第1章　嵌入式系统入门

时下计算机领域最热门的一个词可能就是嵌入式系统了。翻开计算机的杂志和书籍，许多地方都能见到嵌入式系统的字眼。其实，不仅在书籍杂志上，嵌入式系统和每个普通人的生活联系都是很紧密的。本章从应用角度出发介绍什么是嵌入式系统，带读者进入嵌入式系统开发的领域，主要内容包括：

嵌入式系统定义

嵌入式系统的应用领域

嵌入式系统的发展趋势

1.1 什么是嵌入式系统

对于没有接触过的人来说，嵌入式系统这个词可能显得比较深奥，甚至充满一些神秘色彩。其实嵌入式系统和普通人的生活非常紧密，如日常生活中使用的手机、微波炉、有线电视机顶盒等，都属于嵌入式系统。与通常使用的PC机相比，嵌入式系统的形式变化多样、体积小，可以灵活地适应各种设备的需求。因此，可以把嵌入式系统理解为一种为特定设备服务的，软件硬件可裁剪的计算机系统。嵌入式系统的英文名称是Embedded System。

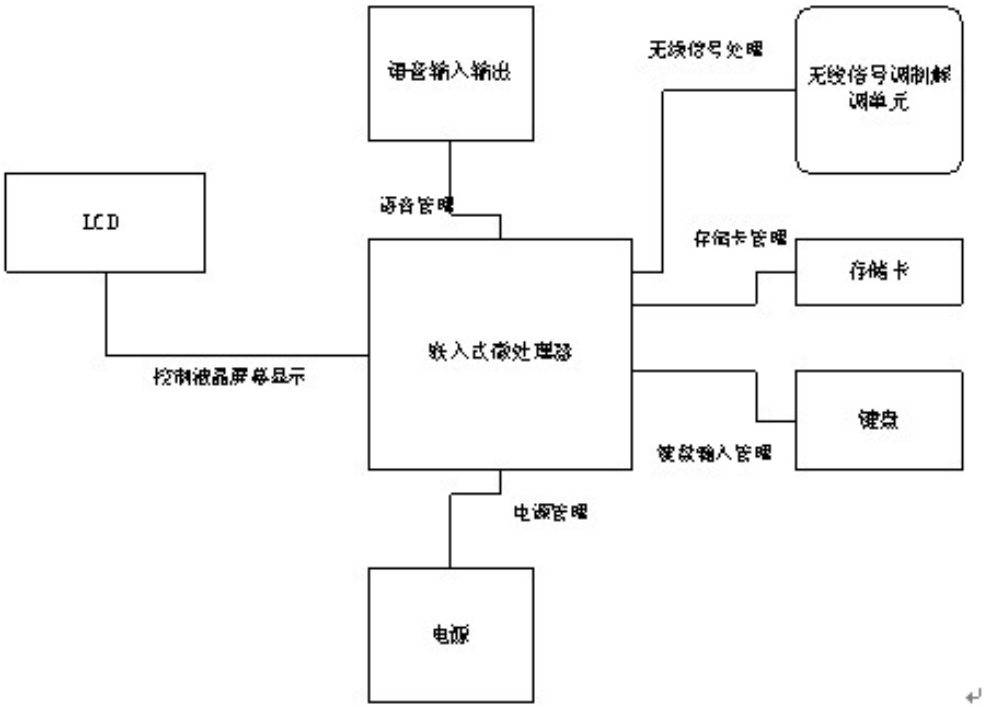
从嵌入式系统的定义可以看出，一个嵌入式系统具备了体积小、功能专一、软硬件可裁剪的特点。这些特点也能反映出嵌入式系统与传统的PC机有着不同之处。本书使用常见的ARM嵌入式系统为例讲解嵌入式Linux系统移植和开发技术。

1.2 嵌入式系统应用领域

从嵌入式系统的特点可以看出它的应用领域是很广泛的。不仅在家电上，在其他的领域也有很大的需求。本节介绍一些嵌入式应用的领域。

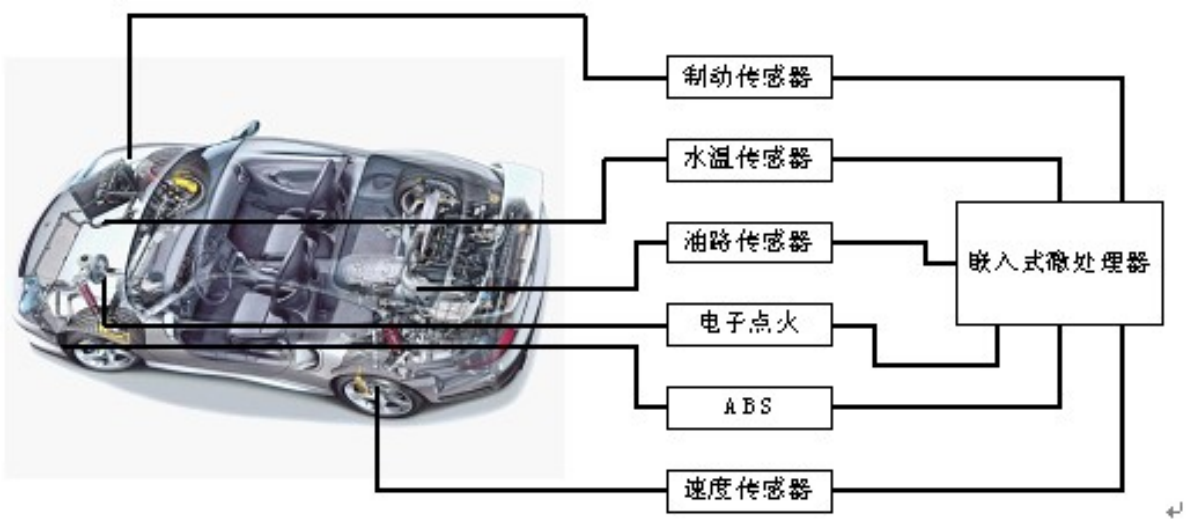
1.2.1 家用电器

家用电器里最常见的可能就算是手机了。手机是一个典型的嵌入式系统。



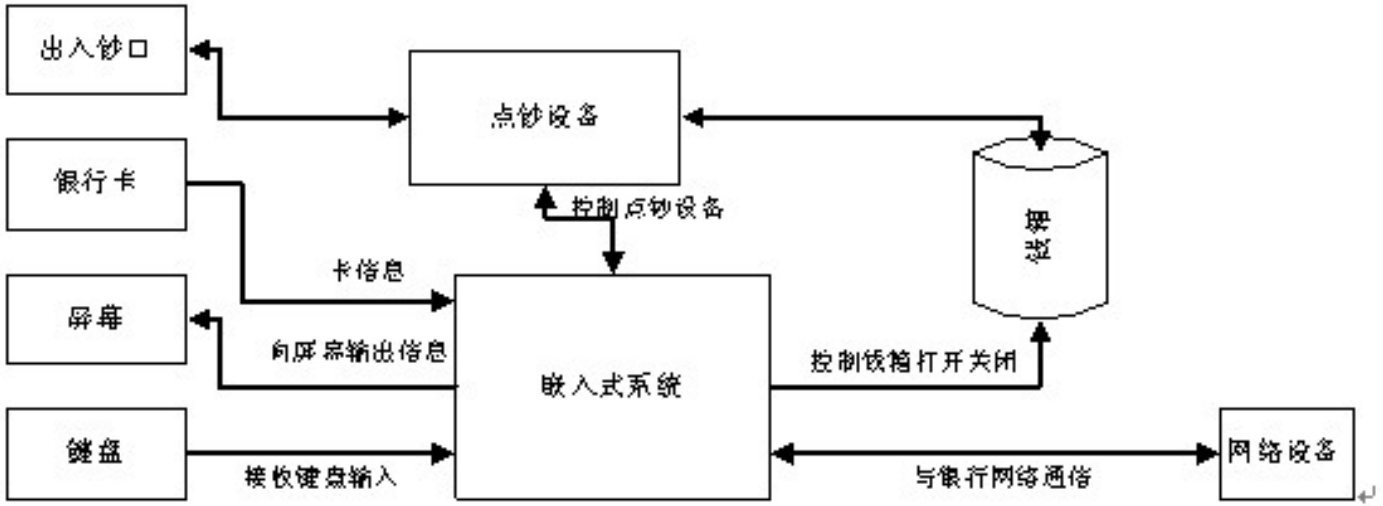
1.2.2 交通工具

大家最常使用的交通工具就是汽车了，不管是公交车、私家车还是其他的各种专用车辆，都有嵌入式系统的身影。



1.2.3 公共电子设施

银行的ATM自动取款机是一种常见的公共电子设备。

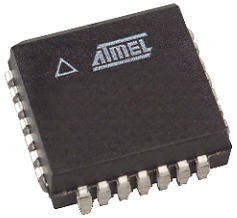


1.3 嵌入式系统发展

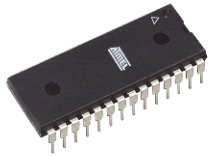
从1946年第一台现代电子计算机诞生以来，计算机始终朝着两个方向发展：一个方向是体积大型化、处理能力超强的大型计算机；另一个方向是超体积小型话、功能多样化发展。这两个发展方向没有高低之分，都是由于实际需求带动发展。计算机大型化发展出现了很多的超级计算机，各国都在研制自己的大型计算机。在计算机小型化的发展道路上，可谓是种类繁多。不仅有个人电脑（PC），还有各种个人数字助理（PDA）。嵌入式系统是计算机系统小型化发展的一个热门的分支。

1.3.1 嵌入式微控制器

嵌入式微控制器也就是传统意义上的单片机，它可以说是目前嵌入式系统的前身。单片机就是把一个计算机的主要功能集成到了一个芯片上，简单说就是一个芯片就是一个计算机。它的特点是体积小、结构简单、便于开发以及价格经济。



扁平封装的8051系列芯片



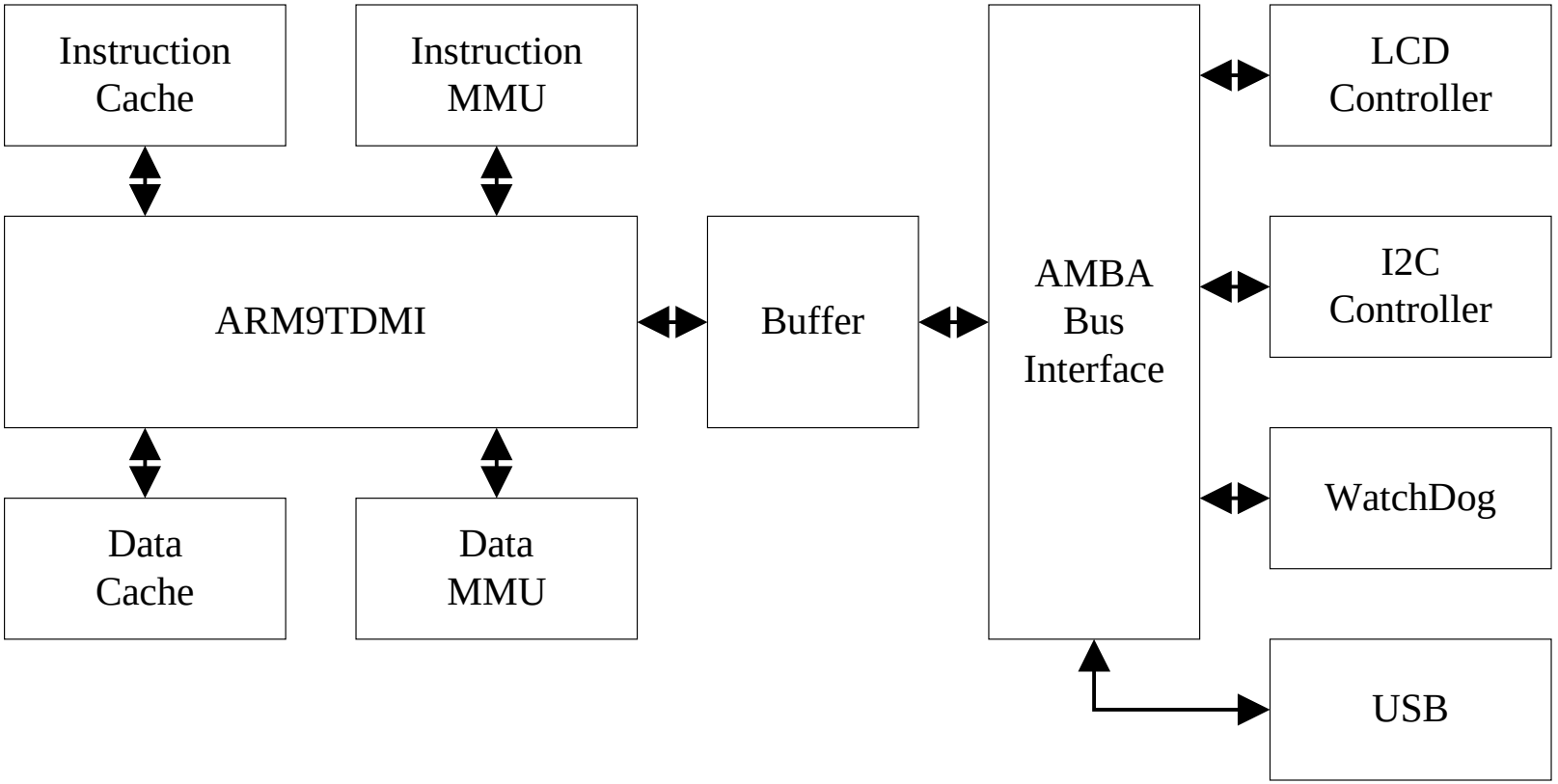
直列封装的ATMega8芯片

1.3.2 嵌入式微处理器

单片机的发展时间较早，处理能力很低，只能用在一些相对简单的控制领域。嵌入式微处理器是近几年随着大规模集成电路发展同步发展起来的。与单片机相比，嵌入式微处理器的处理能力更强。目前主流的嵌入式微处理器都是32位的，而单片机多是8位和16位的。

嵌入式处理器在一个芯片上集成了复杂的功能，同时一些微处理器还把常见的外部设备控制器也集成到芯片内部。以ARM芯片为例，ARM体系在内部规定了一个32位的总线，厂商可以在总线扩展外部设备控制器。三星的ARM9芯片S3C2440A把常见的串行控制器、RTC控制器、看门狗、I2C总线控制器甚至LCD控制器等都集成在了一个芯片内，可以提供强大的处理能力。

1.3.2 嵌入式微处理器


Instruction Cache
Instruction MMU
ARM9TDMI
Buffer
AMBA Bus Interface
LCD Controller
I2C Controller
WatchDog
USB
Data Cache
Data MMU

1.3.3 未来嵌入式系统发展的方向

随着微电子技术的不断发展以及电子制造工艺的进步（目前已经可以制造出65纳米的芯片），嵌入式系统硬件的体积会不断缩小，系统稳定性也在不断增强，可以把更多的功能集成在一个芯片上。另外在功耗方面也不断降低，这样使嵌入式设备在自带电源的情况（如使用电池）会使用更长的时间，而且设备的功能也更强大。

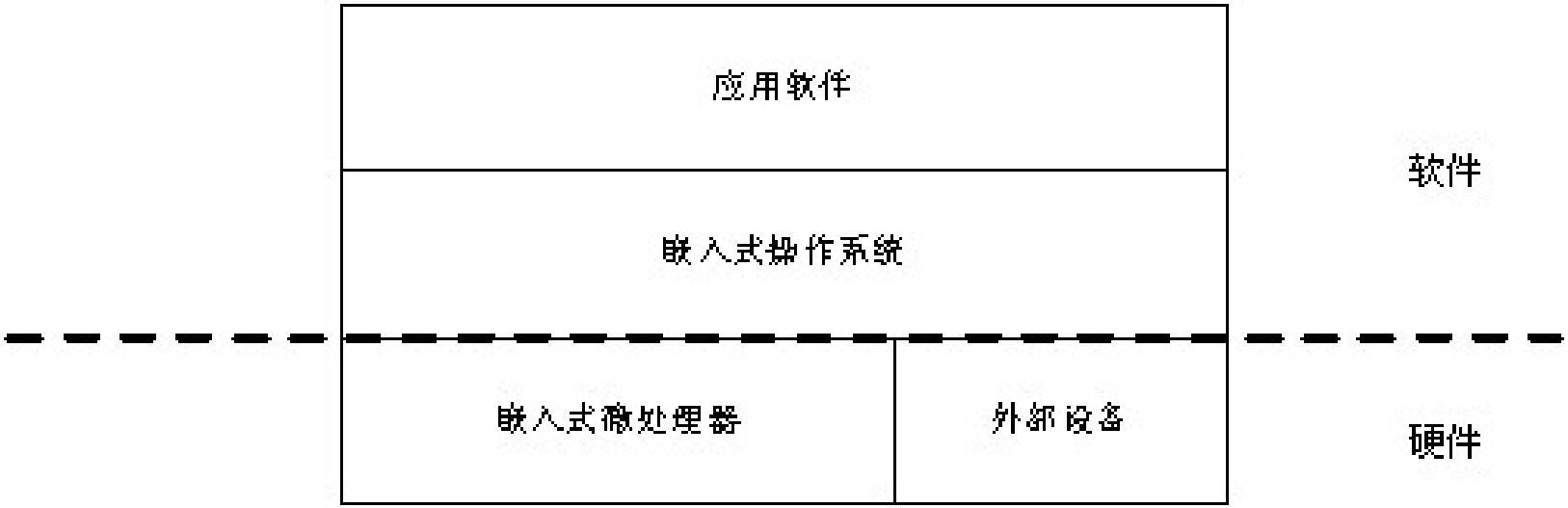
此外随着网络的普及和IPv6技术的应用，越来越多的嵌入式设备也会加入到网络中。将来家中的微波炉或者洗衣机都可以通过无线接入网络，被其他设备控制。

1.4 典型的嵌入式系统组成

嵌入式系统与传统的PC机一样，也是一种计算机系统，是由硬件和软件组成的。硬件包括了嵌入式微控制器和微处理器，以及一些外围元器件和外部设备；软件包括嵌入式操作系统和应用软件。

与传统的计算机不同的是，嵌入式系统种类繁多。许多的芯片厂商、软件厂商加入其中，导致有许多的硬件和软件，甚至解决方案。一般来说，不同的嵌入式系统软硬件是很难兼容的，软件必须修改而硬件必须重新设计才能使用。虽然软硬件种类多，但是不同的嵌入式系统还有很多相同之处的。

1.4 典型的嵌入式系统组成



1.5 小结

本章是全书的第一章，为读者介绍了嵌入式系统的基本常识、组成结构，通过实例给读者对嵌入式系统一个初步的认识。本章的知识相对比较笼统，偏重一些概念方面的介绍，读者可以结合实际生活理解，加深对嵌入式系统的认识。第2章讲解嵌入式软件和硬件系统基本知识。

第2章　嵌入式软硬件系统

在学习嵌入式开发之前，需要了解一下嵌入式系统基本知识。嵌入式系统是由软件和硬件组成的，与传统的PC不同，在设计嵌入式系统的时候通常是软件和硬件都需要设计。对于一个嵌入式系统开发来说，无论是硬件开发人员还是软件开发人员，都需要掌握基本的软件和硬件知识。本章的主要目的是通过讲解基本的软硬件知识，帮助读者建立嵌入式系统概念，主要内容包括：

模拟电路和数字电路

基本的数制转换

计算机的工作原理

软件的基础知识

操作系统概要

2.1 电路基本知识

初中物理知识讲过，电流就是电流通过的路径。一个最简单的电路是由电源、负载和导线构成的。复杂的电路还有电阻、电容、晶体管、集成电路等元件。这些元件的功能不同，通过不同的组织方式构成了不同功能的电路。无论什么样的电路，最终的功能都是处理电子信号的。按照电子信号的工作方式可以把电路分成模拟电路和数字缏贰

2.1.1 什么是模拟电路

处理模拟信号的电路称作模拟电路。模拟信号的特点是信号是线性变化的，意思是信号变化是连续的。如经常使用的收音机、电视机和电话都是使用的模拟信号。常见的模拟电路有变压电路、放大电路。评估一个模拟电路常见的参数有放大率、信噪比和工作频率等。模拟电路是数字电路的基础，数字电路可以看做是模拟电路的一种特殊形式。

2.1.2 什么是数字电路

数字电路顾名思义是处理数字信号的电路，通常数字电路具有逻辑运算和逻辑处理的功能。与模拟信号不同，数字信号使用电压的高低或者电流的有无表示逻辑上的1或0，因此数字电路可以方便地表示出二进制数。数字电路可以分成脉冲电路和逻辑电路两部分，脉冲电路负责对信号变换和测量；逻辑电路负责处理数字逻辑。

与模拟电路不同，数字电路关心的是信号状态的变化。通过数字逻辑可以处理复杂的二进制信息，因此数字电路是计算机的基础。由于数字电路的工作特点，它具有电路结构简单、容易加工制造等优点，可以大批量的生产制造，成本也变得低廉。数字电路广泛应用在测量、科学计算、自动控制等领域。

2.1.3 数制转换

计算机是由数字电路构成的，其内部数据的传输和处理都使用二进制方式。日常生活中普遍使用十进制方式表示数字，所以在使用计算机的时候需要用到数制转换。常见的有二进制到十进制的转换，从事嵌入式开发经常会用到十六进制，有的时候还会用到八进制。

二进制的特点是“逢2进1”。如十进制的0对应二进制的0，十进制的1对应二进制的1，十进制的2对应二进制的10，以此类推。从这个推演规律中可以看出，二进制数从右往左每个位数都是2的位数次幂。举个例子，二进制数1010转换为十进制数：

$(1010)_2 = (2^3 x1) + (2^2 x0) + (2^1 x1) + (2^0 x0) = 8 + 0 + 2 + 0 = (10)_{10}$

2.1.3 数制转换

十六进制	二进制
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

2.2 计算机组成原理

现代计算机的构造越来越复杂，功能也日新月异。但是计算机的组成结构从本质来说仍然是相同的。一个计算机系统硬件是由中央处理器、存储系统、总线系统和输入输出系统几个基本部分组成的。本节从计算机系统结构发展的角度介绍计算机组成和工作原理。

2.2.1 计算机体系的发展

计算机是由硬件系统和软件系统两大部分组成的。按照功能又可以划分为指令系统、存储系统、输入输出系统等。计算机体系结构简单的说就是研究计算机各系统和组成部分结构的一门学问。计算机的诞生到现在时间仅有半个世纪，但是计算机体系结构却有很大的发展，出现了许多的体系结构设计思想和设计方法。从存储结构来说可以把计算机体系分成冯诺依曼结构和哈佛结构。

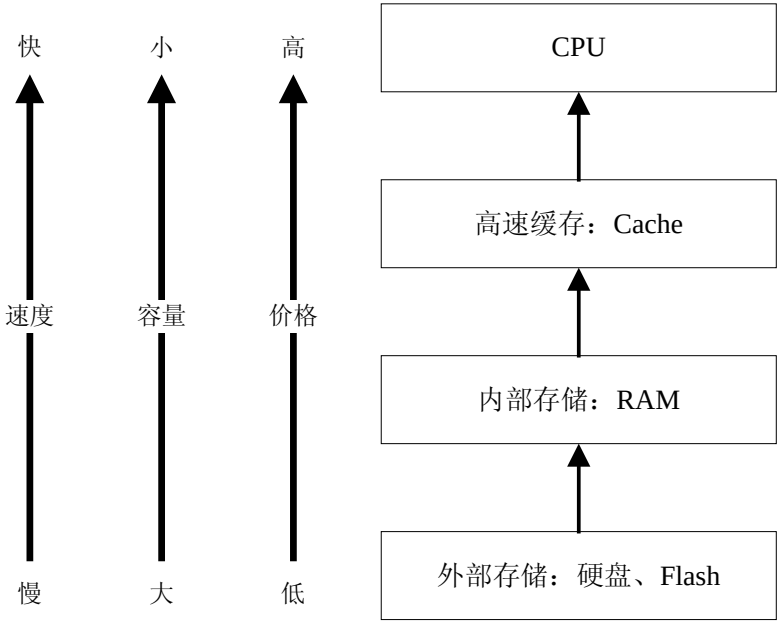
2.2.2 中央处理器

中央处理器英文全称是（Central Process Unit），简称CPU，是一个计算机系统的核心。CPU是由运算器、控制器、寄存器和内部总线组成的。在CPU之外再加入总线，存储设备输入输出设备就可以构成一个完整的计算机系统。

CPU有几个重要的参数，包括工作频率、字长、指令集和缓存。工作频率通常是用户最多听到的参数，一个CPU的工作频率包括了主频和外频以及外部总线频率。主频是CPU的实际工作频率，外频是CPU工作的基准频率，还有一个是总线的工作频率。一般来说，工作频率越高的CPU执行指令的速度就越快，但是也不完全如此。

2.2.3 存储系统

计算机在工作中需要从内部存储器中读取指令和数据，并且把计算的结果存入外部存储器。由于材料和价格因素的限制，计算机的存储器件在容量、速度等方面需要匹配。存储系统的作用就是设计一个让各种存储器相互配置达到最优性价比的方案。



2.2.4 总线系统

总线是CPU连接外部设备的通道。通常包括数据总线（DataBus）、地址总线（AddressBus）和控制总线（ControlBus）。地址总线负责向外部设备发送地址信息；数据总线负责从外部设备读取或者写入信息；控制总线负责发送信号控制外部设备。

计算机的总线系统是由总线和相应的控制器构成的。如嵌入式系统中常见的I^2C总线和SPI总线，特点是控制简单，成本低廉，本书后面会做具体介绍。其他的还有PCI总线，支持复杂的功能和很高的系统吞吐量。

总线的出现规范了CPU和外设之间的通信标准，简化了外部器件的设计。使用一些通用的总线可以有效降低开发成本。

2.2.5 输入输出系统

输入输出系统由外部设备和输入输出控制器组成，是CPU与外部通信的系统。CPU通过总线与输入输出系统相连。由于外部设备的速度差异，CPU可以使用不同的方式控制外部设备的访问。常见的有轮询方式、中断控制方式和DMA方式。

程序中断方式最简单，CPU通过不断的查询某个外部设备的状态，如果外部设备准备好，就可以向其发送数据或者读取数据。这种方式由于CPU不断查询总线，导致指令执行受到影响，效率非常低。

2.3 软件基础知识

嵌入式系统的基础是硬件，软件是嵌入式系统的灵魂。离开了软件，一个系统的功能就无法发挥。因此软件设计开发是嵌入式系统开发的一个重要环节。本节介绍软件的基础知识，开发流程以及基本的技术。

2.3.1 什么是软件

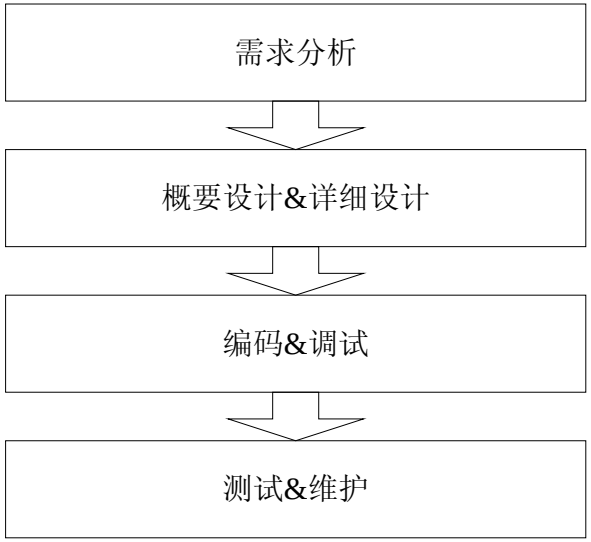
使用过计算机的读者都使用过各种各样的软件，例如最常见的Word文字处理软件，还有上网使用的浏览器等。严格的说，软件是由程序和文档构成的，程序是一组按照特定结构组织的指令和数据集合。

通常软件可以分成系统软件和应用软件，以及目前兴起的介于二者之间的中间件软件。系统软件是使用计算机提供的基本功能，例如操作系统和数据库系统。它们都不是针对某种特殊需求，而是面向通用的领域。应用软件是针对某种特殊需求设计的，一般来说具有专门的功能。比如MP3播放软件就是针对播放音乐设计的。

软件的另一个组成部分是文档。随着软件复杂程度的提高，文档也越来越重要。常见的软件文档有开发文档和用户文档，前者面向开发人员，后者面向最终用户。软件开发人员应该树立起编写文档的好习惯。

2.3.2 软件开发流程

软件开发流程是软件在开发过程中需要走的步骤，经过几十年的发展形成一套公认的开发流程。大致可以分成4个部分：需求分析、设计、编码调试、测试和维护。



2.3.3 常见的软件开发模型

软件的发展过程中，经过前人的总结，设计出几种软件开发模型。软件开发模型指定了软件开发流程中的规范和参考原则，指导开发人员按照特定的步骤工作。但是，由于现实的差异性，很少有适用与所有软件的开发模型，有一些经过验证比较有效的模型供开发人员参考。常见的有瀑布模型、演化模型、增量模型、变换模型、螺旋模型、喷泉模型、智能模型、快速应用开发模型等

2.3.4 计算机编程语言

计算机内部是通过执行指令完成各种操作的，无论是指令还是数据在计算机内部都使用二进制表示，对于用户的识别和输入都很困难。计算机编程语言就是为解决这个问题设计的。计算机编程语言是一种有规范格式和语法供人类描述计算机指令的字符串集合。举个例子来说，计算机内部使用二进制10100101表示一个求加法操作，计算机语言可以通过add字符串表示这个加法操作，便于人类识别。

计算机语言可以分成机器语言、汇编语言和高级语言。其中机器语言是供计算机本身识别的，为二进制串。汇编语言是对机器语言的抽象，其实质与机器语言是相同的。汇编语言的指令与机器语言是一一对应的。此外，汇编语言还设计了伪指令和宏指令，帮助编程人员提高开发效率。汇编语言是依赖体系结构的，在一种CPU上能执行的汇编语言在其他的CPU很可能就无法执行了。

2.3.5 数据结构

数据的物理结构描述数据元素的存储结构，依赖于具体的计算机实现。例如，一个统计表格是数据元素之间的逻辑结构，但是把表格存放到计算机中需要考虑存储结构，可以按照行的顺序存储，也可以按照列的顺序存储，这就是数据的物理结构。

数据结构还定义了数据元素上的操作方法，通常也称作算法。算法可以理解为一种思路。例如，对10个无序的数字按照大小排序，可以有冒泡排序、二分排序、插入排序等多种方法。在计算机编程中，一个好的算法可以起到事半功倍的效果。

2.4 操作系统知识

现代计算机的应用软件都是在操作系统下面工作的。嵌入式系统早期应用程序是直接运行在CPU上，比如单片机。随着嵌入式系统硬件处理能力的提升，应用也越来越复杂，目前主流的嵌入式系统都配备了操作系统，应用软件使用操作系统提供的功能。本节介绍操作系统的知识。

2.4.1 什么是操作系统

操作系统是一类特殊的系统软件。它管理整个系统的所有硬件和软件，通常是整个计算机系统中最接近硬件的系统软件。操作系统屏蔽了硬件的底层特性，向应用软件提供了一个统一的接口。对于应用软件来说，不需要知道硬件的具体特性，使用操作系统提供的接口即可完成相应的功能。除此之外，操作系统通过特定的算法统筹安排整个计算机系统软硬件资源，使计算机的资源利用率更高，甚至获得比硬件更多的功能。

操作系统是软件领域一个重要部分。常见的嵌入式操作系统有μcLinux、vxWorks等。本书第4章将详细讲解嵌入式Linux操作系统以及与其他系统之间的对比。

2.4.2 操作系统发展历史

最早的计算机没有操作系统。在同一时间，用户只能通过打孔机等外部设备把程序输入，计算机按照程序执行。如果程序出现问题，整个机器就会停止工作。后来把常用的程序设计成库装入计算机，方便用户使用，这可以算是操作系统的雏形。

早期的操作系统多种多样，在大型机领域，几乎每个系列的计算机都有自己的操作系统。这种方式造成资源很大的浪费，同样功能的程序在不同的机器上由于操作系统的不同无法运行。后来，AT&T公司在小型机上开发成功了UNIX操作系统（几乎同时C语言也诞生了），并且免费发放，用户可以修改其代码。UNIX的这种授权方式得到广泛应用，被移植到了各种计算机上，是现代操作系统的开端。UNIX操作系统的设计思想也是现在许多操作系统参考的基础。

2.4.3 操作系统由什么组成

按照软件的结构划分，操作系统可以分成内核、驱动程序和程序库。内核是操作系统的核心，也是整个系统软件的核心。一般来说，内核从抽象的层面提供最基本的功能，通常代码短小精炼。驱动程序是计算机系统必不可少的一类系统软件，系统是和驱动程序打交道而不会直接访问硬件，硬件的具体细节有驱动程序完成，是软硬件的接口。程序库是操作系统向用户提供的程序接口。

2.4.4 几种操作系统的设计思路

层次结构的设计思想是把内核需要提供的功能划分出层次，最底层仅提供抽象出来的最基本的功能，每一层利用下面的一层的功能，依此类推最上面的一层可以提供丰富的功能。这种设计思路结构清晰是操作系统内核的一大进步。

微内核结构是19世纪80年代产生的内核结构。其设计思想是内核提供最基本最核心的功能，注重把系统的服务功能和基本操作分开。例如，内核只提供中断处理、内存管理等基本功能，网络传输数据之类的功能可以设计成一个系统服务完成。这种设计思路使得内核的设计更加简单，内核可以根据需要启动或者关闭系统服务，极大的提高了整个系统的工作效率。此外，微内核还会设计一个硬件抽象层，对内核屏蔽硬件底层特性，让内核可以专注提供各种功能。使用微内核结构的系统越来越多，常见的Linux和WindowsNT都采用了微内核的设计思想。

2.4.5 操作系统分类

按照系统对任务的处理相应时间可以把操作系统分成分时系统和实时系统。分时系统中，不同用户的进程按照一定的策略分别得到CPU资源，未能得到资源的用户只能等待。实时系统则不然，任务是按照优先级和响应时间分配的，在一个设定的响应时间内，任务必须得到响应。例如导弹拦截系统，在收到导弹拦截请求后需要在特定的时间内得到响应。实时操作系统常用在军火、航天、电信等领域。分时操作系统应用很广泛，Linux就是一个性能优越的分时操作系统。

2.5 小结

本章概括了嵌入式开发领域软硬件的基础知识，包括电路、计算机组成原理、软件的基本知识以及操作系统。本章的知识点比较广泛，读者只需要了解即可，全书在涉及到的地方会详细讲解各知识点。第3章讲解ARM处理器。

第3章　ARM处理器

ARM既是一种嵌入式处理器体系结构的缩写，也是一家公司的名字。目前有数十家公司使用ARM体系结构开发自己的芯片，支持的外部设备和功能丰富多样。ARM体系相比其他的体系具有结构简单、入门快等特点。使用ARM核心的处理器虽然众多，但是核心都是相同的。因此，掌握了ARM的体系结构，用不同的处理器只要是基于ARM核心都能很快上手。本章的主要内容包括：

微处理器和微控制器的关系

ARM处理器介绍

ARM体系结构

ARM的功能选型

3.1 微处理器和微控制器

微处理器可以根据应用领域大致分成通用微处理器、嵌入式微处理器和微控制器。通用微处理器主要用于高性能计算，如常用的PC机的CPU就是一个通用微处理器；嵌入式微处理器是针对某种特定应用的高能力计算，如MP3的解码、移动电话的控制等；微控制器主要用于控制某种设备，通常集成了多种外部设备控制器，处理指令的能力一般不是很强，但是价格低廉，多用在汽车、空调等设备上。

微控制器除了针对专门设备设计以外，还具备微处理器不具备的特点。如很好的环境适应性，可以在特殊的高温或者低温环境工作。这些特点一般的微处理器是不具备的。目前的嵌入式微处理器大多集成了外部设备控制器，功能不断增强，价格也在下降。使用嵌入式微处理器替代微控制器俏蠢捶⑰沟那魇啤

3.2 ARM处理器介绍

ARM是英文Advanced RISC Machines的缩写，中文译为高性能RISC机器。从名称可以看出，ARM是一种基于RISC架构的高性能处理器。实际上ARM同时也是它的设计公司的名字。与其他的嵌入式芯片不同，ARM是由ARM公司设计的一种体系结构，主要用于出售技术授权，并不生产芯片。其他芯片设计公司可以通过购买ARM的授权，设计和生产基于ARM体系的芯片。

3.2.1 ARM微处理器的应用领域

在前面提到ARM已经渗透到许多的应用领域。

1. 工业控制
2. 无线通信
3. 网络应用
4. 消费电子产品

3.2.2 ARM的功能特点

ARM核心的处理器采用RISC体系结构，具有以下优点：

芯片体积小，功耗低，制造成本低，性能优异

支持Thumb（16位）和ARM（32位）两种指令集，8位和16位设备兼容性好

由于采用RISC架构，在内部大量使用寄存器，执行指令速度快

大部分的指令都是操作寄存器，只有很少指令会访问外部内存

采用多级流水线结构处理速度快

支持多种寻址方式，数据存取方式灵活

指令长度固定，便于编译器操作以及执行指令

3.3 ARM指令集

指令集指一个微处理器所有指令的集合，每种微处理器都有自己的指令集。在第2章讲过处理器的指令集可以分成CISC（复杂指令集）和RISC（精简指令集）两种，ARM处理器使用RISC（精简指令集）。

精简指令集的最大特点是所有的指令占用相同的存储空间。ARM处理器支持ARM和Thumb两种指令集：ARM指令集工作在32位模式下，指令长度都是32比特；Thumb指令集工作在16位模式下，指令长度都是16比特。

ARM指令集按照功能可以分为算术运算指令、逻辑运算指令、分支指令、软件中断指令和程序数据装载指令等。

3.3.1 算术运算指令

算术运算指令用于普通数据计算。常见的指令有ADD、ADC、SUB和SBC。

1. ADD指令
2. ADC指令
3. SUB指令
4. SBC指令

3.3.2 逻辑运算指令

逻辑运算不同与算术运算。逻辑运算按照逻辑代数的运算法则操作数据，得到逻辑结果。

1. AND指令
2. EOR指令
3. MOV指令

3.2.3 分支指令

在汇编语言中，代码的跳转都是通过分支指令完成了，ARM的分支指令比较简单，本书介绍最基本的分支指令——B指令。

B指令可以根据设置的条件跳转到指定的代码地址。

格式：B{条件} <地址>

B指令是分支跳转指令。程序中遇到B指令会立即跳转到指定地址，然后继续从新的地址开始运行程序。高级语言（例如C语言）的goto语句常被翻译成B指令。

3.2.4 数据传送指令

数据传送指令用于CPU和存储器之间的数据传送，是ARM处理器唯一能与外部存储器交换数据的一类指令。

1. 单一数据传送指令
2. 多数据传送指令

3.4 ARM的结构

基于ARM的芯片有许多，功能结构也不同，但是最基本的是ARM核。无论学习哪种ARM类型的处理器，基本的内容都是一样的。本节介绍ARM体系结构。内容相对比较抽象，读者可以在后面的开发过程中结合本节知识深入体会。

3.4.1 ARM体系结构的命名方法

ARM体系结构的命名可以分成两部分，一部分是ARM体系版本的命名，另一部分是ARM体系版本的处理器命名。ARM体系到目前一共发布了9个系列的版本，每种版本都可以支持不同的指令集和特殊功能。

ARM	v（版本号）	x1（指令集）	x2（指令集）

指令集缩写	含义
T	Thumb指令集
M	长乘法指令集
E	增强DSP指令集
J	支持Java加速器
SIMD	多媒体功能扩展指令集

3.4.2 处理器系列划分

在确定了一种**ARM**体系结构后，可以形成一系列处理器。不过，处理器的命名主要是功能上的一些细小差别，基本的核心是相同的。

ARM	x y z	m

功能缩写	含义
T	支持Thumb指令集
D	支持片上调试
M	支持快速乘法器
I	支持嵌入式ICE调试
E	支持增强DSP指令
J	支持Java程序加速
F	支持浮点运算单元
-S	综合版本，支持所有功能

3.4.3 处理器工作模式

ARM微处理器可以在工作中随时切换状态。切换工作状态不会影响工作模式和寄存器的内容。但是ARM体系要求在处理器启动的时候应该处于ARM状态。ARM处理器使用操作寄存器的0位表示工作状态，取值是1时代表Thumb状态，取值是0时是ARM状态。可以使用BX指令可以切换状态。当处理器启动的时候操作寄存器取值为0，保证了默认进入ARM状态。

ARM处理器支持7种工作模式，这对一些通用处理器来说确实有点多。不过，通过分析可以发现，ARM的工作模式大多都是处理外部中断和异常的，只不过是对异常和中断的分类比较详细。

7种工作模式定义请参考。

3.4.3 处理器工作模式

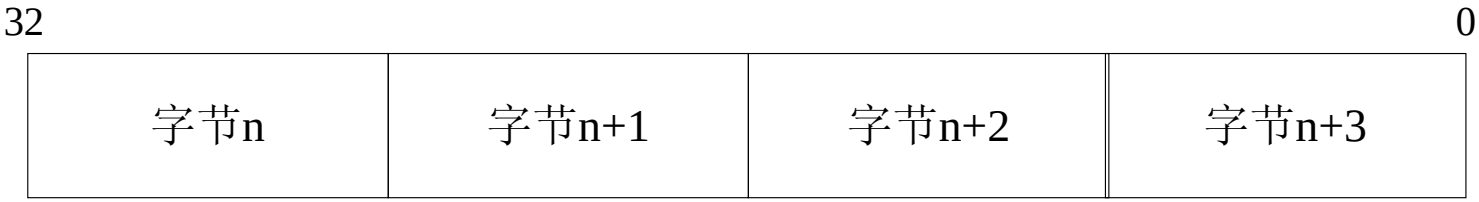
功能缩写	含义
T	支持Thumb指令集
D	支持片上调试
M	支持快速乘法器
I	支持嵌入式ICE调试
E	支持增强DSP指令
J	支持Java程序加速
F	支持浮点运算单元
-S	综合版本，支持所有功能

3.4.4 存储系统

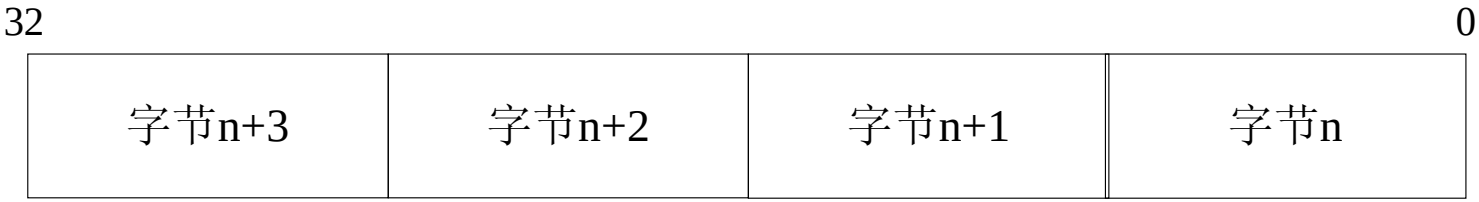
嵌入式微处理器大多采用一种线性的存储管理模式，ARM也是如此。这种管理模式的特点是，系统内所有的存储器和外部设备都被安排到一个统一的地址空间，通过地址映射到不同的设备，访问某个设备只需要访问该设备映射的内存地址即可。线性地址空间便于处理器的管理和用户操作。

在操作超过8位的数据时，存在两种不同的访问方法：大字端模式和小字端模式。

3.4.4 存储系统



大字端模式



小字端模式

3.4.5 寻址方式

寻址就是根据指令中的地址码找出操作数地址的过程，是计算机中很重要的一个部分。对编写程序来说，不同的寻址方式是存取速度和存取空间权衡的一个考虑因素。本书介绍7种常见的ARM处理器寻址模式。

