嵌入式系统的构建

试 用 教 材

清华大学自动化系

2003 年 1 月

前 言......1
第一章 嵌入式系统的硬件构成......3
1.1. 嵌入式系统硬件......3
1.1.1. 嵌入式处理器......3
1.1.1.1. 嵌入式微处理器(Embedded Microprocessor Unit, EMPU)......3
1.1.1.2. 嵌入式微控制器(Microcontroller Unit, MCU)......4
1.1.1.3. 嵌入式 DSP 处理器(Embedded Digital Signal Processor, EDSP)......4
1.1.1.4. 嵌入式片上系统(System On Chip)......5
1.1.1.5. 嵌入式处理器的选择......5
1.1.2. 存储器......6
1.1.2.1. ROM......7
1.1.2.2. RAM......7
1.1.3. 输入输出设备......8
1.1.3.1. 液晶显示......8
1.1.3.2. 触摸屏......9
1.1.3.3. 语音输入输出技术......10
1.1.3.4. 键盘......11
1.1.4. 电源转换与管理......13
1.1.4.1. 电源 IC 分类......13
1.1.4.2. 电源 IC 的特点......14
1.1.4.3. 电源 IC 选用指南......14
1.2. 嵌入式系统硬件开发相关技术......15
1.2.1. 接口技术......15
1.2.1.1. 并行接口......16
1.2.1.2. 串口......16
1.2.1.3. USB......17
1.2.1.4. PCMCIA 和 CF......18
1.2.1.5. 红外线接口......19
1.2.2. 总线......20
1.2.2.1. ISA......20
1.2.2.2. PCI......20
1.2.2.3. I2C 总线......21
1.2.2.4. SPI 总线......21
1.2.2.5. PC104 总线......22
1.2.2.6. CAN 总线......23
1.2.3. 嵌入式系统开发常用的硬件调试和编程技术......24
1.2.3.1. 微代码支持的串口调试......24
1.2.3.2. 编程技术......24
1.2.3.3. JATG 与 IEEE1149 协议简介......25
1.2.4. 3.3V 和 5V 装置的互连......26
1.3. 嵌入式系统开发示例——EZ 开发板......27
1.3.1. 系统性能......27
1.3.2. 系统硬件设计......28
1.3.2.1. CPU 与存储器模块......28

1.3.2.2. LCD 显示模块......28
1.3.2.3. 串口模块......29
1.3.2.4. 电源模块......29
1.3.2.5. 进入 BOOTSTRAP 电路模块......30
1.3.3. TQFP 和 LQFP 器件的焊接方法......30
1.3.4. 硬件调试......31
第二章 操作系统......32
2.1. 基础知识......32
2.1.1. 操作系统功能......32
2.1.2. 操作系统发展史......32
2.1.3. Linux 与嵌入式 Linux......33
2.2. 操作系统内核......34
2.2.1. 内存管理......34
2.2.1.1. 内存管理功能......34
2.2.1.2. 内存分割......34
2.2.1.3. 虚拟内存......35
2.2.1.4. Linux 的内存管理机制......37
2.2.2. 进程与中断管理......40
2.2.2.1. 进程描述与控制......41
2.2.2.2. 并发控制：互斥与同步......45
2.2.2.3. 并发控制：死锁处理......52
2.2.2.4. 中断及中断处理......56
2.2.2.5. Linux 的进程与中断管理机制......58
2.2.3. 调度机制......63
2.2.3.1. 调度类型......63
2.2.3.2. 单处理器调度......65
2.2.3.3. 多处理器调度......67
2.2.3.4. 实时调度......69
2.2.3.5. Linux 的调度机制......72
2.2.4. I/O 设备......74
2.2.4.1. I/O 设备描述参数......74
2.2.4.2. I/O 技术的演变......74
2.2.4.3. I/O 设备逻辑描述......75
2.2.4.4. I/O 缓冲技术......77
2.2.4.5. 磁盘调度......79
2.2.5. 文件管理......82
2.2.5.1. 文件与文件系统......82
2.2.5.2. 文件组织与访问......83
2.2.5.3. 文件共享......86
2.2.5.4. 记录分块......86
2.2.5.5. 外围存储设备管理......87
2.2.5.6. Linux 的文件系统管理......88
2.3. 用户界面......89
2.3.1. 图形用户界面......90

2.3.1.1. 基本知识......90
2.3.1.2. 关键技术......90
2.3.2. 智能化用户界面......92
2.3.2.1. Agent 技术......92
2.3.2.2. Agent 技术与用户界面的结合......95
2.3.3. Linux 下的用户界面......96
2.3.3.1. X Window 简介......96
2.3.3.2. X 服务器......97
2.3.3.3. 窗口管理器......99
第三章 嵌入式 Linux......100
3.1. 嵌入式 Linux 内核......100
3.1.1 嵌入式 Linux 综述......100
3.1.2 uCLinux......100
3.1.2.1 uClinux 的内存管理......101
3.1.2.2 uCLinux 内核结构......101
3.1.2.3 内存保护......102
3.1.2.4 编程接口的改变......102
3.1.2.5 uCLinux 的应用程序库......103
3.1.2.6 uCLinux 内核运行方式......104
3.1.2.7. uClinux 支持的文件系统......104
3.2. 嵌入式设备的文件系统......104
3.2.1. 闪存（Flash Memory）介绍......104
3.2.2. 第二版扩展文件系统 Ext2fs（Extended 2 Filesystem）......104
3.2.3. 临时文件系统 tmpfs（Temporary Filesystem）......105
3.2.4. 日志闪存文件系统版本 2－JFFS2（Journalling Flash Filesystem）..106
3.2.4.1 概述......106
3.2.4.2. JFFS 的设计原理：......106
3.3 嵌入式用户界面......108
3.3.1. GUI 开发工具综述......108
3.3.1.1. Xfree86 4.X（带帧缓冲区支持的 X11R6.4）......108
3.3.1.2. Microwindows......109
3.3.1.3. FLTK......109
3.3.1.4. Qt/Embedded......110
3.3.2 MicroWindows 剖析......111
3.3.2.1. 分层设计......111
3.3.2.2. 设备驱动层......111
3.3.2.3. 设备无关的图形引擎层......111
3.3.2.4. API（基于 Win32）......112
3.3.2.5. 消息传递机制......112
3.3.2.6. 窗口操作......112
3.3.2.7. 客户区域和绝对坐标......113
3.3.2.8. 设备上下文......113
3.3.3. Microwindows 的移植和中文化......113
3.3.3.1. 针对 uCLinux 所作的修改......114

3.3.3.2. 中文化处理......114
3.3.4. 应用实例－电子文本阅读器......115
3.3.4.1 功能要求以及方案选择：......115
3.3.4.2 具体实现......115
第四章 交叉编译......117
4.1. 编译原理......117
4.1.1. 基础知识......117
4.1.1.1. 编译的一般过程......117
4.1.1.2. 与编译器相关的程序......118
4.1.1.3. 编译器的移植......119
4.1.2. 词法分析......119
4.1.2.1. 词法的形式化描述......119
4.1.2.2. 词法分析程序的设计......122
4.1.3. 语法分析......123
4.1.3.1. 自顶向下的语法分析......124
4.1.3.2. 自底向上的语法分析......126
4.1.4. 中间代码......128
4.1.5. 代码优化......130
4.2. 交叉编译技术......133
4.2.1. 交叉编译......133
4.2.2. GCC 交叉编译器......134
4.2.2.1. GCC 编译流程......134
4.2.2.2. Linux 环境下的 GCC 交叉编译器......137
第五章 嵌入式 Linux 的软件开发环境......142
5.1. 交叉编译环境......142
5.1.1. 编译开发环境的建立......142
5.1.1.1 安装交叉编译环境......142
5.1.1.2. 添加设备驱动和内核模块......142
5.1.2. 可执行文件......143
5.1.2.1. Coff 文件格式......143
5.1.2.2. elf 文件格式......143
5.1.2.3. flat 文件格式......143
5.2. 调试技术......144
5.2.1. 常见调试方法......144
5.2.2. 内存调试......145
5.2.2. 系统调用跟踪......148
5.2.3. 程序调试......149
5.3. 系统引导和内核启动......154
5.3.1. Bootloader 程序的设计与实现......154
5.3.1.1. 硬件平台的通信......154
5.3.1.2. 硬件平台初始化程序......154
5.3.1.3. 硬件平台监控程序......155
5.3.2. uCLinux 移植......155
5.3.2.1. 第一阶段......155

5.3.2.2. 第二阶段......160
第六章 设备驱动程序......161
6.1. 概述......161
6.1.1. 设备类型......162
6.1.2. 设备号......162
6.1.3. 模块化编程......163
6.2. 设备文件接口......164
6.2.1. 用户访问接口......164
6.2.2. 文件操作......165
6.2.2.1. file_operations 结构......165
6.2.2.2. file 结构......167
6.2.3. I/O 操作......167
6.3. 中断处理......169
6.3.1. 注册中断处理程序......170
6.3.2. 实现中断处理程序......172
6.4. 应用实例......173
6.4.1. 字符型设备......173
6.4.1.1. 按键......173
6.4.1.2. 触摸屏......175
6.4.2. 网络设备......184
6.4.2.1. 网络驱动的框架......184
6.4.2.2. 网卡驱动程序的加载方法......187
6.4.2.3. CS8900A 芯片特点......188
6.4.2.4. CS8900A 芯片驱动程序的实现......190
6.4.2.5. 网络设备驱动程序的编译......196
6.4.2.6. 网络驱动程序的测试......197
附 录......199
A. 参考文献......199
B. 参考网站......200

前 言

近些年来，随着以计算机技术，通讯技术为主的信息技术的快速发展和 Internet 的广泛应用,传统的控制学科正在发生变革，出现了许多新的生长点。伴随而来的一个现象是控制专业的相当多的学生在毕业后进入了计算机，通讯行业，以致有人说学控制没有用，自动化专业可以取消了。这些情况的出现使我们控制教育工作者反复思考，传统的控制应如何拓宽它的领域？控制专业应该教什么才使学生感到有用？

近些年我们在嵌入式系统及其应用的科研工作中采用了信息产业中的最新技术，打破了学科之间的界限，感到控制的出路原来很多，尽管处处是挑战。过去我们熟悉的“控制”有很大的局限性.：一是不考虑硬件的限制, 二是不考虑控制器的复杂性及计算能力， 三是不注重实用性和效益。在微处理器，微传感器和微型执行元件不断推出新产品的形势下,控制的思路与手段正经历着巨大的变化。在经过一番艰苦的实践摸索之后，我们对控制学科的研究和教学有了一些新的认识。本教材就是在我们这些年科研工作的基础上总结出来的，它还比较粗糙，还需要今后花大力气把它完善与提高。现在拿出来作为试用教材供大家参考，希望能为控制学科教学内容的更新作出我们的一点贡献。

嵌入式系统的定义

嵌入式系统是指用于执行独立功能的专用计算机系统。它由包括微处理器、定时器、微控制器、存储器、传感器等一系列微电子芯片与器件，和嵌入在存储器中的微型操作系统、控制应用软件组成，共同实现诸如实时控制、监视、管理、移动计算、数据处理等各种自动化处理任务。嵌入式系统以应用为中心，以微电子技术、控制技术、计算机技术和通讯技术为基础，强调硬件软件的协同性与整合性，软件与硬件可剪裁，以满足系统对功能、成本、体积和功耗等要求。

最简单的嵌入式系统仅有执行单一功能的控制能力，在唯一的 ROM 中仅有实现单一功能的控制程序，无微型操作系统。复杂的嵌入式系统，例如个人数字助理（PDA）、手持电脑（HPC）等，具有与 PC 几乎一样的功能。实质上与 PC 的区别仅仅是将微型操作系统与应用软件嵌入在 ROM、RAM 和/或 FLASH 存储器中，而不是存贮于磁盘等载体中。很多复杂的嵌入式系统又是由若干个小型嵌入式系统组成的。

嵌入式系统的硬件/软件特征

嵌入式系统的硬件必须根据具体的应用任务，以功耗,成本,体积, 可靠性,处理能力等为指标来选择。嵌入式系统的核心是系统软件和应用软件，由于存储空间有限，因而要求软件代码紧凑，可靠，大多对实时性有严格要求。

早期的嵌入式系统设计方法，通常是采用“硬件优先”原则。即在只粗略估计软件任务需求的情况下，首先进行硬件设计与实现。然后，在此硬件平台之上，再进行软件设计。因而很难达到充分利用硬件软件资源，取得最佳性能的效果。同时，一旦在测试时发现问题，需要对设计进行修改时，整个设计流程将重新进行，对成本和设计周期的影响很大。这种传统的设计方法只能改善硬件/软件各自的性能,在有限的设计空间不可能对系统做出较好的性能综合优化，在很大程度上依赖于设计者的经验和反复实验。

90 年代以来随着电子系统功能的日益强大和微型化，系统设计所涉及的问题越来越多，难度也越来越大。同时硬件和软件也不再是截然分开的两个概念，而是紧密结合、相互影响的。因而出现了软硬件协同（codesign）设计方法，即使用统一的方法和工具对软

件和硬件进行描述、综合、和验证。在系统目标要求的指导下，通过综合分析系统软硬件功能及现有资源，协同设计软硬件体系结构，以最大限度地挖掘系统软硬件能力，避免由于独立设计软硬件体系结构而带来的种种弊病，得到高性能低代价的优化设计方案。

嵌入式操作系统

目前流行的嵌入式操作系统可以分为两类：一类是从运行在个人电脑上的操作系统向下移植到嵌入式系统中，形成的嵌入式操作系统，如微软公司的 Windows CE 及其新版本，SUN 公司的 Java 操作系统，朗讯科技公司的 Inferno，嵌入式 Linux 等。这类系统经过个人电脑或高性能计算机等产品的长期运行考验，技术日趋成熟，其相关的标准和软件开发方式已被用户普遍接受，同时积累了丰富的开发工具和应用软件资源。

另一类是实时操作系统，如 WindRiver 公司的 VxWorks，ISI 的 pSOS，QNX 系统软件公司的 QNX，ATI 的 Nucleus，中国科学院凯思集团的 Hopen 嵌入式操作系统等，这类产品在操作系统的结构和实现上都针对所面向的应用领域，对实时性高可靠性等进行了精巧的设计，而且提供了独立而完备的系统开发和测试工具，较多地应用在军用产品和工业控制等领域中。

Linux 是 90 年代以来逐渐成熟的一个开放源代码的操作系统。 PC 机上的 Linux 版本在全球数以百万计爱好者的合力开发下，得到了非常迅速的发展。90 年代末 uClinux，RTLinux 等相继推出，在嵌入式领域得到了广泛的关注，它拥有大批的程序员和现成的应用程序，是我们研究开发工作的宝贵资源。

学习嵌入式系统的意义

从控制意义上说，嵌入式系统涉及系统最底层的，芯片级的信息处理与控制。在某种意义上，对这些“微观”世界的了解与驾驭正是控制的真正目的。嵌入式系统与通常意义上的控制系统在设计思路和总体架构方面有许多不同之处，而这些不同之处恰恰是传统控制学科教学中较少教给学生的。在当今信息化社会中，嵌入式系统在人们的日常工作和生活中所占的份额，可能已超过传统意义的控制系统，这就是为什么我们的学生感到学的没有用，而有用的又没有学的原因。在嵌入式系统及开发环境方面，目前仍有许多问题尚在研究发展之中，如，嵌入式系统的硬件软件协同设计方法；面向多目标，多任务的微内核嵌入式操作系统；分布嵌入式系统的实时性问题，分布式计算，分布式信息交互与综合处理；以及嵌入式系统的多目标交叉编译和交叉调试工具的研究等。我们希望通过这本教材再配合我们的实验开发平台，学习嵌入式系统的一些基本理论和硬件软件综合设计的方法与技能，亲自动手，实现一个嵌入式系统的解决方案，为今后的深入研究打下一个初步基础。

“嵌入式系统”作为自动化学科一门理论与实际密切结合的，知识与技术含量较高的综合性专业课程，必将随着信息产业的发展而逐渐趋于成熟。

本教材第一章由刘森同学执笔，第二，四章由杨占敏同学执笔，陈清阳同学参与编写了其中一部分，第三，五章由钟忻同学执笔，第六章由杜威，李强同学执笔。沈卓立，郭东航，桂伟力，陈建平等同学的工作成果和经验许多被吸收到本教材中。在此，向他们表示诚挚的谢意。

全书由慕春棣策划和组织编写，并负责审校。由于我们的水平有限，许多问题还在摸索之中，加之编写比较仓促，教材中肯定有许多错误和不确切之处，恳请读者批评指正。

慕春棣

2003 年 1 月于清华园

第一章 嵌入式系统的硬件构成

1.1. 嵌入式系统硬件

嵌入式系统是以应用为中心，计算机技术为基础，软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。嵌入式系统硬件一般包括处理器、存储器、外设器件和电源等。

1.1.1. 嵌入式处理器

嵌入式系统的核心部件是各种类型的嵌入式处理器，据不完全统计，到 2000 年全世界嵌入式处理器的品种总量已经超过 1000 多种，流行体系结构有 30 几个系列，其中 8051 体系的占有多半。生产 8051 单片机的半导体厂家有 20 多个，共 350 多种衍生产品，仅 Philips 就有近 100 种。现在几乎每个半导体制造商都生产嵌入式处理器，越来越多的公司有自己的处理器设计部门。嵌入式处理器的寻址空间一般从 64kB 到 16MB，处理速度从 0.1 MIPS 到 2000 MIPS，常用封装从 8 个引脚到 144 个引脚。根据其现状，嵌入式计算机可以分成下面几类。

1.1.1.1. 嵌入式微处理器(Embedded Microprocessor Unit, EMPU)

嵌入式微处理器的基础是通用计算机中的 CPU。在应用中，将微处理器装配在专门设计的电路板上，只保留和嵌入式应用有关的功能，这样可以大幅度减小系统体积和功耗。为了满足嵌入式应用的特殊要求，嵌入式微处理器虽然在功能上和标准微处理器基本是一样的，但在工作温度、抗电磁干扰、可靠性等方面一般都做了各种增强。

和工业控制计算机相比，嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点，但是在电路板上必须包括 ROM、RAM、总线接口、各种外设等器件。嵌入式微处理器及其存储器、总线、外设等安装在一块电路板上，称为单板计算机。如 STD-BUS、PC104 等。近年来，德国、日本的一些公司又开发出了类似“火柴盒”式名片大小的嵌入式计算机系列 OEM 产品，台湾研华公司也推出了类似的模组化系统 SOM（System On Module）。

嵌入式处理器目前主要有 Am186/88、386EX、SC-400、Power PC、68000、MIPS、ARM 系列等。

嵌入式微处理器又可分为 CISC 和 RISC 两类。大家熟悉的大多数台式 PC 都是使用 CISC 微处理器，如 Intel 的 x86。 RISC 结构体系有两大主流：Silicon Graphics 公司（硅谷图形公司）的 MIPS 技术，ARM 公司的 Advanced RISC Machines 技术。此外 Hitachi（日立公司）也有自己的一套 RISC 技术 SuperH。

RISC 和 CISC 是目前设计制造微处理器的两种典型技术，虽然它们都是试图在体系结构、操作运行、软件硬件、编译时间和运行时间等诸多因素中做出某种平衡，以求达到高效的目的，但采用的方法不同，因此，在很多方面差异很大，它们主要有：

（1）指令系统：RISC 设计者把主要精力放在那些经常使用的指令上，尽量使它们具有简单高效的特色。对不常用的功能，常通过组合指令来完成。因此，在 RISC 机器上实现特殊功能时，效率可能较低。但可以利用流水技术和超标量技术加以改进和弥补。而 CISC 计算机的指令系统比较丰富，有专用指令来完成特定的功能。因此，处理特殊任务效率较高。

（2）存储器操作：RISC 对存储器操作有限制，使控制简单化；而 CISC 机器的存储器操作指令多，操作直接。
（3）程序：RISC 汇编语言程序一般需要较大的内存空间，实现特殊功能时程序复杂，不易设计；而 CISC 汇编语言程序编程相对简单，科学计算及复杂操作的程序社设计相对容易，效率较高。
（4）中断：RISC 机器在一条指令执行的适当地方可以响应中断；而 CISC 机器是在一条指令执行结束后响应中断。
（5）CPU：RISC CPU 包含有较少的单元电路，因而面积小、功耗低；而 CISC CPU 包含有丰富的电路单元，因而功能强、面积大、功耗大。
（6）设计周期：RISC 微处理器结构简单，布局紧凑，设计周期短，且易于采用最新技术；CISC 微处理器结构复杂，设计周期长。
（7）用户使用：RISC 微处理器结构简单，指令规整，性能容易把握，易学易用；CISC 微处理器结构复杂，功能强大，实现特殊功能容易。
（8）应用范围：由于 RISC 指令系统的确定与特定的应用领域有关，故 RISC 机器更适合于专用机；而 CISC 机器则更适合于通用机。

1.1.1.2. 嵌入式微控制器(Microcontroller Unit, MCU)

嵌入式微控制器又称单片机，顾名思义，就是将整个计算机系统集成到一块芯片中。嵌入式微控制器一般以某一种微处理器内核为核心，芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、WatchDog、I/O、串行口、脉宽调制输出、A/D、D/A、Flash RAM、EEPROM 等各种必要功能模块。为适应不同的应用需求，一般一个系列的单片机具有多种衍生产品，每种衍生产品的处理器内核都是一样的，不同的是存储器和外设的配置及封装。这样可以使单片机最大限度地和应用需求相匹配，从而减少功耗和成本。

和嵌入式微处理器相比，微控制器的最大特点是单片化，体积大大减小，从而使功耗和成本下降、可靠性提高。微控制器是目前嵌入式系统工业的主流。微控制器的片上资源一般比较丰富，适合于控制，因此称微控制器。

嵌入式微控制器目前的品种和数量最多，比较有代表性的通用系列包括 8051、P51XA、MCS-251、MCS-96/196/296、C166/167、MC68HC05/11/12/16、68300 等。另外还有许多半通用系列，如支持 USB 接口的 MCU 8XC930/931、C540、C541。目前 MCU 占嵌入式系统约 70％的市场份额。

特别值得注意的是近年来提供 X86 微处理器的著名厂商 AMD 公司，将 Am186CC/CH/CU 等嵌入式处理器称之为 Microcontroller, MOTOROLA 公司把以 Power PC 为基础的 PPC505 和 PPC555 亦列入单片机行列。TI 公司亦将其 TMS320C2XXX 系列 DSP 做为 MCU 进行推广。

1.1.1.3. 嵌入式 DSP 处理器(Embedded Digital Signal Processor, EDSP)

DSP 处理器对系统结构和指令进行了特殊设计，使其适合于执行 DSP 算法，编译效率较高，指令执行速度也较高。在数字滤波、FFT、谱分析等方面 DSP 算法正在大量进入嵌入式领域，DSP 应用正从在通用单片机中以普通指令实现 DSP 功能，过渡到采用嵌入式 DSP 处理器。嵌入式 DSP 处理器有两个发展来源，一是 DSP 处理器经过单片化、EMC 改造、增加片上外设成为嵌入式 DSP 处理器，TI 的 TMS320C2000/C5000 等属于此范畴；二

是在通用单片机或片上系统（SOC）中增加 DSP 协处理器，例如 Intel 的 MCS-296。

推动嵌入式 DSP 处理器发展的一个重要因素是嵌入式系统的智能化，例如各种带有智能逻辑的消费类产品，生物信息识别终端，带有加解密算法的键盘，ADSL 接入、实时语音压解系统，虚拟现实显示等。这类智能化算法一般都是运算量较大，特别是向量运算、指针线性寻址等较多，而这些正是 DSP 处理器的长处所在。

嵌入式 DSP 处理器比较有代表性的产品是 Texas Instruments 的 TMS320 系列和 Motorola 的 DSP56000 系列。TMS320 系列处理器包括用于控制的 C2000 系列，移动通信的 C5000 系列，以及性能更高的 C6000 和 C8000 系列。DSP56000 目前已经发展成为 DSP56000，DSP56100，DSP56200 和 DSP56300 等几个不同系列的处理器。

DSP 的设计者们把重点放在了处理连续的数据流上。在嵌入式应用中，如果强调对连续的数据流的处理及高精度复杂运算，则应该选用 DSP 器件。

1.1.1.4. 嵌入式片上系统(System On Chip)

随着 VLSI 设计的普及化及半导体工艺的迅速发展，可以在一块硅片上实现一个更为复杂的系统，这就是 System On Chip(SOC)。各种通用处理器内核将作为 SOC 设计公司的标准库，和许多其它嵌入式系统外设一样，成为 VLSI 设计中一种标准的器件，用标准的 VHDL 等语言描述，存储在器件库中。用户只需定义出整个应用系统，仿真通过后就可以将设计图交给半导体工厂制作样品。这样除个别无法集成的器件以外，整个嵌入式系统大部分均可集成到一块或几块芯片中去，应用系统电路板将变得很简洁，对于减小体积和功耗、提高可靠性非常有利。

SOC 可以分为通用和专用两类。通用系列包括 Motorola 的 M-Core，某些 ARM 系列器件，Echelon 和 Motorola 联合研制的 Neuron 芯片等。专用 SOC 一般专用于某个或某类系统中，不为一般用户所知。一个有代表性的产品是 Philips 的 Smart XA，它将 XA 单片机内核和支持超过 2048 位复杂 RSA 算法的 CCU 单元制作在一块硅片上，形成一个可加载 JAVA 或 C 语言的专用的 SOC，可用于公众互联网如 Internet 安全方面。

1.1.1.5. 嵌入式处理器的选择

针对各种嵌入式设备的需求，各个半导体芯片厂商都投入了很大的力量研发和生产适用于这些设备的 CPU 及协处理器芯片。用于嵌入式设备的处理器必须高度紧凑、低功耗、低成本。针对每一类应用来说，开发者对处理器选择都是多种多样的，掌上电脑就是一例，如表 1.1.所示。

与全球 PC 市场不同的是没有一种微处理器和微处理器公司可以主导嵌入式系统，仅以 32 位的 CPU 而言，就有 100 种以上嵌入式微处理器。由于嵌入式系统设计的差异性极大，因此选择是多样化的。设计者在选择处理器时要考虑的主要因素有：

（1）调查市场上已有的 CPU 供应商。有些公司如 Motorola、Intel、AMD 很有名气，而有一些小的公司如 QED 虽然名气很小，但也生产很优秀的微处理器。另外，有一些公司，如 ARM、MIPS 等，只设计但并不生产 CPU，他们把生产权授予世界各地的半导体制造商。ARM 是另外一种近年来在嵌入式系统有影响力的微处理器制造商，ARM 的设计非常适合于小的电源供电系统。Apple 在 Newton 手持计算机中使用 ARM，另外有几款数字无线电话也在使用 ARM。

（2）处理器的处理速度。一个处理器的性能取决于多个方面的因素：时钟频率，内部

寄存器的大小，指令是否对等处理所有的寄存器等。对于许多需用处理器的嵌入式系统设计来说，目标不是在于挑选速度最快的处理器，而是在于选取能够完成作业的处理器和 I / O 子系统。如果你的设计是面向高性能的应用，那么建议你考虑某些新的处理器，其价格极为低廉，如 IBM 和 Motorola 的 Power PC。以前 Intel 的 i960 是销售极好的 RISC 高性能芯片，但是最近几年却遇到强劲的对手，让位于 MIPS、SH 以及后起之星 ARM。

（3）技术指标。当前，许多嵌入式处理器都集成了外围设备的功能，从而减少了芯片的数量，进而降低了整个系统的开发费用。开发人员首先考虑的是，系统所要求的一些硬件能否无需过多的胶合逻辑（Glue Logic）就可以连接到处理器上。其次是考虑该处理器的一些支持芯片，如 DMA 控制器，内存管理器，中断控制器，串行设备、时钟等的配套。

（4）处理器的低功耗。嵌入式微处理器最大并且增长最快的市场是手持设备、电子记事本、PDA、手机、GPS 导航器、智能家电等消费类电子产品，这些产品中选购的微处理器典型的特点是要求高性能、低功耗。许多 CPU 生产厂家已经进入了这个领域。

（5）处理器的软件支持工具。仅有一个处理器，没有较好的软件开发工具的支持，也是不行的，因此选择合适的软件开发工具对系统的实现会起到很好的作用。

（6）处理器是否内置调试工具。处理器如果内置调试工具可以大大大的缩小调试周期，降低调试的难度。

（7）处理器供应商是否提供评估板。许多处理器供应商可以提供评估板来验证你的理论是否正确，验证你的决策是否得当。

表 1.1. 部分掌上电脑处理器一览

厂家/型号	处理器	速度
卡西欧 Cassiopeia E-100 系列	MIPS-based NEC VR4121	131 MHz
康柏 Aero 2100 系列	MIPS-based NEC VR4111	70 MHz
菲利浦 Nino 500 系列	MIPS-based Toshiba PR31700	75 MHz
惠普 Jornada 400 系列	Hitachi SH-3 7709a	100 MHz /133 MHz
3Com PalmPilot™ 系列	Motorola DragonBall 68VZ328	33 MHz
苹果 MessagePad 2000/2100	Intel StrongARM SA-110	160MHz
康柏 iPAQ H3650	Intel StrongARM SA-1110	206MHz

1.1.2. 存储器

存储器的物理实质是一组或多组具备数据输入输出和数据存储功能的集成电路，用于充当设备缓存或保存固定的程序及数据。存储器按存储信息的功能可分为只读存储器 ROM（Read Only Memory）和随机存储器 RAM（Random Access Memory）。

表 1.2.　常用存储器分类

	种类	存储器单元构造	读出速度	数据变更		数据保持功率	相对位成本
				变更方式	写入速度		
ROM	EEPROM	2 晶体管＋隧道区域	同上	写动作（重写）	10ms	不要	10
	EPROM	1 晶体管	同上	紫外线擦除＋电气写入	（1－9）×100us	不要	1.2
	FLASH	1 晶体管	同上	电器擦除＋电气写入	10us	不要	1
	OTP	1 晶体管	同上	不可变更数据（仅可一次电气写入）		不要	0.8
	MASK ROM	1 晶体管	同上	不可		不要	0.5
RAM	DRAM	1 晶体管＋1 电容器	（1－9）×10ns	写动作（重写）	（1－9）×10ns	（1－9）×100uW	1
	SRAM	4 晶体管＋2 负载元件	（1－9）×10ns	写动作（重写）	（1－9）×10ns	（1－9）×100uS	4
	NVRAM	SRAM 单元＋EEPROM 单元	100ns	存储器动作（片批）（重写）	10ms	不要	1000

1.1.2.1. ROM

ROM 中的信息一次写入后只能被读出，而不能被操作者修改或删除，一般由芯片制造商进行掩膜写入信息，价格便宜，适合于大量的应用。一般用于存放固定的程序，如监控程序、汇编程序等，以及存放各种表格。EPROM（Erasable Programmable ROM）和一般的 ROM 不同点在于它可以用特殊的装置擦除和重写它的内容，一般用于软件的开发过程。

1.1.2.2. RAM

RAM 就是我们平常所说的内存，主要用来存放各种现场的输入、输出数据，中间计算结果，以及与外部存储器交换信息和作堆栈用。它的存储单元根据具体需要可以读出，也可以写入或改写。RAM 只能用于暂时存放程序和数据，一旦关闭电源或发生断电，其中的数据就会丢失。现在的 RAM 多为 MOS 型半导体电路，它分为静态和动态两种。静态 RAM 是靠双稳态触发器来记忆信息的；动态 RAM 是靠 MOS 电路中的栅极电容来记忆信息的。由于电容上的电荷会泄漏，需要定时给与补充，所以动态 RAM 需要设置刷新电路。但动态 RAM 比静态 RAM 集成度高、功耗低，从而成本也低，适于作大容量存储器。所以主内存通常采用动态 RAM，而高速缓冲存储器（Cache）则使用静态 RAM。

动态 RAM 按制造工艺的不同，又可分为动态随机存储器（Dynamic RAM）、扩展数据输出随机存储器（Extended Data Out RAM）和同步动态随机存储器（Synchronized Dynamic

RAM）。DRAM 需要恒电流以保存信息，一旦断电，信息即丢失。它的刷新频率每秒钟可达几百次，但由于 DRAM 使用同一电路来存取数据，所以 DRAM 的存取时间有一定的时间间隔，这导致了它的存取速度并不是很快。另外，在 DRAM 中，由于存储地址空间是按页排列的，所以当访问某一页面时，切换到另一页面会占用 CPU 额外的时钟周期。EDO-RAM 同 DRAM 相似，但在把数据发送给 CPU 的同时可以去访问下一个页面，故而速度要比普通 DRAM 快 15~30%。SDRAM 同 DRAM 有很大区别，它使用同一个 CPU 时钟周期即可完成数据的访问和刷新，即以同一个周期、相同的速度、同步的工作，因而可以同系统总线以同频率工作，可大大提高数据传输率，其速度要比 DRAM 和 EDO-RAM 快很多（比 EDO-RAM 提高近 50%）。

1.1.3. 输入输出设备

嵌入式系统中输入形式一般包括触摸屏、语音识别、按键、键盘和虚拟键盘。输出设备主要有 LCD 显示和语音输出。

1.1.3.1. 液晶显示

液晶显示屏（liquid crystal display：LCD）用于显示 GUI（图象用户界面）环境下的文字和图象数据，适用于低压、微功耗电路。

从选型角度，我们将常见液晶分为以下几类：段式（也称 8 字）、字符型和图形点阵。

段式液晶：常见段式液晶的每字为 8 段组成，即 8 字和一点，只能显示数字和部分字母，如果必须显示其它少量字符、汉字和其它符号，一般需要从厂家定做，可以将所要显示的字符、汉字和其它符号固化在指定的位置，比如计算器和电子表所用的液晶。

字符型液晶：顾名思义，字符型液晶是用于显示字符和数字的，对于图形和汉字的显示方式与段式液晶无异。字符型液晶一般有以下几种分辨率，8×1，16×1、16×2、16×4、20×2、20×4、40×2、40×4 等，其中 8（16、20、40）的意义为一行可显示的字符（数字）数，1（2、4）的意义是指显示行数。

图形点阵式液晶：我们又将其分为 TN、STN（DSTN）、TFT 等几类。这种分类需从液晶材料和液晶效应讲起，请参考液晶显示原理。

TN 类液晶由于它的局限性，只用于生产字符型液晶模块；而 ST（DSTN）类液晶模块一般为中小型，既有单色的，也有伪彩色的；TFT 类液晶，则从小到大都有，而且几乎清一色为真彩色显示模块。除了 TFT 类液晶外，一般小液晶屏都内置控制器(控制器的概念相当于显示卡上的主控芯片)，直接提供 MPU 接口；而大中液晶屏，要想控制其显示，都需要外加控制器。

从色彩上分，LCD 显示屏分为单色、灰度和彩色三种，价格由低到高，单色 LCD 的点阵只能显示亮和暗，通常只用于低端的不需显示图形的场合；带灰度级的 LCD 常用的有 2bit 4 级灰度和 4bit 16 级灰度，可以显示简单的带有层次的图形或图象；彩色 LCD 的色彩以颜色数为标准。彩色 LCD 分为有源（Active）及无源（Passive）型两种，有源型就是常见的 TFT（Thin Film Transistor，薄膜晶体管）LCD，特点是显示清晰、分明、视角大，但价格高。之所以如此，是因为有源 LCD 更新屏幕的频率较快，而且它屏幕上的每个象素，分别是由一个独立的晶体管控制的（无源的就不是）。这样，也导致了有源矩阵 LCD 的一个缺点，就是这种显示器要使用相当多的晶体管，造价也就高。无源型就是常见的 STN（super-twist ed nematic，超扭曲向列型） LCD，最显著优点是造价低。

按背光将液晶分类，有透射式、反射式、半反半透式液晶三类，因为液晶为被动发光型显示器，所以必须有外界光源，液晶才会有显示，透射式液晶必须加上背景光，反射式液晶需要较强的环境光线，半反半透式液晶要求环境光线较强或加背光。

字符类液晶，带背光的一般为 LED 背光，以黄颜色（红、绿色调）为主。一般为+5V 驱动。单色 STN 中小点阵液晶，多用 LED 或 EL 背光，EL 背光以黄绿色（红、绿、白色调）

常见。一般用 400—800Hz、70—100V 的交流驱动，常用驱动需要约 1W 的功率。中大点阵 STN 液晶和 TFT 类液晶，多为冷阴极背光灯管（CCFL/CCFT），背光颜色为白色（红、绿、蓝色调）。一般用 25k—100kHz，300V 以上的交流驱动。

1.1.3.2. 触摸屏

嵌入式系统中的触摸屏分为电阻式、电容式和电感式三种，其中电阻式触摸屏最为常用。

电阻触摸屏的工作部分一般由三部分组成，如图 1.1.所示：两层透明的阻性导体层、两层导体之间的隔离层、电极。阻性导体层选用阻性材料，如铟锡氧化物（ITO）涂在衬底上构成，上层衬底用塑料，下层衬底用玻璃。隔离层为粘性绝缘液体材料，如聚脂薄膜。电极选用导电性能极好的材料（如银粉墨）构成，其导电性能大约为 ITO 的 1000 倍。

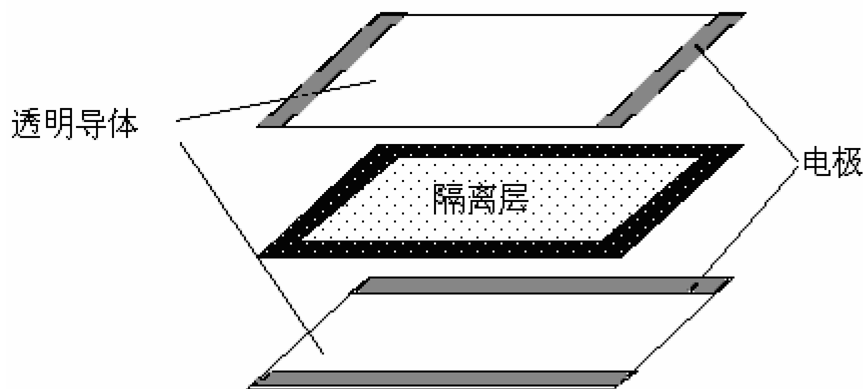


图 1.1.　电阻触摸屏结构

触摸屏工作时，上下导体层相当于电阻网络，如图 1.2.所示。当某一层电极加上电压时，会在该网络上形成电压梯度。如有外力使得上下两层在某一点接触，则在电极未加电压的另一层可以测得接触点处的电压，从而知道接触点处的坐标。比如，在顶层的电极(X+,X－)上加上电压，则在顶层导体层上形成电压梯度，当有外力使得上下两层在某一点接触，在底层就可以测得接触点处的电压，再根据该电压与电极(X+)之间的距离关系，知道该处的 X 坐标。然后，将电压切换到底层电极（Y+,Y－）上，并在顶层测量接触点处的电压，从而知道 Y 坐标。这就是所有电阻技术触摸屏共同的最基本原理。对电阻式触摸屏的控制有专门的芯片，如 BB(Burr-Brown)公司生产的芯片 ADS7843。很显然，控制芯片要完成两件事情：其一，是完成电极电压的切换；其二，是采集接触点处的电压值（即 A/D）。

电容式触摸屏是一块四层复合玻璃屏，玻璃屏的内表面和夹层各涂一层 ITO，最外层是只有 0.0015mm 厚的矽土玻璃保护层，夹层 ITO 涂层作工作面，四个角引出四个电极，内层 ITO 为屏层以保证工作环境。当用户触摸电容屏时，由于人体电场，用户手指和工作面形成一个耦合电容，因为工作面上接有高频信号，于是手指会吸收一个很小的电流，这个电流分别从屏的四个角上的电极中流出，且理论上流经四个电极的电流与手指头到四角的距离成比例，控制器通过对四个电流比例的精密计算，得出位置。

电感式触摸屏的工作原理是在触摸笔中安装 LC 谐振线圈，通过改变与安装有激励线圈及感应线圈的触摸屏之间的空间距离，使电磁场发生变化从而计算出触点的位置。因为这种触摸屏是安装在液晶显示屏的后面，而普通的电阻式和电容式触摸屏需要安装在液晶显示屏的前面，两者相比，使用电感式触摸屏，输入笔不必接触屏幕，可以减少对屏幕的磨损，同时大大提高输入的灵敏度。由于触摸屏安装在显示屏的后面，也增加显示的清晰度和亮度，减少背光的使用，进而可以减少系统功耗。

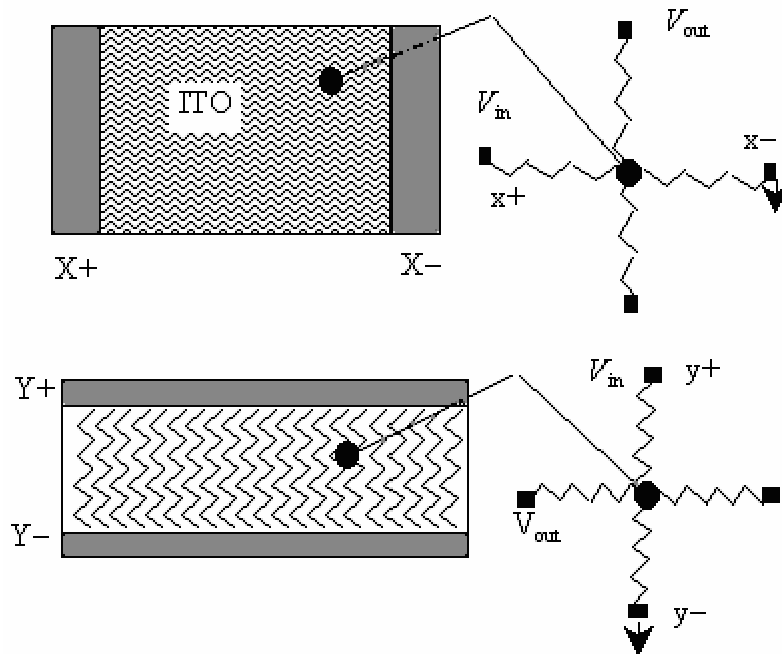


图 1.2. 工作时的导体层

表 1.3. 三种触摸屏技术的比较

触摸屏类型	工作原理	触摸方式	安装方式	透明度	易用性
电阻式	电压测量	笔、手指点压	显示屏前	一般	好
电容式	电容耦合	金属笔尖、手指接触	显示屏前	比电阻式好	一般
电感式	电磁谐振	笔尖接近感应	显示屏后	好	好

1.1.3.3. 语音输入输出技术

用户不断地要求所用的嵌入式装置更小、更轻便同时又更易于使用，能解决这三方面问题的一种可行技术就是语音识别。这种技术由于去掉了传统的输入器件，故具有更小和更轻便的特点。语音识别输入的实现可以在嵌入式处理器功能足够强大时用相应的软件实现，也可以使用专用芯片增加一个硬件功能模块。前者对嵌入式系统硬件配置的要求较高，如处理器的速度和存储器的容量等。后者则通过专门的 DSP 芯片来完成语音识别输入过程中的所有运算工作，不会加重系统主处理器的负担。这种专用的语音识别芯片现在已经有多种产品出现了。如 sensory 公司的语音识别芯片系列，内部采用神经网络技术来识别训练过的单词或短语，准确度高于 99%。并且芯片高度集成化，减少了所需外部元器件的数目。可以快速和方便地嵌入到现存的和新设计的产品中，适用于多种需要语音控制的嵌入式产品。

语音识别技术以识别方法来分，有模板匹配法、随机模型法和概率语法分析法。这三种方法都属于统计模式识别方法。它的识别过程大致如下：首先选定语音的特征作为识别参数的模板，然后采用一可以衡量未知模式和参考模式（即模板）的似然度的测量函数，最后选用一种最佳准则及专家知识作为识别策略，对识别候选者作最后判决，得到最好的识别结果作为输出。以识别范围来分，分为语音从属（speaker-dependent）模式和语音独立（speaker-independent）模式。语音从属意味着必须有培训系统，而且通常它只可识别培训系统的人所讲的词。语音独立系统则可识别几乎所有讲话人的词。从目前水平来看，语音从

属模式下的模板匹配法用得比较广泛。

语音识别技术在嵌入式系统上的使用，不仅可以通过声音命令来控制设备，还可将输入的声音转换为文本，使得用户就能用声音口述需要输入的文本。如果再加上语音合成输出功能，就可以在嵌入式系统中实现书面语言和口头语言的双向转换，从而构成完整的语音输入输出功能。

1.1.3.4. 键盘

键盘输入作为最常用的输入设备仍有其不可替代的作用。

首先，对传统键盘作一个简单的介绍。

- 传统键盘的介绍

键盘的结构通常有两种形式：线性键盘和矩阵键盘。在不同的场合下，这两种键盘均得到了广泛的应用。

线性键盘由若干个独立的按键组成，每个按键的一端与微机的一个 I/O 口相连。有多少个键就要有多少根连线与微机的 I/O 口相连，因此，只适用于按键少的场合。

矩阵键盘的按键按 N 行 M 列排列，每个按键占据行列的一个交点，需要的 I/O 口数目是 N+M，容许的最大按键数是 N×M。显然，矩阵键盘可以减少与微机接口的连线数，简化结构，是一般微机常用的键盘结构。根据矩阵键盘的识键和译键方法的不同，矩阵键盘又可以分为非编码键盘和编码键盘两种。

◇ 非编码键盘

非编码键盘主要用软件的方法识键和译键。根据扫描方法的不同，可以分为行扫描法、列扫描法和反转法三种。

◇ 编码键盘

编码键盘主要用硬件来实现键的扫描和识别，通常使用 8279 专用接口芯片，在硬件上要求较高。

- 新型键盘的硬件和软件实现原理

有些特殊情况下，在组成一个最小的单片机系统的过程中，由于通用的 I/O 口有限，而又需要大量的按键输入，这就要求一种新的键盘结构，即用尽量少的 I/O 口实现尽可能多的键盘输入。经过分析，实际上用 N+1 个 I/O 口，辅以适当的接口电路，是可以实现 N×N 个按键的。现以 6 个端口实现 5×5 的按键为例来叙述。

1. 硬件实现

图 1.3.所示为用 6 个 I/O 口来实现 25 个按键的示意图。

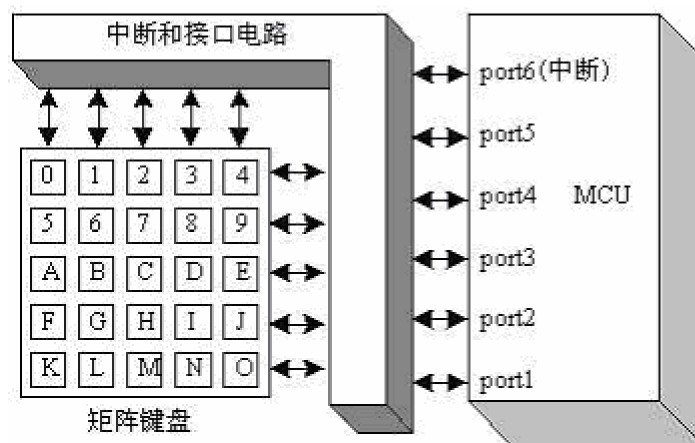


图 1.3.　6 个 I/O 口实现的 5×5 按键矩阵的示意图

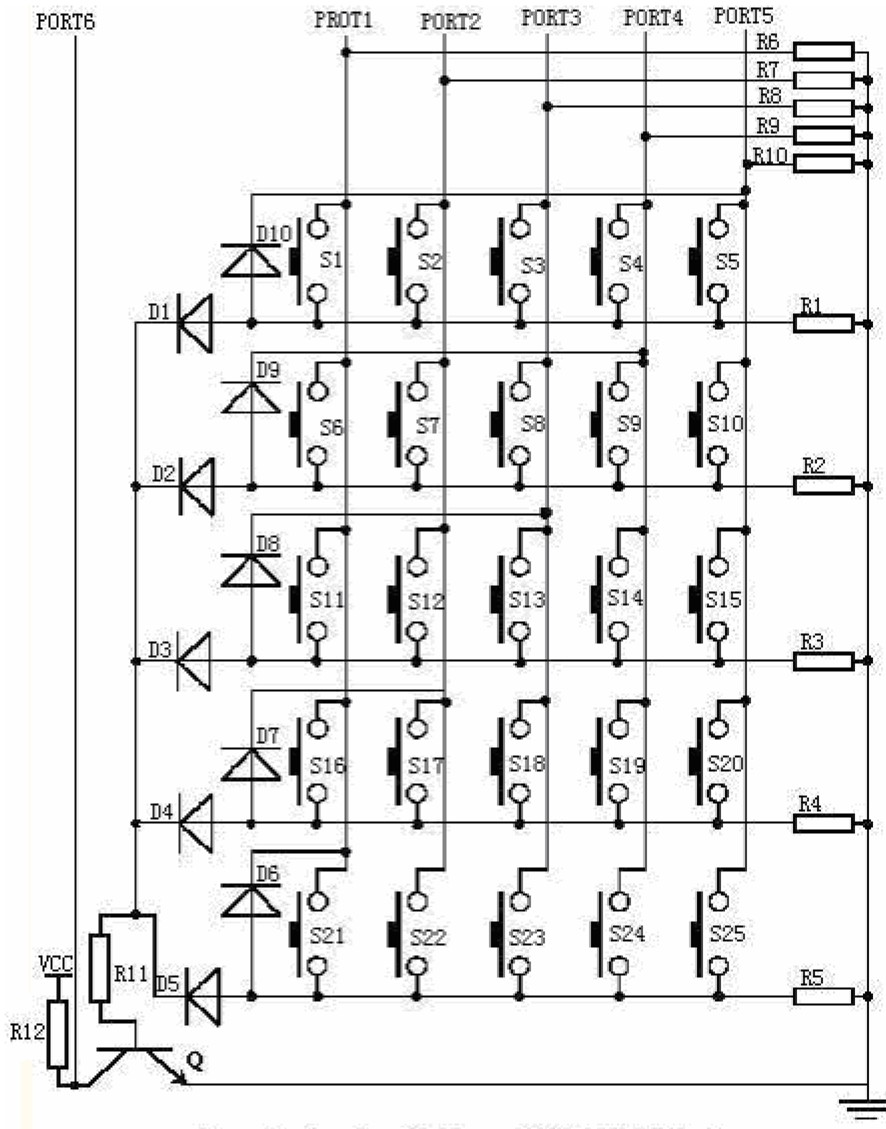


图 1.4.　6 个 I/O 口实现的 5×5 按键矩阵的原理图

具体的物理实现电路如图 1.4.所示。

由图 1.3.和图 1.4.可见，硬件部分分为两块：一块是普通键盘矩阵，另外一块是中断和接口电路，主要由相应数目的二极管和电阻组成。具体对 6 个 I/O 口的情况，实现 5×5 的按键矩阵的中断和接口电路（图 1.4.）共需要 10 只二极管、12 只电阻和 1 只三极管。10 只二极管按其在电路中所起的作用可分为两组：第一组包括 D6、D7、D8、D9 和 D10，用于保证按键信息的单一流向；第二组包括 D1、D2、D3、D4 和 D5，它们在电路上对 NPN 三极管的基极构成"或"的逻辑关系，对单片机进行初始化。除了 PORT6（其要求具有中断功能）以外，其余的 I/O 口均被置成高电平，这样当有键按下时，三极管的基极由低变高，三极管导通；集电极由高电平跳变成低电平，向单片机发出中断信号，从而启动键盘扫描程序。

按键的识别主要靠软件来实现，需要编写键盘扫描程序。为了更好地说明键盘扫描的过程，假设编号为 S12 的键（见图 1.4.）被按下，扫描程序已经启动，扫描的具体过程如表 1.4.所列。

表 1.4. 键盘扫描过程

P1	P2	P3	P4	P5	P6	说明
OH	I	I	I	I	I	PORT1 被置成高电平输出，其他端口作为输入
H	L	L	L	L	H	读入各端口的值，为 100001，与编码表进行比较，京判断没有键按下，继续下一步
I	OH	I	I	I	I	PORT2 被置成高电平输出，其他端口作为输入
L	H	H	L	L	L	PORT2 和 PORT3 与 S12 相连，此时均为高电平，读入各端口的值为 011000，与编码表比较，可以判断有键按下并且键值唯一

P－PORT　O－输出　I－输入　H－高电平　L－低电平

1.1.4. 电源转换与管理

电源是电子产品中一个组成部分，为了使电路性能稳定，往往还需要稳定电源。设计者要根据产品的要求来选择合适的电源 IC。为了合理地选择电源 IC，首先要了解各种电源 IC 及其特点。

1.1.4.1. 电源 IC 分类

根据不同的工作原理可将电源分成三类：线性稳压电源、开关稳压电源及电荷泵电源。它们各自都有一定的特点及适用范围，这里分别作一简介。

（1） 线性稳压电源

线性稳压电源是因其内部调整管工作在线性范围而得名。一般认为线性稳压电源的输入电压与输出电压之间的电压差（一般称为压差）大，调整管上的损耗大，效率低。但近年来开发出各种低压差（LDO）的新型线性稳压器 IC，一般可达到达输出 100mA 电流时，其压差在 100mV 左右的水平（甚至于到 70-80mv 的水平），某些小电流的低压差线性稳压器其压差仅几十毫伏。这样，调整管的损耗较小，效率也有较大的提高，因此可减少功耗。另外，线性稳压电源外围元件最少、输出噪声最小、静态电流最小，价格也便宜。

（2）开关稳压电源

在便携式电子产品中，开关稳压电源主要指 DC/DC 变换器。由于器件中有一个工作在开关状态的晶体管（一般是 MOSFET），故称为开关电源。开关管工作于饱和导通及截止两种状态，所以开关管管耗小并且与输入电压大小无关，效率较高（一般可达 80～95%）、可以输出大电流、静态电流小。随著集成度的提高，许多新型 DC-DC 转换器仅需要几只外接电感器和滤波电容器。但是，这类电源控制器的输出脉动和开关噪音较大、成本相对较高。

DC/DC 变换器 IC 包括升压式（VOUT>VIN）、降压式（VOUT< VIN）和反相式（也叫电压反转式）等电路。降压式主要用于工作电流大于 1A 以上的场合，如笔记本式计算机等。反相式 DC/DC 变换器的特点是可以获得负电压，并且可获得大于输入电压的负压，即|-VOUT| >VIN，可输出较大的电流。但采用电荷泵电路来获得负压更为简单，并且有带线性稳压输出的电荷泵 IC，所以电压反转式 DC/DC 变换器也很少用。

（3）电荷泵电源

电荷泵的工作原理是利用高频振荡器控制电容的充、放电，将能量由输入传给负载（输出）。其特点是体积小，电路结构简单，可产生输入的反相或倍压输出。基本的电荷泵电路

的成本较低，它的优点是无需电感，只需要几个外接电容器，体积较小，效率高达 95 %。由於开关不停地动作，产生的噪声比较大，静态电流也比较大。另外，这种电路的输出电压只能取输入电压的倍数，例如，用四个内部开关和一个外接电容器就能够产生 2 VIN、1/2 VIN 或-VIN 的输出。使用几个这样的充电泵可以得到其它倍数的输出电压，然而成本和静态电流也会增加。所以，在传统的设计中，电荷泵很少直接地接到电源上，而是把它接在稳压器後面，用它产生辅助电源，为某一个器件供电。

1.1.4.2. 电源 IC 的特点

由上节可见，电源 IC 种类繁多，它们的共同特点有：

（1）工作电压低

一般的工作电压为 3.0～3.6V。有一些工作电压更低，如 2.0、2.5、2.7V 等；也有一些工作电压为 5V，还有少数 12V 或 28V 的特殊用途的电压源。

（2）工作电流不大

从几毫安到几安都有，但由于大多数嵌入式电子产品的工作电流小于 300mA，所以 30～300mA 的电源 IC 在品种及数量上占较大的比例。

（3）封装尺寸小

近年来发展的便携式产品都采用贴片式器件，电源 IC 也不例外，主要有 SO 封装、SOT-23 封装，μMAX 封装及封装尺寸最小的 SC-70 及最新的 SMD 封装等，使电源占的空间越来越小。

（4）完善的保护措施

新型电源 IC 有完善的保护措施，这包括：输出过流限制、过热保护、短路保护及电池极性接反保护，使电源工作安全可靠，不易损坏。

（5）耗电小及关闭电源功能

新型电源 IC 的静态电流都较小，一般为几十μA 到几百μA。个别微功耗的线性稳压器其静态电流仅 1.1μA。另外，不少电源 IC 有关闭电源控制端功能（用电平来控制），在关闭电源状态时 IC 自身耗电在 1μA 左右。由于它可使一部分电路不工作，可大大节省电能。例如，在无线通信设备上，在发送状态时可关闭接收电路；在未接收到信号时可关闭显示电路等。

（5）有电源工作状态信号输出

不少便携式电子产品中有单片机，在电源因过热或电池低电压而使输出电压下降一定百分数时，电源 IC 有一个电源工作状态信号输给单片机，使单片机复位。利用这个信号也可以做成电源工作状态指示（当电池低电压时，有 LED 显示）。

（6）输出电压精度高

一般的输出电压精度为±2～4%之间，有不少新型电源 IC 的精度可达±0.5～±1%；并且输出电压温度系数较小，一般为±0.3～±0.5mV/℃，而有一些可达到±0.1mV/℃的水平。线性调整率一般为 0.05%～0.1%/V，有的可达 0.01%/V；负载调整率一般为 0.3～0.5%/mA，有的可达 0.01%/mA。

（7）新型组合式电源 IC

升压式 DC/DC 变换器的效率高但纹波及噪声电压较大，低压差线性稳压器效率低但噪声最小，这两者结合组成的双输出电源 IC 可较好地解决效率及噪声的问题。例如，数字电路部分采用升压式 DC/DC 变换器电源而对噪声敏感的电路采用 LDO 电源。这种电源 IC 有 MAX710/711，MAX1705/1706 等。另一种例子是电荷泵+LDO 组成，输出稳压的电荷泵电源 IC，例如 MAX868，它可输出 0～-2VIN 可调的稳定电压，并可提供 30mA 电流；MAX1673 稳压型电荷泵电源 IC 输出与 VIN 相同的负压，输出电流可达 125mA。

1.1.4.3. 电源 IC 选用指南

选择电源 IC 不仅仅要考虑满足电路性能的要求及可靠性，还要考虑它的体积、重量及

成本等问题。这里给出一些选择基本原则，供参考。

（1）优先考虑升压式 DC/DC 变换器

采用升压式 DC/DC 变换器不仅效率高并且可减少功耗（减小整个电源体积及重量）。例如 MAX1674/1675 高效率、低功耗升压式 DC/DC 变换器 IC，其静态电流仅 16μA，在输 出 200mA 时效率可达 94%，在关闭电源时耗电仅 0.1μA，并可选择电流限制来降低纹波电压。

（2）采用 LDO 的最佳条件

当要求输出电压中纹波、噪声特别小，输入输出电压差不大，输出电流不大于 100mA 时，采用微功耗、低压差（LDO）线性稳压器是最合适的。例如，采用 3 节镍镉、镍氢电池或采用 1 节锂离子电池，输出 3.0～3.3V 电压，工作电流小于 100mA 时，电池寿命较长，并且有较高的效率。

（3）需负电源时尽量采用电荷泵

便携式仪器中往往需要负电源，由于所需电流不大，采用电荷泵 IC 组成电压反转电路最为简单，若要求噪声小或要求输出稳压时，可采用带 LDO 线性稳压器的电荷泵 IC。例如，MAX1680/1681，输出电流可达 125mA，采用 1MHz 开关频率，仅需外接两个 1μF 小电容，输出阻抗 3.5Ω，有关闭电源控制（关闭时耗电仅 1μA），并可组成倍压电路。另一种带稳压输出的电荷泵 IC MAX868，它输出可调（0～-2×VIN），外接两个 0.1μF 电容，消耗 35μA 电源电流，可输出 30mA 稳压的电流，有关闭电源控制功能（关闭时耗电仅 0.1μA），小尺寸μMAX 封装。

（4）不要追求高精度、功能全的最新器件

电源 IC 的精度一般为±2%～±4%，精度高的可达±0.5%～±1%，要根据电路的要求选择合适的精度，这样可降低生产成本。功能较全的器件价格较高，所以不需要关闭电源功能的或产品中无微处理器（μP）或微控制器（μC）的，则无需选择带关闭电源功能或输出电源工作状态信号的器件，这样不仅可降低成本，并且尺寸更小。

（5）不要“大马拉小车”

电源 IC 最主要的三个参数是，输入电压 VIN、输出电压 VOUT 及最大输出电流 Iomax。根据产品的工作电流来选择：较合适的是工作电流最大值为电源 IC 最大输出电流 Iomax 的 70～90%。例如最大输出电流 Iomax 为 1A 的升压式 DC/DC 变换器 IC 可用于工作电流 700～900mA 的场合，而工作于 20～30mA 时，其效率则较低。如果产品有轻负载或重负载时，最好选择 PFM/PWM 自动转换升压式 DC/DC 变换器，这不仅在轻负载时采用 PFM 方式耗电较小，正常负载时为 PWM 方式，而且效率也高。这种电源 IC 有 TC120、MAX1205/1706 等。

（6）输出电流大时应采用降压式 DC/DC 变换器

便携式电子产品大部分工作电流在 300mA 以下，并且大部分采用 5#镍镉、镍氢电池，若采用 1～2 节电池，升压到 3.3V 或 5V 并要求输出 500mA 以上电流时，电池寿命不长或两次充电间隔时间太短，使用不便。这时采用降压式 DC/DC 变换器，其效率与升压式差不多，但电池的寿命或充电间隔时间要长得多。

（7）DC/DC 变换器中 L、C、D 的选择

电感 L、输出电容 C 及续流二极管或隔离二极管 D 的选择十分重要。电感 L 要满足在开关电流峰值时不饱和（开关峰值电流要大于输出电流 3～4 倍），并且要选择合适的磁芯以满足开关频率的要求及选择直流电阻小的以减少损耗。电容应选择等效串联电阻小的电解电容（LOW ESR），这可降低输出纹波电压，采用三洋公司的有机半导体铝固体电解电容（一般为几十～几百毫欧）有较好效果。二极管必须采用肖特基二极管，并且要以满足大于峰值电流为要求。

1.2. 嵌入式系统硬件开发相关技术

1.2.1. 接口技术

CPU 与外部设备、存储器的连接和数据交换都需要通过接口设备来实现，前者被称为 I/O 接口，而后者则被称为存储器接口。存储器通常在 CPU 的同步控制下工作，接口电路比

较简单；而 I/O 设备品种繁多，其相应的接口电路也各不相同，因此，习惯上说到接口只是指 I/O 接口。

1.2.1.1. 并行接口

所谓“并行”，是指 8 位数据同时通过并行线进行传送，这样数据传送速度大大提高，但并行传送的线路长度受到限制，因为长度增加，干扰就会增加，容易出错。

表 1.5. 25 针并口功能一览表

针脚	功能	针脚	功能
1	选通端(STROBE)，低电平有效	10	确认(ACKNLG)，低电平有效
2	数据位 0(DATA0)	11	忙(BUSY)
3	数据位 1(DATA1)	12	缺纸(PE)
4	数据位 2(DATA2)	13	选择(SLCT)
5	数据位 3(DATA3)	14	自动换行(AUTO FEED)，低电平有效
6	数据位 4(DATA4)	15	错误(ERROR)，低电平有效
7	数据位 5(DATA5)	16	初始化(INIT 低电平),低电平有效
8	数据位 6(DATA6)	17	选择输入(SLCT IN 低电平)，低电平有效
9	数据位 7(DATA7)	18 到 25	地线(GND)

并口的工作模式主要有如下几种：

(1) SPP 标准工作模式。SPP 数据是半双工单向传输的，传输速率仅为 15Kb/s，速度较慢，但几乎可以支持所有的外设，一般设为默认的工作模式。

(2) EPP 增强型工作模式。EPP 采用双向半双工数据传输，其传输速度比 SPP 高，可达 2MB/s。EPP 可细分为 EPP1.7 和 EPP1.9 两种模式，目前较多外设使用此工作模式。

(3) ECP 扩充型工作模式。ECP 采用双向全双工数据传输，传输速率比 EPP 要高。

1.2.1.2. 串口

在嵌入式系统的开发和应用中，经常需要使用上位机实现系统的调试及现场数据的采集和控制。一般是通过上位机本身配置的串行口，通过串行通讯技术，和嵌入式系统进行连接通讯。

串行口的典型代表是 RS－232－C 及其兼容插口，25 针串行口还具有 20mA 电流环接口功能，用 9、11、18、25 针来实现。RS-232-C 是美国电子工业协会 EIA（Electronic Industry Association）制定的一种串行物理接口标准。RS 是英文“推荐标准”的缩写，232 为标识号，C 表示修改次数。RS-232-C 总线标准设有 25 条信号线，包括一个主通道和一个辅助通道，在多数情况下主要使用主通道，对于一般双工通信，仅需几条信号线就可实现，如一条发送线、一条接收线及一条地线。RS-232-C 标准规定的数据传输速率为每秒 50、75、 100、150、300、600、1200、2400、4800、9600、19200 波特。其针脚功能如表 1.6.和 1.7.所示。

一般微机提供标准的 RS232C 接口，该接口采用负逻辑，与 CMOS、TTL 电路的相连需要专用集成电路进行电平转换。一般应用情况下，RC232C 的最高传输速率为 20 kb/s, 最大传输线长为 30 米。相比较而言，它的传输速率低、传输距离近、抗共模干扰能力差，在条件较恶劣的现场控制中，很难实现数据的正常传输和获取。在要求通信距离为几十米到上千

米时，广泛采用 RS485 接口。RS485 采用差分接收和驱动，提高抗共模干扰驱动能力，并且提供多点应用，同一线上最多可接 32 个驱动器和接收器，最大传输速率 10Mb/s (12m)，最大传输距离为 1200m（10kb/s），可以较好的实现现场数据的获取和控制。

表 1.6.　25 针串口功能一览表

针脚	功能	针脚	功能
1	空	11	数据发送(－)
2	发送数据(TXD)	12 到 17	空
3	接收数据(RXD)	18	数据接收(＋)
4	发送请求(RTS)	19	空
5	发送清除(CTS)	20	数据终端准备好(DTR)
6	数据准备好(DSR)	21	空
7	信号地线(SG)	22	振铃指示(RI)
8	载波检测(DCD)	23 到 24	空
9	发送返回（＋)	25	接收返回(一)
10	空		

表 1.7.　9 针串口功能一览表

针脚	功能	针脚	功能
1	载波检测(DCD)	6	数据准备好(DSR)
2	接收数据(RXD)	7	发送请求(RTS)
3	发送数据(TXD)	8	发送清除(CTS)
4	数据终端准备好(DTR)	9	振铃指示(RI)
5	信号地线(SG)		

1.2.1.3. USB

USB全称Universal Serial Bus（通用串行总线）。USB接口是现在比较流行的接口，用于将使用USB的外围设备连接到主机。在USB 的网络协议中，每个USB 的系统有且只有一个host， 它负责管理整个USB系统，包括USB Device 的连接与删除、Host 与USB Device 的通信、总路线的控制等等。Host端有一个Root Hub，可提供一个或多个USB 下行端口。每个端口可以连接一个USB Hub或一个USB Device。USB Hub 是用于USB 端口扩展的，即USB Hub 可以将一个USB端口扩展为多个端口。

表 1.8.　USB 的针脚定义

针脚	功能
1	+5 伏电压（VCC）
2	数据通道（-DATA）
3	数据通道（+DATA）
4	地线(GND)

USB 最大的好处在于能支持多达 127 个外设，并且可以独立供电。普通的串、并口外设都要额外的供电电源，而 USB 接口可以从主机上获得 500mA 的电流，并且支持热拔插，真正做到即插即用。一个 USB 接口可同时支持高速和低速 USB 外设的访问，由一条 4 芯电缆连接，其中 2 条是正负电源，传送的是 5V 的电源，2 条是数据传输线，数据线是单工的，在整个的一个系统中的数据速率是一定的，要么是高速，要么是低速。高速外设的传输速率

为 12Mbps，而低速外设的传输速率先 1.5Mbps。新出台的 USB2.0 标准的最高传输速率可达 480Mbps，是目前 USB1.1 标准的 40 倍。

1.2.1.4. PCMCIA 和 CF

- PCMCIA 标准

PCMCIA 全名为 Personal Computer Memory Card International Association，中文意思是"国际个人电脑存储卡协会"。凡符合此协会定义的界面规定技术所设计的界面卡，便可称为 PCMCIA 卡或简称为 PC 卡。以前这项技术标准只适用于存储器扩充卡，但后来还扩展到存储器以外的外部设备，如网络卡、视频会议卡及调制解调器等。

PCMCIA 卡共分成四种规格，分别是 TYPEⅠ、TYPEⅡ、TYPEⅢ及 CardBus。由于 CardBus 属于需要高频宽外设的界面规格，而且不常见，这里集中介绍前三类规格，即 TYPEⅠ、TYPEⅡ及 TYPEⅢ，它们常被应用于一般的外设规格上。TYPEⅠ的规格：面积为 8.56×5.4cm，厚度则为 0.33cm；适用于一般存储器扩充卡。TYPEⅡ的规格：面积为 8.56×5.4cm，厚度则为 0.5cm；应用范围包括 Modem 卡、Network 卡、视频会议卡等。TYPEⅢ的规格：面积为 8.56×5.4cm，厚度为 1.05cm；应用范围为硬盘。

从外观上看，这三种 PCMCIA 卡的尺寸都是 8.56cm×5.4cm，其实，它们的区别在卡的厚度。TYPEⅠ的厚度最薄，最适合用于存储器扩充卡；TYPEⅡ则常用于数据传输、网络连接等产品，所以 Modem 卡和 Network 卡都是 TYPEⅡ规格的；TYPEⅢ方面，因为较厚，所以它适合取代机械式的储存媒体，如硬盘。

PCMCIA 卡除了轻巧、方便携带外，它有个和 USB (Universal Serial Bus)外设相同的特色，就是"热插拔"(Hot Plugging)功能。所以 PCMCIA 规格的设备可于电脑开机状态时安装插入，并能自动通知操作系统作设备的更新，省去不少安装的麻烦。

- Compact Flash 标准

90 年代初，当消费性数码电子产品尚在研制时，Sandisk 和 Canon（佳能）等几家公司就洞悉到急需新的存储介质与之相适应，通过业界的沟通，Sandisk 和 KODAK（柯达）、CASIO（卡西欧）、Canon（佳能）结成战略性伙伴，制定新一代的基于 RAM 和 ROM 技术的固态非易掉失的存储介质标准：Compact Flash 标准。到 1994 年，Sandisk 推出第一块可抹写的 CF 卡（属于 EPROM）。随后，在 1995 年，由 125 家厂商联盟组成一个非盈利性质的，旨在共同推广 CF 标准的协会——CompactFlash Association（简称 CFA）。

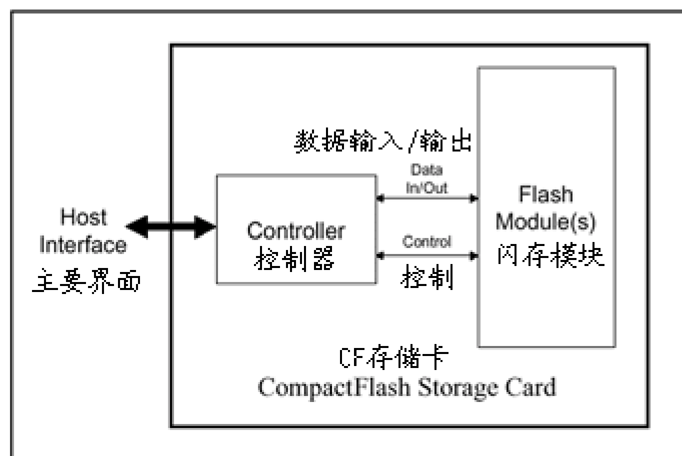


图 1.5. CF 卡的工作原理图

CF 卡分 2 种，TYPE I 为 43mm ×36mm ×3.3mm（CF I），TYPE II 为 43mm ×36mm ×5mm（CF II），其厚度还不到目前的 PCMCIA TypeII 卡的一半，体积只有 PCMCIA 卡的 1/4，给大家的感觉就像是 PCMCIA 卡的缩小版。但要注意的是 CF I 的卡槽较窄不能兼容 CF II 卡，而 CF II 卡槽则可兼容 CF I 卡。CF 卡遵从 ATA-IDE 工业设计标准，CF 卡的连接装置与

PCMCIA 卡相似，只不过 CF 卡是 50-pin（PCMCIA 卡 68-pin）， CF 卡可以很容易的插入无源 68-pin TypeII 适配卡并完全符合 PCMCIA 电力和机械接口规格。CompactFlash 卡同时支持 3.3 伏和 5 伏的电压，我们知道大部份的数字集成电路的供电要么是 5V 要么 3.3V。CF 卡为方便适应不同数字集电路，可以在电压为 3.3V-5V 间运行，这也就意味着你可以选择其范围内的任一电压，CF 卡的兼容性还表现在它把 Flash Memory 存贮模块与控制器结合在一起，这样使用 CF 卡的外部设备就可以做得比较简单，而且不同的 CF 卡都可以用单一的机构来读写，不用担心兼容性问题，

1.2.1.5. 红外线接口

由于利用红外线接口进行文件传输不用连线，且速度较快，达 4M/s，不失为短距离双机通讯的一种好方法。进行红外线通讯时需注意：将具有红外线通讯功能的两个系统靠近，且发送口大致在同一水平线上，注意两系统之间的距离不能相差太远，一般在一到两米，角度相差不超过 30 度。

红外线通讯是一种廉价、近距离、无连线、低功耗和保密性较强的通讯方案，原来主要应用在无线数据传输方面，但目前已经逐渐开始在无线网络接入和近距离遥控家电方面得到应用。

红外线接口大多是一个 5 针插座，其管脚定义如下：

表 1.9. 红外线接口 5 针插座管脚定义

针脚	功能
1	IRTX（Infrared Transmit，红外传输）
2	GND（电源地线）
3	IRRX（Infrared Receive，红外接收）
4	NC（未定义）
5	VCC（电源正极）

IRDA （Infrared Data Association，红外数据协会）提供的红外通讯电路标准方案如图 1.6.所示。

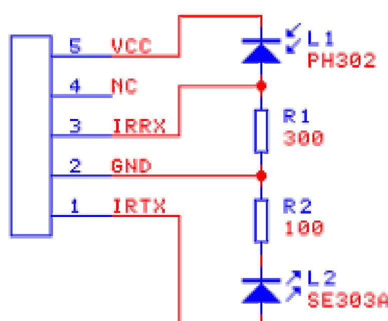


图 1.6. 红外通讯电路标准方案

红外发射电路由红外线发射管 L2 和限流电阻 R2 组成。当主板红外接口的输出端 IRTX 输出调制后的电脉冲信号时，红外线发射管将电脉冲信号转化为红外线光信号发射出去。电阻 R2 起限制电流的作用，以免过大的电流将红外管损坏。当 R2 的阻值越小，通过红外管的电流就越大，红外管的发射功率也随电流的增大而增大，发射距离就越远，但 R2 的阻值不能过小，否则会损坏红外管或主板红外接口！

红外接收电路由红外线接收管 L1 和取样电阻 R1 组成。当红外接收管接收到红外线光信号时，其反向电阻会随光信号的强弱变化而相应变化，根据欧姆定律可以得知通过红外接收

管 L1 和电阻 R1 的电流也会相应变化，而在取样电阻两端的电压也随之变化，此变化的电压经主板红外接口的输入端 IRRX 输入主机。由于不同的红外接收管的电气参数不同，所以取样电阻 R1 的阻值要根据实际情况作一定范围的调整。

1.2.2. 总线

总线就是各种信号线的集合，是计算机各部件之间传送数据、地址和控制信息的公共通路。总线的主要参数有：

- 总线的带宽

总线的带宽指的是一定时间内总线上可传送的数据量，即我们常说的每秒钟传送多少 MB 的最大稳态数据传输率。与总线带宽密切相关的两个概念是总线的位宽和总线的工作时钟频率。

