第2章

嵌入式系统硬件体系结构

嵌入式系统开发与硬件平台是紧密相连的，没有硬件支持的嵌入式开发是空中楼阁。本章将学习嵌入式处理器硬件体系结构及其相关知识，为后面章节的学习打下基础。学习本章后，读者将掌握和了解如下知识内容。

- 嵌入式硬件的相关基础知识。
- 嵌入式硬件平台基本组成。
- ARM 系列微处理器简介。

2.1 相关基础知识

为了学习嵌入式系统硬件知识，有必要对一些与硬件相关的基础知识做简单介绍。

2.1.1 嵌入式微处理器

1. 嵌入式微处理器的组成

嵌入式系统的中央微处理器，简称 CPU，是嵌入式系统中最重要的一个部分。CPU 又由运算器和控制器两大部分组成，如图 2.1 所示。

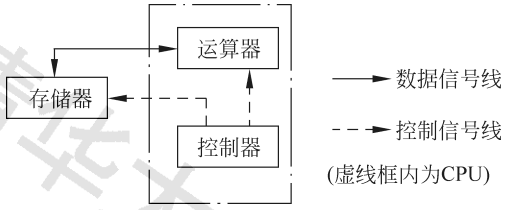

图 2.1 微处理器的组成

- 运算器：用来完成算术运算和逻辑运算，运算的中间结果被暂存在运算器内。
- 控制器：用来控制、指挥程序和数据的输入、运行，处理运算结果。它是计算机组成的神经中枢，指挥全机各部件自动、协调地工作。

2. 微处理器的重要指标

1）主频、倍频、外频

主频指 CPU 的时钟频率，即 CPU 运算时的工作频率。一般说来，主频越高，CPU 的处理速度也就越快。外频指系统总线的工作频率，CPU 的外频决定整块主板的运行速度。倍

频则指 CPU 外频与主频相差的倍数。三者是有十分密切的关系的：主频＝外频×倍频。

2）缓存

缓存大小是 CPU 的重要指标之一，缓存的结构和大小对 CPU 速度的影响非常大。CPU 内缓存的运行频率极高，一般是和微处理器同频运作，工作效率远大于系统内存和硬盘。

L1 Cache(一级缓存)是 CPU 第一层高速缓存，分为数据缓存和指令缓存。内置的 L1 高速缓存的容量和结构对 CPU 的性能影响较大，不过高速缓冲存储器均由静态 RAM 组成，结构较复杂，在 CPU 管芯面积不能太大的情况下，L1 级高速缓存的容量不可能做得太大。一般服务器 CPU 的 L1 缓存的容量通常在 32～256KB。

L2 Cache(二级缓存)是 CPU 的第二层高速缓存，分内部和外部两种芯片。内部的芯片二级缓存运行速度与主频相同；外部的二级缓存则只有主频的一半。L2 高速缓存容量也会影响 CPU 的性能，原则是越大越好。以前家庭用 CPU 容量最大的是 512KB；现在笔记本电脑中也可以达到 2MB；而服务器和工作站用 CPU 的 L2 高速缓存更高，可以达到 8MB。

L3 Cache(三级缓存)分为两种，早期的是外置，现在则都是内置的。它的应用可以进一步降低内存延迟，同时提升大数据量计算时处理器的性能。

2.1.2 嵌入式微处理器的流水线技术

1. 微处理器的流水线技术

通常微处理器处理一条指令要经过 3 个步骤：取指(从存储器装载一条指令)、译码(识别将要被执行的指令)、执行(处理指令并将结果写回寄存器)。流水线技术通过多个功能部件并行工作缩短程序执行时间，提高微处理器的运行效率和吞吐率。微处理器的三级流水线技术如图 2.2 所示。它在同一时间周期并行执行若干指令的取指、译码、执行操作，运行效率是逐条执行指令的 3 倍。

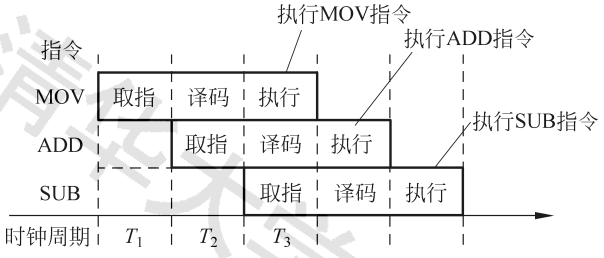

图 2.2 微处理器的三级流水线技术

2. 嵌入式微处理器的流水线

嵌入式微处理器 ARM7 采用典型的三级流水线，嵌入式微处理器 ARM9 则采用五级流水线技术，ARM11 更是使用了八级流水线。通过增加流水线级数，简化了流水线的各级逻辑，进一步提高了处理器的性能。各型号嵌入式微处理器的流水线技术如图 2.3 所示。

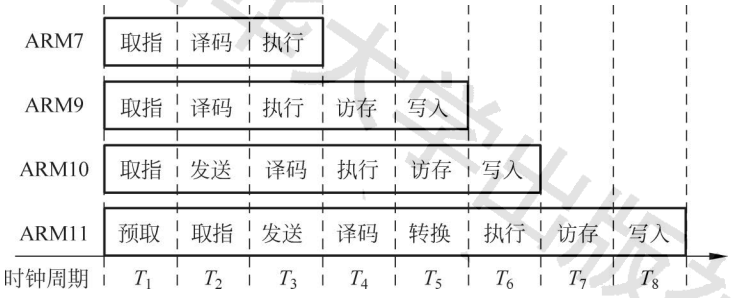

图 2.3　各型号微处理器的流水线技术

2.1.3　寄存器与存储器

寄存器在 CPU 内部，它的访问速度快，但容量小、成本高，它没有地址，用名字来标识（如 AX、BX 等）；存储器在 CPU 的外部，它的访问速度比寄存器慢，容量大、成本低，存储单元用地址来标识。下面分别进行介绍。

1. 寄存器

寄存器（register）是 CPU 的组成部分，是 CPU 内部用来存放数据的一些小型存储区域，用于暂时存放参与运算的数据和运算结果。寄存器是一种时序逻辑电路，这种时序逻辑电路只包含存储电路。寄存器的存储电路是由锁存器或触发器构成的，一个锁存器或触发器能存储 1 位二进制数，所以由 N 个锁存器或触发器可以构成 N 位寄存器。寄存器是 CPU 内部的元件，它拥有非常高的读写速度，所以寄存器之间的数据传送非常快。

根据寄存器的作用，可分为控制寄存器、状态寄存器和数据寄存器 3 大类。控制寄存器用来控制外部设备；状态寄存器用于外部设备向 CPU 报告设备目前的工作状态；数据寄存器用于暂时存放数据。

有些外部设备把一些特定功能的存储单元称为寄存器，这些特定功能的存储单元，其物理结构跟内存单元不一样，但作用跟内存单元一样，都能保存信息。设计人员给外部设备的每个寄存器都分配一个地址，CPU 根据地址访问某个寄存器，该寄存器则发生相应的动作：或接收数据总线上的数据（对应写操作），或把自己的数据送到数据总线上（对应读操作）。当 CPU 访问某个寄存器时，同一个外设的其他寄存器和其他外设的寄存器由于没有 CPU 的指令不会发生动作。

2. 随机存储器（RAM）

存储器芯片是一种由数以百万计的晶体管和电容器构成的集成电路（IC）。嵌入式系统的存储器中，最常见的一种是动态随机存储器（DRAM），在 DRAM 中晶体管和电容器合在一起就构成一个存储单元，代表一个数据位。电容器保存一位二进制信息位——0 或 1（电容器有无电荷表示数据 1 或 0）。晶体管在电路中起开关的作用，它能让内存芯片上的控制线路读取电容器上的数据，或改变其状态，如图 2.4 所示。

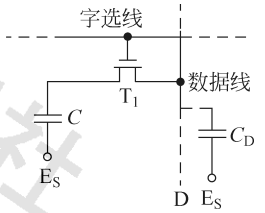

图 2.4　DRAM 的存储单元

电路中的电容器就像一个会泄漏的小桶,只需几毫秒,一个充满电子的小桶就会漏得一干二净。因此,为了确保动态存储器能正常工作,必须由 CPU 或内存控制器对所有电容器不断地进行充电,使它们在电子流失殆尽之前能保持 1 值。为此,内存控制器会先行读取存储器中的数据,然后再把数据写回去。这种刷新操作每秒钟要自动进行数千次。

将很多 DRAM 基本存储单元连接到同一个列线(位线)和同一个行线(字线)组成一个矩阵结构,位线和字线相交,就形成了存储单元的地址,如图 2.5 所示。

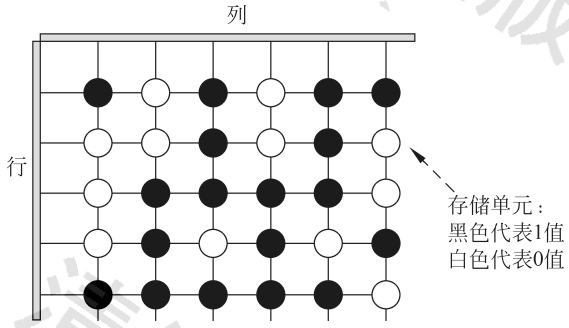

图 2.5 存储单元矩阵

DRAM 工作时会向选定的列线发送电荷,以激活该列上每个位元处的晶体管。写入数据时,行线路会使电容保持应有状态。读取数据时,由灵敏放大器测定电容器中的电量水平。如果电量水平大于 50%,就读取 1 值;否则读取 0 值。计数器会跟踪刷新序列,即记录哪些行被访问过,以及访问的次序。完成全部工作所需的时间极短,需要以纳秒(十亿分之一秒)计算。

3. 内存中数据存放的大小端模式

嵌入式系统中,存储是以字节为单位的,每个地址单元对应着一字节,一字节为 8 位。位数大于 8 位的处理器(如 16 位或 32 位的处理器),由于寄存器宽度大于一字节,因此根据多字节存储方式的不同,就有了大端存储模式和小端存储模式。

(1) 大端模式:数据的高字节保存在内存的低地址中,而数据的低字节保存在内存的高地址中。

(2) 小端模式:数据的高字节保存在内存的高地址中,而数据的低字节保存在内存的低地址中。

例如,一个 32 位宽的数的十六进制表示为 0x01234567,如果是小端模式,则存储方式(地址从低位开始)为 0x67 0x45 0x23 0x01,如果是大端模式,则存储方式为 0x01 0x23 0x45 0x67,如图 2.6 所示。

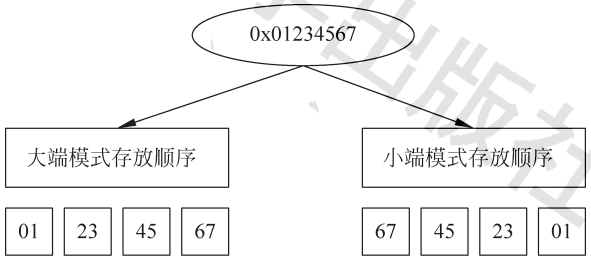

图 2.6 内存中数据存放的大、小端模式

2.1.4　总线

总线(BUS)是接口电路与 CPU 或者接口电路与 I/O 外部设备之间连接的主要形式，是各功能部件之间传送信息的公共通路。嵌入式系统大都采用总线结构，这种结构的特点是采用一组公共的信号线作为嵌入式系统各部件之间的通信线，这组公共信号线称为总线。

在嵌入式系统的应用中，有些场合所用的系统结构并不需要很复杂，只需要用微处理器与为数不多的外围设备构成一个小系统；有些场合则需要在若干插件之间或子系统之间，都有各自的总线，把各功能部件连接起来，组成一个彼此传递信息和对信息进行处理的整体。因此，总线是各功能部件联系的纽带，在接口技术中扮演着重要的角色。

总线的基本功能是实现信息交换和信息共享。它主要由传输信息的物理介质和管理信息传输的协议组成。

通信协议是指通信双方的一种约定。约定包括对数据格式、同步方式、传送速度、传送步骤等问题做出的统一规定，通信双方必须共同遵守。

1. 总线时序协议

连接到总线上的模块分为主设备和从设备两种形式。主设备可以启动一个总线事务(总线周期)，从设备则响应主设备的请求。每次只能有一个主设备控制总线，但同一时间可以有一个或多个从设备响应主设备的请求。为了同步主/从设备的操作，必须制定一个时序协议。

时序指总线上协调事件操作运行的时间顺序，也叫定时。时序协议分为同步时序协议和异步时序协议。

对于同步时序，总线上所有事件共用同一时钟脉冲进行过程的控制。

总线中包含时钟信号，它传送由相同长度的 0、1 交替的规则信号组成的时钟序列。一次 1 和 0 的转换称为一个时钟周期或总线周期，它定义了一个时间槽。总线上所有其他设备都能读取时钟线，而且所有的事件都在时钟周期的开始时发生。总线信号可以在时钟上升沿发生(稍有延迟)，大多数事件占用一个时钟周期。

同步读操作的时序如图 2.7 所示。图中设备 1 发出起始信号(Start)来标识总线上地址和控制信号的出现，它同时发出读信号(Read)，并将设备 2 的地址放到地址总线上。设备 2 检测到识别地址则在延迟 1 个时钟周期后，将数据和响应信号放到总线上。

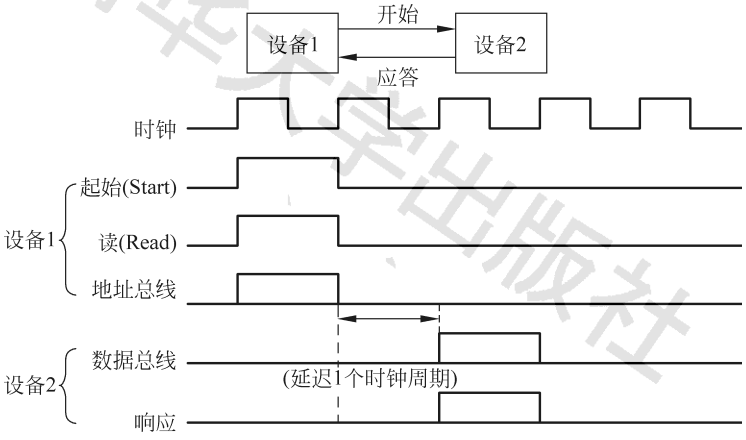

图 2.7　同步时序

在同步时序协议中,事件出现在总线的时刻由总线时钟来确定。所有事件都出现在时钟的前沿,大多数事件只占据单一时钟周期。在异步时序协议中,事件出现在总线的时刻取决于前一事件的出现,即建立在握手或互锁机制的基础上。

2. 异步时序协议的握手协议

异步时序操作由源或目的模块发出特定的信号来确定,双方相互提供联络信号。

总线异步时序协议的基本构件是握手协议,所谓"握手",即当两个设备要通信时,一个设备准备好接收,另一个设备准备好发送。实现握手功能需要两根信号线,一根表示查询(enq),另一根表示应答(ack)。在握手过程中,有专用通信线来传输数据。

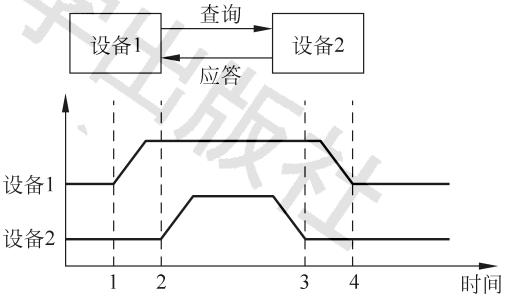

图 2.8 握手协议的 4 个阶段

握手协议的数据传送过程有 4 个阶段,如图 2.8 所示。

握手协议工作过程的各个阶段说明如下。

- 阶段 1:设备 1 升高输出电平发出查询信号,它告诉设备 2 应准备接收数据。
- 阶段 2:当设备 2 准备好接收数据时,它升高它的输出电平发出应答信号。这时,设备 1 和设备 2 均已准备就绪,并开始发送或接收。
- 阶段 3:一旦数据传送完毕,设备 2 降低它的输出电平,表示它已经接收完数据。
- 阶段 4:设备 1 检测到设备 2 的应答信号变低,设备 1 也降低它的输出电平。

在握手结束时,双方握手信号均为低电平,就像开始握手前一样。因此,系统回到其初始状态,为下一次以握手方式传输数据作准备。

3. 总线仲裁方式

对多个主设备提出的占用总线请求,必须在优先级或公平抢占的基础上进行仲裁。由中央仲裁器或设备的仲裁器根据优先级策略进行裁决。

4. 总线标准

总线标准指通过总线将各个设备连接成一个系统所必须遵循的规范。设备只要遵循相应总线标准就可以连接到该总线上去。总线标准主要包括以下内容。

- 机械特性:规定总线的物理连接方式,包括插头、插座的形状、大小,信号针的大小、间距、排列方式等。
- 电气特性:规定与电有关的一些特性,如信号电平的定义,建立时间、保持时间、转换时间,直流特性、交流特性、负载能力等。
- 引脚功能特性:规定总线每一根线的名称、定义、功能和逻辑关系。
- 协议(时序)特性:规定总线每一根线什么时间有效,什么时间失效。

2.1.5 I/O 端口

嵌入式微处理器与通用处理器的一个重要区别在于,嵌入式系统集成了众多不同功能的 I/O 模块,以满足不同产品的需要。

I/O 端口又称 I/O 接口,它是微处理器对外控制和信息交换的必经之路,是 CPU 与外

部设备连接的桥梁，在 CPU 与外部设备之间起信息转换和匹配的作用。I/O 端口有串行和并行之分，串行 I/O 端口一次只能传送一位二进制数信息，而并行 I/O 端口一次能传送一组二进制数信息。

I/O 接口电路由寄存器和逻辑电路组成。I/O 接口的引脚通常会提供多种功能。设计时究竟选用多功能引脚的哪种功能，可以根据用户需要来确定。I/O 接口电路的位置，如图 2.9 所示。

图 2.9　I/O 接口电路的位置

每种外设的操作都是通过读写设备上的寄存器进行的。外设寄存器也称为 I/O 端口，通常被连续编址。

CPU 对外设 I/O 端口物理地址的编址方式有两种：一种是 I/O 映射方式(I/O-mapped)，另一种是内存映射方式(Memory-mapped)。具体采用哪种取决于 CPU 的体系结构。

有些体系结构的 CPU(如 PowerPC、m68k)通常只实现一个物理地址空间(RAM)。在这种情况下，外设 I/O 端口的物理地址就被映射到 CPU 的单一物理地址空间中，成为内存的一部分。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需设立专门的外设 I/O 指令。这就是内存映射方式(Memory-mapped)。

另外一些体系结构的 CPU(如 X86)为外设专门实现了一个单独的地址空间，称“I/O 地址空间”或者“I/O 端口空间”。这是一个与 RAM 物理地址空间不同的地址空间，所有外设的 I/O 端口均在这一空间进行编址。CPU 通过设立专门的 I/O 指令(如 x86 的 IN 和 OUT 指令)来访问这一空间的地址单元(即 I/O 端口)。这就是所谓的“I/O 映射方式”(I/O-mapped)。与 RAM 物理地址空间相比，I/O 地址空间通常都比较小，如 x86 CPU 的 I/O 空间就只有 64KB(0～0xffff)。这是 I/O 映射方式的一个主要缺点。

嵌入式系统开发板中，常用的 I/O 接口有以下几种。

(1) UART 接口，这是一种遵循工业异步通信标准的接口，又称串行接口。

(2) 通用并行接口，嵌入式系统的通用并行接口主要提供输入、输出、双向功能。

(3) I^2C 接口，是一种由 PHILIPS 公司开发的两线式串行总线，为音频和视频设备而开发设计，如今应用更为广泛。

(4) LCD 显示屏接口，用以支持 LCD 控制器、LCD 驱动器、LCD 显示屏和背光电路。

(5) 触摸屏接口，用于支持触摸屏操作。

(6) A/D 接口，用于连接模拟量的输入、输出，进行 A/D 和 D/A 转换。

(7) 以太网接口，用于连接网络。

2.1.6　中断

微处理器与外部设备的数据传输方式通常有以下 3 种：查询方式、中断方式和直接存储器访问(DMA)方式。

查询方式指 CPU 不断查询外部设备的状态。如果外设准备就绪则开始进行数据传输；如果外设还没准备好，CPU 则进入循环等待状态。显然，这种方式会浪费 CPU 大量时

间,降低了 CPU 的利用率。

中断方式指,当外部设备准备与 CPU 进行数据传输时,外部设备首先向 CPU 发出中断请求,CPU 接收到中断请求并在一定条件下,暂时停止原来的程序并执行中断服务处理程序,执行完毕以后再返回原来的程序继续执行。

所有的处理器至少有一个引脚被用作中断输入,外设控制芯片也有个引脚用作中断输出,把这些管脚连接起来,当外部设备上有事件发生时,其控制器将通过产生一个硬件中断的方式来通知处理器。

中断处理的各个阶段如图 2.10 所示。

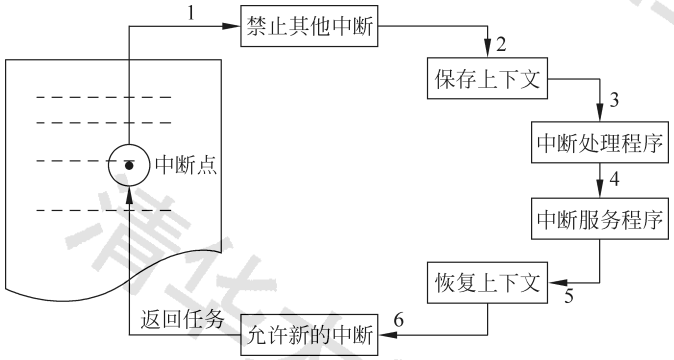

图 2.10 中断处理的各个阶段

(1) 禁止其他中断:当发生中断时,嵌入式微处理器将禁止其他中断的产生,以便进行中断处理。

(2) 保存上下文:进入处理程序,首先要保存当前模式下没有被自动分组保护的部分寄存器。

(3) 中断处理程序:处理程序确定外部中断源,并执行相应的中断服务程序。

(4) 中断服务程序:针对中断源的具体要求进行处理,并复位该中断。

(5) 恢复上下文:从中断服务返回到中断处理程序后,处理程序负责恢复上下文。

(6) 允许新的中断:从中断处理返回,回到被中断的程序继续执行。

直接存储器访问(DMA,Direct Memory Access)指一种快速传送数据的机制。数据传递可以从适配卡到内存、从内存到适配卡或从一段内存到另一段内存。DMA 技术的重要性在于,利用它进行数据传送时不需要 CPU 的参与。每台计算机主机板上都有 DMA 控制器,在实现 DMA 传输时,是由 DMA 控制器直接掌管总线,因此,存在着一个总线控制权转移问题。即在 DMA 传输前,CPU 要把总线控制权交给 DMA 控制器,DMA 脱离 CPU,独立完成数据传送。在结束 DMA 传输后,DMA 控制器立即把总线控制权再交回给 CPU。