1. 立即寻址
2. 寄存器寻址方式
3. 寄存器偏移寻址
4. 寄存器间接寻址
5. 基址寻址
6. 多寄存器寻址
7. 栈寻址

3.4.5 寻址方式

操作名称	功能
LSL（Logical Shift Left）逻辑左移	寄存器的二进制位从右往左移动，空出的位补0
LSR（Logical Shift Right）逻辑右移	寄存器的二进制位从左往右移动，空出的位补0
ASR（Arithmetic Shift Right）算术右移	移位过程中保持符号位不变，即如果源操作数为正数，则字的高端空出的位补0，否则补1
ROR（Rotate Right）循环右移	寄存器的低端移出的位填入字的高端空出的位
RRX（Rotate Right eXtended by 1 place）带扩展的循环右移	操作数右移一位，高端空出的位用原C标志值填充

3.5 ARM的功能选型

随着嵌入式应用的发展，ARM芯片的使用也不断增多。但是，由于ARM公司的技术授权，许多厂商都在生产基于ARM核的芯片，给用户的选择带来一定困难。本节从ARM芯片的结构和功能出发，介绍在ARM芯片的选型过程中需要注意的问题，并且在最后给出几种ARM芯片介绍。

3.5.1 ARM的选型原则

基于ARM核的处理器众多，功能相差也很大。选型主要从应用角度出发，根据功能的需求，是否有升级要求以及成本等多方面考虑。下面从技术角度介绍一下ARM选型考虑的因素。

1．ARM核心
2．时钟控制器
3．内部存储器
4．中断控制器
5．GPIO
6．实时钟RTC
7．串行控制器
8．WatchDog
9．电源管理功能
10．DMA控制器
11．I^2C接口
12．ADC和DAC控制器
13．LCD控制器
14．USB接口
15．I^2S接口

3.5.1 ARM的选型原则

芯片型号	供应商	FLASH容量（Bytes）	SRAM容量（Bytes）
AT91FR4081	ATMEL	1M	128K
SAA7750	Philips	384K	64K
HMS30C7202	Hynix	192K	无
LC67F500	Snayo	640K	32K

3.5.2 几种常见的ARM核处理器选型参考

介绍了ARM的功能选型以后，这里介绍在不同领域里几种ARM核的芯片。

1. Intel的IXP处理器
2. Philips的LPC处理器
3. 三星的S3C244X处理器

3.6 小结

本章介绍了ARM处理器相关的知识。包括微处理器和微控制器的概念和差异，介绍了ARM的体系结构特点和功能选型，最后给出了几个不同领域ARM核的芯片介绍。本章的内容偏重理论的较多，读者需要建立相关名词和术语的概念，在后面章节涉及到具体应用的时候会用到。随着实践的增多，读者会不断加深对这些概念的理解。第4章介绍嵌入式Linux基本知识。

第4章　嵌入式Linux

Linux是嵌入式领域应用最广泛的操作系统之一。本书的主题也是嵌入式Linux开发，在进入具体内容之前，有必要了解嵌入式Linux系统相关的知识。本章从介绍嵌入式系统开始引入Linux的概念，主要内容如下：

什么是嵌入式操作系统

常见的嵌入式操作系统对比

嵌入式Linux系统入门

常见的嵌入式Linux系统

4.1 常见的嵌入式操作系统

嵌入式操作系统通俗的说就是为嵌入式系统设计的操作系统，是运行在嵌入式硬件上的一类系统软件。嵌入式系统负责管理系统资源为用户提供调用接口，方便用户应用程序开发。一般来说，嵌入式操作系统是由启动程序（Bootloader）、核心（Kernel）、根文件系统（Root File System）组成的。通过特殊的烧录工具把编译好的嵌入式系统文件映像烧写到目标板的只读存储器（ROM）或者Flash存储器。

一个嵌入式系统的性能好坏很大程度上决定了整个嵌入式系统的性能。按照实时性能，嵌入式系统操作系统可以分成实时系统和分时系统。实时系统主要用在控制和通信领域，分时系统主要用在消费类电子产品。本节介绍几种常见的嵌入式操作系统。

4.1.1 VxWorks

VxWorks是美国**WindRiver**公司（国内也称作风河公司）开发的高性能实时嵌入式操作系统。其特点是使用了自己开发的**WIND**内核，有着很高的实时性能。

VxWorks系统内核是由进程管理、存储管理、设备管理、文件管理、网络协议等组成。内核占用很小的存储空间，最小的**WIND**内核可以配置到编译后仅有十几**KB**大小。精炼的内核保证了优异的实时性能。**VxWorks**系统被用在美国的火星探测器上，可见其稳定性和实时性确实很高。

4.1.2 Windows CE

WindowsCE是微软公司为嵌入式产品设计的一种嵌入式操作系统，主要针对需要多线程、多任务而且资源有限的设备。该系统采用模块化设计，开发人员可以定制不同的功能。WindowCE系统支持丰富的外部硬件设备，包括键盘、鼠标、触摸板、串口、网口、USB、音频设备等。并且该系统有与Windows一致的图形界面，可以很好的提高用户体验。

WindowsCE的一个最大特点就是支持上千个微软Win32编程接口（Microsoft Win32 API）。在Windows下开发过应用程序的程序员可以很快的上手WindowsCE。此外，WindowsCE还支持PC机上的模拟器，用户可以从模拟器上开发应用，调试完毕后再下载到目标板执行，提高了开发效率。

WindwosCE系统设计简单灵活，主要应用在各种小型设备，例如掌上电脑、餐厅点餐器等设备上。

4.1.3 PalmOS

Palm是3Com公司开发的一种掌上电脑产品。PalmOS是为该掌上电脑专门设计的一种32位嵌入式操作系统。PalmOS在设计的时候就充分的考虑到掌上电脑资源紧张的情况，适合内存较小的掌上电脑使用。除此之外，PalmOS提供了一个开发的操作系统接口，其他厂商和用户可以为其编写应用程序。目前运行在PalmOS系统上的应用程序已经超过1千种。

PalmOS最大限度的考虑了节能和硬件资源问题，提供了良好的电源管理功能和合理的内存管理功能。Palm设备的内存都是可读写的RAM，所以访问速度非常快。此外PalmOS还有很强的同步能力，可以与PC机同步数据。

4.1.4 Symbian

SymbianOS中文名称叫做塞班系统，是由诺基亚、索尼爱立信、摩托罗拉等几家移动通信设备制造商联合设计的嵌入式操作系统。塞班系统主要针对手机，设计目标是简单易用。目前在移动智能终端方面塞班的市场占有率非常大。

塞班系统有一个强大的核心，支持对象导向系统和Sun Java语言。该系统的应用程序主要使用C++和Java开发，从应用开发角度可以缩短开发周期。塞班系统本身支持多种外部设备，而且为厂商和用户留有丰富的接口并且提供了开发工具。用户可以很快的在塞班系统开发应用程序。目前塞班系统已经广泛的应用在诺基亚手机上。

4.2 嵌入式Linux操作系统

4.1节介绍的几种嵌入式操作系统都是商业系统。虽然有良好的性能和开发工具支持，但是对于学习嵌入式开发的人来说，无论从成本和学习难度方面都是不小的挑战。本节介绍著名的Linux操作系统以及嵌入式领域的应用。

4.2.1 什么是Linux

许多读者可能都听说过Linux操作系统。Linux系统是一个免费使用的类似UNIX操作系统，最初运行在x86体系结构，目前已经被移植到数十种处理器上。Linux最初由芬兰的一位计算机爱好者Linus Torvalds设计开发，经过十余年的发展，现在该系统已经是一个非常庞大、功能完善的操作系统。Linux系统的开发和维护是由分布在全球各地的数百名程序员完成的，这得益于它的源代码开放特性。

与商业系统相比，Linux系统在功能上一点都不差，甚至在许多方面要超过一些著名的商业操作系统。Linux不仅支持丰富的硬件设备、文件系统，更主要的是它提供了完整的源代码和开发工具。对于嵌入式开发来说，使用Linux系统可以帮助用户从底层了解嵌入式开发的全过程，以及一个操作系统内部是如何运作的。学习Linux系统开发对初学者有很大的帮助。

4.2.2 Linux与UNIX的不同——GPL版权协议介绍

UNIX是一种商业系统的名称也是注册商标，有着严格的商业版权。Linux系统在界面功能方面与UNIX很相似，但是在版权方面有很大不同。Linux使用了GNU的GPL版权协议，实际上，Linux系统的发展很大程度上也依赖了GPL版权协议。GNU是美国自由软件基金会创建的一个非盈利组织，GNU致力于设计和推广自由软件，它的所有软件都是基于GPL版权协议的。

GPL版权协议仅是多种软件协议中的一种，实际上，开发源代码的版权协议还有许多。与传统的商业软件不开放源代码相比，采用GPL版权协议的开放源代码（简称开源）软件对于用户的影响很大。用户可以自由加入到某个软件的开发中，不断的升级和开发新的软件和功能，极大的促进了软件行业的发展。同时，普通用户也可以读到一些顶尖高手编写的程序，从中学习知识，这也是GPL版权协议的一个初衷。

4.2.3 Linux发行版

Linux系统是开放的，任何人都可以制作自己的系统，因此出现了许多厂商和个人都在发行自己的Linux系统。据统计，目前Linux的发行版已经超过300种，而且还在不断增加。如此多的发行版，对于任何一个人都是不可能完全学习的，本节介绍几种国内常见的Linux发行版供读者参考。

1. RedHat
2. Debian
3. Ubuntu

4.2.4 常见的嵌入式Linux系统

在4.2.3节讲到的都是安装在PC机上的Linux系统，本节介绍几种嵌入式领域用到的Linux发行版。通常这些发行版被统称为“嵌入式Linux系统”。

1. RT-Linux嵌入式系统
2. μClinux嵌入式系统

4.3 小结

本章讲解了嵌入式Linux系统的入门知识，以及一些常见的嵌入式操作系统。嵌入式Linux系统是新兴的一门技术，还在不断的发展中。目前的嵌入式Linux系统种类繁多，但是万变不离其宗。读者在了解这些系统的同时，还是需要从掌握基本的Linux系统开始入手。第5章介绍如何搭建嵌入式Linux开发环境。

第5章　建立软件开发环境

工欲善其事，必先利其器。在进行嵌入式软件开发工作之前，必须建立一个开发环境。开发环境包括了操作系统、编译器、调试器、集成开发环境、各种辅助工具等。嵌入式Linux开发需要在主机上开发目标系统的程序，建立主机开发环境可以在Linux系统下，也可以是Windows系统。在两种系统建立开发环境各有利弊，本章讲解Linux系统和Windows系统如何搭建嵌入式开发环境，主要内容如下：

安装独立的Linux发行版

搭建Linux发行版下的开发环境

在Windows系统安装Linux系统模拟环境

Linux系统常见命令和工具

Windows系统常用工具

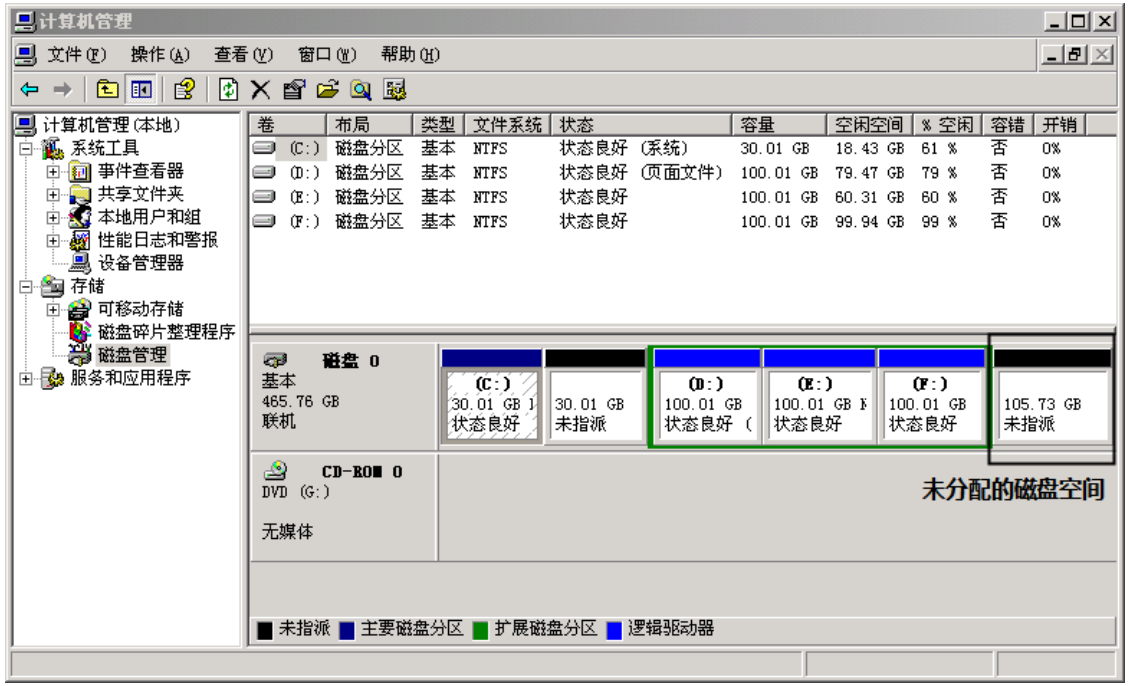
ADS集成开发环境

5.1 独立的Linux系统

本节介绍的Linux系统需要单独安装在PC机的磁盘分区，读者在操作之前需要做好准备，备份好数据。目前的Linux系统发行版有许多，Ubuntu Linux界面贴近用户，软件安装配置简单，适合初学者学习使用。本书使用Ubuntu Linux 8.04发行版本作为嵌入式Linux开发的主机环境。

5.1.1 安装Ubuntu Linux

在安装之前，需要确认有足够的磁盘空间供安装Ubuntu Linux。在笔者的机器上使用Windows的磁盘管理程序查看硬盘分区。



5.1.2 关键一步－系统的网络设置

Ubuntu Linux使用了apt管理软件包。apt是一种给予Debian系统deb包的管理器，最大的特点就是从网络安装软件包，并且能自动获取每个软件包的依赖关系，安装正确的软件包。由此可见，网络对于Ubuntu Linux系统的重要性。

在安装其他的软件包之前，首先是设置Ubuntu的网络。Ubuntu的网络配置文件主要有/etc/network/interfaces和/etc/resov.conf两个文件。其中，/etc/network/interfaces文件是配置网络接口卡（网卡）和网关地址，内容如下：

```
iface lo inet loopback    # 环回设备loopback配置
iface eth0 inet static        # 第一个以太网卡设备eth0，使用静态配置
address 192.168.0.120     # 配置IP地址
netmask 255.255.255.0     # 配置掩码
gateway 192.168.0.1       # 设置网关
```

5.1.3 其他必要的设置

设置好网络之后，最关键的就是设置apt的源。apt使用/etc/apt/sources.list文件记录软件包服务器的配置信息，这些软件包服务器地址被称作源。在使用apt命令安装软件包的时候，会从源服务器下载并安装软件包。

Ubuntu自己的源服务器在美国，在其他国家都有镜像服务器，读者可以选择一个离自己较近的服务器，可以提高下载文件速度。在国内目前有两个比较好的源服务器：一个是由163提供的，一个是由cn99.com提供的。

5.1.4 怎样安装卸载软件

Ubuntu安装卸载软件都非常方便，使用apt-get命令可以完成软件的管理。具体格式如下：

apt-get install <软件包名称>

apt-get uninstall <软件包名称>

5.1.5 怎样配置系统服务

在Ubuntu下配置系统服务非常简单，需要一个名为sysv-rc-conf的软件包。使用“sudo apt-get install sysv-rc-conf”命令安装软件包，安装完毕后在Shell终端输入“sudo sysv-rc-conf”，出现一个文本界面。其中最左边是系统的服务名称，右边依次是系统运行级别1～6。每个系统服务在对应的系统级别下都可以选择‘X’表示在该级别下启动，去掉‘X’表示不启动。

用户根据需要选择以后，键入字母‘Q’保存退出。

5.1.6 安装主要的开发工具

Ubuntu Linux把主要的开发工具打包放在一起，安装的时候直接安装一个软件包就可以把基本的开发工具和程序都装到系统。

（1）安装基本的开发工具。

（2）检查开发工具是否安装成功。

5.1.7 安装其他的开发工具和文档

主要开发工具安装完毕后，仅能保证编译和调试程序。对于大部分开源软件来说，还需要**autoconf**、**automake**等工具。其他工具的安装命令如下：

sudo apt-get install autoconf automake1.9

sudo apt-get install flex bison

sudo apt-get install manpages-dev

sudo apt-get install binutils-doc cpp-doc gcc-doc glibc-doc stl-manual

5.2 运行在Windows上的Linux系统

对于多数没有使用过Linux系统的读者来说，初次使用Linux开发会遇到许多问题。初学者可以通过首先在Windows系统下使用类似Linux的模拟环境熟悉一下。此外，在Linux模拟环境下可以完成大多数的Linux系统操作。Windows下的Linux模拟环境有许多，其中应用最广泛的是Cygwin系统。

5.2.1 什么是Cygwin

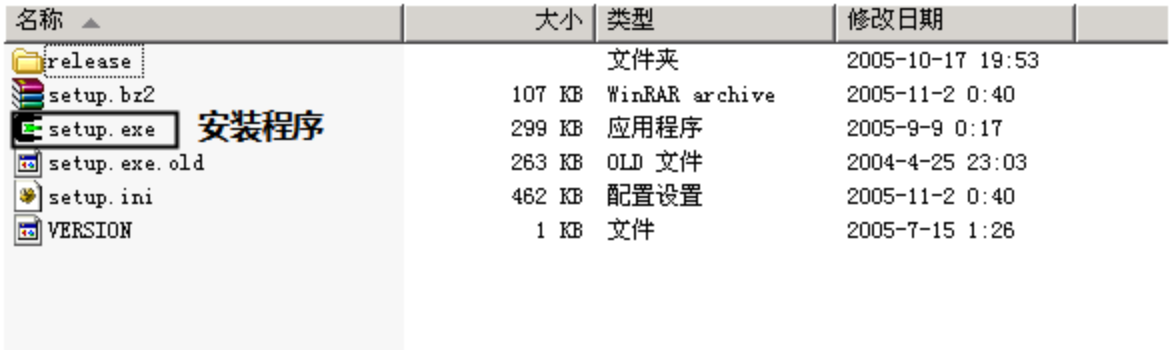
Cygwin是Cygnus公司开发的运行在Windows平台的Linux系统模拟环境，该软件是自由软件。Cygwin对学习Linux使用，以及Windows和Linux系统之间应用程序的移植都有很大帮助。在嵌入式开发领域，由于Cygwin的优点被越来越多开发人员使用。

Cygwin的设计思想十分巧妙。与其他工具不同的是，Cygwin没有逐个把Linux下的工具移植到Windows系统，而是在Windows系统上设计了一个Linux系统调用中间层。Linux系统调用中间层的作用是在Windows系统模拟Linux的系统调用，之后只需要把Linux下的工具在Windows系统重新编译，做一些较小的修改即可移植到Windows系统。

Cygwin几乎移植了Linux系统常用的所有开发工具到Windows系统，使用户感觉就好像在Linux系统下工作，为用户在Windows下开发Linux程序提供了保障。

5.2.2 如何安装Cygwin

Cygwin的安装比较简单。Cygwin支持网络在线安装和从本地安装两种模式，由于Cygwin的服务器在国外，建议国内用户下载Cygwin的本地安装包从本地安装。首先从http://www.cygwin.net下载软件安装包，本书使用的是2.5.10.22版本，其他版本的安装过程相同。



5.2.3 安装开发环境

Cygwin在安装包中自带了绝大多数的Linux软件和工具在Windows系统的移植版本。默认的软件包选项自带了基本的开发工具，安装好无需配置可以使用GNU的开发环境。为了验证是开发环境是否安装成功，查看各开发工具的版本：

```
$ gcc --version
gcc (GCC) 3.4.4 (cygming special) (gdc 0.12, using dmd 0.125)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.

$ gdb --version
GNU gdb 6.3.50_2004-12-28-cvs (cygwin-special)
Copyright 2004 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-cygwin".
```

5.3 Linux常用工具

目前大多数的Linux发行版都提供了图形界面作为默认界面，但是，命令行工具在Linux仍然很重要。Linux工具的特点是一个程序包含的功能尽量专一，不同的程序通过文件、管道等进程间数据共享的方法可以组合使用，达到处理复杂功能的目的。学习使用Linux系统，命令行工具是基础。GNU的命令行工具都有相同的特点，初学者从一些基本的工具入手，比较容易学习。

5.3.1 Linux Shell和常用命令

使用过DOS系统和Windows中断控制台的人对命令行界面都有一定体会。与这些系统不同，Linux的命令行是通过一种叫做Shell的程序提供的。Shell程序负责接受用户的输入，解析用户输入的命令和参数，调用相应的程序，并给出结果和出错提示。Linux支持多种Shell程序，早期的Shell程序功能比较单一，现在主流的Linux发行版使用bash作为默认的Shell。Bash支持功能强大的脚本、命令行历史记录、终端彩色输出等功能。Shell是Linux的外壳，用户通过Shell使用系统提供的功能。

在Linux系统中，仅有内核还是不够的，需要应用程序支持才能发挥内核提供的功能。无论是Linux发行版还是嵌入式Linux开发板上的系统，都提供了常见的一些命令。

5.3.1 Linux Shell和常用命令

命令	作用	常用参数	参数作用
ls	列出指定目录的列表，包括文件和子目录。默认是当前目录。	-l	以列表方式查看
		-a	显示隐含文件和目录
		-h	以便于人阅读的方式查看文件的大小
ln	建立连接	-s	软连接
		-f	连接是一个目录
df	查看磁盘空间	-h	以便于人阅读的方式查看文件的大小
du	查看指定目录占用的空间。默认是当前目录	-h	以便于人阅读的方式查看文件的大小
pwd	修改用户密码。非root用户只能修改自己的密码。		
chmod	修改文件或目录的读写权限。	-R	递归调用
chgrp	修改文件或目录的用户组。	-R	递归调用
chown	修改文件或目录的所有者。	-R	递归调用
date	查看日期		
cat	输出文件内容到屏幕。		
echo	回显一个字符串或者环境变量到屏幕。		
uname	查看机器名称。		
ps	查看进程状态	-e	查看系统所有进程
kill	杀死指定进程	-9	强制杀死进程

5.3.1 Linux Shell和常用命令

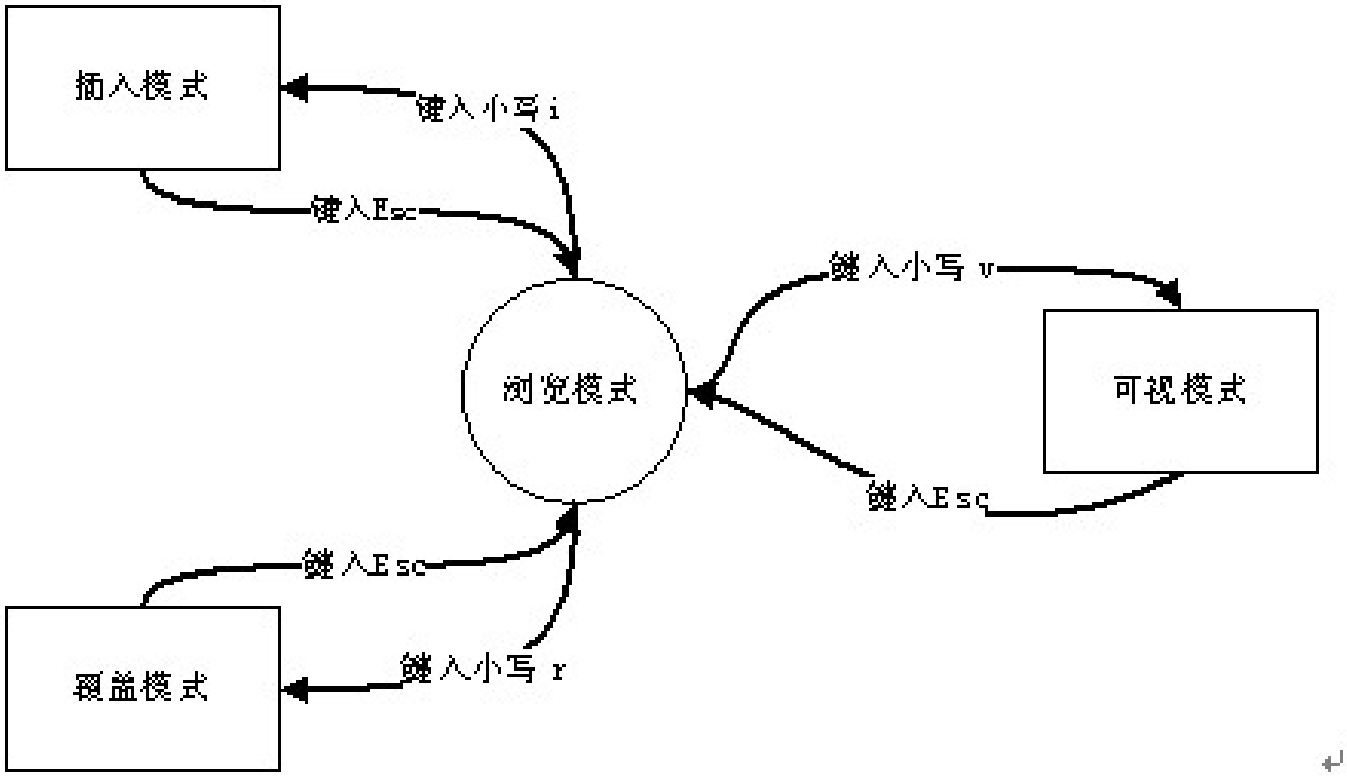
命令	作用
ifconfig	查看和配置网卡
fdisk	磁盘分区工具
mkfs	磁盘格式化
insmod	加载内核模块
lsmod	内核模块列表
modprobe	内核模块管理工具
reboot	重启机器
halt	停机

5.3.2 文本编辑工具vi

Linux系统的文本编辑工具有许多，其中使用最广泛的就是vi编辑器了。vi编辑器的功能十分强大，并且非常小，适合安装在嵌入式系统使用。vi虽然功能强大，但是对于初学者来说，上手比较困难，初学者往往被vi奇怪的操作弄的失去学习的信息。本书有关vi的使用仅涉及基本操作，目的是帮助初学者学习vi的基本操作。更高级的vi操作读者可以参考vi的帮助文档。

vi编辑器支持编辑模式、浏览模式、插入模式和可视模式四种模式。其中，插入模式包括了插入文本和替换文本两种模式。当启动vi的时候，默认进入浏览模式。浏览模式只能查看和删除文档内容，但是不能修改；编辑模式用户可以修改文档内容，与普通的文本编辑器相同；覆盖模式下用户输入的内容会覆盖光标所在位置的文本；可视模式提供了一种选择文本的方法，可以使用键盘完成鼠标选择文本的功能。

5.3.2 文本编辑工具vi



5.3.3 搜索工具find和grep

find和grep是Linux系统最常用的两个搜索工具。这两个工具不同的是find用于查找文件，grep用于查找文件内容。

grep支持正则表达式（一种描述字符串特征的语法），通过在一个或多个文件中搜索字符串，符合的内容被送到屏幕显示。grep工具不会修改文件内容。grep通过返回值表示搜索状态，如果搜索成功返回0，如果失败返回1，如果搜索的文件不存在返回2。因此，grep可以用于shell脚本。

5.3.4 FTP工具

FTP是标准的互联网文件传输协议，被广泛的应用于网络文件传输，是不同机器间文件传输简单有效的方法。FTP协议允许传输二进制和文本文件。在许多系统上都提供了FTP客户端软件，用来从FTP服务器下载或者上传文件。本节介绍的FTP客户端工具可以在Linux系统和Windows系统上使用，是一种简单易用的文件传输手段。

5.3.4 FTP工具

命令名称	含义
dir	列出服务器的目录
cd	改变服务器上的目录
lcd	改变本地目录
ascii	使用文本方式传输文件
binary	使用二进制方式传输文件
bye	退出ftp工具
hash	显示文件传输进度
get	从服务器下载文件
put	上传文件到服务器
!	切换到shell界面，在shell中使用exit命令可以退回ftp界面

5.3.5 串口工具minicom

串口是嵌入式开发使用最多的通信方式。Linux系统提供了一个串口工具minicom，可以完成复杂的串口通信工作。本节介绍minicom的使用。首先是安装mimicom，在Ubuntu Linux系统shell下输入“$ sudo apt-get install minicom”回车后即可安装minicom软件。软件安装好后，第一次使用之前需要配置minicom。

5.5 Windows常用工具

嵌入式开发的开发环境和运行环境往往不是同一台机器。作为开发环境，**Windows**下通常运行一些客户端和代码管理工具、文档管理工具等。本节介绍**Windows**下常用的两个工具。

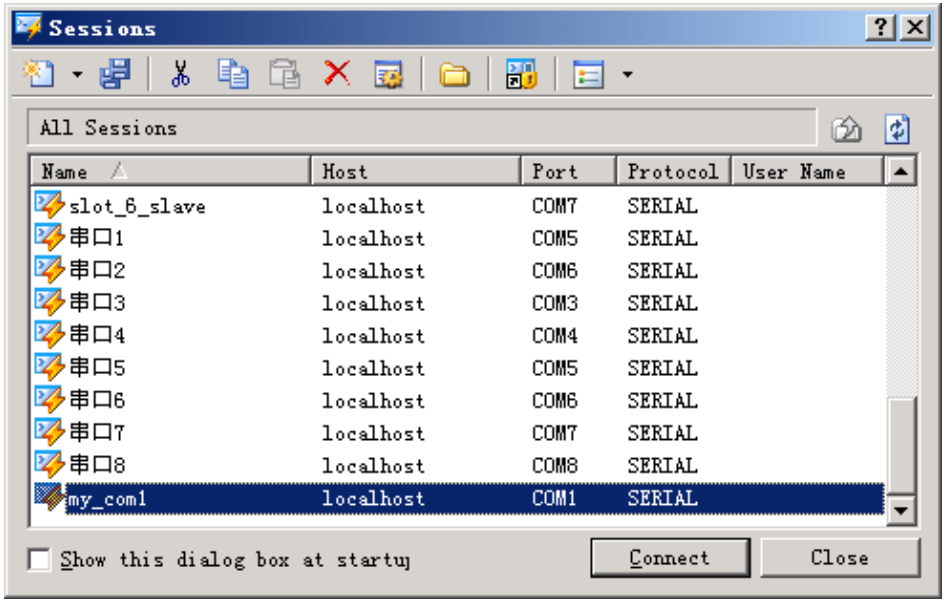
5.5.1 代码编辑管理工具SourceInsight

SourceInsight是一个功能强大的代码管理工具。该工具可以轻松管理代码庞大的工程，提供了丰富的编辑功能，支持函数、变量的类型定义查看、跳转等。**SourceInsight**对C语言代码支持最好，本节介绍**SourceInsight**安装和使用。

1. 设置**SourceInsight**工程
2. **SourceInsight**特色功能

5.5.2 串口工具XShell

在5.3.5节介绍了Linux下的串口工具minicom，本节介绍一个Windows下比较好用的串口工具XShell。实际上，XShell不仅支持串口连接，还可以连接Telnet服务器、SSH服务器等。



5.6 ARM集成开发环境ADS

集成开发环境英文**Integrated development environment**，缩写是**IDE**。在没有**IDE**之前，开发软件过程中的编辑，编译，调试需要不同的工具操作，不仅效率低而且容易出错。**IDE**的作用是把编辑、编译和调试等工具集成在一起，并且向用户提供一个图形界面的开发环境。**ARM**开发有标准的开发环境**ARM Development Suit**，简称**ADS**。

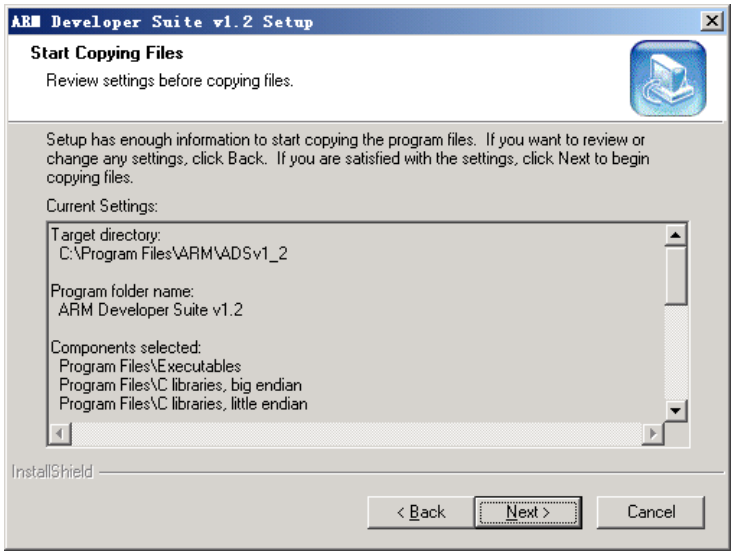
5.6.1 ADS集成开发环境介绍

ADS是ARM公司推出的ARM集成开发工具，目前最新版本是1.2，只能在Windows系统安装。ADS包括程序库、命令行开发工具、图形界面、调试工具和代码编辑器等。本节介绍一下ADS自带的命令行工具。

1. **C语言编译器armcc**
2. **C++语言编译器armcpp**

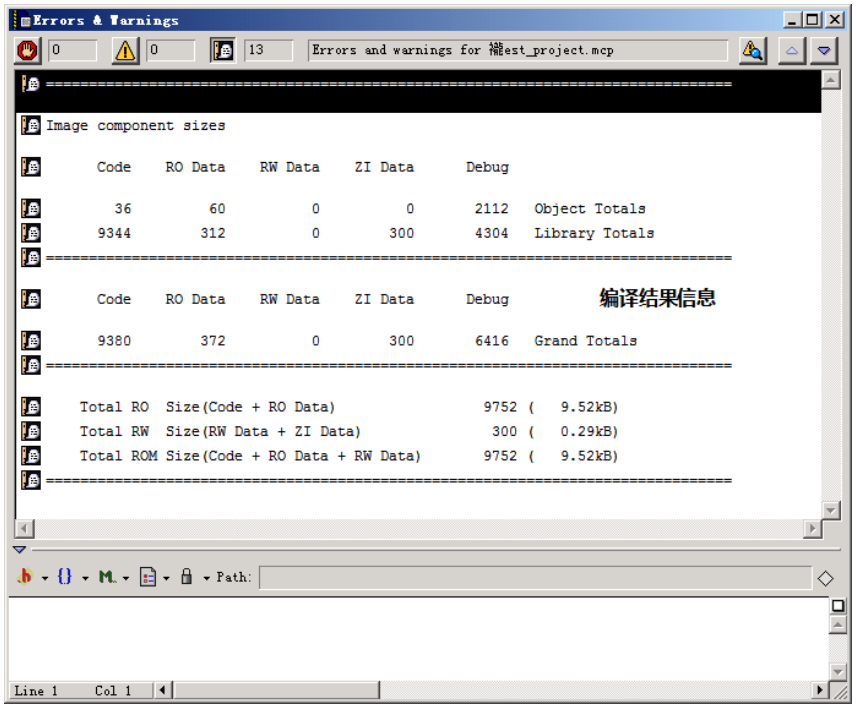
5.6.2 配置ADS调试环境

在使用ADS之前，需要安装ADS开发环境。ADS是一个商业软件，需要支付版权费用才可以使用，对于没有购买版权的用户会受到功能限制。



5.6.3 建立自己的工程

本节介绍如何在ADS环境下建立自己的工程并且编译生成目标文件。



5.7 小结

本章讲解了嵌入式Linux开发环境，包括系统环境、开发工具、辅助工具等。开发工具是嵌入式开发过不可缺少的，每种工具都有自己的用途和范围，读者应该多实践，掌握常见的开发工具使用方法。第6章开始介绍基本的程序开发知识。

第6章　第一个Linux应用程序

学习嵌入式程序开发首先从最简单的程序开始。一个最基本的Linux应用程序可以涵盖编程的所有基本知识，通过编写Linux可以快速入门程序开发。本章目的是通过实际的程序向读者介绍Linux程序的基本框架和工作流程。主要内容如下：

编写一个最基本的应用程序

分析程序的执行过程

程序生成过程

程序编译过程管理

6.1 向世界问好 – Hello,World!

很多的编程书籍都以输出一行“**Hello,World!**”向初学者展示如何编写程序。这个程序很简单，却展示了C程序的基本要素：语法格式、引用头文件、调用库函数等。本节展示了程序的编辑、编译和执行相关知识。

6.1.1 用vi编辑源代码文件

在5.3.2小节介绍了vi编辑器的用法，现在使用vi编辑器编写第一个源代码文件。具体操作过程如下：

1. 创建源代码文件hello_test.c
2. 编写源代码
3. 保存退出

6.1.2 用gcc编译程序

编辑好源文件**hello_test.c**文件后，需要把它编译成可执行文件才可以在**Linux**下运行。在控制台模式当前目录，输入以下命令完成编译：

gcc hello_test.c

6.1.3 执行程序

到目前为止，第一个程序已经编译好了，下面就是执行程序了。在大多数的Linux系统上，都是通过一个名为PATH的环境变量来管理系统可执行程序的路径的，但是不幸的是这个变量里并没有包含了当前路径的‘./’，所以需要按照下面的方式执行程序：

```
./a.out
```

执行a.out程序后，输出结果如下：

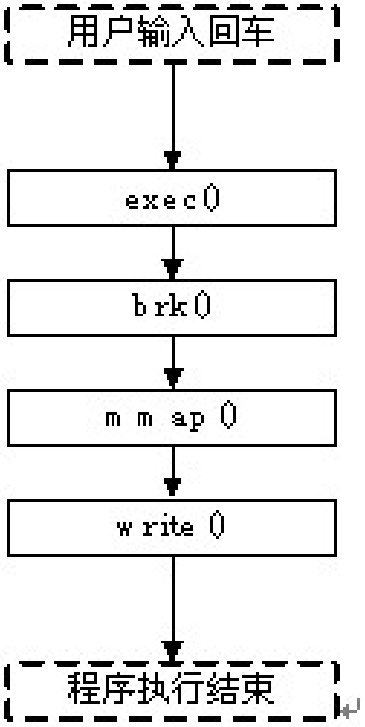
```
[root@slash dev_test]# ./a.out
Hello,World!
```

6.2 程序背后做了什么

前面讲了程序如何编辑和编译，并且展示了程序的输出结果。可能有人会问程序是如何输出在屏幕上的。带着这个问题，这一节从程序加载和执行的过程，分析一下Linux应用程序是如何在计算机上运行的。实例6-1所示的基本程序涵盖了执行一个Linux应用程序所有的细节。

6.2.1 程序执行的过程

一个Linux程序的加载和执行过程。



6.2.2 窥视程序执行中的秘密

上面的程序执行过程比较难理解，这里推荐使用strace工具分析这个程序的执行流程。在执行a.out的那个目录，输入以下命令：

```
[dev_user@localhost dev_user]$ strace ./a.out
```

6.2.3 动态库的作用

Linux系统有两种程序库，一种是被称作静态库（static library），在程序连接的时候，把库函数的目标代码和程序连接在一起；还有一种就是前面提到的动态库（shared library），从英文字面翻译，动态库可以翻译为共享库。

动态库是Linux系统最广泛的一种程序使用方式，它的工作原理是相同功能的代码可以被多个程序共同使用。在程序加载的时候，内核会检查程序使用到的动态库是否已经加载到内存，如果没有加载到内存，则从系统库路径搜索并且加载相关的动态库；如果动态库已经被加载到内存，程序可以直接使用而才 爰釉亍

从动态库的工作原理可以看出，任何一个动态库仅会被系统加载一次。使用程序动态库还有一个好处，就是可以减小应用程序占用的空间和加载时间。

6.3 程序如何来的 - 编译的全部过程

在6.1.2节，通过命令行输入gcc hello_test.c就可以编译出一个可执行文件a.out。在使用gcc编译c语言源代码文件的时候，gcc隐含进行了两个过程：编译和连接。所以确切的说，应该是编译连接c语言源代码文件，本节就讲一下这个过程。

6.3.1 编译源代码

编译的目的是把人书写的高级语言代码翻译成目标程序的语言处理程序，编译用的程序（例如gcc）也可以称为编译系统。

一个编译系统把一个源程序翻译成目标程序的工作过程分为五个阶段：词法分析、语法分析、中间代码生成、代码优化和目标代码生成。其中主要阶段是词法分析和语法分析，也可以称为源代码分析，分析过程中发现有语法错误，给出提示信息。

1. 词法分析
2. 语法分析
3. 中间代码生成
4. 代码优化
5. 目标代码生成

6.3.2 链接目标文件到指定的库

源代码经过编译以后，需要链接才可以在Linux系统运行，链接的作用是把代码中调用的系统函数和对应的系统库建立关系，设置程序启动时候的内存，环境变量等，以及程序退出的状态，释放占用的资源等操作，这些背后的工作对用户都是隐含的。gcc在连接用户目标文件的时候会根据用户代码使用不同的函数连接对应的动态或者静态库（根据连接选项，默认是动态库），同时，还会对所有的目标文件连接固定的预编译好的系统目标文件，这几个预编译好的目标文件用来完成程序初始化，结束时的环境设置等。

6.4 更简单的办法——用Makefile管理工程

6.1.2节讲述了可以使用如下的方法编译一个链接动态库的程序：

gcc hello_test.c

以及使用如下的方法编译一个静态程序：

gcc –static hello_test.c

6.4.1 什么是Makefile

Makefile是一个文本文件，是GNU make程序在执行的时候默认读取的配置文件。Makefile有强大的功能，它记录了文件之间的依赖关系，通过比对目标文件和依赖文件的时间戳，决定是否需要执行相应的命令；同时，Makefile还可以定义变量，接收用户传递的参数变量，通过这些元素的相互配合，省去了繁杂的编译命令，不仅节省时间，也减小了出错的概率。

6.4.2 它是如何工作的

Makfile的工作原理是通过比对目标文件和依赖文件的时间戳，执行对应的命令。

Makfile的语法结构如下：

（目标文件）：（依赖文件1）（依赖文件2）（依赖文件…）

（命令1）

（命令2）

（命令…）

6.4.3 如何使用Makefile

仍然以编译hello_test.c文件为例，下面的步骤使用Makefile编译和管理hello_test.c。

（1）创建Makefile文件。在hello_test.c所在的目录输入vi Makefile。

（2）输入Makefile的内容。在vi插入模式，输入下面的内容。

```
hello_test : hello_test.c
gcc –o hello_test hello_test.c
clean :
rm -fr hello_test *.o *.core
```

（3）使用make管理程序。

6.4.4 好的源代码管理习惯

在一个软件项目中，往往会将不同功能的代码放在不同的文件中，这时候，一个好的代码管理方法就显得很重要，凌乱的代码分布不仅对调试带来很多麻烦，对以后的升级和维护都是一个不小的挑战。这里给出几个代码管理的建议：

1．把不同功能的代码放在不同的文件，并且把必要的函数放在对应的头文件

2．对软件模块划分层次

6.5 小结

本章从一个简单的应用程序入手，介绍了开发一个Linux应用程序的流程。通过实例分析了编译和连接的原理，并且剖析了程序执行的过程。最后讲述了Makefile管理工程文件如何管理软件代码。请读者多实践，只有不断的实践才能对这部分的知识深入理解。第7章讲解Linux应用程序开发的基础知识。

第7章　Linux应用程序编程基础

Linux系统的应用程序是为了完成某项或者某些特定任务的计算机程序，应用程序和文档组成了软件。应用程序都是在操作系统基础上运行的，Linux应用程序运行在用户模式，可以通过Shell或者图形界面与用户交互。应用程序运行在独立的进程，拥有自己独立的地址空间，通俗的说，在一个应用程序看来，他自己拥有计算机的资源，并不知道其他应用程序的存在。本章讲解Linux应用程序开发的重要概念，主要内容如下：

