 显示器的接口及时序
- 掌握 S3C2410/S3C2440 LCD 控制器的使用方法
- 了解帧缓冲区的概念，掌握如何设置帧缓冲区来显示图像

13.1　LCD 和 LCD 控制器

13.1.1　LCD 显示器

1. LCD 的种类

LCD (Liquid Crystal Display)，即液晶显示器，是一种采用了液晶控制透光度技术来实现色彩的显示器。它与传统的 CRT 显示器相比有很多优点：轻薄、能耗低、幅射小等，市场占有率越来越大。LCD 有多种类型，比如 STN、TFT、LTPS TFT、OLED 等，各有优缺点。

STN (Super Twisted Nematic，超扭曲向列)，有 CSTN 和 DSTN 之分，是 4 种 LCD 屏中最低端的一种，仅有的优点就是功耗低，在色彩鲜艳度和画面亮度上相对于 TFT 和其他 LCD 屏存在明显不足，在日光下几乎不能显示，而且响应时间长达 200ms 左右，播放动画或视频拖影非常明显。

TFT (Thin Film Transistor，薄膜晶体管)可以大大缩短屏幕响应时间，其响应时间已经小于 80ms，并改善了 STN 连续显示时屏幕模糊闪烁，有效提高了动态画面的播放力，呈现画面色彩饱和度、真实效果和对比度都非常不错，完全超越 STN，只是功耗稍高，是目前最为主流的液晶显示类型，不仅在 MP3、MP4 产品上大量应用，在桌面液晶显示器、笔记本电脑、手机等产品上的应用也非常普遍。

LTPS (Low Temperature Polycrystalline Silicon，低温多晶硅)由 TFT 衍生的新一代的技术产品，可以获得更高的分辨率和更丰富的色彩。LTPS LCD 可以提供 170°的水平和垂直可视角度，显示响应时间仅 12ms，显示亮度达到 500Cd/m2，对比度可达 500:1，这就是

一些桌面液晶屏性能越来越出色的原因。虽然 LTPS LCD 已经出现多年了，但由于 LTPS TFT 液晶屏幕的制造需要高于制造传统 TFT 屏的技术水平，目前仅少数知名大厂能制造。

OLED (Organic Light Emitting Diode，有机发光二极管)各种物理特性都具备领先优势，色彩明亮、可视角度超大、非常省电，是未来发展的主流，只是目前受技术与成本限制，未能广泛普及。目前，彩色 OLED 比较广泛地存在中低端产品中，尺寸为 1 英寸左右。随着开发的深入，更大尺寸、更高色数的 OLED 屏幕已经开始应用。

2. LCD 的接口

CPU 或显卡发出的图像数据是 TTL 信号（0～5V、0～3.3V、0～2.5V 或 0～1.8V），LCD 本身接收的也是 TTL 信号。由于 TTL 信号在高速率的长距离传输时性能不佳，抗干扰能力也比较差，后来又提出了多种接口，比如 LVDS、TDMS、GVIF、P&D、DVI 和 DFP 等。它们实际上只是将 CPU 或显卡发出的 TTL 信号编码成各种信号以便传输，在 LCD 那边将接收到的信号进行解码得到 TTL 信号。

由于数字接口标准尚未统一，所以使用 LCD 时需要根据其手册了解具体接口定义。也是基于数字接口标准尚未统一的原因，市场上大多 LCD 都采用模拟信号接口，LCD 需要先通过 ADC 将模拟信号转换为数字信号才能显示。

但是不管采用何种数字接口，本质的 TTL 信号是一样的。

（1）对于 STN LCD。

STN LCD 的数据传输方式有 3 种：4 位单扫（4-bit single scan）、4 位双扫（4-bit dual scan）、8 位单扫（8-bit single scan）。所谓“单扫”是指对于一整屏的数据，从上到下、从左到右，一个一个地发送出来；“双扫”是指将一整屏的数据分为上下两部分，同时地从上到下、从左到右，一个一个地发送出来。“4 位”、“8 位”是指发送数据时使用多少个数据线；需要注意的是，4 位双扫方式也是用到 8 根数据线，其中 4 根用于上半屏数据，另外 4 根用于下半屏数据。

除数据信号外，还有其他控制信号，所有 TTL 信号如表 13.1 所示。

表 13.1　　STN LCD 的 TTL 信号

信号名称	描　述
VFRAME	帧同步信号
VLINE	行同步信号
VCLK	像素时钟信号
VD[7:0]	数据信号
VM	AC 偏置信号
PWREN	电源开关信号

（2）对于 TFT LCD。

TFT LCD 的 TTL 信号与 STN 类似（如表 13.2 所示），只是其数据信号多达 24 根，对应像素值中的每一位。

表 13.2 TFT LCD 的 TTL 信号

信号名称	描述
VSYNC	垂直同步信号
HSYNC	水平同步信号
HCLK	像素时钟信号
VD[23:0]	数据信号
LEND	行结束信号（这不是必需的）
PWREN	电源开关信号

13.1.2 S3C2410/S3C2440 LCD 控制器介绍

1. S3C2410/S3C2440 LCD 控制器的特性与结构

S3C2410/S3C2440 LCD 控制器被用来向 LCD 传输图像数据，并提供必要的控制信号，比如 VFRAME、VLINE、VCLK、VM 等。可以支持 STN LCD 和 TFT LCD，其特性如下（BPP 表示 bit per pixel，即每个色素使用多少位来表示其颜色）。

（1）STN LCD。

- 支持 3 种扫描方式：4 位单扫、4 位双扫和 8 位单扫。
- 支持单色（1BPP）、4 级灰度（2BPP）和 16 级灰度（4BPP）屏。
- 支持 256 色（8BPP）和 4096 色（12BPP）彩色 STN 屏（CSTN）。
- 支持分辨率为 640 × 480、320 × 240、160 × 160 以及其他规格的多种 LCD。
- 虚拟屏幕最大可达 4MB。
- 对于 256 色，分辨率有 4096 × 1024、2048 × 2048、1024 × 4096 等多种。

（2）TFT LCD。

- 支持单色（1BPP）、4 级灰度（2BPP）、16 级灰度（4BPP）、256 色（8BPP）的调色板显示模式。
- 支持 64K (16BPP)和 16M (24BPP)色非调色板显示模式。
- 支持分辨率为 640 × 480、320 × 240 及其他多种规格的 LCD。
- 虚拟屏幕最大可达 4MB。
- 对于 64K 色，分辨率有 2048 × 1024 等多种。

S3C2410/S3C2440 LCD 控制器提供了驱动 STN LCD、TFT LCD 所需的所有信号（如表 13.1、表 13.2 所示），另外，还特别提供颇外的信号以支持 SEC 公司（Samsung Electronics Company）生产的 TFT LCD (称为 SEC TFT LCDs)。这 3 类信号中很大部分是复用的。

S3C2410/S3C2440 LCD 控制器的内部结构如图 13.1 所示。

REGBANK 是 LCD 控制器的寄存器组，含 17 个寄存器及一块 256 × 16 的调色板内存，用来设置各项参数。而 LCDCDMA 则是 LCD 控制器专用的 DMA 信道，可以自动地从系统总线（System Bus）上取到图像数据，这使得显示图像时不需要 CPU 的干涉。VIDPRCS 将 LCDCDMA 中的数据组合成特定的格式（比如 4 位单扫、4 位双扫和 8 位单扫等），然后从 VD[23:0]发送给 LCD 屏。同时 TIMEGEN 和 LPC3600 负责产生 LCD 屏所需要的控制时序，

例如 VSYNC、HSYNC、VCLK、VDEN，然后从 VIDEO MUX 送给 LCD 屏。其中 LPC3600 专用于 SEC TFT LCD。

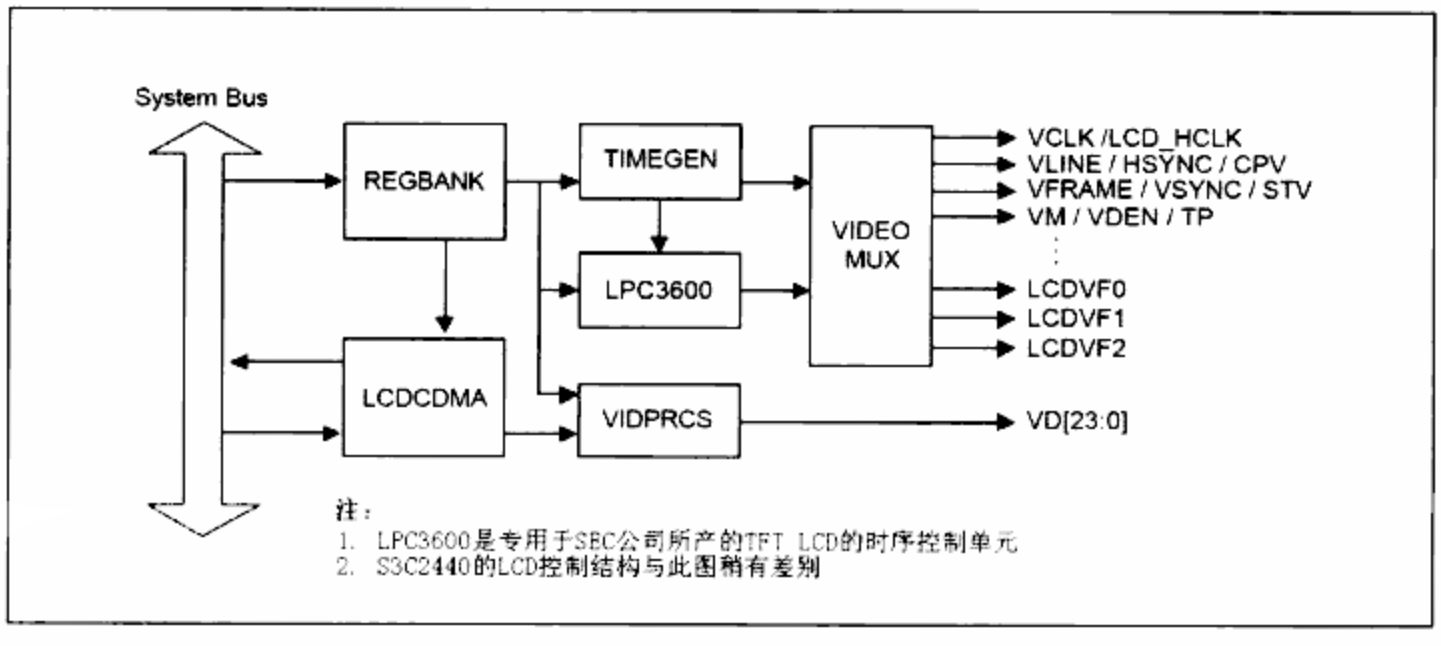


图 13.1 S3C2410/S3C2440 LCD 控制器结构图

LCDCDMA 中有两个 FIFO：FIFOH 容量为 16（1 个字为 4 个字节）个字 FIFOL 容量为 12 个字。当使用“双扫”方式时，FIFOH、FIFOL 分别用于传输上半屏、下半屏数据；当使用“单扫”方式时，只用到 FIFOH。当 FIFO 为空或者其中的数据已经减少到设定的阈值时，LCDCDMA 自动地发起 DMA 传输从内存中获得图像数据。

2. 显示器上数据的组织格式

对于屏幕上的一整幅图像，它的数据是如何组织的？无论是 CRT 显示器还是 LCD 显示器，它们都有相同的概念。

一幅图像被称为一帧（frame），每帧由多行组成，每行由多个像素组成，每个像素的颜色使用若干位的数据来表示。对于单色显示器，每个像素使用 1 位来表示，称为 1BPP；对于 256 色显示器，每个像素使用 8 位来表示，称为 8BPP。

显示器从屏幕的左上方开始，一行一行地取得每个像素的数据并显示出来，当显示到一行的最右边时，跳到下一行的最左边开始显示下一行；当显示完所有行后，跳到左上方开始下一帧。显示器沿着“Z”字行的路线进行扫描，使用 HSYNC、VSYNC 信号来控制扫描路线的跳转，HSYNC 表示“是跳到最左边的时候了”，VSYNC 表示“是跳到最上边的时候了”。

在工作中的显示器上，可以在四周看见黑色的边框。上方的黑框是因为当发出 VSYNC 信号时，需要经过若干行之后第一行数据才有效；下方的黑框是因为显示完所有行的数据时，显示器还没扫描到最下边（VSYNC 信号还没有发出），这时数据已经无效；左边的黑框是因为当发出 HSYNC 信号时，需要经过若干像素之后第一列数据才有效；右边的黑框是因为显示完一行的数据时，显示器还没扫描到最右边（HSYNC 信号还没有发出），这时数据已经无效。显示器只会依据 VSYNC、HSYNC 信号来取得、显示数据，并不理会该数据是否有效，何时发出有效数据由显卡决定。

VSYNC 信号出现的频率表示一秒钟内能显示多少帧图像，称为垂直频率或场频率，这就是我们常说的“显示器的频率”；HSYNC 信号出现的频率称为水平频率。

显示器上，一帧数据的存放位置与 VSYNC、HSYNC 信号的关系如图 13.2 所示。

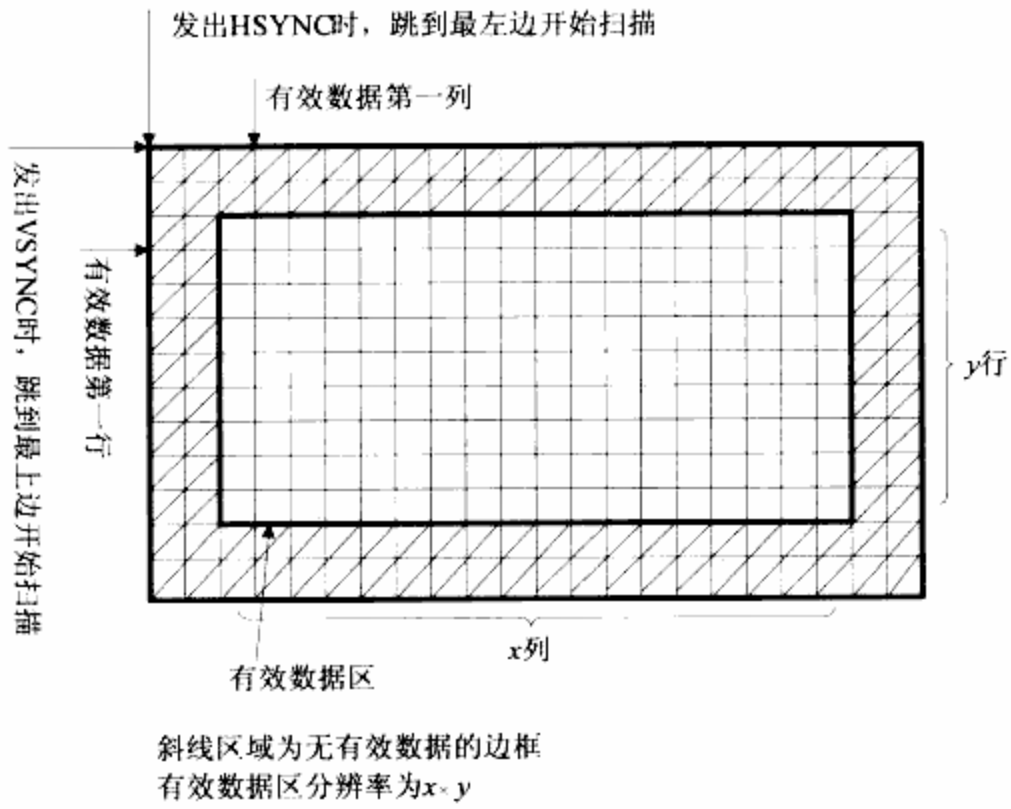


图 13.2 显示器扫描一帧数据的过程

“有效数据”的行数、列数，即分辨率，它与 VSYNC、HSYNC 信号之间的“距离”等，都是可以设置的，这由显卡来完成，这些设置在下一节对 TFT LCD 的操作中都可以看到。

3. TFT LCD 的操作

目前市场上主流的 LCD 为 TFT LCD，本章基于 TFT LCD 来介绍 LCD 控制器的使用。对于 STN LCD，它所涉及的操作是类似的。

先了解 TFT LCD 的时序，这使得我们在设置各个寄存器时有个形象的概念。每个 VSYNC 信号表示一帧数据的开始；每个 HSYNC 信号表示一行数据的开始，无论这些数据是否有效；每个 VCLK 信号表示正在传输一个像素的数据，无论它是否有效。数据是否有效只是对 CPU 的 LCD 控制器来说的，LCD 根据 VSYNC、HSYNC、VCLK 不停地读取总线数据、显示。

下面讲解时序图，请参考图 13.3。

（1）VSYNC 信号有效时，表示一帧数据的开始。

（2）VSPW 表示 VSYNC 信号的脉冲宽度为（VSPW+1）个 HSYNC 信号周期，即（VSPW+1）行，这（VSPW+1）行的数据无效。

（3）VSYNC 信号脉冲之后，还要经过（VBPD+1）个 HSYNC 信号周期，有效的行数据才出现。所以，在 VSYNC 信号有效之后，总共还要经过（VSPW+1+VBPD+1）个无效的行，它对应图 13.2 上方的边框，第一个有效的行才出现。

（4）随后即连续发出（LINEVAL+1）行的有效数据。

（5）最后是（VFPD+1）个无效的行，它对应图 13.2 下方的边框，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）。

现在深入到一行中像素数据的传输过程，它与上面行数据的传输过程相似。

（1）HSYNC 信号有效时，表示一行数据的开始。

（2）HSPW 表示 HSYNC 信号的脉冲宽度为（HSPW+1）个 VCLK 信号周期，即（HSPW+1）个像素，这（HSPW+1）个像素的数据无效。

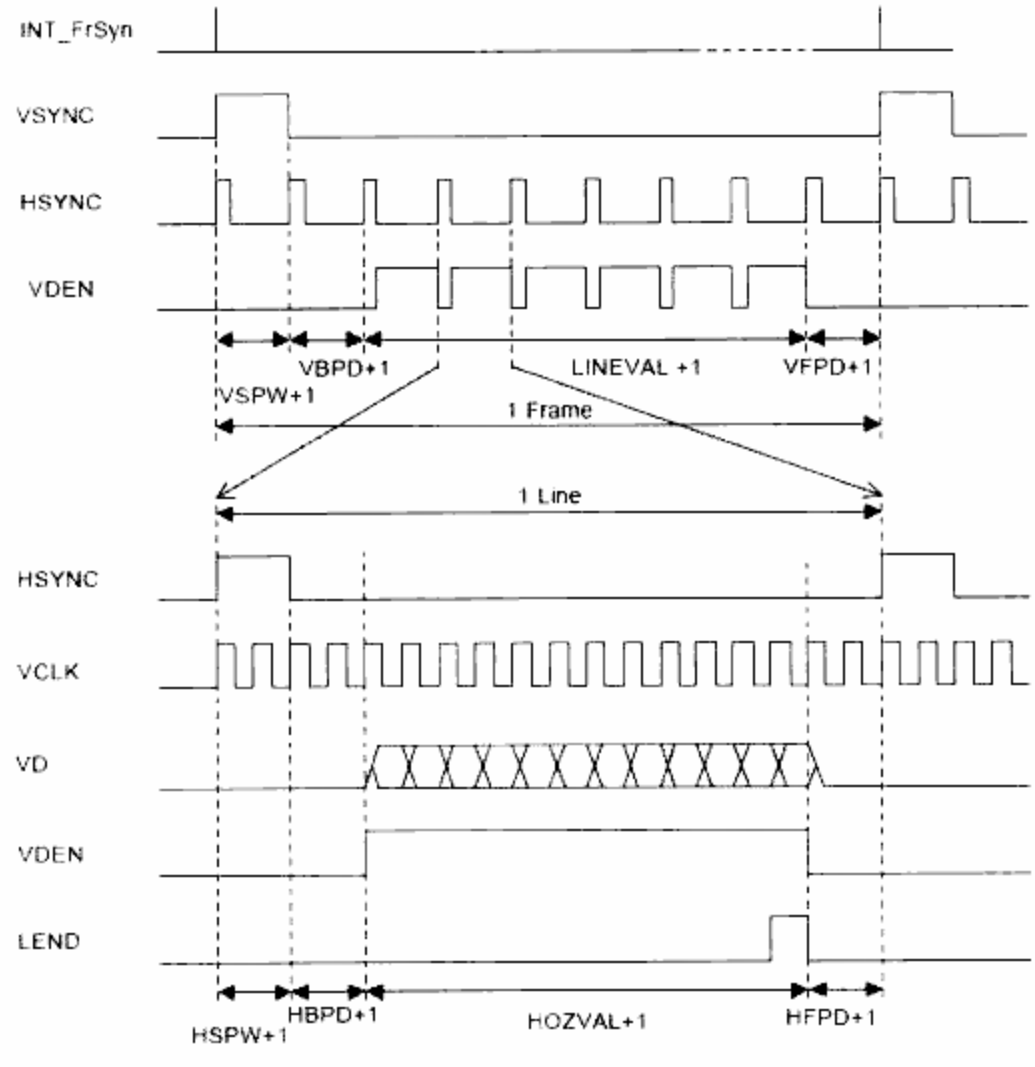


图 13.3 TFT LCD 时序图

（3）HSYNC 信号脉冲之后，还要经过（HBPD+1）个 VCLK 信号周期，有效的像素数据才出现。所以，在 HSYNC 信号有效之后，总共还要经过（HSPW+1+HBPD+1）个无效的像素，它对应图 13.2 左边的边框，第一个有效的像素才出现。

（4）随后即连续发出（HOZVAL+1）个像素的有效数据。

（5）最后是（HFPD+1）个无效的像素，它对应图 13.2 右边的边框，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）。

时序图中各信号的时间参数都可以在 LCD 控制寄存器中设置，VCLK 作为时序图的基准信号，它的频率可以如下计算：

```
VCLK(Hz) = HCLK/[(CLKVAL+1×2]
```

VSYNC 信号的频率又称为帧频率、垂直频率、场频率、显示器的频率，它可以如下计算：

```
Frame Rate = 1/ [ { (VSPW+1) + (VBPD+1) + (LIINEVAL + 1) + (VFPD+1) } ×
        {(HSPW+1) + (HBPD +1)+ (HFPD+1) + (HOZVAL + 1) } ×{ 2 × ( CLKVAL+1 ) /
( HCLK ) } ]
```

将 VSYNC、HSYNC、VCLK 等信号的时间参数设置好之后，并将帧内存（frame memory）的地址告诉 LCD 控制器，它即可自动地发起 DMA 传输从帧内存中得到图像数据，最终在上述信号的控制下出现在数据总线 VD[23:0]上。用户只需要把要显示的图像数据写入帧内存

中，在这之前，先了解一下各种格式的图像数据在内存中如何存储。

显示器上每个像素的颜色由 3 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）的调色板显示模式，支持 64K (16BPP)和 16M(24BPP)非调色板显示模式。下面只介绍 256 色（8pp）、64K (16BPP)和 16M (24BPP)色显示模式下，图像数据的存储格式。

（1）16M (24BPP)色。

16M (24BPP)色的显示模式就是使用 24 位的数据来表示一个像素的颜色，每种原色使用 8 位。LCD 控制器从内存中获得某个像素的 24 位颜色值后，直接通过 VD[23:0]数据线发送给 LCD。为了方便 DMA 传输，在内存中使用 4 个字节（32 位）来表示一个像素，其中的 3 个字节从高到低分别表示红、绿、蓝，剩余的 1 个字节数据无效。是最低字节还是最高字节无效，这是可以选择的，如图 13.4 所示。

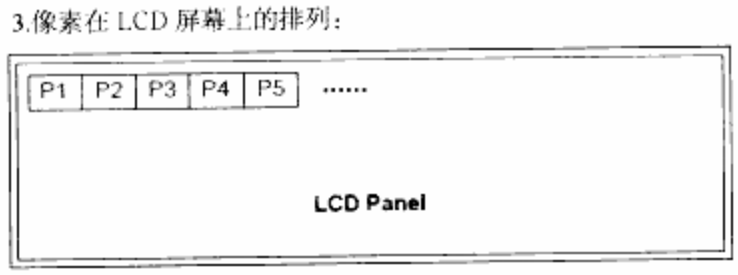
1.BSWP=0、HWSWP=0、BPP24BL=0 时，
内存中像素的排列格式如下：

	D[31:24]	D[23:0]
000H	忽略	P1
004H	忽略	P2
008H	忽略	P3

2.BSWP=0、HWSWP=0、BPP24BL=1 时，
内存中像素的排列格式如下。

	D[31:8]	D[7:0]
000H	P1	忽略
004H	P2	忽略
008H	P3	忽略

3.像素在 LCD 屏幕上的排列：



4.像素色值与 VD[23：0]引脚的对应关系：

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

图 13.4　24BPP 模式下像素颜色的数据格式

（2）64K（16BPP）色。

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、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从高到低分成3个5位来表示红色、绿色和蓝色，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）。

一个4字节可以表示两个16BPP的像素，使用高2字节还是低2字节来表示第一个像素，这也是可以选择的。

显示模式为16BPP时，内存数据与像素位置的关系如图13.5所示。在5:5:5:1格式下，VD[18]、VD[10]、VD[2]数据线上的值是一样的，都表示透明度，图13.5中的“NC”表示没有连接（not connect）。

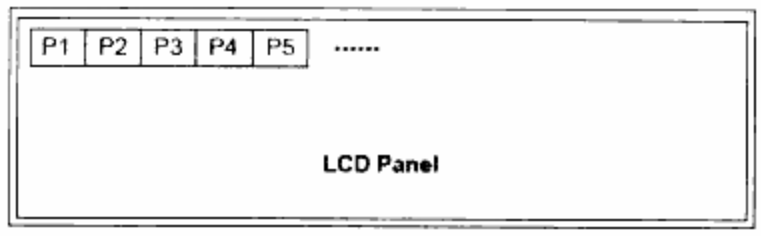
1.BSWP=0、HWSWP=0时，
内存中像素的排列格式如下：

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

2.BSWP=0、HWSWP=1时，
内存中像素的排列格式如下：

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

3.像素在LCD屏幕上的排列：



4.像素色值与VD[23：0]引脚的对应关系：

(5:6:5)

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

(5:5:5:1)

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

图13.5 16BPP模式下像素颜色的数据格式

（3）256色（8BPP）。

256色（8BPP）的显示模式就是使用8位的数据来表示一个像素的颜色，但是对三种原色平均下来，每个原色只能使用不到3位的数据来表示，即每个原色最多不过8个级别，这不足以表示更丰富的颜色。

为了解决 8BPP 模式显示能力太弱的问题，需要使用调色板（Palette）。每个像素对应的 8 位数据不再用来表示 RGB 三种原色，而是表示它在调色板中的索引值；要显示这个像素时，使用这个索引值从调色板中取得其 RGB 颜色值。所谓调色板就是一块内存，可以对每个索引值设置其颜色，可以使用 24BPP 或 16BPP。S3C2410/S3C2440 中，调色板是一块 256 × 16 的内存，使用 16BPP 的格式来表示 256 色（8BPP）显示模式下各个索引值的颜色。这样，即使使用 256 色（8BPP）的显示模式，最终出现在 LCD 数据总线上的仍是 16BPP 的数据。

一个 4 字节可以表示 4 个 8BPP 的像素，字节与像素的对应顺序是可以选择的，如图 13.6 所示。

1.BSWP=0、HWSWP=0 时，
内存中像素的排列格式如下：

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

2.BSWP=1、HWSWP=0 时，
内存中像素的排列格式如下：

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

3.像素在 LCD 屏幕上的排列：

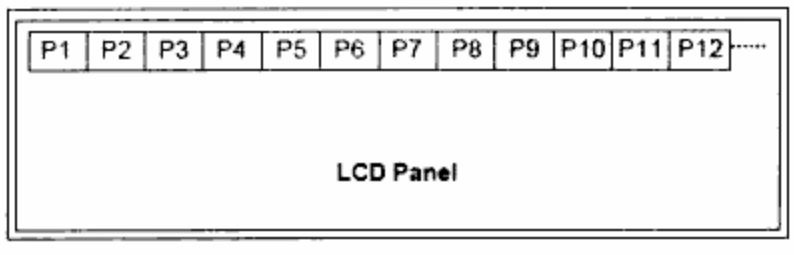


图 13.6 8BPP 模式下内存数据与像素位置对应关系

调色板中数据存放的格式与上面所描述的 16BPP 显示模式相似，也分两种格式：5:6:5、5:5:5:1。调色板中数据的格式及与 LCD 数据线 VD[23:0]的对应关系，如表 13.3 和表 13.4 所示。

表 13.3　　5:6:5 格式下调色板的数据格式

序　号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	地　址
00H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000400
01H	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D000404
……																	……
FFH	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	0X4D0007FC
VD 引脚号	23	22	21	20	19	15	14	13	12	11	10	7	6	5	4	3	

表 13.4 5:5:5:1 格式下调色板的数据格式

序号	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	地址
00H	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	I	0X4D000400
01H	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	I	0X4D000404
……																	……
FFH	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	I	0X4D0007FC
VD 引脚号	23	22	21	20	19	15	14	13	12	11	7	6	5	4	3	2	