视频讲解

2.1.7 数据编码

带有微处理器的硬件系统设计离不开数据编码。为了对这一问题有比较清楚的认识,下面举一个简单示例说明是如何进行数据编码的。

设用微处理器控制一串彩灯(LED 发光二极管),如图 2.11 所示。

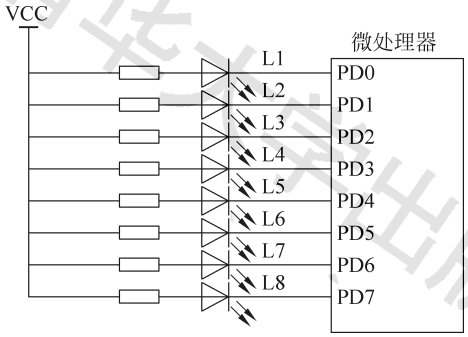

图 2.11　微处理器控制彩灯

假设当连接微处理器的引脚处于低电平时相应的彩灯发光，处于高电平时相应的彩灯不发光(灭)。进一步假设不发光的引脚电平为 1(高电平)，发光的引脚电平为 0(低电平)。

(1) 当彩灯 L1 发光时，PD0 引脚为低电平，其余引脚均为高电平，可以表示为以下对应值：

PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
1	1	1	1	1	1	1	0

这时，可用二进制数表示为：11111110。若将这种情况按十六进制编码，其值为：FEH。

(2) 若要彩灯 L8 发光，其余均不发光，则有：

PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
0	1	1	1	1	1	1	1

这时，用二进制数表示为：01111111。它的十六进制编码为：7FH。

(3) 若希望两边亮，中间暗，则有：

PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
0	1	1	1	1	1	1	0

这时，用二进制数表示为：01111110。它的十六进制编码为：7EH。

以此类推，可以编写出各种情况的编码来。

2.2　嵌入式系统硬件平台

嵌入式微处理器芯片不能独立工作，它需要必要的外围设备给它提供基本的工作条件。嵌入式硬件平台由嵌入式处理器和嵌入式系统外围设备组成，其结构如图 2.12 所示。

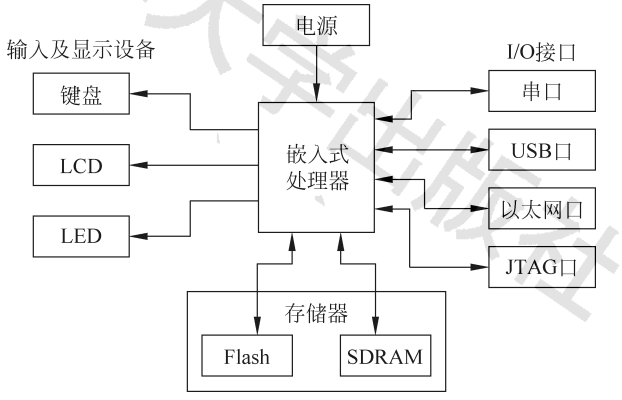

图 2.12　嵌入式系统硬件结构

1. 嵌入式处理器

嵌入式处理器与普通计算机的处理器一样,是嵌入式系统的核心部件。但由于其功耗、体积、成本、可靠性、稳定性、速度、处理能力等方面均受到应用要求的制约,在实际应用中,只保留和嵌入式应用紧密相关的功能硬件,去除了其他的冗余功能部分,这样就以最低的功耗和资源实现了嵌入式应用的特殊要求。

嵌入式处理器通常包括处理器内核、地址总线、数据总线、控制总线、片上 I/O 接口电路及辅助电路(如时钟、复位电路等)。

嵌入式处理器可以分为 3 类:嵌入式微处理器、嵌入式微控制器和嵌入式 DSP(Digital Signal Processor,数字信号处理器),简要介绍如下。

(1) 嵌入式微处理器:嵌入式微处理器和通用计算机中的微处理器相对应(通常称 CPU)。在实际应用中,一般将嵌入式微处理器装配在专门设计的一小块电路板上,使用时,插接到应用电路板上,这样可以满足嵌入式系统体积小、功耗低、应用灵活的要求。

(2) 嵌入式微控制器:嵌入式微控制器通常又称单片机,它将 CPU、存储器和其他外设封装在同一片集成电路里。

(3) 嵌入式 DSP:这种微处理器专门对离散时间信号进行快速计算,以提高执行速度。DSP 广泛应用于数字滤波、图像处理等领域。

本书中所讲的嵌入式处理器主要指嵌入式微处理器。

2. 嵌入式系统中的存储设备

嵌入式系统中的外围设备指嵌入式系统中用于完成存储、通信、调试、显示等辅助功能的其他部件。常用的嵌入式外围设备按功能可以分为存储设备、通信设备和显示设备。

存储设备在嵌入式系统开发过程中非常重要,常见的存储设备有 RAM、SRAM、ROM、Flash 等。根据掉电后数据是否丢失,存储器可以分为 RAM(随机存取存储器)和 ROM(只读存储器),其中 RAM 的访问速度比较快,但掉电后数据会丢失,而 ROM 掉电后数据不会丢失。

1) RAM、SRAM、DRAM

RAM 即通常所说的内存。RAM 分为 SRAM(静态存储器)和 DRAM(动态存储器)。

SRAM 利用双稳态触发器保存信息,只要不掉电,信息不会丢失。

DRAM 利用 MOS(金属氧化物半导体)电容存储电荷来储存信息,因此必须通过不停地给电容充电来维持信息。DRAM 的成本、集成度、功耗等明显优于 SRAM。

通常所说的 SDRAM 是 DRAM 的一种,它是同步动态存储器,利用单一的系统时钟同步所有的地址、数据和控制信号。SDRAM 不但能提高系统表现,还能简化设计,提供高速的数据传输,在嵌入式系统中经常使用。

2) Flash

Flash 是一种非易失闪存,它具有和 ROM 一样掉电后数据不会丢失的特性。Flash 是目前嵌入式系统中广泛采用的存储器,它的主要特点是按整体/扇区擦除和按字节编程,具有低功耗、高密度、小体积等优点。Flash 主要分为 NOR Flash 和 NAND Flash 两种。

NOR Flash 的特点是在芯片内执行,可以直接读取芯片内储存的数据,因而速度比较快。NOR Flash 地址线与数据线分开,所以 NOR Flash 型芯片可以像 SRAM 一样连在数据线上,可以以“字”为基本单位操作,因此传输效率很高,应用程序可以直接在 Flash 内运

行，不必再把代码读到系统 RAM 中运行。它与 SRAM 的最大不同在于，写操作需要经过擦除和写入两个过程。Flash 在写入信息前必须擦除，否则写入数据会导致错误；每次擦除只能擦除一个扇区，不能逐字节地擦除。

NAND Flash 能提供极高的单元密度，可以达到高存储密度。NAND Flash 读和写操作采用 512 字节的块，每个块的最大擦写次数超过 100 万次，是 NOR Flash 的 10 倍，这些特性使 NAND Flash 越来越受到人们的青睐。

NAND Flash 芯片共用地址线与数据线，内部数据以块为单位进行存储，直接将其作为启动芯片比较难。NAND Flash 是连续存储介质，适合放大文件。擦除 NOR Flash 时是以 64～128KB 的块进行的，执行一个写入或擦除操作的时间为 5s；擦除 NAND Flash 是以 8～32KB 的块进行的，执行相同的操作最多只需要 4ms。

NAND Flash 的单元尺寸几乎是 NOR Flash 器件的一半，生产过程更为简单，NAND Flash 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

在使用寿命（耐用性）方面，NAND Flash 中每个块的最大擦写次数是 100 万次，而 NOR Flash 的最大擦写次数是 10 万次。NAND Flash 存储器除了具有 10∶1 的块擦除周期优势，典型的 NAND Flash 块尺寸要比 NOR Flash 器件小 8 倍，每个 NAND Flash 存储器块在给定的时间内的删除次数要少一些。

NOR Flash 与 NAND Flash 的比较见表 2.1。

表 2.1　NOR Flash 与 NAND Flash 的比较

NOR Flash	NAND Flash
接口时序同 SRAM，易使用	地址/数据线复用，数据位较窄
读取速度较快	读取速度较慢
擦除速度慢，以 64～128KB 的块为单位	擦除速度快，以 8～32KB 的块为单位
写入速度慢	写入速度快
随机存取速度较快，支持 XIP（eXecute In Place，芯片内执行），适用于代码存储。在嵌入式系统中，常用于存放引导程序、根文件系统等	顺序读取速度较快，随机存取速度慢，适用于数据存储（如大容量的多媒体应用）。在嵌入式系统中，常用于存放用户文件系统等
单片容量较小	单片容量较大，提高了单元密度
最大擦写次数 10 万次	最大擦写次数 100 万次

3. JTAG 接口

JTAG（Joint Test Action Group，联合测试行动小组）是一种国际标准测试协议（IEEE 1149.1 兼容），主要用于芯片内部测试。现在多数的高档微处理器都支持 JTAG 协议。

JTAG 最初是用来对电路和芯片进行边界扫描测试的，JTAG 的基本原理是在器件内部定义一个 TAP（Test Access Port，测试访问端口），通过专用的 JTAG 测试工具对器件内部节点进行测试。通过电路的边界扫描测试技术，对由具有边界扫描功能的芯片构成的印制电路板，通过相应的测试设备，可检测已安装在印制电路板上的芯片功能，检测印制电路板连线的正确性，同时，可以方便地检测该印制电路板是否具有预定的逻辑功能，进而对由这块印制电路板构成的数字电路设备进行故障检测和故障定位。

现在 JTAG 除用于电路边界扫描测试之外，还常用于可编程芯片的在线编程。

由于JTAG经常使用排线连接,为了增强抗干扰能力,在每条信号线间都加上一根地线。JTAG电路原理及实物如图2.13所示,其中74HC244起隔离及驱动作用,通常称为"JTAG小板"或"仿真器"。图中电阻单位为Ω。

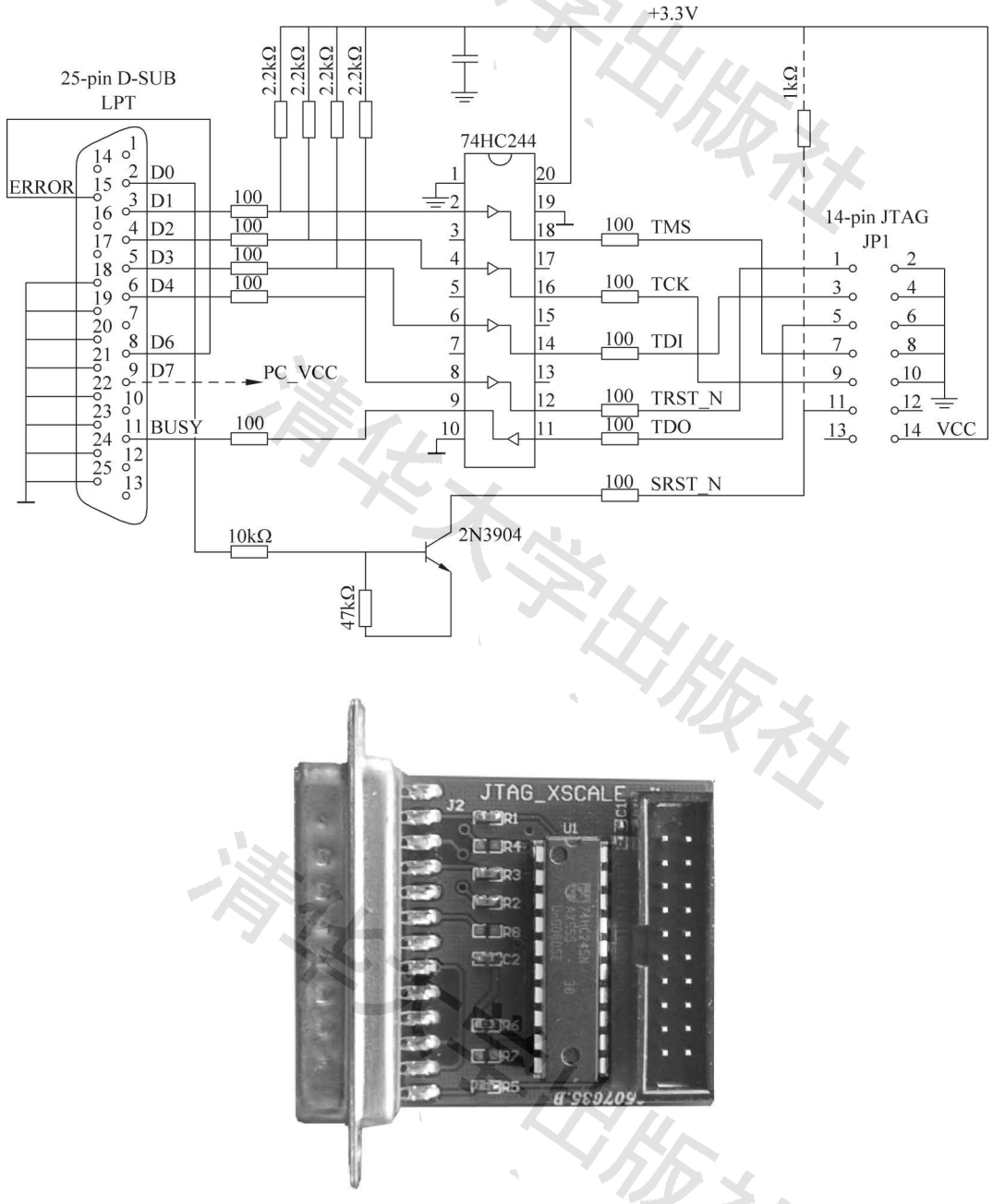

图2.13 JTAG电路原理图及实物图

JTAG引脚定义如下。

(1) TCK为TAP操作提供一个独立、基本的时钟信号,TAP的所有操作都是通过这个时钟信号来驱动的。

(2) TMS 用来控制 TAP 状态机的转换,通过 TMS 信号可以控制 TAP 在不同的状态间相互转换,TMS 信号在 TCK 信号的上升沿有效。

(3) TDI 是数据输入的接口,所有输入特定寄存器的数据都要通过 TDI 一位一位串行输入。

(4) TDO 是数据输出的接口,所有从特定寄存器输出的数据都要通过 TDO 一位一位串行输出。

(5) TRST 可以用来对 TAP 控制器进行复位,该信号线可选,TMS 也可以对其复位。

(6) VTREF 接口信号电平参考电压一般直接接 Vsupply,它可以用来确定 ARM 的 JTAG 接口逻辑电平。

(7) RTCK 可选项,由目标端反馈给仿真器的时钟信号,用来同步 TCK 信号的产生,不使用时直接接地。

(8) System Reset 可选项,与目标板上的系统复位信号相连,可以直接对目标系统复位,同时可以检测目标系统的复位情况。为了防止误触发应在目标端加上适当的上拉电阻。

(9) USER IN 用户自定义输入,可以接到一个 I/O 口上,用来接受上位机的控制。

(10) USER OUT 用户自定义输出,可以接到一个 I/O 口上,用来向上位机反馈一个状态。

在嵌入式系统中,通过 JTAG 接口既可以对目标板系统进行测试,也可以对目标板系统的存储单元(Flash)编程。我们经常用简易 JTAG 接口直接烧写嵌入式系统 Flash 存储器。这种烧写方式通过一根并口电缆和一块信号转换集成电路板,建立起 PC 与开发板之间的通信。这样就可以将启动引导程序烧入空 Flash 存储器中,从而实现自启动。

嵌入式系统目标板的 JTAG 接口通过“JTAG 小板”与宿主机连接的方式如图 2.14 所示。

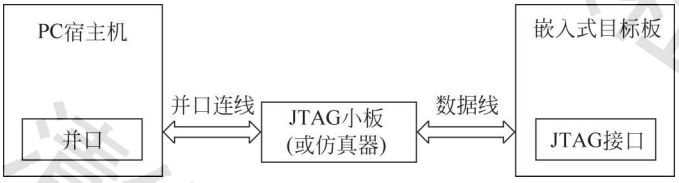

图 2.14　JTAG 接口与宿主机的连接

关于 JTAG 和 IEEE 1149 标准

随着芯片的整合度越来越高、尺寸越来越小,芯片内部的复杂度也随之不断上升,半导体制程中可能的各种失效状况、材料的缺陷以及制程偏差等,都有可能导致芯片中电路连接的短路、断路以及元件穿隧效应等问题。而这样的物理性失效必然导致电路功能或者性能方面的缺陷,因此产业界需要具备广泛的高效率测试方式,来提供大规模集成电路设计的完整的验证解决方案。

JTAG 小组在 1986 年,针对芯片、印刷电路板以及完整系统上的标准化测试技术,提出了标准的边界扫描体系架构(Boundary-Scan Architecture Standard Proposal);在 1988 年,该小组与 IEEE 组织合作,开始进行标准的开发,并命名为 1149.1,并于 1990 年正式发布。

2.3 ARM微处理器体系

2.3.1 ARM公司及ARM体系结构

1. ARM公司简介

ARM(Advanced RISC Machines),可以认为是一个公司的名字,也可以认为是对一类微处理器的通称,还可以认为是一种技术的名字。ARM公司是专门从事基于RISC技术芯片设计开发的公司,作为知识产权供应商,本身不直接从事芯片生产,靠转让设计许可,由合作公司生产各具特色的芯片。世界各大半导体生产商从ARM公司购买其ARM微处理器核,根据各自不同的应用领域,加入适当的外围电路,从而形成自己的ARM微处理器芯片进入市场。

ARM公司的产品以耗电少、成本低、重用性为特点,并以优异的产品性能和合作伙伴模式著称于世。20世纪90年代初,ARM率先推出32位RISC微处理器芯片系统(SOC)知识产权(IP)公开授权概念,从此改变了半导体行业。ARM公司通过出售芯片技术授权,建立起新型的微处理器设计、生产和销售商业模式。正是这种商业模式,使采用ARM技术的微处理器在各类电子产品,汽车、消费娱乐、成像、工业控制、存储、网络、安保和无线通信等领域得到广泛应用。

这一商业模式的成功,源自ARM公司多年的经营经验和辉煌的发展历史。ARM公司的前身是英国Acorn公司,产品主要是基于8位计算机的产品,最初的4个工程师意识到要设计自己的CPU,必须要有出色的性能和较低的成本以及低功耗和重用性。经过两年苦心研发,1985年4月,公司第一个集成了2.5万个晶体管的低成本Acorn RISC问世。1990年,Acorn公司联合苹果公司及芯片厂商VLSI公司共同投资成立了ARM公司,公司成立伊始,就当时的市场形势,分析了公司面临的机遇和挑战,意识到虽然公司在产品的能耗、成本和系统应用方面远远超出竞争对手,但没有自己的专利技术,在市场份额和效益方面也存在不足。而当时便携式设备、嵌入式控制、汽车电子市场等新的技术正在兴起,必须依靠合作伙伴来共同发展才能把市场做大。基于这样的思想,公司开始了ARM产品的研发。2001年,世界顶级的半导体公司和生产厂商纷纷取得了ARM公司的专利授权,ARM公司成为IP市场最为炫目的一颗明珠,销售收入达到了1.46亿英镑(2.25亿美元),远远超过了其他竞争对手。

2016年7月,日本软银集团以320亿美元收购了ARM公司,开始了ARM公司的新一轮跃进。

目前,采用ARM技术知识产权(IP)核的微处理器,即通常所说的ARM微处理器,已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统、军用系统等各类产品市场,基于ARM技术的微处理器应用约占据了32位RISC微处理器70%以上的市场份额,ARM技术已经渗入我们生活的各个方面。

2. 处理器的架构和ARM微处理器体系

处理器的架构是CPU厂商给属于同一系列的CPU产品定的一个规范,是区分不同类型CPU的重要标示。目前市面上的CPU指令集分类主要分有两大阵营,一个是以Intel、

AMD 公司为首的复杂指令集 CPU，另一个是以 IBM、ARM 公司为首的精简指令集 CPU。不同品牌的 CPU，其产品的架构也不相同。例如，Intel、AMD 的 CPU 是 X86 架构的；IBM 公司的 CPU 是 PowerPC 架构的；ARM 公司是 ARM 架构。

ARM 微处理器目前包括 ARM7 系列、ARM9 系列、ARM10 系列、Cortex-M 系列、Cortex-R 系列和 Cortex-A 系列，此外还有其他厂商基于 ARM 体系结构的处理器。除了具有 ARM 体系结构的共同特点以外，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

3. 总线体系结构

根据计算机的存储器结构及其总线连接形式，计算机系统可以被分为冯·诺依曼总线体系结构和哈佛总线体系结构。

冯·诺依曼结构中，存储器内部的数据存储空间和程序存储空间是合在一起的，它们共享存储器总线，即数据和指令在同一条总线上通过时分复用的方式进行传输。这种结构在高速运行时，不能达到同时取指令和取操作数的目的，从而形成了传输过程的瓶颈。冯·诺依曼总线体系结构被大多数微处理器所采用。冯·诺依曼总线体系结构如图 2.15 所示。

随着微电子技术的发展，以 DSP 和 ARM 为应用代表的哈佛总线技术应运而生。在哈佛总线体系结构的芯片内部，数据存储空间和程序存储空间是分开的，所以哈佛总线结构在指令执行时，可以同时取指令（来自程序空间）和取操作数（来自数据空间），因此具有更高的执行效率。修正的哈佛总线结构还可以在程序空间和数据空间之间相互传送数据。哈佛总线体系结构如图 2.16 所示。

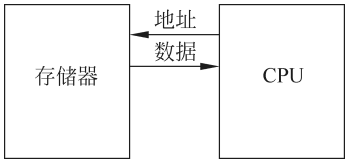

图 2.15　冯·诺依曼总线体系结构

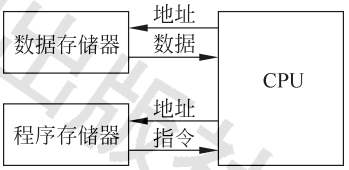

图 2.16　哈佛总线体系结构

目前，ARM 嵌入式系统微处理器内核都采用哈佛总线体系结构，而早期的 ARM7 采用的则是冯·诺依曼结构。

4. ARM 微处理器的特点

采用 RISC 架构的 ARM 微处理器一般具有如下特点。

（1）体积小、低功耗、低成本、高性能。

（2）支持 Thumb（16 位）/ARM（32 位）双指令集，能很好地兼容 8/16 位器件。

（3）大量使用寄存器，指令执行速度更快。

（4）大多数数据操作都在寄存器中完成。

（5）寻址方式灵活简单，执行效率高。

（6）指令长度固定。

2.3.2　ARM 系列微处理器简介

下面简单介绍 ARM 系列的微处理器，使读者对 ARM 体系有一个较为完整的认识。

1. ARM7 系列微处理器

ARM7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求比

较严格的消费类应用。

其中,ARM7 的 S3C44B0 是目前使用较为广泛的 32 位嵌入式 RISC 处理器,属低端 ARM 处理器核,价格比较便宜。

2. ARM9 系列微处理器

ARM9 系列微处理器为可综合处理器,使用单一的处理器内核提供了微控制器、DSP、Java 应用系统的解决方案,极大地减少了芯片的面积和系统的复杂程度。ARM9 系列微处理器提供了增强的 DSP 处理能力,很适合于那些需要同时使用 DSP 和微控制器的应用场合。