C内存管理

ANSI C文件读写操作

POSIX文件读写操作

7.1 内存管理和使用

内存管理是计算机编程的一个重要部分，也是许多程序员头疼的一个部分。在目前的嵌入式系统中，资源仍然是有限的。在程序设计的时候，内存管理十分重要。C程序的内存管理灵活，接口简单，这也是初学者容易出错的根本，读者在学习本节内容的时候应注重多实践。本节首先讲解Linux程序的基本结构，之后介绍C程序的内存管理函数，最后给出了C程序内存管理的实例。

7.1.1 堆和栈的区别

1．栈

栈（stack）是一个由编译器分配释放的区域，用来存放函数的参数，局部变量等。操作方式类似于数据结构教材里的栈。当调用函数时，被调用函数的参数和返回值被存储到当前程序的栈区，之后被调用函数在为自身的自动变量和临时变量在栈区上分配空间。当函数调用返回时，在栈区内的参数返回值，自动变量和临时变量等会被释放。这就是为什么C语言函数参数如果不是指针的话，被修改的参数结果用户无法得到的原因。

函数的调用和栈的使用方式保证了不同函数内部定义相同名字的变量不会混淆。栈的管理方式是FILO（First In Last Out），称作先进后出，学过数据结构的读者知道，栈内的数据是在一个方向管理的，先到达的数据最后被读出来，生活中就有这样的例子，比如火车的栈，就是车头先进入，但是出来的时候却是车尾先出来，车头最后出。

7.1.1 堆和栈的区别

2．堆

堆（heap）一般位于bss段和栈之间，用来动态分配内存。这段区域由程序员管理，程序员利用操作系统提供的分配和释放函数使用堆区的内存。如果程序员在堆上分配了一段内存，却没有释放，在目前主流的操作系统上，退出时会被操作系统释放。但是这并不是一个好的习惯，因为堆区的空间不是无限的，过多姆峙浠岬贾露涯诖嬉绯觯　绦蛞斐Ⅰ踔帘览！

堆的管理与栈不同，操作系统在堆空间维护一个链表（请参考数据结构相关定义），每次程序员从堆分配内存的时候操作系统会从堆区扫描未用空间，当一个空间的大小符合申请空间的时候，就把此空间返回给程序员，同时会把申请的空间加入链表；当程序员释放一个空间的时候，操作系统会从堆的链表中删除指定的节点，并且把释放的空间放回未用空间。

7.1.2 内存管理函数malloc()和free()

C程序有两个主要的内存管理函数：malloc()函数负责分配内存；free()函数释放malloc()分配的内存。这两个函数都是C标准库stdlib.h头文件定义的，Linux系统函数原型：

```
void *malloc(size_t size);
void free(void *ptr);
```

7.1.3 实用的内存分配函数calloc()和realloc()

在C程序开发项目中，还有两个实用的内存分配函数：calloc()用来分配一块新内存；realloc()用来改变一块已经分配的内存的大小。这两个函数都是在C标准库的stdlib.h头文件定义的。

1. calloc()函数
2. realloc()函数

7.1.4 内存管理编程实例

本节给出一个内存管理编程实例。该实例代码展示了**malloc()**函数，**calloc()**函数分配内存空间写入字符串，之后使用**realloc()**函数重新分配内存空间，最后释放动态分配的内存。程序在分配的内存空间内写入字符串，通过打印字符串到屏幕展示内存分配的结果。

7.2 ANSI C文件管理

本节重点讲解ANSI C文件库。包括文件指针的概念；文件和流之间的关系；文本和二进制文件；和文件的基本操作。ANSI的C标准文件库封装了文件的系统调用，为了提高效率还加入了文件缓冲机制，提供记录的方式读写文件，并且具有良好的可移植性和健壮性，是Linux C语言最基本的文件编程。

7.2.1 文件指针和流

文件是可以永久存储的，有特定顺序的一个有序，有名称的字节组成的集合。在Linux系统中，通常能见到的目录、设备文件和管道等，都属于文件，具有不同的特性。本节描述的ANSI文件只能用于普通文件操作。

ANSI文件操作提供了一个重要的结构-文件指针FIL。文件的打开、读写和关闭以及其他访问都要通过文件指针完成。FILE结构通常作为FILE*的方式使用，因此被称作文件指针，这个结构在stdio.h头文件的定义：

```
typedef struct {
int level;
unsigned flags;
char fd;
unsigned char hold;
int bsize;
unsigned char _FAR *buffer;
unsigned char _FAR *curp;
unsigned istemp;
short token;
} FILE;
```

7.2.2 存储方式

ANSI C规定了两种文件的存储方式：文本方式和二进制方式。文本文件也称作ASCII文件，每个自己存储一个ASCII码字符，文本文件存储量大，便于对字符操作，但是操作速度慢；二进制文件降数据按照内存中的存储形式存放，二进制文件的存储量小，存取速度快，适合存放中间结果。

在Linux系统上，文件的存放都是按照二进制方式存储的，用户在打开的时候，根据用户指定的打开方式进行存取。

7.2.3 标准输入，标准输入和标准错误

Linux系统为每个进程定义了标准输入、标准输出和标准错误三个文件流，也称作I/O数据流。系统预定义的三个文件流有固定的名称，因此无需创建可以直接使用。stdin是标准输入，默认是从键盘读取数据；stdout是标准输出，默认向屏幕输出数据；stderr是标准错误，默认是向屏幕输出数据。

三个I/O数据流定义在stdio.h头文件里，程序在使用前需要引用相关头文件。C标准库函数printf()默认使用stdout输出数据，用户也可以通过重新设置标准I/O，把程序的输入输出结果定向到其他设备。

7.2.4 缓冲

标准文件I/O库提供了缓冲机制，目的是为了减少外部设备的读写次数。同时，使用缓冲也能提高应用程序的读写性能。标准文件I/O提供了三种类型的缓冲：

全缓冲。

行缓冲。

不带缓冲。

7.2.5 打开关闭文件

ANSI C文件库定义了打开文件函数fopen()和关闭文件函数fclose()，定义如下：

FILE *fopen(const char *path, const char *mode);

int fclose(FILE * stream);

mode参数	说明
r或rb	为读打开文件
w或wb	为写打开文件，并把文件长度置为0（清空文件）
a或ab	在文件结尾添加打开
r+或r+b或rb+	为读和写打开
w+或w+b或wb+	为写打开文件，并把文件长度置0（清空文件）
a+或a+b或ab+	在文件结尾读写打开

7.2.6 读写文件

一但成功打开一个文件后，就可以进行文件操作了，ANSI C文件库提供了三种不同类型的文件读写函数：

每次一个字符的I/O。

每次一行的I/O。

成块数据的I/O。

1. 每次一个字符的文件读写函数
2. 每次一行的文件读写函数
3. 成块数据的文件读写函数

7.2.7 文件流定位

在读写文件的时候每个文件流都会维护一个文件流指针，表示当前文件流的读写位置，在打开文件的时候文件流指针位于文件的最开头（使用‘a’方式打开的文件，文件流指针位于文件最后），当读写文件流的时候，读写文件流的函数会不断改变文件流当前位置。当用户在写入一些数据后，如果需要读取之前写入的数据，或者需要修改指定文件位置的数据，就需要用到文件流定位功能。为此，ANSI文件I/O库提供了文件流定位函数，定义如下：

```
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
void rewind(FILE *stream);
```

7.2.8 ANSI C文件编程实例

在本节最后，给出一个文件编程实例，打开一个文件，向文件写入三个字符串，然后重新定位文件流读写指针到文件起始位置，从文件读取刚写入的三个字符串到另一个缓冲，并且打印读出来的字符串。

7.3 POSIX文件I/O编程

POSIX是可移植操作系统接口的简写，英文全程是Portable Operating System Interface。最初由IEEE（Institute of Electrical and Electronics Engineers，IEEE）开发，目的是为了提高UNIX环境下的应用程序可移植性。实际上POSIX并不局限于UNIX，只要符合此标准的操作系统的系统调用是一致的，例如Linux和Microsoft Windows NT。POSIX是一组操作系统调用的规范，本节介绍其中的文件I/O编程规范。

7.3.1 底层的文件I/O操作

和ANSI文件操作函数不同的是，POSIX文件操作的函数基本上和计算机设备驱动的底层操作（例如read，write等）是一一对应的。读者可以把POSIX文件操作理解为对设备驱动操作的封装。由此也可以看出，POSIX文件操作是不带数据缓冲的。

7.3.2 文件描述符

POSIX文件操作也使用文件描述符来标识一个文件。与ANSI文件描述符不同的是，POSIX文件描述符是int类型的一个整数值。POSIX文件描述符仅是一个索引值，代表内核打开文件记录表的记录索引。在一个系统中，文件打开关闭比较频繁，因此同一个POSIX文件描述符的值在不同时间可能代表不同的文件。

任何打开的文件都将被分配一个唯一标识该打开文件的文件描述符，为一个大于等于0的整数。需要注意的是，对于一个进程来说，打开文件的数量不是任意大小的。POISX没有规定一个进程可以打开文件的最大数目，不同的系统有不同的规定，例如Linux系统默认一个进程最多可以打开1024个文件，用户可以在console模式下通过ulimit –n命令查看系统允许进程打开文件的数量。

7.3.3 创建/打开/关闭文件

POSIX使用open()函数打开一个文件，使用creat()函数创建一个新文件，这两个函数在手册里常常放在一起介绍，因为open()函数在指定一定参数的情况下会隐含调用creat()函数创建文件。

1. open()函数
2. creat()函数
3. close()函数

7.3.4 读写文件内容

POSIX文件操作使用read()函数和write()函数对文件读写，和ANSI文件操作的fread()函数和fwrite()函数不同的是read()函数和write()函数是不带缓冲的，并且不支持记录方式。write()函数的定义如下：

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

7.3.4 读写文件内容

mode参数	说明
r或rb	为读打开文件
w或wb	为写打开文件，并把文件长度置为0（清空文件）
a或ab	在文件结尾添加打开
r+或r+b或rb+	为读和写打开
w+或w+b或wb+	为写打开文件，并把文件长度置0（清空文件）
a+或a+b或ab+	在文件结尾读写打开

7.3.5 文件内容定位

每当打开一个文件的时候，都会有一个与文件相关联的读写位置偏移量，相当于一个文件指针。文件偏移量是一个非负整数，表示相对于文件开头的偏移。通常情况下，文件的读写操作都是从当前文件偏移量开始，读写之后使文件偏移量增加读写的字节数。打开文件时，如果不指定O_APPEND方式，文件偏移量默认是从0开始。

POSIX文件操作提供了lseek()函数用于设置文件偏移量，函数定义如下：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

7.3.5 文件内容定位

参数值	解释
SEEK_SET	从文件开始处设置文件偏移量
SEEK_CUR	从当前文件偏移量设置
SEEK_END	从文件结尾处设置文件偏移量

7.3.6 修改已打开文件属性

fcntl()函数提供了获取或者改变已打开文件性质的功能，函数定义如下：

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

7.3.7 POSIX文件编程实例

最后给出一个**POSIX**操作文件的例子，完成文件的创建，读写等操作。

7.4 小结

本章讲解了Linux应用程序开发有关的基本技术，包括程序在内存中的结构、文件管理等内容。计算机应用程序是可以直接与用户打交道的，用户的功能需求几乎都是通过应用程序实现的。读者应该掌握应用程序的基本结构，为后面的编程开发打下基础。第8章讲解应用程序开发最常用的技术之一，多线程和多进程程序开发。

第8章　开发多进程/线程程序

现代计算机操作系统有两大功能：硬件控制和资源管理。资源管理主要是对软件资源的管理。现代计算机操作系统都是基于多任务的，为用户提供了多任务的工作环境。多个任务可以分享CPU的时间片，达到了在一个CPU上运行多个任务的目标，也为用户应用程序的开发创造了便利的环境。本章讲解Linux多进程和多线程开发相关知识，主要内容如下：

进程的概念

线程的概念

多进程和多线程的工作原理

多进程和多线程开发

8.1 多进程开发

Linux是一个UNIX类兼容的操作系统，为用户提供了完整的多进程工作环境。UNIX类操作系统最大的特点就是支持多任务，学习和使用多任务编程对Linux编程十分有必要。本节将讲解Linux多进程程序开发。

8.1.1 什么是进程

一般把进程定义成正在运行的程序的实例，简单的说，进程就是一个正在运行的程序。在第七章编写的代码，经过编译后，生成了一个可执行的文件，称作一个程序。当运行可执行文件以后，操作系统会执行文件中的代码，在CPU上运行的这组代码被称作进程。

从进程的概念不难看出，进程是一个动态的概念。实际上，一个进程不仅包含了正在运行的代码，也包括了运行代码所需要的资源（包括用户用到的资源，和操作系统需要用的资源）。操作系统通过一个称作PCB（进程控制块：Process Control Block）的数据结构管理一个进程。在操作系统看来，进程是操作系统分配资源的最小单位。

8.1.2 进程环境和属性

在Linux系统，C程序总是从main()函数开始的，当用户编写好的程序在运行的时候，操作系统会使用exec()函数运行程序，在调用main()函数之前，exec()系统调用会先调用一个特殊的启动例程，负责从操作系统内核读取程序的命令行参数，为main()函数准备好工作环境。

一个进程除了能获得操作系统提供的环境变量外，还具备自身的基本属性，主要包括：

进程号（PID：Process ID）：

父进程号（PPID：Parent Process ID）：

进程组号（PGID：Process Group ID）：

真实用户号（UID：User ID）：

真实组号（GID：Group ID）：

有效用户号（EUID：Effective User ID）：

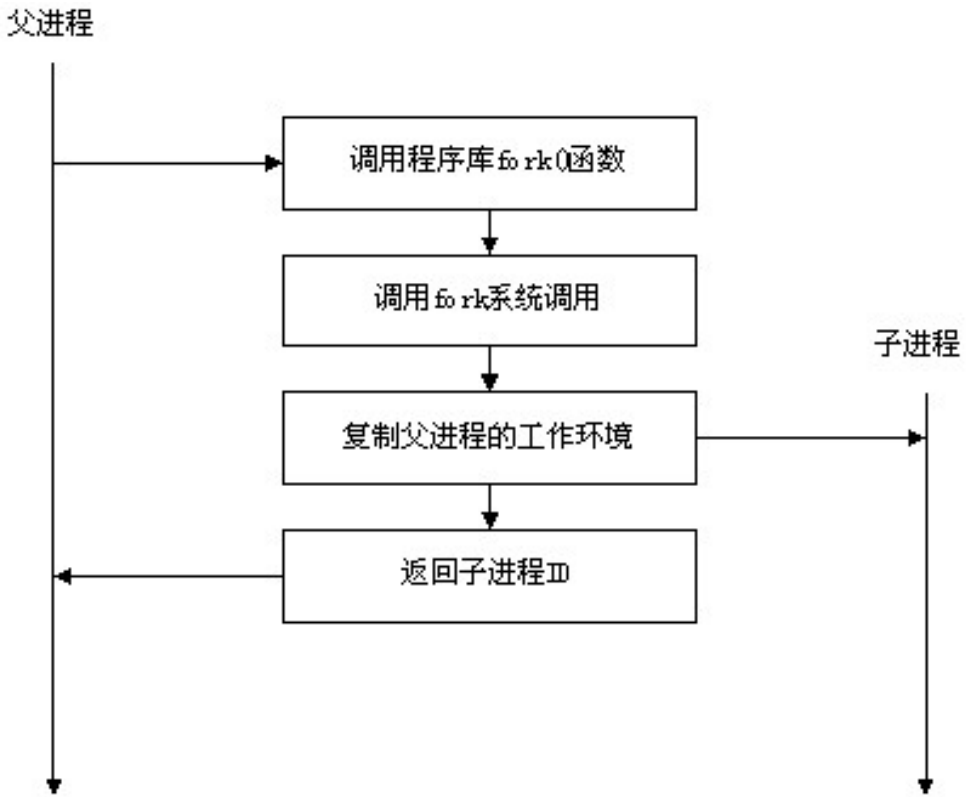
有效组号（EGID：Effective Group ID）：

8.1.3 创建进程

Linux系统通过fork()系统调用创建一个进程，fork()函数定义如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

8.1.3 创建进程



8.1.4 等待进程结束

子进程虽然是独立于父进程的，但是和父进程直接是有直接关系的。子进程从属于父进程，整个系统的进程是一个倒过来的树形结构，所有的进程都是从init进程创建来的。当进程结束的时候它的父进程会收回子进程的资源。这时会产生一个问题，当父进程创建子进程以后，两个进程是无序运行的。如果父进程先于子进程结束，那么子进程就会因为找不到父进程的进程号而无法通知父进程，导致资源无法释放。因此，需要一种方法让父进程知道子进程在什么时候结束。

Linux系统提供给了一个waitpid()函数，它的作用是等待另外一个进程结束，函数定义如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

8.1.5 退出进程

Linux提供了几个退出进程相关的函数exit()、_exit()、atexit()和on_exit()。exit()函数的作用是退出当前进程，并且尽可能释放当前进程占用的资源。_exit()函数作用也是退出当前进程，但是并不试图释放进程占用的资源。atexit()函数和on_exit()函数作用都是为程序退出时指定调用用户的代码，区别在于on_exit()函数可以为设定的用户函数设定参数。这几个函数的定义：

```
#include <stdlib.h>
int atexit(void (*function)(void));
int on_exit(void (*function)(int , void *), void *arg);
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

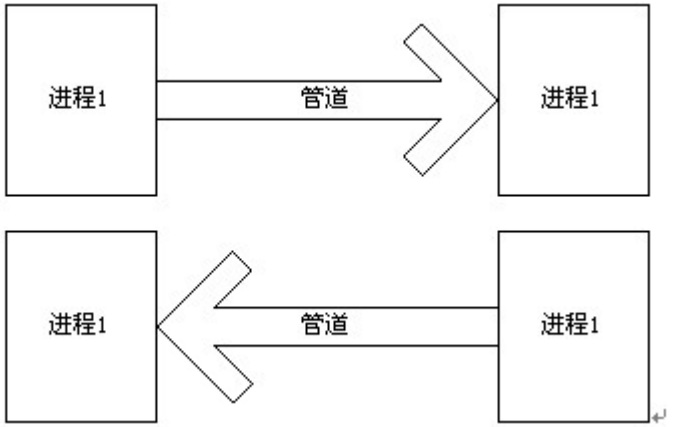
8.1.6 常用进程间通信的方法

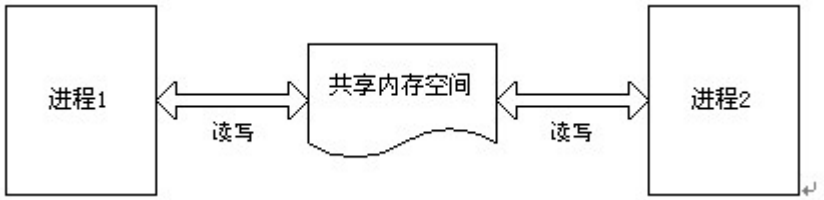
在支持多进程的操作系统里，用户可以创建多个进程，分别处理不同的功能。多进程机制为处理不同的数据带来好处。但是，在实际处理过程中，经常需要在不同的进程之间传递数据。如有两个进程，一个读取不同用户的配置文件并且解析配置文件，另一个进程需要把每个用户的配置发送到远程的服务器，这样的两个进程需要数据的传递，这个时蚓突嵊玫浇　碳澍ㄐ拧

Linux提供了多种进程间通信的方法，常见的包括管道、FIFO、消息队列、信号量、共享存储以及通过socket也可以实现不同进程间的通信。本节简述管道和共享内存两种进程间通信方法。

1. 管道
2. 共享内存

8.1.6 常用进程间通信的方法





8.1.7 进程编程实例

在本节的最后，给出一个多进程编程的综合实例，程序会创建两个进程，在父进程和子进程之间通过管道传递数据，父进程向子进程发送字符串“exit”表示让子进程退出，并且等待子进程返回；子进程查询管道，当从管道读出字符串“exit”的时候结束。

8.2 多线程开发

多进程为用户编程和操作带来了便利。但是对于操作系统来说，进程占有系统资源，进程的切换也给操作系统带来了额外的开销。每次创建新进程会把父进程的资源复制一份到子进程，如果创建多个进程的话，会占用大量的资源。此外，进程间的数据共享也需要操作系统的干预。由于进程的种种缺点，提出了线程的概念。

8.2.1 线程的概念

线程是一种轻量级的进程。于进程最大的不同是，线程没有系统资源。线程是操作系统调度的最小单位，可以理解为一个进程是由一个或者多个线程组成的。在操作系统内核中，是按照线程作为调度单位来调度资源的。在一个进程内部，多个线程之间的资源是共享的。也就是说，一个进程内部所有线程拥有相同的代码地址空间和数据空间，任意线程可以访问其他所有线程的数据。

8.2.2 进程和线程对比

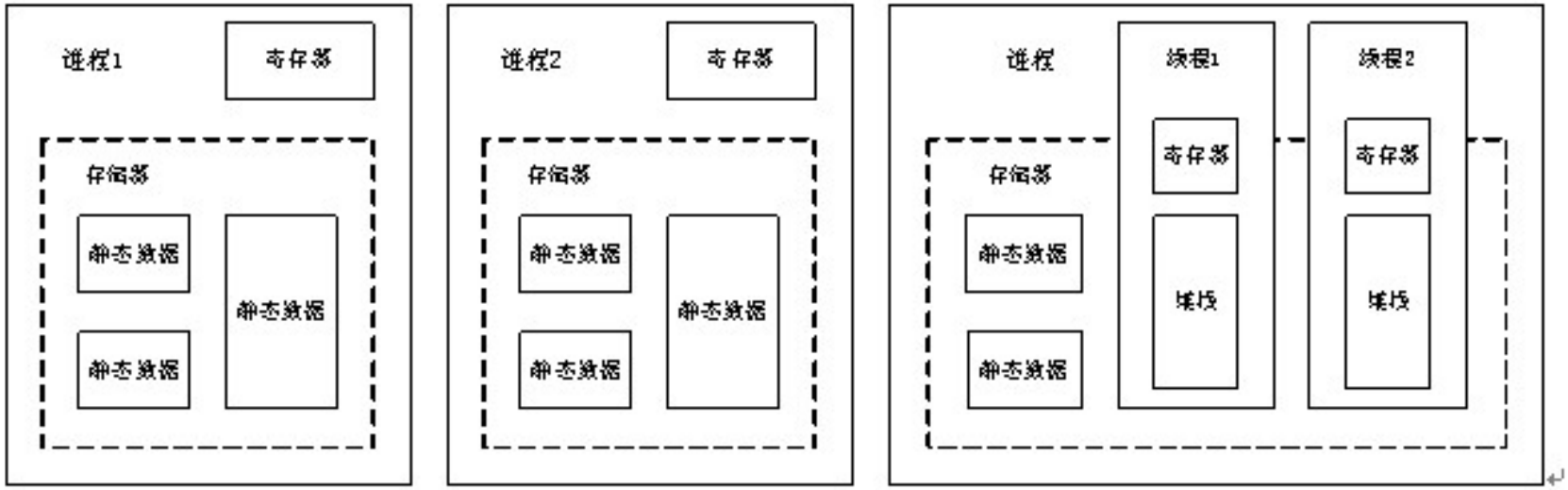
进程和线程有许多相似之处，但是也有许多不同：

资源分配不同。

工作效率不同。

执行方式不同。

8.2.2 进程和线程对比


进程1
寄存器
存储器
静态数据
静态数据
静态数据
进程2
寄存器
存储器
静态数据
静态数据
静态数据
进程
线程1
线程2
寄存器
寄存器
存储器
静态数据
静态数据
堆栈
堆栈

8.2.3 创建线程

Linux系统开发多线程程序大多使用pthread库，pthread库是符合POSIX线程标准的一个应用库，提供了线程的管理和操作方法。pthread库对线程操作的函数基本都以pthread开头，创建线程的函数定义如下：

```
#include <pthread.h>
int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

8.2.4 取消线程

线程的退出有几种条件，当线程本身的代码运行结束后，会自动退出；或者线程代码中调用**return**也会导致线程退出；还有一种情况是通过其他的线程把一个线程退出，**pthread**库提供了**pthread_cancel()**函数用来取消一个线程的执行。函数定义如下：

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

8.2.5 等待线程

在线程操作实例中，主线程使用sleep()函数暂停自己的运行，等待新创建的线程结束。使用延迟函数的方法在简单的程序中还能对付，但是复杂一点的程序就不好用了。由于线程的运行时间不确定，导致程序的运行结果无法预测。pthread库提供了一种等待其他线程结束的方法，使用pthread_join()函数等待一个线程结束，函数定义如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

8.2.6 使用pthread库线程操作实例

本节最后给出一个多线程操作实例，在主程序中创建两个线程**mid_thread**和**term_thread**，**mid**线程不断等待**term**线程终止它，并且每**2**秒打印一次等待的次数。**term**接收从主函数传进来的**mid**线程的**id**。如果线程**id**合法，就调用**pthread_cancel()**函数结束**mid**线程。

8.3 小结

本章讲解了Linux应用程序开发最重要的两种技术：多进程和多线程。多进程和多线程技术是应用最广泛的技术之一，使用该技术可以并发处理业务流程，充分利用计算机资源提高业务处理能力。理解多进程和多线程最关键的是建立并发工作的概念，读者在学习的时候应该多实践，通过实践加深理解。第9章讲解目前最常用的应用网络程序开发。

第9章　网络通信应用

在信息社会，随着互联网的普及，网络应用越来越广泛，通过互联网传输信息成为PC的必备要素。在嵌入式设备上，也越来越多的利用网络传输信息。Linux操作系统从一开始就提供网络功能，并且，Linux上的socket库兼容BSD socket库，为开发网络应用提供良好的支持。对应用程序员来说，掌握socket开发可以快速的实现网络应用程序。本章主要内容如下：

TCP/IP协议簇介绍

socket通信的概念

通过socket进行面向数据流的通信

通过socket进行面向数据报的通信

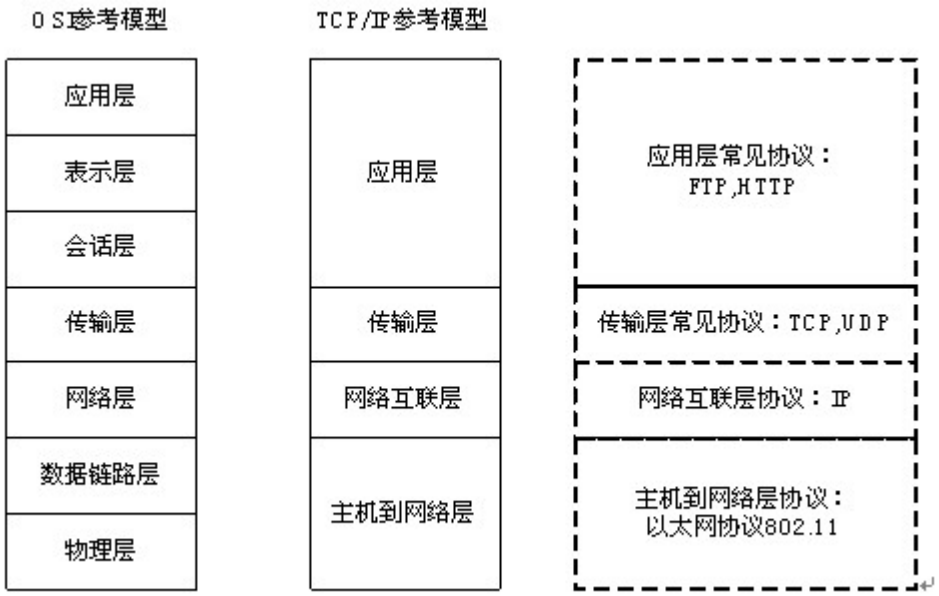
socket开发的高级应用

9.1 网络通信基础

互联网（internet）是目前世界上应用最广泛的网络，最早从美国军方的科研项目ARPA（Advanced Research Projects Agency）发展而来。互联网采用TCP/IP协议传输数据，虽然TCP/IP协议并不是ISO规定的标准协议，但是作为应用最广泛的协议已经成为大规模网络通信的事实标准。本节介绍TCP/IP协议簇以及其中重要的IP协议、TCP协议和UDP协议。

9.1.1 TCP/IP协议族

TCP/IP协议实际上是由一组协议组成的，通常也称作TCP/IP协议簇。根据ISO/OSI参考模型对网络协议的规定，对网络协议划分为7层。



9.1.2 IP协议

在图看出，IP协议工作在传输层，负责数据包的传输管理。IP协议实现两个基本功能：寻址和分段。寻址是IP协议提供的最基本功能，IP协议根据包头中目的地址传送数据报文。在传送数据报文过程中，IP协议可以根据目的地址选择报文在网络中的传输路径，这个过程称作路由。

分段是IP协议一个重要功能。由于不同类型的网络之间传输的网络报文长度是不同的，为了能适应在不同的网络中传输TCP/IP协议报文，IP协议提供分段机制帮助数据包穿过不同类型的网络。IP协议在协议头记录了分段后的报文数据，但是IP协议并不关心数据的内容。

9.1.2 IP协议

<table>
<tr><td>版本</td><td>IHL</td><td>服务类型</td><td colspan="2">总长度</td></tr>
<tr><td colspan="3">标识（Identification）</td><td>标记</td><td>段偏移量</td></tr>
<tr><td colspan="2">生存时间</td><td>协议</td><td colspan="2">头校验码</td></tr>
<tr><td colspan="5">源地址</td></tr>
<tr><td colspan="5">目的地址</td></tr>
<tr><td colspan="4">选项</td><td>填充</td></tr>
</table>

9.1.3 TCP协议

TCP协议是一个传输层协议。如图9-1所示，TCP协议位于网络互联层后，是IP协议的上层协议。TCP是一个面向连接的可靠传输协议。在一个协议栈处理程序中，如果发现数据包的IP层后携带了TCP头，会把数据包交给TCP协议层处理。TCP协议层对数据包排序并进行错误检查，按照TCP数据包头中的序列号排序，如果发现排序队列中少某个数据包，则启动重传机制重新传送丢失的数据包。

TCP协议层处理完毕后，把其余数据交给应用层程序处理，如FTP的服务程序和客户程序。面向连接的应用几乎都使用TCP协议作为传输协议。TCP传输协议有高度可靠性，可以最大限度保证数据在传递过程中不丢失。

9.1.4 UDP协议

UDP与TCP一样是传输层协议，但是UDP协议没有控制数据包的顺序和出错重发机制。因此，UDP的数据传输时不稳定的。通常UDP被用在对数据要求不是很高的场合，如查询应答服务等。使用UDP作为传输层协议的有NTP（网络时间协议）和DNS（域名服务系统）。

UDP另一个重要问题就是安全性不高。由于UDP没有连接的概念，在一个数据传输过程中，UDP数据包可以很容易的被伪造或者篡改。

9.1.5 学习分析协议的方法

网络协议一般都比较抽象，给人感觉枯燥。学习网络协议需要一个直观的认识，推荐读者使用网络协议分析的工具分析协议。目前有很多的网络协议分析工具，著名的**Sniffer**就是一款专业的网络协议分析利器，本书介绍一个比较流行的工具**Ethereal**，这是一个开源的网络协议分析工具，功能十分强大，使用**libpcap**库做数据包解析，使用**GTK+**库做界面，由于这两个库是跨平台的，所以**Ethereal**可以在多种平台使用。**Ethereal**最大的特点是支持用表达式书写包过滤条件，同时支持常见协议的深度分析，如**HTTP**，**SIP**等。**Ethereal**最新版本已经更名为**WireShark**，官方网站是**http://www.wireshark.org**，官方网站有软件的使用手册以及下载。

软件的安装本书不做介绍，安装过程一般不需要选择，按照提示一步一步进行即可。本节介绍**WireShark**软件的使用。

9.2 socket通信基本概念

socket常被翻译成套接字或者插口，socket实际上就是网络上的通信端点。使用者或应用程序只要连接到socket便可以和网络上任何一个通信端点连接，传送数据。socket封装了通信的细节，在Linux系统，为使用者提供了类似文件描述符的操作方法，程序员可以不必关系通信协议内容而专注应用程序开发。根据数据传送方式，可以把socket分成面向连接的数据流通信和无连接的数据报通信。

9.2.1 创建socket对象

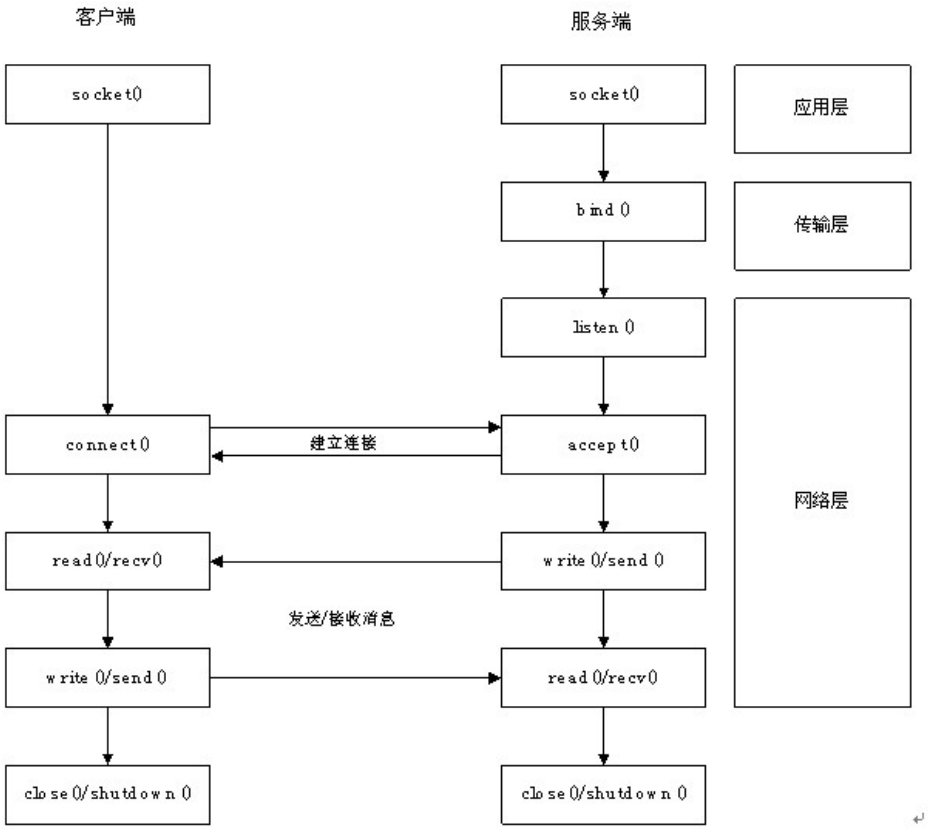
在使用socket通信之前，需要创建socket对象。对应用程序员来说，soeket对象就是一个文件句柄，通常使用socket()函数创建socket()对象。函数定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

9.2.2 面向连接的socket通信实现

面向连接的数据流通信在TCP/IP协议簇是使用TCP作为传输层协议通信，按照TCP协议的要求，通信双方需要在传输数据前建立连接，术语上称作“TCP的三次握手”。

9.2.2 面向连接的socket通信实现



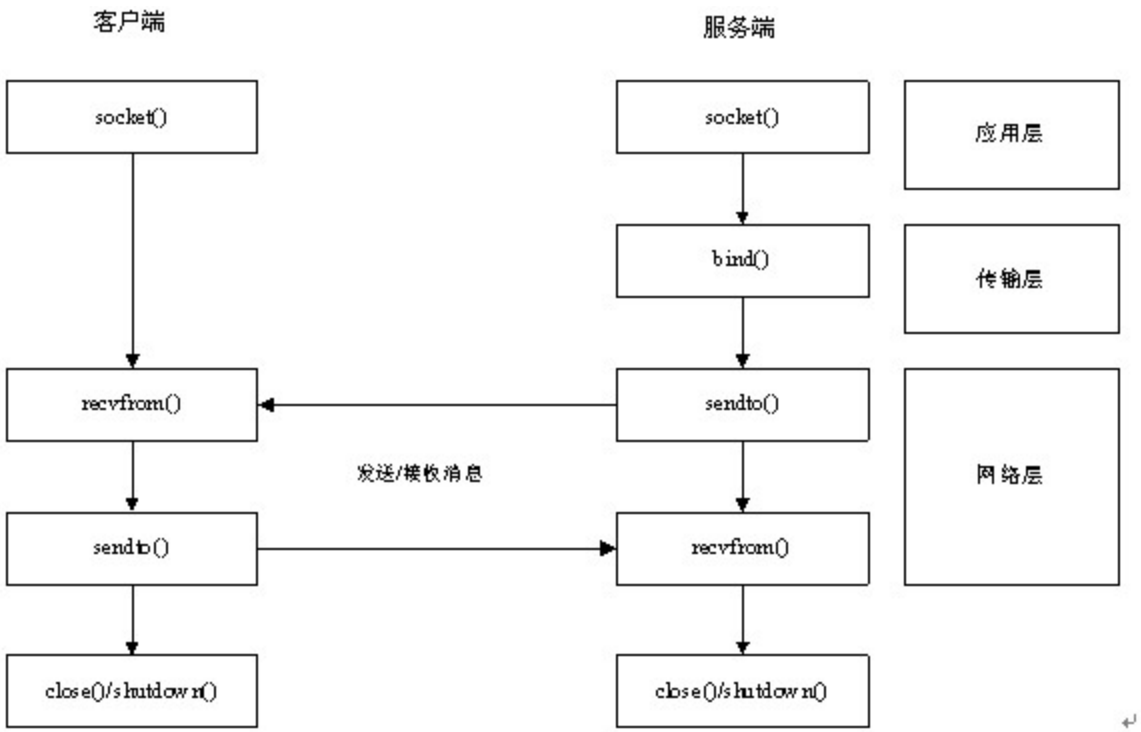
9.2.3 面向连接的echo服务编程实例

本节给出一个**echo**服务的编程实例，**echo_serv.c**是服务端源代码，提供创建服务端，绑定套接字到本机**IP**和**8080**端口，当收到客户端发送的字符串就在屏幕打印出来，并且把字符串发送给客户端，如果客户端发送”**quit**”，服务器端退出。

9.2.4 无连接的socket通信实现

无连接的socket通信相对于建立连接的流socket简单，因为数据传输过程中不能保证到达，常用在一些对数据要求不高的地方，如在线视频等。无连接的套接字不需要建立连接，省去了维护连接的开销，所以，同样环境下一般比流套接字传输数据速率快。实际应用中，一些应用软件会自己维护无连接的套接字数据传输状态。无连接的套接字使用TCP/IP协议簇的UDP协议传输数据。

9.2.4 无连接的socket通信实现



9.2.5 无连接的时间服务编程实例

无连接的套接字通信比较简单，本节给出一个获取时间的例子，服务端程序time_serv.c负责创建socket并且绑定到本机9090端口，然后等待客户端发出请求，当收到客户端发送的请求时间命令”time”以后，生成当前时间的字符串发送给客户端。客户端建立socket以后，直接向指定的服务端发送请求时间命令，之后等待服务端返回，发送退出命令，关闭连接。

9.3 socket高级应用

9.2节介绍了socket编程的基础知识，包括面向连接的流通信和无连接的数据报通信，并且给出了例子。由于网络通信过程中有许多不确定因素，数据的传输不可能每次都正确，需要对数据发送和接收做超时处理；对于一个服务器来说，需要同时管理多个客户端的连接。这些技术就是本节要介绍的。

9.3.1 socket超时处理

实际的网络通信数据常会因为各种网络故障导致传输失败，在应用程序里需要对数据发送和接收做对应的超时处理，超时指的是预先假定一次数据传输需要的时间，如果超过这个时间没有得到反馈，认为数据传输失败，socket库提供了两个强大的函数setsockopt()和getsockopt()用来设置套接字和得到套接字参数，函数定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

9.3.2 使用select机制处理多连接

当服务端根据客户端的请求创建多个连接以后，每个连接对应不同的套接字，因为recv()函数默认是阻塞的，会造成在等待一个客户端套接字返回数据的时候整个进程阻塞，而无法接收其他客户端套接字数据。这时候需要一个可以处理多个连接的方法，socket库提供了两个函数select()和poll()用来等待一组套接字句柄的读写操作。

Linux系统提供了select()函数和poll()函数两个网络套接字复用的工具。使用select()函数和poll()函数可以向系统说明在什么时间需要安全的使用网络套接字描述符。如程序员可以通过这两个函数的返回结果知道哪个套接字描述符上有数据需要处理。使用select()函数和poll()函数后，程序可以省去不断的轮询网络套接字描述符的步骤。在后台运行网络程序，当被监听的网络套接字有数据的时候，系统会触发应用程序。因此，使用select()和poll()函数可以显著提高网络应用程序的工作效率。

9.3.3 使用poll机制处理多连接

poll()函数提供与select()函数类似的功能，解决了select()函数存在的一些问题，并且函数调用方式也更加简单。函数定义如下：

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

事件名称	解释
POLLIN	有数据可读
POLLRDNORM	有普通数据可读
POLLRDBAND	有优先数据可读
POLLPRI	有紧迫数据可读
POLLOUT	写数据不会导致阻塞
POLLWRNORM	写普通数据不会导致阻塞
POLLWRBAND	写优先数据不会导致阻塞
POLLMSG	SIGPOLL消息可用
POLLER	指定的文件描述符发生错误（仅出现在revents域）
POLLHUP	指定的文件描述符挂起事件（仅出现在revents域）
POLLNVAL	指定的文件描述符非法（仅出现在revents域）

9.3.4 多线程环境socket编程

select()函数和poll()函数可以解决一个进程需要同时处理多个网连接的问题，但是使用select()函数和poll()函数监控文件句柄仍然需要等待，如果后面还有需要处理的工作仍然不能同步完成。可以利用第8章介绍的多线程技术在一个进程里处理不同的网络连接，为每个连接建立不同的线程，进程不会因为读写函数阻塞，多个连接操作数据也不会互相干扰。多线程处理连接的时候需要注意一个线程直接的全局变量是共享的，所以每个连接对应的套接字句柄应该保存在线程内部。

9.4 小结

本章讲解socket编程。首先介绍了TCP/IP协议簇，在第2节讲解socket编程的基础知识，包括面向连接的流套接字和无连接的数据报编程，这部分是socket编程最基本的，初学者应该把这两种类型的套接口通信模式与TCP/IP协议的层次关系对比，可以更快的理解里面的含义。第3节介绍了socket编程的一些高级技术，主要是集中在如何对阻塞数据处理，包括超时机制，多线程机制，读者可以参考Linux的man手册得到更多的细节。第10章另一种常见的通信方式－串口通信编程。

第10章　串行口通信编程

目前主流的PC机尤其是笔记本电脑，串口已经很少见到了，但是串口却是嵌入式开发中最常用的硬件接口。串口有驱动简单的特点，几乎所有的嵌入式开发板和设备都提供了串口。在嵌入式开发中，串口通常用来打印设备状态信息和命令行，甚至于有的时候只能通过串口得到设备状态（如设备刚启动的时候）。本章介绍串口基本的软硬件知识，以及如何在应用程序中利用串口收发数据，主要内容如下：

串口硬件介绍

常见的串口协议

串口应用程序入门

利用串口通过手机发送短信

10.1 串口介绍

在计算机领域，串口可以说是历史悠久而且应用广泛。从最早的PC机到目前工业控制领域广泛应用的工业计算机、以及嵌入式系统等，都提供了串口。串口有功能简单、成本低、便于连接等优点，是许多嵌入式系统必备接口之一。

10.1.1 什么是串口

串口是串行接口（Serial Port）的简称，是计算机一种常用的接口，由于连线少、通讯控制简单而得到广泛的使用。串口有几种标准，常见的一种称作RS232接口的标准是在1970年由美国电子工业协会（EIA）和几家计算机厂商共同制定的。RS232标准应用广泛，其全称是“数据终端设备（DTE）和数据通讯设备（DCE）串行二进制数据交换接口”，该标准定义了串口的电气接口特性和各种信号电平等。