- 总线的位宽

总线的位宽指的是总线能同时传送的数据位数，即我们常说的 32 位、64 位等总线宽度的概念。总线的位宽越宽则总线每秒数据传输率越大，也即总线带宽越宽。

- 总线的工作时钟频率

总线的工作时钟频率以 MHz 为单位，工作频率越高则总线工作速度越快，也即总线带宽越宽。

1.2.2.1. ISA

IBM 公司于 1981 年推出的基于 8 位机 PC/XT 的总线，称为 PC 总线。1984 年 IBM 公司推出了 16 位 PC 机 PC/AT，其总线称为 AT 总线。然而 IBM 公司从未公布过他们的 AT 总线规格。为了能够合理地开发外插接口卡，由 Intel 公司，IEEE 和 EISA 集团联合开发了与 IBM/AT 原装机总线意义相近的 ISA 总线，即 8/16 位的“工业标准结构”(ISA-Industry Standard Architecture)总线。

ISA 总线有 98 只引脚。其中 62 线的一段基于 8 位的 PC 总线，可以独立使用，连接 8 位的扩展卡，而 62 线与 36 线相加后就扩展成标准的 16 位 ISA，连接 16 位的扩展卡。

ISA 总线的主要性能指标如下：

（1） I/O 地址空间 0100H-03FFH
（2） 24 位地址线可直接寻址的内存容量为 16MB
（3） 8/16 位数据线
（4） 62+36 引脚
（5） 最大位宽 16 位(bit)
（6） 最高时钟频率 8MHz
（7） 最大稳态传输率 16MB/s
（8） 中断功能
（9） DMA 通道功能
（10） 开放式总线结构，允许多个 CPU 共享系统资源

1.2.2.2. PCI

1991 年下半年，Intel 公司首先提出了 PCI 的概念，并联合 IBM、Compaq、AST、HP、DEC 等 100 多家公司成立了 PCI 集团，其英文全称为：Peripheral Component Interconnect Special Interest Group(外围部件互连专业组)，简称 PCISIG。PCI 有 32 位和 64 位两种，32 位 PCI 有 124 引脚，64 位有 188 引脚，目前常用的是 32 位 PCI。32 位 PCI 的数据传输率为 133MB／s，大大高于 ISA。

PCI 总线的主要性能

（1） 支持 10 台外设
（2） 总线时钟频率 33.3MHz/66MHz
（3） 最大数据传输速率 133MB/s
（4） 时钟同步方式
（5） 与 CPU 及时钟频率无关
（6） 总线宽度 32 位（5V）/64 位（3.3V）
（7） 能自动识别外设

1.2.2.3. I2C 总线

在现代电子系统中，有为数众多的 IC 需要进行相互之间以及与外界的通信。为了提供硬件的效率和简化电路的设计，PHILIPS 开发了一种用于内部 IC 控制的简单的双向两线串行总线 I^2C(inter IC 总线)。I^2C 总线支持任何一种 IC 制造工艺，并且 PHILIPS 和其他厂商提供了种类非常丰富的 I^2C 兼容芯片。作为一个专利的控制总线，I^2C 已经成为世界性的工业标准。

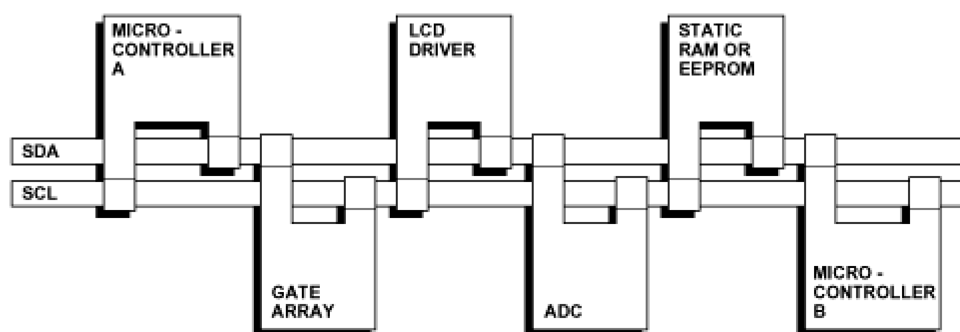


图 1.7. I^2C 总线示意图

每个 I^2C 器件都有一个唯一的地址，而且可以是单接收的器件（例如：LCD 驱动器）或者可以接收也可以发送的器件（例如：存储器）。发送器或接收器可以在主模式或从模式下操作，这取决于芯片是否必须启动数据的传输还是仅仅被寻址。I^2C 是一个多主总线，即它可以由多个连接的器件控制。

早期的 I^2C 总线数据传输速率最高为 100Kbits/s，采用 7 位寻址。但是由于数据传输速率和应用功能的迅速增加，I^2C 总线也增强为快速模式（400Kbits/s）和 10 位寻址以满足更高速度和更大寻址空间的需求。

I^2C 总线始终和先进技术保持同步，但仍然保持其向下兼容性。并且最近还增加了高速模式，其速度可达 3.4Mbits/s。它使得 I^2C 总线能够支持现有以及将来的高速串行传输应用，例如 EEPROM 和 Flash 存储器。

1.2.2.4. SPI 总线

串行外围设备接口 SPI（serial peripheral interface）总线技术是 Motorola 公司推出的一种同步串行接口。Motorola 公司生产的绝大多数 MCU（微控制器）都配有 SPI 硬件接口，

如 68 系列 MCU。SPI 总线是一种三线同步总线，因其硬件功能很强，所以，与 SPI 有关的软件就相当简单，使 CPU 有更多的时间处理其他事务。

由 SPI 连成的串行总线是一种三线同步总线，总线上可以连接多个可作为主机的 MCU，装有 SPI 接口的输出设备，输入设备如液晶驱动、A/D 转换等外设，也可以简单连接到单个 TTL 移位寄存器的芯片。总线上允许连接多个能作主机的设备，但在任一瞬间只允许有一个设备作为主机。总线的时钟线 SCK 由主机控制，另外两根分别是：主机输入/从机输出线 MISO 和主机输出/从机输入线 MOSI。典型的结构如图 1.8.所示。

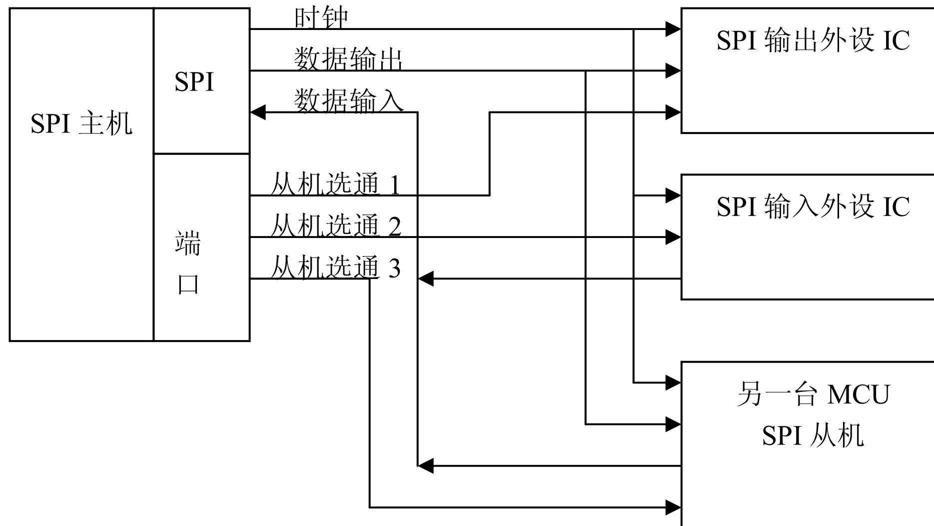


图 1.8.　SPI 总线示意图

系统可以简单，也可以复杂，主要有以下几种形式：

（1）一台主机 MCU 和若干台从机 MCU。

（2）多台 MCU 互相连接成一个多主机系统。

（3）一台主机 MCU 和若干台从机外围设备。

主机和哪台从机通讯通过各从机的选通线进行选择。

SPI 是全双工的，即主机在发送的同时也在接收数据，传送的速率由主机编程决定；时钟的极性和相位也是可以选择的，具体的约定由设计人员根据总线上各设备接口的功能决定。

1.2.2.5. PC104 总线

1992 年 IEEE 开始着手为 PC 和 PC/AT 总线制定一个精简的 IEEEP996 标准（草稿），PC104 作为基本文件被采纳，叫做 IEEE P996.1 兼容 PC 嵌入式模块标准。可见，PC104 是一种专门为嵌入式控制而定义的工业控制总线。我们知道 IEEE-P996 是 PC 和 PC/AT 工业总线规范，IEEE 协会将它定义 IEEE-P996.1，很明显 PC104 实质上就是一种紧凑型的 IEEE-P996，其信号定义和 PC/AT 基本一致，但电气和机械规范却完全不同，是一种优化的、小型、堆栈式结构的嵌入式控制系统总线。

PC104 有两个版本，8 位和 16 位，分别与 PC 和 PC/AT 相对应。PC104PLUS 则与 PCI 总线相对应，在 PC104 总线的两个版本中，8 位 PC104 共有 64 个总线管脚，单列双排插针和插孔，P1：64 针，P2：40 针，合计 104 个总线信号，PC104 因此得名。当 8 位模块和 16 位模块连接时，16 位模块必须在 8 位模块得下面。P2 总线连结在 8-位元模块中是可选的。

PC104PLUS 是专为 PCI 总线设计的，可以连接高速外接设备。PC104PLUS 在硬件

上通过一个 3×40 即 120 孔插座连接，PC104PLUS 包括了 PCI 规范 2.1 版要求的所有信号。为了向下兼容，PC104PLUS 保持了 PC104 的所有特性。因此 PC104PLUS 规范包含了两种总线标准：ISA 和 PCI，可以双总线并存。

PC104PLUS 与 PC104 相比有以下 3 个特点：

（1）相对 PC/104 连接，增加了第三个连结接口支持 PCI 总线

（2）改变了组件高度的需求，增加模块的柔韧性

（3）加入了控制逻辑单元，以满足高速总线的需求

1.2.2.6. CAN 总线

CAN，全称为“Controller Area Network”，即控制器局域网，是国际上应用最广泛的现场总线之一。起先，CAN 被设计作为汽车环境中的微控制器通讯，在车载各电子控制装置 ECU 之间交换信息，形成汽车电子控制网络。比如：发动机管理系统、变速箱控制器、仪表装备、电子主干系统中，均嵌入 CAN 控制装置。CAN 是一种多主方式的串行通讯总线，基本设计规范要求有高的位速率，高抗电磁干扰性，而且能够检测出产生的任何错误。当信号传输距离达到 10Km 时，CAN 仍可提供高达 50Kbit/s 的数据传输速率。

为促进 CAN 以及 CAN 协议的发展，1992 在欧洲成立了 CiA（CAN in Automation）。在 CiA 的努力推广下，CAN 技术在汽车电控制系统、电梯控制系统、安全监控系统、医疗仪器、纺织机械、船舶运输等方面均得到了广泛的应用。现已有 400 多家公司加入了 CiA，CiA 已经为全球应用 CAN 技术的权威。

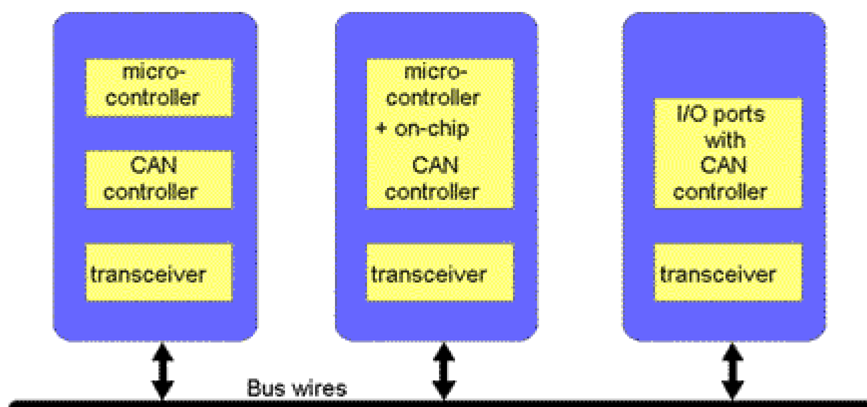


图 1.9.　CAN 总线示意图

CAN 的主要特性

（1）低成本；

（2）极高的总线利用率；

（3）很远的数据传输距离（长达 10Km）；

（4）高速的数据传输速率（高达 1Mbit/s）；

（5）可根据报文的 ID 决定接收或屏蔽该报文；

（6）可靠的错误处理和检错机制；

（7）发送的信息遭到破坏后，可自动重发；

（8）节点在错误严重的情况下具有自动退出总线的功能；

（9）报文不包含源地址或目标地址，仅用标志符来指示功能信息、优先级信息。

1.2.3. 嵌入式系统开发常用的硬件调试和编程技术

1.2.3.1. 微代码支持的串口调试

传统上，首先用于开发嵌入式系统的工具是内部电路仿真器（ICE），它是一个相对昂贵的部件，用于植入微处理器与总线之间的电路中，允许使用者监视和控制微处理器所有信号的进出。它提供了总线工作的清晰情况，避免了许多对硬件软件底层工作状况的猜测。

过去，一些工作依赖 ICE 为主要调试工具，用于整个开发过程。但是，一旦初始化软件对串口支持良好的话，多数的调试可以不用 ICE 而直接使用串口开始调试。下面以 EZ328 的 Bootstrap 模式为例加以说明。

EZ328 支持 Bootstrap 模式，允许用户通过 UART 控制器初始化目标板和下载程序到目标板的 RAM 和 FLASH 中。一旦程序被下载，就能够被执行，给了用户一个简单的用于故障分析的调试环境和更新 FLASH 存储器中程序的途径。

在 bootstrap 模式下，MC68EZ328 的 UART 控制器被初始化到 9600baud，无奇偶校验，8 位字符和 1 个停止位，准备下载程序或者数据。为了下载数据或者程序，你必须将代码转换成 bootstrap 格式文件，是一种包含 bootstrap 记录的 text 文件。

在你下载程序到系统储存器中之前，你要使用 EZ328 的内部寄存器来初始化目标系统。这些内部寄存器可以被作为存储器，使用 bootstrap 记录来写入相应的内容。

bootstrap 的设计提供了一个 8 字节长的指令缓存，用户可以向其中下载 68000 的指令。这个特点使得用户可以在存储系统无效或者单 MPU 系统的情况下运行 68000 指令。该指令缓存从 0xFFFFAA 开始。

Bootstrap 模式是 MC68EZ328 的三种操作模式（常规，仿真和 bootstrap）中的一种。它具有最高的优先级。为了进入 bootstrap 模式，用户必须驱动/EMUBRK 信号为低电平并执行系统复位。复位后，bootstrap 复位向量会在内部产生。这些两个字长的复位向量被装载到内核的堆栈指针和程序计数器中。然后内建的 bootstrap 程序将开始运行并接收数据传送。

1.2.3.2. 编程技术

在正式调试最小系统之前，必须做的一件事就是将调试程序的目标板部分烧写到系统的 FLASH ROM 中。烧写的方法有几种：

(1) 使用编程器，就是在芯片焊接之前，先通过编程器将代码烧写道 FLASH 中，再将 FLASH 芯片焊接到目标板上。使用编程器进行编程特别适合于 DIP 封装芯片的编程，如果是其它类型的封装，则必须要使用相应的适配器。这种方法的缺点是需要手工进行待编程芯片的插入、锁定等工作，容易造成芯片方向错误，引脚错位等，导致编程效率降低。

(2) 使用板上编程器编程（OBP），这种方法是在板上所有芯片包括 FLASH 芯片已经焊装完毕之后，再对可编程芯片进行编程。通过专用电缆将电路板与外部计算机连接，由计算机的应用程序进行板上可编程芯片的代码或数据写入，芯片擦除、编程所需要的电压、控制信号、地址数据和相关命令都由板外的编程控制器提供。使用板上编程器进行板上编程时，需要关断目标板上 CPU 的电源或将其外部接口信号设置为高阻状态，以免与编程时的地址、数据和控制信号发生冲突。这种方法的缺点是需要在电路板上设计编程用的接口、隔离等辅助电路，在编程时通过跳线或 FET 开关进行编程与正常工作的状态转换。这样会增加每个电路板芯片的数量，造成产品成本的增加。

(3) 在系统编程（ISP、ISW），是指直接利用系统中带有 JTAG 接口的器件，如 CPU、CPLD、FPGA 等，执行对系统中程序存储器芯片内容的擦除和编程操作。一般而言，高档微处理器均带有 JTAG 接口，系统程序存储器的数据总线、地址总线和控

制信号直接接在微处理器上。编程时，使用 PC 机内插卡或并行接口通过专用电缆将系统电路板与 PC 机联系起来，在 PC 机上运行相关程序，将编程数据及控制信号传送到 JTAG 接口的芯片，再利用相应指令从微处理器的引脚按照 FLASH 芯片的编程时序输出到 FLASH 存储器。这种编程方法的条件是系统中必须存在带有 JTAG 接口或与之兼容的芯片，如微处理器。优点是系统板上不需要增加其它与编程有关的附属电路，减小了电路板的尺寸，同时，避免了对微小封装芯片的手工处理，特别适用于电路板尺寸有严格限制的手持设备。缺点是编程速度慢，对于代码长度小的编程比较适合。

1.2.3.3. JATG 与 IEEE1149 协议简介

JTAG 是英文"Joint Test Action Group（联合测试行为组织）"的词头字母的简写，该组织成立于 1985 年，是由几家主要的电子制造商发起制订的 PCB 和 IC 测试标准。JTAG 建议于 1990 年被 IEEE 批准为 IEEE1149.1-1990 测试访问端口和边界扫描结构标准。该标准规定了进行边界扫描所需要的硬件和软件。自从 1990 年批准后，IEEE 分别于 1993 年和 1995 年对该标准作了补充，形成了现在使用的 IEEE1149.1a-1993 和 IEEE1149.1b-1994。JTAG 主要应用于：电路的边界扫描测试和可编程芯片的在系统编程。

电路的边界扫描测试技术:用具有边界扫描功能的芯片构成的印刷板,可通过相应的测试设备,检测已安装在印刷板上的芯片的功能，检测印刷板连线的正确性，同时，可以方便地检测该印刷板是否具有预定的逻辑功能,进而对由这种印刷板构成的数字电气装置进行故障检测和故障定位。在系统编程在上节已经提过，这里不再重复。

在硬件结构上，JTAG 接口包括两部分：JTAG 端口和控制器。与 JTAG 接口兼容的器件可以是微处理器（MPU）、微控制器（MCU）、PLD、CPL、FPGA、ASIC 或其它符合 IEEE1149.1 规范的芯片。IEEE1149.1 标准中规定对应于数字集成电路芯片的每个引脚都设有一个移位寄存单元，称为边界扫描单元 BSC。它将 JTAG 电路与内核逻辑电路联系起来，同时隔离内核逻辑电路和芯片引脚。由集成电路的所有边界扫描单元构成边界扫描寄存器 BSR。边界扫描寄存器电路仅在进行 JTAG 测试时有效，在集成电路正常工作时无效，不影响集成电路的功能。具有 JTAG 接口的芯片内部结构如图 1.10.所示。

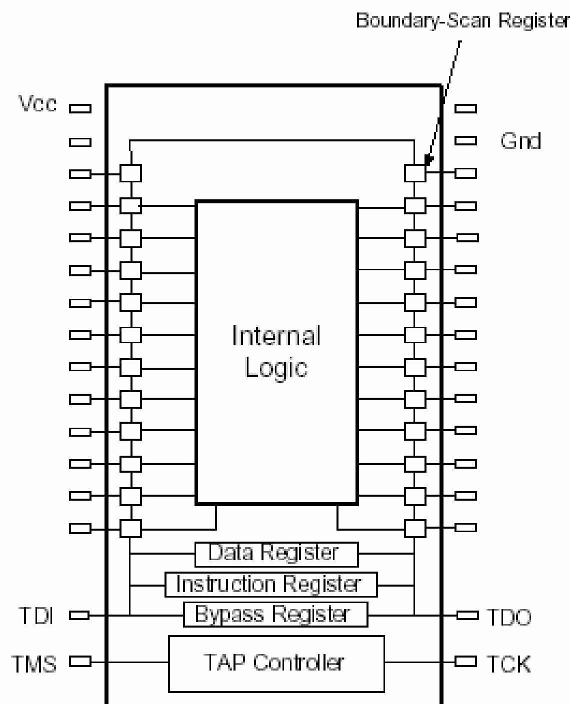


图 1.10. JTAG 接口芯片内部结构图

在对多个具有 JTAG 芯片编程时，可以组成 JTAG 菊花链结构（Daisy chain）,是一种特殊的串行编程方式。每片 TDI 输入端与前面一片的 TDO 输出端相连，最前面一片的 TDI 端和最后一片的 TDO 端与 JTAG 编程接口的 TDI、TDO 分别相连。如图 1.11.所示。链中的器件数可以很多，只要不超出接口的驱动能力即可。通过状态机控制，可以使非正在被编程器件的 TDI 端直通 TDO 端，这样就可以使数据流形成环路，对各器件按序进行编程。使用者可以通过读取每个芯片特有的识别码知道该器件在链中的位置。

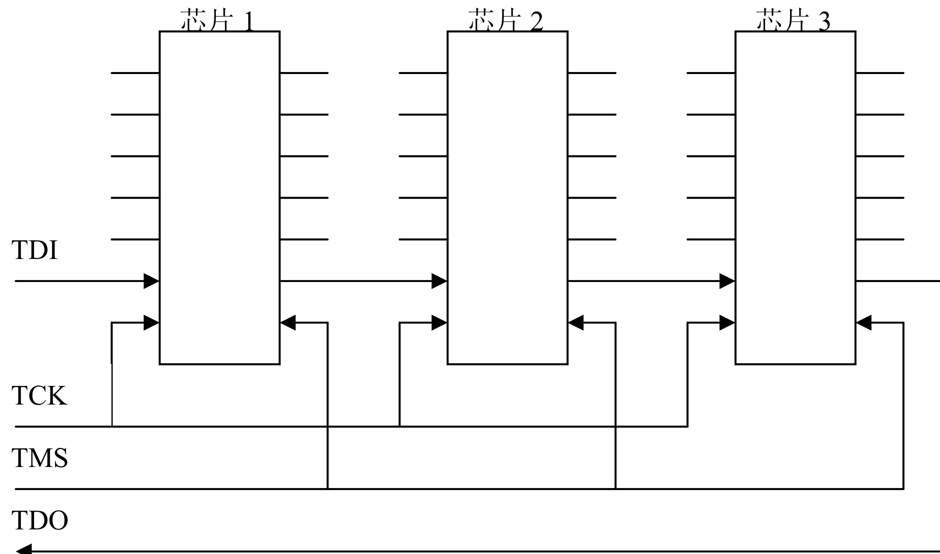


图 1.11.　JTAG 菊花链结构图

（1）指令寄存器 IR：由两个或更多个指令寄存单元和指令译码器组成，通过它可以串行输入执行各种操作的指令。
（2）数据寄存器组：是一组基于电路的移位寄存器。操作指令被串行装入由当前指令所选择的数据寄存器。随着操作的执行，测试结果被移出。
（3）边界寄存器 DR：在内部逻辑电路和各引脚之间均插入了一串边界扫描单元，形成了由 TDI 到 TDO 之间的边界寄存器链
（4）旁路寄存器 BP：它只是 1 位寄存器。它的一端与 TDI 相连，另一端与 TDO 相连。在指令控制下，由 TDI 输入的数据可以直接经由本片的旁路寄存器送到 TDO。使用旁路寄存器，可以越过片 1、片 2 的边界寄存器，仅经过它们的旁路寄存器直接向片 3 输入数据。
（5）测试访问端口（TAP）控制器：TAP 控制器是一个 16 状态的莫尔型同步时序电路，响应于测试时钟 TCK 的上升沿。在 TCK 和 TMS 协同配合下确定来自 TDI 的串行数据是指令码还是测试码，进而产生 ClockIR、ClockIR、UpdateIR、UpdateDR、ShiftDR 和 Mode、Control 等信号，实现对 IR 和 DR 的设置和控制。
（6）测试总线：这种芯片至少有四个供边界扫描用的附加引脚 TCK、TMS、TDI 和 TDO，还可以另设一个引脚 TRST。TCK 是测试时钟输入引脚，TMS 是测试方式选择引脚，TDI 是测试用输入引脚，TDO 是测试用输出引脚。这四个引脚构成了测试总线。TRST 是供 TAP 控制器复位用的。

1.2.4. 3.3V 和 5V 装置的互连

连接 3.3V 设备到 5V 设备需要考虑到驱动器和接收器的逻辑电平是否匹配。图 1.12.描述了用于 5V CMOS，5V TTL 和 3.3V TTL 的逻辑电平标准。可以看到，5V TTL 和 3.3V TTL 的逻辑电平是相同的，而 5V CMOS 逻辑电平与前两者是不同的。这在连接 3.3V 系统到 5V 系统时是必须考虑的。

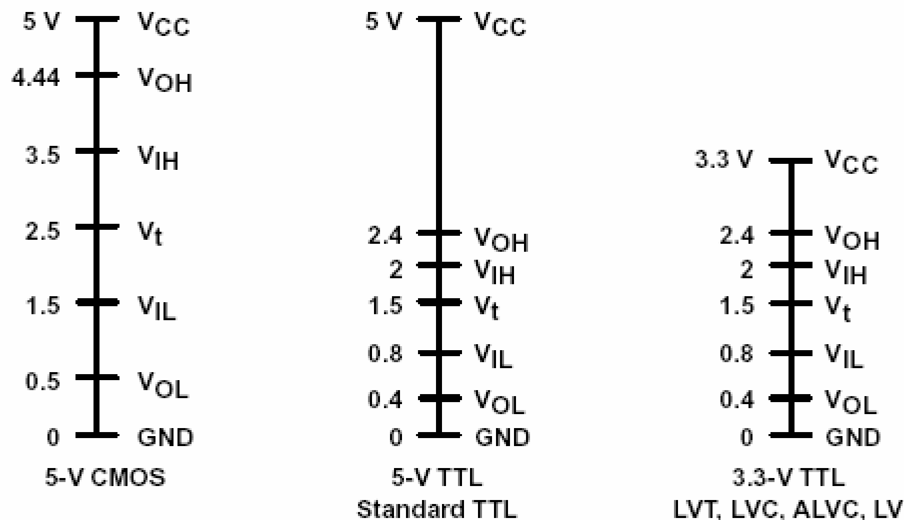


图 1.12. 5V CMOS，5V TTL 和 3.3V TTL 开关电平标准

（1）5V TTL 装置驱动 3.3V TTL 装置。5V TTL 和 3.3V LVC 的逻辑电平是相同的。因为 5V 容忍度的装置可以经受住 6.5V 的直流输入，所以 5V TTL 连接 3.3V 且容忍度为 5V 的装置时，可以不需要额外的元器件。TI 的 CBT（crossbar technology）开关可以用来从 5V TTL 向 3.3V 且容忍度不为 5V 的装置传送信号。该开关通过使用一个外部的产生 0.7V 压降的二极管和 CBT（门极到源极的压降为 1V），从而产生 3.3V 的电平。

（2）3V TTL 装置（LVC）驱动 5V TTL 装置。两者逻辑电平是相同的，连接可以不需要外部电路或装置。

（3）5V COMS 装置驱动 3.3V TTL 装置。两个不同的逻辑电平连接在一起，进一步分析 5V COMS 装置的 VOH 和 VOL 与 3.3V LVC 装置的 VIH 和 VIL 电平，虽然存在不一致的地方，但有 5V 容忍度的 3.3V 装置可以在 5V CMOS 电平输入下工作。使用 5V 容忍度的 LVC 装置，5V CMOS 驱动 3.3V LVC 是可能的。

（4）3.3V TTL 装置驱动 5V CMOS 装置。3.3V LVC 的 VOH 是 2.4V（输出电平可达 3.3 V），而 5V CMOS 装置的最小 VIH 要求是 3.5V。因此，用 3.3V LVC 或其他 3.3V 标准的装置驱动 5V CMOS 装置是不可能的。解决该问题就需要用到专用芯片，如 TI 的 SN74ALVC164245 和 SN74LVC4245 等。这些芯片一边采用 3.3V 电平供电，另一边采用 5V 电平供电，可以使 3.3V 逻辑部分驱动 5V CMOS 装置。

1.3. 嵌入式系统开发示例——EZ 开发板

1.3.1. 系统性能

（1）主处理器：

采用 EZ328 CPU，20MHz，3.3V 供电，3.4MIPS，24 addr/16 data，100 TQFP 封装

（2）外围存储器：

8MByte EDO RAM

4MByte FLASH

（3）调试接口：

一个 RS232 接口

板上 9 个备用按键，已与处理器中断引脚连接

一个电源拨动开关，一个系统操作模式选择拨动开关

板上留有一个 34pin 和一个 40pin 双排过孔焊盘，方便系统外围设备接入和信号分析

（4）LCD 液晶和触摸屏接口：

留有 20 针标准点阵 LCD 和触摸屏接口

板载 12 位触摸屏采样芯片

（5）LED 信号指示灯：

1 个系统板良好指示灯

1 个电源指示灯

2 个串口通讯数据上传与下行指示灯

2 个网络连接与通讯指示灯

9 个中断输入指示灯

（6）10M 以太网接口

（7）3V 适配器或者 2 节 5 号电池供电

（8）开发板所支持的操作模式

EZ328 正常模式

EZ328 的 bootstrap 模式

（9）调试环境

台式机平台为 linux

提供 uclinux 内核、GCC 环境

1.3.2. 系统硬件设计

1.3.2.1. CPU 与存储器模块

EZ 的 DRAM 控制器支持两个 bank 的 DRAM，每个 bank 最多为 4MB，可以有 256K×16、512K×8、512K×16、1M×8、1M×16、4M×8 和 4M×16 DRAM 几种形式。通过 EZ 的\CSD0 和\CSD1 两个引脚来选择两个 bank。同时提供行地址触发\RAS 和列地址触发\CAS 信号。我们选择使用 4M×16bit EDO DRAM 芯片 MT4LC4M16R6TG-5，10 位列地址线，12 位行地址线。其\RAS 接 EZ 的\RAS0，\CASL 和\CASH 分别接 EZ 的\CAS1 和\CAS0。FLASH 选用 FUJISTU 公司的 MBM29LV160TE-70TN，3.3V 供电。单片容量为 2M×8bit/1M×16bit，使用两片，组成系统 4M 的 FLASH 闪存。选择单片 1M×16bit 的格式时，一片的\CE 连接 EZ 的\CSA0，另一片的连接 EZ 的\CSA1，两片的\WE 都接 EZ 的\UWE；

1.3.2.2. LCD 显示模块

EZ328 的 LCD 控制器可以支持在黑白显示模式下 640×512 和灰度显示模式下 320×240 的单屏单色 STN LCD。

EZ328 LCD 控制器中包含的 STN LCD 面板的标准信号如下：

表 1.10. STN LCD 控制信号列表

LD[3:0]	LCD 数据总线。用来传输要在 LCD 上显示的象素点数据
LFLM	LCD 帧标识信号。用来指示新一帧显示的开始。LFLM 在一帧的第一个行脉冲后有效，并保持有效直到第二个行脉冲。紧接着变为无效并保持到下一帧
LLP	LCD 行脉冲信号。用来锁存一行数据到 LCD 面板上。当一行象素点数据被送到 LCD 时有效，然后在 8 个象素时钟周期内无效
LCLK	LCD 移位时钟信号。输出数据到 LCD 面板上与该时钟同步
LACD	LCD 交替晶体方向，用来控制液晶方向的交替

LCD 采用了清华液晶公司的 THMG320240A，电压 VCC 范围是 2.7-3.3V，输入的高电平最小值是 0.8×VCC。可见 EZ 和该块 LCD 可以直接连接。对比度调节电路如下：

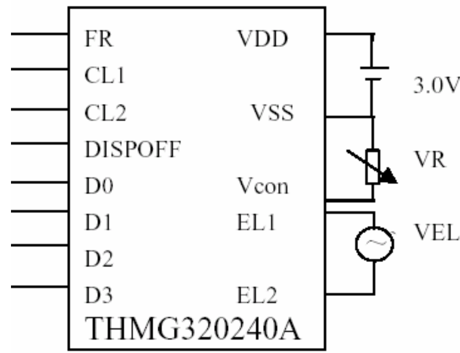


图 1.13.　LCD 对比度调节电路

1.3.2.3. 串口模块

使用 MAXIM 公司的 MAX3232，工作电压为＋3.3V。

1.3.2.4. 电源模块

选用 MAX1677，输入电压范围为＋0.7-＋5.5V，产生＋3.3V 和+5V 电压输出，其中＋5V 电压备用。

相关电阻值有如下计算公式：

R1 = R2 [(VOUT / 1.25V) - 1]，VOUT 取＋5V，R2 在 10k$_{\Omega}$－200kΩ之间取值。最后选择 R2 和 R1 分别为 20k$_{\Omega}$和 62k$_{\Omega}$。

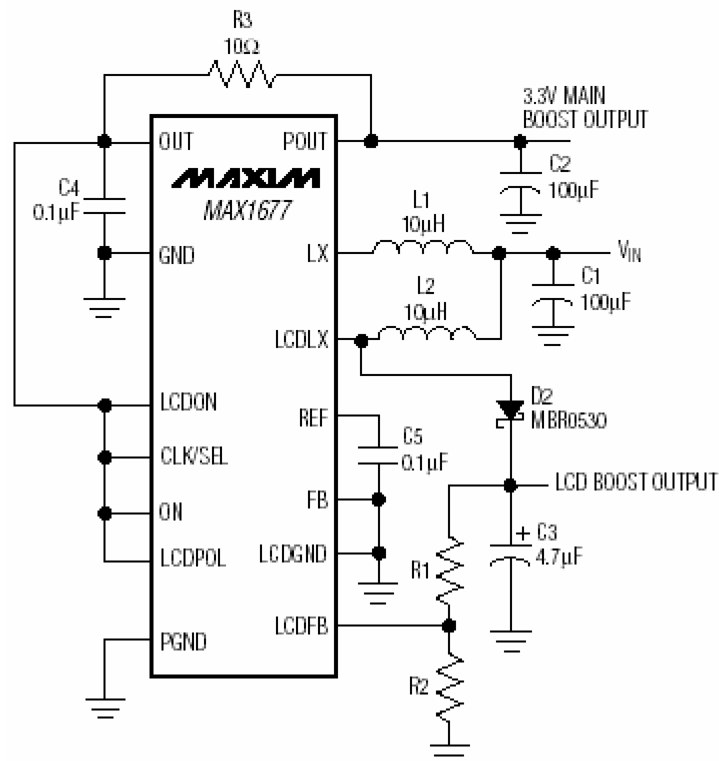


图 1.14.　电源模块电路

1.3.2.5. 进入 BOOTSTRAP 电路模块

开发板需要使用到 EZ328 的 BOOTSTRAP 模式，因此要有相关的电路保证。只要驱动 EZ328 的/EMUBRK 引脚为低电平并执行系统复位，就可使 EZ328 进入 bootstrap 模式，其内建的 bootstrap 程序就会开始运行，初始化芯片上集成的 UART 控制器并准备接收数据。这样开发者就可以通过 UART 控制器写寄存器来初始化开发板，然后下载程序到开发板的 RAM 和 FLASH 中运行调试。

在 RESET 部分，使用了六路施密特触发器 74HC14 和六路反相器 74AC05。触发器用来整理\RESET 波形，反相器用来产生\EMUBRK 信号并且保证它和\RESET 之间的时序关系如下：

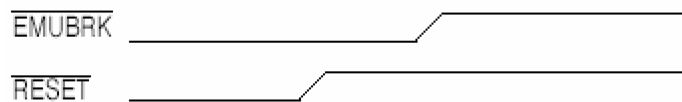


图 1.15. RESET 电路进入 BOOTSTRAP 时序图

1.3.3. TQFP 和 LQFP 器件的焊接方法

在嵌入式系统硬件设计中，会大量用到表贴元器件特别是芯片，制作少量样机时，非常需要进行手工焊接。下面介绍焊接TQFP和LQFP器件的焊接方法，供读者在实际的焊接工作中参考。

1．所需工具和材料

合适的工具和材料是做好焊接工作的关键：

（1）焊锡丝，直径为0.4mm或0.5mm；
（2）电烙铁，要求烙铁尖要细，顶部的直径在1mm以下，功率为25W（不需选用功率过大的）；
（3）无腐蚀型松香焊锡膏；
（4）吸锡网，宽度为1.8mm；
（5）无水乙醇（酒精）含量不少于99.8%；
（6）防静电尖头镊子（不要平头）和一组专用的焊接辅助工具（两端有尖的、弯的各种形状）；
（7）一把小硬毛刷（非金属材料）；
（8）数字万用表；
（9）放大镜，最小为10倍，可根据自己的实际情况选用头戴式、台灯式或手持式。

2．焊接操作过程

首先检查QFP的引脚是否平直，如有不妥之处，可事先用尖头镊子处理好。PCB上的焊盘应是清洁的。

（1）在焊盘上均匀涂上一层焊锡膏，由于焊锡膏有粘性，除了有助焊作用，还方便芯片的定位。
（2）用尖镊子或其它的方法小心地将QFP器件放在PCB上，然后再用尖镊子夹QFP的对角——无引脚处，使其尽可能的与焊盘对齐（要保证镊子尖不弄偏引脚，以免校正困难）。要确保器件的放置方向是正确的（引脚1的方向）。
（3）另一只手拿一个合适的辅助工具（头部尖的或是弯的）向下压住已对准位置的QFP器件。
（4）将烙铁尖加上少许的焊锡，焊接芯片对角的两点引脚。此时不必担心焊锡过多而使相邻的引脚粘连，目的是用焊锡将QFP固定住。这时再仔细观察QFP引脚与焊盘是否对得很正，如不正应及早处理。
（5）按上步的方法焊接另外对角的两引脚，使其四面都有焊锡固定，以防焊接时窜位。
（6）这时便可焊接所有的引脚了。

3．焊接要领

（1）在焊接时要保持烙铁尖与被焊引脚是并行的。

（2）尽可能防止焊锡过量而发生连接现象，如果出现粘连也不必立即处理，待全部焊接完毕后再统一处理。同时也要避免发生虚焊现象。

（3）焊接时用烙铁尖接触每个QFP引脚的末端，直到看焊锡注入引脚。可随时向烙铁尖加上少量焊锡。

（4）电烙铁不要长时间的停留在QFP引脚上，以免过热损坏器件或焊锡过热而烧焦PCB板。

4．清理过程

（1）焊完所有的引脚后要清除多余的焊锡。在需要清除焊锡的地方将吸锡网贴在该处，如有必要可将吸锡网浸上焊锡膏。用电烙铁尖贴在被吸点边缘的吸锡网上，吸锡网有了热量就会把多余的焊锡吸在吸锡网上以解除粘连现象。存留在吸锡网上的焊锡可随时给以剪掉清理或加热甩掉。

（2）用10倍（或更高倍数）放大镜检查引脚之间有无粘连假焊现象，或将万用表调到测试二极管极性一挡（一般此时两测试笔之间短路会有蜂鸣声），检查芯片两相邻引脚是否出现不应该的短路。如有必要可重新焊接这些引脚。

（3）检查合格后需清洗电路板上的残留焊锡膏，以保证电路板的清洁美观，更能看清焊接效果。

（4）用毛刷浸上无水乙醇，然后用毛刷沿引脚方向顺向反复擦拭，用力要适中，不要用力过大。要用足够的酒精在QFP引脚处仔细擦拭，直到焊锡膏彻底消失为止。如有必要可更换新的酒精擦拭，使得清洗的电路板及器件更美观。

（5）最后再用放大镜检查焊接的质量。焊接效果好的，应该是焊接器件与PCB之间，有一个平滑的熔化过渡，看起来明亮，没有残留的杂物，焊点清晰。如发现有问题之处，再重新焊接或清理引脚。

（6）擦拭过的线路板应在空气中干燥30 分钟以上使得QFP下面的酒精能够充分挥发。

上述介绍的焊接技术是我们在实际的焊接工作中积累的一些经验。焊接QFP器件原本就不是很难，只要细心观察精心操作，就会得到满意的焊接成果。

1.3.4. 硬件调试

PCB 板制作完成后，要对比较重要的电路进行分块焊接，分块调试。

（1）首先完成电源模块，检查输出电压是否符合要求；

（2）然后焊接 CPU 和晶振，上电后检查晶振起振是否正常；

（3）焊接 MAX3232，检查串口电压转换是否正常；

（4）焊接 9 针串口接头，和 PC 上位机连接，使用 BBUG 程序检查是否正常进入 BOOTSTRAP 模式；

（5）焊接上 EDO 芯片，使用 BBUG 检查对存储器能否进行正常读写；

（6）连接上 LCD 模块，通过 BBUG 在 EDO RAM 中写入显示信息，检查 LCD 显示是否正常；

（7）焊接上 FLASH 芯片，使用 BOOTLOADER 对 FLASH 进行读写；

（8）完成系统其他部分。

第二章 操作系统

2.1. 基础知识

2.1.1. 操作系统功能

操作系统是充当计算机用户和计算机硬件之间的一个中介，并用于管理计算机资源和控制应用程序运行的计算机程序。

简单的讲，操作系统一般会提供以下服务：

- **程序运行** 一个程序的运行离不开操作系统的配合，其中包括指令和数据载入内存，I/O 设备和文件系统的初始化等等。
- **I/O 设备访问** 每种 I/O 设备的管理和使用都有自己的特点。而操作系统接管了这些工作，从而使得用户在使用这些 I/O 设备的过程中会感觉更方便。
- **文件访问** 文件访问不仅需要熟悉相关 I/O 设备（磁盘驱动器等）的特点，而且还要熟悉相关的文件格式。另外，对于多用户操作系统或者网络操作系统，从计算机安全角度考虑，需要对文件的访问权限做出相应的规定和处理。这些都是操作系统所要完成的工作。
- **系统访问** 对于一个多用户或者网络操作系统而言，操作系统需要对用户系统访问权限做出相应的规定和处理。
- **错误检测和反馈** 当操作系统运行时，会出现这样那样的问题。操作系统应当提供相应的机制来检测这些信息，并且能对某些问题给出合理的处理或者报告用户。
- **系统使用纪录** 在一些现代操作系统中，出于系统性能优化或者系统安全角度考虑，操作系统会对用户使用过程纪录相关信息。
- **程序开发** 一般操作系统都会提供丰富的 API 供程序员开发应用程序，并且很多程序编辑工具，集成开发环境等等也都是通过操作系统提供的。

而计算机有很多资源，它们分别用于数据的传输、处理或存储以及这些操作的控制。这些资源的管理工作就交给了操作系统。

2.1.2. 操作系统发展史

- **串行处理系统**

在二十世纪四五十年代，电子计算机发展初期，没有操作系统的概念，人们通过一个由显示灯、跳线、某些输入输出设备同计算机打交道。当需要执行某个计算机程序时，人们通过输入设备将程序灌入计算机中，然后等待运行结果。如果中间出现错误，程序员就得检查计算机寄存器、内存甚至是一些元器件以找出原因所在；如果顺利完成，结果就从打印机上打印出来。人们称这种工作方式为串行处理方式。随着计算机技术的发展，一些较为成型的软件开始出现，比如说，调试器、I/O 驱动等。

- **简单批处理系统**

由于早期的计算机系统十分昂贵，人们希望通过某种方式提高计算机的利用率。于是批处理的概念就被引入了。

在早期的批处理系统中，功能相对比较简单，其核心思想就是借助某个称为监视器的软件，用户不需要直接和计算机硬件打交道，而只需要将自己所要完成的计算任务提交给计算机操作员。在操作员那里，所有计算任务按照一定的顺序被成批输入计算机中。当某个计算任务结束之后，监视器会自动开始执行下一个计算任务。

- 多道程序设计批处理系统

即便是采用了批处理技术，并不能对计算机资源进行有效利用。一个很头疼的问题就是 I/O 设备的操作速度往往比处理器慢很多。当某个批处理任务需要访问 I/O 设备的时候，处理器往往处于空闲状态。基于这方面的考虑，多道程序设计思想被引入了批处理系统中。通常，多道程序设计也可被称为多任务。即多道程序设计批处理系统也可称为多任务批处理系统。

多道程序设计思想的引入允许某个计算任务在等待 I/O 操作的时候，计算机可以转而执行其它计算任务。从而提高处理器的利用率。

- 分时系统

在多任务批处理系统中，计算机资源的利用率得到了很大提高。问题是如果用户希望能够干预计算任务的执行该怎么办？我们需要引入一种交互模式来实现这一功能。

分时的概念引入了。在分时系统中，处理器时间按照一定的分配策略在多个用户中间共享。在实际的单处理器系统中，是多个任务交替获取处理器控制权，交替执行，从而提供更好的交互性能。

- 现代操作系统

现代操作系统技术是在综合了以上四种典型的操作系统技术的基础上提出的操作系统实现方式，它适应了现代计算机系统管理和使用的要求。其主要特征是多任务、分时、而且很多系统都开始陆续加入多用户功能。现代操作系统一般包括：

（1） 进程及进程管理，

（2） 内存及虚拟管理，

（3） 信息保护和安全，

（4） 调度和资源管理，

（5） 模块化系统化设计。

所有这些我们将在后续章节给予详细阐述。

2.1.3. Linux 与嵌入式 Linux

Linux 作为一个典型的现代网络型操作系统，其中所涉及到的技术实现涵盖了操作系统技术的最新成果。它是一个多用户多任务操作系统，支持分时处理和软实时处理（后面将给予阐述），并带有微内核特征（如模块加载/卸载机制），具有很好的定制特性。由于它是开放源码的，全世界很多科学技术人员在不断对它完善的同时，还增加了越来越多的新功能，比如说支持硬实时任务处理等。Linux 作为一个现代操作系统的典型实现，可以说是一个计算机业与时俱进的产物，它不断更新，不断完善，其新功能的加入和完善速度超过了现今世界任何一种操作系统。

功能的不断增加和完善，灵活多样的实现，可定制的特性、开放源码等等，使得它的应用日益广泛，大到服务器和计算机集群，小到 PDA 和控制器，可以说是无处不见。

而 Linux 在嵌入式系统应用方面尤其显示出其优越性。可以预见，Linux 的强大的网络功能将赋予嵌入式系统对网络天然的亲和力，从而为嵌入式系统的网络互连和功能扩展准备了广阔的发展空间。

2.2. 操作系统内核

2.2.1. 内存管理

在单任务操作系统中，内存被分成两部分，一部分给了操作系统的驻留程序，另一部分则分给了用户进程。而在多任务操作系统中，后一部分需要继续细分给不同的进程。这个工作是由操作系统的内存管理机制实现的。

一个操作系统的内存管理效率对它的性能有着很重要的影响。一般而言，我们希望内存在通过有效的分配之后能够容纳更多的任务以提高 CPU 的利用率。

2.2.1.1. 内存管理功能

一般而言，内存管理需要完成以下功能：二次定址、保护、共享、逻辑组织和物理组织。下面给予简单阐述：

首先，进程调入调出内存存在随机性，我们需要内存管理提供二次定址功能使得任务再次调入内存能够和先前一样正常运行。

其次，不同的进程为了自己的正常运行需要有自己的私有空间，内存管理需要提供相应的保护机制以容许这种私有空间的存在。

再次，为了能够协同完成更高级的功能，不同进程之间需要有不同形式的交互，比如说访问对方数据，使用对方进程代码等等，而内存管理所完成的共享功能可以保证顺利实现。

另外，由于计算机采用的是线性存储设备，而计算机程序本身为了完成自身的逻辑任务就有了适合自己特点的逻辑划分，如果操作系统或计算机硬件能够顺利实现这个逻辑划分和线性存储之间的顺利转换对程序本身实现的相对独立性等方面不无裨益。因此内存管理需要提供相应的逻辑组织功能。

最后一点需要指出的就是，计算机存储设备是多样的，如何合理的管理这些存储设备以提高操作系统性能也是内存管理的功能要求的，这就是内存管理的物理组织功能。

2.2.1.2. 内存分割

计算机程序最终是要装载到内存中才能运行的。这就涉及到多个进程如何在内存中合理占用空间的问题，也就是内存分割的问题。从操作系统产生发展至今，已经提出了不少内存分割机制，诸如固定分割、动态分割、分页和分段等四种内存分割机制。

- **固定分割（Fixed Partitioning）**

固定分割，顾名思义，就是将内存固定划分为若干大小相同或不同的区域。当程序装入时，选择与其大小最相近的一个区域将其载入即可。这种方式的缺点就是引入了内部碎片（Internal Fragmentation），即每个程序需要占用一整块区域，而它的大小很难与区域大小完全相符，区域大小多出来的部分也无法被其它程序使用，从而造成内存资源的浪费。

- **动态分割（Dynamic Partitioning）**

为了克服固定分割机制的缺陷，人们提出了动态分割机制，其中心思想就是内存分割出来的区域大小具有一定的随意性，它可以在内存空间允许的情况下根据装入的程序大小

来在内存中分割出相同大小的一片区域并装入该程序。这样就避免了每个分区内部碎片产生的可能。但是当内存中没有分配的区域大小不足于装载任何程序的时候就产生了内存资源的浪费，这样的浪费叫做外部碎片（External Fragmentation）。

- **分页（Paging）**

固定分割会导致内部碎片，而动态分割会导致外部碎片。都可能造成内存资源的巨大浪费。这时候我们在想，能否把内存平均分割成大小相对较小的一些区域，这些区域通常被叫做页帧（Page Frame），然后把它们分配给需要装载的程序。问题是当页帧较小时，可能任何程序都要比它的大很多，产生无法装载的问题。这个时候我们在想是否可以将程序本身也分割成与内存分区大小一致的程序段，这样就可以装入内存运行了。这就是分页机制的基本思想。我们可以很容易想到，这样的内存分割方法不会产生外部碎片。而内部碎片也只可能在程序的最后一个程序段装入内存分区时产生，因为它俩的大小一般是不相同的。而由于内存分区相对较小，这样的话，内存空间的利用率仍然可以很高。

在分页机制中，程序在载入内存时需要实现一个程序所在的虚拟内存空间地址，即逻辑地址，到物理内存空间地址，即物理地址的转换，这个转换通常是由一个或若干个查表动作来完成的，而这个表即所谓的分页表。

- **分段（Segmentation）**

在分页机制中，程序本身也被顺序分割成与内存分区大小相同的程序段。程序的这种分割方法缺乏直观性。分段机制的引入克服了这个缺点。在分段机制中，程序及其数据被分割成若干有意义的段，如我们通常所讲的程序段，数据段等等，而这些段是大小不一的。从这点来看，分段机制和动态分割机制类似。所不同的是在分段机制中，每个程序可以占有若干个不同的内存分区，而这些分区可以是不连续的。不过，分段机制和动态分割机制一样存在外部碎片问题，但不存在像固定分割机制和分页机制那样的内部碎片问题。

在分页机制中，程序在载入内存时同样需要实现一个程序所在的虚拟内存空间地址，即逻辑地址，到物理内存空间地址，即物理地址的转换，这个转换通常是由一个查表动作来完成的，而这个表即所谓的分段表。

2.2.1.3. 虚拟内存

虚拟内存机制基于分页技术或者分页与分段两种技术的结合，它是现代操作系统的一个显著特征。虚拟内存技术的实现需要有硬件支持，并得到操作系统配合共同完成，从而能够提供给每个进程一个几乎不受限制的虚拟内存空间。

虚拟内存机制的实现不仅需要操作系统方面的软件支持，而且需要有相应的硬件支持，比如地址转换功能支持等。

硬件支持主要包括地址转换功能，以及一些为了提高软件支持效率而做出的相应的支持，如 MMU 等。在这一节我们主要介绍虚拟内存机制在操作系统方面的实现。

操作系统方面的支持需要考虑三个问题：

（1） 系统是否需要虚拟内存支持；

（2） 系统对内存分割机制的选择问题；

（3） 内存管理算法的选择问题。

前两个问题的回答有赖于硬件支持，而第三个问题则是纯属软件问题，主要包括分页的取放、清理和替换方法；驻留部分管理和负荷控制等几个方面，下面给予分别阐述：

- **分页的取放**

当需要某一分页载入内存使用时，分页的取放动作就需要连续发生了。

分页的“取”可以有两种方法，一种是只有当它被要求时才会被载入内存；另一种是，即便某个分页未被要求也会被载入内存。后一种方法的出现主要是考虑到计算机系统的外部存储设备的处理速度和延迟比较大，而程序在外部存储设备中可以通过某种工具（如碎片整理程序）实现连续性存储，一次性读取多个分页会提高系统整体性能。

而分页的“放”在采用分页机制的情况下，不存在算法的选择问题。但是如果采用分段机制的话，则需要做一些考虑，比如说是选择和段大小最接近的一个空闲内存区域放置呢，还是选择第一个适合的空闲内存区域放置等等。

- 分页的替换

当内存区域已经被分页全部占用之后，仍然需要有新的分页载入时，就涉及到一个如何选择内存中现有分页中的哪个被替换的问题。

谈到分页替换的时候，我们首先应当了解内存中的某些区域由于被操作系统的某些核心程序所占用，是无法被替换的。

分页替换有一些基本算法，这其中包括最优算法、LRU 算法、FIFO 算法和 CLOCK 算法。下面分别给予简单阐述。

最优算法就是将内存中再次访问时间间隔最长的那个分页替换掉。这种算法由于需要所有分页在任何时间被访问的确切纪录，而计算机是没法知道未来究竟哪个分页会在什么时间被访问到，因此在现实中是无法实现的。但是它却因为可以给出一个最优答案，所以常常作为其它算法性能评价时的一个参考。

LRU（Least-Recently-Used）算法是将内存中在过去时间里最早访问到的那个分页替换掉。这种方法在实践中证明通常是很有效的，并且非常接近最优算法的性能。但是由于需要对每个分页的访问时间进行纪录，在实现起来存在一定的困难。

FIFO（First-In-First-Out）算法也称为排队算法，遵循先进先出原则，即最先被载入内存的那个分页将被最先替换掉。这种算法比较容易实现，但是其性能相对较差。

CLOCK 算法综合了 LRU 算法的性能优势和 FIFO 的实现优势。现在很多实际使用的分页替换算法都是它的变种。由于在算法进行过程中，分页的替换犹如在一个时钟盘上旋转轮流选择，因此也就被形象地称为 CLOCK（时钟）算法。

最简单的 CLOCK 算法是这样的：

首先需要给每一个页帧增加一个标志位 U，该位在对应页帧第一次装入分页时置位为 1，而当该页再次被访问时仍然置位为 1。

在 CLOCK 算法中所有可以用来替换的页帧顺序组成一个首尾相连的环，另外需要有一个指针标志下一个候选页帧。当某页被替换后，该指针指向环中的下一个页帧。

当需要某页帧被替换时，该指针就在环中顺序移动，直到找到一个其标志位 U 置位为 0 的页帧，然后将其替换为需要装载的分页，同时安装上面所讲的规则将该页帧标志位 U 置位为 1，并将指针指向被替换页帧的下一个页帧。

在指针顺序移动过程中，如果遇见标志位 U 置位为 1 的页帧，此时，指针在滑过该页帧的同时，该页帧的标志位 U 需要被置位为 0。

- 分页的清理

分页的清理是与分页的“取”动作相对的。它所要解决的问题就是当一个被做过修改的分页需要写回外部存储设备的时候，应该采取什么方案。同样也有两种方案，一种是只有当该页要被替换的时候才会被写回，另外一种就是在该页被选择替换之前和其它页一起成批被写回。这两种方法都存在各自的缺点，前者会导致分页替换的额外操作，即被修改分页的回写动作；而后者则在成批回写的某分页在内存中再次被修改时出现对该分页的回写属于无用操作的情况。

- 驻留部分管理

驻留部分管理是指对各进程驻留在内存中分页的管理，主要包括两个方面，一个是驻留部分大小的选择；一个是分页替换范围的选择。

驻留部分大小的选择直接决定了每个进程在内存中将要占用多少页帧，主要包括两种，一种是固定分配法，即操作系统根据进程特点分配给固定数目的页帧，这个数目在进程的生存周期中固定不变；一种是可变分配法，即操作系统在进程初始化时根据其特点分配给一定数量的页帧，而在进程运行过程中根据其运行性能，比如说分页替换的频度来实时决定给该进程分配新的页帧数量，以便提高其运行性能。后者比前者更灵活，从而可以实时改善程序运行性能。

而分页替换范围的选择也有两种，一种是局部替换，即当某个进程需要加载入新的分页时，新分页只能从操作系统分配给该进程的内存区域中获取可替换的页帧；一种是全局替换，即当上述情况发生时，新分页可以在全部内存区域中寻找可被替换的页帧。后者比起前者来讲更容易实现。

- 加载控制

加载控制主要关心的是在内存中驻留的进程数量的确定问题。

一般而言，当内存中驻留进程的数量过少时，所有进程同时阻塞的情况发生的可能性就会加大，从而导致 CPU 闲置的可能性增大；而当内存中驻留进程的数量过多时，就会导致缺页中断频繁发生，同样会降低 CPU 的利用率。

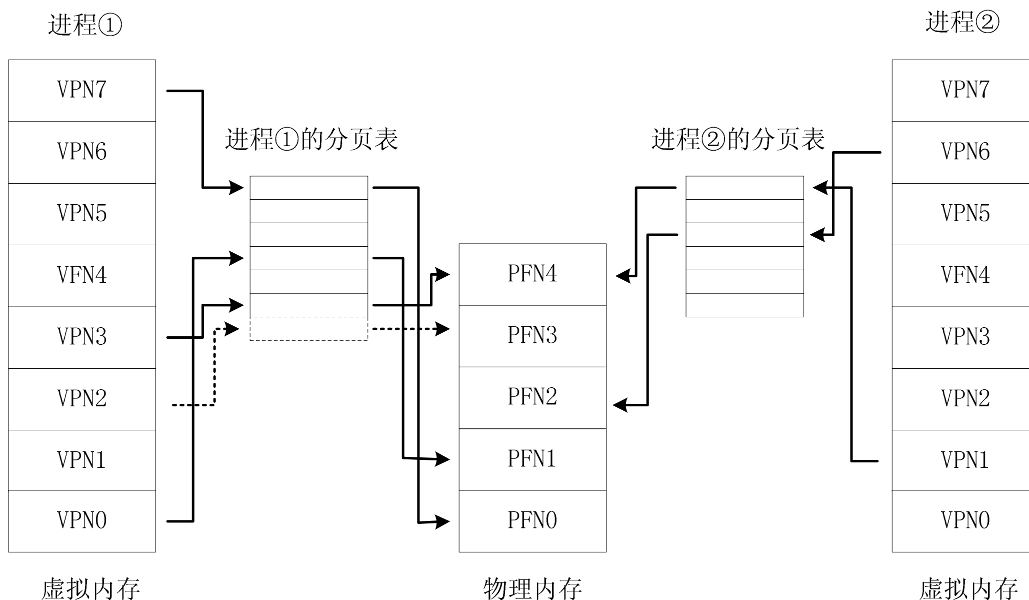


图 2.1. 一个抽象的虚拟内存模型

2.2.1.4. Linux 的内存管理机制

在 Linux 的内存管理机制中，分段极少用到，而倾向于使用分页，主要原因有二：

(1) 当所有进程共享一个线性地址空间的话内存管理会变得更简单；

(2) 这样可以让 Linux 有更强的可移植性：有些 RISC 处理器对分段的支持很有限。

为了更清楚地说明 Linux 的内存管理机制，我们先介绍一个基于分页的虚拟内存抽象模型（图 2.1.）

在该模型中，有两个进程，进程①和进程②，都在各自的虚拟内存空间中分别被划分为八个虚拟分页（Virtual Page），并有固定的编号。而物理内存空间也被分为与虚拟分页同样大小的四个不同的页帧（Page Frame），并有固定的编号。各进程通过其进程分页表实现其虚拟分页到物理内存页帧的映射，以备进程加载，如图 2.1 所示，进程①的分页 VPFN7 通过对应的分页表映射到物理内存的页帧 PFN0。

在这个虚拟内存模型中，一个逻辑地址通常包括两个部分，它所在的虚拟分页的固定编号（分页编号）和它在该分页中的地址偏移（页内偏移）；而与在分页表中与每个虚拟分页对应的分页表条目在理论上讲包括三方面的信息：该条目是否有效；该条目所描述的物理内存页帧编号；访问控制信息，如读写控制、是否可执行代码等。

- 地址转换与进程分区表

逻辑地址是存在于进程虚拟地址空间的东西，计算机硬件只认识物理地址。因此在程序执行过程中需要完成一个逻辑地址向物理地址转换的过程。在基于分页机制的虚拟内存管理模式中，逻辑地址向物理地址的一般转换过程（图 2.2）如下：

首先，系统取出进程要访问的逻辑地址所包含的信息，即分页编码和页内偏移；然后根据前者，在进程分页表中查出对应的物理内存中的页帧编码，而由于页帧和虚拟分页大小完全相同，页内偏移原封不动地下传；最后根据所取得的页帧编码和下传来的页内偏移就得到该逻辑地址在物理内存中的物理地址。

通常这一过程是由硬件，如内存管理单元（MMU）来实现的，当然也有软件实现。

在大多数系统中，每个进程只有一个分页表，而在现代计算机中，由于计算机寻址空间很大，每个进程可能占用很大的虚拟内存空间。比如说，在 Pentium 中，每个进程可有 2^{31} 字节的虚拟地址空间。而如果每个分页的大小为 2^{9} 字节的话，每个进程的分页表的条目数就可能多达 2^{22} 个。但是，分给每个分页表的物理内存空间不可能很大。为了解决这个问题，大多数虚拟内存解决方案是将分页表存放在虚拟内存空间，而不是直接放在物理内存空间。这就是说进程分页表像其它分页一样需要同样的分页管理机制。当一个进程在运行时，至少是它的分页表的一部分需要放在物理内存中，这其中包括正在运行的分页的分页表条目。

有些处理器，如 Intel x86 系列，StrongARM 系列，使用二级解决方案来组织大型的分页表。在这种方案中，针对每个进程有一个分页目录（第一级），用于管理指向分页表（第二级）的条目。这样如果每个分页目录的长度是 L1，而每个分页表的条目的最大数目是 L2 的话，那么每个进程就可以有多到 L1×L2 的分页。

而有些处理器，如 Alpha 处理器，则采用三级解决方案。显然，依照上面二级解决方案类推，如果每一级的参数依次为 L1、L2、L3，则每个进程就可以有多到 L1×L2×L3。Linux 总是假定处理器采用三级解决方案（图 2.3）。每一个分页表中的条目包括一个指向下一级分页表的页帧编码。而一个虚拟地址至少包括四个部分，三个级别的分页表索引和一个页内偏移地址。另外计算机系统（一般是 MMU）本身需要提供一个预先设定的页目录基址。要将一个虚拟地址转换为一个物理地址，需要顺序进行以下步骤：在第一级，根据预设页目录基址和虚拟地址中的第一级索引（页目录索引）得到一个分页表条目的地址，这个条目包含中间页目录基址（第二级）和相关控制信息；在第二级，根据第一级得到的基址和虚拟地址中相应级的索引（中间页目录索引）得到一个包含页表基址和相关控制信息的分页表条目的地址；而在第三级，根据第二级得到的基址和虚拟地址中相应级的索引（分页编号）得到一个包含物理内存页帧编号和相关控制信息的分页表条目的地址；在最后，根据第三级得到的物理地址页帧的基址结合虚拟地址中给出的页内地址偏移得到实际的物理地址，从而最终完成虚拟地址到物理地址的转换过程。

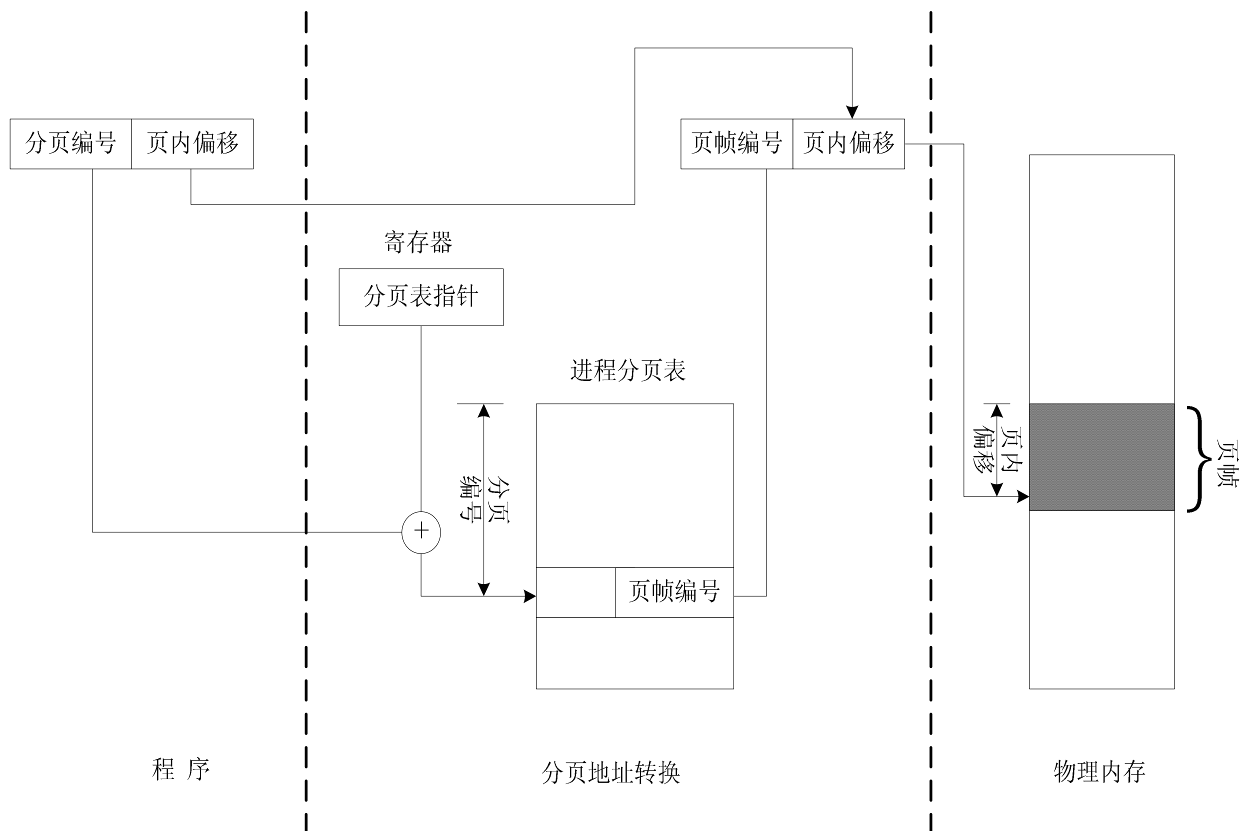


图 2.2. 分页地址转换机制

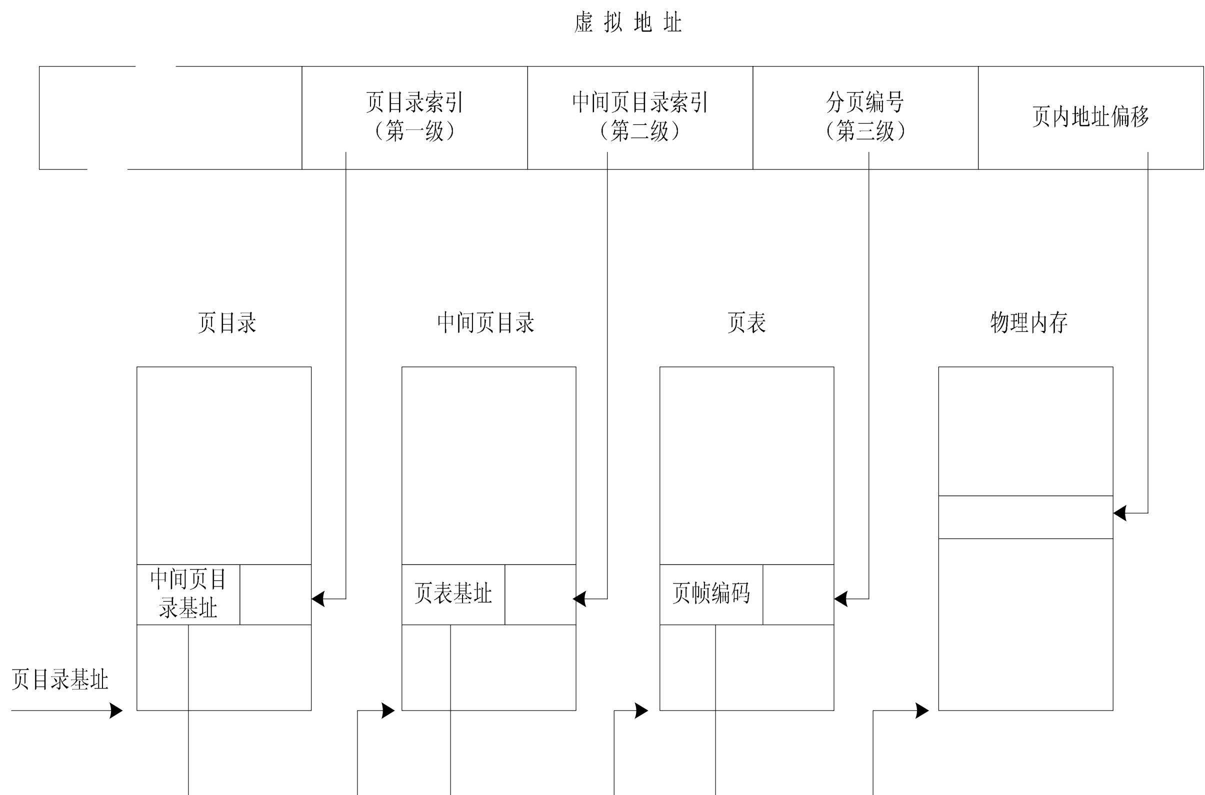


图 2.3. Linux 的三级分页地址转换机制

但事实上，正如前面所提到过的，不同的处理器可以识别的分页地址转换解决方案是不同的，Linux 为了实现跨平台，就提供了一系列的宏定义来实现从其假定的三级解决方案向处理器实际使用的解决方案的转换。事实证明这种处理方法是很有效的。

- **分页加载请求（Page Demanding）**

当进程某个分页被加载到物理内存时，其分页表中就会增加一项，用于实现在该分页内的所有虚拟地址到对应的物理地址的转换过程。当某个虚拟地址在该进程所有装载入的分页所包括的范围以外的时候，该虚拟地址就没有对应的物理地址存在，如图 2.1 所示的情况，对进程①而言，当虚拟地址位于分页 VPN2 所包括的地址范围内时，虚拟内存管理就会产生一个寻页错误中断，要求操作系统将该分页载入内存中，同时该进程进入阻塞状态。

当操作系统响应请求将该页载入物理内存之后，如图 2.1 所示，VPN2 被载入物理内存页帧 PFN3，同时进程①的分页表增加一个新条目（如虚线框所示），该进程将被重新激活，并继续原来的操作。

- **分页替换（Page Replacement）**

在 Linux 中，分页替换操作也称为页交换（Page Swapping）。在执行这个操作之前，首先需要判断该替换的分页。在 Linux 中采用 LRU 分页替换算法。根据该算法选择准备被替换的分页，而后判断该分页在载入内存之后是否有改动，如果没有的话，可以直接被替换，如果有改动，则需要该分页在外围存储设备中作永久保存之后才可以被替换。

前面讲过，LRU 替换算法的效率非常接近最优替换算法。

- **共享虚拟内存（Sharing Virtual Memory）**

虚拟内存机制可以让共享内存变得很容易。所有的内存访问是通过分页表，而每个进程都有其独立的分页表。如果两个进程需要共享一个物理内存页帧，则这个页帧编号只需要同时出现在这两个进程分页表条目中就行了。

从图 3.1 可以看出两个进程共享物理内存页帧 PFN4。对进程一而言，该页帧对应的虚拟分页是 VFN3，而对进程②来讲，对应的则是 VFN1。这也就是说，在虚拟内存机制中，共享的虚拟内存可以处于各进程虚拟地址空间的不同位置，由此可见，虚拟内存机制的引入给共享内存的实现带来一定的灵活性。

2.2.2. 进程与中断管理

对每一个现代操作系统而言，其基本任务之一就是进程管理。操作系统需要为进程分配资源；实现进程间共享和交换信息；保护进程资源；以及实现进程间同步，为此，操作系统需要为每一个进程维护一个特定的数据结构用于描述该进程的状态和资源占用情况，从而实现对进程的管理和控制。

在一个单处理器多道程序设计系统中，多个进程是交替执行的。而在一个多处理器系统中，多个进程不仅可以在某个处理器上交替执行，而且还可以在多个处理器上被并行处理。不管是交替方式还是并行处理都会导致进程并发现象，这不仅给操作系统也给程序员带来一系列的麻烦。

在现代操作系统中，进程管理因为线程的引入变得更加复杂。在一个多线程系统中，进程成了资源管理器，而线程成为程序的基本执行单元。

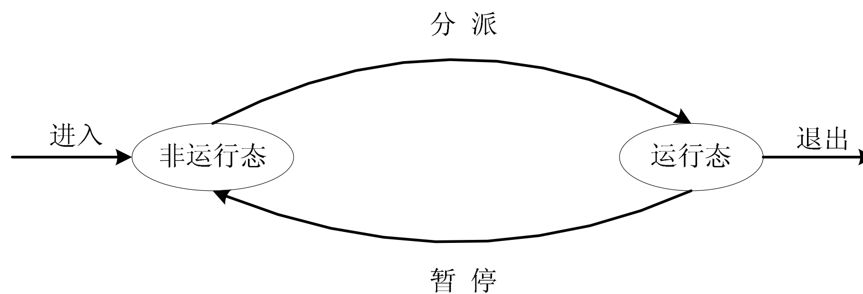
中断是现代计算机系统普遍采用和支持的技术。因为中断响应的本质是占用一段处理器时间，必然与进程有密切的联系，因此本章在介绍进程管理的同时也对中断与定时管理作相应的阐释。

2.2.2.1. 进程描述与控制

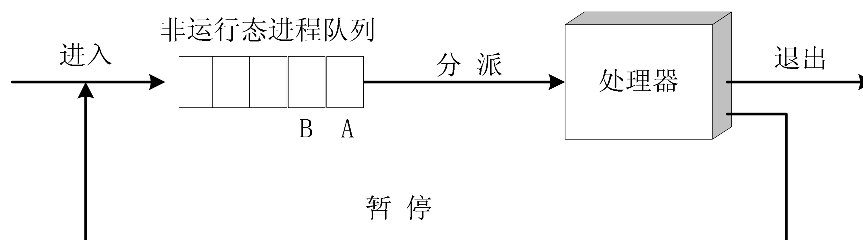
- **进程状态**

进程状态的认定直接影响到进程描述与控制的复杂度。而人们对进程状态的认定也是一个随着对计算机系统本身的认识并结合进程管理与控制的需要来逐渐细化的。

进程状态的最简单模型就是二值模型（图 2.4），即运行态和非运行态。在这样一个简单模型中，操作系统决定进程是否处于运行态，而处于非运行态的进程要么是永远退出，要么就是加入到一个先入先出的非运行态进程队列，等候下一次运行。



(a)状态转移图



（b）队列模型

图 2.4. 进程状态描述的二值模型

二值模型虽然简单，但是在现实中却很难应用。比如说在图 2.4 所示的进程队列中，有 A 和 B 两个进程一前一后被加入队列中，此时，该分派 A 占用处理器了，但是 A 这个时候正在等待某个 I/O 操作的完成，则整个队列将会因此而等待，而事实上，进程 B 本来可以在这段等待时间里完成所有操作后正常退出的，换言之，二值模型虽然简单，但是却会造成处理器时间的浪费。

为弥补这一缺憾，三值模型提出，即将原有二值模型中的非运行状态细化为（准备）就绪状态和阻塞状态，分别对应于图 2.4 所示的进程 B 和 A 所处的状态，显而易见，三值模型是二值模型的进一步细化。

另外，再加上两个实践证明有效的状态，即创建和退出，从而得到普遍应用的进程状态描述的五值模型（图 2.5）。

首先来看这个五值模型的各个状态转移情况：

(1) **无 → 创建** 一个新的进程被创建用于执行一个程序。

(2) **创建 → 就绪** 当操作系统可以执行其它进程时，它会首先将以前创建的进程转为（准备）就绪状态。大多数操作系统都会对存在的进程数或者给它们分配的虚拟内存空间的大小有某些限制，以确保系统性能不至于降低。

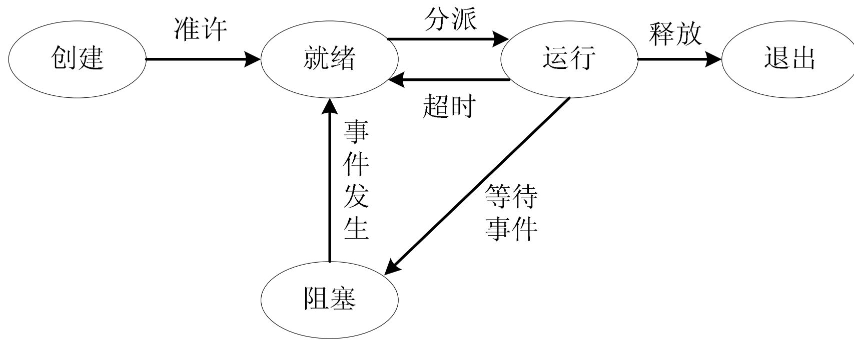
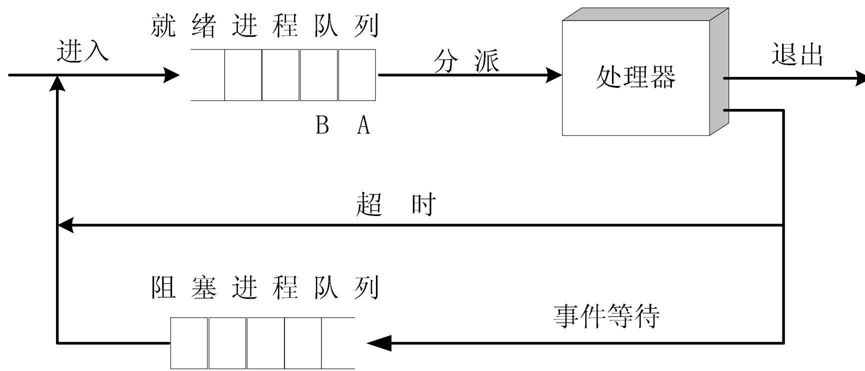
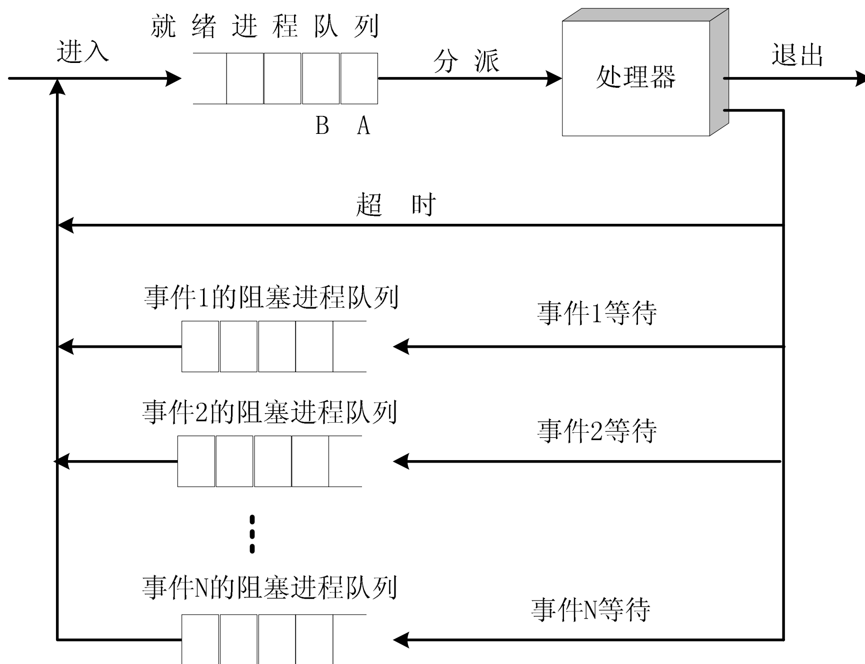


图 2.5. 进程状态描述的五值模型



（a）单事件等待的队列模型



（b）多事件等待的队列模型

图 2.6. 进程五值状态描述的队列模型

(3) **就绪 → 运行** 当轮到某个处于（准备）就绪状态的进程运行时，操作系统会将其状态转换为运行态。究竟哪个进程被选择运行，需要依据一定的控制算法，后面将会提到。

(4) **运行 → 退出** 当某个进程已经完成自身的任务或者因为某种原因终止了的话，操作系统会将其状态从运行状态转换为退出状态。

(5) **运行 → 就绪** 事实上在现代操作系统中，尤其是在单处理器系统中，系统并非是在一直执行某个进程，而是往往分配给每个进程一个处理器时间片，在这个时间片里，进程完全占有处理器。这样，在大多数情况下，一个处于运行状态的进程往往是因为操作系统分配给自己的时间片已经耗尽，需要从运行状态退出，一般的处理方法是按照超时处理将其状态转换为（准备）就绪状态。

(6) **运行 → 阻塞** 当一个正在运行中的进程需要某个事件发生后才能继续运行时，操作系统将其状态从运行状态转换为阻塞状态。这样操作系统可以运行其它进程。

(7) **阻塞 → 就绪** 当某个处于阻塞状态的进程被告知它所等待事件已经发生之后，其状态就会被操作系统从阻塞状态转换为（准备）就绪状态，以便其下次继续运行。

另外需要说明的是还存在两种形式的状态转换，一个是从（准备）就绪状态到退出状态的转换，另一个是从阻塞状态到退出状态的转换。当父进程退出时，它所创建的子进程也会跟着退出，这时就可能出现这两种情况。

下面给出进程状态描述的五值模型的两种队列图，图 2.6（a）是只有一种事件等待的情况，而图 2.6（b）则是多事件等待的情况。

- **进程描述**

操作系统为了管理和控制进程，它必须知道进程的位置和进程属性，后者包括进程的标识、状态信息和控制信息。

1) 进程位置

谈到进程的位置就必须了解进程的物理形式。一说到某个进程我们就自然而然地将它和一段程序以及和这段程序相关的数据联系起来；同时，操作系统执行该进程时需要维持一个或多个堆栈用于跟踪过程调用以及过程间参数调用；最后，操作系统为了控制和管理该进程需要维护一系列与该进程相关的信息。操作系统所维护的这些信息通常被称为进程控制块（Process Control Block）。与进程相关的程序、数据、堆栈和进程控制块信息统称为进程映像。

进程映像如何在计算机系统中存储，和操作系统的内存管理机制有关。如前一章所提到的，现代操作系统的内存管理基于分段或分页或者两者的综合。一般而言，进程映像总是有一部分驻留在物理内存中，而其它部分则存放在外围设备中。仅就进程的定位而言，操作系统会在物理内存中维护一个主进程表，其中的各个条目包含一个指向一个进程映像的指针，从而标明进程位置。

2) 进程标识

在几乎所有的操作系统中进程标识都是一个唯一的数值型标识。这个标识可以作为在操作系统运行环境中进程的“身份证”，根据它，不仅可以找到对应的进程映像，而且还可以在内存管理、I/O 管理、文件系统管理等方面派上用场。这就好比，我们的身份证作为唯一标识可以作为我们身份的认定，而且在外出旅行、银行信贷、房产按揭等方面派上用场。

另外每个进程内部仍然需要维护一些其它的标识，比如说创建它的父进程的标识、使用它的用户标识等等，这些信息有助于进程的管理和控制中。

3) 进程状态信息

进程运行时的寄存器信息就构成了进程状态信息。当某个运行中的进程被暂时停止时，相应的进程状态信息需要作某些必要的处理，以便进程恢复运行时能够恢复到暂停之前的状态从而能够继续正常运行。

4) 进程控制信息

进程控制信息也就是前面所提到的进程控制块中所包含的信息，它是操作系统用于控制和协调各个运行进程所需要的额外信息，可以大体上分为六类，如表 2.1.