前几年市场较为流行的 S3C2410 即为 ARM9E 系列微处理器。

3. Xscale 系列微处理器

Xscale 系列微处理器是基于 ARMv5TE 体系结构的解决方案,是一款全性能、高性价比、低功耗的 32 位处理器。它也支持 16 位的 Thumb 指令和 DSP 指令集,已使用在数字移动电话、个人数字助理和网络产品等场合。Xscale 处理器是 Intel 公司推出的一款源于 ARM 内核的微处理器。

4. Cortex 系列微处理器

ARM Cortex 系列处理器可向操作系统平台的设备和用户应用提供全方位的解决方案,其应用范围包括超低成本的手机、高端智能手机、平板电脑、数字电视、机顶盒、网络服务器等。该系列处理器的主频速度已经达到 2GHz。该系列微处理器包括 Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A15、Cortex-A55 和 Cortex-A75 等,它们均共享同一架构,具有完整的应用兼容性,支持传统的 ARM、Thumb 指令集和新增的高性能紧凑型 Thumb-2 指令集。

ARMCortex 系列微处理器是 ARM 公司推出的第二代微处理器,它的发展历程如图 2.17 所示。

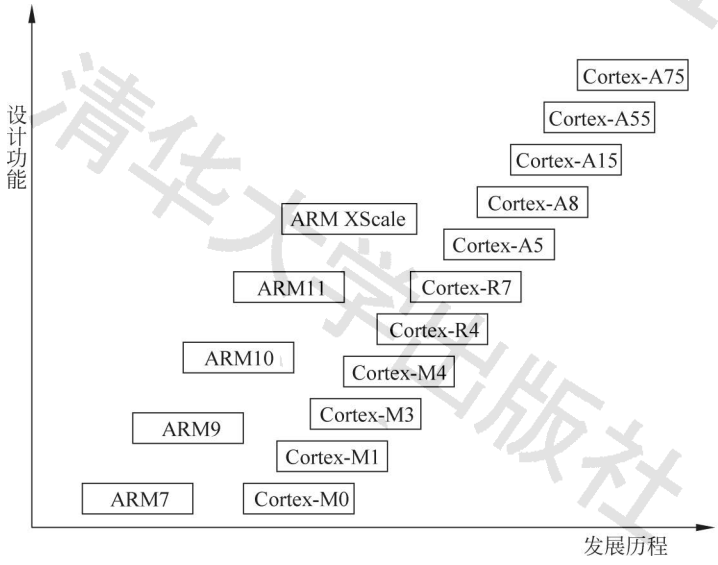

图 2.17 ARM 微处理器发展历程

2.4　微处理器的结构

2.4.1　RISC 体系结构和 ARM 设计思想

1. RISC 体系结构

传统的 CISC(Complex Instruction Set Computer,复杂指令集计算机)结构有其固有的缺点,即随着计算机技术的发展而不断引入新的复杂的指令集,为支持这些新增的指令,计算机的体系结构越来越复杂。然而,在 CISC 指令集的各种指令中,其使用频率却相差悬殊,大约有 20%的指令被反复使用,占整个程序代码的 80%。而余下的 80%的指令却不经常使用,在程序设计中只占 20%,显然,这种结构是不太合理的。

基于以上不合理性,1979 年美国加州大学伯克利分校提出了 RISC(Reduced Instruction Set Computer,精简指令集计算机)的概念。RISC 结构优先选取使用频率最高的简单指令,避免复杂指令;将指令长度固定,指令格式和寻址方式种类减少;以控制逻辑为主。到目前为止,RISC 体系结构也还没有严格的定义,一般认为,RISC 体系结构应具有如下特点。

- 采用固定长度的指令格式,指令归整、简单,基本寻址方式有 2～3 种。
- 使用单周期指令,便于流水线操作执行。
- 大量使用寄存器,数据处理指令只对寄存器进行操作,只有加载/存储指令可以访问存储器,以提高指令的执行效率。

除此以外,ARM 体系结构还采用了一些特别的技术,在保证高性能的前提下尽量缩小芯片的面积,并降低功耗。

- 所有的指令都可根据前面的执行结果决定是否被执行,从而提高指令的执行效率。
- 可用加载/存储指令批量传输数据,以提高数据的传输效率。
- 可在一条数据处理指令中同时完成逻辑处理和移位处理。
- 在循环处理中使用地址的自动增减来提高运行效率。

当然,和 CISC 架构相比较,尽管 RISC 架构有上述的优点,但不能认为 RISC 架构就可以取代 CISC 架构。事实上,RISC 和 CISC 各有优势,而且界限并不那么明显。现代的 CPU 往往采用 CISC 的外围,内部加入 RISC 的特性,如超长指令集 CPU 就是融合了 RISC 和 CISC 的优势,成为未来的 CPU 发展方向之一。

2. ARM 设计思想

有许多客观需求促使嵌入式系统的微处理器的设计与通用微处理器有很大的不同。

首先,便携式的嵌入式系统往往电源资源相对贫乏,为降低功耗,ARM 微处理器被设计成较小的核,从而延长其运行时间。

由于成本问题和物理尺寸的限制,嵌入式系统不可能使用大体积的外存设备,所以存储量有限是嵌入式系统又一个限制。这就要求嵌入式系统需要使用高密度代码。

另外,由于嵌入式系统对成本敏感,例如,对于数码相机之类的产品,在设计时每一分钱成本都需要考虑,因此,一般选用速度不高、成本较低的存储器,以降低系统成本。

ARM 内核不是纯粹的 RISC 体系结构,这是为使它能够更好地适应其嵌入式的应用领

域。在某种意义上,ARM 内核的成功,正是它没有在 RISC 概念上陷入太深。因为对嵌入式系统的应用项目来说,系统的关键并不单纯在于微处理器的速度,而在于系统性能、功耗和成本的综合权衡。

2.4.2 ARM Cortex 微处理器结构的最小系统设计

1. 什么是最小系统

嵌入式微处理器芯片自己是不能独立工作的,需要一些必要的外围元器件给它提供基本的工作条件。因此,一个 ARM 最小系统一般包括以下几部分。

(1) ARM 微处理器芯片,这是嵌入式最小系统的心脏。

(2) 电源电路、复位电路、晶振电路,为嵌入式最小系统提供电源以及复位和时钟信号。

(3) 存储器(Flash 和 SDRAM),微处理器芯片内部没有存储器,需要外扩存储器。

(4) UART(RS232 及以太网)接口电路。这是嵌入式最小系统不可缺少的一部分,以便与外界通信联系。

(5) JTAG 调试接口。这也是不可缺少的,操作系统软件的下载与烧写都要通过它来完成。

通常,为了系统的调试方便,可以通过 I/O 口连接若干 LED,用以指示系统的工作状态。

2. Cortex A8 核心开发板

由最小系统组成的电路开发板称为核心板。嵌入式最小系统结构及 Cortex A8 核心板实物如图 2.18 所示。

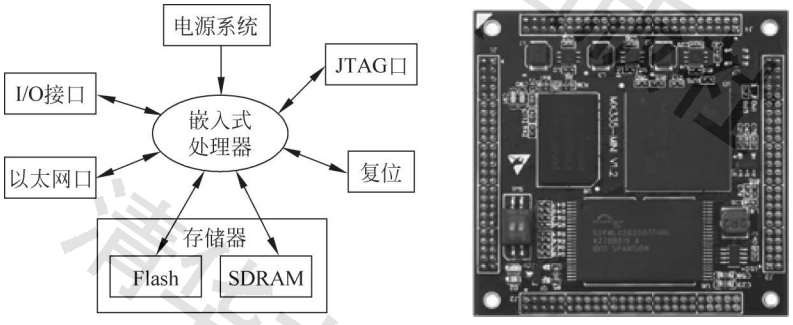

图 2.18 嵌入式最小系统结构及 Cortex A8 核心板实物图

2.4.3 Cortex A8 微处理器结构

Cortex A8 微处理器 Samsung S5PV210 核心板系统包括如下几部分。

(1) CPU : Samsung S5PV210 基于 Cortex A8,运行主频 1GHz。

(2) DDR2 RAM : 512MB 工作在 200MHz 外频上。

(3) Flash: 512MB SLC NAND Flash。

(4) Ethernet CON: 100MB 网络控制器。

Cortex S5PV210 微处理器结构示意图及核心板实物图如图 2.19 所示。

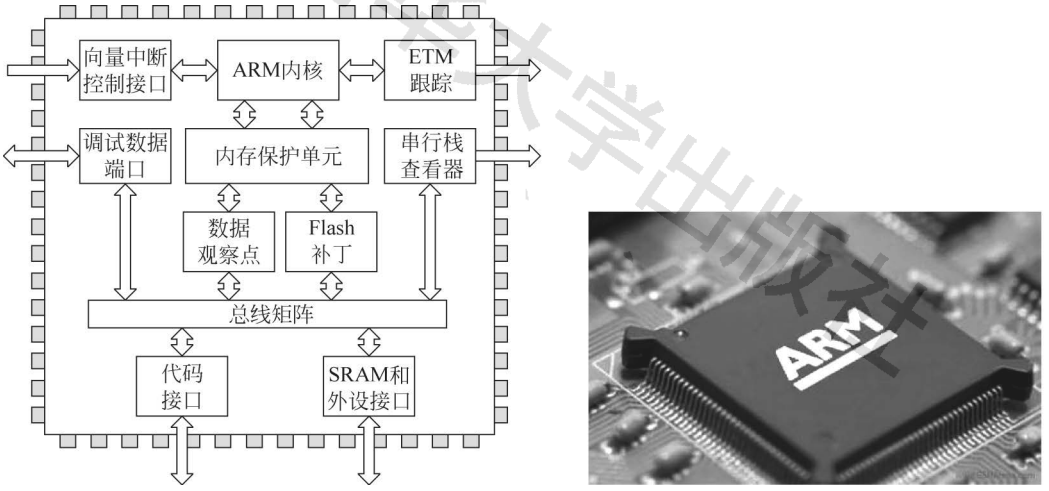

图 2.19　Cortex S5PV210 微处理器结构示意图及实物图

2.4.4　Cortex A8 的存储地址空间

嵌入式微处理器芯片 S5PV210 是采用 ARM Cortex A8 内核的微处理器，现以该处理器为例，说明 Cortex A8 的存储地址空间。S5PV210 存储器地址映射如图 2.20 所示。

从图 2.20 中可以看到，S5PV210 的引导区分为两部分，分别是 0x0000_0000 ～ 0x1FFF_FFFF 和 0xD000_0000 ～ 0xDFFF_FFFF 的地址空间。系统上电后，从引导区开始执行 BootLoader 引导程序。

图 2.20　S5PV10 存储器地址映射

2.4.5　Cortex A8 的 GPIO 端口

通用 I/O 接口(General Purpose IO，GPIO)是嵌入式系统中一种非常重要的 I/O 接口。它具有使用灵活、可配置性好、硬件代价小等优点，在嵌入式系统中广泛应用。


视频讲解

在嵌入式系统中常常有数量众多、结构简单的外部设备/电路，而且，许多这种设备/电路使用时只要求控制一位端口，即只要有开/关两种状态就够了。

例如，控制某个 LED 发光二极管的点亮与熄灭，或者通过获取某个引脚的电平属性来判断外围设备的状态。对这些设备/电路的控制，使用传统的串行口或并行口都不合适。所以在微控制器芯片上一般都会提供一个“通用可编程 I/O 接口”，即 GPIO。

每个 GPIO 端口通常至少有两个寄存器，一个为“I/O 端口控制寄存器”，另一个为“I/O 端口数据寄存器”。

1. S5PV210 微处理器的 GPIO 端口分组

采用 ARMCortex A8 内核的 S5PV210 微处理器共有 237 个可复用的 GPIO 端口和 142 个内存接口引脚，分成 35 组通用 GPIO 端口和两组内存端口，如表 2.2 所示。

表 2.2 Cortex A8 的 GPIO 端口

端口分组	端口引脚数
GPA0	8 个 输入/输出引脚-2×UART 带控制流
GPA1	4 个 输入/输出引脚-2×UART 不带控制流或 1×UART 带控制流
GPB	8 个 输入/输出引脚-2×SPI 总线接口
GPC0	5 个 输入/输出引脚-I^2S 总线接口,PCM 接口,AC97 接口
GPC1	5 个 输入/输出引脚
GPD0	4 个 输入/输出引脚-I^2C 总线接口,PWM 接口,扩展 DMA 接口,SPDIF 接口
GPD1	6 个 输入/输出引脚
GPE0,1	13 个 输入/输出引脚-摄像头接口,SD/MMC 接口
GPF0,1,2,3	30 个 输入/输出引脚-LCD 接口
GPG0,1,2,3	28 个 输入/输出引脚-3×MMC 通道,SPI,I^2S,PCM,SPDIF 各种接口
GPH0,1,2,3	32 个 输入/输出引脚-摄像头通道接口,键盘,最大支持 32 位可中断接口
GPI	低功率 I^2S、PCM 接口
GPJ0,1,2,3,4	35 个 输入/输出引脚-Modem IF,HIS,ATA 接口
MP0_1,2,3	20 个 输入/输出内存端口引脚
MP0_4,5,6,7	32 个 输入/输出内存端口引脚
MP1_0～8	71 个 DRAM1 端口引脚
MP2_0～8	71 个 DRAM2 端口引脚
ETC0,ETC1,ETC2,ETC4	28 个输入/输出 ETC 端口及 JTAG 端口

2. Cortex A8 的常用 GPIO 寄存器

在使用 Cortex A8 微处理器时,由于大多数引脚都是可复用的,因此需要对每个引脚进行配置。Cortex A8 架构的 S5PV210 微处理器有 4 种 GPIO 寄存器,它们是控制寄存器 GPxnCON、数据寄存器 GPxnDAT、上拉/下拉寄存器 GPxnPUD、掉电模式上拉/下拉寄存器 GPxnPUDPDN。现对这 4 种 GPIO 寄存器的功能简述如下。

(1) GPIO 寄存器地址表。

S5PV210 微处理器手册给出了 GPIO 寄存器地址,下面以 GPC0 端口组为例,列出各 GPIO 端口地址,如表 2.3 所示。

表 2.3 GPC0 端口组控制寄存器地址(基址=0xE020_0060)

寄存器	地址	描述	初始值
GPC0CON	0xE020_0060	GPC0 端口组控制寄存器	0x00000000
GPC0DAT	0xE020_0064	GPC0 端口组数据寄存器	0x00
GPC0PUD	0xE020_0068	GPC0 端口组上拉/下拉寄存器	0x0155

(2) 端口控制寄存器 GPxCON(x = A,B,C,D,E,F,G,H,I,J)。

每个 I/O 端口都有一个 CON(端口控制)寄存器,用于控制 GPIO 引脚的功能。该寄存器每 4 位控制一个引脚。

- 当输入 0000 时,引脚设置为输入口,可以从引脚读入外部输入的数据。
- 当输入 0001 时,引脚设置为输出口,向该位写入的数据被发送到对应的引脚上。

例如，GPC0CON 端口控制寄存器的定义如表 2.4 所示（其他端口控制寄存器定义类似）。

表 2.4　GPC0CON 端口控制寄存器定义（地址＝0xE020_0060）

GPC0CON	位	描　　述	初始状态
GPC0CON[4]	[19:16]	0000 = 输入，0001 = 输出， 0010 = I^2S_1_SDO，0011 = PCM_1_SOUT，0100 = AC97SDO， 0101 ～ 1110 = 保留，1111 = GPC0_INT[4]	0000
GPC0CON[3]	[15:12]	0000 = 输入，0001 = 输出， 0010 = I^2S_1_SDI ，0011 = PCM_1_SIN，0100 = AC97SDI， 0101 ～ 1110 =保留，1111 = GPC0_INT[3]	0000
GPC0CON[2]	[11:8]	0000 = 输入，0001 = 输出， 0010=I^2S_1_LRCK，0011=PCM_1_FSYNC，0100=AC97SYNC， 0101 ～ 1110 =保留，1111 = GPC0_INT[2]	0000
GPC0CON[1]	[7:4]	0000 = 输入，0001 = 输出， 0010=I^2S_1_CDCLK，0011=PCM_1_EXTCLK，0100=AC97RESETn， 0101 ～ 1110 =保留，1111 = GPC0_INT[1]	0000
GPC0CON[0]	[3:0]	0000 = 输入，0001 = 输出， 0010=I^2S_1_SCLK，0011=PCM_1_SCLK，0100=AC97BITCLK ， 0101 ～ 1110 =保留，1111 = GPC0_INT[0]	0000

（3）端口数据寄存器 GPxDAT（x = A，B，C，D，E，F，G，H，I，J）。

每个 I/O 端口都有一个 DAT（数据）寄存器，它是一个读写寄存器。该寄存器每 1 位与一个硬件引脚对应。

- 当端口被设置为输出端口时，如果向 GPxDAT 的相应位写入数据 1，则该引脚输出高电平；如果向 GPxDAT 的相应位写入数据 0，则该引脚输出低电平。
- 当端口被设置为输入端口时，可以读取 GPxDAT 相应位的数据，得到端口电平状态。

例如，GPC0DAT 端口数据寄存器的定义如表 2.5 所示。

表 2.5　GPC0DAT 端口数据寄存器的定义

GPC0DAT	位	描　　述	初始状态
GPC0DAT[4:0]	[4:0]	决定输入或者输出的电平状态	0x00

（4）端口上拉/下拉寄存器 GPxPUD（x＝A，B，C，D，E，F，G，H，I，J）。

每个 I/O 端口都有一个 PUD（上拉/下拉使能）寄存器，该寄存器控制了每个端口组的上拉/下拉电阻的使能/禁止。根据对应位的 0/1 组合，设置对应端口的上拉/下拉电阻功能是否使能。如果端口的上拉电阻被使能，则无论在哪种状态（输入、输出、DATAn、EINTn 等）下，上拉电阻都起作用。

例如，GPC0PUD 端口上拉/下拉寄存器的定义如表 2.6 所示。

表 2.6　GPC0PUD 端口上拉/下拉寄存器的定义

GPC0PUD	位	描述	初始状态
GPC0PUD[n]	[2n+1:2n] n=0～4	00＝禁止上拉/下拉，01＝下拉使能， 10＝上拉使能，　　11＝保留	0x0155

3. GPIO 寄存器功能设置应用示例

【例 2-1】 设在 Cortex A8 微处理器 GPIO 端口的 GPC0[2]引脚连接一个 LED 发光二极管,如图 2.21 所示。现对该端口的控制寄存器 GPC0CON 和数据寄存器 GPC0DAT 进行设置,使 LED 发光二极管点亮或熄灭。对于本例,上拉/下拉寄存器不需要设置。

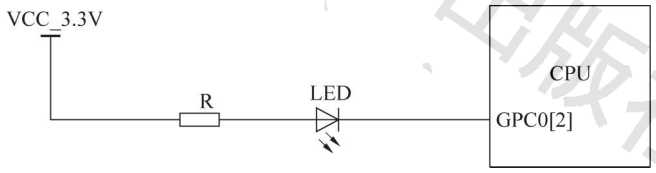

图 2.21 GPIO 端口的引脚连接 LED 发光二极管

1) 问题分析

若要使一个 LED 发光二极管点亮,必须有一个正向电压,即寄存器引脚端必须是低电平。反之,若要使 LED 发光二极管熄灭,则寄存器引脚端必须为高电平。也就是说,寄存器引脚输出低电平时,LED 发光二极管点亮;寄存器引脚输出高电平时,LED 发光二极管熄灭。

2) GPC0 的端口控制寄存器 GPC0CON 的设置

经上述分析,需要把 GPC0[2]引脚设置为输出模式,也就是 GPC0CON[2]引脚设为输出模式。按表 2.4 可知,GPC0CON[2]=$(0001)_2$。

GPC0CON 的设置如图 2.22 所示。

GPC0CON[4]	GPC0CON[3]	GPC0CON[2]	GPC0CON[1]	GPC0CON[0]
[19:16]	[15:12]	[11:8]	[7:4]	[3:0]
0000	0000	0001	0000	0000

图 2.22 GPC0CON 寄存器的设置

所以,设置 GPC0CON[2]为输出模式的值用二进制表示为:

0000 0000 0001 0000 0000

也可以表示为:(1 << 8)

即:GPC0CON=(1 << 8)

3) 端口数据寄存器 GPC0DAT 的设置

GPC0DAT 有 5 位([4:0]),每一位对应一个 GPIO 端口引脚。当该寄存器的某位设置为 1 时,则对应引脚输出高电平;该寄存器的某位设置为 0 时,对应引脚输出低电平。

所以,在 GPC0CON[2]已经设置为输出模式的前提下,GPC0DAT 设置为 0x04 时,GPC0[2]引脚输出高电平,GPC0DAT 设置为 0x00 时,GPC0[2]引脚输出低电平。

即:

GPC0DAT=0x04 时,GPC0[2]引脚输出高电平,LED 发光二极管熄灭。

GPC0DAT=0x00 时,GPC0[2]引脚输出低电平,LED 发光二极管点亮。

本章小结

本章介绍了嵌入式系统相关的基础知识，嵌入式系统硬件平台的基本组成，ARM 系列微处理器，以及 Cortex A8 架构 S5PV210 微处理器的 GPIO 寄存器。本章重点要掌握嵌入式系统硬件平台的组成，这是学习和应用嵌入式系统的基础。

习　题

1. 什么是“握手协议”？试叙述“握手协议”的工作过程。
2. 中断处理经过了哪几个阶段？
3. 在嵌入式系统中，JTAG 接口有什么作用？
4. ARM 的设计思想是什么？
5. 试叙述嵌入式最小系统的组成，并说明各部件的作用。
6. 在例 2-1 中，若把连接 LED 发光二极管的 GPIO 引脚更改为 GPC0[1]，要控制 LED 发光二极管点亮或熄灭，则应怎样对该端口的控制寄存器 GPC0CON 和数据寄存器 GPC0DAT 进行设置？