标准串口协议支持的最高数据传输率是115Kbps。一些改进的串口控制器支持更高甚至460Kbps的数据传输率，如增强型串口ESP（Enhanced Serial Port）和超级增强型串口Super ESP。

RS232串口使用D型数据接口，最初有9针和25针两种连接方式。随着计算机技术的不断进步，25针的串口连接方式已经淘汰，目前所有的RS232串口都使用9针连接方式。

10.1.2 串口工作原理

串口通过直接连接在两台设备间的线发送和接收数据，两台设备通信最少需要三根线（发送数据、接收数据和接地）才可以通信。以最常见的RS232串口为例，通信距离较近时（<12m），可以用电缆线直接连接标准RS232端口。如果传输距离远，可以通过调制解调器（MODEM）传输。因为串口设备工作频率低且容易受到干扰，远距离传输会造成数据丢失。

10.1.2 串口工作原理

针号	功能说明	缩写
1	数据载波检测	DCD
2	接收数据	RXD
3	发送数据	TXD
4	数据终端准备	DTR
5	信号地	GND
6	数据设备准备好	DSR
7	请求发送	RTS
8	清除发送	CTS
9	振铃指示	BELL

10.1.3 串口流量控制

常见的串口工具软件都提供了RTS/CTS与XON/XOFF选项。这两个选项对应RS232串口的两种流量控制方式。串口流量控制主要应用于调制解调器的数据通讯，对于普通RS232串口编程，了解一点流量控制方面的知识是有好处的。

1.什么是串口流量控制

2.硬件流量控制

3.软件流量控制

10.2 开发串口应用程序

Linux操作系统对串行口提供了很好的支持。Linux系统中串口设备被当做一个字符设备（第22章详细讲解）处理。PC机安装Linux系统后在/dev目录下有若干个ttySx（x代表从0开始的正整数）设备文件。ttyS0对应第一个串口，也就是Windows系统下的串口设备COM1，以此类推。

10.2.1 操作串口需要用到的头文件

在Linux系统操作串口需要用到以下头文件：

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <unistd.h>
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    <termios.h>
#include    <errno.h>
```

10.2.2 串口操作方法

操作串口的方法与文件类似，可以使用与文件操作相同的方法打开和关闭串口、读写以及使用**select()**函数监听串口。不同的是，串口是个字符设备，不能使用**fseek()**之类的文件定位函数。此外，串口是个硬件设备，还可以设置串口设备的属性。

10.2.3 串口属性设置

10.1节讲解串口的基本知识，提到串口的基本属性包括波特率、数据位、停止位和奇偶校验等参数。Linux系统通常使用termios结构存储串口参数，该结构在termios.h头文件定义如下：

```
struct termios
{   unsigned short  c_iflag;
unsigned short  c_oflag;
unsigned short  c_cflag;
unsigned short  c_lflag;
unsigned char  c_line;
unsigned char  c_cc[NCC];
};
```

10.2.4 与Windows串口终端通信

本节给出一个和Windows串口终端通信的例子。两台PC通过串口相连，其中一台PC运行Windows系统，通过XShell软件（5.5.3节介绍）打开COM1；另一台PC运行Linux系统。

10.3 串口应用实例-手机短信发送

手机是目前使用最广泛的通信设备之一，许多手机都提供了与PC互联的功能，其中最重要的一个接口就是串口（一些提供USB接口的手机指令收发是把USB设备虚拟一个串口设备进行通信的）。在GSM（全球数字移动电话网络）协议中规定了一组AT指令用于手机与其他设备通信，其中提供了发送短信的方法。本节讲解如何利用手机发送短信并且给出了实例。

10.3.1 PC与手机连接发送短信的物理结构

在进行本章的试验之前需要建立一个手机和PC机之间的连接。

10.3.2 AT指令介绍

AT指令集是GSM网络中网络设备之间发送控制信息的标准指令集。GSM网络终端设备（TE）或者数据终端设备（DTE）可以向终端适配器（TA）发送AT指令。使用AT指令，用户可以控制DTE发送短信息、呼叫电话、读写电话本、发送传真等。

AT指令由手机制造商诺基亚、爱立信、摩托罗拉等共同研制，其中包括了短消息（SMS）控制功能。对SMS的控制有Block模式、文本模式和协议数据（PDU）模式三种。目前主要使用PDU模式，其它两种模式逐步被淘汰。

10.3.3 GSM AT指令集

GSM07.05协议中定义了一组与SMS（短消息）有关的指令。

AT指令	功能
AT+CMGC	向DTE发送一条短消息
AT+CMGD	删除存储在SIM卡中指定的短消息
AT+CMGF	发送短消息模式：0-PDU模式；1-文本模式
AT+CMGL	打印存储在SIM卡中的短消息
AT+CMGR	读取短消息内容
AT+CMGS	发送短消息
AT+CMGW	把准备发送的短消息存储在SIM卡
AT+CMSS	发送存储在SIM卡的短消息
AT+CNMI	显示接收到的短消息
AT+CPMS	短消息存储设备选择
AT+CSCA	设置短消息中心号码
AT+CSCB	使用蜂窝广播消息
AT+CSMP	设置文本模式参数
AT+CSMS	选择短消息服务方式

10.3.4 PDU编码方式

通常发送短信使用PDU模式，在GSM协议中对PDU模式发送短信的数据做了规范。使用PDU模式发送短信需要接收号码、短消息中心号码和短消息内容三个数据。这三个数据的定义方法如下：

1. 接收号码生成方法
2. 短消息中心号码生成方法
3. 短消息内容生成方法
4. 组合成完整的消息格式

10.3.5 建立与手机的连接

PC与手机通过串口进行连接，早期的手机提供了数据线。手机串口数据线一端连接到手机上，另一端可以直接连接到PC的串口，这种方式不需要额外的驱动。最近几年生产的手机大多数都提供了USB接口，通过手机的USB驱动程序在手机与PC之间建立一个虚拟的串口设备。通常手机厂商提供的是适合Windows系统的驱动程序，Linux系统可以使用一个名为Gnokii的手机驱动软件。本节以NOKIA6300手机为例，讲解在Linux下如何对手机编程发送短消息。

提示：NOKIA6300使用S40系统，提供USB接口，其他使用类似系统的手机也可以采用采用类似的方法驱动手机。NOKIA早期的有些手机（比如NOKIA1110）提供了串口数据线，可以直接操作。

10.3.6 使用AT指令发送短信

10.3.4节讲解如何生成PDU模式的数据，在生成符合PDU模式的数据后，可以通过AT+CMGF指令和AT+CMGS指令发送一条短信。以10.3.4节的内容为例，使用AT指令发送短消息过程如下：

AT+CMGF=0<回车>

OK

AT+CMGS= msg_len<回车>

entire_msg<Ctrl+Z发送>

10.4 小结

本章讲解串口组成和工作原理、编程方法，并在最后给出一个操作手机发送短信的实例。串口的工作原理简单，是两台计算机设备之间传递数据的简单方式。串口编程入门比较容易，读者可以在自己的计算机上进行串口编程试验。第11章介绍Linux嵌入式系统的图形界面开发。

第11章　嵌入式GUI程序开发

许多嵌入式设备都提供了图形界面。由于嵌入式设备输入输出设备的限制，键盘和鼠标等传统的输入设备不便于使用，通过图形界面，可以很好的完成人机交互。在嵌入式Linux系统，有许多的图形库可以使用。本节重点介绍在嵌入式Linux上使用最广泛的Qt程序库，主要内容如下：

嵌入式Linux图形库介绍

Qt开发环境搭建

开发Qt应用程序

搭建嵌入式Qt工作环境

在嵌入式Linux系统使用Qt应用程序

11.1 Linux GUI介绍

GUI是Graphic User Interface的缩写，中文意思是图形用户接口。目前，几乎所有的操作系统都提供了GUI，GUI也逐渐成为操作系统图形界面的代名词。与其他的商业系统如Windows不同，Linux系统的开放特性让许多图形界面都可以运行在Linux系统下。实际上，Linux内核本身并没有图形处理能力，所有Linux系统的图形界面都是作为用户程序运行的。本节介绍Linux图形界面的发展和常见的几种图形界面。

11.1.1 Linux GUI的发展

从1981第一个计算机图形界面诞生到现在，计算机图形界面有着飞速的发展。与图形界面发展相应的是计算机硬件处理能力的不断提高。最初的图形界面仅提供了很简单的功能，而且不支持鼠标操作，受到硬件的限制，颜色位数也很低。在计算机图形界面发展过称中，XWindow、MacOS、Windows是发展的最好的三个系统。

XWindow采用C/S结构设计，几乎是Unix类系统图形界面的标准。XWindow的服务器向客户端提供图形输出能力，因此，一个XWindow服务器可以支持多个图形客户端。在多用户和多任务方面XWindow比其他图形系统更胜一筹。

11.1.2 常见的嵌入式GUI

Linux系统本身并没有图形界面，但是由于其开放性，有许多的自由软件图形库和图形界面。本节介绍几种目前最流行的图形界面。

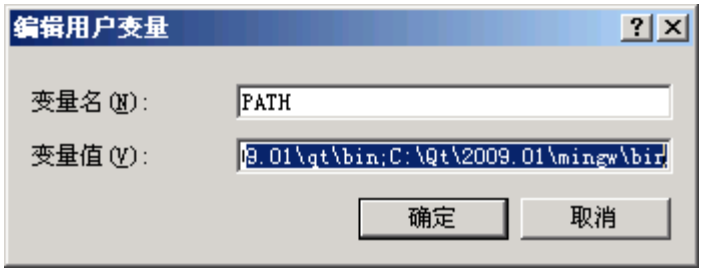
1. GNOME
2. KDE
3. Qt
4. MiniGUI

11.2 开发图形界面程序

Qt程序库是一个跨平台的程序库。Qt程序库提供了一套完整的开发环境，目前可以运行在Windows、Linux和MacOS。本书推荐在Windows环境下使用Qt开发环境，好处是可以与其他的开发工具一同使用。

11.2.1 安装Qt开发环境

在使用Qt开发环境之前，首先需要从http://www.qtsoftware.com/downloads/sdk-windows-cpp下载Windows版的Qt集成开发环境。下载完毕后，双击安装程序开始安装，安装过程比较简单，使用默认的配置即可。安装完毕后Qt开发环境被安装到c:\qt目录下。在使用开发环境之前，需要配置一下Qt开发环境。



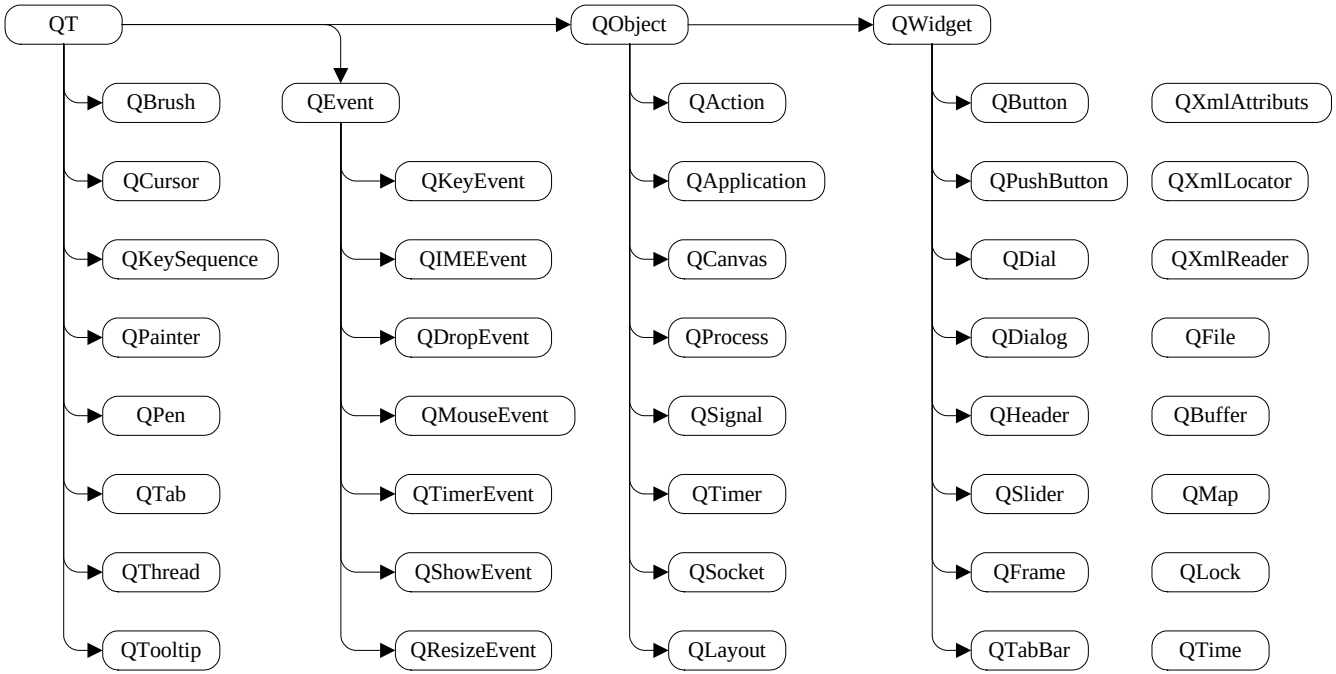
11.2.2 建立简单的Qt程序

Qt图形库的结构设计的非常合理，因此开发图形程序比较简单，本节先从一个最简单的例子入手，开发第一个Qt图形界面程序。

1. 基本的Qt图形界面应用程序
2. 文本界面风格的HelloWorld程序
3. 带有功能响应的Qt应用程序

11.2.3 Qt库编程结构

Qt图形库是一个组织严谨的C++类库。



11.3 深入Qt编程

在了解了Qt的库结构后，本节从几个稍微复杂点的例子入手，讲解Qt程序如何管理多个空间以及响应不同的事件。

11.3.1 使用Widget

11.2节的例子一个应用程序中只有一个控件，因此对于控件的布局不需要过多管理。通常有实际功能的应用程序都不止一个控件，因此需要对控件的布局进行管理，否则控件在窗体上的位置可能不固定。

Qt提供了QWidget机制管理窗体上控件的布局。QWidget是一个布局管理类，可以把QWidget理解为一个控件容器，在一个容器内可以容纳多个控件，容器可以设置控件的相对位置等。实际上，Qt支持层次关系的布局，在一个布局里面还可以有子布局，可以把窗体上的控件组织到不同的布局里面，最后把多个布局放到一个布局里，这样不仅能按照区域管理控件，也可以集中管理所有的控件。

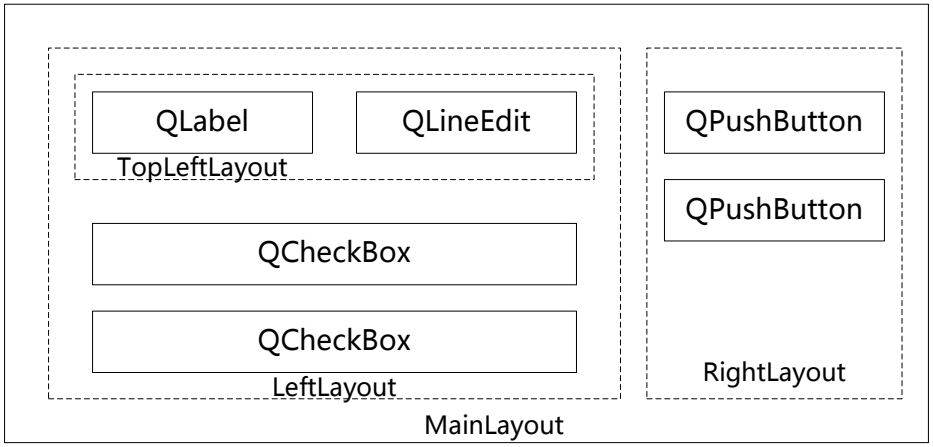
11.3.2 对话框程序设计

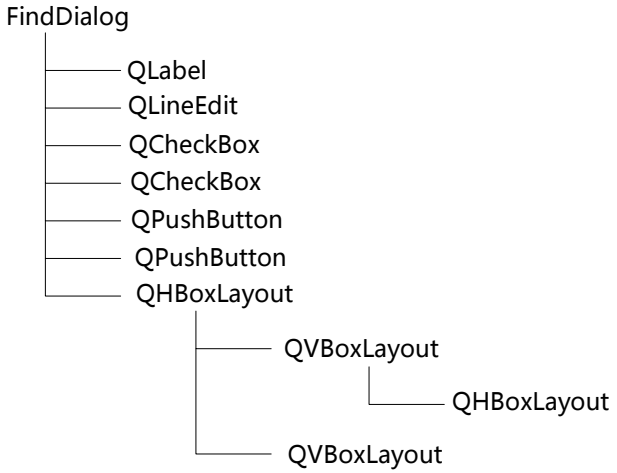
对话框是图形界面中经常见到的一类界面。对话框通常用来完成一类特定的功能，例如打开文件对话框，颜色设置对话框等。本节介绍如何使用Qt建立一个类似Windows系统查找对话框的界面。

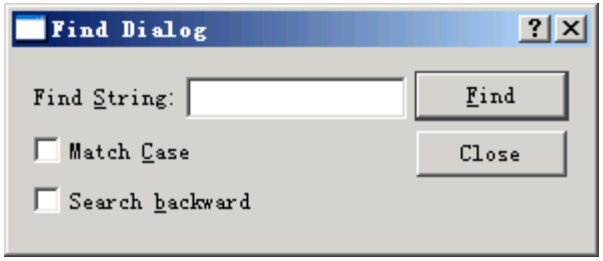
本节提供的例子共有三个文件，FindDialog.h文件声明了FindDialog类，FindDialog.cpp文件是FindDialog类的实现，main.cpp使用FindDialog创建应用程序。

1. 对话框头文件
2. 对话框实现代码
3. 创建FindDialog对话框

11.3. 2 对话框程序设计





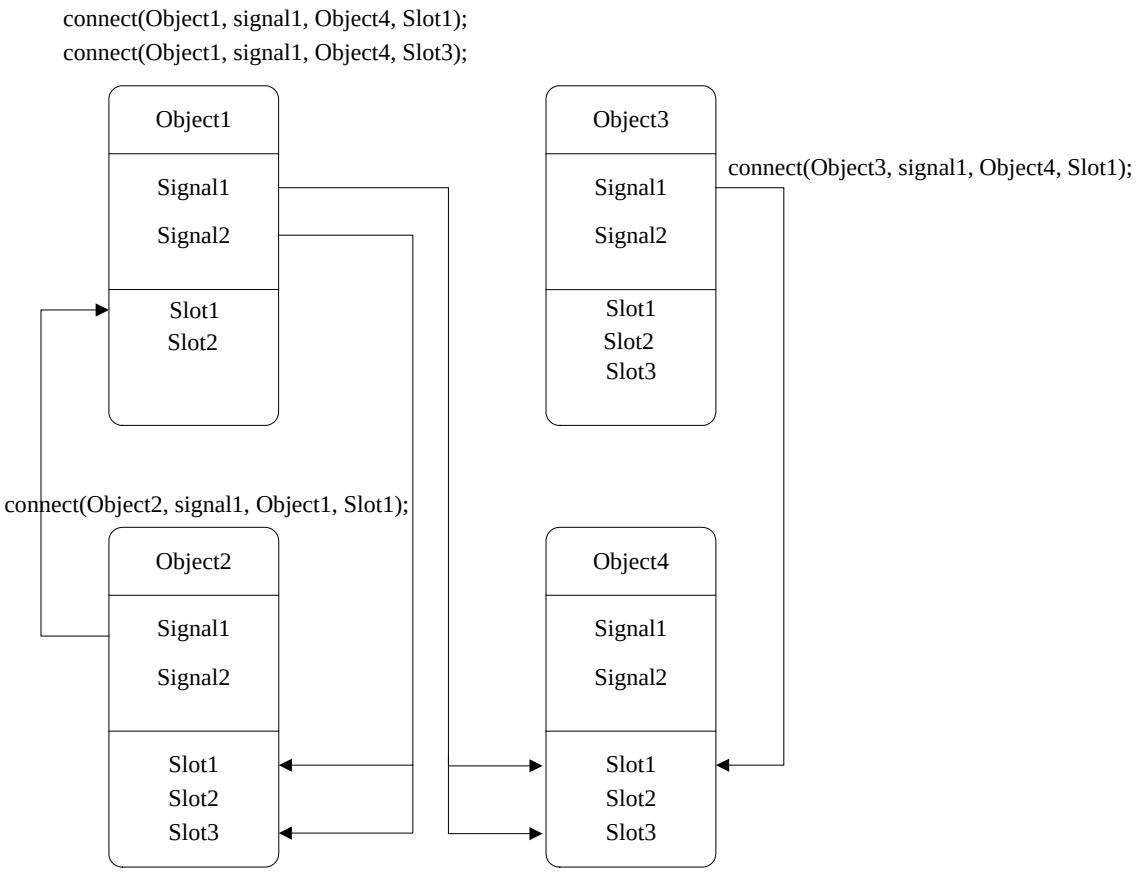


11.3.3 信号与槽系统

在例11-3读者会发现使用了connect()函数把按钮的单击事件与一个处理函数连接起来，connect()函数的原型如下：

connect(Object1, signal, Object2, slot);

11.3.3 信号与槽系统


connect(Object1, signal1, Object4, Slot1);
connect(Object1, signal1, Object4, Slot3);
Object1
Signal1
Signal2
Slot1
Slot2
Object3
Signal1
Signal2
Slot1
Slot2
Slot3
connect(Object3, signal1, Object4, Slot1);
connect(Object2, signal1, Object1, Slot1);
Object2
Signal1
Signal2
Slot1
Slot2
Slot3
Object4
Signal1
Signal2
Slot1
Slot2
Slot3

11.4 移植Qtopia到ARM开发板

Qt有一个专门为嵌入式系统使用的版本Qtopia。Qtopia是一组软件包的集合，使用Qtopia可以方便的在嵌入式Linux系统上建立和使用图形界面。Qtopia本身提供了管理界面和一些工具，用户程序也可以直接无需Qtopia的管理界面直接运行。用户在主机编写的Qt应用程序只需要经过交叉编译就可以运行在嵌入式开发板上。

11.4.1 需要的资源

在编译Qtopia之前，需要从Qt的官方网站下载下面的软件包：

konqueror-embedded-snapshot-20030705.tar.gz

qt-embedded-2.3.7.tar.gz

qt-x11-2.3.2.tar.gz

qtopia-free-1.7.0.tar.gz

tmake-1.11.tar.gz

11.4.2 准备工作

在开始编译Qtopia之前首先搭建编译软件包的环境，下面的步骤使用相对路径，可以在任意目录进行。

1. 解压缩所有软件包
2. 修改软件包目录名称

11.2.3 编译主机Qt工具

Qtopia交叉编译需要用到一些主机的库和应用程序，首先需要在主机编qt-x11库和qtopia库。

1. 编译qt-x11库
2. 编译qt库

11.2.4 交叉编译qtopia

在编译好qt主机开发工具后，开始交叉编译qtopia。qtopia的编译过程比较简单，如下：

```
cd qtopia/src
./configure  -platform linux-arm-g++
make
cd -
```

11.4 小结

本章介绍了Linux系统常见的图形环境，以Qt图形界面为例详细讲解了Qt的使用和开发过程，最后介绍了如何移植Qt到目标板。Qt是一个应用广泛的开源图形开发环境，读者在学习的过程中需要多实践，结合Qt的文档，探索Linux图形开发技术。第12章讲解软件开发过程中的项目管理方法。

第12章　软件项目管理

软件项目的管理是软件开发过程中很重要的一项工作。好的管理方法是一个软件项目成功的前提，而使用好软件管理工具能让软件项目开发事半功倍。开源软件项目的开放特性导致开发一个软件项目的人员可能分散在不同的地理位置，如Linux内核开发人员分布在全球数十个国家。开源软件项目开发中的交流与合作难度比任何一种商业软件都要高，因此需要管理软件项目的软件帮助开发人员完成协作和交流的问题。本章介绍在开源软件项目常用的管理技术，主要内容如下：

软件版本的概念

如何控制软件版本

开发文档管理

Bug缺陷跟踪系统

12.1 源代码管理

源代码是一个软件最重要的部分，软件的二进制程序都是从源代码编译生成的。学过计算机编程的读者在学习过程中都编辑过一些源代码，可以完成一些简单的功能，但是在学习编程过程中可能很少会体会到对源代码的管理问题。对于一个软件来说，无论从源代码的数量还是软件的功能，都远比一个小程序复杂。软件开发式多人合作的过程，对软件开发过程管理就很必要了。

12.1.1 什么是软件的版本

软件开发过程中通常会把完成某个功能的代码打包，用数字和字母的组合为软件的源代码或者二进制文件命名，表示完成一个阶段的工作，这种软件阶段性的名字称作软件版本。软件的版本不是随意命名的，有一定规律，不同的软件开发组织都有自己的软件命名方法。本节介绍几种常见的名规则。

1. GNU软件版本命名规则
2. 常见的软件版本命名含义

主版本号	子版本号	修正版本号	编译版本号

12.1.2 版本控制的概念

随着软件开发过程中不断的修改错误和发布新的功能，软件的版本随之增多。此外，发布给用户的版本与开发的版本往往不是一致的，12.1节讲到的多是发布给用户的版本，在开发过程中很可能会有许多的“中间版本”。

版本控制的目的就是解决软件开发过程中的版本问题。在开发过程中常会遇到同一文件多人修改，或者多人修改代码后同时提交的问题。

版本控制的一个重要功能是记录每个版本信息，在发生错误时能回退到某个指定的版本。试想一下，如果每个人都在修改自己的文件，当提交的时候发现问题，这个时候如果不能回退到之前某个可用的版本，工作可能就前功尽弃。软件版本控制还需要提供代码比对功能，帮助用户比较不同版本之间的差异。

12.2 版本控制系统Subversion

在开源软件领域，有许多的版本控制软件。早期的版本控制软件有大名鼎鼎的CVS，现在应用最广泛的软件版本控制软件是Subversion版本管理系统。此外，Linux内核开发团队使用了自己开发的GIT版本管理系统，也是一个不错的选择。

Subversion最大的特点是使用简单。Subversion继承了CVS仓库的概念，但是做了很多的改进，支持Unicode编码，并且提供了许多功能。本节介绍Subversion的安装配置和使用方法。

12.2.1 在Linux系统使用Subversion服务端

本节介绍在Ubuntu Linux 8.04版本上安装Subversion。Ubuntu Linux使用apt管理软件包，安装Subversion的过程非常简单。安装配置过程如下：

12.2.2 在Windows系统使用TortoiseSVN客户端

Windows提供相良好的图形界面，嵌入式开发中，通常把客户端工具安装在Windows系统，如编辑工具、代码管理工具等；而把编译环境等放在一个Linux系统。这样充分利用了两个系统的优势。

在Windows系统下有许多开源的Subversion客户端。TortoiseSVN是目前使用最广泛的Subversion客户端，该客户端界面简洁、功能丰富，并且比较稳定。软件安装配置过程如下：

12.3 开发文档管理——常见的开发文档

随着技术的进步，小作坊式的软件开发年代已经过去。目前的软件开发能力在不断提升，用户对软件的功能和性能要求也越来越高，软件开发质量受到关注。在软件开发过程中，各种数据和代码的管理需要经过统筹安排和管理，出现了各种软件开发文档，用于控制软件开发过程。

软件开发文档是与开发阶段对应的。一般来说，每个阶段至少产生一种文档。软件开发文档描述了在软件开发各阶段中不同的任务。不同组织和公司有不同的开发文档和规范，使用不同的开发模型产生的文档内容也不相同。

软件开发文档指导不同阶段相关人员的工作，不同阶段的人员也产生某个阶段的最终文档。如需求设计文档指导项目经理做软件的框架设计，产生概要设计文档；程序员使用概要设计文档了解软件某部分的功能，然后做具体的细化设计详细设计文档。按照软件开发的阶段，通常会生成以下钢治牡担

12.3.1 可行性研究报告

软件项目的可行性研究报告需要列举出需要的技术、人员、资金、时间周期以及法律等方面的因素，最终目的是论证一个软件项目是否可以开发。可行性研究报告通常由软件团队的高层，或者是软件项目发起人、投资人等参加。可行性研究报告中还应当对于现有的资源给出几种不同的解决方案供讨论使用。软件可行性研究报告的结果直接决定了一个软件项目是否启动。

12.3.2 项目开发计划

项目开发计划的目的是使用文件形式，把开发过程中各工作的负责人、开发进度以及需要的经费预算、所需的软件和硬件等都描述出来。后续的工作根据项目开发文档安排调配资源。项目开发文档是整个开发项目的资源描述文档，在编写的时候要从开发组织的实际情况出发，合理安排资源

12.3.3 软件需求说明书

软件需求说明书是软件开发组织于用户之间的接口文档，是整个软件开发的基础。软件需求说明书是软件供求双方对软件功能的一个具体描述文档，通常由软件开发组织编写。该文档包括了软件的开发任务、功能约定、开发周期等，用户根据软件开发组织的需求设计提出自己的意见，修改笮行纬勺钪瘴牡怠Ｐ枨笏得魇槎匀砑　　⑲橹　此岛茎匾　砑　　5.纳杓坪筒馑怨ぷ鞠际钦攵孕枨笪牡到　械摹

12.3.4 概要设计

概要设计文档说明了整个程序设计的框架和工作流程，是详细设计文档的基础。概要设计描述整个系统的处理流程、模块划分、接口设计以及出错处理等内容。概要设计的好坏决定了软件的优劣，通常是项目经理设计该文档，并且经过讨论后形成最终文档。

12.3.5 详细设计

详细设计是一个软件模块或者流程的具体描述文档。详细设计文档包括具体程序的功能描述、性能要求、输入输出格式、算法、存储分配等内容。对于简单的软件可以不做详细设计，在代码中做相应详细的注释即可。对应大型的软件，至少要在关键流程做详细设计，并且尽量保证详细设计文档与代码的对应关系便于维护管理。详细设计文档一般由程序员编写。

12.3.6 用户手册

前面介绍的几种文档都是软件开发组织使用的，文档结构规范、内容使用术语较多，便于开发组织内部交流。用户手册编写的目的是要使用非术语描述软件系统具有的功能和使用方法。用户在阅读使用手册后可以了解软件的功能和用途，并且通过说明书可以操作软件。

用户说明书通常包括软件的功能、运行环境、操作方法以及示例、常见出错问题及解答。用户手册要保证内容简洁，易于用户理解。

12.3.7 其他文档

在软件开发过程中还会产生其他一些文档，常见的有测试计划、测试报告、开发进度表和项目总结报告等。其中，测试是比较重要的部分，一个软件的优劣测试起到很大作用。测试是与开发并进的，包括单元测试、集成测试、功能测试和完整性测试等。测试的目的是发现软件中的缺陷，帮助改进软件的健壮性。12.3节详细介绍软件缺陷管理。

12.4 使用trac管理软件开发文档

trac是一个开源的项目管理软件，集成了Wiki（可译为百科全书）功能用于文档管理。该工具能与Subversion很好的结合，方便用户从文档中参考程序代码。

Wiki是互联网上的一种新型应用方式，功能类似于百科全书。Wiki提供一种在线编辑的方式，任何用户都可以编辑文档内容，Wiki可以把经过编辑和修改的内容合并成最终可阅读的文档。

12.4.1 安装trac

trac是一组Python语言编写的脚本，运行在Apache服务器上，后台使用SQL数据库。安装trac需要安装Apache服务器、MySQL数据库和Python语言解析器。

提示：12.3.2节在安装Bugzilla软件的时候也用到了Apache服务器。

Apache服务器是目前互联网上应用最广泛的HTTP服务器，提供Web页面服务。Python是一种面向对象的脚本语言。安装Apache服务器和Python语言解析器非常简单，使用apt软件包管理工具安装过程如下：