- **进程控制**

进程控制不仅包括对进程创建过程的控制而且还包括对进程状态切换的控制。另外，出于对操作系统某些关键数据如进程控制模块等的保护，进程的执行模式分为两种，即拥有更高权限的内核执行模式和拥有较低权限的用户执行模式。从而进程控制就增加了对进程执行模式切换的控制。

表 2.1. 进程控制信息

类别	详细内容
进程状态和调度信息	用于操作系统调度，其中包括：进程状态、运行时优先级、调度相关信息和进程需要等待的事件等。
维护某种数据结构所需的相关信息	一个进程和其它进程可能会同处于一个队列、环或者其它数据结构中。而进程间也可能存在父子关系。通常需要进程控制块维护指向相关进程的指针以实现这些数据结构
进程间通讯	这包括两个独立进程间通讯所需的各种标志、信号和消息
进程权限	每个进程会有一定的权限，这其中包括可访问的存储区域和可执行的指令集，另外还包括可调用的系统工具和服务等。
内存管理	一系列指向所分配的分段表或分页表信息的指针
资源拥有和使用情况	进程所控制的资源需要指明，比如说打开的文件等。而使用处理器和其它资源的历史纪录一般也需要指明，这部分信息可供操作系统做进程调用时参考。

进程创建的原因可能是多方面的，比如说，有新的批处理任务移交给操作系统完成；有一个新的用户登陆系统；需要提供一项新的服务；现有进程派生出来的子进程等等。但是一旦操作系统决定需要创建一个新进程，它需要完成一系列操作：赋予新进程一个唯一的标识；给其进程映像分配足够的存储空间；初始化其进程控制块；设置某些外部关系连接，比如说，出于调度的需要，该进程需要放入一个（准备）就绪的进程链表中，因此在它被创建时需要反映这些外部关系；另外还有一些额外的数据结构如系统日志等的创建和增加。

进程状态的切换需要操作系统对要切换状态的进程顺序完成以下动作：

（1） 保存处理器上下文，这其中包括它的程序计数器和其它寄存器的内容；

（2） 刷新其进程控制块的相关内容，这其中包括进程状态信息，也有其它一些相关信息比如状态切换的原因，日志等等；

（3） 将该进程的进程控制块转移到相应的新的队列中；

（4） 如果有必要，比如，当该进程处于运行态时，操作系统需要根据某个算法，选择下一个进程来取代该进程。

对于操作系统仅支持内核模式和用户模式的情况，处于内核模式时的进程对内存、处理器及其所有指令、寄存器有完全的控制权；而当处于用户模式时赋予这样的权限怎可能会因为用户进程的某个可能出现的错误（经常发生）而使得系统完全崩溃，事实上也是不

必要的，而当用户进程需要访问一些敏感资源的时候，可以通过系统调用转入内核模式，获得更高级权限从而完成任务。

- 进程和线程

进程作为现代操作系统的一个重要特征，在提高计算机利用率等方面起了很重要的作用。通常人们认为，进程不仅是操作系统的基本执行单元；而且同时也充当了相应资源的管理角色。

随着线程概念的引入，这两个功能逐渐分离，进程就只是一个单纯的资源管理器，它管理着与进程相关的资源如进程映像、各类访问权限等等；而线程成为操作系统的基本执行单元，受操作系统调度和控制。当然，线程也常常被称为轻量级进程，而这个时候的进程也常常被称作任务。

同样，与线程相联系的必然也有相应的描述和控制。线程的状态主要有（准备）就绪、阻塞、运行等三种，而对线程可以有四种基本的操作，即线程的派生、挂起、唤醒和结束。和进程的一个很大的不同之处是线程的阻塞并不一定会导致它所属的进程的阻塞。从这一点来看，多线程比起多进程来将会有更大的灵活性。

另外需要特别指出的是多线程机制的实现可以分为两大类，一类是用户级线程实现；另一类是内核级线程实现。一个纯用户级线程实现的系统，线程管理完全由应用程序自己完成，而内核完全不用管。系统的最小执行单位是进程。而一个纯内核级线程实现的系统，线程管理完全由操作系统内核完成。当然，实际的系统大多会是这两种实现方式的混合。

2.2.2.2. 并发控制：互斥与同步

不管是基于单处理器的多道程序设计，基于多处理器的多处理设计，还是基于多机系统的分布式处理设计，对操作系统而言，最基本的东西就是并发控制（Concurrency Control）。并发控制涉及到一系列问题，比如，进程间通讯、资源竞争和共享、进程同步、进程处理器时间分配等等。

并发现象会在三种不同的情况下产生，即多道程序同时运行时、应用程序基于多进程或多线程设计时以及操作系统基于多进程或多线程设计时。

并发控制实现的最基本方法就是进程间互斥，也就是说，当某个进程正在执行某个动作的时候，其它进程应当避免获得同样的权限。互斥的基本实现方法有两类，即软件方法和硬件方法。基本软件实现方法采用的是所谓的“忙碌-等待”（Busy Waiting）技术。基本硬件方法的指导思想就是通过硬件方法使得进程在执行某个需要互斥的动作的时候不会被中断。另外还有第三类方法，它们常常为操作系统或程序编译器所采用，即信号量（Semaphore）、管程（Monitor）和消息传递（Message Passing）。

- 并发

让我们先来看一个并发的简单实例，以便对并发控制的必要性有一个初步认识。

有一段例程，需要完成这样一个简单任务：即首先（步骤一）从输入设备如键盘读入一个字符，然后（步骤二）将该字符存放在一个全局变量中，最后（步骤三）将该全局变量的值在输出设备如屏幕上输出。总共就这么三步。现在任何一个进程如果需要的话，就可以调用这一段例程来接收并显示一个字符。通过简单分析我们就知道，只有当这三个步骤不间断地一气呵成的话，才会出现我们所希望的正确结果，否则就可能出现逻辑上的错误。

从上一章的内存管理机制我们知道虚拟内存可以很容易的实现进程间共享，这种共享不仅包括执行代码也包括各种数据。上面这段例程的存储方法也采用这种方法，即将其存

放在一个任何进程都可以访问的固定内存区域，这样可以节省内存空间。

下面我们来看在没有并发控制的情况下，单处理器的交替执行模式和多处理器的并行处理模式下两个进程都来调用这个例程时可能产生的后果。

为了简便起见，两个进程分别被称为进程①、进程②。

(1) 交替执行模式

我们来考虑这样一种情况，当进程①执行到该例程的步骤二时，这个时候全局变量中已经有了一个进程①希望输出的字符；此时，该轮到进程②执行该例程，进程①运行中断；同样，当进程②执行到例程的步骤二时，全局变量的内容就被赋予了新值，而这个时候，进程①重获处理器时间，进程②中断；进程①继续执行该例程的第三步，结果是输出的进程②接收的字符，最后，进程①结束；进程②重获处理器时间，执行第三步，输出它希望输出的字符。

显而易见，由于这两个进程交替执行，就导致进程①希望输出的字符被错误的输出进程②要输出的字符，这是我们不希望看到的。

(2) 并行模式

在并行模式下，进程①和进程②同时运行，但是仍然需要调用同一块内存区域的这个例程，修改同一块内存区域的那个全局变量。显然，涉及到对同一个全局变量的访问必然有先后顺序之分，假定进程①先于进程②去访问该全局变量。和交替执行模式一样，最后的输出结果仍然在进程①这里出错。

从以上的简单实例，我们可以看出，尽管进程的交替执行模式和并行模式有着本质的不同，但是在并发控制方面遇到的问题却是类似的。

上面所提到的例程属于运行过程中不能中断只能连贯执行的程序段，我们称之为程序的临界区。另外，有些计算机资源，如打印机，部分 I/O 设备，在接到某个任务的时候，只能在完成该任务之后才能开始下一个任务，这样的计算机资源我们称之为临界资源。临界资源往往是和某个程序的临界区联系在一起的。比如说，打印机，属于临界资源，在接到一个打印任务时，就进入打印程序代码执行阶段，此时如果被中断的话，最后打印结果可能是一个任务的输出与另外一个任务的输出交替出现，所以该段程序代码只能连贯执行。

- **并发控制：软件方法**

首先应当说明的是软件方法的应用有一个限制，就是它只能用在计算机系统共享内存的情况下使用。并且假定，在这样的计算机系统中，对同一内存单元的同时发出的访问请求只能是顺序进行。事实上，这个假设是普遍成立的。

其基本思想是使用一个或若干个内存单元存放一些标记，用于标志进程是否进入临界区或者是该轮到哪个进程进入临界区，而未进入临界区的进程则根据这些标记来决定自己是否应该进入临界区，比如，当标记表明该轮到自己进入临界区而其它进程没有进入临界区的话，该进程就可以进入临界区来完成自己的任务。

下面介绍两种常见的算法：

(1) Dekker 算法

Dekker 算法用于两个进程的互斥控制。Dekker 算法的标记有三个：一个表明该轮到哪个进程进入临界区（用 T 表示），该标记是二值的，对应着两个进程；另外两个分别表明对应进程是否在临界区（用 F 表示）。

当其中一个进程希望进入临界区的时候，它首先将自己的 F 标记设置为真；然后检查另外一个进程的 F 标记，如果该标记为假，则前一进程可以直接进入临界区，否则，它就会去查 T 标记，看是否轮到它进入临界区，如果是的话，它就会不断去查询另外一个进程的标记 F，当该标记被刷新为假的时候，该进程就可以进入临界区；等它跳出临界区后的第一件事就是刷新 T 标记，以便让另外一个进程进入该临界区。如此便完成它运行时的一

个循环。

(2) Peterson 算法

Peterson 算法比 Dekker 算法简单，而且其正确性更容易验证。并且，Peterson 算法可以很容易地扩展以用于多个进程的互斥控制。

先来看两个进程间互斥控制的处理方法。此时，Peterson 算法的标记同样也是三个：一个是表明是该轮到哪个进程进入临界区（用 T 表示），该标记是二值的，对应着两个进程；另外两个分别表明对应进程是否在临界区（用 F 表示）。

当其中一个进程希望进入临界区的时候，它首先将自己的 F 标记设置为真；然后将 T 标记设置成轮到另外一个进程进入临界区对应的值；接着该进程就开始不断的检查 T 标记和与另外一个进程对应的 F 标记，当这两个标记不能表明该轮到另外一个进程进入临界区，并且这个进程就在临界区的话，前一个线程就可以放心大胆的进入临界区了，当它跳出临界区的第一件事就是将它对应的 F 标记设置为假即可。如此便完成它运行时的一个循环。

从上面对 Peterson 算法的描述可以发现它和 Dekker 算法的最大区别就在于 Dekker 算法对某一进程是否该进入临界区，其判断是嵌套结构的，这个嵌套结构会随着所判断进程的数量的增多而变得非常复杂，甚至是很难判定其有效性的。而 Peterson 算法只需要一步判断，对于多进程的互斥控制情况，它可以很方便的进行扩展，而不需要在程序结构上作大的改动，因此可以很方便地扩展到多个进程的互斥控制问题的解决上。

- **并发控制：硬件方法**

并发控制的硬件方法有两种，一个是禁止中断法，另外一个是内置特殊机器指令法。

禁止中断法的基本要领就是系统通过设置原子指令来保证进程在进入临界区的时间里其执行流程不会被中断。这种方法不是很理想，一是在单处理器的交替执行模式将会受到干扰，从而可能造成系统性能下降；二是在多处理器系统中，这种方法失效。

下面我们来看内置特殊机器指令法。

前面我们已经提到，在共享内存的情况下，同时访问某个内存地址的动作只能顺序执行。受这个启发，从互斥控制角度考虑，计算机硬件设计者就开始在处理器内设置一些在一个指令周期完成的不会被中断的却能顺序完成若干动作的特殊机器指令以辅助实现互斥控制。这其中包括附加判断的设置指令（Test and Set Instruction）和交换指令（Exchange Instruction）。由于这两个指令在很多常见的指令集中都涉及到，下面给予简单介绍：

（1） 附加判断的设置指令（TSI）

该指令的基本要领就是对指令的传入参数进行判断，如果该值是 0 的话，则将其重置为 1，并返回真值；否则不对传入参数进行修改，直接返回假值。

在该指令的基础上我们可以很方便地给出多个进程的互斥控制算法如下：

系统需要维护一个全局变量 B，该值初始化为 0，当某个进程发现该值为 0 的时候就可以进入临界区。

现在有一个进程希望进入临界区，它就会去不断使用指令 TSI，它的传入参数是全局变量 B，当 TSI 返回值为真的时候，该进程就可以安全的进入临界区，当它跳出临界区后的第一件事就是要将 B 重置为 0。如此便完成它运行时的一个循环。

显然，由于 TSI 指令属于原子操作，不可能同时有数量多于一个的进程获得全局变量 B 的值为 0 的情况发生，这样也就杜绝了同时有数量多于两个的进程进入临界区，从而保证了互斥。

（2） 交换指令（EI）

交换指令实现一个寄存器和一个内存单元的内容互换的原子操作。下面来看这个指令是如何来实现多个进程互斥的：

同样，系统需要维护一个全局变量 B，其初始值是 0，而每个进程需要维护一个局部

变量 k。

如果某个进程希望进入临界区，它首先将自己的局部变量 k 设置为 1，然后和 B 不断互换各自的值（这个动作使用交换指令 EI），直到 k 为 0 为止，当 k 为 0 的时候该进程跳出前面的这个循环，直接进入临界区，等它跳出临界区之后第一件事就是使用 EI 指令互换 B 和 k 的内容。如此便完成它运行时的一个循环。

同样，使用 EI 指令，也不可能同时有数量多于一个的进程获得全局变量 B 的值为 0 的情况发生，这样也就杜绝了同时有数量多于两个的进程进入临界区，从而保证了互斥。

综上所述，我们可以看出，互斥控制的硬件方法有着很多优点：比如说，对共享内存的计算机系统都适用；可以方便实现多个进程的互斥控制；比较简单而且容易判定其有效性；能够支持多个临界区，而每个临界区都有一个与之相对应的标志性变量。当然，它也有一些缺点，比如说，“忙碌-等待”机制的存在，导致处理器时间的浪费；死锁情况可能发生。

- 信号量

互斥控制的信号量方法遵循这样一个基本原理：两个或更多的进程可以通过某些简单的信号来相互协调，这样一个进程就可以被强制在程序的某个特定位置停止执行，直到它接收到某个特定信号为止。为了传递这些信号，需要引入一个被称为信号量（Semaphore）的特殊变量。如果某个进程需要通过信号量传递一个信号，那它只需要执行一个发送信号的原语即可；而如果它想要通过信号量来接收一个信号的话，也只需执行一个等待信号的原语就行了；如果对应的信号没有收到的话，该进程就一直在这个特定位置停留，直到它收到指定信息为止。

设定信号量为 *s*，发送信号的原语是 *signal(s)*，等待信号的原语是 *wait(s)*。我们可以得到这样一个有着一个成员变量和两个成员函数的对象：

(1) *s* 为一整体变量，初始值为一非负整形值；

(2) *wait(s)*执行 *s* 减一操作，如果进程执行完该原语之后，*s* 为负，则该进程进入阻塞状态；

(3) *signal(s)*执行 *s* 加一操作。如果执行完该原语之后，*s* 非正，则一个被 *wait(s)*抛进阻塞状态的进程就可以从阻塞状态转为运行状态，继续执行后续操作。

需要提醒注意的有两点，一是对信号量而言，只有上面所提到的两种操作，别无其它；二是上面的两个原语属于原子操作，即在运行过程中是不能被中断的。

上面所讲的信号量是一个一般化定义。事实上还有一个二值型的信号量，其取值只能是 0 和 1。从理论上讲，二值型信号量和一般化形式的信号量功能相同，并且更容易实现。

对于任何一种形式的信号量，都需要有一个队列容纳用于等待信号量的进程。理论上来讲，没有限定究竟是哪个进程首先出列，我们只知道最公平的方式就是遵循先进先出规则。但是，也不能因为优先级问题而让某个进程无限期的呆在队列里。

下面我们来看利用信号量如何实现互斥：

系统需要维护一个全局变量，即信号量 *s*，并初始化为 1。

进程实现互斥的程序流程就是：第一步，执行等待信号的原语 *wait(s)*；第二步，进入临界区；第三步，跳出临界区之后第一个操作是执行原语 *signal(s)*。

下面对该互斥控制算法的有效性进行分析：

如果某个进程需要进入临界区，它首先执行等待信号的原语 *wait(s)*，如前所述，如果此时无进程在临界区中，则在 *s* 减一之后，*s* 为 0，这个时候进程直接就进入临界区；如果此时已经有进程在临界区了，也就是说，已经有进程执行原语 *wait(s)*，信号量 *s* 为负，这个进程就只有进入阻塞进程队列中等待处于临界区内的进程跳出临界区。

当某个进程跳出临界区之后，它第一件事就是执行原语 *signal(s)*，这个时候，如果阻

塞进程队列中有进程的话，系统就会依据一定的选择算法将在阻塞队列中等待的某个进程选中，将其状态从阻塞状态转为运行状态，这样，该进程就可以进入临界区了。如此便完成它运行时的一个循环。

从上面的分析不难看出，在临界区中同一时刻只能有一个进程运行。可见，信号量机制在互斥控制方面是有效的。

- 管程

信号量机制为互斥控制提供了一个比较粗糙但是功能却很强并且很灵活的实现方式。但是，实践证明，采用信号量机制来实现进程间互斥给编程带来一定困难。对于一个比较复杂的程序，可能有多个临界区，从而就需要有多个信号量来对这些临界区进行控制，问题就出来了：虽然对应每个信号量只有两个原语操作，但是多个信号量最终将导致多个操作在程序里面四处散布，这样我们就很难看出这些操作对它们各自作用的信号量的整体作用效果，从而为程序设计和分析带来一定的难度。

管程机制是专为编程语言设计的一个互斥控制方法，它和信号量机制功能相同，但是却更容易控制。该控制方法已经在很多编程语言中实现。甚至已经有了它的程序库，人们就可以很方便地将它插入自己需要有互斥控制的对象或程序中。

管程一般被定义成这样一个软件模块，它拥有自己的内部数据块，一个或多个例程以及一个初始化程序块。管程的主要特征如下：

(1) 内部数据块只能由管程所拥有的例程来访问，而拒绝外部数据访问；
(2) 进程要进入管程，必须是通过调用它的例程；
(3) 一次只能有一个进程进入管程；而其它发出进入管程请求的进程只能延缓进入，直到管程允许为止。

因为管程一次只能让一个进程进入，所以可以保证进程间互斥。管程的内部数据块在同一时刻只能由一个进程访问，因此，某个共享数据就可以被当作管程的内部数据块得到保护。而如果该内部数据块对应的是某个计算机资源，那么管程就可以为进程提供对该资源的互斥访问手段。

而在进程执行过程中，会碰见这样一种情况：当某个进程进入临界区内，同时因为某个条件没有得到满足而挂起。在这种情况下，如果简单套用管程的话，就会造成其它希望进入管程的进程因为该进程而毫无意义地等待。为此，我们需要引入一种机制，使得进入管程的进程不仅可以挂起，而且还可以释放管程以便其它进程能够进入管程。为此，管程内部增加了一些条件变量。管程的具体实现可以有两种，一种是结合信号量的实现；一种是结合通报和广播（Notify and Broadcast）机制的实现（图 2.7）。

由于后一种比前一种有着明显的优势，我们只介绍后一种实现方式。

结合通报和广播的管程实现需要定义两个原语：

(1) 等待原语 *cwait(c)* ：某个已经获得管程的进程需要等待条件 *c* 满足时调用该原语操作后挂起，并列入与条件 *c* 对应的进程队列中，同时释放管程给其它进程；
(2) 通报原语 *cnotify(c)* ：当某个拥有管程的进程调用该原语操作后，对应条件 *c* 的进程队列就会被告知条件 *c* 已经满足，而该进程继续运行。结果，当管程空闲时，该等待进程队列中的某个进程就可以重新获得管程。

下面我们来看多个进程利用这个管程实现的一般程序流程：

如果某个进程想要获取管程，必须在管程入口处排队，根据一定的选择算法（通常是 FIFO 算法），它将被选择进入管程；在进入管程之后，如果不需要等待某个继续执行条件，或者要将某个条件满足的消息通报给对应的等待进程队列，该进程可以继续执行相关操作直到释放管程退出；另外，如果进程在拥有管程时需要等待某个条件，这时候，它会调用等待原语而后挂起并释放管程给其它进程；当某个等待进程队列被告知其等待的条件已经

满足的时候，该进程队列中的某个进程将由挂起状态转为（准备）就绪状态，此时，该进程会不断检测其等待条件的满足情况，一旦管程被释放之后，它就会重新进入管程继续其操作。如此便完成它运行时的一个循环。

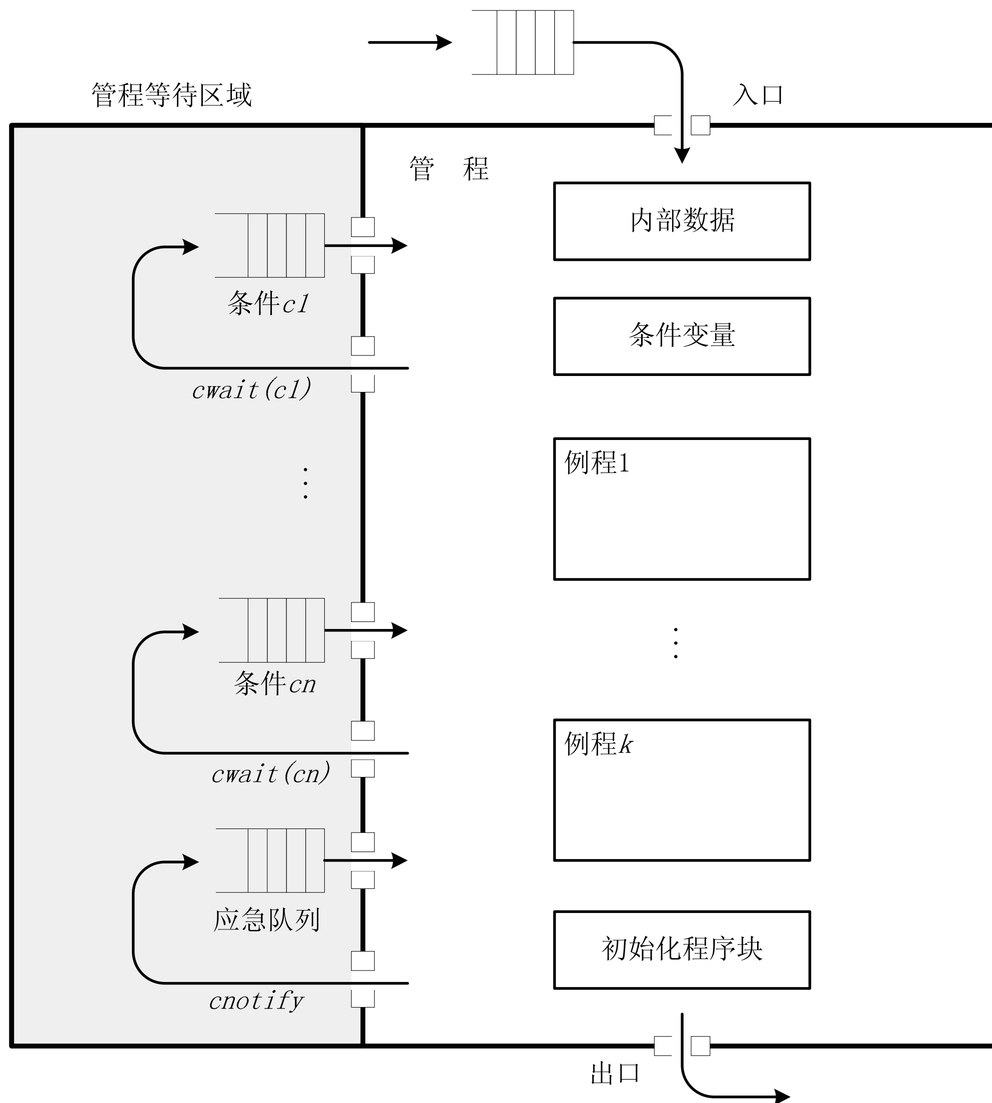


图 2.7. 管程结构

另外为了防止某些进程可能无限期地等待下去，通报原语可以再增加一个操作：给每个等待进程赋一个看门狗时间。这样，当该进程得知其等待条件已经满足并且等待时间已经超出这个时间赋值时，它可以直接继续其后续操作。

另外，使用通报原语可以很方便地被另外一个操作即广播原语 *cbroadcast(c)*代替。*cbroadcast(c)*与 *cnotify(c)*所不同的就是前者会通知与条件 *c* 对应的等待进程队列的所有进程该条件已经被满足，这样，该队列中的所有进程都将从挂起状态转为（准备）就绪状态。

- **消息传递**

当两个进程需要交互的时候，两个基本条件需要满足，即同步和通讯。进程间需要同步以便实现互斥；通过通讯来交换信息。有一种方法可以同时满足这两点要求，这就是消息传递机制。消息传递除了在单处理器和多处理器系统中得到应用之外，还广泛应用于分

布式系统中。

消息传递系统有多种形式，我们在这里对这类系统的典型特征给予阐述。通常，消息传递机制包括两个原语，一个是发送原语，用于向某个指定目标发送特定信息；另外一个接收原语，用于从某个指定消息源接收某个消息，形式化描述为：

send(destination, message)；

receive(source, message)。

消息传递机制的实现需要解决一系列问题，其中包括同步、寻址、消息格式及其排队算法，下面给予分别阐述：

(1) 同步

一个进程只有当另外一个进程向它发送消息之后才能接收到该消息，从而两进程交互必然需要某种程度上的同步。

通常一个进程发送或接受消息时的状态有两种，阻塞和非阻塞。因此其发送或接受方式也就有了阻塞方式和非阻塞方式。

先来看进程发送消息的情况。一般情况下，从进程的执行效率角度来看，我们希望这样一个发送进程在发送消息过程中处于非阻塞方式，这样不仅能够使得它在相同的时间里可以发送多条消息，而且能够完成更多其它的工作。但是有一种情况不能忽视，即如果发生某条消息在传送过程中遗失的情况就会导致进程重复发送消息，如此反复对计算机资源的消耗将会很大。另外，由于非阻塞方式下的发送原语无法保证某条消息会被正确接受，因此，程序员必须要考虑消息接受的认定：进程必须要接收相应的消息来判断它所发送的消息是否被正确接收。

至于某进程接收信息，通常情况下，这样一个进程只有在接收到某条信息之后才能继续其后继动作，因为，对于一个接收进程而言，其最理想的工作方式就是阻塞式，即当它需要接收某条信息的时候就进入阻塞状态，等待信息到来。但是这样一种方式依然存在缺点。比如说，在分布式系统中，可能出现的情况就是消息遗失从而导致该接收进程没有收到它所等待的消息，那样的话接收进程就会无限期等待下去。这时我们可以考虑采用非阻塞接收方式，或者给出等待时间上限。这样的解决方案虽然解决了进程从阻塞状态恢复的问题，但是依然可能导致消息丢失问题，比如说，接收进程执行接收原语操作，然后发送进程执行发送原语操作就会导致消息丢失。还有一个解决方案就是在接收进程执行接收原语操作之前先测试一下是否有消息已经发出，确定之后再考虑是否执行接收原语操作。

(2) 寻址

发送消息需要知道消息的接收方的地址，而接收消息则需要知道消息的提供方的地址，这就涉及到一个消息的寻址问题。

消息的寻址方式多种多样，大致可以分为两大类，一类是直接寻址方式；一类是间接寻址方式。

在直接寻址方式中，消息在发送时需要指明接收方的地址，而消息的接收方式可以随意。而在间接寻址方式中，消息并不是直接发送给接收方，而是发送到某个共享数据结构如小溪队列中，这里可以暂时存放消息；通常这样的数据结构被称为邮箱，发送进程将信息发给改邮箱，而接收进程则从邮箱中取出它所需要的消息。

由于间接寻址方式使得需要交互的进程间的耦合度低，从而为消息传递机制的实现带来了更大的灵活度，因而得到广泛使用。

(3) 消息格式

消息的格式取决于其实现目标和应用环境。通常情况下，消息长度是固定的，这样可以减少一些不必要的开销，如额外的计算或判断。当然也有消息不等长的情况，这个时候消息格式本身可能就复杂些，比如说需要加入有关自身长度方面的信息等等；而且信息的

发送方和接收方也需要对此进行额外的判断。

(4) 消息排队算法

当有多个消息需要接收时，就涉及到该如何选取下一个要接收的消息的问题。一般可以采用先入先出（FIFO）算法来应付。但是如果某些消息紧急，需要提前被接受怎么办？这个时候我们可以考虑赋予每个消息一个合适的优先级，然后，让接收到的消息按照优先级的高低进行排序。另一个可选择的算法就是赋予接收进程察看消息队列的权限，这样它就可以在察看过程中选择它认为最先需要处理的消息。

(5) 互斥

下面我们来看看如何使用消息传递机制来实现进程间互斥。

对于多个需要借用消息传递机制来实现互斥的进程，它们之间消息传递的寻址方式采用间接方式并共享同一个邮箱，该邮箱初始化后仅有一个不含任何内容的消息。

首先我们需要对消息传递机制的两个原语操作作进一步限定：

send(destination, message) ：非阻塞方式。

receive(source, message) ：阻塞方式；

如果邮箱中有一个消息，则它只传递给一个等待进程，而其它进程进入阻塞状态；如果邮箱里没有任何消息，则所有进程都会进入阻塞状态，直到有一个消息的时候，其中某个进程将会被激活并收到这个消息。

下面我们来看在采用消息传递机制时，某个进程的一般程序流程：

首先，进程从邮箱接收消息，执行接收原语操作，如果邮箱中有一个消息，并且该进程接收到这个消息的话，进程直接进入临界区继续运行，当它跳出临界区的第一件事就是将它接收到的这个消息再返回邮箱，这个时候就是执行发送原语操作了。如此便完成它运行时的一个循环。

2.2.2.3. 并发控制：死锁处理

并发控制在有了比较好的互斥算法之后，并不一定能够保证各进程能够合乎逻辑地正常运行，事实上还存在一个非常让人头疼的问题，这就是死锁问题。死锁问题的简单描述就是若干个进程在争夺系统资源或者进行交互的时候陷入永久性的阻塞状态。此时，计算机只是在做无用功。针对死锁问题，没有一个通用的有效解决方案，只是存在一些比较常用的应对方案，比如，死锁的预防、检测和避免。

- 死锁

死锁现象必然要牵涉到若干个进程请求资源时产生矛盾。举个很简单的例子：有两个进程 A 和 B，都需要资源 x 和 y 才能正常完成其任务。只就资源而言，A 的一般程序流程是：获取资源 x，获取资源 y，释放资源 x，释放资源 y；而 B 的一般程序流程则是：获取资源 y，获取资源 x，释放资源 y，释放资源 x。假如两个进程运行开始时，A 和 B 分别获取了资源 x 和 y，在各自完成相应操作之后，开始等待另外一个进程释放自己所需要的资源，但是每个进程又只有当获取了对方占用的资源之后才会释放自己占用的资源，这样的话，两个进程就陷入了无休止的阻塞状态，运行陷入停滞，即出现所谓的死锁现象。

考察各种死锁现象我们不难看出死锁现象发生的三个必要条件：

(1) 进程间互斥：每次只能有一个进程使用某个资源；

(2) 占用并等待：进程已经占用了某个资源并且同时等待其它资源空闲以供占用；

(3) 非抢占：进程所占用的资源只能由进程自己释放，而不能由外力强迫它释放。

事实上在很多情况下，这三个条件都是我们所希望的。比如说条件（1）可以保证进程执行

结果合乎逻辑；而进程在对其所占用资源进行操作的中途如果被强制释放自己所占用的资源，这样不仅可能会给系统运行增加不必要的负担，如需要保存和恢复进程所占用资源的内容等等，而且在某些情况下会导致进程执行过程中的逻辑混乱。上面三个条件只是死锁现象发生的必要条件，而只有当这第四个条件发生的时候，死锁才真正发生：

(4) 循环等待：由于相互需要对方占有的资源，从而形成了一个闭合的等待进程链。

事实上，对条件（4）中描述的情况也正是死锁的一个定义，也是死锁发生的充分条件。正是由于前三个条件的存在，所以当条件（4）具备时，死锁才会无可避免的发生。这四个条件就构成了死锁发生的充分必要条件。

- **死锁的预防**

如果上面所讲到的四个条件无法同时满足，那么死锁现象就不会发生。而这四个条件前三个为死锁发生的必要条件，只有第四个为充分条件，因此，预防死锁发生的方法可以有两类，一类是防止死锁的三个必要条件不会同时发生，即所谓的间接预防方法；另外一类是不让条件（4）发生，即所谓的直接方法。

1） 进程间互斥

一般而言，进程间互斥是进程正常运行所必需的，所以无法废弃。当然，对有些资源的访问，可能在某些情况下可以不需要互斥，比如说多进程对某个文件执行读操作。为了避免死锁的发生，我们应当在程序设计中尽量少使用进程间互斥。

2） 占用并等待

对于占用并等待的情况可以有一个解决方法就是如果某个进程非要一次性占用所有它所需要的资源的话，那就索性让它在那里一直等到所有资源都空闲的时候再发出资源占用申请吧。这样可以避免该进程陷入死锁，但是事实上，却是很不可取的，因为这样可能会让进程进入漫长的等待中。

当然也还有别的解决方法，如引入多线程机制。象上面所提到的进程 A 和 B 竞争资源 x 和 y 的情况。我们可以考虑，A 和 B 都设计成拥有两个线程的进程，各自的两个线程分别用于对两个资源的占用和相应处理。这样，当出现起始时的 A，B 分别占有 x，y，实际上是各自内部的某个线程在分别占用这两个资源。当这两个线程完成相应操作后，会将处理结果送达进程的某个数据区域，然后死掉，同时也会释放自己所占用的资源。这样，两个进程内部的另外两个线程就可以启动去占有刚刚释放的资源，就避免了死锁。

3） 非抢占

在实际应用中对正在占用资源的进程实施抢占式的做法，是可能的，但不见得可行。由于某个进程在对其所占用的资源进行操作时，如果被要求强制释放资源，那它为了在未来的某个时刻重新占有该资源并正常运行的话，就必须涉及到与该资源相关信息的保存和恢复。只有当这样的信息能够很容易保存和恢复的话，对互斥资源实施抢占式的做法才是可行的。

4） 循环等待

循环等待的一个解决方法就是对进程访问互斥资源的顺序有一个限定。假如我们将所有可供进程访问的互斥资源按照先后顺序排成一个队列的话，如果进程已经占用了其中某个资源的话，那当它想申请占有新的资源的话，这新的资源只能来自于排在它已经占有资源后面的那些正在闲置的资源了。

这个解决方法的有效性很容易通过反证法证明。事实上，假如仍然存在循环等待的话，必然存在两个资源，其申请顺序在占有它们的两个进程中是相反的。而这是上述解决方案所不允许的。因此在采用该解决方案之后不可能再出现循环等待的情况。

这种方法的不足之处就在于它限定了进程访问互斥资源的顺序，缺乏一定的灵活性。

- 死锁的避免

在上面讨论的死锁的预防方法中，总是对进程本身的实现或者进程对互斥资源的访问有一定限制，这样的做法会削弱计算机系统的并发功能。为此，人们提出了死锁的避免方法。这种方法可以赋予进程更大的并发性和灵活性。

避免死锁的方法主要有两种，一种是限制进程启动法，即如果某个进程启动后会导致死锁，那么就不要启动该进程；另一种是限制资源分配法，即如果某个资源分配给进程后会导致死锁的话，那就不要分配。而这两种方法都涉及到是否导致死锁的判定问题。它们虽然能够在系统运行过程中对死锁进行动态判定，但是都需要对进程的未来资源请求状况有足够的了解。下面对这两种方法分别作一阐述。

1） 限制进程启动法

假设系统有 n 个进程和 m 种不同类型的资源，并有以下向量和矩阵定义：

$$R=(R_1,R_2,\cdots,R_m),$$
$$V=(V_1,V_2,\cdots,V_m),$$
$$C=\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$$
$$A=\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

其中，R_j 是第 j 种资源的总量，R 称为系统资源向量；

V_j 是第 j 种资源的剩余量，V 称为系统剩余资源向量；

C_{ij} 是进程 i 对第 j 种资源的最大需求量，C 称为进程资源最大需求矩阵；

A_{ij} 是进程 i 实际占有的第 j 种资源的数量，A 称为进程实际资源占有矩阵；

$i=1,2,\cdots,n;\quad j=1,2,\cdots,m.$

显然有以下关系成立：

$$R_j=V_j+\sum_{k=1}^{n}A_{kj};$$
$$C_{kj}\le R_j;$$
$$A_{kj}\le C_{kj};$$

其中，$k=1,2,\cdots,n$，其余下标定义与上同。

有了以上定义，我们可以给出下一个新进程即第（n+1）个进程启动的必要条件是

$$R_j\ge C_{(n+1)i}+\sum_{k=1}^{n}C_{kj}$$

其中各下标定义与上同。

2） 限制资源分配法

限制资源分配法也被称作银行家算法。为便于阐述这个算法，我们首先给出两个定义，一个是系统状态，另外一个是系统的安全状态。所谓系统状态是指在某个指定时刻系统的各个进程对互斥资源的占用情况，反过来讲，也就是互斥资源的分配情况；而系统的安全状态则是指能够让所有进程都能完成其任务并且不会导致死锁的系统状态。系统的不安全状态则是与其安全状态相对的。

限制资源分配法的基本原则就是确保系统永远处于安全状态。如果某个进程要求分配给它一些资源，那我们可以假设其申请全部满足，这样可以相应地刷新系统状态，然后检查系统是否仍然处于安全状态。如果是的话，就满足该进程的资源分配申请，否则挂起该进程，直到能够保证安全地分配给它资源为止。

死锁的避免方法比起它的预防方法来讲，有着很多优越性，当然它也存在一些不足之处：

（1） 需要预先给出每个进程对每种资源的最大需求量；

（2） 所考虑的进程是相互独立的，也就是说它们执行的先后顺序是不受限制的；

（3） 可供分配的互斥资源的数目是一定的；

（4） 当进程占用资源的时候不能退出。

- **死锁的检测**

死锁的预防方法过于保守，主要表现在对互斥资源访问和进程执行的限制上。而死锁的避免方法走向了另外一个极端：它不存在对互斥资源访问和进程执行的限制。死锁的检测方法要优于前两种方法，使用这种方法，互斥资源只要空闲就能被分配给需要它的进程：操作系统会周期性地执行某种算法以便能够发现前面所讲过的循环等待情况。

1）死锁的检测算法

检测算法执行的周期可快可慢，这要看死锁发生的频率到底如何。下面介绍一种比较常见的检测算法。

前面在死锁的避免方法中给出的向量和矩阵 R, V, C, A 依然有效。另外需要给出一个请求矩阵 Q：

$$Q = \begin{pmatrix} Q_{11} & Q_{12} & \cdots & Q_{1m} \\ Q_{21} & Q_{22} & \cdots & Q_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{n1} & Q_{n2} & \cdots & Q_{nm} \end{pmatrix}$$

其中，Q_{ij} 代表进程 i 对第 j 种资源的需求量。

在该检测方法的进程过程中，不会死锁的进程会被标示。在初始阶段，所有进程都未被标示，算法执行的步骤如下：

第一步，标示出矩阵 A 中元素全为零的行所对应的进程；

第二步，初始化一个临时向量 W 等于系统剩余资源向量 V；

第三步，在 Q 中找出某一行使其序号 i 满足条件 $Q_{ik} \le W_k$， $k = 1,2,\cdots,m$ 。如果找

不出这么一行，则算法停止执行；

第四步，如果找到了这么一行，则标示其对应的进程 i 并执行赋值操作 $W_k = W_k + A_{ik}$，

$k = 1,2,\cdots,m$。

当该算法停止执行之后，如果依然还有进程没有被标示，那就说明存在死锁现象。

2）恢复操作

当检测到死锁现象之后，我们需要执行某种恢复操作以便让系统继续有效运行。下面是一些可能的操作，按复杂程度排序：

（1） 放弃所有的死锁进程。这是最常用的方法；

（2） 将所有进程的状态恢复到过去的某个时刻的状态，然后重新启动所有的进程；

（3） 一个跟着一个地退出某个死锁中的进程，直到死锁现象消失；

（4） 对某些资源实行抢占式再分配，直到死锁现象消失；

- 一个综合处理方案

根据前面的阐述，我们可以看出死锁的处理方法各有千秋，都有自己的优缺点。一个很自然的想法是能否把它们融合起来提出一个综合的死锁处理方案。这是可能的，比如下面的一个方案：

（1） 将资源分成不同的类；

（2） 对不同的资源类按先后顺序排成一个线性队列；

（3） 在每一个资源类内部，则采用最适合该类的死锁处理方法。

2.2.2.4. 中断及中断处理

几乎所有的计算机系统都提供了中断机制以便让系统中的其它模块能够中断正在执行中的进程从而完成一些紧急任务，这就是中断。最常见的中断类型主要有四类，即软件中断、定时中断、I/O 中断和硬件故障等（表 2.2）。

中断主要是用来提高计算机系统的处理效率。比如说，多数外围设备的动作速度和处理器相比要慢很多。如果我们需要打印机打印某个文档时，如果没有中断机制存在的话，进程会在每次写操作完成之后等待打印机完成相应操作并告知处理器，而后处理器才能继续下一个指令的执行。但是，如果引入中断机制的话，处理器只需要在执行一些简单指令，比如，初始化打印机，告诉打印机要打印的文档数据在内存中的位置，最后向打印机发送打印指令之后，就可以转到后继指令的执行，而这个时候，打印机正在慢吞吞地执行相关的打印任务；当打印机打印完所交给的任务之后，会采用中断方式，告诉处理器指派的打印任务已经完成，这个时候，处理器会中断当前任务并处理这个中断。显然，在引入中断之后，处理器的利用率提高了，可以在同样多的时间可处理更多的指令或任务。

表 2.2. 中断类型

类型	描　述
软件中断	在程序运行过程中，由某条指令的执行所引起的中断。比如说，溢出、除零、非法机器指令等等。
定时中断	由处理器内部定时器产生。定时中断不仅是计算机计时的基础，同时还是操作系统完成很多复杂操作的前提，比如说，进程的交替执行等等。
I/O 中断	由 I/O 控制器产生。
硬件故障	由于某个硬件故障，如电源故障或内存奇偶检验错误等引起的进程执行中断。

- **中断**

从进程角度看，中断可以看成其正常执行过程的一个小插曲。进程本身无需关心中断如何处理，等中断处理结束之后，进程继续原来正常的执行过程。为引入中断机制，一般的指令周期需要加入中断处理（图 2.8）。

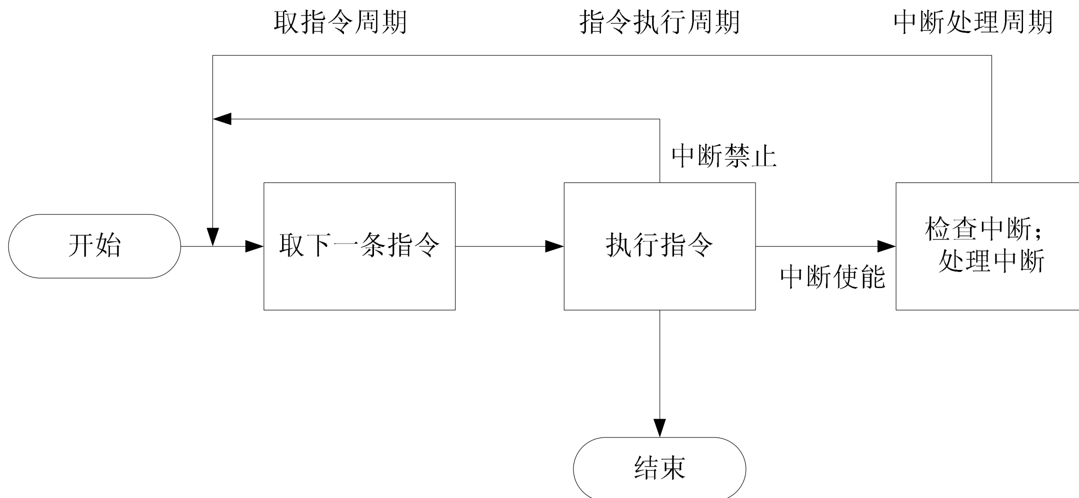


图 2.8. 带中断处理的指令周期

在中断处理周期中，处理器会检查是否有中断发生。如果没有中断发生，处理器继续其正常的指令处理，在进程看来，就是自己正没有丝毫影响地持续运行着呢；如果有中断发生的话，处理器就会跳出其正常的指令处理顺序，转为执行与所发生中断对应的一个中断处理例程。通常，中断处理例程是操作系统的一部分，用于判断中断的类型并采取相应的处理措施。

- **中断处理**

中断会引发一系列的事件发生。中断的一般处理流程如下：

第一步，某个中断源（如软件、I/O、定时器甚至是硬件故障）向处理器发出中断信号；

第二步，处理器在响应中断之前完成当前指令；

第三步，处理器检测是否存在中断，并发给中断源一个确认信号，这个确认信号可以让中断源撤出其中断信号；

第四步，处理器将处理器时间交给中断处理例程之前需要做一些准备工作，这其中主要包括保存在中断点时当前程序的状态。最低限度的保存信息应该包括程序状态字信息和当前程序下一条指令的地址；

第五步，处理器执行中断处理例程；

第六步，中断例程执行完毕后，处理器保存与中断例程相关的必要信息，处理器恢复被中断程序在中断点时的状态；

第七步，处理器继续原来的指令执行过程。

- **多中断**

多中断是指同时有多个中断需要处理的情况。通常有两种基本方法来处理这种情况。一个就是在处理某个中断的时候不能被打断，如果这时有新的中断信号进来，就只能先挂起再说；另外一种方法就是允许中断处理嵌套，也就是说，可以给不同的中断定义合理的优先级，这样当某个较低优先级的中断正在处理时，可以被一个更高优先级的中断打断，从而形成一个嵌套结构。

前一种方法比较简单。但是如果某个中断处理时间过长，就会导致新的中断等待时间过长，即便后来被处理也毫无意义。后一种方法比较可行，但是如果嵌套层次太深的话，实现起来就比较繁琐了。

2.2.2.5. Linux 的进程与中断管理机制

- Linux 进程状态

Linux 进程状态有五种分别为运行态，可唤醒阻塞态、不可唤醒阻塞态、僵死状态和停滞状态（表 2.3）。

表 2.3. 进程状态

进程状态	英文对照	状 态 描 述
运行态	Running	进程正在或准备运行。进程被标示为运行态，可能会被放到可运行进程队列中了。之所以出现这种情况，是因为在 Linux 中标示和入列并非原子操作。可以认为进程处于随时可以运行的（准备）就绪状态。
可唤醒阻塞态	Interruptible	进程处于等待队列中，待资源有效时被激活，也可由其他进程通过发送信号或者由定时器中断唤醒后进入就绪队列。
不可唤醒阻塞态	Uninterruptible	进程处于等待队列中，待资源有效时被激活，不可由其它信号或定时器中断唤醒。
僵死状态	Zombie	进程已经结束运行且释放大部分资源，但尚未释放进程控制块。
停滞状态	Stopped	进程运行停止，通常是由进程接收到一个信号所致。当某个进程处于调试状态时也可能被暂停运行。

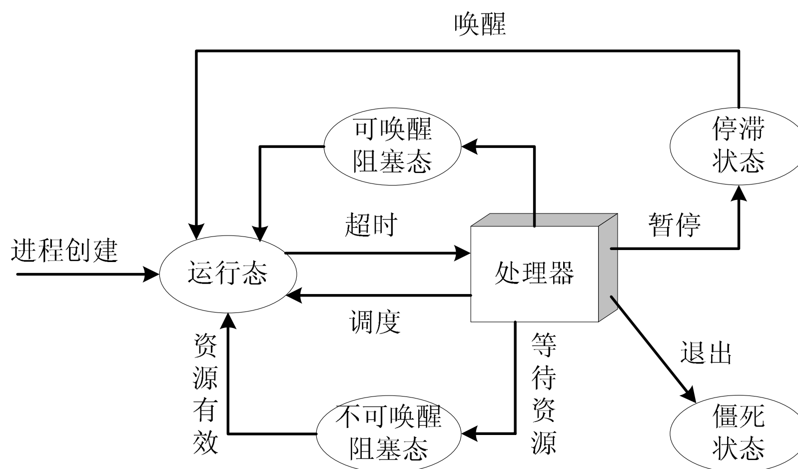


图 2.9. Linux 状态转移图

用户进程一经创建，就开始了这五种进程状态的转移（图 2.9）。进程创建时的状态为不可唤醒阻塞态。当它的所有初始化工作完成之后，被其父进程激活，状态被标示为运行态，进入可运行进程队列。依据一定的进程调度算法，某个处于可运行进程队列的进程被

选中，从而获得处理器使用权。在使用过程中，有四种情况发生：第一种情况是当分配给它的时间片结束之后，该进程会要求放弃其处理器使用权而后回到可运行进程队列中去；第二种情况是进程在运行过程中，需要用到某个资源，但是该资源并非空闲，则进程转为不可唤醒阻塞态，当资源申请得到满足之后，进程会自动转成运行态；第三种情况就是进程因为受到某种系统信号或者通过系统调用转入停滞状态，此时，进程同样会因为某种信号激发而转入运行态；第四种情况是进程自行退出结束其任务，进入僵死状态，等待系统收回它所占有的资源。

- **Linux 进程控制块**

进程控制块是操作系统最重要的数据结构之一，在 Linux 中用一个结构来表示，即 task_struct。在 Linux 早期版本中，进程至多只能有 512 个，自 2.4.0 版本之后，最大进程数由系统的物理内存来决定，并有一个具体算法在系统实时运行时确定。Linux 进程控制块结构很复杂，但是我们可以将它按功能具体分成如下九种。

（1）进程状态：即上面所提到的五种状态的其中一种；

（2）调度信息：操作系统调度时需要的信息以便对进程实施公平调度；

（3）进程标识：需要特别指出的是 Linux 从其安全角度出发，它的进程标识多达八个，不过可分为用户标识和用户组标识两大类；

（4）进程间通讯：Linux 支持的进程间通讯机制不仅包括传统 Unix 系统中的信号、管道和信号量机制，也支持 Unix System V 进程间通讯的共享内存、信号量和消息队列机制；

（5）进程间关联：从进程派生角度看，进程间有父子关系，兄弟关系；而从进程管理角度看，进程往往处于一个进程的双向链表中，进程间有前后关系。进程需要维护相应的指针来阐明这些关系。

（6）时间和定时器：进程需要维护其创建时间，从而决定分配给它的处理器时间片的消耗情况；另外，进程在发送信号等方面需要和定时器打交道，因此需要维护与进程相关的内部定时器；

（7）文件系统：进程在运行期间会打开某个文件，因此需要维护它所打开文件的相关信息；

（8）虚拟内存：进程一般都要用到虚拟内存，Linux 内核需要相关信息来跟踪进程对内存的使用状况；

（9）处理器相关信息：各个寄存器的内容等。

- **Linux 内核同步机制**

1) 非抢占

Linux 内核是非抢占式的，具体表现在三个方面：

首先，任何进程如果以内核模式运行的话，除非它自愿交出处理器的控制权，否则，任何其它进程都没法打断它的运行；

其次，一个进程正在内核模式下运行，这个时候，中断或异常处理可以中断它的正常运行，但是当处理完中断后，该进程将重获处理器控制权；

第三，中断或异常处理过程只能被中断或异常处理中断。

2) 原子操作

防止某个操作被中断的最简单的方法就是将这个操作通过硬件技术用一个指令来完成。在执行过程中不能被中断的操作就被称作原子操作，它是实现很多更复杂更灵活的互斥控制机制的基础。

3) 中断禁止

当某段可执行代码过于冗长，而不能用原子操作来实现的时候，我们就需要考虑更好

的互斥控制算法。中断禁止方法是互斥控制的一个主要方法。它能够确保硬件中断不会对内核模式运行下的进程造成干扰，从而实现互斥。

当然仅仅有中断禁止，并不一定就能保证这样的进程运行不受影响。比如说，一个“缺页”异常就可以挂起该进程。

4) 内核信号量

如果想要保护互斥资源的话，一个想当然的方法就是给这个互斥资源加上一把“锁”，这样其它进程就没法访问，而等互斥资源访问完毕就解除这把“锁”，这样其它进程就可以访问该资源。在 Linux 中，这样的“锁”有两种，一种是内核信号量；一种是自旋锁。前者在单处理器系统和多处理器系统都能使用，而后者则只能用在多处理器系统中。

在 Linux 中定义在信号量上的两个原子操作分别为：

(1) 减一操作 down()：当进程希望访问互斥资源的话，它调用该操作，信号量的值减一；
(2) 加一操作 up()：当进程访问互斥资源完毕，它调用该操作，信号量的值加一。

在 Linux 中信号量是由一个结构来表示，即 semaphore，在初始化时，它的值初始化为 1，当然也可以初始化为其它的正整数，那样能允许多个进程同时访问互斥资源。这个结构主要有两个成员：

(1) 一个整型变量 count：也就是信号量的值，如果该变量值非负的话，进程可以访问互斥资源，该变量值的改变只能由上面的两个原子操作来完成；
(2) 一个等待进程链表指针 wait：如果某个希望访问互斥资源的进程在执行完减一操作之后发现信号量的值为负值的话，进程将会挂起，并进入该链表。

5) 自旋锁

自旋锁的思想就是在不断循环中坚持反复尝试获取一个资源（一把“锁”），直到成功为止。这通常是通过类似 TSI 机器指令的操作进行循环来实现的。自旋锁最重要的特点就是进程在等待“锁”被释放时一直占据着 CPU。一般而言，只能在极短的操作过程中才使用自旋锁。特别是决不能在阻塞操作中持有锁。甚至为了保证尽量短时间地占有锁，可在取得自旋锁以前阻塞当前处理器的中断。

自旋锁的基本前提是进程在某个处理器上忙等（busy and wait）一个资源，而另外一个进程在不同的处理器上正使用这个资源，这只有在多处理器系统中才可能。在单处理器系统中，如果一个系统试图获取一个已被占用的自旋锁的话，就会陷入死循环。而多处理器算法对于任意数目的处理器都应该适用。这要求进程必须严格遵守一个规则，那就是当它持有自旋锁时决不能放弃对处理器的控制权（对于 Linux 而言，就是决不能在释放自旋锁之前调用系统的进程调度调度函数）。在单处理器系统中，这样就可以保证进程不会陷入死循环。

- **Linux 进程间通讯**

前面已经提到过 Linux 支持多种进程间通讯机制。我们这里只看其中比较重要的五种，这就是信号（Signal）、管道（Pipe and Named Pipe）、信号量（Semaphore）、消息队列（Message Queue）和共享内存（Shared Memory）。

1） 管道

管道允许两个进程进行生产者/消费者模式的数据通讯。它是一个先入先出（FIFO）队列，一个进程从队列的一端不断地写入数据，而另一端则是另外一个进程不断地从中读出数据。

当一个管道被创建时，它由一个固定的大小。当管道还有空间时，写进程就可以不断地向其中写数据，否则要么退出，要么转入阻塞状态，直到管道由省与空间为止；相对地，当管道中有数据的时候，读进程就会不断地从中读取它所需要的数据，如果其中没有数据

的话，读进程要么退出，要么进入阻塞状态等候有新的数据可读。

管道可分为两种类型，一种是非命名管道，即我们通常所说的管道；另外一种是命名管道。一般的管道只能供存在“血缘”关系的进程共享；而命名管道则没有这个限制。

2） 消息队列

消息是一个有着特定类型的字符块。而消息队列则是在 Linux 内核中维护的一个消息链表。对于每一个消息队列都有一个唯一的队列标识。在 Linux 中，一个队列的初始化是由函数 msgqueue_initialize 来完成；而新消息的入列则是调用函数 msgqueue_addmsg 来完成；从某个队列取出一个消息调用函数 msgqueue_getmsg。

消息在发送时会表明其类型，而消息的接收者可以依据 FIFO 规则或者只根据类型来接收消息。当消息队列已经满员了的时候，需要向该消息队列发送消息的进程就只好挂起等待了。而当一个进程试图从一个空消息队列中读取消息的时候依然会挂起等待，但是，一个进程在试图读取业已存在的某个特定类型消息时如果失败了，该进程则不会因此挂起。

3） 共享内存

进程间通讯最快的方式就数共享内存机制了。共享内存，顾名思义，就是一个内存区域由多个进程共享。进程在读写该区域的时候和对其它内存区域进行读写时采用的方式没什么两样。当然，对每个进程而言，它对这块区域的访问权限不见得是一样的，有的可能是只读的，有的或许是只写的，而有的则是可读可写的。

需要明确一点的是，互斥并不见得就是共享内存机制必须提供的，但是在进程访问共享内存区域的时候则必要有互斥的存在，也就是说，访问进程应该提供互斥控制。

4） 信号量

这里所讲到的信号量和前面所提到的内核信号量比较相似，比如，都用信号量值的增减来决定互斥资源的访问权。但是，用于进程间通讯的信号量机制要比内核信号量更复杂更灵活：

首先，每个用于进程间通讯的信号量都有一组信号量值，而不是如内核信号量那样的只有一个值，这样就可以同时保护若干个相互独立的互斥资源了。需要注意的是，当信号量在初始化时需要指明其拥有的信号量值的数目。

第二，进程间通讯的信号量机制引入了故障保护功能：如果一个进程死掉了，却没有及时恢复它所占用过的信号量的话，这些信号量会在故障保护机制的作用下，自动恢复到它的初始状态。这样可以防止使用同一个信号量的其它进程陷入死锁。

下面，我们来看看使用进程间通讯的信号量机制的一般程序流程：

第一步，通过系统调用 sys_semget()来获得信号量标识；

第二步，通过系统调用 sys_semop()来对相关信号量值执行检测和减一操作。如果所有的检测成功的话，执行减一操作，退出该调用之后，申请进程被允许进入相应的互斥资源。如果检测发现，某些互斥资源正在使用中，申请进程通常会刮起等待直到这些互斥资源被释放为止；

第三步，当使用完互斥资源之后，再次通过系统调用 sys_semop()来对所有相关信号量值执行加一操作；

第四步，这是一个可选择操作，就是通过系统调用 sys_semctl()将上面用到的信号量从系统中清除。

5） 信号

信号是一种通过软件方法通知进程某个异步事件已经发生的进程间通讯机制。它和硬件中断比较类似，只是所有信号没有优先级之分。

信号在内核内部或者进程间传递。一个信号的传递是通过刷新对应进程表的某个变量来实现的。信号的处理可以在进程被唤醒之后，也可以是在它刚从某个系统调用返回时。

表 2.4. Linux 信号

值	名 称	描 述
1	SIGHUP	挂起；当内核认为用户进程在无效运转时就向它发出该信号
2	SIGINT	键盘中断
3	SIGQUIT	键盘退出
4	SIGILL	非法指令
5	SIGTRAP	调试断点
6	SIGABRT / SIGIOT	异常终止
7	SIGBUS	总线错误
8	SIGFPE	浮点异常
9	SIGKILL	进程强制终止
10	SIGUSR1	用户定义信号 1
11	SIGSEGV	非法段操作；进程企图访问在其虚拟地址空间之外的地址
12	SIGUSR2	用户定义信号 2
13	SIGPIPE	向一个没有接收进程的管道数据
14	SIGALRM	实时钟
15	SIGTERM	进程终止
16	SIGSTKFLT	协处理器栈错误
17	SIGCHLD	子进程停止
18	SIGCONT	继续运行
19	SIGSTOP	停止进程运行
20	SIGTSTP	从 TTYT 发出的进程终止信号
21	SIGTTIN	后台进程需要输入
22	SIGTTOU	后台进程需要输出
23	SIGURG	Socket 紧急
24	SIGXCPU	处理器时间片耗完
25	SIGXFSZ	文件大小超长
26	SIGVTALRM	虚拟计时器钟
27	SIGPROF	描述计时器钟
28	SIGWINCH	窗口改变大小
29	SIGIO /SIGPOLL	I/O 空闲
30	SIGPWR	电源故障
31	SIGSYS /SIGUNUSED	未使用
32	SIGRTMIN /SIGSWI	软件中断

Linux 进程对具体某个信号作何反应是由其进程控制块中的信号屏蔽机制和处理函数选择机制来决定的。由于 Linux 的信号屏蔽机制是通过一个 32 位变量的位掩码来决定的，所以 Linux 最多只能接受 32 种信号（表 2.4.），对于被屏蔽掉的信号，进程将不会对它的发生有任何反应；而其处理函数可以是自定义函数，也可以是系统的缺省处理函数。

- **Linux 中断与定时服务**

计算机系统中断与定时的一般机制前面已经讲过。我们在这里主要讲 Linux 在实现过程中的几个概念：bottom half、tasklet 和 softirq。

1） bottom half

在处理硬件中断的时候，一般是要关闭中断允许的，以免再次中断。问题是，如果关闭的时间过长，就有可能失掉重要的外部中断信号。因此，关闭中断的时间不宜太长。但有的时候某个中断处理过程占用时间会很长。为了解决这个矛盾，Linux 采用了将中断处理例程一分为二的办法，即分为 top half 和 bottom half 两部分。

通常，top half 读取必要的数据，并保存在某个特定的缓冲区中，通知 bottom half 后即退出。中断处理例程的这一段程序代码的执行不能被中断，但是因为时间很短，系统是可以忍受的。而剩余工作则会交给 bottom half 部分在适当的时候完成。而这一部分的执行过程中系统就可以继续接收新的中断了。

2） tasklet

tasklet 是 Linux 2.4 版本引入的一个新概念，我们可以把它理解为一种多线程的 bottom half 机制。它与 bottom half 的主要区别在于不同的 tasklet 可以同时运行在不同的处理器上，这样就可以更加有效地利用多处理器的计算能力了。

3） softirq

softirq 与 bottom half 和 tasklet 联系紧密。它与 bottom half 的区别是，softirq 是支持多处理器的，与 tasklet 的不同是，一个 tasklet 只能在一个处理器上运行，而 softirq 则没有这个限制。

2.2.3. 调度机制

在多道程序设计系统中，操作系统需要为各个进程合理分配各种资源，从而使得不仅不会让它们的运行相互产生冲突，而且还希望能对这些资源进行充分利用。在计算机系统中，最宝贵并且最容易引起各进程竞争的资源就是处理器。如何公平合理地分配给各进程足够的处理器时间，并且尽可能提高处理器的利用率同样是操作系统原理研究的重要内容。而这个任务是由操作系统的调度机制完成的。

调度机制一向都是学术研究的热点。人们已经提出了很多这样那样的调度算法，现在主要集中在多处理器调度上，同时，还有进程的实时调度方面。

2.2.3.1. 调度类型

前面已经讲过了，调度是多道程序设计的关键，我们可以将调度大致分为四种类型（表 2.5.），其中 I/O 调度我们留在后面的 I/O 设备章节阐述。这里主要阐述单处理器调度，多处理器调度和实时调度都会遇见的三种类型，即长期调度、中期调度和短期调度。

长期调度在一个进程创建时起作用，它将决定哪一个程序会被系统选中并创建一个新的进程来运行它；而中期调度则是决定进程在内存中的换入换出，其主要机制在内存管理章节已经有阐述，即归根到底中期调度是一个内存分页的换入换出机制问题；短期调度则

是真正决定哪个等待进程该获得处理器控制权的问题，这就是我们通常所说的调度，包括单处理器调度（Uniprocessor Scheduling）、多处理器调度（Multiprocessor Scheduling）和实时调度（Real-Time Scheduling）等。调度通过影响进程的行为来影响系统的性能。

表 2.5. 调度类型

调度类型	描　述
长期调度	决定是否产生新的进程
中期调度	决定是否将进程调入内存
短期调度	决定处理器该执行哪个进程
I/O 调度	决定一个 I/O 空闲设备该处理哪个进程的 I/O 请求

下面我们来着重看看短期调度及其评价标准：

通常是当某个事件的发生导致需要挂起某个进程或者需要某个进程以抢占式方式替换另外一个正在运行中的进程以取得处理器控制权的时候，短期调度就被激活。这样的事件可以是时钟中断、I/O 中断、操作系统调用或者信号等。

短期调度的衡量标准是双重的，我们可以按照是用户相关还是系统相关来分，也可以按照是否是性能相关来分。

用户相关的标准主要是以用户的主观感受为准，比如说响应时间就是一个很典型的用户相关的评价标准；而系统相关的标准主要是看处理器的使用效率等因素。比如说计算任务的吞吐量。在不同的系统中，这两种评价标准所占有的地位显然是不一样的。比如说，在单用户系统中，我们评价一个调度算法的好坏主要看系统的响应时间，而不会去关心处理器的利用效率有多高。

表 2.6. 短期调度算法的评价标准

	性能相关	非性能相关
用户相关	（1） 响应时间：对于交互式进程而言，它是指从提交请求到返回结果之间的时间间隔； （2） 周转时间：一个进程从创建到完成任务之间的时间间隔； （3） 截止时间：当某个进程完成任务有个截止时间的话，操作系统应当尽可能满足这个要求。	可预测性： 一个给定的任务应该能在相同期限内，并且在相同资源消耗的情况下完成，而不管系统的负荷是否有变化。
系统相关	（1） 吞吐量：调度算法应该在单位时间里完成尽可能多的计算任务； （2） 处理器利用率：处理器忙所占的百分比。	（1） 公平性：如果没有用户或系统干预，每个进程应当被平等对待； （2） 强制优先级：如果进程的运行被赋予优先级，那么调度程序应当能够遵循这个优先级的规定性； （3） 资源负荷平衡：系统资源应当得到充分应用。

系统性能相关的标准主要是一些可以量化的、并且通常是可以很方便测量出来的，如反应时间和吞吐量。而非系统性能相关的标准则是一些本质上不能量化的或者不能很方便测量出来的量，如可预测性等。表 2.6 给出了按照这些标准的组合，一共有 9 项。

2.2.3.2. 单处理器调度

业已提出的单处理器调度算法主要包括这么几种：FCFS（First Come First Served）、SPN（Shortest Process Next）、SRT（Shortest Remaining Time）、HRRN（Highest Response Ratio First）、反馈算法（Feedback）和循环执行算法（Round Robin），下面分别给予阐述。

- **FCFS 算法**

FCFS 算法是一种非常简单的调度算法，通常也被称为 FIFO（First In First Out）。当一个进程转为（准备）就绪状态的时候，它就被加到一个等待进程队列的队尾。当某个正在运行中的进程停止运行时，根据 FCFS 调度算法，队首的一个进程，也是等待时间最长的哪个进程将会出列并开始运行。

该算法是非抢占式的。

- **循环执行算法**

循环执行算法的基本指导思想是，将处理器时间分成一个一个的时间片（Time Slice），从等待进程队列中选择下一个运行进程的方法和 FCFS 一样，所不同的是，每一个被选中的进程一次所占有的处理器时间顶多是这么一个时间片，而后，下一个被选中的进程将会以抢占式的方式来顶替该进程取得处理器的控制权。而将处理器时间分片是靠定时器中断来实现的，即每当一个时间片耗完之后，会有一个中断产生，告知调度程序，然后调度算法就被激活，选取下一个要运行的进程，开始新一轮循环。

循环执行算法有一个缺点就是它在处理 I/O 操作密集型进程和处理器使用密集型进程时会带来的资源使用的不平衡。I/O 密集型进程会导致 I/O 设备忙，而处理器使用却很少；而处理器使用密集型进程则会让处理器连轴转，而 I/O 设备却往往处于空闲状态。这会导致该算法在进程间的公平性和资源使用的平衡度方面表现比较差。于是就有人提出了虚拟循环执行算法（图 2.10.）。