第3章

嵌入式Linux操作系统

本章的知识只需在PC上就可完成。学习本章内容后读者将掌握如下知识。

- Linux基本概念。
- Linux的目录结构。
- Linux的常用命令。
- Linux的文本编辑器。
- Linux系统的启动过程。

视频讲解

3.1 Linux基本概念

Linux的出现，最早开始于一位名叫Linus Torvalds的计算机业余爱好者，当时他是芬兰赫尔辛基大学的学生。他的目的是想设计一个代替Minix（由一位名叫Andrew Tannebaum的计算机教授编写的一个操作系统示教程序）的操作系统，这个操作系统可用于386、486或奔腾处理器的个人计算机上，并且具有Unix操作系统的全部功能，因而开始了Linux雏形的设计。

嵌入式系统其发展已有20多年的历史，国际上也出现了一些著名的嵌入式操作系统，如VxWorks，Palm OS，Windows CE等，但这些操作系统均属于商品化产品，价格昂贵且由于源代码不公开导致了诸如对设备的支持、应用软件的移植等一系列的问题。而Linux作为一种优秀的自由软件，近几年在嵌入式领域异军突起，成为有潜力的嵌入式操作系统。

从应用上讲，Linux有4个主要部分：内核、Shell、文件系统和实用工具。

1. Linux内核

Linux内核是整个Linux系统的灵魂，Linux系统的能力完全受内核能力的制约。Linux内核负责整个系统的内存管理、进程调度和文件管理。Linux内核的容量并不大，一般一个功能比较全面的内核也不会超过1MB，而且大小可以裁减，这个特性对于嵌入式是非常有好处的。合理地配置Linux内核是嵌入式开发中很重要的环节，对内核的充分了解是嵌入式Linux开发的基本功。

下面简单介绍Linux内核功能，Linux内核的功能大致有如下几个部分。

1）进程管理

进程管理功能负责创建和撤销进程，以及处理它们和外部世界的连接。不同进程之间的通信是整个系统的基本功能，因此也由内核处理。除此之外，控制进程如何共享CPU资源的调度程序也是进程管理的一部分。概括地说，内核的进程管理活动就是在单个或多个

CPU 上实现多进程的抽象。

2）内存管理

内存是计算机的主要资源之一，用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源上为每个进程都创建了一个虚拟寻址空间。内核的不同部分在和内存管理子系统交互时使用一套相同的系统调用。

3）文件管理

Linux 在很大程度上依赖于文件系统的概念，Linux 中的每个对象都可以被视为文件。

4）设备控制

几乎每个系统操作最终都会映射到物理设备上。除了处理器、内存以及其他有限的几个实体外，所有的设备控制操作都由与被控制设备相关的代码来完成，这段代码叫作设备驱动程序。内核必须为系统中的每个外设嵌入相应的驱动程序。

5）网络功能

网络功能也必须由操作系统来管理，因为大部分网络操作都和具体的进程无关。在每个进程处理这些数据之前，数据报必须已经被收集、标识和分发。系统负责在应用程序和网络之间传递数据。另外，所有的路由和地址解析问题都由内核进行处理。

2. Linux Shell

Shell 是 Linux 系统下的命令解释器，它提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行，类似于 Microsoft Windows 的 Command 命令。

Linux 内核与界面是分离的，它可以脱离图形界面单独运行，同样也可以在内核的基础上运行图形化的桌面。在 Linux 的图形用户界面(GUI)下，终端窗口就是 Shell。

每个 Linux 系统的用户可以拥有自己的用户界面或 Shell，用以满足自身专门的 Shell 需要。

3. Linux 文件系统

Linux 的文件系统和 Microsoft Windows 的文件系统有很大的不同。Linux 只有一个文件树，整个文件系统是以一个树根/为起点的，所有的文件和外部设备都以文件的形式挂接在这个文件树上，包括硬盘、软盘、光驱、调制解调器等，这和以“驱动器盘符”为基础的 Microsoft Windows 系统有很大区别。

Linux 的文件系统如图 3.1 所示。

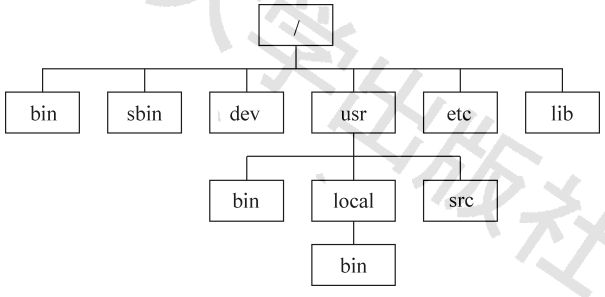

图 3.1　Linux 文件系统的目录结构

下面对各主要目录做一个简要的介绍。

1) /bin 和/sbin

这两个目录通常存放 Linux 基本操作命令的执行文件,其中的内容是一样的,二者的主要区别是:/sbin 中的程序只能由 root(系统管理员)来执行。

2) /dev

这是一个非常重要的目录,它存放着各种外部设备的镜像文件。在 Linux 中,所有的设备都当作文件进行操作。例如,第一个硬盘的名字是 hda,硬盘中的第一个分区是 hda1,第二个分区是 hda2,第一个光盘驱动器的名字是 hdc。用户可以非常方便地像访问文件一样对外部设备进行访问。

3) /lib

该目录用来存放系统动态链接共享库。Linux 系统内核内置的已经编译好的驱动程序存放在/lib/modules/kernel 目录下。几乎所有的应用程序都会用到这个目录下的共享库。因此,不要轻易对这个目录进行操作。

4) /usr

该目录用来存放用户应用程序和文件,类似于 Windows 下的 ProgramFiles 目录。Linux 系统内核的源码存放在 usr/src/kernels 目录下。

5) /etc

该目录用来存放系统的各种配置文件,系统在启动过程中需要读取其参数进行相应的配置。其中:

- /etc/rc.d 目录存放启动或改变运行级时运行的脚本文件及目录。
- /etc/passwd 文件为用户数据库,其中的字段给出了用户名、真实姓名、起始目录、加密的口令和用户的其他信息。
- /etc/profile 文件为系统环境变量的配置文件。

内核、Shell 及文件系统一起形成了基本的操作系统结构。它们使得用户可以运行程序、管理文件以及使用系统。此外,Linux 操作系统还有许多被称为实用工具的程序,能辅助用户完成一些特定的任务。

3.2 Linux 常用操作命令

通常,在完成 Linux 安装后,就可以进入与 Windows 类似的图形化窗口界面。这个界面是 Linux 图形化界面 X 窗口系统的一部分。要注意的是,X 窗口系统仅仅是 Linux 的一个应用软件(或称为服务),它不是 Linux 自身的一部分。为了让 Linux 系统能高效、稳定地工作,建议读者尽可能地使用 Linux 的命令行界面,也就是在 Shell 环境下工作。

Linux 中运行 Shell 的环境是图形用户界面下的“终端”窗口,读者可以单击“终端”启动 Shell 环境。

Linux 中的命令非常多,本节只讲解最常用的一些操作命令。要了解更多的命令使用方法,请查阅相关资料和书籍。

3.2.1　文件目录相关命令


视频讲解

Linux 中有关文件目录的操作是最常用的，与文件目录相关的命令如表 3.1 所示。

表 3.1　文件目录相关命令

命　令	命令含义	程序所在目录
ls	显示文件名(相当于 DOS 的 dir 命令)	/bin
cd	切换目录(相当于 DOS 的 cd 命令)	Shell 内部提供
cp	复制文件(相当于 DOS 的 copy 命令)	/bin
mkdir	创建新目录(相当于 DOS 的 md 命令)	/bin
rm	删除文件(相当于 DOS 的 del 命令)	/bin
rmdir	删除空目录(相当于 DOS 的 rd 命令)	/bin
mv	移动文件，另兼有更换文件名的作用	/bin
pwd	显示目前所在目录	/bin
cat	显示文本文件内容	/bin
env	查看环境设置	/usr/bin
find	查找文件	/usr/bin
grep	寻找某字串内容	/bin
more	分屏显示文本文件内容或输出结果	/bin
mtools	与 MS-DOS 兼容的操作命令集	/usr/bin
su	用于切换用户	/bin
df	查看磁盘使用情况	/bin
uname	查看当前 Linux 版本信息，使用时要带参数-r	/bin

1. ls 命令

1) 作用

ls 的功能为列出目录的内容。该命令类似于 DOS 下的 dir 命令。

2) 命令格式

```
ls [-选项] [目录或文件名]
```

3) 命令选项

- -a：显示指定目录下所有子目录与文件名，包括隐藏文件。
- -l：以长格式来显示文件的详细信息。

4) 示例

查看当前根目录下的文件。

```
[root@localhost  /]# ls
bin    dev    home    lib        misc    opt    root    tftpboot    usr
boot etc   initrd   lost+found   mnt     proc    sbin    tmp     var
```

查看当前 root 目录下的所有文件，包括隐藏文件。

```
[root@localhost  root]#ls  -a
..esd_auth            .gtkrc                .tcshrc      ebook
...fonts.cache-1      .gtkrc-1.2-gnome2     .viminfo     tmp
```

查看当前 root 目录下的文件属性。

```
[root@localhost  root]#ls  -l
total  72
-rw-r--r--     1  root  root      965    Jan  3  2007    anaconda-ks.cfg
drwx------     4  root  root      4096   Jan  3  2007    evolution
-rw-r--r--     1  root  root      49492  Jan  3  2007    install.log
```

每行显示的信息依次是：文件类型与权限、链接数、文件属主、文件属组、文件大小、建立或最近修改的时间、文件名。

显示的信息中，开头是由 10 个字符构成的字符串，其中第一个字符表示文件类型，它可以是下述类型之一。

- -：普通文件
- d：目录
- l：符号链接
- b：块设备文件
- c：字符设备文件

后面的 9 个字符表示文件的访问权限，分为 3 组，每组 3 位。

第 1 组表示文件属主的权限，第 2 组表示同组用户的权限，第 3 组表示其他用户的权限。每一组的 3 个字符分别表示对文件的读、写和执行权限。

2. 文件权限的表示

用户对文件的读、写和执行权限(简称文件权限)如下所示。

- r：读权限。
- w：写权限。
- x：执行权限，对于目录，表示可进入权限。

文件权限也可用数字表示，其约定如下。

- 数字 0：无权限。
- 数字 1：可执行。
- 数字 2：写权限。
- 数字 4：读权限。

可用数字求和来表示多权限的组合。

例如，用户对某一文件拥有可读、可写、可执行的权限，则可表示为 7(1+2+4=7)，对另一文件拥有可读、可执行的权限，则可表示为 5(1+4=5)。

若对文件拥有可读、可写的权限，则可表示为 6(4+2=6)，若对文件拥有可写、可执行的权限，则可表示为 3(1+2=3)。

总结文件权限的对应关系，如表 3.2 所示。

表 3.2 权限对应关系

字符表示	数字表示	对应权限	字符表示	数字表示	对应权限
-	0	无权限	wx	3	写和执行
x	1	只能执行	rx	5	读和执行
w	2	只写	rw	6	读和写
r	4	只读	rwx	7	读、写和执行

有时，用 3 位数字来表示文件权限，其中每位数字分别表示文件拥有者、同组用户、不同组用户的权限。

例如：

600：表示文件拥有者具有读写权限，其他用户均无任何操作权限。

777：表示文件拥有者、同组用户、不同组用户均具有读、写和执行的权限。

3. cd 命令

1）作用

改变工作目录，该命令与 DOS 下的 cd 命令作用是相同的。

2）命令格式

```
cd  [目录路径/]目录名
```

3）示例

将目录/usr/test 设为当前目录。

```
[root@localhost  root]#cd  /usr/test
[root@localhost  test]#pwd
/usr/test
```

命令 pwd 显示当前目录的绝对路径(从根目录/开始)。

4. mkdir 命令

1）作用

创建一个目录，该命令类似于 DOS 下的 md 命令。

2）命令格式

```
mkdir  [目录路径/新目录名]
```

5. cp 命令

1）作用

复制文件，可以使用通配符，该命令类似于 DOS 下的 copy 命令。

2）命令格式

```
cp  [选项]  [源文件路径]源文件名  目标路径[目标文件名]
```

3）示例

在/tmp 目录下，新建一个子目录 mysub，并将/usr/test 目录下的所有文件复制到 mysub 目录下：

```
[root@localhost  root]#mkdir  /tmp/mysub
[root@localhost  root]#cp  /usr/test/*.*  /tmp/mysub
```

6. rm 命令和 rmdir 命令

1）作用

- rm 为删除指定文件，可以使用通配符，该命令类似于 DOS 下的 del 命令。
- rmdir 为删除指定的目录，该目录必须为空目录。

2) 命令格式

- rm [选项] 文件名
- rmdir 目录路径/目录名

3) 命令选项

rm 的命令选项如下。

- -i: 询问是否删除(y 表示是,n 表示否)。
- -f: 不询问是否删除。
- -r: 递归删除整个目录,同 rmdir。

4) 示例

删除前面示例中在/tmp 目录下建立的子目录 mysub。由于 rmdir 只能删除空目录,因此,要先将该目录下的所有文件删除。

```
[root@localhost root]#rm -f /tmp/mysub/*.*
[root@localhost root]#rmdir /tmp/mysub
```

7. cat 命令

1) 作用

cat 为在屏幕上显示文本文件内容的命令。

2) 命令格式

```
cat 文件名
```

3) 示例

设有一个文本文件 a.txt,应用 cat 显示其内容:

```
[root@localhost abc]#cat a.txt
Hello,I'm writing to this file
```

cat 命令也常用于查看当前 Linux 系统的版本,例如:

```
[root@localhost root]# cat /proc/version
Linux version 2.6.35-22-generic (buildd@rothera) (gcc version 4.4.5 (Ubuntu/
Linaro 4.4.4-14ubuntu4) ) #33-Ubuntu SMP Sun Sep 19 20:34:50 UTC 2010
```

8. pwd 命令

1) 作用

pwd 命令用来查看当前工作目录的完整路径。

2) 命令格式

```
pwd
```

3) 示例

显示当前所在的目录路径。

```
[root@localhost abc]# pwd
/mnt/abc
```

3.2.2　磁盘及系统操作


视频讲解

在 Linux 中与磁盘操作及系统操作相关的命令如表 3.3 所示。

表 3.3　与磁盘及系统操作相关的命令

命　　令	命 令 含 义	程序所在目录
fdisk	硬盘分区及显示分区状态的工具程序	/sbin
df	检查硬盘所剩(所用)空间	/bin
free	查看当前系统内存的使用情况	/usr/bin
mount	挂载某一设备成为某个目录名称	/bin
umount	取消挂载的设备	/bin
du	检查目录所用的空间	/usr/bin
mkbootdisk	制作启动盘	/sbin
shutdown	整个系统关机	/sbin
reboot	重启系统	/sbin
login	用户登录	/bin
logout	用户注销	Shell 内部提供

1. fdisk 命令

1) 作用

fdisk 命令可以用来给磁盘进行分区,查看磁盘情况等,往往使用参数-l 来显示系统的分区情况。

2) 命令格式

```
fdisk [选项]
```

3) 命令选项

```
-l 显示系统的分区情况
```

4) 示例

```
[root@localhost root]#fdisk -l
Disk /dev/sda: 8589MB, 8589934592 bytes
255 heads,63 sectors/track, 1044 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
  Device Boot        Start       End     Blocks        Id      System
/dev/sda1  *           1          13     104391        83      Linux
/dev/sda2             14        1004    7960207+       86      Linux
/dev/sda3           1005        1044     321300        82      Linux swap
```

2. df 命令

1) 作用

检查硬盘所剩(所用)空间。

2) 命令格式

```
df [选项]
```

3) 命令选项

- -h：以1024KB=1MB的方式显示磁盘的使用情况。
- -H：以1000Bytes为换算单位的方式显示磁盘的使用情况。

4) 示例

```
[root@localhost  root]#df  -h
Filesystem                Size        Used    Avail   Use%    Mounted on
/dev/sda2              7.5G      4.8G     2.4G    67%     /
/dev/sda1              99M       9.3M     85M      10%     /boot
None                   78M        0       78M      0%      /dev/shm
[root@localhost  root]#df  -H
Filesystem                 Size      Used    Avail    Use%     Mounted on
/dev/sda2                  8.1G      5.1G    2.6G     67%     /
/dev/sda1                  104M      9.7M    89M      10%     /boot
None                       82M        0      82M      0%      /dev/shm
```

3. free 命令

1) 作用

free命令的功能是查看当前系统内存的使用情况，它显示系统中剩余及已用的物理内存和交换内存，以及共享内存和被核心使用的缓冲区。

2) 命令格式

```
free  [选项]
```

3) 命令选项

- -b　　以B(字节)为单位显示。
- -k　　以KB为单位显示。
- -m　　以MB为单位显示。

4) 示例

```
[root@localhost  root]#free  -b
            total  used       free      shared    buffers    cached
Mem       162107392  83656784  78450688    0       9613312    33099776
-/+ buffers/cache:    40943616    121163776
Swap:   329003008       0      329003008
[root@localhost  root]#free  -m
            total       used        free      shared    buffers    cached
Mem             154          79          74         0          9         31
-/+ buffers/cache:  39     115
Swap:       313        0         313
```

4. mount 命令

1) 作用

挂载某一设备使之成为某个目录名称。

2) 命令格式

```
mount [选项] <-t 类型> [-o 挂载选项] <设备> <挂载点>
```

3）命令选项

- -t：该参数配合选项用于指定一个文件系统分区的类型。
- -o：该参数配合选项用于指定一个或多个挂载选项。

其具体可供选择的内容见表 3.4 所示。

表 3.4　mount 命令的参数选项

命令参数	对应选项	选项说明
-t	vfat	挂载 Windows 95/98 的 FAT32 文件系统
	ntfs	挂载 WinNT/2000 的文件系统
	hpfs	挂载 OS/2 用的文件系统
	ext2、ext3、nfs	挂载 Linux 用的文件系统
	iso9660	挂载 CD-ROM 光盘
-o	ro，rw	挂载区为只读(ro)或读写(rw)
	async，sync	挂载区为同步写入(sync)或异步写入(async)
	auto，noauto	允许此挂载区被 mount -a 自动挂载(auto)
	dev，nodev	是否允许在此挂载区上建立档案，dev 为可以
	exec，noexec	是否允许此挂载区上拥有可执行的二进制文件
	user，nouser	是否允许此挂载区让普通用户拥有 mount 的权限
	defaults	默认值为 rw，suid，dev，exec，auto，nouser，async
	remount	重新挂载

4）命令使用说明

挂载设备之前，首先要确定设备的类型和设备名称，确定设备名称可通过使用命令 fdisk-l 查看。

要卸载已经挂载的设备，使用 umount 命令。

umount 命令是 mount 命令的逆操作，umount 命令的作用是卸载一个文件系统。例如，将光驱装载到 /mnt/cdrom 目录后，若要取出光盘，必须先使用 umount 命令进行卸载，否则无法取下。它的参数使用方法和 mount 命令是一样的，命令格式如下。

```
umount  <挂载点|设备>
```

5）示例

【例 3-1】 挂载一个 Linux 分区，将其挂载到/mnt 目录下(/mnt 称为挂载点)。

```
[root@localhost  root]# mount -t ext3 /dev/hdb1 /mnt
```

【例 3-2】 挂载硬盘的 Windows 分区，将其挂载到/mnt/wind 目录下。

(1) 用 fdisk-l 查看硬盘的 Windows 分区在 linux 下的设备名称。

```
[root@localhost /]# fdisk -l
Disk /dev/hda: 500.1 GB, 500105249280 bytes
255 heads, 63 sectors/track, 60801 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

  Device Boot      Start         End      Blocks   Id  System
/dev/hda1               1        5100    40965718+  2d  Unknown
```

```
/dev/hda2          5101       60801    447418282+   f  W95 Ext'd (LBA)
/dev/hda5          5101       24223    153605466    b  W95 FAT32
/dev/hda6         24224       43346    153605466   2d  Unknown
/dev/hda7         43347       60801    140207256   2d  Unknown

Disk /dev/sda: 21.4 GB, 21474836480 bytes
255 heads, 63 sectors/track, 2610 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14        2610    20860402+  8e  Linux LVM
```

(2) 将设备名称为/dev/hda5 的 Windows 分区挂载到/mnt/wind。

```
[root@localhost  root]#mount -t vfat  /dev/hda5  /mnt/wind
```

/mnt/wind 称为挂载设备/dev/hda5 的挂载点。在挂载了硬盘的 Windows 分区之后，可直接访问 Windows 下的磁盘内容。

【例 3-3】 挂载 U 盘。

U 盘的挂载方法同硬盘的挂载方法是一样的，先用 fdisk -l 查看 U 盘的设备名称，U 盘一般是以 sdb 出现的。设 U 盘的名称为/dev/sdb1，则其挂载命令如下。

```
[root@localhost  /]#mount  -t  vfat  /dev/sdb1  /mnt/usb
```

卸载 U 盘的命令如下。

```
[root@localhost  /]#umount /mnt/usb
```


视频讲解

3.2.3 打包压缩相关命令

Linux 常用的压缩及解压缩命令如表 3.5 所示。

表 3.5 Linux 常用的压缩及解压缩命令说明

压缩工具	解压工具	压缩文件扩展名	压缩工具	解压工具	压缩文件扩展名
gzip	gunzip	.gz	compress	uncompress	.Z
zip	unzip	.zip	tar	tar	.tar

1. gzip 命令

1) 作用

对单个文件进行压缩或对压缩文件进行解压缩，压缩文件名后缀为.gz。

2) 命令格式

```
gzip  压缩或解压缩文件名
```

3) 命令选项

- -d：对压缩文件进行解压缩。
- -r：递归方式查找指定目录并压缩其中所有文件或解压缩。

- -v：对每个压缩文件显示文件名和压缩比。
- -num：用数值 num 指定压缩比，num 取值 1～9，其中 1 代表压缩比最低，9 代表压缩比最高，默认值为 6。

4）示例

```
[root@localhost  test]# gzip test.txt
[root@localhost  test]# ls
test.txt.gz
```

2. tar 命令

1）作用

对文件进行打包或解包，打包文件名后缀为.tar。利用 tar 命令，可以把多个文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于网络传输是非常有用的。注意，打包与压缩是两个不同的概念，打包只是把多个文件组成一个总的文件，不一定被压缩。

2）命令格式

```
tar ［选项］ 目标文件名　源文件列表
```

3）命令选项

- -A 或--catenate：新增文件到已存在的备份文件。
- -c 或--create：建立新的备份文件。
- -f <备份文件>或--file=<备份文件>：指定备份文件。
- -r 或--append：新增文件到已存在的备份文件的结尾部分。
- -t 或--list：列出备份文件的内容。
- -u 或--update：仅置换较备份文件内的文件更新的文件。
- -v 或--verbose：显示指令执行过程。
- -w 或--interactive：遭遇问题时先询问用户。
- -x 或--extract 或--get：从备份文件中还原文件。
- -z 或--gzip 或--ungzip：通过 gzip 指令处理备份文件。

