嵌入式系统开发原理与实验

杨延军 主编
杨延军 王志军 赵建业 段晓辉 编著

北京大学出版社
2010 年 6 月

内 容 提 要

本书是北京大学电子信息科学基础实验中心《电子信息科学基础实验课程丛书》之一，在北京大学电子信息科学基础实验课程体系中属于研究创新层次。全书共分为八章，每章都包括背景知识介绍和实验两个部分，内容涉及从底层的汇编语言设计到高层的图形界面程序设计，基本涵盖了嵌入式 Linux 软件开发的主要方面。

本书以 GNU/Linux 操作系统作为实验平台，嵌入式开发板采用最流行的 ARM 处理器，对于没有开发板的读者还可以使用附录中介绍的 QEMU 完成本书的大部分实验内容。本书可以作为高等院校电子信息类本科生嵌入式相关课程的教材，也可以作为教师和工程技术人员的参考书。

丛 书 序 言

本科教育是北京大学长远发展中最基础、最重要的工作之一，而实验教学是本科教育特别是一些基础学科教育的重要组成部分，是衡量学校教育质量的重要指标，是培养学生的实验能力以及实践与创新精神的重要过程，是培养高水平、创新型人才的重要手段，同时也是新的形势对高等教育教学的迫切要求。

我校根据改革开放以后国内高等教育形势、规模和人才需求结构的变化，借鉴国际上先进的教学理念并结合我国的实际情况，制定了“加强基础、淡化专业、因材施教、分流培养”的教学改革十六字方针。为了适应电子信息技术的发展，全面培养电子与信息科学类专业的高素质创新型人才，我校于 2000 年 9 月成立了北京大学电子信息科学基础实验中心，全面负责全校电子信息类基础实验课程的实施、改革和建设工作。

根据我校电子信息科学类专业本科生理论基础扎实、人数相对工科院校较少的特点，近年来，实验教学中心进行了具有研究、综合型大学特点的“电子信息科学基础实验课程体系”的建设。形成了模块化、多层次，具有理工相结合特色的实验课程体系，并将专业基础实验课程纳入到课程体系中来。实验课程既与相应理论课程相互呼应，同时又保持了自身的体系与特色。

在学校和信息科学技术学院领导的关心和支持下，实验教学中心组织在教学第一线的骨干教师，总结多年来实践教学和改革经验，并参考兄弟院校的实践教学改革成果，编写了电子信息科学基础实验课程丛书。该套丛书具有理工相结合的特色，在实验内容选择上注重深度，注重启发性、研究性和综合性，同时将 EDA 等技术有机地融入到实验课程中去，以便全面地培养学生的综合研究能力和创新意识。

电子信息科学基础实验课程丛书共 12 本，涵盖了北京大学电子信息科学基础实验课程体系的 4 个层次。

基础实验层次：

《电路基础实验》、《电子线路实验》、《数字逻辑电路实验》

综合设计层次：

《电子线路计算机辅助设计》、《微机原理与接口技术》、《可编程逻辑电路设计》

研究创新层次：

《电子系统设计》、《嵌入式系统开发原理与实验》

专业基础层次：

《通信电路实验》、《光电子实验》、《集成电路设计》、《数字信号处理实验》

本套丛书的筹划和编写得到了原电子学系主任王楚教授的关心和指导；信息科学技术学院陈徐宗副院长，实验教学中心顾问唐镇松教授、沈伯弘教授在整个编写过程中，都进行了严格把关和悉心指导。同时，北京大学教务部、北京大学出版社和信息科学技术学院的领导都给予了大力支持和帮助。在此，向他们致以崇高的敬意，并表示衷心的感谢。向所有关心、支持和帮助过本丛书编写、修改、出版、发行工作的各位同仁致以诚挚的谢意。

限于作者的水平和经验，丛书中尚存疏漏和不足，敬请专家和读者不吝指教。

北京大学
电子信息科学基础实验中心
2010 年 6 月

前　言

本书是根据北京大学电子信息科学基础实验中心“嵌入式系统实验”课程需要编写的教材，可作为高等院校相关专业以及对这一领域知识感兴趣的读者的参考教材。

“嵌入式系统实验”是信息科学技术学院一门本科生选修课，目的是为没有任何嵌入式开发经验的本科生介绍相关的开发工具、硬件和开发方法。通过本课程的学习，学生可以了解 Linux 操作系统的基本原理，掌握在 Linux 操作系统环境下进行软件开发的基本方法，对 ARM 处理器和相关硬件平台也有初步的认识，为今后从事相关的设计与开发打下基础。

本书共分为八章，每章都包含了背景知识介绍和实验两个部分，内容涉及了开发环境的建立与使用、ARM 汇编语言程序设计、操作系统驱动编写和图形界面程序设计。操作性的实验都包含完整的实验步骤，可以帮助入门的读者完成实验内容。

本书第一章主要对嵌入式系统的基本概念进行介绍，同时介绍了本书所使用的软硬件环境，在最后的实验项目中介绍了在虚拟机中安装 Ubuntu Linux 操作系统的方法。第二章介绍了 Linux 操作系统的使用方法，并对 Linux 操作系统下的开发工具进行了介绍。本章的实验项目是在硬件平台安装嵌入式 Linux 及相关软件。第三章主要介绍了 ARM 处理器的体系结构与指令系统，并在实验中对汇编语言进行练习。第四章介绍了本书所使用的处理器及开发板，实验内容是通过汇编和 C 语言对开发板硬件进行操作。第五章介绍了 Linux 内核及文件系统的相关知识，在实验中要完成编译 Linux 内核与构建嵌入式 Linux 的网络文件系统。第六章介绍了 Linux 操作系统环境下的程序开发，以实现嵌入式开发板上的网络聊天程序。第七章介绍了 Linux 操作系统驱动的编写，在实验中要完成对字符设备驱动的编写与测试。第八章介绍了 Qt 图形界面程序的设计方法，其中的实验内容是利用 Qt 库编写简单的应用程序。

对于没有硬件平台的读者可以采用附录 A 中介绍的 QEMU 软件对嵌入式平台进行仿真，并在仿真环境中完成嵌入式实验。本书编者对 QEMU 软件进行了修改，相应的补丁文件保存在本书光盘中，或者联系本书编者获得最新的补丁文件。采用 QEMU 完成实验的过程可以参考附录 A 中的介绍。本书附录 B 介绍了开源软件 OpenEmbedded 及使用 OpenEmbedded 构建嵌入式文件系统的方法，可以作为读者的参考。

本课程实验部分可以在六周完成，每周安排四个学时。其中第一章实验作为课余练习，增加学生熟悉 Linux 操作系统的时间。第三和第四章的实验可以在四个学时内完成，其余每章的实验都安排在一次课内完成。对于没有 Linux 操作系统使用经验的学生，按照前面安排的进度完成实验可能会有些困难，因此要求不熟悉 Linux 操作系统的学生多在课余时间接触 Linux，并做好每次实验的课前预习。

在六周实验之后，学生可以分成 2 或 3 人一组，完成一个比较复杂的嵌入式开发项目。内容可以是底层的操作系统移植及驱动开发，或者是较复杂的嵌入式应用或游戏。共安排四周时间完成项目实验，最后利用一次课的时间对项目完成情况进行验收和交流。

本书所附带的光盘中包含了完成实验所必需的软件工具与源代码（不包含实验自编代码），部分操作系统自带的软件包并没有包含。光盘中的软件工具都是从网络下载的开源软件，相关的版权信息请参考软件包内部的说明文件。光盘中的部分补丁文件由本书编者编写，版权属于北京大学。

本书由杨延军主编，其中第一章由段晓辉编写，第三章由王志军编写，第四章由赵建业编写，其余章节由杨延军编写，全书由杨延军统编。本书的编写还得到了顾瑶瑶、童霆、孟红玲、高凌翔、马晟厚、郭武等同学的帮助，很多选修了“嵌入式系统实验”课程的同学也为本书的编写提出

了宝贵的意见，在此对他们表示感谢。

本书的编写得到北京大学教材建设立项的支持，对此表示衷心感谢。同时感谢北京大学教务部和北京大学出版社的大力支持。本书编写过程中还参考了国内外相关参考书籍和兄弟院校的实践教学改革成果，在此特向所有关心支持本书编写、出版、发行工作的同仁致以诚挚的谢意。

由于作者的水平有限，时间仓促，书中难免出现不足和错误，敬请读者批评指正。

编者
于北京大学
2010 年 6 月

目　录

第一章　嵌入式系统概述

本章将介绍嵌入式系统最基本的概念，其中包括什么是嵌入式系统、嵌入式系统开发的基本模式以及本书所使用的软硬件环境。本章的目的是让读者对嵌入式系统有初步的认识，并了解本书所讲述的主要内容。

1.1　嵌入式系统的定义

要给嵌入式系统下准确的定义是很难的，因为嵌入式系统本身的界定就不是非常清晰，不同的人会对嵌入式系统有不同的定义。为了在本书中明确嵌入式系统的含义，假定可以把所有电子系统从简单到复杂进行排序，那么所得到的结果可能如表 1.1 所示。

表 1.1: 电子系统的排序

序号	核心元器件	示例
1	简单集成电路	电子门铃、稳压电源等
2	简单微处理器	智能玩具、洗衣机控制等
3	8 位微处理器	工业控制、电子仪表等
4	32 位微处理器	路由器、手机等
5	高性能处理器	微机、主干路由器等
6	多核 64 位处理器	科学计算、大型计算机等

本书认为一个电子系统可以被称作嵌入式系统必须具备如下几个条件：

1. 有处理器作为核心元器件

实践证明，处理器是一种非常高效的数字电路。通过合理的指令集设计，再配合优化的程序设计，处理器可以完成非常复杂的电路功能。而我们认为嵌入式系统就是一种复杂的电路系统，只能由处理器作为核心实现。

2. 软件系统以操作系统为核心

与硬件的情况类似，当软件复杂度提高之后，也必须对软件的功能进行抽象，而这种抽象的结果就是操作系统。操作系统的存在是对硬件系统（包括处理器和外设）的统一抽象，使得应用程序可以脱离硬件细节，在更高的层次上编写程序逻辑，既可以降低编程难度，也有利于软件的移植。

3. 不是普通微机和大型的服务器

这个条件的存在主要是微机和服务器平台已经非常成熟，硬件上有严格的产业标准，软件工具也非常丰富。使得它们与其他处理器组成的系统有非常大的区别，因此就把它们从嵌入式系统中排除出去了。不过这里需要注意服务器中的一些组件本身通常是一个小的嵌入式系统。

另外一个经常被引用的嵌入式系统的定义是：“嵌入式系统是以应用为中心，以计算机技术为基础，采用可裁剪的软硬件，适用于对功能、可靠性、成本、体积、功耗等有严格要求的专用计算机系统。”

这个定义很好地阐述了嵌入式系统的特点——专用计算机系统。嵌入式系统虽然包含一个处理器，但其中的程序被预先编制好，用户一般不能对其进行修改，甚至可能无法察觉它的存在。

典型的嵌入式系统会运行一个嵌入式的操作系统，硬件资源被操作系统调度，这也为嵌入式软件的开发和移植提供了非常大的方便。

经过上面的分析，可以看出符合本书对嵌入式系统定义的，就应该主要分布在表 1.1 中序号为 3 或 4 的范围之内。但这个范围的边界同样模糊：对于一些简单的单片机系统，可能并不包含操作系统在内，有时候却可以被称作 “深嵌入式系统”（Deep Embedded System）。另一方面，对于一些高端掌上设备，例如掌上电脑，虽然也经常被称为嵌入式系统，但由于它们可以运行多种不同的应用，更加接近一个通用的计算平台。

1.2 嵌入式处理器简介

作为嵌入式系统的硬件核心，嵌入式处理器的发展非常迅速，很多公司都推出了应用于不同环境的嵌入式处理器产品。从低端的 8 位和 16 位处理器，到高端的 32 位和 64 位处理器，嵌入式处理器的种类繁多，功能也千差万别。选择合适的处理器，往往是嵌入式系统设计的第一步。本节主要介绍几款应用非常广泛的中高端处理器。

1.2.1 MIPS

MIPS（microprocessor without interlocked pipeline stages）处理器是精简指令集计算机（reduced instruction set computer，RISC）处理器的先驱之一，由于其对指令集系统进行改进，去掉了有可能造成流水线等待的复杂指令，使处理器的整体性能有了很大的提高。由于其指令集设计得非常经典，很多计算机体系类的教材都采用 MIPS 的结构来介绍 RISC 系统。

在嵌入式领域，MIPS 处理器主要的应用范围有：机顶盒、游戏机（比如 Sony 的 PS 系列游戏机）、Cisco 的路由器等。其主要的特点体现在功耗低、工具丰富等。另外，由于最近很多厂商都提供 MIPS 的 IP 核，使得 MIPS 被更多地应用到嵌入式的环境中。

1.2.2 Power PC

Power PC 是 Apple、IBM 和 Motorola 三家著名的处理器提供商共同推出的基于 RISC 体系的高性能处理器，虽然最初的设计目的是抢占 Intel 的 PC 处理器市场，但由于其架构设计方便灵活，可伸缩性好，目前从高端工作站到嵌入式方面都有应用。

1.2.3 ARM

ARM（advanced RISC machine）是主要面向嵌入式领域设计的高性能处理器，其主要特点是体积小、功耗低、成本低。虽然 ARM 公司只对处理器的体系结构进行设计，并不实际生产芯片，但 ARM 在全球有众多的合作伙伴，很多公司都推出了面向不同应用的 ARM 处理器芯片。这使得 ARM 处理器在市场上具有很高的占有率，应用非常广泛。

1.2.4 DSP 系列

DSP（digital signal processing）是数字信号处理的简称，它们以其出色的数据处理能力，在嵌入式系统中的应用也非常广泛。在计算密集型的嵌入式系统中，DSP 芯片几乎是无法替代的。

在很多应用环境中，DSP 处理器还经常同其他嵌入式处理器组成双核处理器或者多核处理器，有些公司还推出了同时包含 DSP 和 ARM 核心的双核嵌入式芯片。如德州仪器（Texas Instrument，TI）公司的 OMap 系列处理器。

1.2.5 SoPC

SoPC(system on programmable chip)是在 SoC 的基础上提出的新概念。SoPC 技术是指在可编程芯片中实现一个处理器系统,软硬件同时在一个芯片中设计。大规模可编程芯片的出现为这种技术提供了可能。这种方式为软硬件协同设计提供了非常大的灵活度,整个嵌入式系统包括处理器都可以在一个平台上进行开发与仿真,很大地提高了嵌入式产品的开发效率。

很多可编程器件厂家提供了 SoPC 的解决方案,如 Altera 公司的 Nios 和 Xilinx 公司的 MicroBlaze。

1.3 嵌入式操作系统简介

嵌入式系统的软件核心是操作系统。与嵌入式处理器的多样性类似,嵌入式操作系统也有很多选择。应用比较普遍的嵌入式操作系统可以支持很多种不同的处理器体系结构,使得应用程序的移植更加容易,开发界面也比较统一。具体选择什么样的操作系统是由整个系统的各方面需求决定的,如成本因素、性能因素、资源占用情况等。以下介绍在嵌入式系统中应用比较多的几款操作系统。

1.3.1 Windows CE

Windows CE 是 Microsoft 公司开发的适用于嵌入式设备的操作系统。Windows CE 将 Windows API 应用到更小、更节省资源的操作系统上,提供平坦式的内存管理、抢占式的多任务和中断处理功能。Windows CE 主要应用于掌上型 PC 或手持式 PC。它的主要特点有以下几个方面:

友好的操作界面。提供和 Windows 类似的图形用户接口(graphic user interface,GUI)界面,可以很好地被 Windows 用户所接受。

方便的开发工具。Microsoft 提供了 Platform Builder 为 Windows CE 的内核进行定制,用 Embedded Visual C++ 在 PC 平台进行 Windows CE 的软件开发。由于开发界面以及应用程序接口和普通 Windows 开发非常类似,开发人员很容易掌握这些开发工具的使用。

网络和通信的支持。Windows CE 不仅支持 Winsock 编写常规的 Internet 应用程序,还支持部分 WinInet 库以便应用程序可以通过 HTTP 操作检索文档。在通信方面,Windows CE 支持标准的串口、红外接口、拨号网络串行线路网际协议(serial line internet protocol,SLIP) /点到点协议(point-to-point protocol,PPP)等功能。

国际化的支持。Windows CE 内部完全使用 Unicode 字符串,使得 Windows CE 可以很好地支持不同国家的语言界面,应用程序不用做特殊的处理就可以运行在不同的语言环境中。

数据库的支持。Windows CE 操作系统本身就提供了简单的数据库支持,它提供的 API 不能在其他版本的 Windows 中使用。Windows CE 的数据库只提供了一个层次、四种排列索引,但已经可以满足大多数普通用户的需要。

1.3.2 VxWorks

VxWorks 是 Wind River 公司(目前被 Intel 收购)开发的具有工业领导地位的高性能实时操作系统内核。VxWorks 从 1983 年设计成功以来,已经经过了广泛的验证,成功地应用于航天、航空、舰船、医疗、通信等关键领域。而且 VxWorks 得到了许多软硬件厂商的支持,拥有丰富的 VxWorks 扩展组件,很多平台都提供了基于 VxWorks 的板级支持包(board supporting package,BSP),使得应用程序移植的工作量大大减少。唯一可能让一般用户不选择 VxWorks 的原因是它和它的开发工具的销售价格,因此只有在比较高端和关键的产品中才选择 VxWorks。

VxWorks 的开发工具叫 Tornado,它包含三个高度集成的部分:一套强大的交叉开发工具;高性能、可裁剪的操作系统 VxWorks;连接宿主机和目标机的多种通信方式,包括以太网、串口、

ICE 或 ROM 仿真器等。Tornado 同时还支持另外一套强大的开发、调试工具，Wind Power。甚至第三方的一些工具也可以通过插件形式加入到 Tornado 框架中。

VxWorks 的主要特点如下：

强大的操作系统核心。VxWorks 的微内核 Wind 具有快速多任务切换、抢占式任务调度等特点，有很好的实时性。它的可裁剪组件很多，用户可以根据需要方便地通过交叉开发环境进行配制。

良好的兼容性。VxWorks 兼容 POSIX 1003.1b 标准，应用程序可以比较方便地在不同的运行环境间移植。

网络的实时优化。VxWorks 的 TCP/IP 协议在兼容 BSD4.4 的基础上，在实时性方面有较大的提高。因此基于 BSD4.4 UNIX 的应用程序不但可以方便地移植到 VxWorks 中，而且网络的实时性还可以提高。

优秀的开发工具。Tornado 在 Windows 平台上提供了一个很好的 VxWorks 开发、仿真工具，同样的界面可以应用于不同的开发平台，很大程度地提高了 VxWorks 的开发效率。

1.3.3 uC/OS

uC/OS 是源代码公开的实时嵌入式操作系统内核，其性能完全可以与商业产品竞争。自 1992 年问世以来，已经被成功地移植到了多种开发平台，在嵌入式系统开发中有很好的应用前景。

uC/OS 的最新版本是 uC/OS-II，它的主要特点如下：

公开源代码。和 Linux 的开放源代码类似，uC/OS 公开源代码的特性使得它的发展非常迅速，而且 uC/OS 的代码非常清晰易读，包含详尽的注释，可以让开发者真正了解内核的工作原理。

可移植性。uC/OS 的代码绝大部分是用 ANSI C 写的，汇编语言的使用被压缩到了最少。可以说 uC/OS 可以被移植到大部分的微处理器平台，特别是在低端系统 Linux 不能适用的环境下，uC/OS 依然可以发挥它的作用。

实时操作系统内核。uC/OS 是完全抢占式实时内核，支持多任务，系统调用的执行时间具有可确定性。

1.3.4 Linux

Linux 操作系统内核是 1991 年由芬兰学生 Linux Torvalds 首先公开发布的，它的发行遵循 GPL（GNU general public license）协议，在 Internet 上不断被发展和完善。由于 Linux 的开放源代码的特性，很多优秀的程序员加入到 Linux 的开发行列，使得 Linux 的发展非常迅速，成为今天无论在服务器平台还是在嵌入式平台都非常具有竞争力的操作系统。Linux 的成功不仅仅是 Linux 内核本身的功劳，还有与之配套的丰富的 GNU 工具，Linux 操作系统和这些工具不但功能非常强大，而且完全免费，遇到问题的时候还可以在相应的邮件列表上求助以得到全世界专家的迅速帮助。

简单归纳起来，Linux 操作系统的主要特点有以下几个方面：

良好的跨平台特性。Linux 已经被移植到多种硬件平台，支持 X86、Alpha、Sparc、MIPS、PPC、ARM 等现有大部分芯片，对于大部分嵌入式系统的开发只需要进行很少量的代码移植。GNU 工具的跨平台特性还允许在 PC 平台上为其他硬件平台进行交叉开发，提高开发效率。

GPL 的许可证协议使得任何人都可以免费获得 Linux 的源代码，并可以对其随意进行修改，以满足特殊的需要。这是 Linux 最吸引人的特性，它不仅大大降低了开发费用，也减少了开发周期。因为在世界范围，很可能有人因为同样的目的已经完成了类似的开发，并根据 GPL 的精神把他们的代码公开出来。

高度模块化的设计使得移植 Linux 到其他平台或者为 Linux 增加、减少、修改部件都变得相对容易。这也是 Linux 操作系统应用如此广泛的原因。目前网络上流行的针对 Linux 操作系

统的补丁非常多，适用于不同的平台和不同的应用目的，每种都有其独特的地方。例如有专门提高实时性的 RTLinux，为适应资源受限系统的 TinyLinux，和为低端不支持 MMU 处理器修改的 uCLinux 等。

具有强大的网络处理能力。如今很多嵌入式系统的开发是和网络、特别是 Internet 服务密切相关的。Linux 在 TCP/IP 的支持上恐怕是所有嵌入式操作系统中最好的了，完全可以胜任普通路由器的工作。

拥有极好的稳定性、高效性和安全性。Linux 从开发的最初就致力于提高性能和安全性，而性能对于嵌入式系统来说是至关重要的，在特殊的应用中，安全性也不容忽视。

硬件的支持非常广泛。Linux 提供了很好的驱动程序接口，同时开放源代码的特性也使得更多的人为 Linux 开发驱动程序。常见的硬件设备基本都可以被 Linux 操作系统支持，即使比较特殊的设备也很可能会有类似的驱动程序进行参考，从而加快驱动程序开发的速度。

广泛的文件系统支持。Linux 对文件系统的支持非常多，除了自己的 ext2/ext3 文件系统和常见的 MSDOS，VFAT，NTFS，网络文件系统（NFS）之外，对嵌入式设备更有特殊的支持：如专门为 flash 设备开发的 jffs2 和压缩的只读文件系统 cramfs。

丰富的应用程序支持。随着开放源代码社区的扩大，越来越多的应用程序被移植到了 Linux 系统中来，仅仅是图形环境就可以有 X、QT Embedded、DirectFB、MiniGUI 等多种选择。很多商用软件也逐渐开始支持 Linux，使 Linux 的应用前景更加广阔。

除此之外，Linux 本身兼容 POSIX 1003.1 标准，并包含 UNIX System V 和 BSD 4.3 的大部分特征。如多任务、多用户、多处理器的支持，虚拟内存支持，动态、静态共享库的支持，多种不同可执行文件格式的支持。所有这些都大大方便了应用程序的开发。

由于 Linux 的桌面应用还不广泛，主要由专业人员和一些爱好者在使用，因此同 Windows 相比，劣势主要在于难于上手，多数开发人员对于 Linux 系统还是不够熟悉。

1.4 本书所使用的软硬件环境

1.4.1 实验环境

嵌入式系统的开发主要在通用微机上进行，而嵌入式开发板主要作为开发的测试平台，对编写的代码进行检验和调试。典型的开发环境如图 1.1 所示：

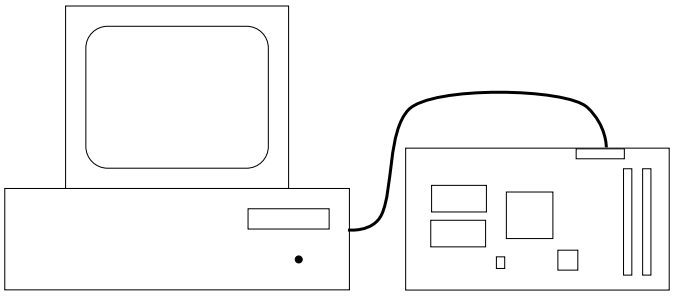

图 1.1: 嵌入式设备与开发主机的连接

嵌入式开发板需要同开发主机相连，连接的电缆一般有如下几种：

1. PC 与开发板的 JTAG 接口

JTAG 是很多数字电路芯片都包含的接口。它主要包括四根联络线：TDI、TDO、TCK 和 TMS。JTAG 接口最初的设计目的是对电路板上芯片管脚的连接情况进行检查，而如今它能完成的功能远不止如此。嵌入式开发中通常利用 JTAG 接口对嵌入式芯片内部或者外部的 flash 进行编程，也可以利用 JTAG 接口对程序进行调试。

图 1.2 是 JTAG 接口的原理示意图，其中 TRST 引脚是 JTAG 的内部复位信号。芯片内部每个管脚都包含一个 JTAG 单元互相连接形成串行移位寄存器。TDI 的串行数据可以按位移

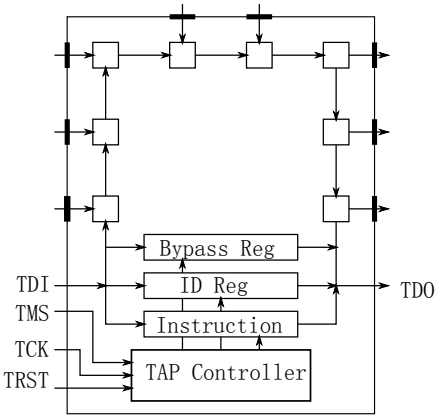

图 1.2: JTAG 结构示意图

到芯片的各个管脚,最终从 TDO 移出。通过这种方式,JTAG 逻辑可以对芯片的每个管脚的电平状态进行设置,也可以把管脚的电平状态读出。除此之外,JTAG 的状态机还可以接受不同的 JTAG 指令,实现芯片的特定操作。

2. PC 串口与开发板的串口

串口在嵌入式设备中是比较简单的接口,让串口正常工作相对比较容易,开发人员可以通过编写监控程序通过串口与开发主机进行通信,用来实现简单的调试功能。大多数嵌入式系统的软件都会通过串口输出其工作状态,甚至可以接收从串口输入的命令。通过一些工具软件,开发人员甚至可以通过串口对开发板上的程序进行指令级调试。

3. PC 网络口与开发板网络口

如果要与开发板传递大量的数据,串口的速度就捉襟见肘了。相对于串口来说,网络口的通信速度要快得多。传输一个 1M 大的文件,使用串口大约需要 10 分钟时间(波特率为 115200bps)而使用网络就只需要大约 10 秒钟(波特率为 10Mbps)。

本书使用的硬件平台主要有如下两个:一是 Intel 的 Sitsang 开发板,其主要参数如下:

- 处理器:XScale 核心的 PXA 255
- 存储器:64M flash、64M Sdram
- 外部接口:10M 以太网、2 个 USB 主设备、1 个 USB 从设备

另外一个是自主开发的 ARM 开发板,主要参数如下:

- 处理器:ATMEL 公司的 AT91SAM9261
- 存储器:64M NAND flash、64M Sdram
- 外部接口:100M 以太网、2 个 USB 主设备、2 个串口、1 个 USB 从设备

本书全部实验内容都在 Linux 操作系统下完成,嵌入式开发板上也主要运行 Linux 操作系统。对于没有硬件平台的读者,可以参考第九章对 QEMU 的介绍,使用模拟环境完成大部分的实验练习。本书所用到的部分软件版本如下:

台式 PC 系统(开发主机):

- 发行版:Ubuntu Linux 10.04
- Linux 内核:2.6.32
- Gcc 版本:4.4.3
- Glibc 版本:2.11.1

嵌入式开发板:

- Linux 内核:2.6.33
- 交叉 Gcc 版本:4.4.3
- 交叉 Glibc 版本:2.9

1.4.2 下载文件内容

本书全部实验中所需要软件工具及程序源代码的下载地址都可以向作者索取(电子邮箱地址:yangyj@pku.edu.cn)。其中操作系统与 Linux 发行版所提供的软件并没有包括。对于一些需要修改源代码才可以使用的软件(例如 QEMU),下载文件中也提供了其源代码与补丁文件。

1. Source 目录中包含了本书所涉及的大部分源代码,并根据源代码类型分成多个目录

Toolchain 子目录包含了生成工具链的源代码,其中包括 gcc、glibc、binutils 的源代码。这个目录中还有自动生成工具链的工具 crosstools-ng,不过这个软件在生成工具链的时候会自动从 Internet 下载需要的源代码。

Kernel 子目录包含了内核的源代码及本书所使用的内核补丁文件。

Graphic 子目录包含了 QTE 的源代码,tslib 的源代码。

QEMU 子目录包含了 QEMU 的源代码及本书所使用的补丁文件。

U-Boot 子目录包含了本书所使用的 U-Boot 源代码及补丁文件。

BusyBox 子目录包含了 BusyBox 软件的源代码。

2. PreBuild 目录中包含了预先编译好的工具,在实验中可以直接使用

Toolchain 子目录包含了 CodeSourcery 所提供的工具链及使用 crosstools-ng 编译的工具链,实验中可以选择任何一个使用。

Graphic 子目录包含了编译好的 QTE,用来交叉编译 Qt 源代码。它必须安装到/usr/local 目录中。

FileSystem 子目录包含了可以作为 NFS 根文件系统的压缩包。

Images 子目录包含了可以直接写入到本书所用开发板 flash 中的映像文件。其中 uImage 为内核映像,u-boot.bin 为 U-Boot 的映像,at91.jffs2 为文件系统的映像。

3. Tools 目录包含了本书使用的部分工具软件

SAM-BA 子目录包含了 Atmel 公司的 SAM-BA 软件及相关的映像文件及脚本。

4. Docs 目录包含了本书所使用及参考的一些电子文档

Datasheets 子目录包含本书所使用硬件的数据手册。

Sch 子目录包含了本书所使用硬件的原理图。

1.5 实验 :安装 Linux 操作系统

实验目的

1. 了解 Linux 的安装方法
2. 通过安装 Linux 加深对 Linux 系统的理解

实验原理

一、Linux 发行版简介

我们一般说的 Linux 操作系统实际上应该叫做 GNU/Linux 操作系统。因为 Linux 只是操作系统的内核,操作系统还必须包括一系列 GNU 工具才能被用户所使用。目前世界上有很多组织和公司都会定期或不定期推出不同版本的 Linux 与 GNU 工具的组合套装,方便用户的安装和使用,这就是 Linux 的发行版。

目前 Linux 的发行版非常多,很多都具有自己的特色和特定的使用人群。常见的发行版就有近百个,这里简单介绍几个最著名的发行版。

1. RedHat 系列

RedHat 公司是比较早期提供 Linux 发行版的组织,Redhat Linux 曾占有非常大的 Linux 市场份额。目前 Redhat 公司仍然在推出企业级的 Linux 解决方案和面向一般用户的 Fedora Core 发行版。他们最大的贡献就是提出了 rpm 格式的软件打包格式,使得软件的安装与维护变得简单。

2. Debian 系列

Debian GNU/Linux 是一款严格遵循 Unix 哲学和开源哲学的 Linux 发行版，它拥有庞大的软件库和方便的包管理方式，在世界范围也拥有非常多的拥护者。Debian 的软件包扩展名为 deb，是除了 rpm 格式外的另外一种应用很广泛的包管理格式。Debian 系统的 apt 工具使得软件包的维护和升级都十分方便。

最近一款非常受欢迎的发行版 Ubuntu 就是基于 Debian 系统构建的，但它的设计目标是人性化。它被称为是最容易使用的 Linux 发行版，因此推荐新手使用 Ubuntu 作为初次尝试 Linux 的选择。

3. Gentoo

Gentoo 发行版的历史并不长，它最大的特点在于其所有软件都可以通过源代码编译安装。这对于很多狂热的开源爱好者和希望对 Linux 系统有更深入了解的人来说都是最好的选择。通过完全从源代码编译安装 Linux，可以控制编译参数使得所安装的操作系统完全根据当前主机进行优化，也可以对操作系统进行最大程度的个性化。

4. 其他

其他可以选择的 Linux 发行版也很多，例如面向服务器的 CentOS、对中文支持较好的红旗 Linux 等。当用户对 Linux 系统掌握得很好的时候，就可以根据自己的需要选择合适自己的 Linux 发行版了。

二、Linux 系统的安装方法简介

随着近几年 Linux 系统在桌面应用上的不断发展，在 PC 机上安装 Linux 操作系统已经并不像几年前那么困难了。很多发行版都提供了很友好的安装界面，即使是第一次安装 Linux 的新手也可以很容易地安装成功。

在电脑上安装 Linux 操作系统最简单的方法是使用一个发行版的安装光盘。大多数发行版都提供了图形化的安装界面，只要使用光盘启动电脑，就可以按照安装程序的提示一步步进行安装了。考虑到大多数电脑都安装了 Windows 操作系统，而且不想失去原有的 Windows 程序，所以在安装 Linux 之前首先要考虑的是如何让 Linux 和 Windows 系统共存。

方法一：在虚拟机中安装 Linux 操作系统。

VirtualBox 是一个虚拟机软件，它可以从 www.virtualbox.org 上免费获得。这个软件可以模拟出完整的计算机硬件，用户可以在这个虚拟的环境中安装 Linux 操作系统。为了区分这两个操作系统，在虚拟机中运行的被称为“客户系统”，与之对应在真实环境中运行的操作系统叫做“主系统”。客户系统相对于主系统来说只是一个应用程序，而在客户系统中运行的程序一般是无法知道自身是运行在虚拟系统之上的。在 VirtualBox 中安装 Linux 是最简便和安全的方法，安装之前只需要在原 Windows 系统中安装一个 VirtualBox 软件，建议初学者使用这个方法。VirtualBox 中的 Linux 和一个实际的 Linux 没有什么区别，只是运行的效率会略有下降，安装和使用的时候可以随意测试，而不用担心损坏主系统。

方法二：使用光盘或者 USB 盘上运行的 Live 系统

很多发行版都提供了 Live CD 作为一种既可以体验 Linux 操作系统，又不必对计算机做任何修改的安装方式。只要把 Live CD 放在光驱中并将计算机设为光驱启动，就可以启动到 Linux 操作系统。这个系统完全从光盘上运行，并不操作主机的硬盘，所以对主机没有特殊的要求，也不用担心损坏数据。只是在这个系统中，用户对系统的修改不能永久地保存起来。一种解决办法就是把 Live CD 安装到 USB 盘上，通过特殊的文件系统技术，可以实现对用户修改的保存。

方法三：直接把 Linux 安装到硬盘。

如果要把 Linux 真正安装到硬盘上与 Windows 同等的存在，就必须为 Linux 的安装准备一个空闲的硬盘分区，这个工作可以用 Windows 的磁盘管理工具完成，也可以在安装 Linux 的过程中进行。安装系统的时候必须把 Linux 安装到独立的硬盘分区，安装完成后，Linux 一般会

安装一个启动管理器到硬盘的 MBR 区，可以通过这个启动管理器在系统启动的时候选择是进入 Linux 系统还是进入原来的 Windows 系统。

仅仅安装 Linux 操作系统可能会需要 1GB 的硬盘空间，但为了更好地使用和完成后续的实验，最好准备 5GB 以上的空间。一般情况下 Linux 操作系统会占用多个硬盘分区，分区可以是文件系统的一个目录，也可以作为虚拟内存使用。虚拟内存并不是必须的，但为了可以运行更需要内存的应用，最好还是建立一个存储空间为物理内存 2 倍的虚拟内存分区。虚拟内存分区在 Linux 系统中被称为交换分区（swap partition）。

三、Linux 系统中硬盘的分区方法

在 Linux 系统中，几乎所有的硬件都通过/dev 目录下的文件来标识，硬盘的分区有特定的表示办法。对于 PC 平台，一般可以支持 4 个 IDE 设备，在 Linux 系统下被分别称为 had、hdb、hdc 和 hdd。但由于目前最新的 Linux 系统都采用 SCSI 仿真的方式来访问 IDE 设备，所以在较新内核的 Linux 系统中，4 个 IDE 设备被称作 sda、sdb、sdc 和 sdd。而每个硬盘最多有 4 个主分区，它们被分别编号为 1~4，例如对 sda 硬盘的 4 个主分区会分别编号为 sda1、sda2、sda3 和 sda4。而如果主分区的类型是扩展分区，则在这个扩展分区中还可以继续分成许多逻辑分区。Linux 系统中，每个硬盘的逻辑分区从 5 开始编号，也就是说 sda5 就代表硬盘的第一个逻辑分区，sda6 代表第二个，如此类推。

例如一个用 Windows 操作系统分区的硬盘，包含 “C、D、E” 三个分区，在 Linux 操作系统下看就应该是：

```
/dev/sda1          第一个主分区，Windows 下的 C 盘。
/dev/sda2          第二个主分区，硬盘的扩展分区，包括 Windows 下的 C、D 盘。
/dev/sda5          第一个扩展分区，Windows 下的 D 盘。
/dev/sda6          第二个扩展分区，Windows 下的 E 盘。
```

除了前面介绍的交换分区，Linux 系统需要把所有文件保存在硬盘的某个或多个分区内。在安装 Linux 的过程中，如果采用手动分区的方式，安装软件会询问每个分配给 Linux 系统的硬盘分区的挂载点。所谓挂载点就是 Linux 文件系统的某个目录。例如 “/” 代表文件系统的最顶层，把 “/” 挂载到 “/dev/sda3” 就是把最顶层的文件系统放置到第一个硬盘的第三个主分区内。

最简单的 Linux 系统分区方式就是只给 Linux 一个分区，并把 “/” 挂载到这个分区上。其他常见的还可以挂载到单独分区的挂载点有：

```
/home                  一般用户的个人目录
/boot                  系统启动目录
/var                   可放置服务程序的数据文件、日志文件等
```

把哪些目录挂载到单独分区要根据自己的需求决定，例如为了在重装 Linux 操作系统的时候能保留个人数据，设置单独的 “/home” 分区就非常方便。

实验内容

一、安装 Linux 操作系统到计算机

可以采用任何的发行版安装 Linux 操作系统，这里使用 Ubuntu 10.04 发行版，以在 VirtualBox 3.2.2 中安装作为示例。

第一步：下载 Ubuntu 的安装映像文件：ubuntu-10.04-desktop-i386.iso，这个文件在虚拟机中使用 “虚拟介质管理” 工具可以注册为虚拟光盘，在虚拟机中使用。如果要在真正的 PC 中安装 Linux 就必须把它真正刻录到光盘上。

第二步：在 VirtualBox 中新建一个虚拟机，选择 Linux 操作系统，一切设置可以接受缺省值，虚拟机建立后在设置中将光驱介质设置为下载的映像文件。

第三步：启动新建的虚拟机，它将通过 “光驱” 启动，经过一段时间会显示出 Ubuntu 的安装界面。界面上可以选择安装过程使用的语言，并包含一个菜单：

```
Try Ubuntu 10.04 LTS
Install Ubuntu 10.04 LTS
```

选择 “Install Ubuntu 10.04 LTS” 开始安装系统。选择 “Try Ubuntu 10.04 LTS” 就可以启动一个光盘版的 Ubuntu，用户可以利用这个光盘先对 Linux 作初步的体验，最后双击桌面上的 Install 图标开始安装系统。

第四步：Ubuntu 的安装过程非常简单，按照屏幕的提示选择安装语言、所在时区、键盘类型，然后是对硬盘进行分区；对于虚拟机，可以选择第一个选项自动分区。如果是在真实的计算机上安装，也可以选择自动分区，但这样就会把整个硬盘清空，原来的操作系统和数据就都没有了。最后输入用户登录信息就可以开始拷贝文件了。图 1.3 是安装前的最后一个界面，显示了前面进行的基本设置，如果没有什么需要修改的就按 “Install” 开始安装。

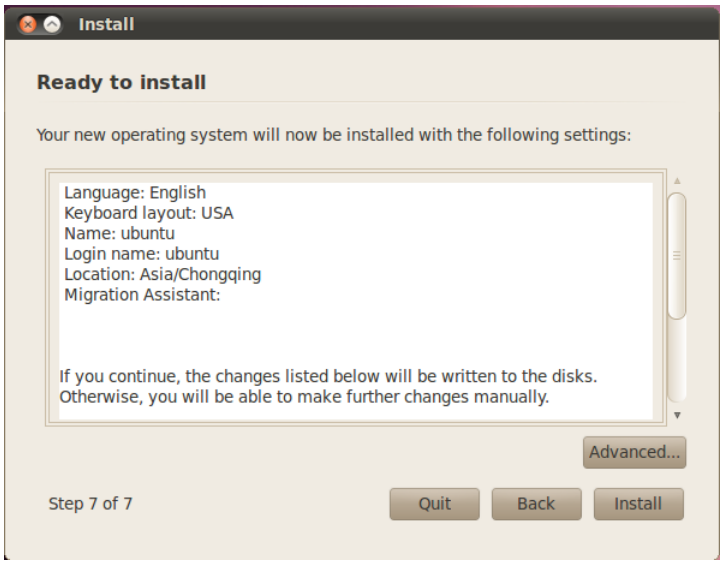

图 1.3: 安装确认界面

第五步：安装完毕后重新启动虚拟机系统就可以进入 Linux 系统了。

二、对系统进行最初的设置

在较新版本的 Linux 发行版中，安装完系统一般都会进入到图形界面，但在我们的实验中，经常要输入文字命令。一种方法是在图形界面下运行虚拟终端（Terminal）程序，在这个程序中输入命令，另外一种方法是切换到文本登录界面，在文本界面下输入命令。一般情况下可以按 “Ctrl+Alt+Fn” 组合键（n 可以是 1 到 6）切换到文本界面，再按 “Alt+F7” 组合键回到图形界面。

Ubuntu 为了提高安全性，默认禁止 root 用户登录，由于没有设置 root 用户的密码，也不能通过 su 命令切换到 root，一切需要超级用户的操作都通过 sudo 命令实现。如果需要多次使用 sudo 命令，可以用 sudo 的 `-s` 参数，获得具有 root 权限的 shell（即交互界面）。

```
$ sudo -s
#
```

上面的命令中 “$” 为命令提示符，实际输入的命令是后面的部分。不同的系统设置得到的命令提示符可能不同，但一般都是以 “$” 结尾，“$” 符合前面可能有用户名和当前路径的信息。本书中所有命令都是以 “$” 或者 “#” 作为命令提示符，其中前者代表普通用户可以使用的命令，后者代表必须由 “root” 用户输入的命令。上面示例的第一行为输入的命令 `sudo -s`，经过密码验证后获得了 root 用户权限，因此提示符变成了 “#”。

如果用户更喜欢直接使用 root 用户，需要执行下面的操作为 root 用户设置可用的密码：

```
$ sudo passwd root
```

这条命令首先需要输入登录用户的密码，对用户身份进行验证，然后以 root 用户身份执行 passwd 命令，就可以为 root 用户设置一个新的密码，这样就可以用 root 用户登录了。

为了在 Ubuntu 环境中进行程序开发，还必须安装一些开发相关的程序包，可以用下面的命令安装：

```
# aptitude install build-essential
```

这条命令的提示符为 “#”，表示必须由 root 用户执行，因为给系统安装软件是特权行为。aptitude 是 Ubuntu 中的包管理工具，它会从网上下载需要的软件包进行安装。实际上 aptitude 基于 apt 系列命令，上面的命令也可以通过下面等效的命令实现：

```
# apt-get install build-essential
```

由于 aptitude 是 apt 系列命令的高层封装，因此使用上更加方便，对于可能的软件包冲突等事件可以更好地处理。

软件管理工具的一个配置文件是/etc/apt/sources.list，用来定义下载软件包的源地址。对于 Ubuntu 10.04 来说，这个文件可能包含如下行：

```
deb cdrom:[Ubuntu 10.04 ...]/ lucid main restricted
deb http://archive.ubuntu.com/ubuntu lucid main restricted
deb-src http://archive.ubuntu.com/ubuntu lucid main restricted
```

sources.list 文件的语法结构是：

```
    deb uri distribution components
```

上面例子中第一行表示安装包的源地址为光盘，如果网络条件好的话可以删除。第二行定义了一个 Ubuntu 源的地址，其中 http://archive.ubuntu.com/ubuntu 为下载路径，为发行版的名字，后面的是 main restricted 是软件的类型。如果想选择一个更快的 Ubuntu 源的话，就可以更新 uri 的值。第三行所代表的是源代码的下载路径，因为 Ubuntu 包含的大多数是开源软件，如果需要用 apt 命令下载源代码的话就必须有 deb-src 的定义。deb-src 定义的语法与 deb 完全按相同。

aptitude 命令除了 install 命令之外，还支持多种软件包维护命令，下面是一个不完全的命令列表：

remove	卸载软件包
purge	卸载软件包并删除其配置文件
update	下载新/可升级软件包列表
full-upgrade	执行升级，可能会安装和卸载软件包
search	按名称或表达式搜索软件包
show	显示一个软件包的详细信息
clean	删除已下载的软件包文件

例如用 aptitude 命令查找一个名字为 cscope 的软件：

```
$ aptitude search cscope
p   cscope    - Interactively examine a C program source
```

命令的输出列出所有包含关键字 cscope 的软件包，这里只列出了一个。最前面的 “p” 表示软件还没有安装，对于已经安装的软件，这个位置显示的为 “i”。“-” 号后面的部分是软件包的简单描述，要看具体的软件信息可以用 aptitude 的 show 命令。

```
$ aptitude show cscope
```

这个命令输出的信息比较多，基本包含了用户需要知道的一切有关这个软件的信息。

还有一些常用的软件包也可以在刚安装完系统的时候安装，例如：

```
# aptitude install bison flex automake libtool
```

当在使用一个软件或者编译一个程序的时候出现找不到某个软件或者文件的错误，就可以首先用 aptitude search 命令查找包含这个文件的软件包名字，然后用 aptitude install 命令进行安装。一般 C 语言头文件会包含在 `-dev` 的软件包中。例如编译程序的时候说找不到 ncurses.h 文件，由于 ncurses 是一个 C 语言开发包，所以相关的软件包的名字应该类似于 libncurses-dev。通过执行

```
$ aptitude search ncurses
```

可以查找到很多相关的包，通过对比文件名得知需要安装 `libncurses5-dev`，其中 “5” 是软件的版本号。

第二章　Linux 作为开发平台

经过了十多年的发展，Linux 已经成为一个非常成熟的操作系统，而不仅仅是一个开发环境。它包含了丰富的应用程序，完全可以满足日常办公和娱乐的需求，而了解它的最好方式就是把 Linux 安装到计算机上，亲自去操作和体验。

本章将介绍 Linux 操作系统的基本命令行界面，并对 Linux 系统下的程序设计环境和工具进行简要介绍。

2.1　Linux 操作系统入门

目前流行的 Linux 发行版非常多，Ubuntu、Fedora、Core、Cent OS 和 Debian 等都是很好的选择。但 Linux 毕竟不是 Windows，虽然图形界面的操作感有些类似，其设计理念是完全不同的，下面几个概念需要特别注意：

1. root 用户具有最高和最“危险”的权限。Linux 是一个真正的多用户操作系统，它不同于 Windows 的地方在于它的权限管理非常严格，使用普通的账户可以完成日常一般的工作而不会对系统的配置造成破坏或者改变，如果要对系统的配置进行修改就要使用超级用户。一般一个系统只有 root 才具有这个最高权限，而且系统对 root 用户的操作几乎没有任何限制，所以使用这个用户就变得非常“危险”。即使是有经验的用户也有可能因为失误造成无法挽回的损失，因此在使用 Linux 的时候尽量避免使用 root 账户，只有在需要的时候才用 su 命令切换到 root 用户工作，并在工作的时候仔细检查每条输入的命令，工作完毕后退出 su 状态。
2. Linux 系统是高度可配置性的。在 Linux 系统中，无论是 Linux 内核，还是应用程序，一切都可以重新配置。特别是对于嵌入式平台，系统的配置方式可能千差万别。Linux 的灵活引入了更多的复杂性，但对于开发人员来说，充分了解你使用的工具是非常必要的。每个软件都会有相应的说明文档和配置文件，当你需要了解这个工具的时候，读读文档是很好的帮助。虽然 Linux 下的应用软件非常多，但只要掌握了最基本的原理，新的软件或者系统也会很容易上手。

由于开发使用 Linux 的时候，多为使用 Linux 的文字界面，即使是使用图形界面登录，我们也常常打开一个称为“虚拟终端”的程序来输入文字命令，因为这样方式的工作更高效，也更具有普遍意义。所以下面将要介绍 Linux 系统中最常用的命令，这些命令几乎每次工作都要用到，它们的功能可能很强大，这里只介绍最简单和常用的用法，力求尽快地把读者带入 Linux 的大门。

2.1.1　文件操作命令

在 Linux 操作系统中，几乎所有东西都是文件，文件可以代表目录，可以代表系统中的设备，可以代表进程间通信的管道或 Socket。文件名的长度没有直接限制，可以使用除了“/”外的任何字符。文件的扩展名没有实际含义，仅供用户使用方便。Linux 与 Windows 操作系统不同的一个特点是文件名需要区分大小写，也就是在同一个目录内可以包含 README 和 readme 两个文件，而这在 Windows 下是不允许的。这个特点在与 Windows 系统共享文件的时候可能会造成问题。例如这样的目录在 Linux 被压缩后，如果在 Windows 下解压就会使得同名的文件只剩下一个。

1. ls：文件列表（list）

Linux 下很多命令都是由两个字母组成，这可以提高键入的效率。当我们知道它们是哪个英文单词的缩写的时候就可以很容易地记住。比如这个 ls 就是 list 这个单词的缩写。

ls 命令用来显示文件列表，它的可用参数很多。Linux 下命令的参数一般是一个减号加一个字母。大部分参数还有相对的“长格式”，长格式的参数以两个减号开始，减号后面往往是比较长的完整单词或者短语。长格式参数的目的是增加可读性和便于记忆。ls 的常用命令参数介绍下面几个：ls -l：显示详细的文件信息，其中 -l 代表“long”，表示采用“长”格式来显示文件信息（不过“long”并不是 -l 的长格式）。例如：

```
$ ls -l
total 8
-rw-r--r-- 1 embedded users 1509 05-16 15:56 sound.c
drwxr-xr-x 3 embedded users 4096 2005-11-01 text
... ...
```

上面的命令列表中，“$”代表命令提示符，提示符后面为输入的命令，命令的输出显示在命令的下方。

这个命令输出内容的最上面一行为当前目录中的文件个数，再下面的信息是每个文件的基本资料。格式中从左到右包含文件的类型、链接数、所属用户和组（上面例子中，文件属于 embedded 用户，users 组）、文件大小、文件修改日期与文件名。在文件类型部分，开头的字母有多种可能，其中：

- “-”代表普通文件；
- “d”代表目录文件；
- “l”表示符号链接（类似 Windows 下的快捷方式）。

类型部分还包含了权限控制位，它共有九位，每三位一组，三组分别代表了文件所有者、文件所在组和其他用户对此文件的访问权限。每组的三位按顺序分别代表了文件的读（r）、写（w）和执行（x）权限。有相应的权限就用相应的字母表示，否则用“-”号表示没有这个权限。例如 -rwxr-xr-- 代表文件所有者可以读写和执行这个文件（rwx），同组成员可以读和执行（r-x），但其他用户就只能读这个文件（r--）。

需要注意的是，并不是任何权限组合都是有效的。例如具有 x 权限但不具有 r 权限是没有意义的，用户必须具有文件的读权限才可以执行它。对于目录来说就更复杂一些，具有 r 权限的目录就可以列出其中的内容，但必须同时具有 x 权限才可以进入这个目录。如果不具备目录的写权限就无法对目录中的文件进行增加、删除或者改名。如果拥有目录的写权限，即使不具备其内部文件的写权限，也可以将其删除。

默认情况下，ls 命令不显示文件名以小数点开始的文件，Linux 文件系统中以“.”开头的文件代表“隐藏文件”，如果要显示所有的文件，可以用 ls 的“-a”参数。

ls 命令还可以对文件列表进行排序，例如 ls -S 命令根据文件的大小进行排序，ls -t 命令根据文件的修改时间排序。

如果把目录名作为 ls 命令的参数，ls 将输出这个目录中包含的所有文件。但如果用户需要查看这个目录本身的信息，就需要使用 -d 参数：

```
$ ls -dl test
drwxrwx--- 1 root root 0 01-11 14:54 test
```

文件权限的改变可以使用 chmod 命令，这个命令支持两种输入格式，一种是八进制格式，就是把三位的权限位当作二进制数，“-”代表 0，否则代表 1。例如“-rwxr-xr--”就是 654，命令的格式如下：

```
$ chmod 654 file_name
```

chmod 的另外一种输入格式是字母格式，用 g 代表组，u 代表所有者，o 代表其他用户，a 代表全部的权限位。下面是几个例子：

- chmod g+r file：表示组用户增加读权限；
- chmod a-x file：表示所有用户取消执行权限。

文件所属的用户和组可以使用 chown 命令改变，但这个命令只能用 root 用户执行，其格式如下：

```
chown user:group file_name
```

如果 file_name 是个目录，并且希望可以把目录中的全部文件都改变属主，就可以使用 “-R” 参数。这个参数代表 Recursive，即递归执行。

2. cd：进入目录（change directory）

cd 的参数为目录名，也叫路径名，当进入一个目录的时候，这个目录就成为当前的工作目录，或称为 “当前目录”。很多文件操作命令的默认目录就是当前目录。例如不带参数的 ls 命令就是显示当前目录中的文件。

用 “ls -a” 命令可以看到两个特殊的目录：“.” 和 “..”，它们分别代表当前目录和上一级目录（实际上它们都是硬链接，可以参考后面 ln 命令的介绍）。所以要进入上一级目录就可以使用 “cd ..”（注意 “.” 和 cd 之间要有空格）。

不带参数的 cd 命令表示进入当前用户的主目录（home directory），用户的主目录是在建立用户的时候指定的，一般在/home/ 目录下，与登录名同名。例如用户的登录名是 bgates，他的主目录就可能是 “/home/bgates”，这个目录在目录名中可以用 ~ 代替。例如 “~/work/” 就代表 “/home/bgates/work”。

如果想知道当前的所在目录，就可以用 “pwd”（present working directory）命令，这条命令可以把当前的工作目录显示出来。

当使用一个目录名作为命令参数的时候，既可以使用绝对目录，也可以使用相对目录。绝对目录以 “/” 开头，代表从文件系统的最根部开始表示，例如上面例子中的 “/home”。相对目录不以 “/” 开头，表示省略了绝对目录中的 “当前目录” 部分。例如当前目录为 “/home/bgates/” 时，相对目录可写为 “work”，代表的绝对目录是 “/home/bgates/work”。

另外一个常用的形式是 “cd -”，这个命令表示进入上一个工作目录，当需要在两个工作目录不断切换的时候这个命令就非常有用了。

3. mv：移动文件（move）

把一个文件从一个位置移动到另外一个位置。第一个参数是被移动的文件，第二个参数是移动到的位置。例如：

```
$ mv foo ~
```

这里使用了前面介绍的 ~ 代表主目录。如果 mv 的第二个参数是一个文件名，就相当于把这个文件改名，或者说把一个文件 “移动” 到另外的名字所指定的位置去了。

4. cp：复制文件（copy）

cp 命令的参数顺序同 mv 命令一样，它用来把一个文件复制到另外一个位置。默认情况这个命令不复制目录。如果要复制整个目录就可以用 “-r” 参数。例如 “cp /home/user1 /home/user2 -r”。更加 “强大” 的是 “-a” 参数，用它不但可以复制子目录，还可以保留所复制文件的相关属性，使用起来更加方便。

5. rm：删除文件（remove）

rm 命令可以把作为参数的文件从文件系统中删除。如果要删除目录中的内容，则需要加上 “-r” 参数。实际比较常用的命令参数组合是 “rm -rf”，其中 f 参数表示删除每个文件的时候不需要用户确认，直接立即删除。所以使用 “rm -rf” 命令的时候一定要小心，免得误删文件。从这个命令我们还可以看出当多个参数组合在一起的时候可以不必写成 “rm -r -f”，而把两个参数字母组合成一个参数。

6. mkdir：新建目录（make dir）

mkdir 命令的参数是新建目录的名字，可以同时建立多个新目录。但如果新建的目录包含好多层的时候就要使用 “-p” 命令。例如：“mkdir -p test/work/one”。

7. rmdir：删除目录（remove dir）

这个命令比较简单，只要把要删除的目录名作为参数就可以了。但这条命令只能删除空目录，删除非空目录可以用 “rm -rf” 命令。

8. df：显示磁盘的占用情况（disk free）

直接运行 df 命令就可以看到文件系统中每个挂载点的空间利用率。不加参数使用时，显示磁盘空间的单位是 “512 字节的块”；使用 “-h” 参数可以用 K、M 或 G 作为单位，增加输出的可读性。例如：

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda1             5.8G  4.4G  1.1G  81% /
udev                  244M  224K  244M   1% /dev
/dev/hdd5              18G   16G  543M  97% /home
/dev/hdd2             6.9G  5.3G  1.7G  76% /usr/local
```

9. du：显示目录和文件的空间占用（disk usage）

du 命令默认显示当前目录及其子目录所占用的空间，常用的参数是 “-sh”，其中 “-h” 参数与 df 名的 h 参数含义相同；“-s” 参数表示仅计算指定的目录。如果使用了 s 参数而没有指定目录的话代表当前目录。例如：

```
$ du -sh
45M     .
```

10. cat：显示文本文件的内容（concatenate）

查看一个文件内容最简单的方法就是使用 cat 命令。它把文件的内容输出到标准输出设备上（一般是屏幕）。对于比较长的文件，可以使用 more 或者 less 命令，这两个命令都支持在文件中浏览。more 命令比较简单，当输出内容显示满屏的时候就停止，按空格继续显示一屏，按 “q” 键退出程序。less 命令支持的功能更多，可以支持反向的字符串查询。如果需要更多操作可以使用后面介绍的编辑器 vi。

与显示相关的命令还有 tail 和 head，它们分别用来显示文件的最后几行和开头几行，使用 “-n” 参数可以指定显示的行数。这两个命令的优点就是简单，运行的效率高。例如用 tail 命令可以方便地查看日志文件的最新记录。head 命令可以用来查看程序文件前面的说明文本。

11. ln：建立文件的链接（link）

这个命令的格式是

```
ln [option] source [dest]
```

它为源文件建立了一个链接。不加参数的 ln 命令建立源文件的 “硬链接”（hard link）s；与之对应的是 “软链接”（soft link），也被称为 “符号链接”（symbolic link），在 ln 命令中使用 “-s” 参数建立符号链接。

硬链接直接与文件的实际内容相连，建立之后与源文件没有区别，删除了源文件后，硬链接仍然可以使用。前面介绍 ls 命令的时候，用 -l 参数所显示的 “文件链接数” 指的就是所有硬链接的个数，只有文件所有的硬链接都被删除了，文件内容才真正被删除。符号链接只是对源文件名的链接，如果删除了源文件，符号链接就 “断链” 了。

实际使用中硬链接不能做目录的链接（前面提到的 “.” 和 “..” 是特殊情况），不能做跨越文件系统的链接，也不能做不存在文件的链接，所以符号链接应用更多。

其实 Linux 操作系统中的多数命令都是一个可执行程序，它可以接受命令行的参数来执行不同的功能。Linux 下的大部分命令都接受 “–help” 这个参数（注意 “help” 的前面是两个减号）。这个参数的功能是在文字终端上打印一个简略的说明，告诉用户这个命令的基本用法。例如执行：

```
$ ls --help
```

就可以得到 ls 命令的简单帮助，如果需要得到更详细的帮助文档还可以使用 man 命令。

2.1.2 man 命令

man 是 manual 的缩写，这个命令用来查看其他命令或者函数的“手册”。例如想查找 mv 命令的用法就可以执行：

```
$ man mv
```

有时候一个命令的手册很长，在 man 的界面中还可以使用一些简单的查找功能。例如按下“/”键就可以输入一个关键字在手册中查找；按“n”键来查找下一个匹配等。详细的用法可以通过

```
$ man man
```

命令来查看 man 命令的手册。man 命令的手册分为 9 个部分，每个部分是对不同类型命令的说明：

1. 可执行程序和 shell 命令
2. Linux 系统调用
3. C 函数库
4. 特殊文件（例如/dev 目录中的文件）
5. 系统配置文件格式
6. 游戏
7. 其他杂项
8. 系统管理命令
9. 内核函数调用

例如 open 既是一个命令也是一个 C 函数。如果直接使用“man open”就只能看到排列在第一部分的作为命令的 open。如果要查看作为库函数的 open 就必须把 3 作为参数传递给 man：

```
$ man 3 open
```

除了 man 命令，在 Linux 还可以使用 info 命令来查看命令的说明。info 命令往往可以提供比 man 命令更多的信息，而且它的格式化更好，支持简单的文档间跳转，好像在用浏览器查看 html 文件。但是 info 的文档并没有像 man 那么普及，很多的命令仅提供了 man 的文档。

2.1.3 vi 命令

vi 是所有 Linux 发行版都提供的一个文本编辑软件。但 vi 和 Windows 下的任何其他编辑器都不类似，初次接触很可能完全无法了解它的用法。但在某些特殊环境，如嵌入式系统中，也许 vi 就是唯一的选择。另外，vi 是所有 Linux 和 UNIX 都包括的命令，掌握了它的用法就可以在所有 Linux 系统中进行编辑了。

vim 是 vi 的“超集”，它提供更加丰富的功能，更加适合编程的使用。扩展的功能包括语法高亮、多级撤销、可视选择等。大多数 Linux 发行版提供的都是 vim。vim 的另一个扩展是 gvim，它提供了图形化的工作前端，可以使用鼠标和操作菜单。执行

```
$ vi file_name
```

进入 vi 的界面。vi 是一个有“模式”的编辑器，它具有“命令”、“命令行”和“编辑”三种主要模式。刚打开的 vi 处于命令状态。按“i”键进入插入状态就可以进行文字编辑了。编辑完成后按“Esc”键则可以返回命令状态，在命令状态按“:”键进入命令行状态，在命令行状态输入 wq< 回车 >（write and quit）就可以存盘退出。

vi 的命令状态主要是通过按键完成特定的功能，例如：

- x：删除光标位置的字符
- dd（连续按两次 d 键）：是删除并保存当前行到缓冲区
- p：将缓冲区中内容输出到光标位置。
- yy: 保存当前行到缓冲区
- u：取消上一次操作
- h，l，j，k：左、右、下、上移动光标

提供字母键作为移动光标的主要原因有两个，一是为了在不提供方向键的终端上使用 vi，另一个原因是按这些按键的时候不必过多地移动手指，提高操作的效率。

上面提到的命令都要注意区分大小写，它们前面还可以加数字前缀，例如：

```
20j
```

就是下移 20 行光标。

在命令状态按 “:” 进入命令行状态，输入的 “:” 在窗口的最低端显示，如下面的例子所示。删除这个输入的冒号可以退出命令行状态。这个状态用来输入比较复杂的命令，例如查找替换：

```
:s/One/1/g
```

s 命令代表替换。第一个 “/” 号后面的是查找内容，第二个 “/” 后面是替换内容，第三个 “/” 后面是一个标志，g 代表替换本行的所有目标。

vi 的查找命令支持正则表达式，参考后面介绍 grep 的部分，表 2.1 中列举的正则表达式在 vi 中同样支持。

在命令行状态的一个重要功能就是帮助，使用

```
:help :s
```

就可以查看上面介绍的 s 命令的说明。在查看帮助的时候，如果有其他条目的链接，则可以按 “Ctrl+]” 组合键跳转过去，然后按 “Ctrl+T” 组合键返回。

到目前为止已经介绍过好多 “求助” 的方法了。主要目的是告诉大家如何利用 Linux 来自己帮助自己，这是加快掌握 Linux 的捷径。因为没有任何一本书可以包揽 Linux 的所有功能和用法，只有在遇到难题的时候学会如何自己解决，才能真正掌握 Linux 的精髓。

2.1.4 bash 相关命令与技巧

在 Linux 系统中，响应用户输入指令的程序被称为 shell。比较常用的 shell 有 csh、ksh 和 bash。由于多数发行版的默认 shell 都是 bash，这里就简单介绍一下它的操作特点。

1. 命令补全

命令补全功能可以大大提高在命令行工作的效率，当输入一个比较长的命令的时候，用户可以只输入这个命令的开头几个字母，然后按 <Tab> 键。如果当前系统可以搜索到的命令中以这几个字母开头的只有一个，系统就会自动把这个命令补全；否则再次输入 <Tab> 键可以显示出所有满足条件的命令，用户可以继续输入字母直到满足的命令只有一个的时候再按 <Tab> 键补全。

命令补全功能还适用于目录名和文件名的补全，在其他软件的辅助下还可以支持命令行参数的补全。例如当前目录中只有一个目录，用户可以执行

```
$ cd <Tab>
```

就可以把目录名补全。

2. 历史功能

bash 会自动记录用户输入的所有命令，用 history 命令可以查看输入命令的历史。默认情况下，bash 会记录最近输入的 500 条命令。

在命令提示符下，按 “上” 方向键可以调出以前输入的命令，也可以用 “下” 方向键回到之后输入的命令。

3. 环境变量

环境变量相当于 shell 的程序变量，用户可以对它们进行修改，来影响 shell 或者影响 shell 启动的程序。习惯上，环境变量的名字都采用大写字母，使用 env 命令可以查看当前 shell 中的所有环境变量。使用环境变量的时候在它的名字前面加 “$” 符号，例如：

```
$ cd $HOME
```

就可以进入 HOME 环境变量指定的目录。设置环境变量可以直接用等号赋值，如：

```
$ DEBUG=3
```

但这样设置的环境变量只能被 shell 本身使用，并不能影响在当前 shell 下执行的程序。例如有如下的脚本[1]程序：

```
#!/bin/sh
echo $DEBUG
```

则执行这个脚本获得的结果仍然是空值，也就是设置的 DEBUG 环境变量并没有对这个程序起作用。如果要让设置的环境变量能影响到 shell 所执行的程序，就要把这个变量 “导出”，用 export 命令。例如：

```
$ export DEBUG
```

我们还可以把变量的赋值与导出写在一个语句中：

```
$ export DEBUG=3
```

如果直接运行 export 命令而不使用任何参数，就可以查看当前 shell 所导出的全部变量。如果需要取消一个变量的导出，需要使用 “-n” 参数。例如要取消前面导出的 DEBUG 变量，可以执行：

```
$ export -n DEBUG
```

下面介绍一个最常用的环境变量：PATH。这个环境变量指定了 shell 搜索命令的路径顺序。由于大部分 shell 命令是文件系统中的可执行程序，但我们在执行这些程序的时候可以不必指定路径，shell 根据 PATH 环境变量所指定的路径顺序来寻找同名的程序，所以如果没有设置 PATH 变量，执行简单的 ls 命令也要指定完整的路径/bin/ls 了。

shell 不会在当前目录中查找可执行程序，除非你设置了 “.” 在 PATH 环境变量中。但请不要这么做。因为一个恶意的用户很可能会在他的目录中写了一个假冒的 ls 程序，这样当别人进入这个目录并试图用 ls 列出文件列表的时候，实际执行的就可能是这个冒牌的 ls 了。如果当前目录中有其他与系统程序同名的可执行文件（例如 test），也会影响依赖于这个文件的脚本执行。同时设置了 “.” 到 PATH 环境变量还会使系统每次进入新的目录都要更新可用的可执行程序列表，降低了系统的效率。所以在执行当前路径里面的程序时候也要指定路径，最简单的方式就是使用 “./” 作为相对路径。

如果执行一个命令的时候发生如下错误信息：

```
command not found
```

[1]脚本的概念在第六章会有详细介绍

则可能的原因只有两个，一是命令真写错了，二是 PATH 环境变量的设置出了问题，造成本应该可以搜索到得命令没有搜索到。所以检查的时候也主要从这两个方面检查。

与路径有关的另外一个重要环境变量是 LD_LIBRARY_PATH。与 PATH 变量类似，它是 shell 用来寻找动态加载库的路径搜索顺序。有时候一些应用程序会因为这个变量没有设置正确而无法运行。

常用的环境变量还有 $PWD，它表示当前的路径名，其中的内容与 pwd 命令的输出一致。上面曾提到的 $HOME 环境变量代表用户的主目录。

4. 配置文件

bash 常用的配置文件有两个，一个是 ~/.bashrc，一个是 ~/.bash_profile，它们是用户运行 bash 的时候首先执行的脚本文件。其中前者是 bash 作为非登录程序运行的时候自动执行的，后者是 bash 作为登录程序或者在运行的时候使用了 "`--login`" 参数时自动执行的。用户可以把每次登录或者打开终端（每打开一个终端窗口都要执行一个 bash 程序）都要执行的命令放到这两个文件中。比较常用的就是把环境变量的设置放到这两个文件中，如在 ~/.bashrc 文件中增加：

```
$ export PATH=/usr/local/qte/bin:$PATH
```

这个设置就会在每次打开一个终端的时候生效。

5. 输入输出重定向和管道

在命令行界面下，用户命令一般从键盘接收数据，并把输出显示在显示屏上。但从编程的角度来看，程序从键盘输入与从文本文件输入没有本质区别，把输出内容显示到屏幕与输出到文件也十分类似。操作系统为应用程序提供了统一的输入输出界面，shell 利用这个特性实际提供给应用程序的只是两个 "标准文件"，即标准输入（stdin）和标准输出（stdout），默认情况标准输入就是键盘，标准输出就是显示器，不过 shell 也允许用户重新定义这两个文件。

例如要把 ls 命令的输出保存到文件中，就可以用 ">" 符号把标准输出重新定义为其他文件：

```
$ ls -l > filelist.txt
```

上面的命令中，如果把 ">" 换成 "»"，就可以把 ls 命令的输出添加到 filelist.txt 文件的末尾，而不是替代原有的内容。

在实际操作中，采用上面的方法并不能把所有的输出内容都保存到 ">" 符号后面的文件中，例如用 ls 命令列举不存在路径的文件时：

```
$ ls /no_such_file > filelist.txt
ls: cannot access /no\_such\_file: No such file or directory
```

生成的文件 filelist.txt 中将不会包含任何信息，这是因为系统中还有一个 "标准错误输出" 文件，程序的错误提示被输出到这个文件中，默认它也是计算机屏幕。为把标准错误输出的内容也输出到文件中需要使用 "2>" 符号（注意这两个字符间不能有空格），因为 2 是标准错误输出文件的编号（0 是标准输入，1 是标准输出），"2>" 就代表把 2 号文件重新定义，例如：

```
$ ls /no\_such\_file 2> err.txt
```

此时屏幕上不会显示任何错误消息，上面例子中看到的错误消息已经被重定向到 err.txt 文件中了。

为了把标准输出和标准错误输出的内容都定向到同一个文件中，可以使用 "&>" 符号，例如：

```
$ ls /no\_such\_file &> all.txt
```

与重新定义标准输出类似，Linux 下页可以重新定义标准输入。重新定义标准输入的符号是 "<"，例如 wc 命令（word count 的缩写，这个命令用来计算输入的单词个数）就可以把文件重定向为标准输入：

```
$ wc < filelist.txt
 37  264  2151
```

上面命令输出的三个数字分别是 filelist.txt 文件所包含的行数、单词个数和字符个数。

由于程序的输入和输出都是文件，shell 还允许用 “|” 符号把两个程序连接起来，让前面程序的输出作为后面程序的输入，因此 “|” 被称为管道符号，这种技术也被称为 “管道”。例如：

```
$ ls -l | wc
 37  264  2151
```

通过管道技术可以把多个命令组合在一起使用，实现比较强大的功能，这在 Linux 或者 UNIX 世界中，是非常常见的。Linux/UNIX 包含很多命令行的小工具，它们都使用可读文本作为输入或者输出，它们设计的基本思想就是 “只做好一件事情并能很好地同其他程序合作”。这种设计理念使得 Linux/UNIX 的 shell 功能非常强大，几乎没有需要管理员用 C 编程的机会。

6. 几种引号的用法

在 shell 中会用到双引号、单引号和反引号。当一个命令的参数可能包括特殊字符（如空格和星号等）的时候，就可以用双引号括起来。例如要进入一个叫 “new file” 的目录，就可以用下面的写法：

```
$ cd "new file"
```

当然也可以采用 “\” 号转义：

```
$ cd new\ file
```

单引号和双引号的用法基本相同，区别在于对环境变量的处理上。双引号中包括的环境变量会被用它的值替代，单引号中会把 “$” 作为普通字符处理。例如：

```
$ echo "You are $USER"
You are bgates
$ echo 'You are $USER'
You are $USER
```

echo 命令用来显示一个字符串，而环境变量 USER 中记录的是当前登录的用户名。

反引号是 “`” 键，标准 pc 键盘上与 “~” 共用一个按键。反引号中包括的是一个完整的命令，它会用这个命令的执行结果来替代自身。例如：

```
$ echo `date`
2009-10-20
```

显示的结果就是当前的日期。

2.1.5 进程和作业管理

UNIX 从一开始就支持多个作业的同时运行，Linux 也继承了这个优点。如果一个程序的运行比较耗费时间，我们可以把它放在后台运行，前台仍然可以得到提示符运行新的命令。把程序放入后台的方法就是在输入命令行的后面加上一个 “&” 符号，例如：

```
$ ./MyProg &
$
```

可以用 jobs 命令查看当前后台有哪些程序在运行。jobs 命令的输出给每个后台运行的程序都编了序号。如果要把第 5 个程序调整到前台运行可以用 “fg 5” 命令。如果要把前台的程序调整

到后台运行则可以直接按 “Ctrl+z” 先把进程挂起，然后输入 bg 命令把挂起的进程放入后台执行。

ps 命令用来查看系统中运行的所有进程。它的参数很多，比较常用的是 aux，这个参数组合不但显示了系统中的所有进程，还提供与这些进程相关的很多有用信息，往往我们最关心的是它们的进程号——PID。配合文本过滤命令 grep，可以只显示我们感兴趣的部分。例如查找系统中的所有 rxvt 进程可以用：

```
$ ps aux| grep rxvt
bgates  5252  0.0  0.3   6312  1608 tty1     S    May18   0:00 rxvt
bgates  8768  0.0  0.5   7896  2820 tty1     S    May21   0:03 rxvt
```

由于 grep 的过滤，这个列表没有表头，可以用下面的命令显示列表中每项的说明：

```
$ ps aux | head -n 1
USER   PID  %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

这个列表中 USER 表示运行程序的用户，PID 是进程的 ID 号，%CPU 和%MEM 分别代表进程占用 CPU 和内存的百分比，VSZ 是进程的大小（包括程序、数据和堆栈），RSS 是进程占用的 RAM 大小，TTY 是进程的控制终端，STAT 是进程的状态：R 代表运行，S 代表睡眠，Z 代表僵尸进程，等等。START 是进程的启动时间，TIME 是进程占用的 CPU 时间，COMMAND 是进程启动时的执行命令。

另外一个查看进程的命令是 top，它可以按照 CPU 的利用率实时地显示当前系统的进程状态，经常用这个命令查看系统中哪些进程运行不正常，过多地占用了 CPU。

kill 命令用来给运行的进程发送信号，它取这样的名字是由于 kill 命令默认发送终止运行的信号到进程。这个命令以进程的 PID 作为参数：

```
$ kill 5252
```

上面这条命令用来终止 5252 号进程的运行。如果 5252 号进程是当前用户运行的（root 用户可以操作其他用户的进程），kill 命令就可以把这个进程终止。不过这个命令的生效还有两个前提：一是 5252 号进程没有对终止信号进行截获；二是该进程没有死锁。对于常规不能 “杀” 掉的进程可以使用 “-9” 参数给进程发送不能被截获的 “KILL” 信号，如果使用这个参数也不能杀掉的进程就会变成 “僵尸进程”。

killall 命令以程序名作为参数，用来终止所有同名的进程。例如：

```
$ killall rxvt
```

这个命令会关闭所有以 rxvt 命令启动的进程。

2.1.6 系统管理类命令

1. passwd：修改登录密码

Linux 系统的用户信息都记录在/etc/passwd 文件中，早期的 passwd 文件还记录了用户的密码（经过加密），后来为了提高系统的安全性，用户经过加密的密码被记录在了/etc/shadow 目录中。普通用户可以用 passwd 命令更改自己的密码，root 用户还可以用这个命令更改其他用户的密码：

```
# passwd user\_name
```

2. ifconfig：IP 地址配置

不加参数的 ifconfig 命令可以显示 Linux 系统中所有网络接口的配置情况。常用的以太网在 Linux 被称为 eth，第一块网卡是 eth0，第二块网卡就是 eth1，以此类推。配置 eth0 的 ip 地址可以用下面的命令：

```
# ifconfig eth0 192.168.0.50 netmask 255.255.255.0
```

这条命令配置 eth0 设备的地址是 192.168.0.50，网络掩码是 255.255.255.0。

配置默认的网关用 route 命令，下面的命令配置网关为 192.168.0.1：

```
# route add default gw 192.168.0.1
```

多数 Linux 发行版在系统启动的时候用一个脚本对网络地址进行配置，在 Debian/Ubuntu 系列的系统中，这个脚本是/etc/init.d/networking，root 用户可以用下面的命令重新对网络进行初始化：

```
# /etc/init.d/networking restart
```

3. poweroff：关机命令

在 Linux 下关机有很多种方法，最简单的命令就是 poweroff，它可以正常地终止系统所有进程，最后关闭计算机。类似的还有 reboot 命令，它用来重新启动系统。还有一个命令是 shutdown，它支持一些特殊的功能，其语法格式如下：

```
shutdown [参数] time [提示信息]
```

其中的提示信息会发给所有登录的用户，而 time 参数表示系统还有几分钟要关闭，留给其他用户一些时间保存自己的工作。例如：

```
$ shutdown -r +5
```

表示系统将在 5 分钟后重启。“-r” 参数表示重启，表示关机用 “-h” 参数

4. mount：挂载文件系统

使用任何设备上的文件系统（例如硬盘和 USB 磁盘）前，必须先挂载（mount）它们。在 Linux 下执行 mount 命令就可以看到当前挂载的设备情况：

```
$ mount
/dev/hda1 on / type auto (rw,noatime)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec)
udev on /dev type tmpfs (rw,nosuid)
devpts on /dev/pts type devpts (rw,nosuid,noexec)
```

为了挂载 USB 盘，首先要有个挂载点，也就是一个普通目录，例如

```
# mkdir /mnt/usbdisk
```

USB 盘在 Linux 中被虚拟为 scsi 设备，因此可以用下面的命令挂载：

```
# mount -t vfat /dev/sdc1 /mnt/usbdisk
```

上面的命令假定 USB 盘被识别为 “/dev/sdc1” 设备，而且具有一个分区。其中 “-t” 参数指定了所挂载文件系统的类型，“vfat” 是 Windows 系统中的 FAT32 类型。挂载成功之后就可以在 “/mnt/usbdisk” 目录中访问 USB 盘中的文件内容了。

取消文件系统的挂载需要执行 umount 命令

```
# umount /mnt/usbdisk
```

为了保证文件操作的最终结果能被写入 USB 盘，在取下 USB 盘之前必须执行 umount 操作。

2.1.7 其他命令

1. 压缩解压文件

网络上传播的文件经常采用压缩方式存储以节省带宽，在 Linux 下常用的压缩解压程序是 gzip 和 bzip2。一般可以通过文件的扩展名来了解文件的压缩格式：.gz 扩展名表示 gzip 压缩，.bz2 扩展名表示 bizp2 压缩。

用 bzip2 压缩文件的命令如下：

```
$ bzip2 file\_name
```

解压文件用 bunzip2 命令：

```
$ bunzip file\_name.bz2
```

gzip 的使用方法完全相同，只要用 gzip 替换 bzip2，用 gunzip 替换 bunzip2 就可以了。

2. tar：文件打包/解包命令。

tar 是在 Linux 中最常见的打包格式，它可以把多个文件和目录打包成一个文件，一般这样的文件的扩展名为.tar。打包的文件经常配合 gzip 或 bzip2 程序进行压缩。Linux 下的 tar 命令可以自动调用这两个程序，生成压缩的打包文件。如果要建立压缩文件包，可以用下面命令：

```
$ tar cfvz foo.tar.gz dir1 dir2 file1 file2
```

最后面的四个参数列举了要打包压缩的文件。“cfvz” 参数的含义如下：

- c：表示压缩动作；
- f：表示从文件解压（默认是从磁带设备，这是 tar 程序的原始目的）；
- v：表示详细显示操作过程，列举每个被操作的文件；
- z：表示文件采用 gzip 压缩格式。如果要使用 bzip2 的压缩格式，这个参数要换成 j。

生成的文件取名为 foo.tar.gz，这也是这种文件标准的命名方式，从这个文件名就可以知道这个文件是先用 tar 进行打包然后再用 gzip 进行压缩的。

对于压缩文件的解压可以用下面的命令：

```
$ tar xfvz foo.tar.gz
```

其中 x 参数表示解压，其他参数含义和上条命令一致。如果把 x 参数换成 t，就表示对文件进行测试，并不实际进行解压。

3. diff：比较两个文件的不同。

diff 命令用来比较两个文本文件的区别，并把比较的结果以文本的方式输出。它的输出结果可以作为 patch 命令的输入，例如生成 Linux 补丁的命令就可能如下：

```
$ diff -urN linux linux-my > mypatch.diff
```

上面的命令把两个目录的区别输出到 mypatch.diff 文件，根据这个文件就可以把 linux 目录中的内容更新为 linux-my 目录中的内容，或者进行相反的更新。这个 diff 文件的内容可能如下：

```
--- linux/Makefile 2008-09-30 15:54:32.000000000 +0800
+++ linux-my/Makefile       2009-10-10 06:13:53.000000000 +0800
@@ -190,8 +190,8 @@
 # Default value for CROSS_COMPILE is not to prefix executables
 # Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
 export KBUILD_BUILDHOST := $(SUBARCH)
-ARCH           ?= $(SUBARCH)
```

```
-CROSS_COMPILE  ?=
+ARCH           ?= arm
+CROSS_COMPILE  ?= arm-none-linux-gnueabi-

 # Architecture as present in compile.h
 UTS_MACHINE    := $(ARCH)
```

补丁文件可以包含多个文件的差别信息，每个差别信息包含三行相关文件的说明。其中以“---”开始的一行是原始文件的基本信息，包括文件名和文件时间；以“+++”开始的一行是修改后的目标文件信息；而以“@@”开始的行说明了原始文件与目标文件的修改位置。

对于前面的例子，“-190,8”表示源文件的修改位置为从 190 行开始的 8 行，“+190,8”则表示目标文件的修改位置也是从 190 行开始的 8 行。接下来的信息是具体的文件内容。其中以“-”开始的行表示从源文件中删除的内容，以“+”开始的行表示在目标文件中增加的内容。其他的内容就是修改位置的上下文，一般包含三行上下文信息以确认修改的位置。这个例子中共有 6 行上下文，2 行删除，2 行插入，因此共有 10 行信息。

4. patch：为文件“打补丁”

diff 命令输出的主要用于源代码的更新。例如在 Linux 内核源码发行的时候，如果每次更新一个版本就把所有源代码文件打包放到服务器上供人下载，就会额外增加很多网络流量。因为好多内核文件并没有变化，有变化的文件也可能只是改变了很少的几行。所以我们可以在内核网站看到好多以 patch 开头的文件名，这些文件就是内核的“补丁”，它们只记录新版本和旧版本的不同，只要下载这个补丁就可以从旧版本源码得到新版本源码，这样就可以大大减小下载文件的尺寸。patch 命令就是为这个目的设计的，不光 Linux 内核采用这种方式发行，几乎所有的开源软件都是这样的。因此在软件开发过程中这条命令的使用频率非常高。

例如给 2.6.26 版本 Linux 内核的补丁为 patch-2.6.27，应用这个补丁后就可以得到 2.6.27 版本的 Linux 内核。假设 Linux 源码在当前目录下的 Linux 目录中，就可以用下面的命令“打补丁”。

```
$ patch < patch-2.6.27
```

“patch”命令常用的参数有

- -R：取消补丁，即恢复打补丁前的数据；
- -p：由于补丁文件记录了路径的信息。

关于 -p 参数有必要做更多的解释。以前面的 diff 文件例子中包含的 linux/Makefile 文件为例，当我们在 linux 目录中运行 patch 命令的时候，如果仍然使用和上面相同的命令，patch 就会找不到 linux/Makefile 这个文件，我们必须把前面的“linux/”去掉。这时候就必须用 -p 参数，表示忽略的目录级数。由于我们只需要忽略一级目录，最后写出的命令如下：

```
$ patch -p1 < patch-2.6.27
```

由于 diff 文件中还包含了三行的上下文信息，有时候即使文件的行号发生了变化，patch 命令仍然可以找到对应的修改位置。如果仍然找不到修改的位置，patch 命令还会忽略部分的上下文信息，以寻找合适的修改位置。当一切努力都失败之后，patch 命令会把失败的补丁部分保存为扩展名为.rej 的文件，而如果源文件已被修改，其未修改的版本会被保存为.orig 文件。

为了在真正应用补丁之前检验补丁的内容，可以使用 patch 命令的 `--dry-run` 参数，这个参数仅打印 patch 命令的执行结果而不真正修改源文件。例如：

```
$ patch -p1 --dry-run < patch-2.6.27
patching file arch/arm/mach-pxa/leds.h
patching file arch/arm/mach-pxa/Makefile
Hunk #2 succeeded at 23 (offset 1 line).
patching file arch/arm/Kconfig
Hunk #1 FAILED at 653.
```

```
1 out of 1 hunk FAILED -- saving rejects to file arch/arm/Kconfig.rej
```

5. find：在文件系统中寻找文件

find 命令的功能很多，最简单的就是在指定目录下按文件名查找文件，命令的格式如下：

```
find [目录] -name [文件表达式]
```

例如：

```
$ find . -name "*.txt"
```

这条命令在当前目录中查找所有以.txt 结尾的文件。

find 除了可以按文件名查找，还可以按修改时间（-mtime）、文件属主（-user）、文件权限（-perm）等查找。例如查找修改日期在 3 天以前的文件：

```
$ find /var/log -mtime +3
```

find 命令比较强大的功能是 “-exec” 参数实现的，这个参数后面可以另加一条命令，使所有找到的文件都用这个新命令进行处理。例如要删除所有扩展名为.tmp 的文件：

```
$ find . -name "*.tmp" -exec rm -f {} \;
```

这个命令的最后比较特殊，“{} \;” 是 exec 参数所要求的。

“查找并执行” 这个功能还可以通过 xargs 命令来实现，xargs 可以避免有时候用 exec 参数所出现的 “参数列太长” 错误。由于 xargs 是单独的命令，所以要用 “|” 来传递参数：

```
$ find . -name "*.tmp" | xargs rm -f
```

这种方式不必添加 “{} \;”，显得更加清晰和明白。

6. grep：文件内查找

grep 命令的格式如下：

```
grep [参数] 模式 [文件名]
```

参数部分常用的有：

- -r：递归查找，即还查找子目录中文件；
- -i：不区分大小写查找；
- -n：打印所在文件的行号。

模式部分为一个正则表达式（regular expression），用于匹配特定的字符串，正则表达式在 Linux 中用得很多，这里介绍几种简单的匹配规则，如表 2.1 所示：

配合 grep 命令，下面的命令在当前目录中的所有文件里查找 “MODVERSIONS” 这个字符串：

```
$ grep -r "MODVERSIONS" *
arch/s390/defconfig:# CONFIG\_MODVERSIONS is not set
arch/ppc/configs/pmac\_defconfig:CONFIG\_MODVERSIONS=y
 ... ...
```

grep 把它找到的匹配行显示出来，格式为 “文件名: 匹配行内容”。

7. su：切换用户 (switch user)

当用户想使用另外一个用户名时就可以使用 su 命令，例如 “su tomcat”。不带参数使用时表示切换到超级用户，这可能是 su 命令最常用的方式。

su 命令还可以加一个 “-” 参数，表示使用新用户的登录环境，就如同重新使用了新用户登录一样。

表 2.1: 正则表达式

表达式	含义
x.y	“.” 代表任意字符，“x.y” 匹配 x ＋任何一个字符＋y，例如 “xcy”，“xky”
x\.y	“\” 用来转义后面的字符，“x\.y” 就匹配 “x.y” 本身
a[x0-9]b	“[]” 中是多选 1 的，因此这个表达式匹配 “axb” 或者 “a3b” 等，但不匹配 “ab”
s[^xyz]t	“[]” 中的 “^” 表示不选择，这个表达式可以匹配 “sdt” 但不匹配 “syt”
^x	“^” 代表行首，这个表达式匹配以 x 开始的字符串，例如 “xzy”
x$	“$” 代表行尾，这个表达式匹配以 x 结束的字符串，例如 “zyx”

如果要退回到前一个用户，需要使用 “exit” 命令。

8. sudo：以其他用户身份执行命令

与 su 命令类似，这个命令也用来改变当前用户的 “身份”，只不过并不提供一个交互界面，仅仅用新用户的身份执行一条命令。默认情况下这个命令也是使用 root 作为新身份，在前一章中，我们使用 “sudo passwd root” 就是这样的一个例子。

9. dmesg：查看 Linux 内核消息

在编写 Linux 驱动程序或者对 Linux 进行系统管理的时候，经常需要察看操作系统输出的提示或警告信息。这些内核消息是在内核代码中以 printk 函数打印到系统终端上的，在本书第七章对内核驱动进行介绍的时候将对内核消息做更多说明。

在很多情况下，内核消息对用户不是一直可见的，普通用户可以使用不带参数的 dmesg 察看内核消息，而 root 用户可以使用参数对内核消息进行控制。使用 -c 参数将在打印内核消息之后清除内核消息缓冲区；-n 参数用来指定内核显示消息的记录级别；-s 参数用来指定 Linux 内核消息缓冲区的大小。

Linux 下面的常用命令还有很多，限于篇幅就不多介绍了，熟练地掌握这些命令是可以高效工作的前提，但也不要指望可以一蹴而就，掌握 Linux 操作系统需要日常不断的积累，只有多使用才能熟练，否则只会逐渐忘记。

2.2 GCC 及 GNU 工具简介

GCC (GNU Compiler Collection) 是 GNU 项目的编译器套件，它包括 C、C++、Fortran、Java 等语言的编译程序，其中本书用到的是 C 语言编译器 gcc 和 C++ 语言的编译器 g++。gcc 和 g++ 的用法类似，下面以 gcc 为例进行介绍。

2.2.1 gcc 用法简介

gcc 最简单的使用格式如下：

```
$ gcc hello.c
```

正常情况下，上面这条语句会把名字为 hello.c 的 C 源程序编译链接生成可执行代码，可执行文件的名字为 a.out。如果要指定不同的执行文件名字，就要使用 -o 参数，例如下面的命令格式：