```
$ sudo apt-get install apache2 python
```

12.4.2 配置Trac基本设置

Trac安装完毕后，开始配置Trac。

（1）首先进入Apache的工作目录，建立trac的工作目录。

（2）修改好trac工作目录属性后，开始设置trac的工作脚本：

（3）改命令的作用是以www-data用户的身份运行trac-admin工具，为trac目录设置初始的工作脚本。键入命令后回车，出现一个文本的设置向导：

12.4.3 配置trac全局脚本

Trac安装完毕后，接下来配置Trac的全局脚本。

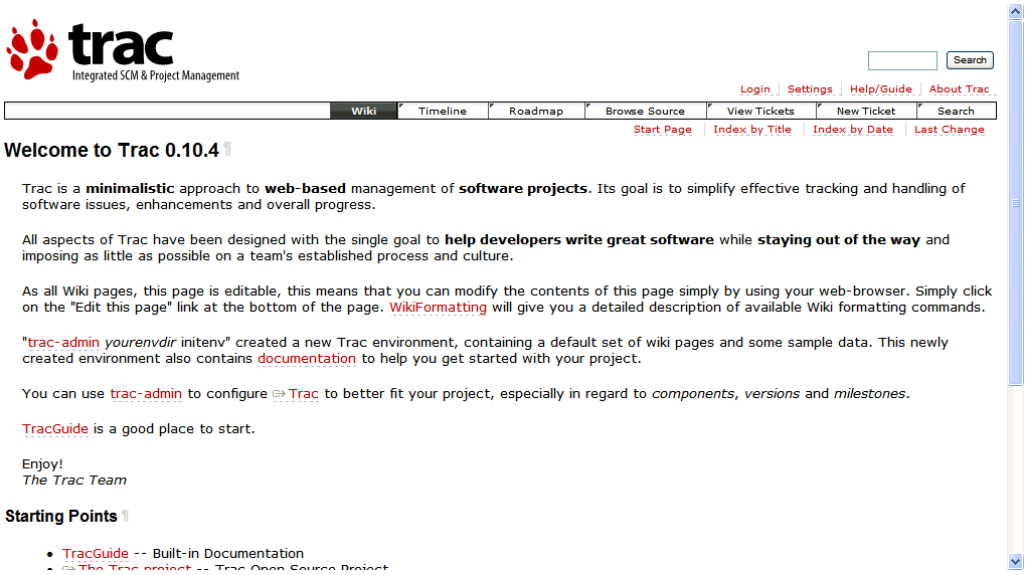
（1）进入Apache的cgi-bin目录，设置cgi-bin目录的所有者，然后建立到trac安装目录的连接。

（2）配置好脚本后，把Trac加入到Apache的默认站点。编辑/etc/apache2/sites-available/default文件。

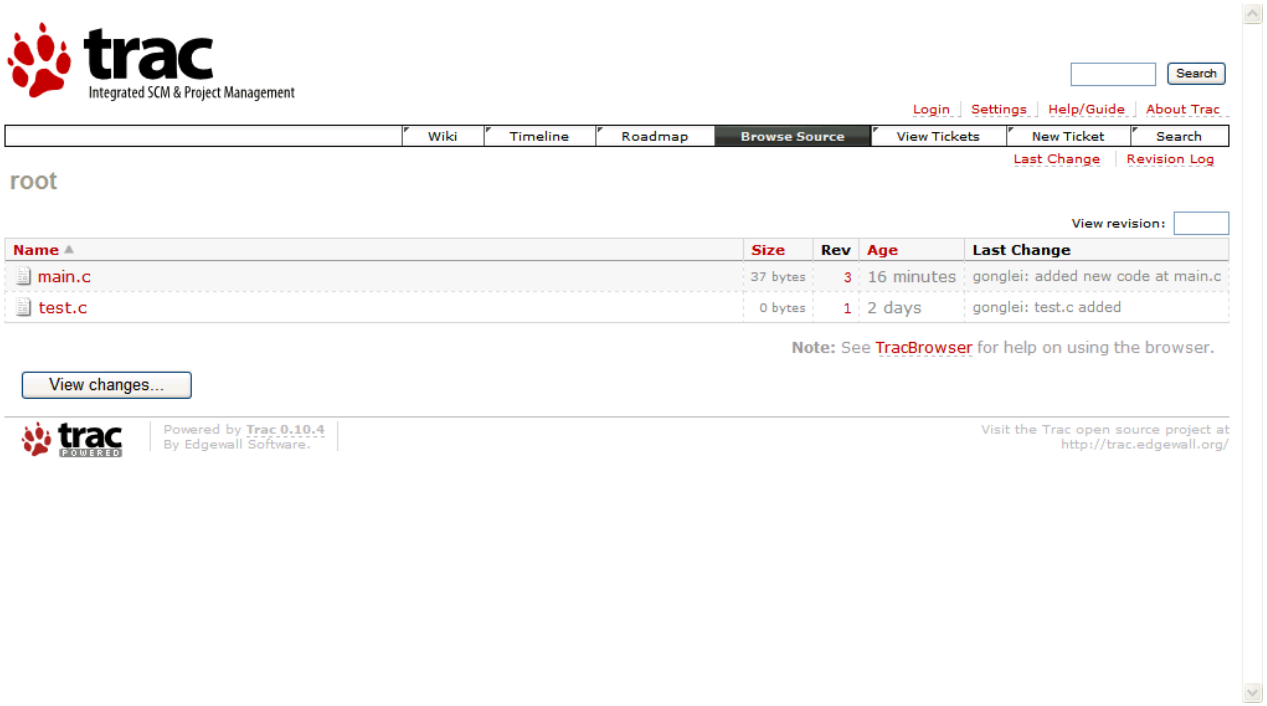
（3）接下来配置trac的密码文件：

12.4.4 设置trac的Web界面

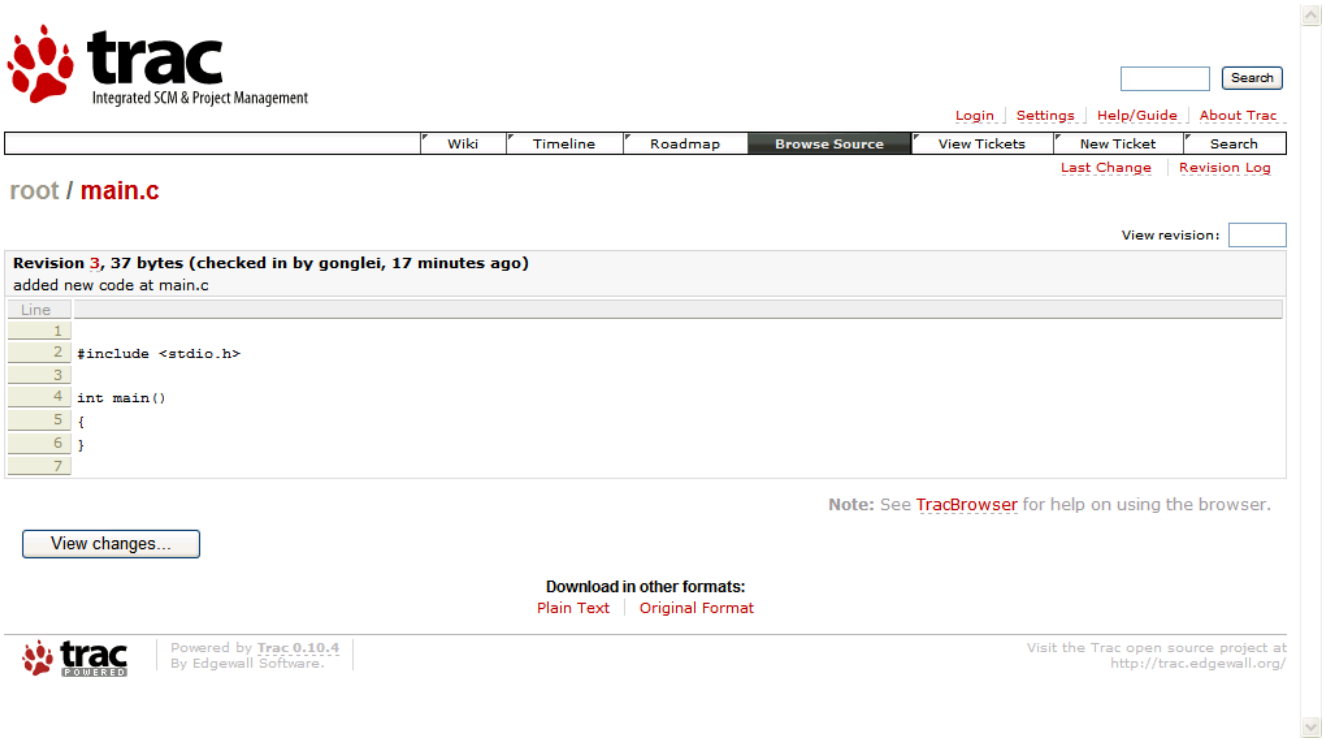
设置用户名和密码结束后，需要安装Trac的Web界面。trac使用组件完成各种功能，trac的组件被打包成egg文件格式，在安装egg文件之前需要安装python-setuptools。



12.4.4 设置trac的Web界面



12.4.4 设置trac的Web界面



12.5 Bug跟踪系统

软件开发过程中关键的问题是处理开发过程中出现的各种缺陷，也称作**Bug**。软件质量的高低很大程度上由**Bug**多少决定。**Bug**不仅影响软件的质量，而且直接决定了用户对软件的体验。对于嵌入式系统，一个很小的**Bug**可能导致很严重的后果。实际上，每个软件组织在开发过程中都对**Bug**管理投入很多，但是许多软件由于**Bug**太多无法维护最终导致失败。因此在开发过程中，对**Bug**管理要有足够重视。

12.5.1 Bug管理的概念和作用

Bug在英文中有“臭虫，虫子”的意思。软件开发领域习惯把程序中的错误比作“臭虫”，使用“**Bug**”一词表示软件中的缺陷、错误。软件开发过程中**Bug**一词应用范围非常广，从功能上的错误到死机、机器重启、程序访问异常等致命错误都称之为**Bug**。

没有不存在**Bug**的软件，但是一个优秀的软件应该保证**Bug**的数量降低到最少。软件中产生**Bug**的原因非常多，有的是由于程序员疏忽造成的，有的是逻辑上出了问题，有的甚至是在设计的时候留下的缺陷。一般来说，越是明显的**Bug**，修改难度越低；越是不容易产生的**Bug**，越有可能包含深层次的原因，反而不易调试和跟踪。

12.5.2 使用Bugzilla跟踪Bug

Bugzilla是一个开源的Bug管理系统，它提供了许多专业的Bug管理系统具备的功能。Bugzilla提供了Bug报告、查询、记录产生等功能。Bugzilla主要特点如下：

使用Web界面，无需安装客户端，方便使用

提供邮件自动通知功能

支持任意数量和类型的附件

自定义丰富的字段类型，便于描述Bug

使用MySQL数据库，方便数据迁移

12.5.2 使用Bugzilla跟踪Bug

Bugzilla

Bugzilla Version 2.22.1-debian2

Bugzilla Main Page

This is where we put in lots of nifty words explaining all about Bugzilla.

But it all boils down to a choice of:

Search existing bug reports
Enter a new bug report
Summary reports and charts

Login:
Password:
Login [Forgot my Password]

Open a new Bugzilla account

Add to Sidebar (requires a Mozilla browser like Mozilla Firefox)

Enter a bug # or some search terms:
Find [Help]

Actions: Home | New | Search | Find | Reports | Requests | New Account | Log In

12.6 小结

本节介绍了软件开发过程中的管理技术，包括代码的版本控制、文档管理和Bug管理。这三部分内容从不同角度对软件质量进行管理和控制，是软件开发过程中不可缺少的辅助手段。对于没有使用过版本控制和Bug管理系统的人来说，这些东西不仅是一个熟练的过程，更重要的是建立软件管理的意识，需要一个适应的过程。建议初学者在设计开发一个软件程序的时候养成书写文档和代码注释的好习惯。第13章讲解ARM体系结构。

第13章　ARM体系结构及开发实例

学习一个处理器的编程和使用，最主要的是了解处理器的体系结构，需要了解的部分通常包括处理器的编程模型、指令结构、内存管理等。本书第3章简要介绍了ARM处理器，本章详细讲解目前应用最广泛的ARM9体系结构，这部分知识比较抽象，偏重于理论，在本章最后给出了基于三星S3C2440A的ARM9处理器编程实例，帮助读者理解。主要内容如下：

ARM体系结构介绍

ARM体系结构的编程模型

ARM体系结构内存管理

S3C2440A常见的接口和控制器

S3C2440A处理器接口编程实例

13.1 ARM体系结构介绍

ARM处理器是从商业角度出发设计的RISC微处理器。ARM处理器继承了RISC体系结构，但是充分考虑到了实际应用。ARM设计主要关心设计的简单性，体现着硬件的组织和实现以及简洁的指令集。此外，ARM体系结构还引进了CISC体系结构的一些优点，因此获得较小的处理器功耗和较小的芯片面积。

13.1.1 ARM体系结构

在开发ARM芯片的时候，当时已经有许多机器都提出并使用了RISC技术。基于RISC技术的处理器有很多，但是只有少数的处理器完全使用RISC技术，许多基于RISC技术的处理器还吸收了其他体系结构的特点。

ARM是基于RISC技术的，但是在RISC技术的基础上又吸收了其他体系结构的优点，并且根据实际情况研究设计。ARM体系结构从Berkerley RISC体系结构发展而来，从商业角度出发，优化整合了一些处理器特征。ARM采用的结构特征包括：

Load/Store体系结构

固定的32位指令

3地址指令格式

ARM体系结构放弃的特征如下：

1. 寄存器窗口
2. 延迟转移
3. 所有指令单周期执行

13.1.2 ARM指令集介绍

每种处理器都包含了自己的指令集，ARM处理器属于精简指令集处理器，指令数量不多，本节介绍ARM指令集。

1. ARM指令集特点
2. 指令的后缀

13.2 编程模型

建造房子需要房模，同样的，编程象建造房子，不同的房子需要不同的模型，编程也需要编程模型，可以简单的理解编程模型就是一个模板，是一种解决问题的通用规则，有了编程模型，在遇到类似问题的时候就有了解决问题的方法。每个处理器体现结构都有自己的编程模型，ARM9也提供了一组编程模型，本节介绍其中的重点部分。

13.2.1 数据类型

ARM9微处理器支持三种数据类型，字节（8位）；半字（16位）；字（32位）。其中，半字需要2字节对齐，字需要4字节对齐。字节对齐的含义就是，在内存中存放数据的地址必须是某个数的倍数，以ARM9微处理器为例，半字需要2字节对齐，在内存中，如果存放一个16字节的数据，这个存放地址必须是2的倍数。同样的，4字节对齐要求存放数据的地址必须是4的倍数。

字节对齐是微处理器的硬性要求，主要是为了处理器的寻址方便，字节对齐是由加载器自动设置的，无需人为干预，但是程序员应当注意，在自定义数据结构的时候，尽量保持数据结构是2字节或4字节对齐的，否则加载器可能会把数据结构表示的数据自动加载到字节对齐的内存位置，这时如果程序不是很严谨，会导致直接访问数据出错的问题。

13.2.2 处理器模式

ARM处理器提供了7种工作模式，工作模式名称及含义如下：

用户模式（user）：

快速中断模式（fiq）：

外部中断模式（irq）：

特权模式（sve）：

数据访问终止模式（abt）：

未定义指令终止模式（und）：

系统模式（sys）：

13.2.3 寄存器

ARM处理器有31个通用寄存器和6个状态寄存器，共37个寄存器。37个寄存器按照处理器模式进行划分，有些寄存器被限定只能在特定模式下访问，有些寄存器可以在任何处理器模式下访问。其中，通用寄存器R0~R14、程序计数器PC以及特定的两个状态寄存器在任何处理器模式下都可以访问。

ARM处理器的37个寄存器按照功能被分成两个大类如下：

通用寄存器。

状态寄存器。

13.2.4 通用寄存器

ARM9的通用寄存器包括R0～R15，可以分为三类：

1. 未分组寄存器R0～R7
2. 分组寄存器R8～R14
3. 程序计数器PC（R15）

13.2.5 程序状态寄存器

寄存器R16用作CPSR（Current Program Status Register，当前程序状态寄存器）。CPSR 可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR（Saved Program Status Register，备份的程序状态寄存器）。SPSR寄存器在处理器进入某种异常模式的时候保存之前的CPSR寄存器值，用于异常处理后恢复CPSR状态。用户模式和系统模式不属于异常模式，因此没有SPSR寄存器。

13.2.6 异常处理

在ARM9体系机构中，提供了5种异常处理的模式：

1. FIQ（Fast Interrupt Request）
2. IRQ（Interrupt Request）
3. ABORT（中止）
4. Software Interruupt(软件中断)
5. Undefined Instruction(未定义指令)

13.2.7 内存和内存I/O映射

内存是计算机系统的重要资源，是程序运行时存储的区域，ARM9微处理器提供了内存控制器，提供了虚拟地址到物理地址映射，存储器访问权限控制，以及高速缓存支持。内存控制器通常也称为MMU（Memory Management Unit）内存管理单元。

Cache通过预测CPU即将要访问的内存地址（一般都是顺序的），预先读取大块内存供CPU访问，来减少后续的内存总线上的读写操作，以提高速度。然而，如果程序中长跳转的次数很多，Cache的命中率就会显著降低，随之而来，大量的替换操作发生，于是，过多的内存操作反而降低了程序的性能。

13.3 内存管理单元

许多嵌入式微处理器都由于没有 MMU而不支持虚拟内存。没有内存管理单元所带来的好处是简化了芯片设计，降低了产品成本。多数嵌入式设备外部存储设备和内存空间都十分有限，所以无需复杂的内存管理机制。没有 MMU 的管理，操作系统对内存空间是没有保护的，操作系统和应用程序访问的都是真实的内存物理地址。

13.3.1 内存管理介绍

虚拟存储器（Virtual Memory）的思想是允许程序占用的空间超过内存大小，把程序划分为大小固定的页，由操纵系统根据程序运行的位置把正在执行的页面调入内存，其他未使用的页面则保留在磁盘。如一个系统有16M字节的内存需要运行32M自己大小的程序，通过虚拟存储技术，操作系统把程序中需要执行的程序段装入内存，程序其他部分存放在磁盘。当程序运行超出装入在内存的部分后，操作系统自动从磁盘加载需要的部分。

虚拟存储技术中，操作系统通过内存页面管理内存空间。在一个操作系统中内存页面的大小是固定的。程序被划分成与内存页面大小相同的若干块，便于操作系统加载程序到内存。操作系统根据内存配置决定一次可以加载多少程序页面到内存。在一个实际的系统中，虚拟存储通常是由硬件（内存管理单元MMU）和操作系统配合完成的。

13.3.2 内存访问顺序

计算机可以访问的地址是有限制的，通常称作有效地址范围。地址范围的大小与计算机总线宽度有直接关系，如32位总线可以访问的地址范围是0x0~0xFFFFFFFF（4G）空间。程序可以访问的地址空间与总线支持的地址空间相同，但是在实际系统中，地址空间所有的地址都是有效的。通常把程序能访问的地址空间称作虚拟地址空间。

在计算机系统上，受到内存空间大小的限制，实际的地址空间远小于虚拟地址空间。可访问的物理内存空间称作物理地址空间。程序访问的虚拟地址空间的地址需要转换为物理地址空间的地址才能访问。

在没有使用虚拟存储技术以前，程序访问的空间就是物理地址空间，访问地址无需转换。使用虚拟存储技术后，程序访问虚拟地址空间的地址需要经过内存管理单元的地址转换机制变换后，得到对应的物理地址才能访问。而转换的过程对用户来说是不可见的。

13.3.3 地址翻译过程

一台使用16位地址位宽的计算机可以访问的地址范围是0x0000~0xFFFF，共有64K可地址，称作地址空间大小是64K。地址空间表示一个机器可以访问的实际物理内存大小。

使用虚拟存储的计算机系统可以提供超过内存地址空间大小的虚拟内存空间。如在64K地址空间的计算机上可以提供1M字节大小的虚拟地址空间。提供超过实际地址空间的虚拟空间后，计算机系统需要通过地址转换才能保证程序正确运行。

13.3.4 访问权限

现代的多用户多进程操作系统，需要MMU才能达到每个用户进程都拥有自己的独立的地址空间的目标。使用MMU，OS划分出一段地址区域，在这块地址区域中，每个进程看到的内容都不一定一样。例如MICROSOFT WINDOWS操作系统，地址4M-2G处划分为用户地址空间。进程A在地址0x400000映射了可执行文件。进程B同样在地址0X400000映射了可执行文件。如果A进程读地址0X400000读到的是A的可执行文件映射到RAM的内容。而进程B读取地址0X400000时则读到的是B的可执行文件映射到RAM的内容。这个时候就需要访问权限机制来处理不同进程访问同一地址内存的问题。

在ARM体系结构上使用Entry表来控制内存访问，在进行虚拟地址和实际地址映射的时候，通过查内存表映射到不同的物理内存地址上。

13.4 常见接口和控制器

本章1~4节介绍了ARM的体系结构，由于ARM是一种体系结构的规范，在实际的应用中，厂家在设计基于ARM的微控制器时候都是采用ARM的内核，在外面还根据需要扩展一些接口，不同厂家之间的接口可能不同，但是功能都是大同小异的。本节通过三星的S3C2440A处理器讲解ARM微处理器常见的接口和控制器。

13.4.1 GPIO接口

GPIO的英文全称是General-Purpose Input / Output Ports，中文意思是通用I/O端口。在嵌入式系统中，经常需要控制许多结构简单的外部设备或者电路，这些设备有的需要通过CPU控制，有的需要CPU提供输入信号。并且，许多设备活电路只只要求有开/关两种状态就够了，比如LED的亮和灭。对这些设备的控制，使用传统的串口或者并口就显得比较复杂，所以，在嵌入式微处理器上通常提供了一种“通用可编程I/O端口”，也就是GPIO。

一个GPIO端口至少需要两个寄存器，一个做控制用的“通用IO端口控制寄存器”，还有一个是存放数据的“通用I/O端口数据寄存器”。数据寄存器的每一位是和GPIO的硬件引脚对应的，而数据的传递方向是通过控制寄存器设置的，通过控制寄存器可以设置每一位引脚的数据流向。

13.4.2 中断控制器

ARM9可以识别两种类型的中断：正常中断请求（Normal Interrupt Request，IRQ）和快速中断请求（Fast Interrupt Request，FIQ），因此，S3C2440A的所有中断都可以归类为 IRQ 或 FIQ。S3C2440A的中断控制器对每一个中断源都有一个中断悬挂位（Interrupt Pending Bit）。

S3C2440A 用如下四个寄存器控制中断的产生和对中断进行处理：

中断优先级寄存器（Interrupt Priority Register）。

中断模式寄存器（Interrupt Mode Register）。

中断悬挂寄存器（Interrupt Pending Register）。

中断屏蔽寄存器（Interrupt Mask Register）。

13.4.3 RTC控制器

RTC的全称是实时时钟（Real Time Clock），当系统断电的时候，RTC控制器可以使用备份电池操作。RTC控制器可以使用STRB/LDRB/指令向CPU发送8比特位宽的BCD编码数据，这些数据包括年月日小时，分钟，秒等时间数据。RTC控制器使用外部32.768kHz的外部晶振工作，同时可以提供闹钟功能。S3C2440A微处理器的RTC控制器没有2000年问题，同时可以向实时操作系统内核提供微妙级别的时钟中断。

RTC控制器可以使用备份电池驱动，通过CPU的RTCVDD脚提供电源。当系统掉电的时候，CPU以及RTC控制器被阻塞，只有备份电池驱动晶振和BCD格式的时间计数器工作。RTC控制器可以在掉电或者带电状态下，在指定时间发出报警信号。在带电情况下，回产生INT_RTC中断，在掉电模式下，还会产生PMWKUP信号。

13.4.3 RTC控制器

寄存器名称	作用
实时钟控制器RTCCON （Real Time Clock Control Register）	RTC控制器的主要控制器寄存器，控制是否读写时间值的BCD寄存器
TICON（Tick Time Counter）	Tick值计数器
RTC报警控制器RTCALM （RTC Alarm Control Register）	设置报警打开以及报警时间

13.4.4 看门狗定时器

S3C2440A微处理器内部集成了硬件看门狗定时器，并且与复位电路相连，程序员通过设置看门狗定时器的控制寄存器就可以操作看门狗操作。S3C2440A有两个与看门狗相关的寄存器WTDAT和WTCNT，在打开看门狗定时器之前，需要向WTCNT寄存器写入看门狗定时器的初始计数值，之后打开看门狗定时器，当系统出现程序跑飞的情况后，看门狗会自动复位微处理器。

13.4.5 试验：使用GPIO点亮LED

S3C2440A微处理器提供了8组GPIO端口，共有130个引脚。这些GPIO端口均提供了数据寄存器，控制寄存器以及上拉寄存器，并且使用内存映射方式对寄存器编址，程序员通过访问对应寄存器的内存地址就可以操作对应的端口和引脚。

13.5 小结

本章讲述了ARM体系结构，任何一个处理器的体系结构都是不好理解的，大量的术语和复杂的结构给初学者造成了不小的学习困难。在学习中，应当先从整体把握，对ARM的体系结构有一个整体的认识，然后在具体每个部分学习，最好的学习方法是能通过ADS调试环境在开发板上调试，通过设置寄存器观察外部器件的变化，在ADS环境中自带了许多的例子可以参考，尤其是ARM启动时的设置，通过研究ARM的启动代码，会学习到很多的知识。第14章讲解嵌入式Linux Bootloader。

第14章　深入Bootloader

Bootloader一词在嵌入式系统中应用广泛，中文意思可以解释为“启动加载器”。顾名思义，Bootloader是一个在系统启动时工作的软件。由于启动时候涉及到硬件和软件的启动，Bootloader是一个涉及到硬件和软件衔接的重要系统软件。本章从Bootloader的原理出发，分析Bootloader的基本功能，同时介绍了常见的Bootloader系统软件，并且给出了U-Boot这款Bootloader在mini2440开发板的移植过程。本章主要内容如下：

Bootloader的基本知识和工作原理

常见的几种Bootloader介绍和对比

U-Boot的工程结构和工作流程

移植U-Boot到mini2440开发板

14.1 初识Bootloader

对于没有接触过嵌入式系统的人来说，**Bootloader**的功能虽然可以理解，但是缺乏一个直观的认识。本节以大家熟知的**PC**为例，介绍**PC**的启动工作流程，然后引入嵌入式**Bootloader**的概念，给读者一个感性的认识，帮助初学者揭开嵌入式系统**Bootloader**的面纱。

14.1.1 PC（个人电脑）上的Bootloader

不少初学者都会对标题有或多或少的疑惑，觉得PC机从哪里来的Bootloader。很少有人会说PC机有Bootloader。实际上PC的BIOS（主板上固化的一段程序，常说的“基本输入输出系统”）和硬盘或其他磁盘设备的引导记录在扮演着和嵌入式系统中Bootloader类似的作用，读者可以把这两部分的系统程序理解为PC的Bootloader。

Bootloader是系统加电后运行的第一段程序，一般来说，Bootloader为了保证整个系统的启动速度，要在很短的时间内运行。PC机的Bootloader由BIOS和MBR组成。其中，BIOS固化在PC机主板的一块内存内；MBR是PC机内硬盘主引导扇区（Master Boot Recorder）的缩写。

14.1.2 什么是嵌入式系统的Bootloader

PC机的体系结构相对固定，多数厂商采用相同的架构，甚至外部设备的连接方式都完全相同。并且，PC机有统一的设计规范，操作系统和开发人员不用为系统启动发愁，启动的工作都是由BIOS来完成的。不仅如此，PC的BIOS还为操作系统提供了访问底层硬件的中断调用。

嵌入式系统虽然硬件差异大，但是仍然有相同的规律可寻。在同一体系结构上，外部设备的连接方式、工作方式可能不同，但是CPU的指令、编程模型是相同的。由于和PC系统的差异，在嵌入式系统中，需要开发人员自己设计Bootloader。所幸的是，开发人员不用从零开始为每个系统编写代码，一些开源软件组织以及其他公司已经设计出了适合多种系统的Bootloader。这些Bootloader软件实际上是为嵌入式系统设计的一个相对通用的框架。开发人员只需要根据需求，按照不同体系结构的编程模型，以及硬件连接结构，设计与硬件相关的代码，省去了从头开发的繁琐流程。

14.1.3 嵌入式系统常见的Bootloader

Bootloader是嵌入式软件开发的第一个环节，它把嵌入式系统的软件和硬件紧密衔接在一起，对于一个嵌入式设备的后续开发至关重要。Bootloader初始化目标硬件，给嵌入式操作系统提供硬件资源信息，并且装载嵌入式操作系统。在嵌入式开发过程中Bootloader往往是难点，开源的Bootloader在设计思想上往往有一些相通之处。本节介绍两款常见的Bootloader供读者参考。

1. U-Boot系统加载器
2. ViVi系统加载器

14.2 U-Boot分析

Bootloader代码是嵌入式系统复位后进入操作系统前执行的一段代码。通过Bootloader的代码初始化处理器的各寄存器以及其他外部设备，建立存储器映射图以及初始化堆栈，为操作系统提供基本的运行环境。由于嵌入式系统的硬件的多样性，不可能有通用的Bootloader，因此需要根据具体硬件特点移植。本节以目前应用比较广泛的U-Boot为例讲解嵌入式系统Bootloader移植的方法。

14.2.1 获取U-Boot

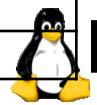
U-Boot的源代码可以从ftp://ftp.denx.de/pub/u-boot/获得。使用匿名用户身份登陆到U-Boot的FTP服务器后，进入pub/u-boot目录，该目录包含了U-Boot所有代码。本书使用U-Boot 1.1.6版本代码作为分析的样本。

14.2.2 U-Boot工程结构分析

学习一个软件，尤其是开源软件，首先应该从分析软件的工程结构开始。一个好的软件有良好的工程结构，对于读者学习和理解软件的架构以及工作流程都有很好的帮助。

U-Boot的源代码布局和Linux类似，使用了按照模块划分的结构，并且充分考虑了体系结构和跨平台问题。

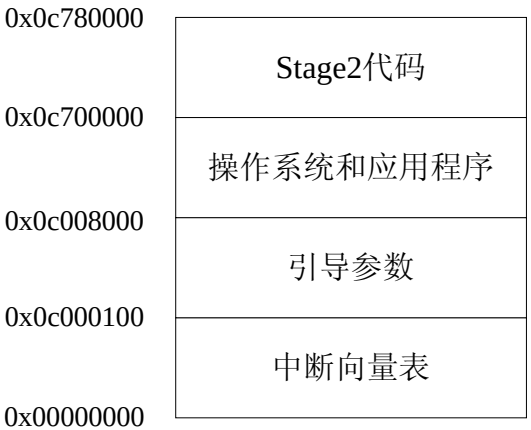
子目录名	作用
board	开发板相关的定义和结构
common	包含U-Boot用到的各种处理函数
cpu	各种不同类型的处理器相关代码
doc	U-Boot文档
drivers	常用外部设备驱动程序
examples	存放U-Boot开发代码样例
fs	文件系统有关的代码，包括cramfs、ext2、fat等常见文件系统
include	U-Boot用到的头文件
lib_arm	ARM体系结构有关的数据定义和操作
lib_generic	U-Boot通用的操作函数
net	常用的网络协议，包括bootp、rarp、arp、tftp等
post	上电自检相关代码
rtc	实时钟有关操作
tools	U-Boot有关的数据代码



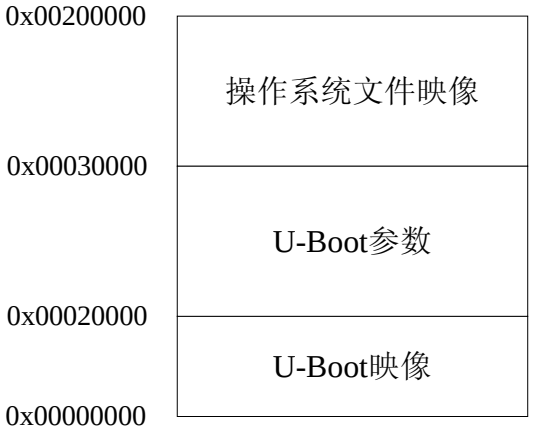
14.2.3 U-Boot总体工作流程

与大多数**Bootloader**类似，**U-Boot**的启动分成**stage1**和**stage2**两个阶段。**stage1**使用汇编语言编写，通常与**CPU**体系紧密相关，如处理器初始化和设备初始化代码等，该阶段在**start.S**文件中实现。

14.2.3 U-Boot总体工作流程



SDRAM映像

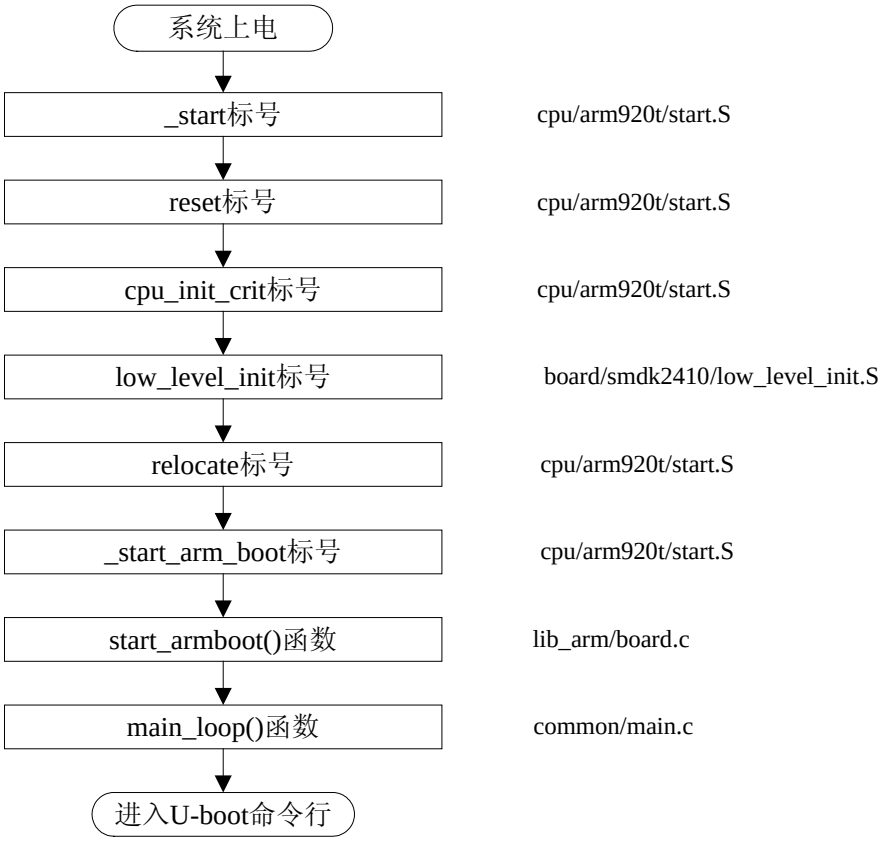


Flash存储器映像

14.3 U-Boot启动流程分析

U-Boot支持许多的处理器和开发板，主要是该软件有良好的架构，本节以使用ARM处理器的smdk2410开发板为例分析U-Boot的启动流程，在其他的处理器架构上U-Boot也执行类似的启动流程。

14.3 U-Boot启动流程分析



14.3.1 _start标号

在U-Boot工程中，每种处理器目录下都有一个start.S文件，该文件中有一个_start标号，是整个U-Boot代码的入口点。

14.3.2 reset标号

reset标号的代码在处理器启动的时候最先被执行。

14.3.3 cpu_init_crit标号

cpu_init_crit标号处的代码初始化ARM处理器关键的寄存器。

14.3.4 lowlevel_init标号

lowlevel_init标号位于board/smdk2410/lowlevel_init.S文件。

14.3.5 relocate标号

relocate部分的代码负责把**U-Boot Stage2**的代码从**Flash**存储器加载到内存。

14.3.6 start_armboot()函数

start_armboot()函数主要初始化ARM系统的硬件和环境变量，包括Flash存储器、FrameBuffer、网卡等，最后进入U-Boot应用程序主循环。

14.3.7 main_loop()函数

main_loop()函数做的都是与具体平台无关的工作，主要包括初始化启动次数限制机制、设置软件版本号、打印启动信息、解析命令等。

（1）设置启动次数有关参数。

（2）程序339~348行是Modem功能。

（3）接下来设置U-Boot的版本号，初始化命令自动完成功能等。

（4）在进入主循环之前，如果配置了启动延迟功能，需要等待用户从串口或者网络接口的输入。

（5）在各功能设置完毕后，程序454行进入一个for死循环，该循环不断是使用readline()函数（463行）从控制台（一般是串口）读取用户的输入，然后解析。

14.4 移植U-Boot到开发板

U-Boot虽然直至众多处理器和开发板，但是嵌入式系统的硬件是千差万别的，在使用U-Boot的时候，仍然需要针对自己的开发板做适当的修改。幸好U-Boot是一个结构设计合理的软件，在移植过程中严格按照U-Boot的工程结构移植容易取得成功。本节介绍如何移植U-Boot程序到ARM开发板。

14.4.1 U-Boot移植的一般步骤

从14.2节对U-Boot代码的分析可以看出， U-Boot移植工作主要分成处理器相关部分和开发板相关部分。由于U-Boot已经支持目前绝大多数处理器，因此处理器移植的工作相对较少，主要是修改一些配置。对于开发板部分的移植，需要参考硬件线路的外围器件的手册。U-Boot移植大致可以分析下面的步骤：

1. 检查U-Boot工程是否支持目标平台
2. 分析目标平台类似工程目录结构
3. 分析目标平台代码
4. 建立新的开发板平台目录
5. 对照手册修改平台差异部分代码
6. 调试新代码

14.4.2 移植U-Boot到目标开发板

移植U-Boot到新的目标平台会有许多问题。为了减少出错和工作量，在建立一个新的目标平台的时候可以直接拷贝现有类似平台的代码目录，然后在现有基础上修改。如移植到mini2440开发板，可以按照下面的步骤操作：

1. 建立新目标板工程目录
2. 向配置文件加入新开发板配置
3. 预编译新开发板的代码
4. 修改目标板配置
5. 编译新的配置并且下载执行

14.4.3 移植U-Boot常见问题

在移植U-Boot过程中会遇到很多问题，最主要的是无法一开始无法启动U-Boot。代码中很多地方设置有误都会导致无法启动，对于Stage1的代码来说，系统的出错信息是无法打印到串口或者其他设备的，此时可以使用JTag调试器调试目标开发板。

对于汇编编写的代码，一般都与系统硬件息息相关，在编写的时候需要非常仔细。最好准备好ARM体系结构手册和S3C2440A芯片手册，并且认真阅读编程模型相关的章节，对硬件的初始化流程要细心分析。

此外建议尽可能的把目标板外围硬件设备的初始化工作放在Stage2阶段，最好能使用C语言编写，避免使用汇编的调试周期长的问题，提高开发效率。

14.5 小结

本章介绍了Bootloader的概念已经常见的两款Bootloader软件，重点分析了U-Boot的结构和启动代码，并且给出了U-Boot移植的分析。Bootloader是一种与硬件联系紧密的软件，在学习的时候要结合硬件手册，对照代码分析提高学习的效率。第15章讲解Linux内核代码结构。

第15章　解析Linux内核

内核是操作系统的核心，通常说的Linux是指Linux操作系统的内核，是一组系统管理软件的集合。Linux内核是目前最流行的开源操作系统之一，由于其代码的高度开放性，越来越多的人参与到Linux内核的研究和开发中。Linux内核的功能也在不断提高，性能在不断改进。操作系统内核是软件开发领域比较深的技术点，需要结合软硬件知识才能深入理解。本章由浅入深讲解Linux内核，带领读者进入嵌入式开发比较深入的领域，主要内容如下：

如何获取Linux内核代码

Linux内核功能解析

Linux内核代码布局

Linux内核镜像结构

15.1 基本知识

Linux内核是Linux操作系统不可缺少的组成部分，但是内核本身不是操作系统。许多Linux操作系统发行商如RedHat、Debian等都采用Linux内核，然后加入用户需要的工具软件和程序库，最终构成一个完整的操作系统。嵌入式Linux系统是运行在嵌入式硬件系统上的Linux操作系统，每个嵌入式Linux系统都包括了必要的工具软件和程序库。

15.1.1 什么是Linux内核

内核是操作系统的核心部分，为应用程序提供安全访问硬件资源的功能。直接操作计算机硬件是很复杂的，内核通过硬件抽象的方法屏蔽了硬件的复杂性和多样性。通过硬件抽象的方法，内核向应用程序提供了统一和简洁的接口，应用程序设计复杂程度降低。实际上，内核可以被看做是一个系统资源管理器，内核管理计算机系统中所有的软件和硬件资源。

应用程序可以直接运行在计算机硬件上而无需内核的支持，从这个角度看，内核不是必要的。在早期的计算机系统中，由于系统资源的局限，通常采用直接在硬件上运行应用程序的办法。运行应用程序需要一些辅助程序，如程序加载器、调试器等。随着计算机性能的不断提高，硬件和软件源都变得复杂，需要一个统一管理的程序，操作系统的概念也逐渐建立起来。

15.1.2 Linux内核版本

Linux内核版本号采用两个“.”分割的三个数字来标示，形式为“X.Y.Z”。其中，X是主要版本号，Y是次要版本号，Z代表补丁版本号。奇数代表不稳定的版本；偶数代表稳定的版本。“稳定”和“不稳定”是相对的，如Linux内核1.1.0相对于1.0.0来说是“不稳定”版本，但是与1.1.1对比是“稳定”版本。在Linux内核开发过程中，“不稳定”版本通常是在原有版本基础上增加了新的功能或者新的特性。

15.1.3 如何获取Linux内核代码

在PC上，一般的Linux发行版都提供了内核代码。嵌入式系统没有固定的发行版，需要用户自己获取内核代码。Linux内核代码的官方站点是http://www.kernel.org，该站点提供了2.4和2.6所有版本的代码和补丁，用户可以打开该地址找到和自己所在物理位置就进的站点，下载自己需要的内核版本代码。高版本Linux内核代码文件比较大，对于国内的用户推荐使用ftp方式下载，或者使用断点续传工具下载，具体情况可根据读者自身的网络情况选择。

下载Linux内核代码后，会得到一个类似“linux-2.6.xx.tar.gz”或者“linux-2.6.xx-tar.bz2”形式的压缩文件，“xx”代表版本号。在Linux系统上，通常把这个文件存放在/usr/src目录下，便于以后使用。

15.1.4 编译内核

学习Linux内核最好的开始是编译一次Linux内核代码，通过配置Linux内核可以对内核代码有一个初步的了解。本节介绍一下在PC机上如何编译生成2.6版本的内核目标文件，在本书第20章移植Linux部分会讲解如何交叉编译用于ARM体系结构的Linux内核。

与2.4版本相比，2.6版本内核代码编译相对较容易。内核编译主要分成配置和编译两部分，其中配置是关键，许多问题都是出在配置环节。Linux内核编译配置提供多种方式：

make config：

make menuconfig：

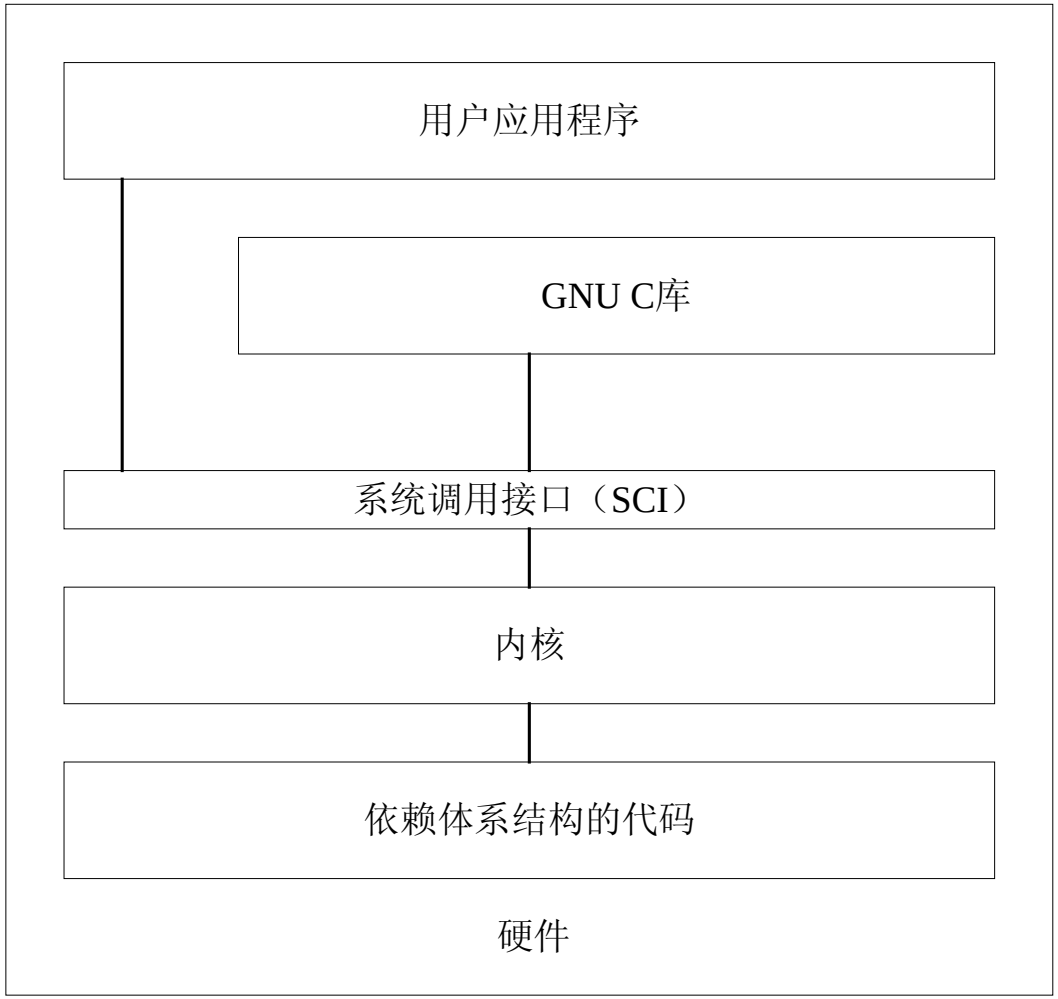
make xconfig：

make oldconfig：

15.2 Linux内核的子系统

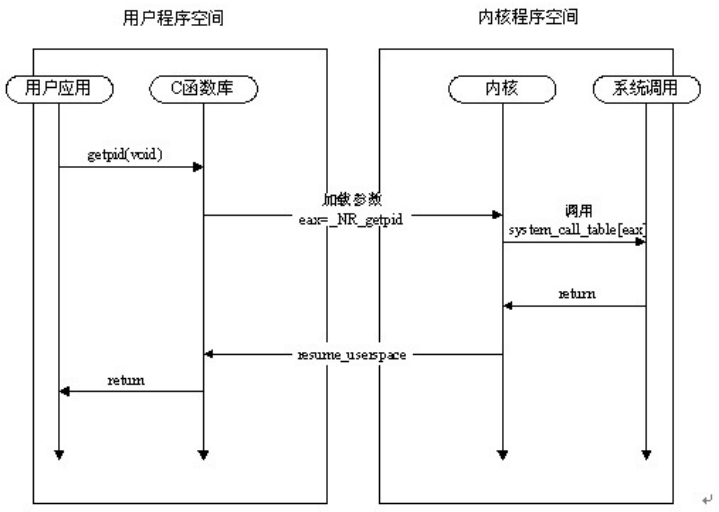
内核是操作系统的核心。Linux内核提供很多基本功能，如虚拟内存、多任务、共享库、需求加载、共享写时拷贝（Copy-On-Write）以及网络功能等。增加各种不同功能导致内核代码不断增加。Linux内核把不同功能分成不同的子系统的方法，通过一种整体的结构把各种功能集合在一起，提高了工作效率。同时还提供动态加载模块的方式，为动态修改内核功能提供了灵活性。

15.2.1 系统调用接口



15.2.1 系统调用接口

系统调用是Linux内核提供的，用户空间无法直接使用系统调用。在用户进程使用系统调用必须跨越应用程序和内核的界限。Linux内核向用户提供了统一的系统调用接口，但是在不同处理器上系统调用的方法各不相同。Linux内核提供了大量的系统调用，本书从系统调用的基本原理出发讲解Linux系统调用的方法。



15.2.2 进程管理子系统

当一个用户进程被加载后，会进入就绪态，被加入到就绪态队列，CPU时间被轮转到就绪态队列后，切换到进程的代码，进程被执行，当进程的时间片到了以后被换出。如果进程发生I/O操作也会提前被换出，并且存放到等待队列，当I/O请求返回后，进程又被放入就绪队列。

Linux系统对进程队列的管理设计了若干不同的方法，主要的目的是提高进程调度的稳定性。

15.2.3 内存管理子系统

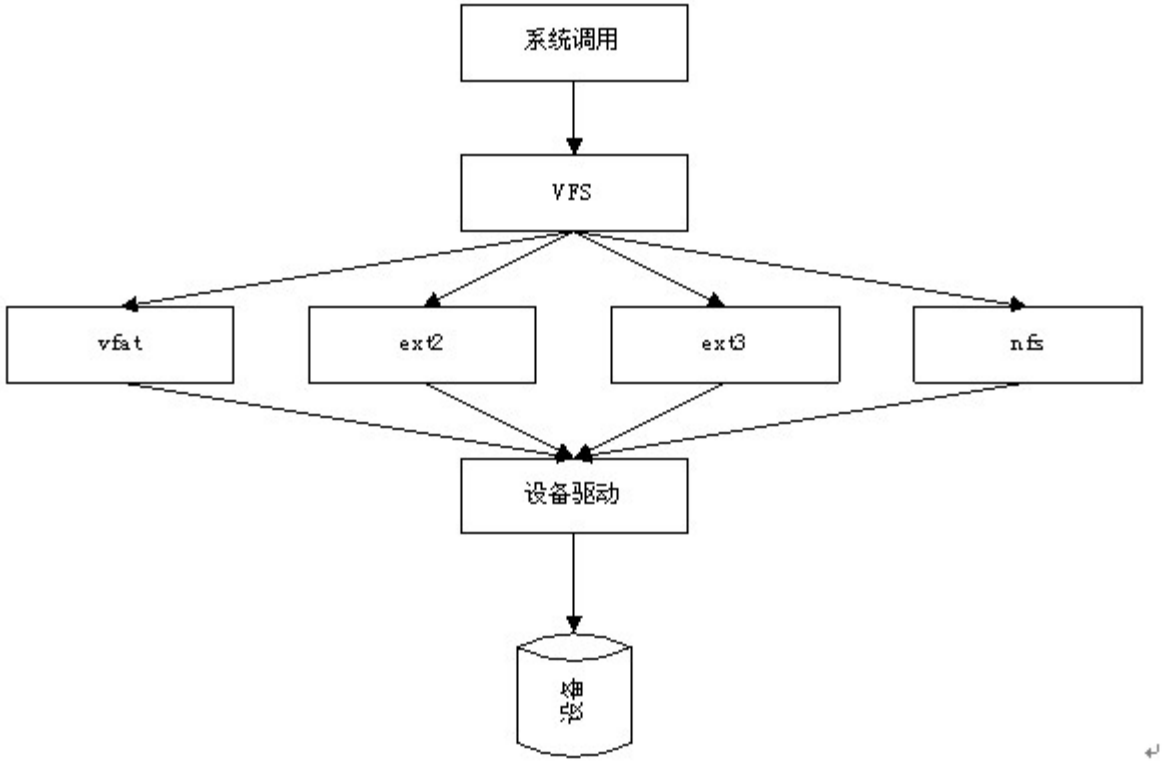
内存是计算机的重要资源，也是内核的的重要部分。使用虚拟内存技术的计算机，内存管理的硬件按照分页方式管理内存。分页方式是把计算机系统的物理内存按照相同大小等分，每个内存分片称作内存页，通常内存页大小是4KB。Linux内核的内存管理子系统管理虚拟内存与物理内存之间的映射关系，以及系统可用内存空间。

内存管理要管理的不仅是4KB缓冲区。Linux提供了对4KB缓冲区的抽象，例如slab分配器。这种内存管理模式使用4KB缓冲区为基数，然后从中分配结构，并跟踪内存页使用情况，比如哪些内存页是满的，哪些页面没有完全使用，哪些页面为空。这样就允许该模式根据系统需要来动态调整内存使用。

15.2.4 虚拟文件系统

虚拟文件系统，即VFS（Virtual File System）是Linux内核中的一个软件抽象层。它通过一些数据结构及其方法向实际的文件系统如ext2，vfat等提供接口机制。通过使用同一套文件 I/O 系统调用即可对Linux中的任意文件进行操作而无需考虑其所在的具体文件系统格式；更进一步，文件操作可以在不同文件系统之间进行。

15.2.4 虚拟文件系统

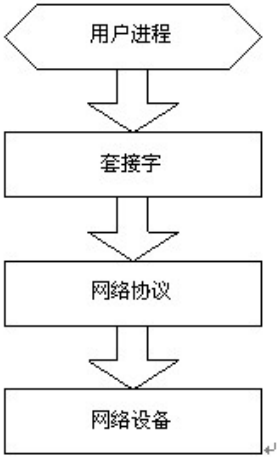


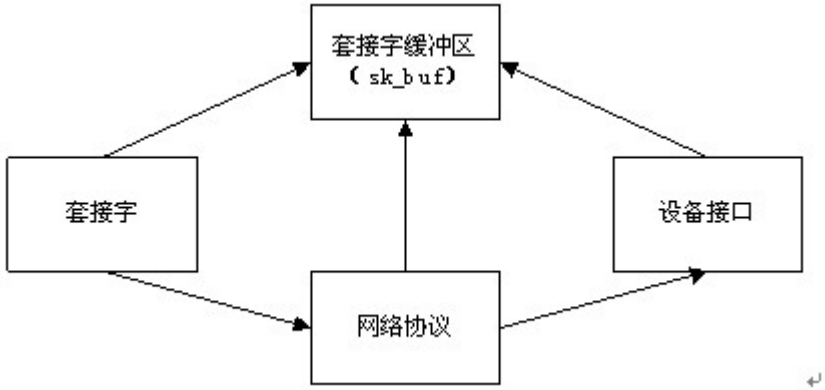
15.2.5 网络堆栈

第九章介绍了Linux系统下如何编写网络应用程序，使用socket通过TCP/IP协议与其他机器通信，和前面介绍的内核子系统相似，socket相关的函数也是通过内核的子系统完成的，担当这部分任务的是内核的网络子系统，一些资料里也把这部分代码称为“网络堆栈”。

Linux内核提供了优秀的网络处理能力和功能，这与网络堆栈代码的设计思想是分不开的，Linux的网络堆栈部分沿袭了传统的层次结构，网络数据从用户进程到达实际的网络设备需要四个层次。

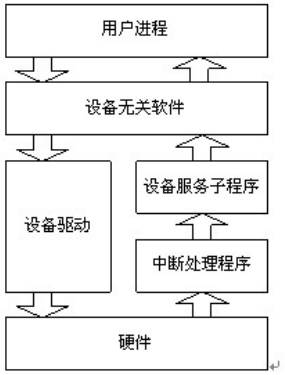
15.2.5 网络堆栈





15.2.6 设备驱动

Linux内核对设备按照主设备号和从设备号的方法访问，主设备号描述控制设备的驱动程序，从设备号区分同一个驱动程序的不同设备。也就是说，主设备号和设备驱动程序对应，代表某一类型的设备，从设备号和具体设备对应，代表同一类的设备编号。如使用IDE接口的两个硬盘，主设备号都相同，但是从设备号不同。Linux提供了mknod命令创建设备驱动程序的描述文件，后面设备驱动相关章节具体讲解。Linux内核这种主从设备号的分类方法可以很好的管理设备。



15.2.7 依赖体系结构的代码

Linux内核支持众多体系结构，内核把与设备无关的代码放在arch目录，对应的头文件放在include/asm-<体系名称>目录下。这样的划分代码结构清晰，同时提高了代码的复用率。在arch目录里，每个子目录对应一种体系结构，存放这种体系结构对应的代码，如果代码较多会单独建立一个目录，例如arch/arm目录下，有一个kernel目录，存放的是kernel目录中在arm体系结构上特有的函数或者实现方法；在arch/i386目录存放了Intel X86体系结构的代码，不仅有kernel目录，而且还有多个目录，例如mm目录包含了x86体系上内存管理的实现方法，math-emu包含了x86体系上浮点数模拟的实现等。读者在阅读内核代码的时候可以从一个体系结构代码入手，对不同体系结构移植代码的主要工作是arch里面的代码。

15.3 Linux内核代码的工程结构

随着Linux内核功能的不断增加，内核代码也在飞速增长，目前2.6版本的内核代码早已达到数百万行。如此庞大的代码量，不仅给学习带来困难，对代码的维护也是一个不小的挑战，幸好Linux内核开发人员早就考虑到了这一点，使得Linux内核代码组织有序，本节重点讲述Linux内核代码的结构。

15.3.1 源代码目录布局

15.1.3节讲述了如何获取到Linux内核代码，通常会存放在/usr/src目录下，如果是2.6版本的内核解压后会得到例如’linux-2.6.xx’类型的目录，这个目录下存放的就是Linux内核代码。进入内核代码目录，查看文件列表，会看到许多的目录和文件，如果读者的系统有tree这个命令或者脚本，可以查看到Linux内核代码的文件数，那会是一个很庞大的结构。好在Linux内核代码的工程组织是很好的，对于不同版本的内核，在工程组织上是基本一致的，有的仅是功能上的差别。

15.3.2 几个重要的Linux内核文件

当用户编译一个Linux内核代码后，会生成几个文件：vmlinuz，initrd.img，以及System.map，如果读者配置过grub引导管理器程序，会在/boot目录下看到这几个文件。

1. vmlinuz文件
2. initrd文件
3. System.map文件

15.4 内核编译系统

Linux内核代码的复杂，需要一个强大的工程管理系统，幸好GNU提供了Makefile机制，此外，内核的开发者们还提供了KBuild机制。通过Makefile和KBuild的结合，可以出色的管理Linux内核代码。Linux内核的编译系统和代码结构是紧密联系的，了解内核编译系统对分析内核代码和编译内核都有帮助作用。

15.4.1 内核编译系统基本架构

Linux内核编译系统有5种类型的文件。

文件类型	作用
Makefile	顶层Makefile文件
.config	内核配置文件
arch/$(ARCH)/Makefile	机器体系Makefile文件
scripts/Makefile.*	所有内核Makefiles共用规则
kbuild Makefiles	其它Makefile文件

15.4.2 内核顶层Makefile分析

编译内核代码的时候，顶层**Makefile**文件在开始编译子目录下的代码之前，设置编译环境和需要用到的变量。顶层**Makefile**文件包含通用部分，**arch/$(ARCH) /Makefile**包含该体系架构所需的设置。因此**arch/$(ARCH)/Makefile**会设置一些变量和少量的目标。

1. 设置变量
2. 增加预设置项
3. 目录表
4. 引导映像
5. 编译非内核目标
6. 编译引导映像命令
7. 定制编译命令
8. 预处理连接脚本

15.4.3 内核编译文件分析

Linux内核代码使用KBuild作为Makefile的基础架构。Kbuild定义了若干的内置变量，本节介绍Kbuild的主要内置变量和常用方法。

1. 目标定义
2. 内嵌对象 - obj-y
3. 可加载模块 - obj-m
4. 导出符号目标
5. 库文件 - lib-y
6. 目录递归
7. 编译标记
8. 依赖关系
9. 特殊规则
10. $(CC)支持功能

15.4.4 目标文件清除机制

"make clean"命令删除在编译内核生成的大部分文件如主机程序。列举在$(hostprogs-y)、$(hostprogs-m)、$(always)、$(extra-y)和$(targets)中目标文件都将被删除。 代码目录数中的"*.[oas]"、"*.ko"文件和一些由编译系统产生的附加文件也将被删除。附加文件可以使用$(clean-files)进行定义。例如：