虚拟循环执行算法的进程创建、入列以及被选中运行部分与 FCFS 算法别无二致。所不同的是，加入了一个辅助进程队列。这样，当一个原来因为 I/O 操作挂起的进程现在如果它申请的 I/O 操作已经完成的话，它不是进入普通的等待进程队列，而是进入辅助进程队列。当一个新的处理器时间片来临的时候，辅助进程队列里的进程比普通的等待进程队列里的进程优先获取处理器控制权，而其运行时间将不会超过它在上一次自己获取的时间片里用剩下来的时间。

这两个算法都是抢占式的。

- **SPN 算法**

SPN 算法是在一个进程运行完毕之后，选择需要最短运行时间的进程运行。

SPN 算法实践起来的一个难点就是如何对进程的运行时间进行估量。可以通过程序员来进行估量。但是这样做并不保险，因为可能会出现偏差。现在比较常用的方法就是通过实验的方法，多次重复得到一组时间值，要么直接取平均，要么取加权平均，以便获得较为客观的运行时间估计。

SPN 算法实施起来有一个危险，就是因为不断会有运行时间较短的进程出现在等待队列中，因此，某些运行时间较长的进程可能会永远得不到处理器时间片。解决方法也不是没有，比如说可以对每个进程可等待的时间做一个上限，当这个上限值达到之后，不管三七二十一，它就得运行。

该算法是非抢占式的。

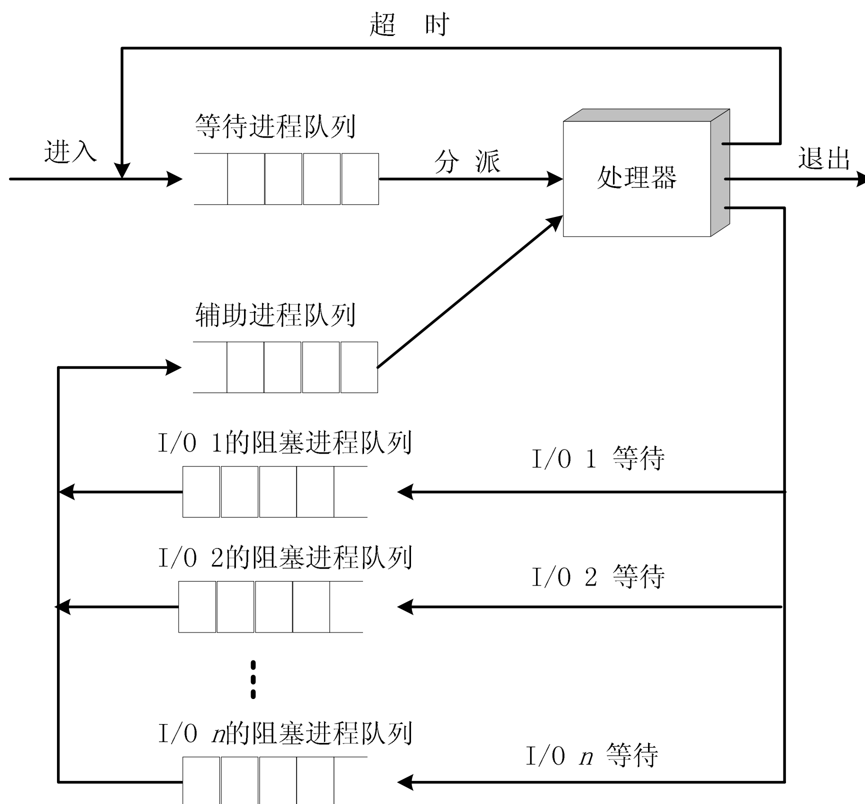


图 2.10. 虚拟循环执行算法

- **SRT 算法**

SRT 算法可以看作 SPN 算法的一个抢占式版本。在这个算法中，调度程序总是选择剩余时间最短的进程运行。当一个新的进程加入到等待进程队列时，它可能比现在正在运行的进程所需要的剩余时间更短，这个时候，调度程序就会中断现在正在运行的进程转而执行新加入的这个进程。和 SPN 算法类似，SRT 算法仍然需要估算进程运行所需时间。同样也面临需要较长运行时间的进程很难得到处理器时间的问题。

- **HRRN 算法**

HRRN 算法的基本思想就是调度程序选择下一个运行进程时遵循最高响应时间比率原则。下面我们给出响应时间时间比率的计算方法：

$$R = \frac{w+s}{s}$$

其中，w 是进程等待获取处理器控制权所耗费的时间，

s 是进程运行所耗费时间预期。

- **反馈算法**

如果我们事先不知道各进程运行时的时间参数，如，运行耗费时间、等待时间等等，这个时候，SPN、SRT、HRRN 等算法是没法使用的。反馈算法的提出就是希望能够通过对进程运行已经耗费的时间进行统计，从而决定下一个运行进程该是谁，其指导原则就是过去运行总时间最短者最先得到处理器时间。

反馈算法的一个比较简单的实现可以是：当一个进程被创建时，它进入优先级最高的等待队列，每一次运行完一段时间后，它会被放入一个优先级更低的一个等待队列等待下一次运行，如此递推，直到它执行完任务为止。这里需要明确一点的就是，优先级高的等

待队列里的进程比优先级较低的等待队列里的进程优先得到处理器控制权。从而使得运行时间比较短的进程可以更早得完成任务。

2.2.3.3. 多处理器调度

多处理器系统按照各处理器之间的关系一般可以分为两类：

(1) 松耦合型：这样的系统通常是一些自治子系统的组合。每个子系统都有自己独立的处理器、内存和 I/O 和操作系统；

(2) 紧耦合型：这样的系统通常是一组处理器共用一个内存，并处于一个操作系统的控制之下。

而紧耦合型多处理器系统仍然可以进一步划分成两种，其中一种是主/从型，即其中一块处理器属于通用型，功能比较强；而另外的处理器则属于专用型，功能相对单一，两种类型的处理器组成主从关系。

另外一种紧耦合型多处理系统是对等型系统，即各处理器在功能等方面是对等的，没有差别。我们后面要讲的多处理器调度算法就是针对这样一种对等型紧耦合多处理器系统。

- 总体设计

多处理器系统与单处理器系统相比给调度算法的设计带来几个新问题，一般需要考虑三方面的事情：按处理器分配进程问题、单个处理器上的多道程序设计问题、进程实际分派问题。不管是哪个问题，我们需要明白的就是具体算法的设计是同具体应用的特点以及处理器的个数是相关的。

(1) 按处理器分配进程

如何将进程的运行和具体的处理器对应起来是多处理系统调度算法首先要考虑的问题。在对等型紧耦合多处理器系统中，最简单的调用方法就是将这多个处理器统看成一个处理器资源库，当有进程需要使用处理器时，就从这个库中拿出相应的资源分配给它。这样，处理器的分配方法就有两种，静态方法和动态方法。

处理器分配的静态方法是指当某个进程一旦分配到某个处理器上运行之后，在它的整个生命周期中，该进程不会再更改处理器了。静态分配方法比较简单，这是它的优势所在，但是却有一个缺点，就是它会经常导致各处理器负荷不均衡，也就是说，某些处理器可能一直忙着，而有的处理器则可能正在那里睡大觉呢！

而处理器分配的动态方法则允许进程在其运行过程中可以选择不同的处理器。由于在紧耦合多处理器系统中，处理器共享内存，进程的各类信息对各个处理器都是可见的，迁移代价是可以忽略的。

而进程间的地位并非是平等的。有些是核心进程，有些只是一般的用户进程。不同种类的进程如何去占用处理器呢？主要有两种方式。一种是主/从方式，在这种方式中，操作系统的核心部分总是运行在一个特定的处理器上，而另外的处理器只运行用户进程。这种方式比较简单，但是问题是，如果这个特定的处理器出现故障的时候，会导致整个系统的崩溃，而且主处理器可能会存在性能瓶颈。

另外一种方式就是对等方式：操作系统可以在任何一个处理器上运行，并且每一个进程自己从处理器资源库中选择处理器以实现自动调度。这种方式比较复杂。

事实上很多系统的实现是取这两种极端方式的折衷。

(2) 单处理器上的多道程序设计

当采用处理器分配的静态方法时，一个新问题就随之而来：各个处理器是否应该采用多道程序设计？在单处理器系统中，我们注意到如果不采用多道程序设计的话，处理器可

能会因为进程在长时间等待某个 I/O 操作而造成计算资源的浪费。在多处理器系统中是否也一定会需要有这样的考虑呢？

在传统的多处理器系统中，各个处理器之间要求同步的频率很低，或者说各处理器之间相对独立地工作着，在这种情况下，每个处理器在功能上等同于一个单处理器系统，这个时候我们希望它们能够采用多道程序设计方式运作，这样能够显著提高处理器的利用率。但是如果各处理器之间的联系是属于紧耦合型，需要不断地进行同步，这个时候，是不是要采用多道程序设计就不好说了。当有很多处理器存在的时候，我们所主要关心的事情并不是某个或者某些处理器的利用率，而更关心的是这个系统整体上的性能。当一个应用程序是多线程的话，只有当所有线程都在运行时，程序的性能才会有显著提高；反之，可能会表现很次。

(3) 进程分派

首先我们需要明确一点的是，在单处理器系统中性能较好的调度算法在多处理器系统并不见得就一样能取得好的性能。我们知道，在单处理器系统中，基于优先级或者基于反馈的调度算法比起先到先运行这样的简单算法来讲，性能更优越。但是在多处理系统中，可能一些简单的算法会更好一些，而类似基于反馈的这样的算法反而可能会导致系统性能的下降。

另外，需要注意的是，在多线程调度中，还有一些新的问题需要考虑。

- 进程调度

在很多传统的多处理器系统中，进程并不是直接分配给处理器：所有的处理器统一起来形成一个计算资源库，进程按照一定的规则，比如说 FIFO 或者基于优先级等等，形成一个或者若干个队列，按照某种调度算法，这些进程会被一一选中在这个计算资源库中的某个特定的处理器上运行。不管怎么讲，我们都可以将这样的系统称为提供给进程队列的多路计算服务器。

研究表明，调度算法的差异所带来的多处理器系统性能上的差异不是很明显，并且随着处理器数目的增加，这种影响会变得越来越小。

- 线程调度

现在针对多处理器系统的线程调度是学术界的一个研究热点，产生了很多方案。这其中有四种方案比较突出，即负荷共同承担法（Load Sharing）、组调度法（Gang Scheduling）、处理器指定分配法（Dedicated Processor Assignment）以及动态调度法（Dynamic Scheduling）。下面分别给予介绍。

(1) 负荷共同承担

在这个调度方法中，进程并不会分配给某个特定的处理器，而是所有等待运行的线程用一个统一的等待队列来维护。当某个处理器空闲时，它会从这个队列中按照一定的规则选取一个线程来运行，而不需要去管这个线程到底是属于哪个进程的。

采用这个线程调度算法主要有这么三个优点：

其一，系统的计算负荷在所有处理器中可以很方便地实现平均分布；

其二，不需要有一个中央级的调度程序存在，当某个处理器空闲时，调度程序就在这个处理器上运行并选择下一个需要运行的线程；

其三，该调度算法和单处理器调度算法存在天然联系，因此，单处理器调度算法可以很容易推广到这里来使用。

但是，这个调度算法也存在一些缺点，比如说，由于等待线程队列统一管理，对于多处理器而言，必须要有互斥控制。如果同时访问该队列的处理器数目较多的时候，就会产生性能上的瓶颈；而且，所有线程统一管理就会导致一个应用程序的所有线程不大可能同时都获得了处理器控制权，这样就会导致程序性能上的损失。

(2) 组调度

组调度就是将一些相互存在耦合的线程同时一对一地指定给同样数目的处理器并运行。组调度策略存在一些优点，比如说，对于几个相互关联的进程，如果并行处理的话，就会减少同步阻塞，减少进程切换，从而提高性能；另外，由于一个调度决策会同时影响一组进程，从而也降低了调度程序执行的频度。

(3) 处理器指定分配

处理器指定分配算法是组调度策略的一个极端情况，其主要思想是，在一个应用程序的生命周期中，将一组处理器都分配给该程序。也就是说，让这个程序的执行进程的每个线程在其生命周期中都能够占用一个处理器。尽管说有的时候某个线程为了等待 I/O 操作的完成进入阻塞状态使得它所占用的处理器处于空闲状态造成计算资源的浪费。在处理器数目较多的情况下，应用程序的性能比单个处理器的利用率更重要；并且由于这种方式可以减少进程切换，从而提高应用程序的性能，所以，采用这么一种调度方式在多处理器系统中还是不错的。

(4) 动态调度

在某些应用中，我们可以借助一些编程语言工具或者系统工具对其中进程所拥有的线程数目做实时修改。这样我们就可以在进程运行过程中实时调整系统的负荷，从而使系统资源得到最大限度的利用，这就是动态调度的基本思想。

2.2.3.4. 实时调度

实时系统逐渐成为人们研究的热点。而其中的操作系统或者更确切的说，操作系统中的调度部分是实时系统的重中之重。小到实验室设备，大到航空航天，实时系统得到了广泛的应用。

那么什么是实时系统呢？人们一般认为，实时的特征有两点，那就是系统不仅需要给出合乎逻辑的计算结果，而且其处理时间还需要满足一定的要求，比如说不能超过某个截止时间等等。我们可以把实时系统分为两类，一类是硬实时（Hard Real-Time）；另外一类则是软实时（Soft Real-Time）。所谓硬实时是指如果系统对某个实时任务的处理未能在某个截止时间开始或者结束的话，最终的结果将是灾难性的，这就意味着即便是处理结果合乎逻辑但是仍然毫无意义；而在软实时系统中，处理任务启动或者结束的截止时间只是一个期望值，并不见得必须满足；即便是处理时间超过了截止时间，也是有意义的。

另外我们还可以按照处理任务的特点将实时系统划分为周期性（Periodic）和非周期性（Aperiodic）两种。一个非周期性的任务启动或者结束有一个截止时间必须要满足；而周期性任务则是指处理任务需要在某个时间周期里面完成处理。

- **实时操作系统特点**

实时操作系统与普通操作系统的区别主要表现在五个方面，这就是其任务处理的确定性、响应灵敏度、用户参与控制、可靠性以及故障保护措施上。

实时操作系统应当能够做到这一点：实时任务处理的开始时间和结束时间应当是确定的、可预测的、这在实时处理上显得非常重要。从这方面来讲，在实时操作系统中，系统的吞吐量和确定的任务处理相比，要次要的多，实时操作系统的任务处理具备一定的确定性。

衡量系统的确定性有一个比较好的指标，就是系统从接到要求处理的中断和对应的处理任务启动这两个事件发生的时间间隔。一般操作系统这个时间参数很大，而且可能会有几个数量级的变动。而在实时操作系统中，它应当很小，并且比较稳定，有一个上限值。

和系统的确定性相关联的是系统的响应灵敏度，它可以定义为系统响应请求的时间。它所关心的是，系统在确认任务请求之后，需要花费多长时间来处理完相关的计算任务。不难看出，系统的确定性和系统的响应灵敏度共同构成了系统对外界时间的反应时间。

用户参与控制是实时操作系统不同于一般操作系统的又一重要特点。在一般操作系统中，用户所能做的至多只能是希望系统如何如何，而最后系统完成的如何，那就看系统自己的了。而实时操作系统就不同了：当用户发出控制指令后，系统应严格按照指令办事，并且用户可以实现对系统尽可能多的控制，甚至于到系统的调度算法上。

实时操作系统为了完成一些对处理时间敏感的计算任务，必须要强调可靠性。对于一般的系统而言，某个运行错误可能导致的系统重启或者处理能力下降等问题对系统本身影响并不大。而在实时系统中，这样的错误可能是灾难性的。

计算机系统在运行过程中，难免会出现某些运行故障。由于这些突发故障可能会带来灾难性后果，我们就更关心一旦故障发生之后，应该怎么办的问题。一般实时系统都有比较完备的故障保护措施，用于在系统发生运行故障之后，尽可能多地保护系统的运行结果，从而为系统的再恢复打下基础。另外，有些系统会有冗余设计，从而降低系统运行的故障率。

- **实时调度（Real-Time Scheduling）**

在考察实时调度算法时，我们可以从三个角度对各种各样的调度算法进行分类：首先看系统是否做调度分析；如果有的话，然后就看是静态分析还是动态分析；最后看这个分析结果是否直接影响到进程的实时分派。据此，我们可以将各种实时调度算法大体上分为四类，即 ST 类、SPP 类、DP 类、DBE 类。

ST（Static Table-driven）类调度算法就是根据相关信息做出一个切实可行的实时调度计划表，并在系统实时运行时严格按照这个表来调度进程。这类方法比较适合周期性任务。调度分析所需要的信息包括周期性到达时间、执行时间、周期性结束截止时间以及各个任务的优先级等。调度算法希望能够制定出一个进程调度计划表来满足所有周期任务的要求。这个方法是确定性的，可以预测实际运行效果，但是却不灵活。并且如果某个任务的特征有所改变的话，就需要整个调度计划表重新编制。

SPP（Static Priority-driven Preemptive）类调度算法也是需要做出一个静态调度计划表来，但是这么一个表只是在赋予各个任务优先级的时候提供参考，在系统实际运行时，按赋予任务的优先级来进行抢占式调度。这类调度算法的典型代表就是后面要讲到的 RMS 调度算法。

在 DP（Dynamic Planning-based）类调度算法中，一个实际有效的调度计划不是在系统运行之前制定的，而是在运行中动态规划的；一个新的计算任务只有当它提出的要求能够被满足时才会被系统接受并处理，调度算法将决定何时会分派该任务。

DBE（Dynamic Best Effort）类调度算法中，不需要做任何调度分析，系统竭力去满足所有计算任务所提出的要求，并且会毫不留情地将不可能满足要求的进程丢弃掉，即便这样的进程已经被启动了。这类算法是很多商业化实时操作系统所广泛应用的，比较容易实现。但是有一个缺点，就是对任务的截止时间等参数无法预知，这样就会在一些没法满足条件的任务身上花费不必要的处理器时间。

- **时限调度（Deadline Scheduling）**

目前大多数的实时操作系统都有一个相同的设计目标，那就是获得尽可能快的中断处理和任务分派速度。事实上，这对实时操作系统的性能并不见得有益。实时操作系统更关心的是能否在规定的时间完成处理任务，而不会受系统负荷变化的干扰。

近些年来有一些很好的实时调度算法出现，只是这些算法需要提供关于任务本身的更多的信息，这其中包括：

(1) 任务准备运行时的时间；
(2) 任务开始处理的时间上限；
(3) 任务完成处理的时间上限；
(4) 任务处理所耗费的时间；
(5) 资源需求；
(6) 任务的优先级；
(7) 一个计算任务最好能够分成两个子任务：一个是迫切需要尽快处理的部分，这部分一般有一个硬性的时限规定；一个是有更多选择余地的部分。

进行实时调度时，如果考虑各种时间上限的话，我们就需要认真回答几个问题，比如说，下一个供调度的任务是谁？我们选择何种抢占方式？

对于第一个问题的回答，我们给出一个不仅是在单处理器系统而且在多处理器系统中都成立的一个结论，这就是，下一个供调度的任务其时间上限与调度时刻离得越近，最终未能满足时间上限要求的任务所占的比例就越少。

另一个问题是关于抢占方式的选择。如果我们强调的是任务开始处理的时间上限，那么一个非抢占式的调度方法就很有效。在这种情况下，获得处理器控制权的任务可以在它处理完自己追切需要尽快处理掉的部分之后进入阻塞状态，从而使得其它任务得以尽快启动。

- **RMS 调度（Rate Monotonic Scheduling）**

RMS 调度算法前面已经提过，它采用静态调度法赋予各个任务在实时运行时所需要的优先级，各个任务在实际运行依照这个优先级来实现运行时调度。而优先级的指定是以任务的周期时间为基础的。显而易见，RMS 方法是面向周期性处理任务的。

图 2.11 给出了周期性任务的相关参数。一个周期性任务的任务周期是指这样一个任务相邻两次重复处理的开始时间的间隔；对应的处理频度（赫兹）则是这个周期的倒数（周期单位取秒）。其中处理时间是指周期性任务的实际处理所占用的时间。

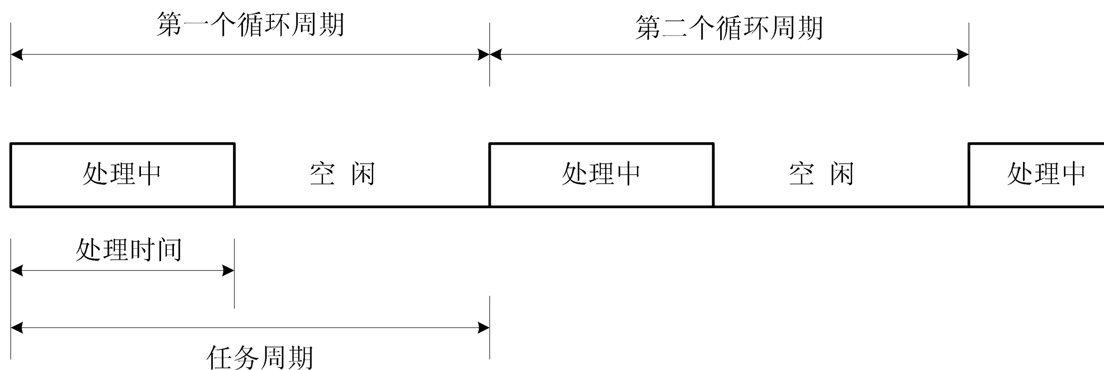


图 2.11. 周期性任务时间参数

在 RMS 调度算法中，任务的优先级的高低与其任务周期的长短成反比，也就是说，一个任务的任务周期越短，它的优先级就越高。按照优先级的规定，如果有多个任务等待处理，那么总是任务周期最短的那个先开始处理。

一个判别周期性调度算法是否有效的判据就是看是否所有的硬性截止时间要求都被满足了。假设有 n 个周期性任务，每一个都有一个固定的周期和一个固定的执行时间。如果要使这个判据可能成立的前提就是下面这个不等式必须成立：

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\le 1$$

其中，C_k 是第 k 个任务的处理时间，

T_k 是第 k 个任务的任务周期，$k=1,2,\cdots,n$ 。

也就是说如果一个周期性调度算法能够满足所有的硬性截止时间的要求，那么所有任务对处理器的利用率的总和应该不会超过 1 这个上限，而对于一个具体的算法而言，这个上限会更低一些。比如说，对于 RMS，就有下面这个不等式成立：

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\cdots+\frac{C_n}{T_n}\le n(2^{1/n}-1)$$

不等式中各个变量和下标的意义与上同。

2.2.3.5. Linux 的调度机制

谈到 Linux 的调度机制，我们一般指的是 Linux 普通版本的调度机制。事实上，由于 Linux 的开放源码，更多的调度机制得以方便的引入，并与 Linux 的一般调度机制和谐共存，这里值得一提的就是在 Linux 上实现了实时调度机制的 RT-Linux 的双内核机制。我们首先阐述 Linux 的一般调度机制，而后对 RT-Linux 的实时调度机制给予详细阐述。

- **Linux 的一般调度机制**

Linux 属于典型的多用户多任务操作系统。它采用分时技术，进程交替执行，实现所谓的“假并行”。它主要有三种调度算法，一个是基于优先级的循环执行法，二是 FIFO 算法，三是传统的基于优先级的循环执行法。前两种调度算法都是软实时的，而第三种则并非实时的。

Linux 进程的优先级是动态变化的。调度程序周期性地检查进程的工作状态并对其优先级进行修改。这样，一个长时间没有取得处理器控制权的进程将会被赋予较高的优先级，而对一个长时间占用处理器时间的则赋给较低的优先级，从而让各进程在计算资源的占用方面获得平衡。

Linux 进程分为实时和普通两种。由于 Linux 同时对这两种进程进行调度。为了保证实时进程总是先于普通进程执行，实时进程总是赋给更高的优先级。普通进程的优先级赋值是 0-999，而实时进程则至少是 1000。

需要特别指明的是 Linux 核心是非抢占式的，只能实现软实时，普通的 Linux 内核是不适合硬实时应用。

- **RT-Linux**

为了保持原有 Linux 的强大功能，这其中包括，网络连接、用户界面等，同时又能够满足硬实时应用的要求，新墨西哥州立大学的 FSM 实验室提出了用虚拟机（Virtual Machine）技术改造 Linux 内核的思想。具体做法是，运行一个简单的实时调度模块，将 Linux 内核作为这个模块控制下的一个任务，而其它用户级别的实时任务具有比 Linux 内核更高的优先级。实时调度模块的调度算法是基于优先级的抢占式调度方法，速度快，系统在满足硬实时应用方面有很好的效果。

1） RT-Linux 的双内核架构

在 RT-Linux 中，Linux 内核作为一个任务运行在一个小的实时内核之上的（图 2.12）。事实上，对这个实时内核而言，Linux 只是在没有实时任务运行的前提下才获得处理器控制权的。Linux 任务不能阻塞中断，对自己的运行与否没有决定权。所有这些实现的核心技术就是在 RT-Linux 中实现了一个硬件控制设备的软件模拟器，即采用了所谓的虚拟机技术。

当 Linux 发出中断禁止请求时，实时就会截获这个请求，记录下来，并假装对这个请求做出回应，而实际上 Linux 禁止硬件中断的请求是不被实际硬件理会的。不管 Linux 的内部状态如何，它都要为实时任务的处理让路。当一个中断信号到来的时候，实时内核会截获这个中断，并作相应的处理：如果有一个与该中断对应的实时中断处理例程的话，该例程即被激活；如果该例程希望这个中断能和 Linux 内核共享，或者根本就没有这样一个实时处理例程的话，中断就会被挂起。当 Linux 请求该中断时，一个对应的软件模拟的中断信号就会送给 Linux 内核，同时，相应的硬件中断重新使能。

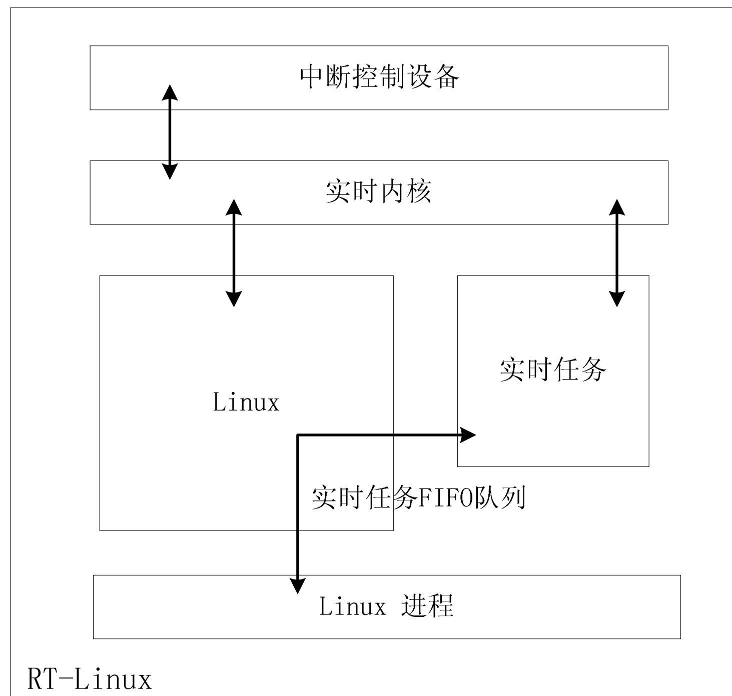


图 2.12. RT-Linux 的双内核架构

从这里实现的虚拟机技术我们可看出，不管 Linux 的工作状态如何，不管它是否对实际硬件中断设备发出了禁止或者使能请求，这些都不会影响 RT-Linux 实时内核对中断控制设备的控制。原因只有一个，因为，Linux 只是和一个由实时内核虚拟出来的硬件中断设备打交道。而真正和这些硬件设备打交道的只是 RT-Linux 的实时内核。

2） RT-Linux 的实时调度算法

RT-Linux 的设计是高度模块化的。实时调度算法的实现也不例外。系统提供缺省的实时调度模块，比如说实时 FIFO 调度算法。如果这个算法不能满足具体要求，我们可以很方便地将它替换成满足要求的调度模块。现在，RT-Linux 有两个可供选择的实时调度算法：一个是 EDF 算法，即最早达到截止时间的实时进程最早运行；另外一个就是 RMS 算法。

2.2.4. I/O 设备

操作系统的核心内容是内存管理、进程管理等。而 I/O 设备作为计算机系统与外界的交互和通讯媒介，其管理和控制也不容忽视。而且，由于 I/O 设备千差万别，这就给操作系统的 I/O 设备管理和控制造成了不少的难度和麻烦。

在 I/O 设备中，最重要的也许就数磁盘了。磁盘作为操作系统和各类应用程序以及数据文件的存储介质，其管理和调度的效率同样是操作系统性能的重要影响因素。

I/O 设备尽管很多，但是我们可以根据其信息传递的发送者和接收者的不同来将其分为三类，即人机交互型、计算机系统内交互型、通讯类型。其中人机交互型 I/O 是指用于人机交互的 I/O 设备，如，显示器、键盘、鼠标或者打印机；而对计算机系统内交互型 I/O 而言，所传递信息只是计算机内部可识别的，如，磁盘驱动器、传感器、控制器等；而通讯类型 I/O 则用于计算机系统与外部设备的交互，这样的 I/O 设备包括调制解调器、网卡等等。

2.2.4.1. I/O 设备描述参数

I/O 设备千差万别，总结起来，可以看出 I/O 设备在六个方面显示出各自不同其它 I/O 设备的特征，在 I/O 设备管理和控制中需要考虑到：

(1) 数据传输速率　不同 I/O 设备之间的数据传输速率可以相差几个数量级。比如说现在的磁盘传输峰值速率可以达到 100Mbps，而键盘输入数据传输速率可能慢到几分钟才一个字节。
(2) 功能应用　I/O 设备的功能界定会影响到操作系统的相关设计。 比如说，一个磁盘 I/O 需要操作系统具备文件系统管理功能。
(3) 控制复杂度　打印机可能只需要很简单的控制界面，而磁盘控制界面可能就更复杂些。如何能够隐藏这些复杂性，使得使用简便，是操作系统需要考虑的事情。
(4) 传输数据的计量单位　是块传输方式，还是流传输方式，这都会对操作系统的设计产生影响。
(5) 数据编码方式　不同的 I/O 设备有着不同的编码方式，这其中包括字符含义的规定和不同的奇偶校验等。
(6) 出错情况　I/O 设备在运行过程中出现的错误及其报告方式、错误所导致后果以及可能的反应都因 I/O 设备的不同而不同。

2.2.4.2. I/O 技术的演变

和计算机系统其它软硬模块存在一个演化的过程一样，I/O 技术也在不断地发展演化。从早期的由处理器直接控制到现在的 I/O 模块自成一体，I/O 技术的演变总共可以分为六个阶段：

第一阶段，处理器直接控制外围设备；

第二阶段，可编程 I/O 模块引入，处理器与某些 I/O 设备工作的具体细节有了某种程度的分离，不过这阶段，处理器需要在进行 I/O 操作过程中挂起等待直到操作完成；

第三阶段，在可编程 I/O 模块的基础上引入中断，这样，处理器可以在 I/O 设备完成

相应操作的同时去执行其它指令；

第四阶段，DMA 技术引入，这个时候，I/O 模块可以通过 DMA 技术和内存直接打交道而不需要通过处理器才能完成每一次的 I/O 读写；

第五阶段，I/O 模块功能进一步增强，可以看作一个专用的处理器。而中央处理器得到了进一步解放；

第六阶段，I/O 模块功能再一次提升，开始拥有自己的独立的内存。

从以上发展趋势我们不难看出，I/O 技术的演变过程实际上也是一个计算机系统中央处理器从 I/O 繁重的事务处理中不断得到解脱的过程。虽然说，I/O 技术已经发展到了第六阶段，但是，值得注意的是，直接内存访问（DMA）技术仍然在计算机系统 I/O 技术中扮演着非常重要的角色。

其中第五阶段的 I/O 模块被称为 I/O 通道，第六阶段的 I/O 模块被称为 I/O 处理器。一般地，两者可统称为 I/O 通道。

2.2.4.3. I/O 设备逻辑描述

前面我们已经看到，I/O 设备在六个方面各有特色，并且 I/O 技术也不断在演化，可能出现即便是同一种 I/O 设备，却用不同技术实现的情况。这就给操作系统的 I/O 设备管理与控制带来了难度。一般而言，在这方面，操作系统设计需要考虑两方面的问题，一是效率问题，因为 I/O 设备是系统性能的瓶颈所在，效率就尤显重要；二是抽象程度问题，我们总是希望所有的 I/O 设备都能有一个统一的接口便于上层系统和应用的操作，因此，需要对各 I/O 设备进行某种程度甚至是多个层次的抽象以便统一管理和操作。

多道程序设计在某种程度上可以提高效率，它可以让某些进程在等待 I/O 处理的同时，别的进程仍然可以继续运行。然而，即便是这样，仍然会有瓶颈存在，那就是当进程在内存和磁盘之间交换（swap）的时候，需要涉及到对磁盘的频繁访问，而磁盘访问本身就是一个很典型的 I/O 操作，我们将在后面章节详细阐述相关内容。在这一小节中我们主要讨论与抽象问题有关的 I/O 设备的逻辑描述。

I/O 设备逻辑描述本质上是一个层次化描述。通过对较低一层的抽象和某些细节的隐藏，从而达到对上层提供更抽象更一般化的操作界面，从而达到对各类 I/O 设备的某种抽象层次上的统一。

按照这种层次化思想，我们可以根据各类设备的不同特点抽象出适合它们的逻辑描述形式。在图 2.13.中给出了最常见的三种。

我们先来看看其中最简单的一种，就是本地外围设备 I/O 的逻辑描述，如图 2.13（a）：

(1) 逻辑 I/O：逻辑 I/O 模块将 I/O 设备看作一个逻辑资源，而不需要去考虑具体某个细节怎么实现的，它提供用户进程所需的一些一般性的 I/O 功能接口，从而能够让用户进程只需要进行类似打开、关闭、读、写这样的简单命令就可以对 I/O 设备进行操作。

(2) 设备 I/O：在设备 I/O 层，各种操作申请和一些数据被转换成一连串的 I/O 指令、通道命令以及控制指令。这里，缓冲技术可用来提高效率。

调度与控制：I/O 操作的排队和调度在这一层完成，同时也包括对操作的控制。在这一层，中断被处理，I/O 状态得以收集和报告，软件和硬件才真正打交道。

对于一个通讯设备而言，如图 2.13（b）所示，其逻辑描述形式和本地外围设备的很相像。最主要的不同点就是逻辑 I/O 模块被换成了通讯架构模块。而这个模块本身可能就包含很多层，比如说 TCP/IP 等等。

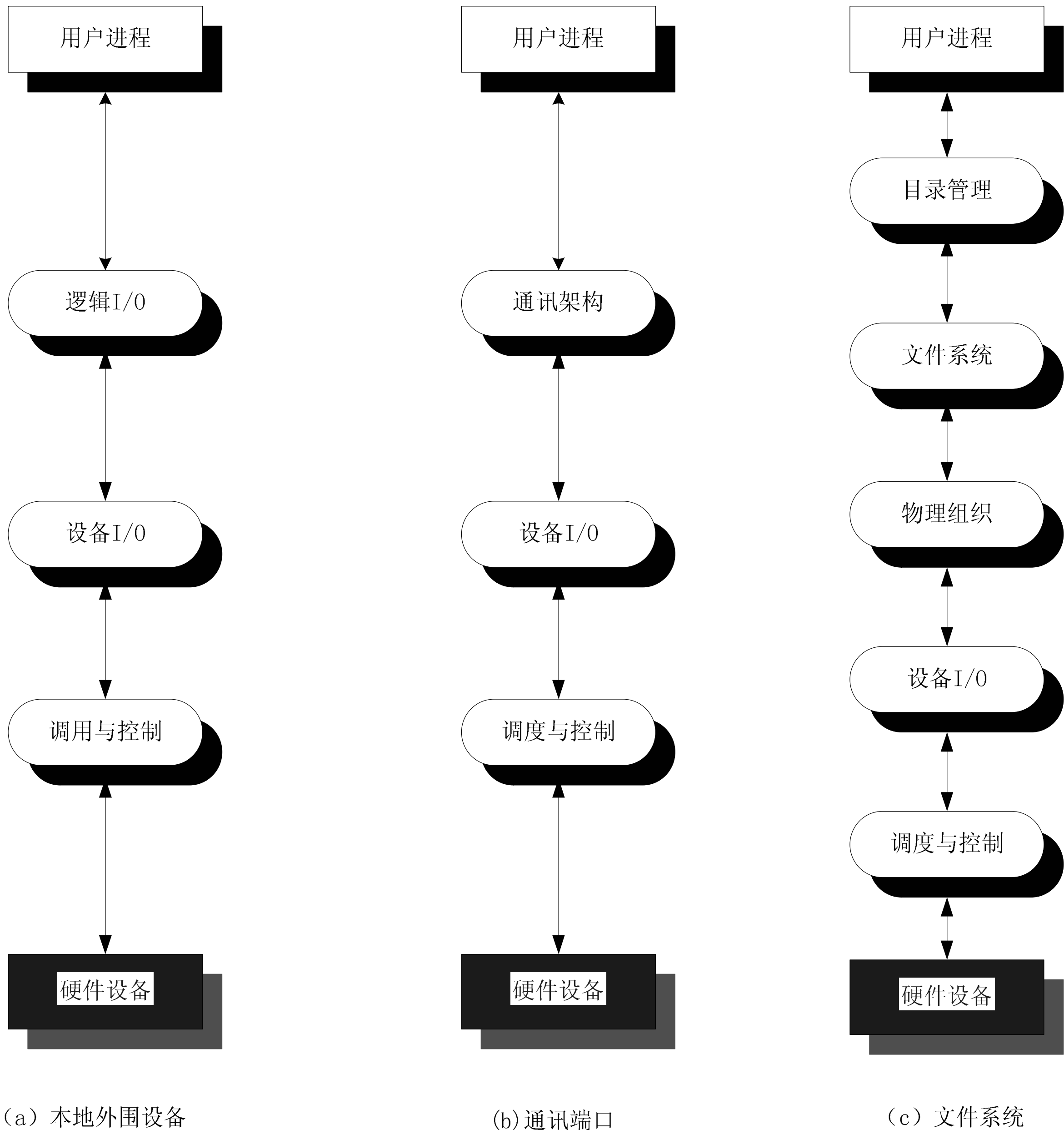


图 2.13. 一个 I/O 逻辑描述形式

图 2.13（c）给出了支持文件系统的外围存储设备 I/O 的一个很典型的逻辑表达形式。它有着本地外围设备 I/O 所没有的三个层次：

(1) 目录管理：在这一层，以字符串形式给出的文件名被转换成一个通过文件描述符或者索引表格直接或间接指向相应文件的标识。这一层也处理用户进程所发出的某些针对文件目录的操作，如创建、删除、重新组织等等。
(2) 文件系统：这一层负责文件的逻辑组织，并且会处理用户发出的某些操作申请，比如打开、关闭、读、写等。而访问权限管理也在这一层实现。
(3) 物理组织：对文件和记录位置的逻辑表达必须在适当时候转换成外围存储设备中的物理地址，外围存储空间的分配需要合理管理，这些都是物理组织层的任务所在。

2.2.4.4. I/O 缓冲技术

我们知道，现代操作系统引入了虚拟内存管理，这为进程管理和程序编写带来了极大方便。现在我们来考虑在一般情况下，进程需要完成对某 I/O 设备的读写操作时会发生什么情况。

先来看进程在其虚拟内存中间完成读写操作情况。当进程需要和 I/O 设备交换信息的时候，进程需要在其虚拟内存空间保留一块儿内存区域，来容纳相应的信息，这就有可能导致死锁：当进程发出 I/O 操作申请之后，如果进程挂起，并在 I/O 操作完成之前被换出物理内存，进程就会进入阻塞状态以等待 I/O 操作完成，同时 I/O 操作也会进入阻塞状态以等待进程换入物理内存的情况。。因此在读写操作过程中，这块区域所在的分页应当始终处于物理内存中，从而这个 I/O 操作就会影响正常的虚拟内存管理。

为了避免这种情况的出现，可以引入 I/O 缓冲技术，以此来提高操作系统性能。为便于讨论，可以将 I/O 设备分成两类，一类是流传输设备，也称为字符设备，在这类设备中，传输数据的最小单位是字节，并且只允许按顺序访问，一般不使用缓冲技术；另外一类是块设备，其传输数据的基本单位是可寻址的块，大多数块设备允许随机访问，而且常常采用缓冲技术。

(1) 单缓冲

最简单的 I/O 缓冲技术就是单缓冲（Single Buffering），如图 2.14（b）。当用户进程发出一个 I/O 操作请求，操作系统就会在操作系统内核所拥有的物理内存空间开辟一块连续存储区域用作缓冲内存。

对块设备而言，一个典型的 I/O 读操作包含两步：首先，将一个单位的信息块读入缓冲内存，而后，这个信息块被转移到用户进程空间。这样做比起无缓冲的情况主要有两个优越性，首先可以提高操作系统效率：由于信息块读取的分步进行可以带来操作的并行化；并且，进程的虚拟内存管理不再会受其 I/O 操作的影响，避免出现死锁等麻烦。与此相适应，操作系统本身的设计需要增加新的内容。

假设每读入一个信息块的时间是 T，而相邻两次 I/O 读申请的时间间隔是 C。我们可以对无缓冲技术和单缓冲技术的性能做一个粗略比较：可以计算出无缓冲时，每个信息块所耗费时间是 $T + C$，而在单缓冲情况下，所耗费时间则是 $\max(C,T) + M$，其中 M 是将缓冲内存的信息块转移到用户进程空间所耗费的时间，通常，这个时间很少。不难看出，在单缓冲情况下，系统性能会有很大提高。

(2) 双缓冲

对单缓冲的一个改进就是将缓冲内存块增加一个变成两个交替读入信息块，如图 2.14（c）所示。这样当用户进程正从某一个缓冲内存块转移数据时，操作系统就可以同时从 I/O 设备将新的信息块读入另一个缓冲内存块。

对于块设备而言，我们同样可以得出在双缓冲情况下，大致估算出读入每个信息块所耗费的时间是 $\max(C,T)$。如果 $C < T$ 的话，双缓冲技术可以提高快设备的利用率，即单位时间其传入的信息量将会增加；而如果 $C > T$ 的话，双缓冲技术则可以在某种程度上减少用户进程等待新数据块读入的次数。

(3) 循环缓冲

我们已经看到，双缓冲在某种程度上使得信息在 I/O 设备间的流动更加顺畅。但是如

果进程在短时间内需要不断读取信息的时候，双缓冲技术也就不够用了，我们需要引入循环缓冲技术。

在循环缓冲技术中，操作系统内存空间开辟有若干个缓冲数据块，这些数据块最终首尾相接形成一个循环链表结构，如图 2.14（d）所示。在操作系统不断地从 I/O 设备读入

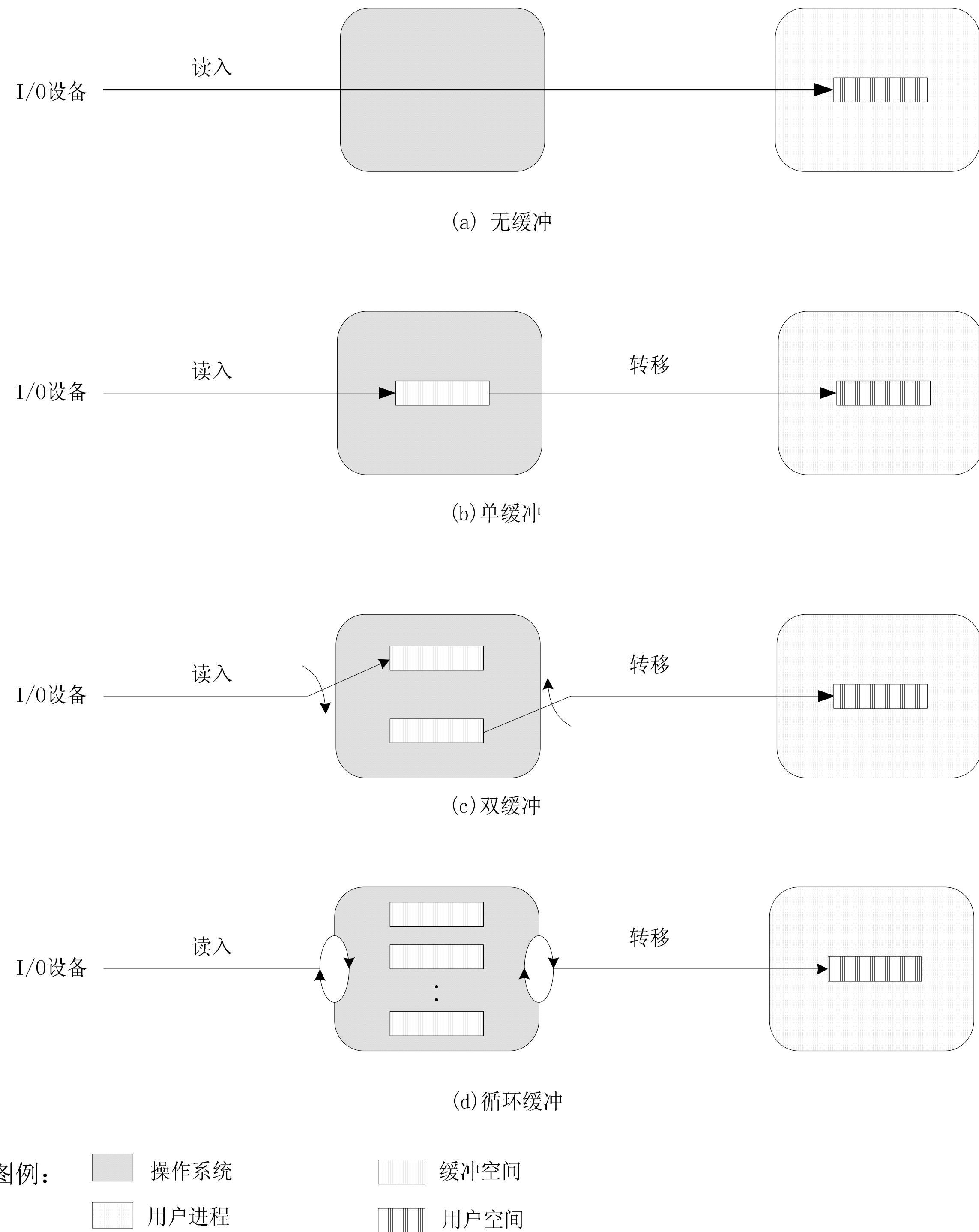


图 2.14. I/O 缓冲技术（I/O 读）

信息的同时，用户进程也在不断地转移已读入的信息。如此循环往复。只要缓冲数据块的大小和个数选择恰当，我们不仅能够最终保证进程等待的时间降至最低，而且还能保证数据读入的先后顺序的正确性，而后者在字符设备缓冲中很重要。

上述三种缓冲技术都可以用于流设备的情况。这个时候，流设备读入信息可以以字节为单位，也可以以某个特定长度的信息块为单位，比如说以屏幕一行的信息量为一次性读写量对信息进行读写操作。

2.2.4.5. 磁盘调度

在过去三十多年里，处理器和内存的速度增长远远超过了磁盘访问速度的增长，现在处理器的速度已经达到了 GHz，而磁盘访问速度至少比它要低两个数量级。而且这种趋势似乎要保持很久。而我们知道，计算机操作系统，应用程序和各类数据大多是保存在磁盘中，而且虚拟内存管理中的进程的分页往往是在内存和磁盘之间倒来倒去。磁盘管理的性能往往是计算机系统性能的瓶颈。磁盘管理的性能受磁盘性能参数和磁盘调度算法的影响，同时也受文件系统的影响。我们先来看磁盘参数和磁盘调度算法。

- **磁盘性能参数**

磁盘 I/O 操作的具体细节取决于计算机系统、操作系统、I/O 控制模块特征以及磁盘控制电路等因素。图 2.15 给出了磁盘 I/O 的一般传输时序。

图 2.15. 磁盘 I/O 传输时序

当磁盘驱动器工作时，磁盘以一个恒定的速度高速旋转，现在普通的 PC 机硬盘的磁盘旋转速度可以达到 7200RPM，随着磁盘技术的发展，这个速度会更高。当需要访问磁盘的某个区域时，磁头会首先移动到相应盘面的相应磁道，然后等待磁盘旋转直到所需要的扇区到达磁头为止，这个时候就开始进行数据传输了。磁头从开始位置移动到相应磁道所耗费的时间通常被称为寻道时间（Seek Time）；而后等待磁盘旋转到所需扇区所耗费的时间被称为旋转延迟时间（Rotational Time），或者也称为潜伏时间（Latency Time）。而寻道时间和旋转延迟时间的总和通常被称为访问时间（Access Time）。而数据传输所耗费的时间被称为传输时间（Transfer Time）。

其中，寻道时间 Ts 的计算方法如下：

$$Ts = m \times n + s$$

其中，m 为跨越每个磁道的平均时间，

n 为需要跨越的磁道的数量，

s 为启动时间。

而旋转延迟时间与转速有关。比如对于一个转速 7200RPM 的硬盘，盘面旋转一周的时间大约为 8.3ms，那么平均而言，旋转延迟时间就是 4.2ms。

传输时间 T 的计算方法如下：

$$T = \frac{b}{r \times N}$$

其中，b 为要传送的字节数，

r 为磁盘转速

N 为每磁道字节数

除了访问时间和传输时间外，还有若干个和磁盘 I/O 操作相关的时间参数。当进程提出 I/O 申请后，它通常是要进入一个等待进程队列以等待设备空闲。而如果设备与其它磁盘驱动器共享 I/O 通道的话，这个时候，仍然需要一个等待通道空闲所耗费的时间。

从上面给出的磁盘性能参数及其计算方法可知，如果要想提高一个特定磁盘的传输速率，我们只可能在访问时间上下功夫。一个理想的情况就是，需要访问的数据正好在磁头下方，如此，旋转延迟时间为零；并且所有数据都按照访问顺序放在同一磁道，并且正好在磁头所在的磁道，这样子的话，可以取得最高的传输速率。如果希望磁盘能够提供较高的传输速率，我们则需要磁头的位置距离下一个待访问数据尽可能近，这其中不仅包括磁道距离，而且也包括扇区距离。磁头的这个距离带有随意性，因此，磁头总是要做频繁移动，从而使得访问时间大大增加。

- 磁盘调度算法

在多道程序设计计算机系统中，每一个 I/O 设备通常会有多个 I/O 申请形成队列。在这种情况下，如何选择下一个要处理的 I/O 申请对磁盘传输速度有很大影响。而每一种选择策略就是一个磁道调度算法。

如果我们选择是随意的，那么我们每次访问的磁道的分布也具有随机性，从而导致磁头寻道时间大大增加，导致磁盘传输性能大大降低。

一个很公平的选择方法就是先入先出规则（FIFO）。即最先提请的 I/O 申请最早得到处理。这种选择策略在只有少数几个 I/O 申请并且所有申请访问的扇区属于扎堆的情况下，能取得比较好的性能。但是如果 I/O 申请很多，并且访问的磁盘区域比较随机的话，我们需要寻找更好的选择策略。下面介绍六种这样的选择策略。

（1） 遵从 I/O 申请进程优先级原则（PRI）

对于一个基于优先级的操作系统而言，磁盘调度将受制于系统的进程调度规则。这也是从系统整体性能来考虑的。

（2） 遵从进程的后进先出原则（LIFO）

有的时候，最先处理最新到达的任务往往会取得一些令人意想不到的结果。比如说在事务处理系统中，将设备提交给最新用户往往能减少磁头的移动量。

FIFO、PRI、以及 LIFO 调度原则都是仅仅从 I/O 请求的申请者的角度提出来的。如果我们知道任何时刻磁头的位置的话，那么我们可以从访问目标的角度出发来设计一些算法。下面将给予阐述。

（3） 遵从最短服务时间最先处理原则（SSTF）

SSTF 调度原则在选择下一个待处理的 I/O 请求的时候，会首先选择一个起始访问位置与磁头所在位置相距最近的一个 I/O 申请，这样可以保证每次磁头总能花费最少的寻道时间。当然，每次保证最少的寻道时间并不能保证最终平均寻道时间最短。但是这种调度方法却能提供比 FIFO 更好的性能。

（4） 反复单向扫描（SCAN）

一般对于某个 I/O 设备而言，会不断有新的 I/O 申请加入其等待服务队列，在这种情

况下，上面几种磁盘调度算法都可能导致某个 I/O 申请始终被凉在一边不被搭理。只有当队列全空时，这样的申请才会被处理。

而 SCAN 调度算法能避免这一点。其基本算法流程是，磁头启动后一直按照某一个固定的方向（按照磁道号排序升或者降）移动，在申请等待队列中的一些访问目标位置正好位于这个方向上的申请被依次处理，直到磁头到达终点（磁道号最大或者最少）后，又开始反向移动，在移动过程中，按照正向移动同样的方法对申请等待队列中的 I/O 申请进行处理，到达原来起点之后再重复开始的过程，如此周而复始地反复顺序扫描磁盘表面，并对符合条件的 I/O 申请进行处理。

表 2.7. 磁盘调度算法比较

FIFO **（起始磁道 100）**		**SSTF** **（起始磁道 100）**		**SCAN** **（起始磁道 100）** **（磁道号升序）**		**C-SCAN** **（起始磁道号 100）** **（磁道号升序）**	
下一访问磁道	访问磁道数	下一访问磁道	访问磁道数	下一访问磁道	访问磁道数	下一访问磁道	访问磁道数
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
平均寻道数	55.3	平均寻道数	27.2	平均寻道数	27.8	平均寻道数	35.8

（5） 单向扫描原则（C-SCAN）

C-SCAN 调度算法只允许磁头单向扫描盘面，当到达终点后，快速返回起点，然后继续原来的扫描过程，在扫描过程中，同时检查 I/O 申请等待队列中的申请，只有发现其访问位置处于磁头位置的时候就加以处理。如此周而复始，完成磁盘调度任务。

（6） N 步单向扫描原则（N-step-SCAN）

SSTF、SCAN 和 C-SCAN 调度算法都可能出现一个很让人恼火的事情，即如果在某个磁道一时间有很多 I/O 申请的话，就会导致磁头移动到该磁道后长时间不能解脱的情况，从而导致其它申请需要作长时间等待。

N-Step-SCAN 调度算法的提出就是为了克服这一现象。这种算法的基本内核仍然是 SCAN 的，所不同的是，原来的 I/O 申请等待队列被分成长度为 N 的一个个子队列。SCAN 可供选择的 I/O 申请是这些子队列中的某一个，一旦选中，就会不断执行直到整个队列处理完，然后转向下一个子队列继续同样的操作。在处理过程中，难免会有新的 I/O 申请加入，这个时候，新的 I/O 申请不能加入正在处理的子队列中，即便它的长度可能已经小于 N 了，而是要么加入其它某个长度小于 N 的未处理子队列中，要么形成一个新的子队列。

（7） （FSCAN）

FSCAN 调度算法的基本思想和 N-Step-SCAN 算法类似，所不同的是在该算法中，只有两个 I/O 申请等待队列。当某一个队列开始处理时，另外一个队列为空。所有新加入的

I/O 申请都会进入这后一个空队列。而对前一个队列的处理直到所有申请都处理完毕之后，才转入第二个队列的处理，此后，所有新加入的申请将会被压入前一个队列，如此循环，周而复始。

表 2.7.给出了几种常用的磁盘调度算法比较。

2.2.5. 文件管理

在计算机应用中，信息的传输、处理和存储是三个最关键性的问题。而其中信息的存储方式是其它问题的讨论基础。一个应用难免要生成各种各样的数据信息，不论是暂时的、还是永久性的，都涉及到信息的存储问题。前面已经讲到过的内存管理所讨论的主要是信息的暂时存储和管理问题。而本节将着重讨论永久信息存储问题。

首先我们知道，永久信息的存储介质是外围存储设备，诸如，硬盘、软盘和光盘等。它们的存储方式所涉及到的一个关键性概念就是文件。鉴于此类存储设备在计算机系统中的重要地位，很多操作系统都会将对此类存储介质的存储管理纳入自己的设计范围内进行仔细考虑。于是，文件管理就产生了。

文件管理的核心部分是文件系统。它对操作系统的性能影响颇大。现在人们已经提出了很多不同类型的文件系统，形成了操作系统研究与实践的一道亮丽的风景线。

2.2.5.1. 文件与文件系统

- 文件

在讨论文件时我们通常还会遇到其它三个概念，即字段、记录和数据库。

字段是数据的基本单位。一个字段通常是一个有着特定长度和类型的值。比如说雇员姓名等。字段可分为两种：长度固定或者长度可变。对于后者，每个字段通常包括两到三个子域：字段的值、字段的名称、然后可选择的就是字段的长度。有的时候，字段可以靠一个特殊字符来表示它的结束，从而不需要有字段的长度，比如说在某些字符串定义以 NULL 作为结束标志。

纪录是若干个相关的字段所组成的集合，可以被某些应用程序作为一个整体来看待。比如说，一个雇员的纪录包括这样一些字段：姓名、年龄、工种、住址、联系方式等等。和字段一样，记录的长度也分固定和可变两种。

文件是一些相似纪录的集合。文件都有一个唯一标示的文件名，应用程序或用户可以通过它来对文件定位。文件可以被创建或者删除。通常一些访问权限的设定也都出现在文件这一层次。而在一些更高级的系统中，访问权限控制可以细分到纪录级、甚至是字段级。

数据库用来对一些相关数据进行管理，它通常是通过文件管理相关程序对一些可能属于不同类型的文件进行统一管理。一般而言，这些文件或者数据之间的关系是明确界定了的。

文件通常是用户或者程序管理信息和数据经常要用到的。针对文件有一些比较典型的操作：比如说，取出文件中的一个或若干个记录；向文件中插入一个或者若干个记录；从文件中删除一个或者若干个记录等等。

还有一个和文件联系比较紧密的概念就是文件目录，它为一组文件的统一管理提供了一个简单而有效的工具，从而成为所有文件管理系统中不可或缺的一部分。

- 文件系统

文件系统向用户或程序提供一个使用文件的统一界面，从而能够使得对文件的各类操

作能够在更加抽象更加简便的层次上进行。文件系统的引入通常有以下几种目的：

（1） 满足用户管理数据的需要，这其中包括数据存储和对数据的操作；

（2） 尽可能保证文件中数据的有效性；

（3） 性能优化，以提高系统的吞吐量和响应速度；

（4） 提供不同类型的存储设备的 I/O 支持；

（5） 消除或降低数据丢失或遭破坏的可能性；

（6） 提供一个标准的 I/O 界面；

（7） 在多用户系统中，向多个用户提供 I/O 支持等等。

2.2.5.2. 文件组织与访问

文件组织是指其中纪录的逻辑结构。它是按照这些记录访问方式来划分的。在外围存储设备中存在的文件的物理结构有赖于具体的存储设备的分块和分配的策略，这个在后面阐述，这一节主要讲述文件组织的类型。

选取一个文件组织类型需要从几个方面来考虑：信息访问的速度、信息更新的难易程度、存储的经济性以及可靠性等。

不同的应用对这几个方面的要求各有不同，需要在具体应用中进行权衡，毕竟有些方面相互之间会产生矛盾。比如说，如果想要存储经济，就不好强求访问速度高了。因为一般而言，访问速度提高意味着需要更多的冗余信息来提供文件或记录索引，这就意味着有效存储信息所占比例将会有所下降。

现在已经有很多文件组织类型。我们可以总结出五种基本的类型，其它的文件组织类型要么是其中一种，要么就是其中若干种的混合。这五种基本的文件组织类型就是堆文件、顺序文件、索引顺序文件、快速文件等五种组织类型（图 2.16）。其性能比较见表 2.8。

（1） 堆文件

文件组织形式中最简单的莫过于堆文件了。在堆文件组织形式中，数据依照它们被存储的顺序依次排放在存储设备中，每一个记录就代表每一次连续数据的存储。记录间有一个特定的分隔符作为界定。每一个记录都有若干个字段，其中最主要的就是记录名和记录的值。

由于在堆中，除了记录信息，没有其它的冗余信息，对信息的检索只能用完全搜索的办法，这就是说，如果我们需要寻找某个特定的记录，那我们就需要依次查找每个记录直到目标出现；而我们如果想要查找有着某个特定值的记录的时候，那就麻烦了，——我们需要搜遍所有记录以找出所有符合条件的。

（2） 顺序文件

最常见的文件结构是顺序文件结构。在这种文件类型中，所有的记录都有着相同的格式：等长、字段数量相同、字段顺序也一致。因此，各个记录只需要保有各自的值即可，而记录名和记录的长度等信息可以提取出来作为文件自身的属性统一存储，这样可以提高文件存储的经济性。

在每一个记录中有一个特殊的字段，就是关键字。关键字唯一标识记录，通常，记录是按照关键字顺序来存储的。

顺序文件组织在应付批处理应用中通常是高效的。而在类似交互式应用中，涉及到大量频繁的纪录查询或更新的时候，顺序文件结构的表现就很差。

通常，顺序文件的物理存储采用链表结构：若干个记录共用一个物理块，而每一个物理块都会有一个指针指向下一个文件所用到的物理块。

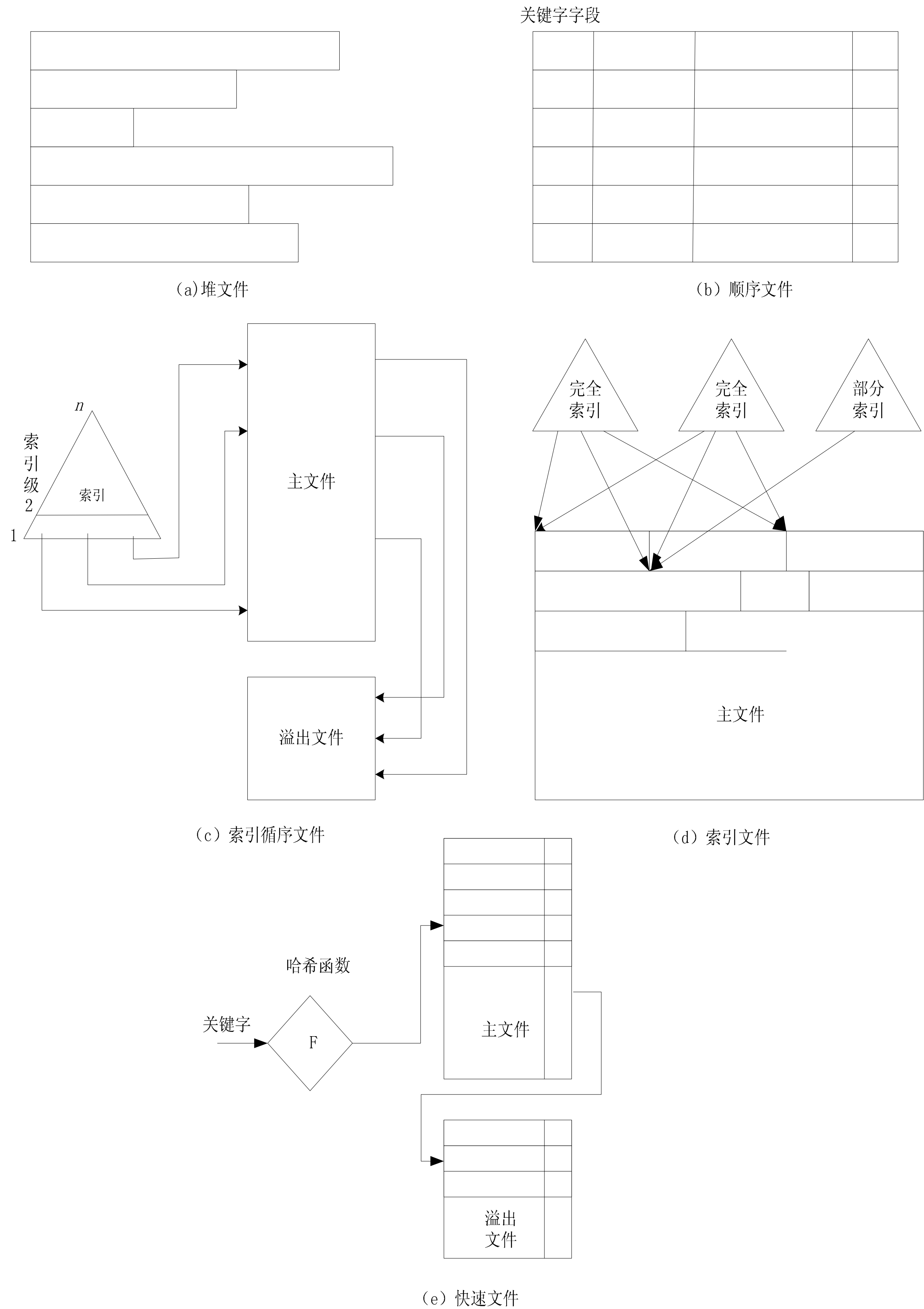


图 2.16. 文件组织结构

（3）索引顺序文件

为了克服顺序文件在随机访问时的缺点，人们提出了索引顺序文件类型。

索引顺序文件与顺序文件相比，所不同的是加入了两个新的特征：一个可以支持随机访问的索引和一个溢出文件。索引能够使得用户或程序很快找到所需要的纪录；而溢出文件和日志文件比较类似，这样溢出文件中的记录可以通过它们前继记录所给出的指针来实现快速访问。

在简单的索引顺序文件中，索引只有一级，这时的索引就是一个简单的顺序文件，在这个文件中的记录包含两个字段，一个是该纪录的关键字，另外一个则是一个指向主文件某个纪录的指针。当需要查找某个字段时，首先在索引文件中查找与该字段对应的关键字距离最近的一个索引，获得该索引之后，就可以根据该索引给出的指针转到主文件中的对应位置继续搜索直到找到所需要的字段为止。

如果需要插入一个新的纪录。它会被插入溢出文件中。而与其关键字距离最近的前继纪录的指针将会被刷新并指向该记录。

另外为了获得更高的访问速度，可以引入多级索引结构。

（4）索引文件

索引顺序文件有一个局限性就是它只对纪录的某一个特定字段建立索引。因此，如果我们按照一个并非建立了索引的字段值来搜索的话，索引顺序文件和顺序文件一样都会表现很次。但是，很多时候我们需要按照不同的字段来搜索纪录。我们需要这种灵活性。

索引文件的引入就是为了迎合这种需求。索引文件实际上是一个多索引结构，它针对记录中可能会成为搜索对象的字段建立多个索引文件。在这种索文件组织结构中，记录的访问可以完全通过索引直接完成，并且记录的长度是可以变化的。所有的索引可以分为两种类型，一种是完全索引，它对所有记录都建立索引；另一种是部分索引，它只对某些感兴趣的记录建立索引。由于记录允许变长，所以某些记录所拥有的字段别的可能就没有。当有新的记录加入的时候，所有的索引文件都需要更新。

（5）快速文件

快速文件组织形式利用哈希表技术，给定一个关键词，可以很快地找到对应的纪录所在的物理块的地址。

表 2.8. 文件组织结构性能比较

	存储空间经济性		**数据更新速度**		**数据取出速度**		
文件组织形式	纪录变长	记录等长	纪录大小相等	纪录大小更大	取出单个纪录	取出一组纪录	取出全部记录
堆文件	A	B	A	E	E	D	B
顺序文件	F	A	D	F	F	D	A
索引顺序文件	F	B	B	D	B	D	B
索引文件	B	C	C	C	A	B	D
快速文件	F	B	B	F	B	F	E

其中，A 表示极好，B 表示好，C 表示充分好，D 表示需要一些额外的工作，E 表示可能

需要很多工作，F 表示无法用于该目的。

2.2.5.3. 文件共享

在一个多用户系统中，一个文件通常需要在多个用户中共享，这个时候就有两个问题需要考虑：一个是访问权限，另外一个就是并发访问管理。

- 访问权限

为了实现多用户系统的文件访问权限控制，不同的用户或用户组的权限被分别规定。我们可以通过限定可访问文件的用户或用户组来实现对文件访问权限的控制。

通常针对一个文件有这么一些访问权限的规定，比如，执行、读、写、删除等等，并且各个基本权限可以自由组合形成更为精确的规定性。

通常每一个文件都有一个所有者。一般而言，这个文件的所有者也是创建该文件的用户，他对该文件具有完全的访问权限。另外，每一个多用户系统还有一个拥有至高权限的用户，也称为管理员，他对所有文件都拥有完全的访问权限，从而可以方便地对系统进行有效管理。

- 并发访问管理

当多个用户拥有对某个文件的写权限的时候，我们就需要引入相应的互斥控制机制来维持文件本身逻辑的完整性。一个比较简单的做法就是当某个用户需要写这个文件时，拒绝其它用户访问，直到该用户执行完操作为止。而更为精细的做法则是在记录级上实现互斥，这样就可以使得一个用户在对文件某个记录进行操作的同时，另外一个用户则可以对文件中另外一个记录进行操作，从而提高系统效率。

2.2.5.4. 记录分块

上面所讲述提到的文件组织结构是计算机中记录的逻辑结构，它们最终是要在存储在外围存储设备中的，这就涉及到记录的物理结构问题。在物理结构中，记录是以物理块的形式组织的。

现在就有一个问题提出了，物理块究竟应该是等长的还是变长的。毫无疑问，变长可以带来灵活性，我们可以按照记录逻辑结构的特点随意定义块的长度，但是这样却会在 I/O 缓冲、内存缓冲的分配和管理上带来麻烦。事实上，在大多数系统中，物理块长度一定。这样就又有一个问题出来了，物理块的长度如何界定呢？物理块如果越长，而且文件的记录按顺序排放在物理块上的话，每次 I/O 访问就可以往内存读入更多的记录，从而能够提高 I/O 设备的吞吐量。但是如果文件的纪录是随机存储的话，也就说随意散落在各个物理块中，那样的话，物理块越长，每次读入内存的记录中无用的就越多。事实上，对于这种情况，我们可以通过碎片整理程序之类的工具实现文件记录的集中存储，从而提高每次读入纪录的有效部分所占比率。

给定一个记录的长度，我们就涉及到如何实现分块的问题，主要有三种分法：

（1） **固定分块法**：记录定长。每个物理块都存放整数个纪录，而这个数目都是一样的。这种方法会导致每个物理块最后可能会有固定大小的存储空间被浪费。

（2） **变长可跨越分块法**：使用变长纪录，打包进块，没有浪费存储空间，而某些记录可能被分开存储在两个块中，由后续块的指针来标明连续性。

（3） **变长不可跨越分块法**：使用变长纪录，但不允许一个记录分开在两个块中存储。因此，如果下一个记录比块的剩余空间大的话，该剩余空间就只能被浪费掉了。

固定分块法适合记录定长的顺序文件存储。而变长可跨越分块法能够有效利用存储空间，并且不限制纪录的大小，灵活性很好，但是却很难实现。变长不可跨越分块法会导致存储空间浪费，并且纪录的长度受限于物理块的长度。

在现代计算机中，虚拟内存技术得到广泛应用，这个时候我们在考虑记录分块的时候就需要注意到一个现实，这就是记录分块技术常常要和虚拟内存技术相关的硬件打交道的。由于虚拟内存的最小管理单位是分页，因此一个比较理想的 I/O 传输的基本单位就是分页。但是通常情况下，分页往往是比较小的，因此在某些计算机系统中就采取将多个分页的长度和作为一个物理块的大小，从而能够实现一次 I/O 访问能够读取整数个分页的目的。

2.2.5.5. 外围存储设备管理

在文件系统管理中，对文件的物理存储管理也是一个很重要的内容。这里文件的物理存储管理主要就是指外围存储设备管理。

外围存储设备管理包括两个方面，一个是文件分配，另外一个就是空闲存储空间管理。

- **文件的存储空间分配**

文件的存储空间分配涉及到三个方面的内容：一是文件创建之初，分配给它的存储空间大小如何确定的问题；二是文件存储空间总是分成若干个内部连续的存储划分，这样一些划分的大小如何确定；三是采用什么样的数据结构来对这些分区进行统一管理，也就是涉及到一个文件分配表的选择问题。

对于第一个问题的回答有两个，一个是预分配法；一个是动态分配法。所谓预分配法就是在文件创建之初按其长度的最大值一次性分配给相应大小的存储空间。这种分配方法在某些可以预知文件最大长度的情况下是很奏效的，但是对于一些难于预料文件最大程度的情况，为了避免最后所分配区域不够用的情况，预分配的存储空间总是比实际所用到的要大，从而造成存储空间的浪费。而动态分配法与预分配法相比就具有更大的灵活性，它只有当预先分配的存储空间不够用时再动态分配一定数量的存储空间。

存储划分的大小的选取需要从多个角度多加权衡，才能得到比较满意的答案。如果划分的连续存储区域比较大的话，不仅可以提高某些数据访问的速度，同时还可以减小文件分配表的大小。综合起来，权衡利弊，我们可以有以下两种选择：

(1) 变长、较长的连续区域划分：这种方式可以提高系统性能，避免存储空间的浪费，而文件分配表也会比较小。问题是，存储空间不好使用，这个道理和前面讲过的物理块变长的情况比较类似。

(2) 物理块：物理块长度较小，定长，容易分配，没有考虑连续性，分配起来比较容易，有一定的灵活性。至于说连续性方面的缺陷，如果采用某些类似文件碎片整理程序的工具对存储空间进行整理的话，则可以有些补偿，从而能够在某种程度上提高系统的性能。事实上，物理块是在外围存储空间设备管理中广泛使用的。

至于说采取什么样的数据结构来对这些存储区域的连续划分进行管理，我们有三种可以选择的方案，即连续分配法、链状分配法和索引分配法。

所谓连续分配法就是在文件创建之初时将一片连续的物理块分配给文件。这是一种预分配法，采用变长存储划分。

和连续分配法相对的是链状分配法，它不要求各物理块是连续摆放的，事实上可以是在存储空间中随机摆放的，所有物理块按逻辑顺序通过指针穿成一个单向链表。

索引分配法则在主文件之外对它维护一个索引文件，用于索引主文件中各个物理块，从而实现对各物理块的快速访问。

● 空闲存储空间管理

就像已分配的存储空间需要管理一样，未分配的存储空间也需要进行合理的管理。常用的方法主要有三种，即位示图、空闲块链和索引。

位示图维护一个向量，其每一位与存储空间中的一个物理块一一对应。位值为 0 时表示与该位对应的物理块处于空闲状态。如果是 1 的话，则说明物理块已经被征用了。

空闲块链则是使用链表技术将所有空闲存储区域划分按照一定的逻辑顺序连接起来。类似的，索引方法也就是利用索引技术将所有空闲存储区域划分进行统一管理。

2.2.5.6. Linux 的文件系统管理

Linux 的一大特色就是能够支持很多种文件系统，这其中比较常用的包括 EXT2、EXT3、VFAT、NTFS、ISO9660、JFFS、ROMFS、NFS 等，并且继续还有更新的文件系统得到支持。每一种文件系统都有着自己独特的组织结构和操作，给统一管理带来一定的麻烦。而 Linux 引入了虚拟文件系统（Virtual File System）用于对这些文件系统的统一管理，事实证明，这种方法是很奏效的。

VFS 只存在于内存中， 它在系统启动时创建，在系统关闭时注销。VFS 的作用就是对各类文件系统作进一步抽象，最终实现各类文件系统展现在用户面前的是一个统一的操作界面，并且能够提供一个统一的应用编程接口。

在所有可用的文件系统中，最重要的一种就是 EXT2，它是 Linux 自行设计并且具有较高效率的一种文件系统，并作为 Linux 可执行文件的标准文件组织形式。

文件操作难免数据在内存和外围存储设备之间的大量传输。如何解决内存和外围存储设备 I/O 之间在数据传输速度方面的差异，也是 Linux 文件管理系统需要面对的问题。Linux 的办法是采用缓冲技术和哈希表技术。

由于 Linux 天生就是一个网络型操作系统，多用户多道程序设计，Linux 在文件访问控制方面作了周详的考虑，引入了访问权限按用户组进行控制的方法。而在并发访问控制方面，引入了文件锁方法。