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

gcc 命令在编译源程序的过程中，实际经过了预处理、编译、汇编和链接的四个过程。通过不同的 gcc 参数可以把这四个过程分解：

```
$ gcc -E hello.c -o hello.i
```

“-E” 参数仅对源文件进行预处理，只有 #include、#define 这些预处理指令被分析和解释，可以通过生成的 hello.i 文件查看结果。

```
$ gcc -S hello.i
```

“-S” 参数对源文件进行编译。上面的命令执行后会生成名字为 hello.s 的汇编源文件。这个扩展名为 “.s” 的文件就是 gcc 的编译结果。

```
$ gcc -c hello.s
```

“-c” 参数对源文件进行汇编，也就是把汇编语言翻译成二进制机器代码。生成的文件扩展名为 “.o”，实际上是格式为 ELF 的目标文件。

```
$ gcc -o hello hello.o
```

最后一步就是生成可执行代码了，-o 参数指定了输出文件的名称。生成可执行代码的过程还蕴含着对代码的链接（gcc 实际会调用 ld 命令进行链接，这里也可以直接使用 ld 命令，参考后面对 ld 的介绍），用来找到所有符号的定义，让程序在执行的时候可以找到相关代码。

gcc 的命令行选项非常多，表 2.2 仅介绍最常用的部分。

表 2.2: gcc 基本参数

选项	说明
-o 文件名	指定输出文件名，默认文件名是 a.out
-c	只编译不链接，生成目标文件
-IDIRNAME	将 DIRNAME 加入头文件搜索的默认路径中
-LDIRNAME	将 DIRNAME 加入库文件搜索的默认路径中
-DFOO	定义符号 FOO，相当于程序代码 #define FOO
-lfoo	把库文件 libfoo 链接到目标文件，优先选择动态库
-static	执行静态链接
-g	在可执行文件中包含标准调试信息
-O	优化编译过的代码，常用的有 -O2、-O3、-Os
-ansi	支持 ANSI/ISO C 的标准语法
-pedantic	允许发出 ANSI/ISO C 标准所列出的所有警告
-Wall	打开所有警告，有利于发现程序隐藏的错误

2.2.2 binutils 简介

除了 C 语言的编译器，在软件开发的过程中还经常需要用到一些操作二进制文件的工具，这个工具包就是 binutils。这个软件包括许多命令，这里详细介绍一下最常用的几个，对于 ARM 系统的交叉环境来说，这些命令还要加上 arm-linux-前缀（参考 2.4 节）。

1. ar

ar 命令用来建立和修改程序库。库文件是一组编译后的二进制文件集合，里面包含的代码可以被其他程序重用，通过链接加入到其他程序的可执行程序中。

例如我们编写了两个 C 源文件：test1.c 和 test2.c，我们可以用 gcc 进行编译：

```
$ gcc -c test1.c
$ gcc -c test2.c
```

这样就生成了两个目标文件：test1.o 和 test2.o。然后用 ar 命令将它们组合成一个可用的程序库：

```
$ ar r libtest.a test1.o test2.o
```

生成的 libtest.a 文件就是一个标准的静态链接程序库，在 Linux 里经常被称为归档文件（ar 是 archive 的缩写）。如果我们在 gcc 编译的时候制定了参数 -ltest，就会把这个库文件中的目标代码链接进来。ar 命令常用的参数有：

- d 从归档文件中删除文件
- t 打印归档文件中的文件
- q 在归档文件中追加
- r 替换归档文件中的已有文件或者追加新文件
- v（于其他参数配合）输出较多的信息

2. ld

这个命令直接用到的机会虽然不多，但几乎在每次编译程序的时候都会被 gcc 调用，它的作用就是把多个目标代码组合成一个可执行的程序文件，这个过程叫做链接。在链接的时候，ld 会检查程序中使用了哪些库文件，并把对应的二进制目标代码加入到可执行文件中。

gcc 命令中与链接相关的参数都可以在 ld 命令中使用，例如 -l 与 -L 参数。下面的命令将 start.o 和 main.o 两个目标文件链接成为可执行文件 main。

```
$ ld start.o main.o -lmath -o main
```

ld 在查找库文件的时候，并不是搜索系统中的所有库文件，这样编译时间会很慢。所以除了标准的库文件，我们在链接的时候都要通过 -l 参数来指定库文件的名字，有时还要使用 -L 参数指定库文件所在的路径。如果 ld 查找了所有命令行指定的库文件和目标代码，都没有找到某些符号的定义，链接过程就会失败，并打印如类似下面的错误消息：

```
hello.o(.text+0x107b): In function `main ':
:undefined reference to `foo '
```

如果在编译程序的过程中出现这样的错误，原因就是缺少某些链接文件，或者在程序中写错了某个符号的名字。

3. nm

nm 命令用来列出目标文件中的符号清单，并给出每个符号的类型。例如执行：

```
$ nm libtest.a
```

输出数据会类似于：

```
test1.o
       U bzero
00000066 T checkcfg
00000078 T checkinput
….
```

符号的类型比较常见的是 “T”，表示代码部分。如果类型为 “U” 则表示没有定义，没有定义的符号必须在其他目标文件或者库文件中定义才能链接成功。

nm 命令经常和 grep 命令一起来查找特定的符号信息。例如：

```
$ nm libtest.a | grep TestAll
```

4. strip

strip 命令用来去掉目标文件中的符号表信息，一般我们用这个命令去掉目标文件的调试信息，以减少可执行文件的尺寸。例如：

```
$ strip --strip-debug test1.o
```

上面的命令仅去掉了目标文件中的调试信息，如果不加任何参数执行，就会去掉目标文件的全部符号信息。

5. objdump

objdump 命令用来列出目标文件的信息。常用的参数如下：

- -d 参数可以对目标文件进行反汇编
- -h 参数可以显示目标文件各段的头部信息
- -t 参数可以显示目标文件的符号表

6. objcopy

objcopy 命令用来转换目标文件的格式。常用的输出格式为 binary 格式与 srec 格式。本书将用到 binary 格式，它把目标文件转换为可以加载到内存的映像文件，在没有操作系统的情况下，可以直接把映像加载到内存并执行。objcopy 的调用示例如下：

```
$ objcopy hello.elf -O binary hello.bin
```

2.2.3 GNU 工具应用示例

首先我们选用一个非常简单的 C 语言代码作为示例，这个代码包括一个头文件和两个 C 源文件：

```
-----------------  myheader.h  --------------
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

void myprintf(void);

#endif
----------------- myheader.c ------------------
#include <stdio.h>

void myprintf(void)
{
       printf("Hello gcc!\n");
}
----------------- main.c ------------------
#include "myheader.h"

main()
{
       myprintf();
}
```

紧接着编译两个 C 语言的代码

```
$ gcc -c main.c myheader.c
```

这个命令将生成两个目标文件：main.o 与 myheader.o。用 ar 命令把 myheader.o 放入到静态库中：

```
$ ar r libmyheader.a myheader.o
ar: creating libmyheader.a
```

现在要生成最终的可执行文件，如果按照下面的命令操作：

```
$ gcc -o main main.o
/tmp/ccUHJnDu.o: In function `main':
main.c:(.text+0x12): undefined reference to `myprintf'
collect2: ld returned 1 exit status
```

命令会返回错误，说找不到 myprintf 这个函数的定义。因此必须要 “告诉”gcc，myprintf 这个函数在什么地方。一种办法是把 myheader.o 这个目标文件也放到 gcc 的命令行：

```
$ gcc -o main main.o myheader.o
```

另外一种方法是使用刚刚生成的库文件 libmyheader.a

```
$ gcc -o main main.o -lmyheader -L.
```

库文件实际上就是目标文件的集合，所以在这里 libmyheader.a 的作用与 myheader.o 是一样的。生成的可执行文件的名字为 “main”。

2.3 Makefile 简介

如果编写的程序只有很少的源文件，我们可以使用 gcc 命令一个个地编译，最后链接生成可执行文件。但当源文件的数目非常多的时候，仅把这些文件编译就具有很大的工作量。当你觉得一个工作比较复杂而且单调，就要想到 Linux 一定存在某些工具让这个工作变得简单。Makefile 就是为了让编译和链接程序的工作更加高效而出现的。实际上，Makefile 只不过是 GNU make 命令所使用的输入文件，make 命令可以根据 Makefile 的指示完成一系列的工作。Makefile 不仅仅能用来对源文件进行编译，还可以对编译后的文件进行安装、卸载，甚至执行任何你可以通过命令行输入的命令。

当在终端中运行 “make” 命令的时候，它会在当前目录中寻找 makefile 或者 Makefile 文件。注意到 Linux 对文件名是有大小写区分的，一个目录中可能同时存在 makefile 与 Makefile 两个文件，此时前者具有优先权。但习惯上，我们都使用后者作为 make 命令的输入。

2.3.1 Makefile 基本语法

Makefile 文件支持的语法一共包括五部分：显式规则、隐含规则、变量的定义、文件包含和注释。

1. 显式规则。

显式规则说明了如何生成一个或多个目标文件。这是在 Makefile 中明显指出的规则、依赖关系和生成命令。

2. 隐含规则。
   隐含规则可以让我们比较粗糙而简略地书写 Makefile，这是由 make 所支持的。
3. 变量的定义。
   在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 文件包含。
   主要是像 C 语言中的 include 一样，包含另外一个 Makefile，其关键字也是 “include”。
5. 注释。
   Makefile 中只有行注释，其注释是用 “#” 字符作为先导。

Makefile 的基本语法元素是 “目标” 和生成这个目标的依赖关系与执行的命令：

```
target: dep1 dep2 ...
<Tab>command1
<Tab>...
```

target 就是一个目标，它一般是一个文件，也就是后续的一系列命令所要生成的文件。冒号后面的就是这个目标所依赖的文件与其他目标。make 命令会根据依赖关系和文件生成时间来判断是否要重新生成一个目标。例如，若 target 的文件生成时间晚于它所依赖的所有文件，那么就可以判断出这个目标不必重新生成，生成这个目标的命令都不必执行，也就节省了程序编译的时间；而一旦用户修改了 target 所依赖的某个文件，使得那个文件的生成时间更晚，make 命令就会重新生成 target 文件了。

<Tab>command1 代表一个以制表符开始的命令，<Tab> 就是键盘上的 “Tab” 键所输入的内容。Makefile 的格式要求所有的命令必须由制表符开始。由于印刷技术无法区分制表符与空格，所以这里都用 <Tab> 代表一个制表符。生成一个目标可以包含多个命令，每个命令都必须以制表符开始，不能用空格代替，当 make 命令决定要重新生成一个目标的时候，会依次执行其包括的每一条命令，如果遇到错误就会停止。如果希望 make 命令忽略遇到的错误，可以在命令的前面再加一个 “-” 号，make 在执行这条命令的时候如果出现错误就会忽略并继续执行后面的命令。

例如你在当前目录中有一个包含下面内容的 Makefile：

```
all:
<Tab>-rm tempfile
disp:
<Tab>@echo Hello make
```

make 命令可以使用目标作为它的参数，也就是让 make 命令根据 Makefile 生成这个目标。如果没有指定任何目标，make 命令就使用 Makefile 中的第一个目标作为默认。这样，当你输入 make 命令并回车之后，make 命令就自动执行上面的 rm 命令，把当前目录中的 tempfile 文件删除。rm 命令前面的 “-” 就是告诉 make 命令：如果当前目录没有 tempfile 文件，rm 命令所返回的错误可以忽略。

这个例子的第一个目标叫 “all”，它没有依赖关系，也就是说无论如何，其后面的命令都要执行。但如果当前目录中存在一个名字为 “all” 的文件，由于它的文件时间比所有其依赖的文件都新（因为没有依赖文件），其后的命令就无法执行了。这时候必须指定 “all” 只是一个目标，而不是文件。在 Makefile 中用 “.PHONY” 目标来表示，所以在前面的 Makefile 中，还必须加一行代码：“.PHONY: all disp”，才比较完整。

上面的 Makefile 还用到了 “@” 符号，这个符号告诉 make 命令不必输出其后命令的命令本身。因为默认情况下 make 命令在执行时，会把它所要执行命令的整个命令行显示出来，而对于 echo 命令来说这个功能就有点多余了。如果没有这个 “@”，make disp 命令的输出如下：

```
echo Hello make
Hello make
```

加上 “@” 符号后，就只输出第二行了。

再看一个简单的例子，这个例子用来编译上一节的源文件：

```
main: myheader.o main.o
<Tab>gcc -o main myheader.o main.o
main.o: main.c myheader.h
<Tab>gcc -Wall -c main.c
myheader.o: myheader.c myheader.h
<Tab>gcc -Wall -c myheader.c
.PHONY: clean
clean:
<Tab>-rm myprog *.o -f
```

这个例子是 Makefile 的最基本的格式：每行文字要么是目标的说明，要么是执行的命令。如果当前目录中只有三个源文件和这个 Makefile 文件，当不带参数执行 make 命令的时候，make 发现第一个目标是 “main”，它将成为默认目标，要先用 “gcc -o main myheader.o main.o” 命令生成 “main”，而 “main” 又依赖于 myheader.o 和 main.o，它们的生成也有它们的生成规则……经过一系列的递归，“main” 最终会被生成；除非在执行命令的过程中发生错误，make 的进程才会异常终止。

前面的例子都是介绍 Makefile 的显式规则，而实际上 GNU make 比较聪明，有些规则可以不必写出来，make 知道.o 文件是由.c 文件通过 C 编译器生成的，也知道它一定依赖于同名的.c 文件，所以下面的两行代码：

```
main.o: main.c myheader.h
<Tab>gcc -Wall -c main.c
```

完全可以写成：

```
main.o: myheader.h
```

这就是 make 的一个隐含规则，只是 make 命令会默认使用 $(CC) $(CFLAGS) -c 来编译目标文件。这里的 $(CC) 和 $(CFLAGS) 就是 Makefile 中的变量，默认情况下 $(CC) 变量的值是 “cc”，在 Linux 操作系统下一般等同于 gcc，$(CFLAGS) 变量一般为空值。

2.3.2 Makefile 中变量的用法

Makefile 中变量的引用采用 “$” 前导并用括号把变量包含在其中，如前面的 “$(CC)” 所示。变量赋值有多种形式，所使用的操作符有 “=”、“:=” 和 “?=”，例如：

```
CC=gcc
```

如果赋值命令中同样使用了变量，就需要对变量进行值的替换，也叫展开。采用 “=” 号赋值的时候，变量的赋值只有到了变量被引用的时候才展开，而采用 “:=” 号赋值的时候，变量的值在赋值的时候就已经展开了。这个特性在下面的例子中可以看得非常明显。

```
ONE=value
TWO=$(ONE)
THREE=$(TWO)
FOUR:=$(THREE)
TWO=new
all:
        @echo $(THREE) $(FOUR)
```

对于上面的 Makefile，如果执行 make 的话，输出的结果是 “new value”，也就是说 $(THREE) 的值为 new，而 $(FOUR) 的值为 value。根据 “=” 与 “:=” 的区别，下面的定义是非法的：

```
CFLAGS=$(CFLAGS) -g # 错误用法
```

因为它引入了递归，此时只能使用 “:=” 符号进行赋值。

还有一种是增量赋值，使用 “+=” 符号。其含义与 C 语言的 “+=” 符号类似，就是在原始值的后面再增加一个部分，只是在两个部分之间会自动加入分隔符。例如：

```
CFLAGS=-g
CFLAGS+=-Wall
```

在 Makefile 中可以直接使用系统的环境变量，例如 $(HOME)，$(USER) 等。但在 Makefile 中定义的变量具有更高优先级，例如我们在 Makefile 中定义了 $(CC) 变量，它就会取代环境变量中的 $(CC)。如果希望环境变量具有更高的优先级，可以在 make 命令后面加 “-e” 参数。

使用 “-e” 参数可能会不小心变更了 Makefile 中不想被更改的变量。这时最好用 “?=” 符号给特定的变量赋值，使得它们可以被环境变量取代。这个赋值符号的含义是：如果这个变量没有被定义，则执行赋值操作，否则赋值操作不被执行。所以这种赋值方式具有最低的优先级，可以在不使用 “-e” 参数的时候被环境变量覆盖。

另外，Makefile 中还可以使用一些自动变量，它们为书写 Makefile 带来了很多方便：

```
$@                      规则中目标文件的名称
$<                      依赖文件中的第一个
$?                      依赖文件中所有比目标更新的
$^                      所有的依赖文件，并去掉重复出现的名字
$+                      所有依赖的文件，包含重复出现的部分
```

2.3.3 Makefile 中的函数

Makefile 中还可以使用函数，函数的格式与变量的引用类似，也采用 “$” 符号开始，如：

```
$(function argu)
```

括号中分别是函数的名称与参数，下面介绍几个常见的函数。

1. $(subst from,to,text)
   subst 函数对 “text” 中的文本进行替换，把其中出现的 “from” 全部替换为 “to”。例如：

```
A:=$(subst ee,EE,teeth)
```

$(A) 的值应为 tEEth。

2. $(patsubst pattern,replacement,text)

patsubst 函数也是用于文本替换的，它支持模式替换。“pattern” 参数就是查找的模式，它允许使用 “%” 作为通配符。比较常用的情况是用来替换文件的扩展名。例如：

```
A := $(patsubst %.c,%.o,a.c b.c c.c d.c)
```

$(A) 的值应为 “a.o b.o c.o d.o”。

3. $(wildcard pattern)

wildscard 函数用来通配文件系统中的文件。“pattern” 参数为采用通配符描述的文件列表。例如：

```
A := $(widscard *.c)
```

则 $(A) 包括当前目录下的全部 C 源文件的文件名。

4. $(shell command)

shell 函数用来执行一个 shell 命令，它的用法和 BASH 中的反引号用法 “‘” 类似，例如：

```
A := $(shell date)
```

$(A) 中的值就是 shell 命令 “data” 的执行结果。

下面这个例子主要演示了变量和函数的用法，前面加上了行号。

```
 1 CC=gcc
 2 CFLAGS=-O2 -Wall -ansi
 3 PROGRAM=myprog
 4 SRC := $(wildcard *.c)
 5 OBJS := $(patsubst %.c,%.o, $(SRC))
 6 HEADERS := $(wildcard *.h)
 7 $(PROGRAM): $(OBJS)
 8         $(CC) $(CFLAGS) $(OBJS) -o $(PROGRAM)
 9 %.o : %.c $(HEADERS)
10         $(CC) $(CFLAGS) -c $< -o $@
11 clean:
12         rm -f $(PROGRAM) $(OBJS)
```

例子的前两行用来定义常规的 $(CC) 与 $(CFLAGS) 变量。第 3 行定义了程序的可执行文件为 myprog。第 4 行使用了 wildcard 函数，定义 $(SRC) 变量为目录中的全部扩展名为 c 的源文件。第 5 行用 patsubst 函数将 $(SRC) 变量中的 “.c” 全部替换为 “.o”，即 $(OBJS) 变量中的内容是全部 c 程序文件的编译结果。第 6 行同样用 wildcard 函数定义了 $(HEADERS) 变量，内容为全部的 “.h” 头文件。第 7 行定义了第一个 Makefile 目标位前面定义的可执行文件，它依赖于 $(OBJS)，即全部的目标文件。第 8 行为生成可执行文件的命令。第 9 行使用通配符定义了任何一个扩展名为 “.o” 的文件都依赖于其源文件及全部的头文件。第 10 行定义了生成 “.o” 文件的命令，其中用到了上小节介绍的自动变量。第 11 行定义了 clean 目标，用来删除生成的文件，其实现命令在第 12 行定义。

2.4 交叉编译原理

前面介绍的编译过程都只能产生本地执行的代码。例如在 x86 平台使用 gcc 工具编译所产生的可执行代码是适用于 x86 平台的机器代码。但嵌入式 Linux 的开发一般都在一台运行 Linux 的开发主机上进行，通过一套交叉编译工具把应用程序或者 Linux 内核编译成可以在目标系统中执行的二进制代码。这个编译过程就叫做 “交叉编译”，所使用的编译工具称作 “交叉工具链”（cross toolchain）。

2.4.1 交叉工具链的生成

C 语言的交叉工具链同样是由 GCC 作为核心，只不过它的工作方式发生了变化。从前面介绍的 GCC 工作的四个步骤来看，C 语言的预处理部分完全相同，编译、汇编和链接过程都根据目标代码的体系架构进行了调整。因此，GCC 实际上是一款可以产生很多不同平台代码的编译器，只不过在它被编译为可执行程序之后，它所能生成的目标代码类型就确定了。同时 GCC 作为一个可移植的应用程序，可以运行在不同的硬件平台，因此就产生了各种不同的组合。例如我们前面接触到的是在 x86 平台运行的可以生成 x86 平台代码的 GCC；还可能存在 ARM 平台运行的可以生成 ARM 平台代码的 GCC，这两者都不是交叉工具链。而如果是在 x86 平台运行的可以生成 ARM 平台代码的 GCC 或者在 x86 平台运行的可以生成 MIPS 平台代码的 GCC 就是交叉工具链了。

交叉工具链依然主要由 glibc、binutils 和 GCC 构成，只不过需要使用它们的源代码，经过 GCC 的编译，得到特定用途的交叉编译工具。由于上述三个工具互相依赖，所以在编译的时候要采用特定的步骤：

1. 编译 binutils
2. 建立内核头文件
3. 编译 GCC 的 C 语言部分
4. 编译 libc
5. 编译完整的 GCC

较新的工具链都支持 “根系统”（Sysroot）的功能，可以通过 –with-sysroot 编译参数告诉被编译的工具根系统的位置。所谓根系统可以认为是一个完整的文件系统，但这个文件系统中文件是用在目标平台的。例如一个典型根系统中的文件结构如图 2.1 所示：

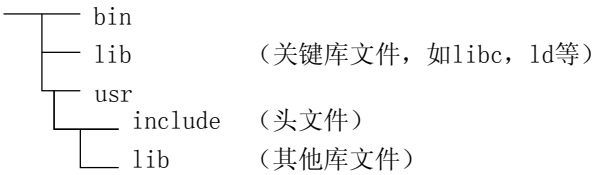

图 2.1: 工具链的 sysroot 结构

这个根系统中的文件是在编译工具链的过程中逐步建立的，在补充了其他的必要文件后，可以直接作为目标平台的文件系统来使用。

交叉编译工具中的命令一般都有一个前缀来与普通命令区分，例如 arm-linux-gcc 表示 ARM 平台 Linux 操作系统下的 C 语言编译器（arm-elf-gcc 也是比较常见的交叉编译命令，它不和特定的操作系统联系，生成的代码可以脱离操作系统直接运行，这种二进制代码被称为 flat 格式）。“arm-linux” 前缀目前已经逐步被抛弃，取而代之采用更为复杂的四元组格式：

```
arch-vendor-kernel-system
```

其中 arch 代表目标代码的体系名称，如 arm，powerpc 等。vendor 部分是用户可定义的，一般可以设置为 “none”。kernel 部分常用的设置为 “linux” 表示在 Linux 操作系统平台使用。system 部分在 ARM 体系中常见的是 “gnu” 和 “gnueabi”。其中 gnueabi 表示使用 GNU-EABI 兼容的二进制格式，是目前推荐的方式。GNU 代表使用的 c 库为 glibc。EABI（embedded application binary interface）是较新推出的针对嵌入式的程序二进制接口协议标准，已经被证明在某些领域具有更高的执行效率。如表 2.3 所示是一个常见元组的列表，它们一般可以根据应用环境任意组合使用。

这个工具链前缀格式在很多软件工具中使用，具有一定的通用性，所以在生成工具链的时候最好按照这个标准。本书中将使用 “arm-none-linux-gnueabi” 作为推荐的工具链前缀。

开源的软件大多采用 automake 和 autoconfig 软件生成配置文件，所以它们的编译方式也非常类似。基本的过程包含下面三个步骤：

表 2.3: 常见四元组的选择

Arch	Vendor	Kernel	System
alpha	pc	linux	gun
arm/armeb	unknown	freebsd6.2	gnueabi
i386/i486/i586/i686	none	elf	uclibc
powerpc/powerpc64	build_pc	uclinux	uclibceabi
mips/mipsel/mips64			

```
$ ./configure
$ make
$ sudo make install
```

其中./configure 命令用来对源代码进行配置，生成 Makefile 文件。第三步的命令用来把软件安装到系统中，由于安装的位置可能需要更高的用户权限才可以写入数据，往往需要 root 权限才可以安装软件。

我们在用 configure 命令进行配置的时候，常见的参数有 `--target`、`--host` 和 `--build`。其中 “target” 用来指定目标平台，“host” 用来指定目标程序的运行平台，“build” 用来指定当前编译的平台。这三个参数所设定的值就是前面曾经提到的四元组前缀。例如在编译 gcc 的时候如果指定了 target 为 `arm-none-linux-gnueabi`，则生成的 gcc 程序就可以编译出运行在 arm 平台上的可执行程序来。build 的参数指定了运行配置程序的平台，也就是开发机平台，在 x86 体系的主机上，这个值可能是 `i486-pc-linux-gnu`。

另一个要提到的参数是 `--prefix`，它指定了编译结果的安装路径，这样，当执行 make install 的时候，编译好的二进制程序和其他数据文件就会安装到指定的目录中。

下面我们分别来看一下编译工具链的几个过程，以下假设工具链的安装位置保存在 $PREFIX 环境变量中。

1. 编译 binutils

编译 binutils 的过程以编译 binutils-2.15 为例。首先要下载源代码，然后解压源文件到 binutils-2.15 目录：

```
$ tar xfvj binutils-2.15.tar.bz2
```

建立一个新的目录，然后进入到这个目录进行编译，免得中间结果和源代码混在一起。

```
$ mkdir binutils-build
$ cd binutils-build
```

然后是对代码进行配置，使用 binutils-2.15 目录中的 configure 程序：

```
$ ../binutils-2.15/configure --target=arm-none-linux-gnueabi \
> --prefix=$PREFIX --with-sysroot=$SYSROOT
```

为了获得好的显示效果，上面的命令采用了分行方式输入：当接近终端右端的时候输入 “\” 然后回车，会在下一行获得 “>” 的提示符（后面包含一个空格），可以继续输入命令。在实际使用的时候，可以直接连续输入，终端会自动换行。最后是编译与安装：

```
$ make
$ make install
```

2. 建立内核头文件

编译 gcc 之前需要配置好 arm 的内核，可以参考后面章节中对 Linux 内核配置的部分，选择适合 arm 平台的配置，在配置完内核之后就可以把需要的头文件复制到需要的地方。

```
$ mkdir -p $PREFIX/usr/include
$ cp -a include/linux include/asm-generic $PREFIX/usr/include/
$ cp -a include/asm-arm $PREFIX/usr/include/asm
```

较新的内核支持用如下命令完成相同的任务：

```
$ make headers_install INSTALL_HDR_PATH=$PREFIX/usr
```

3. 编译 GCC 的 C 语言部分

由于目前还没有可用的交叉编译的 libc，所以 GCC 的编译受到一定限制，好在还可以编译出 C 的编译器，可以完成对 glibc 的编译，之后才可以完整地编译 GCC。编译 GCC 的配置语句列举如下，make 与 make install 命令不再重复。

```
$ ../configure --target=arm-none-linux-gnueabi\
>  --prefix=$PREFIX --with-sysroot=$SYSROOT --enable-languages=c \
> --disable-shared --disable-libssp --disable-libgomp
```

4. 编译 glibc

编译 glibc 的配置命令如下：

```
$ ../configure --host=arm-none-linux-gnueabi --prefix=$PREFIX
```

这里配置 glibc 的时候指定了“`--host`”参数，因此编译的时候会使用前面生成的交叉编译版本的 gcc 编译器：`arm-none-linux-gnueabi-gcc` 来编译。最终生成可以在 ARM 平台使用的库文件，这些库文件在交叉编译应用程序的时候将链接到 ARM 平台的可执行代码中。

5. 编译完整的 GCC

在编译了 glibc 之后，我们就可以编译完整的 GCC 了，下面的配置例子编译了 c 与 c++ 两个编译器。

```
$ ../configure --target=arm-none-linux-gnueabi \
> --prefix=$PREFIX --with-sysroot=$SYSROOT \
> --enable-languages=c,c++ --enable-shared
```

编译工具链是一个耗时而且繁琐的过程，需要对每个软件包的内容与配置参数有深入的了解。同时开源软件的特点就是版本更替非常快，不同的源代码版本之间会有兼容性的问题，有些版本可能需要特殊的编译参数，有些版本还可能需要对源代码进行修改。因此上面介绍的编译过程也并不是完善的，实际操作中可能发生编译错误，或者即使编译成功，在使用的时候也会出现问题。实际在编译工具链的时候，需要充分利用网络上其他人的编译经验，参考别人的编译步骤与编译过程，免得给后续的工作带来不必要的麻烦。

随书下载文件中提供了编译好的工具链，只要使用这个工具链就可以满足大部分的需求，只有很特殊的情况才需要对交叉工具链进行调整，自己手动重新进行编译。下载文件中的工具链在 toolchain 目录中，名称为 `arm-eabi.tar.bz2`。需要把它安装到 `/usr/local/x-tools` 目录中。对于 gcc 4.x 之前版本的工具链，安装的目录不能更改，如果把编译工具安装到其他目录，在使用中可能会遇到问题。

在/usr/local/x-tools/目录中执行

```
# tar xfvj /cdrom/PreBuild/Toolchain/arm-eabi.tar.bz2
```

进行安装。安装完毕后 C 语言的编译器保存在/usr/local/x-tools/arm-none-linux-gnueabi/bin/目录中，名称为 `arm-none-linux-gnueabi-gcc`（本书提供的工具链还支持 `arm-linux-` 前缀，`arm-linux-gcc` 只是一个符号链接）。

交叉工具链在网络上也很容易下载到，例如在 CodeSourcery 的网站上就可以下载很多不同版本的工具链，用户可以根据需要进行下载。

2.4.2 自动生成工具链

生成交叉工具链是嵌入式开发中最基本的步骤，而生成过程又非常依赖用户的经验，因此有人编写了一个工具链的自动生成软件，可以根据用户的配置，生成特定的工具链。这个软件的名称叫 `crosstools-ng`，可以在网上免费下载，下载文件中提供了该软件的 1.7.0 版本。

`crosstools-ng` 软件的基本原理仍然是按照前小节介绍的方法对相应的源代码进行编译，只不过它把这个过程自动化了，用户可以通过配置界面选择所使用的 gcc、binutils 和 glibc 的版本，还可以对工具链的其他细节进行配置。当配置完成后，启动工具链的编译过程，`crosstools-ng` 会自动下载需要的软件源代码，并对代码进行必要的修改，最后把编译好的工具安装到指定的位置。

从网络下载到的 `crosstools-ng` 软件为源代码的压缩包，解压之后可以参考里面的 README 文件了解其安装过程。一种方便地安装方法是在软件的解压目录中执行

```
$ ./configure --local
$ make
```

crosstools-ng 在配置的过程中可能会出错，一般是缺少一些工具或头文件，此时可以用 aptitude 命令进行安装。编译成功后，当前目录就会生成一个叫 ct-ng 的可执行脚本，然后执行

```
$ ./ct-ng menuconfig
```

可以打开一个文本对话框，这个对话框是一个基于文本菜单的配置界面，可以对工具链的许多方面进行配置，如图 2.2 所示。这个配置界面风格被 Linux 操作系统下的许多软件采用，最著名的就是 Linux 内核的配置。

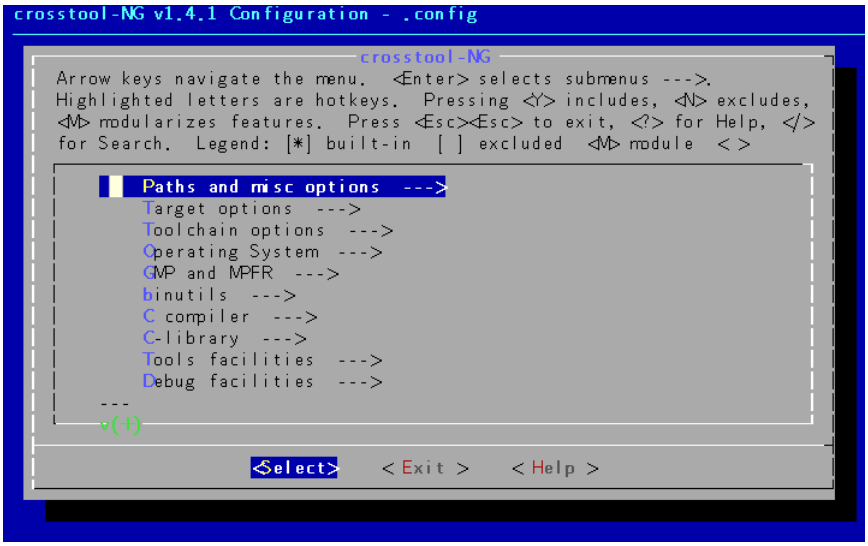

图 2.2: crosstools-ng 的配置界面

在 “Paths and misc options” 菜单中，主要设置一些路径信息，例如下载源代码的保存路径（Local tarballs directory），编译源码的工作路径（Working directory）和 prefix 参数的设置（Prefix directory）。

在 “Target options” 菜单中，主要对目标处理器的特性进行设置。如体系架构（Target Architecture），我们这里设置为 “arm”，选择使用 EABI（Use EABI），端模式选择 “小端”（Endianness，参考第三章的介绍）。

“Toolchain options” 菜单主要设置了是否采用根系统路径(Use sysroot’ed toolchain)和对四元组的 Vendor 部分进行设置。“Operating System” 部分主要对 Linux 内核的部分进行设置,可以选择不同的内核版本。“GMP and MPFR” 部分用来设置多精度运算的相关库信息。“binutils”、“C compiler” 和 “C-library” 部分用来选择工具链的源代码版本和对工具链进行深入的微调。“Tools facilities” 包含了一些附加的软件工具。“Debug facilities” 包含了一些帮助软件开发与调试的工具。

在选择内核版本的时候尽量选老一点的内核,或者在 glibc 的 “Minimal supported kernel version” 中设置较早的内核版本,否则在使用不满足 glibc 的要求的内核版本时,会出现 “Kernel too old” 的错误提示。

配置完成后,配置内容保存在了当前目录中的.config 文件中(在 samples 目录中也有一些默认的配置文件可供参考,下载文件中提供的 `arm-eabi.tar.bz2` 就是利用这个工具生成的,其配置文件主要基于 samples 目录中的 `arm-none-linux-guneabi.config` 文件)。此时执行

```
$ ./ct-ng build
```

就可以启动编译过程。如果没有发生任何异常错误,交叉工具链就可以顺利安装完成,编译过程的所有提示信息也都保存在了安装目录中的 build.log.bz2 文件中。

2.5 boot loader 简介

嵌入式设备同普通微机的一个重要区别就是它的特异性。对于微机来说,同样的操作系统可以运行在 586 处理器上,也可以运行在 Pentium 处理器上,这一方面得益于微机系统的兼容性,另一方面是微机中的 BIOS 隐藏了部分复杂硬件细节。而对于嵌入式系统,由于应用的千差万别,硬件设计根本没有兼容可言,要想在上面运行一个通用的操作系统,就必须进行代码级别的移植工作。为了使这个移植过程简单一些,也为了给安装操作系统提供方便,嵌入式系统一般也包含一段类似于 BIOS 的程序,这就是 boot loader,即 “启动管理器”。

为了正常启动 Linux 操作系统,boot loader 必须做好以下几件事情:

- 初始化最基本的硬件,如设置 CPU 速度、内存的时序、中断向量、检测 ram 的大小等。
- 初始化必要的设备,用来读取 Linux 的内核和根文件系统。
- 准备好一块连续的内存给内核,当需要 ramdisk 的时候还要为 ramdisk 准备一块连续的内存空间。
- 把内核和 ramdisk(如果需要)复制到准备好的内存区。
- 关闭 MMU 和高速缓冲存储器,设置 r0 寄存器为 0,r1 为设备的体系标志号。体系标志号可以在内核源码的 linux/arch/arm/tools/mach-types 文件中看到。
- 最后调用内核,放弃执行权利。

实际的 boot loader 应该有两部分的功能。一个是前面提到的启动 Linux 操作系统,这个过程应该可以自动完成,达到系统的加电启动效果。另外一个功能是为还没有安装操作系统的环境准备的,可以用来从外部 “下载”Linux 内核和文件系统到本地的存储介质上。为此,boot loader 经常会提供一个交互的环境,用来执行用户输入的命令,命令的输入和输出往往通过一个串口进行。

ARM 平台常见的 bootloader 有 U-Boot、Blob 和 RedBoot。

2.5.1 RedBoot 简介

RedBoot 是著名嵌入式开源操作系统 ECOS 的一个应用,是一款非常强大的启动管理器。它最大的特点有如下几点:

1. 核心为实时操作系统，扩展性强
2. 支持 flash 分区表，使 flash 的管理更方便。
3. 支持 gdb 的远程调试

RedBoot 通过串口与用户交互，工作情况下会打印出提示符：

```
RedBoot>
```

用户可以在提示符后面输入命令，RedBoot 的基本命令格式如下：

```
<command> <option,parameters>
```

每个命令还可以采用缩写方式，只要不会与其他命令引起混淆就可以了。如 fis 可以写成 fi。

RedBoot 命令主要分为下面几个部分。

1. 内存查看
   命令格式：

```
dump -b <location> -l [<length>]
```

用来显示特定地址的数据。这个命令在系统调试期间可以用来检查内存映射 IO 的内容。

2. flash 管理
   这部分是 RedBoot 比较危险的命令，因为要直接操作 flash，所以在操作之前一定要检察好参数，否则很容易不小心破坏 flash 中的数据。

```
fis init [-f]
```

用来对 flash 进行初始化，-f 参数表示对整个 flash 空间进行格式化。

```
fis free
```

用来显示 flash 的剩余空间，查看哪些空间没有被用到。

```
fis list
```

用来显示已经建立的 flash 分区

```
fis create -b <mem_base> -l <length> [-f <flash_addr>] [-e <entry_point>]
[-r <ram_addr>] [-n] <name>
```

这个命令用来建立一个 flash 分区，在执行这个命令之前，分区内容必须存在于 RAM 中（用下面介绍的 load 命令调入内存）。建立的分区名字为 <name> 的分区，长度是 <length>，FLASH 地址从 <flash_addr> 开始，它在 RAM 中的位置是 <mem_base>。<ram_addr> 的地址是这个分区通过 fis load 加载时的起始地址，<entry_point> 是它在被执行的时候的入口地址（只有在不知道分区类型的时候才需要这个参数）。

```
fis load name
```

用来把一个分区加载到内存。

```
fis delete name
```

用来删除特定名字的分区

```
fis erase -f <flash_addr> -l <length>
```

用来从 <flash_addr> 删除 flash 的内容，长度是 <length>

```
fis lock -f <flash_addr> -l <length>
fis unlock -f <flash_addr> -l <length>
```

用来对 flash 进行加锁标记和解锁。

3. 启动参数和脚本

fconfig 命令用来设置 RedBoot 的一些默认参数和启动脚本。执行

```
RedBoot> fconfig -l
```

会显示当前的配置情况。不加参数时，可以对各个参数进行设置。例如：

```
 1 RedBoot> fconfig
 2 Run script at boot: false true
 3 Boot script:
 4 Enter script,terminate with empty line
 5 >> fis load linux
 6 >> go 0xa0200000
 7 >>
 8 Boot script timeout (1000ms resolution): 3
 9 Use BOOTP for network configuration: false
10 Local IP address: 192.168.0.100 192.168.0.100
11 Default server IP address: 192.168.0.1
12 GDB connection port: 9000
13 Network debug at boot time: false
14 Update RedBoot non-volatile configuration - are you sure (y/n)? y
15 ... Unlock from 0x03f80000-0x03fc0000: block\_size: 40000,blocks: 80
16 .
17 ... Erase from 0x03f80000-0x03fc0000: .
18 ... Program from 0xa000bd78-0xa000c178 at 0x03f80000:
19 addr:3f80000,data:a000bd78,size:400.
20 ... Lock from 0x03f80000-0x03fc0000: .
```

上面的列表中，第 1 行为在提示符下输入的命令，从第 2 行到 14 行都是交互性的命令对参数进行设置。RedBoot 先打印出参数的描述和默认值，用户在冒号后面输入新的值。例如第 2 行用来设置是否在启动时自动执行 RedBoot 的启动脚本，默认为 false，用户输入 true。最后只有在回答了第 14 行的问题

```
Update RedBoot non-volatile configuration - are you sure (y/n)? y
```

之后，所做的修改才写入到 flash 中，重新启动开发板才能使设置生效。注意 RedBoot 的输入不支持行编辑功能，输入错误只能用退格删除重新修改，如果在输入命令过程中按了方向键，输入的命令往往就不能识别了。

RedBoot 的启动脚本其实就是一个命令列表，上面例子中的两行脚本让开发板在加电之后自动运行 Linux 操作系统。

如果设置了自动运行脚本，RedBoot 在启动的时候会显示如下提示：

```
== Executing boot script in 3.000 seconds - enter ^C to abort
```

此时必须在设定的时间内按下 “Ctrl+C” 组合键，否则就会自动运行设定的脚本。

4. 加载文件命令

RedBoot 可以和开发主机通过串口或网口传输文件，采用的命令就是 load：

```
load [-r] [-v] [-d] [-h <host>] [-m {TFTP | xyzMODEM | disk}]
[-b <base_address>] <file_name>
```

一般在使用这个命令的时候有下面几种方式：

从串口加载文件到开发板

```
RAM:load -r -b 0xa0200000 -m xMODEM zImage
```

从网口加载文件到开发板

```
RAM:load -r -b 0xa0200000 zImage
```

当采用网卡加载文件的时候，必须已经通过 fconfig 命令设置好了服务器和自己的 IP 地址，服务器必须正确运行了 TFTP 服务程序。

5. 其他常用命令

```
help <topic>
```

显示（特定）命令的帮助信息。

```
reset
```

重新启动开发板。

```
go <ram_addr>
```

跳转到指定地址运行

2.5.2 U-Boot 简介

U-Boot 全称为 Universal Boot Loader，顾名思义，U-Boot 的设计目标就是适用更多的平台和操作系统。目前 U-Boot 所支持的平台几乎与 Linux 一样多，操作系统平台除了 Linux 还支持 NetBSD、VxWorks、QNX、LynxOS 等。

U-Boot 的用户接口也主要是通过串口进行交互的命令行界面，U-Boot 的提示符为：

```
U-Boot>
```

U-Boot 的命令主要分为信息类、内存类、flash 操作类、执行类、网络类、环境变量类、文件系统类和其他特殊命令。我们仅介绍最常用的命令，其他命令的用法可以参考 U-Boot 的手册与在线帮助。注意 U-Boot 的输入同样不支持行编辑功能，输入错误只能用退格删除重新修改，如果在输入命令过程中按了方向键[1]，输入的命令往往就不能识别。

1. 信息类命令

```
help
```

简写为 “h” 或者 “?”。不带参数的 help 命令用来显示当前支持的全部命令。而具体要查看某个命令的细节可以把具体的命令作为参数。有时在编译 U-Boot 的时候为了节省存储空间，并没有把详细的用法编译到可执行代码中，通过 help 只能看到简单的命令说明。

```
flinfo
```

可简写为 “fli”

用来显示 flash 的基本信息，包括 flash 的类型，起始地址等。

2. 内存操作类

[1] 使用 bash 习惯后非常容易通过按上箭头键试图获得历史命令，但在 bootloader 环境中是不支持命令历史的，此时最好回车重新获得提示符后再输入命令

```
cp[.b .w .l] source target count
```

用来在内存中复制数据。

内存操作类命令一般都支持类型扩展，即在命令后面加上 ".b" 表示操作的数据类型为字节，".w" 表示数据类型为 16 位，".l" 表示数据类型为 32 位。所以 cp 的 count 参数也必须根据 cp 命令的操作类型来计算。

cp 命令还可以在 flash 与内存之间传递数据，即如果 cp 发现传递的目的内存在 flash 空间，就会调用适当的写入算法到 flash 中。

```
md[.b .w .l] address
```

显示内存内容。

```
mm[.b .w .l] address
```

这个命令可以用来修改一个地址单元的内容，修改完毕自动地址增量，直到用户输入一个不是十六进制数的内容为止。例如：

```
U-Boot> mm.w 100000
00100000: 0000 ? 0101
00100002: 0000 ? 0303
00100004: 0000 ? .
```

其中 "?" 后面的内容为用户输入，当用户输入 "." 时，修改终止。由于用户使用了 ".w" 扩展，所以地址每次增量为 2。

```
nm[.b .w .l] address
```

与 mm 命令不同，这个命令不对内存地址进行增量，可以不断地修改同一个地址的内容。这对于修改寄存器地址是非常方便的。

```
mw[.b .w .l] address value [count]
```

这个命令可以把一段连续的内存空间写入同样的 value 值。如果省略 count 参数，则仅写入一个内存单元。

3. flash 操作类

flash 的写入可以用 cp 命令，但在写入 flash 之前必须先清除，就要用到 erase 命令。erase 命令的语法格式有如下几种：

```
erase start end
```

删除 flash 的地址从 start 到 end

```
erase start +len
```

删除 flash 的地址从 start 开始的 len 字节

```
erase bank N
```

删除 flash 的第 N 个 bank

```
erase N:SF[-SL]
```

删除 flash 的第 N bank 的 SF 到 SL 扇区

```
erase all
```

删除全部 flash 内容

由于我们经常还会用到 NAND 类型的 flash，U-Boot 的 nand 系列命令也是必须掌握的。nand 的系列命令都以 “nand” 开始，

```
nand info
```

显示 NAND flash 的基本信息，例如：

```
U-Boot> nand info
Device 0: NAND 64MiB 3,3V 8-bit,sector size 16 KiB
```

```
nand read/write[.jffs2] addr off size
```

这条命令用于从偏移地址 off 开始读 size 大小的内容到内存地址 addr 或从偏移地址 off 开始写 size 大小的内容，源数据从内存地址 addr 开始。如果是操作 Jffs2 文件系统，还必须在 read/write 命令后加上 “.jffs2” 后缀。例如：

```
U-Boot> nand write.jffs2 0x220000000 0x00400000 0x12cc924
```

```
nand erase          [off size]
```

删除 NAND flash 的从 off 地址开始的 size 大小内容。如果不指定参数，则删除全部内容。

```
nand bad
```

显示 NAND flash 的坏块情况。

4. 执行类

```
bootm [addr [arg...]]
```

启动保存在 addr 位置的操作系统镜像，arg 可以用来指定 initrd 映像的地址。addr 参数可以在编译 U-Boot 的时候设置默认值，U-Boot 也会聪明地把最近加载的映像初始地址作为 bootm 的默认值。一般我们都通过 tftp 命令（参考网络类命令）加载 Linux 映像，然后直接用不加参数的 bootm 命令启动 Linux 操作系统。

```
go addr [arg ...]
```

从一内存地址开始执行程序。

U-Boot 支持特殊二进制格式的程序，当把程序加载到某一内存地址后，可以用 “go” 命令执行这个程序。

5. 网络类

网络类的命令主要是通过特定的协议加载可执行代码或者操作系统内核到内存。也包括通过串口传递文件的命令。以下是主要命令列表：

- bootp：通过 BOOTP/TFTP 协议获取 IP 地址并加载二进制文件
- dhcp：通过 DHCP 协议获得 IP 地址并加载二进制映像文件
- loadb：通过 Kermit 协议利用串口加载二进制映像文件
- loads：利用串口加载 S-Record 格式的二进制文件
- rarp：通过 RARP/TFTP 协议加载二进制映像文件
- tftp：通过 TFTP 协议加载二进制映像文件
- ping：检查是否可以和一个 IP 地址进行通信

例如通过 tftp 命令加载 Linux 操作系统内核到内存的命令如下：

```
U-Boot> tftp 20008000 uImage
```

U-Boot 就会根据环境变量中设置的 “serverip”（参考环境变量类命令），利用 TFTP 协议从服务器下载 uImage 文件，并把它保存在 20008000 的内存地址。然后就可以用 bootm 命令启动操作系统了。

ARM Linux 内核的加载地址一般在内存开始后的第 0x8000 个位置。例如开发板的内存物理地址从 20000000 开始，那内核的加载地址就是 20008000。这个地址在内核源代码中指定，一般不要更改。

6. 环境变量类

U-Boot 利用环境变量来控制一些命令的运行，这与 Linux 操作系统中的环境变量含义类似。基本的操作系统环境变量的命令有：

- printenv：打印出全部的环境变量值
- setenv：设置一个环境变量的值
- saveenv：把环境变量保存到非易失性的存储器中

其中 printenv 和 saveenv 都不需参数。setenv 的命令格式如下：

```
setenv name [value]
```

即把 “name” 环境变量设置为 “value”，如果 “value” 值省略，就相当于删除了 “name” 环境变量。

常见的环境变量有：

- ipaddr：用来设置实验板的 IP 地址。
- netmask：网络地址的子网掩码。
- gatewayip：网关的 IP 地址。
- serverip：用来设置开发主机的 IP 地址。部分命令（如 tftp）使用这个 IP 地址作为服务器的默认地址。
- bootargs：Linux 内核启动时使用的启动参数。
- bootcmd：默认的启动命令。
- bootdelay：系统启动之后，在执行默认启动命令前需要等待的时间。
- bootfile：默认的下载文件名。

例如设置实验板的 IP 地址：

```
U-Boot> setenv ipaddr 192.168.0.100
```

7. 其他命令

- versoin：显示 U-Boot 的版本号；
- reset：重新启动开发板。

2.6 实验 :开发环境建立

实验目的

1. 了解 Linux 环境常见服务的配置方法
2. 了解嵌入式开发的常用工具

实验内容

一、Linux 环境基本操作

在 Linux 系统中 root 用户具有最高的权限，在一般的情况下我们尽量少用 root 用户操作，以免不小心破坏系统。在 Ubuntu 系统中启用了 root 账户之后就可以用 su 命令切换到 root 用户。切换到 root 用户后，可以为自己建立一个新账户。在虚拟终端中输入如下命令：

```
# adduser your_name
```

这里再次提醒：“#” 号是超级用户的命令提示符，$ 是一般用户的提示符，其后才是输入的具体命令。建立新用户后还要把这个用户加入到 admin 组，否则新建的用户无法使用 sudo 命令。这一特性是 Ubuntu 10.04 中的/etc/sudoers 文件设定的，对于其他发行版则需要参考这个文件的设定。把用户加入其他组的命令如下：

```
# usermod -a -G admin your_name
```

退出当前登录窗口（Logout），用新建立的用户登录，用这个用户来练习本章介绍的 Linux 命令。

当需要使用 root 用户时，可以使用 sudo -s 命令获得 root 权限。很多系统命令都要求使用 root 用户，但不要所有任务都使用 root。今后实验中假定登录的用户名为 embedded，用户的主目录为/home/embedded。

二、配置 tftp 和 nfs 服务

开发主机所使用的 Linux 发行版并不重要，但必须至少包含下面几个软件包：

gcc：GNU C 编译器；

make：GNU make 软件；

tftpd：简单文件传输协议服务，用来把编译好的内核下载到嵌入式系统中；

nfsd：网络文件系统服务，可以方便地在嵌入式系统和开发主机之间共享文件。

对于 Ubuntu 发行版，默认安装的时候很多软件包都没有安装，可以使用 aptitude 命令进行安装。

1. 安装 tftp

```
# aptitude install tftpd-hpa
```

同时最好把 tftp 的客户程序也安装上，便于检查服务器是否安装正确：

```
# aptitude install tftp-hpa
```

TFTP（trivial file transfer protocol）是一种简单的不需要用户验证的文件传输协议。由于它比较简单，就比较适合在资源有限的环境中应用，例如下载 Linux 内核到还没有操作系统的嵌入式板子上。在下载内核的过程中，开发主机作为 tftp 服务器，嵌入式系统通过一个监控程序（即 boot loader，Sitsang 开发板使用的是 RedBoot，Atmel 开发板用的是 U-Boot）把内核下载到 SDRAM 中，之后再将内核写入到 flash 里。

在 Ubuntu 系统中启动 TFTP 服务之前需要先修改文件/etc/default/tftpd-hpa，这个文件的内容默认如下：

```
#Defaults for tftpd-hpa
RUN_DAEMON="no"
OPTIONS="-l -s /var/lib/tftpboot"
```

Linux 系统中，把在后台运行的服务程序称为 daemon，所以很多服务器程序的名字最后都用 d 结尾，表示 “daemon”。这里要把 RUN_DAEMON=”no” 修改为 RUN_DAEMON=“yes”，否则 tftpd 服务是无法启动的。

从这个文件还可以看出 “OPTIONS” 部分是给 tftpd-hpa 传递的参数。“-l” 参数表示 tftpd 软件作为一个单独程序启动(standalone),而不是被 inetd 启动。“-s” 参数指定了 tftpd 软件的下载目录在 “/var/lib/tftpboot/” 中,即只有这个目录中的文件才可以被客户端下载。

最后执行命令:

```
# /etc/init.d/tftpd-hpa start
```

这个命令将启动 tftpd 服务。

上面的这个过程具有一定的普遍性,Ubuntu Linux 中的系统服务在/etc/init.d/ 目录中都有与之对应的启动脚本,多数与服务本身的名字相同,例如上面的 tftpd-hpa,这种脚本支持三种常用的参数:start、stop、restart,分别用来对其代表的服务进行启动、停止和重启。而/etc/default/ 目录中有很多服务程序的配置文件,主要对一些默认参数进行设置。

tftp 服务启动之后,可以复制一个文件到/var/lib/tftpboot 目录中,用 tftp 命令测试服务器是否配置正确。下面的命令假设/var/lib/tftpboot 目录中有 zImage 文件,在/home/embedded 目录下执行如下操作:

```
$ tftp
(to)
tftp> connect localhost
tftp> get zImage
tttp> quit
```

上面例子中的 “tftp>” 是 tftp 的提示符。“connect” 是连接主机的命令,后面接主机的名称,其中 “localhost” 代表本机。“get” 命令用来下载一个文件,其参数是下载文件的名字。“quit” 是退出 tftp 的命令。退出 tftp 之后可以在当前目录中用 ls -l 命令查看下载是否成功。注意比较原始文件与下载文件的长度。

2. 配置 NFS 服务

安装 nfs 的命令如下:

```
# apt-get install nfs-kernel-server
```

在正确安装了 nfsd 之后,修改它的配置文件/etc/exports,在里面输入如下一行:

```
/home   192.168.0.0/24(rw,no_root_squash,no_all_squash,no_subtree_check)
```

exports 文件的具体格式和含义可以参考 “man exports” 命令,上面的配置把 “/home” 目录共享出来,而且只有 IP 地址为 192.168.0.0/24 的主机可以访问。“192.168.0.0/24” 是一种常见的表示 IP 地址范围的方法,由于 IP 地址可以表示成 32 位的整数,其中 “/” 后面的 24 就表示 32 位整数中的前 24 位是确定的,后面 8 位可以随意。因此其表示的地址范围是 “192.168.0.1——192.168.0.254”。括号中表示的是共享的参数,“rw” 表示客户机可以读写服务器的共享内容,no_root_squash 和 no_all_squash 表示不对客户的用户 ID 进行重新映射。

然后执行命令

```
# /etc/init.d/nfs-kernel-server restart
```

重新启动 nfs 服务 (每次修改了系统服务的配置文件或者参数,都需要重新启动服务来使修改生效)。对于不熟悉 NFS 系统的用户,可以在本机检测一下 NFS 的配置是否生效:

```
# cd /
# mkdir test
# mount 192.168.0.101:/home/embedded/ test
```

这组命令在 “/” 目录下建立了 test 目录,然后用 mount 命令将共享的 NFS 目录挂载到 test 目录。命令中的 IP 地址是本机的地址。如果这个命令成功,应该可以在 test 目录中看到 embedded 目录中的内容。注意在本机挂载 NFS 目录的时候要避免 “循环挂载”,也就是不要把/home/embedded 目录挂载到其子目录中。

检验成功后就可以将 “/test” 目录卸载，然后删掉建立的临时目录 “test”：

```
# cd /
# umount test
# rmdir test
```

还可以使用 exportfs 命令检查当前所有的共享目录。例如：

```
# exportfs
/home                192.168.0.0/24
```

3. 配置 ftp 服务

在本机配置 ftp 服务会给嵌入式开发带来很多方便，安装 ftp 服务器仍使用 apt-get 命令：

```
# apt-get install vsftpd
```

ftp 服务器有很多种，我们这里使用的是 vsftpd。服务器的启动也是使用/etc/init.d/ 下的同名脚本程序：vsftpd。

```
# /etc/init.d/vsftpd start
```

它的配置文件是/etc/vsftpd.conf

三、交叉环境的建立

为了在 x86 架构的 Linux 主机上编译出能在 ARM 平台上运行的 Linux 内核和应用程序，我们必须使用一套交叉编译环境。下载文件中提供的 arm-eabi.tar.bz2 就是一套使用 crosstools-ng 生成的交叉编译工具。如果 “/usr/local/x-tools/” 目录不存在，说明还没有安装这个工具，此时用下面的命令把它安装到开发主机上：

```
# mkdir /usr/local/x-tools/
# cd /usr/local/x-tools/
# tar xfvj /cdrom/PreBuild/Toolchain/arm-eabi.tar.bz2
```

为了方便地使用这些工具，可以把可执行文件所在的路径加入到 PATH 环境变量中：

```
$ export PATH=/usr/local/x-tools/arm-none-linux-gnueabi/bin/:$PATH
```

此时输入

```
$ arm-none-linux-gnueabi-gcc
arm-none-linux-gnueabi-gcc: no input files
```

这表示交叉编译工具安装正确并且环境变量设置正确。

四、minicom 的使用

minicom 是 Linux 下面一个类似 Windows 中超级终端的工具，它可以通过串口和其他计算机或者外设进行通信。设置 minicom 的可以直接在终端中运行下面的命令：

```
$ minicom -s
```

minicom 是一个文本界面下的程序，上面的命令会打开如下的文本对话框：

```
        +-----[configuration]------+
        | Filenames and paths      |
        | File transfer protocols  |
```

```
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
+--------------------------+
```

直接在终端中运行 minicom 命令会打开一个文件界面的窗口，里面将显示从串口接收到的数据。在这个界面下按 “Ctrl_A+O” 组合键可以同样打开上图的配置界面，这个组合键的使用方法是先同时按下 “Ctrl” 和 “A”，放开所有键后再按 “O” 键。类似地，可以使用 “Ctrl_A+Z” 组合键来查看 minicom 的帮助。minicom 中，所有命令都是以 “Ctrl_A” 开始的。

上图为 minicom 的配置菜单，移动光标选择 “Serial port setup” 来设置通信串口。此时将打开如下图的配置界面。

```
+-----------------------------------------------+
| A -    Serial Device      : /dev/ttyS0        |
| B -  Lockfile Location    : /var/lock         |
| C -   Callin Program      :                   |
| D -  Callout Program      :                   |
| E -    Bps/Par/Bits       : 115200 8N1        |
| F - Hardware Flow Control : No                |
| G - Software Flow Control : No                |
|                                               |
|    Change which setting?                      |
+-----------------------------------------------+
```

在这个界面下按 “A”—“G” 键可以选择对应行的配置功能。例如按 “A” 键可以在 “Serial Device” 后面修改 minicom 所使用的串口设备名称，需要注意在 Linux 下面串口 1 的名称是/dev/ttyS0（在 Windows 下面为 COM1），串口 2 是/dev/ttyS1。硬件流控（Hardware Flow Control）和软件流控（Software Flow Control）都设置为 “No”。设置完成后，按回车键返回上一级菜单。按 “E” 键用来设置串口的波特率、奇偶校验和停止位的长度等参数。设置串口参数的窗口如下图所示。

```
+---------[Comm Parameters]----------+
|                                    |
|     Current: 115200 8N1            |
| Speed            Parity      Data  |
| A: <next>        L: None     S: 5  |
| B: <prev>        M: Even     T: 6  |
| C:   9600        N: Odd      U: 7  |
| D:  38400        O: Mark     V: 8  |
| E: 115200        P: Space          |
|                                    |
| Stopbits                           |
| W: 1             Q: 8-N-1          |
| X: 2             R: 7-E-1          |
|                                    |
| Choice,or <Enter> to exit?         |
+------------------------------------+
```

可以用 “:” 号前面的快捷键选择相应的功能，波特率（Speed）选择为 115200，奇偶校验（Parity）选择为 “N” 表示无奇偶校验，数据长度（Data）选择 8，停止位（Stopbits）长度为 1。这个配置参

数可以简写为 “115200 8N1”。

设置完毕可以按回车键退出到上级菜单，然后选择 “Save setup as dfl” 保存为默认设置。选择 “Exit” 退出菜单进入到 minicom 界面。需要记住的是退出 minicom 要使用命令 “Ctrl_A+Q”，与其他 minicom 命令类似，先要按下 “Ctrl_A”，抬起所有按键后再按 “Q”。如果直接关掉 minicom 所在的终端而没有先退出 minicom 的话，就有可能造成 minicom 没有正常退出，下次启动 minicom 会显示对应的串口已经被使用。实际上在实验中没有必要关闭 minicom，这样可以保留所有的操作记录，便于发生错误的时候查找原因。

其他比较有用的命令有：“Ctrl_A+W”，表示当 minicom 的显示内容超过一行的长度的时候，会自动进行换行操作，否则过长的内容就会被截断，截断的部分无法看到。这种方式叫做 Line Wrap，在输入较长的命令的时候也很有用。“Ctrl_A+B”，可以用来查看 minicom 的显示历史，可以用 “PgUp” 与 “PgDn” 按键进行翻页，最后用 “Esc” 键退出。

五、在嵌入式平台安装 Linux

本节实验使用基于 Atmel AT91SAM9261 的嵌入式平台，关于平台的硬件细节可以参考第四章的介绍，这里仅介绍如何在此嵌入式平台上安装 Linux 操作系统。

实验平台上包含两个 flash 设备：一个是串行 flash，在 Atmel 的数据手册和工具中被称为 dataflash；另外一个是 NAND flash 设备。dataflash 的容量比较小，在我们的实验中用来保存 CPU 的启动代码、U-Boot 代码和 Linux 内核代码。NAND flash 容量比较大，在本实验中用来保存嵌入式 Linux 的文件系统。由于 AT91SAM9261 的早期芯片不支持从 NAND flash 启动，所以 dataflash 不能被省略。

AT91SAM9261 处理器包含内部的 ROM，在芯片加电的时候，如果 BMS 引脚为高，这部分代码就会启动。如果在 dataflash 中没有安装任何可以引导的程序，ROM 代码就通过串口或者 USB 口（从设备）与外部程序交互，实现执行特定程序的功能（参考 4.3 节）。这个功能就叫做 SAM-BA（Boot Assistant），ATMEL 公司的同名软件可以利用这个功能实现硬件的测试和 flash 的写入。

实验平台上的 S14 跳线帽可以控制 dataflash 的片选信号。当 S14 被断开的时候，CPU 将找不到 dataflash 设备，因此必然进入 SAM-BA 模式。而如果 S14 被连接而 dataflash 中也有合法的启动代码的时候，CPU 将不会进入 SAM-BA 模式而直接运行启动代码。我们这里要重新写入 dataflash，因此摘掉 CPU 板上的 S14 跳线帽，并用 USB 线连接主机与开发板。此时打开设备电源，开发板将无法找到串行 flash 设备，系统将进入 SAM-BA 模式，等待串口或 USB 口的信号。本实验使用 USB 口进行通信，在 Linux 环境下，首先要加载 usbserial 的驱动，让 Linux 把接入的 USB 设备识别为串行通信设备。加载驱动的命令为 modprobe，具体的命令如下：

```
# modprobe usbserial vendor=0x03eb product=0x6124
```

上面命令中的 vendor 和 product 的设置就是 AT91SAM9 芯片的 USB 识别号，此时可以通过 lsusb 命令检查是否识别成功。

```
# lsusb -d 03eb:6124
Bus 004 Device 006: ID 03eb:6124 Atmel Corp
```

lsusb 命令用来查看系统中的 USB 设备，-d 参数指定了查询的设备识别号，如果该命令没有任何输出，就说明没有找到 USB 设备，需要检查 USB 连接，跳线帽设置，或者电源情况。

下载文件中提供的 SAM-BA 软件版本为 `sam-ba_cdc_2.9.linux.zip`，需要解压到工作目录中：

```
$ unzip /cdrom/Tools/sam-ba_cdc_2.9.linux.zip
```

以上命令假定下载文件保存的路径为 /cdrom，并用 unzip 命令将软件目录解压到当前的路径中。解压完成后当前目录中多了一个名为 sam-ba_cdc_2.9.linux 的目录，其中的 sam-ba 为其软件的主程序。

现在运行 SAM-BA 软件（需要使用 root 权限），提示窗口中应该可以正确识别连接所用的接口（图 2.3 中是 “/dev/ttyUSB0”），从提示窗口中选择板卡类型为 “AT91SAM9261-EK”，点击 “Connect” 连接实验板。

图 2.3: SAM-BA 建立连接界面

此时 SAM-BA 软件会初始化开发板的 SDRAM，成功后打开程序主界面，如图 2.4 所示。程序的功能和使用可以参考软件手册，这里只利用其脚本功能为开发板写入 bootloader 代码。

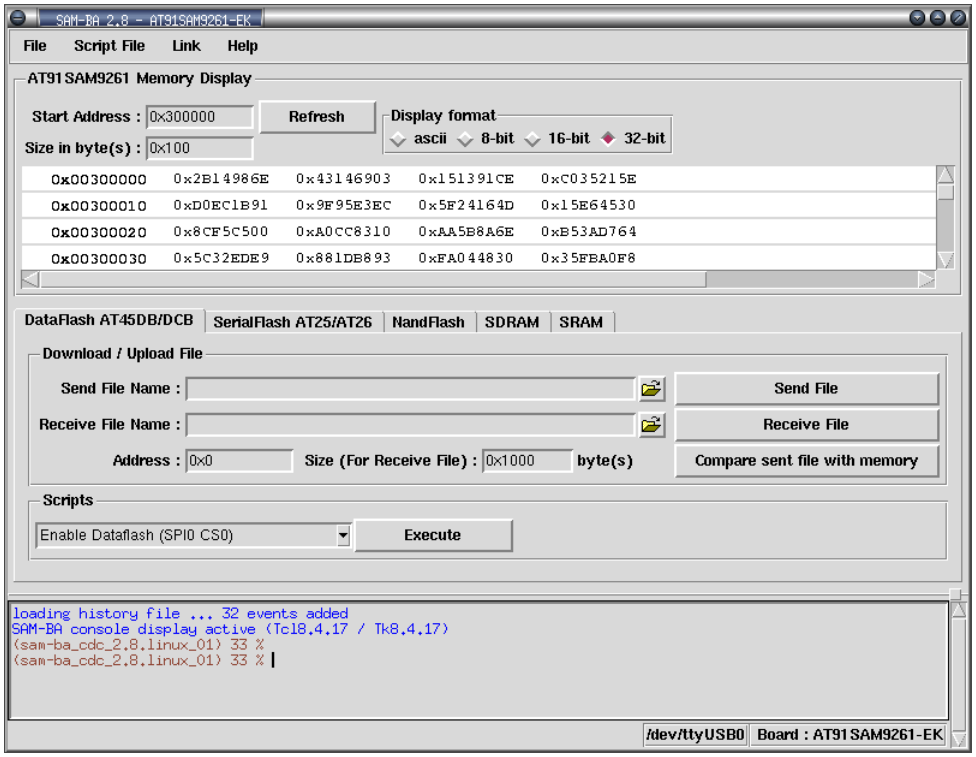

图 2.4: SAM-BA 工作界面

由于要操作 flash 设备，此时把 S14 跳线帽接好，SAM-BA 软件支持 tcl 脚本，可以从菜单 Script File 中选择 Execute Script 命令执行下载文件中提供的 install.tcl 脚本，完成 bootloader 和内核的安装。执行脚本之前要把需要的文件（见后面说明）复制到 sam-ba 软件的目录中。这个脚本的内容如下：

```
#############################################################
#  proc uboot\_env: Convert u-boot vars in a string
#     to flash in the region reserved for environment variables
#############################################################
proc set\_uboot\_env {nameOfLstOfVar} {
   upvar $nameOfLstOfVar lstOfVar

   # sector size is the size defined in u-boot CFG\_ENV\_SIZE
```

```
   set sectorSize [expr 0x4000 - 4]

   set strEnv [join $lstOfVar "\0"]
   while {[string length $strEnv] < $sectorSize} {
       append strEnv "\0"
   }
   set strCrc [binary format i [::vfs::crc $strEnv]]
   return "$strCrc$strEnv"
}

############################################################
#  Main script: Load the linux demo in DataFlash,
#               Update the environment variables
############################################################
set bootstrapFile        "dataflash_at91sam9261ek.bin"
set ubootFile                "u-boot.bin"
set kernelFile                "uImage"
set ubootEnvFile        "ubootEnvtFileDataFlash.bin"

## DataFlash Mapping
set baseAddr                0xC0000000
set bootStrapAddr        0x00000000
set ubootAddr                0x00008400
set ubootEnvAddr             0x00004200
set kernelAddr                0x00042000

# u-boot variable
set kernelUbootAddr [format "0x%08X" [expr $baseAddr + $kernelAddr]]
set kernelLoadAddr        0x20008000

## NandFlash Mapping
set rootfsAddr        0x00400000
set kernelSize        [format "0x%08X" [file size $kernelFile]]

lappend u_boot_variables \
   "ethaddr=3a:1f:34:08:54:54" \
   "bootdelay=3" \
   "baudrate=115200" \
   "stdin=serial" \
   "stdout=serial" \
   "stderr=serial" \
   "serverip=192.168.0.1" \
   "ipaddr=192.168.0.100" \

puts "-I- === Initialize the DataFlash access ==="
DATAFLASH::SelectDataflash AT91C_SPI0_CS0

puts "-I- === Erase all the DataFlash blocs and test ==="
DATAFLASH::EraseAllDataFlash

puts "-I- === Load the bootstrap in the first sector ==="
DATAFLASH::SendBootFile $bootstrapFile

puts "-I- === Load the u-boot in the next sectors ==="
send_file {DataFlash AT45DB/DCB} "$ubootFile" $ubootAddr 0

puts "-I- === Load the u-boot environment variables ==="
set fh [open "$ubootEnvFile" w]
```

```
fconfigure $fh -translation binary
close $fh
send_file {DataFlash AT45DB/DCB} "$ubootEnvFile" $ubootEnvAddr 0

puts "-I- === Load the Kernel image ==="
send_file {DataFlash AT45DB/DCB} "$kernelFile" $kernelAddr 0
```

这个脚本的 `lappend u_boot_variables` 部分命令设置了 U-Boot 的环境参数，可以根据需要进行修改。脚本文件保存在下载文件的 PreBuild/Images 目录中，在这个目录中还有 dataflash_at91sam9261ek.bin、u-boot.bin 和 uImage 文件，把这三个文件连同脚本复制到 SAM-BA 软件的目录中，利用软件 Script File 菜单下的 Execute Script File 命令执行这个脚本。脚本的执行过程要花一定的时间，注意观察下面窗口的提示信息（实际上下面的小窗口有一个最后是 % 的提示符，也可以输入执行常见的 shell 命令）。如果没有红色的错误信息提示就表示脚本执行成功。

脚本中用到了复制过来的文件，其中 dataflash_at91sam9261ek.bin 文件为启动代码，它被写入到了串行 flash 的 0 地址；u-boot.bin 文件是 U-Boot 的二进制代码，它被写入到了 0x8400 地址；uImage 文件是被 mkimage 命令封装过的 Linux 内核，它实际上只是添加了 Uboot 可以识别的头部信息的真正 Linux 内核：zImage，它被写入到了 0x42000 地址。最终串行 flash 中的地址分配如图 2.5 所示：

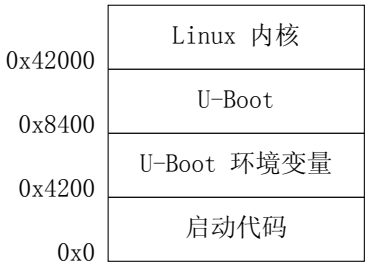

图 2.5: 串行 flash 布局

flash 写入成功后，用交叉串口线连接开发板的调试串口与微机的串口 1（/dev/ttyS0）。关闭 SAM-BA 软件，运行 minicom，然后重启开发板，就可以在 minicom 中看到 U-Boot 提示符。如果没有看到任何提示信息则需要检查串口线的连接、串口参数设置。如果只看到了 “RomBOOT” 则需要检测 S14 跳线的设置或者可能是 SAM-BA 写入没有成功。

U-Boot 被写入 flash 中后，会在芯片复位后自动运行。它将通过串口输出提示信息，这个信息就可以通过 minicom 软件显示，而用户在 minicom 中输入的按键也会被 U-Boot 软件通过串口接收，这样就实现了简单的用户交互。第 2.5 节中介绍的命令行界面都是通过这种方式进行的。因此不要把 minicom 理解为一个和嵌入式芯片直接相关的软件，它只是与芯片进行串口通信的工具而已。

接下来我们要在只安装了 U-Boot 和 Linux 内核的 flash 中安装一个完整的嵌入式 Linux。为此我们还需要的数据是 Linux 的文件系统。下载文件中提供的 at91.jffs2 是压缩为 JFFS2 格式的文件系统映像，下面的一系列命令将把这个文件系统写入到 NAND flash 中。

首先我们清空 NAND flash，避免遗留的数据对操作造成影响：

```
U-Boot> nand erase
```

如果有必要，还要设置开发板的 IP 地址（假定为 192.168.0.100）和开发主机的 IP 地址（假定为 192.168.0.1）。这两个参数在前面的 install.tcl 脚本中已经进行了设置：

```
U-Boot> setenv ipaddr 192.168.0.100
U-Boot> setenv serverip 192.168.0.1
```

可以通过 ping 命令检查网络连接是否正常：

```
U-Boot> ping 192.168.0.1
192.168.0.1 is alive
```

上面的命令显示 “192.168.0.1 is alive” 表示可以和这个 IP 地址建立通信，否则就要检测 IP 地址的设置和网线的连接是否正常。

写入文件系统先要把映像文件通过 TFTP 协议下载到开发板的内存，然后在用 U-Boot 的命令写入 NAND flash。先把 at91.jffs2 文件系统示例映像复制到 tftp 的主目录（/var/lib/tftpboot）中，然后执行 tftp 命令：

```
U-Boot> tftp 22000000 at91.jffs2
```

把文件系统写入到 flash 的特定位置上：

```
U-Boot> nand write.jffs2 0x22000000 0x00400000 0x12cc924
```

保存设置的环境变量：

```
U-Boot> saveenv
```

启动 Linux 检查设置是否正确：

```
U-Boot> boot
```

如果 Linux 可以正常启动，就说明安装过程没有问题，否则根据启动的情况检查出错的地方，完成实验内容。

第三章　ARM 体系结构与指令系统

现代通用计算机都遵循存储程序数字计算机的原理。存储程序的概念是由著名数学家冯 · 诺依曼等人在 1946 年总结并明确提出来的。其基本思想是计算机以中央处理器(central processing unit,CPU)为中心,把程序(指令)和数据放在同一个存储器中,指令和数据一样可以送到 CPU 中进行运算。相对于冯 · 诺依曼结构,还可以选择哈佛体系结构,它为程序和数据提供了各自独立的存储器。这样一来,程序和数据不必再竞争同一个端口,为数字信号处理提供了较高的性能,但很难在哈佛机上编写出一个自修改的程序。

通用处理器是一个执行存储器中指令的有限状态机。处理器的体系结构描述了从用户角度看到的处理器的属性,如指令集、可见寄存器、存储器管理表结构和异常处理模式等。那些用户看不到的处理器的结构、属性,如流水线结构、透明的高速缓存(cache)、步行表(table-walking)硬件及后备缓冲(translation look-aside buffer)则属于处理器组织的问题。

3.1　ARM 处理器概述

ARM 公司生产的 ARM 处理器作为通用处理器,经过近 20 年的发展,现已成为 32 位嵌入式 RISC 处理器的领导者,占据了小体积、低功耗、低成本和高性能的嵌入式应用领域的领先地位,广泛应用于手持计算、移动通信、汽车电子、工业控制及多媒体数字消费等各类产品市场。然而,ARM 公司既不生产芯片,也不销售芯片,它是一个设计公司,是知识产权(IP)供应商,靠转让设计许可由合作伙伴来生产各具特色的 ARM 芯片。现在,ARM 公司在全球有超过 100 家的合作伙伴,世界前 25 大半导体公司中有 23 家都采用了 ARM 的技术授权,从而促生了大量的第三方开发工具、软件和制造支持,保证了基于 ARM 处理器设计的产品可以成本低、速度快地投入市场。

ARM 处理器的发展历程:

- 1985 年,第一个 ARM 处理器在英国剑桥的 Acorn Computer 公司诞生,那时 ARM 代表 Acorn RISC Machine ;
- 20 世纪 80 年代后期,基于 ARM 处理器的 Acorn 微型计算机成为英国学校的主流机器;
- 1990 年,成立了 Advanced RISC Machine 公司,简称为 ARM 公司;
- 20 世纪 90 年代,ARM 公司推出 32 位嵌入式 RISC 处理器,居世界领先地位;
- 21 世纪初,ARM10、StrongARM、XScale 系列处理器的开发显著提高了 ARM 的性能,基于 ARM 处理器的嵌入式应用领域更为广阔。

ARM 体系结构的特点:

- 具有大量的寄存器,可用于多种用途;
- 固定长度(3 地址)指令格式;
- 寻址方式简单;
- Load-Store 指令集结构及多寄存器 Load-Store 指令,可以批量传输数据,提高传输效率;
- 每条指令都条件执行,提高指令的执行效率;
- 通过协处理器指令集来扩展 ARM 指令集;

- Thumb 16 位指令集，改善指令代码密度。

ARM 处理器芯片的应用领域：

- 手持无线设备，超过 85% 的无线终端设备（PDA、手机等）都采用了 ARM 技术；
- 蓝牙技术，有 20 多家公司的蓝牙产品采用了 ARM 技术；
- 联网设备，采用 ARM 技术的 ADSL 芯片组正在获得竞争优势；
- 消费电子，ARM 技术在数字音频播放器、数字机顶盒和游戏机等方面应用广泛；
- 汽车电子，基于 ARM 技术的驾驶、安全和车载娱乐设备也有广泛应用；
- 海量存储设备，采用 ARM 技术的存储产品，如硬盘、微型闪存卡和可读写光盘等；
- 图像处理，采用 ARM 技术的数字相机、打印机等；
- 安全产品，采用 ARM 技术的智能卡和 SIM 卡的安全使用。

3.2 ARM 指令集结构

计算机指令集结构的设计是计算机体系结构设计的核心问题，是软硬件功能分配最主要的界面，它历来是计算机体系结构的设计者、系统软件设计者和硬件设计者所共同关注的问题。

3.2.1 指令集设计

指令集是传统程序员所看到计算机体系结构的主要属性，它主要包括：指令集功能、寻址方式和指令格式。

一、指令集功能

大多数指令集结构所支持的操作类型如表 3.1 所示。所有的指令集结构一般都会对前三种类型的操作提供相应的指令，另外还会对基本的系统功能提供一些指令支持。

表 3.1: 指令集结构中操作的分类

操作类型	实例
算术和逻辑运算	整数的算术和逻辑操作：加、减、与、或等
数据传输	Load/Store 等
控制	分支、跳转、过程调用和返回等
系统	操作系统调用、虚拟存储器管理等
浮点	浮点操作：加、乘等
十进制	十进制加、乘、十进制到字符的转换等
字符串	字符串移动、比较、搜索等
图形	像素操作、压缩/解压缩操作等

二、Load-Store 指令集结构

根据在 CPU 中操作数的存储方法，指令集结构可分为堆栈结构、累加器结构和通用寄存器结构。通用寄存器结构是现代指令集结构类型的主流，这种结构又可细分为三种类型：寄存器 -寄存器型、寄存器 -存储器型和存储器 -存储器型。

- 寄存器 -寄存器型：算数逻辑单元（arithmetic logic unit，ALU）指令只包含寄存器操作数，不包含存储器操作数；
- 寄存器 -存储器型：ALU 指令包含寄存器和存储器操作数；
- 存储器 -存储器型：ALU 指令只包含存储器操作数，不包含寄存器操作数。

后两种结构类型中，指令中的操作数类型不同，指令字长多种多样，每条指令的执行时钟周期也不一样。这些多样性会增加编译器和 CPU 实现的难度，同时对存储器的频繁访问将导致存储器访问的瓶颈问题。

寄存器 -寄存器型结构中，指令格式简单，指令字长固定，各种指令执行时钟周期数相近，是一种简单的代码生成模型。该结构又称 Load-Store 指令集结构，是 RISC 体系结构的主要组成部分。其缺点是指令条数多，因而其目标代码较大。

三、寻址方式

在通用寄存器指令集结构中，一般利用寻址方式指明指令中的操作数是一个常数、一个寄存器操作数或者是一个存储器操作数。表 3.2 列出了当前指令集结构中所使用的一些操作数寻址方式。在指令集结构中采用多种寻址方式可以显著减少程序的指令条数，但同时也增加了实现的复杂度和使用这些寻址方式的指令的执行时钟周期数。因此，在指令集的设计中需对两者进行权衡。

表 3.2: 指令集结构中所使用的一些操作数寻址方式

寻址方式	指令实例	含义
寄存器寻址	ADD R4,R3	[R4] ← [R4]+[R3]
立即数寻址	ADD R4,#3	[R4] ← [R4]+3
寄存器移位寻址	ADD R4,100 (R1)	[R4] ← [R4]+Mem[100+R1]
寄存器间接寻址	ADD R4, (R1)	[R4] ← [R4]+Mem[R1]
索引寻址	ADD R3, (R1+R2)	[R3] ← [R3]+Mem[R1+R2]
直接寻址	ADD R1,(1001)	[R1] ← [R1]+Mem[1001]
存储器间接寻址	ADD R1,@(R3)	[R1] ← [R1]+Mem[Mem[R3]]
自增寻址	ADD R1, (R2)+	[R1] ← [R1]+Mem[R2] [R2] ← [R2]+d
自减寻址	ADD R1, -(R2)	[R2] ← [R2] - d [R1] ← [R1]+Mem[R2]
缩放寻址	ADD R1,100（R2）[R3]	[R1] ← [R1]+Mem[100+[R2]+[R3]*d]

四、指令格式

指令集中的每条指令由操作码和地址码组成，指令集结构的设计就是要确定操作码和地址码字段的大小及其组合方式，以及各种寻址方式的编码方法。寻址方式的表示在指令集结构中有着极其重要的位置。

通常，在指令集中有两种表示寻址方式的方法。一种是将寻址方式编码于操作码中，操作码同时描述指令的操作和相应的寻址方式；另一种是为每个操作数设置一个地址描述符，来表示该操作数的寻址方式。对 Load/Store 类型指令集结构，由于只有 1~3 个操作数，而且只有有限的几种寻址方式，一般采用将寻址方式编码于操作码中的方式。

指令集编码格式一般分为三种方式：

（a）变长编码格式

操作码	地址描述符 1	地址码 1	……	地址描述符 n	地址码 n

（b）固定长度编码格式

操作码	地址码 1	地址码 2	地址码 3

（c）混合型编码格式

操作码	地址描述符	地址码	
操作码	地址描述符	地址码 1	地址码 2
操作码	地址描述符 1	地址描述符 2	地址码

第一种是变长编码格式，这种编码方式适合于指令集结构包含多种寻址方式和操作类型时，多数 CISC 体系结构的指令集采用这种编码格式；第二种是固定长度编码格式，它将操作类型和

寻址方式组合编码在操作码中，所有指令长度是固定的，适合于寻址方式和操作类型较少时，一般 RISC 体系结构的指令集采用这种编码格式；第三种是混合型编码格式，其目的是通过提供一定类型的指令字长，期望能够兼顾降低目标代码长度和降低译码复杂度两个目标，Intel 80x86 指令集结构采用这种编码方式。

3.2.2 RISC 体系结构

1980 年以前，指令集设计的主要趋势是强化指令功能，实现软件功能向硬件功能转移，基于这种指令结构而设计实现的计算机系统称为复杂指令集计算机（complex instruction set computer，CISC）。1980 年，Patterson 和 Ditzel 对指令集结构的合理性进行了深入研究，研究结果表明，CISC 结构存在如下缺点：

- CISC 结构指令系统中，各种指令的使用频率相差悬殊。80% 的时间仅使用 20% 的指令；
- CISC 结构指令系统的复杂性带来了计算机系统的复杂性，增加了研制时间和成本；
- CISC 结构指令系统的复杂性是以有限的硅资源为代价，导致芯片面积增大；
- CISC 结构指令系统中许多复杂指令的复杂操作，反而使运行速度减慢；
- CISC 结构指令系统中指令的不均衡性，不利于使用先进的计算机体系结构技术（如流水线技术）来提高系统的性能。

针对上述缺点，Patterson 和 Ditzel 提出了精简指令计算机系统（RISC）的设想。这是开发高性能处理器过程中的一个进步。RISC 结构指令系统往往提供较少的精选出来的指令，使计算机体系结构更加简单、合理和高效。RISC 结构指令系统特点如下：

- RISC 结构指令系统使用固定（32 位）指令长度，可采用硬连线的指令译码逻辑；
- RISC 结构指令系统选取使用频率最高的指令，并补充一些有用的指令；
- RISC 结构指令系统每条指令功能简单，并在一个机器周期内执行；
- RISC 结构指令系统采用 Load/Store 结构；
- RISC 结构指令系统采用大的寄存器堆，所有寄存器可用于任何用途；
- RISC 结构指令系统使用流水线技术提高系统的性能；
- RISC 结构指令系统使得芯片面积小、功耗低、开发时间短、设计成本低。

RISC 结构指令系统由于采用固定指令长度，造成了其代码密度低。当应用在特定领域时，将导致在取指令时使用更大的存储器带宽，造成更高的存储器功耗。

3.2.3 ARM 指令集结构

一、ARM 指令集功能

ARM 指令集包含如下几类指令：

- 数据处理指令，如加、减和乘；
- Load/Store 存储器访问指令；
- 转移指令，程序的执行从一部分切换到另一部分；
- 协处理器指令，通过增加协处理器来扩展指令集；
- 异常及中断指令；
- 程序状态存储器（PSR）传输指令。

二、ARM 寻址方式

ARM 指令集支持如下几类寻址方式：

- 寄存器寻址：ADD R0,R1,R2 ；R0← R1+R2
- 立即数寻址：ADD R3,R3,#1 ；R3← R3+1

- 寄存器移位寻址：ADD R3,R2,R1,LSL #3 ；R3← R2+8*R1
- 寄存器间接寻址：LDR R0,[R1]；R0← [R1]
- 寄存器变址寻址：LDR R0,[R1,#4]；R0← [R1+4]
- 多寄存器寻址：LDMIA R1,{R0,R2,R5}；R0← [R1]；R2← [R1+4]；R5← [R1+8]
- 堆栈寻址：PUSH ；POP
- 相对寻址：B rel ；PC← (PC)+rel
- 块复制寻址

三、ARM 指令格式

ARM 指令集提供了简单的指令功能和支持简单的寻址方式，将其寻址方式编码在操作码中。为了简化指令译码，并充分发挥流水线的效率，所有 ARM 指令的字长均是 32 位，采用三地址指令格式。

四、采用 RISC 体系结构

ARM 体系结构采用了若干 RISC 体系结构的特征，如 Load/Store 体系结构、固定的 32 位指令和三地址指令格式。但有的 RISC 特征被 ARM 设计者所放弃，如寄存器窗口（因大量寄存器会占用很大的芯片面积）、延迟转移（简化异常处理）和所有指令单周期执行（尽管 ARM 大多数指令是单周期执行的）。ARM 体系结构的简单性在 ARM 的硬件组织和实现上比指令系统显得更明显。

ARM 体系结构采用了 RISC 体系结构的思想基础，即把简单的硬件和指令集结合起来，但它仍然保留一些 CISC 体系结构的特征。ARM 指令集比纯粹的 RISC 有更多的格式，这使它的指令译码较 RISC 体系结构复杂，但同时也带来了较高的代码密度，使得 ARM 体系结构获得了面积小、功耗低和高性能。

ARM 指令集的代码密度仍然不如 CISC 指令集的代码密度，同时 ARM 处理器主要应用于小型嵌入式系统，这些系统又要求较高的代码密度。ARM 公司在某些版本的处理器中，加入了一个称为 Thumb 结构的指令集。Thumb 指令集是原来 32 位 ARM 指令集的 16 位压缩方式，并在指令流水线中使用了动态解压缩硬件。Thumb 指令集扩展的这一优势，使得 ARM 的代码密度优于大多数 CISC 处理器的代码密度。应用程序中可以灵活地将 ARM 指令和 Thumb 指令混合编程。

3.3 ARM 流水线组织

在计算机体系结构设计中，为了提高处理器的处理速度，经常在处理器中采用流水线技术，这是一种性价比比较高的方法。

3.3.1 流水线技术

一、流水线技术

所谓流水线技术，是指将一个重复的时序过程，分解成为若干个子过程，而每一个子过程都可有效地在其专用功能段上与其他子过程同时执行。每个子过程称为流水线的 “级” 或 “段”，流水线的段数成为流水线的 “流水深度”。

考虑处理器的一条指令分为 6 个段：

1. 取指（FI）：从存储器读取指令；
2. 指令译码（DI）：确定操作功能；
3. 计算操作数地址（CO）：实现寻址；
4. 取操作数（FO）：从存储器中取操作数；
5. 执行指令（EI）：执行指令操作，结果写入寄存器；
6. 写操作数（WO）：结果存入存储器。

显然，处理器每单独执行一条指令需 6 个时钟周期，即指令的执行时间为 6 周期。为提高处理器的处理速度，处理器的操作可以这样来组织：当第一条指令刚执行完第一段时，下一条指令就开始执行第一段，如图 3.1 所示。理论上讲，这样的流水线应比没有重叠的指令执行快 6 倍。宏观上看，每条指令的执行时间为 1 周期，即吞吐率为 1 条指令/周期。

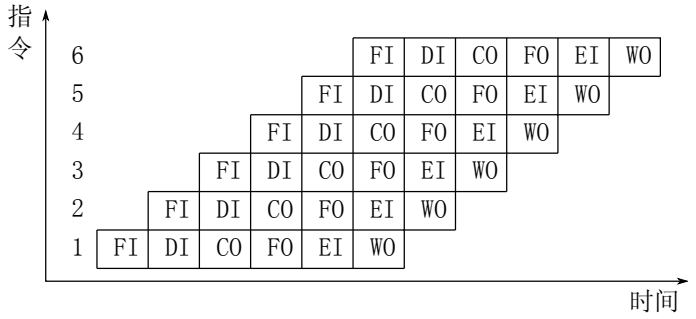

图 3.1: 流水线的指令执行

二、流水线中的相关

从图 3.1 中可以看到，如果流水线中的每条指令都相互独立，流水线将被填满，充分发挥流水线的功能。但实际中，流水线中流动的指令可能会相互依赖，即它们之间存在着相关关系。流水线的操作不会十分流畅，这些相关性会导致指令执行的停顿（stall），这些暂停周期称为 “流水线气泡”。一般来说，流水线中的相关主要分为以下三种类型：

1. 结构相关：当指令在重叠执行过程中，硬件资源满足不了指令重叠执行的要求，发生资源冲突。如两条指令的存储器访问冲突，后面执行的指令必须暂停；
2. 数据相关：当一条指令需要用到前面指令的执行结果，而这些指令均在流水线中重叠时，后面执行的指令必须暂停；
3. 控制相关：当流水线遇到分支指令或其他改变 PC 值的指令时，则会使后面紧接该条指令的几条指令的执行无效。

3.3.2 ARM 架构的流水线设计

ARM 公司在 1990~1995 年间开发的采用冯 · 诺依曼体系结构的 ARM6 和 ARM7，采用了 3 级流水线技术。1995 年后，ARM 公司推出了几个采用哈佛体系结构的 ARM 核（如 ARM9），采用了 5 级流水线技术。

一、ARM 架构的 3 级流水线

ARM 架构的 3 级流水线包括下列 3 段：

1. 取指：从存储器中取出指令，放入流水线。
2. 译码：指令被译码，并为下一周期准备数据通路的控制信号。这一级指令占用译码逻辑，不占用数据通路。
3. 执行：指令占用数据通路，寄存器被读取，操作数被移位，ALU 产生结果并回写到目的寄存器。

在任意时刻，可能有 3 种不同的指令占用这 3 级中的每一级，因此，每一级中的硬件必须能够独立操作。

ARM 3 级流水线的处理器由于采用冯 · 诺依曼体系结构，指令和数据放在同一个存储器中，其性能受到存储器带宽的限制。3 级流水线的执行（几乎）在每一个时钟周期都访问存储器，或者取指令，或者存取数据。存在严重的结构相关。

二、ARM 架构的 5 级流水线

ARM 架构的 5 级流水线处理器，采用了哈佛体系结构，具有分开的指令和数据存储器，改善了 3 级流水线中的结构相关的问题。同时，将指令的执行分为 5 段，减少了每个时钟周期内所完成的工作量，进而容许使用更高的时钟频率，明显地增加了处理器的性能。

ARM 架构的 5 级流水线包括下列 5 段：

1. 取指：从存储器中取出指令，放入指令流水线。
2. 译码：指令被译码，从寄存器堆中读取寄存器操作数。在寄存器堆中有 3 个操作数读取端口，大多数指令能在 1 个周期内读取其操作数。
3. 执行：把一个操作数移位，产生 ALU 结果。如果指令是 Load/Store，则在 ALU 中计算存储器的地址。
4. 缓冲/数据：如果需要，则访问数据存储器；否则，ALU 的结果只是简单地缓冲 1 个时钟周期，以便所有的指令具有同样的流水线流程。
5. 回写：将指令产生的结果回写到寄存器堆，包括从存储器重读取的数据。

三、ARM 流水线相关的控制

ARM 公司采用了一些与流水线相关的控制技术，来避免或减少流水线中相关的产生。

1. 结构相关控制技术

采用分离式指令 Cache 和数据 Cache，使取指和存储器的数据访问不再发生冲突，同时也解决了相应的数据通路问题。ALU 中采用单独加法器来完成地址计算，使执行周期的运算不再产生资源冲突。

2. 数据相关控制技术

定向（旁路或短路）技术。将某条指令产生的结果从其产生的地方直接送到后面的指令需要的地方，避免停顿的产生。例如，当定向硬件检测到前一个 ALU 运算结果的写入寄存器是当前 ALU 操作的原寄存器时，那么控制逻辑将前一个 ALU 的运算结果直接定向到 ALU 的输入端，后一个 ALU 操作数就不必从源寄存器中读取操作数。

3. 流水线互锁技术

当数据相关不能使用定向技术解决时，通过 “流水线互锁” 功能部件检测到数据相关，流水线暂停执行后面的所有命令，直到能通过定向技术来解决该数据相关为止。当然，这要在流水线中插入停顿，使指令序列的执行时间增加了 1 个时钟周期，但使流水线损失最小，保证指令的顺畅执行。

编译器的 “流水线调度” 技术。编译器在编译源程序时，通过优化、组织代码顺序来消除数据相关所带来的停顿。

4. 控制相关控制技术

指令预测技术。尽早预测、判断分支转移是否成功，从而采取相应的预测转移和延迟转移，以降低分支转移的损失.