注：① 0x4D000400 是调色板的起始地址。

② 5:5:5:1 格式下，VD18、VD10 和 VD2 三个数据线中都是亮度值 I，即最低位的值。

③ 当 LCDCON5 寄存器中的 VSTATUS、HSTATUS 有效时，不能读写调色板。

各种模式下用来传输红、绿、蓝三种原色的颜色值的 VD 数据线如表 13.5 所示。

表 13.5 各模式下用来传输 RGB 三原色的数据线

	24BPP	16BPP/8BPP 5:6:5 格式	16BPP/8BPP 5:5:5:1 格式
红色	VD[23:16]	VD[23:19]	VD[23:19]+VD[18]
绿色	VD[15:8]	VD[15:10]	VD[15:11]+VD[10]
蓝色	VD[7:0]	VD[7:3]	VD[7:3]+VD[2]

没有用到的数据线其电平为 0，从这个观点来看，无论是 24BPP 模式还是 16BPP、8BPP 模式，24 根数据线 VD[23:0]都被用到了。事实上，一个 TFT LCD 能处理的像素位宽是固定的，即它数据线的数目是固定的，红、绿、蓝 3 类信号线总是连接到各字节中的高位；软件设置 24BPP、16BPP、8BPP 以及调色板等，只会影响到色值的精度。

4. 使用 TFT LCD 时 LCD 控制器的寄存器设置

LCD 控制器中 REGBANK 的 17 个寄存器可以分为 6 种，如表 13.6 所示。

表 13.6 REGBANK 寄存器组分类

名称	说明
LCDCON1～LCDCON5	用于选择 LCD 类型、设置各类控制信号的时间特性等
LCDSADDR1～LCDSADDR3	用于设置帧内存的地址
TPAL	临时调色板寄存器，可以快速地输出一帧单色的图像
LCDINTPND LCDSRCPND LCDINTMSK	用于 LCD 的中断，在一般应用中无需中断
REDLUT GREENLUT BLUELUT DITHMODE	专用于 STN LCD
TCONSEL	专用于 SEC TFT LCD

对于 TFT LCD，一般情况下只需要设置前两种寄存器；在 8BPP 模式下，如果想快速地输出一帧单色的图像，可以借助于 TPAL 寄存器。下面分别介绍。

（1）LCD 控制寄存器 LCDCON1。

用于选择 LCD 类型、设置像素时钟、使能 LCD 信号的输出等，格式如表 13.7 所示。

表 13.7　　LCDCON1 寄存器格式

功　能	位	说　明
LINECNT	[27:18]	只读，每输出一个有效行其值减一，从 LINEVAL 减到 0
CLKVAL	[17:8]	用于设置 VCLK（像素时钟） 对于 TFT LCD：VCLK = HCLK / [(CLKVAL+1)×2] (CLKVAL >= 0)
MMODE	[7]	设置 VM 信号的反转效率，用于 STN LCD
PNRMODE	[6:5]	设置 LCD 的类型，对于 TFT LCD 设为 0b11
BPPMODE	[4:1]	设置 BPP，对于 TFT LCD： 0b1000 = 1 bpp 0b1001 = 2 bpp 0b1010 = 4 bpp 0b1011 = 8 bpp 0b1100 = 16 bpp 0b1101 = 24 bpp
ENVID	[0]	LCD 信号输出使能位，0：禁止，1：使能

（2）LCD 控制寄存器 LCDCON2。

用于设置垂直方向各信号的时间参数，格式如表 13.8 所示，请参考图 13.3 TFT LCD 时序图。

表 13.8　　LCDCON2 寄存器格式

功　能	位	说　明
VBPD	[31:24]	VSYNC 信号脉冲之后，还要经过（VBPD+1）个 HSYNC 信号周期，有效的行数据才出现
LINEVAL	[23:14]	LCD 的垂直宽度：（LINEVAL+1）行
VFPD	[13:6]	一帧中的有效数据完结后，到下一个 VSYNC 信号有效前的无效行数目：VFPD+1
VSPW	[5:0]	表示 VSYNC 信号的脉冲宽度为（VSPW+1）个 HSYNC 信号周期，即（VSPW+1）行，这（VSPW+1）行的数据无效

（3）LCD 控制寄存器 LCDCON3。

用于设置水平方向各信号的时间参数，格式如下表所示，请参考图 13.3TFT LCD 时序图。

表 13.9　　LCDCON3 寄存器格式

功　能	位	说　明
HBPD	[25:19]	HSYNC 信号脉冲之后，还要经过（HBPD+1）个 VCLK 信号周期，有效的像素数据才出现
HOZVAL	[18:8]	LCD 的水平宽度：（HOZVAL+1）列（像素）
HFPD	[7:0]	一行中的有效数据完结后，到下一个 HSYNC 信号有效前的无效像素个数：HFPD+1

（4）LCD 控制寄存器 LCDCON4。

对于 TFT LCD，这个寄存器只用来设置 HSYNC 信号的脉冲宽度，位[7:0]的数值称为 HSPW，表示脉冲宽度为（HSPW+1）个 VCLK 周期。

（5）LCD 控制寄存器 LCDCON5。

用于设置各个控制信号的极性，并可从中读到一些状态信息，格式列表如表 13.10 所示。

表 13.10　　LCDCON5 寄存器格式

功　能	位	说　明
VSTATUS	[16:15]	只读，垂直状态，请参考图 13.3 TFT LCD 时序图。 00：正处于 VSYNC 信号脉冲期间 01：正处于 VSYNC 信号结束到行有效之间 10：正处于行有效期间 11：正处于行有效结束到下一个 VSYNC 信号之间
HSTATUS	[14:13]	只读，水平状态，请参考图 13.3 TFT LCD 时序图。 00：正处于 HSYNC 信号脉冲期间 01：正处于 HSYNC 信号结束到像素有效之间 10：正处于像素有效期间 11：正处于像素有效结束到下一个 HSYNC 信号之间
BPP24BL	[12]	设置 TFT LCD 的显示模式为 24BPP 时，一个 4 字节中哪 3 个字节有效（图 13.5）。 0：LSB 有效（低地址的 3 字节）；1：MSB 有效（高地址的 3 字节）
FRM565	[11]	设置 TFT LCD 的显示模式为 16BPP 时，使用的数据格式。 0 表示 5:5:5:1 格式；1 表示 5:6:5 格式
INVVCLK	[10]	设置 VCLK 信号有效沿的极性。 0：在 VCLK 的下降沿读取数据；1：在 VCLK 的上升沿读取数据
INVVLINE	[9]	设置 VLINE/HSYNC 脉冲的极性。 0：正常的极性；1：反转的极性
INVVFRAME	[8]	设置 VFRAME/VSYNC 脉冲的极性。 0：正常的极性；1：反转的极性
INVVD	[7]	设置 VD 数据线表示数据（0/1）的极性。 0：正常的极性；1：反转的极性
INVVDEN	[6]	设置 VDEN 信号的极性。 0：正常的极性；1：反转的极性
INVPWREN	[5]	设置 PWREN 信号的极性。 0：正常的极性；1：反转的极性
INVLEND	[4]	设置 LEND 信号的极性。 0：正常的极性；1：反转的极性

续表

功　能	位	说　明
PWREN	[3]	LCD_PWREN 信号输出使能。 0：禁止；1：使能
ENLEND	[2]	LEND 信号输出使能。 0：禁止；1：使能
BSWP	[1]	字节交换使能，参考图 13.6。 0：禁止；1：使能
HWSWP	[0]	半字（2 字节）交换使能，参考图 13.5。 0：禁止；1：使能

（6）帧内存地址寄存器 LCDSADDR1～LCDSADDR3。

帧内存可以很大，而真正要显示的区域被称为视口（view point），它处于帧内存之内。这 3 个寄存器用于确定帧内存的起始地址，定位视口在帧内存中的位置。

图 13.7 给出了帧内存和视口的位置关系：

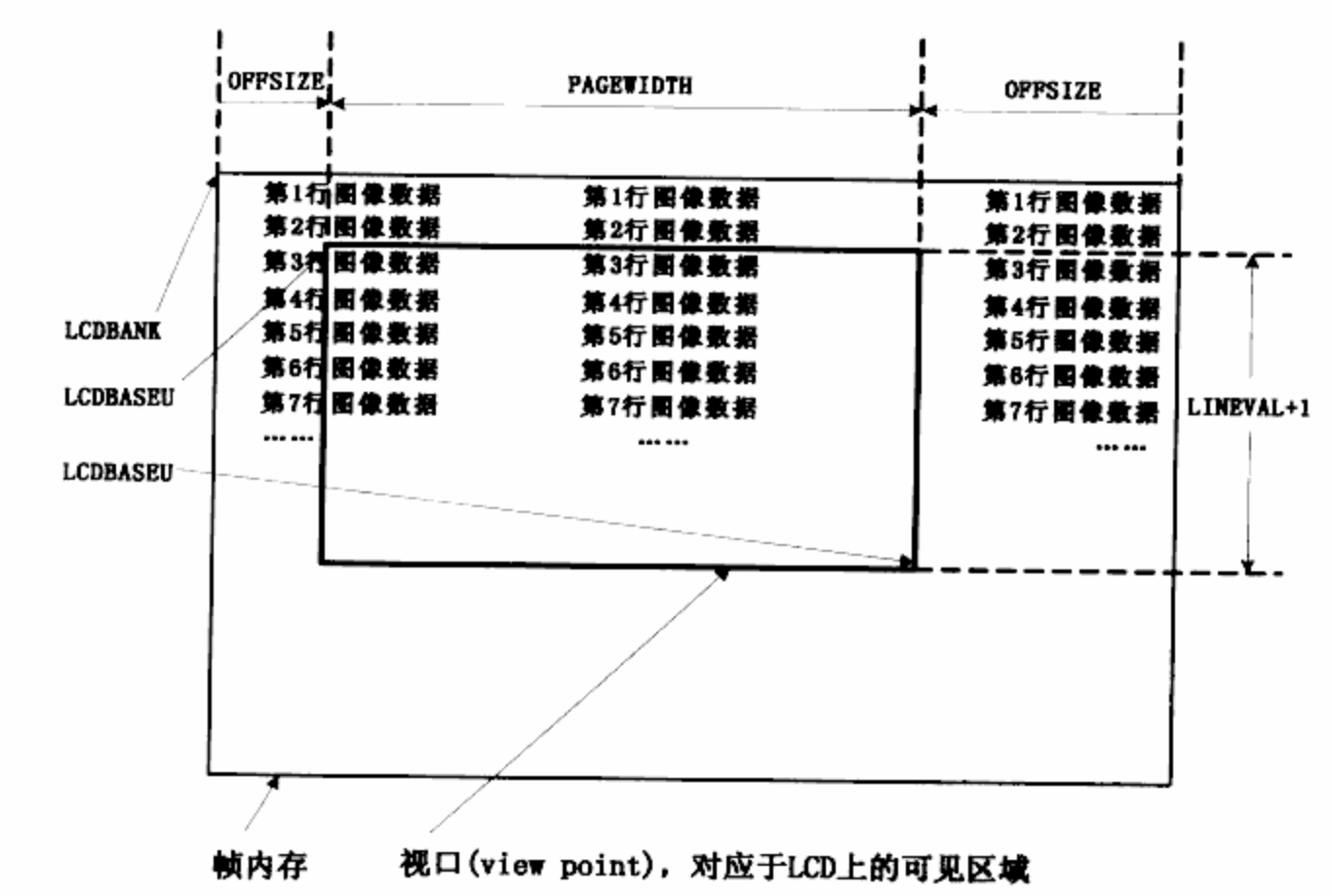


图 13.7　帧内存与视图的位置关系

各寄存器格式如表 13.11、13.12、13.13 所示。

表 13.11　　LCDSADDR1 寄存器格式

功　能	位	说　明
LCDBANK	[29:21]	用来保存帧内存起始地址 A[30:22]，帧内存起始地址必须为 4MB 对齐
LCDBASEU	[20:0]	对于 TFT LCD，用来保存视口（view point）所对应的内存起始地址 A[21:1]，这块内存也称为 LCD 的帧缓冲区（frame buffer）

表 13.12 LCDSADDR2 寄存器格式

功 能	位	说 明
LCDBASEL	[20:0]	对于 TFT LCD，用来保存 LCD 的帧缓冲区结束地址 A[21:1]，其值可如下计算： LCDBASEL = LCDBASEU + (PAGEWIDTH + OFFSIZE) × (LINEVAL + 1)

注：可以修改 LCDBASEU、LCDBASEL 的值来实现图像的移动，不过不能在一帧图像的结束阶段（LCDCON1 寄存器的 LINECNT 为 0 时）进行修改，因为此时 LCD 控制器会优先取得下一帧的数据，之后才改变这些值，这样的话，这些数据与新的帧缓冲区就不一致。

表 13.13 LCDSADDR3 寄存器格式

功 能	位	说 明
OFFSIZE	[21:11]	参考图 13.7，表示上一行最后一个数据与下一行第一个数据间地址差值的一半，即以半字为单位的地址差（0 表示两行数据是紧接着的，1 表示它们之间相差 2 个字节，依此类推）
PAGEWIDTH	[10:0]	视口（view point）的宽度，以半字为单位

注：OFFSIZE、PAGEWIDTH 的值只能在 ENVID (LCDCON1 寄存器的信号输出使能位)为 0 时修改。

（7）临时调色板寄存器 TPAL。

如果要输出一帧单色的图像，可以在 TPAL 寄存器中设定这个颜色值，然后使能 TPAL 寄存器，这种方法可以避免修改整个调色板或帧缓冲区。

TPAL 寄存器格式如表 13.14 所示。

表 13.14 TPAL 寄存器格式

功 能	位	说 明
TPALEN	[24]	调色板寄存器使能位。 0：禁止；1：使能
TPALVAL	[23:0]	颜色值。 TPALVAL[23:16] :红色 TPALVAL[15:8] :绿色 TPALVAL[7:0] :蓝色

注：临时调色板寄存器 TPAL 可以用在任何显示模式下，并非只能用在 8BPP 模式下。

13.2 TFT LCD 显示实例

13.2.1 程序设计

本实例的目的是从串口输出一个菜单，从中选择各种方法进行测试，比如画线、画圆、显示单色、使用调色板等。

13.2.2 代码详解

本实例源码在/work/hardware/lcd 目录下，与 LCD 相关的代码有 3 个文件：lcddrv.c、

framebuffer.c 和 lcdlib.c（及相应的头文件）。

（1）lcddrv.c 封装了对 LCD 控制器、调色板的访问函数，可以设置 LCD 的显示模式、开启/关闭 LCD、设置调色板等。

（2）framebuffer.c 直接操作帧缓冲区（frame buffer），实现了画点、画线、画同心圆、清屏等函数。

（3）lcdlib.c 调用前两个文件提供的函数在 LCD 上进行各种操作。

程序的结构如图 13.8 所示。

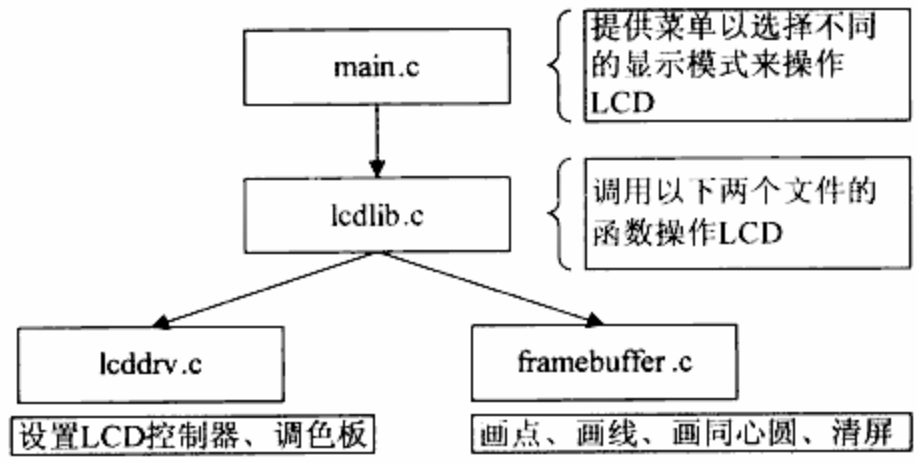


图 13.8 LCD 实例的程序结构

1. main.c

main.c 的代码很简单，其主体如下：

```
21          c = getc();
22          printf("%c\n\r", c);
23          switch (c)
24          {
25              case '1':
26              {
27                  Test_Lcd_Tft_8Bit_240320();
28                  break;
29              }
30
31              case '2':
32              {
33                  Test_Lcd_Tft_16Bit_240320();
34                  break;
35              }
36
37              case '3':
38              {
39                  Test_Lcd_Tft_8Bit_640480();
40                  break;
41              }
42
43              case '4':
44              {
45                  Test_Lcd_Tft_16Bit_640480();
46                  break;
```

```
47          }
48
```

它根据串口的输入选择是以“240×320、8bpp”、“240×320、16bpp”、“640×480、8bpp”还是“640×480、16bpp”的显示模式来操作LCD，所调用的4个函数Test_Lcd_Tft_8Bit_240320、Test_Lcd_Tft_16Bit_240320、Test_Lcd_Tft_8Bit_640480、Test_Lcd_Tft_16Bit_640480都在lcdlib.c中实现。

2. lcdlib.c

8BPP模式将用到调色板，其操作比16BPP模式稍为复杂，但是大部分仍是相似的。下面以Test_Lcd_Tft_8Bit_240320为例进行说明。

```
11 /*
12  * 以240×320、8bpp的显示模式测试TFT LCD
13  */
14 void Test_Lcd_Tft_8Bit_240320(void)
15 {
16     Lcd_Port_Init();                          // 设置LCD引脚
17     Tft_Lcd_Init(MODE_TFT_8BIT_240320);       // 初始化LCD控制器
18     Lcd_PowerEnable(0, 1);                    // 设置LCD_PWREN有效,它用于打开LCD
的电源
19     Lcd_EnvidOnOff(1);                        // 使能LCD控制器输出信号
20
```

第16行设置所涉及的GPIO引脚用于LCD功能。

第17行调用Tft_Lcd_Init函数初始化LCD控制器，即设置各个控制信号的时间特性、设置LCD的显示模式、设置帧缓冲区的地址等，它是lcddrv.c中最复杂的函数，在后面会详细分析这个函数。

进行第16、17行的初始化之后，只要“打开”LCD，帧缓冲区中的数据就会被LCD控制器自动地发送到LCD上去显示。“打开”操作由第18、19行来完成。

第18行发出LCD_PWREN信号。对于有电源开关控制引脚的LCD，可以使用LCD_PWREN来打开或关闭LCD。LCD_PWREN信号的极性可以设置。

第19行使能LCD控制器输出信号。这时，帧缓冲区中的数据就开始在LCD上显示出来了。

接下来就是按照设定的流程进行各类操作了，比如画线、清屏等，代码如下：

```
21     Lcd_Palette8Bit_Init();          // 初始化调色板
22     ClearScr(0x0);                   // 清屏
23     printf("[TFT 64K COLOR(16bpp) LCD TEST]\n");
24
25     printf("1. Press any key to draw line\n");
```

```
26    getc();
27    DrawLine(0  , 0  , 239, 0  , 0);        // 颜色为 DEMO256pal[0]
28    DrawLine(0  , 0  , 0  , 319, 1);        // 颜色为 DEMO256pal[1]
29    DrawLine(239, 0  , 239, 319, 2);        // …
30    DrawLine(0  , 319, 239, 319, 4);
31    DrawLine(0  , 0  , 239, 319, 8);
32    DrawLine(239, 0  , 0  , 319, 16);
33    DrawLine(120, 0  , 120, 319, 32);
34    DrawLine(0  , 160, 239, 160, 64);
35
36    printf("2. Press any key to draw circles\n");
37    getc();
38    Mire();
39
40    printf("3. Press any key to fill the screem with one color\n");
41    getc();
42    ClearScr(128);  //  输出单色图像，颜色值等于 DEMO256pal[128]
43
44    printf("4. Press any key to fill the screem by temporary palette\n");
45    getc();
46    ClearScrWithTmpPlt(0x0000ff);          // 输出单色图像，颜色为蓝色
47
48    printf("5. Press any key to fill the screem by palette\n");
49    getc();
50    DisableTmpPlt();                 // 关闭临时调色板寄存器
51    ChangePalette(0xffff00);         // 改变整个调色板为黄色，输出单色图像
52
53    printf("6. Press any key stop the testing\n");
54    getc();
55    Lcd_EnvidOnOff(0);
56 }
57
```

将上面调用的函数分为 3 类。

（1）清屏函数 ClearScr、画线函数 DrawLine，都是通过 framebuffer.c 中的 PutPixel 函数来设置帧缓冲区中的数据，以像素为单位修改颜色来实现的。

（2）Lcd_Palette8Bit_Init 函数设置调色板，ChangePalette 函数通过设置调色板来实现清屏功能，不涉及帧缓冲区，它在 lcddrv.c 中实现。

（3）ClearScrWithTmpPlt 函数则是通过临时调色板寄存器来快速地输出单色的图像，也不涉及帧缓冲区，它在 lcddrv.c 中实现。

lcddrv.c、framebuffer.c 文件中的各个函数才是本实例的关键。可以认为 lcddrv.c 是对操作各寄存器的封装，framebuffer.c 则是对操作图像数据的封装。先看 lcddrv.c 文件。

3. lcddrv.c

这个文件中的函数重点在于 Tft_Lcd_Init、Lcd_Palette8Bit_Init。

（1）Lcd_Port_Init 函数。

设置所涉及的 GPIO 引脚用于 LCD 功能。GPIO 功能的设置对读者来说已经很熟悉了，不再赘述。

（2）Tft_Lcd_Init 函数。

用于初始化 LCD 控制器，即设置各个控制信号的时间特性、设置 LCD 的显示模式、设置帧缓冲区的地址等。

首先是对 5 个控制寄存器 LCDCON1～5 的设置，代码如下：

```
34 /*
35  * 初始化 LCD 控制器
36  * 输入参数:
37  * type: 显示模式
38  *      MODE_TFT_8BIT_240320  : 240*320 8bpp 的 TFT LCD
39  *      MODE_TFT_16BIT_240320 : 240*320 16bpp 的 TFT LCD
40  *      MODE_TFT_8BIT_640480  : 640*480 8bpp 的 TFT LCD
41  *      MODE_TFT_16BIT_640480 : 640*480 16bpp 的 TFT LCD
42  */
43 void Tft_Lcd_Init(int type)
44 {
45     switch(type)
46     {
47     case MODE_TFT_8BIT_240320:
48         /*
49          * 设置 LCD 控制器的控制寄存器 LCDCON1~LCDCON5
50          * 1. LCDCON1
51          *    设置 VCLK 的频率: VCLK(Hz) = HCLK/[(CLKVAL+1)×2]
52          *    选择 LCD 类型: TFT LCD
53          *    设置显示模式: 8BPP
54          *    先禁止 LCD 信号输出
55          * 2. LCDCON2/3/4
56          *    设置控制信号的时间参数
57          *    设置分辨率，即行数及列数
58          * 现在，可以根据公式计算出显示器的频率
59          * 当 HCLK=100MHz 时，
60          * Frame Rate = 1/[{(VSPW+1)+(VBPD+1)+(LIINEVAL+1)+(VFPD+1)} ×
```

```
61          *              {(HSPW+1)+(HBPD+1)+(HFPD+1)+(HOZVAL+1)} ×
62          *              {2x(CLKVAL+1)/(HCLK)}]
63          *            = 60Hz
64          * 3. LCDCON5
65          *    设置显示模式为 8BPP 时，调色板中的数据格式为 5:6:5
66          *    设置 HSYNC、VSYNC 脉冲的极性(这需要参考具体 LCD 的接口信号)：反转
67          *    字节交换使能
68          */
69         LCDCON1 = (CLKVAL_TFT_240320<<8) | (LCDTYPE_TFT<<5) | \
70                   (BPPMODE_8BPP<<1) | (ENVID_DISABLE<<0);
71         LCDCON2 = (VBPD_240320<<24) | (LINEVAL_TFT_240320<<14) | \
72                   (VFPD_240320<<6) | (VSPW_240320);
73         LCDCON3 = (HBPD_240320<<19) | (HOZVAL_TFT_240320<<8) | (HFPD_240320);
74         LCDCON4 = HSPW_240320;
75         LCDCON5 = (FORMAT8BPP_565<<11) | (HSYNC_INV<<9) | (VSYNC_INV<<8) | \
76                   (BSWP<<1);
77
```

代码中的注释可以帮助读者理解这些代码，比较困难的是时间参数 VSPW、VBPD、VFPD、HSPW、HBPD、HFPD、CLKVAL 的设置。对于 CRT 显示器，当它的频率在 60Hz 时，人眼会感到明显的闪烁；而对于 LCD，在 60Hz 时显示效果就已经很好。这些值可以从 LCD 的数据手册了解到，或使用经验值，或自行调整，并根据下述公式确认显示频率在 60Hz 左右或之上。

```
Frame Rate = 1/ [ { (VSPW+1) + (VBPD+1) + (LIINEVAL + 1) + (VFPD+1) } ×
        {(HSPW+1) + (HBPD +1)+ (HFPD+1) + (HOZVAL + 1) }×{ 2 ×( CLKVAL+1 ) /
( HCLK ) } ]
```

接下来是对地址寄存器 LCDSADDR1～LCDSADDR3 的设置，请参考图 13.7 帧内存与视图的位置关系。在本程序中，帧内存与视图吻合，即图中的 OFFSIZE 为 0，LCDBANK、LCDBASEU 指向同一个地址（它们是同一地址的不同位）。

需要注意是，8BPP 的显示模式要用到调色板，帧缓冲区中的数据不是像素的颜色值，而调色板中的索引值，真正的颜色值在调色板中。

代码如下：

```
78         /*
79          * 设置 LCD 控制器的地址寄存器 LCDSADDR1～LCDSADDR3
80          * 帧内存与视口(view point)完全吻合，
81          * 图像数据格式如下(8BPP 时，帧缓冲区中的数据为调色板中的索引值)：
82          *        |----PAGEWIDTH----|
83          *   y/x  0   1   2      239
```

```
84          *     0   idx idx idx ... idx
85          *     1   idx idx idx ... idx
86          * 1. LCDSADDR1
87          *    设置LCDBANK、LCDBASEU
88          * 2. LCDSADDR2
89          *    设置LCDBASEL: 帧缓冲区的结束地址A[21:1]
90          * 3. LCDSADDR3
91          *    OFFSIZE等于0, PAGEWIDTH等于(240/2)
92          */
93         LCDSADDR1 = ((LCDFRAMEBUFFER>>22)<<21) | LOWER21BITS (LCDFRAMEBUFFER>>1);
94         LCDSADDR2 = LOWER21BITS((LCDFRAMEBUFFER+ \
95                   (LINEVAL_TFT_240320+1) ×(HOZVAL_TFT_240320+1) ×1)>>1);
96         LCDSADDR3 = (0<<11) | (LCD_XSIZE_TFT_240320/2);
97
```

第93行将帧缓冲区的开始地址写入LCDSADDR1寄存器。

第94行先计算帧缓冲区的结束地址，再取其位[21:1]存入LCDSADDR2中。这个地址值在本实例中即是“LCDFRAMEBUFFER + 320 × 240 × 1”，其中的“×1”表示在8BPP中一个像素使用1个字节来表示（对于16BPP，就是“×2”）。

在设置寄存器的最后，禁止临时调色板寄存器，现在还没用到它。

```
98         /* 禁止临时调色板寄存器 */
99         TPAL = 0;
100
```

最后，将显示模式的主要参数记录下来，在framebuffer.c中需要用到。

```
101        fb_base_addr = LCDFRAMEBUFFER;
102        bpp = 8;
103        xsize = 240;
104        ysize = 320;
105
```

其他显示模式的寄存器设置非常相似，不再赘述。

需要说明的是，显示模式为8BPP时，LCDCON5中BSWAP位设为1，表示“字节交换使能”，这时帧缓冲区中的数据与屏幕上的像素位置关系如图13.6所示；显示模式为16BPP时，LCDCON5中HWSWAP位设为1，表示“半字交换使能”，这时帧缓冲区中的数据与屏幕上的像素位置关系如图13.5所示。它们都是“低地址的数据”对应“位置靠前”的像素。

（3）Lcd_Palette8Bit_Init函数。

设置调色板中的数据：调试板大小为256 × 16，而8BPP模式中每个像素的索引值占据8位，刚好有256个索引值。代码如下：

```
296 /*
297  * 设置调色板
298  */
299 void Lcd_Palette8Bit_Init(void)
300 {
301     int i;
302     volatile unsigned int *palette;
303
304     LCDCON5 |= (FORMAT8BPP_565<<11); // 设置调色板中数据格式为 5:6:5
305
306     palette = (volatile unsigned int *)PALETTE;
307     for (i = 0; i < 256; i++)
308         *palette++ = DEMO256pal[i];
309 }
310
```