● 文件系统管理

在 Linux 中，普通文件和目录文件保存在称为块物理设备的磁盘或者磁带上。一个 Linux 系统支持若干个块物理设备，每个设备可以定义一个或者多个文件系统。

每个文件系统是由逻辑块的序列组成的，一个文件系统所在的存储空间一般被划分为几个用途不相同的部分，即引导块、超级块、索引节点（inode）区以及数据区等。

(1) 引导块：处于文件系统的开头，通常为一个扇区，其中存放引导程序，用于读入并启动操作系统。
(2) 超级块：用于记录文件系统的管理信息。特定的文件系统定义了特定的超级块。
(3) 索引节点区：一个文件（或目录）占用一个索引节点。第一个索引节点是该文件系统的根节点。利用根节点，可以把一个文件系统挂在另外一个文件系统的非叶子节点上。
(4) 数据区：用于存放文件数据或者管理数据。

在 Linux 实时运行时所支持的文件系统都维护在一个文件系统注册链表中。该链表的每一个节点对应一个文件系统，其中包含了文件系统类型等信息，并且维护了一个指针用于指向下一个文件系统节点。文件系统类型的注册和注销可以通过两种途径来完成，一种是在编译 Linux 内核时确定，并在系统初始化时通过内嵌的函数调用向注册链表登记；另外一种就是利用 Linux 的模块加载/卸载机制。

与任何一种 Unix 操作系统一样，Linux 并不是通过设备标识来访问某个文件系统，而是将它们通过文件系统加载（mount）机制形成一个统一的树形结构以便访问。需要注意的是，在文件系统加载之前，必须要确定该文件系统类型是被实时运行中的 Linux 系统支持的，换句话说，该文件系统类型应该能在 Linux 实时运行时的文件系统注册链表中找到。同样，也可以通过对应的文件系统卸载（unmout）机制将某个业已加载上去的文件系统从所形成的树形结构中剔除。

- **虚拟文件系统**

虚拟文件系统是物理文件系统与服务之间的一个接口。它对 Linux 实时运行时所支持的每一个物理的文件系统进行抽象，使得不同的文件系统在 Linux 内核以及系统中运行的其它进程看来都是相同的。

虚拟文件系统的功能包括：

(1) 记录可用的文件系统类型；

(2) 将设备同对应的文件系统联系起来；

(3) 处理一些面向文件的通用操作；

(4) 涉及到针对文件系统的操作时，虚拟文件系统把它们映射到与控制文件、目录以及 inode 相关的物理文件系统。

当某个进程发布了一个面向文件的系统调用时，Linux 内核将调用虚拟文件系统中相应的函数，这个函数处理一些与物理结构无关的操作，并且把它重定向为真实文件系统中相应的函数调用，后者则用来处理那些与物理结构相关的操作。

2.3. 用户界面

从用户角度来看，操作系统更重要的是用户界面。随着计算机硬件技术的飞速发展，计算机的计算能力在应付普通应用方面显得绰绰有余，这个时候，人们更关心的是人机交互的便易性、友好性等方面，并且使得用户界面越来越成为任何一种操作系统不可或缺的一部分。正因为如此，微软的 Windows 操作系统尽管在性能方面无法和 Linux 相比，但是因为其用户界面的易用性而继续其对市场的垄断。当然，这其中也有上层应用软件的原因，不过，对一般性应用而言，Linux 的上层应用已经足够丰富，并且有快速增加的势头。

最开始的计算机用户界面很容易让人们联想起控制板，上面有一些指示灯、还有一些按钮，甚至于会只有跳线。这样的用户界面只有专业人士才能应用自如。

随着计算机技术的发展，计算机的应用开始面向普通大众。字符型用户界面产生了。Linux 的控制台、DOS 的命令提示符状态等等都是字符型用户界面的典型代表。在字符型用户界面，界面的基本显示单元是 ASCII 编码的字符。当然，通过精细编程，字符型用户界面也可以达到很好的交互。但是，由于基本显示单元是字符，并不能得到灵活自然的效果。

图形用户界面的引入，使得计算机的交互能力得到显著增强。在苹果公司第一次引入图形用户界面的时候，人们因为它会消耗更多的计算机资源而为它的前途担忧。但是在微软公司的极力倡导下，图形用户界面技术得到了广泛的应用和完善，各类图形用户界面层出不穷。而微软公司也因为其采用了图形用户界面的 Windows95 系统而开始登上操作系统领域的霸主地位。

人工智能技术的发展为计算机技术的进步不断注入新的活力，这也表现在对计算机用户界面上。语音识别、手写输入、Agent 技术等等为计算机用户界面友好性易用性的增强开辟了新的途径。可以预见，未来计算机用户界面的进步将主要集中在智能化方面。

由于字符型用户界面现在应用越来越少，而且比较简单，我们将集中讲解图形用户界

面和智能化用户界面。

2.3.1. 图形用户界面

2.3.1.1. 基本知识

图形用户界面（GUI）是迄今为止计算机系统中最为成熟的人机交互技术。一个好的图形用户界面的设计不仅要考虑到具体硬件环境的限制，而且还要考虑到用户的喜好等等。图形用户界面的引入主要是从用户角度出发的，因此用户自身的主观感受对图形用户界面的评价占了很大比重，比如说，易用性、直观性、友好性等等。另外从纯技术的角度看，仍然也会有一些标准需要考虑，比如说，跨平台性、对硬件的要求等等。在嵌入式系统开发和应用中，我们所考虑的问题主要还是集中在图形用户界面对硬件的要求以及对硬件类型的敏感性方面，在提供给用户的最终界面方面还只是要求简单实用就够了。

虽然不同的 GUI 系统因为其使用场合或服务目的的不同，具体实现互有差异，但是总结起来，一般在逻辑上可以分为以下几个模块（图 2.17）：底层 I/O 设备驱动（显示设备驱动、鼠标驱动、键盘驱动等）、基本图形引擎（画点、画线、区域填充）、消息驱动机制、高层图形引擎（画窗口、画按钮）以及 GUI 应用程序接口（API）。另外，为了实现 GUI 系统，一般需要用到操作系统内核提供的功能，如线程机制、进程管理。当然，不可避免的需要用到内存管理、I/O 管理。甚至还可能有文件管理。

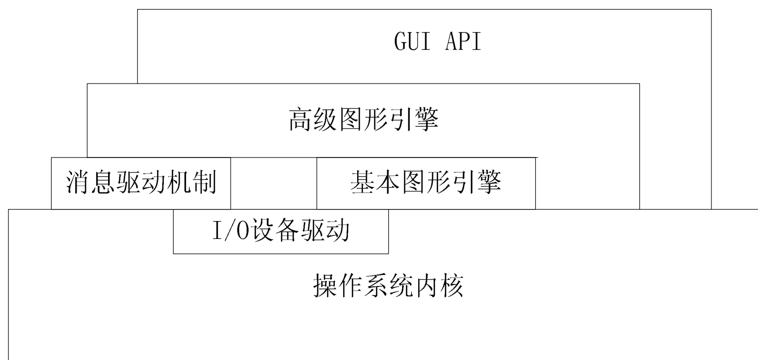


图 2.14. GUI 的一般架构

2.3.1.2. 关键技术

- **模块化**

计算机系统设计的一个大趋势，就是系统设计的模块化。模块化的初衷就是实现外部调用与内部实现的分离，从而在带来系统架构清晰化的同时，为系统的维护和改进也打下基础。

上面在提到 GUI 的一般架构的时候事实上就是从模块化设计的思想出发对 GUI 体系结构进行逻辑上的划分（图 2.17）。

诸如显示驱动、鼠标驱动、键盘驱动等构成了 GUI 的硬件基础。而由于诸如此类设备的多样性，需要对其进行抽象，并提供给上层一个统一的调用接口，而各类设备驱动则自成一体，形成一个 GUI 设备管理模块。当然，从操作系统内核的角度看，GUI 设备管理模

块则是操作系统内核的 I/O 设备管理的一部分。

在 GUI 体系结构中，消息是一个非常重要的概念，它不仅是底层 I/O 硬件和 GUI 上层进行交互的基础，同时也是各类 GUI 组件如窗口、按钮等相互作用的重要媒介。一个 GUI 系统的消息驱动机制的效率对该系统的性能，尤其是对响应速度等性能的影响很大。

基本图形引擎模块完成一些基本的图形操作，诸如，画点、画线、区域填充等。它直接和底层 I/O 设备打交道，同时，多线程或者多进程机制的引入也为基本图形模块的实现提供了很大的灵活性。

高级图形引擎模块则在消息传递机制和基本图形引擎的基础上完成对诸如窗口、按钮等的管理。

GUI API 则是提供给最终程序员的编程接口，使得他们能够利用 GUI 体系所提供的 GUI 高级功能快速开发 GUI 应用程序。

- 消息驱动机制

任何一个 GUI 系统都会有相应的消息驱动机制，在某些 GUI 系统中，消息驱动也被称为事件驱动。

在消息驱动的应用程序中，计算机外设发生的事件，如键盘击键、鼠标摁键等等都由系统收集，将其以事先约定的格式翻译成特定的消息。应用程序一般都包含有自己的消息队列，系统将消息发送到应用程序的消息队列中，应用程序可以建立一个消息处理循环，在这个循环中不断地读取消息并处理消息，直到有特定的消息传来为止。图 2.18 给出了消息驱动机制的一般流程。

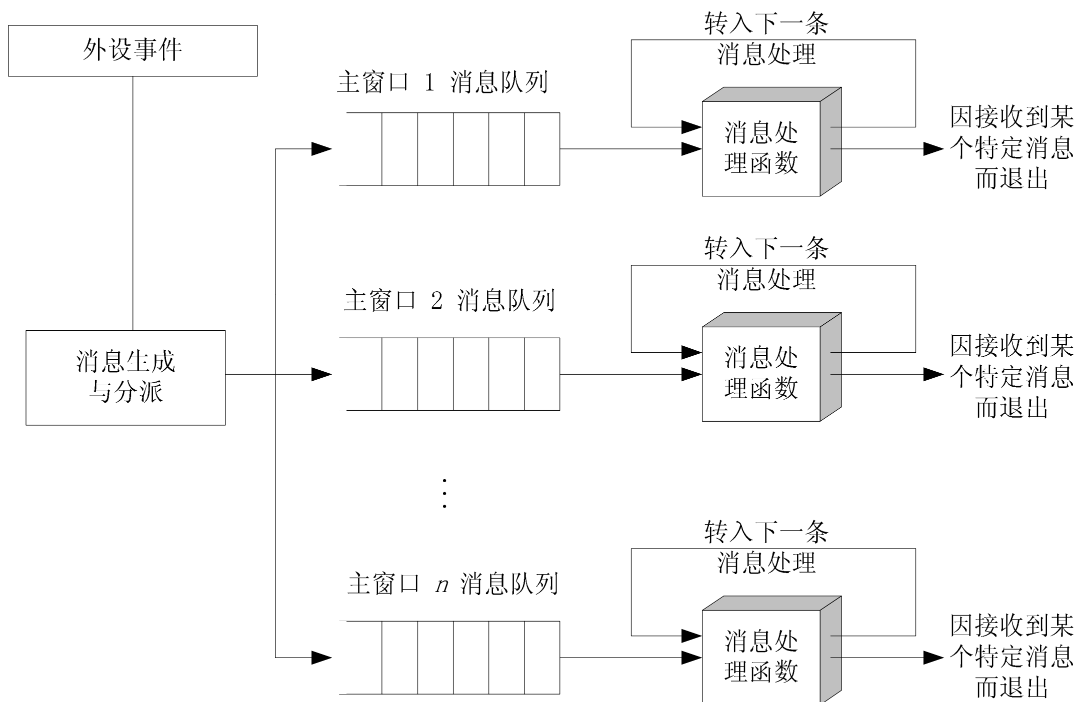


图 2.18. 消息驱动机制的一般流程

一般地，消息由一个整型的标识和一些附加参数组成。

应用程序一般提供一个处理消息的标准函数，在消息循环中，系统可以调用此函数，应用程序再此汉书中处理相应的消息。

- **GUI 系统移植**

随着计算机硬件的发展，各类 GUI 设备日益多样化，GUI 系统的跨平台移植问题显得日益突出。比如说，单单显示技术这一项，就有帧缓冲、VGA、SVGA 等等多种。而输入技术也有鼠标、键盘、触摸笔等等。

为了尽可能减少平台移植的代价，模块化技术得到广泛应用。通过模块化技术可以将与硬件平台相关的部分在 GUI 系统中单列成一个模块，或者干脆就从 GUI 系统中独立出来，成为操作系统核心的一部分。而 GUI 系统的上层结构则是建立在一个抽象的与平台无关的结构之上的，这样的话，就可以使得 GUI 系统移植仅仅局限在与硬件平台相关的部分上，这不仅能够减少移植的代价，而且也能够保护应用程序开发人员的利益，因为这样他们利用上层 GUI API 开发的应用程序可以不用作修改就可以运行在不同硬件平台上。

- **GUI 系统的微型化**

GUI 系统的微型化是在嵌入式系统开发中常常要遇到的。现在一般的 GUI 系统为了各方面的因素考虑，比如说为了程序员开发的便利性，常常会有很多冗余设计，这里所谓的冗余设计是指某些较高层次的 API 并非必须，而是可以由较低层的 API 组合而成的。GUI 系统的微型化首先是拿这样的 API 开刀把它们给裁减掉。另外我们需要注意的是，在某些具体应用中，可能使用到的 API 函数很少，可能只是一个完整 GUI API 系统的一个很小的子集，在这种情况下，虽然说为了满足一般应用需要的 API 仍然很多，但是我们却可以根据某些特定应用的需要只取一般 API 集合的一个小的子集，从而实现 GUI 系统的微型化。

2.3.2. 智能化用户界面

智能化用户界面是将人工智能技术融入到用户界面中去的产物。现在比较成熟的智能化用户界面技术有语音输入、手写识别。而正在蓬勃兴起的是利用 Agent 技术对用户界面的改造和提升上。

而人工智能的最新发展动向是分布式 Agent，并开始出现利用 Agent 理论对传统人工智能技术进行统一和改造，比如说，利用 Agent 概念将人工智能的主要分支如问题求解、推理、学习等等的描述统一起来，从而改变了传统人工智能技术各分支老死不相往来的局面。因此，本章只讲述 Agent 技术以及它与用户界面的结合上，看看它是怎么使得用户界面更智能化、人性化，从而为用户使用计算机带来极大的便利。

2.3.2.1. Agent 技术

Agent 的概念首次出现在 Minsky1986 年出版的“思维的社会（The Society of Mind）”，Minsky 认为社会中的某些个体经过协商可求得问题的解，这些个体就是 Agent，根据一定的规则组成一个多 Agent 系统（MAS）。还认为 Agent 是具有技能的个体（1994），Agent 应当具有社会交互性和智能性。

此后，人们对 Agent 的理解真是见仁见智，比如，Hewitt 认为定义 Agent 与定义智能一样困难。Wooldridge 和 Jennings 认为 Agent 应具有自主性、社会交互性、反应能力。

从 Agent 模型角度来看，有思考型 Agent（如 BDI 表示和推理）、反应型 Agent（不会推理，感知-动作）和两者混合型。

- **Agent 的研究方法**

Agent 理论的研究主要有逻辑方法和经济学方法，此外还有混沌学方法。

自 Hintikka 的关于 knowledge 和 belief 做出先驱性工作以来，在描述 Agent 推理方面

已经有许多工作。在刻画组成 Agent 思维状态的各元素间的关系方面的代表是 Cohen 和 Levesque 的 Intention 理论，尽管在这方面取得了很大进展，但仍有一些相当基本的问题存在。

首先，与可能世界语义有关的问题还不能认为已经得到了解决，如逻辑全知问题。可能世界语义被许多研究者采用，但它一般来说不能表示现实世界有限资源的 Agent 模型。一个解决方法是 Rosenschein 和 Kaelbling 的情景自动机方法，然而，还不知道怎样用这种方法描述 Desire 和 Intention，尽管作过一些努力。

在那些描述了不同意识属性的逻辑中，也许最重要的问题是与 Intention 有关，Intention 和 Action 之间的关系没有很满意的解释。关于用那些意识属性来刻画 Agent 也是有争论的，目前一种流行的方法是使用 Beliefs，Desires，和 Intentions 地组合（即 BDI 结构）。

一般来说，Agent 理论中使用的逻辑包含了多个相互间有复杂关系的模态词，难于在这样的逻辑上展开 Agent 理论的研究。

关于 Agent 理论的地位还有一些混淆，一些研究者认为 Agent 理论是用来描述 Agent 的 Specification；有的认为 Agent 理论是用于知识表示方面；还有的认为应是对认知科学和哲学中的一些概念形式化。

对于 Agent 理性的研究有基于逻辑和基于对策论的两种基本方法。在哲学上，认为合乎逻辑的是理性的，为此提出了各种逻辑体系，定义了公理系统和推理规则，来证明一些特定的命题是否成立，认为一个合理的行为可从当前的信念合乎逻辑地推导出来，这就是逻辑理性。对于思维状态模型的研究大都属于这一流派。另一种方法是采用对策理论和决策理论，其信念模型是描述如果采用一个行动将会发生什么，为每个后果都赋予概率。愿望模型是用实数表示那些可能状态的效用，一个合理的行动是使得期望效用最优化的行动，这需要依据信念和愿望通过概率计算得到，这就是效用理性。从概念角度来看，逻辑方法实现了理性的推理，决策理论方法通过最优化主观效用而实现了理性的决策。从技术角度看，使用符号推理的逻辑理性无法使效用最优化，而使用数值分析的决策论理性也忽略了推理环节。对于一个处于动态环境中资源有限的 Agent 来说，既需要对世界进行推理也需要做出获得最大收益的合理决策。

此外，随着复杂系统理论的研究和发展，人们开始逐渐考虑借助复杂系统理论来研究如何实现 Agent 的智能。这其中比较典型的就是 Agent 的混沌学研究方法。从本质上讲，Agent 应该是非线性的，因为从复杂系统论的观点看，只有非线性才能表现出相当的智能行为。Agent 的混沌学研究方法通过研究 Agent 的行为轨迹来研究其智能本质。这种方法可以比较容易的对 Agent 本质达到全局性的把握，并且可能成为 Agent 理论模型向实际应用转化的一个中介。

就 Agent 理论整体来说，需要融合这三个流派的研究成果。

- **Agent 的逻辑学方法**

Agent 的逻辑学方法的首要任务是要确定 Agent 的思维状态模型，也就是说，Agent 的逻辑学方法首先认定 Agent 是一个思维机器。

现在在 Agent 的逻辑学方法研究中，Agent 思维的 BDI 模型影响比较大。所谓 BDI 模型是指 Agent 的思维模型包括三个基本要素，即信念（Belief）、愿望（Desire）和意图（Intention）。并以这个为基础，展开了对这三个要素的语义和相互关系的研究。

虽然针对BDI思维模型的研究已经初见成效，但是仍然存在理论与实践相脱节的问题，主要表现在：

(1) 在所使用的逻辑描述和实际系统结构之间缺乏清晰的关系，特别是，可能世界模型对于实现系统过于抽象；

(2) 这些逻辑描述对 Agent 的推理能力都做了不现实的设定。
所以，BDI 思维模型还很难对技术实践起真正的指导作用，仍然还是人工智能专家实验室里的玩具。

- **Agent 的经济学方法**

八十年代中期开始出现的Agent经济学研究方法以Agent的自利性和效用理性为前提，采用对策论等方法对 Agent 展开了广泛而深入的研究。

经济学研究的核心对象之一是市场，市场的基本组成包括物品、交易者、消费者、生产者和市场规则等因素。市场的最优状态是竞争平衡，是所有 Agent 都满意的状态。这些对阐述多 Agent 系统（MAS）的宏观理论提供了简洁的方式，也颇有说服力。但经济学成果的许多理论前提过于理想化，比如经济学的许多研究往往对个体内部结构的阐述较为简单化，追求设计的机制具有“激励相容性”，即个体不必经过深思熟虑只要坦诚相待，那么个体和群体的性能就都会达到最优化。信息完备的理论前提意味着每个个体都有“完美理性”，而在实际中是不可能达到的。MAS 研究的重要前提之一就是 Agent 的“有限理性”前提，Agent 的计算能力、掌握的信息等都会各不相同，这是符合现实的，也体现出 MAS 宏观理论与经济学/社会学的深刻差别。但经济学的许多成果仍值得 MAS 研究者借鉴并有必要在不同的理论前提下进行重新考察。

- **Agent 的混沌学方法**

随着 Agent 理论的发展，人们逐渐认识到 Agent 本身的非线性本质，结合非线性理论的研究和发展，开始出现 Agent 理论的混沌学研究方法。

表 2.9. Agent 思维状态的动力学解释

Agent 相关概念	动力学解释
Agent	进程（Process）
Agent 所拥有的知识	某个进程状态所包含的信息
Agent 所采取的举动	进程状态空间的一个状态迁移
Agent 的喜好	进程状态空间中的全局吸引子或排斥子
Agent 的目标	进程状态空间中某个局部吸引子
Agent 的意愿	收敛于某个局部吸引子的一段进程轨迹
Agent 的情绪（喜悦或痛苦）	进程运行状态与吸引子或排斥子距离的远近
Agent 的自学习	进程相空间的改变

Agent 理论的一个关键之处就是试图把握一个目的性，即 Agent 所作的一切都是为了实现某个或某些既定目标。再对这个理论进行细化，就会发现其中所隐含的优化问题。理性 Agent 总是被描述为采取各种行动来最大限度的实现其既定目标。实现这个目的性和最优化问题，就需要赋予 Agent 某种适宜的结构，包括思维和意图等方面。在这方面的一个典型的例子就是将 Agent 看成意愿（Wishing）、信念（Knowing）、喜好（Liking）、和意图（Intention）的统一体。

一般来讲，信念刻画的是 Agent 对世界的认识；喜好则是表现 Agent 自身希望世界是什么样子的；意愿描述的是 Agent 对实现某个目标的承诺；而意图所表述的是 Agent 对某个行为或动作的承诺。从混沌动力学观点来看，Agent 的思维状态有了新的解释（表 2.9）。Agent 的动力学研究方法的一大优点就是能够很容易实现 Agent 理论和具体实现之间的平滑过渡。唯一不足的地方就是复杂系统动力学理论仍然还处于起步阶段，需要有充分成熟的成果之后才能真正对 Agent 技术实现起指导作用。

2.3.2.2. Agent 技术与用户界面的结合

尽管说 Agent 理论的研究仍然处于众说纷纭，莫衷一是的局面，IT 业界在 Agent 技术的实践方面已经走了很长时间，充分体现了计算机行业的一个显著特征，就是“干了再说”。Agent 技术与软件技术的结合成就了软件 Agent 的出现。软件 Agent 作为 Agent 技术的软件实现，它是能够按照某个用户或者其它软件模块的意愿实现一定的功能。软件 Agent 技术逐渐渗透到计算机日常使用的方方面面，我们可以对软件 Agent 从功用角度分类，可以分为桌面 Agent（Desktop Agent）、互联网 Agent（Internet Agent）、企业网 Agent（Intranet Agent）等等。每一种 Agent 都不可避免地会提高用户界面的智能化程度。而我们在这里主要介绍 Agent 技术与用户界面结合的典型范例，即桌面 Agent。

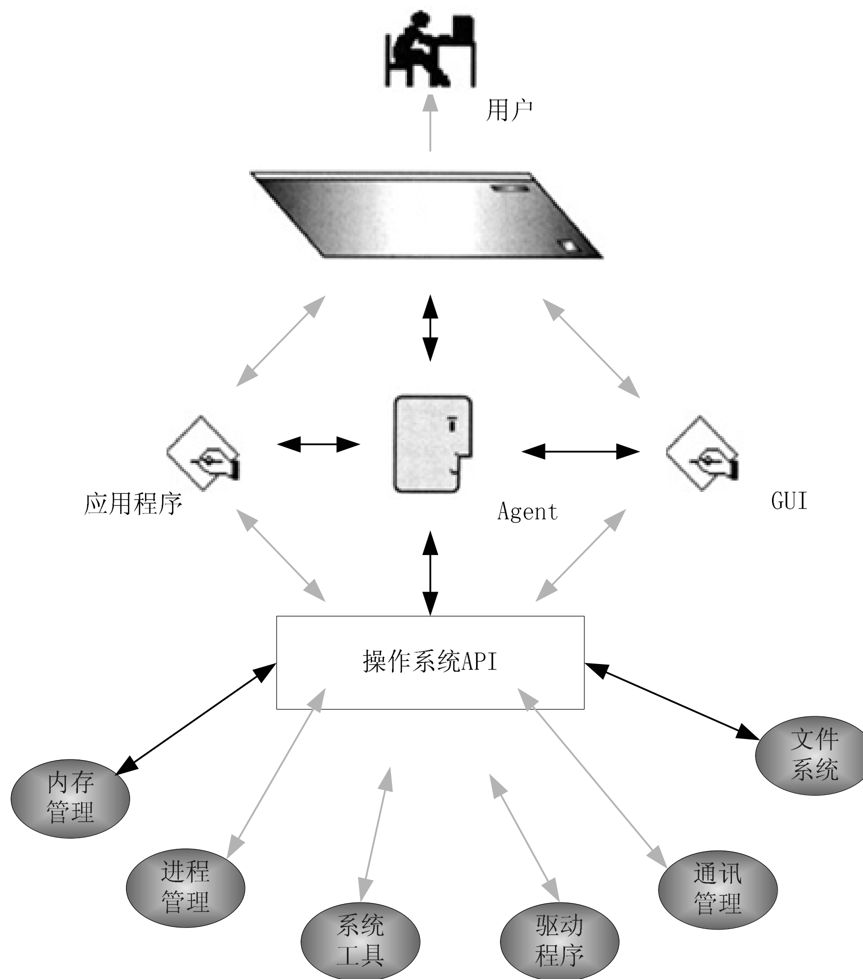


图 2.19. 操作系统 Agent

桌面 Agent 主要是用户界面 Agent，可分为三类，一类是操作系统 Agent；第二类是应用程序 Agent；第三类是应用程序组 Agent。这三类 Agent 分别在用户使用操作系统、某个特定的应用程序或某组应用程序时起到一些辅助性作用，从而提高用户界面的智能化和人性化，提高用户的工作效率。

- **操作系统 Agent**

操作系统 Agent（图 2.19）有两种，一种是操作系统智能化工具，它用于监视操作系

统级事件的发生并按照用户的调度安排执行一系列涉及到系统服务的任务；另一种是操作系统用户界面 Agent，它将 Agent 技术引入到对操作系统用户界面主要是图形用户界面中并与之相结合，为用户开始和完成某个任务操作提供友好的用户界面。表 2.10.给出操作系统 Agent 属性模型。

表 2.10. 操作系统 Agent 属性模型

属 性	描 述
环境	操作系统
技能	安装、定制、维护、自动化、文件管理、协助
知识	操作系统、网络、GUI、用户
通讯手段	GUI API、操作系统 API、用户界面

- **应用程序 Agent**

应用程序 Agent 可以作为某个特定应用程序的一部分，也可以独立于该应用程序。它主要执行一些需要用户与应用程序交互的任务，推动任务进程的自动化。表 2.11.给出了应用程序 Agent 的属性模型。

表 2.11. 应用程序 Agent 的属性模型

属 性	描 述
环境	应用程序
技能	定制、自动化、协助
知识	应用程序、用户、操作系统
通讯手段	应用程序 API、操作系统 API

- **应用程序组 Agent**

应用程序组 Agent 和应用程序 Agent 比较类似，只是这里与 Agent 发生关系的对象不再是应用程序而是一组应用程序。

2.3.3. Linux 下的用户界面

现在 Linux 下的用户界面主要有两种，一种字符型用户界面，通常指控制台；另外一种就是图形用户界面，通常是指 X Window 系统。我们下面只讲述 X Window 系统。

2.3.3.1. X Window 简介

X 是一个客户/服务器型的视窗系统（图 2.20）。和其它视窗系统一样，你可以在它的视（Display，一套键盘、鼠标及显示设备）上同时运行多个应用程序，而每个应用程序都拥有自己的一个主窗口。应用程序并不是直接和视打交道，X 的视是由 X 服务器（X Server）控制，应用程序要显示什么东西只需通过某个指定的通讯通道向 X 服务器发出申请就行了。除了处理应用程序的显示请求之外，X 服务器还负责接收鼠标、键盘事件，并将它们送到相应的应用程序。这里，应用程序被称为 X 客户（X Client）。

注意这里的通讯通道可以是网络也可以是进程间通讯常常提及的共享内存。因此，X

系统的客户和服务器可以在同一台机器上，也可以不在同一台机器上。一个应用程序不管是在本地运行，还是在远程某台机器上运行，都可以实现在本地显示。X 系统的这种特性即所谓的网络透明。

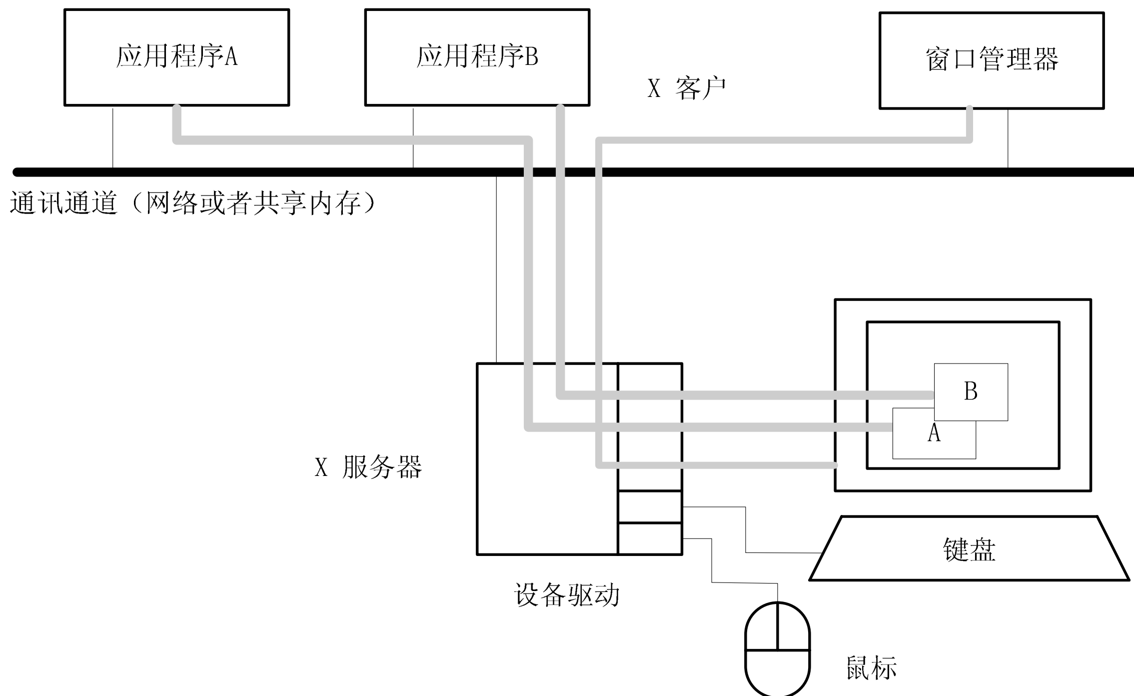


图 2.20. X Window 系统一般架构

需要注意的是 X 服务器主要提供视服务，即接收键盘鼠标等输入设备事件并将它们传送给对应的应用程序，同时接收并处理应用程序发出的显示请求，而对诸如按钮、窗口这样的用户界面并没有作定义或者有相应的实现，这些东西都留给了 X 客户程序来自行完成。

在 X Window 系统中，用户界面进一步被分成两个部分，一部分是窗口管理器（Window Manager），它负责管理你的桌面环境，包括窗口的移动、缩放、创建和注销等等；另外一部分是应用程序界面，这部分的功用包括决定标题栏的显示方式等等，它最终决定你的应用程序的观感。通常来讲，对于一个 X Window 系统而言，总是有一个独立的客户程序来负责实现窗口管理器的功能（图 2.20）。而由于应用程序界面是内置于应用程序的，因此你可以发现在同一个显示屏上会有不同风格的界面类型出现。

2.3.3.2. X 服务器

X 服务器是整个 X 体系的基础。它的主要功能包括：

(1) 处理显示请求：接收从客户程序发出的显示请求并加以处理。此类显示请求包括窗口的创建、配置和注销，字体处理和文本显示，画线、区域填充，位图显示等。
(2) 处理键盘鼠标事件：X 服务器检测键盘击键和鼠标摁键事件，并在必要时候将它们传送给相应的客户程序。
(3) 客户间通讯：客户程序之间通讯需要传递一些信息，在这里，X 服务器起到一个中介桥梁的作用。
(4) 网络连接处理：X 服务器需要同时维护与本地和远程客户程序连接。

● 显示服务

X 的显示基础是窗口（Window）。所有一切的输出都在窗口中。在 X Window 系统中，X 是及其廉价的资源，你可以轻松拥有数百个窗口，而不像其它视窗系统中，由于每个窗口对应一个打开的文件，从而受存储资源的限制，你能拥有数十个窗口就不错了。

在 X Window 系统中，所有的窗口组成了一个树状的层次结构，在这个树的根部是所有窗口的根窗口，它覆盖了整个显示屏。每个应用程序都有一个主窗口，而这些主窗口都是根窗口的子窗口。而每个应用程序会在主窗口之内继续创建新的窗口，这些窗口又成为了相应主窗口的子窗口，如此循环下去形成了一个严整的树形结构（图 2.21）。

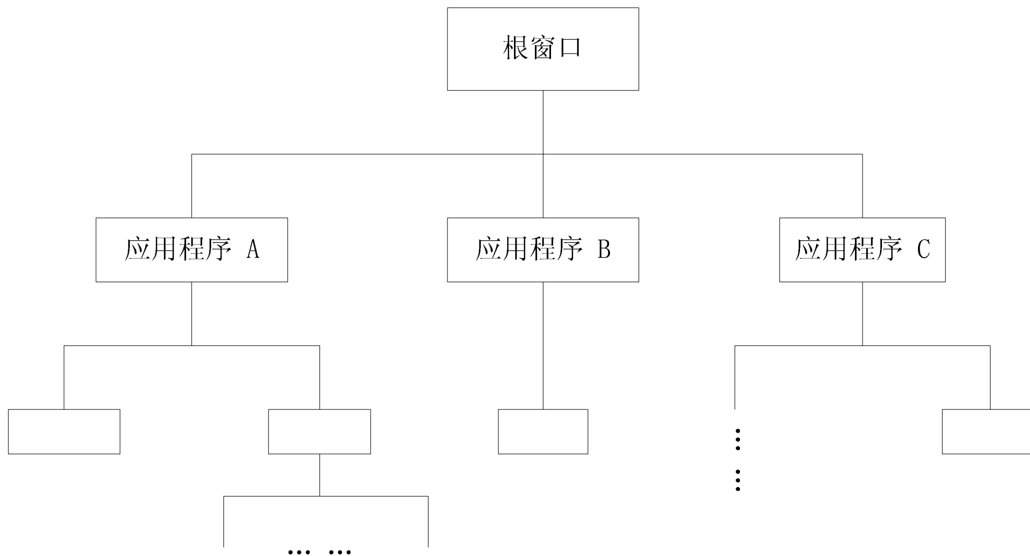


图 2.21. X Window 系统的树形视窗结构

在 X Window 系统中，允许多屏显示，即一个视可以拥有多个显示屏幕，从而能够拥有更大的视野。另外，它还支持虚拟显示屏：你可以通过移动鼠标从一个虚拟屏幕转换到另外一个虚拟屏幕，而这些虚拟屏幕共用同一个显示设备，这样做的结果就是无形中扩大了显示设备的视野。

理论上讲，一个窗口可以无限大，但是，除了根窗口之外，所有的窗口都会因为是某个窗口（父窗口）的子窗口所以其可见部分不能超出其父窗口的可视范围，换句话说，任何一个窗口的显示区域都不可能超过根窗口的显示区域。

在 X Window 系统中，所有的视窗都是画出来的，当某个视窗由不可见到可见的时候，X 服务器会向相应的客户程序告知这个事件，而客户程序会调用相应的计算逻辑来重复原来的画窗口请求过程。

● 客户/服务器间通讯

客户程序的显示输出实际上是一个请求/响应过程。由于客户程序向 X 服务器发送的对象显示请求，而并非位图显示请求，所以即便是在网络环境中，这种请求/响应过程也是很高效的，使得这种通讯即便在很低的带宽条件下都能顺畅进行。图 2.22 给出了一个 X 文本显示请求的示意图。

而服务器则是使用“事件”将键盘鼠标的动作或者某些状态的改变通知给客户程序，而客户程序在接收到这些事件之后会做出相应的反应。

为了提高客户/服务器之间通讯的吞吐量，它们之间的通讯采取了异步方式进行。在一

个性能稳定的网络传输环境中或者仅仅只是本地机的共享内存的情况，这种通讯方式是很有效的。当然，也可以强制采用同步传输方式。

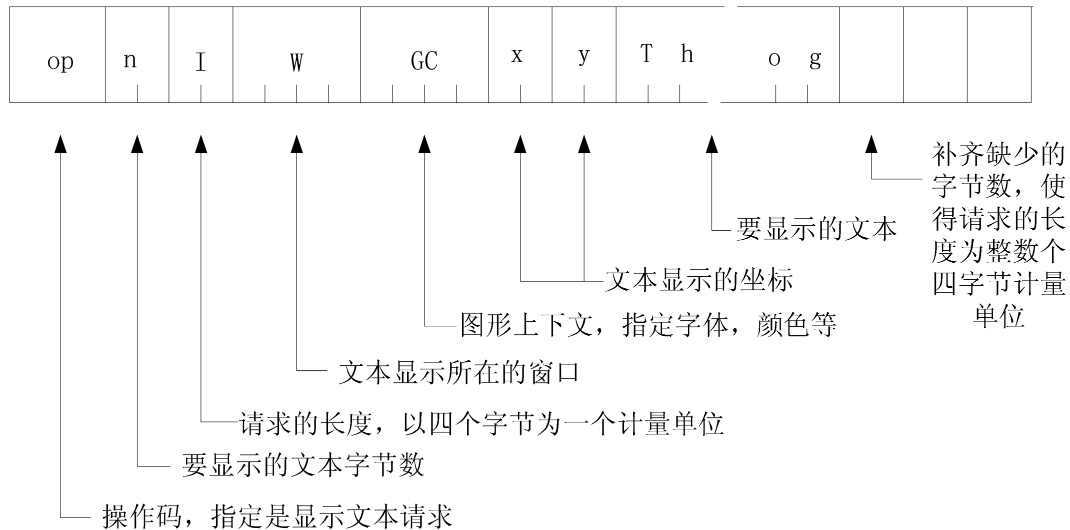


图 2.22. 一个文本显示请求的示意图

2.3.3.3. 窗口管理器

窗口管理器至少应当包括以下功能：

(1) 配置应用程序窗口：这其中包括窗口的移动、缩放等。
(2) 窗口的徽标化：即通过某一个特定的徽标来代表某个窗口，从而可以在该窗口不用的时候，节省屏幕显示空间，而在要使用该窗口时，可以很方便地通过该徽标打开这个窗口。
(3) 窗口的布局管理：比如说多个窗口是平铺还是叠放等等。
(4) 键盘焦点管理：确定键盘事件应当送给哪个应用程序。
(5) 调色板管理：由于各个应用程序的显示可能需要适合自己的某个特定的调色板。窗口管理器有责任当对应各个应用程序的窗口可见时，安装上对应的调色板，而在它不可见的时候，再把这个调色板卸载下来。

另外，窗口管理器还可能提供一些可选择的功能：

(1) 下拉菜单。
(2) 可配置性：即窗口管理器本身的某些操作特性或者快捷方式是可配置的。
(3) 组窗口操作：有些窗口管理起允许你选中一组窗口，然后对它们同时执行某一个操作，比如说，整体移动、整体徽标化等。
(4) 虚拟桌面：提供一个比物理显示区域大的虚拟桌面。
(5) 可编程性：最近出现的一些窗口管理器是可编程的。

第三章 嵌入式 Linux

3.1. 嵌入式 Linux 内核

3.1.1 嵌入式 Linux 综述

Linux 正在嵌入式开发领域稳步发展，这是因为 Linux 源代码开放并遵循 GPL（GNU Public License），所以任何对将 Linux 定制于 PDA 或者其他手持设备感兴趣的人都可以从因特网免费下载其内核和应用程序，并开始移植或开发。同时，不同的 Linux 的改良版本也迎合了嵌入式和实时的应用。它们包括 Fsmlabs 公司的 RTLinux（实时 Linux）、uCLinux（用于非 MMU 设备的 Linux）、MontaVista 的 Hard Hat Linux（用于 ARM、MIPS、PPC 的 Linux 分发版）、ARM-Linux（ARM 上的 Linux）和其它 Linux 系统。

嵌入式 Linux 主要可以分为两类：第一类是在利用 Linux 强大功能的前提下，使它尽可能的小，以满足许多嵌入式系统对体积的要求，如 uClinux；第二类是将 Linux 开发成实时系统尤其是硬（firm）实时系统，应用于一些关键的控制场合，如 RTLinux、Hard Hat Linux 等。

3.1.2 uCLinux

目前，全球每年生产的 CPU 的数量在二十亿颗左右，超过 80%应用于专用性很强的各类嵌入式系统。其中又有相当一部分面向低端市场。为降低硬件成本及运行功耗，有一类 CPU 在设计中取消了内存管理单元（Memory Management Unit，简称 MMU）功能模块。如 Motorola 公司的 M68328、M68EN322 、MC68360、DragonBall 系列(68EZ328、68VZ328)，ColdFire 系列(5272、5307)，ARM7TDMI and MC68EN302、ETRAX、Intel i960、PRISMA、Atari 68k 等等。

最初，运行于这类没有 MMU 的 CPU 之上的都是一些很简单的单任务操作系统，或者更简单的控制程序，甚至根本就没有操作系统而直接运行应用程序。在这种情况下，系统无法执行复杂的任务，或者执行时效率很低。此外，所有的应用程序都需要针对不同的硬件平台进行重写，这要求程序员十分了解硬件特性。这些都阻碍了应用于这类 CPU 之上的嵌入式产品开发的速度。

标准的 Linux 内核采用虚拟内存管理技术来提高系统运行效率，这种设计在硬件上需要有微处理器内嵌的内存管理单元（MMU）的支持。

因此，在许多没有 MMU 的嵌入式应用中标准 Linux 内核关于虚拟内存管理部分的代码就变得冗余了，甚至会对系统整体性能产生负面的影响。uCLinux 正是为了解决这一问题而开发的。它是一种专为嵌入式系统设计的 Linux，这里字母 u 即为 micro（微小）的意思，字母 C 是 Control 的缩写，可见 uCLinux 是为微控制领域量身定做的 Linux 版本。uCLinux 的设计思想就是通过对标准 Linux 内核的裁减，去除虚拟内存管理部分代码，并且对内存分配进行优化，从而达到提高系统运行效率的目的。它经过各方面的小型化改造，形成了一个高度优化的、代码紧凑的嵌入式 Linux，虽然它的体积很小，但是仍然保留了 Linux 的大多数的优点：稳定、良好的移植性、优秀的网络功能、完备的对各种文件系统的支持、以及标准丰富的 API。它的主要特征如下：

(1) 通用 Linux API
(2) 内核体积 ＜ 512 KB
(3) 内核 +文件系统＜900 KB
(4) 完整的 TCP/IP 协议栈
(5) 支持大量其它的网络协议
(6) 支持各种文件系统，包括 NFS、ext2、ROMfs and JFFS、MS-DOS 和 FAT16/32

3.1.2.1 uClinux 的内存管理

内存是操作系统内核管理的最重要的资源之一。每个进程都有自己逻辑上独立的内存空间，操作系统内核的一个重要功能就是通过对内存的管理，来保证进程空间的独立性和安全性。

计算机系统中包含有不同形式的几种存储器,这些存储器的容量与存取速度各不相同，速度的差距相比容量的差距来说要小几个数量级——这种差距尤其体现在 RAM 和磁盘之间——因此如果能够用容量较大的低速存储介质磁盘来模拟相对高速的介质 RAM，就能够提高系统的运行性能。

虚拟内存的设计思想就是系统通过不断的将进程正在使用的部分装入 RAM，而将其余部分（包括暂时没有用到的数据和堆栈等）存储到磁盘上，从而使对 RAM 需求较大的程序可以在 RAM 容量相对较小的系统中顺利的运行。

大多数虚拟内存系统，如标准版本的 Linux 系统，采用分页（Paging）技术。分页技术是把系统的内存划分成页面，每个页面可以独立的在内存和磁盘之间进行交换，特定平台的页面大小是固定的，通常是 4K。

由于虚拟内存的存在，产生了两种不同的地址空间：物理地址空间和逻辑地址空间。物理地址空间是指一个系统中可用的真实的硬件地址空间，比如一个具有 128M 内存的系统，有效的物理地址空间就是 0 到 0x8000000。逻辑地址空间是一个进程“认为”自己拥有的地址空间，因而不同进程的逻辑空间都是相同的。物理地址和逻辑地址都是按照页面来进行划分的。

逻辑地址和物理地址之间的转换工作是由内核和内存管理单元（MMU）共同完成的，许多现代 CPU 中都集成有 MMU 模块。内核告诉 MMU 如何为每个进程分配逻辑页面，而 MMU 在进程提出内存请求时完成实际的转换工作。当地址转换无法完成的时候，如，由于给定的逻辑地址不合法或者由于逻辑页面没有对应的物理页面时，MMU 就给内核发送“页面错误”信号。MMU 也负责增强内存保护，如果一个应用程序试图在它的内存中对一个已经标明是只读的页面进行写操作，MMU 报错，并交由内核处理。MMU 的主要好处在于快速，为了获得同样的效果，缺少 MMU 时操作系统将不得不使用软件为每个进程的每一次内存引用进行校验，可以想象，这个操作是非常频繁的，结果系统的处理能力很大程度上消耗在了对内存引用的校验上，系统性能受到极大的影响。而在有 MMU 参与的情况下，内核只是偶尔参加工作，比如在发生页面错误的时候，这些情况与全部的内存引用数量相比是十分微小的。

3.1.2.2 uCLinux 内核结构

由于去除了与 MMU 相关的代码，uClinux 可以生成精简的内核与应用程序，这一特点对于嵌入式系统来说优势是非常显著的，同时也使得 uClinux 与标准 Linux 有了本质的区别。但是，这种差别对一般应用程序开发者来说几乎是透明的，熟悉 Linux 的程序员会发

现，在uClinux下开发应用程序与标准Linux没有太大区别。但是作为维护和改进操作系统内核的程序员而言，了解这种区别就非常必要了。由于没有对MMU的支持，uClinux缺乏内存保护机制和虚拟内存模块，这是uClinux与标准Linux最本质的区别所在。由此一些底层的系统调用也有不同程度的变化。UCLinux的内核结构图如图3.1.

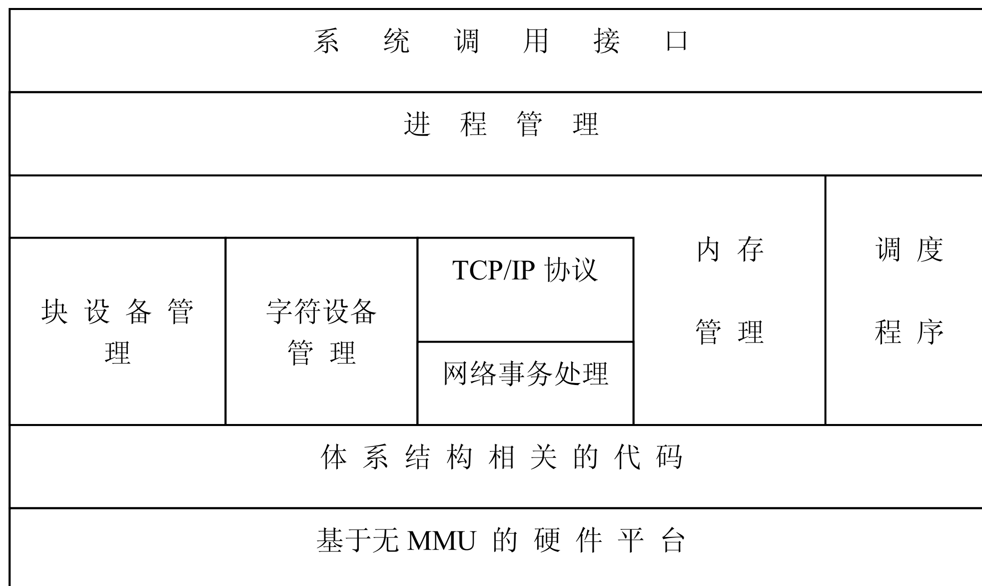


图 3.1. UCLinux 内核结构

3.1.2.3 内存保护

没有内存保护的系统最突出的问题是当指向非法的内存地址的时候，系统无法及时得到错误信息，这样就有可能导致系统运行不稳定，甚至系统崩溃。所以在这样的系统上开发程序，必须在编程和调试过程中保持警惕，保证系统的健壮性和安全性。

运行没有虚拟内存模块的Linux，有三个主要问题是需要特别注意的：

(1) 必须保证由内核载入的进程能够在各自独立的内存空间中运行。保证这一点的一种方法是在程序载入内存以前就确定进程将占据的地址范围。另一种方法是在程序代码生成时只使用相对地址——uClinux对这两种方法都有支持。

(2) 内存的分配和释放都是直接在一维的内存映像中进行的，非常频繁的动态内存分配会在系统内存中产生很多碎片，从而导致系统内存不足。因而，为了改进采用动态内存分配方式的应用程序的执行性能，uClinux通过改写malloc()系统调用，使内存在一个内存块区的池中进行预分配。

(3) uClinux 不支持虚拟内存，系统无法保证不同内存页面载入到相同的地址，因此无法实现内存页面的换入与换出。因而在嵌入式系统中，一般而言，那些内存需求比系统实际具有的物理内存大的程序无法运行。

3.1.2.4 编程接口的改变

由于在处理器硬件上缺乏MMU模块，因而系统的编程接口也相应的有了一些变化。其

中最大的变化就是没有了 fork( )和 brk( )系统调用。

fork( )系统调用为进程复制一个子进程。标准 linux 中 fork 系统调用是通过写拷贝(copy-on-write)内存页面来实现的。在没有 MMU 支持的情况下，uClinux 不能可靠的复制一个进程，也无法进行页面的写拷贝。

uClinux 通过 vfork( )系统调用来替代 fork( )。当父进程调用 vfork( )创建子进程时，两个进程共享父进程的内存空间，包括堆栈，然后 vfork()挂起父进程直到子进程调用 exec( )创建自己的内存空间或者调用 exit( )退出。

下面通过对 hyped(一个简单的网络精灵程序)的分析，可以说明标准的 fork()调用的作用。hyped 通过不断侦听某一知名端口来确定客户连接。当客户连接建立以后，hyped 给出连接的信息，然后调用 fork( )，产生子进程接受客户连接的信息，为客户连接提供服务。同时，释放父进程继续侦听该端口。

需要注意的是系统的多任务特性并没有因为缺乏 fork( )系统调用受到影响。但是对于像 hyped 程序这样依赖于 fork( )调用的应用程序就需要进行重写。由于父子进程公用同一内存空间，所以在一些特殊的情况下，父子进程的行为需要改变，以确保进程的安全与健壮。此外，很多依靠子进程来完成主要任务的较新型的程序需要在负载很重的情况下，仍然保持系统的交互能力。这些程序如果要在 uClinux 下实现相同的功能可能也需要彻底的改写。

uClinux 既没有自动增长的堆栈，也没有 brk( )系统调用，所以用户空间的程序必须用 mmap( )命令来分配内存。一个程序的堆栈空间大小在编译时就被确定了。

为了适应缺乏 MMU 单元的硬件平台，uClinux 针对标准的 Linux 内核做了改动。 内核中的内存管理模块,通过软件提供对内存的基本管理,摆脱了对 MMU 单元的依赖。在 uClinux 内核目录树中，/mmnommu 目录下的代码就是用于替换在 MMU 单元支持下工作的/mm 目录代码的。一些子模块被添加、修改和重写了。负责内核和用户空间内存分配与释放的例程被重写了。支持透明的分页交换机制的代码被去除，添加了支持 PIC(position independent code)的程序载入机制。可执行文件采用了一种头部非常精简的格式：flat。另一些支持 elf 格式的程序载入机制，采用由内核在程序运行前决定的绝对地址。

这两种方式都有各自的优势与不足：传统的 PIC 格式精简而快速，但是针对特定的体系结构，应用程序有尺寸上的限制，比如本方案采用的 m68k 平台，PIC 格式的应用程序大小不得超过 32k。采用绝对地址的方式可以突破这种尺寸上的限制，但是有可能会导致程序运行时的溢出。

3.1.2.5 uCLinux 的应用程序库

uCLinux 小型化的另一个做法是重写了应用程序库。uClibc 是专门针对开发嵌入式 Linux 系统的 C 函数库,相对于越来越大且越来越全的 GNU C Library（glibc）库，uClib 对 libc 做了精简。它远远小于 glibc，但是几乎所有 glibc 支持的应用都能被 uClibc 很好的支持。从 glibc 向 uClibc 移植应用程序通常只需要重新编译一下即可。此外，uClinux 还提供 uClibm 数学库。

uCLinux 对用户程序采用静态链接的形式，这种做法会使应用程序变大，但是 uClinux 基于无 MMU 的内存管理决定了只能采取这样的形式，同时这种做法也更接近于通常嵌入式系统的做法。

3.1.2.6 uCLinux 内核运行方式

uCLinux 的内核有两种可选的运行方式：可以在 flash 上直接运行，也可以加载到内存中运行。后者可以减少内存需要。

Flash 运行方式(XIP)：把内核的可执行映像烧写到 flash 上，系统启动时从 flash 的某个地址开始逐句执行。这种方法实际上是很多嵌入式系统采用的方法。

内核加载方式：把内核的压缩文件存放在 flash 上，系统启动时读取压缩文件在内存里解压，然后开始执行，这种方式相对复杂一些，但是运行速度可能更快（RAM 的存取速率要比 Flash 高）。

3.1.2.7. uClinux 支持的文件系统

uCLinux 系统通常采用 romfs 文件系统，这种文件系统相对于一般的 ext2 文件系统要求更少的空间。空间的节约来自于两个方面：首先内核支持 romfs 文件系统比支持 ext2 文件系统需要更少的代码；其次 romfs 文件系统相对简单，在建立文件系统超级块（superblock）需要更少的存储空间。Romfs 文件系统不支持动态擦写保存，对于系统需要动态保存的数据采用虚拟 ram 盘/JFFS 的方法进行处理（ram 盘将采用 ext2 文件系统）。

3.2. 嵌入式设备的文件系统

嵌入式操作系统需要一种以结构化格式存储和检索信息的方法；这就需要文件系统的参与。嵌入式存储设备通常主要是 RAM 和作为永久存储媒质的 Flash。用户可以根据可靠性、健壮性和增强功能等需求来选择这些设备的文件系统的类型.

3.2.1. 闪存（Flash Memory）介绍

闪速存储器（Flash Memory）又称 PEROM（Programmable and Erasable Read Only Memory）,是 Intel 公司在 80 年代末 90 年代初推出的，由于它的众多优点而深受用户的青睐。Flash Memory 的两个主要特点是可以按整体/扇区擦除和按字节编程。它是完全非易失的，可以在线写入,并且可以按页连续字节写入,读出速度高。Flash 芯片划分成很多扇区，把一位从 0 重置为 1 不能通过对该位单独操作来实现，而必须擦除整个扇区。Flash 芯片的寿命就用擦除周期来衡量。通常的寿命为每个扇区可擦除 100，000 次。为了避免任意一个扇区在其他扇区之前达到这个极限，大多数 Flash 芯片用户会尽量保证擦除次数在各扇区之间均匀分布，这一过程称为“磨损均衡”(wear leveling)。

3.2.2. 第二版扩展文件系统 Ext2fs（Extended 2 Filesystem）

Ext2fs 是 Linux 事实上的标准文件系统，它已经取代了它的前任 — 扩展文件系统（或 Extfs）。Extfs 支持的文件最大为 2 GB，支持的最长文件名为 255 个字符，而且它不支持索引节点（包括数据修改时间标记）。Ext2fs 对 Extfs 进行了改进；它的优点是：

(1) Ext2fs 支持达 4 TB 的内存。

(2) Ext2fs 文件名称最长可以到 1012 个字符。
(3) 当创建文件系统时，管理员可以选择逻辑块的大小（通常大小可选择 1024、2048 和 4096 字节）。
(4) Ext2fs 实现了快速符号链接：不需要为此目的而分配数据块，并且将目标名称直接存储在索引节点（inode）表中。这使性能有所提高，特别是在速度上。

因为 Ext2 文件系统的稳定性、可靠性和健壮性，所以几乎在所有基于 Linux 的系统（包括台式机、服务器和工作站 — 甚至一些嵌入式设备）上都使用 Ext2 文件系统。然而，当在嵌入式设备中使用 Ext2fs 时，它有一些缺点：

(1) Ext2fs 是为象 IDE 设备那样的块设备设计的,这些设备的逻辑块大小是 512 字节，1 K 字节等这样的倍数。这不太适合于扇区大小因设备不同而不同的闪存设备。
(2) Ext2fs 没有提供对基于扇区的擦除 / 写操作的良好管理。在 Ext2fs 中，为了在一个扇区中擦除单个字节，必须将整个扇区复制到 RAM，然后擦除，然后重写入。考虑到闪存设备具有有限的擦除寿命，在此之后就不能使用它们，所以这不是一个特别好的方法。
(3) 在出现电源故障时，Ext2fs 不是防崩溃的。
(4) Ext2fs 不支持磨损均衡，因此缩短了扇区 / 闪存的寿命。
(5) Ext2fs 没有特别完美的扇区管理，这使设计块驱动程序十分困难。

3.2.3. 临时文件系统 tmpfs（Temporary Filesystem）

当 Linux 运行于嵌入式设备上时，该设备就成为功能齐全的单元，许多守护进程会在后台运行并生成许多日志消息。另外，所有内核日志记录机制，象 syslogd、dmesg 和 klogd，会在 /var 和 /tmp 目录下生成许多消息。由于这些进程产生了大量数据，所以允许将所有这些写操作都发生在闪存是不可取的。由于在重新引导时这些消息不需要持久存储，所以这个问题的解决方案是使用 tmpfs。

tmpfs 是基于内存的文件系统，它主要用于减少对系统的不必要的闪存写操作这一唯一目的。因为 tmpfs 驻留在 RAM 中，所以写 / 读 / 擦除的操作发生在 RAM 中而不是在闪存中。因此，日志消息写入 RAM 而不是闪存中，在重新引导时不会保留它们。tmpfs 还使用磁盘交换空间来存储，并且当为存储文件而请求页面时，使用虚拟内存（VM）子系统。tmpfs 的优点包括：

(1) 文件系统大小可以根据被复制、创建或删除的文件或目录的数量来缩放,使得能够最理想地使用内存。
(2) 因为 tmpfs 驻留在 RAM，所以读和写几乎都是瞬时的。即使以交换的形式存储文件，I/O 操作的速度仍非常快。

tmpfs 的一个缺点是当系统重新引导时会丢失所有数据。因此，重要的数据不能存储在 tmpfs 上

3.2.4. 日志闪存文件系统版本 2－JFFS2（Journalling Flash Filesystem）

3.2.4.1 概述

瑞典的 Axis Communications 开发了最初的 JFFS，Red Hat 的 David Woodhouse 对它进行了改进。 第二个版本，JFFS2，作为用于微型嵌入式设备的原始闪存芯片的实际文件系统而出现。JFFS2 文件系统是日志结构化的，这意味着它基本上是一长列节点。每个节点包含有关文件的部分信息 — 可能是文件的名称、也许是一些数据。相对于 Ext2fs，JFFS2 因为有以下这些优点而在无盘嵌入式设备中越来越受欢迎：

(1) JFFS2 在扇区上直接执行闪存擦除／写／读操作，比 Ext2 文件系统效率更高。
(2) JFFS2 提供了比 Ext2fs 更好的崩溃／掉电安全保护。当需要更改少量数据时，Ext2 文件系统将整个扇区复制到内存（DRAM）中，在内存中合并新数据，并写回整个扇区。这意味着为了更改单个字，必须对整个扇区（64 KB）执行读／擦除／写例程 — 这样做的效率非常低。要是运气差，当正在 DRAM 中合并数据时，发生了电源故障或其它事故,那么将丢失整个数据集合,因为在将数据读入 DRAM 后就擦除了闪存扇区。JFFS2 附加文件而不是重写整个扇区，并且具有崩溃／掉电安全保护这一功能。
(3) 这可能是最重要的一点：JFFS2 是专门为象闪存芯片那样的嵌入式设备创建的，所以它的整个设计提供了更好的闪存管理。

3.2.4.2. JFFS 的设计原理：

JFFS 结合闪存的特殊属性，对标准的日志文件系统结构进行了简化，它保证当系统非正常关闭时操作的可靠性，是闪存设备上保存经常修改数据的文件系统的理想选择。

- **存储形式**

最初的 JFFS 是一个纯粹的 Log-structured 日志型文件系统（LFS)。包含数据的节点和元数据顺序存储在闪存芯片上，在存储空间中严格按照线性推进。在最初的 JFFS 中，日志中只有一种节点。每一个这样的节点对应一个索引节点。这种节点的头部包含它所属的索引节点号和这个索引节点的当前文件系统的元数据。此节点还可能携带不定长的数据。即使节点所属的索引节点相同，它们的顺序也是不同的。每个节点都保存一个版本号。每个节点都比和它属于同一索引节点的所有老节点拥有更高的版本号。除了普通索引节点的元数据，每个节点还包含所属索引节点的名字和父索引节点的索引节点号。此外，每个节点还可能包含一定长度的数据，当数据存在时，节点还记录了数据在文件中的位置。

- **加载和文件操作**

当被加载时，系统扫描整个存储介质，每个节点都将被读取和记录。储存在节点中的数据提供了充足的信息来重建整个目录层次和每个索引节点各部分数据到物理位置的完全映射。一旦文件系统被加载，JFFS 将储存上述的所有信息，每次目录查找将立即由保存在系统核心中的数据结构完成，如果要读文件，数据将立即从存储介质的正确位置读取到提供的缓冲区中。要改变元数据，比如改变所有者或者权限，直接在日志中写入一个正确记

录更新的元数据的新节点就可以实现。写文件也是一样，只不过所写的节点还包括一部分新数据。

- 清除机制

当节点包含的数据和比它新的节点的数据重复，这个节点将被抛弃，同样，那些不包含数据并且元数据已经被新节点更新的节点也将被抛弃。被这些要抛弃的节点占据的空间被称“脏空间”（dirty space）。索引节点的删除通过在索引节点的元数据中设置一个删除标志来实现。所有与之相关的节点将被加上同样的标志，当和此节点相关的最后的文件操作关闭之后，它所有的节点都将被抛弃。

JFFS 的操作原则非常简单，它采用向存储介质上写入一个新节点的方法来记录文件系统的每次改变，直到存储空间被用完。这时，系统就需要收回脏空间。

清除机制的目标是对日志中的第一个擦除块进行清除。每次回收时，先检查日志头部的节点。如果它是闲置的，它将被跳过，日志头部移至下一个节点。如果这个节点仍然有效，它将被抛弃。清除机制将在日志的尾部写入包含数据或元数据的新节点。新节点写入的内容将包括当前的有效数据，它包含了原来节点的数据。如果被抛弃的节点的一部分数据已经被别的节点更新，则新节点的数据自然有一部分与原节点的不同。

按照这种方式，清除机制将日志的头部不断推进，直到一个完整的擦除块被闲置，这时这个块将被擦除并加入日志的尾部等待重新被使用。通过以上方法，JFFS 在日志尾部维持系统的最新状态，从而实现了系统的防断电机制。同时它实现了 Log-structured 日志文件系统和闪存特性的结合，修改后的数据被直接写到日志尾部新的可用空间上，避免了擦除原来区域然后回写数据这样的费时操作，从而提高了擦除的效率。

- 均衡磨损

上述清除机制在存储介质上线性推进，写入新节点，从而擦除日志中的旧块，这样确实可以提供完善的磨损均衡，每个块都保证被擦除同样的次数。然而当被擦除的这个块只包含
有用的节点时，擦除依然不可避免。如果文件系统包含的主要是静态数据，那么这些静态数
据将在存储介质上整体移动。这意味着有些块会在不必要的时候被擦除。

- JFFS2 的改进

改进的 JFFS 的组织更加复杂，有别于简单的循环日志格式，从而避免了严格按顺序清除机制带来的过多的不必要的擦除。在改进的 JFFS 中，每个擦除块都被单独处理。这意味着清除机制在一个块执行之后，可以作出智能决策，选择执行的下一个块，从而提高了效率。每个擦除块都处在一定的状态。它的状态首先取决于它的内容。改进的 JFFS 保存一定数量的链表（linked list）。链表的每个结构代表一个单独的擦除块。在 JFFS 文件系统正常运行时，大多数的擦除块会在 clean_list 或者 dirty_list 中出现，他们分别代表了存满有效节点的块和至少包含一个被抛弃节点的块。在一个新的文件系统中，多数的擦除块在 free_list 上，这些擦除块只包含一个有效节点，作为该擦除块被完全并且正确擦除的标志。
改进的 JFFS 的操作基本上和最初的 JFFS 相同，它顺序的写入不同类型的节点，当一个块写
满时，从 free_list 中取出一个新块，从新块的开始处继续写入。当 free_list 的大小到达一个下限时，清除机制会启动，擦除旧块、产生新块。

清除机制使用一个非常简单的随机方法来决定选择哪一个块。如果一个随机量除以 100 的余数非 0，就从 dirty_list 中选择一个块。否则，如果取余结果为 0，就从 clean_list 中选择一个块。用这种方法，优化了清除机制，在 99％的情况下，会重新利用了那些含有

被抛弃节点的块，提高了擦除效率，减少了不必要的擦除；在 1%的情况下，擦除存满有效节点的块，来保证数据在闪存上循环移动，从而达到磨损均衡。这样，在经过一段时间以后，可以使数据在存储介质上均匀的分布，充分保证不会有擦除块在其他擦除块之前损坏。

由于擦除操作通常是在启动清除机制的时候进行的，如果单纯的考虑磨损均衡，会导致频繁的不必要的擦除。而如果只考虑擦除的效率，不考虑均衡磨损，带来的只能是闪存使用寿命的降低。采用上述方法，巧妙的解决了清除机制和磨损均衡之间的矛盾，实现了两者的高效的结合。

3.3 嵌入式用户界面

3.3.1. GUI 开发工具综述

从用户的观点来看，图形用户界面（GUI）是系统的一个最至关重要的方面：用户通过 GUI 与系统进行交互，所以 GUI 应该易于使用并且非常可靠。此外，它不能占用太多的内存，以便在内存受限的微型嵌入式设备上无缝执行。所以，它应该是轻量级的，并且能够快速装入。另一个要考虑的重要方面涉及许可证（License）问题。一些 GUI 分发版具有允许免费使用的许可证，甚至在一些商业产品中也是如此。另一些许可证则要求在将 GUI 合并入项目中时支付版税。

大多数开发人员可能会选择 XFree86,因为 XFree86 为他们提供了一个能使用他们喜欢的工具的熟悉环境。但是市场上较新的 GUI，象 Century Software 的 Microwindows（Nano-X）和 Trolltech 的 QT/Embedded，与 X 在嵌入式 Linux 的竞技舞台中展开了激烈竞争，这主要是因为它们占用很少的资源、执行的速度很快并且具有定制窗口构件的支持。

3.3.1.1. Xfree86 4.X（带帧缓冲区支持的 X11R6.4）

XFree86 Project, Inc. 是一家生产 XFree86 的公司，该产品是一个可以免费重复分发、开放源码的 X Window 系统。X Window 系统（X11）为应用程序以图形方式进行显示提供了资源，并且它是 UNIX 和类 UNIX 的机器上最常用的窗口系统。它很小但很有效，它运行在为数众多的硬件上，它对网络透明并且有良好的文档说明。X11 为窗口管理、事件处理、同步和客户机间通信提供强大的功能，并且大多数开发人员已经熟悉了它的 API。它具有对内核帧缓冲区的内置支持，并占用非常少的资源，这非常有助于内存相对较少的设备。X 服务器支持 VGA 和非 VGA 图形卡，它对颜色深度 1、2、4、8、16 和 32 提供支持，并对渲染提供内置支持。最新的发行版是 XFree86 4.1.0。

它的优点包括：

(1) 帧缓冲区体系结构的使用提高了性能。

(2) 占用的资源相对很小，在 600 K 到 700 K 字节的范围内，这使它很容易在小型设备上运行。

(3) 非常好的支持:在线有许多文档可用,还有许多专用于 XFree86 开发的邮递列表。

(4) X API 非常适合扩展。

它的缺点包括：

(1) 比最近出现的嵌入式 GUI 工具性能差。
(2) 此外，当与 GUI 中最新的开发工具，象专门为嵌入式环境设计的 Nano-X 或 QT/Embedded，相比时，XFree86 似乎需要更多的内存。

3.3.1.2. Microwindows

Microwindows 是 Century Software 的开放源代码项目，设计用于带小型显示单元的微型设备。它有许多针对现代图形视窗环境的功能部件。象 X 一样，有多种平台支持。

Microwindows 体系结构是基于客户机/服务器（Client/Server）的。它具有分层设计：最底层是屏幕和输入设备驱动程序（关于键盘或鼠标）来与实际硬件交互。在中间层，可移植的图形引擎提供对线的绘制、区域的填充、多边形、裁剪以及颜色模型的支持。在最上层，Microwindows 支持两种 API：Win32/WinCE API 实现，称为 Microwindows（在 3.3.3 中将详细介绍）；另一种 API 与 GDK 非常相似，它称为 Nano-X。Nano-X 用在 Linux 上。它是象 X 的 API，用于占用资源少的应用程序。

Microwindows 支持 1、2、4 和 8 bpp（每像素的位数）的 衬底显示，以及 8、16、24 和 32 bpp 的真彩色显示。Microwindows 还支持使它速度更快的帧缓冲区。Nano-X 服务器占用的资源大约在 100 K 到 150 K 字节。

原始 Nano-X 应用程序的平均大小在 30 K 到 60 K。由于 Nano-X 是为有内存限制的低端设备设计的，所以它不象 X 那样支持很多函数，因此它实际上不能作为微型 X（Xfree 4.1）的替代品。

可以在 Microwindows 上运行 FLNX，它是针对 Nano-X 而不是 X 进行修改的 FLTK（快速轻巧工具箱 Fast Light Toolkit）应用程序开发环境的一个版本，将在下一节专门介绍。

Nano-X 的优点包括：
(1) 与 Xlib 实现不同，Nano-X 仍在每个客户机上同步运行，这意味着一旦发送了客户机请求包，服务器在为另一个客户机提供服务之前一直等待，直到整个包都到达为止。这使服务器代码非常简单，而运行的速度仍非常快。
(2) 占用很小的资源

Nano-X 的缺点包括：
(1) 联网功能部件至今没有经过适当的调整（特别是网络透明性）。
(2) 还没有太多现成的应用程序可用。
(3) 与 X 相比，Nano-X 虽然近来正在加速开发，但仍没有那么多文档说明而且没有很好的支持。

3.3.1.3. FLTK

FLTK 是一个简单但灵活的 GUI 工具箱，它在 Linux 世界中赢得越来越多的关注，它特别适用于占用资源很少的环境。它提供了您期望从 GUI 工具箱中获得的大多数窗口构件，如按钮、对话框、文本框以及出色的“赋值器”选择（用于输入数值的窗口构件）。还包括滑动器、滚动条、刻度盘和其它一些构件。

针对 Microwindows GUI 引擎的 FLTK 的 Linux 版本被称为 FLNX。FLNX 由两个组件构成：Fl_Widget 和 FLUID。Fl_Widget 由所有基本窗口构件 API 组成。FLUID（快速轻巧的用户界面设计器(Fast Light User Interface Designer, FLUID)）是用来产生 FLTK 源

代码的图形编辑器。总的来说，FLNX 是能用来为嵌入式环境创建应用程序的一个出色的用户界面构建器。

Fl_Widget 占用的资源大约是 40 K 到 48 K，而 FLUID（包括每个窗口构件）大约占用 380 K。这些非常小的资源占用率使 Fl_Widget 和 FLUID 在嵌入式开发世界中非常受欢迎。

优点包括：

(1) 习惯于在象 Windows 这样已建立得较好的环境中开发基于 GUI 的应用程序的任何人都会非常容易地适应 FLTK 环境。

(2) 它的文档包括一本十分完整且编写良好的手册。

(3) 它使用 GPL 进行分发，所以开发人员可以灵活地发放他们应用程序的许可证。

(4) FLTK 是一个 C++ 库（Perl 和 Python 绑定也可用）。面向对象模型的选择是一个好的选择，因为大多数现代 GUI 环境都是面向对象的；这也使将编写的应用程序移植到类似的 API 中变得更容易。

(5) Century Software 的环境提供了几个有用的工具，诸如 ScreenToP 和 ViewML 浏览器。

它的缺点是,普通的 FLTK 可以与 X 和 Windows API 一同工作，而 FLNX 不能。它与 X 的不兼容性阻碍了它在许多项目中的使用。

3.3.1.4. Qt/Embedded

Qt/Embedded 是 Trolltech 新开发的用于嵌入式 Linux 的图形用户界面系统。Trolltech 最初创建 Qt 作为跨平台的开发工具用于 Linux 台式机。它支持各种有 UNIX 特点的系统以及 Microsoft Windows。作为最流行的 Linux 桌面环境之一的KDE就是用 Qt 编写的。

Qt/Embedded 以原始 Qt 为基础，并做了许多出色的调整以适用于嵌入式环境。Qt Embedded 通过 Qt API 与 Linux I/O 设施直接交互。那些熟悉并已适应了面向对象编程的人员将发现它是一个理想环境。而且，面向对象的体系结构使代码结构化、可重用并且运行快速。与其它 GUI 相比，Qt GUI 非常快，并且它没有分层，这使得 Qt/Embedded 成为运行基于 Qt 的程序的最紧凑环境。

Trolltech 还推出了 Qt 掌上机环境（Qt Palmtop Environment，俗称 Qpe）。Qpe 提供了一个基本桌面窗口，并且该环境为开发提供了一个易于使用的界面。Qpe 包含全套的个人信息管理（Personal Information Management (PIM)）应用程序、因特网客户机、实用程序等等。然而，为了将 Qt/Embedded 或 Qpe 集成到一个产品中，需要从 Trolltech 获得商业许可证。（原始 Qt 自版本 2.2 以后就可以根据 GPL 获得 。）

它的优点包括：

(1) 面向对象的体系结构有助于更快地执行

(2) 占用很少的资源，大约 800 K

(3) 抗锯齿文本和混合视频的象素映射

它的缺点是：Qt/Embedded 和 Qpe 只能在获得商业许可证的情况下才能使用。

3.3.2 MicroWindows 剖析

3.3.2.1. 分层设计

Microwindows 从原理上采用分层设计的方法，每一层次都完成特定的功能，并且能够在不影响其它层次的基础上针对不同的应用进行改编或者重写。在最底层，显示屏、鼠标、触摸屏等的驱动程序提供了与交互相关的硬件设备的访问。在中间层，是一个精简的图形引擎，提供了划线、区域填充、多边形等多种基本的图形功能。最上层为图形应用程序提供了丰富的编程接口函数(API)，通过这些接口函数可以定制桌面和窗口的外观。目前 Microwindows 提供两套 API 接口，以便能够更好的适应不同平台的应用程序的移植：

(1) 与 Win32/Win CE 基本兼容的 API

(2) 采用 X 体系的 Nano-X API

3.3.2.2. 设备驱动层

设备驱动程序的接口定义在 device.h 文件中。中间层提供的与设备无关的图形引擎例程就是通过调用设备驱动程序跟硬件设备交互。这就保证了当平台硬件设备发生变化的时候，只需要改写相应的驱动程序，上层的代码都无需修改。