尽早计算出分支转移的 PC 值（分支的目标地址）。在有些 ARM 架构处理器流水线的译码阶段，增加了一个专用的加法器来计算分支的目标地址。

3.4 ARM 存储器结构

存储器是计算机的核心部件之一，其性能直接关系到整个计算机系统性能的高低。如何以合理的价格，设计容量和速度满足计算机系统存储系统要求，始终是计算机体系结构中的关键问题之一。

3.4.1 存储器层次

一、存储器层次

现代处理器能以很高的速率执行指令，为了发挥其潜在的性能，必须配套一个合适的存储器系统。存储器有三个相互矛盾的主要指标：速度、容量和价格/位（简称价位）。一般来说，速度越快，价位越高；速度越慢，价位越低，但容量会越大。同时满足高速度、大容量、低价位的存储器是很难得到的。走出这种困境的唯一方法，是采用多种存储器技术，构成存储器层次，如图 1 所示。

M1、M2、…、Mn 为用不同技术实现的存储器。最靠近 CPU 的 M1 速度最快、价位最高、容量最小；而离 CPU 最远的 Mn 则相反。整个存储器系统要达到的目标是：从 CPU 来看，该存储

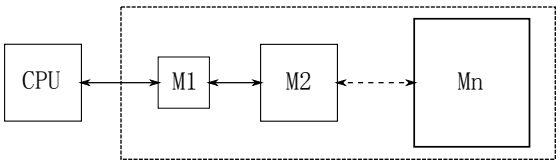

图 3.2: 多级存储器层次

器系统的速度接近 M1，而容量和价位接近于 Mn。该存储器系统满足这种目标是基于局部性原理，即绝大多数程序访问的指令和数据都是相对集中的。

设计存储器系统时，我们可以将 CPU 使用的程序和数据放在尽可能靠近 CPU 的存储器中。在存储器层次中，任何一层存储器中的数据一般都是下一层存储器中的数据的子集。越靠近 CPU 的存储器，CPU 对它的访问频率越高，若该层中没有 CPU 所需数据，则 CPU 再访问下一层，以此类推。

在计算机体系结构中，存储器的层次主要体现在缓存—主存和主存—辅存两个存储层次上，见图 3.3 所示。缓存一般由速度快、容量小的 Cache 组成；主存一般由 DRAM 等组成，其速度低于缓存，容量大于缓存，主存与缓存间的数据调动是由硬件自动完成的，对程序员是透明的；辅存一般由硬盘、磁带等组成，其容量比主存大得多，速度则比主存慢得多，CPU 不能直接访问辅存，辅存与主存间的数据调动是由硬件和操作系统来实现。

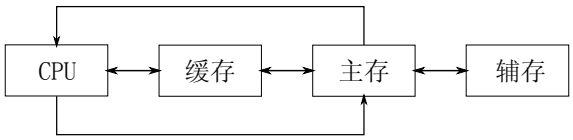

图 3.3: 两种存储层次

二、ARM 架构存储器层次

ARM 架构处理器通常应用于嵌入式系统中，一般没有辅存，主要采用缓存—主存层次。典型的 ARM 架构存储器层次如下：

- 处理器中的寄存器组，其访问时间为几个 ns ；
- 片上 Cache 存储器，容量在 4~32KB 之间，访问时间约为 10ns ；
- 主存储器 DRAM，容量在几 MB 到几百 MB，访问时间约为 100ns。

3.4.2 Cache

为了填补 CPU 和主存之间在速度上的差距，现代计算机都在 CPU 和主存之间设置一个高速、小容量的缓存 Cache。Cache 是按块进行管理的，Cache 和主存均被分割成大小相同的块，数据以块为单位调入 Cache。由于 Cache 所保存的主存内容是变化的，因此 Cache 必须同时保存数据及其在主存中的地址。

一、映像规则

当要把一个块由主存调入 Cache 时，就有个如何放置的问题。由主存地址映像到 Cache 地址称为地址映像，一般有 3 种映像方式：

1. 直接映像。图 3.4 示出了直接映像方式中，主存与 Cache 中字块的对应关系。

- 主存和 Cache 被分为大小相同的块，每块大小为：2^b 字节。
- 主存地址分为：(m+b)位，m=t+c，c 位指 Cache 的字块地址，t 位指 Cache 中主存字块标记，b 位指字块内选择字节，则主存有 2^m 个字块，Cache 中有 2^c 个字块。
- 每个主存块与唯一一个 Cache 块相对应，映像关系：Cache 块号 = 主存块号 mod 2^c。
- 缓存接到 CPU 来的主存地址后，根据 c 位字段找到相应的 Cache 字块，再根据该字块标记与主存地址的高 t 位比较，若符合且有效，则表示所要的主存数据已在 Cache 中，即已命中，可

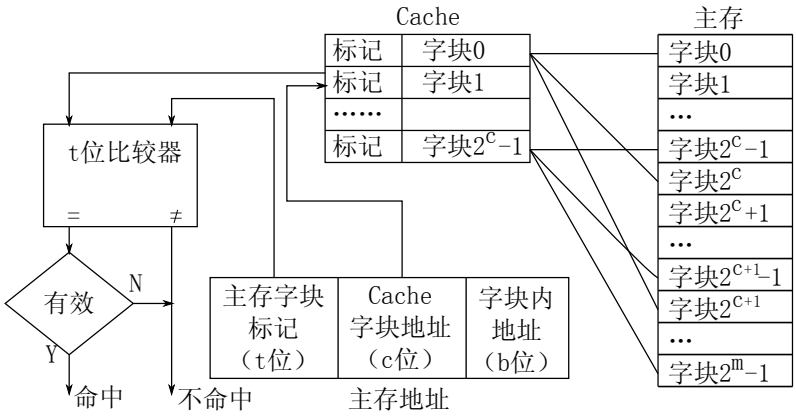

图 3.4: 直接映像方式

根据 b 位地址从 Cache 中取得数据；若不符合或无效，表示不命中，则从主存中读入数据，并填入 Cache 中相应的字块。

该方式的优点是实现简单，Cache 访问速度最快。但缺点是每个主存块只能固定地对应某个缓存块，即使其他缓存块空着也不能占用，不能高效地使用缓存的存储空间。

2. 全相联映像。图 3.5 给出了全相联映像方式中的主存与 Cache 中字块的对应关系。

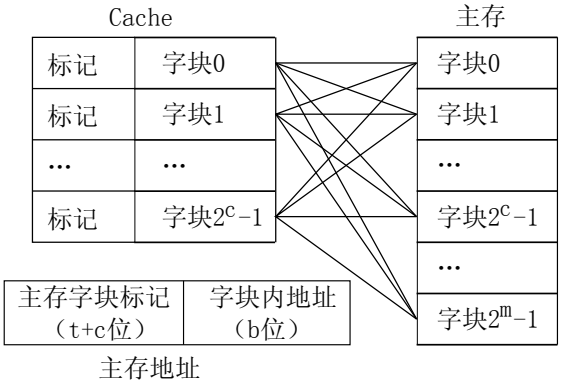

图 3.5: 全相联映像方式

- 主存和 Cache 被分为大小相同的块，每块大小为：2^b 字节；
- 主存地址分为：(m+b)位，m=(t+c)位指 Cache 中主存字块标记，b 位指字块内选择字节，则主存有 2^m 个字块，Cache 中有 2^c 个字块；
- 每个主存块可映像到 Cache 中的任意一个字块；
- 缓存接到 CPU 发送来的主存地址后，根据主存地址的高 m 位与缓存中标记比较，若符合且有效，则表示所要的主存数据已在 Cache 中，即已命中，可根据 b 位地址从 Cache 中取得数据；若不符合或无效，表示不命中，则从主存中读入数据，并填入 Cache 中的字块。

该方式的优点是灵活，命中率高，缩小了块冲突，可高效地使用缓存的存储空间；但缺点是 Cache 标记位数多，每次访问 Cache 要和全部标记位进行比较，这种比较通常采用 “按内容寻址” 的相联存储器(CAM)来完成，需要的逻辑电路多，成本较高，速度较慢。

3. 组相联映像。组相联映像是对直接映像方式和全相联映像方式的一种折衷。图 3.6 所示给出了组相联映像方式中，主存与 Cache 中字块的对应关系。

- 主存和 Cache 被分为大小相同的块，每块大小为：2^b 字节。
- 主存地址分为：(m+b)位，m=s+q，s=t+r，q=c-r。整个 Cache 空间被分为 2^q 个组，每个组内包含 2^r 个块(路)，s 位指 Cache 中主存字块标记，b 位指字块内选择字节，主存有 2^m 个字块，Cache 中有 2^c 个字块。
- 每个主存块与唯一一个组相对应(直接映像特征)，映像关系：组号 = 主存块号 mod 2^q。

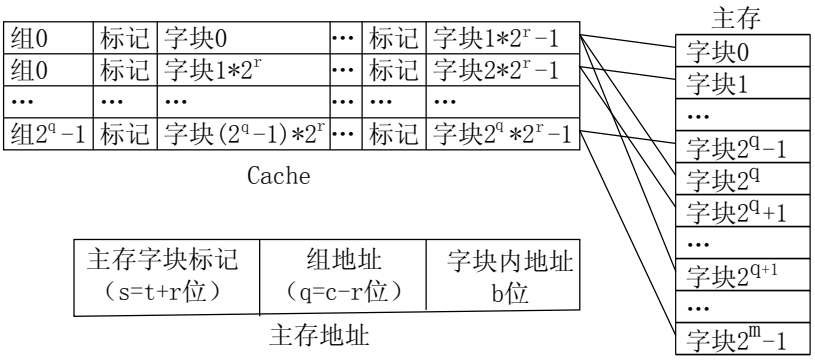

图 3.6: 组相联映像方式

该块可被放入这个组中的任何一个位置（全相联映像特征）。r=0 时，它就是直接映像方式，r=c 时，它就是全相联映像方式。

• 缓存接到 CPU 发送来的主存地址后，根据 q 位字段找到相应的 Cache 组，再根据该字块标记与主存地址的高 s 位 “按内容寻址” 进行比较，若符合且有效，则表示所要的主存数据已在 Cache 中，即已命中，可根据 b 位地址从 Cache 中取得数据；若不符合或无效，表示不命中，则从主存中读入数据，并填入 Cache 中的字块。

组相联映像方式的性能及其复杂性介于直接映像和全相联映像两者之间。每组中含的路数越多，则相联度越高，Cache 的失效率就越低。但分析结果表明：每组中含的路数多到一定程度时，再提高每组中的路数对系统性能的改善已不明显。

二、替换算法

当新的主存块需要调入 Cache 中时，会出现该块映像到的一组（或一个）Cache 块已全被占满的情况，这时就产生了一个替换算法（策略）问题。对于直接映像，Cache 中的替换别无选择，只有一个块；对于全相联映像和组相联映像，Cache 中有多个可选择的块，其替换算法主要有三种：

1. 随机法：这种方法随机地选择被替换的块，均匀使用一组中的各块。该方法简单、易于硬件实现；但没有考虑 Cache 块过去被使用的情况，没有反映程序的局部性，因此，失效率高。

2. 循环法：这种方法类似于先进先出（FIFO）法，选择最早调入的块作为被替换的块。该方法简单、易于硬件实现；其缺点是可能把一些经常使用的程序块（如循环程序）也作为最早进入 Cache 的块而被替换出去。

3. 最近最少使用法 LRU：这种方法选择近期最少被访问的块作为被替换的块。该方法最能体现程序的局部性原理，命中率最高；但该方法复杂，难以用硬件实现，实际中经常只是近似地实现。

三、写策略

处理器对 Cache 的访问主要是读访问，对所有指令 Cache 的访问都是 “读”，对数据 Cache 的访问多数操作也是 “读”。统计表明，处理器对存储器的 “写” 操作只占所有操作的 10% 左右。处理器对存储器的写策略主要有两种方法：

1. 写直达法：这种方法在 “写” 操作时不仅把数据写入 Cache 中相应的块，而且也写入主存中相应的块。采用该方法时，在写操作时 CPU 必须降到主存的速度，等待写操作的结束。减少写等待的一种常用优化技术是采用写缓冲器（write buffer），CPU 将数据写入该缓冲器，就可以继续执行其他任务，写缓冲器再以主存速度将数据传送到主存中。该方法实现简单，主存随时更新。

2. 写回法：这种方法每次只把数据写入 Cache 相应的块中，该块只有在被替换时才被写回主存。为了减少在被替换时的写回，常在 Cache 中设置污染位标志。如果有新的数据要调入一个已污染的块中，那么该块必须先写回主存。该方法速度快，而且在最终值写回主存之前，一个位置可以多次写入，降低了对外部写带宽的要求。但实现起来较复杂，而且 Cache 和主存的不一致性也难于管理。

四、ARM 架构的 Cache 设计

1989 年，ARM 公司生产的 ARM3 首次使用了片上 Cache，极大地提高了 ARM 处理器的性能。ARM 架构的 Cache 组织一般采用全相联映像和组相联映像方式，替换算法一般采用随机法和循环法，写策略一般采用写直达法和写回法。下面以基于 ARM 架构的 Intel XScale 的 Cache 组织为例加以说明。

ARM 架构的 Intel XScale 的 Cache 组织采用了分开的指令和数据 Cache。它们都采用组相联映像方法，32Kbytes 的 Cache 空间被分为 32 组，每组有 32 路（块），每路有 8 个字（32bytes）。

虚拟地址的最低 2 位用于选择一个字中的字节，2～4 位用于选择组中的块，5～9 位用于选择 Cache 中的组，其他位用于主存字块标记。替换算法采用循环法，写策略采用写直达法和写回法。写直达法中有 8 项写缓冲器，每项有 16bytes；写回法中每块有 2 位染位标志，一位用于低 16bytes，另一位用于高 16bytes。两种写策略可通过内存管理单元（memory management unit，MMU）进行设置管理。

3.4.3 存储器管理

计算机系统中经常同时运行多个进程，每个进程都有自己的地址空间。如一个程序可以有自己的代码段、数据段和堆栈段。由于每个程序只用到全地址空间的一小部分，因此可以将全地址空间划分为较小的块（页面或段），并以块为单位分配给各进程。进程间的切换是由存储器管理单元支持的，通常分为段式管理和页式管理。程序运行时，必须将程序空间中给出的逻辑地址映像到主存的物理地址。

一、段式管理

段式管理将存储器划分为可变长的块，称为段。段的最小长度为 1 个字节，最大长度因机器而异，通常为 2^{16}～2^{32} 字节。由于段的长度是可变的，其逻辑地址一般用两个字表示：一个为段号，另一个为段内位移。逻辑段和物理段之间的关系保存在段表（数据结构）中，逻辑地址到主存的物理地址的转换如图 3.7 所示。

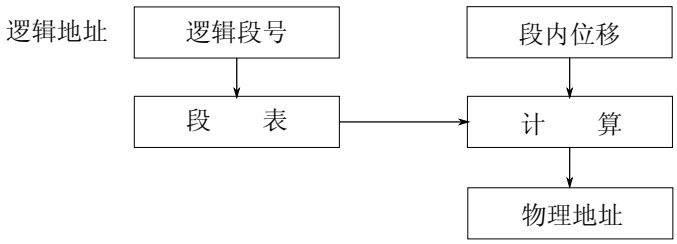

图 3.7: 段式存储器管理方法

二、页式管理

页式管理将存储器划分为固定大小的块，称为页，通常为 4 ～ 64KB。其地址都是单一、固定长度的地址字，由页号和页内位移两部分组成。逻辑页和物理页之间的关系保存在页表中，逻辑地址到主存的物理地址的转换如图 3.8 所示。

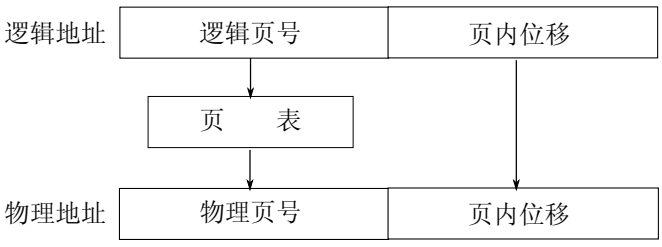

图 3.8: 页式存储器管理方法

段式管理会形成存储器碎片而造成段外存储器空间的浪费，页式管理会由于页内未满而造成页内存储器空间的浪费。现在大多数处理器都使用页式管理方法。

三、地址转换后备缓冲器

段式管理或页式管理中的段表或页表都保存在主存中，它们是以逻辑段号或逻辑页号作为索引的数据结构，含有所要查找的物理地址。因此，每次访存都要对主存进行两次访问：一次是读取段表或页表得到所需的物理地址，第二次才访问数据本身。这在实际使用中对系统的性能影响极大。

一般采用快表（translation lookaside buffer，TLB）来解决这个问题。以页式管理为例，TLB 是一个专用的高速缓冲器，用于存放最近经常用到的页表项，即页表 Cache。与数据和指令 Cache 一样，页表 Cache 也包括标记和数据。标记中存放的是逻辑页号，数据中则存放物理页号和其他一些控制信息，TLB 采用全相联映像方式。这样，在地址变换中，一般直接查 TLB 就可以了。只有偶尔在 TLB 不命中时，才需要访问主存中的页表。使用 TLB 进行地址转换见图 3.9。

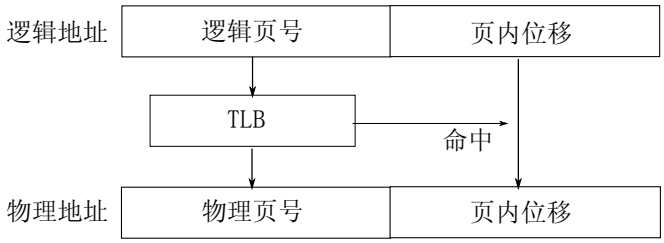

图 3.9: TLB 进行地址转换

四、ARM 体系结构中的 MMU

ARM 体系结构中的 MMU 主要完成 3 个基本功能：

- 将虚拟地址（逻辑地址）转换为物理地址；
- 控制存储器访问权限，中止非法访问；
- 设置指令 Cache 和数据 Cache 策略。

ARM 处理器通过访问其片上的系统控制协处理器（CP15）的各个寄存器，来协助完成以上 3 个基本功能。

1. 存储器粒度。根据不同的应用方式，ARM 架构中的存储器粒度（granularity）可使用如下的单位：

- 段（section）：为 1MB 的存储器块。
- 大页：为 64KB 的存储器块。在大页中，将访问控制施加到单独的 16KB 页上。
- 小页：为 4KB 的存储器块。在小页中，将访问控制施加到单独的 4KB 页上。
- 微页（ting page）：某些最新的 CPU（如 Intel XScale）支持 1KB 的微页。

页面的大小是存储器的重要参数之一。页表的大小与页面大小成反比，较大的页面可以节省实现地址转换所需要的存储空间，使 TLB 实现简单、快速转换；而较小的页面可以节省存储器空间。ARM 架构通常采用的粒度为 4KB 小页。

2. 页转换。ARM 一般采用页式管理方法。简单计算表明，用单个表保存页表转换关系需要一个很大的表：如果一页为 4KB，那么 32 位地址中的 20 位必须被转换，将需要大小为 2.5MB 的表，这显然不可接受。多数 ARM 页式系统使用两级页表。如地址的高 10 位用于确定一级页表目录中相应的二级页表；地址的次高 10 位确定物理页数，则两级页表共需要 8KB 的存储空间即可管理 4MB 的主存。

ARM MMU 具有一个保存最近使用的页转换的 TLB。若 TLB 未命中，则需到主存中访问页表，该访问通过步行表（table-walking）硬件实现。对于指令和数据 Cache 分开的处理器，可能有分开的指令和数据 TLB。ARM 页转换见图 3.10。

五、ARM 体系结构中存储器访问机理

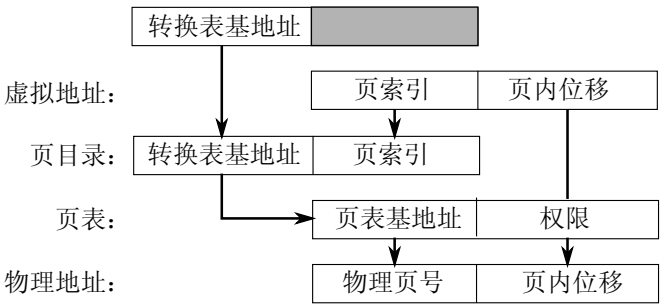

图 3.10: ARM 页转换流程

参考图 3.11，ARM 处理器需要访问存储器时：

- 根据程序中的虚拟地址首先访问相应的 Cache，若命中，则读取 Cache 中相应数据；
- 若未命中，则通过 ARM MMU 根据权限访问主存：首先进行虚拟地址到物理地址间的转换，先访问 TLB，若命中，则根据权限读取主存中相应数据；
- 若未命中，则通过步行表硬件访问主存页表进行地址转换，再读取主存中相应数据；
- Cache 读取硬件通过物理地址访问主存，更新 Cache 中相应的块；
- 主存访问故障将通过访问控制硬件通知处理器；
- 主存写操作通过 Cache 写策略和物理地址访问主存。

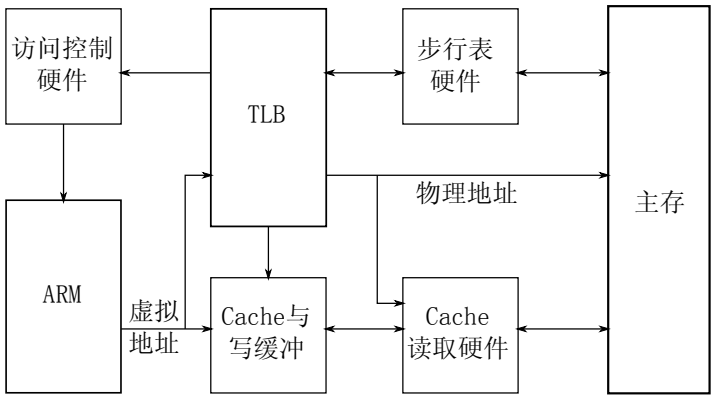

图 3.11: ARM 体系结构中存储器访问流程

3.5　ARM 寄存器组织

处理器指令集定义了操作，程序员可以用这些操作来改变包含了处理器系统的状态。这些状态是由处理器可见寄存器和系统存储器中数据项的值构成的。每一条指令可以看作是完成从指令执行前的状态到指令完成后的状态的转换。尽管典型的处理器有许多不可见寄存器参与指令的执行，但在指令执行前后这些寄存器的值不重要，只有可见寄存器中的值才有意义。可见寄存器属于计算机体系结构的范畴。

3.5.1　ARM 处理器模式

ARM 体系结构处理器支持表 3.3 所列的 7 种处理器模式。

除了用户模式之外的其他 6 种处理器模式称为特权模式（privileged modes）。特权模式中除系统模式外的其他 5 种处理器模式，又称为异常模式。

大多数应用程序都运行在用户模式下。在此模式中，程序不能访问某些被保护的系统资源，也不能直接改变其运行模式；在特权模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式的改变。

表 3.3: ARM 处理器的 7 种处理器模式

处理器模式	描述
用户模式(usr)	正常程序执行模式
快速中断模式(fiq)	用于高速数据传输合通道处理
外部中断模式(irq)	用于通常的中断处理
管理模式(svc)	供操作系统使用的一种保护模式
中止模式(abt)	用于虚拟存储或存储器保护
未定义模式(und)	用于通过软件仿真硬件的协处理器
系统模式(sys)	用于特权级的操作系统任务

处理器模式可以通过异常发生及处理过程进行切换,也可以通过软件控制进行切换。这种体系结构可以使操作系统控制整个系统的资源。

当应用程序发生异常时,处理器进入相应的异常模式。每一种异常模式中都有一组寄存器,供相应的异常处理程序使用。这样就可以保证在进入异常模式时,用户模式下的寄存器不被破坏。

用户模式与系统模式具有完全一样的寄存器。系统模式属于特权模式,不受用户模式的限制,可以访问所有的系统资源,也可以直接进行处理器模式的切换。系统模式不可以由异常过程进入,它主要供操作系统任务使用。这样可以保证当异常发生时,任务状态不被改变。

3.5.2 ARM 状态下的寄存器

ARM 处理器有两种工作状态:

- ARM:32 位,执行字对准的 ARM 指令;
- Thumb:16 位,执行半字对准的 Thumb 指令。

ARM 和 Thumb 之间状态的切换不影响处理器模式或寄存器的内容。

ARM 状态下处理器总共有 37 个寄存器:

- 31 个通用寄存器,包括程序计数器(PC)。这些寄存器都是 32 位的;
- 6 个状态寄存器,这些寄存器也是 32 位的,但只使用了其中的 12 位。

ARM 处理器共有 7 种处理器模式,在每一种处理器模式中有一组相应的寄存器组,如表 3.4 所示。在任何时候,15 个通用寄存器(R0~R14)、1 或 2 个状态寄存器和程序状态寄存器都是可见的。

阴影部分表明用户或系统模式使用的一般寄存器已被异常模式的另一寄存器所取代

一、通用寄存器

通用寄存器(R0~R15)可以分为以下 3 类:

- 不分组寄存器 R0~R7 ;
- 分组寄存器 R8~R14 ;
- 程序计数器 R15(PC)。

R0~R7 是不分组寄存器。这意味着在所有处理器模式下,指的都是同一个物理寄存器。它们是真正的通用寄存器,没有被系统用于特别的用途。

R8~R14 是分组寄存器。每个寄存器对应两个以上的物理寄存器,所访问的物理寄存器取决于当前的处理器模式。若要访问特定的物理寄存器而不依赖当前的处理器模式,则要使用规定的名字。

寄存器 R8~R12 各有两组物理寄存器:一组为 fiq 模式,另一组为除 fiq 以外的模式。如在 fiq 模式下,使用 R8-fiq~R12-fiq ;在用户模式下,使用 R8-usr~R12-usr 等。系统没有将寄存器

表 3.4: ARM 状态下的寄存器

用户模式	系统模式	管理模式	中止模式	未定义模式	中断模式	快中断模式
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8-fiq
R9	R9	R9	R9	R9	R9	R9-fiq
R10	R10	R10	R10	R10	R10	R10-fiq
R11	R11	R11	R11	R11	R11	R11-fiq
R12	R12	R12	R12	R12	R12	R12-fiq
R13	R13	R13-svc	R13-abt	R13-und	R13-irq	R13-fiq
R14	R14	R14-svc	R14-abt	R14-und	R14-irq	R14-fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR-svc	SPSR-abt	SPSR-und	SPSR-irq	SPSR-fiq

R8~R12 用于任何特殊用途，但当中断处理非常简单时，仅仅使用寄存器 R8~R12，不必进行现场保护和恢复，可实现快速中断处理。

寄存器 R13~R14 各有 6 组物理寄存器。1 个用于用户模式和系统模式，其他 5 个用于 5 种异常模式。采用下面的名字形式来区分各个物理寄存器：

```
R13<mode>,R14<mode>
```

其中 <mode> 可以是下面几种模式之一：usr、svc、abt、und、irq 和 fiq。

R13 通常用作堆栈指针，称为 SP。每种异常模式都有自己的物理 R13，应用程序初始化该 R13，使其指向该异常模式专用的栈地址。当进入异常模式时，异常处理程序将其他寄存器的值保存在 R13 所指的栈中；当退出异常处理程序时，重新将栈中的寄存器值弹出。这样就使异常处理程序不会破坏被其中断了的程序的运行现场。

R14 用作子程序链接寄存器，也称链接寄存器 LR。在 ARM 体系中有两种特殊用途：

- 每一种处理器模式自己的 R14 用于存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，R14 被设置成该子程序返回地址。在子程序中，当把 R14 的值复制到程序计数器 PC 中时，子程序即返回。
- 当异常中断发生时，该异常模式特定的物理 R14 被设置成该异常模式将要返回的地址（某些异常有一个常数的偏移量），异常返回的方式与子程序返回方式基本相同。

R15 用作程序计数器，称为 PC。由于 ARM 状态指令是字对齐的，PC 值的第 0 位和第 1 位总是为 0。

在 PC 值的读操作中，由于 ARM 结构采用了流水线机制，读取的 PC 值为当前指令地址加上 8 个字节。也就是说，对于 ARM 指令集，PC 指向当前指令的下两条指令的地址。读取的结果值的位 [1:0] 始终是 0。读 PC 值主要用于快速地对临近的指令和数据进行位置无关寻址，包括程序中的位置无关分支。

在 PC 值的写操作中，当成功地向 R15 写入一个地址时，程序将跳转到该地址执行。通常希望写到 R15 中值的位 [1:0]=0b00，具体要求 ARM 各版本有所不同：

- ARM 体系结构版本 3 及以下版本，写到 R15 中值的位 [1:0] 被忽略，即写入 R15 的地址值将与 0xFFFFFFFC 做与操作；
- ARM 体系结构版本 4 及以上版本，程序必须保证写到 R15 中值的位 [1:0] =0b00，否则将产生不可预知的后果。

二、程序状态寄存器

程序状态寄存器有 CPSR（当前程序状态寄存器）和 SPSR（程序状态保存寄存器）。CPSR 可在任何处理模式下被访问；每一种异常处理模式下都有一个专用的物理寄存器 SPSR，用于异常发生时保存 CPSR 中的内容。当异常中断程序退出时，可以用 SPSR 中保存的值来恢复 CPSR。在用户模式或系统模式中不可访问 SPSR，否则将产生不可预知的后果。

CPSR 和 SPSR 具有相同的格式，如下：

31	30	29	28	27	26 8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	DNM（RAZ）	I	F	T	M4	M3	M2	M1	M0

N（Negative）、Z（Zero）、C（Carry）和 V（oVerflow）统称为条件标志位。大部分的 ARM 指令可以根据 CPSR 中的这些条件位决定指令是否执行。各条件标志位的具体含义如表 3.5 所示。

表 3.5: CPSR 中的条件标志位

标志位	含义
N	若结果是带符号二进制补码，N=1 表示运算结果为负数；N=0 表示运算结果为整数或零
Z	Z=1 表示运算结果为零；Z=0 表示运算结果不为零。用于比较指令，Z=1 表示运算结果两数大小相等
C	在加法指令中（包括比较指令 CMN），运算结果产生进位（无符号上溢出）时 C=1，否则 C=0 ； 在减法指令中（包括比较指令 CMP），运算结果产生借位（无符号下溢出）时 C=0，否则 C=1 ； 对于包含移位操作的非加法/减法运算指令，C 为移出值的最后一位； 对于其他非加法/减法运算指令，C 的值通常不改变
V	对于加法/减法运算指令，当操作数和运算结果为二进制补码形式的带符号数时，V=1 表示符号位溢出；对于非加法/减法运算指令，V 的值通常不改变

通常以下指令会改变 CPSR 中的条件标志位：

- 比较指令：如 CMP、CMN、TEQ、TST 等；
- 一些算术运算和逻辑运算指令的目标寄存器不是 R15 时；
- MSR 指令可以向 CPSR/SPSR 中写入新值；
- MRC 指令将 R15 作为目标寄存器时，可以把协处理器产生的条件标志位的值传送到 ARM 处理器；
- 一些 LDM 指令的变种指令可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回；
- 一些带 “位设置” 的算术和逻辑指令的变种指令，也可以将 SPSR 的值复制到 CPSR 中，这种操作也主要用于从异常中断程序中返回。

CPSR 中的位 [27] 称为 Q 表示位。用于在 ARM v5 及以上版本的 E 系列处理器中，指示在增强型 DSP 指令中是否发生了溢出或饱和。

CPSR 中的低 8 位 I、F、T 及 M[4:0] 称为控制位。当异常中断发生时会改变控制位；在处理器特权模式下，通过软件也可以改变这些位。

- 中断禁止位：

I=1 则禁止 IRQ 中断；
F=1 则禁止 FIQ 中断。

- T 控制位：

对于 ARM v4 及以上版本的 T 系列处理器：
T=0 表示执行 ARM 指令；
T=1 表示执行 Thumb 指令。
对于 ARM v5 及以上版本的非 T 系列处理器：
T=0 表示执行 ARM 指令；
T=1 表示强制下一条指令产生未定义指令异常。
对于 ARM v3 及以下版本及 ARM v4 的非 T 系列处理器：
T 控制位应为 0。
处理器模式控制位：

CPSR 中的位 M[4:0] 称为模式位，用于控制处理器模式。具体含义如表 3.6所示。

表 3.6: 模式位 M[4:0] 的含义

M[4:0]	处理器模式	可访问的寄存器
0b10000	用户模式	PC、R14∼R0、CPSR
0b10001	fiq 模式	PC、R14-fiq∼R8-fiq、R7∼R0、CPSR、SPSR-fiq
0b10010	irq 模式	PC、R14-irq∼R13-irq、R12∼R0、CPSR、SPSR-irq
0b10011	管理模式	PC、R14-svc∼R13-svc、R12∼R0、CPSR、SPSR-svc
0b10111	中止模式	PC、R14-abt∼R13-abt、R12∼R0、CPSR、SPSR-abt
0b11011	未定义模式	PC、R14-und∼R13-und、R12∼R0、CPSR、SPSR-und
0b11111	系统模式	PC、R14∼R0、CPSR（ARM v4 及以上版本）

CPSR 中的位 [26:8] 用于将来 ARM 版本的扩展。应用软件最好不要操作这些位，以免与 ARM 将来版本的扩展发生冲突。

3.5.3 Thumb 状态下的寄存器

CPSR 中的 T 标志位决定是执行 Thumb 还是 ARM 指令，如置位则执行 Thumb 指令，否则执行 ARM 指令。通常 Thumb 指令与 ARM 指令的转换是通过执行一条转移和交换指令完成的，如 BX 指令。Thumb 与 ARM 间状态的切换不影响处理器的模式和寄存器的内容。

Thumb 状态下的寄存器集是 ARM 状态下的寄存器集的子集。程序可以直接访问 8 个通用寄存器（R0∼R7）、PC、SP、LR 和 CPSR。每一种特权模式都有一组 SP、LR 和 SPSR，如表 3.7 所示。

在 Thumb 状态下，程序计数器 PC 使用第 2 个二进制位来选择另一半字，写到 R15 中的最低位被忽略。这样指令的目的地值是 R15 中的值与 0xFFFFFFFE 做与操作。

Thumb 状态寄存器与 ARM 状态寄存器的映射关系如表 3.8 所示。

3.5.4 协处理器寄存器

ARM 体系结构支持通过增加硬件协处理器来扩展其指令集的机制。如果协处理器硬件不存在，扩展指令的执行将产生未定义指令异常，通过这种未定义指令陷阱可以进行协处理器的软件仿真。

- ARM 架构支持多达 16 个逻辑协处理器；

表 3.7: Thumb 状态下的寄存器

用户模式	系统模式	管理模式	中止模式	未定义模式	中断模式	快中断模式
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
SP	SP	SP-svc	SP-abt	SP-und	SP-irq	SP-fiq
LR	LR	LR-svc	LR-abt	LR-und	LR-irq	LR-fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR-svc	SPSR-abt	SPSR-und	SPSR-irq	SPSR-fiq

• 每个协处理器可有多达 16 个专用寄存器，其大小可以是任意合理的数，只有在特权模式下才可被访问；它们的状态由控制 ARM 寄存器指令的镜像指令来控制；

• 协处理器使用 Load-Store 体系结构，有对内部寄存器操作的指令，有从存储器读取数据装入寄存器以及将寄存器数据存入存储器的指令，以及与 ARM 寄存器传送数据的指令。

ARM 协处理器及其功能如表 3.9 所示。

系统控制协处理器（CP15）用于控制 MMU、Cache、写缓冲以及其他一些系统属性。ARM 各版本 CP15 中的寄存器功能略有不同，下面以采用 Intel XScale 核的 CPU 为例介绍 CP15 中的寄存器及其功能。Intel XScale CP15 寄存器结构如表 3.10 所示。

• 寄存器 0 包含 2 个只读寄存器：ID 和 Cache 类型寄存器，通过改变 MRC 指令的 opcode-2 域进行访问，当 opcode-2=0 时，则访问 ID 寄存器；

• 寄存器 1 包含 2 个寄存器：控制和辅助控制寄存器，通过改变 MRC 指令的 opcode-2 域进行访问，当 opcode-2=0 时，则访问控制寄存器；

• 寄存器 14 包含 1 个断点寄存器组：2 个指令断点地址寄存器（IBCR0 和 IBCR1）、1 个数据断点地址寄存器（DBR0）、1 个可设置的数据屏蔽/地址寄存器（DBR1）和 1 个数据断点控制寄存器（DBCON）。

3.6 ARM I/O 结构

输入/输出系统简称 I/O 系统，它包括 I/O 设备和 I/O 设备与处理器的连接。在计算机体系结构中，I/O 完成与外部系统的信息交换，是冯 · 诺依曼结构计算机系统四大组成部分之一。I/O 系统所涉及的设备数量、种类、类型都比较繁杂，对计算机整体性能的影响也大不一样，其中存储设备对计算机系统的整体性能影响最大。

3.6.1 AMBA 总线

ARM 架构中的处理器及内核一般都没有 I/O 的部件和模块，构成 ARM 架构的处理器中的 I/O 都通过某种接口与片上其他外围宏单元进行通信。ARM 公司提出了 AMBA（advanced microcontroller bus architecture）总线，使片上不同宏单元的连接实现标准化。

AMBA 规范定义了 3 种总线（见图 3.12）：

表 3.8: Thumb 状态寄存器与 ARM 状态寄存器的映射关系

Thumb 状态	映射关系	ARM 状态
R0	→ → →	R0
R1	→ → →	R1
R2	→ → →	R2
R3	→ → →	R3
R4	→ → →	R4
R5	→ → →	R5
R6	→ → →	R6
R7	→ → →	R7
		R8
		R9
		R10
		R11
		R12
SP	→ → →	SP（R13）
LR	→ → →	LR（R14）
PC	→ → →	PC（R15）
CPSR	→ → →	CPSR
SPSR	→ → →	SPSR

表 3.9: ARM 协处理器

协处理器号	功能
15	系统控制
14	性能监控
13∼8	保留
7∼4	用户
3∼1	保留
0	DSP

• AHB（advanced high-performance bus）：用于连接高性能的系统模块。支持突发数据传送方式和单个数据传送方式，所有时序都以单一时钟沿为准；支持分时处理，提高总线效率；使用中心多路器总线方案，而不是三态驱动的双向总线；支持更宽的 64 位或 128 位数据总线配置。

• ASB（advanced system bus）：用于连接高性能的系统模块。支持突发数据传送方式；支持多主机；使用三态驱动的双向总线；支持 32 位数据传送；有逐步被 AHB 所取代之趋势。

• APB（advanced peripheral bus）：用于连接低性能的外围部件。提供较简单的静态总线接口，减小开销。

3.6.2 存储器和存储器映像 I/O

一、地址空间及存储器格式

ARM 体系结构使用 2^{32} 个 8 位字节的单一线性地址空间。这些字节单元的地址是一个无符号的 32 位数值，范围为：0∼2^{32}-1。

ARM 支持的数据类型有：字节（8 位）、半字（16 位）和字（32 位）。因此，ARM 的地址空间也可以看作是 2^{30} 个 32 位的字单元，这些字单元的地址应可以被 4 整除（字对准），即该地址的

表 3.10: Intel XScale CP15 寄存器结构

寄存器(CRn)	操作码(opcode-2)	访问	功能
0	0	读/写 -忽略	ID 寄存器
0	1	读/写 -忽略	Cache 类型
1	0	读/写	控制
1	1	读/写	辅助控制
2	0	读/写	转换表基地址
3	0	读/写	页域访问控制
4	0	不可预知	保留
5	0	读/写	故障状态
6	0	读/写	故障地址
7	0	读 -不可预知/写	Cache 操作
8	0	读 -不可预知/写	TLB 操作
9	0	读/写	Cache 锁定
10	0	读/写	TLB 锁定
11-12	0	不可预知	保留
13	0	读/写	程序 ID(PID)
14	0	读/写	断点寄存器组
15	0	读/写	协处理器访问控制

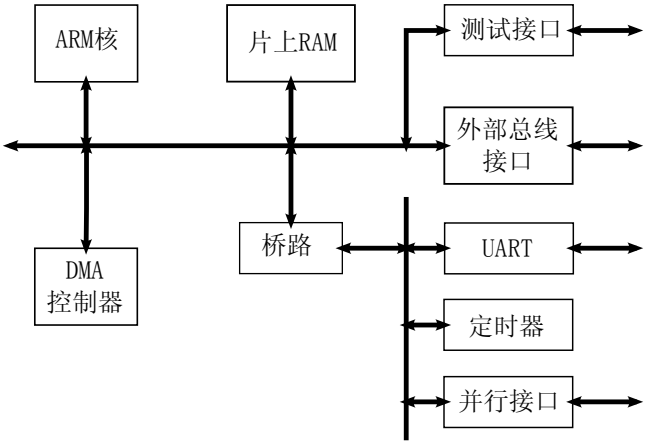

图 3.12: ARM 架构的 AMBA 总线系统

低两位为:0b00 ;还可以将 ARM 的地址空间也可以看作是 2^{31} 个 16 位的半字单元,这些半字单元的地址应可以被 2 整除(半字对准),即该地址最低位为:0b0。

各存储单元的地址作为 32 位的无符号数,可以进行常规的整数运算。这些运算的结果进行 2^{32} 取模,即运算结果上溢出和下溢出时,地址将会发生卷绕。

在 ARM 体系中,每个字包含 4 个字节单元;每个半字包含 2 个字节单元。在字或半字中,哪个字节是高位字节,哪个字节是低位字节,则有两种不同的格式,如图 3.13 所示:

- 大端模式(big-endian):地址为 A 的字单元包括字节单元 A、A+1、A+2、A+3,这些字节单元由高位到低位的字节顺序为 A、A+1、A+2、A+3 ;地址为 A 的字单元包括半字单元 A、A+2,这些半字单元由高位到低位的半字顺序为 A、A+2 ;地址为 A 的半字单元包括字节单元 A、A+1,这些字节单元由高位到低位的字节顺序为 A、A+1。
- 小端模式(little-endian):地址为 A 的字单元包括字节单元 A、A+1、A+2、A+3,这些字节单元由高位到低位的字节顺序为 A+3、A+2、A+1、A ;地址为 A 的字单元包括半字单元 A、A+2,这些半字单元由高位到低位的半字顺序为 A+2、A ;地址为 A 的半字单元包括字节单元

A、A+1，这些字节单元由高位到低位的字节顺序为 A+1、A。

- 两种模式难分优劣，ARM 体系两者都支持，默认为小端模式。

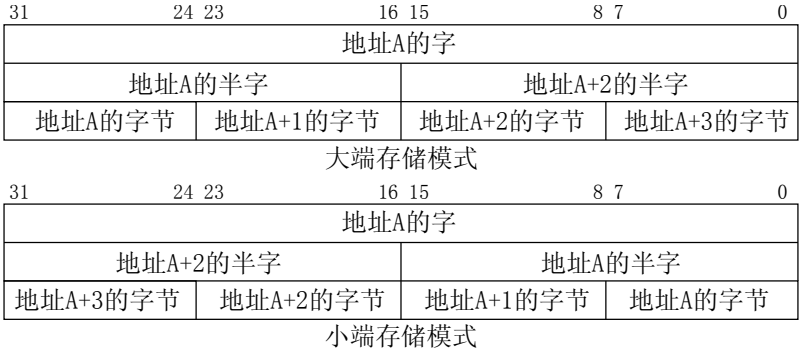

图 3.13: ARM 体系结构的存储模式

二、存储器映像 I/O

ARM 系统完成 I/O 功能的标准方法是使用存储器 I/O 映像，即把 I/O 端口地址作为特殊的存储器地址。这种方法将存储器空间划分成几部分，对其中特定的存储器地址加载或存储时，则提供 I/O 功能。典型情况下，从存储器映像 I/O 地址加载用于输入，而向存储器映像 I/O 地址存储用于输出。

所有 ARM 处理器的 I/O 寄存器都映射在 32 位存储器地址空间中，不同于 x86 体系采用的单独的 I/O 空间，在 ARM 体系中通过寄存器访问 I/O 与访问存储器的机制是一样的，使用相同的指令。不过 I/O 的输入/输出与真正的存储器读/写仍然有所不同，存储器的单元可以重复读多次，其读出的值是一致的；而 I/O 设备的连续两次输入值可能会有所不同。这些差异会影响到存储系统的 Cache 和写缓冲作用，因此，存储器映像 I/O 的地址空间设置成非 Cache 和非缓冲的。

3.6.3　中断和直接存储器存取

为了提高 I/O 处理的能力，ARM 体系处理器都具有中断处理的功能。对于高档次的 ARM 体系处理器，还具有直接存储器存取（direct memory access，DMA）的功能。

中断是指计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后再返回现行程序的断点处继续执行。中断可以提高计算机的实时响应能力和运行效率。

ARM 体系处理器对于一些要求 I/O 处理速率较高的事件安排了快速中断（fast interrupt request，FIQ），而对其余的 I/O 源则安排了一般中断（interrupt request，IRQ）。

- FIQ 中断的最大延迟为 29 个处理周期，有以下及部分组成：

1. Tsynmax：请求信号通过同步器锁住的最长时间为 4 个周期；
2. Tldm：最长指令执行完成时间为 20 个周期；
3. Texc：数据中止异常进入时间为 3 个周期；
4. Tfiq：FIQ 进入时间为 2 个周期。

- FIQ 中断的最小延迟仅包括 Tsynmax 和 Tfiq 为 5 个处理周期。
- IRQ 中断延迟的计算与 FIQ 相类似。若必须容许 FIQ 有更高的优先级，那么进入 IRQ 处理程序的延迟时间是随机的。

在 I/O 数据流量比较大，中断处理比较频繁时，处理器负担加重，明显影响系统的性能。一般采用直接存储器存取来解决这个问题（参见图 3.14）。

主存和 DMA 接口之间有一条数据通路，因此 I/O 设备与主存间数据块的交换就不需要处理器的介入，从而使处理器省去了设备服务及现场保护和恢复；DMA 工作速度比中断方式高，处理器要处理的中断仅仅在出错时或缓冲区满时。

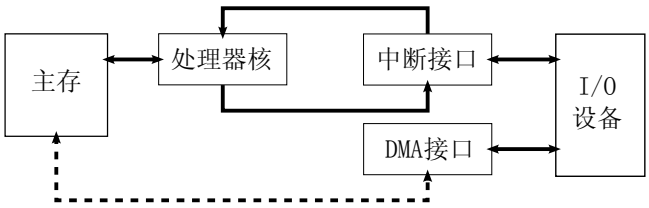

图 3.14: DMA 和中断两种方式的数据通路

当出现 DMA 和 CPU 同时访问主存的冲突时，可采用如下三种办法：

- 停止 CPU 访问主存；
- 周期挪用：DMA 挪用或窃取总线占用权几个主存周期（常采用此办法）；
- DMA 与 CPU 交替访问。

ARM 架构处理器只有在一些高档的 CPU 才有 DMA 的功能，如：基于 StrongARM 的 Intel SA-1110 及基于 XScale 的 Intel PXA250 等。

3.7 ARM 体系结构版本及命名方法

ARM 体系结构在其发展过程中经历了多次修订，将来还会继续发展。迄今为止，ARM 体系结构共定义了 7 个版本，各版本中还有一些变种定义了指令集中不同的功能。ARM 处理器系列中，只要它们支持的 ARM 体系结构版本相同，则基于它们的应用软件将是兼容的。

3.7.1 ARM 体系结构版本

ARM 体系结构的 7 个版本特点如下：

1. 版本 v1
   该版本只由 ARM1 实现，从未用于商用产品。它包括：
   - 基本的数据处理指令（不包括乘法指令）；
   - 基于字节、字和多字的 Load/Store 指令；
   - 包括子程序调用指令 BL 在内的分支指令；
   - 供操作系统使用的软件中断指令。

   该版本只有 26 位的寻址空间，现已不再使用。
2. 版本 v2
   该版本在 v1 的基础上，增加了下列特点：
   - 乘法和乘加法指令；
   - 协处理器支持指令；
   - FIQ 模式中 2 个以上分组的寄存器；
   - SWP 指令和 SWPB 指令。

   该版本只有 26 位的寻址空间，现已不再使用。
3. 版本 v3
   该版本较以前的版本有了较大改进：

- 32 位的寻址空间，除了版本 v3G 外，强制要求与以前 26 位体系版本兼容。版本 v3 和版本 v3G 的区别现已无意义。
- 当前程序状态信息由原来的 R15 寄存器移到一个新的当前程序状态寄存器（CPSR）中，并增加了一个程序状态保存寄存器（SPSR）；
- 增加了两种处理器模式，以便操作系统代码可以有效地使用数据中止、取指中止和未定指令异常；

- 增加了 MRS 指令和 MSR 指令，用于访问 CPSR 寄存器和 SPSR 寄存器；
- 修改了原来从异常返回的指令。

4. 版本 v4

该版本在 v3 的基础上，增加了下列特点：

- 半字的 Load/Store 指令；
- 加载（Load）带符号的字节和半字数据指令；
- 增加了 T 变种，用于支持 Thumb 状态的 16 位指令集；
- 增加了使用用户模式寄存器的处理器特权模式。

该版本明确指明了哪些指令将会引起未定义指令异常。该版本也不再强求与以前 26 位体系版本兼容。

5. 版本 v5

该版本在 v4 的基础上，增加了下列特点：

- 提高了 T 变种中 ARM/Thumb 混合使用的效率；
- 令 T 变种指令和非 T 变种指令一样，使用相同的代码生成技术；
- 增加了一个前导零计数（count leading zeros）指令，该指令可以使整数除法和中断优先级操作更为有效；
- 增加了软件断点指令；
- 为协处理器设计提供了更多可选择指令；
- 更加严格地定义了乘法指令对标志位的设置。

6. 版本 v6

2001 年 ARM 公司发布了 ARM 体系版本 v6，其主要特点是增加了 SIMD（single instruction multiple data）功能。该功能为嵌入式应用系统提供了高性能的音频/视频处理技术，它可以使音频/视频处理性能提高 2∼4 倍。该版本首先在 2002 年发布的 ARM11 处理器中使用。

7. 版本 v7

目前最新的 ARM 体系版本是 v7，这个版本的核心定义了三个不同系列的应用方向，其中 A 系列主要用于复杂的应用处理器；R 系列用于需要实时处理能力的嵌入式环境；M 系列应用于对功耗体积要求更高的深嵌入式环境。

在指令集上，支持新的 Thumb-2 指令，提供更高的效率和性能；支持 NEON 指令（即 advanced SIMD），性能比 SIMD 提升更多；支持 VFPv3，提供更好的浮点运算性能。

3.7.2 ARM 体系的变种

ARM 体系版本定义了一些实现特定功能的变种：

1. Thumb 指令集（T 变种）

Thumb 指令集是 ARM 指令集的重编码子集。Thumb 指令长度为 ARM 指令长度的一半：16 位，这样使用 Thumb 指令集可以提高指令的代码密度。

与 ARM 指令集相比，Thumb 指令集有如下两个限制：

- 完成相同的操作，Thumb 代码通常需要更多的指令，因此，在系统对时间要求苛刻的场合，ARM 指令更为合适；
- Thumb 指令集不包括异常处理所需的指令，因此，在异常中断处理时，需要使用 ARM 指令。

这些限制决定了 Thumb 指令集总是与相应版本的 ARM 指令集配合使用。对支持 Thumb 指令集的 ARM 体系版本，使用字符 T 来表示。

有两种版本的 Thumb 指令集：

- Thumb 指令集版本 v1：该版本用于 ARM 体系结构版本 v4 的 T 变种；
- Thumb 指令集版本 v2：该版本用于 ARM 体系结构版本 v5 的 T 变种。

Thumb 指令集版本 v2 比 v1 具有以下特点：

- 增加/修改了一些指令，用于提高 ARM 指令和 Thumb 指令混合使用时的效率；
- 增加了软件中断指令；
- 更加严格地定义了 Thumb 乘法指令对条件标志位的设置。

这些改变与 ARM 体系版本 v4 和 v5 之间的变化密切相关。因而实际中并不使用 Thumb 版本号，而是使用相应的 ARM 版本号。

2. 长乘法指令（M 变种）

ARM 指令集的 M 变种增加了两种用于长乘法操作的指令。一种用于完成 32 位整数乘 32 位整数，生成 64 位整数的长乘法操作；另一种用于完成 32 位整数乘 32 位整数，然后加上 32 位整数，生成 64 位整数的长乘加法操作。

长乘法操作意味着需要较大的乘法器，因此，在芯片尺寸要求苛刻、乘法性能不太重要的场合，没有必要增加这种变种的功能。

对支持长乘法指令的 ARM 体系版本，使用字符 M 来表示。

3. 增强型 DSP 指令（E 变种）

ARM 指令集的 E 变种增加了一些附加指令，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。主要包括：

- 几条新的 16 位乘法和乘加指令；
- 实现饱和的带符号数的加法和减法指令。这种算法在计算产生溢出时，结果使用最大正数或最小负数而不进行环绕；
- 双字数据操作指令。包括加载指令 LDRD、存储指令 STRD 和协处理器的寄存器传送指令 MCRR/MRRC ；
- Cache 预取指令 PLD。

E 变种首先在 ARM 体系版本 v5T 中使用，用字符 E 来表示。在 ARM 体系版本 v5 以前的版本中，以及在非 M 变种和非 T 变种的版本中，E 变种是无效的。

在以前的 E 变种中，为包含双字加载指令 LDRD、存储指令 STRD 和协处理器的寄存器传送指令 MCRR/MRRC，以及 Cache 预取指令 PLD。这种 E 变种记作 ExP，其中 x 表示缺少，P 表示上述几种指令。

4. Java 加速器 Jazelle（J 变种）

ARM 的 Jazelle 技术提供了 Java 加速功能。Jazelle 技术使 Java 代码的运行速度比普通 Java 虚拟机提高了 8 倍，而功耗却降低了 80%。

J 变种首先在 ARM 体系版本 v4TEJ 中使用，使用字符 J 来表示。

5. ARM 媒体功能扩展（SIMD 变种）

ARM 的 SIMD 媒体功能扩展为包括音频/视频处理在内的应用系统提供了优化功能，它可以使音频/视频处理性能提高 4 倍。主要特点如下：

- 音频/视频处理性能提高 2∼4 倍；
- 可以同时进行两个 16 位操作数或四个 8 位操作数的运算；
- 提供了小数算术运算；
- 用户可定义饱和运算的模式；
- 两套 16 位操作数的乘加/乘减运算；
- 32 位乘以 32 位的小数乘法运算；
- 同时 8/16 位选择操作。

目前 SIMD 变种只在 ARM 体系版本 v6 中使用。

3.7.3 ARM/Thumb 体系结构版本的命名格式

完整的 ARM/Thumb 体系结构版本的命名格式，由下面几部分组成的字符串来表示：

- 字符串 ARMv
- ARM 指令集版本号，目前为 1∼7 ；

• 表示变种的字符(除 M 外)。在 ARM 体系结构版本 v4 及以上版本中,M 变种为标准配置,因而通常不被列出;

• 使用字符 x 表示排除某种功能,如 ExP 表示不包含 LDRD、STRD、MCRR、MRRC 及 PLD 指令。

表 3.11 列出了目前有效的 ARM/Thumb 体系结构版本的名称及含义。这些名称提供了描述 ARM 体系结构版本特点的简便表示方法。

表 3.11: ARM/Thumb 体系结构版本名称及含义

名称	ARM 指令集	Thumb 指令集	长乘法指令	DSP 指令
ARMv3	3	N	N	N
ARMv3M	3	N	Y	N
ARMv4Xm	4	N	N	N
ARMv4	4	N	Y	N
ARMv4TxM	4	1	N	N
ARMv4T	4	1	Y	N
ARMv5xM	5	N	N	N
ARMv5	5	N	Y	N
ARMv5TxM	5	2	N	N
ARMv5T	5	2	Y	N
ARMv5TExP	5	2	Y	Y[a]

名称	T 变种	E 变种	J 变种	SIMD 变种
ARMv5TEJ	Y	Y	Y	N
ARMv6	Y	Y	Y	Y
ARMv5TE	5	2	Y	Y

[a]除 LDRD、STRD、MCRR、MRRC 及 PLD 指令

3.8 ARM 处理器核

ARM 处理器核是在 ARM 核的基础上,增加了 Cache、存储器管理单元 MMU、协处理器 CP15、AMBA 接口及其他宏单元所构成。目前,常被采用的 ARM 核有 ARM7TDMI、ARM9TDMI、ARM10TDMI、StrongARM 及 XScale 等。在这些 ARM 核的基础上,构成了各系列的 ARM 处理器核。

3.8.1 ARM7 系列

一、ARM7 系列中 ARM 核

该系列被广泛采用的 ARM 核为 ARM7TDMI,它支持 Thumb 压缩指令集,支持片上调试(debug),支持增强型乘法器(multiplier),支持片上断点。

具体特点如下:

• 采用 ARM 体系结构 v4T 版指令,支持 16 位 Thumb 指令集;

• 支持 32 位整数运算,32×32 位乘法运算(结果为 64 位);

• 采用 3 级流水线组织;

• 嵌入式 ICE 模块提供了片内调试功能,提供 JTAG 调试开发接口;

• 提供 32 位存储器接口、MMU 接口、协处理器接口、debug 接口,以及时钟与总线等控制信号。

标准的 ARM7TDMI 核是“硬”的宏单元，以物理版图提供，定制为适当的工艺技术。它还有一个可综合的“软”版本：ARM7TDMI-S，以高级语言模块的形式提供，可以使用任何目标工艺的适当的单元库来综合，综合过程中可以对 ARM 核的功能进行选择。

二、ARM7 系列中 ARM 处理器核

基于 ARM7TDMI 核的 ARM 处理器核有 ARM710T、ARM720T 和 ARM740T。它们在 ARM7TDMI 核的基础上增加了 8KB 的指令和数据混合 Cache；AMBA 总线连接外部存储器和外围器件；集成了写缓冲器及 MMU（ARM710T/ARM720T）或存储器保护单元（ARM740T）。

具体的组织结构特点如下：

- 8KB 的指令和数据 Cache 采用 4 路组相联结构，随机替换算法，写直达策略；
- 在 ARM710T/ARM720T 中，MMU 实现了存储器管理功能，使用了系统控制协处理器 CP15，TLB 采用 64 数据项的相联 Cache，最近使用算法；
- 在 ARM740T 中，使用存储器保护单元代替 MMU，它不支持虚拟地址到物理地址的转换，但以较小的代价提供了基本的保护及 Cache 控制功能，这适合于运行固定软件系统的嵌入式应用。

3.8.2 ARM9 系列

一、ARM9 系列中 ARM 核

该系列被广泛采用的 ARM 核为 ARM9TDMI，它将 ARM7TDMI 的功能明显地提高到更高的性能水平。它支持 Thumb 压缩指令集，支持片上调试（debug）。

具体有如下特点：

- 采用 ARM 体系结构 v4T 版指令，支持 16 位 Thumb 指令集；
- 支持 32 位整数运算，32×32 位乘法运算（结果为 64 位）；
- 采用 5 级流水线组织；
- 嵌入式 ICE 模块提供了片内调试功能，提供 JTAG 调试开发接口，支持硬件单步调试，及异常时设置断点；
- 提供 32 位存储器接口、MMU 接口、协处理器接口、debug 接口，以及时钟与总线等控制信号；
- 使用分开的指令与数据存储器接口，改善了每条指令的时钟数（clock per instruction，CPI）；
- 协处理器接口支持片上浮点协处理器、数字信号处理或其他专用的硬件加速要求；
- 低电压操作以降低功耗，2.5V 工作，甚至 1.2V。

ARM9E-S 是 ARM9TDMI 核可综合的“软”版本，它实现的是扩展的 ARM 指令集。除了 ARM9TDMI 支持的 ARM 体系结构 v4T 版本的指令外，ARM9E-S 还支持完整的 ARM 体系结构 v5TE 版本的指令，如信号处理指令集的扩展。

二、ARM9 系列中 ARM 处理器核

基于 ARM9TDMI 核的 ARM 处理器核有 ARM920T 和 ARM940T。它们在 ARM9TDMI 核的基础上增加了分开的指令和数据 Cache；指令和数据端口通过 AMBA 总线主控单元合并在一起；集成了写缓冲器及 MMU（ARM920T）或存储器保护单元（ARM940T）。见图 3.15。

具体的组织结构特点如下：

- 在 ARM920T 中，采用分开的指令和数据 Cache，大小都是 16KB。采用 64 路组相联的分段式 CAM-RAM 组织、随机或循环替换算法、写直达或写回策略。写缓冲器可以保存 4 个地址和 16 个数据字。
- 在 ARM940T 中，采用分开的指令和数据 Cache，大小都是 4KB，由 4 个 1KB 的段构成。采用全相联的 CAM-RAM 组织，随机或循环替换算法，写直达或写回策略。写缓冲器可以保存 4 个地址和 8 个数据字。

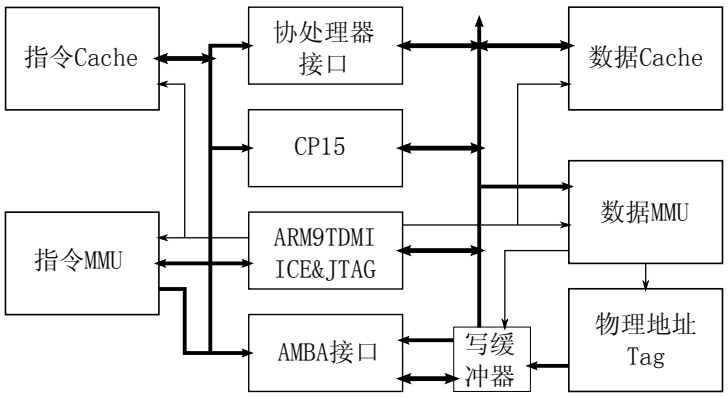

图 3.15: ARM920T 处理器核的组织结构

• 在 ARM920T 中，MMU 实现了存储器管理功能，具有分开的指令和数据 MMU。使用了系统控制协处理器 CP15，分开的指令和数据 TLB 采用 64 数据项的相联 Cache，支持 64KB 的大页、4KB 的小页及 1KB 的微页转换。

• 在 ARM940T 中，使用分开的指令和数据存储器保护单元代替 MMU，它不支持虚拟地址到物理地址的转换，是 ARM920T 处理器核的简化版本，但其性能仍相当高。

基于 ARM9E-S 核的 ARM 处理器核有 ARM946E-S 和 ARM966E-S。它们在 ARM9E-S 核的基础上增加了分开的指令和数据 Cache（ARM946E-S）或嵌入紧耦合 SRAM（ARM966E-S）、AMBA 接口及嵌入时跟踪宏单元。它们没有地址转换硬件，主要为嵌入式应用提供软 IP 核。

3.8.3　ARM10 系列

一、ARM10 系列中 ARM 核

该系列被广泛采用的 ARM 核为 ARM10TDMI。对于同样的工艺、同样的芯片面积，ARM9TDMI 的性能为 ARM7TDMI 的 2 倍，而 ARM10TDMI 的性能则为 ARM9TDMI 的 2 倍。ARM10TDMI 在系统结构上主要采用增加时钟速率和减少每条指令平均时钟数（CPI）两大措施。

1. 增加时钟速率：

ARM 核所支持的最高时钟速率是由任意流水线级的、最慢的逻辑路径所决定的。ARM10TDMI 保留了与 ARM9TDMI 相类似流水线结构，采用特别的方式优化每一级，从而支持更高的时钟速率。

• 在取指和存储器访问阶段，提前提供下一周期所需的地址；
• 在执行阶段，采用改进的电路技术和结构；
• 指令译码阶段无法支持高时钟速率，流水线中增加“发射（issue）”一级，即采用 6 级流水线结构。

2. 减少 CPI

想要有效地减少 CPI，首先必须考虑存储器的带宽。ARM7TDMI 采用单一 32 位的存储器，ARM9TDMI 采用 32 位分开的指令和数据存储器，ARM10TDMI 则采用 64 位存储器。这有效地消除了指令带宽瓶颈，使得能够在处理器组织中加入若干改善 CPI 的特性：

• 采用逻辑功能更强的转移预测；
• 采用非阻塞的存取执行，使流水线的执行阶段不产生停顿；
• 64 位的数据存储器使得在每个时钟周期传送 2 个寄存器的存取指令。

二、ARM10 系列中 ARM 处理器核

基于 ARM10TDMI 核的 ARM 处理器核为 ARM1020E，采用 ARM 体系结构 v5TE 版本指令。它与 ARM920T 的组织非常相似，不同之处在于 Cache 的大小和总线的宽度。

具体的组织结构特点如下：

• 采用分开的指令和数据 Cache，大小都是 32KB。采用 64 路组相联的分段式 CAM-RAM 组织，随机或循环替换算法，写直达或写回策略。写缓冲器有 8 块，每块可以保存 1 个地址和 64 位数据，还有一个用于处理 Cache 更新的 4 块大小的写缓冲器。

• 采用分开的指令和数据 AMBA、AHB 总线接口。两个 Cache 的数据总线带宽都是 64 位。这使得指令 Cache 每周期可提供 2 条指令；数据 Cache 在多寄存器存取时，每周期可提供 2 个字。

• MMU 基于 2 个 64 表项的 TLB，每个 Cache 有 1 个。TLB 还支持可选择的锁定，以保证关键的实时代码部分不会被替换。

3.8.4 Intel XScale

Intel XScale 架构处理器是由 Intel 公司发布，用于新一代无线手持式应用产品开发的嵌入式处理器，是 Intel 个人互联网用户端架构（PCA）开发式平台中的应用子系统和通信子系统中的嵌入式处理器。其 Intel XScale 处理器核具有以下特点：

• 与 ARM 体系结构 v5TE 版本指令集兼容；

• 采用 7/8 级流水线。包括动态跳转预测和分支目标缓冲器；

• 指令 Cache：32KB，32 路组相联，可锁定；

• 数据 Cache：32KB，32 路组相联，写直达或写回；

• 还可以将该数据 Cache 中的最大 28KB 定义成数据 RAM 使用；

• 微小数据 Cache：2KB，2 路组相联；

• 指令存储器管理单元 IMMU：32 块 TLB，全相联，可锁定；

• 数据存储器管理单元 DMMU：32 块 TLB，全相联，可锁定；

• 4~8 块填充缓冲器用于提高处理器从存储器读取数据时的性能；

• 8 块写缓冲器用于提高处理器向存储器存储数据时的性能；

• 增加了协处理器 CP0 用于 DSP 处理，包括乘法/累加 MAC，提供 40 位累加器和 8 条新的运算指令；支持语音数据处理的 16 位 SIMD ；

• 增加了协处理器 CP14 用于性能监控、时钟和电源管理以及软件调试；

• 协处理器 CP15 中增加了功能；

• debug 中支持硬件断点、硬件观察点、BKPT 指令、异常中断和 JTAG 接口电路等；

• 高性能，低功耗。处理器时钟可达 1GHz，功耗 1.6W，性能达 1200MIPS。

3.9 ARM 指令系统

ARM 指令系统属于 RISC 指令系统。ARM 指令集可以分为数据处理指令、Load/Store 存储器访问指令、转移指令、协处理器指令、信号处理指令和异常中断指令等。

3.9.1 ARM 指令概述

标准的 ARM 指令字长为固定的 32 位。有些 ARM 核还可以执行 16 位的 Thumb 指令集，它是 ARM 指令集的子集。

ARM 处理器一般支持下面 6 种数据类型：

• 8 位有/无符号字节类型数据；

• 16 位有/无符号半字类型数据；

- 32 位有/无符号字类型数据。

一条典型的数据处理指令语法格式如下：

```
< opcode> {<cond>}{S}<Rd>,<Rn>,<Operand2>
```

其中：<opcode> 为指令助记符，如：ADD 表示算术加操作指令；{<cond>} 表示指令执行的条件；{S} 表示指令的操作是否影响 CPSR 的值；<Rd> 表示目标寄存器；<Rn> 表示包含第 1 个操作数的寄存器；<Operand2> 表示第 2 个操作数。

上述格式中，<Operand2> 表示第 2 个操作数。它的组成非常灵活，可以是立即数，也可以是逻辑运算数。这使得 ARM 指令可以在读取数值的同时进行算术和移位操作，即 ARM 的一条指令具有多功能的特点。

{<cond >} 表示指令执行的条件。在 ARM v5 以前的版本，所有的指令都是条件执行的。从 ARM v5 版本开始，引入了一些无条件执行的指令。所谓条件执行，即是根据 CPSR 中的条件标志位来决定是否执行该指令。当条件满足时执行该指令，条件不满足时该指令被当作一条 NOP 指令。

{<cond>} 条件码占 4 位，共有 16 个，其含义和助记符如表 3.12 所示。

表 3.12: ARM 指令的条件码

条件码	助记符	含义	CPSR 中条件标志位值
0000	EQ	相等/等于 0	Z=1
0001	NE	不相等	Z=0
0010	CS/HS	进位/无符号数大于或等于	C=1
0011	CC/LO	无进位/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	未上溢出	V=0
1000	HI	无符号数大于	C=1 且 Z=0
1001	LS	无符号数小于或等于	C=0 或 Z=1
1010	GE	有符号数大于或等于	N=V
1011	LT	有符号数小于	N≠V
1100	GT	有符号数大于	Z=0 且 N=V
1101	LE	有符号数小于或等于	Z=1 或 N≠V
1110	AL	总是(always)执行	任何状态
1111	NV 未定义 AL	从不(never)执行 指令执行结果不可预知 总是(always)执行	ARM v3 之前 ARM v3 及 ARM v4 ARM v5 及以上版本

可条件执行的指令可以在其助记符的扩展域加上条件码助记符，从而使得该指令在特定的条件下执行，例：

```
ADD     R0, R1, R2          ;R0=R1+R2
ADDHI   R0, R1, R2          ;当 CPSR 中的 C=1 且 Z=0 时,R0=R1+R2
```

3.9.2 ARM 数据处理指令

这里将 ARM 数据处理指令分为：基本数据操作指令、乘法指令和杂类数据处理指令。

一、基本数据操作指令

ARM 基本数据操作指令语法格式为：

```
<opcode>{<cond>}{S} <Rd>,<Rn>,<Operand2>
```

它们使用 3 地址格式，即 2 个源操作数和 1 个目标寄存器 <Rd>。其中第 1 个操作数总是寄存器 <Rn>，第 2 个操作数 <Operand2> 可能是立即数、寄存器或移位后的寄存器。

1. 第 2 操作数 <Operand2> 的寻址方式。

- 立即数寻址

<Operand2> 由一个 8 位的常数 <immed_8> 循环右移偶数位得到。其中循环右移的位数由一个 4 位的二进制数 <rotate_imm_4> 的 2 倍（0、2、4、6 … 30）表示，则有：

<Operand2>=<immed_8> 循环右移（2×<rotate_imm_4>）

并不是每一个 32 位的常数都是合法的立即数，只有能通过上述方法得到的才是合法的立即数。如：0xff、0x104 等等，而 0x101、0x102、0xff1 则是不合法的，例如：

```
ADD         R0, R1, #1          ;R0=R1+1
```

- 寄存器直接寻址

<Operand2> 即为寄存器中的数值，例如：

```
ADD         R0, R1, R2          ;R0=R1+R2
```

- 寄存器移位寻址

<Operand2> 由第 2 个操作数寄存器 <Rm> 进行移位操作得到。其中移位数由 <Shift_imm> 决定。则有：

<Operand2>=<Rm> 移位操作 <Shift_imm>

具体的移位操作方式有以下几种：

ASR　算术右移

LSL　逻辑左移

LSR　逻辑右移

ROR　循环右移

RRX　扩展的循环右移（带进位标志循环右移 1 位）

例如：

```
ADD         R0, R1, R2, LSL #3          ;R0=R1+R2×2^3
```

- 寄存器间接移位寻址

<Operand2> 由第 2 个操作数寄存器 <Rm> 进行移位操作得到。其中移位数由移位位数寄存器 <Rs> 决定。则有：

<Operand2>=<Rm> 移位操作 <Rs>

具体的移位操作方式有以下几种：

ASR　算术右移

LSL　逻辑左移

LSR　逻辑右移

ROR　循环右移

例如：

```
ADD         R0, R1, R2, LSR R4          ;R0=R1+R2/2^R4
```

2. 基本数据操作指令。

基本数据操作指令又可分为数据传送指令、算术逻辑运算指令和比较指令。共有 16 条指令，如表 3.13 所示。

表 3.13: 基本数据操作指令

指令含义	指令语法	指令操作
传送	MOV{<cond>}{S}<Rd>,<Op2>	Rd =<Op2>
传送非	MVN{<cond>}{S}<Rd>,<Op2>	Rd =NOT<Op2>
加法	ADD{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2>+ Rn
带位加法	ADC{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2>+ Rn+C
减法	SUB{<cond>}{S}<Rd>,<Rn><Op2>	Rd = Rn -<Op2>
带位减法	SBC{<cond>}{S}<Rd>,<Rn><Op2>	Rd = Rn -<Op2>- NOT C
反减	RSB{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2>- Rn
带位反减	RSC{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2>- Rn- NOT C
逻辑与	AND{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2> AND Rn
逻辑或	ORR{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2> OR Rn
逻辑异或	EOR{<cond>}{S}<Rd>,<Rn><Op2>	Rd =<Op2> EOR Rn
位清零	BIC{<cond>}{S}<Rd>,<Rn><Op2>	Rd =NOT<Op2> AND Rn
比较	CMP{<cond>}{S}<Rn>,<Op2>	标志位: <Rn>-<Op2>
比较反值	CMN{<cond>}{S}<Rn>,<Op2>	标志位: <Rn>+<Op2>
位测试	TST{<cond>}{S}<Rn>,<Op2>	标志位: <Rn> AND<Op2>
相等测试	TEQ{<cond>}{S}<Rn>,<Op2>	标志位: <Rn> EOR <Op2>

• 数据传送指令用于向寄存器传入一个常数。该指令包括一个目标寄存器 <Rd> 和一个源操作数，源操作数的构造方法见上述的 < Operand2 > 寻址方式，同时更新 CPSR 中相应的条件标志位；

• 算术逻辑运算指令包括一个目标寄存器 <Rd> 和两个源操作数，指令的运算结果存入目标寄存器，同时更新 CPSR 中相应的条件标志位；

• 比较指令更新 CPSR 中相应的条件标志位，但不保存运算结果。

例 1：R0 中的内容乘 7：

```
RSB         R0, R0, R0, LSL #3          ;R0=R0×8-R0
```

例 2：将 R2、R3 中的 64 位整数与 R4、R5 中的 64 位整数相减，结果存入 R2、R3 中：

```
SUBS  R2, R2, R4          ;减低有效字,设置标志
SBC   R3, R3, R5          ;带位减高有效字
```

例 3：R0 带进位标志循环右移 1 位、取反，再 “与”R2，结果存入 R1：

```
BICNES  R1, R2, R0, RRX          ;条件执行,设置标志
```

例 4：R0 逻辑左移 2 次后与 R1 比较，根据结果更新 CPSR 中的 N、Z、C 和 V：

```
CMPGT  R1, R0, LSL #2                    ;条件执行
```

二、乘法指令

ARM 有两类乘法指令：一类为 32 位的乘法指令，即乘积结果为 32 位；另一类为 64 位的乘法指令，即乘积结果为 64 位。共有 6 条指令，如表 3.14 所示。

表中 C 表示运算 (Rm*Rs)[31:0]+ RdLo 所产生的进位。

其中：

<Rd> 为目标（32 位乘积结果/加法结果）寄存器；

<RdLo> 为存放 64 位乘积结果/加法结果低 32 位寄存器；

表 3.14: 乘法指令

指令含义	指令描述
32 位乘法	MUL{<cond>}{S}<Rd>,<Rm>,<Rs> Rd=(Rm*Rs)[31:0]
32 位乘法累加	MLA{<cond>}{S}<Rd>,<Rm>,<Rs><Rn> Rd=(Rm*Rs+Rn)[31:0]
64 位有符号乘法	SMULL{<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs> RdHi=(Rm*Rs)[63:32], RdLo=(Rm*Rs)[31:0]
64 位有符号乘法累加	SMLAL {<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs> RdLo=(Rm*Rs)[31:0]+ RdLo, RdHi=(Rm*Rs)[63:32]+ RdHi+C
64 位无符号乘法	UMULL {<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs> RdHi=(Rm*Rs)[63:32], RdLo=(Rm*Rs)[31:0]
64 位无符号乘法累加	UMLAL {<cond>}{S} <RdLo>,<RdHi>,<Rm>,<Rs> RdLo=(Rm*Rs)[31:0]+ RdLo, RdHi=(Rm*Rs)[63:32]+ RdHi+C

<RdHi> 为存放 64 位乘积结果/加法结果高 32 位寄存器；

<Rm> 为第一个乘数所在的寄存器；

<Rs> 为第二个乘数所在的寄存器；

<Rn> 为第三个操作数的寄存器，该操作数是一个加数。

例如：

```
MLA    R0, R1, R2, R3        ;R0=R1*R2+R3
UMLAL  R1, R2, R3, R4        ;R2 R1=R3*R4+R2 R1
```

三、杂类数据处理指令

1. CLZ 前导零计数指令

指令语法格式：

```
CLZ {<cond>} <Rd>,<Rm>
```

其中：

<Rd> 为目标寄存器；

<Rm> 为操作数寄存器。

指令操作：该指令计算 <Rm> 中值的最高端 0 的个数，结果存入 <Rd> 中。若 <Rm> 为 0，则结果为 32 ；<Rm> 中位 [31] 为 1，则结果为 0。该指令不影响 CPSR 中条件码标志，且只适用于 ARM v5 及以上版本。

例：规范化 <Rm> 中的值，使其最高位为 1：

```
CLZ   Rd, Rm
MOVS  Rm, Rm, LSL Rd
```

2. 状态寄存器与通用寄存器的数据传送指令

当需要保存或修改当前模式下的 CPSR 或 SPSR 的内容时，应先将这些内容传送到一般寄存器中，对选择的位进行修改，然后再将数据回写到状态寄存器中。即：读取—修改—写回。ARM 中有两条用于在状态寄存器与通用寄存器间的数据传送指令。

- MRS 将状态寄存器的内容传送到通用寄存器中的指令语法格式：

```
MRS {<cond>} <Rd>,CPSR
MRS {<cond>} <Rd>,SPSR
```

- MSR 将通用寄存器的内容或一个立即数传送到状态寄存器中的指令语法格式：

```
MSR {<cond>}  CPSR_<fields>, #<immediate>
MSR {<cond>}  CPSR_<fields>, <Rm>
MSR {<cond>}  SPSR_<fields>, #<immediate>
MSR {<cond>}  SPSR_<fields>, <Rm>
```

其中：

<immediate> 为要传送的立即数，其构成方法见 3.9.2 小节中的说明；

<fields> 为状态寄存器中需要操作的位域。状态寄存器中位 [7:0] 是控制位域，用 c 表示；位 [15:8] 是扩展位域，用 x 表示；位 [23:16] 是状态位域，用 s 表示；位 [31:24] 是条件标志位域，用 f 表示。

例：将处理器当前模式切换到管理模式：

```
MRS  R0, CPSR          ;读取 CPSR
BIC  R0, R0, #0x1F     ;修改,去除当前模式
ORR  R0, R0, #0x13     ;修改,设置管理模式
MSR  CPSR_c, R0        ;写回,仅修改 CPSR 中的控制位域
```

3.9.3 ARM Load/Store 存储器访问指令

这里将 ARM 存储器访问指令分为：基本 Load/Store 指令、批量 Load/Store 指令和存储器与寄存器数据交换指令。

一、基本 Load/Store 指令

Load 指令从存储器中读取数据存入寄存器中，Store 指令则将寄存器中的数据保存在存储器中。基本 Load/Store 指令可分为两类：一类为操作 32 位字数据和 8 位无符号字节数据；另一类为操作 16 位半字数据和 8 位有符号字节数据以及双字数据。其指令的寻址方式为带偏移量的变址方式，表示为：基址 + 变址寻址。

1. 字及无符号字节 Load/Store 指令的寻址方式

字及无符号字节 Load/Store 指令的语法格式一般为：

```
<opcode>{<cond>} <Rd>,<address_mode>
```

其中：

<Rd> 为目标寄存器；

<address_mode> 为操作数的存储器地址，其寻址方式如表 3.15 所示。

表 3.15: 字及无符号字节 Load/Store 指令的寻址方式

寻址方式	寻址操作
立即数偏移	[<Rn>, #+/-<offset_12>]
寄存器偏移	[<Rn>, +/-<Rm>]
移位寄存器偏移	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]
前变址立即数偏移	[<Rn>, #+/-<offset_12>]!
前变址寄存器偏移	[<Rn>, +/-<Rm>]!
前变址移位寄存器偏移	[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!
后变址立即数偏移	[<Rn>], #+/-<offset_12>
后变址寄存器偏移	[<Rn>],+/-<Rm>
后变址移位寄存器偏移	[<Rn>], +/-<Rm>, <shift> #<shift_imm>

<Rn>：为基址寄存器；

<Rm>：为索引（偏移）寄存器；
<offset_12>：为 12 位的立即数；
<shift>：表示移位操作，<shift_imm>：为移位操作的位数。移位操作包括：
ASR 算术右移
LSL 逻辑左移
LSR 逻辑右移
ROR 循环右移
RRX 扩展的循环右移（带进位标志循环右移 1 位）

前变址（pre-indexed）：表示用生成的操作数地址访问存储器，并将该操作数地址写入基址寄存器；

后变址（post-indexed）：表示先用基址寄存器中的地址访问存储器，然后再生成操作数地址，并将该操作数地址写入基址寄存器。

2. 半字、有符号字节及双字 Load/Store 指令的寻址方式

半字、有符号字节及双字 Load/Store 指令的语法格式一般为：

```
<opcode>{<cond>}<Rd>, <address_mode>
```

其中：
<Rd> 为目标寄存器；
<address_mode> 为操作数的存储器地址，其寻址方式如表 3.16 所示。

表 3.16: 半字、有符号字节及双字 Load/Store 指令的寻址方式

寻址方式	寻址操作
立即数偏移	[<Rn>, #+/-<offset_8>]
寄存器偏移	[<Rn>, +/-<Rm>]
前变址立即数偏移	[<Rn>, #+/-<offset_8>]!
前变址寄存器偏移	[<Rn>, +/-<Rm>]!
后变址立即数偏移	[<Rn>], #+/-<offset_8>
后变址寄存器偏移	[<Rn>], +/-<Rm>

表 3.16 中各参数的用法与表 3.15 基本相同，这里的 <offset_8> 为 8 位的立即数。

3. 基本 Load/Store 指令

基本 Load/Store 指令有 14 条，如表 3.17 所示。

其中：

用户模式的读取和写入：表示在特权模式下使用该命令时，存储系统将按用户模式下的权限进行存储器访问操作，常用在异常中断程序中；

双字（64 位）的读取和写入：寄存器一个为 <Rd>，另一个为 <R(d+1)>，<Rd> 必须是偶数寄存器，且不能为 <R14>；该指令适用于 ARM v5TE 及以上版本。

例：

```
LDR   R0, [R1, #-4]            ;R0=[R1-4];
LDR   R0, [R1, R2, LSL#2]!     ;R0=[R1+R2*4],且 R1= R1+R2*4;
LDRH  R0, [R1], #2             ;R0=[R1] 低 16 位,然后 R1= R1+2;
STRB  R3, [R5, #0x200]         ;[R5+0x200]=R3 低 8 位,且 R5=R5+0x200;
STRH  R0, [R1], #8             ;[R1]=R0 低 16 位,然后 R1= R1+8;
STRD  R4, [R9,#24]             ;[R9+24]=R4,[R9+28]=R5
```

二、批量 Load/Store 指令

ARM 批量 Load/Store 指令可以一次将指令列表中的各个寄存器从存储器中加载或存储到存储器中。一种指令形式还可以加载或存储用户模式的寄存器，用于保存或恢复用户处理状态。另一种形式可以从 SPSR 中恢复 CPSR 作为从异常处理返回的一部分。这些指令常用于块数据操作、数据栈操作以及子程序操作。

表 3.17: 基本 Load/Store 指令

指令含义	指令语法
字数据读取	LDR{< cond >} <Rd>, <address_mode>
字节数据读取	LDR{< cond >}B <Rd>, <address_mode>
用户模式的字节数据读取	LDR{<cond>}BT <Rd>, <address_mode>
双字数据读取	LDR{<cond>}D <Rd>, <address_mode>
半字数据读取	LDR{<cond>}H <Rd>, <address_mode>
有符号的字节数据读取	LDR{<cond>}SB <Rd>, <address_mode>
有符号的半字数据读取	LDR{<cond>}SH <Rd>, <address_mode>
用户模式的字数据读取	LDR{<cond>}T <Rd>, <address_mode>
字数据写入	STR{<cond>} <Rd>, <address_mode>
字节数据写入	STR{<cond>}B <Rd>, <address_mode>
用户模式的字节数据写入	STR{<cond>}BT <Rd>, <address_mode>
双字数据写入	STR{<cond>}D <Rd>, <address_mode>
半字数据写入	STR{<cond>}H <Rd>, <address_mode>
用户模式的字数据写入	STR{<cond>}T <Rd>, <address_mode>

1. 批量 Load/Store 指令的寻址方式

批量 Load/Store 指令的语法格式一般为：

```
<opcode>{<cond>} <address_mode><Rn>{!}, <registers>{^}
```

其中：

<Rn>：为基址寄存器；

{!}：表示前变址（pre-indexed）；

<registers>：为寄存器列表；

{^}：表示用户模式的操作或带状态寄存器的操作；

<address_mode>：为寻址方式，如表 3.18 所示。

批量 Load/Store 指令中的寄存器列表与存储器单元的对应关系满足这样的规则：编号小的寄存器对应存储器中低的地址单元；编号大的寄存器对应存储器中高的地址单元。

对于块寻址方式，Load 和 Store 指令可以采用相同的寻址方式，数据向存储器中存入和从存储器中读出的方式相同。

对于堆栈寻址方式，由于堆栈操作是一种后进先出的存储形式，数据写入存储器和从存储器中读出的顺序不同。当栈指针指向最后入栈的有效数据（栈顶）时，称为满栈；当栈指针指向被入栈的下一个数据空位时，称为空栈。当数据入栈时存储器地址向上增长称为递增堆栈；当数据入栈时存储器地址向下增长称为递减堆栈。于是，堆栈寻址有 4 种方式：