调色板中用 16BPP 的格式表示颜色，第 304 行设置调色板中数据的格式为 5:6:5（另一种格式为 5:5:5:1）。

第 307、308 行将数组 DEMO256pal 中的数据写入调色板中。这个数组中的数据并没有什么特别之处，读者可以自己构造。

（4）ChangePalette 函数。

以给定的颜色值填充整个调色板，代码如下：

```
311 /*
312  * 改变调色板为一种颜色
313  * 输入参数:
314  *     color: 颜色值，格式为 0xRRGGBB
315  */
316 void ChangePalette(UINT32 color)
317 {
318     int i;
319     unsigned char red, green, blue;
320     UINT32 *palette;
321
322     palette=(UINT32 *)PALETTE;
323     for (i = 0; i < 256; i++)
324     {
325         red   = (color >> 19) & 0xff;
326         green = (color >> 10) & 0xff;
327         blue  = (color >>  3) & 0xff;
```

```
328         color = (red << 11) | (green << 5) | blue; // 格式 5:6:5
329
330         while ((LCDCON5>>16) == 2);     // 等待直到 VSTATUS 不为“有效”
331         *palette++ = color;
332     }
333 }
334
```

第 325～328 行从 0xRRGGBB 格式的 color 变量中，提取 8 位红色值的高 5 位、8 位绿色值的高 6 位、8 位蓝色值的高 5 位组成 5:6:5 格式的 16BPP 颜色值。

第 330 行检测当前 VSYNC 信号的状态，如果它处于有效的状态，则等待。前面说过，读写调色板时，VSTATUS、HSTATUS 不能处于有效状态。这里当 VSTATUS 不是“有效”状态时，HSTATUS 也不可能是“有效”状态。

第 331 行将新数据写入调色板。

（5）Lcd_PowerEnable 函数。

用于控制是否发出 LCD_PWREN 信号。对于有电源开关控制引脚的 LCD，可以使用 LCD_PWREN 来打开或关闭 LCD。LCD_PWREN 信号的极性可以设置。代码如下：

```
335 /*
336  * 设置是否输出 LCD 电源开关信号 LCD_PWREN
337  * 输入参数:
338  *     invpwren: 0 表示 LCD_PWREN 有效时为正常极性
339  *               1 表示 LCD_PWREN 有效时为反转极性
340  *     pwren:    0 表示 LCD_PWREN 输出有效
341  *               1 表示 LCD_PWREN 输出无效
342  */
343 void Lcd_PowerEnable(int invpwren, int pwren)
344 {
345     GPGCON = (GPGCON & (~(3<<8))) | (3<<8);   // GPG4 用做 LCD_PWREN
346     GPGUP  = (GPGUP & (~(1<<4))) | (1<<4);    // 禁止内部上拉
347
348     LCDCON5 = (LCDCON5 & (~(1<<5))) | (invpwren<<5);  // 设置 LCD_PWREN 的
极性: 正常/反转
349      LCDCON5 = (LCDCON5 & (~(1<<3))) | (pwren<<3);      // 设置是否输出
LCD_PWREN
350 }
351
```

（6）Lcd_EnvidOnOff 函数。

用于控制是否使能 LCD 控制器输出各个 LCD 信号，当设置如控制寄存器、地址寄存器之后，即可调用此函数输出各个 LCD 信号，这样，帧缓冲区中的数据即发送给 LCD。代码

如下：

```
352 /*
353  * 设置 LCD 控制器是否输出信号
354  * 输入参数:
355  * onoff:
356  *      0 : 关闭
357  *      1 : 打开
358  */
359 void Lcd_EnvidOnOff(int onoff)
360 {
361     if (onoff == 1)
362         LCDCON1 |= 1;         // ENVID ON
363     else
364         LCDCON1 &= 0x3fffe; // ENVID Off
365 }
366
```

（7）ClearScrWithTmpPlt、DisableTmpPlt 函数。

请参考表 13.13 TPAL 寄存器格式，ClearScrWithTmpPlt 函数设置颜色值并使能 TPAL 寄存器，这使得 LCD 上显示单一颜色的图像。DisableTmpPlt 函数停止 TPAL 寄存器的功能，继续输出帧缓冲区中的图像。它们的代码如下：

```
367 /*
368  * 使用临时调色板寄存器输出单色图像
369  * 输入参数:
370  *     color: 颜色值，格式为 0xRRGGBB
371  */
372 void ClearScrWithTmpPlt(UINT32 color)
373 {
374     TPAL = (1<<24)|((color & 0xffffff)<<0);
375 }
376
377 /*
378  * 停止使用临时调色板寄存器
379  */
380 void DisableTmpPlt(void)
381 {
382     TPAL = 0;
383 }
384
```

4. framebuffer.c

此文件中有4个函数：画点函数PutPixel、画线函数DrawLine、绘制同心圆函数Mire、清屏函数ClearScr，后3个函数都是基于PutPixel函数实现的。PutPixel函数是framebuffer.c文件的核心，它在帧缓冲区中找到给定坐标的像素的内存，然后修改它的值，代码如下：

```
08 extern unsigned int fb_base_addr;
09 extern unsigned int bpp;
10 extern unsigned int xsize;
11 extern unsigned int ysize;
12
13 /*
14  * 画点
15  * 输入参数:
16  *     x、y : 像素坐标
17  *     color: 颜色值
18  *     对于16BPP: color的格式为0xAARRGGBB (AA = 透明度),
19  *     需要转换为5:6:5格式
20  *     对于8BPP: color为调色板中的索引值,
21  *     其颜色取决于调色板中的数值
22  */
23 void PutPixel(UINT32 x, UINT32 y, UINT32 color)
24 {
25     UINT8 red,green,blue;
26
27     switch (bpp){
28         case 16:
29         {
30             UINT16 *addr = (UINT16 *)fb_base_addr + (y * xsize + x);
31             red   = (color >> 19) & 0xff;
32             green = (color >> 10) & 0xff;
33             blue  = (color >>  3) & 0xff;
34             color = (red << 11) | (green << 5) | blue; // 格式5:6:5
35             *addr = (UINT16) color;
36             break;
37         }
38
39         case 8:
40         {
41             UINT8 *addr = (UINT8 *)fb_base_addr + (y * xsize + x);
```

```
42              *addr = (UINT8) color;
43              break;
44          }
45
46          default:
47              break;
48      }
49  }
50
```

第 8～11 行的 4 个变量在 lcddrv.c 中的 Tft_Lcd_Init 函数中设置，PutPixel 函数根据它们来确定给定坐标的像素在帧缓冲区中的地址。

第 30、41 行分别计算 16BPP、8BPP 模式下给定坐标的像素在帧缓冲区中的地址。对于 16BPP 模式，每个像素占据 2 字节；对于 8BPP 模式，每个像素占据 1 字节。

对于 16BPP 的显示模式，第 31～34 行从 0xAARRGGBB 格式的 color 变量中，提取 8 位红色值的高 5 位、8 位绿色值的高 6 位、8 位蓝色值的高 5 位组成 5:6:5 格式的 16BPP 颜色值。

最后，第 35、42 行将颜色值（对于 8BPP 模式，为调色板中的索引值）写入帧缓冲区中，这样，在下一次显示的时候，新颜色即可显示出来。

13.2.3 实例测试

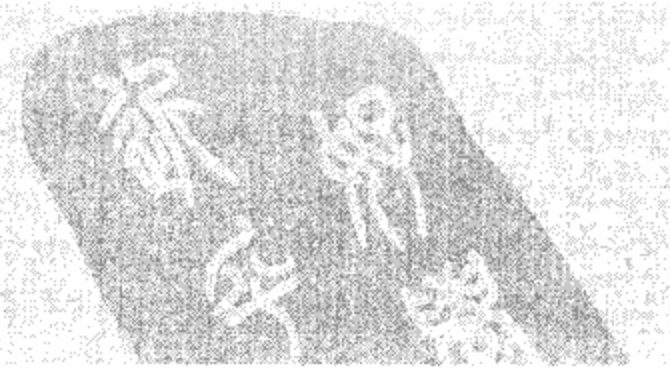
本程序在 main 函数中通过串口输出一个菜单，用于选择 LCD 的显示模式并进行测试。实验步骤如下。

（1）使用串口将开发板的 COM0 和 PC 的串口相连，打开 PC 上的串口工具并设置其波特率为 115200、8N1。

（2）在 LCD 目录下执行 make 命令生成 lcd 可执行程序，烧入 NAND Flash 后运行。

（3）在 PC 串口工具上，可以看到如下菜单：

```
##### Test TFT LCD #####
[1] TFT240320 8Bit
[2] TFT240320 16Bit
[3] TFT640480 8Bit
[4] TFT640480 16Bit
Enter your selection:
```



（4）可以输入 1、2、3 或 4，然后按照提示输入任意键即可一步一步地观察到 LCD 中图像的变化。

（5）最后又会出现第（3）步骤的菜单，可以再次选择。

第 14 章　ADC 和触摸屏接口

本章目标

- 了解 S3C2410/S3C2440 ADC 和触摸屏的结构
- 了解电阻触摸屏的工作原理和等效电路图
- 了解 S3C2410/S3C2440 触摸屏控制器的多种工作模式
- 掌握 S3C2410/S3C2440 ADC 和触摸屏的编程方法

14.1　ADC 和触摸屏硬件介绍及使用

14.1.1　S3C2410/S3C2440 ADC 和触摸屏接口概述

S3C2410/S3C2440 的 CMOS 模数转换器（ADC，Analog to Digital Converter）可以接收 8 个通道的模拟信号输入，并将它们转换为 10 位的二进制数据。在 2.5MHz 的 A/D 转换时钟下，最大的转化速率可达 500KSPS（SPS：samples per second，每秒采样的次数）。

S3C2410/S3C2440 都提供触摸屏的接口，不过它们有所不同。S3C2410 的触摸屏接口向外提供 4 个控制信号引脚（nYPON、YMON、nXPON、XMON）和 2 个模拟信号输入引脚（AIN[7]、AIN[5]），这 6 个引脚通过 4 个晶体管与触摸屏的 4 个引脚相连。而 S3C2440 提供了与触摸屏直接相连的 4 个引脚，不再需要外接晶体管。

S3C2410/S3C2440 ADC 和触摸屏接口有如下特性。

- 分辨率：10 位。
- 微分线性度误差：±1.0 LSB。
- 积分线性度误差：±2.0 LSB。
- 最大转换速率：500KSPS。
- 低功耗。
- 供电电压：3.3V。
- 输入模拟电压范围：0 ~ 3.3V。
- 片上采样保持功能（On-chip sample-and-hold function）。
- 普通转换模式。

- 分离的 *x*/*y* 轴坐标转换模式。
- 自动（连续）*x*/*y* 轴坐标转换模式。
- 等待中断模式。

ADC 和触摸屏接口结构如图 14.1、14.2 所示。

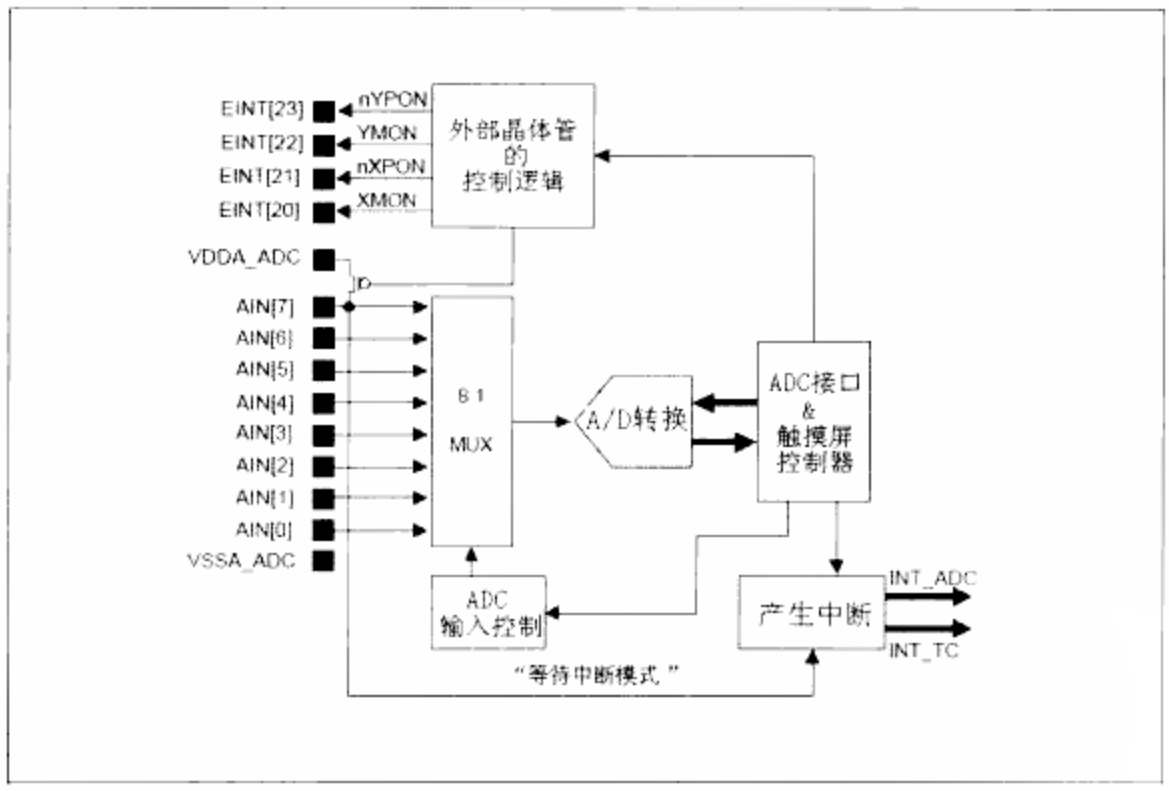


图 14.1　S3C2410 ADC 和触摸屏接口结构

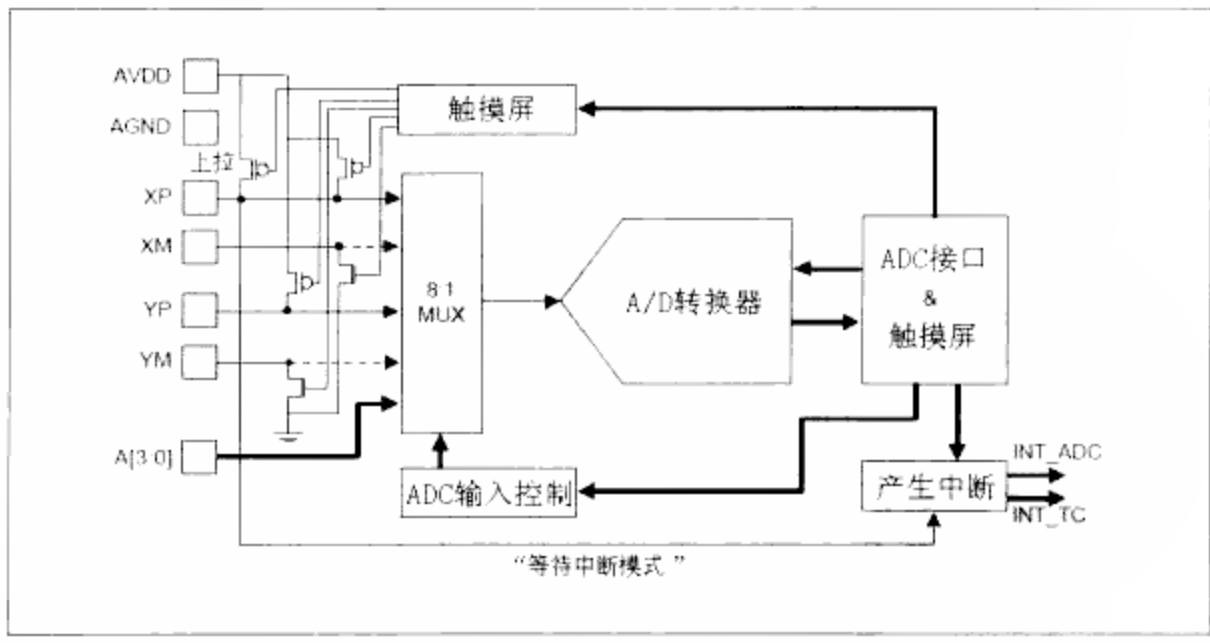


图 14.2　S3C2440 ADC 和触摸屏接口结构

从图 14.1、14.2 可以知道，ADC 和触摸屏接口中只有一个 A/D 转换器（A/D Converter），可以通过设置寄存器来选择对哪路模拟信号（多达 8 路）进行采样。图中有两个中断信号：INT_ADC、INT_TC，前者表示 A/D 转换器已经转换完毕，后者表示触摸屏被按下了。

对于 S3C2410，在使用触摸屏时，AIN[7]和 AIN[5]被用来测量 XP、YP 的电平，只剩下 AIN[6]、AIN[4:0]共 6 个引脚用于一般的 ADC 输入。对于 S3C2440，在使用触摸屏时，引脚 XP、XM、YP 和 YM 被用于和触摸屏直接相连，只剩下 AIN[3:0]共 4 个引脚用于一般的 ADC 输入；当不使用触摸屏时，XP、XM、YP 和 YM 这 4 个引脚也可以用于一般的 ADC

输入。

S3C2410与触摸屏的连接比S3C2440复杂，需要增加几个外接晶体管，如图14.3所示。

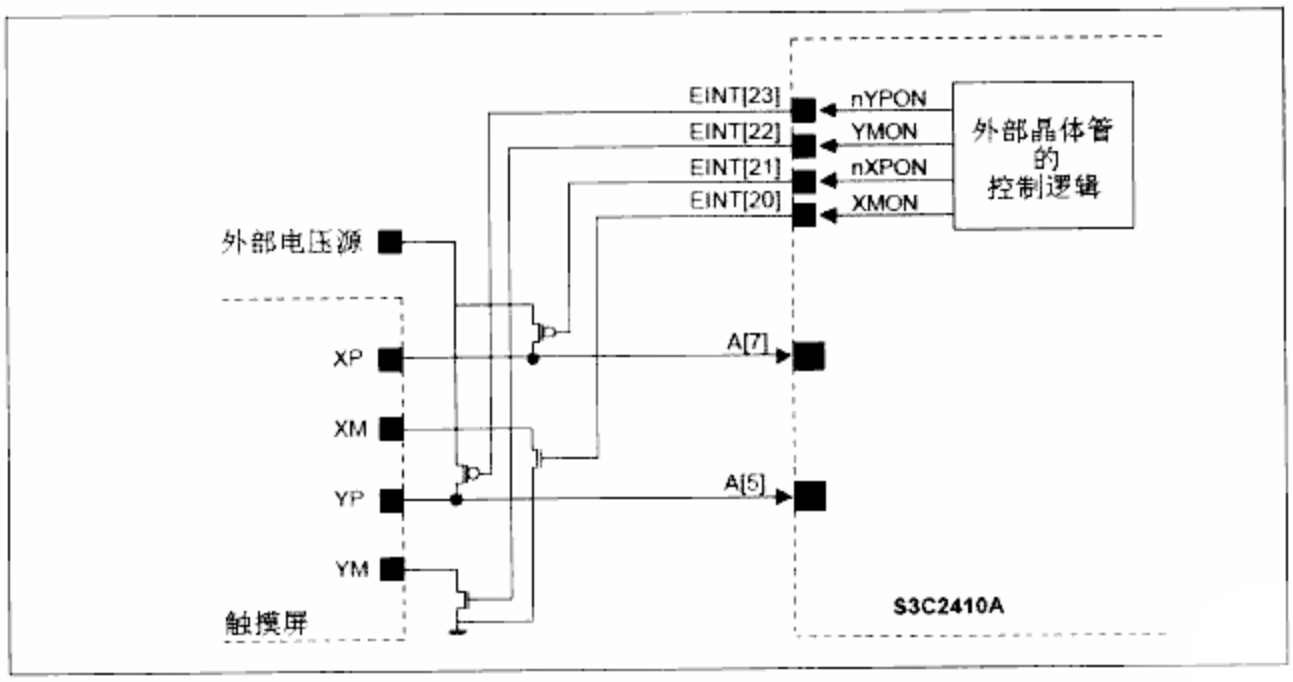


图14.3 S3C2410与触摸屏的连接电路

14.1.2 S3C3410/S3C2440 ADC接口的使用方法

ADC的启动方式有两种：手工启动、读结果时就自动地启动下一次转换。也有两种方法获知当前转换是否已经结束：查询状态位、转换结束时发出中断。

ADC的操作只涉及3个寄存器：ADCCON、ADCTSC、ADCDAT0。下面介绍它们的用法，有关触摸屏的数据位将在14.1.3小节介绍。

这两个寄存器的格式如表14.1、14.2所示。

表14.1 ADCCON寄存器的格式

名　称	位	说　明
ECFLG	[15]	只读，A/D转换结束标志。 0：正在转换；1：转换结束
PRSCEN	[14]	决定A/D转换器的时钟是否预分频。 0：不使用；1：使用
PRSCVL	[13:6]	预分频系数，取值0～255：A/D时钟 = PCLK/(PRSCVL + 1)。 注意：A/D时钟必须小于PCLK的1/5
SEL_MUX	[5:3]	选择进行A/D转换的通道。 对于S3C2410，取值如下。 000：AIN 0；001：AIN 1；010：AIN 2；011：AIN 3； 100：AIN 4；101：AIN 5；110：AIN 6；111：AIN 7(XP) 对于S3C2440，取值如下。 000：AIN 0；001：AIN 1；010：AIN 2；011：AIN 3； 100：YM；　101：YP；　110：XM；　111：XP

续表

名　称	位	说　明
STDBM	[2]	选择静态模式（Standby mode）。 0：正常模式；1：静态模式
READ_START	[1]	读转换数据时是否启动下一次转换。 0：不启动；1：启动
ENABLE_START	[0]	启动 A/D 转换（当 READ_START 为 1 时，此位无效）。 0：无作用；1：启动 A/D 转换（转换真正开始时，此位被清 0）

表 14.2　ADCDAT0 寄存器的格式

名　称	位	说　明
UPDOWN	[15]	对于触摸屏，使用“等待中断模式”时 0：触摸屏被按下；1：触摸屏没有被按下
AUTO_PST	[14]	决定是否使用自动（连续）*x*/*y* 轴坐标转换模式。 0：正常转换；1：自动（连续）*x*/*y* 轴坐标转换
XY_PST	[13:12]	手动 *x*/*y* 轴坐标转换模式。 00：无操作；　01：*x* 轴坐标转换 10：*y* 轴坐标转换；11：等待中断模式
Reserved	[11:10]	保留
XPDATA （普通 ADC 转换数据）	[9:0]	*x* 轴坐标转换数据值（或普通 ADC 转换数据值） 数值范围：0～0x3FF

ADC 的使用分 4 个步骤。

（1）设置 ADCCON 寄存器，选择输入信号通道，设置 A/D 转换器的时钟。

使能 A/D 转换器时钟的预分频功能时，A/D 时钟的计算公式如下：

```
A/D 时钟 = PCLK / ( PRSCVL + 1)
```

注意　A/D 时钟最大为 2.5MHz，并且应该小于 PCLK 的 1/5。

（2）设置 ADCTSC 寄存器，使用设为普通转换模式，不使用触摸屏功能。

ADCTSC 寄存器多用于触摸屏，对于普通 ADC，使用它的默认值即可，或设置其位[2]为 0。ADCTSC 寄存器的格式在 14.1.3 小节介绍。

（3）设置 ADCCON 寄存器，启动 A/D 转换。

如果设置 READ_START 位，则读转换数据（读 ADCDAT0 寄存器）时即启动下一次转换；否则，可以通过设置 ENABLE_START 位来启动 A/D 转换。

（4）转换结束时，读取 ADCDAT0 寄存器获得数值。

如果使用查询方式，则可以不断读取 ADCCON 寄存器的 ECFLG 位来确定转换是否结束；

否则可以使用 INT_ADC 中断，发生 INT_ADC 中断时表示转换结束。

14.1.3 触摸屏原理及接口

1. 电阻触摸屏的原理

触摸屏已经在现实生活中大量使用，比如银行的自动存/取款机等。触摸屏的种类有很多，比如超声波触摸屏、红外触摸屏、电容触摸屏、电阻触摸屏等。电阻触摸屏由于造价低廉，在电气上可以直接接入用户的系统中而得到大量使用。电阻触摸屏有几种类型，比如“四线”，“五线”和“八线”。线越多，精度就越高，温度漂移也越少，但基本的操作是一样的。它本质上是个电阻分压器，将矩形区域中触摸点（x,y）的物理位置转换为代表 x 坐标和 y 坐标的电压。

S3C2410/S3C2440 的触摸屏接口可以驱动四线电阻触摸屏，四线电阻触摸屏的等效电路如图 14.4 所示。图中粗黑线表示相互绝缘的两层导电层，当按压时，它们在触点处相连；不同的触点在 x、y 方向上的分压值不一样，将这两个电压值经过 A/D 转换后即可得到 x、y 坐标。

下面根据其等效电路说明触摸屏的工作过程。

（1）平时触摸屏没有被按下时，等效电路如图 14.5 所示。

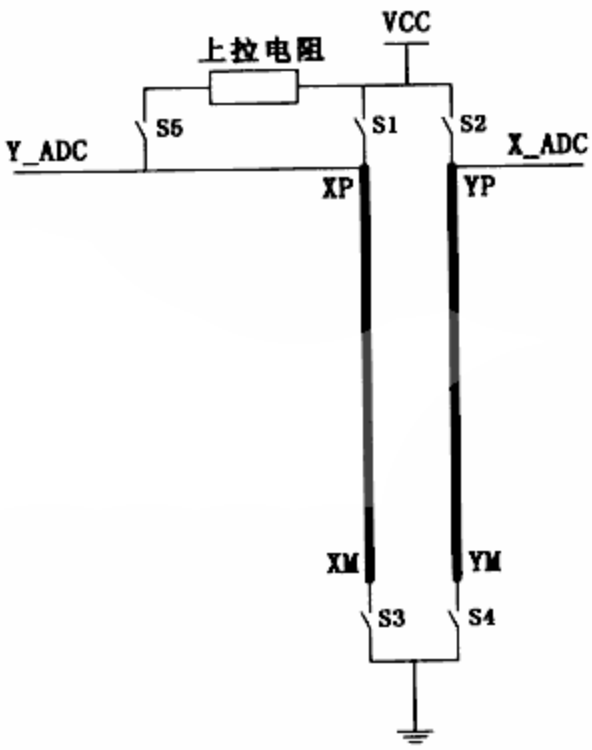


图 14.4 四线电阻触摸屏的等效电路

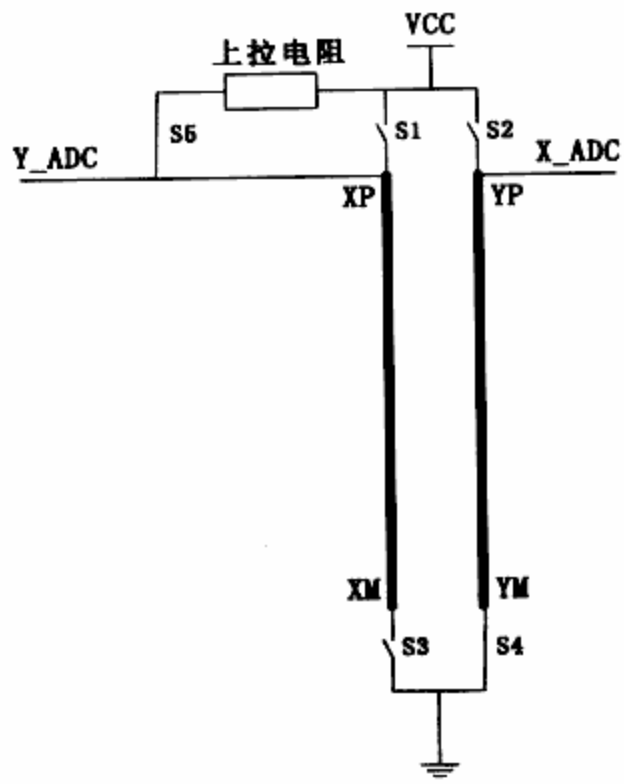


图 14.5 触摸屏处于“等待中断模式”时的等效电路

S4、S5 闭合，S1、S2、S3 断开，即 YM 接地、XP 上拉、XP 作为模拟输入（对 CPU 而言）、YP 作为模拟输入（对 CPU 而言）、XM 高阻。

平时触摸屏没有被按下时，由于上拉电阻的关系，Y_ADC 为高电平；当 x 轴和 y 轴受挤压而接触导通后，Y_ADC 的电压由于连通到 y 轴接地而变为低电平，此低电平可做为中断触发信号来通知 CPU 发生“Pen Down”事件，在 S3C2410/S3C2440 中，称为等待中断模式。

（2）采样 X_ADC 电压，得到 x 坐标，等效电路如图 14.6 所示。

S1、S3 闭合，S2、S4、S5 断开，即 XP 接上电源、XM 接地、YP 作为模拟输入（对 CPU 而言）、YM 高阻、XP 禁止上拉。这时，YP 即 X_ADC 就是 x 轴的分压点，进行 A/D 转换后就得到 x 坐标。