Microwindows 提供基于 Linux 2.2.x 内核的 FrameBuffer 设备驱动程序。FrameBuffer 在 Linux 系统中通过/dev/fb0 设备文件进行工作，通过 mmap( )系统调用将显示缓存映射至系统内存中。

显示屏驱动是系统中最为复杂的驱动程序之一，但在 Microwindows 系统中它被赋予了很好的可移植性。事实上，在系统中实际与硬件设备直接交互的只有有限的几个接口，比如显示驱动程序需要提供的最基本的读写像素点、画水平/垂直线等图形功能。这些例程都是通过对显存中相应的地址内容进行操作来实现其功能的。

3.3.2.3. 设备无关的图形引擎层

Microwindows 系统中最核心的图形函数是在图形引擎层通过调用下层的硬件设备驱动程序实现的。用户应用程序通常不直接调用引擎层的例程，而是调用最上层所提供的编程接口。将核心的图形引擎例程独立于应用程序接口主要是基于以下考虑：核心的例程在 Client/Server 环境中总是驻留在 Server 端，这些例程调用的位图与字体格式经过优化处理，使得执行速度更快，所以这些格式通常与应用程序所使用的不同。另外，核心例程常常使用指针以产生更复杂高效但逻辑性较差的代码，而不采用应用程序通常使用的 ID 号。在 microwindows 的源代码中，核心的例程通常包含在以下各个文件中：

devdraw.c： 与描画和填充线、圆、多边形，文本与位图的显示，颜色转换相关的代码

devclip.c： 与剪切功能相关的代码

devmouse.c：与鼠标指针相关的代码

devkbd.c： 与键盘操作相关的代码

devpalX.c： 与调色板相关的代码

3.3.2.4. API（基于 Win32）

该层的例程负责处理 Client/Server 操作和窗口管理操作，如画 title bar，关闭打开对话框之类。系统提供的两套不同的 API 都是在核心引擎层与设备驱动层之上实现的。Microwindows API 的基本模型都是首先初始化显示屏、键盘和鼠标驱动，然后在一个 select( )循环中挂起，等待事件的到达。当一个事件发生了，消息就被传递到用户程序，在那里被解析为具体的行为事件，然后产生一个图形操作请求，调用核心引擎层的例程进行操作，由 API 层检测并传递例程调用所需传递的参数。从以上描述可以看出 API 层与引擎层的分工不同，API 定义了窗口概念和相对坐标系统。然后所有的相对坐标都将转化为显示屏的绝对像素坐标，并结合图形与显示的上下文，作为参数传递给引擎层进行实际的图形操作。

3.3.2.5. 消息传递机制

在 Microwindows API 之间最基本的通信机制是消息传递。一个消息包含有一个约定的消息号、两个参数：wParam 和 lParam。消息被储存在应用程序的消息队列中，可以通过调用函数 GetMessage( )获取。当等待消息时，应用程序被阻塞。一些消息和硬件事件相关，如 WM_CHAR 消息代表键盘输入、WM_LBUTTONDOWN 代表鼠标左键被按下。同时，窗口的创建与清除事件分别对应 WM_CREAT 和 WM_DESTROY 消息。在通常情况下，每个消息都对应于一个用 HWND 标识的窗口。在获取消息后，应用程序通过调用 DispatchMessage( )分派消息到所对应的窗口处理。当窗口建立的时候，该窗口所对应的各种消息的处理函数同时被定义，所以系统知道向哪一窗口传递消息。

消息传递机制允许核心的 API 通过对应各种事件的消息传递来实现各种功能，如窗口的创建、绘制、移动等等。通常情况下，相关的窗口操作消息都由 DefWindowsProc 函数来进行默认的处理，这样就使得所有窗口的动作在行为上具有一致性，当某一窗口需要特殊的操作时，用户可以通过改写处理程序就可以满足要求。

以下函数直接处理消息

SendMessage：直接将一消息传递给窗口

PostMessage：将消息加入应用程序消息队列中，等待分派

PostQuitMessage：将 WM_QUIT 消息放入队列中，当应用程序读取该消息时中止

GetMessage：将应用程序阻塞直至等待的消息出现

DispatchMessage：将消息分派到对应的窗口处理程序

3.3.2.6. 窗口操作

- **窗口的创建和清除**

一个 Microwindows 应用程序的入口点是 WinMain 函数,而不是通常情况下的 Main()。在 Microwindows API 中，最基本的显示单元是窗口，窗口定义了一个显示区域和与其相关的各种消息的处理函数。可以通过预先定义的类型，如按键(button)、文本框(edit boxes)等来定制窗口，同时也可以由用户定义特殊的窗口类型。无论通过什么方式定义类型，创建窗口和消息通信的方法是相同的。以下就是一些与创建清除窗口相关的函数：

RegisterClass：定义一个新的窗口类型名称和相关的窗口函数

UnRegisterClass：清除一个窗口类型的定义
CreateWindowEx：建立一个类型的窗口实例
DestroyWindow：清除一个窗口实例
GetWindowLong：返回窗口信息
SetWindowLong：设置窗口信息

- **窗口显示、隐藏和移动**

ShowWindow 函数允许设置窗口属性为可视或者隐藏。该属性也可以在窗口创建的过程中，由 CreateWindowEx 实现。窗口的移动包括窗口位置或者大小的变化。当窗口位置改变时，WM_MOVE 系统发送消息；当窗口大小改变时，系统发送 WM_SIZE 消息。

- **窗口绘制**

Microwindows 系统在其它窗口发生移动，导致某一窗口需要被绘制或重新绘制的时候，发送 WM_PAINT 消息给相关的窗口过程。这时，由应用程序决定调用图形操作函数来绘制窗口。Microwindows 为每个窗口维护一个 update 域，当 update 非空时就向窗口发送 WM_PAINT 消息。为了速度方面的考虑，WM_PAINT 消息只在应用程序队列里没有其它消息的情况下才会发送，这保证了应用程序对窗口的重绘可以通过一步完成，而不会被分割成好多步骤。如果不希望等待，可以调用 UpdateWindow 函数强制进行窗口重绘。

3.3.2.7. 客户区域和绝对坐标

每一个窗口在显示屏上绘制的时候都参照显示屏像素点的绝对坐标进行。Microwindows API 允许应用程序编程人员在窗口中不包括标题栏的区域使用以窗口左上角为基准的相对坐标，这部分区域称为客户区域。GetClientRect 函数和 GetWindowRect 函数将返回客户区域和窗口的绝对坐标。ClientToScreen 函数和 ScreenToClient 函数则完成绝对坐标与相对坐标之间的相互转换。

3.3.2.8. 设备上下文

应用程序必须在调用图形绘制 API 函数前设置设备上下文。一些信息如目前采用的坐标系统、当前窗口在程序执行过程中相当长的时间内都是不变的，所以没有必要传递给每一个调用的函数，因而可以通过设备上下文的设置，将这些相对持久的信息通知系统。同时，如当前前景色、当前背景色等很多属性也在设备上下文中设置。
可以通过调用 GetDC 来得到目前的设备上下文，当结束一系列绘制以后，调用 ReleaseDC 函数释放 DC 对象。

3.3.3. Microwindows 的移植和中文化

由于 Microwindows 程序都是使用 C 语言编写的，所以有很强的可移植性。针对 m68k 硬件平台，uCLinux 操作系统对 Microwindows 进行移植，只需要修改 Microwindows 的编译配置文件，采用 m68k 编译器重新编译代码，就可以生成能够在 m68k 平台下运行的程序。而针对 uClinux 的具体情况，还需要在源代码中进行相应的修改。

3.3.3.1. 针对 uCLinux 所作的修改

由于 uClinux 不同于标准 Linux，所以以标准 Linux 内核为支持目标开发的Microwindows源代码也必须做出相应的修改才能适应uClinux系统。
最主要的问题是uClinux不提供fork()系统调用，而以vfork()调用取代。所以在uClinux代码中fork( )的使用需要进行修改。

一个简便的方法就是利用宏定义，在所有用到 fork( )系统调用的文件开头都加上如下的定义：

```
#ifdef uCLinux
#undef fork
#define fork() vfork()
#endif
```

这样就可以非常简便的将所有的fork( )调用都用vfork( )替换掉了。

另外的一个问题是需要在打开FrameBuffer设备/dev/fb0时将显示屏的基本参数传递给设备驱动程序

在scr_fb.c中的fb_open(PSD psd)函数中修改如下：

```
psd->xres = psd->xvirtres = 320;
psd->yres = psd->yvirtres = 240;
psd->linelen = 40;
psd->size = 320*30;
```

3.3.3.2. 中文化处理

为了使 Microwindows 系统实现对简体中文汉字的支持，需要对于引擎层的的devfont.c做相应的修改。

在devfont.c文件中定义了microwindows关于字体操作的核心数据结构和操作。由于Microwindows采用面向对象的设计方法，因而只要重新定义一系列针对简体中文的数据结构和操作函数，向系统注册，就可以完成系统的中文化。

需要重新定义的数据结构与函数如下：

```
static MWFONTPROCS hzk_procs = {
    MWTF_ASCII,          /* routines expect ASCII*/
    hzk_getfontinfo,
    hzk_gettextsize,
    NULL,                /* hzk_gettextbits*/
    hzk_destroyfont,
    hzk_drawtext,
    hzk_setfontsize,
    NULL,                /* setfontrotation*/
    NULL,                /* setfontattr*/
};
```

该结构定义了中文字体操作的一系列方法。

在显示中文文本以前，创建设备上下文时，以HZKFONT为参数调用CreateFont函数创建汉字字体，再调用SelectObject函数将汉字字体设置为窗口绘制的设备上下文，然后调

用 DrawText 函数即可在窗口显示汉字，在完成显示之后调用 DeleteObject 函数将字体对象释放：

```
oldfont=SelectObject( hdc,CreateFont(12,0,0,0,0,0,0,0,0,0,0,0,
FF_DONTCARE|DEFAULT_PITCH,
                    "HZKFONT") );
DrawText(hdc, "你好！", -1, &rect, DT_SINGLELINE|DT_CENTER|DT_VCENTER);
DeleteObject( SelectObject(hdc,oldfont) );
```

3.3.4. 应用实例－电子文本阅读器

基于 uClinux 操作系统，我们在 Motorola DragonBall EZ328 的硬件平台上开发了电子文本阅读器。它提供给用户的功能主要是可以进行中文纯文本的阅读。此外，用户还可以下载文件进行更新。

3.3.4.1 功能要求以及方案选择：

从支持中文显示的要求出发，我们采用了 Microwindows 的图形开发工具，它对中文的支持较好，可以选择不同的字体，并且实现起来也很方便。从用户需要下载文件的要求出发，我们采用了基于闪存的可写文件系统 JFFS。从我们的硬件平台出发，考虑到 EZ328 的处理速度有限，我们采用了 Microwindows 上基于 Win32 的 API，它对系统的资源的占用较 Nano-X 要少。此外，我们采用简单的按键来进行人机交互，实现文件打开、关闭以及翻页浏览等功能。

3.3.4.2 具体实现

- **中文纯文本浏览的实现**

由于 Microwindows 支持中文显示，所以中文的纯文本浏览实现相对比较简单。只要利用系统调用函数打开文本文件，根据显示屏的尺寸、字体大小以及每行的间距计算出需要读取的行数和列数，将每一行读到一个字符数组里面，然后调用 Drawtext 函数进行中文文本的显示。而翻页功能则可以通过读取文本文件不同的位置很方便的实现。

- **对换行符的处理**

不同的操作系统，对于换行的处理是不同的，有的是一个字符，有的是两个字符。所以很多 MS Windows 的文档在 Linux 下打开时在每行的末尾会有一些乱码。在读取这些文件时，必须在每行的末尾进行判断，过滤掉那些乱码，使显示正常。

对英文和汉字混合显示的处理：

由于一个汉字占两个字节，而一个英文只占一个字节，所以在同时显示的时候，在一行的末尾会有可能出现半个汉字的情况，这样也会出现乱码。解决方法有两种，一种是统一对待，汉字英文都占两个字节，但这样会浪费存储空间。考虑到嵌入式设备存储空间非常有限，所以我们采用第二种方法，即在显示时动态判断，如果是半个汉字，则放到下一行显示，从而解决了这一问题。

- **按键交互的实现**

当 Microwindows 应用程序进程完成设备以及窗口的初始化以后，它就停留在 select()

循环，等待事件发生。由于我们采用的按键并不是 Microwindows 支持的标准输入设备（如键盘、鼠标），所以需要进行特殊处理。考虑到按键是开发板上唯一的输入设备，所以我们截断了 select()循环，在进程轮询事件以前先打开按键设备并读取返回值。如果按键没有被按下，则进程睡眠，一旦相应的按键（如打开关闭键、翻页键）被按下，分配给该按键的中断产生并唤醒该进程，同时返回相应的值（参加设备驱动有关章节）。根据得到的返回值，发送相应的窗口消息给应用程序主窗口，由主窗口回调函数采取相应的操作（如打开关闭文件，翻页浏览），从而实现按键与应用程序的交互功能。

- **文件下载的实现**

首先需要实现对 JFFS 的支持，具体步骤如下：

对 uClinux 内核进行重新编译，加入对 JFFS 的支持。然后根据硬件平台的闪存类型，修改驱动程序。对闪存进行分区，划分出合适的存储空间用于保存下载的文件。创建设备节点，将 JFFS 加载到相应的闪存分区。

当加入 JFFS 支持以后，对闪存写文件变得非常方便，所以实现文件下载功能也就非常容易了。我们采用串口进行通讯，在下位机（开发板）上运行文件接收程序，在上位机（PC）上运行文件发送程序，从而实现了文件的下载功能。

第四章 交叉编译

4.1. 编译原理

4.1.1. 基础知识

我们在编写计算机程序时所用的编程语言多为高级语言，如 C/C++、Java 等，而计算机只能执行机器代码，因此需要一种工具来完成从源程序到机器代码的转换，这种转换工具就是编译器。

编译器是将一种语言翻译成另一种语言的计算机程序,它将源程序作为输入，产生用目标语言编写的等价（有时经过优化）程序。目标语言可以是机器代码，也可以是另一种语言，比如汇编等。

编译原理是编译器的技术基础，它在计算机科学技术中占有相当重要的地位。它的出现可以让我们更快捷地编写计算机可运行的程序。同时，编译本身也包含了许多软件技术，它蕴含了计算机学科中解决问题的思路、抽象问题和解决问题的方法。编译原理几乎可以称为计算机专业的“高等数学”。

4.1.1.1. 编译的一般过程

不论编译器的功能多么强大，它的实质都是一样的，都是把某种以数字和符号为内容的高级编程语言转换成机器语言指令的集合。编译程序的基本结构都是相似的，如图 4.1. 所示。

编译过程一般可以分为六步，即词法分析、语法分析、语义分析、中间代码生成、中间代码优化，目标代码生成。有些编译程序在生成目标代码之后，可能还会再增加一步优化操作，对目标代码再次进行优化。每一步分别具有不同的功能，它们整体协作完成一次编译过程。

（1）词法分析：输入源程序，通过对源程序字符串的扫描和分解，将其转化成一个一个的单词符号，这些单词符号构成一个单词序列。单词符号是语言的基本组成部分。

（2）语法分析：把词法分析的单词符号串分解成一个一个的句型或者句子，确定输入的单词符号串是否符合给定的语法。

（3）语义分析：给出各个句型和句子的含义。

（4）中间代码生成：把不同的句型和句子按照高级语言的语义翻译成中间代码。

（5）中间代码优化：遵循程序等价变换的原则，把中间代码加工变换成节省运行时间和存储空间的目标中间代码。

（6）目标代码生成：实现最后的转换，把中间代码转换成特定的机器语言。

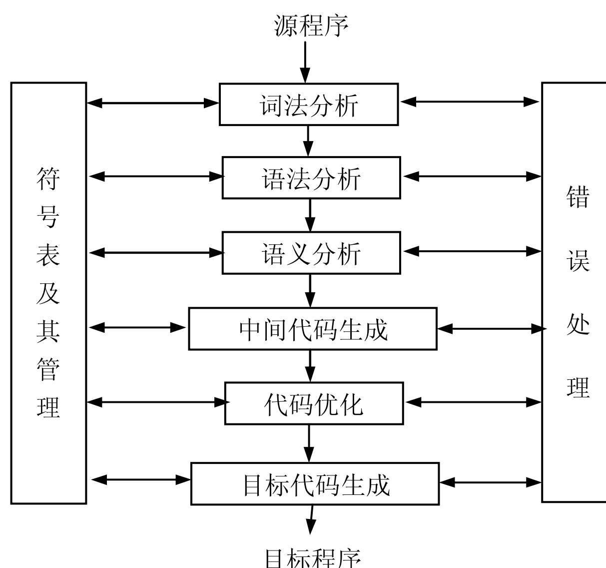


图 4.1. 编译程序的基本结构

在编译的过程中，编译器把源程序的各类信息和编译个阶段的中间信息保存在不同的符号表中，表格管理程序负责构造，查找和更新这些表格。错误处理程序主要功能是处理各个阶段中出现的错误，比如给出错误的性质和错误在源文件中的位置等。

编译器可以用机器语言或者汇编语言编写，也可以用高级语言编写，甚至还可以利用语言的词法和语法的形式化描述通过一些工具自动生成。

4.1.1.2. 与编译器相关的程序

本小节主要描述与编译器相关或者和编译器一起使用的其它程序。

（1）解释程序：解释程序（Interpreter）和编译器一样是一种语言翻译程序，它和编译器的不同之处在于，它直接执行源程序，或者叫做脚本语言程序。程序在解释执行时的速度一般比将它编译成目标代码后执行的要慢上许多，但是也有很多优点，比如说，简单、调试容易、可以方便地实现跨平台等。

（2）汇编程序：汇编程序（Assembler）是用于特定计算机上的汇编语言的翻译程序。

（3）连接程序：编译器和汇编程序都经常依赖于连接程序（Linker)。它把在不同的目标文件中编译或者汇编的代码收集到一个可直接执行的文件中。连接程序还连接目标程序和用于标准库函数的代码，级连接目标程序和由计算机的操作系统所提供的资源。

（4）装入程序：编译器、汇编程序或者连接程序生成的代码经常还不能直接执行：它们的主要存储器访问可以在存储器的任何位置，只是在逻辑上相互之间存在一个固定的关系，最终位置的确定和某个起始位置相关。通常称这样的代码是可重定位的。装入程序（Loader）可处理所有与指定的基地址或起始地址相关的可重定位的地址。装入程序的引入使得可执行代码的编译更加灵活。通常装入程序是作为操作系统的一部分在后台运行。

（5）预处理器：预处理器是在编译开始之前由编译器调用来删除注释、包含其它文件以及执行宏替代。

（6）调试器：调试程序用于对目标代码的调试，从而达到排除代码中存在的错误。

4.1.1.3. 编译器的移植

前面已经讲到，要想让计算机源程序运行就必须通过编译器把这个源程序编译生成相应计算机上的目标代码才能够运行，而编译的前提是必须要有一个机器语言组成的可以直接运行的编译器。第一个编译器只能使用机器语言编写，而现在就不需要这样了。如果想编写某种特定语言的编译器，可以直接利用已经有编译器的其它语言来编写，然后利用已有的编译器来编译，从而得到一种新的可以直接运行的编译器。

通常而言，一个编译器所编译生成的目标代码一般是这个编译器所在的机器上的机器语言编码。但是，也有例外的情况，这种情况常常在嵌入式系统开发中遇到，就是利用运行在某机器上的编译器编译某个源程序生成在另外一台机器上运行的目标代码，这种编译器即所谓的交叉编译器（Cross Compiler）。

4.1.2. 词法分析

词法分析的主要工作是把源程序读作字符串并将其分成若干个单词符号（token），单词符号和自然语言中的单词类似，每一个单词符号都表示源程序中信息单元的字符序列，因此可以把单词符号简单的叫做单词。

4.1.2.1. 词法的形式化描述

所有的程序设计语言都可看作是由数字、字母和其它一些有意义的符号构成的字符串组成的。具有特定意义的符号串构成单词。若干个单词按照某种语法规则构成句子。程序可以看作是句子的有序集合。

对于特定的程序设计语言，尽管构成这种语言的符号是有限的，但不同功能的程序中的单词和句子是不同的。单词有无穷多个，词法形式化描述的目的就是采用一定的方法来定义构成单词的规则。

某种程序设计语言中允许使用的全部符号的集合叫做它的字符集。字符集通常是某种标准字符集（如ASCII）的子集，用 Σ 来表示。为了词法分析的方便，需要把字符集中的字符分成几类，为了书写的方便，用“→”来表示“定义为”，用“|”表示“或者”。比如，常见的字符分类如下：

字母 → A|B|C|…|Z|…;
数字 → 0|1|2|…|9;
算术运算符 → +|-|*|/|…;
关系运算符 → >|<|=|!=|>=|…;
分界符 → ,|:|;|(|)|…;

单词是具有特定意义的符合某种规则的字符串，一般程序设计语言中的单词可以分成以下几类：

单词 → 关键字|标志符|常数|运算符|分界符|…

利用正则表达式可以描述单词的词法规则，这里仅仅利用正则表达式的推导规则与性质给出各类单词的正则表达式，不再介绍有关正则表达式的内容，这并不影响这一部分的理解。

（1）关键字

关键字是语言中规定的保留字，例如 C/C++语言中的“if”，“for”等。C/C++语言中的关键字可以表示为：

关键字 → if|else|while|do|for|switch|case|auto|register|static|int|long|char|sizeof|return|…

（2）标志符

多数程序设计语言的标志符都定义为以字母开头的字母数字串，因此标志符可以定义为：

标志符 → 字母（字母|数字）*

其中的（）*表示 0 个或者任意多个限定字符（这里的字符可以使字母或者数字）的任意集合。为了限定标志符的长度，可以引用一对花括号，用两个上下标表示最大和最小的重复数，比如：

标志符 → 字母{字母|数字}$^{32}_{0}$

（3）运算符

运算符的表示比较简单，可以表示为：

运算符 → 算术运算符|赋值运算符|关系运算符|位运算符|…

算术运算符 → +|-|*|/|%|…；

赋值运算符 → =|{算术运算符|关系运算符|位运算符}=

关系运算符 → >|<|==|!=|>=|<=

位运算符 → &|`|`|^|>>|<<|~

（4）分界符

分界符可以表示为：

分界符 → ,|:|;|(|)|…；

（5）常数

常数的种类比较多，各种语言对常数的定义并不完全一致，一般的定义如下：

(+|-)(d1 d2 d3…)(. r1 r2 r3…)E(+|-)(n1 n2 n3…)…

其中(d_1 d_2 d_3…)表示整数部分，(. r_1 r_2 r_3…)表示小数部分，(n_1 n_2 n_3…)表示十进制的阶码。常数可以描述如下：

无符号常数 → <整数部分><小数部分><指数部分>

整数部分 → 空|(数字)+

小数部分 → 空.|. (数字)+

指数部分 → 空|E(+|-|空) (数字)+

根据上述五部分的描述，经过扩充就可以得到一个比较完整的单词描述。

有了单词的正则表达式,如何根据该表达式编制各类单词的词法分析程序呢?词法分析程序是对组成单词的符号进行识别与处理的过程，为了便于进行词法分析程序的设计和刻画内部处理过程的细节，人们引入了状态转换图。下面举例说明状态转换图的概念。

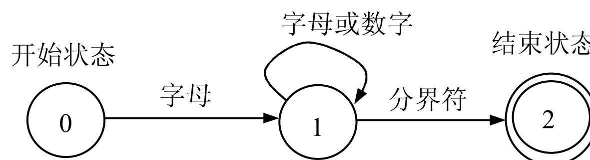


图 4.2. 标志符的状态转换图

设标志符的正则表达式为：

标志符 → 字母（字母|数字）*

可见，标志符的第一个字符必须是字母，后面是字母或者数字。当输入一个非字母数字的字符时，说明标志符结束，把整个输入过程分成几个状态，令已经准备好接收一个标志符的状态为开始状态，用 0 表示，接收到第一个字符的状态为 1 状态，从 0 状态到 1 状态画一条弧线，弧线上注明输入的字母，则状态转换图如 4-2 所示。

有了状态转换图，只需要为每一个状态设计一段处理程序，就很容易设计出标志符的分析程序（参考图 4.3.）。

由此可见，状态转换图可以反映单词识别的动态过程，它可以从单词的正则表达式推出，可以把各类单词的正则表达式用相应的状态转换图表示（图 4.4.）。

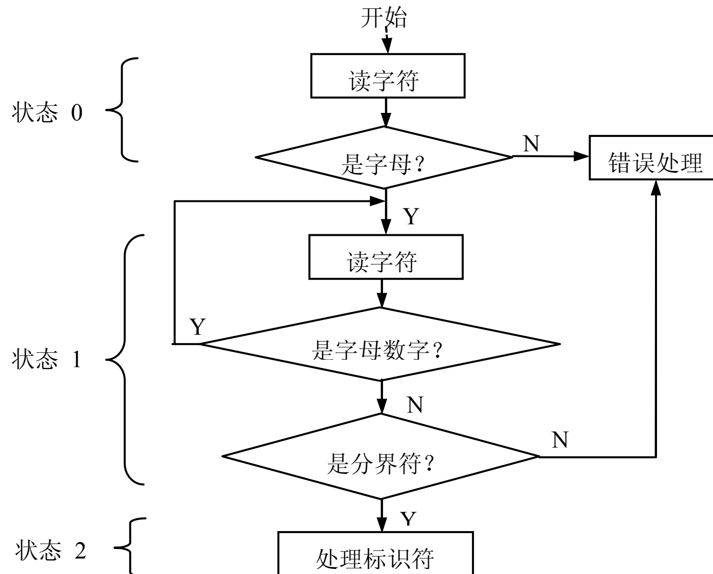


图 4.3. 标志符处理流程

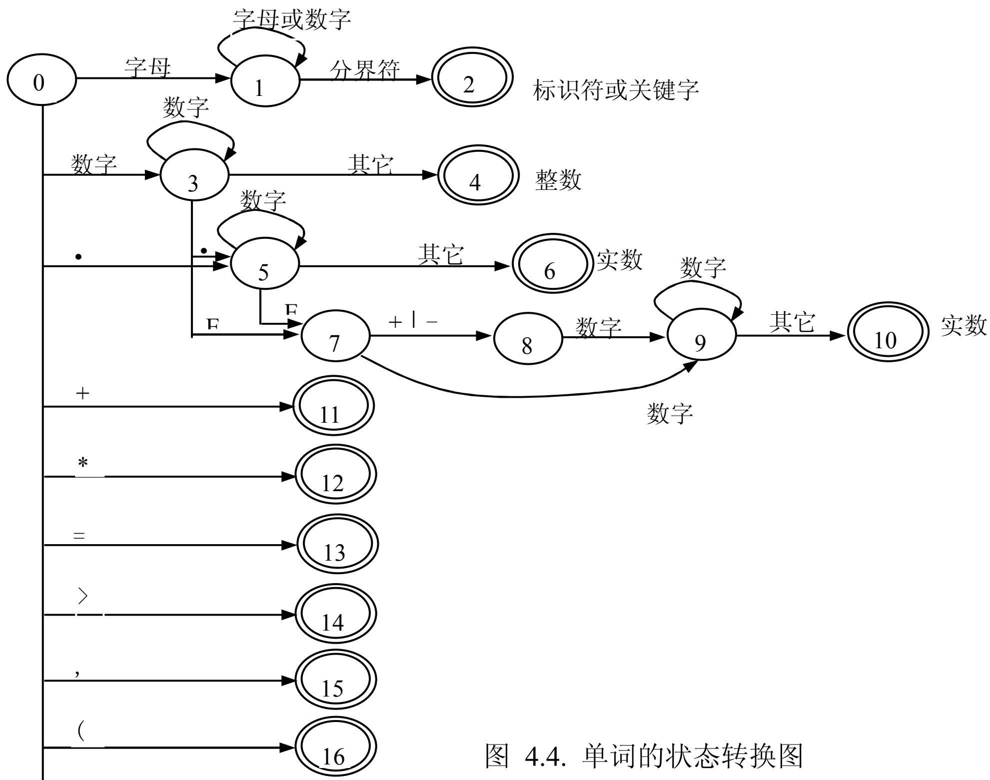


图 4.4. 单词的状态转换图

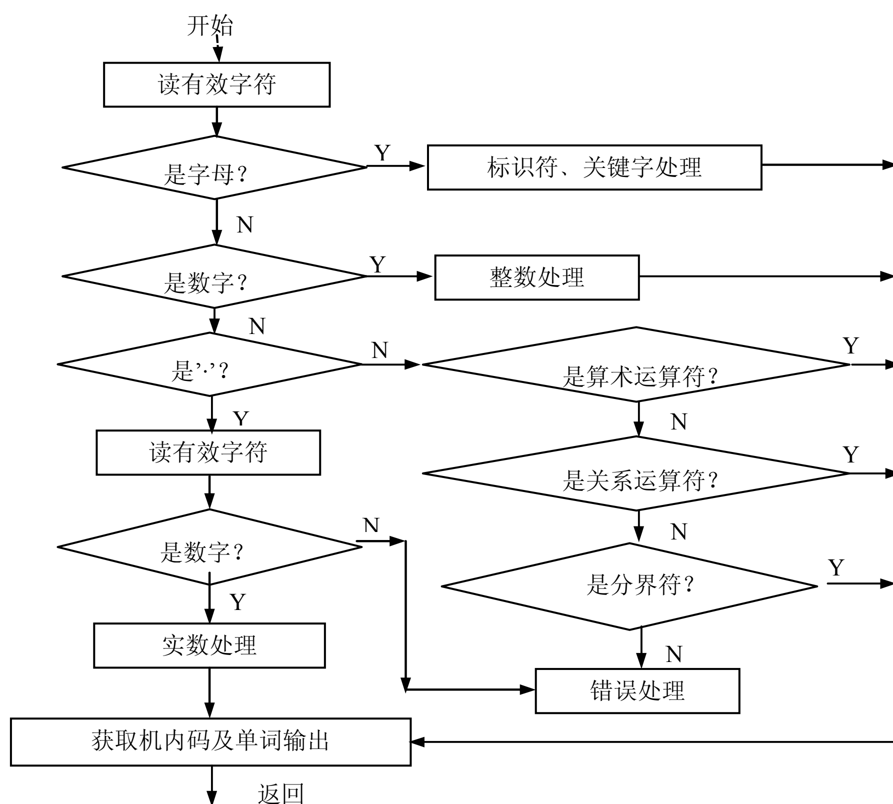


图 4.5. 词法分析流程

根据单词的正则表达式，得到词法分析的状态转换图之后，就可以为每一个状态设计一段处理程序即可得到词法分析程序。

4.1.2.2. 词法分析程序的设计

根据语言的词法描述和状态转换图，就可以实现词法分析程序了。词法分析程序的输入一般是源程序，这种源程序大多保存在文本文件中或者特定的缓冲区中。词法分析程序的输出应该包括两部分：单词信息和错误信息。

单词应该表示成一种统一的二元式：（单词种别码，单词自身值），其中单词种别码说明单词的类型，常用预定义好的一个整数类型来表示，单词自身值可以使单词本身，也可以是单词的地址。

错误信息一般采用格式化的形式来记录每种错误的位置、类型以及原因。词法分析中一般的错误有：非法字符，标志符不合法（比如以数字开头等）和标志符过长等错误。

词法分析的处理流程大致如图 4.5.所示。

对于不同的词法分析程序，处理流程基本上是相同或者相似的，因此词法分析程序可以利用已有的工具自动生成。自动生成词法分析程序的关键是如何实现从词法规则到处理程序的自动转换。如果能够把特定词法规则自动转换成状态转换图，并把状态转换图用矩阵的形式存储在内存中，供词法分析程序识别各类单词使用，就可以实现词法分析程序的自动生成。目前，已经有工具可以根据提供的词法，利用有限状态自动机，生成词法分析

程序了。比如，LEX 就是一般 UNIX 系统提供的一种用于根据词法规则自动生成相应词法分析的实用程序。有关 LEX 的使用方法可以参考网站 http://www.combo.org/lex_yacc_page 和 http://www.linux.org/docs/ldp/howto/Lex-YACC-HOWTO-1.html 。

4.1.3. 语法分析

任何一种语言都是由句子组成的，句子是由单词组成的。词法分析是识别构成语言的单词，语法分析便是识别构成语言的句子，确定某一个特定的句子是符合于某种语言的语法规则。对于自然语言，可以用有限的语法规则来描述无穷多的句子；类似的，对于程序设计语言，可以用有限的文法规则来描述无穷多的程序设计语句。

比如，句子“cats eat mice”是一个主谓宾结构的句子，用图形的形式表示如下：

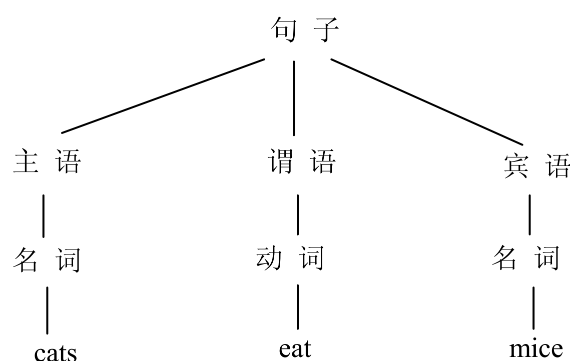


图 4.6. 句子的语法规则

上面的语法规则可以形式化的描述如下：

<句子> → <主语> <谓语> <宾语>
<主语> → <名词>
<名词> → cats
<谓语> → <动词>
<动词> → eat
<宾语> → <名词>
<名词> → mice

应用上述的规则可以推导句子如下：

<句子> → <主语> <谓语> <宾语>
→ <名词><谓语><宾语>
→ cats <谓语><宾语>
→ cats <动词><宾语>
→ cats eat <宾语>
→ cats eat <名词>
→ cats eat mice

利用这个规则，同样可以推出以下的句子：“mice eat cats”“cats eat cats”“mice eat mice”，这些句子的语法都是正确的，但是语义显然并不正确，因此语义和语法是不同的。

句型分析有两大类比较成熟的方法，自顶向下分析方法和自底向上的分析方法，因此语法分析也相应的可以分成两大类。自顶向下的语法分析方法主要有递归下降法、状态矩

阵法和 LL（1）分析法，自底向上的语法分析方法主要有算符优先法、优先数法和 LR 分析法。

4.1.3.1. 自顶向下的语法分析

自顶向下分析的基本原理是从语法定义的句型规则开始，推出句子。例如，在 C 语言中，整形变量定义的句型如下（S 是表示变量定义语句的符号）：

S → <TYPE> <IDS><SC>
<TYPE> → int
<IDS> → <IDS>,id
<IDS> → id
<SC> → ;

那么对于语句“ int i,j; ”，利用最左推导的原则，可以自顶向下的分析如下：

S → <TYPE> <IDS><SC>
→ int <IDS><SC>
→ int <IDS>, id <SC>
→ int id , id <SC>
→ int i , id <SC>
→ int i , j <SC>
→ int i , j ;

下面以一般表达式的分析过程为例，简要说明递归下降法的原理。表达式一般可以表示成如下的形式：

E → E + T ｜E －T ｜T
T → T * F ｜T / F｜F　　　　表达式描述（1）
F → (E)｜i

这里 E 表示一个加/减法表达式，T 表示一个乘/除法表达式，F 表示表达式的一个操作数，可以看出它既可以是一个括号表达式，也可以是一个常数或标志符。上面的表达可以明显的看出表达式的分析是一个递归的过程。把上面的表示形式等价的修改为（稍后将会说明为什么做这种修改）：

E → T｛ +T｜－T ｝
T → F｛ *F｜ / F ｝　　　　表达式描述（2）
F → (E)｜i

可以看出，只要对每一组算符（加减、乘除、括号）分别写一个处理程序，然后通过程序之间的嵌套调用，就可以实现上述表达式的分析过程。递归处理程序的基本结构如下：

```
AddMinusProc   //处理加、减的函数
 { PlusDivideProc;
      While ( word == `+' || word == ` - ' )
       {   Readword; //读取下一个单词
           PlusDivideProc;
           NewOP;     //根据 word 产生一条指令
       }
 }
PlusDivideProc   //处理乘、除的函数
```

```
{   ParenthesisProc；
    While ( word == `*' || word == ` / ' ) do
        {    Readword；
             ParenthesisProc；
             NewOP；
        }
  }
ParenthesisProc   //处理括号的函数
{   if ( word == `(' )
    {   Readword；
        AddMinusProc；
        If ( word == `)' )；
        else ErrorProc；//错误处理函数
}
else if ( word == `标识符'   )
          process  标识符；
    else ErrorProc；
Readword；
}
```

上述的处理程序中，在进入每一个分析处理子函数时，已经读出了一个单词 word，而在退出子函数时，又读出了下一个单词放在 word 中。

递归下降法的优点在于处理的子函数和文法的规则直接相对应，但是它对文法也有要求，它不允许文法中出现左递归和回溯，这就是改写表达式文法的原因。

在上面的表达式描述（1）中，递归和回溯是很明显的，比如对于 A*B+C 进行自顶向下的分析时：

① E → E +T
  → E+T+T
  → E+T+T+T+…

② E → E +T
  → E+F
  → E+i
  → E+A

分析①无法产生表达式 A*B+C，因为文法中含有左递归；分析②错误的使用了 T→F，同样导致无法产生这个表达式，因为存在多个首元素相同的规则，从而无法判断应该选择那条规则，必须读出多个单词才能判断，即存在回溯现象。

实际上，可以通过等价修改文法的定义来消除左递归和回溯，上面的表达式描述（2）就是表达式描述（1）消除后的结果。具体的消除方法这里不再说明。

L L（1）分析使用显式栈而不是递归调用来完成自顶向下的语法分析，栈和递归调用在实际效果上是等价的，这里不再详细介绍 LL（1）方法，感兴趣的读者可以参考有关书籍。

4.1.3.2. 自底向上的语法分析

自底向上的基本过程是：自左向右逐个扫描输入的单词串，一边将输入的单词移入分析栈，一边检查分析栈顶的一串单词是否与某条规则的右部相同，若相同，则把栈顶单词串替换成相应规则的左部，否则将单词移入栈中，并继续扫描输入的单词串，直到结束为止。若最终栈内的单词为某个句型的符号，那么输入的单词串就是正确的句子。

对于 4.1.3.1 中的语句采用自底向上的分析，过程如下：

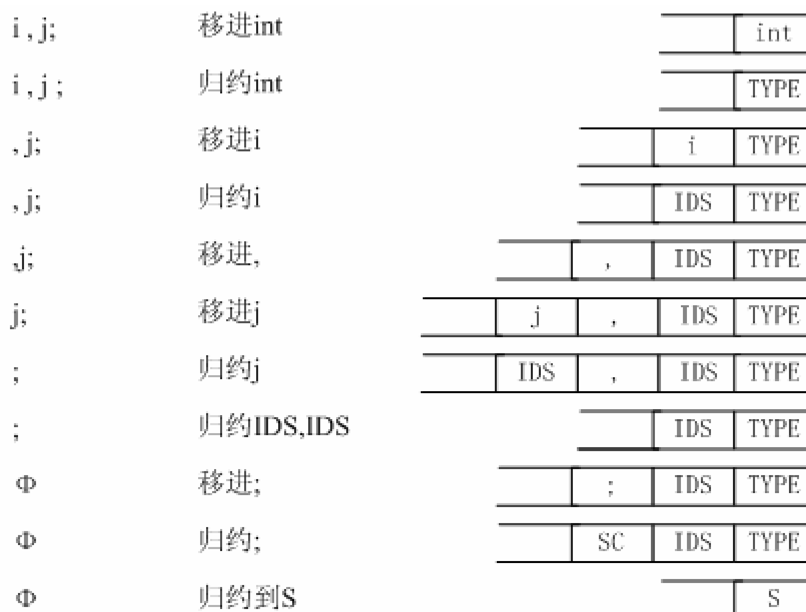


图 4.7. 自底向上分析的过程

自底向上的语法分析方法主要有算符优先法、优先数法和 LR 分析法。

算符优先法特别适用于表达式的分析，它的基础是各个算符之间的优先级。例如，通常的四则运算规则是先乘除，后加减，并且同级的运算从左结合，此时，算符乘“*”要优先于算符“+”、减“-”；而左边的算符“*”要优先于右边的算符“*”，同样，左边的算符“/”也优先于右边的算符“/”。

可以用“>” “<” “=”来表示算符的优先关系：a>b 表示左边算符 a 的优先级高于右边运算符 b，a<b 表示左边算符 a 的优先级低于右边运算符 b，a=b 表示左边算符 a 的优先级等于右边运算符 b。利用这种表示方法，可以生成一个描述各个算符之间的优先矩阵，通常的四则运算的优先矩阵如下（#是表达式结束标志符）：

根据这个优先矩阵，就可以设计两个先进后出的栈，即运算符栈和运算数栈，开始分析时，运算符栈预先压入一个表达式结束标志符“#”，运算数栈是空的。算符优先法的分析步骤如下，假设 a 为存放新读入符号的单元：

①从输入串中读入一个符号存入 a；

②如果 a 是运算数，把它压入运算数栈，然后转到步骤①；

③如果 a 是运算符，那么与当前运算符栈顶θ的元素比较优先级。如果 a>θ，那么把 a 压入运算符栈，转到步骤①；如果 a<θ，那么弹出θ和相应个数的运算数，生成θ的运算指令，并把生成的运算结果压入运算数栈，然后转到步骤③；如果 a=θ，那么从运算符

栈中弹出θ并废弃 a，或者结束分析过程；如果 a 和θ没有优先关系，说明表达式出现错误，应该做错误处理。

	+	-	*	/	（	）	i	#
+	＞	＞	＜	＜	＜	＞	＜	＞
-	＞	＞	＜	＜	＜	＞	＜	＞
*	＞	＞	＞	＞	＜	＞	＜	＞
/	＞	＞	＞	＞	＜	＞	＜	＞
（	＜	＜	＜	＜	＜	＝	＜	×
）	＞	＞	＞	＞	×	＞	×	×
i	＞	＞	＞	＞	×	＞	×	×
#	＜	＜	＜	＜	＜			＝

图 4.8. 四则运算的优先矩阵

这个分析过程可以利用流程图表示，如图 4.9.所示。

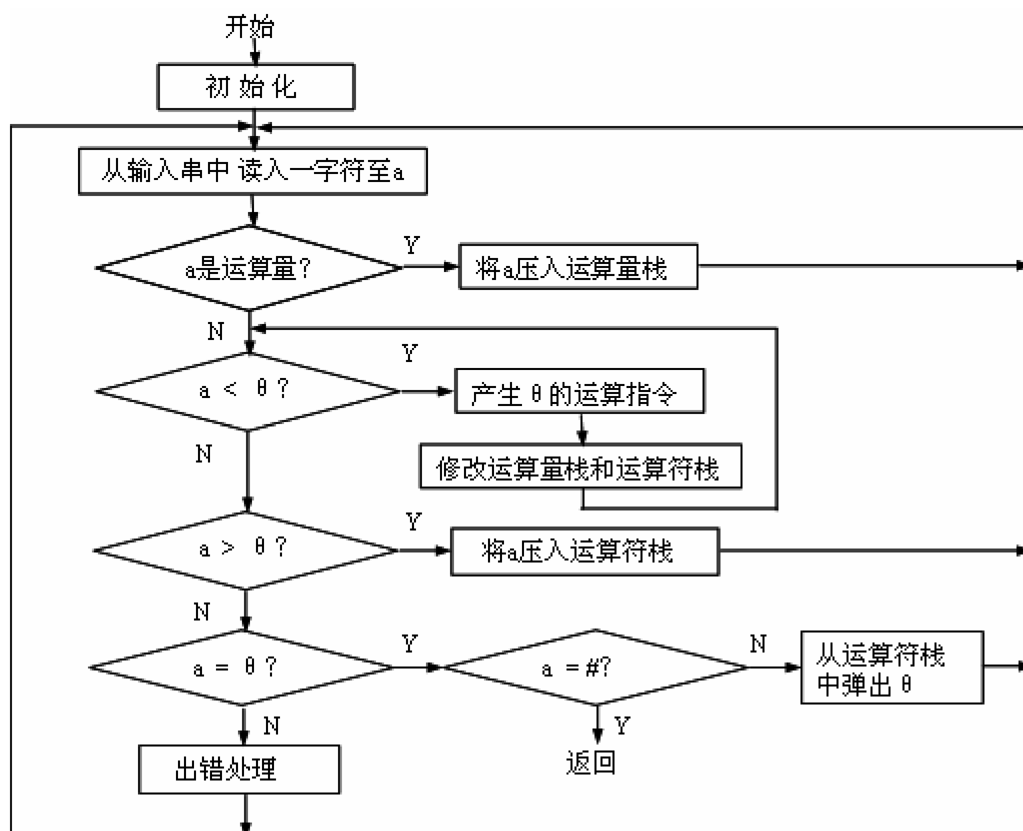


图 4.9. 算符优先法的分析流程

算符优先法简单明了而且容易实现，因此常常被用来分析和处理各种表达式，但是这种分析仅仅依靠算符的优先关系来处理，可能会把一些不合语法规则的句子当成合法句子

来分析，比如“(AB)-CD+/”不合语法，但是也能产生正确的结果。在实际的应用中，一般还要在分析之前先进行语法检查。

同时可以看出，应用算符优先法，当终结符较多时，优先矩阵将占据较大的内存空间，同时查优先矩阵表的时间长，为此，在算符优先法的基础上，可以采用优先数法进行语法分析。

优先数法使用两个优先函数 f，g 代替优先矩阵进行算符优先法的语法分析，其中 f 表示栈内优先函数，g 表示栈外优先函数，f、g 的定义如下：若 $a<b$，则 $f(a)<g(b)$；若 $a>b$，则 $f(a)>g(b)$；若 $a=b$，则 $f(a)=g(b)$，其中 a、b 为任意两个运算符。

优先数法不需要存储优先矩阵，因此节省了大量的存储空间。不过，它要求必须预先构造出两个优先函数，目前已经有比较成熟的利用优先矩阵来构造优先函数的方法。

LR 分析方法也是一种自底向上的语法分析方法，LR 分析器由输入缓冲器、分析栈、分析表和工作程序组成。它的基本思想是：工作程序自左向右从缓冲器中逐个读出单词进行分析，分析栈中存入一系列的单词和状态 $s_0x_1\ s_1x2s_2\cdots x_ms_m$.，在每种状态和输入单词下，从分析表查找相关的操作并执行。

LR 分析方法一般用于语法分析程序的自动生成。对于不同的语言，LR 分析方法的基本流程是相同的，只是分析表不同。YACC 可以根据特定的语法规则和分析表生成相应的语法分析程序，有关 YACC 的文档可以参考 4.1.2.2 中给出的网站。

4.1.4. 中间代码

对于编译程序来说，中间代码是介于源程序和机器语言之间的一种抽象语言。中间代码可以采用很多形式，几乎有多少种编译器就有多少种中间代码形式。生成中间代码的主要目的是为了代码优化和方便移植。编译程序常用的中间代码有逆波兰式、三元式和四元式，其中四元式的应用尤为广泛。

逆波兰式是由波兰科学家卢卡西维奇创造的一种表示方法。逆波兰式把运算数写在前边，运算符写在后边，即后缀式（平常的写法是中缀式，运算符在运算数的中间）。例如 A+B 需要写成 AB+。这种表示方法不需要括号，它可以根据运算数和运算符的前后位置来确定运算的次序。比如（A+B）*（C+D）可以表示成 AB+CD+*。

逆波兰式的计算过程需要一个栈，计算时从左至右的扫描表达式，遇到运算数就压入栈，遇到运算符就弹出相应个数的运算数进行运算，并把结果压入栈中，得到的最终结果就是表达式的结果。

三元式由运算符（OP）、运算数 1（ARG1）、运算数 2（ARG2）三部分组成。例如，语句 D=(A+B)*(A-C)可以表示成：

序号	OP	ARG1	ARG2
（1）	+	A	B
（2）	-	A	C
（3）	*	（1）	（2）
（4）	=	D	（3）

可以看出，三元式需要引用其它三元式的结果，因此一般还需要维护一张间接码表，用于记录中间结果，当三元式的编号改变时，就必须改变相应的编号。四元式利用临时变量避免了这种情况。

四元式是一种普遍采用的中间代码形式，四元式包括四部分：运算符（OP）、运算数1（ARG1）、运算数2（ARG2）和运算结果（RES）。例如赋值语句D=(A+B)*(A-C)可以表示成：

序号	OP	ARG1	ARG2	RES
（1）	+	A	B	T1
（2）	-	A	C	T2
（3）	*	T1	T2	T3
（4）	=	T3	#	D

表中，T1、T2、T3是中间临时变量，#表示没有这一项为空。

各种中间代码的表示方法中，操作符一般都和机器指令相类似。

程序设计语言中常见语句有算数表达式、布尔表达式、赋值语句、条件语句、循环语句等，下面对这些语句的四元式表达略作说明。在下面的说明中，利用（OP，ARG1，ARG2，RES）的形式来表示一个四元式。

- **算术表达式和赋值表达式**

一般的算术表达式和赋值表达式的四元式表达形式比较简单，可以参考前面的例子。这里对表达式中隐式的类型转换作一下说明。

不同类型的变量是不同的，比如整型变量一般是占用4个字节的存储空间，其中存放着变量的二进制数据，如果整型变量是无符号型的数据，它的有效位数就是32位，能表示的最大数值就是$2^{32}-1$，如果整型变量是带符号型的数据，它的最高位用来表示正负，因此有效位是31位。相应的，浮点数也有自己的表示方法。

在一般的语言中，一般允许在同一个表达式中出现不同的数据类型，编译器能够自动的实现不同类型之间的变换，比如，在C语言中，下面的语句是非常常见的：

```
int i=5;
int k;
float pi=3.2;
k=i*pi;
```

最后的赋值语句存在两处类型转换，它首先把i转化成浮点型数据执行浮点数的乘法运算，然后把运算的结果再转换成整型数据赋给k，如果用i2f来表示从整型到浮点型的变换操作，用f2i来表示从浮点型到整型的变换操作，这个赋值语句的中间代码可以表示如下：

（i2f, i, #, T1）
（*, T1, pi, T2）
（f2i, T2, #, T3）
（=, T3, #, k）

其它种类的类型转换可以类似的写出相应的中间代码。

- **布尔表达式**

布尔表达式一般作为控制语句的条件式，它用在if…else、while、for等语句中。布尔表达式有两种产生中间代码的方法，一种是按照算术表达式的方法产生，另一种是把布尔表达式翻译成条件语句。

例如布尔表达式 A && (B || C)，如果按照算术表达式的方法，产生的中间代码如下：

（||, B, C, T1）
（&&, A, T1, T2）

如果按照条件表达式产生，那么应该解释成如下的结构（类 C 描述）：

```
if (A)
   if (B)return true;
   else return C;
else
   return false;
```

这种翻译的好处是可以减小在实际运行过程中的条件判断次数，目前的编译器一般都采用这种处理方法。

- **条件、循环语句**

条件和循环语句中出现了程序的跳转，因此中间代码中必须同时记录四元式的地址以便于跳转。

例如：

```
if（i<5）k=i+5;
else if(i<10) k=i;
else k=i-5;
```

这个条件语句的中间代码可能（这和编译器的实现有关）如下（A 开头的标志符表示地址）：

```
地址    中间代码
A1     （<, i, 5, A6）
A2     （<, i, 10, A9）
A3     （-, i, 5, T1）
A4     （=, T1, #, k）
A5     （jmp, A10, #, #）
A6     （+, i, 5, T2）
A7     （=, T2, #, k）
A8     （jmp, A10, #, #）
A9     （=, i, #, k）
A10    （…）
```

又如，循环语句 while(sum<100) sum=sum+sum/2;的中间代码可能如下：

```
地址    中间代码
A1      （/, sum, 2, T1）
A2      （+, sum, T1, T2）
A3      （=, T2, #, sum）
A4      （<, sum, 100, A1）
```

4.1.5. 代码优化

所谓优化指的是为提高由编译程序生成出来的目标程序的质量而进行的各项工作。这里所说的质量是指目标程序所占的存贮空间(即程序的静态长度)的大小和运行目标程序所需的时间(即程序动态长度)的多少。优化既要设法缩短存贮空间，又要设法提高运行速度，但重点是提高运行速度。

一般说来，一个不考虑优化的编译程序，其生成出来的目标程序的质量比较差。这是因为这类编译程序是依据源语言的语法规则去统一处理各种源程序酌，因而较多的考虑了它们的共性而较少考虑个别的个性。手工编写程序正好相反，它是针对具体问题来编制程

序的，因此可以密切结合问题的特殊性而给出特殊的处理。正因如此，所以编译程序生成出来的目标程序的长度比手编程序的长度要长好几倍，运行时间甚至高达十几倍。因此，优化是十分必要的。

优化工作涉及的面非常广。从优化与机器的关系出发，可分为与机器无关的优化和与机器相关的优化。与机器无关的优化主要包括合并常量运算、消去公共子表达式、外提循环中的不变表达式、强度削减等，与机器有关的优化主要包括、寄存器的优化、多处理机的优化、特殊指令的优化、无用代码的消除等。

另外，从优化与源程序的关系出发，又可把优化分为局部优化和全局优化。局部优化指的是在只有一个人口、一个出口的线性程序块上的优化，它不存在转进转出分叉汇合的问题，处理起来比较简尽，花费的代价比较小。全局优化指的是在非线性程序块上的优化。因为程序块是非线性的，所以需要分析程序块的控制流程，甚至是整个源程序的控制流程，需要考虑全部信息，处理比较复杂，花费的代价也比较大，但功效经常比较好。

下面介绍几种常见的优化方法。为了便于阅读，这里采用的是类似于 C 语言语法和四元式的例子，实际上一般的优化都在中间代码中进行。对于较复杂的程序结构，比如循环，省略了中间代码的表示。

（1）合并已知量

编译优化的一条原则是能在编译时算出的量尽量在编译时算出，这样就节省了目标运行的时间。合并已知量的运算就是出于这样的目的。

例如：源语句：c＝2*3.14*r; 的四元式表达形式为：

```
（*,2,3.14, T1）
（*, T1, r, T2）
（=, T2, #, c）
```

在优化时，第一个四元式的常量运算会直接算出，成为下面列出的形式。

```
(*, 6.28, r, T1)
(=, T1, #, c)
```

（2）消除多余运算

问题的自然表示常常包含有多余运算，例如求解：

```
c1=2*PI*r1;
c2=2*PI*r2;
（*, 2, PI, T1）
（*, T1, r1,T2）
（=, T2, #, c1）
（*, 2, PI, T3）
（*, T3, r2,T4）
（=, T4, #, c2）
```

对于程序员来说，上面的写法是非常自然的，但是它重复了计算过程 2*PI，在程序优化时就可以提取运算相似的部分，利用临时变量，执行一次运算即可：

```
_2PI=2*PI;
c1=_2PI *r1;
c2=_2PI*r2;

（*, 2, PI, T1）
（*, T1, r1,T2）
（=, T2, #, c1）
```

```
(*, T1, r2,T3)
(=, T3, #, c2)
```

(3)循环中不变运算的外提

和循环无关的运算称为循环中不变运算，把它提到循环过程外一次执行，可以节省相当多的运算步骤。例如：

```
for(C=0,i=0;i<100;i++)
{
  C=2*PI*a[i]+C;
}
```

由于运算过程 2*PI 对循环始终不变，优化时可以外提：

```
_2PI=2*PI;
for(C=0,i=0;i<100;i++)
{
  C=_2PI*a[i]+C;
}
```

这样就节省了 99 次的乘法运算过程，对于多层嵌套、次数非常多的循环过程，这种优化可以大大的节省运算时间。

(4)循环中运算强度的削减

指令运算加法比乘法快的多。一般来说，乘法的运算强度大于加法。所谓强度削减是指用指令强度低的运算代替指令强度高到运算。实际上一般主要考虑与循环控制变量有关的运算。例如：

```
for(i=0;i<100;i++)
{
  k=5*i+12;
  printf("%d\n",k);
}
```

可以优化为：

```
for(k=7,i=0;i<100;i++)
{
  k=k+5;
  printf("%d\n",k);
}
```

表达式和循环过程的优化是优化中非常重要的两部分，随着编译技术的提高，优化的范围越来越大，效果也越来越好。在最后，给出一个优化的例子，以供读者思考。

在 C/C++和类似的语言中，++i 和 i++这两个语句是不同的，前者的++是前缀运算符，表示在使用 i 值之前先加 1，后者的++是后缀运算符，表示在使用 i 值之后再加 1，因此，sum=3*(i++)和 sum=3*(++i)的得到的结果是不同的。在 for 循环语句中，最常见的循环过程就是类似于 for(i=0;i<N;i++)的形式，由于++运算存在两种形式，对于这种循环形式来说，第三个表达式选择++i 或者 i++的形式都没有区别，但是如果考虑优化的话，两者就是不同的。参考++i 和 i++的中间代码就可以看出这两条语句的差别：

考虑 sum=i++和 sum=++i，它们的中间代码如下（INC 表示自增运算）：

```
sum=++i:(INC, i, #, i)
        (=, i, #, sum)
sum=i++:(+, i, 1, T1)
```

（=, T1, #, sum）

可以看出，i++生成了一个临时对象，这就需要更多的存储空间，对于 for 循环来说，++i 应该是更好的写法。关于两个运算的区别请参考有关 C/C++语言的书籍。

4.2. 交叉编译技术

4.2.1. 交叉编译

随着硬件平台和操作系统的多样化，软件向不同平台移植的工作变得越来越繁复。交叉编译技术的引入为软件的不同平台移植创造了便利条件。在交叉编译技术中有两种比较典型的实现，一个我们称之为 Java 模式，即 Java 的字节码编译技术；另外一个我们称之为 GNU GCC 模式，即通常所讲的 Cross GCC 技术。

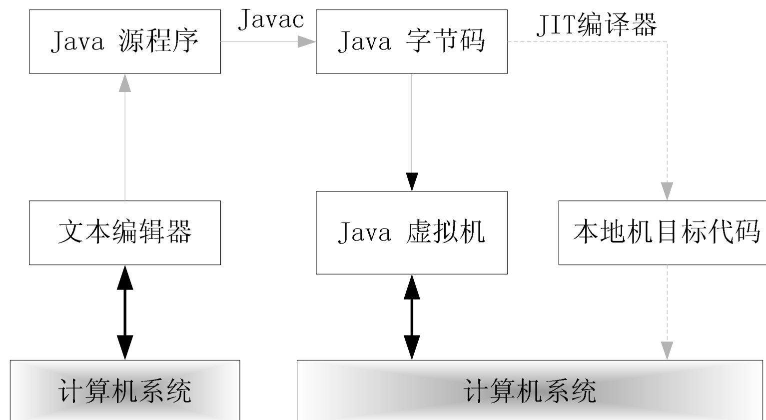


图 4.10. Java 模式

Java 模式（图 4.10.）的最大特点是引入了一个自定义的虚拟机（Java Virtual Machine），即 Java 虚拟机（JVM）。所有 Java 源程序都会首先被编译成在只有在这个虚拟机上才能执行的“目标代码”——字节码（Bytecode）。在实时运行时，可以有两种运行方式，一种是编译所获得的字节码由 JVM 在实际计算机系统上执行；另外一种方式是通过 Java 实时编译器（Just-In-Time Compiler）将字节码首先转换成本地机可直接执行的目标代码，而后交给实际的计算机系统运行，这实际上是一个两次编译过程，一次是非实时的、一次是实时的。由于在这里，第一次非实时编译时，Java 编译器生成的是基于 JVM 的“目标代码”，我们可以将它的编译技术也称为交叉编译。

GCC 模式（图 4.11.）通过 Cross GCC 直接生成目标平台的目标代码，从而能够直接在目标平台上运行。这里的关键是 Cross GCC 的生成和选择问题。我们需要根据目标平台的不同，选择针对这个平台的 Cross GCC。

GCC 模式和 Java 模式的最大不同在于 GCC 直接生成目标平台的目标代码，而 Java 模式首先只是生成字节码，只有在有 JIT 编译器的参与下才会进一步生成目标平台的目标代码。研究表明，Java 模式虽然可以通过两个编译过程生成目标代码，但是因为两次编译的优化存在相互冲突，最终的目标代码的执行效率也不是很高。而 GCC 模式由于直接能够生成目标代码，其执行效率一般很高。

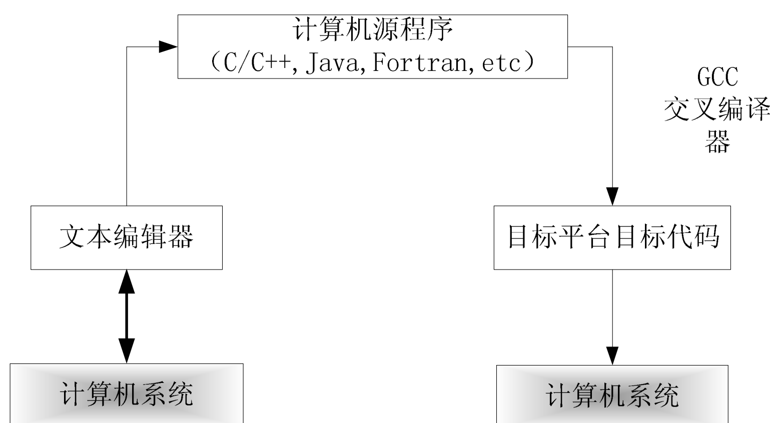


图 4.11. GCC 模式

值得一提的是在 GCC 中有 Java 语言编译器，它不仅能够生成和 Java 源程序对应的字节码，而且也能够直接生成目标平台的目标代码，这就为 Java 语言编制的程序的执行效率的提高提供了新的编译手段。只是，到目前为止，GCC 对 Java 语言类库的支持仍然很不完全，这主要表现在对其图形用户界面类库的支持上。

4.2.2. GCC 交叉编译器

GCC 的主要目的是为 32 位 GNU 系统提供一个好的编译器。GCC 所有源代码是和硬件平台无关的，当然有一些必要的硬件参数是需要给出的。

4.2.2.1. GCC 编译流程

在讲述 GCC 对给出的计算机源程序进行编译之前，我们需要明确一点的就是 GCC 为了实现与具体计算机硬件平台无关的编译过程，它使用了 RTL 语言用于对目标平台的指令进行一般描述。在 RTL 语言中，指令被表述成与之一一对应的描述性语言，而寄存器也被相应的伪寄存器代替。下面让我们来看看 GCC 的一般编译的详细流程：

- **源码解析**

这一阶段主要是对计算机源程序进行词法分析、语法分析以及语义分析，并最终生成一个对应于源程序的带有语义注释的语法树。其中语法树的表示并不完全遵循 C 语言语法。

- **语法树的优化**

这个阶段的优化是对语法树本身的优化。主要的优化就是基于树的内嵌。比如说，有一个函数 A，它的函数体是调用 B，而 B 的函数体是调用 C，表现在语法树上则是一个三级子树的结构，其优化结果就是三级子树最后有可能只收缩为一级，在这一级中，是函数 A 的声明，而函数体则内嵌的是函数 C 的函数体。当然也有例外的情况：如果函数 C 被另外的函数调用的话，这里的内嵌就不是那么容易。

- **RTL 代码生成**

在这一阶段语法树被使用寄存器传递语言（Register Transfer Language）编写的代码代替。在这一阶段开始涉及到和目标平台某些参数相关的代码。优化内容包括一些服务于 if

语句的表达式，其中包括布尔操作、比较和条件表达式。在这一阶段，编译器会检测和确定递归的深度，并对循环语句和 switch 语句的优化作出安排。另外，在 RTL 代码生成阶段最后会决定有些函数调用是否需要直接用对应的函数体替换，即所谓的函数内嵌（Function Inlining）。

- **函数调用优化**

这一阶段参考上一步的结果，对某些递归函数调用和某些函数直接调用实施函数内嵌操作。

- **转移指令优化**

在这一阶段，编译器会简化转移指令。它会销掉在程序运行过程中不会被执行到的代码。它也会将某些转移指令直接用转移后执行的操作系列代替，以避免多执行一步转移指令，从而提高程序执行效率。

- **寄存器扫描**

在这一阶段，编译器纪录每个寄存器第一次的使用和最后一次的使用，从而为公用子表达式的消除作指导。

- **转移指令线程化**

这个优化阶段是可选择的，对应的编译选项是-fthread-jumps。

- **SSA 优化**

所谓 SSA（Static Single Assignment）是指，在 SSA 形式下，每个变量只能进行一次性赋值，这与我们在 C/C++或者 Java 中见到的静态变量赋值类似，这种方法可以做很多优化，比如说，在多层嵌套的条件语句中实现条件常值的传播，并且其计算复杂度是线性的。这一步也是可选择的，选项是-fssa。

- **公用子表达式削减**

这一阶段也是可选择的，就看你对最终可执行程序的代码长度的要求了。这一步也做条件常值的传播。如果条件常值的传播过程导致条件转移，则需要重新执行转移指令优化。

- **循环语句优化**

在这次优化中，编译器会把循环语句中值不随循环次数变化的表达式从循环语句中提取出来，以避免这样的语句每次循环时的重复计算。当然有些优化是可选择的，比如说考虑把循环语句打开以平直方式而非循环方式执行。

- **二次公用子表达式优化**

如果-frerun-cse-after-loop 编译选项被选定，需要在循环语句优化之后再次进行公用子表达式优化。

- **数据流分析**

在这一步中，程序被分成若干个基本块，然后确定在程序的每一步中究竟是哪一个伪寄存器是出于工作状态。在数据流分析的过程中，会销掉程序中不可能执行到的循环；不可能用到的数据的计算；合并某些内存访问指令的简单的加/减操作，取而代之的是自动增一/减一操作。

- **指令合并**

这一步，编译器试图合并一些在数据流上相联系的一组指令，使得这样的两三个指令被一个指令代替。从而能够简化指令操作，提高程序运行效率。

- **寄存器移动**

在这一步中，编译器首先确定某些匹配限制需要一个指令重新载入，而这种重载还是一个从一个寄存器到另外一个寄存器的情况。当确定有这种情况时，编译器会对相关指令的寄存器使用方式做出适当修正，从而使得在这种情况下的指令重载不需要寄存器间移动这一操作。

- **指令时序调整**

在这一阶段中，编译器会查找某些指令的输出值还没有得到，而这个值却要被后续指令使用的情况，在 RISC 机器上，内存装载和浮点指令常常会出现这种情况。这个时候，编译器会重新调整相关指令的顺序，从而避免运行过程中不必要的管道延迟。

指令时序调整会被执行两次，第一次是在指令合并之后立即执行，第二次是在重载之后。

- **寄存器类优先级选择**

在这一步，编译器扫描 RTL 代码以找出那一类寄存器更适合所给出的伪寄存器。

- **局部寄存器分配**

在这一步中，编译器会将为在程序的每个基本块处于工作状态的伪寄存器指定一个实际的寄存器。由于这些基本块在程序是线性分布的，因此，局部寄存器分配操作可以很快。

- **全局寄存器分配**

在这一步中，编译器会为那些影响力超过程序的一个基本块范围的伪寄存器分配实际的寄存器。

- **重载**

这一步将会记住寄存器的分配情况。而那些没有得到寄存器分配的伪寄存器会集合起来形成一个堆栈。然后编译器就会寻找那些无效指令，之所说是无效的，是因为它所涉及到的某个值最终不是在寄存器中，或者是在一个类型不匹配的寄存器中。这个时候，编译器会通过这样一种方法来修正：它会加入一些指令，这些指令完成将出问题的值暂时重载入寄存器中。

- **二次指令时序调整**

这次指令时序调整是为了避免由于内存装载可能带来的管道延迟。

- **基本块重新排序**

这一步的目标还是为了提高程序执行的效率。它会根据某种性能描述或者评价信息来对基本块的执行顺序进行调整。如果类似的信息不存在的话，它会考虑利用各种各样的静态分析来对某些性能评价作出预测，从而指导对基本块的排序。这样的信息包括，某些指令的执行频度，程序执行分支的可能性等等。

- **二次转移指令优化**

这次包括交叉转移的情况，同时也会清除没有任何意义的转移指令，比如说，转移后不进行任何操作的指令。

- **可延迟性分支的时序安排**

在这一步中，某些可延迟执行的分支操作将会被找出，这通常是转移指令或者函数调用等。

- **多步分支指令合并**

在很多 RISC 机器中，分支指令通常会在执行时序上相邻两指令的地址转移量有个最大值限制。因此，对于某些相距较远的分支，可能需要多步分支指令转移才能实现。在这一步中，编译器会确定各分支指令间的地址转移量，参照其最大值限制，考虑合并某些不必要的分支指令，从而使得原有的多步分支指令序列能够用一个指令数较少的指令序列代替。

- **寄存器使用优化**

在程序的寄存器分配方面，一般的原则是只在程序的单个基本块起作用的伪寄存器将会被分配实际的寄存器，而对于那些没有得到分配的伪寄存器将会被集合起来用一个实际的堆栈来替代。在这一步中，某些本来已经分配了实际寄存器的伪寄存器可能会被收回给它的分配，而强迫它进入实际的堆栈。

- **输出与源程序对应的汇编语言程序**

在这一步，实际程序的汇编语言代码生成。在这里，某些和机器特征相关的优化工作将会被完成。

- **调试信息输出**

在这一步，编译器会输出没有得到实际寄存器分配的伪寄存器的相关信息。

4.2.2.2. Linux 环境下的 GCC 交叉编译器

从上面 GCC 的一般编译流程就可以知道，GCC 最终输出的是汇编语言源程序。想要进一步编译成我们所需要的机器代码，需要引入一些新的工具，比如，汇编程序等。Binutils 工具集提供了一些这样的工具，事实上，GCC 和 Binuitls 是经常在一起捆绑使用的。另外，对 C 语言而言，需要有相应的函数库支持 C 语言源程序的编译。而在 Linux 中应用最多的就是 GLibc 了。此外，在 Linux 环境下，生成相应的交叉编译器还需要 Linux 内核头文件的支持。

- **GNU Binutils**

GNU Binutils 的主要工具有两个，一个是连接程序 ld，另外一个是汇编程序 as。除此之外，它还包括一些工具，比如说：用于将地址转换成文件名和行序号的 addr2line；用于显示 ELF 格式目标文件的相关信息等等。Binutils 所支持的平台种类很多，不仅包括很多种 Unix 平台，甚至于还包括 Wintel 系统。其主要目的是为 GNU 系统，包括 GNU/Linux 系统提供汇编和连接工具。

- **GNU GCC**

GNU GCC 就是上面提到的 GCC，由于它和 Binutils、Glibc 同样都是 GNU 维护的，因此冠名 GNU。GCC 主要是为 GNU 系统提供 C 编译器，这是它的最初目标。当然现在开始支持多种语言，这其中包括 C/C++、Fortran、Java、Objective-C、甚至还有 Ada。

- **GNU GLibc**

任何一个 Unix 体系的操作系统都需要一个 C 库，用于定义系统调用和其它一些基本的函数调用，比如说 open、malloc、printf、exit 等等。

而 GNU Glibc 就是要提供这样一个用于 GNU 系统，特别是 GNU/Linux 系统的 C 库。Glibc 最初设计就是可移植的。尽管说源码体系非常复杂，但是我们仍然可以通过简单的 configure/make 来生成对应平台的 C 函数库。

- **GCC 交叉编译器生成**

1） 一般流程

第一步，取得 Binutils、GCC、Glibc 的源码。这里，你可能需要一些附件用于加密解密或者本地化等功用，这些也都需要获得其源码。

第二步，配置并编译 Binutils 取得我们所需要的汇编和连接程序。

第三步，配置并编译 GCC 源码生成 GCC 编译器。一般是 C 编译器首先生成，然后以这个为基础在结合下一步要生成的 Glibc 的 C 函数库，再编译生成其它编译器。

第四步，配置 Glibc 并编译生成 Glibc 的 C 函数库。

第五步，再次配置和编译 GCC 源码，生成其它语言的编译器，如 C++编译器等。

下面我们给出一个交叉编译器的生成实例，它是面向 ARM 平台的 Linux 开发环境。

2） 一个交叉编译器的生成实例

在过去，困扰嵌入式 Linux 开发者的一个很头疼的问题就是如何获取针对特定平台的

交叉编译工具包。由于不同平台的特异性，要获得针对不同平台的交叉编译工具包，需要知道所用平台的硬件体系结构，汇编语言规范，某些硬件参数的规定等等。随着 Linux 在嵌入式系统中的应用越来越广泛，针对一些主流硬件平台（主要指微处理器），如 ARM，Motorola 等的嵌入式 Linux 的交叉编译工具包生成技术越来越成熟，并开始得到比较系统的维护，可以比较方便的获取。

（1）可执行文件格式

现在有很多可执行文件格式，并且对某种特定文件格式的支持也相差很大。Linux（还有其它很多操作系统）都将 ELF 格式作为其标准格式。早期的 Linux 版本使用 a.out 格式，而现在都不大用了，除非特定的应用必需这种支持。

在选择交叉编译工具包时，需要注意应用程序所需的可执行文件格式。下面给出一些和 ELF 格式相关的可执行文件格式：

表 4.1 执行文件格式

执行文件格式	描述
strongarm-elf	对应的比较典型的交叉编译工具包 SA1 是由 Cygnus 在 1999 年九月提供。这个工具包用于生成针对 StrongARM 1 处理器的 ELF 执行文件。
strongarm2-elf	对应的比较典型的交叉编译工具包 SA2 也是由 Cygnus 提供，用于生成针对 StrongARM 2 处理器的 ELF 执行文件。
arm-elf	对应比较典型的交叉编译工具包是 SA2 和开放源码的 SA1，如果是后者，则需要 binutils, gcc, newlib，可以用来生成针对 ARM v4 Little Endian 系统的 ELF 可执行文件。
arm-linux	对应的交叉编译工具包最好是由开放源码生成，当然也可以由Cygnus公司的SA1和SA2来生成，不过需要打一些补丁。 注意：已经是可执行代码的交叉编译工具可以从下面这个FTP站点获取， ftp://ftp.netwinder.org/users/c/chagas/arm-linux-cross 它在Redhat Linux系统可以正常运行。

关于 ELF 格式的一个存在的问题就是对于类似上面所列的不同的 ELF 格式需要有相应不同的工具包。这就是说，对大多数在 Linux 下的工作而言，你应当使用“arm-linux”格式的工具包。

（2）交叉编译器

交叉编译器允许开发者在一台主机上编译生成某一个目标系统的二进制代码。要得到一个交叉编译工具包是一个很困难的过程。你可以从下面这个 FTP 站点获取所需要的资源：
ftp://ftp.netwinder.org/users/c/chagas/arm-linux-cross
该文件为 Linux_on_Assabet.tar.gz.
解压之后，主要有以下压缩文件：

binutils-2.10.tar.gz，
gcc-2.95.2.tar.gz，　gcc-2.95.2-diff-991022，　gcc-2.95.2-diff-991022-inhibit_libc，
glibc-2.1.3.tar.gz，　glibc-crypt-2.1.tar.gz，　glibc-linuxthread-2.1.3.tar.gz，
linux-2.4.0.tar.gz，　patch-2.4.0-rmk1，　diff-2.4.0-rmk1-np1。