- 满递减堆栈：<Rn> 指向栈内有效数据的最低地址，堆栈时存储器地址向下增长；
- 空递减堆栈：<Rn> 指向堆栈下方的第一个空位，堆栈时存储器地址向下增长；
- 满递增堆栈：<Rn> 指向栈内有效数据的最高地址，堆栈时存储器地址向上增长；
- 空递增堆栈：<Rn> 指向堆栈上方的第一个空位，堆栈时存储器地址向上增长。

2. 批量 Load/Store 指令

批量 Load/Store 指令可以分为 5 条，如表 3.19 所示。

其中：

{!}：可选后缀。表示指令执行后，将操作数的存储器地址写入基址寄存器 <Rn> 中；

{^}：可选后缀。不可以在用户模式和系统模式下使用，分两种情况：第一：LDM 指令寄存器列表中包含 PC（R15），则除了正常的批量寄存器传送外，还将 SPSR 复制到 CPSR，可用于异常处理返回；第二：LDM 指令寄存器列表中不包含 PC（R15）或 STM 指令，则寄存器列表为用户模式下的寄存器，并按用户模式下的权限进行存储器的访问操作。

表 3.18: 批量 Load/Store 指令的寻址方式

方式	助记符	寻址操作
块寻址	IA	每次传送后地址增 1，编号最小寄存器 =Rn，编号最大寄存器 =Rn+4*(寄存器数 -1)
	IB	每次传送前地址增 1，编号最小寄存器 =Rn+4，编号最大寄存器 =Rn+4*(寄存器数)
	DA	每次传送后地址减 1，编号最小寄存器 = Rn-4*(寄存器数 -1)，编号最大寄存器 =Rn
	DB	每次传送前地址减 1，编号最小寄存器 = Rn-4*(寄存器数)，编号最大寄存器 =Rn-4
堆栈寻址	FD	满递减堆栈，Load（IA）：编号最小寄存器 =Rn，编号最大寄存器 =Rn+4*(寄存器数 -1) Store（DB）：编号最小寄存器 = Rn-4*(寄存器数)，编号最大寄存器 =Rn-4
	ED	空递减堆栈，Load（IB）：编号最小寄存器 =Rn+4，编号最大寄存器 =Rn+4*(寄存器数) Store（DA）：编号最小寄存器 = Rn-4*(寄存器数 -1)，编号最大寄存器 =Rn
	FA	满递增堆栈，Load（DA）：编号最小寄存器 = Rn-4*(寄存器数 -1)，编号最大寄存器 =Rn Store（IB）：编号最小寄存器 =Rn+4，编号最大寄存器 =Rn+4*(寄存器数)
	EA	空递增堆栈，Load（DB）：编号最小寄存器 = Rn-4*(寄存器数)，编号最大寄存器 =Rn-4 Store（IA）：编号最小寄存器 =Rn，编号最大寄存器 =Rn+4*(寄存器数 -1)

表 3.19: 批量 Load/Store 指令

指令含义	指令语法
批量存储器字读取	LDM{<cond>} <address_mode>, <Rn>{!}, <registers>
用户模式的批量存储器字读取	LDM{<cond>} <address_mode>, <Rn>, <registers_without_pc>{^}
带状态寄存器的批量存储器字读取	LDM{<cond>} <address_mode>, <Rn>{!}, <registers_and_pc>{^}
批量存储器字写入	STM{<cond>} <address_mode>, <Rn>{!}, <registers>
用户模式的批量存储器字写入	STM{<cond>} <address_mode>, <Rn>, <registers>{^}

例 1：采用事先递减寻址方式向存储器写入数据：

```
;[R1-4]=R12、[R1-8]=R11、[R1-12]=R6
;[R1-16]=R5、[R1-20]=R4、[R1-24]=R3
;且:R1=R1-24
STMDB   R1!, {R3-R6, R11, R12}
```

例 2：子程序中将 8 个字从 R0 指向的位置复制到 R1 指向的位置：

```
STMFD   SP!, {R2-R9, LR}    ;寄存器堆栈,保护 R2~R9 和返回地址;
LDMIA   R0!, {R2-R9}        ;R0 指向位置的 8 个字读入 R2~R9;
STMIA   R1, {R2-R9}         ;R2~R9 中的 8 个字写入 R1 指向的位置;
LDMFD   SP!, {R2-R9, PC}    ;寄存器出栈,恢复 R2~R9,子程序返回。
```

三、存储器与寄存器数据交换指令

这种交换指令将字或无符号字节的 Load 和 Store 指令组合在一条指令中，形成一个原子操作。因此，该指令可以作为一种信号量用于进程间的同步和互斥。ARM 有两条这样的指令。指令语法格式如下：

字交换指令：

```
SWP{<cond>} <Rd>, <Rm>, [<Rn>]
```

字节交换指令：

```
SWP{<cond>}B <Rd>, <Rm>, [<Rn>]
```

其中：

<Rd>：目标寄存器，用于存放从存储器中读取的数据；

<Rm>：操作数寄存器，包含将要保存到存储器中的数据；

<Rn>：基址寄存器，包含将要访问的存储器单元地址；

<Rd> 和 <Rm> 为同一个寄存器时，指令交换寄存器和存储器中的数据；

<Rn> 不可以与 <Rd> 或 <Rm> 相同，PC 不能用做指令中的任何寄存器。

例：

```
SWP   R1, R2, [R3]    ;R1=[R3],  [R3]=R2;
SWPB  R1, R1, [R2]    ;R1 内容(低 8 位)与 [R2] 单元内容(低 8 位)互换
```

3.9.4 ARM 转移指令

ARM 转移指令用于实现程序的跳转。ARM 中实现程序跳转有两种方法：一种是直接向 PC（R15）写入目标地址值，可以实现 4GB 地址空间的任意跳转，即所谓的长跳转；另一种是使用转移指令，可以实现向前或向后 32MB 地址空间的跳转。

在 ARM v5 及以上版本中，实现了 ARM 指令集和 Thumb 指令集的混合使用，转移指令可根据目标地址的位 [0] 来确定目标程序的类型，即指令的执行既实现了子程序调用，又可以实现处理器状态的改变。

ARM 转移指令可以分为 5 条，如表 3.20 所示。

表 3.20: ARM 转移指令

指令含义	指令语法
跳转指令	B{<cond>} <target_address>
带返回（将 PC 保存到 LR 中）的跳转指令	BL{<cond>} <target_address>
带状态切换（ARM 或 Thumb）的跳转指令	BX{<cond>} <Rm>
带返回且切换到 Thumb 状态的跳转指令	BLX <target_address>（无条件执行）
带返回和状态切换的跳转指令	BLX{<cond>} <Rm>

其中：

<target_address>：程序相对偏移地址，一般由汇编器计算产生；

<Rm>：目标地址寄存器，内含跳转的目标绝对地址，其位 [0] 的值为 0 时，目标地址处为 ARM 指令，其位 [0] 的值为 1 时，目标地址处为 Thumb 指令。

例 1：执行 10 次循环：

```
        MOV R0, #10      ;初始化计数器:R0=10
LOOP    …                ;循环操作
        SUBS  R0, #1     ;计数器减 1:R0=R0-1,设置条件码
        BNE   LOOP       ;若计数器 R0 0,则重复循环操作
        …                ;否则中止循环操作
```

例 2:调用 Thumb 子程序

```
        CODE32           ;ARM 代码
        …
        BLX   TSUB       ;调用 Thumb 子程序
        …
        CODE16           ;Thumb 代码
TSUB    …                ;Thumb 子程序
        BX    R14        ;返回 ARM 代码
```

3.9.5 ARM 协处理器指令

ARM 体系结构支持通过增加协处理器(companion processor,CP)来扩展其指令集的机制。协处理器有它们自己专用的寄存器,其状态的改变完全是协处理器内部的操作,ARM 处理器并不参与这些操作。程序的控制流操作由 ARM 处理器负责,因而,协处理器指令只同数据处理和数据传送有关。每个协处理器忽略属于 ARM 处理器和其他协处理器的指令。当一个协处理器不能完成属于它的协处理器指令时,将产生未定义指令异常中断,在该异常中断处理程序中,可以通过软件模拟该硬件操作。

ARM 协处理器指令包括以下三类:

- 数据操作:用于 ARM 处理器通知 ARM 协处理器执行特定的操作,该操作完全是协处理器内部的操作,不涉及 ARM 寄存器和存储器单元。
- Load/Store:用于 ARM 协处理器从存储器读取数据装入其寄存器,或将协处理器寄存器中的数据写入存储器。ARM 处理器负责产生存储器地址,而支持的数据类型和传送的字数则由协处理器自己决定。
- 数据传送:用于 ARM 处理器寄存器与 ARM 协处理器寄存器间的数据传送操作。

ARM 协处理器指令可以分为 5 条,如表 3.21 所示。

表 3.21: ARM 协处理器指令

指令含义	指令语法
CP 数据操作	CDP{<cond>} <cp#>, <opcode_1>, <CRd>, <CRn>, <CRm>, {, <opcode_2>}
CP 从存储器读数据	LDC{<cond>} {L} <cp#>, <CRd>, <address_mode>
CP 向存储器写数据	STC{<cond>} {L}<cp#>, <CRd>, <address_mode>
ARM 寄存器到 CP 寄存器数据传送	MCR{<cond>} <cp#>, <opcode_1>, <Rd>, <CRn>, <CRm>, {, <opcode_2>}
CP 寄存器到 ARM 寄存器数据传送	MRC{<cond>} <cp#>, <opcode_1>, <Rd>, <CRn>, <CRm>, {, <opcode_2>}

其中:

<cp#>:协处理器编号;

<opcode_1>:协处理器将要执行操作的操作码 1 ;

<opcode_2>：协处理器将要执行操作的操作码 2 ；

<CRd>, <CRn>, <CRm>：协处理器寄存器，其解释与协处理器有关；

{L}：长数据操作，如双精度数据；

<address_mode>：寻址方式，支持以下 4 种方式：

1. 立即数偏移：[<Rn>,#+/-<offset_8>*4]；
2. 前变址立即数偏移：[<Rn>,#+/-<offset_8>*4]!；
3. 后变址立即数偏移：[<Rn>],#+/-<offset_8>*4 ；
4. 非偏移：[<Rn>],<option>，<option> 可用作协处理器来扩展指令；

<Rd>：ARM 寄存器，用于与协处理器寄存器间的数据传送。

例：

```
; 若 Z=1,则协处理器 p3 使用其寄存器 C1, C5, C7
; 执行操作码 1 为 6、操作码 2 为 4 的操作;
CDPEQ     p3, 6, C1, C5, C7, 4
; [R2+4]= 协处理器 p8 的 CR8,且 R2=R2+4;
STC       p8, CR8, [R2+4]!
; 协处理器 p15 使用其寄存器 C0, C2 执行操作码 1
; 为 2、操作码 2 为 4 的操作,并将协处理器寄存
; 器中的数据传送到 ARM 寄存器 R5 中
MRC       p15, 2, R5, C0, C2, 4
```

3.9.6 ARM 信号处理指令

ARM 体系结构 v5TE 中增加了用于信号处理的指令。这些指令提高了处理器处理 16 位数据乘法及饱和（saturating）加/减法的能力。

饱和运算是 DSP 指令特有的功能，当运算结果超出数据类型所能表示的范围时，它返回一个能够表示的、最接近的值。这正好与传统处理器模 2^{32} 的算法相反。

所有 ARM 信号处理的指令都不影响 CPSR 中的标志位 N、Z、C 和 V。在 v5TE 体系中，增加了 CPSR 中位 [27] 的标志 Q，称为粘接溢出标志。当 ARM 信号处理中的乘加指令和饱和加/减指令有溢出时，该标志置位，且一直保持到由 MSR 指令复位为止。

ARM 信号处理共有 9 条指令，如表 3.22 所示。

其中：

<Rd>：为目标（32 位乘积结果/加法结果）寄存器；

<RdLo>：为存放 64 位加数/加法结果低 32 位寄存器；

<RdHi>：为存放 64 位加数/加法结果高 32 位寄存器；

<Rm>：为第一个乘数所在的寄存器；

<Rs>：为第二个乘数所在的寄存器；

<Rn>：为第三个操作数的寄存器，该操作数是一个加数；

<x>：为 B 或 T，B 表示使用 Rm 低端位 [15:0]，T 表示使用 Rm 高端位 [31:16]；

<y>：为 B 或 T，B 表示使用 Rs 低端位 [15:0]，T 表示使用 Rs 高端位 [31:16]；

SAT：表示饱和运算。

例：

```
SMULBT  R3, R2, R1    ;R3=R2[15:0]*R1[31:16]
QDSUBLT R9, R0, R1    ;R9= SAT(R0-SAT(R1*2))
```

表 3.22: ARM 信号处理指令

指令含义	指令描述
有符号乘法 (16×16) 结果为 32 位	SMUL<x><y>{<cond>} <Rd>,<Rm>, <Rs> Rd=(Rm*Rs)[31:0]
有符号乘法累加 (16×16) 结果为 32 位	SMLA<x><y>{<cond>} <Rd>,<Rm>, <Rs>, <Rn> Rd=(Rm*Rs+Rn)[31:0]
有符号乘法 (32×16) 结果为高 32 位	SMULW<y>{<cond>} <Rd>,<Rm>, <Rs> Rd=(Rm*Rs)[47:16] 48 位乘法结果的高 32 位
有符号乘法累加 (32×16) 结果为 32 位	SMLAW<y>{<cond>} <Rd>,<Rm>, <Rs>, <Rn> Rd=(Rm*Rs) [47:16]+Rn 48 位乘法结果的高 32 位作为加数
有符号乘法累加 (16×16) 结果为 64 位	SMLAL<x><y> {<cond> <RdLo>,<RdHi>,<Rm>,<Rs> RdHi: RdLo = RdHi: RdLo +(Rm*Rs)[31:0]
饱和加	QADD {<cond>}<Rd>,<Rm>,<Rn> Rd=SAT(Rm+Rn)
饱和减	QSUB {<cond>}<Rd>,<Rm>,<Rn> Rd=SAT(Rm-Rn)
饱和乘 2 加	QDADD {<cond>}<Rd>,<Rm>,<Rn> Rd=SAT(Rm+SAT(Rn*2))
饱和乘 2 减	QDSUB {<cond>}<Rd>,<Rm>,<Rn> Rd=SAT(Rm-SAT(Rn*2))

3.9.7 ARM 异常及中断指令

在 ARM 体系结构中，处理器使用异常（exception）来处理在执行程序时所发生的意外事件。ARM 支持 7 种类型的异常（如表 3.23 所示），它们可被分为三类：

- 指令执行引起的直接异常：软件中断（software interrupt，SWI）、未定义指令和预取指中止（访问指令存储器故障）；
- 指令执行引起的间接异常：数据中止（访问数据存储器故障）；
- 外部产生的异常：复位、中断和快速中断。

由于上述多种异常可以同时产生，各异常需定义优先级以便确定处理器处理异常的顺序。优先级排列如表 3.23 所示。

表 3.23: ARM 异常

异常类型	处理器模式	向量地址	优先级
复位	管理（svc）	0x00000000	1（最高）
未定义指令	未定义（und）	0x00000004	6（最低）
软件中断（SWI）	管理（svc）	0x00000008	6（最低）
预取指中止	中止（abt）	0x0000000C	5
数据中止	中止（abt）	0x00000010	2
中断（IRQ）	中断（irq）	0x00000018	4
快速中断（FIQ）	快速中断（fiq）	0x0000001C	3

一、异常的进入

当异常发生时，ARM 尽量完成当前处理的指令（复位异常立即中止当前指令），然后再去处理异常。在处理异常指令之前，必须对处理器状态进行保存，以便在异常处理程序完成后，原来的程序能够继续执行。

ARM 处理器对异常中断处理的相应过程如下：

- 将链接寄存器 LR_mode(R14)设置成返回地址。
- 将当前程序寄存器 CPSR 的内容,保存到将要执行的异常中断对应的特权模式(如表 3.23 所示)SPSR_mode 中。所有异常发生时,处理器都进入相应的特权模式。
- 设置当前程序寄存器 CPSR 的相应的位。包括相应的处理器模式、ARM 状态执行、禁止中断(IRQ)及条件禁止中断(FIQ)。
- 将 PC 赋值,使程序从表 3-24 给出的相应异常中断向量地址处开始执行。

上述的 ARM 处理器对异常中断的相应过程可用如下的伪代码描述:

```
R14_<exception_mode>=return link
SPSR_<exception_mode>=CPSR
CPSR[4:0]= exception mode number
CPSR[5]=0    /* ARM 状态执行 */
if <exception_mode>==Reset or FIQ then
CPSR[6]=1    /*  禁止 FIQ 中断 */
CPSR[7]=1    /*  禁止 IRQ 中断 */
PC= exception vector address
```

二、异常的退出

一旦异常中断处理完毕,应保证程序正常返回。从异常中断处理程序中返回包括两个基本操作:一个是将 SPSR_mode 中的内容恢复到 CPSR 中;另一个是将 LR_mode 中的内容恢复到 PC 中。上述两种操作必须在一条指令中同时完成,否则将无法实现。ARM 指令在目标寄存器为 PC 时,操作码后面的修饰符 S 即表示这种特殊形式的操作。

下面给出不同异常的返回指令:

- 复位异常:不需要返回,继续执行;
- 软件中断和未定义指令异常:需返回到异常产生时指令的下一条指令,SPSR_mode 保存的为异常产生时指令的下一条指令。则返回指令为:

```
MOVS PC, LR_mode
```

- IRQ 和 FIQ 异常:需返回到异常产生时指令的下一条指令,SPSR_mode 保存的为异常产生时指令的第二条指令。则返回指令为:

```
SUBS PC, LR_mode, #4
```

- 预取指中止异常:需返回到异常产生时的指令处,SPSR_mode 保存的为异常产生时指令的下一条指令。则返回指令为:

```
SUBS PC, LR_mode, #4
```

- 数据中止异常:需返回到异常产生时的指令处,SPSR_mode 保存的为异常产生时指令的第二条指令。则返回指令为:

```
SUBS PC, LR_mode, #8
```

三、ARM 异常产生指令

ARM 有两条异常中断产生指令:软中断(SWI)指令产生 SWI 异常,可以为用户模式下的应用程序提供系统功能调用;断点中断指令(BKPT)在 ARM v5 及以上版本中引入,用于产生软件断点,供调试程序使用。指令语法格式如下:

- 软中断指令: SWI{<cond>} <immed_24>
- 断点中断指令: BKPT <immed_16>

其中：

<immed_24>：24 位立即数，被操作系统用来判断用户程序请求的服务类型；

<immed_16>：16 位立即数，被调试软件用来保存额外的断点信息。

3.9.8 Thumb 指令简介

在 ARM 体系的 T 变种版本中，同时支持 ARM 指令集和 Thumb 指令集，并且支持两者的相互调用。

Thumb 指令是 ARM 指令压缩形式的子集，它并没有改变 ARM 体系低层的程序设计模型。在 ARM 指令流水线中实现 Thumb 指令，是先进行动态解压缩，然后再作为标准的 ARM 指令来执行。Thumb 是不完整的，它只支持一些通用功能，处理器需结合 ARM 指令一起完成所有的功能。应用程序可以将 ARM 和 Thumb 子程序混合编程，用来提高系统的性能或代码密度。

所有的 Thumb 指令都是 16 位的，都有相对应的 ARM 指令，特别适合于存储系统数据总线为 16 位的应用系统。大多数 Thumb 指令是无条件执行的（只有转移指令中含条件执行），其数据处理指令采用 2 地址格式，支持 8 位字节、16 位半字和 32 位字数据类型。由于采用高密度编码，Thumb 指令格式没有 ARM 指令格式显得规则。

处理器根据 CPSR 中的 T 位来确定指令类型：当 T=0 时为 ARM 指令；当 T=1 时为 Thumb 指令。处理器复位后，ARM 启动并执行 ARM 指令；通过执行一条交换转移指令 BX 可以转换到 Thumb 指令；在 Thumb 状态时，执行 Thumb BX 指令则可以转换到 ARM 状态。进入异常中断总是进入 ARM 状态，且异常中断总是返回到异常发生前的状态。

Thumb 指令集可以分为数据处理指令、Load/Store 存储器访问指令、转移指令和异常中断指令。Thumb 指令集中没有状态寄存器访问指令、存储器与寄存器交换（信号量）指令、协处理器指令及信号处理指令。

Thumb 指令集只能对限定的 ARM 寄存器进行操作。Thumb 指令对低（Lo）8 个通用寄存器 R0～R7 具有全部访问权限，少数指令可以使用高（Hi）寄存器 R8～R15。CPSR 的条件标志位由算术和逻辑操作设置并控制条件转移。

本书中没有详细介绍 Thumb 指令集。这并不是 Thumb 指令集不重要，而是因为大部分的 Thumb 指令与 ARM 指令相类似，只不过在寄存器、立即数和寻址等方面有部分差异，在了解了 ARM 指令集的基础上很容易理解 Thumb 指令。

3.10 ARM 汇编语言程序设计

在实际的 ARM 应用系统实现中，ARM 汇编语言程序设计占有重要地位。它主要实现系统初始化代码和某些对系统运行效率影响较大的、需要进行手工优化的程序段。虽然实际中大量的代码都用 C 语言编写，但要实现系统中的一些关键功能，还需要使用 ARM 汇编语言程序设计。

本小节主要介绍在 Linux GNU 环境下的汇编程序设计，使用的编译器是交叉工具链中的 as 命令（实际 gcc 命令也会根据文件的扩展名自动调用这个工具），这里介绍的多数语法单元（除了 ARM 处理器指令）也可以使用在其他平台，包括 x86 平台。这里使用的汇编语言程序的源文件扩展名为 “.S”，注意 “S” 字母需要大写，小写 “s” 的扩展名代表不需要预处理的源文件。

3.10.1 ARM 汇编中的语句格式

ARM 汇编语言语句一般包括符号（symbol）、指令、指示符、伪指令和注释。

一、符号与标号

汇编语言中用符号来给一些汇编元素命名，例如变量和函数的命名。符号的命名可以用英文字母、数字、下划线等。符号名后面加上 “:” 代表这是一个标号，一般代表当前程序或者数据的地址。

二、常量

ARM 汇编语言中常量主要有数字常量、字符串常量。数字常量有三种表示方式：十进制数（例如 123，1，0）；十六进制数（例如 0x123，0xab），八进制数（以数字 “0” 开始，例如 0124），字符串常量与 C 语言中的定义相同，写在一对双引号之间，特殊字符用 “\” 转义。

三、注释

ARM 汇编语言最常用的注释符号是 “@”，这个字符后面的内容被视为注释。由于可以类似 C 语言的预处理工具进行预处理（扩展名为 S），所以也可以使用 C 语言中的注释方法。例如 “//” 表示单行注释，“/*” 与 “*/” 表示多行注释。另外还可以使用 “;” 和 “#” 作为单行注释。

3.10.2 ARM 汇编中的指示符

指示符不像汇编指令那样在处理器运行期间由处理器执行，而是在汇编器对源程序汇编期间由汇编程序处理的，提供一些定义、指示等操作。GNU 汇编中使用的指示符都以 “.” 开头，例如 “.global”。

一、符号定义指示符

符号定义指示符用来定义汇编语言中使用的符号。常用的这类指示符有：

.global：定义一个全局变量

.set：为一个符号赋值，等效的指示符还有 “.equ”

.equiv：与 “.set” 类似，只不过当符号已经定义过的时候，编译器会认为错误。

例：

```
.global sum1          ;声明一个全局变量
.set sum1, 0xff       ;给该变量赋值
```

二、数据定义指示符

数据定义指示符用于内存结构定义及内存分配。常见的这类指示符有

.byte：定义一系列字节类型数据

.long：定义 32 位的数据（数据宽度与平台相关）

.ascii：定义字符串常量

.2byte/.4byte/.8byte：在 ARM 平台分别代表 2/4/8 字节的数据列表

例：

```
.ascii “Hello Word!\000”
.byte 74, 0x25, 0127
```

三、汇编控制指示符

汇编控制指示符用于控制汇编程序对源程序的汇编操作。

四、条件汇编

“.if” 指示符控制汇编程序的操作类似其他语言的 “判断” 或 “条件执行” 等流程控制语句，它们能够根据条件把一段代码包括在汇编语言程序内或者将其排除在程序之外。

语法格式：

```
.if logical expression
Instructions or derectives
{.else
Instructions or derectives
}
.endif
```

同时还有一些 “.if” 指示符的变种，如 “.ifdef”（是否定义了某符号），“.ifge”（表达式大于等于 0）等。

五、宏定义

“.macro” 和 “.endm” 指示符用于定义一个宏，程序中可以通过宏指令多次调用该代码段，在宏定义中还可以使用 “.exitm” 指令从宏中退出。

语法格式：

```
.macro macname macargs...
.endm
```

其中 macargs 表示名称为 “macname” 的宏的参数，在宏内部可以通过 “\” 加上参数名来引用这个参数。

例如：

```
.macro DBG val
LDR  R6, =\val
.endm
```

六、其他杂类指示符

杂类指示符用于定义段、指令状态及符号引用等操作。

.align expr1[,expr2,expr3]：通过填加补丁字节使当前位置按 2^{expr1} 地址对齐，填充字节可以使用 expr2，最多填充字节由 expr3 指定。

.code16/.thumb：指示后面的指令为 16 位的 Thumb 指令

.code32/.arm：指示后面的指令为 32 位的 ARM 指令

.section：定义源代码的段名

.text：定义为 “.text” 段

3.10.3 ARM 汇编中的伪指令

ARM 汇编器支持下面几种伪指令，方便用户源程序的编写。这些伪指令在汇编时将被替换成相应的 ARM 或 Thumb 指令。

一、ADR 伪指令

ADR 伪指令用于将基于 PC 或基于寄存器的地址值调进寄存器中。

语法格式：

```
ADR register,expr
```

其中：

register 为目标寄存器

expr 为基于 PC 或基于寄存器的地址表达式，取值范围：

- -255~255 非字对准地址；
- -1020~1020 字对准地址

二、ADRL 伪指令

ADRL 伪指令类似于 ADR 伪指令，用于将基于 PC 或基于寄存器的地址值调进寄存器中，只是比 ADR 可加载更大范围的地址。

语法格式：

```
ADRL register,exr
```

其中：

register 为目标寄存器

expr 为基于 PC 或基于寄存器的地址表达式，取值范围：

- -64K~64K 非字对准地址；
- -256K~256K 字对准地址

三、LDR 伪指令

LDR 伪指令用于将一个 32 位常数或一个地址值调进寄存器中。

语法格式：

```
LDR register, =[expr | label-expr]
```

其中：

register 为目标寄存器

expr 为 32 位的常数，汇编器将根据 expr 的值如下处理该伪指令：

- expr 在 MOV 或 MVN 的取值范围内，LDR 将由 MOV 或 MVN 指令替代；
- expr 超出 MOV 或 MVN 的取值范围，汇编器将 expr 放入数据缓冲池内，同时用一条基于 PC 的 LDR 伪指令读取 expr

label-expr 为基于 PC 的地址表达式或外部表达式：

- 当 label-expr 为基于 PC 的地址表达式时，汇编器将 label-expr 放入数据缓冲池内，同时用一条基于 PC 的 LDR 伪指令读取 label-expr ；
- 当 label-expr 为外部表达式或者非当前段的表达式时，汇编器将一个链接重定位伪操作插入目标文件中，链接器在链接时生成该地址。

四、NOP 伪指令

NOP 伪指令用于空操作。

语法格式：

```
NOP
```

汇编器在汇编时将该伪指令替换为 ARM 中的空操作，如：MOV R0,R0

3.10.4 ARM 汇编语言程序格式

ARM 汇编语言程序可以由多个源文件组成,每个源文件的汇编语言以段(section)为单位进行组织。段可分为代码段和数据段,一个 ARM 汇编源程序至少有一个代码段,较大的程序可以包含多个代码段和多个数据段。

下面以 “Hello World” 程序为例说明 ARM 汇编源程序的基本结构:

```
        .section    .rodata
        .align 2
str:
        .ascii      "Hello world!\000"
        .text
        .align 2
        .global     main
main:
        MOV         IP, SP
        STMFD       SP!, {FP, IP, LR, PC}
        SUB         FP, IP, #4
        SUB         SP, SP, #8
        LDR         R0, .L3
        BL          puts
        SUB         SP, FP, #12
        LDMFD       SP, {FP, SP, PC}
        .align 2
.L3:
        .word       str
```

3.11 实验 :ARM 汇编语言程序设计

实验目的

1. 了解在没有操作系统前提下编写测试程序的方法
2. 对 ARM 汇编进行初步了解
3. 对程序的链接原理进行感性的认识

实验内容

一、无操作系统下的汇编程序设计

下面一段程序是循环点亮开发板调试 LED 的程序,程序中的 LEDADD 是调试 LED 的内存映射地址,写入这个地址的内容将反映在 LED 上。

```
#define LEDADD          0x10000010

        .macro  DBG     val

        LDR             R6, =LEDADD
        LDR             R7, =\val
        STRB            R7, [R6]

        .endm
loop:
        DBG     0x2f4f8f
        LDR     R2, =0x01010100
1:
        SUB     R2, R2, #1
```

```
        CMP     R2, #0
        BGT     1b

        DBG     0X00
        LDR     R2, =0x01000000
1:
        SUB     R2, R2, #1
        CMP     R2, #0
        BGT     1b

        B       loop
```

由于这个程序并没有一个类似 C 语言中的 main 函数，所以在编译的时候必须指定链接器链接各部分目标代码的相对位置。下面是一个链接脚本例子：

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
SECTIONS
{
        . = 0x20001000;   /* 程序的起始位置 */

        . = ALIGN(4);
        .text      :      /* 代码部分 */
        {
          start.o(.text)  /* start.o  的代码 */
          *(.text)        /* 其他模块的代码（本例中没有）*/
        }
        . = ALIGN(4);
        .rodata : { *(.rodata) }  /* 只读数据部分 */

        . = ALIGN(4);
        .data : { *(.data) }      /* 已初始化的数据部分 */
        . = ALIGN(4);
        __bss_start = .;
        .bss : { *(.bss) }        /* 未初始化数据部分 */
        _end = .;
}
```

链接脚本首先定义可执行程序的输出类型为 elf32-littlearm，接下来是最重要的 SECTION 部分，这里定义了编译后程序各部分在内存中的分布。在我们这个程序中，很多部分并不存在，但在比较复杂的与 C 语言的混合编程中就必须存在。

关于链接脚本的具体语法可以参考 ld 命令的文档，在上面的例子中需要注意每个 “=” 号的前后都需要由空格分隔。

编译这个文件对应的 Makefile 如下（假设程序名称为 start.S，链接脚本文件的名字是 RAM.lds）：

```
all:
        arm-none-linux-gnueabi-gcc -c -o start.o start.S
        arm-none-linux-gnueabi-ld -TRAM.lds  start.o -o boot
        arm-none-linux-gnueabi-objcopy -O binary boot boot.bin
        arm-none-linux-gnueabi-objdump -d boot > bootdump
```

其中由 ld 命令生成的文件 boot 是所谓的 ELF 文件格式，并不能直接加载到内存运行，必须用 objcopy 命令将其转换为二进制格式，也就是内存映像格式。因此 boot.bin 才是最终需

要加载到内存中执行的文件。最后用 objdump 命令对 boot 文件进行反汇编，可以根据生成的 bootdump 文件分析链接的过程是否是对的。bootdump 文件的前面几行如下：

```
boot:     file format elf32-littlearm

Disassembly of section .text:

20001000 <loop>:
20001000:        e59f6038        ldr        r6, [pc, #56]        ; 20001040 <loop+0x40>
20001004:        e59f7038        ldr        r7, [pc, #56]        ; 20001044 <loop+0x44>
20001008:        e5c67000        strb        r7, [r6]
2000100c:        e51ff2f0        ldr        pc, [pc, #-752]         ; 20000d24 <loop-0x2d
20001010:        e59f2030        ldr        r2, [pc, #48]        ; 20001048 <loop+0x48>
20001014:        e2422001        sub        r2, r2, #1        ; 0x1
```

可以看出第一条指令的地址为我们在 RAM.lds 中设置的 20001000，因此在以后的实验内容中也必须将其加载到这个地址。

由于这个程序并没有进行硬件的初始化，所以只能在 bootloader 中被调用执行，利用 bootloader 已经初始化好的执行环境。首先把 boot.bin 放到主机的 tftp 服务的主目录中，然后在 U-Boot 中对其进行测试。

```
U-boot> tftp 20001000 boot.bin
#
U-boot> go 20001000
```

由于示例程序是一个死循环程序，因此在执行程序之后只能通过复位开发板的办法重新获得 U-Boot 的提示符。

二、汇编语言和 C 语言的混合编程

只要在汇编程序中适当的位置跳转到 main 函数，就可以简单地实现混合编程。此时编写的源程序包含一个或多个 C 语言源程序，并包含一个汇编程序。汇编程序可以比较简短，仅做基本的环境初始化，而在我们的实验环境，可以只包含一条跳转语句。例如：

```
.globl main
b main
```

假定我们编写的汇编程序为 start.S，C 语言程序为 main.c。由于在链接脚本中指定了 start.o 排在所有代码之前，所以可以保证程序运行的第一条指令就是 start.S 中的第一条指令。否则程序的执行顺序就无法确定。因为 main.o 中的第一条代码不一定是其中 main 函数的代码。

下面请利用混合编程的方法，用 C 语言控制调试 LED，编写比较复杂的显示花样，并修改 Makefile 文件，以适应同时编译两个源文件的需求。

第四章　处理器与开发板

在第三章的介绍中，我们已经详细了解了 ARM 体系结构、指令系统和 ARM 处理器核的很多特点。实际的 ARM 处理器产品不同于大家比较熟悉的 x86 架构处理器，它不但包括中央处理单元、片上高速缓存，还包括很多外设模块。例如几乎每款 ARM 处理器芯片都包括片上的串口，仅仅通过电平转换就可以直接与标准串口相连。这种设计一方面使得基于 ARM 处理器的硬件设计变得简单，很少需要其他外围芯片；另一方面使得 ARM 产品缺乏如同 x86 架构的标准，不同厂家的 ARM 处理器在实现同样的外设时可能采用不同的软件接口，给软件开发带来一定的负担。

本章主要介绍 Atmel 公司的 AT91SAM9261 芯片，并介绍以这款芯片为核心的硬件开发平台。后面的实验内容也主要在这个开发平台上实现，如果应用在其他平台需要做必要的改动。

4.1　AT91SAM9261 芯片概述

AT91SAM9261 芯片基于 ARM926EJ-S 处理器核，具有 DSP 指令扩展，支持 Java 加速功能，具有 16KB 的数据高速缓存和 16KB 的指令高速缓存，具有 MMU 内存管理单元，时钟频率达到 190MHz。

AT91SAM9261 芯片还集成了如下外设：

- 额外的内部存储器：包含 32KB 的内部 ROM，160KB 的内部 SRAM。
- 外部总线接口：支持 SDRAM，NAND Flash，CF，和其他静态内存。
- LCD 控制器：支持主动与被动模式。STN 模式支持最高 16 位彩色，TFT 模式最高支持 2048×2048 分辨率，16M 颜色。
- USB 控制器：USB 2.0 全速（12Mbps）2 个主设备端口，1 个从设备端口
- 总线阵列：支持不同的启动模式，内存重映射；支持 5 个主设备与 5 个从设备
- 复位控制单元：上电复位，复位源确定与复位信号控制。
- 关闭与唤醒控制单元
- 时钟发生单元：包括 1 个 32768Hz 的低功耗时钟源，和 1 个 3~20MHz 的时钟震荡电路与两个锁相环。
- 电源管理单元：4 个可编程外部时钟源。
- 高级中断控制器：可以分别屏蔽、八级优先级中断源，3 个外部中断源和 1 个快速中断源。
- 调试单元：支持调试功能的串口。
- 定时器单元：20 位内部定时器与 12 位内部计数器
- 看门狗单元：12 位计数器、单次编程、密码保护。
- 实时钟单元：32 位实时钟，采用慢时钟单元驱动。
- 3 个 32 位并行输入输出控制器：与外设共用接口，每个 IO 线都有中断触发能力，每个 IO 线都可以设置异步输出、上拉电阻和漏级开路。
- 19 个外设 DMA 通道

- 多媒体卡接口：支持 SD 卡，MMC 卡，自动协议控制，支持 PDC、MMC、SD 兼容的高速数据传输。
- 3 个异步串行控制口：发送与接收模块有独立的时钟与帧同步控制，支持 I2S 模拟接口与时分复用。
- 3 个通用异步/同步串行口（USART）：支持 IrDA 调制与解调接口，支持 ISO7816 T0/T1 智能卡，支持 RS485 协议。
- 2 个 SPI 接口：8 位到 16 位数据长度、4 个外部片选。
- 3 通道 16 位定时/计数器：支持 PWM，3 个外部时钟输入、每通道 2 个多用 IO。
- I^2C 接口：支持主模式，支持全部 Ateml 的 EEPROM 设备。
- JTAG 接口支持

AT91SAM9261 的硬件框图如图 4.1 所示。

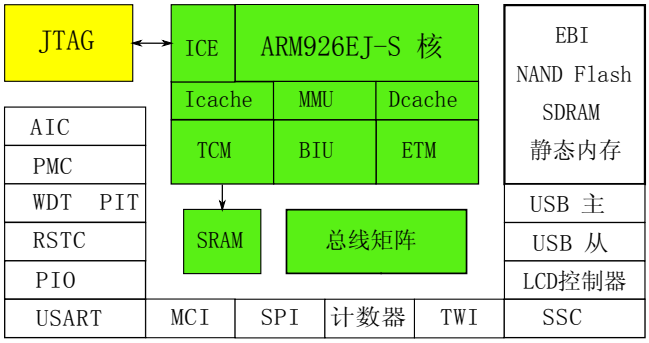

图 4.1: ATSAM9261 硬件框图

4.2 处理器内存布局

ARM 处理器采用内存映射的方式来管理外设的寄存器，不同外设的寄存器单元被分配到了处理器存储空间的不同位置，如表 4.1 所示。如果要在软件中访问处理器的外设单元，就必须了解特定外设寄存器的物理地址。

表 4.1: AT91SAM9261 的内存布局

内存起始地址	描述
0x0000 0000	内部存储器（256M）
0x1000 0000	外部总线接口片选 0（256M）
0x2000 0000	外部总线接口片选 1（256M）
0x3000 0000	外部总线接口片选 2（256M）
0x4000 0000	外部总线接口片选 3（256M）
0x5000 0000	外部总线接口片选 4（256M）
0x6000 0000	外部总线接口片选 5（256M）
0x7000 0000	外部总线接口片选 6（256M）
0x8000 0000	外部总线接口片选 7（256M）
0x9000 0000	未定义（1518M）
0xF000 0000	集成外设接口

表 4.2 中为 AT91SAM9261 的外设寄存器起始地址，其中最后两个外设的映射地址在内部存储器的范围内。

表 4.2: AT91SAM9261 外设控制寄存器分布

起始地址	外设单元
0xFFFA 0000	定时器/计数器 TC0,TC1,TC2
0xFFFA 4000	USB 从设备端口 UDP
0xFFFA 8000	多媒体卡接口 MCI
0xFFFA C000	I^2C 总线接口 TWI
0xFFFB 0000	串口 0 USART0
0xFFFB 4000	串口 1 USART1
0xFFFB 8000	串口 2 USART2
0xFFFB C000	同步串口 0 SSC0
0xFFFC 0000	同步串口 1 SSC1
0xFFFC 4000	同步串口 2 SSC2
0xFFFC 8000	SPI 接口 0 SPI0
0xFFFC C000	SPI 接口 1 SPI1
0xFFFC D000	保留
0xFFFF C000	系统控制单元 SYSC
0xFFFF EA00	动态内存控制器 SDRAMC
0xFFFF EC00	静态内存控制器 SMC
0xFFFF EE00	内存矩阵 MATRIX
0xFFFF F000	中断控制器 AIC
0xFFFF F200	调试串口 DBGU
0xFFFF F400	并口 A PIOA
0xFFFF F600	并口 B PIOB
0xFFFF F800	并口 C PIOC
0xFFFF FA00	保留
0xFFFF FC00	电源控制器 PMC
0xFFFF FD00	复位控制器 RSTC
0xFFFF FD10	关闭唤醒控制器 SHDWC
0xFFFF FD20	实时钟 RTT
0xFFFF FD30	周期性定时器 PIT
0xFFFF FD40	看门狗定时器 WDT
0xFFFF FD50	通用后备寄存器 GPBR
0xFFFF FD60	保留
0x0050 0000	USB 主设备端口
0x0060 0000	LCD 控制器

AT91SAM9261 内部的 160KB SRAM 可以分成三个部分:SRAM_A、SRAM_B 与 SRAM_C,其大小都可以通过软件配置。其中 SRAM_A 和 SRAM_B 为 ARM926EJ-S 核的紧耦合内存(TCM,Tightly Coupled Memory),分别用来保存指令与数据。它们可被配置的大小是 0KB、16KB、32KB 与 64KB,其余的部分为 SRAM_C。在软件重映射(见后文)之前,SRAM_C 可以在地址 0x300000 访问,而 SRAM_A 与 SRAM_B 可以分别在地址 0x100000 和 0x200000 访问,SRAM_A 与 SRAM_B 可以通过 CP15 协处理器重新映射到内存的其他地方去。

为了获得更灵活的启动方式,AT91SAM9261 的内部存储器支持可配置的映射模式。当 BMS 引脚为高的时候(BMS=1),芯片复位启动的时候内部 ROM 映射到 0 地址,也即系统从内部 ROM 中的代码开始运行。当 BMS 引脚为低的时候(BMS=0),芯片复位启动的时候,0 地址

为外部总线片选 0(NCS0)所选择的内存模块,也即系统从外部片选 0 的存储设备中的代码开始运行。

AT91SAM9261 还可以通过软件的方式重新映射内部存储器(REMAP),经过重映射后,内部 0 地址映射为 SRAM_C,这样可以方便的更改 ARM 处理器的中断向量,便于程序开发。

4.3 AT91SAM9261 内部启动逻辑

AT91SAM9261 芯片内部固化了一段启动代码,这部分代码保存在内部 ROM 中,当系统复位的时候,如果 BMS 引脚读出为 1,则系统从这部分代码开始启动。因此这部分代码就像系统的 BIOS 或者一个简单的 Bootloader,可以使基于 AT91SAM9261 芯片的开发更加方便。

图 4.2 是 AT91SAM9261 的启动代码流程,首先它进行系统的设备初始化,主要对调试串口和 USB 从设备端口进行配置。设备配置成功后,我们可以从串口读到 “RomBOOT” 的信息,接下来软件会按顺序分别从 SPI 和 NAND Flash 设备查找启动代码,二者都失败就会进入 SAM-BA 启动模式。

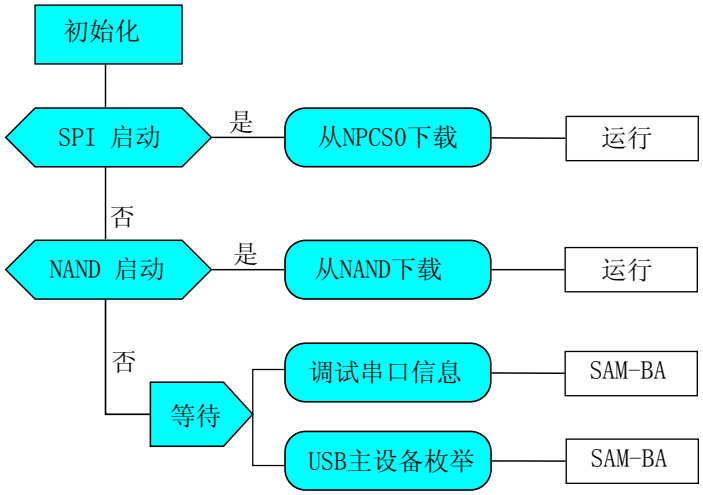

图 4.2: AT91SAM9261 启动代码流程

无论从 SPI 设备启动还是从 NAND Flash 启动,其执行的步骤都是类似的。首先要从 SPI 或者 NAND Flash 设备上寻找合法的启动程序。判断存储内容是否合法的根据是程序的最开始必须是 ARM 的 7 个中断向量,每个中断向量都是一条跳转语句或者 LDR 加载 PC 值语句,而第六个中断向量的位置是要加载的程序内容的大小。如果找到了合法的加载程序,就把程序代码复制到内部 SRAM 中,然后执行软件内存重映射,将 SRAM 映射到 0 地址,然后再把 PC 指针赋值为 0,开始执行刚加载的程序。这部分程序一般为更加通用的 Bootloader 代码。

如果在 SPI 和 NAND Flash 设备中都没有找到合法的启动代码,启动程序就转入了我们已经接触过的 SAM-BA 模式。在这个模式下,芯片等待调试串口或者 USB 口的消息,并对已经输入的命令进行处理。第二章实验中介绍的 SAM-BA 软件就是利用这个协议,如表 4.3 所示,来实现加载程序和对 Flash 的编程的。

4.4 AT91SAM9261 的集成外设

不同的 ARM 芯片都是基于 ARM 公司推出的处理器核心,造成它们区别的主要是它们所集成的外设。我们在选择 ARM 芯片的时候也会通过考察它们的外设功能来进行取舍。一种外设功能是集成于片内,还是需要在外部添加芯片,对于软件设计来说差别不大,但对硬件设计却有很大的影响。当大部分需要的外设都是集成在片内的时候,硬件设计就变得简单,产品所占用的空间就会比较小,设备外形设计就更加灵活。

表 4.3: SAM-BA 命令格式

命令	说明	参数	例子
O	写一个字节	Address,Value#	O2000001,CA#
o	读一个字节	Address,#	o2000001,#
H	写一个半字	Address,Value#	H2000002,CAFE#
h	读一个半字	Address,#	h2000002,#
W	写一个字	Address,Value#	W2000004,DEADBEEF#
w	读一个字	Address,#	w2000004,#
S	发送文件	Address,#	S2000000,#
R	接收一个文件	Address,Bytes#	R2000000,1200#
G	运行	Address#	G200200#
V	显示版本号	无参数	V#

当一个厂家推出 ARM 芯片产品的时候，也往往会根据不同的市场需求，设计不同的处理器芯片。常见的处理器类别主要有应用处理器和嵌入式处理器，其中前者要求芯片有更好的性能，支持复杂的人机界面，往往应用于有多媒体需求的手持设备。嵌入式处理器则要求芯片具有更高的性价比，实时处理能力更强，功耗更低，外形和封装更经济。另外还有一些是面向特殊应用的处理器产品，如对安全性有高要求的应用需要部分程序具有独立的地址空间。

AT91SAM9261 芯片所包含的外设非常丰富，封装形式为 217 管脚的 BGA 封装，主要面向手持型智能设备应用。下面我们就对 AT91SAM9261 的部分常用外设进行介绍，如果需要了解更加详细的内容和对其他外设的说明，请参考 AT91SAM9261 的数据手册。

4.4.1 时钟发生器

对于同步数字逻辑来说，时钟是最重要的组成部分，AT91SAM9261 的时钟发生电路主要由两个锁相环（phase locked loop，PLL）、一个主振荡器和一个 32.768KHz 的低功耗振荡器组成。

32.768KHz 的低功耗振荡器被称为 SLCK（slow clock），是 AT91SAM9261 芯片中唯一的永久性时钟，在芯片复位和芯片时钟配置改变之后，主要依靠 SLCK 维持正常的芯片运行和重新建立稳定的其他时钟信号。SLCK 依靠连接在 XIN32 和 XOUT32 两个管脚间的石英晶体提供。

主振荡器被称为 MAINCK，也可以由石英晶体提供，对应的芯片管脚为 XIN 和 XOUT。可选的晶体频率为 2-20MHz，我们的开发板选择比较常用的 18.432MHz 晶体。芯片复位后，主振荡器是被禁止的，软件可以通过 CKGR_MOR 寄存器的 MOSCEN 位来控制主振荡器的使用。当软件允许使用主振荡器后，还必须设置一个启动计数器 OSCOUNT，这个计数器用来计算主振荡器的稳定时间。OSCOUNT 计数器每 8 个 SLCK 周期执行减 1 运算，当计数到 0 的时候，PMC_IMR 寄存器的 MOSCS 位被设置，表示主振荡器已经稳定有效。由于 OSCOUNT 最大可以设置为 255，所以主振荡器的稳定时间最长为大约 62 毫秒。OKGR_MOR 寄存器的地址为 0xffffc20，其定义如下：

二进制位	15-8	1	0
名称	OSCOUNT	OSCBYPASS	MOSCEN

其中 OSCBYPASS 位仅在不使用主晶体振荡器时使用，此时用户需要在 XIN 引脚输入一个外部时钟作为主振荡器的替代时钟。当 OSCBYPASS 设置为 1 时，MOSCEN 必须同时设置为 0 才能使外部时钟生效。

处理器的主振荡器频率还可以通过 CKGR_MCFR 寄存器获得。虽然振荡器的频率一般由系统设计者提供，但软件可以通过这个寄存器独立的获得这个频率值。CKGR_MCFR 寄存器的地址为 0xffffc24，其位定义如下：

二进制位	16	15-0
名称	MAINRDY	MAINF

处理器根据 SLCK 来计算主振荡器的频率，当主振荡器输出稳定之后，MAINRDY 位开始对主振荡器进行计数，直到 SLCK 的第 16 个下降沿才结束计数，同时设置 MAINRDY 标志位为 1，表示 MAINF 中的计数值有效。因此 MAINF 中的数值为 16 个 SLCK 周期的计数值，可以根据这个关系算出外接主振荡器的频率。

AT91SAM9261 包含两个锁相环，分别是 PLLA 与 PLLB，它们提供的时钟信号被称为 PLLACK 与 PLLBCK。锁相环的输出频率为 80MHz 到 240MHz，最小的输入频率为 1MHz。锁相环的控制通过 CKGR_PLLAR（地址：0xfffffc28）与 CKGR_PLLBR（地址：0xfffffc2c）。其中 CKGR_PLLBR 的定义如下：

二进制位	29-28	26-16	15-14	13-8	7-0
名称	USBDIV	MULB	OUTB	PLLBCOUNT	DIVB

CKGR_PLLAR 的定义与之类似，只是其 29 位固定为 1，28 位没有定义。其他各位名称分别是把上表中的 “B” 替换为 “A”。后文用没有 “B” 的名称表示同时适用于两个 PLL。

PLL 锁相环的输入为 MAINCK，输出频率根据如下公式计算：

PLLCK=MAINCK(MUL+1)/DIV。

当处理器复位后，DIV 的值为 0，此时 PLL 的输出为 0，软件必须根据需要的频率重新设置 MUL 与 DIV 的值。当 MUL 设置为 0 时，锁相环会停止工作，相应的功耗会降低。当重新启动 PLL 或者改变 PLL 的参数的时候，PMC_SR 的 LOCK 位（LOCKA 或 LOCKB）会置为 0，同时 PLLCOUNT 中的数值会被放入一个计数器，以 SLCK 的速度进行减计数，当计数为 0 的时候 LOCK 为重新置位，表示 PLL 的输出已经稳定。PLLCOUNT 的值必须由用户进行合理的设置。其中 OUT 的内容需要根据 PLL 的输出频率进行确定，当输出频率在 80MHz 到 200MHz 之间时，OUT 设置为 “00”，当输出频率在 190MHz 到 240MHz 之间时，OUT 需要设置为 “10”。

在 Atmel 提供的启动代码中，CKGR_PLLAR 被设置为 0x2060BF09，CKGR_PLLBR 被设置为 0x10483F0E。根据 MAINCK 为 18.432MHz，通过计算可以得到两个锁相环的输出频率为：

PLLACK=198.656MHz，PLLBCK=96.1097MHz。

除了前面提到的 4 个时钟源以外，AT91SAM9261 还定义了一些派生时钟，用来控制片上的不同外设，以获得节省功耗的效果。这些时钟包括：

- MCK，即主时钟，用来给芯片中永远运行的模块提供时钟，例如中断控制器和内存控制器
- PCK，处理器时钟，可以在处理器处于休眠模式中被关闭
- 外设时钟，为集成外设提供的时钟，在数据手册中也被称为 MCK。它们可以独立地被关闭以节省能量。
- UDPCK，USB 从设备时钟
- 由 PCK 管脚驱动的时钟信号

派生时钟由 PMC 来控制，其中 MCK 与 PCK 的关系如图 4.3 所示。

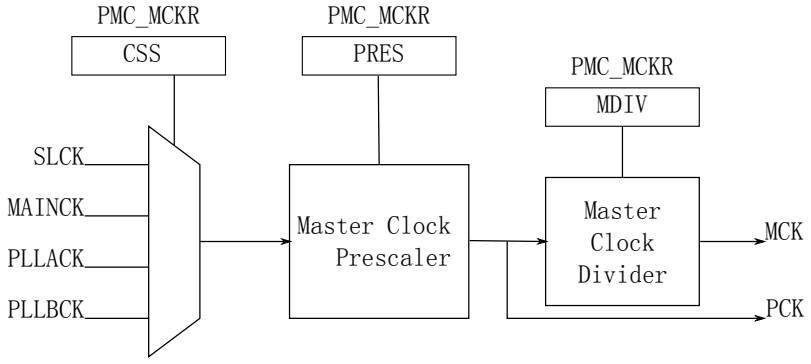

图 4.3: 时钟发生器结构图

MCK 可以从时钟发生器的四个时钟源选择，当 CSS 设置为 “00” 时，SLCK 被选择；当 CSS 设置为 “01” 时，MAINCK 被选择；当 CSS 设置为 “10” 时，PLLACK 被选择；当 CSS 设置为 “11” 时，PLLBCK 被选择。PRES 用来对选择的时钟进行分频，其取值范围是 0-6，对应的分频系数为 2^{PRES}，其输出就是 PCK。MDIV 用来 PCK 进行分频，其取值范围为 0-2，对应的分频系数为 2^{MDIV}。PMC_MCKR 寄存器的地址在 0xfffffc30，其定义如下：

二进制位	9-8	4-2	1-0
名称	MDIV	PRES	CSS

USB 从设备时钟固定由 PLLB 产生，此时 PLLBCK 的频率必须为 48MHz、96MHz 或者 192MHz，经过分频后获得 48MHz 的 USB 时钟，允许的频率误差为 0.25%。其中的分频系数由 CKGR_PLLBR 中的 USBDIV 决定，为 2^{USBDIV}。

集成外设的时钟控制由 PMC_PCER（地址：0xfffffc10）、PMC_PCDR（地址：0xfffffc14）与 PMC_PCSR（地址：0xfffffc18）寄存器来控制。每个寄存器的第 2 到第 31 位用来控制编号为 2 到 31 的集成外设的时钟。当在 PMC_PCER 中某二进制位写入 1 的时候，表示对应集成外设的时钟被允许，当在 PMC_PCDR 中某二进制位写入 1 的时候，表示对应集成外设的时钟被禁止，当前的时钟状态可以从 PMC_PCSR 寄存器读出。集成外设的编号可以参考数据手册，例如 PIOA、PIOB 和 PIOC 的外设编号分别为 2、3、4，USB 主设备的编号为 20。

PMC 还支持四个可编程的时钟引脚 PCK0-PCK3，PCKx 可以被独立地从 SLCK、MAINCK、PLLACK 与 PLLBCK 中选择，同时还支持对输入时钟的预分频。设置 PCKx 的寄存器为 PMC_PCKx，地址从 0xfffffc40 到 0xfffffc4c，定义如下：

二进制位	4-2	1-0
名称	PRES	CSS

PRES 与 CSS 的含义同 PMC_MCKR 寄存器相同。PCKx 可以通过 PMC_SCER（地址：0xfffffc00）和 PMC_SCDR（地址：0xfffffc04）寄存器被独立地禁止和允许。由于 PCKx 在被重新配置时可能引起时钟的毛刺，当修改 PCKx 的配置时，应该先把对应的时钟禁止，修改成功后再重新允许。PMC_SCER 与 PMC_SCDR 寄存器的定义如下：

二进制位	17	16	11	10	9	8	7	6	0
名称	HCK1	HCK0	PCK3	PCK2	PCK1	PCK0	UDP	UHP	PCK

其中 HCK1 为 LCD 控制器的时钟控制，HCK0 和 UHP 为 USB 主设备的时钟控制，UDP 为 USB 从设备的时钟控制。PCK 为处理器时钟的控制，它只能被禁止，然后通过中断时间把处理器唤醒。时钟的禁止状态可以从 PMC_SCSR（地址：0xfffffc08）读出，对应位含义与上表相同。

4.4.2 高级中断控制器

AT91SAM9261 的中断控制器（AIC）用来管理芯片的外部中断和集成外设中断，并根据配置情况触发 ARM 处理器的 IRQ 或者 FIQ 中断。如图 4.4 所示，AIC 一共支持 32 个可单独屏蔽的中断源，其中 0 号中断对应 FIQ 管脚，1 号中断对应系统设备（如 PIT，RTT，PMC 等），2 号到 31 号中断对应集成的外设或外部中断（与集成外设的编号对应）。AIC 支持 8 个不同的优先级，允许高优先级的中断优先被服务。外部中断还可以被设置上升沿触发/下降沿触发或高电平触发/低电平触发，内部中断源可以被设置为电平触发或者边沿触发（具体电平或边沿并不需要用户关心）。

每个中断源的触发模式和优先级由 AIC_SMR 寄存器来确定，与 32 个中断源对应，AIC_SMR 寄存器实际包含 32 个寄存器，编号从 0 到 31，偏移地址从 0x00、0x04 到 0x7C。其定义如下：

二进制位	6-5	2-0
名称	SRCTYPE	PRIOR

对于外部中断源来说，其中 SRCTYPE 为 “00” 代表低电平触发，“01” 代表下降沿触发，“10” 代表高电平触发，“11” 代表上升沿触发。对于内部中断源，“00”、“10” 都代表电平触发，“01”、“11” 都代表边沿触发。PRIOR 位代表 0-7 的优先级，其中 7 为最高级。

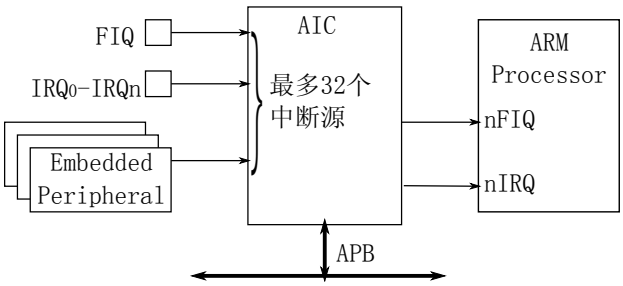

图 4.4: 中断控制器结构图

AIC 支持对 32 个中断源的向量化，通过设置 AIC_SVR（编号 0 到 31，偏移地址从 0x80 到 0xFC）寄存器为相应中断服务程序的入口地址，当中断发生时，AIC_IVR 寄存器（偏移 0x100）中的值就是设置在 AIC_SVR 中的值，只要通过指令

```
LDR PC, [PC, #-0xF20]
```

就可以跳转到相应的中断服务程序入口，加快了中断处理程序的执行速度。例如在发生 IRQ 中断的时候，ARM 处理器会执行位于 0x00000018 位置的代码，当这个代码是

```
LDR PC, [PC, #0xF20]
```

时，由于此时的 PC 值为 0x00000020，下一条要执行的语句的地址由 0x00000020 - 0xF20 = 0xFFFFF100 决定，这个地址恰好为 AIC_IVR 寄存器的位置。这种方式在对实时处理要求比较高的场合非常有用。如果 AIC 的 32 个中断源都没有触发中断，在 AIC_IVR 中读出的值为设置在 AIC_SPU 寄存器（偏移 0x134）中的值。

当读取 AIC_IVR 寄存器的时候，表示中断服务程序开始执行，当中断服务程序执行完毕之后，需要写入 AIC_EOICR 寄存器（偏移 0x130），具体写入的内容无关紧要，处理器通过这个写入过程来确定中断服务程序已经完成。

AIC 的每个中断源可以独立被屏蔽。AIC_IECR 寄存器（偏移 0x120）用来设置允许中断，当其某二进制位设置为 1，则相应编号的中断号被允许，设置为 0 的位不起作用。AIC_IDCR 寄存器（偏移 0x124）用来设置禁止中断，当其某二进制位设置为 1，则相应编号的中断号被屏蔽，设置为 0 的位不起作用。AIC_IMR 寄存器（偏移 0x110）用来读取中断的屏蔽状态，读出为 1 的位表示相应的中断被允许。

AIC 的 0 号中断会触发 FIQ，同时还允许通过 AIC_FFER 寄存器（偏移 0x140）设置其他中断源触发 FIQ 中断，与之对应的还有 AIC_FFDR 寄存器（偏移 0x144）禁止 FIQ 中断和 AIC_FFSR 寄存器（偏移 0x148）返回快速中断的设置状态。

4.4.3 通用 IO 管脚

AT91SAM9261 支持三组并行通用 IO 管脚（PIO），每组 IO 管脚可包含 32 位可编程的输入/输出管脚。每个管脚可以作为通用 IO 使用，也可以作为某个集成外设的功能管脚使用。当作为通用 IO 使用时，每个管脚可以设置为中断输入源，当管脚电平发生变化时触发中断。每个管脚还提供了小于半个时钟周期的抖动消除（glitch filter）功能，支持类似漏极开路的多驱动模式，并支持软件可控的内部上拉电阻。

每组 PIO 控制器都有一系列的控制寄存器，表 4.4 仅给出了这些寄存器的偏移地址，它们的起始地址可以参考表 4.2。

以上寄存器每个都是 32 位宽度，因此寄存器的每一位都对应一个 PIO 的管脚。例如 PIO_PER 寄存器的最低位对应 0 号管脚，最高位对应 31 号管脚。当 PIO_PER 寄存器某二

表 4.4: PIO 控制器寄存器说明

偏移地址	描述	名称	访问方式
0x0000	PIO 使能寄存器	PIO_PER	只写
0x0004	PIO 禁止寄存器	PIO_PDR	只写
0x0008	PIO 状态寄存器	PIO_PSR	只读
0x0010	输出使能寄存器	PIO_OER	只写
0x0014	输出禁止寄存器	PIO_ODR	只写
0x0018	输出状态寄存器	PIO_OSR	只读
0x0020	抖动过滤使能寄存器	PIO_IFER	只写
0x0024	抖动过滤禁止寄存器	PIO_IFDR	只写
0x0028	抖动过滤状态寄存器	PIO_IFSR	只读
0x0030	输出设置寄存器	PIO_SODR	只写
0x0034	输出清除寄存器	PIO_CODR	只写
0x0038	输出状态寄存器	PIO_ODSR	
0x003C	管脚数据状态寄存器	PIO_PDSR	只读
0x0040	中断使能寄存器	PIO_IER	只写
0x0044	中断禁止寄存器	PIO_IDR	只写
0x0048	中断屏蔽寄存器	PIO_IMR	只读
0x004C	中断状态寄存器	PIO_ISR	只读
0x0050	多驱动使能寄存器	PIO_MDER	只写
0x0054	多驱动禁止寄存器	PIO_MDDR	只写
0x0058	多驱动状态寄存器	PIO_MDSR	只读
0x0060	上拉电阻禁止寄存器	PIO_PUDR	只写
0x0064	上拉电阻使能寄存器	PIO_PUER	只写
0x0068	上拉电阻状态寄存器	PIO_PUSR	只读
0x0070	A 功能选择寄存器	PIO_ASR	只写
0x0074	B 功能选择寄存器	PIO_BSR	只写
0x0078	功能状态寄存器	PIO_ABSR	只读
0x00A0	输出写入使能寄存器	PIO_OWER	只写
0x00A4	输出写入禁止寄存器	PIO_OWDR	只写
0x00A8	输出写入状态寄存器	PIO_OWSR	只读

进制位被设置为 1 时，表示对应的管脚被选择为通用 IO 功能，设置为 0 时，对应管脚的设置状态不改变。PIO_PDR 寄存器在某二进制位被设置为 1 时，对应管脚被选择为外设功能，即不作为通用 IO 使用。PIO_PDR 寄存器设置为 0 的二进制位对应的管脚状态不改变。

并不是所有的 PIO 管脚都与集成外设复用管脚，而每个 PIO 管脚也最多与两个外设功能复用，分别被称为 A 功能与 B 功能。例如 PIO C 的 0 号管脚同时还可以作为 NAND Flash 的 OE 信号，也可以作为外部片选 NCS6 来使用。如果要使用某个管脚的 A 功能或者 B 功能，就要设置 PIO_ASR 或者 PIO_BSR 寄存器的相应二进制位为 1。同时要设置 PIO_PDR 寄存器的相应位。

当 PIO 管脚被设置为外设功能的时候，即相应的 PIO_PSR 位读出为 0，其输入和输出状态完全由相应外设来控制。当 PIO 管脚被设置为通用 IO 的时候，这个管脚可以被配置为输入或者输出，相应的设置寄存器为 PIO_OER 和 PIO_ODR，当 PIO_OER 设置为 1（这里不再强调特定二进制位与管脚的对应，但含义和前面的叙述相同）时，PIO 管脚被设置为输出，当 PIO_ODR 设置为 1 时，PIO 管脚被设置为输入。输入/输出状态可以从 PIO_OSR 寄存器读出，1 表示输出，0 表示输入。

当 PIO 管脚设置为输出的时候，可以用 PIO_SODR 和 PIO_CODR 寄存器设置输出的电平。当 PIO_SODR 寄存器设置为 1 的时候，PIO 管脚输出为 1；当 PIO_CODR 寄存器设置为 1 的时候，PIO 管脚输出为 0。PIO 管脚的输出电平可以通过 PIO_ODSR 寄存器获得。对 PIO 管脚电平的设置可以先于对其输入/输出的控制，即当设置好输出电平的时候，如果对应管脚为输入，则输出的电平设置并不反应到管脚上，只有当通过 PIO_OER 设置了该引脚输出时，其设置的电平才可以输出到管脚。

对一组 32 个管脚的电平设置需要写入 PIO_SODR 和 PIO_CODR 两个寄存器才能完成，这在某些情况会带来控制时序的问题。AT91SAM9261 允许写入 PIO_ODSR 寄存器来直接控制管脚输出的电平。不过这个操作还要受到 PIO_OWER 和 PIO_OWDR 寄存器的控制。其中 PIO_OWER 设置为 1 允许管脚通过写入 PIO_ODSR 寄存器来改变电平，PIO_OWDR 寄存器设置为 1 禁止管脚通过写入 PIO_ODSR 寄存器来改变电平。系统复位的时候，所有管脚都是禁止通过 PIO_ODSR 来设置电平的。

当 PIO 管脚被设置为输入的时候，可以通过 PIO_PDSR 来读取管脚的电平状态。实际上只要 PIO 控制器的时钟是被允许的，通过 PIO_PDSR 寄存器就可以获得管脚的电平，而不管这个管脚是被配置为输入还是输出，甚至配置为外设控制。

每个 PIO 管脚还可以被配置为中断源，当管脚的电平发生变化时，可以触发 PIO 控制器的中断。这个功能通过 PIO_IER、PIO_IDR 和 PIO_IMR 寄存器来进行设置。当用 PIO_IER 把管脚设置为中断源的时候，不管管脚是否被配置为输入或者外设功能，都可以用来触发中断。当 PIO 控制器在相邻时钟周期采集到不同的管脚电平的时候，就可以触发中断，相应的 PIO_ISR 寄存器位被设置，如果 PIO_IMR 位被设置，则 PIO 控制器的中断线被设置有效。需要注意的是如果当 PIO_ISR 寄存器被读出的时候，所有的 32 个中断标志都会被清除，所以程序必须能够处理多个中断同时触发的情况。

PIO 控制器还有一些寄存器用来控制管脚的特殊功能，它们的使用方法和前面介绍的寄存器非常类似。例如 PIO_PUER 和 PIO_PUDR 寄存器用来控制内部的上拉电阻是否有效。系统复位后，内部的上拉电阻都是有效的，即 PIO_PUSR 的值为 0。内部上拉可以确保管脚有确定的输出值，不会由于没有被驱动而处于悬空状态。PIO_MDER 和 PIO_MDDR 用来控制管脚是否采用漏极开路（OD 门）的方式输出，漏极开路的输出方式允许多个驱动源采用线与（wire-and）的方式连接，只有全部输出为高的时候，管脚状态才为高。PIO_IFER 和 PIO_IFDR 寄存器用来设置是否允许输入管脚的抖动过滤功能。典型的按键开关就是很好的例子，当开关接合与断开瞬间，开关的输出会在高低电平之间不断切换，被称为抖动现象，如果不进行处理就会造成开关控制的误操作。AT91SAM9261 的抖动过滤功能可以过滤掉小于半个时钟周期时间的抖动，同时只有超过一个时钟周期的脉冲电平才会被接受。

4.4.4 通用串行口

串行口的特点在于它的协议比较简单，最少只要三根线就可以实现全双工通信。而且串行通信的协议历史比较悠久，台式微机一般都在主板上包含串行通信口，许多智能设备也可以通过串口和其他设备或微机通信。因此串行口也叫通用串行口，它最大的优势就在其通用性。与目前非常流行的通用串行总线（USB）不同，串行口并不是总线结构，一般只应用在点对点的环境中，而且其硬件实现成本与软件控制逻辑都比较简单，在嵌入式领域还占有非常重要的位置。

AT91SAM9261 包含三个通用串行口，支持很多不同种类的异步与同步的串行通信协议，具体支持的通信模式如下：

- 5 ～ 9 位异步串行通信：支持最高位先发与最低位先发，支持 1、1.5、2 位的停止位，支持多种不同的奇偶校验模式，支持软件与硬件的握手协议，支持多址通信；
- 5 ～ 9 位高速同步通信：支持最高位先发与最低位先发，支持 1、2 位的停止位，支持多种奇偶校验模式，支持软件与硬件的握手协议，支持多址通信；
- 支持 RS485 控制信号；
- 支持智能卡的 ISO7816、T0、T1 协议；

- 红外 IrDA 模块的调制与解调；
- 支持自环、自动回复等测试模式。

AT91SAM9261 的调试逻辑也包含一个简化的串口，称作 “调试串口”，它只能工作在异步模式，并且只支持 8 位的数据通信，不过它的控制逻辑与这里将要介绍的通用串行口完全兼容（仅实现的功能）。接下来要介绍的寄存器与控制逻辑大多适用于全部 4 个串口，如需更详细的寄存器信息请参考数据手册。

基本的寄存器内存地址与简单描述如表 4.5 所示：

表 4.5: 异步串行口寄存器说明

偏移地址	描述	名称	读写模式
0x0000	控制寄存器	US_CR	只写
0x0004	模式寄存器	US_MR	读写
0x0008	中断允许寄存器	US_IER	只写
0x000C	中断禁止寄存器	US_IDR	只写
0x0010	中断屏蔽寄存器	US_IMR	只读
0x0014	状态寄存器	US_SR	只读
0x0018	接收数据保持寄存器	US_RHR	只读
0x001C	发送数据保持寄存器	US_THR	只写
0x0020	波特率发生寄存器	US_BRGR	读写

芯片被复位后，串口设备的接收模块是被禁止的，需要设置 US_CR 的 RXEN 控制位来允许接收。在串行通信过程中，接收到的数据保存在 US_RHR 寄存器中，同时 US_SR 寄存器的 RXRDY 标志被置位。当 US_RHR 寄存器的内容被读出的时候，RXRDY 位被自动清除。如果 US_RHR 寄存器没有被及时读出，新的串行数据又被接收，US_SR 寄存器的 OVRE 标志位被置位，旧的串行数据被新的数据所覆盖。程序可以通过设置 US_CR 的 RSTSTA 位来清除 OVRE 标志。RSTSTR 位还可以用来清除串口模块的其他错误状态，如接收数据的奇偶校验错（US_SR 寄存器的 PARE 标志位），或者帧结构错误（没有检测到合法停止位，US_SR 的 FRAME 标志位）。

芯片被复位后，串口设备的发送单元是被禁止的，需要设置 US_CR 的 TXEN 控制位来允许发送。当串口发送被允许后，US_SR 的 TXRDY 标志位有效，表示可以写入 US_THR 寄存器。当程序写入 US_THR 寄存器后，写入的内容就会移动到内部移位寄存器，并按照设置的波特率按位发送出去，此时 TXRDY 标志位依然有效，直到给 US_THR 写入第二个字节。当移位寄存器中的数据发送完毕后，US_THR 寄存器中的内容被移动过去继续发送，TXRDY 标志重新有效。当 US_THR 寄存器与内部移位寄存器同时为空的时候，US_SR 的 TXEMPTY 标志位被设置。

串口设备的发送和接收还可以通过 US_CR 寄存器的 TXDIS 位与 RXDIS 位来控制。当 TXDIS 设置为 1 的时候，如果串口设备正在发送字符，发送会在完成 US_THR 中的字符后被禁止。当 RXDIS 为被设置为 1 的时候，如果串口正在等待起始位，则接收立即被禁止，否则将等到当前字符被接收完毕才会禁止。如果软件设置了 US_CR 的 RSTTX 或者 RSTRX 位，串口设备的发送和接收立即被禁止，而不管当前串口的状态如何。

前面介绍的几个寄存器的详细信息如下：

US_CR 寄存器的部分内容如下，其中各控制位只有写入 1 时有效，写入 0 没有任何作用。

二进制位	8	7	6	5	4	3	2
名称	RSTSTA	TXDIS	TXEN	RXDIS	RXEN	RSTTX	RSTRX

US_MR 寄存器的部分内容如下：

二进制位	19	15-14	11-9	5-4
名称	OVER	CHMODE	PAR	USCLKS

CHMODE 用来设置串口的工作模式，包含两个二进制位。当设置为 “00” 时，代表一般工作模式；当设置为 “01” 时，代表自动回显模式；当设置为 “10” 时，代表本地自环模式；当设置为 “11” 时表示远端自环模式。最后三种都用于串口的检测。

PAR 用来设置串口的奇偶校验模式，包含三个二进制位。当设置为 “000” 时，代表偶校验；当设置为 “001” 时，代表奇校验；当设置为 “010” 时，代表校验位添 0 ；当设置为 “011” 时，代表校验位添 1 ；当设置为 “1xx” 时，代表没有奇偶校验。

US_SR 寄存器的部分标志位如下：

二进制位	9	7	6	5	1	0
名称	TXEMPTY	PARE	FRAME	OVRE	TXRDY	RXRDY

US_BRGR 寄存器的位设置如下：

二进制位	18-16	15-0
名称	FP	CD

US_BRGR 寄存器用来设置串口设备的波特率，对于调试串口来说，这个寄存器仅能设置为 0 ～ 65535 之间的数值，当设置为 0 时，波特率时钟被禁止，串口处于不工作状态。设置为其他值时，波特率的计算公式如下：

$$\text{波特率} = \frac{MCK}{CD \times 16}$$

其中 MCK 为芯片主时钟，CD 为 US_BRGR 中的设置值（clock division）。

对于其他三个 “完整” 串口设备，波特率的设置比较复杂，还与串口的工作模式相关。下面仅对异步通信模式下设置方法进行说明。

US_MR 寄存器中的 USCLKS 控制位可以用来选择串口设备的时钟源，当其设置为 “00” 时，表示串口时钟源选择为 MCK，当设置为 “01” 时，表示串口时钟源为 MCK/8，设置为 “11” 时，表示采用外部时钟，即同步模式。

波特率时钟主要基于一个 16 位的分频器，分频系数由 US_BRGR 寄存器的 CD（最低 16 位）设置，分频器输入由 US_MR 的 USCLKS 指定。当 CD 设置为 0 时，分频器与串口都不工作。具体的波特率还与 US_MR 的 OVER 控制位和 US_BRGR 的 FP 设置有关，可以用公式表达如下：

$$\text{波特率} = \frac{\text{所选时钟}}{(CD + FP/8) \times 8 \times (2 - OVER)}$$

其中 FP 可以取的值为 0~7，用来对分频系数进行微调，相当于分频系数的分数部分。由于 OVER 的值只能是 0 或者 1，所以实际的波特率为时钟频率除以 8 倍或者 16 倍的分频系数值。

4.4.5 SPI 总线

SPI（serial peripheral interface）总线是一种同步串行总线标准。SPI 最初由摩托罗拉公司提出，支持在主设备与从设备之间的全双工通信。数据传输由主设备发起，同一个主设备可以连接多个从设备，但每个从设备必须具有独立的片选信号。SPI 总线包含 2 条数据信号线和 2 条控制信号线：

1. MOSI（master out slave in）：由主设备传递到从设备的数据线。
2. MISO（master in slave out）：由从设备传递到主设备的数据线。同一时间不允许有多于一个从设备进行数据传输。
3. SPCK（serial clock）：由主设备驱动的时钟信号，用来控制数据线的传输速度，每个时钟周期传递一个数据位。
4. NSS（slave select）：用来选择从设备的控制线。

SPI 总线的优点主要有以下几个方面：

1. 全双工通信，较大的数据吞吐率。
2. 协议灵活，支持不同的字长和时钟特性。
3. 硬件接口简单，从设备不需要精确的时钟，不需要地址译码，不需要总线仲裁。

4. 信号单向传输，便于电流隔离。

SPI 总线的缺点主要有如下方面：

1. 相对其他串行协议，需要的管脚较多。
2. 没有流控信号，没有从设备应答机制。
3. 只支持一个主设备。
4. 数据传输的距离比较短。

支持 SPI 协议的器件很多，在嵌入式领域有很广泛的应用。典型的 SPI 总线连接模式如图 4.5 所示。

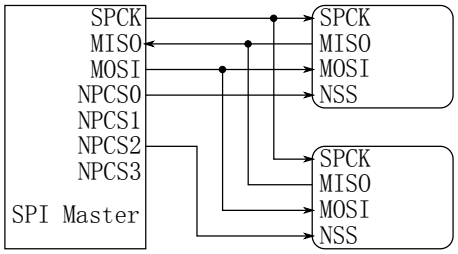

图 4.5: SPI 总线的连接模式

AT91SAM9261 的 SPI 接口可以工作在主模式或者从模式下。其相关寄存器的描述如表 4.6 所示。当模式寄存器(SPI_MR)的 MSTR 位设置为 1 的时候表示主模式，NPCS0 到 NPCS3 引脚被设置为输出，SPCK 与 MOSI 引脚被驱动，而 MISO 引脚被设置为输入。当 MSTR 位被设置为 0 的时候，相应的引脚都工作在从模式下，NPCS0 引脚作为 NSS 信号，而 NPCS1 到 NPCS3 引脚不被使用。具体的数据传输模式在主模式和从模式下是一致的，由共同的寄存器进行设置。

表 4.6: SPI 相关寄存器说明

偏移地址	描述	名称	访问方式
0x00	控制寄存器	SPI_CR	只写
0x04	模式寄存器	SPI_MR	读写
0x08	接收数据寄存器	SPI_RDR	只读
0x0C	发送数据寄存器	SPI_TDR	只写
0x10	状态寄存器	SPI_SR	只读
0x14	中断允许寄存器	SPI_IER	只写
0x18	中断禁止寄存器	SPI_IDR	只写
0x1C	中断屏蔽寄存器	SPI_IMR	只读
0x30	片选寄存器 0	SPI_CSR0	读写
0x34	片选寄存器 1	SPI_CSR1	读写
0x38	片选寄存器 2	SPI_CSR2	读写
0x3C	片选寄存器 3	SPI_CSR3	读写

为了适应不同的 SPI 接口，AT91SAM9261 的 SPI 协议配置非常灵活。时钟的极性可以由片选寄存器的 CPOL 位设置，相位可以由 NCPHA 位设置，这样可以获得四种不同的时钟配置。典型的数据传输时序如图 4.6 所示。

片选寄存器具体的定义如下：

二进制位	31-24	23-16	15-8	7-4	3	1	0
名称	DLYBCT	DLYBS	SCBR	BITS	CSAAT	NCPHA	CPOL

其中 CPOL 设置为 0 时，表示时钟信号无效状态为低电平，否则表示时钟信号无效状态为高电平。NCPHA 设置为 0 时数据在时钟变为有效的边沿发生变化，在另外一个边沿被输入端锁定。

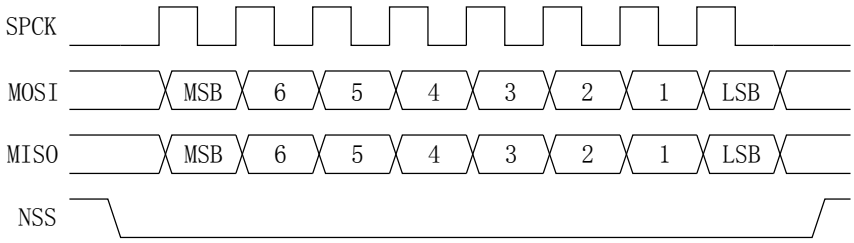

图 4.6: SPI 时序,CPOL=0,NCPHA=0

否则表示数据在时钟变为无效的边沿发生变化,在另外一个边沿被输入端锁定。

CSAAT 位设置为 0 时,片选信号在数据传输结束后立即无效;设置为 1 时,片选信号持续有效到另外一个不同的从设备被选择。

BITS 用来设置传输的数据位数,设置为 0~8 分别对应一次传输的数据为 8 位到 16 位。SCBR 用来设置 SPI 主模式下的波特率,具体的数值为 MCK/SCBR。DLYBS 用来设置从片选有效到 SPCK 变为有效的延时。当设置为 0 时,代表这个延时为 SPCK 时钟周期的一半,否则代表的延时为 DLYBS/MCK。DLYBCT 用来设置在片选持续有效的前提下,连续数据传输之间的延时,具体延时的算法为 32×DLYBCT/MCK。

SPI 控制寄存器的设置如下:

二进制位	24	7	1	0
名称	LASTXFER	SWRST	SPIDIS	SPIEN

其中 SPIEN 设置为 1 时表示 SPI 设备有效,SPIDIS 设置为 1 时表示 SPI 设备无效,SWRST 设置为 1 时执行软件复位功能,LASTXFER 设置为 1 表示当前传输为最后的数据,传输完毕即将片选设置为无效,需要与 CSAAT 位配合使用。这四个二进制位在设置为 0 的时候都没有效果。

SPI 模式寄存器的设置如下:

二进制位	31-24	19-16	7	4	2	1	0
名称	DLYBCS	PCS	LLB	MODFDIS	PCSDEC	PS	MSTR

其中 MSTR 设置为 1 代表 SPI 工作在主模式下,否则代表从模式。PCSDEC 设置为 0 时,代表用片选线直接连接从设备,这样最多可以连接 4 个从设备;当 PCSDEC 设置为 1 时,表示片选线通过 4-16 译码器连接从设备,这样最多可以连接 16 个从设备。具体的从设备选择由 PCS 位来决定,当 PCSDEC 设置为 0 时,NPCS0 到 NPCS3 的选择分别对应的 PCS 值为 “xxx0”,“xx01”,“x011”,“1111”。如果 PCSDEC 设置为 1,NPCS[3:0] 的输出与 PCS 的设置相同。

SPI 模块包含两个保持寄存器(TDR 与 RDR)和一个移位寄存器。当 SPI 工作在主模式的时候,首先要设置 PCS 位以选择特定的从设备,然后当处理器写入 SPI_TDR 寄存器的时候,一个数据传输就开始了,数据被立即写入到移位寄存器并在 SPI 总线上进行传输。当一个二进制位移出到 MOSI 数据线的时候,MISO 数据线上的数据被移入到移位寄存器,发送数据与接收数据将同时完成。移位寄存器的数据移出完毕的时候,其内容会立即更新到 RDR 寄存器。如果在数据传输过程中写入 TDR 寄存器,其内容会保持到本次传输结束,在接收数据保存到 RDR 寄存器后立即写入到移位寄存器开始下一次传输。

SPI 的数据传输需要和状态寄存器进行配合,状态寄存器的部分标志定义如下:

二进制	24	9	3	2	1	0
名称	SPIENS	TXEMPTY	OVERS	MODF	TDRE	RDRF

当 TDRE 位为 1 的时候,表示 TDR 寄存器为空,此时才可以写入 TDR 寄存器。而写入 TDR 寄存器之后,TDRE 位自动被清 0。当数据传输结束后,TXEMPTY 位被设置,表示移位寄存器为空,SPI 时钟可以被关闭。

接收数据写入到 RDR 寄存器后,RDRF 位会被设置,当数据被读出后,RDRF 被清 0。如果 RDR 寄存器在新数据到来之前没有被读出,OVERS 位会被设置,代表发生了数据溢出。

4.4.6 I²C 总线

I²C 总线标准是一种具有多主设备、多从设备支持的串行总线。I²C 标准最初由 Philips 公司提出，主要面向一些低速的接口设备，如模数/数模转换芯片和各类传感器芯片等。AT91SAM9261 芯片的 I²C 支持在数据手册中被称为 TWI（two wire interface），它对 I²C 标准的支持情况见表 4.7

表 4.7: Atmel TWI 与标准 I²C 的对比

标准 I²C	Atmel TWI
标准模式速率（100KHz）	支持
快速模式速率（400KHz）	支持
7/10 位从设备寻址	支持
起始字节	不支持
重复起始条件	支持
ACK 与 NACK 控制	支持
快速模式下的斜率控制与输入过滤	不支持
时钟延展	支持

I²C 的总线规定了两条信号，其中 TWD（I²C 标准中称为 SDA）为双向的数据信号，TWCK（I²C 标准中称为 SCL）为双向的时钟信号。这两个信号都被设计成漏极开路，因此可以同其他 I²C 设备进行线与。典型的 I²C 总线拓扑结构如图 4.7 所示。

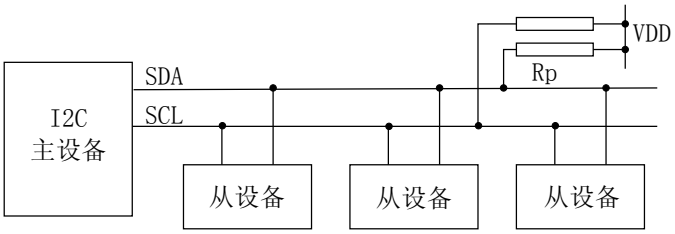

图 4.7: I²C 总线的连接

I²C 设备在进行数据传输的时候，数据信号只在时钟信号为低时变化，在时钟上升沿进行采样。而如果在时钟信号为高的时候数据信号发生变化，则代表两种特殊的控制信号：起始位和停止位。其中数据信号由高变低代表起始位，表示一次数据传输的开始；数据信号由低变高代表停止位，表示一次数据传输的结束。图 4.8 为 I²C 总线的时序图。

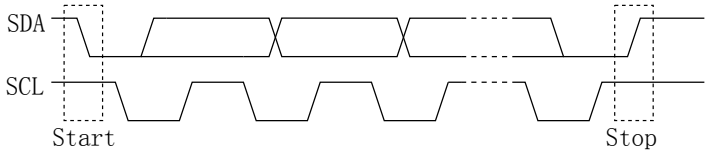

图 4.8: I²C 总线的时序

I²C 数据传输由主设备产生一个起始位开始，然后传递 7 个地址位（对于 7 位地址模式），指定通信的从设备。接下来传递的一位是读写位，0 表示写入，1 表示读出。再接下来的一位由从设备驱动，如果从设备的地址与主设备发出的地址相同，从设备将把 SDA 信号拉低，表示确认这次数据传输（ACK）。当从设备地址被确认，真正的数据传输就开始了。如果是写入数据，则数据线仍由主设备驱动，从设备在每个字节传输之后也要驱动一位的 ACK 信号。如果是读出数据，则数据线由从设备驱动，主设备仅在应答时输出 ACK 位。最后由主设备产生一个停止位表示数据传输结束。所有的 I²C 数据与地址，都是先传递最高位，最后传递最低位。

TWI 相关寄存器如表 4.8 所示。

其中 TWI_CR 的位定义如下：

表 4.8: TWI 相关寄存器说明

偏移地址	描述	名称	访问方式
0x00	控制寄存器	TWI_CR	只写
0x04	主设备寄存器	TWI_MMR	读写
0x08	从设备寄存器	TWI_SMR	读写
0x0C	内部地址寄存器	TWI_IADR	读写
0x10	时钟发生寄存器	TWI_CWGR	读写
0x20	状态寄存器	TWI_SR	只读
0x24	中断允许寄存器	TWI_IER	只写
0x28	中断寄存器	TWI_IDR	只写
0x2C	中断屏蔽寄存器	TWI_IMR	只读
0x30	接收保持寄存器	TWI_RHR	只读
0x34	发送保持寄存器	TWI_THR	只写

二进制位	7	6	5	4	3	2	1	0
名称	SWRST	QUICK	SVDIS	SVEN	MSDIS	MSEN	STOP	START

这些位仅写入 1 时有效，其中 START 表示输出起始位，STOP 表示输出停止位，MSEN 表示允许主设备模式，MSDIS 表示禁止主设备模式，SVEN 表示允许从设备模式，SVDIS 表示禁止从设备模式，QUICK 表示输出 SMBus 协议的快速命令，SWRST 表示进行软件复位。

TWI_MMR 的位定义如下：

二进制位	22-16	12	9-8
名称	DADR	MREAD	IADRSZ

其中 IADRSZ 表示内部地址的字节数，MREAD 为 0 表示写入操作，否则表示读出操作，DADR 用来记录从设备的地址。

TWI_CWGR 的位定义如下：

二进制位	18-16	15-8	7-0
名称	CKDIV	CHDIV	CLDIV

其中 CLDIV 是 SCL 为低的时间，公式为 $((CLDIV \times 2^{CKDIV}) + 4) \times T_{MCK}$。CHDIV 是 SCL 为高的时间，公式为 $((CHDIV \times 2^{CKDIV}) + 4) \times T_{MCK}$

TWI_SR 的部分二进制位定义如下：

二进制位	8	6	2	1
名称	NACK	OVER	TXRDY	RXRDY

其中 RXRDY 在接受到数据并且没有被读出的时候为 1。TXRDY 在很多情况会被置 1，有 MSEN 被设置的时候；TWI_THR 中的数据被发送到数据线的时候；从设备没有应答的时候。OVER 在接收到的数据被新的数据覆盖的时候置 1。NACK 在从设备没有应答的时候置 1。

TWI 模块具有六种工作模式：

1. 主设备发送；
2. 主设备接收；
3. 多主设备发送；
4. 多主设备接收；
5. 从设备发送；
6. 从设备接收。

这里仅简单介绍前两种方式。

主设备发送模式需要设置 MSEN 位有效，同时设置 DADR 和 CKDIV。当数据被写入 TWI_THR 之后，TWI 模块就会输出一个起始位及设置的从设备地址。读写位的输出根据

MREAD 决定，对于发送模式 MREAD 的值为 0。在 ACK 位采样时刻（对于 7 位地址是第 9 个时钟周期），主设备把数据线置高，并对数据线采样，如果应答有效，则 TWI_SR 的 TXRDY 位被置为有效，同时把发送保持寄存器中的内容发送到数据线上。TXRDY 有效将持续到 TWI_THR 被写入新的数值。如果要输出停止位，则必须在 TWI_CR 寄存器中设置 STOP 位。

如果设置了 TWI_CR 的 START 位，则 TWI 进入主设备接收模式。TWI 模块首先输出一个起始位及设置的从设备地址，读写位的输出和设置都应该是 1。如果输出地址被正确应答，数据线上的数据会被读入到 TWI_RHR 寄存器，同时 TWI_SR 的 RXRDY 位被设置。RXRDY 位会在读出 TWI_RHR 的时候被复位。如果要输出停止位，则必须在接收最后一个字节的数据之前设置 STOP 位。因此如果仅需要读入一个字节就必须在开始的时候同时设置 TWI_CR 的 START 与 STOP 位。

4.5 嵌入式开发板

本书所使用的嵌入式开发板的硬件指标如下：

- 处理器 AT91SAM9261，主频 198M。
- 64M SDRAM 加 64M NAND Flash 加 2M 串行 Flash。
- 2 个 USB 主设备，1 个 USB 从设备，2 个串行口
- I2S 声卡，具有麦克风、线性输入和立体声输出。
- 10/100M 自适应网络接口
- 5 寸 TFT 液晶模块，640×480 分辨率
- 触摸屏输入，ADS7843 触摸屏控制器
- 10 个用户可配置 LED 灯
- 2 个用户可配置按键
- 40 针用户扩展接口
- 用户可编程模块

在硬件结构上，电路板分为 CPU 板与用户扩展板两个部分，其中 CPU 板主要包括处理器、SDRAM 和 Flash 设备。用户扩展板通过两个 50 脚双排插针与 CPU 板连接，包括其余的外部设备。

4.5.1 可编程 CPLD 模块

用户扩展板上的复杂可编程逻辑器件（complex programmable logic device，CPLD）模块采用 Altera 公司的 MAX II 系列的 240T 实现，主要连接了 AT91SAM9261 的外部总线，可以实现扩展的板级寄存器，以实现特定的功能。当前提供的代码实现了四个 8 位寄存器，其中第一个寄存器用来控制板上的八个 LED 灯的显示，第二个寄存器用来控制 LCD 的几个特殊控制管脚，其余两个寄存器没有使用，可以作为内存使用。

示例的 VHDL 代码如下：

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity blr is
        port(
```

```
                addr : in std_logic_vector(2 downto 0); --address bus
                nrd : in std_logic;       --output enable, low for read
                nwr : in std_logic;       --low for write
                rst : in std_logic;
                ncs : in std_logic;
                data: inout std_logic_vector(7 downto 0); -- data bus
                led : out std_logic_vector(7 downto 0);
                lcd_l: out std_logic;
                lcd_u: out std_logic;
                lcd_m: out std_logic
                );
end blr;

architecture Behavioral of blr is

type ram is array(3 downto 0) of std_logic_vector(7 downto 0);
signal ram1  : ram;

begin
        -- 外设控制
        led <= not ram1(0);
        lcd_l <= ram1(1)(0);
        lcd_u <= ram1(1)(1);
        lcd_m <= ram1(1)(2);
        process(rst, nrd, nwr, addr,ncs)
        begin
                if rst = '0' then
                        data <= (others => 'Z');
                        ram1(0) <= "00000000";
                        ram1(1) <= "00000100";
                        ram1(2) <= "00000000";
                        ram1(3) <= "00000000";
                elsif((ncs = '0')and addr(2) = '1') then
                        if(nrd = '0') then
                                data <= ram1(conv_integer(addr(1 downto 0)));
                        elsif(nwr = '0') then
                                ram1(conv_integer(addr(1 downto 0))) <= data;
                                data <= (others => 'Z');
                        end if;
                end if;
        end process;
end Behavioral;
```