```
#drivers/pci/Makefile
clean-files := devlist.h classlist.h
```

15.4.5 编译辅助程序

内核编译系统支持在编译（compliation）阶段编译主机可执行程序。为了使用主机程序需要两个步骤：第一个步骤使用hostprogs-y变量告 诉内核编译系统有主机程序可用。第二步给主机程序添加潜在的依赖关系。有两种方法，在规则中增加依赖关系或使用$(always)变量。具体描述如下。

1. 简单辅助程序
2. 组合辅助程序
3. 定义共享库
4. C++语言使用方法
5. 辅助程序编译控制选项
6. 何时建立辅助程序
7. 使用hostprogs-$(CONFIG_FOO)

15.4.6 KBuild变量

KBuild内置了一些变量供顶层Makefile使用，顶层Makefile文件导出下面这些变量：

VERSION, PATCHLEVEL, SUBLEVEL, EXTRAVERSION

这几个变量定义了当前内核版本号。很少体系体系**Makefiles**文件直接使用他们，常用**$(KERNELRELEASE)**代替。

15.5 小结

Linux内核代码非常庞大复杂，对任何人来说学习都是一个不小的挑战，本章讲解了Linux内核的工程结构和代码结构，从嵌入式系统开发的角度来说，大多数没有必要一行一行的研究内核代码，开发人员需要了解内核的机构和工作流程，以及常见的开发方法即可，学习内核最基本的技能是编译内核，在此基础上通过学习驱动开发和内核移植。第16章讲解嵌入式Linux内核启动过程。

第16章　嵌入式Linux启动流程

在多数计算机上，从Linux开机到进入系统的命令行或者图形界面时间并不长。计算机在背后做了什么工作，会展现出一个功能强大的系统，本章分析Linux系统启动流程。学习和掌握Linux启动的流程对了解Linux内核工作流程有很大帮助。Linux系统初始化可以分成两大部分：内核初始化和系统初始化。本章分析从打开电源开关到进入用户界面Linux系统的工作，主要内容如下：

Linux内核初始化概览

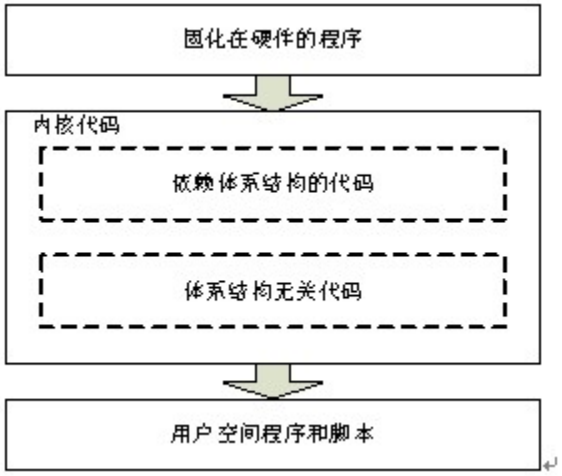
进入内核前的工作

内核初始化

如何进入用户空间

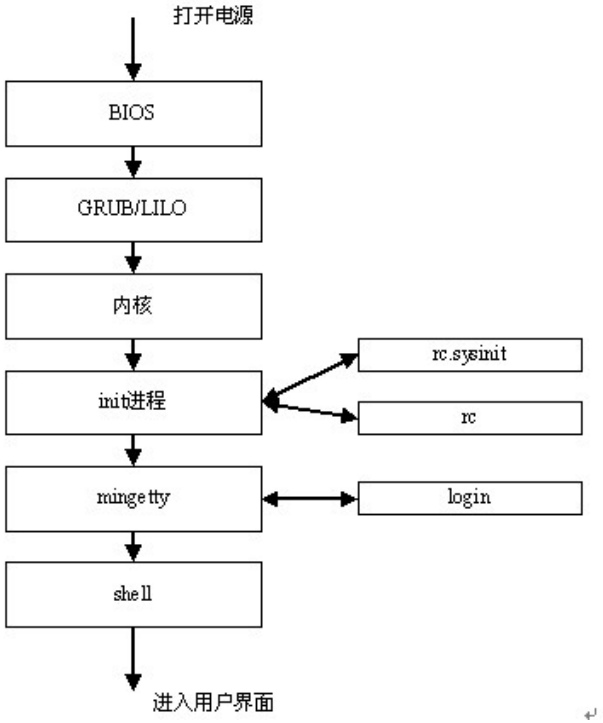
16.1 Linux内核初始化流程

从前面讲解的知识知道，操作系统是用户应用和计算机硬件之间的桥梁。操作系统管理整个系统的所有软硬件资源，并且向用户应用程序提供接口。在操作系统初始化的时候，系统内核检测计算机硬件，加载驱动并且设置软件环境，本节详细讲解**Linux**内核初始化所做的工作。



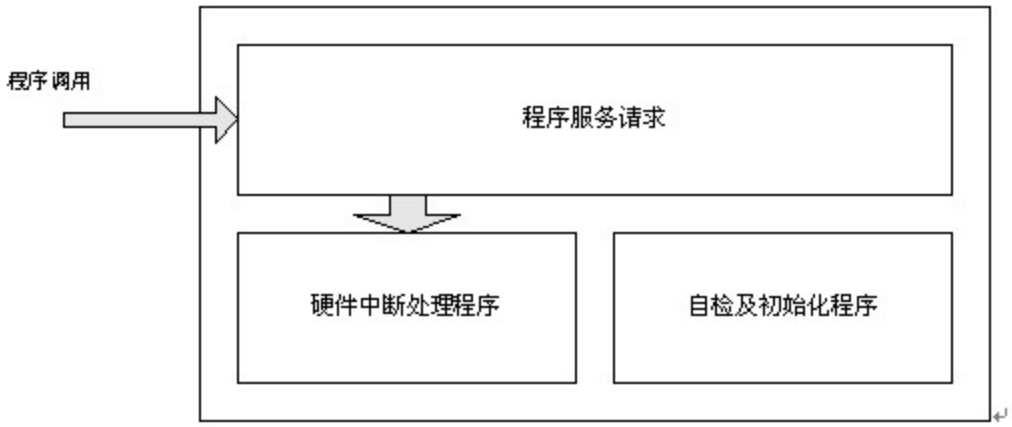
16.2 PC机的初始化流程

先给出一个**PC**机的初始化流程图。



16.2.1 PC机BIOS功能和作用

图16-3是PC机BIOS的功能结构。



16.2.2 硬盘的数据结构

PC机最常见的外部存储设备是硬盘驱动器。硬盘可以存储大量的数据，并且具有断电信息不丢失的特点。硬盘上的数据组织格式随不同操作系统不完全相同。无论什么系统，对硬盘的数据组织方式有何不同，都包含了一个引导记录的数据结构。引导记录（英文全称Main Boot Record，简称MBR）是位于硬盘0磁道0柱面的第一个扇区。一个扇区有512字节，MBR占用了开始的446字节。

16.2.2 硬盘的数据结构

16.2.3 完整的初始化流程

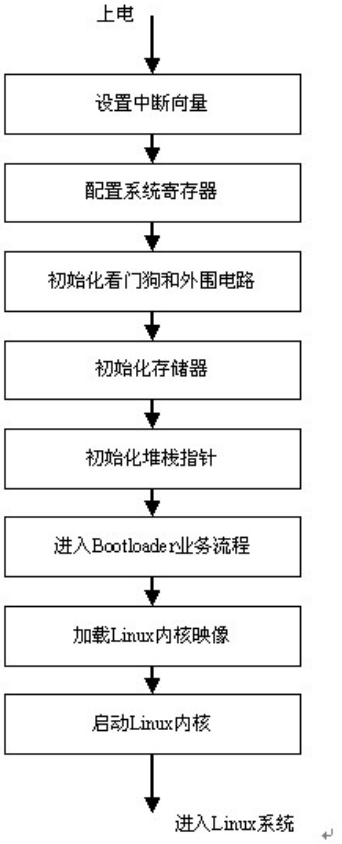
在弄清楚PC机BIOS和硬盘引导程序的结构和作用后，说一下PC机的启动流程。PC机加电或者Reset后硬件系统会复位，复位后寄存器CS=0xFFFF，寄存器IP=0x0000。CPU从FFFF:0000H处执行指令，这个地址只有一条JMP（跳转）指令，跳转到系统自检程序，也就是进入了BIOS程序存放的位置。执行自检程序通过后，BIOS根据配置把软盘或者硬盘（光盘也是同样道理）的MBR扇区读入系统0000:7C00H处，执行MBR的代码。

MBR的代码通常由操作系统修改，也可以由其他程序（例如GRUB引导器）修改。如果机器安装了GRUB引导软件，执行MBR的代码会启动GRUB引导软件。系统的控制权交由GRUB引导软件处理，GRUB根据分区的配置信息，找到硬盘对应分区上Linux内核文件并且加载到内存，然后跳转到内核代码位置，最后把系统控制权交给Linux内核。

16.3 嵌入式系统的初始化

嵌入式系统的多样性和复杂性，一般不像PC机那样配置BIOS，系统中也没有像BIOS那样的固件。用于启动的代码必须由用户完成，通常称这部分代码为Bootloader程序，整个系统的启动就由它完成。Bootloader初始化硬件设备、建立内存空间的映射，将系统的软硬件环境设定在一个合适的状态，为加载操作系统内核和应用程序准备一个正确的环境。Bootloader依赖实际硬件环境，通常不存在一个通用的标准。对于不同的嵌入式系统，Bootloader程序内容也不相同。本书以ARM处理器为例介绍嵌入式系统的初始化。

16.3 嵌入式系统的初始化



16.4 Linux内核初始化

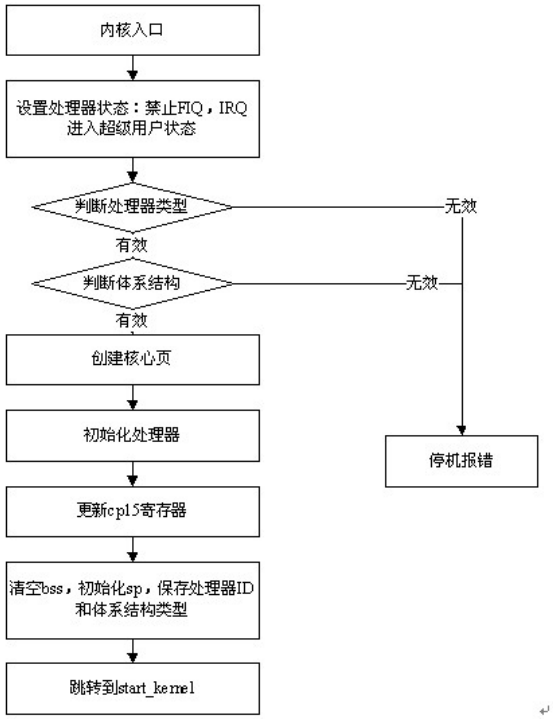
Linux内核在不同处理器体系结构上启动代码不全相同，但是启动的流程基本一致，本书根据嵌入式开发的需要从ARM核分析Linux内核初始化过程。

16.4.1 解压缩内核映像

对于大多数嵌入式应用，由于存储器空间的限制，大多数编译后的内核映像都是压缩存放，所以进入内核的第一步首先是解压缩内核映像。

16.4.2 进入内核代码

接下来就进入了真正的内核代码。在有MMU的处理器上，系统会使用虚拟地址，通过MMU指向实际物理地址。



16.5 启动init内核进程

在start_kernel()函数最后调用了rest_init()函数，此函数用来创建内核init进程，这也是内核态的最后的工作。代码如下：

```
static void rest_init(void)
{
 kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
 unlock_kernel();
 current->need_resched = 1;
  cpu_idle();
}
```

16.6 根文件系统初始化

Linux内核启动完毕后，首先就是要创建根文件系统，用户空间所有的操作都依赖根文件系统。本节介绍根文件系统的结构，并结合代码分析根文件系统的初始化过程。

16.6.1 根文件系统介绍

在内核代码启动完之后，进入文件系统初始化的阶段，Linux需要加载根文件系统。Linux的根文件系统可以分两类：虚拟根文件系统和真实根文件系统。Linux内核的发展趋势是把更多的功能放在用户空间完成，可以保持内核的精简。虚拟根文件系统也是各Linux发行厂商采用的一种方式，可以把初始化的工作在虚拟的根文件系统完成，最后再切换到真实的文件系统。

1. 传统的initrd根文件系统
2. initramfs根文件系统
3. 根文件系统的挂载

16.6.2 挂载虚拟文件系统

挂载文件系统在**kernel_init()**函数实现，本节重点分析该函数。

1．基本参数初始化

2．创建系统第一个进程

3．挂载根文件系统

16.7 内核交出权限

Linux内核通过sys_fork()函数，之后在调用sys_execve()函数创建一个新的进程。系统启动后，核心态创建名为init的第一个用户进程。实现这种逆向迁移，Linux内核并不调用用户层代码。实现逆向迁移通常做法是在用户进程的核心栈压入用户态的SS，ESP，EFLAGS，CS，EIP等寄存器伪装成用户进程，然后通过trap进入核心态，最后通过 iret指令返回用户态。

16.8 init进程

也是一个链接，具体指向哪个程序可以由用户配置。

init进程的主要任务是按照inittab配置文件提供的信息创建进程，由于进行系统初始化的进程都是由init进程创建的，所以init进程也称为系统初始化进程。

inittab配置文件的格式是没一行一个配置项，有如下结构：

id:rstate:action:process

每项有四个字段，字段之间用“:”分割，如果某个字段没有设置，直接留空。下面解释各字段含义：

1. **id字段**
2. **rstate字段**
3. **action字段**
4. **process字段**

16.9 初始化RAM Disk

现代计算机的内存容量越来越大，并且价格也不断下降。内存具备了相对外存储器访问速度快，价格低廉的优势。Linux系统支持一项功能，可以指定一块内存区域作为文件分区。用户可以像使用普通文件分区一样使用内存。本节介绍这种内存管理技术。

16.9.1 RAM Disk介绍

Linux系统提供一种特殊的功能-“初始化内存盘”，英文名Initial Ram Disk。RAM Disk技术与压缩映像技术结合，使用该技术后Linux系统可以从容量较小的内存盘启动。使用系统内存的一部分作为根文件系统，可以不使用交换分区。换句话说，使用内存盘技术可以把Linux系统完全嵌入内存，不依赖其他外部存储设备。

使用RAM Disk技术，系统不工作在硬盘活其他外部设备上，消除了读写延迟；根文件系统和操作完全运行在CPU/RAM环境下，系统速度和可靠性方面比较好；此外，根文件系统也不会因为非法关机导致被破坏。

RAM Disk唯一的一个缺点是对内存有一定的要求，要获得较好的性能，内存容量是不能太小的，目前PC机的内存一般都很大，在内存运行根文件系统没有问题。嵌入式系统如果配备了较大的内存也可以考虑使用RAM Disk技术。

16.9.2 如何使用RAM Disk

RAM Disk也称作RAM盘，作用是在内存中使用一块内存区域虚拟出一个硬盘。使用RAM Disk需要在编译内核的时候，在“Block Device”选项中选择“BlockDevice”菜单项。在设置“Block Disk”的时候，需要设置RAM Disk的参数。修改设置后，需要重新编译Linux内核才能使用。编译带有RAM Disk选项的内核。

16.9.3 实例：使用RAM Disk作为根文件系统

本实例以创建一个Apache网络服务器为例，展示如何创建从当前存在的Linux系统创建基于RAM Disk的根文件系统。创建一个Apache网络服务器，只需要把httpd配置文件服务程序放入根文件系统映像，并且加入启动文件即可。下面是具体的操作过程。

（1）首先在Linux创建/minilinux目录，以后以此目录创建根文件系统。

（2）接下来制作RAM Disk映像，启动计算机的时候设置RAM Disk大小至少大于/minilinux目录文件大小。

（3）制作RAM Disk完毕后，修改/mnt/ram/etc目录下的fstab文件。

16.10 小结

Linux的启动过程非常复杂，不仅在内核状态包含处理器模式切换，系统权限的获取，设置系统中断，初始化硬件和软件环境等，在用户态也包含了功能强大而复杂的脚本，读者应该认真学习，通过实践理解Linux启动过程的各个步骤，为学习Linux系统打下良好的基础。第17章讲解Linux文件系统。

第17章　Linux文件系统

Linux系统的一个重要特点就是“一切都是文件”，从这个特点看出文件的重要性。与其他系统一样，文件的管理是通过文件系统实现的。Linux的文件系统不仅具备普通的文件管理功能，还有许多特殊的功能，本章从文件系统的基本概念入手讲解Linux文件系统，主要内容如下：

Linux系统如何管理文件

文件系统的工作原理

常见的本地文件系统

网络文件系统

内核映射文件系统

17.1 Linux文件管理

在介绍文件系统原理之前，读者首先对文件以及文件的管理应该有初步认识。Linux系统的文件管理非常灵活，而且提供强大的功能，本节介绍Linux文件管理的基本概念。

文件和目录的概念

在进入本章内容之前，首先给出Linux系统里面文件和目录相关的几个概念。

文件系统是磁盘上特定格式的文件块集合，操作系统通过特定的结构可以方便的查找和访问集合内某个磁盘块。

文件是建立在文件系统概念上的，是存储在文件系统中一组磁盘块数据的命名对象。一个文件可以是空文件（没有占用磁盘块），也可以由任意多个磁盘块（由文件系统限制）组成。

17.1.2 文件的结构

文件是Linux系统处理数据的基本单位，实际上，Linux系统所有的数据以及其他实体都是按照文件组织的。本节介绍文件相关的知识。

1. 文件的构成
2. 文件的命名方法
3. 文件名通配符

17.1.3 文件的类型

Linux系统按照文件中数据的特点对文件划分不同的类别，称作文件类型。文件划分类型后，系统处理文件可以分类处理。应用程序按照系统划分的文件类型处理文件，可以提高工作效率。Linux内核把文件类型归类如下：

1. 普通文件
2. 文本文件
3. 二进制文件
4. 目录
5. 设备文件

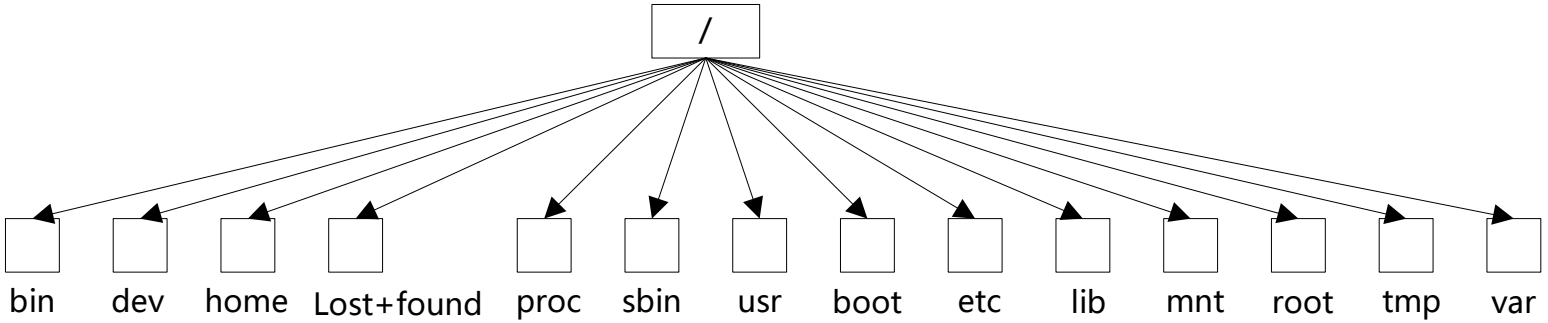
17.1.4 文件系统的目录结构

Linux系统继承了UNIX系统的特点，文件系统的目录有约定的结构，并且每个目录也有约定的功能定义。在Linux系统中，除了根目录（/）以外，所有的磁盘分区和设备都是组织在文件系统里的，根目录（/）是所有文件和目录的开始。

Linux系统有一种特殊的文件，叫做链接文件，链接文件内保存了被链接文件的存放路径，链接文件可以存放在任意路径下。通过链接文件方便了用户访问某个文件，同时也给脚本编写带来便利，在脚本中可以指定访问一个确切的文件名，这个文件是个链接文件，链接到具体的文件，只需要根据不同情况修改链接文件而不需要修改脚本。链接文件分为符号链接和硬链接两种：

1. 符号链接
2. 硬链接

17.1.4 文件系统的目录结构



17.1.5 文件和目录的存取权限

Linux系统提供chmod命令设置文件的访问权限。只有文件主和超级用户可以使用chmod修改文件的访问权限。chmod命令在执行的时候会检查文件主和调用程序的用户ID，通过比较判断是否能执行修改权限操作。chmod命令提供了两种修改文件访问权限的方式如下：

1. 符号方式
2. 数字方式

17.1.6 文件系统管理

多数存储设备（如硬盘和Flash）可以分成多个分区，每个分区可以有不同类型的文件系统。在Linux系统中，文件系统可以根据需要随时装载。在系统刚启动时候，只有根文件系统被安装上。根文件系统的文件主要是保证系统正常运行的操作系统的代码文件，以及若干语言编译程序、命令解释程序和相应的命令处理程序等构成的文件，此外还有大量的用户文件空间。根文件系统一旦安装上，在整个系统运行过程中是不能卸下的，它是系统的基本部分。

其他文件系统（例如光盘文件系统）可以根据需要作为子系统动态安装到主系统中。/mnt目录是为挂载文件系统设置的。挂载文件系统是很简单的，对于没有格式化的分区首先是格式化：

```
mkfs -c /dev/hda1
```

17.2 Linux文件系统原理

文件系统通过把存储设备划分成块，然后把文件分散存放在文件块的方式把数据存储在设备中。文件系统的管理核心是对文件块的管理。文件系统要维护每个文件的文件块分配信息，而且分配信息本身也要存储在存储设备上。不同的文件系统有着不同的文件块分配和读取方法。

通常有两种文件系统分配策略：块分配（block allocation）和扩展分配（extention allocation）。块分配是每当文件大小改变的时候重新为文件分配空间，扩展分配是预先给文件分配好空间，只有当文件超出预分配的空间时候一次性为文件分配连续的块。

Linux支持众多文件系统，实际的文件系统块分配算法非常复杂，文件系统直接影响操作系统的稳定性和可靠性。Linux的文件系统可以大致分成非日志文件系统和日志文件系统。

17.2.1 非日志文件系统

日志是记录文件系统操作的手段，非日志文件系统不记录文件系统的更新操作。记录日志有很多有点，但是非日志文件系统通常也工作稳定。在某些情况下，非日志文件系统存在不少问题。如写入文件操作，首先是更新文件的磁盘分区占用信息meta-data，然后写入文件内容。如果恰巧在更新文件meta-data信息还没有写入文件内容的时候，机器发生意外断点情况，会造成严重后果。读取未写入完成的文件会造成数据不一致的结果。

Linux系统启动的时候会使用fsck程序检查磁盘。fsck程序根据/etc/fstab文件描述的文件系统检查文件系统的meta-data信息是有效的。系统关闭的时候仍然会调用fsck程序把所有文件系统的缓冲数据写入到物理设备，并确认文件系统被彻底卸载。

17.2.2 日志文件系统

日志文件系统是在传统的非日志文件系统上加入了记录文件系统操作日志功能。日志文件系统的设计思想是把文件系统的操作写入日志记录，在磁盘分区保存日志记录。使用日志文件系统最大的好处是可以在系统发生灾难如掉电的时候最大限度保证数据的完整性。

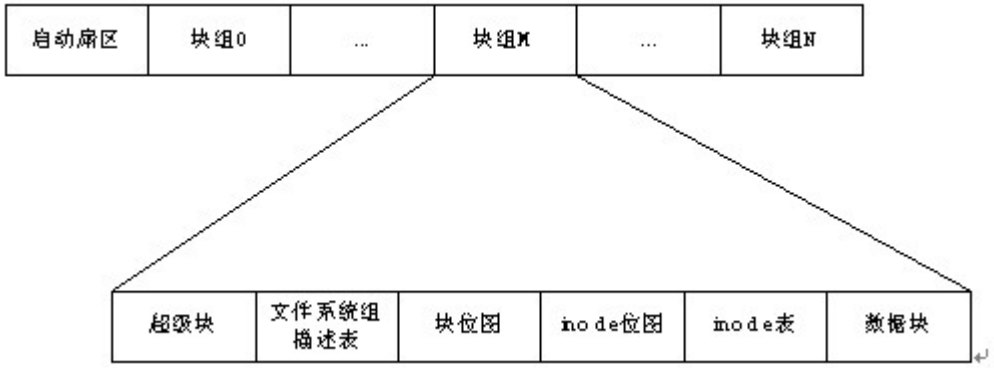
对日志文件系统的文件做写操作之前，首先是操作日志文件。如果恰巧发生断电故障，重新启动计算机后，**fsck**程序会根据日志记录恢复发生故障前的数据。使用日志文件系统后，文件系统所有操作都会记录到日志。系统每间隔一段时间会把更新后的**meta-data**和文件内容从缓冲区写入磁盘。与非日志文件系统不同的是，在更新**meta-data**之前，系统仍然会向日志文件写入一条记录。

17.3 常见的Linux文件系统

在了解了文件系统的原理和管理方法后，本节介绍几种实际应用的文件系统，包括在PC机上应用广泛的ext2和ext3文件系统，以及嵌入式开发最常见的JFFS文件系统。文件系统的实现虽然不同，但是核心的思想基本是一致的，不同的文件系统考虑的侧重点不同。了解常见的文件系统有助于嵌入式Linux系统学习。

17.3.1 ext2文件系统

ext2文件系统是Linux系统使用最广泛的文件系统。它的设计思想是由一系列逻辑上线性排列的数据块构成，每个数据块大小相同。所有的数据块被划分成若干个分组，每个组包含相同个数的数据块，整个文件系统的布局。



17.3.2 ext3文件系统

ext3文件系统直接从ext2文件系统发展来的，完全兼容ext2文件系统。目前ext3文件系统已经很稳定，用户可以直接过渡到ext3文件系统。除增加了日志文件功能外，ext3文件系统的结构与ext2文件系统完全相同。ext3既可以仅对meta-data进行日志操作，也可以同时对文件数据块做日志。具体来说，ext3提供日志、预定和写回三种日志模式。

日志（Journal ）模式：

预定（Ordered）模式：

写回（Writeback）模式：

JDB提供了日志记录、原子操作和事务三种接口方式，功能如下：

日志记录。

原子操作。

事务。

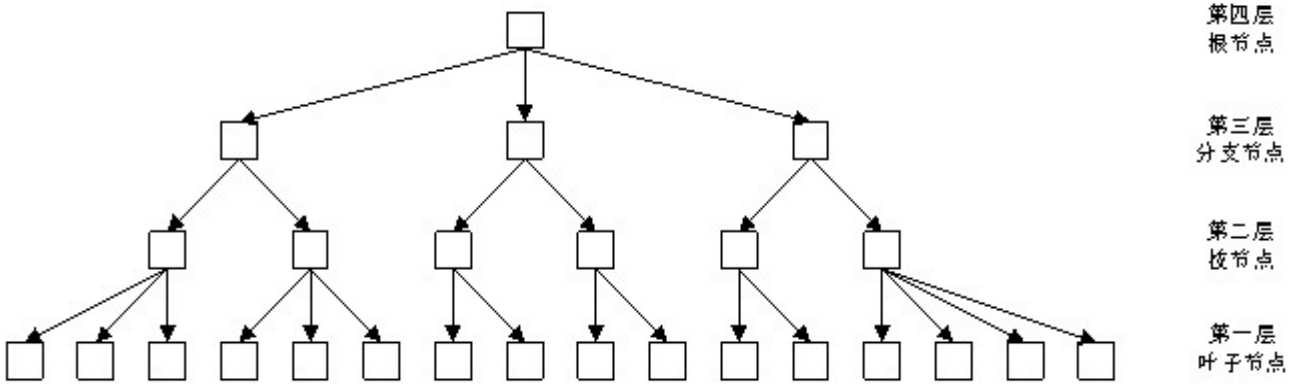
17.3.3 ReiserFS文件系统

与ext3文件系统不同，ReiserFS文件系统是一个完全重新设计的文件系统。ReiserFS文件系统适合大数据量的存储需求，可以轻松管理超大文件。ReiserFS借鉴了面向对象思想，该文件系统分成语义层和存储层。其中，语义层用来管理命名空间和接口定义，存储层管理具体的磁盘空间。语义层解析对象名然后通过键与存储层联系，存储层通过键确定数据在磁盘上的存储位置。在ReiserFS文件系统中，键的值是唯一确定的。以下是语义层和存储层的详细介绍。

1．语义层

2．存储层

17.3.3 ReiserFS文件系统



幻术屏蔽位（固定数据）	节点类型
节点总长度（节点头部长度+数据长度）	
头部CRC校验码	

17.3.4 JFFS文件系统

JFFS文件系统是一个在闪存上使用广泛的读/写文件系统，普遍使用在嵌入式系统，目前使用最多的是第二版本，即JFFS2。

1. 闪存的特性和限制
2. 闪存转换层
3. JFFS2文件系统介绍
4. JFFS2文件系统的垃圾回收
5. JFFS2文件系统的缺点

17.3.5 cramfs文件系统

cramfs是这个问题的一种解决方法。**cramfs**文件系统是专门针对闪存设计的只读压缩的文件系统，其容量上限为**256M**，采用**zlib**压缩。文件系统类型可以是**ext2**或**ext3**。

cramfs文件系统不需要一次性地将文件系统中的所有内容都解压缩到内存之中，而只是在系统需要访问某个位置的数据的时侯，马上计算出该数据在**cramfs**中的位置，将其实时地解压缩到内存之中，然后通过对内存的访问来获取文件系统中需要读取的数据。**cramfs**中的解压缩以及解压缩之后的内存中数据存放位置都是由**cramfs**文件系统本身进行维护的，整个过程对用户透明，对开发人员来说，既方便，又节省了存储空间。

17.4 其他文件系统

前面两节介绍的文件系统特点都是管理本地的普通文件，本节介绍两种特殊的文件系统，一个是用于网络共享的NFS网络文件系统，还有一个是用于内核数据接口的/proc文件系统。这两种系统不同于普通的文件系统，体现了Linux设计的巧妙以及网络功能的强大。

17.4.1 网络文件系统

NFS（Net File System）网络文件系统，最早由 Sun微系统公司开发，后经IETF扩展，现在能够支持在不同类型的系统之间通过网络进行文件共享。也就是说NFS 可用于不同类型计算机、操作系统、网络架构和传输协议运行环境中的网络文件远程访问和共享。

在同一网络中使用NFS可以方便的在不同用户之间共享目录。如一个软件开发团队的成员可以把共同使用的服务器的目录通过NFS映射到本地机器的一个目录，然后通过访问本地目录就可以完成对远程服务器的访问。使用NFS访问远程机器的目录省去了登陆输入用户名密码的繁琐步骤。嵌入式系统使用NFS映射目录，用户可以像使用本地文件系统一样从嵌入式设备中存取数据，减少了文件上传下载操作。

17.4.2 /proc影子文件系统

Linux内核提供了一个特殊的文件系统，挂载到/proc，用于在系统运行时访问内核数据结构、改变内核设置。/proc是一个虚拟的文件系统，是一种内核以及内核模块用来向进程（Process）发送信息的机制。

/proc由内核控制，没有承载/proc的实际设备。因为/proc主要存放由内核控制的状态信息，所以大部分这些信息的逻辑位置位于内核控制的内存。使用‘ls –l’命令查看/proc目录看到大部分文件都是0字节；但是查看这些文件的时候，确实可以看到一些信息。这是因为/proc文件系统和其他常规的文件系统一样把自己注册到虚拟文件系统层（VFS）。

17.5 小结

本章介绍了Linux系统重要的一部分-文件系统，包括文件系统的原理，管理方式，常见的文件系统等。分析了文件系统的数据结构操作方式，并且给出了实例操作。读者应当从实际出发，多动手，通过实践学习文件系统。另外需要注意的是，对文件系统的一些操作有风险，应注意关键数据的备份，以防止文件被破坏。第18章讲解建立嵌入式Linux交叉编译工具链。

第18章　建立交叉编译工具链

工欲善其事，必先利其器。嵌入式Linux开发不能缺少的就是开发工具，其中最基本的是编译工具。和传统的编译方式不同，嵌入式系统开发需要在不同的计算机上编译出开发板需要的程序，所用的编译工具也与传统的编译工具不同。本章讲解如何构建嵌入式Linux开发需要的交叉编译工具链，主要内容如下：

交叉编译工具链介绍

手工构建交叉编译工具链

使用脚本构建交叉编译工具链

交叉编译工具链常见的问题

18.1 什么是交叉编译

交叉编译是伴随嵌入式系统的发展而来的，传统的程序编译方式，生成的程序直接在本地运行，这种编译方式称作本地编译（Native Compilation）；嵌入式系统多采用交叉编译的方式，在本机编译好的程序是不能在本机运行的，需要通过特定的手段（例如烧写，下载等）安装到目标系统上执行。这种编译运行的方法比较繁琐，是受到实际条件限制的。大多数的嵌入式系统目标板系统资源都很有限，无论是存储空间还是CPU处理能力，都很难达到编译程序的要求。而且很多目标板是没有操作系统的，需要通过其他的机器编译操作系统和应用程序。

18.2 需要哪些东西

建立交叉编译工具链可以在多种平台上进行，本书建议使用x86体系结构的PC机，在Linux系统下进行。这种选择不是强制的，是因为x86体系结构是使用最广泛的。同时，使用Linux系统可以避免许多开发环境的设置。建立交叉编译工具链需要以下的准备：

磁盘空间：

源代码：

命令行：

其他工具：

18.3 手工创建工具链

构建交叉编译器首先是确定目标平台。在GNU系统，每个目标平台都有一个明确的格式和名称，这些信息可以在构建工具的过程中识别工具的正确版本。因为编译交叉工具链的过程中会有两套编译器环境，明确了平台名称和格式才能保证生成工具不出错。

在运行gcc的时候，gcc会在路径中查找包含指定目标规范的应用程序路径。GNU的目标规范格式为CPU-PLATFORM-OS。如x86目标机名称是i686-pc-linux-gnu，ARM目标平台名称是arm-linux-gnu。构建交叉工具链通常有以下3种方法：

1. 分步骤手工编译
2. 通过脚本编译
3. 直接获取交叉编译工具链

18.3.1 准备工作-获取源代码

手工编译交叉编译工具链需要编译用到的库和源代码，首先是下载如下源代码：

Linux内核代码

glibc库

工具程序

内核线程包

编译器

18.3.2 开始了-建立工作环境

构建交叉编译工具链需要建立一个工作环境，包括建立工作目录和环境变量。工作目录是交叉编译工具链构建过程中使用的目录，工作目录没有特殊要求，用户可以根据自己的喜好建立。本书假定在当前用户的用户目录下建立一个armtools作为工作目录。

mkdir armtools

18.3.3 建立linux内核头文件

交叉编译器需要通过内核头文件获取目标平台支持的系统函数调用的信息。因此，需要拷贝内核的头文件。但是，直接拷贝内核头文件是不行的，还需要对内核做简单的配置，让内核脚本生成目标平台的头文件。需要注意的是，Linux内核版本和编译器版本依赖比较强，一个版本的编译器不能保证编译所有内核版本。

（1）首先在$PRJROOT/kernel目录下解压缩内核源代码

（2）接下来进入内核代码目录配置目标平台的头文件：

（3）Linux内核配置的顶层界面，按照功能划分分成若干项，与生成目标平台头文件相关的是System Type项。使用光标键移动到该选项，回车进入配置界面。

（4）目标平台选择完毕后，直接回车，回到了系统类型配置界面。

（5）配置内核结束后，检查是否生成正确的内核头文件include/linux/version.h和include/linux/autoconf.h，这两个文件是编译glibc需要的。

（6）最后一步是把生成的内核头文件复制到交叉编译工具链目录。

18.3.4 编译安装binutils

binutils是交叉编译工具链里一个重要的工具包，由GNU提供。binutils包括了连接器、汇编器和用于目标文件和档案的工具。binutils工具包主要针对二进制代码的维护。

1. **binutils工具包内容**
2. **编译安装binutils工具包**

18.3.5 编译安装gcc的c编译器

这一步建立arm-linux-gcc交叉编译器，但是这个gcc编译器是没有glibc库支持的。也就是说，此编译器只能用于编译内核、Bootloader等不需要glibc库支持的程序。glibc库也要使用此编译器，所以创建这个编译器确切的说是为了建立glibc库。有的文档把这个步骤叫做gcc的第一次编译，这是相对后面的编译过程讲的。

18.3.6 编译安装glibc库

GNU glibc库是Linux系统程序非常重要的组成部分。如果用户开发目标平台的应用程序，则必须编译安装glibc库。glibc-2.3.2版本推荐先安装以下的工具：

GNU make 3.79或更新的版本；

GCC 3.2或更新的版本；

GNU binutils 2.13或更新的版本。

18.3.7 编译安装gcc的c, c++编译器

第一次编译的gcc没有glibc支持，编译好glibc以后，需要重新编译gcc用于支持glibc库。需要注意的是，第一次编译的gcc只能支持C语言程序编译，现在编译的gcc可以支持C和C++语言。本次编译也叫做第二次编译。

18.3.8 最后的工作

到目前为止，已经完成了分步骤构建交叉工具链的所有工作。最后还需要测试一下构建的工具链是否符合要求。

18.4 使用脚本创建工具链

手工构建交叉编译工具链不仅步骤繁琐，而且容易出错。为了简化构建工具链的过程，减小出错，Linux社区设计了一套编译工具链的脚本Crosstool。Crosstool是一组建立交叉编译环境的脚步工具，通过指定不同参数，Crosstool脚本可以建立指定版本的gcc编译器和glibc程序库。该脚本也是一个开源项目，读者如果出于工作需要建议使用crosstools构建交叉编译工具链。

18.5 更简便的方法 – 获取已编译好的交叉编译环境

以上两节讲的构建交叉编译工具链的方法都需要手工干预。本节介绍一种更简便的方法，直接下载已经编译好的工具链。Linux社区提供了多个版本的ARM平台交叉编译工具链。下面是三个不同版本编译器的下载地址。

ARM Linux

下载地址：
ftp://ftp.arm.linux.org.uk/pub/armlinux/toolchain/

提供版本：**arm-linux-gcc版本2.95.3, 3.0和3.2**

Handhelds.org

下载地址：**ftp://ftp.handhelds.org/projects/toolchain/**

提供版本：**arm-linux-gcc版本3.3.2和3.4.1**

ELDK 4.1

下载地址：**ftp://ftp.denx.de/pub/eldk/4.1/**

提供版本：**arm-linux-gcc版本4.0.0**

18.6 小结

本章讲解了搭建嵌入式Linux开发环境最关键的技术－建立交叉编译工具链，交叉编译是嵌入式开发不可缺少的一个工作环节。由于GNU工具和库的版本依赖关系很强，建立交叉编译工具链的过程可能会遇到各种问题，学习创建交叉编译工具本身能学到许多有关嵌入式Linux系统、程序库的知识。第19章讲解使用交叉编译工具链建立BusyBox命令系统。

第19章　强大的命令系统BusyBox

BusyBox是嵌入式系统常用的一个命令系统，它的功能强大、占用存储容量小，这些优点都适合嵌入式系统。本章从**BusyBox**的原理出发介绍了**BusyBox**的编译安装，以及如何应用在嵌入式系统。**BusyBox**的编译安装都是比较容易的，读者可以轻易的把**BusyBox**移植到嵌入式开发板上。本章主要内容如下：

BusyBox起源

BusyBox工作原理

在**PC**机上安装**BusyBox**

移植**BusyBox**到**ARM**开发版

19.1 BusyBox简介

BusyBox是Linux平台的一个工具集合。BusyBox可以包含最基本的系统命令如ls和cat，还可以包含功能更复杂的程序如grep和find，甚至可以把HTTP服务器也集成在一个软件包内。BusyBox把Linux系统常用的命令和工具以及服务程序集成在一个可执行文件内，通常体积在1MB字节左右。如果单独存放每条命令，可能需要几MB甚至几十MB存储空间，这对存储空间紧张的嵌入式系统来说是很难接受的。BusyBox是很适合嵌入式系统的，本节介绍BusyBox的工作原理和安装流程。

19.1.1 简单易懂的BusyBox

BusyBox项目最初是在**1996**年发起的，当时嵌入式系统并没有开始流行。**BushBox**最初的目的是被设计为一个安装在软盘上的命令系统，因为当时还没有可以移动的大容量可擦写存储介质，软盘是最常用的存储介质。使用过软盘的读者知道，它的容量很小，对于今天的计算机来说几乎没有什么用武之地。**BusyBox**可以把常见的**Linux**命令打包编译成一个单一的可执行文件。通过建立链接，用户可以像使用传统的命令一样是用**BusyBox**。

BusyBox的出现是基于**Linux**共享库。对于大多数**Linux**工具来说，不同的命令可以共享许多东西。如查找文件的命令**grep**和**find**，虽然功能不完全相同，但是两个程序都会用到从文件系统搜索文件的功能，这部分代码可以是相同的。**BusyBox**的聪明之处在于把不同工具的代码，以及公用的代码都集成在一起，从而大大减小了可执行文件的体积。

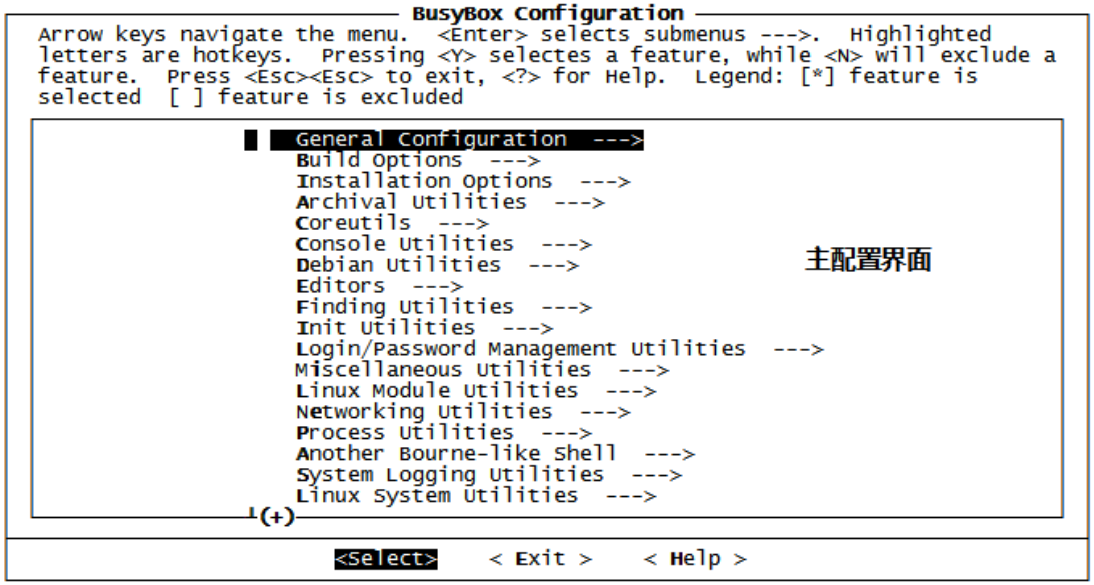
19.1.2 BusyBox工作原理

BusyBox利用了shell传递给C语言main()函数的参数，回想一下C语言main()函数的定义：