要产生你所需要的工具包，可以运行下面的脚本：

```
#脚本开始---------------------
# 循序在Linux控制台下运行下面的命令
# 如果你想改变下面的交叉编译开放源码总目录的话，可以将下面所示目录
# /opt/xcompiler 修改成你所希望的目录名
XCOMPILER_SOURCE_DIR=/opt/xcompiler

# 如果你想改变交叉编译器安装目录的话，可以将下面所示目录/usr/local/arm-linux 修
# 改成你所希望的目录名
ARM_INSTALL_DIR=/usr/local/arm-linux
PATH=$ARM_INSTALL_DIR/bin:$PATH
mkdir -p $XCOMPILER_SOURCE_DIR
mkdir -p $ARM_INSTALL_DIR

# 将所下载的开放源码软件包解压，得到三个压缩软件binutils-<version>.tar.gz,
# gcc-<version>.tar.gz, glibc-<version>.tar.gz， 并将这些压缩文件放到
# $XCOMPILER_SOURCE_DIR\sources目录下。
# 其中<version>是必选项，指明相应开放源码模块的版本号。比如，对于binutils的2.10
# 版本，解压后的源码目录为$XCOMPILER_SOURCE_DIR\binutils-2.10
cd $XCOMPILER_SOURCE_DIR
tar -xzvf Linux_on_Assabet.tar.gz

# 将所需开放源码解压到上面所说的交叉编译开放源码目录，这些源码包括binutils，gcc，
# glibc等三个模块，对应的目录分别是$XCOMPILER_SOURCE_DIR\binutils-<version>,
# $XCOMPILER_SOURCE_DIR\gcc-<version>,
# $XCOMPILER_SOURCE_DIR\glibc-<version>。
# 并打上相应的补丁
cd $XCOMPILER_SOURCE_DIR
mkdir -p src build
cd src
tar -xzvf ../sources/binutils-2.10.tar.gz
tar -xzvf ../sources/gcc-2.95.2.tar.gz
tar -xzvf ../sources/glibc-2.1.3.tar.gz
cd glibc-<version>
tar -xzvf ../sources/glibc-crypt-2.1.tar.gz
tar -xzvf ../sources/glibc-linuxthread-2.1.3.tar.gz
tar -xzvf ../sources/linux-2.4.0.tar.gz
cd ..
gzip -dc ../sources/gcc-2.95.2-diff-991022.gz |
            patch -d ./gcc-2.95.2 -p0

# 注意：这里会给出两次显示错误信息："Hunk #1 FAILED at 1." 可以不用管。
gzip -dc sources/gcc-2.95.2-diff-991022-inhibit_libc.gz | patch -d
```

```
          gcc-2.95.2 -p0
gzip -dc sources/patch-2.4.0.gz | patch -p1 -d src/linux

mkdir -p $ARM_INSTALL_DIR/arm-linux/include
cd $XCOMPILER_SOURCE_DIR/src/linux
make assabet_config
make menuconfig
# 连摁两次ESC键，然后摁ENTER键
make dep

#执行下面这两个命令是把Linux内核中GCC所必需的头文件拷贝到GCC可以找到的目录中
cp -dR include/asm-arm $ARM_INSTALL_DIR/arm-linux/include/asm
cp -dR include/linux $ARM_INSTALL_DIR/arm-linux/include/linux

cd $XCOMPILER_SOURCE_DIR/build
mkdir -p binutils gcc glibc
cd $XCOMPILER_SOURCE_DIR/build/binutils

#配置Binutils，注意其目标平台为我们范例中所需要的arm-linux
../../src/binutils-2.10/configure                  --target=arm-linux
--prefix=$ARM_INSTALL_DIR

#使用本地编译器编译binutils并安装
make
make install

cd $XCOMPILER_SOURCE_DIR/build/gcc

#配置GCC，注意这里指定了所需要的头文件目录为linux内核的头文件目录
../../src/gcc-2.95.2/configure                     --target=arm-linux
--prefix=$ARM_INSTALL_DIR                         --disable-languages
--with-headers=$XCOMPILER_SOURCE_DIR/src/linux/include

#编译并安装GCC
make -i
make -i install

cd $XCOMPILER_SOURCE_DIR/build/glibc

#配置GLibc，指定所用的编译器是刚刚编译生成的交叉编译器arm-linux-gcc，并注意需要
#指定所需头文件所在的目录

CC=arm-linux-gcc       ../../src/glibc-2.1.3/configure       arm-linux
--enable-add-ons                  --prefix=$ARM_INSTALL_DIR/arm-linux
```

```
--with-headers=$XCOMPILER_SOURCE_DIR/src/linux/include
make
make install

#重新配置并编译GCC，注意这次编译生成的编译器是C++编译器
cd $XCOMPILER_SOURCE_DIR/build/gcc
gzip -dc ../../sources/gcc-2.95.2-diff-991022-inhibit_libc.gz | patch -R
-d ../../src/gcc-2.95.2 -p0
../../src/gcc-2.95.2/configure                          --target=arm-linux
--prefix=$ARM_INSTALL_DIR --with-cpu=strongarm --enable-languages=c++
--with-headers=../../src/linux/include
make -i
make -i install

#脚本结束---------------------
```

（3）相关问题

在 ARM-Linux 中我们可能会遇到几个问题。

首先遇到的问题是一个标准问题。由于交叉编译器的生成需要生成四个模块，但是由于这四个模块由不同的 GNU 团体来维护，这就存在一个标准问题，比如说对可执行文件格式的命名，binutils 和 gcc 所采用的某些命名标准是不一样的，这就导致同一个对象有不同的指称，对交叉编译造成了一些不必要的障碍。而这是现在 GNU 软件存在的一个比较普遍的问题。如何能够实现各类命名的统一是和实现 API 的统一同等重要的问题。

另外一个问题就是被 Linux 所使用的标准 C 库是 glibc。这个标准库可以编译成带版本号或不带版本号两种。究竟这两种形式有什么不同还不清楚，但是一个程序用前者编译生成的可执行文件在后者所组成的动态连接库环境中就不能正常运行，反之亦然。

还有一个和 ARM-Linux 动态库关系比较大的问题和“重定址”有关。这是有关如何加载动态连接库的问题。可以参考 ftp.netwinder.org/users/p/patb/public_html/elf_relocs.html

第五章 嵌入式 Linux 的软件开发环境

5.1. 交叉编译环境

5.1.1. 编译开发环境的建立

在嵌入式系统的开发中，特别是针对非 X86 平台的系统，交叉编译环境的建立是一个非常重要的环节。交叉编译环境为在 PC 机上开发、编译和调试针对目标平台的代码提供了很大方便，有效的缩短了开发周期，降低了二次开发的难度。

基于 uCLinux 的应用开发环境一般是由目标系统硬件开发板和宿主 PC 机所构成。硬件开发板用于操作系统和应用软件的运行，而操作系统内核的编译、应用软件的开发和调试则需要借助宿主 PC 机来完成。双方之间一般通过串口建立连接关系。

在软件开发环境建立方面，由于 uCLinux 及相关工具集都是开放源码的项目，所以大多数软件都可以从网络上下载获得。首先要在宿主机上安装标准 Linux 发行版，比如 Red-Hat Linux，接下来就可以建立交叉开发环境。

5.1.1.1 安装交叉编译环境

- 编译工具的安装

针对 uCLinux 目前有两套编译工具：m68k-coff 和 m68k-elf，它们都是 GNU 组织开发的优秀的编译器 GCC 的不同应用版本。它们的区别在于形成最终 flat 目标码之前的中间代码格式分别是 coff 和 elf 类型。elf 格式的编译器比 coff 格式的编译器有许多优越性(见后)，建议使用 m68k-elf 交叉编译器。目前不断有新版本的 elf 编译器发布，安装也越来越方便，通常只需要在宿主机 Linux 系统的根目录下解开一个压缩文件就可以自动安装整套编译工具包。编译工具包中除了交叉编译器以外，还有链接器（ld）、汇编器（as）以及一些为了方便开发的二进制处理工具，包括生成静态库工具（ar、ranlib）、二进制码察看工具（nm、size）、二进制格式转换工具（objcopy）。这些都要安装在宿主机上。

- **uCLinux** 内核的安装

利用已安装的交叉编译器编译生成运行于目标机上的 uCLinux 内核。与标准 Linux 相同的是，uCLinux 内核可以以配置的方式选择需要安装的模块，而增加系统的灵活性。

- 应用程序库的安装

用交叉编译器编译 uC-libc 和 uC-libm 源码，生成 libc.a 应用程序库和 libm.a 数学库。

- 其他工具的安装

用 GCC 编译 elf2flt 源码，生成格式转换工具 elf2flt。用 GCC 编译 genromfs 源码，得到生成 romfs 工具 genromfs。

5.1.1.2. 添加设备驱动和内核模块

经过以上的准备工作之后，下面要针对特定应用所需要的设备编写或改造设备驱动程序。有一些设备驱动，uCLinux 本身就已经具有。即便没有，基于 uCLinux 开放源码的特

性，用户也可以很方便地把自己的驱动程序加入内核。如果用户对系统实时性，特别是硬实时有特殊的要求，uCLinux 可以加入 RT-Linux 的实时模块。完成这些工作，一个嵌入式应用开发平台就已经搭建好了，在此之上，根据不同需要可以开发不同的嵌入式应用。

5.1.2. 可执行文件

像对文件系统一样，Linux 支持的二进制格式或是在内核建造时内置进内核的，或者可以作为模块被装入。内核保持一个所支持的二进制格式列表，当试图执行一个文件时，依次试用每一种二进制格式直到有一种成功。通常 Linux 支持的二进制格式是 a.out 和 ELF。可执行文件不必完全装入内存，而是使用了一种称为请求调页的技术。随着每一部分可执行映像被进程使用，它被读入内存。未被使用的部分映像可以从内存中淘汰。uCLinux 系统使用 flat 可执行文件格式，目前也支持 elf 文件格式。下面解释这几种可执行文件格式。

5.1.2.1. Coff 文件格式

Common object file format，一种通用的对象文件格式；这是最早与 uClinux 整合的 m68k 编译器，比较成熟。它的主要特征有：

- 应用程序可以在其生成的目标文件中添加独立于系统的信息，同时不影响已有工具对该目标文件的访问。
- 该文件格式为调试器等应用预留了空间加入所需的符号信息。
- 程序员可以通过编译选项改变目标文件的生成方式。

5.1.2.2. elf 文件格式

Executable and Linkable Format，可执行和可链接格式。一种为 Linux 系统所采用的通用文件格式，支持动态连接和重定位；由 UNIX 系统实验室设计的这种目标文件格式已牢固树立了 Linux 中最常使用的格式的地位。尽管和其他目标文件格式如 COFF 相比有轻微性能开销，但 ELF 被认为更灵活。ELF 可执行文件包含可执行代码，有时被称为正文(text)，以及数据(data)。静态链接映像被链接器(ld)或链接编辑器建造成一个单一的映像，包含运行此映像所需的所有代码和数据。映像还指明此映像在内存中的布局，以及此映像中第一条执行的代码的地址。

5.1.2.3. flat 文件格式

uClinux 目前支持 BFLT（binary flat）格式的可执行文件，elf 格式的文件都可以通过转换器 elf2flt 转化为 BFLT 格式。在编译的时候，如果检查到参数中有－elf2flt 选项，elf2flt 就会被执行，从而生成 BFLT 格式的文件。

elf 格式有很大的文件头，flat 文件对文件头和一些段信息做了简化，可执行程序小。当用户执行一个应用时，内核的执行文件加载器将对 flat 文件进行进一步处理，主要是对 reloc 段进行修正（详见 fs/binfmt_flat.c）。

需要 reloc 段的根本原因是，程序在连接时连接器所假定的程序运行空间与实际程序载入的内存空间不同。假如有这样一条指令：

```
jsr app_start;
```

这一条指令采用直接寻址，跳转到 app_start 地址处执行，连接程序将在编译完成时计算出 app_start 的实际地址（假设实际地址为 0x10000），这个实际地址是根据 ld 文件计算出来。但实际上由于内存分配的关系，操作系统在加载时无法保证程序按 ld 文件加载。这时如果程序仍然跳转到绝对地址 0x10000 处执行，通常情况只能得到错误的执行结果。

一个解决办法是增加一个存储空间，用于存储 app_start 的实际地址，假设使用变量 addr 表示这个存储空间。则以上这句程序将改为：

```
movl addr, a0;
jsr (a0);
```

增加的变量 addr 将在数据段中占用一个 4 字节的空间，连接器将 app_start 的绝对地址存储到该变量。在可执行文件加载时，可执行文件加载器根据程序将要加载的内存空间计算出 app_start 在内存中的实际位置，写入 addr 变量。系统在实际处理时不需要知道这个变量的确切存储位置（也不可能知道），系统只要对整个 reloc 段进行处理就可以了（reloc 段有标识，系统可以读出来）。处理很简单只需要对 reloc 段中存储的值统一加上一个偏置（如果加载的空间比预想的要靠前，实际上是减去一个偏移量）。偏置由实际的物理地址起始值同 ld 文件指定的地址起始值相减计算出。

5.2. 调试技术

有很多方法可以用来监控运行着的用户空间程序：可以为其运行调试器并单步调试该程序，添加打印语句，或者添加工具来分析程序。下面将介绍几种可以用来调试 Linux 上运行的程序的方法。顺带介绍四种调试问题的情况，这些问题包括段错误，内存溢出和泄漏，还有挂起。

5.2.1. 常见调试方法

当您的程序中包含错误时，很可能在代码中某处有一个条件，您认为它为真（true），但实际上是假（false）。找出错误的过程也就是在找出错误后推翻以前一直确信为真的某个条件过程。

以下几个示例是您可能确信成立的条件的一些类型：

(1) 在源代码中的某处，某变量有特定的值。
(2) 在给定的地方，某个结构已被正确设置。
(3) 对于给定的 if-then-else 语句，if 部分就是被执行的路径。
(4) 当子例程被调用时，该例程正确地接收到了它的参数。

找出错误也就是要确定上述所有情况是否存在。如果您确信在子例程被调用时某变量应该有特定的值，那么就检查一下情况是否如此。如果您相信 if 结构会被执行，那么也检查一下情况是否如此。通常，您的假设都会是正确的，但最终您会找到与假设不符的情况。结果，您就会找出发生错误的地方。

调试是您无法逃避的任务。进行调试有很多种方法，比如将消息打印到屏幕上、使用调试器，或只是考虑程序执行的情况并仔细地揣摩问题所在。

在修正问题之前，您必须找出它的源头。举例来说，对于段错误，您需要了解段错误发生在代码的哪一行。一旦您发现了代码中出错的行，请确定该方法中变量的值、方法被调用的方式以及关于错误如何发生的详细情况。使用调试器将使找出所有这些信息变得很简单。如果没有调试器可用，您还可以使用其它的工具。

5.2.2. 内存调试

C 语言作为 Linux 系统上标准的编程语言给予了我们对动态内存分配很大的控制权。然而，这种自由可能会导致严重的内存管理问题，而这些问题可能导致程序崩溃或随时间的推移导致性能降级。

内存泄漏（即 malloc()内存在对应的 free() 调用执行后永不被释放）和缓冲区溢出（例如对以前分配到某数组的内存进行写操作）是一些常见的问题，它们可能很难检测到。这一部分将讨论几个调试工具，它们极大地简化了检测和找出内存问题的过程。

- MEMWATCH

MEMWATCH 由 Johan Lindh 编写，是一个开放源代码 C 语言内存错误检测工具，您可以自己下载它（http://www.linkdata.se/sourcecode.html）。只要在代码中添加一个头文件并在 gcc 语句中定义了 MEMWATCH 之后，您就可以跟踪程序中的内存泄漏和错误了。MEMWATCH 支持 ANSI C，它提供结果日志纪录，能检测双重释放（double-free）、错误释放（erroneous free）、没有释放的内存（unfreed memory）、溢出和下溢等等。

清单 5.1. test1.c

```
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
   char *ptr1;
   char *ptr2;

   ptr1 = malloc(512);
   ptr2 = malloc(512);

   ptr2 = ptr1;
   free(ptr2);
   free(ptr1);
}
```

清单 5.1.中的代码将分配两个 512 字节的内存块，然后指向第一个内存块的指针被设定为指向第二个内存块。结果，第二个内存块的地址丢失，从而产生了内存泄漏。

现在我们编译示例程序 5.1. 的 memwatch.c。下面是一个 makefile 示例：

gcc -DMEMWATCH -DMEMWATCH_STDIO test1.c memwatch.c -o test1

当您运行 test1 程序后，它会生成一个关于泄漏的内存的报告。 清单 5.2.展示了示例

memwatch.log 输出文件。

清单 5.2. test1 memwatch.log 文件

```
MEMWATCH 2.67 Copyright © 1992-1999 Johan Lindh

...
double-free: <4> test1.c(15), 0x80517b4 was freed from test1.c(14)
...
unfreed: <2> test1.c(11), 512 bytes at 0x80519e4
{FE FE FE FE FE FE FE FE FE FE FE FE ..............}
Memory usage statistics (global):
N)umber of allocations made:     2
   L)argest memory usage :    1024
   T)otal of all alloc() calls:    1024
   U)nfreed bytes totals :     512
```

MEMWATCH 为您显示真正导致问题的行。如果您释放一个已经释放过的指针，它会告诉您。对于没有释放的内存也一样。日志结尾部分显示统计信息，包括泄漏了多少内存，使用了多少内存，以及总共分配了多少内存。

- **YAMD**

YAMD 软件包由 Nate Eldredge 编写，可以查找 C 和 C++ 中动态的、与内存分配有关的问题。在撰写本文时，YAMD 的最新版本为 0.32。请下载 yamd-0.32.tar.gz（http://www3.hmc.edu/~neldredge/yamd）。执行 make 命令来构建程序；然后执行 make install 命令安装程序并设置工具。

一旦您下载了 YAMD 之后，在 test1.c 上使用它。删除 #include memwatch.h 并对 makefile 进行如下小小的修改：

```
gcc -g test1.c -o test1
```

下面运行：

```
./run-yamd test1
```

得到下面的输出如清单 5.3.

清单 5.3. test1 YAMD 输出

```
YAMD version 0.32
Executable: test1
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal deallocation of this block
Address 0x40025e00, size 512
```

```
...
ERROR: Multiple freeing At
free of pointer already freed
Address 0x40025e00, size 512
...
WARNING: Memory leak
Address 0x40028e00, size 512
WARNING: Total memory leaks:
1 unfreed allocations totaling 512 bytes

*** Finished at Tue ... 10:07:15 2002
Allocated a grand total of 1024 bytes 2 allocations
Average of 512 bytes per allocation
Max bytes allocated at one time: 1024
24 K alloced internally / 12 K mapped now / 8 K max
Virtual program size is 1416 K
End.
```

YAMD 显示我们已经释放了内存，而且存在内存泄漏。让我们在清单 5.4.中另一个样本程序上试试 YAMD。

清单 5.4. test2.c

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *ptr1;
   char *ptr2;
   char *chptr;
   int i = 1;
   ptr1 = malloc(512);
   ptr2 = malloc(512);
   chptr = (char *)malloc(512);
   for (i; i <= 512; i++) {
      chptr[i] = 'S';
   }
   ptr2 = ptr1;
   free(ptr2);
   free(ptr1);
   free(chptr);
}
```

用和运行 test1 同样的方式运行 test2：

./run-yamd test2

相应的输出如清单 5.5.

清单 5.5. test2 YAMD 输出

```
Running test2
Temp output to /tmp/yamd-out.1243
*********
./run-yamd: line 101: 1248 Segmentation fault (core dumped)
YAMD version 0.32
Starting run: /usr/src/test/test2/test2
Executable: /usr/src/test/test2/test2
Virtual program size is 1380 K
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal allocation of this block
Address 0x4002be00, size 512
ERROR: Crash
...
Tried to write address 0x4002c000
Seems to be part of this block:
Address 0x4002be00, size 512
...
Address in question is at offset 512 (out of bounds)
Will dump core after checking heap.
Done.
```

MEMWATCH 和 YAMD 都是很有用的调试工具，它们的使用方法有所不同。对于 MEMWATCH，您需要添加包含文件 memwatch.h 并打开两个编译时间标记。对于链接（link）语句，YAMD 只需要 -g 选项。

5.2.2. 系统调用跟踪

strace 命令是一种强大的工具，它能够显示所有由用户空间程序发出的系统调用。strace 显示这些调用的参数并返回符号形式的值。strace 从内核接收信息，而且不需要以任何特殊的方式来构建内核。将跟踪信息发送到应用程序及内核开发者都很有用。在清单 5.6. 中，分区的一种格式有错误，清单显示了 strace 的开头部分，内容是关于调出创建文件系统操作（mkfs）的。strace 确定哪个调用导致问题出现。

清单 5.6. mkfs 上 strace 的开头部分

```
execve("/sbin/mkfs.jfs", ["mkfs.jfs", "-f", "/dev/test1"], &
 ...
 open("/dev/test1", O_RDWR|O_LARGEFILE) = 4
 stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
 ioctl(4, 0x40041271, 0xbfffe128) = -1 EINVAL (Invalid argument)
 write(2, "mkfs.jfs: warning - cannot setb" ..., 98mkfs.jfs: warning -
 cannot set blocksize on block device /dev/test1: Invalid argument )
  = 98
 stat64("/dev/test1", {st_mode=&, st_rdev=makedev(63, 255), ...}) = 0
 open("/dev/test1", O_RDONLY|O_LARGEFILE) = 5
 ioctl(5, 0x80041272, 0xbfffe124) = -1 EINVAL (Invalid argument)
 write(2, "mkfs.jfs: can\'t determine device"..., ..._exit(1)
  = ?
```

5.2.3. 程序调试

- **gdb**

您可以从命令行使用 gdb 程序（Free Software Foundation 的调试器）来找出错误，也可以从诸如 Data Display Debugger（DDD）这样的几个图形工具之一使用 gdb 程序来找出错误。您可以使用 gdb 来调试用户空间程序或 Linux 内核。这一部分只讨论从命令行运行 gdb 的情况。

使用 gdb program name 命令启动 gdb。gdb 将载入可执行程序符号并显示输入提示符，让您可以开始使用调试器。您可以通过三种方式用 gdb 查看进程：

(1) 使用 attach 命令开始查看一个已经运行的进程；attach 将停止进程。

(2) 使用 run 命令执行程序并从头开始调试程序。

(3) 查看已有的核心文件来确定进程终止时的状态。要查看核心文件，用下面的命令启动 gdb。

gdb programname corefilename

要用核心文件进行调试，您不仅需要程序的可执行文件和源文件，还需要核心文件本身。要用核心文件启动 gdb，请使用 -c 选项：

gdb -c core programname

gdb 显示哪行代码导致程序发生核心转储。

在运行程序或连接到已经运行的程序之前，请列出您觉得有错误的源代码，设置断点，然后开始调试程序。您可以使用 `help` 命令查看全面的 gdb 在线帮助和详细的教程。

- **kgdb**

kgdb 程序（使用 gdb 的远程主机 Linux 内核调试器）提供了一种使用 gdb 调试 Linux 内核的机制。kgdb 程序是内核的扩展，它让您能够在远程主机上运行 gdb 时连接到运行用 kgdb 扩展的内核机器。您可以接着深入到内核中、设置断点、检查数据并进行其它操作（类似于您在应用程序上使用 gdb 的方式）。这个补丁的主要特点之一就是运行 gdb 的主机在引导过程中连接到目标机器（运行要被调试的内核）。这让您能够尽早开始

调试。请注意，补丁为 Linux 内核添加了功能，所以 gdb 可以用来调试 Linux 内核。

使用 kgdb 需要两台机器：一台是开发机器，另一台是测试机器。一条串行线（空调制解调器电缆）将通过机器的串口连接它们。您希望调试的内核在测试机器上运行；gdb 在开发机器上运行。gdb 使用串行线与您要调试的内核通信。

请遵循下面的步骤来设置 kgdb 调试环境：

(1) 下载您的 Linux 内核版本适用的补丁。

(2) 将组件构建到内核，因为这是使用 kgdb 最简单的方法。（请注意，有两种方法可以构建多数内核组件，比如作为模块或直接构建到内核中。举例来说，日志纪录文件系统（Journaled File System，JFS）可以作为模块构建，或直接构建到内核中。通过使用 gdb 补丁，我们就可以将 JFS 直接构建到内核中。）

(3) 应用内核补丁并重新构建内核。

(4) 创建一个名为 .gdbinit 的文件，并将其保存在内核源文件子目录中（换句话说就是 /usr/src/linux）。文件 .gdbinit 中有下面四行代码：

```
set remotebaud 115200
symbol-file vmlinux
target remote /dev/ttyS0
set output-radix 16
```

(5) 将 append=gdb 这一行添加到 lilo，lilo 是用来在引导内核时选择使用哪个内核的引导载入程序。

```
image=/boot/bzImage-2.4.17
label=gdb2417
read-only
root=/dev/sda8
append="gdb gdbttyS=1 gdb-baud=115200 nmi_watchdog=0"
```

清单 5.7 是一个脚本示例，它将您在开发机器上构建的内核和模块引入测试机器。您需要修改下面几项：

best@sfb：用户标识和机器名。

/usr/src/linux-2.4.17：内核源代码树的目录。

bzImage-2.4.17：测试机器上将引导的内核名。

rcp 和 rsync：必须允许它在构建内核的机器上运行。

清单 5.7. 引入测试机器的内核和模块的脚本

```
set -x
rcp best@sfb: /usr/src/linux-2.4.17/arch/i386/boot/bzImage /boot/bzImage-2.4.17
rcp best@sfb:/usr/src/linux-2.4.17/System.map /boot/System.map-2.4.17
rm -rf /lib/modules/2.4.17
rsync -a best@sfb:/lib/modules/2.4.17 /lib/modules
chown -R root /lib/modules/2.4.17
lilo
```

现在我们可以通过改为使用内核源代码树开始的目录来启动开发机器上的 gdb 程序了。在本示例中，内核源代码树位于 /usr/src/linux-2.4.17。输入 gdb 启动程序。

如果一切正常，测试机器将在启动过程中停止。输入 gdb 命令 cont 以继续启动过程。

一个常见的问题是，空调制解调器电缆可能会被连接到错误的串口。如果 gdb 不启动，将端口改为第二个串口，这会使 gdb 启动。

下面我们来看一个实例来了解一下 kgdb 调试内核的情况。

清单 5.8 列出了 jfs_mount.c 文件的源代码中被修改过的代码，我们在代码中创建了一个空指针异常，从而使代码在第 109 行产生段错误。

清单 5.8. 修改过后的 jfs_mount.c 代码

```
int jfs_mount(struct super_block *sb)
{
...
int ptr;                    /* line 1 added */
jFYI(1, ("\nMount JFS\n"));
/ *
* read/validate superblock
* (initialize mount inode from the superblock)
* /
if ((rc = chkSuper(sb))) {
            goto errout20;
      }
108         ptr=0;                   /* line 2 added */
109         printk("%d\n",*ptr);       /* line 3 added */
```

清单 5.9. 在向文件系统发出 mount 命令之后显示一个 gdb 异常。kgdb 提供了几条命令，如显示数据结构和变量值以及显示系统中的所有任务处于什么状态、它们驻留在何处、它们在哪些地方使用了 CPU 等等。清单 5.9. 将显示回溯跟踪为该问题提供的信息；where 命令用来执行反跟踪，它将告诉被执行的调用在代码中的什么地方停止。

清单 5.9. gdb 异常和反跟踪

```
mount -t jfs /dev/sdb /jfs

Program received signal SIGSEGV, Segmentation fault.
jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
109               printk("%d\n",*ptr);
(gdb)where
#0 jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
#1 0xc01a0dbb in jfs_read_super ... at super.c:280
#2 0xc0149ff5 in get_sb_bdev ... at super.c:620
#3 0xc014a89f in do_kern_mount ... at super.c:849
#4 0xc0160e66 in do_add_mount ... at namespace.c:569
#5 0xc01610f4 in do_mount ... at namespace.c:683
#6 0xc01611ea in sys_mount ... at namespace.c:716
#7 0xc01074a7 in system_call () at af_packet.c:1891
```

```
#8 0x0 in ?? ()
(gdb)
```

下一部分还将讨论这个相同的 JFS 段错误问题，但不设置调试器，如果您在非 kgdb 内核环境中执行清单 5.8 中的代码，那么它使用内核可能生成的 Oops 消息。

- **Oops**

Oops（也称 panic，慌张）消息包含系统错误的细节，如 CPU 寄存器的内容。在 Linux 中，调试系统崩溃的传统方法是分析在发生崩溃时发送到系统控制台的 Oops 消息。一旦您掌握了细节，就可以将消息发送到 ksymoops 实用程序，它将试图将代码转换为指令并将堆栈值映射到内核符号。在很多情况下，这些信息就足够您确定错误的可能原因是什么了。请注意，Oops 消息并不包括核心文件。

让我们假设系统刚刚创建了一条 Oops 消息。作为编写代码的人，您希望解决问题并确定什么导致了 Oops 消息的产生，或者您希望向显示了 Oops 消息的代码的开发者提供有关您的问题的大部分信息，从而及时地解决问题。Oops 消息是等式的一部分，但如果不通过 ksymoops 程序运行它也于事无补。

ksymoops 需要几项内容：Oops 消息输出、来自正在运行的内核的 System.map 文件，还有 /proc/ksyms、vmlinux 和 /proc/modules。关于如何使用 ksymoops，内核源代码 /usr/src/linux/Documentation/oops-tracing.txt 中或 ksymoops 手册页上有完整的说明可以参考。Ksymoops 反汇编代码部分，指出发生错误的指令，并显示一个跟踪部分表明代码如何被调用。

首先，将 Oops 消息保存在一个文件中以便通过 ksymoops 实用程序运行它。清单 5.10. 显示了由安装 JFS 文件系统的 mount 命令创建的 Oops 消息，问题是由清单 5.8. 中添加到 JFS 安装代码的那三行代码产生的。

清单 5.10. ksymoops 处理后的 Oops 消息

```
ksymoops 2.4.0 on i686 2.4.17. Options used
... 15:59:37 sfb1 kernel: Unable to handle kernel NULL pointer dereference at
virtual address 0000000
... 15:59:37 sfb1 kernel: c01588fc
... 15:59:37 sfb1 kernel: *pde = 0000000
... 15:59:37 sfb1 kernel: Oops: 0000
... 15:59:37 sfb1 kernel: CPU:    0
... 15:59:37 sfb1 kernel: EIP:    0010:[jfs_mount+60/704]

... 15:59:37 sfb1 kernel: Call Trace: [jfs_read_super+287/688]
[get_sb_bdev+563/736] [do_kern_mount+189/336] [do_add_mount+35/208]
[do_page_fault+0/1264]
... 15:59:37 sfb1 kernel: Call Trace: [<c0155d4f>]...
... 15:59:37 sfb1 kernel: [<c0106e04 ...
... 15:59:37 sfb1 kernel: Code: 8b 2d 00 00 00 00 55 ...

>>EIP; c01588fc <jfs_mount+3c/2c0> <=====
```

```
...
Trace; c0106cf3 <system_call+33/40>
Code; c01588fc <jfs_mount+3c/2c0>
00000000 <_EIP>:
Code; c01588fc <jfs_mount+3c/2c0>   <=====
   0: 8b 2d 00 00 00 00      mov      0x0,%ebp      <=====
Code; c0158902 <jfs_mount+42/2c0>
   6:   55                  push      %ebp
```

接下来，您要确定 jfs_mount 中的哪一行代码引起了这个问题。Oops 消息告诉我们问题是由位于偏移地址 3c 的指令引起的。做这件事的办法之一是对 jfs_mount.o 文件使用 objdump 实用程序，然后查看偏移地址 3c。Objdump 用来反汇编模块函数，看看您的 C 源代码会产生什么汇编指令。清单 5.11. 显示了使用 objdump 后您将看到的内容，接着，我们查看 jfs_mount 的 C 代码，可以看到空值是第 109 行引起的。偏移地址 3c 之所以很重要，是因为 Oops 消息将该处标识为引起问题的位置。

清单 11. jfs_mount 的汇编程序清单

```
109  printk("%d\n",*ptr);

objdump jfs_mount.o

jfs_mount.o:    file format elf32-i386

Disassembly of section .text:

00000000 <jfs_mount>:
   0:55               push %ebp
  ...
  2c:    e8 cf 03 00 00      call       400 <chkSuper>
  31:    89 c3                     mov       %eax,%ebx
  33:    58                      pop       %eax
  34:    85 db                     test  %ebx,%ebx
  36:    0f 85 55 02 00 00 jne     291 <jfs_mount+0x291>
  3c:    8b 2d 00 00 00 00 mov   0x0,%ebp << problem line above
  42:    55                  push      %ebp
```

- **kdb**

Linux 内核调试器（Linux kernel debugger，kdb）是 Linux 内核的补丁，它提供了一种在系统能运行时对内核内存和数据结构进行检查的办法。请注意，kdb 不需要两台机器，不过它也不允许您像 kgdb 那样进行源代码级别上的调试。您可以添加额外的命令，给出该数据结构的标识或地址，这些命令便可以格式化和显示基本的系统数据结构。目前的命令集允许您控制包括以下操作在内的内核操作：

(1) 处理器单步执行；

(2) 执行到某条特定指令时停止；
(3) 当存取（或修改）某个特定的虚拟内存位置时停止；
(4) 当存取输入 / 输出地址空间中的寄存器时停止；
(5) 对当前活动的任务和所有其它任务进行堆栈回溯跟踪（通过进程 ID）；
(6) 对指令进行反汇编。

5.3. 系统引导和内核启动

5.3.1. Bootloader 程序的设计与实现

在一个嵌入式平台的实现中，BSP（Board Support Packet）占有十分重要的地位。在操作系统运行以前，BSP 构成了整个嵌入式系统的软件环境，管理着诸如存储器、中断等几乎所有的硬件资源，为包括操作系统在内的其它嵌入式软件的载入与调试提供支持。我们主要介绍基于 Motorola DrangonBall EZ/VZ 系列微处理器的 Bootloader。

该 Bootloader 程序包含了 BSP 的主要功能，它由以下三部分构成：

(1) 上位机与硬件平台的通信程序
(2) 硬件平台初始化程序
(3) 硬件平台监控程序

5.3.1.1. 硬件平台的通信

Motorola 的 DragonBall EZ/VZ 微处理器在 BootStrap 运行模式下可以通过串行端口直接与宿主机进行通信，并且允许用户直接向系统主存（RAM）下载并运行可执行程序。

本平台的 BSP 软件包就是利用这一功能实现上位机与硬件平台之间的通信的。首先通过 J3 跳线将目标平台的 CPU 运行状态置于 BootStrap 模式下，在上位机方面，运行一个 Perl 程序（ez328boot.pl）负责初始化硬件平台的串行端口，进而建立上位机与硬件平台的通信连接，并将监控程序和设备驱动程序下载到系统的 RAM 中。然后在上位机运行 Minicom 程序(一个通信管理程序，在此处被设置为管理串口)负责监控目标平台的软件运行情况，并提供与目标平台的交互。

5.3.1.2. 硬件平台初始化程序

在监控程序控制平台硬件资源以前，需要对整个平台进行初始化。这部分工作是在上节提到的 Perl 程序中由 ezboot_init 程序段实现的，主要通过对系统寄存器的设置进行以下的初始化：

(1) 状态寄存器（SCR）复位。
(2) 主存储器片选，包括 Flash Memory 和 RAM。
(3) 中断管理，包括打开或者屏蔽各级中断。

在程序实现过程中，每设置一个寄存器都会向终端回显符号*，用于监控初始化的进程。

5.3.1.3. 硬件平台监控程序

BSP 的监控程序主要完成对硬件平台的资源管理，并且提供与用户的交互。

针对本平台设计的 BSP 监控程序主要提供以下功能：

(1) 上位机文件下载

(2) 察看 Flash Memory 信息

(3) 擦除 Flash Memory 中的内容

(4) 烧写内核映像文件

(5) 引导 Flash Memory 中的内核启动

监控程序通过循环等待用户输入命令，根据用户的指令来触发系统动作，当用户需要向开发平台 RAM 中下载程序时，运行 xrecv 命令，这时在平台上启动了一个 Xmodem(一种串行传输协议)的服务器程序等待用户进行文件传输。用户可以通过上位 PC 机的 Minicom 程序采用 Xmodem 协议发送需要的文件。

如果系统判断用户键入的是 Flash_pg 命令，则调用 Flash 驱动程序将 RAM 中的文件烧写到 Flash 之中，起始地址用宏 BOOT_LINUX 表示。

通过 gomain 命令，可以从 BOOT_LINUX 地址开始运行烧写在 Flash 中的内核映像，同时系统的控制权也从 Bootloader 程序移交给了操作系统内核。

5.3.2. uCLinux 移植

uCLinux 的移植工作主要包括启动代码的修改，内核的链接与装入，参数传递与内核引导。uCLinux 内核布局分为特定于体系结构的部分和与体系结构无关的部分。在启动的第一阶段，内核中特定于体系结构的部分（arch 目录）首先执行，设置硬件寄存器、配置内存映射、执行特定于体系结构的初始化，然后将控制转给内核中与体系结构无关的部分。系统的其余部分在第二个阶段期间进行初始化。内核树下的目录 arch 由不同的子目录组成，不同的子目录对应不同的体系结构(M68K,MIPS,ARM,I386,SPARC,PPC 等).我们以 M68K 体系为例来介绍内核的启动过程。

5.3.2.1. 第一阶段

这个阶段是体系结构相关的。通常代码都集中在一个完全由 m68k 汇编代码写成的 crt0.S 文件中，这个文件位于/arch/<target>/下。

- **启动代码（crt0.S）**

crt0 (C RunTime 0) ，顾名思义，它包含了操作系统的初始化启动代码，这些代码将在 C 语言代码之前执行。针对不同的目标硬件，需要编写不同的 crt0 代码。.S 代表 crt0 文件通常用汇编代码写成，它的目标代码通常最早被链接和执行，即 CPU 一加电就跳到这里执行，然后由它启动内核的其他部分。

Crt0.S 定义一个特殊的符号如_start，代表它的默认起始地址，同时它也是整个内核二进制镜像的起始标志。

crt0.S 主要完成以下功能：

（1） 定义数据段、代码段、bss 段起始地址变量，并对 bss 段进行初始化

（2） 通过寄存器设置初始化系统硬件

（3） 关闭中断
（4） 初始化 LCD 显示
（5） 将数据段数据复制到内存中，已备内核运行时使用
（6） 跳转至内核起始函数 start_kernel
（7） 主要寄存器的修改

在系统初始化过程中，比较主要的几个寄存器相应的设置如下：

片选组基地址寄存器(CSGBA)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	GBA 28	GBA 27	GBA 26	GBA 25	GBA 24	GBA 23	GBA 22	GBA 21	GBA 20	GBA 19	GBA 18	GBA 17	GBA 16	GBA 15	GBA 14	-
RESET	0x0000															
ADDR	0x(FF)FFF100,102,104,106															

该寄存器设置了片选地址范围的高 15 位地址，启动 Flash 的起始地址是 0x01000000，故而该寄存器值应该设置为 0x0800。

movew #0x0800， 0xfffff100

片选寄存器(CSA，CSB)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	GBA 28	GBA 27	GBA 26	GBA 25	GBA 24	GBA 23	GBA 22	GBA 21	GBA 20	GBA 19	GBA 18	GBA 17	GBA 16	GBA 15	GBA 14	-
RESET	0x0000															
ADDR	0x(FF)FFF100,102,104,106															

RO：选中的存储介质是否只读。
Flash：选中存储介质是否为 Flash。
BSW：数据总线宽度。0=8 位；1=16 位
WS：表示在内部信号/DTACK 确认片选信号返回以前插入的等待状态数。
SIZE：所选中的存储介质容量。
EN：片选有效。0=无效；1=有效

片选信号选中的存储介质是两片 Flash Memory，容量为 2MB，数据宽度为 16 位，故而该寄存器值设为 0x0189。

movew #0x0189， 0xfffff110
movew #0x0189， 0xfffff112

DRAM 存储配置寄存器(DRAMMC)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	GBA 28	GBA 27	GBA 26	GBA 25	GBA 24	GBA 23	GBA 22	GBA 21	GBA 20	GBA 19	GBA 18	GBA 17	GBA 16	GBA 15	GBA 14	-
RESET	0x0000															
ADDR	0x(FF)FFF100,102,104,106															

控制 DRAM 器存器(DRAMC)

DRAM 片选组寄存器(CSGBD)

同 CSGBA

DRAM 片选寄存器(CSD)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	R0	SOP	ROP	UPSIZ1	UPSIZ0	COMB	DRAM	FLASH	BSW	WS2	WS1	WS0	SIZ2	SIZ1	SIZ0	EN
RESET	0x0200															
ADDR	0x(FF)FFF116															

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	EN	RM	BC1	BC0	CLK	EDO	PGSZ		WS1	WS0	MSW	LSP	SLW	LPR	RST	DWE
RESET	0x0000															
ADDR	0x(FF)FFFC02															

8MB，16bit 数据宽度的 DRAM，以上各寄存器配置如下

```
movew    #0x8f00，   0xfffffc00       /* DRAM configuration */
movew    #0x9667，   0xfffffc02       /* DRAM control */
movew    #0x0000，   0xfffff106       /* DRAM at 0x00000000 */
movew    #0x068f，   0xfffff116       /* 8Meg, 16bit, enable, 0ws */
```

中断屏蔽寄存器(IMR)

该寄存器各位设置为 1 即将相应的中断屏蔽。在系统初始化和内核启动期间，为了保证系统的安全，需要将各级（本平台为 7 级）中断屏蔽，在系统控制权转交给内核后，由内核在适当时候将中断打开。

```
movel    #0x007FFFFF, %d0      /* IMR */
movel    %d0,        0xfffff304
```

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	R0	SOP	ROP	UPSIZ1	UPSIZ0	COMB	DRAM	FLASH	BSW	WS2	WS1	WS0	SIZ2	SIZ1	SIZ0	EN
RESET	0x0200															
ADDR	0x(FF)FFF116															

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	EN	RM	BC1	BC0	CLK	EDO	PGSZ		WS1	WS0	MSW	LSP	SLW	LPR	RST	DWE
RESET	0x0000															
ADDR	0x(FF)FFFC02															

LCD 控制器的初始化

DragonBall EZ 的 LCD 控制器为操作系统的 LCD 驱动程序提供 LCD 屏的数据和信息，并且将显示数据通过周期性的 DMA 方式从系统内存的一块区域传递给 LCD 屏显示出来。在数据传递的过程中占用非常小的总线带宽，从而对整个系统的运行性能影响不大。

可以通过对以下几个寄存器的设置来对 LCD 控制器进行编程：

LCD 起始地址寄存器(LSSA)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	R0	SOP	ROP	UPSIZ1	UPSIZ0	COMB	DRAM	FLASH	BSW	WS2	WS1	WS0	SIZ2	SIZ1	SIZ0	EN
RESET	0x0200															
ADDR	0x(FF)FFF116															

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	EN	RM	BC1	BC0	CLK	EDO	PGSZ		WS1	WS0	MSW	LSP	SLW	LPR	RST	DWE
RESET	0x0000															
ADDR	0x(FF)FFFC02															

该寄存器设置用于存储显示数据的内存块的起始地址，这里我们设置为 splash_bits，该全局变量在内核映像载入内存的时候赋值，将该段内存初始化为包含版本信息的位图，在图形用户界面启动前，LCD 将显示该位图。

movel #splash_bits， 0xfffffA00

LCD 屏宽寄存器(LXMAX)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	-						XM9	XM8	XM7	XM6	XM5	XM4				
RESET	0x03F0															
ADDR	0x(FF)FFFA08															

该寄存器设置 LCD 显示屏的点阵数宽度

LCD 屏高寄存器(LYMAX)

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	-							YM8	YM7	YM6	YM5	YM4	YM3	YM2	YM1	YM0
RESET	0x01FF															
ADDR	0x(FF)FFFA0A															

该寄存器设置 LCD 显示屏的点阵数高度

我们采用的 LCD 屏是 320×240 点阵格式的，所以两个寄存器的值应该设置如下：

```
movew    #0x0140，  0xFFFFFa08    /* LXMAX */
movew    #0x00EF，  0xFFFFFa0a    /* LYMAX */
```

在 crt0.S 文件做完初始化工作，通过跳转语句跳转到 start_kernel 后，开始启动操作系统，进入硬件体系结构（arch）无关的代码。

- **内核链接和装入**

内核可执行文件由许多链接在一起的目标文件（以 elf 为例）组成。Elf 文件有许多段，如文本段（text）、数据段（data）、bss 段等等。这些段都是由一个称为链接器脚本（linker description，简称 ld）的文件链接并装入的。这个链接器脚本的功能是将输入目标文件的各段映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。.ld 文件是位于 arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个段并将它们装入内存中特定偏移量处，所以必须针对系统的内存映射（见 crt0.S）来定制。

.ld 文件根据地址映像表编写如下：

```
MEMORY
    {
    romvec : ORIGIN = 0x01000000, LENGTH = 0x00400
    flash   : ORIGIN = 0x01000400, LENGTH = 0x400000-0x00400
    eflash : ORIGIN = 0x01400000, LENGTH = 1
    ramvec : ORIGIN = 0x00000000, LENGTH = 1024
    ram     : ORIGIN = 0x00020000, LENGTH = 0x00800000 - 0x20000
    eram    : ORIGIN = 0x00800000, LENGTH = 1
    }
SECTIONS
{
    .romvec :
    {
    _romvec = . ;
    __rom_start = . ;
    } > romvec
        .text :
    {
    text_start = . ;
        *(.text)
    _etext = . ;
    __data_rom_start = ALIGN ( 4 ) ;
```

```
    } > flash
.eflash :
{
_flashend = . ;
} > eflash
```

其中 romvec 段存放的是中断向量表，flash 段是闪存中内核映像与主文件系统的地址。如果要在 DragonBall EZ/VZ 芯片的 Normal 模式下内核可以直接从 Flash 启动。则需要把代码段.text 载入到 flash 中。

通过对上述两个文件的修改，系统的第一阶段初始化已经完成，内核已经被载入到相应的地址，并且系统的控制权也移交给内核。

5.3.2.2. 第二阶段

这个阶段完成参数传递和内核引导。start_kernel 调用 setup_arch 作为执行的第一步，在其中完成特定于体系结构的设置。这包括初始化硬件寄存器、标识根设备和系统中可用的 DRAM 和闪存的数量、指定系统中可用页面的数目、文件系统大小等等。所有这些信息都以参数形式从启动代码传递到内核。

setup_arch 还需要对闪存、系统寄存器和其它特定设备执行内存映射。一旦完成了特定于体系结构的设置，控制就返回到初始化系统其余部分的 start_kernel 函数。这些附加的初始化任务包含：

(1) 设置陷阱
(2) 初始化中断
(3) 初始化计时器
(4) 初始化控制台
(5) 调用 mem_init，计算内存的页面数量
(6) 初始化 slab 分配器，并为 VFS、缓冲区高速缓存等创建 slab 高速缓存
(7) 建立各种文件系统，如 proc、ext2 和 JFFS2
(8) 创建 kernel_thread，它执行文件系统中的 init 命令并显示提示符。如果在 /bin、/sbin 或 /etc 中没有 init 程序，那么内核将执行文件系统的 /bin 中的 shell。

第六章 设备驱动程序

6.1. 概述

在我们刚开始接触计算机，学习基础知识时就知道根据冯.诺依曼结构，典型的计算机系统可分为运算器、控制器、存储器、输入输出设备五部分。其中输入输出设备，在计算机系统中起着至关重要的作用，它不仅是人机交互的桥梁，更是对计算机性能的有力扩展。在嵌入式系统中，也是如此。作为一个实用的嵌入式系统，无论是用于人机交互的基本外设，如触摸屏、小键盘、LCD 等，还是用来完成其他具体应用功能的硬件板卡都属于输入输出设备。如何有效的管理这些输入输出设备，是操作系统要解决的一个重要问题。本章将介绍 linux/uclinux 中设备管理的基本原理，并重点介绍设备驱动程序的实现细节。

linux 作为 unix 的一个变种，它继承了 unix 的设备管理方法，将所有的设备看作具体的文件，通过文件系统层对设备进行访问。所以在 linux/uclinux 的框架结构中，和设备相关的处理可以分为两个层次－－文件系统层和设备驱动层。设备驱动层屏蔽具体设备的细节，文件系统层则向用户提供一组统一的规范的用户接口。这种设备管理方法可以很好的做到“设备无关性”，使 linux/uclinux 可以根据硬件外设的发展进行方便的扩展,比如要实现一个设备驱动程序，只要根据具体的硬件特性向文件系统的提供一组访问接口即可。整个设备管理子系统的结构如图 6.1 所示。

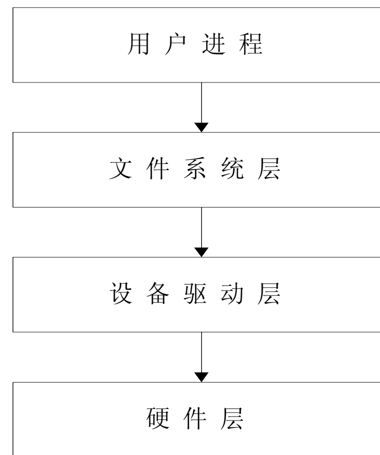


图 6.1 设备驱动分层示意图

- **用户进程**　用户进程一般位于内核之外，当它需要操作设备时，可以就像访问普通文件一样，通过调用 read(),write()等文件操作系统调用来完成对设备文件的访问和控制。

- **文件系统层**　它位于用户进程层下面，属于内核空间，基本功能是执行适合于所有设备的输入输出功能，使用户透明的访问文件。通过本层的封装，设备文件在上一层看来就和普通文件没有区别，也拥有读、写和执行权限，拥有和它对应的索引节点等。在用户进程发出系统调用要求输入输出操作时，文件系统层就处理请求的权限，通过设备驱动层的接口将任务传到驱动程序。

- **设备驱动层**　设备驱动程序位于内核中，它根据文件系统层的输入输出请求来操作硬件上的设备控制器，完成设备的初始化、打开释放、以及数据在内核和设备间的传递等操作。

6.1.1. 设备类型

Linux 中的设备可以分为三类：字符设备，块设备和网络设备。其中字符设备没有缓冲区，数据的处理是以字节为单位按顺序进行的，它不支持随机读写。普通打印机、系统的串口以及终端显示器是比较常见的字符设备，嵌入式系统中简单的按键、触摸屏、手写板也都属于字符设备。块设备是指那些在输入/输出时数据处理以块为单位的设备，它一般都采用了缓存技术，支持数据的随机读写。典型的块设备有硬盘、cd-rom 等。对用户来说块设备和字符设备的访问接口都是一样的，都是一组基于文件的系统调用，如 read，write 等，它们实现上细节的区别仅在内核和驱动程序的软件接口上。

Linux 中网络设备的实现方法不同于字符型设备和块型设备，它面向的上一层不是文件系统层而是网络协议层，设备节点只有在系统正确初始化网络控制器之后才能建立。内核和网络设备驱动程序间的通信，与字符设备驱动程序、块设备驱动程序与内核间的通信也是完全不一样的。具体的问题将在 6.4.2 小节中详细介绍。

一个运行的 linux 系统，当前使用的设备可以通过文件/proc/devices 查看。

6.1.2. 设备号

传统的设备管理上，除了设备类型外，linux/uclinux 内核还需要一对被称作为主设备号、次设备号的参数，才能唯一的标识设备。

主设备号标识设备对应的驱动程序。系统中不同的设备可以有相同的主设备号，主设备号相同的设备使用相同的驱动程序。例如，/dev/null 和/dev/zero 都由驱动程序 1 管理，而所有的 tty 和 pty 都由驱动程序 4 管理。内核利用主设备号将设备与相应的驱动程序对应起来。

次设备号用来区分具体驱动程序的实例。从上面可以知道，一个主设备号可能有多个设备与之对应，这多个设备正是在驱动程序内通过次设备号来进一步区分的。次设备号只能由设备驱动程序使用，内核的其他部分仅将它作为参数传递给驱动程序。

向系统添加一个驱动程序相当于添加一个主设备号，字符型设备主设备号的添加和注销分别通过调用函数 register_chrdev()和 unregister_chrdev()实现，这两个函数原型详见<linux/fs.h>文件。

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

这两个函数运行成功是返回 0，运行失败返回一个负的错误码。参数 major 对应所请求的主设备号， name 对应设备的名字，fops 对应于和该设备对应的一个结构，这个结构在后面 6.2.2 中有详细介绍。

在设备注册时主设备号的获取可以通过动态分配或指定一个固定值的方法获得。在嵌入式设备中外设较少，所以一般采用指定的方法就可以完成系统的功能。在文件系统中创建一个设备节点的命令为 mknod。具体用法为：

mknod　设备名 设备类型 主设备号 次设备号

在 linux/uclinux 中和主设备号次设备相关的宏有 MAJOR(dev)、MINOR(dev)和 MKDEV(ma,mi)。其中 MAJOR(dev)用来获取设备 dev 的主设备号，MINOR(dev) 用来获取设备 dev 的次设备号，MKDEV(ma,mi)功能是根据主设备号 ma 和次设备号 mi 来得到相应的 dev。这三个宏中 dev 为 kdev_t 结构，这个结构根据内核各个版本不同而不同，但主要功能是用它来保存设备号。

这些宏的定义和 kdev_t 的定义见</linux/include/linux/kdev_t.h>文件。

6.1.3. 模块化编程

在 unix 系统中是不支持模块化编程的，这为驱动程序的动态添加设置了很多障碍。例如每添加一个设备就必须从头重新编译一次内核，这给系统功能的扩展带来了很多不便。但是在 linux 系统中提供了一种全新的模块化机制：“module”。利用这种机制，可以根据需要在不重新编译内核的情况下，将编译好的模块动态的插入运行中的内核，或者将内核中已经存在的一个模块移走。可以看出这种机制为驱动程序开发调试提供了很大的方便。

在运行的系统中可以通过 lsmod 察看内核中已经动态加载的模块。模块的安装和从内核中卸载可以通过以下命令实现，他们操作的对象是经过编译但没有连接的.o 文件。

```
insmod　xxxx
rmmod　xxxx
```

对应的模块化编程，源程序中必须至少提供 init_module()和 cleanup_module()两个函数。一个简单的模块化程序如下 module_demo.c 所示。

```
/*------------ module_demo.c ------------*/
#define MODULE
#include <linux/module.h>
int init_module(void)
{
    printk("\nhello，world!\n\n”);
    return 0;
}
void cleanup_module(void)
{
    printk("\n Bye Bye\n\n");
}
```

module_demo.c 可以通过下面命令来编译：

```
gcc -c -D__KERNEL__ -DMODULE -o module_demo module_demo.c
```

在我们得到了 module_demo 后，就可以用 insmod 命令把它动态的加入内核了。

```
# insmod module_demo
# hello world!
# rmmod module_demo
# Bye Bye
```

6.2. 设备文件接口

在前面的介绍中，我们已经对 linux 下设备驱动程序有了大致的了解，接下来，我们将看看驱动程序对它管理的设备能够完成哪些不同的具体操作。

6.2.1. 用户访问接口

在 linux 系统中，对用户程序而言，设备驱动程序隐藏了设备的具体细节，对各种不同设备提供了一致的接口，一般来说是把设备映射为一个特殊的设备文件，用户程序可以象对其它文件一样对此设备文件进行各种操作。在系统内部，I/O 设备的存取通过一组固定的入口点来进行，这组入口点是由每个设备的设备驱动程序提供的。一般来说，字符型设备驱动程序能够提供如下几个入口点：

- **open** 入口点

打开设备准备 I/O 操作。对字符设备文件进行打开操作，都会调用设备的 open 入口点。open 子程序必须对将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则 open 子程序必须设置一些标志以表示设备处于忙状态。其调用格式为:

int open(char *filename, int access);

该函数表示按 access 的要求打开名为 filename 的文件, 返回值为文件描述字, 其中 access 有两部分内容: 基本模式和修饰符, 两者用" "("或")方式连接。修饰符可以有多个, 但基本模式只能有一个。access 的规定如表 6.1.。

表 6.1. access 的规定

基本模式	含义	修饰符	含 义
O_RDONLY	只读	O_APPEND	文件指针指向末尾
O_WRONLY	只写	O_CREAT	文件不存在时创建文件, 属性按基本模式属性
O_RDWR	读写	O_TRUNC	若文件存在, 将其长度缩为 0, 属性不变
O_BINARY	打开一个二进制文件		
O_TEXT	打开一个文字文件		

open()函数打开成功, 返回值就是文件描述字的值(非负值), 否则返回-1。

- **close** 入口点

close()函数的作用是关闭由 open()函数打开的文件, 其调用格式为:

int close(int handle);

该函数关闭文件描述字 handle 相连的文件。

- **read** 入口点

从设备上读数据。对于有缓冲区的 I/O 操作，一般是从缓冲区里读数据。read()函数的调用格式为:

int read(int handle, void *buf, int count);

read()函数从 handle(文件描述字)相连的文件中, 读取 count 个字节放到 buf 所指的缓冲

区中，返回值为实际所读字节数，返回-1 表示出错。返回 0 表示文件结束。
2） write 入口点

往设备上写数据，对于有缓冲区的 I/O 操作，一般是把数据写入缓冲区里。write()函数的调用格式为:

int write(int handle, void *buf, int count);

write()函数把 count 个字节从 buf 指向的缓冲区写入与 handle 相连的文件中，返回值为实际写入的字节数。

- **ioctl 入口点**

执行读、写之外的操作。函数原型为：

int ioctl(int fd, int cmd, …)

参数 cmd 不经修改的传递给驱动程序，可选的 arg 参数无论是指针还是整数值，都以 unsigned long 的形式传递给驱动程序。

6.2.2. 文件操作

6.2.2.1. file_operations 结构

在 linux 系统里，设备驱动程序所提供的这组入口点由一个结构来向系统进行说明，这个结构就是 strcut file_operations,它是一组具体操作的集合，包括打开设备，读取设备等。我们在向内核注册设备的函数中，一个参数就是指向 strcut file_operations 的指针。下面具体介绍完成这些操作的函数，如果开始时觉得比较费解的话，可以先跳过这一部分，等到自己编写驱动程序时再来查阅（或结合本章最后所附的实例），这样更便于理解。

方便起见，先把 struct file_operations 的原型写出来，使读者有一个初步的印象：

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, const char *, int);
    int (*readdir) (struct inode *, struct file *, void *, filldir_t);
    int (*select) (struct inode *, struct file *, int, select_table *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

接下来我们按顺序具体介绍每个函数。

```
Int (*lseek) (struct inode *, struct file *,  off_t, int);
```

用来修改一个文件的当前读写位置，并将新位置作为正的返回值返回，出错时返回一

个负值。如果驱动程序没有设置这个函数，相对于文件尾的定位操作失败，其他定位操作修改 file 结构（稍后介绍）中的位置计数器，并成功返回。

Int (*read) (struct inode* , struct file *, char *,int);

用来从设备中读取数据。

- 如果返回值等于 count 参数传递给 read 系统调用的值，所请求的字节数传输就成功完成了，这是最理想的情况；
- 如果返回值是正的，但是比 count 小，只有部分数据成功传送。这种情况因设备的不同可能有许多原因。大部分情况下，程序会重新读数据。
- 如果返回值为 0，表示已经到达了文件尾。
- 负值意味着发生了错误。具体数值指明了发生何种错误，并在<linux/errno.h>中定义其类型。
- 如果设备暂时无数据到达，应该调用阻塞，使之进入等待队列，而不用重复的对设备进行读操作。

Int (*write) (struct indoe *, struct file *,const *,int);

向设备发送数据。如果没有这个函数，write 系统调用向调用程序返回一个-EINVAL。Write 返回值的规则与 read 相似：

- 如果返回值等于 count，则完成了请求数目的字节传送。
- 如果返回值是正的，但小于 count，只传输了部分数据，返回值代表成功写入的字节数。
- 如果返回值为 0，什么也没写，这并不是发生了错误，而且也没有什么缘由需要返回一个错误编码，在这种情况下，标准库会重复调用 write。
- 返回值为负值意味着发生了错误。

Int (*readdir) (struct inode *, struct file *, void *, filldir_t)

它仅用于目录，所以对设备节点来说，这个操作应该为 NULL。

Int (* select)(struct inode *, struct file , int, select_table *);

Select 一般用于程序询问设备是否可读和可写，或是否一个“异常”条件发生了。如果指为 NULL，系统假设设备总是可读和可写的，而且没有异常需要处理。这个操作可以和阻塞（block）交互使用，我们将在稍后的中断中做详细介绍。

Int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);

系统调用 ioctl 提供一种调用设备相关命令的操作。如果设备不提供 ioctl 入口点，对于任何内核没有定义的请求，ioctl 系统调用将返回-EINVAL。当调用成功时，返回给调用程序一个非负返回值。

Int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);

Mmap 用来将设备内存映射到进程内存中。如果设备不支持这个操作，mmap 系统调用将返回-ENODEV。

Int (*open) (struct inode *, struct file *);

一般是在设备节点上的第一个操作。如果不声明这个操作，系统默认“打开”永远成功，但不会通知驱动程序。

Void (*release) (struct inode *, struct file *);

当节点被关闭时调用这个操作。也可以置 NULL。

Int (*fsync) (struct inode *, struct file *);

刷新设备。如果驱动程序不支持，fsync 系统调用返回-EINVAL。

Int (*fasync) (struct inode *, struct file *, int);

这个操作用来通知设备 FASYNC 标志的变化，用异步触发通知设备有数据到来。(具体有关异步触发机制将在 6.2.3I/O 操作中讲述)

6.2.2.2. file 结构

在<linux/fs.h>中定义的 struct file 是设备驱动程序所适用的又一个最重要的数据结构。file 结构代表一个“打开的文件”，它与由 struct inode 表示的“磁盘设备文件”有所不同。file 结构由内核在打开时创建而且在关闭前作为参数传递给操作在设备上的函数。在文件关闭后，内核释放这个数据结构。

file 结构是由系统默认生成的，驱动程序从不去填写它，只是简单的访问别处创建的结构，所以，下面的讲述只是为了让读者搞清楚这个结构，如果不感兴趣的话，可以大胆地忽略这些字段，而直接去使用它。

在内核源代码中，指向 struct file 的指针通常称为 file 或 filp（“文件指针”）。现在来看看 struct file 中最重要的字段：

mode_t f_mode

文件模式由 FMODE_READ 和 FMODE_WRITE 识别。你可能需要在你的 ioctl 函数中查看这个域来检查读/写权限，但由于内核在调用你的驱动程序的 read 和 write 前已经检查了权限，你无需在这两个操作中检查权限。例如，一个不允许的写操作在驱动程序还不知道的情况下就已经被内核拒绝了。

Loff_t f_ops

当前读/写位置。Loff_t 是一个 64 位数值。如果驱动程序需要知道这个值，可以直接读取这个字段。如果定义了 lseek 函数，应该更新 f_ops 的值。当传输数据时，read 和 write 也应更新这个值。

Unsigned short f_flags

文件标志，如 O_RDONLY、O_NONBLOCK 和 O_SYNC。驱动程序为了支持非阻塞型操作需要检查这个标志，而其他标志很少用到。注意，检查读/写权限应该查看 f_mode 而不是 f_flags。所有这些标志都定义在<linux/fcntl.h>中。

Struct inode *f_inode

打开文件所对应的 i 节点。Inode 指针是内核传递给所有文件操作的第一个参数，所以你一般不需要访问 file 结构的这个字段。在某些特殊情况下你只能访问 struct file 时，你可以通过这个字段找到相应的 i 节点。

Struct file_operations *f_op

与文件对应的操作，我们在前面已经讲过，这里不再赘述。

Void *private_data

系统调用 open 函数前将这个指针置为 NULL。驱动程序可以将这个字段用于任意目的或者简单忽略这个字段。驱动程序可以用这个字段指向已分配的数据，但是一定要在内核释放 file 结构前的 release 函数中清除它。Private_data 是跨系统调用保存状态信息的非常有用的资源。

6.2.3. I/O 操作

- ioctl

通常补充设备读写操作的功能之一就是控制硬件，最常用的通过设备驱动程序完成控制动作的方法就是实现 ioctl 函数。如前所述，ioctl 系统调用为驱动程序执行相关操作提供了一个与设备相关的入口点。与 read 和其他操作不同，ioctl 是与设备相关的，它允许应用程序访问被驱动硬件的特殊功能——配置设备以及进入或退出操作模式。而这些“控制操

作”一般情况下无法通过 read/write 文件操作完成，比如改变串口的波特率。

- **阻塞型 I/O**

当进行 read 操作时，如果没有数据可读，而又没有到达文件末尾，这时有两种处理办法：阻塞型和非阻塞型操作。

在阻塞型操作的情况下，应该实现下列操作：

- 如果进程调用 read，但还没有数据，进程必须阻塞。当数据到达时，进程被唤醒，并将数据返回给调用者。可以调用如下函数之一让进程进入睡眠状态：

```
void interruptible_sleep_on(struct wait_queue **q);
void sleep_on(struct wait_queue **q);
```

然后用对应的如下两个函数中的一个唤醒进程：

```
void wake_up_interruptible(struct wait_queue **q);
void wake_up(struct wait_queue **q);
```

进程睡眠，就是进入等待队列。等待队列很容易使用，你不需要对它的内部细节了解的非常清楚，只要按以下步骤处理就可以了：先声明一个 struct wait_queue *变量。你需要为每一个可以让进程睡眠的事件预备这样一个变量；将该变量的指针作为参数传递给不同的 sleep_on 和 wake_up 函数。

唤醒进程使用的是与进程睡眠时相同的一个队列，因此，必须为每一个可能阻塞进程的事件建立一个等待队列。

那么，同样使进程进入睡眠状态，interruptible_sleep_on 与 sleep_on 又有什么区别呢？sleep_on 不能被信号取消，但 interruptible_sleep_on 可以，也就是说，前者适用于不可中断进程，后者适用于可中断进程。wake_up_interruptible 和 wake_up 也同样如此。

在驱动程序中，由于进程仅在进行 read 或 write 操作期间才会睡眠在驱动程序代码上，所以你应该调用 interruptible_sleep_on 和 wake_up_interruptible。

2）如果进程调用了 write，缓冲区又没有空间，进程也必须阻塞，而且它必须使用与用来实现读操作的等待队列不同的队列。当数据写进设备后，输出缓冲区中空出部分空间，唤醒进程，write 调用成功完成，如果缓冲区中没有请求的 count 个字节，则进程可能只是完成了部分写操作。

相对于阻塞型操作对进程的处理，非阻塞型操作立即返回，此时，如果进程在没有数据就绪时调用了 read，或者在缓冲区没有空间时调用了 write，系统简单的返回-EAGAIN。

- select

在使用非阻塞型 I/O 时，应用程序要经常利用 select 系统调用，此外还用来实现不同源输入的多路复用。为了保存所有正在等待文件（或设备）的信息，linux2.0.x 的 select 系统调用使用了 select_table 结构。当 select 发现无需阻塞时，它返回 1；当进程应该等待，它代替 read 或 write 使进程进入睡眠状态，在这种情况下，要在 select_table 结构中加入等待队列，并且返回 0。

其实我们可以直观上理解 select 的含义，在进程要求的资源之间选择，当其中没有一个可以接收或返回数据时，进程才真正进入睡眠状态。

下面我们谈谈 select 与 read 和 write 的交互

select 调用的目的是判断是否有 I/O 操作会阻塞。从这个方面说，它是对 read 和 write 的补充。由于 select 可以让驱动程序同时等待多个数据流，select 在这方面也是很有用的。

Select 的工作是由函数 select_wait 和 free_wait 完成的，select_wait 是声明在

<linux/sched.h>里的内嵌函数，而 free_wait 则是在 fs/select.c 中定义的。它们使用的数据结构是 struct select_table_entry 数组，每一项都是有 struct wait_queue 和 struct wait_queue **组成的。前者是插入到设备等待队列的实际数据结构（当调用 sleep_on 时以局部变量形式存在的数据结构），而后者是在所选条件有一个为真时，则将当前进程从队列中删除时所需要的“句柄”。Select_wait 将下一个空闲的 select_table_entry 插入到指定的等待队列中。当系统调用返回时，free_wait 利用对应的指针删除自己等待队列中的每一项。

- 异步触发

尽管大多数时候阻塞型和非阻塞型操作的组合，以及 select 可以有效地查询设备，但某些时候用这种技术管理就不够高效了。举个例子，一个在低优先级执行长计算循环的进程，但它需要尽可能快地处理输入的数据。如果输入的通道是键盘，你可以向进程发送信号（使用“INTR”字符，一般就是 Ctrl-C），但是这种信号是 tty 层的一部分，在一般字符设备中没有用到。此外，任何输入数据都应该产生一个中断，而不仅仅是 Ctrl-C。一般的方法是不行的，这里必须用到异步触发。

要打开异步触发机制，用户程序必须执行两个步骤。首先，它们指定进程是文件的“属主”。文件属主的用户 ID 保存在 filp->f_owner 中，可以通过 fcntl 系统调用的 F_SETOWN 命令设置这个值。此外，为了确实打开异步触发机制，用户程序还必须通过另外一个 fcntl 命令设置设备的 FASYNC 标志。

在完成这两个步骤后，无论何时新数据到达，输入文件都产生一个 SIGIO 信号。信号发送给存放在 flip->f_owner 的进程（如果是负值，则是进程组）。

下面的列表从驱动程序的角度给出了如何实现这种操作的详细过程：

1) 当调用 F_SETOWN 时，除了对 flip->f_owner 赋值以外什么也不做。
2) 当调用 F_SETFL 打开 FASYNC 标志时，驱动程序的 fasync 函数被调用。无论 FASYNC 的值何时发生变化，该函数都被调用，通知驱动程序该标志的变化，以便驱动程序能够正确的响应。在文件被打开时，这个标志默认是被清零的。
3) 当数据到达时，向所有注册异步触发的进程发送 SIGIO 信号。

尽管实现的第一步很简单——在驱动程序端没有什么可做的——其他步骤则为了跟踪不同的异步接收者，要涉及一个动态数据结构，同时可能有个多个接收者。然而，这个动态数据结构不依赖于某个特定设备，内核提供了一套合适的通用的实现方法。

简单的，只需根据如下原型调用两个函数：

```
int fasync_helper(struct inode * inode, struct file * filp, int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct * fa, int sig);
```

当打开文件的 FASYNC 标志被修改时，从感兴趣进程列表上增加或删除文件可以调用前者，当数据到达时，则应该调用后者。

异步触发机制使用的数据结构与 struct wait_queue 结构非常相似，因为两者都设计等待事件。不同之处是，前者使用 struct file 代替了 struct task_struct 。

6.3. 中断处理

在现代操作系统中，中断是发挥硬件尤其是 cpu 性能的一个重要方面。一般情况下操作系统向具体的硬件发出一个请求操作，该硬件就在自己的设备控制器控制下工作，在它完成所请求的任务时，利用中断来通知操作系统，操作系统根据它的状态调用相应的处理函数进行处理，这样就避免了在硬件工作时操作系统的无效等待，提高了系统的运行效率。在 linux 中为中断的管理提供了很好的接口，从应用编程角度来看编写一个中断处理程序

只要根据具体应用实现中断服务子程序，并利用一系列 Linux API 函数向内核注册该服务子程序就行了，具体的调度处理在 linux 内部实现。

6.3.1. 注册中断处理程序

向内核注册中断处理程序主要实现两个功能，一是注册中断号，二是注册中断处理函数。在 linux 中对应的中断处理注册函数为：

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags, const char *device, void *dev_id);
```

返回值：

request_irq 返回 0 表示成功，返回-INVAL 表示 irq>15 或 handler==NULL，返回-EBUSY 表示中断已经被占用且不能共享。

参　数:

unsigned int irq

该参数表示所要申请的中断号。中断号可以在程序中静态的指定，或者在程序中自动探测。在嵌入式系统中因为外设较少，所以一般静态指定就可。

unsigned long flags

flags 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的一个选项是 SA_INTERRUPT。如果 SA_INTERRUPT 位置 1，表示这是一个快速处理中断程序，如果 SA_INTERRUPT 位为 0 表示这是一个慢速处理中断程序。快速处理程序运行时，所有中断都被屏蔽，而慢速处理程序运行时，除了正在处理的中断外，其它中断都没有被屏蔽。Flags 另外两个选项是中断号是否可以被共享。中断号可以被共享的情况下，要求每一个共享此中断的处理程序在申请中断时在 flags 里设置 SA_SHIRQ，这些处理程序之间以 dev_id 来区分。如果中断由某个处理程序独占，则 dev_id 可以为 NULL。

const char *device

device 为设备名，将会出现在/proc/interrupts 文件里。

void *dev_id

dev_id 为申请时告诉系统的设备标识。

void (*handler)(int irq, void* device, struct pt_regs* regs)

handler 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号。dev_id 为申请时告诉系统的设备标识。regs 为中断发生时寄存器内容。device 为设备名，将会出现在/proc/interrupts 文件里。

中断信息释放函数相应的如下，它的参数意义同上。

```
void free_irq(unsigned int irq, void *dev_id)
```

中断处理程序中中断号的自动探测主要是通过<linux/interrupt.h>中声明三个函数来实现的。

```
extern unsigned long probe_irq_on(void);
extern int probe_irq_off(unsigned long);
extern unsigned int probe_irq_mask(unsigned long);
```

当要探测中断号时，驱动程序首先关闭所有的中断，并打开所有没有分配的中断号，

然后让设备产生一个中断。这时候设备产生的中断通过可编程中断控制器被传递到内核，Linux 内核再读取中断状态寄存器并把它的内容返回到设备驱动程序。非 0 的返回值表示在刚才探测时中发生了一或多个中断。接下来驱动程序关闭所有没有分配的中断号，再让设备产生一个中断进行验证。如果这时候内核检测不到中断，就表示刚才得到的返回值是一个可用的中断号，这样驱动程序就可以用该中断号向内核注册它了。具体的步骤如下：

(1) 关闭所有的设备中断；
(2) sti()打开没有分配的中断号；
(3) irqs = probe_irq_on()；
(4) 要探测的设备产生一个中断；
(5) 系统得到设备中断；
(6) irq = probe_irq_off(irqs)得到设备中断号;

在一个设备驱动程序向内核注册了中断服务程序后，中断到来时的调度就由内核的中断处理子系统来完成了。中断处理子系统的一个主要任务是根据中断号找到正确的中断处理代码段。如图 6.2 所示 Linux 中维护了一个 irq_action 指针指向的中断函数处理向量表，该表由 irqaction 结构组成。每一个 irqaction 结构都包括了一个中断处理程序的信息，如中断服务程序的地址，中断的标志 flags 以及设备名和设备 ID 等，这个结构的定义在< linux/interrupt.h >中。

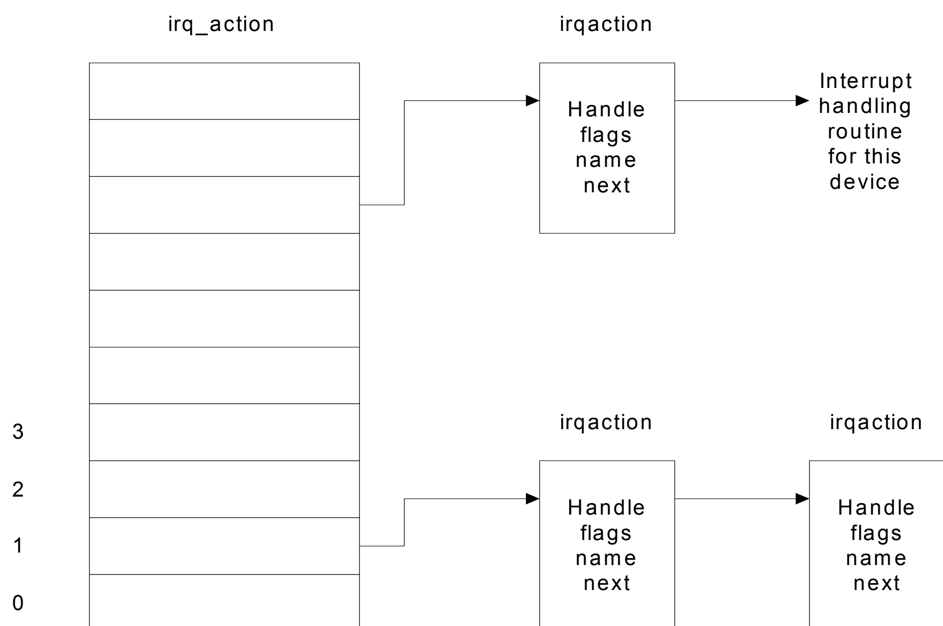


图 6.2 中断函数处理向量表示意图

当系统检测到中断的时候， linux 必须首先读取可编程中断控制器的状态寄存器来确定该中断的来源。然后把这个来源转换成 irq_action 向量表中的偏移。找到了这个偏移，就等于找到和这个中断号对应的中断处理函数信息，然后调用这个中断号的所有的 irqaction 数据结构中的中断处理例程。如果发生的中断没有对应的中断处理程序，系统就会记录下一个错误。