这段代码的主体实现了一个 RAM，其片选线 ncs 连接了 AT91SAM9261 的 NCS0，地址线连接关系如下：

```
add0:A2, add1:A3, add2:A4
```

因此这个 RAM 的物理地址分别为 0x1000 0010，0x1000 0014，0x1000 0018，0x1000 001C。数据的宽度为 8 位。

4.5.2 触摸屏控制器

在嵌入式设备中，触摸屏是取代鼠标的重要输入设备，按照触摸屏面板的种类，可以分为电阻式、电容式、音波式、红外线式以及近场感应式，等等。目前最常见的是电阻式与电容式，我们

的开发板选择的是电阻式触摸屏。

电阻式触摸屏的基本原理可以通过图 4.9 来表示

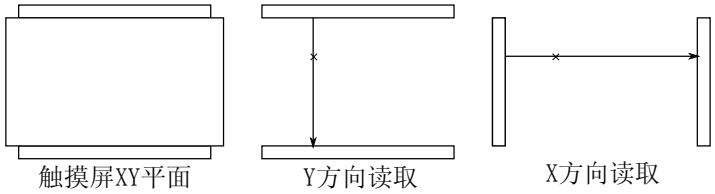

图 4.9: 电阻式触摸屏原理

首先电阻式触摸屏主要由两层透明的线性电阻平面组成(实际工艺更加复杂一些,这里仅从基本原理上进行解释),一个电阻层在 X 方向上每端有一个电极,称为 X 方向电阻层,另外一个电阻层在 Y 方向上每端有一个电极,称为 Y 方向电阻层。当触摸表面没有压力的时候,两个电阻层是绝缘的,而当在触摸表面上施加压力的时候,两个电阻层之间就会形成接触电阻。此时可以通过如下步骤获得接触点的坐标。

1. 在 Y 方向电阻层的电极两端输入直流电压
2. 由于两层电阻之间有接触电阻,在 X 方向电阻层电极处就可以读出稳定的直流电压。根据线性关系就可以得出 Y 方向的坐标。
3. 除去 Y 方向电阻层的电压,在 X 方向电阻层电极两段输入直流电压。
4. 同样在 Y 方向读出电压值,据此获得 X 方向的坐标。

由于这种触摸屏需要四根控制线,因此被称为四线制电阻式触摸屏。市场上有很多芯片支持这种触摸屏的控制,我们选用的芯片是 ADS7843。ADS7843 芯片的数据接口支持 SPI 串行协议,支持 8 位或者 12 位的电压分辨率,还包含两个附加的模拟输入。

ADS7843 共有 16 个管脚,各个管脚的简单描述见表 4.9。ADS7843 的硬件连接还是比较简单的,X+、Y+、X-、Y- 分别连接四线触摸屏的四根相应连线;DOUT、DIN、CS 与 DCLK 为芯片的 SPI 接口,可以直接与相应的 SPI 信号相连。

表 4.9: ADS7843 芯片的管脚描述

管脚号	管脚名称	描述
1	+Vcc	芯片电源
2	X+	X+ 输出,ADC 输入通道 1
3	Y+	Y+ 输出,ADC 输入通道 2
4	X-	X- 输出
5	Y-	Y- 输出
6	GND	芯片电源地
7	IN3	ADC 输入通道 3
8	IN4	ADC 输入通道 4
9	Vref	参考电压输入
10	+Vcc	芯片电源
11	PENIRQ	触摸屏输入中断
12	DOUT	串行数据输出
13	BUSY	芯片忙状态输出
14	DIN	串行数据输入
15	CS	片选信号
16	DCLK	外部时钟输入

ADS7843 支持多种数据读取模式,最快支持每 15 个时钟周期获取一次 ADC 数据。但为了更好地与 SPI 主设备接口,往往采用 24 个时钟周期的读取方式,这样可以在三次 8 位的 SPI

通信过程中获得 ADC 的数据。其中第一个字节为通过 SPI 接口输出的命令字，后两个输入的字节就是此次 ADC 转换的结果。命令字节的格式如下：

7	6	5	4	3	2	1	0
S	A2	A1	A0	MODE	SER/DFR	PD1	PD0

其中 S 为起始位，一定为高电平；A2-A0 指定不同的 ADC 采样模式；MODE 用来控制模数转换的位数，高电平为 8 位，低电平为 16 位；SER/DFR 位用来控制模数转换是否采用差分方式，高电平表示单端方式，低电平表示差分方式，这两种方式的差别如图 4.10 所示；PD1-PD0 用来设置芯片的省电模式。

A2-A0 设置为二进制 001 表示由 X+ 端输入，Y 方向上输出直流电压；A2-A0 设置为二进制 101 表示由 Y+ 端输入，X 方向上输出直流电压。单端方式与差分方式的区别是在单端模式下，ADC 的参考电压由 Vref 来确定，这样驱动触摸屏的模拟开关内阻会影响触摸屏的精度；差分模式下，ADC 的参考电压直接由驱动触摸屏的电压提供，使得触摸屏的输出电压相对于参考电压完全线性。

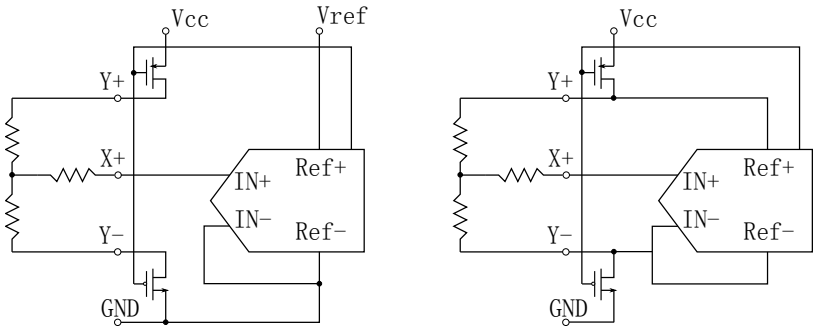

图 4.10: ADS7843 的单端方式（左）与差分方式（右）对比

PD1-PD0 设置为二进制 00 时，表示进入省电模式，并保持 PENIRQ 的中断功能；设置为二进制 01 时，表示进入省电模式，同时关闭 PENIRQ 的中断功能；设置为二进制 10[1] 表示不进入省电模式，并保持 PENIRQ 的中断功能；设置为二进制 11 时，表示不进入省电模式，同时关闭 PENIRQ 的中断功能。

4.5.3 TFT LCD 接口

LCD 显示屏是嵌入式设备常用的人机接口显示设备。常用的 LCD 设备分为 TFT 与 STN 两种类型。由于 TFT LCD 在很多方面上都优于 STN LCD，因此 STN 显示屏的应用范围主要局限于低成本的对显示效果要求不高的应用中。常见的 TFT LCD 显示屏包含表 4.10 所示的数字信号。

表 4.10: 典型 TFT LCD 包含的数字信号

信号名称	信号描述
Bn-B0	n+1 位蓝色分量数据输入
Gn-G0	n+1 位绿色分量数据输入
Rn-R0	n+1 位红色分量数据输入
DCLK	点时钟
DE	数据有效指示
VS	垂直扫描输入信号
HS	水平扫描输入信号
MODE	DE/HV 模式选择

[1] 部分芯片不支持这个模式

TFT LCD 有时也被称为主动方式 LCD，显示设备的点阵信息通过数字接口，不断的从 LCD 控制器传送到显示屏。数据传输过程是按照像素点输入的，分为水平扫描和垂直扫描两个过程。每个水平扫描过程输入显示屏一行的数据，每个垂直扫描输入显示屏一帧的数据。即垂直扫描包含多个水平扫描过程，用于按行填充整个显示屏的点阵数据。

TFT LCD 有两种常见的接口模式，一种被称为 DE 模式，另一种被称为 HV 模式。有些设备会同时支持两种模式，往往通过 MODE 信号来对模式进行选择。DE 模式的接口比较简单，其时序图如图 4.11 所示。

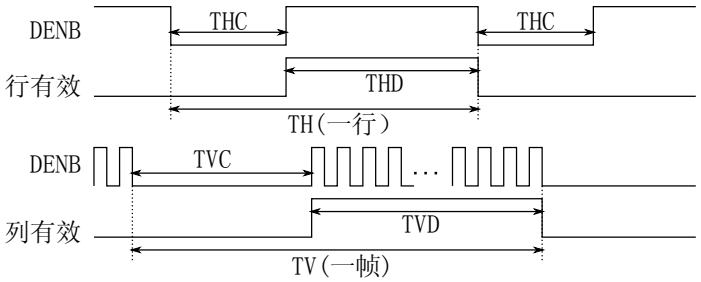

图 4.11: TFT LCD 的 DE 模式时序

在 DE 模式下，只有在 DE 信号有效（图中为高电平）时，RGB 数据线上的数据才是有效的，并在点时钟的有效沿进行采样。每次 DE 有效其间，LCD 控制器需要输出一整行的 RGB 数字信号。因此如果显示设备每行有 640 个显示点，则在图 4.11 中的 THD 就为 640 个点时钟周期。而由于 LCD 设备对输入数据需要时间进行处理，因此在每行数据输入之后还需要等待 THC 个点时钟周期。等待时间内 DE 信号为无效。这个过程就是 LCD 显示器的水平扫描过程。

在 DE 模式的垂直扫描过程中，如果显示设备有 480 个显示行，则经过 480（TVD）个水平扫描过程就可以完成整个 LCD 设备的扫描输入。同样由于 LCD 的数据处理时间，每帧数据输入之后还需要等待 TVC 个水平扫描周期才可以进行下一次垂直扫描过程。

TFT LCD 的 HV 模式时序图如图 4.12 所示。主要的握手信号是 HS 与 VS，分别为水平扫描同步和垂直扫描同步。HS 信号持续有效 Thwh 个点时钟周期表示水平扫描过程开始，但实际的点数据还要经过 Thbp 个点时钟周期才真正送到数据线上。点数据经过 THD 个点时钟周期后，还必须等待 Thfp 个点时钟周期才可以开始下一次水平扫描。

与 HV 模式的水平扫描方式类似，垂直扫描以 VS 信号持续有效 Tvwh 个水平扫描周期开始，然后再等待 Tvbp 个水平扫描周期开始输入第一行的水平扫描过程。经过 TVD 个有效数据的水平扫描过程之后，仍然要等待 Tvfp 个水平扫描过程时间才可以开始下一次垂直扫描。

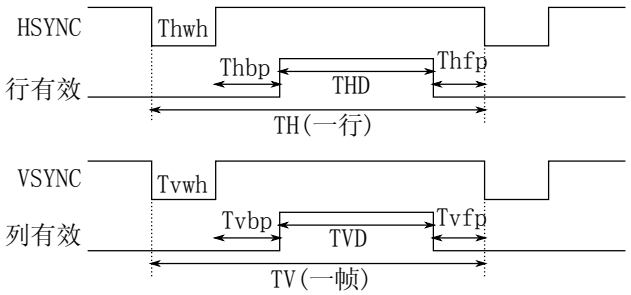

图 4.12: TFT LCD 的 HV 模式时序

除了前面介绍的数字信号，部分 LCD 设备还支持如屏幕翻转等其他数字信号。如 L/R 信号用来控制屏幕显示的左右翻转，当该信号为有效的时候，显示效果与该信号无效的时候成左右镜面对称。

LCD 设备的模拟输入有设备的电源与地的输入、LCD 背光电源的输入等。常见 LCD 背光有冷阴极荧光灯（cold cathode fluorescent lamp，CCFL）与 LED 背光两种。其中 CCFL 背光管应用比较广泛，但需要用逆变器获得灯管所需的高压，比较费电而且有触电危险。LED 背光是比较新的背光技术，不需要高压，逐渐被更多的 LCD 设备使用。

LCD 设备的背光亮度经常由 ADJ 信号来控制,这个信号可以是模拟信号,也可以由 PWM 的数字信号来驱动。通过改变 PWM 信号输入方波的占空比,可以调整数字信号输出的有效值,达到控制背光亮度的效果。

4.6 实验 :控制片上外设

实验目的

1. 加深对 AT91SAM9261 处理器及其开发板的了解
2. 进一步熟悉 GNU 开发工具和 Linux 操作系统

实验内容

1. 编写 LED 驱动程序,控制 DS1 和 DS2 闪烁

与第三章的实验不同,这里控制的 DS1 和 DS2 分别连接在 AT91SAM9261 的 PA24 和 PA26 管脚,编写程序通过对 PIO 进行设置,控制这两个 LED 灯按照固定的频率进行闪烁。

2. 编写串口通信的简单测试程序,实现通过 minicom 和微机进行简单通信

开发板的调试串口已经通过串口线连接到微机的串口,可以通过对 UART 相应寄存器的设置,实现:

- 输出提示信息
- 将串口接受到的数据加 1,再通过串口发送回来

由于 U-Boot 在硬件启动后已经对开发板的调试串口进行了设置,我们可以利用已经设置好的环境,仅使用 US_SR、US_THR 和 US_RHR 寄存器完成实验内容。实验的效果是当用户输入 “a” 字母时,minicom 中的输出为 “b” 字母,余此类推。

3. 编写按钮中断处理程序

开发板的按钮 BP1 和 BP2 分别通过 PA30 和 PA31 连接到处理器,通过 PIO 的设置,实现按钮触发中断,在中断处理程序中把一个 8 位数据显示到 8 个 LED 上,按 BP1 时对数据执行加 1 运算,当按 BP2 时执行减 1 运算。

第五章　嵌入式 Linux 基本原理

5.1　操作系统概述

现代的计算机系统是非常复杂的数字设备，一般包含一个或多个处理器、相当大容量的主存、时钟芯片、磁盘等存储设备、网络接口、终端及其他输入输出设备。为了正确的使用它们，需要编写一个程序来管理所有这些部件，并为用户提供一个接口或者虚拟机，从而更容易理解和进行程序设计。处于这一层次的软件就是操作系统。操作系统的存在使得计算机复杂的硬件特性被隐藏起来，为用户提供了一台更容易编程的等价的 “扩展机器”。

操作系统一般需要实现以下功能：

- 有效地组织和管理计算机系统中的硬件和软件资源
- 合理组织计算机工作流程，控制程序执行
- 向用户提供各种服务，使用户可以灵活、方便、有效地使用计算机，使整个计算机系统可以高效地运行。

5.1.1　操作系统的分类

根据操作系统功能特征的不同，可将其分为批处理系统、分时系统和实时系统三种基本类型。

1. 批处理操作系统

批处理操作系统的工作方式是：首先收集全部作业，然后由操作员将这些作业组成一批作业输入计算机，并由一个特殊的程序依次执行每个作业，最后由操作员将作业结果返回给用户。批处理操作系统分为单道批处理和多道批处理两种类型，前者在内存中只能驻留当前正在运行的作业，后者允许内存中同时驻留多道作业，不同作业之间交替执行。

由于用户自己不能干预作业的运行，发现错误也就不能及时改正，所以这种操作系统只适用于成熟的程序。批处理操作系统的优点是：作业流程自动化，效率高，吞吐量高；缺点是：无交互手段，调试程序困难。

2. 分时操作系统

分时操作系统的工作方式是：一台主机连接多个用户终端，允许多个用户以分时方式共享一台主机。分时操作系统将 CPU 的时间分为多个时间片，以这个时间片为单位，轮流为每个终端用户服务。

分时系统具有多路性、交互性、独占性和及时性的特征。多路性是分时系统可以多个用户同时使用的特性，交互性是指用户可以和程序交互、干预程序的运行，独占性是指对每个用户都感觉到好像是全部 CPU 时间都为自己服务（实际是虚假的独占），及时性是指系统对用户的请求可以及时地响应。

3. 实时操作系统

实时操作系统是指计算机能够及时响应外部事件的请求，在规定的时间内可以完成对该事件的处理，并控制所有的实时设备和实时任务协调一致地工作的操作系统。相对于分时操作系统，实时操作系统要求有更高的可靠性和完整性。按照对实时处理的后果不同，实时操作系统可以分成软实时系统和硬实时系统。对于软实时系统，未能及时完成处理只会带来额外的代价，而

且这种代价是可以被接受的;而硬实时系统在未能及时完成处理的情况下,往往会造成灾难性的后果。

根据操作系统在使用环境上的不同,还可以分为嵌入式操作系统、个人计算机操作系统、网络操作系统和分布式操作系统。本章主要讨论的是嵌入式操作系统。

5.1.2 嵌入式操作系统的特点

嵌入式系统本身的特点使得嵌入式操作系统和它们所基于的硬件系统紧密相关。一般来说,典型的嵌入式操作系统具备以下几个特点

1. 资源受限,成本低

嵌入式系统一般属于专用设备,硬件上有规模小、重量轻、成本低等要求,这就使得操作系统的运行环境受到了限制。因此大部分嵌入式操作系统都具有代码体积小、运行效率高、资源需求少、系统灵活可配制等特点。同时,简洁的代码也为把操作系统移植到新的硬件平台提供了方便,从一个方面降低了开发成本。

2. 具有处理异步并发事件的能力

在实际环境中,嵌入式系统大多数是事件驱动系统,而且处理的外部事件是多发而且并发的随机事件。嵌入式操作系统应该能有效地处理这些并发事件,所以嵌入式操作系统一般都具有多进程、多任务运行机制,还可以采用线程和轻量线程等技术获得更快的资源和任务切换速度。

3. 响应速度快,需要实时处理能力

一般来说,嵌入式系统要求对外部事件的反应是快速的、确定的、可重复实现或者周期性的,不管系统当前的内部状态如何都是可以预测的,即使在尖峰负荷下,系统的响应也应该在严格的时间内完成。这实际上是对实时操作系统的要求,所以很多情况下,嵌入式操作系统和实时操作系统是等价的。

4. 可靠性要求高

嵌入式系统有时候会有很高的可靠性要求。例如汽车的 ABS 系统,软件的任何一次失效都可能造成生命的威胁。

5. 快速启动、出错处理和自动复位功能

即使系统有很高的可靠性,也有可能由于不可预测的环境等因素而发生错误,所以是否具有故障检测和修复功能也成为嵌入式操作系统的一种要求。在最坏的情况下,系统应该可以自动复位来恢复系统的运行。此时如果操作系统可以快速地启动,就可以保证嵌入式系统很快地恢复功能。

6. 采用交叉环境开发

嵌入式操作系统一般运行在专用的 CPU 上,由于硬件资源的限制,嵌入式系统软件的开发几乎不可能在目标设备上进行。它需要在专门的开发平台进行交叉开发,开发环境和运行环境并不相同,开发平台叫宿主系统,嵌入式系统的运行系统叫目标系统。

5.1.3 操作系统的基本概念

1. 进程

进程是操作系统中最重要的一个概念,操作系统的其他内容都是围绕进程展开的。一个进程就是一个正在执行的程序,包括它的程序计数器(PC)、寄存器、堆栈和变量的当前值。进程有时候也被称为任务,现代的操作系统大都允许多个任务同时被处理,任何一个任务都可以被认为完全占有 CPU,而操作系统就要负责管理 CPU 在各个任务之间快速切换,轮番运行这些任务中的某一个。

2. 调度

任务管理的基本功能就是任务调度,即实现由某个事件或者某个时间驱动的任务切换。在多任务系统中,每个任务都会有一个与之关联的优先级,代表这个任务的重要程度。在进行任务调度时,根据任务优先级是否可以动态改变,调度算法可以分为静态调度和动态调度两种。根据调度过程是否采用抢占方式,调度算法可以分成抢占方式和非抢占方式。抢占方式指的是当前运行的任务可以被高优先级的任务剥夺运行权利,一般实时操作系统都采用抢占方式,因为这样

可以较好地保证高优先级任务的时限要求。

3. 系统调用

操作系统要提供给用户程序一个统一的硬件接口，用户程序通过这个接口对系统硬件进行访问，这里的“接口”就是系统调用。提供标准的系统调用不仅可以简化用户程序的操作，还提高了用户程序的移植性。

4. 可重入性

在多任务环境下，多个用户进程可能通过系统调用同时执行一个系统函数，这时就必须保证和不同进程相关联的系统数据不会被破坏，这种特性就叫做可重入性。可重入的函数只使用局部变量，即变量保存在 CPU 寄存器或堆栈中。如果使用全局变量，就必须对全局变量进行保护。

5. 代码临界区

代码临界区指处理时不可分割的代码，一旦这部分代码开始执行，则不允许任何中断打入，因此这段代码在某一时刻最多只有一个进程可以访问。使用临界区是在多任务环境下处理不可重入代码的一种有效手段。

6. 上下文切换

进程的上下文是对进程执行全过程的静态描述，如 CPU 的工作状态、各寄存器的值、进程的当前状态、与该进程相关的核心数据等。当发生任务切换的时候，就必须保存原运行进程的上下文，恢复新运行进程的上下文，这个过程叫做上下文切换。一个比较特殊的上下文切换是操作系统本身和用户程序之间的上下文切换，这时候系统将在用户空间和内核空间进行切换。

7. 内核

内核是操作系统的内部核心程序，它通过系统调用向外部提供对计算机设备的核心管理功能。现代的 CPU 大都支持不同的运行级别，内核程序一般具有最高的执行权限，而用户程序的权限则比较低，这样可以保证内核数据不会受到破坏。通常一个程序会通过系统调用执行内核代码，这时它的运行状态被称为核心态，而其他时候的状态则称为用户态。

内核的设计分为两种流派，一种是宏内核设计，操作系统所有的系统相关功能全部被封装到一个内核程序中。另外一种是微内核设计，内核只包含最基本、最核心的功能（如任务、内存管理），而其他管理功能（文件系统、网络等）则作为外部独立程序。微内核的设计具有很好的灵活性，系统结构比较清晰，便于维护，但执行效率不如宏内核设计。

5.1.4 嵌入式操作系统的其他关注点

设计目标：通用操作系统的设计目标是追求最大的吞吐率，使整体性能最佳；而实时操作系统设计的目标是采用各种算法和策略，始终保证系统行为的可预测性。

调度原则：通用操作系统为了达到最佳整体性能，其调度原则是公平；而实时操作系统多采用基于优先级的可剥夺的调度策略。

内存管理：通用操作系统广泛使用了虚拟内存的技术，为用户提供一个功能强大的虚拟机，但因虚存机制引起的缺页调页现象会给系统带来不确定性，因此实时系统很少或有限地使用虚存技术。

系统响应时间 (system response time)：是指系统发出处理要求到系统给出应答信号的时间。

任务切换时间 (context-switching time)：是指任务之间切换使用的时间。

中断延迟 (interrupt latency)：是指计算机接收到中断信号到操作系统做出响应，并完成切换转入中断服务程序的时间。

5.2 操作系统基本功能模块

5.2.1 内存管理

内存是计算机系统非常重要的资源，特别是在嵌入式系统这样资源有限的系统中，内存资源更加显得珍贵，操作系统必须认真地对内存进行管理，以高效地利用它们。内存管理程序需要记

录哪些内存在使用，哪些内存是空闲的，在进程需要的时候为进程分配内存空间，在进程使用内存完毕后要释放这些空间，在内存无法装入所有的进程时，内存管理程序还要负责在磁盘和内存之间做交换。

一、分区式存储管理

最简单的内存管理方式是单一的连续内存分配。在这种方式中，内存被分为系统区域和用户区域，应用程序被加载到用户区，可以使用用户区的全部空间。这种方式实现简单，管理方便，但只适用于单任务系统，对内存资源的浪费比较严重。

对于多任务系统来说，最简单的方法是把全部内存直接划分为 N 个区域（可能不是均等的），操作系统本身占用其中一个分区，当一个进程开始运行的时候采用一定的算法将它分配到一个空闲的内存块中，当没有空闲分区可用时，进程被挂起。一般来说，这种固定分区的方式会划分多个不同尺寸的内存空间，根据程序的大小，分配当前空闲的、适当大小的分区。

分区式存储管理引入了内存碎片的问题。其中占用分区内未被利用的空间称为内碎片，占用分区之间难以利用的小空闲分区称为外碎片。这些碎片在一定程度上造成了内存空间的浪费，操作系统应该采用必要的算法减少内存碎片的产生。

固定分区的优点是容易实现、系统开销小，缺点是分区数固定，限制了并发程序数目，内碎片比较普遍。与固定分区相对应的技术是动态分区，它的特点是在装入程序时按照初始值分配内存，在执行过程中按需要分配或改变分区的大小。这种方法虽然可以避免内碎片的产生，但经过多次内存分配之后，外碎片的现象比较严重，很可能造成系统有很多的剩余内存，但是由于分散在不同的分区之间，而不能被集中利用，导致不能加载需求内存比最大空闲块大的程序。

二、分页式管理

前面介绍的内存管理方式为进程分配的空间是连续的，使用的地址也都是物理地址，现在大部分 CPU 都提供逻辑地址的支持，把进程的地址空间和实际的内存空间分离，增加内存管理的灵活性。

分页式管理将程序的逻辑地址空间划分为固定大小的页，而物理内存被划分为同样大小的页框。当程序加载时，可将任意页放置到内存的任意一个页框内，这些页框不必连续，因此进程可以按照页面的大小分散在物理内存的不同位置。这种管理方式的优点是没有外碎片，内碎片的大小不超过一个页面。而且由于程序不必连续存放，这样就便于改变程序占用空间的大小，适应程序动态生成数据对地址空间需求的增加。

一个简单的分页管理方式可以把程序使用的地址分为两个部分，前一部分是页号，后一部分是页内地址。所有的页号在内存中体现为一个表格，用来表示特定的页号对应于实际内存中的哪一个页框。当程序对某个逻辑地址进行访问时，处理从某个特定的寄存器得到页表的起始地址，然后根据页号作为索引，找到相应的页框起始位置，这个起始地址再加上页内地址作为偏移量就得到了实际的物理地址。这个过程通常是需要硬件支持的，这部分硬件被称为 MMU，可以参考第三章的介绍。

目前大部分的嵌入式系统采用的处理器都具有 32 位的寻址能力，如果分页的大小是 4K 字节，32 位的地址空间将有 1M 个页，如果是 64 位的系统，页表几乎不可能被保存在内存中。对于多任务系统，每个进程都要有自己的地址空间，拥有自己的页表，这样页表的数目就更加庞大了。为了避免在内存中保存庞大的页表，大部分处理器都采用了多级页表，32 位的逻辑地址可以被划分成三个部分，前面 10 位是第一级页表的索引，中间 10 位是二级页表索引，后面 12 位是内存的偏移。这样每一级页表都有 1024 个条目，一般每个条目为四个字节。这样，每个页表所占用的空间正好与内存页框相同，都是 4K 个字节。

当采用多级页表的时候，不必把全部的页表都保存在内存中，对于图 5.1 的分页方式，如果一个进程只需要 8MB 的空间，它就可以只填写三个页表。因为每个二级页表可以代表 1024 个内存页框，相当于 4MB 的存储空间，两个二级页表就可以表示全部的 8MB 空间，再加上一级页表，就只需要三个页表就可以了。

实际的系统中，有采用二级和三级页表的实现方式，虽然级别越多，系统的灵活性越大，但也带来了更多的复杂性，在 32 位的系统中，二级页表已经完全可以满足实际的需要了。Linux 操

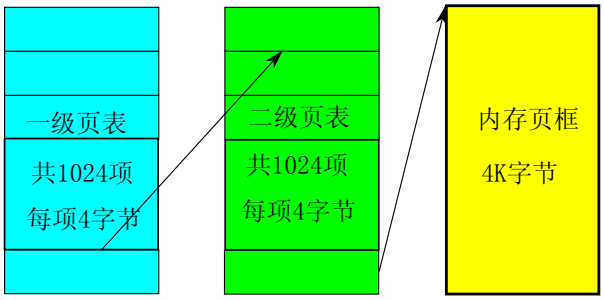

图 5.1: 二级页表示意图

作系统为了兼容 64 位的系统，使用了三级页表，但实际应用到 32 位系统的时候，多余的一级页表就被编译器优化掉了。

虽然分页机制和硬件紧密相关，但具体页表项所包含的信息却大致相同。图 5.2 给出了一个 ARM 处理器的页表项。

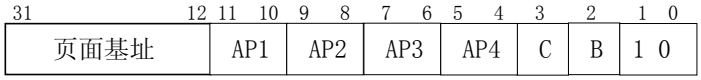

图 5.2: ARM 处理器“小页”的页表项

不同处理器的页表项大小可能不一样，但 32 位是一个比较普遍的尺寸。页表项中最重要的是页框号，即页面的基址。毕竟页映射就是找到这个值，对于上面的分页方式，页框号应该是一个 20 位的高位地址，这个地址加上 12 位的偏移地址就是实际的物理地址。ARM 处理器的页表项相对比较简单，其中 AP1 到 AP4 定义了四个子页的访问权限，“C” 位定义了是否可以缓存（cachable），虽然高速缓存对于常规内存来说可以提高访问速度，但对于内存映射的 I/O 就会有副作用。因为在这种情况下，假如操作系统在循环地等待某个 I/O 设备对它发出命令的响应，保证硬件是不断地从设备中读取信息而不是在读取一个旧的被缓存的副本是非常重要的。“B” 位定义了是否可以缓冲（bufferable），与 “C” 位的情况类似，主要用于写入内存的时候避免由于缓冲而延缓写入实际地址所造成的副作用。更典型的页表项还包含其他一些重要的位。

“在/不在” 只占一个二进制位，表示当前的页面是否存在于内存中，如果这个位是 1 则代表该表项的页框号是有效的，否则表示该页面不在内存中，将引发一个缺页中断。操作系统可以在缺页中断的处理程序中把要求的页面从外存调入到内存，然后重新执行刚才的内存访问操作。这就是虚拟内存技术实现的基础，由于页表对应的页面不一定存在于内存中，页面也被称做 “虚页”。如果系统不支持虚拟内存，有效的页面将全部在内存中，“在/不在” 位将永远是 1。

保护位可以被操作系统利用来管理内存的访问权限。最简单的情况是这个位只有一位，0 表示读写，1 表示只读。更复杂一些的方式是采用三位，分别指出是否允许读、写和执行。

修改和访问位用来记录页的使用情况。在写入一页的时候由硬件自动设置修改位，该位在操作系统重新分配页框的时候很有用。如果一页已经被修改过，则在需要保存该页的时候必须写回磁盘，否则只要简单地丢弃就可以了，因为它在磁盘上的副本仍然有效。不论是读还是写，在该页被访问的时候都设置访问位，它的值用来帮助操作系统在发生缺页的时候选择被淘汰的页。不再使用的页要比正在使用的页更适合被淘汰。这个位在很多页面置换算法中都起到重要的作用。由于在嵌入式系统中一般不使用虚拟内存，有关页面的置换算法请参考其他书籍。

三、分段的内存管理

为了允许程序和数据能被划分为逻辑独立的地址空间，以帮助实现共享和保护，一些处理器引入了分段的概念。如著名的 x86 系列。在段式存储器管理中，程序的地址空间被划分为若干个段，这样每个进程有一个二维的地址空间，程序通过分段可以被划分为多个模块，如代码段、数据段、共享段等。每个段都可以是 0 到允许的最大值之间的任何一个，不同段的长度也可以不同，而且通常也都不同。段的长度在运行期间还可以改变，例如堆栈段在数据被压入的时候会增加、数据被弹出的时候会减少。在分段方式下的寻址必须提供一个段号和一个段内的偏移地址。

与分页方式不同，段作为一个逻辑实体可以被程序员利用，页对程序员则是透明的。一个段可以包括一个过程、一个数组、一个堆栈，或者一些数值变量，但一般它不会同时包含多种不同的内容，因此不同的段可以采取不同的保护方式，这样的保护有助于找到程序的错误。

由于一个段可以包含很大的地址空间，管理起来可能会不很方便，因此部分处理器支持在段内分页的方法，就是在每个段中采用前面介绍的分页机制。

四、内存分配和回收

为了对内存进程管理，操作系统必须跟踪内存的使用情况，知道哪些内存被使用，哪些内存是空闲的。当为进程分配内存的时候要用一定的策略寻找空闲内存，当进程释放内存的时候，还要尽量高效地把释放的内存回收为空闲内存。在一般的操作系统中，常见的三种内存管理方式是：位图、链表、“伙伴（buddy）”系统。

1. 位图方式

使用位图记录内存的使用情况时，内存被划分为小到几个字、大到几千字节的分配单元，每个单元对应于位图的一位，用 0 表示相应的内存单元空闲，否则表示被占用。内存和分配单元的大小决定了位图的大小。在内存大小固定的前提下，分配单元越小，需要的位图越大。因为位图也要保存在内存中，为了节省内存，分配单元就不能太小，但过大的分配单元在内存分配时就可能发生比较多的浪费。另外一个值得注意的问题是当要分配 N 个连续的分配单元时，内存管理程序必须搜索位图，找到连续的 N 个 0，这个操作是比较耗时的。

采用位图方式分配和回收内存都比较简单，只要改变位图相应位的空闲标志就可以了。

2. 链表方式

另一种跟踪内存的方式是维护一个记录已分配和空闲内存段的链表。其中一个内存段可以是一个进程，或者是两个进程之间的空闲区。每个链表项至少包含起始地址、段长度、空闲/占用标志和指向下一表项的指针。

内存段在链表中的排列顺序通常是从内存低端到高端，这样当占用的内存空间被释放的时候，链表的更新比较方便。在把这断内存标记为空闲之后，可以根据它的前后两个邻居是否是空闲内存来合并相临空闲块，以满足更大内存块的申请。

在链表中分配内存有下面几种常用的方法：

首先适配法：按照内存段链表搜索，直到找到一个足够的空闲区进行分配。如果空闲区和要分配的大小不一致，就将该空闲区分为两部分，一部分被分配，另一部分仍然空闲。这种算法的分配和释放速度都比较快，但在内存低端会产生较多的小内存碎块。

下次适配法：和首先适配法比较类似，只是每次搜索空闲块都是从上一次找到的空闲块位置开始。这种方法可以使空闲块的分布比较均匀，搜索速度也比较快，但不易保留大的空闲块。

最佳适配法：这种方法搜索整个链表，找出恰好够用的最小空闲区进行分配。虽然这种方法可以使得外碎片比较小，但由于小碎片的难以利用，会造成碎片较多。最佳适配的优点是较大的空闲区可以被保留。

最差适配发：和上面的方法类似，不过是找出最大的空闲区进行分配。这样基本不留下小的空闲区，不易形成碎片。但这种方法使得系统缺乏大的空闲区，当有对内存需求较大的进程要运行时，其要求可能不被满足。

3. 伙伴方式

在采用链表方式管理内存的时候，为了加快空闲块的搜索速度，可以按照大小排序空闲块作为一个单独的链表。这样当按照最佳适配法分配内存的时候，从头按顺序找到的第一个合适大小的空闲块就是最小的，速度和首先适配法一致。但这样做的缺点是合并相临空闲块的开销太大，而如果不进行合并，内存将很快分裂出大量的进程无法使用的小空闲区。

伙伴系统是一种比较常用的内存管理机制，它利用计算机二进制寻址的特点，加速合并相临空闲块的速度。伙伴系统只分配 2 的整数次幂的内存块，所以它会包含大小为 1、2、4、8 直到内存总容量大小的内存块链表。例如对容量为 1MB 的内存，就有从 1 字节到 1MB 字节的 21 个链表。初始时只有 1MB 的链表有一个 1MB 的空闲区，其他的链表为空。当进程申请一个 80KB 的内存时，最小的符合 2 整数次幂的内存块为 128KB，但目前没有这样的空闲块，因此 1MB 的内存分割成两个 512KB 块（buddy），前面的 512KB 继续被分割成两个 256KB，前面的 256KB 再被分成两个 128KB，最后前面的 128KB 被分配给了申请的进程。

内存的申请总是在适合大小的内存块链表中搜索，如果找不到空闲块就用更大的内存块分割得到。内存释放的时候也只需要在一个链表中进行搜索连续空闲块，如果合并出了更大的空闲块才到“上一级”链表中继续进行合并操作。从这一点就可以看出伙伴系统的内存分配和释放速度很快，但伙伴系统的缺点也很明显，就是内碎片可能会比较严重，例如上面申请 80KB 空间的例子，其中有 48KB 的空间被浪费了。

五、虚拟存储器

现代的操作系统大部分都提供虚拟存储器的支持，它们在装入程序的时候，不必将程序全部读入内存，而只需将当前需要执行的部分页或段读入内存，就可以让程序开始执行。在程序执行过程中，如果需要执行的指令或要访问的数据尚未加载到内存，则发生缺页中断，操作系统再将相应的页或段调入内存，然后继续执行程序。另一方面，操作系统可以将内存中暂时不用的页或段调出，保存在外存上，从而腾出空间作为其他用途。于是从用户角度看，该系统拥有的内存容量将比实际容量大很多，因此称为虚拟存储器。

虚拟存储技术的引入，提供给用户一个大于实际物理内存的虚拟存储空间。这使得在较小的可用内存中执行较大的用户程序成为可能，并且可以在内存中容纳更多的程序并发执行。其最大的优点是对应用程序完全透明，应用程序可以假定自己全部保存在内存中。

虽然虚拟存储器技术的使用有很多优点，但在对实时性要求比较高的嵌入式操作系统中却不能使用。一方面是硬件的限制，对于不包含 MMU 的处理器，实现虚拟存储器比较困难，增大系统开销，得不偿失。另一方面是由于缺页中断造成的外存读写对系统的实时性会有负面的影响。因为缺页中断往往无法预测，外存读写的速度远小于内存访问速度，系统的实时性就会下降。

5.2.2 进程管理

一、进程的创建和取消

虽然进程被定义成一个正在执行的程序，但进程和程序是两个密切相关的不同概念，它们在以下几个方面存在区别和联系：

进程是动态的，状态可以不断变化；程序是静态的，能长久地保存。程序是有序代码的集合；进程是程序的执行。进程通常不可以在计算机之间迁移；程序通常对应着文件，可以在计算机间复制。进程的组成包括程序，还包含数据和进程状态信息。通过多次执行，一个程序可以对应多个进程；通过调用关系，一个进程可以包括多个程序。进程可以创建其他进程，而程序并不能形成新的程序。

一些简单的操作系统可能会在系统启动的时候把所有需要的进程都建立好，这样在系统运行的时候就不用考虑进程的创建了。在 Linux 系统中，除了第一个进程是系统创建之外，其他进程是由 fork 系统调用建立的，它将创建一个与调用进程相同的副本。调用 fork 的进程被称为父进程，新生成的进程叫做子进程。父进程和子进程可以继续进行 fork 调用，产生更多的进程，子进程可以通过 exec 类系统调用执行其他程序，这样就可以形成一个深度任意的进程树。

在使用 fork 调用产生新进程的时候，子进程将具有独立的地址空间，并拥有其父进程私有数据的一份副本。这样操作的时间和空间消耗都很大，现代的操作系统一般采用下面三种方法来解决：

写时复制技术。子进程并不实际对父进程的数据进行复制，而只是和父进程共享同一个只读副本。只有在子进程对数据进行修改时，系统才把涉及到的内存页复制到自己的地址空间来。

轻量级进程或线程。轻量级进程或线程允许子进程和父进程共享进程的核心数据结构，包括内存页表、打开的文件等。

使用 vfork 系统调用。vfork 建立的线程可以和其父进程共享全部的内存空间，为了避免数据的冲突，父进程一般要阻塞到线程结束再继续运行。

为实现进程的管理和调度，操作系统维护一个所有进程的表格，每个进程在这个表格中占有一个表项。该表项包含进程的状态、它的程序计数器、栈指针、内存分配状况、打开文件状态、调

度信息，以及其他在进程由运行转到就绪态时必须保存的信息，以保证进程在中断之后可以继续启动。

进程的取消要比创建直观得多，它不涉及复杂的算法，一般只要回收占用的内存，在系统进程表中删除相应的表项，更新相关的内核数据结构就可以了。

进程在创建之后，其状态会根据条件不断的变化。三种基本的状态是运行态、就绪态和阻塞态。运行态的进程是当前实际占有处理器的进程，当它被调度程序剥夺了运行权利但它本身在逻辑上还是可以继续运行的状态是就绪态。当一个进程在逻辑上不能继续运行时，它就处于阻塞态，典型的例子是它在等待可以使用的输入。这三种状态和它们之间的转换关系可以由图 5.3 表示。

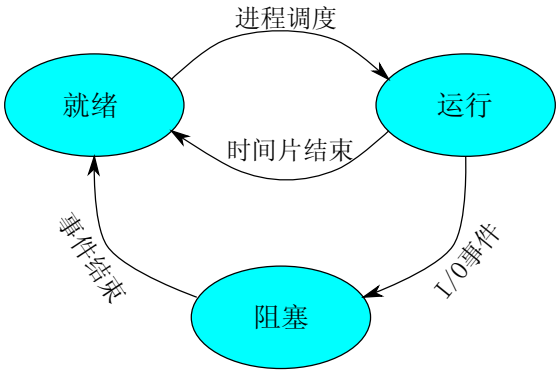

图 5.3: 进程状态转换图

进程可以通过 sleep 系统调用挂起一段时间，或者通过 wait 系统调用挂起到一个特定的事件发生。

二、进程调度

一个好的调度算法应当考虑很多方面，其中包括：

- 公平。确保每个进程获得合理的 CPU 份额。
- 效率。尽可能地利用 CPU 的全部资源。
- 响应时间。使交互用户的响应时间尽可能短。
- 周转时间。使批处理用户等待输出的时间尽可能短。
- 吞吐量。使单位时间内处理的作业数量尽量多。

显然这些方面不可能全部满足，调度程序必须采取一定的策略，不同用处的操作系统在调度策略上会有很大的差别，下面对两种常用的调度方法做一个简单的介绍。

1. 时间片轮转调度

时间片轮转调度算法是最古老、最简单、最公平也是应用最广泛的调度算法。每个进程被分配一个时间片，即该进程可以连续运行的时间。如果在这个时间片结束的时候进程还在运行，则强制进行一次进程切换，让另外一个就绪的进程开始运行。调度程序只要维护一个所有就绪进程的列表，就可以很好地实现这种算法。需要权衡的问题是如果时间片设计得太短会导致过多的进程切换，降低了 CPU 的效率；而太长的时间片又可能使交互程序的响应时间过长。

2. 优先级调度

对于使用进程优先级的系统，优先级高的进程要被率先执行。为了防止优先级高的进程无休止地运行下去，高优先级的进程必须定期主动放弃 CPU 进入睡眠状态（也可以说是阻塞状态的一种），或者采用动态优先级算法降低进程的优先级，以使低优先级的进程有机会运行。对于优先级相同的进程，一般还要采用时间片轮转方法进行调度。

在采用优先级调度算法的实时系统中，经常会发生一种 “优先级反转” 的现象，这时高优先级的进程会被低优先级进程影响而推迟执行。为了更好地理解这个概念，下面介绍一个优先级反转的例子。

如图 5.4 所示，系统中有三个进程 a、b、c，它们之间优先级关系为 a > b > c。初始状态 a、b 进程挂起，c 正在运行，并正在使用资源 i（时间点 1）。a 就绪后准备抢占 c 的执行但因为同样要使用资源 i，而继续挂起（时间点 2）。b 并不需要使用资源 i，在就绪后抢占了 c 的运行（时间点 3）。b 执行完毕之后，c 继续运行（时间点 4），直到 c 放弃资源 i，a 才可以抢占 c 的运行（时间点 5）。在这个过程中，本来 a 只要等待 c 占用资源 i 的时间就可以了，但由于进程 b 的存在，使得 a 的等待时间延长了。所以这里说在进程 a 和 b 之间发生了优先级反转。避免优先级反转的方法是动态改变占有资源的进程的优先级，在上面的例子中，c 在占有资源并同时发现 a 也需要这个资源的时候，至少要具有和 a 相同的优先级才可以避免被 b 的抢占。

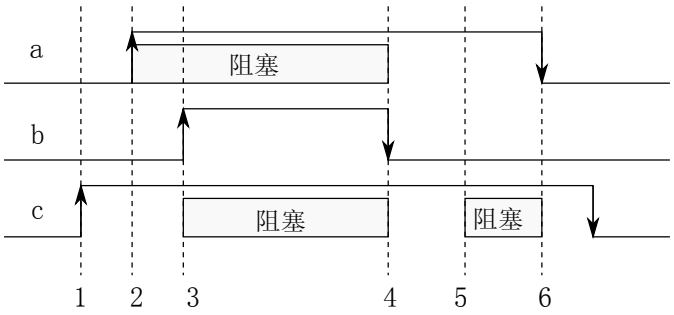

图 5.4: 优先级反转

在支持虚拟内存的系统中，如果就绪的进程被部分或者全部放在磁盘上，对它的调度就会需要更多的时间，对系统效率产生不可忽视的影响。在这样的系统中，一般采取两级的调度策略，底层调度程序只负责调度存在于内存中的进程，一个高级调度程序负责把在内存中驻留足够长的进程换出，并把磁盘上等待时间过长的进程调入内存。

三、进程的同步和互斥

系统的进程之间往往不是孤立的，它们可能需要互相协作，因此操作系统必须提供一种机制允许进程之间达到同步和安全地传递消息。正确的处理进程之间的协作是操作系统设计非常重要的部分，了解它们对于设计多线程程序的设计也有很好的帮助。

进程协作首先要解决的问题就是避免竞争条件。竞争条件是多个进程共享一些彼此都可以访问的存储区，其中可能蕴涵的访问冲突。例如进程 A 和进程 B 同时往一个数组中添加元素，共享变量 top 指示了当前的写位置。假设在某一时刻进程 A 准备向数组中添加数据，读到 top 的值为 5，但在实际写入之前发生了一次时钟中断，操作系统进行了一次进程切换，进程 B 开始运行。进程 B 也要向数组中添加元素，它读到的 top 值也为 5，然后它在数组的第五个位置写入了数据，并把 top 值更新为 6。当进程 A 恢复运行的时候，它仍然以为 top 变量的值是 5 然后在数组的第五个位置写入它的数据，这样就把进程 B 的数据覆盖了。

上面的例子只是实际情况的特例，现实中只要涉及到共享内存、共享文件，以及共享任何资源的情况都会引发类似的错误。前面例子的症结就在于进程 A 在对共享变量 top 的使用没有完结之前进程 B 就使用了它。因此必须使用一种“互斥”来确保当一个进程在使用一个共享元素时，其他进程不能进行同样的操作。这实际上就是前面提到的临界区概念，对进程共享元素的访问和操作都要在临界区那进行。一个好的互斥算法需要满足下面 4 个条件：

- 任何两个进程不能同时处于临界区。
- 不应对 CPU 的数目和速度作任何假设。
- 临界区外的进程不得阻塞其他进程。
- 不得使进程在临界区外无休止地等待。

最简单的互斥手段是采用关闭中断的方法。任务在进入临界区之后立即关闭中断，在离开临界区之前再打开中断。中断禁止之后，任务切换就不能发生，因此避免了竞争条件。但长时间关闭中断对系统的响应能力会产生影响，而且如果用户进程可以关闭中断的话就可能使系统失去响应能力，所以这种方法只适用于在操作系统内部、绝对可信的环境中使用。

1981 年 G. L. Petterson 提出了下面的互斥算法：

```
#define N    2          /* 进程数为 2 */
int turn;               /* 标志位 */
int interested[N];      /* 所有初始值为 0 */

void enter_region(int process)      /* 进程号为 0 或 1 */
{
    int other;                      /* 另一个进程号 */

    other = 1 - process;
    interested[process] = TRUE;     /* 表示想要进入临界区 */
    turn = process;                 /* 设置标志 */
    while (turn == process && interested[other] == TRUE); /* 空语句 */
}

void leave_region(int process)
{
    interested[process] = FALSE;    /* 离开标志 */
}
```

进程 0 必须调用 enter_region(0) 进入临界区，进程 1 则调用 enter_region(1) 进入临界区。在两个进程几乎同时调用这个函数的时候，它们都会修改 turn 变量，现在假设 turn 的最终值为 0，则进程 0 将在 while 语句处忙等，直到顺利进入临界区的进程 1 通过调用 level_region(1) 离开临界区为止。

有些 CPU 提供可以同时对变量进行检测和修改的汇编语句，这使得进入临界区的函数变得简单，因为这个语句使对共享变量的检查和修改操作不能被调度程序中断，排除了发生竞争条件的根本原因。假设这条指令是 tsl (test and set lock)，解决方案如下面的汇编程序所示：

```
enter_region:
    tsl r0, flag    ; 复制 flag 到 r0 寄存器,并将 flag 设置为 1
    cmp r0, #0      ; 检测上面保存的 flag 的值是否为 0
    jnz enter_region; 如果不等于 0,说明已经被其他进程上锁,再次循环
    ret             ; 成功进入临界区

leave_region:
    mov flag, #0    ; 置 flag 为 0
    ret
```

上面这两种解决方案有一个共同的缺点是都存在忙等，采用忙等的方式不但浪费计算机的时间，在优先级调度的系统中，高优先级的忙等会使低优先级的程序没有运行的机会。因此操作系统必须提供更高级的解决方案，使不能进入临界区的进程挂起，而不是长时间地等待下去。

E. W. Dijkstra 在 1965 年提出 “信号量” 的概念，并用两种通信原语 P 和 V 对信号量进行操作。对一信号量执行 P 操作时，检查其值是否大于 0，若是则把这个值减 1，否则进程进入睡眠（P 操作并没有返回）。执行 V 操作时，如果有一个或多个进程在这个信号量上睡眠，就由系统选择其中的一个将其唤醒，使它的 P 操作成功返回。此时信号量的值仍然为 0，但在其上睡眠的进程少了一个。

P、V 原语是单一的、不可分割的原子操作，操作系统可以在执行它们的时候暂时关闭中断来简单地实现原子性，具体就是在检测信号量、修改信号量以及在需要时使进程睡眠。因为这些操作需要的时间是很少的，短暂的关闭中断不会带来副作用。

四、进程间通信（inter-process communication）

前面介绍的进程同步实际就是进程间通信的一种方式，它们的特点是通信速度快，但传送信息量小、编程复杂。实际操作系统中会提供多种更加高级的通信方式，以适应于不同的应用场

合，例如共享内存、邮箱、队列、管道、套接字等。

五、死锁

在多任务系统中，当进程对设备、文件等资源取得独占性的访问权的时候，就有可能出现死锁，此时发生死锁的进程将被挂起，永远不能恢复运行，它们占有的资源也将不能被释放。例如，假设进程 A 拥有资源 1 的占有权，进程 B 拥有资源 2 的占有权，此时进程 A 申请希望得到资源 2，但因为它被进程 B 占有，所以进程 A 被挂起；接下来进 B 试图得到资源 1 的占有权，显然进程 B 也将不能被满足要求而被挂起，于是死锁就在进程 A 和进程 B 之间发生了。

死锁的定义是：假若在一个进程集合中的每个进程都在等待只能由该集合的其他一个进程才能引发的事件，那么这种状态就被视为死锁。由于死锁可能造成严重后果，操作系统必须尽量避免死锁，或者在死锁发生的时候采用必要的手段解除死锁。

死锁的发生有下面四个必要条件：

- 互斥条件：每个资源要么是被分配给一个进程，要么是空闲的。
- 部分分配条件：已经得到资源的进程还可以申请新的资源。
- 非剥夺条件：已经被分配的资源不能被剥夺，只能由占有它的进程释放。
- 循环等待条件：系统中有两个和两个以上进程组成的环路，该环路中每个进程都在等待相邻进程正在占用的资源。

根据死锁发生的条件，有四种处理死锁的策略。

1. 忽略死锁

简单的忽略死锁的存在是最简单的解决方案，该方案的存在基于如下事实：有些理论上可能的死锁在实际应用中发生的几率太小，而处理这种死锁的代价却相对太大。另外不同的应用系统和不同的死锁类型也会对死锁有不同的处理方案。例如在关键领域，要彻底地避免死锁的发生，不论代价有多大。而在一般的应用中，如果死锁发生的概率是 1 年 1 次，而系统每月都可能因为不同的外在原因而复位，那忽略这种死锁就是完全可以接受的了。

2. 检测死锁并恢复

操作系统可以每隔一段时间运行一种算法来检验是否有死锁存在，并通过一定的手段使进程从死锁状态解脱出来。检测死锁的算法有很多，这里只介绍一下最简单的基于资源分配图的方法，其他的方法请参考相关书籍。

下面的例子假设每种类型只存在一种资源，例如一个系统只有一台打印机、一台扫描仪。然后以下面的方式构造一个有向图，称为资源分配图（图 5.5）：有向图有两类节点，圆形节点代表进程、方形代表资源。由资源到进程节点的有向边表示资源被那个进程占用；由进程到资源的有向边表示当前进程在申请该资源。如果在资源分配图中存在一个环的话就说明存在一个死锁。例如：

进程 A 拥有资源 1、3，并需要资源 2
进程 B 拥有资源 2，并需求资源 3
进程 C 不拥有资源，需求资源 3、4
……

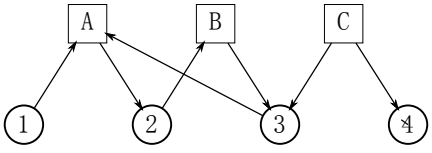

图 5.5: 有死锁的资源分配图

从图 5.5 中可以看出进程 A、B 之间存在死锁，解决死锁的办法可以有如下几种：

资源剥夺法。将一个进程占有的资源强制收回，以破坏上面存在的死锁。这种方法和资源的特性相关，一般比较困难，甚至不大可能，或者可能需要人工的干预。

回退法。记录进程占有不同资源时候的状态，当检测到死锁时候，把一个进程的状态恢复到更早的某个状态，那时它还没有获得引起死锁的资源，只要把该资源分配给另外一个与之竞争的

进程就避免了发生死锁的条件。

杀死进程法。杀死一个或若干进入死锁状态的进程是最直接、最简单的解决死锁的方案。这样就可以以最小的代价避免更大的损失。如果被杀死的进程还可以通过重新启动来恢复的话，采取这种方案就很可取了。

3. 采用特殊的资源分配算法，避免死锁的发生

最理想的死锁解决方案是采用一种精巧的算法来避免发生死锁，但如果要实现这样的算法就必须事先知道进程对资源的需求情况。例如著名的银行家算法可以很好地在保证安全的前提下分配资源，在许多操作系统的书籍中都有论述。但由于进程对资源需求的不可预见性使得这些算法都没有太大的实用价值。

4. 通过破除四个必要条件之一来防止死锁的发生

根据产生死锁的四个必要条件，通过保证其中至少有一个条件不成立，那么死锁就不会发生。采用这种方式要比寻找合适的资源分配算法容易和实际得多。下面分别对四个条件进行分析，以确保这个方法的可行性。

互斥条件。如果资源不被单一的进程独占，那么死锁就不会发生。采用这种方式避免死锁的典型例子是经常应用于打印机的 Spooling 技术。此时打印机作为一种资源只被打印服务程序访问，其他进程通过和打印服务程序通信来实现打印任务，由于打印服务程序不会申请其他资源，别的进程也不会和它争夺打印机，死锁一般就不会发生。但 Spooling 技术在磁盘空间的申请上还是可能发生竞争而导致死锁。例如两个进程分别占用了可用 Spooling 磁盘空间的一半而等待更多的空间，那么必然引起死锁。但实际发生这种类型死锁的几率是很小的，Spooling 技术已经大大降低了一种更普遍死锁的几率。

部分分配条件。如果进程必须一次申请全部需要的资源，而且只有全部的资源都可以获得的时候才分配给进程，保证它不再需要更多的资源，那么进程就可以运行结束。当资源分配失败的时候，进程阻塞，直到全部需要的资源可以获得。但实际上很多进程并不能预知运行其间需要的全部资源，因此另外一种解决方案是当进程申请新资源的时候，暂时释放当前占用的所有资源，然后再试图一次获得全部需要的资源。

非剥夺条件。破坏资源不可剥夺的条件比较困难，前面在介绍资源剥夺法破坏死锁状态时讨论了这个问题。

循环等待条件。破坏循环等待条件有几种方法，一种是保证每个进程在任何时刻只能占用一种资源，如果要申请第二个，必须先释放第一个。但存在很多情况使得这个限制是无法忍受的。另外一种方法是给所有资源一个全局编号，进程在申请资源的时候只能按照编号的顺序进行，这样就避免了资源需求的环路。这个算法的限制实际可以放宽到进程在申请新资源的编号必须比当前占有的所有资源编号高，例如一个进程申请了 7 号 8 号资源并随后释放了它们，就没有必要阻止它现在申请 1 号资源。

5.2.3 设备管理

除了处理器和内存之外，一个计算机系统还包括各种各样的设备，它们都具有各自的特点，具有不同的控制和数据传输方式。这些设备以及与设备有关的各种控制和支持部件都是系统的资源，都必须由操作系统统一管理，以达到高效、安全的使用。一种应用广泛的操作系统必须能够尽可能多地支持各种不同的设备，所以设备控制部分的源代码往往会成为操作系统中最大的一部分，设备操作的复杂性也使得设备管理成为操作系统中最繁杂和最与硬件紧密相关的部分。

一、设备的基本概念

在操作系统中，设备的概念是广义的，它既可以指进行实际输入/输出操作的物理设备，也可以指控制这些设备的支持部件，甚至可以指为提高设备利用率而采用某种软件技术形成的逻辑设备和虚拟设备。由于各种设备的功能都和数据的输入或输出有关系，一般称作输入/输出设备，或者 I/O 设备。

设备的种类繁多，从不同的角度可以进行不同的分类。

1. 按照功能分类

- 输入设备，指将各种数据送入计算机的设备，如键盘、麦克风、温度传感器等。

- 输出设备，指将处理加工过的数据显示、输出、再生出来的设备，如打印机、显示器等。
- 存储设备，指能将信息暂时或长久保存起来的设备，如磁盘、磁带等。
- 供电设备，指向系统提供电力的部件和设备，如电池、UPS 等。
- 网络设备，指能够组成网络，并通过各种网络协议与网络上的其他设备进行通信的部件，如调制解调器、网卡等。

2. 按照设备的数据组织方式分类

- 块设备，用于存储信息，信息的存取以 “块” 为单位，具有一定的数据结构，以存储设备为代表。其基本特点是数据传输速率比较高，可以通过寻址读写块设备的任何一块。
- 字符设备，以字符为基本的单位进行数据的组织和传输，其基本特点是传输速率一般较低，不可以寻址。大部分的常见设备都是字符设备，如显示终端、打印机、串行口等。

这种分类方式在实际中使用得比较多，但它的分类标准不是很严格，而且不是所有设备都可以按照这个标准进行分类，例如一些设备既可以用块方式访问也可以按照字符方式访问，另外一些设备如时钟、网卡等则不好划分到任何一类中。

3. 其他分类方式

按照一个设备是否是实际存在的物理设备，可以把设备分成 “物理设备” 和 “逻辑设备”。按照设备资源的属性可以分成 “独占设备”、“共享设备” 和 “虚拟设备”，前面介绍的 Spooling 技术就是一种虚拟设备的实现。按照数据传输速率来分可以分为 “高速设备” 和 “低速设备”，人机交互设备一般都是低速设备，而其他通信设备一般都是高速设备。按照数据寻址方式的不同，还可以分成 “顺序读写设备” 和 “随机读写设备”。

二、设备管理子系统的功能

输入输出设备是操作系统管理的重要资源，但由于设备的多样性，使得设备管理的实现比较困难。一般来说，设备管理部分的功能主要关注下面三个方面：

1. 硬件抽象

用户总是希望可以方便地使用 I/O 设备，但是硬件的 I/O 过程都比较复杂，涉及到大量的 I/O 细节，如寄存器、中断、控制命令、状态机等。为了弥补硬件接口带来的不便，操作系统的设备管理部分就必须提供简便易用的高级逻辑接口，实现抽象接口到物理接口的转化，即将高级的逻辑操作转化为低级的物理操作，以便掩蔽设备物理操作和组织的细节。另外，抽象接口除了掩蔽硬件细节外还要掩蔽依赖于硬件的软件细节，这两种细节合称机器相关细节，即依赖于硬件的细节。设备抽象接口可以是由设备管理功能接口和文件系统功能接口共同提供的，也就是说设备接口可以包含在文件系统的统一接口中。

具体来说，需要掩蔽的硬件细节包括下面几个方面：

- 与接口寄存器相关的 I/O 操作过程。这个过程由准备、发出命令、数据传输、控制状态（轮询或者中断）及错误处理等构成。
- 错误处理是 I/O 控制软件的重要部分，总的说来，错误应当在尽可能接近硬件的层次来处理。如果控制器发现了一个读错误，它就应该努力自行纠正。如果控制器不能处理该错误，就应当交给相应的管理程序来处理。很多错误都是偶然发生的，往往重复一次操作就可以消除错误。仅当底层不能正确处理错误时，才交给高层来处理。
- 掩蔽设备的物理细节和不同设备的物理性质的差别，提供抽象的统一的设备逻辑性质。同一个程序在不同次的运行中，所使用的设备很可能是不同的，这是由于不同的环境和不同的系统配置造成的。对物理设备的逻辑抽象可以让程序不必关心物理设备的操作和组织细节，而只看到具有逻辑名称和逻辑性质的逻辑设备和逻辑操作，不关心具体 I/O 设备是哪一个具体设备或者是哪个具体文件，设备改变而程序不必改变、不必修改、不必重新编译。虽然不同的设备有很大的不同，它们的设备驱动程序也差别很大，但用户可以不必关心，一切由此带来的问题应该由操作系统来管理，所使用的设备对于应用程序是透明的。

I/O 设备与文件系统接口的统一的必然性体现在用户程序经常需要在文件和设备间变换实际操作的对象，可行性体现在对文件的访问和对设备的访问有共性可循。输入和输出设备可以抽象地看作一个顺序数据的源和目标。在每个用户面前表现的是一个可以被一组类似于为使用文件而提供的简单通用操作所控制的虚设备集合。实际上，如果假设对文件的存取都是顺序的，

虚设备和文件没有什么区别，虚设备可以用那些为存取文件提供的相同逻辑来操作。

计算机系统对输入/输出设备的控制模式主要有：轮询方式、中断方式、DMA 方式和 I/O 通道。

轮询方式是早期计算机系统采用的 I/O 技术，处理器定时对所有设备进行查询，如果有要求就加以处理，完成后返回继续工作。由于处理器的速度比外设要快得多，一般不会发生处理不及时的情况，但由于轮询方式浪费了处理器的时间，在现代系统中已经很少使用。

中断方式是一种在异步对外设进行操作的同步技术，可以在外设需要处理器的时候通知处理器，是操作系统管理外设的重要手段。采用中断方式可以避免轮询方式中的低效率，但操作系统必须尽可能地把中断隐藏到操作系统的内部，让用户尽可能少地干预。

许多外设控制器还支持 DMA 方式传递数据，采用这种方式可以高效地传递大块的数据而不必占用处理器的时间。当使用 DMA 的时候，除了要向控制器提供要传递的数据起始地址，还要提供相应内存操作的起始地址及传递的字节数。当传递开始的时候，DMA 控制器使用总线，进行实际的数据传递。因此操作系统只要启动 DMA 操作，当此操作结束的时候，数据就已经准备好了。在整个数据传输过程中，CPU 可以做一些其他的事情，提高处理器和 I/O 的并行操作程度。

I/O 通道是 DMA 方式的扩展，它可以进一步减少 CPU 的干预，即把对一个以数据块为单位的操作，改变为对一组数据块及相关控制和管理为单位的操作。通道是一种特殊的 I/O 处理机，它有很强的 I/O 处理功能，可以独立的完成系统处理器交付的 I/O 操作任务。在这种方式中，处理器只需进行 I/O 操作的委托，其他后续的所有 I/O 操作都可以由通道自己进行。

2. 系统效率

系统效率是衡量操作系统性能的一个重要指标，输入/输出设备的性能经常会成为系统性能的瓶颈。CPU 性能越高，输入/输出设备性能和它的反差也越大，如何让两者协调工作而不降低处理器的性能，是设备管理的重要任务。在硬件上为提高效率产生了中断、DMA 等技术，软件上不但要配合使用这些技术，还可以用纯软件的技术来进一步挖掘效率和利用率的潜力。操作系统可以借助抽象接口通过共享等技术在内部实现设备管理的优化，让用户从复杂的硬件优化编程中解脱出来。

1）并发

在多任务系统中支持设备的并发使用是提高效率的重要手段。对于可以共享的设备（非独占设备），系统可以根据每个设备的特征来全局调度、安排设备的操作。例如对多个磁盘的读写操作可以进行排序，以使操作间磁头的移动距离尽可能地短。对于独占的设备，可以采用前面介绍的 Spooling 技术来实现并发。

2）缓冲

缓冲是一个广泛采用的技术，广义地讲，缓冲就是在通信过程中，为了双方的速度匹配而引入的中间层次，这个层次的速度比通信双方中较慢的一方快，而与较快的一方更为匹配。这种技术已经被众多的实践证明是非常有效的。

缓冲技术不但被应用在软件层次上，也经常有纯硬件的缓冲实现。按照不同的层次，缓冲可以分为下面四种：

- CACHE，即高速缓存
- I/O 设备或控制器内部的硬件缓冲区，如磁盘控制器的缓冲区用来把恒定速率的比特流保存为以字符为单位的块结构数据。
- I/O 系统在内存中开辟的缓冲区，如操作系统在内存中开设的文件系统缓冲。
- Spooling 技术实质上也是缓冲技术，是为慢速 I/O 设备在外存中开设的缓冲。

前两种是纯硬件的缓冲区，后两种需要软件的管理，称为软件缓冲区。

3. 保护

操作系统对外设的管理还要体现在让用户可以安全正确地使用设备，即由设备传递或管理的数据应该是安全的，不会被破坏和非法访问。

三、设备管理软件

设备管理子系统的功能最终都要由程序来实现，其基本的思想是分层结构，也就是把设备管理软件组织成一系列的层次，其中底层与硬件相关，把硬件与高层的软件隔离，而高层软件则向

应用层提供友好的、清晰一致的接口。

I/O 软件的层次一般分为下面四个：

1. 中断处理程序

在中断方式下，处理器和 I/O 设备之间的数据传输的大致步骤如下：当某个进程需要外设数据的时候，先发出指令启动设备进行数据准备，并保证该设备的中断被允许，以便在需要的时候中断服务程序可以被执行。当此进程发出启动设备的命令之后，就放弃处理器处于挂起状态，等待相关 I/O 操作完成。当 I/O 操作完成之后，外设触发中断信号，处理器响应中断，转向预先设计好的中断处理程序对数据传输工作进行相应处理。最后得到数据的进程转为就绪状态，等待在随后被调度程序选择继续运行。

在实际的系统中，中断发生的频率可能非常高，如果中断不能被及时处理就可能对系统产生灾难性的后果。因此中断处理程序的设计必须非常高效，并能尽快地返回以避免丢失其他中断。一般操作系统负责系统中全部中断的管理，其中实际传输数据的处理和控制则由相应的设备驱动程序来负责。

2. 设备驱动程序

操作系统中直接管理和控制 I/O 设备的程序叫做驱动程序，每种设备都会有一个或者多个驱动程序与之对应，操作系统通过与驱动程序的交互达到对设备进行控制的目的。如果操作系统不提供驱动程序的，用户程序就必须直接对硬件进行操作，这不但提高了编程的复杂度，也可能因为错误的硬件操作对系统产生破坏，在安全性比较高的系统中，直接操作硬件甚至是不被允许的。

驱动程序的设计也会被分成多个层次，相同类型的 I/O 设备可以被抽象成类似的软件模型，将实际的硬件操作部分在驱动程序中独立出来。这样，当同种类型的新设备出现时，就可以只编写和硬件相关部分完成驱动程序的设计。一般而言，驱动程序需要完成下面四个方面的工作：

- 向有关输入输出设备的各种控制器发出控制命令，并且监督它们的正确执行，进行必要的错误处理。
- 对各种可能的有关设备排队、挂起、唤醒等操作进行处理。
- 执行确定的缓冲区策略。
- 进行比寄存器接口更高层次的特殊处理，如代码转换等，它们是硬件相关的，所以不能被放到高层次的软件中处理。

对于使用中断的设备，驱动程序也要包含相应的接口为操作系统调用，这部分程序需要对中断进行后续的处理。如传递数据的正确性检验、有可能的错误处理、有可能的进程唤醒、有可能的后续 I/O 操作等。

具体驱动程序实现的策略是与操作系统紧密相关的，不同的操作系统会有不同的设计思想。

3. 与设备无关的操作系统软件

设备管理软件的层次性设计会使得操作系统中存在大量的与具体硬件无关的设备管理代码。这部分代码和驱动程序的划分不是很明显，具体策略的不同会有不同的实现方法。例如考虑到效率因素，一些本来可以独立于设备的代码就可以在设备驱动程序中实现。

与设备无关软件的基本任务是实现一般设备都需要的 I/O 功能，并且向用户层软件提供一个统一的接口。

4. 用户空间的 I/O 软件

除了操作系统内部的 I/O 软件，还有一小部分 I/O 管理程序是通过库函数和用户程序链接在一起的，甚至还有完全运行于内核之外的 I/O 软件。例如在 C 语言中使用很普遍的 printf 函数，它可以用来格式化一个字符串，然后通过 write 系统调用将这个字符串输出到外设。标准的 I/O 库包含了许多涉及 I/O 的过程，它们都是作为用户程序的一部分运行的。

5.3 编译 Linux 内核

5.3.1 Linux 内核代码结构

内核(kernel)是 Linux 操作系统的核心,它负责管理系统的进程、内存、设备驱动、文件和网络系统,决定了整个系统的行为方式。可以说内核就是操作系统本身,我们所使用的 Linux 命令实际只是在这个操作系统上的应用程序。

Linux 的内核源码在网络上一般通过一个很大的压缩包提供,例如光盘上提供的 linux-2.6.33.tar.bz2 就是版本号为 2.6.33 的内核源代码压缩包。Linux 的内核版本号由三组或四组数字组成,中间由小数点分开。其中前两组表示主版本号,第三组表示发行号,可能存在的第四组代表稳定版的最新更正号。其中如果第二组是偶数,表示稳定的内核,否则表示开发中的内核,例如我们使用的 2.6.33 版本就是稳定版。稳定版本内核的新的发行号版本主要用来纠正已经发现的错误和提供新的功能、驱动。而开发版本则可能会在数据结构和具体实现方法上与之前的内核版本有比较大的变动,并不适合一般的开发所应用。最新的内核版本可以在 http://www.kernel.org 上得到。

2.4 系列的内核对 ARM 处理器的支持有限,必须使用 ARM-Linux 官方网站所提供的补丁。但到了 2.6 的内核之后,ARM-Linux 网站已经直接把处理器相关的优化和支持补丁添加到标准内核中,因此也不再需要单独打补丁了。但对于不同的开发板,还需要使用不同的专用补丁。例如要准备适合实验板的内核源码文件:

首先解压内核源码(以下操作假定在/home/embedded 目录中进行),并建立一个名为 linux 的符号链接,方便对目录的访问:

```
$ tar xfvj linux-2.6.33.tar.bz2
$ ln -s linux-2.6.33 linux
```

为 linux 源代码打上定制的补丁(下面示例中,部分命令的输出被忽略):

```
$ cd linux
$ zcat ../2.6.33-at91.patch.gz | patch -p1
$ cat ../2.6.33-at91-pku.patch | patch -p1
```

在应用补丁的时候,要根据补丁的格式采用不同的命令,常见的格式是用 bzip 压缩(扩展名为 bz2)或用 gzip 压缩(扩展名为 gz),或者没有压缩。当应用多个补丁的时候也要注意应用顺序,因为有些补丁具有依赖关系,这个依赖关系往往可以从文件名上看得出来。如果一个补丁依赖于另外的补丁,它会在文件名中使用其所依赖的补丁文件名作为前缀,例如在前面的例子中,第二个补丁的文件名前缀包含第一个补丁文件名前缀。

以上操作执行完毕后,一套完整的 ARM Linux 内核代码树就保存在了/home/embedded/linux 下面,部分目录的具体内容如图 5.6 所示。

5.3.2 内核编译步骤

通过重新编译内核,我们可以修改内核所包含的软件模块和硬件支持模块,以达到量体裁衣的目的,在一定程度上调整操作系统的性能和运行效率。对于资源受限的嵌入式系统,调整内核的配置更加显得关键了,我们总会希望我们的系统运行得更快、占用的空间更小。由于嵌入式系统的硬件设备经常是比较特殊的,我们必须对内核进行修改以驱动自己的硬件,而每次修改之后都要对内核进行重新编译。另外 Linux 操作系统并不是微内核结构,操作系统的大部分功能都集中在一个不到 2MB 的核心里面。这个尺寸对于很多嵌入式设备来说,已经显得很大了,必须通过对内核进行重新配置,把不必要的部分裁减出去。

```
├── init                内核初始化代码
├── kernel              内核核心部分：进程、模块、定时
├── mm                  内存处理部分
├── arch                平台相关的代码
│   └── arm             基于 ARM CPU 的内核代码
│       └── boot        引导部分代码
│           ├── configs 部分开发板的预定义配置文件
│           ├── kernel  内核核心的处理器相关部分
│           ├── lib     硬件相关工具函数
│           ├── mm      内存处理的处理器相关部分
│           ├── include arm相关的头文件
│           └── mach-xxxx 不同开发平台的相关部分
├── fs                  文件系统部分
├── net                 网络相关代码
├── ipc                 System V 的进程通信
├── lib                 通用内核函数
├── include             头文件
│   ├── linux           系统、内核相关部分
│   └── asm-generic     平台无关部分
├── drivers             设备驱动程序
│   ├── block           块设备驱动程序
│   ├── char            字符设备驱动程序
│   ├── net             网络设备驱动程序
│   ├── video           视频设备驱动程序
│   └── misc            杂项设备驱动程序
└── Documentation       内核文档资料
```

图 5.6: 内核代码树结构

编译内核的基本步骤是在源代码第一层目录下依次执行 make config; make zImage; make modules。下面对每一部分进行分别介绍。

make config 命令用来对内核进行配置，配置的结果保存在.config 文件中。配置内核是编译内核最主要的环节，在这个过程中，我们可以决定哪些部分被编译到内核，哪些部分被裁减，哪些部分被编译成模块。被编译成模块的部分可以在内核运行时动态加载，减少了系统资源的占用。

make config 命令的用户界面很不友好，替代的命令有 make menuconfig 和 make xconfig，后者需要 X 的支持，一般选择 make menuconfig 就很方便了。另外源码中有一些预定义的配置文件保存在 configs 目录中，如 configs/pku_defconfig。可以通过下面命令把某个预定义配置保存为当前配置。

```
$ make pku_defconfig
```

make menuconfig 是一个菜单的界面，通过这个菜单们可以对源代码的功能模块进行浏览。其中带有 “—>” 符号的表示这个菜单项包含子选项，选项前面符号的含义如下：

“< >” 表示这个选项可以作为模块编译到内核；

“[ ]” 表示这个选项不能作为模块编译，只能静态编译；

“<M>” 表示已经选择这个选项作为模块；

“<Y>” 和 “[Y]” 表示这个选项已经被静态编译到内核。

操作的时候按 “M” 表示模块，“Y” 表示包含到内核，“N” 表示不包含。多数的选项可以还通过 <Help> 按钮来显示一个简短的帮助信息。

除了 make menuconfig 命令之外，内核的编译系统还支持其他几种配置界面，它们是：

- make xconfig，图形界面的配置工具，基于 qt 库。
- make gconfig，图形界面的配置工具，基于 gtk 库。
- make allnoconfig，仅配置 “强烈建议” 的模块到内核，其他配置都不选择。

内核配置完毕之后就可以用 make zImage 命令生成最终的内核映像，如果编译成功，可以在 arch/arm/boot/目录中找到 “zImage” 文件。这个内核映像是经过压缩的，并且增加了一个用来

解压内核的程序，而没有压缩的内核被保存为 vmlinux 文件，其符号表被保存为 System.map 文件。这两个文件在源代码的根目录下，可以用来对内核进行分析和调试。

对于非交叉编译，还可以运行 make install 来把内核安装到操作系统中，但在嵌入式开发中，往往只能使用手动安装内核，具体参考本章的实验。

make modules 命令用来对前面设置为 “<M>” 的部分进行编译，生成对应的.ko 文件，成为动态加载的模块。

make modules_install 命令把生成的内核模块安装到文件系统/lib/modules/<kernel-version> 目录中，并通过 depmod 命令检查模块的依赖关系，把结果保存在 modules.dep 文件中。

make mrproper 命令的目的是清除源码中一切由于编译产生的中间文件，得到一个 “干净” 的源码树。因此这个命令并不是每次都要执行，一般只有在怀疑源码编译的过程出现异常，需要恢复源代码从头开始的时候才需要。

make distclean 命令也用来删除源代码中的生成文件，但它比 mrproper 更彻底，还会删除编辑器产生的临时文件和用 patch 命令生成的.rej 和.orig 文件。这个命令可以在比较两个版本 Linux 源码的不同之前使用，用来生成正确的补丁文件。

在配置内核的时候，各种编译选择非常多，而且互相关联，每个选项对应一个宏的定义，所有的配置信息最终保存在内核代码目录中的.config 文件中。这个文件可以直接用编辑器修改，修改之后必须执行 make oldconfig 使得所做的修改生效。下面的列表是一个.config 文件的头几行：

```
#
# Automatically generated make config: don't edit
# Linux kernel version: 2.6.33
# Thu Jul 19 17:12:59 2010
#
CONFIG_ARM=y
CONFIG_SYS_SUPPORTS_APM_EMULATION=y
CONFIG_GENERIC_GPIO=y
CONFIG_GENERIC_TIME=y
CONFIG_MMU=y
# CONFIG_NO_IOPORT is not set
CONFIG_GENERIC_HARDIRQS=y
CONFIG_TRACE_IRQFLAGS_SUPPORT=y
CONFIG_HARDIRQS_SW_RESEND=y
CONFIG_GENERIC_IRQ_PROBE=y
CONFIG_RWSEM_GENERIC_SPINLOCK=y
```

make mrproper 和 make distclean 命令会把.config 删除，所以在执行这两个命令之前保存一下这个文件是一个比较好的习惯。

比较重要的编译选项有：

1. “Code maturity level options” 代码成熟度选项

   这里边有两个选项，其中 Prompt for development and/or incomplete code/drivers 一般要选择，否则很多编译选项就看不到了。

2. “General setup” 一般设置

   这部分内容非常多也都非常重要，涉及到了很多 Linux 操作系统比较基础的支持。例如与进程间通信关系密切的 “System V IPC” 和 “POSIX Message Queues” 等都在这里进行设置。

3. “Loadable module support” 可加载内核选项

   这个就是 Linux 最吸引人的模块机制，如果不准备把所有的内容都编译到内核，就要选择下面的 “Enable loadable module support” 和 “Module unloading”。“Automatic kernel module loading” 选项允许内核根据需要自动加载内核模块。这个功能对嵌入式设备一般不那么重要。

4. “System Type” 系统类型

这个菜单用来设置具体的处理器和平台类型，这里我们要设置 ARM system type 为 Atmel AT91，并进一步选择 Atmel AT91 Processor 的型号。

5. “Kernel Feature” 内核特色

这个菜单里面有个比较重要的选项 “Preemptible Kernel”，它使得 Linux 可以在内核态进行任务切换。对于实时性要求比较高的嵌入式系统来说，设置这个选项会减少系统在关键任务上的反应时间。

6. “Boot options” 启动选项

与启动相关的选项在这个目录中，比较常用的是 “Default kernel command string”，这里用来设置内核的启动参数，在系统开发的初期，这个参数可能要经常修改。内核参数多数是一系列由等号连接的 “赋值对”，例如 “root=/dev/mtdblock3 console=ttyS0” 设置了两个内核参数——root 和 console。常用的内核参数在后面还有更多的介绍，多数 bootloader 支持在执行内核的时候设定内核参数，这为内核调试提供了很大的方便。

7. “Networking” 网络

与网络相关的一些配置（除了网卡驱动）都在这个菜单目录下，比较重要的是其中的 “Networking options”，有关 TCP/IP 的一些设置都在里面。如果要使用网络根文件系统（File System 菜单中设置），就必须设置 “kernel level autoconfiguration” 选项。

8. “File systems” 文件系统

与 Windows 系统只支持很少的文件系统不同，Linux 在虚拟文件系统（VFS）的支持下，可以很容易地支持多种不同的文件系统。对于嵌入式系统，我们只选用需要的文件系统支持就可以了。在嵌入式系统中比较常用的是 “Journalling Flash File System v2” 和 “Compressed ROM file system”，它们在 “Miscellaneous filesystem” 子菜单中。前面提到的网络根文件系统在 “Network File Systems” 中设置，这里要设置 “NFS file system support” 和 “Root file system on NFS”。如果这里看不到后一个选项，就可能是前面的 “kernel level autoconfiguration” 没有设置。

9. “Device Drivers” 驱动程序

驱动程序是 Linux 操作系统代码量最大的一个部分，对于不同的硬件平台，必然包括不同的驱动程序，当系统增加了新的硬件，它的驱动程序也很可能就可以在这里找到。下面几个条目都是 Device Drivers 之下的子目录。

10. “Memory Technology Devices (MTD)” 存储技术设备

MTD 设备一般指用来保存 Linux 内核和文件系统的 Flash、Ram 等设备，这部分对于嵌入式系统是很关键的，因为大部分嵌入式设备采用 Flash 来保存文件系统。

11. “Network device support” 网络设备支持

ARM 平台下支持的网络设备并不多，主要在 “Ethernet (10 or 100Mbit)” 里面。我们用到的是 “DM9000”。

12. “Input device support” 输入设备支持

这个部分包括鼠标、键盘等输入设备的支持，嵌入式平台中常用的是触摸屏。其中 “Event interface” 是一个通用的输入设备接口方式，一般要选择支持，否则部分程序可能无法正常工作。

13. “Character devices” 字符类设备

这个菜单下有一些非常标准的字符类设备，我们平时与嵌入式的交互就是通过终端（console）这个字符设备进行的。由于我们这个终端是 “连接” 在串口上的，所以要在 “Serial drivers” 中选择合适的串口设备驱动，否则我们就失去了与设备最方便的交互手段。

14. “USB support”USB 支持

这个部分包括大部分的 USB 设备的支持，由于 USB 设备的跨平台性很好，多数能在微机上使用的 USB 设备都可以很好地在嵌入式设备中使用。另外这部分还包括 USB 主设备的驱动（“USB Host Controller Drivers”）和 USB 从设备的驱动（“USB Gadget Support”）。前者用来支持嵌入式设备中的 USB 主控芯片，后者让嵌入式设备作为 USB 设备连接到其他主机上。在 Linux 中 USB 从设备被称为 “USB Gadget”。

15. 其他

还有部分的设置是和其他硬件相关的，有兴趣了解的话可以直接阅读提供的帮助文档，或者

在内核代码的 Documentation 目录阅读相关说明。

5.3.3 Linux 内核启动流程

了解 Linux 操作系统的启动过程在实际的开发过程中非常重要，因为在系统启动失败的时候，可以根据 Linux 的启动流程，一步步找到失败的环节来定位问题。由于 Linux 操作系统还在不断的发展，这里的介绍是以目前最新的 2.6.33 版本内核为基础。

首先看一下编译内核的过程中最后几条提示信息，下面的列表是在编译 AT91SAM9261 开发板内核的过程中截取的。

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

我们已经知道源代码根目录下的 vmlinux 是没有压缩的 Linux 内核，而 zImage 是压缩过的内核，从 vmlinux 到 zImage 要经过如下的过程：

1. 使用 objcopy 命令获得二进制格式的 Linux 内核代码：Image。
2. 将 Image 通过 gzip 命令压缩为 piggy.gz。
3. 汇编 piggy.S 获得 piggy.o，piggy.o 中已经包含了 piggy.gz 作为数据。
4. 汇编 head.S，编译 misc.c，将 head.o、piggy.o、misc.o 链接成 vmlinux。
5. 使用 objcopy 命令获得最终的二进制格式的 zImage

piggy.S 的内容比较简单，其中定义了 input_data 和 input_data_end，并将 piggy.gz 包含了进来：

```
        .section .piggydata,#alloc
        .globl  input_data
input_data:
        .incbin "arch/arm/boot/compressed/piggy.gz"
        .globl  input_data_end
input_data_end:
```

head.S 中的代码是 Linux 内核在启动的时候最先执行的代码，其中主要是对硬件平台做最基本的初始化，并建立了调用 C 语言程序的基本运行环境，接着调用 misc.c 中的 decompress_kernel 函数解压内核，最后跳转到解压后的内核进行执行。

misc.c 中的代码仅负责解压 gzip 压缩的内核，其中 decompress_kernel 函数包含如下内容：

```
        makecrc();
        putstr("Uncompressing Linux...");
        gunzip();
        putstr(" done, booting the kernel.\n");
        return output_ptr;
```

其中 “Uncompressing Linux” 就是 Linux 内核在启动时候所能看到的第一条提示信息，它甚至不是通过 printk 函数（参考第六章的介绍）打印的，在 dmesg 命令中也看不到。

解压后的 Linux 内核入口地址是 _stext，这部分代码在 arch/arm/kernel 目录中的 head.S 中，连同 head-common.S，这个代码主要对体系、处理器和平台的类型进行检查，初始化 MMU，建立页表项并启动虚拟地址，最后调用了 start_kernel 函数。

到 start_kernel 就已经进入了 C 语言的世界了，它存在于 init/main.c 中。这个函数看起来十分简单，只是依次调用了一系列函数用于初始化内核的各个部分，并最终完成 Linux 操作系统的启动。下面就来简单介绍一些比较重要的初始化函数。

setup_arch 函数负责与具体平台相关的初始化工作，例如获得 RAM 的位置和容量。setup_arch 还会初始化一块叫做 bootmem 的内存为内核启动使用，并调用 paging_init 来打开 MMU。setup_arch 函数位于 arch/<host>/kernel/setup.c

trap_init 一般位于 arch/<host>/kernle/traps.c，它负责初始化处理器的中断向量，并负责部分的中断处理，但实际的中断允许还要在后面才打开。

init_IRQ 函数负责初始化和硬件相关的中断处理子系统，例如中断控制器的初始化，而最终的中断线允许是在驱动程序调用 request_irq 的时候进行的。

time_init 初始化系统的时钟硬件，安装时钟中断的处理程序，并配制时钟定时产生 “tick” 中断，这个中断的处理程序一般叫做 do_timer_interrupt。

console_init，用来在内核的虚拟终端系统初始化之前显示内核的消息。如果系统配制了串口终端，这个函数还负责这个串口终端的早期初始化工作。

calibrate_delay 用来校准内核的 BogoMips 参数，这个参数体现了系统循环等待的速度，可以保证在速度不同的机器上使用延时可以比较准确。BogoMips 的校准依赖于 jiffies 值，这个值是系统从启动开始的 “tick” 数目，因此在执行这个函数之前必须保证 time_init 是成功的。

check_bugs 函数定义在 include/asm-<host>/bugs.h 中，它可以用来检测已知的处理器 bug 并进行适当的处理。

rest_init 函数是 start_kernel 调用的最后一个函数，它用来释放不再需要的内存，建立内核线程 init 来完成内核的启动。当根文件系统被正常加载之后，init 通过调用 run_init_process 来执行 init 程序。init 程序的搜索顺序是：内核参数 init=<program_name> 指定的程序，然后是/sbin/init、/etc/init、/bin/init、/bin/sh。

当 init 程序开始运行之后，Linux 内核的启动就完成了，接下来开始的是用户态进程的初始化，由/etc 目录下的一些脚本文件控制。

5.4 Linux 文件组织结构简介

5.4.1 Linux 文件目录

在 Linux 操作系统中，文件采用目录树的结构进行组织，目录树的根结点被称作 “root”，用符号 “/” 表示。文件系统中的每个目录都可以作为一个 “挂载点” 将实际的文件系统设备（如 Falsh，硬盘等）挂载（mount）上。一个完整的 Linux 文件系统必须包含支持操作系统启动的一切文件，它最少包含下面这些目录：