4）示例

将文件包 filetest.tar.gz 解包的命令如下。

```
#tar - zxvffiletest.tar.gz
```

3.2.4　网络相关命令


视频讲解

Linux 有许多与网络相关的命令，下面介绍几个常用的网络操作命令。

1. ifconfig 命令

1）作用

用于查看和配置网络接口的地址和参数，包括 IP 地址、网络掩码和广播地址。它的使用权限是超级用户。

2）命令格式

- 查看网卡配置信息：ifconfig

· 设置网卡：ifconfig eth0 ［主机 IP 地址］

eth0 代表第 1 块网卡，eth1 代表第 2 块网卡，若主机上仅安装了一块网卡，则为 eth0。

3）示例

```
[root@localhost /]# ifconfig
eth0  Link encap: Ethernet HWaddr 00: 11: 11: 11: 23: 5A
      inet addr: 192.168.1.15 Bcast: 192.168.1.255 Mask: 255.255.255.0
      inet6 addr: fe80::208:2ff:fee0:c18a/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets: 26931 errors: 0 dropped: 0 overruns: 0 frame: 0
      TX packets: 3209 errors: 0 dropped: 0 overruns: 0 carrier: 0
      collisions: 0 txqueuelen: 1000
      RX bytes: 6669382 (6.3 MiB) TX bytes: 321302 (313.7 KiB)
      Interrupt: 11

lo    Link encap: Local Loopback
      inet addr: 127.0.0.1 Mask: 255.0.0.0
      inet6 addr: ::1/128 Scope: Host
      UP LOOPBACK RUNNING MTU: 16436 Metric: 1
      RX packets: 1381 errors: 0 dropped: 0 overruns: 0 frame: 0
      TX packets: 1381 errors: 0 dropped: 0 overruns: 0 carrier: 0
      collisions: 0 txqueuelen: 0
      RX bytes: 1354690 (1.2 MiB) TX bytes: 1354690 (1.2 MiB)
```

重新设置网卡的 IP 地址，改设为 192.168.1.100，则使用以下命令。

```
[root@localhost /]# ifconfig eth0 192.168.1.100
```

注意：用 ifconfig 命令配置的网络参数不需重启就可生效，但机器重新启动以后其设置将会失效。

2. ping 命令

1）作用

ping 命令用于检测网络连接情况，从而判断主机联网是否连接正常。

2）命令格式

```
ping [IP 地址]
```

3）示例

```
[root@localhost /]# ping 192.168.1.1
PING 192.168.1.1  (192.l168.1.1)  56(84) bytes of data.
64 bytes from 192.168.1.1 icmp_seq = 0  ttl = 64  time = 2.439 ms
64 bytes from 192.168.1.1 icmp_seq = 1  ttl = 64  time = 1.149 ms
64 bytes from 192.168.1.1 icmp_seq = 2  ttl = 64  time = 1.043 ms
64 bytes from 192.168.1.1 icmp_seq = 3  ttl = 64  time = 1.038 ms

--- 192.168.1.1 ping statistics ---
4 packets transmitted, 4 received. 0 % packet loss, time 7004ms
rtt min/avg/max/mdev = 1.038/1.924/2.439/1.529 ms, pipe 2
```

3.3　Linux 的文本编辑器

视频讲解

3.3.1　Vi 文本编辑器

Vi 是 Linux 系统的第一个全屏幕交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，足以见其生命力之强，而强大的生命力是其强大的功能带来的。由于大多数读者在此之前都已经用惯了 Windows 的 Word 等编辑器，因此，在刚刚接触 Vi 时总会或多或少不适应，但只要习惯之后，就能感受到它的方便与快捷。

1. Vi 的模式

Vi 有 3 种模式，分别为命令行模式、插入模式和底行模式。各模式的功能具体进行介绍如下。

1）命令行模式

用户在用 Vi 编辑文件时，最初进入的为一般模式。在该模式中可以通过上下移动光标进行“删除字符”或“整行删除”等操作，也可以进行“复制”“粘贴”等操作，但无法编辑文字。

2）插入模式

只有在该模式下，用户才能进行文字编辑输入，用户可按 Esc 键回到命令行模式。

3）底行模式

在该模式下，光标位于屏幕的底行。用户可以进行文件保存或退出操作，也可以设置编辑环境，如寻找字符串、列出行号等。

2. Vi 的基本流程

（1）进入 Vi，即在命令行下输入 Vi hello（文件名）。此时进入的是命令行模式，光标位于屏幕的上方，如图 3.2 所示。

图 3.2　进入 Vi 命令行模式

（2）在命令行模式下输入 i 进入插入模式，如图 3.3 所示。可以看出，在屏幕底部显示有“插入”字样表示插入模式，在该模式下可以输入文字信息。

（3）在插入模式中，按 Esc 键，则当前模式转入命令行模式，此时在底行行中输入：wq（存盘退出）进入底行模式，如图 3.4 所示。

图 3.3 进入 Vi 插入模式

图 3.4 进入 Vi 底行模式

这样,就完成了一个简单的 Vi 操作流程:命令行模式→插入模式→底行模式。由于 Vi 在不同的模式下有不同的操作功能,因此,读者一定要时刻注意屏幕最下方的提示,分清所在的模式。

3. Vi 的各模式功能键

(1) 命令行模式常见功能键如表 3.6 所示。

表 3.6 Vi 命令行模式功能键

功 能 键	功 能 说 明
I	切换到插入模式,此时光标位于开始输入文件处
A	切换到插入模式,并从目前光标所在位置的下一个位置开始输入文字
O	切换到插入模式,且从行首开始插入新的一行
Ctrl+B	屏幕往后翻动一页
Ctrl+F	屏幕往前翻动一页
Ctrl+U	屏幕往后翻动半页
Ctrl+D	屏幕往前翻动半页
0	光标移到本行的开头
G	光标移动到文章的最后
nG	光标移动到第 n 行
$	移动到光标所在行的行尾
n	光标向下移动 n 行

续表

功　能　键	功 能 说 明
/name	在光标之后查找一个名为 name 的字符串
? name	在光标之前查找一个名为 name 的字符串
X	删除光标所在位置的一个字符
dd	删除光标所在行
ndd	从光标所在行开始向下删除 n 行
yy	复制光标所在行
nyy	复制光标所在行开始的向下 n 行
p	将缓冲区内的字符粘贴到光标所在位置(与 yy 搭配)
U	恢复前一个动作

(2) 插入模式的功能键只有一个,也就是 Esc 退出到命令行模式。

(3) 底行模式常见功能键如表 3.7 所示。

表 3.7　Vi 底行模式功能键

功　能　键	功 能 说　明
: w	将编辑的文件保存到磁盘中
: q	退出 Vi(系统对做过修改的文件会给出提示)
: q!	强制退出 Vi(对修改过的文件不作保存)
: wq	存盘后退出
: w [filename]	另存一个名为 filename 的文件
: set nu	显示行号,设定之后,会在每一行的前面显示对应行号
: set nonu	取消行号显示

3.3.2　gedit 文本编辑器


视频讲解

除了 vi 之外,Linux 下还有一个功能同样强大的编辑器 gedit。gedit 是一个 GNOME 桌面环境下兼容 UTF-8 的文本编辑器。它简单易用,有良好的语法高亮,对中文支持很好,支持包括 GB2312、GBK 在内的多种字符编码,是一款自由软件。

gedit 是一个功能强大的文本编辑器,类似于 Windows 系统下面的记事本,它的功能比 Windows 系统的记事本更强大,还具有行号显示、括号匹配、文本自动换行、自动文件备份等功能,适合编写程序代码。

1. gedit 的启动

gedit 的启动方式有多种,可以从菜单启动,也可以从终端命令行启动。从菜单启动时,选择桌面顶部的“应用程序”|“附件”|“文本编辑器”命令即可打开;从终端启动,只需要输入代码 $ gedit 再按 Enter 键即可。

gedit 启动之后的主界面如图 3.5 所示。

2. 窗口说明

读者可以看到 gedit 启动的界面和 Windows 中的“写字板”程序相似。窗口上有菜单栏、工具栏、编辑栏、状态栏等。

图 3.5 gedit 主界面

3. 常用的技巧

1) 打开多个文件

要从命令行打开多个文件,请输入"gedit file1.txt file2.txt file3.txt"命令,然后按下 Enter 键。

2) 将命令的输出输送到文件中

例如,要将 ls 命令的输出输送到一个文本文件中,请输入"ls | gedit",然后按下 Enter 键。ls 命令的输出就会显示在 gedit 窗口的一个新文件中。

3) 更改"突出显示模式"以适用各种文件

例如,更改以适应 html 文件的步骤为,依次选择菜单中的"查看"|"突出显示模式"|"标记语言"|HTML,即可以彩色模式查看 html 文件。

4) 插件

gedit 中有多种插件可以选用,这些插件极大地方便了用户处理代码,常用的包括以下几种。

- 文档统计信息:选择菜单栏中的"工具"|"统计文档"命令,出现"文档统计信息"对话框,里面显示了当前文件中的行数、单词数、字符数及字节数。
- 高亮显示:选择"视图"|"高亮",然后选择需要高亮显示的文本。
- 插入日期/时间:选择"编辑"|"插入时间和日期"命令,则在文件中插入当前时间和日期。
- 跳到指定行:选择"查找"|"进入行"命令,之后输入需要定位的行数,即可跳到指定的行。

5) 常用的快捷键

gedit 常用的快捷键如表 3.8 所示。

表 3.8 gedit 常用的快捷键

快捷键	功能说明	快捷键	功能说明
Ctrl+Z	撤销	Ctrl+Q	退出
Ctrl+C	复制	Ctrl+S	保存
Ctrl+V	粘贴	Ctrl+R	替换
Ctrl+T	缩进		

3.4　Linux 启动过程

了解 Linux 的常见命令之后，下面介绍 Linux 的启动过程。Linux 的启动过程包含了 Linux 工作原理的精髓，在嵌入式系统的开发过程中非常需要这方面的知识积累。

3.4.1　Linux 系统的引导过程

许多人对 Linux 的启动过程感到很神秘，因为所有的启动信息都在屏幕上一闪而过。其实，Linux 的启动过程并不像启动信息所显示的那样复杂，它主要分成以下两个阶段。

(1) 启动内核。在这个阶段，内核装入内存并在初始化每个设备驱动器时打印信息。

(2) 执行程序 init。装入内核并初始化设备后，运行 init 程序。init 程序处理所有程序的启动，包括重要系统精灵程序和其他指定在启动时装入的软件。

首先，当用户打开 PC 的电源后，CPU 将自动进入实模式，这时 BIOS 进行开机自检，并按 BIOS 中设置的启动设备(通常是硬盘)进行启动引导。BIOS 通常是转向硬盘的第一个扇区，寻找用于装载操作系统的指令。装载操作系统的这个程序就是 BootLoader。针对不同的硬件平台，需要有专门的 Bootloader 程序。Bootloader 程序依赖于特定的硬件。

Linux 里面的 BootLoader 通常是 lilo 或者 grub，从 Red Hat Linux 7.2 起，GRUB(GRand Unified Bootloader)取代 lilo 成为默认的启动装载程序。

对于嵌入式 Linux 系统，经常使用另一款功能强大的 BootLoader：Blob。Blob 是 Boot Loader Object 的缩写，它遵循 GPL，源代码完全开放。Blob 既可以用来简单地调试，也可以启动 Linux 内核。

下面以 Red Hat 为例，简单介绍 Linux 在 PC 上运行时的启动过程。

当用户打开 PC 的电源，BIOS 开机自检，按 BIOS 中设置的启动设备(通常是硬盘)启动，接着启动设备上安装的引导程序 lilo 或 grub 开始引导 Linux。Linux 首先进行内核的引导，接下来执行 init 程序。init 程序调用 rc.sysinit 和 rc 等程序，rc.sysinit 和 rc 完成系统初始化和运行服务的任务后，返回 init；init 启动 mingetty 后，打开终端供用户登录系统，用户登录成功后进入 Shell，这样就完成了从开机到登录的整个启动过程。启动流程如图 3.6 所示。

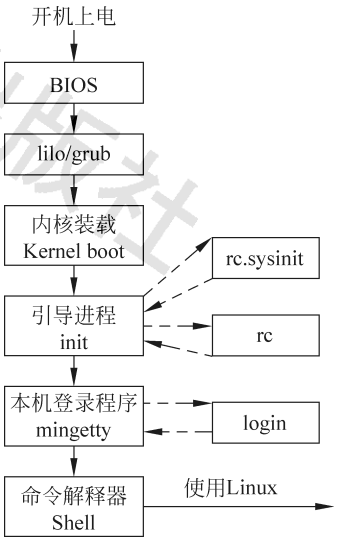

图 3.6　Linux 的启动过程

1. 启动内核

计算机启动时，BIOS 装载 MBR，然后从当前活动分区启动，LILO 获得引导过程的控制权后，会显示 LILO 提示符。此时如果用户不进行任何操作，LILO 将在等待指定时间后自动引导默认的操作系统，而如果在此期间按下 Tab 键，则可以看到一个可引导的操作系统列表，选择相应的操作系统名称就能进入相应的操作系统。

当用户选择启动 Linux 操作系统时，LILO 就会根据事先设置好的信息从 ROOT 文件系统所在的分区读取 Linux 映象，然后装入内核映象并将控制权交给 Linux 内核。Linux

内核获得控制权后,以如下步骤继续引导系统。

(1) Linux 内核一般是压缩保存的,因此,它首先要进行自身的解压缩。内核映象前面的一些代码完成解压缩。

(2) 如果系统中安装有可支持特殊文本模式的、且 Linux 可识别的 SVGA 卡,Linux 会提示用户选择适当的文本显示模式。但如果在内核的编译过程中预先设置了文本模式,则不会提示选择显示模式。该显示模式可通过 LILO 或 RDEV 工具程序设置。

(3) 内核接下来检测其他的硬件设备,例如硬盘、软驱和网卡等,并对相应的设备驱动程序进行配置。这时,显示器上出现内核运行输出的一些硬件信息。

(4) 接下来,内核装载 ROOT 文件系统。ROOT 文件系统的位置可在编译内核时指定,也可通过 LILO 或 RDEV 指定。文件系统的类型可自动检测。如果由于某些原因装载失败,则内核启动失败,最终会终止系统。

2. 执行 init 程序

利用 init 程序可以方便地定制启动期间装入哪些程序。init 的任务是启动新进程和退出时重新启动其他进程。例如,在大多数 Linux 系统中,启动时最初装入 6 个虚拟的控制台进程,退出控制台窗口时,进程死亡,然后 init 启动新的虚拟登录控制台,因而总是提供 6 个虚拟登录控制台。控制 init 程序操作的规则存放在文件/etc/inittab 中。Red Hat Linux 默认的 inittab 文件如下。

```
#inittab This file describes how the INIT process should set up the system in a certain
#run-level.
#Default runlevel.The runlevels used by RHS are:
#0-halt(Do NOT set initdefault to this)
#1-Single user mode
#2-Multiuser,without NFS(the same as 3,if you do not have networking)
#3-Full multiuser mode
#4-unused
#5-X11
#6-reboot(Do NOT set initdefault to this)
id:3:initdefault:

#system initialization
si::sysinit:/etc/rc.d/rc.sysinit
10:0:wait:/etc/rc.d/rc 0
11:1:wait:/etc/rc.d/rc 1
12:2:wait:/etc/rc.d/rc 2
13:3:wait:/etc/rc.d/rc 3
14:4:wait:/etc/rc.d/rc 4
15:5:wait:/etc/rc.d/rc 5
16:6:wait:/etc/rc.d/rc 6
#Things to run in every runlevel
ud:once:/sbin/update

#Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

#When our UPS tells us power has failed,assume we have a few minutes of
```

```
power left. Schedule a
# shutdown for 2 minutes from now.
# This does, of course, assume you have powered installed and your UPS
connected and working
# correctly.
pf::powerfail: /sbin/shutdown -f -h 2 "Power Restored; Shutdown Cancelled"

# Run gettys in standard runlevels
1: 2345: respawn: /sbin/minggetty tty1
2: 2345: respawn: /sbin/minggetty tty2
3: 2345: respawn: /sbin/minggetty tty3
4: 2345: respawn: /sbin/minggetty tty4
5: 2345: respawn: /sbin/minggetty tty5
6: 2345: respawn: /sbin/minggetty tty6
# Run xdm in runlevel 5

x: 5: respawn: /usr/bin/X11/xdm -nodaemon
```

Linux 有个运行级系统，运行级是表示系统当前状态和 init 应运行哪个进程并保持在这种系统状态中运行的数字。在 inittab 文件中，第一个项目指定启动时装入的默认运行级。

上例中是个多用户控制台方式，运行级为 3。然后，inittab 文件中每个项目指定第 2 个字段的项目用哪种运行级（每个字段用冒号分开）。因此，对运行级 3，下列行是相关的。

```
13: 3: wait: /etc/rc.d/rc 3
1: 2345: respawn: /sbin/minggetty tty1
2: 2345: respawn: /sbin/minggetty tty2
3: 2345: respawn: /sbin/minggetty tty3
4: 2345: respawn: /sbin/minggetty tty4
5: 2345: respawn: /sbin/minggetty tty5
6: 2345: respawn: /sbin/minggetty tty6
```

最后 6 行建立 Linux 提供的 6 个虚拟控制台。第一行运行启动脚本/etc/rc.d/rc 3，将运行目录/etc/rc.d/rc3.d 中包含的所有脚本，这些脚本表示系统初始化时要启动的程序。一般来说，这些脚本不需要编辑或改变，是系统默认的。

3.4.2　ARM Linux 操作系统

ARM Linux 是一种常见的嵌入式操作系统，主要运行在以 ARM 为核心的处理器上。根据运行的层次，可以划分为三大部分：启动引导（Bootloader）、操作系统内核（Linux Kernel）和文件系统（File System）。

启动引导程序 Bootloader 非常像 PC 中的 BIOS 程序，主要负责初始化系统的最基本设备，通常主要包括 CPU、网络、串行接口。当基本部分初始化成功后，会把操作系统的镜像文件装载到内存中，最后把 CPU 的控制权交给内核程序。

内核接管系统后，会重新检查外部器件的运行状态，初始化所有外部硬件设备，加载驱动程序，检查系统参数表，装载文件系统，运行 SHELL 程序，等待用户输入命令，或直接运行设定好的应用程序。内核在运行的过程中，会把基本的初始化信息打印到终端（通常是串口 0

或LCD),并且通过终端接收用户命令,它负责控制应用程序的运行状态,实现对整个系统的控制。Linux内核是Linux的最核心部分,内核的优劣决定了整个系统是否稳定与高效。

文件系统是一种数据结构,使操作系统明确存储介质(Flash或硬盘等)上的文件,即在存储介质上组织文件的方法。文件系统通常占用大部分的存储空间,主要负责保存应用程序和数据,由Linux内核管理。

Bootloader、KERNEL、FS(FILE SYSTEM)都存储在Flash中,运行时,根据需要被加载到内存里。图3.7给出了板上内存的地址空间分布:MEMORY MAP。

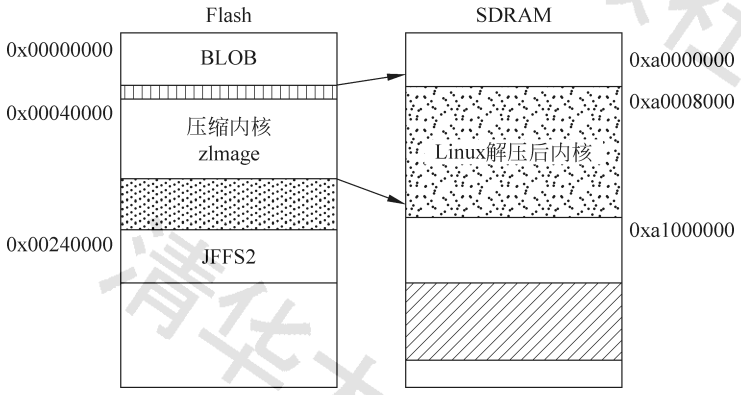

图3.7 内存的地址空间分布:MEMORY MAP

3.5 数据共享与数据传输


视频讲解

3.5.1 应用串口通信协议传输数据

1. 串口通信协议

串口通信协议由Xmodem、Ymodem、Zmodem等协议组成。

Xmodem协议是一种应用于串口通信的文件传输协议。这种协议以包为传输信息的单位来传输数据,并且每个包都使用一个校验和过程来进行错误检测。1个包=128字节,传输速度较慢。

Ymodem协议由Xmodem协议演变而来,传输效率及可靠性均较高,它的1个包=1024字节。Ymodem一次传输可发送或接收几个文件。

Zmodem协议也是由Xmodem协议演变而来,以连续的数据流发送数据,传输效率更高。

2. Windows系统主机传输文件到Linux系统开发板

当需要把Windows系统主机的文件传输到Linux系统开发板时,可以使用本方法来实现。首先,用串口通信数据线连接Windows系统主机和Linux系统开发板,如图3.8所示。

1) 在Windows系统主机端设置发送文件

在Windows系统主机的桌面"开始"菜单中,选择"程序"|"附件"|"通信"|"超级终端"项,打开"连接描述"对话框,填写超级终端连接的名称,单击"确定"按钮,如图3.9所示。Windows 7以后版本没有自带"超级终端",可以从网络上搜索下载"超级终端"软件。

图 3.8　用串口通信数据线连接 Windows 系统主机和 Linux 系统开发板

在打开的“连接到”对话框中选择串口通信的连接端口为 COM1 端口，单击“确定”按钮，如图 3.10 所示。

图 3.9　设置超级终端的连接名称

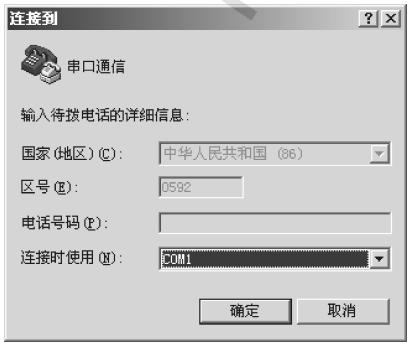

图 3.10　选择 COM1 为连接端口

在打开的“COM1 属性”对话框中，设置端口的各个重要参数值：每秒位数（波特率）为 115200，数据位为 8 位，奇偶校验为“无”，停止位为 1，数据流控制为“无”，如图 3.11 所示。设置好端口的参数值后，单击“确定”按钮。