（3）采样 Y_ADC 电压，得到 y 坐标，等效电路如图 14.7 所示。

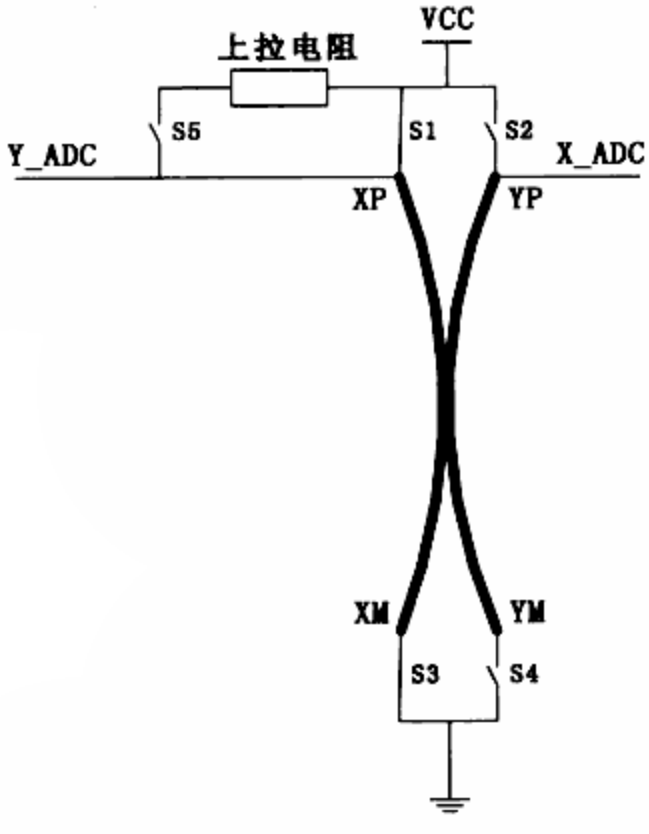


图 14.6　读取 x 坐标时的等效电路

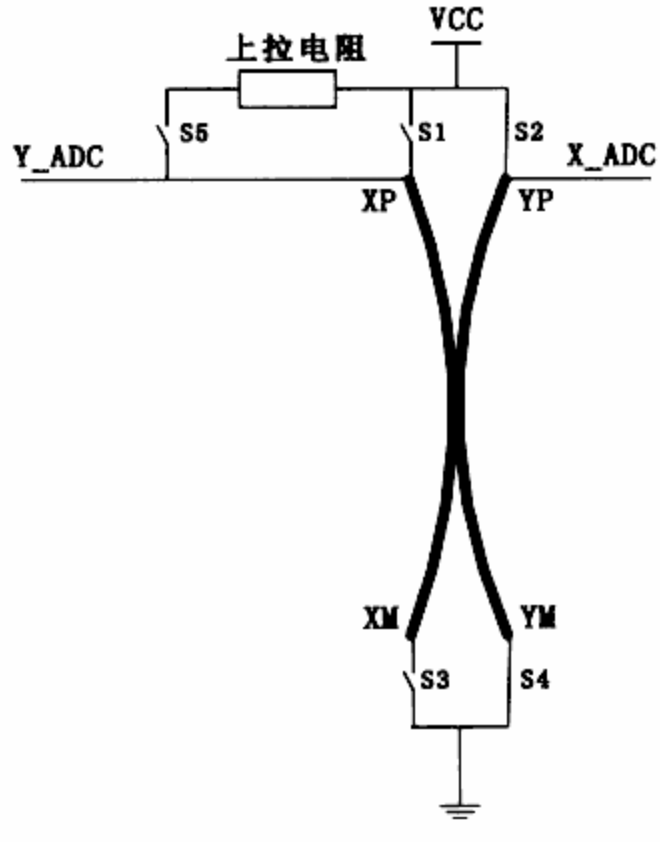


图 14.7　读取 y 坐标时的等效电路

S2、S4 闭合，S1、S3、S5 断开，即 YP 接上电源、YM 接地、XP 作为模拟输入（对 CPU 而言）、XM 高阻、XP 禁止上拉。这时，XP 即 Y_ADC 就是 y 轴的分压点，进行 A/D 转换后就得到 y 坐标。

2. S3C2410/S3C2440 触摸屏接口

与上面描述的触摸屏工作过程的 3 个步骤对应，触摸屏控制器也有 4 种工作模式。

（1）等待中断模式（Waiting for Interrupt Mode）。

设置 ADCTSC 寄存器为 0xD3 即可令触摸屏控制器处于这种模式。这时，它在等待触摸屏被按下。当触摸屏被按下时，触摸屏控制器将发出 INT_TC 中断信号，这时触摸屏控制器要转入以下两种工作模式中的一种，以读取 x、y 坐标。

对于 S3C2410，当触摸屏被按下（Pen Down）或松开（Pen Up）时，都产生 INT_TC 中断信号。

对于 S3C2440，可以设置 ADCTSC 寄存器的位[8]为 0 或 1 时，表示等待 Pen Down 中断或 Pen Up 中断。

（2）分离的 x/y 轴坐标转换模式（Separate X/Y Position Conversion Mode）。

这分别对应上述触摸屏工作过程的第 2、3 步骤。设置 ADCTSC 寄存器为 0x69 进入 x 轴坐标转换模式，x 坐标值转换完毕后被写入 ADCDAT0，然后发出 INT_ADC 中断；相似地，设置 ADCTSC 寄存器为 0x9A 进入 y 轴坐标转换模式，y 坐标值转换完毕后被写入 ADCDAT1，然后发出 INT_ADC 中断。

（3）自动（连续）*x/y* 轴坐标转换模式（Auto(Sequential)x/yPosition Conversion Mode）。

上述触摸屏工作过程的第2、3步骤可以合成一个步骤，设置ADCTSC寄存器为0x0C，进入自动（连续）*x/y* 轴坐标转换模式，触摸屏控制器就会自动转换触点的 *x*、*y* 坐标值，并分别写入ADCDAT0、ADCDAT1寄存器中，然后发出INT_ADC中断。

（4）普通转换模式（Normal Conversion Mode）。

不使用触摸屏时，触摸屏控制器处于这种模式。在这种模式下，可以通过设置ADCCON寄存器启动普通的A/D转换，转换完成时数据被写入ADCDAT0寄存器中。

14.1.2小节讲述ADC接口就是工作于这种模式。

ADCTSC寄存器被用来选择触摸屏的工作模式，它的格式如表14.3所示。

表14.3 ADCTSC寄存器的格式

名　称	位	说　明
Reserved/UD_SEN	[8]	对于S3C2410，此位必须为0。 对于S3C2440，此位表示将检测哪类中断（触点按下、触点松开）。 0：按下（Stylus Down/Pen Down）；1：松开（Stylus Up/Pen Up）
YM_SEN	[7]	对于S3C2410，用于选择YOMN的输出值。 0：YMON输出0（YM：高阻）；1：YMON输出1（YM：GND） 对于S3C2440，YM开关使能。 0：YM驱动禁止（高阻）；1：YM驱动使能（接地）
YP_SEN	[6]	对于S3C2410，用于选择nYPON的输出值。 0：nYPON输出0（YP：外部电压）；1：nYPON输出1（YM接AIN[5]） 对于S3C2440，YP开关使能。 0：YP驱动禁止（接外部电压）；1：YP驱动使能（接模拟输入）
XM_SEN	[5]	对于S3C2410，用于选择XOMN的输出值。 0：XMON输出0（XM：高阻）；1：XMON输出1（XM：GND） 对于S3C2440，XM开关使能。 0：XM驱动禁止（高阻）；1：XM驱动使能（接地）
XP_SEN	[4]	对于S3C2410，用于选择nXPON的输出值。 0：nXPON输出0（XP：外部电压）；1：nXPON输出1（XM接AIN[7]） 对于S3C2440，XP开关使能。 0：XP驱动禁止（接外部电压）；1：XP驱动使能（接模拟输入）
PULL_UP	[3]	XP上拉使能。 0：使能上拉；1：禁止上拉
AUTO_PST	[2]	是否使用自动（连续）*x/y* 轴坐标转换模式。 0：普通转换模式；1：自动（连续）*x/y* 轴坐标转换模式
XY_PST	[1:0]	手动测量 *x*、*y* 轴坐标。 00：无操作模式；01：测量 *x* 轴坐标； 10：测量 *y* 轴坐标；11：等待中断模式

注意 ① 处于等待中断模式时，XP_SEN 必须设为 1（XP 接模拟输入），PULL_UP 必须设为 0（使能上拉）。

② AUTO_PST 设为 1 时，必须处于自动（连续）*x/y* 轴坐标转换模式下。

对于 S3C2410，当触摸屏控制器处于等待中断模式时，触摸屏被按下时可以不断发出 INT_TC 中断信号以便进入自动（连续）*x/y* 轴坐标转换模式转换 *x*、*y* 坐标。发出中断信号的间隔可以通过 ADCDLY 寄存器来设置。

对于 S3C2440，当 CPU 处于休眠模式时，触摸屏被按下时可以不断发出 INT_TC 中断信号以唤醒 CPU。发出中断信号的间隔可以通过 ADCDLY 寄存器来设置。

另外，对于普通转换模式、分离的 *x/y* 轴坐标转换模式、自动（连续）*x/y* 轴坐标转换模式，都可以通过 ADCDLY 寄存器来设置采样的延时时间。

ADCDLY 寄存器格式如表 14.4 所示，在等待中断模式时，延时时钟为 X-tal(3.68MHz)，其他情况为 PCLK，可以如图 14.8 所示。

表 14.4　　ADCDLY 寄存器的格式

名　称	位	说　明
DELAY	[15:0]	采样的延时值，或发出中断的间隔值

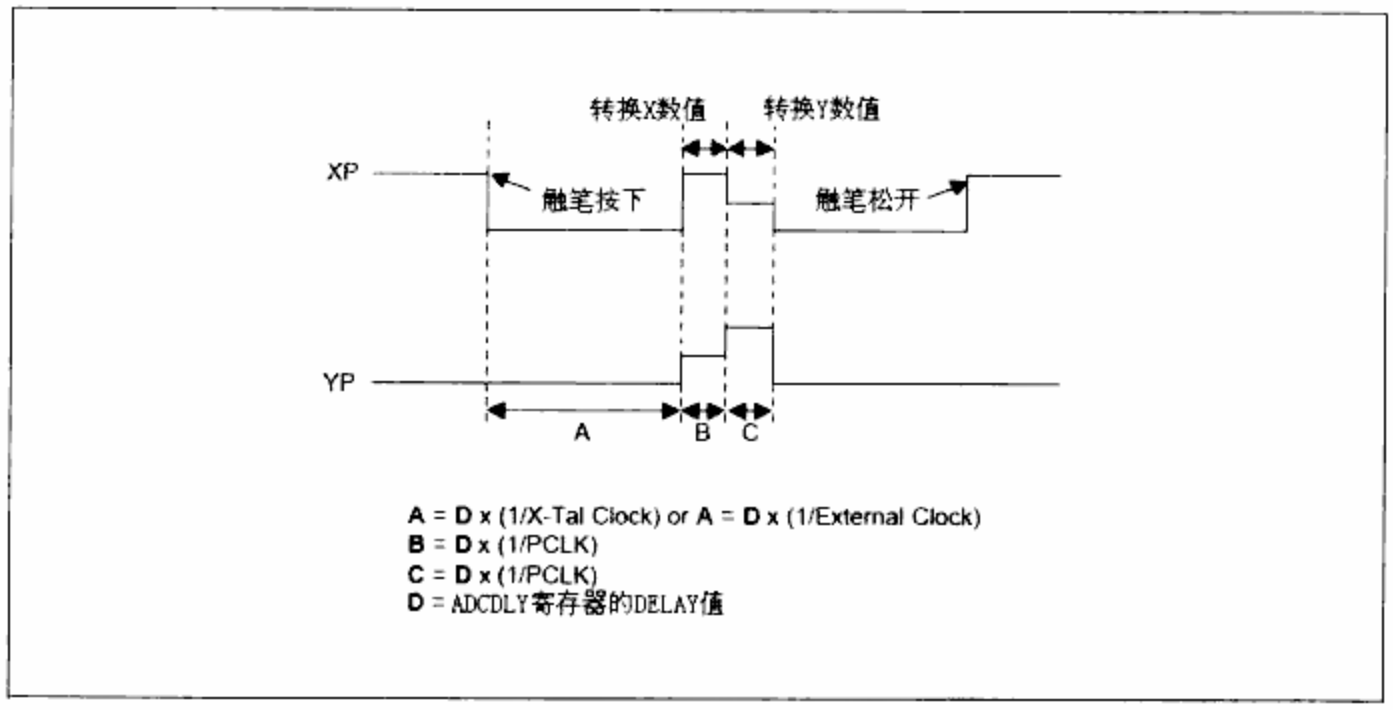


图 14.8　ADC 和触摸屏采样延时图

ADCDAT1 寄存器的格式如表 14.5 所示。它与 ADCDAT0 寄存器格式相似，ADCDAT1 寄存器中保存 *y* 坐标值；而 ADCDAT0 寄存器中保存普通 A/D 转换的值或 *x* 坐标值。

表 14.5　　ADCDAT1 寄存器的格式

名　称	位	说　明
UPDOWN	[15]	对于触摸屏，使用“等待中断模式”时如下。 0：触摸屏被按下；1：触摸屏没有被按下
AUTO_PST	[14]	决定是否使用自动（连续）*x/y* 轴坐标转换模式。 0：正常转换；1：自动（连续）*x/y* 轴坐标转换

续表

名　称	位	说　明
XY_PST	[13:12]	手动 x/y 轴坐标转换模式。 00：无操作；　01：x 轴坐标转换 10：y 轴坐标转换；11：等待中断模式
Reserved	[11:10]	保留
YPDATA	[9:0]	x 轴坐标转换数据值

14.2 ADC和触摸屏操作实例

14.2.1 硬件设计

本开发板中，模拟输入引脚 AIN0、AIN1 外接可调电阻器，电路如图 14.9 所示。

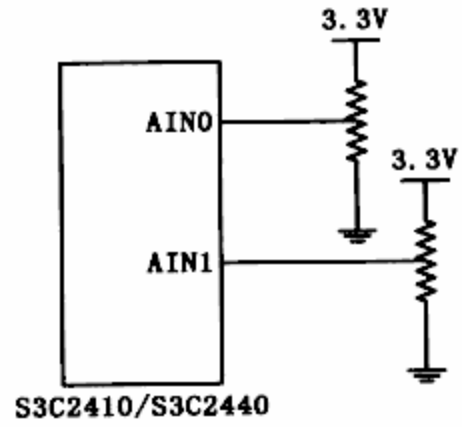


图 14.9　模拟通道 AIN0、AIN1 的外接电路

图 14.9 中的两个电阻是可调电阻器，程序将通过 AIN0、AIN1 这两个通道采集、转换电压值。

触摸屏的接口是标准的，它的电路图如图 14.4 所示。

14.2.2 程序设计

本实例程序将提供一个菜单，可以从中选择测试 ADC 或触摸屏。当测试 ADC 时，程序不断测量 AIN0、AIN1 的电压，并在串口上显示出来。当测试触摸屏时，只是测试触笔按下、松开的事件，并且把按下时的采集到的 x、y 坐标打印出来，它们只是原始的数据。

本实例的源码在/work/hardware/adc_ts 目录下，主要文件为 adc_ts.c。main.c 文件通过串口输出两个菜单供用户选择是测试 ADC 还是触摸屏，它们分别对应 adc_ts.c 中的 Test_Adc、Test_Ts 函数。

14.2.3 测试 ADC 的代码详解

1. ADC 主入口函数 Test_Adc

ADC 测试函数 Test_Adc 代码如下：

```
94 /*
95  * 测试 ADC
96  * 通过 A/D 转换，测量可变电阻器的电压值
97  */
98 void Test_Adc(void)
99 {
100     float vol0, vol1;
101     int t0, t1;
102
103      printf("Measuring the voltage of AIN0 and AIN1, press any key to
exit\n\r");
104     while (!awaitkey(0))    // 串口无输入，则不断测试
105     {
106         vol0 = ((float)ReadAdc(0)*3.3)/1024.0;  // 计算电压值
107         vol1 = ((float)ReadAdc(1)*3.3)/1024.0;  // 计算电压值
108         t0   = (vol0 - (int)vol0) * 1000;   // 计算小数部分，程序中的 printf
无法打印浮点数
109         t1   = (vol1 - (int)vol1) * 1000;   // 计算小数部分，程序中的 printf
无法打印浮点数
110         printf("AIN0 = %d.%-3dV    AIN1 = %d.%-3dV\r", (int)vol0, t0,
(int)vol1, t1);
111     }
112     printf("\n");
113 }
114
```

第 106、107 行先调用 ReadAdc 函数发起 A/D 转换，返回 10 位转换值（最大值为 1023）；然后计算实际的电压值（S3C2410/S3C2440 ADC 模拟信号最大电压为 3.3V）。

2. ReadAdc 函数：设置、启动 ADC，获取转换结果

ADC 操作核心的函数为 ReadAdc，代码如下：

```
68 /*
69  * 使用查询方式读取 A/D 转换值
70  * 输入参数:
71  *     ch: 模拟信号通道，取值为 0~7
72  */
73 static int ReadAdc(int ch)
74 {
75     // 选择模拟通道，使能预分频功能，设置 A/D 转换器的时钟 = PCLK/(49+1)
```

```
76      ADCCON = PRESCALE_EN | PRSCVL(49) | ADC_INPUT(ch);
77
78      // 清除位[2]，设为普通转换模式
79      ADCTSC &= ~(1<<2);
80
81      // 设置位[0]为1，启动A/D转换
82      ADCCON |= ADC_START;
83
84      // 当A/D转换真正开始时，位[0]会自动清0
85      while (ADCCON & ADC_START);
86
87      // 检测位[15]，当它为1时表示转换结束
88      while (!(ADCCON & ADC_ENDCVT));
89
90      // 读取数据
91      return (ADCDAT0 & 0x3ff);
92  }
93
```

程序流程与前面介绍的ADC的4个工作步骤一一对应。

（1）第76行选择模拟通道，使能预分频功能，设置A/D转换器的时钟。

本程序中，PCLK为50MHz，所以A/D转换器的时钟为50MHz/(49 + 1) = 1MHz，小于最大A/D时钟2.5MHz。

（2）第79行清除ADCTSC寄存器位[2]，设为普通转换模式，ADCTSC寄存器格式如表14.3所示。

（3）第82行设置ADCCON寄存器位[0]，启动A/D转换。

ADC的启动有两种方式，如果使用“读启动”方式（此时ADCCON寄存器位[1]被设为1），则读一下ADCDAT0寄存器即可启动；如果使用手动方式，设置ADCCON寄存器位[0]即可启动。

（4）第88行循环检测ADCCON寄存器的位[15]，直到它为1，这表示表示A/D转换结束。

也可以使用中断方式，当A/D转换结束时，ADC会发出INT_ADC中断信号。

（5）最后，第91行读取ADCDAT0即可得到转换的数据（低10位为有效数据）。

14.2.4 测试触摸屏的代码详解

触摸屏的操作稍微复杂，下面将程序流程图和触摸屏控制的状态转换图合并在一起（如图14.10所示），以便后面的代码分析。

1. 触摸屏的主入口函数Test_Ts

Test_Ts函数进行初始化、开启ADC中断之后，就不再参与触摸屏的操作，这都通过中

断服务程序来写成。代码如下：

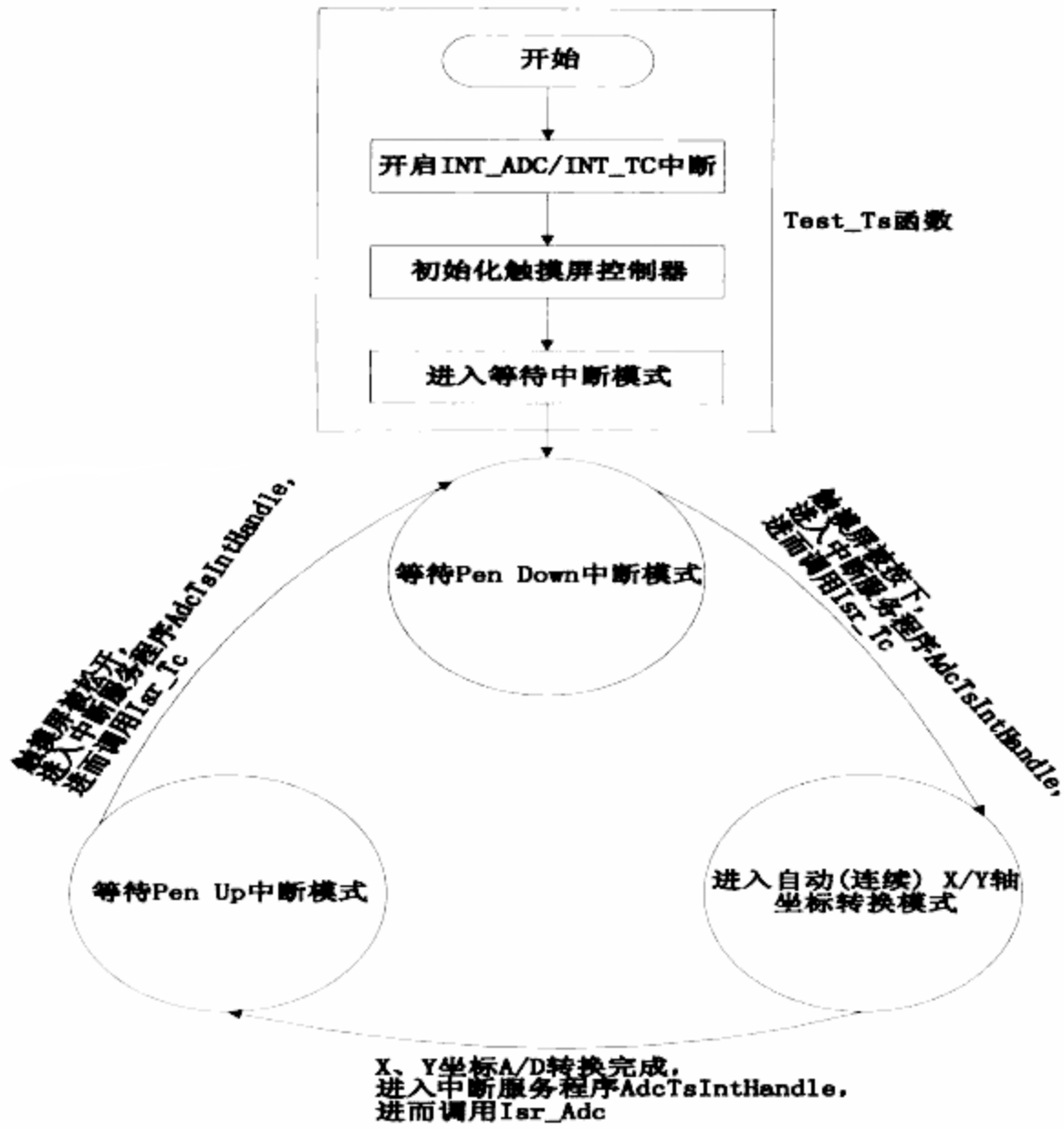


图 14.10 触摸屏测试程序流程图和触摸屏控制器状态转换图

```
/*
 * 测试触摸屏，打印触点坐标
 */
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);

    /* 采样延时时间 = (1/3.6864M)*50000 = 13.56ms
     * 即按下触摸屏后，再过 13.56ms 才能采样
```

```
202     */
203     ADCDLY = 50000;
204
205     wait_down_int();    /* 进入“等待中断模式”，等待触摸屏被按下 */
206
207     printf("Touch the screem to test, press any key to exit\n\r");
208     getc();
209
210     // 屏蔽ADC中断
211     INTSUBMSK |= BIT_SUB_TC;
212     INTSUBMSK |= BIT_SUB_ADC;
213     INTMSK |= BIT_ADC;
214 }
215
```

第192行设置ADC中断的处理函数，第193～195行开启ADC中断。ADC中断有两类：INT_TC和INT_ADC，前者表示触摸屏被按下（Pen Down）或松开（Pen Up），后者表示A/D转换结束。

第198行使能预分频功能，设置A/D转换器的时钟为PCLK/(49 + 1)，本程序中，PCLK为50MHz，所以A/D转换器的时钟为50MHz/(49 + 1) = 1MHz，小于最大A/D时钟2.5MHz。

第203行设置延时时间。

第205行调用wait_down_int()宏，令触摸屏控制器进入“等待中断模式”，等待触摸屏被按下。

wait_down_int()、wait_up_int()、mode_auto_xy()都是宏定义，它们用于设置触摸屏进入“等待Pen Down中断模式”、“等待Pen Up中断模式”、“自动（连续）*x/y*轴坐标转换模式”。读者可以参考14.1.3小节中的“S3C2410/S3C2440触摸屏接口”理解这些代码。

需要注意以下几点。

（1）对于S3C2410，ADCTSC的位[8]属于保留位，只能设为0；当处于“等待中断模式”时，无论是“Pend Down”中断还是“Pen Up”中断都可以检测到。

（2）对于S3C2440，ADCTSC的位[8]为0、1时分别表示等待Pen Down中断或Pen Up中断。

（3）要进入“自动（连续）*x/y*轴坐标转换模式”，XP、XM、YP、YM的状态不必理会，触摸屏在采样时会自动控制它们。

这些宏的定义如下：

```
/* 设置进入等待中断模式，XP_PU,XP_Dis,XM_Dis,YP_Dis,YM_En
 * (1)对于S3C2410，位[8]只能为0，所以只能使用下面的wait_down_int，
 *    它既等待Pen Down中断，也等待Pen Up中断
 * (2)对于S3C2440，位[8]为0、1时分别表示等待Pen Down中断或Pen Up中断
 */
```

```
54 /* 进入"等待中断模式"，等待触摸屏被按下 */
55 #define wait_down_int() { ADCTSC = DOWN_INT | XP_PULL_UP_EN | \
56                          XP_AIN | XM_HIZ | YP_AIN | YM_GND | \
57                          XP_PST(WAIT_INT_MODE); }
58 /* 进入"等待中断模式"，等待触摸屏被松开 */
59 #define wait_up_int()   { ADCTSC = UP_INT | XP_PULL_UP_EN | XP_AIN | XM_HIZ | \
60                          YP_AIN | YM_GND | XP_PST(WAIT_INT_MODE); }
61
62 /* 进入自动(连续) x/y 轴坐标转换模式 */
63 #define mode_auto_xy()  { ADCTSC = CONVERT_AUTO | XP_PULL_UP_DIS | XP_PST
(NOP_MODE); }
64
```

然后，程序就在 208 行等待串口的输入，以退出测试。等待期间，完全通过中断来驱动触摸屏的操作。

最后退出时，第 211～213 行屏蔽 ADC 中断。

2. 触摸屏中断处理函数：转换触摸屏的工作模式

从图 14.10 可知，执行 Test_Ts 函数之后，触摸屏控制器处于“等待 Pen Down 中断模式”。这时，如果按下触摸屏，则发生 INT_TC 中断，进入 AdcTsIntHandle 中断处理函数。它很简单，只是判断当前中断是 INT_TC 还是 INT_ADC，然后分别调用它们的中断服务程序，代码如下：

```
174 /*
175  * ADC、触摸屏的中断服务程序
176  * 对于 INT_TC、INT_ADC 中断，分别调用它们的处理程序
177  */
178 void AdcTsIntHandle(void)
179 {
180     if (SUBSRCPND & BIT_SUB_TC)
181         Isr_Tc();
182
183     if (SUBSRCPND & BIT_SUB_ADC)
184         Isr_Adc();
185 }
186
```

INT_TC 的中断服务程序 Isr_Tc 代码如下：

```
115 /*
116  * INT_TC 的中断服务程序
117  * 当触摸屏被按下时，进入自动(连续) x/y 轴坐标转换模式;
```

```
118 * 当触摸屏被松开时，进入等待中断模式，再次等待 INT_TC 中断
119 */
120 static void Isr_Tc(void)
121 {
122     if (ADCDAT0 & 0x8000)
123     {
124         printf("\nStylus Up!!\n");
125         wait_down_int();    /* 进入"等待中断模式"，等待触摸屏被按下 */
126     }
127     else
128     {
129         printf("\nStylus Down: ");
130
131         mode_auto_xy();     /* 进入自动(连续) x/y 轴坐标转换模式 */
132
133         /* 设置位[0]为 1，启动 A/D 转换
134          * 注意: ADCDLY 为 50000，PCLK = 50MHz
135          *       要经过(1/50MHz) ×50000=1ms 之后才开始转换 x 坐标
136          *       再经过 1ms 之后才开始转换 y 坐标
137          */
138         ADCCON |= ADC_START;
139     }
140
141     // 清除 INT_TC 中断
142     SUBSRCPND |= BIT_SUB_TC;
143     SRCPND    |= BIT_ADC;
144     INTPND    |= BIT_ADC;
145 }
146
```

第 122 行首先判断是“Pen Down”中断还是“Pen Up”中断，如果是“Pen Up”中断，表示触碰完成，在第 125 行中通过 wait_down_int()宏令触摸屏控制器进入“等待 Pen Down 中断模式”，等待下一次操作。

如果是“Pen Down”中断，则在第 131 行通过 mode_auto_xy()宏令触摸屏控制器进入“自动（连续）*x/y* 轴坐标转换模式”，然后在第 138 行启动 A/D 转换。也可以使用“分离的 *x/y* 轴坐标转换模式”手动地分别转换 *x* 坐标、*y* 坐标。