```
int main( int argc, char *argv[] )
```

19.1.3 安装BusyBox

安装BusyBox需要从源代码开始编译。首先是获取源代码，从BusyBox的官方网站（http://busybox.net/downloads/）下载。这个链接里有多个版本，但是高版本的BusyBox存在一些问题，编译过程中容易出错。



19.2 交叉编译BusyBox

BusyBox最大的特点是占用存储空间小，在PC机使用优势不明显。本节介绍如果在嵌入式系统上配置安装BusyBox。交叉编译BusyBox需要有交叉编译环境，在第18章已经讲解如何建立交叉编译环境，现在使用已经建立好的环境为例介绍交叉编译BusyBox。

（1）首先查看19.1.3节编译得到的busybox可执行文件的类型：

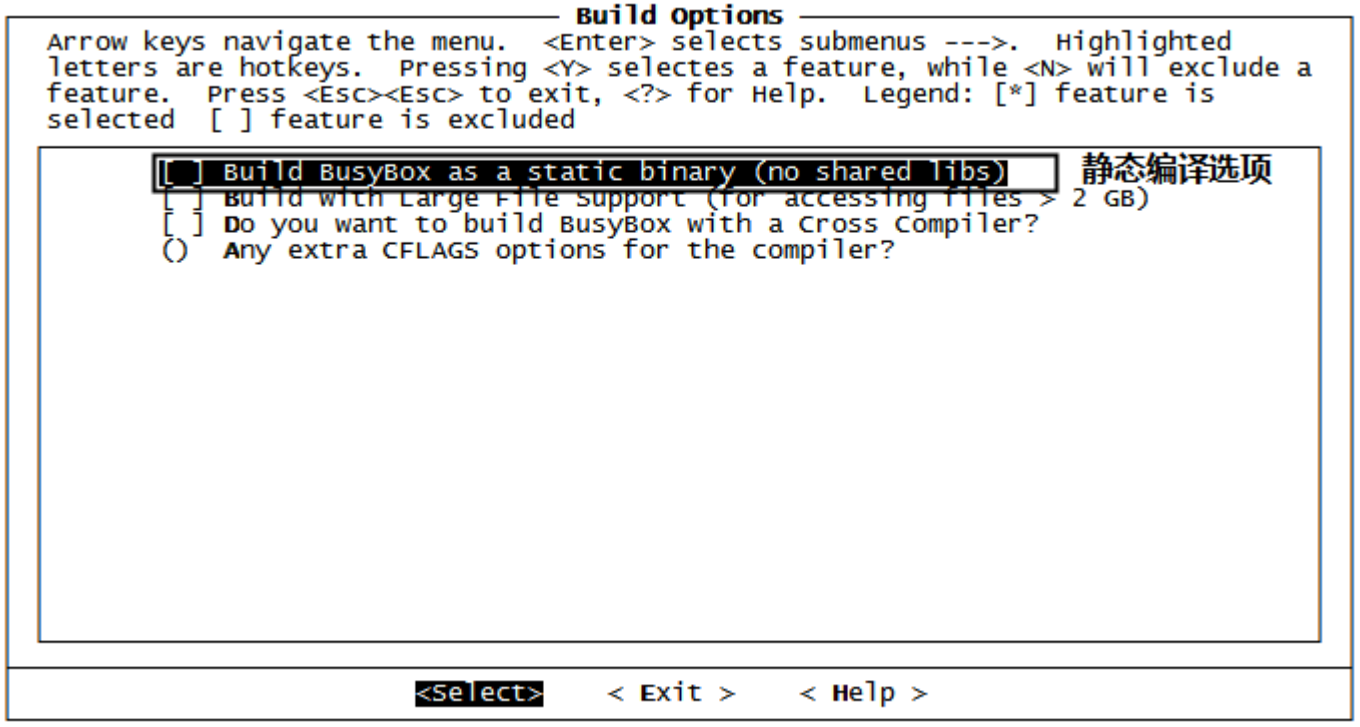
（2）在交叉编译BusyBox之前先分析一下BusyBox的Makefile。

（3）至此，已经找到如何使用交叉编译，下面可以编译ARM开发板使用的BusyBox了。

（4）使用“make menuconfig”命令进入BusyBox的配置界面，选择Build Options子菜单。

（5）配置完毕后，保存退出。重新编译BusyBox，编译完成后，查看可执行文件类型。

19.2 交叉编译BusyBox



19.3 使用BusyBox

使用**BusyBox**的最终目的是在嵌入式开发板上应用。本节从**BusyBox**在嵌入式系统中的初始化过程出发，介绍如何移植到开发板。其中，需要读者了解第**16**章介绍的嵌入式**Linux**启动流程，便于理解本节的内容。

19.3.1 BusyBox初始化

在第16章Linux启动流程介绍了Linux内核的初始化代码kernel/init/main.c里有几个语句如下：

```
if (execute_command)
  execve(execute_command,argv_init,envp_init);
 execve("/sbin/init",argv_init,envp_init);
```

19.3.2 目标板BusyBox安装

了解了BusyBox的工作流程后，安装BusyBox就变得很简单了。主要的任务是设置好BusyBox与Linux内核的结合点。

可以通过两种手段设置BusyBox到开发板。一种方法是在PC机上制作cramfs镜像。这种方法好处是便于操作，但是需要注意的是BusyBox的路径不能搞错。缺点是出错后修改麻烦，每次都需要烧写开发板的Flash。

19.4 小结

本章介绍了BusyBox的工作原理，编译安装和移植到开发板的技巧。读者亲自动手实践，对于理解BusyBox的工作流程有很大帮助。另外，对于有兴趣的读者可以挑选BusyBox源代码的文件，阅读代码理解BusyBox的设计思想和工作流程。第20章讲解嵌入式Linux开发的重点－内核移植。

第20章　Linux内核移植

软件移植的概念简单的说就是让一套软件在指定的硬件平台上正常运行。移植至少包括了两个不同的硬件或者软件平台。对于应用软件来说，移植主要考虑操作系统的差异，重点在修改系统调用。本章的重点是**Linux**内核移植，需要考虑的硬件平台的差异，涉及较多知识。主要内容如下：

Linux内核移植要点

内核体系结构框架

从现有代码移植内核

20.1 Linux内核移植要点

Linux本身对内存管理（MMU）有很好的支持。因此，在移植的时候首先要考虑到目标硬件平台是否支持MMU。以ARM平台为例，ARM7内核的CPU不支持MMU，无法直接把Linux内核代码移植到ARM7核的硬件平台。μcLinux是专门针对ARM7这类没有MMU的硬件平台设计的，它精简了MMU部分代码。本书的目标平台是S3C2440A，该处理器基于ARM9核，支持MMU，可以直接移植Linux 2.6版本的内核代码。

1. 目标平台
2. 内存管理单元（MMU）
3. 内存映射
4. 存储器

20.2 平台相关代码结构

Linux内核代码目录基本是安装功能块划分的，每个功能块的代码存放在一个目录下。如mm目录存放内存管理单元相关代码；ipc存放了进程间通信相关的代码；kernel存放进程调度相关代码等。

arch目录下每个平台的代码都采用了与内核代码相同的目录结构。以arch/arm目录为例，该目录下mm、lib、kernel、boot目录与内核目录下对应目录的功能相同。此外，还有一些以字符串mach开头的目录，对应不同处理器特定的代码。从arch目录结构可以看出，平台相关的代码都存放到arch目录下，并且使用与内核目录相同的结构。使用SourceInsight工具可以看到许多的同名称函数，原因就是内核代码调用的函数是平台相关的，每个平台都有自己的实现方法。对于内核来说，使用相同的名字调用，通过编译选项选择对应平台的代码。

20.3 建立目标平台工程框架

Linux内核2.6版本已经对ARM处理器有很好的支持，并且对三星公司的S3C2440提供一定支持。但是，嵌入式硬件系统的差别很大，移植Linux内核到新的开发板仍然需要修改或者增加针对特定硬件的代码。

20.3.1 加入编译菜单项

修改arch/arm/mach-s3c2410/Kconfig文件，在“endmenu”之前加入下面的内容：

```
87 config ARCH_MINI2440
88   bool "mini2440"
89   select CPU_S3C2440
90   help
91     Say Y here if you are using the mini2440.
```

20.3.2 设置宏与代码文件的对应关系

在设置宏与代码文件对应关系之前，首先建立一个空的代码文件。在**arch/arm/mach-s3c2410**目录下建立**mach-mini2440.c**文件，用于存放与**mini2440**开发板相关的代码。

建立**mach-mini2440.c**文件后，修改**arch/arm/mach-s3c2410/Makefile**文件，在文件最后加入**mach-mini2440.c**文件的编译信息：

```
43 obj-$(CONFIG_ARCH_MINI2440) += mach-mini2440.o
```

20.3.3 测试工程框架

工程框架配置修改完毕后，需要进行简单的测试，根据测试结果判断框架是否搭建成功。回到内核代码顶层目录，输入“**make ARCH=arm CROSS_COMPILE=arm-linux-menuconfig**”命令，出现内核设置图形界面。

加载默认配置文件后，可以开始配置新增加的菜单。进入“**System Types**”菜单项，打开“**S3C24XX Implementations**”菜单，出现一个目标开发板的列表：

[ ] Simtec Electronics BAST (EB2410ITX)
[ ] IPAQ H1940
[ ] Acer N30
[ ] SMDK2410/A9M2410
[ ] SMDK2440
[ ] AESOP2440
[ ] QQ2440/mini2440
[ ] Thorcom VR1000
[ ] HP iPAQ rx3715
[ ] NexVision OTOM Board
[ ] NexVision NEXCODER 2440 Light Board
[ ] mini2440

20.4 建立目标平台代码框架

在20.2.2节编译的内核代码最后出现了链接错误，提示vmlinux.lds文件链接失败。lds文件是GNU ld工具使用的一种脚本文件，该文件描述了如何分配链接后的内存区域和地址等信息，通过lds文件报的错误可以顺藤摸瓜分析问题所在。

20.4.1 ARM处理器相关结构

首先打开**arch/arm/kernel/vmlinux.lds**文件，找到**815**行，代码如下：

```
815 ASSERT((__proc_info_end - __proc_info_begin), "missing CPU support")
```

20.4.2 建立machine_desc结构

Linux内核提供了MACHINE_START和MACHINE_END宏供建立machine_desc结构使用，建议使用宏建立结构。

打开arch/arm/mach-s3c2410/mach-mini2440.c文件，加入下面的代码：

```
53 MACHINE_START(MINI2440, "MINI2440")
54   .phys_ram = S3C2410_SDRAM_PA,
55   .phys_io  = S3C2410_PA_UART,
56   .io_pg_offst  = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
57   .boot_params  = S3C2410_SDRAM_PA + 0x100,
58   .init_irq = mini2440_init_irq,
59   .map_io   = mini2440_map_io,
60   .init_machine = mini2440_init,
61   .timer    = &s3c24xx_timer,          63 MACHINE_END
```

20.4.3 加入处理函数

在mach-mini2440.c文件加入MINI2440结构中指定的几个函数定义如下：

```
 52 void __init  mini2440_init_irq(void)
53 {
54 }
55
56 void __init  mini2440_init(void)
57 {
58 }
59
60 void __init  mini2440_map_io(void)
61 {
62 }
```

20.4.4 加入定时器结构

在MINI2440结构定义中，使用了一个名为s3c24xx_timer的sys_timer结构变量，该变量定义在arch/arm/mach-s3c2410/timer.c文件定义如下：

```
252 struct sys_timer s3c24xx_timer = {
253   .init   = s3c2410_timer_init,      // 定时器初始化函数
254   .offset   = s3c2410_gettimeoffset,    // 读取定时器延时
255   .resume   = s3c2410_timer_setup    // 恢复定时器
256 };
```

20.4.5 测试代码结构

回到内核源代码根目录，执行“**make ARCH=arm CROSS_COMPILE=arm-linux- bzImage**”开始编译内核。这次编译没有出错信息，会得到正确的编译结果，查看**arch/arm/boot**目录已经有目标文件**Image.gz**，表示已经编译生成运行于**ARM**处理器的内核。

到目前为止，已经可以编译工作在**ARM**处理器上的代码，但是内核代码还不能启动，因为还没有加入实际的代码，在下节介绍如何加入目标平台相关的代码。

20.5 构建目标板代码

Linux内核已经为ARM处理器设计好了代码框架，只要按照这个框架加入针对某种开发板和处理器的代码即可工作。加入代码还是按照前面提到的原则，能使用已有的通用代码尽量使用，并且尽可能的参考现有开发板代码的处理方法。

20.5.1 处理器初始化

首先在mach-mini2440.c文件加入处理器初始化代码如下：

```
56 void __init mini2440_init(void)
57 {
58   set_s3c2410ts_info(&mini2440_ts_cfg);    // 注册触摸屏结构
59   set_s3c2410udc_info(&mini2440_udc_cfg); // 注册UDC结构
60   set_s3c2410fb_info(&mini2440_lcdcfg);     // 注册LCD结构
61 }
```

20.5.2 端口映射

端口映射函数设置S3C2440处理器的I/O端口描述结构、时钟频率、串口等，代码如下：

```
150 void __init mini2440_map_io(void)
151 {
152   s3c24xx_init_io(mini2440_iodesc,
ARRAY_SIZE(mini2440_iodesc));
153   s3c24xx_init_clocks(12000000);
154   s3c24xx_init_uarts(mini2440_uartcfgs,
ARRAY_SIZE(mini2440_uartcfgs));
155   s3c24xx_set_board(&mini2440_board);
156   s3c_device_nand.dev.platform_data = &bit_nand_info;
157 }
```

20.5.3 中断处理

内核提供了一个s3c24xx_init_irq()处理函数，因此中断处理函数直接引用即可：

```
186 void __init mini2440_init_irq(void)
187 {
188   s3c24xx_init_irq(); // 调用系统提供的中断处理函数
189 }
```

20.5.4 定时器处理

内核提供了一个定时器处理函数结构如下：

```
struct sys_timer s3c24xx_timer = {
.init    = s3c2410_timer_init,        // 定时器初始化函数
.offset    = s3c2410_gettimeoffset,    // 获取定时器值
.resume    = s3c2410_timer_setup    // 恢复定时器设置
};
```

20.5.5 编译最终代码

到目前为止，已经添加了所有与mini2440开发板有关的代码，保存文件后，可以开始编译内核。回到内核代码根目录，执行“make ARCH=arm CROSS_COMPILE=arm-linux- bzImage”重新编译代码，最终在arch/arm/boot目录下生成bzImage文件，针对开发板的内核代码编译成功。

通过U-Boot或者其他的Bootloader工具可以把代码烧写到开发板的Flash存储器，然后重新启动开发板，可以从液晶屏看到启动过程打印的提示信息。

提示：由于没有设置USB控制器和声音控制器，因此开发板上USB和声音相关的功能都无法使用。

20.6 小结

本章介绍了如何移植Linux内核代码到新的开发板。移植内核到新的硬件平台是一件繁琐的事情，读者在移植过程中需要有耐心。移植内核代码要涉及到软件和硬件相关的知识，读者应该结合硬件提供的电路图纸和手册，利用现有的软件工具和代码，构建符合需求的内核代码。第21章讲解Linux内核和应用程序的调试技术。

第21章　内核和应用程序调试技术

调试程序的目的是定位程序中的问题。调试程序无外乎几种方式：查看程序运行时内部数据、跟踪程序运行、查看信号量的变化。调试器就是一类帮助程序员调试程序的工具。本章讲解Linux系统下最基本的调试器gdb使用方法，以及使用kdb调试内核的技术，主要内容如下：

GDB调试器介绍

基本的调试技术

调试意外终止的程序

使用printk()调试Linux内核

使用KGDB调试Linux内核

21.1 使用GDB调试应用程序概述

GDB是GNU开源组织发布的一款调试器，提供了丰富的功能。GDB调试器不仅能调试普通的应用程序，还可以调试正在运行的进程和线程，甚至Linux内核。GDB是一个开源的调试器，不仅能调试C语言编写的代码，还可以调试Ada、C++、Java、Pascal等语言编写的程序。GDB支持Linux、Windows等多种平台，可以非常方便的调试各种类型的程序。不过，GDB最大的不足是一个命令行的工具，对初学者来说入门比较麻烦，尤其是用惯了VisualC++之类图形化调试器的开发人员。

GDB的功能可以分成4类：提供多种方式加载被调试的程序；为程序设置断点，可以根据用户设置的表达式设置断点；检查程序运行过程中各种状态和和信号的变化；可以动态改变程序执行的环境。本节介绍GDB在Linux环境下的各种调试技术。

21.2 基本的调试技术

GDB的功能通过内部的命令和启动时命令行提供，命令行的格式如下：

gdb [options] [executable-file [core-file or process-id]]

gdb [options] --args executable-file [inferior-arguments ...]

在GDB的命令行可以输入参数和选项，包括指定被调试程序的参数和进程号等。一般使用“gdb <被调试程序名>”的形式启动GDB，不需要指定参数。有关GDB的参数和内置命令可以参考附录。

21.2.1 列出源代码

列出代码是一个必要功能。对于GDB这种命令行的调试器来说，调试过程中屏幕的信息在不断更新，如果没有查看代码功能，用户操作很不方便。GDB显示代码的命令是“list”。

（1）启动gdb调试器：

（2）进入gdb调试环境后，输入命令“list”，然后回车，gdb打印出最开始的代码。

（3）在调试环境下，继续输入“list”回车后。

（4）在调试过程中，代码会经常改变，使用行号的方法很不方便。gdb提供了通过函数名显示代码的功能。

21.2.2 断点管理

调试中最常用的功能就是断点。断点的意思是给程序代码某处做一个标记，当程序运行到此处的时候就会停下来，等待用户的操作。断点通常被设置在程序出错的前面几行，当程序运行到断点以后，程序员通过单步运行程序，并且查看相关变量状态，可以定位错误。

1. 设置断点
2. 查看断点
3. 关闭断点
4. 删除断点

21.2.3 执行程序

执行程序比较简单，gdb提供了“run”和“continue”两个命令。这两个命令共同特点是，在遇到用户设置的断点后会停下来。不同的是，run命令仅用在程序最开始执行的时候。也就是说，run命令把整个程序运行起来，程序运行以后不能使用run命令，因为程序不能被反复调试运行。continue命令只能在程序运行后执行，主要用在程序被断点停住以后，通过continue命令继续执行。

调试程序可以控制程序单步执行，有两个命令“next”表示执行下一条语句，“step”表示跳转到函数内部执行。执行程序的命令在21.1.1.6节介绍。

21.2.4 显示程序变量

gdb提供“print”和“display”两条显示命令。这两条命令的功能基本相同，区别在于display可以锁定显示的变量或者寄存器，当执行程序时，每执行一次都会显示被锁定的变量。print命令只能在调用的时候显示指定的变量或者寄存器值。

21.2.5 信号管理

gdb的一个特色是能模拟操作系统向被调试的应用程序发送信号。使用“signal <信号名称>”发出指定的信号。

信号名称	含义
SIGHUP	程序挂起
SIGINT	向程序发出中断
SIGQUIT	退出信号
SIGILL	遇到非法指令
SIGKILL	杀死进程信号
SIGSEGV	段错误
SIGALRM	警告信号
SIGTERM	程序终止信号
SIGSTOP	程序停止信号
SIGCHLD	子进程信号
SIGPOLL	轮询信号

21.1.6 调试实例

在学习了gdb的基本使用方法以后，本节给出一个gdb调试的实例。

21.3 多进程调试

在嵌入式中，需要采集多种信号，或者响应外部设备发送的某种协议请求。对于这种需求，往往需要在Linux系统设置一些进程，本节介绍多进程和多线程程序调试方法。

gdb提供了多进程程序的调试能力，其调试过程对用户来说很简单，用户只需要指定进程的ID和带有调试信息的程序文件即可调试，其余的过程与普通程序调试基本类似。

21.4 调试意外终止的程序

如果读者编译例21-1的程序后运行，会得到一个出错提示如下：

```
$ ./HttpDemo
Segmentation fault (core dumped)
```

这个提示的意思是段错误，程序中出现了访问非法地址或者段越界的错误。段错误是一种严重的程序错误，出现这类错误后，程序无法继续运行，会异常终止。“core dumped”意思是程序出错时候的环境已经被转存。

转存环境信息是Linux内核提供的一种功能，当应用程序发生致命错误退出的时候，内核会把出错时候的环境信息记录下来，存放到一个文件，称为core文件。gdb可以识别core文件的格式，并且程序恢复出错时候的状态信息，方便调试。

21.5 内核调试技术

普通程序在调试过程中有操作系统的支持，可以跟踪变量和信号，读写内存。相比之下，内核的调试过程就“艰苦”多了，不仅没有操作系统的支持，调试手段本身就很复杂。此外，内核调试过程中，不仅有来自软件的信号，也有来自硬件的中断，调试时候要特别注意。**Linux**内核调试方面，提供了多种调试方法，本节介绍几种常见的内核调试方法。

21.5.1 printk打印调试信息

printk()是内核提供的一个打印函数，作用是向终端打印信息，是一种最常用的**Linux**内核调试技术。通常内核使用**printk()**函数打印提示信息和出错信息。在内核调试最普遍的办法是使用**printk()**函数在可能出错的地方打印，帮助调试。内核使用**printk()**函数而不使用**printf()**函数，原因是**printf()**函数是由**glibc**库提供的，**Linux**内核的函数是不能依赖任何程序库的，否则制作出的映像文件就无法被加载。

printk()函数的用法与**printf()**一致。不同的是，**printf()**函数是可被中断的，而**printk()**函数不会被中断。实际使用的效果是，**printk()**函数输出的内容不会被其他程序打断，保证了输出的完整性。

21.5.2 使用/proc虚拟文件系统

printk()函数打印是一种简单易用的内核调试手段。但是，使用printk()函数存在两个比较大的缺点：每次要打印内核的内容都需要重新编译内核，操作麻烦，调试效率低；大量使用printk()函数会降低系统性能，甚至使系统运行速度明显变慢。

printk()函数打印的内核通过syslogd进程记录到磁盘的log文件。每次打印输出syslogd都会同步输出文件，因此每次打印都要引起磁盘操作。长期使用printk()函数还会导致磁盘文件过大。为了解决printk()函数带来的负面影响，内核开发者通常使用/proc文件系统。

21.5.3 使用KDB调试工具

KDB是SGI公司开发的Linux内核调试器，遵循GPL协议开放源代码。KDB作为内核的一个插件嵌入到内核代码，为内核开发人员提供调试内核的方法。KDB适合调试内核空间的代码，包括内核模块和设备驱动等。

21.5.4 KDB调试指令

KDB提供了丰富的调试命令，可以设置断点，读写内存和寄存器，跟踪堆栈等，本章介绍常用的KDB命令。

1. 断点类指令
2. 流程控制类指令
3. 内存操作指令
4. 堆栈跟踪指令
5. 寄存器指令

21.6 小结

本章讲解了嵌入式Linux开发中内核和应用程序的调试技术。调试是软件开发必不可少的一个环节，也是一个程序员的基本功。调试程序要从程序的功能的工作流程出发，使用调试工具分析程序执行过程中变量、信号、寄存器值的变化是否正确，找出问题所在。调试程序是一个操作性很强的工作，读者应该多实践，不断提高调试程序能力。第22章讲解Linux设备驱动开发。

第22章　Linux设备驱动

驱动程序英文全称Device Driver，也称作设备驱动程序。驱动程序是用于计算机和外部设备通信的特殊程序，相当于软件和硬件的接口，通常只有操作系统能使用驱动程序。在现代计算机体系结构中，操作系统并不直接于硬件打交道，而是通过驱动程序于硬件通信。本章讲解Linux设备驱动程序的知识，主要内容如下：

设备驱动程序的功能和用途

编写Linux内核模块

Linux内核驱动分类

PCI总线介绍

22.1 设备驱动介绍

驱动程序是附加到操作系统的一段程序，通常用于硬件通信。每种硬件都有自己的驱动程序，其中包含了硬件设备的信息。操作系统通过驱动程序提供的硬件信息与硬件设备通信。由于驱动设备的重要性，在安装操作系统后需要安装驱动程序，外部设备才能正常工作。Linux内核自带了相当多的设备驱动程序，几乎可以驱动目前主流的各种硬件设备。

在同一台计算机上，尽管设备是相同的，但是由于操作系统不同，驱动程序是有很大差别的。但是，无论什么系统驱动程序的功能都是相似的，可以归纳为下面三点：

初始化硬件设备。

向操作系统提供统一的软件接口。

提供辅助功能。

22.2 Linux内核模块

Linux内核模块是一种可以被内核动态加载和卸载的可执行程序。通过内核模块可以扩展内核的功能，通常内核模块被用于设备驱动、文件系统等。如果没有内核模块，需要向内核添加功能就需要修改代码、重新编译内核、安装新内核等步骤，不仅繁琐而且容易保出错，不易于调试。

22.2.1 内核模块简介

前面介绍过，Linux内核是一个整体结构，可以把内核想象成一个巨大的程序，各种功能结合在一起。当修改和添加新功能的时候，需要重新生成内核，效率较低。为了弥补整体式内核的缺点，Linux内核的开发者设计了内核模块机制。从代码的角度看，内核模块是一组可以完成某种功能的函数集合。从执行的角度看，内核模块可以看做是一个已经编译但是没有连接的程序。

内核模块是一个应用程序，但是与普通应用程序有所不同，区别在于：

运行环境不同。

功能定位不同。

函数调用方式不同。

22.2.2 内核模块的结构

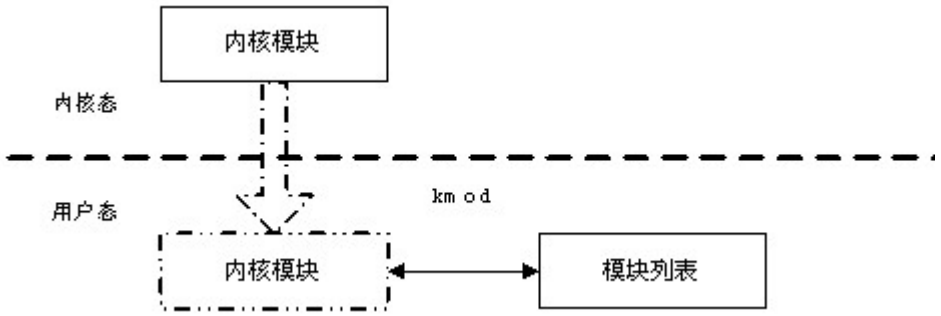
内核编程与用户空间编程最大的区别就是程序的并发性。在用户空间，除多线程应用程序外，大部分应用程序的运行是顺序执行的，在程序执行过程中不必担心被其他程序改变执行的环境。而内核的程序执行环境要复杂的多，即时最简单的内核模块也要考虑到并发执行的问题。

设计内核模块的数据结构要十分小心。由于代码的可重入特性，必须考虑到数据结构在多线程环境下不被其他线程破坏，对于共享数据更是应该采用加锁的方法保护。驱动程序员的通常错误是假定某段代码不会出现并发，导致数据被破坏而很难于调试。

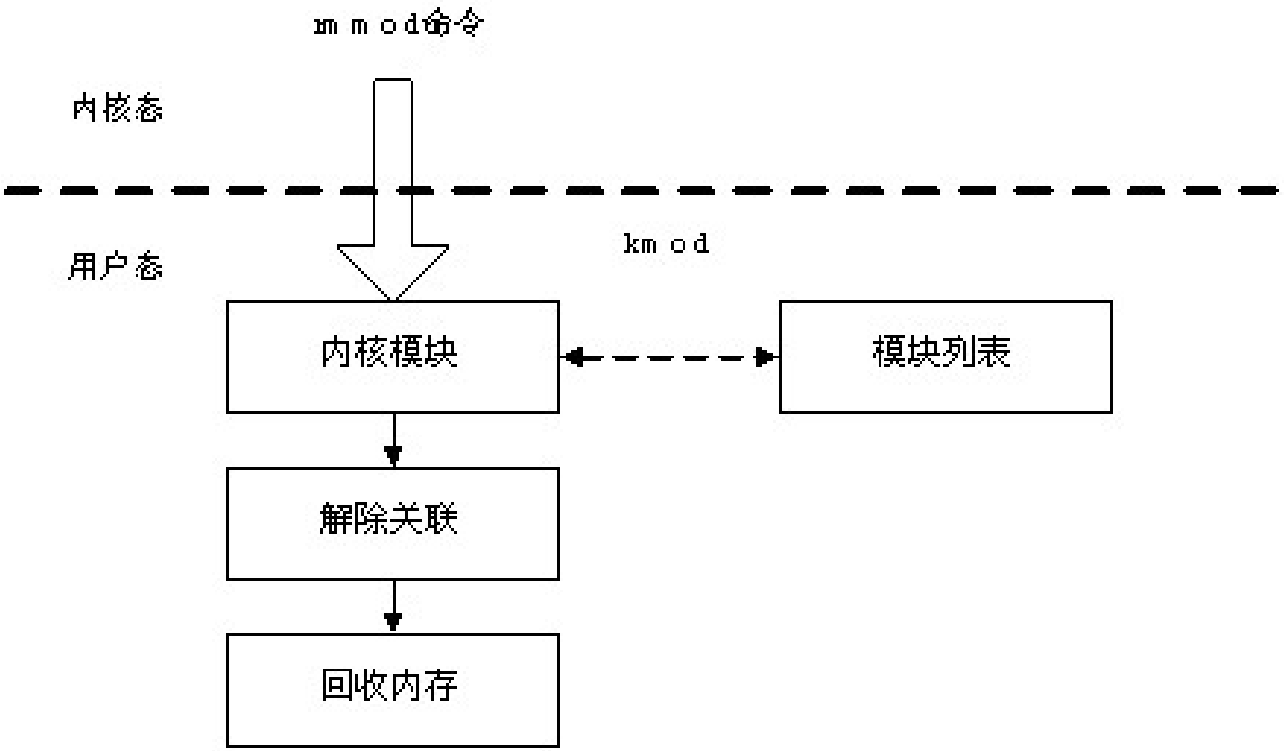
22.2.3 内核模块的加载和卸载

Linux内核提供了一个kmod的模块用来管理内核模块。kmod模块与用户态的kmodule模块通信，获取内核模块的信息。本节介绍一下内核模块的加载和卸载。

1. 内核模块加载
2. 内核模块卸载



22.2.3 内核模块的加载和卸载



22.2.4 编写一个基本的内核模块

第6章给出了一个输出“Hello,World!”的C程序作为基本的编程入门程序。基本的内核模块代码实例

```
1 /* 内核模块: ModuleHelloWorld.c */
2 #include <linux/init.h>
3 #include <linux/module.h>
4 #include <linux/kernel.h>
5
6 MODULE_LICENSE("GPL");
7 MODULE_AUTHOR("GongLei");
8
9 /* init function */
10 static int __init hello_init(void)
11 {
12   printk(KERN_ALERT "(init)Hello,World!\n");
13   return 0;
14 }
15
16 /* exit function */
17 static void __exit hello_exit(void)
18 {
19   printk(KERN_ALERT "(exit)Hello,World!\n");
20 }
21
22 module_init(hello_init);
23 module_exit(hello_exit);
```

22.2.5 编译内核模块

编译内核模块需要建立一个**Makefile**，主要目的是使用内核头文件，因为内核模块对内核版本有很强的依赖关系。

22.2.6 为内核模块添加参数

驱动程序常需要在加载的时候提供一个或者多个参数，内模块提供了设置参数的能力。通过**module_param()**宏可以为内核模块设置一个参数。定义如下：

module_param(参数名称，类型，属性)

22.3 Linux设备驱动

Linux系统把设备驱动分成字符设备、块设备和网络设备三种类型。内核为设备驱动提供了注册和管理的接口，设备驱动还可以使用内核提供的其他功能以及访问内核资源。本节首先介绍一种常见的总线结构，然后分别介绍三种类型设备驱动的特点，并且给出一个字符设备的例子。其他类型的设备驱动在后面章节介绍。

22.3.1 PCI局部总线介绍

早期的计算机有众多总线标准。从最初的8位总线到16位总线，到目前主流的32位总线，不同厂商都制定了自己的总线标准。不同的总线设备给设备驱动的设计带了麻烦，直到后来PCI局部总线出台这种局面才得到缓解，并且逐步成为事实上的标准。

PCI是英文Peripheral Component Interconnect的缩写，中文意思是外设部件互连标准。PCI局部总线标准最早由英特尔公司为制定，最初主要应用在PC机。目前已经被越来越多的嵌入式系统已经其他类型的计算机系统使用。设计PCI的原因是由于之前的总线有许多的缺点，归纳总结为以下几点：

总线速度过慢。

总线地址分配方法复杂。

总线资源共享效率低。

22.3.1 PCI局部总线介绍

函数成员名称	含义
open	在打开设备的时候内核会调用该函数
read	对设备进行读操作的时候内核会调用该函数
write	对设备进行写操作的时候内核会调用该函数
release	当用户关闭设备的时候会内核调用该函数
ioctl	该函数用于向设备传递控制信息或者是获取设备状态
llseek	文件指针定位函数，设置文件读写位置
poll	用于查询设备是否可读写

22.3.2 Linux设备驱动基本概念

在Linux系统中，所有的资源都是作为文件管理的，设备驱动也不例外，设备驱动通常是作为一类特殊的文件存放在/dev目录下。查看/dev目录得到系统所有设备的列表如下：

```
$ ls -l /dev
brw-r----- 1 root   disk      2,  84 Mar 28 14:44 fd0u1040
brw-r----- 1 root   disk      2,  88 Mar 28 14:44 fd0u1120
brw-r----- 1 root   disk      2,  28 Mar 28 14:44 fd0u1440
crw------- 1 root   root      4,   1 Mar 28 14:46 tty1
crw-rw---- 1 root   dialout   4,  10 Mar 28  2009 tty10
crw-rw---- 1 root   dialout   4,  11 Mar 28  2009 tty11
```

22.3.3 字符设备

字符设备是Linux系统最简单的一类设备。应用程序可以像操作普通文件一样操作字符设备。常见的串口、调制解调器都是字符设备。编写字符设备驱动需要使用内核提供的register_chardev()函数注册一个字符设备驱动。函数定义如下：

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

22.3.4 块设备

与字符设备相比，块设备要复杂的多。最主要的差别是块设备带有缓冲，字符设备没有。块设备传输数据只能以块作为单位读写，字符设备是以字节作为最小读写单位的。块设备对于I/O请求有对应的缓冲区，可以选择响应的顺序，如采用特定的调度策略等；字符设备职能顺序访问。此外，块设备提供了随机访问的能力，而字符设备之顺序读取数据。

块设备提供了一个类似字符设备的访问函数结构**block_device_operations**，定义如下：