图 3.11　设置端口参数

2）在 Linux 系统开发板端设置接收文件

在开发板端设置接收文件的操作很简单，只需通过 minicom 窗口，进入准备接收数据文件的目录等待发送来的文件即可。

3）发送数据

在 Windows 系统主机端，继续前面的“超级终端”窗口操作。

在超级终端的串口通信窗口的“发送”菜单中，选择“发送文件”项，如图 3.12 所示。

在弹出的“发送文件”对话框中，单击“浏览”按钮，选择需要传送的数据文件；然后在“协议”下拉列表框中，选择 Xmodem 协议，如图 3.13 所示。

这时，在“为串口通信发送 Xmodem 文件”窗口可以看到数据传送的过程，如图 3.14 所示。

文件传输完毕后，在开发板的接收数据文件的目录中可以看到传送的文件。

图 3.12 在超级终端选择“发送文件”菜单项

图 3.13 选择发送的文件和 Xmodem 协议

图 3.14 传输数据

3. Linux 系统主机传输数据到 Linux 系统开发板

经常需要把在 Linux 系统主机上经过交叉编译后的文件传输到 Linux 系统开发板运行,可以使用本方法来实现传送文件。

首先,用串口通信数据线连接 Linux 系统主机和 Linux 系统开发板,如图 3.15 所示。

1) 在开发板端设置接收文件

通过 minicom 窗口操作开发板端文件系统,进入准备接收数据文件的目录,等待发送来的文件。

图 3.15　用串口通信数据线连接 Linux 系统主机和 Linux 系统开发板

2）从 Linux 系统主机端发送文件

在 minicom 窗口中，按下 Ctrl＋A＋S 快捷键，弹出选择传输数据协议的对话框，如图 3.16 所示。

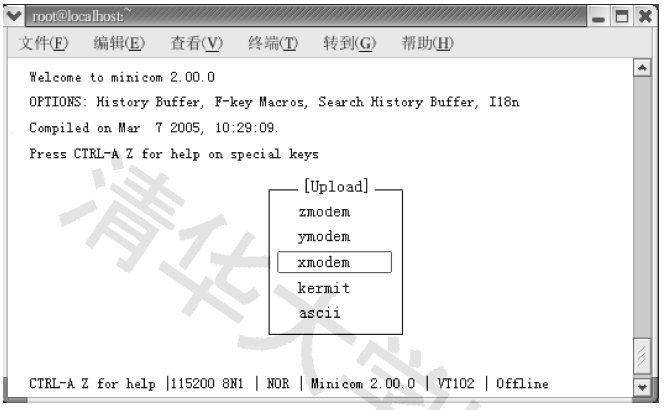

图 3.16　在 minicom 窗口中弹出选择传输数据协议的对话框

用键盘上的方向键移动光标至 xmodem 项后，按 Enter 键确定。进入 Linux 系统主机端的文件系统，按“上”“下”方向键移动文件目录前面的小方块，按两次空格键进入选定的目录，若要返回到上一级目录，则选中[..]后按两次空格键，如图 3.17 所示。

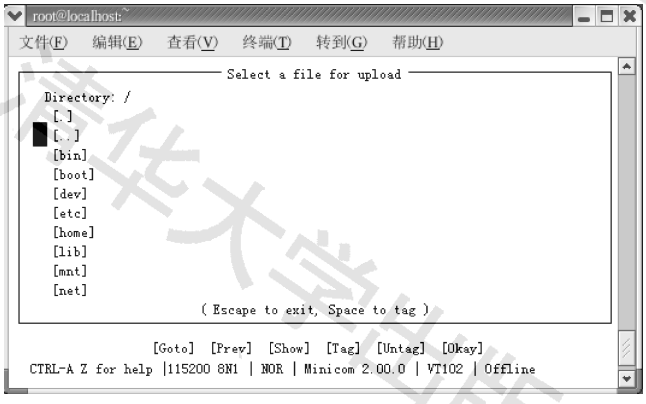

图 3.17　按“上”“下”方向键移动小方块，选择需要的文件目录

进入需要的文件目录后，按“上”“下”方向键移动小方块，选择需要传送的文件。找到需要的文件按空格键选中该文件；若要取消选中，则再次按空格键，如图 3.18 所示。

选中需要传输的文件，按 Enter 键，则开始发送文件。当显示 Transfer incomplete 时，表示文件传输完毕，如图 3.19 所示。

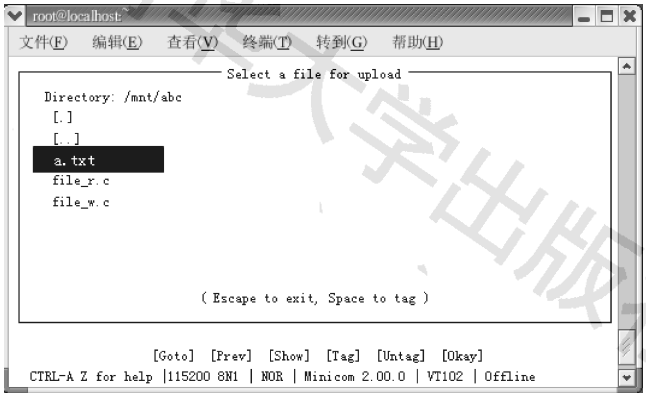

图 3.18 按空格键选中需要的文件

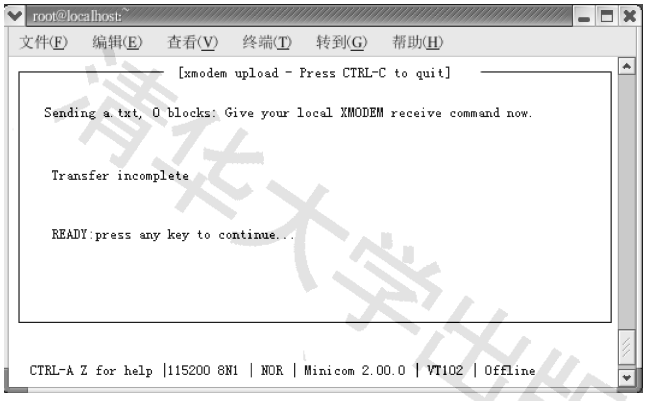

图 3.19 发送文件完毕

视频讲解

3.5.2 在 VMware 虚拟机中设置 Windows 与 Linux 系统的数据共享

在 VMware 虚拟机中可以设置 Windows 与 Linux 系统的共享。设 Windows 操作系统的 VMware 中安装有 Linux 操作系统,通过 VMware 虚拟机可以设置 Windows 与 Linux 系统的共享,如图 3.20 所示。

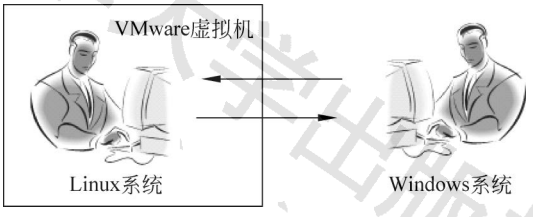

图 3.20 在 VMware 虚拟机中设置 Windows 与 Linux 系统的数据共享

1. 安装 VMware Tools

在 VMware 虚拟机中选择“虚拟机(VM)”菜单,在弹出的下拉菜单中选择 Install VMware Tools 项,Linux 系统桌面上会出现一个名为 VMware Tools 的光盘图标。

双击 VMware Tools 光盘图标，打开光盘，复制 VMware Tools. tar. gz 文件到/home 目录下，将其解压至/home/vmware-tools-distrib 目录下。进入安装目录/home/vmware-tools-distrib 中，在终端运行如下命令。

```
./vmware - install.pl
```

安装过程中会有一些文件安装路径的提示问题，按 Enter 键即可，如图 3.21 所示。

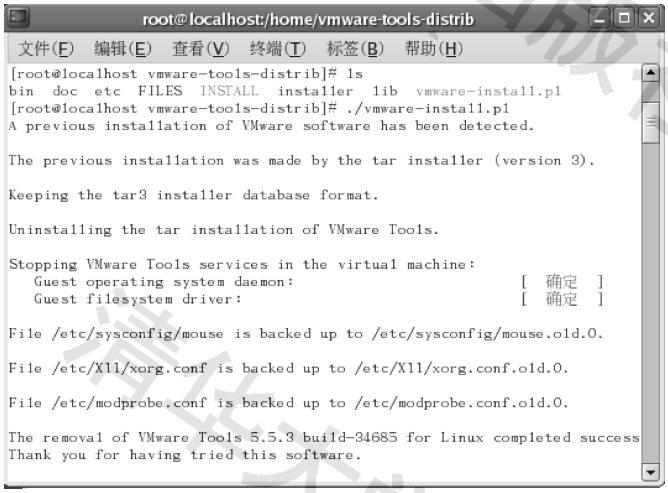

图 3.21　在终端运行 vmware-install. pl

2. 设置共享文件夹

选择 VMware 虚拟机“虚拟机(VM)”菜单中的“设置(Settings)”项，弹出“虚拟机设置”对话框。选择“选项”选项卡，在左侧选择“共享文件夹”项，然后单击右侧的“添加”按钮，添加 Windows 系统中的共享文件夹，如图 3.22 所示。

图 3.22　设置 Windows 系统的共享文件夹

3. 在 Linux 系统中操作 Windows 系统的共享文件夹

在 Linux 系统中,打开/mnt 目录,可以看到其中存在一个 hgfs 目录。打开/mnt/hgfs 目录,可以看到 Windows 系统的共享文件夹,如图 3.23 所示。因而在 Linux 系统中,可以很方便地对这些共享文件夹的文件进行复制、删除、修改等操作。

图 3.23 在 Linux 系统中操作 Windows 系统的共享文件夹

本章小结

本章介绍了 Linux 的基本概念、Linux 文件系统的概念、嵌入式 Linux 系统中常用的命令、Linux 系统的文本编辑器的使用、Linux 系统的启动过程等。这些都是 Linux 中最基础、最常见的概念,掌握和理解这些知识对进一步学习和使用 Linux 系统有很大帮助,因此,必须多上机练习,熟练掌握它们。

习 题

1. 查看 Linux 目录结构,说出下列目录放置的数据类型。

```
/etc/:
/etc/rc.d/init.d/:
/usr/bin:
/bin:
/sbin:
/dev:
```

2. Bootloader 有什么作用? 为什么不作为操作系统的一部分加以实现?

3. 叙述在主机端配置 NFS 服务的过程。

4. 应用串口协议传输,把主机端的一个文件传输到开发板上,并记录下操作过程。

5. 在 VMware 虚拟机中建立 Windows 操作系统与 Linux 操作系统的数据共享文件目录。

第4章

嵌入式Linux程序开发基础

本章主要介绍嵌入式Linux操作系统下程序设计所需要的基础知识。学习本章后，读者应掌握如下知识。

- 嵌入式Linux编译器GCC的使用。
- “文件包含”处理。
- Make命令和Makefile文件。
- 嵌入式Linux汇编语言程序设计基础知识。
- Linux Shell编程方法。
- 位运算。

4.1 嵌入式Linux编译器

目前，嵌入式系统的主流编程语言是C语言，它的强大功能和可移植性让它能在各种硬件平台上应付自如。C语言兼有汇编语言和高级语言的优点，既适合于开发系统软件，也适合于编写应用程序。

4.1.1 Linux下C语言编译过程

Linux下的C语言程序设计主要涉及编辑器、编译链接器、调试器及项目管理工具。

1. 编辑器

嵌入式Linux下的编辑器与微软Windows下的Word、记事本等文字编辑工具一样，完成对源程序代码的编辑功能。最常用的编辑器有vi和gedit等。

2. 编译链接器

编译过程包括词法、语法和语义的分析、中间代码的生成和优化、符号表的管理和出错处理等。在嵌入式Linux中，最常用的编译器是GCC编译器。GCC编译器是GNU推出的功能强大、性能优越的多平台编译器，其执行效率与一般编译器相比平均要高20%～30%。

3. 调试器

调试器可以方便程序员调试程序，但不是代码执行的必要工具。在编程的过程中，调试所消耗的时间远远大于编写代码的时间。因此，有一个功能强大、使用方便的调试器是很必要的。

4. 项目管理器

嵌入式Linux中的项目管理器make类似于Windows中Visual C++中的“工程”，它是一种控制编译或者重复编译软件的工具。另外，它还能自动管理软件编译的内容、方式和时

机,使程序员能够把精力集中在代码的编写上而不是在源代码的组织上。

4.1.2 GCC 编译器及基本使用方法

在嵌入式 Linux 中,通常使用 GCC 编译器将源代码编译成执行程序,下面对 GCC 编译器及使用方法作详细介绍。

1. GCC 编译器

Linux 系统下的 GCC(GNU C Compiler)是 GNU 项目所推出的功能强大、性能优越的多平台编译器。GCC 可以在多种硬件平台上编译出可执行程序,其执行效率与一般的编译器相比要高 20%~30%。因此,特别适合在嵌入式系统开发编译应用程序。

GCC 编译器能将 C、C++语言源程序、汇编语言源程序和目标程序编译、连接成可执行文件,如果没有给出可执行文件的名字,GCC 将自动生成一个名为 a.out 的文件。

在 Linux 系统中,可执行文件没有统一的后缀,系统从文件的属性来区分可执行文件和不可执行文件。而 GCC 则通过文件名后缀来区别文件的类别,表 4.1 是 GCC 所遵循的部分约定规则。

表 4.1 GCC 所支持文件名后缀的部分约定规则

文件名后缀	对应的语言
.a	由目标文件构成的档案库文件
.C,.cc 或.cxx	C、C++源代码文件
.h	程序所包含的头文件
.i	已经预处理过的 C 源代码文件
.ii	已经预处理过的 C++源代码文件
.m	Objective-C 源代码文件
.o	编译后的目标文件
.s	汇编语言源代码文件
.S	经过预编译的汇编语言源代码文件

GNU 计划:GNU 是由 Richard Stallman 开发的一个与 UNIX 兼容的软件系统。它的目标是创建一套完全自由的操作系统。大多数 Linux 软件是经过自由软件基金会的 GNU(www.gnu.org)公开认证授权的,因而通常被称为 GNU 软件。GNU 软件免费提供给用户使用,许多流行的 Linux 实用程序(如 C 编译器、Shell 和编辑器)都是 GNU 软件应用程序。

GNU 计划有很多实用程序,最常用的有文件实用程序、文本实用程序、查找实用程序和 Shell 实用程序。这些实用程序都包含在 Linux 系统中,安装 Linux 操作系统时,会自动安装到计算机中。

2. GCC 的执行过程

虽然称 GCC 是 C 语言的编译器,但使用 GCC 由 C 语言源代码文件生成可执行文件的过程不仅仅是编译的过程,而是要经历 4 个相互关联的步骤:预处理(也称预编译,Preprocessing)、编译(Compilation)、汇编(Assembly)和链接(Linking)。

下面通过一个具体的例子来说明 GCC 是如何完成上述 4 个步骤的。

设有程序 hello.c，其源代码如下。

```
#include <stdio.h>
int main()
 {
    printf("Hello! This is our embedded world!\n");
    return 0;
}
```

1）预处理阶段

在该阶段，命令 GCC 首先对源代码文件中的文件包含(include)、预编译语句(如宏定义 define 等)进行分析。编译器将上述代码中的 stdio.h 编译进来，并且用户可以使用 GCC 的选项-E 进行查看，该选项的作用是让 GCC 在预处理结束后停止编译过程。

GCC 命令的一般格式如下。

```
gcc [选项] 要编译的文件 [选项] [目标文件]
```

其中，目标文件可缺省，若目标文件缺省，则 GCC 默认生成可执行的文件，命名为：编译文件名.out。

```
[root@localhost GCC]#gcc -E hello.c -o hello.i
```

在此处，选项-o 指目标文件，由表 4.1 可知，.i 文件为已经过预处理的 C 原始程序。以下列出了 hello.i 文件的部分内容。

```
typedef int (*__gconv_trans_fct) (struct __gconv_step *,
       struct __gconv_step_data *, void *,
       __const unsigned char *,
       __const unsigned char **,
       __const unsigned char *, unsigned char **,
       size_t *);
…
#2 "hello.c" 2
int main()
 {
    printf("Hello! This is our embedded world!\n");
    return 0;
 }
```

由此可见，GCC 确实进行了预处理，它把 stdio.h 的内容插入 hello.i 文件中。

2）编译阶段

在这个阶段，GCC 首先要检查代码的规范性、是否有语法错误等，以确定代码实际要做的工作，在检查无误后，GCC 把代码翻译成汇编语言。用户可以使用-S 选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

```
[root@localhost GCC]#gcc -S hello.i -o hello.s
```

以下列出了 hello.s 的内容，可见 GCC 已经将其转化为汇编了，感兴趣的读者可以分析

一下这一段简单的C语言语句是如何用汇编代码实现的。

```
.file "hello.c"
.section .rodata
.align 4
.LC0:
.string "Hello! This is our embedded world!"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
movl $0, %eax
addl $15, %eax
addl $15, %eax
shrl $4, %eax
sall $4, %eax
subl %eax, %esp
subl $12, %esp
pushl $.LC0
call puts
addl $16, %esp
movl $0, %eax
leave
ret
.size main, .-main
.ident "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"
.section.note.GNU-stack,"",@progbits
```

3) 汇编阶段

在该阶段,把编译阶段生成的.s文件转换成目标文件,在此使用选项-c就可看到汇编代码已转化为.o的二进制目标代码了,如下所示。

```
[root@localhost GCC]#gcc -c hello.s -o hello.o
```

4) 链接阶段

在链接阶段,所有的目标文件被安排在可执行程序中的恰当的位置,同时,该程序所调用的库函数也从各自所在的函数库中链到合适的地方。

重新查看这个小程序,发现在这个程序中并没有定义printf的函数实现,且在预编译所包含的stdio.h中也只有该函数的声明,而没有定义函数的实现。那么,是在哪里实现printf函数的呢?最后的答案是:系统把这些函数实现都做到名为libc.so.6的库文件中了,在没有特别指定时,GCC会到系统默认的搜索路径/usr/lib下进行查找,即链接到libc.so.6库函数中,这样就能实现函数printf了,这也就是链接的作用。

完成了链接后,GCC就可以生成可执行文件,如下所示。

```
[root@localhost GCC]#gcc hello.o -o hello
```

运行该可执行文件，其结果如下。

```
[root@localhost GCC]# ./hello
Hello! This is our embedded world!
```

3. GCC 的基本用法和选项

在使用 GCC 编译器的时候，必须给出一系列必要的调用参数和文件名称。GCC 编译器的调用参数有 100 多个，这里介绍其中最基本、最常用的参数。

GCC 一般用法是：

```
gcc [源程序名] [options] [执行文件名]
```

其中，options 为编译器所需要的参数。

- -c：仅编译，不链接成可执行文件，编译器只是由输入的.c 等源代码文件生成.o 为后缀的目标文件，通常用于编译不包含主程序的子程序文件。
- -o：编译并链接成可执行文件。如果不给出这个选项，GCC 就生成默认的可执行文件 a.out。
- -g：产生符号调试工具(GNU 的 gdb)所必要的符号信息，要想对源代码进行调试，就必须加入这个选项。
- -O：对程序进行优化编译、链接，采用这个选项，整个源代码会在编译、链接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、链接的速度就相应地要慢一些。
- -O2：比-O 更好的优化编译、链接，当然整个编译、链接过程会更慢。

例如，一个名为 test.c 的 C 语言源代码文件，要编译成一个可执行文件 test，最简单的办法就是：

```
#gcc  test.c  -o  test
```

这时，预编译、编译链接一次完成，将源程序编译成可执行文件。

若不指定编译后的可执行文件名，如：

```
#gcc  test.c
```

则生成一个系统预设的名为 a.out 的可执行文件。

对于稍为复杂的情况，比如有多个源代码文件、需要连接档案库或者有其他比较特别的要求，就要给定适当的调用选项参数。

4. GCC 使用函数库

1) 函数库

函数库可以看作是事先编写好的函数集合，它与主函数分离，从而增加程序开发的复用性。Linux 中的函数库可以分成 3 种类型：静态函数库、共享函数库和动态函数库。

静态函数库的代码在编译时就已经链接到所开发应用程序中，而共享函数库和动态函数库都只是在程序运行时才载入。由于共享函数库和动态函数库并没有在程序中包括库函数的内容，只是包含了对库函数的引用，因此，应用程序所占用的存储空间较小。

注意：共享函数库与动态函数库是有区别的。动态库使用的库函数不是在程序运行时开始载入，而是在程序中的语句需要使用该函数时才载入，并且可以在程序运行期间释放动态库所占用的内存，腾出空间供其他程序使用。

Linux系统的函数库都存放在/lib和/usr/lib目录下，通常库文件名由“lib＋库名＋后缀”组成。一般静态库的后缀名为.a，动态库的后缀名为由“.so＋版本号”组成。

例如，数学共享库的库名为libm.so.5，这里的标识符为m，版本号为5。

2）相关路径选项

通常库文件的路径不在系统默认的路径下，因此，GCC在编译时要指定库文件的位置，这里就需要用到相关路径选项：-I dirname，该选项将dirname所指定的目录加入程序头文件目录列表中，这时GCC就会到相应的位置查找对应的目录。

比如在/root/test下有两个文件。

- 文件hello1.c

```
/* hello1.c */
#include <my.h>
int main()
{
  printf("Hello!!\n");
  return 0;
}
```

- 文件my.h

```
/* my.h */
#include <stdio.h>
```

可在GCC命令行中加入-I选项。

```
[root@localhost test]# gcc hello1.c -I /root/test/ -o hello1
```

这样，GCC就能够执行出正确结果。

在include语句中，尖括号<>表示在标准路径中搜索头文件，双引号" "表示在本目录中搜索。故在上例中，可把hello1.c的#include <my.h>改为#include “my.h”，就不需要加上-I选项了。


视频讲解

4.2 “文件包含”处理

1. 头文件

在C语言中，需要利用头文件来定义结构、常量以及声明函数的原型。大多数C语言的头文件都存放在/usr/include及其子目录下，可以在这个目录很容易地见到stdio.h、stdlib.h等熟悉的面孔。

引用以上目录中的头文件在编译的时候无须加上路径，但如果程序中引用了其他路径的头文件，需要在编译的时候用-I参数。

2. “文件包含”处理

“文件包含”处理，意思是把另外一个源文件的内容包含到本程序中来。其作用是减少

编写程序的重复劳动，即把一些要重复使用的东西，编写到一个“头文件”(*.h)中，然后在程序中用#include 命令来实现“文件包含”的操作。

在进行较大规模程序设计时，“文件包含”处理也十分有用。为了适应模块化编程的需要，可以将组成 C 语言程序的各个功能函数分割到多个程序文件中，分别由不同人员负责编程，最后用#include 命令将它们嵌入一个总的程序文件中去。