第 142～144 行清除 INT_TC 中断。

3. 在 ADC 中断处理函数中获取 *x*、*y* 坐标

在“自动（连续）*x/y* 轴坐标转换模式”下，*x*、*y* 坐标都转换完毕之后，产生 INT_ADC

中断，进入 AdcTsIntHandle 中断处理函数，它进而调用 INT_ADC 的中断服务程序 Isr_Adc，代码如下：

```
148 /*
149  * INT_ADC 的中断服务程序
150  * A/D 转换结束时发生此中断
151  * 先读取 x、y 坐标值，再进入等待中断模式
152  */
153 static void Isr_Adc(void)
154 {
155     // 打印 x、y 坐标值
156     printf("xdata = %4d, ydata = %4d\r\n", (int)(ADCDAT0 & 0x3ff), (int)
(ADCDAT1 & 0x3ff));
157
158     /* 判断是 S3C2410 还是 S3C2440 */
159     if ((GSTATUS1 == 0x32410000) || (GSTATUS1 == 0x32410002))
160     {   // S3C2410
161         wait_down_int();    /* 进入"等待中断模式"，等待触摸屏被松开 */
162     }
163     else
164     {   // S3C2440
165         wait_up_int();      /* 进入"等待中断模式"，等待触摸屏被松开 */
166     }
167
168     // 清除 INT_ADC 中断
169     SUBSRCPND |= BIT_SUB_ADC;
170     SRCPND    |= BIT_ADC;
171     INTPND    |= BIT_ADC;
172 }
173
```

首先，第 156 行从 ADCDAT0、ADCDAT1 寄存器中读出 x、y 坐标值，并打印出来。

然后，通过第 161 或 165 行（对于 S3C2410）或 158 行（对于 S3C2440）令触摸屏控制器进入“等待 Pen Up 中断模式”，等待触摸屏被松开。S3C2410 的触摸屏控制器既等待 Pen Down 中断，也等待 Pen Up 中断；S3C2440 的触摸屏控制器可以分开设置：等待被按下或（和）等待被松开。

最后，第 169～171 行清除 INT_ADC 中断。

14.2.5 实例测试

本程序在 main 函数中通过串口输出一个菜单，用于选择测试 ADC 或触摸屏。操作步骤

如下。

（1）使用串口将开发板的 COM0 和 PC 的串口相连，打开 PC 上的串口工具并设置其波特率为 115200、8N1。

（2）生成可执行程序，adc_ts_2410 或 adc_ts_2440，烧入 NAND Flash 后运行。

① 对于 S3C2410，在 ADC_TS 目录下执行“make CPU = S3C2410”命令，生成 adc_ts_2410。

② 对于 S3C2440，在 ADC_TS 目录下执行“make CPU = S3C2440”命令，生成 adc_ts_2440。

（3）在 PC 上串口工具上可以看到如下菜单：

```
##### Test ADC and Touch Screem #####
[A] Test ADC
[T] Test Touch Screem
Enter your selection:
```

（4）输入“A”以测试 ADC，可以看到如下字样：

```
Measuring the voltage of AIN0 and AIN1, press any key to exit
AIN0 = 1.102V    AIN1 = 1.108V
```

然后使用螺丝刀调节可变电阻器 ADJ0、ADJ1，可以在串口工具上看到它们的电压值不断变化。

最后按任意键退回到选择菜单。

（5）输入“T”以测试触摸屏，可以看到如下字样：

```
Touch the screem to test, press any key to exit
```

点击触摸屏可以在串口工具上看到触点坐标，可以看到类似如下字样：

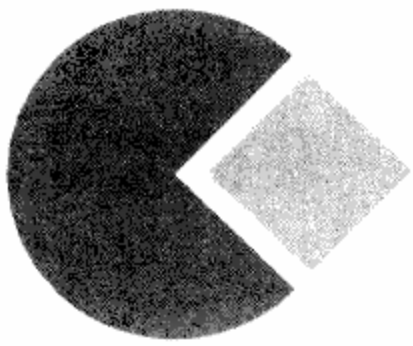
```
Stylus Down: xdata =  498, ydata =  516
```

松开触摸屏时也可以在串口工具中看到如下提示：

```
Stylus Up!!
```

最后按任意键退回到选择菜单。

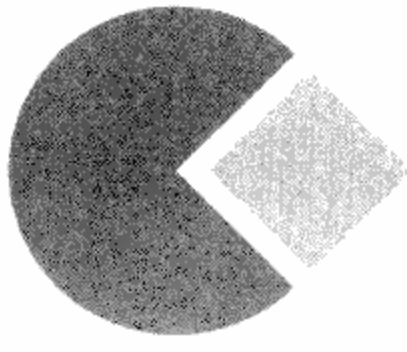
注意 触摸屏的实际使用中还要考虑初始校正、去除抖动、拖曳等功能。



第3篇 嵌入式Linux系统移植篇

- 移植U-Boot
- 移植Linux内核
- 构建Linux根文件系统
- Linux内核调试技术

……



第 15 章　移植 U-Boot

本章目标

- 了解 Bootloader 的作用及工作流程
- 了解 U-Boot 的代码结构、编译过程
- 移植 U-Boot
- 掌握常用的 U-Boot 命令

15.1　Bootloader 简介

15.1.1　Bootloader 的概念

1．Bootloader 的引入

从前面的硬件实例可以知道，系统上电之后，需要一段程序来进行初始化：关闭 WATCHDOG、改变系统时钟、初始化存储控制器、将更多的代码复制到内存中等。如果它能将操作系统内核复制到内存中运行，无论从本地（比如 Flash）还是从远端（比如通过网络），就称这段程序为 Bootloader。

简单地说，Bootloader 就是这么一小段程序，它在系统上电时开始执行，初始化硬件设备、准备好软件环境，最后调用操作系统内核。

可以增强 Bootloader 的功能，比如增加网络功能、从 PC 上通过串口或网络下载文件、烧写文件、将 Flash 上压缩的文件解压后再运行等，这就是一个功能更为强大的 Bootloader，也称为 Monitor。实际上，在最终产品中用户并不需要这些功能，它们只是为了方便开发。

Bootloader 的实现非常依赖于具体硬件，在嵌入式系统中硬件配置千差万别，即使是相同的 CPU，它的外设（比如 Flash）也可能不同，所以不可能有一个 Bootloader 支持所有的 CPU、所有的电路板。即使是支持 CPU 架构比较多的 U-Boot，也不是一拿来就可以使用的（除非里面的配置刚好与你的板子相同），需要进行一些移植。

2．Bootloader 的启动方式

CPU 上电后，会从某个地址开始执行。比如 MIPS 结构的 CPU 会从 0xBFC00000 取第一

条指令，而 ARM 结构的 CPU 则从地址 0x0000000 开始。嵌入式开发板中，需要把存储器件 ROM 或 Flash 等映射到这个地址，Bootloader 就存放在这个地址开始处，这样一上电就可以执行。

在开发时，通常需要使用各种命令操作 Bootloader，一般通过串口来连接 PC 和开发板，可以在串口上输入各种命令、观察运行结果等。这也只是对开发人员才有意义，用户使用产品时是不用接串口来控制 Bootloader 的。从这个观点来看，Bootloader 可以分为以下两种操作模式（Operation Mode）。

（1）启动加载（Boot loading）模式。

上电后，Bootloader 从板子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。产品发布时，Bootloader 工作在这种模式下。

（2）下载（Downloading）模式。

在这种模式下，开发人员可以使用各种命令，通过串口连接或网络连接等通信手段从主机（Host）下载文件（比如内核映象、文件系统映象），将它们直接放在内存运行或是烧入 Flash 类固态存储设备中。

板子与主机间传输文件时，可以使用串口的 xmodem/ymodem/zmodem 协议，它们使用简单，只是速度比较慢；还可以使用网络通过 tftp、nfs 协议来传输，这时，主机上要开启 tftp、nfs 服务；还有其他方法，比如 USB 等。

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，但是它会延时若干秒（这可以设置），等待终端用户按下任意键，而将 U-Boot 切换到下载模式。如果在指定时间内没有用户按键，则 U-Boot 继续启动 Linux 内核。

15.1.2 Bootloader 的结构和启动过程

1. 概述

在移植之前先了解 Bootloader 的一些通用概念，对理解它的代码会有所帮助。

嵌入式 Linux 系统从软件的角度通常可以分为以下 4 个层次。

（1）引导加载程序，包括固化在固件（firmware）中的 boot 代码（可选）和 Bootloader 两大部分。

有些 CPU 在运行 Bootloader 之前先运行一段固化的程序（固件，firmware），比如 x86 结构的 CPU 就是先运行 BIOS 中的固件，然后才运行硬盘第一个分区（MBR）中的 Bootloader。在大多嵌入式系统中并没有固件，Bootloader 是上电后执行的第一个程序。

（2）Linux 内核。

特定于嵌入式板子的定制内核以及内核的启动参数。内核的启动参数可以是内核默认的，或是由 Bootloader 传递给它的。

（3）文件系统。

包括根文件系统和建立于 Flash 内存设备之上的文件系统。里面包含了 Linux 系统能够运行所必需的应用程序、库等，比如可以给用户提供操作 Linux 的控制界面的 shell 程序、动态连接的程序运行时需要的 glibc 或 uClibc 库等。

（4）用户应用程序。

特定于用户的应用程序，它们也存储在文件系统中。有时在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。常用的嵌入式 GUI 有：Qtopia 和 MiniGUI 等。

显然，在嵌入系统的固态存储设备上有相应的分区来存储它们，如图 15.1 所示为一个典型的分区结构。

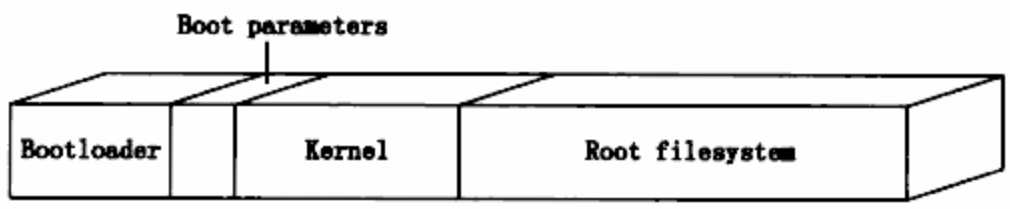


图 15.1 嵌入式 Linux 系统中的典型分区结构

“Boot parameters”分区中存放一些可设置的参数，比如 IP 地址、串口波特率、要传递给内核的命令行参数等。正常启动过程中，Bootloader 首先运行，然后它将内核复制到内存中（也有些内核可以在固态存储设备上直接运行），并且在内存某个固定的地址设置好要传递给内核的参数，最后运行内核。内核启动之后，它会挂接（mount）根文件系统（“Root filesystem”），启动文件系统中的应用程序。

2. Bootloader 的两个阶段

Bootloader 的启动过程可以分为单阶段（Single Stage）、多阶段（Multi-Stage）两种。通常多阶段的 Bootloader 能提供更为复杂的功能以及更好的可移植性。从固态存储设备上启动的 Bootloader 大多都是两阶段的启动过程。第一阶段使用汇编来实现，它完成一些依赖于 CPU 体系结构的初始化，并调用第二阶段的代码；第二阶段则通常使用 C 语言来实现，这样可以实现更复杂的功能，而且代码会有更好的可读性和可移植性。

一般而言，这两个阶段完成的功能可以如下分类。

（1）Bootloader 第一阶段的功能。

- 硬件设备初始化。
- 为加载 Bootloader 的第二阶段代码准备 RAM 空间。
- 复制 Bootloader 的第二阶段代码到 RAM 空间中。
- 设置好栈。
- 跳转到第二阶段代码的 C 入口点。

在第一阶段进行的硬件初始化一般包括：关闭 WATCHDOG、关中断、设置 CPU 的速度和时钟频率、RAM 初始化等。这些并不都是必需的，比如 S3C2410/S3C2440 的开发板所使用的 U-Boot 中，就将 CPU 的速度和时钟频率的设置放在第二阶段。

甚至，将第二阶段的代码复制到 RAM 空间中也不是必需的，对于 NOR Flash 等存储设备，完全可以在上面直接执行代码，只不过相比在 RAM 中执行效率大为降低。

（2）Bootloader 第二阶段的功能。

- 初始化本阶段要使用到的硬件设备。
- 检测系统内存映射（memory map）。
- 将内核映象和根文件系统映象从 Flash 上读到 RAM 空间中。
- 为内核设置启动参数。

- 调用内核。

为了方便开发，至少要初始化一个串口以便程序员与 Bootloader 进行交互。

所谓检测内存映射，就是确定板上使用了多少内存、它们的地址空间是什么。由于嵌入式开发中 Bootloader 多是针对某类板子进行编写，所以可以根据板子的情况直接设置，不需要考虑可以适用于各类情况的复杂算法。

Flash 上的内核映象有可能是经过压缩的，在读到 RAM 之后，还需要进行解压。当然，对于有自解压功能的内核，不需要 Bootloader 来解压。

将根文件系统映象复制到 RAM 中，这不是必需的。这取决于是什么类型的根文件系统，以及内核访问它的方法。

将内核存放在适当的位置后，直接跳到它的入口点即可调用内核。调用内核之前，下列条件要满足。

（1）CPU 寄存器的设置。

- R0=0。
- R1=机器类型 ID；对于 ARM 结构的 CPU，其机器类型 ID 可以参见 linux/arch/arm/tools/mach-types。
- R2=启动参数标记列表在 RAM 中起始基地址。

（2）CPU 工作模式。

- 必须禁止中断（IRQs 和 FIQs）。
- CPU 必须为 SVC 模式。

（3）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一样来调用内核：

```
void (*theKernel)(int zero, int arch, u32 params_addr) = (void (*)(int, int,
u32))KERNEL_RAM_BASE;
…
theKernel(0, ARCH_NUMBER, (u32) kernel_params_start);
```

3. Bootloader 与内核的交互

Bootloader 与内核的交互是单向的，Bootloader 将各类参数传给内核。由于它们不能同时运行，传递办法只有一个：Bootloader 将参数放在某个约定的地方之后，再启动内核，内核启动后从这个地方获得参数。

除了约定好参数存放的地址外，还要规定参数的结构。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。标记，就是一种数据结构；标记列表，就是挨着存放的多个标记。标记列表以标记 ATAG_CORE 开始，以标记 ATAG_NONE 结束。

标记的数据结构为 tag，它由一个 tag_header 结构和一个联合（union）组成。tag_header 结构表示标记的类型及长度，比如是表示内存还是表示命令行参数等。对于不同类型的标记使用不同的联合（union），比如表示内存时使用 tag_mem32，表示命令行时使用 tag_cmdline。

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，如下所示：

```
struct tag_header {
        u32 size;
        u32 tag;
};

struct tag {
        struct tag_header hdr;
        union {
                struct tag_core        core;
                struct tag_mem32   mem;
                struct tag_videotext   videotext;
                struct tag_ramdisk ramdisk;
                struct tag_initrd  initrd;
                struct tag_serialnr    serialnr;
                struct tag_revision    revision;
                struct tag_videolfb    videolfb;
                struct tag_cmdline cmdline;

                /*
                 * Acorn specific
                 */
                struct tag_acorn   acorn;

                /*
                 * DC21285 specific
                 */
                struct tag_memclk  memclk;
        } u;
};
```

下面以设置内存标记、命令行标记为例说明参数的传递。

（1）设置标记 ATAG_CORE。

标记列表以标记 ATAG_CORE 开始，假设 Bootloader 与内核约定的参数存放地址为 0x30000100，则可以以如下代码设置标记 ATAG_CORE：

```
params = (struct tag *) 0x30000100;

params->hdr.tag = ATAG_CORE;
params->hdr.size = tag_size (tag_core);
```

```
params->u.core.flags = 0;
params->u.core.pagesize = 0;
params->u.core.rootdev = 0;

params = tag_next (params);
```

其中，tag_next 定义如下，它指向当前标记的末尾：

```
#define tag_next(t) ((struct tag *)((u32 *)(t) + (t)->hdr.size))
```

（2）设置内存标记。

假设开发板使用的内存起始地址为 0x30000000，大小为 0x4000000，则内存标记可以如下设置：

```
params->hdr.tag = ATAG_MEM;
params->hdr.size = tag_size (tag_mem32);

params->u.mem.start = 0x30000000;
params->u.mem.size  = 0x4000000;

params = tag_next (params);
```

（3）设置命令行标记。

命令行就是一个字符串，它被用来控制内核的一些行为。比如"root=/dev/mtdblock 2 init=/linuxrc console=ttySAC0"表示根文件系统在 MTD2 分区上，系统启动后执行的第一个程序为/linuxrc，控制台为 ttySAC0（即第一个串口）。

命令行可以在 Bootloader 中通过命令设置好，然后按如下构造标记传给内核。

```
char *p = "root=/dev/mtdblock 2 init=/linuxrc console=ttySAC0";
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);
```

（4）设置标记 ATAG_NONE。

标记列表以标记 ATAG_NONE 结束，如下设置：

```
params->hdr.tag = ATAG_NONE;
params->hdr.size = 0;
```

15.1.3 常用 Bootloader 介绍

现在 Bootloader 种类繁多，比如 x86 上有 LILO、GRUB 等。对于 ARM 架构的 CPU，有 U-Boot、Vivi 等。它们各有特点，下面列出 Linux 的开放源代码的 Bootloader 及其支持的体系架构，如表 15.1 所示。

表 15.1 **开放源码的 Linux 引导程序**

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 BUIS 的 Linux 引导程序	是	否	否
BLOB	是	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
Vivi	是	Mizi 公司针对 SAMSUNG 的 ARM CPU 设计的引导程序	否	是	否

对于本书使用的 S3C2410/S3C2440 开发板，U-Boot 和 Vivi 是两个好选择。

Vivi 是 Mizi 公司针对 SAMSUNG 的 ARM 架构 CPU 专门设计的，基本上可以直接使用，命令简单方便。不过其初始版本只支持串口下载，速度较慢。在网上出现了各种改进版本：支持网络功能、USB 功能、烧写 YAFFS 文件系统映象等。

U-Boot 则支持大多 CPU，可以烧写 EXT2、JFFS2 文件系统映象，支持串口下载、网络下载，并提供了大量的命令。相对于 Vivi，它的使用更复杂，但是可以用来更方便地调试程序。

15.2 U-Boot 分析与移植

15.2.1 U-Boot 工程简介

U-Boot，全称为 Universal Boot Loader，即通用 Bootloader，是遵循 GPL 条款的开放源代码项目。其前身是由德国 DENX 软件工程中心的 Wolfgang Denk 基于 8xxROM 的源码创建的 PPCBOOT 工程。后来整理代码结构使得非常容易增加其他类型的开发板、其他架构的 CPU（原来只支持 PowerPC）；增加更多的功能，比如启动 Linux、下载 S-Record 格式的文件、通过网络启动、通过 PCMCIA/CompactFLash/ATA disk/SCSI 等方式启动。增加 ARM 架构 CPU 及其他更多 CPU 的支持后，改名为 U-Boot。

它的名字“通用”有两层含义：可以引导多种操作系统、支持多种架构的 CPU。它支持如下操作系统：Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS 等，支持如

下架构的 CPU：PowerPC、MIPS、x86、ARM、NIOS、XScale 等。

U-Boot 有如下特性。

- 开放源码。
- 支持多种嵌入式操作系统内核，如 Linux、NetBSD、VxWorks、QNX、RTEMS、ARTOS、LynxOS。
- 支持多个处理器系列，如 PowerPC、ARM、x86、MIPS、XScale。
- 较高的可靠性和稳定性。
- 高度灵活的功能设置，适合 U-Boot 调试、操作系统不同引导要求、产品发布等。
- 丰富的设备驱动源码，如串口、以太网、SDRAM、Flash、LCD、NVRAM、EEPROM、RTC、键盘等。
- 较为丰富的开发调试文档与强大的网络技术支持。
- 支持 NFS 挂载、RAMDISK（压缩或非压缩）形式的根文件系统。
- 支持 NFS 挂载、从 Flash 中引导压缩或非压缩系统内核。
- 可灵活设置、传递多个关键参数给操作系统，适合系统在不同开发阶段的调试要求与产品发布，尤对 Linux 支持最为强劲。
- 支持目标板环境变量多种存储方式，如 Flash、NVRAM、EEPROM。
- CRC32 校验，可校验 Flash 中内核、RAMDISK 镜像文件是否完好。
- 上电自检功能：SDRAM、Flash 大小自动检测，SDRAM 故障检测，CPU 型号。
- 特殊功能：XIP 内核引导。

可以从 http://sourceforge.net/projects/U-Boot 获得 U-Boot 的最新版本，如果使用过程中碰到问题或是发现 Bug，可以通过邮件列表网站 http://lists.sourcef orge.net/ lists/ listinfo/U-Boot-users/获得帮助。

15.2.2 U-Boot 源码结构

本书在 U-Boot-1.1.6 的基础上进行分析和移植，从 sourceforge 网站下载 U-Boot-1.1.6.tar.bz2 后解压即得到全部源码，U-Boot 源码目录结构比较简单。

U-Boot-1.1.6 根目录下共有 26 个子目录，可以分为 4 类。

（1）平台相关的或开发板相关的。

（2）通用的函数。

（3）通用的设备驱动程序。

（4）U-Boot 工具、示例程序、文档。

这 26 个子目录的功能与作用如表 15.2 所示。

表 15.2　　U-Boot 顶层目录说明

目　录	特　性	解 释 说 明
board	开发板相关	对应不同配置的电路板（即使 CPU 相同），比如 smdk2410、sbc2410x
cpu	平台相关	对应不同的 CPU，比如 arm920t、arm925t、i386 等；在它们的子目录下仍可以进一步细分，比如 arm920t 下就有 at91rm9200、s3c24x0
lib_i386 类似		某一架构下通用的文件

续表

目 录	特 性	解释说明
include	通用的函数	头文件和开发板配置文件，开发板的配置文件都放在 include/configs 目录下，U-Boot 没有 make menuconfig 类似的菜单来进行可视化配置，需要手动地修改配置文件中的宏定义
lib_generic		通用的库函数，比如 printf 等
common		通用的函数，多是对下一层驱动程序的进一步封装
disk	通用的设备驱动程序	硬盘接口程序
drivers		各类具体设备的驱动程序，基本上可以通用，它们通过宏从外面引入平台/开发板相关的函数
dtt		数字温度测量器或者传感器的驱动
fs		文件系统
nand_spl		U-Boot 一般从 ROM、NOR Flash 等设备启动，现在开始支持从 NAND Flash 启动，但是支持的 CPU 种类还不多
net		各种网络协议
post		上电自检程序
rtc		实时时钟的驱动
doc	文档	开发、使用文档
examples	示例程序	一些测试程序，可以使用 U-Boot 下载后运行
tools	工具	制作 S-Record、U-Boot 格式映象的工具，比如 mkimage

U-Boot 中各目录间也是有层次结构的，虽然这种分法不是绝对的，但是在移植过程中可以提供一些指导意义，如图 15.2 所示。

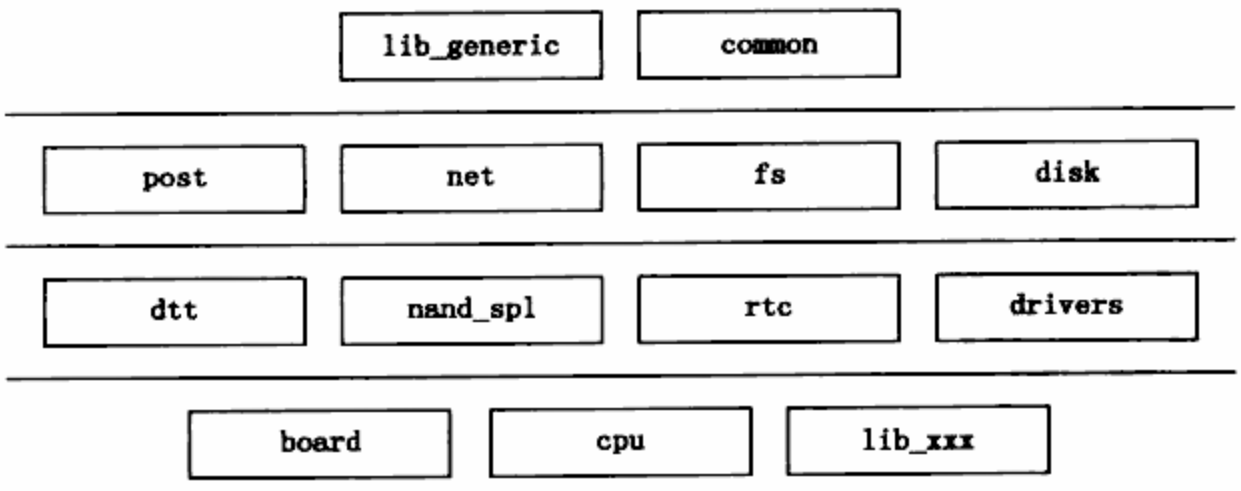


图 15.2 U-Boot 顶层目录的层次结构

比如 common/cmd_nand.c 文件提供了操作 NAND Flash 的各种命令，这些命令通过调用 drivers/nand/nand_base.c 中的擦除、读写函数来实现。这些函数针对 NAND Flash 的共性作了一些封装，将平台/开发板相关的代码用宏或外部函数来代替。而这些宏与外部函数，如果与平台相关，就要在下一层次的 cpu/xxx(xxx 表示某型号的 CPU)中实现；如果与开发板相关，就要在下一层次的 board/xxx 目录（xxx 表示某款开发板）中实现。本书移植的 U-Boot，就

是在 cpu/arm920t/s3c24x0 目录下增加了一个 nand_flash.c 文件来实现这些函数。

以增加烧写 yaffs 文件系统映象的功能为例，即在 common 目录下的 cmd_nand.c 中增加命令。比如 nand write.yaffs，这个命令要调用 drivers/nand/nand_util.c 中的相应函数，针对 yaffs 文件系统的特点依次调用擦除、烧写函数。而这些函数依赖于 drivers/nand/nand_base.c、cpu/arm920t/s3c24x0/nand_flash.c 文件中的相关函数。

目前 U-Boot-1.1.6 支持 10 种架构，根目录下有 10 个类似 lib_i386 的目录；31 个型号（类型）的 CPU，cpu 目录下有 31 个子目录；214 种开发板，board 目录下有 214 个子目录，很容易从中找到与自己的板子相似的配置，在上面稍作修改即可使用。

15.2.3 U-Boot 的配置、编译、连接过程

1. U-Boot 初体验

U-Boot-1.1.6 中有几千个文件，要想了解对于某款开发板，使用哪些文件、哪个文件首先执行、可执行文件占用内存的情况，最好的方法就是阅读它的 Makefile。

根据顶层 Readme 文件的说明，可以知道如果要使用开发板 board/<board_name>，就先执行“make <board_name>_config”命令进行配置，然后执行“make all”，就可以生成如下 3 个文件。

- U-Boot.bin：二进制可执行文件，它就是可以直接烧入 ROM、NOR Flash 的文件。
- U-Boot：ELF 格式的可执行文件。
- U-Boot.srec：Motorola S-Record 格式的可执行文件。

对于 S3C2410 的开发板，执行“make smdk2410_config”、“make all”后生成的 U-Boot.bin 可以烧入 NOR Flash 中运行。启动后可以看到串口输出一些信息后进入控制界面，等待用户的输入。

对于 S3C2440 的开发板，烧入上面生成的 U-Boot.bin，串口无输出，需要修改代码。

在修改代码之前，先看看上面两个命令“make smdk2410_config”、“make all”做了什么事情，以了解程序的流程，知道要修改哪些文件。

另外，编译 U-Boot 成功后，还会在它的 tools 子目录下生成一些工具，比如 mkimage 等。将它们复制到/usr/local/bin 目录下，以后就可以直接使用它们了，比如编译内核时，会使用 mkimage 来生成 U-Boot 格式的内核映象文件 uImage。

2. U-Boot 的配置过程

在顶层 Makefile 中可以看到如下代码：

```
SRCTREE      := $(CURDIR)
……
MKCONFIG     := $(SRCTREE)/mkconfig
……
smdk2410_config :   unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

假定在 U-Boot-1.1.6 的根目录下编译，则其中的 MKCONFIG 就是根目录下的 mkconfig

文件。$(@:_config=)的结果就是将“smdk2410_config”中的“_config”去掉，结果为“smdk2410”。所以“make smdk2410_config”实际上就是执行如下命令：

```
./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0
```

再来看看mkconfig的作用，在mkconfig文件开头第6行给出了它的用法：

```
06 # Parameters: Target Architecture CPU Board [VENDOR] [SOC]
```

这里解释一下概念，对于S3C2410、S3C2440，它们被称为SoC(System on Chip)，上面除CPU外，还集成了包括UART、USB控制器、NAND Flash控制器等设备（称为片内外设）。S3C2410/S3C2440中的CPU为ARM920T。

下面分步骤分析mkconfig的作用。

（1）确定开发板名称BOARD_NAME，相关代码如下：

```
11 APPEND=no    # Default: Create new config file
12 BOARD_NAME=""    # Name to print in make output
13
14 while [ $# -gt 0 ] ; do
15  case "$1" in
16  --) shift ; break ;;
17  -a) shift ; APPEND=yes ;;
18  -n) shift ; BOARD_NAME="${1%%_config}" ; shift ;;
19  *) break ;;
20  esac
21 done
22
23 [ "${BOARD_NAME}" ] || BOARD_NAME="$1"
```

对于“./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0”命令，其中没有“--”、“-a”、“-n”等符号，所以第14～22行没做任何事情。第11、12行两个变量仍维持原来的值。

执行完第23行后，BOARD_NAME的值等于第1个参数，即“smdk2410”。

（2）创建到平台/开发板相关的头文件的链接。

略过mkconfig文件中的一些没有起作用的行，如下所示：