1. /dev 设备文件目录，这个目录中包含的并不是传统的文件，它们分别是系统中各种设备的标识，操作系统通过它们和系统中的硬件进行交互。每个设备包含设备类型、设备号等属性，系统就是通过设备的主设备号和子设备号对硬件进行区别的，如：

```
# ls /dev/kmem -l
crw-r-----    1 root     root       1,   2 1970-01-01 08:30 /dev/kmem
```

最前面的 “c” 表示 kmem 是一个字符设备，接下来是它的权限位，日期前面的 “1, 2” 表示它的主设备号是 1，子设备号是 2。具体设备号对应于操作系统的哪些设备可以参考内核代码树中的 “Documentation/devices.txt” 文件。

设备文件使用 mknode 命令建立。如

```
# mknode /dev/kmem c 1 2
```

将在/dev 目录下建立 kmem 文件。

2. /etc 系统软件配置目录，这个目录中包含操作系统本身和大部分软件的配置文件。比较重要的有以下几个：

- passwd 包含用户的基本信息，如登录名、主目录等
- group 用户的组信息
- shadow 加密的用户密码文件
- fstab 系统中存在的文件系统设备和它们的挂载点
- inittab 系统的第一个进程 init 的配置文件，它决定了系统在不同运行级别分别要启动哪些进程。

3. /proc 与内核进行接口的虚拟文件系统挂载点，包含内核参数、进程参数等。
4. /bin 应用程序目录，至少包含基本的命令：sh，ls，cp，mv 等。
5. /sbin 系统程序目录。
6. /lib 动态和静态链接库目录。
7. /usr 用户数据目录，一些命令将放置到/usr/bin 和/usr/sbin 中。
8. /mnt 可以由用户挂载其他文件系统的挂载点。
9. /tmp 临时文件目录。

5.4.2 Linux 文件系统的建立

嵌入式 Linux 的文件系统经常由 BusyBox 构建，它被非常形象地称为嵌入式 Linux 系统中的“瑞士军刀”，因为它将许多常用的 UNIX 命令和工具结合到了一个单独的可执行程序中。虽然与相应的 GNU 工具比较起来，BusyBox 所提供的功能和参数略少，但在比较小的系统（例如启动盘）或者嵌入式系统中，已经足够了。

BusyBox 在设计上就充分考虑了硬件资源受限的特殊工作环境，它采用一种很巧妙的办法减少自己的体积：所有的命令都通过“插件”的方式集中到一个可执行文件中，在实际应用过程中通过不同的符号链接来确定到底要执行哪个操作。例如最终生成的可执行文件为 BusyBox，当我们为它建立一个符号链接 ls 的时候，就可以通过执行这个新命令实现列目录的功能。采用单一执行文件的方式最大限度地共享了程序代码，甚至连文件头、内存中的程序控制块等其他操作系统资源都共享了，对于资源比较紧张的系统来说，真是最合适不过了。

在 BusyBox 的编译过程中，可以非常方便地加减它的“插件”，最后的符号链接也可以由编译系统自动生成，下面我们就来一步步的用 BusyBox 从无到有的建立一个全新的 Linux 文件系统。

光盘中提供的是 2009 年 8 月 BusyBox 发布的 1.14.3 版本，压缩包大小为 2.1M 左右，最新的源代码可以从 http://www.busybox.net 下载。将源码解压之后，进入到 busybox-1.14.3 目录中，运行 make menuconfig 可以打开它的编译界面。这个界面和 Linux 内核编译有些接近，如图 5.7 所示。

BusyBox 所提供的默认设置是比较全面的，我们往往需要去掉一些不常使用的命令。图 5.7 中“—Applets”下面的配置内容用来选择 BusyBox 所支持的命令，上面的“BusyBox Settings —>”对 BusyBox 的基本编译选项进行配置。

默认情况 BusyBox 采用动态链接，但在嵌入式系统的调试初期，为了避免库文件对系统的启动产生影响，最好先采用静态链接，当确保库文件没有问题的时候再重新采用动态链接。这个设置选项是编译菜单中的“BusyBox Settings —> Build Options —> Build BusyBox as a static binary (no shared libs)”。

如果需要交叉编译，还需要设置交叉编译工具的命令前缀，即“BusyBox Settings —> Build Options —> Cross Compiler prefix”设置为

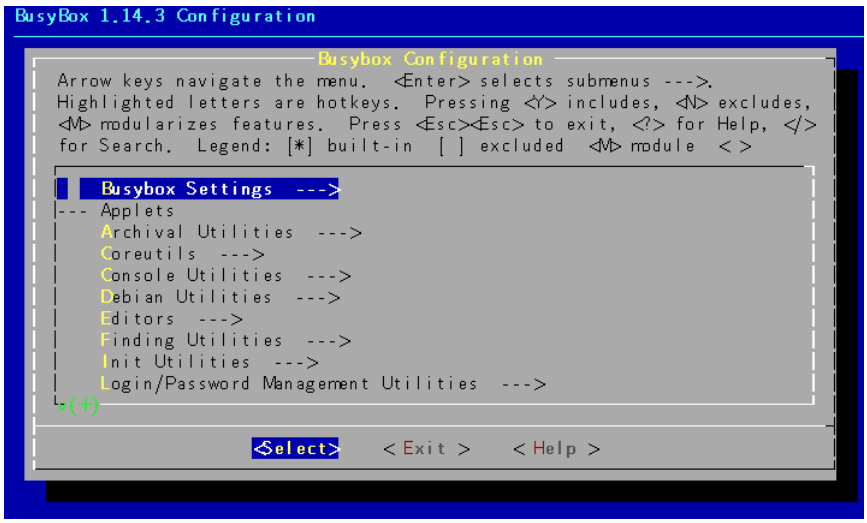

图 5.7: BusyBox 的编译配置界面

```
arm-none-linux-gnueabi-
```

然后执行 make，如果没有发生错误就会在当前目录中生成 busybox 文件，用 file 命令检查一下是否正确：

```
# file busybox
busybox: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux
2.4.18,  statically linked, stripped
```

最后运行 make install 命令，BusyBox 将在未来的根文件系统（默认是在 _install 目录）中建立/usr、/bin、/sbin 等目录。从中可以看到编译好的 BusyBox 可执行文件和其他应用命令的符号链接。典型的 busybox 文件大小在动态链接的情况下是 600KB 左右，静态链接为 800KB 左右，用它实现的文件系统可以控制在 2MB 左右。但就目前为止，我们得到的还不是一个完整可用的文件系统。必须要在这个基础上添加一些必要的文件，让它可以工作。

参考一个正常的 Linux 系统就会发现，BusyBox 建立的文件系统还缺少很多文件。下面三行命令建立了常见 UNIX 系统中包含的一些目录，虽然它们不全是必须的，但建立它们更符合标准一些。这些命令都是在新文件系统的根目录中执行的，第三条命令的执行还必须要有 root 权限。

```
# mkdir mnt root var tmp proc boot etc lib
# mkdir var/{lock,log,mail,run,spool}
# chown 0:0 -R *
```

如果 BusyBox 采用了动态链接的方式编译，还需要把 BusyBox 所需要的动态库：libcrypt.so.1、libc.so.6、ld-linux.so.2 放到 lib 目录中。最好按照标准的方式建立相应的文件和链接，可以参考下面的列表：

```
-rwxrwxrwx 1   92519  ld-2.3.2.so
lrwxrwxrwx 1      11  ld-linux.so.2 -> ld-2.3.2.so
-rwxrwxrwx 1 1190032  libc-2.3.2.so
lrwxrwxrwx 1      13  libc.so.6 -> libc-2.3.2.so
-rwxr-xr-x 1   18348  libcrypt-2.3.2.so
lrwxrwxrwx 1      17  libcrypt.so.1 -> libcrypt-2.3.2.so
```

etc 文件夹是许多系统配置文件保存的地方，这些文件非常重要，如果配置错误，就可能影响系统的启动。BusyBox 源代码 example/bootfloopy/etc 目录中的文件算是一个简单的例子，我们可以把其中的文件复制过来作为我们的基础。（在 example/bootfloopy 目录中的一些脚本和文档也很值得阅读）

首先 inittab 文件是系统启动后所访问的第一个脚本文件，后续启动的文件都由它指定。这个文件的格式和普通微机 Linux 上的 inittab 是有区别的，它不支持运行级别，每行命令的格式如下：

```
<id>:<runlevels>:<action>:<process>
```

其中 <runlevels> 部分被忽略。

<id> 是后面指定的 <process> 将要运行的终端名程，例如 tty0 表示第一个控制台，ttyS0 表示串口 1 终端，这个部分可以空白。

<action> 部分包含 sysinit,respawn,askfirst,wait,once, restart,ctrlaltdel 和 shutdown。init 程序首先执行 <action> 部分为 sysinit 的命令，然后是 wait 部分，等 wait 部分结束执行 once 部分。restart、ctrlaltdel 和 shutdown 都是系统进入相应状态才执行的部分。而如果 <action> 部分是 respawn 或 askfirst，指定的命令都会执行，而且是一旦它们退出就重新执行。askfirst 与 respawn 唯一的区别就是它在执行指定的命令之前会先询问一下。

一个简单的 inittab 的例子如下：

```
::sysinit:/etc/init.d/rcS
tty0::respawn:-/sbin/getty 38400 tty0
tty2::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/bin/mount / -o remount,ro
```

其中第一行指定了系统的启动脚本为/etc/init.d/rcS ；第二行指定在第一个虚拟终端打开一个登录会话；第三行指定在第三个虚拟终端打开一个无须登录验证的 shell ；第四行指定了当按下 “Ctrl+Alt+Del” 组合键时候的执行的命令；最后两行指定了关机时候执行的操作。

fstab 文件定义了文件系统的各个 “挂接点”，需要与实际的系统相配合，一个简单的 fstab 文件如下：

```
proc            /proc           proc    defaults        0       0
/dev/hda1       /               ext2    rw,noauto       0       1
devpts          /dev/pts        devpts  defaults        0       0
```

其中第三行是为 UNIX PTYs 准备的，telnetd 要用到。

profile 文件是终端登录之后首先运行的脚本，这里可以不去管它，我们比较关心的是系统在无人登录的情况下有哪些程序要执行。从 inittab 文件可以看出系统启动之后要运行/etc/init.d/rcS 这个脚本，我们需要启动的程序和需要进行的设置都可以写在这里面，一个可能的 rcS 文件如下：

```
#! /bin/sh
echo -e 'Starting System'

ifconfig lo 127.0.0.1
ifconfig eth0 192.168.0.100
```

```
hostname -F /etc/hostname

/bin/mount / -o remount,rw
/bin/mount /proc
/bin/mount /dev/pts

/usr/sbin/telnetd

dmesg > /var/log/dmesg
```

为了满足终端登录用户验证的要求，etc 目录下还需要有 passwd、group 和 shadow (在编译 BusyBox 时如果不选择 shadow 功能将不需要这个文件)，而这些文件至少要包含 root 用户的定义，如

```
passwd:
root:x:0:0:root:/root:/bin/sh

group:
root:x:0:

shadow:
root::12179:0:99999:7:::
```

所示，其中如果 shadow (对于不支持 shadow 的系统则是 passwd) 文件的第一个冒号和第二个冒号之间没有内容，表示这个用户登录不需要密码。如果需要设定密码或者增加新的登录用户，就可以参考开发主机上的相应文件，或者在目标系统启动之后用 passwd 命令和 adduser 命令完成。

最后还可以给目标机起一个名字，在/etc 目录下建立文件 hostname，将取好的名字写到里面。前面介绍的启动脚本 rcS 通过 hostname 命令把文件的内容设置为机器名。

文件系统的安装随着应用环境的不同差别比较大。在嵌入式环境中，一般只要通过特殊的打包工具将文件系统打包，并烧录到非易失性存储器中就可以了。例如对于 JFFS2 类型的文件系统就可以用 mkfs.jffs2 命令生成文件系统的映像。

mkfs.jffs2 命令可以把一个目录中的内容生成为 JFFS2 文件系统的映像。一般可以用这个命令把一个 NFS 的文件系统打包成 JFFS2 格式，并通过 boot loader 或其他工具写入到 Flash 中去。常用的 mkfs.jffs2 命令的参数如下：

- -r 用来指定输入的目录，默认情况下是当前目录。mkfs.jffs2 命令会将这个目录中的所有内容输出到目标文件中。
- -p 用来指定输入的文件系统映像大小。如果没有指定 -p 选项，输出文件的大小就只与输入目录中的文件总大小有关。如果指定较大的映像文件大小，不足的部分会用空数据进行填充，生成的文件系统会包含多余的空闲空间用来保存新的文件。
- -e 用来指定 Flash 设备的擦除块大小。这个参数的设置必须与具体的 Flash 硬件相关，否则生成的文件系统就会无法使用。这是由 Flash 设备只能按块来擦除所决定的，文件系统的组织必须符合 Flash 的擦除块尺寸。
- -s 用来指定保存数据的节点页尺寸。这个参数的默认值是 4KB，过小的页尺寸会使得生成的文件映像过大。
- -n 用来指定不添加清除标记。对于 NAND Flash，由于有自己的校验块，所以不需要使用清除标记。如果挂载文件系统的时候出现很多的类似 “CLEANMARKER node found at 0x00800000 has totlen 0xc != normal 0x0” 的内核警告消息，就是在生成文件系统的时候忘记了设置 -n 参数。

- -o 用来指定输出文件的名称。

例如可以使用如下命令为 NAND Flash 生成一个 JFFS2 格式的文件系统映像:

```
$ /usr/sbin/mkfs.jffs2 -r nfsroot/ -e 0x4000 -n -o root.jffs2
```

文件系统安装之后重新启动目标设备,应该就可以使用新建立的文件系统了。如果系统启动失败则可以通过内核消息查找错误的原因。如果错误信息提示不能正确的挂载文件系统,问题可能出在内核对文件系统的支持上。文件系统比较容易出问题的是用户的验证和动态链接库的加载。最简单的检测办法是把 busybox 编译为静态链接并设置内核参数 init=/bin/sh 来避开对用户登录的验证。

5.5 实验 :构造嵌入式 Linux 环境

实验目的

1. 了解嵌入式 Linux 开发环境
2. 掌握编译、裁减 Linux 内核的方法
3. 了解 Linux 内核文件的基本结构
4. 了解 Linux 文件系统结构

实验内容

一、建立编译环境,熟悉交叉编译工具

按照 5.4.2 节的说明交叉编译 BusyBox。

二、编译裁减内核

在工作目录下把内核解开,并应用提供的补丁。在 Linux 目录中依次执行下面的命令编译内核,命令后面括号中的内容是对命令的解释:

```
$ make pku_defconfig(选择一个预定义的配置文件 pku_defconfig 作为当前配置)
$ make oldconfig(对.config 文件进行分析,使之符合当前内核代码的配置情况)
$ make zImage(生成内核映像)
$ make modules(如果部分驱动被设置为模块,这里对它们进行编译)
```

如果编译成功,生成的内核将在 linux/arch/arm/boot/ 目录中。为了让生成的内核可以被 U-Boot 使用,还必须用 mkimage 命令对其进行 “打包” 成 U-Boot 可识别的映像。如果系统中不存在 mkimage 命令,可以在 Ubuntu 系统中运行如下命令安装

```
# aptitude install uboot-mkimage
```

mkimage 命令的使用方法如下:

```
mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name \
-d data_file[:data_file...] image
```

-A 设置体系结构为 “arch”

-O 设置操作系统为 “os”

-T 设置映像类型为 “type”

-C 设置压缩方式为 “comp”

-a 设置加载地址为 “addr” (16 进制)

-e 设置入口地址为 “ep” (16 进制)

-n 设置映像名称为 “name”

-d 使用的源数据文件为 “datafile”

-x 设置不加载到内存执行(execute in place)

我们在实验中使用的 mkimage 命令格式如下:

```
$ mkimage -A arm -O linux -C none -n linux-2.6 -a 0x20008000 -e \
> 0x20008040 -d arch/arm/boot/zImage uImage
```

-T 指定映像类型,可以取以下值:

standalone、kernel、ramdisk、multi、firmware、script、filesystem;

-C 指定映像压缩方式,可以取以下值:

none 不压缩;

gzip 用 gzip 的压缩方式;

bzip2 用 bzip2 的压缩方式;

-a 指定映像在内存中的加载地址,映像下载到内存中时,要按照用 mkimage 制作映像时这个参数所指定的地址值来下载;

-e 指定映像运行的入口点地址,这个地址就是 -a 参数指定的值加上 0x40(因为前面有个 mkimage 添加的 0x40 个字节的头)。

还可以使用 mkimage 命令显示映像头部信息:

```
mkimage -l image
```

把新生成的 uImage 复制到/var/lib/tftpboot 目录中就可以使用 bootloader 下载测试我们新编译的内核了。

Linux 下的超级终端程序是 minicom,第一次运行的时候可能需要通过 -s 参数对其进行设置。设置的参数可以参考第二章实验的内容。需要注意的是在 Linux 下,串口 1 的名称为/dev/ttyS0,串口 2 的名称是/dev/ttyS1。运行 minicom 程序连接到开发板,按 “Ctrl_A” 组合键之后按 “Z” 键,可以看到 minicom 的帮助菜单,例如此时按 “Q” 就可以退出 minicom,这个命令在打开帮组菜单之前可以用 “Ctrl_A” 然后按 “Q” 激活。

配置好 minicom 并看到 U-Boot 的提示符之后,需要注意 U-Boot 的环境变量设置是否正确。注意 serverip 和 ipaddr 的设置,只有 IP 地址设置正确才能通过 TFTP 从主机下载内核映像文件。

确定开发主机的 TFTP 正常工作之后,执行

```
U-boot> tftp 20008000 uImage
U-boot> bootm
```

运行加载的内核,察看运行结果,由于刚才并没有把加载的 uImage 文件写入到 flash 中,因此在下次启动的时候仍需要重新加载内核到开发板。因为采用网络加载内核的方式很迅速,所以在开发阶段没有必要把内核烧入到 Flash 中。

如果修改了内核的配置,需要对内核重新进行编译。如果编译过程发生错误,你可能需要重新配置内核或者执行 make clean 删除编译过程生成的临时文件,甚至执行 make mrproper 来恢复代码树的异常。如果这两个方法都不能解决问题,则最好从头开始这个实验,即从解压内核代码、给内核应用补丁开始。

三、构造嵌入式 Linux 网络文件系统

用 root 用户解压光盘中提供的文件系统压缩包:

```
# tar xfj arm_root_v3.tar.bz2
```

这个压缩包里包含完整的文件系统,可以看到 etc、bin、dev 等目录。由于 dev 目录中的设备文件属于特殊文件,只能由 root 用户创建,因此执行解压命令不能使用没有 root 权限的用户。否则在解压的过程中就会出现如下的错误信息:

```
tar: armroot/dev/video0: Cannot mknod: Operation not permitted
tar: Error exit delayed from previous errors
```

如果在 tar 命令中使用了 v 参数,第一行的错误可能由于其他显示信息的不断出现而不被发觉。但第二行错误信息是在命令的最后输出的,提示用户前面出现过错误。因此时刻注意命令的提示信息是非常重要的,否则这个隐藏的错误就会影响后面的实验结果。

为了让系统可以在网络根文件系统上启动,还必须设置内核的启动参数(Default kernel command string)。其中假定嵌入式文件系统的路径为 "/home/embedded/armroot/"。

```
U-boot> setenv bootargs root=/dev/nfs rw nfsroot=192.168.0.101:\
/home/embedded/armroot/ ip=192.168.0.100::192.168.0.1:255.255.255.0:\
embedarm:eth0:off console=ttyS0,115200
```

这行参数包含了五个内核参数的设置。"root=/dev/nfs" 指定了根设备为/dev/nfs,"nfsroot=…" 的部分指定了网络文件系统根目录的挂载点是 IP 为 192.168.0.101(服务器 IP)主机上面的/home/embedded/newroot/ 目录,本机的 TCP/IP 地址等相关设置是由 "ip=…" 部分设定,具体设定内容是 192.168.0.100,网关 192.168.0.1,子网掩码 255.255.255.0,主机名 embedarm,采用 eth0 设备,"console = ttyS0,115200" 设定终端采用 ttyS0,波特率为 115200。在 IP 地址的设置中,要考虑到和 bootloader 中的设置统一,以避免麻烦。

重新编译内核之后把新生成的 uImage 复制到/var/lib/tftpboot 目录,用 bootloader 加载到开发板,运行测试。在执行内核前应先确定 NFS 服务正确运行,指定的 root 路径中有正确的文件系统。正常的话应该可以看到在 minicom 终端上打印出 login 的提示符。/home/embedded/newroot/etc/目录中的 passwd 文件决定了哪些用户可以登录,及他们的密码是多少。示例文件系统中提供 passwd 文件中的 root 用户的密码为空。

文件系统的内容可以完全使用 BusyBox 一点一点的完成,不过工作量比较大,感兴趣的读者可以进行尝试。

采用网络文件系统的好处主要有以下几点:

- 便于文件系统修改。由于嵌入式的文件系统在主机上,修改、添加或删除文件系统的内容都可以用主机的工具进行。
- 减少 Flash 的操作次数,提高工作效率。
- 可以构建比较大的文件系统进行测试,文件系统的尺寸不受 Flash 的限制。

网络文件系统的正常使用是后续嵌入式开发实验的前提,如果不能正常挂载可以从下面几点考虑:

1. 内核参数的设置是否正确。可以通过内核打印的消息来检查。
2. 网络 IP 地址是否设定正确。检查服务器的 IP 地址(ifconfig 命令)
3. 网络是否正确连接。在服务器端用 ping 命令测试是否和嵌入式系统连接。
4. 服务器的 NFS 服务是否正常。在服务器端进行 mount 测试,检查/etc/exports 文件的目录和权限设置。

5. 检查开发主机/var/log/daemon.log 文件，这个文件为服务程序的日志，可以通过它分析发生错误的原因。

第六章　Linux 环境程序设计

在前面的章节中，本书对 Linux 操作系统系统的使用和基本概念进行了初步的介绍，读者可以在 Linux 操作系统的环境下使用各种嵌入式开发工具。从本章开始，介绍的内容转向 Linux 平台程序设计，也就是编写运行在 Linux 操作系统下的应用程序。第三章和第四章中编写的程序虽然也在 Linux 环境下开发，但它们在运行的时候并不需要开发板上运行 Linux 操作系统，是完全运行在硬件平台的程序。本章所介绍的程序是运行在 Linux 操作系统环境中的，多数情况下既可以运行在开发平台的 Linux 操作系统中也可以运行在开发板的 Linux 操作系统中。因此，本章所介绍程序的运行要在第五章成功地把 Linux 操作系统运行在开发板的基础之上。

6.1　shell 脚本编程

6.1.1　脚本编程简介

脚本（script）程序与普通程序的最大区别就是它的执行方式，脚本程序不需要经过编译器编译，而直接由解释器解释执行。解释执行的方式注定了脚本的执行效率没有编译执行的程序高。不过脚本语言一般语法结构比较简单，便于学习，采用脚本编程往往可以获得比较高的编程效率，节省人工成本。

脚本语言特别适合在系统维护中使用。因为它学习起来比较容易，不需要拥有很多的程序设计知识，也不需要很多复杂的编译调试工具。很多稍微复杂的系统维护任务都可以通过编写脚本的方式来完成，把需要多次重复的任务变成了一个简单的脚本程序。别人编写的脚本程序也很容易被其他脚本调用，久而久之就会形成非常多的工具集。许多脚本程序都是通过许多人的不断完善而变得非常强大，用户群也越来越多，常见的脚本语言有 Perl、Python、Tcl、shell 等。

Perl 脚本最初为在 UNIX 系统中更好地处理文本报表而设计，经过多年的发展已经成为一种被广泛应用的高级脚本语言。Perl 脚本对文本文件的处理功能依然非常强大，在图像处理、系统管理、网络编程、数据库编程等方面也有其用武之地。

Python 是最近几年发展非常快的一种脚本编程语言，它的设计理念非常重视代码的可读性，其最显著的语法特点是使用缩进作为代码块的分隔符。采用缩进来区分代码块可以强制程序员很好地格式化程序代码，这也是 Python 保证可读性的一种方式。Python 与 Perl 同样具有非常广大的开源社区支持，支持不同应用的代码库非常多，应用范围很广，有逐步取代 Perl 的趋势。

Tcl 同样是一种通用的脚本语言，它经常被嵌入到其他应用程序中用来给软件提供一定的编程能力。一般意义上的 Tcl 脚本经常用于图形界面程序的设计、软硬件的测试、网络编程等方面。另外由于 Tcl 的某些版本比较小巧，也可以被应用到嵌入式系统环境中。

shell 是一种比较特殊的脚本工具，因为它同时也是用户输入命令的解释器，因此在 shell 脚本中可以使用所有可以在命令行使用的 Linux 工具。反过来，在命令行也可以使用所有的 shell 编程语法元素。但为了使脚本获得更好的移植性，一般在 shell 脚本中只使用 UNIX 系统中的标准命令行工具。由于目前 Linux 系统中的标准 shell 是 bash，因此本小节主要介绍 bash 脚本。

Linux 下的脚本程序一般是一个可执行的文本文件（具有执行权限），我们在之前的章节中也看到了很多 shell 脚本的实例。脚本文件一般以 # 符号作为文件中的注释，用于说明文件信

息或者对程序进行注释。但脚本的第一行是特殊的，它以 “#!” 开始，感叹号后面的内容是脚本的解释程序路径和执行参数。例如下面是一个 Perl 脚本，其解释程序为 perl：

```
#!/usr/bin/perl -w
print "Hello, World\n"
```

6.1.2 shell 脚本介绍

1. shell 变量

shell 变量实际上就是 shell 的环境变量，前面章节已经介绍很多，这里只介绍一些特殊的变量。

当向 shell 脚本传递命令行参数的时候，可以在脚本中通过 $0，$1，...，$9 来访问。其中 $0 是脚本的名称，$1 是第一个命令行参数，$2 是第二个命令行参数，以此类推。另外 $# 代表传递到脚本的参数个数，$* 代表全部的命令行参数，$@ 变量也是代表全部命令行参数，只不过每个参数用引号分别引用。

另外 $$ 代表脚本运行的当前进程 ID 号，$! 代表后台运行的最后一个进程的进程 ID 号，$- 显示 shell 使用的当前选项，可以由 set 命令设置，$? 显示最后一个命令的退出参数，一般 0 代表正常退出。

在脚本中还可以通过 read 命令从用户的输入获得变量的值，例如下面的例子中，第一个 “Test” 就是用户输入的：

```
$ read INPUT
Test
$ echo $INPUT
Test
```

2. 条件测试

在程序设计中，条件测试经常在程序分支或者循环处获得应用。shell 脚本允许对数字、字符串甚至文件进行测试。条件测试的格式有两种，一种是使用 test 命令：

```
test condition
```

另外一种是使用 “[” 命令[1]，“[” 命令与 “]” 符合进行配对使用的时候可以使得程序的可读性更好。只是要注意在 “[” 之后要加入空格，否则 shell 无法把 “[” 作为单独的命令。

```
[ condition ]
```

对于文件测试，可以使用如表 6.1 的测试参数：

表 6.1: 文件测试参数

参数	描述	参数	描述
-d	为目录	-f	为正规文件
-L	为符号链接	-r	文件可读
-s	文件非空	-w	文件可写
-x	文件可执行	-u	文件具有 suid 位

例如

```
$ test -d /usr/
$ echo $?
0
```

[1]在/usr/bin/ 目录下的确存在文件名为 “[” 的命令

对于字符串测试，可以通过表 6.2 所列参数测试字符串的特性：

表 6.2: 字符串测试参数

参数	描述
=	两个字符串相等
!=	两个字符串不同
-z	字符串为空
-n	字符串非空

例如（注意参数须用空格分割）：

```
$ [ $EDITOR = "vi" ]
$ echo $?
$ 0
```

数值测试的时候，数字需要用引号引用，参数见表 6.3，用法与字符串类似：数值测试的时候，可以

表 6.3: 数值测试参数

参数	描述	参数	描述
-eq	两数字相等	-ne	两数字不等
-gt	前者大于后者	-lt	前者小于后者
-ge	前者大于或者等于后者	-le	前者小于或者等于后者

用 expr 命令进行简单的数值运算，计算的时候可以使用 +、-、*、 / 四种符号。注意由于 * 号在 shell 中的特殊含义，在使用的时候需要用反斜线转义。另外在符号与操作数之间也需要用空格分隔，例如：

```
$ expr 10 \* 2 + 8
28
```

如果多个条件之间要进行逻辑运算，可以用 -a 表示逻辑与，-o 表示逻辑或，! 表示逻辑非。例如：

```
$ [ -f /bin/sh -a "10" -lt "8" ]
$ echo $?
1
```

3. 程序分支控制

几乎所有的程序语言中，都用 if 语句作为分支控制，shell 编程也不例外，它的分支控制语法如下：

```
if 条件 1 then 命令 1
elif 条件 2 then 命令 2
else 命令 3
fi
```

if 语句必须以单词 fi 终止，即把 if 单词倒序写出。这个规律在 shell 编程中有很多例子，正序与倒序写出的单词成对的标记一个语法单元，就像 C 语言中的大括号一样，必须成对出现。

elif 和 else 为可选项，elif 是 else if 的缩写，含义是如果前面的条件不成立，那么进行新的条件测试。一个 if 语句中可以包含多个 elif 部分，进行多个条件的测试。最后一个条件分支就是 else 部分，包含所有的条件都不满足时执行的语句。例如下面的一个脚本示例：

```
#!/bin/sh
if [ -d /home/one ]
```

```
then
  echo "One is here"
else
  echo "One's not here"
fi
```

如果需要对一个变量进行多次条件测试，除了使用 elif 语句之外，还可以使用 case 语句，case 语句的语法格式如下：

```
case 值 in
模式 1)
   命令 1
   ;;
模式 2)
   命令 2
   ;;
*)
   命令 n
   ;;
esac
```

case 语句以逆序的 case 单词结尾，其间包括多个用来匹配测试值的模式，它会测试每个模式的匹配情况，一旦模式符合，其所在分支的命令就会开始执行，直到执行至两个分号，其他模式不再进行测试。* 号可以用来匹配一切模式，所以在 case 语句中往往会在最后一个模式中使用星号来匹配前面漏下的模式。参见下面的示例：

```
#!/bin/sh
echo -n "Are you sure? [y/n]"
read ANS
case $ANS in
  y|Y|yes|Yes) echo "Go on"
  ;;
  n|N|no|No) echo "Bye"
  exit 0
  ;;
  *) echo "Wrong selection"
  exit 1;
  ;;
esac
```

4. 循环控制

shell 支持 for 循环、until 循环和 while 循环三种循环控制格式。首先来看一下 for 循环的语法结构：

```
for 变量 in 列表
do
  命令
done
```

for 语句会对列表中的每一个变量值执行一次在 do 与 done 之间的命令块，例如：

```
$ for i in `ls`
> do
> echo $i
> done
1.txt
2.txt
```

这个例子是在命令行输入的，由于第一行没有把命令输入完整，shell 给出了新的提示符。其实这个命令可以输入在一行，不过需要几个分号进行分隔：

```
$ for i in `ls`; do echo $i; done
```

这条命令把 ls 命令的输出，即当前目录下的所有文件，作为循环的列表，在循环体中将每个文件名显示出来。for 循环的这种用法在对多个文件处理类似操作的时候非常有用。

until 循环的语法结构如下：

```
until 条件
do
   命令
done
```

循环体内的命令不断重复执行，直到条件测试为真的时候停止。

while 循环的语法结构如下：

```
while 条件
do
   命令
done
```

与 until 循环的语义类似，只不过循环在条件测试为假的时候停止。

在循环控制语句中，还有两个非常常用的命令。一个是 break，用来从循环体中跳出。另外一个是 continue，用来结束本次循环，跳转到循环体的第一条语句开始下次循环。

5. 函数

函数在程序设计中是一种重要的代码重用方式。如果一段代码会在程序中多次使用，最好的解决方案就是把这段代码设计成一个函数，然后在程序中对其进行多次调用。

shell 中定义函数的方法如下，其中 function 关键字可以被省略：

```
function 函数名 ()
{
命令
}
```

调用函数与调用另外一个脚本的主要区别是函数与被调用者在同一个 shell 环境中执行，而调用脚本的时候会创建一个独立的环境运行脚本。

如果函数需要传递参数，需要使用特殊的 shell 变量 $1...$9。例如

```
#!/bin/bash
function hello
{
          echo “Hello $1!”
}
hello Alan
hello Bill
```

函数可以用 return 命令设定一个返回值。与 shell 中的习惯一致，0 代表正确返回，1 代表函数发生异常。

6.2 Linux 环境程序基础

回顾第五章对操作系统的介绍，操作系统提供给应用程序很多的系统调用可以方便程序的设计。例如打开文件、读入文件内容、把数据写入文件、运行一个新程序、分配更多的内存、获得

当前的时间，等等。Linux 操作系统作为 POSIX 兼容的类 UNIX 操作系统，在系统调用方面与 Windows 操作系统有很多的区别，因此在编程上有很多自己的特点。不过由于 Linux 支持更多的硬件平台，其系统调用也与其他 UNIX 操作系统非常接近，因此在 Linux 环境下编写的程序具有更好的可移植性。

嵌入式 Linux 内核从源代码的层次上同普通的 Linux 内核并没有区别，因此为嵌入式 Linux 编写程序与普通的 Linux 程序设计并没有本质的不同。只是由于嵌入式系统的特点，嵌入式 Linux 编程和调试一般在开发主机上进行，并通过交叉编译的方式生成目标平台可执行代码，下载到目标平台运行。

同样是由于嵌入式系统的特点，嵌入式 Linux 平台的编程以 C 语言为最常用的编程语言。C 语言具有编译效率高、学习门槛低、用户群体大等特点，甚至连 Linux 内核也都主要是由 C 语言编写的。几乎所有的 Linux 操作系统都已经包含了完整的 C 库，提供了很多库函数支持，因此在嵌入式系统中使用 C 语言进行开发不用增加额外的库文件，程序的移植性也很好。

6.2.1 文件处理

C 语言中的文件处理函数主要有两类，一类是 C 语言教程中介绍比较多的标准 I/O 库函数，即围绕 FILE 数据类型的文件操作函数。这类函数会对文件的访问进行缓冲，以提高文件处理的效率。另外一类是无缓冲的文件处理函数，这类函数在操作特殊文件或者需要自己处理文件缓冲区的时候使用。本节只对后一类文件操作函数进行介绍，因为它们在嵌入式的环境中用的比较多。

对 Linux 内核来说，所有打开的文件都通过文件描述符来索引。每个文件描述符都是一个非负的整数，当一个进程打开一个文件的时候，内核为这个文件分配一个文件描述符，这个描述符相对这个进程是唯一的。在第二章介绍 shell 的时候，我们介绍过当 shell 执行一个新进程的时候，会自动给这个进程打开三个文件，其中 0 号描述符对应标准输入设备，1 号描述符对应标准输出设备，而 2 号描述符对应标准错误输出。

打开文件的操作由 open 函数实现，open 函数的说明如下：

```
#include <fcntl.h>
int open(const char *pathname,int oflag, ...)
```

函数返回值即为所打开文件的文件描述符，如果返回值小于 0 则表示发生了错误。使用这个函数的时候必须要检查是否发生错误，并对可能的错误进行处理。open 函数的参数 pathname 为要操作的文件名，而文件的打开方式由 oflag 参数决定，这个参数的取值必须为以下三者之一：

- O_RDONLY 以只读方式打开
- O_WRONLY 以只写方式打开
- O_RDWR 以读写方式打开

除了上面三个参数之外，open 函数还接受一些附加参数，这些参数以 OR(|)的方式与上面三个参数进行运算作为 open 函数的第二个参数(参考后面的例子)。完整的参数列表可以通过

```
$ man 2 open
```

命令来获得，这里仅介绍最常用的几个参数：

O_APPEND 以追加的方式写入文件，即打开文件时文件指针移到文件最后；

O_CREAT 如果文件不存在，则建立新的文件。这个参数需要 open 函数使用第三个参数 mode 来指定新文件的访问权限。关于 mode 参数，后面还有介绍；

O_EXCL 当与 O_CREAT 同时使用时，如果文件已经存在，则 open 函数返回错误。对文件是否存在的检查和建立新文件是一个“原子操作”，不会引起同步错误；

O_TRUNC 当打开的文件具有写入权限的时候，把文件的长度置为 0。

如果调用 open 的目的是建立新的文件，还可以直接使用 creat 函数：

```
#include <fcntl.h>
int creat(const char *pathname,mode_t mode);
```

creat 函数等效于调用：

```
open (pathname,O_WRONLY | O_CREAT | O_TRUNC, mode);
```

关闭文件的函数为 close，其函数说明如下：

```
#include <unistd.h>
int close(int filedes);
```

其输入参数为文件描述符，正常情况下返回值为 0。

每个打开的文件都有一个相关的文件指针，表示当前文件的读写位置。文件指针一般由一个非负整数表示，代表相对于文件起始的偏移位置。除非使用了 O_APPEND 标志，用 open 函数打开文件的文件指针都是 0，表示从文件的开始进行操作。如果要改变文件指针的位置，可以使用 lseek 函数：

```
#include <unistd.h>
off_t lseek(int filedes,off_t offset,int whence);
```

lseek 函数的 filedes 参数为文件的描述符，offset 参数表示文件的偏移指针，这个参数的解释由 whence 参数决定。whence 参数的取值可以有下面三种：

- SEEK_SET 表示偏移指针相对于文件的开始计算
- SEEK_CUR 表示偏移指针相对于当前位置计算
- SEEK_END 表示偏移指针相对于文件的末尾计算

lseek 函数的返回值为新的文件指针，所以可以通过下面的调用得到当前的文件指针位置：

```
currpos = lseek(fd,0,SEEK_CUR);
```

从文件中读取数据用 read 函数：

```
#include <unistd.h>
ssize_t read(int filedes,void *buf,size_t nbytes);
```

read 函数的参数 filedes 为文件描述符，buf 为读取数据的存储地址，nbytes 为希望读取数据的字节数。其返回值为实际读取到数据的字节数，例如当达到文件末尾的时候实际读出的数据字节数就可能小于期望的值。如果当前的文件指针已经是文件末尾，read 函数的返回值就会为 0。read 函数总是从文件指针的位置开始读取数据，并在读取结束后把文件指针移动到本次读取数据最后一个字节的后面。

把数据写入到文件用 write 函数：

```
#include <unistd.h>
ssize_t write(int filedes,const void *buf,size_t nbytes);
```

write 函数的参数 filedes 为文件描述符，buf 为要写入数据的存储地址，nbytes 为希望写入的字节数。一般情况下 write 函数的返回值与 nbytes 相同，但在磁盘空间已满等特殊情况下，write 函数的返回值会小于 nbytes。

对于一些特殊的文件，例如设备文件，还经常使用 ioctl 函数：

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

ioctl 的第一个参数为文件描述符，第二个参数是一个与具体设备相关的请求代码，第三个参数并不是必需的，一般会是一个指针，用来作为命令请求的具体参数。ioctl 的返回值为 0 的时候表示一切正常，返回值为 -1 表示出现了错误，具体的错误编码由 errno 指定。

下面一个例子展示了无缓冲文件处理部分函数的最基本用法。

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

main()
{
        int fd,i;

        fd = open("/dev/dleds", O_RDWR);
        if (fd > 0) {
                i = 3;
                if(write(fd, &i, 4)==-1)
                        printf("%s\n",strerror(errno));
                close(fd);
        }
}
```

前面在 open 函数中提到建立文件的 mode 参数，主要是用来给新建的文件设置访问权限。如同在 chmod 命令中接收的九种访问权限，可选的 mode 参数也有九个，含义也完全相同，它们可以用 “|” 任意组合成需要的访问权限。

- S_IRUSR 用户读权限
- S_IWUSR 用户写权限
- S_IXUSR 用户执行权限
- S_IRGRP 组用户读权限
- S_IWGRP 组用户写权限
- S_IXGRP 组用户执行权限
- S_IROTH 其他用户读权限
- S_IWOTH 其他用户写权限
- S_IXOTH 其他用户执行权限

实际建立文件的访问权限还与当前进程的 umask 设置相关，umask 的设置可以通过 umask 函数来修改，它仅有一个 mode 参数，与 open 的 mode 参数含义相同，但其含义是去掉新文件的相应权限位。例如 umask 包含 S_IROTH，则建立的文件就不会包含这个权限位。umask 的定义如下：

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

umask 的返回值是之前的 umask 设置。在 shell 中，可以通过 umask 命令来设置默认的 umask 组合。

如果要检查对某个文件是否具有特定的访问权限，可以使用 access 函数。

```
#include <unistd.h>
int access(const char *pathname,int mode);
```

access 函数的 mode 参数指定了需要检查的访问权限，其可选如下四种标志的任意组合：

- R_OK 具有读权限
- W_OK 具有写权限
- X_OK 具有执行权限
- F_OK 文件是否存在

access 如果返回 0 则表示具有特定的访问权限，否则表示不具有这个权限。

6.2.2 进程相关

1. 进程的输入参数

在使用 shell 运行程序的时候，可以在命令行指定一个或多个程序运行所需要的参数，这个功能在 C 语言中是通过 main 函数的函数参数的方式来实现的。main 函数的原型定义如下：

```
int main(int argc,char *argv[]);
```

其中 argc 为命令行参数的个数，而 argv 为一个字符串数组，包含了命令行的全部内容。下面的示例程序展示了这种用法。

```
int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < argc;  i++)     /* 显示全部的命令行参数 */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

通过运行这个程序可以看出，argv[0] 就是这个可执行程序的名字，真正的命令行参数部分是从 argv[1] 开始的。如果在运行程序的时候没有指定任何参数，则 argc 的值为 1，argv[1] 的值为 NULL。在使用命令行参数的时候，需要尽量符合 GNU 软件的习惯，例如 `-h` 就是显示帮助信息，`-o` 就是更改输出文件的名字等。

除了命令行参数，当前的环境变量也可能影响程序的运行。获得特定环境变量设置的函数为 getenv，其定义如下：

```
#include <stdlib.h>
char *getenv(const char *name);
```

getenv 的 name 参数为环境变量的名字，返回值为指向环境变量的设置值的指针。如果没有找到指定的环境变量，则返回 NULL。如果一个程序的行为可以由配置文件、环境变量和命令行参数来改变，这三者的优先级从低到高应该是配置文件最低，而命令行参数最大。

标准的 C 语言程序还要提供一个整数类型的返回值，也就是 main 函数的函数返回值。一般情况下返回 0 表示程序正常结束，否则表示程序发生了异常退出。在 Bash 中，可以用 $? 来获得上一个程序的返回值。例如：

```
$ ls
main  main.c
$ echo $?
0
```

2. 进程的权限信息

我们已经知道，每个进程都有一个唯一的 ID 与之对应，程序中可以通过 getpid 函数获得进程自己的 ID。getpid 和其他一些获得进程信息的函数定义如下：

```
#include <unistd.h>
pid_t getpid(void);          获得调用进程的进程 ID
pid_t getppid(void);         获得调用进程的父进程 ID
uid_t getuid(void);          获得调用进程的真实用户 ID
uid_t geteuid(void);         获得调用进程的有效用户 ID
gid_t getgid(void);          获得调用进程的真实组 ID
gid_t getegid(void);         获得调用进程的有效组 ID
```

所谓的真实用户 ID 和组 ID 就是我们实际的登录用户名与所在的组，而有效用户 ID 和组 ID 多数情况下都与真实用户 ID 和组 ID 相同。当一个可执行程序的 set-uid 位被设置并且在 shell 中运行这个程序的时候，这个执行进程的有效用户 ID 就是这个可执行程序的所有者。这个功能是在为了访问特定的文件而必须提高用户的权限的时候才需要的。当一个进程的有效 ID 和真实 ID 不同的时候，可以通过下面的函数改变自己的有效 ID。

```
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

对于 root 用户来说，通过 setuid 命令可以指定任意的 uid，而一般的用户只能在真实 ID 和有效 ID 之间进行切换。

3. 建立新进程

进程可以通过调用 fork 函数来建立新的进程。fork 函数的定义如下：

```
#include <unistd.h>
pid_t fork(void);
```

可以看出 fork 函数的定义非常简单，但真正理解它却有一定的难度。一个进程在调用了 fork 函数之后变成了两个进程。其中新建立的进程为子进程，对它来说 fork 函数返回值为 0。原来的进程为父进程，对它来说 fork 函数的返回值为子进程的 ID 号。

对于子进程和父进程来说，它们都会继续从 fork 函数的位置执行下去，它们具有同样的代码，区别仅仅是 fork 的返回值。一般情况下，子进程会调用 exec 类的函数执行新的程序，直到此时，子进程才变成了与父进程完全不同的进程。下面的代码示例是一个典型的 fork 函数的使用方式。

```
if ((pid = fork()) < 0) {
        printf("fork error");
        exit(0);
} else if (pid == 0) {       /* 子进程 */
        ...;                 /* 子进程执行代码 */
```

```
} else {
        ....;                   /* 父进程执行代码 */
        wait(&status);
}
```

在执行完 fork 函数之后，父子进程拥有同样的代码、数据和堆栈。但它们的数据采用“写复制”的方式共享。也就是说对于一个全局变量，如果父子进程都没有对它进行修改，则它们访问内存空间是一致的。如果其中一个进程修改了这个全局变量，父子进程所访问到的内容就不同了。

需要注意的是对于父进程已经打开的文件描述符，在执行 fork 函数之后是与子进程共享的，并且拥有共同的文件指针。因此如果不经过同步处理，父子进程同时写入同一个文件描述符会造成文件内容的混乱。

当子进程结束的时候，内核会给父进程发送消息，报告子进程的终止状态。如果父进程在子进程终止之前结束，则 init 进程成为子进程的新“父亲”。为了获得子进程的终止状态，父进程必须调用 wait 函数。wait 系列函数有好几种，最简单的定义如下：

```
#include <sys/wait.h>
pid_t wait(int *statloc);
```

其中 statloc 的值在 wait 返回后为子进程的返回状态，wait 的返回值为子进程的 ID 号。调用 wait 函数将可能阻塞，直到子进程终止。如果子进程在终止的时候，其父进程没有调用 wait 函数，则子进程变为“僵尸”（Zombie）进程，在 ps 命令中状态显示为“Z”。

4. 执行新程序

仅仅通过 fork 函数只能对原有的进程进行复制，而不能执行新的程序代码。实际上 Linux 操作系统中运行的所有的进程都是通过 fork 函数复制，然后再通过 exec 函数载入新程序的。exec 函数一共有六种形式，

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

这六种形式主要有三个区别：

- 前四种形式的第一个参数以 pathname 作为参数，即包含路径信息。而后两种形式只使用文件名，通过 PATH 环境变量来找到具体可执行文件的位置。
- execl、execlp、execle 采用参数列表的方式接收命令行参数（“l” 字母代表 list），即每个命令行参数作为单独的函数参数，最后用空字符串作为结束。而另外三种 exec 函数采用向量方式接收命令行参数（“v” 字母代表 vector），即所有命令行参数作为字符串数组进行参数传递。
- execle、execve（最后的 “e” 代表 environment）把环境变量作为字符串数组传递给新程序。

下面是一个 exec 函数的示例。

```
char  *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };  /* 环境变量 */
if ((pid = fork()) < 0) {
        err_sys("fork error");
} else if (pid == 0) {  /* 子进程 */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
            "MY ARG2", (char *)0, env_init) < 0)
        err_sys("execle error");
}
```

6.2.3 信号

信号(signal)是 UNIX 操作系统中非常重要的概念,它为应用程序提供了一种处理异步事件的方法。例如用户在程序运行过程中按下了 “Ctrl+C” 组合键终止程序运行,这个事件对于运行的程序来说,是异步发生的,类似于硬件的中断。因此应用程序并不知道事件何时发生,也不必不断查询特定的事件是否发生,可以仅仅告诉操作系统:当特定时间发生的时候,应该如何去处理。

一些常见的信号如下:

SIGABRT:这个信号在程序调用 abort 函数的时候发生,默认动作是终止程序。

SIGALRM:这个信号在定时器超时的时候发生,定时器可以通过 alarm 函数设置。

SIGBUG:特定的硬件错误或者内存访问错误会产生这个信号。

SIGCHLD:当一个进程结束运行的时候,它会给自己的父进程发送 SIGCHLD 信号。如果父进程需要得到子进程的退出状态,就必须处理这个信号,并用 wait 函数得到。

SIGINT:这个信号在用户按下 “Ctrl+C” 组合键,试图终止程序的时候产生。

SIGKILL:这个信号用来终止程序的运行,是不可以被应用程序截获的。

SIGPIPE:当一个进程向一个管道(pipe)写入数据时,如果管道的另外一端已经关闭,则产生这个信号。当向一个类型为 SOCK_STREAM 的已经断开连接的 socket 写入数据的时候也会发生这个信号。

SIGQUIT:当用户按下退出键时(一般是 “Ctrl+’ 组合键)发生这个信号给全部的前台进程。

SIGSEGV:当程序访问了非法的内存段的时候发生这个信号。

SIGTERM:kill 程序默认会发送这个信号给其他进程用来终止程序的运行。

如果程序希望对特定的信号进行处理,可以使用 signal 函数:

```
#include <signal.h>
void (*signal(int signo,void (*func)(int)))(int);
```

这个函数的定义比较不常见,因为它使用了两个函数指针。首先,signal 函数的返回值是参数为 int 的函数指针,其第一个参数为信号代码,第二个参数也是一个参数为 int 的函数指针。上面所说的函数指针除了可以是真正的函数指针外,还可以选择为 SIG_IGN(表示忽略该信号,SIGKILL 和 SIGSTOP 不能被忽略)和 SIG_DFL(表示使用默认的信号处理办法)。

signal 函数的含义就是为 signo 信号设置处理函数为 func,其返回值是该信号之前的处理函数指针,如果发生异常,signal 函数的返回值为 SIG_ERR。

下面是一个例子:

```
if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        printf("signal(SIGALRM) error\n");
    alarm(10);
    if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        printf("read error\n");
    alarm(0);
    write(STDOUT_FILENO, line, n);
    ……

static void sig_alrm(int signo)
{
    write(STDOUT_FILENO, "ALARM, Be quick\n", 16);
    return;
}
```

注意到 sig_alrm 处理函数的参数实际上为所处理的信号编码。因此可以通过一个处理函数处理多个信号事件。

如果要给某个进程发送信号，Linux 提供了 kill 与 raise 函数，其中 raise 只能给自己发送信号。它们的定义如下：

```
#include <signal.h>
int kill(pid_t pid,int signo);
int raise(int signo);
```

对于 kill 函数来说，pid 参数有如下几种选择：

- pid>0 把信号发送给进程 ID 为 pid 的进程
- pid=0 把信号发送给组 ID 与自己相同的所有进程，不过要排除特定的系统进程和没有权限的进程。
- pid<0 把信号发送给组 ID 为 pid 绝对值的所有进程，不过要排除掉没有权限的进程。
- pid=1 把信号发送给系统中所有的进程，同样需要排除特定的进程。

所谓具有发送信号的权限是指超级用户可以给任何进程发送信号，对于一般用户发送信号的真实用户 ID 或者有效用户 ID 必须等于接收信号进程的真实用户 ID 或者有效用户 ID。

6.3 Linux 下的串口编程

串口在 Linux 系统中被当作一种终端接口，即 tty 接口，在系统中与其他终端接口使用同样的控制机制，属于 POSIX.1 定义的标准接口，称为 termios。从程序员的角度来看，termios 是一个数据结构和一些控制这些数据结构的函数，这个数据结构包含了对终端各种特性的完整描述（比如串口的波特率、停止位、奇偶校验等），而这些函数就可以理解为用以操控、改变这些特性。下边列出 termios 的数据结构：

```
struct termios{
        tcflag_t c_iflag; //输入状态标志
        tcflag_t c_oflag; //输出状态标志
        tcflag_t c_cflag; //控制状态标志
        tcflag_t c_lflag; //本地状态标志
        cc_t c_line;      //行参数
        cc_t c_cc[NCCS]   //控制字
};
```

串口同时作为 I/O 设备，在 Linux 操作系统中被当作文件来处理，因此使用相同的系统调用如 write( )、read( )、open( )。下面例程的开始，首先打开串口，然后填充 termios 数据结构，并使之生效，最后通过串口文件读写数据实现通信。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

#define BAUDRATE B115200        //波特率 115200
#define MODEMDEVICE "/dev/ttyS0"        //串口的设备名
#define _POSIX_SOURCE 1
```

```
#define FALSE 0
#define TRUE 1

volatile int STOP = FALSE;

main()
{
        int fd, res;
        FILE *fp;
        struct termios oldtio, newtio;        //控制串口的结构体
        char buf[255];

        fp = fopen("log.txt", "a");        //以添加方式打开文件
        if (!fp) {
                perror("fopen");
                exit(1);
        }
        fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY);        //打开串口设备
        if (fd < 0) {
                perror(MODEMDEVICE);
                exit(-1);
        }

        tcgetattr(fd, &oldtio);        // 保存当前的串口设置,以便最后恢复
        bzero(&newtio, sizeof (newtio));

        newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

        //CS8:8 数据位,1 停止位,没有奇偶校验
        //LOCAL: 没有 modem 控制
        //CREAD: 使能接收数据

        newtio.c_iflag = IGNPAR | ICRNL;

        //IGNPAR  :忽略奇偶校验
        //ICRNL   :将 CR 转换为 NL (否则 CR 将不能作为输入的结束)

        // 原始模式输出
        newtio.c_oflag = 0;
        newtio.c_lflag = ICANON;
        //下面使设置生效
        tcflush(fd, TCIFLUSH);        //清 modem 信号线
        tcsetattr(fd, TCSANOW, &newtio);        //参数 TCSANOW 使设置立即生效

        while (STOP == FALSE) {
                res = read(fd, buf, 255);
                buf[res] = 0;        //字符串的结尾,这样才能打印
                printf(":%s:%d\n", buf, res);        //输出到终端上
                fprintf(fp, ":%s:%d\n", buf, res);        //输出到文件
                if (buf[0] == 'z')
                        STOP = TRUE;        //直到输入 Z 跳出循环
        }
        //恢复初始状态
        tcsetattr(fd, TCSANOW, &oldtio);
        printf("done\n");
        close(fp);
}
```

6.4 Linux 下的网络编程

随着 Internet 的普及，越来越多的主机和设备被连接到网络中，嵌入式设备的发展方向也要求越来越多的网络应用。而 Linux 操作系统为嵌入式系统的网络应用提供了完美的解决方案。Linux 操作系统对各种网络协议的支持非常完备和高效，很多路由器和防火墙都是使用 Linux 操作系统作为其运行平台。本小节将简单介绍 Linux 操作系统下的网络通信程序的编写方法和主要通信函数的使用。

6.4.1 网络通信基本原理

为了把信息通过网络进行传递，在通信的两端必须遵循同样的通信协议。而为了在复杂的网络环境中可靠、高效的传递信息，网络协议的设计就非常重要。为了便于对网络协议进行分析和设计，人们常常把复杂的网络协议分成多个层次，最常见的就是 OSI 的七层网络模型，如表 6.4 所示。

1. OSI 七层网络模型

表 6.4: OSI 7 层网络模型

7	应用层
6	表示层
5	会话层
4	传输层
3	网络层
2	数据链路层
1	物理层

物理层：处理与物理传输介质有关的机械的、电气的和过程的接口，所涉及的是设备之间从一端到另一端传输比特时要遵守的规则。

数据链路层：提供介质访问控制及逻辑链路控制，目的是使物理链路更加可靠，主要提供的服务是差错检测与控制，保证数据帧的可靠传输。

网络层：实现所谓的点到点通信，包括寻径及流量控制，实现数据包经过不同网络节点的传输。

传输层：弥补网络层服务与用户需求的差距，确保交付的数据是无差错的、按序的、没有丢失和重复的。

会话层：实现应用程序之间通信的控制机制，例如全双工与半双工的选择。

表示层：提供两个应用之间进行交换的数据格式，比如压缩和加密。

应用层：提供应用程序的访问手段，一般认为除了前面六层的上层网络应用都属于应用层。例如文件传输、电子邮件等。

虽然国际上规定了标准的网络七层协议，但是真正按照这个标准设置的网络协议却不是很多，我们常用的 TCP/IP 协议族就是一个四层的协议族。

2. TCP/IP 协议简介

TCP/IP 协议是一种互联网协议，可用于不同计算机平台在网络上的通信。在 Internet 上任何基于 TCP/IP 协议的计算机和网络设备都被称为主机（host），而每一部主机都有自己的 IP 地址，主机之间通过路由协议根据 IP 地址进行通信。

如果用上面提到的 ISO 模型来描述的话，TCP 对应于 ISO 的传输层，IP 对应于 ISO 的网络层，从这一定义可以知道 TCP 是建立在 IP 之上的协议。一般情况我们不按照 ISO 模型而重新定义 TCP/IP 模型，而认为它由四个层次组成（如果包括物理层则有五个层次）：

网络接入层：接收 IP 数据包并发送，或者接收物理帧并抽出 IP 数据包交给 IP 层。

互联网层：即 IP 层，利用网际协议来提供不同网络的计算机间的点到点通信。

传输层：即 TCP 层，提供应用程序间的端到端通信。

应用层：严格说不属于 TCP/IP 协议的内容，只是使用到了 TCP/IP，符合其相应的协议标准而已。

TCP/IP 传输层是由 TCP 和 UDP 两个并列的协议组成的。其中，TCP（transport control protocol 传输控制协议）是面向连接的，提供高可靠性服务；UDP（user datagram protocol 用户数据报协议）是无连接的，提供高效率服务。

6.4.2 socket 编程与相关数据结构

我们曾经多次提到 Linux 系统中几乎所有东西都是文件，网络系统也是一样。虽然没有一个设备文件与网络设备对应，但 Linux 提供了一个系统调用叫 socket，这个系统调用将返回一个文件描述符，程序员可以用与操作普通文件描述符类似的方法操作它，来实现网络通信，因此 Linux 系统中的网络编程又称为 socket 编程。socket 的中文翻译是 “套接字”，它不但在 UNIX 编程环境中获得普遍应用，在 Windows 系统中也有了类似的概念，并逐渐成为网络编程的主流。

socket 用来实现两个不同程序之间的通信，这两个程序可以运行在不同的主机、甚至不同的操作系统之上。为了确定通信两端的程序，不但需要两端主机的 IP 地址，还需要另外一个 16 位的整数作为应用程序的标识，称为端口号。例如大家熟知的 HTTP 协议的端口号为 80，TELNET 的端口号为 23，通过不同的端口可以在一个主机运行多个程序进行 TCP 连接。通信两端的端口地址是可以不同的，往往通信双方的服务器端使用固定的端口地址，客户端使用随意的端口地址。

由于不同的计算机平台可能具有不同的字节顺序，当它们通过网络进行通信的时候就可能引起混乱。因此在 TCP 通信的时候必须指定统一的字节顺序——网络字节顺序。Linux 提供了一系列函数对字节顺序进行转换。例如 htons，是 host to network short 的缩写，就是可以把主机字节顺序的 short 类型转换为网络字节顺序。程序员可以不必关心主机的字节顺序直接调用这类函数来获得程序的移植性。

htons 主机到网络字节顺序，short 类型；
htonl 主机到网络字节顺序，long 类型；
ntohs 网络到主机字节顺序，short 类型；
ntohl 网络到主机字节顺序，long 类型。

一般情况下，即将从主机发送到网络的数据需要转变到网络字节顺序，而从网络接收到的数据需要转变到主机字节顺序。

建立一个套接字 socket 也就是建立了一个通信中的端点，我们可以通过程序中指定的远程套接字地址，建立从本地套接字到远程套接字的通信。

为了保持套接字函数调用参数的一致性，Linux 系统定义了一种通用的套接字地址结构，在系统头文件 <Linux/socket.h> 中定义如下：

```
struct sockaddr
{
   unsigned short sa_family;
   char           sa_data[14];
};
```

其中的 sa_family 为套接字的协议簇地址类型，如 TCP/IP 协议簇版本 4（即 IPV4）的地址类型为 AF_INET，IPV6 的地址类型为 AF_INET6，如果不关心具体的版本，可以使用 AF_UNSPEC；sa_data 中存储具体的协议地址，不同的协议簇有不同的地址格式，socket 提供了一些函数来填充这个地址字段。

套接字支持的每种协议簇都定义了自己的套接字地址结构，以前缀 sockaddr_ 开始。TCP/IP 协议的套接字地址结构是 sockaddr_in，在系统头文件 <linux/in.h> 中定义：

```
struct sockaddr_in {
short int            sin_family;          /* 地址类型,AF-xxx*/
unsigned short int   sin_port;            /* 端口号 */
struct in_addr       sin_addr;            /*Internet 地址 */
unsigned char        sin_zero[8];         /* 占位字节 */
};
```

成员 sin_family 的存储区域与通用套接口定义中的 sa_family 相同。对于 IPV4 的网络连接，sin_family 的值将被初始化为 AF_INET。

成员 sin_port 为套接口地址定义 TCP/IP 端口号，它的值必须是网络字节形式。

成员 sin_addr 的定义在结构 in_addr 中，in_addr 定义如下：

```
struct in_addr
{
      u32 s_addr;
};
```

s_addr 同样是以网络字节序的形式代表 IP 地址。

成员 sin_zero[8] 是整个结构以 16 字节的形式对齐。

如果要把一个以字符串形式表示的 IP 地址转化为适当的格式填充到 in_addr 结构中，可以使用 inet_pton 函数，例如：

```
struct sockaddr_in sa;
inet_pton(AF_INET, "192.168.0.100", &(sa.sin_addr));
```

对于服务器程序，如果需要使用本机的 IP 地址，可以使用宏 INADDR_ANY 来作为 32 位的 IP 地址。

```
sa.sin_addr.s_addr = INADDR_ANY;
```

如果要进行反向的转换，即把 in_addr 结构的 IP 地址转换为字符串格式，可以使用 inet_ntop 函数，例如：

```
inet_ntop(AF_INET, &(sa.sin_addr), ip, INET_ADDRSTRLEN);
```

其中 ip 参数为一个可以保存 IP 地址字符串的指针。

另外一个非常强大的函数是 getaddrinfo，它的定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *node, const char *service,
        const struct addrinfo *hints, struct addrinfo **res);
```

其中 node 参数是主机名称或者 IP 地址。service 参数可以是端口号（如 “80”），也可以是特定服务的名称（如 “http”）。addrinfo 结构体的定义如下：

```
struct addrinfo {
    int              ai_flags;     // 如 AI_PASSIVE
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // 协议类型
    size_t           ai_addrlen;   // ai_addr 的存储空间大小
    struct sockaddr *ai_addr;      // 套接字地址结构
    char            *ai_canonname; // 主机名称
    struct addrinfo *ai_next;      // 链表的下一个
};
```

其中 ai_flags 设置为 AI_PASSIVE 表示使用本机的地址。ai_family 用来设置 IP 协议类型。ai_socktype 用来设置套接字的类型，SOCK_STREAM 是流套接字；SOCK_DGRAM 是数据报套接字；SOCK_RAW 是原始套接字，只对 Internet 协议有效，可以用来直接访问 IP 协议。参数 ai_protocol 表示协议类型，0 代表任何类型。

addrinfo 结构实际是一个链表，可以描述多个不同类型的地址。在 getaddrinfo 函数中，hints 参数用来设置输入的参数，res 参数作为返回值，通过一个链表把网络地址信息表示出来。例如需要获得本机的地址信息可以设置 getaddrinfo 的第一个参数为 NULL 并设置 hints 的如下三个参数：

```
hints.ai_family = AF_INET; // IPV4 地址
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;  // 使用本机的地址
```

如果需要获得其他主机的地址信息，就不必设置 ai_flags 参数。具体的使用方法可以参考下一个小节的程序示例。

如果在调用 getaddrinfo 函数的时候发生错误，函数将返回一个不为 0 的错误号码，并可以通过 gai_strerror 函数把这个错误码转化为对应的错误消息。

在使用完作为返回值的链表 res 之后，要记得用 freeaddrinfo 函数释放其占用的资源。这个函数的调用方式如下：

```
freeaddrinfo(res);
```

6.4.3 TCP 网络通信程序的流程

图 6.1 描述了 TCP 客户机和服务器之间的交互过程，这是使用 TCP 协议进行网络通信程序的基本模型。服务器程序首先进行初始化操作：调用函数 socket 创建一个套接字，函数 bind 将这个套接字与服务器公认地址绑定在一起，函数 listen 将这个套接字转换成倾听套接字（listening socket），然后调用函数 accept 来接受客户机的请求。在通信的另外一端，客户机调用函数 socket 创建一个套接字，然后调用函数 connect 来与服务器建立连接。连接建立之后，客户机与服务器通过读、写套接字来进行通信。

函数 socket 用来创建套接字文件描述符。其定义如下：

```
int socket(int domain,int type,int protocol);
```

参数 domain 指定要创建的套接字的协议族地址类型，参数 type 指定套接字类型，参数 protocol 指定协议类型，通常设置为 0，表示使用默认协议。这样一个常见的 TCP 套接字创建操作如下：

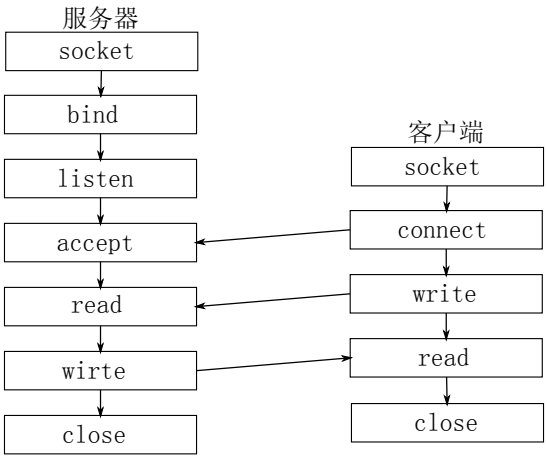

图 6.1: 简单 TCP/IP 交互过程

```
sockfd=socket(AF_INET,SOCK_STREAM,0);
```

如果函数执行发生异常，将返回值为 -1 的错误码，程序需要对这个返回值进行检查以保证程序的正常运行。下面介绍的 connect、bind、accept、listen、recv、send 等函数都是在发生错误的时候返回 -1，程序必须对函数的返回值进行检查。

函数 connect 用来与其他主机建立连接，其定义如下：

```
int connect(int sockfd, struct sockaddr * servaddr, int addrlen);
```

参数 sockfd 是函数 socket 返回的套接字描述符；参数 servaddr 指定远程服务器的套接字地址，包括服务器的 IP 地址和端口号；参数 addrlen 指定这个套接字地址的长度。在程序中的调用方法示例如下：

```
struct addrinfo *si;
connect(sockfd, si->ai_addr, si->ai_addrlen);
```

函数 bind 将本地地址和端口与套接字绑定在一起，为等待其他主机连接做准备，其定义如下：

```
int bind(int sockfd, struct sockaddr * myaddr, int addrlen);
```

参数 sockfd 是函数 socket 返回的套接字描述符；参数 myaddr 是本地地址；参数 addrlen 是套接字地址的长度。在程序中的调用方式示例如下：

```
struct addrinfo *si;
bind(sockfd, si->ai_addr, si->ai_addrlen);
```

bind 函数的最常见错误是 “Address already in use”，即端口号已经被其他程序使用，因为同样一个端口号不能被多个程序同时使用。但有时发生这个错误的原因是一个服务程序已经退出，但操作系统会阻止这个程序所使用的网络端口立刻被其他程序使用，以避免可能的安全漏洞。这个特性可以采用下面的函数调用来关闭：

```
int yes = 1;
setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int));
```

函数 listen 将一个套接字转换为倾听套接字，对一个套接字调用了 listen 之后，客户端就可以连接了。listen 函数的定义如下：

```
int listen(int sockfd,int backlog);
```

参数 sockfd 是指定要转换的套接字描述符；参数 backlog 设置请求队列的最大长度，表示在 accept 函数之前同时最多有多少客户连接到这个套接字。

函数 accept 从倾听套接字的连接队列中接收一个连接，其定义如下：

```
int accept(int sockfd,struct sockaddr * addr,int *addrlen);
```

参数 sockfd 是指定倾听套接字描述符；参数 addr 为指向一个套接字地址结构的指针；参数 addrlen 是地址数据的长度。函数的返回值也是一个套接字，程序可以用这个套接字与客户端进行通信。如果程序不需要继续接受客户端连接，可以用 close 函数关闭原来的倾听套接字。

当客户端与服务器建立连接之后，虽然可以通过函数 read 和 write 完成数据的通信，但对于 socket 通信来说，可以用 send 和 recv 系统调用更好地进行套接字操作。

```
int send(int sockfd, const void *buf, int len, int flags);
int recv(int sockfd, void *buf, int len, int flags);
```

参数 sockfd 指定读写操作的套接字描述符；参数 buf 指定数据缓冲区；参数 len 指向接收或发送数据量的大小；参数 flags 一般可以设置为 0。

服务器端的参考程序如下：

```
#include <string.h>
#include <error.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main(int argc, char *argv[])
{
        //定义两个 socket 套接字，一个用于监听，一个用于通信
        int listensock, connsock;
        struct addrinfo hints, *si;
        const char buff[] = "Hello! Welcome here!\n";
        char recvbuf[100];
        int rv;

        memset(&hints, 0, sizeof(hints));
```

```
        hints.ai_family = AF_INET; // IPV4 地址
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_flags = AI_PASSIVE;  // 使用本机的地址

        // 填充 addrinfo 信息,为后续调用使用
        if ((rv = getaddrinfo(NULL, "5000", &hints, &si)) != 0){
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
                exit(1);
        }

        if((listensock = socket(si->ai_family, si->ai_socktype,
                si->ai_protocol)) == -1){ // 创建一个套接字
                perror("socket");
                exit(1);
        }

        rv = 1;
        if (setsockopt(listensock, SOL_SOCKET, SO_REUSEADDR, &rv,
                sizeof(int)) == -1) { // 允许立刻再使用本端口
                perror("setsockopt");
                exit(1);
        }

        // 绑定到本机地址
        if (bind(listensock, si->ai_addr, si->ai_addrlen) == -1){
                close(listensock);
                perror("bind");
                exit(1);
        }
        freeaddrinfo(si); // 释放 addrinfo 资源

        if (listen(listensock, 10) == -1){ // 开始监听
                perror("listen");
                exit(1);
        }

        // 接收客户端的连接,获得新的套接字
        connsock = accept(listensock, NULL, NULL);
        close(listensock); // 关闭监听的端口
        if (send (connsock, buff, sizeof(buff), 0) == -1)
                perror("send");
        close(connsock);
}
```

客户端的参考程序如下：

```
#include <string.h>
#include <error.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
```

```
int main(int argc, char *argv[])
{
        char recvbuff[100];
        int sockfd;
        struct addrinfo hints, *si;
        int rv;

        if (argc != 2) {
                printf("Usage: %s ip\n", argv[0]);
                exit(0);
        }

        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;

        // 获得服务器地址的 addrinfo 信息
        if ((rv = getaddrinfo(argv[1], "5000", &hints, &si)) != 0){
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
                exit(1);
        }

        if((sockfd = socket(si->ai_family, si->ai_socktype,
                si->ai_protocol)) == -1){ // 建立套接字
                perror("socket");
                exit(1);
        }

        // 连接服务器
        if (connect(sockfd, si->ai_addr, si->ai_addrlen) == -1){
                perror("connect");
                exit(1);
        }
        freeaddrinfo(si); // 释放 addrinfo 结构

        if (recv (sockfd, recvbuff, sizeof(recvbuff), 0) == -1)
                perror("recv");
        printf("%s\n", recvbuff);
        close(sockfd);
}
```

6.4.4 同时操作多个文件描述符

在前面介绍的程序流程中，如果没有客户端连接到服务器，服务器会在 accept 函数处停下来等待，直到有客户端连接 accept 函数才会返回。对于 recv 函数也是类似，如果没有从网络端口获得新的数据，recv 函数也要一直等待到数据到来为止。这种类型的 I/O 操作被称为阻塞式的输入输出，当程序停止在这样的函数时，程序将无法同时处理其他 I/O 操作，这在一些程序中是不能允许的。例如常见的 telnet 程序，它在运行的时候可以同时接受用户的输入和网络传递过来的数据。而如果 telnet 程序阻塞在 recv 函数上就无法及时地获得用户的输入，因此必须由其他的解决方法。

一种解决办法是采用多线程或者多进程的方式，一个线程或进程采用阻塞的方式读取用户的输入，另外一个线程或进程采用阻塞的方式读取网络数据。这种方式虽然可以解决问题但还会遇到进程同步、数据同步等其他问题，把一个本来简单的问题复杂化了。

另外一种解决方法是采用非阻塞的输入输出。在非阻塞模式下，原来可能阻塞的函数操作会立刻返回，得到一个错误号——EAGIN，表示这个操作目前无法完成。采用非阻塞方式，程序可以交替查询需要操作的文件描述符，看哪个文件描述符有可用的数据就对哪个文件描述符进行操作。但这种解决方案的明显缺点就是效率太低，大量的 CPU 时间浪费到了循环查询上，这在多任务的环境中是不可取的。

要把一个文件描述符设置成非阻塞模式，可以在调用 open 函数的时候指定 O_NONBLOCK 选项，或者采用 fcntl 函数调用如下：

```
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

还有一种解决方案是采用异步 I/O，也就是当在文件描述符上可以执行特定的操作后，由系统内核来通知应用程序。但这也不是很好的解决方法，因为并不是所有的系统都支持异步 I/O，而且当接收到内核信号之后，程序仍然需要轮训查看可能接受操作的文件描述符。

因此，这个问题的经典解决办法是采用一种文件输入输出多路复用技术，即通过一个函数可以监视多个文件描述符的状态。最常见的这类函数就是 select，它的定义如下：

```
#include <sys/select.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
        struct timeval *timeout);
```

“select” 函数的第一个参数 n 需要设置为所监视的文件描述符的最大值加 1。中间的三个参数是三个文件描述符的集合，“readfds” 是需要监视读操作的文件描述符集合；“writefds” 是需要监视写操作的文件描述符集合；exceptfds 是需要监视其他异常事件的文件描述符集合。对于不关心的操作类型，可以把相应的描述符集合设置为 NULL。最后的参数 timeout 设置了一个超时的时间，允许 select 函数在这个时间之后即使没有出现任何信号也可以返回。

fd_set 类型的文件描述符集合可以用一系列宏来操作。其中 FD_ZERO(fd_set *set) 用来初始化作为参数的集合，删掉其中的全部描述符。FD_SET(int fd, fd_set *set) 用来把 fd 加入到集合中。FD_CLR(int fd, fd_set *set) 用来把 fd 从集合中去除。FD_ISSET(int fd, fd_set *set) 用来检查 fd 是否存在于集合之中。

timeval 结构体的定义如下：

```
struct timeval {
    int tv_sec;  // 秒
    int tv_usec; // 微秒
};
```

可以设置为一个最小单位是微秒的等待间隔。如果设置的等待时间为 0，select 函数立即返回，实际上相当于立刻对所有文件描述符进行查询。而如果 timeout 的值设置为 NULL，则表示 select 函数将一直等下去，永远不会超时。

如果 select 函数在非超时的情况下返回，三个文件描述符集合会只包含允许操作的文件描述符，可以通过 FD_ISSET 宏来进行检验。下面是一个使用 select 函数的一个例子：

```
tv.tv_sec = 0;
tv.tv_usec = 100; // 最多等待 100 毫秒
do{
   FD_ZERO(&rset);  // 初始化 rset
   FD_SET(0, &rset); // 监视标准输入
   FD_SET(fd, &rset);  // 监视 fd
   if (select (fd +1, &rset, NULL, NULL, &tv) < 0) {
      printf("select error!\n");
```

```
      return -1;
   }
   else {
      if (FD_ISSET(fd, &rset)) { // 如果是 fd 描述符上的事件
         i = read(fd, buf, 1023);
         if (i ==0) return 0;
         buf[i] = 0;
         printf("%s", buf);
      }
   }
}while(1);
```

另外一个适合这一场景的函数调用是 poll 系统调用，它的定义如下：

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

参数 ufds 是一个类型为 pollfd 的结构体数组，数组的元素个数由 nfds 来确定。timeout 是一个以毫秒为单位的整数，如果设置为 0 表示不进行等待，如果设置为 -1 表示永远等待下去。从 poll 函数的参数个数可以看出它的调用方法比 select 简单，pollfd 的结构体定义也很简单，其中 fd 为关心的文件描述符，events 为关心的事件类型，revents 为函数返回时实际发生的事件。poll 函数的返回值为 -1 表示发生错误，为 0 表示超时，大于 0 表示发生了事件的文件描述符个数。

具体的事件类型是下面几个宏的按位或：

```
POLLIN    输入数据准备好
POLLOUT   数据可以立刻输出
POLLPRI   有高优先级的输入数据准备好
```

对于 revents，还可能存在以下几种位组合：

```
POLLERR   文件描述符上发生了错误
POLLHUP   对于 socket 描述符，表示对方断开连接
POLLNVAL 描述符不合法
```

一个示例程序如下：

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

//省略了套接字 s1 与 s2 的初始化部分

ufds[0].fd = s1;
ufds[0].events = POLLIN; // 检查输入数据
ufds[1] = s2;
ufds[1].events = POLLIN; // 检查输入数据
// 等待 3.5 秒
```

```
rv = poll(ufds, 2, 3500);
if (rv == -1) {
    perror("poll"); // 调用 poll() 时发生错误
} else if (rv == 0) {
    printf("Timeout occurred! No data after 3.5 seconds.\n");
} else {
    // 检查 s1 上的事件
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    // 检查 s2 上的事件
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

6.4.5 UDP 编程

前面介绍的 TCP 网络编程被称为是面向连接的，与之对应的就是无连接的 UDP 编程，本小节将对 UDP 的网络通信方式进行介绍。

所谓的无连接就是在 UDP 编程的时候，客户端无须使用 connect 系统调用连接到服务器就可以发送数据，而服务器也无须使用 accept 等待用户的连接。同时 UDP 的数据包格式也比 TCP 数据包简单，因此采用 UDP 的方式进行通信效率更高。付出的代价是 UDP 通信方式是不可靠的，系统并不保证发送出去的数据都可以被接收到，也不能保证发送的多个数据按照同样的顺序到达对端，不过一旦数据被接收到，数据包的正确性是被保证的。因此 UDP 通信经常被用到需要高速反应，对丢包不敏感的网络业务中。例如网络游戏、流媒体播放等。另外由于 UDP 协议栈比较简单，对系统资源占用少，也多应用在一些嵌入式领域。例如我们用到的 TFTP 协议就是基于 UDP 的。

虽然 UDP 协议是不可靠的通信协议，但如果在 UDP 协议之上建立高层的可靠协议，仍然可以实现可靠的数据传输。例如 TFTP 协议就是这样，服务器把每个数据包进行编号，而客户端需要对每个数据包进行应答，通知服务器已经收到了对应包号的数据。如果一个数据包丢失了，服务器就可以超时重发，由此保证所有的数据包都被接收到，并可以按照包号进行排序。对于多媒体类的应用，由于丢失个别数据包不会对播放效果造成影响，数据包的时效性也很重要，因此也就没有必要重传，非常适合使用 UDP 协议。

既然 UDP 协议是无须连接的，因此在发送 UDP 数据包的时候需要使用另外一套发送与接收函数[1]。

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
                            const struct sockaddr *to, socklen_t tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
                              struct sockaddr *from, int *fromlen);
```

sendto 与 recvfrom 函数与 send 和 recv 函数非常类似，只不过多了两个与远程 IP 地址和端口对应的参数。其中 recvfrom 函数在调用的时候只需给 fromlen 参数赋值，当函数返回的时候，from 参数就会被填充为发送数据方的地址与端口信息。而在调用 sendto 函数的时候，由于对方的 IP 是确定的，to 参数的内容可以根据 recvfrom 函数的返回值或者是通过 getaddrinfo 函数来填充。

两个函数的返回值与 send 和 recv 函数含义相同：当返回 -1 时表示函数调用发生错误，否则返回实际发送或者接收到的字节数。下面一个程序示例是一个 UDP 的服务程序，它会把接收到的 UDP 数据打印出来，然后退出。

[1]如果事先通过 connect 函数连接了服务器，也可以使用 send 和 recv 函数操作 UDP 数据包

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <errno.h>
#include   <string.h>
#include   <sys/types.h>
#include   <sys/socket.h>
#include   <netinet/in.h>
#include   <arpa/inet.h>
#include   <netdb.h>
#define MAXBUFLEN 100
int main(void)
{
     int sockfd;
     struct addrinfo hints, *si;
     int rv;
     int numbytes;
     struct sockaddr_in their_addr;
     char buf[MAXBUFLEN];
     size_t addr_len;

     memset(&hints, 0, sizeof hints);
     hints.ai_family = AF_INET ; // IPv4 地址
     hints.ai_socktype = SOCK_DGRAM; // UDP 协议
     hints.ai_flags = AI_PASSIVE; // 使用本机地址
     if ((rv = getaddrinfo(NULL, "4999", &hints, &si)) != 0) {
         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
         return 1;
     }
     if ((sockfd = socket(si->ai_family, si->ai_socktype,
                si->ai_protocol)) == -1) {
           perror("socket");
           exit(1);
     }
     if (bind(sockfd, si->ai_addr, si->ai_addrlen) == -1) {
           close(sockfd);
           perror("bind");
           exit(1);
     }
     freeaddrinfo(si);
     printf("waiting to recvfrom...\n");
     addr_len = sizeof their_addr;
     if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
         (struct sockaddr *)&their_addr, &addr_len)) == -1) {
         perror("recvfrom");
         exit(1);
     }
     buf[numbytes] = '\0';
     printf("got message \"%s\"\n", buf);
     close(sockfd);
     return 0;
}
```

对应的客户端程序如下：

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main(int argc, char *argv[])
{
      int sockfd;
      struct addrinfo hints, *si;
      int rv;
      int numbytes;
      if (argc != 3) {
          fprintf(stderr,"usage: %s hostname message\n", argv[0]);
          exit(1);
      }
      memset(&hints, 0, sizeof hints);
      hints.ai_family = AF_INET;
      hints.ai_socktype = SOCK_DGRAM;
      if ((rv = getaddrinfo(argv[1], "4999", &hints, &si)) != 0) {
          fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
          return 1;
      }
      // loop through all the results and make a socket
      if ((sockfd = socket(si->ai_family, si->ai_socktype,
                  si->ai_protocol)) == -1) {
            perror("socket");
            exit(1);
      }
      if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
                  si->ai_addr, si->ai_addrlen)) == -1) {
          perror("sendto");
          exit(1);
      }
      freeaddrinfo(si);
      printf("sent %d bytes to %s\n", numbytes, argv[1]);
      close(sockfd);
      return 0;
 }
```

6.5 程序调试原理

从软件开发的角度来讲，对软件的调试是非常重要的一环。任何稍具规模的软件都不是一次能够编写成功的，都需要不断地测试和修改，在修改的过程中，调试是定位问题的主要方法。熟练地掌握调试技术是提高软件编程效率的重要保证。在 Linux 环境中最常用的调试工具是 gdb，它的功能非常强大，在这一小节中，我们主要介绍它最基本的功能，满足最基本的调试需求。

如果要对 C 语言编写的代码进行调试，首先要从编译选项开始入手。前面介绍 gcc 的时候提到了 -g 选项，如果在编译的时候没有指定这个选项就不会在二进制代码中包含调试信息，也就无法进行源码级调试了。另外一个需要注意的选项是 -O，如果在编译的时候打开了优化选项，编译出的代码可能对程序的语句进行了调整，有些变量甚至可能被优化掉，这些都会给调试

带来麻烦，因此在调试阶段最好不要打开程序的优化。

6.5.1 程序的加载和运行

最常用的启动 gdb 的方式是以可执行程序为参数运行 gdb。这样就可以把程序加载进来等待调试。输入的命令和 gdb 的输出与下面类似：

```
$ gdb a.out

GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

gdb 的命令直接在(gdb)的提示符后输入，运行程序的命令是 “run” 可以只输入 “r” 来执行此命令。

如果调试的是一个已经运行的程序，可以在(gdb)提示符后面输入 “attach” 命令，并以要调试程序的进程号作为命令的参数。命令执行之后，运行的程序进入暂停调试状态，我们就可以使用 gdb 对其进行操作了。

还有一种情况是当程序运行异常终止，生成 core 文件。core 文件是程序在异常终止时的内存映射，相当于程序结束前的一个快照。在 Linux 下可以用 ulimit 命令加 -c 参数来指定这个文件的大小，如果把这个值设置为 0 就不会产生 core 文件；而如果设置为 unlimited 就可以产生任意大小的 core 文件，core 文件的大小应该同程序所占用的内存相符。我们还可以对这个 core 文件进行调试分析，找出异常的原因。此时需要在执行 gdb 的时候使用 -c 参数。例如：

```
$ gdb a.out -c core
```

输出的最后几行与下面类似：

```
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x080483c2 in main () at testzero.c:7
p[3] = i/j;
```

从最后两行可以看出是在 main 函数中，testzero.c 文件的第 7 行发生意外的。通过进一步的检查：

```
(gdb) p j
$1 = 0
```

p 命令是打印变量的当前值，可以看到这里 j 的值是 0，因此发生了除 0 错误。

如果要为运行的程序设置命令行参数，就要在运行被调试程序前使用 set args 命令，具体的命令参数直接添加到 set args 命令后面。

gdb 包含了一个全面的帮助系统，如果对哪些命令不熟悉就可以使用 help 命令求助。例如输入 help p 就可以得到对 p 命令的全部说明。

6.5.2 设置程序的断点和单步调试

设置断点是程序调试的最基本的手段，当我们发现程序的错误，要对它进行调试的时候，第一步就是根据错误的现象初步判断可能出问题的代码位置，然后在相应的位置前面设置断点，这样当程序执行到那个位置的时候就会停下来，我们就可以调试了。

设置断点的命令是 break，break 命令后面接的参数可以是一个函数名或者一个程序行号。每个断点有个编号，它会在设置断点的时候显示

```
(gdb) break 7

Breakpoint 1 at 0x80483ad: file test.c,line 7.
```

如果想了解当前的断点情况，就可以用 info break 命令查看所有断点。

删除断点用 “delete” 命令，它的参数是断点的编号。或者用 “clear” 命令，它的参数与 break 命令相同，是相应的函数或者行号。

对于断点还可以使用 “disable” 命令让其失效，或者使用 “enable” 命令恢复断点。这两个命令的参数是断点的编号。

gdb 断点的另外一个功能是设置条件。我们可以在设置断点的命令中加上 “if” 子句，设置断点在特定的条件下中断。例如：

```
(gdb) break 7 if j =0
```

则此断点只有在 j = 0 的条件下才会中断。

还有一种非常有用的断点在 gdb 中被称为 “watchpoint”，它用来监视某个变量，当这个变量的值被改变的时候程序中断，这个功能在调试一些比较隐蔽的错误时非常有用。下面是个例子：

```
(gdb) watch p[2]
Hardware watchpoint 2: p[2]
(gdb) c
Continuing.
Hardware watchpoint 2: p[2]

Old value = -17 '?
New value = 99 'c'
main () at test.c:9
9         p[3] = ‘b’;
```

程序中断之后常用四种方式让程序继续执行。一种是 “continue” 命令（缩写 c），可以让程序继续全速运行；一种是 “next” 命令（缩写 n），让程序单步执行下一个程序行。还有一种是 “step” 命令（缩写 s），它也是执行下一个程序行，但是如果下一行程序是一个函数调用，step 命令会进入到函数内部停止到函数的第一条指令前，而 next 命令会把它当成一条语句执行。最后一种是 “finish”，它让程序执行到当前函数结束，并打印出返回值。

在（gdb）提示符下直接回车表示重复执行上一条命令（如果有的话）。这在单步调试的时候非常方便，不必重复输入命令，直接回车就可以达到目的。

6.5.3 其他常用的命令

“list” 命令用来查看程序的源文件。命令的参数可以是行号、函数名，不带参数的 list 命令可以显示前一条 list 命令所打印文件内容的后续部分。

“backtrace”（缩写 bt）命令用来查看当前的函数调用堆栈。例如：

```
(gdb) bt
#0  try (a=1, b=4, c=5, d=6) at testit.c:10
#1  0x08048667 in main (argc=5, argv=0xbffff694) at testit.c:59
```

表明程序正在 try 函数中，而 try 函数又是 main 函数调用的。try 后面括号中的内容是调用它时使用的参数。这个命令在查看 core 文件的时候也很有用，可以用来发现程序崩溃时的调用路径。

如果需要对某个变量进行 “长期” 监视，除了可以每次使用 p 命令显示变量值，更好的办法是使用 “display” 命令，它同样可以接受一个表达式作为参数，并在程序每次断点的时候显示表达式的值。删除监视表达式可以使用 “delete display” 命令。

6.5.4 远程调试

在嵌入式系统的环境里，由于各种资源受限，直接用 gdb 调试往往并不方便，这时候就可以使用 gdb 的远程调试功能。gdb 的远程调试包含客户端与服务器两个部分，服务器部分是交叉编译的 gdbserver 程序，可以运行在目标平台。客户端是工具链的一部分，在开发主机运行并可以 “理解” 目标平台的二进制代码。客户端与服务器可以通过串口或者 TCP 网络进行连接，在开发主机对目标代码进行调试，而目标代码同时也在目标平台真实运行。

例如在目标平台调试 a.out 文件，可以由 gdbserver 进行调用

```
$ gdbserver /dev/ttyS0 a.out
```

此时 a.out 并没有开始运行，而是等待客户端连接，并在客户端的控制下调试程序。客户端运行在开发主机，开发主机上必须有包含调试信息的 a.out 文件（在目标平台运行的 a.out 可以没有调试信息）。这里的/dev/ttyS0 是目标平台的串口设备。

```
$ arm-none-linux-gnueabi-gdb a.out
gdb> target remote /dev/ttyS0
gdb> set sysroot <library-path>
```

最后一条命令用来设置 gdb 寻找动态链接库的位置，设置的路径就是工具链中 sysroot 的路径。经过这样的设置之后就可以通过前面介绍过的 gdb 调试命令对 a.out 程序进行调试了。这里的/dev/ttyS0 是开发主机的串口设备，而在目标平台调用的参数对应的是目标平台的串口设备，虽然名字相同，但对应的物理设备是不同的，只是它们通过串口线连接在一起。

如果要通过网络方式连接 gdbserver，在目标平台应该这样设置：

```
$ gdbserver localhost:3333 a.out
```

其中的 3333 可以是任何一个可用的 TCP 端口。

或者可以用 gdbserver 连接到一个已经运行的程序上，同样是通过进程的 PID 号：

```
$ gdbserver --attach localhost:3333 <PID>
```