在设计嵌入式系统应用程序时，需要用头文件把目标板生产商提供的各种库函数及数据类型、函数声明集中到一处。这可以保证数据结构定义的一致性，以便程序的每一部分都能以同样的方式看待一切事情。

【例 4-1】 计算∑n=1+2+3+…+100 求和运算的程序示例。

1) 直接把所有运算代码都编写在 main()函数中

```
#include <stdio.h>
int main()
  {
    int x = 100,s = 0,i = 1;
    while(i <= x)
      {
        s = s + i;
        i++ ;
    }
    printf(" sum =  %d\n",s);
    return 0;
}
```

2) 把加法运算部分编写成一个函数

为了让加法部分能重复使用，将加法部分写成一个函数 int mysum(int n)，再在主函数中调用它。

```
#include <stdio.h>
int mysum(int n);
int main()
{
    int x = 100;
    int s = 0;
    s = mysum(x);
    printf(" sum =  %d\n",s);
    return 0;
 }
 int mysum(int n)
 {
   int i = 1,ss = 0;
   while(i <= n){
       ss = ss + i;
       i++ ;
     }
   return (ss);
 }
```

注意：上述程序中的第2行语句

```
int mysum(int n);
```

是必不可少的。由于mysum(int n)函数的定义是从第11行语句开始，而调用mysum(int n)函数的语句在第7行，因此，要在调用之前声明这个函数。

3) 分割成多个独立功能的函数

下面进一步将程序中具有独立功能的mysum()函数分割出来。该程序可分割为下列3个Linux C程序：mysum.h、mysum.c和ex_sum.c。

- 程序mysum.h

```
1  /*   mysum.h   */
2  int mysum(int n);
```

- 程序mysum.c

```
1. /* mysum.c */
2. int mysum(int n)
3.   {
4.     int i=1,ss=0;
5.     while(i<=n){
6.         ss=ss+i;
7.         i++;
8.         }
9.     return (ss);
10. }
```

- 主程序ex_sum.c

```
1.  /*   ex_sum.c   */
2.  #include<stdio.h>
3.  #include "mysum.h"
4.  int main()
5.   {
6.     int x=100;
7.     int s=0;
8.     s=mysum(x);
9.     printf("sum= %d\n",s);
10.    return 0;
11.  }
```

在Linux环境下，执行编译程序命令。

```
gcc  ex_sum.c  mysum.c  -o  sum
```

此命令将ex_sum.c和mysum.c编译成一个在Linux环境下的可执行文件sum。

在Linux环境下运行可执行文件sum。

```
./sum
```

结果如下。

```
sum=5050
```

4.3　make 命令和 Makefile 工程管理


视频讲解

4.3.1　认识 make

首先通过前面编译求和程序 sum 的例子来认识 make 命令和 Makefile 文件的作用及用法。该程序涉及 mysum.h、mysum.c 和 ex_sum.c 等 3 个文件，编写一个 Makefile 文件如下。

```
sum:  ex_sum.o  mysum.o
    gcc  ex_sum.o  mysum.o  -o  sum
ex_sum.o: ex_sum.c
    gcc  -c  ex_sum.c
mysum.o:  mysum.c  mysum.h
    gcc  -c  mysum.c
```

注意：命令 gcc　ex_sum.o　mysum.o　-o　sum 前面不是空格，而是按下 Tab 键的制表符号位。

将其保存为 Makefile，文件名没有后缀。然后，在 Linux 环境下执行 make，其运行结果如下。

```
[root@localhost sum]# make
gcc  -c  ex_sum.c
gcc  -c  mysum.c
gcc  ex_sum.o  mysum.o  -o  sum
```

将 ex_sum.c 和 mysum.c 编译成在 Linux 环境下的可执行文件 sum。

从上面的例子可以看出，Makefile 是 make 读入的配置文件。在一个 Makefile 中通常包含如下内容。

- 需要由 make 工具创建的目标体(target)，通常是目标文件或可执行文件。
- 要创建的目标体所依赖的文件(dependency_file)。
- 创建每个目标体时需要运行的命令(command)。

Makefile 的格式为：

```
target: dependency_files
        command
```

注意：在 Makefile 中的 command 前必须有 Tab 符，否则在运行 make 命令时会出错。

上面的例子很简单，完全没有必要使用 Makefile，而直接用编译命令就可完成编译任务。下面再看一个稍微复杂一点的例子。

【例 4-2】　应用 Makefile 文件编译程序的示例。

假设有下面这样的一个程序，该程序涉及 mytool1.h、mytool2.h、mytool1.c、mytool2.c，其源代码如下。

1）主程序 main.c

```
1.  /* main.c */
2.  #include "mytool1.h"
```

```
3.  #include "mytool2.h"
4.  int main(int argc,char *argv)
5.  {
6.      mytool1_print("hello");
7.      mytool2_print("hello");
8.  }
```

2) 头文件 mytool1.h 源程序

```
1.  /* mytool1.h */
2.  #ifndef _MYTOOL_1_H
3.  #define _MYTOOL_1_H
4.  void mytool1_print(char *print_str);
5.  #endif
```

3) 文件 mytool1.c 源程序

```
1.  /* mytool1.c */
2.  #include  "mytool1.h"
3.  void mytool1_print(char *print_str)
4.  {
5.      printf("This is mytool1 print: %s\n ", print_str);
6.  }
```

4) 头文件 mytool2.h 源程序

```
1.  /* mytool2.h */
2.  #ifndef _MYTOOL_2_H
3.  #define _MYTOOL_2_H
4.  void mytool2_print(char *print_str);
5.  #endif
```

5) 文件 mytool2.c 源程序

```
1.  /* mytool2.c */
2.  #include "mytool2.h"
3.  void mytool2_print(char *print_str)
4.  {
5.      printf("This is mytool2 print: %s\n ",print_str);
6.  }
```

我们可以这样来编译。

```
gcc  -c  main.c
gcc  -c  mytool1.c
gcc  -c  mytool2.c
gcc  -o  main main.o  mytool1.o  mytool2.o
```

这样的话,就可以产生 Linux 下的执行文件 main,而且也不太麻烦。但是,如果有一天我们修改了其中一个文件(如 mytool1.c),那么难道还要重新输入上面的命令? 如果把事情想得更复杂一点,假若程序有几百个源程序,难道也要编译器重新一个一个地去编译? 所以,此时就需要一个工程管理器能够自动识别更新了的文件代码,同时又不需要重复输入冗

长的命令行，这样，make 工程管理器就应运而生了。

实际上，make 工程管理器也就是个“自动编译管理器”，这里的“自动”指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入 Makefile 文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有 Linux 下的项目编程均会涉及它，希望读者能够认真学习本节内容。

对于上面的那个程序来说，可以编写一个如下的 Makefile 文件。

```
1.  main: main.o  mytool1.o  mytool2.o
2.      gcc  -o  main  main.o  mytool1.o  mytool2.o
3.  main.o: main.c
4.      gcc  -c  main.c
5.  mytool1.o: mytool1.c mytool1.h
6.      gcc  -c  mytool1.c
7.  mytool2.o: mytool2.c mytool2.h
8.      gcc  -c  mytool2.c
```

这里，每个 gcc 前都是 Tab 制表符，不是空格。

接着就可以使用 make 了。make 会自动读入 Makefile(也可以是 makefile)并执行对应目标体的编译命令语句，同时找到相应的依赖文件，如下所示。

```
[root@localhost maketest]# make
gcc  -c  main.c
gcc  -c  mytool1.c
gcc  -c  mytool2.c
gcc  -o  main main.o  mytool1.o  mytool2.o
[root@localhost maketest]# ls
Makefile main main.c main.o mytool1.c  mytool1.h  mytool1.o  mytool2.c
mytool2.h  mytool2.o
```

可以看到，make 执行了 Makefile 中一系列的命令语句，并生成了 main.o 目标体，最后生成了 Linux 下的执行文件 main。可以运行这个执行文件，其结果如下。

```
[root@localhost maketest]# ./main
This is mytool1 print: hello
This is mytool2 print: hello
```

如果再次运行 make，这时，make 会自动检查相关文件的时间戳。

首先，在检查 main、main.o、mytool1.o 和 mytool2.o 这 4 个文件的时间戳之前，它会向下查找那些把 main.o、mytool1.o 或 mytool2.o 作为目标文件的时间戳。如果这些文件中任何一个的时间戳比它们新，则用 gcc 命令将此文件重新编译。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作。这就是 make 工作的基本流程。

4.3.2　Makefile 变量

为了进一步简化编辑和维护 Makefile，make 允许在 Makefile 中创建和使用变量。变量是在 Makefile 中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令以及 Makefile 文件中其他部

分。在 Makefile 中的变量定义常用的有两种方式：一种是递归展开方式；另一种是简单方式。

递归展开方式定义的变量是在引用该变量时进行替换的，即如果该变量包含了对其他变量的引用，则在引用该变量时一次性将内嵌的变量全部展开。

简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除变量的嵌套引用。

递归展开方式的定义格式为：VAR＝var

简单扩展方式的定义格式为：VAR：＝var

make 中的变量无论采用哪种方式定义，使用时格式均为：$(VAR)。

下面给出例 4-2 中用变量替换修改后的 Makefile，这里用 OBJS 代替 main.o、mytool1.o 和 mytool2.o，用 CC 代替 gcc。这样在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了 Makefile 维护的工作量。

经变量替换后的 Makefile 如下所示。

```
1.  OBJS = main.o mytool1.o  mytool2.o
2.  CC = gcc
3.  main: $(OBJS)
4.      $(CC)  $(OBJS)  -o  main
5.  main.o: main.c
6.      $(CC)  -c  main.c
7.  mytool1.o: mytool1.c  mytool1.h
8.      $(CC)  -c  mytool1.c
9.  mytool2.o: mytool2.c  mytool2.h
10.     $(CC)  -c  mytool2.c
```

可以看到，此处变量是以递归展开方式定义的。

由于常见的 GCC 编译语句中通常包含了目标文件和依赖文件，而这些文件在 Makefile 文件中目标体的一行已经有所体现，因此，为了进一步简化 Makefile 的编写，引入了自动变量的概念。自动变量通常可以代表编译语句中出现目标文件和依赖文件等，并且具有本地含义(即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件)。表 4.2 列出了 Makefile 中常见的自动变量。

表 4.2　Makefile 中常见的自动变量

命令格式	含　义
$*	不包含扩展名的目标文件名称
$＋	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$<	第一个依赖文件的名称
$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
$@	目标文件的完整名称
$^	所有不重复的依赖文件，以空格分开
$%	如果目标是归档成员，则该变量表示目标的归档成员名称

自动变量的书写比较难记，但是熟练了之后会非常方便，请读者结合下例中自动变量改写的 Makefile 进行记忆。

```
1.  OBJS = main.o mytool1.o mytool2.o
2.  CC = gcc
3.  main: $(OBJS)
4.      $(CC) $^ -o $@
5.  main.o: main.c
6.      $(CC) -c $< -o $@
7.  mytool1.o: mytool1.c mytool1.h
8.      $(CC) -c $< -o $@
9.  mytool2.o: mytool2.c mytool2.h
10.     $(CC) -c $< -o $@
```

另外，在 Makefile 中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

4.3.3　Makefile 规则

Makefile 的规则是 make 进行处理的依据，它包括了目标体、依赖文件及其之间的命令语句。一般的，Makefile 中的一条语句就是一个规则。在上面的例子中，都显式地指出了 Makefile 中的规则关系，如 $(CC) -c $< -o $@。为了简化 Makefile 的编写，make 定义了两种类型的规则：隐式规则和模式规则，下面就分别对其进行介绍。

1. 隐式规则

在使用 Makefile 时，有些语句经常使用，而且使用频率非常高，隐式规则能够告诉 make 使用默认的方式来完成编译任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。make 会自动按隐式规则来确定如何生成目标文件。例如，编译例 4-2 程序的 Makefile 就可以写成：

```
1.  OBJS = main.o  mytool1.o  mytool2.o
2.  CC = gcc
3.  main: $(OBJS)
4.      $(CC)  $^  -o  $@
```

为什么可以省略后 6 句呢？因为 make 的隐式规则指出：所有.o 文件都可自动由.c 文件使用命令 $(CC)　-c　<file.c>　-o　<file.o>生成。这样 main.o、mytool1.o 和 mytool2.o 就会分别调用 $(CC)　-c　$<　-o　$@来生成。

2. 模式规则

由于 make 的隐式规则只能对默认的变量进行编译操作，对于用户自定义的变量，则不能识别操作。为了让隐式规则适合更普遍的情况，模式规则规定，在定义目标文件时需要用%字符。%表示一个或多个任意字符，与文件名匹配。在依赖文件中同样可以使用%，只是依赖文件中%的取值取决于其目标文件。

例如，%.c 表示以.c 结尾的文件名(文件名的长度至少为 3)，而 s.%.c 则表示以 s.开头、以.c 结尾的文件名(文件名的长度至少为 5)。

对于上面的 Makefile 文件,若使用模式规则,其代码如下。

```
1.  OBJS = main.o  mytool1.o  mytool2.o
2.  CC = gcc
3.  main: $(OBJS)
4.      $(CC) $^  -o  $@
5.   %.o: %.c
6.      $(CC)  -c  $<  -o  $@
```

4.3.4 make 命令的使用

使用 make 命令非常简单,要运行当前目录下的 Makefile,只需直接输入 make,则可建立在 Makefile 中所指定的目标文件。此外,make 还有丰富的命令行选项,可以完成各种不同的功能。表 4.3 列出了常用的 make 命令行选项。

表 4.3 make 的命令行选项

命令格式	含义
-C <dir>	读入指定目录下的 Makefile
-f <file>	读入当前目录下的<file>文件作为 Makefile
-i	忽略所有的命令执行错误
-I <dir>	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令,但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录,则打印当前目录名

4.4 嵌入式 Linux 汇编语言程序设计

汇编语言的优点是执行速度快,可以直接对硬件进行操作。作为最基本的编程语言之一,汇编语言虽然应用的范围不算很广泛,但重要性却毋庸置疑,因为它能够完成许多其他语言所无法完成的功能。在嵌入式系统的设计应用中,虽然绝大部分代码是用 C 语言编写的,但仍然不可避免地在某些关键地方需要使用汇编语言的代码。

大多数情况下,嵌入式系统的设计人员不需要使用汇编语言,因为即便是硬件驱动这样的底层程序,在嵌入式 Linux 系统中也可以完全用 C 语言来实现。但是,在移植 Linux 到某一特定的嵌入式硬件环境下时,则需要汇编程序。

嵌入式 Linux 系统下,用汇编语言编写程序有以下两种不同的形式。

1) 完全汇编代码

完全汇编代码,即整个程序全部用汇编语言编写。尽管是完全的汇编代码,但嵌入式 Linux 系统下的汇编工具也吸收了 C 语言的长处,使得设计人员可以使用 #include、#ifdef 等预处理指令,并能通过宏定义来简化代码。

2) 内嵌汇编代码

内嵌汇编代码,即可以把汇编代码片段嵌入 C 语言程序中。

对于初学者来说，用汇编语言指令编写程序是一件比较困难的事情，需要专门学习该课程知识。这里仅对嵌入式 Linux 汇编语言格式与 DOS/Windows 汇编语言格式的不同之处做一些简单介绍。

4.4.1　嵌入式 Linux 汇编语言格式

1. 嵌入式 Linux 汇编语言程序结构

在嵌入式 Linux 汇编语言程序中，程序是以程序段(Section)的形式呈现的。程序段是具有特定名称的相对独立的指令或数据序列。

程序段分为代码段(Code Section)和数据段(Data Section)两种类型。代码段的主要内容为执行代码；数据段则存放代码段运行时需要用到的数据。

一个汇编语言程序至少要有一个代码段。

2. 嵌入式 Linux 汇编语言的语法格式

嵌入式 Linux 汇编语言的语法格式和 DOS/Windows 下的汇编语言语法格式有较大的差异。DOS/Windows 下的汇编语言代码都是 Intel 格式；嵌入式 Linux 的汇编语言代码采用的是 AT&T 格式，两者在语法格式上有着很大的不同，主要区别如下。

(1) 在 AT&T 汇编格式中，寄存器名要加上%作为前缀；在 Intel 汇编格式中，寄存器名不需要加前缀。

例如：

```
AT&T 格式:    pushl   %eax
Intel 格式:   push    eax
```

(2) 在 AT&T 汇编格式中，用 $ 前缀表示一个立即操作数；在 Intel 汇编格式中，立即数的表示不用带任何前缀。

例如：

```
AT&T 格式:    pushl   $1
Intel 格式:   push    1
```

(3) AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 AT&T 汇编格式中，目标操作数在源操作数的右边；在 Intel 汇编格式中，目标操作数在源操作数的左边。

例如：

```
AT&T 格式:    addl    $1,   %eax
Intel 格式:   add     eax,    1
```

(4) 在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀 b、w、l 分别表示操作数为字节(byte，8 比特)、字(word，16 比特)和长字(long，32 比特)；在 Intel 汇编格式中，操作数的字长是用 byte ptr 和 word ptr 等前缀来表示的。

例如：

```
AT&T 格式:    movb   val,   %al
Intel 格式:   mov   al,   byte ptr val
```

(5) 在 AT&T 汇编格式中,绝对转移和调用指令(jump/call)的操作数前要加上 * 作为前缀,在 Intel 格式中则不需要。

远程转移指令和远程子调用指令的操作码,在 AT&T 汇编格式中为 ljump 和 lcall;在 Intel 汇编格式中则为 jmp far 和 call far,即:

```
AT&T 格式:   ljump   $section,   $offset
             lcall   $section,   $offset
Intel 格式:  jmp   far   section:offset
             call   far   section:offset
```

与之相应的远程返回指令如下。

```
AT&T 格式:   lret   $stack_adjust
Intel 格式:  ret   far   stack_adjust
```

(6) 在 AT&T 汇编格式中,内存操作数的寻址方式是 section:disp(base, index, scale);在 Intel 汇编格式中,内存操作数的寻址方式为:section:[base + index * scale + disp]。

表 4.4 是一些内存操作数的两种格式示例。

表 4.4 内存操作数的两种格式示例

汇编格式	语法格式

AT&T 格式	movl −4(%ebp), %eax
	movl array(, %eax, 4), %eax
	movw array(%ebx, %eax, 4), %cx
	movb $4, %fs:(%eax)

Intel 格式	mov eax, [ebp − 4] mov eax, [eax * 4 + array] mov cx, [ebx + 4 * eax + array] mov fs:eax, 4


视频讲解

4.4.2 嵌入式 Linux 汇编程序示例

【例 4-3】 编写一个最简单的 AT&T 格式的汇编程序。

用编辑工具编写如下汇编源程序,并将其保存为 hello.s。

```
#hello.s
.data                                   # 数据段声明
        msg : .string "Hello, world!\n" # 要输出的字符串
        len = . - msg                   # 字串长度
.text                                   # 代码段声明
.global _start                          # 指定入口函数

_start:                                 # 在屏幕上显示一个字符串
      movl   $len,   %edx               # 参数三:字符串长度
      movl   $msg,   %ecx               # 参数二:要显示的字符串
      movl   $1,   %ebx                 # 参数一:文件描述符(stdout)
      movl   $4,   %eax                 # 系统调用号(sys_write)
      int    $0x80                      # 调用内核功能
```

```
                                    # 退出程序
    movl   $0,   %ebx               # 参数一：退出代码
    movl   $1,   %eax               # 系统调用号(sys_exit)
    int    $0x80                    # 调用内核功能
```

在本程序中，调用 Linux 内核提供的系统调用 sys_write 显示一个字符串，再应用系统调用 sys_exit 退出程序。在 Linux 内核源文件 include/asm-i386/unistd.h 中，可以找到所有系统调用的定义。

【例 4-4】 用 Intel 格式编写一个与例 4-3 相同的简单汇编程序。

```
; hello.asm
section .data                       ; 数据段声明
    msg  db  "Hello, world!", 0xA   ; 要输出的字符串
    len  equ  $ - msg               ; 字串长度
section .text                       ; 代码段声明
global _start                       ; 指定入口函数
_start:                             ; 在屏幕上显示一个字符串
    mov  edx,  len                  ; 参数三：字符串长度
    mov  ecx,  msg                  ; 参数二：要显示的字符串
    mov  ebx,  1                    ; 参数一：文件描述符(stdout)
    mov  eax,  4                    ; 系统调用号(sys_write)
    int  0x80                       ; 调用内核功能
                                    ; 退出程序
    mov  ebx,  0                    ; 参数一：退出代码
    mov  eax,  1                    ; 系统调用号(sys_exit)
    int  0x80                       ; 调用内核功能
```

4.4.3　编译嵌入式 Linux 汇编程序

下面以编译汇编语言源程序 hello.s 为例，说明编译嵌入式 Linux 汇编程序的方法。

1. 汇编命令

汇编的作用是将用汇编语言编写的源程序转换成二进制形式的目标代码。对于使用标准的 AT&T 语法格式编写的汇编程序，可以使用以下汇编命令。

```
[root@localhost asm] # as  -o  hello.o  hello.s
```

或应用交叉汇编命令。

```
[root@localhost asm] # arm-linux-as  -o  hello.o  hello.s
```

对于 Intel 语法格式编写的汇编程序，需要使用 NASM 汇编器的汇编命令进行汇编操作。

```
[root@localhost asm] # nasm  -f  elf  hello.s
```

2. 链接命令

由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码。链接器通常用来将多个目标代码连接成一个可执行代码，这样可以先将整个程序分成几个模块来单独开发，然后才将它们组合（链接）成一个应用程序。Linux

使用 ld 作为标准的链接程序,经链接后的程序就成为可执行程序。

```
[root@localhost asm] # ld  -s  -o  hello  hello.o
```

或应用交叉链接命令:

```
[root@localhost asm] # arm-linux-ld  -s  -o  hello  hello.o
```

3. 运行程序

```
[root@localhost asm] #  ./hello
Hello,  world!
```


视频讲解

4.5 嵌入式 Linux shell 编程

shell 是用户与 Linux 系统间的接口程序,它允许用户向操作系统输入需要执行的命令。当用户在终端窗口输入命令,系统会利用解释器解释这些命令从而执行相应的操作。完成这一解释功能的机制就是 shell。shell 编程就是把 shell 命令编写成可执行的脚本文件。

4.5.1 shell 的语法基础

1. shell 脚本文件

1) shell 脚本文件结构

shell 脚本文件结构格式是固定的。首先看一个简单 shell 脚本文件的示例。

```
#!  /bin/bash
echo  "Hello  World!"
```