```
struct block_device_operations {
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
long (*compat_ioctl) (struct file *, unsigned, unsigned long);
int (*direct_access) (struct block_device *, sector_t, unsigned long *);
int (*media_changed) (struct gendisk *);
int (*revalidate_disk) (struct gendisk *);
int (*getgeo)(struct block_device *, struct hd_geometry *);
struct module *owner;
};
```

22.3.5 网络设备

在Linux内核中，网络设备是一类特殊的设备，因此被单独设计为一种类型的驱动。与其他设备不同的是，网络设备不是通过设备文件访问的，在/dev目录下不会看到任何网络设备。因此，网络设备的操作不是通过文件操作实现的。

Linux内核为抽象了网络设备界面，为其定义了一个接口用于屏蔽网络环境下各种网络设备的差别。内核对所有网络设备的访问都通过这个抽象的接口，接口对上层网络协议提供相同的操作方法。有关网络设备详细的知识请参考第23章。

22.4 字符设备驱动开发实例

在Linux内核驱动中，字符设备是最基本的设备驱动。字符设备包括了设备最基本的操作，如打开设备、关闭设备、I/O控制等。学习其他设备驱动最好从字符设备开始。本节给出一个字符设备驱动示例，帮助读者学习字符设备驱动开发。

22.4.1 开发一个基本的字符设备驱动

首先给出一个字符设备实例，功能是建立一个名为“GlobalChar”的虚拟设备，设备内部只有一个全局变量供用户操作。

24.4.2 测试字符设备驱动

为了测试编写的字符设备是否能正常工作，可以编写一个应用程序测试一下能否正常读写字符设备。

22.5 小结

本章介绍了Linux驱动开发的基础知识。包括如何开发内核模块、Linux设备驱动原理、驱动程序分类等，并且给出了一个字符设备驱动开发实例。内核模块是开发设备驱动的基础，因此必须掌握。在后面的几章会详细讲解Linux设备驱动开发。第23章讲解网络设备驱动开发。

第23章　网络设备驱动程序

计算机与外界通信是通过网卡完成的。网卡包括了网络控制器和网络接口两部分，网卡不仅是一个网络数据收发的设备，还肩负网络底层协议处理任务。网络设备在Linux内核是一类复杂的设备，在学习网卡驱动的时候需要掌握网络和内核协议栈的基本知识。本章从网络基本知识入手，逐步介绍网络协议和内核协议栈，最后讲解网卡驱动编程，并且给出了嵌入式系统常见的DM9000网卡驱动分析。主要内容如下：

OSI网络参考模型

TCP/IP协议入门

以太网工作原理

内核网络设备驱动

网络数据包在内核中的处理流程

23.1 网络基础知识

网卡是使用网络的必备设备之一。网卡的主要功能是处理网络上的数据，在学习网卡驱动之前需要掌握必要的网络知识。本节介绍了网络的基本模型和参考结构，然后介绍了使用最广泛的TCP/IP协议，最后介绍了以太网的知识。

23.1.1 ISO/OSI网络参考模型

世界上目前有多种不同架构的网络，目前最流行的互联网就是一种多架构的网络集合。多种不同架构的网络互联的根本问题是不同网络架构术语之间的统一，OSI网络参考模型正是为解决这个问题提出的。

OSI网络参考模型全称是开发系统互联参考模型（Open System Interconnection Reference Model），是由国际标准化组织ISO提出的一个网络互联模型。虽然目前没有一个网络是完全按照OSI网络参考模型设计的，但是该模型对网络协议之间的互联起到很大作用。

OSI网络参考模型是一个逻辑结构，采用分层的概念划分网络，任何两种网络协议只要采用OSI网络参考模型设计都能相互通信。计算机网络通信是一个复杂的过程，OSI采用的分层思想简化了网络的设计。分层是一种构造技术，通过明确定义每一层的功能规范了网络数据传输。

23.1.1 ISO/OSI网络参考模型

23.1.2 TCP/IP协议

目前最应用最广泛的网络就是互联网了，互联网采用了TCP/IP协议作为通信协议。TCP/IP协议是由许多协议组成的协议簇，其中最主要的就是TCP（传输控制）协议和IP（互联网）协议。TCP/IP协议最早由美国国防高级研究计划署在ARPANET上实现，随着不断的发展成为了目前互联网使用最广泛的协议，已经成为计算机网络通信事实上的标准协议。

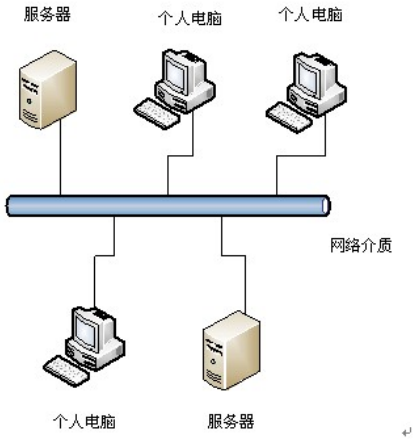
1. TCP/IP协议简介
2. TCP/IP协议分层结构
3. 网络接口层协议
4. 网络层
5. 传输层
6. 应用层

23.1.2 TCP/IP协议

TCP/IP协议分层	OSI网络参考模型
应用层	应用层
	表示层
	会话层
传输层	传输层
网络层	网络层
网络接口层	数据链路层
	物理层

23.2 以太网基础

以太网是目前局域网使用最广泛的通信标准，最初由施乐公司提出。以太网是一个技术标准而不是一种具体的网络。该标准定义了在局域网（LAN）内使用的电缆类型和信号处理方法。最初的以太网设备之间使用10Mbps速率传递数据包，目前最高的以太网速率已经能达到10Gbps。许多厂商都开发了支持以太网的软件和硬件，因此，以太网是开发性最好的局域网标准。



23.2.1 工作原理

以太网中一个节点发送数据的工作过程如下：

（1）监听网络是否有信号在传输，如果有信号，表示网络处于繁忙状态，则继续监听，直到网络空闲为止。

（2）如果没有检测到网络上的传输信号则发送数据。

（3）在传输数据过程中继续监听，如果发现网络有信号冲突（其他节点也发送数据），则执行退避算法。退避算法会随机等待一段时间，重复执行步骤1。

（4）如果发送过程中没有冲突，则数据包发送成功，在发送下一个数据包前必须延迟一个固定时间（每种网络有自身的规定）后才可以执行步骤1。

23.2.2 常见以太网标准

从以太网的产生到目前已经产生了多种以太网标准，每种标准之前最大的差异就是传输速度的提高。下面介绍几种常见的以太网标准。

1. 标准以太网
2. 快速以太网
3. 千兆以太网
4. 万兆以太网

23.2.3 拓扑结构

以太网支持总线型和星型拓扑结构。总线型结构特点是使用电缆少、价格便宜，但是管理成本高，网络故障不易定位。此外，总线型网络采用共享访问机制，容易造成网络拥塞。早期的以太网使用同轴电缆作为传输介质，通常使用总线结构，主要是便于连接。总线型网络适合规模小的网络，并且网络中的节点很少变动。

星型结构的特点是管理方便、容易扩展、网络故障容易定位，但是需要专用的网络设备作为网络交换核心，并且需要更多的网线，成本较高。星型网络使用双绞线作为传输介质，需要集线器或者交换机作为网络核心节点，通过双绞线把所有节点连接到核心节点构成星型网络结构。星型网络枷弒总线型网络简单，并且可以通过网络级联扩展网络容量。

23.2.4 工作模式

以太网中最基本的设备就是以太网卡了，以太网卡可以在半双工和全双工模式下工作。半双工模式基于以太网的CSMA/CD机制工作。传统的以太网使用半双工模式，在同一时间只能一个方向传输数据，当两个或两个以上节点传输数据的时候会导致网络产生数据冲突，降低了网络效率。

全双工模式使用点对点连接方式，这种方式没有冲突产生。全双工使用双绞线中两个独立的线路，也就是说发送和接收在不同的线路上进行，在没有安装介质的情况下就可以提高带宽。在全双工模式下，冲突检测电路不再使用，因此一个全双工连接只用一个端口，在连接的双方向都提供了100%的效率。

23.2 网卡工作原理

网卡的全程是网络适配器或者网络接口卡（Network Interface Card，NIC），是计算机联网的必备设备。通常说的网卡是PC机上的概念，网卡包括了网络控制芯片和网络接口两部分组成。在嵌入式系统中，通常把网络接口芯片和网络接口与其他芯片都安装在同一块线路板上，甚至有的CPU内部集成了网络控制器，仅需要在外部引出网络接口即可。

作为网络通信的一个概念，本书把网卡作为一个整体部分介绍。在计算机系统中，网卡负责把用户传递的数据转换为网络能识别的电信号，并且把网络上电信号还原成数据传递给用户。网卡的技术参数涉及到带宽、总线接口、电气接口等。网卡还提供了网络数据包的存取控制、数据缓存等功能。

23.3 内核网络分层结构

Linux内核对网络驱动程序使用统一的接口，并且对于网络设备采用面向对象的思想设计。Linux内核采用分层结构处理网络数据包。分层结构与网络协议的结构匹配，既能简化数据包处理流程，又便于扩展和维护。

23.3.1 内核网络结构

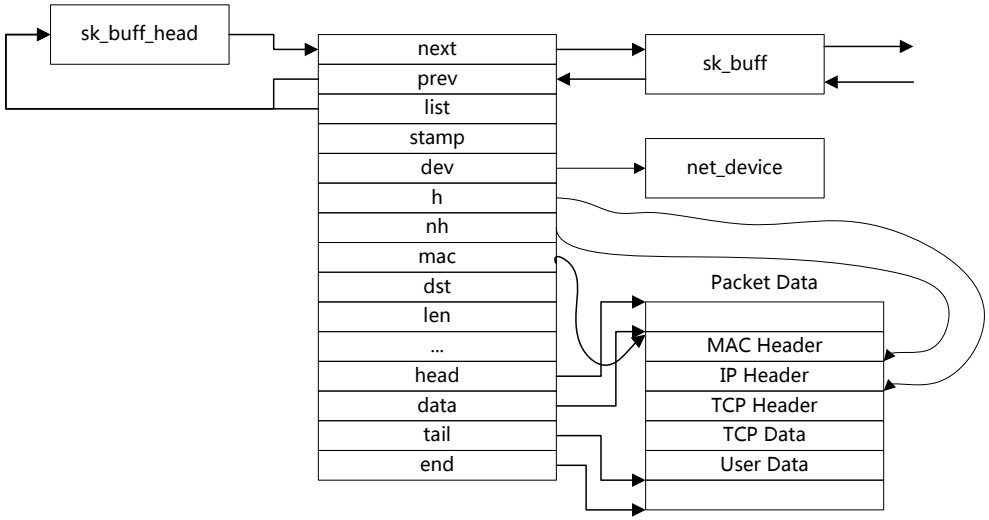
在Linux内核中，对网络部分按照网络协议层、网络设备层、设备驱动功能层和网络媒介层的分层体系设计。

网络驱动功能层主要通过网络驱动程序实现。在Linux内核，所有的网络设备都被抽象为一个接口处理，该接口提供了所有的网络操作。前面提到过的net_device结构表示网络设备在内核中的情况，也就是网络设备接口。网络设备接口既包括软件虚拟的网络设备接口，如环路设备，也包括了网络硬件设备，如以太网卡。

Linux内核有一个dev_base的全局指针，指向一个设备链表，包括了系统内的所有网络设备。该设备链表每个节点是一个网络设备。在net_device结构中提供了许多供系统访问和协议层调用的设备方法，包括初始化、打开关闭设备、数据包发送和接收等。

23.3.2 与网络有关的数据结构

内核对网络数据包的处理都是基于**sk_buff**结构的，该结构是内核网络部分最重要的数据结构。网络协议栈中各层协议都可以通过对该结构的操作实现本层协议数据的添加或者删除。使用**sk_buff**结构避免了网络协议栈各层来回复制数据导致的效率低下。



23.3.3 内核网络部分的全局变量

Linux内核网络部分有几个重要的全局变量，网络接口和协议都使用这些全局变量完成网络功能。

与协议有关的全局变量

与包类型有关的全局变量

网络设备接口中的全局变量

23.4 内核网络设备驱动框架

在Linux内核网络设备是一类特殊的设备。网络设备虽然借用了传统设备（字符设备/块设备）的一些概念，却有其自身特点。对用应用程序来说，访问网络设备不需要通过文件句柄，而是通过socket网络接口。因此，网络设备结合了设备驱动和内核网络协议，结构和工作流程都比较复杂。

23.4.1 net_device结构

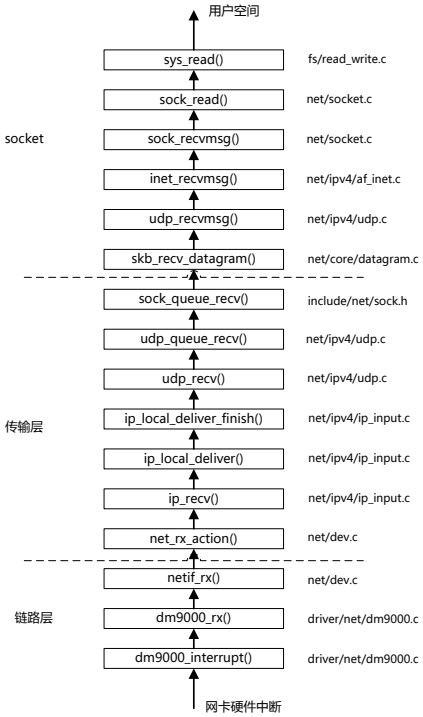
Linux内核中网络设备最重要的数据结构就是net_device结构了，它是网络驱动程序最重要的部分。net_device结构保存在include/linux/netdevices.h头文件，理解该结构对理解网络设备驱动有很大帮助。net_device结构十分庞大，书中不列出结构的代码，仅给出结构的解释，请读者自己参考Linux2.6.18内核代码。

内核中所有网络设备的信息和操作都在net_device设备中，无论是注册网络设备，还是设置网络设备参数，都用到该结构。下面介绍主要数据成员。

1. 设备名称
2. 总线参数
3. 协议参数
4. 链接层变量
5. 接口标志

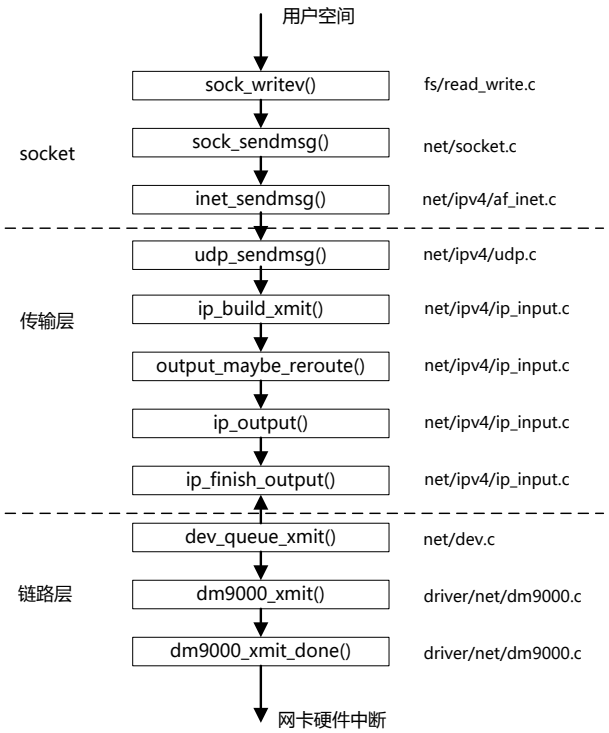
23.4.2 数据包接收流程

在Linux内核中，一个网络数据包从网卡接收到用户空间需要经过链路层、传输层和socket的处理，最终到达用户空间。



23.4.3 数据包发送流程

本节还是以UDP数据包发送流程为例，介绍在DM9000网卡上如何发送一个数据包。



23.5 实例：DM9000网卡驱动分析

DM9000是嵌入式系统中常用的一款网络控制芯片，提供了丰富的功能和开发接口，具有很高的性价比，因此得到广泛应用。本节介绍**DM9000**网卡芯片的工作原理以及在**Linux**下的驱动分析。

23.5.1 DM9000芯片介绍

DM9000是DAVICOM公司的一款高度集成低功耗快速以太网处理器，该芯片集成了MAC和PHY。DM9000可以和CPU直接连接，支持8位、16位和32位数据总线宽度。该芯片支持10M和100M自适应以太网接口，内部有16K的FIFO以及4K双字节SRAM，支持全双工工作。

DM9000内部还集成了接收缓冲区，可以在接收到数据的时候把数据存放到缓冲区中，链路层可以直接把数据从缓冲区取走。与其他的网卡控制芯片相比，DM9000从硬件连接到驱动都相对简单适合初学者学习。

23.5.2 网卡驱动程序框架

在一个网络驱动程序中，一般都提供了一个platform_driver结构变量。platform_driver结构包括了网卡驱动的相关操作函数，通过platform_driver_register()函数注册到内核设备驱动列表。内核会根据驱动程序中设备描述设置网卡的中断和定时器，并且在网络数据包到来的时候调用网卡对应的处理函数。

通常，网卡需要向内核提供下面几个接口函数：

probe：

remove：

suspend：

resume：

23.5.3 DM9000网卡驱动主要数据结构

DM9000网卡驱动位于driver/net/dm9000.c文件，有两个主要的数据结构dm9000_driver和board_info。其中，dm9000_driver是platform_driver结构，定义如下：

```
1204 static struct platform_driver dm9000_driver = {
1205   .driver = {
1206     .name    = “dm9000”,
1207     .owner   = THIS_MODULE,
1208   },
1209   .probe   = dm9000_probe,
1210   .remove  = dm9000_drv_remove,
1211   .suspend = dm9000_drv_suspend,
1212   .resume  = dm9000_drv_resume,
1213 };
```

23.5.4 加载驱动程序

在dm9000.c文件的第1229行和1230行使用模块加载宏和卸载宏设置了模块的初始化函数dm9000_init()和卸载函数dm9000_cleanup()。

23.5.5 停止和启动网卡

停止网卡是在用户使用**ifdown**命令设置网卡暂时停止，用户的命令通过系统调用最终会调用网卡驱动的停止函数，对于**DM9000**网卡驱动来说是**dm9000_stop()**函数：

23.5.6 发送数据包

网卡驱动程序需要向内核提供两个发送数据包的回调函数，一个用于发送数据包，一个用于数据包发送完毕后的处理。

23.5.7 接收数据包

DM9000向内核提供了dm9000_rx()函数，在内核收到DM9000网络控制器的接收数据包中断后被内核调用。dm9000_rx()函数使用了一个自定义的dm9000_rxhdr结构，该结构与DM9000网络控制器提供的数据包接收信息对应。dm9000_rx()函数定义如下：

23.5.8 中断和定时器处理

网络设备驱动需要提供中断处理函数和定时处理函数供内核使用。中断处理函数当网络控制器向CPU发出中断后，由内核中断处理函数调用。定时器处理函数是由内核的一个定时器周期的调用。

23.6 小结

在网络通信中，计算机通过网卡（包括网络控制器和网络接口）与其他网络节点通信。由于不同的网络有不同协议，网卡的设计不仅需要兼顾到网络上数据包的处理，还涉及到主机网络协议栈的接口。网卡驱动在Linux内核是一类复杂的设备驱动，读者在学习网卡设备驱动的时候要立足从网络协议入手，在了解了网络协议和内核协议栈工作流程的后学习网卡驱动编写会比较容易入手。

第24章　Flash设备驱动

Flash存储器是近几年来发展最快的存储设备，通常也称作闪存。Flash属于EEPROM（电可擦除可编程只读存储器），是一类存取速度很高的存储器。它既有ROM断电可保存数据的特点，又有易于擦写的特点。Flash可以在断电的情况下长期保存信息，因此被广泛的应用在PC机的BIOS和嵌入式系统的存储设备。本章主要内容如下：

Flash存储器的硬件结构和存储原理

Linux内核MTD设备支持

Flash编程框架

Flash驱动实例

24.1 Linux Flash驱动结构

Linux内核对Flash存储器有很好的支持。内核设计了一个MTD结构支持Flash设备，用户只需要按照MTD的要求设置Flash设备的参数并且提供驱动就可以让Flash设备很好的工作。本节介绍内核MTD的系统结构。

24.1.1 什么是MTD

MTD是英文Memory Technology Device的缩写，中文可以直接译为内存技术设备。MTD是Linux内核为支持闪存设备的一个驱动中间层。对内核其他部分来说，MTD屏蔽了闪存设备的细节；对于闪存设备驱动来说，只需要向MTD中间层提供接口就可以向内核提供闪存设备支持。Linux内核提供了一些与MTD相关的术语，解释如下：

JEDEC：J

CFI：

OOB：

ECC：

EraseSize：

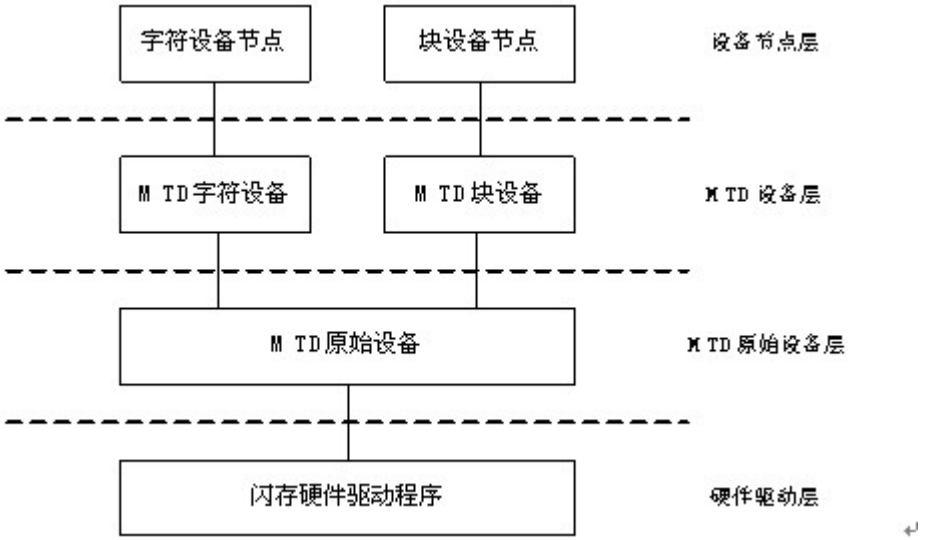
BusWidth：

NAND：

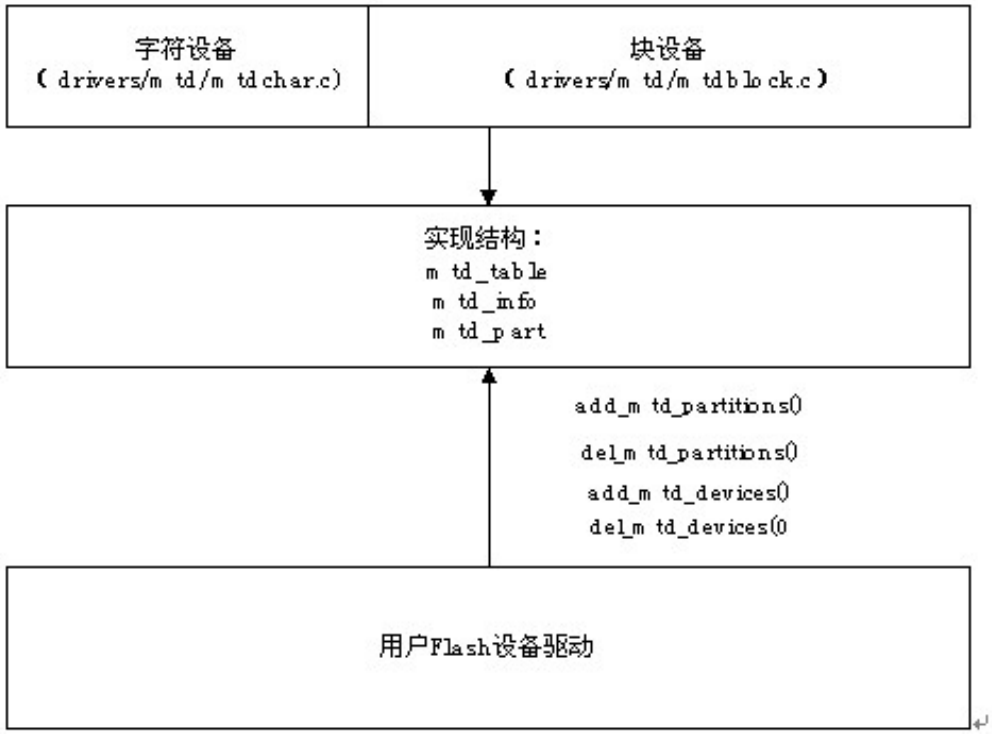
NOR：

24.1.2 MTD系统结构

Linux内核MTD设备相关代码在drivers/mtd目录下，设计MTD的目的是让新的闪存设备使用更简单。MTD设备可以分层四层。



24.1.2 MTD系统结构

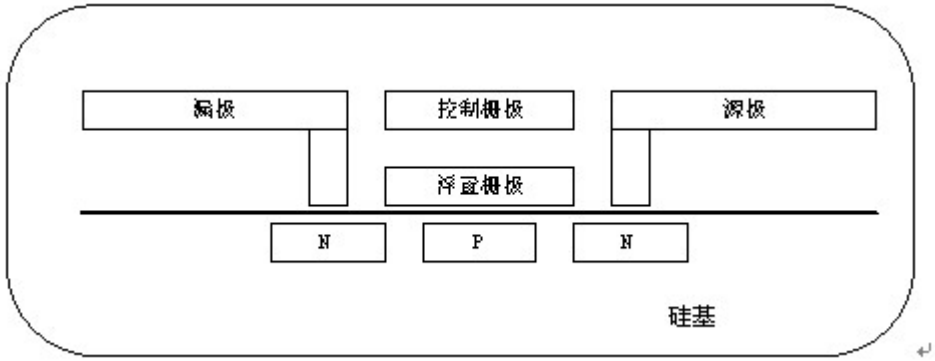


24.2 Flash设备基础

NAND和NOR是两种不同的Flash存储技术，它们各有不同，适合不同的工作范围。编写一个闪存设备的驱动不仅需要了解MTD的结构，还需要知道闪存设备的硬件原理。本机介绍两种不同的闪存工作原理，并且比较它们之间的异同。

24.2.1 存储原理

NAND和NOR闪存都使用三端器件作为存储单元，学过模拟电子技术的读者可能了解一种叫做场效应管的器件，与此原理类似。三端器件分别有源极、漏极和栅极，栅极利用了电场效应控制源极于漏极之间的通断。



24.2.2 性能比较

NOR Flash技术最早是Intel公司研发出来的，紧接着东芝公司研发出了NAND Falsh技术。NOR闪存的特点是支持片内执行（Execute In Place），应用程序不必加载到RAM就可以直接运行，简化了软件开发。NOR闪存的读取速率非常高，但是容量通常不大，因为容量大成本会显著增大。此外，NOR闪存的写入和擦除速度非常慢，不利于大量数据存储。

NAND Flash特点是存储密度高，写入和擦除速度都比NOR Flash快，适合大数据量存储。但是，NAND Flash需要特殊的存储电路控制，并且空的或者已经擦除的单元才能写入数据，所以必须在写入数据之前先擦除块。

24.3 内核MTD层

由于NOR和NAND Flash的物理特性差异，Linux内核设计了MTD层用于管理两种不同类型的Flash设备。MTD层对内核空间其他部分屏蔽了Flash的差异，有几个比较重要的数据结构，在设计闪存驱动的时候需要了解，本节以Linux内核2.6.18版本为例介绍。

24.3.1 mtd_info结构

mtd_info结构是MTD原始设备层的一个重要结构，该结构定义了大量的关于MTD的数据和操作，定义在include/linux/mtd/mtd.h头文件。mtd_info结构成员主要由数据成员和操作函数两部分组成。

24.3.2 mtd_part结构

mtd_part结构描述分区，该结构中的mtd_info结构成员用于描述本分区，加入到mtd_table全局变量中。结构定义在drivers/mtd/mtdpart.c。

```
27 /* Our partition node structure */
28 struct mtd_part {
29   struct mtd_info mtd;
30   struct mtd_info *master;
31   u_int32_t offset;
32   int index;
33   struct list_head list;
34   int registered;
35 };
```

24.3.3 mtd_partition结构

内核代码drivers/mtd/mtdpart.c文件内定义了添加和删除闪存分区的函数如下：

int add_mtd_partitions(struct mtd_info *master, const struct mtd_partition *parts, int nbparts);

int del_mtd_partitions(struct mtd_info *master);

24.3.4 map_info结构

NOR Flash驱动使用map_info结构作为核心数据结构。该结构定义了NOR Flash的基址、位宽、大小等信息以及闪存的操作函数。

24.3.5 nand_chip结构

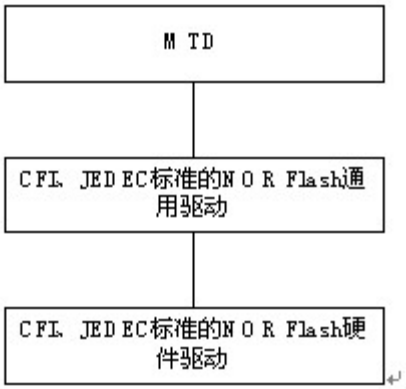
NAND闪存使用nand_chip结构描述存储设备信息，该结构定义在include/linux/mtd/nand.h头文件：

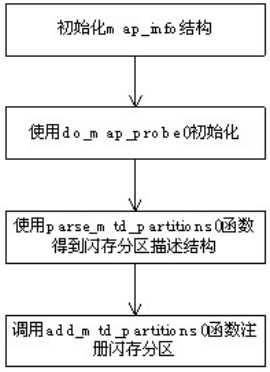
24.4 Flash设备框架

由于NOR Flash和NAND Flash设备的差异，内核对这两种类型的Flash设备设计了不同的驱动和管理方式。NOR Flash的操作相对要简单一些，NAND Flash不仅需要提供读写函数，还需要提供ECC校验函数和坏块管理函数。

24.4.1 NOR Flash设备驱动框架

Linux内核提供了map_info结构描述NOR Flash设备，驱动程序围绕该结构操作，通过内核提供的注册函数把芯片的信息提交给内核，并且提供必要的操作函数。



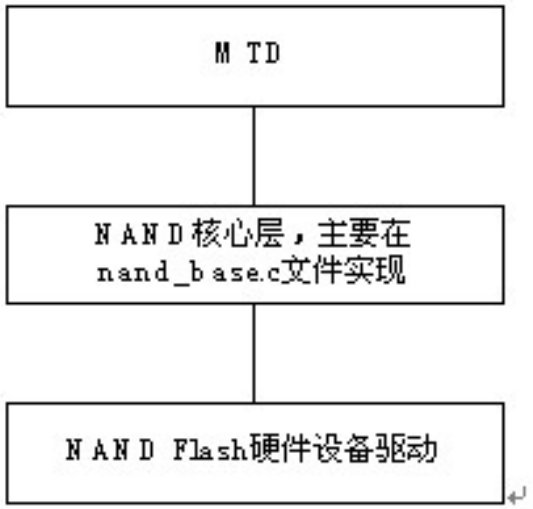


NOR Flash驱动程序注册流程

NOR Flash驱动程序卸载流程

24.4.2 NAND Flash设备驱动框架

NAND Flash设备使用nand_chip结构描述。



24.5 Flash设备驱动实例——NOR Flash设备驱动剖析

在介绍了Linux内核对MTD的支持以后，本节给出两个Flash设备驱动实例，并且分析驱动的工作原理。由于Flash设备具有硬件平台无关的特点，本节介绍的两种Flash设备驱动适应于不同的硬件平台。

NOR Flash设备驱动一般来说比较简单，主要因为NOR Flash硬件的操作比较简单，可以像操作内存一样使用，因此驱动程序无需提供太多功能。本节以内核2.6.18版本代码drivers/mtd/maps/sharpsl-flash.c文件为例分析NOR Flash驱动程序。

24.5.1 数据结构

该驱动程序是SHARP某种设备驱动，程序比较简单。首先在39行定义了一个mtd_info结构用于描述NOR设备：

```
39 struct map_info sharpsl_map = {
40   .name = "sharpsl-flash",
41   .size = WINDOW_SIZE,
42   .bankwidth = BANK_WIDTH,
43   .phys = WINDOW_ADDR
44 };
```

24.5.2 驱动初始化

通过查找模块初始化和清除宏能得到该驱动的初始化函数为**init_sharpsl()**以及清除函数**cleanup_sharpsl()**。

24.5.3 驱动卸载

驱动卸载调用了cleanup_sharpsl()函数，主要是释放NOR驱动占用的资源，函数定义如下：

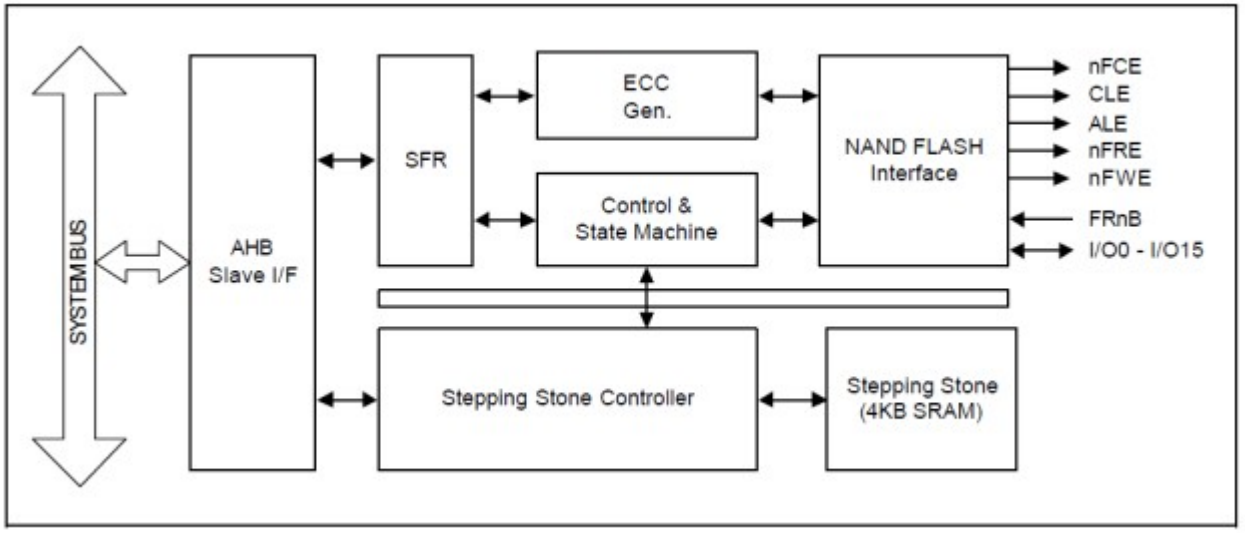
```
101 static void __exit cleanup_sharpsl(void)
102 {
103   if (mymtd) {
104     del_mtd_partitions(mymtd);
105     map_destroy(mymtd);
106   }
107   if (sharpsl_map.virt) {
108     iounmap(sharpsl_map.virt);
109     sharpsl_map.virt = 0;
110   }
111 }
```

24.6 Flash设备驱动实例——NAND Flash设备驱动分析

NAND Flash由于需要控制器才能操作，驱动程序相对复杂。在NAND Flash驱动程序中，主要工作是配置NAND控制器。内核2.6.18版本代码drivers/mtd/nand/s3c2410.c文件包含了S3C2410、S3C2412和S3C2440平台上NAND Flash驱动。本节重点分析S3C2440平台上NAND Flash驱动程序。

24.6.1 S3C2440 NAND控制器介绍

S3C2440芯片内部集成了NAND Flash控制器，在芯片手册Figure 6-1给出了一个NAND Flash控制器的功能框图。



24.6.2 数据结构

S3C2410、S3C2412和S3C2440这三个处理器在NAND控制器部分存在一些差异。为了在三款处理器上都可以驱动同样的NAND Flash闪存，在drivers/mtd/nand/s3c2410.c文件中定义了s3c2410_nand_mtd结构和s3c2410_nand_info结构存放相关信息。s3c2410_nand_mtd结构定义如下：

```
92 struct s3c2410_nand_mtd {
93   struct mtd_info     mtd;          // 内核mtd_info结构
94   struct nand_chip    chip;      // 内核nand_chip结构
95   struct s3c2410_nand_set   *set;    // 封装后的mtd_partition结构，加入了s3c2440的一些属性
96   struct s3c2410_nand_info  *info; // s3c2440状态信息，私有数据
97   int       scan_res;
98 };
```

24.6.3 驱动初始化

驱动初始化调用s3c2410_nand_init()函数，定义如下：

```
796 static int __init s3c2410_nand_init(void)
797 {
798   printk("S3C24XX NAND Driver, (c) 2004 Simtec Electronics\n");
799
800   platform_driver_register(&s3c2412_nand_driver);
801   platform_driver_register(&s3c2440_nand_driver); // 注册S3C2440平台驱动数据结构
802   return platform_driver_register(&s3c2410_nand_driver);
803 }
```

24.6.4 驱动卸载

驱动卸载时调用s3c2410_nand_remove()函数，主要是释放驱动占用的资源。

24.6.5 初始化NAND控制器

s3c2410_nand_inithw()函数负责初始化NAND控制器。

24.6.6 设置芯片操作

s3c2410_nand_init_chip()函数初始化nand_chip结构，并且设置相关的操作，函数操作流程如下：

（1）程序508~514行设置通用的读写缓冲函数、查找芯片函数等。

（2）然后在函数516行使用switch语句根据不同平台设置NAND控制器的配置。

（3）设置完毕后，在548~551行设置mtd_info结构和nand_chip结构之间的相互指向关系。

（4）最后，程序553~576行设置ECC相关操作。

24.6.7 电源管理

S3C2440平台上NAND闪存的电源管理包括s3c24xx_nand_suspend()函数和s3c24xx_nand_resume()函数。

24.7 小结

本章讲解了Flash存储器相关的硬件知识和Linux对Flash设备的支持，包括Flash的硬件结构、工作原理以及MTD的系统结构。本章给出了两种类型Flash设备的驱动程序框架，并且给出实际驱动实例。理解硬件驱动开发的关键是了解硬件工作原理，读者在理解Flash硬件工作原理后理解Flash设备驱动会更容易。第25章讲解USB设备驱动。

第25章　USB驱动开发

USB是目前最流行的系统总线之一。随着计算机周围硬件的不断扩展，各种设备使用不同的总线接口，导致计算机外部总线种类繁多，管理困难。USB总线正是因此而诞生的。USB总线提供了所有外部设备的统一连接方式，并且支持热插拔，方便了厂商开发设备和用户使用设备。本章详细介绍USB相关的知识，主要内容如下：

USB总线体系结构介绍

USB体系工作流程

Linux内核如何实现USB体系

USB设备驱动开发实例

25.1 USB体系介绍

USB是英文Universal Serial Bus的缩写，中文称为“通用串行总线”。USB是一个总线协议标准，最初由Intel、NEC、Compaq、DEC、IBM、Microsoft等公司联合制定的。到目前为止USB共有1.0、1.1和2.0三个标准，主要区别是传输速率不同，体系结构也有一定的差别。

25.1.1 USB设计目标

USB的设计目标是对现有的PC机体系进行扩充，但是目前不仅是PC机，许多的嵌入式系统都开始支持USB总线和接口标准。USB设计主要遵循下面几个原则。

易于扩充外部设备：

灵活的传输协议：

设备兼容性好：

接口标准统一：

USB接口标准具有下面几个方面的特色：

1. 易用性
2. 应用广泛
3. 健壮性

25.1.1 USB设计目标

分类	传输率	应用	特点
低速设备	10-20kbps	键盘、鼠标等输入设备	易用、支持热插拔、价格低
中速设备	500kbps-10Mbps	宽带网络接入设备	易用、支持热插拔
高速设备	25-500Mbps	音视频设备、磁盘	易用、带宽高、支持热插拔

25.1.2 USB体系概述

USB接口标准支持主机和外部设备之间进行数据传输。在USB体系结构中，主机预定了各种类型外部设备使用的总线带宽。当外部设备和主机在运行时，USB总线允许添加、设置、使用和拆除外设。

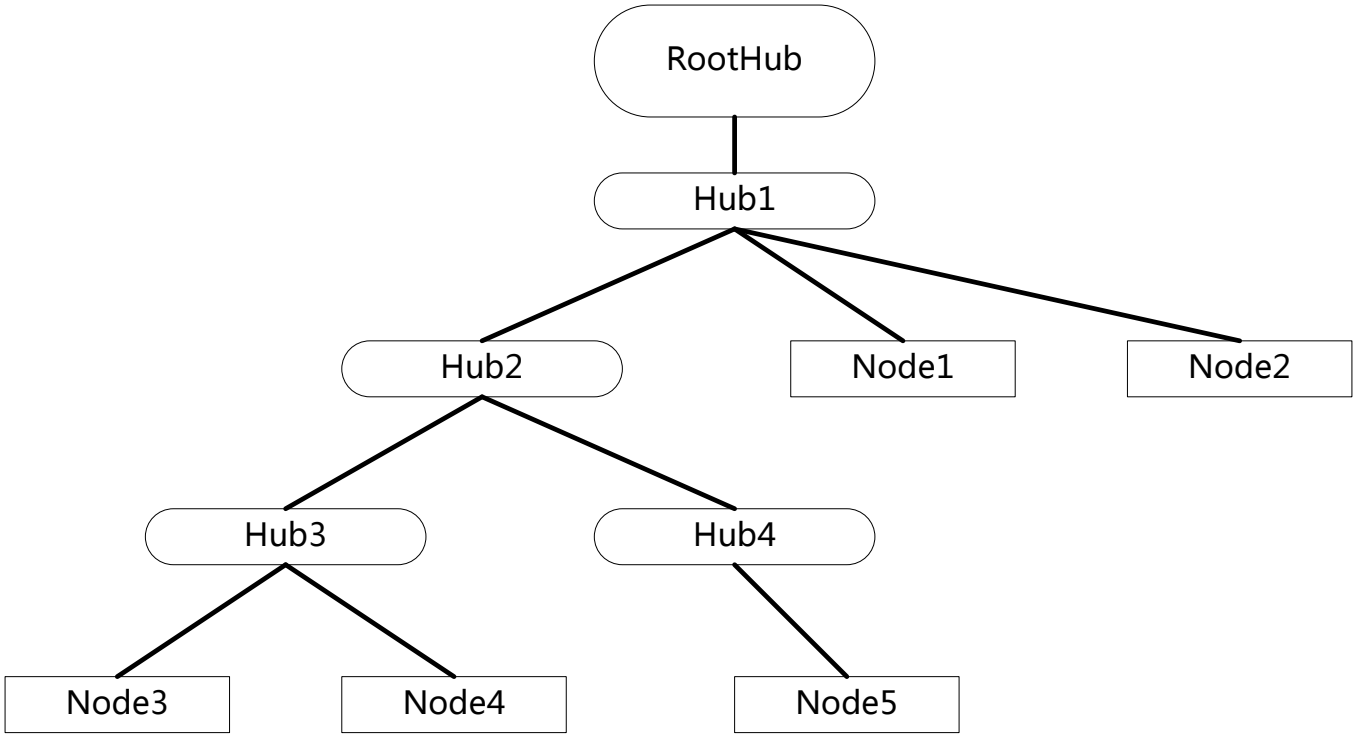
在USB体系结构中，一个USB系统可以分成USB互联、USB设备和USB主机三个部分。USB互联是USB设备和USB主机之间进行连接通信的操作，主要包括：

总线拓扑结构：

数据流模式：

USB调度：

25.1.2 USB体系概述



25.1.3 USB体系工作流程

USB总线采用轮询方式控制，主机控制设置初始化所有的数据传输。USB总线每次执行传输动作最多可以传输三个数据包。每次开始传输时，主机控制器发送一个描述符描述传输动作的种类和方向，这个数据包称作标志数据包（Token Packet）。USB设备收到主机发送的标志数据包后解析出数据自己的数据。

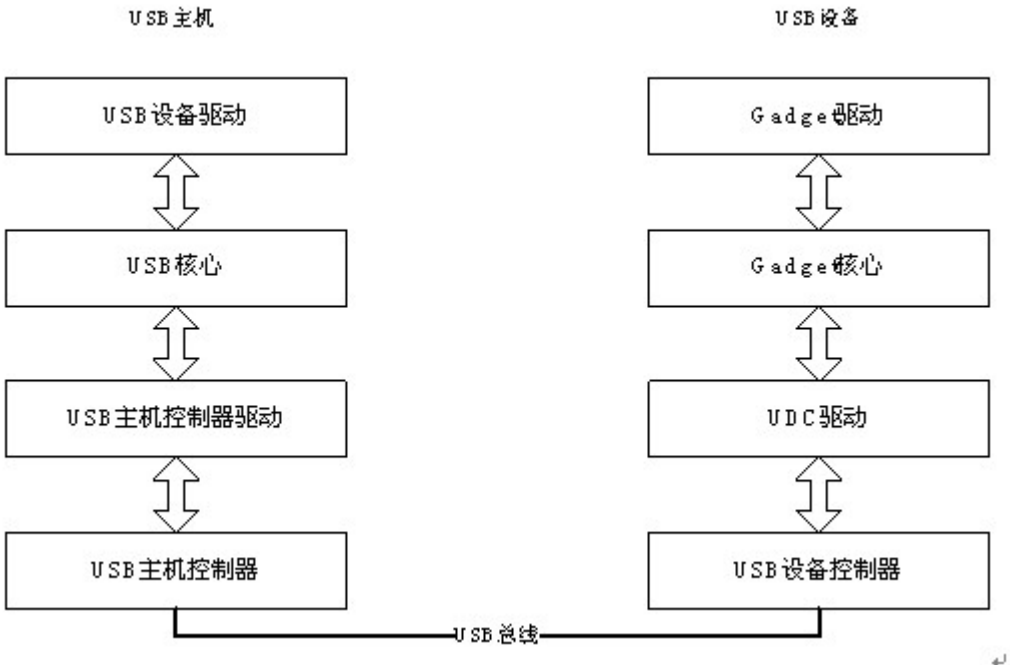
USB数据传输的方向只有两种：主机到设备或者设备到主机。在一个数据传输开始时，由标志包标示数据的传输方向，然后发送端开始发送包含信息的数据。接收端发送一个握手的数据包表明数据是否传送成功。在主机和设备之间的USB数据传输可以看做一个通道。USB数据传输有流和消息两种通道。消息是有格式的数据，而流是没有数据格式的。USB有一个缺省的控制消息通道，在设备启动的时候被创建，因此设备的设置查询和输入控制信息都可以使用缺省消息控制通道完成。

25.2 USB驱动程序框架

Linux内核提供了完整的USB驱动程序框架。USB总线采用树形结构，在一条总线上只能有唯一的主机设备。Linux内核从主机和设备两个角度观察USB总线结构。本节介绍Linux内核USB驱动程序框架。

25.2.1 Linux内核USB驱动框架

在本节开始介绍了**Linux**内核从主机和设备两个角度观察**USB**总线。



25.2.2 主机驱动结构

USB主机控制器有三种类型：

OHCI，英文全称是Open Host Controller Interface。OHCI是用于SiS和Ali芯片组的USB控制器。

UHCI，英文全称是Universal Host Controller Interface。UHCI用于Intel和AMD芯片组的USB控制器。UHCI类型的控制器比OHCI控制器硬件结构要简单，但是需要额外的驱动支持，因此从理论上说速度要慢。

EHCI是USB2.0规范提出的一种控制器标准，可以兼容UHCI和OHCI。

1．USB主机控制器驱动

2．OHCI控制器驱动

25.2.3 设备驱动结构

USB协议规定了许多种USB设备类型。Linux内核实现了音频设备、通信设备、人机接口、存储设备、电源设备、打印设备等几种USB设备类。

1. 基本概念
2. 设备驱动结构
3. USB请求块

25.2.4 USB驱动程序框架

Linux内核代码driver/usb/usb-skeleton.c文件是一个标准的USB设备驱动程序。编写一个USB设备的驱动可以参考usb-skeleton.c文件，实际上，可以直接修改该文件驱动新的USB设备。下面以usb-skeleton.c文件为例分析usb-skel设备驱动框架。

1. 基本数据结构
2. 驱动程序初始化和注销
3. 设备初始化
4. 设备注销

25.3 USB 驱动实例剖析

USB体系支持多种类型的设备。在Linux内核，所有的USB设备都使用usb_driver结构描述。对于不同类型的USB设备，内核使用传统的设备驱动模型建立设备驱动描述，然后映射到USB设备驱动，最终完成特定类型的USB设备驱动。

25.3.1 USB串口驱动

USB串口驱动关键是向内核注册串口设备结构，并且设置串口的操作。下面是一个典型的USB设备驱动分析。

1. 驱动初始化函数
2. 驱动释放函数
3. 串口操作函数

25.3.2 USB键盘驱动

USB键盘驱动与串口驱动结构类似，不同是的使用USB设备核心提供的usb_keyboard_driver结构作为设备核心结构。下面讲解USB键盘驱动重点部分。

1．驱动初始和注销

2．设备检测函数

3．设备断开连接函数

25.4 小结

本章讲解了Linux内核USB驱动体系结构、USB设备驱动结构等知识，并在最后给出了两个USB设备驱动开发实例。USB是目前流行的总线接口之一，物理接口简单，但是协议和操作非常复杂。读者在学习USB设备驱动开发过程中，掌握好USB的体系结构，会得到事半功倍的效果。