```
30 #
31 # Create link to architecture specific headers
32 #
33 if [ "$SRCTREE" != "$OBJTREE" ] ; then
……
45 else
46      cd ./include
47      rm -f asm
```

```
48      ln -s asm-$2 asm
49 fi
50
```

第 33 行判断源代码目录和目标文件目录是否一样，可以选择在其他目录下编译 U-Boot，这可以令源代码目录保持干净，可以同时使用不同的配置进行编译。不过本书是直接在源代码目录下编译的，第 33 行的条件不满足，将执行 else 分支的代码。

第 46～48 行进入 include 目录，删除 asm 文件（这是上一次配置时建立的链接文件），然后再次建立 asm 文件，并令它链接向 asm-$2 目录，即 asm-arm。

继续往下看代码：

```
51 rm -f asm-$2/arch
52
53 if [ -z "$6" -o "$6" = "NULL" ] ; then
54      ln -s ${LNPREFIX}arch-$3 asm-$2/arch
55 else
56      ln -s ${LNPREFIX}arch-$6 asm-$2/arch
57 fi
58
59 if [ "$2" = "arm" ] ; then
60      rm -f asm-$2/proc
61      ln -s ${LNPREFIX}proc-armv asm-$2/proc
62 fi
63
```

第 51 行删除 asm-$2/arch 目录，即 asm-arm/arch。

对于“./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0”命令，$6 为“s3c24x0”，不为空，也不是“NULL”，所以第 53 行的条件不满足，将执行 else 分支。

第 56 行中，LNPREFIX 为空，所以这个命令实际上就是“ln -s arch-$6 asm-$2/arch”，即“ln -s arch-s3c24x0 asm-arm/arch”。

第 60、61 行重新建立 asm-arm/proc 文件，并让它链接向 proc-armv 目录。

（3）创建顶层 Makefile 包含的文件 include/config.mk，如下所示：

```
64 #
65 # Create include file for Make
66 #
67 echo "ARCH   = $2" >  config.mk
68 echo "CPU    = $3" >> config.mk
69 echo "BOARD  = $4" >> config.mk
70
71 [  "$5"  ] && [ "$5" != "NULL" ] && echo "VENDOR = $5" >> config.mk
```

```
72
73 [ "$6" ] && [ "$6" != "NULL" ] && echo "SOC    = $6" >> config.mk
74
```

对于“./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0”命令，上面几行代码创建的 config.mk 文件内容如下：

```
ARCH   = arm
CPU    = arm920t
BOARD  = smdk2410
SOC    = s3c24x0
```

（4）创建开发板相关的头文件 include/config.h，如下所示：

```
75 #
76 # Create board specific header file
77 #
78 if [ "$APPEND" = "yes" ]          # Append to existing config file
79 then
80       echo >> config.h
81 else
82       > config.h                  # Create new config file
83 fi
84 echo "/* Automatically generated - do not edit */" >>config.h
85 echo "#include <configs/$1.h>" >>config.h
86
```

APPEND 维持原值“no”，所以 config.h 被重新建立，它的内容如下：

```
/* Automatically generated - do not edit */
#include <configs/smdk2410.h>
```

现在总结一下，配置命令“make smdk2410_config”，实际的作用就是执行“./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0”命令。假设执行“./mkconfig $1 $2 $3 $4 $5 $6”命令，则将产生如下结果。

（1）开发板名称 BOARD_NAME 等于$1。

（2）创建到平台/开发板相关的头文件的链接，如下所示：

```
ln -s asm-$2 asm
ln -s arch-$6 asm-$2/arch
ln -s proc-armv asm-$2/proc        # 如果$2 不是 arm 的话，此行没有
```

（3）创建顶层 Makefile 包含的文件 include/config.mk，如下所示：

```
ARCH   = $2
CPU    = $3
```

```
BOARD  = $4
VENDOR = $5      # $5 为空，或者是 NULL 的话，此行没有
SOC    = $6      # $6 为空，或者是 NULL 的话，此行没有
```

（4）创建开发板相关的头文件 include/config.h，如下所示：

```
/* Automatically generated - do not edit */
#include <configs/$1.h>"
```

从这 4 个结果可以知道，如果要在 board 目录下新建一个开发板<board_name>的目录，则在 include/config 目录下也要建立一个文件<board_name>.h，里面存放的就是开发板<board_name>的配置信息。

U-Boot 还没有类似 Linux 一样的可视化配置界面（比如使用 make menuconfig 来配置），要手动修改配置文件 include/config/<board_name>.h 来裁减、设置 U-Boot。

配置文件中有以下两类宏。

（1）一类是选项（Options），前缀为“CONFIG_”，它们用于选择 CPU、SOC、开发板类型，设置系统时钟、选择设备驱动等。比如：

```
#define CONFIG_ARM920T        1          /* This is an ARM920T Core   */
#define CONFIG_S3C2410        1          /* in a SAMSUNG S3C2410 SoC    */
#define CONFIG_SMDK2410       1          /* on a SAMSUNG SMDK2410 Board  */
#define CONFIG_SYS_CLK_FREQ     12000000     /* the SMDK2410 has 12MHz input clock */
#define CONFIG_DRIVER_CS8900 1           /* we have a CS8900 on-board */
```

（2）另一类是参数（Setting），前缀为“CFG_”，它们用于设置 malloc 缓冲池的大小、U-Boot 的提示符、U-Boot 下载文件时的默认加载地址、Flash 的起始地址等。比如：

```
#define CFG_MALLOC_LEN       (CFG_ENV_SIZE + 128*1024)
#define CFG_PROMPT          "100ASK>"/* Monitor Command Prompt   */
#define CFG_LOAD_ADDR        0x33000000  /* default load address */
#define PHYS_FLASH_1        0x00000000   /* Flash Bank #1 */
```

从下面的编译、连接过程可知，U-Boot 中几乎每个文件都被编译和连接，但是这些文件是否包含有效的代码，则由宏开关来设置。比如对于网卡驱动 drivers/cs8900.c，它的格式为：

```
#include <common.h>          /* 将包含配置文件 include/config/<board_name>.h */
...
#ifdef CONFIG_DRIVER_CS8900
/* 实际的代码 */
...
#endif  /* CONFIG_DRIVER_CS8900 */
```

如果定义了宏 CONFIG_DRIVER_CS8900，则文件中包含有效的代码；否则，文件被注释为空。

可以这样认为，“CONFIG_”除了设置一些参数外，主要用来设置U-Boot的功能、选择使用文件中的哪一部分；而“CFG_”用来设置更细节的参数。

3. U-Boot的编译、连接过程

配置完后，执行“make all”即可编译，从Makefile中可以了解U-Boot使用了哪些文件、哪个文件首先执行、可执行文件占用内存的情况。

先确定用到哪些文件，下面所示为Makefile中与ARM相关的部分。

```
117 include $(OBJTREE)/include/config.mk
118 export      ARCH CPU BOARD VENDOR SOC
119
...
127 ifeq ($(ARCH),arm)
128 CROSS_COMPILE = arm-linux-
129 endif
...
163 # load other configuration
164 include $(TOPDIR)/config.mk
165
```

第117、164行用于包含其他的config.mk文件，第117行所要包含文件的就是在上面的配置过程中制作出来的include/config.mk文件，其中定义了ARCH、CPU、BOARD、SOC等4个变量的值为arm、arm920t、smdk2410、s3c24x0。

第164行包含顶层目录的config.mk文件，它根据上面4个变量的值确定了编译器、编译选项等。其中对我们理解编译过程有帮助的是BOARDDIR、LDFLAGS的值，如下所示：

```
88 BOARDDIR = $(BOARD)
...
91 sinclude $(TOPDIR)/board/$(BOARDDIR)/config.mk  # include board specific
rules
...
143 LDSCRIPT := $(TOPDIR)/board/$(BOARDDIR)/U-Boot.lds
...
189 LDFLAGS += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE) $(PLATFORM_LDFLAGS)
```

在board/smdk2410/config.mk中，定义了“TEXT_BASE = 0x33F80000”。所以，最终结果如下：BOARDDIR为smdk2410；LDFLAGS中有“-T board/smdk2410/U-Boot.lds -Ttext 0x33F80000”字样。

继续往下看Makefile：

```
166 ##########################################################################
167 # U-Boot objects....order is important (i.e. start must be first)
```

```
168
169 OBJS  = cpu/$(CPU)/start.o
...
193 LIBS  = lib_generic/libgeneric.a
194 LIBS += board/$(BOARDDIR)/lib$(BOARD).a
195 LIBS += cpu/$(CPU)/lib$(CPU).a
...
199 LIBS += lib_$(ARCH)/lib$(ARCH).a
200 LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/
libjffs2.a \
201 fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
202 LIBS += net/libnet.a
...
212 LIBS += $(BOARDLIBS)
213
……
```

从第 169 行得知，OBJS 的第一个值为“cpu/$(CPU)/start.o”，即“cpu/arm920t/ start.o”。第 193～213 行指定了 LIBS 变量就是平台/开发板相关的各个目录、通用目录下相应的库，比如：lib_generic/libgeneric.a、board/smdk2410/libsmdk2410.a、cpu/arm920t/ libarm920t.a、lib_arm/libarm.a、fs/cramfs/libcramfs.a fs/fat/libfat.a 等。

OBJS、LIBS 所代表的.o、.a 文件就是 U-Boot 的构成，它们通过如下命令由相应的源文件（或相应子目录下的文件）编译得到。

```
268 $(OBJS):
269             $(MAKE) -C cpu/$(CPU) $(if $(REMOTE_BUILD),$@,$(notdir $@))
270
271 $(LIBS):
272             $(MAKE) -C $(dir $(subst $(obj),,$@))
273
274 $(SUBDIRS):
275             $(MAKE) -C $@ all
276
```

第 268、269 两行的规则表示，对于 OBJS 中的每个成员，都将进入 cpu/$(CPU)目录（即 cpu/arm920t）编译它们。现在 OBJS 为 cpu/arm920t/start.o，它将由 cpu/arm920t/start.S 编译得到。

第 271、272 两行的规则表示，对于 LIBS 中的每个成员，都将进入相应的子目录执行“make”命令。这些子目录中的 Makefile，结构相似，它们将 Makefle 中指定的文件编译、连接成一个库文件。

当所有的 OBJS、LIBS 所表示的.o 和.a 文件都生成后，就剩最后的连接了，这对应 Makefile

中如下几行：

```
246 $(obj)U-Boot.srec:      $(obj)U-Boot
247                 $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
248
249 $(obj)U-Boot.bin:   $(obj)U-Boot
250                 $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
251
……
262 $(obj)U-Boot:       depend version $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)
263                 UNDEF_SYM=`$(OBJDUMP) -x $(LIBS) |sed -n -e 's/.*\(__u
_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\
264                 cd $(LNDIR) && $(LD) $(LDFLAGS) $$UNDEF_SYM $(__OBJS) \
265                   --start-group $(__LIBS) --end-group $(PLATFORM_LIBS) \
266                   -Map U-Boot.map -o U-Boot
267
```

先使用第 262～266 的规则连接得到 ELF 格式的 U-Boot，最后转换为二进制格式 U-Boot.bin、S-Record 格式 U-Boot.srec。LDFLAGS 确定了连接方式，其中的“-T board/smdk2410/U-Boot.lds -Ttext 0x33F80000”字样指定了程序的布局、地址。board/smdk2410/U-Boot.lds 文件如下：

```
28 SECTIONS
29 {
30       . = 0x00000000;
31
32       . = ALIGN(4);
33       .text      :
34       {
35         cpu/arm920t/start.o   (.text)
36         *(.text)
37       }
38
39       . = ALIGN(4);
40       .rodata : { *(.rodata) }
41
42       . = ALIGN(4);
43       .data : { *(.data) }
44
45       . = ALIGN(4);
46       .got : { *(.got) }
```

```
47
48      . = .;
49      __u_boot_cmd_start = .;
50      .u_boot_cmd : { *(.u_boot_cmd) }
51      __u_boot_cmd_end = .;
52
53      . = ALIGN(4);
54      __bss_start = .;
55      .bss : { *(.bss) }
56      _end = .;
57 }
```

从第 35 行可知，cpu/arm920t/start.o 被放在程序的最前面，所以 U-Boot 的入口点在 cpu/arm920t/start.S 中。

现在来总结一下 U-Boot 的编译流程。

（1）首先编译 cpu/$(CPU)/start.S，对于不同的 CPU，还可能编译 cpu/$(CPU)下的其他文件。

（2）然后，对于平台/开发板相关的每个目录、每个通用目录都使用它们各自的 Makefile 生成相应的库。

（3）将 1、2 步骤生成的.o、.a 文件按照 board/$(BOARDDIR)/config.mk 文件中指定的代码段起始地址、board/$(BOARDDIR)/U-Boot.lds 连接脚本进行连接。

（4）第 3 步得到的是 ELF 格式的 U-Boot，后面 Makefile 还会将它转换为二进制格式、S-Record 格式。

15.2.4 U-Boot 的启动过程源码分析

本书使用的 U-Boot 从 NOR Flash 启动，下面以开发板 smdk2410 的 U-Boot 为例。

U-Boot 属于两阶段的 Bootloader，第一阶段的文件为 cpu/arm920t/start.S 和 board/smdk2410/lowlevel_init.S，前者是平台相关的，后者是开发板相关的。

1. U-Boot 第一阶段代码分析

（1）硬件设备初始化。

依次完成如下设置：将 CPU 的工作模式设为管理模式（svc），关闭 WATCHDOG，设置 FCLK、HCLK、PCLK 的比例（即设置 CLKDIVN 寄存器），关闭 MMU、CACHE。

代码都在 cpu/arm920t/start.S 中，注释也比较完善，读者有不明白的地方可以参考前面硬件实验的相关章节。

（2）为加载 Bootloader 的第二阶段代码准备 RAM 空间。

所谓准备 RAM 空间，就是初始化内存芯片，使它可用。对于 S3C2410/S3C2440，通过在 start.S 中调用 lowlevel_init 函数来设置存储控制器，使得外接的 SDRAM 可用。代码在 board/smdk2410/lowlevel_init.S 中。

注意 lowlevel_init.S 文件是开发板相关的，这表示如果外接的设备不一样，可以修改 lowlevel_init.S 文件中的相关宏。

lowlevel_init 函数并不复杂，只是要注意这时的代码、数据都只保存在 NOR Flash 上，内存中还没有，所以读取数据时要变换地址。代码如下：

```
129 _TEXT_BASE:
130 .word   TEXT_BASE
131
132 .globl lowlevel_init
133 lowlevel_init:
134     /* memory control configuration */
135     /* make r0 relative the current location so that it */
136     /* reads SMRDATA out of FLASH rather than memory ! */
137     ldr     r0, =SMRDATA
138     ldr r1, _TEXT_BASE
139     sub r0, r0, r1
140     ldr r1, =BWSCON /* Bus Width Status Controller */
141     add     r2, r0, #13*4
142 0:
143     ldr     r3, [r0], #4
144     str     r3, [r1], #4
145     cmp     r2, r0
146     bne     0b
147
148     /* everything is fine now */
149     mov pc, lr
150
151     .ltorg
152 /* the literal pools origin */
153
154 SMRDATA:        /* 13个寄存器的值 */
155     .word ……
156     .word ……
```

第 137～139 行进行地址变换，因为这时候内存中还没有数据，不能使用连接程序时确定的地址来读取数据。

第 137 行中 SMRDATA 表示这 13 个寄存器的值存放的开始地址（连接地址），值为 0x33F8xxxx，处于内存中。

第 138 行获得代码段的起始地址，它就是第 130 行中的“TEXT_BASE”，其值在 board/smdk2410/config.mk 中定义为“TEXT_BASE = 0x33F80000”。

第 139 行将 0x33F8xxxx 与 0x33F80000 相减，这就是 13 个寄存器值在 NOR Flash 上存放的开始地址。

（3）复制 Bootloader 的第二阶段代码到 RAM 空间中。

这里将整个 U-Boot 的代码（包括第一、第二阶段）都复制到 SDRAM 中，这在 cpu/arm920t/start.S 中实现，如下所示：

```
164 relocate:                          /* 将 U-Boot 复制到 RAM 中 */
165     adr     r0, _start             /* r0:当前代码的开始地址 */
166     ldr     r1, _TEXT_BASE         /* r1:代码段的连接地址 */
167     cmp     r0, r1                 /* 测试现在是在 Flash 中还是在 RAM 中 */
168     beq     stack_setup            /* 如果已经在RAM中(这通常是调试时直接下载到RAM中),
                                       * 则不需要复制
                                       */
169
170     ldr     r2, _armboot_start/* _armboot_start 在前面定义，是第一条指令的运行地址 */
171     ldr     r3, _bss_start         /* 在连接脚本U-Boot.lds中定义,是代码段的结束地址 */
172     sub     r2, r3, r2             /* r2 = 代码段长度 */
173     add     r2, r0, r2             /* r2 = NOR Flash 上代码段的结束地址 */
174
175 copy_loop:
176     ldmia   r0!, {r3-r10}          /* 从地址[r0]处获得数据 */
177     stmia   r1!, {r3-r10}          /* 复制到地址[r1]处 */
178     cmp     r0, r2                 /* 判断是否复制完毕 */
179     ble     copy_loop              /* 没复制完，则继续 */
```

（4）设置好栈。

栈的设置灵活性很大，只要让 sp 寄存器指向一段没有使用的内存即可。

```
182     /* Set up the stack                                    */
183 stack_setup:
184     ldr r0, _TEXT_BASE             /* _TEXT_BASE 为代码段的开始地址，值为 0x33F80000 */
185     sub r0, r0, #CFG_MALLOC_LEN        /* 代码段下面，留出一段内存以实现malloc */
186     sub r0, r0, #CFG_GBL_DATA_SIZE  /* 再留出一段内存，存一些全局参数 */
187 #ifdef CONFIG_USE_IRQ
188     sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)          /*
IRQ、FIQ 模式的栈 */
189 #endif
190     sub sp, r0, #12                /* 最后，留出 12 字节的内存给 abort 异常，
                                       * 往下的内存就都是栈了
                                       */
191
```

到了这一步，读者可以知道内存的使用情况了，如图 15.3 所示（图中与上面的划分稍有不同，这是因为在 cpu/arm920t/cpu.c 中的 cpu_init 函数中才真正为 IRQ、FIQ 模式划分了栈）。

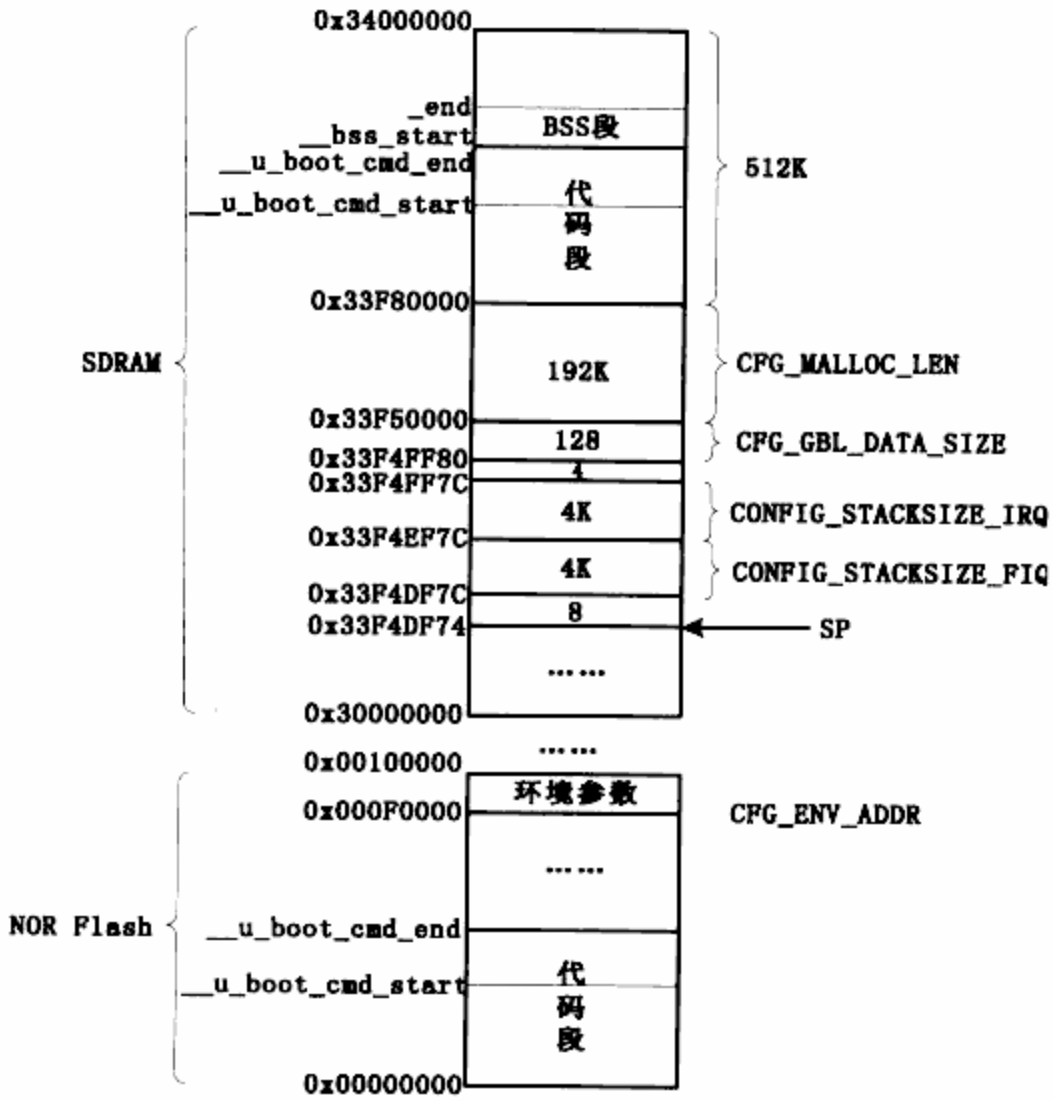


图 15.3 U-Boot 内存使用情况

（5）跳转到第二阶段代码的 C 入口点。

在跳转之前，还要清除 BSS 段（初始值为 0、无初始值的全局变量、静态变量放在 BSS 段），代码如下：

```
clear_bss:
    ldr     r0, _bss_start      /* BSS 段的开始地址，它的值在连接脚本 U-Boot.lds 中确定 */
    ldr     r1, _bss_end        /* BSS 段的结束地址，它的值在连接脚本 U-Boot.lds 中确定 */
    mov r2, #0x00000000

clbss_l:str r2, [r0]            /* 往 BSS 段中写入 0 值 */
    add     r0, r0, #4
    cmp     r0, r1
    ble     clbss_l

```

现在，C 函数的运行环境已经完全准备好，通过如下命令直接跳转（这之后，程序才在

内存中执行），它将调用 lib_arm/board.c 中的 start_armboot 函数，这是第二阶段的入口点。

```
223     ldr     pc, _start_armboot
224
225 _start_armboot: .word start_armboot
226
```

2. U-Boot 第二阶段代码分析

它与 15.1.2 节中描述的 Bootloader 第二阶段所完成的功能基本上一致，只是顺序有点小差别。另外，U-Boot 在启动内核之前可以让用户决定是否进入下载模式，即进入 U-Boot 的控制界面。

第二阶段从 lib_arm/board.c 中的 start_armboot 函数开始，程序流程如图 15.4 所示。

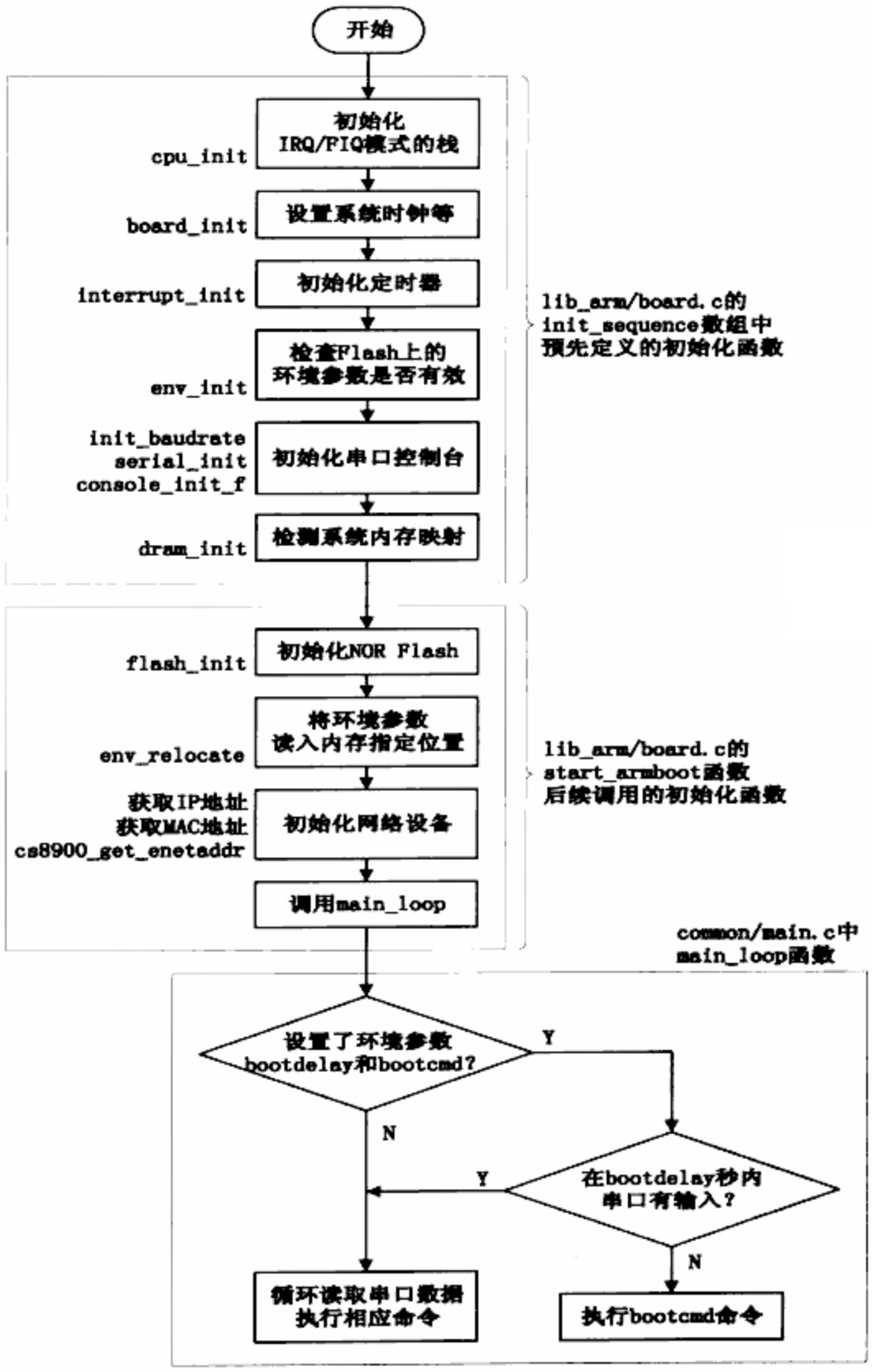


图 15.4 U-Boot 第二阶段流程图

移植 U-Boot 的主要工作在于对硬件的初始化、驱动，所以下面讲解时将重点放在硬件的操作上。

（1）初始化本阶段要使用到的硬件设备。

最主要的是设置系统时钟、初始化串口，只要这两个设置好了，就可以从串口看到打印信息。

board_init 函数设置 MPLL、改变系统时钟，它是开发板相关的函数，在 board/smdk2410/smdk2410.c 中实现。值得注意的是，board_init 函数中还保存了机器类型 ID，这将在调用内核时传给内核，代码如下：

```
        /* arch number of SMDK2410-Board */
        gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;    /* 值为 193 */
```

串口的初始化函数主要是 serial_init，它设置 UART 控制器，是 CPU 相关的函数，在 cpu/arm920t/s3c24x0/serial.c 中实现。

（2）检测系统内存映射（memory map）。

对于特定的开发板，其内存的分布是明确的，所以可以直接设置。board/smdk2410/smdk2410.c 中的 dram_init 函数指定了本开发板的内存起始地址为 0x30000000，大小为 0x4000000。代码如下：

```
int dram_init (void)
{
        gd->bd->bi_dram[0].start = PHYS_SDRAM_1;        /* 即 0x300000000 */
        gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;    /* 即 0x4000000 */

        return 0;
}
```

这些设置的参数，将在后面向内核传递参数时用到。

（3）U-Boot 命令的格式。

从图 15.3 可以知道，即使是内核的启动，也是通过 U-Boot 命令来实现的。U-Boot 中每个命令都通过 U_BOOT_CMD 宏来定义，格式如下：