在每一个设备的中断处理程序中，首先要根据中断状态寄存器来判断产生中断的原因，比如是发生了错误还是完成了一个请求操作等。在确定了原因之后，设备驱动程序可能还需要做更多的工作，比如将中断分为“上半部”、“下半部”两部分，将比较耗时的工作放

入“下半部”处理等。关于“上半部”和“下半部”的原理详见下面 6.3.2 中断处理的实现部分。

在系统中多个设备共享一个中断号情况下，它们的中断处理函数会在其中一个设备产生中断时被全部调用。所以每一个中断处理函数，还应该有处理不是它本身产生中断而被调用的能力。

6.3.2. 实现中断处理程序

在我们知道中断处理的框架后接下来的任务就是根据实际任务实现中断处理程序的具体功能。一般的中断处理程序主要任务是唤醒那些在设备上睡眠的进程，告诉它们进入运行态的条件已经具备，使之进行相应的处理。具体的实现见 6.4 的应用实例。

中断处理的一个主要特点是他们必须在中断时间内运行，这使得它的行为受到些限制。一般在中断产生时，系统都要暂时关闭其它中断，如果该中断是快速中断，它可以在很短的时间内完成，这对其它的中断影响很小。但对于那些耗时很多的中断该怎么处理呢?在现代操作系统中处理这种情况主要是将一个中断处理分离成"上半部"和"下半部"两个阶段。“上半部”在屏蔽中断的上下文中运行，用于完成关键性的处理动作，它就是在 request_irq 注册函数中注册的 handle 例程。“下半部”主要是处理那些相对来说并不是非常紧急的，比较耗时的任务，所以“下半部”的处理都是在中断返回后由系统调度的，不在中断服务上下文中执行。“下半部”主要有一个函数指针数组和一个位掩码组成。当内核准备处理异步事件时，他就调用 do_bottom_half，当中断处理程序需要运行下半部处理时，只要调度 mark_bh 即可，该函数设置了掩码变量的一个位，用来将相应下半部处理函数注册到执行对列。和下半部处理相关的函数定义如下：

void mark_bh(int nr);

参数：int nr 表示指向要激活的 bh 的号码，它是在头文件<linux/interrupt.h>中定义的符号常数，它标记位掩码中要设置的位。每个下半部 bh 相应的处理函数由拥有它的驱动程序提供。

在静态模式下，下半部的管理是通过以下函数实现的：

static void (*bh_base[32])(void);

该函数的代码在< kernel/softirq.c >中，它定义了一个由 32 个函数指针组成的数组，采用索引方式来访问。

void init_bh(int nr,void (*routine)(void));

该函数为第 nr 个函数指针赋值为 routine。

void remove_bh(int nr);

它的动作与 init_bh()相反，卸下 nr 函数指针。

bh_base 32 个函数指针数组中很多位置都被系统使用了，如果我们要注册一个自己的下半部处理函数，就必须首先查询<include/linux/interrupt.h>，从中选择一个空着的位置。

6.4. 应用实例

uClinux 是一个开源项目，世界各地的爱好者都参与了开发，从而促进整个 uClinux 家族的不断壮大和发展。作为使用者来讲，如果可以找到已经存在的且比较成熟的驱动程序，就没有必要费力气去自己开发了。在选择硬件平台时，也可以根据现有的驱动程序，选择外围设备。比如，可以到 linux/drivers/目录下看看都有哪些驱动已经发布了，这些都是编制的非常成功，并被很多人使用验证的程序。

但从学习的角度来讲，找一个比较简单的驱动读一读，对自己的提高还是很有好处的。所以，在这一节中我们将通过几个具体的例子，让大家切身体会一下设备驱动程序的各个细节是如何实现的。

6.4.1. 字符型设备

按键和触摸屏是两种比较典型的字符设备。在本节中，我们将以这两种设备的驱动程序为例做进一步的学习。

6.4.1.1. 按键

简单起见，我们以一个按键的实现（图 6.3.）为例来讲述驱动的编写。平台上的一个按键和外部中断 IRQ6 相连，当按键按下时，引脚输入低电平触发中断。下面我们来看驱动程序的设计。

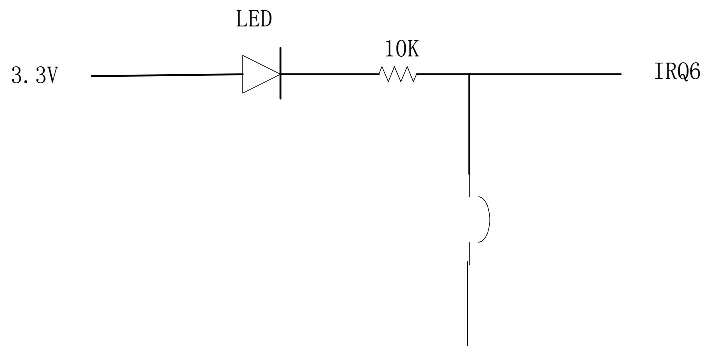


图 6.3　按键的硬件连接

- **设备初始化**

驱动程序在 init_keyboard()中实现向系统注册主次设备号，设备名，并初始化寄存器，如下：

```
void init_dev_set(void)
{
    ICR = 0x00;                //低电平触发中断
    PDDIR = 0x80;              //设置 PD7 为输入
    PDSEL = 0x80;              //PD7 作为 I/O 与外部连接
    PDKBEN = 0x00;             //键盘中断使能
```

```
int init_keyboard(void)
{
#define keyboard_major    50    //手动分配主设备号为 50
#define keyboard_minor    0  //次设备号为 0
          int rc;
          rc=register_chrdev(keyboard_major, " keyboard", & keyboard_fops);
                                                            //向系统注册字符设备
                    if(rc<0)                                  //register_chrdev()的返回值小于于零，注册失败
printk("Panic! Could not register keyboard-Driver\n");
else
        init_dev_set();
          return rc;
      };
```

- **注册中断和中断处理程序**

在 open 函数中向内核注册中断，如下：

```
static int keyboard_open(struct inode *inode, struct file *file)
{
rc=request_irq(IRQ_MACHSPEC|IRQ6_IRQ_NUM,   keyboard_interrupt,   IRQ_FLG_STD,   "
keyboard-IRQ", NULL/*Userdata!!!*/);//向内核注册中断
      if(rc) //返回值不为零，则注册失败
      {
        printk("keyboard-Driver: Error while installing interrupt handler\n");
        return -ENODEV;
      };

      MOD_INC_USE_COUNT;
      return 0;
}

static void keyboard_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{                   //中断处理程序
      ISR |= (1<<18);
      wake_up_interruptible(&wq);                  /*唤醒队列*/
#ifdef DEBUG
      printk("I've woken up the process\n");
#endif
      return 0;
}
```

- **read 的实现**

```
static int keyboard_read(struct inode *inode, struct file *file, char *buffer, int size)
{
char * ch;
```

```
#ifdef DEBUG
        printk("I'm reading the device!\n");/*调用读函数时的调试信息*/
#endif
    interruptible_sleep_on(&wq);
#ifdef DEBUG
      printk("I'm wake up!\n");
#endif
    return 0;
}
```

6.4.1.2. 触摸屏

触摸屏是一种具有操作方便、直观、使用灵活等优点的新颖的计算机信息输入设备，随着各种信息终端在生活中的普及，它的使用也越来越广泛。

在本实验平台中使用的触摸屏控制器 ADS7843 是由 Burr-Brown 公司生产的 4 线电阻式触摸屏转换接口芯片。这款芯片，也是 uClinux 发布版本所支持的触摸屏的标准硬件型号。　触摸屏的工作原理可以参见第一章相关的内容。

- **ADS7843 的性能参数及引脚介绍**

ADS7843 是一个内置 12 位模数转换、低导通电阻模拟开关的串行接口芯片。供电电压 2.7~5 V，参考电压 VREF 为 1 V~+VCC，转换电压的输入范围为 0~ VREF，最高转换速率为 125 kHz。在 125kHz 吞吐速率和 2.7V 电压下的功耗为 750 μW，而在关闭模式下的功耗仅为 0.5 μW。因此，ADS7843 以其低功耗和高速率等特性，被广泛应用在采用电池供电的小型手持设备上。

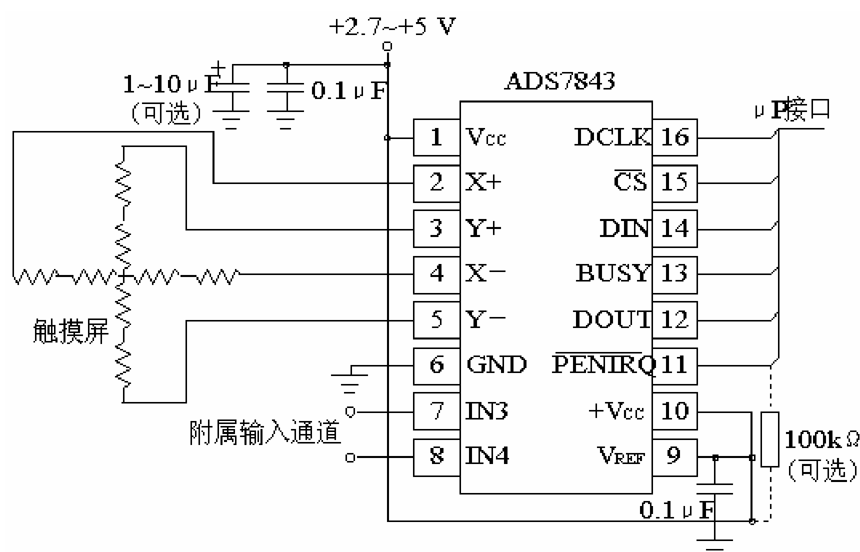


图 6.4　ADS7843 外围电路参考连接

ADS7843 采用 SSOP-16 引脚封装形式，温度范围是−40～+85°C 。它具有两个辅助输入（IN3、IN4），可设置为 8 位或 12 位模式。其外部连接电路如图 6.4 所示，该电路的

基准电压确定了转换器的输入范围。ADS7843 的主要引脚功能可参见表 6.2:

表 6.2　ADS7843 引脚功能说明

引脚号	引脚名	功 能 描 述
1,10	$+V_{CC}$	供电电源 2.7~5 V
2,3	X+,Y+	接触摸屏正电极，内部 A/D 通道
4,5	X−,Y−	接触摸屏负电极
6	GND	电源地
7,8	IN3,IN4	两个附属 A/D 输入通道
9	V_{REF}	A/D 参考电压输入
11	$\overline{PENIRQ}$	中断输出，须接外拉电阻（10 kΩ或 100 kΩ）
12,14,16	DOUT,DIN,DCLK	串行接口引脚，在时钟下降沿数据移出，上升沿移进
13	BUSY	忙指示，低电平有效
15	$\overline{CS}$	片选

- **ADS7843 控制字**

为了完成一次电极电压切换和 A/D 转换，需要先通过串口往 ADS7843 发送控制字，转换完成后再通过串口读出电压转换值。标准的一次转换需要 24 个时钟周期，如图 6.5 所示。由于串口支持双向同时进行传送，并且在一次读数与下一次发控制字之间可以重叠，所以转换速率可以提高到每次 16 个时钟周期，如图 6.6 所示。如果条件允许，CPU 可以产生 15 个 CLK 的话（比如 FPGAs 和 ASICs）,转换速率还可以提高到每次 15 个时钟周期，如图 6.7 所示。

表 6.3　ADS7843 控制字

bit7(MSB)	bit6	bit5	bit4	bit3	bit2	bit1	bit0
S	A2	A1	A0	MODE	SER/$\overline{DFR}$	PD1	PD0

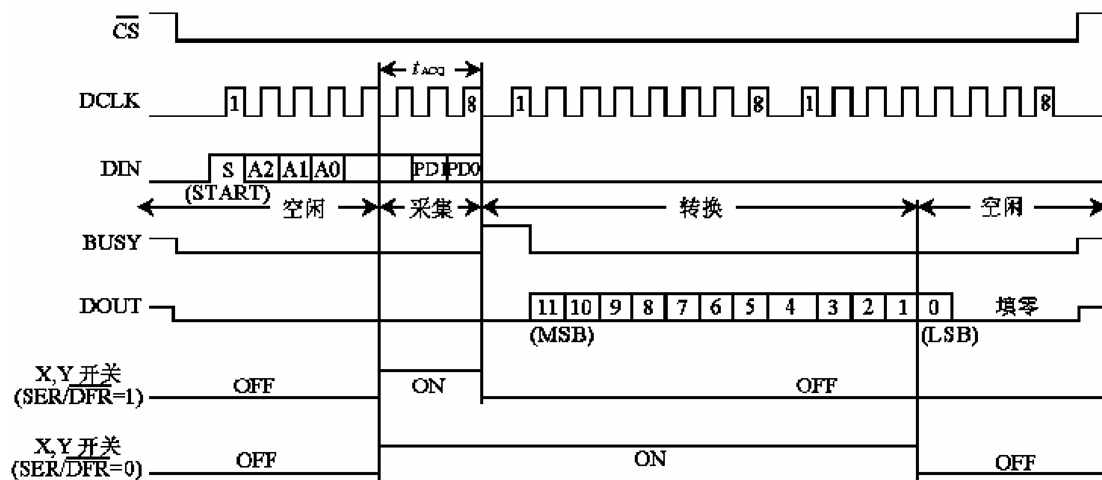


图 6.5　A/D 转换时序（每次转换需要 24 个时钟周期）

其中 S 为数据传输起始标志位，该位必为"1"。A2~A0 进行通道选择，MODE 用来选择 A/D 转换的精度，"1"选择 8 位，"0"选择 12 位。SER/选择参考电压的输入模式。PD1、

PD0 选择省电模式："00"省电模式允许，在两次 A/D 转换之间掉电，且中断允许；"01"同"00"，只是不允许中断；"10"保留；"11"禁止省电模式。

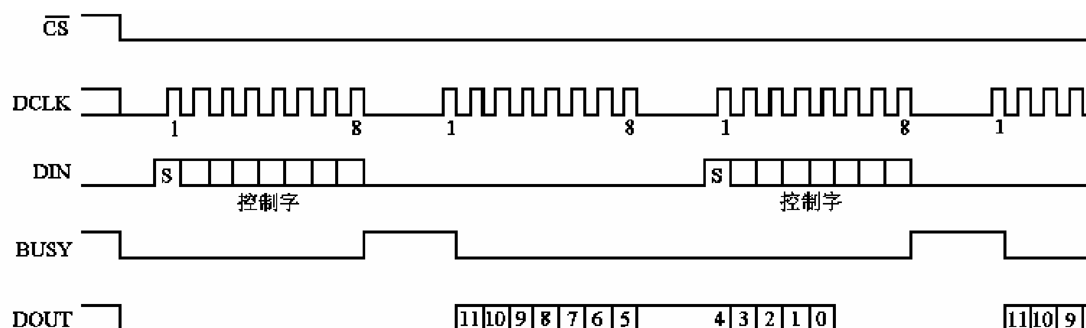


图 6.6　A/D 转换时序（每次转换需要 16 个时钟周期）

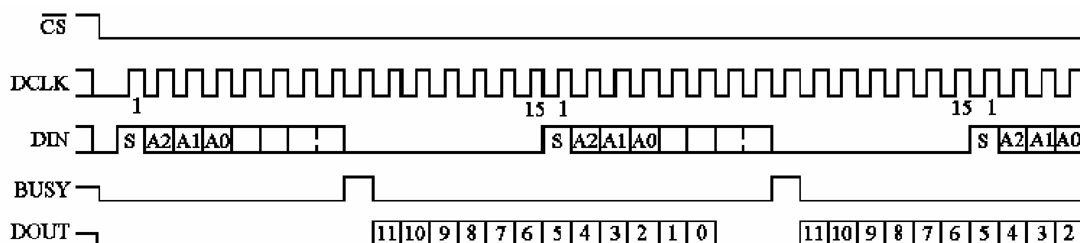


图 6.7　A/D 转换时序（每次转换需要 15 个时钟周期）

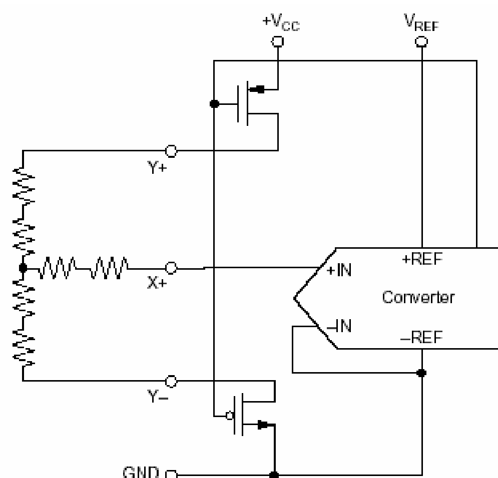


图 6.8　　单端模式下的简化电路

- **ADS7843 模式设置**

ADS7843 有差分（defferential）和单端（single-ended mode）两种工作模式。这两种模式对转换后的精度和可靠性有一些影响，如果将 A/D 转换器配置为读绝对电压（单端模式）方式，那么驱动电压的下降将导致转换输入数据的错误。而如果配置为差分模式，则可以避免上述错误。当触摸屏被按下时，有两种情况可影响接触点的电压：一种是当触摸到显示屏时，会导致触摸屏外层振动；另一种是触摸屏顶层和底层之间的寄生电容引起

的电流振荡以及在 ADS7843 输入引脚上引起的电压振荡。这两种情况都可导致触摸屏上的电压发生振荡以及增加 DC 值稳定的时间。

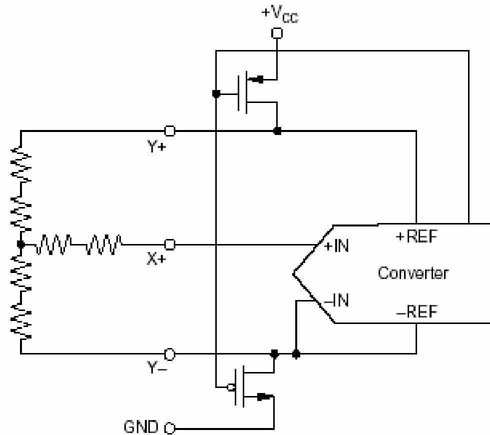


图 6.9　差分模式下的简化电路

在单端模式中，一旦在触摸屏上监测到一次触摸事件，电路系统将发送一串控制字节给 ADS7843，并要求它进行一次转化。然后 ADS7843 将在获取周期的起始点通过内部 FET 开关给面板提供电压，而这将导致触摸屏电压的升高。正如上面所介绍的，上升的电压在最终稳定之前会振荡一段时间。当获取周期结束后，所有的 FET 开关关闭，A/D 转换器进入转换周期。如果在转换周期期间，没有发出下一个控制字节，ADS7843 将进入低功耗模式并等待下一条指令。由于面板上分布有大量电容，特别是滤波噪音，因此，应该注意设置好对应于 X 坐标或 Y 坐标上的电压。在单端模式中，输入电压必须在送控制字的最后三个时钟周期期间设置，否则将产生错误。

除了内部 FET 开关从获取周期开始到转换周期结束期间一直保持打开状态以外，差分模式的操作类似于单端模式。加在面板上的电压将成为 A/D 转换器的基准电压，提供一个度量比操作。这意味着如果加在面板上的电压发生变化（由于电源、驱动电阻、温度或触摸屏电阻等原因），A/D 转换器的度量比操作将对这种变化进行补偿。如果在当前转换周期发向 ADS7843 的下一个控制字节所选择的通道与前一个控制字节相同，那么在当前转换完成后开关仍然不会关闭。

在这两种模式中，ADS7843 只有 3 个时钟周期可以从触摸屏上获取（取样）输入模拟电压，因此，为了 ADS7843 可以获取正确的电压，输入电压必须在 3 个时钟周期的时间范围内设置好。基于这种特性，如图 6.10 的软件设计就是消除触摸屏信号抖动的一种方法。对面板的点击将引起触摸屏的电压快速升高到最终值，为了得到正确的转换数据，电压采样必须在触摸屏完全设置好时完成。获取电压值的方式有两种：一是采用单端模式，即采用相对较慢的时钟扩展获取时间（三个时钟周期）；二是采用差分模式，即用相对较快的时钟在第一个转换周期内设置电压，在第二个周期获取准确电压。该方式的两个控制字节相同，且内部 X/Y 开关在首次转换后不会关闭。由于首次转换期间电压还不稳定，因此应当丢弃首次转换的结果。使用第二种方式的另一个优点是功耗低，因为在全部转换后，ADS7843 会进入低功耗模式来等待下一次取样周期：对于慢时钟，下一次取样可能在当前转换结束后立即进入采样周期，而没有时间进入低功耗模式。实际在单端模式下不能使用快速时钟。

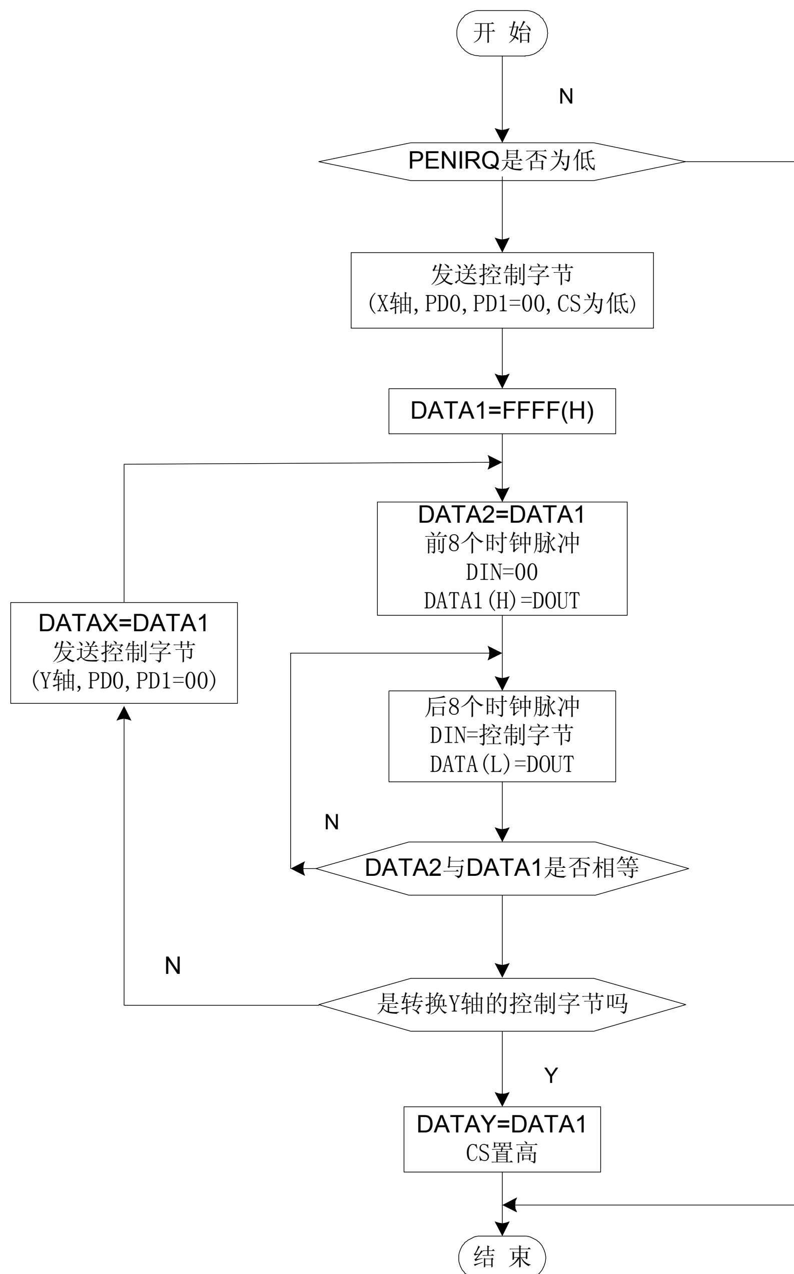


图 6.10 消除触摸屏信号抖动的一种方法

差分模式还具有以下两个优点：第一个优点是能够在不扩展转换器获取时间的条件下用很长的设置时间处理触摸屏，即触摸屏电压可以有足够的时间稳定下来。第二个优点是 ADS7843 通过快速时钟可以进入低功耗模式，从而可以节约电池能量。因此，通常建议使用差分模式。

- **PEN 中断引脚的使用**

PEN 中断引脚的主要作用是让设计者可以完全控制 ADS7843 的低功耗操作模式。图 6.11 所示是其模式操作连接示意图。

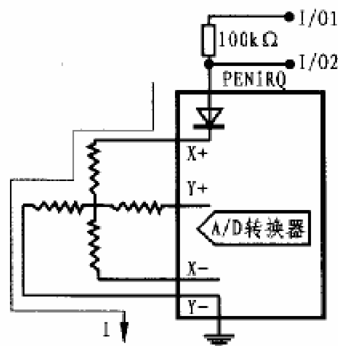


图 6.11 ADS7843（PD0，PD1=00）

图中，I/O 1 和 I/O 2 是引自 CPU 的通用目的输入/输出口。当电源加入系统且转换器被设置（PD1，PD0=00）之后，器件进入低功耗模式。而当未触摸面板时，ADS7843 内部的二极管没有偏压，因此没有电流流过（忽略漏流）；当触摸面板时，Y－将提供一条电流（I）通路，这时 X＋、X－和 Y＋处于高阻状态，电流经过 100 $k\Omega$ 电阻和中断二极管，PENIRQ 被拉低，从而通过 I/O 2 上一个不超过 0.65V 的电压唤醒 CPU，然后 CPU 再拉低 I/O 1 和 I/O 2 上的电位，同时对 ADS7843 控制寄存器写一个字节以进行转换初始化。为了转换 PENIRQ 二极管上的偏置电压，CPU 必须拉低 I/O 1 和 I/O 2 上的电压，否则，如果在转换期间二极管上有一个前向偏压，那么附加的电流将引起错误的输入数据。

- **ADS7843 与 CPU 的连接**

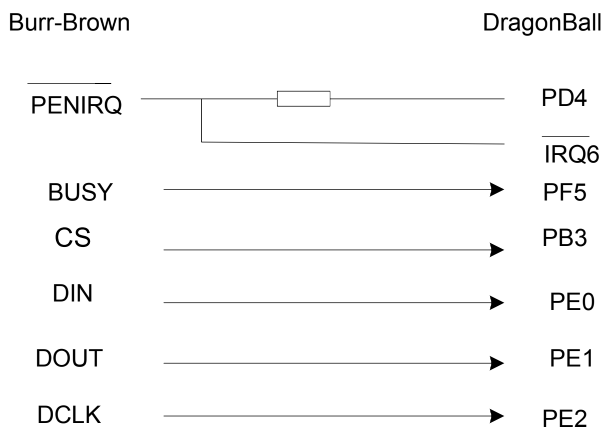


图 6.12 ADS7843 与 CPU 的连接

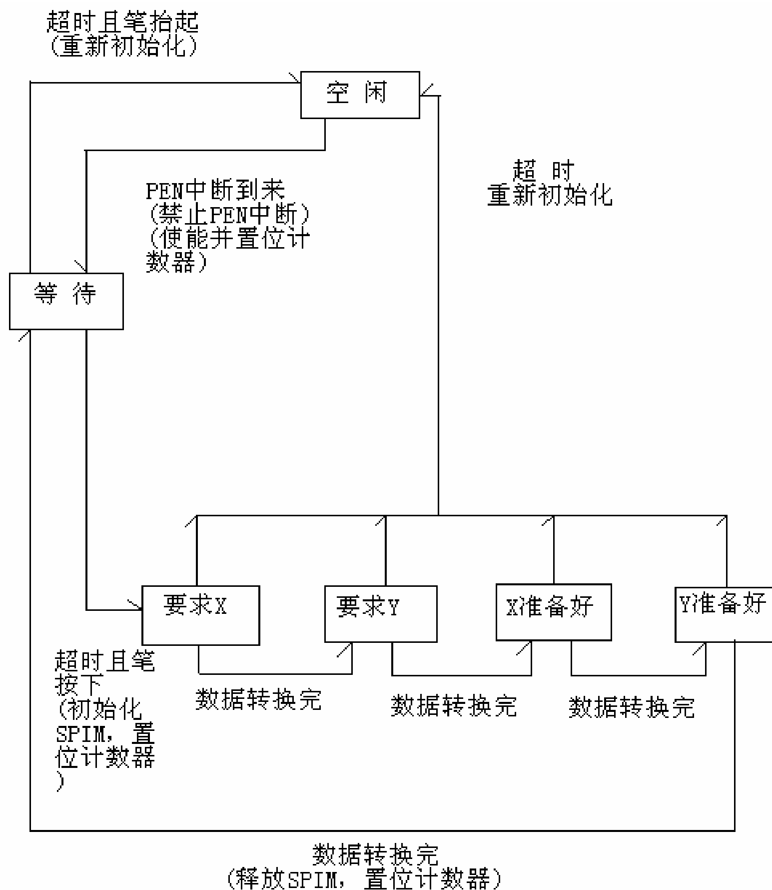


图 6.13 软件流程

- **触摸屏驱动程序**

在 uClinux 的发布版中已经集成了一个由 Philippe Ney 编写的触摸屏驱动程序，最新的版本是 1.5。经过众多 uClinux 爱好者的共同维护，这个驱动程序已经工作的非常成熟和稳定了,并且具有很好的版本兼容性。

该程序的流程如下图 6.13.

驱动程序为触摸屏定义了 7 个不同的状态，分别表征流程图中的各个不同阶段：

```
#define TS_DRV_ERROR       -1
#define TS_DRV_IDLE         0
#define TS_DRV_WAIT         1
#define TS_DRV_ASKX         2
#define TS_DRV_ASKY         3
#define TS_DRV_READX        4
#define TS_DRV_READY        5
```

下面我们将就驱动程序的关键部分进行讲解。

1） 初始化操作

```
int mc68328digi_init(void) {
  int err;
  printk("%s: MC68328DIGI touch screen driver\n",__FILE__);
```

```
    /* Register the misc device driver */
    err = misc_register(&mc68328_digi);
    if(err<0)
      printk("%s: Error registering the device\n", __FILE__);
    else
      printk("%s: Device register with name: %s and number: %d %d\n",
       __FILE__, mc68328_digi.name,10,mc68328_digi.minor);
    /* Init prameters settings at boot time */
    init_ts_settings();
    printk("setting is successful!\n");
    return err;     /* A non zero value means that init_module failed */
  }
```

驱动程序通过 mc68328digi_init()这个函数，在系统启动时向内核注册自己，并进行相关的硬件设置。

其中用到的 Misc 设备是一种混杂型设备名，它并不是一种单一的设备形式，而是系统为了方便驱动程序的扩展而设立的。凡是用户想要添加的硬件设备，都可以隶属于这个设备名。

Misc 的主设备号已经由系统分配为 10，次设备号需要用户指定。在我们讨论的驱动中次设备号指定为 9。

程序用一个数据结构 current_params 来描述触摸屏的各个参数值，在 init_ts_settings()中对该数据结构进行了初始化。

2) 打开操作

驱动程序在 ts_open()中向内核注册了两个中断，分别是触摸屏外部触发中断和与外设数据交换中断。调用的函数如下：

```
request_irq(IRQ_MACHSPEC | IRQ5_IRQ_NUM,handle_pen_irq,
            IRQ_FLG_STD,"touch_screen",NULL);
request_irq(IRQ_MACHSPEC | SPI_IRQ_NUM,(void *)handle_spi_irq,
            IRQ_FLG_STD,"spi_irq",NULL);
```

驱动程序在 init_ts_drv()中对触摸屏的状态值，计时器，以及各个寄存器值做了初始化。

3) 读函数

系统调用 ts_read()函数，如果没有数据可读，则用 interruptible_sleep_on(&queue->proc_list)阻塞进程。此时，程序进入等待状态，直到外部有中断到来。

一旦有中断到来，驱动程序在 ts_select()中判断中断源，并选择相应的等待队列进行处理。对于获取的数据，驱动程序用 put_user()将其放入用户空间。

4) I/O 操作

对于触摸屏状态参数的获取或者是设置相关参数，如采样间隔等，都可以通过调用 ts_ioctl()来完成。

在本函数中，根据用户传递的 cmd，分别获取、设置当前触摸屏参数，关键代码如下：

```
switch(cmd) {
```

```
  case TS_PARAMS_GET:  /* 获取内部参数，首先检查用户空间是否可写*/
    err = verify_area(VERIFY_WRITE,(char *)arg,sizeof(current_params));
    if(err) return err;
    p_in  = (char *)&current_params;
    p_out = (char *)arg;
    for(i=0;i<sizeof(current_params);i++)
      put_user(p_in[i],p_out+i);
    return 0;
  case TS_PARAMS_SET:  /* 设置内部参数，首先检查用户空间是否可读*/
    err = verify_area(VERIFY_READ,(char *)arg,sizeof(new_params));
    if(err) {
      return err;
    }
    /* ok */
    p_in  = (char *)&new_params;
    p_out = (char *)arg;
    for(i=0;i<sizeof(new_params);i++) {
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0))
      get_user(p_in[i],p_out+i);
#else
      p_in[i] = get_user(p_out+i);
#endif
    }
```

通过该函数，用户可以方便的获取和调整触摸屏的各个工作参数，以适应不同的应用场合。

5) 获取位置坐标

读取触摸屏此时触摸点位置，设计思路可参见软件流程图 6.13：

```
switch(ts_drv_state) {

  case TS_DRV_ASKX:
    if(IS_BUSY_ENDED) {     /* 假如忙信号释放，则 */
      ask_x_conv();         /* 继续进行转换*/
      ts_drv_state++;
    }
    else  fall_BUSY_enable_PENIRQ();  /* 否则，再次循环  */
    break;

  case TS_DRV_ASKY:
    ask_y_conv();
    ts_drv_state++;
    break;

  case TS_DRV_READX:
    read_x_conv();
```

```
    ts_drv_state++;
    break;

  case TS_DRV_READY:
    read_y_conv();
    swap_xy(&current_pos.x, &current_pos.y);
    ts_drv_state = TS_DRV_WAIT;
    break;

  case TS_DRV_WAIT:
    DISABLE_SPIM_IRQ;
    release_SPIM_transfert();
    toggle_PEN_IRQ_2_dedicated();
    set_timer_irq(&ts_wake_time,sample_ticks);
    break;

  case TS_DRV_ERROR:
    if(IS_BUSY_ENDED) {              /* 如果忙信号释放 */
      release_SPIM_transfert();
      cause_event(CONV_ERROR);
      init_ts_state();
    }
    else fall_BUSY_enable_PENIRQ();  /* 否则，再次循环*/
    break;

  default:
    init_ts_state();
  }
```

限于篇幅，其他细节不过多介绍，感兴趣的读者可以对照源程序进一步研究。

6.4.2. 网络设备

从整体角度考虑，linux 网络子系统可以分为硬件层、设备驱动层、网络协议层和应用层。可以看出，它的实现也采用了分层的思想。其中网络协议层得到的数据包通过设备驱动的发送函数被发送到具体的通信设备上，通信设备传来的数据也在设备驱动程序的接收函数中被解析并组成相应的数据包传给网络协议层。要实现一个网络设备驱动程序的主要工作只是根据具体的硬件设备向它的高层提供服务而已，这和字符设备、块设备的思路都是一样的。

6.4.2.1. 网络驱动的框架

Linux 的设计者们为了简化物理网络设备的多样性，对所有的设备进行了抽象并定义了一个统一的接口。所有对网络硬件的访问都是通过这一接口进行的，接口为所有类型的

硬件提供了一个一致化的操作集合。任意一个网络接口均可看成一个发送和接收数据包的实体。在 linux 中这个统一的接口就是 device 结构，它操作的数据对象——数据包是通过结构 sk_buff 来封装的。整个网络设备驱动程序工作原理如图 6.14 所示:

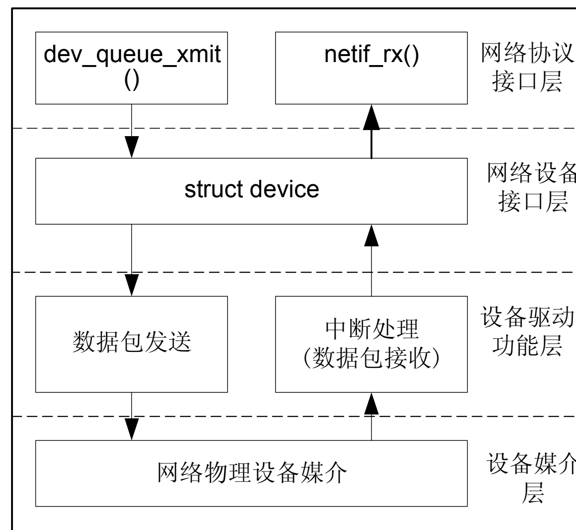


图 6.14 Linux 网络驱动程序体系结构

在 Linux/uclinux 中，整个网络接口驱动程序的框架可分为四层，从上到下分别为协议接口层、网络设备接口层、提供实际功能的设备驱动功能层、以及网络设备和网络媒介层。这个框架在内核网络模块中已经搭建好了，我们在设计网络驱动程序时，要做的主要工作就是根据上层网络设备接口层定义的 device 结构和底层具体的硬件特性，完成设备驱动的功能。在 linux 中网络设备接口层是 linux 的设计者们对所有网络设备的一个抽象，它提供了对所有网络设备的操作集合，所谓的网络设备接口，它既包括纯软件网络设备接口，如环路（loopback），也可以包括硬件网络设备接口，如以太网卡。在 linux 中这个接口是由数据结构 struct device 来表示的。数据结构 device 是整个框架的中枢，它定义了很多供系统访问和协议层调用的设备标准的方法，包括供设备初始化和往系统注册用的 init 函数，打开和关闭网络设备的 open 和 stop 函数，处理数据包发送的函数 hard_start_xmit，以及中断处理函数等，接口状态统计函数等。

在网络驱动程序部分主要有两个数据结构，一个是 sk_buff,TCP/IP 中不同协议层间以及和网络驱动程序之间数据包的传递都是通过这个结构体来完成的，这个结构体主要包括传输层、网络层、连接层需要的变量，决定数据区位置和大小的指针，以及发送接收数据包所用到的具体设备信息等。它的详细定义可以参阅内核源代码<include/linux/skbuff.h>。

另一个就是上面主要提到的 device 结构，它的定义在<include/linux/netdevice.h>中。这个结构是网络驱动程序的核心，它定义了封装所有网络设备接口所需要的信息，下面将结合 linux/uclinux2.0 内核介绍它主要的成员变量和方法，并作简单的分析。在 linux/uclinux 2.4 的内核中增加了一些成员变量，但是整体框架还是一样的，具体可以参照他们的源代码。

整个 device 在结构上可以分为两个部分：“可见的”和“不可见的”。可见部分主要是在 space.c 使用的部分，它由那些在静态 device 结构中显式赋值的域组成。不可见的部分则是在网络子系统内部使用，而且是可以改变的的部分。

```
struct device
{
```

```
/************************可见部分***********************/
char                    name;           /*device' name          */

unsigned long           rmem_end;       /* shmem "recv" end     */
unsigned long           rmem_start;     /* shmem "recv" start   */
unsigned long           mem_end;        /* shared mem end       */
unsigned long           mem_start;      /* shared mem start     */
unsigned long           base_addr;      /* device I/O address   */
unsigned char           irq;            /* device IRQ number    */

volatile unsigned char  start,          /* start an operation   */
volatile unsigned char  interrupt;      /* interrupt arrived    */
unsigned long           tbusy;          /* transmitter busy must be long for bitops */
struct device           *next;

int         (*init)(struct device *dev);    /* init function        */

unsigned char           if_port;        /* Selectable AUI, TP,..  */
unsigned char           dma;            /* DMA channel          */

struct enet_statistics* (*get_stats)(struct device *dev);

/************************不可见部分***********************/
unsigned long           trans_start;    /* Time (in jiffies) of last Tx     */
unsigned long           last_rx;        /* Time of last Rx                  */
unsigned short          flags;          /* interface flags (a la BSD)       */
unsigned short          family;         /* address family ID (AF_INET)*/
unsigned short          metric;         /* routing metric (not used)        */
unsigned short          mtu;            /* interface MTU value              */
unsigned short          type;           /* interface hardware type          */
unsigned short          hard_header_len;/* hardware hdr length              */
void                    *priv;          /* pointer to private data          */

/*接口信息域*/
unsigned char           broadcast[MAX_ADDR_LEN];/* hw bcast add             */
unsigned char           pad;            /* make dev_addr aligned to 8 bytes */
unsigned char           dev_addr[MAX_ADDR_LEN]; /* hw address               */
unsigned char           addr_len;   /* hardware address length              */
unsigned long           pa_addr;    /* protocol address                     */
unsigned long           pa_brdaddr; /* protocol broadcast addr              */
unsigned long           pa_dstaddr; /* protocol P-P other side addr         */
unsigned long           pa_mask;    /* protocol netmask                     */
unsigned short          pa_alen;    /* protocol address length              */
```

```
    struct dev_mc_list        *mc_list;    /* Multicast mac addresses          */
    int                       mc_count;    /* Number of installed mcasts       */

    struct ip_mc_list         *ip_mc_list;     /* IP multicast filter chain      */
    __u32                     tx_queue_len;    /* Max frames per queue allowed   */

    unsigned long             pkt_queue;       /* Packets queued      */
    struct device             *slave;          /* Slave device        */
    struct net_alias_info     *alias_info;     /* main dev alias info */
    struct net_alias          *my_alias;       /* alias devs          */

    struct sk_buff_head       buffs[DEV_NUMBUFFS];

    int               (*open)(struct device *dev);
    int               (*stop)(struct device *dev);
    int               (*hard_start_xmit) (struct sk_buff *skb,
                                          struct device *dev);
    int               (*hard_header) (struct sk_buff *skb,
                                          struct device *dev,
                                          unsigned short type,
                                          void *daddr,
                                          void *saddr,
                                          unsigned len);
    int               (*rebuild_header)(void *eth, struct device *dev,
                                          unsigned long raddr, struct sk_buff *skb);
    void              (*set_multicast_list)(struct device *dev);
    int               (*set_mac_address)(struct device *dev, void *addr);
    int               (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
    int               (*set_config)(struct device *dev, struct ifmap *map);
    void              (*header_cache_bind)(struct hh_cache **hhp, struct device *dev,
                                          unsigned short htype, __u32 daddr);
    void              (*header_cache_update)(struct hh_cache *hh, struct device *dev,
                                          unsigned char *    haddr);
    int               (*change_mtu)(struct device *dev, int new_mtu);
    struct iw_statistics* (*get_wireless_stats)(struct device *dev);
};
```

6.4.2.2. 网卡驱动程序的加载方法

linux 目前网络设备驱动程序的加载有两种方式。一种是系统启动时，由内核自动检测并静态加载，我们称之为“启动初始化方式”，另一种是通过模块化机制在系统运行过程中根据需要由用户或系统进程以动态加载,我们称之为“模块初始化方式”。

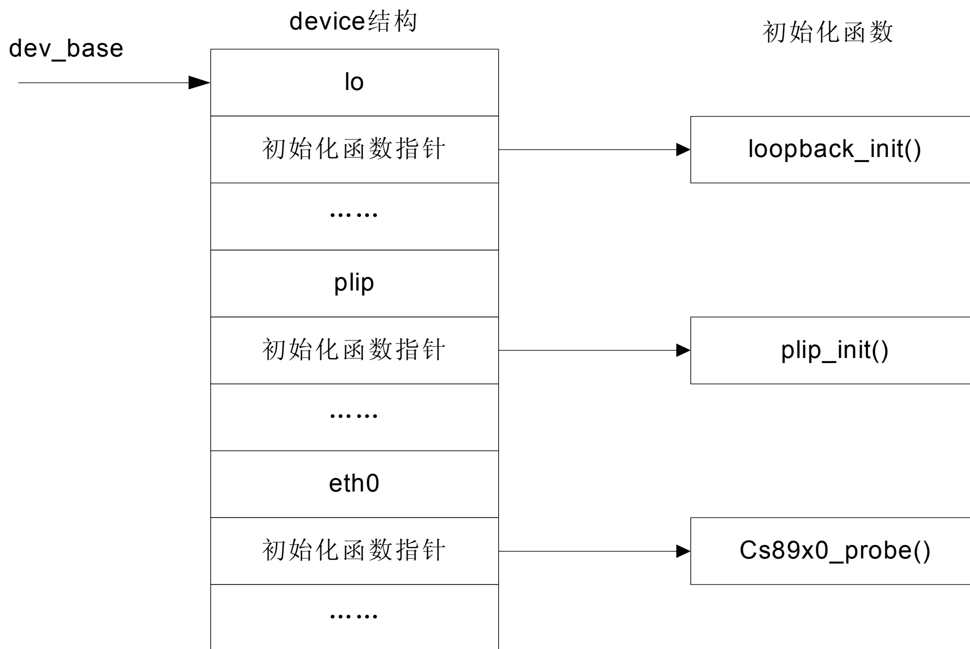


图 6.15 网络设备表初始化后示意图

这两种方法实现的途径虽然不一样，但最终的结果却是相同的。他们都是向内核的网络接口管理表(由指针 dev_base 指向的一个链表)加入一个 device 结构。dev_base 指针是系统运行过程中管理网络设备的接口，所有成功注册的网络设备都是这个链表的一个节点。因为在嵌入式系统中模块化动态加载的意义不是很大，所以以下的介绍中将偏重于前者。

- **启动初始化方式**

这种方式下，内核在被编译时已经将所有需要支持的网络设备接口注册到网络设备管理表 dev_base 上，在启动时根据 dev_base 头指针遍历整个网络设备管理表，调用每一个节点的 init()函数对相应的网络接口设备进行初始化，如果初始化失败，就从 dev_base 指向的链表中删除该节点。这样在内核启动后，网络设备管理表中剩下的接口都是存在的，而且都已经被正确初始化了。网络设备管理表初始化后如图 6.15 所示。

其中设备接口向网络设备管理表注册自己的代码是在</drivers/net/space.c>中，这在后面 cs8900A 芯片驱动程序的编译部分还将详细看到。

- **模块初始化方式**

模块初始化方式加载驱动是在 shell 命令 insmod 触发的，这和普通的模块化方法编程原理是一样，详细可见 6.1 节模块化编程。

6.4.2.3. CS8900A 芯片特点

CS8900A 芯片是一个高度集成的以太网控制器芯片，它集成了 ISA 总线接口，曼彻斯特编码/解码器，片上 RAM，10BASE－T 收发器，数据链路控制器 MAC，芯上存储管理器等，是嵌入式平台实现 10M 以太网连接的很好的选择方案。

要实现一个芯片的驱动，首先要对这个芯片的工作逻辑有很清楚的认识，但在这里不可能详细的介绍 CS8900A 芯片，所以如果要彻底的搞懂整个驱动的细节可以参阅 CS8900A 的 data sheet。以下仅从驱动程序框架方面介绍 CS8900A 芯片驱动的主要特点：

- EEPROM

如果不使用 cs8900A 芯片的默认设置，EEPROM 是必须要操作的，因为芯片的 MAC 地都存在这里，另外还有一些用户设置，比如工作模式等。

表 6.4 I/O 模式端口分配表

Offset	Type	Description
0000h	Read/Write	Receive/Transmit Data(Port 0)
0002h	Read/Write	Receive/Transmit Data(Port 1)
0004h	Write-only	TxCMD(Transmit Command)
0006h	Write-only	Txlength(Transmit Length)
0008h	Read-only	Interrupt Status Queue
000Ah	Read/Write	PacketPage Pointer
000Ch	Read/Write	PacketPage Data(Port 0)
000Eh	Read/Write	PacketPage Data(Port 1)

- 工作模式

CS8900A 有两种工作模式：MEMORY MODE 和 I/O MODE。其中 MEMORY MODE 下在编程操作上较为简单，对任何寄存器都是直接操作,不过这需要硬件上多根地址线和网卡连。 I/O MODE 则较为麻烦，因为这种模式下对任何寄存器操作均要通过 I/O PORT 0X300 写入或读出，但这种模式在硬件上实现比较方便，而且这也是芯片的默认模式，它的传输效率是 MEMORY MODE 的 96%左右，两者几乎是一样的，所以在 uclinux 中采用这种工作模式，下面主要介绍 I/O 模式下 CS8900A 芯片驱动程序的实现。

在 I/O 模式下，PacketPage memory 被映射到 CPU 的 16 个连续 I/O 端口上，也就是 8 个 16 位的 I/O 端口上。在芯片被加电后，I/O 基地址的缺省值被置为 300h，不过这在程序中是可以改变的。这 8 个 16 位 I/O 端口详细的功能和偏移地址如表 6.4 所示:

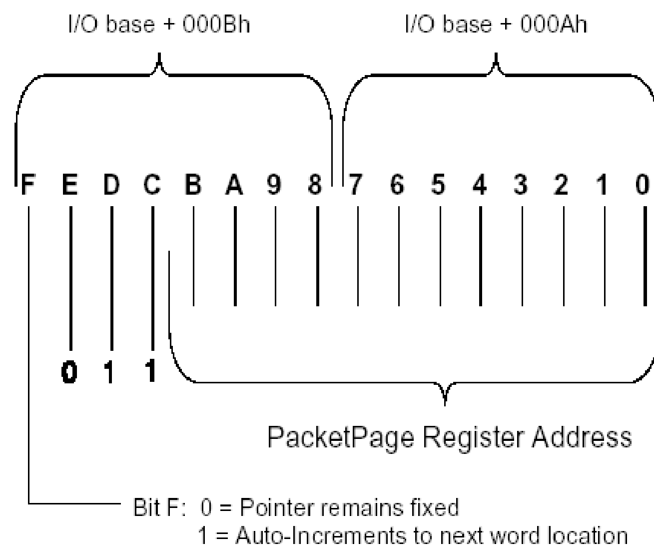


图 6.16 PacketPage Pointer 寄存器示意图

我们要访问 CS8900A 内部的寄存器中的任何一个，必须首先设置 PacketPage Pointer。该端口的低 12 位表示了我们要访问的内部寄存器的地址，接着的三个位（C，D 和 E）是不能改变的，我们只要把它们置为 011b 即可。它的最高位表示了我们要访问的是一个寄存器还是一组连续地址的块寄存器。详细如图 6.16 所示。

如果我们正确设置了 PacketPage pointer 的值，目标寄存器的内容然后被映射到 PacketPage Data 端口（I/O base + 000Ch）上。如果我们访问的是一组连续地址的块寄存器，PacketPage 指针会在本次访问结束后，自动的移动到下一个内部寄存器的位置。

- **endian模式**

endian 的意思是"字节排列顺序"，表示一个字或双字在内存中或传送过程中的字节顺序。一般情况下，我们是不需要关心字节排列顺序，但若要涉及跨平台之间的通信和资源共享，就不得不考虑这个问题了。因为在计算机的二进制系统中，字节排列顺序有两种情况：一种称为 big-endian，它把最高位字节放在最前面；另一种是 little-endian，它把最低位字节放在最前面。

CS8900A 是一个 little-endian 的 ISA 设备，但是一般的网络字节顺序采用 big-endian 模式，CS8900A 为了减少软件上的复杂度，在内部自动的进行了 byte-swaps 的处理，将网络字节转换为 little-endian 模式，这也是芯片的默认模式。但是 dragonball 系列都是 big-endian 的，所以在驱动上我们还需要将 little-endian 转换为 big-endian。所以在 uCcs8900.c 中可以看到以下代码：

```
#ifndef CONFIG_UCCS8900_HW_SWAP
#include <asm/io.h>
#else
#include <asm/io_hw_swap.h>
#endif
```

其中宏 CONFIG_UCCS8900_HW_SWAP 表示芯片是否进行 byte-swaps 处理。

6.4.2.4. CS8900A 芯片驱动程序的实现

- 初始化函数

网络设备的探测是在初始化函数里完成的。该函数唯一的参数是一个指向设备的指针，其返回值是 0 或者一个负的错误代码。在采用“启动初始化方式”加载驱动程序时，该函数在<drivers/net/Space.c>中被注册进内核，在 init 进程启动时被 net_dev_init()调用。一般情况下它的流程如下:

(1) 测设备是否存在；
(2) 检测中断号和 I/O 地址；
(3) 填充 device 结构大部分属性字段；
(4) 调用 ether_setup(dev)；
(5) 调用 kmalloc 申请需要的内存空间；

ether_setup 是一个通用的设置以太网接口的函数。由于以太网卡有很好的共性，device 结构中许多有关的网络接口信息都是通过调用 ether_setup 函数统一设置的。它会默认的设置一些字段，如果满足于这些默认的设置，那么可以调用这个函数即可。也可以在调用该函数之后再改动。

对于 CS8900A 来说，在 uclinux 中初始化函数是通过 cs89x0_probe()和 cs89x0_probe1()函数来实现的，以下是其部分代码,完整的代码在</drivers/net/uCcs8900.c>中。

```
int cs89x0_probe(struct device *dev)
{
    int base_addr = CS8900_BASE;
    return cs89x0_probe1(dev, base_addr);
}
```

其中 CS8900_BASE 在</drivers/net/uCcs8900.c>一开始时被定义为 0x10000300。它是 I/O 被映射到的基地址。

```
static int cs89x0_probe1(struct device *dev, int ioaddr)
{
    irq2dev_map[0] = dev;
    ……
    /* 初始化寄存器，建立片选和芯片工作方式*/
    *(volatile unsigned    char *)0xfffff42b |= 0x01;        /* output /sleep    */
    *(volatile unsigned short *)0xfffff428 |= 0x0101;      /* not sleeping     */
    *(volatile unsigned    char *)0xfffff42b &= ~0x02;    /* input irq5       */
    *(volatile unsigned short *)0xfffff428 &= ~0x0202; /* irq5 fcn on       */
    *(volatile unsigned short *)0xfffff102 = 0x8000;      /* 0x04000000      */
    *(volatile unsigned short *)0xfffff112 = 0x01e1;      /* 128k,2ws,FLASH, en */
    ……
    /* 初始化设备结构 */
    if (dev->priv == NULL) {
        dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
                memset(dev->priv, 0, sizeof(struct net_local));
        }
    dev->base_addr = ioaddr;
    lp = (struct net_local *)dev->priv;
     ……
    /* 取得芯片类型*/
    rev_type = readreg(dev, PRODUCT_ID_ADD);
    lp->chip_type = rev_type &~ REVISON_BITS;
    lp->chip_revision = ((rev_type & REVISON_BITS) >> 8) + 'A';
    lp->send_cmd = TX_AFTER_ALL;
    ……
    /*注册接口方法*/
    dev->open                  = net_open;
    dev->stop                  = net_close;
    dev->hard_start_xmit       = net_send_packet;
    dev->get_stats             = net_get_stats;
    dev->set_multicast_list    = &set_multicast_list;
    dev->set_mac_address       = &set_mac_address;
    ……
```

ether_setup(dev);
}

- **设备打开函数与关闭函数**

打开和关闭一个网络接口是由 *ifconfig* 命令来完成的。当使用 *ifconfig* 为一个接口赋地址时，它完成两项工作。第一，它通过 ioctl(SIOCSIFADDR)(即 Socket I/O Control Set InterFace ADDRess)来赋地址，接着它通过 ioctl(SIOCSIFFLAGS)(即 Socket I/O Control Set InterFace FLAGS) 对 dev->flag 中的 IFF_UP 置位来打开接口。

ioctl(SIOCSIFADDR)是和设备无关的，在它中仅设置 dev->pa_addr，dev->family，dev->pa_mask 和 dev->pa_brdaddr 四个域，没有驱动程序函数被调用。不过后一个命令 ioctl(SIOCSIFFLAGS)则调用设备的 *open* 函数。类似地，当一个接口关闭时，*ifconfig* 使用 ioctl(SIOCSIFFLAGS)来清除 IFF_UP，并调用驱动程序的 *stop* 函数。

设备驱动程序在 *Open* 函数中请求它需要的系统资源，并启动网络设备接口，如果驱动程序不准备使用共享中断，它还需要将 irq2dev_map 数组中对应的位赋为 1。*Stop* 则正好相反，它先关闭接口，释放系统资源，在不使用共享中断的情况下，将 irq2dev_map 数组中对应的位赋为 0。irq2dev_map 是一个很重要的数组阵列，它由 IRQ 号寻址，驱动程序正是利用这个数组将中断号映射到自己的 device 结构指针上，这是在不使用接口处理程序的情况下，一个驱动程序支持一个以上接口的唯一方法。

CS8900A 是一个 ISA 设备，他不支持共享中断，但是这在嵌入式系统外设比较少的情况下对系统的性能没什么影响。另外，在接口可以和外界通信以前，我们还需要将芯片上的硬件地址复制到 dev->dev_addr 指针指向的空间上，不过这个工作也可以在初始化函数 cs89x0_probe1 中完成。

一般情况下设备打开函数 net_open 的基本流程如下：

(1) 没有在初始化函数中注册中断号和 I/O 地址，则在设备打开时要进行注册；
(2) 将该设备挂到 irq2dev_map 中。若使用基于中断的数据接收方式，以后就可以通过中断号和 irq2dev_map 数组直接查找相应的设备了；
(3) 初始化物理设备的寄存器；
(4) 设置接口相应的 dev 的私有数据结构(dev->priv)中的一些字段；
(5) 设置 dev 中的 tbusy，interrupt 和 start 等字段；

设备关闭函数 net_close()与打开函数动作恰好相反，不再赘述。

- **数据包发送函数**

当系统需要发送数据时，它首先把数据打包成一个完整的sk_buff结构体，然后调用 hard_start_transmit()函数把它发送到网络设备接口上。一般的网络接口芯片传输数据包的流程如下：

(1) 通过标志位 tbusy 判断上次数据包的传输是否完成。若 tbusy＝0 就跳转到下一步，否则看上次传输是否已超时，若未超时就以不成功返回，若已超时，则初始化芯片寄存器，置 tbusy＝0，然后继续下一步；
(2) 将 tbusy 标志位打开；
(3) 将数据包传给硬件让它发送；
(4) 释放缓存区 skb；
(5) 修改接口的一些统计信息；

在CS8900A芯片的I/O模式下，数据包的发送流程如下：

(1) 发送一个传输命令到 TxCMD 端口（I/O base + 0004h），使芯片进入发送状态；
(2) 将要发送数据帧的长度发送到 TxLength 端口 (I/O base + 0006h)；

(3) 通过 PacketPage pointer 端口读取 BusST 寄存器（寄存器 18），判断 Rdy4TxNOW 位（第 8 位）的值。如果 Rdy4TxNOW 值为 1，则跳到第 4 步；如果 Rdy4TxNOW 位的值为 0，驱动程序将等待一段时间，再判断 Rdy4TxNOW 的值，直到它为 1 为止。另外，如果程序中 Rdy4TxiE(寄存器 BufCFG 的第 8 位)被置为 1，当 CS8900A 的发送缓冲区可写时，Rdy4Tx (寄存器 BufEvent 的第 8 位)将被置为 1，并触发一个中断，这时候就不需要判断 Rdy4TxNOW 了。

(4) 程序发送函数将反复执行写指令，将数据发送到接收/发送数据端口（(I/O base + 0000h)。

结合上面两个流程，下面介绍uClinux中数据包发送函数的具体实现:

```
static int net_send_packet(struct sk_buff *skb, struct device *dev)
{    /*判断 tbusy 标志*/
     if (dev->tbusy) {
          int tickssofar = jiffies - dev->trans_start;
          if (tickssofar < 5)
               return 1;
          if (net_debug > 0) printk("%s: transmit timed out, %s?\n", dev->name,
                    tx_done(dev) ? "IRQ conflict" : "network cable problem");
          /* Try to restart the adaptor. */
          dev->tbusy=0;
          dev->trans_start = jiffies;
     }

     /*判断发送数据包*/
     if (skb == NULL) {
          dev_tint(dev);
          return 0;
     }

     if (set_bit(0, (void*)&dev->tbusy) != 0)
          printk("%s: Transmitter access conflict.\n", dev->name);
     else {
          struct net_local *lp = (struct net_local *)dev->priv;
          unsigned long ioaddr = dev->base_addr;
          unsigned long flags;

          save_flags(flags);
          cli();

          outw(lp->send_cmd, ioaddr + TX_CMD_PORT);
          outw(skb->len, ioaddr + TX_LEN_PORT);

          if ((readreg(dev, PP_BusST) & READY_FOR_TX_NOW) == 0) {
               restore_flags(flags);
               printk("cs8900 did not allocate memory for tx!\n");
```

```
            return 1;
        }
        outsw(ioaddr + TX_FRAME_PORT,skb->data,(skb->len+1) >>1);
        restore_flags(flags);
        dev->trans_start = jiffies;
    }
    dev_kfree_skb (skb, FREE_WRITE);
    return 0;
}
```

- 中断处理函数

目前几乎所有的网络设备接口都是以中断方式工作的，接口触发中断表明两种事件中的一种发生了：一个新包到达或一个包发送完成。中断例程可以通过检查硬件设备上的中断状态寄存器来判断是什么事件触发了中断。

中断处理程序中对于“发送完成”事件的处理首先将 dev->tbusy 值清为 0，然后调用网络下半部函数 *net_bh。如果*网络下半部函数 net_bh 真的运行了，它就会试图发送所有等待的数据包。

另一方面，中断处理程序中对于新数据包到达事件的处理也不是很复杂，它只需要调用数据包接收子函数 net_rx()就行了。

实际上，当 *netif_rx* 被接收函数调用时，它所进行的实际操作只有标志 *net_bh*。换句话说，核心在一个下半部处理程序中完成了所有网络相关的工作。一般的中断服务程序的基本流程如下：

(1) 定发生中断的具体网络接口
(2) 打开标志位 dev->interrupt，表示本服务程序正在被使用；
(3) 读取中断状态寄存器，根据寄存器判断中断发生的原因。有两种可能，一种是有新数据包到达；另一种是上次的数据传输已完成。
(4) 若是因为有新数据包到达，则调用接收数据包的子函数 net_rx()；
(5) 如果中断是上次传输引起，则通知协议的上一层，修改接口的统计信息，关闭标志位 tbusy 为下次传输做准备；
(6) 关闭标志位 interrupt。

}

uclinux 中 CS8900A 驱动程序的中断处理函数的实现代码如下:

```
void cs8900_interrupt(int irq, void *dev_id, struct pt_regs * regs)
{
    struct device *dev = (struct device *)(irq2dev_map[/* FIXME */0]);
    struct net_local *lp;
    int ioaddr, status;

    dev = irq2dev_map[0];
    dev->interrupt = 1;
    ioaddr = dev->base_addr;
    lp = (struct net_local *)dev->priv;

    while ((status = readword(dev, ISQ_PORT))) {
```

```
        /*判断中断类型*/
        switch(status & ISQ_EVENT_MASK) {
        case ISQ_RECEIVER_EVENT:
            /* Got a packet(s). */
            net_rx(dev);
            break;
        case ISQ_TRANSMITTER_EVENT:
            lp->stats.tx_packets++;
            dev->tbusy = 0;
            mark_bh(NET_BH);     /* Inform upper layers. */
            if ((status & TX_OK) == 0) lp->stats.tx_errors++;
            if (status & TX_LOST_CRS) lp->stats.tx_carrier_errors++;
            if (status & TX_SQE_ERROR) lp->stats.tx_heartbeat_errors++;
            if (status & TX_LATE_COL) lp->stats.tx_window_errors++;
            if (status & TX_16_COL) lp->stats.tx_aborted_errors++;
            break;
        case ISQ_BUFFER_EVENT:
            if (status & READY_FOR_TX) {
                dev->tbusy = 0;
                mark_bh(NET_BH);     /* Inform upper layers. */
            }
            if (status & TX_UNDERRUN) {
                lp->send_underrun++;
                if (lp->send_underrun > 3) lp->send_cmd = TX_AFTER_ALL;
            }
            break;
        case ISQ_RX_MISS_EVENT:
            lp->stats.rx_missed_errors += (status >>6);
            break;
        case ISQ_TX_COL_EVENT:
            lp->stats.collisions += (status >>6);
            break;
        }
    }
    dev->interrupt = 0;
    return;
}
```

- **数据包接收函数 net_rx()**

从上面可以看到，在网络上有新数据包到达时，中断处理函数仅仅调用数据包接收子函数 net_rx()即可。一般情况下 net_rx 函数的操作流程如下：

(1) 申请 skb 缓存区存储新的数据包；
(2) 从硬件中读取新到达的数据；
(3) 调用函数 netif_rx()，将新的数据包向网络协议的上一层传送；

(4) 修改接口的统计函数。

相应的在 uClinux 中的代码如下:

```
static void net_rx(struct device *dev)
{
	struct net_local *lp = (struct net_local *)dev->priv;
	int ioaddr = dev->base_addr;
	struct sk_buff *skb;
	int status, length;

	status = inw(ioaddr + RX_FRAME_PORT);
	length = inw(ioaddr + RX_FRAME_PORT);

	/* 分配 sk_buff 缓冲区. */
	skb = alloc_skb(length, GFP_ATOMIC);
	if (skb == NULL) {
		printk("%s: Memory squeeze, dropping packet.\n", dev->name);
		lp->stats.rx_dropped++;
		return;
	}
	skb->len = length;
	skb->dev = dev;
	insw(ioaddr + RX_FRAME_PORT, skb->data, length >> 1);
	if (length & 1)
		skb->data[length-1] = inw(ioaddr + RX_FRAME_PORT);
	skb->protocol=eth_type_trans(skb,dev);

	netif_rx(skb);
	lp->stats.rx_packets++;
	return;
}
```

6.4.2.5. 网络设备驱动程序的编译

在一个驱动程序代码编制完成后，接下来的工作就是把它加进内核进行编译。以下我们将结合 CS8900A 的驱动程序，看看如何把一个驱动程序加进 uclinux 的代码中。

（1）首先在 uclinux/linux/arch/m68knommu 目录下 config.in 文件中网络设备字段里有如下变量定义：

```
bool 'Cirrus Logic Crystal LAN cs8900 ethernet' CONFIG_UCCS8900
if [ "$CONFIG_UCCS8900" != "n" ]; then
	bool' Hardware byte-swapping support for cs8900 ethernet'
		CONFIG_UCCS8900_HW_SWAP
	if [ "$CONFIG_ALMA_ANS" = "y" ]; then
```

```
            hex 'Base Address for cs8900 ethernet' CS8900_BASE 0x10200300
        else
            # ucsimm case
            hex 'Base Address for cs8900 ethernet' CS8900_BASE 0x10000300
        fi
fi
```

在这段程序里定义系统了 CONFIG_UCCS8900、 CONFIG_UCCS8900_HW_SWAP 以及 I/O 的映射地址。

(2) 在 uclinux/linux/drivers/net 目录下 space.c 文件中有如下程序段：

```
extern int cs89x0_probe(struct device *dev);
…………
static int ethif_probe(struct device *dev)
{
        …………
        #ifdef CONFIG_UCCS8900
              && cs89x0_probe(dev)
        #endif
        …………
}
```

这部分的功能是将 cs8900 的初始化函数连入 dev_base 指向的链表中。

(3) 在 uclinux/linux/drivers/net 目录下 makefile 文件中有如下代码段：

```
ifeq ($(CONFIG_UCCS8900),y)
    L_OBJS += uCcs8900.o
Endif
```

正是因为有上面三部分，在 make xconfig 出现图形选单中就可以选择'network device support'，'Cirrus Logic Crystal LAN cs8900 ethernet' 和'Hardware byte-swapping support for cs8900 ethernet'选单。这样在 make 时生成的 image.bin 中就加入了对 CS8900A 芯片的支持。

6.4.2.6. 网络驱动程序的测试

在带 CS8900A 芯片驱动的内核启动起来后就可以对它进行测试了。测试可以根据启动 init 进程运行到 cs89x0_probe()函数时显示的初始化信息判断网卡是否已经成功被探测到。如果一切正常，相关的信息如下：

```
…………
cs89x0:cs89x0_probe(0x0)
cs89x0: Setting up uCcs8900 Chip Select & IRQ ioaddr = 0x10000300
cs89x0.c: v2.4.3-pre1 Russell Nelson <nelson@crynwr.com>, Andrew Morton <andrewm@uow.edu.au>
eth0: cs8900 rev J found at 0x10000300
cs89x0 media RJ-45, IRQ 20, programmed I/O, MAC 01:00:3b:5c:01:00
cs89x0_probe1() successful
    …………
```

```
eth0: using half-duplex 10Base-T (RJ-45)
…………
```

内核起来之后运行 uclinux 支持的网络命令，比如 ping、ftp,http 等来进行测试。不过要运行这些命令之前必须打开网络设备，具体的步骤如下:

```
# ifconfig eth0 166.111.73.25
# route add –net 166.111.73.255 netmask 255.255.255.0 eth0
# ifconfig
eth0      Link encap:Ethernet    HWaddr 01:00:3B:5C:01:00
          inet addr:166.111.73.25    Bcast:166.111.73.255    Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST    MTU:1500    Metric:1
          RX packets:7094 errors:0 dropped:1818 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:20 Base address:0x300

lo        Link encap:Local Loopback
          inet addr:127.0.0.1    Mask:255.0.0.0
          UP LOOPBACK RUNNING    MTU:16436    Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

其中 ifconfig 命令给目标板网络接口配置 IP 地址，route 命令配置网关和子网掩码。经过以上步骤网络就配置好了，可以通过 ping 命令来测试:

```
# ping 166.111.73.1
PING 166.111.73.1 (166.111.73.1): 56 data bytes
64 bytes from 166.111.73.1: icmp_seq=0 ttl=30 time=10.0 ms
64 bytes from 166.111.73.1: icmp_seq=1 ttl=30 time=10.0 ms

--- 166.111.73.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 10.0/10.0/10.0 ms
```

附 录

A. 参考文献

[1] 臧春华、蒋璇，数字设计引论，高等教育出版社，2000
[2] 何小艇主编，电子系统设计，浙江大学出版社，2000
[3] 刘森、何希顺、何荣森，HPC 相关新技术的发展及其应用，电子产品世界，2001.4
[4] 刘森、何希顺、慕春棣，掌上电脑研制中的关键技术，电子技术应用，2000.12
[5] 何希顺，基于 StrongARM 的掌上处理机的设计和实现，中科院博士学位论文
[6] 李林功，全面理解 RISC 正确评价 CISC，电工教学，1997.6
[7] 韩纪庆，RISC 与 DSP 的结构比较及在嵌入式应用中的方案选择，微电子学与计算机，1994.5
[8] 尤一鸣、傅景义、王俊省，单片机总线扩展技术，北京航天航空大学出版社，1993
[9] 金西、黄汪，Linux 操作系统是嵌入式系统新的选择，微计算机信息
[10] Andrew S. Tanebaum、Albert S. Woodhull，操作系统：设计与实现，清华大学出版社
[11] David A.Rusling 著、朱珂等译，Linux 编程白皮书，机械工业出版社，2000
[12] 邹思轶等，嵌入式 Linux 设计与应用，清华大学出版社，2002
[13] 王学龙，嵌入式 linux 系统设计与应用，清华大学出版社，2001.8
[14] Alessandro Rubini、Jonathan Corbet，linux 设备驱动程序，中国电力出版社，2000.4
[15] 胡希明等，Linux 内核源代码分析，浙江大学出版社，2001
[16] 李善平等，linux 内核 2.4 版源代码分析大全，机械工业出版社，2002.1
[17] 陈莉君，深入分析 linux 内核源代码，人民邮电出版社，2002.8
[18] Kenneth C.Louden 著、冯博琴、冯岚等译，编译原理及实践，机械工业出版社，2000
[19] David Woodhouse，JFFS:The Journalling Flash File System，Red Hat Inc
[20] 史忠植，智能主体及其应用，科学出版社，2000.12
[21] Andrew S.Tanenbaum，Distributed Operating Systems，清华大学出版社，1997.2
[22] M.Bradshaw, An Introduction to Software Agents, In Software Agents, 1996
[23] David K.Lewis, Counterpar Theory and Quantified Modal Logic, The Journal of Philosophy, Vol.65, Issue 5(Mar. 7, 1968), 113-126
[24] 董军、潘云鹤，Agent 的混沌动力学行为，信息与控制，第 28 卷第 1 期，1999.2
[25] Kiss, G. and Reichgelt, H.，Towards a semantics of desires，In Werner, E. and Demazeau, Y., editors, Decentralized AI 3 - Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-91), pages 115-128. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands，1992
[26] Eduardo Alonso, How Individuals Negotiate Societies, In: CMAS98, 18-25
[27] William Stalling，Operating Systems: Internals and Design Principles 3rd Ed，清华大学出版社，1997
[28] Sun Microsystems Computer Corporation，The Java Virtual Macine Specification，1995

B. 参考网站

[1] http://e-www.motorola.com/：motorola 公司主页，可以查到其产品的技术资料。

[2] http://www-s.ti.com/：TI 公司主页，可以查到其产品的技术资料。

[3] http://www.cirrus.com/en/support/：Cirrus 公司产品的技术资料。

[4] http://www.cs.cmu.edu/~wearable/software/assabet.html：CMU 可穿戴计算计划中支持 Assabet 部分。他们针对 Assabet 提供了一套很不错的安装 Linux 的方案，同时还提供了 Assabet Linux 环境下 PCMCIA 的大部分实现。

[5] http://www.arm.linux.org.uk：ARM Linux 主页，拥有很多关于 ARM Linux 方面的信息。

[6] http://developer.intel.com/design/strong/quicklist/eval-plat/sa-1110.htm：Intel 的面向 Assabet 开发者的主页。

[7] http://www.handhelds.org/：该组织的目标是推动面向手持或可穿戴计算机的开放源码软件开发。现阶段，该网址主要针对 iPAQ 和 Linux，但他们计划逐渐改变这点。

[8] http://www.lart.tudelft.nl/：LART 计划主页。这是一个“开放硬件”启动计划，用于创作一个基于 SA1100 处理器的免费的主板设计。他们实现了一个启载器“blob”，可以让 LART 系统（现在是 Assabet）从闪存启动 Linux。

[9] http://www.combo.org/lex_yacc_page/：lex 和 yacc 的主页。

[10] http://www.linux.org/docs/ldp/howto/Lex-YACC-HOWTO-1.html：lex 和 yacc 的 how-to。

[11] http://www.objsw.com/CrossGCC/：有关 GNU 交叉编译器的常见问题解答。

[12] http://busybox.lineo.com/：BusyBox 是一个开放源码工具，用于将一个基本的 Linux 安装版本所占用的空间降到最低。很多现有的虚拟磁盘映像都用到这个工具。

[13] http://www.linux-mtd.infradead.org/：这个团体将其工作重心放在针对内存设备特别是闪存的一个通用的 Linux 子系统。对于版本号为 2.4.0-test5-rmk1-np1 的 Linux 内核，一个 Assabet 的 StrataFlash 不用 MTD 子系统。现在有很多计划将 Assabet 驱动移植到 MTD，但是还没有真正实现。

[14] http://www.uClinux.org：uCLinux 主页。

[15] http://www.hhcn.org/：华恒嵌入式 Linux 技术资料网站。

[16] http://www.linux.org/：Linux 主页。

[17] http://www.kernel.org/：Linux 内核主页。

[18] http://www.netwinder.org/：Netwinder 计划。提供在 Netwinder 上运行 Linux 的有关信息，其中很多对所有的 SA11XX 系统都有用。

[19] http://xcopilot.cuspy.com：提供一个可在 Assabet Linux 上运行的“xcopilot”（一个 PalmPilot 的模拟器）版本。

[20] http://sources.redhat.com/：提供很多开放源码项目，包括 cygmon，cygwin，gcc，binutils，和 glibc。

[21] http://www.redhat.com/docs/manuals/gnupro/：gnu project 的 manuals。

[22] http://www.viewml.com/：一个面向 Microwindows 的网络浏览器。

[23] http://www.microwindows.org/：一个小型的视窗系统。它向上层应用程序提供两套可选择的 API：一个面向 Windows 开发者，另外一个面向 X Windows 用户。

[24] http://www.pocketlinux.com/：一个完全遵从 GPL 的操作系统，它可以在任何可以运行 Linux 的计算设备上运行，其开发的主要目标平台是 Vtech Helio，不过最近开始转向康柏的 iPAQ。这个项目由 Transvirtual 支持。