在开发主机通过网络连接 gdbserver 的命令如下：

```
gdb> target remote localhost:3333
```

6.6 软件版本控制

在常见的软件编程类的教科书中，版本控制经常是一个被忽视的话题，人们往往在实际工作之后才逐渐接触到版本控制软件，也往往在有了更多的软件编程经验之后才意识到版本控制是多么重要的一件事情。而在开源软件领域，由于开源软件开发的特殊方式，版本控制软件的应用非常广泛，几乎全部开源软件在其开发过程中都使用了一种或者多种版本控制软件。如果要获取和修改这些软件的源代码，就必须对版本控制软件有一定的了解。

版本控制系统（version control system）是源代码或者其他文本数据[1]进行版本管理的工具。简单地说，版本控制系统可以记录源代码的全部开发历史，可以非常方便地对不同的版本进行对比，可以容易的恢复到早期的代码版本。例如当用户不小心把一个文件删除或者做了不能恢复的修改时，就可以利用版本控制软件把更早的版本恢复。如果在某个版本中发现了 bug，也可以利用版本之间源代码的比较，找到 bug 引入的位置。对于一般简单应用来说，有了版本控制软件就可以不用频繁的对源代码进行备份，版本控制系统可以简单地帮你完成这一工作。

版本控制系统在开源社区获得应用的一个重要原因是其可以简化多人合作开发软件的流程。试想如果多人同时开发一个软件，他们每个人都要拥有该软件的源代码。如果一个人对软件做了修改，他必须通知其他所有人，把 patch 文件通过邮件发送给大家。如果开发人员比较多，代码开发速度也比较快的话，保持所有人都拥有最新的代码就是非常困难的事情。如果一个开发人员不基于最新的代码进行开发，一方面可能造成重复劳动，另一方面也可能使得他做的 patch 无法被使用最新代码的开发人员应用。因此如果没有一个工具对源代码进行管理，要完成像 Linux 内核这样的开源项目简直无法想像。

6.6.1 Git 简介

最原始的版本控制系统是本地版本控制系统。这种系统会把不同的版本信息保存在本地一个特殊的目录中，比较典型的是保存后一个版本与前一个版本的 patch 文件。当需要一个特定版本的时候，只需要把累计的 patch 文件应用到原始版本就可以了。显然本地版本控制系统并没有解决多人协作的问题。

较新的版本控制系统是集中式的版本控制。源代码的全部版本信息被集中保存在一个服务器上，不同的客户端可以从这个服务器获取源代码或者向这个服务器提交修改。这种版本管理方式获得了很大的成功，例如 CVS、Subversion 等很多开源工具都采用了集中式的管理方式。这种版本控制方式简化了对开发人员的管理，开发人员可以随时从服务器获得最新的代码，并只需把代码提交到一个服务器。

为了了解使用版本控制系统的流程，这里介绍几个术语：

- Checkout，获取最新代码，简写为 co
- Checkin，提交代码，简写为 ci
- Branch，代码分支

在使用集中式版本控制系统的时候，开发人员如果要编写代码，首先要 checkout 最新的代码，然后进行编辑。当准备提交的时候，仍然要 checkout 一次，如果有其他开发者在两次 checkout 之间提交了代码，版本管理系统会自动把新代码 “合并” 进来。此时开发人员就可以继续 checkin 代码，把修改更新到服务器。如果在合并或者提交的时候发生了冲突，开发者必须先解决冲突才能继续。冲突的原因可能是不同的开发者修改了同一个文件的相同部分，或者其他系统无法自动合并的修改。

最常见的集中式版本控制系统就是 CVS，很多开源软件都提供 CVS 的代码库。匿名访问的 CVS 服务器一般采用 pserver 协议，通过输入用户名和密码从服务器上下载代码。首先用通过 -d 参数指定代码的路径，并用 login 命令登录服务器，之后用 checkout 命令下载项目代码。例如：

[1] 虽然也可以对二进制文件进行版本控制，但多数 VCS 系统对二进制文件的处理能力远不如文本文件

```
$ cvs -d :pserver:anoncvs@cvs.some.host:/cvs login
Logging in to :pserver:anoncvs@cvs.some.host:2401/cvs
CVS Password:
$ cvs -d :pserver:anoncvs@cvs.some.host:/cvs checkout some_project
```

集中式版本控制系统的最大缺点是如果集中的服务器出现问题，就可能造成比较大的损失，因为所有的用户端都只是包含一个版本的源文件，如果服务器的硬盘损坏，也许所有的历史记录就无法恢复了。另外一个缺点就是用户本地只有最新的代码树，进行版本比较、提交代码等工作必须联网来完成。

分布式的版本控制系统（DVCS）是目前最流行的版本控制软件，例如 Git，Mercurial 等。每个 DVCS 的客户端在 checkout 的时候都会把全部版本信息更新到本地，开发者甚至可以在离线的情况下提交代码。采用分布式的版本管理方式使开发者可以获得更加灵活的协作方式，在本小节里，将对 Git 版本控制系统做简单的介绍，并对使用 Git 进行开发和协作的方式进行初步的介绍。

Git 版本控制系统是 Linux 内核的最初设计者 Linus 在 2005 年为 Linux 的内核开发而设计的版本控制软件。它的设计目标主要集中在如下几个方面：

- 速度和效率
- 设计思想简单
- 对并行设计的支持（Linux 有上千个并行开发分支）
- 完全的分布式

由于 Git 在 Linux 内核开发上的成功，使得 Git 迅速获得了推广，越来越多的开源软件开始使用 Git 作为其版本控制工具。从远程服务器获取源代码拷贝的方法是使用 git clone 命令，下面的示例省略了输出信息：

```
$ git clone git://www.someserver.com/some/project.git
```

命令成功完成后，在当前目录的 project/.git 子目录中就会包含一个完整的源代码数据库，在 project 目录中包含了最新的代码版本作为工作目录，可以在这个工作目录中使用 git 命令获取其他版本代码或者进行代码的修改与提交。

Git 使用配置文件记录用户信息，并在提交代码修改的时候将用户的名字与邮件地址记录到提交记录中。因此在使用 Git 之前最好设置一个合适的参数。对用户名与邮件地址的设置可以参考下面的例子：

```
$ git config --global user.name "Bill"
$ git config --global user.email bill@example.com
```

如果要在本机建立一个新的源代码数据库，需要依次执行如下的命令：

```
$ git init
$ git add *.c
$ git commit -m "Initial commit"
```

在这个命令列表中同样省略了输出信息。其中 git add 命令用来设置在数据库中添加的文件，而 git commit 命令用来真正把添加的文件提交到数据库中。在提交的时候需要为这个版本的修改内容设置一个提示信息，在上面的例子中使用 -m 参数设置一个简短的信息。如果不提供 -m 参数，git 会调用编辑器让用户输入信息。

如果修改了 hello.c 文件并需要把修改提交到数据库中，用户需要首先用 git add 将其添加到临时提交区域，最后用 git commit 命令提交到数据库。如果要添加的文件都不是新文件，即已经存在于版本控制系统中，就可以用 git commit -a 命令将其一次提交到数据库，而省略了 git add 的过程。

如果要查看当前工作目录的状态，可以使用 git status 命令：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

目前还没有提到分支（branch）的概念，这个概念会在下一个小节进行介绍。在 Git 系统中，建立分支与分支合并是代价很小的操作，开发人员会非常频繁地建立分支。默认情况下，版本数据库的主分支被称为 master，所以可以在前面例子中看到 “On branch master”。这个例子中的工作目录是 “干净” 的，对于经过修改的工作目录，git status 所提示的信息更具参考性。对于刚接触 Git 不久的用户，可以根据 git status 信息的提示，完成期望的操作。例如下面的示例：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Makefile
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   hello.c
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README
```

如果不想让一个文件出现在 Untracked files 区域，可以建立一个.gitignore 文件，把一些文件名添加到这个文件中。例如可以把 C 语言编译的临时目标文件 “*.o” 写入到.gitignore 中（这里支持通配符）。

如果要查看代码的修改历史，可以用 git log 命令。git log 命令可以列出每个代码版本的基本信息，例如：

```
commit e456f4be9182800b254281466eae8965fee2a8d3
Author: Bill <bill@example.com>
Date:   Wed Jan 20 21:26:08 2010 -0500

    Initial commit
```

其中在 commit 后面的一串数字是这次提交后的全部数据的 SHA 校验值，Git 系统用 SHA 校验来对每个版本进行标识。在实际使用中可以使用这个校验值的前几位对这个版本进行引用。例如在不发生混淆的情况下，可以只使用 e456 来标识这个版本，如果有多个版本以 e456 开头，就必须使用更多的数字。

如果需要标记一个特定的版本，可以使用 git tag 命令。例如

```
$ git tag -a v0.1 -m "First relaese"
```

在当前的工作目录，可以使用 git checkout 命令获取任何一个代码版本。例如可以通过 git checkout master 命令获得最新版本代码，可以通过 git checkout v0.1 获得前面标记过的版本，也可以通过 SHA 来选定版本。

6.6.2 Git 分支

Git 分支的使用也很灵活，具体如何使用分支需要根据不同的场景来决定。下面考虑一种最常用的场景：

一个软件经过一段时间的开发，已经发布了 0.1 版本，并交付使用，目前正在进行后续开发。忽然接到用户反应 0.1 版本具有一个 bug，需要立即修改。由于当前添加的代码还不稳定，不能在当前版本修改 bug。因此需要回退到 0.1 版本，并建立一个新分支进行修改。完成这个过程的 git 命令列表如下：

```
$ git checkout v0.1    # 回退到 v0.1 版本
$ git branch hotfix    # 建立新的分支:hotfix
$ git checkout hotfix  # 把当前分支转到 hotfix
$ git commit -a -m     # 修改之后提交
$ git checkout master  # 回到主分支
```

此时的代码历史由图 6.2 所示，每个代码版本（图中用圆角矩形表示）有一个唯一的编号，后继的版本由一个有向线段指向自己的父版本，其中 C2 为标记为 v0.1 的版本。master 与 hotfix 为两个分支的指针，目前分别指示在 C3 和 C4 上。

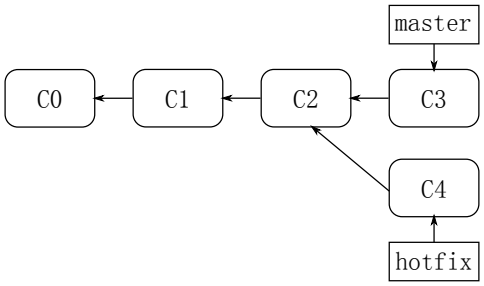

图 6.2: 建立分支的 Git 历史

回到 master 分支之后，显然我们应该把在 hotfix 分支修改的 bug 应用到主分支。此时可以执行合并操作：

```
$ git merge hotfix
```

执行合并操作会建立一个新的源代码版本 C5，master 分支的指针同时移动到这里。显然 C5 具有两个父版本，如图 6.3 所示。

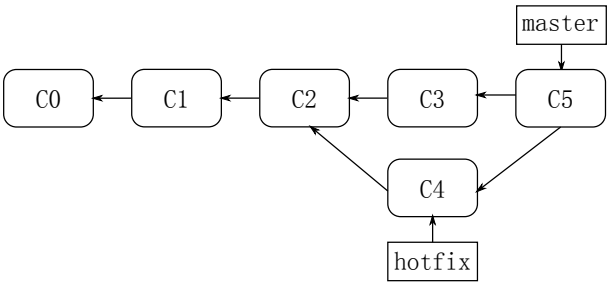

图 6.3: 分支合并后的 Git 历史

如果在图 6.2 中 C3 版本不存在，master 指针指向 C2 版本，那么在执行 merge 操作的时候，可以直接把 master 指针移动到 C4 位置，不必创建 C5 版本。

在执行合并操作的时候，git 会找到两个分支的共同祖先，图 6.3 中是版本 C2，获取两个分支对于共同祖先的 patch 文件。合并的结果是通过对于共同祖先应用两个 patch 文件获得的。

如果在合并时出现冲突，例如修改了同样的文件，就会得到如下的提示信息：

```
$ git merge hotfix
Auto-merged hello.c
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit the result.
```

发生冲突的文件中会包含两个 patch 中冲突的部分，如下面代码所示：

```
<<<<<<< HEAD:hello.c
        printf("Hello World!\n");
=======
        printf("Hello world!\n"); // Print greating message
>>>>>>> hotfix:hello.c
```

代码中 “=======” 是分隔线，与 “<<<<<<<” 之间部分是当前分支的代码（Git 使用 HEAD 表示当前分支），与 “>>>>>>>” 之间部分是 hotfix 分支代码。用户必须手工解决这部分冲突，删除提示信息与分隔线，最后重新提交修改。

如果一个分支不再需要了，可以用如下命令把这个分支删除：

```
$ git branch -d hotfix
```

需要注意，删除分支并不会把分支所指示的特定版本删除，而仅仅删除了分支所代表的指针，用户仍然可以通过 checkout 命令回到原来分支的版本。

如果要与其他开发人员协作开发，就要使用远程分支。对于前面用 git clone 命令复制下来的代码库，实际上已经包含了一个叫做 origin 的远程服务器，可以通过 origin/master 来访问这个服务器上的 master 分支。如果需要查看当前都有哪些远程服务器，可以用 git remote 命令来查看：

```
$ git remote -v
origin    git://www.someserver.com/some/project.git
```

增加或者删除一个远程服务器需要使用 git remote add/rm 命令：

```
git remote add [remotename] [url]
git remote rm [remotename]
```

从远程服务器更新代码需要使用 git fetch 命令

```
git fetch [remotename]
```

如果要把本地的分支提交到远程，需要使用 git push 命令。不过提交代码需要远程服务器赋予权限，远程服务器的 url 一般是 SSH 的安全通道，具有如 git@host.domain 的形式。下面的例子把本地的 master 分支提交到 origin 远程服务器。

```
$ git push origin master
```

6.7 实验 :Linux 平台 C 编程

实验目的

1. 掌握嵌入式 Linux 平台 C 语言开发的基本步骤
2. 掌握调试程序的方法
3. 熟悉 gcc 的使用

实验内容

一、“Hello world!” 程序

在命令行下输入：

```
$ vi hello.c
```

然后在编辑器中输入如下代码：

```
#include<stdio.h>
int main(void)
{
     printf(“hello, linux gcc world!\n”);
     return 0;
}
```

在命令行上键入以下命令编译此源程序

```
$ gcc hello.c
```

在没有任何指定的情况下，编译器默认的将输出文件命名为 a.out。注意在运行生成的可执行文件的时候，因为当前目录并不在 Linux 的默认搜索范围之内，文件名前面要指定运行文件的路径。

```
$ ./a.out
hello,linux gcc world!
```

我们肯定不希望所有的可执行文件都叫 a.out，我们需要不同的文件有不同的名字，这时可以使用 -o 选项，它可以定义文件的名字，如下所示：

```
$ gcc hello.c -o hello
$ ./hello
hello,linux gcc world!
```

用交叉编译工具重新编译这个程序，下载到开发板中运行，查看运行结果。参考的编译命令如下：

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

在开发板中可以使用 Flash 中存在的内核和文件系统，也可以采用 NFS 文件系统。使用 NFS 的好处是不必使用 ftp 下载执行文件，可以直接用 cp 命令复制到 root 文件系统所在的位置。在开发平台环境可能需要下面的命令来运行程序：

```
$ chmod +x hello_arm
$ ./hello_arm
hello,linux gcc world!
```

二、网络编程

利用 SOCKET 编程实现客户端与服务器之间的网络通信,服务器端的程序和客户端的程序可以实现交互聊天的功能。运行时要求服务器程序在实验板上运行,客户端的程序在调试主机上运行。

第七章　Linux 模块化驱动程序原理

Linux 的驱动程序有固定的编程接口，用户程序可以在完全不知道硬件细节的情况下对硬件进行操作。因此为硬件编写驱动程序就是把硬件的操作通过标准内核接口抽象化的过程。

Linux 操作系统驱动程序的另外一个重要特点是其优秀的模块化设计。这种设计可以使驱动程序能够独立于内核的其他部分，只在需要的时候才加载到内核，不需要的时候还可以干净地从内核中删除。这样不仅使得系统的运行更加高效，也使得调试驱动程序变得简单，而不用每次修改驱动都重新启动系统。

7.1　Linux 驱动编写基础

在把 Linux 操作系统移植到一个新的平台、或者给已经运行 Linux 的平台增加新的硬件设备之后，部分陌生的硬件设备并不能被 Linux 系统所识别，这时候就需要对这些硬件编写或移植驱动程序。为 Linux 编写驱动程序往往不必从头开始，因为有大量的开源驱动可以作为参考，对于比较常用的硬件，往往已经有人写了驱动并公开出来，经过简单地修改就可以应用到自己的平台。

在编写驱动程序的时候还要注意几个基本的概念：

- 编写访问硬件的内核代码时不要给用户强加任何策略，因为不同的用户会有不同的需求，驱动程序应该处理如何使硬件可以使用的问题，而将怎样使用硬件的问题留给上层应用。这可以说是硬件驱动灵活性的基础，灵活的驱动可以给用户更多的发挥空间。

- 由于驱动程序是运行在内核态，而 Linux 操作系统对内核态的程序完全信任，所以驱动程序中的错误可能会导致整个系统的崩溃，在编写驱动的时候就要尽量避免错误出现，特别要注意的是缓冲区溢出、非法指针等错误。一般来说，保持驱动程序的简洁和清晰是避免错误的有力手段。

- 作为驱动的另一个要求是不要占用太多的系统资源，比如内存和 CPU 时间，否则会使系统的性能严重下降，只能作为失败的驱动程序而被抛弃。与常规应用程序可以访问到很大的栈空间不同，内核的所有代码共用很小（可以小到 4096 字节）的栈，所以尽量不要在内核代码中使用占有内存比较多的自动变量，节省内核的栈资源。

- 如果驱动程序可以被多个进程访问，还要注意处理并发的问题，永远不能假定用户空间只能按特定的方式访问内核资源。

在 Linux 系统中，多数设备驱动通过设备号与文件系统中的设备文件建立对应的关系，一个设备号可以分解为主设备号和子设备号。当应用程序用 open 系统调用打开一个设备文件的时候，内核通过该文件的主设备号从一个驱动列表中找到这个设备号对应的驱动程序，并调用这个驱动的 open 函数接口。而子设备号的解释就由设备驱动程序来负责，内核仅仅把子设备号传递给驱动程序。

早期的内核只支持固定的设备编号，就是说在编写驱动之前必须确定设备的设备号并建立好设备文件，在内核源码的 Documentation/devices.tex 文件中有所有设备编号的列表，其中字符设备和块设备分别编号，互相不会干扰。可这种策略限制了可用设备的总数，也存在设备号冲突的问题，所以在设计新的驱动程序的时候，倾向于采用动态分配设备号的策略。不过在嵌入式设备中，很少需要频繁更换硬件，驱动程序采用固定的设备编号是最简洁的一种实现。

Linux 系统可以将设备粗略地看成三种类型：字符设备、块设备和网络接口。其中字符设备是能够像字节流一样被访问的设备，例如字符终端和串口。对这种类型设备的设备文件进行操作与对普通文件操作非常类似，只是对大部分设备的操作只能是顺序进行，不能移动操作的指针。这样的设备驱动程序一般要实现 open、close、read、write 系统调用。块设备是可以容纳文件系统的设备，它与字符设备的区别对用户是透明的，只在内核中要实现不同的接口，同时还要支持挂载操作。网络接口设备和前两种设备完全不同，它并不在文件系统上有对应的设备文件。对网络设备的访问是通过唯一的名字，如 eth0 来进行的。和网络设备的通信也是采用一套不同的和数据包传输相关的函数进行的，如 socket、bind、listen、accept、connect 系统调用。

7.1.1 最简单的 Linux 驱动示例

首先我们还是来看一个最简单的驱动程序，和传统的编程介绍一样，它只是一个 “Hello, World!” 程序，并不与任何一种硬件打交道。

```
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>

static int __init hello_init(void)
{
        printk(KERN_ALERT "Hello, world\n");
        return 0;
}

static void __exit hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, cruel world\n");
}
MODULE_LICENSE("Dual BSD/GPL");

module_init(hello_init);
module_exit(hello_exit);
```

与 2.4 内核相比，2.6 内核在模块的编译方法上有了很大的变动，这里仅介绍 2.6 内核的情况。在 2.4 内核的平台上，编译驱动模块的方法请参考内核文档。

为 2.6 的内核编写驱动模块最简单的办法是写一个只有一行语句的 Makefile：

```
obj-m := hello.o
```

其中假定驱动程序的源代码文件名为 hello.c。与之相对应的 make 语句有些复杂，编译的过程如下所示：

```
$ make -C /home/embedded/linux/linux-2.6.33 M=`pwd` modules
make: Entering directory `/home/embedded/linux/linux-2.6.33'
  CC [M]  /home/embedded/drivers/2.6/hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/embedded/drivers/2.6/hello/hello.mod.o
  LD [M]  /home/embedded/drivers/2.6/hello/hello.ko
make: Leaving directory `/home/embedded/linux/linux-2.6.33'
```

这条命令实际利用 -C 参数，转换到内核代码的路径中执行 make 命令，利用内核的 Makefile 来进行编译。这里使用正确的内核代码非常重要，因为驱动可以看作内核代码的一部分，每个驱动只能对应特定版本的内核，如果在加载驱动的时候模块与当前运行的内核版本不匹配，加载就可能失败。make 语句还通过 M 变量把驱动所对应的路径传递给编译系统（注意 ‘pwd‘所使用的引号，它是把 pwd 命令的输出——即当前目录——传递给 M 变量），而编译的目标名称是 modules。内核的编译系统会包含驱动目录中的 Makefile，完成对驱动的编译。因此前面所写的 Makefile 变成了内核复杂 Makefile 集合的一部分，并不能简单地独立分析。

如果我们的驱动由两个源文件 file1.c 和 file2.c 共同生成，我们的 Makefile 就要进行如下修改：

```
obj-m := hello.o
hello-objs := file1.o file2.o
```

当然，如果觉得前面的 make 命令过于冗长，也可以把这行 make 语句写到一个脚本中，减轻每次输入的麻烦。其实还有更复杂一些的 Makefile 的写法，可以获得只用 make 一个单词执行全部命令的效果：

```
ifneq ($(KERNELRELEASE),)
    # called from kernel build system: just declare what our modules are
    obj-m := hello.o
else
    KERNELDIR = /home/embedded/linux-2.6.33
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
        rm -rf *.o *.ko *.mod.c .tmp_versions .depend .*.cmd core
endif
```

ifneq 的用法在前面介绍 Makefile 的时候没有提到，因为它并不是那么常用。但从它的名字也很容易猜到它的含义：用来判断两个参数是否相等（if not equal）。上面的例子中它比较了 $(KERNELRELEASE) 与一个空字符串（逗号后面什么都没有），当其不相等的时候 Makefile 就只包含了如下内容：

```
    # called from kernel build system: just declare what our modules are
    obj-m := hello.o
```

这与前面简单 Makefile 的内容是一致的。但实际上 $(KERNELRELEASE) 是没有定义的，也就是 ifneq 的判断没有成立，此时的 Makefile 等效于

```
    KERNELDIR = /home/embedded/linux-2.6.33
    # The current directory is passed to sub-makes as argument
    PWD := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
        rm -rf *.o *.ko *.mod.c .tmp_versions .depend .*.cmd core
```

因此在执行 make 命令的时候就等同于如下的命令：

```
$ make -C $(KERNELDIR) M=$(PWD) modules
```

这条命令实际上与前面对应简单 Makefile 的编译命令相同，最后也会由对应内核的编译系统包含这个 Makefile，不过由于内核的编译系统中定义了 $(KERNELRELEASE)，实际包含的内容是 ifneq 有效的部分。这样就可以看出这个复杂些的 Makefile 实际上与前面哪个是等效的。

2.6 版本内核编译出的模块扩展名为.ko，加载模块所用的函数界面也有所区别，所以如果使用了比较老的发行版还必须升级 insmod 相关的命令，本章以后部分的讲解以 2.6 内核为主，同时也推荐新的设计尽量使用新的 Linux 内核。

把驱动模块加载到内核需要用到 insmod 命令，它和移除模块的 rmmod 命令的执行都要求用户具有 root 的执行权限。例如要加载前面编写的 hello.ko 模块，需要执行下面的命令：

```
# insmod ./hello.ko
```

加载成功之后可以在/proc/modules 文件中看到 hello 这个模块。使用 cat /proc/modules 或者 lsmod 查看当前已经加载的模块。卸载模块采用：

```
# rmmod hello
```

对于驱动程序来说，它并没有一个 main 函数的入口地址，它只提供了标准的接口函数给内核，内核在需要的时候对这些函数进行调用。上面这个简单的例子介绍了两个最基本的接口：init_module 和 cleanup_module，它们分别在模块被加载和卸载的时候被调用，因此在 init_module 中主要做一些设备的初始化工作，如分配内存、注册设备等。在 cleanup_module 中就要做相反的工作，因为在内核里系统不会自动清理不用的内存。

一般在模块初始化的时候会进行资源的申请，因此要对有可能的申请失败进行处理。特别是在部分资源申请成功的情况下，申请新的资源失败了，这时也应该把申请成功的资源释放掉。

2.3.13 版本的内核之后，在驱动程序中允许使用自己定义的函数名作为初始化和清除函数，这样做的好处是可以让这个函数名取得更有意义，有利于程序的调试。在使用这种方式的时候我们要包含 <linux/init.h> 这个头文件，并且在驱动程序中做如下的声明：

```
module_init(my_init);
module_exit(my_cleanup);
```

my_init 和 my_cleanup 是自定义的函数名，否则必须使用 init_module 和 cleanup_module 作为函数的名称。

hello_init 函数定义中的 ___init 看起来有些特别，这个描述符是 gcc 的一个扩展功能，它告诉内核 hello_init 函数仅用来作为初始化操作，在其执行完毕之后就可以把这个函数占用的内存释放。另外一个类似的描述符是 ___initdata，这个描述符所修饰的数据结构也仅在内核或驱动的初始化阶段使用，在初始化结束之后其内存也可以安全释放。这种设置可以为内核节省宝贵的内存空间，在可能的情况尽量使用这两个描述符。

hello_exit 函数的 ___exit 描述符则用来通知内核这个函数仅用在模块卸载的时候。因此当这个模块被静态编译到内核的时候，这个函数根本不会链接到内核中，这不但减少了内核本身的尺寸，也减少了内核对内存空间的占用。

MODULE_LICENSE(”Dual BSD/GPL”) 这段程序只是为这段程序定义了版权信息，如果没有这个定义，在加载 hello.o 的时候会有一个警告信息：

```
Warning: loading ./hello.o will taint the kernel: no license
```

宏 MODULE_LICENSE 对模块的版权进行了说明，如果不使用这个宏，在这个模块中将不能使用其他声明为 GPL 的函数。其他的宏还有 MODULE_DESCRIPTION 和 MODULE_AUTHOR 等，可以分别对模块进行简单描述和说明模块的作者。

7.1.2 printk 函数简介

由于在内核中无法使用标准的 C 库函数，所以就连最常见的 printf 都不能使用，上面程序中使用的 printk 函数是专门提供给内核使用的，它的用法和 printf 基本类似，可以在驱动程序中用来输出调试信息。

在 7.1.1 节驱动示例中，如果在字符终端上执行加载命令，就可以看到 “Hello, world” 的输出，在清除模块的时候也可以在终端上看到卸载的信息。不过如果使用图形界面的虚拟终端，你会看不到加载和卸载模块时候输出的信息，而必须用 dmesg 命令察看内核最近输出信息的历史记录。由于保存内核信息的空间有限，当容量不足的时候，早期的信息就会被抛弃，dmesg 就只能看到最新的消息。

我们在 printk 函数的字符串参数中使用了 KERN_ALERT，它实际上扩展为字符串：“<1>”，而这部分信息没有输出到终端。实际上这个数字是内核信息的日志级别，只有超过了当前日志级别的信息才会输出到终端。当前内核的日志级别可以在/proc/sys/kernel/printk 文件中看到。这个文件包含了四个整数，其中前两个是控制台的当前日志级别和默认日志级别。我们在 printk 的参数中使用较高的日志级别就是要保证信息得到输出。在 <linux/kernel.h> 头文件里一共定义了 8 个级别 (0-7) 的输出，从高到低分别由如表 7.1 所示：

表 7.1: 内核日志级别

KERN_EMERG	最高级别，一般只用来打印崩溃信息
KERN_ALERT	需要立即处理的信息
KERN_CRIT	关键信息，一般用来显示严重的硬件和软件错误
KERN_ERR	用来显示硬件错误
KERN_WARNING	显示不会造成严重错误的警告信息
KERN_NOTICE	显示需要引起注意的信息
KERN_INFO	显示一般信息，例如驱动所发现的硬件列表
KERN_DEBUG	用来显示调试信息

如果在调用 printk 的时候没有指定日志级别，那就会使用 DEFAULT_MESSAGE_LOGLEVEL 这个内核常量作为默认日志级别。目前在 2.6.33 内核中，默认的日志级别是 KERN_WARNING。

为了方便地在输出调试信息和不输出调试信息之间转换，用一个宏变量作为开关是比较合适的。如下面的代码所示：

```
#undef mydebug
#ifdef HELLO_DEBUG
#define mydebug(fmt, arg...)  printk("<1> hello: " fmt "\n", ##arg)
#else
#define mydebug(fmt, arg...)
#endif
```

在这样的定义中，我们可以使用 mydebug 代替 printk 来输出调试信息，并可以方便地通过是否定义 HELLO_DEBUG 来决定是否把调试信息输出。

printk 与 printf 的一个区别是：printk 是 “行驱动” 的，也就是说只有收到一个换行符，数据才会真正输出到终端，否则就不会有任何信息输出。另一个值得注意的问题是我们在调试嵌入式设备的时候，经常是从串口获得显示信息，如果我们使用 printk 过于频繁的话，串口的传输速度就会成为瓶颈，这样会造成系统的性能下降甚至停止反应。

7.2 内核的编译系统

我们使用 make 命令编译 Linux 内核的时候利用了内核源代码的 Makefile 文件，我们通过编写驱动的 Makefile 实际上又扩展了内核源代码的 Makefile 文件。在本小节将对 Linux 的编译系统进行更多的介绍，以了解如何把一个驱动代码添加到内核源文件中，可以在编译内核的同时被编译，甚至静态链接到内核中。

7.2.1 内核的 Makefile

总体来说，内核的编译系统非常复杂，除了数量众多的 Makefile 文件，还包括很多其他的配置文件、脚本文件、甚至是 C 语言的源文件。我们很少需要改变内核编译系统的核心部分，多数内核任务都仅需要对编译系统进行非常少的修改。内核的编译系统主要包含下面几个部分

1. 顶层的 Makefile 文件
2. 记录内核配置信息的.config 文件
3. 不同平台的 Makefile 文件：arch/$(ARCH)/Makefile
4. 具体的规则：scripts/Makefile.*
5. 其他配置 Makefile

内核编译系统的核心就是其具体的规则，对于一般的任务，我们只需要知道顶层的 Makefile 和平台 Makefile 负责生成最终的内核与驱动模块，它们包含了.config 文件和内核中的其他 Makefile。

最后一种类型 Makefile 在内核中是最多的，几乎每个子目录中都有一个，它们被称为 kbuild Makefile。这些 Makefile 一般包含格式如下的内容[1]：

```
obj-$(CONFIG_TOUCHSCREEN_AD7877)        += ad7877.o
obj-$(CONFIG_TOUCHSCREEN_AD7879)        += ad7879.o
obj-$(CONFIG_TOUCHSCREEN_ADS7846)       += ads7846.o
```

回顾前面编译内核模块的 Makefile，里面包含的内容就是对 obj-m 的赋值，好像与这里的 Makefile 格式相似。实际上驱动的 Makefile 就是 kbuild 格式，这里讨论的语法将同样适用于普通的驱动。

再看一下内核配置文件.config 文件的内容就会使得问题变得清晰：

```
# CONFIG_TOUCHSCREEN_AD7877 is not set
CONFIG_TOUCHSCREEN_AD7879=m
CONFIG_TOUCHSCREEN_ADS7846=y
```

.config 文件是根据用户的配置自动生成的，里面包含了一系列以 CONFIG_ 开头的配置变量。这些变量的值只能为 “m”、“y” 或者为空。因此在综合了所有 Makefile 之后，编译系统或根据配置情况得到三个目标文件列表。“$(obj-)” 列表内是所有不必编译的目标；“$(obj-m)” 列表内是所有编译为模块的目标；“$(obj-y)” 列表内是所有静态链接到内核的目标。

7.2.2 Kconfig 文件

在内核源码的子目录中，另外一个与配置系统相关的重要的文件就是 Kconfig 文件。这个文件的内容决定了我们在配置内核时所看到的配置选项。在配置内核的时候，可以在终端看到如下的提示信息：

[1]这部分代码取自 drivers/input/touchscreen

```
scripts/kconfig/mconf arch/x86/Kconfig
```

mconf 程序通过分析 arch/$(ARCH)/Kconfig 文件来获得可用的配置选项。这个 Kconfig 文件会根据配置情况调用其他目录中的 Kconfig 文件，共同组成一个配置信息的“数据库”。一条典型的配置信息在 Kconfig 文件中具有如下的格式：

```
config TOUCHSCREEN_ADS7846
        tristate "ADS7846/TSC2046 and ADS7843 based touchscreens"
        depends on SPI_MASTER
        depends on HWMON = n || HWMON
        help
          Say Y here if you have a touchscreen interface using the
          ADS7846/TSC2046 or ADS7843 controller, and your board-specific
          setup code includes that in its table of SPI devices.
```

每条配置信息以 config 关键字引导，config 后面的字符串加上 CONFIG_ 的前缀就成为我们在.config 文件中看到的变量名称。CONFIG_* 变量不但可以在 Makefile 中使用，也可以在内核的 C 代码中使用。例如下面的代码中带 _MODULE 后缀的变量被定义则表示相应的配置选项被配置为模块：

```
#if defined(CONFIG_HWMON) || defined(CONFIG_HWMON_MODULE)
        struct attribute_group  *attr_group;
        struct device           *hwmon;
#endif
```

接下来 tristate 命令定义了这条配置有三种状态，分别是“不选择”，“选择”和“作为模块”。命令后面的字符串为在配置界面中显示的提示信息。与这条命令具有类似功能的命令还有：

- bool，只能选择为“是”或者“否”。
- string，可以输入字符串
- int，可以输入数字
- hex，可以输入十六进制数字

depends 命令定义了这条配置的依赖关系。当依赖关系不满足时，这条配置信息根本不会显示给用户。

help 命令用来显示一个帮助信息。

例如我们想在内核中增加一个新的触摸屏驱动，就可以首先把编写好的源文件 test_ts.c 拷贝到 touchscreen 目录中，然后在这个目录的 Kconfig 文件中添加如下内容：

```
config TOUCHSCREEN_TEST
        tristate "test touchscreens"
```

同时在 Makefile 中增加如下内容就可以了。

```
obj-$(CONFIG_TOUCHSCREEN_TEST) += test_ts.o
```

7.3 字符设备驱动的编写

在实际开发中，如果要对一个硬件从头开始编写驱动程序，最方便的做法就是编写一个字符设备类型的驱动。字符设备驱动的接口比较简单，比较稳定，也相对比较独立。使用字符设备接口来编写驱动并不需要对内核代码有非常深入的了解，只要按照比较固定的框架来编写就可以了。

7.3.1 字符设备的注册与注销

一个字符型设备的驱动需要在模块的初始化函数中注册自己，并在清除函数中注销。注册字符设备的过程分为两个过程，首先就是要为设备获取一个或多个设备号，负责这个功能的函数如下：

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

此函数的 first 参数是驱动所对应的设备编号中的第一个，count 参数表示设备编号的个数，因此这个函数可以申请一共 count 个连续的设备编号。设备编号包括主设备号与子设备号两个部分，一般由下面的宏生成：

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

还记得/dev 目录下的设备文件么？这些特殊的文件同样具有主设备号与子设备号的属性，还可以从文件的属性中看出它们是块设备还是字符设备。在本章开始处提到的设备文件与驱动的对应关系就是这个设备编号。

register_chrdev_region 函数的最后一个参数是设备的名称，它将在系统的 proc 目录中的 devices 文件中显示出来。

需要注意的是这个函数可能运行失败（例如申请的设备号已经被其他驱动申请），因此在程序中要对这个错误进行检验。下面是一个典型的设备号注册代码：

```
dev_t id;
id = MKDEV(MAJOR_NUM, 0);
result = register_chrdev_region(id, 1, DEVICE_NAME);
if (result < 0) {
        printk("my device: unable to get a major %d. \n", MAJOR_NUM);
        return result;
}
```

注册字符设备的第二个步骤就是用 cdev_add 函数将一个 cdev 结构体注册到内核。这个结构体包含了一个重要的参数 ops，它是类型为 struct file_operations 的一个结构体，它的内容决定了内核如何对用户程序操作设备文件的动作进行反应。关于 ops 参数会在后面详细介绍，我们先来看一下 cdev_add 函数的定义：

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

这个函数的 num 和 count 参数与 register_chrdev_region 函数一样，定义了设备号的一个连续范围。cdev 结构体是每个字符设备都具备的数据结构，它一般由 cdev_alloc() 函数进行初始化，具体的注册代码如下示例。其中 my_cdev 为指向 cdev 结构体的指针，id 参数为 MKDEV 生成的设备编号。

```
my_cdev = cdev_alloc();
my_cdev->ops = &mydev_ops;
result = cdev_add(my_cdev, id, 1);
if (result < 0) {
        printk("my device: unable to add the device.\n");
        unregister_chrdev_region(id, 1);
        return result;
}
```

如果注册失败还应该取消前面申请的设备号码，这个函数的定义如下，通过函数的名字与参数，无须对它进行过多解释就可以了解其用法。

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

当注销驱动程序的时候，除了要调用上面的函数，还要调用 cdev_add 的“反函数”：cdev_del。

```
    cdev_del(my_cdev);
    unregister_chrdev_region(id, 1);
```

在很多驱动程序中，我们可以看到 cdev 结构体作为一个元素保存在自定义的结构体内，例如：

```
struct my_data {
        int flag;
        struct cdev my_cdev;
        ...
};
```

此时在初始化 my_cdev 的时候就要用到 cdev_init 函数：

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

7.3.2 重要的数据结构

file_operatoins 是 Linux 字符设备驱动体系中最重要的一个结构体，完整的文件操作接口定义在 linux/fs.h 文件中。依 Linux 版本的不同，这个结构体可能会有一定的区别，它包含了一系列的函数指针，下面是一个主要函数的列表：

```
struct file_operations {
    struct module *owner;        // 文件的属主
    loff_t (*llseek) (struct file *, loff_t, int);        // 修改文件的当前读写位置
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); // 文件读操作
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t); // 仅用于目录操作
                // 查询设备的读写状态
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
                // ioctl 提供了执行设备特定命令的方法
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

```
    // 映射设备内存到进程地址空间
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *); // 打开文件
    int (*flush) (struct file *);         // 文件关闭之前调用的操作,应进行收尾工作
    int (*release) (struct inode *, struct file *); // 关闭文件
        // 刷新待处理的数据
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);  // 异步通知设备的 FASYNC 标志
    int (*lock) (struct file *, int, struct file_lock *); // 文件加锁
……
};
```

2.6.33 版本的内核在 file_operations 结构体中包含了 24 个函数指针。这个结构体包含了这么多的接口只是为了满足全部设备和文件的不同操作,在设计驱动程序的时候,没有必要把每个接口都实现,实际大部分设备也都只实现了不到一半的函数。没有实现的接口系统会进行默认的处理,比如默认的 open 函数将永远返回成功。

这个结构体的赋值一般采用如下的形式:

```
struct file_operations mydev_ops = {
      .owner       = THIS_MODULE,
      .write       = mydev_write,
      .ioctl       = mydev_ioctl,
      .open        = mydev_open,
      .release     = mydev_release
};
```

一个驱动的 open 接口一般要进行初始化设备、检查次设备号和对作为参数传入的 file 结构体进行需要的设置。这个接口一般是用户操作设备文件所执行的第一个操作,但内核允许将这个接口设置为 NULL。如果驱动没有声明 open 接口,设备文件打开总是成功,只是驱动程序无法得到通知。

open 接口的第二个参数是一个指向 file 结构体的指针。struct file 是一个非常重要的内核数据结构,它包含了这个设备被打开时候的一些基本信息,如访问权限、打开参数等。所有的文件操作接口都有它作为参数,习惯上被称为 filp。对于驱动程序比较重要的是它的 f_op 指针和 private_date 指针。f_op 指针就是前面注册驱动时指定的 file_oprations 结构体,在 open 接口函数中可以对这个指针进行修改,所作的修改在 open 函数返回时生效。private_data 指针是 file 结构体中比较常用的变量,它在设备刚被打开的时候是 NULL,可以完全由驱动程序使用或者忽略。一般可以用它保存一些在设备打开期间所使用的数据,这些数据可以在驱动的其他接口中使用。

驱动程序的 open 函数定义与用户空间的 open 系统调用完全不同。其他接口函数如 read、write 等也都与用户空间的系统调用不同。虽然这些接口函数的名字与对应的系统调用相同,但用户空间执行相应系统调用的时候并不是直接调用这些接口函数。内核代码会把用户空间传递过来的参数重新组织,并使用新的参数调用正确的驱动接口函数。

另外一个重要的数据结构是 inode 结构体,它是 open 接口的第一个参数。inode 结构体包含了很多有关文件系统的信息,而与驱动程序相关的元素主要有 i_rdev 和 i_cdev。前者的类型是 dev_t,其内容是打开设备的实际设备号。后者的类型是指向 struct cdev 的指针,其内容是设备文件对应驱动的 cdev 结构体。为了方便地从 inode 结构体获得设备文件的设备号,内核提供了两个宏来获取设备的主设备号与子设备号:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

驱动的 release 接口执行的是 open 接口的相反操作，例如在 open 接口中初始化了一个指针，就要在 release 中释放它占用的内存。

7.3.3 内核数据和用户数据的交换

如果用户程序要和硬件传递数据，就很可能要使用 file_operations 结构中的 read 和 write 接口。这两个接口的定义很类似，实现方法也差不多，下面就通过 write 来进行介绍。

```
ssize_t (*write) (struct file *filep, const char __user *buf, size_t count, loff_t *f_pos
```

它的第一个参数是 file 结构体，第二个参数是从用户空间传递过来的指针，第三个参数为传递数据的字节数，第四个参数指示的是文件当前的读写位置。buf 参数前面的 ___user 修饰符说明这个指针来自用户空间，而直接访问用户空间的指针是不安全的，因为用户空间运行的程序有可能是恶意的，驱动程序必须判断指针的合法性。为了安全地把用户空间的数据传递到内核空间，需要做的工作很多，内核提供的 copy_from_user 函数就专为这一目的而设计。与之对应的还有 copy_to_user 函数，这两个函数的定义如下：

```
unsigned long copy_to_user(void __user *to,
              const void *from, unsigned long count);
unsigned long copy_from_user(void *to,
              const void __user *from, unsigned long count);
```

这两个函数可以正确地在内核与用户空间传递数据，如果用户空间的指针不合法就不会有数据传递发生。两个函数的返回值为还没有拷贝成功的字节数，因此调用代码中需要对这个返回值进行检查。下面是使用 copy_from_user 函数的示例：

```
unsigned char *kbuf=kmalloc(BUF_SIZE, GFP_KERNEL);
if (!kbuf) return -ENOMEM;
if ((err = copy_from_user(kbuf, buf, size)) != 0)
        return -EFAULT;
```

这个程序段首先用 kmalloc 函数在内核中开辟了一个大小为 BUF_SIZE 的缓冲区，关于 kmalloc 函数将在后面进行介绍。然后通过 copy_from_user 函数将用户数据移动到内核中来。如果传递数据发生错误，需要返回 -EFAULT 给上层的应用。

read 和 write 函数在发生错误的时候需要返回一个负数，如果在读写操作中没有发生错误，write 和 read 函数都要返回已经传递的字节数。代表数据读写位置的 f_pos 参数也应相应更新，因为驱动的 llseek 接口需要 f_pos 参数的正确更新才可以工作。在有些驱动中并不支持数据的定位，例如串口数据是实时传输的，在实时数据流中定位是没有意义的。对于这种不支持数据定位的设备驱动，在 open 接口中需要调用如下函数：

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

这样就可以在内核层次禁止用户对此设备进行定位操作，驱动程序也不必实现 llseek 接口。

7.3.4 ioctl 接口

在很多情况下，读写操作并不能方便地完成硬件的特定功能，比如设定串口的传输速率，弹出光驱中的盘片等，这时候就会用到 ioctl 这个系统调用。内核驱动的 ioctl 接口定义如下：

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

这个接口的第三个参数代表一个从用户空间传递过来的命令编号，在驱动程序中常用一个常量来表示。在这个接口的实现中一般会采用一个 switch 语句对不同的命令分别进行处理。驱动程序通常在一个头文件中用宏来定义所用到的命令编号，用户空间的程序同样可以使用这个头文件定义的宏来调用 ioctl 系统调用。

虽然命令编号是驱动程序自己定义的，但按照惯例都会定义一些在系统中唯一的数值作为命令编号。这样可以避免把一个命令对应到错误的设备上，错误的操作会得到一个 EINVAL 的错误消息，而不是对设备进行了错误的操作。

命令编号在内核中有一定的生成规则，将编号定义成四个部分，分别包含 “类型码”（8 位）、“序号”（8 位）、“传输方向” 和 “传输数据的字节数”。类型码在一个驱动中是不变的，而一般在不同的驱动中是不同的，所以常可以使用设备的主设备号来填充。传输方向可能的取值是 _IOC_NONE（无数据传送），_IOC_READ，_IOC_WRITE，或者是后两者的逻辑或。传输方向是相对用户程序表示的，例如 _IOC_READ 表示数据从内核传递到用户空间。

<linux/ioctl.h> 中提供了一些宏可以帮助驱动程序定义 ioctl 的标号：

```
_IO(type, nr)                   没有附加参数的 ioctl
_IOR(type, nr, datatype)        有读参数的 ioctl
_IOW(type, nr, datatype)        有写参数的 ioctl
_IORW(type, nr, datatype)       有读写参数的 ioctl
```

type 所代表的是前面提到的类型码，内核的类型码列表可以在 Documentation/ioctl-number.txt 文件中查看。nr 则代表在这个驱动中的命令序号，可以从 0 直到 255。dataitem 代表实际操作的变量，系统可以通过 sizeof 函数得到在这个命令中实际传递的数据大小。在传递的参数为指针的时候，仍然要使用 copy_from_user 或 copy_to_user 函数复制指针所指的内容。

下面是两个 ioctl 编号的定义例子：

```
#define IOCTL_SET_ITEM  _IOW(MAJOR_NUM, 0, char *)
#define IOCTL_CLR_USE   _IO(MAJOR_NUM, 1)
```

内核驱动在实现 ioctl 接口的时候经常需要对命令编号进行解码，内核同样提供了一些宏来实现这个功能：_IOC_TYPE(nr)、_IOC_NR(nr)、_IOC_DIR(nr) 和 _IOC_SIZE(nr)，分别用来获得编号的类型、序号、方向和字节数。

在 ioctl 系统调用中，实际传递的数据长度可能会比较小，这时候可以使用在 asm/uaccess.h 里面定义的下面两个宏：

```
put_user(datum, ptr);
get_user(local, ptr);
```

这两个函数在执行速度上会比 copy_xx_user 函数更快，所以更适合单个数据的拷贝。具体拷贝数据的字节数与 ptr 的类型相关。如果 ptr 是指向 char 类型的指针，则将传递 1 个字节的数据。

put_user 与 get_user 会在传递数据前检查指针的合法性，如果指针不合法将会返回错误 -EFAULT。如果需要尽一步的提高效率，可以先调用 access_ok 函数检查指针的合法性，之后使用不检查指针的数据传递函数。这几个函数或宏的定义如下：

```
int access_ok(int type, const void *addr, unsigned long size);
__put_user(datum, ptr);
__get_user(datum, ptr);
```

access_ok 的第一个参数可以是 VERIFY_READ 或 VERIFY_WRITE，但这个方向是相对于内核说的。addr 参数为要检查的用户指针，size 参数为数据的字节数。具体在实现 ioctl 接口时常见的代码见下面的代码列表：

```
if (_IOC_TYPE(cmd) != MAJOR_NUM) return -ENOTTY;  // 检查类型码
if (_IOC_NR(cmd) > MY_IOC_MAXNR) return -ENOTTY;  // 检查序号
f (_IOC_DIR(cmd) & _IOC_READ)              // 检查读操作指针
    err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)  // 检查写操作指针
    err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;
switch(cmd) {
    case IOCTL_SET_ITEM:
        __get_user(item, (int __user *)arg);
......
```

7.3.5 内存资源的访问

驱动程序的一个很重要的功能就是对硬件资源进行访问。ARM 芯片的外设已经把它们的 I/O 资源映射到一个固定的内存空间，在它们的驱动程序中需要通过对这些物理地址的读写来控制外设的行为。可实际的物理地址在 Linux 中是不可以直接访问的，必须通过页面映射机制把物理地址映射到线性地址的一个页面中才可以进行访问。这可以通过调用 ioremap() 函数来实现。

```
void __iomem *ioremap(phys_addr_t offset, unsigned long size)
```

例如要修改调试 LED 的状态就可以把它对应的物理地址 0x10000010 通过 ioremap() 函数映射到线性地址中，通过对得到的线性地址的访问间接地修改了相应的物理地址的内容。

```
cpld_base = ioremap(0x10000000, 0x01000);
*((unsigned long *)(cpld_base + 0x10)) = DLED_VALUE;
```

这里使用 ioremap 函数的时候，第一个参数采用了调试 LED 对应物理地址页的边界，映射的尺寸也正好是一个页的大小。这是为了兼容性考虑的，在 2.6 版本的内核中 ioremap 函数没有这样的限制。

上面的用法还有一个移植性的问题。ioremap 函数是把 IO 地址空间重新映射，对于一些平台，这些地址不能作为指针来使用，而必须使用如 readb 等特殊的 IO 空间操作函数。不过由于 ARM 平台没有这个限制，我们为了方便直接使用了指针。

使用 ioremap 函数映射的物理地址需要使用 iounmap 函数释放，在驱动程序中可以在卸载函数中取消内存映射，例如：

```
iounmap((void *)(cpld_base));
```

实际上对于比较大并且连续的内存块，目标板往往在内核启动的时候就预先分配了虚拟地址，这个定义是静态的，可以从平台相关文件中可以看到这个分配表，例如：

```
static struct map_desc my_cpld_desc[] __initdata = {
        {
                .virtual        = 0xFEF70000,
                .pfn            = __phys_to_pfn(0x10000000),
                .length         = 0x100,
                .type           = MT_DEVICE
        }
};
```

这个结构体中，定义了物理地址 0x10000000 的虚拟地址为 0xfef70000，内核驱动将在适当的地方调用 iotable_init 函数使这个映射生效。这样在访问这些物理地址的时候就可以直接对虚拟地址操作了。静态的映射一般只在平台初始化代码中使用，很少用在独立的驱动中。例如 ARM 处理器的外设寄存器空间经常通过这种方式进行映射，这会给需要经常使用处理器外设的硬件驱动带来方便。对于修改调试 LED 的显示就可以直接修改它在内核中对应的虚拟地址：

```
#define DLED_BASE 0xfef70010
*(unsigned long *)DLED_BASE = DLED_VALUE;
```

可以用来进行静态映射的虚拟地址并不是随意的，根据 ARM 平台的虚拟地址布局[1]，可以被平台代码使用的虚拟内存区域为从 VMALLOC_END 开始到 0xFEFFFFFF 的相对很小的区域。VMALLOC_END 是平台相关的一个值，一般不会设置的比 0xFEFFFFFF 低很多。对于 2.6.33 版本内核的 AT91SAM9261 平台，VMALLOC_END 的值为 0xFEE00000，而芯片本身已经映射了从 0xFEF78000 到 0xFEFFFFFF 的空间[2]。

如果需要在内核中动态申请内存，最常用的函数就是 kmalloc。它与 C 语言中的 malloc 函数类似，但不能简单地把它理解为 malloc 函数的内核版本，它与 malloc 相似的仅仅是名字与编程目的。对应的释放由 kmalloc 分配内存的函数是 kfree。这两个函数的定义如下：

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

内核对内存的管理策略使得 kmalloc 实际分到的内存至少是 32 或 64，最大是 128K[3]，size 参数一般选取比 2 的某次幂稍小的值，如果申请了与 2 的幂过于接近的值可能会造成接近一半的内存浪费。使用 kmalloc 申请的内存是没有初始化的，需要的时候应该用 memset 函数初始化。

flags 参数最常用的值为 GFP_KERNEL，表示普通的内核内存分配。它最显著的特点是在内存不足的时候会引起缺页中断，这将使引发 kmalloc 调用的进程进入睡眠。这一特性也使得这个标志不能用在原子操作中，调用 kmalloc 的函数也必须是可以重入的。

如果必须在原子操作中申请内存，或者在没有相关联进程的上下文中申请内存（例如在中断服务程序或内核线程中），就需要使用另外一个标志：GFP_ATOMIC。使用这个标志的 kmalloc 函数不会睡眠，但在内存不足的时候可能会用掉最后一页内存。

7.3.6 互斥与信号量

Linux 作为抢占式的多任务操作系统必须在内核代码中实现有效的同步与互斥的方法，以避免可能的竞争条件。所有的竞争条件都是由资源的共享造成的，因此只要是在驱动中访问

[1] 参考内核文档 arm/memory.txt
[2] 参考代码 arch/arm/mach-at91/at91sam9261.c
[3] 这个尺寸部分出于可移植性的考虑，实际上如果要申请 KB 量级的内存，最好使用 get_free_page 类的函数

全局变量或者其他共享资源都要小心，多数情况下需要在临界区内来完成对资源的操作。经典的临界区实现方法是采用信号量来完成互斥，在 Linux 内核中，信号量的数据结构为 struct semaphore，定义在 asm/semaphore.h 文件中。初始化这个结构体可以使用下面的函数：

```
void sema_init(struct semaphore *sem, int val);
```

Linux 的信号量作为互斥量使用，也就是只有两个值，0 代表获得锁，1 代表释放锁。为了方便，内核提供了两个宏用来声明并初始化互斥的信号量：

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

Linux 内核中操作信号量的函数为：

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

如果信号量的值大于 0，down 函数对信号量的值减 1，并进入临界区。如果信号量的值为 0，down 函数会进入睡眠。down_interruptible 函数是 down 函数的可中断版本，一般在代码中都会使用可中断版本，除非有足够的理由让进程无法中断。这里的“可中断”指的是当进程进入睡眠的时候，可以被信号所唤醒。例如用户可以使用 kill 命令把一个睡眠的进程结束。

使用 down_interruptible 要注意检查返回参数，只有当其返回 0 时表示真正进入了临界区。down_trylock 函数不会进入睡眠，当信号量的值为 0 时，它会立即返回一个不为 0 的错误值。up 函数用来给信号量加 1，它不会进入睡眠。

关于睡眠这里需要再明确一下。如果内核代码是执行在用户空间的上下文之内，即内核代码被一个用户空间进程所调用，那么 down 函数的睡眠就意味着这个用户进程进入睡眠状态。因为这个用户进程不能获得特定的信号量资源，就被从内核的就绪进程列表中去除，如果它没有被唤醒就不会被内核调度。

在硬件中断服务程序中，内核代码并不运行在用户进程的上下文之内，因此也就不能进入睡眠。这个概念非常重要，在编写内核代码的时候一定要注意什么时候可以睡眠，什么时候不能睡眠，否则就可能造成内核的死锁。

另外一个需要注意的问题是在获取信号量之后要避免进入睡眠。因为这会使得其他需要这个信号量的进程同样进入睡眠，而且非常容易引起内核的死锁。内核有很多函数都会引发睡眠，例如曾经介绍过的 kmalloc 和 copy_to_user，这些操作内存的函数都可能由于内存不足或者缺页而进入睡眠。

除了 semaphore，Linux 还提供了另外一种互斥机制，称为自旋锁（spinlock）。自旋锁一般由一个二进制位来实现，想获得锁的代码将不断检查这个位，当没有其他进程拥有这个锁时，这个二进制位处于清除状态，此进程将获得这个锁并设置这个二进制位。测试与设置的操作为原子操作，不能被中断。

自旋锁一般应用在不允许睡眠的环境中，由于它采用死循环的方式检测锁的状态，因此进程拥有锁的时间都尽可能短，否则就会大大影响系统的性能。在内核代码中初始化自旋锁的方式有两种，一种是静态初始化，一种是动态的设置：

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
void spin_lock_init(spinlock_t *lock);
```

使用自旋锁需要在代码中包含 linux/spinlock.h，获得锁与释放锁的调用函数定义如下：

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

这里有四个获得锁的函数调用与四个对应的释放锁的调用。其中 spin_lock 是最简单的，它并没有考虑中断服务的情况。spin_lock_irqsave 函数在获得锁的同时还关闭了中断，并把当前的中断状态保存在了 flags 参数中。spin_lock_irq 在获得锁的时候也关闭了中断，不过并不保存中断状态，可以在确定中断没有被关闭的情况下使用。spin_lock_bh 在获得锁的同时仅关闭了软件中断，硬件中断仍然可以被响应。

在使用自旋锁和信号量的时候需要非常注意，不小心地使用锁变量很容易引起共享冲突或者内核死锁。要在文档中明确说明一个函数是否需要在获得锁之后调用，也要明确说明在一个函数中将获得哪些锁。良好的文档习惯会给自己或别人带来很大帮助，一段即使是自己写的代码在几个月后也会如同别人的代码一样难懂。如果在获得一个锁变量的时候调用了另外一个需要获取这个锁的函数就直接造成了内核的死锁。

如果在代码中需要获得多个锁，必须要按照一定的顺序获取锁。这个顺序应该在整个内核代码中贯彻，因为任何两段按照不同顺序获取两个锁变量的代码都是潜在的死锁原因。一般的准则是，先获得局部的锁变量，再去获得操作系统核心代码的锁；先获得普通的信号量，再去获得更加危险的自旋锁。

在有些情况下，共享的资源只是一个整型变量。例如用一个变量来记录设备被打开的次数，使用前面介绍的同步机制会显得大材小用。这时可以使用内核提供的另外一个同步数据结构：atomic_t，即原子变量。它定义在 asm/atomic.h 文件中，可以用来保存一个整型的变量，并提供一系列函数可以对这个变量进行原子操作。例如可以做加法、减法、测试等等。初值可以用宏 ATOMIC_INIT 静态的设置：

```
atomic_t v = ATOMIC_INIT(0);
```

其他一些常用的操作主要有：

```
void atomic_set(atomic_t *v, int i); // 设置 v 的值为 i
int atomic_read(atomic_t *v);  // 读出 v 的值
void atomic_add(int i, atomic_t *v);  // 将 v 的值加上 i
void atomic_sub(int i, atomic_t *v);  // 将 v 的值减去 i
void atomic_inc(atomic_t *v);  // v 值加 1
void atomic_dec(atomic_t *v);  // v 值减 1
int atomic_inc_and_test(atomic_t *v); // v 值加 1 并测试结果是否为 0
int atomic_dec_and_test(atomic_t *v); // 测试结果为 0 则返回真，否则返回假
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v); // 将 v 的值加上 i 并返回结果
```

下面的例子利用原子变量实现了设备文件只能同时被打开一次：

```
static atomic_t available = ATOMIC_INIT(1);
static int my_open(struct inode *inode, struct file *filp)
{
    if (! atomic_dec_and_test (&available)) {
```

```
        atomic_inc(&available);
        return -EBUSY;  // 已经被打开过
    }
    ...
}
static int my_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&available); // 释放设备
    return 0;
}
```

如果需要“原子的”操作特定的二进制位，Linux 内核的 asm/bitops.h 中还定义了一些相关的函数。这些函数的用法和前面介绍的原子变量类似，而它们的定义在不同平台会有一些区别。下面列表中的函数在 ARM 平台中第一个参数是 unsigned int，第二个参数是 unsigned long *。

```
void set_bit(bit, addr); // 设置 addr 指针所指元素的二进制位
void clear_bit(bit, addr);  // 清除 addr 指针所指元素的二进制位
void change_bit(bit, addr);  // 改变位 addr 指针所指元素的二进制位
test_bit(nr, void *addr);  // 获得二进制位的值
int test_and_set_bit(nr, void *addr);  // 测试并设置二进制位
int test_and_clear_bit(nr, void *addr);  // 测试并清除二进制位
int test_and_change_bit(nr, void *addr);  // 测试并改变二进制位
```

7.3.7 阻塞 I/O 的处理

在驱动程序执行 read 函数的时候，并没有提到一个常见问题的解决方法：如果数据没有准备好怎么办？如果数据会在一段时间之后到达，就不能简单的返回 0，因为 0 代表到达了数据的末尾。在执行 write 函数的时候也是一样，如果由于写入缓冲区满而不能完成写入也会带来麻烦。因为用户程序会期待一次把数据全部写入，如果只写入了部分内容则可能是发生了磁盘满或者其他严重错误。

解决这个问题的办法就是让调用 read 或者 write 的进程进入睡眠，当可以完成全部读写任务的时候再把进程唤醒，这就是前面章节提到过的阻塞模式。内核驱动经常需要实现阻塞方式的读写操作，在具体介绍阻塞 I/O 之前，还必须详细分析一下内核的睡眠。

虽然前面提到的很多内核函数都可以造成当前运行的进程进入睡眠状态，但睡眠只是这些函数的直接作用，如果需要进程明确在什么时间进入睡眠状态就完全是另外一件事情了。当进程请求的资源没有准备好，就是一个非常好的让进程睡眠的时机。不过让进程睡眠的前提是必须有办法让进程被唤醒，Linux 内核通过一个称为等待队列的数据结构记录睡眠的进程，这个队列记录了所有相关于某事件的睡眠进程，当需要把这些进程唤醒的时候就可以使用这个等待队列。

如果要操作一个队列，只要知道队列的“头”就可以了。在 Linux 内核代码中，等待队列的头是定义在 linux/wait.h 中的结构体 wait_queue_head_t。它可以用宏静态的声明：

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地初始化：

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

Linux 内核通过宏 wait_event 来让当前运行的进程进入睡眠。这个宏有几种变化形式：

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

这里的 queue 就是一个等待队列的头元素，condition 是一个将来会变为“真”的布尔表达式。condition 的值会在进程进入睡眠和唤醒的时候被检查，只有在其值为“真”的时候睡眠的进程才可能被唤醒。condition 的表达式可能会被检查多次，因此每次检查不应该具有副作用。

比较常用的睡眠函数是 wait_event_interruptible，它允许睡眠的进程被信号中断。当进程被正常唤醒的时候，这个函数的返回值为 0，否则就代表睡眠被中断，此时驱动程序应该返回 -ERESTARTSYS 错误。包含 timeout 参数的睡眠函数允许一个最长的睡眠时间，当这个时间过去之后，函数会正常返回 0，而不管 condition 的具体值是什么。

一个睡眠的进程无法唤醒自己，因此唤醒的工作只能交给其他进程或者是中断服务程序。唤醒进程的函数同样需要用到等待队列的头元素，两种常用的唤醒函数定义如下：

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up 函数会唤醒所有使用 queue 进入睡眠的进程，而 interruptible 版本只能唤醒可中断的睡眠。需要注意的是这些进程是否真的会醒来还要依赖于它们的唤醒条件。如果要保证每次唤醒操作仅能唤醒一个在这个队列上睡眠的进程，就必须设置等待队列的 WQ_FLAG_EXCLUSIVE 属性，而且不能使用 wake_up 函数，具体的用法请参考其他资料。

如果应用程序采用非阻塞方式调用 read 接口，filp 结构的 f_flags 标志就会包含 O_NONBLOCK 的设置，这时候如果没有准备好的数据，read 函数只要返回 EAGAIN 就可以了，否则就必须实现阻塞。例如：

```
dev = filp->private_data;  /* 驱动私有数据 */
if (filp->f_flags & O_NONBLOCK)
     return -EAGAIN;
if (wait_event_interruptible(dev->queue,(data_available())))
     return -ERESTARTSYS;
```

7.3.8 硬件中断的处理

中断是硬件在需要获得处理器对它的关注的时候，发送给处理器的一个通知信号。例如串口设备是一个慢速设备，如果它不支持中断，CPU 就必须不断查询串口是否有数据到达。这样就带来一个矛盾：查询间隔过短会浪费大量的 CPU 时间；查询间隔过长就会增加串口的反应时间。采用中断方式就可以很好地解决串口查询的问题，只有当数据到达的时候串口才向 CPU 发送中断，CPU 就可以暂时放下手头的工作，首先处理中断事件，把串口数据读出。串口数据读出之后，CPU 可以返回之前的工作继续执行。

在前面小节介绍的阻塞 I/O 的环境下，经常可以利用硬件中断来唤醒睡眠的用户进程。还是用串口设备来举例，当串口没有数据时，阻塞的读操作将会进入睡眠。串口中断发生时，如果读设备准备好，就可以唤醒睡眠的进程。

很多硬件具有给处理器发送中断的能力，之前的章节对中断的概念有过很多的介绍，这里仅讨论在 Linux 系统中如何处理硬件的中断。Linux 系统中可以定义一个函数对应为中断信号的服务程序，为了把一个特定的中断信号和驱动程序联系起来要使用 request_irq 函数：

```
int request_irq(unsigned int irq,
        irqreturn_t (*handler)(int, void *, struct pt_regs *),
        unsigned long flags, const char *dev_name, void *dev_id);
```

第一个参数是申请的中断号，Linux 系统中每个中断事件有一个唯一的中断编号与之对应。

第二个参数是指向中断服务函数的指针。中断服务程序 handler 接受三个参数，其中第一个参数是中断号；第二个参数是 request_irq 传递过来的 dev_id 指针，这个指针经常用来指示设备的私有数据；第三个参数是进入中断时候的寄存器值，这个参数很少被用到，一般用来监控或调试。handler 的返回值可以是 IRQ_HANDLED 和 IRQ_NONE，前者代表中断被处理，后者表示没有被处理（中断共享的时候内核无法判断具体是哪个驱动负责相应的中断信号，所有申请过这个中断号的服务程序都会被调用，服务程序必须判断是不是自己的设备触发了中断）。

第三个参数是标志位，SA_INTERRUPT 代表这是一个“快速”中断服务程序，表示这个中断服务程序会在中断被禁止的情况下运行。这个只是在少数情况下使用，如计时中断，因此一般不需要设置这个标志，SA_SHIRQ 表示此中断可以共享，SA_SAMPLE_RANDOM 表示这个中断可以为随机数的产生做贡献，也就是说这个中断的触发是不可预期的。

第四个参数是申请中断的设备名，它会在/proc/interrupts 文件中显示出来。

最后一个参数是一个 void 类型的指针，它将被作为参数传递给中断服务程序。这个指针一般用来指向驱动的私有数据，这样在服务程序中就不需要用额外的代码获得这些数据。

request_irq 函数返回 0 值表示调用成功，否则返回一个负值作为错误码。中断被申请成功之后，只要内核发现相应的中断信号就会调用 handler 所指定的处理程序。

与之对应的释放中断拥有权的函数比较简单，定义如下：

```
void free_irq(unsigned int irq, void *dev_id);
```

中断服务程序是由硬件事件所触发，并不像其他的内核程序是由用户程序所调用，因此在设计中断服务程序的时候要有一些限制。显然在中断服务程序中不能操作用户空间的数据，因为根本没有用户进程与之联系。另外就是在中断服务程序中不能睡眠，所有可能引起睡眠的操作都要避免，例如前面提到的 kmalloc 函数必须要使用 GFP_ATOMIC 参数。最后就是在中断服务程序中不能调用 schedule，它是专门用来激活进程切换的内核函数。

中断在 Linux 系统中的发生频率是很高的，由于中断服务程序运行在内核态，所以能否快速地从中断服务程序返回是提高系统性能的关键。一般来说，中断服务程序要尽可能少地占用 CPU 时间，如果一个操作需要比较长的时间才可以完成，就需要把它分解成两个部分，把必须尽快在中断服务中实现的部分称为顶半部（top half），而另一部分可以利用内核的推迟执行功能放到以后的时间来执行，这部分成为底半部（bottom half）。

中断服务程序的顶半部就是使用 request_irq 所注册的处理函数，它所完成的工作一般包括清除中断标记，保存设备数据到内核缓冲区，设定中断服务的底半部。

中断服务程序的底半部用来执行比较耗时、甚至会引起睡眠的操作，内核中常用的底半部实现方法有 tasklet 和 workqueues。它们同样可以用在非中断环境中，需要注意的是 tasklet 代码同样不允许睡眠，只有在 workqueues 中才允许睡眠。

tasklet 是比较方便的推迟执行的方法，定义一个 tasklet 可以用内核提供的宏：

```
DECLARE_TASKLET(name,function,data);
```

其中 name 是 tasklet 的名称，它实际上是一个类型为 struct tasklet_struct 的结构体，很多函数都是以这个结构体的指针作为参数。例如驱动程序在适当的时候通过调用下面的函数可以把 name 所指定的函数加到内核的 tasklet 链表中，它将在之后合适的时间尽快被执行。

```
tasklet_schedule(&name);
```

function 是被设置推迟执行的函数，即这个函数将在随后的时间被执行。函数的格式是：

```
void function (unsigned long)
```

data 是传递给上面函数的参数。

tasklet 在运行的时候同样没有对应的用户进程，为了与硬件中断相区别，它被称为软中断，同样受到中断服务程序的若干限制，例如不能进入睡眠等。tasklet 在使用上有如下的几个特点：

- tasklet 在运行前可以被多次使用 tasklet_schedule 被调度运行，但仅能运行一次。因此在驱动中要充分考虑这种情况，当中断发生的频率比较高的时候，tasklet 被多次调度的几率还是很高的。
- 当系统的负载不重的时候 tasklet 会立即运行，不过永远不会迟于下一个时钟中断。
- tasklet 永远不会在多个处理器上同时运行，而且永远只运行在注册 tasklet 的处理器上。

workqueues 是在 2.5 版本内核中引入的推迟执行方法，它与 tasklet 的主要区别是 tasklet 运行在软中断上下文，没有一个进程与之关联，workqueues 有与之关联的内核进程，而且可以睡眠。

多数情况驱动程序不需要自己建立 workqueues，只要使用内核提供的共享队列就可以了。使用共享队列的时候与 tasklet 的用法类似，首先要声明 workqueues：

```
static struct work_struct your_wq;
INIT_WORK(&my_wq, my_wq_function, &data);
```

然后用 schedule_work(&my_wq) 来让 my_wq_function 函数推迟执行。这个函数的定义如下：

```
void my_wq_function (void *)
```

7.3.9 计时与延时

Linux 支持一种并不太精确的计时方法，其工作原理是基于处理器的时钟中断，时钟系统按照固定的频率触发中断，中断发生的次数就用来作为内核的计时单位。较新版本的内核都允许对时钟中断的频率进行设置，设置的值可以用 HZ 全局变量来访问，而中断的计数值是 jiffies，一般在驱动程序中经常用到的是 jiffies 变量的变化值，用来作为对时间的测量。

比较常用的延时函数是 schedule_timeout，它的定义如下：

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

它允许用户进程睡眠 timeout 个时钟中断的时间。schedule_timeout 函数在调用前需要设置进程的状态，如果睡眠过程没有被信号所中断，函数就会返回 0 值。典型的调用代码如下面列表所示：

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

如果需要等待的时间比较短，少于一个时钟中断的周期，就不能使用基于时钟中断的延时办法。此时可以使用内核中基于循环忙等的延时函数：

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

Linux 系统在启动初始阶段会对系统执行计数循环的速度进行计算。内核根据这个参数就可以知道每个纳秒、微秒或者毫秒能够执行多少次计数循环。这三个函数就是根据这个原理实现的，它们分别用来完成纳秒、微秒和毫秒级的延时。但这种延时操作比较浪费 CPU 资源，一般仅在较短延时的时候使用。实际上内核在同一个头文件中还定义了几个基于睡眠的延时函数，当对延时的要求并不是那么精确的时候可以使用：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

其中 msleep 用来进行毫秒级延时，msleep_interruptible 在延时的时候可以被信号所中断，ssleep 用来提供不可中断的秒量级的延时。

有时候驱动代码希望一个原本包含延时的函数能立即返回，并设定延时后需要执行的代码在一段确定的时间后运行。这个需求有些类似 tasklet 或者 workqueues 的概念，但这两个方法都不能实现确定时间的延时。内核为这个需求提供的函数被称为内核计时器（kernel timer），它的相关函数和数据结构如下：

```
#include <linux/timer.h>
struct timer_list {
        /* ... */
        unsigned long expires;
        void (*function)(unsigned long);
        unsigned long data;
};
void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

timer_list 的结构体可以用 TIMER_INITIALIZER 来静态生成或者用 init_timer 函数来初始化。之后就要设置这个结构体中最重要的三个元素，其中 expires 元素为计时器定时值，内核把它与 jiffies 值进行比较，如其小于或等于 jiffies 值，function 所指定的超时函数就会被调用，这个函数以 data 元素作为输入参数。

当 timer_list 结构体被设置好之后就可以调用 add_timer 函数向内核注册超时函数，在定时点到达之前，驱动程序可以调用 del_timer 函数取消这个计时器。超时函数所运行的环境也是软件中断上下文，同样受到不能睡眠等限制。在超时函数中可以再次对这个超时函数进行注册（当然要相应改变 expires 的值），可以实现一段代码的定时重复执行，例如可以利用这个办法实现对鼠标坐标的定时读取。

7.3.10 其他一些常用技术

驱动程序中定义的各种符号在驱动程序被加载以后是可以被内核使用的，默认情况下驱动程序中所有定义为“非静态”（没有定义为 static）的符号在整个内核空间可见。这种机制允许驱动程序的部分功能被其他驱动调用，称为模块的堆叠技术。一般情况不需要导出符号给其他驱动程序使用，否则还有可能隐含符号名称与其他内核模块发生冲突的危险，除了把符号定义为静态之外，还可以通过宏：

```
EXPORT_NO_SYMBOLS;
```

来显式地禁止导出任何符号。而显式地导出符号可以用宏

```
EXPORT_SYMBOL(symbol);
```

来操作。内核中所有导出的符号列表可以在/proc/ksyms 中看到。

有时候可能需要在加载内核模块的时候给模块传递一些参数，这时可以用宏 MODULE_PARM 来完成。例如定义一个字符串型和一个整型的参数可以采用下面的语句：

```
char *driver_para1;
int driver_para2 = 0;
MODULE_PARM (driver_para2, "i");
MODULE_PARM (driver_para1, "s");
```

MODULE_PARM 的第二个参数表示模块参数的数据类型，目前内核支持的有 b（byte）、h（short）、i（interger）、l（long）、s（string）。对于字符串类型的参数，insmod 会负责分配内存的工作。

下面一个简单的驱动程序代码，利用模块参数访问和设置内核高位内存区的值（这个区域一般作为 I/O 的映射）。

```
static int blr = 0xf0000028;
static int newvalue = 0;
module_param(blr, int, S_IRUGO);
module_param(newvalue, int, S_IRUGO);

static int blr_init(void)
{
        printk(KERN_ALERT "blr:%x, old value: %x\n",
               blr, *(unsigned long *)blr);
        if (newvalue != 0) *(unsigned long *)blr = newvalue;
        return 0;
}

module_init(blr_init);
```

当加载需要参数的模块时，在 insmod 命令的后面要添加参数的赋值部分。例如：

```
# insmod blr.ko newvalue=0xff
```

7.3.11 关键的内核头文件

<linux/module.h> 内核模块程序都要包含这个文件

<linux/uaccess.h> copy_to_user，copy_from_user 等函数的定义

<linux/fs.h> file_operations 结构有关的很多函数都定义在这个头文件里，或者被这个头文件所包含。

<linux/sched.h> 与进程调度有关的函数与常量定义在这个头文件里。

<linux/ioctl.h> 定义 ioctl 编号的宏需要包含这个头文件。

<linux/init.h> 使用 module_init 宏必须包含这个头文件。

<linux/interrupt.h> 使用 request_irq 和 tasklet 需要包含这个头文件。

<asm/irq.h> 包含中断编号的一些常量。

<asm/hardware.h> 目标板的许多常量，如板级寄存器等，都定义在这个头文件中。

7.4 Linux 设备驱动模型

虽然字符设备是最常见的硬件设备类型，但在实际的内核代码（2.6.33 版本）中，真正按照前面规则编写的字符设备非常少。因为从内核的 2.5 版本开始，内核开发人员开始构建统一的内核驱动模型。采用统一的内核驱动模型可以把系统中的设备组织成清晰的拓扑结构，并会在如下方面带来好处：

- 统一的电源管理系统
- 使用 sysfs 虚拟文件系统同用户空间进行接口
- 从总线级别支持设备的热插拔
- 将设备进行分类
- 控制驱动和设备的生命周期
- 统一的设备拓扑描述方式，减少代码的重复
- 可以获得设备的连接方式，对设备进行枚举

为了采用统一的模型描述众多迥异的硬件设备，Linux 内核采用了比较复杂的数据结构。其中最核心的是 kobject 结构体，其次是描述更高层次抽象的 kset 和 subsystem。但在编写驱动程序的时候很少会对底层的数据结构进行操作，因此这里仅介绍比较高层的相关函数，这样就可以很好地理解内核中的大多数驱动程序了。

从/sys 目录中的目录结构就可以对内核的驱动模型有初步的认识。例如其中的 class 子目录包含如 sound、input 等设备分类，从这些子目录中可以找到所有同样类别的设备。在 bus 子目录中又包含如 pic、usb 等总线子目录，在总线的子目录中包含各种属于这个总线的设备与驱动的列表。

/sys 目录中的 bus 与 class 在内核中都是用 subsystem 来描述的，内核驱动一般会通过 subsystem 所提供的注册函数将驱动注册到自己所属的总线和设备类型中去。

对于嵌入式系统来说，很多电路板上的设备并不是连接到通用总线上，而是直接通过数据总线和地址总线连接到 CPU 上。这种类型的设备被称为平台设备（platform device），它也可以看作一种特殊的总线，本小节主要对平台设备的使用进行介绍。

一个平台设备的驱动程序首先要填充一个 platform_driver 结构体。例如在 i2c-gpio.c[1] 中的代码是这样对它进行赋值的：

```
static struct platform_driver i2c_gpio_driver = {
        .driver         = {
```

[1] i2c-gpio 驱动用 GPIO 管脚来实现 I2C 的通信协议

```
                .name   = "i2c-gpio",
                .owner  = THIS_MODULE,
        },
        .probe          = i2c_gpio_probe,
        .remove         = i2c_gpio_remove,
};
```

在驱动程序的初始化函数中，使用 platform_driver_register 函数将其注册到平台设备中：

```
ret = platform_driver_register(&i2c_gpio_driver);
if (ret)
        printk(KERN_ERR "i2c-gpio: probe failed: %d\n", ret);
```

在退出函数中，使用 platform_driver_unregister 函数将驱动注销：

```
platform_driver_unregister(&i2c_gpio_driver);
```

驱动程序在平台中注册之后，并不能立刻对硬件进行初始化和控制。内核代码还必须在其他地方注册相应的设备，并提供驱动运行的相关参数。注册设备的代码一般在具体硬件平台的源文件中。例如在 arch/arm/mach-at91/at91sam9261_devices.c 文件中就包含对 AT91SAM9261 芯片外设的设备注册代码。

驱动程序一般把运行需要的参数组织为结构体，通过这种方式可以提高代码的通用性，避免在把驱动应用到新硬件平台的时候重新修改驱动，而只要更改传递给驱动的参数就可以了。i2c-gpio.c 中所包含的参数决定了使用哪个 GPIO 管脚作为 I2C 的数据线和时钟线，还有对管脚漏级开路的设置和时钟频率的设置。

```
static struct i2c_gpio_platform_data pdata = {
        .sda_pin                = AT91_PIN_PA7,
        .sda_is_open_drain      = 1,
        .scl_pin                = AT91_PIN_PA8,
        .scl_is_open_drain      = 1,
        .udelay                 = 2,            /* ~100 kHz */
};
```

在内核中注册平台设备的函数是 platform_device_register，它的参数是一个 platform_device 的结构体，这个结构体的 dev.platform_data 元素是一个指向驱动参数的指针，内核驱动程序就是这样和其参数建立联系的。例如在 at91sam9261_devices.c 中是这样注册的：

```
static struct platform_device at91sam9261_twi_device = {
        .name                   = "i2c-gpio",
        .id                     = -1,
        .dev.platform_data      = &pdata,
};
...
platform_device_register(&at91sam9261_twi_device);
```

如果需要同时添加多个设备，可以使用 platform_add_devices 函数，它的参数是一个 platform_device 的数组和数组的元素个数，例如：

```
platform_add_devices(devices, ARRAY_SIZE(devices));
```

ARRAY_SIZE 宏用来获得数组的元素个数。

现在把 platform_device 和 platform_driver 结构体联系起来分析，其中 platform_device 中的 name 元素与 platform_driver 中的 driver.name 元素的值相同，内核就是根据这个关系把驱动与具体的设备联系起来的。

Linux 内核中的所有总线系统都包含一个 match 函数，用来匹配这个子系统中的设备和驱动。例如 PCI 总线是根据 PCI ID 来对驱动进行匹配，而 platform 系统可以看作一个虚拟总线，它根据设备和驱动的名字进行匹配。当一个新的设备或驱动被注册到内核的时候，内核就会调用总线的 match 函数，如果匹配成功，就会调用驱动的 probe 函数。probe 函数负责设备的初始化和把设备添加到适当的类别框架中。例如在 i2c-gpio.c 中把设备注册到内核的 I2C 系统中。

如果设备驱动还需要使用一定的硬件资源，例如设备寄存器的地址和设备所使用的中断线，则还需要定义一个资源数组。这种设计使得驱动程序可以不依赖于固定的寄存器地址或中断号。例如 at91sam9261_devices.c 中对 USB 主设备的资源定义如下：

```
static struct resource usbh_resources[] = {
        [0] = {
                .start  = AT91SAM9261_UHP_BASE,
                .end    = AT91SAM9261_UHP_BASE + SZ_1M - 1,
                .flags  = IORESOURCE_MEM,
        },
        [1] = {
                .start  = AT91SAM9261_ID_UHP,
                .end    = AT91SAM9261_ID_UHP,
                .flags  = IORESOURCE_IRQ,
        },
};
static struct platform_device at91sam9261_usbh_device = {
/* 省略部分内容 */
        .resource       = usbh_resources,
        .num_resources  = ARRAY_SIZE(usbh_resources),
};
```

当驱动程序中需要获得资源的分配情况时，可以调用 platform_get_resource 获得内存或者 I/O 的配置，调用 platform_get_irq 获得中断号的配置。例如：

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
irq = platform_get_irq(pdev, 0);
```

7.5 实验 :Linux 驱动程序设计

实验目的

1. 了解 Linux 驱动程序的一般结构
2. 掌握设计 Linux 驱动程序的方法
3. 掌握驱动程序中断的处理方法

实验内容

一、测试 Hello World 驱动

由于实验的操作系统环境是 2.6 的内核，因此采用 2.6 内核的编译方法。在主机上编译 7.1.1 小节中提供的代码，并在实验板上加载测试。

二、编写调试 LED 的驱动

在目标板上有 8 个用于调试的 LED,我们在第三章的实验中已经接触过它。现在在 Linux 操作系统的环境下编写一个驱动,然后再编写一个测试程序用这些 LED 显示特定的数据。

实现此字符设备驱动的 open,release 和 ioctl 接口,实现把 8 位无符号数显示在 LED 上。参考程序如下所示:

```
#include <linux/module.h>   /* Specifically, a module */
#include <linux/init.h>
#include <linux/fs.h>       /* The character device definitions are here */
#include <linux/cdev.h>
#include "cpld.h"

#define CPLD_BASE 0xfee70010
// static unsigned long cpld_virtual;
static struct cdev *my_cdev;

static int cpld_open (struct inode *inode, struct file *filp)
{
        return 0;
}

static int cpld_release (struct inode *inode, struct file *filp)
{
        return 0;
}

static int cpld_ioctl(struct inode *inode, struct file *file,
                unsigned int ioctl_num, unsigned long ioctl_param)
{
        switch (ioctl_num) {
                case IOCTL_SET_VALUE:
                        *(unsigned char *)(CPLD_BASE) = ioctl_param & 0xff;
                        break;
        }
        return 0;
}

struct file_operations cpld_ops = {
        .owner          = THIS_MODULE,
        .ioctl          = cpld_ioctl,   /* ioctl */
        .open          = cpld_open,
        .release = cpld_release  /* a.k.a. close */
};

static int __init cpld_init(void)
{
        int result;
        dev_t id;

        id = MKDEV(MAJOR_NUM, 0);
        result = register_chrdev_region(id, 1, DEVICE_NAME);
        if (result < 0) {
                printk("cpld: unable to get a major %d. :(\n", MAJOR_NUM);
                return result;
        }
        my_cdev = cdev_alloc();
        my_cdev->ops = &cpld_ops;
```

```
        my_cdev->owner = THIS_MODULE;
        result = cdev_add(my_cdev, id, 1);
        if (result < 0) {
                printk("cpld: unable to add the device.\n");
                unregister_chrdev_region(MKDEV(MAJOR_NUM, 0), 1);
                return result;
        }
        return 0;
}

static void __exit cpld_cleanup(void)
{
        cdev_del(my_cdev);
        unregister_chrdev_region(MKDEV(MAJOR_NUM, 0), 1);
}
module_init(cpld_init);
module_exit(cpld_cleanup);
MODULE_LICENSE("GPL");
```

上面的代码不包含 cpld.h 文件，这个文件给出了一些常量和 IOCTL_SET_LED 的定义。读者需要自己添加必要的代码。

驱动程序编好后，在/dev 目录建立设备文件与之连接，试着增加 write 函数的实现，这样就可以简单地通过 echo 命令把字符串数据输入重定向到这个设备中，对驱动的写操作进行检测。

用 mknod 命令建立一个 cpld 设备，主设备号为 101，从设备号为 0：

```
# mknod /dev/cpld c 101 0
```

最简单的测试方法是用如下命令：

```
# echo -e -n "\x34\x56" > /dev/cpld
```

这条命令把十六进制数 “3456” 输入到 cpld 设备中，可以通过改变参数的内容，观察调试 LED 的状态。

如果要测试 ioctl 函数的实现，必须要编写一个测试程序，通过 ioctl 系统调用使 8 个发光二极管按一定规律点亮。成功之后，可以再修改驱动程序，不使用固定的虚拟地址，而只把 LED 看成物理地址为 0x10000010 的内存单元，使用 ioremap 函数映射新的虚拟地址来实现上面同样的功能。

三、编写按钮的驱动（选做）

编写开发板上按钮的字符设备驱动程序，支持阻塞读操作。在没有按键按下的时候，应用程序的读操作被阻塞，用按键的中断信号把睡眠的读进程唤醒，否则直接读出按钮的键值。其中，中断服务程序的示例如下：

```
#include <linux/module.h>   /* Specifically, a module */
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <mach/gpio.h>