```
U_BOOT_CMD(name,maxargs,repeatable,command,"usage","help")
```

各项参数的意义如下。

① name：命令的名字，注意，它不是一个字符串（不要用双引号括起来）。

② maxargs：最大的参数个数。

③ repeatable：命令是否可重复，可重复是指运行一个命令后，下次敲回车即可再次运行。

④ command：对应的函数指针，类型为(*cmd)(struct cmd_tbl_s *, int, int, char *[])。

⑤ usage：简短的使用说明，这是个字符串。

⑥ help：较详细的使用说明，这是个字符串。

宏 U_BOOT_CMD 在 include/command.h 中定义，如下所示：

```
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage,
help}
```

Struct_Section 也是在 include/command.h 中定义，如下所示：

```
#define Struct_Section  __attribute__ ((unused,section (".u_boot_cmd")))
```

比如对于 bootm 命令，它如下定义：

```
U_BOOT_CMD(
        bootm, CFG_MAXARGS,    1,  do_bootm,
         "string1",
          "string2"
);
```

宏 U_BOOT_CMD 扩展开后如下所示：

```
cmd_tbl_t __u_boot_cmd_bootm __attribute__ ((unused,section (".u_boot_cmd")))
= {"bootm", CFG_MAXARGS, 1, do_bootm, "string1", "string2"};
```

对于每个使用 U_BOOT_CMD 宏来定义的命令，其实都是在".u_boot_cmd"段中定义一个 cmd_tbl_t 结构。连接脚本 U-Boot.lds 中有如下代码：

```
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;
```

程序中就是根据命令的名字在内存段__u_boot_cmd_start～__u_boot_cmd_end 找到它的 cmd_tbl_t 结构，然后调用它的函数（请参考 common/command.c 中的 find_cmd 函数）。

内核的复制和启动，可以通过如下命令来完成：bootm 从内存、ROM、NOR Flash 中启动内核，bootp 则通过网络来启动，而 nboot 从 NAND Flash 启动内核。它们都是先将内核映象从各种媒介中读出，存放在指定的位置；然后设置标记列表以给内核传递参数；最后跳到内核的入口点去执行。具体实现的细节不再描述，有兴趣的读者可以阅读 common/cmd_boot.c、common/cmd_net.c、common/cmd_nand.c 来了解它们的实现。

（4）为内核设置启动参数。

U-Boot 也是通过标记列表向内核传递参数。并且，在 15.1.2 小节中内存标记、命令行标记的示例代码就是取自 U-Boot 中的 setup_memory_tags、setup_commandline_tag 函数，它们都是在 lib_arm/armlinux.c 中定义。一般而言，设置这两个标记就可以了，在配置文件 include/configs/smdk2410.h 中增加如下两个配置项即可：

```
#define CONFIG_SETUP_MEMORY_TAGS    1
#define CONFIG_CMDLINE_TAG          1
```

对于 ARM 架构的 CPU，都是通过 lib_arm/armlinux.c 中的 do_bootm_linux 函数来启动内

核。这个函数中，设置标记列表，最后通过“theKernel (0, bd→bi_arch_number, bd→bi_boot_params)”调用内核。其中，theKernel 指向内核存放的地址（对于 ARM 架构的 CPU，通常是 0x30008000），bd→bi_arch_number 就是前面 board_init 函数设置的机器类型 ID，而 bd→bi_boot_params 就是标记列表的开始地址。

15.2.5 U-Boot 的移植

开发板 smdk2410 的配置适用于大多数 S3C2410 开发板，或是只需要极少的修改即可使用。但是目前 U-Boot 中没有对 S3C2440 的支持，需要我们自己移植。

本书基于的 S3C2410、S3C2440 两款开发板，它们的外接硬件相同。

- BANK0 外接容量为 1MB，位宽为 8 的 NOR Flash 芯片 AM29LV800。
- BANK3 外接 10M 网卡芯片 CS8900，位宽为 16。
- BANK6 外接两片容量为 32MB、位宽为 16 的 SDRAM 芯片 K4S561632，组成容量为 64MB、位宽为 32 的内存。
- 通过 NAND Flash 控制器外接容量为 64MB，位宽为 8 的 NAND Flash 芯片 K9S1208。

对于 NOR Flash 和 NAND Flash，如图 15.5 所示划分它们的使用区域。由于 NAND Flash 的“位反转”现象比较常见，为保证数据的正确，在读写数据时需要使用 ECC 校验。另外，NAND Flash 在使用过程中、运输过程中还有可能出现坏块。所以本书选择在 NOR Flash 中保存 U-Boot，在 NAND Flash 中保存内核和文件系统，并在使用 U-Boot 烧写内核、文件系统时进行坏块检查、ECC 校验。这样，即使 NAND Flash 出现坏块导致内核或文件系统不能使用，也可以通过 NOR Flash 中的 U-Boot 来重新烧写。

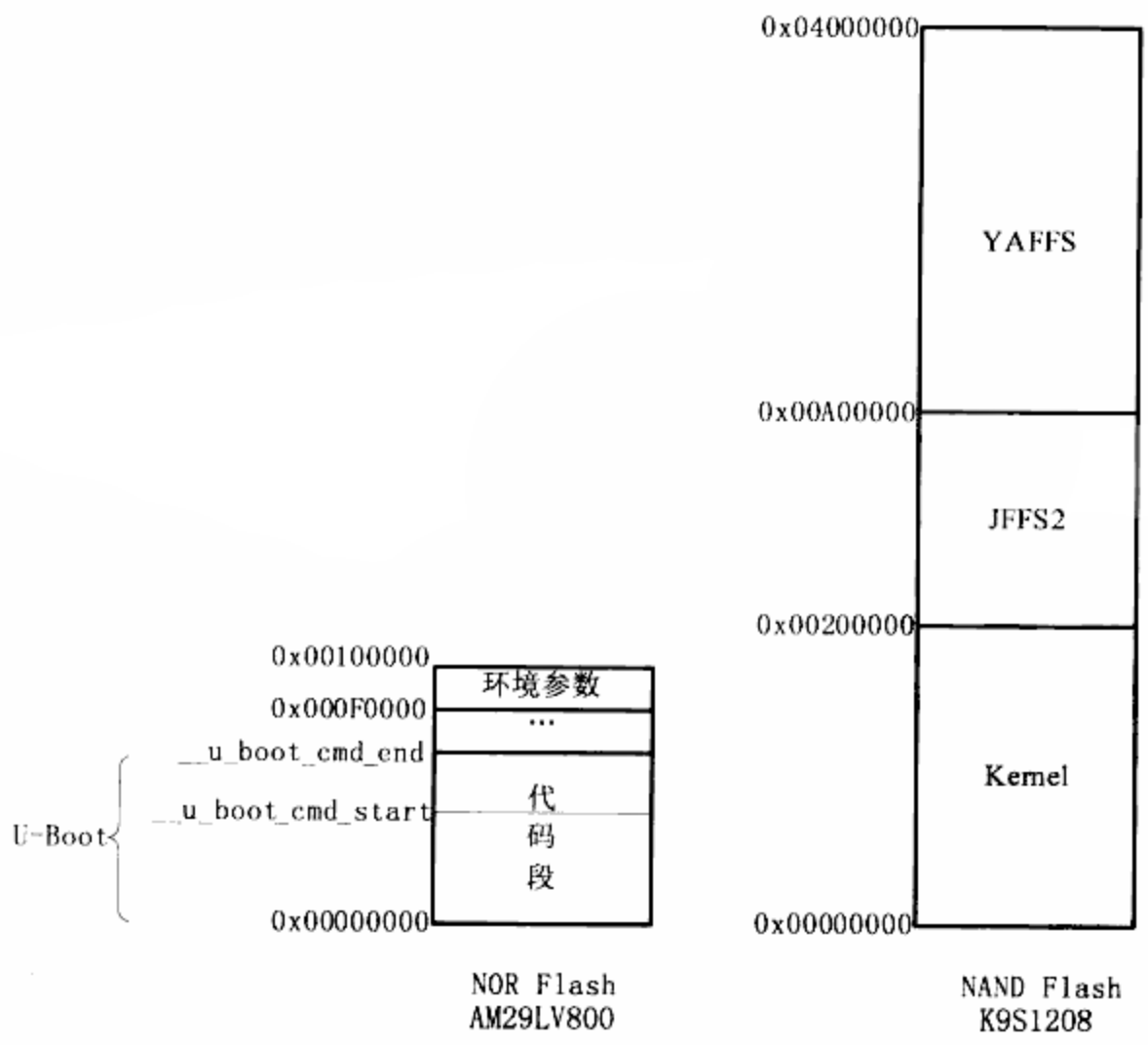


图 15.5 开发板固态存储器分区划分

smdk2410 开发板已经支持 NOR Flash 芯片 AM29LV800，U-Boot 本身也已经支持 JFFS2 文件系统映象的烧写。下面一步一步移植 U-Boot（所有的修改都在补丁文件 U-boot-1.1.6_100ask_24 X 0.Patch 里，读者可以直接打补丁），增加如下新功能。

- 同时支持本书使用的 S3C2410 和 S3C2440 开发板。
- 支持串口 xmodem 协议。
- 支持网卡芯片 CS8900。
- 支持 NAND Flash 读写。
- 支持烧写 yaffs 文件系统映象。

1. 同时支持 S3C2410 和 S3C2440

我们将在开发板 smdk2410 的基础上进行移植。

（1）新建一个开发板的相应目录和文件。

为了不破坏原来的代码，在 board 目录下将 smdk2410 复制为 100ask24x0 目录，并将 board/100ask24x0/smdk2410.c 改名为 100ask24x0.c。

根据前面描述的配置过程可知，还要在 include/configs 目录下建立一个配置文件 100ask24x0.h，可以将 include/configs/smdk2410.h 直接复制为 100ask24x0.h。

还要修改两个 Makefile，首先在顶层 Makefile 中增加如下两行：

```
100ask24x0_config   :                unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t 100ask24x0 NULL s3c24x0
```

然后在 board/100ask24x0/Makefile 中进行如下修改（因为前面将 smdk2410.c 文件改名为 100ask24x0.c 了）：

```
COBJS   := smdk2410.o flash.o
改为:
COBJS   := 100ask24x0.o flash.o
```

（2）修改 SDRAM 的配置。

SDRAM 的初始化在 U-Boot 的第一阶段完成，就是在 board/100ask24x0/lowlevel_init.S 文件中设置存储控制器。

检查一下 BANK6 的设置：位宽为 32，宏 B6_BWSCON 刚好为 DW32（表示 32 位），无需改变；另外还要根据 HCLK 设置 SDRAM 的刷新参数，主要是 REFCNT 寄存器。

本书所用开发板的 HCLK 都设为 100MHz，需要根据 SDRAM 芯片的具体参数重新计算 REFCNT 寄存器的值（请参考第 6 章）。代码修改如下：

```
126 #define REFCNT 1113    /* period=15.6μs, HCLK=60Mhz, (2048+1-15.6*60) */
改为
126 #define REFCNT 0x4f4   /*        period=7.8125μs,        HCLK=100Mhz,
(2048+1-7.8125*100) */
```

对于其他 BANK，比如网卡芯片 CS8900 所在的 BANK2，原来的设置刚好匹配，无需更改；而对于 BANK1、BANK2、BANK4、BANK5、BANK7，在 U-Boot 中并没有使用到它们

外接的设备，也不需要理会。

（3）增加对 S3C2440 的支持。

S3C2440 是 S3C2410 的改进版，它们的操作基本相似。不过在系统时钟的设置、NAND Flash 控制器的操作等方面有一些小差别。它们的 MPLL、UPLL 计算公式不一样，FCLK、HCLK 和 PCLK 的分频化设置也不一样，这在下面的代码中可以看到。NAND Flash 控制器的差别在增加对 NAND Flash 的支持时讲述。

本章的目标是令同一个 U-Boot 二进制代码既可以在 S3C2410 上运行，也可以在 S3C2440 上运行。首先需要在代码中自动识别是 S3C2410 还是 S3C2440，这可以通过读取 GSTATUS1 寄存器的值来分辨：0x32410000 表示 S3C2410，0x32410002 表示 S3C2410A，0x32440000 表示 S3C2440，0x32440001 表示 S3C2440A。S3C2410 和 S3C2410A、S3C2440 和 S3C2440A，对本书来说没有区别。

对于 S3C2410 开发板，将 FCLK 设为 200MHz，分频比为 FCLK:HCLK:PCLK=1:2:4；对于 S3C2440 开发板，将 FCLK 设为 400MHz，分频比为 FCLK:HCLK:PCLK=1:4:8。还将 UPLL 设为 48MHz，即 UCLK 为 48MHz，以在内核中支持 USB 控制器。

首先修改 board/100ask24x0/100ask24x0.c 中的 board_init 函数，下面是修改后的代码：

```
33 /* S3C2440: MPLL = (2*m * Fin) / (p * 2^s), UPLL = (m * Fin) / (p * 2^s)
34  * m = M (the value for divider M)+ 8, p = P (the value for divider P) + 2
35  */
36 #define S3C2440_MPLL_400MHZ        ((0x5c<<12)|(0x01<<4)|(0x01))
37 #define S3C2440_UPLL_48MHZ         ((0x38<<12)|(0x02<<4)|(0x02))
38 #define S3C2440_CLKDIV             0x05   /* FCLK:HCLK:PCLK = 1:4:8, UCLK = UPLL */
39
40 /* S3C2410: Mpll,Upll = (m * Fin) / (p * 2^s)
41  * m = M (the value for divider M)+ 8, p = P (the value for divider P) + 2
42  */
43 #define S3C2410_MPLL_200MHZ        ((0x5c<<12)|(0x04<<4)|(0x00))
44 #define S3C2410_UPLL_48MHZ         ((0x28<<12)|(0x01<<4)|(0x02))
45 #define S3C2410_CLKDIV             0x03    /* FCLK:HCLK:PCLK = 1:2:4 */
46
```

上面代码针对 S3C2410、S3C2440 分别定义了 MPLL、UPLL 寄存器的值。开发板输入时钟为 12MHz(这在 include/configs/100ask24x0.h 中的宏 CONFIG_SYS_CLK_FREQ 中定义)，读者可以根据代码中的计算公式针对自己的开发板修改系统时钟。

下面是针对 S3C2410、S3C2440 分别使用不同的宏设置系统时钟。

```
58 int board_init (void)
59 {
60     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
61     S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();
62
```

```
63      /* 设置 GPIO */
64      gpio->GPACON = 0x007FFFFF;
65      gpio->GPBCON = 0x00044555;
66      gpio->GPBUP = 0x000007FF;
67      gpio->GPCCON = 0xAAAAAAAA;
68      gpio->GPCUP = 0x0000FFFF;
69      gpio->GPDCON = 0xAAAAAAAA;
70      gpio->GPDUP = 0x0000FFFF;
71      gpio->GPECON = 0xAAAAAAAA;
72      gpio->GPEUP = 0x0000FFFF;
73      gpio->GPFCON = 0x000055AA;
74      gpio->GPFUP = 0x000000FF;
75      gpio->GPGCON = 0xFF95FFBA;
76      gpio->GPGUP = 0x0000FFFF;
77      gpio->GPHCON = 0x002AFAAA;
78      gpio->GPHUP = 0x000007FF;
79
80      /* 同时支持 S3C2410 和 S3C2440 */
81      if ((gpio->GSTATUS1 == 0x32410000) || (gpio->GSTATUS1 == 0x32410002))
82      {
83          /* FCLK:HCLK:PCLK = 1:2:4 */
84          clk_power->CLKDIVN = S3C2410_CLKDIV;
85
86          /* 修改为异步总线模式 */
87          __asm__(    "mrc    p15, 0, r1, c1, c0, 0\n"    /* read ctrl register   */
88                      "orr    r1, r1, #0xc0000000\n"      /* Asynchronous         */
89                      "mcr    p15, 0, r1, c1, c0, 0\n"    /* write ctrl register  */
90                      :::"r1"
91                      );
92
93          /* 设置 PLL 锁定时间 */
94          clk_power->LOCKTIME = 0xFFFFFF;
95
96          /* 配置 MPLL */
97          clk_power->MPLLCON = S3C2410_MPLL_200MHZ;
98
99          /* 配置 MPLL 后，要延时一段时间再配置 UPLL */
100         delay (4000);
101
```

```
102         /* 配置UPLL */
103         clk_power->UPLLCON = S3C2410_UPLL_48MHZ;
104
105         /* 再延时一会 */
106         delay (8000);
107
108         /* 机器类型ID，这在调用Linux内核时用到 */
109         gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
110     }
111     else
112     {
113         /* FCLK:HCLK:PCLK = 1:4:8 */
114         clk_power->CLKDIVN = S3C2440_CLKDIV;
115
116         /* 修改为异步总线模式 */
117         __asm__(    "mrc   p15, 0, r1, c1, c0, 0\n"    /* read ctrl register   */
118                     "orr   r1, r1, #0xc0000000\n"      /* Asynchronous         */
119                     "mcr   p15, 0, r1, c1, c0, 0\n"    /* write ctrl register  */
120                     :::"r1"
121                     );
122
123         /* 设置PLL锁定时间 */
124         clk_power->LOCKTIME = 0xFFFFFF;
125
126         /* 配置MPLL */
127         clk_power->MPLLCON = S3C2440_MPLL_400MHZ;
128
129         /* 配置MPLL后，要延时一段时间再配置UPLL */
130         delay (4000);
131
132         /* 配置UPLL */
133         clk_power->UPLLCON = S3C2440_UPLL_48MHZ;
134
135         /* 再延时一会 */
136         delay (8000);
137
138         /* 机器类型ID，这在调用Linux内核时用到，这个值要与内核相对应 */
139         gd->bd->bi_arch_number = MACH_TYPE_S3C2440;
140     }
```

```
141
142     /* 启动内核时，参数存放位置。这个值在构造标记列表时用到 */
143     gd->bd->bi_boot_params = 0x30000100;
144
145     icache_enable();
146     dcache_enable();
147
148     return 0;
149 }
150
```

最后一步：获取系统时钟的函数需要针对 S3C2410、S3C2440 的不同进行修改。

在后面设置串口波特率时需要获得系统时钟，就是在 U-Boot 的第二阶段，lib_arm/board.c 中 start_armboot 函数调用 serial_init 函数初始化串口时，会调用 get_PCLK 函数。它在 cpu/arm920t/s3c24x0/speed.c 中定义，与它相关的还有 get_HCLK、get_PLLCLK 等函数。

前面的 board_init 函数在识别出 S3C2410 或 S3C2440 后，设置了机器类型 ID：gd→bd→bi_arch_number，后面的函数可以通过它来分辨是 S3C2410 还是 S3C2440。首先要在程序的开头增加如下一行，这样才可以使用 gd 变量。

```
DECLARE_GLOBAL_DATA_PTR;
```

S3C2410 和 S3C2440 的 MPLL、UPLL 计算公式不一样，所以 get_PLLCLK 函数也需要修改，如下所示：

```
56 static ulong get_PLLCLK(int pllreg)
57 {
58     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
59     ulong r, m, p, s;
60
61     if (pllreg == MPLL)
62         r = clk_power->MPLLCON;
63     else if (pllreg == UPLL)
64         r = clk_power->UPLLCON;
65     else
66         hang();
67
68     m = ((r & 0xFF000) >> 12) + 8;
69     p = ((r & 0x003F0) >> 4) + 2;
70     s = r & 0x3;
71
72     /* 同时支持 S3C2410 和 S3C2440 */
73     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
```

```
74          return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
75      else
76          return((CONFIG_SYS_CLK_FREQ * m * 2) / (p << s));   /* S3C2440 */
77 }
78
```

由于分频系数的设置方法也不一样，get_HCLK、get_PCLK 也需要修改。对于 S3C2410，沿用原来的计算方法，else 分支中是 S3C2440 的代码，如下所示：

```
85 /* for s3c2440 */
86 #define S3C2440_CLKDIVN_PDIVN          (1<<0)
87 #define S3C2440_CLKDIVN_HDIVN_MASK     (3<<1)
88 #define S3C2440_CLKDIVN_HDIVN_1        (0<<1)
89 #define S3C2440_CLKDIVN_HDIVN_2        (1<<1)
90 #define S3C2440_CLKDIVN_HDIVN_4_8      (2<<1)
91 #define S3C2440_CLKDIVN_HDIVN_3_6      (3<<1)
92 #define S3C2440_CLKDIVN_UCLK           (1<<3)
93
94 #define S3C2440_CAMDIVN_CAMCLK_MASK  (0xf<<0)
95 #define S3C2440_CAMDIVN_CAMCLK_SEL   (1<<4)
96 #define S3C2440_CAMDIVN_HCLK3_HALF   (1<<8)
97 #define S3C2440_CAMDIVN_HCLK4_HALF   (1<<9)
98 #define S3C2440_CAMDIVN_DVSEN        (1<<12)
99
100 /* return HCLK frequency */
101 ulong get_HCLK(void)
102 {
103     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
104     unsigned long clkdiv;
105     unsigned long camdiv;
106     int hdiv = 1;
107
108     /* 同时支持S3C2410和S3C2440 */
109     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
110         return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
111     else
112     {
113         clkdiv = clk_power->CLKDIVN;
114         camdiv = clk_power->CAMDIVN;
115
```

```
116         /* 计算分频比 */
117
118         switch (clkdiv & S3C2440_CLKDIVN_HDIVN_MASK) {
119         case S3C2440_CLKDIVN_HDIVN_1:
120             hdiv = 1;
121             break;
122
123         case S3C2440_CLKDIVN_HDIVN_2:
124             hdiv = 2;
125             break;
126
127         case S3C2440_CLKDIVN_HDIVN_4_8:
128             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK4_HALF) ? 8 : 4;
129             break;
130
131         case S3C2440_CLKDIVN_HDIVN_3_6:
132             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK3_HALF) ? 6 : 3;
133             break;
134         }
135
136             return get_FCLK() / hdiv;
137     }
138 }
139
140 /* return PCLK frequency */
141 ulong get_PCLK(void)
142 {
143     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
144     unsigned long clkdiv;
145     unsigned long camdiv;
146     int hdiv = 1;
147
148     /* 同时支持 S3C2410 和 S3C2440 */
149     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
150         return((clk_power->CLKDIVN & 0x1) ? get_HCLK()/2 : get_HCLK());
151     else
152     {
153         clkdiv = clk_power->CLKDIVN;
154         camdiv = clk_power->CAMDIVN;
```

```
155
156         /* 计算分频比 */
157
158         switch (clkdiv & S3C2440_CLKDIVN_HDIVN_MASK) {
159         case S3C2440_CLKDIVN_HDIVN_1:
160             hdiv = 1;
161             break;
162
163         case S3C2440_CLKDIVN_HDIVN_2:
164             hdiv = 2;
165             break;
166
167         case S3C2440_CLKDIVN_HDIVN_4_8:
168             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK4_HALF) ? 8 : 4;
169             break;
170
171         case S3C2440_CLKDIVN_HDIVN_3_6:
172             hdiv = (camdiv & S3C2440_CAMDIVN_HCLK3_HALF) ? 6 : 3;
173             break;
174         }
175
176         return get_FCLK() / hdiv / ((clkdiv & S3C2440_CLKDIVN_PDIVN) ? 2:1);
177     }
178 }
179
```

现在重新执行“make 100ask24x0_config”和“make all”生成的U-Boot.bin文件既可以运行于S3C2410开发板，也可以运行于S3C2440开发板。将它烧入NOR Flash后启动，就可以在串口工具（设置为115200,8N1）中看到提示信息，可以输入各种命令操作U-Boot了。

（4）选择NOR Flash的型号。

但是，现在还无法通过U-Boot命令烧写NOR Flash。本书所用开发板中的NOR Flash型号为AM29LV800，而配置文件include/configs/100ask24x0.h中的默认型号为AM29LV400。修改如下：

```
#define CONFIG_AMD_LV400 1    /* uncomment this if you have a LV400 flash */
#if 0
#define CONFIG_AMD_LV800 1    /* uncomment this if you have a LV800 flash */
#endif
```

改为：

```
#if 0
```

```
#define CONFIG_AMD_LV400 1    /* uncomment this if you have a LV400 flash */
#endif
#define CONFIG_AMD_LV800 1    /* uncomment this if you have a LV800 flash */
```

本例中 NOR Flash 的操作函数在 board/100ask24x0/flash.c 中实现，它支持 AM29LV400y 和 AM29LV800。对于其他型号的 NOR Flash，如果符合 CFI 接口标准，则可以在使用 drivers/cfi_flash.c 中的接口函数；否则，只好自己编写了。如果要使用 cfi_flash.c，如下修改两个文件。

在 include/configs/100ask24x0.h 中增加以下一行：

```
#define CFG_FLASH_CFI_DRIVER    1
```

在 board/100ask24x0/Makefile 中去掉 flash.o：

```
COBJS    := 100ask24x0.o flash.o
改为:
COBJS    := 100ask24x0.o
```

修改好对 NOR Flash 的支持后，重新编译 U-Boot：make clean、make all。运行后可以在串口中看到如下字样：

```
Flash:  1 MB
```

现在可以使用 loadb、loady 等命令通过串口下载文件，然后使用 erase、cp 命令分别擦除、烧写 NOR Flash 了，它们的效率比 JTAG 高好几倍。

2. 支持串口 xmodem 协议

上面的 loadb 命令需要配合 Linux 下的 kermit 工具来使用，loady 命令通过串口 ymodem 协议来传输文件。Windows 下的超级终端虽然支持 ymodem，但是它的使用界面实在不友好。而本书推荐使用的 Windows 工具 SecureCRT 只支持 xmodem 和 zmodem。为了方便在 Windows 下开发，现在修改代码增加对 xmodem 的支持，即增加一个命令 loadx。

依照 loady 的实现来编写代码，首先使用 U_BOOT_CMD 宏来增加 loadx 命令：

```
/* 支持 xmodem, www.100ask.net */
U_BOOT_CMD(
        loadx, 3, 0,    do_load_serial_bin,
        "loadx   - load binary file over serial line (xmodem mode)\n",
        "[ off ] [ baud ]\n"
        "    - load binary file over serial line"
        " with offset 'off' and baudrate 'baud'\n"
);
```

其次，在 do_load_serial_bin 函数中增加对 loadx 命令的处理分支。也是依照 loady 来实现：

```
481     /* 支持 xmodem */
```

```
482     if (strcmp(argv[0],"loadx")==0) {
483         printf ("## Ready for binary (xmodem) download "
484             "to 0x%08lX at %d bps...\n",
485             offset,
486             load_baudrate);
487
488         addr = load_serial_xmodem (offset);
489
490     } else if (strcmp(argv[0],"loady")==0) {
491         printf ("## Ready for binary (ymodem) download "
492             "to 0x%08lX at %d bps...\n",
……
```

第 481～490 行就是为 loadx 命令增加的代码。

在第 288 行调用 load_serial_xmodem 函数，它是依照 load_serial_ymodem 实现的一个新函数：

```
36 #if (CONFIG_COMMANDS & CFG_CMD_LOADB)
37 /* 支持 xmodem */
38 static ulong load_serial_xmodem (ulong offset);
39 static ulong load_serial_ymodem (ulong offset);
40 #endif
……
995 /* 支持 xmodem */
996 static ulong load_serial_xmodem (ulong offset)
997 {
……
1003        char xmodemBuf[1024];              /* 原来是 ymodemBuf,这只是为了与函数名称一致 */
……
1008        info.mode = xyzModem_xmodem;       /* 原来是 xyzModem_ymodem，对应 ymodem */
……
```

首先在文件开头增加 load_serial_xmodem 函数的声明，然后复制 load_serial_ymodem 函数为 load_serial_xmodem，稍作修改。

① 将局部数组 ymodemBuf 改名为 xmodemBuf，并在后面使用到的地方统一修改。这只是为了与函数名称一致。

② info.mode 的值从 xyzModem_ymodem 改为 xyzModem_xmodem。

重新编译、烧写 U-Boot.bin 后，就可以使用 loadx 命令下载文件了。

3. 支持网卡芯片 CS8900

使用串口来传输文件的速率太低，现在增加对网卡芯片 CS8900 的支持。

本书使用开发板的网卡芯片 CS8900 的连接方式与 smdk2410 完全一样，所以现在的 U-Boot 中已经支持 CS8900 了，它的驱动程序为 drivers/cs8900.c。只要在 U-Boot 控制界面中稍加配置就可以使用网络功能。使用网络之前，先设置开发板 IP 地址、MAC 地址，服务器 IP 地址，比如可以在 U-Boot 中执行以下命令：

```
setenv ipaddr 192.168.1.17
setenv ethaddr 08:00:3e:26:0a:5b
setenv serverip 192.168.1.11
saveenv
```

然后就可以使用 tftp 或 nfs 命令下载文件了，注意：服务器上要开启 tftp 或 nfs 服务。比如可以使用如下命令将 U-Boot.bin 文件下载到内存 0x30000000 中：

```
tftp 0x30000000 U-Boot.bin
或
nfs 0x30000000 192.168.1.57:/work/nfs_root/U-Boot.bin
```

可以修改配置文件，让网卡的各个默认值就是上面设置的值。在此之前，先了解网卡的相关文件，这有助于移植代码以支持其他连接方式的 CS8900。

首先，CS8900 接在 S3C2410、S3C2440 的 BANK3，位宽为 16，使用 WAIT、nBE 信号。在设置存储控制器时要设置好 BANK3。代码在 board/100ask24x0/lowlevel_init.S 中，如下所示：