将文件保存为 hello.sh。

shell 脚本文件的第一行必须以符号#!开头。例如本例中的

```
#!  /bin/bash
```

符号#!用来告诉系统它后面的参数是用来执行该文件的解释器。在这个例子中,要使用/bin/bash 解释器来解释并执行程序。

2) 添加 shell 脚本文件可执行权限

添加 shell 脚本文件可执行权限的命令如下。

```
chmod  +x  [文件名]
```

例如:

```
chmod  +x  hello.sh
```

3) 执行已经添加可执行权限的 shell 脚本文件

```
[root@localhost shell] # ./hello.sh
Hello  World!
```

也可以直接使用 sh 命令来执行 shell 脚本文件。

```
[root@localhost shell] # sh  hello.sh
Hello  World !
```

4）shell 脚本文件的注释语句

在 shell 脚本文件中，以 # 开头的语句表示注释。但脚本第一行用 #! 开头的语句不是注释，而是说明 shell 脚本文件的解释器。

2. shell 的变量及配置文件

与其他编程语言一样，shell 也允许将数值存放到变量中。shell 脚本文件的变量共有 3 种：用户变量、环境变量和系统变量。

1）用户变量

• 变量赋值

一般 shell 脚本文件的变量都是用户变量。给变量赋值时，要使用赋值符号（=）为变量赋值。例如，给变量 a 赋值 Hello World：

```
a = "hello world"  (赋值号 = 的两侧不允许有空格)
```

• 获取变量的值

获取变量的值时，要在变量前面添加 $ 符号。

例如：

```
echo  $ a
```

用户变量的使用如下。

```
[root@localhost shell] # a = "hello world"
[root@localhost shell] # echo  $ a
 hello world
```

2）环境变量

由关键字 export 说明的变量叫作环境变量。

例如：

```
[root@localhost shell] # export  abc = /mnt/shell
[root@localhost shell] # echo  $ abc
 /mnt/shell
```

3）系统变量

shell 脚本文件中用到系统变量的地方不多，主要在用于表示参数时使用，这里不做介绍，仅列出一些常用系统变量。

• $0：当前程序的名称
• $n：当前程序的第 n 个参数，n=1,2,…,9
• $ *：当前程序的所有参数（不包括程序本身）
• $ #：当前程序的参数个数（不包括程序本身）
• $ $：当前程序的 PID
• $!：执行上一个指令的 PID
• $?：执行上一个指令的返回值

4.5.2 shell 的流程控制语句

shell 脚本文件中用条件语句和循环语句来控制程序的执行流程。

1. 条件语句

条件语句从 if 开始,到 fi 结束。满足条件表达式则执行条件语句中的语句块;若不满足条件,则跳过条件语句,继续执行后续语句。

条件语句的格式如下。

```
if  ((  条件表达式  ))
then
      #语句块
fi
```

注意:条件表达式要用双括号括起来。

2. 循环语句

循环语句的格式如下。

```
for  ((循环变量=初值;  循环条件表达式;  循环变量增量))
do
      #循环体语句块
done
```

注意:for 循环的条件要用双括号括起来。

4.5.3 shell 编程示例

【例 4-5】 编写显示 20 以内能被 3 整除的数的 shell 脚本程序。

应用 vi 编辑工具编写如下 shell 脚本。

```
1.  #! /bin/bash
2.  for((i=1; i<20; i++))
3.  do
4.    if(( i%3 == 0))
5.    then
6.       echo $i
7.    fi
8.  done
```

将其保存为 test1.sh,然后再编译并执行程序。

```
[root@localhost shell] # chmod  +x  test1.sh
[root@localhost shell] # ./test1.sh
3
6
9
12
15
18
```

在本程序中,第 2 行~第 8 行为循环语句,在其中第 4 行~第 7 行嵌套了一个条件语句。

【例 4-6】　编写一个“按任意键继续……”的 shell 脚本程序。

应用 vi 编辑工具编写如下 shell 脚本。

```
1.  #! /bin/bash
2.  get_char()
3.  {
4.    stty  - echo
5.    stty  raw
6.    dd  if = /dev/tty  bs = 1  count = 1  2 >/dev/null
7.    stty  - raw
8.    stty  echo
9.  }
10. echo  "Press  any  key  to  continue ....."
11. get_char
```

/dev/null 表示空设备

块大小 bs 为 1,数目也为 1

将其保存为 test2.sh。然后再编译并执行程序。

```
[root@localhost shell] # chmod  + x  test2.sh
[root@localhost shell] # ./test2.sh
Press  any  key  to  continue .....
```

在本程序中的第 2 行～第 9 行为 shell 脚本程序的函数,其函数名为 get_char。在第 11 行调用 get_char 函数。

本程序的核心语句是第 6 行,dd 命令是用指定大小的块复制一个文件,并在复制的同时进行指定的转换。在本语句中的 bs=1,即指定数据块的大小为 1 字节。/dev/null 表示空设备,语句中的 2 >/dev/null 表示输出到空设备中,即没有任何输出。

dd 命令的参数使用格式说明如下。

- if = 文件名或设备：输入的文件或设备。
- bs = bytes：一次读入 bytes 字节,即指定一个块,其大小为 bytes 字节。
- count = blocks：仅复制 blocks 个块,块大小等于 bs 指定的字节数。
- 2 >：表示标准错误信息输出。


视频讲解

4.6　位　运　算

4.6.1　位运算符

在嵌入式 Linux 的程序设计中经常要求在位(bit)一级进行运算,即位运算。按位操作运算的运算符称为位运算符,C 语言提供了 6 种位运算符：&(按位与)、|(按位或)、^(按位异或)、～(取反)、<<(左移)和>>(右移)。

1. 按位与运算

按位与运算符 & 是双目运算符。其功能是参与运算的两数各对应的二进制位相与,只有对应的两个二进制位均为 1 时,结果位才为 1,否则为 0。参与运算的数以补码方式出现。

例如：9 & 5 可写成算式如下。

```
  00001001  (9的二进制补码)
& 00000101  (5的二进制补码)
----------
  00000001  (1的二进制补码)
```

可见 9 & 5 = 1。

按位与运算通常用来对某些位清0或保留某些位。例如,把a的高八位清0,保留低八位,可作a & 0x00ff运算(0x00ff的二进制数为0000000011111111)。

```
main(){
    int a=9,b=5,c;
    c=a&b;
    printf("a=%d/nb=%d/nc=%d/n",a,b,c);
  }
```

2. 按位或运算

按位或运算符|是双目运算符。其功能是参与运算的两数各对应的二进制位相或。只要对应的两个二进制位有一个为1时,结果位就为1。参与运算的两个数均以补码出现。

例如:9|5可写算式如下。

```
  00001001
| 00000101
----------
  00001101(十进制为13)
```

可见9|5=13。

```
main(){
    int a=9,b=5,c;
    c=a|b;
    printf("a=%d/nb=%d/nc=%d/n",a,b,c);
  }
```

3. 按位异或运算

按位异或运算符^是双目运算符。其功能是参与运算的两数各对应的二进制位相异或,当两对应的二进制位相异时,结果为1。参与运算数仍以补码出现。

例如:9^5可写成算式如下。

```
  00001001
^ 00000101
----------
  00001100(十进制为12)
```

```
main(){
    int a=9;
    a=a^5;
    printf("a=%d/n",a);
  }
```

4. 求反运算

求反运算符~为单目运算符,具有右结合性。其功能是对参与运算的数的各二进制位按位求反。

例如:~9的运算为~(0000000000001001),结果为1111111111110110。

再例如，对于一个整数 x，如果要把它的每个位都置 1，那么可以写成：

```
x = ～0 ;   /* 每位都是 0,取反后就是全为 1 了 */
```

这样做的好处是，不管这个整数 x 是多少位的，编译器都会自动生成合适的数。

5. 左移运算

左移运算符<<是双目运算符。其功能是把<<左边的运算数的各二进制位全部左移若干位，由<<右边的数指定移动的位数，高位丢弃，低位补 0。左移 1 位运算如图 4.1 所示。

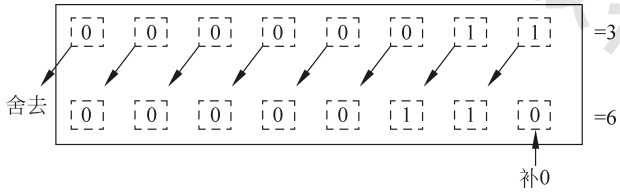

图 4.1　左移 1 位运算

例如，a<<4 指把 a 的各二进制位向左移动 4 位。如 a＝00000011(十进制 3)，左移 4 位后为 00110000(十进制 48)，如图 4.2 所示。

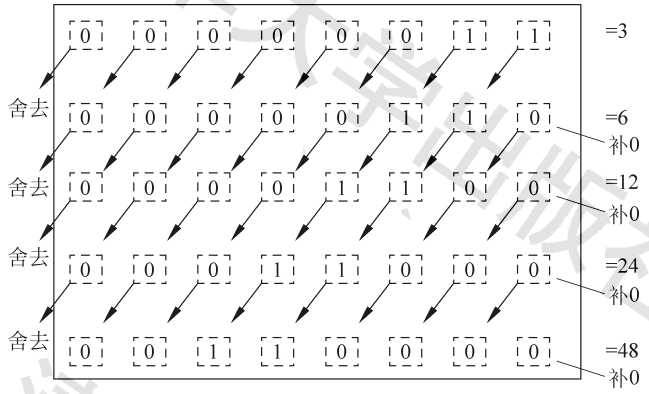

图 4.2　a<<4 的运算过程

6. 右移运算

右移运算符>>是双目运算符。其功能是把>>左边的运算数的各二进制位全部右移若干位，由>>右边的数指定移动的位数。

例如：设 a＝15，a>>2 表示把 00001111 右移为 00000011(十进制 3)。

应该说明的是，对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位补 0，而为负数时，符号位为 1，最高位是补 0 或是补 1 取决于编译系统的规定。

```
main(){
  unsigned a,b;
  printf("input a number: ");
  scanf("%d",&a);
  b = a >> 5;
  b = b&15;
```

```
    printf("a = %d/tb = %d/n",a,b);
}
```

请再看一例：

```
main(){
      char a = 'a',b = 'b';
      int p,c,d;
      p = a;
      p = (p << 8)|b;
      d = p&0xff;
      c = (p&0xff00)>> 8;
      printf("a = %d/nb = %d/nc = %d/nd = %d/n",a,b,c,d);
    } '
```


视频讲解

4.6.2 位表达式

将位运算符连接起来所构成的表达式称为位表达式。在这些位运算符中，其优先级依次为：～(取反运算符)、<<或>>(左移或右移)、&(按位与)或|(按位或)或^(按位异或)。

在嵌入式 Linux 程序设计中，进行赋值运算时经常会用到复合赋值操作符。

例如：

a << = 5 就等价于 a= a << 5

再如：

```
GPDR & = ～ 0xff;
```

将其展开，就成为：

```
GPDR = GPDR & (～0xff);
```

接下来的步骤就简单了：

```
GPDR = GPDR & 0x00;
```

完成了对 GPDR 的清零。

一个常用的操作是用 & 来获取某个或者某些位。

例如，获取整数 x 中的低 4 位可以写成：

```
x &= 0x0F;
x = x & 0x0F;
```

也可以用|、&、<<、>>等配合来设置和清除某位或者某些位。

例如：

(1) x &= 0x1；

即：

```
x = x & 0x1; /*  保留 x 的最后一位不变,其余全为 0  */
```

(2) x &= (0x1 << 5)；

即：

```
x = x & (0x1 << 5); /*  清除 x 的低 5 位,即 x 的后 5 位均为 0  */
```

(3) x |= 0x1;

即:

```
x = x | 0x1; /*  将 x 的最后一位(即第 0 位)设置为 1  */
```

(4) x |= (0x1 << 6);

即:

```
x = x | (0x1 << 6); /*  将 x 的第 6 位设置为 1,x 的其余位不予考虑  */
```

4.6.3　寄存器设置中的位运算应用示例


视频讲解

1. 对寄存器的某位进行赋值的方法

(1) 仅对寄存器 GPIOx 的第 n 位赋 1 值,其余值不变。

```
GPIOx | = (1 << n);
```

即:

```
GPIOx = GPIOx | (1 << n); /*  将 GPIOx 的第 n 位设置为 1  */
```

(2) 仅对寄存器 GPIOx 的第 n 位赋 0 值,其余值不变。

```
GPIOx &= ~(1 << n);
```

即:

```
GPIOx = GPIOx & (~(1 << n)); /*  GPIOx 与第 n 位为 0 的数进行 & 运算  */
```

2. 寄存器设置的应用示例

【例 4-7】 设在 Cortex A8 微处理器 GPIO 端口的 GPC0[3]、GPC0[4]引脚各连接一个 LED 发光二极管,如图 4.3 所示。编写程序,通过对控制寄存器 GPC0CON 和数据寄存器 GPC0DAT 进行赋值操作,使 LED 发光二极管点亮或熄灭。

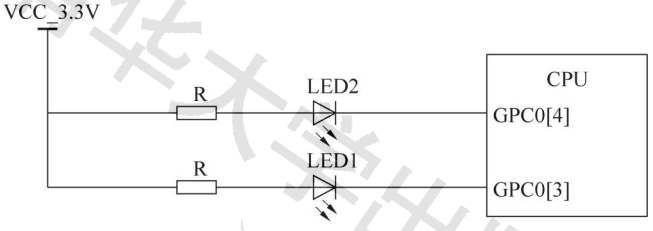

图 4.3　控制 LED 发光二极管点亮或熄灭

(1) 将寄存器 GPC0CON 的地址定义为指针并赋值。

通过查看寄存器地址表(表 2.3)可以知道,GPC0CON 的地址为 0xE020_0060,通过例 2-1 已经知道,要控制 LED 发光二极管点亮或熄灭,需要把 GPC0[3]、GPC0[4]引脚设置为输出模式。那么,怎样为 GPC0CON 寄存器赋值呢?

第 1 步:把 GPC0CON 寄存器的地址定义为 unsigned int 类型的指针。

```
unsigned int * GPC0CON = (unsigned int * )0xE0200060;
```

第2步：添加volatile,防止编译优化。

为了防止在程序的编译过程中,编译器对数据进行优化而使数据发生更改,对指针变量添加volatile进行修饰。这样处理后,编译器不再对该变量进行优化处理。

```
volatile unsigned int * GPC0CON = * (volatile unsigned int * )0xE0200060
```

(2) 设置寄存器GPC0CON为输出模式。

① 首先清除GPC0CON[4]、GPC0CON[3]寄存器的内容。

由于~0xff000的二进制数为：0000 0000 1111 1111 1111,因此,不论GPC0CON[4]和GPC0CON[3]中的值为多少(GPC0CON[4]、GPC0CON[3]为二进制数中的第13~第20位数值),经位运算：

```
* GPC0CON &= ~0xff000;
```

之后,第13~第20位数(GPC0CON[4]、GPC0CON[3])的值一定是0。这样就实现了清除GPC0CON[4]、GPC0CON[3]寄存器中数据的目的。

② 再设置GPC0CON[4]、GPC0CON[3]寄存器为输出模式。

由于0x11000的二进制数为：0001 0001 0000 0000 0000,经位运算：

```
* GPC0CON |= 0x11000;
```

之后,第13位及第17位数(GPC0CON[4]和GPC0CON[3]的个位)的值一定是1,从而将GPC0CON[4]、GPC0CON[3]寄存器设置为输出模式。

(3) 定义GPC0DAT数据寄存器的地址为指针。

通过查看寄存器地址表(表2.3)可知,GPC0DAT的地址为0xE0200064,则定义GPC0数据寄存器指针：

```
volatile unsigned int * GPC0DAT = (volatile unsigned int * ) 0xE0200064;
```

(4) 设置GPC0DAT数据寄存器的值,控制LED发光二极管点亮或熄灭。

GPC0DAT寄存器每一位对应一个GPIO端口引脚,当该寄存器的某位设置为1时,则对应引脚输出高电平,该寄存器的某位设置为0时,对应引脚输出低电平。

GPC0DAT寄存器的初始值为0x00。通过位运算来设置寄存器的值,控制LED发光二极管的点亮及熄灭。

```
while (1)
  {
    GPC0DAT &= ~(3 << 3);
    GPC0DAT |= i << 3;
    i++ ;
    if (i == 4) {i = 0; } //控制循环4次,再重复
  }
```

控制LED发光二极管的点亮及熄灭的循环过程如图4.4所示。

完整程序代码如下。

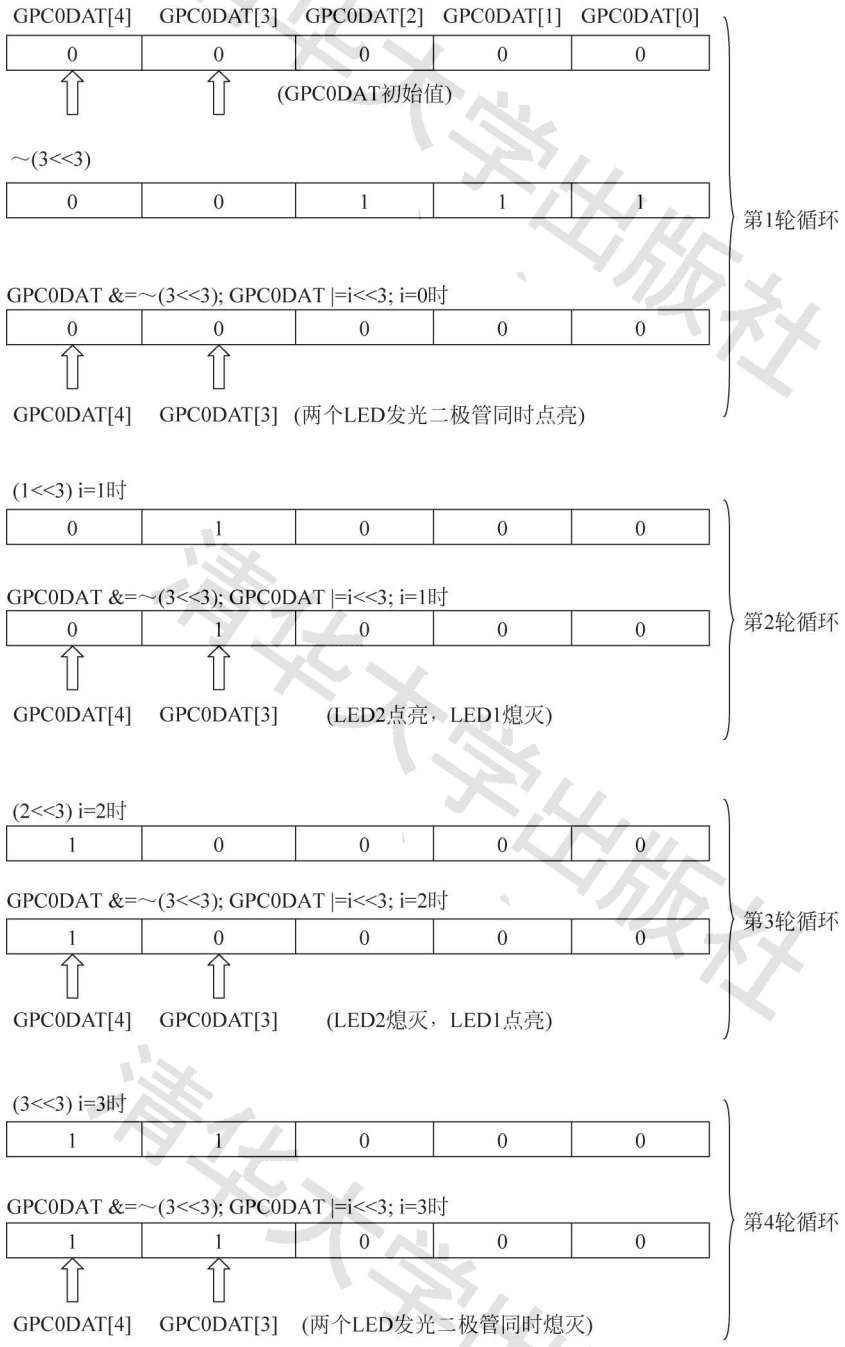

图 4.4　控制 LED 发光二极管点亮及熄灭的过程

```
1  /*************************************
2   * 设置寄存器地址控制 LED 发光二极管 *
3   *************************************/
4  /* GPC0 控制寄存器 */
5  volatile unsigned int *GPC0CON = (volatile unsigned int *) 0xE0200060;
6  /* GPC0 数据寄存器 */
7  volatile unsigned int *GPC0DAT = (volatile unsigned int *) 0xE0200064;
```

```
/*定义延时函数*/
void delay()
{
  int k = 0x100000;
  while (k--);
}
int main(void)
{
  int i = 0;
  *GPC0CON &=  ~0xff000;
  *GPC0CON |= 0x11000;
  while (1)
   {
    *GPC0DAT &= ~(3 << 3);
    *GPC0DAT |= i << 3;
    i++;
    if (i == 4) { i = 0;}     //控制循环4次,再重复
    delay();     //延时
   }
 return 0;
}
```

设置 GPC0CON[4]、GPC0CON[3]引脚为输出模式

GPC0DAT 值为 0 时 LED 点亮,为 1 时熄灭

在 Cortex A8 微处理器的开发板上运行程序,可以看到,两个 LED 发光二极管交替点亮或熄灭,显示闪烁的效果。

本章小结

本章是在嵌入式 Linux 中进行程序设计的基础,首先介绍了 GCC 编译器的使用,并结合了具体的实例进行讲解。虽然它的选项较多,但掌握常用的一些选项即可。之后,又介绍了 make 工程管理器的使用,这里包括 Makefile 的基本结构、Makefile 的变量定义及其规则和 make 命令的使用。还介绍了 shell 编程的基本知识。最后简单介绍了位运算及对寄存器赋值的方法。有了本章的基础,在以后的学习过程中,会感觉轻松很多。

习　题

1. 请查找资料,看看 GNU 所规定的自由软件的具体协议是什么?

2. 什么是 GCC? 试述它的执行过程。

3. 编写一个简单的 C 程序,输出 Hello,Linux.,在 Linux 下用 GCC 进行编译。

4. 针对例 4-1 中∑n=1+2+3+…+100 求和运算的程序,编写一个 Makefile 文件,对其进行编译。

5. 编写程序,如图 4.5 所示,实现用 Cortex A8 的 S5PV210 的 GPIO 端口的 GPC0[2]引脚控制 LED 发光二极管闪烁。

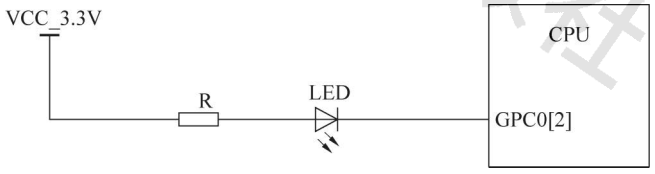

图 4.5　GPIO 端口的 GPC0[2]引脚控制 LED 发光二极管电路