#define CPLD_BASE 0xfee70010

static unsigned char key_value = 0;
```

```
static irqreturn_t btn_isr(int irq, void *dev_id)
{
        key_value++;
        *(unsigned char *)(CPLD_BASE) = key_value;
        return IRQ_HANDLED;
}

static int __init btn_init(void)
{
        int err;
        at91_set_gpio_input(AT91_PIN_PA30, 1);
        at91_set_deglitch(AT91_PIN_PA30, 1);
        err = request_irq(AT91_PIN_PA30, btn_isr, IRQF_SAMPLE_RANDOM |
                IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "btn", NULL);
        if (err) {
                printk(KERN_ALERT "Unable to request irq\n");
                return err;
        }
        return 0;
}

static void __exit btn_cleanup(void)
{
        free_irq(AT91_PIN_PA30, NULL);
}
module_init(btn_init);
module_exit(btn_cleanup);
MODULE_LICENSE("GPL");
```

第八章　嵌入式 Linux 图形编程

8.1　常见的图形编程工具

与 Windows 操作系统不同，在 Linux 环境中的图形用户界面（graphic user interface，GUI）是独立的应用程序，用户可以自由选择各自的桌面环境。在台式机系统中，人们倾向于采用功能比较强大的 X Window 系统。X Window 是平台无关、网络透明、应用可扩展的图形系统，但它只提供了底层的图形接口，直接面向 X Window 编程十分复杂，很难实现比较复杂的应用界面。因此，图形界面开发程序员会选择使用更高层的程序库来编程。最常用的就是 Gtk 和 QT，前者是 GNOME 桌面环境的基础类库，后者被用来开发 KDE。

对于嵌入式系统环境来说，存储空间和内存空间的限制使得上面介绍的图形系统和图形库都不太适合嵌入式应用。嵌入式系统对 GUI 系统提出的要求主要有以下几个方面：

1. 轻型、占用资源少；
2. 高性能；
3. 高可靠性；
4. 可配置。

当视野从桌面转移到嵌入式平台时，图形系统可能的选择甚至更多。不但 Gtk 和 QT 都有面向嵌入式的版本，DirectFB、MiniGUI、Microwindows、Tiny-X 等，都是不错的嵌入式图形库，都有非常广泛的应用。

自从 Linux 内核开始支持 FrameBuffer，应用程序可以直接通过内核接口来控制屏幕的图形显示。这就为开发轻量级的图形系统提供了方便条件，大部分的嵌入式图形系统也都采用了这种技术。

Tiny-X 是 X Window 的一个嵌入式实现，它的最大特点是支持与桌面 X Window 系统一样的强大功能，而其所占用的系统资源却相对小很多。如果采用 Tiny-X 作为图形环境，平时常见的桌面应用就都可以比较容易地移植到嵌入式平台。其主要的缺点是直接面向 X 编程比较复杂，还需要其他的图形库支持。而且复杂的 X 协议也是多数嵌入式环境所不需要的。

MiniGUI 最初由清华大学的学生开发，后来成立了“飞漫”软件公司，是中国人做得比较早也比较好的自由软件之一。它的主要特色有：

1. 遵循 LGPL 条款的自由软件
2. 提供完备的多窗口机制
3. 消息传递机制
4. 多字符集和多字体支持，对汉字的支持比较好
5. 全拼、五笔等汉字输入法的支持
6. BMP、GIF、JPEG、PCX 等常见图像文件的支持
7. Windows 资源文件的支持
8. 小巧。包含全部功能的库文件大小为 300K 左右
9. 可配置性好，可移植性好，高性能，高稳定性

Qt 是由 Trolltech 公司开发的用于嵌入式 Linux 的图形用户界面系统。Trolltech 最初是作为用于 Linux 台式机的跨平台开发工具而创建 Qt 的。它支持各种 UNIX 的操作系统以及 Microsoft Windows。为了占领嵌入式市场，Trolltech 又推出了 Qt Embedded，它以原始 Qt 为

基础，并做了许多出色的调整以适用于嵌入式环境。与其他 GUI 系统相比，Qt Embedded 具有以下优点：

1. Qt Embedded 占用较少的内存和其他资源，是为嵌入式系统开发量身定做的。
2. Qt Embedded 面向对象的体系结构使代码结构化、可重用性更好。Qt GUI 软件结构没有分层，这使得 Qt Embedded 成为用于运行基于 Qt 的程序的最紧凑环境。
3. Qt Embedded 提供的 API 与普通台式机完全兼容，并且可以良好运行在不同的嵌入式处理器上和不同的桌面环境下，使得基于它的上层应用程序开发非常方便可靠，并具有很好的可移植性。

目前 Trolltech 公司已经被 Nokia 公司收购，Qt 软件已经开始版本 4.7 的开发。在 Qt 的发展历史上，曾经衍生出许多种版本，这里简要说明一下它们之间的关系：

1. Qt 是所有 Qt 软件版本的总称，一个特定的 Qt 软件版本在发行文件名上往往用一个多元组来表示，例如本书使用的 qt-everywhere-opensource-4.6.3。
2. Qt 软件分为商用版和开源版，开源版的名称中一般带有 opensource 或者 free 单词。
3. Qt 软件包含面向不同平台的发行版，名称中带有标示平台的单词，如 win/x11/mac/wince 等。
4. Qt 的嵌入式版商业上曾经被称为 Qt/Embedded，Qtopia Core 和 Qt Embedded，其文件名会带有 embedded 或者 `qtopia-core`。
5. Qt 的 win/x11/mac/embedded 版本最后融合到了一起，被称为 Qt Everywhere。

还有一个基于 Qt 的开源项目 Qt Palmtop Environment（QPE），它包含了一整套商务生产、个人信息管理、网络应用和娱乐的应用程序，并且为开发者提供了用于编写嵌入式设备应用程序的面向对象的应用程序接口，从而在嵌入式 Liunx 操作系统的基础上搭建起了一个完整的视窗系统和应用程序开发平台。Trolltech 公司将其发展成为一个非常成功的产品：Qtopia，甚至将 Qt/Embedded 都改名为 Qtopia Core，图 8.1 就是 Qtopia 的应用界面。Qtopia 在 Qt 4.4 版本推出的时候将名字改为了 Qt Extended（Qtopia Core 也同时改名为 Qt Embedded），不过之后就停止了 Qtopia 的开发，而把其部分功能融入了 Qt Embedded。

图 8.1: Qtopia 视窗系统界面

8.2 Qt 编程入门

8.2.1 信号与槽

在图形用户界面编程中，应用程序不再具有非常确定的执行路径，大部分代码都要靠事件来触发。例如用户点击了某个按钮，或者拖动了某个滚动条，这些都是事件，程序员就要对每个这样的事件进行处理，让特定的代码与事件相关联。不同的图形库都有自己处理事件的方案，而在 Qt 中这一机制被称为“信号”和“槽”。

在 Qt 编程环境中，当一个特定事件发生的时候，就有一个对应的信号被“发射”，而可以用

来“接收”和处理这个信号的函数就被称为槽。在 Qt 的类中有很多预定义的信号，用户也可以通过继承来加入自己定义的信号。Qt 类也有很多预定义的槽，但是通常的习惯是加入自己的槽（自定义槽），这样就可以用期待的方式处理我们感兴趣的信号。

Qt 的信号与槽机制具有如下两个特点：

1. 信号和槽的机制是类型安全的

Qt 中的信号同样具有类似于函数的输入参数与返回参数作为自己的特征。因此在连接信号与槽的时候，编译器可以根据这个特征来进行类型匹配。实际上，一个槽的参数个数可以比它接收的信号的参数少，因为它可以忽略额外的参数。

2. 信号和槽是宽松地联系在一起的

一个发射信号的类不用知道也不用注意哪个类的槽要接收这个信号。Qt 的信号和槽的机制可以保证如果你把一个信号和一个槽连接起来，槽会在正确的时间使用信号的参数被调用。信号和槽可以使用任何数量、任何类型的参数。

槽可以用来接收信号，但它们还是正常的成员函数。一个槽不用知道它是否被信号连接，这样就保证了利用 Qt 能够创建真正独立的软件组件。

可以把许多信号和所希望的单一槽相连，并且一个信号也可以和你所期望的许多槽相连。把一个信号和另一个信号直接相连也是可以的。这时，只要第一个信号被发射，第二个信号立刻就被发射。

信号和槽构成了一个强有力的组件编程机制，在 Qt 中使用 connect 函数将信号与槽连接在一起，图 8.2 是一种可能的信号和槽的连接方式。

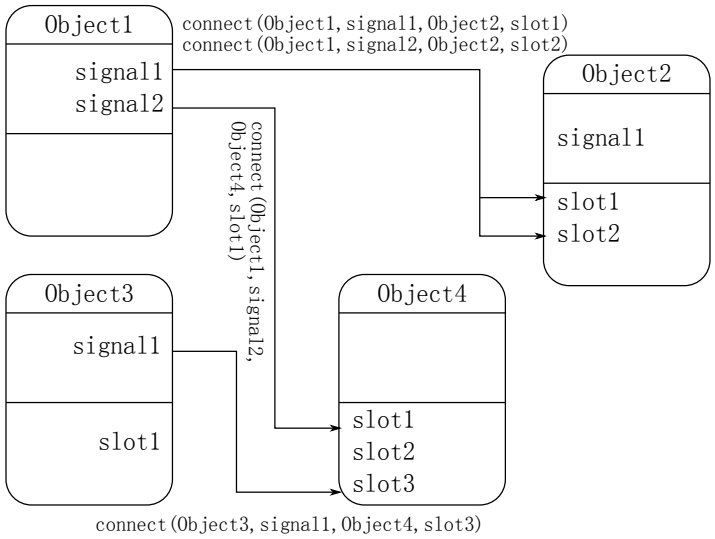

图 8.2: 信号和槽连接的示意图

在 Qt 中，所有以 QObject 作为基类的类都可以使用信号和槽的通信方式，而不管是否是一个图形界面的元素。下面通过一个简单的 C++ 类来说明信号和槽的基本使用方法。

```
class Foo : public QObject
{
    Q_OBJECT
public:
    Foo();
    int value() const { return val; }
public slots:
    void setValue( int );
signals:
    void valueChanged( int );
private:
```

```
    int val;
};
```

这个 Qt 类和一个普通的 C++ 类有着类似的结构，唯一特殊的就是 Q_OBJECT 宏的存在，这个宏的存在使编译器可以正确处理 slot 和 signal 保留字，这样 Foo 才可以使用信号和槽的组件编程。Qt 类通过 signals 保留字标识信号，并用 slots 保留字标识槽，因此类 Foo 具有一个信号 valueChanged 和一个槽 setValue。

槽同一个普通的函数没有本质的区别，可以由应用程序的编写者来实现，下面是槽 Foo::setValue 的一个可能代码：

```
void Foo::setValue( int v )
{
    if ( v != val ) {
        val = v;
        emit valueChanged(v);
    }
}
```

Qt 使用 emit 保留字来发射信号，因此在上面代码中，emit valueChanged(v) 这一行将从对象中发射 valueChanged 信号。

下面是把两个对象连接在一起的一种方法：

```
Foo a, b;
QObject::connect(&a, SIGNAL(valueChanged(int)),
                 &b, SLOT(setValue(int)));
```

a 和 b 是 Foo 类的两个实例，通过调用 connect 函数将 a 的 valueChanged 信号与 b 的 setValue 槽相连接。因此调用 a.setValue(79) 会使 a 发射一个 valueChanged 信号，b 将会在它的 setValue 槽中接收这个信号，也就是 b.setValue(79) 被调用。接下来 b 会发射同样的 valueChanged 信号，但是因为没有槽被连接到 b 的 valueChanged 信号，所以这个信号将被忽略。

注意在代码中只有在 v != val 的时候 setValue 函数才会设置这个值并且发射信号。这样就避免了在循环连接的情况下（比如 b.valueChanged 和 a.setValue 连接在一起）出现无休止的循环。

8.2.2 Qt 版本的 Hello World

我们的第一个 Qt 程序是一个简单的 “Hello World” 例子。它只包含建立和运行 Qt 应用程序所需要的最少代码。图 8.3 是这个程序的屏幕截图。

```
1 #include <QApplication>
2 #include <QPushButton>
3 int main(int argc, char **argv)
4 {
5    QApplication a(argc, argv);
6    QPushButton hello("Hello world!", 0);
7    hello.show();
8    return a.exec();
9 }
```

图 8.3: Hello World 界面

程序最开头的两行是头文件的声明。第 1 行头文件中包含了 QApplication 类的定义。在每一个使用 Qt 的 GUI 应用程序中都必须使用一个 QApplication 实例(程序中第 5 行),这个实例管理了各种各样的应用程序资源,比如默认的字体和光标。我们可以通过命令行参数把一些信息传递给 QApplication,来改变程序的一些行为,如设置程序的外观方案或窗口尺寸。

程序第 2 行的头文件包含了 QPushButton 类的定义。QPushButton 是一个图形用户界面的按钮,在 Qt 世界中,每个图形界面元素都被封装成一个 C++ 类,被称为 Widget,或者 “部件”。每个部件都继承于 QWidget 类,它们是可以处理用户输入和绘制图形的用户界面对象,程序员可以通过它提供的方法改变它的外观属性并实现特定的程序动作。例如 QPushButton 是一个可以显示文本或者图片的按钮,用户通过点击这个按钮触发信号。

Qt 中的每个 Widget 都在独立的头文件中定义,在使用的时候必须包含相应的头文件,就像这里包含 QPushButton 一样。如果使用的 Widget 很多,也可以仅包含 QtGui 这个头文件。QtGui 中已经包含了全部的 Widget 定义。

第 3 行 main 函数是程序的入口。注意这里使用标准 C++ 的 argc 与 argv 函数参数,因为这两个参数将在后面用到。一般情况下 Qt 程序的 main 函数都比较简短,只做一些必要的初始化,然后将程序的控制权转交给 Qt 的事件处理体系。

第 5 行程序定义了 QApplication 的实例,在使用 Qt 的 GUI 元素之前必须首先构建 QApplication 的实例。在实例化 QApplication 的时候,需要把 main 函数的 argc 与 argv 参数传递给它的构造函数,QApplication 将对通过命令行传递的标准 Qt 参数进行处理,并把已经处理的参数从 argv 中去除,argc 的值也将更新。

程序第 6 行创建了一个按钮对象。QPushButton 构造函数的第一个参数是按钮所显示的文本内容,这里设置成 “Hello world!”。它的第二个参数用来设置按钮的 “父类”,这里设置为 0 表示没有父类。

对象的 “父子关系” 是在 QObject 类中实现的,Qt 通过对象的 “父子关系” 可以简化内存的管理,避免不必要的内存泄漏。当一个对象被删除的时候,它所有的子类都会被递归删除,因此程序只要正确设置了它的父类,就不用去为这个对象的删除操心。如果一个 Widget 没有父类,它将成为一个窗口,具有边框和标题栏,可以被用户移动和改变大小。

程序第 7 行调用 show 函数将 hello 按钮显示出来。当你创建一个 Widget 的时候,它是不可见的,程序一般在将所有 Widget 组织好之后才将它们一次显示出来。

程序第 8 行虽然调用了 return,但程序并不会立即返回,因为 QApplication 的 exec 函数会一直运行到用户关闭程序窗口。因此这行程序实际是把控制转交给 Qt 系统。在 exec 函数中,Qt 接受并处理用户和系统的事件并且把它们传递给适当的 Widget。

如果要编译这个 Qt 程序,首先建立一个工程目录,将源文件保存在这个目录中。同时 Qt 提供了一个很好的生成 Makefile 的工具——qmake。只要在工程的目录下运行如下两条命令即可:

```
$ qmake -project
$ qmake
```

前一条命令生成了以工程文件所在目录为文件名的 pro 文件,即工程配置文件。然后下一条命令通过这个配置文件生成最终的 Makefile。此时就可以通过 make 命令生成最终的可执行程序,可执行文件的名字与工程目录相同。例如工程目录的名字为 hello,则自动生成的工程配置文件名称为 hello.pro,最终的可执行文件名称为 hello。

由于 qmake 命令是通过在当前目录及内部的子目录中搜索所有源代码文件并生成工程配置文件的,因此需要把相关的源文件放在单独的工程目录中。而且源文件代码也要使用 cpp 作

为扩展文件名，否则就不会正常获得编译。

正常编译结束后就可以尝试运行程序，还可以尝试标准的 X11 命令行参数，例如使用 -geometry 参数设置程序的显示位置和窗口大小。下面的示例设置 hello 程序显示在坐标（30，100）的位置，窗口的长为 200，高为 50。

```
$ ./hello -geometry 200x50+30+100
```

8.2.3 QWidget 简介

在更加深入地了解 Qt 图形界面编程之前，我们首先要了解一下 QWidget 类。前面已经提到 QWidget 类是所有图形界面部件类的基类，它负责接收系统中的各种事件，并将自己显示在计算机屏幕上。

大部分的 Widget 都是作为其他 Widget 的子类来使用的，毕竟一个程序中窗口的数目会远远小于 Widget 的数目，像 Hello World 例子中把 QPushButton 直接作为窗口的情况非常少见。

所有的 Widget 的构造函数都接受一个 parent 的参数。如果 parent 参数为 0，则这个 Widget 将成为窗口，可以用 setWindowTitle 函数设置窗口的标题，也可以用 setWindowIcon 来设置窗口的图标。如果 parent 参数不为 0，它将成为另外一个 Widget 的子类，只能显示在父类的内部，受到其父类大小的限制。

如果要控制一个 Widget 的显示大小，就要用到 QWidget 的 sizePolicy 属性和 sizeHint 虚函数。sizeHint 函数不需要参数，返回值是 QSize 类，用来表示一个默认的 Widget 尺寸。例如可以这样定义 sizeHint 函数，使得 Widget 的默认尺寸为 100 × 100：

```
QSize MyWidget::sizeHint() const
{
        return QSize(100, 100);
}
```

由于窗口尺寸会被用户改变，QWidget 还需要 sizePolicy 属性决定默认尺寸的含义。这个属性包含对 X 方向和 Y 方向两个参数的设置。例如可以设置 X 方向的默认值为一个最小值，即这个 Widget 的 X 方向长度不能小于 sizeHint 所设置的值。下面的示例设置 X 方向为最小值，而 Y 方向为固定值：

```
setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Fixed);
```

QSizePolicy 类实际定义了 4 个标志常量来决定 Widget 如何改变尺寸大小，如表 8.1 所示。

表 8.1: Widget 尺寸策略标志

常量	描述
QSizePolicy::GrowFlag	可以大于默认值
QSizePolicy::ExpandFlag	尽量获得最大值
QSizePolicy::ShrinkFlag	可以小于默认值
QSizePolicy::IgnoreFlag	忽略默认值设置

其中 Fixed 常量不包含任何一个标志，而 Minimum 实际等于 GrowFlag。

如果要在 Widget 中显示特殊图形，就要用到 QWidget 的 paintEvent 虚函数。这个函数在 Widget 需要被绘制的时候被调用。例如在 Widget 第一次被显示或者需要更新显示的时候，paintEvent 函数都会被调用。

paintEvent 函数可以用 QPainter 类来进行绘制。实际上，使用 QPainter 类直接在 Widget 上绘图只能在 paintEvent 和由 paintEvent 函数调用的函数中实现。下面的例子在 MyWidget 上绘制了一个半径为 3 的实心圆。

```
void Chequer::paintEvent(QPaintEvent *event)
{
        QPainter painter(this);
        painter.setBrush(Qt::SolidPattern);
        painter.drawEllipse(QPoint(10, 10), 3, 3);
}
```

QPainter 类提供了大量用于在 Widget 或其他可绘画类（例如 QPixmap 类）上进行绘画的函数，这些函数包括绘制简单的直线、曲线、饼图、对齐的文本或者位图。

8.2.4 使用 Qt 的图形部件

在前面 Hello World 示例中，我们用到了 QPushButton 这个 Widget。实际上 Qt 提供了非常丰富的 Widget 类库，表 8.2 仅仅是比较常用的 Widget 的简单列表。

表 8.2: Qt 部分 Widget 列表

Widget 类	描述
QPushButton	按钮部件
QCheckBox	复选框
QRadioButton	单选框
QGroupBox	带标题的分组框
QFrame	简单的修饰性 Widget 容器
QListView	类似于 Windows 资源浏览器的图标/文字列表
QTreeView	树形列表
QTableView	表格
QProgressBar	水平进度条
QLCDNumber	类似数码管显示风格的数字显示部件
QLable	显示文本或者图片的标签
QSlider	水平或者垂直的滑动条
QLineEdit	简单文本输入框
QSpinBox	带上下箭头的数字输入框
QComboBox	下拉列表输入框
QTextEdit	显示或修改复杂文本的部件
QScrollBar	水平或者垂直滚动条

如果需要了解更多的 Qt 类信息或者具体一个 Qt 类的使用方法，可以参考 Qt 的联机文档。在完整安装 Qt 的主机上执行 assistant 命令就可以打开如图 8.4 的帮助程序。assistant 程序不但提供了完整的 Qt 类说明还包括非常详细的有关 Qt 编程与 Qt 工具使用的帮助信息，甚至包含丰富的示例程序。在进行 Qt 程序开发的时候，你会发现 assistant 程序非常有帮助。

当在程序中使用的 Widget 数多于一个的时候，如何将它们在窗口布局将成为需要解决的问题。为此 Qt 提供了一些特殊的类对 Widget 的位置和大小进行管理，它们被称为布局管理器。使用布局管理器可以让 Qt 自动地对多个 Widget 的位置和大小进行设置，以充分利用窗口中的可用空间。

比较简单的布局管理办法是使用 QHBoxLayout、QVBoxLayout 和 QGridLayout 布局管理器。其中 QHBoxLayout 会让所有的子类 Widget 按照从左到右的横向顺序排列，而

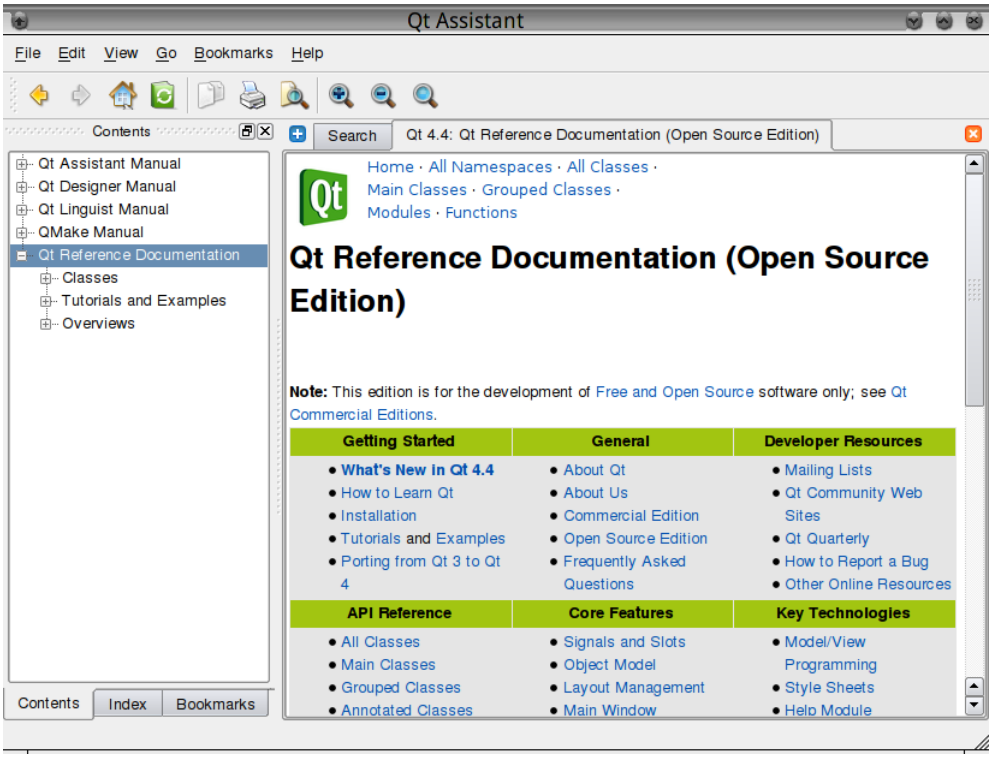

图 8.4: Qt Assistant 联机文档

QVBoxLayout 则使子类 Widget 按照从上到下的纵向排列。QGridLayout 更复杂一些，它将全部空间看成一个均匀分割的方阵，用户可以设定每个 Widget 所占用的方格数。

真正要使用布局管理器的时候，需要把 Widget 一个个通过 addWidget 函数加入到布局管理器中，然后再选定一个父类，一般可以简单地选择 QWidget，并通过 setLayout 函数与前面的布局管理器联系起来。

下面的例子将 5 个按钮添加到了一个 QGridLayout 中。addWidget 的调用格式有两种，其中 3 个参数的版本将按钮添加到了特定的坐标，并占用一个单元格；具有 5 个参数的函数版本不但设置了按钮的坐标，还分别指定了垂直方向占 1 行，水平方向占 4 列。最后的显示效果如图 8.5 所示。

```
QWidget *window = new QWidget;
QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 0, 2);
layout->addWidget(button4, 0, 3);
layout->addWidget(button5, 1, 0, 1, 4);
window->setLayout(layout);
window->show();
```

图 8.5: 使用 QGridLayout 进行布局

8.2.5 自定义 Widget

如果需要实现更加复杂的功能，Qt 提供的标准 Widget 往往不能符合要求，此时就要定义自己的 Widget。C++ 的继承特性使得自定义 Widget 非常简单，我们可以很容易地继承一个现有的 Widget，然后给它添加新的属性，如信号、槽、函数等。下面我们通过一个简单的示例对自定义 Widget 的过程进行演示。首先看一下源程序，其中包含一个头文件和一个 cpp 文件。

头文件：

```
#ifndef __MAIN_H__
#define __MAIN_H__
#include <QWidget>

class MyWidget : public QWidget
{
        Q_OBJECT
public:
        MyWidget(QWidget *parent = 0);
public slots:
        void myVersion(void);
};
#endif
```

主程序：

```
#include <QtGui>
#include "main.h"

MyWidget::MyWidget(QWidget *parent) : QWidget(parent)
{
        QPushButton *quit = new QPushButton("&Quit");
        QPushButton *about = new QPushButton("&About");
        QHBoxLayout *layout = new QHBoxLayout();
        layout->addWidget(about);
        layout->addWidget(quit);
        setLayout(layout);
        connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
        connect(about, SIGNAL(clicked()), this, SLOT(myVersion()));
}

void MyWidget::myVersion(void)
{
        QMessageBox::about(this, "Test", "Version 2");
}

int main(int argc, char *argv[])
{
        QApplication app(argc, argv);
        MyWidget widget;
        widget.setWindowTitle("Test");
        widget.show();
        return app.exec();
}
```

程序的显示界面如图 8.6 所示。

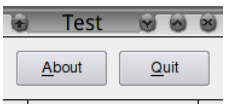

图 8.6: 自定义 Widget

头文件的定义比较简单，但由于使用了 Q_OBJECT 宏，因此必须把类的定义放在单独的头文件中，否则在编译的时候会出现 “undefined reference to ‘vtable for MyWidget’ ” 的错误提示。

MyWidget 类继承于 QWidget，并声明了一个新的槽：myVersion。在类的构造函数中，我们创建了两个按钮，并通过 layout 布局管理器将这两个按钮设置为自己的子类。

在设置按钮标题的时候，我们使用了 & 符号，这个符号用来标记标题中的快捷键，它并没有在图 8.6 中出现，而 & 符号后面的字母 “A” 下面增加了一个下划线。当这个窗口获得焦点的时候，只要同时按下 Alt 键与 A 键就可以触发 about 按钮的 clicked 信号。

在 MyWidget 的构造函数中还使用 qApp 变量，它实际上是 Qt 定义的一个类型为 QApplication 的全局指针变量，它代表了当前的应用程序实例。QApplication 预定义槽 quit 将终止当前程序的运行。

在自定义的槽 myVersion 中，我们使用 QMessageBox 的静态函数 about 显示了一个版本信息提示窗口，窗口的标题为 “Test”，内容为 “Version 2”。

在 main 函数中，我们使用 setWindowTitle 函数为主程序的窗口设置了标题，否则窗口的标题将是可执行程序的名字。

8.2.6 qmake 的更多用法

到目前为止，我们编译 Qt 程序都是先运行 qmake -project，然后再运行 qmake 来生成 Makefile 文件。但 qmake 在生成 pro 文件的时候并不总是非常聪明地了解程序的需要，有时候需要由用户对 pro 文件进行修改。

首先来分析一下 qmake 命令对后面将要给出的 “猜数字” 程序自动生成的 pro 文件：

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += main.h
FORMS += guess.ui
SOURCES += main.cpp
```

TEMPLATE 变量指定了生成 Makefile 的模板。这里是 app 模板，一般应用程序都应该选择 app。另外一个常用的模板是 lib，它用来生成库文件。

TARGET 变量指定了生成可执行文件的名字，默认情况下将使用 pro 文件的名字。DEPENDPATH 用来指定解决依赖关系的目录列表，INCLUDEPATH 用来指定头文件搜索目录列表。

HEADERS 变量指定了程序中所有头文件的名字。FORMS 变量指定了程序中所有 ui 文件的名字。SOURCES 变量指定了程序中所有 cpp 源文件的名字。这三个变量列举的文件名如果多于一个，可以使用空格进行分隔。

如果给当前工程添加了新的文件或者删除了旧的文件，必须重新执行 qmake -project 命令或者直接用文本编辑器修改 pro 文件。另外还有一些情况也必须手工修改 pro 文件，而在手工修改 pro 文件之后，再执行 qmake -project 命令就可能覆盖掉手工修改的部分。

Qt 程序的 pro 文件变更之后，不要忘记运行 qmake 命令来更新 Makefile 文件，然后再执行 make 命令获得最新的可执行程序。

需要手工修改 pro 文件的常见情况有如下几个：

1. 使用附加的 Qt 模块
   Qt 4.x 目前包含如表 8.3 所示几个模块。

表 8.3: Qt 模块

模块名称	描述
core	Qt 核心模块
gui	Qt 图形用户接口模块
network	Qt 网络模块
opengl	OpenGL 图形加速模块
phonon	Phonon 多媒体模块
sql	SQL 数据库模块
svg	Svg 矢量图形模块
xml	Xml 文档支持模块
webkit	WebKit 网络支持模块
qt3support	Qt3 兼容支持模块

其中 core 和 gui 模块是默认就加入工程的，如果需要修改默认值就要使用 QT 变量。例如需要加入网络模块，并且去掉图形接口模块，可以使用如下代码：

```
QT -= gui
QT += network
```

2. 需要编译可调试的代码
   如果需要用 gdb 调试 Qt 程序，必须在编译源代码的时候为 g++ 传递 -g 参数，同时还需要对编译参数进行与 Qt 相关内容的调整。这个工作也可以交给 qmake 来完成。此时需要在 pro 文件中为 CONFIG 参数增加 debug 选项：

```
CONFIG += debug
```

CONFIG 参数在工程中会起到很多作用，其中与编译参数有关的选项还有 debug_and_release 也会经常用到。debug_and_release 选项使得工程同时包含发行和调试编译模式，用户执行 make release 将进行发行模式的编译，执行 make debug 将进行调试模式的编译。

3. 在编译命令行增加宏变量的定义
   qmake 在生成 Makefile 的时候，会根据 pro 文件的 DEFINES 变量的值，在调用 g++ 的参数中增加 -D 参数来定义新的宏变量。源代码可以根据宏变量的定义情况实现不同的功能。例如在 pro 文件中增加：

```
DEFINES += MY_DEBUG
```

就相当于在调用 g++ 命令编译源程序的时候增加 -DMY_DEBUG 参数。

4. 增加 Qt 的多语言支持
   如果要动态调整 Qt 程序界面的显示语言，可以使用 Qt 的国际化支持。这时需要修改 pro 文件的 TRANSLATIONS 参数。关于这个功能在第 8.5.1 小节有详细的介绍。

5. 把数据文件包含到可执行文件
   有时候需要把程序的数据文件直接包含到可执行文件中。这样就不用担心程序需要的数据文件可能在文件传播的过程中会丢失。关于这个功能在第 8.5.2 小节有具体的介绍。

8.3 Qt 编程的可视化辅助设计工具

从前面几个 Qt 图形界面程序例子可以看出，对程序界面的控制完全可以通过程序代码完成，但是随着界面的复杂，设计工作也越来越困难，很难方便直观地看到运行后的样子，因此 Qt 提供了一个可视化的界面设计工具 Qt Designer，可以交互式地采用 “所见即所得” 方式设计图形界面。

8.3.1 Qt Designer 简介

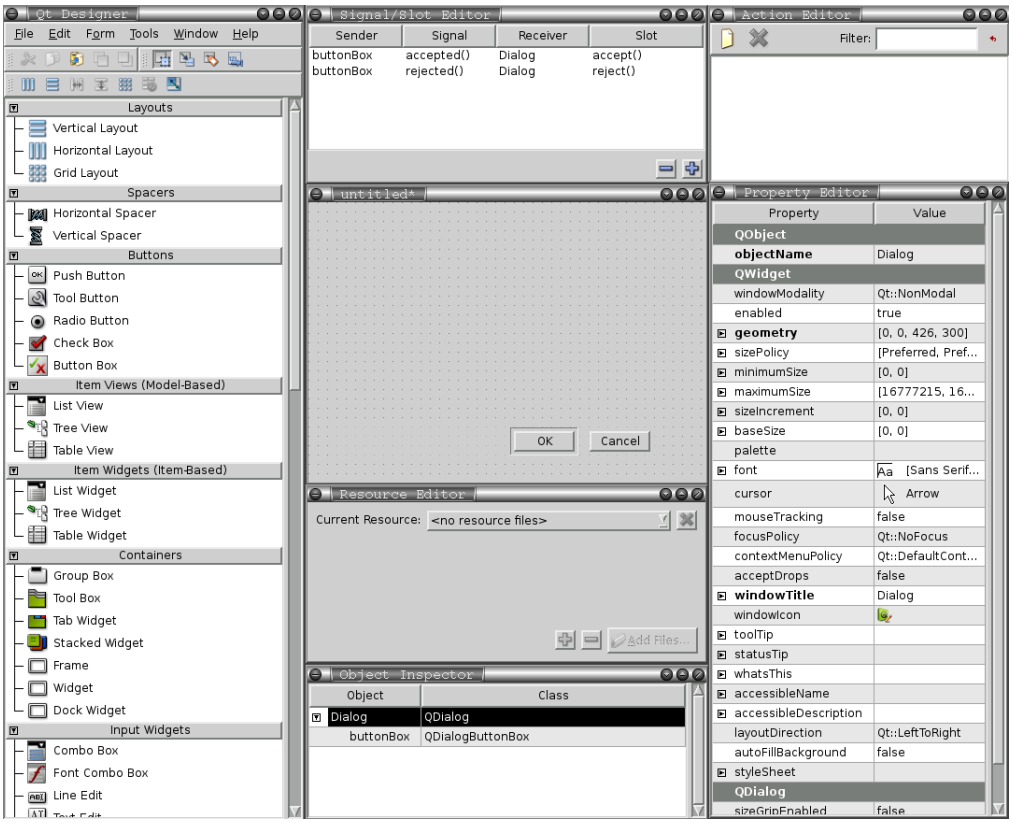

图 8.7: Qt Designer 界面

Qt Designer 是一种跨平台的可视化用户界面设计工具，它使得 GUI 程序设计中最繁重的程序界面外观设计变得简单，维护也更加方便。Qt Designer 本身不是一个完整的集成开发环境（新版 Qt 包含一个名为 Qt Creator 的集成环境），它只用来对界面元素进行修改，并不具备代码的编辑功能。

在 Qt 4.x 版本中，Qt Designer 是一个简单的可视化视窗设计器，其界面效果如图 8.7 所示。它把界面设计的结果保存为扩展名为 ui 的文件格式。ui 文件包含单个对话窗口的 XML 描述，配套的用户界面编译器 uic（user interface complier）可以根据.ui 文件中的 XML 描述生成对应的 C++ 代码。

Qt Designer 的使用方法非常简单，新建窗口的时候可以从模板中选择 Widget、Dialog、Main Window 三个选项。其中 Widget 表示窗口类继承于 QWidget，是一种最灵活的方案，用户可以选择把这个设计作为一个窗口或者是窗口的一部分。Dialog 表示新建的窗口类继承于 QDialog 类，它适合作为一般的对话框窗口。Main Window 表示新建的窗口类继承于 QMainWindow 类，QMainWindow 是 Qt 对复杂应用程序框架的支持类，可以方便地添加菜单、工具条、状态栏和子窗口等。

在具体设计窗口的时候，用户只要把所需要的界面元素从主窗口的列表拖动到要设计的窗口上就可以了。除了可以把常用的按钮、标签之类的视觉元素放置到窗口上，还可以把布局管理

器(layout)放置到窗口,对其内部的 Widget 进行管理。另外还有一种特殊的部件叫做 Spacer,它们用来在布局管理器占位,形成一个空白的区域。

每个放置在窗口上的 Widget 的属性可以通过 Property Editor 窗口进行编辑。例如可以设置 Widget 的名字,设置其显示的字符串等等。最后把设计的结果保存为 ui 文件。

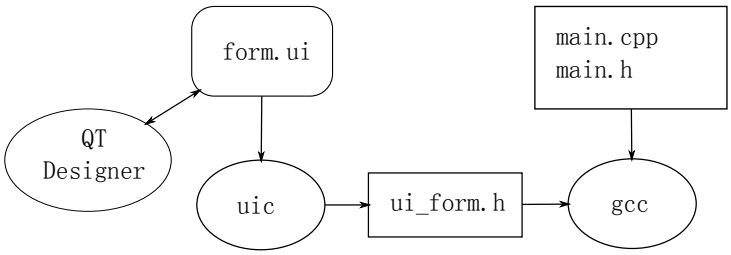

图 8.8: .ui 文件,生成的代码和应用程序代码的关系

图 8.8 描述了 ui 文件与其他工程代码的关系。用户界面编译器 uic 读取 ui 文件,生成 C++ 头文件 ui_form.h。main.cpp 和 main.h 中的应用程序代码包含了 ui_form.h。最后通过 gcc 编译链接生成可执行程序。

8.3.2　ui 文件的使用方法

在使用 Qt Designer 的过程中,窗口界面的设计和代码逻辑的设计必须要分开,这种设计模式非常值得提倡,因为这减少了界面元素与程序逻辑的耦合,可以独立地修改程序的界面而不对程序逻辑产生影响。因此程序员不应该修改 uic 生成的头文件(图 8.8 中的 ui_form.h),而必须采用其他的方式来把界面和程序逻辑联系起来。

例如通过文本编辑工具打开 Qt Designer 设计的 form.ui 文件可能看到如下内容:

```
<ui version="4.0" >
 <class>Form</class>
 <widget class="QWidget" name="Form" >
  <property name="geometry" >
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle" >
   <string>Form</string>
  </property>
  <widget class="QPushButton" name="quit" >
   <property name="geometry" >
...
```

这里只列出了 ui 文件的前面几行。从这个文件可以看出 ui 文件包含了对窗口内容的描述,例如窗口的名字、尺寸,其中包含的 Widget 等。使用 uic 命令生成 ui_form.h 的命令如下:

```
$ uic form.ui > ui_form.h
```

在实际编程中,用户不必键入上面的命令,因为 qmake 所生成的 Makefile 文件会自动调用 uic 命令,把所有 ui 文件转换成 ui_<ui 文件名 >.h 文件。在编写代码的时候可以直接包含 ui_xxxx.h 文件,而不管这个文件是否存在。为了了解 uic 命令都做了什么,我们可以看一下 ui_form.h 文件包含什么内容:

```
class Ui_Form
{
public:
    QPushButton *quit;

    void setupUi(QWidget *Form)
    {
/* 省略部分代码 */
    quit = new QPushButton(Form);
    quit->setObjectName(QString::fromUtf8("quit"));
    quit->setGeometry(QRect(50, 50, 78, 29));
/* 省略部分代码 */
}

namespace Ui {
    class Form: public Ui_Form {};
} // namespace Ui
```

可见 Qt4 定义了一个新的名字空间 Ui，把界面相关的代码封装到了这个名字空间的一个类中。在上面代码中，界面类的名字是 Form，它继承于 Ui_Form 类，其中定义了一个 setupUi 函数，注意这个函数的输入参数是 QWidget，它必须与使用 Qt Designer 建立的窗口类相同。setupUi 函数执行具体的界面初始化代码，例如上面代码中显示的建立按钮，并设置按钮属性的部分。

setupUi 函数需要在主程序中调用，主程序在使用 Form 类的时候一般有三种方法：

1. 直接使用，这种方法最为简单，但受到的限制也比较多。代码示例如下：

```
int main(int argc, char *argv[])
{
        QApplication app(argc, argv);
        QWidget *window = new QWidget; // 需要传递给 setupUi 的参数
        Ui::Form ui;
        ui.setupUi(window);
        window->show();
        return app.exec();
}
```

这种方法不需要编写额外的文件，非常直接，但无法把更多的界面功能添加到程序中，实际中很少采用。

2. 直接继承，这种方法首先要建立一个单独的 Qt 类，类似于前一种方法把 Ui 元素定义为类的私有元素。这样就可以在类中访问界面元素，并可以自由地定义信号和槽与图形界面交互。类的定义要写到单独的头文件中，由于需要把类的实例传递给 setupUi 函数，因此这个类必须继承于 QWidget。代码示例如下：

```
class MyWidget : public QWidget
{
        Q_OBJECT
        public:
        MyWidget(QWidget *parent = 0);

        private:
        Ui::Form ui;
};
```

setupUi 函数可以在类的构造函数中来调用，信号和槽的连接也可以在类的函数中进行。

3. 多重继承，C++ 的多重继承是解决这个问题最好的办法，新建的 MyWidget 类需要同时继承 QWdiget 和 Form。多重继承方法使得图形元素和代码的结合更加紧密和灵活，代码示例如下：

```
class MyWidget : public QWidget, private Ui::Form
{
        Q_OBJECT
        public:
        MyWidget(QWidget *parent = 0);
};
```

采用这种方法构造的 MyWidget 类把所有的图形元素作为自己的私有成员，可以像普通的类成员一样访问，调用 connect 连接信号与槽的时候也更加自然，程序代码显得更加清晰和紧凑。MyWidget 类的构造函数示例如下：

```
MyWidget::MyWidget(QWidget *parent)
{
        setupUi(this);
}
```

在使用 Designer 的时候，另外一个有用的特性是信号和槽的自动连接。只要在定义槽的名称时遵守一定的规则，就可以把它和特定的信号自动连接起来。这个规则的格式如下：

```
void on_<widget name>_<signal name>(<signal parameters>);
```

例如定义了名为 on_pushButton_clicked 的函数，它就会自动和 pushButton 的 clicked 信号连接起来。

下面我们以一个例子回顾 Qt 程序的设计流程，这个程序要实现一个猜数字的小游戏：要猜的数字是一个 4 位的十进制数，4 个数字各不相同，而且最高位可以是 0。游戏者每次猜的数也要符合前面的规定，即可以猜 “1234”，但不能猜 “2234”。对于每次猜的数字，程序都要给出有几个位置猜对了（用 “A” 表示），有几个数字猜对了，但位置不对（用 “B” 表示）。例如答案是 9203，当猜 1234 的时候，计算机要给出信息 “1A1B”，而当猜 6789 的时候要给出 “0A1B”。

首先用 Designer 新建一个窗体，从模板中选择 Widget，这样我们创建的 ui 文件在生成头文件时，窗口的基类就会是 QWidget。然后设计程序的界面如图 8.9 所示。

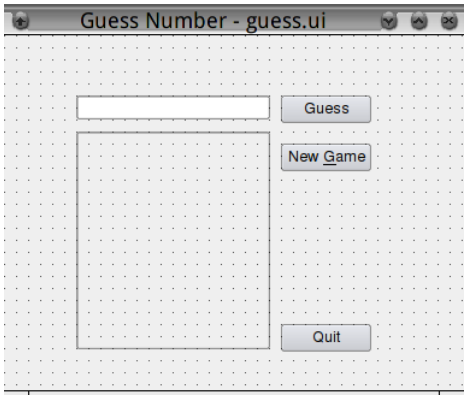

图 8.9: 猜数字的用户界面

修改窗口的名字（objectName）为 GuessNum，修改 “New Game” 按钮的名字为 “newgame”，修改 “Guess” 按钮的名字为 “guess”，输入框为 QLineEdit，名字为 “num”，输出框为 QLabel，名字为 “history”，用来显示每次猜的数字与结果。最后保存结果为 guess.ui 文件。

编写 main.h 文件如下：

```
#ifndef __MY_MAIN__
#define __MY_MAIN__

class QWidget;
#include "ui_guess.h"   // 将由 uic 生成

class GuessWidget : public QWidget, public Ui::GuessNum  //多重继承
{
        Q_OBJECT
public:
    GuessWidget(QWidget *parent = 0);

private:
        int pos[4];  // 记录答案
        void newGuess(void);

public slots:
        void on_guess_clicked();
        void on_newgame_clicked();

};
#endif  //__MY_MAIN__
```

上面头文件中定义的两个槽就利用了自动连接功能。main.cpp 文件这里就不给出了，最终程序的完成留作读者的练习。

8.4 完全面向 Qt 编程

Qt 并不仅仅提供了图形界面程序接口，它还提供了从底层的输入/输出操作到高层的应用程序框架的完整支持库。用户可以只使用 Qt 库实现各种类型的程序，而且编写出的代码还是跨平台的，可以在不同的操作系统环境下编译运行。

8.4.1 简单数据类型

C++ 语言的基本数据类型并不都有很好的跨平台支持，因此，Qt 对一些常用的数据类型进行了重新定义，这样可以获得更好的移植性，而且使用起来也很方便。例如 qint8 代表 signed char，这个类型可以保证在所有平台都是 8 位的有符号数。类似的还有 qint16，qint32，qint64 等，它们分别对应 16 位、32 位、64 位的有符号数。对于无符号数，Qt 还定义了 quint8，quint16，quint32，quint64 这四个数据类型。

字符数据在 Qt 中用 QChar 类型来表示。在 Qt 中所有以大写 Q 开始的类型名都是一个类。QChar 用来代表一个 16 位的 Unicode 字符，它并没有其他内在标记结构，因此对于大多数编译器来说依然是“轻量级”的，可以看作 unsigned short 类型。采用 Unicode 的编码使得处理跨平台和国际化的问题变得简单，程序员不必了解字符编码的具体细节。

对字符进行处理的最常见任务就是了解字符的类别，QChar 提供了一系列函数来完成这个任务。例如 isPrint 函数用来判断字符是否为可打印字符（包括空格），isLetter 函数用来判断字符是否是一个字母，isDigit 用来判断字符是否是一个数字。

如果需要对字符进行转换，Qt 也提供了常见的 toUpper 用来把字符转换为大写，toLower 用来把字符转换为小写。如果需要把数字类的字符转换为整数类型，还可以使用 digitValue 函数。在 Qt 编程中一般不需要使用 8 位的字符类型，但在与其他程序或数据交互的时候可能需要用 toAscii 和 fromAscii 函数完成数据类型的转换。

QString 类型在前面的例子中已经出现过，它实际保存的内容是一个 QChar 的数组，因此可以用来表示 Unicode 的字符串。标准 C 语言中的字符串是一个 char * 类型，Qt 重载了很多操作符使得在 Qt 中操作字符串非常简单。例如可以使用 char * 类型的数据来初始化 QString；可以直接用 == 符号来判断字符串是否相等；也可以直接使用 += 符号在字符串末尾添加新的内容。如果需要把 QString 类型的字符串转换为 char * 类型，可以使用 Qt 提供的宏 qPrintable。

QString 类型在 Qt 中的应用非常广泛，Qt 也为 QString 类型的处理提供了非常多的函数。例如用于在字符串中进行查找替换的函数 replace 就有 7 个重载的版本，可以实现定位的替换，也可以实现特定子串的查找替换，还可以实现根据正则表达式进行的替换。例如：

```
QString x = "Hello world";
x.replace(0, 5, "Good"); // x == "Good world"
x.replace('w', 'W');     // x == "Good World"
```

QString 中对字符的定位是从 0 开始计数的，程序中可以使用 [] 符号选择指定的下标，对于只读访问也可以通过 at 函数来访问特定位置的字符。例如：

```
QString str;
str.resize(1); // 设置字符串的长度
str[0] = 'H';
if (str[0] == 'k') return;
```

如果要把 QString 类型的字符串转换为数字，可以使用 toInt 函数，这个函数的定义如下：

```
int QString::toInt ( bool * ok = 0, int base = 10 ) const;
```

其中 base 为数字的进制，转换成功 ok 的值为 true，函数返回转换的结果。例如：

```
QString num = "DEED";
bool ok;
qint32 val = num.toInt(&ok, 16); // val == 57069
```

如果把一个数字转换为字符串可以用 setNum 函数，或者使用类的静态函数 number。例如：

```
int k = 1234;
QString str = QString::number(k); // str == "1234"
str.setNum(57069, 16);    // str == "deed"
```

在使用 Qt 库编程的时候，推荐在可能的情况下都要使用 QString 类型来操作字符串。除了前面提到的函数，Qt 还提供了非常全面的有关字符串查找、比较、格式转换等方面的函数，使在 Qt 中操作字符串的任务变得很方便。

与 QString 类似的另外一个非常常用的类是 QByteArray，它用来保存一个 byte 类型的数组。QByteArray 所保存的数据并没有编码格式，非常适合用来处理二进制的（非文本）数据包。除了数据保存的内部格式，QByteArray 与 QString 非常类似，前面提到的处理 QString 的函数几乎都有对应的 QByteArray 版本。

8.4.2 文件输入/输出

Qt 使用 QFile 作为文件输入/输出的接口类，提供一种跨平台的文件操作的方法。传统上，文件一般分为文本文件和二进制文件。其中文本文件包含可读文本，可以按行进行读取；二进制文件一般具有特定的数据格式，用文本编辑器打开看到的结果将是乱码。

对于文本文件，经常使用 QTextStream 类来简化文本信息的操作。QTextStream 的使用方法有些类似标准 C++ 中的输入/输出流，下面的例子是一个写入日志文件的简单实现：

```
QFile data("log.txt");
if (data.open(QFile::WriteOnly | QFile::Append)) {
     QTextStream out(&data);
     out << QDateTime::currentDateTime().toString() << ':' << msg << endl;
}
```

程序首先定义了一个 QFile 的实例，并通过构造函数传递了文件的名字。下一行程序使用 open 函数将文件打开，其参数代表打开文件的方法，常见的标志如表 8.4 所示，它们可以在调用时组合使用。

表 8.4: QFile 打开文件的标志

标志	描述
QIODevice::NotOpen	不打开设备
QIODevice::ReadOnly	将设备只读打开
QIODevice::WriteOnly	将设备只写打开
QIODevice::ReadWrite	将设备以读写方法打开
QIODevice::Append	打开文件后将文件指针移到最后
QIODevice::Truncate	打开时文件内容被截断，即清空
QIODevice::Text	打开的文件是文本文件
QIODevice::Unbuffered	不使用内部缓冲

程序第三行通过 QFile 类的实例建立了 QTextStream 类实例，QTextStream 类除了可以在 QFile 之上构造（基于 QIODevice），还可以基于 QByteArray 和 QString，与 QString 类似，QTextStream 也是基于 Unicode 编码来保存文本数据。

程序第四行通过 “<<” 符号将文本信息输出到了文件中，其中 QDateTime 类在 Qt 中用来处理时间与日期，这里调用了 currentDateTime 函数获得当前的时间。

QDataStream 的用法与 QTextStream 类似，只是操作的数据是二进制文本。Qt 支持把各种类型的数据保存到 QDataStream 并在之后将它们读取出来，而且这个过程同样与平台无关。

QFile 类还定义了一些静态函数对磁盘文件进行简单的操作。例如文件复制、删除、改名、判断文件是否存在等：

```
bool copy ( const QString & fileName, const QString & newName )
bool remove ( const QString & fileName )
bool rename ( const QString & oldName, const QString & newName )
bool exists ( const QString & fileName )
```

如果需要对文件系统的信息有更多的了解和获得更多操作，可以使用 QFileInfo 和 QDir 类，有关这两个类的资料可以参考 assistant 文档。

8.4.3 网络编程

我们曾经介绍过使用 socket 来进行网络编程的方法，但在使用 Qt 编程的时候，如果同时需要对 TCP/IP 网络的支持，最好使用 Qt 所提供的网络编程接口。这样做的好处主要有以下几点：

- Qt 的网络接口采用 C++ 语言封装，使用起来更加方便。

- Qt 的网络接口采用事件驱动来实现，进一步提高了易用性。
- Qt 的网络接口具有更好的跨平台性，前面介绍的 socket 接口不能用于 Windows 平台，而 Qt 的网络接口是可以的。

对于面向连接的 TCP 编程，Qt 提供了 QTcpServer 和 QTcpSocket 两个类，其中前者用来实现服务器，后者用来实现客户端。使用 QTcpServer 建立 TCP 监听的过程非常简单，只需要调用 listen 函数就可以实现在特定的端口上等待客户端的连接，而每个新的连接都会触发 newConnection 这个事件。典型的建立监听的代码如下：

```
server = new QTcpServer();
connect(server, SIGNAL(newConnection()), this, SLOT(my_new_conn()));
server->listen(QHostAddress::Any, MY_PORT);
```

当新连接建立的时候，可以调用 nextPendingConnection 获得一个用于通信的 QTcpSocket 类。如果只需要等待新数据的到来，就可以对 QTcpSocket 的 readyRead 信号进行处理，在接受这个信号的槽处理函数中调用 read 函数读取收到的数据。下面的示例代码建立了信号和槽的连接，并关闭了监听的 socket。

```
client = server->nextPendingConnection();
connect(client, SIGNAL(readyRead()), this, SLOT(my_new_data()));
server->close();
```

客户端的程序也很容易实现，下面是一个示例代码：

```
client = new QTcpSocket(this);
connect(client, SIGNAL(readyRead()), this, SLOT(my_new_data()));
connect(client, SIGNAL(connected()), this, SLOT(my_new_conn()));
client->connectToHost(ip->text(), MY_PORT);
```

客户端程序通过 connectToHost 函数连接到服务器，其中第一个参数是一个 QString 类型的 IP 地址，第二个参数是一个整型的端口地址。当连接建立以后就会发射一个 connected 信号，而如果有新数据到来，同样会触发 readyRead 信号。

如果要编写面向非连接的 UDP 网络通信程序，就要求助于 Qt 的 QUdpSocket 类。使用 QUdpSocket 作为服务器方式接收客户端的数据必须首先将其绑定到特定的 UDP 端口。示例代码如下：

```
udp = new QUdpSocket(this);
udp->bind(QHostAddress::LocalHost, MY_PORT);
connect(udp, SIGNAL(readyRead()), this, SLOT(my_new_data()));
```

其中 QHostAddress::LocalHost 表示本机的 IPv4 地址。

当从 UDP 端口获得数据之后，hasPendingDatagrams 函数将返回真值，使用 pendingDatagramSize 函数可以获得接收到数据包的大小，然后可以使用 readDatagram 函数获得具体的数据包内容。下面是一个处理接收数据的示例：

```
while (udp->hasPendingDatagrams()) {
        QByteArray datagram;  // 接收数据缓冲区
        int len;
        len = udp->pendingDatagramSize(); // 接收数据个数
        datagram.resize(len);  // 分配空间
        udp->readDatagram(datagram.data(), len); // 接收数据
```

```
....
}
```

如果仅需要发送 UDP 数据包，则可以省略 bind 函数，直接调用 writeDatagram 函数发送数据。writeDatagram 函数有两种定义：

```
qint64 writeDatagram (const char * data, qint64 size, const QHostAddress & host, quint16 port)
qint64 writeDatagram (const QByteArray & datagram, const QHostAddress & host, quint16 por
```

其中 QHostAddress 类型的地址可以使用这个类的 setAddress 函数来设置，例如：

```
host->setAddress(ip->text());
```

除了前面提到的 QTcpSocket、QTcpServer 和 QUdpSocket 类，Qt 还提供了实现更高层次协议的类。例如实现了 Ftp 协议的 QFtp 类，和实现了 Http 协议的 QHttp 类。使用这两个类可以使得完成相应的协议相关任务变得简单。

在本小节的最后还要提醒一下，如果使用了以上网络相关的 Qt 类，必须要在工程的 pro 文件中增加 network 模块的声明，否则编译程序的时候就会找不到相关的类定义。

```
Qt += network
```

8.5 发布 Qt 程序

8.5.1 Qt 程序的国际化

Qt 对国际化的支持是很好的，它内置支持现在世界上使用的大多数语言，这里我们以中文为例，介绍在 Qt 中进行国际化的主要方法。

Qt 的基本字符串库 QString 使用 Unicode 4.0 作为编码，所以只要在程序中尽量使用 QString，就不会遇到字符编码的问题，特别是对于显示给用户的字符串，在保存的时候最好使用 QString 类型。

如果要在一个程序中支持多种语言，就需要使用 Qt 的 tr 函数对程序中的字符串进行 “封装”。例如下面对按钮显示内容的初始化就使用了 tr 函数：

```
    QPushButton hello( QObject::tr("Password:"), 0 );
```

在程序中，所有用双引号引用的字符串，如果需要显示给用户，就一定要用 tr 函数 “处理” 一下。只有这样，Qt 才可以利用它自己的翻译功能，在程序运行的时候用特定的语言来显示字符串。具体的翻译方法，后面还会具体介绍。这个 tr 函数是 QObject 类的成员函数，所以如果不是在 QObject 的衍生类中使用，则可以使用 QApplication 的 translate 函数代替。

如果要处理的字符串甚至不在一个函数中，还可以使用 QT_TR_NOOP 宏来对字符串进行标记，下面是这样的一个例子：

```
QString FriendlyConversation::greeting(int type)
{
        static const char *greeting_strings[] = {
                QT_TR_NOOP("Hello"),
                QT_TR_NOOP("Goodbye")
```

```
        };
        return tr(greeting_strings[type]);
}
```

如果忘记了用 tr “标记” 字符串常量，会给后期的翻译带来麻烦，我们可以利用编译器来帮助检查，只要禁止编译器自动把 char * 类型转换为 QString 就可以了。这样，如果一个字符串没有被标记，就会产生一个编译错误。最简单的实现方法就是在工程的 pro 文件中增加如下一行：

```
DEFINES += QT_NO_CAST_ASCII
```

标记完程序中的所有字符串之后，就可以利用 Qt 的工具对程序进行处理了，这个过程分为 4 个步骤。

1. 在 pro 文件中添加要翻译的语言，Qt 把翻译的数据保存在扩展名为 ts 的文件中，每种语言一个文件。例如我们的程序叫做 hello，我们就可以把中文的翻译文件取名 hello_zh.ts，在 pro 文件中添加如下内容：

```
TRANSLATIONS = hello_zh.ts
```

如果要准备多个语言的翻译，就可以把多个语言文件都写到 TRANSLATIONS 后面，例如：

```
TRANSLATIONS = hello_zh.ts hello_fr.ts
```

2. 运行 lupdate 工具，根据源程序中的标记情况生成 ts 文件。这个命令以 pro 文件作为参数，例如：

```
$ lupdate hello.pro
```

3. 运行 linguist 工具修改 ts 文件。linguist 是图形界面的程序，用它打开前面生成的 ts 文件，为每个字符串设定翻译的内容。图 8.10 是 linguist 的截图，在 “Done” 列显示 “√” 的为设置完毕的翻译，否则显示 “？” 的是还没有设置的。在 Translation 部分添好翻译内容之后可以单击 “？” 号，将其设置为 “√” 号。全部设置完毕后保存退出。

4. 运行 lrelease 工具，将 ts 文件编译成二进制格式（扩展名为 qm），为程序运行使用。这个命令也使用 pro 文件作为命令参数。

生成 qm 文件之后，还要在程序中调用相应的翻译文件，例如：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTranslator appTranslator;
    appTranslator.load(QString("hello_") + QTextCodec::locale(),
                       qApp->applicationDirPath());
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}
```

程序在发行的时候必须包含 qm 文件，否则翻译模块不能加载。

8.5.2 将数据嵌入程序

Qt 提供了一种与平台无关的将二进制数据嵌入到可执行程序的方法。例如可以把程序的界面翻译文件（qm 文件）嵌入，这样就不用担心在运行程序的时候找不到对应的文件。

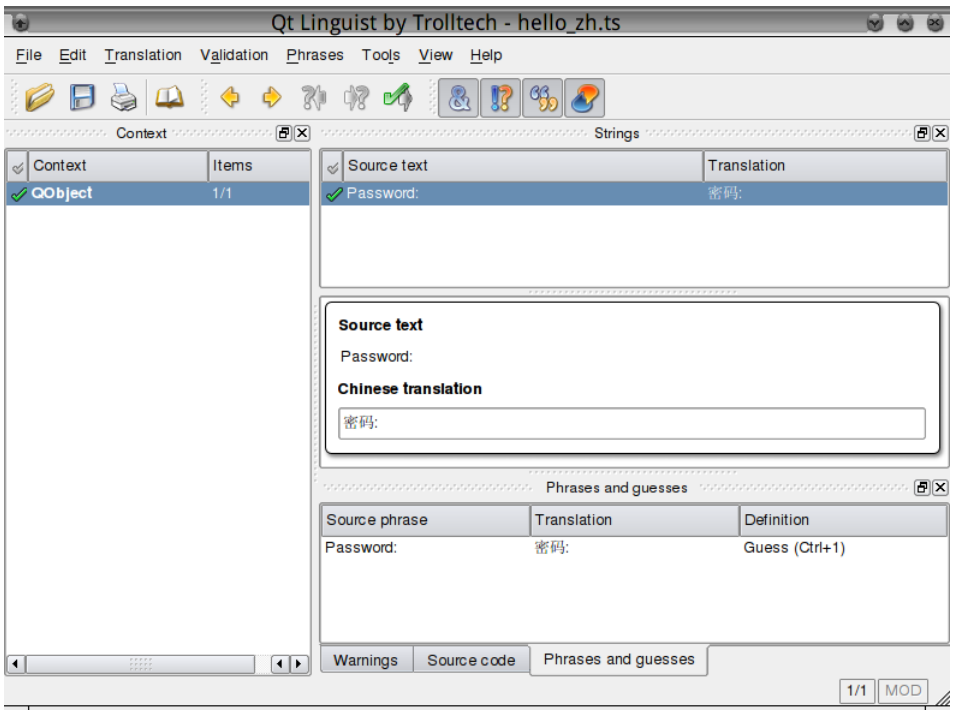

图 8.10: Linguist 界面

Qt 采用扩展名为 qrc 的文件记录二进制文件的信息，这个文件被称为资源文件。例如我们编写一个支持多语言的 Hello World 程序，并把 qm 文件和程序的图标嵌入到可执行程序中。此时我们可以编写如下的资源文件：

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/icon.png</file>
    <file>hello_zh.qm</file>
</qresource>
</RCC>
```

资源文件也是一个 xml 文档，其中 <file> 标签中的内容是需要嵌入可执行程序的文件名。文件名中可以包含路径，在编译工程文件前要保证相应的资源文件确实存在。

在程序中访问资源文件的方法与普通文件是类似的，Qt 的文件输入/输出类 QFile 完全支持资源的读取，只是在使用文件名字的时候需要加上 “:/” 前缀。例如在程序中要访问 images/icon.png 文件就要指定:/images/icon.png。下面的代码可以从已经嵌入到可执行程序的数据中读取应用程序的图标：

```
    qApp->setWindowIcon(QIcon(":/images/icon.png"));
```

Qt 的 qmake 工具可以很好地处理资源文件，用户不用担心其实现细节，只要在 pro 文件中增加对资源文件的声明就可以了。这里使用 RESOURCES 参数，并把资源文件的名字作为其选项。例如：

```
RESOURCES = hello.qrc
```

在执行 make 命令的时候，资源文件 hello.qrc 会被 rcc 命令处理，生成标准的 C++ 代码。这个代码包含了由数据文件生成的常量数组和相关的支撑函数，并最终被链接到可执行文件中。

8.6 Qt 的嵌入式应用

8.6.1 交叉编译 Qt

为了在嵌入式平台使用 Qt，就需要交叉编译 Qt Embedded。我们使用的源码是 qt-everywhere-opensource-src-4.6.3.tar.gz，由于有 configure 脚本的帮助，交叉编译并不是很困难，本书使用的配置命令如下：

```
$ ./configure -xplatform qws/linux-arm-g++ -embedded arm -no-svg \
> -no-qt3support -no-scripttools -no-webkit -no-nis -no-cups -no-qdbus \
> -depths 15,16 -qt-gfx-transformed -confirm-license -opensource -fast \
> -little-endian -prefix /usr/local/qte/ -qt-mouse-tslib
```

其中 xplatform 选项指定了交叉编译的类型，embedded 选项指定了嵌入式处理器为 arm。depths 选项指定了 Qt Embedded 所支持的图像像素位数。“-no-qt3support -no-nis -no-cups -no-qdbus” 等功能是可裁减的部分，可以根据需要进行设置。“-qt-gfx-transformed -qt-mouse-tslib” 是 Qt Embedded 的两个驱动选项，前者用来支持 Qt Embedded 的屏幕旋转功能，后者是 Qt Embedded 的触摸屏 tslib 支持。prefix 选项指定了 Qt Embedded 的安装位置，本例中是/usr/local/qte 目录。

配置成功后，执行 make 命令，就可以完成编译了。最后执行 make install 命令，编译好的二进制文件被安装到/usr/local/qte/ 目录中。这个目录中的库文件在交叉编译时还要使用。如果要在嵌入式板上使用 Qt Embedded 程序，还要把其中的 lib 与 plugins 目录复制到嵌入式板的文件系统中。

前面提到的 tslib 是一个底层的触摸屏库，可以提供给应用程序以统一、方便的触摸屏接口，并可以对触摸屏数据进行滤波、校准等操作，在嵌入式领域应用比较广泛。由于在编译 Qt Embedded 的时候使用了 -qt-mouse-tslib 选项，因此必须提前对 tslib 进行交叉编译。其编译过程也是先配置，再编译，最后安装：

```
$ ./configure --prefix=/usr/local/x-tools/arm-none-linux-gnueabi/arm-none-linux-gnueabi/ \
> --host=arm-linux
$ make
$ make install
```

在配置 tslib 的时候 `--host` 参数表示要进行交叉编译，`--prefix` 指定了安装的目录，这个目录在工具链中，安装后再编译其他依赖于 tslib 的程序的时候就不会找不到头文件或者库文件。tslib 所安装的 libts 动态库和 ts 目录中的插件库在嵌入式系统运行时会被用到，因此要拷贝到目标板的文件系统中。“/etc/ts.conf” 是它的配置文件，bin 目录下还有一些触摸屏的测试程序和校准程序。

交叉编译 Qt 应用程序的方法与非交叉编译类似，不过要使用交叉编译过的 qmake 命令，还要在生成 Makefile 的时候使用 -spec 参数指定 qmake 配置文件。例如：

```
$ /usr/local/qte/bin/qmake -spec /usr/local/qte/mkspecs/qws/linux-arm-g++
```

8.6.2 配置嵌入式 Qt 环境

在嵌入式板中运行 Qt 程序之前，还必须对 tslib 进行配置，tslib 编译之后的动态链接库是 libts-0.0.so.0.1.1，可以把它放到嵌入式文件系统的/lib/tslib 目录中（包括它的几个符号链接），它还有一些插件库，可以放在/lib/tslib/plugins 目录。tslib 的配置文件是 ts.conf，用来设置 tslib 模块的参数和应用顺序。一个简单的设置如下：

```
module_raw input
module variance delta=30
module dejitter delta=100
module linear
```

这个配置设定原始的触摸屏数据的处理顺序如下:

```
raw read --> variance --> dejitter --> linear --> application
```

具体的模块和参数的说明参考 tslib 的相关文档。

在嵌入式板运行 Qt 程序的一个完整的环境变量设置如下所示,相应的值要根据实际情况进行修改。

```
export TSLIB_CONFFILE=/etc/ts.conf
export QT_TSLIBDIR=/lib/tslib
export TSLIB_PLUGINDIR=/lib/tslib/plugins
export TSLIB_TSDEVICE=/dev/input/event0
export QWS_SIZE=640x480
export QWS_MOUSE_PROTO="tslib:/dev/input/event0"
export LD_LIBRARY_PATH=/opt/Qtopia/lib:/lib/tslib/
export QWS_DISPLAY=Transformed:Rot90:0
```

其中 TSLIB_CONFFILE 环境变量指定了 tslib 的配置文件;QT_TSLIBDIR 环境变量指定了 tslib 的库文件配置;TSLIB_PLUGINDIR 指定了 tslib 插件库的位置;TSLIB_TSDEVICE 指定了 tslib 的设备文件;QWS_SIZE 指定了嵌入式设备的显示器分辨率;QWS_MOUSE_PROTO 指定了 Qt 的输入设备协议,示例中包含了 tslib 关键字和对应的设备文件;LD_LIBRARY_PATH 指定了运行 Qt 程序所需要的动态库的位置;QWS_DISPLAY 环境变量指定了 Qt 在显示图像的时候要旋转 90 度。

8.7 实验 :嵌入式图形程序设计

实验目的

1. 掌握图形界面编程的基本方法
2. 熟悉 Qt 类库的使用

实验内容

1. 安装 Qt 开发环境

由于 Qt 的跨平台支持很好,在主机上运行的程序可以不必修改就能在嵌入式平台上编译运行,因此嵌入式 Qt 编程往往都是先在主机上开发调试好。如果要在 Ubuntu 主机上安装 Qt 开发环境,需要安装下面 4 个软件包:

```
# apt-get install libqt4-dev qt4-dev-tools qt4-designer qt4-doc
```

为了把调试好的 Qt 代码进行交叉编译,还必须安装交叉编译的 Qt 库。这个步骤可以根据第 8.6.1 小节编译 Qt 源代码,并执行 make install 完成;也可以将下载文件中的 qte-bin.tar.bz2 解压到/usr/local 目录中得到相同结构。这个压缩包是交叉编译 Qt 生成文件的压缩包,必须把它安装到指定的目录才可以正常工作。

2. 测试 8.2 节中提供的两个完整代码

第 8.2 节中 Hello World 程序和自定义 Widget 的源代码是完整的。首先在开发主机上将这两个测试程序完成，然后采用交叉编译的方法生成这两个程序的 ARM 版本代码，并在目标平台上运行测试。

在目标平台上运行 Qt 程序之前不要忘记对相关的环境变量进行设置，下载文件中所提供的文件系统映像中已经包含了完整的 Qt 库，相应环境变量的设置也写入了脚本，可以直接运行下面的命令进行设置。

```
# . /opt/Qtopia/env.sh
```

env.sh 的内容包含了全部需要设置的环境变量，它被保存在/opt/Qtopia 目录中。注意这个命令在脚本文件名之前有一个小数点和一个空格，这里的小数点可不是代表当前路径，而是等同于 source 命令，表示在当前的进程中执行这个脚本，这样在脚本中设置的环境变量才有效果。

在嵌入式平台运行 Qt 程序的时候还必须加入 -qws 参数，否则会出现如下错误信息：

```
$ ./hello
QWSSocket::connectToLocalFile could not connect:: No such file or directory
No Qt for Embedded Linux server appears to be running.
If you want to run this program as a server,
add the "-qws" command-line option.
```

3. 完成 8.3.2 小节中的猜数字游戏程序

使用 Qt Designer 设计游戏的界面，采用多重继承的方法编写代码，最后完成游戏的功能。

几点提示：

- 获取 QLineEdit 的输入内容可以用它的 text 函数
- 在 QLabel 中可以使用换行符获得多行显示的效果
- 随机数的产生可能用到 qsrand 和 qrand 函数。

附录 A　使用 QEMU 完成实验

A.1　QEMU 简介

QEMU 是一个处理器的仿真软件。与第一章介绍的 VirtualBox 类似，它可以采用软件的方式虚拟一套硬件系统，并在这个虚拟硬件系统中执行程序或者操作系统的代码。与 VirtualBox 不同的是它支持多种不同的处理器仿真，例如 x86、PowerPC、ARM、Sparc 等。使用这个软件可以实现在 x86 平台执行其他平台的二进制代码。

QEMU 对 ARM 处理器有很好的支持。可以仿真执行 armv5tej 的指令集，支持 ARM9E、ARM10E 和 XScale 等多种处理器。还支持多种实际的开发平台，例如 Integrator/CP、Versatile、mainstone、Spitz PDA。本小节将介绍如何用 QEMU 完成大多数嵌入式系统的实验内容，以帮助没有硬件平台的读者完成书中的大部分实验。

虽然 QEMU 支持多种 ARM 平台，但对于完成本书中的全部实验仍有不足，为了能完成更多的实验内容，需要对 QEMU 的源代码进行修改。下载文件提供了本书所使用的 QEMU 源代码与补丁文件。

解压 QEMU 并应用补丁

```
$ tar xfvz qemu-0.12.3.tar.gz
$ cd qemu-0.12.3
$ patch -p1 < ../patch_qemu_0.12.3.diff
```

使用 root 用户编译安装 QEMU，安装后的 QEMU 默认安装到了/usr/local/bin 目录下：

```
# ./configure --target-list="arm-softmmu arm-linux-user"
# make
# make install
```

由于我们只关心 QEMU 对 ARM 处理器的仿真，因此我们在配置 QEMU 的时候指定了 `--target-list` 选项，仅编译 ARM 相关部分，可以大大减少编译的时间。如果在配置的过程中出现找不到 zlib 的错误，则还需要安装相应的程序包：

```
# apt-get install zlib1g-dev
```

QEMU 支持两种仿真模式，一种是用户态模拟，支持在 x86 平台运行交叉编译的 ARM 平台 Linux 应用程序。另外一种是系统模拟，支持在 x86 平台运行交叉编译的 ARM 平台系统级软件，例如 Linux 内核、U-Boot 等。用户态模拟使用的命令是 qemu-arm，系统模拟用的命令是 qemu-system-arm。

我们可以用：

```
$ qemu-system-arm -M ?
```

来查看当前支持的嵌入式平台，本书主要使用 Versatile/pb 平台。Versatile 是一款 ARM 处理器应用的原型测试平台，可以适用于体系结构和 CPU 评估、硬件和软件设计及 ASIC 模拟。在 Linux 内核与 U-Boot 软件中都有对 Versatile 的很好支持，便于读者成功地完成实验内容。

A.2 Flash 操作的仿真

官方的 QEMU 暂时不支持 Versatile 平台的 Flash 仿真，但网络上已经有人完成了对这个功能的扩展，下载文件提供的 QEMU 补丁已经包含了这部分更新。QEMU 用一个映像文件作为仿真的 Flash，文件的内容可以看作完整 Flash 的拷贝，因此写入 Flash 的操作可以在 Linux 系统中用文件工具来实现。

首先使用 dd 命令建立一个空白的映像文件：

```
$ dd if=/dev/zero of=flash.img bs=64k count=1k
```

dd 命令从 if 参数指定的文件读取数据，并写入到 of 参数指定的文件中。每次读取的数据大小由 bs 参数指定，而由 count 参数指定最终读取的块数。由于从/dev/zero 文件中读取的数据都是 0，因此我们获得了一个 64MB 大小的 flash.img 文件，文件的内容全部是 0。

依然使用 dd 命令将 U-Boot 映像写入到 Flash 映像文件中。

```
$ dd if=u-boot.bin of=flash.img bs=64k conv=notrunc
```

由于 u-boot.bin 的长度比 flash.img 文件小，最后的 conv=notrunc 参数使得文件在拷贝结束后 flash.img 文件不被截断。

采用同样的方法还可以把下载文件提供的 Linux 内核与文件系统写入到 Flash 中。

```
$ dd if=uImage.qeum of=flash.img bs=64k seek=8 conv=notrunc
$ dd if=rootfs.qeum of=flash.img bs=64k seek=32 conv=notrunc
```

dd 命令中的 seek 参数指定了写入文件的偏移位置，因此，经过上面的操作，Flash 映像中的内容将分别包含 U-Boot、内核与文件系统。

由于 U-Boot 属于系统软件，因此这里使用 qemu-system-arm 命令启动虚拟机

```
$ qemu-system-arm -M versatilepb -m 256 -nographic -pflash flash.img
```

其中 -M 参数指定了我们私用的平台为 versatilepb。-m 参数指定了虚拟机所拥有的内存（单位为 M）。-nographic 参数关闭了 QEMU 的图形支持，而虚拟机的串口会被重映射到终端，可以完全用命令行的方式来调试系统。-pflash 参数指定了虚拟机所用的 Flash 映像文件，即我们在前面建立的文件。

上面命令的输出结果为：

```
U-Boot 2009.08 (11 月 15 2009 - 22:47:33)

DRAM:   0 kB
Flash: 64 MB
In:    serial
Out:   serial
Err:   serial
VersatilePB #
```

最后获得的 “VersatilePB #” 即 U-Boot 的提示符。可以在其中输入 U-Boot 命令。但这时要退出当前仿真界面的最好办法恐怕就是在其他终端中输入

```
$ killall qemu-system-arm
```

为了更真实地模拟实际的情况，还可以通过 -serial 参数将串口终端进行重定向，例如

```
$ qemu-system-arm  -M versatilepb -m 256 -nographic -pflash flash.img -serial pty
char device redirected to /dev/pts/10
QEMU 0.11.0 monitor - type 'help' for more information
```

可以从提示信息中看出，当前的串口被重新定向到了/dev/pts/10，可以用 minicom -s 命令设置串口终端的参数，连接这个设备。minicom 的具体使用方法可以参考第二章的介绍。设置完成进入 minicom 后就可以看到 U-Boot 的提示符并可以对虚拟机进行操作。

QEMU 对网络有很好的支持，默认情况下就支持一种用户态的网络协议栈。也就是说可以不必对 Linux 系统进行修改，也不必拥有 root 权限就可以在虚拟机中获得比较好的网络支持。例如我们可以在虚拟机的 U-Boot 环境中输入下面的命令：

```
VersatilePB # setenv ipaddr 10.0.2.15
VersatilePB # ping 10.0.2.2
Using MAC Address 52:54:00:12:34:56
host 10.0.2.2 is alive
```

命令中的 10.0.2.2 就是 QEMU 所模拟的一个 IP 地址，就好像虚拟机通过一根交叉网线连接到了一台 IP 地址是 10.0.2.2 的主机一样。同时这个 IP 地址上还支持 DHCP 服务，可以通过 DHCP 服务给虚拟机中运行的操作系统提供 IP 地址。

QEMU 的网络协议栈还可以模拟一个 tftp 的服务器，只要在运行 qemu-system-arm 的时候指定 -tftp PATH 的参数，就可以把 PATH 所对应的目录作为 tftp 服务的主目录，可以在虚拟机内用 tftp 协议下载文件。

例如我们运行 QEMU 的命令如下：

```
 $ qemu-system-arm  -M versatilepb -m 192 -nographic -pflash flash.img \
> -tftp /var/lib/tftpboot/
```

只要把 Linux 内核 uImage.qemu 放到/var/lib/tftpboot 目录下，就可以在虚拟机的 U-Boot 软件中用 tftp 命令下载 Linux 的内核文件。

```
VersatilePB # setenv ipaddr 10.0.2.15
VersatilePB # setenv serverip 10.0.2.2
VersatilePB # tftp uImage.qemu
Using MAC Address 52:54:00:12:34:56
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'uImage.qemu'.
Load address: 0x7fc0
Loading: ##################################################################
         ##################################################################
         ############################
done
Bytes transferred = 1465004 (165aac hex)
```

由于 QEMU 支持 DHCP 协议，我们也可以利用 U-Boot 的 dhcp 命令，自动获取 IP 地址并下载映像文件：

```
VersatilePB # setenv bootfile uImage.qemu
VersatilePB # dhcp
Using MAC Address 52:54:00:12:34:56
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'uImage.qemu'.
Load address: 0x7fc0
Loading: ##################################################################
         ##################################################################
         ############################
done
Bytes transferred = 1465004 (165aac hex)
```

A.3 使用网络文件系统

得益于 QEMU 对网络功能的良好支持，在 QEMU 环境中也很容易使用网络文件系统作为 Linux 的根文件系统。首先要对 NFS 服务的共享配置进行修改，在/etc/exports 加入对本机地址的共享支持，同时加入 insecure 选项。例如本机的 IP 地址为 192.168.0.20，则可以增加如下内容：

```
/home/  192.168.0.20/32(rw,no_root_squash,no_all_squash,insecure)
```

在 exports 文件中设置 insecure 选项是由于在 QEMU 中使用 NFS 会默认使用“非安全”的端口号（端口号大于 1024），如果不设置这个选项就会在错误日志中看到“illegal port”的错误消息，从而网络文件系统将挂载失败。同时为了保证 NFS 服务器工作正常，最好在 exports 文件中仅包含一条共享记录，避免其他记录影响实验的正常完成。

NFS 服务器设置完成之后，可以根据上个小节中介绍的方法，首先将 Linux 内核映像下载到“内存”中（使用 dhcp 命令），然后设置 bootargs 参数，并通过 bootm 命令启动 Linux 内核。

```
VersatilePB # setenv bootargs root=/dev/nfs mem=128M ip=dhcp \
nfsroot=192.168.0.20:/home/embedded/atmel_root/ console=ttyAMA0
VersatilePB # bootm
## Booting kernel from Legacy Image at 00007fc0 ...
```

在执行 bootm 命令之后就应该可以看到 Linux 内核的启动消息了，最后得到一个登录提示符，输入用户名和密码就可以登录到 Linux 系统中。

在设置 bootargs 参数的时候，有几个地方与前面实验中介绍的不同。首先是增加了 mem 参数，用来设置 versatilepb 平台的内存大小。还有 ip 参数并没有设置固定的地址，而是设置为 dhcp，利用 QEMU 所提供的 DHCP 服务自动获得 IP 地址。最后就是 console 参数定义为 ttyAMA0，这是 versatilepb 平台对其串口的命名。如果这个参数设置错误，在内核启动的时候就将无法在 QEMU 的监控界面看到内核启动消息。

为了保证网络文件系统启动正常，还要在文件系统中进行两处修改。一是在文件系统的/dev 目录中，必须存在 ttyAMA0 设备文件。这个设备文件的设备号可以参考下面的代码：

```
# mknod ttyAMA0 c 204 64
# mknod ttyAMA1 c 204 65
```

另外一处修改是/etc/inittab 文件，需要保证在串口 ttyAMA0 上启动登录程序，否则在 Linux 启动结束后就会看不到登录提示符。例如在使用 BusyBox 的 init 程序时，inittab 的相应设置如下：

```
ttyAMA0::respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

如果需要模拟开发板在 LCD 显示器上的输出，就可以在调用 QEMU 命令的时候去掉 nographic 参数，例如：

```
$ qemu-system-arm -M versatilepb -m 256 -pflash flash.img -serial pty \
> -tftp /var/lib/tftpboot/
```

此时一定要用串口模拟，因为 U-Boot 在 LCD 上没有输出，必须用串口对板子进行控制。用 minicom 连接到模拟串口之后，输入 U-Boot 命令控制仿真开发板启动。为了在 Linux 启动之后可以在模拟的 LCD 上看到输出，还需要对前面介绍的参数做小的修改。这个修改和前面有关 ttyAMA0 的修改原理是一致的，首先要在 bootargs 参数中加入 console=tty0，其次还要在 inittab 中加入：

```
tty0::respawn:/sbin/getty 38400 tty0
```

对于显示器设备，在 Linux 中的处理方式与串口类似，在调用 getty 程序的参数中甚至还保留有波特率的设置。Linux 会把一个显示设备虚拟为多个终端，分别被称为 tty0、tty1…

默认情况下 QEMU 通过 SDL 库直接在屏幕上显示虚拟的显示器，如果系统不支持 SDL[1]，QEMU 还可以通过一个虚拟的 VNC 服务器将图像输出。一般 Linux 发行版都带有一个 vinagre 程序，可以用来连接 VNC 服务器并显示相应的图像。

A.4 使用 QEMU 的虚拟硬件

第三章和第四章的实验内容要求对开发板的硬件进行编程控制，而在 QEMU 的环境中，硬件资源都是仿真实现的，因此依然可以采用类似的方法来操作虚拟硬件。

QEMU 针对 versatilepb 硬件实现了鼠标、键盘、LCD 等设备，但使用程序控制比较方便的还是通用串行口。versatilepb 使用的串口是 PrimeCell UART，为了通过它进行简单的串口通信，必须了解如下几个寄存器的设置，如表 A.1 所示。

表 A.1: PL011 串口寄存器

寄存器	偏移地址	描述
UARTDR	0	数据寄存器
UARTFR	0x18	标志寄存器

其中 UARTFR 中我们需要关心的是第 7 个二进制位（TXFE）和第 4 个二进制位（RXFE）。TXFE 如果设置为 1 表示发送 FIFO 为空，可以安全地写入新的数据到数据寄存器；RXFE 如果设置为 0 表示接收 FIFO 不空，这样就需要从数据寄存器读入新的值。另外需要知道的就是 UART 的起始地址，在实验中可以使用第一个串口的起始地址：0x101f1000。程序的加载地址一般可以设置为 0x7fc0，相应需要更新链接脚本的设置。相应的源代码如下所示：

```
#define DR *(unsigned long *)0x101f1000
#define FR *(unsigned long *)0x101f1018
```

[1]在编译 QEMU 的时候如果系统中缺少 SDL 开发包，就会使得编译生成的 QEMU 不支持 SDL

```
#define TXFE (1 << 7)
#define RXFE (1 << 4)

int main(void)
{
        int i;
        for (;;)
        {
                while ((FR & RXFE));
                i = DR;
                while (!(FR & TXFE));
                DR = i;
                if (i == 27) break; // 'Esc' for exit
        }
}
```

测试过程如下：

```
VersatilePB # dhcp boot.bin
Using MAC Address 52:54:00:12:34:56
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'boot.bin'.
Load address: 0x7fc0
Loading: #
done
Bytes transferred = 616 (268 hex)
VersatilePB # go 0x7fc0
## Starting application at 0x00007FC0 ...
Hello, world... ## Application terminated, rc = 0x1B
VersatilePB #
```

如果使用 QEMU 调试比较底层的代码，有一个类似“调试 LED”的设备会让工作变得简单。因此我们提供的 QEMU 也增加了这样一个设备，它的物理地址是 0x10000000，当程序在这个地址写入数据的时候，QEMU 会打印出调试信息，将写入的值显示出来。例如在 U-Boot 中可以这样测试：

```
VersatilePB # nm 10000000
10000000: 00000000 ? ffab
Led value: ffab
10000000: 0000ffab ? 45672
Led value: 45672
```

其中“Led value”就是 QEMU 打印的调试信息。无论被调试的代码是 U-Boot 还是 Linux 内核，只要在 LED 的物理地址中写入数据，就立刻能在屏幕上看到调试信息。利用这个虚拟的 LED 设备就可以完成第三章的实验内容。

在 QEMU 中增加新的设备并不是很复杂，感兴趣的读者可以参考 QEMU 的源代码，或者参考本书下载文件提供的补丁文件，为 QEMU 添加新的仿真硬件，便于自己软件的调试。

附录 B　使用 OpenEmbedded 构建文件系统

B.1　OpenEmbedded 简介

在嵌入式开发中，为嵌入式平台构建文件系统是一件非常繁琐的过程。本书在第五章介绍了修改现有文件系统，并利用 BusyBox 组成文件系统核心程序的步骤。但使用这种方法建立的文件系统只包含最基本的软件，如果要添加其他程序就必须一一进行交叉编译和安装，非常浪费时间也容易出现错误。OpenEmbedded 是一款基于 BitBake 的自动化嵌入式操作系统编译工具。它通过一系列的配置文件控制其工作过程，可以很方便的生成复杂的文件系统。由于 OpenEmbedded 最初是为 PDA 所设计的，使用它可以制作出功能丰富程度类似于桌面发行版的文件系统，包括浏览器、媒体工具、个人信息管理软件等。

其他类似的编译工具还有很多，例如 Buildroot 和 Scratchbox。Buildroot 可能是这类软件中最简单的，它通过一个类似于内核编译的菜单接口控制编译过程，可以编译交叉工具链，也可以交叉编译生成整个文件系统。Buildroot 还提供了简单的应用补丁机制，方便开发人员对源代码的管理。Scratchbox 的工作方式与 Buildroot 和 OpenEmbedded 都不同，它会建立一个特殊的根文件系统环境，主机利用这个环境可以采用非交叉编译的方式生成目标平台代码。具体的实现方法是在这个根文件系统环境中，所用的工具链是目标平台的本地编译工具，并通过 QEMU 来调用这些工具。这样在为目标平台编译程序的时候就不必采用交叉编译的方式，简化了编译过程。采用 QEMU 的好处还体现在编译好的代码可以直接用 QEMU 调用执行，而不必下载到目标平台进行测试。Scratchbox 在 Nokia 的 Maemo 项目中已经被成功地应用。

BitBake 是 OpenEmbedded 的重要组成部分，它是一款基于 Python 脚本的自动化编译工具，简单地讲，它的作用是执行编译任务和管理元数据（metadata）。这些功能与 GNU make 和其他的自动化编译工具有很多的共同之处。BitBake 是由 Gentoo Linux 发行版下的软件包管理工具 Portage 衍生而来，并且在之后成为了 OpenEmbedded 项目的基础。

BitBake 的主要特点如下：

- 使用 Python 实现，类似 Gentoo 的包管理工具。
- 自动下载源代码，默认使用最新的可用版本，应用补丁文件。
- 可以生成交叉编译工具链，版本可配置。
- 支持 glibc 和 uClibc。
- 支持不同的目标平台。
- 可以正确处理软件的依赖关系，生成完整的文件系统。
- 可以被用来编译单个的软件包
- 易于用户使用，并且用户能够比较简单地提供自己编写的元数据。
- 能够简单整合多个利用 BitBake 编译的项目。
- 能够通过继承的方式，在软件包之间分享公共的元数据。

B.2 配置和使用 OpenEmbedded

在使用 OpenEmbedded 之前要首先明确，OpenEmbedded 需要的硬盘空间非常大（可能会超过 20G），其中包括元数据文件、编译需要的源代码、编译生成的目标文件和中间文件。OpenEmbedded 的工作目录不能在符号链接下面，也就是说工作目录路径上的每个目录都不能是符号链接。下面假定我们的工作目录名字是 oe，OpenEmbedded 所需要的文件都将保存在这个目录下。

首先我们要下载 BitBake 软件，并把它放在 oe 目录之下，下载的时候可以到官方网站检查最新的版本下载，这里以 1.8.18 版本为例。下载并解压之后，需要设置 PATH 环境变量以使用 bitbake 命令。

```
$ wget http://download.berlios.de/bitbake/bitbake-1.8.18.tar.gz
$ tar xfvz bitbake-1.8.18.tar.gz
$ export PATH=$PWD/bitbake-1.8.18/bin:$PATH
```

OpenEmbedded 的元数据通过 Git 版本控制系统管理，通过下面的命令可以获取最新的元数据集合：

```
$ git clone git://git.openembedded.net/openembedded
```

接下来要设置本地的配置文件（local.conf），这个文件可以基于参考配置文件（local.conf.sample）修改，相关文件的位置可以参考下面的命令列表。

```
$ mkdir -p build/conf
$ cp openembedded/conf/local.conf.sample build/conf/local.conf
$ vi build/confi/local.conf
```

配置文件主要修改的内容包括如下几行

```
DL_DIR = "/opt/oe/sources"
BBFILES = "/opt/oe/openembedded/recipes/*/*.bb"
DISTRO = "angstrom-2008.1"
MACHINE = "at91sam9261ek"
TARGET_OS = "linux"
```

OpenEmbedded 在进行编译的时候会自动从官方网站下载源代码，DL_DIR 变量指定了源代码的下载位置，示例中假定 oe 目录在 “/opt” 目录下，这里必须使用绝对路径。可以在编译不同镜像时都采用一个相同的 DL_DIR，这样可以避免源代码的重复下载，缩短 BitBake 编译镜像所用时间。

BBFILES 变量指定了元数据的位置，OpenEmbedded 的元数据多数是使用 bb 作为文件的扩展名。

DISTRO 变量指定了一个发行版的名字，不同的发行版包含不同的软件包，angstrom 是使用 OpenEmbedded 的一个发行版，可供多种 PDA 系统使用。这个发行版的定义在 openembedded/conf/distro 目录中。

MACHINE 变量定义了一个开发平台的名字，每个开发平台也由元数据进行定义，平台数据的定义在 openembedded/conf/machine 目录中。其中有一些平台是专门为 QEMU 使用的，方便开发人员的测试，例如 qemuarm 和 qemumips 分别是为 ARM 平台和 MIPS 平台的 QEMU 模拟设置。

TARGET_OS 变量定义了目标平台所运行的操作系统，这里设置为 linux。

在示例配置中还包括了一个 REMOVE_THIS_LINE 的变量定义，必须把它注释掉或者删除才能进行后面的编译。OpenEmbedded 这么做的目的就是希望用户真正阅读了配置文件，而不是仅仅拿过来盲目的使用。

最后设置 BBPATH 环境变量后就可以调用 bitbake 命令编译系统了。BBPATH 用来指定元文件的位置，一般设置为 build 和 openembedded 目录。bitbake 命令需要指定编译的目标类型。例如 “bitbake nano” 可以编译 nano 软件包，“bitbake x11-image” 可以编译包含 x11 的组合软件包。下面是具体调用 bitbake 的示例：

```
$ export BBPATH=/opt/oe/build:/opt/oe/openembedded
$ bitbake x11-image
```

编译 x11-image 需要下载几百个源码包，并分别进行编译，因此需要的时间很长。最后生成的目标文件不光包含嵌入式文件系统，还包括相应的内核与启动管理器等。

参 考 文 献

[1] [美] Wayne Wolf. 嵌入式计算系统设计原理. 孙玉芳等译. 北京：机械工业出版社.

[2] [荷] Andrew S. Tanenbaum. 现代操作系统. 陈向群等译. 北京：机械工业出版社.

[3] [美] David Tansley. Linux 与 UNIX Shell 编程指南. 徐焱译. 北京：机械工业出版社.

[4] [美] Richard Stevens. UNIX 环境高级编程（英文版第二版）. 北京：人民邮电出版社.

[5] [美] Jonathan Corbet. Linux 设备驱动程序（影印版第 3 版）. 南京：东南大学出版社.

[6] [美] Daniel P.Bovet. 深入理解 Linux 内核（影印版第 3 版）. 南京：东南大学出版社.

[7] [加] 布兰切特. C++ GUI Qt 4 编程（第二版）. 闫锋欣等译. 北京：电子工业出版社.

[8] [美] Brian E. Hall. Beej's Guide to Network Programming. 网络资源.

[9] ARM Architecture Reference Manual. ARM 公司.

[10] AT91SAM9261 数据手册. Atmel 公司.

[11] The GNU linker. GCC ld 命令文档.

[12] Linux 内核文档. 网络资源.

[13] PrimeCell UART (PL011) Technical Reference Manual. ARM 公司

[14] Open Embedded 在线文档. 网络资源.