```
#define B3_BWSCON       (DW16 + WAIT + UBLB)
……
/* 时序参数 */
#define B3_Tacs         0x0 /*  0clk */
#define B3_Tcos         0x3 /*  4clk */
#define B3_Tacc         0x7 /* 14clk */
#define B3_Tcoh         0x1 /*  1clk */
#define B3_Tah          0x0 /*  0clk */
#define B3_Tacp         0x3  /*  6clk */
#define B3_PMC          0x0 /* normal */
```

接下来，还要确定 CS8900 的基地址。这在配置文件 include/configs/100ask24x0.h 中定义，如下所示：

```
#define CONFIG_DRIVER_CS8900    1           /* 使用 CS8900 */
#define CS8900_BASE             0x19000300  /* 基地址 */
#define CS8900_BUS16            1           /* 位宽为 16 */
```

网卡 CS8900 的访问基址为 0x19000000，之所以再偏移 0x300 是由它的特性决定的。

最后，还是在配置文件 include/configs/100ask24x0.h 中定义 CS8900 的各个默认地址，如下所示：

```
#define CONFIG_ETHADDR      08:00:3e:26:0a:5b
```

```
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.1.17
#define CONFIG_SERVERIP     192.168.1.11
```

如果要增加 ping 命令，还可以在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_PING，如下所示：

```
#define CONFIG_COMMANDS \
                    (CONFIG_CMD_DFL     | \
                    CFG_CMD_CACHE  | \
                    CFG_CMD_PING        | \
……
```

4. 支持 NAND Flash

U-Boot 1.1.6 中对 NAND Flash 的支持有新旧两套代码，新代码在 drivers/nand 目录下，旧代码在 drivers/nand_legacy 目录下。文档 doc/README.nand 对这两套代码有所说明：使用旧代码需要定义更多的宏，而新代码移植自 Linux 内核 2.6.12，它更加智能，可以自动识别更多型号的 NAND Flash。目前之所以还保留旧的代码，是因为两个目标板 NETTA、NETTA_ISDN 使用 JFFS 文件系统，它们还依赖于旧代码。当相关功能移植到新代码之后，旧的代码将从 U-Boot 中去除。

要让 U-Boot 支持 NAND Flash，首先在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_NAND，如下所示：

```
#define CONFIG_COMMANDS \
                    (CONFIG_CMD_DFL     | \
                    CFG_CMD_CACHE  | \
                    CFG_CMD_PING        | \
                    CFG_CMD_NAND  | \
…
```

然后选择使用哪套代码：在配置文件中定义宏 CFG_NAND_LEGACY 则使用旧代码，否则使用新代码。

使用旧代码时，需要实现 drivers/nand_legacy/nand_legacy.c 中使用到的各种宏，比如：

```
#define NAND_WAIT_READY(nand)    /* 等待 Nand Flash 的状态为“就绪”，代码依赖于具体的开发板 */
#define WRITE_NAND_COMMAND(d, adr)    /* 写 NAND Flash 命令，代码依赖于具体的开发板 */
```

本书使用新代码，下面讲述移植过程。

代码的移植没有现成的文档，可以在配置文件 include/configs/100ask24x0.h 的宏 CONFIG_COMMANDS 中增加 CFG_CMD_NAND 后就编译代码，然后一个一个地解决出现的错误。编译结果中出现的错误和警告如下：

```
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
```

在配置文件include/configs/100ask24x0.h中增加如下3个宏就可以解决上述错误。在Flash的驱动程序中，设备是逻辑上的概念，表示一组相同结构、访问函数相同的Flash芯片。在本书所用开发板中，只有一个NAND Flash芯片，所以设备数为1，芯片数也为1。

```
#define CFG_NAND_BASE          0     /* 无实际意义：基地址，这在 board_nand_init
中重新指定 */
#define CFG_MAX_NAND_DEVICE    1     /* NAND Flash“设备”的数目为1 */
#define NAND_MAX_CHIPS         1     /* 每个NAND Flash“设备”由1个NAND Flash
“芯片”组成 */
```

修改配置文件后再次编译，现在只有一个错误了，“board_nand_init 函数未定义”，如下所示：

```
nand.c:50: undefined reference to `board_nand_init'
```

调用board_nand_init函数的过程为：NAND Flash的初始化入口函数是nand_init，它在lib_arm/board.c的start_armboot函数中被调用；nand_init函数在drivers/ nand/nand.c中实现，它调用相同文件中的nand_init_chip函数；nand_init_chip函数首先调用board_nand_init函数来初始化NAND Flash设备，最后才是统一的识别过程。

从board_nand_init函数的名称就可以知道它是平台/开发板相关的函数，需要自己编写。本书在cpu/arm920t/s3c24x0目录下新建一个文件nand_flash.c，在里面针对S3C2410、S3C2440实现了统一的board_nand_init函数。

在编写board_nand_init函数的之前，需要针对S3C2410、S3C2440 NAND Flash控制器的不同来定义一些数据结构和函数：

（1）在include/s3c24x0.h文件中增加S3C2440_NAND数据结构。

```
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;
```

（2）在 include/s3c2410.h 文件中仿照 S3C2410_GetBase_NAND 函数定义 S3C2440_GetBase_NAND 函数。

```
/* for s3c2440 */
static inline S3C2440_NAND * const S3C2440_GetBase_NAND(void)
{
    return (S3C2440_NAND * const)S3C2410_NAND_BASE;
}
```

既然新的 NAND Flash 代码是从 Linux 内核 2.6.12 中移植来的，那么 cpu/arm920t/s3c24x0/nand_flash.c 文件也可以仿照内核中对 S3C2410、S3C2440 的 NAND Flash 进行初始化的 drivers/mtd/nand/s3c2410.c 文件来编写。为了方便阅读，先把 cpu/arm920t/s3c24x0/nand_flash.c 文件的代码全部列出来，如下所示：

```
01 /*
02  * s3c2410/s3c2440 的 NAND Flash 控制器接口
03  * 修改自 Linux 内核 2.6.13 文件 drivers/mtd/nand/s3c2410.c
04  */
05
06 #include <common.h>
07
08 #if (CONFIG_COMMANDS & CFG_CMD_NAND) && !defined(CFG_NAND_LEGACY)
09 #include <s3c2410.h>
10 #include <nand.h>
11
12 DECLARE_GLOBAL_DATA_PTR;
13
14 #define S3C2410_NFSTAT_READY    (1<<0)
15 #define S3C2410_NFCONF_nFCE     (1<<11)
16
17 #define S3C2440_NFSTAT_READY    (1<<0)
```

```
18 #define S3C2440_NFCONT_nFCE     (1<<1)
19
20
21 /* S3C2410: NAND Flash的片选函数 */
22 static void s3c2410_nand_select_chip(struct mtd_info *mtd, int chip)
23 {
24     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
25
26     if (chip == -1) {
27         s3c2410nand->NFCONF |= S3C2410_NFCONF_nFCE; /* 禁止片选信号 */
28     } else {
29         s3c2410nand->NFCONF &= ~S3C2410_NFCONF_nFCE;   /* 使能片选信号 */
30     }
31 }
32
33 /* S3C2410: 命令和控制函数
34  *
35  * 注意，这个函数仅仅根据各种命令来修改“写地址” IO_ADDR_W 的值(这称为 tglx 方法)，
36  * 这种方法使得平台/开发板相关的代码很简单。
37  * 真正发出命令是在上一层 NAND Flash 的统一的驱动中实现，
38  * 它首先调用这个函数修改“写地址”，然后才分别发出控制、地址、数据序列。
39 */
40 static void s3c2410_nand_hwcontrol(struct mtd_info *mtd, int cmd)
41 {
42     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
43     struct nand_chip *chip = mtd->priv;
44
45     switch (cmd) {
46     case NAND_CTL_SETNCE:
47     case NAND_CTL_CLRNCE:
48         printf("%s: called for NCE\n", __FUNCTION__);
49         break;
50
51     case NAND_CTL_SETCLE:
52         chip->IO_ADDR_W = (void *)&s3c2410nand->NFCMD;
53         break;
54
55     case NAND_CTL_SETALE:
56         chip->IO_ADDR_W = (void *)&s3c2410nand->NFADDR;
```

```
57          break;
58
59          /* NAND_CTL_CLRCLE: */
60          /* NAND_CTL_CLRALE: */
61      default:
62          chip->IO_ADDR_W = (void *)&s3c2410nand->NFDATA;
63          break;
64      }
65 }
66
67 /* S3C2410: 查询NAND Flash状态
68  *
69  * 返回值: 0表示忙,1表示就绪
70  */
71 static int s3c2410_nand_devready(struct mtd_info *mtd)
72 {
73      S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
74
75      return (s3c2410nand->NFSTAT & S3C2410_NFSTAT_READY);
76 }
77
78
79 /* S3C2440: NAND Flash的片选函数 */
80 static void s3c2440_nand_select_chip(struct mtd_info *mtd, int chip)
81 {
82      S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
83
84      if (chip == -1) {
85          s3c2440nand->NFCONT |= S3C2440_NFCONT_nFCE; /* 禁止片选信号 */
86      } else {
87          s3c2440nand->NFCONT &= ~S3C2440_NFCONT_nFCE;   /* 使能片选信号 */
88      }
89 }
90
91 /* S3C2440: 命令和控制函数，与s3c2410_nand_hwcontrol函数类似 */
92 static void s3c2440_nand_hwcontrol(struct mtd_info *mtd, int cmd)
93 {
94      S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
95      struct nand_chip *chip = mtd->priv;
```

```
96
97      switch (cmd) {
98      case NAND_CTL_SETNCE:
99      case NAND_CTL_CLRNCE:
100         printf("%s: called for NCE\n", __FUNCTION__);
101         break;
102
103      case NAND_CTL_SETCLE:
104         chip->IO_ADDR_W = (void *)&s3c2440nand->NFCMD;
105         break;
106
107      case NAND_CTL_SETALE:
108         chip->IO_ADDR_W = (void *)&s3c2440nand->NFADDR;
109         break;
110
111         /* NAND_CTL_CLRCLE: */
112         /* NAND_CTL_CLRALE: */
113      default:
114         chip->IO_ADDR_W = (void *)&s3c2440nand->NFDATA;
115         break;
116      }
117 }
118
119 /* S3C2440: 查询 NAND Flash 状态
120  *
121  * 返回值: 0 表示忙,1 表示就绪
122  */
123 static int s3c2440_nand_devready(struct mtd_info *mtd)
124 {
125     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
126
127     return (s3c2440nand->NFSTAT & S3C2440_NFSTAT_READY);
128 }
129
130 /*
131  * Nand flash 硬件初始化:
132  * 设置 NAND Flash 的时序, 使能 NAND Flash 控制器
133  */
134 static void s3c24x0_nand_inithw(void)
```

```
135 {
136     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
137     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
138
139 #define TACLS   0
140 #define TWRPH0  4
141 #define TWRPH1  2
142
143     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410)
144     {
145         /* 使能 NAND Flash 控制器，初始化 ECC，使能片选信号，设置时序 */
146         s3c2410nand->NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)| (TWRPH0<<4)|
(TWRPH1<<0);
147     }
148     else
149     {
150         /* 设置时序 */
151         s3c2440nand->NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
152         /* 初始化 ECC，使能 NAND Flash 控制器，使能片选信号 */
153         s3c2440nand->NFCONT = (1<<4)|(0<<1)|(1<<0);
154     }
155 }
156
157 /*
158  * 被 drivers/nand/nand.c 调用，初始化 NAND Flash 硬件，初始化访问接口函数
159  */
160 void board_nand_init(struct nand_chip *chip)
161 {
162     S3C2410_NAND * const s3c2410nand = S3C2410_GetBase_NAND();
163     S3C2440_NAND * const s3c2440nand = S3C2440_GetBase_NAND();
164
165     s3c24x0_nand_inithw();  /* Nand flash 硬件初始化 */
166
167     if (gd->bd->bi_arch_number == MACH_TYPE_SMDK2410) {
168         chip->IO_ADDR_R    = (void *)&s3c2410nand->NFDATA;
169         chip->IO_ADDR_W    = (void *)&s3c2410nand->NFDATA;
170         chip->hwcontrol    = s3c2410_nand_hwcontrol;
171         chip->dev_ready    = s3c2410_nand_devready;
172         chip->select_chip  = s3c2410_nand_select_chip;
```

```
173         chip->options      = 0; /* 设置位宽等，位宽为 8 */
174     } else {
175         chip->IO_ADDR_R    = (void *)&s3c2440nand->NFDATA;
176         chip->IO_ADDR_W    = (void *)&s3c2440nand->NFDATA;
177         chip->hwcontrol    = s3c2440_nand_hwcontrol;
178         chip->dev_ready    = s3c2440_nand_devready;
179         chip->select_chip  = s3c2440_nand_select_chip;
180         chip->options      = 0; /* 设置位宽等，位宽为 8 */
181     }
182
183     chip->eccmode      = NAND_ECC_SOFT; /* ECC 校验方式: 软件 ECC */
184 }
185
186 #endif
```

文件中分别针对 S3C2410、S3C2440 实现了 NAND Flash 最底层访问函数，并进行了一些硬件的设置（比如时序、使能 NAND Flash 控制器等）。新的代码对 NAND Flash 的封装做得很好，只要向上提供底层初始化函数 board_nand_init 来设置好平台/开发板相关的初始化、提供底层接口即可。

最后，只要将新建的 nand_flash.c 文件编入 U-Boot 中就可以擦除、读写 NAND Flash 了。如下修改 cpu/arm920t/s3c24x0/Makefile 文件即可。

修改前：

```
COBJS   = i2c.o interrupts.o serial.o speed.o \
          usb_ohci.o
```

修改后：

```
COBJS   = i2c.o interrupts.o serial.o speed.o \
          usb_ohci.o nand_flash.o
```

现在，可以使用新编译的 U-Boot.bin 烧写内核映象到 NAND Flash 去了。

5. 支持烧写 yaffs 文件系统映象

在实际生产中，可以通过烧片器等手段将内核、文件系统映象烧入固态存储设备中，Bootloader 不需要具备烧写功能。但为了方便开发，通常在 Bootloader 中增加烧写内核、文件系统映象文件的功能。

增加了 NAND Flash 功能的 U-Boot 1.1.6 已经可以通过“nand write…”、“nand write.jffs2…”等命令来烧写内核，烧写 cramfs、jffs2 文件系统映象文件。但是在 NAND Flash 上，yaffs 文件系统的性能更佳，下面增加“nand write.yaffs…”命令以烧写 yaffs 文件系统映象文件。

“nand write.yaffs…”字样的命令中，“nand”是具体命令，“write.yaffs…”是参数。nand

命令在 common/cmd_nand.c 中实现如下：

```
U_BOOT_CMD(nand, 5, 1, do_nand,
    "nand    - NAND sub-system\n",
    "info                    - show available NAND devices\n"
    "nand device [dev]       - show or set current device\n"
    "nand read[.jffs2]       - addr off|partition size\n"
    "nand write[.jffs2]      - addr off|partiton size - read/write 'size' bytes
starting\n"
    "    at offset 'off' to/from memory address 'addr'\n"
...
```

先在其中增加“nand write.yaffs…”的使用说明，如下所示：

```
U_BOOT_CMD(nand, 5, 1, do_nand,
    "nand    - NAND sub-system\n",
    "info                    - show available NAND devices\n"
    "nand device [dev]       - show or set current device\n"
    "nand read[.jffs2]       - addr off|partition size\n"
    "nand write[.jffs2]      - addr off|partiton size - read/write 'size' bytes
starting\n"
    "    at offset 'off' to/from memory address 'addr'\n"
    "nand read.yaffs addr off size - read the 'size' byte yaffs image
starting\n"
    "    at offset 'off' to memory address 'addr'\n"
    "nand write.yaffs addr off size - write the 'size' byte yaffs image
starting\n"
    "    at offset 'off' from memory address 'addr'\n"
...
```

然后，在 nand 命令的处理函数 do_nand 中增加对“write.yaffs…”的支持。do_nand 函数仍在 common/cmd_nand.c 中实现，代码修改如下：

```
331         (!strcmp(s, ".jffs2") || !strcmp(s, ".e") || !strcmp(s, ".i"))) {
...
354         }else if ( s != NULL && !strcmp(s, ".yaffs")){
355             if (read) {
356                 /* read */
357                 nand_read_options_t opts;
358                 memset(&opts, 0, sizeof(opts));
359                 opts.buffer = (u_char*) addr;
360                 opts.length = size;
```

```
361                 opts.offset = off;
362                 opts.readoob = 1;
363                 opts.quiet      = quiet;
364                 ret = nand_read_opts(nand, &opts);
365             } else {
366                 /* write */
367                 nand_write_options_t opts;
368                 memset(&opts, 0, sizeof(opts));
369                 opts.buffer = (u_char*) addr;    /* yaffs 文件系统映象存放的地址 */
370                 opts.length = size;              /* 长度 */
371                 opts.offset = off;               /* 要烧写到的 NAND Flash 的偏移地址 */
372                 /* opts.forceyaffs = 1; */       /* 计算 ECC 码的方法，没有使用 */
373                 opts.noecc = 1;                  /* 不需要计算 ECC，yaffs 映
象中有 OOB 数据 */
374                 opts.writeoob = 1;               /* 写 OOB 区 */
375                 opts.blockalign = 1;             /* 每个“逻辑上的块”大小为 1 个“物
理块” */
376                 opts.quiet      = quiet;         /* 是否打印提示信息 */
377                 opts.skipfirstblk = 1;           /* 跳过第一个可用块 */
378                 ret = nand_write_opts(nand, &opts);
379             }
380         } else {
...
385         }
386
```

第 354～379 行就是针对命令“nand read.yaffs…”、“nand write.yaffs…”增加的代码。有兴趣的读者可以自己分析“if (read)”分支的代码，下面只讲解“else”分支，即“nand write.yaffs…”命令的实现。

NAND Flash 每一页大小为（512+16）字节（还有其他格式的 NAND Flash，比如每页大小为(256+8)、(2048+64)等），其中的 512 字节就是一般存储数据的区域，16 字节称为 OOB（Out Of Band）区。通常在 OOB 区存放坏块标记、前面 512 字节的 ECC 校验码等。

cramfs、jffs2 文件系统映象文件中并没有 OOB 区的内容，如果将它们烧入 NOR Flash 中，则是简单的“平铺”关系；如果将它们烧入 NAND Flash 中，则 NAND Flash 的驱动程序首先根据 OOB 的标记略过坏块，然后将一页数据（512 字节）写入后，还会计算这 512 字节的 ECC 校验码，最后将它写入 OOB 区，如此循环。cramfs、jffs2 文件系统映象文件的大小通常是 512 的整数倍。

而 yaffs 文件系统映象文件的格式则跟它们不同，文件本身就包含了 OOB 区的数据（里面有坏块标记、ECC 校验码、其他 yaffs 相关的信息）。所以烧写时，不需要再计算 ECC 值，首先检查是否坏块（是则跳过），然后写入 512 字节的数据，最后写入 16 字节的 OOB 数据，

如此循环。yaffs 文件系统映象文件的大小是（512+16）的整数倍。

注意 烧写 yaffs 文件系统映象时，分区上第一个可用的（不是坏块）块也要跳过。

下面分析上面的代码。

第 369～371 行设置源地址、目的地址、长度。烧写 yaffs 文件系统映象前，一般通过网络将它下载到内存某个地址处（比如 0x30000000），然后通过类似"nand write.yaffs 0x30000000 0x00A00000 $(filesize)"的命令烧到 NAND Flash 的偏移地址 0x00A00000 处。对于这个命令，第 369 行中 opts.buffer 等于 0x30000000，第 370 行中 opts.length 等于$(filesize)的值，就是前面下载的文件的大小，第 371 行中的 opts.offset 等于 0x00A00000。

这里列出不使用的第 372 行，是因为 opts.forceyaffs 这个名字很有欺骗性，它其实是指计算 ECC 校验码的一种方法。烧写 yaffs 文件系统映象时，不需要计算 ECC 校验码。

第 373、374 行指定烧写数据时不计算 ECC 校验码、而是烧入文件中的 OOB 数据。

第 375 行指定"逻辑块"的大小，"逻辑块"可以由多个"物理块"组成，在 yaffs 文件系统映象中，它们是 1:1 的关系。

第 377 行的 opts.skipfirstblk 是新加的项，nand_write_options_t 结构中没有 skipfirstblk 成员。它表示烧写时跳过第一个可用的逻辑块，这是由 yaffs 文件系统的特性决定的。

既然 skipfirstblk 是在 nand_write_options_t 结构中新加的项，那么就要重新定义 nand_write_options_t 结构，并在下面调用的 nand_write_opts 函数中对它进行处理。

首先在 include/nand.h 中进行如下修改，增加 skipfirstblk 成员。

```
struct nand_write_options {
        u_char *buffer;     /* memory block containing image to write */
        ulong length;       /* number of bytes to write */
        ulong offset;       /* start address in NAND */
        int quiet;          /* don't display progress messages */
        int autoplace;      /* if true use auto oob layout */
        int forcejffs2;     /* force jffs2 oob layout */
        int forceyaffs;     /* force yaffs oob layout */
        int noecc;          /* write without ecc */
        int writeoob;       /* image contains oob data */
        int pad;            /* pad to page size */
       int blockalign;      /* 1|2|4 set multiple of eraseblocks to align to */
        int skipfirstblk;   /* 新加，烧写时跳过第一个可用的逻辑块 */
};
typedef struct nand_write_options nand_write_options_t;
```

然后，修改 nand_write_opts 函数，增加对 skipfirstblk 成员的支持。它在 drivers/nand/nand_util.c 文件中，下面的第 301、第 421～424 行的新加的。

```
285 int nand_write_opts(nand_info_t *meminfo, const nand_write_options_t *opts)
286 {
```

```
...
300     int result;
301     int skipfirstblk = opts->skipfirstblk;
...
397         while (blockstart != (mtdoffset & (~erasesize_blockalign+1))) {
...
419                 }
420
421                 if (baderaseblock) {
422                     mtdoffset = blockstart
423                         + erasesize_blockalign;
424                 }
...
```

进行了上面的移植后，U-Boot 已经可以烧写 yaffs 文件系统映象了。由于前面设置“opts.noecc = 1”不使用 ECC 校验码，在烧写过程中会出现很多的提示信息，如下所示：

```
Writing data without ECC to NAND-FLASH is not recommended
```

可以修改 drivers/nand/nand_base.c 文件的 nand_write_page 函数，将它去掉。

修改前：

```
917 case NAND_ECC_NONE:
918     printk (KERN_WARNING "Writing data without ECC to NAND-FLASH is not recommended\n");
```

修改后：

```
917 case NAND_ECC_NONE:
918     //printk (KERN_WARNING "Writing data without ECC to NAND-FLASH is not recommended\n");
```

6. 修改默认配置参数以方便使用

前面移植网卡芯片 CS8900 时，已经设置过默认 IP 地址等。为了使用 U-Boot 时减少一些设置，现在修改配置文件 include/configs/100ask24x0.h，增加默认配置参数，其中一些在移植过程中已经增加的选项这里也再次说明。

（1）Linux 启动参数。

增加如下 3 个宏：

```
#define CONFIG_SETUP_MEMORY_TAGS    1       /* 向内核传递内存分布信息 */
#define CONFIG_CMDLINE_TAG          1       /* 向内核传递命令行参数 */
/* 默认命令行参数 */
#define CONFIG_BOOTARGS         "noinitrd root=/dev/mtdblock 2 init=/linuxrc
```

```
console=ttySAC0"
```

（2）自动启动命令。

增加如下2个宏：

```
/* 自动启动前延时 3s */
#define CONFIG_BOOTDELAY        3
/* 自动启动的命令 */
#define CONFIG_BOOTCOMMAND      "nboot 0x32000000 0 0; bootm 0x32000000"
```

自动启动时（开机3s内无输入），首先执行“nboot 0x32000000 0 0”命令将第0个NAND Flash偏移地址0上的映象文件复制到内存0x32000000中；然后执行“bootm 0x32000000”命令启动内存中的映象。

（3）默认网络设置。

根据具体网络环境增加、修改下面4个宏：

```
#define CONFIG_ETHADDR      08:00:3e:26:0a:5b
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.1.17
#define CONFIG_SERVERIP     192.168.1.11
```

15.2.6 U-Boot的常用命令

1. U-Boot的常用命令的用法

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输入回车。

U-Boot接收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，相关命令有：loadb、loads、loadx、loady和tftpboot、nfs。

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]”。“[]”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表示使用的波特率。如果baud参数省略，则使用当前的波特率；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 表示要下载的文件的名称。如果 loadAddress 省略，存放的地址为配置文件中定义的宏 CFG_LOAD_ADDR；如果 bootfilename 省略，则使用开发板的 IP 地址构造一个文件名，比如开发板 IP 为 192.168.1.17，则默认的文件名为 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。

下载文件成功后，U-Boot 会自动创建或更新环境变量 filesize，它表示下载的文件的长度，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 md、填充内存命令 mw、复制命令 cp。这些命令都可以带上后缀“.b”、“.w”或“.l”，表示以字节、字（2 个字节）、双字（4 个字节）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 个双字到开始地址为 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，输入新数据后回车，地址会自动增加，按“Ctrl+C”键退出。

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”，擦除的地址范围为start～end；“erase start +len”，擦除的地址范围为start～（star+tlen−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的长度，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。是否加后缀“.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格式的映象文件（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，映象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 处的映象文件复制到内存 loadAddr 处，然后，如果环境变量 autostart 的值为“yes”，就启动这个映象。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。

2. U-Boot 命令使用实例

下面通过一个例子来演示如何使用各种命令烧写内核映象文件、yaffs 映象文件，并启动系统。

（1）制作内核映象文件。

对于本书使用的 Linux 2.6.22.6 版本，编译内核时可以直接生成 U-Boot 格式的映象文件 uImage。

对于不能直接生成 uImage 的内核，制作方法在 U-Boot 根目录下的 README 文件中有说明，假设已经编译好的内核文件为 vmlinux，它是 ELF 格式的。mkimage 是 U-Boot 目录 tools 下的工具，它在编译 U-Boot 时自动生成。执行以下 3 个命令将内核文件 vmlinux 制作为 U-Boot 格式的映象文件 uImage，它们首先将 vmlinux 转换为二进制格式，然后压缩，最后构造头部信息（里面包含有文件名称、大小、类型、CRC 校验码等），如下所示。

```
① arm-linux-objcopy -O binary -R .note -R .comment -S vmlinux linux.bin
② gzip -9 linux.bin
③ mkimage -A arm -O linux -T kernel -C gzip -a 0x30008000 -e 0x30008000 -n
"Linux Kernel Image" -d linux.bin.gz uImage
```

（2）烧写内核映象文件 uImage。

首先将 uImage 放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务。

然后运行如下命令下载文件，擦除、烧写 NAND Flash，如下所示。

```
① tftp 0x30000000 uImage 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/uImage
② nand erase 0x0 0x00200000
③ nand write.jffs2 0x30000000 0x0 $(filesize)
```

第 3 条命令之所以使用“nand write.jffs2”而不是“nand write”，是因为前者不要求文件的长度是页对齐的（512 字节对齐）。也可以使用“nand write”，但是需要将命令中的长度参数改为$(filesize)向上进行 512 取整（比如，513 向上进行 512 取整，结果为 512 × 2=1024）后的值。比如 uImage 的大小为 1540883，向上进行 512 取整后为 1541120（即 0x178400），可以使用命令“nand write 0x30000000 0x0 0x178400”进行烧写。

（3）烧写 yaffs 文件系统映象。

假设 yaffs 文件系统映象的文件名为 yaffs.img，首先将它放在主机上的 tftp 或 nfs 目录下，确保已经开启 tftp 或 nfs 服务；然后执行如下命令下载、擦除、烧写，如下所示。

```
① tftp 0x30000000 yaffs.img 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
yaffs.img
② nand erase 0xA00000 0x3600000
```

```
③ nand write.yaffs 0x30000000 0xA00000 $(filesize)
```

这时，重启系统，在U-Boot倒数3s之后，就会自动启动Linux系统。

（4）烧写jffs2文件系统映象。

假设jffs2文件系统映象的文件名为jffs2.img，首先将它放在主机上的tftp或nfs目录下，确保已经开启tftp或nfs服务；然后执行如下命令下载、擦除、烧写，如下所示。

```
① tftp 0x30000000 jffs2.img 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
jffsz.img
② nand erase 0x200000 0x800000
③ nand write.jffs2 0x30000000 0x200000 $(filesize)
```

系统启动后，就可以使用“mount -t jffs2 /dev/mtdblock1 /mnt”挂接jffs2文件系统。

15.2.7 使用U-Boot来执行程序

在前面的实例中使用JTAG烧写程序到NAND Flash，烧写过程十分缓慢。如果使用U-Boot来烧写NAND Flash，效率会高很多。烧写二进制文件到NAND Flash中所使用的命令与上面烧写内核映象文件uImage的过程类似，只是不需要将二进制文件制作成U-Boot格式。

另外，可以将程序下载到内存中，然后使用go命令执行它。假设有一个程序的二进制可执行文件test.bin，连接地址为0x30000000。首先将它放在主机上的tftp或nfs目录下，确保已经开启tftp或nfs服务；然后将它下载到内存0x30000000处，最后使用go命令执行它，如下所示。

```
① tftp 0x30000000 test.bin 或 nfs 0x30000000 192.168.1.57:/work/nfs_root/
test.bin
② go 0x30000000
```